LETTER

HFOD: A hardware-friendly quantization method for object detection on embedded FPGAs

Fei Zhang[1], Ziyang Gao[2], Jiaming Huang[2], Peining Zhen[2], Hai-Bao Chen[2, a)], and Jie Yan[1]

Abstract There are two research hotspots for improving performance and energy efficiency of the inference phase of Convolutional neural networks (CNNs). The first one is model compression techniques while the second is hardware accelerator implementation. To overcome the incompatibility of algorithm optimization and hardware design, this paper proposes HFOD, a hardware-friendly quantization method for object detection on embedded FPGAs. We adopt a channel-wise, uniform quantization method to compress YOLOv3-Tiny model. Weights are quantized to 2-bit while activations are quantized to 8-bit for all convolutional layers. To achieve highly-efficient implementations on FPGA, we add batch normalization (BN) layer fusion in quantization process. A flexible, efficient convolutional unit structure is designed to utilize hardware-friendly quantization, and the accelerator is developed based on an automatic synthesis template. Experimental results show that the resources of FPGA in the proposed accelerator design contribute more computing performance compared with regular 8-bit/16-bit fixed point quantization. The model size and the activation size of the proposed network with 2-bit weights and 8-bit activations can be effectively reduced by 16× and 4× with a small amount of accuracy loss, respectively. Our HFOD method can achieve 90.6 GOPS on PYNQ-Z2 at 150 MHz, which is 1.4× faster and 2× better in power efficiency than peer FPGA implementation on the same platform.

Keywords: convolutional neural networks, quantization, highly-efficient implementation
Classification: Integrated circuits

1. Introduction

Convolution neural networks have been widely adopted to address challenging tasks in computer vision like object detection. Among them, many CNN models have been proposed, such as faster R-convolution neural network (Faster R-CNN) [1], single-shot-multibox-detection (SSD) [2], and you-only-look-once (YOLO) [3, 4]. As a single neural network predicting bounding boxes and class probabilities simultaneously, YOLO performs a better tradeoff between the accuracy and speed when deployed on GPU [5].

Considering the development cycle of FPGA implementation, there has been a steady improvement in FPGA design automation tools over past years [6]. High-level synthesis (HLS) tools like LegUp [7] and Xilinx Vivado HLS [8] enable designers to implement hardware design using a high-level programming language, then automatically compile the code into register-transfer level (RTL) design. Recent tools such as Xilinx SDSoC [9] and Intel FPGA SDK for OpenCL [10] can generate the hardware-software interface automatically so the cost and effort of software-hardware co-design can be relatively reduced. These tools and functions make it possible that time-to-market and aforementioned innovation gap on new accelerator designs can be reduced [11, 12].

In this paper, we propose a hardware-friendly quantization method for object detection on embedded FPGAs, named HFOD. YOLOv3-Tiny network is used to evaluate the performance of the proposed HFOD accelerator both in hardware performance and detection accuracy. The network is quantized and retrained using 2-bit weight, 8-bit bias and 8-bit activation. A HLS design methodology is adopted for efficient development of our FPGA-based YOLO accelerator. Previous HLS research has fully used loop reordering, unrolling, and local double buffering for CNNs [13]. Our FPGA implementation utilizes these optimizations,and uses a low-precision convolutional unit to achieve high throughput across different convolutional layers. The proposed accelerator is implemented on an embedded FPGA development board (PYNQ-Z2) and promising promotion is presented compared with other existing FPGA accelerators.

2. Hardware-friendly low bit quantizaiton

2.1 Batch norm transformation

A YOLO CNN is usually composed of several basic layers: convolution, max-pooling, up-sampling normalization and route [14]. Convolutional (conv) layer is used to extract higher features from input images [15]. The input images, comprising N channels, is convolved with M N-channel $K \times K$ weight filters. Max-Pooling layer maps every input fmap to an output fmap whose every pixel is the max value of a $K \times K$ window of input pixels. Up-sampling layer is introduced in YOLOv3 [16] to improve the detection accuracy on objects of different size. The original purpose of up-sampling is to enlarge the input image so that it can be displayed on a higher resolution view. Batch normalization (BN) [17] has been widely used in training modern CNN.

