International Journal of Reconfigurable and Embedded Systems (IJRES)
Vol. 8, No. 3, November 2019, pp. 206~214
ISSN: 2089-4864, DOI: 10.11591/ijres.v8.i3.pp206-214

A scalable FPGA based accelerator for Tiny-YOLO-v2 using openCL

Yap June Wai, Zulkanain Mohd Yussof, Sani Irwan Md Salim
Faculty of Electronic and Computer Engineering, Universiti Teknikal Malaysia Melaka, Malaysia

Article Info

Article history:

Received Jul 9, 2019
Revised Sep 10, 2019
Accepted Oct 5, 2019

Keywords:

CNN
FPGA
Object detection
OpenCL
Tiny-YOLO-v2

ABSTRACT

Deep Convolution Neural Network (CNN) algorithm have recently gained popularity in many applications such as image classification, video analytic, object recognition and segmentation. Being compute-intensive and memory expensive, CNN computations are common accelerated by GPUs with high power dissipations. Recent studies show implementation of CNN on FPGA and it gain higher advantage in term of energy-efficient and flexibility over Software-configurable-GPUs. The proposed framework is verified by implement Tiny-YOLO-v2 on De1SoC. The design development in this project is HLS approach to ease effort from writing complex RTL codes and provide fast verification through emulation and profiling tools provided in the OpenCL SDK. To best of our knowledge, this is the first implementation of Tiny-YOLO-v2 CNN based object detection algorithm on a small scale De1SoC board using Intel FPGA SDK for OpenCL approach.

Corresponding Author:

Yap June Wai,
Faculty of Electronic and Computer Engineering,
Universiti Teknikal Malaysia Melaka,
Hang Tuah Jaya, 76100, Durain Tunggal, Melaka, Malaysia.
Email: M021710012@student.utem.edu.my

1. INTRODUCTION

Convolutional neural network (CNN), as a well-known deep learning architecture inspired by artificial neural network, has been primarily employed in various applications including image [1, 2] and video classification [3], text recognition [4], speech recognition [5] and object detection [6-8]. The state-of-the-art CNN based algorithms usually consist millions of parameters that require over billion operations to process a single image. This is a great computational challenge for general purpose processors (CPU) to implement CNN-based applications efficiently. Thus, various accelerator such as GPU, FPGA and ASIC have been explored recently to improve the throughput of CNN designs. Currently, FPGA have received huge attention of researchers due to their relatively high performance, low power consumption, reconfigurability and fast development round, especially with the introduction of High Synthesis Tool (HLS), which enable automatic compilation from high-level program (C/C++) to register-transfer-level (RTL) [9-11].

Previous FPGA-based CNN accelerator designs [9-11] mainly focused on optimizing the performance and computational resources only on classification CNN-based algorithms such as AlexNet [1] and VGG [2]. This research focus on developing CNN-based object detection algorithm Tiny-YOLO-v2 that can run on FPGA accelerator, De1SoC. In contrast to classification task, object detection localizes and classify a variable number of objects on an image which indicates that the output of object detection may change from image to image. This is a great challenge to fit computational intensive and memory intensive object detection algorithm: Tiny-YOLO-v2 into De1SoC board which has very limited hardware resources. To the best of our knowledge, this research is the first CNN-based detection algorithm implemented on FPGA, De1SoC using Intel FPGA SDK for OpenCL approach. We summarize the key contributions

as follows: A scalable CNN-based object detection algorithm Tiny-YOLO-v2 that can run on De1SoC using Intel FPGA SDK for OpenCL version 16.1.A novel approach to do the data rearrangement-on-flight to eliminate the need to store the computed output to reduce the memory requirement for hardware. In depth analysis on Tiny-YOLO-v2 and identify the computational and memory complexity.

OpenCL Framework

Intel FPGA SDK for OpenCL allows user to abstract away the traditional hardware FPGA development flow by using high-level synthesis tools. It is an alternative approach to traditional RTL design concept such as Verilog or VHDL with C or C++ synthesis. Figure 1 illustrates the OpenCL-based FPGA accelerator development flow. In the OpenCL framework for SoC specifically, the CPU, in our case, is the ARM Cortex-A9 acts as the OpenCL host and it has bridges interconnect the Cyclone V which it serves as an OpenCL device, forming a heterogeneous computing system. An OpenCL code, which is written in C/C++ like syntax, is translated into hardware image, supported by OpenCL runtime driver. Furthermore, on the host side (ARM), C/C++ code runs on the CPU, providing vendor specific application programming interface (API) to communicate with the implemented kernels on the Cyclone V accelerator. This work utilizes Intel FPGA for OpenCL v16.1 toolset for compiling, emulation and profiling kernels on FPGA.

