Hardware Accelerator for Object Detection using Tiny YOLO-v3

Manan Sharma
Dept. of ECE
NITK, Surathkal
Mangalore, Karnataka, India
sharma.manan@outlook.com

Rahul R
Dept. of ECE
NITK, Surathkal
Mangalore, Karnataka, India
rr92286@gmail.com

Madhusudan S
Dept. of ECE
NITK, Surathkal
Mangalore, Karnataka, India
madhu07apr98@gmail.com

Deepu S.P.
Dept. of ECE
NITK, Surathkal
Mangalore, Karnataka, India
deepusp123@gmail.com

Sumam David S.
Dept. of ECE
NITK, Surathkal
Mangalore, Karnataka, India
sumam@ieee.org

2021 IEEE 18th India Council International Conference (INDICON) | 978-1-6654-4175-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/INDICON52576.2021.9691658

Abstract—**For applications that require object detection to be performed in real-time, this paper presents a custom hardware accelerator, implementing state of the art Tiny YOLO-v3 algorithm. The proposed architecture achieves a reasonable trade-off between the speed of computation (measured in frames per second or FPS) and the hardware resources required. Each CNN layer is pipelined and parameterized to make the complete design re-configurable. The proposed hardware accelerator was synthesized using the SCL(Semi-Conductor Laboratory, India) 180 nm CMOS process and also using Vivado Xilinx software with Virtex Ultrascale+ FPGA as the target device. The pipelined architecture, along with other architectural novelties, provided a higher frame-rate of 32.1 FPS and a performance of 166.4 GOPS at 200 MHz clock frequency.**

Index Terms—**Object detection, Convolutional Neural Network, Hardware accelerator, YOLO, Tiny YOLO-v3**

I. Introduction

Object detection is an essential aspect of applications like self-driving cars. Object detection algorithms use Convolutional Neural Networks (CNNs), which are computationally very expensive, consume a considerable amount of power and hardware resources, and generally need a large number of memory accesses. After training these CNNs on GPUs or other dedicated hardware platforms, the inference can be run on either general-purpose CPU or GPUs. These two options always provide a trade-off: GPUs are faster (higher FPS), but consume more power, whereas general purpose CPUs consume lesser power than GPUs, but are also slower in computation (lower FPS). Hence, the need for custom hardware accelerators (on ASIC, FPGA, etc) arises so as to achieve a reasonable trade-off between these two. State of the art Tiny YOLO-v3 algorithm is employed for the purpose of object detection in this paper, as it provides a high frame-rate at low computational cost, compared to other algorithms like R-CNNs and Single Shot Detector.

Extensive research has been done in the field of hardware accelerator for CNNs. Many hardware accelerators have been implemented on FPGAs, using high-level synthesis. The FPGA implementation by Liu et al. [1] is one of the first work to accelerate the state-of-the-art YOLO-v2 with real-time performance (achieving a speed of 18.6 FPS) as well as $<1\%$ accuracy drop on Intel Arria 10 GX1150 FPGA for public VOC2007 test set. This implementation proposes a flexible Hardware/Software co-design framework using commercial high-level OpenCL language and the standard open-source deep learning framework Caffe. Nguyen et al. [2] presents a high-throughput FPGA implementation of YOLO using Xilinx Virtex-7 VC707 device, which achieves a throughput of 1.877 Tera Operations Per Second (TOPS) at 200 MHz. The VC707 was used to meet the higher requirement of on-chip memory. Wai et al. [3] proposed the implementation of a CNN-based object detection model: Tiny-Yolo-v2 on Cyclone V PCIe Development Kit FPGA board using OpenCL, achieving a peak performance of 21 Giga Operations Per Second (GOPS) and 3.06 FPS under 100 MHz operating frequency. Ahmad et al. [4] proposed an implementation of hardware accelerator for Tiny-YOLOv3, using a hardware-software co-design approach, with Xilinx Virtex 7 FPGA as target platform, and achieving 460.8 GOPS, with a precision of 18 bits for operations.