After batch normalization, the output of the previous convolutional layer is normalized to reduce the internal covariate shift, and its distribution has zero mean and unit variance [18]. The BN transformation is given as follows:

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

where x and y are input and output, respectively, $\mu^{(i)}$ and

[1] Institute of Aeronautics and Astronautics, Northwestern Polytechnic University, Xi'an, 710129, China.
[2] Department of Electrical and Computer Engineering, Shanghai Jiao Tong University, Shanghai, 200240, China.
a) haibaochen@sjtu.edu.cn

DOI: 10.1587/elex.19.20220067
Received February 3, 2022
Accepted March 15, 2022
Publicized March 24, 2022
Copyedited April 25, 2022

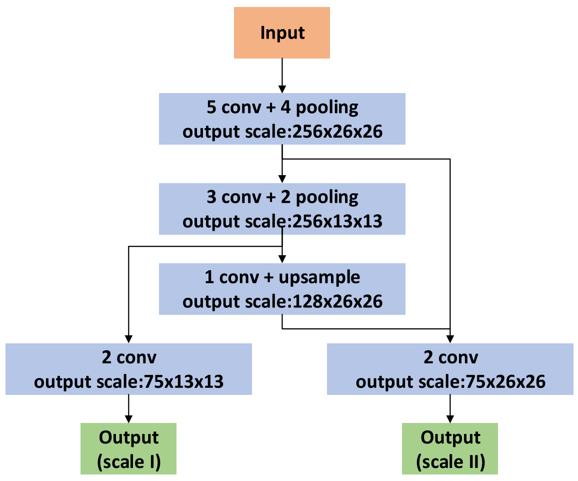

Fig. 1 Dataflow of YOLOv3-Tiny.

$[\varepsilon^{(i)}]^2$ are the channel-wise mean and variance of input x over a mini-batch, respectively. $\gamma^{(i)}$ and $\beta^{(i)}$ are channel-wise trainable parameters, and ϵ is to avoid round-off problems. In inference stage, all parameters are fixed, and these operations will cause significant computing cost and extra access to off-chip memory when directly implemented on FPGA without further optimization [19, 20].

Algorithm 1: Transformed Convolutional Layer

```
Input: $In[N][(R-1)*S+K][(C-1)*S+K]$ // in fmaps,
       $W[M][N][K][K]$ // weights,
       $In_{buf}[Tn][(Tr-1)*S+K][(Tc-1)*S+K]$,
       $Out_{buf}[Tm][Tr][Tc]$ $W_{buf}[Tm][Tn][K][K]$
Output: $Out[M][R][C]$ // out fmaps
for($r=0; r<R; r+=Tr$)
 for($c=0; c<C; c+=Tc$)
  for($m=0; m<M; m+=Tm$){
   for($n=0; n<N; n+=Tn$){
    $In_{buf}$=In[n][r*S : (r+Tr-1)*S+K][c*S : (c+Tc-1)*S+K]
    $W_{buf}$=W[m : m+Tm][n : n+Tn][0 : K][0 : K]
    for($i=0; i<K; i++$)
     for($j=0; j<K; j++$)
      for($tr=0; tr<min(Tr, R-r); tr++$)
       for($tc=0; tc<min(Tc, C-c); tc++$)
        #PIPELINE
        for($tm=0; tm<min(Tm, M-m); tm++$) #UNROLL
          for($tn=0; tn<min(Tn, N-n); tn++$) #UNROLL
           $Out_{buf}$+=$W_{buf} * In_{buf}$}
   $Out[m:m+Tm][r:r+Tr][c:c+Tc]=Out_{buf}$
  }
```

YOLOv3-Tiny is a simplified version of YOLOv3, which has a much smaller number of conv layers compared to YOLOv3 [21]. It requires less storage space and computing overheads, which is suitable to be deployed on resource-constraint devices [22]. Figure 1 shows the dataflow of YOLOv3-Tiny including the number of operations required for convolution.

2.2 Low bit quantizaiton

As mentioned above, the BN layers into previous conv layers can significantly reduce the latency of network inference by removing extra computation overhead, and further reduce the power consumption by decreasing the access to off-chip memory. Since the parameters in BN layer are fixed during inference, then the BN layer is performed as a 1×1 conv layer. Equation (1) is reformulated as follows:

$$\boldsymbol{y} = \gamma_{BN}^{(i)} \boldsymbol{o} + \beta_{BN}^{(i)} \tag{2}$$

where $\boldsymbol{o}$ is the output of previous conv layer, and

$$\gamma_{BN}^{(i)} = \frac{\gamma^{(i)}}{\sqrt{[\sigma^{(i)}]^2+\varepsilon}},\ \beta_{BN}^{(i)} = -\frac{\gamma^{(i)}\mu^{(i)}}{\sqrt{[\sigma^{(i)}]^2+\varepsilon}} + \beta^{(i)}$$

