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

Minsik Kim†, Kyoungseok Oh†, Youngmock Cho†, Hojin Seo,
Xuan Truong Nguyen, and Hyuk-Jae Lee, *Member, IEEE*

***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, Object detection, 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 [1], autonomous driving [2], and unmanned stores [3]. 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 [4], YOLOv4 [5], and EfficientDet [6] 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 [7], NVIDIA Jetson Xavier NX [8]). 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 [9]–[12] have been proposed. In one study [10], the convolution operation is implemented on the FPGA using the General Matrix Multiplication Principle (GEMM). Other work [11] 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 [12]. However, it is challenging to utilize the computing resources fully due to external memory accesses and buffer reuse, leading to low hardware utilization, typically less than 50%. [10] and [11] fail to utilize the DSPs in the Processing Element (PE) fully, leading to low throughput and high latency, rendering them not suitable for real-time applications. Another method [12] requires extraneous off-chip communication, as the Processing System must send the feature map and the weights to the reserved addresses of the DRAM and signal the Programmable Logic to start the convolution when the convolution operation needs to be performed. As external memory access acts as a primary bottleneck, Nguyen *et al.* [13] presents a method to reduce off-chip communication by applying mixed dataflow. However, this method requires large on-chip BRAM footprint ($\geq$ 8MB) and thus is not suitable for low-cost devices. In addition, ShortcutFusion [14] presents a reuse-aware memory allocation that minimizes the external memory transactions for the shortcut layers. Its fixed PE structure, however, limits its MAC utilization for different layers with different shapes of weights.

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

† M. Kim, K. Oh, and Y. Cho contributed equally.

This work is supported in part by the Technology Innovation Program (or Industrial Strategic Technology Development Program – No. 20014490, Development of Technology for Commercializing Lv.4 Self-driving Computing Platform Based on Centralized Architecture) funded by the Ministry of Trade, Industry & Energy (MOTIE, Korea) and in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2021-0-00106, AI accelerator-optimized neural network automatic generation technology and open service platform development). (*Corresponding author: Xuan Truong Nguyen.*)

M. Kim is with the Department of Electrical Engineering and Computer Sciences, Univeristy of Michigan-Ann Arbor, MI 48109, United States (e-mail: minsikky@umich.edu). K. Oh, Y.Cho and H. Seo are with the Department of Electrical and Computer Engineering, College of Engineering, Seoul National University, Seoul 08826, South Korea (e-mail: ehxhfl1000, fudsla, shj2044409@snu.ac.kr). Hyuk-Jae Lee is with the Inter-University Semiconductor Research Center (ISRC) and the Department of Electrical and Computer Engineering, College of Engineering, Seoul National University, Seoul 08826, South Korea (e-mail: hyuk_jae_lee@capp.snu.ac.kr). Xuan Truong Nguyen is with the Department of Next-generation Semiconductor Convergence and Open Sharing System (COSS) and also with the System Semiconductor for AI Engineering (SSAI) program, the Department of Electrical Engineering, Seoul National University, Seoul 08826, South Korea (e-mail: truongnx@snu.ac.kr).

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 - input feature map (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.

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 reviews related works and the background. Section 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) [15] series of object-detection networks. Specifically, with approximately one hundred layers, YOLOv3 [4] 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.

TABLE I
Custom YOLOv3-tiny Network Architecture

Layer	Type	Filter	Input	Output	GOP
0	conv	3*3*3*16	320*320*3	320*320*16	0.088
1	max		320*320*16	160*160*16	
2	conv	3*3*16*32	160*160*16	160*160*32	0.236
3	max		160*160*32	80*80*32	
4	conv	3*3*32*64	80*80*32	80*80*64	0.236
5	max		80*80*64	40*40*64	
6	conv	3*3*64*128	40*40*64	40*40*128	0.236
7	max		40*40*128	20*20*128	
8	conv	3*3*128*128	20*20*128	20*20*128	0.118
9	max		20*20*128	10*10*128	
10	conv	3*3*128*128	10*10*128	10*10*128	0.029
11	max		10*10*128	10*10*128	
12	conv	3*3*128*128	10*10*128	10*10*128	0.029
13	conv	1*1*128*195	10*10*128	10*10*195	0.005
14	yolo				
15	route	12		10*10*128	
16	conv	1*1*128*128	10*10*128	10*10*128	0.003
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	0.236
20	conv	1*1*128*195	20*20*128	20*20*195	0.020
21	yolo				

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. 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, which require a larger memory footprint.

B. Edge-computing Platforms

Edge GPUs and the Power Issue: Compared to desktop GPUs [20], edge GPUs such as NVIDIA Jetson Nano and NVIDIA Jetson Xavier NX significantly scale down the power by compromising performance. The thermal design powers 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 [21], 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. To this end, many implementations of YOLOv3-tiny accelerators [9]–[12] on FPGA devices have been proposed.

C. FPGA Implementations of CNN Accelerators and Their Limitations

Quantization: Mapping a deep neural network into (low-cost) FPGA boards is generally challenging due to their limited

computing resources, i.e., the number of DSPs and BRAMs. Quantization is widely used to reduce DRAM access, memory storage, and energy [22]–[24]. Two common quantization types are quantization-aware training (QAT) [13], [22], [25], [26], and post-training quantization (PTQ) [9]–[12], [14], [27]. In QAT, weight and activation quantization is applied during training to mitigate accuracy degradation, resulting in low-bit representation, such as 5 bits for both weights and activations [25], or mixed one-bit and eight-bit weights [13]. However, QAT requires both retraining and training datasets which may not be disclosed due to privacy. Consequently, PTQ is generally more favorable for rapid network deployment on devices [14], [27]. To avoid a significant computation drop, [27] uses 14 bits for both weights and activations, which requires a considerable amount of DSPs, LUTs, and BRAM, i.e., 1920 DSPs [27]. [14] uses eight bits for both weights and activations with channel-wise scaling or (batch) normalization and a dynamic fixed-point format that requires more resources, i.e., multipliers.

Streaming-like Architecture: One FPGA design of a CNN accelerator is a streaming-like architecture in which each convolutional layer has its own PEs, and all layers are pipelined using dual buffers. Stacking and pipelining multiple layer-wise PEs requires many hardware resources, i.e., 1920 DSPs [27] or 1142 BRAMs [28]. *Such a design is unsuitable for resource-constrained FPGA devices such as Digilent Nexys A7-100T [29], Zedboard [9], and Ultra96 V2 [10]*. Therefore, this approach may be only suitable for binarized networks [28], [30], or a tiny and highly customized network (i.e., 2560 parameters) [27]. Meanwhile, the fused layer design [31] pipelines a few early layers to reduce external memory accesses, especially for input/output feature maps (IFM/OFM) that can be forwarded from one layer to the next without accessing external memory. When the size of weights is dominant, for example, layers 8, 10, 12, and 19, layer fusion is less effective in reducing external memory access for IFMs and OFMs.

Fixed Dataflow Design: Another typical design for a CNN accelerator is to use a layer-by-layer processing strategy ([9]–[12]). Unfortunately, the existing designs are associated with low hardware efficiency for two 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).* For example, [9] and [32] utilize a fixed-size 9×14 systolic array and a 7×7×8 array of multipliers, respectively, that are not fully utilized for conv3×3 and conv1×1, especially conv3×3. As a result, [9] only achieves 32.65% MAC utilization for processing the YOLOv3-tiny network. *Second, the existing designs incur unnecessary data movements, such as those for sizeable IFMs/OFMs of early layers and shortcut connection and concatenation layers.* [14] utilizes a shortcut buffer to reduce external memory accesses for IFMs and OFMs effectively. However, the size of OFMs in shortcut layers, for example, layers 16 and 19 in Table I, may be much smaller than weights, while the shortcut buffer is not utilized for layers with large IFMs and OFMs, such as layers 0, 2, and 4 that account for 71.7% of the total IFM and OFM sizes. Meanwhile, [32] uses a three-layer structure with three PEs to reduce external memory accesses for IFMs and OFMs. The numbers of PEs are determined by balancing the number of computation requirements for three convolution layers, for example, layers 0, 2, and 4. However, the PE allocation given by balancing the workloads of layers 0, 2, and 4, may be highly inefficient for other layer groups.

Other Accelerators: Several approaches have been proposed to reduce external memory access and maximize the reuse of on-chip data. Eyeriss [33] introduces a row-stationary dataflow scheme that reuses a row of the feature map against multiple rows of the filter. Other work [14] 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 large on-chip memory. In addition, REQ-YOLO [26], a YOLOv2-tiny accelerator, requires more than 3000 DSPs and 1300 BRAMs, while a MobileNetv2 accelerator [34] utilizes 2226 INT8 multipliers. An earlier work [13] proposes a mixed dataflow, which is a combination of streaming-like architecture and layer-by-layer processing that reduces the transfer of intermediate feature maps to the off-chip memory.

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

In resource-constrained environments, optimizing both software and hardware is crucial for effective mapping and maximizing hardware utilization. This section outlines our accelerator's architecture and the applied optimization techniques.

A. Optimization in Software for Hardware Mapping

1) Batch Normalization Fusion and Leaky-ReLU Modification: In YOLO networks, a convolution layer consists of convolution, batch normalization, and an activation function (linear or leaky-ReLU). However, batch normalization requires a hardware-expensive division. Similar to other works [30], [35], batch normalization operations are fused into the weights and biases of a convolutional layer[1]. Furthermore, for the leaky-ReLU operation, the negative slope is set to 0.125 instead of 0.1, resulting in an efficient hardware implementation with a 3-bit right shift.

2) Hardware-friendly Quantization: Like [14], [22], [35], our design targets integer-arithmetic-only inference where both weights and activations are represented in integers or eight-bit fixed-point formats. Specifically, we focus on a PTQ method because the training dataset is unavailable due to privacy. Different from QAT [13], [22], [25], a PTQ method may suffer from a significant accuracy drop. To address the problem, we propose a simple yet effective PTQ method. Specifically, we statistically analyze the ranges of weights and activations and determine their layer-wise ranges to balance a quantization step and outliers and then mitigate the accuracy drop. Fig. 1 shows the dynamic fixed-point formats of weights and activations for all layers (See Appendix A for the detailed profiling results). Notably, our proposed quantization captures

[1]https://github.com/AlexeyAB/yolo2_light/blob/master/src/additionally.c (Accessed on 2022.1.9)

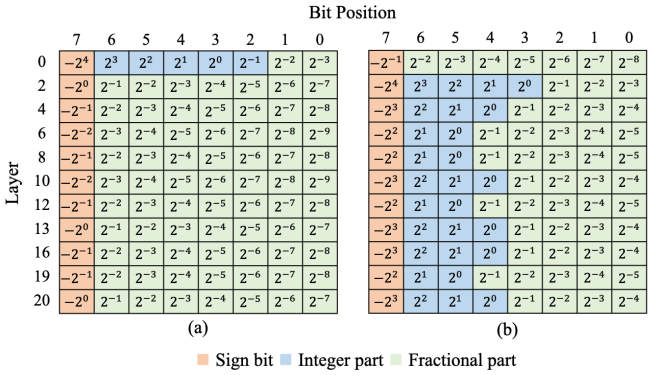

Fig. 1. Dynamic Fixed-point 8-bit Quantization for (a) Weights and (b) Activations (best viewed in color).

data distribution of weights and activations effectively and achieves a mean average precision (mAP) of 81.19% for 150 test images of 60 object classes with a 2.01% decrease in mAP compared to the original floating-point model. More importantly, different from [14], [35], all scaling factors are set to powers of two, leading to an efficient implementation of quantization and activation in the PE using a shifter instead of an expensive multiplication, as shown in Fig. 3. Under strict hardware constraints, we do not extend our layer-wise quantization approach into a channel-wise quantization method like [14] or [23] to avoid storage and logic overheads, although our channel-wise quantization could achieve an mAP drop of 1.48%.