This paper presents a design which implements the following novel modifications compared to other conventional deep learning accelerators - every layer has only one batch normalization and activation unit. The different filters in a CNN layer share the hardware for these two operations. Faster layers reuse the filters, thereby reducing the hardware resources needed, and design of a Finite State Machine(FSM) for Maxpool unit which uses a single comparator, instead of multiple-comparator architecture used in many of the earlier implementations. The architecture discussed can be easily extended to other deep learning models having CNNs and batch normalization layers. The paper is organized as follows. Section II gives an overview of a general Tiny YOLO-v3 architecture. Section III gives a detailed explanation of the proposed implementation of the architecture, the timing calculations and the memory requirements. Section IV shows the experimental results obtained and conclusions are given in Section V.

Madhusudan S, Rahul R and Manan Sharma have contributed equally to this work

978-1-6654-4175-9/21/$31.00 ©2021 IEEE

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on June 02,2025 at 07:02:52 UTC from IEEE Xplore. Restrictions apply.

II. Architectural Overview of Tiny YOLO-v3

YOLO (You Only Look Once)-v3 is a popular single-shot CNN-based algorithm for object detection [5]. It eliminates the need for region proposal step commonly seen in Mask R-CNN and other such two-step detection frameworks. Instead, it directly predicts the bounding box coordinates. Tiny YOLO-v3 is a lighter version of YOLO-v3 containing fewer layers to facilitate its deployment in resource constraint scenarios. It takes in an RGB image of 416×416 resolution and divides it into S×S grids. The architecture of the neural network is shown in Fig 1.

As shown in the diagram, there are two branches in the network, predicting bounding boxes at two different scales. The first scale divides the network into 13×13 grids, while the second scale divides the network into 26×26 grids. If the centre of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each bounding box consists of 5 predictions: x, y, w, h, and confidence. The (x, y) coordinates represent the centre of the box relative to the bounds of the grid cell. The width of the box (w) and height of the box (h) are predicted relative to the whole image. The confidence scores determine the model's confidence in the presence of the detected object inside the given bounding box, and its judged accuracy. Each grid cell also predicts C conditional class probabilities. Hence, the final output prediction is encoded as an S × S × (B × 5 + C) tensor, where B is the number of anchor boxes. Anchor boxes are initial dimensions of the box (height, width) which are resized to the object size using outputs from the neural network. The CNN here is required only to adjust the size of the nearest anchor to the size of the object, and not predict the size of the object. [5]

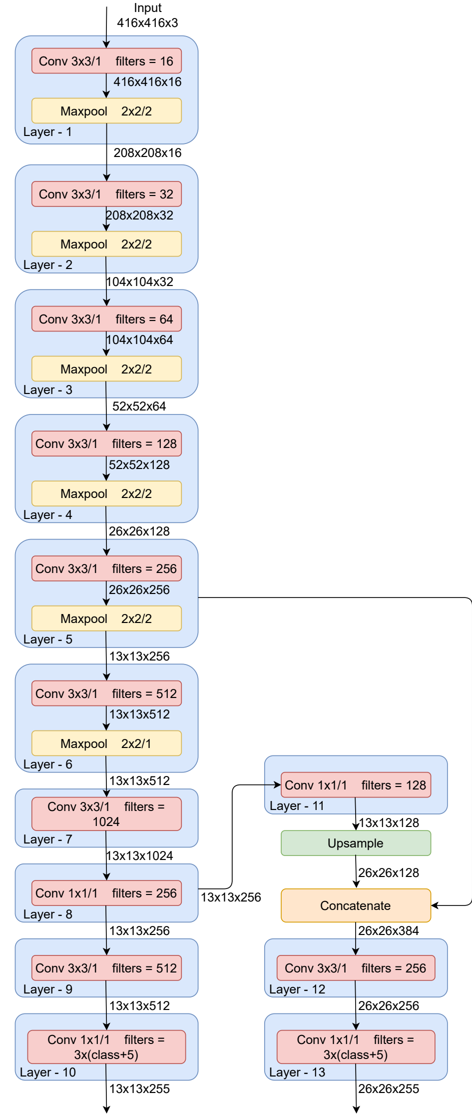

Fig. 1. CNN architecture of Tiny YOLO-v3

A. General CNN Architecture

The four primary operational modules of the CNN used in the Tiny YOLO-v3 algorithm are-

