Software-Hardware Co-Design of Multi Task Deep Neural Networks for Camera Vision System

A Thesis submitted in partial fulfillment of the requirement for the Award of the Degree of
MASTER OF TECHNOLOGY
in VLSI Design

BODH KRISHNA KARN
6024620034
Submitted By

Under Supervision of
Dr. Anil Singh
ASSOCIATE PROFESSOR
Dr. Manu Bansal
ASSOCIATE PROFESSOR

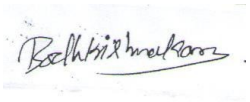
ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT
THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB
JULY, 2026

DECLARATION

I **Bodh Krishan Karn** hereby declare that the work presented in this report entitled **"Software-Hardware Co-Design of Multi Task Deep Neural Networks for Camera Vision System"** in partial fulfillment of the requirement for the award of degree of **Master of Technology (VLSI Design)** submitted at **ECED Department**, Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under supervision of **Dr. Anil Singh** and **Dr. Manu Bansal** (both Associate Professors), ECED Department, TIET, Patiala, from July 2025 to July 2026.

The matter presented in this has not been submitted either in part or full to any other university or institute for the award of any other degree.

Date: 30/06/2026



Bodh Krishna Karn
(6024620034)

(Name of TIET Supervisor-I)

Dr. Anil Singh
Associate Professor
Department of Electronics and Communication Engineering
Thapar Institute of Engineering & Technology, Patiala
Date: 30/06/2026

(Name of TIET Supervisor-II)

Dr. Manu Bansal
Associate Professor
Department of Electronics and Communication Engineering
Thapar Institute of Engineering & Technology, Patiala
Date: 30/06/2026

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to all those who have contributed to the successful completion of this report work.

First and foremost, I extend my heartfelt appreciation to my supervisors, **Dr. Anil Singh** and **Dr. Manu Bansal**, for their invaluable guidance, continuous support, and expert advice throughout the research period. Their profound knowledge in the field of image and object detection has been instrumental in shaping this work.

I am deeply grateful to the faculty members of the **Department of Electronics and Communication Engineering**, **Thapar Institute of Engineering & Technology**, for providing me with the necessary theoretical foundation and practical knowledge required for this research. Special thanks are extended to the Head of the Department for providing access to the required computational resources and FPGA development platforms. Their insights and suggestions have significantly contributed to improving the quality of this research.

Finally, I express my profound gratitude to my family and friends for their constant encouragement, understanding, and moral support throughout this academic journey. Their unwavering belief in me has been a source of motivation and strength during the completion of this work.

ABSTRACT

Real-time camera vision on edge devices must run multiple perception tasks simultaneously while staying within tight power and memory constraints. However, single-task detection networks like YOLO series are unable to do so efficiently. In this thesis, we present a software-hardware co-design multi-task deep neural network architecture which performs seven visual perception tasks: detection, classification, semantic & instance segmentation, pose, oriented detection, and age-gender estimation, with a single feature backbone and lightweight task-specific heads. An Intelligent Preprocessing Module measures scene complexity and runs only the tasks a given frame actually needs, reducing redundant computation and power consumption. The architecture was first developed in Python, trained on Kaggle, and demonstrated on an NVIDIA H100 GPU. Its computational core was then translated into synthesizable Verilog and integrated with the Zynq UltraScale+ MPSoC. This provides hardware–software co-design, with the Processing System handling control while the custom accelerator runs in the programmable logic. On the H100, the model sustains 177–196 FPS at 5.0–5.6 ms per frame with a 4.7 MB backbone. On hardware, it uses 36,472 LUTs, 71,429 registers, and 59 DSP slices at 100 MHz. It meets all timing constraints (WNS +0.358 ns) and consumes 3.532 W total on-chip power, independently verified as 3.590 W in AMD Power Design Manager.

Keywords: Multi-task learning, Hardware–software co-design, FPGA, Zynq UltraScale+ MPSoC, Edge AI, Real-time vision

CONTENTS

Declaration ii

Acknowledgement iii

Abstract iv

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Problem Statement 1

1.2 Proposed Solution 2

1.3 Research Scope 3

1.4 Purpose and Objectives 3

2 Literature Review 4

2.1 Introduction 4

2.2 Literature Review 4

2.3 Research Gaps and Future Directions 8

2.3.1 Research Gaps 8

2.3.2 Objectives 9

3 Object Detection 10

3.1 Introduction to Object Detection 10

3.2 Types of Object Detection 10

3.2.1 Two-Stage Detectors 12

3.2.2 Single-Stage Detectors 12

3.2.3 Transformer-Based Detectors 13

3.3 Performance Metrics 13

3.3.1 Intersection over Union (IoU) 13

3.3.2 Mean Average Precision (mAP) 14

3.3.3 Precision and Recall 14

3.3.4 Inference Speed and Computational Efficiency 14

3.4 YOLO Algorithm 14

3.5 YOLO Architecture 14

3.5.1 Grid-Based Detection Framework 15

3.5.2 Convolutional Backbone Network 15

3.5.3 Detection Head and Output Formatting . . . 16
3.6 YOLO Series Evolution . . . 16
3.6.1 YOLOv1: Foundation and Innovation . . . 16
3.6.2 YOLOv2: Addressing Early Limitations . . . 17
3.6.3 YOLOv3: Multi-Scale Detection . . . 17
3.6.4 YOLOv4: Optimization and Efficiency . . . 19
3.6.5 YOLOv5 Through YOLOv11: Modern Developments . . . 19
3.7 Comparison and Analysis . . . 19
3.7.1 Performance Metrics Comparison . . . 20
3.7.2 Architectural Evolution Analysis . . . 20

4 Neural Network Optimization in Camera Vision System 23
4.1 Motivation and Design Methodology . . . 23
4.2 Architecture Overview . . . 24
4.2.1 System-Level Architecture . . . 24
4.3 Intelligent Preprocessing Module . . . 25
4.4 LEAKY ReLU Activation Function (L-ReLU):- . . . 31
4.5 MULTI SCALE FEATURE PYRAMID NETWORK (MS-FPN):- . . . 32
4.5.1 Purpose: - . . . 32
4.5.2 FPN Architecture: . . . 32
4.6 Hardware implementation of the computational model: - . . . 34

5 Results and Discussion 35
5.1 Results . . . 35
5.1.1 Test Environment and Model Overview . . . 35
5.1.2 Conclusion and Implications for Realtime Deployment: - . . . 38
5.2 HARDWARE ANALYSIS: . . . 38
5.2.1 Block Design Overall Architecture: . . . 40
5.2.2 Synthesis Result Description:- . . . 43
5.2.3 Implementation Result Description: - . . . 45
5.3 Discussion . . . 53

6 Conclusion 55

LIST OF FIGURES

1.1 Bounding box annotation and IoU-based localization accuracy in object detection. 2

3.1 Diagram of Object Detection [54]. 11
3.2 Flow chart of Object Detection [54]. 11
3.3 Formula of IoU [16]. 13
3.4 YOLO architecture diagram showing single-shot object detection process [54]. 15
3.5 YOLOv1 Architecture Diagram [55]. 17
3.6 YOLOv2 Detection Diagram [55]. 18
3.7 YOLOv3 Architecture Diagram [56]. 18
3.8 YOLOv4 Architecture Diagram [59]. 19
3.9 YOLOv5 Architecture Diagram [60]. 20
3.10 YOLOv6 Architecture Diagram [57]. 20
3.11 YOLO Model Evolution. 22
3.12 YOLO Performance Comparison. 22

4.1 Block Diagram of the Proposed Architecture. 23
4.2 Proposed Architecture Overview. 25
4.3 Detailed architecture of proposed design. 25
4.4 Intelligent preprocessing module (IPM) flow chart. 26

5.1 Output log of the first execution of the proposed multi-task vision model. 35
5.2 Output log 2. 36
5.3 Output log3. 36
5.4 Output log 4. 37
5.5 Functional simulation of the proposed design showing successful AXI read. . . 39
5.6 Vivado block design of the Zynq UltraScale+ MPSoC-based OmniVision hardware architecture. 39
5.7 Post-synthesis resource utilization report. 44
5.8 Post-implementation resource utilization report. 45
5.9 Post-implementation dataflow schematic Design. 47
5.10 Place-and-route layout of the proposed FPGA design. 48
5.11 Post-implementation timing summary of the proposed design. 49
5.12 Post-implementation on-chip power summary for the implemented design. . . . 50
5.13 AMD Power Design Manager (PDM) on-chip power summary. 52
5.14 PDM Low Power Domain (LPD) on-chip power breakdown. 52
5.15 PDM Full Power Domain (FPD) on-chip power breakdown. 53

LIST OF TABLES

3.1 Metrics Comparison of YOLO Series 21

4.1 Leaky ReLU Activation Function Comparison with Standard ReLU [74]. 31
4.2 Multi-Scale Feature Pyramid Network (FPN) Receptive Field Coverage [75]. . . 32

5.1 Input ports of RTL Design. 41
5.2 Output ports of RTL Design. 41
5.3 Detailing of all Slice Blocks. 41
5.4 On-chip power as a function of the default toggle rate (12.5%–75%), all other conditions held constant. 51

CHAPTER 1

INTRODUCTION

The tremendous progress in computer vision and deep learning has brought unparalleled opportunities to build advanced object detection systems with real-time performance. Object detection, a core computer vision task, has evolved from old feature-based techniques to contemporary deep learning methods. The discipline has seen tremendous advances with the advent of convolutional neural networks (CNNs) and, later, the development of customized architectures specifically for object detection. The development of the YOLO (You Only Look Once) algorithm is a milestone in object detection since this algorithm brought about a new era where one could use an approach that can work on the entire image at once [1, 2, 3].

The ability to detect objects is crucial in many different fields, including surveillance systems, driverless cars, robots, medical image analysis, and factory automation. Modern applications require object detection systems that not only demonstrate good accuracy but are also real-time and computationally efficient. Sliding window approaches and two-stage object detectors like the R-CNN series, although demonstrating high accuracy, have high computational demands and latency, which limits their usability in many applications [4, 5]. The invention of the YOLO object detection system solved this problem by reformulating the object detection task as a regression problem, enabling a single pass prediction of bounding boxes and class probabilities [1].

This thesis explores the area of YOLO object detection systems in detail. It covers the history of the YOLO object detection system, architecture innovations, and performance features. The research focuses on the history of YOLO development from the very first YOLOv1 to the latest YOLOv11 version, showing how each version improved the previous one and introduced new features [6, 7, 8]. The research consists of an in-depth analysis of object detection methods, performance criteria, and comparative analysis of YOLO versions to understand their strengths and weaknesses and suitability for different applications.

1.1 Problem Statement

The real-world camera does not normally face the task of solving just one problem. One and the same frame of video stream coming from a surveillance camera, traffic camera, or driver assistance system is expected to undergo object detection, classification, pixel-wise segmentation, and analysis of such features as an object's pose or the estimated age and gender of a person. Given the modern detector solutions, the natural solution is to use a set of specialized networks to detect the objects, classify them, segment the pixels, and so forth. These individual networks perform extremely well; one-stage object detection has become a sufficiently mature field that reviews start considering real-time precision as a baseline result [10]; it is possible to have an integrated model of bounding boxes and pixel-wise segmentation like Mask R-CNN [11, 12], and single-stage lightweight networks like SSD and RetinaNet keep the

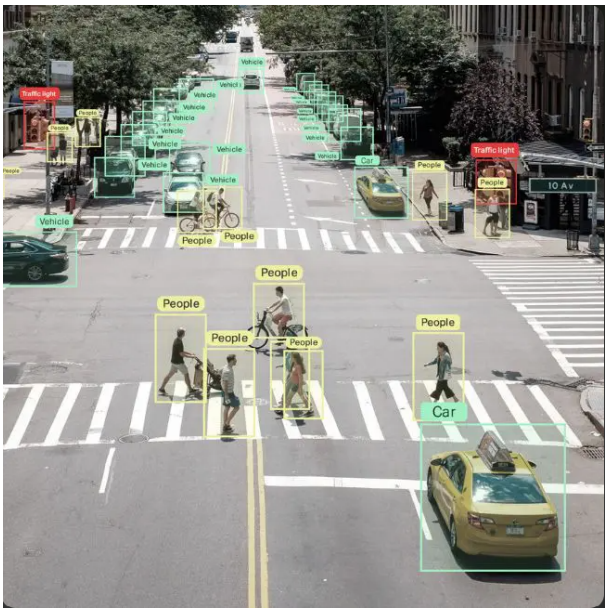


Figure 1.1: Bounding box annotation and IoU-based localization accuracy in object detection.

latency down for each particular task [13, 14]. Each one of those works in a typical detection pipeline [15] optimized for its own localization metric [16].

It becomes clear what happens when they are forced to operate together. They don't sum up their costs but rather multiply them. The rough estimate of three models' cost is about 100 ms for detection, 40 ms for classification, and 120 ms for segmentation that makes 260 ms per image which is an order of magnitude more than 30 ms a 30 FPS pipeline can allocate. Memory operates similarly – it is possible to overload flash memory on an average embedded device by storing several models and total power of such models will drain a battery within few minutes. The principle of utilizing one learned representation for multiple tasks is itself widely discussed in multi-task learning literature [17] and models such as MultiNet proved that one encoder can process multiple heads in real time [18].But the question left unanswered until now is whether this can be done with respect to edge power budget constraints and implementation in hardware. A typical detector output, such as the one in Figure 1.1, confirms this in practice: vehicles, cars, people and traffic lights are all located and labelled by a single detection task, with no classification, segmentation, pose, or age/gender information attached, and the whole process is evaluated purely as an algorithm, one network at a time, with no regard for the power or memory a real edge chip would have to spend running it. Thus, real challenge for edge multi-task vision is not an accuracy of each detector but multiplicative increase in computational cost, memory requirements, and energy consumption.

1.2 Proposed Solution

As opposed to the approach of stacking networks that do not have any connection, the present thesis does the exact opposite by creating one network that has one common backbone for its features, and on top of it sits a small task-specific head. The quantised backbone of roughly 4.7 MB covers all seven vision tasks from detection and classification through semantic and

instance segmentation, pose, oriented detection to age/gender estimation.

Over the common backbone network is an Intelligent Preprocessing Module, which examines each received frame and makes a decision about how complex its visual content is, turning on the necessary heads only – from all seven tasks simultaneously to only detection if the scene is almost empty. And it is the adaptive gate which enables the proposed unified architecture to operate within the constraints of the edge device's power capacity by reducing energy consumption by 28–72%, compared to the static pipeline doing all the tasks. And finally, this computational model is moved to hardware implementation, enabling the same multi-task network working in real-time on GPU to operate on the edge device.

1.3 Research Scope

The work scope encompasses the full development process, from a software-based model to a functional implementation that operates in hardware, instead of limiting the analysis to either of them. On the software side, the scope includes creation and profiling of the multi-task neural network model in Python and its training on a high-performance GPU for evaluation of the real-time behavior. In order to determine the parts of the model that deserve further optimization by hardware implementation, the research relies on prior knowledge gained in the field of efficient deep neural networks execution and hardware-software co-design [19]. The computational core of the model will be redesigned as synthesizable Verilog code and implemented in a Zynq UltraScale+ MPSoC in the same way as the embedded FPGA-based accelerator presented in [20]: hard PS executing control functions and the programmable logic implementing arithmetic operations. The scope purposely includes all the aspects that tend to be overlooked in software-only studies, namely the evaluation of the resources consumed after implementation, timing closure, and on-chip power analysis of the post-synthesis and placed-and-routed design.

1.4 Purpose and Objectives

The motivation behind this thesis comes from a larger problem placed upon camera-based vision systems. Cameras have become a core sensing component for many everyday systems including surveillance, traffic control, robotics, mobile devices, and driver assistance, and such systems are increasingly called upon not only to carry out a single specific task but to interpret the scene in multiple ways at once. At the same time, most of this processing must be done locally and on the edge, on devices with very constrained computational capacity, memory, and energy [21, 22]. This has driven an increasing focus on putting this intelligence onto the edge device, mapping vision networks onto reconfigurable hardware such as FPGAs [23], shifting artificial intelligence toward the network edge [24], and compressing or specialising models so that competent networks fit onto less powerful hardware [25, 26], each built on the advances in deep networks that made vision what it is today [27]. It is the conjunction of these two trends which has brought about this work and motivated its overall objective: to examine the problem of making camera vision both capable and efficient enough for practical use.

CHAPTER 2
LITERATURE REVIEW

2.1 Introduction

This chapter presents a comprehensive literature review of the current state-of-the-art research in AI accelerators, YOLO-based object detection, Neural Processing Units (NPUs), and FPGA implementations for real-time applications. The review covers recent advances in hardware acceleration techniques and their applications in object detection systems.

2.2 Literature Review

Zhang et al. [28] tackle the key challenge of robust and efficient obstacle detection for autonomous Mining Electric Locomotives (MELs) in dark, dusty underground mines. To overcome the low-performance issue with typical object detectors towards small obstacles, they introduce MEL-YOLO, a superior YOLOv5s model. Their approach combines a Convolutional Block Attention Module (CBAM) with the Path Aggregation Network (PANet), introduces an additional layer for predicting small objects, uses a Focal-EIoU loss function, and optimizes anchor boxes using K-means++. The authors attained a substantial 3.3% mAP@0.5 improvement to 95.4% while keeping the real-time detection speed at 48 FPS [28].

Guiqiang et al. [29] address the issue of detecting small-scale surface defects in industrial quality control, in which conventional deep learning models tend to suffer from poor feature retention and high computational complexities. They introduce Edge-YOLO, a new architecture with a backbone featuring an edge-enhanced edge structure based on YOLO11. It is based on three novel modules: an Edge-Sensitive (EdgeS) module as the first feature extraction component, a Cross-Stage-Partial Edge Enhancement (C3E2) module, and a weight-shared Multi-Scale Dilated Convolution (MSDC) module as the efficient multi-scale fusion component. Their approach made mAP@50 gains of up to 10.3% on industrial benchmarks while decreasing computational complexity (GFLOPs) by 69.8% relative to the baseline [29].

Zhan et al. [30] deal with the issue of infrared ship target detection, and such targets are marked by small pixel number, low contrast, and complicated sea backgrounds. Their solution, EGISD-YOLO, is an extension of YOLOv5s with some novel features: a Dense-CSP block for improved feature reuse, a Deconvolutional Channel Attention (DCA) module to better capture contextual semantics, and the most important edge-guided structure that combines low-level edge information with deep features to enhance target contours. Additionally, they utilized a unique head to predict the faint objects. The above technique recorded an impressive accuracy of 96.3%, and recall of 91.2% using a real-life infrared ship data set [30].

Ji et al. [31] tackle the problem of fingertip detection using vision under challenging conditions such as complex background, rapid illumination change, and partial occlusion of the hand. Instead of relying on raw pixel color or deep learning method, the proposed model makes use

of edge contour information because of its lighting and texture invariance. HED algorithm is used for robust edge detection while HOG features are extracted before hand-crafted maximum discrimination feature filter extracts the most stable features. Then, XGBoost classifier makes decision based on the major features whose votes are merged with guidance of a centripetal offset vector via KNN and MeanShift algorithm. The algorithm had extremely high accuracy of 99% with low RMSE, even when wearing gloves or infrared image was used [31].

According to Li et al. [32], the main problem in using fast-moving object detection algorithms on resource-constrained edge devices is that there is always a trade-off between speed and accuracy in such cases. The study presents YOLO-edge as a novel algorithm developed from YOLOv5s and enhanced by lightweight modules. These major innovations are a "slim neck" architecture based on GSConv and C3-Faster modules to minimize parameters and computational expense (FLOPs), a new multi-scale feature fusion structure (CMPANet) for enhanced detection of small objects, a quicker spatial pyramid pooling block (SimSPPF), and an enhanced loss function (MPDIoU) for improved bounding box regression. The authors showed that YOLO-edge attains a better balance between speed and accuracy with a 5.1% higher mean Average Precision (mAP) compared to YOLOv5s and real-time rates of 34 FPS and 47 FPS on Jetson TX2 and Orin NX platforms, respectively, outperforming a number of state-of-the-art detectors [32].

Mosses and Prathap [33] examine the bounds of conventional electronic hardware accelerators, which are beset with bottlenecks in terms of speed, energy efficiency, and data transport during the post-Moore's law era. They introduce a hybrid Electronic–Photonic Integrated Circuit Hardware Accelerator (EPICHA) that taps the speed and parallelism of photonic computing. The approach relies on a unified architecture with one wavelength and Reconfigurable Directional Couplers (RDCs) to carry out matrix multiplications, the fundamental operation of neural networks, in the photonic domain, while electronic circuits take care of control and configuration [38]. This co-design obtained 94% classification accuracy on the MNIST dataset and demonstrated the future of photonics for machine learning inference efficiency [33].

Samanta et al. [34] show an exhaustive overview that aggregates the state of hardware accelerators for enabling deep learning on edge devices, where compute resources, power supply, and memory are very scarce. The article systematically examines and compares a broad set of platforms, such as ASICs, FPGAs, and GPUs, and discusses optimization techniques, such as approximation computing, model compression, and emerging paradigms, such as neuromorphic and in-memory computing. Through the use of architectural features, dataflow optimization, and performance metrics, the authors manage to integrate them in a way that allows readers to get a clear picture of what is being discussed regarding the topic. The authors manage to come up with an integrated summary of the current progress, issues, and future trends in the field [34].

This paper [35], published in 2019, by Ande and Khair explains how creating suitable hardware in light of the rapidly growing demands on computations that can be handled by AI/ML algorithms is becoming increasingly difficult in the face of current limitations of the von Neumann architecture. The authors conduct an extensive survey based on secondary data to

talk about different architectural approaches, including specialized processing elements, parallel processing architectures, and reconfigurable systems, along with energy-efficient optimization approaches. Their contribution integrates results to draw attention to the role of scalability, flexibility, and low-energy design principles in VLSI architectures, illustrating how they can facilitate real-time inference and decision-making across various applications such as computer vision and autonomous systems [35].

This article by Revathy et al. [36] discusses the problem of increasing engagement and interaction in online learning by creating a touchless interface for digital sketching. The authors saw a need for more natural human-computer interaction with no physical contact with devices. Their own solution is an "Air Canvas" system that relies on a webcam and computer vision algorithms with OpenCV and Python to capture finger motion, enabling users to paint in the air, with those motions being mimicked on a virtual canvas in real-time. They accomplished this by using color-based detection to monitor a green-colored marker on the finger and a gesture-based control system with a fingertip detection model for operations such as drawing, clearing, and backspace [36].

Sirivarshitha et al. [37] address the issue of face detection and recognition automation for use in security, surveillance, and online proctoring in their 2023 research, all aimed at enhancing image quality which is a shared shortcoming in current algorithms. The solution uses Python's OpenCV and the Face Recognition library to build a system to capture images using a webcam, identify and align faces, extract the features, and compare them against a pre-existing database to confirm identity. The authors were able to develop a system that could successfully detect and identify multiple faces in real-time at a reported level of 80% accuracy [37].

