Real Time CNN-based Hardware/Software Co-Implementation for Object Detection in Autonomous Driving.

Abstract-- Convolution Neural Network (CNN) is considered now the state of the art of the image processing and classification due to its high accuracy in many applications like autonomous driving and face detection, but in order to achieve this accuracy, this comes with a price of using computationally expensive and high power consumption graphical processing unit (GPU) which is also not always suitable to be used in Real time applications and mobile devices, so coming up with solutions for acceleration and reducing power consumption is a now a need. Field Programmable Gate Array (FPGA) can provide a good alternative due to its low power consumption and dynamic reconfigurability. This work proposes the software implementation of a Faster-RCNN with the hardware implementation of its backbone neural network (VGG) using Vivado® High-Level Synthesis on Xilinx Virtex®-7 and using fixed point calculations instead of floating point.

Abbreviations and Acronyms.

- ReLU: Rectified linear unit.
- Convnet: Convolutional neural network.
- R-CNN: Region based Convolutional neural network.
- ROI: Region of interest.
- HLS: High level synthesis.

Keywords: Real time, Object detection, Autonomous driving.

I. INTRODUCTION.

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use it learn for themselves. Convolution Neural Network (CNN) is type of Deep Neural Network (DNN) that is an application of machine learning. One of the important applications of CNN recently is the autonomous driving which has become a concrete reality and may pave the way for future systems where computers take over the art of driving, but as a Real Time application that has a critical safety priority, it does not only need to achieve very high accuracy but also needs to be fast.

A. Problem Definition

It is worth asking at this point whether distrust of autonomous technology has a semblance of the truth and is it backed by some authoritative research. According to the WHO, approximately 1 million people die due to road accidents every year. And most of the accidents are caused due to avoidable reasons. Considering the fact that more than 90% of fatal car accidents are caused due to human negligence, the assertion that autonomous cars would not lead to a significant decline in the number of car crashes is simply absurd and defies even the most elementary logic.

B. Related Work.

In the literature we saw a lot of Faster-RCNN implementation like in [4] , [6] and in [7] that lead to great results we decided to use the architecture used in [4] to train our custom dataset to extract the weights we will need in our hardware part more of that will be explained later, We see that HLS tool led to great results concerning the hardware part in [8] when constructing a hardware CNN so we decided to use the HLS tool to construct the backbone network of our Faster-RCNN that is VGG-16, We can see some problems tackled when doing a VGG-16 network on a FPGA like in [9].

C. Objective and Contribution.

In this work, we propose a real time CNN-based hardware/software Co-implementation for Object Detection in Autonomous Driving. Implementation is done using TensorFlow and High-level Synthesis methodology (C++ to Verilog). Critical parts in the design are implemented in hardware as it is faster than software. In object detection in autonomous cars, we need very fast response as very small delays may lead to hazardous events.

D. Organization of the Remaining Sections.

The rest of this paper is organized as follows.

In Section II we present the history and Background about Autonomous cars, and an overview about convolution neural networks

In Section III we discuss our proposed architecture.

In Section IV we discuss our Implementation which includes both software and hardware Implementation.

In Section V, validation and results are presented.

A. *History and Background*

1)Autonomous Car Components

The monitoring system of autonomous cars that consists of powerful sensors and LiDAR's view the objects ahead, with much more clarity, precision and a 360-degree view than the human eye and then chart the route of the car accordingly. While autonomous cars are good at identifying hurdles ahead, it cannot distinguish between pedestrians and inanimate objects.

Human-driven cars cannot overrule autonomous cars because, in any given time and situation, they are statistically safer than human-driven cars, and intelligent AI-powered systems are immune to the foibles and temperamental indiscretions of humans.

What is needed is setting public expectations and gradually erasing doubts that the citizens have. Along with companies like Waymo, governments should also step in to allay the fears and disseminate the advantages of autonomous cars — that range from reduction in fuel consumption, less traffic congestion to the decreased probability of a collision. A framework conducive to emerging enterprises and minimal regulations that promote and incentivize innovation is needed, in addition to a dedicated task force and participatory approach between governments and private industry. No system is engineered to be free from the probability of error margin. The error margin can only be made negligible and then the cause of the error can be identified and rectified with a permanent effect. In autonomous cars, because of extensive research and cutting-edge technical equipment used, the error margin is already very low, and it is possible that it may be obliterated altogether once the test phase is over and the technology incorporates new innovations. Considering the enormous potential of autonomous car technology and the benefits it has in store for the society, to write it off and spell doom is akin to throwing the baby along with the bathwater.

Self-driving cars might seem like a recent thing, but experiments of this nature have been taking place for nearly 100 years. In fact, centuries earlier, Leonardo Da Vinci designed a self-propelling cart hailed by some as the world's first robot. At the World's Fair in 1939, a theatrical and industrial designer named Norman Bel Geddes put forth a ride-on exhibit called Futurama, which depicted a city of the future featuring automated highways. However, the first self-driving cars didn't arrive until a series of projects in the 80s undertaken by Carnegie Mellon University, Bundeswehr University, and Mercedes-Benz. The Eureka Prometheus Project proposed an automated road system in which cars moved independently, with cities linked by vast expressways. At the time, it was the largest R&D project ever in the field of driverless cars. Since 2013, US states Nevada, Florida, California, and Michigan have all passed laws permitting autonomous cars; more will surely follow. Autonomous vehicle components. Ouster's OS-1 3D LiDAR sensor captures point cloud data and camera-like images to help autonomous vehicles 'see' their environment how does an autonomous vehicle operate and make sense of what it sees? It comes down to a powerful combination of technologies, which can be roughly divided into hardware and software. Hardware allows the car to see, move and communicate through a series of cameras, sensors and V2V/V2I technology, while software processes information and informs moment-by-moment decisions, like whether to slow down. If hardware is the human body, software is the brain.

