UNIVERSIDADE D
COIMBRA

José Pedro Araújo Azevedo

YOLO Neural Networks on Reconfigurable Logic for vehicle traffic control

Dissertation Submited in Partial Fulfillment for the Degree of Master in Electrical and Computer Engineering Supervised by Professor Jorge Nuno de Almeida e Sousa Almada Lobo and presented to the Department of Electrical and Computer Engineering of Faculty of Science and Technology of the University of Coimbra.

This dissertation work was partially carried out in a business Environment, under and agreement with SNPS(Synopsys) Portugal, with Doctor Pedro Moreira as advisor on the company side.

September 2022

UNIVERSIDADE Đ
COIMBRA

José Pedro Araújo Azevedo

YOLO NEURAL NETWORKS ON RECONFIGURABLE LOGIC FOR VEHICLE TRAFFIC CONTROL

Dissertation submitted to the
University of Coimbra for the degree of
Master in Electrical and Computer Engineering

Supervisors:

Prof. Dr. Jorge Nuno de Almeida e Sousa Almada Lobo
Dr. Pedro Moreira

Jury:

Prof. Dr. Gabriel Falcão Paiva Fernandes

Vowels:

Prof. Dr. Jorge Nuno de Almeida e Sousa Almada Lobo
Prof. Dr. Mário João Simões Ferreira dos Santos

Coimbra, September of 2022

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Dr. Jorge Nuno de Almeida e Sousa Almada Lobo for accepting the proposed topic and providing all the necessary material.

I also thank the Institute of Systems and Robotics (ISR) for the excellent workspace provided.

To Dr. Pedro Moreira, R&D Manager at Synopsys, for accepting this collaboration.

A special thank you to two very close friends for all the coffee breaks and memorable moments. Rafael Vieira, who assisted me and gave me advice on how to resolve certain issues, was a great help. Nuno Mendes contributed to the development of a system for measuring electric current.

To my family and friends who made this possible, my sincere "Thank You".

UNIVERSIDADE D
COIMBRA

INSTITUTE OF SYSTEMS AND ROBOTICS
UNIVERSITY OF COIMBRA

SYNOPSYS®

Abstract

Object detection is a crucial task that has multiple applications, including autonomous vehicles, security, agriculture and many applications in manufacturing and smart cities. The majority of these applications use IoT devices, the problem with these *edge devices* is their limited computing power and limited energy resources. As a result, a significant percentage of their computing is performed in the *cloud*, and as a result, there is always a delay in these solutions, and many of them consume a significant amount of computational power, resulting in high energy consumption. To improve this aspect, it is necessary to perform some computing locally, providing fast, low-cost and energy-efficient solutions.
The proposed solution focuses on implementing YOLOv3,v4,v5 Neural Networks (NN) in Graphics Processing Unit (GPU)s and Central Processing Unit (CPU)s (simulating *cloud* computing) and on Field Programable Gate Array (FPGA)s (simulating *edge* computing). The implementations were tested using the Common Objects in Context (COCO) dataset, involving a total of 14 NN models. In these implementations, energy consumption was measured, allowing for precise comparisons of the most energy-efficient models across different types of hardware. These implementations aim to achieve a balance between accuracy and energy efficiency by taking into account computationally lightweight solutions. To perform inference on the FPGA, two frameworks, Pynq[1] and Vitis-Ai[2], were used. Due to its complexity, parallelization was not performed in the Pynq framework, however, it was possible to achieve 1.23 Frames Per Second (FPS) while achieving 0.09 *frames/joule*. Parallelization within the Vitis-AI framework allowed for the achievement of 57.93 FPS at 2.25 frames per joule.
Finally, object detection models that operate in real time (more than 30 FPS on GPU and FPGA) have been developed, achieving 2.25 *frames/joule* on FPGA implementations and 1.31 *frames/joule* on GPU implementations, respectively.
Comparing the most efficient FPGA implementation to the most efficient GPU implementation, the FPGA is 1.78 times more efficient.

Keywords: FPGA, IoT, YOLO, Object Detection, Edge Computing.

Resumo

A detecção de objetos é uma tarefa crucial que possui várias aplicações, incluindo veículos autônomos, segurança e inúmeras aplicações em fábricas e cidades inteligentes. A maioria destas aplicações usa dispositivos IoT, o problema com textitedge devices é seu possuirem baixo poder computacional e serem limitados pelo baixo consumo de energias. Como resultado, uma percentagem significativa de sua computação é realizada na *cloud* e, deste modo, há sempre um atraso nestas soluções, e a maioria delas utiliza grande poder computacional, promovendo alto consumo de energia. Para melhorar esse aspecto, é necessário realizar alguma computação localmente, fornecendo soluções rápidas, de baixo custo e energeticamente eficientes.
A solução proposta envolve a implementação de Redes Neuronais Neural Networks (NN) YOLOv3,v4,v5 em Graphics Processing Unit (GPU)s e Central Processing Unit (CPU)s (simulando *cloud computing*) e em Field Programable Gate Array (FPGA)s (simulando *edge computing*). As implementações foram testadas utilizando o *dataset* Common Objects in Context (COCO), envolvendo um total de 14 modelos de redes neuronais. Nestas implementações, o consumo de energia foi medido, permitindo comparações precisas dos modelos mais eficientes em termos de energia nos diferentes tipos de hardware. Estas implementações visam alcançar um equilíbrio entre precisão e eficiência energética, levando em consideração soluções computacionalmente leves.
Para realizar a inferência na FPGA, foram utilizadas duas frameworks, Pynq[1] e Vitis-Ai[2]. Devido à sua complexidade, a paralelização não foi realizada na framework Pynq, no entanto, foi possível atingir 1,23 Frames Per Second (FPS) e atingir 0,09 *frames/joule*. A paralelização na framework Vitis-AI permitiu atingir 57,93 FPS e 2.25 *frames/joule*. Por fim, foram desenvolvidos modelos de detecção de objetos que operam em tempo real (mais de 30 FPS em GPU e FPGA), alcançando 2,25 *frames/joule* em implementações na FPGA e 1.31 *frames/joule* em implementações de GPU, respectivamente.
Comparando as implementações mais eficientes em FPGA e GPU, a FPGA é 1.78 vezes mais eficiente.

Palavras-chave: FPGA, IoT, YOLO, Deteção de objetos.

Contents

List of Acronyms **ix**

1 Introduction **1**
1.1 Motivation and Scope 1
1.2 Objectives 2
1.3 Related Work 2
1.4 Key Contributions 5
1.5 Structure of the Dissertation Project 5

2 Background and state-of-the-art **7**
2.1 Convolutional Neural Networks 7
2.1.1 Convolutional Layers 8
2.1.2 Pooling Layers 8
2.1.3 Fully Connected Layers 9
2.1.4 Dropout 9
2.1.5 Activation Functions 10
2.2 YOLO architectures 10
2.2.1 YOLOv3 11
2.2.2 YOLOv4 12
2.2.3 YOLOv5 13
2.2.4 Main Diferences between YOLO, R-CNN and SSD architetures . 14
2.3 Summary/Conclusions 14

3 YOLO Implementations, frameworks and tools used **17**
3.1 Hardware, software and tools used 17
3.1.1 Methods for measuring power consumption. 17
3.2 Implementation Notes 21
3.3 CPU and GPU implementation 21
3.4 FPGA implementation 21
3.4.1 Using Pynq framework 21
3.4.2 Using Vitis-AI framework 24
3.4.2.1 DPUCZDX8G Overview 26
3.4.2.2 Notes on the different environments and examples followed 31
3.4.2.3 YOLOv3, v4, v5 implementation - Tensorflow and Pytorch 31
3.5 Summary/Conclusions 33

4 Results and analysis **37**
4.1 Raw performance 37
4.1.1 CPU and GPU results 37
4.1.2 FPGA results (Pynq framework) 38
4.1.3 FPGA results (Vitis-AI framework) 39
4.2 Comparison between hardware tested 41
4.3 Draw conclusions concerning Neural Networks YOLOv3, v4, v5 41
4.4 Discussion and notes 43

5 Conclusions and Future Work **45**
5.1 Conclusion 45
5.2 Future Work 48

Appendix A List of YOLOv3/v5 environment packages (Desktop) **57**

Appendix B List of YOLOv3/v5 environment packages Pynq **63**

Appendix C Power measurements on CPU/GPU **69**

Appendix D Power measurements on Pynq framework **75**

Appendix E Power measurements on Vitis-AI framework **79**

List of Acronyms

AI Artificial Intelligence.

ANN Artificial Neural Networks.

CNN Convolutional Neural Networks.

COCO Common Objects in Context.

CPU Central Processing Unit.

CSPNet Cross Stage Partitial Network.

DHM Direct Hardware Mapping.

DL Deep Learning.

DNN Deep Neural Learning.

DPU Deep Learning Processor Unit.

DSP Digital Signal Processing.

FC Fully Connected.

FPGA Field Programable Gate Array.

FPN Feature Pyramid Network.

FPS Frames Per Second.

GPU Graphics Processing Unit.

HLS High Level Synthesis.

ICP Input Channel Parallelism.

IoT Internet of Things.

IP Intellectual Property.

LUT Look Up Table.

mAP mean Average Precison.

ML Machine Learning.

NN Neural Networks.

OCP Output Channel Parallelism.

OFA Once For All.

PAN Path Aggregation Network.

PP Pixel Parallelism.

R-CNN Region-based Convolutional Neural Network.

RAM Random-Access Memory.

RCAN Residual Channel Attention Networks.

RL Reinforcement Learning.

RNN Recurrent Neural Networks.

SoC System on a Chip.

SPP Spatial Pyramid Pooling.

SSD Single-Shot Detector.

YOLO You Only Look Once.

Chapter 1

Introduction

1.1 Motivation and Scope

Object detection [3][4] is a method in computer vision [5] that determines if an object is present or not in an image [6]. Examples include recognizing objects with autonomous vehicles, animal detection in agriculture, and people detection in security.

In smart cities there are also numerous applications incorporating object detection. Traffic management is one of them. In Los Angeles, for instance, an intelligent transport solution was implemented to monitor and control traffic flow [7]. Integrated pavement sensor transmit real-time updates of traffic flow to a central traffic management platform, which analyzes the data and instantly adjusts traffic lights to the traffic situation. Simultaneously, historical information is used to predict where traffic can flow, and none of these processes require human intervention [8].

In recent years, Deep Learning (DL) has been widely implemented across a variety of fields. Convolutional Neural Networks (CNN) provide the most accurate solutions to real-world problems in DL [9]. CNN are used in a wide range of everyday applications, including voice recognition, document analysis, face recognition and estimation of human pose [9]. These Neural Networks (NN) make use of Convolutional layers, these extract features from the input data, followed by Classification layers that make decisions [10]. One of the problems found in these NNs is their high computational weight, since they usually require billions of operations in order to process an input.

Generally, the computing power of IoT devices is limited, which makes it challenging to implement deep learning applications on these devices. Field Programable Gate Array (FPGA)s are excellent candidates for this type of problem due to their high processing power, low power consumption, and low latency. Also FPGAs have an inherently parallel architecture that makes them suitable for ML applications [11] [12]. FPGAs circuits are capable of being reconfigured at any time.

More than 55% of the world's population is now urbanized [13] (68% by 2050 is projected [14]). Problems related to health, traffic, pollution, waste management and poor infrastructure arise and hence development of city falls apart [13]. This has prompted the use of technology as a solution for all of these problems and to address them in a more intelligent manner. Hence the Smart Cities concept. With the help of Big Data and

IoT, smart cities ensure a sustainable environment. According to [15], the AI industry applied in smart cities will be worth 190$ billion by 2025. With this massive increase in IoT devices, the junction between IoT and AI is expected. It can be said that this has numerous advantages. For example, bringing the power of AI to IoT, reducing data center operation and maintenance costs.

1.2 Objectives

The objective of this work is to have low-power object detection implementations. For this, we will use the current You Only Look Once (YOLO) architectures (versions: 3, 4 and 5). These Neural Networks will be compared in terms of energy performance (*frames/joule*) while taking their mAP into account. Additionally, conclusions will be drawn on the speed of inference and the detection of multiple objects.

These implementations will be focused on heterogeneous platforms that combine low power consumption, CPU, GPU and FPGA. With these three solutions, we will analyze in detail the most energy efficient solution for object detection.

The implementation in FPGA, will not be carried out from scratch, taking into account the complexity of these Neural Networks. Frameworks suck as Pynq [1] and Vitis-AI [2] that facilitate the deployment of these NN will be addressed.

The steps involved in this work are briefly described in the Figure 1.1.

Figure 1.1: Example of *YOLOv5s* network detection output for an image (left). Diagram of the steps performed (right).

1.3 Related Work

Some works related to this theme will be presented to contextualize this work with the state of the art.

The work done in [16] explains all of the benefits that edge computing can bring. Data processing at the edge can result in faster response times and higher reliability.

Furthermore, bandwidth could be saved if more data could be handled at the edge rather than uploaded to the cloud.

Given the low processing power in IoT devices, new methods have been developed and are being developed to process some of the data locally.

The work presented in [17] proposes a strategy for processing CNN in IoT devices, using the streaming hardware accelerator. This improves energy efficiency by avoiding unnecessary data movement. This accelerator can support the most popular CNN and achieve 434 GOPS/W energy efficiency, which makes it ideal for integration with IoT devices. However, the NN in question (LeNet-5 [18]) is not complex enough to detect multiple objects.

The work described in [6] detects objects using the YOLOv3-tiny NN, which is capable of 7 to 14 Frames Per Second (FPS) with low cost FPGAs. This Neural Networks was modified with 16 - and 8 - bit quantizations, achieving mAP_{50} values of 31.5 and 30.8, respectively, using COCO [19] dataset.

The YOLOv3, YOLOv4, and YOLOv5 NN were compared in [20] for the purpose of Autonomous Landing Spot Detection. As a result, although all of these Neural Networks served the intended purpose, they considered the YOLOv5-l NN to be the best. Furthermore, the differences in Neural Networks architectures are succinctly summarized in this article, which includes comparisons with other articles involving different datasets, image resolutions, and other algorithms in addition to YOLO.

Graphics Processing Unit (GPU)s are currently the dominant programmable architecture for DL accelerators, but as previously stated, more energy-efficient solutions are being sought. As a result, the work in [21] involves the use of a heterogeneous acceleration method (FPGAs + GPU) that outperforms GPU acceleration. They show that Direct Hardware Mapping (DHM) of a CNNs on an embedded FPGA outperforms a GPU implementation in terms of energy efficiency (about a 25% reduction in energy) and execution time (21% latency reduction). A direct comparison of FPGAs and GPUs using DNNs was performed by [22], and the results show that the FPGA in question (Stratix 10) can achieve more GOP/s/Watt than the GPU (Nivida Titan X), about 2.3x to 4.3x speedups.

The framework mentioned in the 1.2, Pynq, was the subject of research and implementation in [23], with one of the implementations using the dataset Cifar10 [24] and a simple NN Lenet-5 [18], as detailed in [25], obtaining a precision of 75.2%. The results obtained with this framework were approximately 42 times faster (compared to CPU implementations), using only 2.063 Watts per 132 images.
Using various accelerators such as Vitis-AI [2] (which makes use of the Deep Learning Processor Unit (DPU) present in some FPGAs) and Finn [26]. The work done in [27] included the creation of a network called LittleNet as well as another YOLOFINN. The implementation in Vitis-AI proved useful consumming arround 3 watts for both Neural Networks, with 123.3 FPS for the LittleNet network and 53.2 FPS for the YOLOFINN NN.

Although the implementation of these networks from scratch using High Level Synthesis

(HLS) would be advantageous from an energy standpoint, this option requires extensive and complex work, so the approach with frameworks that facilitate this entire process is more adequate for the objectives of this dissertation. It should be noted that these frameworks (Pynq and Vitis-AI) are obviously limited. However, given the complexities of YOLO Neural Networks and the material we have access to, they are an excellent starting point for this work. (discussed in 3.1).

Given these works, there is limited information on the implementation of these Neural Networks in the FPGA, however, the existing information does not detail the results that this type of implementation achieves in terms of energy consumption and the measurement process is not fully explained.

1.4 Key Contributions

This work makes the following key contributions:

- Comparison between several YOLO Neural Networks (v3, v4 and v5) using the COCO [19] dataset.
- CPU, GPU and FPGA Pynq framework implementations of YOLO Neural Networks(versions: v3, v3-tiny, v5l, v5s).
- Vitis-AI framework implementations of YOLOv3-tiny and YOLOv5-small Neural Networks.
- Benchmark in terms of performance FPS and energy consumption(kWh and *frames/joule*), taking into account the mean Average Precison (mAP) of the Neural Networks, and comparing to CPU, GPU, and FPGA (Pynq and Vitis-AI) implementations.
- It was proved that performing an inference in reconfigurable logic requires less energy. (The YOLOv3-tiny NN, for instance, spent 6.34×10^{-4} kWh on the GPU and 3.70 kWh on the FPGA.)
- Evaluation of performance (FPS) and energy consumption of the YOLO Neural Networks present in Model Zoo [28].

1.5 Structure of the Dissertation Project

This dissertation is structured as follows:

- Chapter 1 presents the motivation of this work, the main objectives, the related work and contributions provided to the scientific community of this dissertation.
- Chapter 2 provides a concise description of how Convolutional Neural Networks function, as well as distinctions between YOLOv3,v4,v5 architectures.
- Chapter 3 presents the three types of hardware to be tested, the tools used to evaluate the energy consumption and the frameworks used for the implementation in reconfigurable logic.
- The results and analysis, comparisons between hardware are present in Chapter 4.
- Some final considerations are presented in Chapter 5.

Chapter 2

Background and state-of-the-art

The sections that follow explain how traditional Convolutional Neural Networks (CNN) work and some related concepts. A brief introduction to YOLO networks, differences between these networks, and comparisons with other algorithms (Single-Shot Detector (SSD) [29] and faster Region-based Convolutional Neural Network (R-CNN) [30]).

2.1 Convolutional Neural Networks

Neural Networks (NN) teach computers to do what humans do naturally. Deep Learning (DL) models include Artificial Neural Networks (ANN), Recurrent Neural Networks (RNN) and Reinforcement Learning (RL). However, one model in particular has made major contributions to the field of computer vision and image analysis, and that is the Convolutional Neural Networks.
Convolutional Neural Networks are members of a DNN class that can recognize and classify features from from tensors (A tensor is a multidimensional array in which data is stored). Image classification, medical image analysis, language processing, and autonomous or self-driving vehicles are just a few of the many applications of CNN.
Yann LeCun [25] proposed a CNN in 1989 to recognize handwritten characters in zip codes. After several successful iterations, the same scientist presented the famous network Le-Net5 [18].
Alex Krizhevsky [31] presented the AlexNet NN in 2012, which was a deeper and much wider version of the LeNet and won the difficult ImageNet [32] competition by a large margin.

Basic CNN Architeture

Some key components of these NN are as follows:

- Feature extraction - Convolutional/pooling layers that separate and identifies the various features of the image.
- Classification - Fuly connected layer that uses the feature extraction process output to predict the image class.
- As shown in the figure 2.1 architecture diagram, there are numerous CNN layers.

A CNN architecture is built using three types of layers: 1-convolutional layers, 2-pooling layers, and 3-Fully Connected (FC) layers. These will be summarized and defined

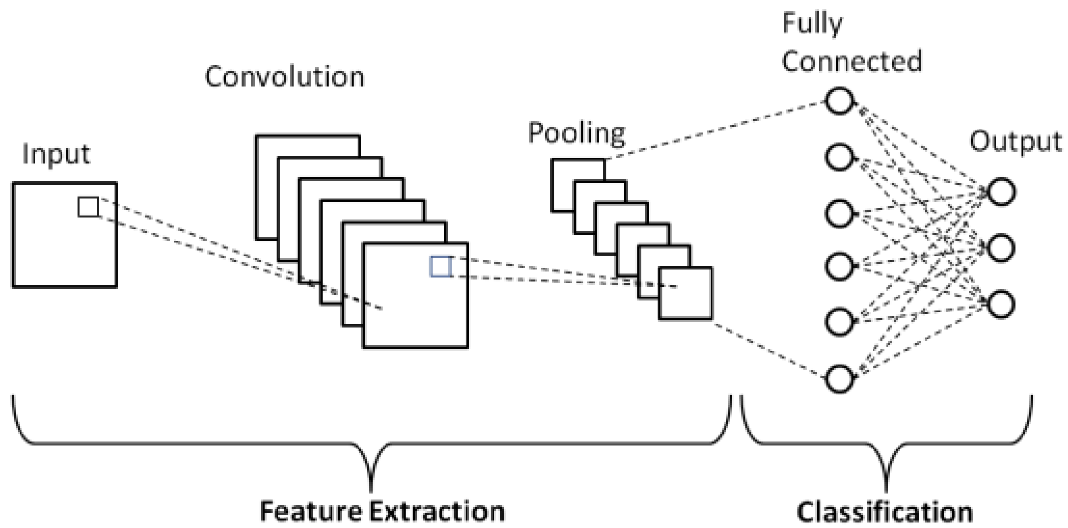

Figure 2.1: CNN architecture diagram example [33].

in subsections: 2.1.1, 2.1.2, 2.1.3.
There are also two important parameters not shown in Figure 2.1, dropout layer and activation function layer, these will be explained in 2.1.4 and 2.1.5.

2.1.1 Convolutional Layers

These are the first layers, wich are used to extract various images features. Convolution is a mathematical operation performed between an input image and a filter of size M x M. The dot product between the filter and the parts of the input image with respect to the size of the filter is calculated by sliding the filter over the input image (M x M). Figure 2.2 shows an example of these calculations.

The resulting feature map contains information regarding the image, including its corners and edges. This feature map is then fed to other layers, which learn additional features from the input image.
When the convolution operation is applied to the input, the convolution layer in CNN passes the result to the next layer. The spatial relationship between pixels is preserved by convolutional layers, which is a great benefit.

2.1.2 Pooling Layers

A pooling layer typically follows the convolutional layer. It seeks to reduce the feature map's size and reduce computational costs. Depending on the type of pooling used, it summarizes the features produced by the convolutional layer.
The largest element from the feature map is used in Max Pooling. Average Pooling computes the average of the elements in a predefined image section size. Sum Pooling computes the total sum of the elements in the predefined section. The Pooling Layer is typically used to connect the Convolutional Layer and the Fully connected Layer.

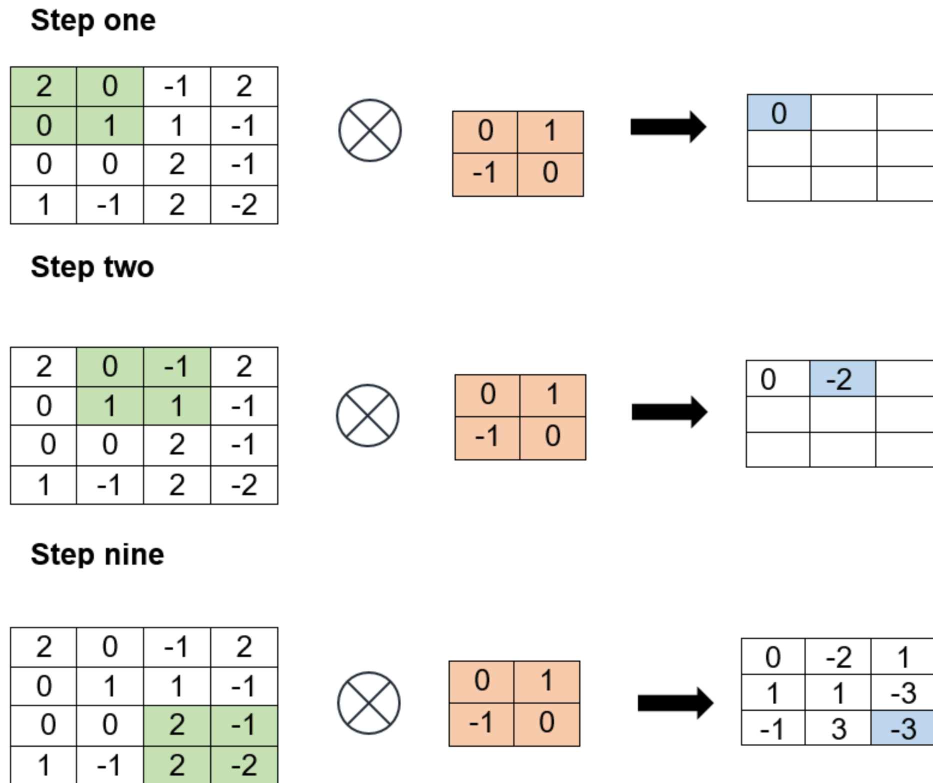

Figure 2.2: Convolutional layer calculations example.

2.1.3 Fully Connected Layers

This layer is typically located at the end of every CNN architecture. Each neuron in this layer is connected to all neurons in the previous layer, an approach known as Fully Connected (FC). It is used as Convolutional Neural Networks classifier. As a type of feed-forward Artificial Neural Networks (ANN), it follows the fundamental method of the conventional multiple-layer perceptron NN [34]. The previous pooling or convolutional layer serves as the input to the FC layer. This input is in the form of a vector, which is created by fattening the feature maps. As shown in Figure 2.3, the output of the FC layer represents the final CNN output [35]. Connecting two layers is necessary because two FC layers outperform one connected layer [36].

2.1.4 Dropout

Overfitting in the training dataset is common when all features are connected to the FC layer. Overfitting occurs when a model performs so well on training data that it has a negative impact on the model's performance when applied to new data [35].
To address this issue, a dropout layer is implemented, in which a small number of neurons are removed from the neural network during training, resulting in a smaller model. When a dropout of 0.4 is reached, 40% of the Neural Networks nodes are randomly removed [36]. Dropout improves the performance of a ML model by preventing overfitting by simplifying the network. During training, it removes neurons from Neural Networks.

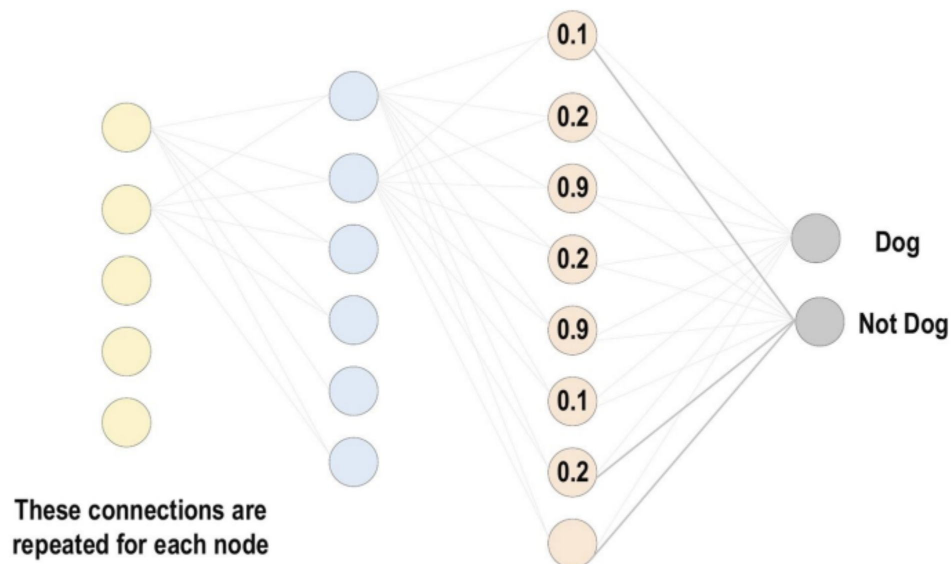

Figure 2.3: Fully Connected layer [35].