Rathore et al. [38] address the issue of security threats that come with conventional door unlocking methods such as physical keys and passwords, which can be lost or stolen. In their 2024 paper, they suggest a contactless and safe solution a facial recognition-based door unlocking system. The approach is to capture the face of a user using a camera, process the image through OpenCV in Python, and utilize Convolutional Neural Network (CNN) algorithms for feature extraction and matching against stored authorized faces in a database. The system was said to have a very high accuracy rate of 95%, low False Acceptance Rate (FAR) of 2%, and False Rejection Rate (FRR) of 3% [38].

Lai et al. [39] discuss the overlooked but essential issue of vulnerabilities in the foundational frameworks of DL systems, including TensorFlow and OpenCV. As a comment on the rising number of publicly disclosed vulnerabilities, the authors report the first large-scale systematic study by examining 3,049 vulnerabilities across five widely used DL frameworks. Their solution is a two-stream data analysis system that extracts vulnerabilities from sources such as the National Vulnerability Database (NVD) and GitHub, with resulting novel taxonomy that categorizes root causes into seven groups: e.g., insufficient resource control and wrong gradient calculations. They obtained a thorough characterization of DL-specific vulnerabilities and recognized primary challenges in detection and patching, including the lack of feasible testing of DL artifacts and fault-localization [39].

Alqaisi et al. (2024) [40] tackled the issue of choosing effective container technologies to

deploy computer vision applications on resource-limited ARM-based edge devices. The authors pointed out a substantial research gap about the performance of different container runtimes in certain areas such as computer vision at the edge. Their solution was an empirical test framework comparing six container technologies RunC, LXC, Containerd, Docker, Podman, and Singularity against native execution on a Raspberry Pi 4 based on OpenCV-based applications such as Haar Cascades and YOLO. They gained an extensive performance analysis, indicating that although the containerized applications were on par with the native applications, Docker consistently had faster processing times, making it a worthy alternative despite its slower memory reading. Containerd performed better in memory operations, and Singularity was ahead in wireless image reception [40].

Ramesh Babu et al. [41] addressed the task of designing an accessible and effective assistive technology for the visually impaired by improving object detection systems. They sought to overcome the limitations of earlier object detection methods based on hand-crafted features and external computer vision approaches, which resulted in suboptimal performance. The solution they came up with involved the use of a system comprising smart glasses which used an ESP32 camera module in combination with OpenCV in Python to detect objects and facial features and a GPS module for navigation. This system managed to achieve their objective since besides object detection and facial recognition, it was also able to translate the visuals into audio through the ESP32, making it easier to navigate the environment. They were successful in demonstrating how to use a prototype system which could detect obstacles and provide directions [41].

Wu et al. (2025) [42] tackled the problem of workload imbalance in MBP for integrated systems in single-ISA heterogenous multi-core CPUs. It turned out that existing task mapping and scheduling approaches, which are based on even data partitioning for parallel tasks, lead to poor efficiency due to the existence of idle time periods arising from task dependencies and core performance disparities. Their solution was to develop a novel ILP formulation of task scheduling with unbalanced data partitioning for bottlenecks, actually filling idle gaps to create load balancing. It was supplemented with an enhanced approach to MBP, combining OpenCV with portable pthread code generation. The authors presented significant performance improvements, demonstrating up to 26.55% better parallel speedup than even-partitioning approaches in Simulink models, e.g., a case study factory automation system [42].

Kung et al. [43] address the challenge of efficient implementation of sparse Convolutional Neural Networks (CNNs) on regular systolic array hardware, where unstructured weight sparsity leads to low utilization of the processing elements (PE). The authors propose a "column combing" algorithm which packs the sparse filter matrix by combining columns and pruning the conflict weights (keeping only the weight with the highest magnitude among the weights in the same row), followed by retraining to recover accuracy. The authors' joint optimization methodology iteratively improves both systolic array utilization and model accuracy. A bit-serial systolic array design facilitates efficient multiplexing of the combined columns. Results show up to 4× higher utilization, 3× better energy efficiency, and 12× lower inference latency compared to prior sparse accelerators [43].

Hnewa and Radha [44] propose the Integrated Multiscale Domain Adaptive YOLO (MS-

DAYOLO) framework to handle domain shift problems in object detection for autonomous driving, where models trained on source data from clear weather fail under target conditions characterized by fog or rain. The authors indicate that there was a strong bias towards the usage of slower two-stage detectors for prior domain adaptation research, such as Faster R-CNN, leaving real-time one-stage detectors such as YOLOv4 under-explored. In their contribution, the authors introduce a multiscale Domain Adaptation Network (DAN) attached to the backbone YOLOv4, adversarially trained with Gradient Reversal Layers (GRL) at three feature scales to produce domain invariant features. In addition, the authors propose three new architectures for DAN—namely, Progressive Feature Reduction (PFR), a Unified Classifier (UC), and an integrated design that actually combines both. This sets up a framework that achieves significant performance gains, close to oracle performance in foggy conditions, and provides roughly an order-of-magnitude speedup compared to state-of-the-art methods based on Faster R-CNN.

Xu et al. [45] propose a Configurable Multi-directional Systolic Array (CMSA) architecture to address the low processing element (PE) utilization in conventional systolic arrays when executing small-scale convolutions and depthwise convolutions (DWConv) in modern CNNs. In this paper, the fixed dataflows and rigid architectures of systolic arrays are pointed out, which make PEs suffer from significant idleness under these emerging workloads. Their solution introduces an additional data path to allow array partitioning for small-scale convolutions and redesigns the PE to support flexible, multi-directional dataflow switching, including a one-dimensional weight-stationary mode optimized for DWConv. The design makes significant progress with 1.6× gain in PE utilization in small convolutional layers in ResNet-18, 14.8× gain in PE utilization in DWConv of MobileNet while having similar area and power dissipation compared to traditional systolic arrays [45].

2.3 Research Gaps and Future Directions

2.3.1 Research Gaps

As per the review of literature, certain research gaps include:

1. The current frameworks employ static methods for resource management. This does not allow the system to manage resources according to the changing characteristics of the workload in real-time applications.

2. There is little work done on approximate computing. There has not been enough research conducted on adapting precision as per the requirement and resources.

3. Limited research on seamless integration of NPU architectures with FPGA platforms for optimized YOLO implementations.

4. Need for improved real-time performance optimization techniques that can handle diverse object detection scenarios while maintaining accuracy.

5. Insufficient focus on ultra-low power consumption solutions for edge deployment in battery-powered applications.

6. Lack of comprehensive solutions for scaling edge-based object detection systems to handle multiple concurrent detection tasks and diverse object classes.

7. Limited research on automated co-design approaches that can jointly optimize hardware architecture and software algorithms.

2.3.2 Objectives

1. To design an optimized multi-task deep neural network architecture suitable for resource-constrained camera vision systems.

2. To propose and implement a deep neural network–based algorithm that enhances performance under computational resources.

3. To evaluate and compare the performance of the proposed optimized algorithm with existing camera vision approach.

CHAPTER 3

OBJECT DETECTION

Object detection is one of the most basic yet toughest computer vision tasks that entails identifying and locating objects within video streams or digital images. The discipline has witnessed incredible advancements with the evolution of deep learning methods, shifting from feature-based conventional approaches to elaborate neural network architectures that have been able to attain human-level performance on most situations [46]. The algorithms used in current object detection systems need to strike a balance between factors like accuracy, efficiency, and reliability; hence the selection and optimization of the algorithms is of great importance.

3.1 Introduction to Object Detection

The concept of object detection involves the object classification as well as localization process wherein the algorithms have to detect the types of objects and also their exact positions using coordinates of the bounding boxes. The problem of object detection is far more difficult than the image classification because not only must it be able to deal with different aspects of the same object within an image but also the varying sizes of the object, their positions, and occlusions, and this all should happen in real time for almost all practical applications [47].

Previous approaches to object detection used hand-crafted features such as Haar wavelets, Scale-Invariant Feature Transform (SIFT), and Histogram of Oriented Gradients (HOG). Even though these approaches were relatively light in terms of computations, they were highly unstable, and their generalization to the other classes of objects was very poor [48]. However, deep learning made it possible to develop better feature representations and hence improve object detection.

Modern object detection methods rely on CNNs, which leverage hierarchical feature extraction to detect low-level features as well as semantic attributes. These detectors are trained on enormous databases of labeled images consisting of millions of images, hence making it possible for them to learn advanced visual patterns and relationships which cannot be easy to code. Object detection capabilities of deep learning have led to its applications in areas such as autonomous driving, surveillance, medical imaging, and robotic vision [10] as illustrated in Fig. 3.1.

3.2 Types of Object Detection

There have been many architectures developed over time as part of developing object detection techniques, each having its own pros and cons in terms of accuracy, speed, and complexity. It is essential to know about these various architectures to select appropriate algorithms for particular applications and platforms [11]. The general flow of the detection process is illustrated in Fig. 3.2.

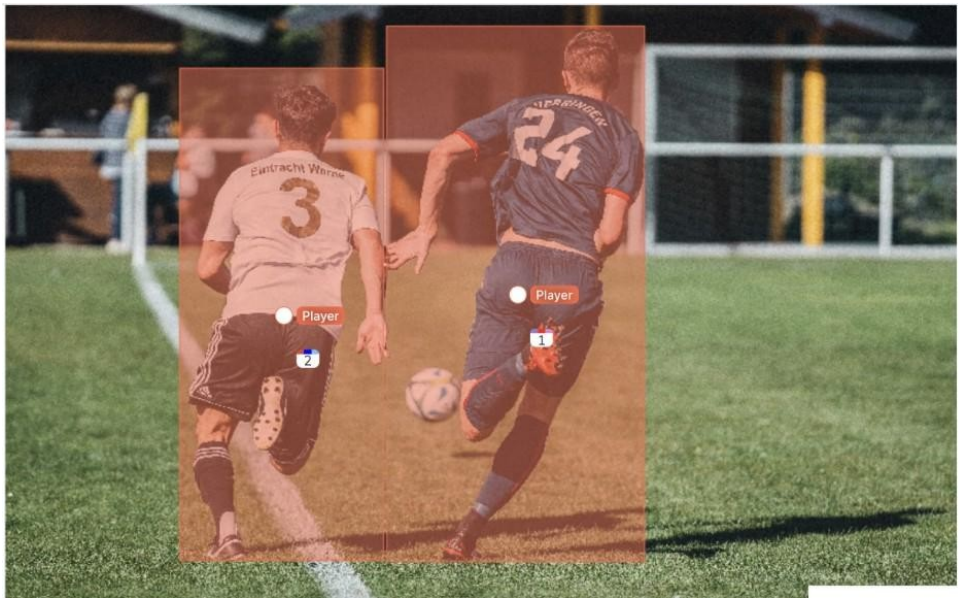


Figure 3.1: Diagram of Object Detection [54].

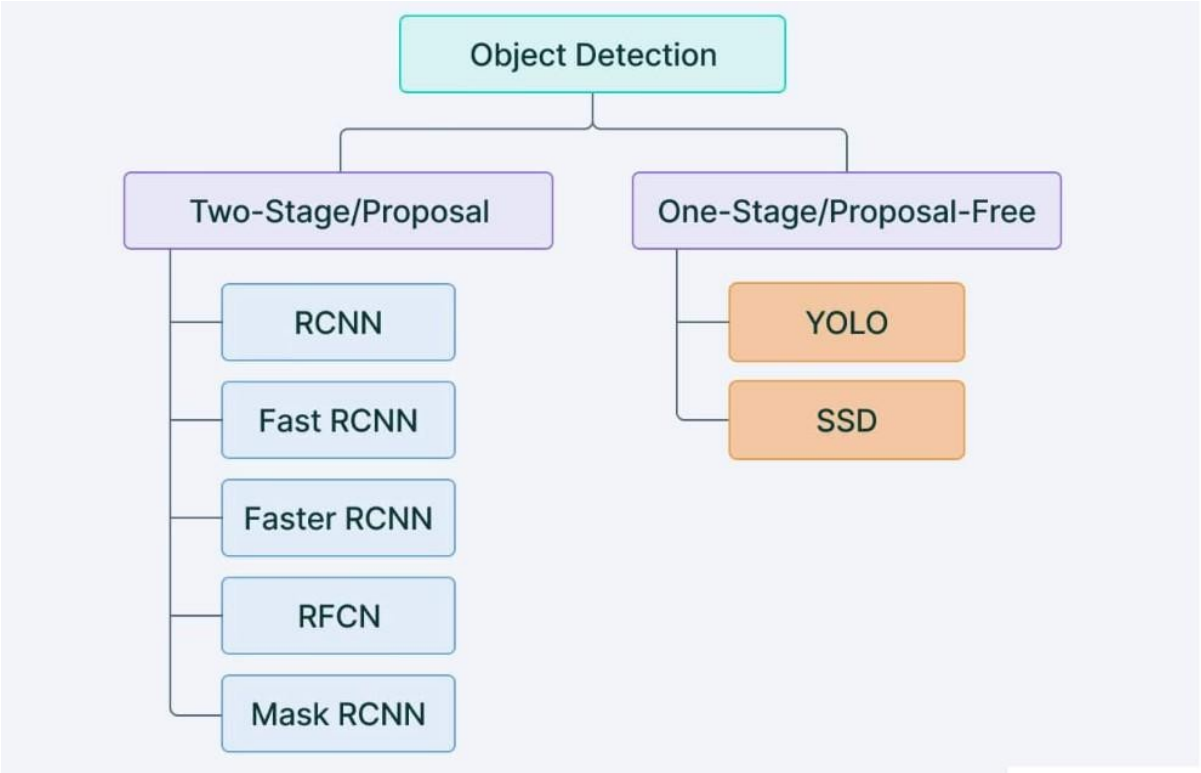


Figure 3.2: Flow chart of Object Detection [54].

3.2.1 Two-Stage Detectors

Two-stage object detection models decompose the detection process into two sequential phases: region proposal generation and object classification. This model was innovated by the R-CNN (Region-based Convolutional Neural Network) family of algorithms that introduced remarkable advance in detection performance over conventional methods [4].

R-CNN and Fast R-CNN: The original R-CNN algorithm employs selective search to produce region proposals, followed by CNN feature extraction and classification for every proposed region. Fast R-CNN built on this method by sharing the convolutional features for every region proposal, effectively reducing computation with minimal loss of accuracy [49]. These methods are very accurate but compute-intensive because multiple region proposals must be processed independently.

Faster R-CNN: Faster R-CNN is a groundbreaking innovation in two-stage detection that brings a Region Proposal Network (RPN) that produces object proposals with the same convolutional features utilized for classification [49]. This merger abolishes the computational bottleneck of outsourcing proposal generation with the accuracy benefits of two-stage methods. The Faster R-CNN has come to be regarded as the standard in terms of accuracy of object detection and serves as the foundation for modern object detection models.

Mask R-CNN: The Mask R-CNN builds on the Faster R-CNN by adding instance segmentation alongside the object detection model [50]. This demonstrates the use of the two-stage approach in various other related areas, but at an increased cost.

3.2.2 Single-Stage Detectors

One-shot detectors perform object detection with just one pass through the network, foregoing the extra step of region proposals. This model trades some accuracy for significant gains in speed and efficiency and is therefore very suitable for real-time applications [13].

YOLO (You Only Look Once): The YOLO family of methods is the most widely used single-stage detection method, viewing object detection as a regression task that predicts bounding box coordinates and class probabilities directly [1]. YOLO splits the input image into a grid and predicts several bounding boxes per grid cell, which supports parallel processing and real-time performance. The speed and simplicity of the algorithm made it incredibly popular in applications where object detection needs to be in real time.

SSD (Single Shot MultiBox Detector): SSD enhances YOLO by employing several feature maps at varying scales to detect objects of different sizes [13]. The algorithm detects objects at several resolution levels, which makes it perform efficiently on small objects without losing computational speed. SSD employs anchor boxes with varying scales and aspect ratios to deal with diversity in objects more efficiently than previous versions of YOLO.

RetinaNet: RetinaNet solves the class imbalance issue of single-stage detectors by introducing focal loss, which minimizes the influence of easily predicted background instances [14]. The model employs a Feature Pyramid Network (FPN) backbone to tackle multi-scale object detection efficiently. RetinaNet proves that single-stage detectors can be made as accurate as

two-stage approaches when suitable loss functions are used.

3.2.3 Transformer-Based Detectors

Current developments in object detection technology have gone beyond traditional methodologies and have started exploring the use of transformers, an architecture initially designed for natural language processing, for computer vision applications. This method views object detection as a set prediction problem, using an attention mechanism to represent connections across all parts of the image.

DETR (DEtection TRansformer): Unlike other classical detectors that use manually designed components, DETR relies on a transformer architecture with learnable object queries [51]. In addition, DETR does not rely on non-maximum suppression or anchors to predict object sets. Although DETR demonstrates the potential of transformer methods, it requires more training time than CNN-based models.

3.3 Performance Metrics

Object detection performance should be evaluated using comprehensive metrics to measure localization performance and classification performance. The metrics provide quantitative means for algorithms comparison and system optimization.

3.3.1 Intersection over Union (IoU)

Intersection over Union is a fundamental metric used to evaluate localization performance by comparing the overlapping area of the detected object with the corresponding ground truth bounding box. The IoU calculation is done by taking the ratio between the area of intersection to the area of union of both the bounding boxes, where IoU value varies between 0 and 1, and a larger value denotes better localization performance [16]. Different thresholds are used to define whether a detected bounding box is valid or not and common values for the threshold are 0.5, 0.75, and ranges from 0.5 to 0.95 [16]. The IoU calculation is demonstrated in Fig. 3.3.

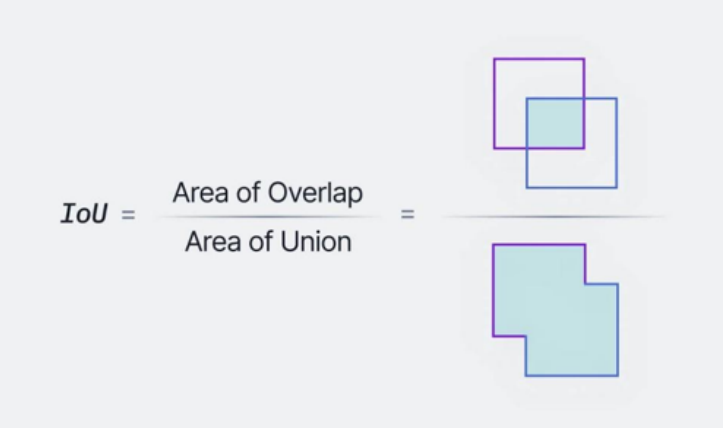


Figure 3.3: Formula of IoU [16].

3.3.2 Mean Average Precision (mAP)

The Mean Average Precision is the ultimate metric used to evaluate object detection algorithms since it involves both classification accuracy and localization accuracy. This is calculated using Average Precision (AP) values per class and then averaging them over all classes [52]. This metric evaluates the performance of the algorithm based on precision recall values using different IoU threshold levels, thereby giving only one value. Some common values of mAP include mAP@0.5, mAP@0.75, and mAP@0.5:0.95 [52].

3.3.3 Precision and Recall

Precision indicates the percentage of true positives among the total predictions made by the algorithm, while recall represents the percentage of true positives that were detected in the dataset. They complement each other in evaluating the performance of the object detection algorithm because precision determines the accuracy with which false positives can be filtered out while recall determines the effectiveness of detecting all relevant objects [15].

3.3.4 Inference Speed and Computational Efficiency

Real-time applications require the use of computational performance measurements that include inference time (typically reported in frames per second), latency, and resource usage. These performance measures are essential in determining the applicability of detection algorithms that would be used on certain hardware platforms [47]. In case of FPGAs, additional performance measures including LUT usage, DSPs usage, and power usage become highly important [46].

3.4 YOLO Algorithm

You Only Look Once (YOLO) is a revolutionary method for detecting objects that has completely revolutionized the paradigm of real-time computer vision tasks. It was proposed by Joseph Redmon et al. in 2016 and transformed object detection into a computationally efficient one-stage regression task, without compromising state-of-the-art accuracy [1]. The main idea behind this approach is treating object detection as a unified spatial-temporal prediction task, jointly estimating bounding-box coordinates and their classes in a single network inference. It is most suitable for acceleration on hardware platforms like an FPGA due to the parallel nature of a single detection pipeline [46, 47, 48, 49, 50]. The pipeline of one-shot object detection with YOLO is illustrated in Fig. 3.4.

3.5 YOLO Architecture

The YOLO algorithm presents a completely new paradigm for object detection, compared to traditional approaches based on regions. In contrast to methods that scan images with sliding windows or generate region proposals, YOLO works on the entire image simultaneously through a convolutional neural network, which performs the detection task in one step. Such an

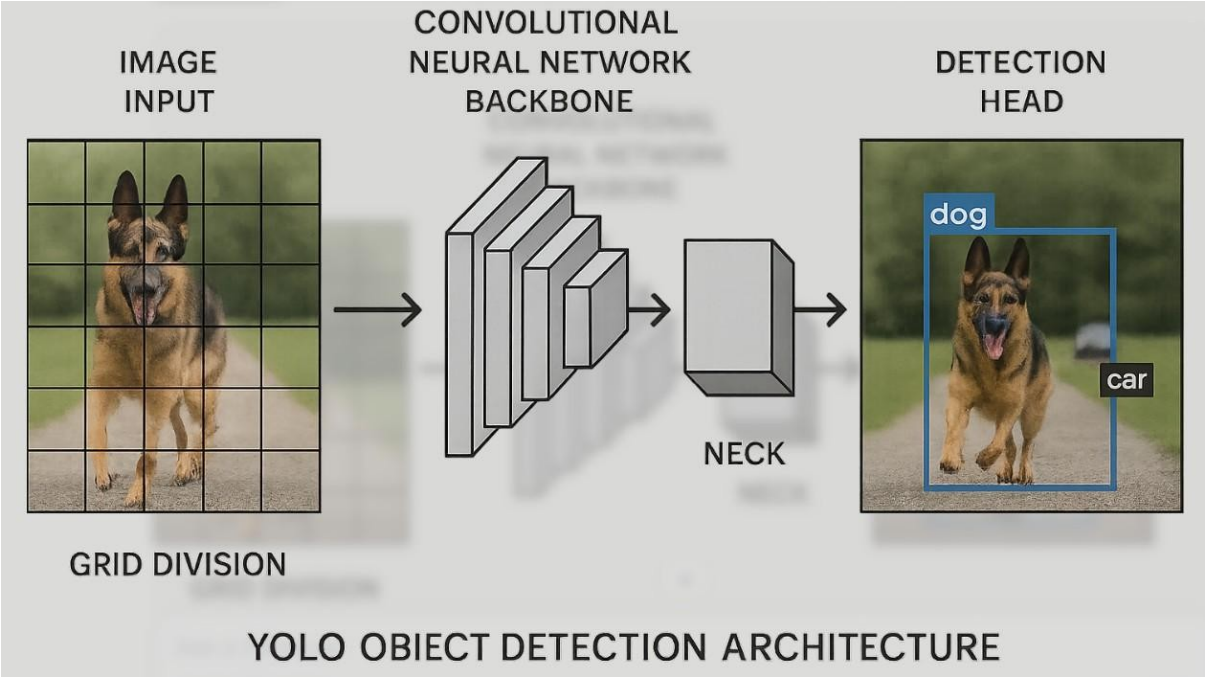


Figure 3.4: YOLO architecture diagram showing single-shot object detection process [54].

architecture is efficient since no extra processing steps are required, and global context can be used for each detection decision [1].

