Degree Project in Computer Science and Engineering

Second Cycle, 30 credits

Comparative Analysis of NPU-Accelerated Edge Computing Platforms for Object Detection Model Implementation

ERON ARIODITO HERMANTO

Comparative Analysis of NPU-Accelerated Edge Computing Platforms for Object Detection Model Implementation

Eron Ariodito Hermanto

Master's Programme in ICT Innovation - Autonomus System

Date: 2025-06-12

Supervisors: Mihhail Matskin, Yichen Yang

Examiner: Carlo Fischione

School of Electrical Engineering and Computer Science

Host company: BitSim NOW AB

Swedish title: Jämförande analys av NPU-accelererade Edge Computing-plattformar för implementering av objektdetekteringsmodeller

Abstract

As the demand grows for low-power, low-latency inference in applications such as surveillance, robotics, and embedded vision, selecting the appropriate combination of model and hardware becomes critical. However, few deployment-oriented studies provide a comprehensive evaluation of performance and long-term cost implications for running state-of-the-art object detection models on edge devices. This thesis explores the deployment and optimization of real-time object detection models on NPU-accelerated edge AI hardware platforms, focusing on the NXP i.MX 8M Plus with integrated NPU and Hailo-8L external NPU.

The core objective of this work is to assess the inference performance, optimization effects, and total cost of ownership (TCO) for candidate edge AI platforms running YOLOv8 and YOLOv9 object detection models. The key challenge is to balance detection accuracy, runtime efficiency, and deployment cost within the limitations of edge computing hardware.

To address this, the study benchmarks multiple configurations, including variations in input shape and camera stream resolution. The evaluation considers metrics such as frames per second (FPS), mean Average Precision (mAP), power consumption, and thermal behavior. Additionally, a five-year TCO analysis is conducted for a deployment scenario involving 100 devices, incorporating hardware pricing, expected failure rates, and energy costs.

The findings indicate that Hailo-8L delivers higher inference throughput and better energy efficiency in terms of FPS per watt. However, the NXP platform proves more cost-effective over time due to its lower energy consumption and higher reported reliability. YOLOv8 models show strong post-quantization performance with minimal accuracy loss, making them well-suited for deployment on constrained devices. YOLOv9 models present varied outcomes, with YOLOv9m maintaining high accuracy after quantization, while YOLOv9s suffers significant degradation.

This research provides a practical framework for selecting edge AI solutions based on specific trade-offs between performance, cost, and model robustness. It enables better decision-making for scalable deployment of intelligent vision systems and offers valuable benchmarks for future research in edge AI.

Keywords

Edge AI, Object Detection, Neural Network Optimization, NPU Acceleration, Embedded Vision, Total Cost of Ownership

Sammanfattning

Eftersom efterfrågan på lågenergi- och låglatensinferens ökar inom applikationer som övervakning, robotik och inbyggd vision, blir valet av lämplig kombination av modell och hårdvara avgörande. Trots detta finns det få driftsättningsorienterade studier som ger en heltäckande utvärdering av prestanda och långsiktiga kostnadskonsekvenser för att köra toppmoderna objektdetekteringsmodeller på edge-enheter. Denna avhandling utforskar därför driftsättning och optimering av realtidsmodeller för objektdetektering på NPU-accelererade hårdvaruplattformar för edge AI, med fokus på NXP i.MX 8M Plus med integrerad NPU samt den externa NPU:n Hailo-8L.

Huvudsyftet med detta arbete är att bedöma inferensprestanda, optimeringseffekter och den totala ägandekostnaden (TCO) för kandidatplattformar för edge AI som kör objektdetekteringsmodellerna YOLOv8 och YOLOv9. Den centrala utmaningen är att balansera detekteringsnoggrannhet, körtidseffektivitet och driftsättningskostnad inom begränsningarna för edge computing-hårdvara.

För att hantera detta jämför studien flera konfigurationer, inklusive variationer i ingångsform och kameraströmresolution. Utvärderingen beaktar mätvärden som bildrutor per sekund (FPS), medelvärdet av medelprecision (mAP), strömförbrukning och termiskt beteende. Dessutom utförs en femårig TCO-analys för ett driftsättningsscenario som involverar 100 enheter, inkluderande hårdvaruprissättning, förväntade felfrekvenser och energikostnader.

Resultaten indikerar att Hailo-8L levererar högre inferensgenomströmning och bättre energieffektivitet i termer av FPS per watt. NXP-plattformen visar sig dock vara mer kostnadseffektiv över tid på grund av dess lägre energiförbrukning och högre rapporterade tillförlitlighet. YOLOv8-modeller uppvisar stark prestanda efter kvantisering med minimal noggrannhetsförlust, vilket gör dem väl lämpade för driftsättning på begränsade enheter. YOLOv9-modeller visar varierande resultat, där YOLOv9m bibehåller hög noggrannhet efter kvantisering, medan YOLOv9s lider av betydande försämring.

Denna forskning tillhandahåller ett praktiskt ramverk för att välja edge AI-lösningar baserat på specifika avvägningar mellan prestanda, kostnad och modellrobusthet. Den möjliggör bättre beslutsfattande för skalbar driftsättning av intelligenta visionssystem och erbjuder värdefulla riktmärken för framtida forskning inom edge AI.

Nyckelord

Edge AI, Objektdetektering, Neural nätverksoptimering, NPU-acceleration, Inbyggd vision, Total ägandekostnad

Acknowledgments

I would like to express my deepest gratitude to everyone who contributed to the completion of my Master's thesis.

First and foremost, I extend my sincere thanks to my KTH supervisor, Professor Mihail Matskin, for granting me the freedom to explore this research independently and for fostering an environment of intellectual curiosity. I am also grateful to my KTH examiner, Professor Carlo Fischione, for his invaluable support throughout this journey.

A tremendous debt of gratitude is owed to Yichen Yang, my supervisor at BitSim NOW. His insightful guidance, dedicated mentorship, and patient support in overcoming numerous challenges were instrumental to the success of this work. I also thank BitSim NOW for providing this exceptional opportunity, and I appreciate the contributions of all its staff who, directly or indirectly, supported my efforts.

Lastly, I am profoundly grateful to my family and friends for their unwavering love, encouragement, and support throughout my studies. My time in Sweden has been a truly enriching experience, and it would not have been the same without their belief in me.

Stockholm, June 2025
Eron Ariodito Hermanto

Table of Contents

List of Figures

List of Tables

Listings

List of Acronyms and Abbreviations

AI	Artificial Intelligence
AP	Average Precision
API	Application Programming Interface
AR	Average Recall
ASPP	Atrous Spatial Pyramid Pooling
CLI	Command Line Interface
CNN	Convolutional Neural Network
COCO	Common Objects in Context
CPU	Central Processing Unit
CU	Compute Unit
CV	Computer Vision
DFC	Dataflow Compiler
FCOS	Fully Convolutional One-Stage Object Detection
FP32	32-bit Floating Point
FPGA	Field Programmable Gate Array
FPN	Feature Pyramid Network
FPS	Frames per Second
GPU	Graphical Processing Unit
HailoRT	Hailo Runtime
HAR	Hailo Archive
HD	High Definition
HEF	Hailo Executable Format
IIOT	Industrial internet of Things
INT8	8-bit Integer
IoT	Internet of Things
IoU	Intersection over Union
LiteRT	Lite Runtime
MAC	Multiply-Accumulate
mAP	Mean Average Precision
MCU	Microcontroller Unit
MPU	Microprocessor Unit
MTBF	Mean Time Between Failures
NMS	Non-Maximum Suppression
NPU	Neural Processing Units

ONNX	Open Neural Network Exchange
PC	Personal Computer
PD	Power Delivery
PE	Processing Unit
PTQ	Post Training Quantization
QAT	Quantization Aware Training
R-CNN	Region Based Convolutional Neural Network
RGB	Red Green Blue
RNN	Recurrent Neural Network
RPN	Region Proposal Network
RT-DETR	Real-Time Detection Transformer
SBC	Single Board Computer
SDK	Software Development Kit
SIMT	Single Instruction, Multiple Thread
SM	Streaming Multiprocessor
SoC	System on Chip
SoM	System on Module
SPP	Spatial Pyramid Pooling
SRAM	Static Random Access Memory
SSD	Single Shot Detector
TCO	Total Cost of Ownership
TinyML	Tiny Machine Learning
TOPS	Tera Operations per Second
TPU	Tensor Processing Unit
YOLO	You Only Look Once

Chapter 1
Introduction

This chapter describes the specific problem that this thesis addresses, the context of the problem, the goals of this thesis project, and outlines the structure of the thesis.

1.1 Background

Edge computing refers to performing data processing closer to the data source, which contrasts with traditional cloud computing where data is sent to centralized data centers for processing [1]. By bringing computation to the "edge," it reduces latency, improves bandwidth efficiency, and strengthens privacy and security, which are crucial for latency-sensitive applications like autonomous vehicles, healthcare, and industrial Internet of Things (IIoT) [1–3]. The ability to deploy machine learning models at the edge further enhances real-time decision-making, particularly in object detection tasks [4].

In machine learning applications, edge computing platforms enable the deployment of models directly on embedded devices, eliminating the need for communication with remote cloud servers. This direct execution is crucial for real-time applications like object detection, which involves identifying and localizing objects in images or video streams [1, 3, 5, 6]. Edge computing plays a pivotal role in technologies like autonomous vehicles, surveillance, and industrial automation [6, 7]. However, these models are computationally intensive and they require highly optimized hardware solutions. To address this complex computational requirement, the industry has turned to Artificial Intelligence (AI) accelerators aptly named the Neural Processing Units (NPUs), which offer substantial improvements in machine learning performance, enabling faster and more energy-efficient object detection [8, 9].

As demand for higher performance in edge machine learning tasks grows, NPUs are becoming increasingly common in affordable edge computing platforms. These devices are designed to deliver substantial performance improvements for applications like object detection, offering better speed and efficiency compared to traditional processors [8, 9]. External NPUs, such as Google’s Coral and Hailo, are often paired with low-cost embedded platforms like the Raspberry Pi to enhance performance while maintaining cost-effectiveness [10–12]. To deploy complex models, i.e. object detection, effectively on these affordable devices, it is crucial to optimize and compress the object detection models, reducing their size and computational demands without sacrificing accuracy [13, 14]. This optimization introduces an additional layer of complexity in the deployment process, as balancing performance with resource limitations is a key challenge.

The surge in commercially available devices featuring dedicated NPUs, spanning a broad range of performance specifications and price points, has posed a new challenge to evaluate their practical effectiveness. While manufacturers' specifications and standardized benchmark results provide initial performance indicators, these metrics often fail to accurately reflect performance in real-world conditions. This research will evaluate NPU-accelerated edge computing platforms, considering their suitability for deploying optimized object detection models that can meet industry needs for affordable, scalable, and energy-efficient solutions.

This study aims to contribute to the field of edge computing by providing a structured comparison of edge platforms for applications. The findings will offer a foundation for researchers

and developers to make informed decisions regarding platform selection, optimize deployment strategies, and identify opportunities for innovation in hardware and software design. By standardizing evaluation methodologies and focusing on real-world applications, this research will support the development of sustainable, efficient, and impactful advancements in edge AI technologies.

1.2 Problem

Industries relying on real-time object detection face challenges in selecting efficient, cost-effective edge AI hardware due to variability in NPU-accelerated platforms. Cloud-based solutions introduce latency, bandwidth constraints, and privacy concerns, making edge computing a necessity [1, 3, 9]. However, limited research compares NPU-powered platforms in terms of performance, optimization impact, and cost-effectiveness.

This study benchmarks two NPU-accelerated edge computing platforms with different price points and performance specifications, analyzing their inference speed, accuracy, energy efficiency, and scalability. The findings will guide developers, businesses, and researchers in making informed decisions for deploying object detection models in real-world edge AI applications. Prompting the following research questions for this thesis:

1. How does the performance (inference speed, accuracy, and energy efficiency) of two NPU-accelerated edge computing platforms compare when deploying optimized object detection models?
2. How does model optimization impact the performance of object detection models when deployed on NPU-accelerated edge computing platforms?
3. What is the cost-performance trade-offs of deploying optimized object detection models on different NPU-accelerated edge computing platforms for real-world applications and how do these devices compare in terms of scalability for edge AI applications?

1.3 Purpose

This thesis evaluates NPU-accelerated edge computing platforms for object detection, benefiting industries seeking efficient, cost-effective edge AI solutions. The findings will help technology companies, manufacturers, and infrastructure managers understand price-to-performance trade-offs, enabling informed hardware selection and optimized resource allocation. Additionally, software developers and researchers will gain insights into software development kits (SDKs), development ecosystems, and optimization techniques, fostering innovation in edge AI adoption.

1.4 Goals

The goal of this thesis is to evaluate and compare NPU-accelerated edge computing platforms for deploying optimized object detection models, providing insights into their performance, optimization impact, and cost-effectiveness for real-world applications. This has been divided into the following three sub-goals as referred to in the research questions:

1. Performance Evaluation – Benchmark and compare the inference speed, resource usage, and energy efficiency of two NPU-accelerated edge computing platforms when deploying optimized object detection models.
2. Optimization Impact Analysis – Assess how model optimization techniques influence performance on different NPU-accelerated platforms, identifying trade-offs between efficiency and accuracy.

3. Cost-Performance and Scalability Assessment – Analyze the price-to-performance ratio of the selected platforms and their scalability for edge AI applications in various industrial scenarios.

Deriving from the research questions and the goals, the deliverables and results of the project are as follows:

1. A comparative benchmark report detailing platform performance across key metrics.
2. Insights into model optimization effects on inference efficiency and accuracy trade-off.
3. A cost-performance analysis to guide hardware selection for industrial edge AI deployments.

1.5 Research Methodology

This thesis adopts a quantitative and experimental research methodology, supported by exploratory and descriptive elements during the initial phase. The research is guided by three main questions focused on performance comparison, optimization impact, and cost-performance trade-offs of NPU-accelerated edge platforms for real-time object detection.

In the exploratory phase, candidate platforms and object detection models are selected through a review of publicly available documentation, performance benchmarks, and compatibility with machine learning frameworks and optimization toolchains. This stage informs the experimental design by narrowing the scope to two widely available edge computing platforms with NPU acceleration.

The experimental phase involves deploying and benchmarking an optimized object detection model on both platforms. Performance metrics are quantitatively measured to answer research question 1 and research question 2. Optimizations are applied using platform-specific toolchains, and pre- vs. post-optimization performance is compared.

For research question 3, a cost-performance analysis is performed using the collected metrics and market prices of the platforms. Additionally, a descriptive comparison of scalability potential is conducted to assess real-world deployment feasibility in edge AI scenarios. More details are provided in Chapter 3.

1.6 Delimitations

This research is delimited by the following factors:

- Number of devices benchmarked: The study is conducted using only two NPU-accelerated edge computing platforms.
- Locally relevant: The research is based on commercially and legally available hardware and software in Sweden, where the research is conducted.
- Platform-specific optimization: Each platform requires unique model optimization and compilation processes, which introduce variations in performance due to differences in toolchains.
- Development board setup: The benchmarking is done on development boards. However, a theoretical exploration of production-ready modules is included, considering potential challenges related to hardware integration, development complexity, and time to market.

1.7 Ethics and Sustainability

This thesis addresses several important ethical, environmental, and economic sustainability considerations in the context of deploying real-time object detection on edge AI platforms. A key

ethical concern is privacy, as object detection systems used in applications such as surveillance and robotics may inadvertently infringe on individuals' rights. Prioritizing local processing on edge devices, rather than relying on cloud-based systems, reduces the risk of data breaches and misuse of personal information. This approach also enhances environmental sustainability by decreasing the dependency on energy-intensive data centers, contributing to reduced energy consumption and carbon emissions.

Energy efficiency is a central theme in both the technical and sustainability evaluation of the platforms. By comparing devices based on power consumption, the research helps identify solutions that deliver acceptable performance while minimizing environmental impact. This not only promotes the responsible use of computational resources but also supports industry trends toward greener technologies. Furthermore, the study considers hardware lifecycle and electronic waste, emphasizing the importance of choosing reliable and efficient systems that extend device longevity and reduce the environmental burden of frequent hardware replacements. Ethically, this includes attention to responsible sourcing and disposal practices to prevent harm to communities and ecosystems.

From an economic sustainability perspective, the thesis includes a Total Cost of Ownership (TCO) analysis over a five-year deployment period for 100 devices. This evaluation accounts for not just initial hardware costs, but also operational factors such as energy usage and device reliability. By highlighting trade-offs between performance and long-term affordability, the research supports informed decision-making for scalable and economically sustainable edge AI deployments. Overall, the thesis promotes a balanced approach that integrates ethical responsibility, environmental awareness, and economic viability in the development and deployment of edge computing solutions.

1.8 Structure of the thesis

Chapter 2 provides a comprehensive overview of Edge AI, with a particular focus on object detection, model optimization techniques, and performance benchmarking. It also reviews commercially available NPU-accelerated edge computing platforms, along with their Software Development Kits (SDKs) and associated development ecosystems.

Chapter 3 outlines the research methodology, detailing the approach to platform selection, system setup, model optimization and deployment, data collection, and analysis.

Chapter 4 describes the implementation phase, emphasizing the practical aspects of deploying and executing object detection models on the selected platforms.

Chapter 5 presents the experimental results and offers an in-depth analysis, focusing on key performance metrics and the trade-offs observed across different configurations.

Finally, Chapter 6 summarizes the main findings of the thesis and suggests potential directions for future research.

Chapter 2
Background

This chapter offers foundational insights into edge AI, including recent advancements in AI accelerators and commercially available NPU-powered devices. It also covers key topics such as object detection, optimization techniques, and the fundamentals of edge AI deployment. Furthermore, the chapter reviews relevant research on the application of object detection within edge computing platforms, highlighting the challenges and progress in this area.

2.1 Edge Computing for AI Applications

As artificial intelligence continues to advance, the demand for real-time processing has driven the adoption of edge computing. Traditional cloud-based AI solutions rely on centralized servers to process data, but this approach can introduce latency and require substantial network bandwidth [1, 3, 6, 15]. Edge computing addresses these challenges by enabling data processing to occur closer to the source, reducing reliance on cloud infrastructure and improving responsiveness. This paradigm shift is particularly beneficial for AI applications that require low latency and operate in environments with limited or intermittent connectivity [2, 4, 7, 8, 15].

2.1.1 Definition and Principles of Edge Computing

Edge computing refers to a distributed computing model where data processing occurs at or near the data source rather than being sent to centralized cloud servers [1]. Unlike cloud computing, which depends on remote data centers, edge computing leverages local devices, gateways, or micro data centers to perform computations closer to where data is generated [4]. Figure 2.1 illustrates the differences between cloud-based AI against edge AI.

The key benefits of edge computing include:

- Low Latency: By processing data locally, edge computing significantly reduces the delay associated with transmitting data to and from remote servers, making it ideal for real-time AI applications such as autonomous vehicles, industrial automation, and smart surveillance [1, 3, 4, 7].
- Reduced Bandwidth Usage: Sending raw data to the cloud for processing can be costly and inefficient, especially in applications generating large volumes of data, such as video analytics. Edge computing minimizes bandwidth requirements by processing and filtering data locally before transmitting only relevant insights to the cloud [4, 16].
- Improved Security and Privacy: Keeping sensitive data on local devices rather than transmitting it to external servers reduces the risk of data breaches and enhances privacy compliance, which is critical in industries such as healthcare and finance [1, 4, 16].

By leveraging these principles, edge computing enables faster and more efficient AI applications, particularly in resource-constrained environments where real-time decision-making is essential.

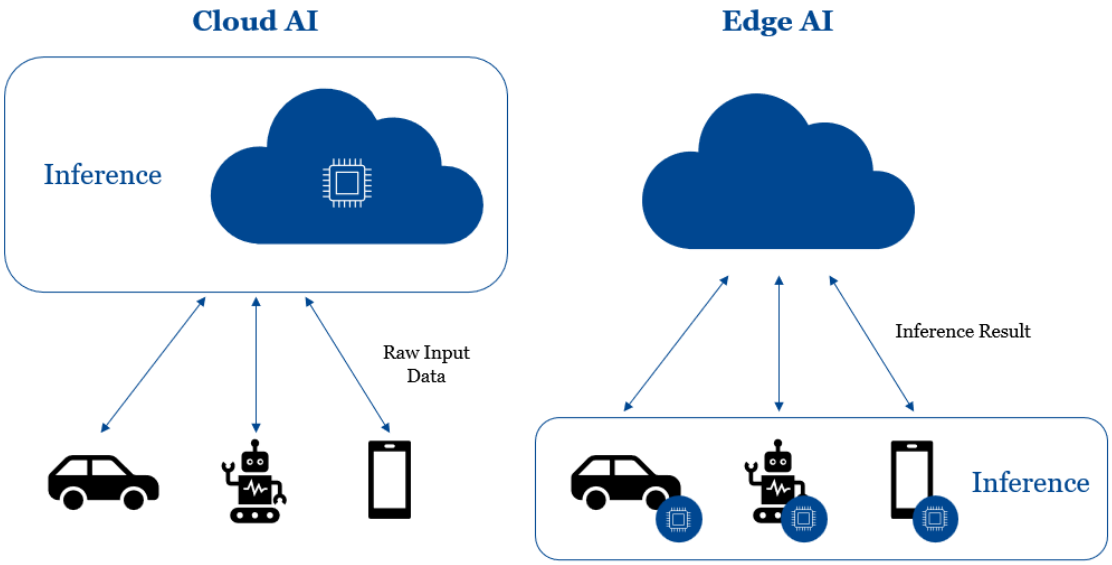


Figure 2.1: **Comparison of cloud-based AI and edge AI.**

2.1.2 Edge AI: Machine Learning on Edge Devices

Edge AI refers to the deployment of machine learning models directly on edge devices such as embedded systems, IoT (Internet of Things) devices, and specialized accelerators. Unlike many traditional AI models that rely on cloud-based inference, Edge AI enables on-device processing, reducing dependence on external infrastructure and enhancing real-time decision-making.

Despite its advantages, deploying AI at the edge presents several challenges:

- Limited Computational Resources: Edge devices typically have lower processing power, memory, and storage compared to cloud-based servers. Running complex AI models efficiently on these constrained devices requires model optimization techniques such as quantization, pruning, and knowledge distillation [4].
- Evolving Technologies: The concept of edge AI is still brand-new and has only begun to reach mass adoption, deployment of edge AI still depends on tools and approaches that were developed for large-scale, server-side AI. It is likely that extremely rapid evolution will see the tools and approaches changing [4, 16].
- Power Constraints: Many edge AI devices operate on battery power, making energy efficiency a crucial factor. Techniques such as hardware acceleration and lightweight model architectures help balance performance with power consumption [8, 9].
- Hardware Heterogeneity: Edge computing environments consist of diverse hardware architectures, ranging from microcontrollers to high-performance graphical processing units (GPUs) and NPUs. Ensuring AI models are compatible across different platforms requires specialized frameworks and optimization strategies [4].

Another important concept within edge AI is embedded machine learning and tiny machine learning (TinyML). Embedded machine learning mostly refers to machine learning inference on embedded systems such as microprocessor unit (MPU), while tiny machine learning goes into even more resource constrained devices such as microcontroller unit (MCU), digital signal processors, and small field programmable gate arrays (FPGAs) [4]. Figure 2.2 illustrates the connection between all the important concepts for edge AI in context to each other.

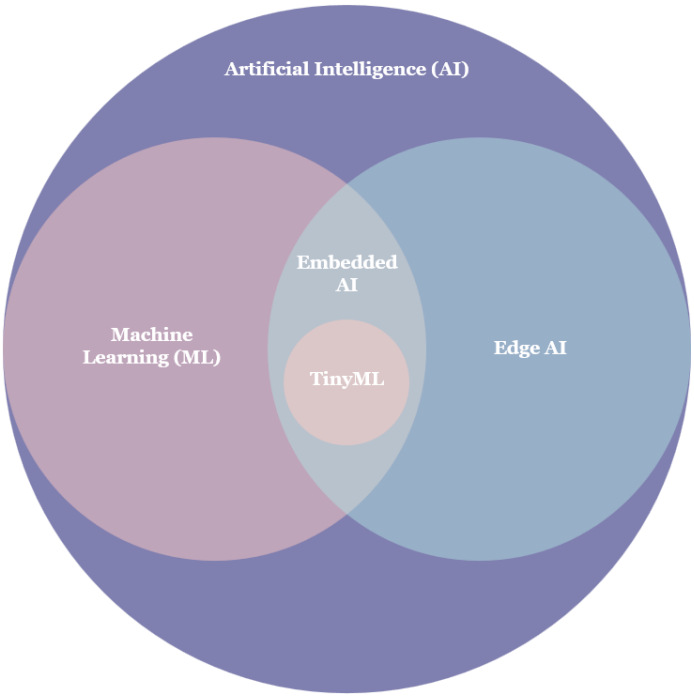


Figure 2.2: Contextual connection of important concepts for TinyML (redrawn based on [4]).

Despite these challenges, advancements in edge AI hardware and software are making it increasingly feasible to deploy sophisticated AI models on resource-limited devices. Technologies such as TensorFlow Lite[1], ONNX (Open Neural Network Exchange), and TinyML[2] enable efficient inference on embedded platforms, paving the way for AI-powered applications in industries such as healthcare, manufacturing, and smart cities [4].

2.2 Neural Processing Units (NPUs) and AI Acceleration

As artificial intelligence becomes increasingly embedded in real-world applications, the demand for both high-performance and energy-efficient computing has grown. Traditional central processing units (CPUs), designed for general-purpose computing, struggle to meet the intensive computational demands of deep learning, particularly for real-time inference tasks. CPUs rely on sequential processing and limited parallelism, making them inefficient for the massive matrix operations and tensor computations required in modern AI workloads [17]. To overcome these limitations, specialized AI accelerators have been developed to optimize deep learning performance [18].

2.2.1 AI Accelerators: GPUs vs NPUs

GPUs were initially designed for rendering complex graphics but have since become widely adopted for AI workloads due to their massively parallel processing capabilities [17, 19]. Their architecture consists of hundreds of streaming multiprocessors (SMs) for NVIDIA or compute units (CUs) for AMD that operate under a Single Instruction, Multiple Thread (SIMT) execution model, enabling them to perform multiple computations simultaneously [20].

[1] Now called LiteRT, many documentations still refer to it as Tensorflow Lite

[2] The term TinyML is a registered trademark by TinyML Foundation, now called Edge AI Foundation

Some notable points about GPU in context of machine learning are as follows:

- High Parallelism: GPUs excel at performing matrix operations, which are fundamental to deep learning tasks such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) [17].
- Flexible Computation: Unlike fixed-function accelerators, GPUs can handle a wide range of AI tasks, from training to inference, making them versatile for different workloads [17].
- Power Consumption: Despite their high performance, GPUs are power-hungry, often consuming dozens to hundreds of watts, which makes them less suitable for edge devices with energy constraints [19, 21].

