A Low-latency FPGA Accelerator for YOLOv3-tiny with Flexible Layerwise Mapping and Dataflow

Minsik Kim[1], Hojin Seo[1], Kyoungseok Oh[1], Youngmock Cho[1], Xuan Truong Nguyen and Hyuk-Jae Lee
Department of Electrical and Computer Engineering
Seoul National University

Abstract—**Object detection models have demonstrated outstanding performance in terms of accuracy. However, mapping convolutional neural network-based object-detection models to memory and computing-constrained devices is still challenging, which commonly leads to accuracy degradation and long latency. To address the problem, this work presents a design methodology to map the YOLOv3-tiny model onto a small FPGA board, in this case the Nexys A7-100T, which only has 0.5 MB on-chip SRAM and 240 DSPs. First, we design four identical MAC arrays to maximize the throughput by utilizing both DSPs and LUTs. Second, to exploit the MACs fully, we propose a dynamic data reuse scheme that handles inter-layer and intra-layer executions effectively under a small on-chip SRAM footprint. To this end, the proposed accelerator achieves an inference speed of 76.75 frames per second and throughput of 95.08 GOPs at 100MHz and consumes power of 2.203W. Specifically, it achieves a hardware utilization rate of 82.53%, thus significantly outperforming current YOLOv3-tiny accelerators.**

Index Terms—**Deep learning, Accelerator, FPGA, YOLOv3-tiny.**

I. Introduction

OBJECT detection, one of the most fundamental problems in computer vision, has been actively studied for a broad range of applications across various domains such as traffic detection [2], autonomous driving [3], and unmanned stores [4]. Many of these applications require low-latency and/or real-time responses (i.e., more than 30 or 60 frames per second). Meanwhile, the performance capabilities of object detection models are undergoing rapid enhancements in terms of both model accuracy and execution times. Notably, single-stage object detectors such as YOLOv3 [5], YOLOv4 [6], and EfficientDet [7] have achieved a good tradeoff between model accuracy and real-time execution using desktop GPUs. However, the execution times of such models dramatically drop on edge devices owing to their huge computational requirements. As a result, scaled and lightweight models such as YOLOv3-tiny are more favorable for real-time execution on edge devices or edge GPUs (i.e., NVIDIA Jetson Nano [10], NVIDIA Jetson Xavier NX [11]). Unfortunately, edge GPUs still consume a considerable amount of power, making them not suitable for many IoT or edge-computing applications.

At present, other alternative solutions such as (low-cost) Field Programmable Gate Arrays (FPGAs) are receiving more attention for DNN accelerators due to their low latency, good power efficiency, high configurability, and rapid prototyping. Particularly, many FPGA implementations of YOLOv3-tiny accelerators ([25], [26], [27], [28]) have been proposed. In one study [26], the convolution operation is implemented on the FPGA using the General Matrix Multiplication Principle (GEMM). Other work [27] proposes a hybrid computing architecture of ARM+FPGA, while a hardware/software co-design approach is proposed to accelerate the inference of the YOLOv3-tiny network [28]. However, it is challenging to utilize the computing resources fully due to external memory accesses and buffer reuse, leading to low hardware utilization. The two aforementioned studies [26] [27] fail to utilize the DSPs in the PEs fully, leading to low throughput. Another method [28] requires extraneous off-chip communication, as the Processing System (PS) must send the feature map and the weights to the reserved addresses of the DRAM and signal the Programmable Logic (PL) to start the convolution whenever the convolution operation needs to be performed.

To address these problems, this work presents an empirical methodology and its implementation in the design of a real-time YOLOv3-tiny accelerator on a small FPGA board, specifically the Nexys A7-100T. The contributions of this paper are as follows:

1) MAC array with a layer-wise configurable adder tree: To achieve a low-latency design, our first concept is to maximize the computing power for the given hardware. Particularly, the computing units and buffers are designed by carefully considering three aspects: (1) full utilization of both DSPs and LUTs, (2) maximization of layer-wise hardware utilization, and (3) enhancing data locality for on-chip buffer reuse.

2) Dynamic data reuse and pipelining: Our second concept is to utilize the computing elements fully by avoiding any stalls stemming from unnecessary data movements. This is achieved by dynamically selecting which data are to be reused in convolution operations - IFM or weights - according to the layer configuration. Consequently, our design avoids unnecessary external memory accesses for early layers, shortcut connections, and concatenation layers, and therefore enhances the overall hardware utilization.

3) On-chip buffer structure for random access of the spatial IFM window: To maximize the PE utilization throughout all layers, we propose a spatially-banked feature map buffer. This buffer structure allows random access to any spatial feature map window in a single cycle latency.

[1] M.S. Kim, H.J. Seo, K.S. Oh, and Y.M. Cho contributed equally.

The authors are with the Department of Electrical and Computer Engineering, Seoul National University. (e-mail: {minsik.kim, shj2044409, fudsla, truongnx}@snu.ac.kr, hjlee@capp.snu.ac.kr)

4) Implementation: The proposed accelerator achieves a real-time inference speed of 76.75 frames per second (fps) and throughput of 95.08 GOPs at 100MHz. In addition, it consumes 2.203W of power and achieves a hardware utilization rate of 82.53%, significantly outperforming all existing YOLOv3-tiny accelerators.

The rest of this paper is organized as follows. Section II briefly reviews related works and the background. Sections III describes the proposed design methodology and implementation. The experimental results and a comparison with earlier works are reported in Section IV. Section V concludes the paper.

II. Background and Related Works

In this section, first we briefly review several related studies of lightweight object detection networks that concentrate on algorithmic architecture optimization for edge devices. Then, we describe several edge-computing platforms such as edge GPUs and FPGA implementations of on-edge object detectors, and their limitations.

A. Lightweight CNNs for object detection

One of the main streams of efficient object detection is one-stage object detection, with well-known typical examples being the You Only Look Once (YOLO) [12] series of object-detection networks. One-stage object detectors perform both localization and classification on the same network [13], though this sacrifices the high detection accuracy of two-stage series networks and the rapid speeds of one-stage networks. Specifically, with approximately one hundred layers, YOLOv3 [5] achieves much higher accuracy than its earlier versions by utilizing a multi-scale feature interaction structure with shortcut connection and concatenation. Following YOLOv3's structure, YOLOv3-tiny [16] is customized in terms of efficiency given its fewer convolutional layers. YOLO-ReT [17], Edge YOLO [18], and YOLoC [19] have made tremendous efforts to benchmark variants of Tiny YOLO series for multiple input resolutions (i.e., 416×416, 320×320, or 256×256), with different options, i.e., the number of layers or the number of filters per layer.

