Received April 25, 2020, accepted May 28, 2020, date of publication June 4, 2020, date of current version June 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3000009

A Novel FPGA Accelerator Design for Real-Time and Ultra-Low Power Deep Convolutional Neural Networks Compared With Titan X GPU

SHUAI LI[1], YUKUI LUO[2], (Student Member, IEEE), KUANGYUAN SUN[3], NANDAKISHOR YADAV[1], (Member, IEEE), AND KYUWON KEN CHOI[1], (Senior Member, IEEE)
[1]VLSI Design and Automation Laboratory, Illinois Institute of Technology, Chicago, IL 60616, USA
[2]Department of Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, IL 60608, USA
[3]Department of Computer Science, Rice University, Houston, TX 77005, USA

Corresponding author: Shuai Li (sli97@hawk.iit.edu)

This work was supported by the Industrial Core Technology Development Program of the Ministry of Trade, Industry and Energy (MOTIE)/Korea Electronics Technology Institute (KETI), South Korea (Development of Camera-Based Real-Time Artificial Intelligence System for Detecting Driving Environment and Recognizing Objects on Road Simultaneously), under Grant 10083639.

ABSTRACT Convolutional neural networks (CNNs) based deep learning algorithms require high data flow and computational intensity. For real-time industrial applications, they need to overcome challenges such as high data bandwidth requirement and power consumption on hardware platforms. In this work, we have analyzed in detail the data dependency in the CNN accelerator and propose specific pipelined operations and data organized manner to design a high throughput CNN accelerator on FPGA. Besides, we have optimized the kernel operations to obtain a high power efficiency. The proposed CNN accelerator supports image classification and real-time object detection with high accuracy. The evaluation results show that our CNN-based FPGA accelerator can achieve 740 Giga operations per second (GOPS) at 200 MHz with kernel power of 12.2 watts on Intel Arria 10 FPGA. For object detection tasks, our system can achieve 105 fps with 56.5 mAP or 25 fps with 73.6 mAP on VOC dataset. Since we use the mixed fixed-point data representation, the detection accuracy is comparable with the GPU-based YOLO V2 framework. The power efficiency of our system is $\sim 3.3\times$ better than Titan X GPU and $\sim 418\times$ better than Intel E5-2620 V4 CPU.

INDEX TERMS Deep neural network accelerator, FPGA, pipeline architecture, parallel computing, mixed fixed-point, object detection, low power.

I. INTRODUCTION

Convolutional neural networks (CNNs) based deep learning techniques are applied in wide fields such as traffic tracking, speech recognition, medical diagnosis, etc. However, The computational complexity of CNN is a heavy burden and hard to be implemented on devices with limited computational resources. The standard convolution layers are implemented by using the multiply-accumulate (MAC) operations. As shown in FIGURE 1, there is 92% [1] of execution time spent on MAC during forward inference in YOLO V2 framework [2].

To cope with such heavy burden computation, most of the deep learning applications are based on the graphics processing unit (GPU) since GPU has large scale single instruction multiple data (SIMD) fabric in a chip and high bandwidth memory and local memory. We call the cross-vendor architecture of the central processing unit (CPU) and

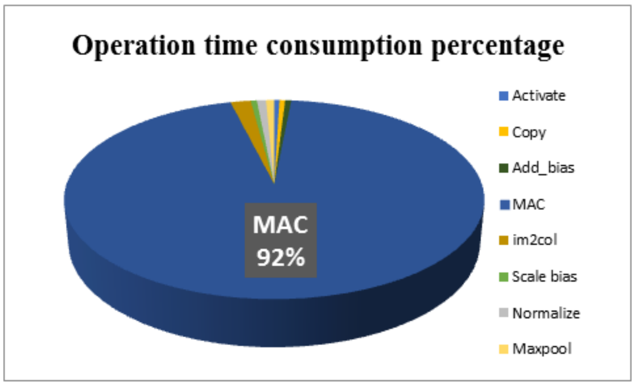


FIGURE 1. Forward inference time consumption in YOLO V2 framework.

The associate editor coordinating the review of this manuscript and approving it for publication was Xujie Li.

other specifications (e.g., GPU, FPGA, DSP) heterogeneous systems. For execution across heterogeneous systems, the general way is to use open computing language (OpenCL) [3] as a bridge to connect between different specifications. Some GPU vendors such as NVIDIA provide specific GPU programming language CUDA [4] and deep learning neural network (DNN) library cuDNN [5] which have a better performance than OpenCL but can only be applied on NVIDIA GPUs. Even though high-end modern GPU has a high computational throughput on DNN performance, it also costs high power dissipation, and collaborating with the workstation is too heavy to carry, which reduces the practicality in the industrial field. FPGA is one of the solutions to replace GPU, which has large processing elements, low power consumption, reconfigurable, and portable characteristics. However, there are some drawbacks to FPGA. Different from GPU, the memory on FPGA does not have a cache structure, and the bandwidth is two generations older than high-end GPU (e.g., the DRAM on FPGA Arria 10 is DDR3, but the memory on GPU Titan X is GDDR5). The design challenges such as data dependency and memory bandwidth make it hard to work in real-time as fast as high-end GPU, especially on low power devices. Furthermore, the limited size of on-chip memory is a bottleneck on FPGA. Data movements between on-chip memory and off-chip (e.g., DRAM) will introduce large power consumption and delay. Our motivation and aim are going to build a high throughput and low power CNN-based FPGA computation system.

This work has the following contributions:

1) We analyze in detail the data dependency in the CNN accelerator and present a high throughput CNN-based FPGA accelerator. Specifically, we use a pipelined MAC operation structure to remove loop-carried data dependency. We also propose the zigzag fetch unit to remove line data dependency.
2) To achieve a high power efficiency, we propose the offline pre-processing and combination of batch normalization (BN) and scale/bias (SB) and approximation expression for kernel computation.
3) We have applied the CNN accelerator on advanced multi-object detection frameworks such as tiny YOLO V2 and full YOLO V2 [2]. To acquire a high accuracy, we use 8-16 bits mixed fixed-point data representation in the object detection task and achieve comparable accuracy compared with Titan X GPU. The demo can be found in [39].
4) Our CNN accelerator provides a definable interface to reconfigure the new CNN model easily, and it supports the Caffe framework [6].

The rest of the paper is organized as follows. In Section II, we introduce the background, which includes previous related work, FPGA implementation methodologies, and preliminary analysis of CNN. In Section III, we present the parallel and pipeline modeling of CNN-based accelerator, and analyze the data dependency in pipeline structure. The high-speed pipeline structure design process is presented in detail in Section IV. Section V describes the bandwidth optimization for power-efficient CNN accelerator. Section VI introduces OpenCL based FPGA implementation and our system. The performance and experimental results are presented in Section VII. Finally, Section VIII concludes the paper.

II. BACKGROUND

A. RELATED WORK

With the success of CNN in the field of computer vision, more and more researchers focus on deploying the CNN implementation on different computational architectures. While considering the power efficiency of the implementation architectures, CNN-based FPGA accelerators have achieved a further step in recent years [7]–[29]. These designs can mainly be divided into three categories. The first category copes with **the optimization in the computational kernels** [10], [13], [14]. Zeng *et al.* [10] used optimized frequency domain (fast Fourier transform or FFT) convolution instead of standard convolution and got the throughput of 669 Giga operations per second (GOPS) on VGG16 net. FFT based convolution is fast for large filters, but state-of-the-art CNN uses small filter sizes such as 3×3. Cong and Xiao [13] proposed an algorithmic modification to reduce the general matrix multiplication (GEMM) computational workload by 22% on average. However, this work only focused on the GEMM part and ignored the bandwidth design on external memory. Some works [14], [28] have applied Winograd convolution to replace the traditional convolution operation. Aydonat *et al.* [14] used Winograd transformation to boost the performance on FPGA, which could achieve peak performance of 1.3 trillion floating-point operations per second (TFLOPS) in fully connected (FC) layer. But in modern DNN applications especially for object detection, the FC layer has a low working efficiency which might cause overfitting. Moreover, this work [14] only evaluated the most basic CNN structure, AlexNet [30], and the scalability may not satisfy the current deeper neural network. The second category is to deal with **bandwidth optimization** to improve throughput [15]–[21]. Suda *et al.* [15] used quantized 16-bit fixed-point operation to improve the throughput. However, the throughput was only 117.8 GOPS which was still far less than the real-time requirement. Li *et al.* [16] applied parallel operation on convolution layers and a batch-based computing method on FC layers to process multiple input images in parallel. However, this method is not suitable for video sequences application which has a temporality input order. Motamedi *et al.* [17] provided a way to develop a method to obtain a feasible weight file for FPGA, and Nurvitadhi *et al.* [18] used special I/O between CPU and FPGA to accelerate the detection process. Farabet *et al.* [19] placed some data processing tasks on DSP to speed up the processing. Some works [17], [20] applied roofline model to design the trade-off between computing throughput and required memory bandwidth to maximize the utilization of FPGA resources, but the performances were only 61.62 GFLOPS in the work of Zhang *et al.* [20] and 84.2 GFLOPS in the work of Motamedi *et al.* [17], respectively, which were far less than real-time application. Gokhale *et al.* [21] proposed a CNN accelerator that could achieve a peak performance of 200 GOPS. The architecture only included three modules: convolution, sub-sampling, and non-linear functions. Guo *et al.* [29] proposed a CNN design with a data quantization strategy and compilation tool which could get 137 GOPS throughput on Zynq XC7Z045 FPGA. Geng *et al.* [8] proposed a quantitative model for mapping CNNs on multi-FPGAs to improve the throughput. However, the power consumption will increase greatly by using an FPGA cluster. Guan *et al.* [27] proposed an end-to-end framework that generated the hardware implementation with RTL-HLS hybrid templates. Ma *et al.* [7] analyzed and optimized the memory access of the CNN accelerator based on multiple design variables to achieve high throughput. The third category is **the model optimization** [9], [22], [23]. Recently, Fujii *et al.* [22] used the pruning technique on the FC layer to reduce 39.8% of parameters in FC layers. Even though the technique does increase the speed of FC layers somehow, it does not improve the throughput. In fact, due to the low efficiency of the FC layer, it is abandoned in most advanced modern CNN models. Hailesellasie *et al.* [23] implemented a reduced parameter CNN on Xilinx ZYNQ FPGA, but it still existed a large gap between structure optimization and cross-platform transplantable. Iandola *et al.* [31] proposed SqueezeNet to reduce the number of parameters by using 1 × 1 kernels and concatenated structure. However, less parameters do not always guarantee a fast speed. For example, standard convolution layers have fewer parameters compared with FC layers, but the processing speed of FC layers is much faster than convolution layers. The convolution time depends on the computation method, e.g., the depthwise convolution [32] is faster than standard convolution. Bai *et al.* [9] adopted depthwise separable convolution to replace the standard convolution for embedded platforms, which could reduce the operations and parameters.