1) 3-Dimensional Convolution: A sequential module which implements MAC (Multiply-and-accumulate) operation between the kernel and a sliding window applied on the input feature. The number of operations, N_{ops}, and the number of clock cycles, N_{cycles} taken by convolution can be calculated as follows

$$N_{cycles} = N_{in} \times K \times K \times H_{out} \times W_{out} \tag{1}$$

$$N_{ops} = 2 \times N_{in} \times K \times K \times N_{out} \times H_{out} \times W_{out} \tag{2}$$

$$N_{ops} = 2 \times N_{cycles} \times N_{out} \tag{3}$$

where N_{in} is the number of channels of input features, N_{out} is the number of filters in that layer, K is the filter size in that layer, H_{out} and W_{out} are height and width of output feature of convolution layer, and the factor of 2 appears in equation (2) as two operations, that is, addition and multiplication-performed in every MAC operation.

2) Batch Normalization: A combinational module performing batch normalization on the output of the convolution layer. During the training of CNN, data is pre-processed to resemble a normal distribution, so as to speed up the CNN training and prevent overfitting. Hence, we need to account for this normalization during the inference phase of the CNN as well. Batch normalization uses mainly four parameters-Scale(γ), Shift(β), Mean($E[m]$) and Variance($Var[m]$), and the corresponding equation is, [6]

$$n = \frac{\gamma}{\sqrt{Var[m] + \epsilon}} \times m + \left(\beta - \frac{\gamma E[m]}{\sqrt{Var[m] + \epsilon}} \right) \tag{4}$$

where m and n are the input and output of batch normalization module respectively, and ϵ is a very small positive number.

3) Activation: A combinational module is used to introduce a non-linear functional mappings between the inputs and response variable. The activation function $f(z)$ being used here is Leaky ReLU

$$f(z) = \begin{cases} 0.01z & \text{for} \quad z < 0 \\ z & \text{for} \quad z \geq 0 \end{cases} . \tag{5}$$

4) Pooling: A sequential module used to reduce the spatial size of feature maps. Maxpooling is a method where a region in the feature map (in Tiny YOLO-v3, a 2x2 grid) is replaced by the maximum value in that region, thereby reducing its dimensions.

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on June 02,2025 at 07:02:52 UTC from IEEE Xplore. Restrictions apply.

III. Proposed Tiny YOLO-v3 Architecture Implementation

The block diagram of the architecture of the proposed Tiny YOLO-v3 accelerator is shown in Fig. 2.

In this implementation, the weights and other parameters of the CNN are obtained by training the CNN on GPU using Tensorflow. These parameters are then quantised and converted to fixed-point binary number format and then stored in the main memory of ASIC, whose functionality was emulated through the testbench. The input video feed is then read frame by frame from memory. The kernel weights and batch normalization constants used for convolution are also read from the main memory of the hardware accelerator every frame. Separate hardware is mapped to each layer of CNN, and these different layers of CNN operate parallelly in a pipelined fashion to ensure higher computational speeds, but the time required per frame is different for each layer. Hence, the computational speed of the entire network depends on the slowest layer and pipelining requires a control signal which stalls the other layers till the slowest layer completes its computations.

In a given layer, convolution operations of different filters are parallelised by mapping separate hardware for convolution corresponding to each filter. Similarly, maxpooling is parallelised by mapping separate hardware corresponding to each output feature map (obtained from convolution). As batch normalisation and activation operations use lesser number of clock cycles, these two operations are scheduled such that every layer can have only one batch normalization and activation unit each, thereby different filters in a given layer can share the hardware. At the hardware level, this scheduling is achieved by multiplexing the convolutional modules' output using a counter for select signal, and the multiplexer output is then passed through batch normalization and activation units, and then demultiplexed to apply as inputs to the parallelised maxpooling units. Thus, this scheduling helps to reduce the hardware resource requirements of the accelerator without reducing the overall speed (FPS).