2.1.5 Activation Functions

Lastly, the activation function is a crucial CNN model parameter. They are used to learn and approximate any type of continuous and complex network variable relationship. In other words, it determines at the NN end which model information should be fire foward and which should not. It introduces nonlinearity into the network. ReLU, Softmax, tanH and Sigmoid are some of the most common activation functions. Each of these functions serves a distinct purpose. For a CNN model with binary classification, the Sigmoid and Softmax functions are preferred, while softmax is typically used for multiclass classification. In conclusion, CNN activation functions determine whether a neuron should be activated.

2.2 YOLO architectures

This section provides a summary of how the YOLO Neural Networks operates. The differences between the three most recent YOLO architectures (versions 3, 4, and 5) and the benefits and drawbacks of each Neural Networks. Newer versions like v6 and v7 [37] will not be addressed.

In 2016, Joseph Redmon [38] introduced a novel approach to object detection, the YOLO network. The authors use a single NN to process the entire image. This network divides the image into regions and forecasts bounding boxes and probabilities for each object. The predicted probabilities are used to quantify these bounding boxes. Compared to classifier-based systems, the model has several advantages. At test time, it examines the entire image, so its predictions are informed by the image's global context. It also predicts with a single network evaluation, as opposed to R-CNN [39], which requires thousands for a single image. This allows it to be 1000x faster than R-CNN and 100x faster than Fast R-CNN [40]. Figure 2.4 shows and briefly explains how this NN(YOLOv1) works.

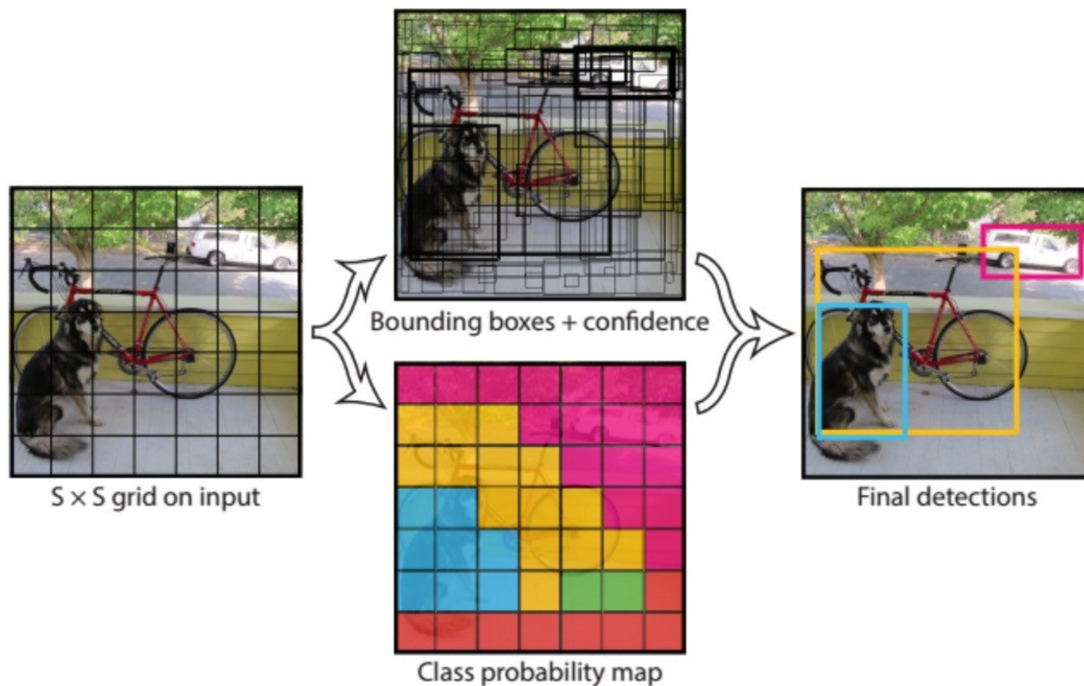

Figure 2.4: The model's operation [38](YOLOv1). It divides the image into an SxS grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an SxS x (B * 5 + C) tensor. [38]

Some terms used in YOLO architectures will be summarized:

- **Backbone**: Feature extractor Network, this "block" aims to extract features from the input, typically with a lot of convolutional layers and some pooling layers.
- **Neck**: The neck includes additional layers between the backbone and the neck and these layers are usually used to collect feature maps from different stages [41]. Usually, the neck is composed of several bottom-up paths and several top-down paths.
- **Head**: As to the head part, it is usually categorized into two kinds, i.e., one-stage object detector and two-stage object detector [41]. The most representative two-stage object detector is the R-CNN series [39], including fast R-CNN [40], faster R-CNN [30].

2.2.1 YOLOv3

Regarding the old versions YOLOv1 [38] and YOLOv2(also know as YOLO9000 [42]), these had some problems in detecting small objects. In this new NN an improved backbone extractor feature was created from the previous "Darknet19" [42] to "Darknet53"[43] to improve this problem. Figure 2.5 shows this backbone architecture.

Residual block, skip connections and up-sampling were implemented to significantly enhance the algorithm's accuracy. YOLOv3 uses similar concept to Feature Pyramid

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 2.5: Architecture of Darknet-53 [43].

Network (FPN) [44] as a neck, the main role of this neck is to extract feature maps from different stages wich is composed of several bottom-up and top-down paths and the head is composed of YOLO layer.
The YOLO layer produces the results after the image has been fed to Darknet53 for feature extraction and the pyramid network for feature fusion.

2.2.2 YOLOv4

The YOLOv4, a modified version of the YOLOv3. It uses Cross Stage Partitial Network (CSPNet) on Darknet, and uses the CSPDarknet53 backbone as feature extractor. The convolution architecture is a modified version of DenseNet [45].
YOLOv4 increased the map by about 10% compared to YOLOv3 and its FPS increased by 12% [41].
Spatial Pyramid Pooling (SPP) layer and Path Aggregation Network (PAN) make up the YOLOv4 neck. In order to increase the receptive field and short out crucial features from the backbone, feature agregation is done using the SPP layer and PAN.
In short, first the image is fed to CSPDarknet53 for feature extraction, then it is fed to PAN for fusion, finally the YOLO layer generates the results. The YOLOv4 similar to YOLOv3 also uses bag of freebies [46] and bag of specials [41] to improve algorithm performance [20].

2.2.3 YOLOv5

Although there is no paper regarding this version, YOLOv5 will be briefly discussed as well as the benefits it brought over previous versions. Glenn Jocher the founder and CEO of Ultralytics [47] released its open source implementation of YOLOv5 repo on GitHub [48].

This version replaces Darknet with Pytorch and uses CSPDarknet as its backbone. This backbone is able to reduce inference speed, improve accuracy, and reduce model size by reducing parameters because it solves repetitive gradient information in large backbones and integrates gradient change into the feature map [20].

YOLOv5 uses Path Aggregation Network as neck, this one uses a new FPN with some bottom ups and top downs layers, with this new PAN it is possible to increase the accuracy of object location and boost the information flow.
This newer version, adopts a Focus layer, the main purpose of the Focus layer is to reduce layers, parameters, FLOPS, CUDA memory and to increase forward and backward speed while minimally impacting mAP [1].
The NN architecture of YOLOv5 is shown in figure 2.6.

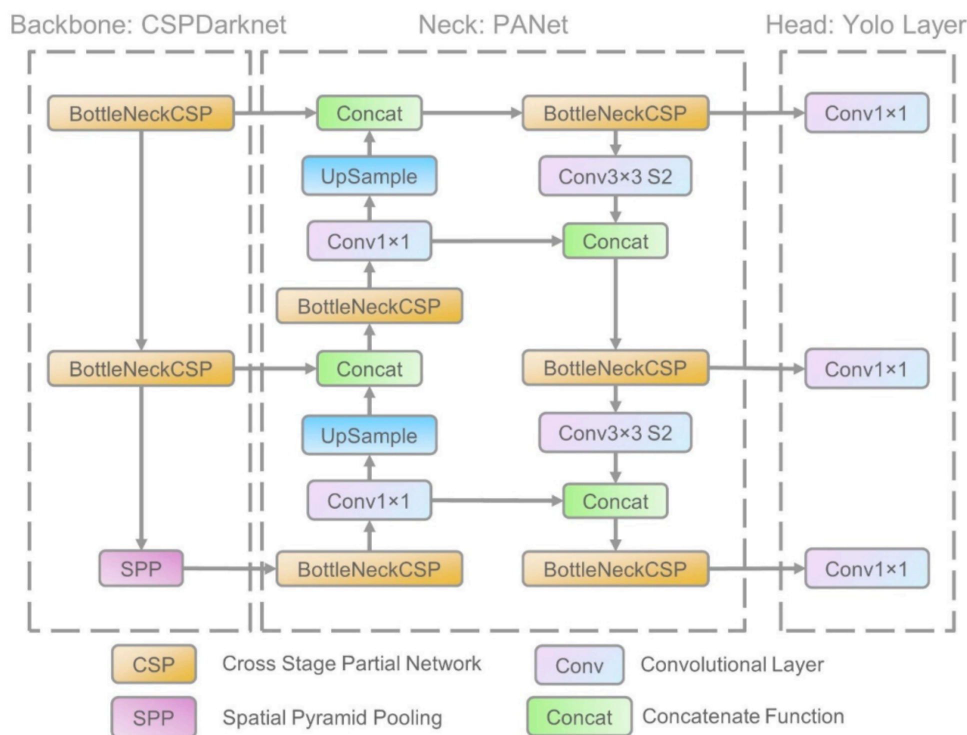

Figure 2.6: Architecture of YOLOv5. It consists of three parts: (1) Backbone: CSPDarknet, (2) Neck: PAN, and (3) Head: YOLO Layer. The data are first input to CSPDarknet for feature extraction, and then fed to PAN for feature fusion. Finally, YOLO Layer outputs detection results (class, score, location, size)[49].

In summary, in most applications, YOLOv5 achieves higher mAP than its predecessors while suffering for a small percentage of its obtained FPS.

[1] https://github.com/ultralytics/yolov5/discussions/3181m1

2.2.4 Main Diferences between YOLO, R-CNN and SSD architetures

Reference	Dataset Used	Algorithms	Conclusions/Findings
[50]	Pill image dataset Training: 5131 images Testing: NA Resolution: NA	Faster R-CNN YOLOv3 SSD	Faster R-CNN has higher mAP, but less than 10 FPS. YOLOv3 can detect multiple bounding boxes, and achieving 51 FPS, SSD 32 FPS .
[51]	Images collected by GF-1 and GF-2 satellites. Training: 825 Testing: 276 Resolution: 300 x 300, 416 x 416, 500 x 500, 1000 x 100	Faster R-CNN YOLOv3 SSD	YOLOv3 has higher mAP. YOLOv3 has 6 times more FPS than Faster R-CNN and 2 times more FPS than SSD.
[52]	MS COCO dataset Training: 118k images Testing: 5k images Resolution: NA	YOLOv3 YOLOv4	Average precision of YOLOv4 is higher than YOLOv3.
[20]	DOTA dataset Training: 11k images Testing: NA Resolution: NA	YOLOv3 YOLOv4 YOLOv5l	YOLOv5l has the highest mAP. YOLOv3 detects faster than the others, but has low Recall. F1 score of YOLOv4/v5l are higher compared to YOLOv3.
[53]	MS COCO dataset Training: 118k images Testing: 5k images Resolution: 640 x 640	YOLOv3 YOLOv4 YOLOv5	YOLOv5l has higher mAP than YOLOv3 and YOLOv4. YOLOv3 still has the most FPS than YOLOv4/v5l.
[54]	sign language dataset Training: 2k images. Testing: NA Resolution: NA	YOLOv3 YOLOv4 YOLOv5	YOLOv5 has the highest : Precision, Recall and $mAP_{0.5}$.
[55]	MS COCO dataset Training: 118k images Testing: 5k images Resolution: 640 x 640	YOLOv3 SSD300 Faster R-CNN	If we are dealing with a small dataset, the best option is R-CNN as it has extremely high mAP and Recall. SSD300 is more efficient and accurate than YOLOv3, however is not good for smaller objects.

2.3 Summary/Conclusions

Due to the conclusions drawn in 2.2.4, YOLO Neural Networks were assumed to be the starting point for this work, with the other mentioned Neural Networks, R-CNN [39], fast R-CNN [40], faster R-CNN [30], and SSD [29], being discarded.
Thus, the most recent versions of YOLO (versions: 3, 4 and 5) will be implemented on different types of hardware: CPU, GPU and FPGA (with frameworks: Pynq [1] and Vitis-AI [2]).

In addition to implementing these Neural Networks, the power consumed by each will be evaluated, obtaining the *frames per joule* of each.

Chapter 3

YOLO Implementations, frameworks and tools used

This chapter describes all the hardware and software used, both for the implementation of YOLO Neural Networks and for measuring energy consumption.

3.1 Hardware, software and tools used

Hardware and Software:

The experimental setup used to implement YOLO Neural Networks consists of a desktop with an Intel core i5-12400 6 - Core CPU, a 12GB RTX 3060 GPU, and 16GB of RAM. For implementations in reconfigurable logic the FPGA Zynq UltraScale+ MPSoC ZCU104 was used.

The desktop used has Windows 10 as the operating system with all updated drivers and CUDA version 11.3.
Pynq and Vitis-AI environments were used in on the FPGA. The board image versions utilized were: Pynq 2.7 and Vitis-AI v2022.1 - v2.5.0.

Auxiliary Tools used:

The power consumption of the Desktop and FPGA was measured using a multimeter (Uni-t UT803. driver: v1.0 [56] and a DC power supply (GPE-3323 GW INSTEK). Some "crocodile" cables were also used to make the necessary connections.

3.1.1 Methods for measuring power consumption.

To measure the energy consumption in the different hardware, we used:

- **Computer/Desktop** - 220 volts were measured at the socket and the same were assumed for the remaining measurements. From an extension coord that is connected to a socket, the cables were cut to be able to measure the Amps in series, from there it is possible to plot the energy consumed by the Desktop, to calculate the

watts just multiply the 220 volts constant for the amps supplied. The Figure 3.1 illustrates the measurement procedure.

$$P = VI \tag{3.1}$$

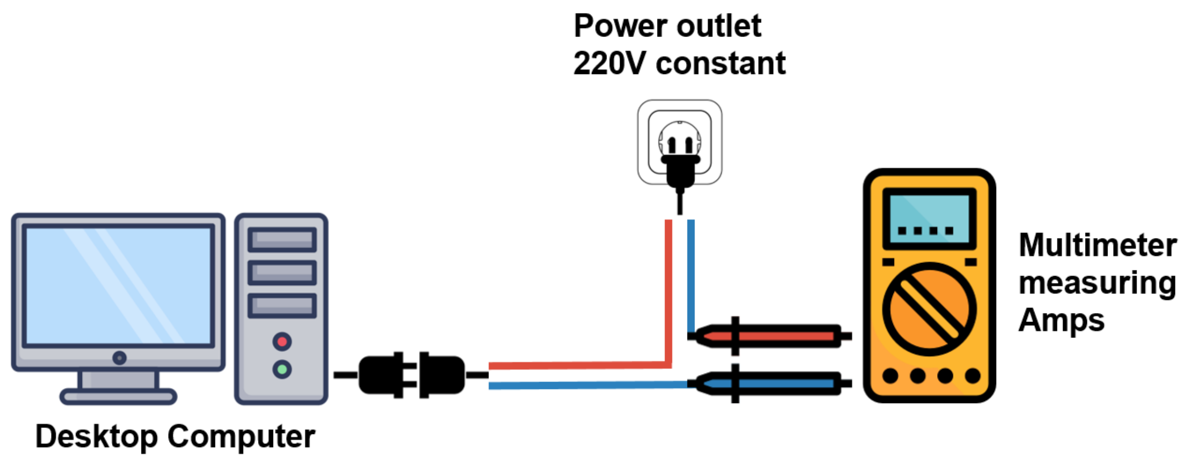

Figure 3.1: Illustration of how Desktop power consumption measurements were performed.

The information regarding to the ampere measurements was exported via (*.xls*), after which the watts were calculated and time plots were created.The figure 3.2 shows an example of power measurement.

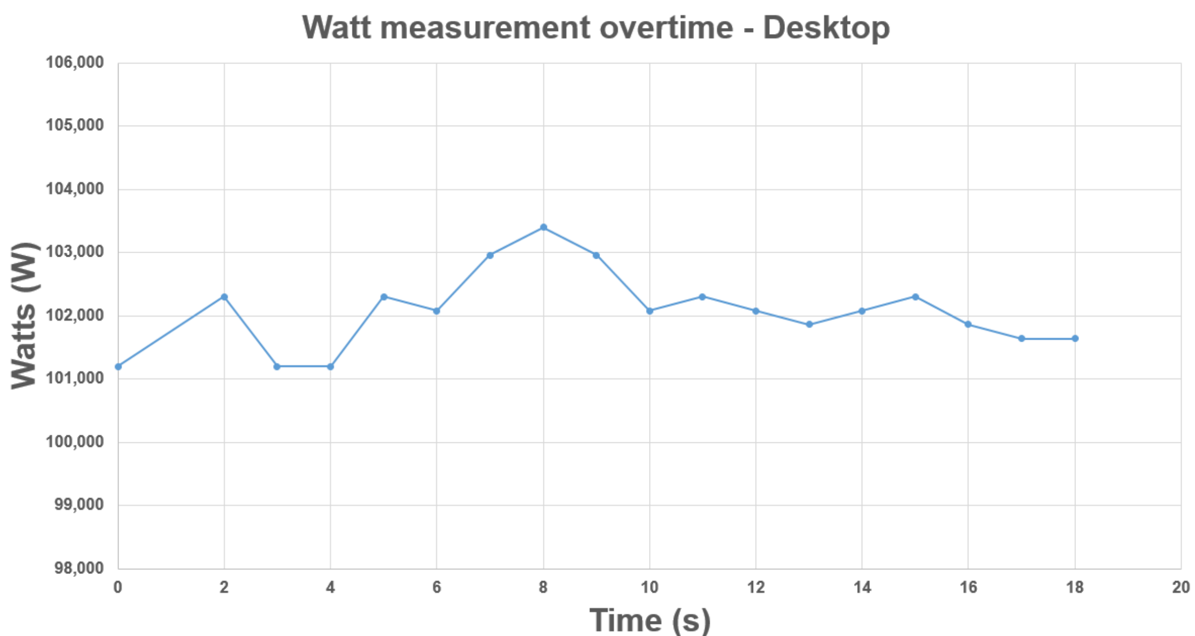

Figure 3.2: Watt measurement during 18 seconds, Desktop in "idle" (Total Watts consumed by the Desktop).

- **FPGA - Using Pynq framework** - There is a *Python package* to calculate the power consumed quite accurately. The figure 3.3 shows an example of power measurement on Pynq framework.

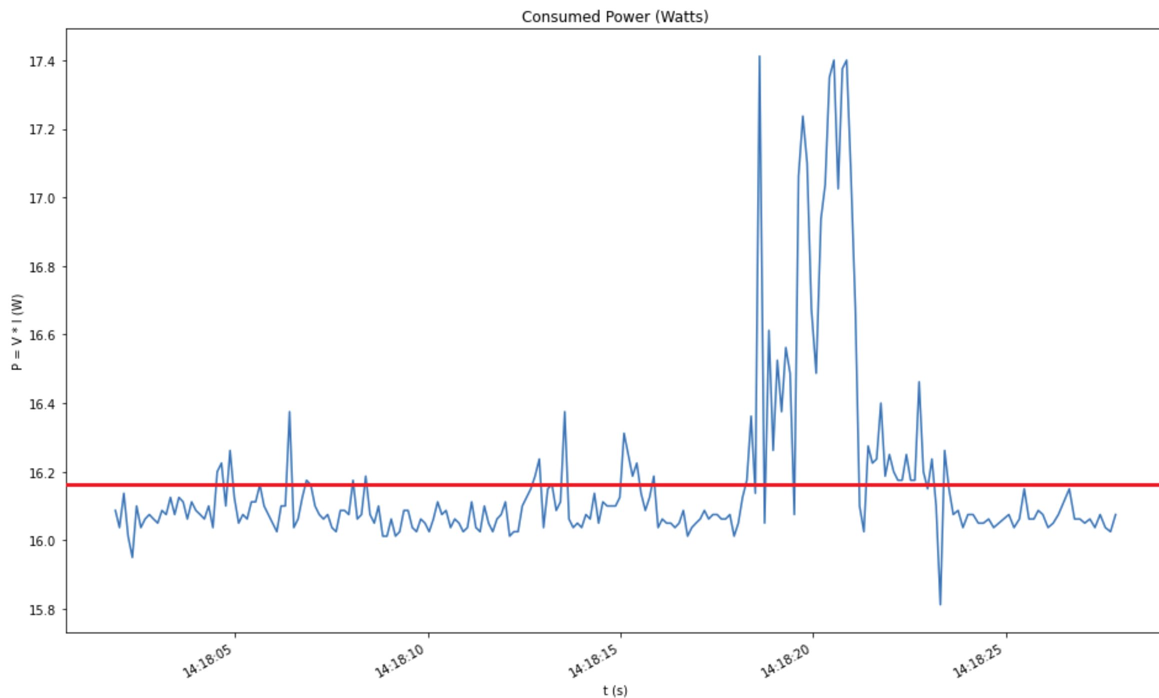

Figure 3.3: Watt measurement overtime, FPGA in ”idle”. The red line indicates the average watts (16.16 Watts).

Every 1 second, 10 measurements are taken and saved in a dataframe, which can then be manipulated to create plots. With so much data saved, all measurements were later added up and divided by the number of measurements taken. This calculates the average wattage consumed by the FPGA.
The figure 3.4 shows the snippet of the Python code to manipulate the dataframe created, thus achieving the minimum and maximum wattage, as well as the average wattage.

```
print('MAX WATTAGE =\n',d.max(),'watts')
print('MIN WATTAGE =\n',d.min(),'watts')
```

```
MAX WATTAGE =
 Mark           1.000
12V_power    17.412
dtype: float64 watts
MIN WATTAGE =
 Mark           0.000
12V_power    15.812
dtype: float64 watts
```

```
#calculate average wattage
da = d['12V_power'].sum()
print(da/d[d.columns[0]].count(),"watts")
```

```
16.16776833976834 watts
```

Figure 3.4: Dataframe manipulation, getting min and max watts, as well as average watts.

- **FPGA - Using Vitis-AI framework** - To perform power measurements with this framework, a different approach is required as it is not possible to perform these measurements using the Pynq framework. For this, a DC power supply (GPE-3323 GW INSTEK) was used to supply constant 12 Volts to the FPGA, and the current was measured using a multimeter. The Figure 3.5 illustrates the measurement procedure.

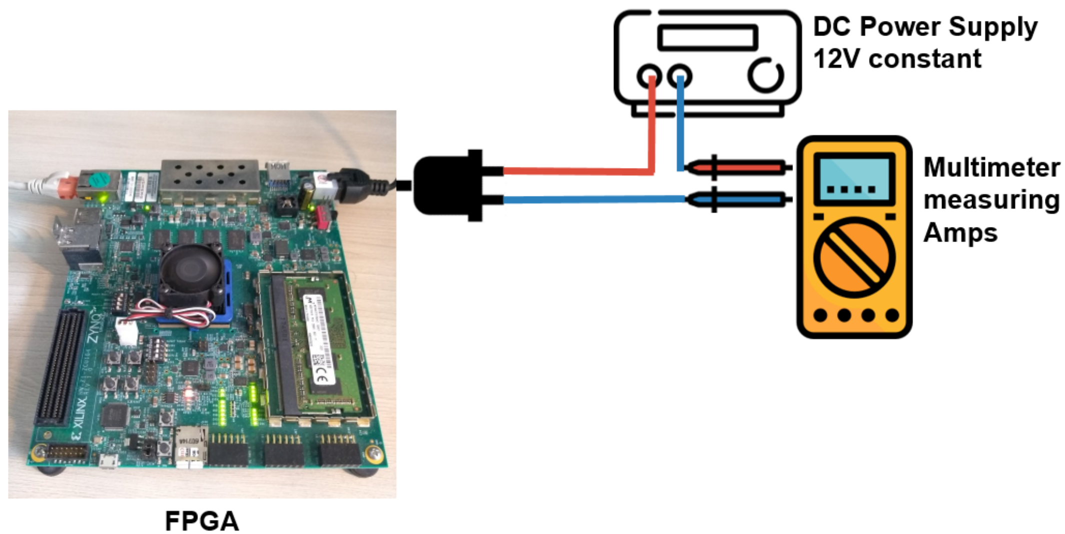

Figure 3.5: Illustration of how measurements of power consumption were performed on the FPGA (17.79 Watts average).

Later, with the data obtained, graphs similar to the one in Figure 3.6 were created.

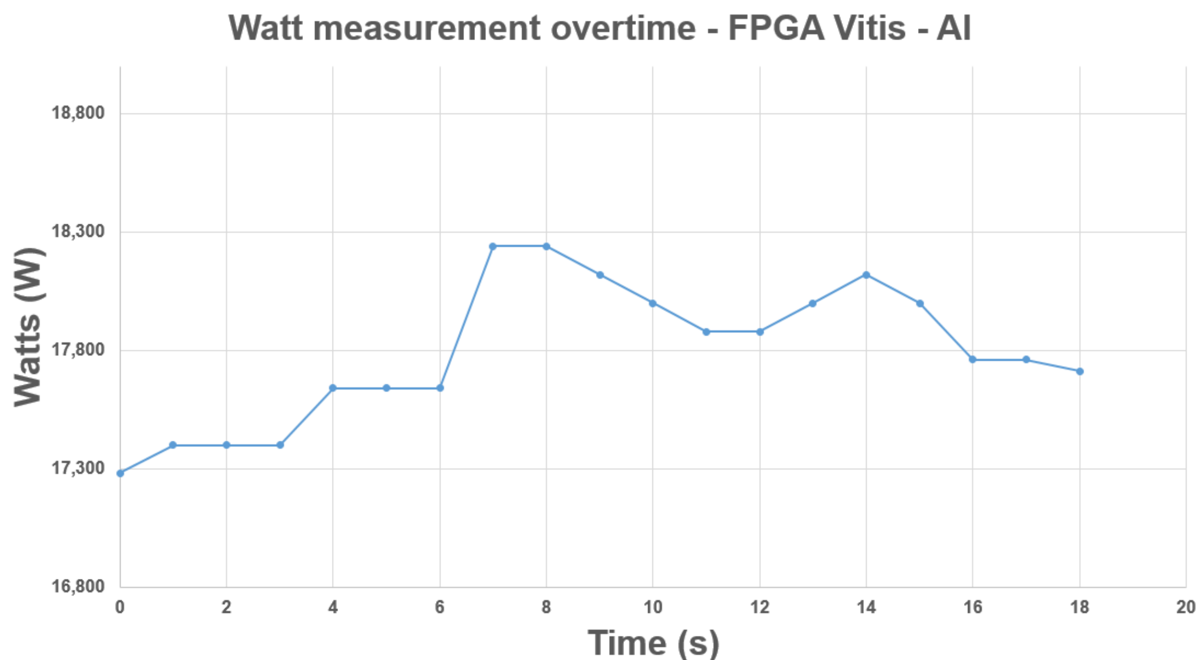

Figure 3.6: Power consumption measurement example on Vitis-AI framework during 18 seconds, FPGA in "idle".

3.2 Implementation Notes

We attempted to utilize implementations that were compatible with the hardware in all three versions. For example, because the Pynq [1] framework does not allow Tensorflow [57], we are limited to using PyTorch [58] in its implementations.
Since there is excellent material with Pytorch implementations of the YOLOv3 [43] and YOLOv5 [48] NN , these will be implemented and tested on different types of hardware. The table in the figure 3.1 shows the differences between the four tested models, input size, $mAP_{0.5:0.95}$, $mAP_{0.5}$ and parameters.

Model	Size	$mAP_{0.5:0.95}$	$mAP_{0.5}$	Parameters (M)
YOLOv3	640 x 640	0.467	0.661	61.9
YOLOv3-tiny	640 x 640	0.190	0.367	8.8
YOLOv5l	640 x 640	0.489	0.675	46.5
YOLOv5s	640 x 640	0.374	0.571	7.2

Table 3.1: Differences between implemented and tested models (MS COCO dataset [19]. These values were obtained using pre-trained models, and 5000 images were used for validation.

