Fixed Point Implementation of Tiny-Yolo-v2 using OpenCL on FPGA

Yap June Wai[1], Zulkalnain bin Mohd Yussof[2], Sani Irwan bin Salim[3], Lim Kim Chuan[4]
Center for Telecommunication Research and Innovation
Faculty of Electronic and Computer Engineering
Universiti Teknikal Malaysia Melaka
Melaka, Malaysia

***Abstract*—Deep Convolutional Neural Network (CNN) algorithm has recently gained popularity in many applications such as image classification, video analytic and object detection. Being compute-intensive and memory expensive, CNN-based algorithms are hard to be implemented on the embedded device. Although recent studies have explored the hardware implementation of CNN-based object classification models such as AlexNet and VGG, there is still a rare implementation of CNN-based object detection model on Field Programmable Gate Array (FPGA). Consequently, this study proposes the fixed-point (16-bit) implementation of CNN-based object detection model: Tiny-Yolo-v2 on Cyclone V PCIe Development Kit FPGA board using High-Level-Synthesis (HLS) tool: OpenCL. Considering FPGA resource constraints in term of computational resources, memory bandwidth, and on-chip memory, a data pre-processing approach is proposed to merge the batch normalization into convolution layer. To the best of our knowledge, this is the first implementation of Tiny-Yolo-v2 object detection algorithm on FPGA using Intel FPGA Software Development Kit (SDK) for OpenCL. Finally, the proposed implementation achieves a peak performance of 21 GOPs under 100 MHz working frequency.**

***Keywords*—FPGA; CNN; Tiny-Yolo-v2; OpenCL; detection**

I. Introduction

Convolutional Neural Network (CNN) is a well-known deep learning architecture inspired by the artificial neural network. It has been primarily employed in various applications including image classification [1] [2] and object detection [3] [4] [5]. Unlike the traditional machine learning algorithms, CNN algorithms are extremely computationally expensive and memory intensive. The state-of-the-art of CNN algorithms usually require millions of parameters and billions of operations to process a single image input. This is a great challenge to implement CNN algorithms on an embedded system due to severe hardware constraints such as computational resources, memory bandwidth, and on-chip memory. Hence, in recent year, Field Programmable Gate Array (FPGA) has become an attractive alternative solution to accelerate CNN-based algorithms due to its relatively high performance, flexibility, energy efficient and fast development cycle, especially with the new release of High-Level-Synthesis (HLS) tool: OpenCL. It greatly reduces the complexity of programming by enabling the auto-compilation from a high-level program (C/C++) to register-transfer-level (RTL).

Prior works [6] [7] have shown the effort of accelerating CNN classification model: AlexNet and VGG through the implementation of 3-Dimension (3D) convolution as General Matrix-Matrix Multiplication (GEMM). Data rearrangement on-the-fly technique is proposed to reduce the memory footprint. In this work, the idea of mapping 3D convolution as GEMM and data rearrangement on-fly are borrowed and these techniques are further applied to perform object detection algorithm: Tiny-Yolo-v2 on both Pascal VOC [8] and COCO [9] object detection datasets. Prior work [10] takes a different approach to accelerate the CNN classification in a deeply pipelined manner. In addition, they proposed the insight of "performance density" as an alternative performance evaluation metric for the fair comparison between their work and prior research work. However, their design implemented floating-point arithmetic which it is unfriendly to the hardware computation. Hence, in this work, the fixed-point arithmetic instead of floating-point arithmetic is implemented to better improve the bandwidth and resources utilization. In addition, a technique to merge the batch normalization into convolution is proposed to reduce the data redundancy. The key contributions are summarized as follows:

- A CNN-based object detection algorithm: Tiny-Yolo-v2 with 16-bit fixed-point arithmetic running on FPGA
- A systematic in-depth analysis on the impact of the precision of the weights on the two detection datasets: Pascal VOC 2007 and COCO.
- A novel approach of merging batch normalization layers and convolutional layer to reduce data redundancy during the inference process.

The rest of the paper is presented as follows. Section 2 briefly describes the background of the research work. In this section, the overview of OpenCL development flow, the architecture of Tiny-Yolo-v2 and performance evaluation metrics are presented in detail. Section 3 briefly discuss the proposed design and the case studies on the impact of precision of the weights for Tiny-Yolo-v2 on the two detection datasets: VOC [10] and COCO [11]. It also studies the mathematical approach to merge the batch normalization operation into the convolutional layer. Section 4 briefly presents the experimental results. Section 5 concludes the paper.