The output operator of quantized conv layer can be expressed as $\boldsymbol{o} = \alpha_a \boldsymbol{q}_a \alpha_w \boldsymbol{q}_w$ where $\alpha_a \boldsymbol{q}_a$, $\alpha_w \boldsymbol{q}_w$ donate the quantized activations and weights. The bias is not used in the convolution layer and $\beta^{(i)}$ in the adjacent BN layer needs to be calculated. Obviously, we can merge BN layers and update the parameters as $\hat{\alpha}_w = \gamma_{BN}^{(i)} \alpha_w, \hat{\boldsymbol{\beta}} = \beta_{BN}^{(i)}$ where $\hat{\alpha}_w$ and $\hat{\boldsymbol{\beta}}$ are tensors over channels. It should be noted that re-training is processing twice before and after the BN fusion. The BN layer fusion method can help the training to converge faster and effectively while reducing the amount of FPGA calculations.

2.3 Transformed convolutional layer

Though YOLOv3-tiny is chosen and compressed as our target small neural network for object detection running on embedded FPGA, whose parameters and fmaps can not fully implemented using on-chip memory. We base our analysis on the design in [13]. To increase utilization rate of weights and input fmaps, this design employs loop transformations, like loop reordering, tiling, and unrolling to reduce storage memory and increase throughput [23]. The transformed loop is widely used as a basic template to construct the FPGA-based accelerator.

Using the optimization in [13], the nested loops are transformed into Algorithm 1. The In_{buf}, Out_{buf}, and W_{buf} which store in block RAM (BRAM), which represent on-chip buffers for input, output, and weights respectively. Since the on-chip memory can not store all parameters and fmaps, transferring data between the on-chip buffers and off-chip memory is necessary. Double-buffering is adopted to overlap data transfer with computation, requiring each memory with twice the capacity. The loops R, C, M, and N are tiled with factors T_r, T_c, T_m, and T_n, respectively. These loop tiling factors decide the size of each on-chip buffer, and the loop order will control how data are transferred. T_m and T_n control how the compute modules are constructed.

3. Accelerator design

3.1 Modeling analysis

Given a specific hardware device with resources budget (e.g., a number of DSP slices and limited bandwidth), one can find the optimal $\langle T_m, T_n, T_r, T_c \rangle$ for a given conv layer but not for the whole model, based on the modeling analysis in [13]. An alternative method is to use unified tailed factors for all layers. The analysis about Total Execution Cycles (TEC), Computational Throughput (CT), and Required Bandwidth (RB) is calculated as follows:

$$
\begin{aligned}
TEC &\approx \left\lceil\frac{M}{T_m}\right\rceil \times \left\lceil\frac{N}{T_n}\right\rceil \times \left\lceil\frac{R}{T_r}\right\rceil \times \left\lceil\frac{C}{T_c}\right\rceil \times T_r \times T_c \times K^2 \\
&\approx \left\lceil\frac{M}{T_m}\right\rceil \times \left\lceil\frac{N}{T_n}\right\rceil \times R \times C \times K^2 \\
CT &= \frac{2\times M\times N\times R\times C\times K^2}{TEC} \approx 2\times T_m\times T_n \\
RB &\approx \frac{\alpha_{in}\times B_{in} + \alpha_w\times B_w + \alpha_{out}\times B_{out}}{TEC}
\end{aligned}
\tag{3}
$$

where $B_{in} = T_n[(T_r-1)S+K][(T_c-1)S+K]$, $B_w = T_mT_nK^2$, $B_{out} = T_mT_rT_c$, $\alpha_{in} = \alpha_w = \frac{M}{T_m}\times\frac{N}{T_n}\times\frac{R}{T_r}\times\frac{C}{T_c}$, $\alpha_{out} = \frac{M}{T_m}\times\frac{R}{T_r}\times\frac{C}{T_c}$. It should be noted that $\alpha_{in}, \alpha_w, \alpha_{out}$ and B_{in}, B_w, B_{out} denote the trip counts and buffer sizes which have mentioned before, respectively. To simplify the formula, we assume the stride $S = 1$ and omit the K in B_{in}, which reduces from $T_n[(T_r-1)S+K][(T_c-1)S+K]$ to $T_nT_rT_c$.

