UNIVERSITÀ DEGLI STUDI DI PAVIA

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL, COMPUTER AND BIOMEDICAL ENGINEERING

MASTER'S DEGREE IN COMPUTER ENGINEERING

MASTER THESIS

ADVANCED VEHICLE SURVEILLANCE: A REAL-TIME MULTI-CAMERA AI-BASED PERSON ACTIVITY RECOGNITION ON A LOW-POWER EMBEDDED GPU

SORVEGLIANZA DI VEICOLO AVANZATA: UN RICONOSCITORE DI ATTIVITÀ UMANA IN REAL-TIME E MULTI-CAMERA BASATO SU RETI NEURALI SU UNA GPU A BASSO CONSUMO EMBEDDED

Candidate: Riccardo Fiocchi

Supervisor: prof. Francesco Leporati

Co-supervisor: prof. Emanuele Torti

Co-supervisor: dott. Marco Gazzoni

ANNO ACCADEMICO 2024/25

Table of Contents

Introduction

Vision-based perception systems have become central components of modern intelligent systems deployed on edge. Applications such as surveillance, human-machine interaction, and advanced driver assistance increasingly rely on real-time interpretation of visual data streams. [1] In these domains, multi-camera configurations are frequently adopted to enlarge the observable environment, reduce blind spots, and improve robustness through spatial redundancy. [2]

The rapid advancement of deep learning has significantly improved the accuracy of visual understanding tasks, enabling reliable detection and activity recognition in complex environments. [3] However, these improvements come at the cost of increased computational demand. Modern neural network models require substantial parallel processing capability and memory bandwidth, which are readily available in high-performance desktop or cloud environments, but become more challenging when deployment is constrained to embedded platforms.

At the same time, perception pipelines are increasingly expected to operate directly on edge devices rather than relying on remote infrastructure. [4] In such scenarios, multi-camera visual input and deep learning inference must coexist with strict power and resource limitations, making real-time execution a non-trivial systems problem.

Within this context, embedded real-time deployment introduces a more restrictive and technically demanding scenario.

Deploying multi-camera AI pipelines on embedded systems introduces a set of tightly coupled constraints. Compared to desktop-class platforms, embedded devices offer reduced computational resources, lower memory bandwidth, and more limited buffering capacity. At the same time, real-time operation imposes frame deadlines on the order of a few tens of milliseconds, while concurrent processing of multiple camera streams increases scheduling complexity and synchronization overhead.

A naive implementation strategy, such as sequentially processing detections and classifications for each frame and for each camera, can lead to latency accumulation and unstable throughput as workload increases. These conditions make it necessary to investigate not only whether the required models can execute on embedded hardware, but whether the complete pipeline can sustain stable real-time behaviour under realistic multi-stream conditions.

The challenges just described are not purely academic. The increasing integration of vision-based AI systems in safety-critical and high-performance environments demands robust embedded solutions capable of sustained real-time operation. Industrial stakeholders in sectors such as automotive engineering actively investigate the feasibility of deploying multi-camera perception pipelines on compact, low-power computational units.

This thesis was developed in collaboration between the University of Pavia [5] and the Nardò Technical Center [6], in coordination with Porsche Engineering [7], within a broader context of evaluating

embedded AI perception architectures in an automotive deployment scenario. [1] The objective was not only to implement a working prototype, but to rigorously assess whether a multi-camera person activity recognition pipeline could maintain deterministic real-time performance when deployed on a low-power embedded GPU platform suitable for on-vehicle integration.

By grounding the investigation in a real-world evaluation context, the work addresses practical deployment constraints that extend beyond laboratory conditions, including multi-stream concurrency, system stability and cross-platform integration. In this scenario, low power consumption was not defined through a strict numerical requirement, but remained an important design objective, since automotive edge systems should limit power demand and thermal load as much as possible while sustaining continuous operation on compact embedded hardware.

The problem addressed in this thesis is the design and validation of a real-time multi-camera person activity recognition system deployed on a low-power embedded GPU platform. The system must process multiple concurrent camera streams, perform human detection and action classification, and sustain a target frame rate of 30 frames per second under embedded hardware constraints.

The central question is whether this objective can be achieved in a stable and scalable way under realistic resource limitations, rather than only in isolated or simplified conditions.

The underlying assumption of this thesis is that deterministic real-time performance on such platforms cannot be achieved through model selection alone, but requires coordinated design across the full system stack, including batching strategy, CUDA [8] implementation, concurrency orchestration, and synchronization policy.

Contributions

The main contributions of this thesis are:

- The design and full implementation of a real-time multi-camera person activity recognition pipeline specifically tailored for deployment on a low-power embedded GPU platform;
- The restructuring of detection and action-classification inference to support batched execution, improving throughput under multi-detection and multi-camera conditions;
- The redesign of custom CUDA kernels toward batch-aware three-dimensional grid configurations, aligning the computational structure of the pipeline with the parallel execution model of the GPU;
- The development of a multi-thread and multi-stream execution architecture that decouples acquisition, inference, and visualization stages in order to reduce synchronization bottlenecks;
- The introduction of a non-blocking latest-frame scheduling strategy that discards stale frames, avoids backpressure accumulation, and preserves responsiveness under constrained embedded resources;
- The implementation of a cross-platform workflow supporting x86-based development and ARM-based deployment on the target embedded platform;
- The experimental validation of sustained 30 FPS operation in a three-camera configuration on the target embedded hardware.

Thesis Organization

Chapter 1 introduces the application context and the technological foundations underlying the work, including the evolution from classical exterior monitoring to semantic perception, the role of deep learning in visual recognition, and the relevance of GPU architectures for embedded AI execution. Chapter 2 presents the target hardware platform, the operational requirements of the perception pipeline, and the constraints that define the design space of the system.

Chapter 3 describes the baseline GPU pipeline developed in the initial workstation environment, outlining its structure, implementation choices, and the main limitations that emerge in the original version. Chapter 4 presents the architectural redesign of the system, focusing on batched inference, CUDA kernel adaptations, multi-thread organization, and GPU stream coordination. Chapter 5 discusses the cross-architecture engineering work required to move from the x86 development environment to the target embedded platform and to integrate the runtime with the camera acquisition system.

Chapter 6 describes the final embedded optimization steps, including GPU-based visualization, remote streaming, and the latest-frame scheduling strategy adopted to preserve responsiveness under constrained resources. Chapter 7 reports the experimental validation of the final embedded system under both single-camera and multi-camera conditions and includes comparison with the baseline implementation. Chapter 8 develops a broader system-level analysis, discussing scaling with the number of detections and active cameras, comparison with desktop execution, and the resulting deployment trade-offs. Finally, Chapter 9 summarizes the main results of the thesis, discusses the practical lessons emerging from the deployment process, and outlines possible directions for future work.

1. Context and Technological Foundations

1.1 Vision-Based Exterior Perception in Modern Vehicles

1.1.1 Evolution from Motion-Based Monitoring to Semantic Perception

Modern vehicles increasingly integrate camera-based systems for environmental perception beyond traditional driving assistance. [2] While early electronic security systems relied primarily on vibration sensors or proximity alarms, contemporary vehicles incorporate multiple exterior cameras capable of continuously monitoring the surroundings. These systems serve purposes ranging from parking assistance and surround-view visualization to event-based recording and remote monitoring functionalities. [1]

A prominent example of exterior monitoring is motion-triggered recording systems, such as those implemented in commercial electric vehicles under so-called “sentinel” modes.[9] In such configurations, cameras record and store footage when motion is detected in the vicinity of the vehicle. This approach enhances post-event analysis by allowing the vehicle owner to review recorded sequences in case of vandalism or suspicious activity. However, motion-based triggering mechanisms operate at a purely reactive level: any detected movement within the camera field of view may result in recording and storage, regardless of its semantic relevance.

While effective for basic surveillance, motion-only strategies present inherent limitations. [10] Urban environments naturally contain continuous pedestrian and vehicle movement, which may lead to large volumes of recorded data with limited value. Furthermore, motion-based systems do not differentiate between benign presence, such as a pedestrian walking near the vehicle and potentially harmful interactions, such as intentional contact or vandalism. [11] As a result, owners may be required to manually review extensive footage to identify relevant events.

These limitations motivate the transition from motion-based monitoring to semantic perception. Instead of reacting to raw movement, an intelligent system should be capable of identifying relevant object classes, such as humans, and interpreting their actions. Distinguishing between observation, proximity, and direct interaction enables more selective responses and gives semantic meaning to the monitored scene.

This evolution also changes the nature of the perception task: the system is no longer required only to detect motion, but to understand what is present in the scene and what is happening. The implications of this shift, both in terms of algorithmic approach and computational cost, are developed in the following sections.

1.1.2 Multi-Camera Architectures for Exterior Coverage

Exterior vehicle monitoring systems typically rely on multiple cameras positioned to provide near-complete coverage of the vehicle perimeter. [2] Unlike single-lens solutions with wide-angle optics, multi-camera configurations are adopted to eliminate blind zones and ensure reliable visibility of all sides of the vehicle. A single 180° field-of-view camera, for instance, cannot simultaneously monitor the front, rear, and lateral regions of a vehicle. Comprehensive coverage therefore requires distributed sensing units strategically positioned to observe different sectors.

In practice, modern vehicles may integrate four or more cameras to achieve 360° visibility. While originally introduced to support surround-view parking assistance, such architectures provide a natural foundation for continuous exterior monitoring. However, expanding from visualization to semantic understanding significantly alters system requirements.

Each additional camera stream increases both input bandwidth and processing demand. Even at moderate resolutions and frame rates, the data throughput becomes substantial. As an illustrative example, a single 1920×1080 stream at 30 frames per second generates approximately:

$$1920 \times 1080 \times 3 \text{ bytes} \times 30 \approx 186 \text{ MB/s} \quad (1)$$

of raw RGB data before compression or processing. Multiplying this by several concurrent camera streams rapidly elevates memory bandwidth requirements and increases the volume of data that must be processed in real-time.

While embedded systems often incorporate compression pipelines and hardware-accelerated pre-processing stages, semantic interpretation still requires per-frame inference operations. Thus, multi-camera coverage, although essential for spatial completeness, inherently amplifies computational pressure when combined with AI-based perception tasks.

1.1.3 Emerging Requirements: Selective Response and Privacy-Aware Monitoring

Beyond spatial coverage, next-generation exterior monitoring systems are expected to move from passive recording toward intelligent, selective response strategies. [11] The objective is not merely to capture all motion events, but to identify situations that are semantically relevant and trigger appropriate actions accordingly.

In this context, distinguishing between benign and harmful interactions becomes critical. A pedestrian walking past the vehicle, a person briefly observing it, or a group gathering nearby may generate motion but do not necessarily require intervention or permanent recording. Conversely, direct physical

interaction, such as attempted intrusion, vandalism, or impact, demands immediate attention and potentially persistent documentation.

This differentiation introduces two intertwined challenges. First, the system must reliably detect and track humans across multiple camera views. Second, it must classify observed behaviours to determine the appropriate response. These responses may include selective notification, event flagging or conditional privacy considerations.

From this perspective, semantic interpretation is valuable not only because it improves scene understanding, but because it enables differentiated system responses. The relevant distinction is no longer simply whether motion occurred, but whether a person is present, how close the person is to the vehicle, and whether the observed behaviour suggests benign presence or direct interaction. This motivates the combined use of human detection and activity recognition within the monitoring pipeline.

1.2 From Classical Image Processing to Deep Learning Based Recognition

1.2.1 Limitations of Classical Image Processing and Shallow Learning

Early computer vision systems were built upon deterministic image processing pipelines.[12] These systems relied on manually engineered operations such as edge detection, gradient operators, corner detectors, colour histograms, and texture descriptors. The extracted features were then fed into shallow classifiers, including support vector machines or decision trees.

Such pipelines followed a two-stage paradigm in which feature extraction was designed by humans, while classification was learned from data.

The computational behaviour of these systems was predictable and relatively efficient. Most operations consisted of linear filtering, convolution with small kernels, thresholding, or pixel-wise transformations, with complexity scaling linearly with image resolution. For embedded systems, this determinism was advantageous.

However, the core limitation of these approaches was representational. The system could only recognize patterns explicitly encoded in the handcrafted descriptors. In other words, performance was bounded by the designer's ability to anticipate which visual structures would be relevant.

As visual tasks increased in complexity, involving cluttered backgrounds, viewpoint changes, occlusion, and varying illumination, manually engineered features proved insufficient. The gap was not primarily in classification algorithms, but in the expressiveness of the representation itself.

This representational bottleneck motivated the transition towards learned hierarchical features. [3]

1.2.2 Deep Neural Networks and Hierarchical Representation Learning

Deep neural networks addressed the core limitation of classical pipelines by unifying feature extraction and classification within a single trainable model.

Rather than manually specifying which features to compute, deep networks learn intermediate representations directly from data through stacked nonlinear transformations. A deep feedforward network can be expressed as a composition:

$$h^{(l)} = g\left(W^{(l)}h^{(l-1)} + b^{(l-1)}\right) \quad (2)$$

where $h^{(l-1)}$denotes the representation produced by the previous layer, $W^{(l)}$and $b^{(l)}$are the weights and bias of layer l, and $g(\cdot)$is the corresponding nonlinear activation function. Successive layers progressively transform the input into higher-level abstractions.

The key insight is that depth enables hierarchical feature learning. Early layers capture simple local patterns; intermediate layers encode increasingly complex structures; deeper layers represent semantic concepts.

While the idea of deep networks existed for decades, their practical usability was historically constrained by vanishing and exploding gradients, limited computational power, and the lack of large, labelled datasets.

The convergence of improved training strategies, large-scale datasets, and massively parallel hardware (GPUs) enabled deep architectures to surpass handcrafted methods in large-scale recognition tasks.

The key consequence of this formulation is that feature extraction becomes part of the learning process itself. Instead of relying on handcrafted descriptors, the network automatically learns hierarchical visual representations directly from data.

1.2.3 Convolutional Neural Networks and Spatial Inductive Bias

Images possess strong spatial structure. To exploit this property and scale well with input resolution, CNNs were introduced. Convolutional Neural Networks (CNNs) introduce two key architectural principles, namely local receptive fields and weight sharing. [13]

Instead of connecting each neuron to every pixel, convolutional layers apply learnable kernels over local regions. A two-dimensional convolution between feature map and kernel can be expressed as:

$$Y(i,j) = \sum_{u=0}^{K-1} \sum_{v=0}^{K-1} W(u,v) X(i+u, j+v) \tag{3}$$

Where X is the input feature map, W the convolution kernel and $Y(i,j)$ the output value computed at spatial position (i, j). The indices u and v iterate over the kernel support.

In practice, convolution operates over multi-channel inputs, producing multiple output feature maps. The parameter count of a convolutional layer is:

$$K^2 \cdot C_{in} \cdot C_{out} \tag{4}$$

Here, K denotes the kernel size, while C_{in} and C_{out} represent the number of input and output channels.

Its computational cost scales as:

$$\text{FLOPs} \approx 2 \cdot H \cdot W \cdot C_{in} \cdot C_{out} \cdot K^2 \tag{5}$$

In this expression, H and W denote the spatial dimensions of the output feature map, so the total cost depends on how many spatial positions the convolution must evaluate, in addition to kernel size and channel dimensions. This scaling highlights a fundamental property: increasing depth and channel width expands the representational capacity of the network, allowing progressively richer visual abstractions to be learned.

Stacking convolutional layers enables hierarchical spatial abstraction while maintaining translation equivariance, a property essential for object detection and recognition.

CNNs therefore solve the representational limitations of classical pipelines by learning features end-to-end. In doing so, they fundamentally shift the bottleneck from feature engineering to computation.

1.2.4 Computational Implications for Detection and Instance-Level Analysis

Image classification requires a single forward pass producing a global prediction. Object detection extends the problem to simultaneous localization and classification across spatial positions. Detection networks perform dense prediction over feature maps, evaluating candidate regions and regressing bounding boxes alongside class probabilities. [14]

This dense inference significantly increases computational demand.

Furthermore, when semantic interpretation is required at the object level, such as per-instance action classification, additional processing must be applied to each detected region. The overall computational load scales with:

$$\text{Load} \propto N_{cam} \cdot N_{inst} \cdot C_{model} \cdot f \tag{6}$$

where N_c is the number of camera streams, N_i the number of instances per frame, C_{net} the computational cost of the employed networks, and f the target frame rate.

When these tasks are combined within a multi-camera system, the computational demand increases further, since the same processing logic must be sustained across several concurrent visual streams. As a result, the problem is not limited to model accuracy alone, but also involves the availability of sufficient computational parallelism and memory bandwidth to execute the workload within real-time constraints. This makes parallel computing architectures, and especially GPU-based systems, particularly relevant for the class of applications considered in this thesis.

1.3 Embedded GPU Architectures for Edge AI

1.3.1 Massively Parallel Architectures and the SIMT Model

Modern computing architectures can be broadly distinguished by how they exploit parallelism. Traditional central processing units (CPUs) are optimized for low-latency execution of sequential and branching workloads. They typically consist of a small number of complex cores, each capable of sophisticated instruction scheduling, branch prediction, and out-of-order execution.

In contrast, graphics processing units (GPUs) are designed for high-throughput execution of massively parallel workloads. Rather than emphasizing complex control logic per core, GPUs dedicate silicon area to a large number of simpler arithmetic units capable of executing the same instruction across many data elements simultaneously.

This execution paradigm is commonly described as Single Instruction, Multiple Thread (SIMT) [15]. Under SIMT, groups of threads execute identical instructions in parallel while operating on different data. Threads are organized hierarchically: individual threads perform elementary computations, threads are grouped into blocks that may cooperate via fast on-chip memory, and blocks are organized into a grid representing a complete parallel execution instance.

Within the hardware, threads are scheduled in small groups that execute in lockstep. This structure enables thousands of concurrent threads to be active simultaneously, allowing the architecture to hide memory latency by switching execution among ready thread groups.

The SIMT model is particularly effective for workloads that can be decomposed into many independent arithmetic operations. High arithmetic intensity and regular memory access patterns are key factors in achieving efficient utilization of such architectures.

1.3.2 Memory Hierarchy and Throughput-Oriented Design

While GPU architectures provide substantial computational throughput, performance is strongly influenced by memory behaviour. GPUs incorporate a hierarchical memory structure designed to balance capacity, latency, and bandwidth. [16]

At the highest level, global memory provides large storage capacity but with relatively high access latency. On-chip memories, such as shared memory and registers, offer significantly lower latency and higher bandwidth but are limited in size. Hardware caches further mitigate repeated access to global memory by exploiting spatial and temporal locality.

Throughput-oriented architectures rely on overlapping computation and memory access. When one group of threads stalls due to memory latency, another ready group can be scheduled. This latency-hiding mechanism is effective only when sufficient parallel work exists and memory accesses are structured to maximize bandwidth efficiency.

Efficient utilization therefore depends not only on arithmetic capability but also on access patterns. Coalesced memory access, data reuse through on-chip memory, and balanced workload distribution are central to achieving sustained performance.

1.3.3 Affinity Between Deep Learning and GPU Architectures

The large-scale adoption of deep neural networks coincided with the widespread availability of programmable GPUs. This convergence was not accidental: the computational structure of deep learning workloads aligns closely with massively parallel architectures. [3]

Deep learning workloads are dominated by dense linear algebra operations, including convolutions, matrix multiplications, and vectorized nonlinear transformations. These operations consist of repetitive arithmetic computations applied over large data tensors with limited control-flow divergence. Such structure aligns naturally with SIMT execution models.

The computational cost expressions 3, 4 and 5 introduced in Section 1.2 reveal that convolutional layers involve large numbers of multiply-accumulate operations across spatial positions and channels. Each output element can be computed independently given the corresponding receptive field, enabling fine-grained parallel decomposition.

As a result, GPUs provide the parallel arithmetic throughput required to evaluate deep networks within practical time constraints. Without massively parallel hardware, the computational burden of modern neural networks would render real-time inference infeasible for many applications.

This hardware–algorithm alignment explains why GPU acceleration has become foundational to both deep learning training and inference.

1.3.4 Embedded GPU Platforms and Edge Deployment Constraints

The increasing demand for artificial intelligence in mobile, automotive, and industrial systems has driven the development of embedded GPU platforms. Unlike data center accelerators, embedded devices must operate within strict power and thermal envelopes while maintaining compact form factors suitable for integration into vehicles and edge equipment.

The push toward on-device AI processing arises from several practical considerations, including reduced latency compared to remote processing, improved data privacy, and independence from network connectivity.

Compared to high-performance desktop GPUs, embedded platforms typically provide fewer parallel processing units, lower memory bandwidth, shared memory resources between CPU and GPU, and configurable power modes that directly influence clock frequencies and throughput.