In this work, we focus on the custom version of YOLOv3-tiny shown in Table I. Designed for multi-camera unmanned stores, the custom network accepts a 320×320×3 input image and outputs 20×20×195 and 10×10×195 tensors to detect and classify objects in sixty classes. It should be noted that the custom network requires 1.236 BFLOPs to process a single image while retaining the original structure of YOLOv3-tiny with hierarchical feature interaction with shortcut connection and concatenation. It is widely known that shortcut connection and concatenation layers require a larger memory footprint. The memory requirements for all of convolutional layers of this network are depicted in Figure 1 with respect to the amount of data movement. The unit of the y-axis is unspecified to reflect the fact that the actual size depends on the quantization scheme.

TABLE I
Custom YOLOv3-tiny Network Architecture

Layer	Type	Filter	Input	Output
0	conv	3*3*3*16	320*320*3	320*320*16
1	max		320*320*16	160*160*16
2	conv	3*3*16*32	160*160*16	160*160*32
3	max		160*160*32	80*80*32
4	conv	3*3*32*64	80*80*32	80*80*64
5	max		80*80*64	40*40*64
6	conv	3*3*64*128	40*40*64	40*40*128
7	max		40*40*128	20*20*128
8	conv	3*3*128*128	20*20*128	20*20*128
9	max		20*20*128	10*10*128
10	conv	3*3*128*128	10*10*128	10*10*128
11	max		10*10*128	10*10*128
12	conv	3*3*128*128	10*10*128	10*10*128
13	conv	1*1*128*195	10*10*128	10*10*195
14	yolo			
15	route	12		10*10*128
16	conv	1*1*128*128	10*10*128	10*10*128
17	upsample		10*10*128	20*20*128
18	route	17,8		20*20*256
19	conv	3*3*256*128	20*20*256	20*20*128
20	conv	1*1*128*195	20*20*128	20*20*195
21	yolo			

B. Edge-computing platforms

Edge GPUs and the power issue: Compared to desktop GPUs [9], edge GPUs such as NVIDIA Jetson Nano [10] and NVIDIA Jetson Xavier NX [11] significantly scale down the power by compromising performance. The thermal design power of NVIDIA Jetson Nano and Xavier NX are 10W and 15W, respectively, which is 15–25x less than that of the NVIDIA RTX 2080Ti, a high-end desktop GPU [18]. Moreover, according to a benchmark analysis of the performance of YOLOv3-tiny on these edge GPUs [14], NVIDIA Jetson Nano and Xavier NX require approximately 7W and 13W of power, respectively, to run half-precision YOLOv3-tiny network inference.

ASIC and FPGAs: CMOS application-specific integrated circuits (ASICs) offer the flexibility to customize CNN accelerators according to specific system requirements, thus providing an ideal platform to implement neural processing units [19]. However, implementing a dedicated ASIC requires a relatively long development time. As network architectures quickly evolve year by year, solutions offered by an FPGA are often considered promising alternatives for systems requiring rapid deployment at a reasonable cost. The advantages of FPGAs are their rapid prototyping, reconfiguration capabilities, and portability designs. These advantages have pushed research toward CNN accelerators based on FPGA devices, even if clear obstacles exist when compared to an ASIC design in which a CNN accelerator is specifically designed using a target technology. Particularly, many implementations of YOLOv3-tiny accelerators ([25], [26], [27], [28]) on FPGA devices have been proposed.

C. FPGA implementations of YOLO-tiny series and their limitations

Streaming-like architecture: One FPGA design of a YOLO-tiny accelerator is a streaming-like architecture in which each convolutional layer has its own MAC unit and all layers are

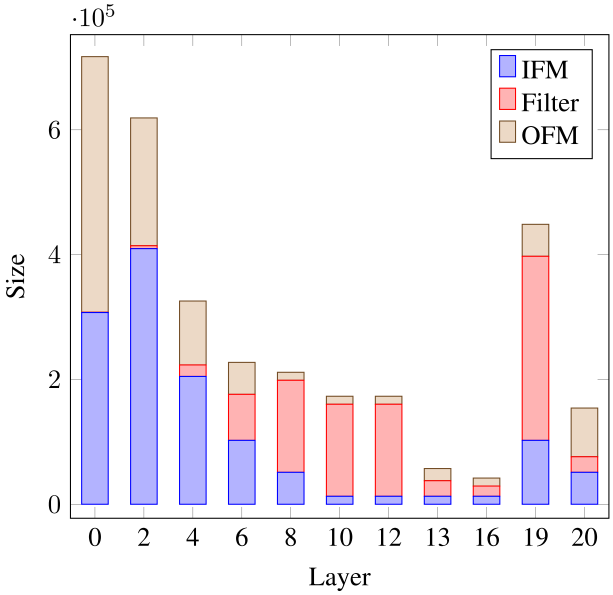

Fig. 1. Memory Requirement of Custom YOLOv3-tiny Network

pipelined using dual buffers. Consequently, the design requires a huge buffer size, which makes it suitable for binarized weight networks[15]. Meanwhile, the fused layer design [21] also pipelines a few early layers of a network to reduce external memory accesses. Unfortunately, this case is less effective at fusing computations of deep layers that have small feature maps, and it is not clear how computing units and buffers for early layers are utilized for the rest of a network.

Fixed dataflow design: Another typical design for a YOLO accelerator is to use a layer-by-layer processing strategy ([25], [26], [27], [28]). Unfortunately, the existing designs are associated with low hardware efficiency for two common reasons. First, they fail to map the computing units effectively to various types of convolution (i.e., conv3×3 and conv1×1 with different numbers of input and output channels). Consequently, an accelerator may not fully utilize all of the computing power (i.e., DSPs on an FPGA) or cannot effectively use the available computing power. Second, the existing designs incur many unnecessary data movements, such as those for large intermediate feature maps of early layers and shortcut connection and concatenation layers. Specifically, external memory accesses and high on-chip memory requirements represent data bottlenecks, causing computing units to be idle while data is being loaded.

Other accelerators: Several approaches have been proposed to reduce external memory access and maximize the reuse of on-chip data. Eyeriss [23] introduces a row-stationary dataflow scheme that reuses a row of the feature map against multiple rows of the filter. Alwani *et al.* [21] present a method that reduces off-chip memory transfers of the intermediate feature maps by modifying the order in which the input data is loaded. Other work [29] proposes a reuse-aware memory allocation scheme to support shortcut connections, achieving approximately 70% MAC efficiency for a given neural network. However, this method requires a large on-chip memory.