Neural Processing Units are specialized hardware accelerators designed to optimize the performance of AI and deep learning workloads, specifically for inference tasks. NPUs are built with architectures that prioritize efficient computation of the operations central to neural networks, such as matrix multiplications, convolutions, and tensor operations [17, 19]. Unlike GPUs, which typically operate at higher precision, NPUs improve power efficiency and computational speed by using lower-precision arithmetic, often 8-bit integer (INT8) calculations instead of 32-bit floating point (FP32) [9, 17]. This precision reduction significantly reduces memory and energy consumption while maintaining an optimal balance between model accuracy and efficiency, allowing machine learning models to be smaller, faster, and more suitable for deployment on resource-constrained devices [22].

The fundamental architecture of an NPU consists of Processing Elements (PEs), though different manufacturers may use varying terminology for these components. These specialized units are designed to perform critical operations, such as multiply-accumulate (MAC) operations, which are the backbone of Convolutional Neural Networks [23–25]. A defining feature of NPUs is their on-chip static random access memory (SRAM), which reduces reliance on external memory, minimizing latency bottlenecks and significantly enhancing processing speed and energy efficiency [24].

2.2.2 Commercial NPU-Accelerated Edge Computing Platforms

In recent years, manufacturers have significantly ramped up the development of NPUs in response to the growing demand for AI-powered applications at the edge [26]. With advancements in semiconductor technology and AI-specific optimizations, companies are increasingly integrating NPUs into their products, enabling more efficient processing for machine learning tasks. This surge in NPU development has resulted in a wide range of solutions, from integrated NPUs embedded in system-on-chip (SoC) designs to external NPU accelerators that can be paired with edge devices for enhanced performance.

Integrated NPUs are those where the Neural Processing Unit is embedded within the SoC alongside the CPU and other components [27]. This integration enables a compact, cost-effective solution for edge devices, which is particularly important for IoT devices and embedded systems with limited space and power requirements. Some examples of commercially available SoC with integrated NPU are the STM32MP2 [28], NXP I.MX8M Plus [29], and Rockchip RK3588 [30].

External NPU accelerators are standalone devices that connect to edge computing platforms to provide additional AI processing power. These accelerators can be used in combination with general-purpose SoCs or microcontrollers to boost performance for AI workloads. Some examples of cormecially available external NPU accelerators are the Google Coral TPU (Tensor Processing Unit) [12], Hailo 8 series [10], and Raspberry Pi AI Hat+ that also utilizes Hailo 8 series NPU [11].

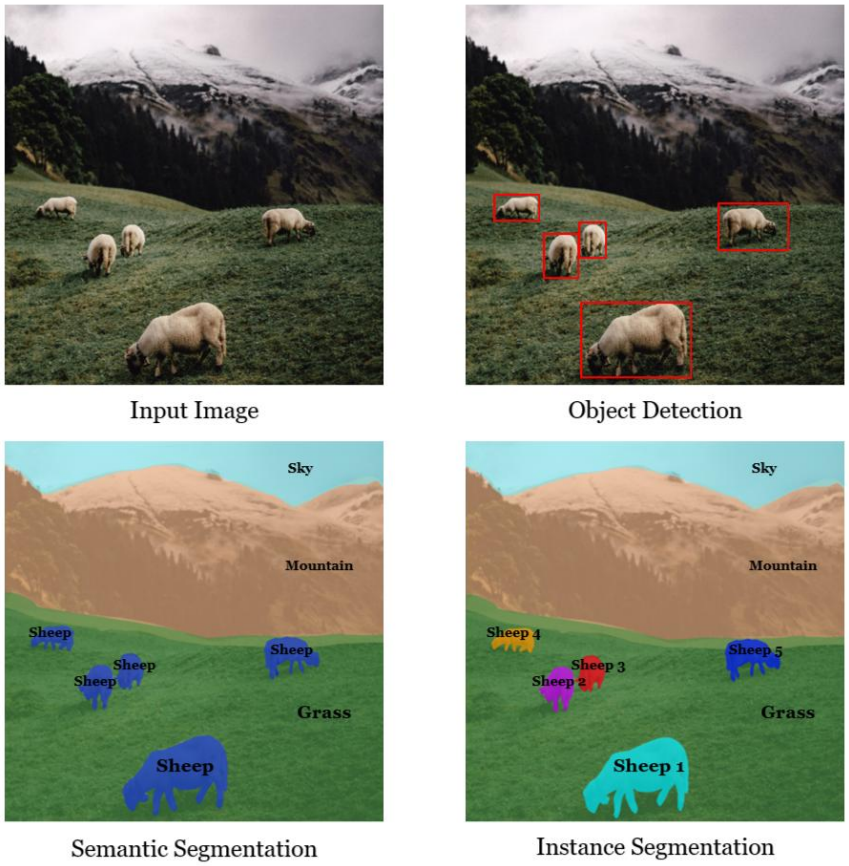


Figure 2.3: **Common computer vision tasks (adapted from [32]).**

2.3 Object Detection

Object detection is a subfield of the broader scientific discipline known as Computer Vision (CV), which focuses on the automatic extraction of meaningful information from images to interpret, analyze, and represent the physical world, either qualitatively or quantitatively [31]. Common tasks in CV include instance segmentation, semantic segmentation, and object detection. Figure 2.3 illustrates these common CV tasks.

Object detection specifically refers to the process of identifying and classifying multiple objects within an image. This involves determining the precise location of each object by drawing a bounding box around it and assigning it to a specific category label [33].

2.3.1 Object Detection Model Architecture

Many popular object detection models follows a general architecture that is typically made up of four key components: the input, backbone, neck, and prediction head as illustrated in Figure 2.4 [34, 35]. The input refers to the image or set of images fed into the model, where the raw data is processed. The backbone is a deep neural network responsible for extracting features from the image, such as textures, edges, and shapes. These features are then passed to the neck, which enhances and fuses the features, preparing them for final predictions. The prediction head takes these processed features and makes the final predictions, including the detection of objects (bounding boxes) and their associated class labels (e.g., person, car, dog).

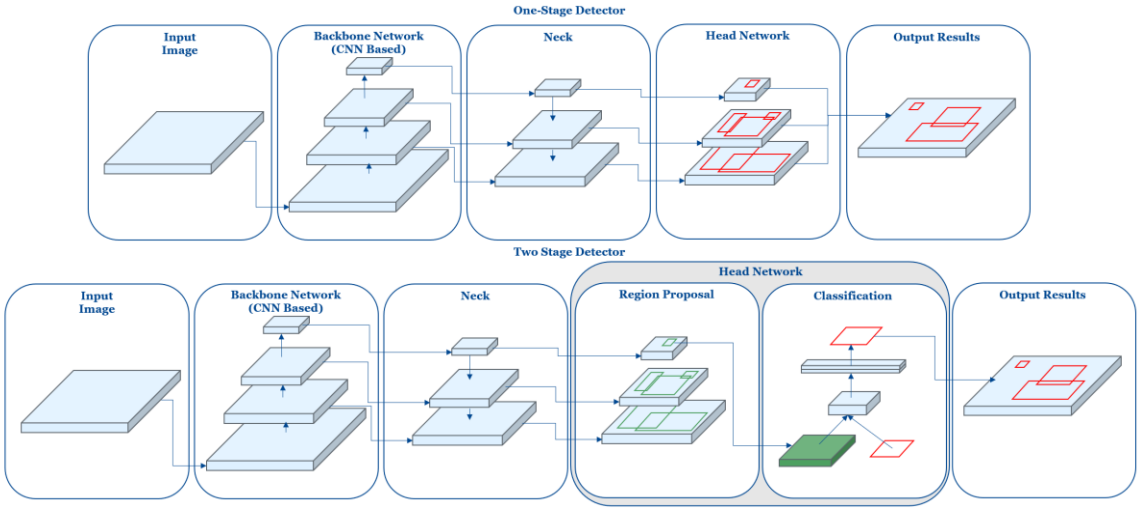


Figure 2.4: **Object detection model architecture (adapted from [36]).**

2.3.1.1 Backbone

The backbone network is a fundamental component in object detection model architecture. It plays a crucial role in extracting and encoding features from the input data, acting as the core feature extractor [34, 37, 38]. The backbone captures both low-level features (such as edges and textures) and high-level features (such as object parts and complex patterns) from the image. Typically, backbones are pretrained on the ImageNet dataset [39], which contains thousands of images with 1,000 different classes. This pretraining enables the backbone to learn essential visual features, such as shapes, textures, and edges, allowing it to effectively process and understand new images during object detection tasks. By leveraging this prior knowledge, the backbone is better equipped to extract meaningful features, reducing the amount of training data and time required for specific object detection tasks. There are several popular backbones used in object detection models, each with their unique strengths and characteristics. Some of the commonly used backbones include ResNet [40], VGG [41], MobileNet [42], and Darknet [43].

2.3.1.2 Neck

The neck in an object detection model is a crucial component that sits between the backbone and the head. Its primary function is to enhance, refine, and combine the feature maps extracted by the backbone to prepare them for the final detection predictions [34, 44]. The neck is especially important for improving multi-scale feature representation, which helps the model effectively detect objects of varying sizes within the image [45, 46].

After the feature has been fused in the neck, it goes through a multi-scale feature pooling layer to further refine and enhance the representation of objects at various scales, ensuring better localization and classification accuracy, particularly for objects of different sizes. This includes techniques such as spatial pyramid pooling (SPP) and atrous spatial pyramid pooling (ASPP) [47, 48].

2.3.1.3 Prediction Head

The prediction head is the final component in an object detection model that takes the refined and processed features from the backbone and neck (which capture different levels of information about the image) and makes the actual predictions [34, 49, 50]. It is responsible for producing the final bounding box coordinates (localization) and class labels (classification) for the objects detected in the image [51, 52].

The primary functions of a prediction head include:

- Object Localization: This component predicts the coordinates of the bounding boxes around detected objects. It adjusts the proposed bounding boxes to accurately fit the objects [49, 52].
- Classification: This component assigns a probability score to each detected object, indicating the likelihood that the object belongs to a particular class [49, 52].
- Confidence Scoring: The head sometimes provides a confidence score for each prediction, indicating how certain the model is about the detection [53].

2.3.2 One Stage and Two Stage Detection

Object detection models can be categorized into two main types: one-stage and two-stage detectors. Both approaches aim to identify and localize objects within images or videos, but they differ in their architecture and processing pipeline as illustrated in Figure 2.4.

2.3.2.1 One Stage Detection

One-stage object detectors directly predict object locations and class labels in a single step from the input image, without a separate region proposal stage [54, 55]. These models treat object detection as a single regression problem, making them fast and efficient, ideal for real-time applications [55]. Some popular examples of one stage object detection models include YOLO (You Only Look Once) [51, 56, 57], SSD (Single Shot Detector) [58], and FCOS (Fully Convolutional One-Stage Object Detection) [49].

2.3.2.2 Two Stage Detection

Two-stage object detectors use a Region Proposal Network (RPN) to first generate candidate object locations before performing classification and bounding box refinement [54, 55]. This extra stage improves accuracy at the cost of speed [55]. Some famous example of two stage detectors are R-CNN (Region Based Convolutional Neural Network) [59] and Faster R-CNN [60].

2.3.3 Non-Maximum Suppression

Non-Maximum Suppression (NMS) is a fundamental post-processing technique employed in object detection systems to eliminate redundant bounding boxes and retain only the most relevant detections [61], as illustrated in Figure 2.5. Object detection models frequently generate multiple overlapping bounding boxes for a single object, each with varying positions, sizes, and confidence scores. For instance, in detecting a car within an image, the model may produce several bounding boxes that overlap significantly but differ slightly in coordinates and confidence levels. NMS addresses this redundancy by selecting the bounding box with the highest confidence score and suppressing all other boxes that have a high Intersection over Union (IoU) overlap with it. This process is repeated iteratively, ensuring that only the most representative bounding boxes are preserved in the final output.

Figure 2.5: **Non-maximum suppression for object detection task (adapted from [61]).**

2.3.4 Common Object Detection Models

Many researchers have come up with their own model for object detection applications prompting the endless choice of models that could be used for this application. However, there are some that have proven their worth either through benchmarks or through popularity, which prompt better support and updates. Some of these famous models are the YOLO [57], SSD [58], and RT-DETR (Real-Time Detection Transformer) [62, 63].

The YOLO family of models is widely regarded for its efficiency, allowing for very fast inference speeds without sacrificing too much accuracy compared to more complex object detection models. YOLO operates using a single neural network that processes an entire image in one pass, dividing it into a grid and predicting bounding boxes and class probabilities simultaneously [57]. This makes it particularly well-suited for real-time applications such as autonomous driving, surveillance, and robotics, where low latency is crucial.

Over the years, the YOLO architecture has undergone multiple iterations, with improvements in accuracy, speed, and adaptability. More recent versions, such as YOLOv5, YOLOv7, and YOLOv8, incorporate enhancements like advanced anchor-free detection, dynamic model scaling, and optimization for hardware-specific acceleration (e.g., TensorRT and OpenVINO) [64]. Additionally, YOLO-based models often come with efficient implementations tailored for edge computing devices, such as EdgeYOLO [65], which is designed to run on low-power hardware like Raspberry Pi, Jetson Nano, and Coral Edge TPU.

A more recent advancement in the field of computer vision is the integration of transformer architectures into object detection models, giving rise to what are known as Vision Transformers (ViTs). These models offer several advantages, including the ability to capture global context, model long-range dependencies, and achieve high detection accuracy, particularly in complex and cluttered scenes [66].

A notable example of this new class of models is RT-DETR, developed by Baidu [62]. RT-DETR is the first real-time, end-to-end object detector based on a transformer architecture. It combines the strengths of transformers, such as their global attention mechanisms and contextual awareness, with innovations like a hybrid encoder and IoU-aware query selection. These features allow RT-DETR to achieve both high accuracy and real-time inference speeds, making it well suited for applications including autonomous driving, robotics, medical imaging, and surveillance [63].

Despite these advantages, a key limitation of transformer-based models is their hardware compatibility [67]. Although optimized for runtime performance, certain operations within these

models are only supported on specific hardware platforms. This limitation can create challenges when deploying such models on edge devices, often requiring significant modifications to the network architecture to ensure compatibility and efficient execution.

2.3.4.1 COCO Pre-trained Model

The Common Objects in Context (COCO) dataset, developed by Microsoft, is a large-scale benchmark widely used in computer vision research. It supports a variety of tasks such as object detection, image segmentation, and image captioning [68]. The dataset contains 330,000 images and covers 80 object categories. Due to its comprehensive annotations and diversity of scenes, COCO has become the gold standard for evaluating the performance of state-of-the-art computer vision models [69]. It provides standardized evaluation metrics, including mean Average Precision (mAP), which enable consistent and robust comparison across different models and methodologies.

Many object detection models are distributed as toolkits or scripts, such as those provided by Ultralytics [70], which facilitate model training and evaluation on custom datasets. However, because of COCO's prominence as a benchmarking dataset, pre-trained model weights are often available. These models are trained on the COCO dataset and serve as a baseline for performance evaluation or as a starting point for fine-tuning on application-specific data.

2.4 Model Optimization for Resource-Constrained Edge AI

Deploying machine learning models on affordable edge devices presents several challenges due to hardware limitations. While NPU has significantly improved the efficiency of machine learning computations on edge devices, most affordable AI-enabled hardware still struggles with limited processing power, memory, and energy efficiency [14, 71]. These constraints make it difficult to run complex deep learning models that typically require high computational resources. To address these challenges, there is a growing need for optimized models that reduce computational load while maintaining high accuracy.

2.4.1 Input Size Adjustment

The input size of an object detection model determines the spatial resolution of the image data processed during inference. Although models are typically trained on a fixed resolution, adjusting the input size at inference time is a common strategy for performance optimization. Smaller input sizes reduce the number of pixel-level computations, thereby lowering the computational workload and increasing throughput, often measured in frames per second (FPS). This makes input resizing a key lever for improving inference speed, particularly in resource-constrained or real-time environments [72, 73].

However, increasing throughput by reducing input size introduces a trade-off. Lower-resolution inputs inherently contain less spatial detail, which can negatively impact the model's ability to detect small or fine-grained objects. The extent of this performance degradation depends on several factors, including the model architecture, the distribution of object sizes in the dataset, and how drastically the input size is reduced.

As a result, selecting an appropriate input resolution for deployment involves balancing the need for real-time inference against the requirement for accurate object detection. Benchmarking across various input sizes is essential for quantifying this trade-off and selecting the optimal configuration for the target application.

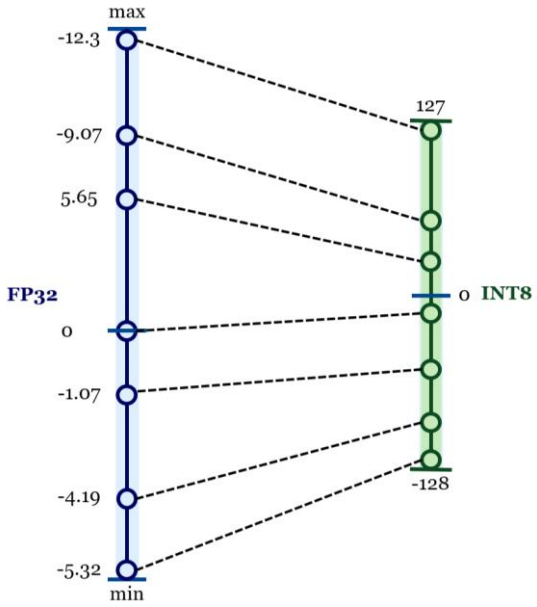


Figure 2.6: **Asymmetric quantization from 32-bit floating point data type to 8-bit integer data type (redrawn based on [75]).**

2.4.2 Quantization

As discussed in section 2.2.1, most NPUs support only limited numerical precisions, such as INT8, in order to enhance power efficiency and computational throughput. Quantization addresses this by reducing the precision of data representation, trading off some degree of model accuracy for improved inference speed and reduced memory usage [13].

Since most machine learning models are originally trained and represented using FP32 data types, quantization typically involves converting these models to INT8 format. This conversion is often performed using asymmetric quantization, where the zero point is shifted to better match the dynamic range of real-world input data [74]. Asymmetric quantization is particularly beneficial when input values are not symmetrically distributed around zero. It helps preserve accuracy while reducing model size and computational complexity. This quantization operation is illustrated in Figure 2.6.

To perform asymmetric quantization, calibration data is required. This data is used to compute two essential parameters: the scale factor (s) and the zero point (z). The scale factor maps the range of real-valued inputs to the quantized integer range, while the zero point adjusts the mapping to align the floating-point zero with a quantized integer value [75]. These parameters are calculated using the following expressions:

$$s = \frac{128 - (-127)}{\alpha - \beta} \tag{1}$$

$$z = \text{round}(-s \cdot \beta) - 2^{\text{bit}-1} \tag{2}$$

Where:

- α and β are the maximum and minimum values observed in the calibration dataset,
- bit represents the bit-width of the target data type (8 for INT8).

Once quantized, data values can be converted from and to the original floating-point format using the same scale and zero point:

$$x_{\text{quantized}} = \text{round}(s \cdot x + z) \tag{3}$$

$$x_{\text{dequantized}} = \frac{x_{\text{quantized}} - z}{s} \tag{4}$$

Quantization for neural network models can be categorized into two categories:

- Post training quantization (PTQ): PTQ convert pre-trained model to a lower precision e.g. from FP32 to INT8, without any additional training [76].
- Quantization aware training (QAT): QAT incorporates the quantization process during training, enabling the model to adapt its parameters to work with lower precision representations. Its main goal is to optimize the loss function while considering quantization effects, ensuring that the final quantized model retains strong performance [76].

2.4.3 Hardware Specific Optimizations

Different hardware manufacturers provide proprietary tools and compilers designed to optimize the execution of neural network models on their respective platforms. These tools are built to fully exploit the specialized capabilities of their NPUs, such as low-power operation, parallel execution, and dedicated memory hierarchies [77–80].

Despite variations in implementation, these toolchains typically follow a similar workflow. A trained model, commonly in ONNX or TensorFlow format, is taken as input. The toolchain then processes the model and outputs a hardware-specific representation, often in a proprietary format, that is optimized for deployment on the target NPU.

Hardware-specific optimization is essential for achieving efficient inference performance on NPUs. These optimizations involve intelligently mapping the model's operations (such as convolutions, activations, and data movement) onto the hardware's computational architecture. The success of this mapping is heavily influenced by the SDK provided by the NPU vendor.

The SDK typically includes compilers, delegates, runtime APIs, and debugging tools. It governs how developers interact with the hardware and determines the level of optimization possible. A well-designed SDK enables precise control over how operations are scheduled, memory is allocated, and workloads are parallelized. As such, the SDK is a critical component in maximizing performance and minimizing resource usage on embedded AI hardware.

2.5 Performance Benchmarking for Edge AI Platforms

Performance benchmarking is essential for evaluating the effectiveness of different Edge AI platforms in real-world deployment scenarios. The benchmarking process involves measuring key performance metrics to assess computational efficiency, model accuracy, and trade-offs between performance and cost.

2.5.1 Key Performance Metrics for Edge AI Evaluation

To accurately compare different Edge AI platforms, it is essential to define key performance metrics that influence the efficiency and effectiveness of these systems. These metrics are broadly categorized into computational and hardware performance, as well as algorithmic (model) performance [4].

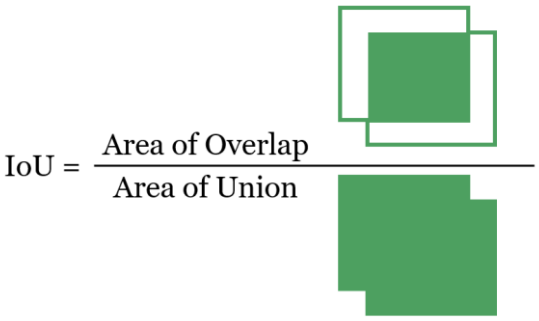


Figure 2.7: Calculation of intersection over union (redrawn based on [83]).

2.5.1.1 Computational and Hardware Performance

Computational and hardware performance metrics focus on the efficiency of the underlying processing units and how well they execute AI workloads [4]. Key metrics include:

- Inference Latency: Measures the time taken for the model to process an input and generate an output [4, 81]. Lower latency is crucial for real-time applications such as autonomous vehicles and surveillance systems.

- Throughput (FPS - Frames Per Second): The number of inferences processed per second, determining the system's capability to handle video and inference streams efficiently [81].

- Power Consumption: Evaluates the energy efficiency of an edge AI device by measuring the power drawn during inference [4]. Power-efficient hardware is critical for battery-powered applications such as drones and IoT devices.

- Memory Footprint: The amount of RAM and cache required to run AI inference, impacting the feasibility of deploying models on resource-constrained edge devices [4, 14].

- Thermal Performance: Measures the heat generated by the edge AI hardware under different workloads, which can impact reliability and longevity [4].

2.5.1.2 Algorithmic (Model) Performance

Algorithmic performance measures the effectiveness of AI models deployed on edge platforms, considering aspects such as accuracy and robustness [4]. The most important metrics to determine the performance of object detection stems from the IoU metrics, which quantifies the amount of overlap between the prediction bounding box to the ground truth bounding box [82], as illustrated in Figure 2.7.

A crucial aspect of evaluating object detection models is understanding the concepts of true positives, false positives, and false negatives, as these serve as the foundation for calculating precision and recall metrics. In object detection, a true positive occurs when the model correctly identifies an object, and the predicted bounding box has an IoU above a predefined threshold with the ground truth. A false positive happens when the model incorrectly predicts the presence of an object in an area where none actually exists, or the prediction is made with no intersection to the ground truth object. Conversely, a false negative occurs when the model fails to detect an object that is present in the image. Unlike classification tasks, the concept of true negatives does not apply to object detection since the primary focus is on identifying and localizing objects rather than distinguishing between object and non-object regions. These concepts are illustrated in Figure 2.8.

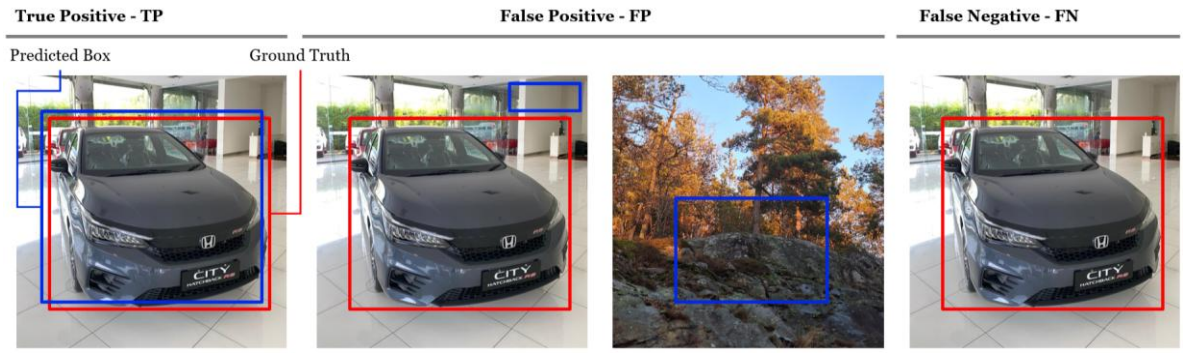


Figure 2.8: **True positive, false positive, and false negative in object detection context [83]).**

With these concepts, we can then calculate two of the most popular metrics used in object detection benchmarks, which are precision and recall. Precision metric aims to quantify the accuracy of positive predictions made by the model, while recall aims to quantify the ability of the model to capture all relevant object in an image [82]. The formula for each is as follows

$$precision = \frac{TP}{TP + FP} \tag{5}$$

$$recall = \frac{TP}{TP + FN} \tag{6}$$

Most benchmarks for object detection end up with a single metric used to compare the performance of these models, which is the mean average precision (mAP). Usually the IoU will be put at a threshold such as mAP@0.5 or mAP@0.75 which means that the mAP is calculated for IoU over 0.5 or IoU over 0.75 [84]. mAP is calculated by averaging the average precision over all the classes as follows

$$mAP = \frac{1}{C}\sum_{i=1}^{C} AP_i \tag{7}$$

2.5.2 Trade-offs Between Performance and Cost

Deploying AI models on edge computing platforms requires carefully balancing performance and cost to meet application-specific requirements. This trade-off often involves optimizing energy efficiency while maintaining sufficient computational power, ensuring high model accuracy without compromising real-time responsiveness, and selecting hardware that offers the best value for its performance [4]. Striking the right balance is crucial for achieving efficient and cost-effective edge AI deployments across various use cases.

2.6 Selecting an Edge Computing Platform

Selecting the right NPU-based edge computing platform is critical for deploying real-time machine learning applications efficiently. NPUs offer dedicated acceleration for AI workloads, enabling faster and more power-efficient inference. However, different NPU architectures, SDKs, and optimization strategies significantly impact deployment outcomes, necessitating a careful selection process.

