A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection

Duy Thanh Nguyen, Tuan Nghia Nguyen, Hyun Kim, *Member, IEEE*, and Hyuk-Jae Lee, *Member, IEEE*

***Abstract*— Convolutional neural networks (CNNs) require numerous computations and external memory accesses. Frequent accesses to off-chip memory cause slow processing and large power dissipation. For real-time object detection with high throughput and power efficiency, this paper presents a Tera-OPS streaming hardware accelerator implementing a you-only-look-once (YOLO) CNN. The parameters of the YOLO CNN are retrained and quantized with the PASCAL VOC data set using binary weight and flexible low-bit activation. The binary weight enables storing the entire network model in block RAMs of a field-programmable gate array (FPGA) to reduce off-chip accesses aggressively and, thereby, achieve significant performance enhancement. In the proposed design, all convolutional layers are fully pipelined for enhanced hardware utilization. The input image is delivered to the accelerator line-by-line. Similarly, the output from the previous layer is transmitted to the next layer line-by-line. The intermediate data are fully reused across layers, thereby eliminating external memory accesses. The decreased dynamic random access memory (DRAM) accesses reduce DRAM power consumption. Furthermore, as the convolutional layers are fully parameterized, it is easy to scale up the network. In this streaming design, each convolution layer is mapped to a dedicated hardware block. Therefore, it outperforms the "one-size-fits-all" designs in both performance and power efficiency. This CNN implemented using VC707 FPGA achieves a throughput of 1.877 tera operations per second (TOPS) at 200 MHz with batch processing while consuming 18.29 W of on-chip power, which shows the best power efficiency compared with the previous research. As for object detection accuracy, it achieves a mean average precision (mAP) of 64.16% for the PASCAL VOC 2007 data set that is only 2.63% lower than the mAP of the same YOLO network with full precision.**

***Index Terms*— Binary weight, low-precision quantization, object detection, streaming architecture, you-only-look-once (YOLO).**

I. Introduction

OBJECT detection is a challenging task in computer vision. Lately, deep learning has been widely adopted in object detection owing to the support of powerful computation devices, such as GPU. Therefore, several promising approaches have been proposed for object detection with deep learning, such as single-shot-multibox-detection (SSD) [1], faster R-convolutional neural network (CNN) [2], and you-only-look-once (YOLO) [3]. YOLO performs one of the best tradeoffs between the accuracy and the speed for object detection. It is a single neural network that predicts the object bounding boxes and class probabilities in a single evaluation.

Although GPU is widely used for processing deep learning algorithms, such as YOLO, it becomes inefficient in optimization, such as the selection of the width of the data bit and scheduling data access by the external memory. Therefore, extensive research has been conducted to design a deep learning accelerator for application-specific integrated circuit (ASIC) and a field-programmable gate array (FPGA) to address this challenge. FPGAs have been widely used for high-efficient deep learning owing to their flexible design and short development cycles. Several implementations use the floating-point representation that has a large computation cost [4]–[6]. Recent works have demonstrated that a floating-point representation is unnecessarily redundant [7], and the CNNs can be retrained and quantized to a very low-bit precision (1 or 2 bits) without significant loss of accuracy [8]–[10]. The quantization enables the design of a fast and power-efficient CNN accelerator using an FPGA that stores the entire quantized CNN model in its on-chip block RAMs of tens to hundreds of Mb. For example, Virtex Ultrascale+ is an FPGA comprising block RAMs (up to 500 Mb) arranged in small units, thereby providing extremely high memory bandwidth and low-power compared with a design using a single big static random access memory (SRAM) or off-chip memory. Thus, FPGA combined with low-bit CNN quantization enables to design a low-power accelerator for deep networks offering a throughput of the order of tera operations per second (TOPS).

Manuscript received October 10, 2018; revised January 30, 2019; accepted March 3, 2019. This work was supported in part by The Project of Industrial Technology Innovation through the Ministry of Trade, Industry and Energy (MOTIE) under Grant 10082585 and in part by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean Government (MSIT) (Development of intelligent semiconductor technology for vision recognition signal processing for vehicle based on multi-sensor fusion) under Grant 2017-0-00721-001. *(Corresponding author: Hyun Kim.)*

D. T. Nguyen, T. N. Nguyen, and H.-J. Lee are with the Inter-University Semiconductor Research Center, Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826, South Korea (e-mail: thanhnd@capp.snu.ac.kr; nghiant@capp.snu.ac.kr; hyuk_jae_lee@capp.snu.ac.kr).

H. Kim is with the Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul 01811, South Korea, and also with the Research Center for Electrical and Information Technology, Seoul National University of Science and Technology, Seoul 01811, South Korea (e-mail: hyunkim@seoultech.ac.kr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2019.2905242

There are a number of FPGA designs using Vivado high-level synthesis (HLS) [4], [6], [11]–[13]. However, these are inefficient in terms of both hardware resource and performance. Zhang *et al.* [4] present a single processing engine (PE) using a theoretical roofline model to design an accelerator for the execution of each layer. However, the accelerator is found to consume a significant portion of the FPGA chip while running at a modest throughput of 61 giga operations per second (GOPS) for a small network of five levels. Designs in [6] and [13] propose a fused-convolutional layer to reduce the off-chip accesses by optimizing the intermediate data between the neighboring layers in a group. Nevertheless, the authors report a significantly larger number of block RAMs (for storing intermediate data) and DSPs (due to additional control logic). Similar to [4], the CNN accelerator in [11] optimizes the data path using loop unrolling and tiling for the enhanced performance of each layer. The authors also use Vivado HLS to design the CNN accelerator for each layer, which is mapped to the PE in their accelerator in a pipelined manner. However, the entire intermediate feature-maps generated from each layer are stored in a double buffer so that this scheme does not scale well when the CNN becomes deeper owing to the demand of large buffers. The design in [14] also faces the same problem even though it delivers high performance for the AlexNet network. Another recent work using Vivado HLS in [12] employs the same optimization as proposed in [4]. In addition, the available resources are partitioned to make multiple convolutional layer processors (CLP) of smaller size rather than a single large CLP. This paper proposes a scheme to decide the number of required CLPs, the resource partitioning among these CLPs, and a scheduling algorithm to utilize their concurrent operations effectively. As the network is not quantized, the intermediate data are generally too large to be stored in on-chip memory. Hence, all CLPs read their inputs and write their outputs to external memory. Consequently, this design requires a very high memory bandwidth. Facing a similar problem, an approach in [15] presents a register-transfer level (RTL) compiler to generate an RTL code for each layer of a given network. In this design, each layer also reads inputs and writes outputs to a dynamic random access memory (DRAM). Each layer operates sequentially, which means that the next layer starts only when the current layer finishes its computation. This nonpipelined processing and frequent accesses to external DRAM lower the processing speed significantly.

Unlike a conventional convolution, the Winograd minimal filtering algorithm introduced in [16] is employed in [17] and [18] to speed up the convolutional computations. In [17], additional optimizations, including loop unrolling and tiling, are proposed to increase the throughput up to 1382 GOPS for AlexNet. With the same filtering algorithm, the design in [18] achieves a throughput of 2.94 TOPS for VGG network. Nevertheless, this design still demands an excessive number of DSPs and look-up tables (LUTs) even though the Winograd algorithm reduces the number of multipliers significantly. Moreover, the design of a single large convolutional layer has an inherent drawback. The authors also report that the performance of the network decreases, as it goes deeper owing to the overhead of data transfer back and forth between the CNN accelerator and external memory.

To reduce expensive external memory accesses, the resolution of number representation is reduced in [19] and [20]. They aggressively quantize the weight and the activation to a single bit. The multiplier-accumulator (MAC) operation is replaced with a low-cost pop-count computation, and the comparator in a max-pooling layer is implemented by an OR gate. Liang *et al.* [20] report the need for the floating-point number for batch normalization to avoid severe degradation of the accuracy. This shows an example that the performance of the binary network is very poor for a challenging data set, such as ImageNet.

For the implementation of YOLO, several FPGA designs have been proposed. Tincy YOLO, presented in [21], uses an extended version of the design in [19] to off-load 12 hidden layers to programmable logics in Zynq Ultrascale+ FPGA. The hardware accelerator processes these hidden layers one by one. Moreover, the first and last layers are run on software causing a low frame rate. Lightweight YOLO-v2 is proposed in [22] to combine a binary network with support vector machine (SVM) regression. The authors design a shared streaming binary convolutional circuit in which each layer is processed sequentially. Although these previous designs succeed in the speed up by reducing the complexity of the algorithm, they do not consider the reduction of external memory accesses.

