Dedicated FPGA Implementation of the Gaussian TinyYOLOv3 Accelerator

Subin Ki, *Student Member, IEEE*, Juntae Park, *Student Member, IEEE*, and Hyun Kim, *Senior Member, IEEE*

***Abstract*—This brief presents a dedicated FPGA implementation of the Gaussian TinyYOLOv3 accelerator using a streamline architecture for object detection in mobile and edge devices. The proposed accelerator employs a hardware-friendly shift-based floating-fixed MAC operator and shift-based quantization method that significantly reduces hardware resources and minimizes accuracy degradation. The pipelined streamline architecture maximizes hardware utilization and stores all parameters in on-chip memory to minimize external memory access. Moreover, the Gaussian modeling-based performance enhancement technique is effectively processed in the programmable system to address the low accuracy issue in lightweight models. The proposed IP implemented on Xilinx XCVU9P achieves a processing speed of 62.9 FPS and an accuracy of 34.01% on the COCO2014 dataset, which demonstrates the superiority of the proposed accelerator over prior research in terms of the trade-off between throughput, hardware resources, and model accuracy.**

***Index Terms*—Convolutional neural network (CNN), field-programmable gate array (FPGA), hardware accelerator, object detection, streamline architecture, TinyYOLOv3.**

I. Introduction

With the recent rapid development of artificial intelligence (AI) models, AI is being increasingly used in various fields, such as computer vision and natural language processing. Among them, object detection based on convolutional neural networks (CNNs) is being commercialized not only for autonomous driving, security, medical equipment, and military equipment [1], but also in various mobile fields closely related to users [2]. In particular, the YOLO model, a representative object detector, is widely used in practical applications owing to its superior trade-off between accuracy and computational complexity compared to other models [3]. However, even the YOLO model has limitations in real-time operations on ubiquitous mobile/edge devices with limited hardware resources and battery capacity because it requires numerous parameters and computations [4]. To address this issue, attempts have been made to use various platforms with embedded GPUs for object detection in real-time [2], [5]. However, GPUs exhibit drawbacks such as significant power consumption and a lack of cost-effectiveness, making the optimization of specific model implementations challenging [4], [6].

As a solution, low-power/high-performance CNN accelerators based on field-programmable gate array (FPGA) platforms that offer excellent power efficiency and design flexibility are receiving increasing attention, and numerous studies have been conducted [7], [8], [9], [10], [11], [12]. Yu and Bouganis [7] are the first to implement a parameterizable FPGA-tailored architecture for TinyYOLOv3, optimizing latency-sensitive applications for deployment on low-end devices with strict resource constraints. However, this architecture has the limitation of causing significant degradation in accuracy. Ahmad et al. [8] achieved high throughput and relatively low power consumption by accelerating the TinyYOLOv3 model through a hardware/software co-design approach; however, this method has the limitation of focusing only on accelerating the convolution (CONV) operation rather than the entire CNN. Adiono et al. [9] implemented all layers in TinyVOLOv3 on the hardware and specifically designed the dataflow and control flow in a hybrid architecture to asynchronously perform data processing and computation processes. However, it has the limitation of relatively low throughput, making it difficult to support real-time operations. Pestana et al. [10] implemented a prototype FPGA including a configurable and scalable core for TinyYOLOv3 and TinyYOLOv4 to achieve high frames per second (FPS). However, this design has the limitation of being accompanied by significant degradation in accuracy. Sharma et al. [12] achieved high throughput by designing a well-pipelined and re-configurable hardware accelerator layer; however, it has the limitation of requiring a relatively large amount of hardware resources. In summary, there is a shortage of practical accelerator research that considers optimization from both the algorithm and architecture perspectives in terms of comprehensive trade-offs, such as accuracy, throughput, and hardware resources.