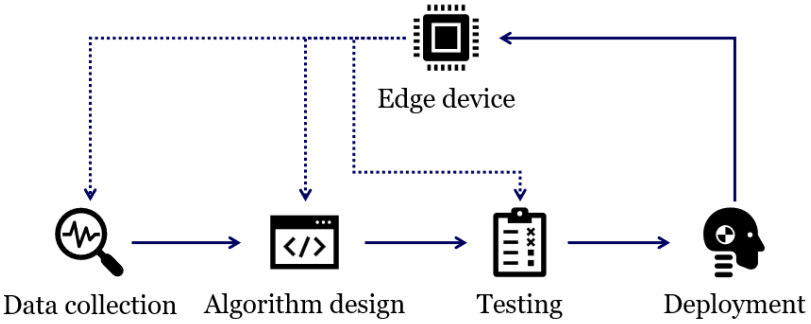


Figure 2.9: **Iterative approach to deploying edge AI application (modified from [4]). The solid line represents the linear development process, while the dashed line illustrates the iterative refinement steps.**

2.6.1 Importance of Platform Selection

Choosing an appropriate NPU-based edge computing platform directly influences inference performance, power efficiency, and ease of integration. The following factors highlight the significance of platform selection:

- Variability in NPU Architectures: Different vendors offer NPUs with unique architectures, optimized for specific AI operations. Some NPUs prioritize power efficiency, while others focus on raw computational speed. Some NPUs also only support certain operations and offload unsupported operations to the CPU.

- Proprietary SDKs and Software Ecosystems: Each NPU manufacturer provides specialized SDKs, such as NXP eIQ, STM32 X-CUBE-AI, and Hailo Dataflow Compiler. The choice of SDK affects model compatibility, optimization potential, and ease of deployment.

2.6.2 Edge AI Deployment

When deploying an edge AI project, the development process often follows an iterative approach, as illustrated in Figure 2.9. This methodology reflects the interdependent nature of the system components, where decisions in one area directly influence constraints and possibilities in others [4]. For instance, selecting a particular object detection model introduces specific requirements in terms of computational complexity, supported operations, and memory footprint—factors that determine whether a given hardware platform can support the model effectively. Conversely, if the hardware platform is predetermined, it may limit the range of compatible models due to restrictions in performance or supported frameworks. Understanding this bidirectional influence is critical, as the primary goal in edge AI deployment is to achieve a solution that meets application-specific requirements for accuracy, efficiency, and real-time performance.

The overall architecture of an edge AI project, as depicted in Figure 2.10, is critical to the successful deployment of object detection applications on edge devices. Essential components such as sensor drivers, hardware APIs, and the device's firmware and operating system significantly influence the system's performance, compatibility, and efficiency. These elements form the foundational integration layer between the AI model and the underlying hardware, affecting how effectively real-time data is captured, processed, and analyzed. As such, careful consideration of these components is vital to ensuring a robust and responsive edge AI deployment.

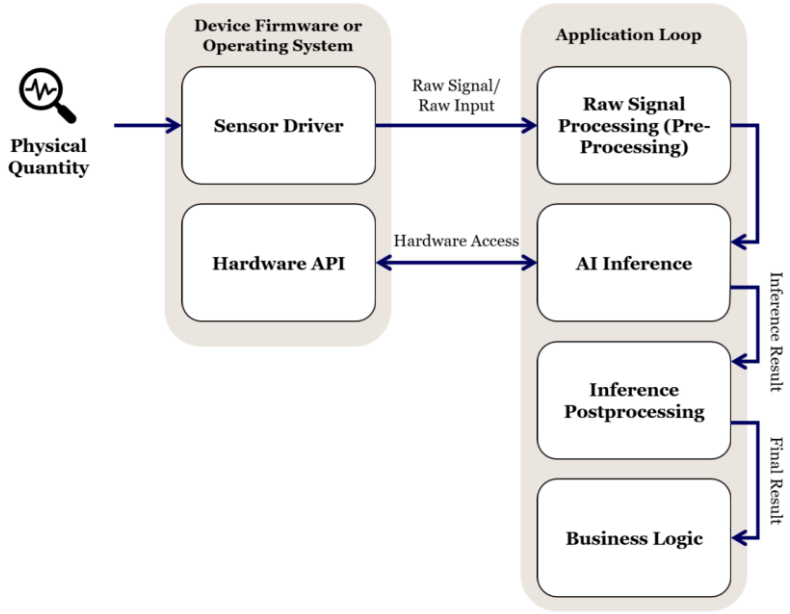


Figure 2.10: Architecture of an edge AI application (modified from [4]).

2.7 Related work

Previous studies provide valuable context for this thesis, particularly regarding model compression techniques and benchmarking across hardware architectures.

2.7.1 Compression of EdgeYOLO model

Although EdgeYOLO is already optimized for edge computing platforms, there is still a demand for even smaller models that can run on more cost-effective hardware. In the paper "Squeezed Edge YOLO: Onboard Object Detection on Edge Devices" by Humes et al. [85], the authors further compressed EdgeYOLO to run on the resource-constrained GAP8 processor, which has only 64KB of L1 RAM, 512KB of L2 RAM, and off-chip RAM. By reducing input size and optimizing convolutional layers and residual blocks, Squeezed Edge YOLO achieved an 8× compression, improving energy efficiency by 76% while delivering a 3.3× increase in throughput. This study illustrates the importance of model streamlining for deployment on constrained hardware.

2.7.2 YOLOv5 benchmark for CPU, GPU, and NPU

In the study "An Efficiency Comparison of NPU, CPU, and GPU When Executing an Object Detection Model YOLOv5" by Abo [21], the author benchmarked the power consumption and inference performance of YOLOv5 across three different processing architectures. The results showed that while NPUs were slower than high-performance GPUs, they offered significantly better energy efficiency compared to both CPUs and GPUs, making them ideal for power-constrained edge applications. These findings underscore the suitability of NPUs for power-sensitive edge deployments, where maintaining energy efficiency is as critical as achieving real-time inference.

Chapter 3
Methodologies

This chapter outlines the research process undertaken in this project. Section 3.1 presents an overview of the research process. Section 3.2 describes the preliminary selection of edge computing platforms and object detection models. Sections 3.3 and 3.4 explain the data collection methodologies employed. Section 3.5 outlines the analytical approach used to interpret the results. Finally, Section 3.6 provides details on the hardware and software configurations to ensure reproducibility.

3.1 Research Process Outline

Figure 3.1 illustrates the sequence of steps followed to carry out this research, which is organized into five main activities. The process begins with the preliminary selection of the edge computing platforms and object detection models. This is followed by initial platform setup and basic deployment testing. Next, pre-trained models are optimized and benchmarked. The optimized models are then deployed on the selected platforms for platform benchmarking. The final step involves analyzing the collected results.

3.2 Preliminary Selection

The project begins with the preliminary selection of both the edge computing platforms and the object detection models. Selecting these components is challenging, as they are interdependent, with each affecting the compatibility and performance of the other. To streamline the process, it was decided to first choose the platforms, followed by the selection of the models.

3.2.1 Platform Selection

Platform selection begins with defining the selection criteria, which are based on the specific requirements of the project. Through web research and consultations with employees from the partner company, several candidate platforms are identified and evaluated.

3.2.1.1 Platform Selection Criteria

The selection criteria encompass various specifications and factors important for choosing an NPU-accelerated device capable of running object detection models. Key considerations include the specifications of both the CPU and NPU, with particular emphasis on the NPU. Additional factors include the operating system, platform cost, and the availability of development boards. It is important to note that some platforms, such as those on NXP chips, refer to the chipsets themselves. Therefore, cost and availability are assessed based on the manufacturers producing single-board computers (SBCs) or system-on-modules (SoMs) that use these chipsets.

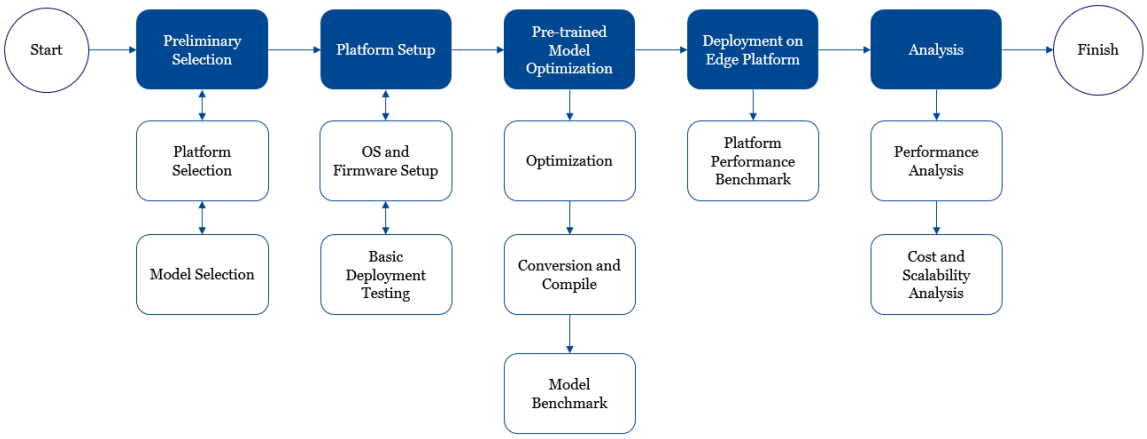


Figure 3.1: **Research process flow diagram.**

3.2.1.2 Candidate Platform Overview

Table 3.1 presents an overview of several candidate platforms considered for this project. The evaluation of each platform's SDK is based on a descriptive assessment of its quality, including the availability of support and documentation related to machine learning applications.

3.2.1.3 Selected Platforms

After careful consideration, two platforms were selected for the remainder of the thesis. The first, representing the mid-lower range of NPU acceleration, is the NXP i.MX8M Plus, offering 2.3 TOPS (Tera Operations per Second) of acceleration. Specifically, the Variscite DART-MX8M-PLUS Evaluation Kit was chosen. To represent the mid-range level of NPU acceleration, the Raspberry Pi 5, paired with the external Hailo-8L accelerator providing 13 TOPS, was selected.

This dual-platform setup enables comparative evaluation of inference performance, power consumption, and cost-efficiency across devices with differing design philosophies, one with integrated NPU and lower cost, and another with external acceleration and higher raw throughput.

3.2.2 Model Selection

Model selection follows the steps of platform selection. Through web research and consultations with employees from the partner company, several candidate models are identified and evaluated

3.2.2.1 Model Selection Criteria

The model selection criteria are based on benchmark metrics reported by the model developers or publicly available sources. Key benchmarks include COCO dataset mAP at an IoU of 0.5 and reported inference latency, with consideration given to the hardware used during those evaluations. Additional factors include the model's architecture, compatibility with the NXP and Hailo platforms, and its perceived popularity within the community.

Table 3.1: **Candidate edge computing platforms overview.**

Platforms	CPU	NPU	SDK	Listed Price	Availability
NXP I.MX95	6x 2.0GHz Arm Cortex-A55 + 800MHz Arm Cortex-M7 + 250MHz Arm Cortex-M33	2 TOPS	Medium	-	Not Available at time of writing
NXP I.MX8M Plus	1.8GHz Quad Cortex™-A53	2.3 TOPS	Medium	169 USD	Available through distributor
STM32MP257	Dual Arm Cortex-A35 @1.2GHz, Cortex-M33 @400MHz	1.2 TOPS	High	331 SEK	Available through distributor
Raspberry Pi 5 + Hailo 8L	Broadcom BCM2712 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU	13 TOPS	Very High	714 SEK (SBC) + 959 SEK (NPU)	Available through distributor and online stores
Raspberry Pi 5 + Hailo 8	Broadcom BCM2712 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU	26 TOPS	Very High	714 SEK (SBC) + 1499 SEK (NPU)	Available through distributor and online stores
STM32N6	Arm Cortex-M55 MCU, 800MHz	0.6 TOPS	Medium	245 SEK	Not Available at time of writing

3.2.2.2 Candidate Models Overview

Table 3.2 provides an overview of the candidate object detection models considered for this project. Each model family includes variants with different parameter sizes, which affect both latency and mAP. For consistency, the results shown are based on the smallest model size within each family.

3.2.2.3 Selected Models

After evaluating the candidates against the criteria, YOLOv8 and YOLOv9 were selected for detailed benchmarking. These models were chosen due to their strong balance of accuracy, inference speed, active development.

For YOLOv8, the nano, small, and medium versions of the pre-trained COCO variants were selected. YOLOv8 is developed and maintained exclusively by Ultralytics and is licensed under AGPL 3.0, which has implications for commercial deployment. However, it remains one of the most actively developed and widely used object detection frameworks.

Table 3.2: **Candidate object detection models overview.**

Models	Reported Latency	COCO Reported mAP	Architecture	Compatibility with NXP	Compatibility with Hailo	Perceived Popularity
YOLOv12 [86]	1.64 ms on T4 TensorRT	40.6	CNN One Stage	Unknown	Unknown	Low
YOLO11 [87]	1.5 ms on A100 TensorRT	39.5	CNN One Stage	Unknown	Compatible	Medium
YOLOv9 (Ultralytics and MIT Version) [88, 89]	-	38.3	CNN One Stage	Compatible	Compatible	High
YOLOv8 [90]	0.99 on A100 TensorRT	37.3	CNN One Stage	Compatible	Compatible	Very High
MobileNet-SSD [91]	7.7 ms on Hailo-8L	24.18	CNN One Stage	Compatible	Compatible	High
Faster R-CNN [60]	-	-	CNN Two Stage	Unknown	Unknown	Low
EfficientDet [46]	10.2 ms on Tesla V100	34.6	CNN One Stage	Compatible	Compatible	High

For YOLOv9, both Ultralytics and a version maintained by Henry Tsui [92] (MIT license) are available. The MIT-licensed version was selected to ensure that the study includes fully open-source models suitable for use without legal constraints. This version includes pre-trained COCO models for the small and medium variants, which were used in the thesis. Notably, the Ultralytics version of YOLOv9 was excluded due to licensing limitations and its overlap with YOLOv8's infrastructure.

3.2.3 Platform Setup Methodology

The platform setup is conducted to ensure that all peripherals, particularly the camera and network communication, are functioning correctly. This process also includes basic deployment testing using demo software provided by each platform manufacturer. The goal is to verify that both the NPU and hardware-level communication are operating as expected.

3.3 Pre-trained Model Optimization Methodology

The pre-trained models are taken from each respective model hub [90, 92], these pre-trained models are then optimized to make sure that each platform can execute it on a hardware level on each respective NPU. Input size adjustments and quantization are the method used to try and compress the model complexity as optimization effort.

3.3.1 Model Input Size Adjustment Methodology

Input size adjustment is applied as a model optimization technique to reduce computational complexity and memory usage during inference. Although originally the model is trained on a 640x640 dataset, smaller input resolutions decrease the number of operations required by convolutional layers, leading to faster processing times in trade of accuracy.

The chosen input sizes for this project are 640×640, 416×416, and 320×320, which are commonly used in object detection models. Additionally, since both YOLOv8 and YOLOv9 utilize a backbone with a maximum stride of 32, it is essential that the input size be divisible by 32 to avoid misalignment during operations such as upsampling and concatenation. If the input size is not divisible by 32, misalignment can occur, potentially causing issues in the network's performance.

3.3.2 Model Quantization Methodology

In this project, model quantization is applied to reduce the size of deep learning models and improve inference speed, making them more suitable for deployment on edge devices with limited computational resources. Furthermore, quantization is necessary due to hardware constraints: both the NXP i.MX8M Plus and the Hailo-8L accelerators are designed to perform optimally with 8-bit integer operations. While the NXP platform does support 16-bit inference, its performance is significantly better with 8-bit quantized models. The Hailo-8L, on the other hand, supports mixed-precision execution with a combination of 8-bit and 4-bit integer operations, allowing for further efficiency improvements when lower-precision representations are viable without significant accuracy loss. All models are quantized to 8-bit as a baseline to ensure compatibility with NXP while enabling HAILO to apply additional internal optimizations through its compiler

The quantization method used in this project is post-training quantization, which enables model compression without requiring additional fine-tuning or retraining. This approach is well-suited for scenarios where training resources are limited or access to original training data is restricted. To preserve model accuracy during quantization, a representative calibration dataset is used to collect activation statistics from the original floating-point model. These statistics guide the mapping of floating-point values to integer ranges, helping to minimize quantization-induced errors and ensure more effective deployment on the target NPUs.

3.3.3 Model Compilation Methodology

To ensure that the optimized models can be executed on the NXP i.MX8M Plus and HAILO-8L platforms, a compilation process is applied to convert the models into platform-specific formats. This step is crucial for achieving compatibility between the model and the target hardware architecture, facilitating efficient inference on the respective NPUs. The compilation process involves using platform-specific toolchains and formats, such as the LiteRT format and delegate for NXP and the Hailo Dataflow Compiler (DFC) toolkit for Hailo. These tools adapt the models to the unique capabilities and constraints of each platform, ensuring optimized execution and maximizing inference performance.

3.3.4 Model Benchmark Methodology

To evaluate the performance of the optimized model, benchmark tests are conducted using the standard COCO dataset. Accuracy is assessed in terms of average precision (AP) and average recall (AR), which are standard metrics for evaluating object detection models. A reference benchmark is first performed on the unoptimized pre-trained model to establish a baseline for comparison; this benchmark is executed on the host computer. The optimized model is then benchmarked directly on the target edge platforms, as the executable files are designed specifically for these devices. This on-

device benchmarking enables a realistic assessment of the model's performance in an edge deployment scenario.

Table 3.3: pycocotools output metrics detail.

Metrics name	Description
Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100]	Mean Average Precision over the IoU threshold of 0.50 to 0.95 with increments of 0.05.
Average Precision (AP) @[IoU=0.50 | area= all | maxDets=100]	Average Precision over the IoU threshold of 0.5.
Average Precision (AP) @[IoU=0.75 | area= all | maxDets=100]	Average Precision over the IoU threshold of 0.75.
Average Precision (AP) @[IoU=0.50:0.95 | area= small | maxDets=100]	Mean Average Precision over the IoU threshold of 0.50 to 0.95 with increments of 0.05 for object with area less than 32^2 pixels.
Average Precision (AP) @[IoU=0.50:0.95 | area=medium | maxDets=100]	Mean Average Precision over the IoU threshold of 0.50 to 0.95 with increments of 0.05 for object with area between 32^2 and 96^2 pixels.
Average Precision (AP) @[IoU=0.50:0.95 | area= large | maxDets=100]	Mean Average Precision over the IoU threshold of 0.50 to 0.95 with increments of 0.05 for object with area greater than 96^2 pixels.
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 1]	Average Recall calculated considering only the top 1 detection per image. It measures the model's ability to find the most relevant object.
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 10]	Average Recall calculated considering only the top 10 detection per image.
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets=100]	Average Recall calculated considering only the top 100 detection per image.
Average Recall (AR) @[IoU=0.50:0.95 | area= small | maxDets=100]	Average Recall calculated for small area less than 32^2 pixels.
Average Recall (AR) @[IoU=0.50:0.95 | area=medium | maxDets=100]	Average Recall calculated for area between 32^2 and 96^2 pixels.
Average Recall (AR) @[IoU=0.50:0.95 | area=medium | maxDets=100]	Average Recall calculated for large area greater than 96^2 pixels.

The benchmarking of the model will employ the widely recognized pycocotools library, a standard evaluation toolkit for the COCO dataset. This tool outputs a suite of twelve distinct metrics, each providing a nuanced perspective on the model's detection capabilities, as detailed in Table 3.3. Among these comprehensive metrics, the primary output that will be the focus of detailed analysis is the mean Average Precision (mAP), often denoted as AP@0.5:0.95. This key metric represents the average of the Average Precision scores calculated at IoU thresholds ranging from 0.5 to 0.95 in increments of 0.05. The mAP provides a holistic measure of the model's accuracy in localizing and classifying objects across varying degrees of overlap with the ground truth, offering a robust overall indicator of the optimized model's detection prowess.

3.4 Platform Benchmark Methodology

Platform benchmarking is conducted to evaluate the performance of the edge computing devices when executing the optimized object detection models. The primary metrics assessed include throughput (frames per second), power consumption, memory usage, temperature, CPU utilization, and latency. These metrics are measured at two commonly used input resolutions: Full High

Definition (HD) at 1920×1080 and HD at 1280×720, to capture the performance impact of varying stream resolutions under realistic deployment scenarios.

3.4.1 Throughput Measurement Methodology

The system's throughput is evaluated based on the FPS observed at the display output. Specifically, it is measured as the interval between the presentation of consecutive frames to the display. This approach ensures that only perceivable throughput is captured, effectively accounting for any frames dropped in the rendering or display queue.

3.4.2 Power Measurement Methodology

Power consumption is measured using an inline wattmeter, which records the total energy usage in watt-hours over a specified period. This approach captures the consumption of the entire device, including all peripherals and any inactive connectors that may draw power even when idle. To account for variability, power usage is monitored over a 10-minute interval, after which the average power consumption in watts is calculated.

$$Power\ Usage\ (Watt) = \frac{Consumed\ Energy\ (watt\ Hour)}{\frac{Elapsed\ time\ (seconds)}{3600\ seconds}}$$

Additional measurements are taken when the device is idle (powered on but not performing any tasks). By subtracting the average idle power consumption from the power usage during inference, the power consumption specifically attributable to the inference process can be isolated.

$$Inference\ Power\ Usage\ (Watt) = Total\ Power\ Usage\ (Watt) - Idle\ Power\ Usage\ (Watt)$$

3.4.3 RAM Measurement Methodology

RAM usage is measured over time using the same averaging approach applied in the power measurements, in order to account for variability. Both overall system memory consumption and the memory usage of the inference script are recorded via software-level APIs. This allows for a more detailed understanding of memory demands associated specifically with inference tasks.

3.4.4 Temperature Measurement Methodology

Chip temperatures are measured using available software APIs and averaged over time to account for fluctuations. For systems with an external NPU, both the NPU and CPU temperatures are recorded separately. In contrast, for systems with an integrated NPU and CPU, a single combined temperature reading is recorded.

3.4.5 CPU Utilization Measurement Methodology

CPU utilization is measured as a percentage of the processor's total capacity. Following the same methodology used for temperature and RAM measurements, CPU usage is recorded over time, and the average utilization is computed using software-level APIs (Application Programming Interfaces). This metric is crucial for understanding the available headroom for tasks such as network communication or other business logic, ensuring that the CPU is not overloaded and can efficiently handle additional processes.

3.4.6 Latency Measurement Methodology

Two types of latency are measured in this study, inference latency and end-to-end system latency. Inference latency refers specifically to the time taken by the platform to process and run the model for a single inference, excluding any preprocessing or postprocessing of the images. In contrast, end-to-end latency encompasses all stages of the process, from capturing the image with the camera to displaying the inference result on the screen.

3.5 Performance Analysis Methodology

The performance analysis methodology consists of two primary components: a comparative evaluation of platform performance, and an assessment of the speed–accuracy trade-off resulting from model optimization efforts.

3.5.1 Platform Comparison Methodology

Platform comparison is conducted using key performance metrics to evaluate both raw processing capability and overall system efficiency. Throughput (FPS) and inference latency are analyzed to assess the inference performance of each platform. Power consumption and thermal behavior are compared to evaluate energy efficiency and heat management.

To evaluate deployment flexibility, a comparison of performance metrics is conducted using both Full HD (1920×1080) and HD (1280×720) camera input resolutions. This helps determine how resolution impacts inference performance and platform suitability in practical scenarios.

3.5.2 Model Optimization Analysis Methodology

The trade-off between inference speed and detection accuracy is examined by evaluating the performance of various model architectures and configurations relative to their original, unoptimized counterparts. The analysis focuses on the impact of two primary optimization techniques: quantization and input size reduction. Accuracy is assessed by using standard mAP metrics, while inference speed is evaluated through latency measurements. By comparing these metrics before and after optimization, the study aims to quantify the efficiency gains achieved and assess the degree of accuracy degradation incurred. This approach provides a structured evaluation of how optimization efforts affect the overall effectiveness of object detection models when deployed on edge computing platforms.

3.6 Cost Analysis Methodology

The cost and scalability analysis are conducted to assess the practical feasibility of deploying object detection models on the evaluated edge computing platforms in real-world scenarios. This evaluation considers factors such as device cost, operational power consumption, thermal efficiency, and resource utilization to determine the economic and technical viability of scaling deployments across multiple units or varied application environments.

The cost analysis is based on a theoretical Total Cost of Ownership (TCO) model. The primary cost drivers considered in this analysis are the price of the SoM or chipset for each platform and their associated power consumption during operation. These components represent the most consistent and impactful variables influencing long-term deployment cost. Other hardware components such as carrier boards, cameras, and peripheral modules are excluded from the core cost calculation due to their variability across deployment contexts; they are instead treated as project-dependent variables subject to specific application requirements.

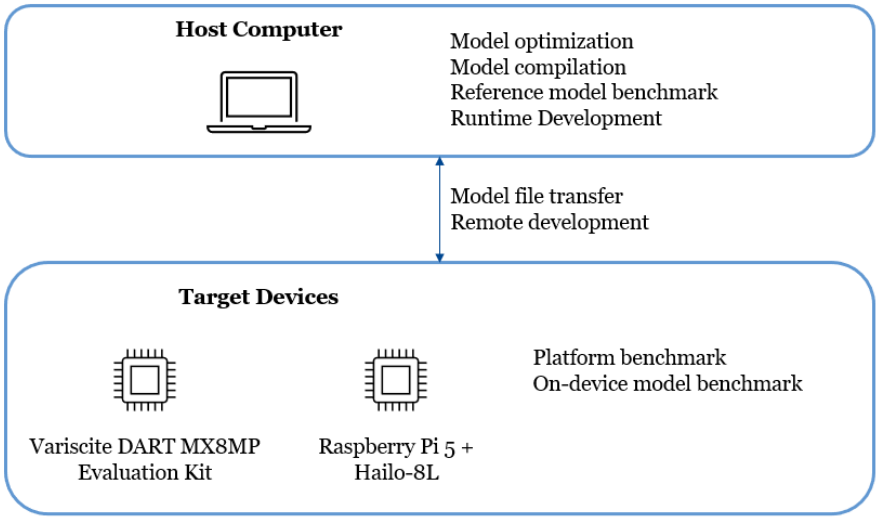


Figure 3.2: Hardware usage flowchart.

3.7 Hardware and Software Details

There are 3 hardware components used to perform the research. A personal computer (PC) and two edge computing platforms. The PC functions as the host computer, providing the environment for model optimization, compilation, reference model benchmarking, and runtime development. The target devices, which are the edge computing platforms, are used to execute the object detection tasks and on-device model benchmarking. This configuration allows for evaluating the efficiency and performance of the models in a realistic edge deployment scenario, as depicted in Figure 3.2.

3.7.1 Host Personal Computer Hardware and Software Used

The detail of the hardware specification used as the host personal computer is shown in Table 3.4, the detail of the software versions is detailed in Table 3.5.

3.7.2 NXP i.MX 8M Plus Hardware and Software Used

The detail of the hardware specification for the NXP I.MX8M Plus platform is shown in Table 3.6, the detail of the software versions is detailed in Table 3.7.

3.7.3 Hailo-8L Hardware and Software Used

The detail of the hardware specification for the Hailo-8L platform is shown in

Table 3.8, the detail of the software versions is detailed in Table 3.9.