Regarding YOLOv4, it has little support for Pytorch , so there were always a lot of bugs despite its implementation, so the implementation in these three types of hardware was not carried out, so it will not be documented, and it will only be in the Vitis-AI framework [2].

The implementations tested in sections 3.3 and 3.4.1 were carried out with inference from a video (taken from: [59]) with a resolution of 1280 x 720 pixels, with 1 minute, this video has 50 FPS, that is, a total of 3000 frames.
Before simply running the video inference, some changes were made to the detection script in order to provide the user with the total time spent in the video inference.

3.3 CPU and GPU implementation

The four models listed in Table 3.1 were tested on CPU and GPU. Using the scripts [48], a video inference and power consumption measurement were performed for each of these models.

Using the two types of hardware tested(CPU and GPU), the power consumed by the Desktop was plotted, as well as the time spent for each inference. The figure 3.7 shows an example of the final result of the watt measurement plot.

3.4 FPGA implementation

3.4.1 Using Pynq framework

Pynq framework is and open-source project from Xilinx [60] [1]. It uses Python language and libraries, so users can take advantage of programmable logic. Pynq was created with the purpose of facilitating the deployment of applications, without users

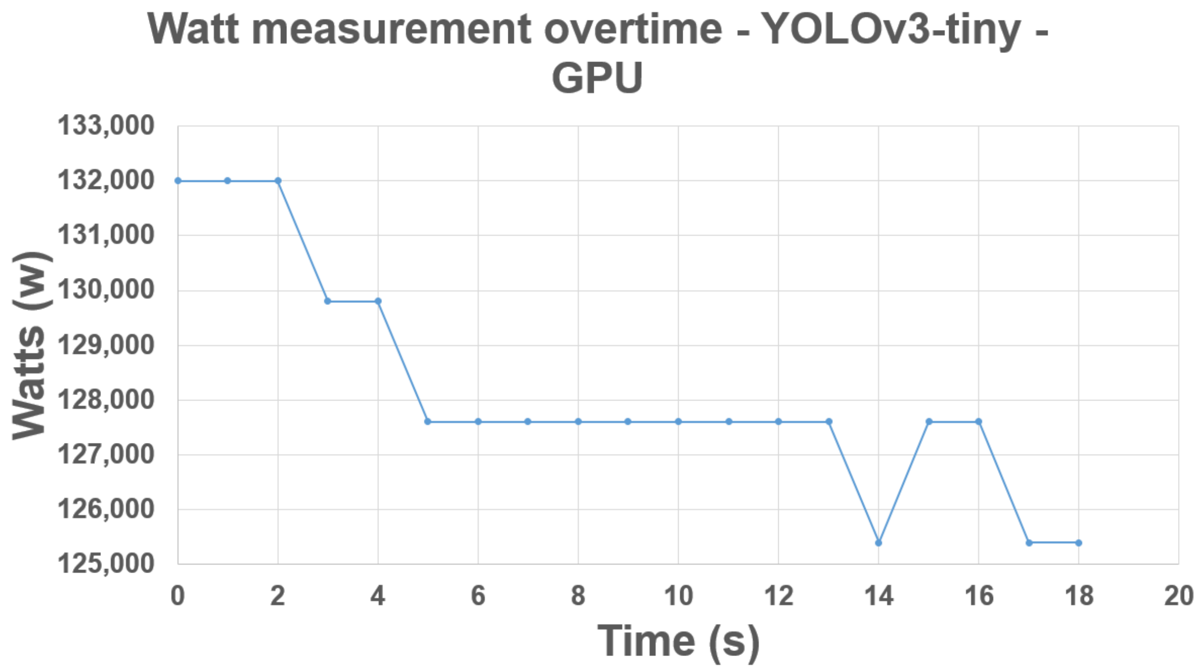

Figure 3.7: Watt measurements overtime, performing inference from a video on GPU using YOLOv3-tiny. Time: 17.81 seconds. (Total Watts consumed by the Desktop, 128.17 watt average).

having a hardware design background [61].
Figure 3.8 presents the possibilities that this framework offers.

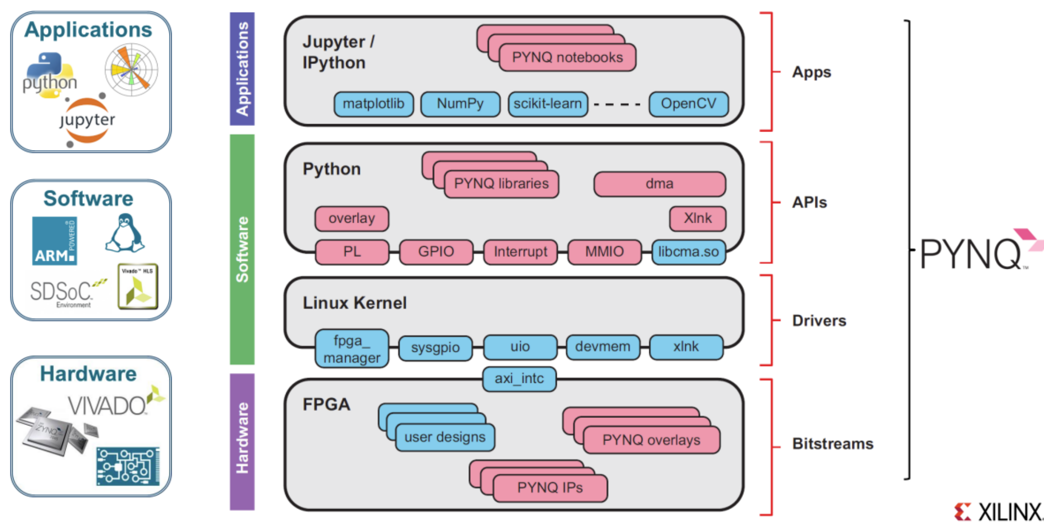

Figure 3.8: Pynq framework dataflow.

With the creation of Jupyter Notebooks [62], the user can make use of several libraries at his disposal, he can also make use of existing overlays or use his own. With this, it is possible to create high performance applications with:

- Parallel hardware execution.
- High frame-rate video processing.
- Hardware accelerated algorithms.
- Real-time signal processing.
- High bandwidth IO and low latency control.

Creating Intellectual Property (IP)s is a challenging task, especially for Neural Networks as complex as YOLO. One solution discovered was the ability to reload the weights of any layer during runtime [23], however, this solution is only relevant to simpler Neural Networks, such as LeNet-5.

Discarding the possibility of accelerating the computation of these networks with the Pynq framework, the models listed in table 3.1 were tested and implemented. These tests were using only the sequential computing power provided by quad-core ARM Cortex-A53 and dual-core Cortex-R5 processors. As with the implementation of these models, the energy consumed is also measured, Figure 3.9 shows a graph of the watts overtime, using the YOLOv5l NN.

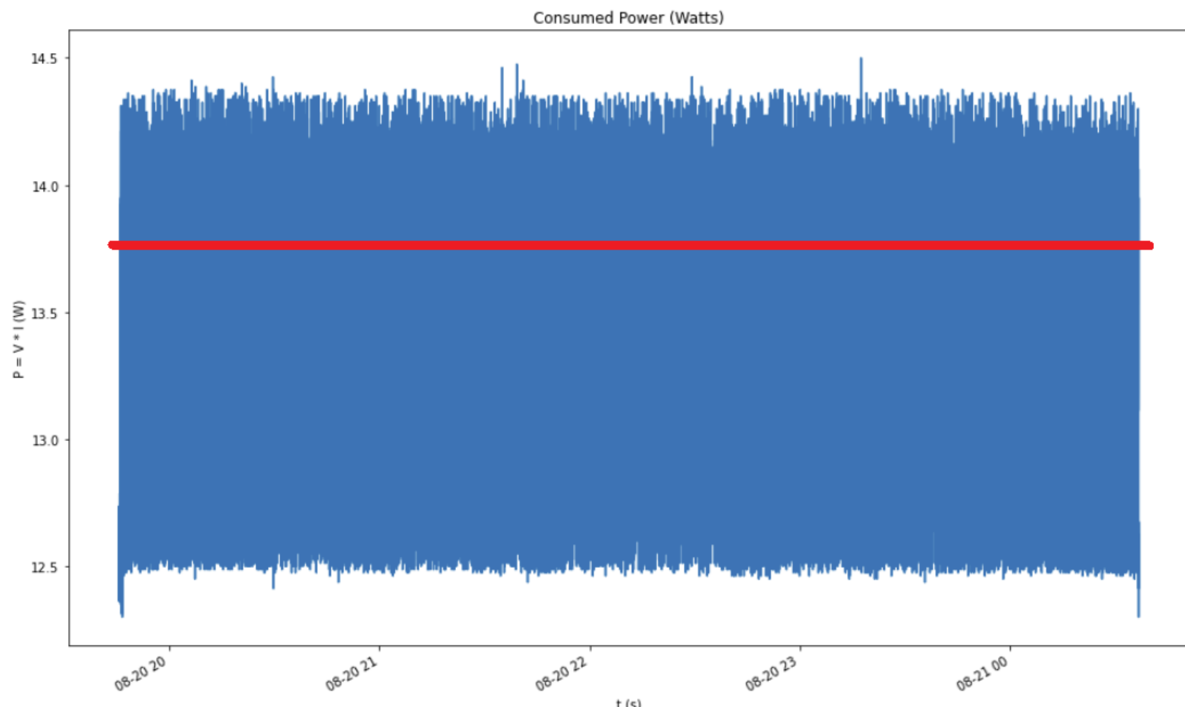

Figure 3.9: Watt measurement overtime, performing inference from a video, using YOLOv5l NN. Time: 16965.2 seconds. (Total Watts consumed by the FPGA, 13.71 Watt average).

3.4.2 Using Vitis-AI framework

Vitis AI [2] is a full-featured AI inference development platform based on Xilinx devices, boards, and Alveo data center acceleration cards. It includes a large number of AI models, optimized Deep Learning Processor Unit (DPU) cores, tools, libraries, and example designs for AI on the edge and in the data center. It is designed with high efficiency and ease of use in mind, allowing AI acceleration on Xilinx FPGAs and adaptive System on a Chip (SoC) to reach its full potential.

Figure 3.10 shows what this framework offers the user.

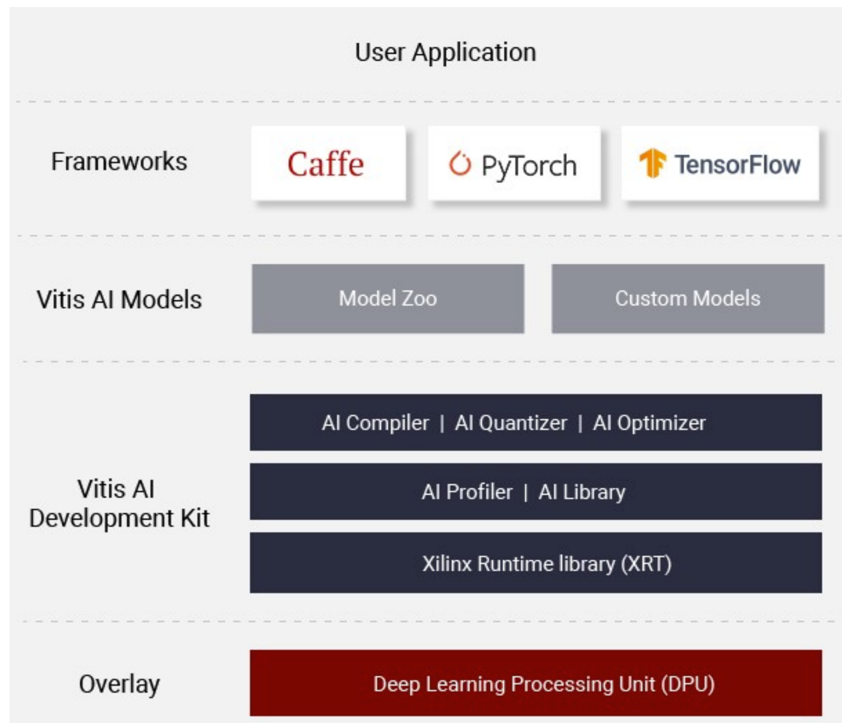

Figure 3.10: Applications possibilities using Vitis-AI .

In the context of Convolutional Neural Networks, Vitis-AI provides:

- Pre-optimized AI models, ready to deploy on Xilinx devices.
- Powerfull open-source AI quantizer that supports pruned and unpruned model quantization, calibration and fine-tunning. (In Machine Learning (ML), pruning is removing unnecessary neurons or weights, reducing the Neural Networks parameters and obviously the size of the model.)
- A user-friendly compilation and deployment flow to meet costumer defined model and operators(This compiler generates a DPU instruction to later be used in the final application).
- Offers the AI library with open-source high-level C++ and Python APIs for maximum portability from edge to cloud.
- Different environments, such as: Tensorflow(versions: 1.15 and 2.8) [57], Pytorch [58] and Caffe [63].

- DPU IP cores that are efficient, scalable, and customizable in order to meet various needs such as throughput, latency, power, and lower precision.

AI Model Zoo

AI Model Zoo is available to all users with deep learning models from popular frameworks including Pytorch, Tensorflow, Tensorflow 2, and Caffe. AI Model Zoo offers optimized and retrainable AI models that allow for faster deployment, performance acceleration, and productization across all Xilinx platforms. Figure 3.11 summarizes the possibilities that the Model Zoo offers.

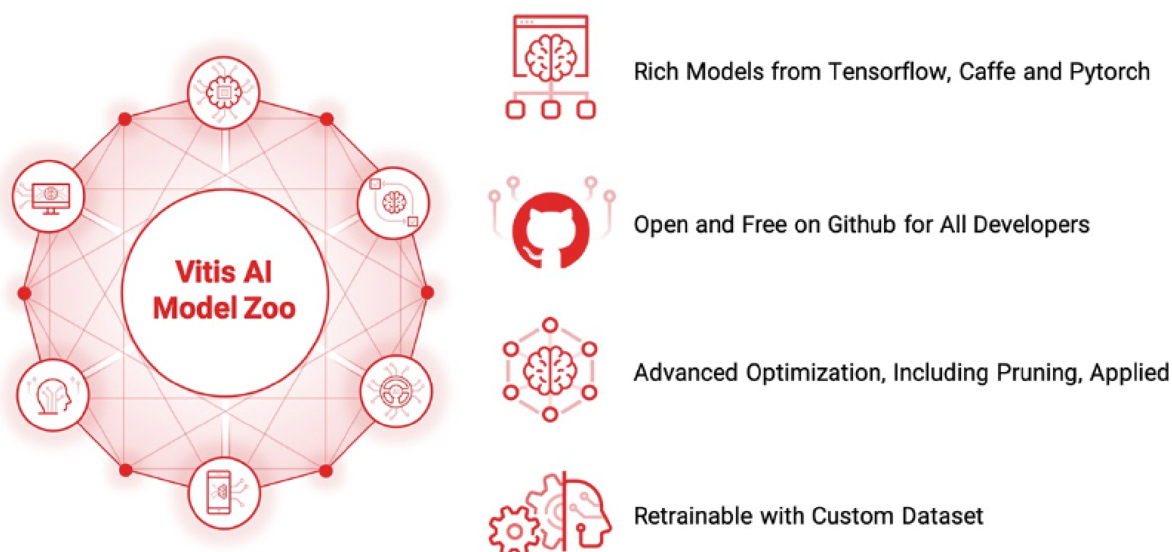

Figure 3.11: Applications possibilities using Vitis-AI .

AI Optimizer, Quantizer and Compiler

With cutting-edge model compression technology, the AI optimizer reduces model complexity by a factor of 5X to 50X with minimal impact on accuracy. The performance of your AI inference is enhanced by the use of deep compression.

By converting 32-bit floating-point weights and activations to INT8 fixed-point, the AI Quantizer can reduce computational complexity without sacrificing prediction accuracy. The fixed-point network model requires less memory bandwidth than the floating-point model, resulting in faster speed and greater power efficiency.

The AI compiler maps the AI model into an instruction set and data flow that is highly efficient. In addition, it performs complex optimizations such as layer fusion, instruction scheduling, and reuses on-chip memory as much as possible. Figure 3.12 illustrates what these tools offer in general.

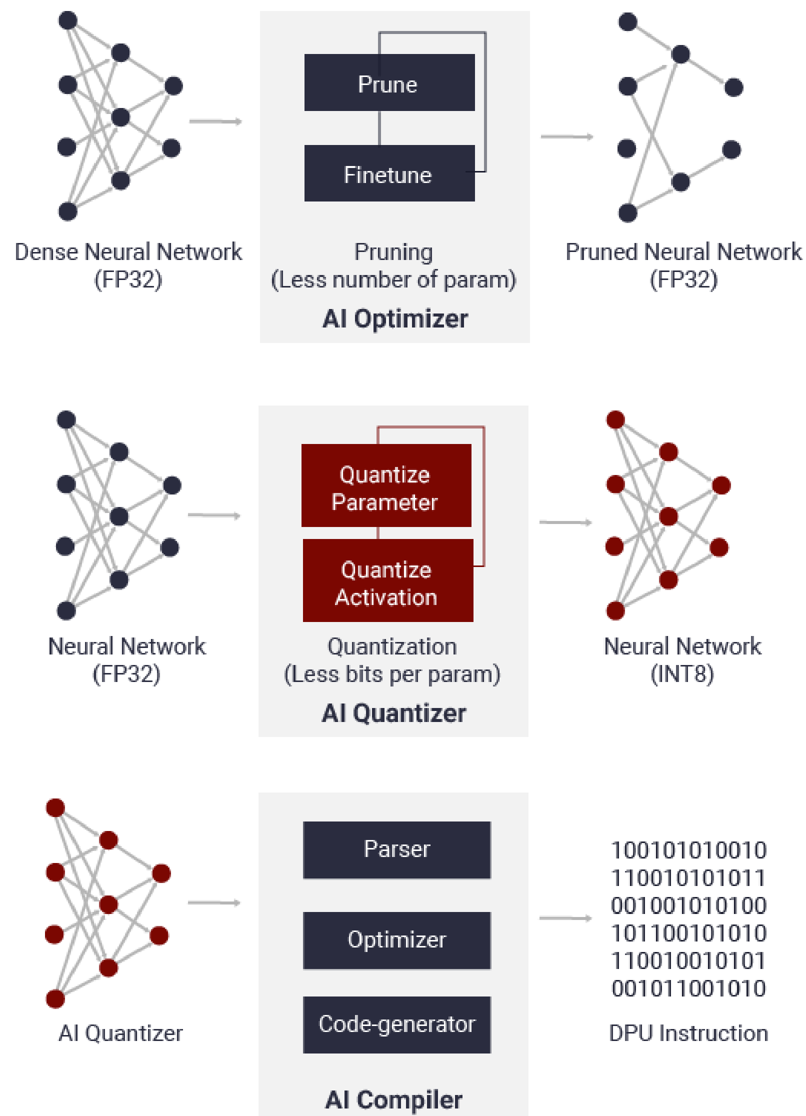

Figure 3.12: Illustration of the possibilities that these tools offer.

AI Profiler, AI Library

The performance profiler enables programmers to conduct an in-depth analysis of the AI inference implementation's efficiency and utilization.

The Vitis AI Library is a set of high-level APIs and libraries designed for AI inference with DPU cores. It is based on the Vitis AI runtime (VART) and provides unified APIs and user-friendly interfaces for the deployment of AI models on Xilinx platforms.

3.4.2.1 DPUCZDX8G Overview

The DPUCZDX8G is the Deep Learning Processor Unit designed for the Zynq UltraScale+ MSoC. It is a configurable computation engine that has been optimized for Convolutional Neural Networks. The degree of parallelism utilized by the engine is a design parameter that can be chosen based on the device or application being targeted. The DPU is a microcoded compute engine with an efficient, optimized instruction set

capable of supporting the inference of the majority of Convolutional Neural Networks.

The DPUCZDX8G top-level interfaces are shown in the Figure 3.13. The Top-Level entity is shown in Figure 3.13, as well the block diagram in Figure 3.14.

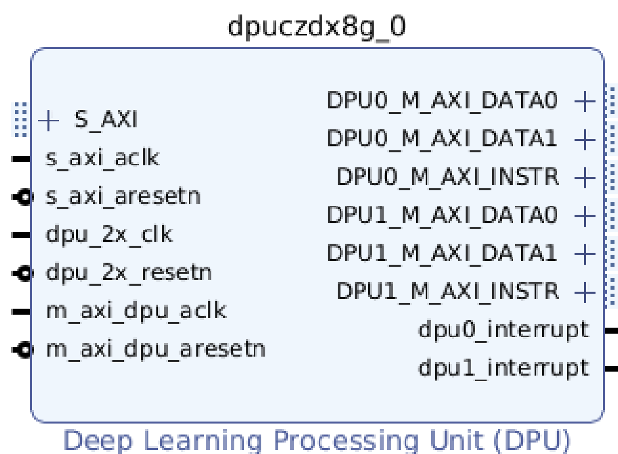

Figure 3.13: DPU IP top level entity [64].

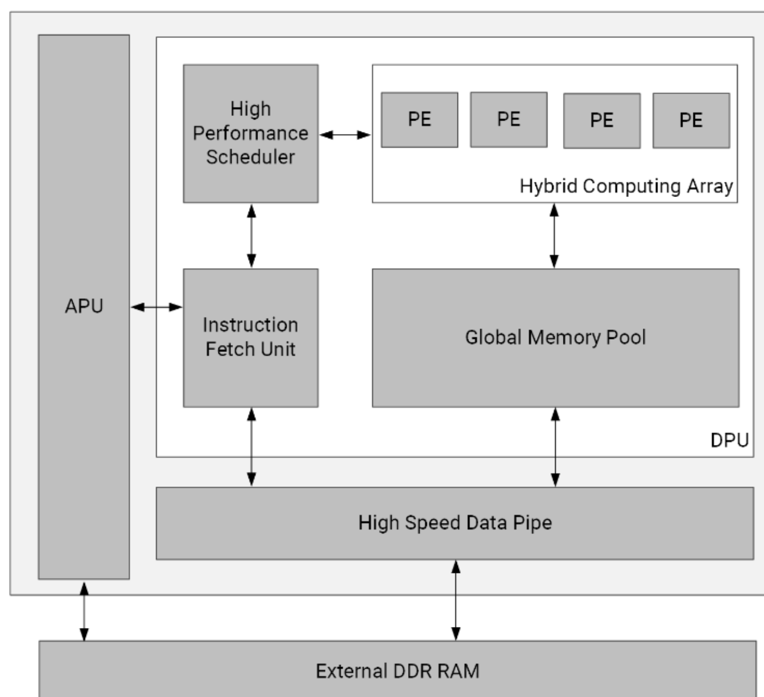

Figure 3.14: DPU IP Top-Level Block Diagram

The diagram 3.15 illustrates the DPUCZDX8 hardware architecture in greater detail. After start-up, the DPUCZDX8G fetches instructions from off-chip memory to control the operation of the computing engine. The instructions are generated by the Vitis AI compiler, which performs significant optimizations, such as layer fusion.

To achieve high throughput and efficiency, input activations, intermediate feature-maps, and output meta-data are buffered on-chip. Data is reused as frequently as possible

to reduce external memory bandwidth needs. A deep pipelined design is used for the computing engine.

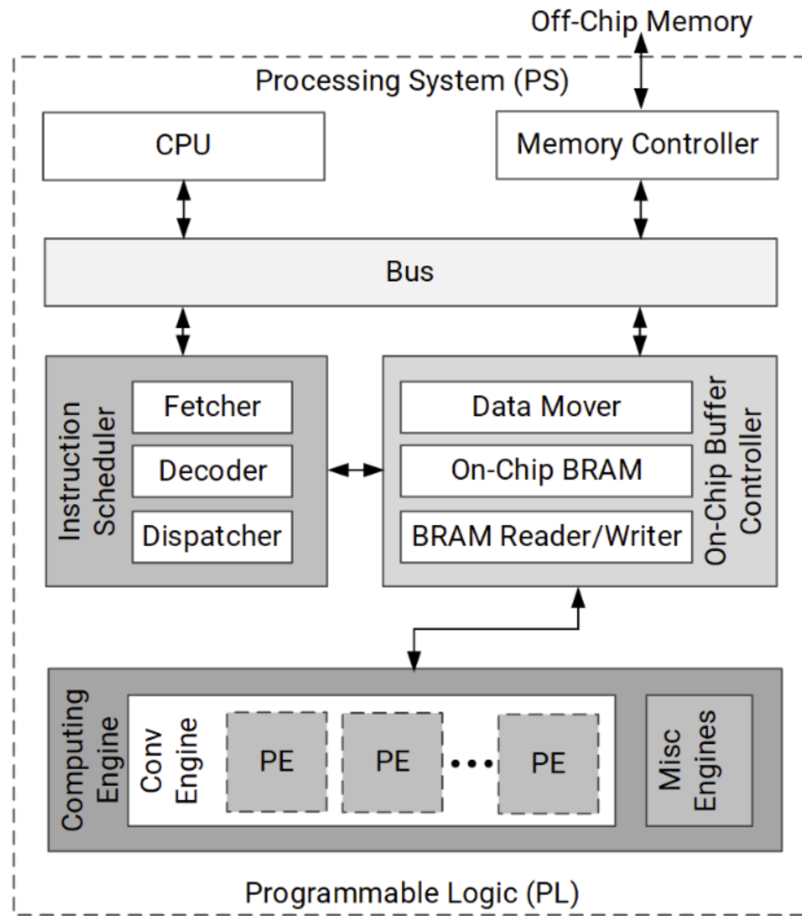

Figure 3.15: DPU Hardware Architecture.

Brief Introduction

The DPUCZDX8G IP provides some user-configurable parameters to optimize resource usage and customize different features. Different configurations can be selected for Digital Signal Processing (DSP) slices, Look Up Table (LUT), block RAM, and UltraRAM usage based on the amount of available programmable logic resources.
There are also options for additional functions, such as channel augmentation, average pooling, depthwise convolution, and softmax. Furthermore, there is an option to determine the number of DPUCZDX8G cores that will be instantiated in a single DPUCZDX8G IP. The DNN features and the parameters supported by the DPUCZDX8G are shown in the table 3.2.

It is also possible to set the number of cores, convolution architecture, DSP cascade and usage, ultra RAM usage.

Features	Description	
Convolution	Kernel Sizes	w, h: [1, 16]
	Strides	w, h: [1, 8]
	Padding	w: [0, kernel_w - 1] h: [0,kernel_h - 1]
	Input Size	Arbitary
	Input Channel	1∼256 * channel_parallel
	Output Channel	1∼256 * channel_parallel
	Activation	ReLU, ReLU6, LeakyReLU, Hard Sigmoid and Hard Swish
Depthwise Convolution	Kernel Sizes	w, h: [1, 256]
	Strides	w, h: [1, 256]
	Padding	w: [0, min(kernel_w - 1,15)] h: [0, min(kernel_h - 1,15)]
	Input Size	Arbitary
	Input Channel	1∼256 * channel_parallel
	Output Channel	1∼256 * channel_parallel
	Activation	ReLU, ReLU6, LeakyReLU, Hard Sigmoid and Hard Swish
Tranposed Convolution	Kernel Sizes	kernel_w/stride_w: [1, 16]
	Strides	kernel_h/stride_h: [1, 16]
	Padding	w: [0, kernel_w - 1] h: [0, kernel_w - 1]
	Input Size	Arbitary
	Input Channel	1∼256 * channel_parallel
	Output Channel	1∼256 * channel_parallel
	Activation	ReLU, ReLU6 and LeakyReLU
Depthwise Transposed Convolution	Kernel Sizes	kernel_w/stride_w: [1, 16]
	Strides	kernel_h/stride_h: [1, 16]
	Padding	w: [0, kernel_w - 1] h: [0, kernel_h - 1]
	Input Size	Arbitary
	Input Channel	1∼256 * channel_parallel
	Output Channel	1∼256 * channel_parallel
	Activation	ReLU, ReLU6, LeakyRelu, Hard Sigmoid and Hard Swish
Max Pooling	Kernel Sizes	w, h: [1, 256]
	Strides	w, h: [1, 256]
	Padding	w: [0, min(kernel_w - 1,15)] h: [0, min(kernel_h - 1,15)]
Average Pooling	Kernel Sizes	w, h: [1, 256]
	Strides	w, h: [1, 256]
	Padding	w: [0, min(kernel_w - 1,15)]∼ h: [0, min(kernel_h - 1,15)]
Element-wise - Sum	Input Channel	1∼256 * channel_parallel
	Input Size	Arbitrary
	Feature Map N	1∼4
Element-wise - Multiply	Input Channel	1∼256 * channel_parallel
	Input Size	Arbitrary
	Feature Map N	2
Concat	Output Channel	1∼256 * channel_parallel
Reorg	Strides	stride * stride * input_channel ≤ 256 * channel_parallel
Fully Connected	Input Channel	Input_channel ≤ 2048 * channel_parallel
	Output Channel	Arbitrary

Table 3.2: Features and parameters supported by the DPUCZDX8G, more information available at [64].