To overcome the limitations of previous studies, this brief proposes an optimal TinyYOLOv3 accelerator that applies various techniques across algorithms and architectures. The contribution of this brief is summarized as follows:

- Through the design of a fully pipelined streamline architecture, all layers are designed to operate in parallel to maximize hardware utilization. In addition, excluding the

Manuscript received 3 April 2023; revised 9 June 2023; accepted 22 June 2023. Date of publication 26 June 2023; date of current version 25 September 2023. This work was supported by the Ministry of Science and ICT (MSIT), South Korea, through the Information Technology Research Center (ITRC) Support Program supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant IITP-2023-RS-2022-00156295. This brief was recommended by Associate Editor Z. Di. *(Subin Ki and Juntae Park are co-first authors.) (Corresponding author: Hyun Kim.)*

The authors are with the Department of Electrical and Information Engineering and the Research Center for Electrical and Information Technology, Seoul National University of Science and Technology, Seoul 01811, South Korea (e-mail: subin97@seoultech.ac.kr; juntaepark@seoultech.ac.kr; hyunkim@seoultech.ac.kr).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSII.2023.3289514.

Digital Object Identifier 10.1109/TCSII.2023.3289514

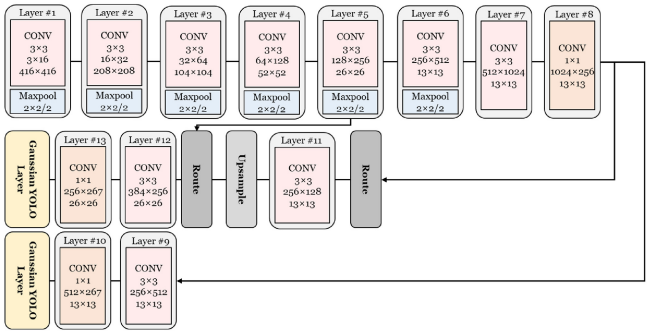

Fig. 1. The structure of the Gaussian TinyYOLOv3.

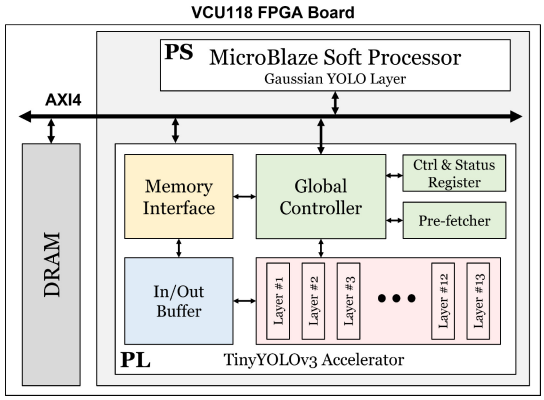

Fig. 2. Block diagram of the overall architecture.

activations of the route and YOLO layers, all parameters are pre-loaded into the on-chip memory, and all operations are designed to be computed only with the on-chip memory, thereby minimizing external memory access and improving the throughput.

- By designing hardware-friendly shift-based quantization and a shift-based floating-fixed multiplication and accumulation (MAC) unit, hardware resource-saving and throughput improvement are achieved with minimal accuracy degradation. In addition, Gaussian modeling in the YOLO layer is implemented in the processing system (PS) of FPGA to minimize the accuracy drop.
- The proposed design is fully implemented on Xilinx XCVU9P and achieves more than 1.7× higher throughput (351.1 GOPS) and 0.91% higher accuracy (34.01% on the COCO2014 dataset) with relatively fewer resources compared to the existing state-of-the-art study [13].

II. Proposed Architecture

A. Overall Architecture

The structure of the Gaussian TinyYOLOv3 model and the overall architecture of the proposed accelerator are illustrated in Fig. 1 and Fig. 2, respectively. In contrast to the recursive architecture that performs the layer operations sequentially and repeatedly by implementing a general-purpose processing engine that can handle the entire model computation process, in the designed streamline architecture, all layer operation units are pipelined and fully utilized by implementing the optimized operation units for each of the 13 layers, except for the Gaussian YOLO layer. The global controller controls

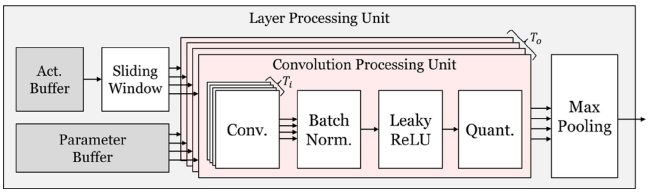

Fig. 3. Block diagram of the processing unit module for each layer.