Table 3.4: Host personal computer hardware specification

Specification	Value
Product Name	HP
Central Processing Unit	Intel Core i7-1165G7@2.80GHz
Graphics Processing Unit	Intel Iris Xe Graphics
Operating System	Ubuntu 22.04.5 LTS on WSL2
RAM	32 GB

Table 3.5: Host personal computer software version used

Software	Version
Python	3.11.12
Ultralytics	8.3.100
onnx2tf	1.27.0
LiteRT	1.2.0
Hailo Dataflow Compiler	4.21.0

Table 3.6: NXP i.MX 8M Plus platform hardware specification

Specification	Value
Evaluation Kit	Variscite DART-MX8M-Plus Evaluation Kit
Camera	Variscite VCAM-AR0821B
CPU Name	NXP i.MX 8M Plus
CPU Type	Quad Core Cortex-A53@1.8GHz
RAM	8 GB
NPU	2.3 TOPS
Operating System	Yocto Scarthgap

Table 3.7: NXP i.MX 8M Plus software version used

Software	Version
Python	3.12.6
LiteRT	1.1.2
OpenCV	4.10.0
GStreamer Core Library	1.24.7

Table 3.8: Hailo-8L platform hardware specification

Specification	Value
Single Board Computer	Raspberry Pi 5
Camera	Raspberry Pi Camera Module V3
CPU Name	Broadcom BCM2712
CPU Type	Quad Core Cortex-A76@2.4GHz
RAM	8 GB
NPU	Hailo-8L (Raspberry AI Hat)@13 TOPS
Operating System	Raspberry Pi OS 64-bit Bookworm

Table 3.9: Hailo-8L platform software version used

Software	Version
Python	3.11.2
HailoRT	4.21.0
OpenCV	4.11.0
GStreamer Core Library	1.22.0

Chapter 4
Implementation

This chapter outlines the implementation of the research methodology and details the practical steps taken throughout the project. It covers the configuration of the devices, initial deployment testing, model optimization, model benchmarking, deployment of models on edge devices, and platform-level benchmarking.

4.1 Device Configuration

This section describes the hardware configurations used for evaluating the object detection inference pipelines on two edge computing platforms: the NXP i.MX 8M Plus and the Hailo-8L. Each configuration includes a processing unit, camera, power supply, thermal management solution, and operating system. The following subsections provide details on each setup.

4.1.1 NXP i.MX 8M Plus

The target device for the NXP implementation is the Variscite DART-MX8M-Plus Evaluation Kit, which comprises the evaluation board and an integrated internal display. For thermal management, the system is equipped with a passive heatsink. The camera used in the setup is the Variscite VCAM-AR0821B, which incorporates the Basler daA3840-mc sensor. This sensor supports resolutions of up to 4K at 30 frames per second and is paired with an S-mount manual focus lens.

The system is powered using a Raspberry Pi USB-C power supply, which provides 5V at 3A. This meets the board's power requirements with the help of a USB-C Power Delivery (PD) board that interfaces with the barrel jack connector used by the evaluation board. To monitor energy consumption, an inline wattmeter is placed between the power supply and the board. The operating system used is the Yocto Scarthgap Recovery OS, which is tailored for embedded Linux development on the NXP platform. The NXP hardware setup is illustrated in Figure 4.1.

4.1.2 Hailo-8L

The Hailo-8L device configuration uses the Raspberry Pi 5 as the host computer. The setup employs the readily available Raspberry Pi AI Hat, which connects the Hailo-8L NPU via the PCIe socket on the Raspberry Pi 5. The camera used is the Raspberry Pi Camera Module v3, which supports resolutions up to Full HD at 50 frames per second and includes autofocus functionality.

Power is supplied by the official Raspberry Pi USB-C power supply, which provides 5V at 5A. Thermal management for the host processor is addressed through the use of the official active cooler module from Raspberry Pi, which assists in cooling the CPU. However, no additional thermal management is implemented for the Hailo-8L NPU module.

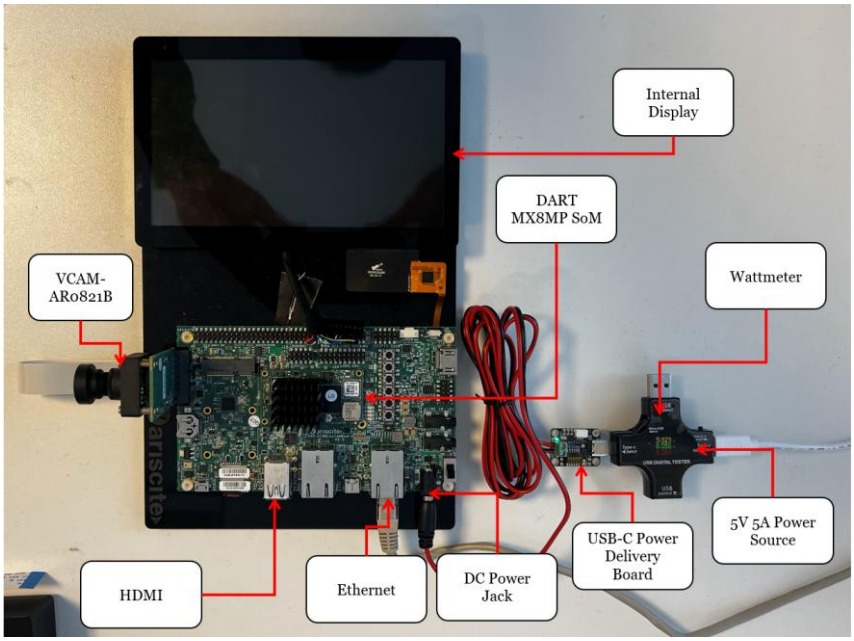


Figure 4.1: **NXP hardware setup implementation.**

The system runs Raspberry Pi OS, a Debian-based Linux distribution officially maintained by the Raspberry Pi Foundation. It is optimized for ARM architecture and provides a lightweight yet functional desktop environment suitable for development and deployment on Raspberry Pi hardware. The OS includes support for a wide range of peripherals, libraries, and development tools, making it well-suited for prototyping and running machine learning inference tasks with minimal overhead. An overview of the Hailo-8L hardware setup is presented in Figure 4.2.

4.2 Basic Deployment Testing

The basic deployment testing was carried out using demo projects provided by the respective device manufacturers. A common feature across both demo implementations is the use of custom GStreamer plug-ins. GStreamer is a multimedia framework used to construct streaming media applications [93]. It operates using a pipeline-based architecture, which is particularly effective for managing the flow of video and inference data. In this context, GStreamer facilitates the handling of camera input, preprocessing, inference execution, and display output in a modular and efficient manner.

4.2.1 NXP i.MX 8M Plus Demo Project

The object detection demo provided by NXP for the i.MX 8M Plus, which incorporates YOLOv4-tiny and NNStreamer, follows a somewhat fragmented inference pipeline architecture. The pipeline is initiated using a Bash script to launch the NNStreamer components, while custom Python scripts are employed externally to perform YOLOv4 post-processing. This separation across scripting environments introduces integration challenges, particularly in terms of error tracing, pipeline customization, and deployment in tightly integrated systems. The need to manage dependencies and ensure smooth data exchange between distinct execution contexts adds further complexity.

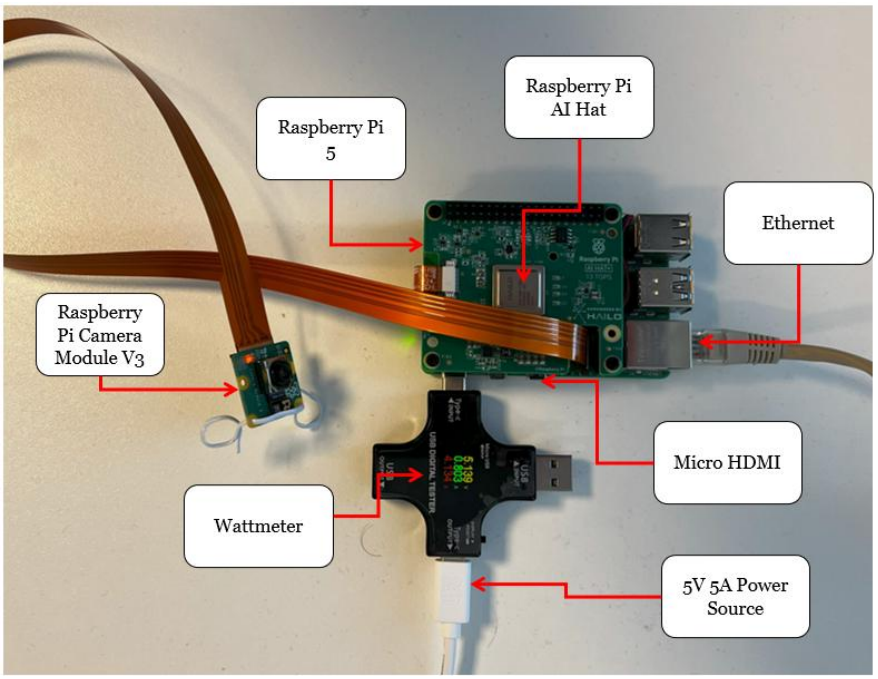


Figure 4.2: **Hailo hardware setup implementation.**

Attempts to consolidate the pipeline into a single Python script, with the goal of improving integration and manageability, have been unsuccessful. Specifically, integrating the custom Python post-processing directly within the NNStreamer pipeline leads to issues caused by the Global Interpreter Lock (GIL), which prevents concurrent execution and results in runtime conflicts. This limitation further constrains the flexibility and scalability of the provided implementation.

4.2.2 Hailo-8L Demo Project

The Hailo-8L object detection demo leverages the proprietary Hailo Apps Infrastructure, a framework that intricately integrates the GStreamer multimedia pipeline directly within Python scripts, working in tandem with the HailoRT inferencing API. This demo showcases the Hailo-8L's capabilities using the YOLOv8n model as a practical example. However, a significant challenge with the Hailo Apps Infrastructure lies in the abstraction of numerous critical variables within a deep hierarchy of Python scripts, making them inaccessible through the provided command-line interface (CLI). This design necessitates direct modification of the underlying source code to alter fundamental inferencing parameters, such as the model's expected input shape and the camera resolution. This lack of direct CLI control over key variables introduces friction and complexity when users need to adapt the demo to different models or hardware configurations, requiring a deeper understanding of the Hailo Apps Infrastructure's internal structure and Python codebase.

4.2.3 Common Problem in Available Demo Project

Based on the evaluation of both demonstration projects, the device-side inference implementation is developed from scratch. This approach provides greater flexibility for modifying parameters, integrating application-specific business logic, and designing the inference pipeline to be as streamlined and modular as possible. The custom implementation ensures consistency across platforms while allowing tailored optimizations for each target device.

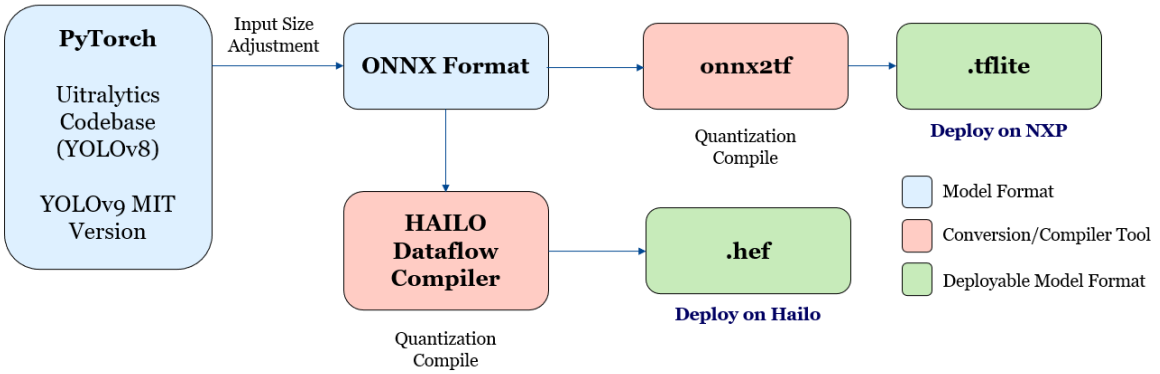


Figure 4.3: **Pre-trained model optimization workflow.**

4.3 Pre-Trained Model Optimization Implementation

The implementation of pre-trained models leverages the existing Ultralytics codebase for YOLOv8 and the MIT-licensed YOLOv9 codebase to convert the PyTorch-formatted model weights into ONNX format. The workflow outlining the transformation of model formats for each platform is illustrated in Figure 4.3.

4.3.1 Input Shape Adjustment Implementation

Both PyTorch and ONNX formats support dynamic input shapes, while formats such as TFLite and HEF (Hailo Executable Format) require static input dimensions. This constraint highlights the importance of input shape adjustment as a model optimization step to reduce computational demand. PyTorch is primarily used for model training and architectural modifications, whereas ONNX serves as an open and interoperable format for representing machine learning models. Due to its flexibility, ONNX is widely adopted as an intermediate representation and is supported by tools such as the Hailo DFC and *onnx2tf*, which are used to convert PyTorch models into HEF and TFLite formats, respectively.

To apply static input sizing, the PyTorch model is exported to ONNX format using a dummy input tensor of the desired dimensions. This export process effectively converts a model with dynamic input shapes into one with fixed input dimensions, compatible with downstream compilers. An example of exporting a PyTorch model to ONNX with a specified static input shape is presented in Listing 4.1.

Listing 4.1: **Input shape adjustment sample code.**

```
with open("../yolo/config/model/v9-s.yaml", "r") as f:
    config = yaml.safe_load(f)

if 'model' in config and 'auxiliary' in config['model']:
    # Change this line depending on whether you want to keep or remove auxiliary content
    config['model']['auxiliary'] = {}  #
import os
import sys

project_root = "/home/erona/YOLOv9MIT/YOLO"
sys.path.append(project_root)

from yolo.model.yolo import create_model
from yolo.tools.dataset_preparation import prepare_weight
```

```
15.
16. model = create_model(model_conf, class_num=80, weight_path=True, export_mode=False)
17.
18. import torch
19. # Use a dummy input with new dimensions
20. dummy_input = torch.randn(1, 3, 320, 320)
21.
22. # Export to ONNX
23. torch.onnx.export(model,
24.                    dummy_input,
25.                    "yolov9_320.onnx",
26.                    opset_version=11,
27.                    input_names=['input'],
28.                    output_names=['output'])
```

4.3.2 Quantization and Compilation Implementation

The quantization process follows a similar procedure for both TFLite and Hailo toolchains. Initially, a calibration dataset is prepared by converting the first 500 images from the COCO dataset into NumPy array format. To ensure that the calibration accurately reflects real-world inference conditions, each image in the calibration dataset undergoes the same preprocessing steps as those used during inference, as described later in Section 4.5.1. Furthermore, it is essential that the input dimensions of the preprocessed images match the static input shape defined during the model export process. The preprocessing and calibration dataset preparation steps are shown in Listing 4.2.

Listing 4.2: Calibration data preparation sample code.

```
 1. import numpy as np
 2. from PIL import Image
 3. import os
 4.
 5. # Paths to directories and files
 6. image_dir = './input/coco500'
 7. output_dir = './output/'
 8. os.makedirs(output_dir, exist_ok=True)  # Create the directory if it doesn't exist
 9.
10. # File paths for saving calibration data
11. processed_data_path = os.path.join(output_dir, "processed_calibration_data_640.npy")
12.
13. # Initialize an empty list for calibration data
14. calib_data = []
15.
16. # Process all image files in the directory
17. for img_name in os.listdir(image_dir):
18.     img_path = os.path.join(image_dir, img_name)
19.     if img_name.lower().endswith(('.jpg', '.jpeg', '.png')):
20.         img = Image.open(img_path).convert("RGB")  # Resize to desired dimensions
21.         img = np.array(img)
22.         img = letterbox(img, (640,640))
23.         img_array = np.array(img) / 255.0  # Normalize to [0, 1]
24.         calib_data.append(img_array)
25.
26. print(type(calib_data))
27. # Convert the calibration data to a NumPy array
28. calib_data = np.array(calib_data)
29.
30. # Scale the normalized data back to [0, 255]
31. processed_calibration_data = calib_data * 255.0
32.
33. # Save the processed calibration data
34. np.save(processed_data_path, processed_calibration_data)
```

```
print(f"Processed calibration dataset saved with shape: {processed_calibration_data.shape} to {processed_data_path}")
```

4.3.2.1 TFLite Quantization and Compilation

Quantization for TFLite is performed during the model conversion stage. The ONNX model is converted to the TFLite format using the *onnx2tf* tool. A critical parameter in this process is *output_integer_quantized_tflite*, which enables the generation of a fully 8-bit integer quantized model. To apply post-training quantization effectively, the previously prepared calibration dataset is specified using the *custom_input_op_name_np_data_path* parameter, allowing the tool to utilize representative input data during the quantization process.

The code snippet demonstrating the quantization and conversion process from ONNX to TFLite format is presented in Listing 4.3.

Listing 4.3: TFLite conversion and quantization sample code.

```
import onnx2tf
import numpy as np
from pathlib import Path

model_name= "v9-s_640"
save_path = Path(str(model_name))
save_path.mkdir()

np_data = [["input", "./calib_npy/processed_calibration_data_640.npy", [[[[0, 0, 0]]]], [[[[255, 255, 255]]]]]]

keras_model = onnx2tf.convert(
            input_onnx_file_path=f"./onnx/{model_name}.onnx",
            output_folder_path=save_path,
            not_use_onnxsim=True,
            verbosity="error",
            output_integer_quantized_tflite=True,
            quant_type="per-tensor",
            custom_input_op_name_np_data_path=np_data
        )
```

4.3.2.2 Hailo Dataflow Compiler Quantization and Compilation

Since Hailo chips are specifically designed to accelerate neural network operations, such as convolutions and activations, the model must define an appropriate end node prior to certain non-linear operations, including slicing and transposing. This requirement influences how models are prepared for deployment. In the case of YOLOv8, this constraint poses minimal difficulty because the model includes a convenient NMS post-processing block. The Hailo toolchain can map this post-processing stage to the CPU, allowing for seamless integration during deployment. However, for YOLOv9, due to the differences between the MIT implementation and Ultralytics' design, additional custom post-processing is required, as it is not natively compatible with Hailo's automated mapping process.

The Hailo Dataflow Compiler utilizes an intermediate file format called the Hailo Archive (HAR), which serves as a checkpoint that captures the model's state between the initial ONNX parsing and the final compilation. The quantization process is initiated using the *optimize* command, which requires the calibration dataset in NumPy format to adapt the model for 8-bit inference. After optimization, the final hardware-executable .hef file is generated using the *compile* command. Figure 4.4 illustrates the workflow of model optimization and compilation using the Hailo DFC.

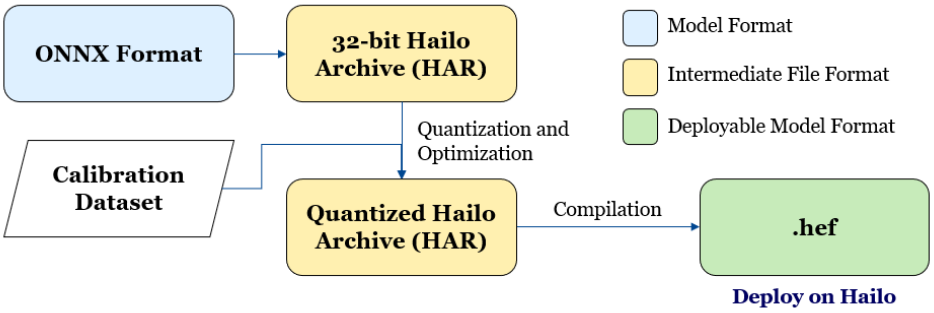


Figure 4.4: **Hailo Dataflow Compiler model optimization and compilation workflow.**

4.4 Model Benchmark Implementation

This section describes the benchmark implementations used to evaluate the object detection models. The benchmarks are divided into two categories: reference benchmarks, which provide a baseline performance evaluation using pre-trained models, and on-device benchmarks, which assess the performance of optimized models deployed on the target hardware. The following subsections detail the methods for both types of benchmark implementation.

4.4.1 Reference Benchmark Implementation

The reference benchmarks were implemented using the built-in validation functions provided by the respective model repositories, namely the Ultralytics YOLOv8 repository and the MIT implementation of YOLOv9. These validation routines are designed to assess object detection performance using standardized evaluation metrics, which ensures consistency and reproducibility across different model architectures.

The evaluation was conducted using the COCO validation dataset from the 2017 release (COCO val2017), which contains 5,000 annotated images. This dataset is widely recognized as a standard benchmark in the field of object detection due to its diverse scenes and comprehensive ground-truth annotations.

Both repositories support integration with the *pycocotools* library. This library enables the calculation of key evaluation metrics, including mAP and AR across multiple IoU thresholds. These metrics provide a detailed assessment of model performance and serve as a baseline for comparison with optimized models deployed on edge computing platforms.

4.4.2 On-device Benchmark Implementation

Since both the TFLite and HEF formats are executable only on their respective hardware platforms, benchmarking must be conducted directly on the target devices to evaluate the performance of the optimized models. This process involves running inference on the COCO validation dataset from the 2017 release, and capturing the output predictions in a JSON file. The resulting file is then used in conjunction with the pycocotools library to compute evaluation metrics such as mAP and AR, providing quantitative insights into model accuracy under real deployment conditions.

Figure 4.5: **General inference implementation pipeline diagram.**

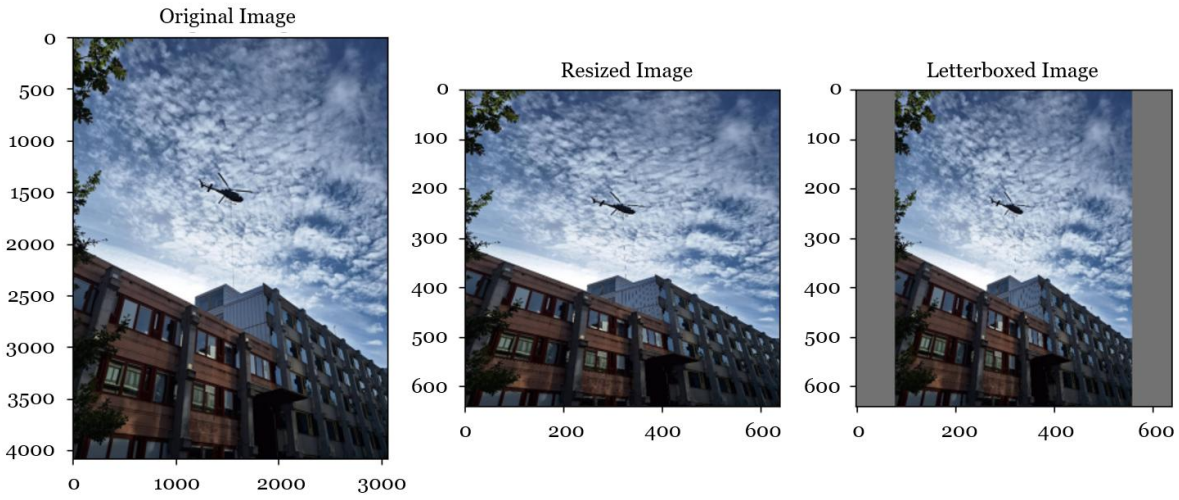


Figure 4.6: **Resize and letterboxing operation for image scaling.**

4.5 Device Inference Implementation

The primary objective of the inference implementation is to enable real-time image capture from a camera, perform object detection inference, and display the results with minimal latency. This process follows a structured pipeline, as illustrated in Figure 4.5. First, image data are captured from the camera and subsequently pre-processed using the OpenCV library to match the input requirements of the model. The pre-processed frames are then passed to the machine learning runtime for inference. The raw outputs from the inference process undergo post-processing to extract meaningful detection results, such as bounding boxes and class labels. Finally, the processed frame is rendered and displayed to the user in real time.

4.5.1 Pre-Processing Implementation

The preprocessing pipeline for both YOLOv8 and YOLOv9 follows an identical procedure across all target devices. Initially, the captured frame, which is represented in INT8 data type, is converted from its original color format to the RGB (Red Green Blue) color space. This conversion standardizes the color representation required by the models.

Following this, a letterboxing technique is applied to resize the RGB image so that it matches the input dimensions expected by the model. Letterboxing preserves the original aspect ratio of the frame by adding padding, typically in the form of neutral gray bars, to the top and bottom or to the sides of the image. This approach prevents distortion that could otherwise occur during the resizing process. An illustration of this step is provided in Figure 4.6.

After the letterboxing process is complete, the image conforms to the required input shape and is ready to be used as input for the YOLO model during inference.

4.5.2 YOLOv8 Post-Processing Implementation

The output of the COCO-pretrained YOLOv8 model is a tensor with the shape (1, 84, 8400). This indicates the presence of 8,400 potential detection points. Each detection point contains 84 values, which include the confidence scores for each of the 80 COCO classes and four values that represent the bounding box coordinates: minimum x, minimum y, width, and height. The number of detection points can vary depending on the input image dimensions. This variation is made possible by the Feature Pyramid Network (FPN) within the neck of the YOLOv8 architecture. The FPN generates feature maps at three different spatial resolutions, which allow the model to detect objects at multiple scales, including large, medium, and small regions.

In the TFLite implementation, the raw output tensor, which typically retains the shape (1, 84, 8400) for a 640 by 640 input resolution, must undergo additional post-processing to produce meaningful detection results. This process includes filtering out low-confidence predictions and applying NMS using the OpenCV library. The NMS procedure uses the detection output along with user-defined thresholds for confidence and IoU It selects the bounding box with the highest confidence and removes any overlapping boxes that exceed the IoU threshold. This process eliminates redundant detections of the same object and results in a refined set of bounding boxes, each with an associated class label and confidence score. These outputs are suitable for visualization or further analytical use.

In the Hailo implementation, the NMS operation is integrated into the model during the compilation process using the Hailo Dataflow Compiler. Although the post-processing is embedded, the parameters for confidence and IoU thresholds remain adjustable. This approach provides a streamlined inference pipeline on the hardware while maintaining flexibility to adapt the detection sensitivity and filtering behavior to suit different application requirements.

4.5.3 YOLOv9 Post-Processing Implementation

The output of the MIT-licensed implementation of YOLOv9 comprises nine distinct tensors. These outputs are organized into three sets, each corresponding to a different detection scale: large, medium, and small. For each scale, the model produces three outputs. The first contains the class confidence scores for each of the 80 COCO classes. The second includes box features, which are learned intermediate representations not directly used during post-processing. The third output provides the bounding box coordinates required for object localization. This multi-scale output structure enables the model to detect objects of varying sizes more effectively.

Post-processing in YOLOv9 begins by retrieving the nine output tensors, which contain class scores, box features, and bounding box adjustments across the three detection scales. Since the model has been quantized, the output tensors are first dequantized into floating-point representations. The outputs are then grouped by scale and reshaped to associate spatial positions with their corresponding class scores and feature information. After reshaping, the predictions from all three scales are concatenated into unified tensors for class probabilities, anchor-related features, and bounding box deltas.