This architecture is designed such that faster layers are stalled and kept idle till the slowest layer completes its operations. Unlike the slowest layer of the CNN, a separate MAC unit may not be required for each of the kernels in the faster layers, as that would consume more hardware resources without giving any additional performance in terms of speed. Hence, each of the faster layers can be designed to reuse the MAC unit hardware for its different kernels, thereby utilising this idle time and also ensuring that their new computational speeds are not less than that of the slowest layer. The number of convolution/MAC units instantiated for each of the faster layers can be reduced by a factor called the $Reusability\ Factor$ (Equation (6)). This helps reduce the hardware resource required for the accelerator without reducing the overall speed/throughput. If N_1 and N_2 are the number of clock cycles taken for computation by the slowest CNN layer and given CNN layer respectively, when a separate MAC/convolution unit is instantiated for each kernel in a layer, then the $Reusability\ Factor$ of the given CNN layer is -

$$Reusability\ Factor = N_1/N_2 \tag{6}$$

1) Implementation of the Convolution module: Kernel and the input feature maps are stored in the main memory of the hardware accelerator. The address-generator module generates the addresses needed for kernel and input features, and the values are accessed one-by-one from memory and sent to the MAC unit, which performs one MAC operation per one clock cycle. The control unit generates the required signal to reset the accumulator after every MAC operation, by keeping count of the number of operations. Architecture of the convolution unit is shown in Fig. 3.

2) Implementation of Batch Normalization module: In an attempt to reduce the number of computations during the inference phase, we introduce two parameters b_1 and b_2 Let,

$$b_1 = \frac{\gamma}{\sqrt{Var[m] + \epsilon}}, \quad b_2 = \beta - \frac{\gamma E[m]}{\sqrt{Var[m] + \epsilon}}$$

These parameters are computed beforehand, and are stored in main memory. Thus the equation (4) can be reduced to the form below.

$$n = b_1 m + b_2 \tag{7}$$

The architecture of the batch normalization unit is shown in Fig. 4.

3) Implementation of Activation module: The activation layer is implemented using a 2:1 MUX whose 'select' signal is the output of a comparator which checks whether the input is positive or negative. The architecture of the Leaky ReLU is shown in Fig. 5.

4) Implementation of Maxpool module: The proposed maxpool module consists of an FSM and a comparator as shown in Fig. 6. The FSM used for maxpooling operation is shown in Fig. 7. In the initial state (State=0), the input value read from main memory (according to the address sent by the FSM) is initialized to be the maximum value. As the FSM goes through the states, the comparator compares the previous maximum value and the present input value read from main memory, and updates the maximum value accordingly. After the final state (State=3), a single maxpool operation is completed and the FSM goes back to initial state. Thus, the FSM uses a single comparator for maxpooling unlike other architectures and reduces the hardware resources required. The number of clock cycles used by maxpool operation is therefore given by

$$N_{pool_cycles} = (4 \times H_{out} \times W_{out})/(S \times S) \tag{8}$$

where S is the stride of maxpool operation and it is either 2 or 1 in Tiny YOLO-v3 architecture.

A. Timing calculations

In a given layer, it can be seen from equations (1) and (8) that the time taken for maxpooling is negligible when compared to the time taken for convolution. Hence, for all timing calculations, only the timing requirements of convolution was considered. The timing details of the CNN architecture shown in Fig. 1, can be computed from equations (2) and (6), and are shown in Table I. It is evident from Table I that the second layer, being the slowest

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on June 02,2025 at 07:02:52 UTC from IEEE Xplore. Restrictions apply.