These constraints transform performance optimization into a system-level engineering challenge. The objective is no longer to maximize absolute throughput, but to sustain predictable behaviour within deadline and lower energy budgets.

For these reasons, embedded GPU platforms represent a particularly suitable compromise for edge AI applications, combining programmable parallel computation with deployment practicality in power-constrained environments. At the same time, their limited computational and memory resources make efficient execution highly dependent on workload organization, data movement control, and scheduling strategy.

In this thesis, these considerations motivate the adoption of an embedded GPU platform as the target execution environment. The specific hardware platform selected for the implementation is introduced in the following chapter.

2. System Constraints and Design Space

2.1 Target Hardware Platform

2.1.1 System-on-Chip Architecture

The target platform for this work is based on the NVIDIA Jetson Xavier NX System-on-Module (SOM) [17], integrated into a custom carrier solution provided by Leopard Imaging [18] for multi-camera automotive applications. [19] The Xavier NX is a heterogeneous system-on-chip (SoC) that integrates general-purpose CPU cores, a CUDA-capable GPU, memory controllers, and specialized accelerators within a single compact module.

The CPU subsystem consists of six NVIDIA Carmel ARMv8.2 cores operating at up to 1.4 GHz, supported by 6 MB of L2 cache and 4 MB of L3 cache. These cores provide the general-purpose execution environment required for operating system services, peripheral coordination, networking, and non-parallel control tasks.

The GPU subsystem is based on NVIDIA's Volta architecture and includes 384 CUDA [8] cores and 48 Tensor Cores. Unlike discrete desktop GPUs, the Xavier NX integrates CPU and GPU components within the same silicon die, sharing access to a unified main memory pool. This integration reduces data transfer latency between CPU and GPU compared to PCIe-based discrete accelerators, but also means that memory bandwidth must be shared across all processing components.

In the adopted Leopard Imaging carrier configuration [19], the platform also provides the physical interface required for multi-camera acquisition. In particular, the carrier exposes six GMSL2 interfaces, corresponding to six 2-lane MIPI CSI-2 camera connections operating at up to 2.5 Gbps per lane. This hardware configuration is relevant because it makes the platform suitable not only for embedded inference, but also for direct integration with synchronized high-bandwidth automotive camera streams.

This heterogeneous architecture enables tightly coupled CPU–GPU cooperation while operating within an embedded power envelope suitable for automotive deployment.

2.1.2 GPU Microarchitecture

The Volta-based GPU [20] integrated in the Xavier NX is organized into six Streaming Multiprocessors (SMs), each containing CUDA cores, registers, and on-chip shared memory. Threads are executed in groups of 32, known as warps, following the SIMT execution model introduced in Chapter 1. [21]

The GPU supports both single-precision (FP32) and half-precision (FP16) floating-point arithmetic. The presence of Tensor Cores enables accelerated mixed-precision matrix operations, which are particularly relevant for deep learning workloads optimized for FP16 execution.

At peak theoretical performance, the Xavier NX delivers approximately:

- ~1.3 TFLOPs FP32;
- ~2.6 TFLOPs FP16 through standard CUDA cores;
- Higher effective throughput when Tensor Cores are utilized for supported operations.

These values represent idealized peak arithmetic throughput under maximum clock conditions. Sustained real-time performance depends not only on arithmetic capability, but also on memory bandwidth, kernel configuration, and power mode settings.

The CUDA programming model [8] exposed by the platform makes it possible to tailor thread organization, memory usage, and execution configuration to the requirements of the embedded workload. In the context of this thesis, this programmability is one of the key reasons why the platform is suitable for custom inference optimization.

2.1.3 Memory Subsystem

The Xavier NX module [17] integrates 8 GB of 128-bit LPDDR4x memory operating at 1600 MHz, providing a theoretical peak memory bandwidth of 51.2 GB/s. This value follows from the effective transfer rate of the memory interface and represents the maximum theoretical data movement rate between DRAM and the memory controller under ideal conditions.

Two distinct aspects of the memory subsystem are relevant in this work: memory **capacity** (8 GB) determines how much data can be stored, whereas memory **bandwidth** (51.2 GB/s) determines how quickly data can be read from or written to memory.

In the present system, memory capacity was sufficient and no out-of-memory conditions were encountered. However, bandwidth remained an important constraint because CPU and GPU share the same DRAM resources in this unified architecture. Camera buffers, intermediate tensors and model parameters therefore compete for the same memory traffic budget.

A useful way to interpret this value is to relate it to the target real-time cycle. At 30 frames per second, one processing interval corresponds to one thirtieth of a second. Under ideal theoretical conditions, this means that at most about:

$$\frac{51.2}{30} \approx 1.7 \text{ GB} \tag{7}$$

could be transferred during a single frame cycle. In practice, the sustained usable bandwidth is lower due to contention, arbitration overhead and access patterns. This makes memory bandwidth a relevant system-level constraint even when total memory capacity is not exhausted.

2.1.4 Power and Performance Modes

The Xavier NX supports multiple configurable power modes, commonly including 10 W, 15 W, and 20 W profiles. [22] These modes adjust CPU and GPU clock frequencies and may limit the number of active cores in order to remain within different power and thermal envelopes.

From the perspective of the hardware platform, these profiles define different operating envelopes rather than different architectures. The same system can therefore expose different computational capabilities depending on the selected power configuration.

For this reason, the available power modes form part of the design space of the thesis: they determine how much compute throughput can be expected from the embedded platform under different operating conditions.

2.1.5 Operational Compute Envelope

Combining the GPU arithmetic capability (~1.3 TFLOPs FP32, ~2.6 TFLOPs FP16) with 51.2 GB/s of memory bandwidth defines the theoretical compute envelope within which the perception pipeline must operate.

Given the presence of multi-camera input streams, a one-stage object detection network executed in FP16 precision, a per-instance classification network executed in FP32 precision, and real-time constraints of 30 frames per second, the embedded platform provides a constrained but programmable execution space for the target application.

The relevant challenge is therefore not only whether the models can run individually, but whether the combined workload can be organized efficiently enough to remain compatible with the hardware limits of the device.

These hardware characteristics define the fixed computational and memory boundaries that shape the architectural decisions presented in the following chapters.

2.2 Perception Pipeline Specification

2.2.1 Functional Overview and Dataflow

At a functional level, the perception pipeline is composed of the following stages:

- Input frame acquisition;
- Pre-processing;
- Detection network forward pass;
- Non-maximum suppression;
- Region cropping;
- Classification network forward pass on detected instances;
- Aggregation of semantic outputs.

For a single camera stream, these stages define the logical dataflow required to transform raw visual input into person-level semantic interpretation. In the multi-camera case, the same functional structure must be sustained concurrently across multiple input streams.

Given the embedded context of the work, the implementation is designed to minimize unnecessary host-device transfers and to keep the main inference path as device-resident as possible. The detailed realization of this strategy, including the role of the CPU and the architectural evolution of the pipeline, is discussed in the following chapters.

2.2.2 Detection Network: YOLOv8s

The detection component is based on the YOLOv8s [23] architecture pretrained on the COCO dataset.[24][25]

The input tensor is characterized by the dimensions:

$$1 \times 3 \times 256 \times 256$$

where:

- 1 is the batch dimension;
- 3 corresponds to RGB channels;
- 256×256 represents spatial resolution.

The output tensor is characterized by the dimensions:

$$1 \times 84 \times 1344$$

where:

- 1 is the batch dimension;
- 84 corresponds to 4 bounding box regression values and 80 class probabilities;
- 1344 represents spatial anchor predictions across multiple feature map scales.

Conceptually, YOLO operates as a dense prediction model. [26] The input image is processed through a hierarchy of convolutional layers that generate feature maps at different spatial resolutions. [14] At each spatial location of these feature maps, the network predicts:

- Whether an object is present,
- The bounding box coordinates relative to that spatial cell,
- The class probabilities.

Thus, the value 1344 reflects the total number of candidate anchor locations across all detection scales. Each of these locations independently outputs a vector of 84 values.

This operational structure directly reflects the convolutional principles described in Chapter 1: spatial weight sharing, hierarchical feature extraction, and parallel evaluation across spatial positions.

The YOLOv8s model contains approximately 11.2 million parameters. Using FP16 precision, the weight memory footprint is approximately:

$$11.2 \times 10^6 \times 2 \text{ bytes} \approx 22.4 \text{ MB} \quad (8)$$

At 256×256 resolution, the computational cost scales with spatial area. Starting from a reported ~28.6 GFLOPs at 640×640 resolution, area scaling yields:

$$28.6 \cdot \left(\frac{256}{640}\right)^2 \approx 4.6 \text{ GFLOPs} \quad (9)$$

Therefore, a single forward pass of the detection network requires approximately 4–5 GFLOPs.

2.2.3 Classification Network: MobileNetV2

The classification stage employs a MobileNetV2 [27] backbone with width multiplier α = 0.35.

The input tensor is characterized by the dimensions:

$$1 \times 1 \times 96 \times 96$$

where:

- 1 denotes batch size;
- 1 corresponds to the grayscale channel;
- 96×96 represents the spatial resolution of the cropped human region.

The output is a 3-dimensional vector representing class probabilities for non-threatening interaction, potentially suspicious behaviour, and harmful interaction.

MobileNetV2 is built upon depth-wise separable convolutions [28] and inverted residual blocks.[29] In contrast to standard convolutions that jointly process spatial and channel dimensions, depth-wise convolutions operate independently on each channel, followed by pointwise (1×1) convolutions that perform channel mixing. This architectural strategy significantly reduces parameter count and computational cost while preserving hierarchical feature extraction capability.

For α = 0.35, the model contains approximately **1.5 million parameters**. Assuming FP32 precision, the raw parameter memory footprint can be approximated as:

$$1.5 \times 10^6 \times 4 \text{ bytes} \approx 5.7 \text{ MB} \quad (10)$$

To estimate the computational cost, the starting point is from the commonly reported MobileNetV2 baseline:

MobileNetV2 (α = 1.0, 224×224) ≈ 300 million multiply-add operations (MACs).

The computational complexity of convolutional networks scales approximately with:

1. The square of the spatial resolution:

$$\left(\frac{96}{224}\right)^2 \approx 0.184 \quad (11)$$

2. The square of the width multiplier:

$$\alpha^2 = 0.35^2 \approx 0.1225 \quad (12)$$

Applying both scaling factors:

$$300M \times 0.184 \times 0.1225 \approx 6.8M \text{ MACs} \quad (13)$$

A multiply-add operation consists of one multiplication and one addition.

Under the common convention:

$$1 \text{ MAC} \approx 2 \text{ FLOPs} \quad (14)$$

the estimated computational cost becomes:

$$6.8M \text{ MACs} \approx 13\text{–}14 \text{ MFLOPs per inference} \quad (15)$$

Thus, while the classifier is significantly lighter than the detection network in terms of arithmetic complexity, its computational cost still scales linearly with the number of detected instances per frame.

2.3 Operational Requirements and Deployment Scenario

2.3.1 Spatial Coverage and Multi-Camera Configuration

As discussed in Chapter 1, exterior vehicle monitoring cannot rely on a single viewpoint if the objective is to observe interactions occurring at different positions around the vehicle body. For this reason, the present system adopts a multi-camera configuration rather than a single wide-angle sensing solution.[2]

The camera modules used in the experimental setup (*LI-IMX390-GMSL2-200H*) [30] feature 200° field-of-view optics and an active resolution of 1937×1217 pixels. In the adopted hardware platform, camera acquisition is supported through the Leopard Imaging carrier interface described in Section 2.1.1, while the present subsection focuses on the spatial role of the cameras within the monitoring scenario.

To address the coverage requirement, a three-camera configuration was adopted. In the experimental test setup, the cameras were positioned approximately 60 degrees apart relative to the Jetson module, enabling extended frontal coverage. In a vehicle deployment scenario, such cameras could be distributed across front and lateral positions to improve continuity of observation. Although full 360° surveillance may ideally require four cameras, the selected three-camera setup was sufficient to represent a meaningful multi-stream embedded workload for the purposes of this thesis.

The adoption of multiple cameras directly increases computational load, since each stream introduces an additional perception path that must be processed within the same real-time budget. Spatial coverage requirements are therefore directly coupled with processing scalability.

2.3.2 Real-Time Requirement Definition

In embedded perception systems, “real-time” execution refers to the ability to produce outputs within a predefined deadline. Unlike hard real-time control systems, where missing a deadline may cause system failure, the present application is associated with visual monitoring and user interaction. Therefore, the relevant deadline is perceptual rather than safety-critical.

Human visual perception typically interprets video sequences as smooth when displayed at approximately 30 frames per second. This corresponds to a frame interval of:

$$\frac{1}{30} \approx 33 \text{ ms} \tag{16}$$

Thus, maintaining a frame processing time below approximately 33 ms per stream provides perceptually smooth visual feedback. This value defines the primary performance target of the system.

The real-time requirement in this context is therefore soft but performance-driven: the system should sustain processing throughput compatible with 30 FPS visual output across all active camera streams, but occasional temporal fluctuations are tolerable provided that overall visual continuity is preserved.

This definition establishes the temporal budget against which computational feasibility must be evaluated.

2.3.3 Continuous Operation Considerations

Exterior vehicle monitoring systems may remain active for extended periods, which makes long-term operating conditions relevant even when the primary validation focus is real-time throughput. In compact embedded platforms, compute load, power consumption, and heat dissipation are interrelated and sustained execution may therefore influence effective performance over time.

For this reason, continuous-operation aspects are acknowledged as part of the broader deployment context, even though the present thesis does not attempt an extended multi-hour thermal characterization.

2.3.4 Power and Thermal Operating Modes

The Jetson Xavier NX platform supports multiple configurable power modes, including 10 W, 15 W, and 20 W profiles. [22] These modes directly influence GPU frequency, CPU frequency, and therefore the computational throughput available to the application.

Although the platform is already categorized as a low-power embedded GPU relative to desktop-class hardware, power consumption remains an important deployment factor in automotive and edge scenarios. Lower-power operation is desirable because it reduces energy demand and generally eases thermal management, but it also reduces peak compute capability.

For this reason, evaluating the system across multiple power envelopes is useful not only to compare raw performance, but also to assess the trade-off between energy efficiency and real-time feasibility under different operating conditions.

3. Baseline GPU AI Pipeline

Having established the hardware constraints and perception pipeline requirements in Chapter 2, the next step is to construct a baseline implementation capable of executing the detection and classification pipeline on a GPU platform.

The purpose of this baseline system is to provide a functional end-to-end reference implementation integrating detection and action classification into a single GPU-resident pipeline. It also establishes a measurable performance reference against which later architectural optimizations can be evaluated.

This chapter is organized around four main aspects of the baseline system. It first introduces the development environment and the TensorRT inference model adopted to execute the neural networks. It then describes the initial GPU pipeline architecture used for frame processing, inference, and debug visualization. After that, it presents the deployment packaging strategy used to generate a portable runtime artifact for the baseline implementation. Finally, it analyses the architectural limitations of this first design, which motivate the redesign introduced in the following chapter.

3.1 Development Environment and Inference

3.1.1 Development Environment

The baseline system was developed and validated in an Ubuntu environment running under Windows Subsystem for Linux (WSL) [31], using an x86 workstation equipped with a GT 1030 NVIDIA GPU. [32] This setup allowed rapid prototyping, debugging, and iterative development before deployment.

The software stack included the CUDA Toolkit [8], NVIDIA TensorRT [33], and ONNX (Open Neural Network Exchange) [34] as intermediate model representation.

Within this environment, CUDA provided the runtime and kernel programming interface for GPU execution, TensorRT provided the optimized inference runtime for deployed networks, and ONNX served as the intermediate representation used to transfer trained models into the TensorRT conversion flow.

The ONNX models for both the YOLOv8s detector and the MobileNetV2-based classifier were converted into TensorRT serialized engine files (*.engine*). These engine files contain the optimized inference graph and are directly executable by the TensorRT runtime. The engine generation process and its constraints are discussed in detail in Section 3.2.

This development environment enabled the construction of a fully GPU-resident inference pipeline before addressing embedded deployment constraints.

3.1.2 TensorRT Inference Execution Model

A TensorRT serialized engine is the executable inference artifact used by the baseline system at runtime. It represents an optimized binary form of the neural network and encapsulates the computational graph together with the execution decisions selected during engine generation, such as precision mode, memory organization and kernel selection.

In practical terms, the baseline application does not execute the original ONNX model directly. Instead, it loads pre-generated TensorRT engines and uses them as the runtime representation of the detection and classification networks. The engine generation process, and the compatibility constraints associated with it, are discussed in Section 3.2.

At runtime, TensorRT inference involves three primary components: *IRuntime*, responsible for deserializing engine files, *ICudaEngine*, representing the optimized inference network, and *IExecutionContext*, responsible for launching inference.

In the baseline implementation, a single runtime instance was created. Two engines were deserialized, one for object detection and one for action classification.

Each engine instantiated its own execution context:

```
IExecutionContext* yoloContext = yoloEngine->createExecutionContext();
IExecutionContext* clsContext = clsEngine->createExecutionContext();
```

Using separate execution contexts allows independent execution of detection and classification while sharing the same runtime instance. Reusing a single runtime is common practice, as it avoids redundant resource allocation while maintaining logical separation between engines.

Each engine exposes input and output *bindings*. A binding is an association between a named tensor in the network and a device memory pointer provided by the user.

For each engine, device memory was allocated explicitly using *cudaMalloc*, input and output bindings were mapped to these device pointers, and static input dimensions were used.

Since both networks were exported with fixed input shapes, the engines operated under static shape configuration. Inference execution was performed using *enqueueV2*, which schedules the network for execution on a specified CUDA stream:

```
yoloIO.ctx->enqueueV2(yoloIO.bindings, stream1, nullptr);
clsIO.ctx->enqueueV2(clsIO.bindings, stream1, nullptr);
```

enqueueV2 submits the inference work to the GPU asynchronously. Execution order is determined by the associated CUDA stream.

In the baseline system, both detection and classification were executed on a single CUDA stream. This results in strictly ordered execution.

The overall execution flow of a single frame after acquisition can be summarized as a sequence consisting of pre-processing CUDA kernels, detection inference, non-maximum suppression, crop extraction, per-detection classification inference, and final aggregation for visualization.

At this stage, no stream overlap or batching strategies were applied. The pipeline represents a fully GPU-resident but sequential execution model.

3.1.3 Engine Precision Characteristics

During engine generation, it was observed that precision mode (FP32 or FP16) depended on builder configuration and hardware capabilities. [35]

In certain development environments, engines were generated in FP32 precision. In others, FP16 precision was selected when supported by the GPU.

Since precision affects memory footprint, arithmetic throughput, and kernel implementation requirements, the baseline implementation was designed to support both FP32 and FP16 execution modes. Separate CUDA kernels were maintained where required to handle the corresponding data representations.

At this stage, precision was treated as a property of the generated engine rather than as a deliberate optimization decision. The engine generation process and precision configuration strategy are discussed in greater detail in Section 3.2.

3.2 TensorRT Engine Generation and Compatibility Constraints

TensorRT deployment is centered around serialized engine files (*.engine*), which represent a GPU-executable form of a neural network optimized for inference. Unlike framework checkpoints (e.g., PyTorch or TensorFlow), a TensorRT engine is not a general model description: it is a compiled inference artifact produced through a hardware-aware build process. Understanding how an engine is generated, and why portability is constrained, is essential for embedded deployment.

3.2.1 ONNX as a Network Intermediate Representation

The starting point for engine generation is a network description in a standardized format. In this work, ONNX was used as the intermediate representation. [36]

At a high level, an ONNX file provides:

- A directed computational graph (operators and tensor dependencies);
- Tensor shapes (static or partially dynamic);
- Data types;
- Trained parameters (weights);
- Operator attributes (kernel sizes, strides, padding, etc.).

Importantly, ONNX describes what the network computes, but it does not prescribe how it should be executed on a specific device. This distinction makes it suitable as an interchange format between training frameworks and deployment toolchains.

For this reason, both the detection and classification models were exported to ONNX as the first deployment step.

Most modern deep learning frameworks provide native ONNX export functionality. In practice, exporting to ONNX requires minimal intervention: the trained model graph and learned parameters are serialized into a standardized representation that preserves layer structure, tensor dimensions, and operator attributes.

Because ONNX serves as a mature and widely supported intermediate format, this step is generally straightforward and rarely constitutes a technical bottleneck. For this reason, the ONNX export procedure itself is not a focus of this thesis. The engineering challenges arise in the subsequent conversion from ONNX to a hardware-optimized TensorRT engine.

3.2.2 Engine Builder Role and Optimization Process

Engine generation is performed by a TensorRT build pipeline (conceptually an “extractor” or “builder”) that converts an ONNX network into an optimized inference engine. This process can be executed programmatically through the TensorRT API or via command-line tools (e.g., *trtexec [37]*).

