A Low-Latency FPGA Implementation for Real-Time Object Detection

Jinming Zhang¹, Lifu Cheng², Cen Li¹, Yongfu Li¹, Guanghui He¹*, Ningyi Xu¹, Yong Lian¹
¹Shanghai Jiao Tong University, Shanghai, China
²Shanghai Aerospace Electronic Technology Institute, Shanghai, China
Email: guanghui.he@sjtu.edu.cn

Abstract—The advancement of object detection algorithms makes them widely used in autonomous systems. However, due to high computational complexity of Convolutional Neural Networks (CNNs), stringent latency requirement is hard to meet for real-time object detection. To address this problem, a low-latency accelerator architecture is proposed in this paper. A fine-grained column-based pipeline architecture with padding skip technique is implemented to reduce the start-up time of pipeline. In order to cut down the computational time of CNN, double signed-multiplication correcting circuit is introduced. In addition, pooling unit with share buffer is proposed to reduce storage cost for pooling layer. To demonstrate our new architecture, we implement the YOLOv2-tiny deep neural network (you-only-look-once) with input size 1280x384 on ZC706 development board, improving the latency by 2.125× to 2.34× compared to previous FPGA accelerator for YOLOv2-tiny.

Index Terms—Low-Latency, YOLO, FPGA

I. INTRODUCTION

In recent years, convolutional neural networks (CNNs) have greatly improved the performance of objection detection algorithms [1], which are widely used in robotics, self-driving vehicles and unmanned aerial vehicles (UAVs). In these autonomous systems, latency is one of the key metrics for obstacle avoidance and emergency procedures.

Advanced object detection algorithms come along with high computational complexity, which require extensive long processing time [2]. Therefore, hardware accelerators based on GPU [3], ASIC [4], and FPGA [5] emerge to reduce inference latency. Since the power consumption of GPU usually reaches about 100W [6], and the development cycle of ASIC is relatively long, FPGA-based hardware accelerator achieves a balance between performance, power and flexibility [7].

FPGA-based accelerators can be classified as Single Computation Engine Architectures (SCEAs) and Streaming Architectures (SAs) [8]. SCEAs share common computation resources temporally for different layers. Next layer can not proceed until the former layer completes its calculation totally in [9], [10] and [11]. Instead, SAs map different layer to different PE to form a pipeline [12]. Existing designs prefer to focus on improving the hardware throughput and energy efficiency, and many techniques have been adopted in CNN accelerators such as model quantization [13], model pruning [14] and Winograd/FFT algorithm [10]. However, not all of these methods are beneficial to latency optimization. For example, batch processing could not reduce latency even though it can improve throughput greatly in [15] and [16]. Thus most of these architectures fail to meet the latency requirement for real-time object detection.

To further optimize the processing latency, [15] and [17] utilize layer-based pipeline to reduce startup latency. Ref. [15] quantizes YOLO [18] to Binary Neural Network (BNN) to store all weight on BRAM. However, for object detection algorithms, over-quantization will lead to non-negligible accuracy drop, especially for large-scale datasets [7].

As such, a low-latency accelerator architecture is proposed in our design, achieving 2.125× latency reduction compared to [17] for YOLOv2-tiny and 2.340× latency reduction compared to [11]. The contributions of this paper are:

1) A low-latency pipeline architecture with padding skip technique is proposed based on latency analysis.
2) A double signed-multiplication correcting circuit is introduced to support two signed multiplications in one DSP, improving the efficiency of DSP and cutting down the computational time of CNN.
3) Pooling unit with share buffer is utilized to reduce storage cost for pooling layer. In this respect, BRAM is reduced by 20.7% for YOLOv2-tiny.

The rest of this paper is organized as follows. Sect. II introduces the latency analysis and corresponding hardware optimization. Sect. III describes the proposed architecture and Sect. IV presents the evaluation results. Finally, Sect. V concludes this paper.

II. OPTIMIZATION FOR LATENCY

In a normal pipeline architecture, the next stage’s process can not proceed until the former layer completes its calculation, resulting in a long computation latency. Besides, the feature maps between two layers have to be stored in DRAM for most embedded devices with limited on-chip memory. The data have to be fetched on-chip again before the next layer’s execution, which consumes considerable time and power. Usually, the data between two adjacent layers are not coupled tightly.