Architectures of DPUCZDX8G

It is important to note that DPU can be configured with various convolution architectures, these have the designations: B512, B800, B1024, B1152, B1600, B2304, B3136, and B4096.
There are three dimensions of parallelism in the DPU: Pixel Parallelism (PP), Input Channel Parallelism (ICP), and Output Channel Parallelism (OCP). The figure 3.16 shows shows a visualization of these three levels of parallelism.

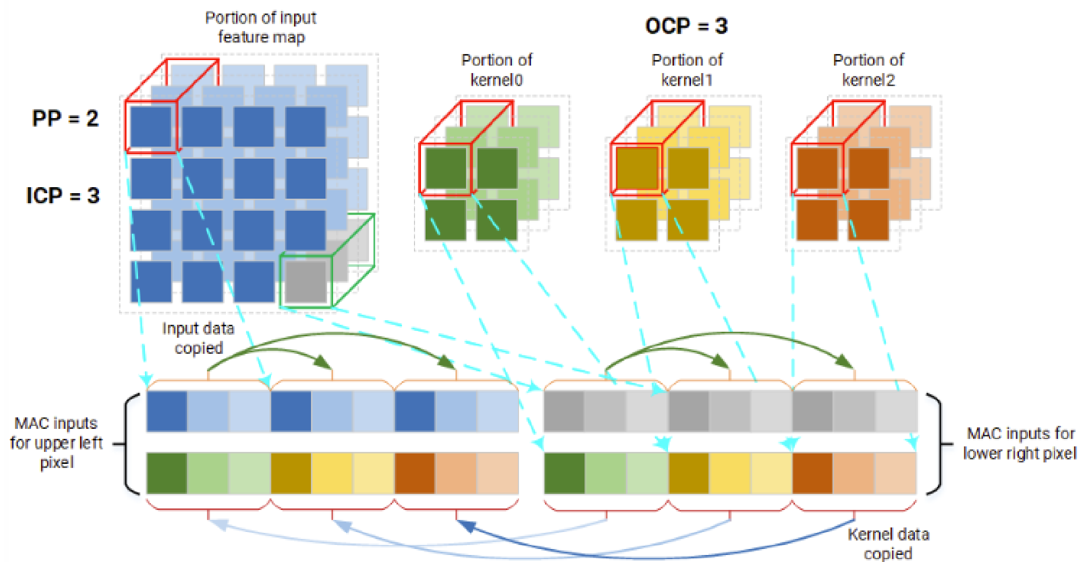

Figure 3.16: DPU Architecture, ICP = 3, OCP = 3, and PP = 2. OCP is equivalent to the number of kernels using during a convolution computation. The pixels used in the figure are arbitrary to maintain clarity [64].

It is important to state that different architectures require different programmable logic resources. Obviously larger architectures can achieve higher performance with more resources. The parallelism for the different architectures is listed in the following Table 3.3. The architecture in bold "B4096" was used in this work.

DPUCZDX8G Architecture name	Pixel Parallelism (PP)	Input Channel Paralelism (ICP)	Output Channel Paralelism (OCP)	Peak Ops (operations/per cycle)
B512	4	8	8	512
B800	4	10	10	800
B1024	8	8	8	1024
B1152	4	12	12	1150
B1600	8	10	10	1600
B2304	8	12	12	2304
B3136	8	14	14	3136
B4096	**8**	**16**	**16**	**4096**

Table 3.3: Parallelism for different convolution architectures [64]. In each clock cycle, the convolution array performs a multiplication and an accumulation, which are counted as two operations. Thus, the peak number of operations per cycle is equal to PP * ICP * OCP * 2.

Resource utilization of the DPUCZDX8G

Table 3.4 shows the resource utilization used by the DPU with a single core example.

DPUCZDX8G Architecture name	LUT	Register	Block RAM	DSP
B512	26391	34141	72	118
B800	28863	40724	90	166
B1024	33796	48144	104	230
B1152	31668	46938	121	222
B1600	37894	58914	126	326
B2304	41640	69180	165	438
B3136	45856	80325	208	566
B4096	51351	98818	255	710

Table 3.4: The data is based on the ZCU102 platform with low RAM usage, channel augmentation, alu parallel = PP/2, conv: leaky ReLU + ReLU6, alu: ReLU6 features, and high DSP usage.

There are other architecture options, such as: ultraRAM that uses more BRAM36k blocks. Note that the architecture used was "B4096" and to use lighter architectures it would be necessary to change the board image to the new architecture, as well as compile the models again.

3.4.2.2 Notes on the different environments and examples followed

With the materials available in [65], two examples (07-yolov4-tutorial and 09-mnist_pyt, one in Tensorflow [57] and the other in Pytorch [58] were developed as a starting point for the work in this framework.

3.4.2.3 YOLOv3, v4, v5 implementation - Tensorflow and Pytorch

YOLOv5 Implementation

Following the example mentioned above "07-yolov4-tutorial" [65] and the materials avaliable [66] we tried to implement the YOLOv5s NN using the Tensorflow environment as in the example, with the following steps:

- Docker setup and Windows setup (vitis-ai-tensorflow-1.15).
- Float training (meeting the DPU requirements using [66], with some tweaks on the "yolo5_small_darknet" architecture).
- Model conversion to Tensorflow frozen graph.("yolov5s.h5" to "yolov5s.pb").
- Model Quantization.
- Evaluate the quantized model.
- Model compilation.
- Model deployment on ZCU104.

Although these tasks seem easy to accomplish, since this framework had all the built-in quantization and compilation dataflow, new problems appeared each time.
During the implementation, using this framework, the versions that support GPU docker were not working, so the training had to be performed on the CPU. In order to speed up the deployment process, training's were performed in a few epochs due to time constraints.

Changing the YOLOv5 NN is a complex and time-consuming task, as is the entire process until deployment. After the pre-deployment steps were completed successfully, the deployment process did not produce satisfactory results. The maximum achievable frame rate was 0.75 FPS.

Following example (09-mnist_pyt) [65], the NN was modified with the addition of one more convolutional layer. After that, the steps followed:

- Train the NN for 3 epochs, achieving 98.95% accuracy.
- Quantizes the model in 2 modes: calibration and test, obtaining 98.99% and 98.94% of accuracy respectively.
- Compiling the model for the targeted board (ZCU104 in this case).
- Finally, a model in the format "CNN_zcu104.xmodel" is obtained that can be used in the Pynq framework.

```
Classifying 10000 digit pictures ...
Overall accuracy: 0.9883
  Execution time: 2.3354s
      Throughput: 4281.8472FPS
```

Figure 3.17: Throughput generated by the model (Mnist Dataset).

Then, starting from the example mentioned above, we tried to change it for the "YOLOv5s" NN.

The level of work complexity in both the Tensorflow [57] and Pytorch [58] environment is identical. So progress over time was very little. Due to constant errors in the consoles, sometimes it was necessary to create a docker from scratch, since the simple update of a package can damage the entire Linux system. The YOLOv5 NN could only be deployed in the Tensorflow environment.
It was also found that Xilinx [60] already has versions of YOLOv5 that work in this framework, however, due to licensing issues, these implementations will not be available to the public in the near future.

YOLOv3 and v4 Implementation

The YOLOv3-tiny NN implementation was successful. Models present in the Model Zoo [28] were also tested and the respective energy consumption measurements were performed. The Table 3.5 shows the tested models as well some metrics, the last two

Model Name	Dataset	Framework	Input size	Pruned	Float Ops	Quantized Acc
YOLOv4	COCO	Dk	416 x 416	0.36	38.2G	NA
YOLOv4	COCO	Dk	416x 416	NO	60.1G	NA
OFA-YOLO	COCO	Pt	640 x 640	0.5	24.62G	0.378
OFA-YOLO	COCO	Pt	640 x 640	0.3	34.72G	0.401
OFA-YOLO	COCO	Pt	640 x 640	NO	48.88G	0.421
YOLOv3	COCO	Tf2	416 x 416	NO	65.9G	0.331
YOLOv4	COCO	Tf	416 x 416	NO	60.3G	0.393
YOLOv4	COCO	Tf	512 x 512	NO	91.2G	0.412
YOLOv3-tiny	COCO	Tf	416 x 416	NO	*	*
YOLOv5small	COCO	Tf	640 x 640	NO	*	*

Table 3.5: Models tested. * (Float ops were not possible to calculate, the Quantized accuracy in these two models was quite low as both were trained for only 5 epochs.)

models referred to in the table were the models implemented from scratch.

To complement the Table 3.5, here is the meaning of some abbreviations:

- Dk - Darknet.
- Pt - Pytorch.
- Tf - Tensorflow 1/2.
- Pruned - Percentage of the pruned model.(Pruning models - use Vitis-AI optimizer tool, reduce the complexity of the model).
- OFA - (Once for all) Optimized models, such as Super-Resolution OFA-Residual Channel Attention Networks (RCAN) and Object Detection OFA-YOLO.

To measure the energy consumption of this ten models, an inference of a 1280 x 720 image was made for 30 seconds in single thread. The figure 3.18 shows the energy consumption during this 30 second inference (with pre-trained model YOLOv4 pruned 0.36%), achieving 18.02 FPS.

3.5 Summary/Conclusions

Technical problems regarding the implementation of YOLO Neural Networks

To begin, the CPU and GPU implementations were the easiest to implement, however, it was difficult and time-consuming to find old versions of Python that supported all of the packages required for the implemented versions. When creating different Python environments for each network, they occasionally became inoperable due to package compatibility issues.

Compared to the Pynq framework, it had greater compatibility problems. As stated in section 3.2, it only supports Pytorch, so the majority of work was accomplished using these two frameworks. Finding all of the packages supported by this framework and the

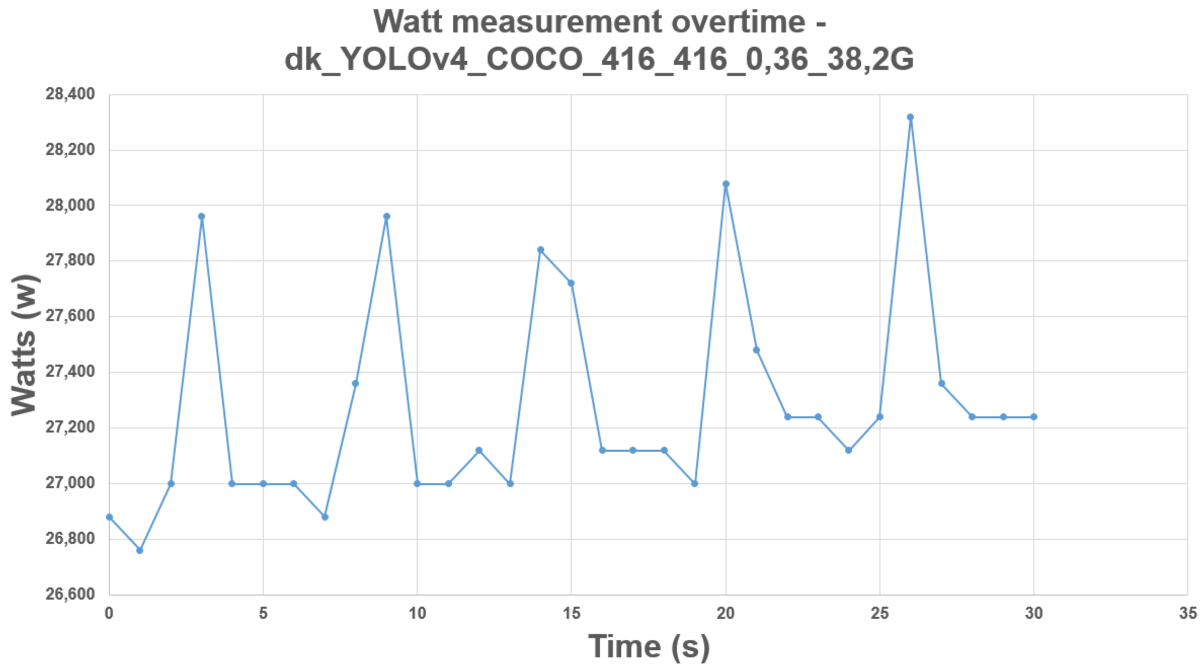

Figure 3.18: Watt measurement overtime - Dk_YOLOv4_COCO_416_416_0.36_38.2G.

YOLO Neural Networks used was a difficult task.

The framework that received the most attention was Vitis-AI. Despite the vast amount of information (available at [67]), there were numerous errors, the majority of which were not displayed in the console output. The fact that the Zynq UltraScale+ MPSoC ZCU104 DPU does not support some YOLOv5 NN operations was not discovered until the final stages of implementation, so the results for this implementation (YOLOv5), although it was possible, the results were not satisfactory. The YOLOv3-tiny was implemented in this framework, obtaining excellent results, 2.25 *frames/joule*.

Performance and energy consumption conclusions for the different implementations

YOLOv3/v5 Neural Networks were implemented on CPU, GPU, and Pynq framework. In the Vitis-AI framework, the YOLOv3-tiny and YOLOv5-small models were implemented from scratch. As well as checking the performance and energy consumption of the models present in the Model Zoo (Table 3.5).

Comparing FPGA and GPU implementations, it was possible to obtain more *frames per joule* using reconfigurable logic, proving that the same amount of data can be computed using less energy.

The FPGA implementation of the YOLOv3-tiny NN achieved 2.25 *frames/joule*, while the GPU implementation achieved 1.25 *frames/joule*.

All the results obtained with Neural Networks (NN) in the three types of hardware (CPU, GPU and FPGA) are present in Chapter 4 and the graphs of the energy consumed

in appendices C, D and E.

Conclusions regarding Pynq and Vitis-AI frameworks.

Although the Pynq framework is more user-friendly, the parallelization of YOLO Neural Networks is quite complex, and no solution was found, therefore, parallelization was not addressed using this framework.

The Vitis-AI framework provides the user with capabilities that help quantification, compilation, and deployment of these types of Neural Networks, despite the fact that this process is time-consuming and requires the knowledge of these tools and the operations that the DPU supports. Currently, it is concluded that the entire process of implementing YOLOv3 and YOLOv4 Neural Networks is feasible, with applications achieving more than 30 FPS. In relation to YOLOv5, the process is more complicated because the entire architecture must be redesigned and *scripts* must be created to test the model on the FPGA.

It is important to note that this framework was introduced at the beginning of 2020, and that over time, all the tools it provides have improved. As a result, it is expected that the entire process of implementing complex Neural Networks will become easier or even, with better performance results.

Chapter 4

Results and analysis

This Chapter presents the results obtained regarding the performance of the models tested on different types of hardware. To make it easier to analyze these results, tables were created for each type of hardware and framework.
In each subsection there is a plot of the power consumed over time for one of the models tested, the graphs of all the four models are presented in the appendices C, D and E. A,B present the packages used for YOLOv3,v5 and Pynq framework environments.

4.1 Raw performance

The results obtained were performed under the inference of a 1-minute 1280x720 video, totaling 3000 frames.
For each created/tested model, the execution time and power consumption during inference were measured. The average watts, FPS, kWh, and *frames/joule* were calculated based on these values.

4.1.1 CPU and GPU results

The Figure 4.1 shows the energy consumption of the YOLOv3 NN on CPU, obtaining an average of 129.68 Watts.
The following Table 4.1 shows the results of the four Neural Networks tested on the CPU and GPU. The most important metrics are: $mAP_{0.5}$, FPS, kWh and *frames/joule*.

Hw	Model	FLOPS	$mAP_{0.5}$	Time (s)	Avg Watts	FPS	kWh	*frames/joule*
CPU	v3	155.9G	0.661	1518.72	129.68 W	1.97	5.47×10^{-2}	0.01523
	v3-tiny	13.2G	0.367	267.96	127.46 W	11.19	9.49×10^{-3}	0.08783
	v5l	109.1G	0.675	1304.10	129.91 W	2.30	4.71×10^{-2}	0.01770
	v5s	16.5G	0.571	308.01	129.14 W	9.74	1.10×10^{-2}	0.07541
GPU	v3	155.9G	0.661	60.81	196.75 W	49.33	3.32×10^{-3}	0.25074
	v3-tiny	13.2G	0.367	17.81	128.19 W	168.44	6.34×10^{-4}	1.31402
	v5l	109.1G	0.675	50.86	199.08 W	58.98	2.81×10^{-3}	0.29629
	v5s	16.5G	0.571	25.71	159.45 W	116.65	1.14×10^{-3}	0.73157

Table 4.1: CPU and GPU results (Inference from a 1 minute video 1280x720 pixels with a total of 3000 frames).

Looking at table 4.1, it can be seen that the GPU has a higher energy efficiency compared to the CPU. Comparing the results obtained in GPU, between the YOLOv3 and

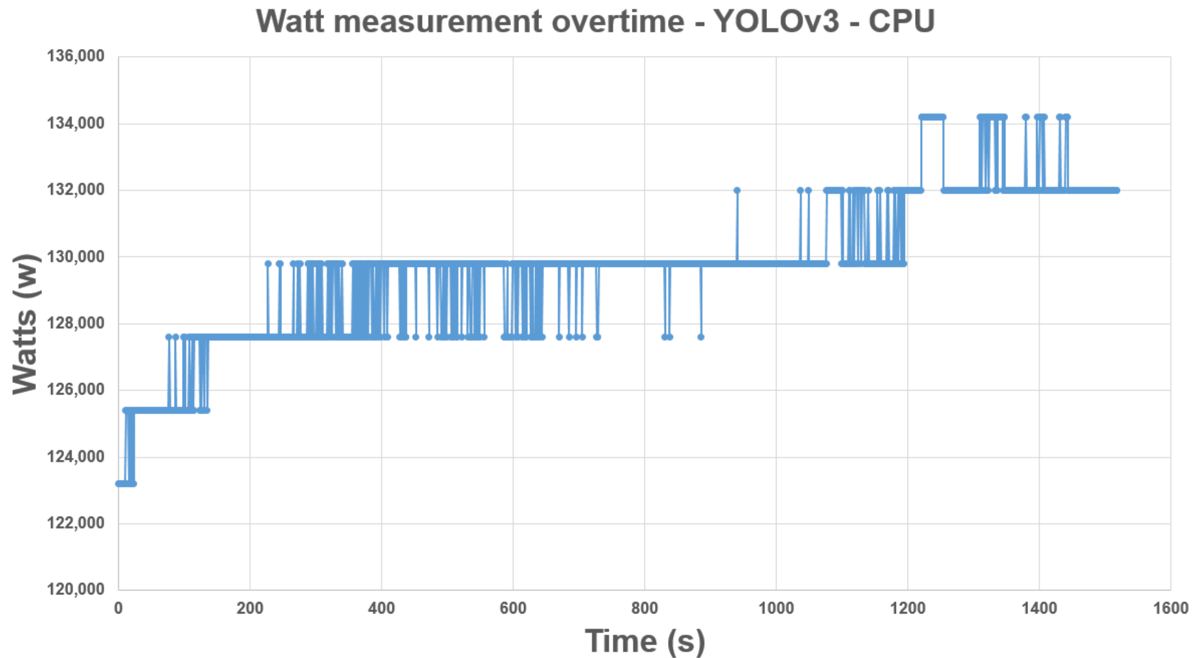

Figure 4.1: Power consumption measurement results, YOLOv3 on CPU.

YOLOv5l NN, which both present an identical $mAP_{0.5}$, we can see that the YOLOv5l NN performs the inference faster, obtaining more 9.65 FPS, while still spending less energy, obtaining higher *frames/joule* than the YOLOv3 NN.

In terms of lightweight versions of these two networks (YOLOv3-tiny and YOLOv5s), YOLOv3-tiny should get more FPS since this NN is generally faster than YOLOv5 versions. Although this NN had the best energy performance results, with lower kWh and higher *frames/joule* than all other Neural Networks, it should be noted that it only has 0.367 $mAP_{0.5}$, which is unsatisfactory for high precision implementations. With about 79% more energy, it is possible to have a solution (YOLOv5s) with significantly higher $mAP_{0.5}$, but getting the frame rate reduced by 30%.

4.1.2 FPGA results (Pynq framework)

The figure 4.2 shows the energy consumption of the YOLOv5l NN.

The following table 4.2 shows the results of the four Neural Networks tested on the Pynq framework. The most important metrics are: $mAP_{0.5}$, FPS, kWh and *frames/joule*.

Model	FLOPS	$mAP_{0.5}$	Time (s)	Avg Watts	FPS	kWh	*frames/joule*
YOLOv3	155.9G	0.661	23171.36	13.75 W	0.13	8.85×10^{-2}	0.00942
YOLOv3-tiny	13.2G	0.367	2441.14	13.51 W	1.23	9.16×10^{-3}	0.09097
YOLOv5l	109.1G	0.675	16965.2	13.71 W	0.18	6.46×10^{-2}	0.01290
YOLOv5s	16.5G	0.571	3653.45	13.44 W	0.82	1.36×10^{-2}	0.06109

Table 4.2: Pynq framework results (Inference from a 1 minute video 1280x720 pixels with a total of 3000 frames).

The results obtained with this framework were identical to those obtained in CPU implementations, obtaining a low energy efficiency in both cases. The maximum FPS

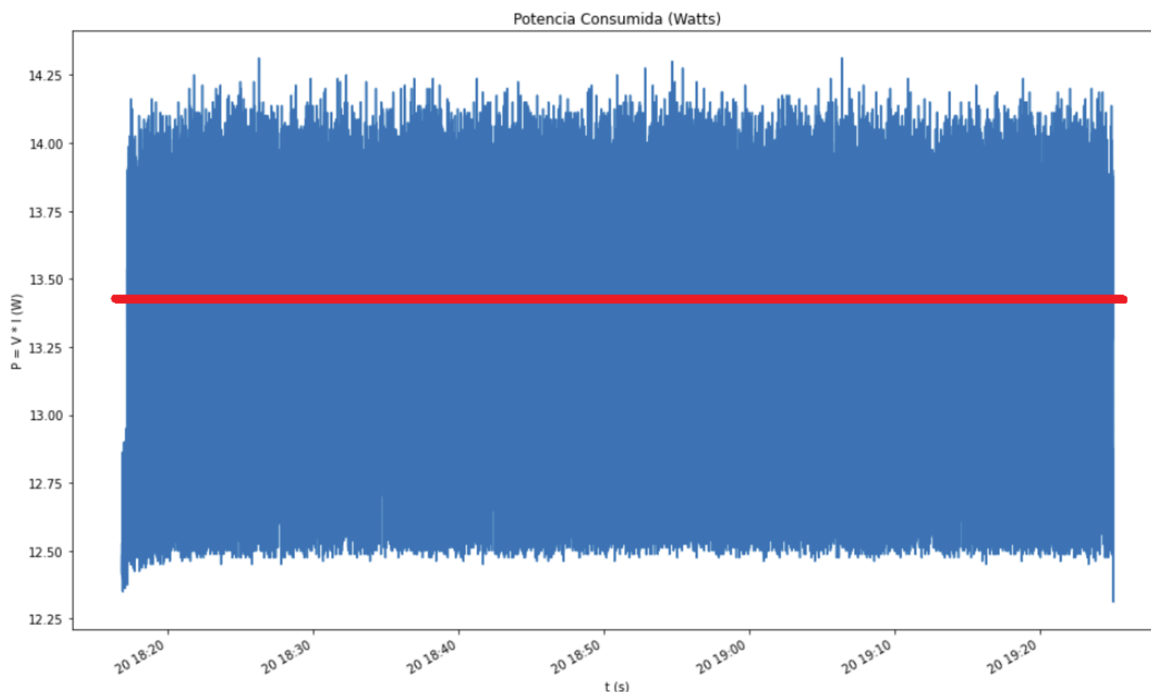

Figure 4.2: Power consumption measurement results, YOLOv5s on Pynq framework. The red line indicates the average watts (13.44 Watts).

obtained were 1.23 and 11.19 in the case of YOLOv3-tiny in Pynq framework and CPU implementations respectively, which is not ideal in real-time implementations.

4.1.3 FPGA results (Vitis-AI framework)

The following figure 4.3 shows the energy consumption of the NN "pt_OFA-YOLO_COCO_640_640_0.5_24.62G", during a 30-second inference under and image with a resolution of 1280x720 pixels.

During the tests of all Neural Networks mentioned in 3.5, inferences of 30 seconds were performed for images with 1280x720 pixels, with this to have accurate results under comparisons with CPU, GPU and Pynq framework.
In the table 4.3 (based on the obtained FPS result), the time it would take to infer 3000 frames was calculated, as well the energy that would be spent.

"Qacc" means Quantized accuracy, as in choosing these models, information is only available on this metric and not on the $mAP_{0.5}$.

To measure the mAP it is necessary to run some scripts that only work in the Vitis-AI environment on the GPU, as these environments were not working properly, it was not possible to evaluate this metric.

However, it is known that the model present in the table 4.3("tf_yolov4_coco_512_512") has a $mAP_{0.5}$ of 0.602, so it is expected that the others have a $mAP_{0.5}$ greater than the Quantized accuracy of about 45%.
Regarding the results obtained, it is possible to obtain a very interesting tradeoff with the OFA models, which are pruned and optimized by the tools present in Vitis-AI(optimizer).

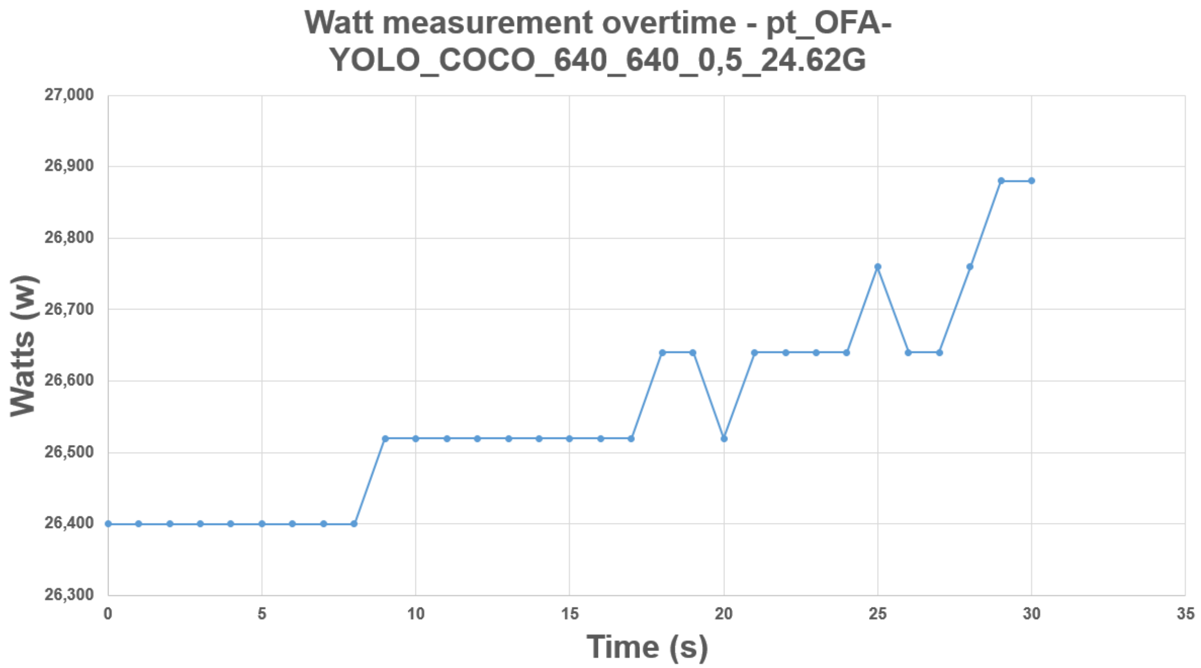

Figure 4.3: Power consumption measurement results, YOLOv4 OFA, pruned 0.5 on Vitis-AI framework.