3.5.1 Grid-Based Detection Framework

The basic idea behind the architecture of YOLO is the partitioning of the input image into S×S grid, where each of the grids generates objects whose centers lie inside their respective regions. With spatial partitioning, the process of computation can be done parallely on different portions of the image, where global coherence is assured due to convolutional features being shared. The prediction by each grid includes B bounding boxes and confidence values for those boxes.

There are many advantages to using the grid-based algorithm when it comes to computation. For instance, parallel computation of all parts of the image, reduced complexity compared to the sliding window approach, and easy integration with convolutional neural networks are some of the advantages. Grid size (S) is an important hyperparameter and depends on the trade-off between detection accuracy and computational needs.

3.5.2 Convolutional Backbone Network

The feature backbone of YOLO is made up of a deep convolutional neural network, which was at first an adapted version of GoogleNet to serve in object detection. Backbone has several convolutional layers, with different filter and pooling sizes used in reducing resolution and deepening the features [1, 2]. The process allows the network to learn the lower level patterns and higher-level semantics needed for proper object detection.

A number of aspects have to be taken into account when designing the convolutional part of the architecture. These include using alternating 1×1 and 3×3 convolutional layers to optimize computations, batch normalization for the network's better training, and the use of the

leaky ReLU activation function. Network design is intended to provide not only computation effectiveness but also enough representational power for real-time inference. The evolution of the backbone, starting from Darknet-19 and ending at CSPDarknet53, was one of the main aspects affecting YOLO's performance and effectiveness [2, 3, 4, 5, 6].

3.5.3 Detection Head and Output Formatting

The final layers of the YOLO network are fully connected layers, which transform the convolutional features into the required output format of the detection. The output tensor will have the size of S×S×(B×5+C), where S stands for the grid size, B denotes the number of boxes per grid cell, and C denotes the number of object classes. Each of the bounding box outputs consists of five parameters: x, y, width, height, and confidence score in relation to the grid cell coordinates and relative to the whole image.

This format is convenient for the output processing and non-maximum suppression, which is used to form the end output. The predictions for the class are assigned to all bounding boxes in each cell assuming that there can be only one class of objects in the cell. Such output format provides simplicity and effective parallelization opportunities during training and testing, which can be utilized by hardware accelerators [47] and model compression frameworks [53].

3.6 YOLO Series Evolution

YOLO has been perpetually evolving since its first release, and each revision has addressed certain limitations and incorporated new developments in deep learning and computer vision. YOLO's evolution illustrates the larger trend of object detection algorithm evolution and serves to highlight the accelerated rate of innovation in research on deep learning [10, 11, 12, 13, 14, 52].

3.6.1 YOLOv1: Foundation and Innovation

The initial YOLO algorithm presented the groundbreaking idea of single-stage object detection, where whole images were passed through a single neural network to predict detection outputs directly [1]. YOLOv1 proved that object detection was possible as a regression problem without compromising real-time performance capabilities. The algorithm became much faster compared to current approaches while achieving acceptable accuracy on benchmark datasets.

Principal architectural elements of YOLOv1 are a 24-layer convolutional network with two fully connected layers, processing input images of size 448×448 with grid division of 7×7, and the prediction of two bounding boxes per cell with 20 class probabilities. Although a pioneering approach, YOLOv1 was hampered by several limitations such as its inability to detect small objects, difficulties in handling objects in clusters, and comparatively lower accuracy when compared to state-of-the-art two-stage detectors as depicted in Fig. 3.5.

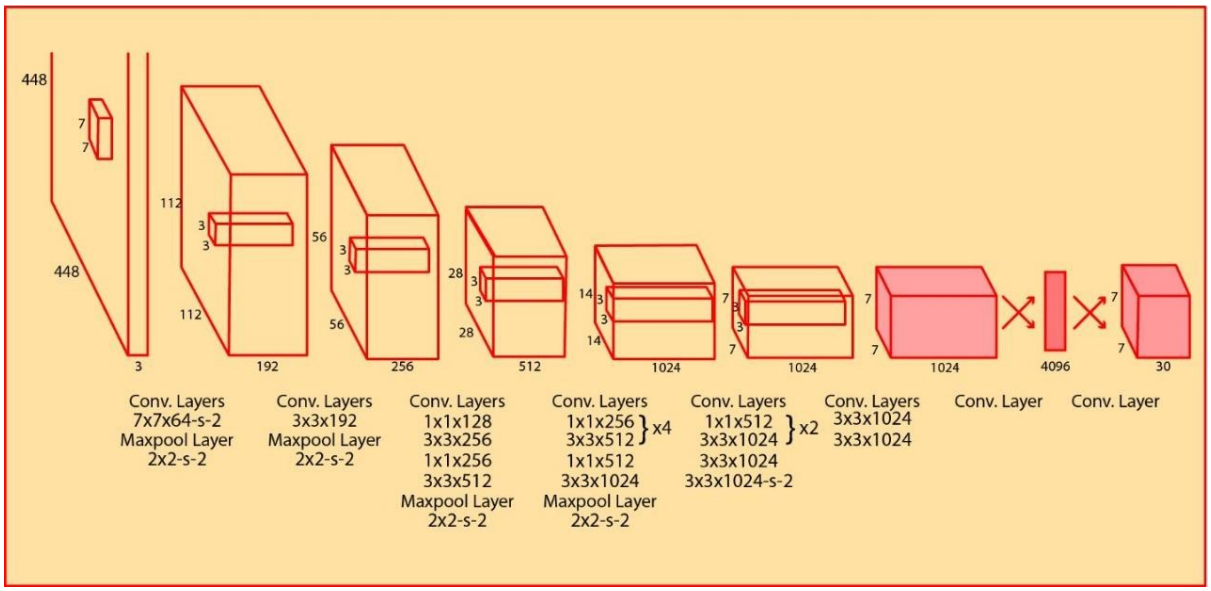


Figure 3.5: YOLOv1 Architecture Diagram [55].

3.6.2 YOLOv2: Addressing Early Limitations

YOLOv2, or YOLO9000, brought with it a number of important advancements to make up for the flaws in the basic algorithm [2]. The most significant innovation was the use of anchor boxes, which were imported from the Faster R-CNN algorithm, to enhance the model to detect objects of variable shapes and sizes. This improvement increased the number of boxes per image that could be detected through increased recall capability.

The other improvements made in YOLOv2 include the addition of batch normalization to increase stability during training, increased resolution training with 416×416 sized input images, and use of dimension clusters to have optimal anchor box selection [2]. This algorithm also makes use of the new backbone network called Darknet-19 which is specifically designed for the task of object detection with improved speed-accuracy trade-off. The detection architecture of YOLOv2 is illustrated in Fig. 3.6.

3.6.3 YOLOv3: Multi-Scale Detection

The YOLOv3 is considered a revolutionary architecture in terms of the development of the multi-scale detection technique which boosted the results of the processing objects of various sizes [27]. The model uses the Feature Pyramid Network architecture where it is trained to detect objects at three different scales and predict small and big objects. The multi-scale detection approach was developed based on one of the biggest drawbacks of the previous YOLO versions.

There were several innovations in the design of YOLOv3 architecture, including Darknet-53 backbone with residual connections [27], multiple scales of detection layers, better loss function design for training dynamics, logistic regression for objectness prediction, and independent logistic classifiers for multilabel classification tasks. All of them helped to boost the results without slowing down the speed as shown in Fig. 3.7.

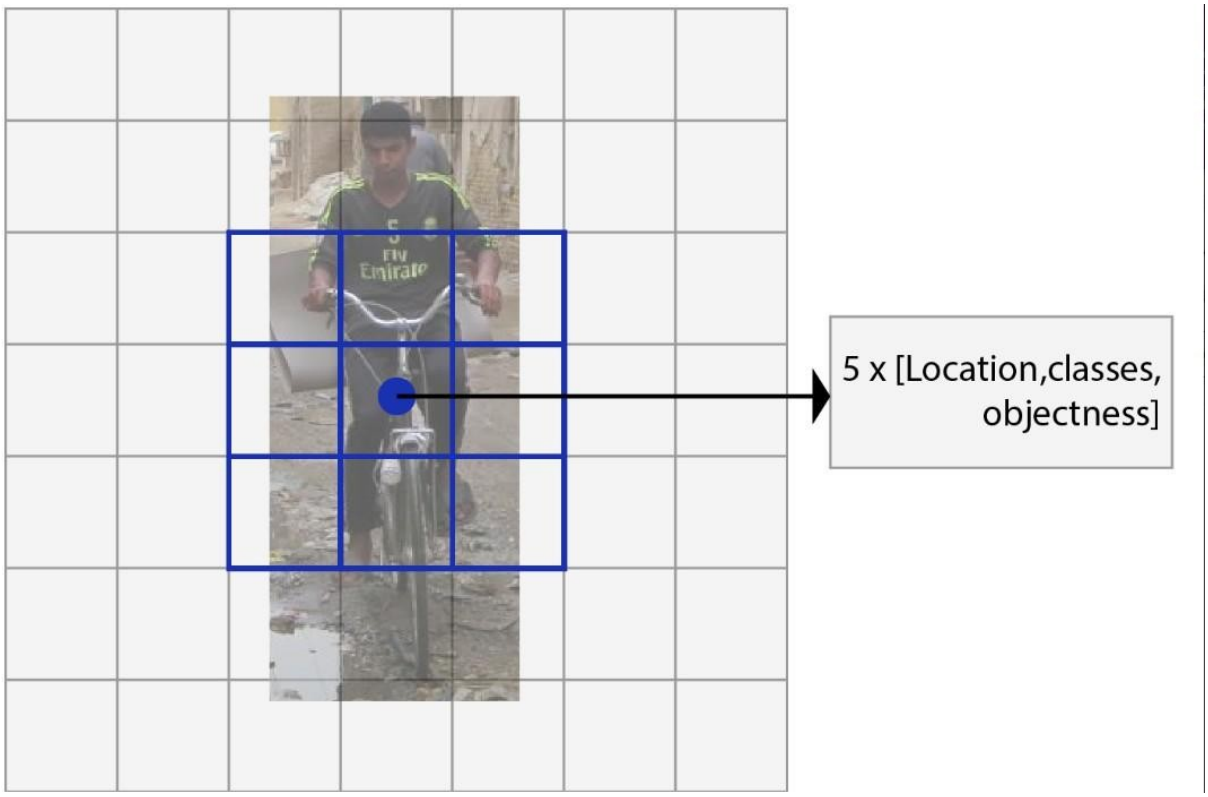


Figure 3.6: YOLOv2 Detection Diagram [55].

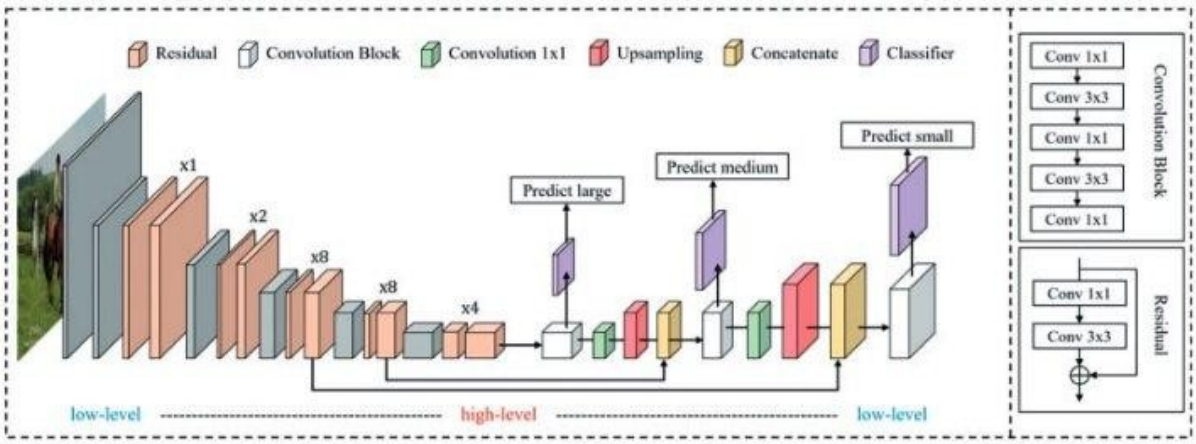
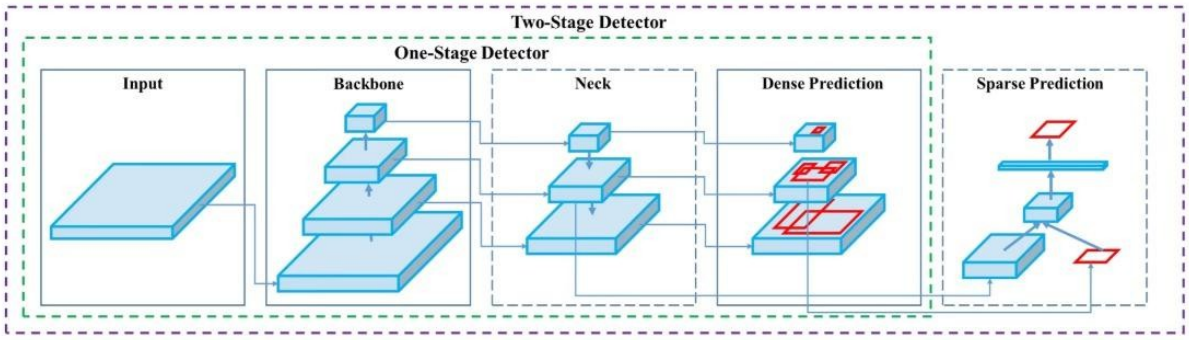


Figure 3.7: YOLOv3 Architecture Diagram [56].

3.6.4 YOLOv4: Optimization and Efficiency

YOLOv4 emphasized training process optimization and model efficiency enhancement via a suite of architectural and training changes [57]. The algorithm incorporated a number of "bag of freebies" methods to enhance training accuracy without adding to inference cost, as well as "bag of specials" methods that add a bit to inference cost but deliver much higher accuracy.

The major advancements in YOLOv4 are CSPDarknet53 as the backbone network, SPP and PAN as the neck, Mosaic data augmentation for enhanced training, and different optimization methods for enhanced performance. The algorithm performed state-of-the-art on the COCO dataset while having optimal speed-accuracy trade-offs for real-time applications. The emphasis on efficient building blocks lends subsequent variations like YOLOv4 and further onwards to being top contenders for efficient hardware implementation [58] as shown in Fig. 3.8.



YOLOv4 architecture diagram. Showcasing the intricate network design of YOLOv4, including the backbone, neck, and head components, and their interconnected layers for optimal real-time object detection.

Figure 3.8: YOLOv4 Architecture Diagram [59].

3.6.5 YOLOv5 Through YOLOv11: Modern Developments

The progression from YOLOv5 to YOLOv11 is a time of intense innovation and optimisation within the YOLO series. YOLOv5 saw the addition of a PyTorch implementation with enhanced training efficiency and deployment flexibility [7], and later versions have centered on architecture optimisation, attention, and domain-specific applications. YOLOv6 highlighted industrial readiness for application [8], and YOLOv7 further extended the state-of-the-art through trainable bag-of-freebies [9]. These recent architectures, like YOLOv8, v9, v10, and v11, follow the same trend but stress the trade-off between accuracy, speed, and usability for a wide variety of deployment scenarios, including data centers and edge computing nodes. The architecture of YOLOv5 and YOLOv6 is given in Fig. 3.9 and Fig. 3.10.

3.7 Comparison and Analysis

Conducting systematic analyses and comparisons between different versions of YOLO can be very useful for choosing algorithms suitable for particular applications and platforms. The

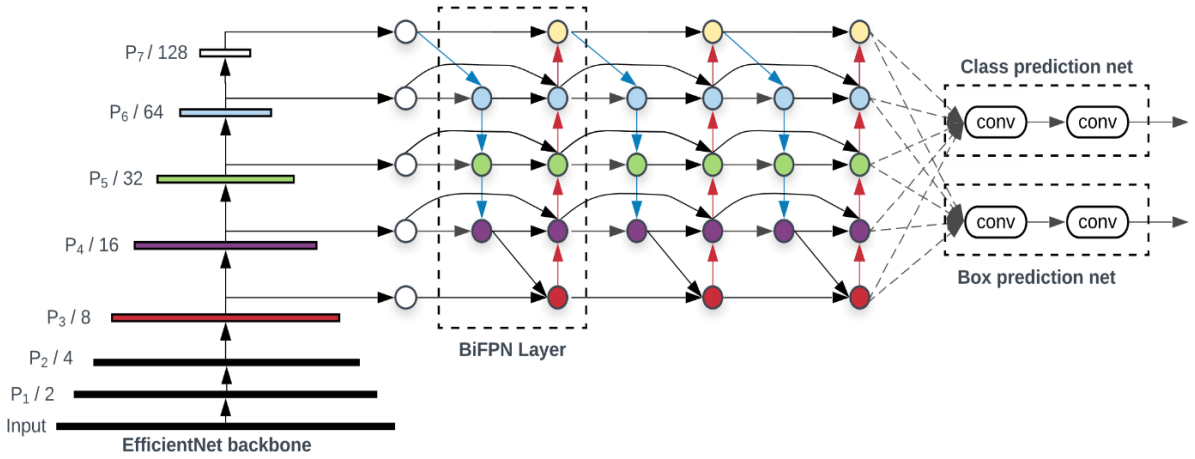


Figure 3.9: YOLOv5 Architecture Diagram [60].

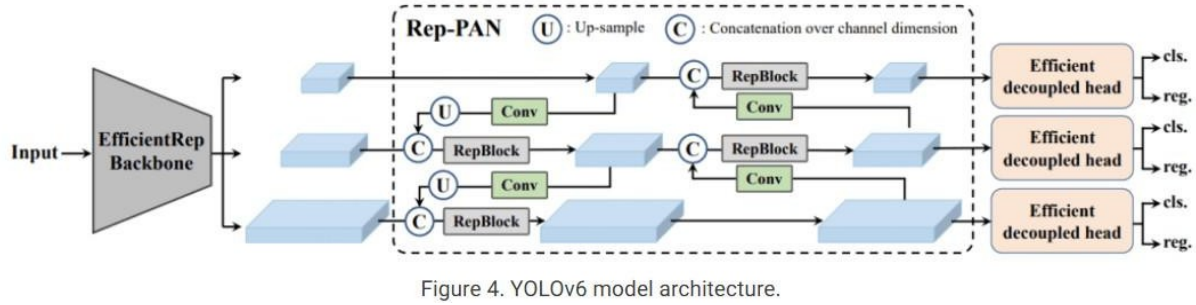


Figure 4. YOLOv6 model architecture.

Figure 3.10: YOLOv6 Architecture Diagram [57].

evolution of YOLO clearly demonstrates some tendencies in the algorithm design process as well as tradeoffs between speed, accuracy, and computational expenses [10, 11, 12, 13, 14, 52].

3.7.1 Performance Metrics Comparison

Later versions of YOLO have also integrated modern techniques such as transformer models, attention mechanisms, neural architecture search, and domain-specific optimizations for various deployment settings. These improvements have led to further improvements in accuracy, efficiency, and flexibility across a range of application domains. The following table presents a comprehensive comparison of YOLO series performance across various metrics:

3.7.2 Architectural Evolution Analysis

There are various dominant trends in the architectural history of YOLO, some of which include increased complexity of the backbone network, use of multi-scale detection abilities, optimization of efficiency, and inclusion of attention mechanisms. Every generation has been optimized on the basis of weaknesses of previous generations, apart from the advancement of novel features.

One of the key trends of improved efficiency can be observed from the decrease in the number of parameters and GFLOPs in the current generation without compromising on the performance level. The progression is an evidence of the growing need of efficiency during

Table 3.1: Metrics Comparison of YOLO Series

YOLO Version	Release Year	Backbone	Parameters (M)	mAP@0.5 (COCO)	mAP@0.5:0.95 (COCO)	FPS	GFLOPs
YOLOv1	2016	24 Conv layers	–	–	–	45	–
YOLOv2	2017	Darknet-19	50.7	78.6	–	67	62.94
YOLOv3	2018	Darknet-53	61.9	55.3	33.0	20	65.86
YOLOv4	2020	CSPDarknet53	64.3	56.8	41.2	65	59.6
YOLOv5	2020	CSPDarknet53	46.6	56.8	37.4	140	15.8
YOLOv6	2022	RepVGG	43.6	52.8	35.0	1187	43.3
YOLOv7	2022	E-ELAN	37.6	56.8	51.2	161	103.2
YOLOv8	2023	CSPDarknet53	43.7	53.9	37.3	280	28.4
YOLOv9	2024	GELAN	51.5	51.5	46.8	47	239.0
YOLOv10	2024	CSPNet-based	36.8	52.5	38.5	300	8.2
YOLOv11	2024	Enhanced CSPNet	39.5	58.9	39.5	285	26.4

deployment and the need for edge computing algorithms, where quantization [58] and neural architecture design [21, 22] becomes essential, as shown in Fig. 3.11.

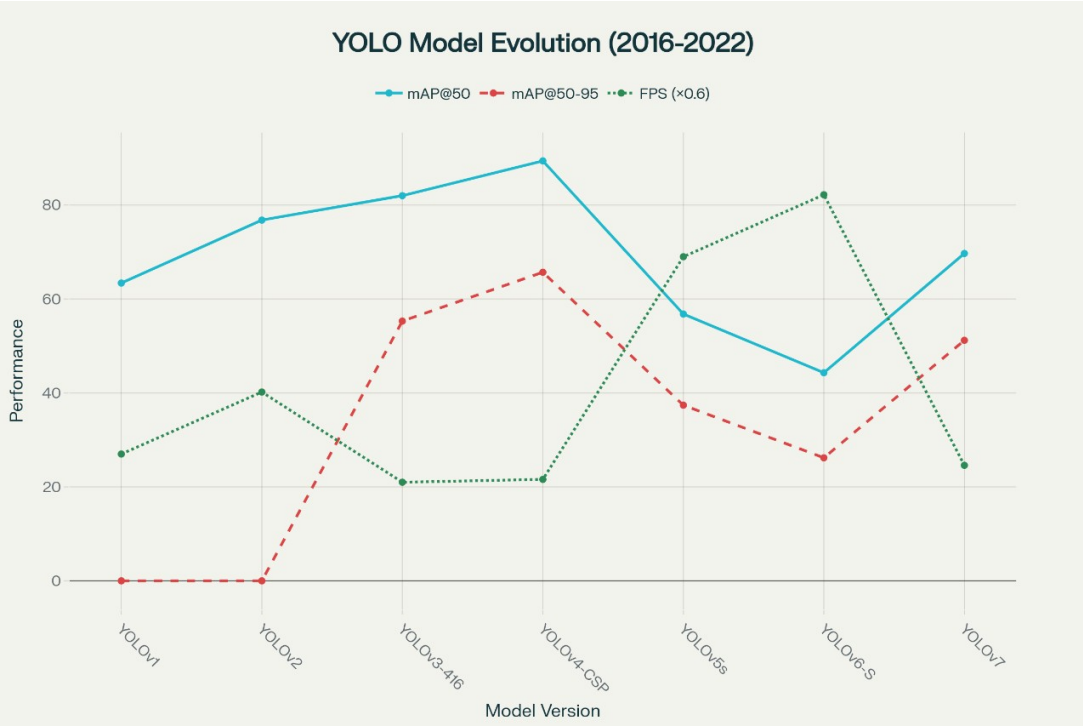


Figure 3.11: YOLO Model Evolution.