To address this problem, the latency analysis is carried on as follow. For 3×3 Conv in Fig. 1(a), only 3 columns of
data are needed from the former layer to output one column results of the next layer. It means that the next layer can start calculation only if the needed columns are ready from the former layer, without waiting for the whole layer’s calculation completed. Motivated on this idea, the column-based pipeline is constructed and the overall latency from input to output is decreased greatly, which is shown in Fig. 1(b). Inference latency can be calculated as:

\[
\text{latency} = t_{\text{trans}} + t_{\text{comp}} = t_{\text{trans}} + \sum_{i=1}^{\text{num columns}} (c_{\text{fill}}^{(i)} + c_{\text{comp}}) \times t_{\text{col}},
\]

where \(t_{\text{col}}\) is the computing time for one column in one convolution layer. For \(i\)-th layer, \(t_{\text{col}}^{(i)}\) is related to the feature map size and allocated DSPs. In the pipeline architecture, \(t_{\text{col}}\) depends on the slowest convolution layer, i.e.

\[
t_{\text{col}} = \max(t_{\text{col}}^{(i)}).
\]

\(c_{\text{comp}}\) is the width of input image. And \(c_{\text{fill}}^{(i)}\) is the number of columns to fill the pipeline for \(i\)-th layer, which is decided by kernel width. Therefore, \(3 \times 3\) Conv needs 3 columns to calculate and \(1 \times 1\) Conv only needs 1 column to startup. \(t_{\text{trans}}\) is the time spent on transfer data to and from off-chip DRAM.

### III. PROPOSED ARCHITECTURE

#### A. Overall Architecture

According to Eq. 1, the inference latency of pipeline architecture consists of 3 parts: data transmission time, setup delay to fill the pipeline and computing time. Correspondingly, new methods to reduce latency is presented in below. Compared to [15], the input images and output results of proposed design are directly transmitted between the FPGA and the host personal computer(PC) without the intervention of DRAM to reduce data transmission time. The fine-grained column-based pipeline architecture is implemented to reduce setup delay and double-signed-multiplier packing technique is used to reduce computing time. Moreover, pooling unit with share buffer could optimize BRAM consumption compared with [15].

Fig. 2 shows the overall architecture design. The trained weights are pre-loaded to DRAM from host PC through the peripheral component interconnect express (PCIe) cable. During the inference stage, input images are first transmitted to the Image Input FIFO column by column through PCIe. Then, these images are processed by the pipelined CONV PE. The CONV PE of each layer is organized as the column-based pipeline, which will process the input data column by column to reduce startup latency. Finally, the detection results of the last layer are sent to the host PC for post-processing.

Especially, for the processing of the first and the last column’s output, the padding can be skipped, which will reduce the start-up time. For example, the start-up time of a \(3 \times 3\) convolution layer with this technique is reduced by 1/3. Besides, without the DRAM access for input images and intermediate results, the power efficiency can be improved as well.

#### B. Parameterized CONV PE Design

The workflow of the CONV PE is designed for the proposed column-based pipelined architecture as illustrated in Fig. 3. In this example, a \(3 \times 3 \times 4\) input feature map is convolved by four \(2 \times 2 \times 4\) kernels to generate \(2 \times 2 \times 4\) outputs with \(C_i = 2\) and \(K_i = 2\).

Due to the limitation of on-chip memory, for each layer, only \(K_i\) groups of kernels can be stored on the chip (the real memory size will be set to \(2K_i\) for Ping-Pong buffer). In each cycle, \(C_i \times K_i\) inputs and \(C_i \times K_i\) weights are fetched to generate \(C_i \times K_i\) products in one CONV PE, which will be accumulated to generate \(K_i = 2\) partial sum. After 8 cycles, 2 output feature map pixels are generated and the fetching order of input feature map is shown in Fig. 3(b). Then the convolution window can shift down to reuse weight. After another 8 cycles, other 2 groups of kernels need to be fetched to re-use feature map. When all kernel groups are calculated, the kernel window can shift right to go on the next column’s convolution. The output order is shown in Fig. 3(c).