To avoid the frequent off-chip access for intermediate data or large interlayer double buffers caused by unoptimized data path in previous works, this paper proposes an efficient Tera-OPS streaming architecture design. YOLOv2 network [3] is used for evaluating the performance of the proposed FPGA design in terms of both hardware performance and detection accuracy. The network is retrained and quantized using 1-bit weight and flexible low-bit activation. The main contributions of this paper are summarized as follows.

1) A binary weight, flexible low-bit activation, hardware-centric quantization, and a retraining method for YOLO CNN are presented. This paper shows that even the binary weight and 3-to-6-bit activation are adequate to realize the desired accuracy of object detection. The advantages of this quantization are as follows: 1) it requires a minimum number of DSPs, as the convolutional kernel contains only summations and 2) binary weight enables storing the entire network model in an on-chip memory to minimize the off-chip accesses, thereby enhancing the performance.
2) A scalable and high-accuracy streaming architecture for real-time object detection is proposed. The intermediate data are reused to minimize the size of the input buffer of each convolution layer while eliminating the accesses to the off-chip memory. The convolutional layers are fully parameterized. Thus, it is easy to change the network structure.
3) The proposed architecture is implemented, and its relative merits are highlighted by comparing with the previous works. A real-time demo for the object detection is also presented.

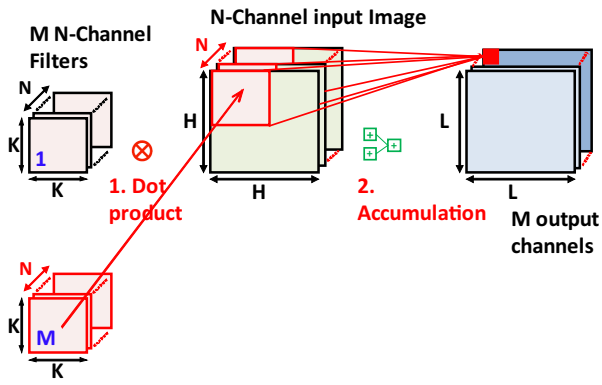

Fig. 1. Convolutional computation.

Algorithm 1 Pseudocode for Original Convolution Layer

```
in[N][H][H]: input images (N channels)
W[M][N][K][K]: weight
out[M][H][H]: output images (M channels)
for oc = 0; oc < M; oc++ do
  for r = 0; r < H; r++ do
    for c = 0; c < H; c++ do
      for ic = 0; ic < N; ic++ do
        for i = 0; i < K; i++ do
          for j = 0; j < K; j++ do
            out[oc][r][c] += W[oc][ic][i][j] * in[ic][r+i][c+j];
```

4) The proposed method can be easily extended to the previous designs as well as YOLO-v2. It can also expect a considerable enhancement in throughput by solving the off-chip access that suffered in the previous designs.

The rest of this paper is organized as follows. Section II introduces the CNN, YOLO-v2, and low-precision network quantization/retraining. Section III presents the optimization of the algorithm for the proposed design. The proposed architecture is elaborated in Section IV. The experimental results are shown in Section V. Finally, Section VI concludes this paper.

II. Background

A CNN is typically composed of basic layers: convolution, normalization, pooling, and fully connected. Considering that the focus is on object detection, the fully connected layer is not discussed in this paper.

A. Conventional CNN

1) Convolutional Layer: The convolutional layer is used to extract higher features from the input image. The convolutional computation is shown in Fig. 1. The input image, comprising N channels, is convolved with M number of N-channel filters to produce an M-channel output image. Each kernel has a size of $K \times K$. Algorithm 1 elaborates the convolutional operation in detail. For simplicity, the stride is assumed to be 1, and the bias is assumed to be 0. As a result, the output image has the same size as the input image.

Algorithm 2 Pseudocode for Original 2×2 Max-Pooling Layer With Stride $= 2$

```
in[N][2*H][2*H] : input images (N channels)
out[N][H][H] : output images (N channels)
for r = 0; r < H; r++ do
  for c = 0; c < H; c++ do
    for ic = 0; ic < N; ic++ do
      out[ic][r][c] = max(in[ic][2*r][2*c],
in[ic][2*r+1][2*c], in[ic][2*r][2*c+1],
in[ic][2*r+1][2*c+1]);
```

The number of operations for a convolutional layer can be calculated as NOPS $=2\times K\times K\times M\times N\times H\times H$. The constant value 2 implies that each MAC needs a multiplication and an addition.

2) Max-Pooling Layer: The max-pooling layer is used to reduce the size of the feature-maps, thereby reducing the amount of computation in the network, and to control the overfitting. The original max-pooling computation is explained in Algorithm 2. The size of the output image from the max-pooling layer is half of that of the input image.

3) Batch Normalization: Batch normalization [23] has proven to be effective in training the CNN. It helps the training to converge faster and prevent the network from overfitting. With batch normalization, the output of each convolutional layer is normalized to reduce the internal covariate shift. It is essential for both the training and inference phases.

The original batch normalization is as follows:

$$y = \frac{\gamma^{(i)}(act - \mu^{(i)})}{\sqrt{[\sigma^{(i)}]^2 + \varepsilon}} + \beta^{(i)} \tag{1}$$

where y and act are the outputs of batch-normalization and convolutional computation, respectively. $\mu^{(i)}$ and $[\sigma^{(i)}]^2$ are the channelwise mean and variance of activations, respectively. $\gamma^{(i)}$ and $\beta^{(i)}$ are the channelwise scale and bias, respectively.

B. Quantization of CNN Using Binary Weight and Low-Bit Activation

1) Binary Weight: Weights of each kernel are represented by only two values as shown in the following:

$$w_j^{b(i)} = \begin{cases} 1, & \text{if } w_j^{(i)} > 0 \\ -1, & \text{if } w_j^{(i)} \leq 0 \end{cases} \tag{2}$$

where $W_j^{b(i)}$ is the binary weight and $W_j^{(i)}$ is the original weight value jth in the ith kernel. For binary weight network, the convolutional layer is formulated as follows:

$$\text{act} = (x \otimes w^{b(i)}) \times \mu_W^{(i)} = x_{W(i)} \times \mu_W^{(i)} \tag{3}$$

where $\mu_W^{(i)}$ is the mean of weights of the ith channel and x is the input activation that is convolved with the ith weight kernel.

2) Uniform Quantization for Activation: Activations are quantized and represented by a fixed number of bits. Provided

the number of bits and quantization step s, the quantized values are computed by (6)

$$q_{\max} = s \times (2^{n-1} - 0.5) \tag{4}$$

$$p(x) = s \times \left(\text{round}\left(\frac{x}{s} + 0.5\right) - 0.5\right) \tag{5}$$

$$q(x) = \begin{cases} q_{\max}, & \text{if } p(x) > q_{\max} \\ -q_{\max}, & \text{if } p(x) < -q_{\max} \\ p(x), & \text{otherwise.} \end{cases} \tag{6}$$

III. Algorithmic Optimization for the Proposed Streaming Architecture

A. Hardware-Centric Quantization

This paper presents a method to train a low-precision model for the proposed streaming hardware accelerator. Previous studies in [8] and [20] show that the last layer is highly sensitive to low-precision quantization. Therefore, the weights in this layer are quantized to an 8-bit fixed point, and the activations are quantized to a 16-bit fixed point to minimize the loss of accuracy of quantization. In other layers including the first layer, the weights and output activations are quantized to 1 bit and 3–6 bits, respectively. It is noteworthy that the input image is in the RGB format for the first layer.

1) Optimization for Batch Normalization: To reduce the number of calculations at the inference phase, (1) is reformulated as follows:

$$y = x_{W(i)} \times \gamma_w^{(i)} + \beta_w^{(i)} \tag{7}$$

where $\gamma_w^{(i)}$ and $\beta_w^{(i)}$ are the new scale and bias factors that can be computed beforehand

$$\gamma_w^{(i)} = \frac{\mu_W^{(i)} \times \gamma^{(i)}}{\sqrt{[\sigma^{(i)}]^2 + \varepsilon}} \tag{8}$$

$$\beta_w^{(i)} = -\frac{\mu_W^{(i)} \times \mu^{(i)}}{\sqrt{[\sigma^{(i)}]^2 + \varepsilon}} + \beta^{(i)}. \tag{9}$$

As the batch normalization parameter is sensitive to small errors, the new scale and bias factors are quantized to 16-bit fixed-point value to minimize the accuracy loss. By using (7), the hardware for batch normalization requires only one multiplication and one addition, thereby reducing the data-path delay.