Autonomous vehicles rely on sophisticated algorithms running on powerful processors. These processors make second-by-second decisions based on real-time data coming from an assortment of sensors. Millions of test miles have refined the technology and driven considerable progress – but there is still a way to go. Driverless car technology companies the race to be the first self-driving car on the road is heating up. Level 5 autonomy is still some time away, but there is plenty else happening in the autonomous space, with aspects of driverless tech already making an appearance in today's mass-produced cars. Early adopters will enjoy benefits such as automatic parking and driving in steady, single-lane traffic.

2)Convolution Neural Networks.

While a neural network is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes Thus, a neural network is either a biological neural network, made up of real biological neurons, or an artificial neural network, for solving artificial intelligence (AI) problems. The connections of the biological neuron are modelled as weights. A positive weight reflects an excitatory connection, while negative values mean inhibitory connections. All inputs are modified by a weight and summed. This activity is referred as a linear combination. Finally, an activation function controls the amplitude of the output. For example, an acceptable range of output is usually between 0 and 1, or it could be −1 and 1.

These artificial networks may be used for predictive modelling, adaptive control and applications where they can be trained via a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information.

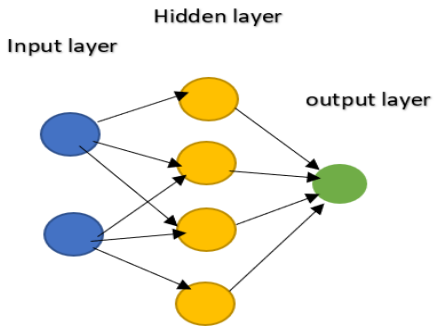


Figure 1 A simple neural network.

We used a dataset of common objects we see normally in our daily life in streets to train our neural network to be able to recognise these objects in any future image that would be our input to the neural network.

The main CNN layers and functions that are used to build our proposed architecture on the SW and HW levels are as follow:

4) Convolution Layer.

The convolution block is concerned to be the main block to build any (CNN) as it contains the most of the computations and is repeated many times through the network, as shown in figure (2), the inputs of this block are the filters also known as kernels of size (R x R) and number of channels equals to C, this number is always equivalent to the number of channels of the input image or input feature map that can be in a size of (K x K) ,every element of the (R x R) kernel is multiplied by a window size e(R x R) of the input image this is done for every channel of the kernel with the corresponding channel of the input image and then accumulated and added to a bias to be a one element of the feature map, then the same kernel is sliding with a certain stride through the input image and again multiplied and accumulated and added to a bias to produce another element of the feature map and the process is repeated till finishing the whole input image, and the whole process is repeated M times (number of filters) to produce M feature maps each of size (N x N) where N equals to (K – R +1) due to fixed stride in our proposed CNN.

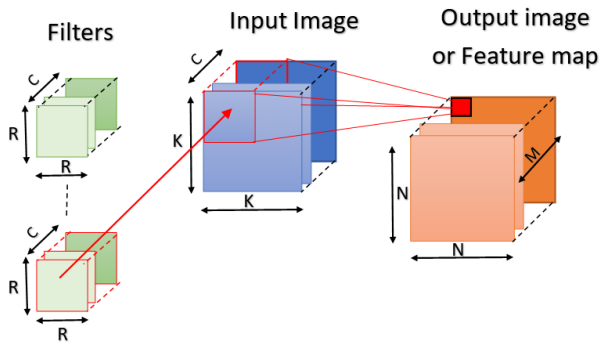


Figure 2: The Convolution Operation

5) Pooling Layer.

A key aspect of Convolutional Neural Networks is pooling layer, applied always after the convolution layers to reduce the dimensions of the output feature map. It has different types like Max pooling, Average, sum and other types. In our work we use max pooling as its better in practice and easier to implement in Hardware.

6) Rectified Linear Unit ReLU.

There are many activation functions used in CNNs like sigmoid and tanh but one of the most effective functions and the one we use in our architecture is the Rectifier linear unit or ReLU. It is an activation function that is applied after every convolution layer to introduce the nonlinearities of the data. It returns zero for the negative values and returns the same input for the zero and positive value and can be written as$f(x) = \max(0, x)$.

7) Fully Connected Layer.

The main objective of the fully connected layer is to take the output of the previous convolution and pooling layers and classify the image. The fully connected layer connects every neuron of one layer to every neuron of another layer, a flatten matrix goes through the fully connected layer and use the SoftMax activation function so that the output of each node is a probability value that is normally between zero and one.

