Journal of Advanced Research in Applied Sciences and Engineering Technology

Journal homepage:
https://semarakilmu.com.my/journals/index.php/applied_sciences_eng_tech/index
ISSN: 2462-1943

Low Power RTL Implementation of Tiny-YOLO-v2 DCNN Hardware Accelerator on Virtex-7 FPGA

Sherry Heshmat Hareth Korisa[1,*], Khaled Ali Shehata[1], Hassan Mostafa[2]

[1] School of Electronics and Communication Engineering Arab academy for science and technology and maritime transport Cairo, Egypt
[2] Department of Electronics and Communications Engineering, Cairo University, Egypt

ABSTRACT

Deep Convolution Neural Networks (DCNNs) are widely used in real-time applications, including image classification, speech recognition, and object detection. However, there are challenges for real-time applications on portable devices like mobile phones or embedded systems. Most object detection models are optimized for desktop configurations, requiring fast GPUs. Shallower networks with fewer computational complexities have been proposed for real-time detection, but ultimately compromise detection accuracy. Performance and complexity trade-offs are significant for computationally complex deep networks. This work aims to implement the CNN-based object detection model Tiny-YOLO-v2 on a Field Programmable Gate Array (FPGA) using Register Transfer Logic (RTL) as a native language. Hardware implementation is synthesized on The AMD Virtex 7 FPGA VC709 Connectivity Kit using VHDL code on Vivado 2020.1. This is the first, up to the authors' knowledge, RTL implementation of the Tiny-YOLO-v2 object identification algorithm on FPGA. The power consumed by the CNN layers equals 7.09W at a frequency of 100MHz.

Keywords:
Convolution Neural Networks (CNN); Field Programmable Gate Array (FPGA); Tiny-YOLO-v2; register transfer logic; power consumption; speed

1. Introduction

CNNs are the state of art driven from the Artificial Neural Networks [1, 2] under the Deep Neural Network (DNN) [3] umbrella which produce intelligent system that can automatically adapt to new situations. CNNs have many applications such as object detection [4, 5], recognizing facial [6, 7], autonomous cars [8-10], and robotics [11-13]. Furthermore, the later usually take many operations to process. Those operations introduce the complexity of implementing the CNN algorithms, as the hardware usually has limited bandwidth and on-chip memory. New Field Programmable Gate Array (FPGA) is adapted by CNNs algorithms [14] due to their relatively high performance and flexibility. Introducing Tiny-Yolo-v2, a well-known CNN object detection algorithm with a fixed point that provides the benefits of low power consumption that can be used with the object detection datasets from Pascal VOC [15] and COCO [16].

* *Corresponding author.*
E-mail address: sherry_heshmat@yahoo.com

https://doi.org/10.37934/araset.54.2.287300

1.1 Introduction to CNN and YOLO Network

CNN has a structure that is inspired by the idea of how the human visual cortex works. Each neuron receives an input from a bunch of neurons from the previous layers, and it utilizes convolution to calculate the input so that the positioning information is within the feature map. Through this process, the number of parameters needed to learn the connection by looking at every single pair of input and output is greatly reduced.

YOLO is one of the most popular CNNs object detection methods, which appeared in research papers in 2016 and 2017 [17]. It works more efficiently in terms of the number of operations compared to other object detection methods [18, 19]. The YOLO network predicts a set of rectangular bounding boxes and assigns a class label [20, 21]. Performing these two tasks in the same network, a single forward pass is able to detect a large number of classes. On the other hand, other object detection methods require the network to make hundreds of predictions and conduct post-processing at a later stage in order to detect each class in an image. Moreover, YOLO network shares its resources among the classes, allowing it to make only a few predictions per image. The architecture is trained with image input and finding objects in the Image Large Scale Visual Recognition Challenge (ILSVRC). YOLO model has undergone continuous evolution [22, 23], resulting in the development of newer and refined models such as the YOLO, YOLO-v2, YOLO-v3, Tiny YOLO-v2, and YOLO-v4 successively.

2. Tiny YOLO-v2 Background

In this paper the hardware Tiny-YOLO-v2 network is implemented as the main principle, behind its development is trade off the detection performance for higher throughput, reducing the network complexity. As a result, the later takes up much less space in the form of model size while maintaining an acceptable trade-off with respect to the network latency due to its significantly smaller network size. Tiny-YOLO-v2 network [22] is a simplified network shown in Figure 1 compared to the full YOLO-v2 network in Figure 2. Compared to YOLO-v2, Tiny-YOLO-v2 network uses half the number of layers and, more importantly, requires only 3x3 and 1x1 convolutions as shown in Table 1.

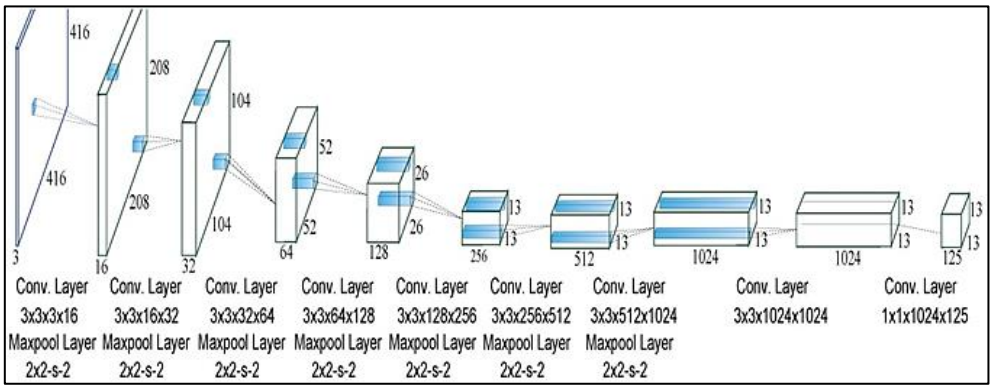

Fig. 1. Tiny-YOLO-v2 architecture with CNN layers [22]

This layer reduction was necessary in order for Tiny-YOLO to fit on embedded-class GPUs, FPGAs, and DSPs. Tiny-YOLO-v2 network has 9 convolution layers, and 6 maximum pooling layers presented in Table 1. Here the convolution layers are followed by leaky Rectifier Linear Unit (ReLU) as an activation function and batch normalization operation. Input image size to the network is (416 × 416) to 20 output classes as the main YOLO but with faster training and detection.

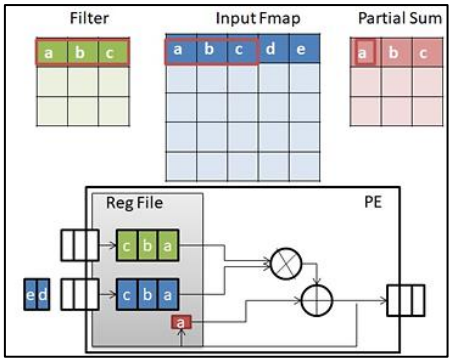

Fig. 2. Parallel engine unit [24]

Table 1
Tiny-YOLO-v2 network CNN layers

Number of layers	Number of filters	Filter size	Stride	Input matrix size	Input depth	Output depth
1	16	3x3	1	416x416	3	16
2	32	3x3	1	208x208	16	32
3	64	3x3	1	104x104	32	64
4	128	3x3	1	52x52	64	128
5	256	3x3	1	26x26	128	256
6	512	3x3	1	13x13	256	512
7	1024	3x3	1	13x13	512	1024
8	1024	3x3	1	13x13	1024	1024
9	125	1x1	1	13x13	1024	125

3. Hardware Low Power Tiny-YOLO-v2 Network RTL Implementation

3.1 CNNs Main Unit Blocks

Firstly, the main CNN block that represents the arithmetic and physical operations is Multiplication and Accumulation (MAC), which is presented by the Parallel Engine (PE) unit in Figure 3. The MAC process is carried out by multiplying the input data of the image input Feature maps (Fmaps) element by element with the training weights, also known as kernels filters. The accumulation operation is carried out, as shown in Figure 2 with strides defined inside each convolution layer; the output of the convolution layers is fed into output Fmaps after the different kernel filters are applied.