B. FPGA IMPLEMENTATION METHODOLOGIES

The CNN-based FPGA methodologies can be divided into two categories. The first category is **high-level synthesis (HLS) methodology**, which refers to C/System C-like synthesis, including the OpenCL method (for Intel/Xilinx FPGA both) or Vivado HLS (for Xilinx FPGA only), and most of the CNN FPGA designs [14], [15], [24] use HLS. This method can accelerate the design and simulate easily, and automatically generate Verilog/VHDL code. Xilinx ZYNQ dramatically changes the development step for FPGA, as each FPGA connected with a computing processing unit (ARM). The communication between the processing system (PS) and FPGA logic is very convenient. For Xilinx Vivado HLS, it supplies some advanced HLS tools such as HLS DNN intellectual property (IPs), HLS linear algebra, and HLS DSPs, which makes the design of CNN FPGA accelerator much easier. Thus more and more designs are based on Xilinx FPGA. On the contrary, the HLS implementation on Intel FPGA is still a challenge. The second category is traditional **register-transfer level (RTL) design method**, e.g., [25], [26]. This method will cost a longer development period but can make good use of the FPGA resources without any redundancy, and have a better sequential logic control in cycle-wise level. In our work, since our platform is Intel FPGA, we use OpenCL based methodology combined with register-transfer level (RTL) math intellectual property (IP) to design our CNN FPGA accelerator.

C. YOLO V2 FRAMEWORK FOR OBJECT DETECTION

Object detection includes two tasks: classification and regression. Classification gives the class which the object belongs to. Regression tells the information for each object's coordinate in an image.

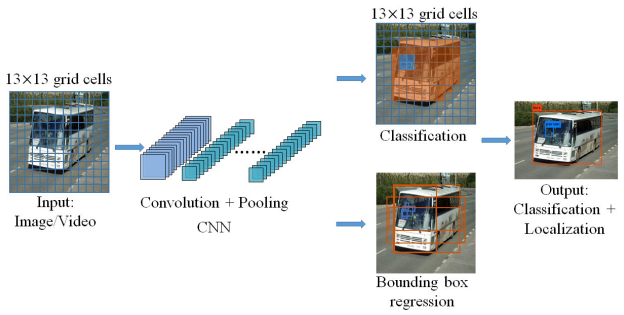


FIGURE 2. YOLO V2 framework for object detection.

We have applied the CNN-based FPGA accelerator on the YOLO V2 framework [2] for object detection. There are two models for YOLO V2: full YOLO V2 and tiny YOLO V2. Full YOLO V2 network consists of 23 convolution layers with a high detection rate, whereas tiny YOLO V2 is a simplified network that consists of 9 convolution layers with a relatively low detection rate but very high speed. As shown in FIGURE 2, YOLO V2 consists of feature extraction part in the first several layers and classification and localization part in the last several layers. Compared with YOLO V1, V2 resizes the input resolution to 416 × 416 because it wants an odd number of locations in the feature map, so there is a single-center cell. YOLO V2 down-sampled the input size by 32 times, and the size of the final OFM is 13 × 13 (since $416/32 = 13$). YOLO V2 abandons the FC layer and use anchor boxes to predict bounding boxes (BB). To obtain better anchor boxes, YOLO V2 uses k-means dimension clusters to train the BB.

The feature extraction part extracts $W \times H \times (B \times 5 + C)$ features, where W and H are the width and height of output feature map (OFM) in the last layer, respectively. In other words, the image is divided into $W \times H$ cells, and for each cell, the feature extractor extracts $(B \times 5 + C)$ features. B is the number of BB, C is the number of classes. There are five parameters for each BB: $\{t_x, t_y, t_w, t_h, t_o\}$, each BB represented by $\{b_x, b_y, b_w, b_h, b_o\}$ can be calculated as [2]:

$$\begin{aligned} b_x &= \sigma(t_x) + c_x \\ b_y &= \sigma(t_y) + c_y \\ b_w &= p_w e^{t_w} \\ b_h &= p_h e^{t_h} \\ b_o &= \sigma(t_o) \end{aligned} \tag{1}$$

where$\{c_x, c_y\}$ is the top left corner of the cell, and $\{\frac{b_x}{W}, \frac{b_y}{H}\}$ is the center of BB relative to the whole image. $\sigma(\bullet)$ denotes sigmoid function.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2}$$

$\{b_w, h_h\}$ is the size of BB relative to bounding box prior $\{p_w, p_h\}$. b_o can be expressed as $Pr(object) \times IOU(b, object)$. Where $Pr(object)$ is the confidence of the object, and $IOU(b, object)$ is the intersection over union (IOU) between predicted BB and ground truth BB.

The loss function can be described as:

$$\begin{aligned} loss_t &= \lambda_{noobj} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} \sum_{k=0}^{B-1} \mathbb{I}_{ijk}^{noobj} b_{oijk}^2 \\ &\quad + \lambda_{obj} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} \sum_{k=0}^{B-1} IOU(BB_{pred}, BB_{gt}) \mathbb{I}_{ijk}^{obj} (1 - b_{oijk})^2 \\ &\quad + \lambda_{class} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} \mathbb{I}_{ij}^{obj} \sum_{c \in classes} MSE(C) \\ &\quad + \lambda_{coord} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} \sum_{k=0}^{B-1} (2 - w_{gt} h_{gt}) \mathbb{I}_{ijk}^{obj} ((b_x - b'_x)^2 \\ &\quad + (b_y - b'_y)^2 + (t_w - t'_w)^2 + (t_h - t'_h)^2) \end{aligned} \tag{3}$$

There are four terms calculated in the loss function: no-object, object, class, and coordinates loss. $\mathbb{I}_{ijk}^{noobj}$ denotes that the kth BB in location (i, j) will be penalized if the IOU is lower than a threshold value T. $\mathbb{I}_{ijk}^{obj}$ denotes that the kth BB in location (i, j) is responsible for that prediction. $\mathbb{I}_{ij}^{obj}$ denotes if the object is in location (i, j). $IOU(BB_{pred}, BB_{gt})$ is the IOU between predicted BB and ground truth BB. Mean squared error (MSE) loss is used for classification in location (i, j). The last term uses the sum of square error (SSE) to calculate the coordination loss with the location information. $\{w_{gt}, h_{gt}\}$ is the ground truth value of the BB relative to the whole image. $\{\frac{b'_x}{W}, \frac{b'_y}{H}\}$ is the ground truth of the center coordinate for the BB. $\{t'_w, t'_h\}$ is the natural logarithm of the ground truth size relative to the bounding box prior. In YOLO V2, we use the default parameter $\lambda_{noobj} = 1, \lambda_{obj} = 5$, $\lambda_{class} = 1, \lambda_{coord} = 1$, and the threshold $T = 0.6$.

D. PRELIMINARY ANALYSIS OF CNN

There are some restrictions on the challenge for designing the architecture of CNN-based FPGA accelerator:

1) Fetch data latency from global memory (DDR3) to FPGA on-chip memory is a bottleneck in the design.
2) Hardware resources on FPGA are limited.
3) Data dependency.

The first limitation is communication between FPGA and CPU. The bandwidth of FPGA is two generations older than the high-end GPU. The data transfer latency between GPU and CPU is orders of magnitude less compared with FPGA, which means directly translate the CNN software from GPU to FPGA without considering the data transfer speed is unacceptable. We need to design a methodology to overcome this drawback in FPGA.

The second limitation comes from the hardware limitation during the design of parallel and pipeline structure. It needs to make a trade-off between the pipeline throughput and hardware cost. Most of the FPGA devices still have a limited on-chip memory size. The highly parallel structure will increase the usage of on-chip memory significantly.

The third limitation is data dependency. Different data dependency types exist inter and intra layers, which affect the throughput heavily. It requires a well-scheduling process to reduce the effect of data dependency.

We need to deal with these restrictions to design the CNN accelerator.

For explanation convenience, we assume each input feature map (IFM) and kernel are in square shapes (as shown in FIGURE 3, the width/height of the IFM is D_F, and the width/height of the kernel is D_K). M IFM combining generate a 3D cube with size of $D_F \times D_F \times M$. N filters processing each input 3D cube can be abstracted as a 4D hypercube $D_K \times D_K \times M \times N$, and the first 3D cube $D_K \times D_K \times M$ is corresponding to input 3D feature maps $D_F \times D_F \times M$. These two 3D cubes result in a 2D output feature map (OFM) $D_g \times D_g$, where D_g is the width/height of the OFM. The 4th dimension N means there are N groups of such $D_K \times D_K \times M$ 3D filters, which results in 3D OFM $D_g \times D_g \times N$.

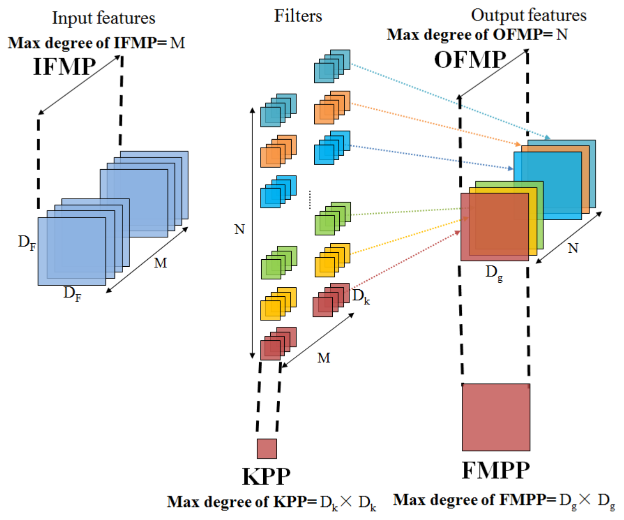


FIGURE 3. Feedforward of convolution and parallelism.

The basic idea of achieving high computation throughput is utilizing parallelism and pipeline mechanism. There are five concurrent mechanisms exist in MAC operations.

1) Input feature maps parallelism (IFMP): There are M IFMs in each layer, and each feature map is independent of the others. We can fetch and compute the feature maps in MAC operation parallel in the M^{th} dimension.
2) Output feature maps parallelism (OFMP): There are N groups of 3D filters in each layer, and each filter group is independent of the others and corresponding to N independent OFMs. We can fetch and compute the N filters parallel in MAC operation in the N^{th} dimension.
3) Feature map plane-parallel (FMPP): For each independent OFM plane, the $D_g \times D_g$ output pixels can be computed concurrently.
4) Kernel plane parallelism (KPP): The 2D convolution requires $D_K \times D_K$ times MAC operations. These MAC operations can be implemented concurrently.
5) Layer parallelism (LP): The data dependency exists between two consecutive layers. However, the next layer can start in a pipeline manner before the previous layer complete.