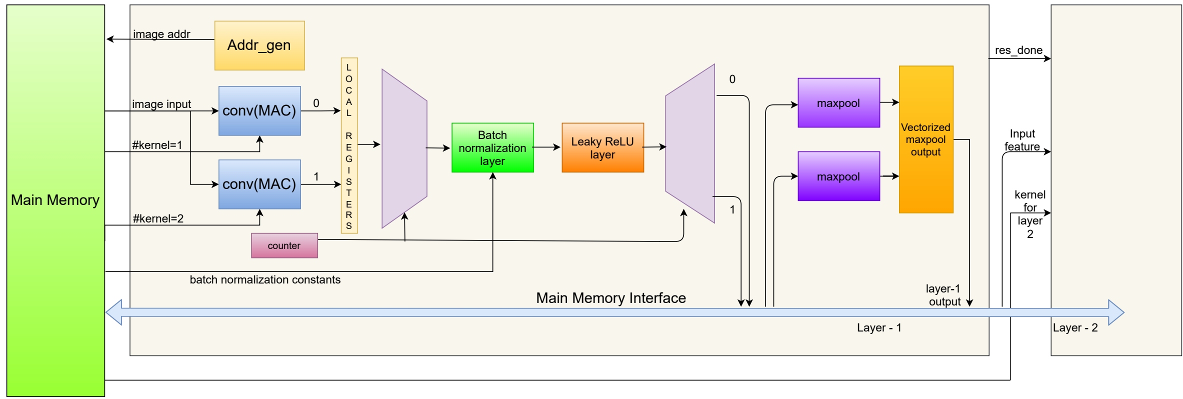

Fig. 2. Block diagram of the architecture of the proposed Tiny YOLO-v3 accelerator

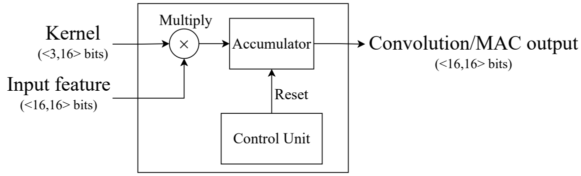

Fig. 3. Architecture of Convolution/MAC unit

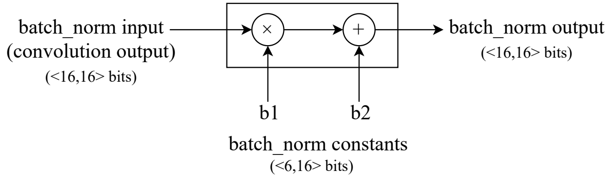

Fig. 4. Architecture of batch-normalization module

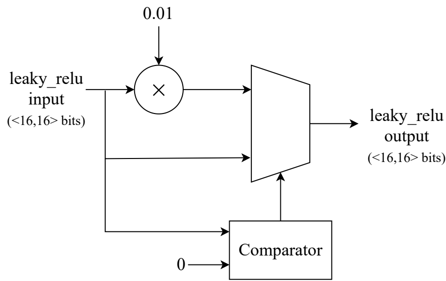

Fig. 5. Architecture of Leaky ReLU module

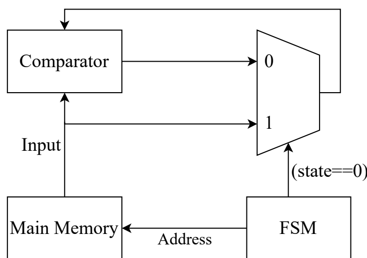

Fig. 6. Architecture of maxpool module

layer, takes the maximum number of clock cycles (6,230,016 cycles) and since we are instantiating the layers in a parallel fashion and pipelining them, the computational time taken per frame by the pipelined architecture would be the same