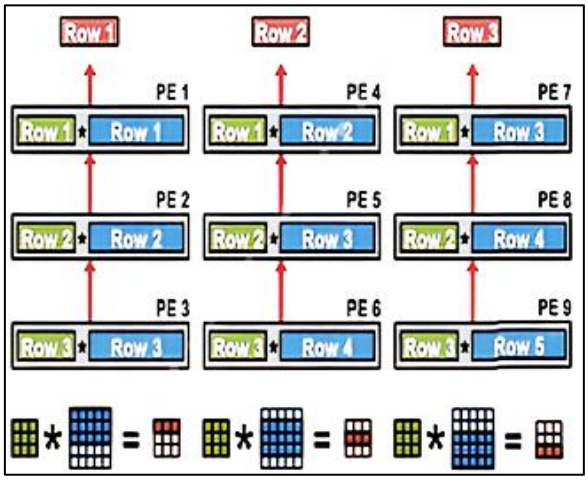

Fig. 3. Row stationary dataflow for 2D CNN reuse [24]

Secondly, after every convolution layer, Rectified Linear Unit (ReLU) layers are often positioned. They serve as an introduction to a non-linear process. The network adapts to each given collection of data thanks to its non-linear operation. ReLU returns all positive values after clamping all negative values to 0. Thirdly, ReLU levels come before Local Response Normalization (LRN) layers. They are employed to improve system accuracy and expedite the training process by normalizing the neuronal response throughout the depth of the same geographical region. Fourthly, each Fmap dimension is decreased by using Maximum Pooling (MaxPooling) layers, which also provide the most crucial feature information.

3.2 Tiny-YOLO-v2 Network RTL Implementation

The layers of the inference phase implementation of the Tiny-YOLO v2 network architecture are given in this section. Row stationary dataflow technique handles PE, which functions as a neuron in our architecture, is the most important unit. The PE unit is seen in Figure 3; it is made up of two Registers Files (RFs) for storing the input and the current filter, and a MAC operation for computing partial sums that only use one memory space. The input can then be reused in the RF again since there are overlaps in the input activations Fmaps between several sliding windows. Multiple PEs can be used to finish the 2-D convolution after each PE processes a 1-D convolution, as seen in Figure 3. Performing a general examination on Table 1, we considered that matrix size (13×13) of the input matrix to be the heart of our system as shown in Figure 4. The reason for this is that convolution layers (6, 7, 8, and 9) repeat it, correspondingly. Furthermore, the other convolution layers input size of matrix is considered to double the size.

13

13

0	1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38
39	40	41	42	43	44	45	46	47	48	49	50	51
52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100	101	102	103
104	105	106	107	108	109	110	111	112	113	114	115	116
117	118	119	120	121	122	123	124	125	126	127	128	129
130	131	132	133	134	135	136	137	138	139	140	141	142
143	144	145	146	147	148	149	150	151	152	153	154	155
156	157	158	159	160	161	162	163	164	165	166	167	168

Fig. 4. Main convolution input matrix size (13 x 13)

3.2.1 Convolution layer (Conv-5)

In Conv-5 the size of the input matrix is (26 × 26) which is double the main convolution input matrix size in Figure 5 with depth 128. As a result of that, implementation of one block of the main convolution size of matrix is done but it will be reused as hardware four times represented by (Division_1, Division_2, Division_3, and Division_4), as shown in Figure 5. In Figure 5, the green shaded bordered line is zero padding technique. Given filter matrix equals (3 × 3) with stride equals 1.

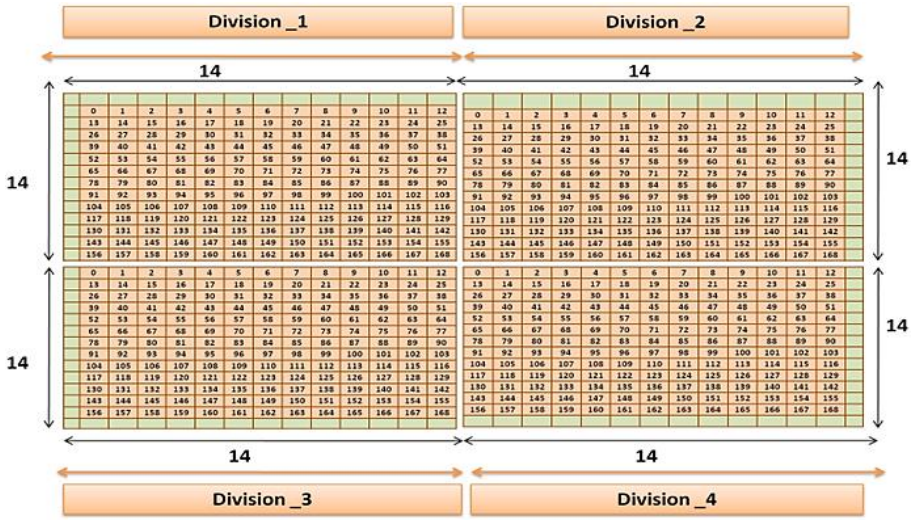

Fig. 5. Tiny-YOLO-v2 Conv-5 input matrix representation

Block diagram in Figure 6 shows Conv-5 implementation. At first reading from 4 input data MEMs and 4 filter MEMs simultaneously. Finishing the whole 128 depths of one division will take 32 internal MEMs to store the output of convolution process, secondly add the data of the 32 MEMs and store it in one MEM called MEM-1 for Division-1. The third step repeats the previous two steps three more times to finish the whole four divisions with depths 128 for one filter. Accordingly, the output of one filter will be stored in 4 MEMs as shown in Figure 6. Finally repeating the previous three steps 256 times to finish the convolution process of the whole 256 filters. The output is 1024 MEMs size of the memory is constant (13 × 13) total as a result of 4 multiplied by 256 which will make the total memories used be equal to 1024. MaxPooling layer is used after it as illustrated in Figure 7 downsizing the memories used to be equal to 256.