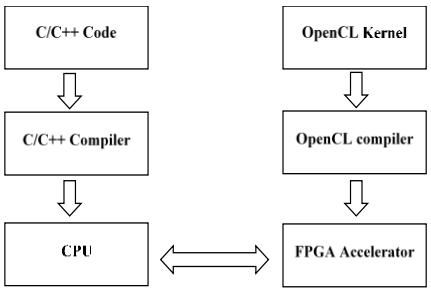

Figure 1. OpenCL-based FPGA development flow

CNN Basic Operations

Convolutional neural network (CNN) is first inspired by the visual cortex of mammals in neuroscience research. It is a machine learning algorithm well suited for both classification and object detection task. As a classical supervised learning algorithm, CNN applies a feed-forward stage for object classification or detection, whereas a backward stage for training. During training, the ground truth outputs are known, and the error is computed between the correct and computed output. The error is then used to back-propagate through the network. In industrial practice, the training process is usually trained offline and the trained CNN network will be used to perform recognition jobs. Figure 2 shows the architecture of Tiny-YOLO-v2 [8], 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 416x416 to 20 output classes.

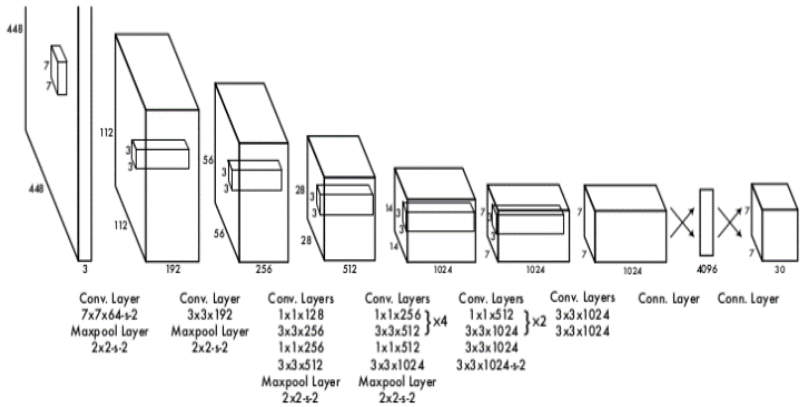

Figure 2. Architecture of Tiny-YOLO CNN

Convolutional Layer

The convolutional layer is the core functional layer of a Deep Neural Network architecture. Previous study [12] proved that the convolutional layer will occupy over 90 % of the feed-forward computation period. Hence, in this work, we will focus the optimization on the convolutional layer. Convolutional layer applies the convolution operation to the image input and pass the result to the next layer. The convolutional operation in CNN model can be expressed as follows:

$$Y[n][w'][h'] = \sum_{w=0}^{FW-1} \sum_{h=0}^{FH-1} \sum_{d=0}^{FD-1} W[n][w][h][d] \times X[W'+h][d] \quad (1)$$

where: Y = Output Feature
W= Trained Weights
X=Input Feature

Activation Functions

Activation function in a CNN architecture is used to transform the input value to the output value before the pooling layer. Sigmoidal activation functions were most often used in CNN are bounded by a maximum and minimum value and thereby causing the saturated neuron in higher layers of neural network. Alternatively, rectifier linear units (ReLU) have been proposed as an activation function. Leaky ReLU Figure 3 in contrast to sigmoidal functions are unbound and it can represent any non-negative real value.