B. Proposed Design Methodologies

Based on observations in Section II, we design our accelerator under three crucial considerations to achieve a low-latency and high PE utilization design.

1) Unified Processing Elements with a Configurable Adder Tree: Different from a streaming-like architecture [27], [28] and a pipeline PE architecture [25], the proposed accelerator consists of a unified PE array to execute a single layer at a time. Instead of a fixed structure as in early studies [9], [14], [32], the proposed PE design could flexibly reshape input/output tensors dynamically to ensure high PE utilization when executing conv3×3, conv1×1, or the first convolution layer (See Section III-C for details).

2) Flexible Layer-wise Mapping and Dataflow: In contrast to a streaming-like architecture [27], [35] and a pipeline PE architecture [25], executing in a layer-by-layer manner can incur large external DRAM accesses to load IFMs and store OFMs. To address the problem, we fuse operations of some (early) convolutional layers (i.e., layers 0, 2, and 4) and execute later convolutional layers one by one. Notably, the number of filters in the early layers is relatively small and can be saved in read-only memory (ROM) to avoid bubble cycles for weight preloading. More importantly, as opposed to earlier methods [25], [31], the computations for layers 0, 2, and 4 are interleaved but not pipelined. (See Section III-D for details).

3) Spatial Buffer Structure with Dynamic Access Patterns: To support various layer configurations (a) and flexible layer-wise mapping and dataflow (b), a new buffer structure that avoids bubble cycles when loading IFMs or storing OFMs is crucial. To meet this requirement, we propose a novel buffer structure that consists of 4×4 spatially banked on-chip SRAMs for feature maps. The buffer structure allows for the loading of a 4×4×16 IFM tensor for conv3×3 execution without bubble cycles (See Section III-E for details).

C. Unified Processing Elements with a Configurable Adder Tree

To achieve a low-latency and high PE utilization design, it is crucial to design a unified yet "flexible" PE array for effectively executing various types of convolution (i.e., conv3×3 and conv1×1 with different numbers of input and output channels in YOLO-like models [4], [15]–[18]). Specifically, YOLOv3-tiny in Table I consists of 3×3 convolution and pointwise convolution operations, which account for 97.8% and 2.2% of the total computations, respectively. Therefore, we propose to construct PE arrays of $3\times3\times T_r\times T_c\times T_i\times T_o$ MACs or multipliers for a given FPGA board. Here, T_r, T_c, T_i, and T_o are *row-wise, column-wise, input-channel, and output-channel-wise* factors that are used to calculate a $T_r\times T_c\times T_o$ tile output tensor computed by the MAC in one cycle. In our design, T_r and T_c are set to two, and T_o and T_i are dynamically selected with respect to layer configurations for high throughput and high PE utilization.

1) High Throughput and Low Latency: To achieve a high throughput design, the product ($T_r\times T_c\times T_i\times T_o$) is designed to fully utilize the computing resources on an FPGA board, i.e., DSPs and LUTs. In particular, for a Nexys A7-100T FPGA board, we choose ($T_r\times T_c\times T_i\times T_o$) as 64 to form a PE array of 576 INT8 multipliers. Given that each DSP slice can be mapped to two INT8 multipliers [36], 480 multipliers are mapped to the DSPs with 100% DSP utilization. Meanwhile, the other 96 multipliers are mapped to LUTs. Notably, with 576 MACs, the design achieves a peak throughput of 112 GOPS at the operating clock frequency of 100MHz.

2) High PE Utilization: Although the product ($T_r\times T_c\times T_o\times T_i$) is fixed, T_o and T_i are dynamically changed according to the shape of a layer. Specifically, 576 MACs are divided into four MAC arrays, where each MAC array consists of four MAC20 blocks and four MAC16 blocks, followed by an adder tree and a multiplexer, as shown in Fig. 2. MAC20 and MAC16 include twenty and sixteen parallel multipliers, respectively. As a result, the eight MAC units can process 144 multiplications to generate eight partial summation results simultaneously. Since eight partial summation results can be processed by a configurable accumulator-multiplexer structure, as shown in Fig. 2(b), our MAC design effectively supports all three convolution types in YOLO-like models, including conv3×3, conv1×1, and the first-layer convolution.

3) MAC Operations for Three Modes of Convolution:

a) 3×3 Mode: 3×3 convolution with more than 16 input channels is the most dominant form of the convolution operation in YOLOv3-tiny, i.e., 97.8% of the total computation. For the 3×3 mode, we design $T_i = 16$ and $T_o = 1$. In this case, the PE 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 convolution between a 3×3×16 IFM window extracted from the 4×4×16 window and the 3×3×16×1 weight per cycle (Conv3×3 case in Fig. 2(b)), eventually producing a 1×1×1 output pixel. In each MAC array, eight partial summation results from MAC20s and MAC16 are summed up to produce one output pixel, utilizing all multipliers and adders (top input of the output multiplexer in Fig. 2(a)). As a result, the four MAC arrays as a whole produce 2×2×1 OFM pixels. Notably, a 3×3 layer, i.e., five 3×3 layers in YOLOv3-tiny, is typically followed by max-pooling layers that can typically cause additional latencies due to data mismatches. For the 3×3 convolutional layers followed by max-pooling, with the four MAC stacks producing a spatial 2×2×1 tensor, the max-pooling operation can be fused into the convolutional layer with no additional buffering.

b) Layer 0 Mode: As the number of input channels C_{in} of layer 0 IFM is 3, $T_i \leq C_{in} = 3$ for layer 0. To ensure high PE utilization, we set $T_o = 4$ so that each MAC array performs 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. Layer 0 case of Fig. 2(b) shows this alignment. Then, each convolution between the 3×3×3 IFM window and the 3×3×3×1 weight is mapped to a pair consisting of a MAC16 and a MAC20 unit, as shown in Fig. 2(a). The outputs of the MAC units within a pair are then summed (middle input of the output multiplexer in Fig. 2(a)). Accordingly, each MAC array outputs 1×1×4 output pixels every cycle. In this way, the PE utilization for layer 0 can reach up to 75%.

c) Pointwise Convolution Mode: Besides 3×3 layers, a pointwise layer, such as layers 13, 16, and 20 in YOLOv3-tiny in Table I, is widely used in YOLO-like models. In this case, we configure $T_i = 8$ and $T_o = 8$. Each MAC array is designed to take 1×1×16×8 weight and a 2×2×16 input feature map window, and then performs $(1 \times 1 \times 16) \otimes (1 \times 1 \times 16 \times 8)$ convolution (Conv1×1 case in Fig. 2(b)). Under this configuration, each of the MAC16 and MAC20 units are provided with 16 inputs, causing underutilization for MAC20 units. 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. In our implementation, 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 (bottom input of the output multiplexer in Fig. 2(a)).

Note that our decision to use a specific number of MAC arrays and their size is heavily dependent on the available on-chip resources. If additional resources are available, we could potentially increase the size of the convolution window, such as moving from a 3×3×16 to a 3×3×32, or even increase the number of MAC arrays to exploit greater parallelism and generate more OFM pixels simultaneously. Conversely, if we were implementing the accelerator on an FPGA board with limited resources,the number of MAC arrays and/or the convolution window sizes could be reduced to work within the constraints of the available resources.

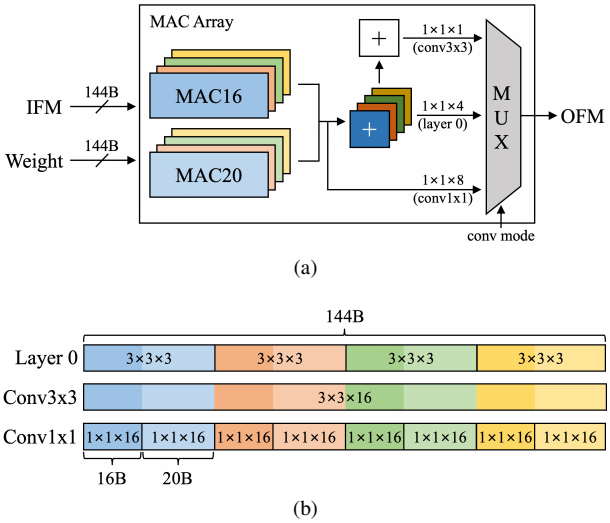

Fig. 2. (a) Hardware Structure of the Proposed Unified MAC Array. (b) Reconfiguring the Unified MAC Array Input for Three Convolution Modes (best viewed in color).

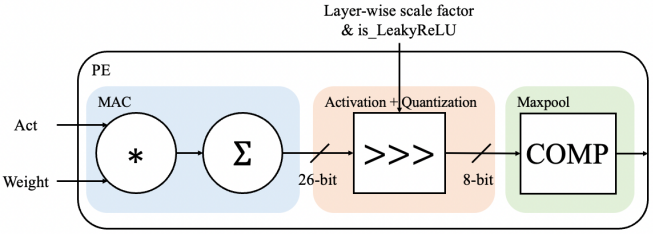

Fig. 3. Proposed PE Structure with Shift-only Activation, Quantization, and No-buffering Max-pooling.

D. Dynamic Data Reuse

Execution of a convolutional layer can be represented by seven nested loops, as shown in lines 3-9 of Fig. 4. As noted in Section III-C, $3 \times 3 \times T_r \times T_c \times T_i \times T_o$ MACs produce a $T_r \times T_c \times T_o$ output. While the inner loops to be unrolled that dictate the computations for each cycle are determined by the structure of the PEs, the interchange between the outer loops also needs to be considered to achieve low latency and reduce the on-chip buffer requirements.

For this, we observe that layers in the YOLOv3-tiny model can be divided into two parts: the early layers (layers 0, 2, and 4) that have dominant feature map sizes over weights ($H_{in}W_{in} >> K^2C_{out}$), and deeper layers (layers 6+) that have larger weights ($K^2C_{out} > H_{in}W_{in}$). Here, K refers to the size of the filter (3 for conv3×3 and 1 for conv1×1), and H_{in} (H_{out}), W_{in} (W_{out}), C_in (C_{out}) are height, width, and channel dimensions of the input (output) feature map of a layer. This observation motivates two different loop orderings: IFM-reuse (Fig. 4(a)) and Weight-reuse (Fig. 4(b)). In the IFM-reuse scheme, an IFM window is convolved with all weights before sliding to the adjacent spatial window. In contrast, the Weight-reuse scheme convolves the entire IFM with a single weight before moving on to the next weight.