II. The Proposed architecture.

The proposed methodology is shown in Figure 3

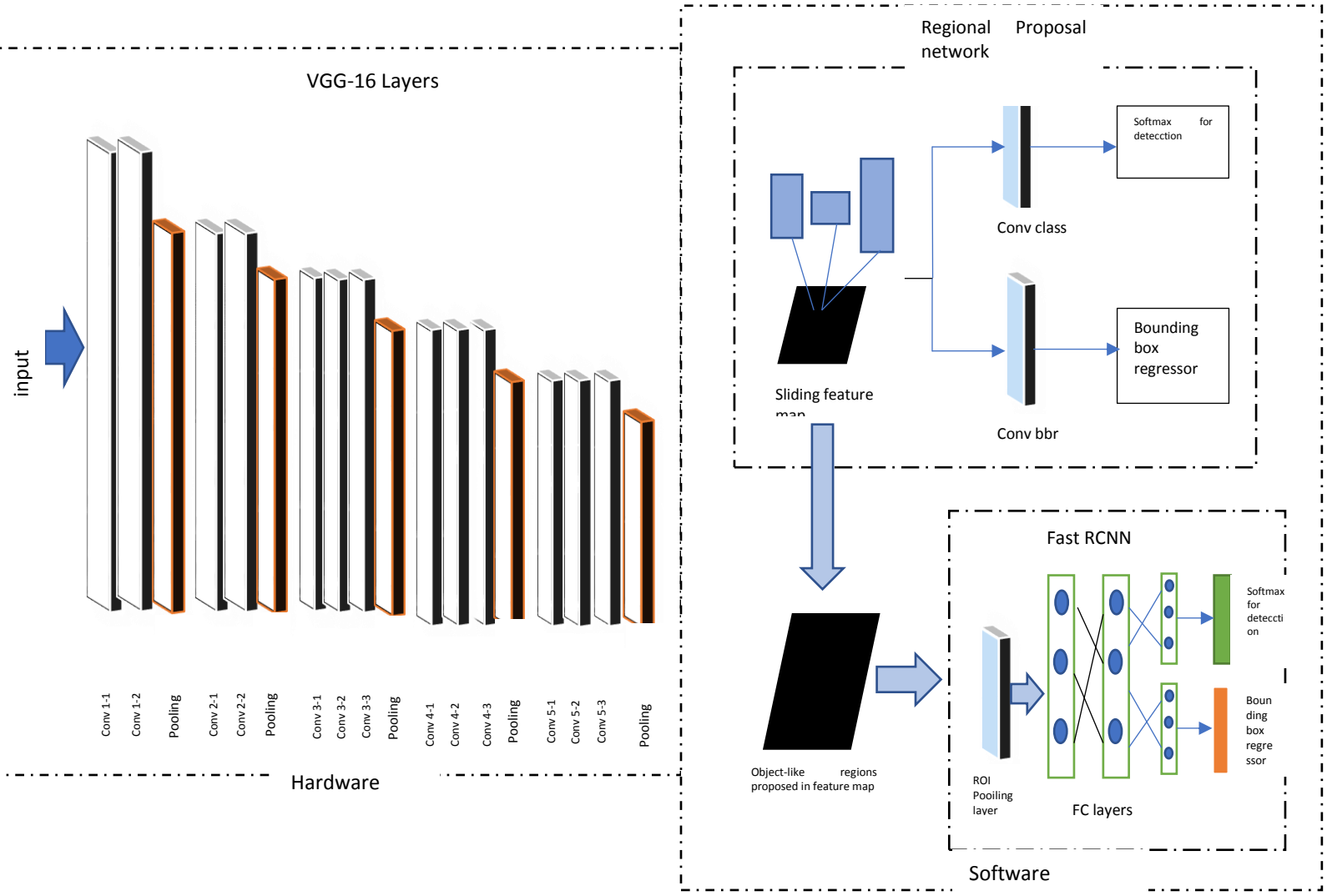


Figure 3 the proposed methodology

Table 1 Comparison between Proposed CNN architectures.

P.O.C	YOLO	SSD	R-CNN	FAST R-CNN	FASTER R-CNN
Accuracy	Low	Low	Very Low	Intermediate	High
Speed	High	Intermediate	Very Low	Low	High
Implementation	Easy	Complex	Easy	Intermediate	Intermediate

In the literature, there are many architectures in the convolutional neural networks as shown in Table 1

In this work, we select the faster RCNN as the backbone for our implementation as it is the best according to autonomous driving constrains. Faster RCNN is the modified version of Fast RCNN. The major difference between them is that Fast RCNN uses selective search for generating Regions of Interest, while Faster RCNN uses "Region Proposal Network", aka RPN. RPN takes image feature maps as an input and generates a set of object proposals, each with an objectness score as output. The below steps are typically followed in a Faster RCNN approach: We take an image as input and pass it to the ConvNet which returns the feature map for that image. Region proposal network is applied on these feature maps. This returns the object proposals along with their objectness score. A RoI pooling layer is applied on these proposals to bring down all the proposals to the same size. Finally, the proposals are passed to a fully connected layer which has a SoftMax layer and a linear regression layer at its top, to classify and output the bounding boxes for objects. To begin with, Faster RCNN takes the feature maps from CNN and passes them on to the Region Proposal Network. RPN uses a sliding window over these feature maps, and at each window, it generates k Anchor boxes of different shapes and sizes. Anchor boxes are fixed sized boundary boxes that are placed throughout the image and have different shapes and sizes. For each anchor, RPN predicts two things: The first is the probability that an anchor is an object (it does not consider which class the object belongs to), second is the bounding box regressor for adjusting the anchors to better fit the object. We now have bounding boxes of different shapes and sizes which are passed on to the RoI pooling layer. Now it might be possible that after the RPN step, there are proposals with no classes assigned to them. We can take each proposal and crop it so that each proposal contains an object. This is what the RoI pooling layer does. It extracts fixed sized feature maps for each anchor, then these feature maps are passed to a fully connected layer which has a SoftMax and a linear regression layer. It finally classifies the object and predicts the bounding boxes for the identified objects.

III. Implementation

A. *Software Implementation*

We know in the software industry that without an interface, libraries, and organized tools, software development proves a nightmare. When we combine these essentials, it becomes a framework or platform for easy, quick, and meaningful software development.ML frameworks help ML developers to define ML models in precise, transparent, and concise ways. ML frameworks used to provide pre-built and optimize components to help in model building and other tasks. There are famous frameworks commonly used as TensorFlow and PyTorch, we expected to use both, so we wanted to know the differences between them.