Model	FLOPS	Qacc	Time (s)	Avg Watts	FPS	kWh	Frames/joule
dk_yolov4_coco_416_416_0.36	38.2G	NA	166.45	27.27	18.02	1.26×10^{-3}	0.66071
dk_yolov4_coco_416_416_	60.1G	NA	226.92	27.80	13.22	1.75×10^{-3}	0.47545
pt_OFA-yolo_coco_640_640_0.5	24.62G	0.378	105.88	26.55	28.33	7.81×10^{-4}	1.06699
pt_OFA-yolo_coco_640_640_0.3	34.72G	0.401	132.69	27.78	22.60	1.02×10^{-3}	0.81371
pt_OFA-yolo_coco_640_640	48.88G	0.421	171.93	28.66	17.44	1.37×10^{-3}	0.60874
tf2_yolov3_coco_416_416	65.9G	0.331	224.54	28.33	13.36	1.77×10^{-3}	0.47159
tf_yolov4_coco_416_416	60.3G	0.393	215.79	28.38	13.90	1.70×10^{-3}	0.48976
tf_yolov4_coco_512_512	91.2G	0.412	287.98	29.42	10.41	2.35×10^{-3}	0.35404
YOLOv3-tiny_coco_416_416	*	*	**51.78**	**25.70**	**57.93**	3.70×10^{-4}	**2.25347**
YOLOv5-small_coco_640_640	*	*	**4030.09**	**26.10**	**0.744**	2.92×10^{-2}	**0.02851**

Table 4.3: Vitis-AI results (Inference from a 1 minute video 1280 x 720 pixels with a total of 3000 frames).

Concerning the results of the implementations carried out from scratch YOLOv3-tiny and YOLOv5-small (highlighted in Table 4.3), the YOLOv3-tiny achieved the highest energy performance. As the YOLOv5 NN is improperly parallelized, inadequate results are obtained.

4.2 Comparison between hardware tested

The NN that stood out the most in terms of energy performance was YOLOv3-tiny on GPU and FPGA obtaining 1.314 and 2.253 *frames/joule* respectively.

The OFA (Once For all) models also exhibit a positive tradeoff between *frames/joule* and mAP, since, for instance, the "pt OFA-yolo_coco_640_640_0.5" model would exhibit a mAP of arround 0.57 (0.20 more than the YOLOv3-tiny NN), and while this model obtains 1.067 *frames/joule*.

With YOLOv5 models (in the Vitis-Ai framework) properly quantized and optimized, it would be possible to have a much better $mAP_{0.5}$, with a (expected) slight decrease in energy consumed.

4.3 Draw conclusions concerning Neural Networks YOLOv3, v4, v5

Figures 4.4, 4.5, 4.6 and 4.7 present the final result of the detection in an image of the YOLOv3/v3-tiny and YOLOv5l/v5s Neural Networks.

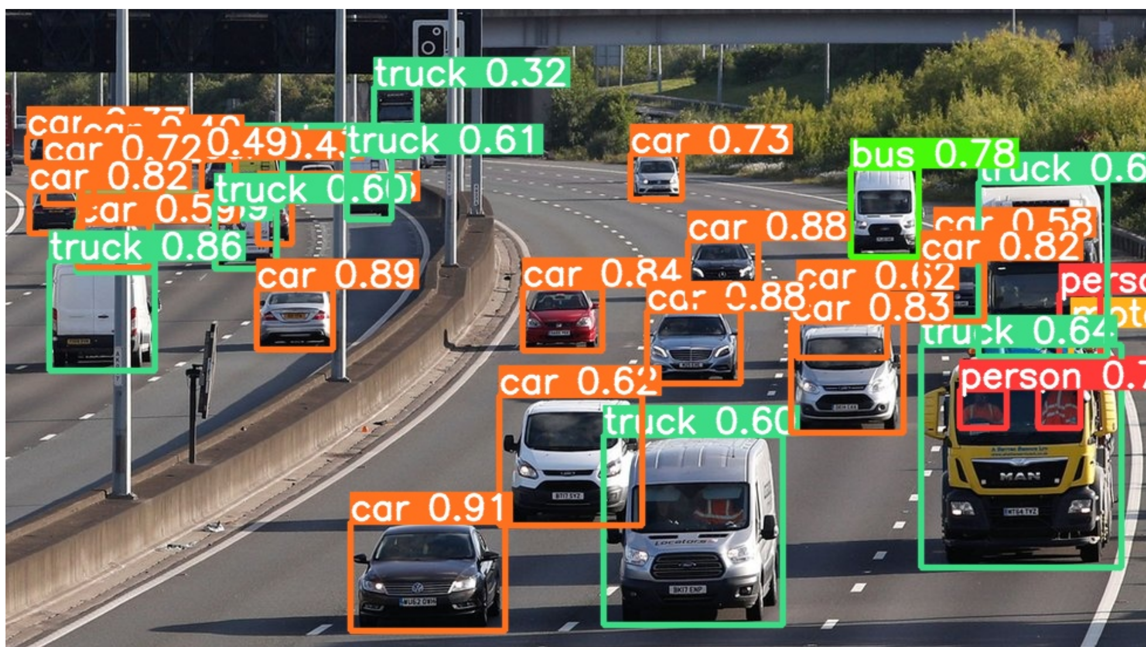

Figure 4.4: Inference results - YOLOv3.

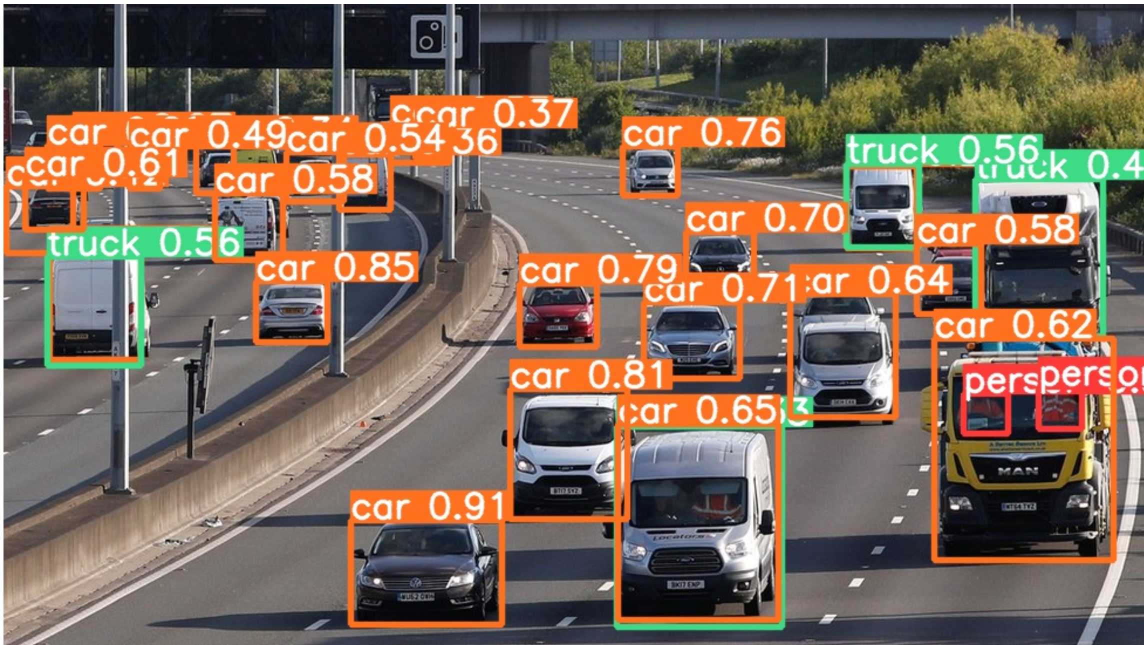

Figure 4.5: Inference results - YOLOv3-tiny.

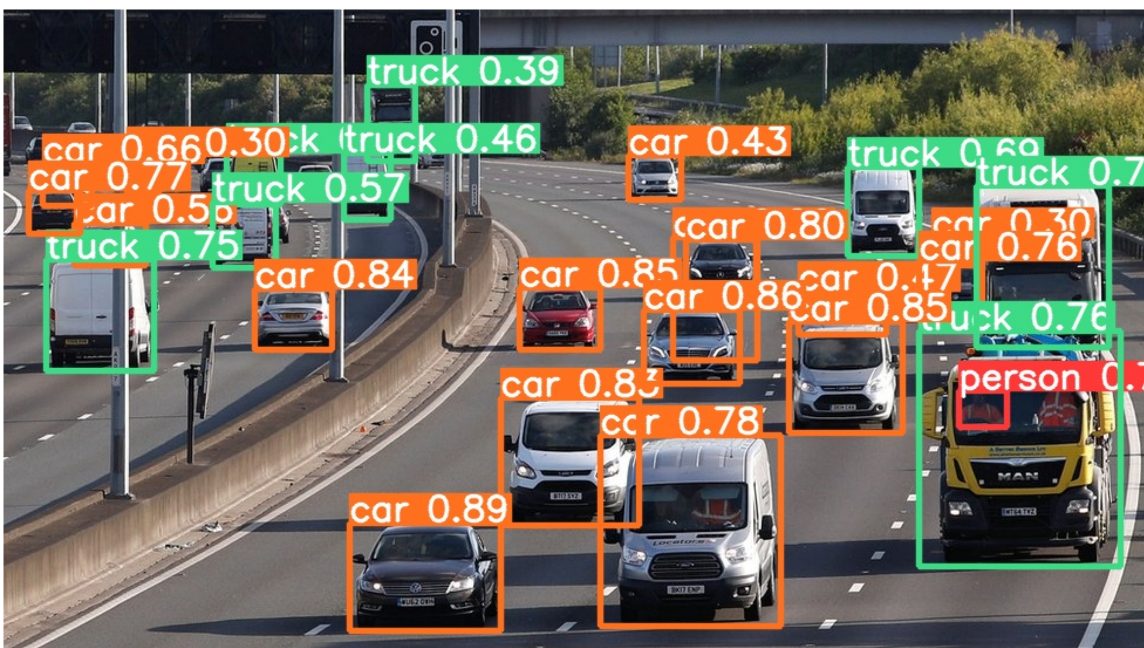

Figure 4.6: Inference results - YOLOv5l.

Given that both networks (YOLOv3 and YOLOv5l) have identical mAPs, the results differ only slightly. YOLOv3, like YOLOv5, occasionally misidentifies certain objects. However, both Neural Networks correctly and accurately detect the majority of vehicles, with YOLOv5l performing the inference faster (As can be seen in Table 4.1).

Lighter Neural Networks (YOLOv3-tiny and YOLOv5s) sometimes detect multiple objects when only one should be detected, which is not ideal, but considering the Neural Networks are quite light, a very positive tradeoff is obtained.

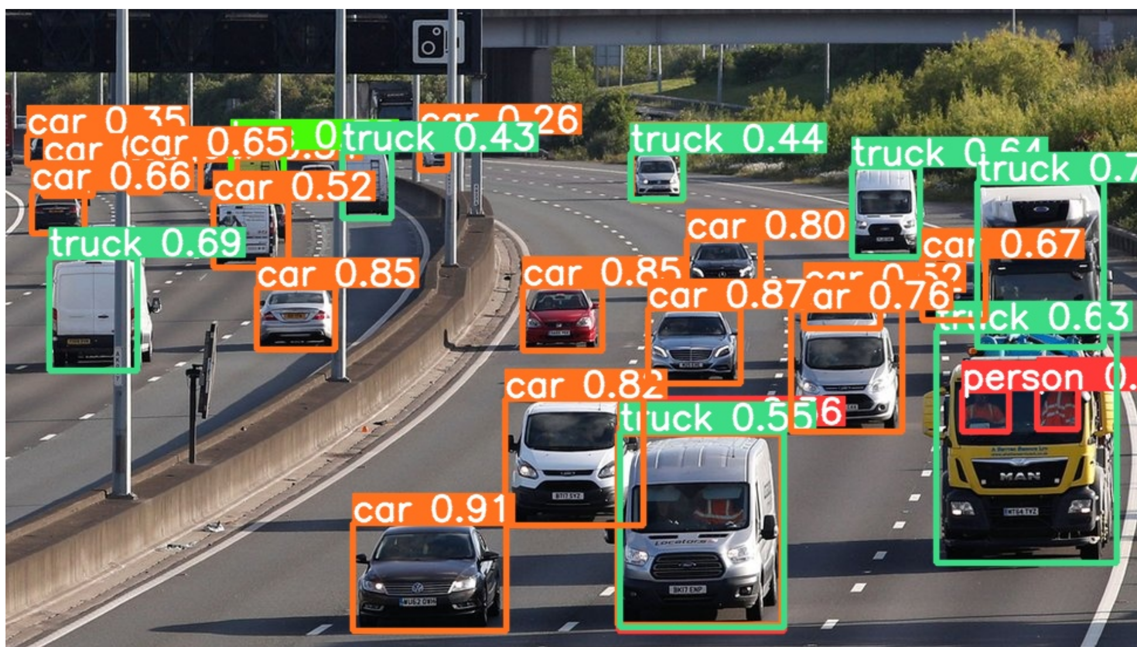

Figure 4.7: Inference results - YOLOv5s.

In this implementation with the COCO-dataset, our results show, in addition to having a $mAP_{0.5}$ larger than YOLOv3, the YOLOv5l NN performs inference 1.19 times faster while consuming 15.36% less energy.

4.4 Discussion and notes

The results show that the implementations in CPU and Pynq framework are not suitable for real-time applications, since the FPS obtained are quite low, and their energy efficiency is also low.

It should be noted that all inferences were performed with videos of a significant resolution (1280x720 pixels), which ultimately results in less FPS than these Neural Networks are usually able to obtain, without performing any kind of pre-processing (simulating raw results that would be obtained if images obtained from surveillance cameras/underwater images were directly processed by the Neural Networks). To obtain functional implementations in real-time applications(at least 30 FPS), it is necessary to use GPU or implementations in FPGAs.

It is important to think about real-time applications in harsh environments (for example: underwater environments, extreme weather environments). GPUs in these situations would not be ideal unless they were low power (for example: NVIDIA Jetson Xavier NX, able to get 35.6 FPS [68] with YOLOv3-tiny NN). GPUs (identical to NVIDIA RTX 3060) are not suitable for this type of applications, given their higher power requirements and energy consumption not to mention additional issues like heat dissipation.

Due to all the benefits that FPGA bring, they are suitable for these environments, with a lower energy consumption, as can be seen by comparing Table 4.1 Table 4.3. Table 4.4 shows the results of implemented Neural Networks on FPGA and GPU that are considered energy-efficient and achieve a respectable number of FPS, taking their energy efficiency into account.

Hardware	Model	Avg Watts	FPS	kWh	frames/joule
GPU	YOLOv3-tiny	128.19	168.44	6.34e-4	1.31402
	YOLO5s	159.45	116.65	1.14e-3	0.73157
FPGA Vitis-AI Framework	dk_YOLOv4_416_0.36	27.27	18.02	1.26e-3	0.66071
	pt_OFA-YOLO_640_0.5	26.55	28.33	7.81e-4	1.06699
	pt_OFA-YOLO_640_0.3	27.78	22.60	1.02e-3	0.81371
	pt_OFA-YOLO_640	28.66	17.44	1.37e-3	0.60874
	YOLOv3-tiny_416	25.70	57.93	3.70e-4	2.25347

Table 4.4: Implementations on GPU and FPGA yielding greater energy efficiency results.

The results of the Table 4.4 are illustrated in the Figure 4.8 for better visualization.

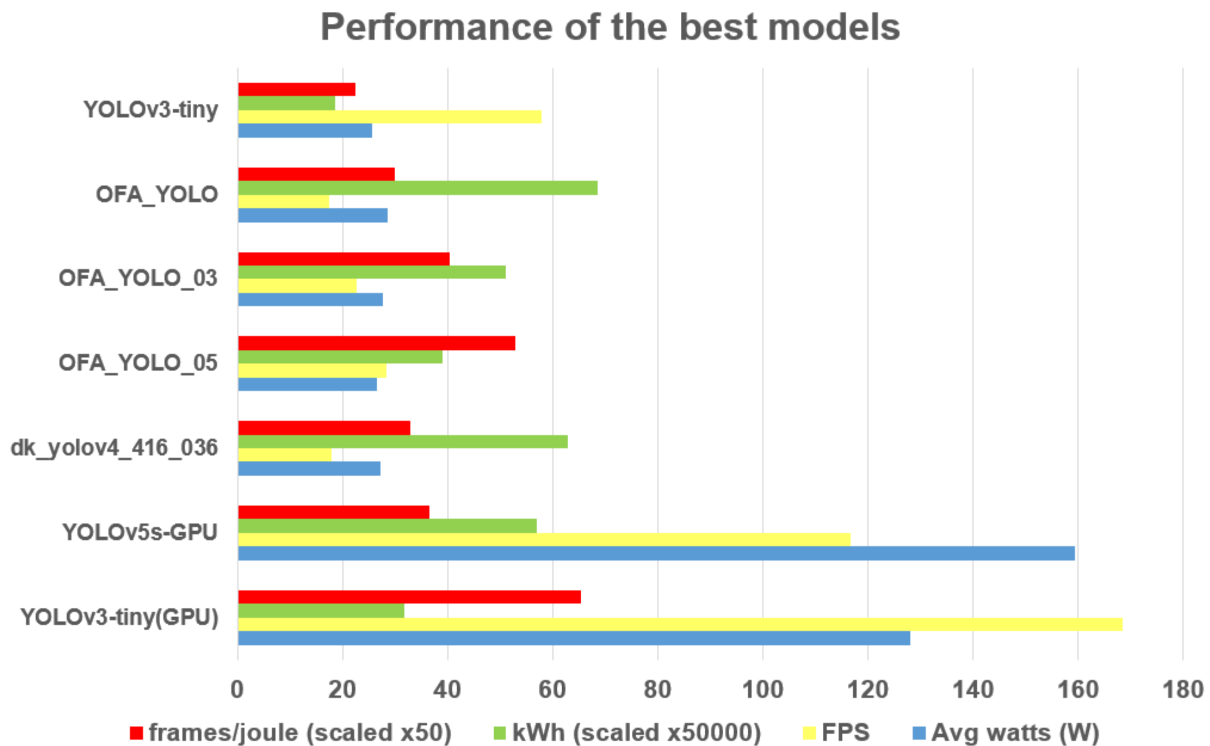

Figure 4.8: Performance of the best models.

Due to temporal constraints, it was not possible to measure the wattage of the "YOLOv5n" NN. However, this one has a $mAP_{0.5}$ of 0.457 (YOLOv3-tiny has a $mAP_{0.5}$ of 0.367) and achieves 135.15 FPS in the same inference.

The results presented in Table 4.4 and Figure 4.8 show that we have interesting implementations that support support YOLOv3,v4,v5 Neural Networks using FPGAs.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

The objective of this dissertation was to explore implementations of YOLO Neural Networks in different types of hardware and to analyze the most efficient solutions, considering the purpose of the application (analyzing vehicle traffic in real time), clearly this work is not restricted to this application, but rather to all others that may involve the use of these Neural Networks.

Table 5.1 summarizes the Neural Networks implemented in different types of hardware, as well as FPS, average watts, kWh and *frames/joule* results.

Hardware	Model	$mAP_{0.5}$	Avg Watts	FPS	kWh	*frames/joule*
CPU	YOLOv3	0.661	129.68 W	1.97	5.47×10^{-2}	0.01523
	YOLOv3-tiny	0.367	127.46 W	11.19	9.49×10^{-3}	0.08783
	YOLOv5l	0.675	129.91 W	2.30	4.71×10^{-2}	0.01770
	YOLOv5s	0.571	129.14 W	9.74	1.10×10^{-2}	0.07541
GPU	YOLOv3	0.661	196.75 W	49.33	3.32×10^{-3}	0.25074
	YOLOv3-tiny	0.367	128.19 W	168.44	6.34×10^{-4}	1.31402
	YOLO5l	0.675	199.08 W	58.98	2.81×10^{-3}	0.29629
	YOLO5s	0.571	159.45 W	116.65	1.14×10^{-3}	0.73157
FPGA Pynq Framework	YOLOv3	0.661	13.75 W	0.13	8.85×10^{-2}	0.00942
	YOLOv3-tiny	0.367	13.51 W	1.23	9.16×10^{-3}	0.09097
	YOLOv5l	0.675	13.71 W	0.18	6.46×10^{-2}	0.01290
	YOLOv5s	0.571	13.44 W	0.82	1.36×10^{-2}	0.06109
FPGA Vitis-AI Framework	dk_YOLOv4_416_0.36	unmeasured	27.27 W	18.02	1.26×10^{-3}	0.66071
	dk_YOLOv4_416	unmeasured	27.80 W	13.22	1.75×10^{-3}	0.47545
	pt_OFA-YOLO_640_0.5	unmeasured	26.55 W	28.33	7.81×10^{-4}	1.06699
	pt_OFA-YOLO_640_0.3	unmeasured	27.78 W	22.60	1.02×10^{-3}	0.81371
	pt_OFA-YOLO_640	unmeasured	28.66 W	17.44	1.37×10^{-3}	0.60874
	tf2_YOLOv3_416	unmeasured	28.33 W	13.36	1.77×10^{-3}	0.47159
	tf_YOLOv4_416	unmeasured	28.38 W	13.90	1.70×10^{-3}	0.48976
	tf_YOLOv4_512	unmeasured	29.42 W	10.41	2.35×10^{-3}	0.35404
	YOLOv3-tiny_416	unmeasured	25.70 W	57.93	3.70×10^{-4}	2.25347
	YOLOv5-small_640	unmeasured	26.10 W	0.744	2.92×10^{-2}	0.02851

Table 5.1: Results obtained from all implementations performed.

For a better visualization of the results, Figures 5.1 and 5.2 illustrate the results of table 5.1.

Table 5.2 indicates the nominal power of the systems at idle, as well as their theoretical maximum value.

Hardware	Nominal power ("idle mode")	Max nominal power (theoretical)	Nominal power obtained with our implementations
CPU	90∼100 W	200∼217 W	125∼150 W
GPU	90∼100 W	340∼360 W	125∼205 W
FPGA - Pynq	11∼12 W	48∼60 W	12∼16 W
FPGA- Vitis	17∼19 W	48∼60 W	25∼30 W

Table 5.2: Nominal power of CPU, GPU and FPGA systems at idle mode, theoretical maximum value and the range obtained in all implementations performed .

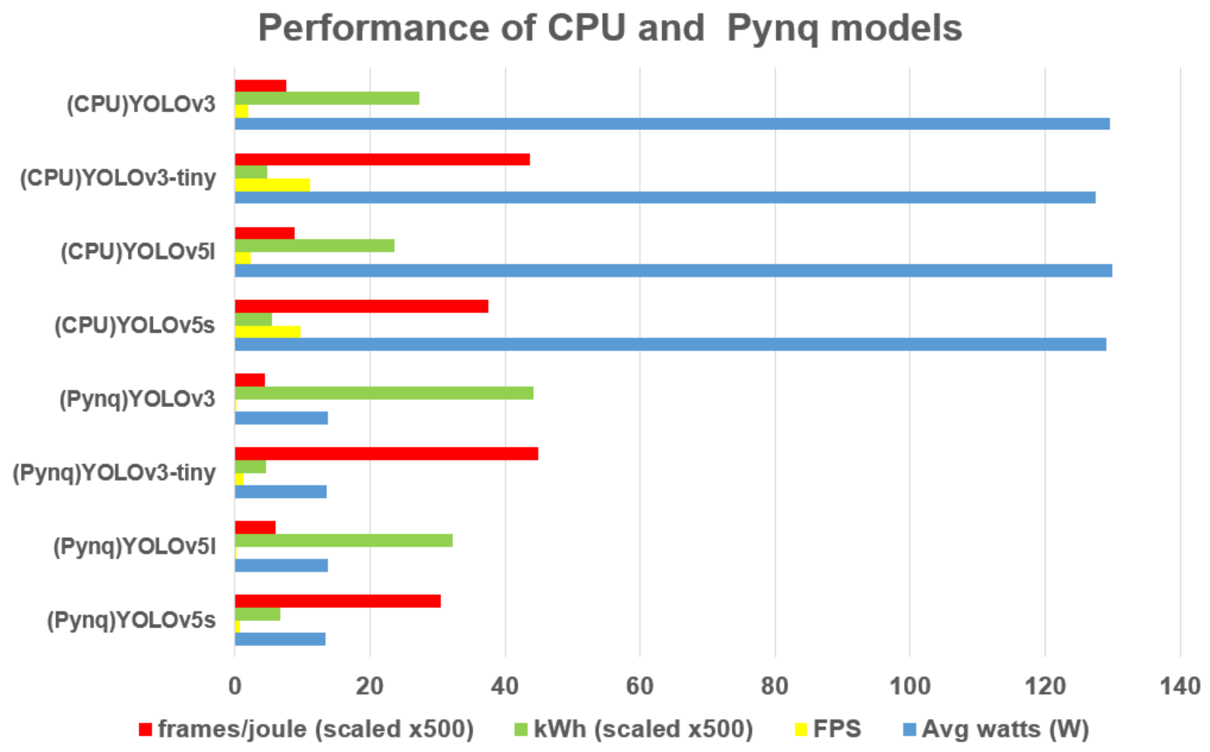

Figure 5.1: Performance of models implemented with CPU and Pynq framework(FPGA).

It is important to remember that the tests were performed for the COCO dataset, wich contains approximately 118k images and 80 classes. Consequently, simpler architectures, such as YOLOv3-tiny, have a lower $mAP_{0.5}$ (0.367) than YOLOv5-small, which has a higher $mAP_{0.5}$ (0.57).
So the choice of the NN to implement in real-time applications has to be adjusted to the dataset to be used, for instance, using a dataset containing the desired classes (cars, motorcycles, trucks, buses, scooters and bicycles) would result in a significantly higher $mAP_{0.5}$. Moreover, GPU comparisons of YOLOv3 and YOLOv5l (Neural Networks with similar $mAP_{0.5}$) reveal that the YOLOv5l NN uses less energy to compute the same number of frames, so it is advantageous to use this architecture for real-time applications.