The next step involves applying predefined anchor boxes and scale factors that are specific to the input resolution and stride of each detection layer. The predicted bounding box adjustments are scaled accordingly and used to modify the anchor boxes, resulting in a set of bounding box predictions. These boxes are filtered using a confidence score threshold to eliminate low-probability detections. Finally, the remaining bounding boxes are passed through the Non-Maximum Suppression (NMS) algorithm, implemented using the OpenCV library, to suppress overlapping detections and retain only the most confident and non-redundant predictions.

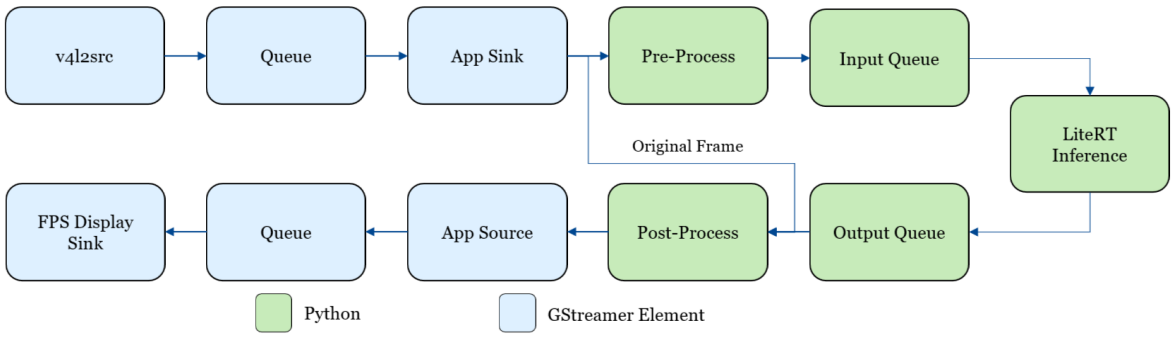


Figure 4.7: **NXP i.MX 8M Plus inference implementation pipeline.**

4.5.4 NXP i.MX 8M Plus Inference Implementation

The implementation on the NXP i.MX 8M Plus platform employs a processing pipeline that integrates GStreamer and OpenCV, as illustrated in Figure 4.7. Camera frames are captured using the Video4Linux2 (V4L2) source element within the GStreamer framework. These frames are then passed to OpenCV through the appsink element, where they undergo preprocessing before being forwarded to the Lite Runtime (LiteRT) inference engine.

The inference process leverages TensorFlow Lite's LiteRT, extended through an external delegate to enable efficient execution on the platform's integrated NPU. Specifically, the *libvx_delegate* is utilized to facilitate this hardware acceleration on the NXP i.MX 8M Plus. This delegate functions as a bridge, translating TensorFlow Lite operations into an OpenVX computational graph that is executed directly on the NPU. The delegate is configured and initialized during the setup of the TensorFlow Lite interpreter, as demonstrated in Listing 4.4.

Listing 4.4: **LiteRT intrepeter initialization with external delegate sample code**

```
delegate_lib = "/usr/lib/libvx_delegate.so"

try:
    # Create the delegate
    delegate = litert.load_delegate(delegate_lib)

    # Initialize the interpreter with the delegate
    self.interpreter = litert.Interpreter(
    model_path=path,
    experimental_delegates=[delegate]
    )
    self.interpreter.allocate_tensors()

    print("Successfully loaded NPU delegate")
```

Upon completion of inference, the output is post-processed alongside the original frame, which is concurrently preserved via a parallel data path from the appsink. The post-processed results are then superimposed on the original frame to generate the final output. This output is subsequently returned to a GStreamer pipeline that incorporates an FPS display sink, which calculates and renders the average FPS performance at the display stage of the inference pipeline.

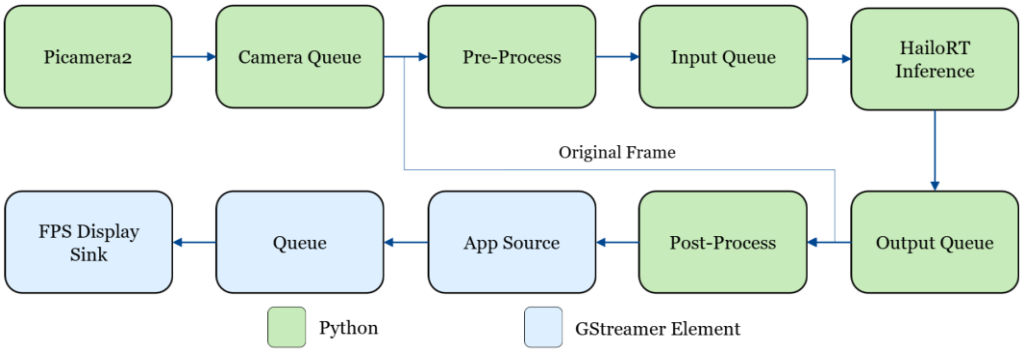


Figure 4.8: **Hailo-8L inference implementation pipeline.**

4.5.5 Hailo-8L Inference Implementation

Similar to the NXP implementation, the Hailo-based deployment utilizes a combined GStreamer and OpenCV pipeline, as illustrated in Figure 4.8. Camera control is managed through the PiCamera2 Python library, which provides access to settings such as frame rate, autofocus, and resolution. The captured frames are then forwarded to a queue, where they are consumed by a dedicated preprocessing thread.

This preprocessing thread applies letterboxing to each frame to ensure the image conforms to the input dimensions required by the model, while maintaining the original aspect ratio. The processed frames are subsequently placed into a model input queue for inference.

Inference is performed using the HailoRT (Hailo Runtime), which supports an asynchronous inference API. This functionality is demonstrated in Listing 4.5. Once inference is complete, the results are transferred to an output queue.

Listing 4.5: **HailoRT asynchronous inference API sample code.**

```
def run(self) -> None:
        with self.infer_model.configure() as configured_infer_model:
            while True:
                batch_data = self.input_queue.get(timeout=1)
                if batch_data is None:
                    break  # Sentinel value to stop the inference loop

                if self.send_original_frame:
                    original_batch, preprocessed_batch = batch_data
                else:
                    preprocessed_batch = batch_data

                bindings_list = []
                for frame in preprocessed_batch:
                    bindings = self._create_bindings(configured_infer_model)
                    bindings.input().set_buffer(np.array(frame))
                    bindings_list.append(bindings)

                configured_infer_model.wait_for_async_ready(timeout_ms=10000)
                job = configured_infer_model.run_async(
                    bindings_list, partial(
                        self.callback,
                        input_batch=original_batch if self.send_original_frame else
preprocessed_batch,
                        bindings_list=bindings_list
                    )
                )
            job.wait(10000)  # Wait for the last job
```

A post-processing thread then processes these inference results in conjunction with the corresponding original frames retrieved from the camera queue. The final annotated frames are passed into a GStreamer pipeline, where they are displayed using the FPS display sink to visualize the output and measure the average display frame rate.

4.6 Device Benchmark

This section outlines the benchmark methods used to evaluate the performance of the inference pipelines on both device configurations.

4.6.1 Throughput Measurement

As discussed in Section 0, throughput measurement is implemented using the fpsdisplaysink element within the GStreamer pipeline. This component leverages GStreamer's internal timing mechanisms to monitor the number of frames rendered per second. By aggregating this data over time, it provides an estimate of the average FPS, thereby offering a practical measure of the system's real-time inference and display performance.

4.6.2 Inference Latency Measurement

Inference latency is measured using benchmark scripts executed on each target device. The approach involves the use of Python's time library to record timestamps immediately before and immediately after the inference function is called. The difference between these two timestamps provides the latency value for a single inference cycle. This method offers a straightforward means of evaluating the execution time required by the model on the hardware. The implementation of this measurement for the NXP platform is provided in Listing 4.6, and the corresponding implementation for the Hailo platform is shown in Listing 4.7.

Listing 4.6: NXP inference latency measurement implementation sample code.

```
inference_tic = time.time()
# set frame as input tensors
self.interpreter.set_tensor(self.input_details[0]['index'], x)

# perform inference
self.interpreter.invoke()

self.time_metrics["inference"] += time.time() - inference_tic
```

Listing 4.7: Hailo inference latency measurement implementation sample code.

```
inference_tic = time.time()
with hpf.InferVStreams(self.network_group, self.input_vstreams_params,
self.output_vstreams_params) as infer_pipeline:
      if self.args.model_arch == "v8":
      infer_pipeline.set_nms_iou_threshold(self.iou)
      infer_pipeline.set_nms_score_threshold(self.conf_all)
      input_data = {self.input_vstream_info.name: np.expand_dims(input_data, axis=0)}
      results = infer_pipeline.infer(input_data)
      result = {}
      for output_stream in self.output_vstream_info:
                result[output_stream.name] = results[output_stream.name]
                if len(self.output_vstream_info) > 1:
                 for key, value in result.items():
                          result[key] = (value.astype(np.float32) -
self.quantization_dict[key][1]) * self.quantization_dict[key][0]

```

```
15. self.time_metrics["inference"] += time.time() - inference_tic
```

4.6.3 End to End Latency Measurement

End-to-end latency is measured during real-time camera inference by comparing frame timestamps. Each timestamp is recorded at the moment the frame is initially captured by the script and then compared to the timestamp recorded when the same frame is rendered through the display sink of the GStreamer pipeline. This method captures the total processing delay from image acquisition to visual output. The implementation details are demonstrated in Listing 4.8.

Listing 4.8: End-to-end latency measurement implementation sample code.

```
 1.  def sink_probe(self, pad, info):
 2.      buffer = info.get_buffer()
 3.      if not buffer or buffer.pts == Gst.CLOCK_TIME_NONE:
 4.          print("No valid timestamp")
 5.          return Gst.PadProbeReturn.OK
 6.
 7.      now = Gst.util_get_timestamp()
 8.      latency_ns = now - buffer.pts
 9.      latency_ms = latency_ns / 1e6
10.
11.      print(f"[System Latency] {latency_ms:.2f} ms")
12.      return Gst.PadProbeReturn.OK
```

4.6.4 Power Usage Measurement

Power consumption is measured using a wattmeter connected in-line between the power supply and the edge computing platforms. This setup enables measurement of the total power usage of the entire board during real-time inference. To isolate the power consumption attributable primarily to the inference process, steps are taken to minimize unrelated power draw. Specifically, for the Variscite DART-MX8M-Plus Evaluation Kit, the internal display is disabled to reduce its impact on power measurements. Instead, an external monitor is used via the HDMI interface for all display outputs.

4.6.5 CPU Utilization, Temperature, and RAM Measurement

CPU utilization, temperature, and memory usage are monitored using software-level APIs that provide access to relevant system metrics. These monitoring functions are integrated into the camera inference script to facilitate real-time data collection during inference execution. The recorded metrics are subsequently exported to a CSV file for post-processing and analysis. The implementation specific to the NXP platform is presented in Listing 4.9, while the corresponding implementation for the Hailo platform is shown in Listing 4.10. It is important to note that the Hailo implementation additionally includes measurement of the external NPU temperature, a parameter that is not applicable to the NXP platform due to the absence of an external NPU.

Listing 4.9: NXP system resource measurements implementation sample code.

```
 1. from time import sleep, strftime, time
 2. import psutil
 3.
 4. def temp_ram_measurement(file, target_name, duration, time_interval):
 5.     for proc in psutil.process_iter(['pid', 'name']):
 6.         if proc.info['name'] == target_name:
 7.             pid = proc.info['pid']
 8.
 9.     start_time = time()
10.
```

```
    with open(file, "a", buffering=1) as log:
        log.write(target_name  + "\n")
        while (time() - start_time < duration):
            mem = psutil.virtual_memory()

            with open("/sys/class/thermal/thermal_zone0/temp", "r") as f:
                temp_str = f.read().strip()
                temp_soc = int(temp_str) / 1000.0

            used_ram = mem.used / (1024 ** 2)

            process = psutil.Process(pid).memory_info().rss / (1024 ** 2)

            cpu_util = psutil.cpu_percent()

            log.write("{0},{1:.2f},{2:.2f},{3:.2f},{4}\n".format(strftime("%Y-%m-%d
%H:%M:%S"),(temp_soc), (used_ram), (process), (cpu_util)))

            log.flush()
            sleep(time_interval)

    print(f"Measurement saved to {file}")
```

Listing 4.10: Hailo system resource measurements implementation sample code.

```
from gpiozero import CPUTemperature
from time import sleep, strftime, time
from hailo_platform import Device
import psutil

def temp_ram_measurement(file, target_name, duration, time_interval):
    for proc in psutil.process_iter(['pid', 'name']):
        if proc.info['name'] == target_name:
            pid = proc.info['pid']

    cpu = CPUTemperature()

    device_infos = Device.scan()
    targets = [Device(di) for di in device_infos]

    start_time = time()

    with open(file, "a", buffering=1) as log:
        log.write(target_name  + "\n")
        while (time() - start_time < duration):
            mem = psutil.virtual_memory()
            temp_cpu = cpu.temperature
            for target in targets:
                temp_hailo = target.control.get_chip_temperature().ts0_temperature

            used_ram = mem.used / (1024 ** 2)

            process = psutil.Process(pid).memory_info().rss / (1024 ** 2)
            cpu_util = psutil.cpu_percent()

            log.write("{0},{1},{2:.1f},{3:.2f},{4:.2f}, {5:.2f}\n".format(strftime("%Y-%m-
%d %H:%M:%S"),str(temp_cpu), (temp_hailo), (used_ram), (process),(cpu_util)))

            log.flush()
            sleep(time_interval)

    print(f"Measurement saved to {file}")
```

Chapter 5
Result and Analysis

This chapter presents the results obtained from the research and provides a detailed analysis of key insights derived from the collected data. The discussion is organized into four main sections: Real-Time Object Detection Performance Analysis, Power Usage and Thermal Performance Analysis, Resource Utilization Analysis, Model Optimization Impact Analysis, and Cost Analysis.

The complete performance benchmarking results for the NXP and Hailo platforms are presented in Appendix A and Appendix B, respectively. Detailed model-specific performance metrics are provided in Appendix C for the NXP platform and Appendix D for the Hailo platform.

5.1 Real-Time Object Detection Performance Analysis

Figure 5.1 and Figure 5.2 visually present a direct comparison of the mean Average Precision (mAP@0.5:0.95) achieved by the various YOLO models against their corresponding pipeline throughput (in Frames Per Second) when executed on the NXP i.MX 8M Plus and Hailo-8L platforms, respectively, under a 1920x1080 (1080p) input stream resolution. As predicted by their respective hardware specifications, the Hailo-8L platform consistently demonstrates a clear advantage in terms of inference throughput across the tested model architectures and input shapes. Notably, even the most computationally intensive models maintained a throughput exceeding 20 FPS on the Hailo platform at 1080p.

Conversely, the NXP i.MX 8M Plus platform exhibited a more constrained throughput performance. Only the YOLOv8n and YOLOv8s configured with the smallest input dimensions (320x320 pixels) were able to break through the 20 FPS mark. The majority of models on the NXP platform operated within a throughput range of 10 to 20 FPS at 1080p, indicating a potential bottleneck for real-time applications demanding higher frame rates. Strikingly, the larger and more complex YOLOv9m and YOLOv8m models, when processing 640x640 input, experienced a significant drop in throughput, falling below 5 FPS on the NXP platform. This stark contrast underscores the performance disparity between the two platforms under high-resolution, computationally intensive workloads.

For the NXP platform, the YOLOv8m at 320x320 input appears to offer a practical balance between detection accuracy and throughput. On the other hand, the Hailo-8L platform is capable of leveraging the full complexity of models like YOLOv9m while maintaining real-time performance, making it a more robust choice for high-accuracy object detection at higher frame rates.

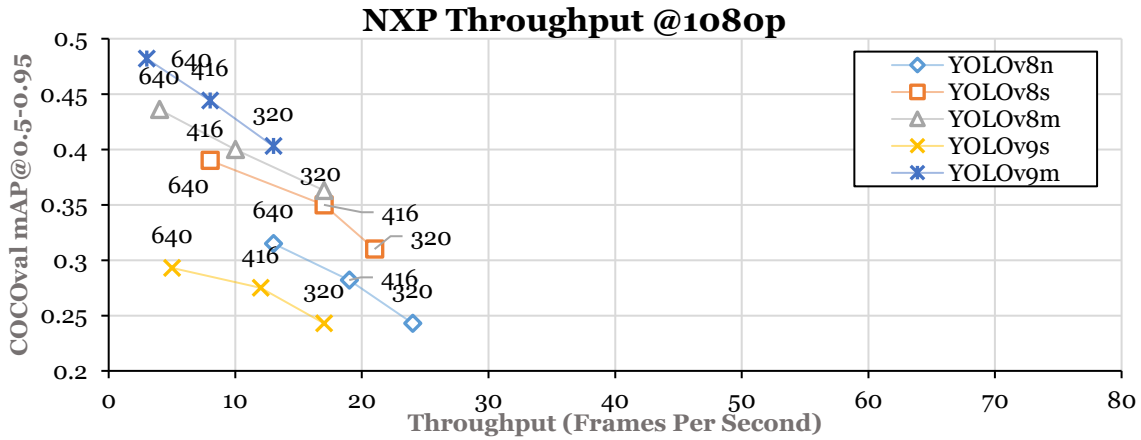


Figure 5.1: **Throughput versus COCOval mAP@0.5:0.95 for the NXP platform using a 1920×1080 camera stream resolution.** ***Note: Data labels indicate the model input shape (e.g., 640 denotes a 640×640 input dimension).***

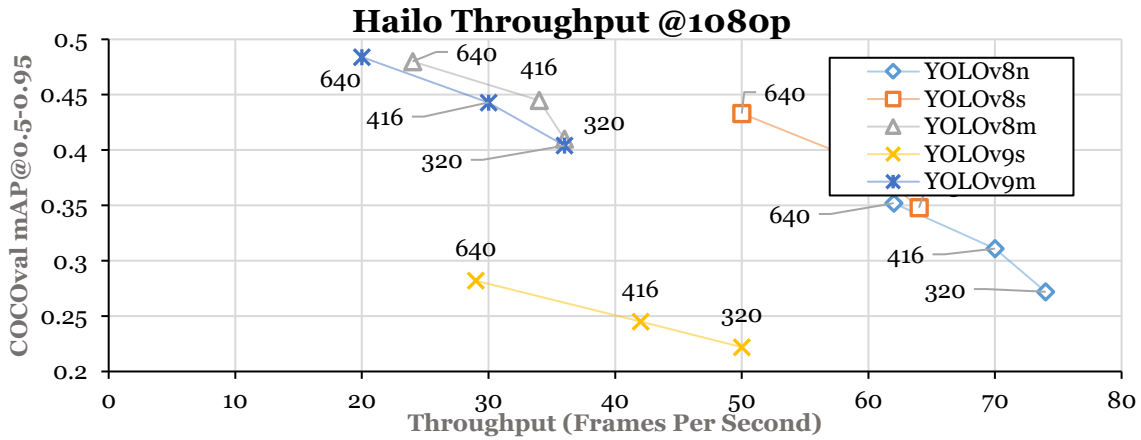


Figure 5.2: **Throughput versus COCOval mAP@0.5:0.95 for the Hailo platform using a 1920×1080 camera stream resolution.**

Further examination of the experimental results under a reduced input stream resolution of 1280x720 (720p), as depicted in Figure 5.3 and Figure 5.4, reveals a general trend of throughput improvement for most models on both platforms. The lower resolution naturally alleviates the computational burden, leading to faster processing times. However, the extent of these improvements is not uniform across all models. A discernible pattern emerges wherein the more complex models, characterized by a higher number of parameters and intricate architectures (such as YOLOv8m and YOLOv9m), exhibit diminishing returns in throughput gain from the reduction in input stream resolution. This suggests that the architectural intricacies and inherent computational demands of these larger models become a more significant limiting factor than the initial input stream resolution, impacting the scalability of performance gains through simple resolution adjustments on both the NXP and Hailo platforms.

Under the 720p stream, the NXP platform achieves up to 30 FPS when running YOLOv8n and YOLOv8s models with a 320×320 input shape, enabling improved real-time performance. Additionally, reducing the stream resolution benefits the YOLOv8m model at a 416×416 input shape, which approaches 20 FPS, making it feasible for use cases such as pallet and package detection, as well as surveillance in industrial floor for safety zone enforcement.

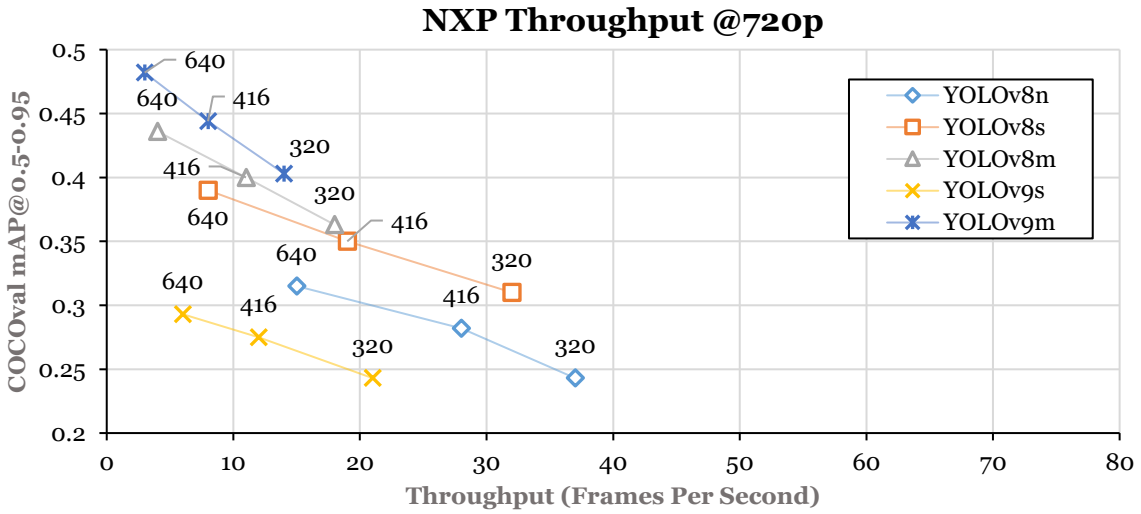


Figure 5.3: **Throughput versus COCOval mAP@0.5:0.95 for the NXP platform using a 1280×720 camera stream resolution.**

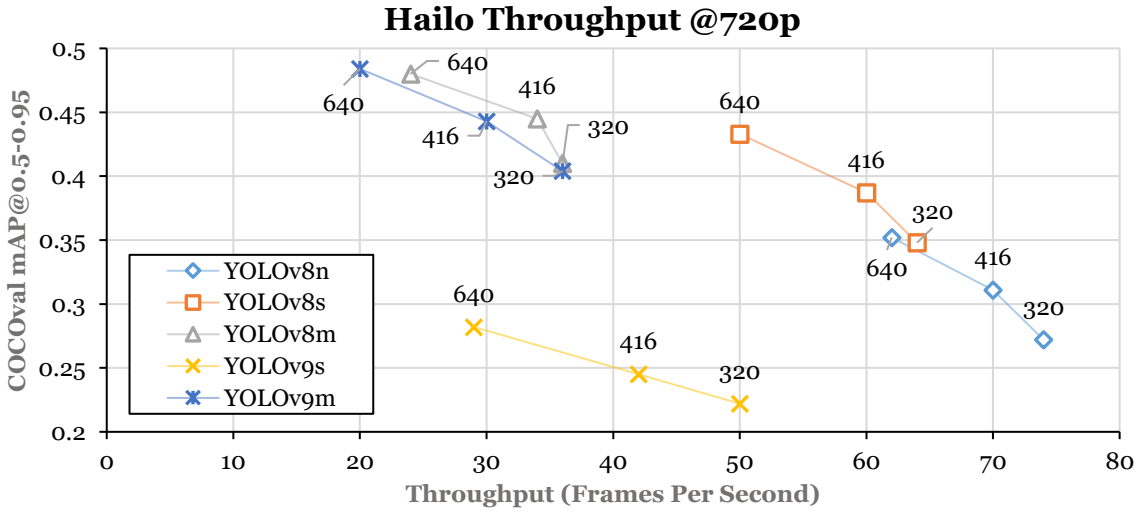


Figure 5.4: **Throughput versus COCOval mAP@0.5:0.95 for the Hailo platform using a 1280×720 camera stream resolution.**

The inference latency for all evaluated models on both platforms is presented in Figure 5.5 and Figure 5.6. On the Hailo platform, only the YOLOv9m model exceeds an inference duration of 50 milliseconds, while all other models demonstrate consistently fast performance, with inference times ranging between 8 and 50 milliseconds. In contrast, the NXP platform exhibits significantly slower inference speeds. Only five models achieve inference times below 50 milliseconds, with the slowest recorded latency reaching 244 milliseconds for the YOLOv9m model with a 640×640 input size. These results further emphasize the substantial performance disparity between the two platforms.

Figure 5.7 and Figure 5.8 illustrate the end-to-end latency of the object detection pipeline for both platforms when streaming at 1080p and 720p, respectively. A notable distinction between the two platforms lies in their end-to-end latency performance. On the NXP platform, even the minimum observed latency does not fall below 100 milliseconds. As model complexity increases, either due to more intricate architectures or larger input resolutions, the end-to-end latency rises accordingly. In some cases, such as running the YOLOv9m model with a 640×640 input shape, latency reaches up to 900 milliseconds.