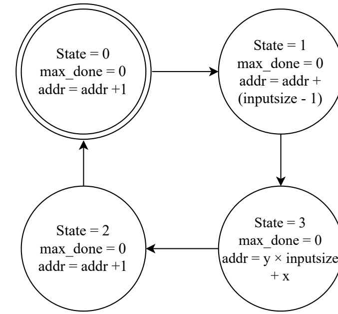

Fig. 7. FSM of 2×2 maxpooling operation

TABLE I
TIMING DETAILS OF DIFFERENT LAYERS OF CNN

Layer	N_{in}	H_{out}, W_{out}	K	N_{cycles}	Reusability Factor
Layer-1	3	416	3	4,672,512	1
Layer-2	16	208	3	6,230,016	1
Layer-3	32	104	3	3,115,008	2
Layer-4	64	52	3	1,557,504	4
Layer-5	128	26	3	778,752	8
Layer-6	256	13	3	389,376	16
Layer-7	512	13	3	778,752	8
Layer-8	1024	13	1	173,056	4
Layer-9	256	13	3	389,376	1
Layer-10	512	13	1	86,528	1
Layer-11	256	13	1	43,264	16
Layer-12	384	26	3	2,336,256	2
Layer-13	256	26	1	173,056	4

as the time taken by the slowest layer (that is, the second layer).

$$T_{pipeline} = 6,230,016 \times T_{clk}$$

For obtaining 30 FPS,

$$T_{clk} = \frac{1}{30 \times 6,230,016} \implies f_{clk} = 186.90048MHz \tag{9}$$

Using the result in Equation (9), at a standard clock frequency of 200 MHz,

$$\text{Number of FPS} = \frac{200 \times 30}{186.9005} = 32.10 \tag{10}$$

For a single CNN layer, number of operations per frame is given by Equation (1). At 200 MHz, the speed of object detection achieved is approximately 32 FPS (Equation (10)).

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on June 02,2025 at 07:02:52 UTC from IEEE Xplore. Restrictions apply.

Also, reusing of filters reduces the number of idle clock cycles in the computation of each of the other twelve faster layers. Thus,

$$\text{GOPS} = \frac{2 \times 6,230,016 \times 32 \times 32.1 \times 13}{10^9} \approx 166.4 \quad (11)$$

B. Memory requirement

The weights obtained by the training of the network include the kernel values and the batch normalization constants. This paper uses a signed fixed-point number system to store these trained parameters, and hence, all the computations are fixed-point computations. The range of kernel values and batch-normalization constants could be covered using 3 bits and 6 bits, respectively for the integer part. Hence, the kernel values were stored as <3,16> signed fixed-point numbers, and the batch normalization constants were stored as <6,16> signed fixed-point numbers. The total number of kernel values (N_{ker}) can be computed by the formula,

$$N_{ker} = N_{in} \times K \times K \times N_{out} \quad (12)$$

Since the kernels values are quantized and converted to <3,16> fixed point binary numbers, the total number of bits required to store the kernel values, $Bits_{ker}$ can be obtained using

$$Bits_{ker} = N_{in} \times K \times K \times N_{out} \times (3 + 16) \quad (13)$$

The total number of batch normalization constants are twice the number of filters in each layer. The batch normalization constants are quantized and converted to <6,16> bit fixed point binary numbers. Therefore, the total number of bits required to store the kernel values, $Bits_{bat}$ is

$$Bits_{bat} = 2 \times N_{out} \times (6 + 16) \quad (14)$$

The intermediate results of the convolutional modules are temporarily stored in the main memory before being fed to the subsequent maxpool modules. Dimensions (and thus the memory requirement) of output feature map of maxpool modules are lesser than that of corresponding convolution modules. Therefore, the results from the maxpool modules, over-write the results from the convolutional modules to reduce the overall memory requirements. Each intermediate value is stored as a <16,16> signed fixed-point number. Therefore the number of bits required to store the intermediate results of a given layer is,

$$Bits_{inter} = N_{out} \times H_{out} \times W_{out} \times (16 + 16) \quad (15)$$

It is evident from Table II and Table III that, 168,064,272 bits or about 20.03 MB is required to store the kernel values, 162,536 bits or about 19.84 KB to store the batch normalization constants and 197,256,800 bits or 23.51 MB to store the intermediate output. Thus the total memory requirement of this architecture is 43.56 MB.