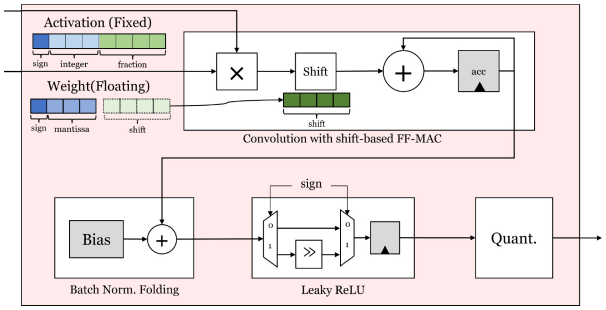

Fig. 4. Block diagram of the convolution processing unit.

the overall data flow and external memory access, and the pre-fetcher determines and controls whether to perform a pre-fetch for data that require memory read. Here, we implement the streamline architecture to store the parameters (*i.e.*, weight and activation) of all layers except for the fifth layer required by the route layer in on-chip memory (*i.e.*, In/Out Buffer); thus, external memory access for parameter fetch can be minimized and high throughput can be achieved with a pipeline scheme. The FPGA PS is designed to handle post-processing, such as Gaussian modeling operations in the Gaussian YOLO layer, which are inefficient to implement in the FPGA PL.

B. Processing Units

The fundamental design of the layer processing unit (PU) in this brief follows the structure illustrated in Fig. 3. The input data are stored in the input buffer (*i.e.*, Act. Buffer in Fig. 3) and passed to the sliding window module. The sliding window module generates a 3×3 window, allowing the input data to be used as the input for the CONV module. A CONV operation is then performed using the activation extracted from the 3×3 window and the weight and bias extracted from the parameter buffer. The data resulting from the CONV operation are passed through the batch normalization (BN) and activation modules (*i.e.*, Leaky ReLU) before being fed into the quantization module. If the layer includes max-pooling, the PU for that layer is implemented using a max-pooling operation module.

The configuration of the CONV PU is shown in Fig. 4. In this brief, CONV operations using floating-point weights and fixed-point features are replaced with shift operations based on the proposed 8-bit shift-based quantization, resulting in a significant reduction in hardware resource. Detailed information on shift-based quantization and the supporting MAC operator is provided in Section III-C. Additionally, to reduce the complexity of BN operations following CONV, BN folding [14] is applied that replaces existing complex operations with the simple addition of bias. The commonly used alpha value for the Leaky ReLU function is 0.01. However, in this brief, to efficiently handle multiplication operations, the alpha value is

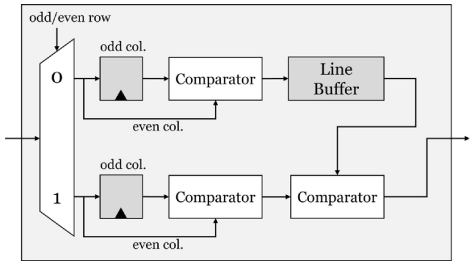

Fig. 5. Block diagram of the max pooling unit.

approximately used as 0.09375 ($=2^{-4}+2^{-5}$). To implement this approximate Leaky ReLU module, the sign of the input is first checked using a multiplexer (MUX) in the corresponding module. If the MUX value is 1, the input is directly output; whereas if it is 0, the values shifted by 4 and 5 are added to replace the multiplier operation, thereby saving the hardware resources of the activation module. Finally, the values from the 16-bit accumulator are quantized to 8-bit through the quantization module and sent as input data to the next layer. In the next layer, a CONV operation is performed using this quantized activation and the pre-quantized weight in weight buffers.

The max-pooling module is configured as shown in Fig. 5. Depending on whether the row of input data is odd or even, the order of the operations is implemented differently to minimize line buffer use. For odd rows, the data for odd columns are first stored, and the larger value between the odd column data and the even column data is stored in the line buffer. For even rows, the same method as for odd rows is applied to compare the column data values of odd and even rows. Finally, max pooling is performed by comparing the larger value with the data stored in the line buffer. This structure enables efficient max-pooling operations by utilizing a line buffer with a size that is half the row size of the input feature map.