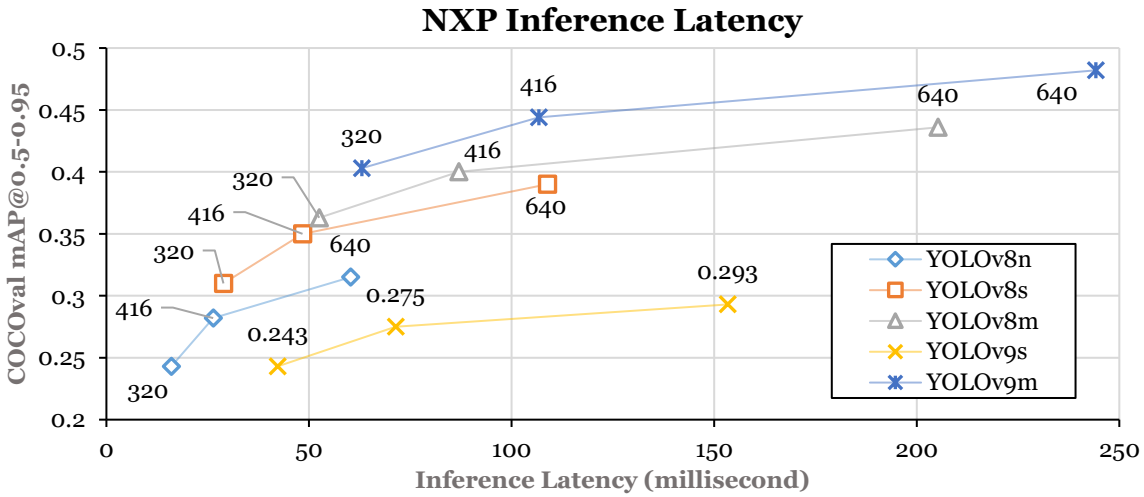


Figure 5.5: **Inference latency versus COCOval mAP@0.5:0.95 for the NXP platform.**

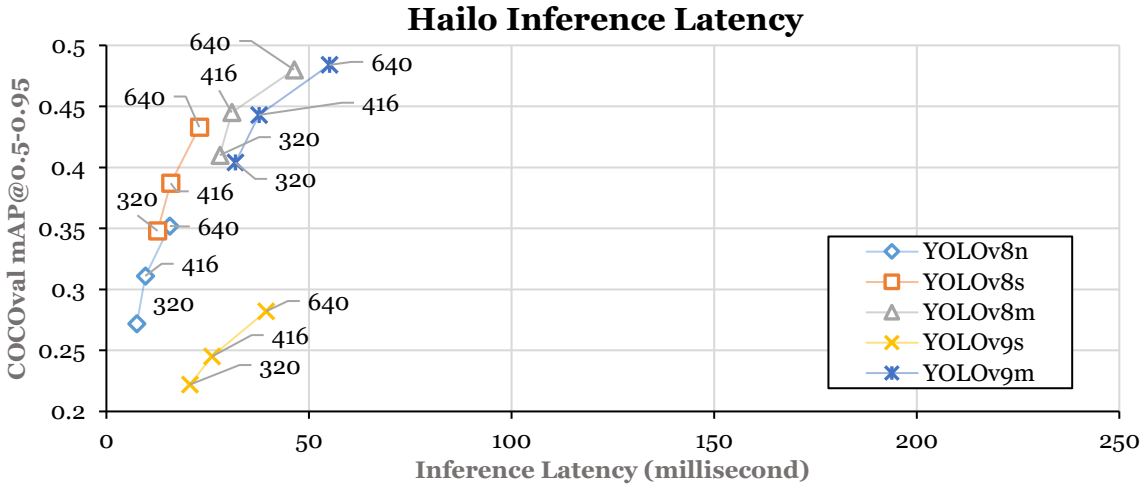


Figure 5.6: **Inference latency versus COCOval mAP@0.5:0.95 for the Hailo platform.**

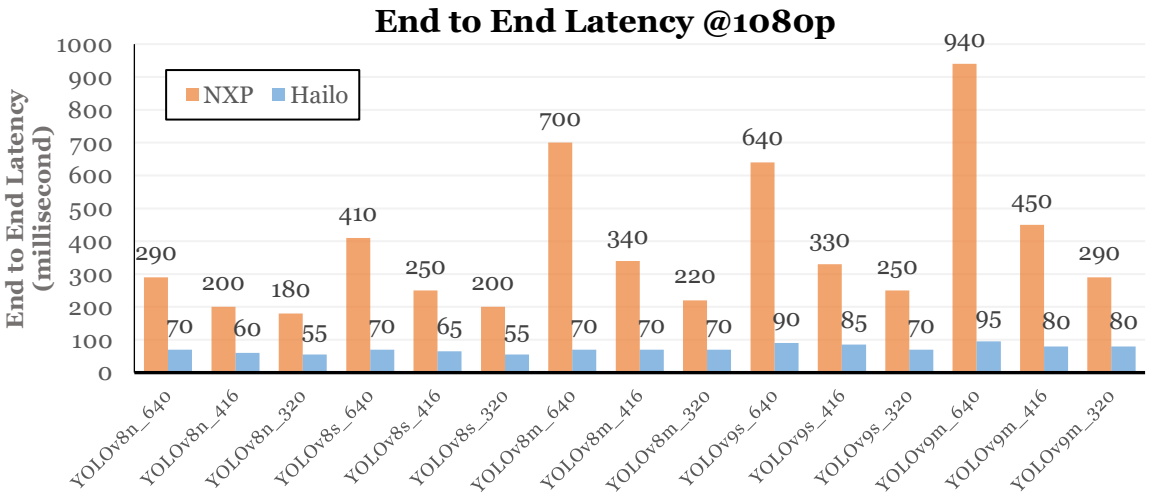


Figure 5.7: **Object detection end-to-end pipeline latency on NXP platform and Hailo platform for camera stream resolution of 1920x1080.**

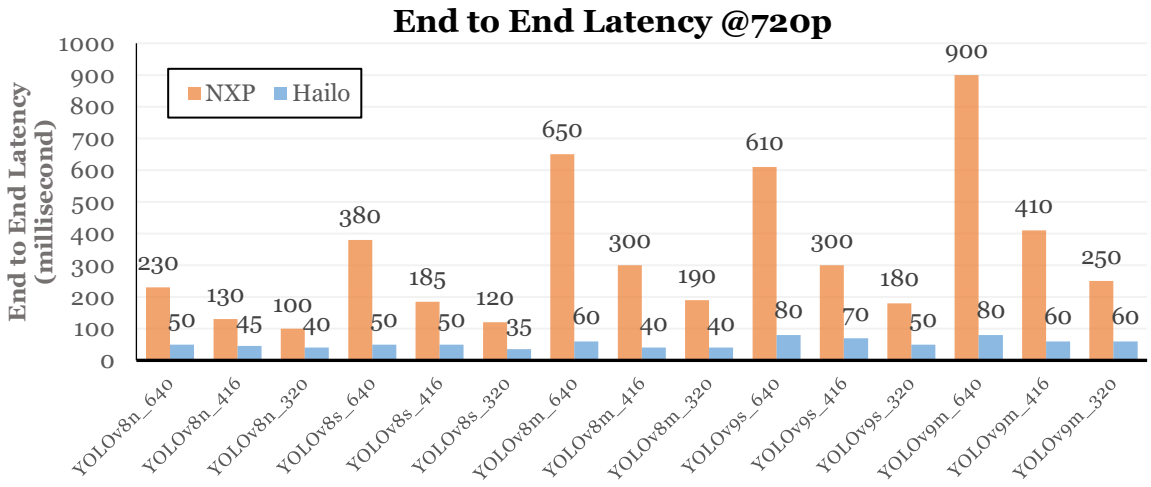


Figure 5.8: **Object detection end-to-end pipeline latency on NXP platform and Hailo platform for camera stream resolution of 1280x720.**

This increase can be attributed to several factors. One contributing factor is the latency introduced by non-inference stages such as frame acquisition from the camera and color space conversion. These stages are delayed due to resource contention, as they must wait for the inference process to complete. This is particularly significant because the LiteRT API used on the NXP platform handles preprocessing, inference, and postprocessing in a sequential manner.

In contrast, the Hailo platform benefits from the asynchronous API available through HailoRT. The latency overhead remains relatively stable regardless of model complexity, adding only around 30 to 40 milliseconds to the inference time. Furthermore, none of the models evaluated on the Hailo platform exceeded 100 milliseconds of total end-to-end latency. This performance makes Hailo more suitable for latency-sensitive applications such as autonomous driving, automated control systems, and robotics, where rapid response from input to action is essential.

5.2 Power Usage and Thermal Performance Analysis

This section analyzes the power consumption and thermal characteristics of the NXP and Hailo platforms during object detection inference at 1080p and 720p resolutions.

5.2.1 Power Usage Analysis

Figure 5.9 illustrates the inference power usage across the NXP and Hailo platforms at two different resolutions, 1080p and 720p, where several kay observations can be drawn

First, a clear correlation exists between camera resolution and power consumption. Lowering the resolution consistently reduces power usage during inference. This trend is evident when comparing each platform's power consumption at 1080p versus 720p. On average, reducing the camera resolution results in a power savings of approximately 0.19 Watts on the NXP platform and 0.56 Watts on the Hailo platform.

Secondly, the NXP platform consistently exhibits lower power consumption compared to the Hailo platform across both camera stream resolutions. The NXP platform averages 2.94 Watts for the 1080p stream and 2.75 Watts for the 720p stream. In contrast, the Hailo platform consumes significantly more power, averaging 5.11 Watts for the 1080p stream and 4.55 Watts for the 720p stream.

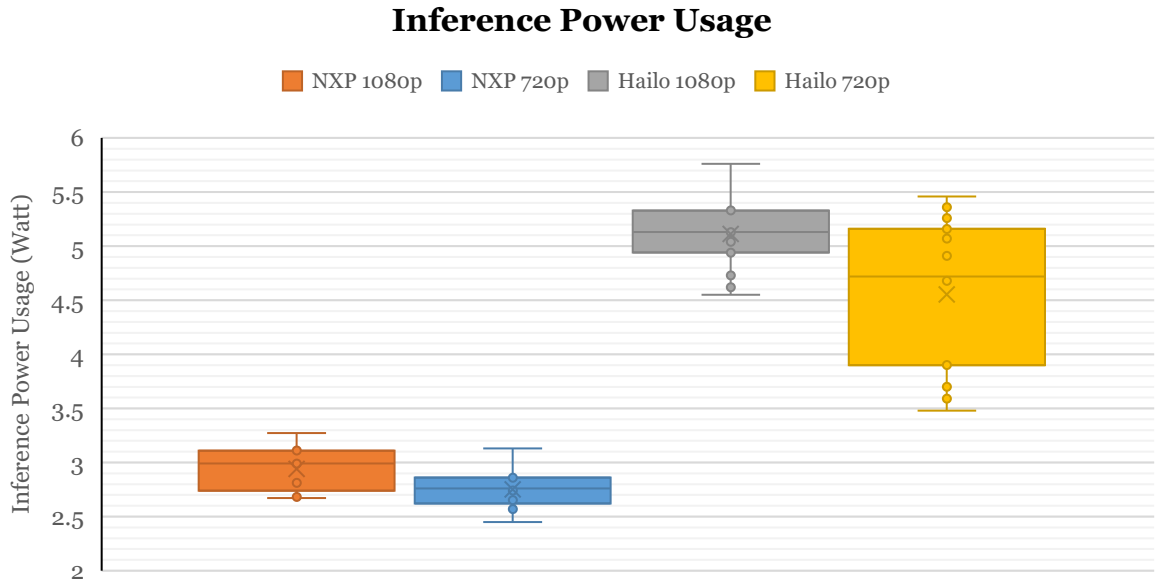


Figure 5.9: **Boxplot of inference power usage for NXP platform and Hailo platform for camera stream of 1920x1080 and 1280x720.**

Third, the variability in power usage differs noticeably between the two platforms. For the NXP platform, both 1080p and 720p camera stream resolutions show a narrow interquartile range in the boxplots, indicating stable and predictable power consumption across inference runs. The Hailo platform under a 1080p stream also demonstrates a relatively tight distribution of power usage. However, under a 720p stream, the Hailo platform exhibits a broader spread. Minimum power consumption in this configuration drops to 3.48 Watts, while the maximum rises to 5.46 Watts. The extended whiskers and wider interquartile range reflect greater variability in power demands when processing 720p streams on the Hailo platform.

When evaluating energy efficiency using FPS per Watt, the Hailo platform performs better overall, averaging 9.73 FPS/Watt compared to NXP's 5.57 FPS/Watt. The most efficient configuration for both platforms is YOLOv8n at 320x320 input size, where Hailo achieves 14.6 FPS/Watt and NXP reaches 13.16 FPS/Watt.

In conclusion, the NXP platform offers more consistent and power-efficient for total energy usage inference performance across both resolutions. The Hailo platform, while more powerful, exhibits higher average consumption and greater variability, particularly at 720p, making power efficiency a key trade-off to consider depending on the application's requirements.

5.2.2 Thermal Performance Analysis

Table 5.1 presents a statistical overview of thermal performance for both the NXP and Hailo platforms when processing camera streams at 1080p and 720p resolutions. On the Hailo setup, temperature measurements include both the CPU and the NPU, as it uses a Raspberry Pi-based system. In contrast, the NXP platform provides CPU temperature data only.

As expected, the Hailo platform's CPU runs cooler than the NXP's across both resolutions. This can be attributed to the active cooling system on the Raspberry Pi, which helps dissipate heat efficiently. The average CPU temperature on Hailo hovers in the low 60s°C, peaking at 67°C. Meanwhile, the NXP CPU operates with an average temperature in the high 70s to low 80s °C, reaching a peak of 83°C.

Despite running hotter, the NXP CPU remains below its thermal throttling threshold of 85°C. This suggests that the chip maintains peak performance under load, even at elevated temperatures. This is a critical aspect for ensuring consistent real-time operation in edge deployments.

Table 5.1: Statistical overview of the thermal performance of the NXP and Hailo platforms during object detection with camera stream resolutions of 1920×1080 and 1280×720.

	NXP Platform		Hailo Platform			
	1080p	720p	1080p		720p	
	CPU	CPU	CPU	NPU	CPU	NPU
Maximum of Average (°C)	82.15	82.1	69.1	48.2	64.33	50.72
Minimum of Average (°C)	73.78	76.97	58.67	38.4	57.45	39.1
Mean of Average (°C)	79.57	78.90	63.17	42.83	61.10	43.85
Standard Deviation	2.10	1.38	2.34	2.96	2.88	3.02
Coefficient of Variation	2.64%	1.75%	3.70%	6.91%	4.71%	6.89%
Peak Recorded Temperature (°C)	83	83	67.2	48.7	66.65	52.3

The thermal behavior of the Hailo NPU is especially notable. With an average temperature around 43°C and a peak of 52.3°C, it shows excellent thermal efficiency. Although the active cooling targets the CPU, the airflow from the fan likely contributes to maintaining favorable NPU temperatures.

When comparing resolutions, both platforms show lower CPU temperatures at 720p. The NXP CPU temperature drops by 0.67°C on average, while Hailo shows a larger reduction of 2.07°C. Interestingly, the Hailo NPU runs slightly hotter at 720p, with a 1.02°C increase in average temperature, possibly due to changes in processing load distribution at different resolutions.

In summary, the NXP platform, though running hotter, stays within safe thermal limits, ensuring performance stability. The Hailo platform benefits from active cooling and maintains both CPU and NPU temperatures well below critical thresholds, offering greater thermal headroom and robust performance under varying conditions.

5.3 Resource Utilization Analysis

This section analyzes the RAM usage and CPU utilization of the NXP and Hailo platforms during object detection inference at 1080p and 720p resolutions.

5.3.1 RAM Usage Analysis

Figure 5.10 and Figure 5.11 illustrate the RAM usage patterns of the NXP and Hailo platforms during object detection pipeline execution at 1080p resolution. A key takeaway from the data is the significantly greater variability in RAM consumption on the NXP platform compared to the consistently stable usage observed on the Hailo platform.

On the NXP, system RAM usage peaks at 1380 MB, with the inference process alone consuming up to 930 MB when running the YOLOv9s model at a 640×640 input resolution. Additionally, the YOLOv9 architecture exhibits noticeably higher memory demands than YOLOv8 on the NXP, suggesting that newer and more complex models place a heavier load on system resources.

In contrast, the Hailo platform maintains stable memory usage across all tested models, with total system RAM averaging around 920 MB and inference RAM averaging just 318 MB. This highlights the memory efficiency of the Hailo platform, particularly in its handling of inference workloads.

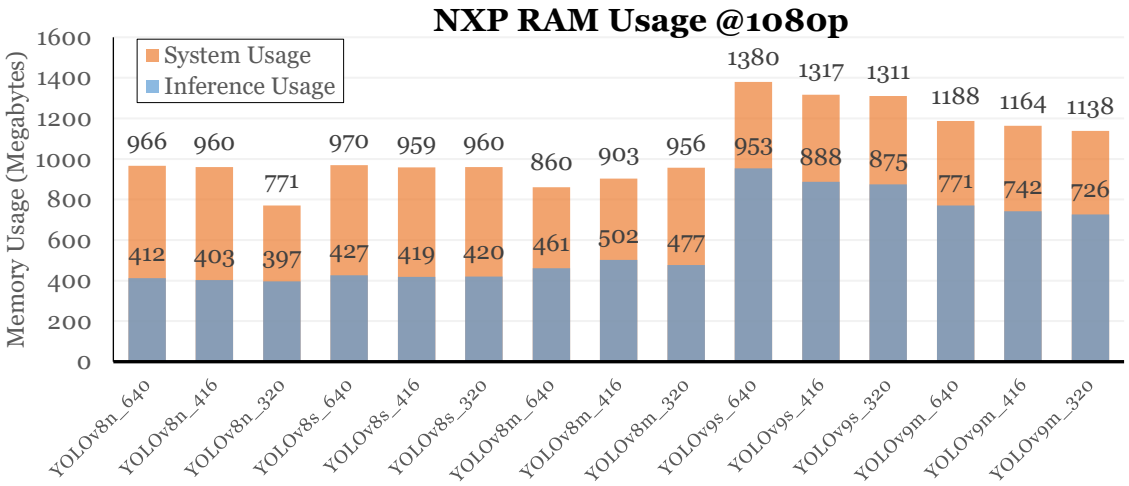


Figure 5.10: **RAM usage of NXP platform when running object detection pipeline with camera stream resolution of 1920x1080.**

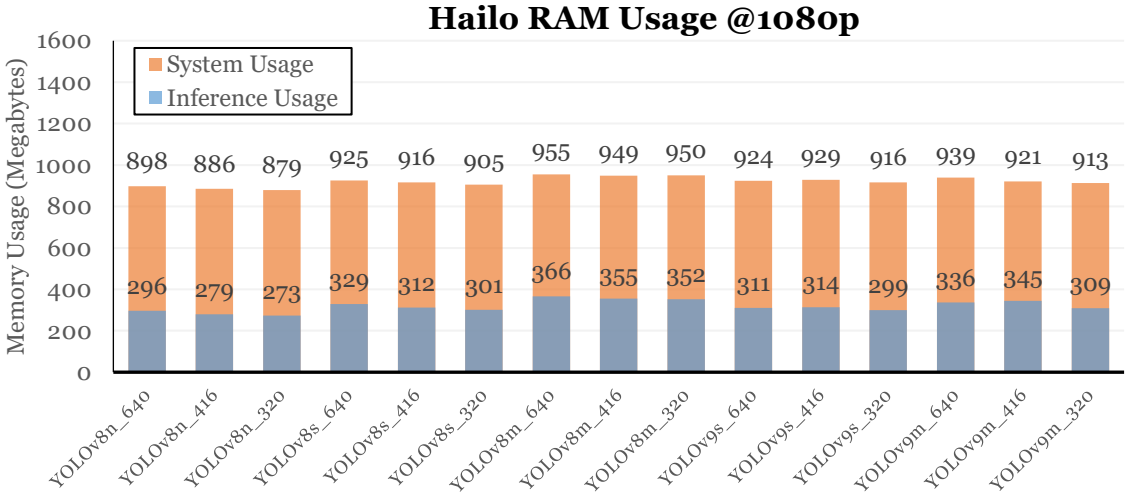


Figure 5.11: **RAM usage of Hailo platform when running object detection pipeline with camera stream resolution of 1920x1080.**

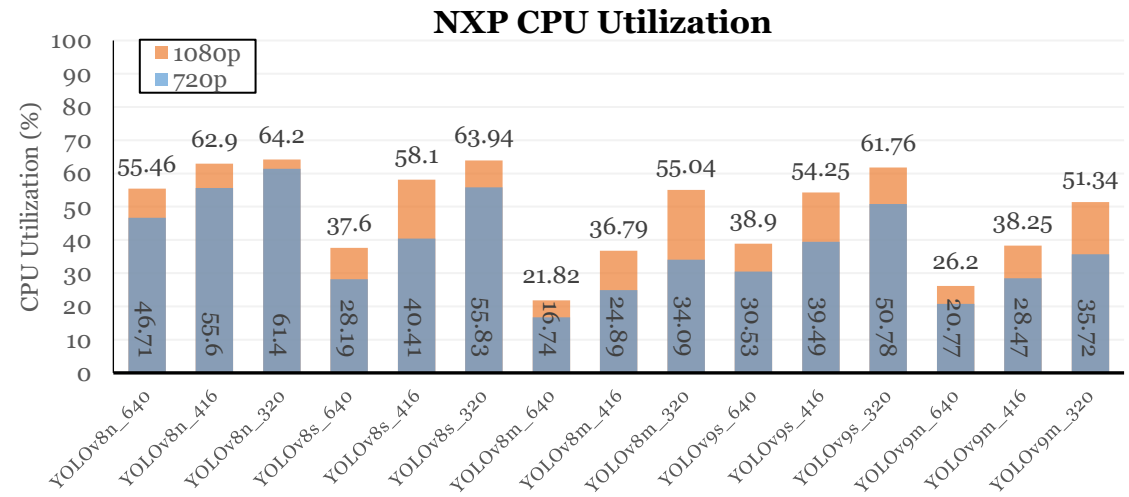


Figure 5.12: **CPU utilization of the NXP platform when running the object detection pipeline with camera stream resolutions of 1920×1080 and 1280×720. (Note: Data labels at the base of each bar**

indicate CPU utilization for 720p, while data labels at the end of each bar indicate CPU utilization for 1080p.)

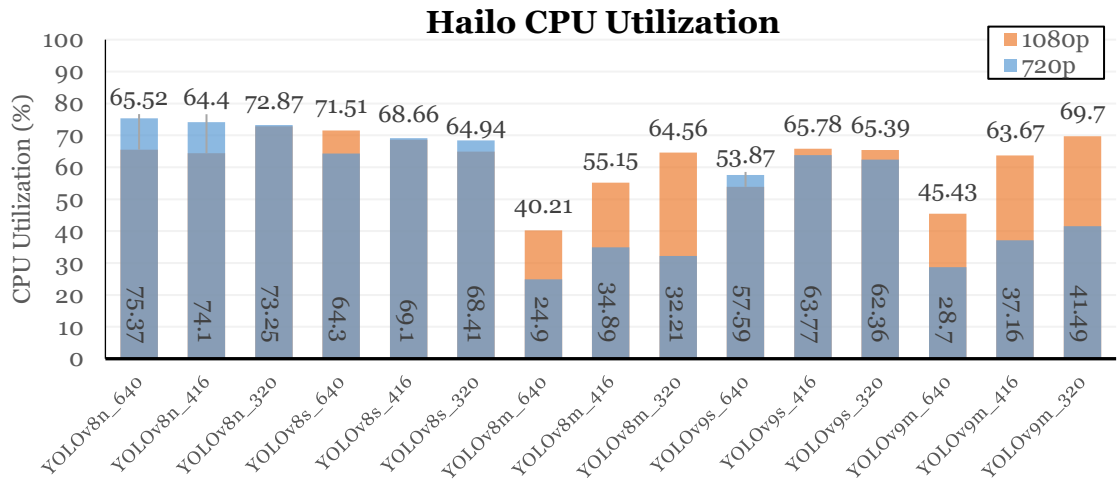


Figure 5.13: **CPU utilization of the Hailo platform when running the object detection pipeline with camera stream resolutions of 1920×1080 and 1280×720.**

Table 5.2: **Quantitative comparison of average system and inference RAM usage on the NXP and Hailo platforms for camera stream resolutions of 1920×1080 and 1280×720.**

	NXP	Hailo
Average System RAM Usage at 1080p (MB)	1053.53	920.33
Average Inference RAM Usage at 1080p (MB)	591.53	318.47
Average System RAM Usage at 720p (MB)	970.87	853.07
Average Inference RAM Usage at 720p (MB)	535.67	271.07
Difference on System RAM Usage Average (MB)	82.67	67.27
Difference on Inference RAM Usage Average (MB)	55.87	47.4

Table 5.2 quantifies the average RAM usage for both platforms at 1080p and 720p, including the reductions achieved by lowering the input resolution. On the NXP, decreasing the resolution from 1080p to 720p results in an average reduction of 82.67 MB in total system RAM usage and 55.87 MB in inference RAM usage. Similarly, the Hailo platform sees a 67.27 MB drop in total RAM usage and a 47.4 MB reduction in inference RAM.

While these reductions confirm that lower resolution slightly alleviates memory usage, the gains are relatively modest. This suggests that RAM consumption is influenced more by the complexity of the selected object detection model than by the resolution of the input stream. Model architecture and inference framework efficiency remain the dominant factors in determining overall memory demand on both platforms.

5.3.2 CPU Utilization Analysis

Figure 5.12 and Figure 5.13 illustrate the CPU utilization for the NXP and Hailo platforms when running the object detection pipeline at 1080p and 720p camera stream resolutions, respectively.

On the NXP platform, a clear pattern emerges. Lower input shapes result in higher CPU utilization for the same model, and simpler models consume more CPU for a given input shape. This is likely because smaller input shapes enable higher frame rates, requiring the CPU to process more frames per second. As a result, CPU usage increases to keep pace with the faster data throughput.

Additionally, reducing the camera resolution to 720p noticeably decreases CPU usage, providing additional headroom for other application-level tasks.

On the Hailo platform, the pattern is less consistent. Some trends are still observable, particularly with more complex configurations such as YOLOv9m and YOLOv8m, where lower input shapes lead to higher CPU utilization similar to the trend seen on NXP. However, for other models, CPU usage remains relatively stable, generally ranging from the mid-60s to low-70s percentage-wise. Interestingly, in some cases, the 720p pipeline even results in higher CPU utilization than 1080p. This is due to the increased frame rate at 720p requiring more CPU involvement, despite the lower resolution.

These results suggest that on the NXP platform, the CPU is more directly involved in managing frame rates and handling pre- and post-processing tasks. Consequently, CPU utilization is more tightly coupled with input shape and model complexity. Lowering the resolution clearly helps alleviate CPU load.

In contrast, on the Hailo platform, the CPU appears to play a more supportive role alongside the dedicated NPU. While some expected trends are visible, other factors, such as system-level data handling, thread scheduling, and synchronization with the NPU, introduce variability. The system might prioritize maintaining higher frame rates at lower resolutions, which could lead to greater CPU activity for managing data pipelines or auxiliary processes, even when model complexity is lower.