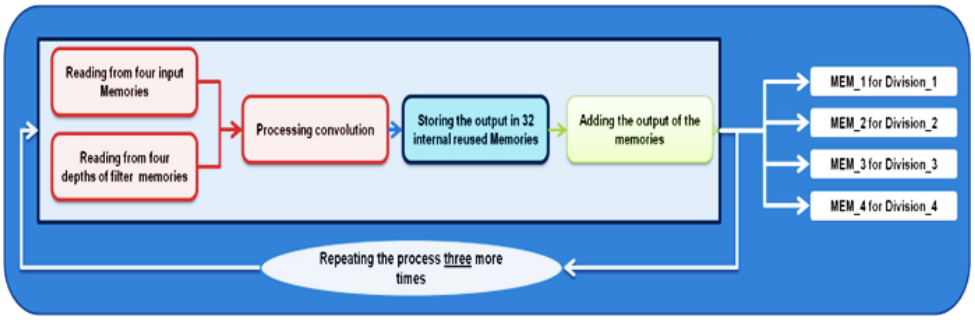

Fig. 6. Tiny-YOLO-v2 Conv-5 implementation blocks for one filter

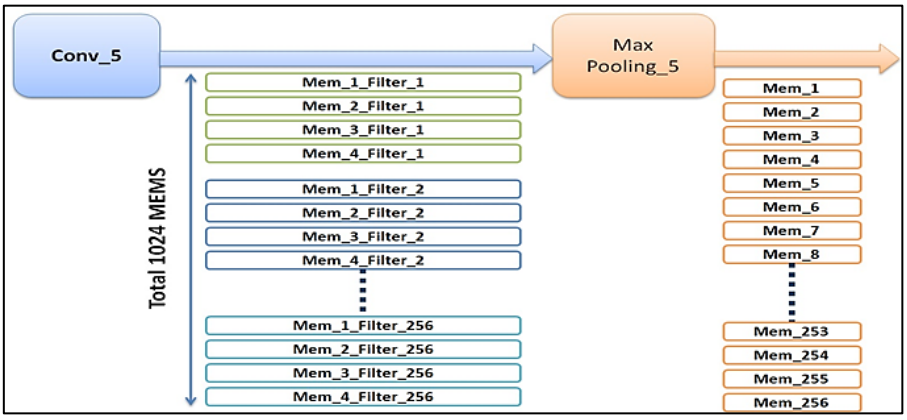

Fig. 7. Memory block diagram used in Tiny-YOLO-v2 Conv-5

3.2.2 Convolution layer (Conv-4)

The input matrix to Conv-4 is (52 × 52) which is the double of that of Conv-5. Therefore, by using the main size matrix (13x 13) number of Divisions will be equal to 16 to finish one filter as shown in Figure 8. As a result, each filter will store its output in 16 memories and here Conv-4 has 128 filters therefore the overall number of memories used are the result of multiplying 16 with 128 equals 2048. Figure 9 shows the total memories used after MaxPooling which will be equal to 512.

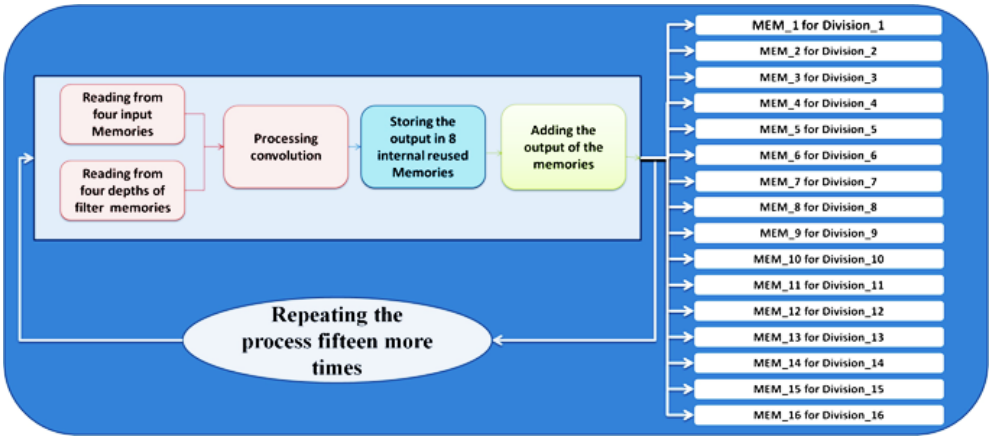

Fig. 8. Tiny-YOLO-v2 Conv-4 implementation blocks for one filter

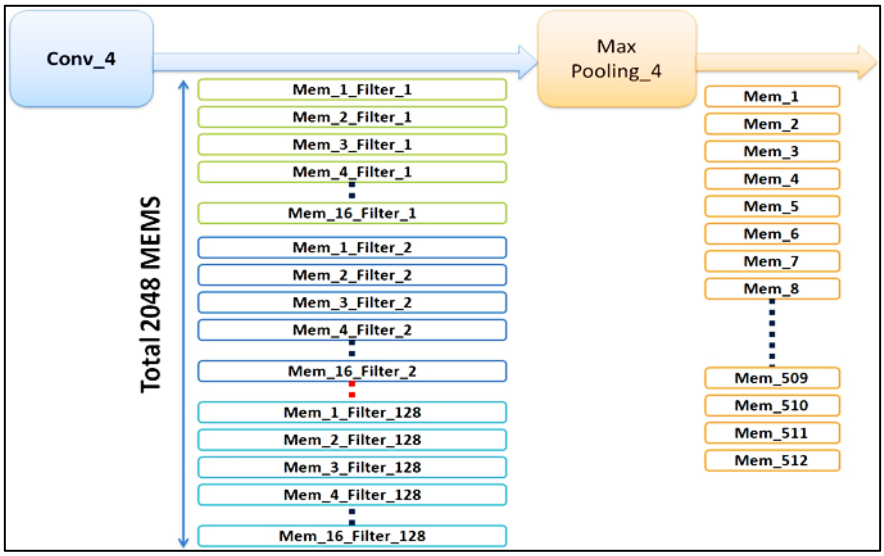

Fig. 9. Memory block diagram used in Tiny-YOLO-v2 Conv-4

3.2.3 Convolution layer (Conv-3)

Conv-3 has an input matrix size of (104 x 104), which is twice as large as that in Conv-4s. The resultant memories taken by one filter to store its output is 64 corresponds to the number of divisions needed as shown in Figure 10. The red shaded corresponds to the division need for Conv-5 to finish one filter that has the main size (13 x 13). Also, the green includes the red part corresponds to the division needed for Conv-4 to finish one filter. Finally, the whole figure represents the total number of divisions which is translated to memories needed to finish one filter. Therefore, Figure 11 explains the data flow operation for one filter. Numbers of filters in Conv-3 is equal to 64. In conclusion, the

total number of memories is 64 multiplied by 64 equals 4096 memories. The MaxPooling block, as seen in Figure 12, reduced the amount of memory utilized in Conv-3 to 1024.