Lines 6-10 of Fig. 4 show the unrolled inner loops, which represent the computations performed by the MAC arrays for every cycle (i.e. convolution between $K \times K \times T_i \times T_o$ weight and $(K + T_r - 1) \times (K + T_r - 1) \times T_i$ IFM). Note that

```
// K×K×C_in×C_out weight stored in on-chip buffer
(1)  Load (K + (T_r − 1)) rows of IFM
(2)  for (r = 0; r < H_out; r += T_r)
(3)    for (c = 0; c < W_out; c += T_c)                    IFM-consuming time
(4)      for (m = 0; m < C_out; m += T_o)                  = ⌈W_out/T_c⌉ ⌈C_out/T_o⌉ ⌈C_in/T_i⌉
(5)        for (n = 0; n < C_in; n += T_i)
(6)          for (t_m = m; t_m < min(m + T_o, C_out); t_m++)   Single-cycle Execution
(7)            for (t_n = n; t_n < min(n + T_i, C_in); t_n++)
(8)              for (t_r = r; t_r < min(r + T_r, H_out); t_r++)
(9)                for (t_c = c; t_c < min(c + T_c, W_out); t_c++)
(10)                 O(t_r, t_c, t_m) += Σ_{i=0}^{K−1} Σ_{j=0}^{K−1} W(i, j, t_n, t_m) * I(t_r + i, t_c + j, t_n)
(11) Discard upper T_r rows of IFM, Load next T_r rows of IFM
(12) Jump to next layer if possible
```

(a)

```
// H_in×W_in×C_in IFM stored in on-chip buffer
(1)  for (m = 0; m < C_out; m += T_o)
(2)    Load T_o weights, Discard previous T_o weights
(3)    for (r = 0; r < H_out; r += T_r)                    Weight-consuming time
(4)      for (c = 0; c < W_out; c += T_c)                  = ⌈H_out/T_r⌉ ⌈W_out/T_c⌉ ⌈C_in/T_i⌉
(5)        for (n = 0; n < C_in; n += T_i)
(6)          for (t_m = m; t_m < min(m + T_o, C_out); t_m++)   Single-cycle Execution
(7)            for (t_n = n; t_n < min(n + T_i, C_in); t_n++)
(8)              for (t_r = r; t_r < min(r + T_r, H_out); t_r++)
(9)                for (t_c = c; t_c < min(c + T_c, W_out); t_c++)
(10)                 O(t_r, t_c, t_m) += Σ_{i=0}^{K−1} Σ_{j=0}^{K−1} W(i, j, t_n, t_m) * I(t_r + i, t_c + j, t_n)
```

(b)

Fig. 4. Loop Ordering for (a) IFM-reuse Scheme with Layer Fusion Combined. (b) Weight-reuse Scheme.

the MAC arrays feature a configurable adder tree that allows different parameters to be used for K and T_o in different layers, as described in Section III-C. The orange boxes in Fig. 4 show the computations that use buffered data. For the IFM-reuse scheme, these computations represent the convolution between $K+(T_r-1)$ rows of IFM and all weights, and for the Weight-reuse scheme, they represent the convolution between T_o weights and the entire IFM. While these computations are performed by the PEs, the next set of data is loaded from external memory (line 11 in Fig. 4(a), line 2 in Fig. 4(b)).

Another difference between the two schemes is the application of layer fusion [31]. For layers that use the IFM-reuse scheme, we apply layer fusion to optimize the computation further. The IFM-reuse scheme generates the OFM in the channel direction first (line 4 in Fig. 4(a)), which means that once T_r rows of the OFM are generated, they are complete in terms of their channel dimensions. These rows can be convolved against the corresponding weights to generate the next layer's feature map directly. Line 12 in Fig. 4(a) shows the control flow for layer fusion, which is explained in more detail in Section III-D1.

1) IFM-reuse + Fused Layers: For Nexys A7-100T, the on-chip memory budget for the IFM buffer is set to 200KB-300KB considering the space to store the weights and a margin to ensure that the design meets the timing constraints. However, the IFM and OFM sizes of layers 0, 2, and 4 exceed 300KB. On the other hand, the size of the weights for layers 0, 2, and 4 are very small compared to the IFM and OFM.

Therefore, the weights for these early layers are stored in the on-chip ROM, and layer fusion [31] is applied to layers 0, 2, and 4 to meet the on-chip buffer requirement and minimize external memory accesses.

Specifically, we select $T_r = 2$, meaning that a 4×320×3 layer 0 IFM needs to be maintained in the on-chip buffer

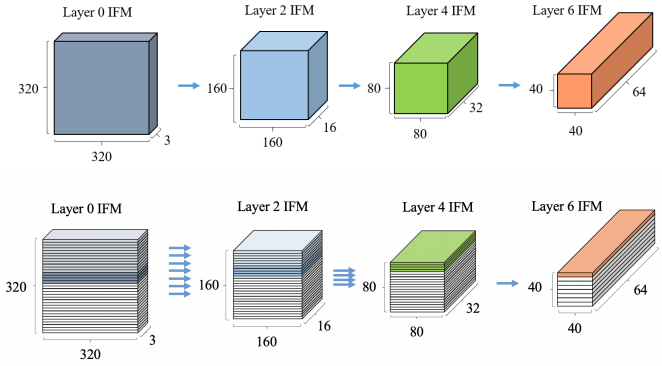

Fig. 5. Illustration of Conventional (Iterative) Forward Pass (top) and Proposed Memory-efficient Fused Layers Forward Pass (bottom) (best viewed in color).

for computation. Performing convolution and max-pooling operations on this tensor yields a 1×160×16 tensor of layer 2 IFM. 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 4 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.

Fig. 5 illustrates the forward pass computation for the IFM-reuse scheme. As opposed to the conventional forward pass, only four rows of each layer's feature map are stored in the on-chip memory at a time. Rows with darker shades represent those presently held in the on-chip buffer, while lighter-shaded rows are discarded after their corresponding computations are finished.

With the above pipelining scheme, the execution time for layer 0 is determined by the following formula:

$$\lceil\frac{H_{out}}{T_r}\rceil \cdot \max(\lceil\frac{W_{out}}{T_c}\rceil\lceil\frac{C_{out}}{T_o}\rceil\lceil\frac{C_{in}}{T_i}\rceil, \frac{T_r \times W_{in} \times C_{in}}{Y}) \quad (1)$$

$\lceil\frac{W_{out}}{T_c}\rceil\lceil\frac{C_{out}}{T_o}\rceil\lceil\frac{C_{in}}{T_i}\rceil$ is the latency of computing $T_r \times W_{out} \times C_{out}$ OFM (IFM-consuming time in Fig. 4(a)), and the $\frac{T_r \times W_{in} \times C_{in}}{Y}$ is the latency of loading another T_r rows of IFM (Y indicates DMA throughput in bytes per cycle).

For example, if the DMA throughput is 4B/cycle, both of these values equal 640 cycles. This means that the DMA load latency is effectively hidden by the computations. Additionally, external memory accesses occur only at layer 0, meaning that layer 2 and layer 4 forward passes incur no external memory accesses. Thus, the execution times for layers 2 and 4 are determined by the computations only.

Unlike [31], which fuses the computation of adjacent layers by mapping fused layers to a dedicated set of MAC units, our architecture utilizes a configurable and unified MAC structure that is only mapped to a single layer at a time stamp. More importantly, our scheduling scheme partially executes a layer in a fused group to output partial OFMs, and invokes its following layer's computation, largely eliminating the off-chip feature map data transfer such as [31]. Specifically, we utilize a line-based dataflow in which several OFM lines of

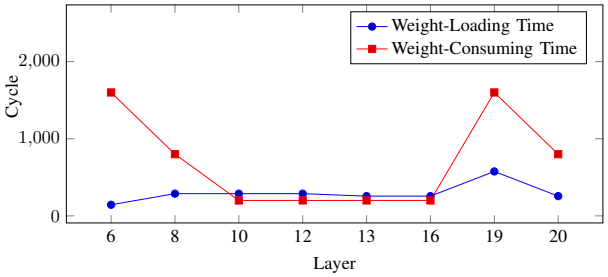

Fig. 6. Weight-Loading Time and Weight-Consuming Time.

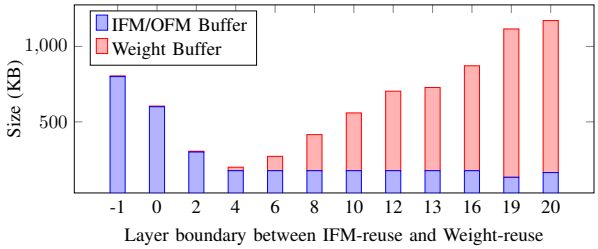

Fig. 7. Buffer Requirement w.r.t. Layer Boundary between IFM-reuse and Weight-reuse (best viewed in color). Layer boundary of -1 (20) indicates that only Weight-reuse (IFM-reuse) scheme is used throughout all layers.

fused layers are cached in on-chip buffers, as shown in Fig. 4 and Fig. 5. Notably, a patch-based dataflow adopted in layer fusion [31] or MCUNetv2 [37] could further reduce the peak memory usage that becomes critical for low-arithmetic intensity layers, including depth-wise or point-wise convolutions in [37]. Unfortunately, it incurs considerable overhead (so-called "recomputation" [31]) that may severely affect performance for high-arithmetic intensity layers, i.e., conv3×3 in YOLO models [4], [5], [15], [16]. The proposed line-based dataflow, however, can avoid such computation overhead at a relatively low cost of on-chip buffer size. In Section III-D3, we provide additional discussion on how to minimize the on-chip buffer requirement, especially for dynamically executing fused IFM-reuse layers and weight-reuse layers.

2) Weight-reuse: From layer 6, the size of the weight increases significantly, whereas the size of the feature maps becomes small enough such that it fits into the on-chip buffer. Thus, the weights should be loaded from DRAM and must be reused extensively in order to hide the DMA latency that arises when loading the weights. Assuming that the feature maps are stored on-chip, the execution time for layers with the weight-reuse scheme applied, excluding the DMA store latencies, can be obtained by the following formula:

$$\lceil\frac{C_{out}}{T_o}\rceil \cdot \max(\lceil\frac{H_{out}}{T_r}\rceil\lceil\frac{W_{out}}{T_c}\rceil\lceil\frac{C_{in}}{T_i}\rceil, \frac{K\times K\times C_{in}\times T_o}{Y}) \quad (2)$$

The weight-consuming time, referring to the time required to reuse a block of weights fully, equals $\lceil\frac{H_{out}}{T_r}\rceil\lceil\frac{W_{out}}{T_c}\rceil\lceil\frac{C_{in}}{T_i}\rceil$ (Fig. 4(b)). The weight-loading time, which is the latency of loading the $K\times K\times C_{in}\times T_o$ weight through DMA, is obtained by dividing the size of the weight by the DMA throughput Y.