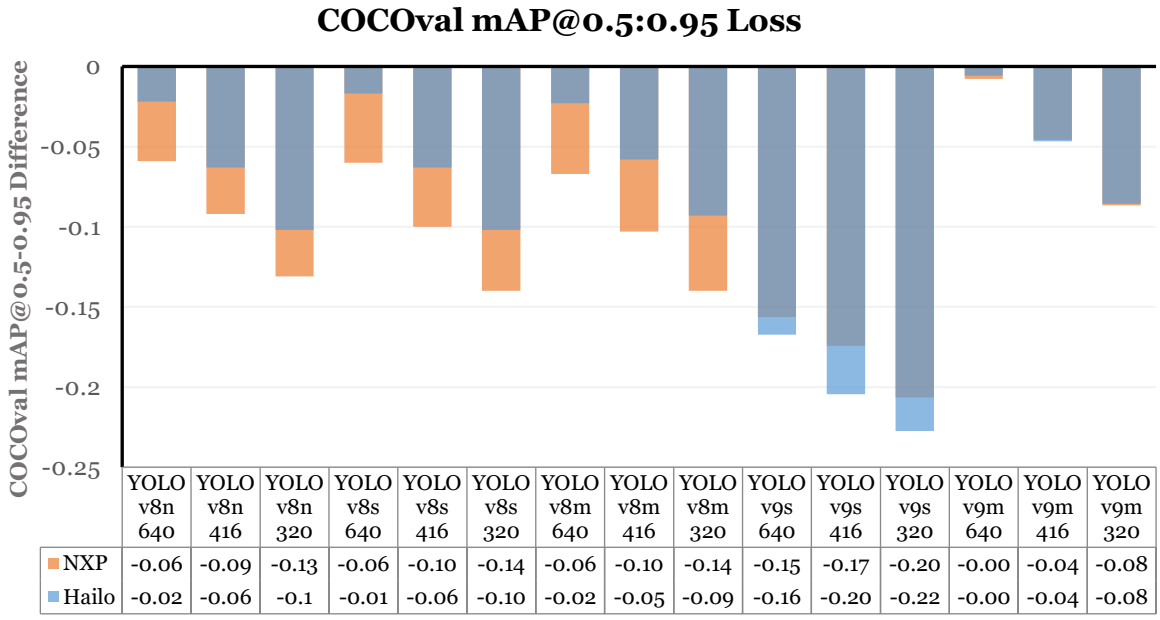
5.4 Model Optimization Impact Analysis

The impact of optimization strategies and input shape variations on mAP was evaluated using the COCO validation dataset. Sample inference results from both the NXP and Hailo platforms are shown in Figure 5.14.

The mAP loss resulting from optimization efforts is shown in Figure 5.15. As anticipated, reducing the input shape from 640 to 416 and further to 320 consistently resulted in a decline in mAP across all tested YOLO models (v8n, v8s, v8m, v9s, v9m) on both the NXP and Hailo platforms. This trend aligns with the established understanding that smaller input sizes reduce the spatial detail available to the model, thereby impairing its ability to detect and localize objects accurately, particularly smaller objects. For the YOLOv8 architecture, the mAP degradation across input shape variations remained relatively consistent, with average losses summarized in Table 5.3.

When comparing the two platforms and their respective quantization pipelines for the YOLOv8 family, the Hailo platform exhibited a smaller and more consistent mAP loss than the NXP. In particular, Hailo retained more accuracy when the input shape was reduced to 416, outperforming the NXP's direct quantization approach in terms of average accuracy loss. This indicates that Hailo’s optimization framework is more effective at preserving model fidelity under reduced input resolutions. In contrast, the NXP platform introduced a noticeable baseline accuracy degradation immediately after quantization even before accounting for further losses due to input shape reduction.

Figure 5.14: **Sample inference results for YOLOv8 and YOLOv9 models on NXP and Hailo platform.**



	YOLO v8n 640	YOLO v8n 416	YOLO v8n 320	YOLO v8s 640	YOLO v8s 416	YOLO v8s 320	YOLO v8m 640	YOLO v8m 416	YOLO v8m 320	YOLO v9s 640	YOLO v9s 416	YOLO v9s 320	YOLO v9m 640	YOLO v9m 416	YOLO v9m 320
NXP	-0.06	-0.09	-0.13	-0.06	-0.10	-0.14	-0.06	-0.10	-0.14	-0.15	-0.17	-0.20	-0.00	-0.04	-0.08
Hailo	-0.02	-0.06	-0.1	-0.01	-0.06	-0.10	-0.02	-0.05	-0.09	-0.16	-0.20	-0.22	-0.00	-0.04	-0.08

Figure 5.15: **COCOval mAP@0.5:0.95 loss on NXP and Hailo platforms for YOLOv8 and YOLOv9.**

Table 5.3: **Quantitative comparison of average COCOval mAP@0.5:0.95 loss on NXP and Hailo platform for YOLOv8.**

	NXP	Hailo
Average mAP Loss Direct Quantization	-0.062	-0.021
Average mAP Loss 416x416	-0.098	-0.061
Average mAP Loss 320x320	-0.137	-0.099

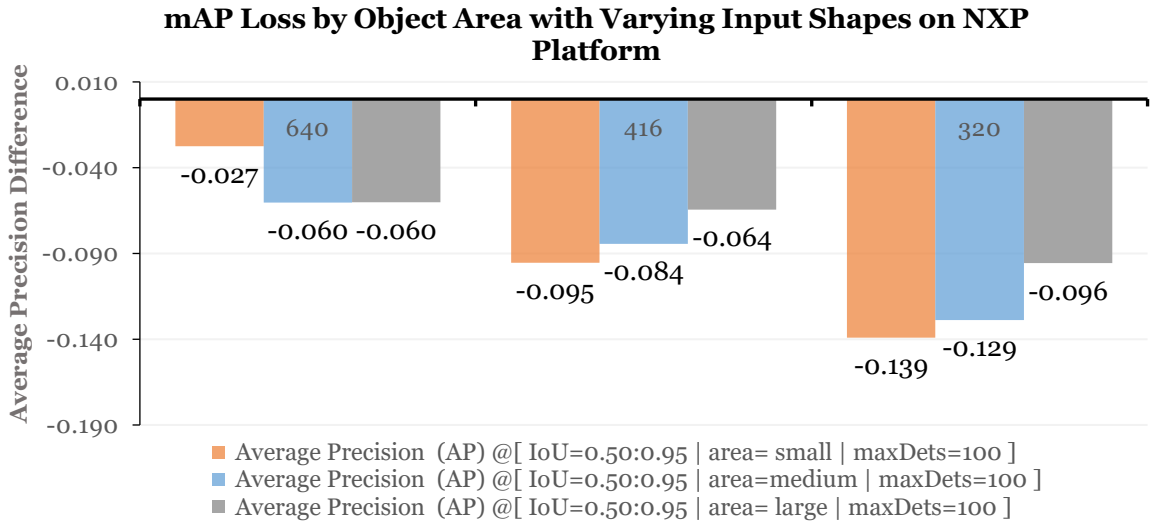


Figure 5.16: **COCOval mAP@0.5:0.95 loss by object area with varying input shapes on NXP platform for YOLOv8 and YOLOv9.**

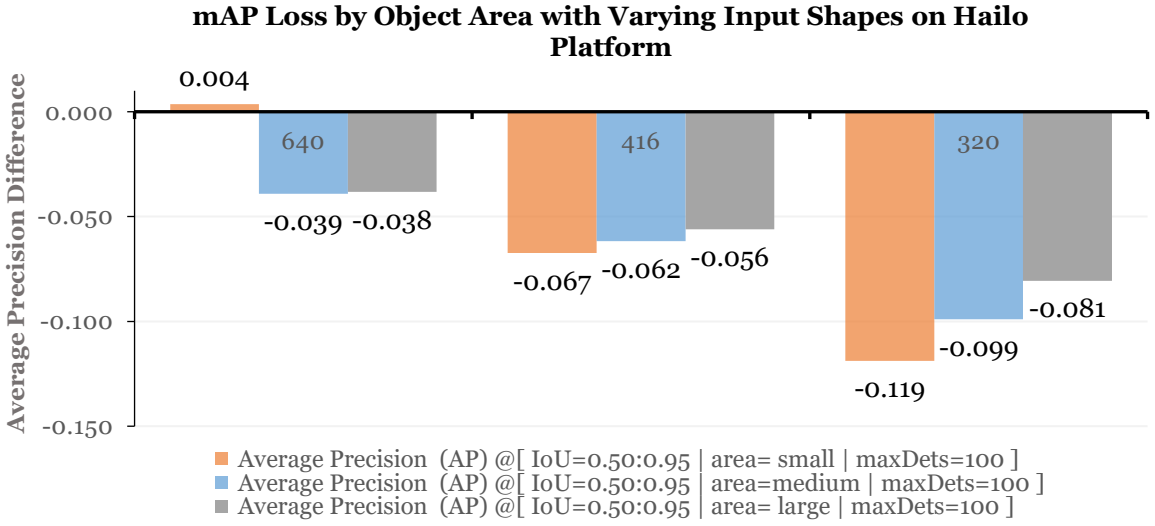


Figure 5.17: **COCOval mAP@0.5:0.95 loss by object area with varying input shapes on Hailo platform for YOLOv8 and YOLOv9.**

The analysis of optimization impact on the YOLOv9 model family presents a more nuanced outcome compared to the YOLOv8 models. The YOLOv9s model experiences a noticeable mAP loss after quantization and input shape reduction, whereas the larger YOLOv9m model maintains remarkably high accuracy, even outperforming YOLOv8 models in terms of post-optimization mAP retention. Notably, direct quantization results in near-zero accuracy degradation for YOLOv9m on both the NXP and Hailo platforms, indicating strong robustness to quantization. While both platforms show similar mAP preservation for YOLOv9m, the NXP platform demonstrates better accuracy retention than Hailo for the smaller YOLOv9s model. This suggests that the effectiveness of a quantization pipeline may vary with model size and complexity, with NXP's approach being more favorable for lighter models in the YOLOv9 family.

Figure 5.16 and Figure 5.17 illustrate the mAP loss across different object area categories when reducing the input shape on the NXP and Hailo platforms, respectively. A consistent trend emerges: smaller objects experience the highest mAP loss, while larger objects are less affected. This is expected, as lower input resolutions reduce spatial detail and disproportionately impact the detection of small objects, which depend on fine-grained visual features. In contrast, larger objects retain more distinguishing characteristics even when the input size is reduced.

This has important implications for real-world deployments. Applications that focus on detecting large objects, such as vehicles, machinery, or infrastructure, may benefit from smaller input sizes, achieving faster inference with minimal accuracy loss. On the other hand, applications that involve detecting small or densely packed objects, such as pedestrians in crowds, tools, or manufacturing defects, are likely to experience greater accuracy degradation. Therefore, the choice to reduce input size should be guided by the characteristics of the target objects and the specific performance and accuracy requirements of the application.

5.5 Cost analysis

The cost analysis presented in this section focuses on the SoM or chipset level, using publicly available prices.

5.5.1 Estimate of Total Cost of Ownership

The analysis assumes a deployment scenario of 100 devices over a five-year period. Costs related to peripherals such as cameras, power supplies, SD cards, and carrier boards are grouped into a variable, since these components are assumed to be the same for both platforms. Additional infrastructure or service-related expenses are not considered in the calculation, as they are assumed to be equivalent across both configurations.

For the NXP configuration, the listed price for the SoM is 1630 SEK, with a reported Mean Time Between Failures (MTBF) of 6,183,000 hours. The MTBF for peripheral components is more difficult to determine due to limited public data. To ensure a conservative estimate, the lowest commonly available MTBF is used, which is approximately 10,000 hours for camera modules. Based on this information, the initial cost of one NXP device is expressed as:

$$Initial\ Cost\ of\ NXP\ = 1630\ SEK + x$$

Assuming continuous operation, the expected number of failures over five years can be estimated using the MTBF values. For the NXP SoM, the failure rate is calculated as:

$$Total\ Expected\ Failures\ Over\ 5\ Years\ = \frac{\left(8760\ \frac{hours}{year}\right)}{6138000\ hours} * 5\ years = \ 0.00714\ Failures$$

For the peripherals, the calculation is:

$$Total\ Expected\ Failures\ Over\ 5\ Years\ = \frac{\left(8760\ \frac{hours}{year}\right)}{10000\ hours} \cdot 5\ years\ =\ 4.38\ Failures$$

Using these figures, the replacement cost over the five-year period can be determined. The total replacement cost for the SoM is:

$$Total\ Replacement\ Cost\ of\ SoM = 0.00714\ Failures\ * 1630\ SEK\ = 11.6382\ SEK$$

And the replacement cost for the peripherals is:

$$Total\ Replacement\ Cost\ of\ Peripherals = 4.38\ Failures\ * x = 4.38\ x$$

To evaluate operational costs, energy consumption is also considered. The highest observed average power usage is used to represent a worst-case scenario. According to public data, the average electricity price in 2025 for the SE3 region, which includes Stockholm, is projected at 564.13 SEK per megawatt-hour. Power consumption from the peripherals is represented by the variable y. The total power consumption of a single NXP device is:

$$Total\ Energy\ Usage = 3.271\ Watt + y$$

The energy cost over five years for one device is then calculated as:

$$Total\ Energy\ Cost = \frac{3.271 + y}{1000000}\ (MW) * 8760\ \frac{hours}{year} * 5\ years * 564.13\ \frac{SEK}{MWh} = 80.82\ SEK +\ 24.71y$$

There is also additional cost such as development and maintenance cost that is represented by the variable z. The same methodology is applied to the Hailo platform. Since the MTBF of the Hailo-8L is not publicly documented, it is conservatively assumed to be equivalent to that of the Raspberry Pi 5, which is rated at 93,800 hours. The outcomes of these calculations for both platforms are presented in Table 5.4 for the NXP configuration and Table 5.5 for the Hailo configuration.

As demonstrated by the cost calculations, although the initial price of each device is relatively similar between the NXP and Hailo configurations, the total cost over a five-year period reveals a more significant difference. When considering both the potential replacement costs and the ongoing energy expenses, the NXP platform proves to be more economical overall. While NXP has slightly higher energy consumption, it benefits from a significantly higher MTBF, which reduces the likelihood and associated cost of hardware replacements during the deployment period.

This cost advantage becomes even more evident when scaled to a larger deployment scenario. In a five-year deployment of 100 devices, the difference in total cost between the two platforms amounts to 82,411 SEK, with NXP being the less expensive option. This highlights the importance of evaluating not only the upfront hardware price but also the operational and maintenance costs that accumulate over time.

In addition, the NXP platform achieves this cost efficiency without sacrificing reliability or system longevity. Its superior durability translates to fewer failures and reduced downtime, which are critical factors in edge computing environments where consistent performance is essential. For organizations aiming to implement large-scale, long-term deployments, especially those with limited budgets or strict operational cost requirements, the NXP configuration presents a more sustainable and financially sound choice.

Table 5.4: **Total cost estimation over a 5-year deployment for the NXP platform.**

Category	Name	Price 1 Device 5 Years	Price 100 Devices 5 Years
Initial Cost	DART SOM MX8MP	1,630 SEK	163,000 SEK
	Peripherals & Other	x	$100x$
Replacement Cost	DART SOM MX8MP Replacement	11.64 SEK	1,163.82 SEK
	Peripherals & Other Replacement	$4.38\,x$	$438\,x$
Energy Cost	Inference Cost	80.82 SEK	8,082.00 SEK
	Peripherals & Other Energy Cost	y	$100y$
Other Costs	Development and Maintenance Costs	z	$100z$
	Total Cost	$1{,}772.46\ \text{SEK} + 5.38x + y + z$	$177{,}246\ \text{SEK} + 538x + 100y + 100z$

Table 5.5: **Total cost estimation over a 5-year deployment for the Hailo platform.**

Category	Name	Price 1 Device 5 Years	Price 100 Devices 5 Years
Initial Cost	Raspberry Pi 5	714 SEK	71,400 SEK
	Hailo-8L AI Kit	959 SEK	95,900 SEK
	Peripherals & Other	x	$100x$
Replacement Cost	Raspberry Pi 5 Replacement	333.4 SEK	33,340 SEK
	Hailo-8L Replacement	447.85 SEK	44,785 SEK
	Peripherals & Other Replacement	$4.38\,x$	$438\,x$
Energy Cost	Inference Cost	142.32 SEK	14,232 SEK
	Peripherals & Other Energy Cost	y	$100y$
Other Costs	Development and Maintenance Costs	z	$100z$
	Total Cost	$2{,}596.57\ \text{SEK} + 5.38x + y + z$	$259{,}657\ \text{SEK} + 538x + 100y + 100z$

5.5.2 Performance-to-Cost Ratio Analysis

To compare the performance-to-cost ratio of the NXP and Hailo platforms, we evaluated the inference speed FPS of the YOLOv8s model with a 416x416 input shape when processing 720p video, as this configuration offered a good balance between model size and inference latency on both devices. The performance-to-cost calculations are shown in Table 5.6.

At the individual device level, the performance-to-cost ratio was calculated to be 0.0117 FPS/SEK for the NXP platform and a significantly higher 0.036 FPS/SEK for the Hailo platform, indicating that Hailo provides more than double the frames per second for each SEK spent on the singular device.

However, when considering the Total Cost of Ownership for a larger deployment of 100 devices over a 5-year period of constant operation, the FPS/SEK ratio changes. Based on our cost calculations (detailed in Section 5.5.1), the Hailo platform still demonstrates a better cost-effectiveness with a ratio of 0.0002 FPS/SEK, compared to the NXP platform's 0.0001 FPS/SEK. While Hailo maintains a better ratio, the difference is less pronounced when accounting for long-term operational costs, suggesting that factors such as power consumption and potential replacement costs play a more significant role in the overall cost-effectiveness at scale.

This analysis underscores the need to evaluate both upfront costs and long-term operational expenses when assessing the cost-to-performance of edge computing platforms for sustained deployments. While Hailo provides a better initial FPS/SEK ratio, the TCO over five years shows a reduced advantage. This highlights the importance of understanding specific cost drivers and aligning platform selection with the application’s performance and budget requirements.

Table 5.6: Performance-to-Cost calculation.

	NXP Platform		**Hailo Platform**	
	Single Device	100 Devices, 5 Years	Single Device	100 Devices, 5 Years
YOLOv8s 416x416 @720p Performance	19 FPS		60 FPS	
Cost	1,630 SEK	177,246 SEK	1,673 SEK	259,657 SEK
Performance-to-Cost Ratio	0.0117 FPS/SEK	0.0001 FPS/SEK	0.036 FPS/SEK	0.0002 FPS/SEK

Chapter 6
Conclusions and Future Works

This chapter presents the conclusions drawn from the work conducted in this thesis, discusses the key limitations encountered during the research, and outlines potential directions for future work in the field.

6.1 Conclusions

Conclusions based on the research questions are as follows:

- RQ1: The NXP i.MX 8M Plus demonstrates significantly lower throughput for real-time object detection when compared to the Hailo-8L. Despite this, the NXP platform performs more efficiently in terms of absolute energy consumption. However, when considering the performance-to-energy ratio, measured in frames per second per watt (FPS/Watt), the Hailo-8L offers a better balance by delivering more inference performance for each unit of power consumed. In terms of thermal characteristics, both platforms maintain stable operation under peak loads without experiencing thermal throttling, indicating that their thermal designs are sufficient for sustained edge AI workloads.

- RQ 2: Quantization of the COCO pre-trained YOLOv8 models results in a modest and consistent mAP loss of about 0.04, indicating good post-quantization performance. In contrast, YOLOv9 models show varied results: YOLOv9s suffers significant degradation with a 0.162 mAP loss, while YOLOv9m retains high accuracy with only a 0.007 loss. Reducing input shape consistently lowers mAP, especially for small objects, while larger objects are less affected. This highlights the trade-off between computational efficiency and detection accuracy, particularly in small-object detection scenarios.

- RQ 3: Over a 5-year deployment involving 100 devices, the NXP platform proves more cost-efficient than the Hailo platform due to its lower energy consumption and higher reported reliability, which contribute to reduced operational and replacement costs. While Hailo delivers nearly double the frames per second per Swedish krona (FPS/SEK) for a single device running YOLOv8s at 416×416, indicating superior cost-to-performance, it is better suited for applications requiring very low latency and high throughput. However, when scaled to a larger deployment, this advantage narrows. This suggests that although Hailo excels in inference efficiency, the NXP platform is the more economical choice for long-term deployments with moderate performance requirements.

6.2 Limitations

One of the primary limitations of this thesis is the restricted access to hardware resources for training purposes. Training custom models, as opposed to relying solely on pre-trained models, would have provided a more accurate assessment of object detection performance tailored to specific application requirements. This limitation consequently narrows the scope of the research.

With access to more capable hardware, it would have been possible to conduct a more comprehensive investigation into the impact of custom model training on the evaluated performance metrics. Additionally, such resources would have enabled techniques like re-training or fine-tuning, which are crucial for preserving model accuracy following quantization or input shape reduction.

6.3 Future Works

A promising direction for future research in Edge AI lies in the exploration of additional model optimization techniques beyond quantization. Methods such as pruning, which involves systematically removing less important weights or neurons from the network, and weight clustering, which reduces the number of unique weight values by grouping them into clusters, offer potential to significantly reduce the model's size and computational demands. These techniques could be particularly valuable for edge deployments, where hardware resources are inherently constrained. Evaluating the trade-offs between computational efficiency and accuracy across different architectures and tasks on various edge platforms would help determine their practical effectiveness.

In addition to classical convolutional neural networks, transformer-based object detection models, such as RT-DETR, represent another compelling avenue for future work. These models are specifically designed for fast inference and have shown impressive results in balancing speed and accuracy on general-purpose hardware. However, their successful deployment on edge computing platforms remains a challenge due to the limited support for transformer operations on many current NPUs. Exploring how such models perform under various optimization schemes, along with investigating hardware-software co-design strategies or middleware frameworks that can better support transformer inference, would be a valuable step forward.

Furthermore, as edge devices become increasingly heterogeneous, future research should also examine cross-platform model portability, adaptive inference pipelines, and dynamic model scaling. These capabilities could enable edge AI systems to automatically adjust computational loads based on available resources or environmental conditions, further enhancing robustness and performance in real-world applications.

References

[1] Xiaofei Wang, *Edge AI: Convergence of Edge Computing and Artificial Intelligence*. Singapore: Springer Singapore Pte. Limited, 2020, ISBN: 978-981-15-6186-3.

[2] Raghubir Singh and Sukhpal Singh Gill, 'Edge AI: A survey', *Internet Things Cyber-Phys. Syst.*, vol. 3, pp. 71–92, 2023. DOI: 10.1016/j.iotcps.2023.02.004

[3] Amy Neustein, *AI, IoT, Big Data and Cloud Computing for Industry 4. 0*, 1st ed. Cham: Springer International Publishing AG, 2023, Signals and Communication Technology Series, ISBN: 978-3-031-29712-0.

[4] Daniel Situnayake and Jenny Plunkett, *AI at the Edge: solving real-world problems with embedded machine learning*. Sebastopol: O'Reilly, 2023, ISBN: 978-1-09-812020-7.

[5] Yuan Zeng, Bin Song, Yuwen Chen, Xiaojiang Du, and Mohsen Guizani, 'Few-Shot Scale-Insensitive Object Detection for Edge Computing Platform', *IEEE Trans. Sustain. Comput.*, vol. 7, no. 4, pp. 726–735, Oct. 2022. DOI: 10.1109/TSUSC.2020.3043758

[6] Khaled B. Letaief, Yuanming Shi, Jianmin Lu, and Jianhua Lu, 'Edge Artificial Intelligence for 6G: Vision, Enabling Technologies, and Applications', *IEEE J. Sel. Areas Commun.*, vol. 40, no. 1, pp. 5–36, Jan. 2022. DOI: 10.1109/JSAC.2021.3126076

[7] Ella Peltonen, Mehdi Bennis, Michele Capobianco, Merouane Debbah, Aaron Ding, Felipe Gil-Castiñeira, Marko Jurmu, Teemu Karvonen, Markus Kelanti, Adrian Kliks, Teemu Leppänen, Lauri Lovén, Tommi Mikkonen, Ashwin Rao, Sumudu Samarakoon, Kari Seppänen, Paweł Sroka, Sasu Tarkoma, and Tingting Yang, '6G White Paper on Edge Intelligence'. arXiv, 30-Apr-2020 [Online]. DOI: 10.48550/arXiv.2004.14850

[8] Alec May, 'Increasing intelligence at the edge with embedded processors', Texas Instrument, Whitepaper, Nov. 2024.

[9] 'Transforming Edge AI: The power of neural processing units in modern microcontrollers', STMicroelectronics, Whitepaper, Dec. 2024.

[10] 'Hailo-8L Entry-Level M.2 AI Acceleration Modules Product Brief', Hailo Technologies, Product Brief, Dec. 2024.

[11] 'Raspberry Pi AI HAT+ Product Brief', Raspberry Pi, Product Brief, Oct. 2024.

[12] 'Coral Accelerator Module datasheet v1.4', Coral Google, G313-06329–00, 2020.

[13] Samer Francy and Raghubir Singh, 'Edge AI: Evaluation of Model Compression Techniques for Convolutional Neural Networks'. arXiv, 2024 [Online]. DOI: 10.48550/ARXIV.2409.02134

[14] Cem Ünsalan, Berkan Höke, and Eren Atmaca, *Embedded Machine Learning with Microcontrollers: Applications on STM32 Development Boards*. Cham: Springer International Publishing, 2025, ISBN: 978-3-031-70911-1 [Online]. DOI: 10.1007/978-3-031-70912-8

[15] Yuyi Mao, Xianghao Yu, Kaibin Huang, Ying-Jun Angela Zhang, and Jun Zhang, 'Green Edge AI: A Contemporary Survey', *Proc. IEEE*, vol. 112, no. 7, pp. 880–911, Jul. 2024. DOI: 10.1109/JPROC.2024.3437365

[16] 'AI at the Edge: The next frontier of the internet of things', AVNET, Whitepaper, 2018.

[17] Antonio Troise, 'CPU, GPU, and NPU: Understanding Key Differences and Their Roles in Artificial Intelligence', *Medium*. 18-Jun-2024 [Online]. Available: https://levysoft.medium.com/cpu-gpu-and-npu-understanding-key-differences-and-their-roles-in-artificial-intelligence-2913a24d0747. [Accessed: 21-Feb-2025]

[18] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner, 'AI and ML Accelerator Survey and Trends', in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022, pp. 1–10 [Online]. DOI: 10.1109/HPEC55821.2022.9926331

[19] Rishika Patel, 'GPU - CPU - NPU: Understanding the Differences', *CIO Influence*. 18-Sep-2024 [Online]. Available: https://cioinfluence.com/hardware/gpu-cpu-npu-understanding-the-differences-and-their-strategic-importance/. [Accessed: 24-Feb-2025]

[20] Babbage, 'Demystifying GPU Compute Architectures', *The Chip Letter*. 04-Feb-2024 [Online]. Available: https://thechipletter.substack.com/p/demystifying-gpu-compute-architectures. [Accessed: 24-Feb-2025]

[21] Michel Delli Abo, *An Efficiency Comparision of NPU, CPU, and GPU When Exceuting an Object Detection Model YOLOv5*. 2024 [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-351119. [Accessed: 21-Feb-2025]

[22] Matej Kutirov, 'Precision in AI — Scaling Laws for Training and Inference', *Medium*. 23-Nov-2024 [Online]. Available: https://medium.com/@matejkutirov/precision-in-ai-scaling-laws-for-training-and-inference-a30cfca9b6e1. [Accessed: 24-Feb-2025]

[23] 'Introducing the NXP eIQ®Neutron Neural Processing Unit (NPU)'. [Online]. Available: https://www.nxp.com/company/blog/introducing-the-nxp-eiq-neutron-neural-processing-unit-npu:BL-INTRODUCING-THE-NXP-EIQ-NPU. [Accessed: 24-Feb-2025]

[24] Kyuho J. Lee, 'Architecture of neural processing unit for deep neural networks', in *Advances in Computers*, vol. 122, Elsevier, 2021, pp. 217–245 [Online]. DOI: 10.1016/bs.adcom.2020.11.001

[25] ‘Arm Ethos-U65 NPU Technical Reference Manual’. [Online]. Available: https://developer.arm.com/documentation/102023/0000/Introduction-to-the-NPU/Description-of-the-neural-processing-unit-. [Accessed: 24-Feb-2025]

[26] ‘Neural Processor Market Size, Share & Growth Report, 2030’. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/neural-processor-market-report. [Accessed: 24-Feb-2025]

[27] John Loeffler, ‘What is an NPU: the new AI chips explained’, *TechRadar*, 15-Jan-2024. [Online]. Available: https://www.techradar.com/computing/cpu/what-is-an-npu. [Accessed: 24-Feb-2025]

[28] ‘STM32MP251C/F STM32MP253C/F STM32MP255C/F STM32MP257C/F Datasheet’, STMicroelectronics, Datasheet DS14284 Rev 4, Jan. 2025.