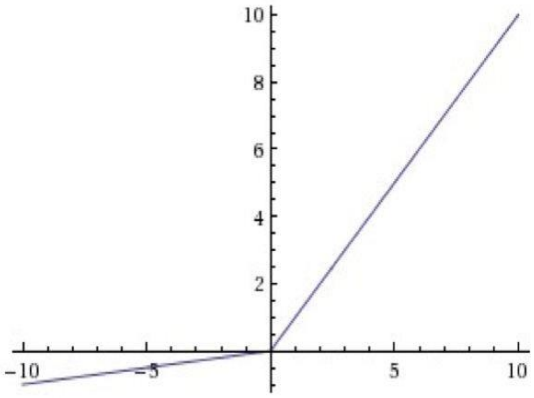

Figure 3. Leaky activation function

Pooling Layer

Pooling layer in general is a form of dimensionally 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 is illustrated as following:

$$S[i, j] = \max\{S[i;, j'] : i \le i' < i+, j \le j' < j + p\} \quad (2)$$

$$S[i, j] = average\{S[i;, j'] : i \le i' < i+, j \le j' < j + p\} \quad (3)$$

Batch Normalization

Batch normalization is implemented in the convolutional layer to provide any layer in Tiny-YOLO-v2 with inputs that are zero mean/unit variance. that the formula for the batch normalization Shows (4). This operation is carried out after the convolution operation and before the activation function. The input layer is normalized by adjusting and scaling the activations. Batch normalization allows each layer to learn in more independent way. This helps to reduce the overfitting because it has a slight regularization effects similar to the dropout.

$$X^{(i)} = \frac{(X^{(i)} - \mu)}{\sqrt{\sigma^2 + \xi}} \quad (4)$$

where: $X^{(i)}$=Input
μ=Mean
σ^2=Variance
ξ =Constant

The rest of the paper is organized as follows. Section 2 presents a detailed case study on computational and memory complexity of Tiny-YOLO-v2 and a detailed description on the proposed accelerator design using General Matrix-Matrix Multiplication (GeMM). Section 3 presents the experimental results and the resource utilization report of our CNN object detection design a section 4 concludes the paper.

2. RESEARCH METHOD

2.1. YOLO object detection algorithm

In this section, we present a detail exploration on the Yolo object detection framework. Prior object detection algorithm repurpose classifiers of localizers to perform detection [6]. These classifiers/localizers are applied to an image at multiple location and various scales. However, Yolo [7, 8] uses a totally different approach to apply a single convolutional network to the full image and predict multiple bounding boxes and class probability for those boxes. This makes Yolo is extremely fast which can run at 45 frames per second on a TitanxGPU. Figure 4 illustrated how object detection task is reframed as some single regression problem straight form image pixels to bounding box coordinates and class probabilities. The input image is divided into S×S grid. Each grid cell predicts B bounding boxes, confidence for those boxes and C class probabilities. These predictions are encoded as an SS×(B*5+C).

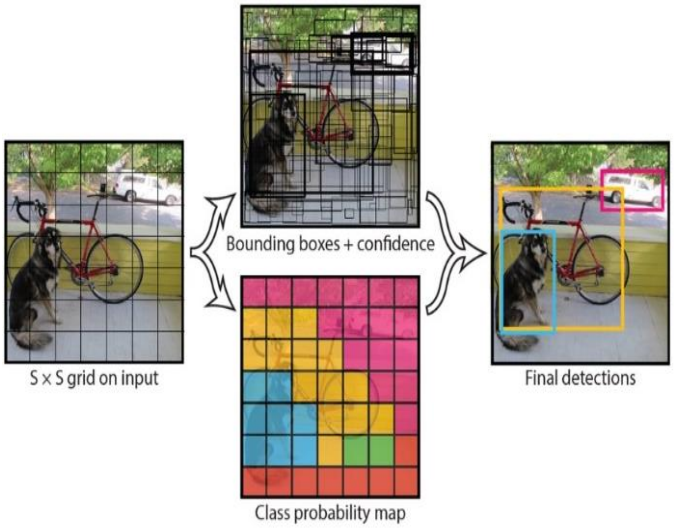

Figure 4. YOLO object D=detection algorithm

2.2. Analysis of computational complexity of Tiny-YOLO-v2

In this section, we present an analysis on the computational complexity and the memory requirement of Tiny YOLO-v2. In convolutional layer, each input feature map is convolved with a sliding window/filter with size K×K, which resulting in a H_{out}×W_{out} output feature map. The number of input and output is N_{in}, N_{out} respectively. Notice that each pixel in an output feature map is the result of addition of Nin pixels that requires K×K of multiplications and additions operations. The total amount of operations in convolution layer can be approximately calculated as shown in (5). Noted that, this equation ignores number of operations for the batch normalization and leaky activation for each layer.

$$\#Operations = 2 \times Nin \times K \times K \times Nout \times Hout \times Wout . \quad (5)$$

The memory requirement is described with space complexity. The main parameter in the Tiny YOLO-v2 is the weight which is used in the convolutional layer. The number of weight in the convolutional layer can be expresses as:

$$\#Weights = Nin \times K \times K \times Nout . \quad (6)$$

Both computational complexity and memory requirement for 9 layers of convolutional layer of Tiny-YOLO-v2 is summarized in Table 1. Noted that, Tiny-YOLO-v2 takes approximately 7.3 billion operations with 15 million of weights just for one image input.