Fig. 6 compares the weight-loading time and weight-consuming time at layers 6+, assuming DMA throughput of 4B/cycle. It can be observed from Fig. 6 that the weight-consuming time exceeds the weight-loading time in layers 6, 8, 19, and 20. Thus, no pipeline stall is incurred in these layers. However, pipeline stalls are inevitable in layers 10, 12, 13, and 16. Nevertheless, the effects of these stalls are negligibly small, as the number of computations required for these layers is only 5.38% of the total.

3) Choosing Layer Boundary Between IFM-reuse and Weight-reuse: The effect of the dynamic data reuse scheme appears in the on-chip buffer requirement. Fig. 7 shows the size of IFM/OFM buffer and the weight buffer with respect to the layer boundary between IFM-reuse and Weight-reuse. IFM-reuse (fused-layer) scheme is effective for minimizing the IFM/OFM buffer requirement because it requires only a few rows of the feature map to be stored in the buffer. However, the IFM-reuse scheme requires all weights to be available in on-chip storage, which can lead to larger buffer space for weights compared to that in the Weight-reuse scheme. Conversely, the Weight-reuse scheme requires a block of weights and the entire feature map to be stored on-chip. Thus, it can be observed from Fig. 7 that as the boundary moves to deeper layers, the IFM/OFM buffer size tends to decrease while the weight buffer requirement increases. Appendix B provides a more detailed analysis of the buffer requirements with respect to the layer boundary.

Our choice of layer boundary 4 achieves the minimum buffer requirement of 199.2 KB for IFM/OFM and the weights overall, as shown in Fig. 7. Monolithic loop ordering that corresponds to choosing a layer boundary of -1 (Weight-reuse only) or 20 (IFM-reuse only), requires very large buffers (804.5KB with IFM-reuse only, 1007KB with Weight-reuse only) making this strategy not feasible on resource-constrained devices. This can result in swapping of the data to the external memory, which can deteriorate both the latency and the energy consumption.

E. Buffer Organization

1) IFM/OFM Buffer: In YOLOv3-tiny, 3×3 convolution accounts for most of the total computation, and all six pooling layers perform 2×2 max pooling. Also, our implementation slides the convolution window in the channel direction first (line 5 of Fig. 4), which is advantageous in minimizing partial sum storage. Thus, a suitable buffer structure should be devised accordingly to support this sliding operation without bubble cycles. To this end, we propose a 4×4 spatially banked on-chip IFM/OFM buffer.

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 Fig. 8 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 with no bubble cycles and removes the need for additional logic for partial-sum saving and delivery.

One advantage of using a 4×4 structure over 3×3 is that the number of computations is quadrupled for conv3×3 with stride 1, while the IFM data to be fetched increases only by 16/9

Fig. 8. Illustration of Spatially Banked Feature Map Buffers.

times, by exploiting the convolutional reuse of the IFM data. In other words, for conv3×3, our design could access four IFM tensors 3×3×16 (=576B) in a fused tensor 4×4×16 (=256B), lowering the on-chip bandwidth requirement by 55.56% (=1-256/576). Moreover, the 2×2 OFM generated from the convolution between a 4×4 window and the corresponding weights can be fed directly to the max-pooling module, removing the need for additional buffering for max-pooling.

The buffer structure is capable of supporting various data patterns for the MAC array, which comes equipped with a configurable adder tree. In layer 0 mode, the IFM is replicated four times by storing a single buffer with the copied IFM in the channel direction. During conv1×1, an IFM window of size 2×2×16 is sent to the MAC array by sweeping four adjacent buffers (i.e. buffers $i, i+1, i+4, i+5$, where i can be 0, 2, 8, or 10). Additionally, neither mode experiences bubble cycles.

The spatially banked buffer structure offers an additional advantage by enabling a free 2×2 spatial upsample. Whenever a 1×1 OFM pixel is generated, the corresponding upsampled 2×2 tensor can be stored in the feature map buffer at no extra cost. This can be achieved by broadcasting the 1×1 tensor to the data bus of the 2x2 adjacent buffers.

Each buffer is implemented using dual-port on-chip BRAM slices. The 16 buffers collectively form an IFM buffer array, consisting of 16 read and write ports. In our design, two buffer arrays are instantiated to double these ports. This doubling of write ports proves beneficial in layers with shortcut connections, like layer 8, as it enables the simultaneous saving of two OFMs. Our proposed buffer structure permits dynamic buffer allocation, enabling efficient handling of layer-wise dataflow, including the need to save both the OFM and its pooled results during the computation of layer 8.

2) Weight Buffer: To support the three different modes of PEs, it is necessary for the weight buffer to provide weights that are aligned with the corresponding IFM window, as shown in Fig. 2(b). Therefore, each word of the weight buffer stores a block of aligned weights that can be directly provided to the PEs. In our implementation, the global weight buffer BRAM width is set to 144 bytes, enabling it to update the weights provided to the PEs every cycle without causing any stalls. In addition, the weight buffer also includes a ROM that is initialized with the weights of the early layers. This feature eliminates the need for external memory accesses to retrieve these weights.

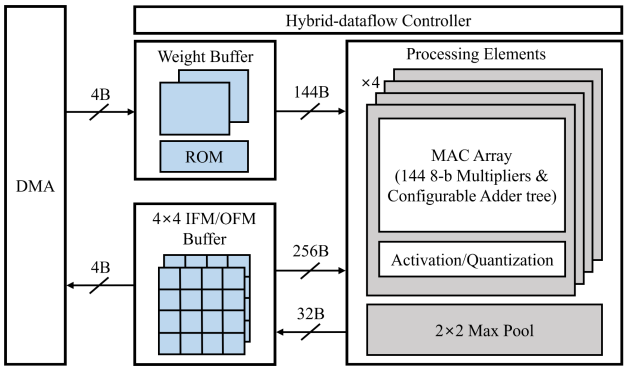

Fig. 9. Illustration of Accelerator Core.

F. Overall Hardware Architecture

Fig. 9 illustrates the microarchitecture of the accelerator engine, which consists of a controller, weight buffer, IFM and OFM buffer, and PEs. AXI read requests are issued to fetch the input image and weights from the external memory, while AXI write requests are issued to send the outputs of layers 13 and 20 to the external memory. The hybrid-dataflow controller is an FSM that schedules the operations of all modules, enabling inter-module pipelining, and decides which reuse scheme to apply for each layer.

As detailed in Section III-E2, the weight buffer transfers 144 bytes of weights to the PEs. In contrast, the IFM/OFM buffer sends out 256 bytes of the IFM per cycle, corresponding to the 4×4×16 IFM window.

The PEs consist of four MAC arrays, activation/quantization modules, and a max-pooling module. They generate 32 bytes of OFM pixels which are stored into the IFM/OFM buffer. The unquantized OFM pixels produced by the MAC arrays are provided to the activation/quantization module, which are implemented using shifters as described in Section III-A2, and the quantized 2×2 OFM pixels are fed to the max-pooling module that performs pooling by comparing their values.

Our accelerator's overall operation can be summarized as following: as the spatially-banked 4×4 IFM buffer and the weight buffer provide the data operands without any bubble cycles to the PEs, the MAC units with configurable adder tree output the variable-shaped OFM, applying proper computing mode and reuse scheme through the controller.

IV. Evaluation

A. FPGA Implementation

1) Overall Architecture: To validate the proposed methodologies, we implement a CNN accelerator on a Digilent Nexys A7-100T board. Like Zedboard [9], and Ultra96 V2 [10], the FPGA board targets IoT applications and therefore has limited hardware resources, i.e., 63400 LUTs, 126800 FFs, 135 BRAM36K, and 240 DSPs. Our design is implemented with Verilog HDL, synthesized, and implemented using Xilinx Vivado. The overall system is described in Fig. 10. The accelerator engine is packaged with AXI interfaces and integrated into the system with a Microblaze processor, a UART, a DRAM controller MIG, and an AXI bus interconnect.

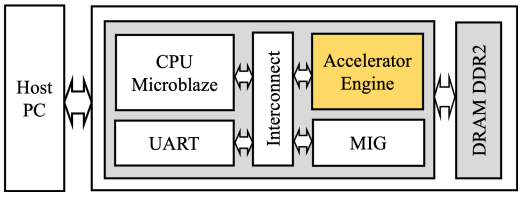

Fig. 10. System Overview of the Experimental Setup.

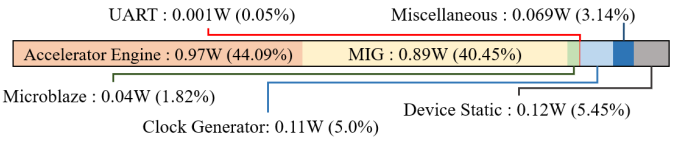

Fig. 11. Power Breakdown (best viewed in color).

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.

The design is verified by testing several images and comparing the outputs from the accelerator with the software-generated results. Also, we profile the execution time for the inference of a single image using hardware counters. The result of the profiling is shown in Fig. 12 and 13.

2) Resource Utilization: Table II 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.

TABLE II
RESOURCE UTILIZATION

Module	LUT	FF	BRAM36K	DSP
Available on Board	63400	126800	135	240
Accelerator Engine	39693 (62.6%)	41892 (33.0%)	84.5 (62.6%)	240 (100%)
MIG	5872 (9.3%)	8871 (26.0%)	0	0
Microblaze	4027 (6.4%)	6816 (5.4%)	8 (5.9%)	0
UART	151 (2.4%)	379 (0.3%)	0	0
Miscellaneous	85 (1.3%)	140 (0.1%)	0	0
Total Utilized	49828 (78.6%)	58098 (45.8%)	92.5 (68.5%)	240 (100%)

3) Energy Breakdown: At a clock frequency of 100MHz, the designed accelerator shows power consumption of 2.203 W. Fig. 11 shows the power breakdown of each module in Fig. 10. The accelerator engine, which is the core module of our system, accounts for 44.1% of the power. This is a highly energy-efficient design compared to other GPU-based implementations.

B. Effect of Flexible Layer-wise Mapping and Dataflow

Fig. 12 presents the timing chart for one inference which consists of the execution times of convolutional computation (COMPUTE), IFM or weight loading (DMA(L)), and OFM storing (DMA(S)) for the layers in YOLOv3-tiny.

1) Unified Processing Elements with a Configurable Adder Tree: As shown in Fig. 12, the unified PE array is assigned to a specific layer at a time. For example in (1), the executions of layers 0, 2, and 4 are interleaved sequentially instead of being pipelined, clearly different from a streaming-like architecture [27], [28] and a pipeline PE architecture [25]. Furthermore, Fig. 13 illustrates how PEs are configured and utilized for conv3×3, conv1×1, or the first convolution layer. The ratio between the GOP and the execution time for computations are mostly consistent across all layers, which suggests high consistent PE utilization.