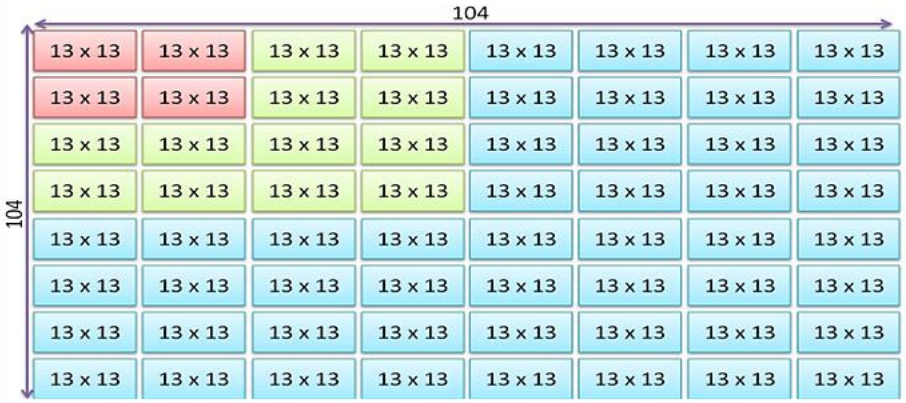

Fig. 10. Number of divisions used in Conv-3 to finish the output of one filter

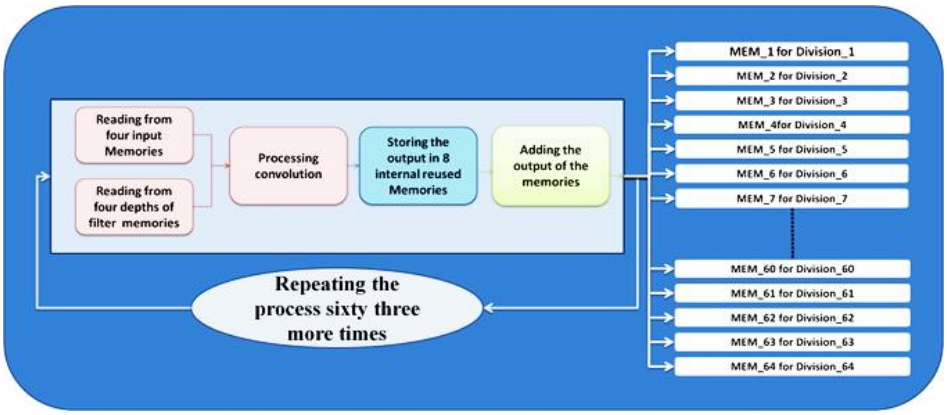

Fig. 11. Tiny-YOLO-v2 Conv-3 implementation blocks for one filter

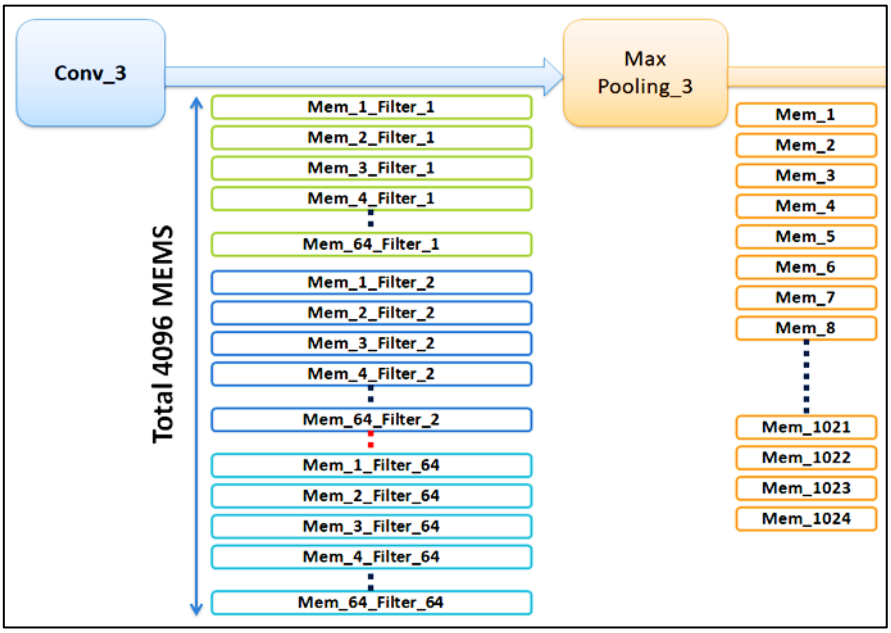

Fig. 12. Memory block diagram used in Tiny-YOLO-v2 Conv-3

3.2.4 Convolution layer (Conv-2)

The input matrix size of Conv-2 is (208 x 208) which is double the size of that in Conv-4. Number of divisions used to finish one filter equals 256 which means reusing the main constant block size (13 x 13) equals 256 times. Figure 13 represents the data flow block diagram to get the output for one filter. The number of memories used to store the output of one filter is equal to 256. For using 32 filters in Conv-2 therefore the total number of memories used will be equal to 32 multiplied by 256 which are 8192 memories as shown in Figure 14. After MaxPooling will minimise number of memories to 2048.

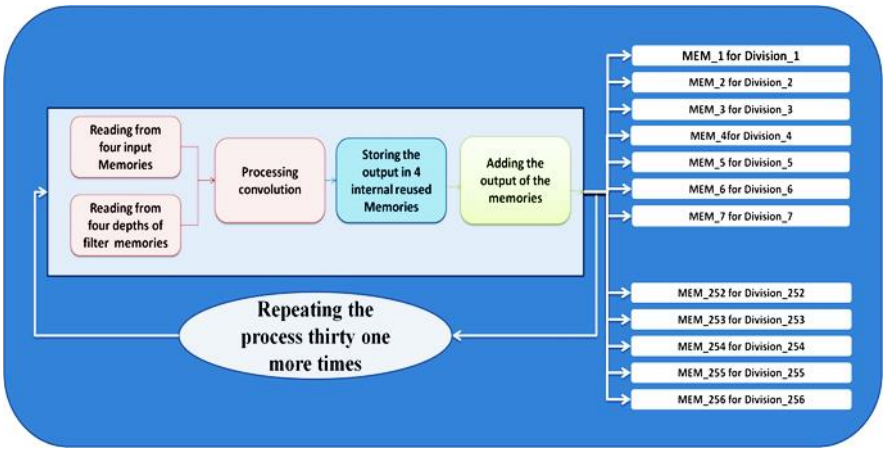

Fig. 13. Tiny-YOLO-v2 Conv-2 implementation blocks for one filter

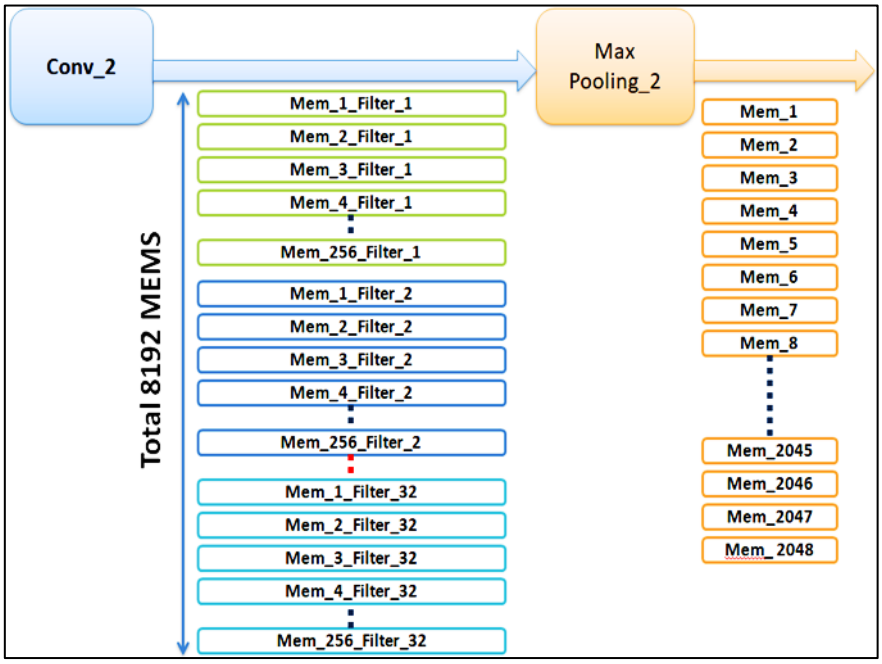

Fig. 14. Memory block diagram used in Tiny-YOLO-v2 Conv-2

3.2.5 Convolution layer (Conv-1)

The input matrix to Conv-1 is the main input picture feature data with matrix size (416 x 416) and 3 depths. Figure 15 illustrates the block diagram of data flow of Conv-1. To get the output of one filter number of memories needed is equal to 1024. As a result, the total output memories are the