TensorFlow: TensorFlow is an open source deep learning framework created by developers at Google and released in 2015, it supports regressions, classifications, and

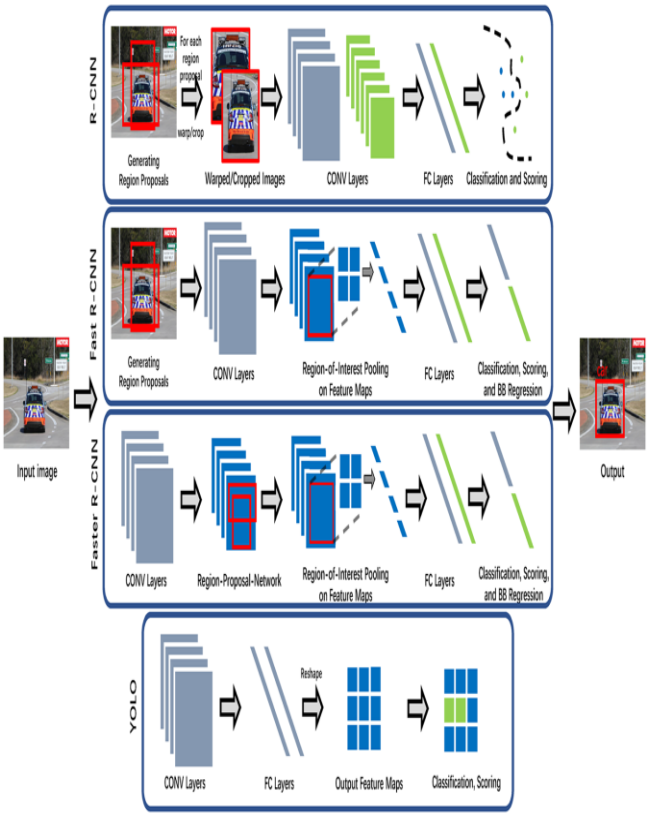


Figure 4 Block diagram for software identification of objects.

neural networks like complicated tasks and algorithms. You can run it on CPUs & GPUs both. TensorFlow creates a static graph, you first must define the entire computation graph of the model and then run your ML model. It's more difficult than PyTorch to learn but it has larger community and easier to find solutions to your problem as it's older than PyTorch.

PyTorch: PyTorch is based on Torch and has been developed by Facebook, it's used by Facebook, IBM, Yandex, and Idiap Research Institute. PyTorch believes in a dynamic graph, you can define/manipulate your graph on-the-go. This is particularly helpful while using variable length inputs in RNNs. Torch is flexible and offers high-end efficiencies and speed. It offers a lot of pre-trained modules. It's easier to learn but it is new compared to TensorFlow, so it has smaller community than TensorFlow.

In the software implementation part, it is divided into 2 parts the first part is we train our custom autonomous driving dataset which is a subset of the VOCtrainval_11-May-2012 dataset and Udacity dataset to train the architecture provided in [4] , we extract a pb file format which is a file format that we can use after that for inference mode, the 2nd part we build a VGG-16 neural network using keras so we can verify that our backbone network we are doing in hardware is working correctly.

1) Full Faster-RCNN implementation

To train the architecture we used we need to put our dataset in a file format called tfrecord which is facilitated by a website called Roboflow, Roboflow makes managing, preprocessing, augmenting, and versioning datasets for computer vision seamless. Developers reduce 50% of their code when using Roboflow's workflow, automate annotation quality assurance, save training time, and increase model reproducibility.

This website is used to change from one format to another to the annotation files and this will be used to generate the formats we will need in training in case you need to train on your custom dataset, We documented the steps that are needed to change the format and to train the architecture In our GitHub repo

https://github.com/Ramy-osama/Hardware-FasterRCNN.

2) VGG-16

In this part we build a full VGG-16 neural network using keras we used took the code in [11] as a reference for our architecture, at first we trained our neural network using the fashion Mnist dataset to make sure that our methodology in extracting weights to the hardware part is correct, the following table shows the architecture used.

Table 2 VGG-16 architecture

Layer (Type)	Output Shape	Param #
Convd2d	(None,28,28,64)	640
Convd2d_1	(None,28,28,64)	36928
Max_pooling2d	(None,14,14,64)	0
Convd2d_2	(None,14,14,128)	73856
Convd2d_3	(None,14,14,128)	147584
Max_pooling2d_1	(None,7,7,128)	0
Convd2d_4	(None,7,7,256)	295168
Convd2d_5	(None,7,7,256)	590080
Convd2d_6	(None,7,7,256)	590080
Max_pooling2d_2	(None,4,4,256)	0
Convd2d_7	(None,4,4,512)	1180160
Convd2d_8	(None,4,4,512)	2359808
Convd2d_9	(None,4,4,512)	2359808
Max_pooling2d_3	(None,2,2,512)	0
Convd2d_10	(None,2,2,512)	2359808
Convd2d_11	(None,2,2,512)	2359808
Convd2d_12	(None,2,2,512)	2359808
Max_pooling2d_4	(None,1,1,512)	0
Flatten	(None,512)	0
Dense	(None,4096)	2101248
Dense_1	(None,4096)	16781312
Dense_2	(None,10)	40970

But we faced a problem when training a full VGG-16 on the fashion mnist dataset as the images in this dataset are of size 28x28 so when passed to the many extraction layers provided in the VGG-16 architecture (convolution, Relu, maxpooling) the size of the feature map will be very small so we can see from the results of training in Table 3 the neural network didn't train.