2) Flexible Layer-wise Mapping and Dataflow: Fig. 12 clearly illustrates our proposed fusion strategy. For example, in (1), DMA(L) and DMA(S) between layers 0 and 2 are omitted. In other words, when partially executing layer 0 (L00 COMPUTE), partial OFMs of layer 0 are generated and stored in global buffers. Next, the partial OFMs are used to execute layer 2 (L02 COMPUTE) without external memory accesses, such as L00 DMA(S) and L02 DMA(L). Notably, due to max-pooling layers, all OFMs of layer 4 are relatively small and can be stored in global buffers instead of external DRAM. Consequently, our proposed fusion method achieves an external reduction similar to those in earlier work [25], [31] without pipeline PEs. In addition, Fig. 12 also demonstrates highly-imbalanced execution times in a consecutive layer group, such as a layer group (0, 2, 4) or a group (13, 16, 19). For example, if layers 13, 16, and 19 are pipelined, the PEs for layers 13 and 16 mostly enter the IDLE state when synchronized with PEs for layer 19. In other words, the proposed flexible layer-wise mapping and dataflow strategy can effectively avoid PE idle rates for highly imbalanced workloads compared to a streaming-like architecture [27], [35] or a pipeline PE architecture [25], [31].

3) Spatial Buffer Structure with Dynamic Access Patterns: Fig. 12 also demonstrates the impact of the proposed spatial buffer structure by illustrating the density of COMPUTE. For example, in (1), the visually dense executions of layers 0, 2, and 4 indicate that there are no stalls for COMPUTE due to several bubble cycles for IFM preloading or OFM saving.

Fig. 13 presents the execution time breakdown for all layers. Owing to the proposed mapping and dataflow process, external memory accesses are nearly eliminated for intermediate IFMs and OFMs, i.e., DMA(L) and DMA(S) for layers 2 and 4. Notably, external memory accesses may not be fully hidden by computations. A physical board test reveals that the actual average DMA throughput is 2.4B/cycle, while the peak bandwidth is 4B/cycle. This leads to stalls during the processing of layer 0, as the DMA latency incurred when loading layer 0 IFM exceeds the IFM-consuming time. Also, layers 10, 12, 13, and 16 have smaller weight-consuming times compared to weight-loading times (Fig. 6), which results in pipeline stalls due to the DMA load latencies.

The physical board test also reveals the external memory bandwidth scalability of our design. Our design requires 1.92 Gigabits per second (Gbps) at a bandwidth efficiency of 60% (=2.4B/4B). For large FPGA boards in Table III, such as VC707 in [13] and KCU1500 in [14], they support DDR3 and DDR4 accordingly, resulting in the theoretical bandwidths of 12.8 GB/sec for DDR3-1600, and 19.2 GB/s for DDR4-2400, respectively. In other words, our assumption in bandwidth

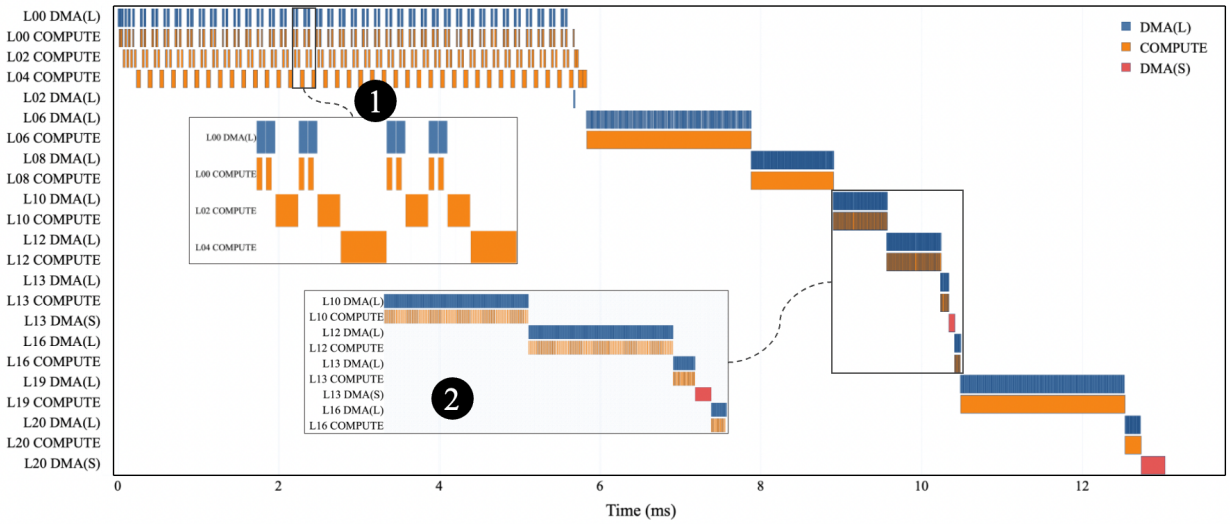

Fig. 12. Gantt Chart of Single Image Inference (best viewed in color).

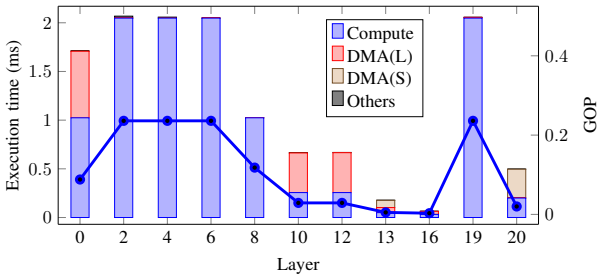

Fig. 13. Latency Breakdown (best viewed in color).

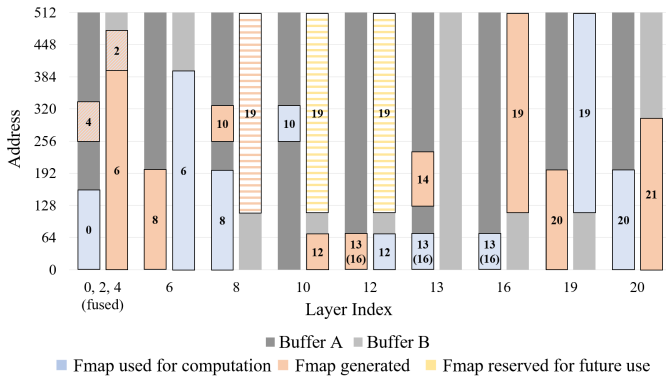

Fig. 14. IFM/OFM Buffer Allocation (best viewed in color).

could be scalable well for other FPGA boards.

C. Dynamic Allocation of Spatially-Banked IFM/OFM Buffer

Similar to typical fixed-data flow architectures [9]–[12], [14], [32], the proposed design utilizes dual global buffers working in a ping-pong manner. However, we propose a dynamic buffer management strategy for flexible layer-wise mapping and dataflow, as illustrated in Fig. 14. The x-axis includes the index (indices) of one layer or fused layers that actively access(es) the IFM/OFM buffer at a time. For a layer or a group of fused layers, two bars represent two buffers marked with their allocated regions along the y-axis address. For example, when executing fused layers (0, 2, 4), Buffer A is allocated for layer 0 (lines 0-159) and layer 4 (lines 256-335), and Buffer B is allocated for layer 2 (lines 400-479) and layer 6 (lines 0-399).

When layer 0 is executed on the unified PE array, it loads the IFMs from external memory into Buffer A and stores the OFMs in Buffer B, i.e., at the region of layer 2. Note that the allocated space for layer 0 is replicated four times for the MAC array of the Layer 0 mode and requires space to store six rows for preloading, unlike layers 2 and 4. Next, when layer 2 is run on the PE array, it loads IFMs from Buffer B and saves OFMs to Buffer A, i.e., at the region of layer 4, and when layer 4 is run on the PE array, it loads IFMs from Buffer A and saves OFMs to Buffer B, i.e., at the segment for layer 6. Notably, during layer fusion, only some of the IFMs and OFMs of the layers are stored in the global buffers, reducing the on-chip buffer requirements for layers 0, 2, 4, and 6 and eliminating external memory accesses for the OFMs of layer 0 and the IFMs and OFMs of layers 2, 4, and 6.

Meanwhile, when executing the other layers, i.e., layers 6, 8, 10, 12, 13, 16, and 19, their IFMs/OFMs in general shrink after a pooling layer or are small enough to be stored entirely in Buffers A and B. Furthermore, the proposed buffer structure and the dynamic buffer management strategy can also support a shortcut layer effectively. For example, the IFMs of layer 19 are formed by concatenating the OFMs of layer 8 and the OFMs of layer 16. When executing layer 8, its OFMs are saved in Buffer A (marked with layer 10) and Buffer B (layer 19). Notably, only a part of segment “19” in Buffer B is allocated to the OFMs of layer 8, resulting in the striped pattern in “19.” After executing layer 16, their OFMs are saved in the remaining part of “19”. Consequently, with the proposed buffer structure, layer 19 can be executed without additional bubble cycles to concatenate the OFMs of layers 16 and 8.

Table IV compares the computation overhead and SRAM usage between line-based dataflow and patch-based dataflow in the context of the fused layer technique [31] and MCUNetv2

TABLE III
PERFORMANCE COMPARISON TO OTHER FPGA ACCELERATORS
(*) INDICATES THAT THE CORRESPONDING VALUE IS NOT REPORTED BY THE PAPERS BUT CALCULATED BY US USING OTHER REPORTED STATISTICS.

	[26]	[9]	[10]	[11]	[12]	[13]	[14]	[34]	This work
Year	2019	2020	2021	2021	2020	2020	2022	2022	2023
Model	YOLOv2-tiny	YOLOv3-tiny	YOLOv3-tiny	YOLOv3-tiny	YOLOv3-tiny	YOLOv3	YOLOv3	MobileNetv2	YOLOv3-tiny
Input Width	416	416	448	416	416	416	416	224	320
Quantization	8b	16b	8b	16b	18b	8b	8b	8b	8b
Dataset	PASCAL VOC	COCOval5k	Custom VOC2007	N/A	N/A	N/A	N/A	ImageNet	Custom
Accuracy drop	0.8%p mAP	2.5%p mAP	2.0%p mAP	N/A	N/A	N/A	N/A	0.05%p acc	2.01%p mAP
Clock	200MHz	100MHz	250MHz	100MHz	200MHz	200MHz	200MHz	333MHz	100MHz
FPGA	ADM-PCIE-7V3	ZYNQ-7020	Ultra96 V2	ZYNQ-7035	Virtex-7 VC707	Virtex-7 VC707	Xilinx KCU1500	ZCU102	Nexys A7-100T
Board Cost	$300	$589	$249	$1499	$5244	$5244	$3670	$3234	$265
BRAM	1326(*)	185	248	248	141	1945	3020	684(*)	185
DSP	3456(*)	160	242	485	2304	2640	2240	1283	240
LUT	637.6k(*)	25.9k	27.3k	N/A	48.6k	230.5k	213.3k	164.4k(*)	50.2k
FF	717.7k(*)	46.7k	38.5k	N/A	93.2k	223.0k	352.0k	N/A	58.1k
Latency	3.18ms	532ms	121ms	192ms	12.08ms(*)	85.76ms(*)	57.57ms	N/A	13.03ms
GOPS	N/A	10.45	31.50	28.99	460.8	767.3	1142	1225.07(*)	95.08
Power	21W	3.36W	4.26W	3.71W	4.81W	N/A	N/A	N/A	2.203W
GOPS/DSP	N/A	0.0653	0.130	0.0598	0.200	0.290	0.510	0.955(*)	0.396
GOPS/W	N/A	3.11	7.40	7.81	95.80	N/A	N/A	N/A	43.16
MAC Utilization	N/A	32.65%(*)	50.00%(*)	29.89%(*)	50.00%(*)	72.70%	69.70%	82.63%	82.53%