product of multiplying 16 with 1024 to be equal to 16384 memories as shown in Figure 16. After passing the output into MaxPooling layer, the number of memories downsized to be 4096.

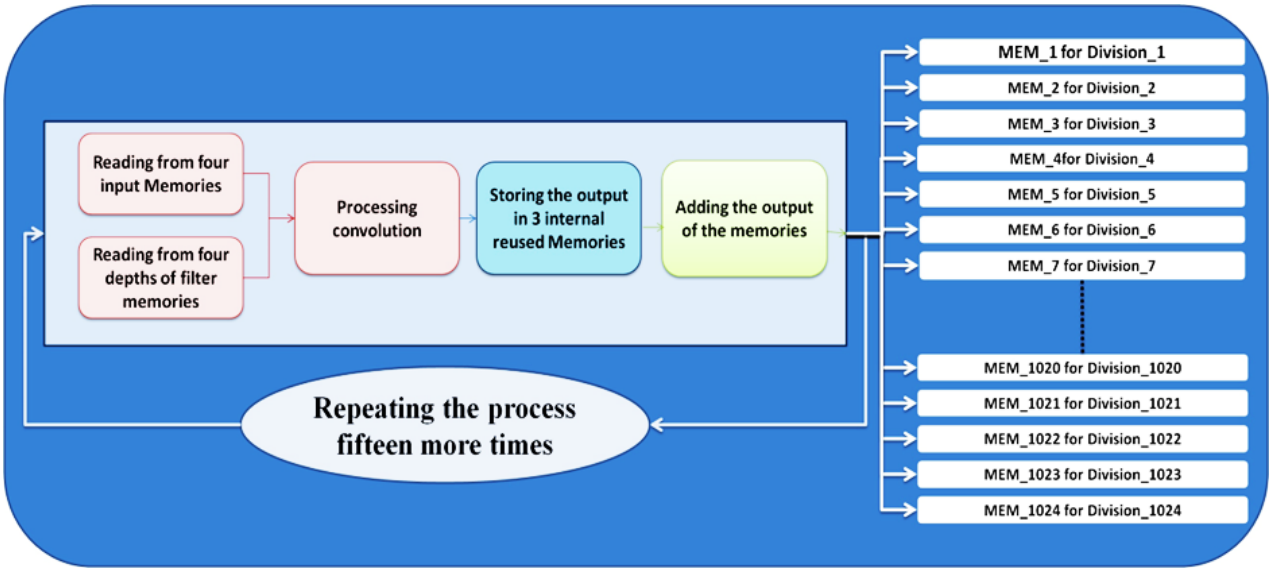

Fig. 15. Tiny-YOLO-v2 Conv-1 implementation blocks for one filter

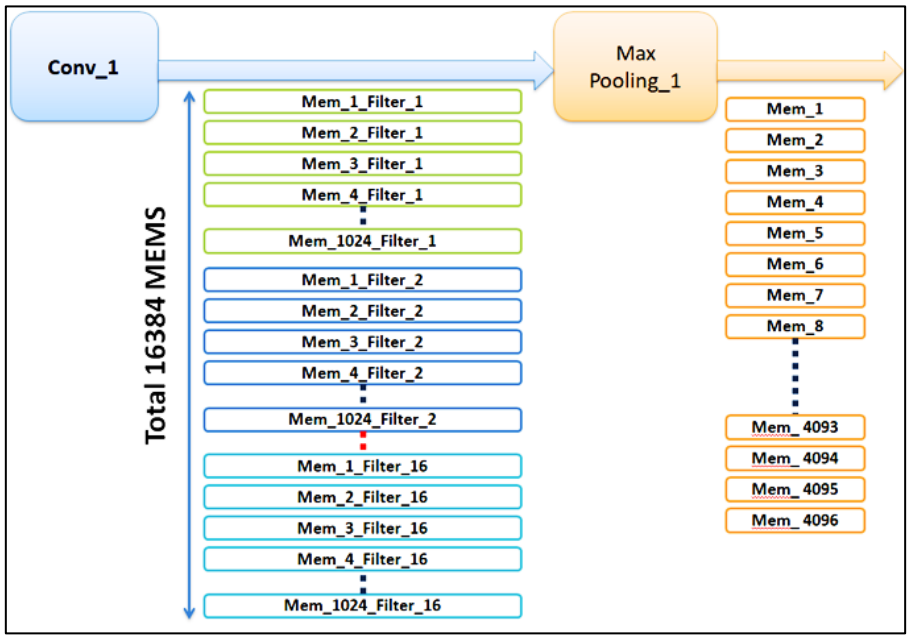

Fig. 16. Memory block diagram used in Tiny-YOLO-v2 Conv-2

3.2.6 Convolution layers (Conv-6, Conv-7, and Conv-8)

Main convolution input matrix with size (13 x 13) is constant through the three layers as presented in Table 1. Also, they are common in the filter size which is (3 x 3) and no MaxPooling block after them. On the other hand, the difference is in the number of filters and their depth too. Figure 17 shows data flow block diagram for Conv-6 to finish all its filters. Taking 64 internal memories to store the result after reading simultaneously from four input depths as illustrated in Figure 17. Conv-6 has 512 filters and 256 depths while it`s outputs are stored in 512 memories. In Conv-7 number of filters are 1024 with depth 512. Making 64 a constant number of internal memories. Therefore, as shown in Figure 18 they must be reused twice to finish the whole number of depths for one filter.