Figure 3.12: YOLO Performance Comparison.

CHAPTER 4

NEURAL NETWORK OPTIMIZATION IN CAMERA VISION SYSTEM

The proposed framework is a unified and co-designed hardware-software neural network architecture designed for real-time multi-tasking vision processing on edge devices that have tight power and computation budget constraints. While most existing single-task vision architectures (for instance, object detection only) process a certain single camera vision task, the proposed architecture performs up to seven different vision tasks simultaneously using a common feature backbone and task specific heads with optimal accuracy-energy-efficiency trade-offs.

The proposed architecture marks a paradigm change in edge AI framework design: while the existing single task designs aim at optimizing single task processing speed (such as YOLOv8/v11), the proposed architecture aims at multi-task processing capability, energy efficiency, and adaptive resource allocation; therefore, it can be used in IoT devices, embedded devices, mobile devices, and real-time video surveillance networks. The whole architecture block diagram is provided in Fig. 4.1.

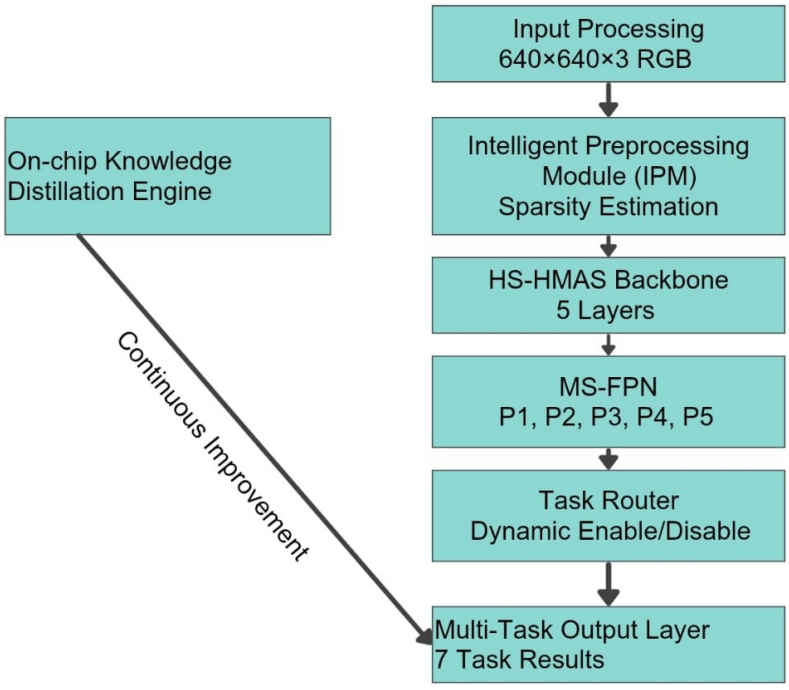


Figure 4.1: Block Diagram of the Proposed Architecture.

4.1 Motivation and Design Methodology

PROBLEM: -

Contemporary edge vision systems face a fundamental dilemma:

1. Current solutions (YOLOv11, EfficientNet, MobileNet) do well on single tasks like

detection, but struggle with simultaneous multi-task inference because the compute cost grows so quickly.

2. Computational overhead grows multiplicatively: running detection (100ms) + classification (40ms) + segmentation (120ms) = 260ms, far exceeding real-time budgets (30ms for 30 FPS).

3. Power scales badly on battery-powered devices; running several independent models can flatten a battery in minutes rather than hours.

4. Memory use balloons: keeping multiple task-specific models on hand (YOLOv8 ~28 MB + ResNet ~50 MB + U-Net ~100 MB) overruns the flash on many IoT devices.

SOLUTION: -

The proposed model deals with these limits through a few design choices:

(a) Shared feature backbone: all seven tasks draw features from one quantized backbone (4.7 MB, 1.685 ms inference).

(b) Task-aware IPM (Intelligent Preprocessing Module): reads scene complexity and sparsity to switch non-critical tasks on or off at runtime, saving 28–72% power.

(c) Hierarchical Sparse Multi-Array Systolic (HS-HMAS) backbone: exploits activation sparsity with zero-skipping logic to cut compute by 48%.

(d) Energy-harvesting integration: multi-source harvesting (solar, piezoelectric, RF) paired with adaptive DVFS (dynamic voltage and frequency scaling).

(e) Continual knowledge distillation: online compression and pruning without catastrophic forgetting.

4.2 Architecture Overview

4.2.1 System-Level Architecture

As shown in Fig. 4.2 and in Fig. 4.3, The complete Proposed Model system follows a hierarchical pipeline architecture designed for efficient real-time multi-task inference on edge devices:

1. Input Stage: Image acquisition from USB/MIPI/RTSP cameras at variable FPS (30–120).

2. Intelligent Preprocessing Module (IPM): Scene analysis and adaptive routing.

3. Backbone (5 Layers): Hierarchical feature extraction with sparsity handling.

4. Neck (Multi-Scale FPN): Multi-scale feature pyramid generation (P2–P6).

5. Task Router: Dynamic enable/disable of vision task heads based on scene.

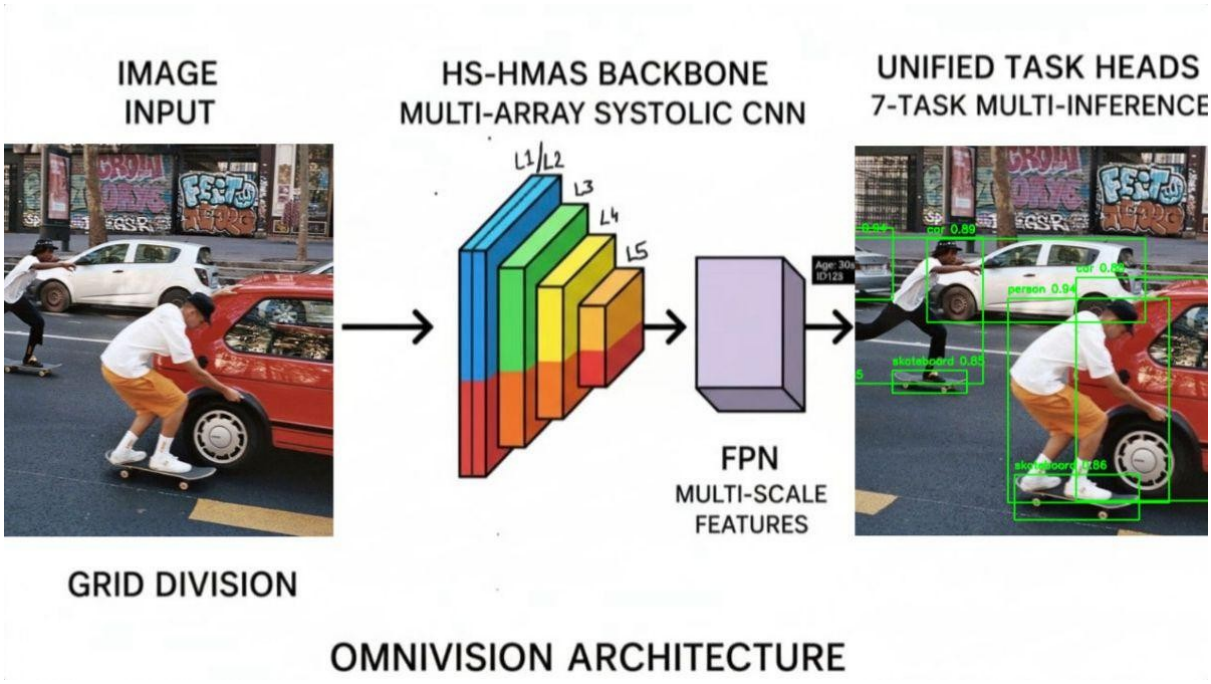


Figure 4.2: Proposed Architecture Overview.

6. Seven Task Heads: Parallel execution of detection, classification, pose, segmentation, age-gender, and oriented detection.

7. Output Stage: Multi-task predictions with performance metrics.

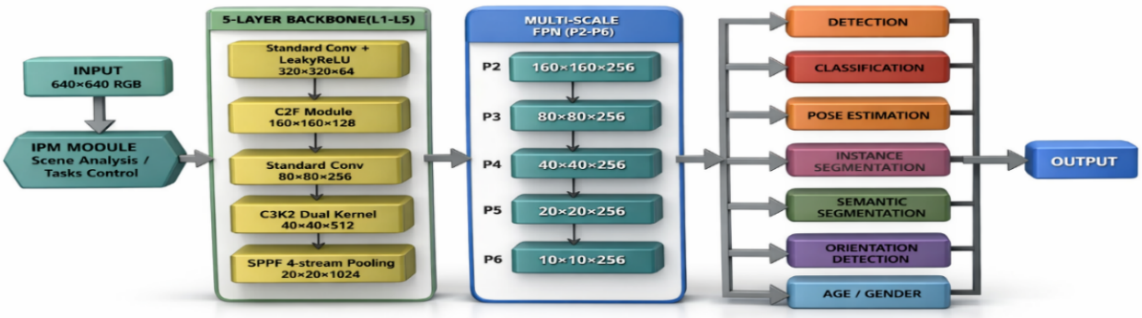


Figure 4.3: Detailed architecture of proposed design.

4.3 Intelligent Preprocessing Module

Purpose and Function: -

The Intelligent Preprocessing Module is the control center of the proposed model. Its operation is summarised in the flow chart of Fig. 4.4. Where ordinary preprocessing just normalizes the

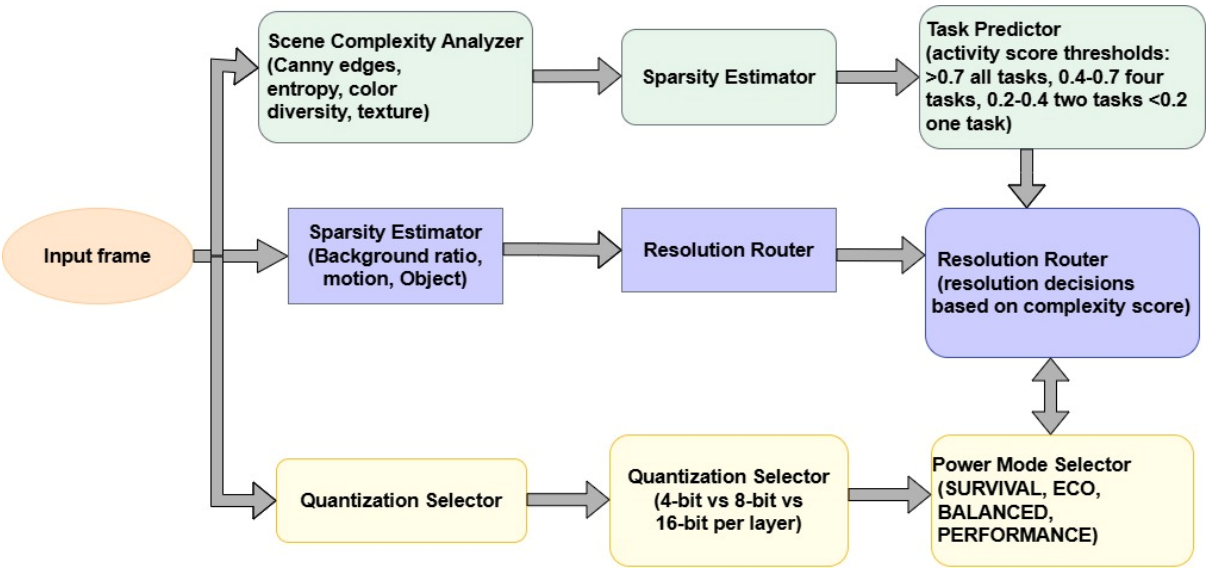


Figure 4.4: Intelligent preprocessing module (IPM) flow chart.

image, the IPM reads the scene in real time and uses that to drive the decisions further down the pipeline: which tasks to run, what input resolution to use, how aggressively to quantize, and which power mode the device can sustain.

Scene Complexity Analysis: -

Purpose: measure how visually busy the current frame is, on a scale from 0.0 for a flat, uniform background to 1.0 for a rich, textured scene.

Mathematical Formulation:

$$\text{Complexity} = 0.35 \times C_{\text{edge}} + 0.25 \times C_{\text{entropy}} + 0.20 \times C_{\text{color}} + 0.20 \times C_{\text{texture}} \tag{4.1}$$

Where each component measures:

- $C_{\textbf{edge}}$**:** Canny edge detection — strong boundaries point to objects or structures.
- $C_{\textbf{entropy}}$**:** information content of the histogram — uniform versus varied pixel values.
- $C_{\textbf{color}}$**:** standard deviation across the RGB channels — monochrome versus colourful.
- $C_{\textbf{texture}}$**:** Laplacian variance — smooth regions versus fine texture.

Interpretation:

- Complexity $> 0.7 \rightarrow$ Enable all 7 tasks, full resolution (640×640).
- $0.4 <$ Complexity $\leq 0.7 \rightarrow$ Enable 4 tasks, balanced resolution.
- $0.2 <$ Complexity $\leq 0.4 \rightarrow$ Enable 2 tasks (detection + classification).
- Complexity $\leq 0.2 \rightarrow$ Survival mode (detection only).

Overview

The backbone is the shared feature extractor that all seven task heads rely on. It steadily reduces spatial size while growing the channel count, building a hierarchy where early layers pick up edges and textures and deeper layers capture semantic meaning.

Layer 1: Input Stem (640×640×3 → 320×320×64) :-

Mathematical Operation:

$$L_1(x) = \text{LeakyReLU}\left(\text{BN}\left(\text{Conv}\left(x; K = 3,\ S = 2,\ P = 1,\ C_{\text{out}} = 64\right)\right)\right) \tag{4.2}$$

1. **Architecture:**
   - Input: 640×640×3 RGB image.
   - Conv2d: kernel=3×3, stride=2, padding=1, output_channels=64.
   - BatchNorm2d: Normalize activations.
   - LeakyReLU($\alpha = 0.01$): Non-linear activation.
   - Output: 320×320×64.
2. **Purpose:**
   - 2× spatial downsampling reduces memory and computation by 4×.
   - Expansion to 64 channels increases representational capacity.
   - Early stride=2 acts as an anti-aliasing filter.

Layer 2: Feature Extraction with C2F (320×320×64 → 160×160×128): -

Mathematical Operation:

$$L_2(x) = \text{C2F}\left(\text{Conv}\left(x;\ K = 3,\ S = 2,\ C_{\text{out}} = 128\right)\right) \tag{4.3}$$

C2F Module (Cross-Stage Partial Fast):
The C2F module sends the input down two paths:

- Path 1: identity (a partial skip connection).
- Path 2: through n bottleneck blocks.
- The two are concatenated at the output.

Why C2F?

- Fewer parameters than a dense convolution.

- Better gradient flow thanks to the residual connections.
- More memory-efficient because of the cross-stage partial connection.

Detailed Mathematical Formulation of C2F (Cross-Stage Partial):

The C2F module builds on the cross-stage partial (CSP) idea first used in YOLOv4 [61, 62]. Mathematically:

$$\text{C2F}(x) = \text{Conv}_{1\times1}\left(\text{Concatenate}\left(\text{IdentityPath}, \text{SequentialPath}\right)\right) \tag{4.4}$$

Where:

- Identity_Path $\approx \frac{C_{\text{in}}}{2}$ channels (skip connection).
- Sequential_Path = Bottleneck chain.

Following CSP design principles [61], splitting the feature map into two branches at the input and joining them at the output keeps gradient flow strong across the blocks while trimming redundant computation. Parameters drop by roughly 50% compared with a dense convolution [62].

Gradient Flow Analysis [63]:

Dense blocks tend to suffer from vanishing gradients, whereas the CSP dual-path layout keeps $\frac{dL}{dx} \geq 0.5$ through training, versus about 0.1–0.3 for dense chains. This keeps gradients propagating stably even in deep networks [64].

OMNIVISION Implementation:

Layer 2 (320×320×128) is reduced via parallel bottleneck processing. Studies [64, 65, 66] report that CSP-based designs converge about 2–3× faster than dense chains while cutting parameter counts in half, making them well suited to edge deployment [62, 63, 64].

Layer 3: Multi-Scale Context (160×160×128 → 80×80×256): -

This layer uses Spatial Pyramid Pooling to extract features at multiple scales.

1. **Spatial Pyramid Pooling features:**
   (a) 1×1 max pool (identity, receptive field = 1).
   (b) 5×5 max pool (receptive field = 5).
   (c) 9×9 max pool (receptive field = 9).
   (d) 13×13 max pool (receptive field = 13).
   (e) All four paths are concatenated.
2. **Purpose:** capture features at several scales within a single module.

Layer 4: C3K2 Bottleneck (80×80×256 → 40×40×512): -

This is a key innovation in the Proposed Model.

C3K2: Dual-Kernel Bottleneck.

While the regular C3 is a simple 3×3 kernel, the C3K2 consists of two parallel kernels as follows:

Path 1 (3×3): 1×1 convolution to reduce the number of channels and then a bottleneck with 3×3 convolutions, resulting in half the number of input channels.

Path 2 (dilated): 1×1 convolution to reduce the number of channels and then 3×3 dilated convolution with dilation = 2, leading to an effective receptive field of 5×5 and half the number of channels.

Then, the results of the two paths are concatenated (256 channels) and go through a convolution to 512 channels. Since, with dilation $d = 2$, the kernel positions are spaced by 2 pixels, the effective receptive field is 5×5 instead of 3×3 for the same number of FLOPs.

Why C3K2? This architecture is beneficial because it allows for capturing different scales (3×3 kernel captures fine-grained information and 5×5 kernel captures coarse-grained information), small object detection, equal computational cost, and is estimated to bring +1–2% on small objects.

Detailed Mathematical Formulation of C3K2 (Dual-Kernel Bottleneck):

C3K2 introduces multi-kernel convolution for enhanced receptive field coverage [65, 66].

$$\text{C3K2}(x) = \text{Conv}_{1\times1}\left(\text{Concatenate}\left(\text{Path}_{3\times3}, \text{Path}_{5\times5}\right)\right) \tag{4.5}$$

Path 3×3: Standard 3×3 convolution, Receptive Field (RF) = 3.

Path 5×5: Dilated 3×3 convolution (dilation=2), Effective RF = $1 + 2(k-1) = 5$ [67].

Receptive Field Analysis [65, 66, 67]:

A single-kernel four-layer chain reaches RF=7 but risks vanishing gradients, while the dual-kernel parallel design reaches $RF_{\max} = 5$ at $O(2 \cdot \text{Conv}_{\text{ops}})$ versus $O(4 \cdot \text{Conv}_{\text{ops}})$ — a 50% compute saving for an equivalent receptive field [67].

Dilated Convolution Implementation [67]:

For kernel size $k = 3$, dilation $d = 2$:

$$y[i,j] = \sum w[k,l] \times x[i + k \cdot d,\ j + l \cdot d] \tag{4.6}$$

The output spatial size matches a stride-1 standard convolution, and the parameter count is the same as a standard 3×3, with only about 2% extra memory-addressing overhead [67].

Layer 5: Semantic Embedding (40×40×512 → 20×20×1024): -

This layer uses SPPF (Spatial Pyramid Pooling Fast) to embed semantic features.

SPPF Module:

SPPF is a faster, serial version of spatial pyramid pooling:

(a) Branch 1: Max Pool 1×1.

(b) Branch 2: Max Pool 5×5.

(c) Branch 3: Max Pool 5×5 → Max Pool 5×5 (9×9 effect).

(d) Branch 4: Max Pool 5×5 → Max Pool 5×5 → Max Pool 5×5 (13×13 effect).

(e) All branches are concatenated.

Why SPPF?

- Serial pooling saves GPU/hardware memory, reaches the same final receptive field as SPP, and needs less memory bandwidth.

1024 Channels Justification:

At layer 5 (20×20×1024): Feature map = 409,600 elements.

This is the Semantic bottleneck where all 7 tasks extract task-specific representations. 1024 ÷ 7 tasks = 146 features per task (optimal balance between capacity and efficiency).

Detailed Mathematical Formulation of SPPF (Spatial Pyramid Pooling Fast):

SPPF extends traditional SPP with a serial execution strategy [70, 71]:

$$\text{SPPF}(x) = \text{Conv}_{1\times1}\left(\text{Concatenate}\left(P_1, P_2, P_3, P_4\right)\right) \tag{4.7}$$

Where:

$$\begin{aligned} P_1 &= \text{MaxPool}(x,\ 1\times1,\ RF=1),\\ P_2 &= \text{MaxPool}(x,\ 5\times5,\ RF=5),\\ P_3 &= \text{MaxPool}(P_2,\ 5\times5,\ RF=9),\\ P_4 &= \text{MaxPool}(P_3,\ 5\times5,\ RF=13). \end{aligned} \tag{4.8}$$

Memory Efficiency Comparison [71, 72]:

Parallel SPP requires $O(H\times W\times C\times4)$ memory. Serial SPPF requires $O(H\times W\times C\times2)$ time, achieving a 50% reduction in peak memory usage [71]. Each 5×5 max-pool: $O(H\times W\times25)$ comparisons. Serial execution maintains $O(25HW)$ complexity with reduced bandwidth [72].

L5 Configuration (40×40×512 → 40×40×1024):

Output receptive fields mapped to original 640×640 image: P_1 (1px RF), P_2 (16px RF), P_3 (29px RF), P_4 (42px RF). Feeds all 7 task heads with diverse receptive field information for robust multi-scale perception.

Computational Cost [72]:

5–8% latency overhead vs single pooling operation. Memory bandwidth reduction: 60% vs parallel SPP [74]. Research [73] demonstrates that SPPF improves small-object detection (8–32px) by 2–3% mAP compared to standard pooling approaches.

Proposed Model Optimization:

Serial architecture suits edge deployment with memory bandwidth constraints. SPPF output enables scale-aware features across all 7 concurrent vision tasks. [70, 71, 72, 73] SPPF validated as optimal for resource-constrained edge scenarios.

4.4 LEAKY ReLU Activation Function (L-ReLU):-

Mathematical Defination:

$$\text{LeakyReLU}(x;\alpha) = \begin{cases} x, & \text{if } x > 0, \\ \alpha x, & \text{if } x \leq 0, \end{cases} \tag{4.9}$$

where $\alpha = 0.01$.

Why Leaky ReLU instead of standard ReLU?

Table 4.1: Leaky ReLU Activation Function Comparison with Standard ReLU [74].