II. Background

In this section, a detail description of the overview of OpenCL based FPGA development flow, the architecture of

Tiny-Yolo-v2 and the performance evaluation metrics used in this work is presented.

A. Overview of OpenCL

Intel FPGA Software Development Kit (SDK) for OpenCL [12] [13] allows the user to avoid the traditional hardware FPGA development flow by using HLS tools. It is an alternative approach to traditional RTL design concepts such as Verilog or VHDL with C or C++ synthesis. Fig. 1 illustrates the OpenCL-based FPGA accelerator development flow. In the OpenCL framework, the Central Processing Unit (CPU) acts as the host and it has bridges interconnect the Cyclone V PCIe FPGA board which it serves as an OpenCL device, forming a heterogeneous computing system. An OpenCL code is translated into hardware image, supported by OpenCL runtime driver. Furthermore, on the host side, C/C++ code runs on the CPU, providing vendor specific Application Programming Interface (API) to communicate with the implemented kernels on the Cyclone V PCIe FPGA board.

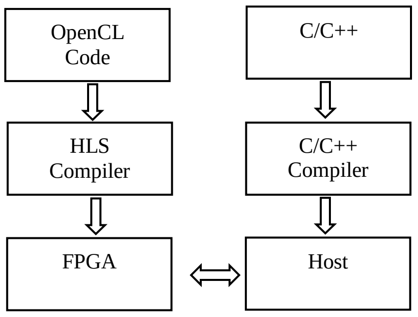

Fig. 1. OpenCL based FPGA Development Flow.

B. Architecture of Tiny-Yolo-v2

In this section, a detail exploration of the Yolo object detection framework is briefly discussed. Unlike prior object detection algorithm [6] which repurpose classifiers to perform detection, Yolo [7][8] uses a different approach to apply a single convolutional network to the full image and predict multiple bounding boxes and class probability for those boxes. Fig. 2 shows the architecture of Tiny-Yolo-v2, which consists of 9 convolutional layers, each with a leaky rectified linear unit (ReLU) based activation function and batch normalization operation interspersed with 6 max-pooling layers and a region layer. Tiny-Yolo-v2 takes input image size 416 x 416 to 20 output classes in VOC datasets whereas 80 output classes in COCO datasets.

C. Convolutional Layer

Tiny-Yolo-v2 employs feedforward process for object detection. A previous study [14] has proved that the convolutional layer will occupy over 90 % of the feed-forward computation period. Hence, in this work, the optimization of the convolutional layer will be the main focus to improve the performance of accelerator. Convolution layers involve billions of multiplication and addition operations between the filters and local regions of input for a single input image. The operations can be represented in (1) as followed:

$$X^{(i)} = \sum_{i=1}^{n} (x^{(n)} \times w^{(k)}) + b \tag{1}$$

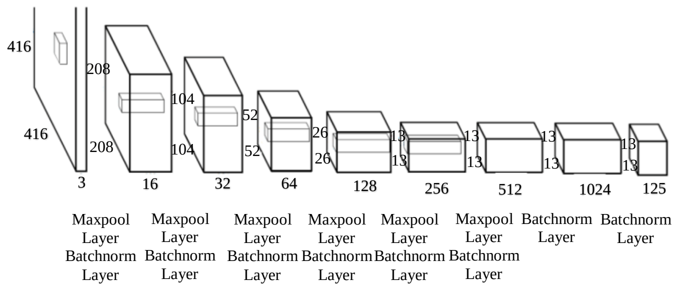
416
416
3
208
208
16
104
104
32
52
52
64
26
26
128
13
13
256
13
13
512
13
13
1024
13
13
125
Maxpool Layer Batchnorm Layer
Maxpool Layer Batchnorm Layer
Maxpool Layer Batchnorm Layer
Maxpool Layer Batchnorm Layer
Maxpool Layer Batchnorm Layer
Maxpool Layer Batchnorm Layer
Batchnorm Layer
Batchnorm Layer

Fig. 2. Tiny-Yolo-v2 Achitecture.

Where:

$X^{(i)}$ = Pixel of output feature0

$x^{(j)}$ = Pixel of input feature

$w^{(k)}$ = Convolution weights

b = Convolution bias

The total amount of operations in the convolution layer can be approximately calculated as shown in (2). Noted that, this equation ignores the number of operations for the batch normalization and leaky activation for each layer.