2) Leaky Rectified Linear Unit: Compared with the rectified linear unit (ReLU), the leaky ReLU helps prevent the neurons from dying during training; thus, it is more stable. The leaky coefficient a is chosen as 0.125 empirically for both the training and inference phases to replace floating-point multiplication by a 3-bit right shift operation

$$g(x) = \begin{cases} x, & \text{if } x > 0 \\ ax, & \text{if } x \leq 0. \end{cases} \tag{10}$$

3) Flexible Low-Bit Activation Quantization: Activations are quantized and represented by a fixed number of bits as shown in (4)–(6). The research in [24] shows that each convolutional layer is quantized using a different number of bits while preserving the accuracy of the quantized network. The layers, which have a large number of parameters, seem to be more redundant. Thus, they can be quantized using less number of bits. Following this finding, the activations from different layers of YOLO CNN are flexibly quantized. For Tiny YOLO-v2 [3] (6.97 GOP) and Sim-YOLO-v2 (a simplified version of YOLO-v2 with 24 layers, 18.95 GOP), the number of bits for activation ranges from 3 to 6 bits. The chosen step size is a power-of-two value so that the quantization requires only shift operations instead of multiplications. It should be noted that this quantization is a nonzero scheme (symmetric quantization). There is no zero value in the quantized output (i.e., the quantized value can be $\pm 1, \pm 3, \ldots$). The zero-center quantization scheme performs a bit worse than the former scheme. Moreover, it has an odd number of quantization levels (i.e., the quantized value can be $0, \pm 1, \pm 2, \ldots$). Thus, one quantized level is wasted. The experiments show that the symmetric quantization performs better than the zero-centered quantization. The quantized network with 1-bit weight and flexible low-bit activation reduces the model size by approximately 30×, and the activation size is reduced by 5.4×. Moreover, the recent FPGA generations have a rich on-chip SRAM resource. For example, 7 Series VC707 FPGA board has 1030 units of 36-Kb block RAMs (approximately 4.6 MB), and Virtex Ultrascale+ FPGA chip includes on-chip memory integration up to 500 Mb. Hence, this quantization enables storing the entire model of Sim-YOLO-v2 (or even much deeper networks) in block RAMs of the FPGA chip. As a result, the off-chip memory accesses are significantly reduced. Thus, the system performance is boosted, and power dissipation is reduced. Besides, the quantization also helps reduce the hardware cost (by removing expensive multiplications).

B. Data-Path Optimization for the Streaming Computation

Algorithm 1 explains the original loop computation for a convolutional layer. To run it efficiently on a dedicated hardware with limited resources, the loop computation needs to be optimized. To solve this problem, the loop reordering and tiling are proposed in [6], [11], [13], and [14]. Nevertheless, the output of the entire intermediate feature-maps from each layer is stored in the block RAMs. Moreover, designs in [6], [11], and [14] use doubled buffer to pipeline the computation. This scheme does not scale well when the CNN becomes deeper because it consumes a large number of block RAMs. For example, Tiny YOLO v2 has nine convolutional layers, and the number of feature-maps is 5.8 million. If each feature-map is quantized to 16 bit, it requires $5.8 \times 2 \times 2 = 23.2$ MB of block RAM for the doubled buffer. To reduce the size of block RAM, studies in [4], [12], [18], [20], and [25] save the intermediate data for each layer in off-chip memory. Hence, the frequent off-chip accesses slow down the computation, thereby consuming more power.

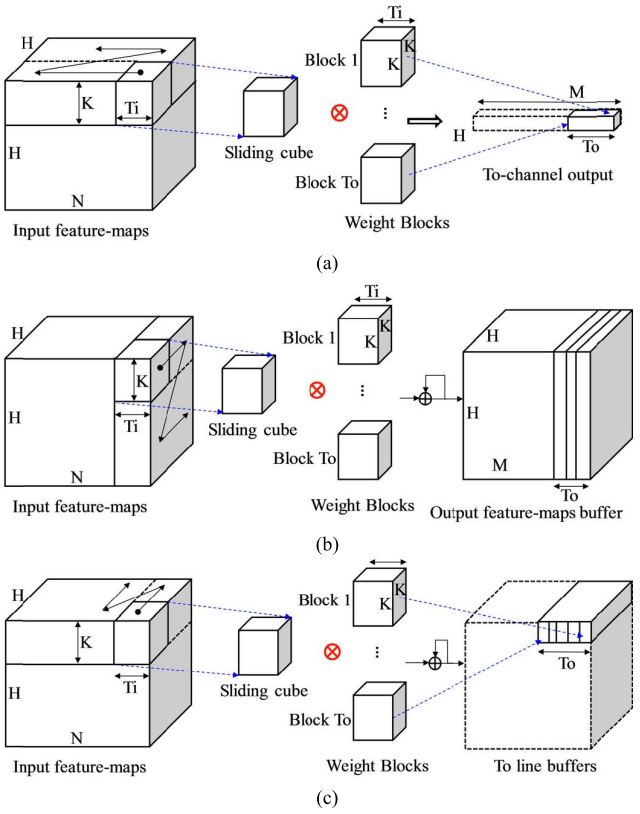

Fig. 2. Scheduling for streaming convolutional layer. (a) No weight reuse. (b) Fully weight reuse. (c) Proposed line-based weight reuse and input feature-maps fully reuse.

The target of this paper is to eliminate the off-chip accesses for intermediate data while minimizing the on-chip SRAM. To achieve this, there should be a data-path optimization to efficiently use the temporary data. As proposed in [4], this paper also uses the block-based computations to achieve the tradeoff between hardware resource and performance. However, scheduling the data movement efficiently in streaming convolutional computation is further investigated. Fig. 2 presents three scheduling schemes covering all the possibilities of weight reuse. The advantages and drawbacks of each strategy are analyzed in the following.

In the first strategy in Fig. 2(a), there is no weight reuse. The input sliding cube moves from the beginning toward the end of the channel dimension. Therefore, each sliding cube is convolved with a new weight block. All these values are accumulated to produce a final output. This scheme has the best locality of the partial sum. Thus, it does not require a temporary buffer for the accumulation. The input buffer size is $K \times N \times H \times Q_A$, where Q_A is the bit width of input feature-maps. To overlap the computation between layers, the number of buffer rows is increased from K to $(K+1)$. However, the weight model needs to be read H^2 times, which is inefficient for a large weight model.

The second scheme shown in Fig. 2(b) maximizes the weight reuse, which is implemented in [11] and [14]. Each weight is reused for the whole input channel (i.e., reuse H^2 times). At a time, T_i input planes are convolved with each of T_o weight blocks. The temporary accumulations are stored in an output buffer. The SRAM size of this doubled output buffer is $2 \times T_o \times H^2 \times Q_S$, where Q_S is the bit width of the accumulation before quantization. To produce the final T_o output feature-maps, the entire input feature-maps are accessed. Hence, to generate output feature-maps, the input feature-maps are repeatedly read M/T_o times. Because the entire input feature-maps are read multiple times, the temporary buffers must be large to store them. Moreover, to pipeline between layers, the buffer size should be doubled, which is $2 \times H^2 \times N \times Q_A$.

The scheme proposed by this paper is shown in Fig. 2(c). The streaming process is explained as follows. The input sliding cube (i.e., $K \times K \times T_i$ pixels) slides along the width of the input image, which is called *a row pass*. The input sliding cube is convolved with T_o weight blocks each time to produce T_o temporary output values. These weight blocks are reused for *a row pass*. These T_o computations are processed in parallel and saved in the line buffers, thereby creating T_o temporary output channels. The input sliding cube then shifts T_i channels toward the end of N-input channels. In the next *row pass*, new T_o weight blocks are fetched and convolved with the sliding cube. The convolutional outputs are accumulated with the corresponding values from line buffers and then saved in the line buffers. This operation repeats $N_i = N/T_i$ times until all the N input channels are computed. The values in the line buffers at this time are the final T_o output channels, which are then forwarded to the next layer. This computation is performed when all the M output channels are forwarded to the next layer. To finish the processing for one line, the entire weight of the model is accessed from the memory. For the next row computation, the sliding cube shifts down by one row and then repeats the above-mentioned process. Therefore, to process the whole input feature-maps, the weights are read H times. As the weights are stored in on-chip SRAM, and the weight prefetching can be applied to hide the latency, the weight accesses do not cause system degradation. The computation of a convolutional layer is performed when all lines are processed. It is noteworthy that each row of input feature-maps is reused K times. Regarding the hardware resource, the input buffer size for pipelining is $(K+1) \times N \times H \times Q_A$, and the temporary accumulation buffer size is $T_o \times H \times Q_S$.