Table 3 Results of training the fashion mnist dataset on VGG-16 architecture

NO.	EPOCH	LOSS	ACCURACY
1	1/10	2.3029	0.0989
2	2/10	2.3028	0.0988
3	3/10	2.3028	0.0971
4	4/10	2.3028	0.0965
5	5/10	2.3028	0.0988
6	6/10	2.3028	0.0990
7	7/10	2.3028	0.0979
8	8/10	2.3028	0.0990
9	9/10	2.3028	0.0992
10	10/10	2.3028	0.0988

3) Light weight VGG-16 architecture

Due to hardware memory constraints to download weights on it and due to RAM constraints, that is needed to extract the weights from software, A light weight VGG-16 was designed that has much less number of parameters while also having the high accuracy to classifying the fashion mnist dataset that is used to verify that the process of extraction of weights and the blocks constructed in hardware is working properly, the architecture is shown in Table 4

Table 4 lightweight VGG-16 architecture

Layer (type)	Output Shape	Param #
Conv2d	(None, 28, 28, 64)	640
Conv2d_1	(None, 28, 28, 64)	36928
Conv2d_2	(None, 28, 28, 64)	36928
Conv2d_3	(None, 28, 28, 64)	36928
Max_pooling2d	(None, 14, 14, 64)	0
Conv2d_4	(None, 14, 14, 64)	36928
Conv2d_5	(None, 14, 14, 64)	36928
Max_pooling2d_1	(None, 7, 7, 64)	0
flatten	(None, 3136)	0
dense	(None, 1024)	3212288
dense_1	(None, 1024)	1049600
dense_2	(None, 10)	10250

The result of training for 20 epochs was nearly 99% and it achieved 92.4% accuracy after testing it against the test dataset

4) Extraction of weights

After training the CNN, weights are extracted from the layers into a text file that is read in our C++ codes to build our architecture in hardware, we use some Python scripts to make the output format readable and compatible with the fixed point calculations we will use in hardware, also some

more scripts was developed to change the images to an array form so It can be easily stored in memory in the hardware part

B. Hardware Implementation.

The hardware implementation main purpose in this work and in the field of the deep neural networks is the acceleration, the acceleration of CNNs -that requires high complexity and computations to achieve the required accuracy- can be solved by using GPUs, but this solution comes not only with a great cost but also a high power consumption, therefore we are targeting FPGA for implementing the accelerating as it can compromise between the power consumption and the speed of the CNN that is necessary for real time applications [13].

Designing the CNN on FPGA introduces many challenges, one of them is the design method, there is a tradeoff between the main two design methods which are RTL and HLS, the RTL direct HW design may achieve higher efficiency but it consumes time and poses complexity for large designs, on the other hand the HLS method offers quick development and provides many automated features like pipelining.

In this work we use the HLS based design approach as it fits our requirement for fast implementation, reasonable efficiency and accuracy within the available resources.

We use Vivado HLS tool and as shown in figure 4 the flow of the design is to write codes in high level language as C or C++ with considering some limitation and instructions and the tool generates VHDL and Verilog codes that can be synthesized, the tools offers the ability of simulating wave forms, and we can target different FPGAs like ZYNQ directly and specify the type of memory easily (FIFIO, ROM, BRAM, RAM).

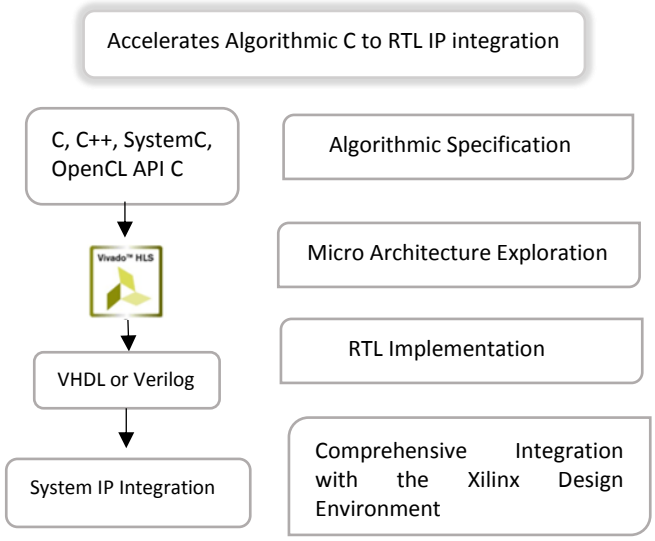


Figure 4: Flow of the HLS design approach using (Vivado HLS).

One other challenge for the CNN design on FPGA is the limited recourses, as known, computations in neural networks are performed traditionally with floating point calculations using either GPU or CPU but when it comes to hardware implementation, using floating point calculation doesn't just make it slower due to the difficulty of controlling the mantissa and the exponent for various operations [12] but also consumes the resources, and to avoid this problem we use fixed point calculations that use much less resources and doesn't affect the performance [1].

Table 5: FPGA resource consumption comparison for multiplier and adder with different types of data.

	Xilinx Logic			
	Multiplier		Adder	
	LUT	FF	LUT	FF
Fp32	708	858	430	749
Fp16	221	303	211	337
Fixed32	1112	1143	32	32
Fixed16	289	301	16	16
Fixed8	75	80	8	8

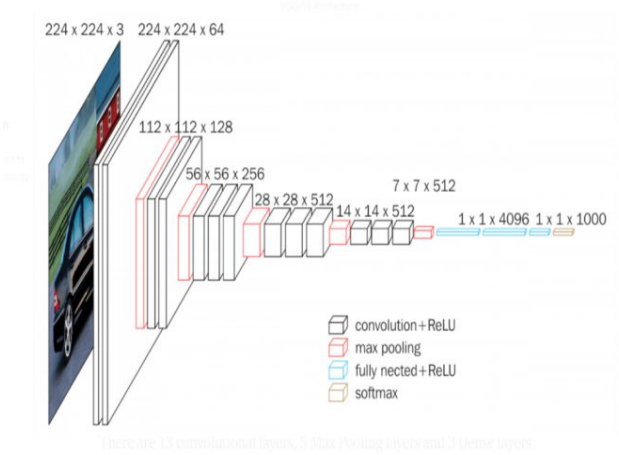


Figure 5 Block diagram of layers of VGG-16.

- As shown in figure 5 the architecture is divided into the following layers:

1) Convolution Layer.

Inputs:

Inputs of this layer are as follows:

Three-dimensional array, two of them represents the size of the image or the feature map which depends on in which layer we are, and the third dimension is the number of channels, for example, the first layer always takes a 3 channels image (RGB).

