Indian Institute of Technology, Bombay

VLSI DESIGN LAB

EE - 705

Detection using the YOLO hardware accelerator on PYNQ board

Name :	Roll Number :
Abhijeet	20D070003
A.Ananthapadmanabhan	20D070010
G Kamalesh	20D070029
Monu Kumar Yadav	20D070053

Professor : Sachin B Patkar

May 3, 2023

Contents

1 PROJECT PROPOSAL

Our endeavor involves implementing YOLOv2 on a PYNQ board for real-time object detection. Leveraging pre-defined weights from Darknet, we'll optimize the algorithm using PYNQ, Jupiter, Vivado, HLS, and Vitis. This hardware accelerator project aims to enhance object detection efficiency, utilizing FPGA-based acceleration to achieve low-latency processing.

2 Object classification:- YOLO

YOLO stands for You Look Only Once and is a network for object detection task consisting the determination of the location on the images where some sort of objects is present, as well classifying objects. YOLO offers a truly integrated solution with a simplified implementation of GPU. Darknet is also developed by the same team so it is quite easy to run the network using the configuration files used in Darknet. This neural network is very useful for the open source community as it is very difficult and hectic to implement a neural network on existing framework due to introduction of new algorithms which are not supported by the frameworks.

2.1 Architecture:

The main advantage of using YOLO is that it doesn't need another algorithm called RPN, which is popular but complex. RPN helps detect objects in things like self-driving cars and aiding disabled individuals. YOLO simplifies this process by handling both detection and classification with just one neural network. It comes in three versions: Regular, Small, and Tiny YOLO. There's also a newer, even smaller version called Fast YOLO, which has fewer layers compared to other models like R-FCN or Faster R-CNN

Specifications:

Tiny YOLO require almost 33.3 MB of space in order to process any image by assuming 32- bit is required by each parameter. The output results can be reproduced just by adding the parameters and calculating memory space for them. Tiny YOLO has 45 millions of weights parameters and size of these weights is almost 170MB. The pooling layers and activation layers does not have any parameters. So they will be calculated during the calculation of memory space.

The tiny/Fast YOLO network model is made uo of some basic layers and they are following:

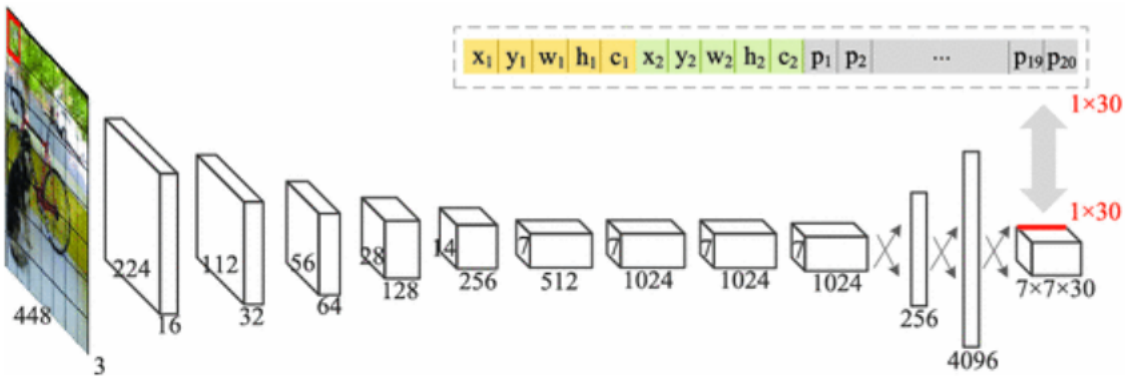

Figure 1: YOLO Architecture

- Convolutional Layer
- Max Pooling Layer
- ReLU Activation Layer
- Fully Connected Layer

3 Dataset

COCO:
Common Object in Context (COCO) dataset is a large scale object detection, segmentation and captioning dataset with several features. For example, object segmentation, recognition in context, 80 object categories, 5 captions per image, labeling etc. the name contains a word context because images in this dataset are taken from daily life scenes and provides contextual information. This dataset is a bunch of 80000 training images and 40000 validation images. This dataset is explored in order to use it with YOLO network as this network is initially trained with COCO dataset.