The ideal case is fully exploiting all the concurrent mechanisms. However, it is impossible to unroll all the five mechanisms simultaneously because of the hardware resources limitation on FPGA. Usually, the on-chip memory on FPGA is not enough to store the weights, IFMs, and intermediate OFMs. Moreover, the number of LUTs and registers cannot afford fully exploited loop unroll instances. To address this problem, it needs to design the parallel structure using the limited hardware and reuse the parallel structure in a pipeline manner. For KPP, the kernel size D_K determines the parallelism. This parameter may vary in different layers (e.g., $1\times1, 3\times3, 5\times5, 11\times11$, etc). To design a general CNN accelerator with an uncertain kernel size, we use a pipeline scheme instead of the parallel structure in the kernel plane. For FMPP, the parallelism is given by $D_g \times D_g$. Usually, it is decided by the minimum output feature size since we hope the accelerator can be applied to all the convolution layers (e.g., the minimum output feature size is 13×13 in AlexNet or YOLO V2, whereas in VGG16 or YOLO V1 it is 7×7). The maximum degree of IFMP and OFMP is M and N, respectively. Due to the limitation of resources, it is impossible to use M and N for parallelism. Usually, we use a value that achieves the maximum throughput but not excess hardware resources.

E. GENERAL CNN COMPUTATION PROCESS

To get the final OFM, the CNN accelerator has to perform different operations such as fetch data/weights unit (FU), MAC, batch normalization (BN), scale/bias operation (SB), activation (ACT), pooling (PL), and write data back (WB).

As shown in FIGURE 4, there are seven stages in each layer. The solid blocks are mandatory modules which include FU and MAC operation, and the dashed blocks are optional operations such as BN, SB, ACT, PL, and WB. The configuration decides whether each module works or not in the host program.

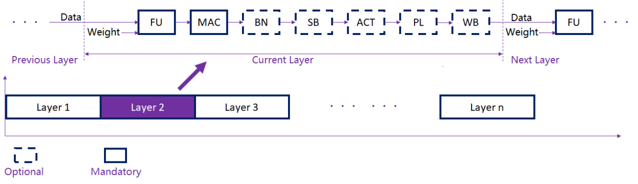


FIGURE 4. Overview of the CNN accelerator computation process and each module.

1) FU: It fetches the weights and feature maps by using a sliding window and arrange them in a convolution pair to prepare for MAC operation.
2) MAC: The traditional input weights can be viewed as a 4D hypercube, the 1st and 2nd dimensions are horizontal and vertical directions in the same weight plane. We can apply the parallel mechanism IFMP and OFMP in the 3rd and 4th dimensions on FPGA.
3) BN: This module will receive the convolution results and do batch normalization operation.
4) SB: This module will receive the result after BN, multiply it with the parameter scale, add it with bias, and then send the data to the next module.
5) ACT: This module will receive the result after SB and do the activation operation. In this accelerator, we use leaky rectified linear unit (LReLU) function.

$$f(x) = \begin{cases} x & if\ x \geq 0 \\ 0.1x & otherwise \end{cases} \tag{4}$$

6) PL: Which is used to down-sampling the input data. The popular idea uses line buffers to store and process the input data line by line. A switch is used to control this module works or not on FPGA.
7) WB: Write back the output data to external memory.

III. PARALLEL AND PIPELINE MODELING OF CNN-BASED FPGA ACCELERATOR

A. PROPOSED HIGH-SPEED PARALLEL DESIGN FOR 3D CONVOLUTION OPERATION

As shown in FIGURE 5, the MAC operation convolves the IFMs with filters and generates the OFMs. 3D convolution can be expressed as:

$$\begin{aligned} &\forall \{f_o, y, x\} \in [1, N] \times [1, D_g] \times [1, D_g] \\ &D_o(f_o, y, x) = \sum_{f_i=0}^{M/vector\_depth} \sum_{k_y=0}^{D_k} \sum_{k_y=0}^{D_k} W_l(f_o, f_i, k_y, k_x) \\ &\qquad \cdot D_i(f_i, y+k_y, x+k_x) \end{aligned} \tag{5}$$

where $D_o(f_o, y, x)$ and $D_i(f_i, y, x)$ are neurons at (x, y) in OFM and IFM, respectively. $W_l(f_o, f_i, k_y, k_x)$ denotes the weights in l^{th} layer. The kernel size is $D_k \times D_k$. f_o is the output feature value, and f_i is the intput feature value.

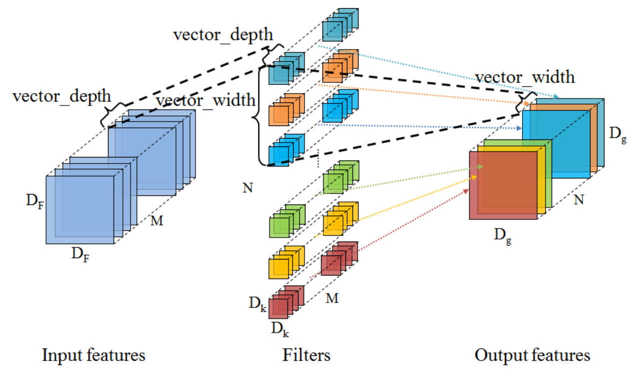


FIGURE 5. The vectorized 3D convolution operation.

```
L1: for(n = 0; n < N; n++){ // output feature map loop
L2:    for(m = 0; m < M; m++){ // input feature map loop
L3:       for(h = 0; h < H; h++ ){ // row in feature map loop
L4:          for(w = 0; w < W; w++){ // column in feature map loop
L5:             for (k1 = 0; k1 < K; k1++){ // row in kernel loop
L6:                for(k2 = 0; k2 < K; k2++){ // column in kernel loop
                sum[n][h][w]+=data[m][h+k1][w+k2]*weight[n][m][k1][k2];
                   } // end of column in kernel loop
                } // end of row in kernel loop
             } // end of column in feature map loop
          } // end of row in feature map loop
       } // end of input feature map loop
    } // end of output feature map loop
```

FIGURE 6. Pseudocode for MAC operation.

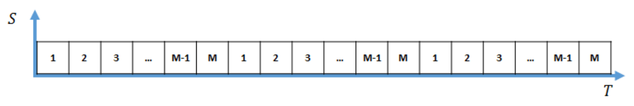
FIGURE 6 illustrates the pseudocode for basic MAC operation. There are six loops totally, the time complexity of convolution operation is:

$$C_{conv}^{org} = O(D_g \cdot D_g \cdot M \cdot N \cdot D_k \cdot D_k) \quad (6)$$

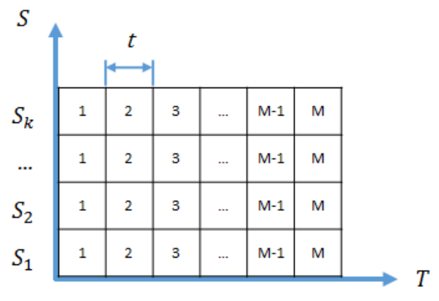
Date vectorize factor **vector_depth** is applied in M IFMs. Thus, the total convolution operation can be reduced **vector_depth** times. The vectorize factor **vector_width** is applied in N OFMs to generate multiple instances of different kernels. The overall convolution operation can speedup by **vector_depth**×**vector_width** times.

$$C_{conv}^{new} = O(D_g \cdot D_g \cdot \frac{M}{vector\_depth} \cdot \frac{N}{vector\_width} \cdot D_k \cdot D_k) \quad (7)$$

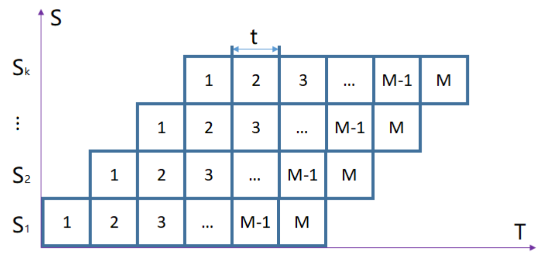
To finish all the MAC operations in 1 cycle, a $D_k \times D_k$ (e.g., 3×3) weight matrix with three IFMs requires 27 (e.g., $3 \times 3 \times 3$) multipliers and 27 adders. Implementing this in parallel will cost a large amount of digital signal processor (DSP) blocks and Look-up tables (LUTs), which is unacceptable for limited hardware resources to fully exploit the parallelism. There is a trade-off between processing capacity and hardware resources or design area. The high parallelization will exhaust a large number of computing units. The fetch unit has to load large data in parallel and store on-chip memory, which makes the place and route congestion. To solve this problem, we applied pipeline architecture to achieve higher throughput.



(a) Sequential operation of M assignments



(b) Unrolled loop with M assignments and k iterations



(c) An ideal pipeline structure with k stages and M assignments

FIGURE 7. General parallel structure vs. pipeline structure.

B. PARALLEL AND PIPELINE MODELING OF CNN ACCELERATOR

As shown in FIGURE 7a, which is the basic loop with M assignments and k iterations If each assignment of this loop takes one clock cycle, the whole loop has a latency of $M \times k$ cycles. This structure uses the least resources but costs the longest execution time. FIGURE 7b depicts the fully unrolled loop with M assignments and k iterations. The total loop has a latency of M cycles. However, it replicates each assignment k times in hardware. This structure has the fastest speed but costs the largest resources which are usually unbearable in the design.

FIGURE 7c is an ideal pipeline structure with k stages. The pipelined loop can work concurrently but there is no area trade-off. As shown in FIGURE 7c, the throughput can be described as:

$$Throughput_k = \frac{M}{T_{total}} = \frac{M}{\sum_{i=1}^{k+M-1} t_{max}(i)} \quad (8)$$

where k is the number of stages, M is the total number of assignments, T_{total} is the total time for all the assignments. $t_{max}(i)$ is the maximum processing time in the i^{th} assignment. However, the CNN accelerator cannot achieve the ideal pipeline structure due to data dependency. In the next section, we will analyze the data dependency and optimize our design.

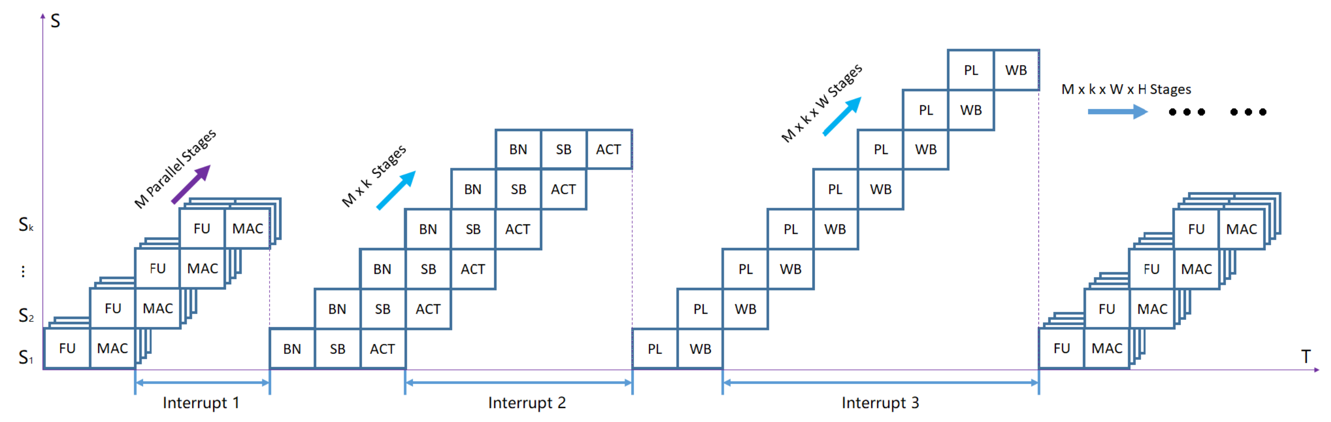


FIGURE 8. K-stage pipeline diagram for CNN accelerator.