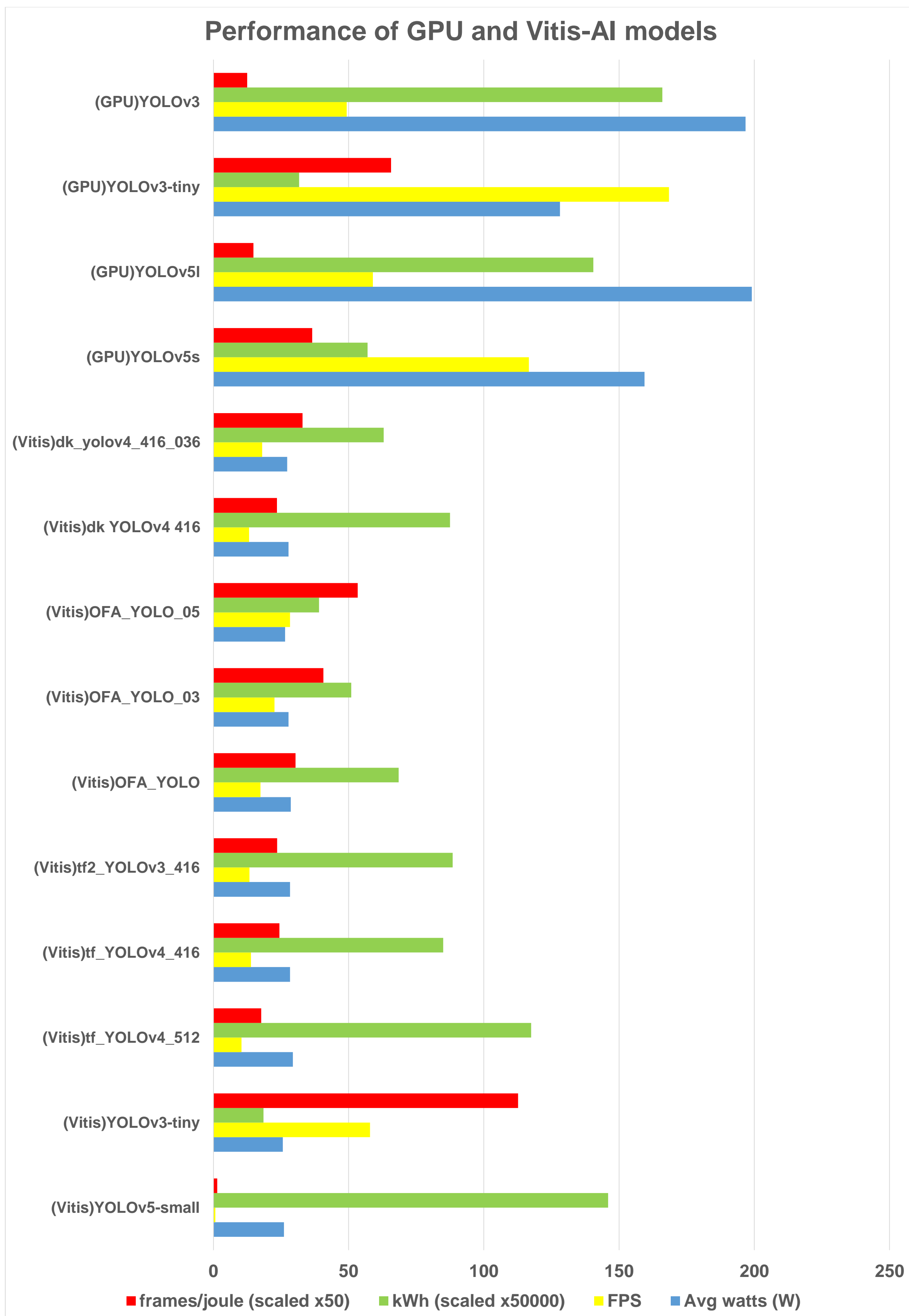

Figure 5.2: Performance of the best models (GPU and FPGA). Note: In Vitis-AI YOLOv5, the kWh are only scaled to 500X.

Regarding the results obtained, implementations in CPU and FPGA (Pynq framework), these are not energy efficient due to their extremely low *frames/joule* value. Using the YOLOv3-tiny NN as a reference, a maximum of 11.19 FPS was obtained on CPU and 1.23 FPS on Pynq, wich are unsatisfactory results for real-time applications.

In terms of FPS and *frames/joule*, GPU and FPGA (Vitis-AI framework) implementations produced the best results. Once again using the YOLOv3-tiny NN as a reference, on GPU, it was possible to achieve 168.44 FPS and on FPGA, 57.93 FPS. In terms of energy efficiency, the FPGA is 1.78 times more efficient than the GPU implementation, achieving 2.25 *frames/joule* compared to 1.31 *frames/joule* for the GPU implementation.

Regarding the Model Zoo pre-trained models (the first eight models listed in Table 5.1 of the Vitis-ai framework), they generally obtained good energy efficiency results. The two Once For All (OFA) models (pruned at 0.5 and 0.3) take full advantage of the quantization and optimization tools present in this framework, these models produce the best *frame/joules* results.

It was found that it is possible to have very efficient FPGA (Vitis-AI) implementations, that can be applied in low power applications, such as robots, however, the entire implementation process is quite complex, and the framework support is still relatively limited considering the most recent Neural Networks(YOLOv5,v6,v7) architectures.

5.2 Future Work

The following list summarizes future work within the scope of this dissertation objectives:

- Test implementations using a suitable data set (for instance, Open Images v6 [69] using the desired classes).
- Modify the latest YOLOv5/v6/v7 Neural Networks to be compatible with Xilinx DPU(Xilinx Zynq® UltraScale+™ Deep Learning Processor- DPU-DPUCZDX8G).
- Perform training, quantization, evaluation, optimization and compilation of this newer architectures. With Vitis-ai Optimizer, perform model pruning and optimize the created models without compromising precision.
- Implementations of YOLO Neural Networks in the FINN framework [26].
- Perform power measurements on all created implementations and compare to GPU implementations.

With all the results obtained, choose the most beneficial NN for low power implementations.

When this essential part of the project is completed, a application from scratch must be developed. One of the goals of this application is to perform the vehicle count through a "*imaginary line*" using the various object detections it performs in real time, in an attempt

to reduce vehicle traffic and accidents. In addition, this work may have other purposes for future work, such as obtaining the instantaneous speed of vehicles and tracking them.

The ultimate goal of this dissertation would be to develop a functional prototype to be implemented in reconfigurable logic, with the intention of being implemented in areas where there is significant vehicle movement. This prototype would be able to detect and classify different types of vehicles (cars, motorcycles, trucks, buses, scooters and bicycles), perform their accounting throughout the day while being powered by batteries and small solar panels.

Bibliography

[1] PYNQ: PYTHON PRODUCTIVITY Website. http://www.pynq.io. Accessed: 2022-04-27.

[2] Xilinx Vitis AI framework Website. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html. Accessed: 2022-06-27.

[3] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. A survey of deep learning-based object detection. 7 2019.

[4] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. Deep learning for generic object detection: A survey. 9 2018.

[5] Abdul Vahab, Maruti S Naik, Prasanna G Raikar, and Prasad S R. Applications of object detection system. *International Research Journal of Engineering and Technology*, page 4186, 2008.

[6] Pedro R. Miranda, Daniel Pestana, João D. Lopes, Rui Policarpo Duarte, Mário P. Véstias, Horácio C. Neto, and José T. de Sousa. Configurable hardware core for iot object detection. *Future Internet*, 13, 11 2021.

[7] Fact Sheet, Ladot, Los Angeles Signal Sync. https://ladot.lacity.org/sites/default/files/documents/ladot-atsac-signals-_-fact-sheet-2-14-2016.pdf. Accessed: 2022-04-17.

[8] Stefanini Group, News. https://stefanini.com/en/trends/news/top-5-applications-of-iot-in-building-smart-cities. Accessed: 2022-07-20.

[9] Ashwin Bhandare, Maithili Bhide, Pranav Gokhale, and Rohan Chandavarkar. Applications of convolutional neural networks.

[10] Tianxu Yue. Convolutional neural network fpga-accelerator on intel de10-standard fpga, 2021.

[11] Ruiqi Chen, Tianyu Wu, Yuchen Zheng, and Ming Ling. Mlof: Machine learning accelerators for the low-cost fpga platforms. *Applied Sciences (Switzerland)*, 12, 1 2022.

[12] Roberto Di Cecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. Caffeinated fpgas: Fpga framework for convolutional neural networks. pages 265–268. Institute of Electrical and Electronics Engineers Inc., 5 2017.

[13] Sujata Joshi, Saksham Saxena, Tanvi Godbole, and Shreya. Developing smart cities: An integrated framework. volume 93, pages 902–909. Elsevier B.V., 2016.

[14] United Nations Department of Economic and Social Affairs. https://www.un.org/development/desa/en/news/population/2018-revision-of-world-urbanization-prospects.html. Accessed: 2022-09-02.

[15] H. M.K.K.M.B. Herath and Mamta Mittal. Adoption of artificial intelligence in smart cities: A comprehensive review. *International Journal of Information Management Data Insights*, 2, 4 2022.

[16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3:637–646, 10 2016.

[17] Li Du, Yuan Du, Yilei Li, Junjie Su, Yen Cheng Kuan, Chun Chen Liu, and Mau Chung Frank Chang. A reconfigurable streaming deep convolutional neural network accelerator for internet of things. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65:198–208, 1 2018.

[18] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[19] MS COCO dataset Website. https://cocodataset.org. Accessed: 2022-03-17.

[20] Upesh Nepal and Hossein Eslamiat. Comparing yolov3, yolov4 and yolov5 for autonomous landing spot detection in faulty uavs. *Sensors*, 22, 1 2022.

[21] Walther Carballo-Hernández, Maxime Pelcat, and François Berry. Why is fpga-gpu heterogeneity the best option for embedded deep neural networks?

[22] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Gee Hock Ong, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can fpgas beat gpus in accelerating next-generation deep neural networks? pages 5–14. Association for Computing Machinery, Inc, 2 2017.

[23] P Y K Cheung. Project title: Pynq classification-python on zynq fpga for neural networks.

[24] CIFAR-10 dataset Website. https://www.cs.toronto.edu/~kriz/cifar.html. Accessed: 2022-04-28.

[25] Y LeCun, B Boser, J S Denker, D Henderson, R E Howard, W Hubbard, and L D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 12 1989.

[26] Xilinx FINN framework github. https://xilinx.github.io/finn/. Accessed: 2022-04-22.

[27] Michal Machura, Michal Danilowicz, and Tomasz Kryjak. Embedded object detection with custom littlenet, finn and vitis ai dcnn accelerators. *Journal of Low Power Electronics and Applications*, 12:30, 5 2022.

[28] Vitis-AI Model Zoo. https://github.com/Xilinx/Vitis-AI/tree/master/model_zoo/model-list. Accessed: 2022-06-27.

[29] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. 12 2015.

[30] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. 6 2015.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks.

[32] ImageNet dataset Website. https://www.image-net.org. Accessed: 2022-03-05.

[33] Medium Website, Binary Image classifier CNN using TensorFlow. https://medium.com/techiepedia/binary-image-classifier-cnn-using-tensorflow-a3f5d6746697. Accessed: 2022-05-22.

[34] Valentina Emilia Balas, Nikos E Mastorakis, Marius-Constantin Popescu, and Valentina E Balas. Multilayer perceptron and neural networks soft computing applications view project elastic waves view project multilayer perceptron and neural networks, 2009.

[35] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, J. Santamaría, Mohammed A. Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8, 12 2021.

[36] upgrade Website, CNN architecture. https://cs231n.github.io/convolutional-networks/. Accessed: 2022-02-25.

[37] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors.

[38] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. 6 2015.

[39] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. 11 2013.

[40] Ross Girshick. Fast r-cnn. 4 2015.

[41] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. 4 2020.

[42] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger.

[43] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. 4 2018.

[44] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection.

[45] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. 4 2014.

[46] Zhi Zhang, Tong He, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of freebies for training object detection neural networks. 2 2019.

[47] Ultralytics Website . `https://ultralytics.com`. Accessed: 2022-03-20.

[48] Ultralytics, YOLOv5 Github . `https://github.com/ultralytics/yolov5`. Accessed: 2022-03-20.

[49] Renjie Xu, Haifeng Lin, Kangjie Lu, Lin Cao, and Yunfei Liu. A forest fire detection system based on ensemble learning. *Forests*, 12:1–17, 2 2021.

[50] Lu Tan, Tianran Huangfu, and Liyao Wu. Comparison of yolo v3, faster r-cnn, and ssd for real-time pill identiication. 2021.

[51] Min Li, Zhijie Zhang, Liping Lei, Xiaofan Wang, and Xudong Guo. Agricultural greenhouses detection in high-resolution satellite images based on convolutional neural networks: Comparison of faster r-cnn, yolo v3 and ssd. *Sensors (Switzerland)*, 20:1–14, 9 2020.

[52] Xiang Long, Kaipeng Deng, Guanzhong Wang, Yang Zhang, Qingqing Dang, Yuan Gao, Hui Shen, Jianguo Ren, Shumin Han, Errui Ding, and Shilei Wen. Pp-yolo: An effective and efficient implementation of object detector. 7 2020.

[53] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021. 7 2021.

[54] Sahla Muhammed Ali. Comparative analysis of yolov3, yolov4 and yolov5 for sign language detection.

[55] Shrey Srivastava, Amit Vishvas Divekar, Chandu Anilkumar, Ishika Naik, Ved Kulkarni, and V. Pattabiraman. Comparative analysis of deep learning image detection algorithms. *Journal of Big Data*, 8, 12 2021.

[56] Unit-T UT803 Website, drivers. `https://instruments.uni-trend.com/download?name=164&scode=80&series=UT800`. Accessed: 2022-07-15.

[57] Tensorflow framework Website. `https://www.tensorflow.org`. Accessed: 2022-04-10.

[58] Pytorch framework Website. `https://pytorch.org`. Accessed: 2022-03-11.

[59] Pixbay Website, link to the video used for inferences. `https://pixabay.com/videos/car-road-transportation-vehicle-2165/`. Accessed: 2022-04-20.

[60] Xilinx Website . `https://www.xilinx.com`. Accessed: 2022-03-02.

[61] Cathal mccabe wrc 2020.

[62] Jupyter Lab Website . `https://jupyter.org`. Accessed: 2022-04-12.

[63] Pytorch Website. https://caffe.berkeleyvision.org. Accessed: 2022-06-27.

[64] Xilinx Vitis-AI Documentation Portal,DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide (PG338). https://docs.xilinx.com/r/en-US/pg338-dpu/Vitis-AI-Development-Kit. Accessed: 2022-06-15.

[65] Vitis-AI Design Tutorials, Github Website . https://github.com/Xilinx/Vitis-AI-Tutorials/tree/1.4/Design_Tutorials/07-yolov4-tutorial. Accessed: 2022-07-02.

[66] Keras YOLOv3 model set Github . https://github.com/david8862/keras-YOLOv3-model-set. Accessed: 2022-06-27.

[67] Vitis AI user guide . https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/Installing-vai_q_tensorflow. Accessed: 2022-07-02.

[68] Haogang Feng, Gaoze Mu, Shida Zhong, Peichang Zhang, and Tao Yuan. Benchmark analysis of yolo performance on edge intelligence devices. *Cryptography*, 6, 6 2022.

[69] Open Images Dataset V6, Website. https://storage.googleapis.com/openimages/web/download.html. Accessed: 2022-09-09.

Appendix A

List of YOLOv3/v5 environment packages (Desktop)

This appendix provides a list of packages required for YOLOv3 and YOLOv5 Neural Networks implementations on CPU and GPU. Each has a distinct Python virtual environment, version 3.7.0. (Note: although the Pytorch CUDA version is 11.1, the installed CUDA version was 11.3, which is functional.)

YOLOv3 packages

```
   Package                 Version
----------------------- --------------------
absl-py                  1.2.0
cachetools               5.2.0
certifi                  2022.6.15
charset-normalizer       2.1.0
click                    8.1.3
colorama                 0.4.5
cycler                   0.11.0
docker-pycreds           0.4.0
fonttools                4.34.4
gitdb                    4.0.9
GitPython                3.1.27
google-auth              2.9.1
google-auth-oauthlib     0.4.6
grpcio                   1.47.0
idna                     3.3
importlib-metadata       4.12.0
kiwisolver               1.4.4
Markdown                 3.4.1
MarkupSafe               2.1.1
matplotlib               3.5.2
numpy                    1.21.6
oauthlib                 3.2.0
opencv-python            4.6.0.66
packaging                21.3
pandas                   1.1.5
```

```
pathtools               0.1.2
Pillow                  9.2.0
pip                     22.1
promise                 2.3
protobuf                3.19.4
psutil                  5.9.1
pyasn1                  0.4.8
pyasn1-modules          0.2.8
pyparsing               3.0.9
python-dateutil         2.8.2
pytz                    2022.1
PyYAML                  6.0
requests                2.28.1
requests-oauthlib       1.3.1
rsa                     4.9
scipy                   1.7.3
seaborn                 0.11.2
sentry-sdk              1.9.0
setproctitle            1.3.0
setuptools              62.3.2
shortuuid               1.0.9
six                     1.16.0
smmap                   5.0.0
tensorboard             2.9.1
tensorboard-data-server 0.6.1
tensorboard-plugin-wit  1.8.1
thop                    0.1.1.post2207130030
torch                   1.8.2+cu111
torchaudio              0.8.2
torchvision             0.9.2+cu111
tqdm                    4.64.0
typing_extensions       4.3.0
urllib3                 1.26.11
wandb                   0.13.0
Werkzeug                2.2.1
wheel                   0.37.1
zipp                    3.8.1
```

YOLOv5 packages

```
    Package                      Version
---------------------------- ---------------------
absl-py                      1.0.0
argon2-cffi                  21.3.0
argon2-cffi-bindings         21.2.0
astunparse                   1.6.3
attrs                        22.1.0
backcall                     0.2.0
beautifulsoup4               4.11.1
```

bleach	5.0.1
cachetools	5.1.0
certifi	2022.5.18
cffi	1.15.1
charset-normalizer	2.0.12
click	8.0.0
colorama	0.4.4
coloredlogs	15.0.1
cvu-python	0.0.1a1
cycler	0.11.0
Cython	0.29.30
debugpy	1.6.2
decorator	5.1.1
defusedxml	0.7.1
entrypoints	0.4
fastjsonschema	2.16.1
flatbuffers	1.12
fonttools	4.33.3
gast	0.4.0
google-auth	2.6.6
google-auth-oauthlib	0.4.6
google-pasta	0.2.0
GPUtil	1.4.0
gputils	1.0.6
grpcio	1.46.1
h5py	3.7.0
humanfriendly	10.0
idna	3.3
imageio	2.19.5
importlib-metadata	4.11.3
importlib-resources	5.9.0
ipykernel	6.15.1
ipython	7.34.0
ipython-genutils	0.2.0
ipywidgets	7.7.1
jedi	0.18.1
Jinja2	3.1.2
jsonschema	4.8.0
jupyter	1.0.0
jupyter-client	7.3.4
jupyter-console	6.4.4
jupyter-core	4.11.1
jupyterlab-pygments	0.2.2
jupyterlab-widgets	1.1.1
keras	2.9.0
Keras-Preprocessing	1.1.2
kiwisolver	1.4.2
libclang	14.0.1

```
Markdown                      3.3.7
MarkupSafe                    2.1.1
matplotlib                    3.5.2
matplotlib-inline             0.1.3
merge-args                    0.1.4
mistune                       0.8.4
mpmath                        1.2.1
nbclient                      0.6.6
nbconvert                     6.5.0
nbformat                      5.4.0
nest-asyncio                  1.5.5
networkx                      2.6.3
notebook                      6.4.12
numpy                         1.21.6
oauthlib                      3.2.0
onnx                          1.10.1
onnx2torch                    1.4.1
onnxruntime                   1.12.0
onnxruntime-gpu               1.12.0
opencv-python                 4.5.5.64
opt-einsum                    3.3.0
packaging                     21.3
pandas                        1.1.5
pandocfilters                 1.5.0
parso                         0.8.3
pickleshare                   0.7.5
Pillow                        9.1.1
pip                           22.0.4
progressbar2                  4.0.0
prometheus-client             0.14.1
prompt-toolkit                3.0.30
protobuf                      3.19.4
psutil                        5.9.1
pyasn1                        0.4.8
pyasn1-modules                0.2.8
pycocotools                   2.0.4
pycparser                     2.21
pydantic                      1.9.1
Pygments                      2.12.0
pyparsing                     3.0.9
pyreadline                    2.1
pyrsistent                    0.18.1
python-dateutil               2.8.2
python-utils                  3.3.3
pytz                          2022.1
PyWavelets                    1.3.0
pywin32                       304
pywinpty                      2.0.6
```

PyYAML	6.0
pyzmq	23.2.0
qtconsole	5.3.1
QtPy	2.1.0
requests	2.27.1
requests-oauthlib	1.3.1
rsa	4.8
scikit-image	0.19.3
scipy	1.7.3
seaborn	0.11.2
Send2Trash	1.8.0
setuptools	62.1.0
six	1.16.0
soupsieve	2.3.2.post1
sparseml	1.0.1
sparsezoo	1.0.0
sympy	1.10.1
tensorboard	2.8.0
tensorboard-data-server	0.6.1
tensorboard-plugin-wit	1.8.1
tensorboardX	2.5.1
tensorflow	2.9.1
tensorflow-estimator	2.9.0
tensorflow-io-gcs-filesystem	0.26.0
termcolor	1.1.0
terminado	0.15.0
thop	0.0.31.post2005241907
tifffile	2021.11.2
tinycss2	1.1.1
toposort	1.7
torch	1.8.2+cu111
torchaudio	0.8.2
torchvision	0.9.2+cu111
tornado	6.2
tqdm	4.64.0
traitlets	5.3.0
typing_extensions	4.2.0
urllib3	1.26.9
vidsz	0.2.0
wcwidth	0.2.5
webencodings	0.5.1
Werkzeug	2.1.2
wheel	0.37.1
widgetsnbextension	3.6.1
wrapt	1.14.1
yolov5	6.1.0
zipp	3.8.0

Appendix B

List of YOLOv3/v5 environment packages Pynq

This appendix lists the packages required to implement YOLOv3 and YOLOv5 Neural Networks within the Pynq framework. All listed packages are compatible with both versions of YOLOv3/v5 Neural Networks.

```
    Package                           Version
--------------------------------- --------------------
2ping                              4.3
alabaster                          0.7.12
anyio                              3.1.0
argon2-cffi                        20.1.0
async-generator                    1.10
atomicwrites                       1.1.5
attrs                              19.3.0
Babel                              2.9.1
backcall                           0.2.0
beautifulsoup4                     4.8.2
bitstring                          3.1.9
bleach                             3.3.0
blinker                            1.4
blosc                              1.7.0
brevitas                           0.7.1
Brotli                             1.0.9
certifi                            2019.11.28
cffi                               1.14.5
chardet                            3.0.4
charset-normalizer                 2.0.12
click                              8.0.1
cloudpickle                        1.3.0
CppHeaderParser                    2.7.4
cryptography                       2.8
cupshelpers                        1.0
cycler                             0.10.0
Cython                             0.29.24
dash                               2.0.0
```

```
dash-bootstrap-components          0.13.1
dash-core-components               2.0.0
dash-html-components               2.0.0
dash-renderer                      1.9.1
dash-table                         5.0.0
dask                               2.8.1+dfsg
dbus-python                        1.2.16
decorator                          4.4.2
defer                              1.0.6
defusedxml                         0.7.1
deltasigma                         0.2.2
dependencies                       2.0.1
distro                             1.4.0
distro-info                        0.23ubuntu1
dnspython                          1.16.0
docker-pycreds                     0.4.0
docutils                           0.17.1
entrypoints                        0.3
et-xmlfile                         1.0.1
finn-base                          0.0.3
finn-dataset-loading               0.0.5
finn-examples                      0.0.4
Flask                              2.0.1
Flask-Compress                     1.10.1
fonttools                          4.33.3
fsspec                             0.6.1
future-annotations                 1.0.0
gitdb                              4.0.9
GitPython                          3.1.27
gpg                                1.13.1-unknown
gTTS                               2.2.3
html5lib                           1.0.1
httplib2                           0.14.0
idna                               2.8
imageio                            2.4.1
imagesize                          1.2.0
imgaug                             0.4.0
importlib-metadata                 1.5.0
imutils                            0.5.4
install                            1.3.5
ipykernel                          5.5.5
ipython                            7.24.0
ipython_genutils                   0.2.0
ipywidgets                         7.6.3
itsdangerous                       2.0.1
jdcal                              1.0
jedi                               0.17.2
Jinja2                             3.0.1
```

joblib	1.1.0
json5	0.9.5
jsonschema	3.2.0
jupyter	1.0.0
jupyter-client	6.1.12
jupyter-console	6.4.0
jupyter-contrib-core	0.3.3
jupyter-contrib-nbextensions	0.5.1
jupyter-core	4.7.1
jupyter-highlight-selected-word	0.2.0
jupyter-latex-envs	1.4.6
jupyter-nbextensions-configurator	0.4.1
jupyter-server	1.8.0
jupyterlab	3.0.16
jupyterlab-pygments	0.1.2
jupyterlab-server	2.5.2
jupyterlab-widgets	1.0.0
jupyterplot	0.0.3
keyring	18.0.1
kiwisolver	1.0.1
language-selector	0.1
launchpadlib	1.10.13
lazr.restfulclient	0.14.2
lazr.uri	1.0.3
locket	0.2.0
lrcurve	1.1.0
lxml	4.5.0
macaroonbakery	1.3.1
Markdown	3.1.1
MarkupSafe	2.0.1
matplotlib	3.5.2
matplotlib-inline	0.1.2
mistune	0.8.4
mnist	0.2.2
more-itertools	4.2.0
mpmath	1.1.0
nbclassic	0.3.1
nbclient	0.5.3
nbconvert	6.0.7
nbformat	5.1.3
nbsphinx	0.8.7
nbwavedrom	0.2.0
nest-asyncio	1.5.1
netifaces	0.11.0
networkx	2.4
notebook	6.4.0
numexpr	2.7.1
numpy	1.22.3

oauthlib	3.1.0
olefile	0.46
opencv-python	4.5.5.64
openpyxl	3.0.3
packaging	20.3
pandas	1.4.2
pandocfilters	1.4.3
parsec	3.9
parso	0.7.1
partd	1.0.0
pathtools	0.1.2
patsy	0.5.1
pbr	5.6.0
pexpect	4.8.0
pickleshare	0.7.5
Pillow	9.1.0
pip	22.0.4
pkg_resources	0.0.0
plotly	5.1.0
pluggy	0.13.0
ply	3.11
prometheus-client	0.10.1
promise	2.3
prompt-toolkit	3.0.18
protobuf	3.20.1
psutil	5.8.0
ptyprocess	0.7.0
py	1.8.1
PyAudio	0.2.11
pybind11	2.8.0
pycairo	1.20.1
pycparser	2.19
pycrypto	2.6.1
pycups	1.9.73
pycurl	7.43.0.2
pyeda	0.28.0
Pygments	2.9.0
PyGObject	3.36.0
pygraphviz	1.5
PyJWT	1.7.1
pymacaroons	0.13.0
PyNaCl	1.3.0
pynq	2.7.0
pynq-dpu	1.4.0
pynq-peripherals	0.1.0
pyparsing	2.4.6
pyRFC3339	1.1
pyrsistent	0.17.3

pytest	4.6.9
pytest-sourceorder	0.5.1
python-apt	2.0.0
python-dateutil	2.8.2
pytz	2022.1
PyWavelets	0.5.1
PyYAML	5.3.1
pyzmq	22.1.0
qtconsole	5.1.0
QtPy	1.9.0
requests	2.27.1
requests-unixsocket	0.2.0
retrying	1.3.3
rise	5.7.1
roman	3.3
scikit-image	0.16.2
scikit-learn	1.0.2
scipy	1.8.0
seaborn	0.11.2
SecretStorage	2.3.1
Send2Trash	1.5.0
sentry-sdk	1.9.4
setproctitle	1.2.2
setuptools	44.0.0
Shapely	1.8.1.post1
shortuuid	1.0.9
simple-term-menu	1.4.1
simplegeneric	0.8.1
simplejson	3.16.0
six	1.14.0
smmap	5.0.0
sniffio	1.2.0
snowballstemmer	2.1.0
soupsieve	1.9.5
SpeechRecognition	3.8.1
Sphinx	4.2.0
sphinx-rtd-theme	1.0.0
sphinxcontrib-applehelp	1.0.2
sphinxcontrib-devhelp	1.0.2
sphinxcontrib-htmlhelp	2.0.0
sphinxcontrib-jsmath	1.0.1
sphinxcontrib-qthelp	1.0.3
sphinxcontrib-serializinghtml	1.1.5
SQLAlchemy	1.3.12
ssh-import-id	5.10
sympy	1.5.1
systemd-python	234
tables	3.6.1

```
tenacity                          8.0.0
terminado                         0.10.0
terminaltables                    3.1.10
testpath                          0.5.0
testresources                     2.0.1
thop                              0.0.31.post2005241907
threadpoolctl                     3.1.0
tokenize-rt                       4.2.1
toolz                             0.9.0
torch                             1.11.0
torchvision                       0.12.0
tornado                           6.1
tqdm                              4.62.3
traitlets                         5.0.5
transitions                       0.7.2
typing_extensions                 4.1.1
ubuntu-advantage-tools            20.3
unattended-upgrades               0.1
urllib3                           1.26.11
uvloop                            0.14.0
voila                             0.2.10
voila-gridstack                   0.2.0
wadllib                           1.3.3
wandb                             0.13.1
wcwidth                           0.1.8
webencodings                      0.5.1
websocket-client                  1.0.1
Werkzeug                          2.0.1
wheel                             0.37.1
widgetsnbextension                3.5.1
wurlitzer                         3.0.2
xlrd                              1.1.0
xlwt                              1.3.0
zipp                              1.0.0
```

Appendix C

Power measurements on CPU/GPU

This appendix provides comprehensive graphs of the energy consumption measurements of the Neural Networks listed in Table 4.1 on CPU and GPU implementations.