4 PYNQZ2 Board

Figure 2: PYNQZ2 Board

5 Software Simulation

In our endeavor to optimize YOLOv2 for FPGA-based acceleration, we adopt a comprehensive strategy aimed at enhancing efficiency. We begin by sourcing essential elements from the Darknet framework and YOLOv2 weights, setting the stage for customization. Next, we tailor Darknet's weight loading function to integrate batch normalization optimizations, while also addressing the resource-intensive operations inherent in FPGA logic through precision optimization techniques.

- Downloaded the Darknet source code from the official repository and procure the YOLOv2 weights from the designated source
- Adapted Darknet's weight (height from 608 to 416) loading function to support customized weights and biases, facilitating integration with combined batch normalization optimizations
- Quantize input/output feature maps, weights, and biases to dynamic fixed-16 precision to reduce resource utilization. Utilize fixed-16 operations to replace resource-intensive float-32 operations, aligning with established research recommendations.

6 HLS Accelerator and Simulation

HLS (High-Level Synthesis) is a methodology used in digital design for converting high-level programming languages (such as C, C++, or OpenCL) into hardware description languages (HDLs) like Verilog or VHDL. This allows hardware designs to be described at a higher level of abstraction than traditional RTL (Register-Transfer Level) design,

making it easier for software developers to design and optimize hardware accelerators.

6.1 Procedure

- We procured a C++ implementation of the YOLOv2 model from and created a testbench to verify the verilog code synthesized by **Vitis HLS** by inputting some sample images.
- The weights and biases for the pretrained model are extracted from the darknet repo and are converted into fixed point 16 bit quantised values.
- The top level function is set and the design is synthesised
- The synthesized design is tested by passing some input images and comparing the predictions with that of the original C++ model.