Four-dimensional array which represents number of kernels, number of channels, and two dimensions for the size.

One-dimensional array of the bias of every kernel applied.

Outputs:
The output is a three-dimensional array of the feature maps which is passed to Relu and Max-pooling layers afterwards. Figure 5 shows the simulation result of convolution layer.

2) Padding.
Inputs:

This is implemented before the convolution, so the input is either the image or the feature map and the main aim of this it is to preserve the image size after convolution so that we stick to the vgg16 CNN architecture.

Outputs:

The output is the padded image and then it will be passed to the convolution layer.

3) Max Pooling Layer.
Inputs:

Retrieve the output of the convolution layer after the Relu operation then halve each dimension by 2 which reduce the number of values to one quarter of the original number (by using a 2x2 window with stride of 2 to avoid any overlap)

Outputs:

The output is the feature maps that contain the maximum value of each four-pixel values in the original input, their number will be equal to the number of the filters and they will be passed to the next convolution layer.

4) RELU Layer.
Inputs:

This layer applies the ReLU operation which is nonlinear operation used to reduce the number of deep layers in the CNN and it basically operates on each value and remove all negative values from the output feature map before Maxpooling layer.

5) Fully-Connected Layer.
Inputs:

This layer applies the Flattening operation in which it takes the output of the previous layer (convolution layer) which is a 3 dimensional array,

Outputs:

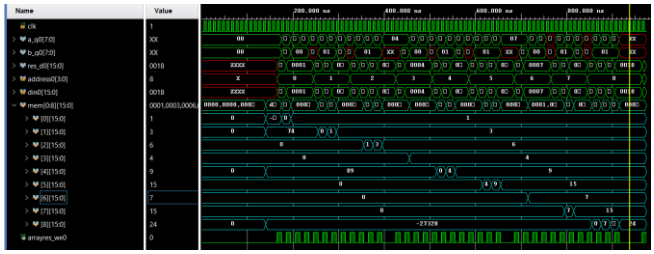
The fully-connected layer “flattens” the previous layer and turns it into single vector which is 1 dimensional array that can be an input for the next stage

***6)* Integration of (Padding-Convolution –Max pooling).**

This layer is integration between the previous layers which work as first layer of vgg16 where all operations of Padding convolution and Maxpooling are applied after each other in this order

- Start the function by padding the input image so the feature map after convolution is same size of input image
- do the convolution with one image RGB , two filters with three channels each
- Do max pooling 2x2 which reduce the output feature map size by 2.

C. An illustrative example



Matrix multiplication is the first example we experimented using the Vivado tool as it is one of the main operations in the project.

Figure 6 Matrix multiplication Simulation Result.

In Figure. 6, the memory signals in the aqua color is output array that has the res_d0 saved in it. It shows clearly that it is the expected output. By tracing the wave form of inputs and the result the timing of everything goes as expected. This operation is basic one in the convolution layer and it gave us a good insight on how to use the tool power like pipelining, optimization techniques and memory implementation. In addition to that the report generated from tool about the resources needed on the FPGA to carry out this process also by referring to table 2 it shows clearly the estimated latency of multiplying 3x3 matrix and generating the output matrix and save it in a memory.

IV. Validation and Results

A. Software Results

The results shown in Figure 7 are of training the Faster-RCNN architecture using our custom dataset based on the architecture in [4], These results are after training for 2000 epoch and reaching loss = 0.102 and we can that the results are good and it can detect to a great extent all the labels that was given in the dataset.