C. DATA DEPENDENCY IN PIPELINE STRUCTURE

Data dependency refers to a situation that a variable relies on the previous status. There are three types of data dependency exists in the pipeline design: loop-carried data dependency, line data dependency, and inter-layer data dependency.

The launch frequency of a new loop iteration is called the initial interval (II). II indicates the number of clock cycles for which the pipeline has to wait before it can process the next loop iteration. In the pipeline structure, we expect that the best II value should be as small as possible. An $II = 1$ means that each clock cycle can process one new loop iteration. However, some complicated operations in the loop cannot finish in one clock cycle which introduces a large II. We can estimate the total latency of a loop by using (9).

$$latency_{\phi} = (N - 1) \times II + latency_{\varphi} \tag{9}$$

where latency$_{\phi}$ is the overall cycle to execute the loop. N is the number of iterations, and latency$_{\varphi}$ is the latency of a single loop iteration.

FIGURE 8 shows a k-stage pipeline diagram for the CNN accelerator. We assume that there are M compute units to fetch data/weight in parallel. After the MAC operation, there is an interruption before BN because BN has to wait for k^2 MAC operations to get one convolution ready, where k is the kernel size. Between ACT and PL, there is the 2nd interruption since the pooling operation requires line data ready (assume the length of a line is W). The 3rd interruption occurs between the WB and the next layer's FU because, in traditional CNN operation, FU will wait for the entire OFMs ready before the next layer's operation. In the next section, we will design the CNN accelerator to deal with these three interruptions.

IV. OPTIMIZATION OF PIPELINE STRUCTURE DESIGN FOR HIGH SPEED CNN-BASED FPGA ACCELERATOR

A. LOOP-CARRIED DATA DEPENDENCY

In FIGURE 8, the pipeline will be interrupted for $k \times k$ (k is the kernel size) MAC operations to wait for one convolution output to finish.

The loop-carried data dependency exists because the outer loop iteration (L5 in FIGURE 6) requires the multiplication from the previous iteration to finish before the inner loop iteration (L6 in FIGURE 6) can start. For example, it requires four cycles ($II = 4$) to finish multiplication and accumulation operation which means each inner iteration has a delay of 4 cycles.

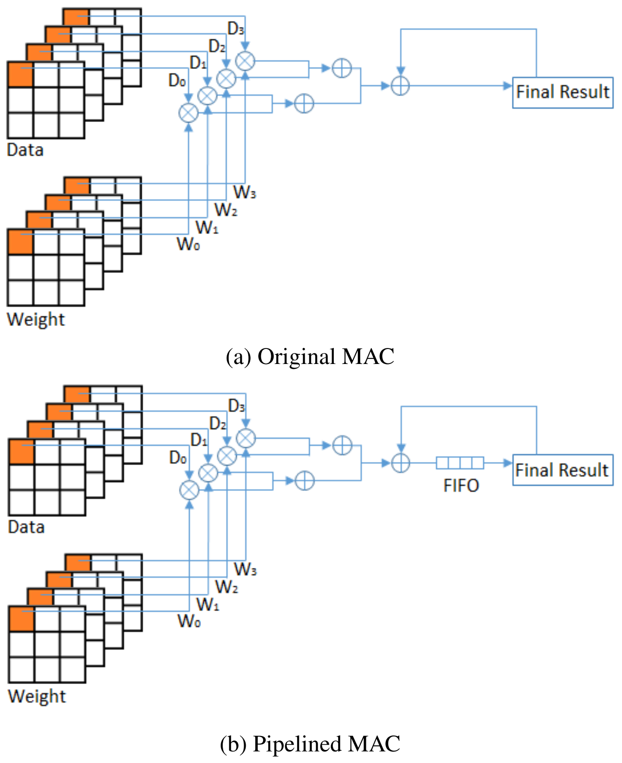


FIGURE 9. Original MAC operation vs. FIFO MAC operation.

FIGURE 9a is the traditional MAC tree operation. Without optimization, it has to wait for $[(N - 1) \times 4 + latency_{\varphi}]$ cycles to get a result. We can remove loop-carried data dependency by increasing the dependence distance as shown in FIGURE 9b. In this new MAC structure, (1) Declare multiple empty buffers (arranged in FIFO structure). (2) Initialize

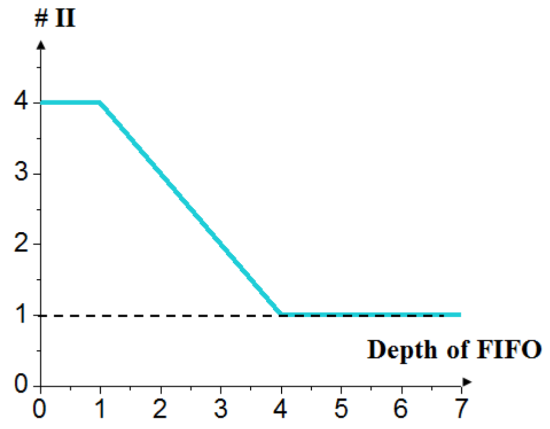


FIGURE 10. II vs. depth of FIFO.

all the buffer values as zero. (3) Use the last buffer (left-most in FIFO) in the array to store the multiplication value. (4) Perform the shift operation by using shift registers to pass the value in the last buffer into previous buffers in each clock cycle. (5) Add the value in the first buffer and multiplication result in the next clock cycle. (6) Write the final value to the result.

The proposed FIFO MAC structure can remove loop-carried data dependency because it is using buffers to store the temporary accumulate values to avoid write after write (WAW) problem and alleviate pipeline stalls. We find that the II can be reduced to one by using four buffers. As shown in FIGURE 10, with more than four buffers, there is no improvement but wasting hardware resources.

By using (9), we can estimate the latency of MAC operation.

$$MAC_{latency\_ratio} = \frac{MAC_{optimized\_latency}}{MAC_{original\_latency}} = \frac{(k^2-1)\times II_{new}+\varphi_{new}}{(k^2-1)\times II_{original}+\varphi_{original}} \quad (10)$$

where II_{new} is one and $II_{original}$ is four. Since we use a FIFO with a depth of four, φ_{new} is four, and $\varphi_{original}$ is one.

According to (10), the original MAC operation will take $(4k^2-3)$ cycles, whereas the FIFO MAC operation only takes (k^2+3) clock cycles. Since kernel size k is larger than 3, the new pipeline MAC operation takes less than 0.36 times clock cycles compared with the original MAC structure, which means the loop-carried data dependency can be reduced below 36% compared with the original design. In the next section, we will optimize the line data dependency.

B. LINE DATA DEPENDENCY

In tradition, it uses a $K \times K$ sliding window from left-up to right-bottom progressive scan line-by-line (as shown in FIGURE 12) to fetch input feature data in each clock cycle and prepare for convolution operation. Since the input data are line-by-line scanned, the order of computation thereafter is also followed line-by-line. This data fetch manner will

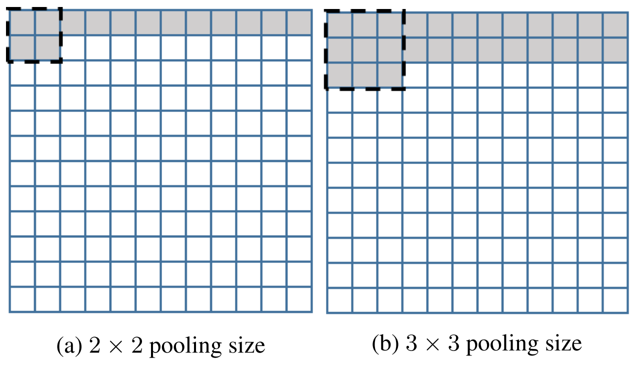


FIGURE 11. Line data dependency in pooling operation.

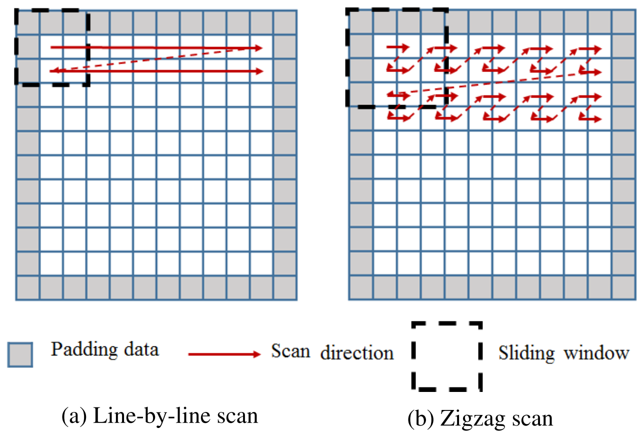


FIGURE 12. Scan manner in the fetch unit.

introduce line data dependency during the assignment of pooling. Equation (11) describes the pooling operation:

$$\forall\{f_o, y, x\} \in [1, N] \times [1, D_g] \times [1, D_g]$$
$$D_o^{pool}(f_o, y, x) = max_{p,q\in[1:P]}(D_o(f_o, y+p, x+q)) \quad (11)$$

The OFM is sub-sampled to $\frac{1}{P}$ of the original feature map size, where P is the pooling size.

As shown in FIGURE 11, to do a $P \times P$ pooling operation, the first $(P-1)$ line OFMs have to be ready and stored in line buffers, which will introduce line data dependency and on-chip memory consumption.

$$latency_{pool\_org} = [(P-1)\times W + P] \times latency_{\phi} \quad (12)$$

Equation (12) describes the line data latency of pooling operation by using traditional linear progressive FU. Where $latency_{\phi}$ is the MAC latency, and W is the length of line buffers ($W \gg P$). We have to wait for $latency_{pool\_org}$ cycles to do the first pooling operation.

To solve this problem, we propose the zigzag scan manner in FU. As shown in FIGURE 12b, we use a $(K+P-1) \times (K+P-1)$ sliding window to fetch the data from the global memory to FPGA local memory with a step size of P. Inside the sliding window, we use a zigzag order to rearrange the input data to P^2 $K \times K$ sub-blocks.