III. Design Methodology for Computing-Resource and Memory-Constraints

A. Overall Hardware Architecture

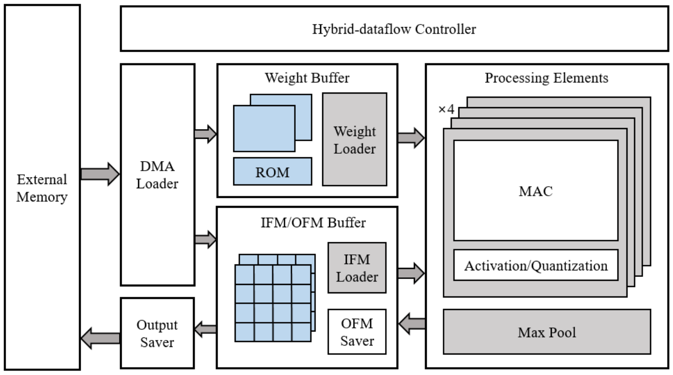

Fig. 2. Illustration of Accelerator Core

The proposed design processes a layer at a time instead of pipelining multiple layers due to limited resources, i.e., the limited number of DSPs. As mentioned in Section II, a layer-by-layer architecture is generally known for relatively low hardware utilization. To achieve high MAC efficiency, we propose a compound solution that includes (1) a MAC array with a multi-mode tree to map to various convolution configurations efficiently (Section III-B), (2) a dynamic data reuse scheme to reduce off-chip memory accesses (Section III-C), and (3) a buffer structure to minimize stalls when fetching data from the buffers to the MACs (Section III-D).

Figure 2 illustrates the microarchitecture of the accelerator engine, which consists of several typical modules, in this case a DMA loader, output saver, processing elements (PEs), buffers for weight, input feature maps (IFM) and output feature maps (OFM), and a controller. The DMA loader issues AXI read requests to fetch the input image and weights from the external memory. On-chip buffers are divided into two types: a weight buffer and an IFM/OFM buffer. The weight buffer includes a ROM initialized with weights of the early layers, eliminating the need for external memory accesses to retrieve these weights. IFM/OFM global buffers are implemented in sixteen banks, supporting sliding window operation without latency overhead. The weight loader and IFM loader modules manage the provision of the appropriate data to the PEs.

The PEs consist of four MAC arrays, activation/quantization modules, and a max pooling module. The four MAC arrays compute spatially adjacent pixels, meaning that they compute a 2×2 OFM as a whole in parallel. This configuration facilitates the pipelining of 2×2 max-pooling layers placed after convolutional layers. The activation/quantization module applies leaky-ReLU or linear activations and quantizes the OFM into the INT8 format. The OFM saver stores the OFM pixels generated by the PEs back into the IFM buffer for the next layers. The route layers and upsample layers are handled by this module as well. The output saver sends the output of layers 13 and 20 to the external memory by issuing AXI write requests. The hybrid-dataflow controller is an FSM that schedules the operations of all modules, enabling inter-module pipelining.

B. MAC Array with a Configurable Adder Tree

As shown in Table I, YOLOv3-tiny consists of 3x3 convolution and pointwise convolution operations, which account for 97.8% and 2.2% of the total computations, respectively. In addition, five 3x3 layers are followed by max-pooling layers that typically cause some stalls due to data mismatches. To address this issue, we construct MACs in four stacks, each of which is capable of computing 3×3×16 convolutions per cycle. For the 3x3 convolutional layers followed by max-pooling, the four stacks output a 2×2 tensor which can be processed by the max-pooling module without stalls.

Furthermore, to guarantee high hardware utilization for different types of convolutions, we propose three different modes for dataflows and MAC array operations.

1) 3×3 Mode: 3x3 convolution with more than 16 input channels is the most dominant form of the convolution operation in YOLOv3-tiny. In this case, the MAC array is provided with a 4×4×16 IFM window and the corresponding 3×3×16×1 weight every cycle. Each of the four MAC arrays performs 144 multiplications per cycle and sums all of the multiplication results, eventually producing a 1×1×1 output pixel. As a result, four MAC arrays as a whole produce 2×2×1 OFM pixels.

2) Layer 0 Mode: Layer 0 IFM has three input channels. Because each of the MAC arrays consists of 144 multipliers (the reasoning behind this number is explained below), letting each MAC array calculate a single output pixel would utilize only $\frac{3\times3\times3}{3\times3\times16}$ = 19% of the computational capacity. We address this low utilization issue by configuring each MAC array to perform the convolution between the 3×3×3 IFM window and 3×3×3×4 weight. In other words, the IFM window is replicated four times to be aligned with each of the four weights. Accordingly, each MAC array outputs 1×1×4 output pixels every cycle. In this way, hardware utilization for layer 0 reaches 75%.

3) Pointwise Convolution Mode: Layers 13, 16, and 20 require pointwise convolution. In this mode, we prepare a 1×1×16×8 weight and a 2×2×16 input feature map window. Each MAC array performs $(1 \times 1 \times 16) \otimes (1 \times 1 \times 16 \times 8)$ convolution. Under this configuration, each MAC array performs 128 multiplications per cycle, which translates to 88.9% utilization. Also, instead of adding all of the multiplication results, it outputs a 1×1×8 tensor. One should note that for maximum utilization, 1×1×16×9 weights can be convolved with a 1×1×16 window. However, we limit the number of output channels to eight for simplicity so that a 2×2×8 tensor is generated every eight cycles.

Figures 3, 4, and 5 depict the detailed hardware structure of the PE that can support the three modes of convolution. Each MAC array consists of a stack of multipliers and an adder tree that sums the multiplication results.

The number of INT8 multipliers in each MAC array in the PEs is set to 144. Thus, the total number of INT8 multipliers in the PE array is 576, as the PE array consists of four MAC arrays.

These numbers were chosen for the following reasons:

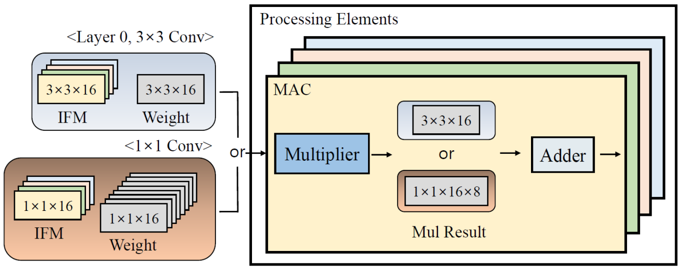

Fig. 3. Illustration of Processing Element

Dominant 3×3 convolution: 3×3 convolution accounts for 97.7% of all MAC operations in the YOLOv3-tiny network. Hence, setting the number of INT8 multipliers in each MAC array to a multiple of nine and mapping 3×3 convolution to each MAC array is a natural choice for maximizing both hardware utilization and throughput.