Table 1. Tiny-YOLO-v2 configurations

Layer	Input				Output			#Operations	#Weights
	Nin	Hin	Win	K	Nout	Hout	Wout		
1	3	416	416	3	16	416	416	149,520,384	432
2	16	208	208	3	32	208	208	398,721,024	4,608
3	32	104	104	3	64	104	104	398,721,024	18,432
4	64	52	52	3	128	52	52	398,721,024	73,728
5	128	26	26	3	256	26	26	398,721,024	294,912
6	256	13	13	3	512	13	13	398,721,024	1,179,592
7	512	13	13	3	1024	13	13	1,594884,096	4,718,592
8	1024	13	13	3	1024	13	13	3,189,768,192	9,438,184
9	1024	13	13	3	125	13	13	43,264,400	128,000

2.3. Implementation of 3D convolution as 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 regions of input. To take advantage of this, we implemented GeMM based convolution like [11, 13]. In fact, GeMM based convolution approach is also been practiced in GPU based accelerator for CNN classification [1, 2] and object detection task [7, 8]. Figure 5 shows that how the first layer of convolution layer (3D) is flattening and rearranged vertically into a 2D matrix through im2col process. For example, the dimension of the input layer for Tiny-YOLO-v2 is 416 ×416×3 (Hin×Win Nin) and the size of kernel is 3×3 (K×K). The Im2Col operation flatten 3D input layer dimension (416×416×3) and rearrange vertically into a 2D matrix of dimension 416×416×3×3×3.

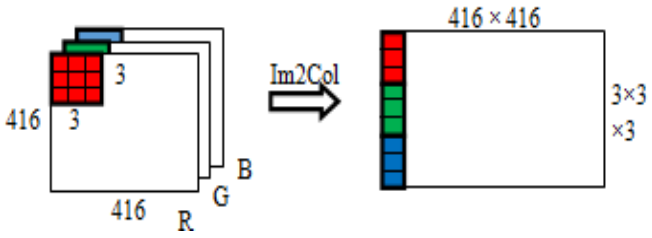

Figure 5. Im2Col operation

Notice that the im2col 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. Hence, we proposed the pseudo code as shown in Figure 6 to perform the im2col operation on-the-fly by storing the corresponding pixels into FPGA's local memory before the matrix multiplication. Implementation of convolution as GeMM a shown in Figure 7.

1. Get current work-item *id* (*global_y, global_x, local_y, local_x, block_y, block_x*)
2. Compute current output pixel coordinate (*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 previous computed output pixel coordinate
4. Read the actual pixels value
5. *value = 0 < id ≤ input_dimension ? input[id]: 0*

Figure 6. Pseudo code Im2Col on-fly

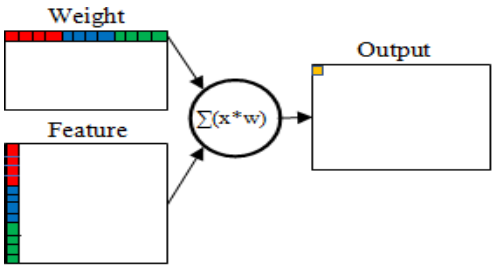

Figure 7. Implementation of convolution as GeMM

The weights of the convolution layer are also stretched out into rows. As an example, Figure 7 illustrates that convolution can be done by performing one large matrix-matrix multiplication [sizexsize]. The pseudo code for the matrix multiplication-based convolution in our proposed design is shown in Figure 8. To accelerate the GeMM operation, two scalable design parameters BLOCK_SIZE and SIMD vectorization factor is introduced. SIMD is representing the factor by which data are vectorized and executed in SIMD manner. This parameter is 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.