The builder performs several transformations and decisions, including:

1. **Graph parsing and validation:** The ONNX graph is parsed and mapped to TensorRT layers. Unsupported operators may require graph rewriting or custom plugins.
2. **Layer fusion and graph optimization:** The builder attempts to fuse compatible operations (e.g., convolution + bias + activation) into a single optimized kernel to reduce memory traffic and kernel launch overhead.
3. **Precision selection and mixed-precision planning:** Depending on configuration, the builder may execute layers in FP32, FP16, or INT8 (if calibrated). Precision decisions affect both throughput and numerical behaviour.

4. **Tactic selection (kernel algorithm search):** For many layers (especially convolutions and GEMMs), multiple kernel implementations exist. The builder benchmarks or heuristically selects “tactics” (specific implementations) based on the target GPU architecture, available libraries, and workspace constraints.
5. **Memory planning:** TensorRT determines tensor layouts, allocates activation buffers, and schedules execution to minimize memory usage while enabling throughput.
6. **Serialization:** The resulting optimized representation is serialized into a .engine file that can later be deserialized and executed by the runtime.

This pipeline is conceptually similar to compilation: the builder converts a high-level network graph into a device-executable plan.

3.2.3 Why TensorRT Engines Are Hardware-Specific

A TensorRT engine is not a generic model file. It is a compiled artifact tailored to a specific GPU architecture and software environment.

During the build process, the TensorRT builder selects kernel implementations (tactics) based on:

- The compute capability of the target GPU;
- Available instruction sets (e.g., Tensor Core support);
- Memory hierarchy constraints;
- Precision acceleration paths.

These decisions are embedded into the engine file.

As a direct consequence: if the compute capability of the target GPU differs from that used during engine generation, the engine will not load. If the engine is deserialized on a system with incompatible CUDA or TensorRT versions, execution may fail deterministically at load time.

This behaviour is not accidental but intrinsic to the hardware-optimized compilation process performed by TensorRT.

Therefore, engine generation must be treated as a target-specific compilation step rather than a universal export operation.

3.2.4 Version Coupling and Compatibility Constraints

Beyond hardware architecture, engine compatibility depends on strict coupling between multiple software components:

- NVIDIA driver stack;
- CUDA runtime version;
- cuDNN runtime version;
- TensorRT runtime version;
- Optional plugin libraries.

A newer TensorRT runtime may not deserialize engines generated with older versions. An older runtime cannot deserialize engines generated with newer versions. CUDA/cuDNN library mismatches can lead to load failures or undefined behaviour.

Even worse, these constraints are often not well documented and must be discovered through empirical validation.

As a result, engine generation and execution must occur within a tightly controlled software stack to ensure deterministic behaviour.

3.2.5 Engine Build Configuration and Practical Trade-offs

In this work, engine generation was performed using the trtexec utility, a command-line frontend to the TensorRT builder API. Although engine creation can appear as a single command invocation, the resulting *.engine* artifact is shaped by a combination of build-time configuration choices and the local software/hardware stack.

From a practical standpoint, engine generation is influenced by several configuration dimensions:

1. **Shape model (static vs dynamic)** – Static-shape engines fix input dimensions at build time and simplify buffer allocation and runtime integration. Dynamic-shape engines require explicit optimization profiles and more complex runtime shape management. In the baseline system, both networks were built with static input dimensions to ensure deterministic behaviour and minimize integration complexity.
2. **Batch semantics (explicit batch)** – Modern TensorRT versions operate in explicit batch mode, where the batch dimension is part of the tensor shape rather than an implicit builder parameter. This affects both ONNX export configuration and runtime binding logic. The baseline engines were built in explicit batch mode with fixed input shapes.
3. **Precision selection (FP32 / FP16 / INT8)** – Precision is a build-time property that directly affects arithmetic throughput, memory footprint, and kernel selection. During early development,

engine precision varied depending on the build environment and builder defaults. As a result, the baseline runtime was implemented to support both FP32 and FP16 data paths. Precision control was later treated as an explicit optimization decision during embedded deployment.

4. **Workspace and tactic selection** – The builder may allocate temporary workspace memory to evaluate and select optimized tactics (kernel implementations). Increasing workspace can enable faster kernels but raises memory requirements. In the baseline conversion phase, once a stable and functional engine was obtained, additional workspace tuning was not prioritized.

It is important to emphasize that these configuration axes are not arbitrary switches that can override the ONNX model structure. For example, enabling dynamic shapes or batching through *trtexec* requires the ONNX graph itself to expose dynamic dimensions. If the exported ONNX model defines fixed input shapes, attempting to enforce dynamic behaviour at build time will result in a build failure. The builder optimizes what the ONNX graph expresses; it does not rewrite the network to introduce unsupported structural changes.

A representative baseline conversion for the YOLO detector was performed using the following command:

```
trtexec --onnx=yolov8s_256.onnx --saveEngine=yolov8s_256.engine
```

The following (partial) build log confirms key properties of the generated engine:

```
&&&& RUNNING TensorRT.trtexec [TensorRT v8601]
[I] Model: yolov8s_256.onnx
[I] Max batch: explicit batch
[I] Precision: FP32
[I] Selected Device: NVIDIA GeForce GT 1030
[I] Compute Capability: 6.1
[I] TensorRT version: 8.6.1
[I] Input binding with dimensions 1x3x256x256 is created.
[I] Output binding with dimensions 1x84x1344 is created.
[I] Created engine with size: 44.0967 MiB
...
```

Several observations can be made from this excerpt. The selected device and compute capability confirm that the engine was built for a specific GPU architecture. The precision mode (FP32) reflects the builder configuration and available hardware support. Explicit batch mode and fixed tensor dimensions confirm the static-shape configuration adopted in the baseline system.

This baseline engine generation step marks the transition from framework-level model representation to hardware-tailored inference artifact. Once stable engine extraction was achieved, the GPU-resident inference pipeline described in the following section could be implemented on top of these serialized engines.

3.3 Baseline GPU Inference Pipeline Architecture

3.3.1 Frame Acquisition and Host-Device Transfer

In the baseline implementation, input frames were acquired from a single camera device using **OpenCV** [38] with the **V4L2 (Video4Linux2)** [39] backend. OpenCV provides a high-level interface for video capture and image manipulation, while V4L2 represents the Linux kernel subsystem responsible for interfacing with camera hardware devices. By initializing the capture on */dev/video0*, frames were retrieved from the connected webcam in a continuous loop using *cap.read*.

OpenCV delivers decoded frames as CPU-resident images, typically in BGR format (CV_8UC3). In order to optimize subsequent host–device transfers, the frame buffer was allocated in **page-locked (pinned) host memory** using *cudaHostAlloc*.

Pinned memory refers to host memory pages that are locked and cannot be paged out by the operating system. Unlike pageable memory, pinned memory enables higher-bandwidth DMA transfers between host and device and is required to achieve fully asynchronous memory copies. A *cv::Mat* header was constructed directly on top of this pinned buffer, allowing OpenCV to write captured frames directly into pinned memory without intermediate copies.

Each frame was therefore stored as a three-channel 8-bit BGR image in pinned host memory.

Before GPU pre-processing and inference, the raw frame was explicitly transferred to device memory. This transfer was encapsulated as the first step of the GPU pre-processing routine and performed using: *cudaMemcpyAsync*.

The use of *cudaMemcpyAsync* in combination with pinned memory enables asynchronous host-to-device transfers on a CUDA stream. This allows the transfer to be scheduled without blocking the CPU, while achieving higher throughput compared to pageable memory transfers. However, an explicit copy is still required for every frame.

The baseline data path can therefore be summarized as frame acquisition on the CPU through OpenCV/V4L2, storage in pinned host memory, asynchronous host-to-device transfer, and subsequent GPU pre-processing and inference.

At this stage, the system operates in a single-camera, single-stream configuration. Frame acquisition, memory transfer, pre-processing, inference, and post-processing are executed sequentially within the same control loop. The next frame is captured only after completion of the previous iteration.

This architecture prioritizes correctness and determinism over concurrency, establishing a clear and controlled baseline pipeline upon which later performance optimizations were introduced.

3.3.2 Engine Initialization and GPU Memory Layout

Before entering the main processing loop, both neural networks (detection and classification) were initialized through TensorRT engine deserialization and explicit device memory allocation. This setup phase establishes the static computational structure of the inference pipeline.

For clarity, the detection engine is described as a representative example in this section. The classification engine follows an identical initialization pattern with its own bindings, buffers, and execution context.

Each network was provided as a precompiled TensorRT engine file. Engine loading was performed through a dedicated utility function that opened the serialized engine file in binary mode, loaded its contents into host memory, created a TensorRT runtime object, and deserialized the engine into a GPU-executable representation.

```
ICudaEngine* loadEngine(const std::string& engineFile, IRuntime*& runtime) {
   std::ifstream file(engineFile, std::ios::binary);
   if (!file) throw std::runtime_error("Failed to open engine file");
   file.seekg(0, std::ios::end);
   size_t size = file.tellg();
   file.seekg(0);
   std::vector<char> buffer(size);
   file.read(buffer.data(), size);
   runtime = createInferRuntime(gLogger);

   return runtime->deserializeCudaEngine(buffer.data(), size);
}
```

The IRuntime object represents the TensorRT execution environment responsible for deserializing engines built for a specific hardware and software configuration. The serialized .engine file is transformed into an ICudaEngine, which encapsulates the optimized inference graph tailored to the target GPU.

In this implementation, a single runtime instance was created, two engines were deserialized for detection and classification, and each engine was assigned its own execution context.

For each engine, an execution context was created:

```
IExecutionContext* context = engine->createExecutionContext();
```

The execution context represents a concrete inference instance of the static engine. While the engine contains the optimized network structure, the context manages execution state and binding associations.

A single CUDA stream was created at initialization, all pre-processing kernels and both inference stages were scheduled on this same stream. This guarantees ordered execution without requiring explicit synchronization between consecutive operations.

TensorRT engines expose their inputs and outputs through **bindings**, where each binding corresponds to a tensor defined in the optimized inference graph.

Bindings were resolved by name:

```
const int inputIndex  = engine->getBindingIndex("yolo_input");
const int outputIndex = engine->getBindingIndex("yolo_output");
```

Tensor dimensions were retrieved programmatically:

```
nvinfer1::Dims inputDims  = engine->getBindingDimensions(inputIndex);
nvinfer1::Dims outputDims = engine->getBindingDimensions(outputIndex);
```

Querying tensor dimensions at runtime ensures that buffer allocation is derived directly from the engine definition. This makes the application resilient to engine changes. For example, if the detection model input were modified from 1×3×256×256 to 1×3×512×512, the program would adapt automatically without requiring hardcoded dimension updates.

For the detection engine:

- Input tensor shape: 1 × 3 × *H* × *W*;
- Output tensor shape: 1 × 84 × *N_anchors*.

The leading dimension 1 corresponds to batch size, fixed in this baseline implementation. The channel dimension 3 corresponds to the BGR colour channels expected by the YOLO detection network.

Similarly, for the classification engine:

- Input tensor shape: 1 × 1 × *H* × *W*;
- Output tensor shape: 1 × 3.

Here, the single input channel reflects grayscale crops extracted from detected regions. Again, the batch dimension is fixed to 1 in this baseline version.

The data type associated with each binding was queried programmatically so that the correct buffer size could be derived directly from the engine definition. Using the tensor dimensions together with the element size associated with the binding data type, device buffers for inputs and outputs were allocated once during initialization and then reused throughout program execution.

For the detection engine, this resulted in one device buffer for the input tensor and one for the output tensor. No allocation or deallocation occurred inside the frame-processing loop. This static memory layout eliminates runtime allocation overhead and prevents memory fragmentation during continuous operation.

The classification engine followed the same initialization pattern, with its own bindings and device buffers.

Inference execution was triggered using:

```
context->enqueueV2(yoloBuffers, stream1, nullptr);
```

The enqueueV2 function schedules inference on the specified CUDA stream using the device pointers provided in the binding array. Because pre-processing kernels and inference calls share the same stream, execution order is preserved deterministically.

At the end of the initialization phase, both engines had been deserialized, two execution contexts had been created, a single CUDA stream was active, all required device buffers had been allocated, and tensor shapes had been resolved dynamically from the engine.

This layout forms the structural backbone of the baseline GPU inference pipeline. The next section describes how input tensors are populated and how detection results are produced and processed entirely on the GPU.

3.3.3 Detection Stage

Once engine initialization and buffer allocation were completed, the detection stage executed entirely on the GPU, transforming the raw camera frame into a compact list of validated bounding boxes corresponding to detected persons.

The detection pipeline begins with GPU pre-processing. After the asynchronous host-to-device transfer described in Section 3.3.1, the raw BGR frame resides in device memory. Pre-processing consists of a sequence of CUDA kernels designed to prepare the image tensor in the exact format expected by the YOLO detection engine.

The pre-processing stage is implemented as a sequence of CUDA kernels that resize the image to 256×256 using bilinear interpolation, convert it from BGR to RGB, transform pixel values from unsigned 8-bit integers to normalized floating-point values in the [0,1] range, and rearrange the data layout from HWC to CHW.

The final floating-point tensor is written directly into the input binding buffer associated with the TensorRT engine. No intermediate staging buffer is required. This design ensures that pre-processing output is immediately consumable by the inference engine.

Inference is then scheduled on the same CUDA stream using enqueueV2, as described previously. The produced output tensor, whose structure was introduced in Chapter 2, has dimensions:

$$1 \times 84 \times N_{\text{anchors}}$$

Since the system is designed to detect humans only, class index 0 (person) is the only class considered during post-processing of the 80 outputted classes. The ONNX model already contains the required activation functions, so no additional sigmoid operation was applied in the CUDA code.

At this stage, the detection output consists of a dense anchor grid: every anchor provides a bounding box prediction and associated confidence score. However, most anchors correspond to background regions and must be discarded. Therefore, a candidate filtering stage is required to reduce the large anchor set into a manageable list of plausible detections.

Candidate filtering is also performed entirely on the GPU. A CUDA kernel is launched with one thread per anchor. Each thread reads the predicted bounding box parameters, extracts the confidence score associated with class 0, and compares the score against a predefined threshold.

If the confidence exceeds the threshold, the bounding box is converted from center-based representation to corner representation:

$$(c_x, c_y, w, h) \rightarrow (x_1, y_1, x_2, y_2) \tag{17}$$

and stored in a device-side detection array.

The detection structure used throughout the pipeline is defined as:

```
struct Detection {
   float x1, y1, x2, y2;  // Bounding box corners
   float score;        // Confidence score
   int   class_id;     // Class ID
};
```

In addition to storing candidate detections, the filtering kernel writes a binary mask indicating whether each anchor passed the threshold. This mask is used in the subsequent compaction stage. The purpose of the mask is to separate the logical decision (valid/invalid anchor) from the physical memory reorganization, allowing the compaction step to produce a contiguous list of valid detections.

Following filtering, a compaction kernel removes invalid entries and generates a contiguous candidate list. A device-side counter tracks the number of surviving detections. This prevents later stages from iterating over all anchors and instead limits processing to meaningful candidates only.

Non-Maximum Suppression (NMS) [40] is then applied to eliminate overlapping bounding boxes. The implemented NMS follows a greedy strategy: each detection is compared against all higher-scoring detections and is suppressed if the Intersection over Union (IoU) [41] exceeds a specified threshold.

The IoU is computed directly on device using the standard formulation:

$$\mathrm{IoU}(A, B) = \frac{\mathrm{area}(A \cap B)}{\mathrm{area}(A) + \mathrm{area}(B) - \mathrm{area}(A \cap B)} \tag{18}$$

Detections that survive suppression are written to the final output array using an atomic increment operation to determine their position in the result buffer. The use of *atomicAdd* [8] is appropriate here because the expected number of surviving detections is small relative to the total number of anchors. In this regime, contention on the atomic counter remains limited. If the number of detections were large, a different parallel write strategy would be preferable to avoid serialization effects.

The total number of final detections is bounded to a reasonable maximum worst case finite number (100), to ensure predictable memory usage.

Only after completion of NMS is minimal synchronization performed. Instead of copying the entire detection buffer back to host memory, only the final detection count is transferred using an asynchronous device-to-host copy.

At the end of the detection stage, only a single integer, the number of final detections, is asynchronously copied from device to host memory. This minimal synchronization step avoids unnecessary transfer of intermediate tensors. At this point, the system holds a compact, device-resident array of validated person detections, ready to be consumed by the classification stage. Having the number of final detections copied allows the pipeline to skip entirely the classification stage if needed, reducing both time and computational cost.

3.3.4 Region Extraction and Classification Stage

Once the detection stage completes, the system holds a compact device-resident array of person bounding boxes and a scalar N representing the number of valid detections. This value directly determines the workload of the classification stage: the action classifier is executed once per detected person instance. In the baseline implementation, classification is therefore performed sequentially for each detection, and the total classification cost scales linearly with the number of detected persons in the frame.

The classification network expects a fixed-size, single-channel input tensor. More precisely, the model operates on grayscale images of spatial resolution 96×96. These constraints are not arbitrary: they derive directly from the pre-processing pipeline used during model training and fine-tuning (EdgeImpulse-based retraining performed by the Nardò team). To ensure inference consistency, the

runtime pre-processing must reproduce the same transformations applied during training. Any deviation in spatial normalization, channel ordering, or intensity scaling would lead to distribution shift and degraded classification performance.

For each detection, a structured GPU pre-processing pipeline generates a normalized grayscale crop compatible with the classifier input tensor. This pipeline is implemented as a staged sequence of CUDA kernels and operates entirely on device memory.

The first step consists of computing per-detection geometric parameters. From the bounding box coordinates produced by the detector, a small device-side parameter structure is generated. This structure encodes the bounding-box top-left corner and dimensions in the original frame, the size of a square canvas defined as X, the offsets required to center the rectangular crop within that square, and the scaling factors needed to map the original crop resolution to the classifier resolution.

This parameterization separates geometric reasoning from pixel manipulation, allowing subsequent kernels to operate using consistent and precomputed mappings.

The pixel-level pre-processing stages are then executed in sequence:

1. **Crop extraction** from the original device-resident frame into a scratch buffer;
2. **Padding to square**, where the rectangular crop is embedded into a square canvas using black padding. This ensures aspect ratio normalization and prevents geometric distortion during resizing;
3. **Bilinear resizing** from the padded square to the classifier input resolution (96×96);
4. **BGR-to-grayscale conversion and normalization**, producing a floating-point tensor written directly into the TensorRT input binding buffer associated with the classification engine.

The grayscale conversion and normalization are performed to match the model's training configuration. The final tensor corresponds to a single-channel image of fixed spatial size. Although the ONNX graph contains an internal transpose operation, the CUDA pre-processing pipeline was designed to match the model's expected ordering, so no additional rearrangement was required at host level.

Once the input tensor for a given detection is prepared, classification inference is scheduled on the same CUDA stream using enqueueV2. The engine produces a floating-point output vector of size three, corresponding to the semantic action classes defined for the application. The SoftMax activation [42] is already embedded within the ONNX graph, allowing the output values to be interpreted directly as normalized class probabilities.

The classification output is then merged with the original detection metadata inside a GPU post-processing kernel. This aggregation stage constructs a compact visualization-oriented structure containing bounding-box coordinates scaled back to the original frame resolution, the predicted action class index, and the camera identifier for future multi-camera extensions.

Because the classification stage is executed sequentially per detection, the same scratch buffers (crop, square canvas, resized image, parameter array) are reused across iterations. This reuse is safe due to the strictly sequential control flow and reduces unnecessary device memory allocations. Only the final per-instance result structure is written to a persistent output buffer, preventing overwriting of previously computed predictions.

At the conclusion of this stage, each detected person instance has been transformed into a localized action prediction, entirely computed on the GPU. The system now holds, in device memory, a compact representation of bounding boxes and associated action classes, ready for visualization.

3.3.5 Debug Visualization and Host Feedback

At the conclusion of the classification stage, the system holds in device memory a compact array of *ActionVis* structures. Each structure encodes the final result for one detected person instance, including bounding box coordinates (already scaled back to original frame resolution), predicted action class and camera identifier.

Unlike intermediate tensors used during inference, these structures are intentionally lightweight. Instead of transferring the full image or large intermediate buffers back to the CPU, only the final per-instance metadata is copied. This design choice minimizes host–device bandwidth usage and preserves the GPU-centric nature of the inference pipeline.

The purpose of the debug visualization stage is qualitative validation. During early development and integration, it is essential to visually verify that detections correspond to actual persons in the scene, that bounding-box localization is geometrically correct, and that action classification aligns with observed behaviour.

To achieve this, the system renders each processed frame with graphical overlays representing detection and classification outputs. This produces a live annotated video stream that mirrors what the camera sees, augmented with bounding boxes and color-coded class labels. Observing this stream allows immediate qualitative inspection of both spatial and semantic correctness.