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

Where:

N_{in} = Number of channels of input feature

K = Filter size

N_{out} = Number of filters

H_{out} = Height of output feature

W_{ou0t} = Width of output feature

The memory requirement is described as space complexity. The main parameter in the Tiny-Yolo-v2 is the weight which is used in the convolutional layer. The number of weights in the convolutional layer can be expressed as (3):

$$\#Weights = N_{in} \times K \times K \times N_{out} \tag{3}$$

Where:

N_{in} = Number of channels of input feature

K = Filter size

N_{out} = Number of filters

Tiny-Yolo-v2 takes approximately 7 billion operations with 15 million weights just for one image input in Pascal VOC. On the other hand, Tiny-Yolo-v2 takes approximately 5.7 billion operations with 12 million weights just for one image input in COCO dataset.

D. Activation Function

Activation function in a CNN architecture is used to transform the input before the pooling layer. Sigmoidal activation functions were most often used in CNN. However, sigmoidal activations are bounded by a maximum and minimum value and thereby causing the saturated neuron in higher layers of the neural network. Alternatively, Leaky ReLU has recently been proposed as an activation function as it can cause the weight update which makes it never activate on any data point again. Leaky ReLU along with respective equation is shown in Fig. 3.

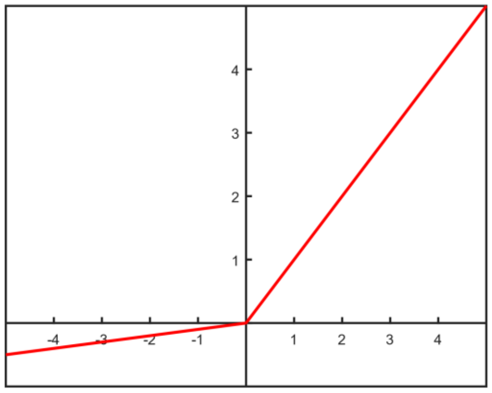

Fig. 3. Leaky Rectified Linear Unit(Leaky ReLU).

E. Pooling Layer

Pooling layer, in general, is a form of dimensional reduction used in CNN. Its goal is to throw away unnecessary information and only preserve the most critical information. Typical pooling functions are maximum pooling and average pooling layer. Max pooling returns the maximum value from the input, where average pooling returns the average value. The formula of max-pooling and average-pooling are illustrated as follows:

$$S[i, j] = \max\{S[i;, j'] : i \le i' < i+, j \le j' < j + p\} \tag{4}$$

$$S[i, j] = \text{average}\{S[i;, j'] : i \le i' < i+, j \le j' < j + p\} \tag{5}$$

F. Batch Normalization Layer

Batch normalization layer is implemented after the convolutional layer to provide any layer in Tiny-Yolo-v2 with inputs that are zero mean or unit variance. The equation of batch normalization is shown in (6). The normalization is performed to previous output of the convolutional layer by subtracting the batch mean and dividing by the batch variance. After batch normalization operation, the output will be shifted and scaled by the bias and scale. The value for these variables: means, bias, scale, and variance are generated in CNN training stage. These values allow each layer to learn in a more independent way and reduce the overfitting because it has a slight regularization effect.

$$X^{(j)} = \frac{(X^{(i)} - \mu)}{\sqrt{\sigma^2 + \xi}} \tag{6}$$

Where:

$X^{(j)}$ = Output Pixel after batch normalization

$X^{(i)}$ = Output Pixel after convolution

μ = Mean

σ^2 = Variance

ξ = Constant

G. Evaluation Metric

This section presents the detail description of the evaluation metric used to measure the performance of the proposed accelerator for Tiny-Yolo-v2 implemented on Cyclone V PCIe Development Kit FPGA board. Prior works [6] [7] [8] measure their accelerator design in term of accuracy and throughput. However, these two metrics are invalid in this case. This work is running object detection model instead of object classification. In contrast to classification task, object detection must localize and classify a variable number of objects on an image which indicates that the output of object detection may change from image to image. Hence, the accuracy of proposed accelerator is measured in term of mean average precision (mAP). Besides, to make a fair comparison on the throughput of proposed accelerator running on Cyclone V PCIe Development Kit to previous accelerator design that running on other FPGA such as Stratix V and Aria 10 GX, performance density is used as the main factor to evaluate the performance of proposed design. To make a fair comparison between the performances achieved in different hardware, the normalized performance of throughput is introduced in work [8]. The equation to calculate performance density is listed as follow:

$$\text{Perf.Density} = \frac{\text{Throughput}}{\text{DSP\_Consumed}} \tag{7}$$

III. Research Methodology

This section presents a detail description on how to implement of 3D convolution as GeMM in the accelerator. In addition, the approach of merging the batch normalization layer into the convolutional layer to reduce the data redundancy is described. Lastly, an in-depth analysis of the precision study of the weights of Tiny-Yolo-v2 on two different object detection datasets: Pascal VOC and COCO object detection datasets is provided.

A. Implementation of 3-Dimension (3D) Convolution as General Matrix-Matrix Multiplication (GeMM)

CNN employs a feedforward process for object detection, involving billions of multiplication and addition operations. Noted that the convolution operation essentially performs multiplication and accumulate operations between the filters and local region of inputs. To take advantage of this, the similar GeMM based convolution with data rearrangement on-fly is used to accelerate the algorithm. Fig. 4 shows that how the first layer of convolution layer of Tiny-Yolo-v2 is flattening and rearranged vertically into a 2-Dimension (2D) matrix through data rearrangement process. For example, the dimension of the input layer for Tiny-Yolo-v2 is 416 × 416 × 3 (Hin × Win × Nin) and the size of the kernel is 3 × 3 (K × K). The input image is flattened and rearranged into matrix B with a dimension of 416 × 416 × 3 × 3× 3. After that, a vector from matrix B is multiplied with a vector from matrix A. The result will be accumulated to be one output in matrix C.

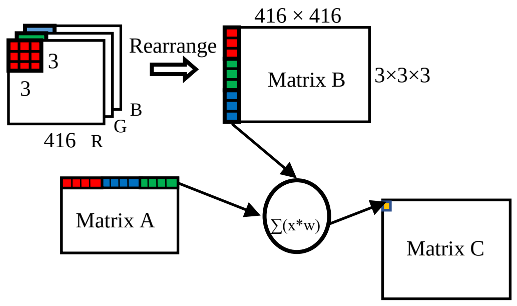

Fig. 4. Convert 3D Convolution into GeMM.

Notice that previous operation comes at the cost. It causes the expansion in memory size if the stride is smaller than the kernel size as pixels are overlapping and duplicated in the matrix. The expansion of memory increases the memory requirement to store the rearranged input feature matrix. To get rid of this, pseudo below is used to perform the similar operation on-the-fly by storing the corresponding pixels into FPGA's local memory before the matrix multiplication.

1) Get current work-item id (global_x, global_y, local_y, local_x, block_x, block_y)
2) Compute current output pixel (channel_out, height_out, width_out) based on current work-item id
3) Compute the actual input feature image (channel_in, height_in, width_in) based on computed output pixel coordinate.
4) Read the actual pixel value
5) Value = 0 < id ≤ input_dimension? Input[id] : 0

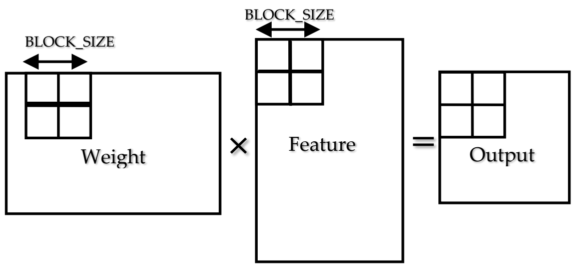

Fig. 5. Block Tiling of GeMM.

Fig. 5 illustrates that the way of how the tiled multiplication is implemented in the design. Instead of fetching the data and performing multiplication one value by one value, the performance is further improved by performing the multiplication in block. Two scalable design parameters BLOCK_SIZE and SIMD vectorization factor are introduced to determine the scale of the block. The BLOCK_SIZE determines how many data is fetched and perform multiplication at one time. In contrast, SIMD determines the factor by which data are vectorized and executed in Single Instruction Multiple Data (SIMD) manner. These parameters are scalable depends on the resources available in FPGA. The performance of the object detection is determined by choosing an appropriate of SIMD and BLOCK_SIZE factor.