The partial output from a layer is input directly to the next layer without going back to external memory. In addition, it is noteworthy that the partial output parameter T_o of a layer is the partial input parameter T_i of the next layer to minimize the cost of the control logics and block RAM banks. The parallelism parameters, such as T_i and T_o, are chosen to achieve the best tradeoff between the hardware cost and performance.

Table I summarizes the three scheduling schemes. It should be noted that Q_A is much smaller than Q_S after quantization. As analyzed previously, the first strategy (i.e., no weight reuse) is not efficient enough owing to the frequent weight read and no input reuse (i.e., no overlapped sliding windows). The second scheme (i.e., full weight reuse) has the best weight reuse among the three schemes, but it operates with framewise computation, which requires a large interlayer doubled buffer

TABLE I
COMPARISON OF THREE SCHEDULING SCHEMES FOR STREAMING PROCESSING

Features	No weight reuse	Fully weight reuse	Line-based weight reuse (proposed)
Input buffer size	$(K+1)\times N\times H\times Q_A$	$2\times H^2\times N\times Q_A$	$(K+1)\times N\times H\times Q_A$
Output buffer size	0	$2\times T_o\times H^2\times Q_S$	$T_o\times H\times Q_S$
Weight read (times)	H^2	1	H
Weight reuse (times)	1	H^2	H
Relative latency	1	H^2	H

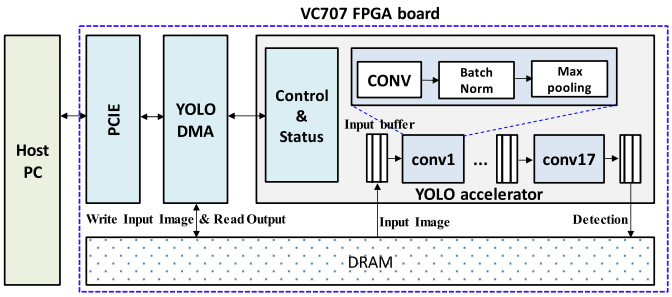

Fig. 3. Overview of the proposed streaming architecture.

for pipelining. In batch mode, the buffer and convolution kernel increase linearly as the batch size increases. The next layer can start only after the entire output of the previous layer is computed. On the other hand, the proposed scheme requires smaller buffers and causes a smaller delay between the layers (i.e., line delay). The weight prefetching can be used to hide the latency of the weight read. It is also noteworthy that the proposed scheme does not incur hardware resource overhead in the batch mode. Therefore, the proposed line-based weight reuse scheme outperforms the other schemes in terms of both hardware cost and performance.

IV. PROPOSED STREAMING ARCHITECTURE

A. Overview of Accelerator Architecture

Fig. 3 presents the overall block diagram of the proposed design, which is straightforward yet proven to be very efficient. The aggressively quantized model is stored entirely in block RAMs. The input to each layer is given line-by-line. Instead of a large doubled buffer as proposed in [6], [11], and [14], each layer requires only an additional line buffer to overlap the computation between layers. In the streaming design, the timing optimization of each layer is crucial. As each layer processes a different amount of computation at a different speed, there must be synchronization between layers for correct operation. This design uses the handshake mechanism to synchronize the operation of all layers. It is noteworthy that the streaming style completely eliminates the off-chip access for intermediate results, which was a limitation of the previous works in [4], [12], [18], [20], and [25]. The DRAM is accessed only for the input image and the final detection results.

B. Streaming Design of the Convolutional Layer

Fig. 4 shows the proposed architecture for a streaming convolutional layer. It is noteworthy that the kernel size (i.e., 3×3) can be changed easily, as the proposed design is fully parameterized. For better understanding, Fig. 4 is explained in conjunction with Fig. 2(c). In Fig. 4(a), the input buffer includes four lines of SRAM (for the case of 3×3 kernel). This additional buffer enables the overlapping of the computation of the current layer and the previous layer. A partial input from the previous layer is written to a line buffer while the data from other three line buffers are sent to T_o PEs for computation. As described in Section III, the input sliding cube slides along the width of the line buffers to send the data to each PE. In each PE, the T_i 3×3 input data are convolved with corresponding 1-bit 3×3 weight kernel, as shown in Fig. 4(b). Then, the 3×3 results are summed up using two-stage ternary adders (i.e., three-input adder). The results from each T_i kernel are input to a pipelined adder tree. The output data from the adder tree are saved temporarily to the buffers. In the next iteration, the next T_i input channels are sent to the PEs, the outputs from adder tree are summed up with the corresponding value in the buffers, and then, the resulting values are saved to the buffers. This iteration is performed when N input channels are sent to the PEs, and the values in the temporary buffer are the final convolutional values. The computation for each line is performed when all M output channels are calculated. This input line is no longer needed, and hence, it can be replaced by the next line. Thus, this feature-maps reuse scheme does not require writing back to DRAM, and consequently, the memory bandwidth can be significantly reduced. The outputs from convolutional computation are input to the batch normalization module that includes both quantization and leaky activation as described in Section III. The output from the batch normalization layers are the final quantized values that are concatenated and forwarded to the next layer. It should be noted that the number of bits to quantize the activations are less than the number of bits to represent the real value of activations. For example, the activations of a layer are quantized to 6 bit, and it needs seven bits to represent the real value of activations

$$\text{real\_value} = \frac{\text{step}}{2} * (2 * \text{quantized\_value} + 1). \quad (11)$$

This representation saves the 1/7th size of SRAM for input line buffers at the cost of one more addition. The SRAM size for the circular buffer is $4 \times C \times N \times Q_A$, where C is the width of the input image. In this streaming design, the partial inputs come to a layer in the same way, as they are read out for computation. Therefore, it is efficient to have a single deep SRAM to store multiple partial-line inputs sequentially, as shown in Fig. 5. This storage scheme utilizes the block RAMs of FPGA effectively. The depth and width of SRAM are $N/T_i \times C$ and $T_i \times Q_A$, respectively. T_i is chosen, such that the SRAM width is not too large, and thus, the number of block RAMs needed for a line buffer is minimum. T_i and T_o are the parallelism factors chosen to achieve the best tradeoff between the performance and hardware cost. It is also noteworthy that the number of bits for the input and output activations can be

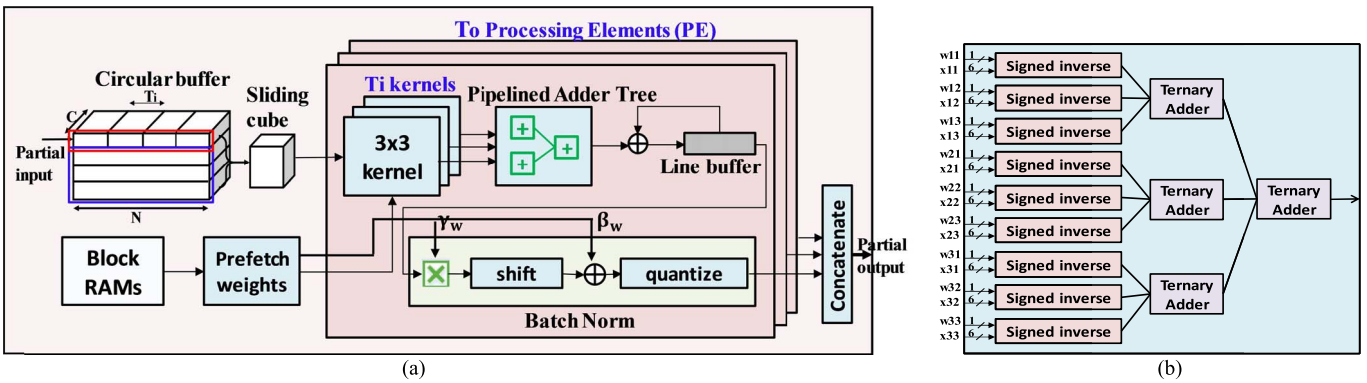

Fig. 4. Streaming design of a convolutional layer. (a) Architecture of a 3 × 3 convolutional layer. (b) Design for 1-bit weight and low-bit activation 3 × 3 Kernel.