Given a fixed clock frequency and a conv layer structure factor tuple $\langle M,N,R,C,K\rangle$, we can estimate the computational throughput and required bandwidth using Equation (3). Without further optimization, the only way to improve performance is increasing T_m and T_n, which leads high bandwidth demand. And it needs more on-chip memory for In_{buf}, Out_{buf} and W_{buf}. The number of BRAMs and required bandwidth may exceed the maximum value before the DSP slices are fully utilized [24]. Also one DSP slice is used to perform only one MAC in this transferred loop, even for low bit operations, which means the potential computing power is not exploited. Parallelism in other dimensions is required to implement better performance with accepted hardware resources and bandwidth [25].

3.2 Convolution unit architecture

To utilize our hardware-friendly quantization and exploit better performance, we reorder the loops $\langle K,K\rangle$ to the innermost, and unroll the loops $\langle M,N,K,K\rangle$ concurrently. In this way, $K\times K$ read requests to the In_{buf} are required per cycle, while the In_{buf} is stored at BRAM which not support more than one request to different addresses per cycle.

The line buffer and window buffer memories implemented are shown in Figure 2(b). Each time through the loop, the window is shifted and filled with one pixel coming from the In_{buf} and $K-1$ pixels coming from the line buffer. Additionally, the input pixel is shifted into the line buffer in preparation to repeat the process on the next line. In the $K\times K$ conv computation, the spatial shift introduced above is undesirable and needs to be eliminated during a few cycles. A common way is extending the iteration domain. In this technique, the loop bounds are increased by a small amount so that the first input pixel is read on the first loop iteration, but the conv operation is not started until later in the iteration space. In YOLOv3-Tiny model, there are only 3×3 and 1×1 two kinds of convolvers to be implemented, so T_n window buffers are required, where each array will be partitioned completely as 9 independent registers. While the size of line buffers is fixed to T_c+2. When performing 1×1 conv operations, the line buffer will be bypassed, and the window buffer will only store one input pixel.The Loops $\langle T_r,T_c\rangle$ is extended to $\langle T_r+K-1, T_c+K-1\rangle$, while the Loops $\langle T_m,T_n,K,K\rangle$ are all unrolled. The performance and required bandwidth are re-calculated as follows:

$$
TEC \approx \left\lceil\frac{M}{T_m}\right\rceil \times \left\lceil\frac{N}{T_n}\right\rceil \times \left\lceil\frac{R}{T_r}\right\rceil \times \left\lceil\frac{C}{T_c}\right\rceil \times (T_r+K-1)(T_c+K-1) \tag{4}
$$

$$
\begin{aligned}
CT &\approx 2\times T_m\times T_n\times \frac{T_r\times T_c\times K^2}{(T_c+K-1)(T_r+K-1)} \\
RB &\approx T_n + \frac{T_mT_nK^2}{T_rT_c} + \frac{T_mT_n}{N}
\end{aligned}
\tag{5}
$$

Compared with the former transferred loop, the modified loop has $\frac{T_r\times T_c\times K^2}{(T_c+K-1)(T_r+K-1)}$ speed-up ratio, introducing K^2 more required bandwidth. To achieve the same speed-up ratio only unrolling the loop $\langle T_m,T_n\rangle$, $2(k^2-1)T_n$ more BRAMs are needed, while $(K-1)T_n$ more BRAMs are required when using the modified loops. Besides, this make one DSP slice perform more than one multiply accumulate calculation in a transferred loop for low bit operations. Therefore DSP slices can provide more performance density using the modified loop. Our method can improve the performance using less BRAMs, especially for the resource-constrained embedded FPGA.