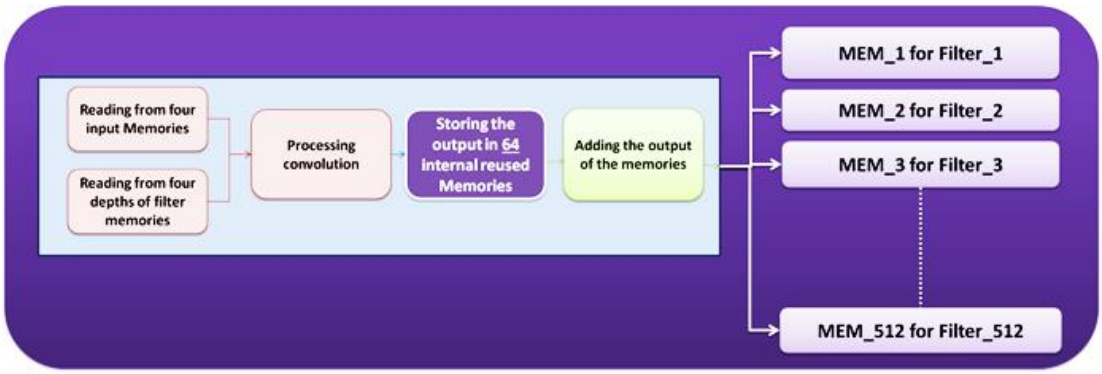

Fig. 17. Tiny-YOLO-v2 Conv-6 implementation blocks

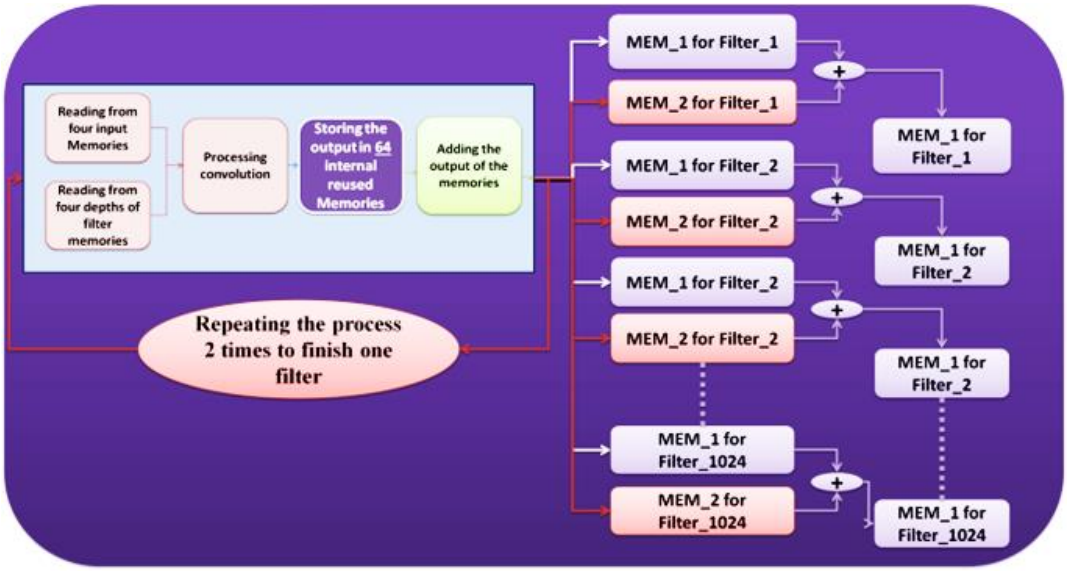

Fig. 18. Tiny-YOLO-v2 Conv-7 implementation blocks

Figure 19 illustrates the hardware blocks used for Conv-8. Conv-8 as shown in Table 1 has 1024 filters with 1024 depths for each. The procedure needs to be done four times in order to obtain the result of one filter, since the internal memory count will remain constant at 64.

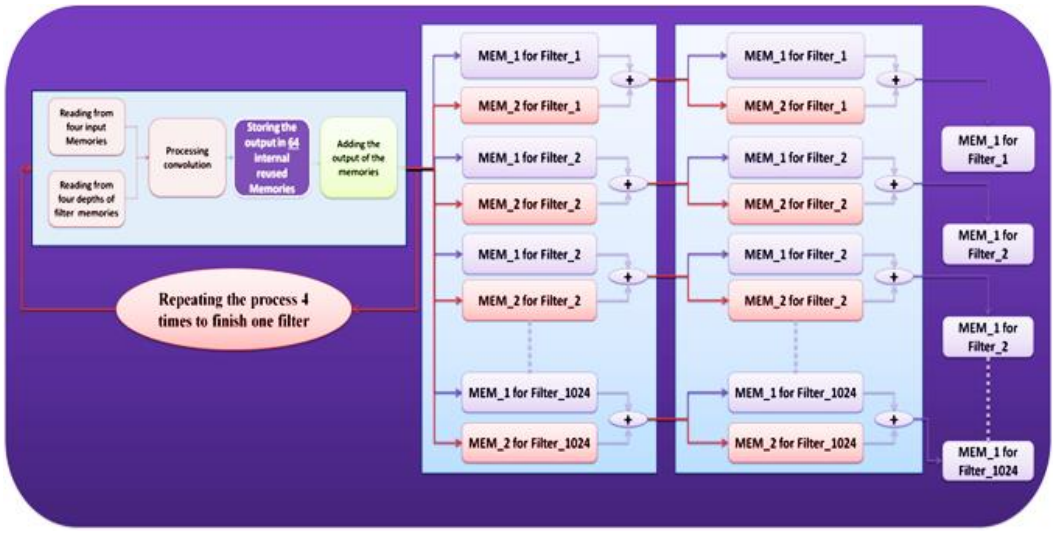

Fig. 19. Tiny-YOLO-v2 Conv-8 implementation blocks

3.2.7 Convolution layer (Conv-9)

Conv-9 has the same input matrix size which is (13 x 13) but it`s filter size is different. The size of the filter is equal to (1 x 1) instead of (3 x 3) and number of filters is 125 with depth equals 1024.

4. Memory Hierarchy

The system has three memory hierarchy layers which are DRAM (SD card), GLOBAL BUFFERS (filter, internal memories reused within the same layer, and memories used to store the output of each layer), and Inter-PE CACHES. Figure 20 illustrates all of these layers in order to reduce energy per access. All of the network's weights and biases are kept in DRAMs. Weights for local reuse and the input part are stored in the Inter-PE CACHES.