C. Floating-Fixed MAC With Shift-Based Quantization

In the design of CNN-based hardware accelerators, the CONV module consumes a significant amount of hardware resources, with multipliers generally accounting for the largest portion. Consequently, various studies have been conducted on efficient MAC design methods [15]. In particular, [16] demonstrated that using a mixed form of floating-point and fixed-point representations can reduce the number of multipliers and the complexity of shifters compared to using only fixed-point multipliers when the bit-width is relatively small (e.g., 8-bit or less). Based on these result, this brief designs a TinyYOLOv3 hardware accelerator by applying a floating-fixed MAC (FF-MAC) operator that quantizes weights and activation maps into 8-bit floating-point and fixed-point formats, respectively, and performs CONV operations with these quantized parameters. In particular, to design a more efficient MAC than that in [16], this brief proposes the FF MAC operator that applies shift-based quantization techniques to minimize hardware resources.

1) Shift-Based Quantization: The proposed shift-based quantization calculates the input activation scale (IS), weight scale (WS), and accumulate scale (AS) for each layer based on the minimum and maximum values of input activations, weights, and accumulate activations of each layer, and performs layer adaptive linear quantization with these parameters as follows:

$$shift = AS - ((IS + M) - (E - B)) \tag{1}$$

where M, E, and B correspond to the mantissa bit, exponent, and bias of the weight, respectively. The difference between the proposed and existing techniques is that IS and AS are used to unify the shift operation direction within shift-based FF-MAC. In the existing FF-MACs, many hardware resources are required in the process of determining the shift direction and shifting to the other direction due to the non-uniform shift direction. In contrast, in this brief, the shift direction is unified in advance to one direction through (1). We unify the shift direction for each layer by adjusting the AS value. For example, in the case of a layer to unify the shift value into a positive number, if the shift value is already positive with the existing AS value, this AS is used as it is, but if the shift value is negative with the existing AS value, the AS value is adjusted to the minimum so that the shift value becomes a positive number. The shift direction unified through AS is determined as a more dominant sign in the sign distribution of shift values within the layer when the existing AS value is used. The finally determined shift value is used for the quantized weights of shift-based FF-MACs in the {sign, mantissa, shift} format.

2) Floating-Fixed MAC: After the shift value is determined through model training including shift-based quantization, the CONV unit in Fig. 4 loads and uses this value. We perform the MAC operation using quantized 8-bit floating weights in {sign(1b), mantissa(3b), shift(4b)} format and quantized 8-bit fixed-point input activation in {sign(1b), integer(3b), fraction(4b)} format. The proposed MAC operator first multiplies the 8-bit input activation and 4-bit weight excluding the shift part. After that, the resulting value is shifted by the weight shift value according to the shift direction determined for each layer, and subsequent values are continuously accumulated. To this end, the proposed shift-based FF-MAC in Fig. 4 is implemented with a multiplier for 8-bit activation and 4-bit weight, a shifter according to the weight shift value, and an accumulator. Consequently, the proposed quantization method and supporting MAC operator demonstrate more efficient hardware performance while maintaining minimal accuracy degradation compared with fixed-point multipliers.

D. Gaussian YOLO Processing

The Gaussian TinyYOLOv3 model is similar in structure to the TinyYOLOv3 model but uses a Gaussian YOLO layer instead of the final YOLO layer for object detection. The existing YOLO layer has only bounding box (bbox) coordinates and class probability information; thus, there is no indicator of how much the bbox matches the object. In contrast, Gaussian YOLO layer [5] applies Gaussian modeling to the bbox for calculating the mean and variance of each coordinate, redesigns the loss function, and utilizing the output variance as uncertainty in post-processing tasks such as non-maximum suppression (NMS) in order to substantially compensate for the relatively low accuracy of lightweight object detectors while maintaining the real-time processing speed. Although complex operations such as exponential and