Figure 7 a left, b top right ,c bottom right Software results

B. Hardware Results

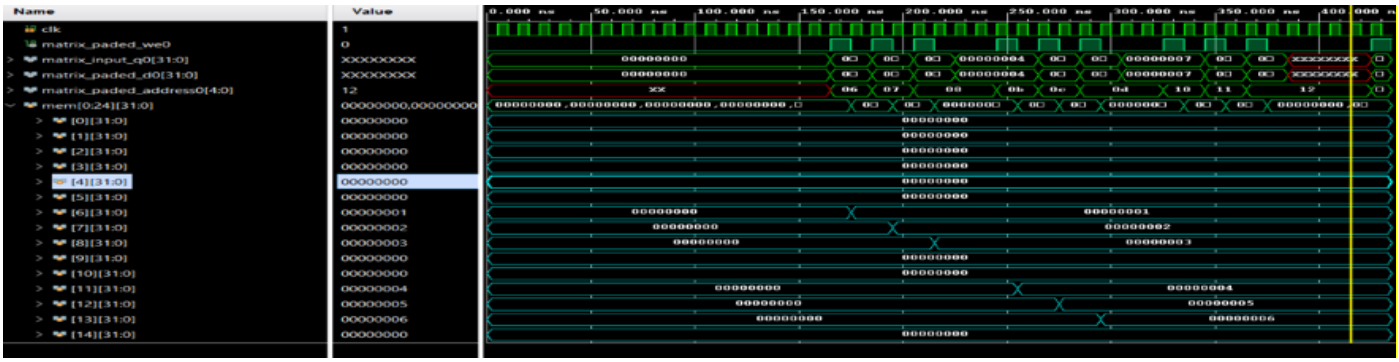


Figure 8 Padding wave form output

Figure 9 Convolution wave form output

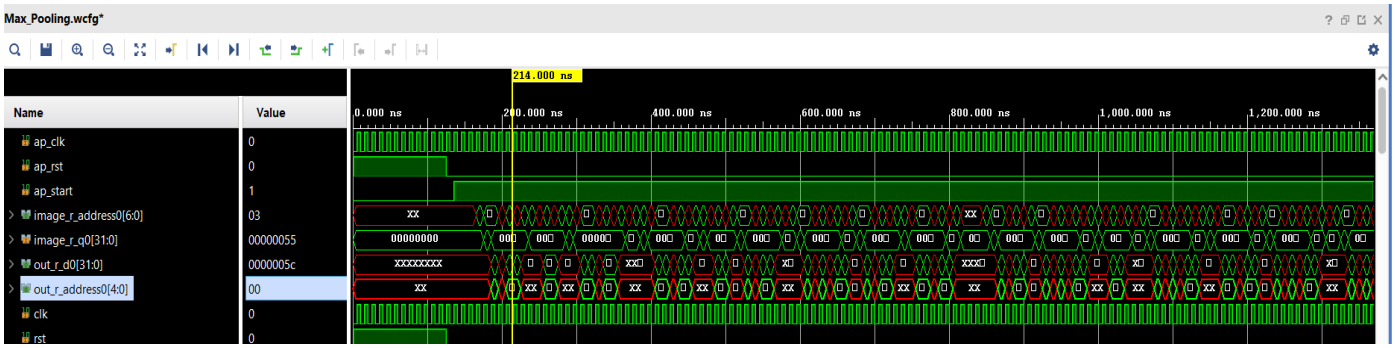
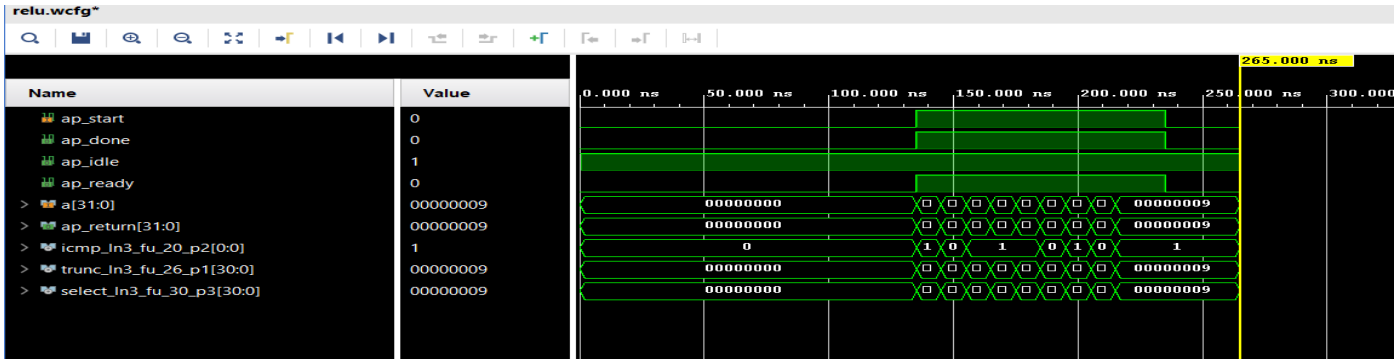


Figure 10 Max pooling wave form output.



.

Figure 11 RELU waveform output

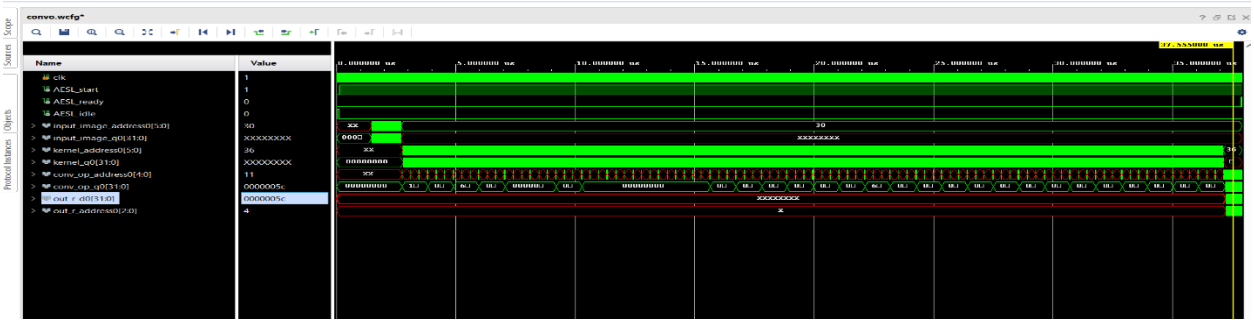


Figure 12 Integrated layer wave form output

.

.

1) Padding

By referring to the Figure 8 it shows that the function output is just as expected and after the required clock cycles the correct values are on the output. Furthermore, the output is being saved in the memory once they are ready on the output and before the memory address changes to the next one the true value should be ready to write in the memory. Tracing this waveform enabled us to ensure that the function worked correct.

2) Convolution.

After the padding function comes the convolution which do the most calculations and need the largest time and resources. Figure 9 shows the simulation of this function on hardware. The write signal is only high when it is correct value is being set on the output.

3) Max pooling.

The Max pooling function differs from the previous functions as it doesn't go through the feature maps row by row, but it uses a window 2x2 which is dealing with two different rows at a time. So, in the wave form the output may go high impedance at some points but the output array is still acting as intended (Figure 10).

4) RELU.

The RELU simulation was the easiest as it only converting the negative values to zero and leave the other values as it is.

5) Fully Connected.

The simulation shows that the 3d array of the output layer of convolution is flattened with the same values to a 1d array which can be passed to other layers

5) Integrated layer

After simulation of all these layers we needed to integrate them in one function to ensure that all of them are working smoothly with each other especially the interface between them .the simulation in figure 12 shows it takes a lot of time before the output is being ready and calculated that explains the high impedance value at the output lasting this long

C. Performance evaluation

1) Performance Metrics

Table 6 Latency of each Layer.