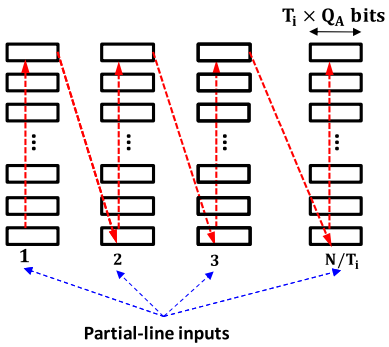

Fig. 5. Memory storage for a line buffer composed of multiple partial lines.

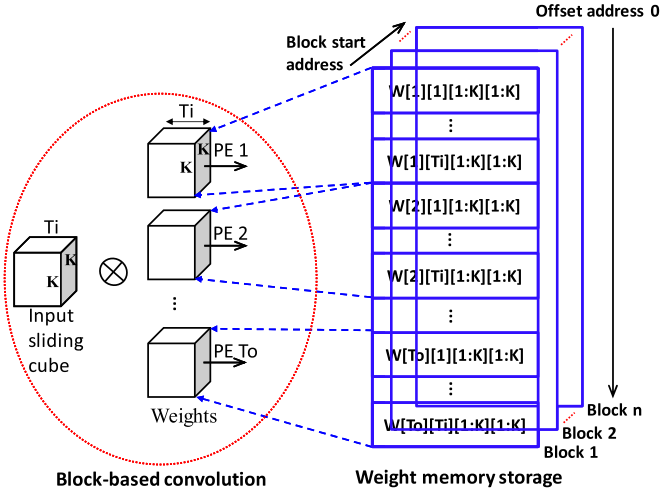

Fig. 6. Weight memory pattern for efficient weight prefetching.

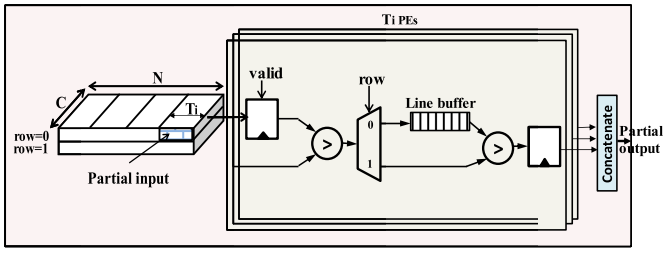

Fig. 7. Streaming architecture of a 2 × 2 partial max-pooling layer.

flexibly changed during the design phase to achieve the best performance.

For signed addition, this paper uses binary adder trees with pipelined registers added between each stage to achieve a high clock speed. The bit width of each pipeline stage is flexibly changed to minimize the hardware cost.

The computation of the convolutional layer has two basic tasks: parameter fetching and computation. To increase the speed of computation, the parameters are fetched beforehand. The size of the weight buffer is doubled, and it works as a ping-pong buffer. The period of each pipeline stage is equal to that of the longest task. This optimization increases the throughput of a layer significantly.

The design of the 1 × 1 convolutional layer is similar to that of the 3 × 3 layer. The only differences are: 1) the input buffer requires only two lines of memory and 2) there is no need of T_i 3 × 3 kernels; instead, the input is multiplied with binary weight and then input to adder tree.

For efficient prefetching of weight in order to speed up the convolutional computation, the weights in block RAMs need to be in a predefined pattern. Fig. 6 describes the proposed weight memory pattern to support partial convolution. Each weight block contains weights for T_o PEs, each of which computes the partial convolution for T_i input channels. Weight blocks are stored sequentially in memory according to the processing order. Weights from memory are loaded to the weight buffer block-by-block at consecutive addresses. This weight block is reused for each line of the T_i input channels. To produce one line of final output, the whole weight for that layer is loaded. Hence, each layer needs to load an image-width number of times to process the entire image. For layers with a large number of kernels, it does not incur many weight reloads from block RAMs, as the width of the input image is rather small.

C. Streaming Design of Max-Pooling Layer

The streaming design of the 2 × 2 partial input max-pooling layer is shown in Fig. 7. It is noteworthy that this layer requires one line of a buffer with a depth of half of the width of the input image. The partial input from the convolutional layer

is latched, and the latched input is then compared with the input. If the current row is even (assume that row counts from zero), the comparison results are stored in the line buffer. Otherwise (i.e., the current row is odd), the comparison results are compared one more time with the value in the corresponding address in the buffer. The final comparison results are concatenated to produce the T_i-channel outputs.

D. Resource-Aware Parallelism

The aforementioned parameters T_i and T_o are chosen to achieve the best performance of the streaming architecture in terms of both throughput and hardware cost. As shown in Fig. 2, the parallelism factor is $T_i \times T_o$. Larger values of T_i and T_o can achieve more throughput in the convolution layer. However, as the network is very deep, each big layer contributes to the hardware resource substantially. Therefore, T_i and T_o should be carefully selected. For a convolutional layer that computes N-channel inputs to produce M-channel outputs, the number of repetitions of the partial computation and accumulation to produce a final output is $N_i = N/T_i$. In addition, T_o output channels are computed in parallel. Thus, the number of repetitions required to compute all output channels is $M_o = M/T_o$. It is noteworthy that T_i and T_o are the divisors of N and M, respectively, to avoid a complicated design and underutilization of the computation kernels.

The delay of each block-based computation for each kernel size (i.e., 1×1 and 3×3) is given as follows:

$$t_{1\times1} = 8 + \log(T_i) \tag{12}$$

$$t_{3\times3} = 10 + \log(T_i). \tag{13}$$

Here, the specific numbers, such as 8 and 10, are present owing to the certain pipeline stages in the convolutional kernel.

Hence, the computation time per line for each layer (depends on the kernel size) is given as follows:

$$T_{1\times1} = (8 + \log(T_i)) \times C \times \frac{N}{T_i} \times \frac{M}{T_o} \tag{14}$$

$$T_{3\times3} = (10 + \log(T_i)) \times C \times \frac{N}{T_i} \times \frac{M}{T_o} \tag{15}$$

where C is the width of the input image. It should be noted that the computation time varies for each group of a layer. For example, owing to the 2×2 max-pooling layer, two lines are computed in CONV1 to produce one line for CONV2. Similarly, CONV2 needs to compute two lines to produce one line for CONV3, and so on. To achieve the maximum efficiency of pipeline processing, the computation time for each layer should be similar. It is noteworthy that the T_i parameter of the current layer is the T_o value of the previous layer. Therefore, the parallelism factors for two consecutive convolutional layers (denoted in layer 1 and 2 in the following) present in different groups should satisfy the following condition:

$$2 \times (10 + \log(T_{i1})) \times C_1 \times \frac{N_1}{T_{i1}} \times \frac{M_1}{T_{o1}} \approx (10 + \log(T_{i2})) \times C_2 \times \frac{N_2}{T_{i2}} \times \frac{M_2}{T_{o2}} \tag{16}$$

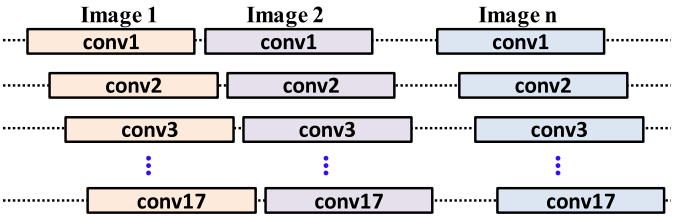

Fig. 8. Batch processing.

where $M_1 = 2\ N_1 = N_2 = 0.5 \times M_2, C_1 = 2\ C_2$, and $T_{o1} = T_{i2}$. Hence, the condition is simplified as

$$(10 + \log(T_{i1})) \times \frac{1}{T_{i1}} \approx (10 + \log(T_{o1})) \times \frac{1}{T_{o2}}. \tag{17}$$

The convolutional layers in the same group process the input images of the same size, $N_1 = M_2$, and $M_1 = N_2$. Therefore, the condition is simplified as follows:

$$(10 + \log(T_{i1})) \times \frac{1}{T_{i1}} \approx (8 + \log(T_{o1})) \times \frac{1}{T_{o2}}. \tag{18}$$

Multiplications are mapped to DSPs in FPGA. Hence, DSPs are used only for the batch normalization module. Therefore, the number of DSPs in each layer except the last layer is only T_o. As the last layer requires multiplications of weights and input feature-maps, the number of DSPs is $T_i \times T_o + T_o$. Given the number of available DSPs, the parallelism parameters of all layers must satisfy the following condition:

$$T_o^{(n)} \times T_o^{(n-1)} + \sum_{i=1}^{n} T_o^{(i)} \leq \text{DSPs}_{\text{available}} \tag{19}$$

where n is the number of layers and $T_o^{(i)}$ is the T_o value of nth layer. For each PE in Fig. 4, there are $4 \times T_i$ ternary adders and $T_i + 1$ binary adders. Therefore, in the convolutional kernel in each layer, there are $T_o \times (5 \times T_i + 1)$ adders. It is noteworthy that the ternary adder is implemented efficiently using the LUT6 and carry chain in the same slice for the Virtex-7 FPGA chip, thereby saving the area and achieving high frequency [26]. Consequently, it requires only two cycles to reduce nine inputs to a single output while guaranteeing a high speed of 200 MHz.