In the baseline implementation, the *ActionVis* array is copied from device to host using *cudaMemcpyAsync [8]*, a stream synchronization ensures that inference and post-processing have completed before the host accesses the data, and the CPU then performs debug rendering using the original frame already available in host memory.

The video frame itself is acquired on the CPU (via OpenCV/V4L2), copied to the GPU for processing, and retained in host memory for visualization. Therefore, debug rendering does not require transferring image data back from the GPU. Only structured inference results are transferred.

On the CPU side, visualization consists of drawing bounding rectangles using pixel coordinates precomputed on the GPU, rendering the class labels associated with each detection, optionally

applying visual effects such as Gaussian blur for selected classes, and displaying the annotated frame in a standard OpenCV window.

Because coordinate scaling and class selection are already resolved on the GPU, the CPU stage performs only lightweight rendering operations. No geometric transformations or classification computations are executed on the host.

It is important to note that this visualization mechanism is not required for inference execution. The computational core of the pipeline remains entirely device-resident and could operate without any rendering step. The visualization layer exists solely to enable rapid qualitative validation during development.

From a systems perspective, this design keeps inference computation entirely on the GPU while limiting host interaction to compact metadata transfers and debug-oriented rendering. Although the baseline approach uses explicit stream synchronization before visualization, it establishes a clear and deterministic execution model suitable for correctness validation. The architectural implications of this initial strategy will be examined in Section 3.5.

3.4 Deployment Packaging Strategy

3.4.1 Design Objective: Self-Contained Runtime Artifact

While the baseline pipeline described in Section 3.3 demonstrates functional end-to-end inference on the development machine, this alone does not satisfy the original project requirement. The feasibility question posed by the industrial stakeholders was not simply whether person detection and action classification could run on a GPU, but whether the complete pipeline could be executed reliably on external machines and testing platforms with minimal setup effort. For this reason, deployment portability was treated as an engineering requirement rather than an auxiliary feature.

In practice, CUDA/TensorRT applications are not distributed as a single executable. Correct execution depends on a set of tightly coupled runtime components, including shared libraries (CUDA runtime, TensorRT runtime, cuDNN, and additional third-party dependencies such as OpenCV), as well as model-specific inference artifacts (TensorRT engine files). Relying on the target machine to provide these components through system-wide installation introduces fragility: runtime failures may occur due to missing libraries, incompatible versions, or inconsistent environment configuration.

Moreover, full installations of CUDA Toolkit, TensorRT SDK, and cuDNN typically occupy several gigabytes of disk space. For example, a complete TensorRT installation may exceed 10 GB when including headers, samples, documentation, and development tools. In an embedded or industrial context, such installations are unnecessary and undesirable when the application requires only a subset of runtime shared libraries to execute inference. Reducing the runtime footprint was therefore an explicit objective of the deployment strategy.

The deployment goal was to generate a self-contained user-space runtime artifact that could be transferred to a new Linux machine and executed immediately, without requiring manual installation of CUDA/TensorRT/cuDNN stacks or environment configuration. In this thesis, this requirement is referred to as “bare-metal capable” deployment in the practical sense: assuming only a standard operating system and NVIDIA driver stack, the deployment package must contain exactly the runtime components required by the application, and no development toolchains or full SDK distributions.

This approach provides several concrete benefits that were essential in the project context: reproducibility, since the same executable and runtime libraries are executed across machines; operational simplicity, because deployment becomes a copy-and-run procedure; reduced footprint, because only the shared objects actually required by the binary are shipped; and performance-validation readiness, since the packaged artifact can be executed and profiled in a controlled manner.

This deployment step is not independent from the inference architecture described previously. The hardware-specific nature of TensorRT engines and the strict coupling between runtime libraries directly shaped how the deployment package was organized. The following sections describe the packaging strategy that was implemented to satisfy portability, reproducibility, and footprint constraints while preserving profiling capability.

3.4.2 Runtime Dependency Encapsulation

Achieving a self-contained deployment artifact requires resolving all runtime dependencies of the application. CUDA and TensorRT applications depend on multiple dynamically linked shared libraries, which are normally resolved through the system linker at runtime. If these libraries are missing or version-incompatible on the target machine, execution fails even if the binary itself is correct.

To address this, the deployment process was integrated directly into the project’s build system through a dedicated *Makefile* [43] target. The *Makefile* serves not only as a wrapper around *CMake* [44] for compilation, but also as an orchestration layer for packaging and dependency closure. This ensures that deployment generation is reproducible and can be triggered with a single command, rather than relying on manual file collection.

After compilation, the deployment target analyses the produced executable using the *ldd* utility. The *ldd* tool lists all shared libraries required by a dynamically linked binary and resolves them to absolute filesystem paths. By parsing this output, the *Makefile* automatically identifies all runtime dependencies of the application.

Each resolved shared object with an absolute path is then copied into a dedicated deploy/lib/ directory. This automated library-closure step ensures that CUDA runtime libraries, TensorRT runtime libraries, cuDNN libraries when required, and OpenCV and C++ runtime dependencies are bundled alongside the executable without requiring full SDK installations on the target system.

Filtering is performed directly on the absolute paths reported by ldd, ensuring that only actual filesystem-backed shared objects are copied. Virtual entries (such as linux-vdso) are naturally excluded by this process.

In addition to shared libraries, the dynamic linker itself (ld-linux-x86-64.so.2) is copied into the deployment directory when available. Rather than relying on the system's default loader and environment configuration (such as LD_LIBRARY_PATH), the executable is launched explicitly through the bundled loader using the --library-path option. Conceptually, execution follows the form:

```
ld-linux-x86-64.so.2 --library-path ./lib ./bin/yolodetector
```

This mechanism guarantees that the binary resolves its dependencies exclusively from the packaged lib/ directory, isolating it from potential mismatches in system-wide installations. The result is deterministic runtime behaviour across machines, provided that the underlying NVIDIA driver stack is compatible.

This encapsulation mechanism concretely implements the deployment objectives defined in Section 3.4.1. By resolving and packaging only the shared libraries directly required by the compiled binary, the system avoids reliance on external CUDA/TensorRT/cuDNN installations while simultaneously reducing the runtime footprint. Execution is no longer dependent on host-specific environment variables or preconfigured library paths, since the bundled dynamic loader enforces resolution against the local lib/ directory.

As a result, the deployment artifact behaves as an isolated and relocatable runtime bundle. All components required for inference execution, executable, shared libraries, dynamic loader, and engine files, reside within a single directory hierarchy. Provided that the target system offers a compatible NVIDIA driver and GPU architecture, the application can be executed without further configuration.

The following section details the organization of the deployment directory and how engine files are integrated into this packaging strategy.

3.4.3 Deployment Folder Structure and Build Orchestration

The dependency resolution strategy described in the previous section (based on ldd) does not, by itself, constitute a deployment solution. The objective of the project was not simply to identify required libraries, but to produce a complete runtime artifact that could be transferred to another machine and executed with minimal external requirements.

In practical terms, this means generating a deployable package, typically distributed as a compressed archive, containing every element required for execution: the executable, the necessary shared libraries, the TensorRT engines and the launch scripts. The creation of this structured folder represents the final step required to complete the deployment strategy.

The resulting deployment directory follows a clear and minimal layout:

Deploy folder structure

- **deploy/**
  - **bin/**
    - yolodetector
  - **lib/**
    - *required shared libraries*
    - ld-linux-x86-64.so.2
  - **engines/**
    - yolo.engine
    - action_cls.engine
  - run.sh
  - run_profile.sh

Figure 1 - Deploy folder structure

Each component has its specific role:

- **bin/** contains the compiled executable;
- **lib/** includes only the shared libraries actually required at runtime, as determined through dependency inspection. The dynamic loader is also bundled to guarantee correct linking behaviour;
- **engines/** stores the serialized TensorRT engines for detection and classification;
- **run.sh** executes the application using the bundled dynamic loader and a local library path;
- **run_profile.sh** provides a profiling-ready entry point based on NVIDIA Nsight Systems, enabling benchmarking of the deployed artifact.

The build system is structured in two layers.

CMake handles compilation and linking. It configures CUDA compilation, sets architecture properties, defines include paths, links TensorRT and OpenCV, and embeds engine path macros directly into the binary.

On top of this, a higher-level *Makefile* orchestrates the complete workflow. The *Makefile*:

1. Triggers a rebuild of the project to ensure binary consistency;
2. Creates the deployment directory structure;
3. Copies the executable into the appropriate location;
4. Resolves and copies the required shared libraries;
5. Bundles the dynamic linker;
6. Generates execution and profiling scripts;
7. Optionally packages the deploy directory into a compressed archive.

The deployment process is therefore not a manual assembly of files, but a reproducible procedure embedded in the build system. The make deploy target encapsulates all these steps, ensuring that the generated artifact always reflects the current build configuration and dependency state.

The integration and lifecycle management of TensorRT engines within this deployment model are discussed in the following section.

3.4.4 Engine Substitution Workflow

The deployment folder described in the previous section includes an engines/ directory containing the serialized TensorRT engines used for detection and classification. For the deploy artifact to function correctly on a given machine, this directory must contain two valid engine files generated specifically for the target hardware.

At first glance, this requirement may appear inconsistent with the deployment philosophy introduced earlier. The entire strategy was designed to avoid installing the full CUDA, TensorRT, and cuDNN stacks on every execution machine. However, engine generation itself requires precisely those components.

This apparent incongruence must be addressed explicitly.

As detailed in Section 3.2, TensorRT engine generation is a hardware-aware compilation process. During extraction, the ONNX network is parsed, optimized, and transformed into a device-specific inference artifact.

This process depends on:

- The GPU architecture;
- The installed TensorRT version;
- The CUDA toolkit;
- The cuDNN library.

The key distinction lies in separating engine **generation**, which is an optimization and compilation step, from engine **execution**, which is the runtime deserialization and inference phase.

The deployment model leverages this separation. To prepare a machine for deployment, the following steps are required:

1. Install the full CUDA/TensorRT/cuDNN stack on the target hardware;
2. Generate the detection and classification engines for that GPU.

Once these steps have been completed, the full SDK installation is no longer required for runtime execution. The deploy artifact only relies on the minimal subset of shared libraries bundled in the *lib/* directory, while the generated engines must be copied into the *engines/* folder.

This establishes an "extract once, reuse many times" principle:

- Heavy dependencies are required only during the engine extraction phase;
- Runtime execution depends solely on the lightweight deployment package;
- The application binary does not need recompilation when engines are substituted.

In practical terms, adapting the baseline system to another compatible machine within the same software and architecture class requires replacing the engine files with versions generated on that hardware, while leaving the executable and bundled runtime structure unchanged.

This separation between runtime packaging and target-specific engine generation was sufficient for the baseline desktop deployment workflow. However, it also exposes a structural limitation of the approach, since the portability of the deployment artifact remains conditioned by the need to regenerate engines for each target configuration. The architectural consequences of this coupling are examined in Section 3.5, while the additional complexity introduced by cross-architecture deployment is addressed later in Chapter 5.

3.4.5 Benchmarking and Profiling Support

Beyond packaging and dependency isolation, the deployment strategy was designed to guarantee controlled execution and reliable performance evaluation. Since feasibility assessment and benchmarking were core objectives of the project, the deployed artifact had to support both normal execution and structured profiling without modification.

For this reason, two execution entry points were generated automatically inside the deploy directory: run.sh and run_profile.sh.

The run.sh script provides a deterministic execution environment for the deployed binary. Instead of launching the executable directly, the script explicitly invokes the bundled dynamic loader (ld-linux-x86-64.so.2) and specifies the local *lib/* directory as the runtime library path.

This approach guarantees that the application links against the bundled shared libraries rather than system-installed versions and also removes dependency on user-specific environment variables such as *LD_LIBRARY_PATH*.

Additionally, the script exports the Qt platform plugin path to ensure correct rendering of the OpenCV display window in environments where Qt is required for GUI support.

Command-line arguments are transparently forwarded to the executable, preserving full flexibility while maintaining strict control over the runtime environment.

The result is a predictable and isolated execution model: the same binary, the same engines, and the same libraries are always used, independent of the host system configuration.

In addition to standard execution, profiling support was integrated directly into the deployment artifact through the run_profile.sh script. This script launches the application using NVIDIA Nsight Systems (*nsys*) and generates a *.nsys-rep* profiling report. The profiling configuration traces CUDA kernel activity, NVTX markers, and operating-system runtime interactions.

Embedding profiling at the deployment level serves a precise purpose: performance validation must be performed on the same artifact that is executed in production-like conditions. This avoids discrepancies between development builds and deployed builds.

To support structured profiling, NVTX (NVIDIA Tools Extension) ranges were inserted throughout the codebase. These markers delineate the main computational phases of the pipeline, including:

1. Frame acquisition;
2. YOLO pre-processing stages;
3. YOLO inference;
4. YOLO post-processing;
5. Classification pre-processing stages;
6. Classification inference;
7. Classification post-processing;
8. Debug visualization

Using NVTX markers instead of simple timestamp measurements provides significantly higher analytical value. NVTX annotations allow correlation between high-level pipeline stages and low-level GPU kernel execution within Nsight Systems, enabling precise identification of bottlenecks and synchronization behaviour.

By integrating both execution and profiling scripts within the deployment directory, the system ensures that the exact same binary, bundled runtime libraries, engines, and profiling configuration are used during testing and evaluation.

This guarantees reproducibility of performance measurements across machines with identical hardware specifications.

The deployment artifact therefore does not merely enable execution; it provides a controlled and analysable runtime environment suitable for quantitative feasibility assessment.

3.5 Architectural Limitations of the Baseline System

3.5.1 Sequential Execution Model and Latency Scaling

The baseline implementation follows a strictly sequential execution model. All stages of the pipeline are executed on a single CUDA stream, and each phase begins only after the previous one has completed. The execution order for a single frame is:

1. Detection pre-processing;
2. YOLO inference;
3. YOLO post-processing;
4. For each detected person:
    - Classification pre-processing;
    - Classification inference;
    - Classification post-processing.

No overlap exists between detection and classification, and classification of multiple detections is performed serially in a loop.

Profiling results obtained from the fully deployed version of the system, tested on a separate machine equipped with an RTX 2080 NVIDIA GPU [45] [46], show the following median execution times (Table 1):

Table 1 - Baseline system stages median latency, executed on RTX 2080 GPU, the classification times are intended per-detection

Stage Name	*Median [ms]*
YOLO pre-processing	0.055
YOLO inference	1.173
YOLO post-processing	1.715
Classification pre-processing	0.006
Classification inference	0.281
Classification post-processing	0.281

The total median cost of the detection stage is therefore approximately:

$$T_{\text{det}} \approx 0.055 + 1.173 + 1.715 \approx 2.94 \text{ ms} \tag{19}$$

The total classification cost per detected person is:

$$T_{\text{cls}} \approx 0.006 + 0.281 + 0.004 \approx 0.291 \text{ ms} \tag{20}$$

In Chapter 2, the computational complexity of the classification network was shown to be significantly lower than that of the detection model in terms of FLOPs. However, the measured latency reveals another picture. Despite the classifier requiring orders of magnitude fewer FLOPs than the detector, the observed runtime ratio is much closer:

$$\frac{T_{\text{det}}}{T_{\text{cls}}} \approx \frac{2.94}{0.291} \approx 10 \tag{21}$$

This illustrates an important point: theoretical FLOP counts do not directly translate into proportional runtime differences, especially when memory access patterns, kernel launch overhead, and TensorRT execution behaviour are considered.

This discrepancy arises because runtime performance is influenced not only by arithmetic workload but also by factors such as memory access patterns, kernel launch overhead, and framework-level execution behaviour within TensorRT.

Since classification is executed once per detection, the total frame time in the baseline model can be approximated as:

$$T_{\text{frame}} \approx T_{\text{det}} + N \cdot T_{\text{cls}} \tag{22}$$

where $N$is the number of detected people.

This scaling behaviour has direct architectural implications:

- With $N = 1$, classification adds approximately 0.29 ms;
- With $N = 5$, classification adds approximately 1.46 ms;
- With $N = 10$, classification adds approximately 2.91 ms;

With ten detections, the classification stage alone becomes comparable to the entire detection stage. Therefore, classification cannot be treated as negligible when multiple subjects are present.

Nsight Systems timeline analysis confirms this linear scaling: classification inference calls appear sequentially after detection post-processing, producing repeated inference blocks proportional to the number of detections.

The structural limitation is therefore clear: frame latency grows linearly with the number of detected people. In the absence of batching, pipelining, or stream-level concurrency, the architecture cannot amortize classification cost across detections. This subsection therefore establishes the baseline latency model that characterizes the first implementation.

3.5.2 Synchronization and Host-Device Coupling

Although the baseline pipeline executes all major computation on the GPU, its execution flow is strongly shaped by explicit synchronization mechanisms between host and device.

Two CUDA synchronization primitives were used in the baseline implementation:

- *cudaStreamWaitEvent*, which establishes a **device-side dependency** between streams. It ensures that work enqueued in one stream begins only after a specific event in another stream has completed, without blocking the CPU;
- *cudaEventSynchronize*, which introduces a **host-side synchronization point**, blocking the CPU thread until the specified event has been reached on the GPU.

After YOLO inference and GPU post-processing, an event was recorded on the detection stream to signal that the final detection buffers were ready. The classification stream was correctly made dependent on this event using *cudaStreamWaitEvent*, guaranteeing proper GPU execution ordering.

For development robustness and debug safety, an additional host-side wait (*cudaEventSynchronize*) was introduced. This ensured deterministic behaviour and prevented any possibility of the CPU accessing incomplete results during early validation phases. However, this host-side synchronization creates a strict phase boundary between detection and classification.

In the baseline architecture, the NVTX range associated with YOLO post-processing was intentionally defined to represent the time interval between the end of YOLO inference and the moment classification pre-processing could safely begin.

Because the host-side wait occurs within this interval, the measured duration of the detection stage includes both kernel execution and synchronization delay. Even if the post-processing kernels themselves are fast, the overall stage duration increases due to the enforced wait. As in the medians shown in 3.5.1, the total duration was even comparable with the inference timing.

The result is a pipeline where the CPU acts as a strict phase controller, enforcing serialized progression between stages. Although this simplifies reasoning and debugging, it also causes the measured duration of a stage to reflect not only computation, but the synchronization structure imposed by the host.

3.5.3 Detection-Driven Workload Variability

Section 3.5.1 established that total frame latency in the baseline system scales linearly with the number of detected persons. The purpose of the present subsection is to examine the architectural consequence of that property: the workload of the system becomes directly dependent on scene content.

Because each detected person triggers a complete classification cycle, frame processing time becomes directly dependent on scene content. The system no longer operates with a constant per-frame computational cost. Instead, its workload is dynamically determined by the output of the detection network itself.

This introduces two structural limitations.

Input-Dependent Latency

In sparse scenes containing one or two subjects, classification contributes minimally to total latency. In dense scenes, however, the cumulative cost of sequential classification becomes significant.

On the RTX 2080 platform used for baseline benchmarking, the median per-detection classification cost is approximately 0.29 ms. While this value appears small in isolation, its linear accumulation produces measurable increases in total frame time as Ngrows.

This effect is clearly visible in Nsight Systems traces. Figure 2 shows a representative timeline segment in which multiple classification inference calls appear consecutively following detection post-processing.

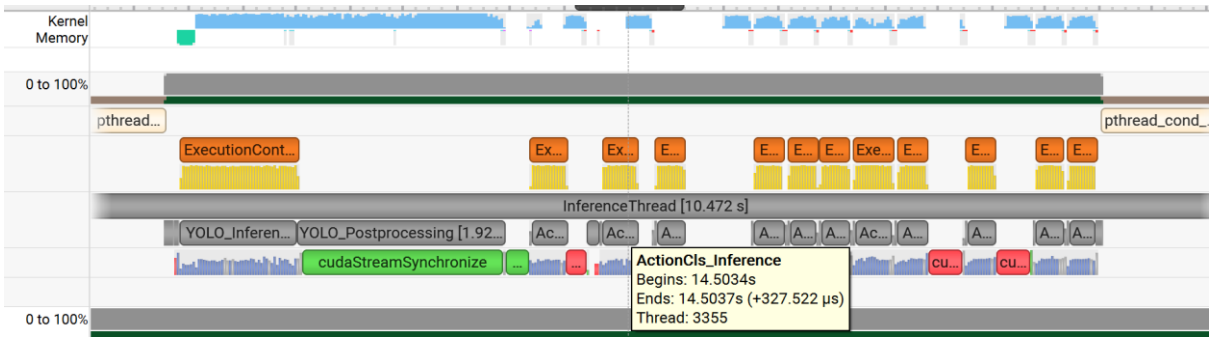

Figure 2 – NVIDIA Nsight Systems screenshot showing how a frame with multiple detections generates a sequence of multiple Action classification engine inferences.