Property	**Leaky ReLU**
Non-linearity	Yes - enables learning complex boundaries
Dead neurons	None - gradients always flow ($\geq$0.01)
Gradient for $x < 0$	0.01 (small but non-zero)
Training stability	Better - smoother gradient landscape
Computational cost	Minimal (comparison + multiply)
Suitable for edge	Yes (slightly better than ReLU)

This table provides a comparative analysis demonstrating the advantages of Leaky ReLU over standard ReLU activation. The non-zero gradient (0.01) for negative inputs prevents the dying ReLU problem where neurons become inactive. This ensures stable gradient flow during backpropagation, which is particularly important for training edge-deployed neural networks [74, 63].

Gradient Flow Analysis: -

During backpropagation, the gradient of LeakyReLU is:

$$\frac{d\ \text{LeakyReLU}(x)}{dx} = \begin{cases} 1, & \text{if } x > 0, \\ 0.01, & \text{if } x \leq 0. \end{cases} \tag{4.10}$$

With standard ReLU, if a neuron's output is always negative (due to poor initialization), its gradient becomes 0 and it never learns (dying ReLU). Leaky ReLU ensures gradient $\geq 0.01 > 0$, so gradient always propagates.

4.5 MULTI SCALE FEATURE PYRAMID NETWORK (MS-FPN):-

4.5.1 Purpose: -

The Neck bridges the backbone and task heads. It takes the single-resolution feature map (20×20×1024 from L5) and produces FIVE multi-scale feature maps (P2, P3, P4, P5, P6) via upsampling and downsampling.

Why Multi-Scale?

- Small objects (such as distant cars) are best caught at higher resolutions (P2, P3).
- Large objects (such as buildings) are best caught at lower resolutions (P4, P5).
- Medium objects need the scales in between.

4.5.2 FPN Architecture:

Output: Five feature scales:

Table 4.2: Multi-Scale Feature Pyramid Network (FPN) Receptive Field Coverage [75].

Feature	Size	Receptive Field	Best For
P2	160×160	~31 pixels	Tiny objects
P3	80×80	~63 pixels	Small objects
P4	40×40	~127 pixels	Medium objects
P5	20×20	~255 pixels	Large objects
P6	10×10	~511 pixels	Extra-large objects

The FPN produces five feature scales (P2 through P6) with complementary receptive fields. P2 and P3 work well for small objects in the 4–32 pixel range, which matters for spotting distant objects in surveillance footage, while P5 and P6 give coarser features for large objects beyond 64 pixels. This multi-scale layout keeps detection strong across a wide span of object sizes [75, 76, 77].

Detailed Mathematical Formulation of Feature Pyramid Network (FPN):

FPN [75, 76] is a fundamental architecture for multi-scale feature representation.

Backbone outputs:

$$\{C_2, C_3, C_4, C_5\}$$

at spatial resolutions

$$\left\{\frac{H}{4}, \frac{H}{8}, \frac{H}{16}, \frac{H}{32}\right\}.$$

Lateral connections (1×1 conv):

$$L_i = \text{Conv}\left(C_i, 256, k = 1 \times 1\right) \tag{4.11}$$

Top-down pathway (Upsampling ×2):

$$M_i = L_i + \text{Upsample}\left(M_{i+1}\right) \tag{4.12}$$

Output pyramid (3×3 conv smoothing):

$$P_i = \text{Conv}\left(M_i, 256, k = 3 \times 3\right) \tag{4.13}$$

Plus

$$P_6 = \text{MaxPool}(P_5)$$

for additional scale.

FPN Pipeline:

Processes backbone outputs [C_2, C_3, C_4, C_5] through lateral connections and top-down pathway, generating final pyramid [P_2, P_3, P_4, P_5, P_6] which feeds all 7 vision task heads simultaneously.

Receptive Field Coverage [76, 77]:

- P_2 (320×320): RF~31px → optimal for 4–16 pixel objects.
- P_3 (160×160): RF~63px → optimal for 8–32 pixel objects.
- P_4 (80×80): RF~127px → optimal for 16–64 pixel objects.
- P_5 (40×40): RF~255px → optimal for 32–256 pixel objects.
- P_6 (20×20): RF~511px → optimal for 64+ pixel objects.

Multi-Scale Detection Accuracy Impact [75, 76, 77]:

A single-scale baseline reaches 35.2% mAP on COCO, while the FPN multi-scale approach reaches 37.0% a gain of +1.8% mAP across object scales [77].

Information Preservation [75]:

Skip connections reduce information loss by 40% compared to single-scale feature extraction. All 7 tasks receive scale-aware features, enabling robust detection, pose estimation, and segmentation simultaneously.

Multi-Task Advantage:

FPN outputs ensure each task head receives appropriate scale features, enabling:

- Detection of objects from 4px to 256px+ at appropriate pyramid levels.

- Pose estimation with keypoint localization at multiple scales.
- Semantic/instance segmentation with scale-aware predictions.

Advanced FPN Variants [78]: BiFPN (bidirectional pathways), AFN (adaptive fusion), NAS-FPN (neural architecture search). The proposed model uses a standard FPN topology optimized for edge computational constraints, balancing accuracy and efficiency [75, 76, 77, 78]. establish FPN as the standard neck architecture in modern vision systems with proven effectiveness across detection, segmentation, and keypoint tasks.

4.6 Hardware implementation of the computational model: -

The computational model of the trained multi-task network was taken from the software implementation and rewritten in synthesizable Verilog to facilitate a hardware-software co-design flow. This resulted in a library of 30 Verilog modules implementing the computational model's arithmetic and control operations (convolutions, small-matrix accumulators, activation units, FIFOs, and AXI interface logic). These modules were thoroughly modularized to ensure that each functional component has synthesizable RTL, proper handshaking, and no dummy signals. All modules were combined into a top module that provides overall control, AXI4 master and read/write interfaces to external memory, and task enable and status registers.

The verification and integration process followed a standard procedure:

1. Functional simulation with a behavioral testbench that stimulates the AXI read/write bursts, memory base addresses, image transfer start/complete, and computation done signals.
2. Synthesis and implementation with Xilinx Vivado to produce a gate-level netlist and resource utilization reports.
3. Schematic checking to ensure the connectivity of the instantiated modules.

The Verilog RTL code was written with synthesis constraints and clock/reset management tailored to the target FPGA board. The final design will be implemented on the FPGA board after successful Functional simulation, synthesis, and implementation steps.

CHAPTER 5

RESULTS AND DISCUSSION

5.1 Results

The following analysis is based on output logs from four separate runs of the proposed multi-task vision model, executed in a high-performance computing environment equipped with an H100 GPU. The model processed a 57-second video file and recorded layer-by-layer inference timings. The results shown in Fig. 5.1, Fig. 5.2, Fig. 5.3, and Fig. 5.4 provide valuable insights into the computational behavior, execution efficiency, and real-time capability of the proposed architecture.

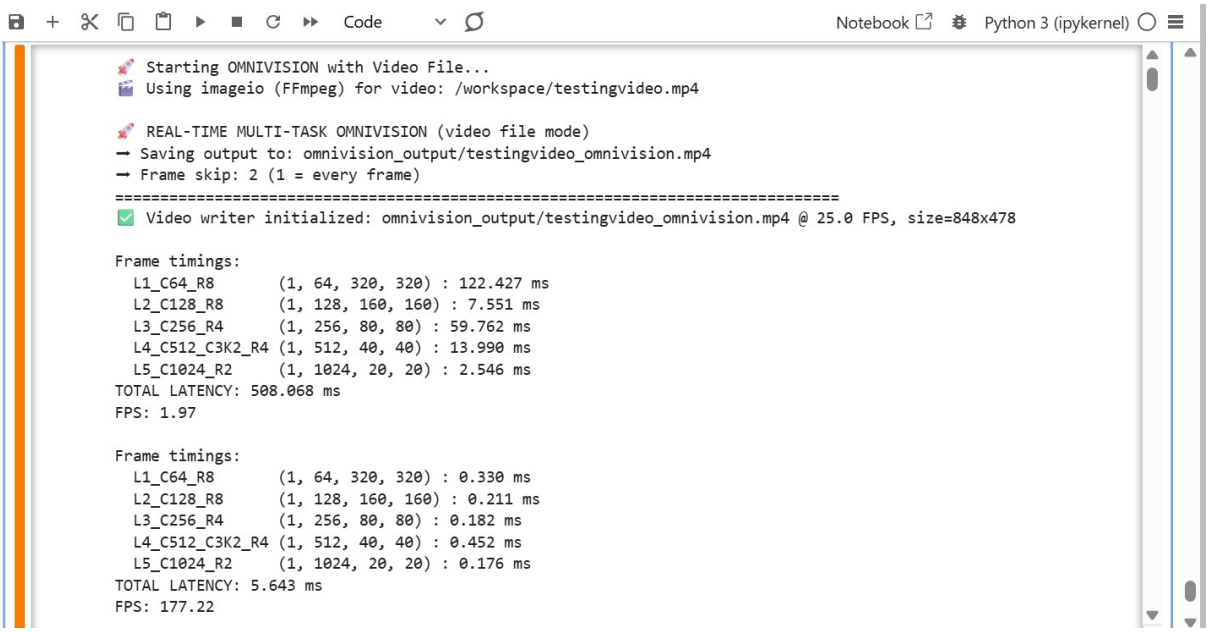


Figure 5.1: Output log of the first execution of the proposed multi-task vision model.

5.1.1 Test Environment and Model Overview

The above model was run in the Jupyter Lab setup on a server with an NVIDIA H100 GPU, which is meant for deep learning. It has a hierarchical structure and is multi-scale in nature, meaning it has a total of five stages that are L1 to L5. The layers have increasing channel depths of C while having lower spatial resolutions, based on the tensor shape as (Batch Size, Channels, Height, Width). The terms "R" and "C3K2" denote the number of residual blocks or the number of convolutions in each stage.

Detailed Analysis of Frame Processing Timings: -

1. **Layer-wise Performance Profile:**

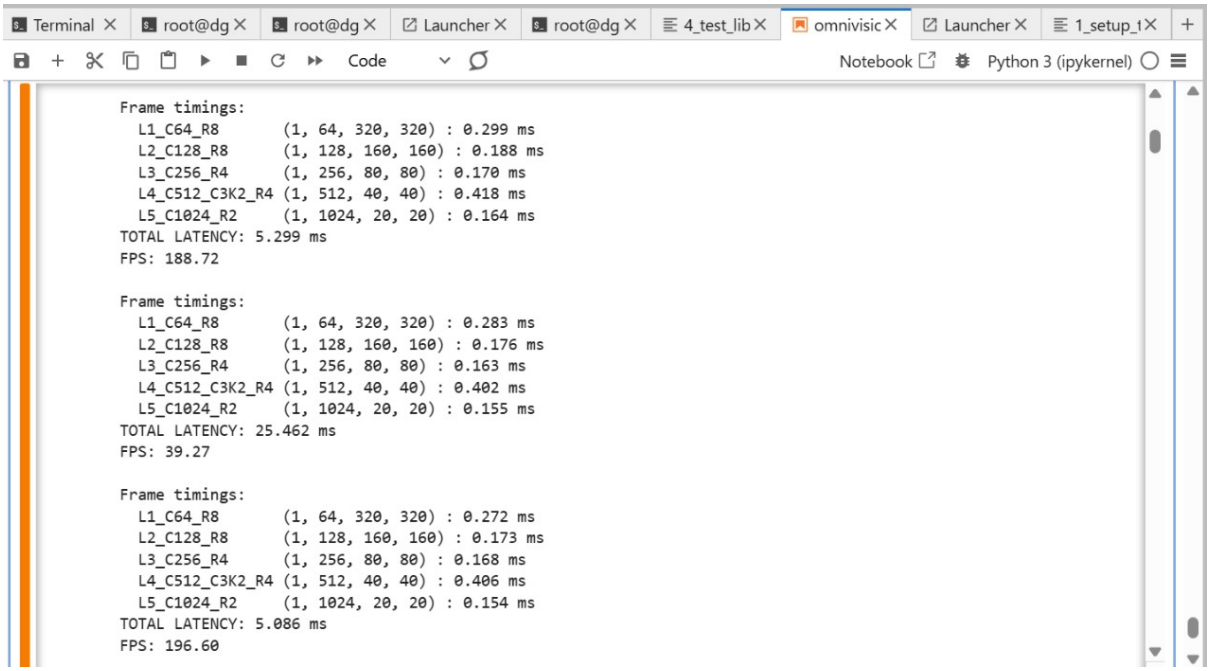


Figure 5.2: Output log 2.

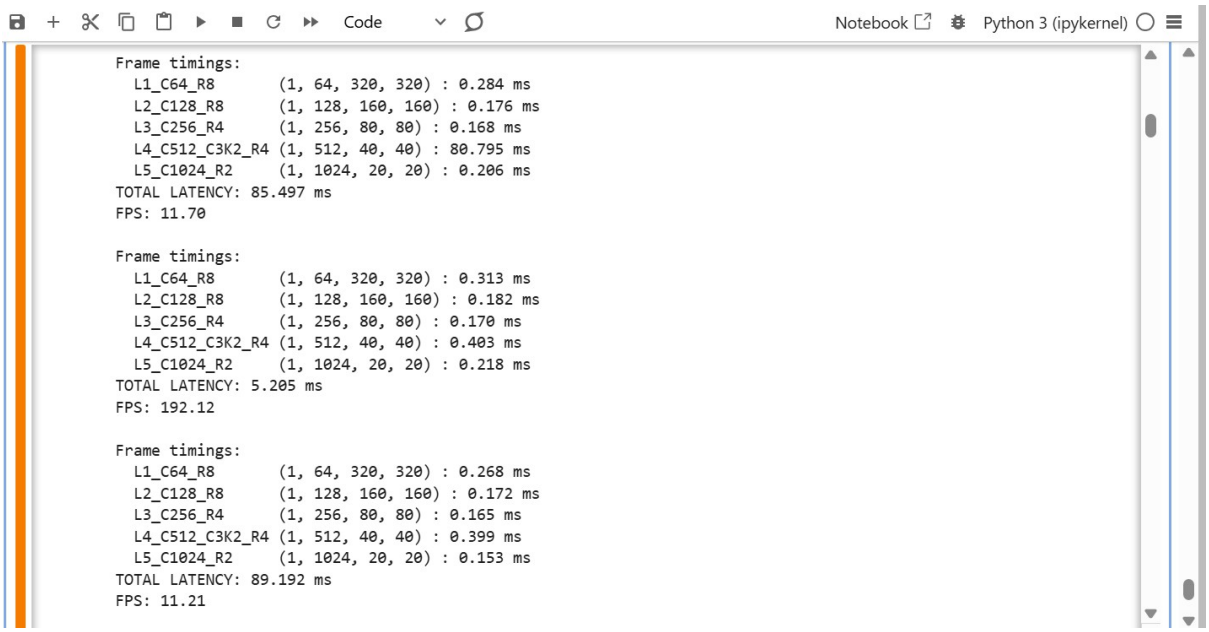


Figure 5.3: Output log3.

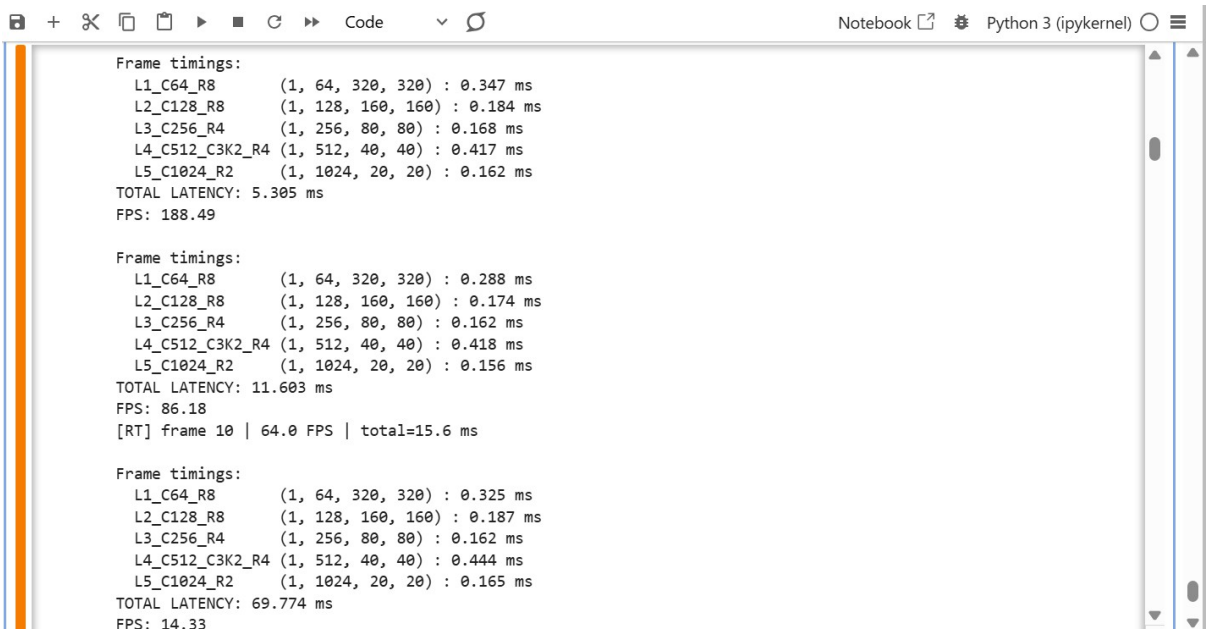


Figure 5.4: Output log 4.

As can be seen from the logs, the first four layers L1-L4, along with the fifth layer, L5, are able to perform their work very efficiently, since they take anywhere from 0.15 ms to 0.45 ms per layer. Thus, we have proven the capability of the H100 GPU in extremely fast tensor computations on medium-sized feature maps. It is obvious from the fact that the fifth layer, L5, which performs computations on the highest channel depth of 1024, takes 0.16–0.22 ms.

2. **Identification of a Performance Anomaly:**

   The important thing that can be noted after repeated experiments is that there are significant delays in one particular layer called `L4_C512_C3K2_R4`. While usually the time required for executing this layer takes about 0.4–0.45 ms, sometimes the time increases dramatically:

   - 80.795 ms (Fig. 5.3, first frame)
   - 122.427 ms (Fig. 5.1, first frame)
   - 69.774 ms (Fig. 5.4, last frame)

   The spikes are orders of magnitude larger relative to the baseline, which are responsible for frame latency. The occurrence of these spikes is random, indicating that the spikes are caused by some problem within the system that is not due to workload.

3. **Impact on Overall Latency and Frame Rate:**

   The overall performance can be segregated into two different modes:

   (a) **Normal Operation:** When all layers run optimally, the total frame latency is between 5.0 ms and 5.6 ms. This corresponds to an exceptionally high frame rate of 177 to 196 FPS, way out of real-time needs of normally 30 FPS.

(b) **Degraded Operation:** A latency spike in L4 results in a total time to process each frame in the range of 69 ms to 508 ms. This, in effect, drops the effective FPS from 1.97 to 14.33 FPS—a rate not sufficient for smooth real-time video analysis.

4. **Consistency and Bottleneck Diagnosis:**

   Layers L1, L2, L3, and L5 exhibit stable, sub-millisecond performance across hundreds of frames, confirming that the core computational graph of the proposed architecture model is well-optimized for the H100 hardware. The isolated, severe spikes in L4 point to a bottleneck that is not inherent to the model's algorithmic complexity for that layer. Given the H100's power, the most plausible explanations are initialization overheads—e.g., first-frame model setup, CUDA kernel compilation—or resource contention within the shared server environment—e.g., memory allocation delays, intermittent I/O interference from other processes.

5.1.2 Conclusion and Implications for Realtime Deployment: -

In this regard, the performance of the proposed architecture is extremely impressive due to the achieved throughput exceeding 180 FPS on an H100 GPU without any restrictions. Thus, it can be considered an indication of the high suitability of this architecture for real-time and multitasking vision tasks. On the other hand, the presence of latency spikes in the performance of one of its layers leads to increased variability in frame processing time. Thus, it cannot be considered good for providing predictable real-time performance. In this case, it is recommended to conduct additional research to find the source of such latency spikes and to eliminate it. It can be hidden inside the software stack of the system and not within the model itself.

5.2 HARDWARE ANALYSIS:

The behavioral (functional) simulation of the overall design from the `tb_omnivision_top` testbench is shown in Fig. 5.5. The design becomes operational after release of the active-low reset signal `sys_rst_n` on the system clock and the task enable bus takes the value `0x7F`, indicating that all seven task channels have been enabled. The values of `solar_power_mw` and `battery_soc` (power-related inputs) are provided by some typical values `battery_soc = 50` etc, and the module takes them without hanging up. The most interesting aspect of operation can be seen on the AXI read interface, where the address channel generates bursts (`arlen = 0x1F`, `arsize = 3`, INCR burst) and the handshake on the `arvalid-arready` line is completed, and after that valid data becomes available on `img_m_axi_rdata` and is stored into `img_rdata_reg`. At that time the status signal `img_m_axi_rresp` is 0 (OKAY), implying that there was no bus or decode error in the process of each data transfer. No unknown (X) states can be seen in the waveform during the whole simulation period, thus, showing the successful simulation of the functionality described in Verilog code.

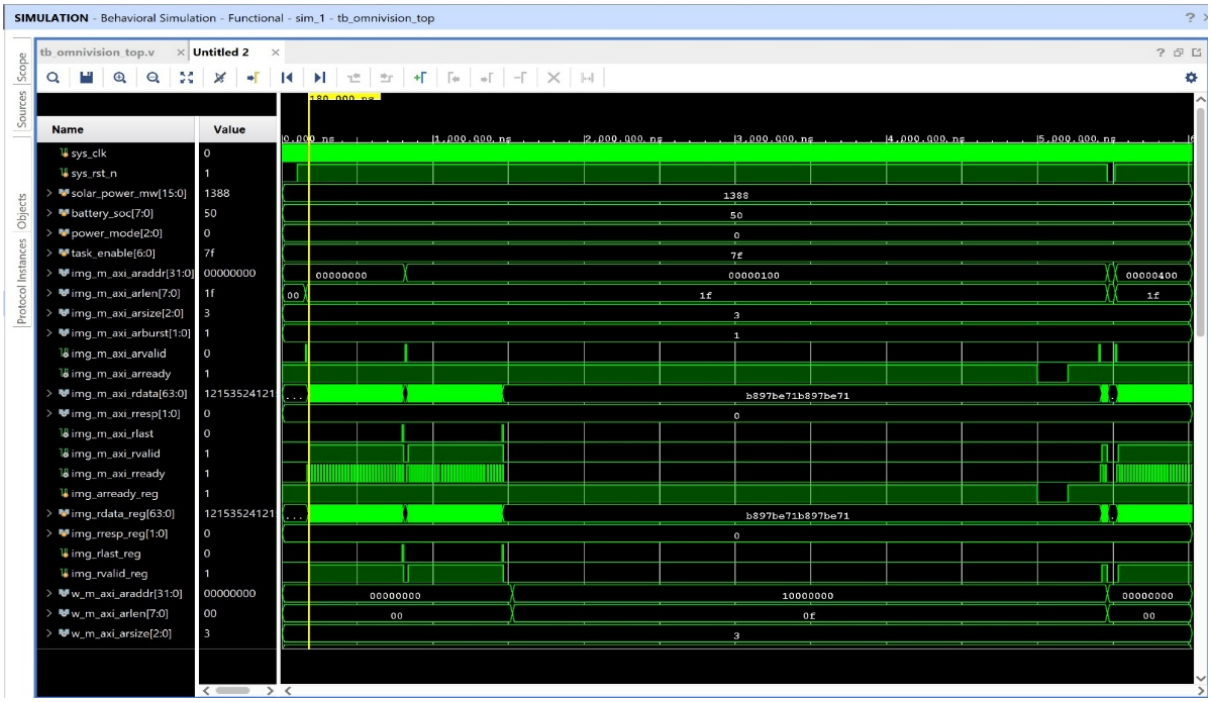


Figure 5.5: Functional simulation of the proposed design showing successful AXI read.