B. Data Pre-Processing by Merging Batch Normalization into Convolutional Layer

Batch normalization is implemented after the convolution process in Tiny-Yolo-v2 to improve the stability of the neural network. The formula of convolution is shown in (1) and batch normalization is shown in (6). Taking advantage of the fact that the input of batch normalization operation is exactly the output of previous convolutional layer. Hence, the input, x(i) in (6) could be substituted with (1) and it will form the equation as shown in (8).

$$X^{(j)} = \left(\frac{\left[\sum_{n=1}^{n} (x^{(n)} \times \omega^{(\kappa)}) + b \right] - \mu}{\sqrt{\sigma^2 + \xi}} \right) \tag{8}$$

where:

$X(j)$ = batch normalization output

$x(n)$ = convolution input

$w(k)$ = convolution weights

b = convolution bias

μ = batch normalization mean

σ^2 = batch normalization variance

ξ = Constant

To further simplify (8), the complicated equation is further rearranged to become (9).

$$X^{(j)} = \left(\frac{\sum_{n=1}^{n} (x^{(n)} \times \omega^{(\kappa)})}{\sqrt{\sigma^2 + \xi}} \right) + \frac{(b - \mu)}{\sqrt{\sigma^2 + \xi}} \tag{9}$$

Now, it is worth noting that all the output of convolution must be divided by value = $\sqrt{\sigma^2 + \xi}$. By obeying the mathematic Distributive Law; that is, a(b+c) = ab+bc, the equation in (9) could be simplified to become (10).

TABLE II. PRECISION STUDY FOR TINY-YOLO-V2 WEIGHTS ON PASCAL VOC 2007 AND COCO DATASETS

Layer	Pascal VOC 2007				COCO			
	min	max	8-bit precision loss (%)	16-bit precision loss	min	max	8-bit precision loss	16-bit precision loss
1	-25.43982	26.42148	9.6%	0.0%	-18.71754	28.54190	6.3%	0.0%
2	-0.66461	0.47238	19.5%	0.2%	-0.65677	0.51022	17.8%	0.1%
3	-1.00267	1.31331	24.9%	0.1%	-0.96951	1.56696	26.1%	0.1%
4	-0.61135	0.79291	20.5%	0.1%	-0.64109	0.76376	19.1%	0.1%
5	-0.44413	0.73024	24.3%	0.1%	-0.41294	0384003	25.2%	0.1%
6	-0.51432	0.63433	29.9%	0.1%	-0.30638	0.63954	34.2%	0.1%
7	-1.39691	1.47183	15.7%	0.1%	-1.58541	1.85696	20.4%	0.1%
8	-0.02337	0.02339	3.6%	0.0%	-0.07274	0.05593	19.0%	0.1%
9	-0.23489	0.19417	6.1%	0.0%	-0.80548	0.54988	18.0%	0.1%

$$X^{(j)} = \sum_{n=1}^{n} \left(x^{(n)} \times \frac{\omega^{(\kappa)}}{\sqrt{\sigma^2+\xi}} \right) + \frac{(b-\mu)}{\sqrt{\sigma^2+\xi}} \qquad (10)$$

Finally, since the value of batch mean, batch standard deviation, weights and constant are pre-trained offline, and all these values will only be loaded one time during the inference process. Hence, to reduce the number of operations and data redundancy, the equation in (10) is transformed to become equation in (11), which is similar to (1) with fine-tuned weights and biases: b_{new} and w_{new}. This b_{new} and w_{new} can be pre-processed before the inference which helps to reduce the number of operations and improve the hardware utilization.

$$X^{(j)} = \sum_{n=1}^{n} \left(x^{(n)} \times \omega^{(\kappa)}_{new} \right) + b_{new} \qquad (11)$$

Where:

$$w_{new} = \frac{\omega^{(\kappa)}}{\sqrt{\sigma^2+\xi}}$$

$$b_{new} = \frac{b-\mu}{\sqrt{\sigma^2+\xi}}$$

C. *Precision Study for Object Detection Datasets*

Tiny-Yolo-v2 is trained using Graphic Processing Unit (GPU) in a 32-bit environment. Hence, the trained weights and bias are usually stored in 32-bit floating point format. However, such high precision is not necessarily in an inference machine. To reduce data redundancy, the best precision required for model Tiny-Yolo-v2 is explored using both COCO and Pascal VOC pre-trained weight from darknet framework. The analysis is done using the MATLAB Fixed-Point Designer Toolbox. Table I shows the range [min, max] of the fine-tuned weight w_{new} (as discussed in the previous section) for all 9 layers in Tiny-Yolo-v2 running on Pascal VOC and COCO object detection datasets. The table also explains the comprehensive precision loss of weights in: 8-bit and 16-bit fixed point representation. Based on the report generated by the toolbox, the 8-bit precision can contribute up to 29.9% of precision loss in Pascal VOC datasets and 34.2% precision loss in COCO datasets. According to this report, the performance of the accelerator is expected to be significantly degraded if the data is represented in 8-bit precision. Hence, 16-bit precision for the convolution weights and 32-bit precision is proposed for the intermediate inner product of weights and input in this work.