TABLE IV
COMPUTATION OVERHEAD AND SRAM USAGE COMPARISON BETWEEN LINE-BASED DATAFLOW AND PATCH-BASED DATAFLOW

Layer 0 Patch Size	Comp. Overhead		IFM Buffer Requirement	
	Layer 0, 2, 4	Overall	Layer 0, 2, 4	Overall
22	3.25x	1.89x	3.95KB (6.3x↓)	100KB
30	1.91x	1.36x	7.70KB (3.25x↓)	100KB
38	1.56x	1.22x	12.70KB (1.97x↓)	100KB

[37]. Assuming no DRAM accesses for intermediate feature maps and no on-chip buffering, patch-based dataflow applied to the IFM-reuse layers (layers 0, 2, and 4) results in a significant reduction in peak IFM buffer usage for these layers, with a maximum decrease of 3.25x compared to the line-based dataflow. However, this benefit comes at the cost of additional computation overhead, leading to a potential 1.89x overall computation increase. Layers 0, 2, and 4 are compute-intensive, as shown in Fig. 13, meaning this added computation directly impacts latency. While patch-based dataflow alleviates buffer requirements for IFM in layers 0, 2, and 4, the IFM buffer's minimum size is set by deeper layers under the weight-reuse scheme (i.e. layers 6 and 19 that have 100KB IFM). Thus, the advantages of patch-based dataflow are less pronounced in scenarios where deeper layers dictate the IFM buffer size.

D. Comparison With Other Works

Our accelerator achieves 76.75 FPS, which is equivalent to a latency of 13.03ms 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*100MHz=115.2 GOPs). The performance comparison between our work and other studies [9], [11]–[14], [26], [34] is summarized in Table III.

Our hardware-friendly 8-bit quantization yields an mAP drop of 2.01%p, which is comparable to [9]'s 16-bit quantization (2.5%p) and [10]'s 8-bit quantization (2.0%p). Notably, our method achieves a similar mAP drop to [10] despite the constraint that our quantization scale factors must be power-of-two numbers. In contrast, [10] uses arbitrary scale factors that need not be powers-of-two. While [26] achieves minimal mAP drop for 8-bit quantization (0.8%p), it applies heterogeneous quantization in which equal-distance-based quantization is applied to some convolutional layers and the power-of-two quantization for other layers. The hardware for supporting this quantization approach may introduce higher hardware overhead compared to our method.

In addition, our accelerator achieves a GOPS of 95.08, which outperforms [9]–[11], but falls short of [13], [14], [26], [34]. While [9]–[11] employ similar numbers of DSPs to our design, their throughputs fall short because [9], [11] require frequent IFM load and OFM store from/to external memory, leading to frequent pipeline stalls, and [10]'s GEMM method incurs both latency overhead and memory overhead due to the im2col operation. Conversely, [13], [14], [26], [34] achieve low latency and/or high throughputs by deploying their pipelined streaming-like architectures on high-cost FPGA boards with almost 10-15x more DSPs and using a 2-3x faster clock than our design.

Our work achieves 0.396 GOPS/DSP, which is higher than other works [9]–[13], but lower than [14], [26], [34]. Notably, although [14] achieves a MAC utilization of 69.70%, which is lower than our 82.53%, its GOPS/DSP is larger than ours thanks to its 2x faster clock. Also, our design utilizes 5.75Kb of on-chip BRAM per MAC. This suggests that the design can effectively utilize computing resources (DSPs) and on-chip memory (BRAMs) thanks to our PE structure, dynamic data reuse, spatial buffer structure, and dynamic allocation. Thus, if the design is scaled by 8x, for example, it will likely fit well with the available DSPs and BRAMs of a large FPGA board such as VC707 with 2800 DSPs and 37Mb BRAMs.

The proposed accelerator achieves the highest MAC utilization of 82.53%, thanks to the configurable adder tree that minimizes idle MAC units for different layer configurations, dynamic data reuse that eliminates the unnecessary external memory transactions and hides the DMA latency with com-

putations, and the spatial buffer structure that incurs no stalls in providing the data to the PEs. Note that the MAC utilization rates of [9]–[12], [34] are not reported in the papers but calculated by us. To ensure fairness, the fact that the number of MAC units can differ from the number of DSPs used is taken into account when calculating these values. For instance, [9] maps each DSP to one 8-bit multiplier whereas our accelerator and [34] map each DSP unit to two 8-bit multipliers.

It is worth noting that some of these works [9], [11], [12], [26] use high-level synthesis to implement their designs. While HLS can reduce the design efforts significantly compared to the register-transfer level (RTL) design, it is generally known to generate less optimal hardware than hardware produced using RTL.

E. Adaptation of Proposed Design Methodology for YOLOv4-tiny

While mainly targeting a particular model, i.e., YOLOv3-tiny, our design methodology may be quickly adapted to YOLO-like models. In this subsection, we demonstrate our methodology to another YOLO variant, YOLOv4-tiny [38]. Like YOLO-like models [4], [15], [16], it also consists of 3×3 and 1×1 convolution operations, which account for 92.2% and 7.8% of the total computations, respectively, as shown in Table V. Compared with YOLOv3-tiny, YOLOv4-tiny stacks more convolution layers and adopts more shortcut-connection layers to concatenate inter-layer feature maps, which results in a slightly better mAP (i.e., 84.80% vs. 83.20%) with a smaller computing requirement (i.e., 0.678 vs. 1.236 GFLOPs). It, however, incurs many challenges in mapping the model to a resource-constrained device, for example, allocating unified buffers for many fused layers and shortcut layers and quantizing shortcut layers.

1) Unified Buffer Organization and Allocation: Like fixed dataflow designs [9]–[12], [14], [32], the proposed architecture utilizes dual global buffers working in a ping-pong manner. Our proposed solution, however, includes (i) a *unique buffer organization strategy* for IFMs/OFM to support efficient convolution operations and (ii) a *flexible layer-wise mapping and dataflow strategy*, allowing for the execution of specific layers at a time, leading to improved efficiency. Specifically, Fig. 15 demonstrates our dynamic buffer management strategy for flexible layer-wise mapping and dataflow with YOLOv4-tiny. The x-axis shows the index (indices) of one layer or fused layers that actively access(es) the IFM/OFM buffer at a time. For example, when executing fused layers (0, 2, 4, 6, 7, 9), Buffer A is allocated for layers 2, 7 and 12, and Buffer B is allocated for layers 0, 4, 6, and 9. Notably, the number of fused layers is determined by minimizing the on-chip buffer usage, as explained in Section III-D3 (Choosing Layer Boundary Between IFM-reuse and Weight-reuse). Notably, during layer fusion, only some of the IFMs and OFMs of the layers are stored in the global buffers, reducing the on-chip buffer requirements for layers 0, 2, 4, 6, 7, and 9 and eliminating external memory accesses for the OFMs of layer 0 and the IFMs and OFMs of layers 2, 4, 6, 7, and 9. Meanwhile, when executing the other layers, i.e., layers 12, 14, 15, 17, 20, 22, 23, 25, 28, 29, 32, and 35, their IFMs/OFMs generally shrink after a pooling layer or are small enough to be stored entirely in Buffers A and B. Furthermore, the proposed buffer structure and the dynamic buffer management strategy can also support shortcut layers, i.e., layers 5, 10, 12, 18, 21, 26, 31, and 34 in YOLOv4-tiny, effectively. In total, our design avoids loading/storing 2.05 MB for IFMs/OFMs, i.e., 89.25% of the total IFMs/OFMs, per inference.

TABLE V
CUSTOM YOLOV4-TINY NETWORK ARCHITECTURE

Layer	Type	Filter	Input	Output	GOP
0	conv	3*3*3*16	256*256*3	256*256*16	0.057
1	max		256*256*16	128*128*16	
2	conv	3*3*16*32	128*128*16	128*128*32	0.151
3	max		128*128*32	64*64*32	
4	conv	3*3*32*32	64*64*32	64*64*32	0.075
5	route	4			
6	conv	3*3*16*16	64*64*16	64*64*16	0.019
7	conv	3*3*16*16	64*64*16	64*64*16	0.019
8	route	7 6			
9	conv	1*1*32*32	64*64*32	64*64*32	0.008
10	route	4 9			
11	max		64*64*64	32*32*64	
12	conv	3*3*64*64	32*32*64	32*32*64	0.075
13	route	12			
14	conv	3*3*32*32	32*32*32	32*32*32	0.019
15	conv	3*3*32*32	32*32*32	32*32*32	0.019
16	route	15 14			
17	conv	1*1*64*64	32*32*64	32*32*64	0.008
18	route	12 17			
19	max		32*32*128	16*16*128	
20	conv	3*3*128*128	16*16*128	16*16*128	0.075
21	route	20			
22	conv	3*3*64*64	16*16*64	16*16*64	0.019
23	conv	3*3*64*64	16*16*64	16*16*64	0.019
24	route	23 22			
25	conv	1*1*128*128	16*16*128	16*16*128	0.008
26	route	20 25			
27	max		16*16*256	8*8*256	
28	conv	3*3*256*256	8*8*256	8*8*256	0.075
29	conv	1*1*256*195	8*8*256	8*8*195	0.006
30	yolo				
31	route	27			
32	conv	1*1*256*64	8*8*256	8*8*64	0.002
33	upsample		8*8*64	16*16*64	
34	route	33 25			
35	conv	1*1*192*195	16*16*192	16*16*195	0.019
36	yolo				

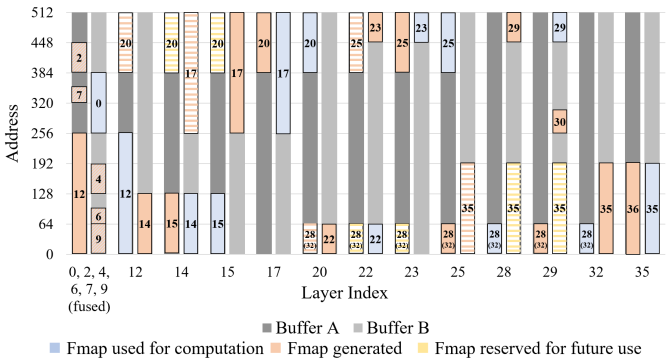

Fig. 15. IFM/OFM Buffer Allocation for YOLOv4-tiny (best viewed in color).