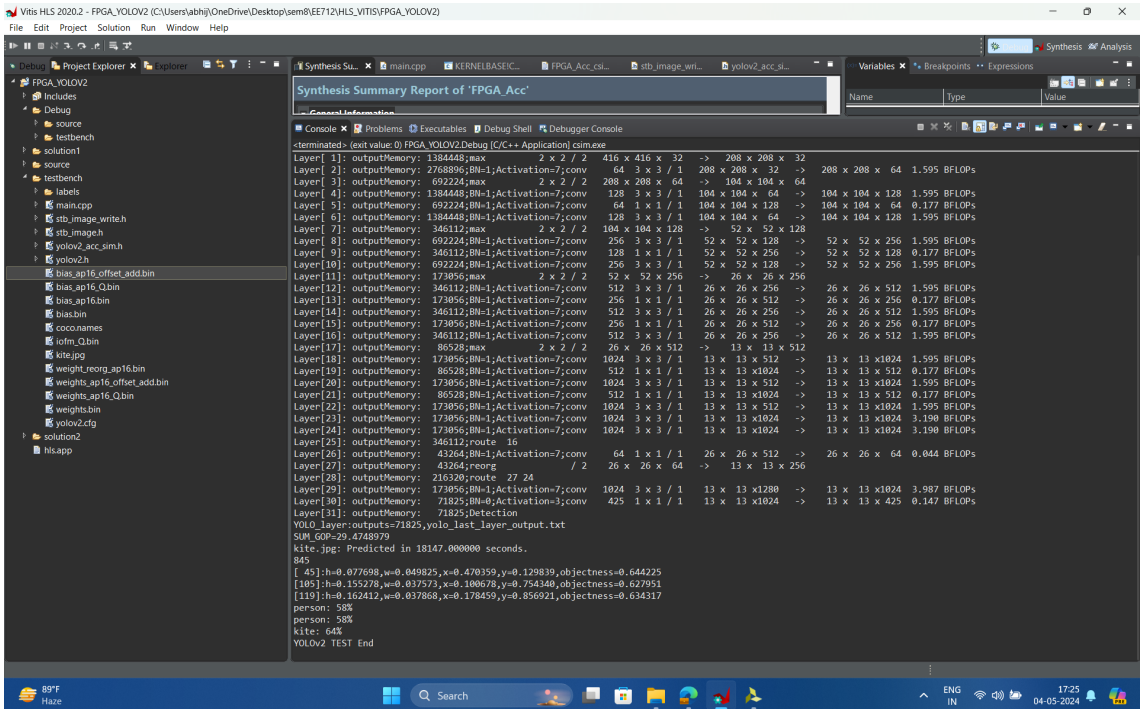

Figure 3: HLS synthesis and Testbench results

7 Vivado Block Design

- First imported the IP created using the HLS vitis.
- We create a module called *clock_wizard*. I configure its output clock to be 150MHz, and I specify that it operates on a set-reset type with an active low signal.
- We connected the ip as follows

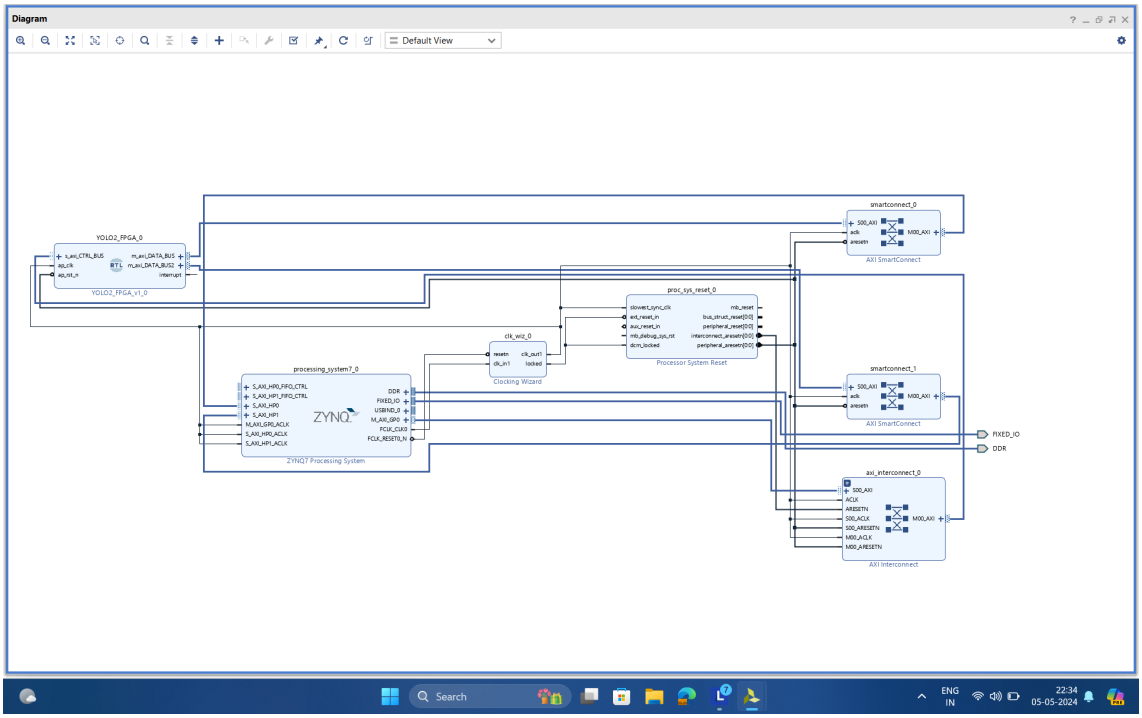

Figure 4: Vivado Block Diagram

- Then we validated the layout and then we Run synthesis and Run Implementation and finally generated the bit-stream
- We Used the yolo.bit and yolo.hwh for pynq