CPU measurements:

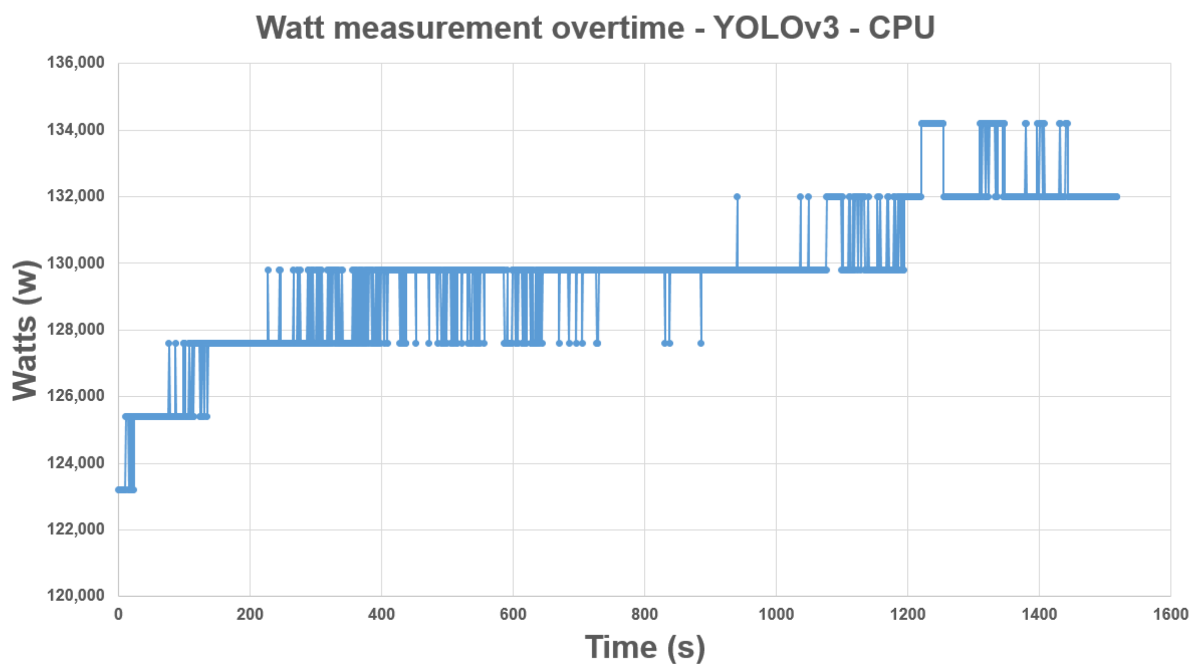

Figure C.1: Watt measurement overtime - YOLOv3 - CPU

Figure C.2: Watt measurement overtime - YOLOv3-tiny - CPU

Figure C.3: Watt measurement overtime - YOLOv5l - CPU

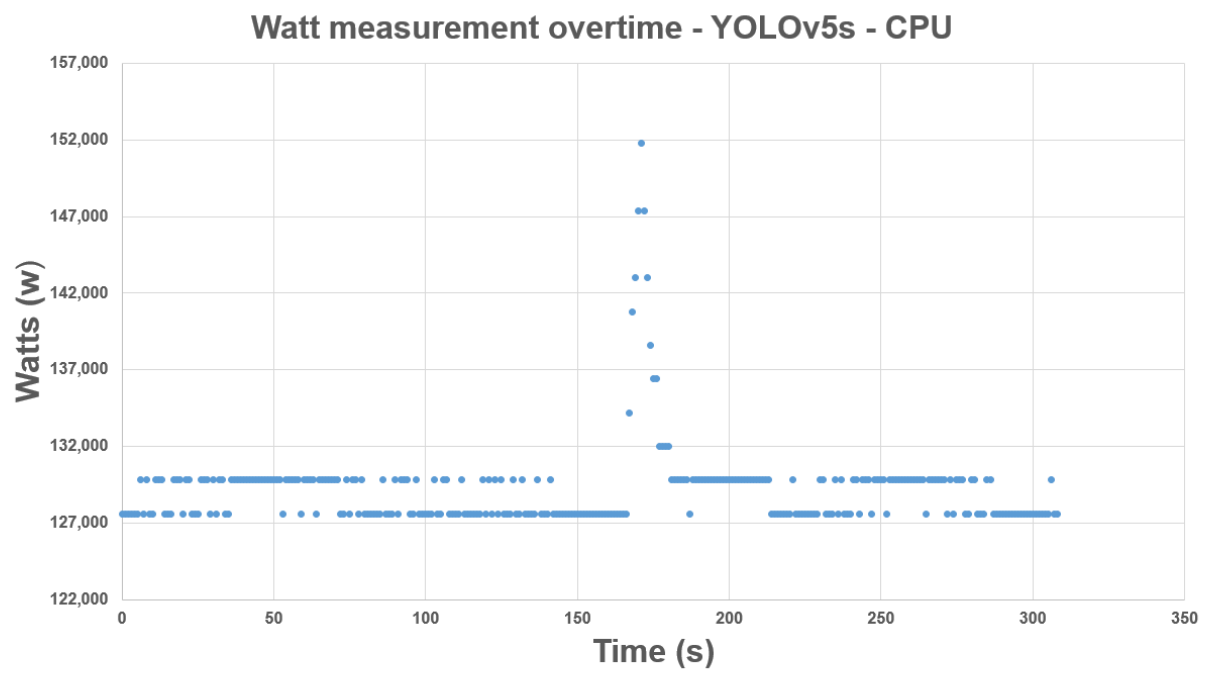

Figure C.4: Watt measurement overtime - YOLOv5s - CPU

GPU measurements:

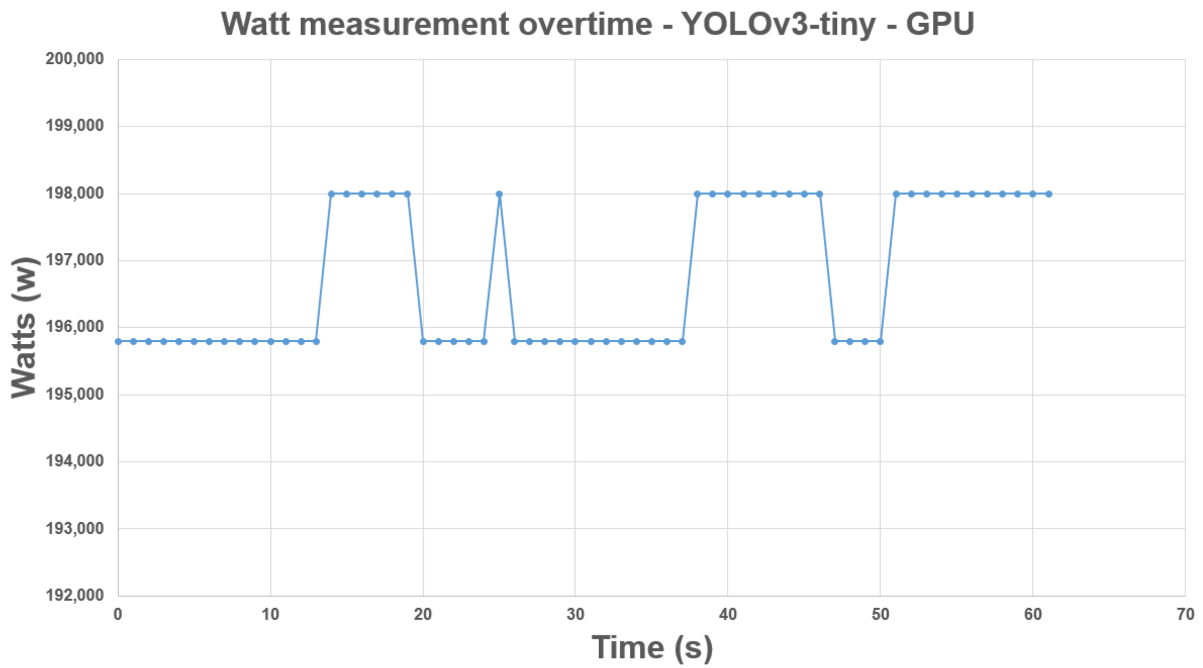

Figure C.5: Watt measurement overtime - YOLOv3 - GPU

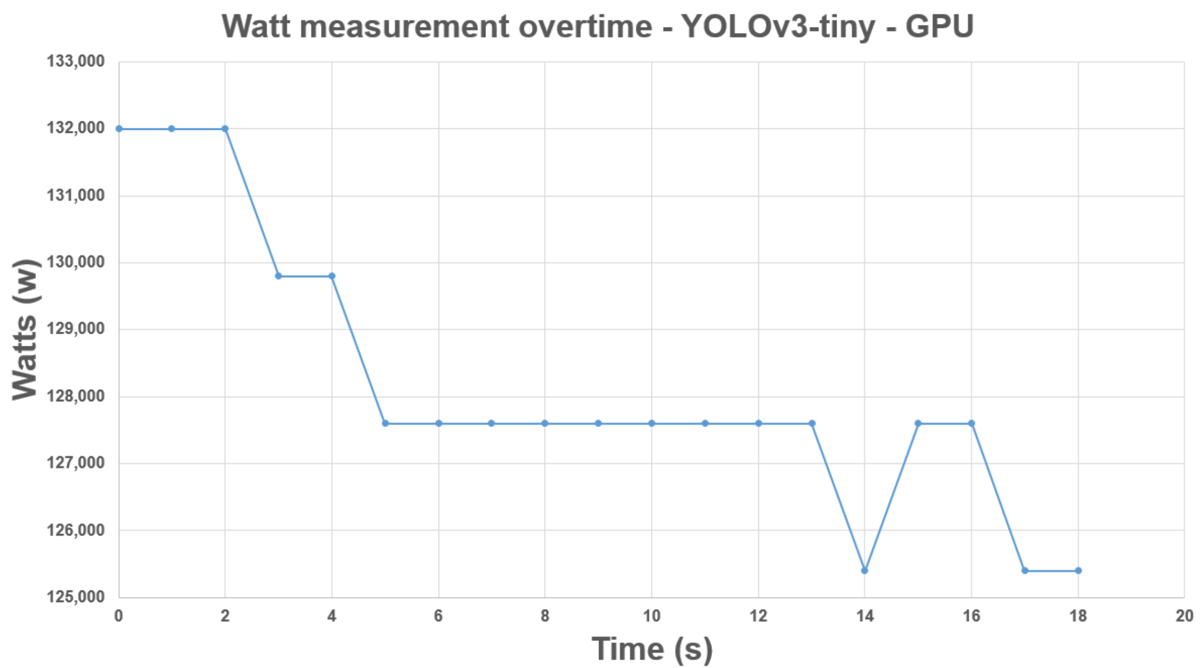

Figure C.6: Watt measurement overtime - YOLOv3-tiny - GPU

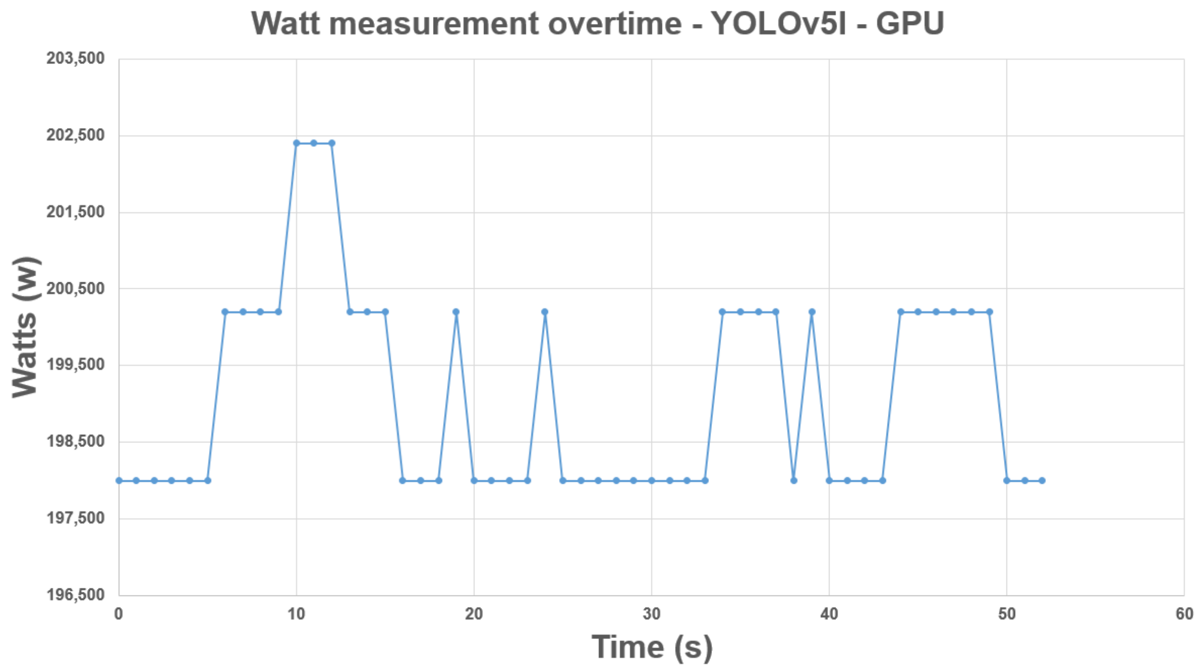

Figure C.7: Watt measurement overtime - YOLOv5l - GPU

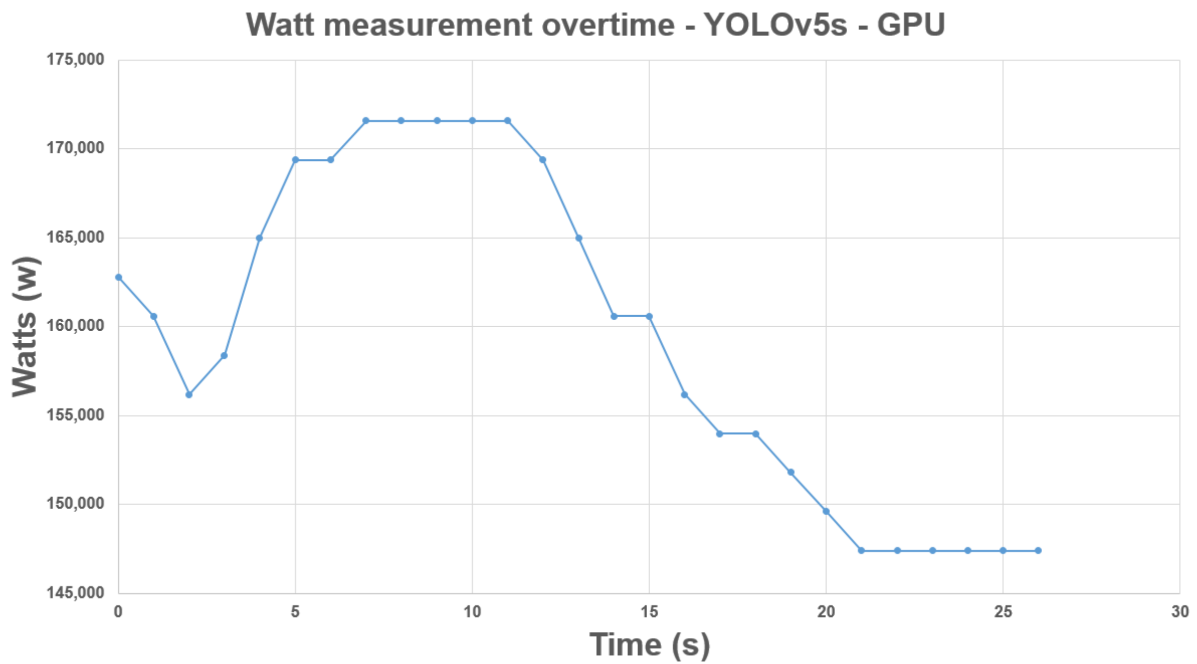

Figure C.8: Watt measurement overtime - YOLOv5s - GPU

Appendix D

Power measurements on Pynq framework

This appendix provides graphs of the energy consumption measurements of the Neural Networks listed in Table 4.2 for the Pynq framework. The fact that the inferences took a long time resulted in graphs a little difficult to analyze, however, the average power was marked with a red line along the inference.

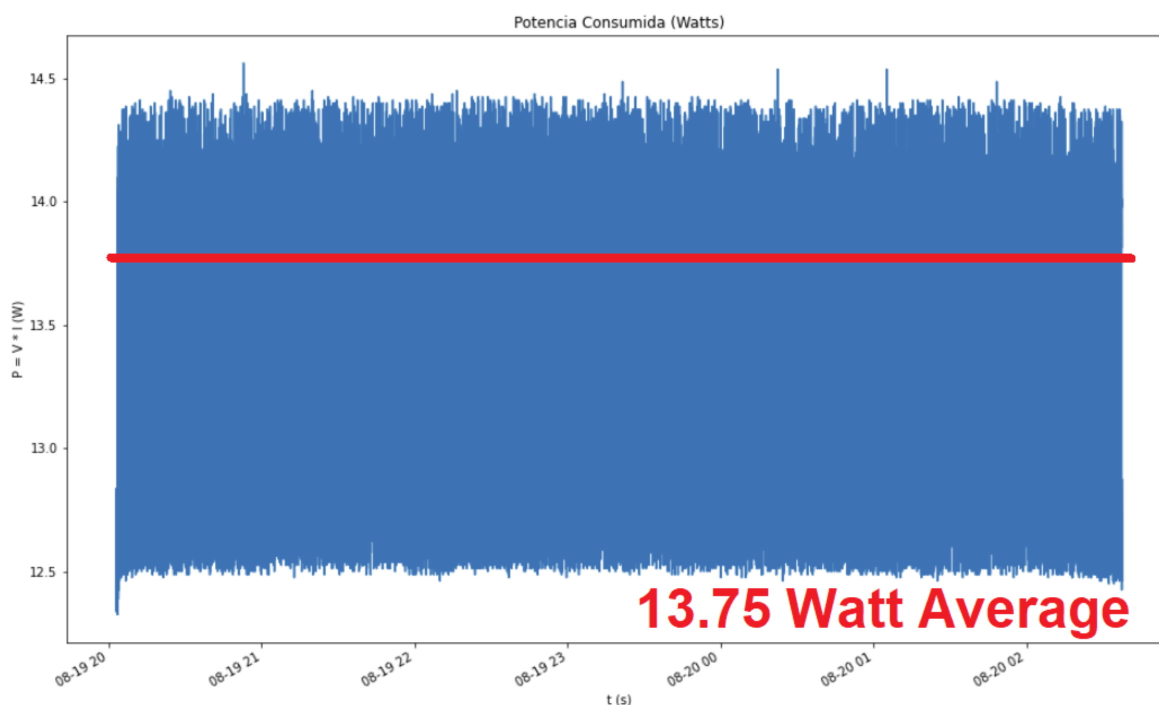

Figure D.1: Watt measurement overtime - YOLOv3 - Pynq

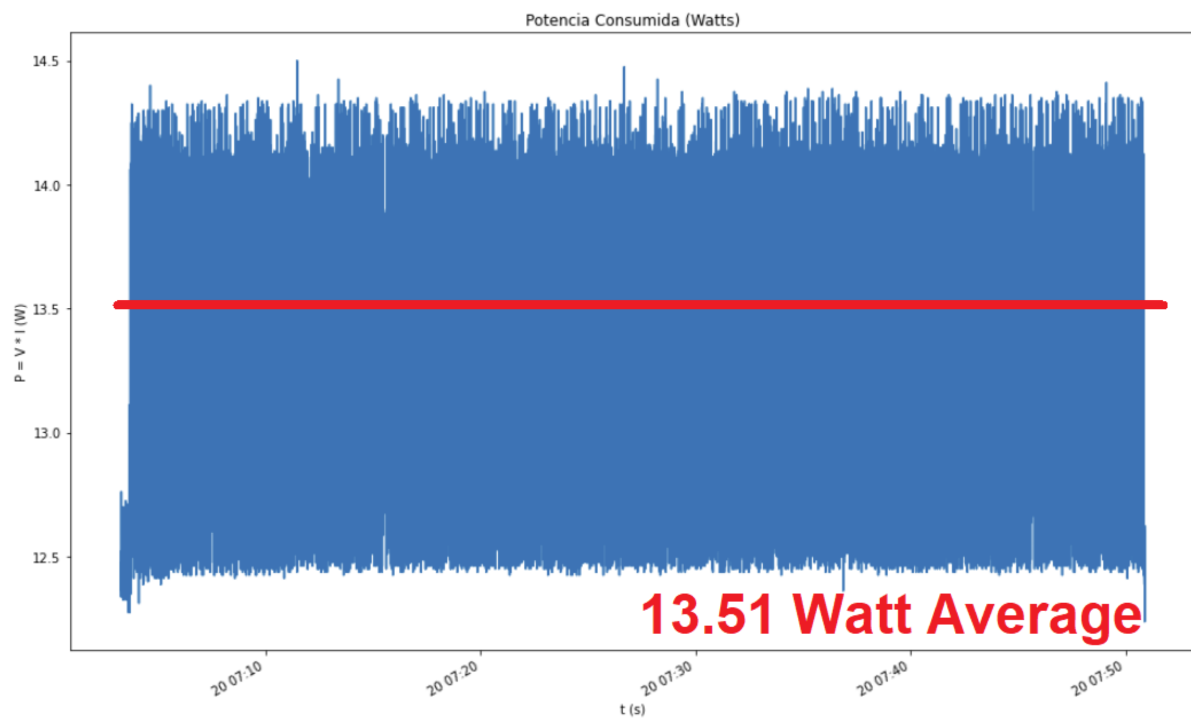

Figure D.2: Watt measurement overtime - YOLOv3-tiny - Pynq

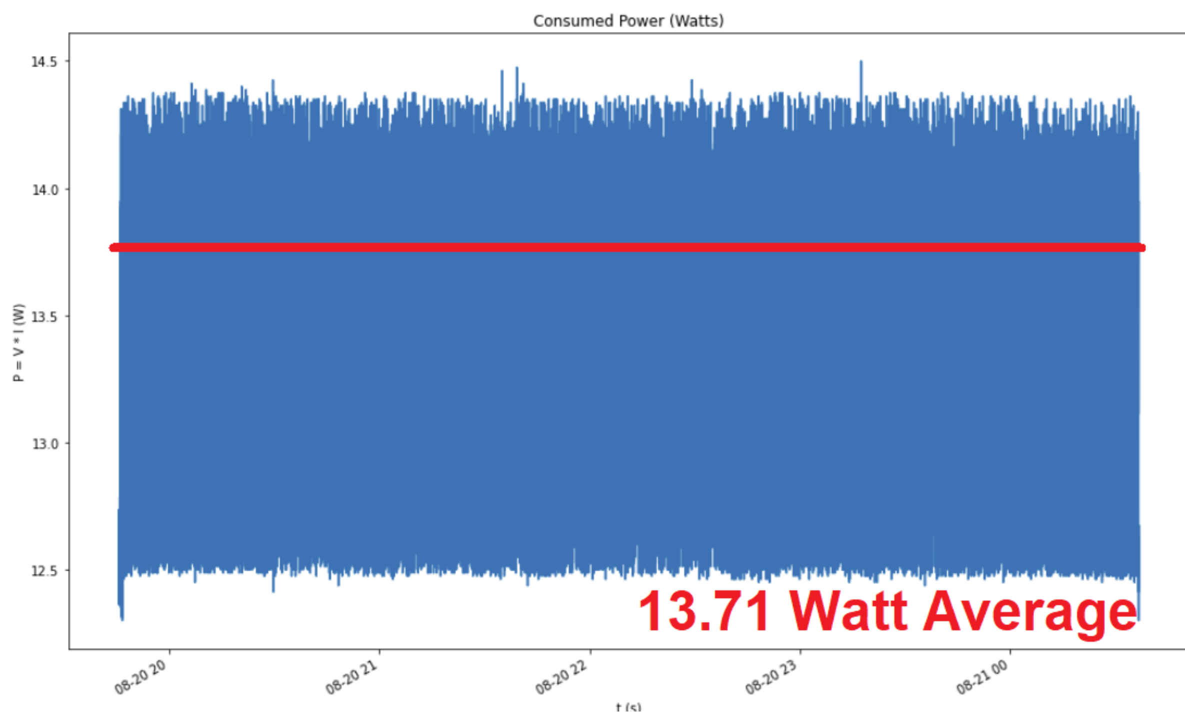

Figure D.3: Watt measurement overtime - YOLOv5l - Pynq

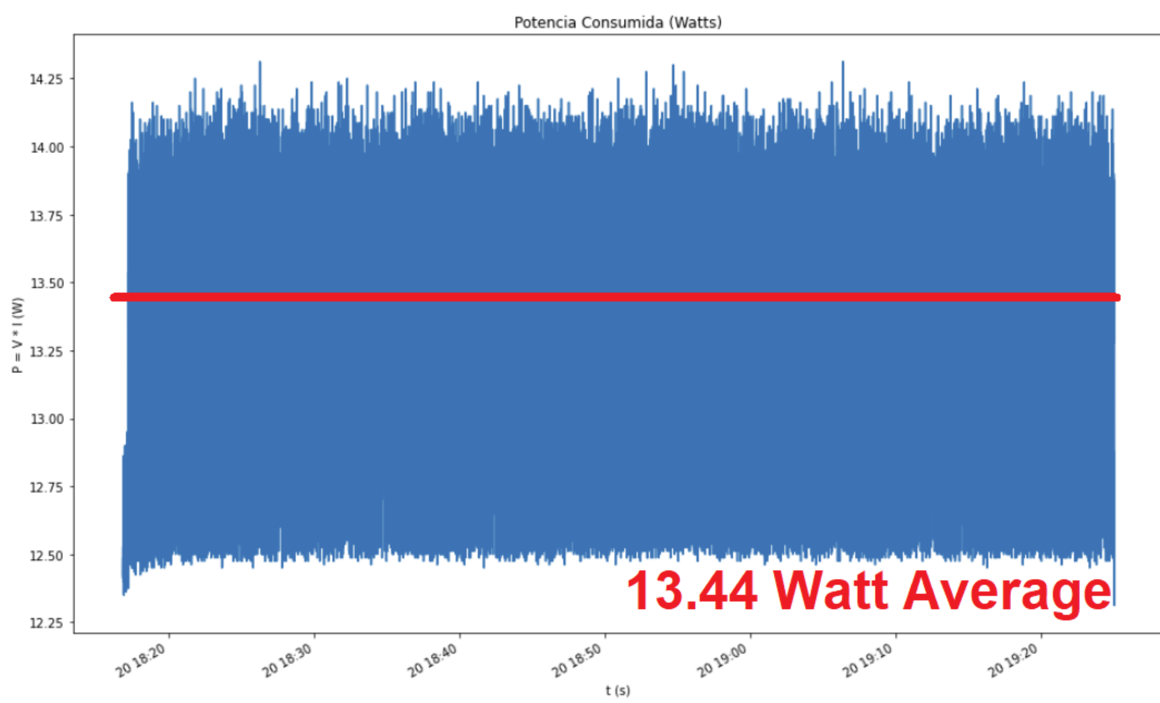

Figure D.4: Watt measurement overtime - YOLOv5s - Pynq

Appendix E

Power measurements on Vitis-AI framework

This appendix presents the graphs of the energy consumption measurements of the Neural Networks mentioned in Table 4.3 in the Vitis-AI framework.