E. Batch Processing

As shown in Fig. 8, the streaming architecture with pipelining enables the ability to run computation in batch mode to fully utilize the pipeline processing. Each layer is delayed by 1-to-4 input lines from the previous layer. Because the network is very deep, the delay from the first layer to the last layer becomes large. If a single frame is processed at a time, the first layers are underutilized for a definite period. The idea of batch processing is that, while the layers at the end of the networks run the first image, the first layers, which finish running the first image, can start running for the second image to utilize the idle period. The batch processing mode increases the throughput significantly.

TABLE II
ACCURACY (MAP) OF QUANTIZED TINY YOLO CNN (W: WEIGHT AND A: ACTIVATION)

Quantization	Accuracy (%)	Weight size (MB)	Activation size (MB)	Multiplication for convolution
Baseline (32-b W, 32-b A)	53.96	60.53	22.0	Yes
1-b W, 6-b A	51.44	2.01	4.1	No
1-b W, 4-b A	49.12	2.01	2.73	No
1-b W, 3-b A	45.0	2.01	2.05	No

To define the processing time and speedup, the T, D, L, and n are defined as follows. T is the frame processing time for the last layer, D is the delay from the first layer to the last layer, L is the latency between the last layers of two consecutive frames, and n is the batch-size. The processing time of a frame for normal mode is $T + D$. The processing time for a batch size of n is $D + n \times T + (n - 1) \times L$. The speedup can be derived as follows:

$$\text{speedup} = \frac{T + D}{T + L + (D - L)/n}. \tag{20}$$

Larger the batch-size results in bigger throughput. It is noteworthy that the accelerator with batch-mode support does not require any additional hardware resource owing to the pipelined processing scheme.

V. EXPERIMENTAL RESULTS

A. Low-Bit Quantization

The quantized network is retrained using Darknet [27] deep learning framework. The YOLO CNN is trained using PASCAL VOC 2007+2012 data set and tested using PASCAL VOC 2007. Table II shows the accuracy of the quantized Tiny YOLO-v2. The quantized network with 1-bit weight and 6-bit activation (i.e., 1-b W and 6-b A) incurs an accuracy loss of approximately 2.5% compared with the full-precision network. The model size is reduced by 30×, and the activation size is reduced by 5.4×.

To justify the efficiency of the proposed streaming architecture in computing the deeper networks, the quantization of a deeper network is performed. This paper selects a simplified version of YOLO-v2 as a baseline for the quantization. This network, the so-called Sim-YOLO-v2, inherits the structure of YOLO-v2 except the last layers for multiscale detection [3]. As the research in [3] points out that the pass-through layers, which fetch features from an earlier layer to the final outputs, enable a modest increase in performance (i.e., 1%), these pass-through layers are removed to save the computation budget. Hence, the Sim-YOLO v2 contains 19 convolution layers and 5 max-pooling layers. Its architecture is the same as Darknet-19 network in [3]. Table III presents the accuracy of the quantized networks in YOLO-v2 and Sim-YOLO-v2 compared with the full-precision network.

The binary version of these above-mentioned networks attains accuracy comparable with that of the full-precision networks, meanwhile, saving a significant amount of computation. These quantized networks are used for the verification of the streaming architecture. It is noteworthy that the deeper networks perform better in terms of quantization. The binary network numbered (3), which has 17 convolution layers and 5 max-pooling layers [removed 2 CONV layers from Sim-YOLO-v2 (2)], achieves an accuracy of 64.16%, which is 12.78% higher than that of the quantized Tiny YOLO-v2 in Table II. In addition, its binary weight size is only 1.88 MB, and therefore, it can be stored entirely in block RAMs in an FPGA. Even though it is three times more complex than Tiny YOLO-v2, a high throughput is achieved by virtue of its streaming architecture. To provide detailed information about the network topology, the architecture of the implemented network is described in Table IV.

TABLE III
ACCURACY (MAP) OF QUANTIZED YOLO-V2 AND SIM-YOLO-V2

Networks	Quantization	Accuracy (%)	Weight size (MB)	Complexity (GOP)
(1) YOLO -v2	Full precision	75.88	258	34.9
	1-b W, 32-b A	71.56	8.1	17.45
	1-b W, 6-b A	71.11	8.1	17.45
(2) Sim-YOLO-v2	Full precision	72.0	79.74	18.95
	1-b W, 32-b A	66.99	2.54	9.48
	1-b W, 6-b A	65.76	2.54	9.48
(3) Sim-YOLO-v2 FPGA	Full precision	66.79	58.28	17.18
	1-b W, 32-b A	64.95	1.88	8.59
	1-b W, 6-b A	65.07	1.88	8.59
	1-b W, 4-to-6-b A	64.16	1.88	8.59

TABLE IV
YOLO-V2 NETWORK ARCHITECTURE AND PARALLELISM FACTORS FOR EACH LAYER

Layer	Type	Size / Stride	Filter number	Input Size	Output Size	Out bit width	PF*** (Ti, To)
0	C*	3×3 / 1	32	416×416×3	416×416×32	6	(3, 32)
1	M**	2×2 / 2		416×416×32	208×208×32	6	(8, 8)
2	C*	3×3 / 1	64	208×208×32	208×208×64	6	(8, 8)
3	M**	2×2 / 2		208×208×64	104×104×64	6	N/A
4	C*	3×3 / 1	128	104×104×64	104×104×128	6	(8, 8)
5	C*	1×1 / 1	64	104×104×128	104×104×64	6	(8, 8)
6	C*	3×3 / 1	128	104×104×64	104×104×128	6	(8, 8)
7	M**	2×2 / 2		104×104×128	52×52×128	6	N/A
8	C*	3×3 / 1	256	52×52×128	52×52×256	6	(8, 8)
9	C*	1×1 / 1	128	52×52×256	52×52×128	6	(8, 8)
10	C*	3×3 / 1	256	52×52×128	52×52×256	6	(8, 16)
11	M**	2×2 / 2		52×52×256	26×26×256	6	N/A
12	C*	3×3 / 1	512	26×26×256	26×26×512	6	(16, 8)
13	C*	1×1 / 1	256	26×26×512	26×26×256	6	(8, 16)
14	C*	3×3 / 1	512	26×26×256	26×26×512	6	(16, 8)
15	C*	1×1 / 1	256	26×26×512	26×26×256	6	(8, 16)
16	C*	3×3 / 1	512	26×26×256	26×26×512	4	(16, 8)
17	M**	2×2 / 2		26×26×512	13×13×512	4	N/A
18	C*	3×3 / 1	1024	13×13×512	13×13×1024	6	(8, 16)
19	C*	1×1 / 1	512	13×13×1024	13×13×512	4	(16, 8)
20	C*	3×3 / 1	1024	13×13×512	13×13×1024	6	(8, 16)
21	C*	1×1 / 1	125	13×13×1024	13×13×125	16	(16, 5)

Note: C*=Convolution, M**=Maxpool, PF***= Parallelism Factors

B. Implementation Result

The analysis in Section IV-D defines the guidelines to achieve balanced pipelining. The parallelism factors for each

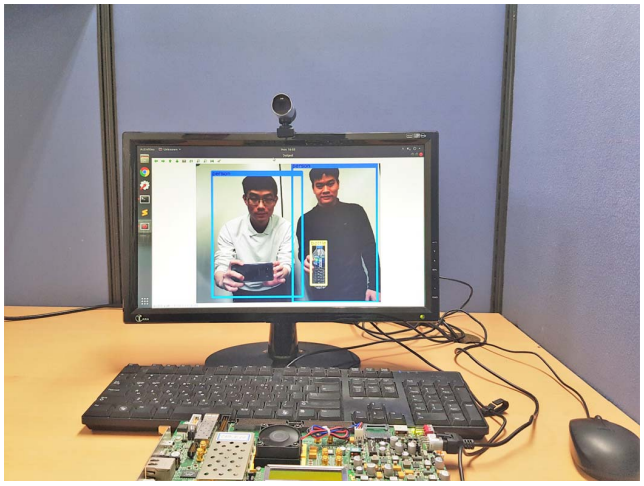

Fig. 9. Live demo of the proposed work on the VC707 FPGA board.