8 PYNQ Results

We implemented the real time processing using- rr

```
from PIL import Image as PIL_Image
import time
from IPython.display import display
orig_img_path = 'common/data/webcam.jpg'
# !fswebcam -d -v -S 10 --no-banner --save {orig_img_path}  --set
    brightness=100%  /dev/video0 2> /dev/null
# while True:
!fswebcam -d /dev/video0 2> /dev/null -S 30 --set brightness=50% --
    save {orig_img_path}
img = PIL_Image.open(orig_img_path)
img
import cv2
import numpy as np
opencv_image = cv2.cvtColor(np.array(img), cv2.COLOR_RGB2BGR)
display(PIL_Image.fromarray(opencv_image))
# time.sleep(20)
```

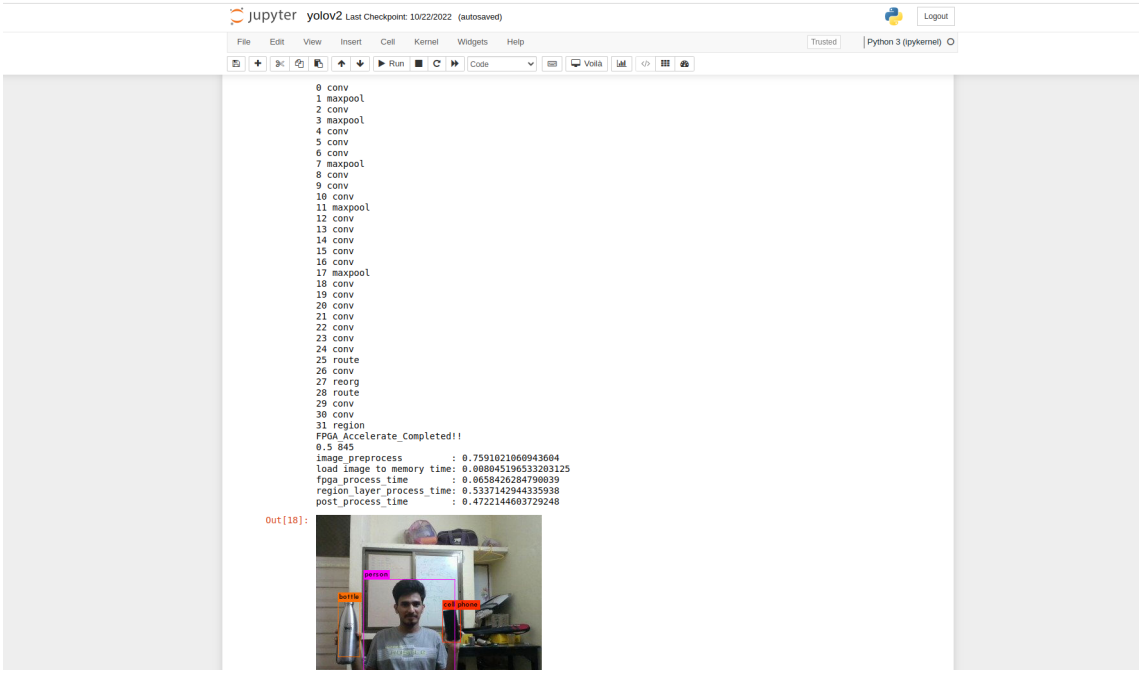