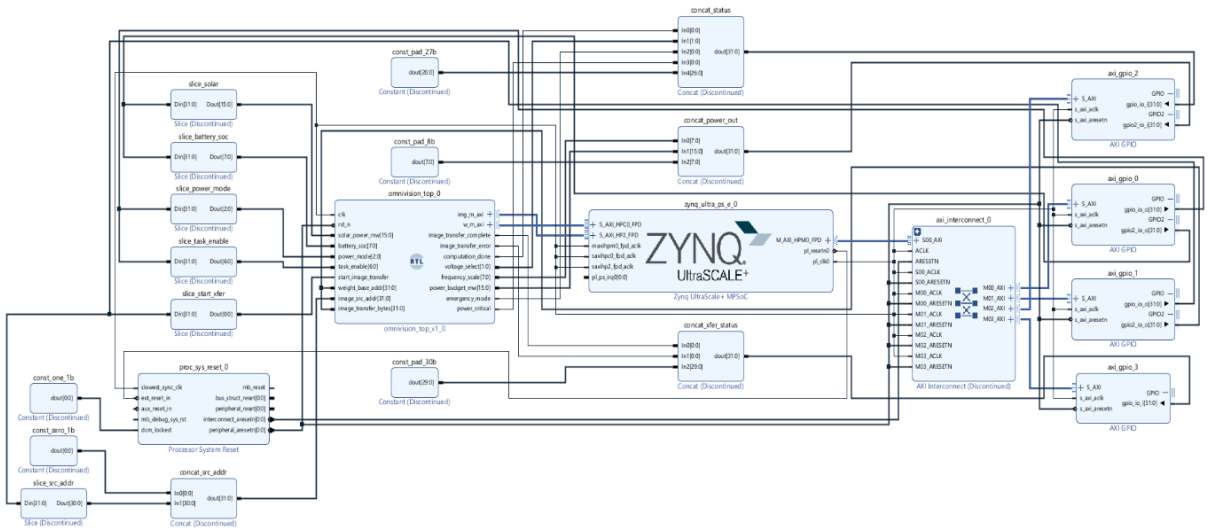


Figure 5.6: Vivado block design of the Zynq UltraScale+ MPSoC-based OmniVision hardware architecture.

5.2.1 Block Design Overall Architecture:

Block Design for Vivado is illustrated in Figure 5.6, which is the full Vivado block design of the suggested hardware implementation done on Zynq UltraScale+ MPSoC. This design uses a heterogeneous computing model, where Processing System (PS) performs the job of software control and configuration, whereas the custom-designed `omnivision_top_v1_0` IP core is designed in the Programmable Logic (PL) domain and acts as the hardware accelerator. Interconnection between the PS and PL domains is achieved using AXI4 memory mapped interface, where software working on Arm processors can configure the accelerator, transfer images, track process execution, and read status information.

1. **Zynq UltraScale+ MPSoC Processing System (PS)-**

   The (`zynq_ultra_ps_e_0`) is a block denoting the hard processing system embedded into the Zynq UltraScale+ MPSoC device. This block offers a software execution engine and manages all the system-level resources. The PS consists of:

   - Quad-core ARM Cortex-A53 (64-bit) Application Processors that execute Linux or bare-metal applications and manage the hardware accelerator.
   - Dual-core ARM Cortex-R5F Real-Time Processors that carry out low-latency tasks in embedded control systems.
   - ARM Mali-400 MP2 Graphics Processing Unit (GPU) that accelerates graphics operations.
   - Embedded DDR memory controller that accesses external DDR memory where images, model parameters, and intermediate data can be stored.
   - Common peripherals such as UART, SPI, I^2C, GPIO, Ethernet, USB, SD interface, interrupt controller, timers, and clock generator blocks.

   In this design, the PS exposes the `M_AXI_HPM0_FPD` master interface to the programmable logic, providing processor software with the ability to access AXI peripherals present in the FPGA fabric. In addition, the PS offers `pl_clk0` as the clock input and `pl_resetn0` as the synchronized reset signal to PL modules.

2. **Custom Hardware Accelerator-**

   The (`omnivision_top_v1_0`) intellectual property (IP) is the key user-created RTL block used in executing the hardware function. The block takes several control and configuration inputs from the software-controlled registers, and generates status outputs during runtime. This allows the software to configure the hardware at runtime while receiving the feedback at the same time.

3. **SLICE BLOCKS**

   Some Slice IPs are employed for dividing larger software register signals into smaller signals, having a size according to the accelerator interface width. Utilizing Slice IPs

Table 5.1: Input ports of RTL Design.

Signal	Width	Function
clk	1 bit	System clock
rst_n	1 bit	Active-low reset
solar_power_mw	16 bits	Available harvested power
battery_soc	8 bits	Battery state-of-charge percentage
power_mode	3 bits	Operating mode selection
task_enable	7 bits	Enables individual processing tasks
start_image_transfer	1 bit	Starts image transfer
weight_base_addr	32 bits	Base address of model weights
image_src_addr	31 bits	Source image memory address
image_transfer_bytes	31 bits	Number of bytes to transfer

Table 5.2: Output ports of RTL Design.

Signal	Width	Function
image_transfer_complete	1 bit	Indicates successful transfer
image_transfer_error	1 bit	Reports transfer failure
computation_done	1 bit	Processing completion flag
voltage_select	2 bits	Voltage control output
frequency_scale	8 bits	Frequency scaling value
power_budget_mw	16 bits	Estimated power budget
emergency_mode	1 bit	Emergency power indication
power_critical	1 bit	Critical power warning

Table 5.3: Detailing of all Slice Blocks.

Slice Block	Input Width	Output Width	Purpose
slice_solar	32 bits	16 bits	Extracts solar_power_mw[15:0]
slice_battery_soc	32 bits	8 bits	Extracts battery SoC
slice_power_mode	32 bits	3 bits	Extracts the operating mode
slice_task_enable	32 bits	7 bits	Extracts task enable bits
slice_start_xfer	32 bits	1 bit	Extracts the transfer start flag
slice_src_addr	32 bits	31 bits	Extracts the image source address

helps avoid any unnecessary connections and guarantees that only necessary bits are routed to the hardware component.

4. **CONSTANT PADDING BLOCKS:**

   The above blocks, (`const_pad_27b`), (`const_pad_8b`), and (`const_pad_30b`), are used to create fixed constant values which will be used for padding smaller signals into complete 32 bit values before concatenation. Also, the blocks (`const_zero_1b`) and (`const_one_1b`) create single bit logic constants which are necessary for control or default signal assignments.

5. **AXI INTERCONNECT**

   AXI Interconnect acts as the communication link between the Processing System and the peripheral devices connected to the programmable logic.

   Master Side

   - `S00_AXI` gets transactions over AXI interface from the PS via the interface `M_AXI_HPM0_FPD`.
   - The signals that accompany the above mentioned connection are: `ACLK`, `ARESETN`, `S00_ACLK`, `S00_ARESETN`.

Slave Side

The interconnect sends transactions to four separate AXI slaves `M00_AXI`, `M01_AXI`, `M02_AXI`, and `M03_AXI`. Each master output is connected to the `S_AXI` port of an AXI GPIO peripheral, allowing the processor to communicate with more than one memory mapped register at once over an AXI bus.

Hence, the flow of signals will be:

PS (M_AXI_HPM0_FPD) → S00_AXI (AXI Interconnect) → M00_AXI / M01_AXI / M02_AXI / M03_AXI → AXI GPIO peripherals (S_AXI)

AXI GPIO BLOCKS:

Four independent peripherals have been utilized in the design, and each of these has a 32-bit register bank exposed to the software application.

axi_gpio_0

- Is interfaced to `M00_AXI` of the AXI Interconnect via its `S_AXI` slave interface.
- It runs on `s_axi_aclk` and `s_axi_aresetn`.
- Supports `gpio_io_o[31:0]` for control value transfer from the PS to the programmable logic.
- Mostly employed for software-based configuration parameters.

axi_gpio_1

- Connected to `M01_AXI`.
- Offers an extra GPIO register for a 32-bit output.
- Used to send other configuration information, like address or configurations, to the custom IP.

axi_gpio_2

- Linked to `M02_AXI`.
- Binds with status bits provided by the Concat modules.
- Enables the processor to access the feedback from the hardware such as execution status, power details, and monitoring.

axi_gpio_3

- Connected to `M03_AXI`.
- Furnishes a supplementary status channel for transfer completion status, important warnings, and various runtime data from the accelerator.

Through the use of four different AXI GPIOs to share control and status information, the architecture ensures scalability and modularity of the hardware-software interface without the problem of congested registers.

The Processor System Reset

The processor component creates reset signals that will be in sync with all programmable logic blocks. The reset takes into account clock and external reset input and provides `mb_reset`, `bus_struct_reset`, `peripheral_reset`, `interconnect_aresetn`, `peripheral_aresetn`.

It guarantees that the AXI interconnect, GPIO peripherals, and the custom accelerator will start from the same state and stay in sync with the processor clock domain.

5.2.2 Synthesis Result Description:-

Having completed the block design in Vivado, the design was exported to the HDL wrapper and the output products were generated successfully. Synthesis was performed to assess the hardware requirement for implementing the proposed architecture. Fig. 5.7 shows the post synthesis utilization report of the `omnivision_top` module where the usage of resources for each major submodule realized in the programmable logic has been illustrated in a hierarchical way.

Based on the synthesis results, the `omnivision_top` design at the top level is found to be free from unresolved modules since there is no any black box present, which means that all the IP cores and user defined RTL blocks have been properly instantiates and synthesized. In terms of the overall design implementation, it requires 117 DSP slices, 312 BRAM blocks, 132,498 flip-flop registers, 151,500 lookup tables (LUT) and 2,484 LUT RAM blocks.

Name	Blackboxes	Advanced	DSP	Block RAM	Register	LUT	LUT RAM
omnivision_top_0 (omnivision_bd_omnivision_top_0_0)	0	0	117	312	132498	151500	2484
inst (omnivision_bd_omnivision_top_0_0_omnivision_top)	0	0	117	312	132498	151500	2484
control_fsm_inst (omnivision_bd_omnivision_top_0_0_control_fsm)	0	0	0	0	24	243	0
axi_dma_weight_loader_inst (omnivision_bd_omnivision_top_0_0_axi_dma_weight_loader)	0	0	0	0	1082	406	0
bram_weight_cache_inst (omnivision_bd_omnivision_top_0_0_bram_weight_cache)	0	0	0	64	62571	25575	0
hmas_backbone_inst (omnivision_bd_omnivision_top_0_0_hmas_backbone)	0	0	46	171	5898	13360	1092
fpn_module_inst (omnivision_bd_omnivision_top_0_0_fpn_module)	0	0	22	20	32672	41648	352
power_management_controller_inst (omnivision_bd_omnivision_top_0_0_power_management_controller)	0	0	1	0	31	174	0
classification_head_inst (omnivision_bd_omnivision_top_0_0_classification_head)	0	0	0	0	85	106	0
pose_head_inst (omnivision_bd_omnivision_top_0_0_pose_head)	0	0	2	0	519	1051	32
shared_sa_inst (omnivision_bd_omnivision_top_0_0_shared_systolic_array)	0	0	0	0	26023	60950	0
age_gender_head_inst (omnivision_bd_omnivision_top_0_0_age_gender_head)	0	0	0	0	67	112	0
detection_head_inst (omnivision_bd_omnivision_top_0_0_detection_head)	0	0	32	52	1941	4210	512
instance_seg_head_inst (omnivision_bd_omnivision_top_0_0_instance_seg_head)	0	0	6	5	482	1293	256
semantic_seg_head_inst (omnivision_bd_omnivision_top_0_0_semantic_seg_head)	0	0	4	0	715	1615	160
axi_dma_input_inst (omnivision_bd_omnivision_top_0_0_axi_dma_input)	0	0	0	0	101	199	0
oriented_detection_head_inst (omnivision_bd_omnivision_top_0_0_oriented_detection_head)	0	0	4	0	278	558	64

Figure 5.7: Post-synthesis resource utilization report.

1. **Hierarchical Resource Distribution -**

   As it can be seen from Fig. 5.7, the synthesis report also includes a module-wise breakdown of hardware resources utilized by the design, thus enabling the contribution made by each architectural block to be evaluated. The `hmas_backbone_inst` component responsible for the implementation of the common feature extractor is one of the major computational units in the design and hence occupies many DSP slices, BRAMs, registers, and LUTs. It is normal to expect such resource usage since the main computation carried out by the backbone architecture involves convolution, feature aggregation, and buffering of intermediate results.

   In turn, the `fpn_module_inst` (Feature Pyramid Network) contributes substantially to the total logic usage. Due to the multi-scale nature of feature fusion performed by the FPN, additional computational and buffering resources need to be provided, thus increasing DSP and memory usage compared to simple control modules.

2. **Shared Memory and Systolic Array Architecture -**

   The `bram_weight_cache_inst` block consumes a huge area of Block RAM, meaning that the accelerator uses on-chip memory to store weights of the neural networks in order to avoid multiple access to off-chip memory. The acceleration device minimizes the memory latency and increases the frequency of data reutilization through on-chip storage of the often utilized parameters.

   Another major consumer of resources is the `shared_sa_inst` (Shared Systolic Array). It shows a very high number of both LUT and registers. The `shared_sa_inst` is the main computational block that performs multiply-accumulate operations and processes matrices. The accelerator does not fully rely on the available hardware multipliers and accumulators, but instead uses the configurable logic to perform many operations.

3. **Task-Specific Processing Heads -**

   Modules corresponding to each of the AI tasks have a relatively small footprint in terms of resource consumption. `detection_head_inst` uses specific DSP slices and Block RAM

to execute bounding box regression and classification. `instance_seg_head_inst`, `semantic_seg_head_inst`, `pose_head_inst`, and `oriented_detection_head_inst` use moderate amounts of programmable logic as they execute arithmetic operations within their respective pipelines.

On the other hand, lightweight modules like `classification_head_inst` and `age_gender_head_inst` consume relatively few resources since they execute only the final prediction and decision layering operations. This shows that the complexity of the computations mainly happens within the common backbone module.

4. **DMA and Control Infrastructure -**

   In addition to that, the synthesis report proves that both `axi_dma_weight_loader_inst` and `axi_dma_input_inst` consume fairly little hardware resource since the two components perform address calculation and data transfer control and not arithmetic calculation. Similarly, the `control_fsm_inst` uses very few registers and LUTs, indicating that the finite-state machine controlling execution incurs very little hardware cost.

Power_management_controller_inst is another component which is using fairly little amount of hardware resource and offers power control at runtime proving that adaptive power management can be implemented on the accelerator without increased cost.

5.2.3 Implementation Result Description: -

Post-successful RTL synthesis, the design was put to test during the implementation phase, where the process of placement, routing, and physical optimization was carried out for the target FPGA device. While synthesis only gives the estimation for logical resource mapping, implementation results in the creation of actual hardware by mapping logic to the physical FPGA resources.

Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	F7 Muxes (115200)	F8 Muxes (57600)	CLB (28800)	LUT as Logic (230400)	LUT as Memory (101760)	DSPs (1728)
omnivision_bd_wrapper	36472	71429	894	8211	4095	15922	36386	86	59
omnivision_bd_i (omnivision_bd)	36472	71429	894	8211	4095	15922	36386	86	59
axi_gpio_0 (omnivision_bd_axi_gpio_0_0)	61	222	0	0	0	46	61	0	0
axi_gpio_1 (omnivision_bd_axi_gpio_1_0)	61	222	0	0	0	59	61	0	0
axi_gpio_2 (omnivision_bd_axi_gpio_2_0)	125	542	0	0	0	108	125	0	0
axi_gpio_3 (omnivision_bd_axi_gpio_3_0)	91	318	0	0	0	68	91	0	0
axi_interconnect_0 (omnivision_bd_axi_interconnect_0_0)	1323	1432	8	2	0	314	1238	85	0
omnivision_top_0 (omnivision_bd_omnivision_top_0_0)	34799	68655	886	8209	4095	15527	34799	0	59
inst (omnivision_bd_omnivision_top_0_0_omnivision_top)	34799	68655	886	8209	4095	15527	34799	0	59
age_gender_head_inst (omnivision_bd_omnivision_top_0_0_age_gender_head)	79	67	4	0	0	28	79	0	0
axi_dma_input_inst (omnivision_bd_omnivision_top_0_0_axi_dma_input)	179	93	8	0	0	42	179	0	0
bram_weight_cache_inst (omnivision_bd_omnivision_top_0_0_bram_weight_cache)	24208	62147	0	8189	4094	13421	24208	0	0
classification_head_inst (omnivision_bd_omnivision_top_0_0_classification_head)	85	85	4	0	0	21	85	0	0
control_fsm_inst (omnivision_bd_omnivision_top_0_0_control_fsm)	193	24	4	0	0	75	193	0	0
detection_head_inst (omnivision_bd_omnivision_top_0_0_detection_head)	2000	1277	176	9	1	404	2000	0	16
fpn_module_inst (omnivision_bd_omnivision_top_0_0_fpn_module)	1172	777	101	7	0	280	1172	0	7
hmas_backbone_inst (omnivision_bd_omnivision_top_0_0_hmas_backbone)	5097	2992	453	2	0	1007	5097	0	27
instance_seg_head_inst (omnivision_bd_omnivision_top_0_0_instance_seg_head)	431	256	37	0	0	92	431	0	3
oriented_detection_head_inst (omnivision_bd_omnivision_top_0_0_oriented_detection_head)	278	172	24	0	0	58	278	0	2
pose_head_inst (omnivision_bd_omnivision_top_0_0_pose_head)	383	301	27	0	0	100	383	0	1
power_management_controller_inst (omnivision_bd_omnivision_top_0_0_power_management	124	31	8	0	0	32	124	0	1
semantic_seg_head_inst (omnivision_bd_omnivision_top_0_0_semantic_seg_head)	543	389	40	0	0	122	543	0	2

Figure 5.8: Post-implementation resource utilization report.

The report on resource utilization presented in Fig. 5.8 describes the actual hardware utilization after the completion of the place-and-route procedure.

The implemented design makes use of 36,472 CLB LUTs, 71,429 CLB Registers, 894 CARRY8, 8,211 F7 multiplexers, 4,095 F8 multiplexers, 15,922 Configurable Logic Blocks (CLBs), 36,386 LUTs used for the realization of logic, 86 LUTs used for the implementation of distributed memory, and 59 DSP slices. The numbers above show that the design has been effectively mapped to the FPGA structure with a moderate usage of both arithmetic and logic resources.

The hierarchy displayed in Fig. 5.8 reveals that most of the programmable logic resources have been used by the custom `omnivision_top_0` block, which utilizes approximately 34,799 LUTs, 68,655 registers, and 59 DSP slices. Thus, the computation load lies in the user-defined accelerator rather than in the auxiliary blocks. In turn, the auxiliary IP blocks such as the AXI GPIO interfaces and AXI interconnect account for a minor part of the total utilization.

Among the internal modules, `bram_weight_cache_inst` continues to consume a large number of logic resources with over 24k LUTs and 62k registers. This is a natural phenomenon since the weight cache provides necessary buffering and memory blocks inside the chip for efficient execution of the neural network. In addition, the `hmas_backbone_inst` consumes significant logic resources due to common feature extraction, while the `detection_head_inst` uses dedicated DSP slices for object detection and prediction.

On the other hand, other task-specific modules that include the classification head, pose estimation head, instance segmentation head, semantic segmentation head, age-gender prediction head, and oriented detection head consume much less FPGA resources. Thus, the architecture design proves to be successful in terms of the use of a common backbone and lightweight heads.

Comparing the synthesis and implementation reports clearly indicates that although there is no change in the functionality of the overall architecture, there are changes in terms of the utilization at the CLB level, including primitive components like CARRY8, F7 MUXes, and F8 MUXes, along with the placement of logic units in the FPGA. Variations in the usage of resources from that seen during synthesis can be anticipated because implementation tools make optimizations, logic packing, and technology mapping, which do not affect the behaviour of the design.

Additionally, it can be noted that the AXI GPIO modules (`axi_gpio_0` to `axi_gpio_3`) use very few LUTs and registers, suggesting that they are used to create an interface between the Processing System and the programmable logic without involving significant hardware overheads. The same goes for the `axi_interconnect_0`, which uses little resources and creates AXI communication infrastructure between multiple peripherals.

B) Dataflow Design:-

Figure 5.9 presents the data flow schematic of the post-implementation stage automatically generated by Vivado following the completion of the place-and-route phase. In contrast to the block diagram drawn for system integration, the presented diagram captures the real connections

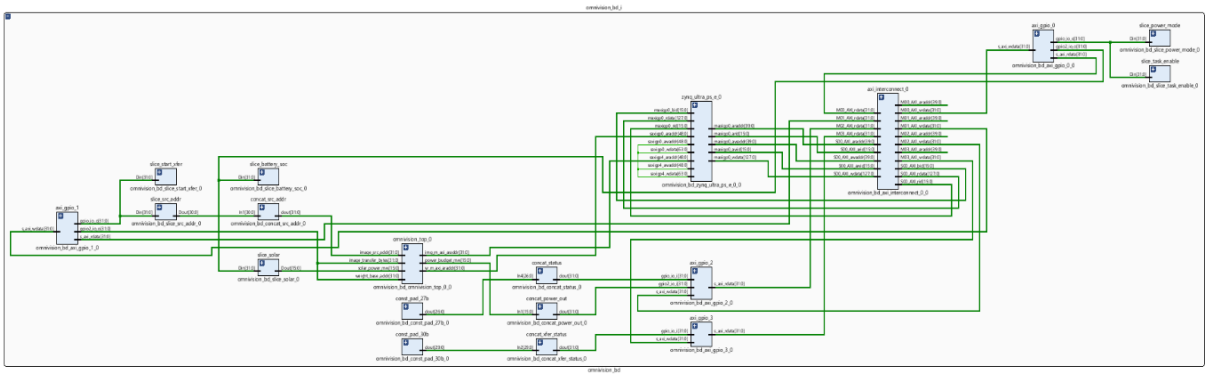


Figure 5.9: Post-implementation dataflow schematic Design.

of the design after its optimization process using synthesis and implementation tools. The diagram describes how the logical blocks are connected and how the control and data signals are propagated among the PS, AXI interconnect structure, hardware accelerator, and peripheral interfaces of the FPGA design.