Each additional detection introduces an additional inference block, forming a contiguous sequence of classification executions. The workload therefore scales in proportion to scene complexity.

Temporal Variability and Reduced Predictability

Beyond increased latency, detection-driven scaling introduces variability in frame duration.

Because the number of detections varies over time, the system does not produce uniform frame processing intervals. Dense frames require more classification cycles than sparse ones, resulting in fluctuating execution time across successive frames.

This variability reduces temporal predictability. Even if the median frame time satisfies real-time constraints, worst-case latency increases with scene density.

While this behaviour is manageable on a high-performance GPU such as the RTX 2080, it becomes increasingly problematic on resource-constrained platforms or in multi-camera configurations.

The baseline architecture does not implement batching, workload aggregation, or concurrency mechanisms to mitigate this effect. As a result, detection output directly amplifies computational demand in a linear and content-dependent manner.

This characteristic represents a fundamental limitation of the sequential baseline design and motivates the introduction of batching and parallel scheduling strategies in later revisions of the system.

3.5.4 Hardware Coupling of TensorRT Engines

Sections 3.2 and 3.4.4 established that TensorRT engines are not generic model files, but target-specific inference artifacts produced through a hardware-aware build process. The relevance of this property at the architectural level is that the baseline deployment workflow remains only partially portable: while the runtime package can be transferred as a self-contained artifact, the validity of the included engines still depends on the configuration under which they were generated.

Engine regeneration therefore becomes necessary whenever the target configuration changes in ways that affect compatibility, for example:

- changes in GPU architecture or compute capability;
- modifications to the ONNX model structure;
- changes in precision mode or batch configuration;
- updates to the TensorRT or CUDA toolchain used during extraction.

The limitation is not that inference cannot be deployed, but that deployment remains coupled to a target-specific engine generation step. In the baseline x86-oriented workflow this constraint was manageable, since engine extraction and runtime execution remained within the same development class. However, the presence of this dependency already indicates that portability is not absolute and may become a more significant engineering issue when the system is transferred to a different execution environment.

3.5.5 Scalability Constraints and Design Rigidity

The baseline system can be characterized as a synchronous, frame-atomic execution model: each frame is processed as an indivisible computational unit, traversing the entire pipeline from acquisition to visualization before the next frame is considered.

This design ensured clarity, correctness, and controlled benchmarking, but it also defined the structural limits of the system. At the GPU level, memory transfer, pre-processing, detection inference, post-processing, classification and visualization were arranged in a strict dependency chain. No double buffering was introduced, no inter-frame pipelining was attempted, and no meaningful overlap was implemented between computational stages.

In addition to GPU-side serialization, the host-side control flow followed a single-threaded structure. Frame acquisition, inference execution, and debug visualization were all executed within the same host thread. As a consequence, the application had to complete the entire processing cycle of one frame, including rendering, before acquiring the next one. This prevented logically distinct stages from progressing concurrently and further reduced the opportunity to hide latency.

From a scalability perspective, the baseline implementation was also tied to a single-camera workflow. Extending the codebase to multiple camera streams would therefore require architectural changes rather than simple parameter adjustments, since the processing logic assumed a single input source, a single detection list, and a sequential classification loop tied to that frame.

At the model level, both networks were initially exported and optimized for single-batch inference. True batching was therefore not available at runtime. Avoiding sequential per-instance execution would require explicit modification of the exported ONNX graphs followed by regeneration of the TensorRT engines. The scalability limitation thus existed at two levels simultaneously: the pipeline structure was sequential, and the inference artifacts themselves were not yet configured to support more efficient grouped execution.

The baseline implementation therefore establishes a fully functional and GPU-resident reference system suitable for correctness validation, performance measurement, and deployment experimentation. At the same time, the analysis presented in this section highlights structural limitations related to scheduling, workload variability, and scalability. These observations motivate the redesign introduced in the following chapter.

4. Architectural Redesign and System Optimization

The baseline system described in Chapter 3 established a fully functional GPU-based pipeline for human detection and action classification. While the implementation demonstrated the feasibility of executing the complete inference workflow on the GPU, the analysis presented in Section 3.5 also revealed structural limitations affecting scalability and resource utilization.

The second development phase therefore focused on redesigning the pipeline architecture to address these limitations. Rather than relying on isolated micro-optimizations of individual kernels, the redesign introduced structural modifications to the execution model, affecting both inference scheduling and the organization of the host-side processing pipeline.

The chapter is organized around three main redesign axes. First, the classification stage was restructured to support batched inference over multiple detections. Second, the host-side pipeline was decoupled through multi-thread execution. Third, GPU scheduling was reorganized through the introduction of multiple CUDA streams. The following sections describe these modifications and the resulting execution model of the second system iteration.

4.1 Batched Classification Redesign and System Optimization

The analysis of the baseline system presented in Section 3.5 identified a structural limitation in the classification stage. In the original pipeline, each detected individual triggered an independent classification inference, causing the total cost of the classification stage to increase with the number of detections present in the scene.

To address this limitation, the redesigned architecture introduces batched classification inference. Instead of executing one inference pass per detected person, the classifier processes multiple detections simultaneously within a single inference operation. [47]

Implementing this redesign required modifications at multiple levels of the pipeline. At a conceptual level, the workflow had to move from sequential per-detection execution to grouped processing. At the model level, the classification engine had to be modified to support batched inputs. At the CUDA level, the surrounding pre-processing and post-processing stages had to be adapted to generate and consume batched data structures.

The following sections therefore present the redesign in four steps: the before/after workflow, the model modification required to support dynamic batching, the adaptation of the surrounding CUDA stages, and the resulting integration into the pipeline.

4.1.1 Batched Classification Workflow

The detection stage produces a set of bounding boxes corresponding to the individuals detected within the processed frame. In the baseline implementation, the classification stage processed these detections sequentially. Each bounding box triggered a complete classification cycle consisting of crop generation, pre-processing, inference, and result processing.

Conceptually, the baseline workflow can be represented as follows:

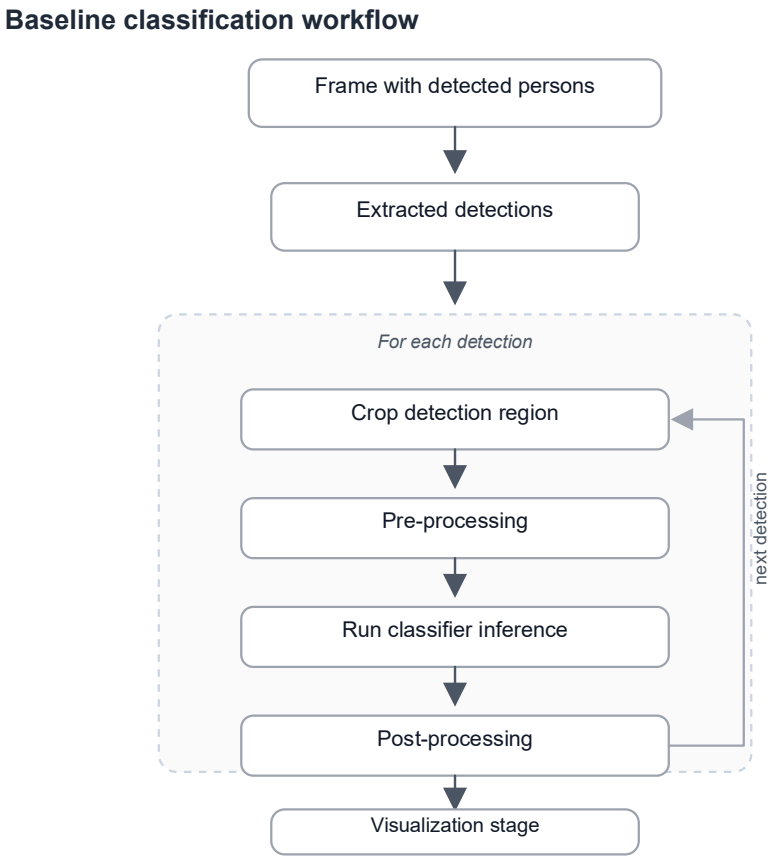

Figure 3 - Classification stage looping for each detected person.

The redesigned pipeline replaces this iterative structure with a batched workflow in which all detections are prepared together and then evaluated through a single classification inference.

The updated workflow can be summarized as follows in the next page.

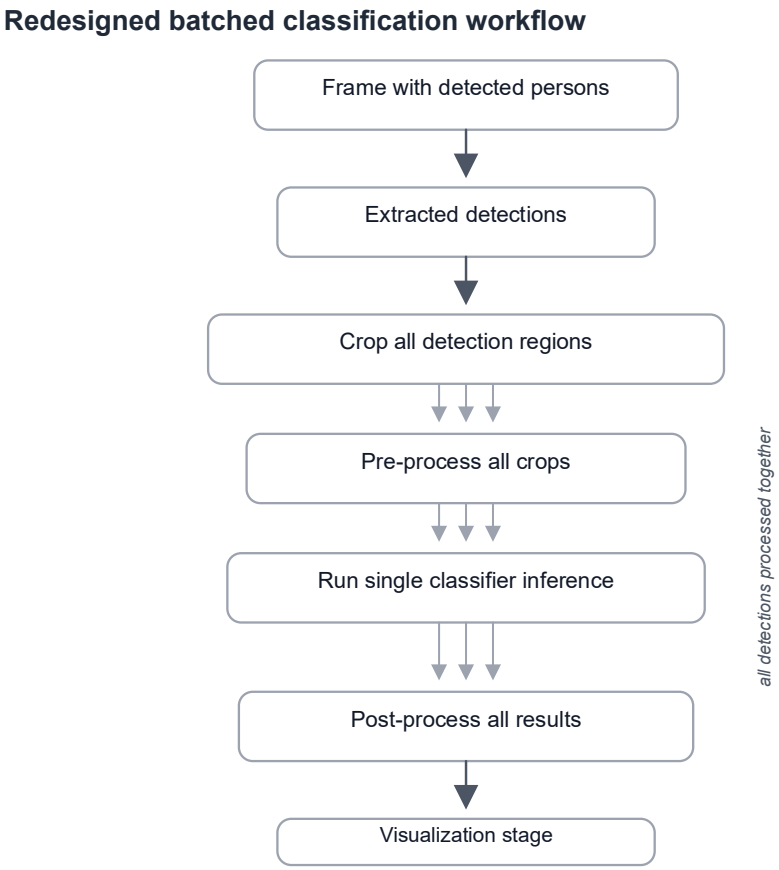

Figure 4 - Classification stage working on each detection in one pass.

In this configuration, the classifier receives a tensor containing multiple inputs simultaneously and produces a corresponding batch of classification outputs. The sequential loop over detections is therefore removed, and the classification stage becomes a single batched inference step integrated into the main processing pipeline.

4.1.2 Dynamic Batch Support in the Classification Model

The original classification model was designed with a fixed batch size of one, meaning that a single input tensor was processed per inference execution. Batched inference requires the model to accept multiple inputs simultaneously, which necessitated modifying the classification model representation.

This modification was implemented by introducing a dynamic batch dimension in the input tensor, so that multiple crops could be provided to the classifier at the same time. Consequently, the classifier output also became batched, producing one classification result for each input crop included in the current batch.

Conceptually, the difference between the original and the modified execution can be represented as follows:

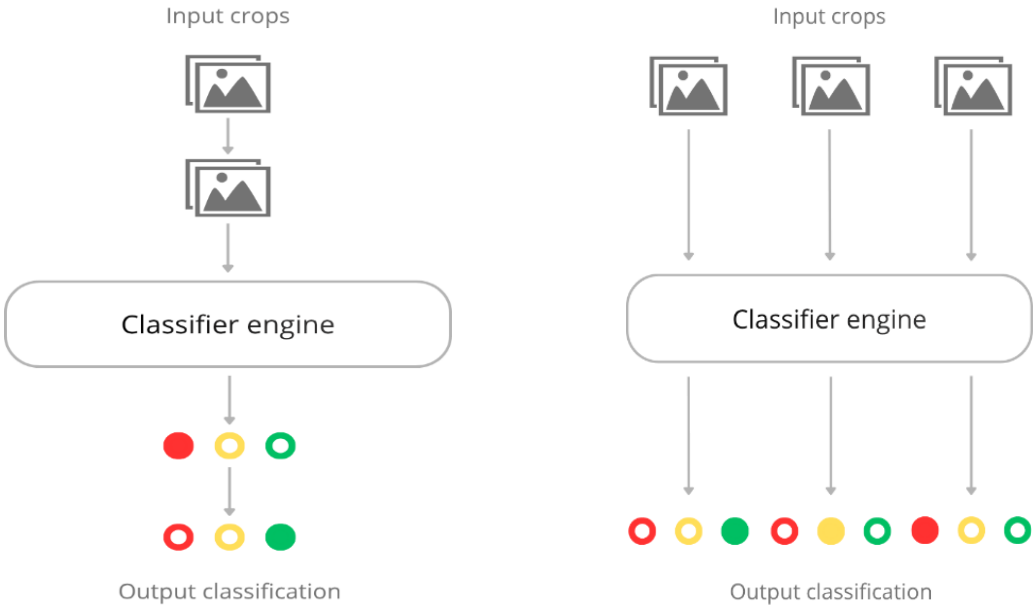

Figure 5 - Visual comparison between static single inference and dynamic batched inference.

In the original configuration, the input tensor had shape:

$$1 \times 1 \times 96 \times 96$$

Modified configuration input shape:

$$N \times 1 \times 96 \times 96$$

where Nrepresents the number of detections to classify within the current frame.

This modification was performed directly on the ONNX model using the NVIDIA Deep Learning Designer interface [48], which allows interactive editing of tensor dimensions within the network graph. After modifying the ONNX representation, the TensorRT engine was regenerated, as previously discussed in Section 3.2. Any modification to the ONNX graph requires a new engine extraction step.

Supporting a dynamic batch dimension also requires a small runtime adjustment. Before executing the classifier inference, the batch size must be explicitly specified through the TensorRT execution context:

```
nvinfer1::Dims4 inDims{N, 96, 96, 1};
clsIO.ctx->setBindingDimensions(0, inDims);
```

This operation configures the input tensor dimensions according to the number of detections present in the current frame.

An alternative implementation based on a fixed maximum batch size was also considered. Under that approach, the classifier would always process an input tensor sized for the maximum expected number of detections, while unused entries would simply be ignored by the surrounding CUDA kernels through guard conditions.

This solution was conceptually simpler, but it remained inefficient at inference level. Although kernel execution could skip invalid entries, the TensorRT engine itself would still evaluate the entire batch, including meaningless inputs corresponding to absent detections. The result would be unnecessary memory traffic and avoidable computation on unused batch elements.

Experimental evaluation showed that dynamically transferring the detection count N and configuring the batch dimension at runtime introduced negligible overhead. Since dynamic batching preserved throughput while avoiding unnecessary inference work, it became the preferred implementation.

4.1.3 Batched Pre-processing and Runtime Integration

Once the classifier engine was modified to support dynamic batch sizes, the surrounding pre-processing and post-processing stages had to be adapted accordingly.

The pre-processing pipeline described in Section 3.3.4 transforms the bounding boxes produced by the detection stage into normalized classifier inputs. In the baseline implementation, these operations were executed sequentially for each detection. In the redesigned version, the same operations had to be applied to multiple detections within the same execution cycle.

To achieve this, the pre-processing CUDA kernels were redesigned to operate on a three-dimensional execution grid. The additional grid dimension corresponds to the detection index, so that each slice of the grid processes one detection within the batch. This allows the same crop extraction, padding, resizing, and grayscale normalization operations to be applied concurrently across all detected regions.

The same principle was extended to the classification post-processing stage. Instead of processing each classification result sequentially, the output tensor produced by the batched inference is processed collectively. The classification probabilities associated with each detection are combined with the corresponding bounding-box information to generate the final visualization structures.

Through this redesign, the pre-processing stage, the classifier inference stage, and the post-processing stage become aligned around the same batch-oriented execution model, enabling the entire classification stage to operate as a grouped pipeline rather than as a sequential loop.

4.2 Pipeline Decoupling Through Multi-Threading

The second architectural improvement introduced in the redesigned system concerns the structure of the host-side processing pipeline. In the original implementation, frame acquisition, inference processing, and visualization were executed sequentially within a single host thread. While this approach simplified control flow, it also tightly coupled stages that did not need to operate in strict lockstep.

To improve execution flexibility and allow different stages of the application to progress more independently, the second iteration introduces pipeline decoupling through multiple host threads. Each thread is assigned a specific functional role within the pipeline, allowing acquisition, inference, and visualization to proceed concurrently while remaining logically coordinated.

The following sections describe the conceptual transition from the original single-thread structure to the decoupled version, the resulting multi-thread organization, and the mechanisms used to coordinate data exchange between stages.

4.2.1 Baseline Sequential Execution Model

In the baseline system, frame acquisition, inference processing, and visualization were all executed within the same host thread. As a result, each stage had to wait for the completion of the previous one before progressing.

Conceptually, the baseline pipeline followed the structure:

Figure 6 - Single thread pipeline dataflow conceptual diagram.

The redesigned architecture preserves the same logical stages, but removes their strict host-side serialization by assigning them to separate execution contexts.

Conceptually, the redesigned pipeline becomes:

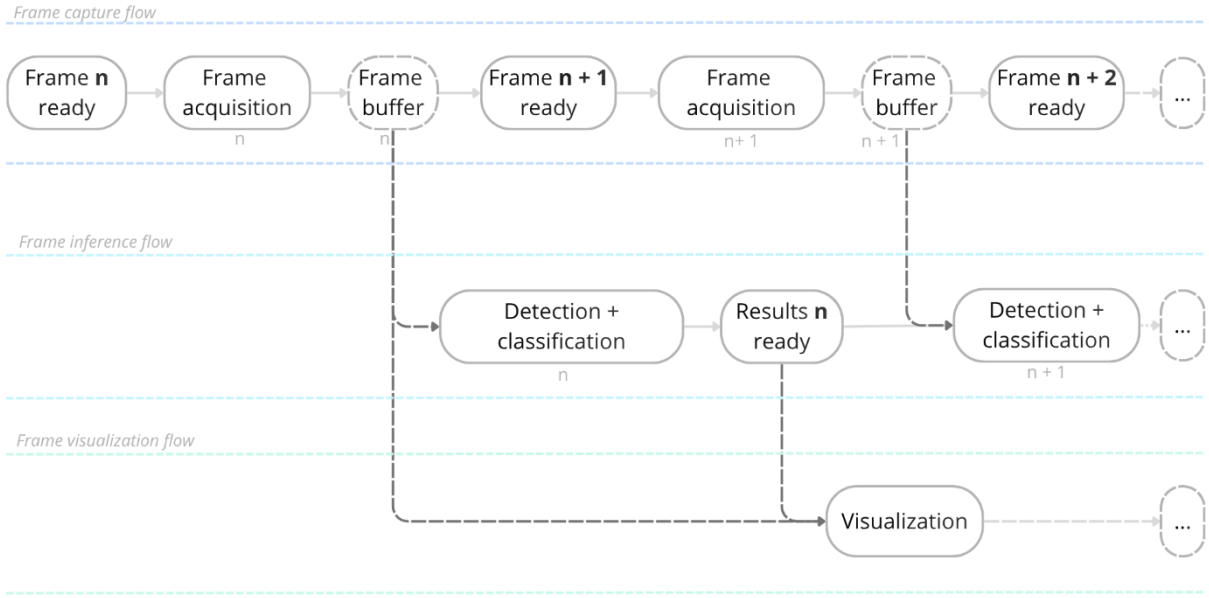

Figure 7 - Multi thread pipeline data flow conceptual diagram.

Although the logical order of the stages remains unchanged, the processing is no longer tied to a single sequential control flow. Instead, the three stages can progress concurrently and synchronize only when shared data must be exchanged.

4.2.2 Multi-Thread Pipeline Architecture

To implement the decoupled pipeline, the application was reorganized into a multi-threaded architecture.

The redesigned system introduces three primary threads:

- Capture Thread;
- Inference Thread;
- Visualization Thread.

The capture thread continuously acquires frames from the camera and inserts them into the processing pipeline.

The inference thread performs the core processing stages of the application. This includes detection pre-processing, detection inference, detection post-processing, and the batched classification stage described in Section 4.1. This thread manages the execution of CUDA kernels and TensorRT inference operations.

The visualization thread handles the generation of debug overlays and rendering of the processed frames.

Each thread therefore corresponds to a specific stage of the processing pipeline and is responsible for the operations associated with that stage.

4.2.3 Thread Coordination and Data Exchange

Separating the pipeline into multiple threads requires mechanisms to coordinate data exchange between stages while preserving correctness.