3.3 System architecture

In this work, we propose a CPU+FPGA heterogeneous architecture to accelerate YOLOv3-Tiny. Figure 2(a) shows an overview of the overall architecture. The whole architecture consists of three compute units, input, output and parameter buffers, an AXI bus for off-chip memory transfer, and a controller to schedule the on-chip memory access and computation. The three compute units work on different configurations of layers: the *Conv* unit for all conv layers, the *Pooling* unit for max-pooling layers and up-sampling layers. The Non-Linearity (*NL*) unit for all the conv layer except two output conv layers. Each PE has Tn×K×K low-bit multipliers and Tn add trees working concurrently. Thus 2×Tm×Tn×K×K MACs can be done each cycle for the convolutional computing unit. Since weights and feature maps of Tiny-YOLOv3 model are too large for embedded FPGA, we store each conv layer's weights and output fmaps in the off-chip RAM. The conv layer starts to run until previous conv layer processing is done on the proposed accelerator. As for max-pooling layers and the up-sampling layer, there are also plenty of data transfers between on-chip and off-chip memory. Since the write back operation of output fmaps occurs every $\frac{N}{T_n}$ conv computing cycles, the pooling and up-sampling operations can be done fully on chip, using extra 8-bit pool buffers $Pool_{buf}$ whose size is the same as Out_{buf} without writing and loading back again. This schedule only increases the required bandwidth for writing back output fmaps, and the required bandwidth is showed as follows:

$$
RB \approx T_n + \frac{T_mT_nK^2}{T_rT_c} + \frac{T_mT_n}{N-T_n} \tag{6}
$$

There are also $2\times T_m$ increased BRAMs for this schedule. Using the modified extended conv loops can achieve high throughput with introducing much more BRAM demands, so we can still achieve good hardware utilization.

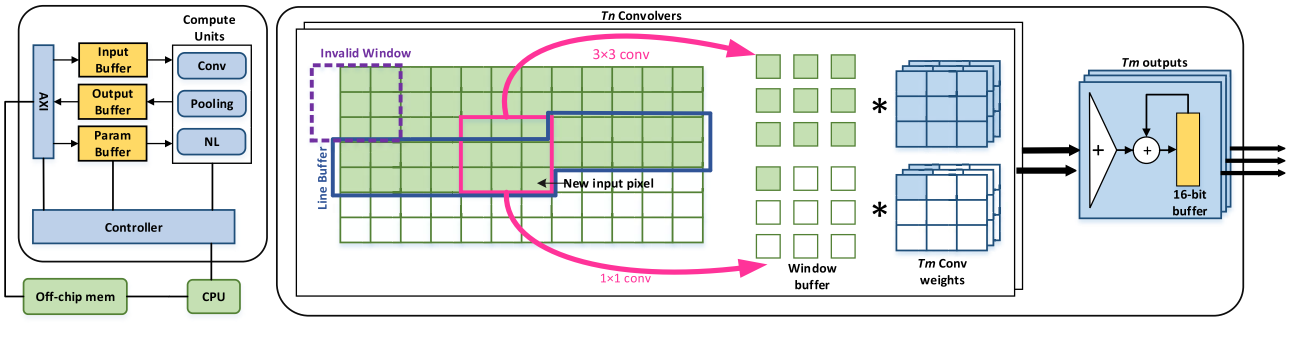

Fig. 2 Architectural diagrams of the proposed HFOD accelerator: (a) the overall architecture; (b) architecture of the Conv unit with line buffer and window buffer. The unit can perform $T_m \times T_n \times K^2$ MACs per cycle.

Table I Accuracy of quantized Tiny-YOLOv3.

Quantization	Accuracy (mAP)	Weight size (MB)	Activation size (MB)
Baseline (32-b W, 32-b A)	61.82	33.22	7.83
w/o BN-fused (2-b W, 8-b A)	59.83	2.08	1.96
w/ BN-fused (2-b W, 8-b A)	56.02	2.08	1.96

Table II FPGA resource utilization.

Resource	BRAM	DSP	FF	LUT
Used	128	205	36007	49751
Available	280	220	106400	53200
Utilization(%)	45	93	33	93

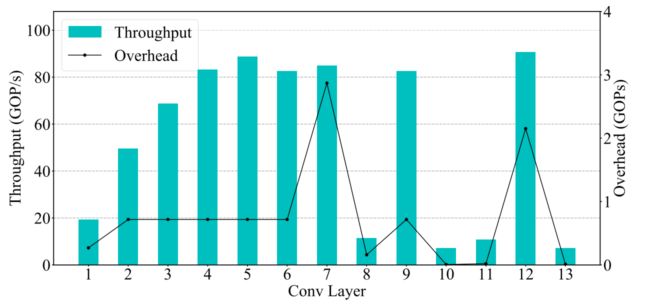

Fig. 3 Throughput and overhead for each conv layer implemented on our proposed accelerator.