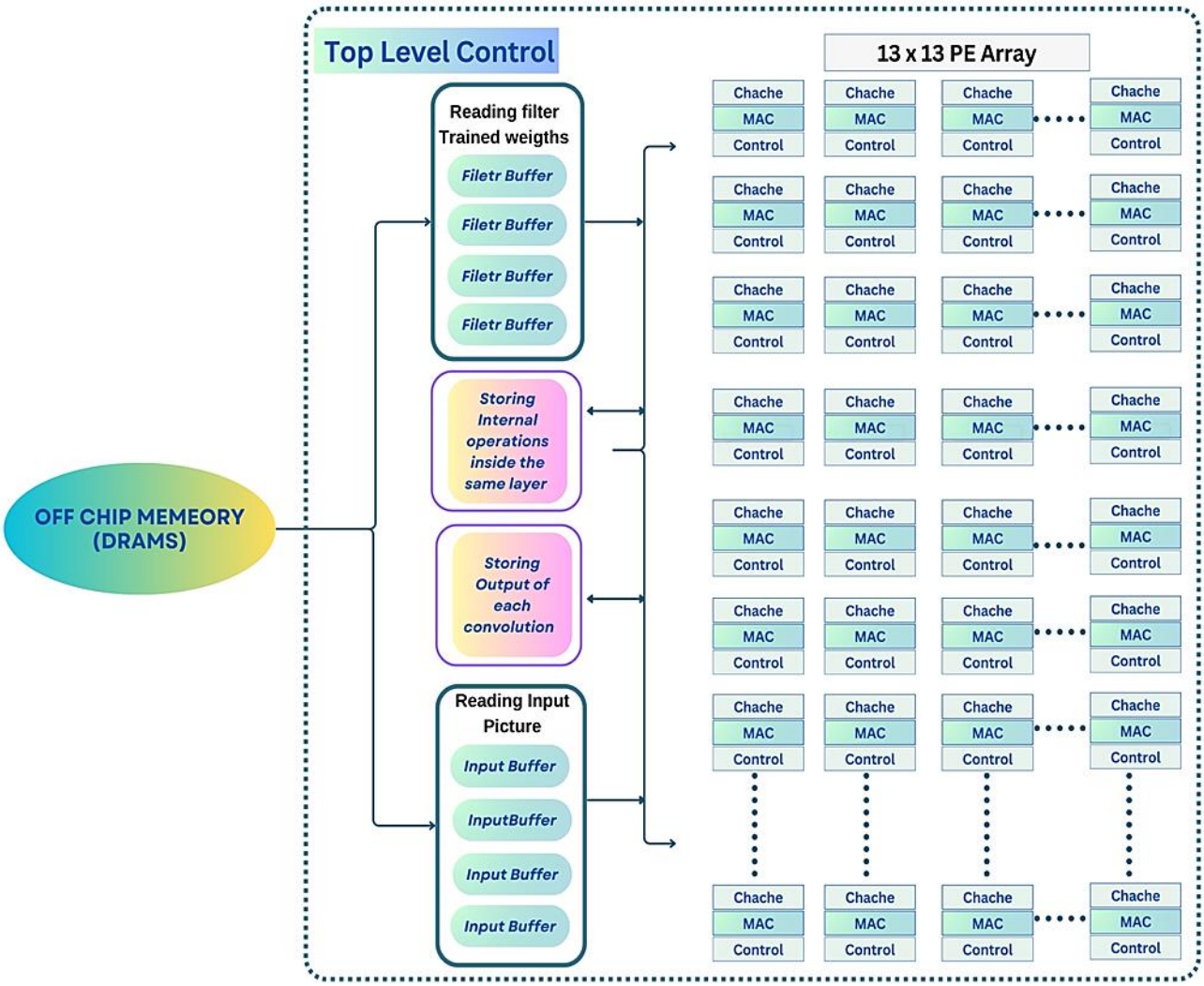

Fig. 20. Memory hierarchy

5. Experimental Results

The hardware implantation of each Convolution layer is done on AMD Virtex 7 FPGA VC709 Connectivity Kit. Table 2 shows the reports for hardware recourse utilization. The convolution layers consume 29% of the LUT, 60.5% of the available BRAM, 28% of the BUFG, 2.3%of the FF and 2.4% of

DSP. The power consumed by the convolution layers equals 7.09W at frequency 100MHz. The detailed utilization recourses of each convolution layer are presented in Table 3. There are more LUT and FF in Conv-1 than in Conv-2. Moreover, Conv-2's LUT and FF are more than Conv-3's, and the same sequence till the ninth convolution layer. The latter illustrates that the size of each CNN layer's input matrix, as previously indicated in Table 1, is what causes the LUT and FF consumption in each CNN layer. BRAM is forced as a constrain to be limited to 127. Furthermore, the last three convolution layers (CNN-7, CNN-8, and CNN-9) were when the DSP was first employed, indicating that the depth of each CNN affects the DSP consumption.

Table 2
Summary of hardware resources utilization

Resources	Available	Utilization
LUT	433200	29%
BRAM	1470	60.5%
BUFG	32	28.12%
FF	866400	2.3%

Table 3
Detailed hardware resources utilization for each convolution layer

Resources	Conv-1	Conv-2	Conv-3	Conv-4	Conv-5	Conv-6	Conv-7	Conv-8	Conv-9
LUT	45841	21116	7319	29869	23905	271	4	0	158
BRAM	127	127	127	127	127	127	0	127	0
BUFG	1	1	1	1	1	1	1	1	1
FF	7068	6668	5568	332	84	0	16	0	85
DSP	0	0	0	0	0	0	28	28	28

The comparison between the suggested design and earlier research on the hardware implementation of the Tiny-YOLO-v2 is presented in Table 4. When it comes to reducing power usage, the proposed design using the Tiny-YOLO-v2 performs noticeably better than the Sim-YOLO-v2 that was provided in [25-27].

Table 4
Detailed comparison of the YOLO-v2 hardware design proposal with previous work

Resources	Sim-YOLO-v2 on GPU [25]	Lightweight YOLO-v2 [26]	Sim-YOLO-v2 [27]	OpenCL Tiny-YOLO-v2 [28]	This work RTL Tiny-YOLO-v2
Platform	GTX Titan X	Zynq Ultrascale+	Virtex-7 VC707	Cyclone V PCIe	AMD Virtex 7 FPGA VC709
Frequency	1GHz	300MHz	200MHz	117MHz	100MHz
BRAM	N/A	1706	1144	N/A	1470
DSPs	N/A	377	272	122	84
FF	N/A	370000	115000	N/A	866400
LUT	N/A	135000	155000	113000	433200
Power (W)	170	N/A	18.29	N/A	7.09
Throughput (GOPs)	1512	610.9	1877	21.6	1.6

4. Conclusions

In conclusion, the paper represents native RTL hardware implementation of Tiny-Yolo-v2 object detection network. The implementation is synthesized on AMD Virtex 7 FPGA VC709 Connectivity Kit operating on a frequency of 100 MHz. Moreover, memory hierarchy are used in the introduced

architecture design to dramatically reduce the power consumption to 7.09W compared to hundreds of Watts in the previous work. Ultimately, the methods employed to get the latter satisfactory result with 7.09W of power included employing native RTL for implementations, pipelining in each CNN layer to obtain the overall result, and reusing some blocks in the Hardware rather than reimplementing them.

Acknowledgement
The Electronics and Communication Engineering department of the Arab Academy for Science and Technology, Maritime Transport, (AASTMT) provided support for this research. I also appreciate that OneLab (https://onelab-eg.com/) and Hammam Lab at Cairo University allowed me to work there and utilize their GPU.

References

[1] Wang, Meng, Weijie Fu, Xiangnan He, Shijie Hao, and Xindong Wu. "A survey on large-scale machine learning." *IEEE Transactions on Knowledge and Data Engineering* 34, no. 6 (2020): 2574-2594. https://doi.org/10.1109/TKDE.2020.3015777

[2] Litjens, Geert, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I. Sánchez. "A survey on deep learning in medical image analysis." *Medical image analysis* 42 (2017): 60-88. https://doi.org/10.1016/j.media.2017.07.005

[3] Zamri, Nurul Farhana Mohamad, Nooritawati Md Tahir, Megat Syahirul Megat Ali, Nur Dalila Khirul Ashar, and Ali Abd Almisreb. "Real time snatch theft detection using deep learning networks." *Journal of Advanced Research in Applied Sciences and Engineering Technology* 31, no. 1 (2023): 79-89. https://doi.org/10.37934/araset.31.1.7989

[4] Amjoud, Ayoub Benali, and Mustapha Amrouch. "Object detection using deep learning, CNNs and vision transformers: A review." *IEEE Access* 11 (2023): 35479-35516. https://doi.org/10.1109/ACCESS.2023.3266093