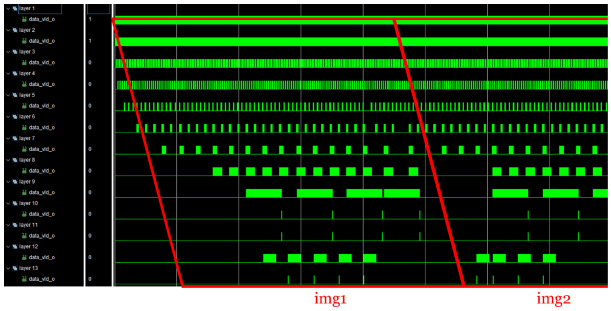

Fig. 6. Waveform of RTL simulation for each layer in the proposed IP.

sigmoid operations included in the Gaussian modeling process and NMS can be easily processed in CPUs, they are not suitable for gate-level designs, making them difficult to implement in FPGA programmable logic (PL) or ASIC designs [19], because each information must be stored in on-chip memory (OCM) during operations and the significant hardware resource usage and additional latency are incurred.

Therefore, in this brief, the FPGA PL and PS are co-designed to process a Gaussian YOLO layer on an ARM processor. Fig. 6 shows a valid output signal for each layer of the proposed accelerator. In the PL, each layer operates in parallel, and the 10th and last CONV layers output two detection values (*i.e.*, 13 × 13 / 26 × 26 output tensors) for each image. The values for each YOLO layer of each input image are stored in the external DRAM as they are generated, and a signal is sent to begin the Gaussian YOLO operation in the PS. The PS sequentially processes the first and second Gaussian YOLO layers, and each YOLO layer performs detection operations including NMS presented in [1]. After completing both Gaussian YOLO layer operations, the box coordinates are calculated and plotted on the input image. While the PS processes the Gaussian YOLO layer and box coordinates, the PL calculates the detection value for the next image and stores the result in external DRAM. The parallelism of the PL is adjusted to ensure that the latency of generating detection values in the PL is shorter than the latency of PS post-processing such that the PS operation can run without stalling. In detail, we try to minimize the processing bottleneck by considering the difference in processing speed between the PL and PS. In the PL, two output tensors for each image are created with an interval of 12.4 ms. On the other hand, the PS consumes 15.9 ms on average to perform the Gaussian YOLO layer operation on the two output tensors along with the execution of the interrupt service routine. Therefore, processing in the PL is delayed for about 3.5 ms (*i.e.*, 12.4% of the operation time for one image) despite the pipeline structure. Nevertheless, as can be seen in the experimental results, the proposed accelerator with this PL-PS co-design achieves the highest throughput and accuracy compared to previous studies.

III. Experimental Results

A. Experimental Environments

In this brief, hyperparameters such as batch size, learning rate, and epochs required for model training follow the same settings of [5]. We trained the reference code implemented in the DarkNet framework on COCO 2014 [20] and Pascal VOC2007 [21] using Titan-XP 2GPU. Xilinx Vivado 22.1 and Vitis 22.1 are used for accelerator implementation, and a Xilinx XCVU9P FPGA board is used as the target board.

TABLE I
Comparison of Logic Resources Among MAC Operators

	LUT	FF	CARRY8	DSP
32b-Fixed-Fixed MAC	38670	5120	2270	4
8b-Fixed-Fixed MAC	21157	4905	2050	0
8b-Floating-Fixed-MAC [16]	18746	4273	1912	0
8b-Proposed	12381	3800	440	0

TABLE II
Evaluation of Shift-Based Quantization Performance

Dataset	Model	mAP (%)			
		Baseline	Quant. [16]	**Proposed**	Drop
		(32-bit)	(8-bit)	**Quant. (8-bit)**	([16]-Proposed)
COCO 2014	TinyYOLOv3	33.1	32.81	32.79	0.02
	Gaussian TinyYOLOv3	34.38	34.07	34.01	0.06
VOC 2007	TinyYOLOv3	68.54	68.22	68.18	0.04
	Gaussian TinyYOLOv3	69.37	69.07	69.02	0.05

B. Shift-Based FF MAC Operator and Shift-Based Quantization

Table I presents the results of synthesizing a MAC comprising a single multiplier and an accumulator using four different methods. Notably, 32b-Naive Fixed-Fixed MAC implies using 32-bit fixed-point format for both weights and activations. Compared with the 32b-fixed-fixed MAC and 8b-fixed-fixed MAC, the proposed shift-based FF-MAC unit uses significantly fewer look-up tables (LUTs) (34.8% and 43%, respectively) and CARRY8s (50% and 60%, respectively). Furthermore, it used 33.8% fewer LUTs than the previous method [16]. Using this MAC unit can save a significant amount of hardware resources when implementing hundreds of MACs.