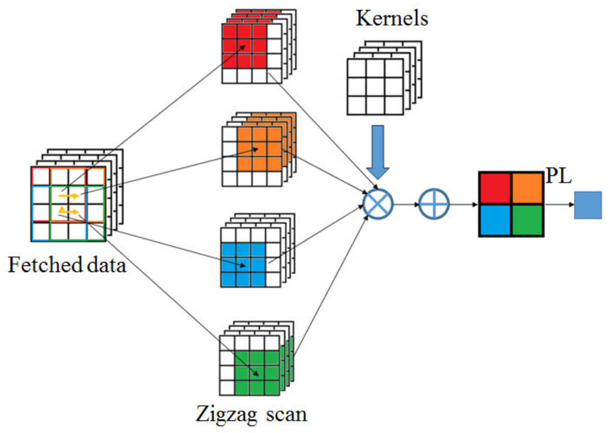


FIGURE 13. **Zigzag fetch unit (as an example of P=2).**

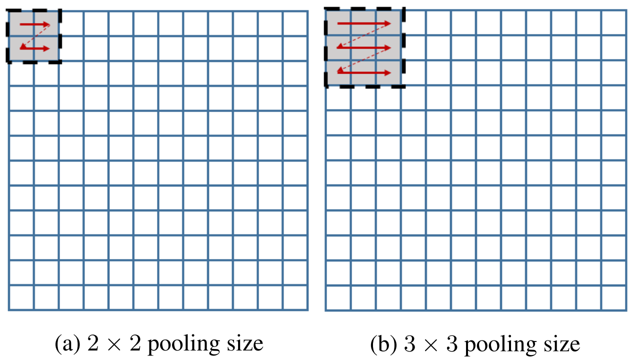


FIGURE 14. **Data dependency in pooling operation after applied ZFU.**

FIGURE 13 depicts the details of the zigzag fetch unit (ZFU). In this example, we assume the kernel size is 3 × 3, stride step size is 1, and the pooling size is 2. We use a 4 × 4 sliding window to fetch the input data. In each sliding window, we use a zigzag scan to fetch four 3 × 3 blocks and do convolution operation. For the pooling operation, we use a comparator to get the maximum value among the four input MAC results. ZFU not only reduces the line data dependency during pooling but also reduces the power consumption since this fetch manner utilizes the strategy of data reuse. One sliding window can fetch p^2 blocks. The reduction of reading external memory can reduce power dissipation on global memory.

$$latency_{pool\_zigzag} = P^2 \times latency_{\phi} \tag{13}$$

After applied ZFU technique, the data dependency of pooling is optimized to ($P^2 \times latency_{\phi}$) cycles (as shown in FIGURE 14), which is less than the original $latency_{pool\_org}$.

C. INTER-LAYER DATA DEPENDENCY

In FIGURE 8, interruption 3 refers to inter-layer data dependency. Normally there are dozens of layers in modern advanced large CNN models. To process CNN in the next layer, it usually has to wait until all the current layer's feature maps have been finished. The data dependency between WB and the next layer's FU is $W_l \times H_l \times latency_{pool}$. W_l and H_l are the width and height of OFM in each layer, respectively. It will introduce a large delay after dozens of such layers, and this is one of the reasons the CNN-based FPGA accelerator is hard to run in real-time. In fact, there is no need to wait for the entire OFMs to be completed. The inter-layer data dependency can be reduced to $(K_{l+1}+P_{l+1}-1) \times W_l \times latency_{pool}$, where $(K_{l+1}+P_{l+1}-1)$ is the length of sliding window in the next layer. K_{l+1} is the kernel size in the next layer, and P_{l+1} is the pooling size in the next layer. The overall inter-layer dependency is far less than $W_l \times H_l \times latency_{pool}$.

V. BANDWIDTH OPTIMIZATION FOR POWER EFFICIENT CNN-BASED FPGA ACCELERATOR

A. MIXED FIXED-POINT DATA REPRESENTATION

FPGA supports a substantial amount of logic for implementing floating-point operations. The original data type is a single-precision floating-point (32-bit) CNN forward computation, which costs plenty of hardware resources to implement MAC operation. The fixed-point representation of the data can save the number of hardware resources. The purpose of fixed-point data type is not only to reduce the hardware resources such as on-chip memory and DSPs but also to increase the data transfer speed since the size of transferred data is reduced. Besides, it can help to reduce the power dissipation on memory. For example, the 16-bit fixed-point data type will halve the memory cost compared with 32-bit floating-point data type.

The conversion between floating-point and fixed-point data can be presented as below:

$$f = N \times \frac{1}{2^m} \tag{14}$$

where N is a fixed-point integer value (in decimal format) which can be represented with n-bit length (in binary format), and m is the quantization factor of fractional length. The larger value of m will introduce the more accurate of the fractional part but with a narrow represent range. We use $<n, m>$ pair to represent the converted fixed-point. For example, in a fixed-point <8, 1> representation, we can represent a signed number within [-64, +63.5], and our fractional part is only precise to a quantum of 0.5. In a <8, 2> representation, we can convert a signed number within [−32, +31.75] with a quantum of 0.25. We can represent 0.75 with fixed-point <8, 2>, but using <8, 1> format will cause a decrease in precision.

In some CNN frameworks (such as YOLO V2), different parameters have a significant different numerical range. For example, there are four different parameter types (weights, IFMs/OFMs, scales, and biases). The scales/biases have a much larger numerical range than weights and IFMs/OFMs. If we use the same bit length (e.g., 8-bit), the scales/biases will introduce a considerable loss in precision. To solve this problem, the bit-width of scales/biases is expanded to use 16-bit instead of 8-bit to keep the precision. Luckily, the number of parameters for scales/biases only takes 0.09% of all the parameters, and 16-bit operation increased on-chip memory cost very slightly. The advantage of this mixed fixed-point design is to save on-chip memory for weights without compromising the precision of computation results in 16-bit.

B. COMBINATION OF BN AND SB OPERATION

Batch normalization (BN) allows us to use higher learning rates to speed up the training process and avoids the gradient vanishing problem and gradient exploding problem. It also acts as a regularizer to eliminate the need for dropout [33] and get a better detection rate. In the BN layer, we normalize each scalar feature independently. For a layer of d-dimensional input $x = (x^{(1)} \ldots x^{(d)})$, we will normalize each dimension as:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[X^{(K)}]}{\sqrt{Var[x^{(k)}] + \varepsilon}} \tag{15}$$

The parameters expectation $E[X^{(K)}]$ and variance $Var[x^{(k)}]$ are statistic acquired during the training process, ε is a hyper-parameter. As shown in (15), to calculate a batch normalization operation it will use two adders (or subtractors), one divider, and one square root operation in hardware. The operation does not only cost DSPs and LUTs on FPGA resources but also increases latency. Besides, the parameters of expectation and variance transferred between external memory and FPGA on-chip memory will introduce a large delay. There are two ways to solve this problem, the 1st method is merging BN/SB into MAC operation [34]. The 2nd method is combining BN and SB operation [35] to simplify the computation.

In SB layer, we introduce a pair of parameters $(\gamma^{(k)}, \beta^{(k)})$, which scale and shift the normalized value:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)} \tag{16}$$

We can combine these two equations as:

$$\begin{aligned} y^{(k)} &= \gamma^{(k)} \frac{x^{(k)} - E[X^{(K)}]}{\sqrt{Var[x^{(k)}] + \varepsilon}} + \beta^{(k)} \\ &= x^{(k)}\{\frac{\gamma^{(k)}}{\sqrt{Var[x^{(k)}] + \varepsilon}}\} + \{\beta^{(k)} - \frac{E[X^{(K)}]\gamma^{(k)}}{\sqrt{Var[x^{(k)}] + \varepsilon}}\} \\ &= x^{(k)}\phi + \theta \end{aligned} \tag{17}$$

where $\phi = \frac{\gamma^{(k)}}{\sqrt{Var[x^{(k)}]+\varepsilon}}$, and $\theta = \beta^{(k)} - \frac{E[X^{(K)}]\gamma^{(k)}}{\sqrt{Var[x^{(k)}]+\varepsilon}}$. Since all of the parameters (such as variance, expectation, scale, and bias) are collected in the training process, we can pre-compute ϕ and θ offline, which can greatly increase the speed of CNN-based FPGA accelerator. Besides, the offline pre-processing can reduce the DSPs and LUTs resources and arithmetic logic unit (ALU) power consumption during inference.

There are two reasons we combining BN and SB instead of merging BN/SB into MAC operation. (1) We use 8-bit weights and 16-bit scales/biases mixed fixed-point data representation. If we fuse BN/SB and weights, the merged weights

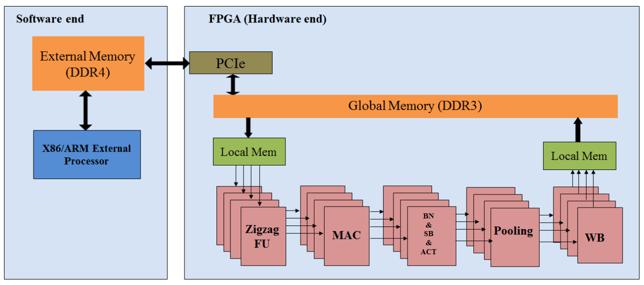


FIGURE 15. Overview pipeline architecture of CNN-based FPGA accelerator.

can only be in 16-bit otherwise the accuracy of 8-bit representation is unacceptable. However, we cannot give up 8-bit weights since its benefit for on-chip memory resource and data transfer bandwidth. (2) Compared with BN/SB combination, the help of merging BN/SB into MAC is very limited. We know that to get one output neuron, it has to compute $(C_{in} \times K \times K)$ MAC operations (where C_{in} is the number of input channels, $K \times K$ is the kernel size), and then apply one BN/SB computation on this MAC output. It needs $(C_{in} \times K \times K + 1)$ MAC operations totally. While merging BN/SB into MAC can only save one MAC operation which is not significant.

The overall pipeline architecture of the CNN-based FPGA accelerator is shown in FIGURE 15. There are five stages in this architecture. We combined BN and SB in the 3rd stage. The ACT is also concatenated after SB in this stage. In the MAC stage, the input weights are represented in 8-bit, and the output of MAC is 16-bit to guarantee high accuracy. The output of BN/SB/ACT (also the input of pooling) is in a 16-bit data type.