Figure 5: Layers

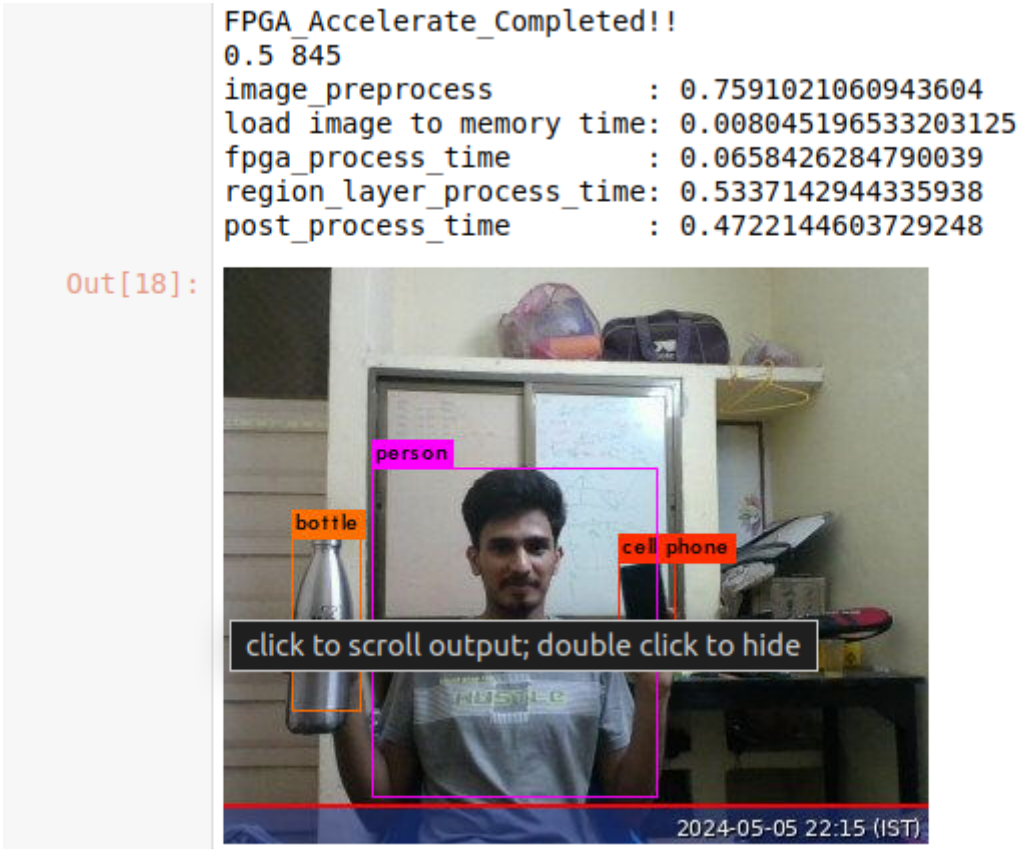

Figure 6: Real Time Image Prediction

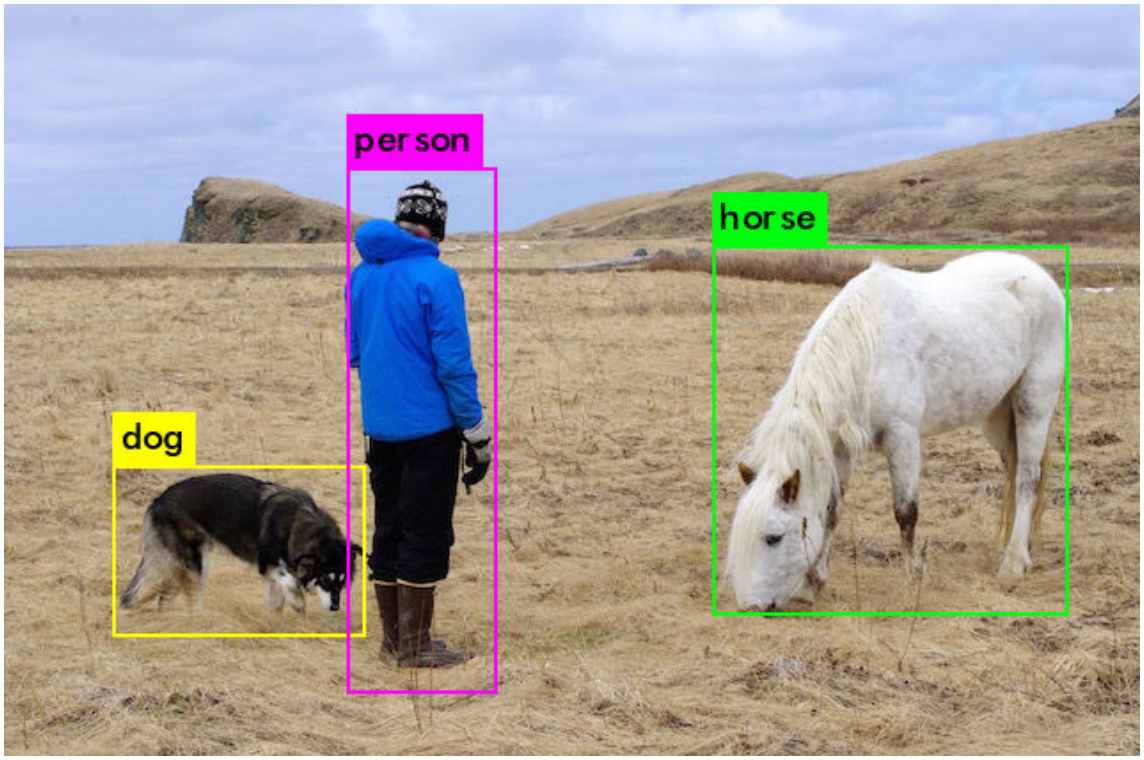

Figure 7: Person Prediction

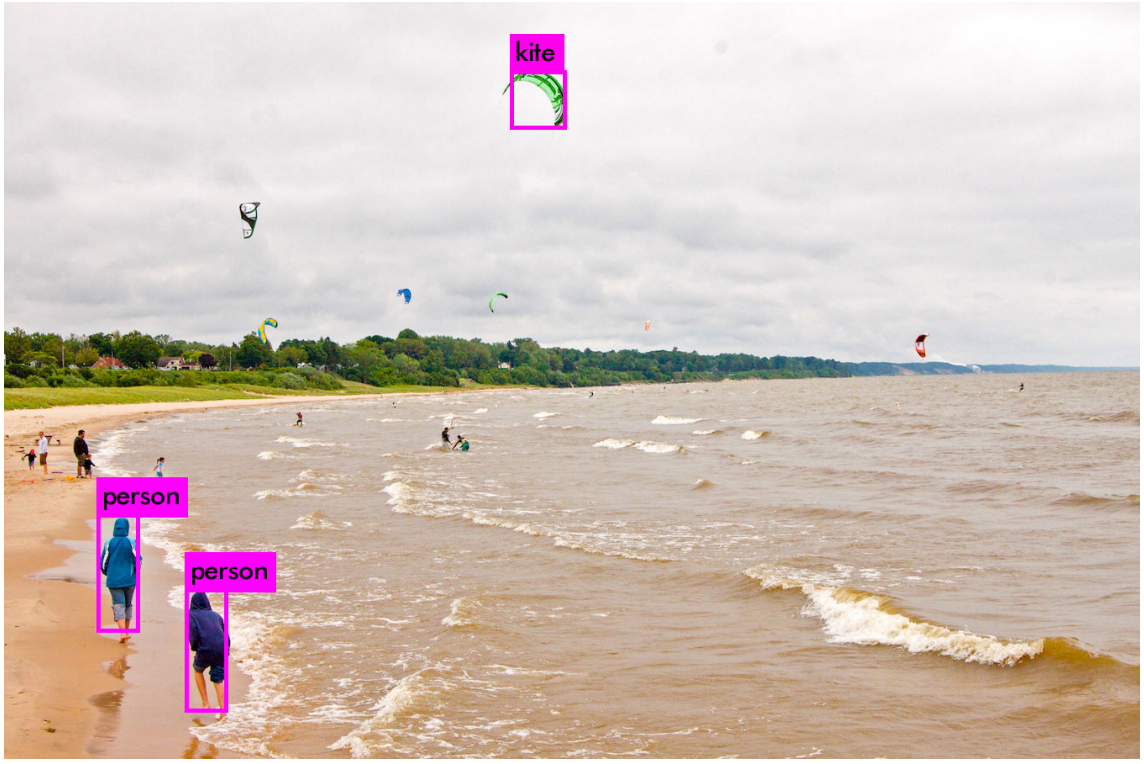

Figure 8: Object Prediction

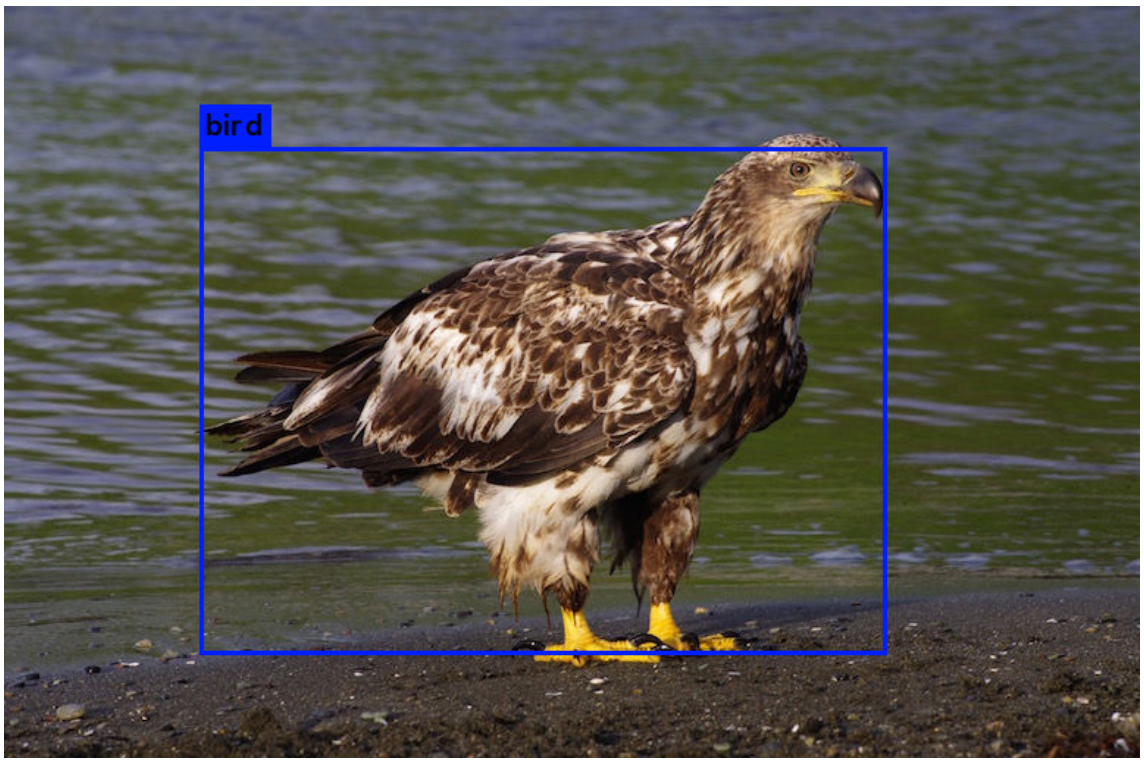

Figure 9: Object Prediction

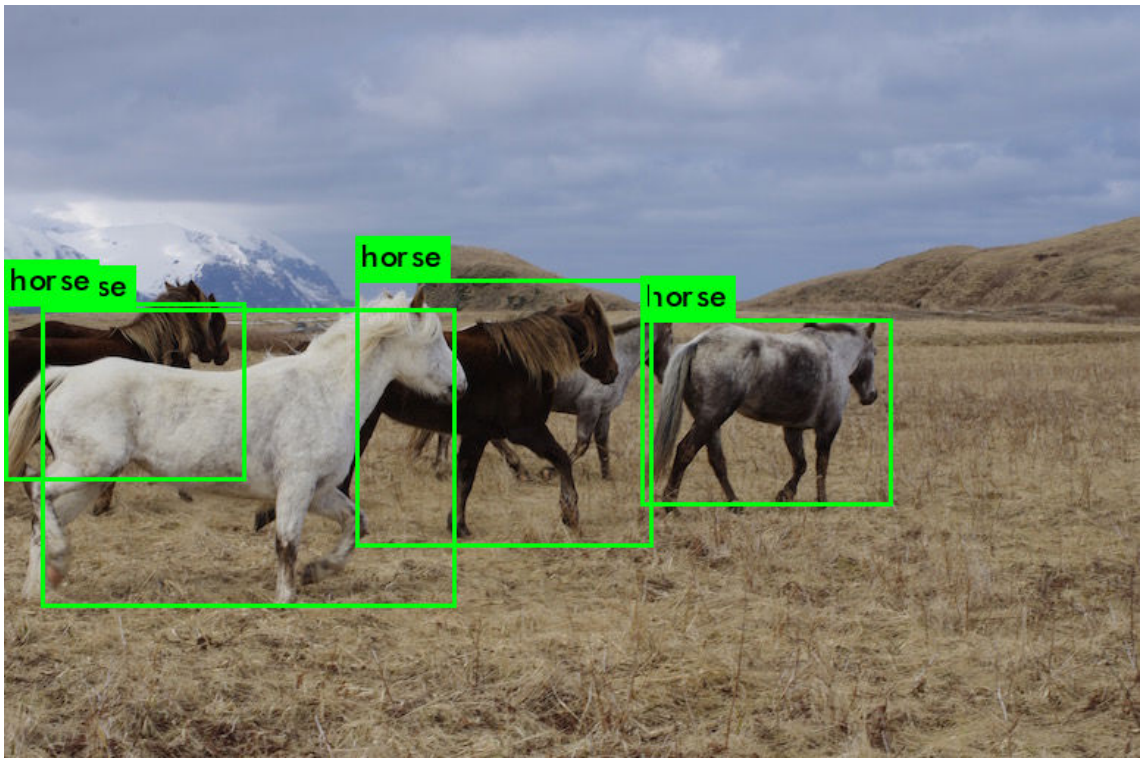

Figure 10: Horses Prediction

9 Individual Contribution

- Initial ideas & execution planning – everyone
- Weigths and biases from DarkNet - Monu & Ananthapadmanabhan
- Test Bench from HLS and debug - everyone
- Vivado IP Creation, and Design - Abhijeet & Kamalesh