Channel size, a multiple of 16: The channel dimensions of all convolutional layers in YOLOv3-tiny are multiples of 16, except for layer 0. Therefore, it is natural to select the number of multipliers in each MAC array to be a multiple of 16. Thus, each MAC array consists of 3×3×16=144 INT8 multipliers.

Number of On-Chip DSP Slices and 2×2 Max-pooling: According to a Xilinx whitepaper [22], two INT8 multiplications with a shared operand can be mapped to a single DSP48 slice. Exploiting the fact that each filter weight is spatially reused in the sliding window operation, we effectively utilize each DSP slice as two INT8 multipliers in every convolutional layer. The Nexys A7-100T FPGA board is packed with 240 DSP slices. As each DSP slice can be mapped to two INT8 multipliers, a total of 480 INT8 multipliers can be instantiated with DSPs. At least four MAC arrays need to be instantiated for 100% DSP utilization. As 576 INT8 multipliers are needed for four MAC arrays, 480 multipliers are mapped to the DSPs, while the other 96 multipliers are mapped to LUTs. In our implementation, the four MAC arrays perform the convolution of four spatially-adjacent 3×3×16 windows(i.e., a 4×4×16 window), and thus generate 2×2 OFM pixels in parallel.

The adder tree is composed of eight accumulators to support the output of 1×1×8 OFM in the pointwise convolution mode (Figure 5). In the 3x3 convolutional layers, the output of the accumulators is summed again to form either a 1×1×1 OFM in 3x3 mode or a 1×1×4 OFM in layer 0 mode (Figure 4). In this way, the adders and accumulators are maximally reused across all configurations. Note that in pointwise convolution mode, each of the 20-input adders is supplied with only 16 inputs.

C. Dynamic Data Reuse Scheme

Convolutional layers, by their nature, possess many data reuse opportunities. For example, the sliding window operation enables the reuse of a filter weight against the entire input feature map (IFM), and the existence of multiple output channels allows the reuse of a single IFM window against many weights. However, the relative amounts of different reuse opportunities vary with respect to the layer configurations.

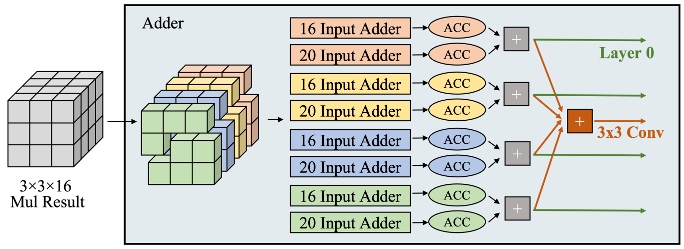

Fig. 4. Adder Configuration in Layer 0, 3x3 Convolution Mode

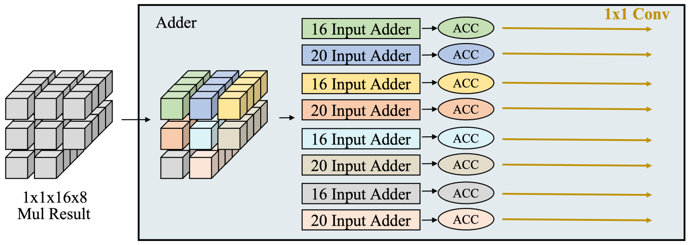

Fig. 5. Adder Configuration in Pointwise Convolution Mode

Moreover, the memory requirement associated with a specific reuse scheme varies as well. To minimize off-chip communication and stalls, a dynamic selection between the IFM reuse and weight reuse schemes based on layer specifications is more effective than employing a single reuse scheme for the entire network.

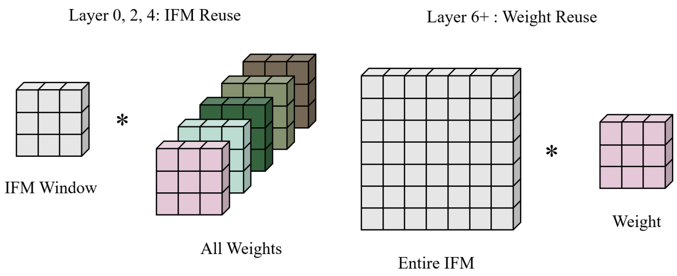

Fig. 6. IFM Reuse and Weight Reuse Schemes

1) Layers 0, 2, and 4 (IFM-Reuse + Fused Layers): The on-chip memory budget for the IFM buffer is set to 281.5KB considering the space to store the weights and a margin to ensure that the design meets the timing constraints. However, the IFM + OFM sizes of layers 0, 2, and 4 exceed 300KB (Figure 1).

Therefore, layer fusion [21] is applied to layers 0, 2, and 4 to meet the on-chip buffer requirement and minimize external memory accesses.

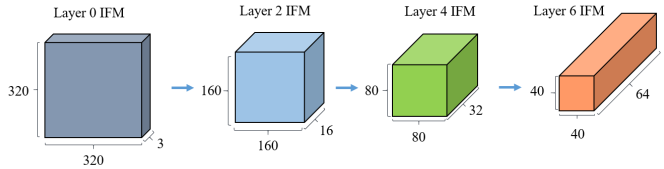

Fig. 7. Illustration of Conventional(Iterative) Forward Pass

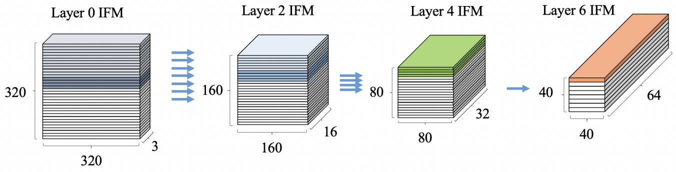

Fig. 8. Illustration of Fused-Layers Forward Pass

A difference between our implementation and the original fused layers technique introduced earlier [21] is that we do not modify the order in which the input data is loaded. In our design, data is loaded in a row-major order (Figure 8). Specifically, a 4×320×3 tensor is maintained for layer 0.

Performing convolution and max-pooling operations on this tensor yields a 1×160×16 tensor of the next convolution layer - layer 2. Once another two rows of the input image are loaded and the upper two rows from the tensor that had been loaded earlier are discarded, another row of the layer 2 feature map can be generated. After repeating this process until four rows of layer 2 feature maps are generated, convolution and max-pooling on this 4×160×16 tensor from layer 2 are performed, which yields a 1×80×32 feature map of layer 4. In this way, only four rows of each layer (which is 25KB) need to be stored in the on-chip buffer. The scheduling for the fused layers is shown in Figure 9.