C. CNN ACCELERATOR FOR OBJECT DETECTION

As shown in FIGURE 16, our CNN-based FPGA accelerator has five pipeline stages, the 1st stage is ZFU which uses the zigzag scan to fetch the data and weight in parallel from global memory to FPGA on-chip buffer. The data are captured from a video camera and resized to a fixed input size (e.g., 416 × 416). The 2nd stage is the MAC operation combined with a FIFO structure to leverage the loop-carried data dependency. In the 3rd stage, we combined BN, SB, and ACT together. In particular, we pre-process the BN and SB offline and reduce the operations to multiplication and addition. For LReLU operation, we get the approximate activation value by using:

$$f(x) = \begin{cases} x & x \geqslant 0 \\ (x \gg 4) + (x \gg 5) & otherwise \end{cases} \tag{18}$$

Since shift operation does not cost hardware resources on FPGA, and adder only takes one cycle, which can reduce the area and save power. Equation (18) is equivalent to (4) since:

$$(x \gg 4) + (x \gg 5) = \frac{x}{2^4} + \frac{x}{2^5} \approx \frac{x}{10} \tag{19}$$

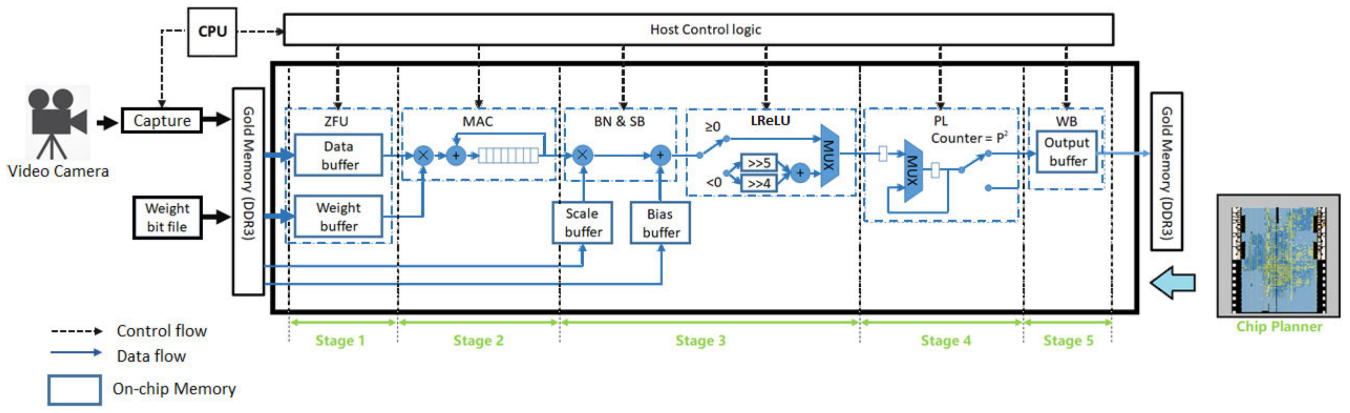


FIGURE 16. The architecture of hardware CNN accelerator system.

In this approach, the precision loss can be neglected. In the 4th stage, we use a comparator to get the maximum value between two inputs. We set a counter to control the output of the pooling result. If the counter reaches P^2, the output will be written to an output buffer. Otherwise, the comparator will continue to work. In the 5th stage, the output data will be sent back to global memory.

D. CNN ACCELERATOR WITH CAFFE INTERFACE

Our CNN-based accelerator supports Caffe [6] interface, which means the CNN accelerator can cooperate with the Caffe framework. As shown in FIGURE 17, our system can load the configuration and weights in Caffe format. The users can easily modify the models in the configuration file, e.g., the size of IFM/OFM, the kernel size, stride step, and the number of layers, etc. On the host side, our system can transfer the configuration and weight to our FPGA format and load the data to the DRAM. The data will load to FPGA on-chip memory through the PCI-E bus. After we finish the processing on FPGA, final results will be written to the DRAM in the host side, and output the classification and regression results.

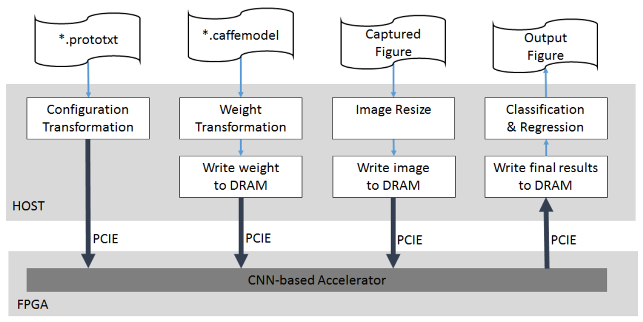


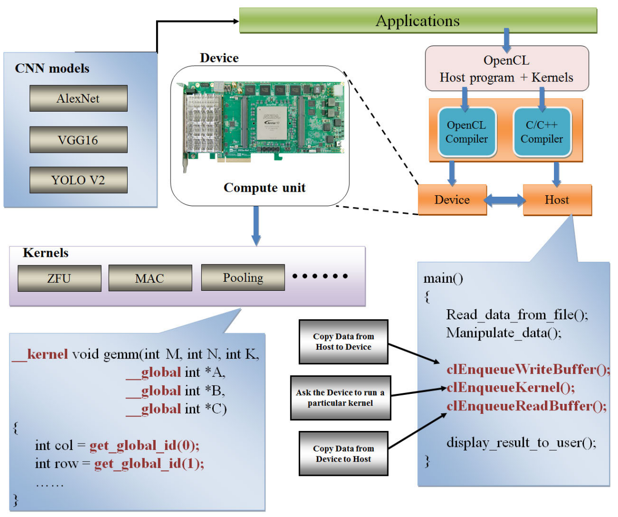
FIGURE 17. CNN-based accelerator with Caffe interface.

VI. EXPERIMENT SETUP

A. OPENCL BASED FPGA IMPLEMENTATION AND SYSTEM INTEGRATION

In this work, we use OpenCL based methodology combined with RTL math IP to design our CNN accelerator on Intel FPGA. As shown in FIGURE 18, the OpenCL framework constants of two components: (1) The host program and host compiler. (2) OpenCL kernel and offline compiler. The host program is working on CPU for managing tasks such as input/output (I/O) and display, and kernels are executed on FPGA to process intensive computation. When the host program sends a command to the kernel, the OpenCL runtime system will copy data from the host to FPGA and create an index space. Each instance of executing kernel runs in a work-item. We can apply any CNN applications on FPGA, e.g., AlexNet, VGG16, YOLO V2, etc.

We build up the FPGA system which uses PCI-E port, the system details listed in TABLE 1.


Applications
CNN models
AlexNet
VGG16
YOLO V2
Device
Compute unit
OpenCL Host program + Kernels
OpenCL Compiler
C/C++ Compiler
Device
Host
Kernels
ZFU
MAC
Pooling
••••••
Copy Data from Host to Device
Ask the Device to run a particular kernel
Copy Data from Device to Host

```
__kernel void gemm(int  M, int N, int K,
                   __global int *A,
                   __global int *B,
                   __global int *C)
{
    int col = get_global_id(0);
    int row = get_global_id(1);
    ……
}
```

```
main()
{
    Read_data_from_file();
    Manipulate_data();

    clEnqueueWriteBuffer();
    clEnqueueKernel();
    clEnqueueReadBuffer();

    display_result_to_user();
}
```

FIGURE 18. OpenCL system overview.

TABLE 1. FPGA system information.

CPU with FPGA system H/W S/W co-design	
CPU	Xeon E5-2620 V4
FPGA	Intel Arria 10
Operating System	CentOS 7

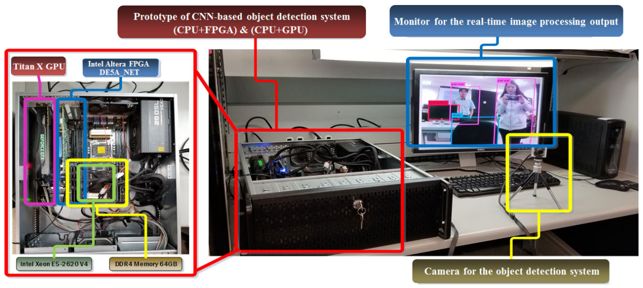

Prototype of CNN-based object detection system (CPU+FPGA) & (CPU+GPU)
Monitor for the real-time image processing output
Titan X GPU
Intel Altera FPGA DE5A_NET
Intel Xeon E5-2620 V4
DDR4 Memory 64GB
Camera for the object detection system

FIGURE 19. The prototype of CNN-based testbed overview.

As shown in FIGURE 19, our testbed integrated CPU, FPGA, and GPU. In Section VII, we will evaluate the performance of speed and power for different CNN-based applications on GPU, CPU, and FPGA, respectively.

B. EVALUATION CRITERIA

1) BANDWIDTH

There are two kinds of bandwidths in our program. One is for fetching data from DRAM to on-chip memory, and the other one is for writing back data from on-chip memory to DRAM. We can get these two functions as follows:

$$B_{wr} = \frac{Data_{width} \times Data_{size}}{kernel_{time}} \tag{20}$$

$$\begin{aligned} B_{rd} = (&Data_{size} \times Data_{width} \\ &+Weight_{size} \times Weight_{width} \\ &+Bias_{size} \times Bias_{width} \\ &+Scale_{size} \times Scale_{width})/kernel_{time} \end{aligned} \tag{21}$$

$$Efficient\ bandwidth\ (GOPS) = \frac{B_{rd} + B_{wr}}{10^9} \tag{22}$$

B_{wr} is the writing bandwidth which only needs to write the computed output data to the DRAM. B_{rd} is the reading bandwidth, which includes data such as input data, weight, bias, and scale. The efficient bandwidth is the total of B_{wr} and B_{rd}. Since we use Giga operations per second (GOPS) to represent the unit of bandwidth, it is divided by 10^9 in (22).

2) PRECISION

Average Precision (AP): For a given task and class, we use the precision/recall curve to evaluate the performance of output. Precision is the proportion of relevant instances among the retrieved instances. The recall rate is defined as the proportion of all positive examples among the total amount of relevant instances. The average precision [36] is used to evaluate the precision over several equally spaced recall levels.

$$AP = \frac{1}{N} \sum_{r \in \{0, \frac{1}{N-1}, \frac{2}{N-1}, \ldots, 1\}} P_{interp}(r) \tag{23}$$

where $P_{interp}(r)$ is the precision at each recall level r, and normally we set $N = 11$.

Mean average precision (mAP) is used to evaluate the average precision of $\mathbb{C}$ classes.

$$mAP = \frac{1}{\mathbb{C}} \sum_{i \in \{0,1,2,\ldots,\mathbb{C}\}} AP(c_i) \tag{24}$$

where $AP(c_i)$ is the average precision for class c_i.

Bounding box evaluation: For the object detection task, intersection over union (IOU) is an evaluation metric to measure bounding box overlap. To be considered a correct detection, the area of overlap a_0 between the predicted bounding box B_p and ground truth bounding box B_{gt} must exceed 0.5 (50%) by (25):

$$a_0 = \frac{area(B_p \cap B_{gt})}{area(B_p \cup B_{gt})} \tag{25}$$

$B_p \cap B_{gt}$ denotes the intersection of the predicted BB and ground truth BB, and $B_p \cup B_{gt}$ is the union of predicted BB and ground truth BB.

3) NUMBER OF FIXED-POINT OPERATIONS (OPs)

According to [37], to compute the number of fixed-point operations (OPs) for standard convolutional kernels, we have:

$$OPs = 2HW(C_{in}K^2 + 1)C_{out} \tag{26}$$

where H, W, and C_{in} are height, width, and the number of channels in IFM, respectively. K is the width/height of kernel (we assume the kernel is a square shape), and C_{out} is the number of channels in OFM.

For fully connected layers, we compute OPs as:

$$OPs = (2I - 1)O \tag{27}$$

where I is the input dimensionality, and O is the output dimensionality.