All different kinds of convolutions can be calculated in proposed architecture, regardless of the size of the input feature maps and the kernels. Hence, a parameterized CONV

---

Fig. 1. (a) Column-based Pipeline for \(3 \times 3\) Conv and (b) Example of Latency Calculation.

Fig. 2. Architecture of the proposed design.
PE can be designed to support general convolution. Moreover, with two parameters $C_i$ and $K_n$, CONV PE can be configured to various parallelism for different network structures and different hardware platforms.

**C. Double Signed-multiplication Packing Method**

Considering that DSP takes on the vast majority of computations in CNN, the utilization of the DSP directly affects the performance of the accelerators in most designs. Meanwhile, since the parameters of the CNN model can be quantized to lower bits without significant accuracy loss [13], more multiplications can be packed into just one DSP48E1 in Xilinx FPGA.

As demonstrated in Fig. 4(a), 2$n$-bit data $A$ and $B$ can be linked as one input of the DSP. With $(n+1)$ bits 0 inserted, crosstalk will not occur between the two multiplication $A \times C$ and $B \times C$. But results are incorrect when the three inputs are signed in Fig. 4(b). Hence, to pack two signed multiplications into one DSP, extra correcting circuit is needed to fix the results.

The upper 2$n$-bit multiplication ($A \times C$) is treated as a signed-unsigned multiplication in the packed DSP, i.e.

$$A \times C = A \cdot (C_n - c_{n-1} \cdot 2^n) = A \cdot C_n - 2^n c_{n-1} \cdot A = A \cdot C_n + \Delta H,$$

where $A$ and $C$ are the $n$-bit signed input ($C = c_{n-1}c_{n-2} \cdots c_0$), $A \times C$ represents the correct multiplication result, $C_n$ represents the value of $C$ interpreted as an $n$-bit unsigned number, and $A \cdot C_n$ represents the upper 2$n$-bit output of the DSP.

Similarly, $B \times C$ is treated as 2$n$-bit signed-unsigned multiplication in the packed DSP, which is formulated as:

$$B \times C = (B_n - b_{n-1} \cdot 2^n) \cdot (C_n - c_{n-1} \cdot 2^n) = B_n \cdot C_n - 2^n (c_{n-1} \cdot B_n + b_{n-1} \cdot C_n) + c_{n-1} \cdot b_{n-1} \cdot 2^{2n} = B_n \cdot C_n - 2^n (c_{n-1} \cdot B_{n-1} + b_{n-1} \cdot C_{n-1}) = B_n \cdot C_n + \Delta L,$$

where $\Delta L$ represents the error needed to be corrected in the lower 2$n$-bit results.

To generate the correct outputs, the correcting circuit is demonstrated in Fig. 5(d). In the packed DSP, a Data Selector is introduced to generate the intermediate parameters depending on the signed bits of $B$ and $C$. Then, an adder can output $\Delta H$ and $\Delta L$, which are added to the results from the DSP.

---

**Fig. 3.** Data flow of proposed column-based pipelined.

**Fig. 4.** Example of packing two multiplications in one DSP.

**TABLE I**

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input</th>
<th>Kernel</th>
<th>DSP $C \times K_n$</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1280 $\times$ 384 $\times$ 3</td>
<td>3 $\times$ 3 $\times$ 16</td>
<td>32</td>
<td>4 $\times$ 16</td>
</tr>
<tr>
<td>2</td>
<td>640 $\times$ 192 $\times$ 16</td>
<td>3 $\times$ 3 $\times$ 32</td>
<td>64</td>
<td>4 $\times$ 32</td>
</tr>
<tr>
<td>3</td>
<td>320 $\times$ 96 $\times$ 32</td>
<td>3 $\times$ 3 $\times$ 64</td>
<td>64</td>
<td>2 $\times$ 64</td>
</tr>
<tr>
<td>4</td>
<td>160 $\times$ 48 $\times$ 64</td>
<td>3 $\times$ 3 $\times$ 128</td>
<td>64</td>
<td>4 $\times$ 32</td>
</tr>
<tr>
<td>5</td>
<td>80 $\times$ 24 $\times$ 128</td>
<td>3 $\times$ 3 $\times$ 256</td>
<td>64</td>
<td>8 $\times$ 16</td>
</tr>
<tr>
<td>6</td>
<td>40 $\times$ 12 $\times$ 256</td>
<td>3 $\times$ 3 $\times$ 512</td>
<td>64</td>
<td>16 $\times$ 8</td>
</tr>
<tr>
<td>7</td>
<td>40 $\times$ 12 $\times$ 512</td>
<td>3 $\times$ 3 $\times$ 512</td>
<td>128</td>
<td>32 $\times$ 8</td>
</tr>
<tr>
<td>8</td>
<td>40 $\times$ 12 $\times$ 512</td>
<td>3 $\times$ 3 $\times$ 512</td>
<td>128</td>
<td>32 $\times$ 8</td>
</tr>
<tr>
<td>9</td>
<td>40 $\times$ 12 $\times$ 512</td>
<td>1 $\times$ 1 $\times$ 40</td>
<td>2</td>
<td>2 $\times$ 2</td>
</tr>
</tbody>
</table>