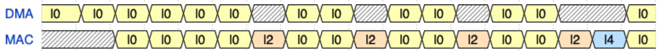

Fig. 9. Scheduling Schematic of Fused Layers

For the above pipelining scheme to incur no stalls, the following inequality must be satisfied for layer 0:

$$\frac{K \times K \times C_{in} \times C_{out} \times 2 \times W_{out}}{X} \geq \frac{2 \times W_{in} \times C_{in}}{Y} \quad (1)$$

The LHS of the inequality is the latency of computing two rows of the OFM, and the RHS is the latency of loading another two rows of IFM. Both LHS and RHS equal 640 cycles in our implementation. Additionally, external memory accesses occur only at layer 0, meaning that layer 2 and layer 4 forward passes incur no external memory accesses.

Each blob in Figure 9 represents either the loading of two rows of the IFM (DMA) or the computing of one row of the next layer IFM (MAC). Note that four rows of layer 2 IFM must be generated to start the layer 2 convolution process. Afterward, layer 2 convolution can take place for every two rows of layer 2 IFM generated. The same principle applies to layer 4.

2) Layers 6+: Weight-Reuse: From layer 6, the size of the weight increases significantly. Thus, the weights should be loaded from DRAM, and both the memory bottleneck and computational bottleneck should be considered. Figure 10 compares the weight-loading time and weight-consuming time at layers 6+.

The weight-consuming time, referring to the time required to reuse one weight fully, is obtained by dividing the number

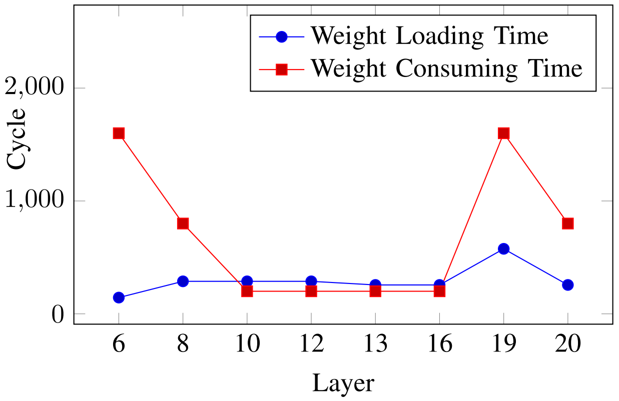

Fig. 10. Memory Bottleneck vs Computational Bottleneck

of multiplications when producing an output channel of the OFM by the number of multipliers, which is 576 in our implementation. The weight-loading time, the time for completely loading one weight through DMA, is obtained by dividing the size of a single weight by the DMA throughput - 4B per cycle. One should note that for pointwise convolution layers (layers 13, 16, 20), the weight-loading time is the latency of loading eight output channels of weights via DMA.

It can be observed in Figure 10 that the weight-consuming time is usually larger than the weight-loading time. Thus, no pipeline stall is incurred. However, pipeline stalls are inevitable in layers 10, 12, 13, and 16. However, the effects of these stalls are negligibly small, as the number of computations required for these layers is only 5.38% of the total.

D. Buffer organization

1) Input Feature Map Buffer: A key concept of the convolution operation is that the filter is convolved with a spatial window of the input feature map. To exploit this fact, our design consists of 16 spatially banked on-chip feature map buffers.

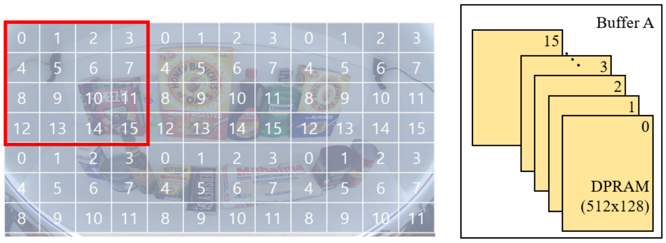

Fig. 11. Illustration of Spatially Banked Feature Map Buffers

In this buffer structure, feature maps are spatially divided into tiles and each bank corresponds to a spatial position in these tiles. For instance, buffer 3 in Figure 11 stores pixels corresponding to the following spatial positions: (0,3), (0,7), (0,11), (4,3), (4,7), and (4,11). Thus, each buffer acts as a bank that can be accessed independently, storing specific spatial feature map pixels. This enables random access of any 4×4 window in a single cycle and removes the need for additional logic for partial-sum saving and delivery.

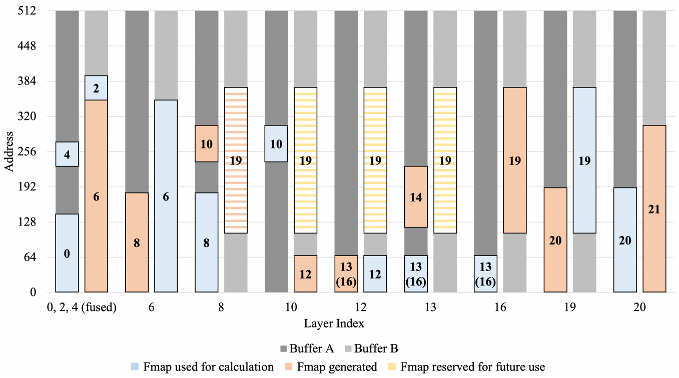

Fig. 12. IFM Buffer Allocation

Another advantage of the spatially banked buffer structure is that the 2x2 spatial upsample essentially becomes free. Whenever a 1×1 OFM pixel is generated, the upsampled 2×2 tensor can be saved to the feature map buffer by broadcasting the 1×1 tensor to the data bus of the four adjacent buffers (i.e. buffers $i, i+1, i+4, i+5$, where i can be 0, 2, 8, or 10.

Each buffer is implemented using dual-port on-chip BRAM slices. The 16 buffers as a whole consist of one IFM buffer array, consisting of 16 read and write ports. In our design, two buffer arrays are instantiated to double these ports. Doubling of the write ports is useful in that two OFMs can be saved simultaneously in layers with shortcut connections, such as layer 8.

The buffer space allocation scheme is shown in Figure 12. Note that during the processing of fused layers 0, 2, and 4, the allocated space for layer 0 is much larger than those for layers 2 and 4, as replication of the IFM window in layer 0 mode occurs when the image loaded from external memory is transferred to the buffer.

Shortcut connections are handled within the buffer as well. When storing layer 8 OFM into the buffer, each 1×1×128 tensor is spaced by 128 bytes. Later, each 1×1×128 tensor of layer 17 OFM is placed in those spacings. This removes extraneous latency overhead for the concatenation operation.