2) Memory Access Reduction and Utilization: Fig. 16 reports the layer-wise utilization with YOLOv4-tiny, whereas the x-axis shows the index (indices) of one (fused) layer(s). For example, our design achieves the MAC utilization of 75.72% for fused layers, thanks to the flexible layer-wise mapping and dataflow strategy. It is worth noting that our computing design and buffer organization strategy allow us to achieve an average MAC utilization of 97.07% for conv3x3 layers (i.e., 12, 14, 15, 20, 22, and 23) and 86.65% for the conv1x1 layers 17 and 25. Meanwhile, the hardware utilization of layers 28, 29, 32,

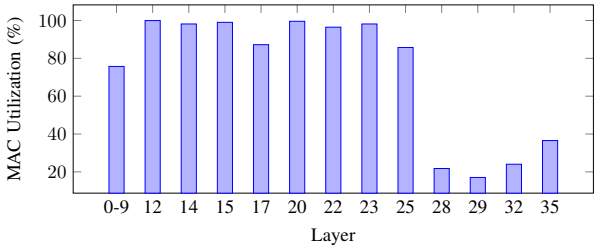

Fig. 16. Layer-wise MAC Utilization of YOLOv4-tiny Layers.

and 35 are relatively low, i.e., 30.09%, on average, due to the limited external memory bandwidth. For example, the peak computing and data rates in our particular setting are 1152 operations and four bytes per cycle, respectively, resulting in a computing-to-data ratio of 288 operations per byte. Layers 28, 29, 32, and 35, however, have computing-to-data ratios of 128.00, 102.04, 128.00, and 219.43 operations per byte, even with the unnecessary IFM/OFM transfers eliminated. To this end, our FPGA implementation with a YOLOv4-tiny model yields 64.63% MAC utilization and achieves a real-time speed of 110.14 frames per second at 100MHz.

3) Performance/Quantization Comparison: Unlike its precedent variants, YOLOv4-tiny includes many shortcut layers (5, 10, 12, 18, 21, 26, 31, and 34), making it more challenging for quantization. The quantized YOLOv4-tiny, however, achieves a reasonable accuracy, i.e., mAP = 78.92%, which is slightly lower than that of YOLOv3-tiny (i.e., 81.19%). Notably, shortcut layers allow concatenating feature maps between shallow and deep layers, resulting in a smaller computing requirement (i.e., 0.678 vs. 1.236 GFLOPs) at a similar accuracy. As a result, YOLOv4-tiny achieves a latency of 9.08 ms per inference, enhancing the framerate by 30.31% compared to YOLOv3-tiny. The mentioned results are reported in Table VI in detail.

For YOLOv4-tiny models, [39], [40] also target low-power and resource-constrained boards, i.e., ZYNQ-7020. Compared with our approach, [39], [40] may slightly reduce the accuracy drop, i.e., -3% and -1.24%, respectively, by adopting 16-bit quantization, while only achieving a relatively low inference speed, for example, 18.03 seconds/inference in [39] and 376 milliseconds/inference in [40]. Meanwhile, our design achieves 9.08ms and 13.03ms per inference for YOLOv4-tiny and YOLOv3-tiny, respectively.

F. Adaptation of Proposed Design for Other FPGAs

To demonstrate the compatibility of our design with other FPGAs, we adapted our design to Zynq 7000 XC7Z020. Zynq 7000 XC7Z020 is used for a broad range of popular low-cost FPGA boards, such as ZedBoard [41] and AMD(Xilinx) ZC702 [42], and boards used in prior works [9], [39], [40]. Table VII presents the resource utilization of our design on Zynq 7000 XC7Z020 for YOLOv3-tiny (left) and YOLOv4-tiny (right). It is worth noting that the LUT and FF counts of the design have dropped for both YOLOv3-tiny and YOLOv4-tiny on this FPGA, due to the processing system, Zynq, included inside the FPGA that removes the need for the Microblaze, UART, and MIG submodules.

TABLE VI
PERFORMANCE COMPARISON TO OTHER YOLOV4-TINY FPGA ACCELERATORS
(*) INDICATES THAT THE CORRESPONDING VALUE IS NOT REPORTED BY THE PAPERS BUT CALCULATED BY US USING OTHER REPORTED STATISTICS.

	[39]	[40]	This work	This work
Year	2021	2022	2023	2023
Model	YOLOv4-tiny	YOLOv4-tiny	YOLOv3-tiny	YOLOv4-tiny
Input Width	416	416	320	256
Quantization	16b	16b	8b	8b
Dataset	COCO	Custom	Custom	Custom
Accuracy drop	3%p mAP	1.24%p mAP	2.01%p mAP	5.88% mAP
Clock	100MHz	N/A	100MHz	100MHz
FPGA	ZYNQ-7020	ZYNQ-7020	Nexys A7-100T	Nexys A7-100T
Board Cost	$264	$264	$265	$265
BRAM	132(*)	96	185	94.5
DSP	149(*)	220	240	240
LUT	30.7k(*)	42.0k	50.2k	45.9k
FF	31.0k	47.7k	58.1k	43.6k
Latency	18.03s	376ms	13.03ms	9.08ms
GOPS	N/A	9.24	95.08	74.45
Power	2.38W	2.86W	2.203W	2.218W
GOPS/DSP	N/A	0.042	0.396	0.310
GOPS/W	N/A	3.23	43.16	33.57
MAC Utilization	N/A	N/A	82.53%	64.63%

TABLE VII
RESOURCE UTILIZATION ON ZYNQ 7000 XC7Z020
YOLOV3-TINY | YOLOV4-TINY

Module	LUT	FF	BRAM36K	DSP
Available on Board	53200	106400	140	220
Accelerator Engine	41646 | 38430	44487 | 28646	84.5 | 82.5	216 | 216
AXI Interconnect	1339 | 970	3609 | 1179	0 | 0	0 | 0
Miscellaneous	514 | 441	1099 | 1107	0 | 0	0 | 0
Total Utilized	43499 | 39841	49195 | 30932	84.5 | 82.5	216 | 216

G. Discussion and Future Works

Conceptually, the proposed design methodology is applicable to other recent YOLO models such as YOLOv7 [43] or v8 [44] because they consist of conv3x3, conv1x1, and shortcut or concatenation layers supported by our methodology. Unfortunately, for real-time applications, YOLOv7 and v8 are more suitable for server-grade accelerators, for example, (167 frames per second on) NVIDIA GPU V100 [45], instead of edge devices. V100, however, has a peak computing power of 112 TFLOPS which is about 1000x higher than the peak throughput of our accelerator (i.e. 115.2 GOPs). Similarly, the popular models such as VGG-16 and ResNet-50 may be efficiently executed by adopting our methodology because they mainly consist of conv3x3 and skip connection layers. They require 30.95 and 7.72 GOPs per inference, so they must be accelerated by more high-end and expensive FPGA boards such as VC707 or ZCU102, for real-time applications, as reported in Table III.

We admit that our methodology may be less effective for lightweight models such as MobileNets and EfficientNet that mainly consist of depth-wise (DW) and point-wise (PW) convolution layers. The reason is that DW and PW layers have lower arithmetic intensity and demand higher memory and bandwidth requirements which limit their overall performances. For example, it is observed that the hardware utilization of conv1x1 layers 28, 29, 32, and 35 in YOLOv4-tiny are relatively low, i.e., 30.09%, on average, because they have computing-to-data ratios of 128.00, 102.04, 128.00,

and 219.43 operations per byte, even with the unnecessary IFM/OFM transfers eliminated. It suggests that the absolute throughput is only about 34.66 GOPS, on average, for those layers. However, we argue that the low utilization phenomenon is also observed by executing other lightweight models like YOLO-ReT [17] and Mobilenet-v2 (MBv2) on edge GPUs [46]. For example, Jetson Nano achieves 24.85 frames per second with MBv2 (marked with width=1.0 in [17]), resulting in a throughput of 15.16 GOPS (=24.85 * 0.61 GOPs). Notably, Jetson Nano has a peak performance of 472 GFLOPS [46], which suggests that the computing utilization is about 3.2% for MBv2.

Compared to domain-specific accelerators, MCUs or general-purpose GPUs provide significant benefits for model architectural exploration. For example, [17] adopts various scaling ratios of 0.5, 0.75, 1.0, and 1.4 to find a proper model for a given latency target, i.e., 33 ms in real-time applications. To this end, we will consider scaling up our design and building a compiler to support more diverse workloads as future work.

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.

References

[1] A. Boukerche and Z. Hou, "Object Detection Using Deep Learning Methods in Traffic Scenarios," *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1–35, 2021.

[2] E. Arnold, O. Y. Al-Jarrah, M. Dianati, S. Fallah, D. Oxtoby, and A. Mouzakitis, "A Survey on 3D Object Detection Methods for Autonomous Driving Applications," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 10, pp. 3782–3795, 2019.

[3] S.-J. Horng and P.-S. Huang, "Building Unmanned Store Identification Systems Using YOLOv4 and Siamese Network," *Applied Sciences*, vol. 12, no. 8, p. 3826, 2022.

[4] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[5] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," *arXiv preprint arXiv:2004.10934*, 2020.

[6] M. Tan, R. Pang, and Q. V. Le, "EfficientDet: Scalable and Efficient Object Detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 10 781–10 790.

[7] NVIDIA Jetson Nano. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/

[8] NVIDIA Xavier NX. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/

[9] Z. Yu and C.-S. Bouganis, "A Parameterisable FPGA-Tailored Architecture for YOLOv3-Tiny," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 16th International Symposium, ARC 2020, Toledo, Spain, April 1–3, 2020, Proceedings 16*. Springer, 2020, pp. 330–344.

[10] 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," *IEEE access*, vol. 9, pp. 141 890–141 913, 2021.

[11] Q. Xiong, C. Liao, Z. Yang, and W. Gao, "A Method for Accelerating YOLO by Hybrid Computing Based on ARM and FPGA," in *2021 4th International Conference on Algorithms, Computing and Artificial Intelligence*, 2021, pp. 1–7.

[12] A. Ahmad, M. A. Pasha, and G. J. Raza, "Accelerating Tiny YOLO v3 using FPGA-based Hardware/Software Co-Design," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.

[13] D. T. Nguyen, H. Kim, and H.-J. Lee, "Layer-Specific Optimization for Mixed Data Flow With Mixed Precision in FPGA Design for CNN-Based Object Detectors," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 6, pp. 2450–2464, 2020.

[14] D. T. Nguyen, H. Je, T. N. Nguyen, S. Ryu, K. Lee, and H.-J. Lee, "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*, vol. 69, no. 6, pp. 2477–2489, 2022.

[15] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[16] P. Adarsh, P. Rathi, and M. Kumar, "YOLO v3-tiny: Object Detection and Recognition using one stage improved model," in *2020 6th international conference on advanced computing and communication systems (ICACCS)*. IEEE, 2020, pp. 687–694.

[17] P. Ganesh, Y. Chen, Y. Yang, D. Chen, and M. Winslett, "YOLO-ReT: Towards High Accuracy Real-time Object Detection on Edge GPUs," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2022, pp. 3267–3277.

[18] S. Liang, H. Wu, L. Zhen, Q. Hua, S. Garg, G. Kaddoum, M. M. Hassan, and K. Yu, "Edge YOLO: Real-Time Intelligent Object Detection System Based on Edge-Cloud Cooperation in Autonomous Vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 12, pp. 25 345–25 360, 2022.