Table II presents a comparison of the mean average precision (mAP) of the proposed shift-based quantization for TinyYOLOv3 and Gaussian TinyYOLOv3. The proposed method shows an average mAP drop of 0.35% and 0.04% compared with the baseline and the existing method [16], respectively. This means that the proposed scheme achieves a significant reduction in logic resources, as shown in Table I, with relatively little accuracy drop. In particular, the proposed Gaussian modeled accelerator achieves higher accuracies of 0.91% and 0.48% on the COCO 2014 and Pascal VOC2007 datasets, respectively, compared with the 32-bit TinyYOLOv3 because of the implementation of Gaussian modeling on the FPGA PS.

C. Performance Comparison With Various Accelerator Designs

Table III presents a comparison of the implementation results of the proposed Gaussian TinyYOLOv3 accelerator with those of previous studies. Results not provided in the previous studies are indicated by a dash (-). The proposed accelerator achieves the highest 62.9 FPS, which is possible because of the well-pipelined and parallel operation of all the logic in the streamline architecture, as shown in Fig. 6. Additionally, it achieves approximately

TABLE III
PERFORMANCE COMPARISON WITH PREVIOUS WORKS

	[17]	[18]	[7]	[10]	[11]	[12]	[13]	**Proposed**
Year	2020	2022	2020	2021	2021	2021	2022	2023
Model	TinyYOLOv2	TinyYOLOv2	TinyYOLOv3	TinyYOLOv3	TinyYOLOv3	TinyYOLOv3	TinyYOLOv3	TinyYOLOv3
Platform	Xilinx XCZU9EG	Xilinx XC7Z045	Xilinx XC7Z020	Xilinx XCKU040	Xilinx XC7Z020	Xilinx XCVU9P	Intel 10AX115	Xilinx XCVU9P
Freq.(MHz)	300	200	100	143	100	200	200	150
OCM(KB)	1,105	508.5	185	984	120	-	6,095	9,166
DSPs	609	448	160	839	208	2693	1122	96
LUTs(k)	95.1	99.4	25.9	139	33.4	17.7	146.1 (Altera ALMs)	132
FFs(k)	90.6	98.9	46.7	-	-	145.7		39.5
Image Size	416×416	416×416	416×416	416×416	416×416	416×416	416×416	416×416
Precision	16b	16b	16b	16b	16b	-	32b	8b
Accuracy(%)	-	-	30.9	-	30.8	-	33.1%	34.01
FPS	16.1	-	1.88	32.4	14.7	32.1	36.3	62.9
Throughput (GOPS)	102	138.8	10.45	180	-	166.4	202	351.1
Power(W)	-	-	3.36	3.87	-	-	-	5.52
Power Eff. (GOPS/W)	-	-	3.11	46.51	-	-	-	63.61

3.2% higher accuracy than previous studies [7], [11] on the COCO dataset. In terms of hardware resources, the proposed accelerator uses a relatively large OCM (*i.e.*, Quantized weight: 8,910KB, Bias+Scale: 4KB, Intermediate activation: 117KB, and DRAM RD/WR: 135KB) owing to the design of the streamline architecture; however, it is implemented with the fewest digital signal processing (DSP) blocks and flip-flops (FFs). In addition, the power efficiency (*i.e.*, throughput per power consumption) of the proposed accelerator achieves 63.61 GOPS/W, which is about 20.5× and 1.4× superior to previous studies [7] (3.11 GOPS/W) and [10] (46.51 GOPS/W), respectively. Consequently, the proposed accelerator is superior to that of previous studies in terms of throughput, hardware resources, and model accuracy trade-offs.