2) Weight Buffer: To support the three different modes of PE, the weight buffer must be able to provide weights of an identical shape to the corresponding IFM window. Thus, the weight buffer should provide a 3×3×4×4 weight for layer 0, a 3×3×16 weight for other 3×3 convolution layers, and a 1×1×16×8 weight for pointwise convolution layers. In our implementation, the global weight buffer BRAM width is set to 1152 bits. Note that for layers with channel sizes larger than 16, the weights are divided into multiple blocks of either 3×3×16 tensors (3×3 convolution) or 1×1×16×8 tensors (pointwise convolution). Each word of the weight buffer stores a block of weights, as shown in Figure 13.

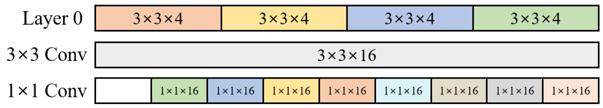

Fig. 13. Row Organization of Weight Buffer

IV. Evaluation

In this section, we present the performance of our accelerator and evaluate it by comparing it with some previous works. Another key aspect of our design is that external memory accesses are diminished owing to the use of fused layers. The effect of this technique is analyzed in subsection A. In subsection B, our accelerator's resource and power utilization rates are presented. In subsection C, we compare our work with other papers that introduce accelerators that also implement YOLOv3-tiny.

A. Dynamic Data Reuse Scheme

The effect of the dynamic data reuse scheme appears in both latency and buffer requirements.

To illustrate the effect of dynamic data reuse scheme, we compare our approach with the common approach of applying only the weight-reuse scheme throughout all layers. Such an approach requires the following inequality to hold in order to hide the DMA-loading latency for layer 0.

$$\frac{K \times K \times C_{in} \times 4 \times 2 \times W_{out}}{X} \geq \frac{2 \times W_{in} \times C_{in}}{Y} \tag{2}$$

The LHS of the inequality is the latency of computing four channels of the two OFM rows, and the RHS is the latency of loading another two rows of the input image from external memory. However, substituting the actual parameters of layer 0 into the LHS yields 160 cycles and the RHS equals 640 cycles. Thus, inequality 2 is not satisfied and a pipeline stall of 480 cycles is incurred for every two rows of the layer 0 OFM generated. The resulting latency increase due to the pipeline stall is 73200 cycles (0.732ms).

Dynamic data reuse can remove these stalls. Applying the IFM-reuse scheme to layer 0 increases the number of computations using the given IFM window. The pipelining inequality now becomes as follows:

$$\frac{K \times K \times C_{in} \times C_{out} \times 2 \times W_{out}}{X} \geq \frac{2 \times W_{in} \times C_{in}}{Y} \tag{3}$$

The LHS now becomes 640 cycles, whereas the RHS stays the same. In this way, the inequality is satisfied and no pipeline stall is incurred. As a result, removing the stalls using the dynamic data reuse scheme leads to an overall speedup of 5.89%.

More importantly, the weight-reuse scheme requires the entire IFM to be available in the buffer such that each weight can be convolved against all of the IFM windows. For YOLOv3-tiny, layers 0 and 2 have the largest IFM sizes - 300 KB and 400 KB, respectively. Considering the BRAM constraint of 607.5 KB on Nexys A7-100T board, on-chip BRAMs do not suffice to hold these feature maps. Thus, applying the weight-reuse scheme can result in more external memory accesses, which can deteriorate both the latency and the energy consumption.

The dynamic data reuse scheme relaxes the buffer requirement significantly, mainly because the IFM-reuse scheme allows the fused layers technique to be applied. With fused layers, only a part of the feature maps needs to be stored in the buffer at a given time frame. Figure 14 illustrates the required buffer sizes when the fused layer technique is applied to multiple layers. It can be observed that fusing layers 0, 2, and 4 minimizes the on-chip buffer requirement.

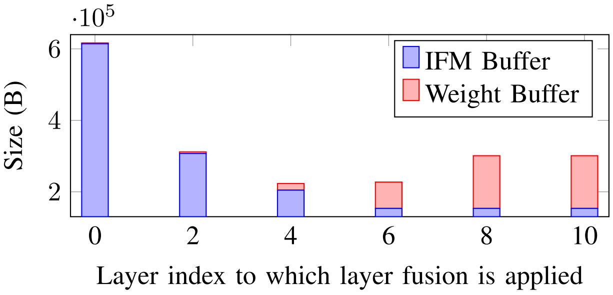

Fig. 14. Buffer Requirement as More Layers are Fused

By reducing the buffer requirement, external memory accesses can also be minimized. Instead of sending the entire set of feature maps of each layer into external memory, our design stores the currently required portion of the feature maps into the on-chip buffers. Figure 15 compares the amount of DRAM access upon the conventional layer-wise iterative forward pass versus that when the fused layers technique is applied. Under the fused layers technique, the IFM and OFM of layers 2 and 4 do not need to be transferred to and from the external memory. As a result, the overall amount of external memory access is reduced by 50%.

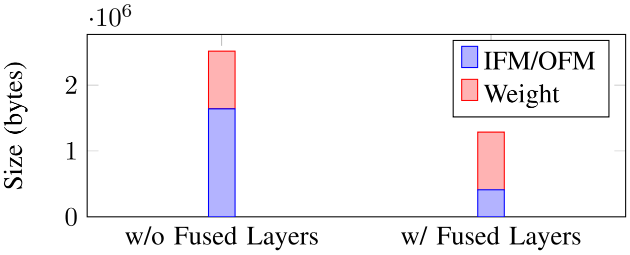

Fig. 15. Effect of Fused Layers to External Memory Access

B. Shortcut Connections and Concatenation

There are two shortcut connections in YOLOv3-tiny: layer 16 takes layer 12 OFM as input and layer 19 takes a concatenation of layer 17 OFM and layer 8 OFM. For both cases, OFMs with shortcut connections are stored in the IFM buffer. The OFMs of layers 8 and 12 each take up 19.5% and 4.9% of the IFM buffer space, respectively. Meanwhile, only 58.6% of the buffer space is occupied at maximum during the forward pass process between layer 8 and layer 19.

Without this technique, the OFMs of layers 8 and 12 would have had to be transferred to external memory and then retrieved when needed. As the sizes of the layer 8 OFM and layer 12 OFM are 50KB and 12.5KB, respectively, this would have required an additional 32,000 clock cycles for such external memory accesses, which could add 2.3% to the total latency.