4. Experimental results

4.1 Low-bit quantization implementation

The hardware-friendly, low-bit quantization is implemented using Pytorch [26]. The YOLOv3-Tiny is trained on PASCAL VOC 2007+2012 dataset and tested on PSCAL VOC 2007. First, we train the network from scratch using full precision, then quantize the conv layer and retrain with full precision BN layer. Finally, fuse the BN layer and re-quantize the merged conv layer and finetune using smaller learning rates. Table I shows the accuracy of the quantized YOLOv3-Tiny. The quantized network with 2-bit weights and 8-bit activations (i.e. 2-b W and 8-b A) incurs an accuracy loss of approximately 5.3 mAP compared with the full-precision network. The model size is reduced by 16×, and the activation size is reduced by 4×. For further showing the effectiveness of the proposed method, we test the application of our model on the infrared aerial dataset. Experimental results on the infrared aerial dataset also show that the proposed channel-wise quantization can significantly improve the performance of the low-bit quantized model and the retraining strategy contributes a lot to the range optimization of parameter quantization.

4.2 Results

We evaluate our design on PYNQ-Z2, which uses a low-cost Xilinx Zynq-7000 SoC containing an XC7Z020 FPGA alongside an ARM Cortex-A9 embedded processor. This board can provide 3.2 GB/s of DRAM bandwidth. Its working frequency is 150 MHz. We adopt Vivado HLS 2018.2 as the primary design tool to develop our accelerator using C/C++ language and export the RTL as a Vivado's IP. The exported RTL is synthesized and implemented in Vivado 2018.2. The placement and routing is completed with Vivado tool set. Based on modeling analysis in Section 4, we choose the tile parameters $\langle T_m, T_n, T_r, T_c \rangle$ which achieve the highest throughput under the hardware resources constrained and limited bandwidth. The tile parameters $\langle T_m, T_n, T_r, T_c \rangle$ is fixed to $\langle 10, 4, 26, 26 \rangle$. After that, the resource utilization of our implementation is reported out, as shown in Table II. We can tell that our accelerator design has fully utilized FPGA's computing performance because the DSPs are almost used up.

Since layers of the network must be run one after the other on the same accelerator, each conv layer's throughput can be separately measured. The throughput and overhead for each conv layer are showed in Figure 3. The conv layer 8, 10, 11, and 13, have the least overhead since they all perform 1×1 conv operations, and these overheads account for less than 2% of the whole network's overhead. Therefore, our accelerator focus on the optimization for 3×3 conv operations to improve overall performance. We can tell that peak performance is achieved when running conv Layer 12, which is 90.6 GOPS.

Table III compares our implementation against the previous FPGA accelerators for YOLO CNN [27, 28, 29, 30, 31, 32] where all numbers are retrieved from the respec-

Table III Comparison of our work with the previous works for YOLO CNN hardware.

	Yu, et al. [27]	Angel Eye [28]	Zynq YOLO [29]	Ma, et al. [30]	Tincy YOLO [31]	Afzal, et al. [32]	Ours
Platform	Zynq 7Z020	Zynq 7Z020	Zynq 7Z020	Virtex7 485t	Zynq Ultrascale+ XCZU3EG	Virtex7 VC707	Zynq 7Z020
Clock (MHz)	100	214	150	143	150	200	150
Precision(Weight, Activation)	(16,16)	(8,8)	(8,8)	(16,16)	(1,3)	(18,18)	(2,8)
Throughput(GOPS)	10.5	62.9	86.4	48	72	460.8	90.6
Look-Up-Table(LUT)	25900	29867	27761	163944	N/A	48583	49751
Flip-Flop(FF)	46700	35489	26600	69014	N/A	93225	36007
DSPs	160	N/A	220	112	N/A	2304	205
Perf. Density(GOPS/DSP)	0.06	N/A	0.39	0.43	N/A	0.2	0.44
Power (W)	3.36	3.5	N/A	N/A	6	4.81	2.5
Power efficiency(GOPS/W)	3.15	17.9	N/A	N/A	12	95.6	36.4
Design Methodology	Vivado HLS	RTL	RTL	Vivado HLS	Vivado HLS	Vivado HLS	Vivado HLS

tive papers. It should be noted that the comparisons with [27, 28, 29] are against the same device while the others are against larger FPGAs [30, 31, 32]. Yu, et al. [27] implements the YOLOv3-Tiny on the same board and Afzal, et al. [32] implements the YOLOv3-Tiny on Virtex7 VC707, while the other designs are based on YOLO or YOLO-Tiny. Throughput is shown in giga-operations-per-second (GOPS). The power efficiency is defined as giga-operations-per-second-per-watt (GOPS/W). The accelerators designed using RTL typically have the higher throughput than high level synthesis tools. It can be seen from Table III that our HFOD accelerator outperforms other FPGA accelerators designed with RTL for YOLO in pure throughput.