- Pyhton code in Jupiter for Real Time Processing - everyone
- Report writting - everyone

10 Conclusion

In this hardware accelerator project, we successfully implemented YOLOv2 on a PYNQ board for real-time object detection, leveraging FPGA-based acceleration to achieve low-latency processing. Through the integration of various technologies such as Darknet, PYNQ, Vivado, HLS, and Vitis, we optimized the YOLOv2 algorithm for efficient object detection.

Utilizing the YOLO architecture eliminated the need for additional complex algorithms like RPN, streamlining the detection and classification process into a single neural network. This not only simplified the implementation but also enhanced the efficiency of object detection. Moreover, by employing the Tiny YOLO model, we minimized resource requirements while maintaining performance, making it suitable for deployment on FPGA-based platforms.

For training and validation, we utilized the COCO dataset, which provided a diverse range of real-world images for testing the accuracy and robustness of our implementation. Through software simulation and hardware synthesis using HLS, we optimized the YOLOv2 model for FPGA-based acceleration, achieving high throughput and low latency.

In the Vivado Block Design phase, we successfully integrated the synthesized IP cores and generated the bitstream for deployment on the PYNQ board. The results obtained from the PYNQ board demonstrated real-time object detection capabilities, as evidenced by the layers visualization and real-time object detection images.

Overall, this project showcases the effectiveness of FPGA-based acceleration in enhancing the efficiency of deep learning algorithms for real-time applications such as object detection. By leveraging a comprehensive strategy combining software and hardware optimizations, we have successfully implemented YOLOv2 on a PYNQ board, paving the way for future advancements in FPGA-based deep learning accelerators