TABLE II
MEMORY REQUIREMENTS OF DIFFERENT LAYERS OF CNN

Layer	N_{in}	N_{out}	K	$Bits_{ker}$	$Bits_{bat}$
Layer-1	3	16	3	8,208	704
Layer-2	16	32	3	87,552	1,408
Layer-3	32	64	3	350,208	2,816
Layer-4	64	128	3	1,400,832	5,632
Layer-5	128	256	3	5,603,328	11,264
Layer-6	256	512	3	22,413,312	22,528
Layer-7	512	1024	3	89,653,248	45,056
Layer-8	1024	256	1	4,980,736	11,264
Layer-9	256	512	3	22,413,312	22,528
Layer-10	512	255	1	2,480,640	11,220
Layer-11	256	128	1	622,592	5,632
Layer-12	384	256	3	16,809,984	11,264
Layer-13	256	255	1	1,240,320	11,220
Total				168,064,272	162,536

TABLE III
MEMORY REQUIREMENTS FOR STORAGE OF INTERMEDIATE VALUES

Layer	N_{out}	H_{out}, W_{out}	$Bits_{inter}$
Layer-1	16	416	88,604,672
Layer-2	32	208	44,302,336
Layer-3	64	104	22,151,168
Layer-4	128	52	11,075,584
Layer-5	256	26	5,537,792
Layer-6	512	13	2,768,896
Layer-7	1024	13	5,537,792
Layer-8	256	13	1,384,448
Layer-9	512	13	2,768,896
Layer-10	255	13	1,379,040
Layer-11	128	13	692,224
Layer-12	256	26	5,537,792
Layer-13	255	26	5,516,160
Total			197,256,800

IV. EXPERIMENTAL RESULTS

The pre-trained weights are obtained from [7] for COCO dataset. Tensorflow library in Python was used for the software implementation of inference phase of the CNN. The RTL design for the Hardware Accelerator was done in SystemVerilog, and synthesized using Cadence RTL compiler with SCL 180 nm CMOS process technology node. The design was also synthesised using Xilinx Vivado software, choosing Virtex Ultrascale+ FPGA as the target device. Arm of the CNN architecture with 26×26 grids detects smaller objects, and the other arm with 13×13 grids detects larger objects, as can be seen in Fig. 8 and Fig. 9 respectively. The combined result of both the arms of the CNN architecture gives better results for objects of all sizes, as in Fig. 10. To obtain the final bounding boxes, anchor box computation and non-max suppression is done in Python, using the final

Fig. 8. Output from the hardware implementation of 26×26 arm of Tiny-YOLO-v3 architecture, as obtained in RTL simulation

Fig. 9. Output from the hardware implementation of 13×13 arm of Tiny-YOLO-v3 architecture, as obtained in RTL simulation

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on June 02,2025 at 07:02:52 UTC from IEEE Xplore. Restrictions apply.

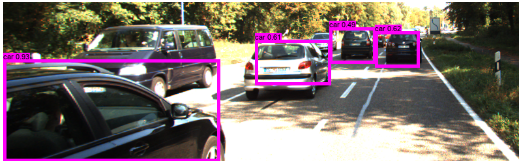

Fig. 10. Final Tiny YOLOv3 RTL simulation output of hardware implementation

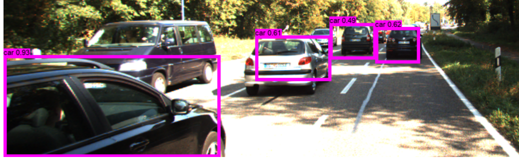

Fig. 11. Final Tiny YOLOv3 output from TensorFlow implementation

TABLE IV
RESOURCE USAGE SUMMARY FOR FPGA SYNTHESIS

Resource	Utilization	Available	Utilization Percentage
LUT	176501	1182240	14.93%
LUTRAM	2944	591840	0.50%
FF	145763	2364480	6.16%
DSP	2693	6840	39.37%

TABLE V
CELL REPORT OF SYNTHESIS USING 180 NM TECHNOLOGY

Type	Instances	Area	Area %
Sequential	69271	4208027.117	19.3
Inverter	76221	540365.850	2.5
Buffer	50	627.000	0.0
Logic	599094	17032918.150	78.2
Total	744636	21781938.116	100.0

CNN layer outputs obtained from RTL simulation. It was observed that the RTL simulation output obtained from the proposed hardware implementation (Fig. 10) is accurate and same as the software implementation output obtained from Tensorflow (Fig. 11), with an RMS error of 0.1282 and maximum error of 0.5908 between the final CNN layer output from Tensorflow and RTL simulation.