In a multi-threaded program, multiple execution flows operate concurrently and may access shared memory structures. Without explicit coordination, simultaneous reads and writes to shared data may lead to inconsistent results, commonly referred to as race conditions.

In the redesigned architecture, shared buffers are used to pass frames and associated metadata between threads. Each stage of the pipeline produces data that becomes the input of the subsequent stage.

To ensure safe access to shared resources, synchronization primitives are required. In this implementation, mutexes are used to protect shared data structures. A mutex (mutual exclusion object) guarantees that only one thread at a time can access a protected resource, preventing concurrent reads or writes that could corrupt the exchanged data.

When a frame is captured, the capture thread writes the frame data into a shared buffer and signals its availability to the inference thread. The inference thread processes the frame through the detection and classification pipeline and generates the corresponding results. These results are then forwarded to the visualization thread, which produces the final debug overlays and display output.

Through this mechanism, the different threads exchange data in a controlled manner while preserving the logical ordering of the pipeline stages.

4.3 Multi-Stream GPU Scheduling

The third architectural modification introduced in the redesigned system concerns the scheduling of GPU operations. In the baseline implementation, all CUDA kernels and TensorRT inference calls were issued to a single execution stream, enforcing strictly sequential GPU execution.

The redesigned architecture introduces multiple CUDA streams so that detection and classification operations can be scheduled independently while still respecting the required data dependencies. The following sections first contrast the baseline single-stream behaviour with the redesigned execution model, then describe the stream organization and the synchronization mechanism used to coordinate dependent stages.

4.3.1 Baseline Single-Stream GPU Execution

In the baseline implementation, GPU execution followed a strictly sequential pipeline in which each stage had to complete before the next one could begin.

Conceptually, the execution flow was:

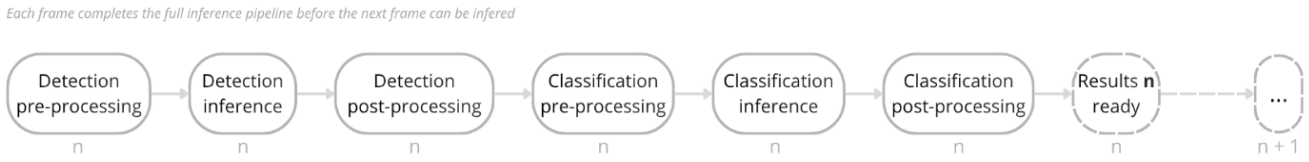

Figure 8 - Single stream data flow conceptual diagram.

All GPU operations were issued to a single execution stream, which enforced this fixed execution order.

The redesigned execution model preserves the same logical stages, but assigns detection-related and classification-related operations to separate streams. Conceptually, the execution flow becomes:

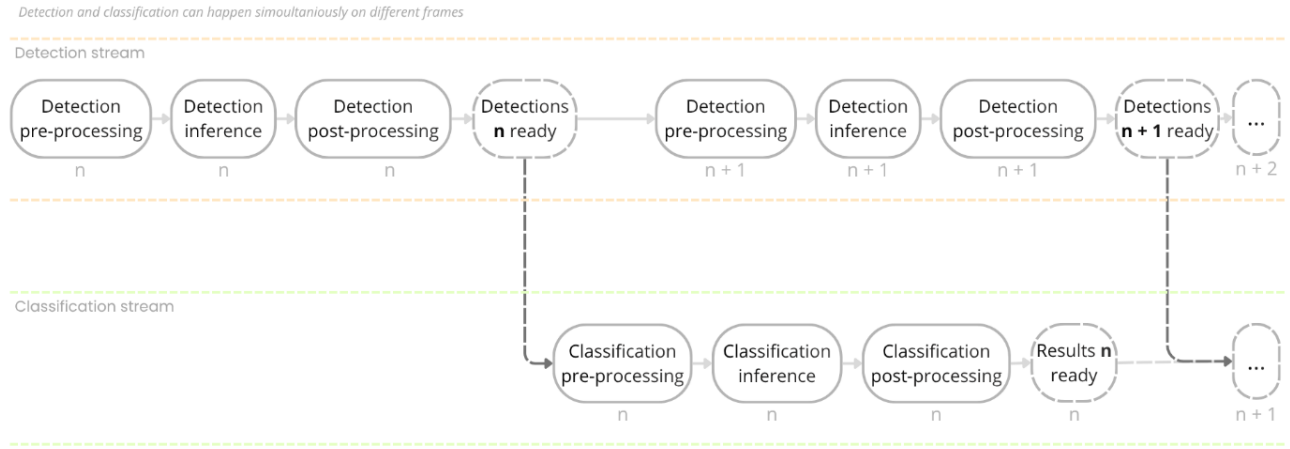

Figure 9 - Multi stream data flow conceptual diagram.

With this configuration, the two stages remain logically dependent but are no longer forced into a single serialized scheduling queue.

4.3.2 Multi-Stream Implementation

To enable this behaviour, the GPU execution pipeline was reorganized using multiple CUDA streams.

In CUDA, a stream represents an ordered queue of operations issued to the GPU. Operations submitted to the same stream are executed sequentially, whereas operations issued to different streams may be scheduled independently by the GPU runtime.

The redesigned system introduces two primary streams: a **detection** stream and a **classification** stream.

All operations related to the detection stage, including pre-processing kernels, YOLO inference, and detection post-processing, are issued to the detection stream.

Similarly, all operations related to the classification stage, including crop pre-processing, classification inference, and classification post-processing, are issued to the classification stream.

Introducing multiple streams requires only minor changes in the program structure. CUDA streams can be created dynamically at runtime, and kernel launches or TensorRT inference calls simply specify the stream in which they should be executed. The logical structure of the pipeline remains unchanged, while the GPU scheduler gains additional flexibility in how operations are executed.

The main adjustment introduced by this architecture is the explicit coordination between streams to ensure that data dependencies are preserved.

4.3.3 Stream Synchronization and Execution Coordination

When operations are executed in different streams, synchronization must be introduced whenever one stage depends on results produced by another stage.

In CUDA, this coordination is commonly implemented using **events**. An event is recorded in a stream once a specific operation has completed. Other streams can then wait for that event before executing operations that depend on the corresponding results.

In the redesigned pipeline, once the detection stage has produced the final detection structures, an event is recorded in the **detection stream**. The **classification stream** waits for this event before beginning its pre-processing and inference operations.

This synchronization is implemented using the CUDA runtime function cudaStreamWaitEvent, which allows one stream to wait for an event generated by another stream without forcing a global synchronization of the GPU. Through this mechanism, the dependency between detection and classification stages is preserved while the two streams remain independently scheduled by the GPU runtime.

4.4 Execution Flow of the Redesigned Pipeline

The modifications introduced in Sections 4.1, 4.2, and 4.3 affect different components of the system, but together they define a single redesigned execution model.

In the redesigned architecture, frame acquisition, inference processing, and visualization operate in separate host threads. Within the inference thread, detection-related GPU operations are issued to the detection stream, while classification-related operations are issued to the classification stream.

Operationally, the workflow can be summarized as follows. The capture thread continuously acquires frames and makes them available to the inference stage. The inference thread performs detection on the detection stream, then uses the resulting bounding boxes to generate batched classifier inputs. These inputs are processed through a single batched classification inference on the classification stream. The resulting action predictions are then combined with the associated bounding boxes to generate the final visualization structures, which are forwarded to the visualization thread.

The visualization thread generates the debug overlays and displays the processed frames independently from the inference stage. As a result, the redesigned system behaves as a staged processing pipeline rather than as a strictly frame-atomic sequential loop.

This integrated execution flow defines the operational structure of the second iteration of the system and serves as the basis for the performance discussion presented in the following section.

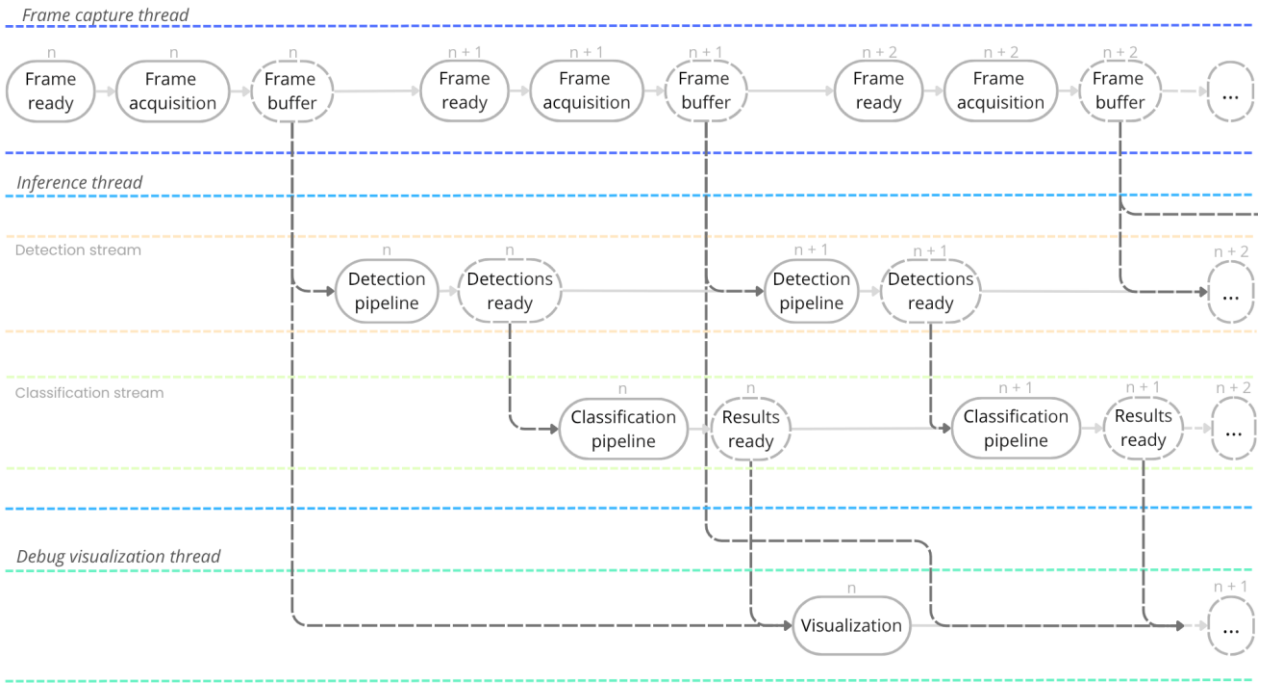

Figure 10 - Pipeline execution data flow conceptual diagram.

4.5 Performance and Scalability Evaluation

The redesign presented in this chapter was motivated by the structural limitations identified in the baseline system analysis of Chapter 3. The purpose of the present section is therefore not to provide a full final-system validation, but to verify that the architectural modifications introduced in this second iteration produce the intended changes in execution behaviour.

For this reason, the following discussion focuses on redesign-level effects: the removal of sequential per-detection classification, the decoupling of host-side stages, and the introduction of overlap between dependent GPU operations. Broader validation and final scalability analysis are deferred to the later evaluation chapters.

The following subsections examine these effects along the same dimensions discussed in Section 3.5, highlighting how the new architecture changes the system's execution characteristics.

4.5.1 Batched Classification and Detection-Count Scalability

The redesigned classification stage processes all detected regions within a frame through a single batched inference operation. Instead of executing a separate classification pass for each detected individual, the detections produced by the detection stage are aggregated into a batch and processed simultaneously by the classifier.

In the redesigned pipeline, the classification stage consists of three main steps:

- batched crop generation and pre-processing of all detected regions;
- a single batched classification inference;
- batched post-processing of the resulting probabilities.

The classifier therefore receives a tensor containing all detected regions as input and produces a corresponding batch of classification outputs within a single inference execution. The batch size is dynamically determined by the number of detections present in the current frame. Profiling of the redesigned pipeline confirms this behaviour: even when multiple individuals are present in the scene, the GPU execution trace shows a single classification inference corresponding to the entire batch of detections.

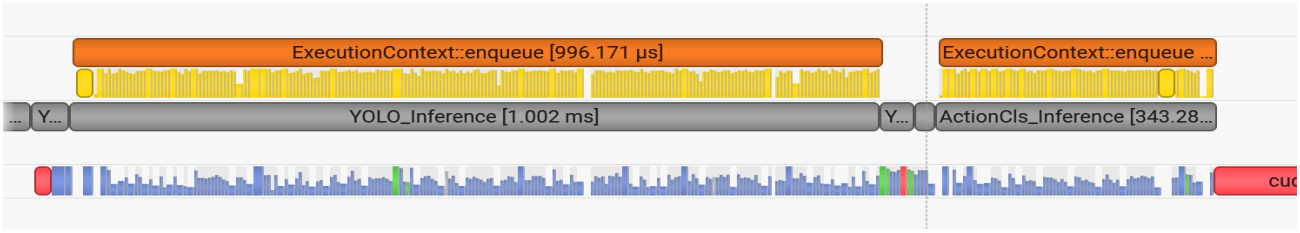

Figure 11 - Nvidia Nsight Systems profiling screenshot showing the full inference pipeline happening on five detected people. Demonstrating the batched classification doing a single inference pass on multiple detections.

This confirms that the redesign successfully removes the repeated per-detection inference pattern visible in the baseline version.

The median execution times measured for the classification stage in the redesigned system are reported below (Table 2).

Table 2 - Action classifier stages latency

Stage	***Median Time (ms)***
ActionCls_Preprocessing_Batched	0.0278
ActionCls_Inference	0.3602
ActionCls_Postprocessing	0.1345

These measurements confirm that the classification stage executes as a single inference operation whose runtime remains approximately constant regardless of the number of detections processed in the frame.

In the baseline system (Section 3.5.3), classification was executed sequentially for each detected individual. For that reason, as shown, the total frame processing time could be approximated as

$$T_{frame} \approx T_{det} + N \cdot T_{cls} \tag{23}$$

where Nrepresents the number of detected individuals.

Because each detection triggered a separate classification inference, scenes containing multiple people produced a proportional increase in classification workload, causing frame latency to grow linearly with the number of detections.

The batched classification strategy therefore modifies the scaling behaviour of the classification stage by aggregating all detections into a single inference execution. As a result, the redesigned system removes the sequential per-detection inference pattern that characterized the baseline implementation, and the sensitivity of classification latency to scene complexity is substantially reduced. A broader discussion of how this property affects final embedded-system scalability is presented later in chapter 8.

4.5.2 Pipeline Overlap and Stage Decoupling

The redesigned architecture modifies the global execution behaviour of the system by introducing both host-side pipeline decoupling and GPU-side concurrent scheduling. These changes affect not only isolated stage timings, but the way frames progress through the system as a whole.

At the host level, the application is divided into three independent threads corresponding to frame acquisition, inference processing, and visualization. Because these stages exchange data through shared buffers rather than waiting for full pipeline completion, frame acquisition can continue while inference is processing previous data, and visualization can proceed independently once results become available.

At the device level, effective overlap is enabled by the use of multiple CUDA streams. Detection-related operations are issued to the detection stream, while classification-related operations are issued to the classification stream. This allows the GPU scheduler to execute operations from different stages more flexibly, provided that the required dependencies are respected.

A key mechanism enabling this behaviour is the use of *cudaStreamWaitEvent* for inter-stream coordination. Once the detection stage completes, an event is recorded in the detection stream, and the classification stream waits only for that specific dependency before continuing. Unlike the host-side synchronization used in the baseline system, this introduces a device-side dependency without globally blocking the CPU or GPU.

Profiling confirms the effect of this change. In the baseline implementation, synchronization points could extend stage timing by more than 1 ms. After replacing the blocking synchronization structure with event-based stream coordination, the measured synchronization overhead dropped below 0.1 ms, allowing the dependent stage to begin almost immediately once the required data became available.

Taken together, these changes transform the execution model from a serialized frame-by-frame progression into a staged pipeline in which different parts of the system can remain active concurrently. The improvement therefore lies not only in faster execution of individual stages, but in improved overall resource utilization and reduced structural idle time.

4.5.3 Overall System Behaviour and Scalability Improvements

The modifications introduced in the second iteration do not fundamentally alter the logical purpose of the pipeline, which remains based on person detection followed by action classification. What changes is the way the system behaves at execution level.

First, batched classification removes the strictly sequential per-detection inference pattern of the baseline system, reducing the dependence of classification cost on the number of detected individuals. Second, host-side decoupling allows acquisition, inference, and visualization to progress more independently. Third, multi-stream scheduling enables dependent GPU stages to be coordinated without enforcing full serialization.

As a result, the redesigned architecture behaves less like an isolated frame-by-frame loop and more like a staged processing pipeline. This improves resource utilization, reduces structural idle time, and provides a more extensible execution model for subsequent development.

Several limitations nevertheless remain. TensorRT engines are still coupled to the configuration used during engine generation, and the current detection pipeline still assumes a single-camera workflow. In addition, the full significance of these architectural improvements becomes more evident when the system is transferred to resource-constrained embedded hardware. These aspects are addressed in the following chapters.

5. Deployment and Cross-Architecture Engineering

The previous chapters described the development and optimization of the detection–classification pipeline on a workstation environment based on an x86-64 architecture and a discrete GPU. As discussed in Chapter 3, the system was accompanied by a deployment strategy based on a self-contained deploy folder packaging the executable, the required runtime libraries, and the inference engines. Within the same architecture class, this design allowed the pipeline to be transferred and executed on different x86 systems without requiring a full development environment.

The next step of the project consists in moving the system from workstation-like environments to an embedded platform. The target device for this deployment is the NVIDIA Jetson Xavier NX module introduced in Section 2.1. Unlike the previous workstation targets, the Jetson platform is based on an ARM architecture and provides a different hardware and software environment. As a result, the deployment strategy previously used for x86 systems cannot be applied directly.

Migrating the pipeline to the Jetson platform therefore introduces a new set of engineering challenges related to architecture compatibility, runtime environment configuration, and hardware interfaces specific to the embedded device. Addressing these constraints requires a dedicated adaptation of the existing pipeline in order to obtain a first deployable embedded version of the system.

The following sections analyse the main constraints introduced by the cross-architecture transition and describe the engineering modifications required to adapt the pipeline to the Jetson platform.

5.1 Cross-Architecture Deployment Constraints

Although the deployment strategy described in Chapter 3 allows the pipeline to be transferred between different workstation systems, this mechanism relies on the assumption that both the development and target machines share the same processor architecture and a compatible runtime environment. Within that setting, the prepared runtime package can be transferred and executed across x86-64 systems, provided that compatible engines are available.

The transition to the Jetson platform breaks this assumption at several levels.

The processor architecture changes. While the development environment is based on x86-64, the Jetson Xavier NX [17] uses an ARM-based processor [49]. These architectures differ in instruction set and binary compatibility, which means that executables compiled for the workstation environment cannot run directly on the embedded platform. Consequently, the binaries generated for workstation deployment cannot be reused as they are on Jetson.

The inference engines must also be regenerated. As discussed in Section 3.5.4, TensorRT engines are generated for the hardware and software environment in which they will execute. Deploying the system

on a new platform such as the Jetson Xavier NX therefore requires local regeneration of the engines starting from the ONNX representations.

The embedded software environment is not identical to the workstation environment. The Jetson unit used in this work is the Leopard Imaging platform described in Section 2.1, which includes a customized set of system libraries. Some components normally available in standard *JetPack* [50] installations, such as the TensorRT tools required for engine generation, were not present in the default system configuration.

The camera acquisition subsystem also differs from the one used in the workstation prototype. The embedded platform is designed to operate with multiple GMSL2 cameras [51] connected through the Jetson CSI interface[52] and managed through the Jetson multimedia framework. This differs significantly from the single-camera OpenCV-based capture approach used during workstation development and requires a corresponding redesign of the acquisition stage.

Finally, the target deployment scenario introduces multiple simultaneous camera streams. While the workstation implementation initially operated on a single video source, the embedded system must support concurrent acquisition and processing of several cameras. This affects not only capture, but also the detection pipeline and the associated memory organization.

Together, these differences define the main constraints that must be addressed to obtain a first deployable version of the pipeline on the Jetson platform.

5.2 Cross-Architecture Build and Deployment Strategy

Once the constraints introduced by the Jetson transition are established, the next step is to define a build and deployment strategy capable of addressing them while preserving the overall development philosophy adopted in the previous chapters. The objective is to keep development centered on the workstation environment while still generating deployable artifacts specifically targeting the ARM embedded platform.

This requirement is more demanding than the workstation-to-workstation deployment case described in Chapter 3, because the target architecture is no longer x86-64 but ARM aarch64. The previous build strategy must therefore be extended so that both compilation and packaging become explicitly architecture aware.

The following sections describe the general idea behind this strategy and the modifications introduced in the build system to implement it.