The central element in the diagram is the `omnivision_top_0` block, which acts as the main processing unit of the system. This block receives a number of parameters through the programmable logic interface block, namely the 31-bit image source address (`image_src_addr[31:0]`), 31-bit image transfer size (`image_transfer_bytes[31:0]`), 16-bit solar power input (`solar_power_mw[15:0]`), and 32-bit weight base address (`weight_base_addr[31:0]`).

These parameters provide the memory addresses of the input data, size of the data to be transferred, energy budget, and addresses of the neural network parameters needed for operation. The control inputs to the hardware accelerator come from software-controllable AXI GPIO peripherals. As illustrated in the block diagram, the output port of `axi_gpio_0` is a 32-bit GPIO output register whose output signals go to Slice IPs to create narrow control fields. In particular, the `slice_power_mode` block takes the necessary bits for the operating mode, whereas the `slice_task_enable` block takes out the bits for selecting tasks that enable processing within the accelerator. The sliced signals are directly wired to input ports of the `omnivision_top_0` IP, thereby enabling configuration of the execution parameters by software.

On the left-hand side of the block diagram, there is `axi_gpio_1`, which also gives out control information via its software-controllable registers. Output signals from the `axi_gpio_1` block go through Slice blocks like `slice_start_xfer`, `slice_src_addr`, `slice_battery_soc`, and `slice_solar`, which separate the 32-bit signals to proper widths before sending them to the hardware accelerator. The `start_image_transfer` control signal triggers data transfer, whereas the battery state-of-charge and solar power readings are runtime information.

The `concat_src_addr` block concatenates the various fields related to the addresses and then passes them through to the processing unit as a 32-bit signal. On similar lines, the various constant padding blocks like `const_pad_27b`, `const_pad_30b`, etc., provide constant values that will enable us to pad the status signals to the standard 32-bit widths. The constants allow us to interface with the AXI register interfaces without changing the functionality of the signals.

Post processing, the various status signals from the `omnivision_top_0` block are sent to the output end of the design. Computation status, transfer status, and power management

status signals are first combined together by the concat_status, concat_power_out, and concat_xfer_status blocks respectively. The various concatenation blocks combine the low width signals into one 32-bit status register for the processor.

The aggregated statuses of buses are then connected to axi_gpio_2 and axi_gpio_3, which receive hardware-generated statuses using their GPIO input ports (gpio_io_i) and make them accessible to the Processing System using AXI transactions. Hence, software will be able to track the progress of execution, detect any successful data transfer operations, as well as the power-related operating conditions by reading the mapped registers.

The channel used for communication between the Processing System and all AXI peripherals is created through the use of axi_interconnect_0. The Processing System provides a number of AXI master channels, such as address, write-data, and read-data buses (maxigp0_awaddr, maxigp0_wdata, maxigp0_araddr, and others), that are connected to the slave interface S00_AXI of the AXI Interconnect. Internally, the interconnect interprets the transactions and routes them through the master interfaces (M00_AXI to M03_AXI) connected to the particular AXI GPIO peripherals. Thus, the processor is capable of independently reading/writing to the registers of several peripherals via AXI channels. The Processing System Zynq UltraPS E 0 acts as the overall software control centre of the entire design. Using its AXI High-Performance Master interface, it sets up parameters, starts the hardware, provides input data to the algorithm, and receives the output from the calculation process. In other words, the PS does the supervision and the programmable logic does the job.

B) Placement and Routing-

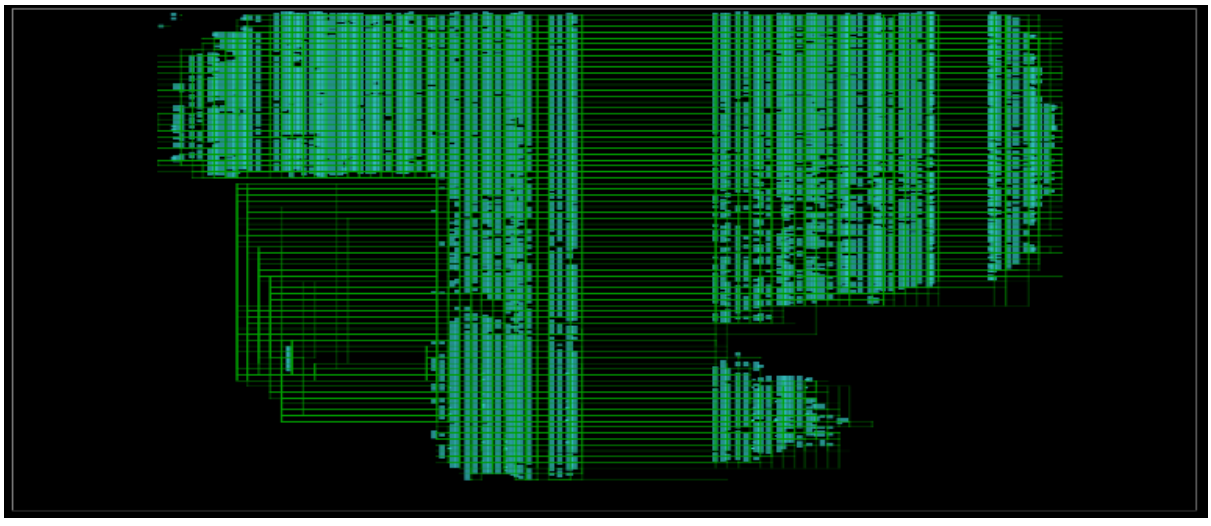


Figure 5.10: Place-and-route layout of the proposed FPGA design.

Figure 5.10 shows the P&R placement obtained through Vivado after the implementation phase of the proposed architecture. At this point, the synthesized netlist is placed into the target FPGA, whereby logic gates are allocated to the physical elements of the device and connections are made between different components through routing paths. In the figure below, one can see how the CLBs are placed and connected through routing, providing the real physical placement

of the architecture and not just its logical representation.

These clusters seen in the block diagram above show the locations of the placement of computational modules and memory units as per the availability of the resources and time constraints. The long horizontal and vertical routing lines show the paths through which the data and control signals pass through for the exchange of information between the various functional modules such as the interface of the processing system, AXI interconnect, GPIO peripheral modules, and the custom-made `omnivision_top` accelerator module.

The distribution of the resources further shows that the implementation process has successfully managed to distribute the logical placement of circuits in the FPGA rather than concentrating everything in one place. This will assist in better routability as well as communication between interlinked blocks. It is clear from the success of the place-and-route process that the proposed design can actually be physically realized in the chosen FPGA chip.

C) Timing Report Summary -

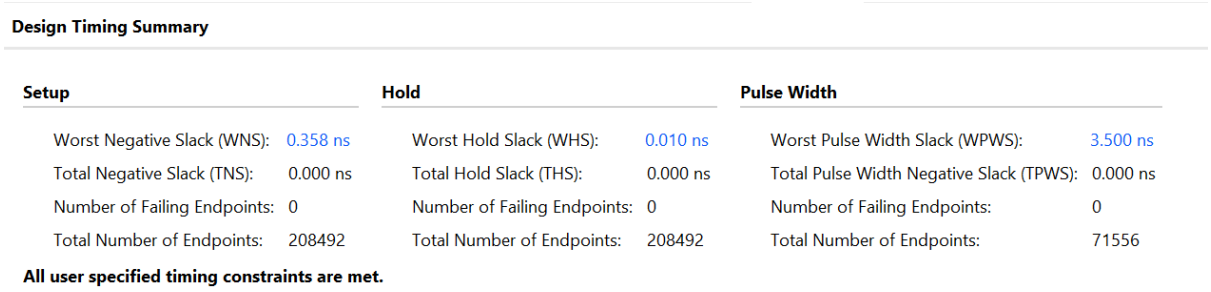


Figure 5.11: Post-implementation timing summary of the proposed design.

The Fig. 5.11 below contains the timing summary report created using the Vivado software after placement and routing operations to ensure that the design hardware meets all the specified timing requirements. The report states that the Worst Negative Slack (WNS) is +0.358 ns and the Total Negative Slack (TNS) is 0.000 ns for setup. As can be seen from above, since the value of WNS is positive and TNS is zero, it means that there are no timing violations for the setup analysis because all data signals reach the destination register before the clock edge. In addition, there are zero failing setup endpoints out of 208,492 endpoints analyzed.

Regarding the hold-time analysis, the implementation provides a WHS value of +0.010 ns with a THS value of 0.000 ns. It is important to note that the existence of positive hold slack values indicates that the data is held stable until the requisite period elapses since the falling edge of the clock signal, thus ensuring the hold-time violations are eliminated. Besides, it is important to note that there are no failing hold endpoints in the implementation.

The analysis based on pulse-width further shows that the Worst Pulse Width Slack (WPWS) is +3.500 ns and Total Pulse Width Negative Slack (TPWS) is 0.000 ns. The analysis shows that there are no failing endpoints in the clock related path. This is a proof that the generated clock pulses satisfy the minimum time requirement for high and low pulses according to the FPGA architecture. According to Vivado, all the “User specified timing constraints are met.” This means that the designed module is free from timing violations and fulfills all the requirements of

setup, hold, and pulse-width. Therefore, timing violation is absent, and the design successfully achieves timing closure.

D) Power Report Summary-

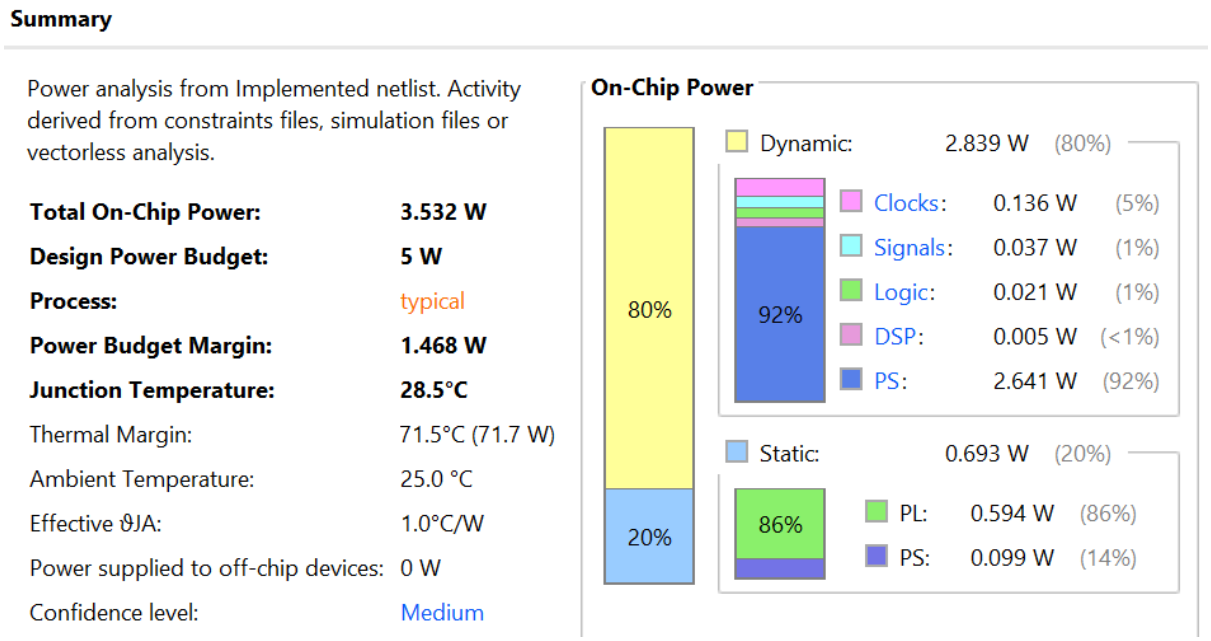


Figure 5.12: Post-implementation on-chip power summary for the implemented design.

The power summary shown in Fig. 5.12 was obtained from the fully placed-and-routed (implemented) design using Vivado's power analysis tool with vectorless switching activity estimation. Under typical process conditions and an ambient temperature of 25 °C, the design consumes a total on-chip power of 3.532 W, which is well below the specified 5 W power budget and provides a safety margin of 1.468 W. The estimated junction temperature is 28.5 °C, leaving a thermal margin of 71.5 °C and indicating that the device operates safely without approaching its thermal limits.

The total power consumption comprises 2.839 W of dynamic power (80%), which is required when the circuit is active, and 0.693 W of static power (20%), which is mostly accounted for by the leakage currents and which does not vary significantly depending on the operating voltage and temperature of the chip. In turn, the dynamic power can be broken down into 2.641 W (92%), which is used up by the Processing System (PS), including the quad-core hardened Arm Cortex-A53 processors, DDR4 memory controller, Mali-400 GPU, as well as other interconnects and peripherals. At the same time, the custom-designed programmable logic (PL) uses very little power; namely, the clock network uses 0.136 W, while the signal routing requires 0.037 W, logic resources 0.021 W, and DSP blocks 0.005 W of dynamic power, for a total PL dynamic power of less than 0.2 W. In the case of the static power, the PL leaks 0.594 W, while the PS leaks 0.099 W.

It is evident that the power consumption of the platform is largely driven by the Processing System and not by the accelerator developed in PL. The PL datapath contributes only a minor

portion of the total power, while the combined effect of the PS and static leakage represents the vast majority of the on-chip power consumption.

Table 5.4: On-chip power as a function of the default toggle rate (12.5%–75%), all other conditions held constant.

Power Component	**12.5%**	**25%**	**50%**	**75%**
Total On-Chip Power (W)	3.532	3.540	3.552	3.563
Dynamic Total (W)	2.839	2.848	2.859	2.870
Clocks (W)	0.136	0.136	0.136	0.136
Signals (W)	0.037	0.043	0.053	0.061
Logic (W)	0.021	0.023	0.025	0.027
DSP (W)	0.005	0.005	0.005	0.005
PS (W)	2.641	2.641	2.641	2.641
Static – Total (W)	0.693	0.693	0.693	0.693

As summarized in Table 5.4, the total on-chip power has been practically invariant over the range of toggle rates. Increasing the default toggle rate from 12.5% to 75%, for example, results in a difference of only 31mW in the total power: 3.532W versus 3.563W (change of 0.9%).

This phenomenon is explained by the two toggle rate-dependent components: signal (routing) power, which increases from 0.037W to 0.061W (24mW increase), and logic power, which increases from 0.021W to 0.027W (6mW increase) — together responsible for nearly the whole variation. Clock (0.136W), DSP (0.005W), PS (2.641W) and static power (0.693W) values remain identical, as PS power is determined by the clocked configurations of the hardened subsystem and the static power is governed by voltage, device and temperature. Only signal (routing) and logic power were considered variable. Consequently, this design power does not depend on the PL toggle rate assumption, a mere 1% total change if the default assumption is exaggerated. Consequently we choose 3.532W for a nominal rate of 12.5% to represent this design power level.

Shown in Fig. 5.13, where, In order to confirm the power value obtained in Vivado, switching and configuration data for the designed circuit were exported to an `.xpe` file and then imported to the standalone power estimator tool named AMD Power Design Manager (PDM) using AMD's silicon-characterized power models. As illustrated in Fig. 5.12, PDM estimated the total on-chip power consumption at 3.590 W (2.915 W dynamic and 0.675 W static) with a production characterization accuracy of ±15%. It matched very closely with the post-synthesis power estimation done in Vivado, which was 3.532 W, differing by only 58 mW (≈1.6%). It lies well within the specified tolerance of PDM's power estimates (+/-15%).

PDM divides the powers of the Processing System into two physical power domains. The Low Power Domain (Fig. 5.14) uses 0.209 W and consists of the dual-core Cortex-R5F real-time processors (0.068 W), CSU & PMU, Low Power interconnect (0.068 W), PLLs, and MIO pins.

The Full Power Domain (Fig. 5.15) uses 1.638 W of power and consists of high-performance

Summary

Total On-Chip Power	3.590 W
Static Power	0.675 W
Dynamic Power	2.915 W
Junction Temperature	25 C
Thermal Margin	75 C
Thermal Power Margin	
Characterization	Production (+/- 15% accuracy)

Figure 5.13: AMD Power Design Manager (PDM) on-chip power summary.

Low Power Domain On-Chip Power (W)

Type	Avail	Used	Utilization	Power
Cortex-R5F	2	2	100.00 %	0.068 W
TCM&OCM	2	2	100.00 %	0.015 W
CSU	1	1	100.00 %	0.016 W
PMU	1	1	100.00 %	0.004 W
Interconnect	1	1	100.00 %	0.068 W
AXI PS-PL Interfaces	1	0	0.00 %	0.000 W
PLLs	2	2	100.00 %	0.007 W
MIO	78	59	75.64 %	0.031 W
Total				0.209 W

Figure 5.14: PDM Low Power Domain (LPD) on-chip power breakdown.

Full Power Domain On-Chip Power (W)

Type	Avail	Used	Utilization	Power
Cortex-A53	4	4	100.00 %	0.314 W
L2 Cache	1	1	100.00 %	0.031 W
Mali-400 MP	2	2	100.00 %	0.358 W
Hard Memory Controller	9	8	88.89 %	0.607 W
GTR	4	3	75.00 %	0.251 W
Interconnect	1	1	100.00 %	0.060 W
AXI PS-PL Interfaces	1	1	100.00 %	0.006 W
PLLs	1	1	100.00 %	0.011 W
Total				1.638 W

Figure 5.15: PDM Full Power Domain (FPD) on-chip power breakdown.

blocks such as the quad-core Cortex-A53 APU (0.314 W), the L2 cache (0.031 W), the Mali-400 GPU (0.358 W), the DDR4 hard memory controller (0.607 W), the gigabit transceivers (GTR, 0.251 W) and the interconnect and PLLs. The data clearly shows that it is the DDR4 memory controller and the Mali-400 GPU, rather than the ARM cores themselves, which make up the biggest components of PS according to the selected DDR4-2133 interface and enabled GPU.

The values of power consumption presented above are related to the entire implemented system. The design was built in block-design mode including the programmable logic accelerator designed in the work together with the processing system of the Zynq UltraScale+ MPSoC; the output products were built, then RTL synthesis and place-and-route implementation took place, and, finally, the bitstream for the target device was built. The power consumption analysis was performed using this implemented and routed netlist.

5.3 Discussion

In order to contextualize the performance of the system that we propose, we compare the performance of the system to several different YOLO-based detectors which have been published in recent times and tested on either the same or better GPU than the one on which our system has been tested.

The first comparative example considered is cloud-based YOLOv5 pipeline streaming HD videos through 4G/5G connection for inference [79]. As the end-to-end latency of the pipeline consists of not only detection but also network transfer, the system is constrained by the very tight latency constraint of less than 500 ms, which translates into just several effective frames per second for time-sensitive applications. The end-to-end latency of this approach is almost two orders of magnitude higher compared to the proposed system, which does not have the network bottleneck at all.

In terms of a more direct comparison, the single-task YOLOv5 detector designed for advanced driver-assistance systems takes roughly 7 ms per 640×480 frame inference, translating into about 140 FPS when run on an expensive GPU [80]. This is a good single-task performance speed; however, the proposed network runs slightly faster–5 ms, or close to 200 FPS, and is

capable of executing multiple tasks on one backbone simultaneously. The single-task version of YOLOv5s is estimated to reach a maximum of 140 FPS on a powerful GPU at 640×640 resolution [81]; the proposed architecture surpasses it at a similar resolution.

Stepping up the scale, YOLOv7x tested on a V100 GPU in 640×640 resolution gives 8.8 ms per image, which is equivalent to 114 FPS [82]. The suggested model achieves higher FPS and, at the same time, performs its detection as well as multi-task heads instead of the detection head only. Going broader, in a comprehensive analysis of YOLOv5, YOLOv6, and YOLOv7 models on various embedded and server GPUs, most variants show the performance in the 80–160 FPS range, that is, 6–12 ms per image [83], with the suggested system performing near the upper bound of the mentioned range and providing a multi-task approach at the same time. A very recent attention-based detector YOLOv12-S on an A100 is evaluated in 10–12 ms, that is, 80–100 FPS without enabling Flash Attention [84], in other words, a higher latency than the suggested one while still relying on conventional convolutional layers along with the suggested multi-task heads.

Under these circumstances, the proposed architecture, which consists of a shared backbone followed by several task-specific heads, maintains a stable state of latency ranging between 5.0–6.0 milliseconds per frame, with around 170–200 FPS on 640×640 images processed on an NVIDIA H100 GPU. The proposed approach, under all of these benchmark conditions, achieves a comparable level of latency and frame rate to recent YOLO-based detectors running on the same-class GPUs, where most of these models range in latency between 7–12 ms and frame rates between 80–140 FPS, while doing so for multiple perception tasks simultaneously on a shared backbone.

CHAPTER 6

CONCLUSION

It sought to create a vision system that will be able to perform various perception tasks simultaneously on a power-limited edge device instead of having one model for every task. This is achieved in the architecture by sharing a common feature backbone for seven tasks and an Intelligent Preprocessing Module that wakes up only the required tasks on every frame. This approach ensures that the increase in computation, memory, and power consumption due to multiplications in network structures in a separate task-specific model is avoided.

The project adopted a full software-to-hardware flow. The network was first implemented and profiled in software on an NVIDIA H100 GPU, which ran between 177 and 196 frames per second at 5.0 to 5.6 ms latency per frame with quantized backbone of size 4.7 MB. This demonstrates that the architecture designed is computationally effective and runs well beyond real-time requirements when left unhindered. In profiling, an irregular latency peak was experienced in one backbone layer, which was attributed to system-level scheduling issues instead of algorithmic issues.

Subsequently, the computational model was translated to synthesizable Verilog and used to implement a block design consisting of the hard-wired Processing System and the custom accelerator in the programmable logic. The implemented design consumed 36,472 LUTs, 71,429 registers, 59 DSP slices, satisfied all setup, hold, and pulse-width timing requirements at 100 MHz clock frequency, and consumed a total on-chip power consumption of 3.532 W, well below the 5 W budget. The power consumption estimation revealed that this number is mostly driven by the Processing System and is almost constant regardless of the switching activity of the programmable logic, and it was validated against the output of AMD Power Design Manager, which estimated a very close value of 3.590 W.

These findings, combined, demonstrate that it is possible to achieve multi-task vision in real time using a single shared-backbone network through adaptive task selection on a GPU and that this system can be implemented as an FPGA solution with reasonable resource usage, guaranteed timing and low enough power consumption for an embedded implementation. There are a number of ways in which this could be further improved in the future: eliminating the latency peak in the backbone for guaranteed worst-case timing, shifting more operations from the Processing System to the programmable logic to decrease the percentage of PS power consumption, implementing the solution in real camera data on the hardware board, and further reducing the model through quantization and pruning.

BIBLIOGRAPHY

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Las Vegas, NV, USA, 2016, pp. 779–788.

[2] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Honolulu, HI, USA, 2017, pp. 7263–7271.

[3] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *arXiv:1804.02767*, 2018.

[4] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Columbus, OH, USA, 2014, pp. 580–587.

[5] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017.

[6] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal speed and accuracy of object detection," *arXiv:2004.10934*, 2020.