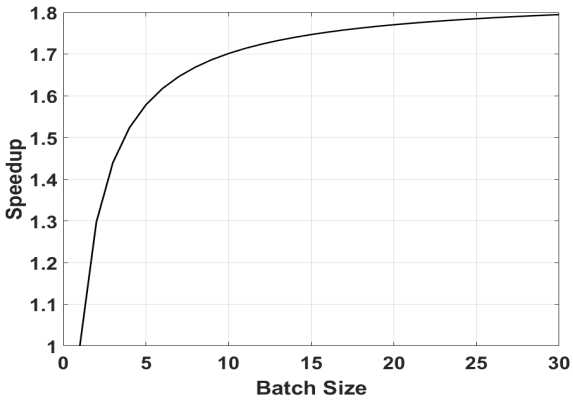

Fig. 10. Throughput improvement with batch processing.

layer are empirically chosen according to these guidelines. These factors satisfy the FPGA resource constraints and the target operating frequency (i.e., 200 MHz). The detailed parallelism factors for each layer are listed in Table IV.

The framework in Fig. 3 is used to verify the operation of the proposed design. Input images and control commands from the host PC are sent to the accelerator through peripheral component interconnect (PCI) Express port. After the computation, the detection results are sent back to the host PC for postprocessing. Fig. 9 demonstrates the real-time object detection of the YOLO network using the proposed design on VC707 Evaluation board. The detector can detect 20 objects in the PASCAL VOC data sets at 30 frames/s.

The batch processing increases the throughput significantly, as it improves the utilization of each convolutional layer. The experiments at 200 MHz show that $T = 7.5$ ms, $D = 8.975$ ms, and $L = 1.43$ ms. According to (20), the dependence of the speedup on the batch size is shown in Fig. 10. The speedup is almost saturated at 1.8 with a batch size of 30. Table V shows the performance of the proposed hardware design with (batch size = 30) and without batch processing. Owing to the intermediate data reuse, the DRAM bandwidth is below 100 MB/s for batch processing mode. A low-cost single-data-rate SDRAM is sufficient for high performance. The utilization of DSPs is below 10%. Most DSPs are consumed by the last layer that requires multiplications between weights and input activations. With batch processing, the throughput is observed to be as efficient as 1.877 TOPS with a power consumption of 18.29 W.

TABLE V

IMPLEMENTATION RESULTS OF THE PROPOSED DESIGN WITH AND WITHOUT BATCH PROCESSING

Features	Performance w/o batch	Performance w/ batch
Device	Virtex-7 VC707 FPGA	
Operating frequency	200 MHz	
Block RAMs (18 Kb)	1144 (55.5%)	
DSPs	272 (9.7%)	
LUTs – FFs	155.2K (51.1%) – 115K (18.9%)	
mAP	64.16%	
DRAM bandwidth	47.2 MB/s	84.96 MB/s
Frame rate (416 × 416)	60.72 fps	109.3 fps
Throughput	1043 GOPS	1877 GOPS
Power	11.11 W	18.29 W

Table VI shows the comparison of the proposed design with the previous works about the YOLO hardware implementation. The performance of the proposed design running the Tiny YOLO-v2 significantly outperforms the Tincy YOLO presented in [21] in terms of throughput, accuracy, and power efficiency. Compared with the performance of Sim-YOLO-v2 running on GTX Titan X GPU in [3], the proposed design with batch processing exhibits higher throughput (1.24 times) and much better efficiency in power consumption (11.54 times). The design in [22] combines the binary CNN (for feature extraction) with parallel SVM (for refined detection) to achieve an accuracy of 3.46% higher than the proposed design. However, the number of classes used for accurate calculation for the work in [22] is not clear. The proposed design achieves 3.1 times higher throughput (with a higher resolution) at 1.5 times slower frequency and 2.68 times better efficiency (GOPS per LUTs). It is notable that deeper network can result in the more efficient proposed architecture.

The comparison between the proposed design and other recent works on CNN hardware implementations is presented in Table VII. For a fair comparison, both small-scale networks in [19] and [30] and large-scale deep networks in [18], [20], and [29] are selected. In the case of low-precision design for small-scale networks, small input size (32×32) and simple network structure result in high throughput and efficient resource utilization. Therefore, most of the previous FPGA designs for binary neural networks (BNNs) are validated using just tiny data set, such as Cifar-10 or MNIST [19], [20], [30], [31]. To legitimately compare to the previous works, the convolutional layers of VGG-16, in this paper, are quantized using 1-bit weight and 2-bit activation because this paper is not optimized for the fully binarized network. The low-precision model of VGG-16 runs on the proposed FPGA design to estimate the accuracy performance. Consequently, as shown in Table VII, the proposed design achieves a throughput of 4420 GOPS with a batch size of 32. In the case of [19], despite the small size of CNN (i.e., 0.1125 GOP) and

TABLE VI
COMPARISON OF THE PROPOSED DESIGN WITH THE PREVIOUS WORKS FOR YOLO CNN HARDWARE

	Sim-YOLO-v2 on GPU [3]	Tincy YOLO [21]	Lightweight YOLO-v2 [22]	This work (Tiny YOLO-v2)	This work (Sim-YOLO-v2)
Platform	GTX Titan X (16nm)	Zynq Ultrascale+ (16 nm)	Zynq Ultrascale+ (16 nm)	Virtex-7 VC707 (28 nm)	Virtex-7 VC707 (28 nm)
Frequency	1 GHz	N/A	300 MHz	200 MHz	200 MHz
BRAMs (18 Kb)	N/A	N/A	1706	1026	1144
DSPs	N/A	N/A	377	168	272
LUTs - FFs	N/A	N/A	135K – 370K	86K – 60K	155K – 115K
CNN Size (GOP)	17.18	4.5	14.97	6.97	17.18
Precision (W, A)(**)	(32, 32)	(1, 3)	(1-32, 1-32)	(1, 6)	(1, 6)
Image Size	416×416	416×416	224×224	416×416	416×416
Frame rate	88	16	40.81	66.56	109.3
Accuracy (mAP) (%)	66.79	48.5	67.6	51.38	64.16
Throughput (GOPS)	1512	72	610.9	464.7	1877
Efficiency (GOPS/kLUT)	N/A	N/A	4.52	5.40	12.11
Power (W)	170	6	N/A	8.7	18.29
Power efficiency (GOP/s/W)	8.89	12	N/A	53.29	102.62

Note: (**): W: Weight, A: Activation

TABLE VII
COMPARISON OF THE PROPOSED DESIGN WITH THE PREVIOUS WORKS FOR OTHER CNN HARDWARE

	FPGA'17 [19]	HiPEAC'17 [30]	Neurocomputing [20]	FFCM'17 [18]	TVLSI'18 [29]	This work
Platform	Zynq XC7Z045	Kintex Ultrascale XCKU115	Stratix-V 5SGSD8	Zynq Ultrascale+	Intel Arria 10 GX 1150	Virtex-7 VC707
Frequency (MHz)	200	125	150	200	200	200
BRAMs (18Kb)	186	1814	2210 (*)	1824	2232 (*)	1214
DSPs	N/A	N/A	384	2520	1518	272
LUTs	46.3K	392.9K	230.9 (*)	600K	138K (*)	104.7K
FFs	N/A	348K	N/A	N/A	N/A	140.1K
CNN Size (GOP)	0.1125	1.2	1.45	5 layers of VGG	30.95	30.74
Precision (W, A)	(1, 1)	(1, 1)	(1, 1)	(16, 16)	(16, 16)	(1, 2)
Image Size	32×32	32×32	224×224	224×224	224×224	224×224
Throughput (GOPS)	2463.8	14814	1964.0	2940.7	715.9	4420
Efficiency (GOPS/kLUT)	53.2	37.7	8.51	4.90	5.19	42.22
Power (W)	11.7	N/A	26.2	23.6	N/A	14.72
Power efficiency (GOP/s/W)	210.58	N/A	74.96	124.6	N/A	300.27

Note: (*) for Intel FPGA: Block RAM (20Kb), Logic cells (ALM)

validation by small images, the power efficiency is much lower than the proposed design. In the case of [30], the proposed design outperforms [30] in terms of efficiency (i.e., throughput per LUTs), despite [30] achieving the highest throughput owing to its BNN structure. However, it should be noted that the design in [30] is operated on the Ultrascale's family FPGA (20 nm), which is more than two times larger than the FPGA board used in this paper, and the BNN structure in [30] cannot support the large input size owing to its simple structure. It proves that the proposed work outperforms other short bit-width CNN implementations.