IV. RESULTS AND DISCUSSION

This section first briefly reports the hardware resource utilization. Then, the proposed design is compared to software implementation (CPU) with the two scalable design parameters BLOCK_SIZE=32 and SIMD=4. Finally, the comparison between the implementation and prior work. The resource utilization report on the Cyclone V PCIe FPGA board is shown in Table II. The proposed design with floating point is unable to fit into the board, as it consumes 161% of logic, 140% of RAM and 64% of DSP blocks which exceed the board hardware limitation. With the proposed data pre-processing technique presented in section 3.2, the design manages to reduce approximately 21% of logic usage, 8% of RAM usage and 6% of DSP usage. To further enhance the optimization, 16-bit of fixed-point arithmetic is implemented as discussed in section 3.3. The hardware resources are significantly reduced compared to floating point arithmetic. Finally, this design achieves 97% improvement in logic consumption, 30% improvement in RAM consumption and 18% in DSP consumption.

TABLE III. SUMMARY OF HARDWARE RESOURCE UTILIZATION

	Logic	RAM	DSP
Floating point	161 %	70 %	59 %
Floating point (with data pre-processing)	140 %	62 %	53%
Fixed-point (16-bit arithmetic with data pre-processing)	64%	40%	41%

TABLE IV. SUMMARY OF THE COMPARISON OF THE PERFORMANCE BETWEEN SOFTWARE IMPLEMENTATION AND FPGA IMPLEMENTATION

Layer	CPU (Intel R Core TM i7-7700)		FPGA	
	Pascal VOC	COCO	Pascal VOC	COCO
1	0.083	0.082	0.047	0.050
2	0.123	0.117	0.028	0.027
3	0.112	0.110	0.020	0.020
4	0.094	0.096	0.018	0.018
5	0.096	0.095	0.017	0.017
6	0.098	0.096	0.018	0.018
7	0.372	0.369	0.063	0.064
8	0.381	0.382	0.128	0.064
9	0.019	0.018	0.004	0.005
Total Execution Time (s)	1.378	1.365	0.339	0.278

The performance comparison between the proposed accelerator for Tiny-Yolo-v2 on VOC dataset and COCO dataset and CPU (Intel® Core™ i7-7700) is depicted in Table III. In overall, the proposed accelerator achieves 21.57 GOPs which is approximately 4 times speedup over software implementation.

In Table IV, the work is compared to other prior HSL-based designs. In this work, the board used: Cyclone V PCIe FPGA which is different from the hardware used in the previous study. To make a fair comparison, the performance density of the proposed design is measured. It clearly indicates that the proposed design implementation achieves comparable performance compared to previous work.

Pictures tested by the design are shown in Fig. 6(a) and Fig. 7. In this work, despite the computation is carried out at lower precision (16-bit) than original Tiny-Yolo-v2 (32-bit), all objects in images can be detected and tagged correctly. Finally, the design achieves the mAP similar with original Tiny-Yolo-v2 running in floating point, and the difference is no more than 1%.

TABLE V. COMPARISON TO PREVIOUS WORK

	[8]	[6]	This work
Device	Stratix-V GXA7	Stratix-VGXA7	Cyclone V PCIe
FPGA Capacity	622K LUTs	622K LUTs	113K LUTs
Model	AlexNet, VGG	AlexNet, VGG	Tiny-Yolo-v2
Design Scheme	OpenCL	OpenCL	OpenCL
Frequency	181MHz	120MHz	117MHz
Precision	float	Fixed(8-16bit)	Fixed(16bit)
Throughput	33.9 GOPs	117.8GOPs	21.6 GOPs
DSP Consumed	162	246	122
Performance Density	0.21GOPS/DSP	0.29GOPS/DSP	0.18OPS/DSP

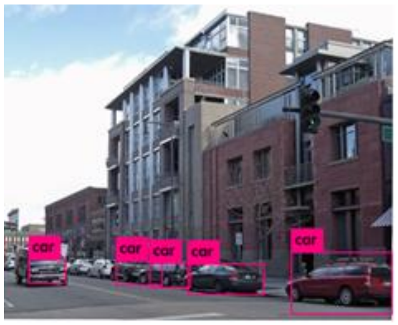

Fig. 6. Tested Image of Car.