The resource utilisation report for the FPGA synthesis is given in Table IV. The synthesis report obtained for SCL-180 nm Process Design Kit (PDK) based implementation is given in Table V.

Comparison of performance of this design with previous works is shown in Table VI. We achieve $8\times$ GOPS as compared to [3] due to the parallelisation of CNN layers and convolution units. Ahmed et.al [4] report higher GOPS, at the tradeoff of usage of more parallel multipliers and adders as they process MAC operations of 3×3 feature map tile using 9 parallel instantiated DSPs in a single clock cycle. But this doesn't get reflected as higher hardware utilisation, because it uses a precision of only 16 bits (resulting in lower accuracy). In addition to this, their design only accelerates the convolutional and batch normalization layers on FPGA, as opposed to the proposed architecture discussed in this paper, which aims to accelerate all the layers of CNN.

V. CONCLUSION

This paper presents a hardware-efficient hardware accelerator for Object detection using Tiny YOLO-v3 algorithm.

TABLE VI
PERFORMANCE COMPARISON WITH PREVIOUS WORKS

	[3]	[4]	This Work
Device	Cyclone-V	Virtex 7 VC707	Virtex Ultrascale+ and custom ASIC
Network	Tiny YOLOv2	Tiny YOLOv3	Tiny YOLOv3
Design Tool	OpenCL	Vivado HLS	Vivado(for FPGA), Cadence(for ASIC)
Design Scheme	HW	HW/SW	HW
Modules accelerated on HW	Convolution, Batch Normalization, Maxpool, Activation	Convolution, Batch Normalization	Convolution, Batch Normalization, Maxpool, Activation
Precision(bits)	16	18	32
Frequency(MHz)	117	200	200
GOPS	21.6	460.8	166.4

The object detection output obtained is observed to be accurate, and the same as the output of the software model of the CNN implemented on Python using TensorFlow. The frame rate achieved is approximately 32.1 FPS at a clock frequency of 200 MHz and throughput of 166.4 GOPS. Thus, a reasonable trade-off, between performance and hardware resources required, is achieved in this implementation. Hence, the architecture proposed in this paper can significantly contribute to object detection in real-time. The future scope of this project includes performing power analysis of the accelerator, designing a memory controller and DMA engine to support high bandwidth memory.

VI. ACKNOWLEDGMENTS

The authors would like to thank the Special Manpower Development Programme for Chips to System Design (SMDP-C2SD) of the Ministry of Electronics and Information Technology (MietY), Government of India for supporting this work.

REFERENCES

[1] X. Xu and B. Liu, "FCLNN: A Flexible Framework for Fast CNN Prototyping on FPGA with OpenCL and Caffe," in *2018 International Conference on Field-Programmable Technology (FPT)*, Dec. 2018, pp. 238–241.

[2] D. T. Nguyen, T. N. Nguyen, H. Kim, and H. 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, Aug. 2019.

[3] Y. J. Wai, Z. bin Mohd Yussof, S. I. bin Salim, and L. K. Chuan, "Fixed Point Implementation of Tiny-Yolo-v2 using OpenCL on FPGA," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 10, 2018.

[4] A. Ahmad, M. A. Pasha, and G. J. Raza, "Accelerating Tiny YOLOv3 using FPGA-Based Hardware/Software Co-Design," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.

[5] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *CoRR*, vol. abs/1804.02767, 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

[6] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, p. 448–456.

[7] J. Redmon. [Online]. Available: https://pjreddie.com/darknet/yolo/

Authorized licensed use limited to: NATIONAL INSTITUTE OF TECHNOLOGY SURATHKAL. Downloaded on June 02,2025 at 07:02:52 UTC from IEEE Xplore. Restrictions apply.