[19] Y. Chen, G. Yin, Z. Tan, M. Lee, Z. Yang, Y. Liu, H. Yang, K. Ma, and X. Li, "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] NVIDIA Geforce RTX 2080Ti. [Online]. Available: https://www.nvidia.com/en-me/geforce/graphics-cards/rtx-2080-ti/

[21] H. Feng, G. Mu, S. Zhong, P. Zhang, and T. Yuan, "Benchmark Analysis of YOLO Performance on Edge Intelligence Devices," *Cryptography*, vol. 6, no. 2, p. 16, 2022.

[22] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort, "A White Paper on Neural Network Quantization," *arXiv preprint arXiv:2106.08295*, 2021.

[23] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv preprint arXiv:1510.00149*, 2015.

[24] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[25] X. Chen, J. Xu, and Z. Yu, "A 68-mw 2.2 Tops/w Low Bit Width and Multiplierless DCNN Object Detection Processor for Visually Impaired People," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 29, no. 11, pp. 3444–3453, 2018.

[26] C. Ding, S. Wang, N. Liu, K. Xu, Y. Wang, and Y. Liang, "REQ-YOLO: A Resource-Aware, Efficient Quantization Framework for Object Detection on FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2019, pp. 33–42.

[27] Y. Kim, J.-S. Choi, and M. Kim, "A Real-Time Convolutional Neural Network for Super-Resolution on FPGA With Applications to 4K UHD 60 fps Video Services," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 29, no. 8, pp. 2521–2534, 2018.

[28] D. T. Nguyen, T. N. Nguyen, H. Kim, and H.-J. Lee, "A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1861–1873, 2019.

[29] Digilent Nexys A7 100T. [Online]. Available: https://digilent.com/shop/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/

[30] Q. H. Vo, N. L. Le, F. Asim, L.-W. Kim, and C. S. Hong, "A Deep Learning Accelerator Based on a Streaming Architecture for Binary Neural Networks," *IEEE Access*, vol. 10, pp. 21 141–21 159, 2022.
[31] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
[32] L. Cavigelli and L. Benini, "Origami: A 803-GOp/s/W Convolutional Network Accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, 2016.
[33] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," *ACM SIGARCH computer architecture news*, vol. 44, no. 3, pp. 367–379, 2016.
[34] W. Jiang, H. Yu, and Y. Ha, "A High-Throughput Full-Dataflow MobileNetv2 Accelerator on Edge FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.
[35] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.
[36] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep Learning with INT8 Optimization on Xilinx Devices," 2017. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8
[37] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning," *arXiv preprint arXiv:2110.15352*, 2021.
[38] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Scaled-YOLOv4: Scaling Cross Stage Partial Network," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 13 029–13 038.
[39] P. Li and C. Che, "Mapping YOLOv4-Tiny on FPGA-Based DNN Accelerator by Using Dynamic Fixed-Point Method," in *2021 12th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*. IEEE, 2021, pp. 125–129.
[40] S. Xu, Y. Zhou, Y. Huang, and T. Han, "YOLOv4-Tiny-Based Coal Gangue Image Recognition and FPGA Implementation," *Micromachines*, vol. 13, no. 11, p. 1983, 2022.
[41] Avnet ZedBoard. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/1-8dyf-11.html
[42] AMD Zynq 7000 SoC ZC702 Evaluation Kit. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html
[43] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 7464–7475.
[44] Ultralytics YOLOv8. [Online]. Available: https://github.com/ultralytics/ultralytics/
[45] NVIDIA V100 Tensor Core GPU. [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/
[46] Jetson Modules. [Online]. Available: https://developer.nvidia.com/embedded/jetson-modules

Minsik Kim received a BSc in Electrical and Computer Engineering from Seoul National University, Seoul, South Korea, in 2023. He is currently pursuing a PhD in Electrical and Computer Engineering at University of Michigan, Ann Arbor, United States. His research interests include accelerator architectures for machine learning workloads and reconfigurable computing.

Kyungseok Oh received a BSc in Electrical and Computer Engineering from Seoul National University, Seoul, Korea, in 2023. His research interests primarily focus on algorithm design and NPU (Neural Processing Unit) architecture.

Youngmock Cho received a BSc in Electrical and Computer Engineering from Seoul National University, Seoul, Korea, in 2022. He is currently pursuing an MSc in the Electrical and Computer Engineering Department at Seoul National University. His research interests primarily focus on the field of Processing-In-Memory (PIM).

Hojin Seo received a BSc in Electrical and Computer Engineering from Seoul National University, Seoul, Korea, in 2023.

Xuan Truong Nguyen received his B.S. in Electrical Engineering from Hanoi University of Science and Technology, Hanoi, Vietnam, in 2011; M.S., and Ph.D. degrees in Electrical Engineering and Computer Science from Seoul National University, Seoul, Korea, in 2015 and 2019, respectively. He worked as a postdoctoral fellow from BK21+ of the Department of Electrical and Computer Engineering (ECE) and the Inter-University Semiconductor Research Center (ISRC) of Seoul National University from April 2019 to August 2021. Since September 2021, he has been a research assistant professor at the Department of Next-generation Semiconductor Convergence and Open Sharing Systems (COSS). His research interests include developing and optimizing hardware-software co-design algorithms and reconfigurable, power-efficient, high-performance computing systems for computer vision and multimedia applications.

Hyuk-Jae Lee received BSc and MSc in electronics engineering from Seoul National University, Seoul, South Korea, in 1987 and 1989, respectively, and a PhD in electrical and computer Engineering from Purdue University, West Lafayette, IN, USA, in 1996. From 1998 to 2001, he was a Senior Component Design Engineer at the Server and Workstation Chipset Division, Intel Corporation, Hillsboro, OR, USA. From 1996 to 1998, he was a Faculty Member at the Department of Computer Science, Louisiana Tech University, Ruston, LS, USA. In 2001, he joined the School of Electrical Engineering and Computer Science, Seoul National University, where he is a Professor. He is the Founder of Mamurian Design, Inc., Seoul, a fabless SoC design house for multimedia applications. His current research interests include computer architectures and SoCs for multimedia applications.

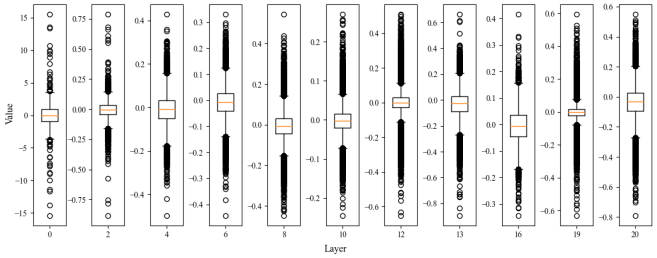

Fig. 17. Layer-wise Weight Distribution.

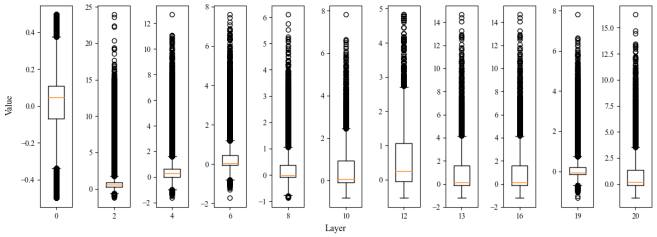

Fig. 18. Layer-wise Activation Distribution.

APPENDIX A
RANGE-BASED QUANTIZATION WITH POWER-OF-TWO SCALE FACTORS

Our quantization method first involves the extraction of the distribution of both weights and activation. While weights can be extracted from the given trained neural network, the values of activations are dependent upon different input images. Thus, we extract the distribution of the activations by performing network forward pass to a small calibration dataset. These distributions are shown in Fig. 17 and 18.

Using this distribution, we apply a simple uniform symmetric quantization with power-of-two scale factors. This can be represented by the following equation:

$$x_{int} = \lfloor \frac{\text{clamp}(x)}{s} \rceil, \ s \in 2^n \text{ for } n \in \mathbb{Z} \tag{3}$$

The approximated value represented by this integer quantization then becomes:

$$\hat{x} = s(x_{int}) \tag{4}$$

In order to find suitable scale factors for each layer, we perform a linear search for the power-of-two scale factor s that minimizes the mean squared error (MSE) between the original floating-point value and the approximated value. MSE-based range setting are known to be less sensitive to the effect of outliers in the distribution, which can be useful for reducing the rounding errors [22].

APPENDIX B
BUFFER REQUIREMENT WITH RESPECT TO LAYER BOUNDARY BETWEEN IFM-REUSE AND WEIGHT-REUSE

Formally, the required IFM/OFM buffer size with layer x as the boundary between IFM-reuse and Weight-reuse can be calculated with the following formula:

$$\begin{aligned}\text{IFM/OFM\_buf} = \max(&\text{IFM\_OFM\_buf}_{0:x}^{\text{IFM\_Reuse}}, \\ &\text{IFM\_OFM\_buf}_{x+1:\text{end}}^{\text{Weight\_Reuse}})\end{aligned}$$

$\text{IFM\_OFM\_buf}_{0:x}^{\text{IFM\_Reuse}}$ represents the IFM/OFM buffer size required by layers 0 to x when IFM-reuse is applied to these layers. Similarly, $\text{IFM\_OFM\_buf}_{x+1:\text{end}}^{\text{Weight\_Reuse}}$ indicates the IFM/OFM buffer size required to process layers $x+1$ to the end layer with Weight-reuse scheme. These values can be calculated by:

$$\begin{aligned}\text{IFM\_OFM\_buf}_{0:x}^{\text{IFM\_Reuse}} = &\sum_{i=0}^{x}(K^i + S^i \cdot T_r - 1)W_{in}^i C_{in}^i \\ &+ H_{out}^x W_{out}^x C_{out}^x\end{aligned}$$

$$\begin{aligned}\text{IFM\_OFM\_buf}_{x+1:\text{end}}^{\text{Weight\_Reuse}} = &\max_{i\in[x+1,\text{end}]}(H_{in}^i W_{in}^i C_{in}^i \\ &+ H_{out}^i W_{out}^i C_{out}^i)\end{aligned}$$

S^i denotes the stride of the convolutional layer i. The summation for IFM/OFM buffer space for the IFM-reuse scheme is due to layer fusion. With layer fusion, the fused layers' feature maps need to be stored in the on-chip buffer together at the same time, whereas for Weight-reuse scheme in which layer-by-layer processing is done the IFM/OFM buffer size is determined by the maximum among single-layer IFM/OFM sizes.

Using similar notation, the weight buffer size can be computed with the following formulas:

$$\begin{aligned}\text{Weight\_buf} = \max(&\text{Weight\_buf}_{0:x}^{\text{IFM\_Reuse}}, \\ &\text{Weight\_buf}_{x+1:\text{end}}^{\text{Weight\_Reuse}})\end{aligned}$$

$$\text{Weight\_buf}_{0:x}^{\text{IFM\_Reuse}} = \sum_{i=0}^{x} K^i K^i C_{in}^i C_{out}^i$$

$$\text{Weight\_buf}_{x+1:\text{end}}^{\text{Weight\_Reuse}} = \max_{i\in[x+1,\text{end}]}(K^i K^i C_{in}^i T_o^i)$$