C. MAC Array with a Configurable Adder Tree

Despite the fact that 3×3 convolution with a channel dimension larger than 16 is the most dominant operation in YOLOv3-tiny, other configurations account for 11.5% of the total computations. In 3×3 mode, the computing unit produces a 2×2×1 OFM at a time by accumulating the result of the convolution between four 3×3×16 IFM windows and the 3×3×16 weight. A naïve approach would be to let the computing unit output the same OFM shape for other layers as well. For layer 0, a convolution between four spatially adjacent 3×3×3 IFM windows and a 3×3×3 weight would produce a 2×2×1 OFM. Similarly, a convolution between four 1×1×128 IFM windows and a 1×1×128 weight would produce a 2×2×1 OFM for layer 13.

However, such an approach leads to the underutilization of the computing unit for these layers and results in a computational bottleneck. To resolve this issue, three different modes of the MAC array are proposed, as described in Section III. The latency effect of these three modes is visualized in Figure 16.

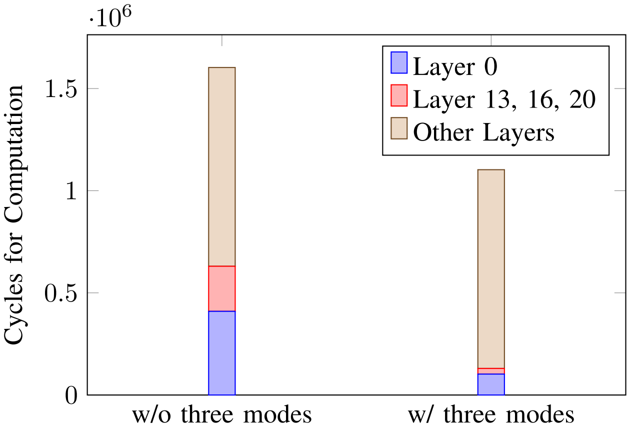

Fig. 16. Latency Effect of the Configurable Adder Tree

The overall speedup resulting from the three modes of the MAC arrays is 28.2%.

D. Power and Resource Utilization

Figure 17 presents the overall architecture of our system. The host PC sends commands and data to a Xilinx Microblaze processor through UART. Microblaze then accesses the DRAM or the accelerator engine according to the command it has received. The DRAM is accessed by Microblaze and the accelerator core via Xilinx MIG IP. Here, our accelerator engine is essentially an AXI IP that can be integrated into any other system.

Table II shows the power consumption of each module in Figure 17. The accelerator engine, which is the core module of our system, accounts for 46.6% of the power.

Table III presents the resource utilization of our system. In total, 78.6% LUTs, 45.8% FFs, 68.5% BRAMs, and 100% DSPs of board resources are utilized. The Accelerator Engine accounts for 62.6% LUT, 33.0% FF, 62.6% BRAM, and 100% DSP of the board resources in total.

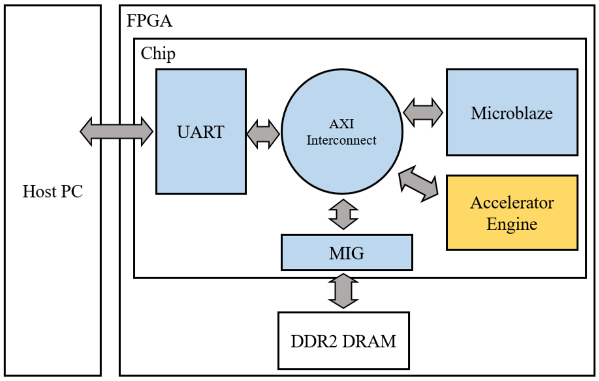

Fig. 17. Illustration of the System

TABLE II
POWER CONSUMPTION

Module	Power	% of Total
Accelerator Engine	0.97W	46.6%
MIG	0.89W	42.8%
Microblaze	0.04W	1.9%
UART	0.001W	0.05%
Clock Generator	0.11W	5.3%
Miscellaneous	0.069W	3.3%
Total	2.08W	100%

TABLE III
RESOURCE UTILIZATION

Module	LUT	% of Total	FF	% of Total	BRAM36K	% of Total	DSP
Available on Board	63400		126800		135		240
Accelerator Engine	39693	79.7%	41892	72.1%	84.5	91.4%	240
MIG	5872	11.8%	8871	15.3%	0		0
Microblaze	4027	8.1%	6816	11.7%	8	8.6%	0
UART	151	0.3%	379	0.7%	0		0
Miscellaneous	85	0.2%	140	0.2%	0		0
Total	49828	100%	58098	100%	92.5	100%	240

E. Comparison With Other Works

We validate our design on a Digilent Nexys A7-100T board. In total, we utilize 185 slices of 18K BRAMs, 240 DSPs, 50.2K LUTs, and 58.1K flip-flops. Our design is synthesized and implemented using Xilinx Vivado. At a clock frequency of 100 MHz, the accelerator shows power consumption of 2.203 W. The accelerator achieves 76.75 FPS, which is equivalent to latency of 13ms per image. Our design shows a throughput of 95.08 GOPS and a throughput efficiency rate of 43.16 GOPS/W. It achieves 82.53% of the maximum achievable throughput with the 576 MACs (=2*576*100 MHz=115.2 GOPs).

Table IV shows the resource utilization in detail along with those from other studies [25], [26], [27], [28]. Yu *et al.* [25] implement a parameterized FPGA-tailored architecture specialized for YOLOv3-tiny. However, its latency is 532ms which is not suitable for edge-computing devices. Adiono *et al.* [26] propose a YOLOv3-tiny accelerator with a clock frequency of 250 MHz on Ultra96 V2, for which the board utilization rate is fairly close to ours. It transforms 2D con-

TABLE IV
PERFORMANCE COMPARISON TO OTHER YOLOV3-TINY FPGA ACCELERATOR

	[25]	[26]	[27]	[28]	This work
Year	2020	2021	2021	2020	2022
Model	YOLOv3-tiny	YOLOv3-tiny	YOLOv3-tiny	YOLOv3-tiny	YOLOv3-tiny
Input Width	416	448	416	416	320
Quantization	16b	8b	16b	18b	8b
Clock	100MHz	250MHz	100MHz	200MHz	100MHz
FPGA	Zedboard	Ultra96 V2	ZYNQ-7035	Virtex-7 VC707	Nexys A7-100T
Board Cost	$499	$249	$1499	$5244	$265
BRAM	185	248	248	141	185
DSP	160	242	485	2304	240
LUT	25.9k	27.3k	N/A	48.6k	50.2k
FF	46.7k	38.5k	N/A	93.2k	58.1k
Latency	532ms	121ms	192ms	N/A	13ms
GOPS	10.45	31.50	28.99	460.8	95.08
Power	3.36W	4.26W	3.71W	4.81W	2.203W
GOPS/DSP	0.0653	0.130	0.0598	0.200	0.396
GOPS/W	3.11	7.40	7.81	95.80	43.16
MAC Utilization	32.65%	13.01%	29.89%	50.00%	82.53%

volution via GEMM; however, a drawback of this approach is that the feature maps must be replicated many times, which represents overhead both in terms of the throughput and memory space. As a result, the method of Adiono *et al.* [26] shows high power consumption and relatively low computational throughput despite the 2.5x clock. Xiong *et al.* [27] employ high-level synthesis (HLS) to implement the pipelining mechanism. Because HLS fails to capture and exploit the model and hardware-specific characteristics fully, the accelerator's throughput falls behind its maximum achievable throughput. Ahmad *et al.* [28] achieve an impressive computational throughput via 2304 DSP slices. Their method uses a highly pipelined and efficient design through a hardware and software co-design methodology. However, its MAC utilization is only 50% due to the redundant off-chip communication between the PL and PS.