The proposed architecture	Latency(cycles)		Latency(absolute)	
	min	max	min	min
	869	869	8.690 us	8.690 us
Padding	79	79	0.790 us	0.790 us
Max pooling	206	206	206	206
RELU	0	0	0	0
Fully-Connected				
Integration of padding & convolution & max pooling	3575	3789	37.570 us	37.890 us

The metrics of performance in our design are the speed mainly of performing the convolution calculations as it gets much more time than other layers. Table 4 indicates the latency of different layers it shows clearly that convolution is the most time consuming. Noting that cascading layers together and increasing the latency time that must be taken in consideration as we are not using the real image size yet nor the exact number of filters in VGG16.

2) The utilization in FPGA

The utilization of the design when simulating it to know how much resources it will use from the available on the FPGA up till now the design is not complete, so we are using very small resources (table 5).

Table 7 Comparison between Utilization of each Layer.

	Total					Utilization (%)				
	BRAM_18K	DSP48E	FF	LUT	URAM	BRAM_18K	DSP48E	FF	LUT	URAM
Convolution	0	3	476	757	0	0	0	0	0	0
Padding	0	0	59	277	0	0	0	0	0	0
Max pooling	0	0	471	1541	0	0	0	0	0	0
RELU	0	0	0	49	0	0	0	0	0	0
Fully Connected										
Integration of padding, convolution and max pooling	1	3	944	2186	0	0	0	0	0	0

D. Comparison with related work

Table 8 Hardware accelerators

Hardware accelerator	Execution time (ms)(forward path)
Proposed architecture	40.94
Pascal titan x	5.32
GTX 1080	7

Table 9 Clock cycles

HW Function	Accelerated Clock cycles
Conv layer 3	$2.147 X10^9$
FC layer 6	$2.147 X10^9$
FC layer 7	$1.649X10^9$
FC layer 8	$0.4034 X10^9$

Table 10 Total LUT, DSP and FF

Conv layer1	LUT's	BRAMs 36k	DSPs	FFs
Optimized RTL	32,968	288	105	9210
SDSOC	19,260	477	95	11,037

1) Our work

Table 11 Clock cycles

Clock	Target	Estimated	Uncertainty
Ap_clk			

Table 12 Total DSP, FF and LUT

Name	BRAM 18K	DSP48E	FF	LUT	URAM
DSP					
Expression					
FIFO					
Instance					
Memory					
Multiplexer					
Register					
Total					
Available					
Utilization (%)					

2) Simple comparison.

The previous tables summarize the total clock cycles,total flip-flops, look up tables and digital signal processors of our work related work in [17].

E. Limitations

Due to implementation on hardware we are always limited with memory so we can't do larger architectures which has larger number of parameter which may have high accuracy, also we may need large number of hardware to implement any architecture which may not be available in commercial FPGAs

V. Conclusions and future work

CNNs are the main algorithm used in object detection due to its very high accuracy, That's why we try to increase the speed of detection and reduce its power consumption like the techniques discussed in the paper by using dedicated hardware (FPGA) for simulation instead of general purpose CPUs and GPUs in the future work we will download our design on an FPGA to test it and run Co-simulation between the hardware part and the software part

REFERENCES

[1] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. 2017. [DL] A Survey of FPGA-Based Neural Network Inference Accelerator

[2] J. Duarte et al., "Fast inference of deep neural networks in fpgas for particle physics," arXiv preprint arXiv: 1804.06913, 2018.

[3] J. Qiu et al., "Going deeper with embedded fpga platform for convolutional neural network," in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 26–35. [Online].

[4] Joseph Fiowa ,TensorFlow-object-detection-faster-rcnn,(2013),GitHub

[5] C. L. Zitnick and P. Dollar, "Edge boxes: Locating object ´ proposals from edges," in European Conference on Computer Vision (ECCV), 2014.

[6] Shaoqing Ren, PY-faster-rcnn) 2015), GitHub: https://github.com/rbgirshick/py-faster-rcnn

[7] K. He, X. Zhang, S. Ren, and J. Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In ECCV, 2014

[8] D. Baptista, F. Morgado-Dias and L. Sousa, "A Platform based on HLS to Implement a Generic CNN on an FPGA," 2019 International Conference in Engineering Applications (ICEA), Sao Miguel, Portugal, 2019, pp. 1-7, doi: 10.1109/CEAP.2019.8883473.

[9] S. Ghaffari and S. Sharifian, "FPGA-based convolutional neural network accelerator design using high level synthesize," 2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS), Tehran, 2016, pp. 1-6, doi: 10.1109/ICSPIS.2016.7869873.

[10] K. Simonyan and A. Zisserman, "Very Deep

Convolutional Networks for Large-Scale Image

[11] Rohit, VGG16-In-Keras (2018),GitHub: https://github.com//1297rohit/VGG16-In-Keras

[12] Yongmei Zhou and Jingfei Jiang, "An FPGA-based accelerator implementation for deep convolutional neural networks," 2015 4th International Conference on Computer Science and Network Technology (ICCSNT), Harbin, 2015,pp.829832,doi:10.1109/ICCSNT.2015.7490869.

[13] Hassan, R.O., Mostafa, H. Implementation of deep neural networks on FPGA-CPU platform using Xilinx SDSOC. *Analog Integr Circ Sig Process* (2020).

[14] L. D. LeCun, Y.; Boser, B.; Denker, J. S.;

Henderson, D.; Howard, R. E.; Hubbard, W.; Jackel,"Backpropagation Applied to Handwritten Zip CodeRecognition," Neural Computation, vol. 1, no. 4. Pp.541–551, 1989.

[15] Y. LeCun and L. Bottou, "Sn: A simulator for connectionist models," in Proceedings of NeuroNimes 88, 1988.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," Adv. Neural Inf. Process. Syst., pp. 1–9, 2012.

[17] Rania O. Hassan1 • Hassan Mostafa1, 2

"Implementation of deep neural networks on FPGA-CPU platform using Xilinx SDSOC"