The next comparison is for the large-scale deep network. Liang *et al.* [20] run AlexNet on their own design. The average throughput is reported as 1964 GOPS, which is 2.25 times lower than the throughput in this paper. Owing to the frequent off-chip accesses, the first layer (i.e., 210 GOPS) is the bottleneck of their design. The research in [18] does not count the first layer of the first group for the average throughput owing to the similar problems. In this layer (183 MOPs), to achieve the stated throughput of 2735 GOPS, the corresponding frame rate is 14 754 frames/s. It is assumed that the output feature-maps use 8-bit data. The average memory bandwidth is 47 GB/s, which is much higher than the maximum bandwidth (19.2 GB/s) that the ZCU102 FPGA board can provide [32], whereas the proposed design requires a very small memory bandwidth (i.e., 85 MB/s) owing to the streaming design and data path optimization. Thus, it achieves much higher performance in terms of both throughput and power efficiency. In conclusion, considering the tradeoff between the hardware resource, CNN size, throughput, and power, the proposed design outperforms the previous works irrespective of the precision.

VI. Conclusion

This paper presents a high-performance hardware-efficient streaming architecture for real-time object detection by quantizing the network and optimizing the data path to eliminate the off-chip access for intermediate data. With batch processing of streaming, the proposed design achieves a throughput of 1.877 TOPS without increasing the hardware cost, which outperforms the most previous designs. It is worth mentioning that the deeper the network is, the more efficient its architecture. Therefore, the proposed design is expected to significantly contribute to the real-time object detection.

References

[1] W. Liu *et al.* (Dec. 2015). "SSD: Single shot multibox detector." [Online]. Available: https://arxiv.org/abs/1512.02325

[2] S. Ren, K. He, R. Girshick, and J. Sun. (Jun. 2015). "Faster R-CNN: Towards real-time object detection with region proposal networks." [Online]. Available: https://arxiv.org/abs/1506.01497

[3] J. Redmon and A. Farhadi. (Dec. 2016). "YOLO9000: Better, faster, stronger." [Online]. Available: https://arxiv.org/abs/1612.08242

[4] 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*, Feb. 2015, pp. 161–170.

[5] H. Sharma *et al.*, "From high-level deep neural models to FPGAs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2016, Art. no. 17.

[6] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2016, Art. no. 22.

[7] D. T. Nguyen, H. Kim, H.-J. Lee, and I.-J. Chang, "An approximate memory architecture for a reduction of refresh power consumption in deep learning applications," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[8] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. (Mar. 2016). "XNOR-Net: ImageNet classification using binary convolutional neural networks." [Online]. Available: https://arxiv.org/abs/1603.05279

[9] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. (Feb. 2016). "Incremental network quantization: Towards lossless CNNs with low-precision weights." [Online]. Available: https://arxiv.org/abs/1702.03044

[10] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. (Feb. 2016). "Binarized neural networks: Training deep neural networks with weights and activations constrained to $+1$ or -1," [Online]. Available: https://arxiv.org/abs/1602.02830

[11] F. Sun *et al.*, "A high-performance accelerator for large-scale convolutional neural networks," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl. Int. Conf. Ubiquitous Comput. Commun.*, Dec. 2017, pp. 622–629.

[12] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 535–547.

[13] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proc. 54th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2017, pp. 1–6.

[14] 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.*, Aug./Sep. 2016, pp. 1–9.

[15] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks," in *Proc. 27th Int. Conf. Field Program. Log. Appl.*, Sep. 2017, pp. 1–8.

[16] S. Winograd, *Arithmetic Complexity of Computation*, vol. 33. Philadelphia, PA, USA: Siam, 1980.

[17] 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*, Feb. 2017, pp. 55–64.

[18] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr./May 2017, pp. 101–108.

[19] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. IEEE Int. Symp. Field-Program. Custom Comput. Mach.*, Feb. 2017, pp. 65–74.

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

[21] T. B. Preußer, G. Gambardella, N. Fraser, and M. Blott, "Inference of quantized neural networks on heterogeneous all-programmable devices," in *Proc. Des., Automat. Test Eur. Conf. Exhib.*, Mar. 2018, pp. 833–838.

[22] 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.

[23] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Int. Conf. Mach. Learn.*, Jun. 2015, pp. 448–456.

[24] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proc. Int. Conf. Mach. Learn.*, Jun. 2016, pp. 2849–2858.

[25] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[26] *7 Series DSP48E1 Slice*, Xilinx, San Jose, CA, USA, 2018.

[27] *Darknet Deep Learning Framework*. Accessed: Oct. 10, 2018. [Online]. Available: https://github.com/pjreddie/darknet

[28] K. Simonyan and A. Zisserman. (Sep. 2015). "Very deep convolutional networks for large-scale image recognition." [Online]. Available: https://arxiv.org/abs/1409.1556

[29] 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.

[30] N. J. Fraser *et al.* (Jan. 2017). "Scaling binarized neural networks on reconfigurable logic." [Online]. Available: https://arxiv.org/abs/1701.03400

[31] Y. Li, Z. Liu, K. Xu, H. Yu, and F. Ren, "A GPU-outperforming FPGA accelerator architecture for binary convolutional neural networks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 14, no. 2, Jul. 2018, Art. no. 18.

[32] *Zynq Ultrascale+ MPSOC Datasheet*, Xilinx, San Jose, CA, USA, 2018.

Duy Thanh Nguyen received the B.S. degree in electrical engineering from the Hanoi University of Science and Technology, Hanoi, Vietnam, and the M.S. degree in electrical and computer engineering from Seoul National University, Seoul, South Korea, in 2011 and 2014, respectively, where he is currently working toward the Ph.D. degree in electrical and computer engineering.

His current research interests include computer architecture, memory systems, and SoC design for computer vision applications.

Tuan Nghia Nguyen received the B.S. degree in electronics and telecommunications from the Hanoi University of Science and Technology, Hanoi, Vietnam. He is currently working toward the M.S. degree in electrical and computer engineering at Seoul National University, Seoul, South Korea.

His current research interests include computer vision, deep learning applications, and computer architecture.

Hyun Kim (M'12) received the B.S., M.S., and Ph.D. degrees in electrical engineering and computer science from Seoul National University, Seoul, South Korea, in 2009, 2011, and 2015, respectively.

From 2015 to 2018, he was a BK Assistant Professor with the BK21 Creative Research Engineer Development for IT, Seoul National University. In 2018, he joined the Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul, where he is currently an Assistant Professor. His current research interests include algorithm, computer architecture, memory, and SoC design for low-complexity multimedia applications and deep neural networks.

Hyuk-Jae Lee (M'04) received the B.S. and M.S. degrees in electronics engineering from Seoul National University, Seoul, South Korea, in 1987 and 1989, respectively, and the Ph.D. degree in electrical and computer engineering from Purdue University, West Lafayette, IN, USA, in 1996.

From 1996 to 1998, he was with the faculty of the Department of Computer Science, Louisiana Tech University, Ruston, LS, USA. From 1998 to 2001, he was a Senior Component Design Engineer with the Server and Workstation Chipset Division, Intel Corporation, Hillsboro, OR, USA. In 2001, he joined the School of Electrical Engineering and Computer Science, Seoul National University, where he is currently a Professor. He is a Founder of Mamurian Design, Inc., Seoul, a fabless SoC design house for multimedia applications. His current research interests include computer architecture and SoC design for multimedia applications.