1. Get current work-item id
2. Do im2col on-the-fly (refer to Figure 6)
3. Compute the address location weights
4. Load input features and weight features into local memory
5. Execute MAC operations depending on SIMD parameters
6. Wait all operation done
7. Add bias to the convolution output
8. Execute batch normalization
9. *Batchnorm=(output[i]–mean[global_y]) *variance[global_y];*
10. Leaky Activation Function
11. Store the convolution output correspondingly

Figure 8. Pseudo code of proposed GeMM convolution

2.4. Data preprocessing

According to (4), part of the computation in batch normalization involves division. However, division operators are very expensive to implement in FPGAs and might degrade kernel performance [14]. In addition, the implementation of division requires a significant amount of hardware resources. Since Tiny-YOLO-v2 applies a feedforward process for object detection and a backward path for training. The network is commonly trained offline and the parameters of σ and β are to be learned during the training process and these values are only load once during the inference process. Hence, it is more reasonable offload the division operation to the host processor and then pass the result as an argument to the kernel. Hence, the (4) above is slightly modified as shown in (7). To avoid the redundancy and hardware resource-intensive operation (division), the following calculation for value β is performed in the host application and then β is passed to the kernel as an argument for all work-items in ND Range to use.

$$X^{(i)} = (X^{(i)} - \mu) \times \beta \quad (7)$$

where: $\beta = \frac{1}{\sqrt{\sigma^2 + \xi}}$

3. RESULTS AND ANALYSIS

In this section, we first present the validation results of proposed architecture developed using OpenCL16.1 Intel FPGA SDK HLS to accelerate the large-scale CNN-based object detection algorithm: Tiny-YOLO-v2 on FPGA board. The hardware specification of the Cyclone V board is summarized in Table 2.

Table 2. Harware specification of FPGA board

Specification	De1SoC Board
FPGA	Cyclone V 5CSEMA5F31C6
HPS	Dual Core ARM Cortext-A9
Logic Element	85k
RAM Blocks	397
DSP Blocks	87
External Memory	1GB DDR3

Secondly, the best performance for the runtime used the parameter configuration BLOCK_SIZE=16 and SIMD=2. Note that it is the best configuration for De1SoC to fit for best performance, as logic resource is very limited. The resource utilization of proposed architecture is summarized in Table 3. According to the report, we can tell that our proposed architecture has almost utilized all the available FPGA's hardware resource.

Table 3. Resource utilization

Resource	Available	Used	Utilization
Logic (AMs)	32070	29186	91%
RAMs	397	391	98%
DSPs	87	87	100%

The execution time of the CNN layers in Tiny-YOLO-v2 implemented on De1SoC is illustrated in Figure 9. It is noted that the final classification time without kernel profiling will be significantly lower as shown in Figure 9 due to the delay involved in generating the profiling report. The total detection time per image using the proposed architecture is 1.40 second, achieving around 5.2 GFLOPs throughput on De1SoC. It is expected that the proposed architecture can achieve much higher throughput on other large-scale FPGA boards: Stratix and Aria FPGA board.

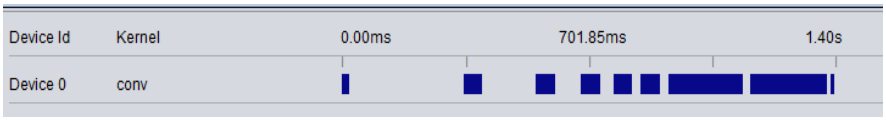

Figure 9. Pseudo code of proposed GeMM convolution

Thirdly, we present on the how data preprocessing mentioned in previous section will help to utilize the hardware resources on FPGA. As we mentioned, De1SoC board has limited available hardware resources to implement large scale CNN-based object detection algorithm such as Tiny-Yolo-v2. Hence, by modifying the equation listed in (4) into (7) to offload part of expensive computation operations such as division and modulus to be done in host program helps to save up to 11% logic element consumption. This leads to significant improvement to the object detection performance, as more resources can be used to further scale up the proposed architecture. The hardware resource utilization using data preprocessing approach is summarize in Table 4.