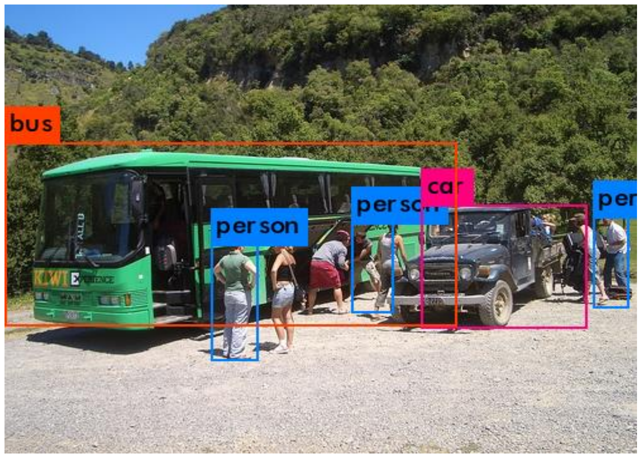

Fig. 7. Tested Image of Bus, Person, and Car.

V. CONCLUSION

In this work, a scalable CNN-based object detection algorithm is implemented on FPGA with fixed-point implementation using OpenCL approach. Further, a way of merging batch normalization layer in the convolutional layer is proposed to improve the performance and reduce the hardware resource. Finally, Tiny-Yolo-v2 is implemented on Cyclone V PCIe FPGA and achieved comparable performance density of 0.18 GOPs/DSP compared to previous work. The proposed implementation can achieve a peak throughput of 21 GOPs under 100 MHz working frequency.

ACKNOWLEDGMENT

Authors would like to thank the support of UTeM Zamalah Scheme to support this work. In addition, authors also thank to the support of Center for Telecommunication Research and Innovation (CeTRI), Faculty of Electronic and Computer (FKEKK), Universiti Teknikal Malaysia Melaka (UTeM).

REFERENCES

[1] A. Krizhevsky, I. Sutskever and GE. Hinton, "Imagenet classification with deep convolutional neural networks", Advances In Neural Information Processing Systems, 2012, pp. 1097-1105,

[2] K. Simonyan, and A.Zisserman, "Very deep convolutional networks for large-scale image recognition", arXiv preprint arXiv:1409.1556, 2014.

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks", Advances In Neural Information Processing systems, 2015, pp. 91-99.

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection", Proceedings of IEEE Conference on Computer Vision and Pattern recognition, 2016, pp. 779-788.

[5] J. Redmon, and Farhadi, "A. YOLO9000: better, faster, stronger", arXiv, 2017.

[6] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.S. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2016, pp. 16-25.

[7] J. Zhang, and J. Li, "Improving the performance of OpenCL-based fpga accelerator for convolutional neural network", Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2017, pp. 25-34.

[8] D. Wang, J. An, and K. Xu, "PipeCNN: An OpenCL-Based FPGA Accelerator for Large-Scale Convolution Neuron Networks", arXiv, vol. 1611.02450, 2016.

[9] J. Ma, L. Chen, Gao, "Hardware Implementation and Optimization of Tiny-YOLO Network", International Forum on Digital TV and Wireless Multimedia Communications, Springer, Singapore, 2017, pp. 224-234.

[10] M. Everingham, L. Van Gool, C.K. Williams, J. Winn and A. Zisserman, "The pascal visual object classes (voc) challenge", International journal of computer vision, 2010, 88(2), pp.303-338.

[11] T.Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C.L. Zitnick, "Microsoft coco: Common objects in context", European conference on computer vision, Springer, 2017, pp. 740-755.

[12] FPGA SDK for OpenCL Programming Guide., Intel, 2017, pp. 70-80.

[13] FPGA SDK for OpenCL Best Practice Guide, Intel, 2017, pp. 17-20.

[14] J. Cong, and B. Xiao, "Minimizing computation in convolutional neural networks", International Conference on Artificial Neural Networks, 2014, pp. 281-290.