VII. EXPPERIMENTAL RESULTS AND PERFORMANCE EVALUATION

A. TRAINING

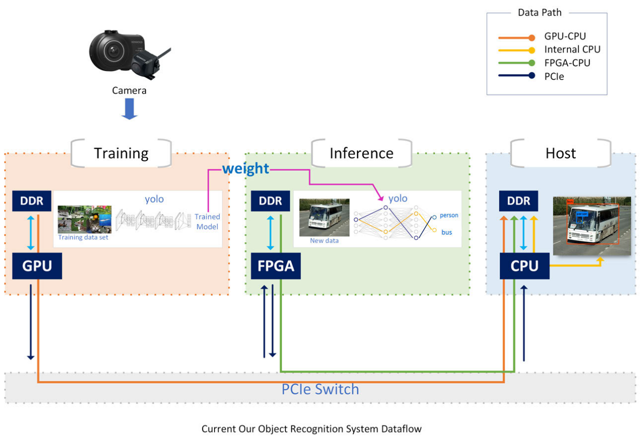


FIGURE 20. System dataflow.

As shown in FIGURE 20, we use Titan X GPU to train the weights. FPGA is used for inference, and CPU is working as

TABLE 2. Hardware usage on FPGA.

Component (total)	Percentage (%)
Logic utilization (427,200)	82
LUTs (707,600)	51
FFs (1,415,440)	37
RAMs (2,531)	54
DSPs (1,518)	27

a host to communicate with GPU or FPGA. In the first stage, we train the pre-train model with a fixed input size of 448 × 448 on the ImageNet dataset for 80 epochs and get the top-1 accuracy of 72.5% and top-5 accuracy of 91%, respectively. In the second stage, we use the pre-train model for transfer learning on the VOC dataset. We want YOLO V2 to be robust on different image sizes, so we change the image size every ten batches randomly during training instead of fixed input image size. We use 300,000 batches to train the model on the VOC dataset. The initial learning rate is set to 0.001. After 70,000 batches, we change the learning rate to 0.0001. After 140,000 batches, the learning rate changes to 0.00001 for fine-tune. Finally, the accuracy achieves 73.6 mAP on the YOLO V2 model for 416 × 416 input image size. Since our model is trained for different input resolutions, there is a trade-off between the speed and accuracy which is suitable for different applications.

B. HARDWARE USAGE ON FPGA

TABLE 2 lists the hardware usage of our CNN accelerator on FPGA when we set the parallelism factors $vector\_depth = 32$ and $vector\_width = 16$. Specifically, the $vector\_width$ of the first IFM is three because there are three channels for the input image. DSPs have dedicated floating/fixed-point arithmetic operation units such as multiply and accumulators. Since we use part of RTL math IP, the synthesis tool (Quartus in Intel FPGA) consumes LUTs instead of DSPs. Our design only costs 27% DSPs. Block RAMs are embedded memory systems on FPGA. LUTs and Registers can build and route arbitrary topologies for programmable logic.

C. RESULTS FOR IMAGENET CLASSIFICATION

We compare our design with the state-of-the-art CNN-based FPGA accelerators listed in TABLE 3. We improved the throughput and power efficiency for the proposed design without compromising the other parameter such as the use of hardware. The peak throughput of our accelerator is achieving 736.9 GOPS on the VGG16 net, and the power efficiency is reaching 27.09 GOPS/watt, which outperforms all the previous works [7], [10], [27], [29].

We have used only 27% DSP resources without compromising the throughput and power efficiency compared with the state-of-the-art design [7]. The power dissipation is also reduced with less use of DSP resources. The power efficiency is 3.27× more than the most advanced multi-FPGA structure-based design [8]. We have also compared our result with MobileNet V2 on Intel Arria 10 SoC FPGA [9], which used more BRAM and DSP than our design.

Through TABLE 3, we find that the throughput of AlexNet is lower than the VGG16 model. The reason is that some kernel sizes of AlexNet are larger than regular size, e.g., 11 × 11 and 5 × 5 compared with 3 × 3. The irregular kernel sizes make the accelerator performance inconsistent among different layers. On the contrary, VGG16 has a regular shape

TABLE 3. CNN-based FPGA implementation for classification.

	[10]	[9]	[29]	[8]	[27]	[7]	Our work	
Year	2018	2018	2018	2018	2017	2018	2019	2019
CNN model	VGG16	MobileNetV2	VGG16	VGG16	VGG19	VGG16	AlexNet	VGG16
FPGA	Stratix V GXA7	Intel Arria 10 SoC	Zynq XC7Z045	Virtex-7 VX690t	Stratix V GSMD5	Intel Arria 10	Intel Arria 10	Intel Arria 10
Clock (MHz)	200	133	150	150	150	200	200	200
BRAMs (36Kb)	1,377*	1,844*	486	1,220	919*	2,232*	1,366*	1,366*
DSPs	256	1,278	780	2,160	1,036	1,518	410	410
LUTs	-	-	182.6K	-	-	-	360K	360K
FFs	-	-	127.7K	-	-	-	523.7K	523.7K
GOP	30.95	0.64	30.95	30.95	39.26	30.93	1.46	30.95
Precision (bits)	16	16	16	16	16	16	8-16	8-16
Latency (ms)	46.3	3.75	226	106.6	107.7	43.2	3.7	42
Throughput (GOPS)	669.1	170.6	137	290	364.4	715.9	394.6	736.9
Power(W)	-	-	9.63	35	25	-	27.2	27.2
Power efficiency (GOPS/W)	-	-	14.2	8.28	14.57	-	14.51	27.09

* Intel FPGA: the size of BRAM refers to 20Kb

TABLE 4. Comparison among CPU, GPU, and FPGA designs for CNN object detection.

	CPU	CPU	GPU	GPU	[11]	[12]	Track 2 [34]	Track 3 [34]	Our work	
Year	-	-	-	-	2018	2018	2018	2018	2019	2019
CNN model	Tiny YOLO V2	YOLO V2	Tiny YOLO V2	YOLO V2	Lightweight YOLO V2	Tincy YOLO	YOLO V2	eSSD-MobileNet V1	Tiny YOLO V2	YOLO V2
Device	Intel E5 2620 V4	Intel E5 2620 V4	Titan X	Titan X	Zynq UltraScale+ ZCU102	Zynq UltraScale+ XCZU3EG	Jetson TX2	Jetson TX2	Intel Arria 10	Intel Arria 10
Clock (MHz)	2,100	2,100	1,400	1,400	300	-	850	850	200	200
BRAMs (36Kb)	-	-	-	-	1,706	-	-	-	1,366*	1,366*
DSPs	-	-	-	-	377	-	-	-	410	410
LUTs	-	-	-	-	135K	-	-	-	360K	360K
FFs	-	-	-	-	370K	-	-	-	523.7K	523.7K
GOP	6.97	34.9	6.97	34.9	14.97	4.5	-	-	6.97	29.6
Weights (WB)	60.5	193	60.5	193	-	-	-	-	15.8	44.2
Precision (bits)	32†	32†	32†	32†	1-32	1-3	16	16	8-16	8-16
mAP	57.1	76	57.1	76	67.6	48.5	38.98§/ 43.0	18.32§/ 65.8	56.5	73.6
fps	0.67	0.16	207‡/ 105	67‡/ 33	40.8	16	-	-	105	25
Throughput (GOPS)	4.67	5.6	1,442.8‡/ 731.9	2,338‡/ 1,151.7	610.9	72	-	-	731.9	740
Power(W)	80	80	219.1	219.1	-	6	-	-	27.2	27.2
Power efficiency (GOPS/W)	0.06	0.07	6.58‡/ 3.34	10.67‡/ 5.26	-	12	-	-	26.9	27.2
Energy (J)	119.4	500	1.06‡/ 2.09	3.27‡/ 6.64	-	0.375	1.54	0.41	0.26	1.09
mAP/energy	0.478	0.152	53.95‡/ 27.36	23.24‡/ 11.45	-	129.3	25.31§/ 27.9	44.47§/ 159.7	218.1	67.65

* Intel FPGA: the size of BRAM refers to 20Kb
† Floating-point
‡ The data is tested when cuDNN is on
§ The accuracy is tested on ImageNet [38] dataset

in each layer, which makes the performance of the CNN accelerator more stable and promising than AlexNet.

D. RESULTS FOR CNN-BASED OBJECT DETECTION

TABLE 4 lists the results for object detection tasks based on our FPGA accelerator on the YOLO V2 framework. The results show that our work can achieve 105 fps on the tiny YOLO V2 network with 56.5 mAP and 25 fps on the full YOLO V2 network with 73.6 mAP on the VOC dataset, respectively. The accuracy and speed are better than the state-of-the-art works [11], [12], [34]. For the Titan X GPU case, we have tested the data when cuDNN is on and off, respectively. cuDNN is a deep learning library to speedup NVIDIA GPU. Compared with Titan X GPU, the power efficiency of our FPGA accelerator can get 26.9∼27.2 GOPS/watt, which is 4.08× better for tiny YOLO V2 and 2.55× better for full YOLO V2, respectively. The average power efficiency is ∼3.3× better compared with Titan X GPU. Compared with Intel E5 2620 CPU, the power efficiency of our FPGA accelerator is 448× better for tiny YOLO V2 and 388× better for full YOLO V2. The average performance is ∼418× better compared with CPU.

Literature [34] summarized the 2018 winners' solution for Low-Power Image Recognition Challenge (LPIRC). There are three tracks in LPIRC. The 1^{st} track is for ImageNet classification on Google Pixel 2 XL phone. The 2^{nd} track is an online competition for object detection on NVIDIA Jetson TX2 GPU, and the 3^{rd} track is onsite competition for object detection without restriction on hardware or software. Track 2 and track 3 use the term mAP/energy as a criterion to evaluate the performance combining accuracy, speed, and power. The winner of track 2 applied several techniques on the YOLO V2 framework such as pipelining, tucker decomposition, 16-bit quantization, merging BN, and input size reduction, etc, and it achieved 27.9 mAP/energy. The winner of the 3^{rd} track used eSSD-MobileNetV1 framework and achieved 159.7 mAP/energy. This structure extracted additional features with depthwise convolution and 1 × 1 convolution and predicted classification and BB with 1 × 1 convolution, which reduced computational resources. Our FPGA design can achieve 218.1 and 67.65 mAP/energy for tiny YOLO V2 and YOLO V2, respectively. The performance is 7.8× and 2.4× better compared with winner of track 2 in [34]. The value in terms of mAP/energy for YOLO V2 is smaller than the winner of track 3 [34] because the criteria mAP/energy is a linear analysis among accuracy, speed, and power. However, the empirical case shows that speed and accuracy is a nonlinear relationship. E.g., when the speed is halved, the accuracy cannot improve twice. For the tiny YOLO V2 framework, our performance is still 1.37× better the winner of track 3. According to the results, our FPGA accelerator outperforms the most advanced Titan X GPU, mobile Jetson TX2 GPU, and CPU respect to power efficiency. Compared with the other FPGA designs [11], [12] for the YOLO framework, our design not only achieves the highest throughput but also has the highest accuracy and power efficiency.