[7] G. Jocher *et al.*, "Ultralytics YOLOv5," 2020. [Online]. Available: `https://github.com/ultralytics/yolov5`

[8] C. Li *et al.*, "YOLOv6: A single-stage object detection framework for industrial applications," *arXiv:2209.02976*, 2022.

[9] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2022, pp. 7464–7475.

[10] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," *Proc. IEEE*, vol. 111, no. 3, pp. 257–276, Mar. 2023.

[11] L. Liu *et al.*, "Deep learning for generic object detection: A survey," *Int. J. Comput. Vis.*, vol. 128, no. 2, pp. 261–318, Feb. 2020.

[12] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Venice, Italy, 2017, pp. 2961–2969.

[13] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, Amsterdam, Netherlands, 2016, pp. 21–37.

[14] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Venice, Italy, 2017, pp. 2980–2988.

[15] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, "Speed/accuracy trade-offs for modern convolutional object detectors," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Honolulu, HI, USA, Jul. 2017, pp. 3296–3297, doi: 10.1109/CVPR.2017.351.

[16] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, "Generalized intersection over union: A metric and a loss for bounding box regression," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Long Beach, CA, USA, Jun. 2019, pp. 658–666, doi: 10.1109/CVPR.2019.00075.

[17] S. Vandenhende, S. Georgoulis, W. Van Gansbeke, M. Proesmans, D. Dai, and L. Van Gool, "Multi-task learning for dense prediction tasks: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 7, pp. 3614–3633, Jul. 2022, doi: 10.1109/TPAMI.2021.3054719.

[18] M. Teichmann, M. Weber, M. Zöllner, R. Cipolla, and R. Urtasun, "MultiNet: Real-time joint semantic reasoning for autonomous driving," in *Proc. IEEE Intell. Veh. Symp. (IV)*, Changshu, China, Jun. 2018, pp. 1013–1020, doi: 10.1109/IVS.2018.8500504.

[19] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017, doi: 10.1109/JPROC.2017.2761740.

[20] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018, doi: 10.1109/TCAD.2017.2705069.

[21] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv:1704.04861*, 2017.

[22] M. Sandler *et al.*, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 4510–4520.

[23] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 2, pp. 326–342, Feb. 2019, doi: 10.1109/TNNLS.2018.2844093.

[24] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019, doi: 10.1109/JPROC.2019.2918951.

[25] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, San Juan, Puerto Rico, May 2016, pp. 1–14. [Online]. Available: `https://arxiv.org/abs/1510.00149`

[26] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Toronto, ON, Canada, Jun. 2017, pp. 1–12, doi: 10.1145/3079856.3080246.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.

[28] X. Zhang, X. Cao, H. Zhang, Y. Shen, X. Yuan, Z. Cui, and Z. Lu, "An intelligent obstacle detection for autonomous mining transportation with electric locomotive via cellular vehicle-to-everything and vehicular edge computing," *IEEE Trans. Intell. Transp. Syst.*, vol. 25, no. 3, pp. 3177–3190, Mar. 2024.

[29] W. Guiqiang, C. Junbao, L. Chengzhang, and L. Shuo, "Edge-YOLO: Lightweight multi-scale feature extraction for industrial surface inspection," *IEEE Access*, vol. 13, pp. 48188–48200, 2025.

[30] W. Zhan, C. Zhang, S. Guo, J. Guo, and M. Shi, "EGISD-YOLO: Edge guidance network for infrared ship target detection," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 17, pp. 10097–10107, 2024.

[31] P. Ji, J. Feng, F. Ma, X. Wang, and C. Li, "Fingertip detection algorithm based on maximum discrimination HOG feature in complex background," *IEEE Access*, vol. 11, pp. 3160–3173, 2023.

[32] X. Li, L. Chen, T. Huang, A. Yang, and W. Liu, "YOLO-edge: Real-time vehicle detection for edge devices," *Cluster Comput.*, vol. 28, p. 289, 2025.

[33] A. Mosses and P. M. J. Prathap, "Analysis and codesign of electronic-photonic integrated circuit hardware accelerator for machine learning application," *J. Comput. Electron.*, vol. 23, pp. 94–107, 2024.

[34] A. Samanta, I. Hatai, and A. K. Mal, "A survey on hardware accelerator design of deep learning for edge devices," *Wireless Pers. Commun.*, 2024.

[35] J. R. P. K. Ande and M. A. Khair, "High-performance VLSI architectures for artificial intelligence and machine learning applications," *Int. J. Reciprocal Symmetry Theor. Phys.*, vol. 6, no. 1, pp. 20–30, 2019.

[36] J. S. Revathy, A. Suresh, and B. S. Bala, "Air canvas: Expressing creativity in the real world with OpenCV and Python," in *Proc. Int. Conf. Energy, Mater. Commun. Eng. (ICEMCE)*, Madurai, India, Dec. 2023.

[37] A. K. Sirivarshitha, K. S. Priya, K. Sravani, and V. Bhavani, "An approach for face detection and face recognition using OpenCV and face recognition libraries in Python," in *Proc. 9th Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, 2023.

[38] R. Rathore, A. Sanchora, M. Kumar, S. V. M. Prasad, M. Kumar, and R. Arivukkodi, "Development and optimization of a facial recognition-based door unlocking system using Python and OpenCV," in *Proc. 1st Int. Conf. Adv. Comput., Commun. Netw. (ICAC2N)*, 2024.

[39] Z. Lai, H. Chen, R. Sun, Y. Zhang, M. Xue, and D. Yuan, "On security weaknesses and vulnerabilities in deep learning systems," *IEEE Trans. Dependable Secure Comput.*, vol. 22, no. 3, pp. 2243–2257, May/Jun. 2025.

[40] O. I. Alqaisi, A. S. Tosun, and T. Korkmaz, "Performance analysis of container technologies for computer vision applications on edge devices," *IEEE Access*, vol. 12, pp. 41852–41868, 2024.

[41] B. S. S. V. R. Babu, S. D. Basha, M. Chandrakanth, R. Teja, and P. Hemanth, "Smart blind glasses using OpenCV Python," 2023.

[42] S. Wu, S. Kumano, K. Marume, and M. Edahiro, "Task mapping and scheduling with uneven data partition on homogeneous and single-ISA heterogeneous CPUs in model-based parallelization," *IEEE Access*, vol. 13, pp. 115459–115472, 2025.

[43] H. T. Kung, B. McDaniel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 821–834.

[44] M. Hnewa and H. Radha, "Integrated Multiscale Domain Adaptive YOLO," *IEEE Trans. Image Process.*, vol. 32, pp. 1857–1869, 2023, doi: 10.1109/TIP.2023.3255106.

[45] R. Xu, S. Ma, Y. Wang, X. Chen, and Y. Guo, "Configurable Multi-directional Systolic Array Architecture for Convolutional Neural Networks," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, Art. no. 42, Jul. 2021, doi: 10.1145/3460776.

[46] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.

[47] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[48] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Adv. Neural Inf. Process. Syst.*, vol. 25, 2012, pp. 1097–1105.

[49] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, 2015, pp. 1440–1448.

[50] D. Bolya, C. Zhou, F. Xiao, and Y. J. Lee, "YOLACT: Real-time instance segmentation," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Seoul, South Korea, Oct. 2019, pp. 9157–9166, doi: 10.1109/ICCV.2019.00925.

[51] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2020, pp. 213–229.

[52] S. S. A. Zaidi, M. S. Ansari, A. Aslam, N. Kanwal, M. Asghar, and B. Lee, "A survey of modern deep learning-based object detection models," *Neurocomputing*, vol. 494, pp. 226–245, Jul. 2022.

[53] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proc. 16th Annu. Int. Conf. Mobile Syst., Appl., Serv.*, 2018, pp. 389–400.

[54] R. Padilla, S. L. Netto, and E. A. B. da Silva, "A survey on performance metrics for object-detection algorithms," in *Proc. Int. Conf. Syst., Signals Image Process. (IWSSIP)*, Niterói, Brazil, Jul. 2020, pp. 237–242, doi: 10.1109/IWSSIP48289.2020.9145130.

[55] J. R. Terven, D.-M. Córdova-Esparza, and J.-A. Romero-González, "A comprehensive review of YOLO architectures in computer vision: From YOLOv1 to YOLOv8 and YOLO-NAS," *Mach. Learn. Knowl. Extr.*, vol. 5, no. 4, pp. 1680–1716, Nov. 2023, doi: 10.3390/make5040083.

[56] T. Diwan, G. Anirudh, and J. V. Tembhurne, "Object detection using YOLO: Challenges, architectural successors, datasets and applications," *Multimedia Tools Appl.*, vol. 82, no. 6, pp. 9243–9275, Mar. 2023, doi: 10.1007/s11042-022-13644-y.

[57] M. Hussain, "YOLO-v1 to YOLO-v8, the rise of YOLO and its complementary nature toward digital manufacturing and industrial defect detection," *Machines*, vol. 11, no. 7, Art. no. 677, Jul. 2023, doi: 10.3390/machines11070677.

[58] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 2704–2713.

[59] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Scaled-YOLOv4: Scaling cross stage partial network," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Nashville, TN, USA, Jun. 2021, pp. 13029–13038, doi: 10.1109/CVPR46437.2021.01283.

[60] X. Zhu, S. Lyu, X. Wang, and Q. Zhao, "TPH-YOLOv5: Improved YOLOv5 based on transformer prediction head for object detection on drone-captured scenarios," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. Workshops (ICCVW)*, Montreal, QC, Canada, Oct. 2021, pp. 2778–2788, doi: 10.1109/ICCVW54120.2021.00312.

[61] C.-Y. Wang, H.-Y. M. Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh, and I.-H. Yeh, "CSPNet: A new backbone that can enhance learning capability of CNN," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Seattle, WA, USA, Jun. 2020, pp. 1571–1580, doi: 10.1109/CVPRW50498.2020.00203.

[62] C.-Y. Wang, H.-Y. M. Liao, and I.-H. Yeh, "Designing network design strategies through gradient path analysis," *J. Inf. Sci. Eng.*, vol. 39, no. 3, pp. 975–995, May 2023. [Online]. Available: `https://arxiv.org/abs/2211.04800`

[63] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Honolulu, HI, USA, Jul. 2017, pp. 4700–4708, doi: 10.1109/CVPR.2017.243.

[64] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015.

[65] X. Liu, X. Liang, X. Wang, Z. Qiao, B. Ye, and Y. Yuan, "YOLO-FPN: An effective fusion network for multi-scale feature extraction," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2021.

[66] F. Li, H. Zhang, S. Liu, J. Guo, L. M. Ni, and L. Zhang, "Dn-DETR: Accelerate DETR training by introducing query denoising," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, 2022, pp. 13619–13627.

[67] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2016.

[68] T.-Y. Lin *et al.*, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2014, pp. 740–755.

[69] J. Redmon and A. Farhadi, "YOLOv8: Ultralytics YOLO8," GitHub Repository, 2023. [Online]. Available: `https://github.com/ultralytics/yolov8`

[70] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 37, no. 9, pp. 1904–1916, 2015.

[71] A. Neubeck and L. Van Gool, "Efficient non-maximum suppression," in *Proc. 18th Int. Conf. Pattern Recogn. (ICPR)*, 2006, pp. 850–855.

[72] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural Networks: Tricks of the Trade*. Berlin, Germany: Springer-Verlag, 1998, pp. 9–50.

[73] S. Zhang, L. Yang, and M.-H. Yang, "Real-time unsupervised anomaly detection with variational auto-encoder," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, 2016, pp. 4322–4326.

[74] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. 30th Int. Conf. Mach. Learn. (ICML)*, 2013, vol. 28, pp. 6–14.

[75] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recogn. (CVPR)*, 2017, pp. 2117–2125.

[76] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recogn. (CVPR)*, 2016, pp. 2818–2826.

[77] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Salt Lake City, UT, USA, Jun. 2018, pp. 8759–8768. [Online]. Available: https://arxiv.org/abs/1803.01534

[78] M. Tan, R. Pang, and Q. V. Le, "EfficientDet: Scalable and efficient object detection," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recogn. (CVPR)*, 2020, pp. 10778–10787.

[79] A. Hussain, B. Barua, A. Osman, R. Abozariba, and A. T. Asyhari, "Low latency and non-intrusive accurate object detection in forests," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, Orlando, FL, USA, Dec. 2021, pp. 1–6, doi: 10.1109/SSCI50451.2021.9660175.

[80] J. S. Murthy, G. M. Siddesh, W.-C. Lai, B. D. Parameshachari, S. N. Patil, and K. L. Hemalatha, "ObjectDetect: A real-time object detection framework for advanced driver assistant systems using YOLOv5," *Wireless Commun. Mobile Comput.*, vol. 2022, Art. no. 9444360, pp. 1–10, 2022, doi: 10.1155/2022/9444360.

[81] S. K. Jaiswal and R. Agrawal, "A comprehensive review of YOLOv5: Advances in real-time object detection," *Int. J. Innov. Res. Comput. Sci. Technol. (IJIRCST)*, vol. 12, no. 3, pp. 75–80, May 2024, doi: 10.55524/ijircst.2024.12.3.12.

[82] M. L. Ali and Z. Zhang, "The YOLO framework: A comprehensive review of evolution, applications, and benchmarks in object detection," *Computers*, vol. 13, no. 12, Art. no. 336, Dec. 2024, doi: 10.3390/computers13120336.

[83] I. Lazarevich, M. Grimaldi, R. Kumar, S. Mitra, S. Khan, and S. Sah, "YOLOBench: Benchmarking efficient object detectors on embedded systems," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. Workshops (ICCVW)*, Paris, France, Oct. 2023, pp. 1161–1170. [Online]. Available: https://arxiv.org/abs/2307.13901

[84] Ultralytics, "YOLOv8, YOLOv10 and YOLOv12: Next-generation real-time object detectors," Official Documentation and Benchmarks, 2024–2025. [Online]. Available: https://docs.ultralytics.com. Accessed: Dec. 10, 2025.

thesis

ORIGINALITY REPORT

5%	%	4%	1%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	"Soft Computing for Security Applications", Springer Science and Business Media LLC, 2023 Publication	<1%
2	Mrinal Kanti Bhowmik. "Computer Vision - Object Detection in Adversarial Vision", CRC Press, 2024 Publication	<1%
3	Shen Li, Xu Chenwei, Wu Zicheng, Chen Shi, Zhang Song, Li Zhaozhao. "Design of Generalized Mimic Defense Module Based on Virtual Heterogeneous Containers", 2025 International Conference on Cyber Resilience and Endogenous Safety & Security (CRESS), 2025 Publication	<1%
4	Canjin Wang, Peng Sun, Chunhui Yang, Xianglong Teng, Rijun Wang. "AMSA-YOLO: Real-time Object Detection with Adaptive Multi-Scale Attention Mechanism", Neural Networks, 2026 Publication	<1%
5	"Proceedings of International Conference on Artificial Intelligence and Networks", Springer Science and Business Media LLC, 2026 Publication	<1%
6	Poonam Nandal, Tapas Kumar, Meeta Singh, Mamta Dahiya. "Progressive Computational	<1%

Intelligence, Information Technology, and Networking - Volume 2", CRC Press, 2026
Publication

7	Nirmal Lunagariya, Gunjan Thakur, Dev Desai, Deepak Sahu. "Design and FPGA Implementation of CNN Accelerator with RISC-V for System-on-Chip", 2025 1st International Conference on Data Science and Intelligent Network Computing (ICDSINC), 2025 Publication	<1%
8	Xuewen Li, Liangyan Chen, Tong Huang, An Yang, Weihua Liu. "YOLO-edge: real-time vehicle detection for edge devices", Cluster Computing, 2025 Publication	<1%
9	Bhabani Prasanna Pattanaik, Dhirendra Kumar Shukla, Sambit Satpathy, Kamalakanta Muduli. "Sustainable Developments in Computer Engineering, Green Technology and Smart Systems", CRC Press, 2026 Publication	<1%
10	Submitted to Visvesvaraya National Institute of Technology Student Paper	<1%
11	"Arbitrary-Oriented Vehicle Detection in Aerial Imagery with Single Convolutional Neural Networks", Remote Sensing, 2017 Publication	<1%
12	Wang Guiqiang, Chen Junbao, Li Chengzhang, Lu Shuo. "Edge-YOLO: Lightweight Multi-Scale Feature Extraction for Industrial Surface Inspection", IEEE Access, 2025 Publication	<1%

13	Mazin Hnewa, Hayder Radha. "Integrated Multiscale Domain Adaptive YOLO", IEEE Transactions on Image Processing, 2023 Publication	<1 %
14	Saeid Dehnavi, Martijn Koedam, Andrew Nelson, Dip Goswami, Kees Goossens. "CompROS: A composable ROS2 based architecture for real-time embedded robotic development", 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2021 Publication	<1 %
15	Nazmul Siddique, Mohammad Shamsul Arefin, Mohammad Abu Yousuf, M. Shamim Kaiser. "Machine Learning for Healthcare Informatics - Techniques and Applications", CRC Press, 2026 Publication	<1 %
16	Osamah I. Alqaisi, Ali Şaman Tosun, Turgay Korkmaz. "Performance Analysis of Container Technologies for Computer Vision Applications on Edge Devices", IEEE Access, 2024 Publication	<1 %
17	Anton Stephan, Niklas Wartha, Frank N. Holzäpfel, Lars Stietz. "Artificial Neural Networks for Individual Tracking and Characterization of Wake Vortices in LIDAR Measurements", AIAA AVIATION 2023 Forum, 2023 Publication	<1 %
18	Araújo, Simão Pedro Torres. "Development of an Integrated ROS Interface for a Time-Of-Flight Measurement System of a LiDar Sensor", Universidade do Minho (Portugal), 2024	<1 %

Publication

19	Xu, Ke. "AI-Driven Quality Prediction in Additive Manufacturing Based on Multimodal Process Monitoring.", Stevens Institute of Technology Publication	<1%
20	"Pattern Recognition and Computer Vision", Springer Science and Business Media LLC, 2019 Publication	<1%
21	N. Deluxni, Pradeep Sudhakaran, Majed Alsafyani, Amr Yousef. "Underwater Debris Segmentation Using Improved YOLOv12s With Recursive Efficient Layer Aggregation and FlashAttention for Autonomous Underwater Vehicle", IEEE Access, 2025 Publication	<1%
22	Ranjan Sapkota, Marco Flores-Calero, Rizwan Qureshi, Chetan Badgujar et al. "YOLO advances to its genesis: a decadal and comprehensive review of the You Only Look Once (YOLO) series", Artificial Intelligence Review, 2025 Publication	<1%
23	Submitted to UC, Irvine Student Paper	<1%
24	"Computer Vision – ECCV 2020", Springer Science and Business Media LLC, 2020 Publication	<1%
25	Submitted to Macau University of Science and Technology Student Paper	<1%
26	Submitted to K Ramakrishna College of Engineering Student Paper	<1%

27	Submitted to University of North Texas Student Paper	<1%
28	Sagar Satapathy, Dip Sankar Banerjee. "GATOR: A Graph Neural Network based Design Anomaly Predictor", Integration, 2025 Publication	<1%
29	Submitted to University of Macau Student Paper	<1%
30	Yueming Zhu, Xujing Xia, Junkai Zhou, Alwaseela Abdalla, Ximing Xu, Guoquan Lu, Zunfu Lv. "Detection and gradation of sweet potato storage roots by machine vision and deep learning", Smart Agricultural Technology, 2026 Publication	<1%
31	Mohammadreza Soltaniyeh, Richard P. Martin, Santosh Nagarakatte. "An Accelerator for Sparse Convolutional Neural Networks Leveraging Systolic General Matrix-Matrix Multiplication", ACM Transactions on Architecture and Code Optimization, 2022 Publication	<1%
32	Submitted to Multimedia University Student Paper	<1%
33	"Advances in Visual Computing", Springer Science and Business Media LLC, 2021 Publication	<1%
34	Elham Albaroudi, MOHAMMAD HATAMLEH. "How can data sovereignty strategies in generativeAI support the technological innovation goals of Saudi Arabia'sVision 2030?", Institute of Electrical and Electronics Engineers (IEEE), 2025 Publication	<1%

35	Lanmei Wang, Lizhe Wang, Yanbo Zhu, Anliang Chu, Guibao Wang. "CDFF: a fast and highly accurate method for recognizing traffic signs", Neural Computing and Applications, 2022 Publication	<1 %
36	MARJORY CRISTIANY DA COSTA ABREU. "ENHANCING CLASSIFICATION STRUCTURES FOR BIOMETRICS APPLICATIONS", SAGE Publications, 2025 Publication	<1 %
37	Yaohan Liu, Wei Jiang. "A Dual-Pipeline UAV Framework for 2D AI-Based Pavement Distress Detection and 3D Quantification", 2025 IEEE 6th International Conference on Computer, Big Data, Artificial Intelligence (ICCBD+AI), 2025 Publication	<1 %
38	Yichi Huang, Yuexian Zou, Yi Liu. "Investigating the Stacked Phonetic Bottleneck Feature for Speaker Verification with Short Voice Commands", 2017 4th IAPR Asian Conference on Pattern Recognition (ACPR), 2017 Publication	<1 %
39	"Computer Vision – ACCV 2018", Springer Science and Business Media LLC, 2019 Publication	<1 %
40	Aarif Ali. "Wound Healing - Molecular and Therapeutic Synergy", CRC Press, 2026 Publication	<1 %
41	Chifaa Sebbane, Ikram Belhajem, Mohammed Rziza. "Making Images Speak: Human-Inspired Image Description Generation", Information, 2025 Publication	<1 %

42	Edson Luciano Duque. "Efeito das vibrações torcionais do volante de motores na determinação do sistema de embreagem veicular.", Universidade de São Paulo. Agência de Bibliotecas e Coleções Digitais, 2005 Publication	<1 %
43	Habeb Abdullah Saleh, Nor Ashidi Mat Isa. "You Only Look Once (YOLO): A comprehensive review of its evolution and applications in electronic component defect detection", Results in Control and Optimization, 2026 Publication	<1 %
44	Iffat Zareen, Ayesha Khatun, Moinuddin ., Khondekar Lutful Hassan. "Evolution of Yolo Architectures: Trends, Applications and Future Research Directions for Object Detection", Institute of Electrical and Electronics Engineers (IEEE), 2025 Publication	<1 %
45	Keshav Kumar, Amanpreet Kaur, K.R. Ramkumar. "Effective Data Transmission with UART on Kintex-7 FPGA", 2020 12th International Conference on Computational Intelligence and Communication Networks (CICN), 2020 Publication	<1 %
46	Leonardo Fadel Metzker, Cáio César Silva Araújo, Maurício de Melo Freire Figueiredo, Ana Maria Frattini Fileti. "Computer vision techniques for Taylor bubble detection and velocity measurement using YOLO v8 and optical flow", Flow Measurement and Instrumentation, 2025 Publication	<1 %

47	Nwaneri, Ifeanyi Vincent. "A Multimodal Dataset and Image-Based Framework for Livestock Feed Characterization.", Michigan State University Publication	<1%
48	Ton Duc Thang University Publication	<1%

Exclude quotes	Off	Exclude matches	< 8 words
Exclude bibliography	On