5.2.1 General Strategy for Cross-Architecture Deployment

The fundamental assumption adopted in the previous deployment workflow remains valid: the build system should be responsible for producing an executable and a corresponding deploy folder

compatible with the final target platform. In other words, the general deployment philosophy does not change. What changes is the target architecture of the generated artifact.

For x86 systems, this meant compiling an x86 executable and packaging the x86 runtime libraries required by that executable. For Jetson deployment, the same logic must now be applied to an ARM aarch64 target. The executable must therefore be compiled against ARM-compatible headers and libraries, and the deploy folder must contain ARM-compatible runtime components.

The first step required to achieve this was to extend the build system so that it could explicitly target multiple architectures. Rather than maintaining separate codebases or separate manual build workflows, the same project structure was preserved and both CMake and the Makefile were extended to distinguish between native x86 builds and Jetson-oriented aarch64 builds.

At a conceptual level, the strategy is therefore based on two coordinated requirements:

1. the build configuration must be able to resolve the correct include directories, libraries, and CUDA target settings for the selected architecture;
2. the deployment logic must be able to package runtime components appropriate for that same target architecture.

This means that the build system no longer produces only a native artifact for the host machine, but can instead generate a deployment package specialized for the intended destination platform. In this way, the same development environment can be used to produce either x86 or ARM deployable versions of the system.

5.2.2 Build-System Implementation for AArch64 Deployment

The concrete implementation of this strategy was based on the interaction of four elements: a containerized build environment, an ARM sysroot extracted from the Jetson software stack, architecture-aware CMake logic, and a Makefile extended with target-specific deployment modes.

The central tool used to enable this workflow was a Docker container. [53] A container is an isolated software environment that packages a specific filesystem, toolchain, and dependency set while remaining executable on the host system. [54] In practice, this makes it possible to reproduce a target software environment in a controlled and portable way, without modifying the host operating system. In this work, the container provided an environment aligned with the Jetson software stack, allowing the cross-build process to be performed on the workstation while targeting the ARM embedded platform.

Inside this container, a dedicated script named *build_cross_aarch64.sh* was used to orchestrate the cross-build procedure. The script first defines the main cross-compilation context: it identifies the TensorRT library location, selects a Jetson sysroot extracted from the target filesystem, and defines the ARM OpenCV configuration path inside that sysroot. It then checks that the sysroot is actually present and usable before proceeding with the build.

Once this environment has been validated, the script enters the project directory, removes any previous build directory, and invokes the existing *Makefile* with explicit aarch64 target parameters. These parameters identify the build as Jetson-oriented, enable the Jetson-specific deployment logic, and forward to *CMake* the dedicated toolchain file, the GPU architecture corresponding to Xavier NX (SM 72), and the OpenCV path derived from the sysroot. In this way, the shell script does not replace the build system: it acts as the outer coordination layer that injects the correct ARM target configuration into the *Makefile* and *CMake* workflow.

The *CMake* configuration is correspondingly extended so that dependency resolution depends on the target architecture rather than on the host architecture. When the build targets aarch64, *CMake* no longer uses the normal host-side discovery logic. Instead, it resolves include directories and libraries through paths associated with the Jetson sysroot. This affects several components of the project, including GStreamer, GLib, TensorRT, CUDA, and Jetson-specific libraries used for buffer handling and EGL-based integration. The same architecture-aware logic is also used to select the correct CUDA target architecture, ensuring that the compiled binary is specialized for the Xavier NX GPU rather than for the host machine.

The *Makefile* complements this mechanism at the deployment level. The original native deployment logic introduced in Chapter 3 is preserved, but it is extended with additional target-specific modes for Jetson deployment. The result is that the same *Makefile* can now generate deployable artifacts for different destination architectures, including both x86 and ARM. For the Jetson case, two deployment variants were implemented. The first targets a standard JetPack-style environment and avoids copying libraries that are already expected to be available on the device. The second is designed for the specific Leopard Imaging setup used in this work, where some expected runtime components were not present by default; in this case, the required libraries are selectively bundled into the deploy folder to guarantee execution.

This distinction is important because it shows that the build system is not merely compiling an ARM binary, but is also capable of adapting the deployment package to the characteristics of the target runtime environment. The same deployment philosophy established earlier is therefore preserved: the build process still generates a structured runtime artifact containing the executable, the engines, the launch scripts, and the runtime libraries required for execution. The difference is that this process is now architecture-aware and can be specialized for either native x86 deployment or ARM Jetson deployment.

The resulting workflow is therefore layered and cooperative. The container provides a reproducible Jetson-aligned software environment. The shell script defines the target-specific cross-build context and triggers the build. *CMake* resolves the correct ARM dependencies and compilation properties. The *Makefile* then packages the resulting executable into the appropriate deploy mode for the selected target platform. Together, these elements create a cross-architecture build and deployment strategy that allows development to remain on the workstation while still producing a deployable ARM artifact for the embedded system.

5.3 Camera Interface Integration

Beyond the cross-architecture build and deployment issues discussed in the previous section, moving the system to the Jetson platform also required a redesign of the frame acquisition stage. The capture mechanism used in the workstation prototype was developed for a generic single-camera setup and relied on the software interfaces available in that environment. On the embedded platform, however, both the camera hardware and the acquisition software stack differ significantly.

As a result, frame acquisition becomes a central architectural issue in the embedded deployment. The goal is not only to restore functional camera support on Jetson, but also to align the acquisition path with the multimedia framework and memory organization of the platform so that captured frames can be integrated efficiently with the GPU-oriented inference pipeline.

The following sections first analyse the limitations of the original acquisition strategy in the embedded context and then describe the redesigned Jetson-oriented acquisition pipeline.

5.3.1 Embedded Camera Acquisition Constraints

In the workstation prototype presented in Chapter 3, the acquisition stage relied on a simple and flexible strategy: frames were captured through a generic interface, stored in host memory, and then transferred to the GPU before entering pre-processing and inference. This approach was sufficient for the initial prototype because it prioritized development simplicity and portability across different desktop environments.

However, this design implicitly assumes that host-side frame acquisition followed by a host-to-device transfer is acceptable from a performance perspective. On desktop systems with powerful discrete GPUs and abundant memory bandwidth, this overhead can remain relatively modest, especially in the single-camera case. On the Jetson Xavier NX, the situation is different. As discussed in Chapter 2, the platform operates under tighter power and memory-bandwidth constraints, so repeatedly transferring full-resolution frames from CPU memory to GPU memory becomes more expensive relative to the available computational budget.

This issue becomes even more significant in the intended multi-camera deployment scenario. Repeating the same host-side acquisition model for several concurrent streams would multiply the number of memory transfers required during each frame cycle and would increase pressure on an already constrained memory subsystem.

At the same time, the Jetson platform provides a multimedia framework designed to expose camera buffers through a path more closely aligned with GPU processing. Rather than treating the captured frame as a conventional CPU image that must later be copied to the device, the platform allows frames to be obtained through memory structures that can be made directly accessible within the GPU processing workflow.

For this reason, the original workstation-oriented acquisition strategy was not simply reused on the embedded platform. Instead, the capture stage was redesigned to integrate with the Jetson multimedia framework and to make the acquired frames available to the GPU without relying on the same conventional host-to-device copy path.

5.3.2 GStreamer-Based Acquisition Strategy

To address the constraints described in the previous sub-chapters, the frame acquisition stage was redesigned around the GStreamer multimedia framework [55] and the camera interfaces provided by the Jetson platform.

GStreamer is a modular pipeline-based multimedia framework in which video processing is represented as a sequence of connected elements. Each element performs a specific task, such as camera sourcing, decoding, format conversion, or application-level frame delivery. On Jetson platforms, this framework forms the standard interface between the camera subsystem and user-level applications, while also exposing access to hardware-accelerated multimedia components and platform-specific buffer management mechanisms.

The redesign of the capture stage leverages this framework to align frame acquisition with the native Jetson multimedia pipeline. In particular, the camera stream is obtained through Jetson-specific GStreamer elements capable of interfacing directly with the embedded camera subsystem and producing frames compatible with the multimedia memory path of the platform.

The key advantage of this approach lies in how captured frames are handled in memory. Instead of receiving camera frames as conventional host-resident images that must later be copied to the GPU, the Jetson multimedia framework allows frames to be produced within buffer structures designed to interact efficiently with GPU processing components. These buffers can then be mapped through the CUDA–EGL [56] interoperability mechanisms provided by the platform, enabling the frame data to be made available to device-side processing without performing a conventional host-to-device copy.

In practical terms, this redesign transforms the capture stage into a GPU-oriented front-end for the inference pipeline. Frames acquired through the camera subsystem are delivered through the GStreamer pipeline into multimedia buffers that can be mapped into the CUDA execution context. Once mapped, the frame data becomes directly accessible to the GPU stages that follow in the pipeline.

An important architectural aspect of this redesign is that it remains localized to the capture thread. The rest of the system, including pre-processing, inference, and post-processing, can remain largely unchanged because the capture thread continues to expose frames through the same internal synchronization mechanisms already used in the original pipeline. What changes is the internal implementation of the acquisition stage.

The concrete implementation of this GPU-oriented acquisition mechanism, is described in the following sections.

5.3.3 GPU-Compatible Frame Acquisition Pipeline

The redesign of the capture stage required a detailed understanding of the Jetson multimedia framework and of the mechanisms through which camera buffers can be shared between the acquisition subsystem and GPU processing components. Unlike the workstation prototype, where frames were obtained as ordinary CPU images, the Jetson platform exposes camera streams through a specialized multimedia pipeline designed to support efficient hardware interaction.

A central concept in this pipeline is the use of **NVMM buffers [57]**. NVIDIA Multimedia Memory refers to buffer objects allocated and managed by the Jetson multimedia subsystem. These buffers are implemented as **DMA-capable memory regions**, meaning they can be accessed directly by hardware components without requiring CPU-mediated copies. In practical terms, this allows the camera interface, multimedia processing elements and the GPU, to interact with the same memory region while avoiding memory transfers.

The importance of this mechanism becomes clear when compared to the acquisition strategy used in the workstation prototype. While acceptable in a desktop environment, such host-to-device transfers become significantly more expensive on embedded hardware where memory bandwidth is limited. By contrast, NVMM buffers allow the camera subsystem to produce frames directly inside memory regions compatible with GPU processing, enabling a more efficient data path through the system.

To access these buffers, the capture stage was redesigned around a **GStreamer pipeline** integrated with the Jetson camera stack. GStreamer represents multimedia processing as a chain of connected elements, each responsible for a specific transformation of the data stream. In this work, the pipeline begins with the nvarguscamerasrc element, which interfaces directly with the Jetson camera subsystem and exposes the camera stream to the multimedia framework.

The pipeline then specifies that the frames should be produced in NVMM-backed buffers through the capability declaration *video/x-raw(memory:NVMM)*. This step ensures that the captured frames remain inside the multimedia memory path rather than being converted into conventional CPU images. The raw camera frames are initially produced in *NV12* format, a common representation used by camera pipelines due to its compact memory layout.

A hardware-accelerated conversion stage is then performed through the *nvvidconv* element. This component is responsible for transforming the incoming frames into a format suitable for the downstream processing pipeline. In the present implementation, the conversion produces RGBA frames while preserving the NVMM memory allocation. The resulting frames therefore remain inside the multimedia buffer system even after the format transformation.

The final stage of the pipeline is an *appsink* element. The purpose of this component is to deliver frames from the GStreamer pipeline to the application layer. Unlike display sinks or encoding sinks, *appsink* allows the application to explicitly retrieve frames from the pipeline through a pull-based interface. This design fits naturally with the architecture of the existing system, where frame acquisition occurs inside a dedicated capture thread.

The capture thread retrieves frames using the *gst_app_sink_pull_sample* function, which blocks until a new frame becomes available. Each retrieved sample contains a *GstBuffer* representing the multimedia buffer produced by the pipeline. Because the pipeline operates entirely in NVMM mode, this buffer contains a pointer to an *NvBufSurface* structure [58] rather than to a conventional host-resident image.

The use of NVMM buffers at this stage is crucial. The frame is not copied into CPU memory and does not undergo a host-to-device transfer before reaching the inference pipeline. Instead, the captured image remains inside the multimedia buffer structure that was originally allocated by the Jetson multimedia subsystem. The capture stage therefore acts as a bridge between the camera pipeline and the GPU processing stages without introducing additional memory transfers.

Once the buffer has been obtained and interpreted as an *NvBufSurface*, the capture thread proceeds with the integration step that allows the GPU to access the frame data. This step relies on the interoperability mechanisms provided by the Jetson platform between multimedia buffers, the EGL graphics interface, and the CUDA runtime. The details of this integration are described in the following section.

5.3.4 CUDA–EGL Integration and Device-Side Frame Access

After acquiring a frame from the GStreamer pipeline in the form of an NVMM buffer, the next challenge is to expose the contents of that buffer to the CUDA execution environment used by the inference pipeline. This is achieved through the interoperability mechanisms available on Jetson platforms between multimedia buffers, the EGL graphics framework, and CUDA.

The first step of this integration is the mapping of the *NvBufSurface* structure obtained from the GStreamer buffer. The surface is initially mapped for device access through the Jetson multimedia API using *NvBufSurfaceMap*, followed by a synchronization step ensuring that the memory region is ready for GPU usage. Once the surface has been prepared, the multimedia framework allows it to be associated with an ***EGLImage*** representation.

EGL is a graphics interoperability framework designed to enable resource sharing between different APIs operating on the same hardware device. Within this framework, an ***EGLImage*** acts as a portable representation of image data that can be shared across multiple subsystems. On Jetson platforms, this capability is used to bridge the multimedia buffer infrastructure and CUDA.

The capture thread therefore invokes *NvBufSurfaceMapEglImage* to obtain an *EGLImage* representation of the NVMM buffer. At this stage the frame exists simultaneously as a multimedia buffer and as a graphics-compatible image resource. This *EGLImage* can then be registered with CUDA using the function *cuGraphicsEGLRegisterImage*.

Registering the EGL image with CUDA creates a **CUDA graphics resource**, allowing CUDA kernels to access the underlying image memory. The resource is subsequently queried using *cuGraphicsResourceGetMappedEglFrame*, which provides a *CUeglFrame* structure containing the

device-accessible pointer to the frame data together with the pitch information describing its memory layout.

The frame data obtained through this mechanism resides directly in GPU-accessible memory. As a result, CUDA kernels can operate on the image without requiring a prior host-to-device memory transfer. This property is the key advantage of the redesigned acquisition pipeline: the frame captured by the camera subsystem is delivered to the GPU processing stages without passing through a CPU staging buffer.

In the present implementation, the device pointer obtained from the *CUeglFrame* structure is used as the input of a lightweight CUDA kernel that performs a format adaptation step before the frame enters the rest of the inference pipeline. This kernel transforms the pitched RGBA representation produced by the multimedia pipeline into the linear BGR format expected by the pre-processing stage of the neural network. Although this conversion step is necessary to maintain compatibility with the existing GPU pre-processing routines, it does not involve host-side memory transfers and therefore preserves the advantages of the device-resident acquisition strategy.

Once the CUDA processing step has completed, the graphics resource is released through *cuGraphicsUnregisterResource*, and the EGL image mapping is removed. The capture thread then signals the availability of a new frame to the rest of the pipeline using the existing synchronization primitives that coordinate the capture, inference, and visualization threads.

Through this sequence of operations, the redesigned capture stage establishes a direct bridge between the Jetson camera subsystem and the CUDA-based inference pipeline. Frames produced by the camera hardware travel through the multimedia framework as NVMM buffers, are mapped into an EGL image representation, and then become accessible to CUDA kernels as device memory. The resulting capture path makes the frame available to the GPU without a conventional host-to-device copy and therefore provides an acquisition mechanism better aligned with the constraints of the embedded deployment context.

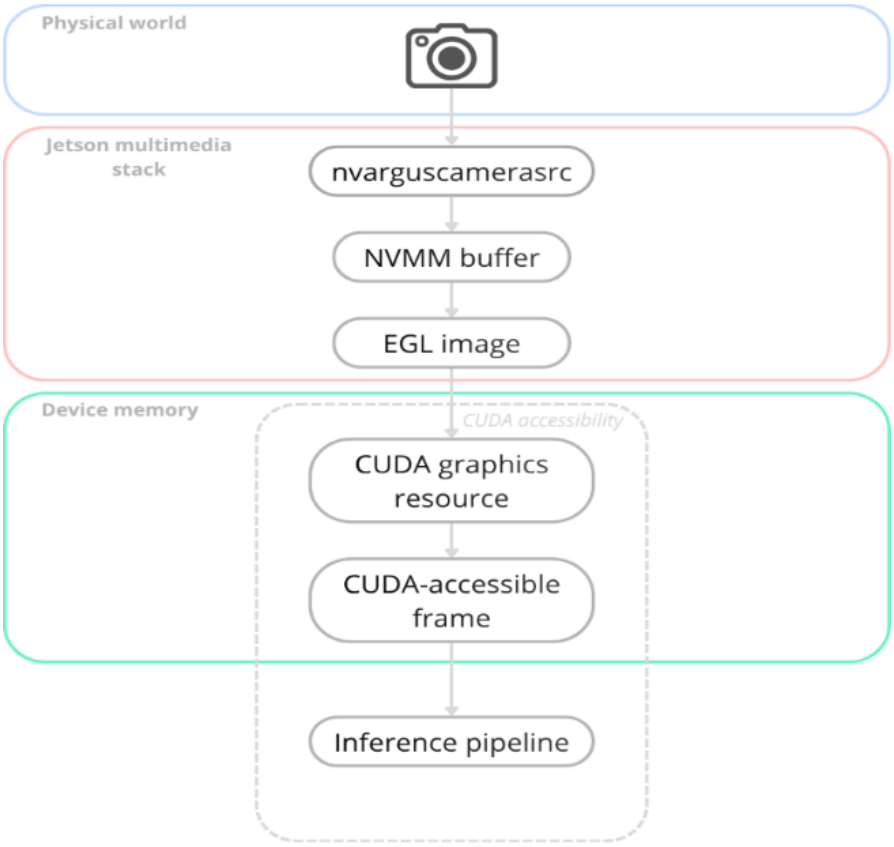

Figure 12 - Frame flow conceptual diagram, from camera acquisition to device memory with CUDA accessibility.

5.4 Multi-Camera Detection Adaptation

With the capture stage successfully redesigned to support the Jetson multimedia pipeline, the next architectural challenge concerns the integration of multiple camera streams into the inference pipeline. The target deployment scenario requires the system to operate with several cameras simultaneously, providing broader environmental coverage than the single-camera prototype used during the early development stages.

The inference architecture itself must now evolve to support multiple concurrent inputs. The original detection pipeline was designed around a single frame per inference iteration, and several parts of the system implicitly relied on this assumption. As a result, simply enabling multiple cameras at the acquisition stage would not be sufficient; the detection pipeline, memory management structures and visualization subsystem, must all be adapted to correctly handle multiple frames from different cameras.

To address this requirement, the system was extended to support multi-camera inputs through a batched detection strategy. This required modifications to several components of the pipeline, including pre-processing, inference, post-processing, memory allocations and debugging visualization stage. The following sections describe the architectural implications of introducing multiple cameras and the modifications required to ensure that the system can process them efficiently.

5.4.1 Architectural Implications of Multi-Camera Input

The original detection pipeline was designed with the implicit assumption that each inference iteration would operate on a single frame. Under this model, the capture thread produces one frame, which is then subject to the detection pre-processing, passed through the detection network and post-processed to obtain the final detections structure.

Introducing multiple cameras fundamentally changes this assumption. The system must now process multiple frames per iteration, each originating from a different camera. This change affects several parts of the architecture.

First, memory management must be adapted to accommodate multiple frames simultaneously. Buffers originally allocated for a single frame must be resized to store data for all active cameras. This includes memory used during pre-processing, the input tensors passed to the detection network, and the structures used to store detection results.

Second, the pre-processing stage must be able to operate on frames from multiple cameras. In the original implementation, pre-processing kernels were designed to transform a single image into the tensor representation expected by the neural network. When multiple cameras are introduced, pre-processing must be performed independently for each frame, while still producing a unified tensor structure compatible with the batched inference interface of the detection model.

Third, the neural network itself must be configured to operate on batched inputs. Similar to the strategy adopted earlier for the classification stage, the detection network must accept a batch dimension corresponding to the number of cameras. This allows multiple frames to be processed simultaneously within a single inference execution, avoiding the overhead associated with repeatedly invoking the inference engine for individual frames.

Post-processing must also be adapted accordingly. The detection results produced by the neural network now correspond to multiple frames within the batch, and each set of detections must be associated with the correct camera. To maintain this association, the internal data structures used to propagate detection results through the pipeline were extended to include the identifier of the source camera.

Finally, the original visualization thread assumed that a single frame was being processed at any given time. When multiple cameras are active, the visualization system must be able to render and display detection results for each camera independently while preserving the correct correspondence between frames and their associated detections.