E. POWER DISSIPATION

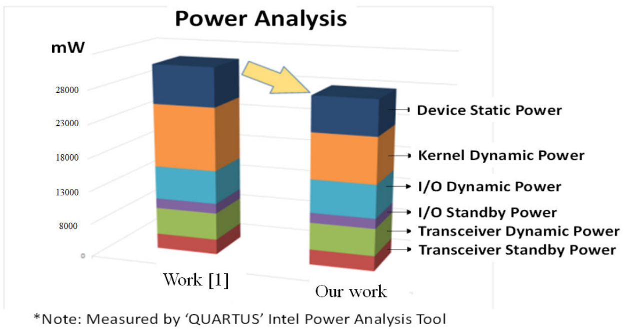


FIGURE 21. Power consumption comparison between [1] and our CNN accelerator.

FIGURE 21 depicts the power comparison between work [1] and our CNN-based FPGA accelerator. There are three components dissipated in the power consumption: transceiver power, I/O power, and kernel power. The transceiver power comes from the high-speed serial interface (HSSI) such as the PCI-E interface. The I/O power is mainly costed by the input/output data transferred between external memory (DDR3) and local memory/registers. The device uses SSTL-15 I/O standard (DDR3 I/O standard). The kernel power comes from the kernel computation part of the accelerator, which includes static and dynamic power. The total power is reduced from 29 W to 27.2 W which is a 6.2% reduction in power. The kernel power is reduced from 15.4 W to 12.2 W, which is reduced by 20%. The power reduction is mainly due to three reasons. (1) Data movement between on-chip memory and off-chip (e.g., DRAM) is reduced by using an optimized pipeline structure. (2) Computational kernels optimization reduces hardware resources. (e.g., offline pre-processing BN/SB and approximation expression reduce DSPs, LUTs, and ALU, etc) (3) Mixed fixed-point data width helps to reduce the power consumption on memory.

VIII. CONCLUSION

In this paper, we present a highly efficient CNN-based FPGA accelerator combined parallel architecture and pipeline structure which supports the Caffe framework on Intel FPGA. For the ImageNet classification task, it can achieve the throughput of 736.9 GOPS on the VGG16 net. Furthermore, we have applied the CNN accelerator on complex advanced multi-object detection frameworks and get the real-time speed of 105 fps for tiny YOLO V2 with 56.5 mAP and 25 fps for full YOLO V2 with 73.6 mAP on VOC dataset, respectively. The power efficiency of our system is ∼ 3.3× better than Titan X GPU and ∼ 418× better than Intel Xeon E5 CPU on average. The CNN-based FPGA accelerator can save kernel power by 20%. Our system can be applied to the industrial field, such as road object recognition for autonomous vehicles or drones. In the future, more studies will be conducted on high-speed and low power applications on the embedded FPGA.

ACKNOWLEDGMENT

The authors thank their colleagues from KETI and KEIT who provided insight and expertise that greatly assisted the research and greatly improved the manuscript.

REFERENCES

[1] S. Li, Y. Luo, K. Sun, and K. Choi, "Heterogeneous system implementation of deep learning neural network for object detection in OpenCL framework," in *Proc. Int. Conf. Electron., Inf., Commun. (ICEIC)*, Jan. 2018, pp. 1–4.

[2] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 7263–7271.

[3] K. Group, "The OpenCL specification," Khronos OpenCL Working Group, Beaverton, OR, USA, Tech. Rep. version 2.1, 2017.

[4] *Cuda Toolkit Documentation: Nvidia Developer Zone—Cuda c Programming Guide V8.0*, Nvidia, Santa Clara, CA, USA, 2017.

[5] (2015). *CuDNN*. Nvidia, Santa Clara, CA, USA. [Online]. Available: https://developer.nvidia.com/cudnn 2015

[6] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. ACM Int. Conf. Multimedia MM*, 2014, pp. 675–678.

[7] 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.

[8] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A framework for acceleration of CNN training on deeply-pipelined FPGA clusters with work and weight load balancing," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 394–3944.

[9] L. Bai, Y. Zhao, and X. Huang, "A CNN accelerator on FPGA using depthwise separable convolution," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 65, no. 10, pp. 1415–1419, Oct. 2018.

[10] 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*, Feb. 2018, pp. 117–126.

[11] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A lightweight YOLOv2: A binarized CNN with a parallel support vector regression for an FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 31–40.

[12] T. B. Preuser, G. Gambardella, N. Fraser, and M. Blott, "Inference of quantized neural networks on heterogeneous all-programmable devices," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 833–838.

[13] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Proc. Int. Conf. Artif. Neural Netw.* Berlin, Germany: Springer, 2014, pp. 281–290.

[14] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL deep learning accelerator on arria 10," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays FPGA*, 2017, pp. 55–64.

[15] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-S. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays FPGA*, 2016, pp. 16–25.

[16] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2016, pp. 1–9.

[17] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of FPGA-based deep convolutional neural networks," in *Proc. 21st Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2016, pp. 575–580.

[18] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martinez, and C. Guestrin, "GraphGen: An FPGA framework for vertex-centric graph computation," in *Proc. IEEE 22nd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2014, pp. 25–28.

[19] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "CNP: An FPGA-based processor for convolutional neural networks," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2009, pp. 32–37.

[20] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays FPGA*, 2015, pp. 161–170.

[21] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2014, pp. 682–687.

[22] T. Fujii, S. Sato, and H. Nakahara, "A threshold neuron pruning for a binarized deep neural network on an FPGA," *IEICE Trans. Inf. Syst.*, vol. 101, no. 2, pp. 376–386, Feb. 2018.

[23] M. Hailesellasie, S. R. Hasan, F. Khalid, F. A. Wad, and M. Shafique, "FPGA-based convolutional neural network architecture with reduced parameter requirements," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[24] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 11, pp. 2072–2085, Nov. 2019.

[25] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.

[26] M. Motamedi, P. Gysel, and S. Ghiasi, "PLACID: A platform for FPGA-based accelerator creation for DCNNs," *ACM Trans. Multimedia Comput., Commun., Appl.*, vol. 13, no. 4, p. 62, 2017.

[27] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 152–159.

[28] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated FPGAs: FPGA framework for convolutional neural networks," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2016, pp. 265–268.

[29] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.

[30] 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.

[31] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," 2016, *arXiv:1602.07360*. [Online]. Available: http://arxiv.org/abs/1602.07360

[32] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*. [Online]. Available: http://arxiv.org/abs/1704.04861

[33] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015, *arXiv:1502.03167*. [Online]. Available: http://arxiv.org/abs/1502.03167

[34] S. Alyamkin *et al.*, "Low-power computer vision: Status, challenges, and opportunities," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 411–421, Jun. 2019.

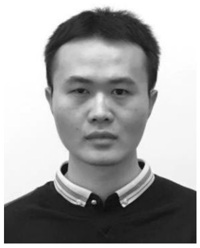
[35] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Automatic compilation of diverse CNNs onto high-performance FPGA accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 2, pp. 424–437, Feb. 2020.

[36] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal visual object classes (VOC) challenge," *Int. J. Comput. Vis.*, vol. 88, no. 2, pp. 303–338, Jun. 2010.

[37] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," 2016, *arXiv:1611.06440*. [Online]. Available: http://arxiv.org/abs/1611.06440

[38] ImageNet. (2016). *ImageNet*. [Online]. Available: http://www.image-net.org/

[39] https://www.bilibili.com/video/av62554595



SHUAI LI received the B.S. and M.S. degrees in electrical engineering from Peking University, Beijing, China, in 2009 and 2012, respectively, and the Ph.D. degree in electrical engineering from the Illinois Institute of Technology, Chicago, IL, USA, in 2019.

His current research interests include high-performance and low-power digital IC design and deep learning.

YUKUI LUO (Student Member, IEEE) received the B.S. degree in automation from the Shanghai University of Engineering Science, Shanghai, China, in 2013, and the M.S. degree in electrical engineering from the Illinois Institute of Technology, Chicago, IL, USA, in 2017. He is currently pursuing the Ph.D. degree with the University of Illinois at Chicago, Chicago.

His research interests include hardware security, machine learning, VLSI design, and computer architecture.

KUANGYUAN SUN received the B.S. degree in computer science from the Illinois Institute of Technology, Chicago, IL, USA, in 2019. He is currently pursuing the master's degree with Rice University.

His current research interests include deep learning algorithms and computer vision.

NANDAKISHOR YADAV (Member, IEEE) received the M.S. degree in electronics from the School of Electronics, Devi Ahilya University, Indore, India, in 2005, the M.Tech. degree in microelectronics from Panjab University, Chandigarh, India, in 2008, and the Ph.D. degree from the Indian Institute of Information Technology and Management, Gwalior, India, in 2015. He is currently a Senior Research Associate at the Design and Automation Laboratory, Illinois Institute of Technology, Chicago, IL, USA. His current research interests include FinFET-based memory design, low-power very large scale integration (VLSI) design, digital VLSI design, circuit optimization for performance, power and variation tolerant design using soft computing, and process variation aware circuits for future VLSI Technology. He is a member of the Indian Microelectronics Society, Chandigarh.

KYUWON KEN CHOI (Senior Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, in 2002.

During the Ph.D. degree, he proposed and conducted several projects supported by NASA, DARPA, NSF, and SRC regarding power-aware computing and communication. Since 2004, he had been with the Takayasu Sakurai Laboratory, The University of Tokyo, Japan, as a Postdoctoral Research Associate, working on leakage power reduction circuit techniques. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, Illinois Institute of Technology. He was a Senior CAD Engineer and a Technical Consultant for low-power system-on-chip (SoC) design in Samsung Semiconductor, Broadcom, and Sequence Design, prior to joining IIT. In the past, he had eight-year industry experience in the area of VLSI chip design from compiler level to circuit level. Last few years, by using his novel low-power techniques, several processor, and control VLSI chips were successfully fabricated in deep-submicrometer CMOS technology and more than 80 peer-reviewed journals and conference papers have been published. He is currently the Director of the VLSI Design and Automation Laboratory (DA-Lab), IIT. His research interests include ultra-low power circuit and IC design for multimedia, and mobile or energy harvesting applications. He is also a TPC member for several IEEE circuit design conferences. For last seven years, DA-Lab has been awarded several projects about hardware design for modern applications, including IoT, AI, deep learning, unmanned vehicles, and energy harvesting, about 1.6 million U.S. dollars from U.S. federal agencies and Korea Government. He has served as the President in the Korean-American Scientists and Engineers Association (KSEA)-Chicagoland Chapter. He is currently the Technical Group Director and an Advisory Election Committee in the KSEA Head Quarter. He is also the Editor-in-Chief of the *Journal of Pervasive Technologies* and a Guest Editor of Springer and Wiley Journals.