V. CONCLUSION

Various techniques that can be employed to achieve an efficient hardware accelerator design for YOLOv3-tiny are explored. The configurable adder tree structure allows high hardware utilization for convolutions of different kernel sizes, and the dynamic data reuse scheme relaxes the buffer requirement for a forward pass, resulting in less external memory access. The spatially banked feature map buffer enables a continuous stream of data to be provided to the PEs and virtually removes the overhead for shortcut connections and upsampling. These techniques are scalable and can be generalized to other convolutional neural networks as well.

ACKNOWLEDGMENT

This work was supported in part by the R&D Program of MOTIE/KEIT (No. 20010582, Development of deep learning based low power HW IP design technology for image processing of CMOS image sensors) and in part by the Technology Innovation Program (No. 20011074, Development of Open Convergence Memory Solution and Platform for Next Generation Memories) funded by the Ministry of Trade, Industry & Energy (MOTIE, Korea).

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[2] Azzedine Boukerche and Zhijun Hou, "Object Detection Using Deep Learning Methods in Traffic Scenarios," *ACM Computing Surveys*, March 2021.

[3] Eduardo Arnold, Omar Y. Al-Jarrah, Mehrdad Dianati, Saber Fallah, David Oxtoby, Alex Mouzakitis, "A Survey on 3D Object Detection Methods for Autonomous Driving Applications," *IEEE*, January 2019.

[4] Shi-Jinn Horng, Pin-Siang Huang, "Building Unmanned Store Identification Systems Using YOLOv4 and Siamese Network," *Appl.Sci*, 10 April 2022.

[5] Redmon, Joseph, and Ali Farhadi. "Yolov3: An incremental improvement," *arXiv preprint arXiv*:1804.02767 (2018).

[6] Alexey Bochkovskiy, et al "YOLOv4: Optimal Speed and Accuracy of Object Detection," *arXiv*, 23 April 2020.

[7] Mingxing Tan, et al "EfficientDet: Scalable and Efficient Object Detection," *IEEE/CVF*, 2020.

[8] Vittorio Mazzi, Aleem Khaliq, et al "Real-Time Apple Detection System Using Embedded Systems With Hardware Accelerators: An Edge AI Application," *IEEE*, 07 January 2020.

[9] NVIDIA Geforce RTX 2080Ti. Available online: https://www.nvidia.com/en-me/geforce/graphics-cards/rtx-2080-ti/ (accessed on 30 September 2022).

[10] NVIDIA Jetson Nano. Available online: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/ (accessed on 30 September 2022).

[11] NVIDIA Jetson Xavier NX. Available online: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/ (accessed on 30 September 2022).

[12] Joseph Redmon, Santosh Divvala, et al "You Only Look Once: Unified, Real-Time Object Detection," *IEEE/CVF*, 2016.

[13] Aditya Lohia, et al "Bibliometric Analysis of One-stage and Two-stage Object Detection," *Libr. Philos*, 2021.

[14] Feng, Haogang, Gaoze Mu, Shida Zhong, Peichang Zhang, and Tao Yuan. "Benchmark analysis of Yolo performance on edge intelligence devices." Cryptography 6, no. 2 (2022): 16.

[15] Quang Hieu Vo, et al "A Deep Learning Accelerator Based on a Streaming Architecture for Binary Neural Networks," *IEEE*, 15 February 2022.

[16] Adarsh, Pranav, Pratibha Rathi, and Manoj Kumar. "YOLO v3-tiny: Object Detection and Recognition using one stage improved model," *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*. IEEE, 2020.

[17] Ganesh, Prakhar, et al. "YOLO-ReT: Towards high accuracy real-time object detection on edge GPUs," *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2022.

[18] Liang, Siyuan, et al. "Edge YOLO: Real-time intelligent object detection system based on edge-cloud cooperation in autonomous vehicles," *IEEE Transactions on Intelligent Transportation Systems*. 2022.

[19] Chen, Yiming, et al. "YOLoC: deploy large-scale neural network by ROM-based computing-in-memory using residual branch on a chip." *arXiv preprint arXiv*:2206.00379 (2022).

[20] Jacob, Benoit, et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.

[21] Alwani, Manoj, et al. "Fused-layer CNN accelerators," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[22] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig, "Deep Learning with INT8 Optimization on Xilinx Devices," (https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8)

[23] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016.

[24] ——, "Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators," *IEEE Micro's Top Picks from the Computer Architecture Conferences*, vol. 37, no. 3, May-June 2017.

[25] Yu, Zhewen, and Christos-Savvas Bouganis. "A parameterisable FPGA-tailored architecture for YOLOv3-tiny," *International Symposium on Applied Reconfigurable Computing*. Springer, Cham, 2020.

[26] T. Adiono, A. Putra, N. Sutisna, I. Syafalni and R. Mulyawan, "Low Latency YOLOv3-tiny Accelerator for Low-Cost FPGA Using General Matrix Multiplication Principle," in *IEEE Access*, vol. 9, pp. 141890-141913, 2021, doi: 10.1109/ACCESS.2021.3120629.

[27] Xiong, Qilin, et al. “A Method for Accelerating YOLO by Hybrid Computing Based on ARM and FPGA,” *2021 4th International Conference on Algorithms*, Computing and Artificial Intelligence. 2021.

[28] Ahmad, Afzal, Muhammad Adeel Pasha, and Ghulam Jilani Raza. “Accelerating tiny YOLOv3 using FPGA-based hardware/software co-design,” *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020.

[29] Nguyen, Duy Thanh, et al. “ShortcutFusion: From Tensorflow to FPGA-Based Accelerator With a Reuse-Aware Memory Allocation for Shortcut Data,” IEEE Transactions on Circuits and Systems I: Regular Papers 69.6 (2022): 2477-2489.