[29] ‘VAR-SOM-MX8M-PLUS_V2.x Datasheet’, Variscite, Datasheet Rev. 1.03, Feb. 2025.

[30] ‘RK3588 Brief Datasheet’, Rockchip, Datasheet, 2022.

[31] Billie F. Spencer, Vedhus Hoskere, and Yasutaka Narazaki, ‘Advances in Computer Vision-Based Civil Infrastructure Inspection and Monitoring’, *Engineering*, vol. 5, no. 2, pp. 199–222, Apr. 2019. DOI: 10.1016/j.eng.2018.11.030

[32] Nirmala Murali, ‘Image Classification vs Semantic Segmentation vs Instance Segmentation’, *Medium*. 29-Apr-2021 [Online]. Available: https://nirmalamurali.medium.com/image-classification-vs-semantic-segmentation-vs-instance-segmentation-625c33a08d50. [Accessed: 17-May-2025]

[33] Ravpreet Kaur and Sarbjeet Singh, ‘A comprehensive review of object detection with deep learning’, *Digit. Signal Process.*, vol. 132, p. 103812, Jan. 2023. DOI: 10.1016/j.dsp.2022.103812

[34] Megha Shroff, ‘Know your Neural Network architecture more by understanding these terms’, *Medium*. 16-May-2023 [Online]. Available: https://medium.com/@shroffmegha6695/know-your-neural-network-architecture-more-by-understanding-these-terms-67faf4ea0efb. [Accessed: 25-Feb-2025]

[35] ‘Object Detection: Models, Architectures & Tutorial [2024]’. [Online]. Available: https://www.v7labs.com/blog/object-detection-guide. [Accessed: 25-Feb-2025]

[36] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao, ‘YOLOv4: Optimal Speed and Accuracy of Object Detection’. arXiv, 23-Apr-2020 [Online]. DOI: 10.48550/arXiv.2004.10934

[37] Van Vung Pham, *Hands-on computer vision with Detectron2: develop object detection and segmentation models with a code and visualization approach*, 1st ed. Birmingham, England ; Packt, 2023, ISBN: 978-1-80056-660-6.

[38] Tingting Liang, Xiaojie Chu, Yudong Liu, Yongtao Wang, Zhi Tang, Wei Chu, Jingdong Chen, and Haibin Ling, ‘CBNet: A Composite Backbone Network Architecture for Object Detection’, *IEEE Trans. Image Process.*, vol. 31, pp. 6893–6906, 2022. DOI: 10.1109/TIP.2022.3216771

[39] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, ‘ImageNet: A large-scale hierarchical image database’, in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255 [Online]. DOI: 10.1109/CVPR.2009.5206848

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, ‘Deep Residual Learning for Image Recognition’. arXiv, 10-Dec-2015 [Online]. DOI: 10.48550/arXiv.1512.03385

[41] Karen Simonyan and Andrew Zisserman, ‘Very Deep Convolutional Networks for Large-Scale Image Recognition’. arXiv, 10-Apr-2015 [Online]. DOI: 10.48550/arXiv.1409.1556

[42] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam, ‘MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications’. arXiv, 17-Apr-2017 [Online]. DOI: 10.48550/arXiv.1704.04861

[43] Joseph Redmon and Ali Farhadi, ‘YOLOv3: An Incremental Improvement’. arXiv, 08-Apr-2018 [Online]. DOI: 10.48550/arXiv.1804.02767

[44] Rakkshab Iyer, Kevin Bhensdadiya, and Priyansh Ringe, ‘Comparison of YOLOv3, YOLOv5s and MobileNet-SSD V2 for Real-Time Mask Detection’, *Int. J. Res. Eng. Technol.*, pp. 2395–0056, Jul. 2021.

[45] Taylan ateş, ‘How To Design An Object Detector Part 2: Designing Neck’, *Medium*. 07-Mar-2023 [Online]. Available: https://medium.com/@taylan.ates417311/how-to-design-an-object-detector-part-2-designing-neck-c920998fc732. [Accessed: 25-Feb-2025]

[46] Mingxing Tan, Ruoming Pang, and Quoc V. Le, ‘EfficientDet: Scalable and Efficient Object Detection’. arXiv, 27-Jul-2020 [Online]. DOI: 10.48550/arXiv.1911.09070

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, ‘Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition’, vol. 8691, 2014, pp. 346–361 [Online]. DOI: 10.1007/978-3-319-10578-9_23

[48] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille, ‘DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs’. arXiv, 12-May-2017 [Online]. DOI: 10.48550/arXiv.1606.00915

[49] Gaudenz Boesch, ‘FCOS: Fully Convolutional One-Stage Object Detection’, *viso.ai*, 28-May-2024. [Online]. Available: https://viso.ai/deep-learning/fcos-object-detection/. [Accessed: 26-Feb-2025]

[50] Faris Kateb, Muhammad Mostafa Monowar, Md. Abdul Hamid, Abu Ohi, and M. Ph. D., ‘FruitDet: Attentive Feature Aggregation for Real-Time Fruit Detection in Orchards’, *Agronomy*, vol. 11, p. 2440, Nov. 2021. DOI: 10.3390/agronomy11122440

[51] S. Nikhileswara Rao, 'YOLOv11 Explained: Next-Level Object Detection with Enhanced Speed and Accuracy', *Medium*. 22-Oct-2024 [Online]. Available: https://medium.com/@nikhil-rao-20/yolov11-explained-next-level-object-detection-with-enhanced-speed-and-accuracy-2dbe2d376f71. [Accessed: 25-Feb-2025]

[52] 'Detection Head'. [Online]. Available: https://www.ultralytics.com/glossary/detection-head. [Accessed: 26-Feb-2025]

[53] *C4W3L08 Anchor Boxes*. 2017 [Online]. Available: https://www.youtube.com/watch?v=RTlwl2bv0Tg. [Accessed: 26-Feb-2025]

[54] Junhyung Kang, Shahroz Tariq, Han Oh, and Simon Woo, 'A Survey of Deep Learning-Based Object Detection Methods and Datasets for Overhead Imagery', *IEEE Access*, vol. 10, pp. 1–1, Jan. 2022. DOI: 10.1109/ACCESS.2022.3149052

[55] SharkYun, 'Computer Vision — Object Detection, One-Stage vs Two-Stage', *Medium*. 10-Nov-2024 [Online]. Available: https://sharkyun.medium.com/computer-vision-object-detection-one-stage-vs-two-stage-b05dbff88195. [Accessed: 26-Feb-2025]

[56] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao, 'YOLOv4: Optimal Speed and Accuracy of Object Detection'. arXiv, 23-Apr-2020 [Online]. DOI: 10.48550/arXiv.2004.10934

[57] Momina Liaqat Ali and Zhou Zhang, 'The YOLO Framework: A Comprehensive Review of Evolution, Applications, and Benchmarks in Object Detection', *Computers*, vol. 13, no. 12, p. 336, Dec. 2024. DOI: 10.3390/computers13120336

[58] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg, 'SSD: Single Shot MultiBox Detector', vol. 9905, 2016, pp. 21–37 [Online]. DOI: 10.1007/978-3-319-46448-0_2

[59] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik, 'Rich feature hierarchies for accurate object detection and semantic segmentation'. arXiv, 22-Oct-2014 [Online]. DOI: 10.48550/arXiv.1311.2524

[60] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, 'Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks'. arXiv, 06-Jan-2016 [Online]. DOI: 10.48550/arXiv.1506.01497

[61] Jatin Prakash, 'Non Maximum Suppression: Theory and Implementation in PyTorch', 02-Jun-2021. [Online]. Available: https://learnopencv.com/non-maximum-suppression-theory-and-implementation-in-pytorch/. [Accessed: 17-May-2025]

[62] Yian Zhao, Wenyu Lv, Shangliang Xu, Jinman Wei, Guanzhong Wang, Qingqing Dang, Yi Liu, and Jie Chen, 'DETRs Beat YOLOs on Real-time Object Detection'. arXiv, 03-Apr-2024 [Online]. DOI: 10.48550/arXiv.2304.08069

[63] Wenyu Lv, Yian Zhao, Qinyao Chang, Kui Huang, Guanzhong Wang, and Yi Liu, 'RT-DETRv2: Improved Baseline with Bag-of-Freebies for Real-Time Detection Transformer'. arXiv, 24-Jul-2024 [Online]. DOI: 10.48550/arXiv.2407.17140

[64] Nidhal Jegham, Chan Young Koh, Marwan Abdelatti, and Abdeltawab Hendawi, 'Evaluating the Evolution of YOLO (You Only Look Once) Models: A Comprehensive Benchmark Study of YOLO11 and Its Predecessors'. arXiv, 31-Oct-2024 [Online]. DOI: 10.48550/arXiv.2411.00201

[65] Shihan Liu, Junlin Zha, Jian Sun, Zhuo Li, and Gang Wang, 'EdgeYOLO: An Edge-Real-Time Object Detector'. arXiv, 15-Feb-2023 [Online]. DOI: 10.48550/arXiv.2302.07483

[66] Shreyanil Kar, 'OBJECT DETECTION USING VISION TRANSFORMED EFFICIENTDET', p. 7400741 Bytes, 2023. DOI: 10.25394/PGS.23481344.V1

[67] 'Unable to export RT-DETR model to TFLite or EdgeTPU · Issue #14398 · ultralytics/ultralytics', *GitHub*. [Online]. Available: https://github.com/ultralytics/ultralytics/issues/14398. [Accessed: 17-May-2025]

[68] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár, 'Microsoft COCO: Common Objects in Context'. arXiv, 21-Feb-2015 [Online]. DOI: 10.48550/arXiv.1405.0312

[69] Shweta Singh, Aayan Yadav, Jitesh Jain, Humphrey Shi, Justin Johnson, and Karan Desai, 'Benchmarking Object Detectors with COCO: A New Path Forward'. arXiv, 27-Mar-2024 [Online]. DOI: 10.48550/arXiv.2403.18819

[70] Glenn Jocher, Jing Qiu, and Ayush Chaurasia, 'Ultralytics YOLO'. Jan-2023 [Online]. Available: https://github.com/ultralytics/ultralytics. [Accessed: 08-May-2025]

[71] Uday Kulkarni, S. M. Meena, Sunil V. Gurlahosur, Pratiksha Benagi, Atul Kashyap, Ayub Ansari, and Vinay Karnam, 'AI Model Compression for Edge Devices Using Optimization Techniques', in *Modern Approaches in Machine Learning and Cognitive Science: A Walkthrough*, vol. 956, V. K. Gunjan and J. M. Zurada, Eds. Cham: Springer International Publishing, 2021, pp. 227–240 [Online]. DOI: 10.1007/978-3-030-68291-0_17

[72] Jeonghun Lee and Kwang-il Hwang, 'YOLO with adaptive frame control for real-time object detection applications', *Multimed. Tools Appl.*, vol. 81, no. 25, pp. 36375–36396, Oct. 2022. DOI: 10.1007/s11042-021-11480-0

[73] G Di Cecio, A Manco, and G Gigante, 'On-board drone classification with Deep Learning and System-on-Chip implementation', *J. Phys. Conf. Ser.*, vol. 2716, no. 1, p. 012059, Mar. 2024. DOI: 10.1088/1742-6596/2716/1/012059

[74] Mohamed Ahmed, 'Symmetric vs. Asymmetric Quantization: Free AI Course'. 03-Jan-2025 [Online]. Available: https://aifordevelopers.io/symmetric-vs-asymmetric-quantization/. [Accessed: 17-May-2025]

[75] Maarten Grootendorst, 'A Visual Guide to Quantization', 19-Feb-2024. [Online]. Available: https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization. [Accessed: 14-May-2025]

[76] Jahid Hasan, 'Optimizing Large Language Models through Quantization: A Comparative Analysis of PTQ and QAT Techniques'. arXiv, 09-Nov-2024 [Online]. DOI: 10.48550/arXiv.2411.06084

[77] 'Rockchip User Guide RKNN Toolkit', Toybrick, User Guide V1.2.1.

[78] ‘X-CUBE-AI Data Brief’, STMicroelectronics, Data Brief DB3788-Rev 11, Dec. 2024.

[79] ‘Hailo Dataflow Compiler User Guide’, Hailo Technologies, User Guide Release 3.27.0, Mar. 2024.

[80] ‘i.MX Machine Learning User’s Guide’, NXP Semiconductors, User Guide UG10166 Rev. LF6.6.52_2.2.0, Jan. 2025.

[81] Jayoung Lee, Pengcheng Wang, Ran Xu, Venkat Dasari, Noah Weston, Yin Li, Saurabh Bagchi, and Somali Chaterji, ‘Benchmarking Video Object Detection Systems on Embedded Devices under Resource Contention’, in *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, Virtual WI USA, 2021, pp. 19–24 [Online]. DOI: 10.1145/3469116.3470010

[82] Henrique Vedoveli, ‘Metrics Matter: A Deep Dive into Object Detection Evaluation’, *Medium*. 15-Sep-2023 [Online]. Available: https://medium.com/@henriquevedoveli/metrics-matter-a-deep-dive-into-object-detection-evaluation-ef01385ec62. [Accessed: 03-Mar-2025]

[83] ‘Intersection over Union (IoU) for object detection - PyImageSearch’. [Online]. Available: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/. [Accessed: 03-Mar-2025]

[84] ‘Mean Average Precision (mAP) Explained: Everything You Need to Know’. [Online]. Available: https://www.v7labs.com/blog/mean-average-precision. [Accessed: 03-Mar-2025]

[85] Edward Humes, Mozhgan Navardi, and Tinoosh Mohsenin, ‘Squeezed Edge YOLO: Onboard Object Detection on Edge Devices’. arXiv, 18-Dec-2023 [Online]. DOI: 10.48550/arXiv.2312.11716

[86] Yunjie Tian, Qixiang Ye, and David Doermann, ‘YOLOv12: Attention-Centric Real-Time Object Detectors’, *ArXiv Prepr. ArXiv250212524*, 2025.

[87] Ultralytics, ‘YOLO11 🚀 NEW’. [Online]. Available: https://docs.ultralytics.com/models/yolo11. [Accessed: 20-May-2025]

[88] Chien-Yao Wang and Hong-Yuan Mark Liao, ‘YOLOv9: Learning What You Want to Learn Using Programmable Gradient Information’, *ArXiv Prepr. ArXiv240213616*, 2024.

[89] ‘YOLOv9: A Leap Forward in Object Detection Technology’. [Online]. Available: https://docs.ultralytics.com/models/yolov9/. [Accessed: 20-May-2025]

[90] Glenn Jocher, Ayush Chaurasia, and Jing Qiu, ‘Ultralytics YOLOv8’. 2023 [Online]. Available: https://github.com/ultralytics/ultralytics

[91] ‘GitHub - hailo-ai/hailo_model_zoo: The Hailo Model Zoo includes pre-trained models and a full building and evaluation environment’. [Online]. Available: https://github.com/hailo-ai/hailo_model_zoo. [Accessed: 20-May-2025]

[92] ‘MultimediaTechLab/YOLO: An MIT License of YOLOv9, YOLOv7, YOLO-RD’. [Online]. Available: https://github.com/MultimediaTechLab/YOLO. [Accessed: 08-May-2025]

[93] ‘What is GStreamer?’ [Online]. Available: https://gstreamer.freedesktop.org/documentation/application-development/introduction/gstreamer.html?gi-language=c. [Accessed: 09-May-2025]

Appendix A: Performance Benchmark Result of NXP Platform

Stream Resolution		Model	YOLOv8n			YOLOv8s			YOLOv8m		
		Input Size	640x640	416x416	320x320	640x640	416x416	320x320	640x640	416x416	320x320
720p		Inference Latency (ms)	60.2	26.4	16	109	48.4	28.9	205	86.9	52.5
		CPU Utilization (%)	46.7	55.6	61.4	28.2	40.4	55.8	16.7	24.9	34.1
		CPU Temperature (°C)	79.4	79.3	77.4	77.4	79.1	82.1	77.7	79.5	80.7
		Inference RAM Usage (MB)	363	339	345	369	360	364	408	443	411
		Total RAM Usage (MB)	878	842	819	888	872	869	777	822	782
		End to End Latency (ms)	230	130	100	380	185	120	650	300	190
		Throughput (FPS)	15	28	37	8	19	32	4	11	18
		Inference Power Usage (Watt)	2.81	2.92	2.81	2.62	2.86	3.13	2.58	2.74	2.93
1080p		CPU Utilization (%)	55.5	62.9	64.2	37.6	58.1	63.9	21.8	36.8	55
		CPU Temperature (°C)	80	78.2	73.8	78.7	82	81.2	78.4	80.9	82.2
		Inference RAM Usage (MB)	412	403	397	427	419	420	461	502	477
		Total RAM Usage (MB)	966	960	771	970	959	960	860	903	956
		End to End Latency (ms)	290	200	180	410	250	200	700	340	220
		Throughput (FPS)	13	19	24	8	17	21	4	10	17
		Inference Power Usage (Watt)	56.2	22.3	12	105	44.4	24.9	201	82.9	48.5

YOLOv9s			YOLOv9m		
640x640	416x416	320x320	640x640	416x416	320x320
153	71.4	42.2	244	107	63
30.5	39.5	50.8	20.8	28.5	35.7
77	78.4	79.2	77.6	79	79.9
907	832	819	720	685	670
1312	1234	1223	1115	1070	1060
610	300	180	900	410	250
6	12	21	3	8	14
2.45	2.62	2.8	2.57	2.65	2.76
38.9	54.3	61.8	26.2	38.3	51.3
78.1	80.5	79.5	78.8	80	81.4
953	888	875	771	742	726
1380	1317	1311	1188	1164	1138
640	330	250	940	450	290
5	12	17	3	8	13
149	67.4	38.2	240	103	59

Appendix B: Performance Benchmark Result of Hailo Platform

Stream Resolution	Model	YOLOv8n			YOLOv8s			YOLOv8m		
	Input Size	640x640	416x416	320x320	640x640	416x416	320x320	640x640	416x416	320x320
720p	Inference Latency (ms)	15.7	9.59	7.51	22.9	15.9	12.6	46.3	30.9	28
	CPU Utilization (%)	75.4	74.1	73.3	64.3	69.1	68.4	24.9	34.9	32.2
	CPU Temperature (°C)	63.2	64.3	63.2	63.9	64.2	63.1	57.5	57.7	57.5
	NPU Temperature (°C)	41.6	40.4	39.1	47.4	43.7	41.9	50.7	45.8	43.6
	Inference RAM Usage (MB)	259	236	233	277	259	256	320	304	298
	Total RAM Usage (MB)	841	822	819	852	838	839	887	875	873
	End to End Latency (ms)	50	45	40	50	50	35	60	40	40
	Throughput (FPS)	62	70	74	50	60	64	24	34	36
	Inference Power Usage (Watt)	5.26	5.16	5.07	5.46	5.36	5.13	3.48	3.92	3.59
1080p	CPU Utilization (%)	65.5	64.4	72.9	71.5	68.7	64.9	40.2	55.2	64.6
	CPU Temperature (°C)	62.8	63.1	62.9	64.9	64.4	63	58.7	61.9	64
	NPU Temperature (°C)	40.1	39.1	38.4	44.8	41.5	40.1	48.2	45.8	44.2
	Inference RAM Usage (MB)	296	279	273	329	312	301	366	355	352
	Total RAM Usage (MB)	898	886	879	925	916	905	955	949	950
	End to End Latency (ms)	70	60	55	70	65	55	70	70	70
	Throughput (FPS)	38	41	43	35	38	40	24	33	36
	Inference Power Usage (Watt)	5.04	4.94	5.16	5.76	5.33	5.13	4.55	5.18	5.4

YOLOv9s			YOLOv9m		
640x640	416x416	320x320	640x640	416x416	320x320
39.4	26.1	20.6	55.1	37.7	31.9
57.6	63.8	62.4	28.7	37.2	41.5
62.2	63.4	62.3	57.5	57.9	58.6
44	42.8	41.4	47.4	44.7	43.4
288	261	248	301	266	260
879	854	842	881	850	844
80	70	50	80	60	60
29	42	50	20	30	36
4.68	4.91	4.72	3.7	3.9	3.97
53.9	65.8	65.4	45.4	63.7	69.7
62.2	64.1	63.5	59.7	63.3	69.1
42.6	41.4	40.5	46.7	45.3	43.8
311	314	299	336	345	309
924	929	916	939	921	913
90	85	70	95	80	80
23	31	35	20	29	33
4.73	5.07	5.07	4.62	5.33	5.4

Appendix C: Model Evaluation Results on NXP Platform

Model	YOLOv8n			YOLOv8s			YOLOv8m	
Input Size	640x640	416x416	320x320	640x640	416x416	320x320	640x640	416x416
Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100]	0.32	0.28	0.24	0.39	0.35	0.31	0.44	0.4
Average Precision (AP) @[IoU=0.50 | area= all | maxDets=100]	0.47	0.42	0.37	0.57	0.52	0.46	0.62	0.58
Average Precision (AP) @[IoU=0.75 | area= all | maxDets=100]	0.34	0.3	0.26	0.43	0.38	0.33	0.47	0.43
Average Precision (AP) @[IoU=0.50:0.95 | area= small | maxDets=100]	0.12	0.07	0.04	0.18	0.11	0.08	0.22	0.15
Average Precision (AP) @[IoU=0.50:0.95 | area=medium | maxDets=100]	0.35	0.3	0.23	0.44	0.39	0.33	0.49	0.45
Average Precision (AP) @[IoU=0.50:0.95 | area= large | maxDets=100]	0.46	0.48	0.46	0.56	0.58	0.55	0.62	0.63
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 1]	0.27	0.24	0.22	0.31	0.29	0.26	0.33	0.31
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 10]	0.43	0.38	0.34	0.5	0.45	0.4	0.54	0.49
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets=100]	0.48	0.42	0.37	0.54	0.48	0.43	0.58	0.53
Average Recall (AR) @[IoU=0.50:0.95 | area= small | maxDets=100]	0.26	0.14	0.09	0.32	0.2	0.14	0.38	0.26
Average Recall (AR) @[IoU=0.50:0.95 | area=medium | maxDets=100]	0.53	0.47	0.41	0.6	0.55	0.5	0.65	0.61
Average Recall (AR) @[IoU=0.50:0.95 | area= large | maxDets=100]	0.63	0.64	0.62	0.71	0.71	0.69	0.74	0.75

	YOLOv9s			YOLOv9m		
320x320	640x640	416x416	320x320	640x640	416x416	320x320
0.36	0.29	0.28	0.24	0.48	0.44	0.4
0.53	0.43	0.41	0.38	0.65	0.61	0.56
0.39	0.32	0.29	0.25	0.53	0.48	0.43
0.11	0.2	0.13	0.08	0.31	0.21	0.16
0.4	0.25	0.31	0.3	0.54	0.5	0.46
0.61	0.53	0.45	0.37	0.64	0.65	0.63
0.29	0.28	0.27	0.24	0.36	0.34	0.32
0.45	0.49	0.45	0.38	0.59	0.54	0.49
0.49	0.55	0.49	0.39	0.63	0.57	0.53
0.19	0.34	0.21	0.11	0.46	0.32	0.23
0.56	0.6	0.56	0.45	0.63	0.66	0.62
0.73	0.68	0.7	0.61	0.76	0.77	0.76

Appendix D: Model Evaluation Results on Hailo Platform

Model	YOLOv8n			YOLOv8s			YOLOv8m	
Input Size	640x640	416x416	320x320	640x640	416x416	320x320	640x640	416x416
Average Precision (AP) @[IoU=0.50:0.95 | area= all | maxDets=100]	0.35	0.31	0.27	0.43	0.39	0.35	0.48	0.45
Average Precision (AP) @[IoU=0.50 | area= all | maxDets=100]	0.51	0.45	0.4	0.61	0.55	0.5	0.66	0.61
Average Precision (AP) @[IoU=0.75 | area= all | maxDets=100]	0.38	0.33	0.29	0.47	0.42	0.37	0.52	0.48
Average Precision (AP) @[IoU=0.50:0.95 | area= small | maxDets=100]	0.17	0.11	0.07	0.24	0.16	0.11	0.29	0.21
Average Precision (AP) @[IoU=0.50:0.95 | area=medium | maxDets=100]	0.39	0.33	0.28	0.48	0.43	0.38	0.53	0.51
Average Precision (AP) @[IoU=0.50:0.95 | area= large | maxDets=100]	0.49	0.52	0.5	0.6	0.61	0.6	0.65	0.66
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 1]	0.3	0.28	0.25	0.34	0.32	0.3	0.37	0.35
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets= 10]	0.5	0.45	0.4	0.56	0.51	0.46	0.6	0.56
Average Recall (AR) @[IoU=0.50:0.95 | area= all | maxDets=100]	0.55	0.48	0.43	0.61	0.55	0.49	0.64	0.59
Average Recall (AR) @[IoU=0.50:0.95 | area= small | maxDets=100]	0.33	0.2	0.13	0.39	0.28	0.18	0.45	0.34
Average Recall (AR) @[IoU=0.50:0.95 | area=medium | maxDets=100]	0.6	0.55	0.48	0.67	0.63	0.57	0.7	0.67
Average Recall (AR) @[IoU=0.50:0.95 | area= large | maxDets=100]	0.72	0.73	0.71	0.77	0.78	0.76	0.79	0.81

	YOLOv9s			YOLOv9m		
320x320	640x640	416x416	320x320	640x640	416x416	320x320
0.41	0.28	0.25	0.22	0.48	0.44	0.4
0.57	0.41	0.38	0.35	0.65	0.61	0.56
0.44	0.3	0.26	0.23	0.53	0.48	0.43
0.16	0.18	0.13	0.08	0.3	0.21	0.15
0.46	0.24	0.28	0.29	0.53	0.5	0.46
0.65	0.53	0.39	0.32	0.65	0.66	0.64
0.33	0.28	0.25	0.23	0.36	0.34	0.32
0.52	0.48	0.43	0.39	0.59	0.54	0.49
0.55	0.53	0.48	0.42	0.63	0.58	0.53
0.25	0.31	0.22	0.13	0.45	0.32	0.23
0.64	0.57	0.55	0.48	0.69	0.66	0.62
0.81	0.68	0.68	0.66	0.76	0.78	0.77

TRITA-EECS-EX-2025:543
Stockholm, Sweden 2025

www.kth.se