Compared to [29], since the algorithm deployed in [29] is YOLO-tiny while YOLOv3-Tiny is adopted in our method, we need to focus on the unit resource consumption not overall consumption which will be influenced by network structure. As shown in Table III, our method both demonstrates the superiority with [29] in terms of both power efficiency and throughput. Compared with most accelerators designed with HLS, our design achieves the best throughput while consuming almost the same FPGA resources. Although Afzal's work [32] has better throughout performance, they consume more resources. Compared to [32], a more light-weight quantization method is adopted in our design, which achieves 2-bit weights and 8-bit activations compression; a novel multiply-add operation unit structure is proposed in our met; and our design is applicable to a wider range of application scenarios. As we known that the multiply-add operation in convolution mainly consumes DSP resources and the construction of logic gates mainly consumes other logic slices such as LUT and FF. As a result, Afzal's work [32] consumes more logic slices to achieve a higher throughtput. By analyzing the throughput of unit logic slice, our accelerator can achieve the best resource utilization and is also much more power efficient. Besides, the quantization method proposed in this paper introduces the BN-fused optimization and avoids extra access to off-chip memory. Owning to the low-bit quantization, when compared with different types of FPGA devices, we further normalize the performance by the number of logic slices used. For our accelerator design, logic slices of FPGA contribute more computing performance compared with regular 8-bit/16-bit fixed point quantization.

5. Conclusion

In this work, we propose HFOD, a hardware-friendly quantization method for object detection, optimizing from both algorithm and hardware. We adopt a channel-wise, uniform quantization method to compress YOLOv3-Tiny model. Weights are quantized to 2-bit while activations are quantized to 8-bit for all convolutional layers. The experimental results show that, our proposed method can significantly compress the YOLO model without introducing large accuracy degradation. The detection accuracy of quantized YOLOv3-Tiny has 5.8 mAP degradation compared with full precision network. As for the hardware design, a flexible, efficient conv unit structure is designed to utilize hardware-friendly quantization. The comparisons with other FPGA-based accelerator designed with HLS for YOLO CNN show that our accelerator presents high throughput and low power consumption and has better resources utilization, which is suitable for embedded FPGAs.

References

[1] S. Ren, *et al.*: "Faster R-CNN: towards real-time object detection with region proposal networks," Advances in Neural Information Processing Systems (2015) 91.

[2] W. Liu, *et al.*: "SSD: single shot multibox detector," European Conference on Computer Vision (2016) 21 (DOI: 10.1007/978-3-319-46448-0_2).

[3] J. Redmon and A. Farhadi: "YOLO9000: better, faster, stronger," The IEEE Conference on Computer Vision and Pattern Recognition (2017) 7263 (DOI: 10.1109/cvpr.2017.690).

[4] M.J. Shafiee, *et al.*: "Fast YOLO: a fast you only look once system for real-time embedded object detection in video," arXiv preprint (2017) arXiv:1709.05943 (DOI: 10.48550/arXiv.1709.05943).

[5] B. Veytsman: "acmart — Class for typesetting publications of ACM," `http://www.ctan.org/pkg/acmart`

[6] G. Zhou, *et al.*: *Body Sensor Networks* (MIT Press, Cambridge, MA, 2008).

[7] A. Canis, *et al.*: "LegUp: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems," ACM Transactions on Embedded Computing Systems **13** (2013) 1 (DOI: 10.1145/2514740).

[8] J. Cong, *et al.*: "High-level synthesis for FPGAs: from prototyping to deployment," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **30** (2011) 473 (DOI: 10.1109/tcad.2011.2110592).

[9] V. Kathail, *et al.*: "SDSoC: a higher-level programming environment for Zynq SoC and Ultrascale+ MPSoC," Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (2016) 4 (DOI: 10.1145/2847263.2847284).

[10] T.S. Czajkowski, *et al.*: "From OpenCL to high-performance hardware on FPGAs," 22nd International Conference on Field Programmable Logic and Applications (2012) 531 (DOI: 10.1109/fpl.2012.6339272).