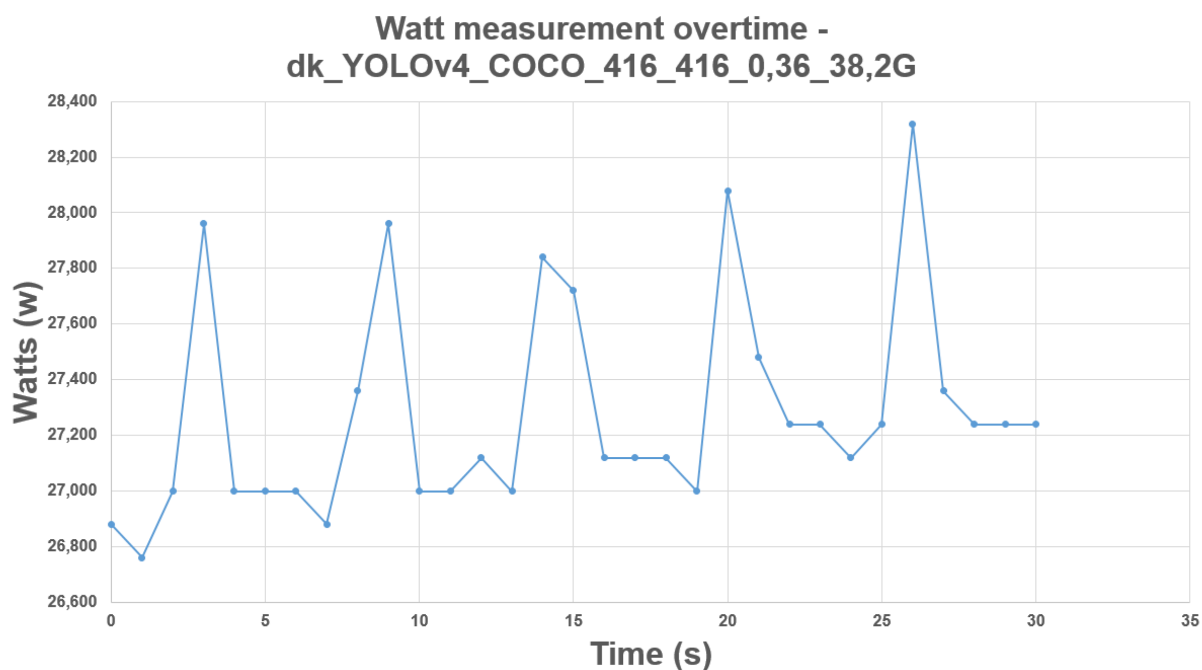

Figure E.1: Watt measurement overtime - Dk_YOLOv4_COCO_416_416_0.36_38.2G (27.26 Watts on average).

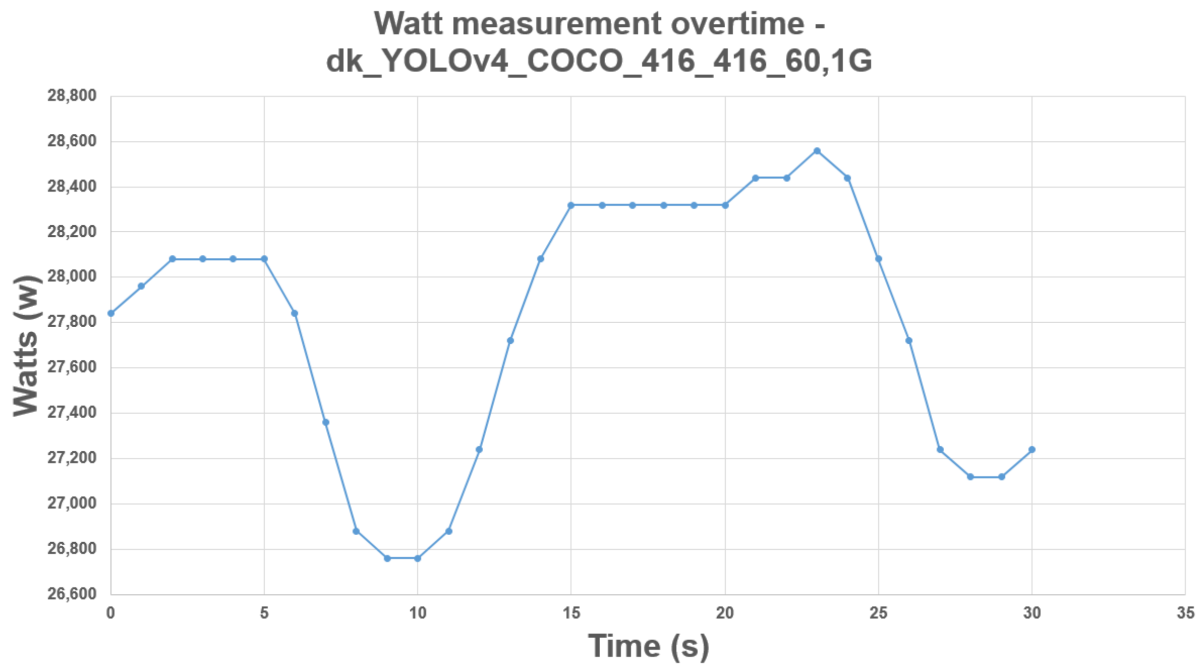

Figure E.2: Watt measurement overtime - Dk_YOLOv4_COCO_416_416_60.1G (27.80 Watts on average).

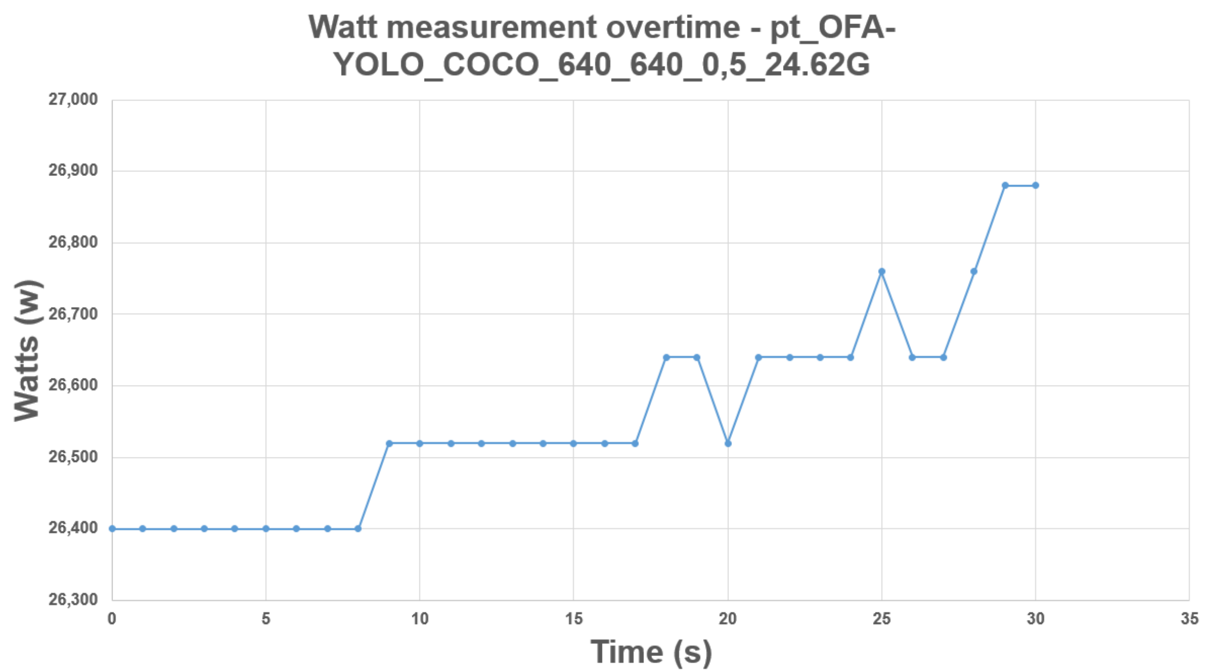

Figure E.3: Watt measurement overtime - Pt_OFA-YOLO_COCO_640_640_0.5_24.62G. (26.55 Watts on average)

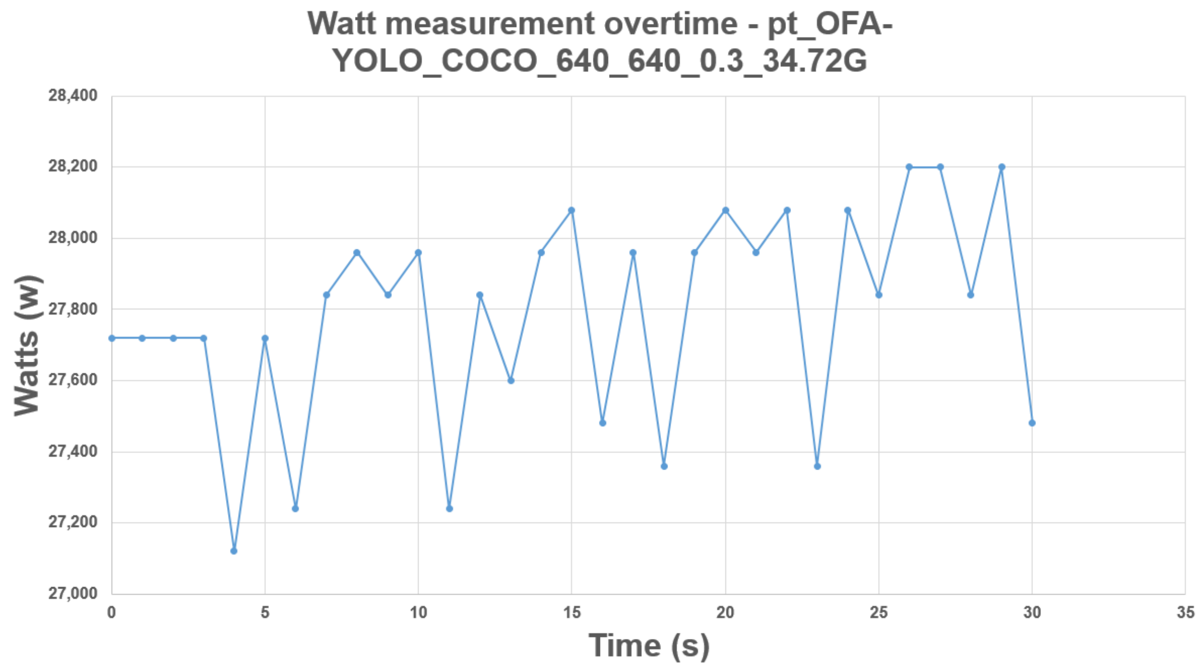

Figure E.4: Watt measurement overtime - Pt_OFA-YOLO_COCO_640_640_0.3_34.72G. (27.78 Watts on average)

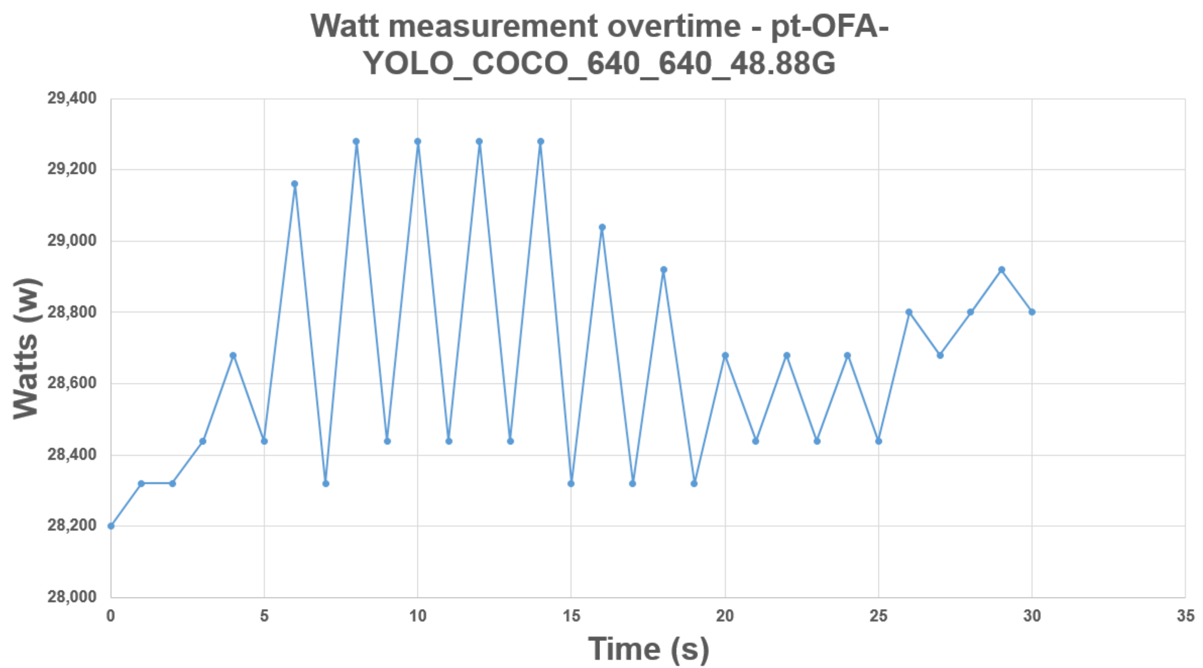

Figure E.5: Watt measurement overtime - Pt_OFA-YOLO_COCO_640_640_48.88G (28.66 Watts on average).

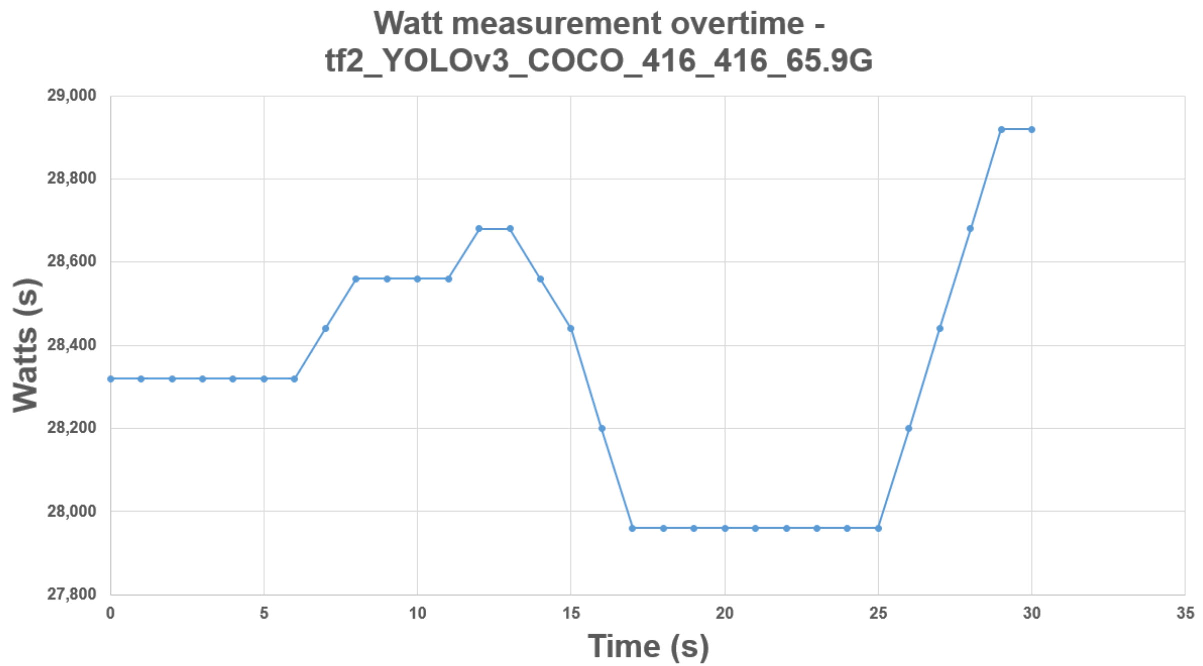

Figure E.6: Watt measurement overtime - Tf2_YOLOv3_COCO_416_416_65.9G (28.33 Watts on average).

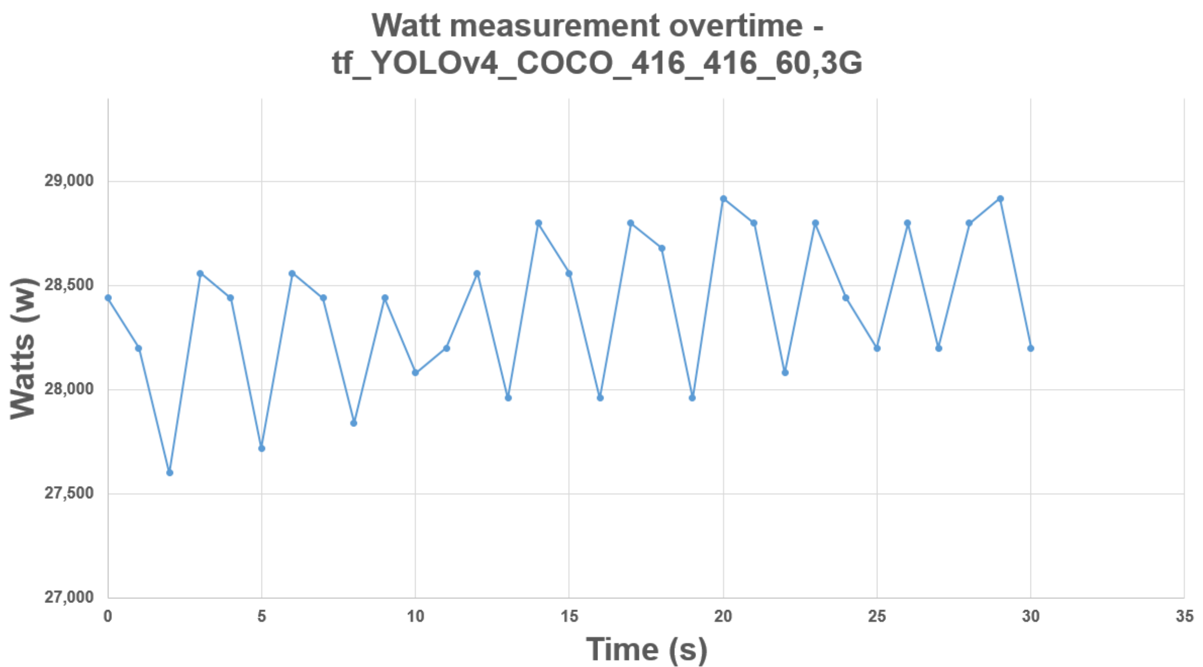

Figure E.7: Watt measurement overtime - Tf_YOLOv4_COCO_416_416_60.3G (28.38 Watts on average).

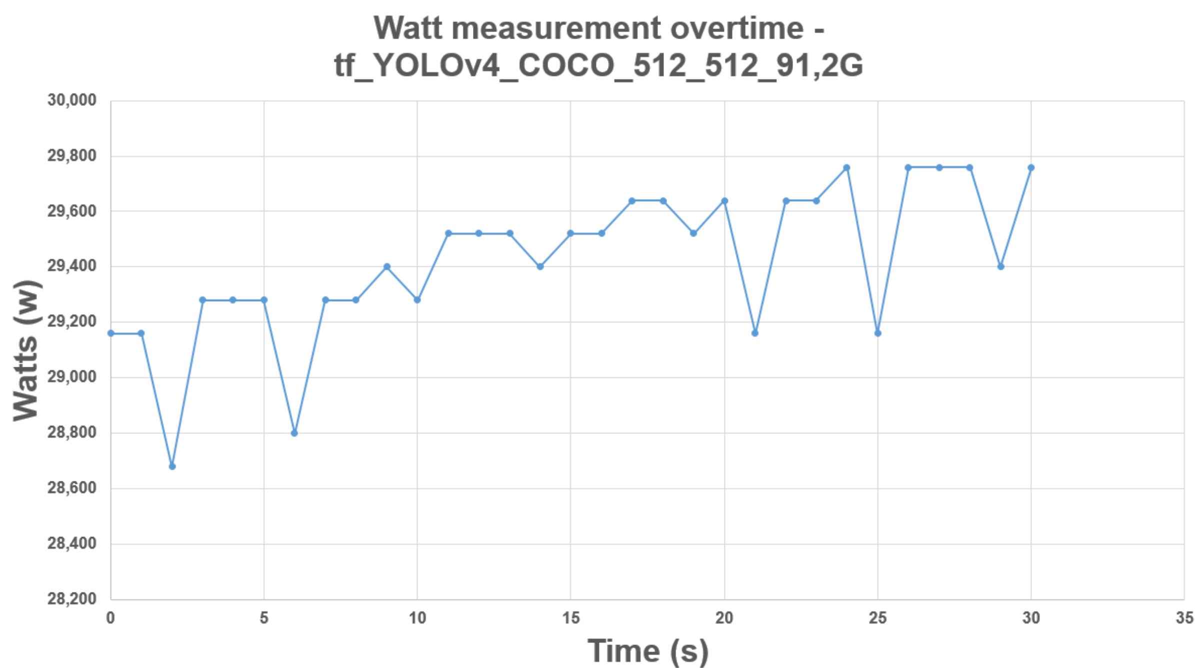

Figure E.8: Watt measurement overtime - Tf_YOLOv4_COCO_512_512_91.2G (29.42 Watts on average).

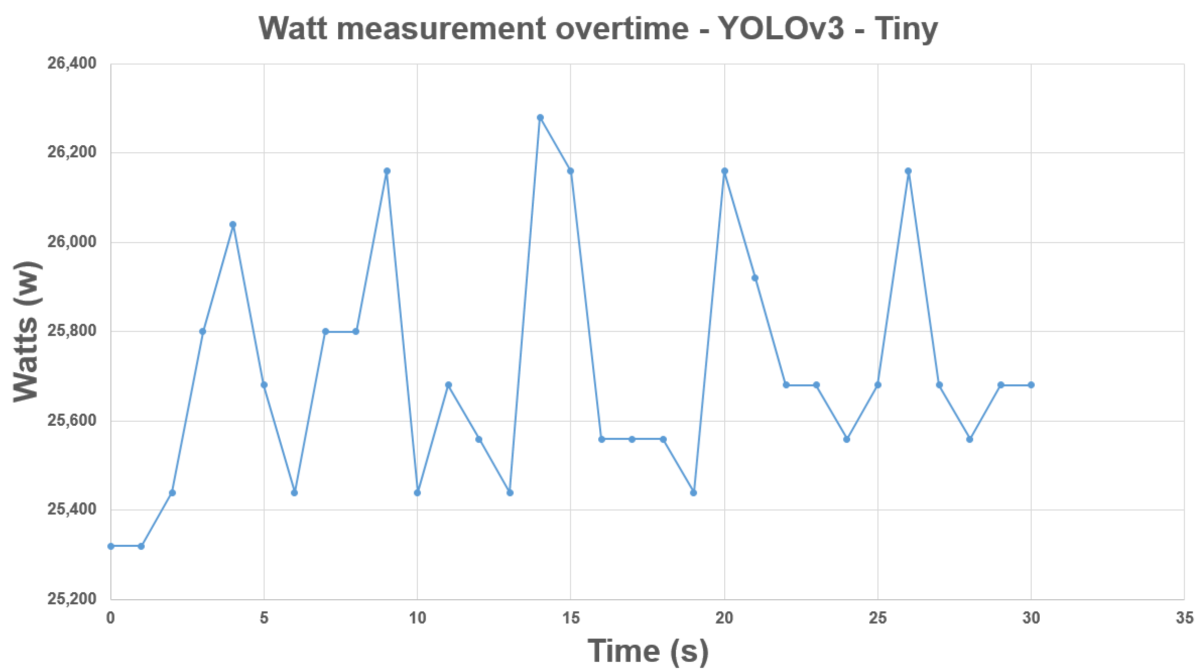

Figure E.9: Watt measurement overtime - YOLOv3-Tiny_coco_416_416 (25.70 Watts on average).

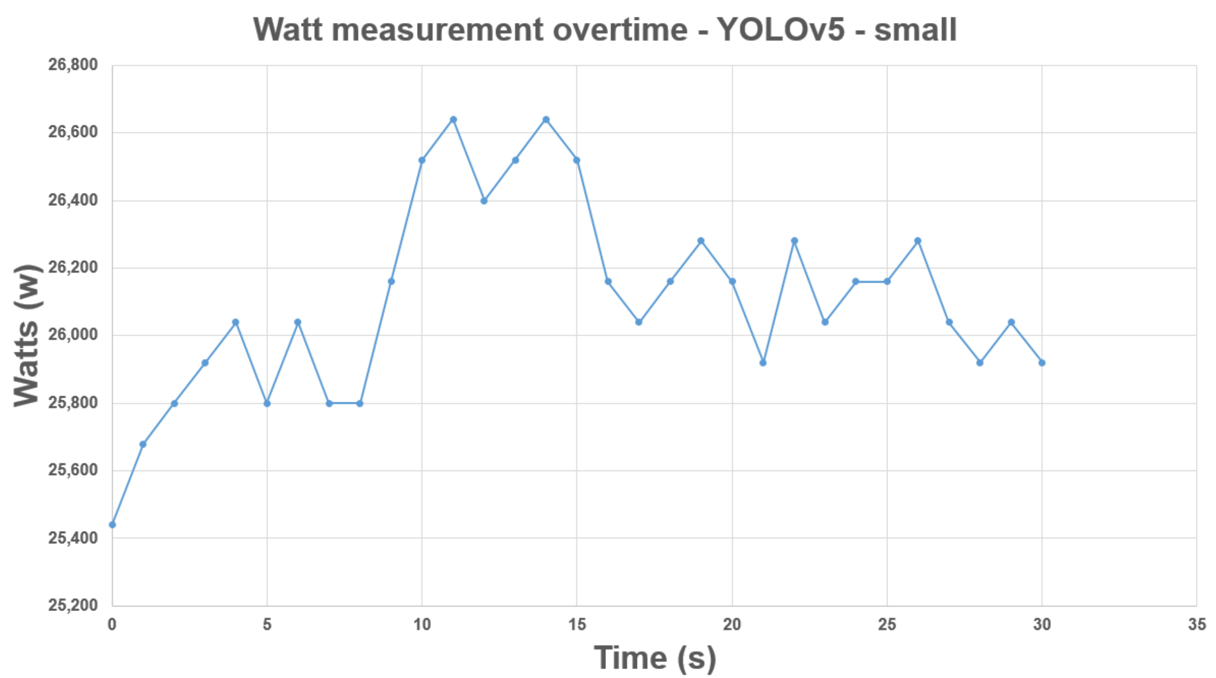

Figure E.10: Watt measurement overtime - YOLOv5-small_coco_640_640 (26.10 Watts on average).