Table 4. Resource utilization using data preprocessing

Resource	(4)	(7)	Utilization
Logic (AMs)	97%	88%	11%
ALUTs	60%	54%	6%
FFs	42%	38%	4%
RAMs	101%	88%	13%
DSPs	93%	84%	9%

Figure 10 shows the design space exploration for Tiny-YOLO-v2 model on De1SoC. It shows that the resource utilization increases around 10% when we increase the BLOCK_SIZE parameter in scale of multiple 4. At the same time, the execution time to inference on input image is improved from 4.16s to 1.40s by increasing the BLOCK_SIZE parameters. Currently, due to the limited hardware resource on De1SoC, BLOCK_SIZE=32 and SIMD=4 is too large to fit in the FPGA device and is not reported. All the object in the image is correctly detected without hurting the detection accuracy when the proposed architecture is scaled from BLOCK_SIZE=4 to BLOCK_SIZE=16.

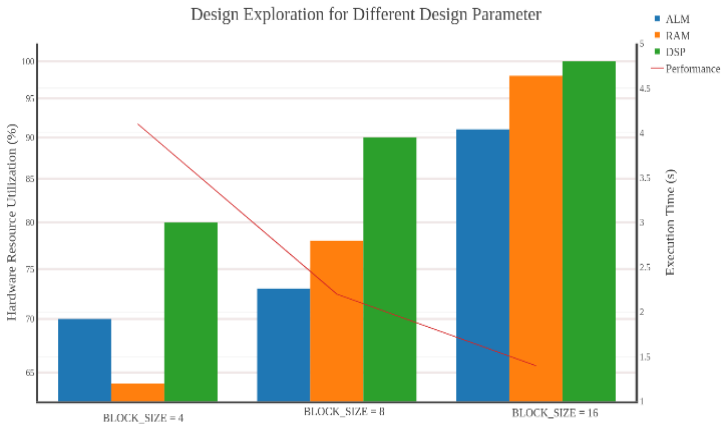

Figure 10. Design space exploration

4. CONCLUSION

In this work, we implemented GEMM based convolution method for convolutional neural network object detection algorithm: Tiny-YOLO-v2. We then present in-depth analysis on Tiny-YOLO-v2 computational and memory complexity. We also implemented im2col operations on-fly during data fetching stage in kernel. Finally, we realize the implementation the first CNN-based object detection algorithm on De1SoC.

ACKNOWLEDGEMENTS

Authors would like to thank Universiti Teknikal Malaysia Melaka (UTeM) and SKIM Zamalah for supporting this research

REFERENCES

[1] Krizhevsky, A., Sutskever, I. and Hinton, G.E. "Imagenet classification with deep convolutional neural networks," In *Advances in neural information processing systems*, pp. 1097-1105, 2012.
[2] Simonyan, K. and Zisserman, A. "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[3] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R. and Fei-Fei, L. "Large-scale video classification with convolutional neural networks," In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 1725-1732, 2014.
[4] Lai, S., Xu, L., Liu, K. and Zhao, J. "Recurrent Convolutional Neural Networks for Text Classification," In *AAAI*, Vol. 333, pp. 2267-2273, 2015.

[5] Abdel-Hamid, O., Mohamed, A.R., Jiang, H., Deng, L., Penn, G. and Yu, D. "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on audio, speech, and language processing*, *22*(10), pp.1533-1545, 2014.
[6] Ren, S., He, K., Girshick, R. and Sun, J. "Faster r-cnn: Towards real-time object detection with region proposal networks," In *Advances in neural information processing systems*, pp. 91-99, 2015.
[7] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. "You only look once: Unified, real-time object detection," In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779-788, 2016.
[8] Redmon, J. and Farhadi, "A. YOLO9000: better, faster, stronger," *arXiv preprint*, 2017.
[9] Wang, D., An, J. and Xu, K. "PipeCNN: An OpenCL-Based FPGA Accelerator for Large-Scale Convolution Neuron Networks." *arXiv preprint arXiv:1611.02450*, 2016.
[10] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B. and Cong, J. "Optimizing fpga-based accelerator design for deep convolutional neural networks," In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161-170, ACM, 2015.
[11] Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S., Seo, J.S. and Cao, Y. "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 16-25, ACM, 2016.
[12] Cong, J. and Xiao, B. "Minimizing computation in convolutional neural networks," In *International conference on artificial neural networks*, pp. 281-290, 2014.
[13] Zhang, J. and Li, J. "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network", In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 25-34, ACM, 2017.
[14] Intel, "FPGA SDK for OpenCL Programming Guide", 2017.