IV. CONCLUSION

This brief proposes a dedicated FPGA implementation of a Gaussian TinyYOLOv3 accelerator using a fully pipelined streamline architecture with the hardware-friendly shift-based FF-MAC operator and PL-PS co-design. Consequently, the proposed accelerator is superior to that of previous research in terms of overall performance and is most suitable for use in mobile/edge devices. We anticipate that the proposed design will accelerate the commercialization of CNN-based object detectors for various mobile/edge devices.

REFERENCES

[1] J. Choi, D. Chun, H. Kim, and H.-J. Lee, "Gaussian YOLOV3: An accurate and fast object detector using localization uncertainty for autonomous driving," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 502–511.

[2] R. J. Wang, X. Li, and C. X. Ling, "Pelee: A real-time object detection system on mobile devices," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 1967–1976.

[3] J. Redmon and A. Farhadi, "YOLOV3: An incremental improvement," 2018, *arXiv:1804.02767*.

[4] 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 Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1861–1873, Aug. 2019.

[5] J. Choi, D. Chun, H.-J. Lee, and H. Kim, "Uncertainty-based object detector for autonomous driving embedded platforms," in *Proc. 2nd IEEE Int. Conf. Artif. Intell. Circuits Syst.*, 2020, pp. 16–20.

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

[7] Z. Yu and C.-S. Bouganis, "A parameterisable FPGA-tailored architecture for YOLOV3-tiny," in *Proc. 16th Int. Symp. Appl. Reconfig. Comput.*, 2020, pp. 330–344.

[8] A. Ahmad, M. A. Pasha, and G. J. Raza, "Accelerating tiny YOLOV3 using FPGA-based hardware/software co-design," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2020, pp. 1–5.

[9] 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. 141890–141913, 2021.

[10] D. Pestana, P. Miranda, J. Lopes, R. Duarte, M. Véstias, H. Neto, and J. De Sousa, "A full featured configurable accelerator for object detection with YOLO," *IEEE Access*, vol. 9, pp. 75864–75877, 2021.

[11] P. Miranda et al., "Configurable hardware core for IoT object detection," *Future Internet*, vol. 13, no. 11, p. 280, 2021.

[12] M. Sharma, R. Rahul, S. Madhusudan, S. Deepu, and D. S. Sumam, "Hardware accelerator for object detection using tiny YOLO-V3," in *Proc. IEEE 18th India Council Int. Conf.*, 2021, pp. 1–6.

[13] V. Herrmann, J. Knapheide, F. Steinert, and B. Stabernack, "A YOLO V3-tiny FPGA architecture using a reconfigurable hardware accelerator for real-time region of interest detection," in *Proc. IEEE 25th Euromicro Conf. Digit. Syst. Design (DSD)*, 2022, pp. 84–92.

[14] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 2704–2713.

[15] H. Chen et al., "AdderNet: Do we really need multiplications in deep learning?" in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 1468–1477.

[16] L. Lai, N. Suda, and V. Chandra, "Deep convolutional neural network inference with floating-point weights and fixed-point activations," 2017, *arXiv:1703.03073*.

[17] S. Zhang, J. Cao, Q. Zhang, Q. Zhang, Y. Zhang, and Y. Wang, "An FPGA-based reconfigurable CNN accelerator for YOLO," in *Proc. IEEE Int. Conf. Electron. Technol.*, 2020, pp. 74–78.

[18] H. Hongmin, L. Xueming, Q. Yadong, H. Xianghong, and X. Xiaoming, "An efficient parallel architecture for convolutional neural networks accelerator on FPGAs," in *Proc. 6th Int. Conf. High Perform. Compilation Comput. Commun.*, 2022, pp. 66–71.

[19] Z. Xu, J. Yu, C. Yu, H. Shen, Y. Wang, and H. Yang, "CNN-based feature-point extraction for real-time visual SLAM on embedded FPGA," in *Proc. IEEE 28th Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2020, pp. 33–37.

[20] T.-Y. Lin et al., "Microsoft COCO: Common objects in context," in *Proc. 13th Eur. Conf. Comput. Vis.*, 2014, pp. 740–755.

[21] M. Everingham, S. Eslami, L. Van Gool, C. Williams, J. Winn, and A. Zisserman, "The Pascal visual object classes challenge: A retrospective," *Int. J. Comput. Vis.*, vol. 111, pp. 98–136, Jan. 2015.