Total: — — 610 — 22.12 ms
D. Share Buffer for Pooling Unit

Pooling layers are common in CNN for down-sampling. Conventional pooling unit in Fig. 5 (a) needs separate line buffer to save output feature for 2 × 2 max-pooling or average-pooling [15]. The depth of buffer is proportional to the width of input image, and the number of buffer is related to parallelism factor of output channel and number of convolution layers in the pipeline architecture.

Share buffer for pooling unit is implemented to reduce consumption of BRAM in our design. In Fig. 5 (b), computation results from CONV PE will compare and update data in BRAM. Leaky ReLU is also carried on in this process. Dual-port BRAM prevents data conflict for both convolution and pooling layer. Since there is no storage cost for pooling layer, BRAM is reduced by 20.7% for YOLOv2-tiny.

IV. IMPLEMENT RESULTS

To demonstrate the performance of the proposed low-latency architecture, a YOLO2-tiny network with input size 1280 × 384 is implemented on Xilinx ZC706 development board and tested on the KITTI dataset [19]. To make a fair comparison with the baseline in [17], the kernel numbers of CONV7 and CONV8 are changed to 512. And fine-tuning for quantized network leads to a better mean average precision (mAP) compared with [17].

A. Parallelism Factors of Each Layer

To balance the pipeline time, each layer’s parallelism factors (C and K) of YOLO (1280 × 384) are chosen depending on the computation complexity, which are listed in Table I. Judging from the list, except from the CONV1 and CONV9, most of the convolution layers are balanced well.

B. Hardware Implementation Results

Based on the parallelism factors given in Table I, the YOLO2-tiny is mapped on Xilinx ZC706 development board. Input images are sent to the FPGA through PCIe from host PC. After the inference phase, the detection results are sent back to the host PC for post-processing.

Table II compares the proposed design with the existing works on YOLO hardware implementation. According to [17], DSP \( \text{efficiency} \) is introduced to evaluate the utilization efficiency of the DSPs, which is defined as:

\[
\text{DSP}_{\text{efficiency}} = \frac{\text{Throughput}}{\beta \times \text{DSP}_{\text{used}} \times f^3}
\]

where the numerator is the actual achieved performance (GOPS) and the denominator is the theoretical DSPs’ throughput. \( \beta \) is the operation that can be handled by one DSP in one cycle (\( \beta = 2 \) in Fix16, \( \beta = 4 \) in Fix8).

As shown in Table II, with the same network model and hardware platform, our proposed design outperforms the DNN-Builder [17] in terms of overall latency (2.125×), FPS (2.142×) and DSP efficiency (1.11×). It means that the column-based pipeline with padding skip technique and double-multiplier packing method proposed in this design work well on improving hardware latency. Moreover, benefit from well-balanced pipeline dataflow, the DSP efficiency of this design obtains 2.46×, 1.41× and 1.69× improvement compared to [9] [10] and [11] respectively.

V. CONCLUSIONS

In this work, a low-latency architecture is presented to optimize the latency for real-time object detection applications based on latency analysis. Column-based pipeline structure, the padding skip technique and the double-multiplier packing method are proposed, reducing the overall latency by 2.125× to 2.34× compared to previous FPGA accelerator for YOLOv2-tiny. And the DSP efficiency of our work achieves 95.2%. Furthermore, pooling unit with share buffer is implemented to reduce BRAM consumption by 20.7%.
REFERENCES