[5] Yadav, Satya Prakash, Muskan Jindal, Preeti Rani, Victor Hugo C. de Albuquerque, Caio dos Santos Nascimento, and Manoj Kumar. "An improved deep learning-based optimal object detection system from images." *Multimedia Tools and Applications* 83, no. 10 (2024): 30045-30072. https://doi.org/10.1007/s11042-023-16736-5

[6] Munanday, Anbananthan Pillai, Norazlianie Sazali, Arjun Asogan, Devarajan Ramasamy, and Ahmad Shahir Jamaludin. "The implementation of transfer learning by convolution neural network (CNN) for recognizing facial emotions." *Journal of Advanced Research in Applied Sciences and Engineering Technology* 32, no. 2 (2023): 255-276. https://doi.org/10.37934/araset.32.2.255276

[7] Munanday, Anbananthan Pillai, Norazlianie Sazali, Wan Sharuzi Wan Harun, Kumaran Kadirgama, and Ahmad Shahir Jamaludin. "Analysis of convolutional neural networks for facial expression recognition on GPU, TPU and CPU." *Journal of Advanced Research in Applied Sciences and Engineering Technology* 31, no. 3 (2023): 50-67. https://doi.org/10.37934/araset.31.3.5067

[8] Geiger, Andreas, Philip Lenz, and Raquel Urtasun. "Are we ready for autonomous driving? the kitti vision benchmark suite." In *2012 IEEE conference on computer vision and pattern recognition*, pp. 3354-3361. IEEE, 2012. https://doi.org/10.1109/CVPR.2012.6248074

[9] Tampuu, Ardi, Tambet Matiisen, Maksym Semikin, Dmytro Fishman, and Naveed Muhammad. "A survey of end-to-end driving: Architectures and training methods." *IEEE Transactions on Neural Networks and Learning Systems* 33, no. 4 (2020): 1364-1384. https://doi.org/10.1109/TNNLS.2020.3043505

[10] Rabecka, V. Darthy, and J. Britto Pari. "Assessing the performance of advanced object detection techniques for autonomous cars." In *2023 International Conference on Networking and Communications (ICNWC)*, p. 1-7. IEEE, 2023. https://doi.org/10.1109/ICNWC57852.2023.10127360

[11] Sharma, Archit, Ahmed M. Ahmed, Rehaan Ahmad, and Chelsea Finn. "Self-improving robots: End-to-end autonomous visuomotor reinforcement learning." *arXiv preprint arXiv:2303.01488* (2023). https://doi.org/10.48550/arXiv.2303.01488

[12] Aradi, Szilárd. "Survey of deep reinforcement learning for motion planning of autonomous vehicles." *IEEE Transactions on Intelligent Transportation Systems* 23, no. 2 (2020): 740-759. https://doi.org/10.1109/TITS.2020.3024655

[13] Ibrahim, Amin S., Adel Refky, and Abdelghany M Abdelghany. "Self-driving car based CNN deep learning model." *International Journal of Advanced Engineering and Business Sciences* 4, no. 3 (2023). https://doi.org/10.21608/ijaebs.2023.208574.1081

[14] Ye, Haobo. "Accelerating convolutional neural networks: Exploring FPGA-based architectures and challenges." In *Journal of Physics: Conference Series*, 2786, no. 1, p. 012004. IOP Publishing, 2024. https://doi.org/10.1088/1742-6596/2786/1/012004

[15] Everingham, Mark, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. "The pascal visual object classes (voc) challenge." *International journal of computer vision* 88 (2010): 303-338. https://doi.org/10.1007/s11263-009-0275-4

[16] Lin, Tsung-Yi, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. "Microsoft coco: Common objects in context." In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pp. 740-755. Springer International Publishing, 2014. https://doi.org/10.1007/978-3-319-10602-1_48

[17] Redmon, J. "You only look once: Unified, real-time object detection." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, p. 779-788. 2016. https://doi.org/10.1109/CVPR.2016.91

[18] Sirisha, U., S. Phani Praveen, Parvathaneni Naga Srinivasu, Paolo Barsocchi, and Akash Kumar Bhoi. "Statistical analysis of design aspects of various YOLO-based deep learning models for object detection." *International Journal of Computational Intelligence Systems* 16, no. 1 (2023): 126. https://doi.org/10.1007/s44196-023-00302-w

[19] Agrawal, Prakhar, Garvi Jain, Saumya Shukla, Shivansh Gupta, Deepali Kothari, Rekha Jain, and Neeraj Malviya. "YOLO algorithm implementation for real time object detection and tracking." In *2022 IEEE Students Conference on Engineering and Systems (SCES)*, pp. 01-06. IEEE, 2022. https://doi.org/10.1109/SCES55490.2022.9887678

[20] Diwan, Tausif, G. Anirudh, and Jitendra V. Tembhurne. "Object detection using YOLO: Challenges, architectural successors, datasets and applications." *multimedia Tools and Applications* 82, no. 6 (2023): 9243-9275. https://doi.org/10.1007/s11042-022-13644-y

[21] Pan, Xingyu, Wenjun Yue, Tonglin Liao, Xinyi Tang, and Teoh Teik Toe. "The application and expansion of the YOLO algorithm in the field of campus cat and dog identification." In *2023 IEEE 11th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, vol. 11, p. 17-22. IEEE, 2023. https://doi.org/10.1109/ITAIC58329.2023.10409034

[22] HC, D. "An overview of you only look once: unified, real-time object detection." *International Journal for Research in Applied Science and Engineering Technology* 8, no. 6 (2020): 607-609. https://doi.org/10.22214/ijraset.2020.6098.

[23] Yao, ZhengBai, Will Douglas, Simon O'Keeffe, and Rudi Villing. "Faster yolo-lite: Faster object detection on robot and edge devices." In *Robot World Cup*, pp. 226-237. Cham: Springer International Publishing, 2021. https://doi.org/10.1007/978-3-030-98682-7_19

[24] Sze, Vivienne, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. "Efficient processing of deep neural networks: A tutorial and survey." *Proceedings of the IEEE* 105, no. 12 (2017): 2295-2329. https://doi.org/10.1109/JPROC.2017.2761740

[25] Redmon, Joseph, and Ali Farhadi. "YOLO9000: better, faster, stronger." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, p. 7263-7271. 2017. https://doi.org/10.1109/cvpr.2017.690.

[26] Nakahara, Hiroki, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. "A lightweight YOLOv2: A binarized CNN with a parallel support vector regression for an FPGA." In *Proceedings of the 2018 ACM/SIGDA International Symposium on field-programmable gate arrays*, p. 31-40. 2018. https://doi.org/10.1145/3174243.3174266

[27] Nguyen, Duy Thanh, Tuan Nghia Nguyen, Hyun Kim, and Hyuk-Jae Lee. "A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, no. 8 (2019): 1861-1873. https://doi.org/10.1109/TVLSI.2019.2905242

[28] Yap, June Wai, Zulkalnain bin Mohd Yussof, Sani Irwan bin Salim, and Kim Chuan Lim. "Fixed point implementation of tiny-yolo-v2 using opencl on fpga." *International Journal of Advanced Computer Science and Applications* 9, no. 10 (2018). https://doi.org/10.14569/IJACSA.2018.091062