[11] M. Saeedi, *et al.*: "Synthesis of reversible circuit using cycle-based approach," J. Emerg. Technol. Comput. Syst. **6** (2010) 1 (DOI: 10.1145/1877745.1877747).

[12] S. Cohen, *et al.*: "Deciding equivalences among conjunctive aggregate queries," J. ACM **54** (2007) 5 (DOI: 10.1145/1219092.1219093).

[13] C. Zhang, *et al.*: "Optimizing FPGA-based accelerator design

for deep convolutional neural networks," Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA) (2015) 161 (DOI: 10.1145/2684746.2689060).

[14] H. Ozaku, *et al.*: "Acceleration of database query processing using FPGA," IEICE Technical Report **120** (2021) 90.

[15] S. Albawi, *et al.*: "Understanding of a convolutional neural network," International Conference on Engineering and Technology (2017) 1 (DOI: 10.1109/icengtechnol.2017.8308186).

[16] J. Redmon and A. Farhadi: "YOLOv3: an incremental improvement," arXiv preprint (2018) arXiv:1804.02767 (DOI: 10.48550/arXiv.1804.02767).

[17] S. Ioffe and C. Szegedy: "Batch normalization: accelerating deep network training by reducing internal covariate shift," arXiv preprint (2015) arXiv:1502.03167 (DOI: 10.48550/arXiv.1502.03167).

[18] G. Zhou, *et al.*: "A multifrequency MAC specially designed for wireless sensor network applications," ACM Trans. Embed. Comput. Syst. **9** (2010) 39:1 (DOI: 10.1145/1721695.1721705).

[19] S.W. Smith, "An experiment in bibliographic mark-up: parsing metadata for XML export," Proceedings of the 3rd. Annual Workshop on Librarians and Computers **3** (2010) 422.

[20] T. Utsunomiya, *et al.*: "FPGA accelerator of cnn using power of 2 approximation and pruning weights," IEICE Technical Report **117** (2018) 119.

[21] J. Qiu, *et al.*: "Going deeper with embedded FPGA platform for convolutional neural network," Proceedings of the International Symposium on Field-Programmable Gate Arrays (2016) 26 (DOI: 10.1145/2847263.2847265).

[22] A. Natarajan, *et al.*: "Investigating network architectures for body sensor networks," *Network Architectures* (Keleuven Press, Dayton, OH, 2007) 322.

[23] M. Yamazaki, *et al.*: "Side channel security of an FPGA pairing implementation with pipelined modular multiplier," IEICE Technical Report **119** (2019) 151.

[24] M. Clark: "Post congress tristesse," TeX90 Conference Proceedings (1991) 84.

[25] J. Qiu, *et al.*: "Going deeper with embedded FPGA platform for convolutional neural network," Proceedings of the International Symposium on Field-Programmable Gate Arrays (2016) 26 (DOI: 10.1145/2847263.2847265).

[26] A. Paszke, *et al.*: "PyTorch: an imperative style, high-performance deep learning library," Advances in Neural Information Processing Systems (2019) 8024.

[27] Z. Yu and C.-S. Bouganis: "A parameterisable fpga-tailored architecture for YOLOv3-tiny," International Symposium on Applied Reconfigurable Computing (2020) 330 (DOI: 10.1007/978-3-030-44534-8_25).

[28] K. Guo, *et al.*: "Angel-eye: a complete design flow for mapping CNN onto embedded FPGA," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. **37** (2018) 35 (DOI: 10.1109/tcad.2017.2705069).

[29] K. Guo, *et al.*: "From model to FPGA: software-hardware co-design for efficient neural network acceleration," 2016 IEEE Hot Chips 28 Symposium (2016) 1 (DOI: 10.1109/hotchips.2016.7936208).

[30] J. Ma, *et al.*: "Hardware implementation and optimization of tiny-YOLO network," Digital TV and Wireless Multimedia Communication (2018) 224 (DOI: 10.1007/978-981-10-8108-8_21).

[31] T.B. Preußer, *et al.*: "Inference of quantized neural networks on heterogeneous all-programmable devices," 2018 Design, Automation & Test in Europe Conference & Exhibition (2018) 833 (DOI: 10.23919/date.2018.8342121).

[32] A. Ahmad, *et al.*: "Accelerating tiny YOLOv3 using FPGA-based hardware/software co-design," 2020 IEEE International Symposium on Circuits and Systems (2020) 1 (DOI: 10.1109/iscas45731.2020.9180843).