These architectural considerations highlight that supporting multiple cameras is not a localized modification, but rather a system-wide adaptation affecting several interconnected components of the pipeline.

5.4.2 Batched Detection Pipeline Implementation

Among the modifications required to support multiple cameras, the most significant concerns the detection stage of the pipeline. The original implementation was designed to process a single frame per inference iteration, which means that extending the system to multiple cameras requires a fundamental restructuring of the detection workflow.

A straightforward solution could consist of executing the entire detection pipeline sequentially for each camera frame. In such an approach, the system would simply loop over the available camera streams and run the detection stage independently for each frame. While technically feasible, this approach would introduce the same architectural limitations previously encountered during the design of the classification stage. Repeatedly invoking the inference engine for individual frames would generate unnecessary overhead, reduce GPU utilization, and complicate the synchronization between pipeline stages.

For these reasons, this solution was not considered as the primary implementation strategy. Instead, the detection stage was redesigned to operate on **batched inputs**, following the same architectural principle previously adopted for the classification pipeline. By grouping frames from multiple cameras into a single batch, the GPU can process them within a single inference execution, improving hardware utilization and maintaining architectural consistency with the rest of the system.

The first step required to enable this strategy was the modification of the original YOLO ONNX model to support **dynamic batching**. This modification follows the same procedure previously described for the classification model in Section 4.1.2. In practice, the batch dimension of the network input tensor was replaced with a dynamic dimension, allowing the inference engine to process multiple frames simultaneously.

The resulting model therefore accepts an input tensor of shape:

$$N \times 3 \times 256 \times 256$$

where *N* represents the number of camera frames processed in each inference iteration.

Unlike the classification pipeline, however, the batch size used by the detection stage remains constant during runtime. The number of cameras is determined during system initialization and does not change while the application is running. As a result, the binding dimensions of the inference engine only need to be configured once during startup, after which the detection stage consistently operates on batches containing one frame per camera.

Once the detection model was adapted to support batched inference, the remainder of the detection pipeline required corresponding modifications. This was done following the same architectural principle previously adopted for batched classification: the CUDA kernels used throughout the detection pipeline were modified to operate on three-dimensional execution grids, where the third grid dimension represents the camera index within the batch. This design allows a single kernel invocation to process frames from all cameras simultaneously.

Thanks to this modification, the pre-processing stage places the frames from all active cameras into the appropriate position within the batched input tensor while executing in parallel on the GPU. Each frame occupies a contiguous region of memory corresponding to its batch index, ensuring compatibility with the input layout expected by the detection network.

In addition to the kernel-grid modification, the post-processing stage required an extra adaptation, because the detection results produced by the network must be associated with the camera frame from which they originate. Since the inference stage processes a batch of frames at once, the post-processing stage must explicitly preserve the relationship between each detection and its corresponding camera.

To achieve this, the detection structure used to store inference results was extended to include an additional field representing the camera identifier. This value is derived directly from the **batch index of the processed frame**, which must be reconstructed during the detection extraction step.

The network output contains predictions for **all frames in the batch**, organized as repeated anchor groups for each batch element. Consequently, the extraction kernel was modified so that each thread processes one anchor prediction belonging to a specific batch element.

The kernel therefore operates over a total number of predictions equal to the product of the batch size and the number of anchors. Each thread computes its **global index** within this combined set of

predictions. From this index, the kernel derives two values: the anchor index within the frame and the batch index corresponding to the camera from which the frame originated. The batch index is then stored inside the detection structure.

In this way, each detection produced by the network carries the identifier of the camera frame from which it originated. This information is subsequently used by downstream stages of the pipeline, including the classification stage and the debug visualization subsystem, to correctly associate detection results with their corresponding camera stream.

Through these modifications, the detection stage evolves from a single-frame inference component into a batched multi-camera processing module. By adopting the same batching strategy previously used for classification and extending the detection data structures to preserve camera identity, the system can efficiently process multiple camera streams while maintaining a coherent and scalable pipeline architecture.

5.4.3 Supporting Runtime and Memory Adjustments

Several additional modifications were required within the program to ensure that the overall system could correctly manage multiple camera streams. Compared to the detection pipeline restructuring described in the previous section, these changes are conceptually simpler, but they are essential for maintaining the consistency of the data flow across the application.

The first set of modifications concerns the allocation of memory buffers used throughout the pipeline. In the original single-camera implementation, buffers storing frames, intermediate pre-processing results, and inference inputs were allocated assuming the presence of a single image at a time. With the introduction of multiple camera streams, these buffers were resized so that their capacity scales with the number of cameras active in the system. In practice, this means that memory allocations that previously reserved space for a single frame were extended to accommodate one frame per camera, preserving the same layout but repeating it for each element of the batch.

A second modification concerns the frame acquisition stage described in Section 5.3. Instead of managing a single camera pipeline, the system now initializes one GStreamer pipeline for each connected camera. Each pipeline is configured with its corresponding sensor identifier and produces frames independently from the others. The frame capture thread is therefore responsible for polling multiple pipelines and retrieving the most recent frame from each one before handing them to the inference stage. In this way, the capture subsystem produces a set of frames corresponding to the current time step of the system, which can then be processed together by the batched detection pipeline.

Another important consequence of introducing the camera index into the detection structure is the simplification of subsequent processing stages. In particular, the classification stage can directly exploit this information to extract image crops corresponding to detected objects from the correct source frame.

During this stage, the system generates cropped image regions corresponding to the bounding boxes produced by the detection network. Because each detection carries its associated camera identifier, the classification pipeline can determine precisely which frame must be used as the source for each crop. This enables the system to generate all object crops from all cameras while preserving their relationship with the original frames.

This design also allows the crops to be stored in **contiguous memory**, which is a critical requirement for efficient inference execution. TensorRT inference engines expect input tensors to be stored as contiguous buffers. By generating crops sequentially and placing them one after the other in memory, the system can construct the classification input tensor directly without introducing gaps between samples.

Without the camera identifier propagated from the detection stage, this memory layout would be significantly more difficult to maintain. The system would not be able to determine from which frame a bounding box originated, and additional metadata or memory padding would be required to preserve the frame association. Such workarounds would either complicate the implementation or risk feeding invalid memory regions to the classification engine. By contrast, the explicit camera index allows the system to maintain a clean and deterministic memory layout while preserving the correct relationship between detections and frames.

Finally, the camera identifier is preserved within the data structures that propagate detection and classification results to the final stages of the pipeline. This information is particularly important for the debug visualization subsystem, which must display the correct annotations on each camera stream. By retaining the camera index throughout the entire processing chain, the system ensures that every detection and classification result can be unambiguously mapped back to the frame from which it originated.

5.4.4 Multi-Camera Debug Visualization

The final adaptation was done on the debug visualization stage. In the original single-camera implementation, the visualization thread received one frame and one set of associated detections, then rendered the corresponding overlays onto that single image. Once the system was extended to support multiple simultaneous camera streams, this model was no longer sufficient.

A first requirement was to ensure that the visualization thread could retrieve the current frames corresponding to all active cameras. Since the embedded pipeline remains GPU-oriented up to the final stages, the undistorted frames for all cameras are stored on device memory as a contiguous block, where each frame occupies a fixed memory region of size *frame_bytes*. The visualization stage therefore begins by iterating over the active cameras, computing the device pointer corresponding to each frame, and issuing an asynchronous *cudaMemcpyAsync* transfer from device memory to a CPU-side cv::Mat buffer.

This design preserves a regular memory layout across the entire pipeline. Because frames are stored contiguously in memory, the visualization thread can retrieve them through a simple offset computation based on the camera index. Once all transfers have been scheduled, a synchronization on the visualization stream ensures that the copied frames are fully available on the CPU before composing the final debug image.

Rather than opening an independent visualization window for each camera, the implementation constructs a single **side-by-side composite image**. The CPU-side frame corresponding to each camera is copied into a dedicated horizontal region of a larger image buffer whose width is equal to the frame width multiplied by the number of cameras. In this way, the resulting debug image contains all camera views arranged in a single row. This choice simplifies visual validation by allowing the full multi-camera output to be observed within one unified display.

A further optimization is then applied to reduce the rendering cost. Since the composite image can become large when multiple full-resolution frames are displayed side-by-side, the final image is downscaled before overlays are drawn. In the current implementation, the full composite frame is resized using area interpolation to produce a smaller display image. This reduces the computational cost of rendering and displaying the debug output while preserving sufficient visual clarity for qualitative inspection.

Once the composite image has been prepared, the overlay stage uses the propagated detection metadata to draw bounding boxes and labels in the correct position. At this point, the camera index field introduced earlier in the detection structure becomes essential. Each action-visualization element retains the identifier of the source camera, together with the bounding-box coordinates and the predicted class. The visualization thread uses this camera index to transform local bounding-box coordinates into **global composite coordinates**.

More precisely, if a detection belongs to camera k, its horizontal coordinates are shifted by an offset equal to k times the camera width, while the vertical coordinates remain unchanged. This places the detection inside the correct camera region of the side-by-side composite image. The resulting bounding box is then scaled according to the resize factor applied to the composite frame before being drawn on the final display image.

This design ensures that the final debug visualization remains fully consistent with the multi-camera inference pipeline. The detection and classification stages may operate on batched inputs and on device-side data structures, but the final debug output still provides a clear and interpretable visual representation of the system state.

5.5 Baseline Embedded Pipeline Analysis

After adapting the system architecture to the Jetson Xavier NX platform and enabling multi-camera operation, the resulting implementation provided a fully functional embedded baseline version of the pipeline. All major components of the system were successfully deployed, including camera acquisition

through GStreamer, batched multi-camera detection, batched classification, and multi-camera debug visualization.

However, the transition from a high-performance desktop GPU to an embedded platform introduces significant differences in available computational resources. For this reason, a new performance analysis was conducted to characterize the runtime behaviour of this first embedded baseline and to identify the main bottlenecks preventing the system from reaching the desired responsiveness.

The insights obtained from this analysis directly motivate the optimizations introduced in the following chapter.

5.5.1 Baseline Embedded Performance Measurements

To evaluate the runtime characteristics of the baseline embedded implementation, the system was profiled on the Jetson Xavier NX platform using the three available power modes: 10 W, 15 W, and 20 W. These modes directly affect the operating frequencies of the CPU and GPU subsystems and therefore have a significant impact on the achievable performance.

The median execution times measured for the main stages of the inference pipeline are reported in Table 3.

Table 3 - Baseline embedded measurements

Stage	***10 W***	***15 W***	***20 W***
YOLO Pre-processing	0.14 ms	0.14 ms	0.12 ms
YOLO Inference	49.89 ms	20.94 ms	18.94 ms
YOLO Post-processing	0.35 ms	0.37 ms	0.31 ms
Classifier Pre-processing	0.23 ms	0.23 ms	0.20 ms
Classifier Inference	16.85 ms	7.30 ms	6.39 ms
Classifier Post-processing	0.14 ms	0.15 ms	0.11 ms
Debug Visualization	~23 ms	~13 ms	~10 ms

The results show that even at the highest power mode, YOLO inference alone requires approximately **19 ms**, while the classification stage requires an additional **6 ms**. The pre-processing and post-processing kernels remain comparatively inexpensive, with execution times consistently below **1 ms**.

When operating in the lower power modes, the inference cost increases significantly. In the **10 W configuration**, the YOLO inference stage alone requires approximately **50 ms**, while the classifier requires approximately **17 ms**. Under these conditions, the two neural networks together account for the majority of the processing time of the inference thread.

For numerical reference, in the 20 W configuration the two inference stages together require approximately 25.3 ms, while all pre-processing and post-processing kernels combined account for less than 1 ms. This means that over **97% of the GPU computation time** is spent executing neural network inference.

The debug visualization stage also introduces a noticeable cost. Because the visualization thread still performs image composition and overlay operations on the CPU, its execution time ranges between approximately 10 ms and 23 ms depending on the selected power mode. While acceptable in the workstation version, this overhead becomes considerably more significant in the embedded baseline and contributes to the gap between the current system behaviour and the desired real-time target.

5.5.2 Visual Validation of the Multi-Camera System

In addition to numerical profiling, the behaviour of the baseline embedded system was also evaluated through visual inspection of the debug output. This validation step is particularly important for a real-time perception pipeline, because the perceived responsiveness of the system contributes directly to its practical usability.

The system successfully produced correct detection and classification outputs for all active camera streams, confirming the correctness of the multi-camera inference pipeline implemented in the previous sections. The side-by-side debug visualization allowed simultaneous observation of all camera feeds together with their corresponding bounding boxes and classification labels.

However, the overall responsiveness of the system remained far from the desired real-time target. In the 20 W configuration, the system produced output close to 20 frames per second, which was sufficient for debugging and qualitative validation but still below the intended 30 FPS objective. In the lower power modes, the behaviour degraded more noticeably: in 15 W mode the observed rate dropped to approximately 15 frames per second, while in the 10 W configuration the output fell below 10 frames per second, producing clearly sluggish visual feedback.

These observations confirm that the baseline embedded implementation is functionally correct, but that its responsiveness remains insufficient for the intended real-time behaviour. The following section therefore analyses the factors responsible for this limitation in order to identify the most relevant optimization opportunities.

5.5.3 Runtime Bottleneck Analysis

The performance measurements and visual observations presented in the previous sections show that, although the baseline embedded pipeline operates correctly, it does not yet achieve the desired real-time responsiveness under most operating conditions. The purpose of the present subsection is therefore to identify the main bottlenecks responsible for that behaviour.

The slowdown is not caused by a single limiting factor, but by the interaction of several bottlenecks affecting different parts of the pipeline. Some of these derive from architectural choices that were acceptable in the workstation implementation but become more problematic in the embedded environment, while others are a direct consequence of the reduced computational resources available on the Jetson platform.

One of the first bottlenecks identified during the analysis concerns the debug visualization stage. In the current implementation, the visualization thread retrieves frames from device memory, composes the multi-camera image on the CPU, and renders bounding boxes and classification labels using OpenCV.

While this design was acceptable in the desktop version of the system, its impact becomes significantly more pronounced on the embedded platform. The Jetson Xavier NX provides considerably lower CPU performance compared to a desktop workstation, and the repeated device-to-host memory transfers required to retrieve the frames further increase the overhead of the visualization stage.

The measured execution times show that the debug visualization stage requires approximately **10 ms in 20 W mode**, increasing to **around 23 ms in the 10 W configuration**. When combined with the inference cost, this additional processing time contributes noticeably to the total frame latency.

More importantly, the visualization stage executes outside the GPU pipeline and therefore introduces additional synchronization points between CPU and GPU processing. As a result, the debug visualization stage not only consumes CPU resources but also disrupts the otherwise GPU-centric design of the pipeline.

Although the debug visualization is intended primarily as a development and monitoring tool, its current implementation significantly affects the overall system responsiveness on the embedded platform.

A second and more fundamental limitation arises from the cost of the neural network inference stages themselves. As shown in the performance measurements presented earlier, most of the execution time of the pipeline is spent performing inference with the detection and classification models.

Even in the most favourable configuration (20 W mode), the combined execution time of the two networks is approximately **25 ms**, while all other GPU kernels together account for less than **1 ms**. This means that neural network inference alone represents over **97% of the total GPU computation time** of the pipeline.

In the lower power modes, this imbalance becomes even more pronounced. In the **10 W configuration**, the detection network requires nearly **50 ms** per batch, while the classification network adds an additional **17 ms**, resulting in a combined inference cost of approximately **67 ms**.

This observation has important implications for the overall system throughput. Because the current pipeline processes each batch of frames sequentially within the inference thread, the frame rate of the entire system is effectively limited by the time required to complete the two inference stages. Even if all other parts of the pipeline were made arbitrarily fast, the achievable frame rate would remain bounded by the inference latency of the neural networks.

For example, in the 10 W configuration the combined inference time of approximately **70 ms** theoretically limits the system to around **15 frames per second** even before accounting for visualization, synchronization, or memory transfer overheads. This behaviour is consistent with the experimentally observed frame rate below **10 FPS** in that configuration. Performing inference on every captured frame batch is a major limiting factor for achieving the desired real-time responsiveness.

The final factor contributing to the observed performance degradation is the increased workload associated with processing multiple camera streams simultaneously.

Unlike the desktop prototype, which initially operated with a single camera input, the embedded implementation processes **three concurrent video streams**. Although the detection stage was redesigned to support batched multi-camera inference, the presence of multiple cameras still increases the overall computational and memory demands of the system.

Each frame acquisition cycle now produces multiple input frames that must be transferred, pre-processed, and included in the batched inference stage. This increases the amount of data that must be handled by the pre-processing kernels and the memory bandwidth required for frame transfers and intermediate buffers.

Furthermore, the debug visualization stage must retrieve and compose frames from all active cameras before producing the final display image, further increasing the data movement between device and host memory.

While the Jetson Xavier NX GPU is capable of executing the batched inference workload, the additional camera streams introduce extra memory traffic and synchronization overhead that were not present in the original single-camera prototype, contributing to the overall system slowdown observed during the embedded validation phase.

5.5.4 Practical Constraints of Embedded Engine Generation

In addition to the runtime bottlenecks discussed, the deployment process itself reveals a practical limitation related to TensorRT engine generation on the embedded platform.

As discussed earlier in Section 3.5.4, TensorRT engines are tightly coupled to the hardware and software environment in which they are generated. As expected, the engines produced on the development workstation cannot be reused directly on the Jetson device and must therefore be regenerated locally starting from the ONNX model representations.

In practice, this requirement introduces additional constraints when working with embedded systems. Using tools such as *trtexec* requires sufficient system memory and the availability of the required runtime libraries on the target device. In the case of the Jetson platform used in this work, the default software configuration did not include all components necessary to perform the engine generation procedure.

To overcome this limitation, a temporary solution was adopted in which additional system resources were provisioned on the device to install the required tools and libraries needed for engine extraction. Once the engines had been generated successfully, the temporary environment modifications were removed and the system was restored to its original configuration.

Although this procedure is performed only during deployment and therefore does not affect runtime behaviour, it highlights an important practical consideration for embedded GPU inference systems: target feasibility must be evaluated not only in terms of runtime execution, but also in terms of the platform's ability to support the initial engine-generation step.

6. Embedded Pipeline Optimization

The baseline embedded implementation described in Chapter 5 successfully enabled the deployment of the multi-camera detection and classification pipeline on the Jetson Xavier NX platform. However, the performance analysis revealed several limitations affecting the responsiveness of the system.

The primary objective of the next development iteration was therefore to improve the responsiveness of the embedded pipeline. Rather than modifying the neural network models themselves, the optimization effort focused on adapting the behaviour of the processing pipeline to better accommodate the computational constraints of the embedded hardware.

This design choice reflects an important practical consideration. In many real-world deployments, trained models represent significant development effort and cannot be easily replaced or retrained. Consequently, improving system performance often requires architectural optimizations at the pipeline level rather than modifications to the models themselves.

The analysis presented in the previous chapter highlighted two aspects of the pipeline that significantly affected the perceived responsiveness of the system: the debug visualization stage, originally implemented using CPU-side rendering operations, and the strict coupling between camera frame acquisition and detection plus classification stages, which forced the system to progress at the speed of the inference thread.

Addressing these limitations required introducing changes to the execution strategy of the pipeline. The modifications presented in this chapter therefore focus on two main directions: relocating the debug visualization operations to the GPU and introducing a more flexible frame processing strategy that decouples camera frame acquisition from the inference throughput of the neural networks.

The following sections describe this design and the implementation details of these optimizations.

6.1 Optimization Strategy Overview

The limitations observed in the baseline embedded system showed that improving pipeline responsiveness could not be achieved simply by refining the existing implementation at the level of individual auxiliary operations.

Within the scope of this work, modifying the neural network models themselves was not considered an appropriate solution. The trained models represent externally developed components of the system, and their integration is part of the design assumptions of this project. They are therefore treated as fixed elements that must be accommodated by the surrounding pipeline architecture.

For these reasons, the optimization effort focused on modifying the execution strategy of the pipeline rather than attempting to alter the computational cost of the neural networks.

Two complementary design changes were introduced to achieve this objective. The first concerns the debug visualization stage, which was redesigned to operate directly on the GPU to eliminate unnecessary CPU processing and reduce synchronization overhead. The second addresses the strict coupling between camera frame acquisition and inference execution. Instead of requiring every captured batch of frames to be processed by the neural networks, the pipeline was modified to allow inference to operate asynchronously with respect to the camera frame rate.

Together, these modifications improve the responsiveness of the system while allowing the inference stages to operate at their natural throughput on the embedded hardware.

The implementation of these changes is described in the following sections.

6.2 Asynchronous Frame Processing Strategy

The baseline embedded implementation described in Chapter 5 revealed that the main limitation of the system was not the correctness of the multi-camera pipeline, but the fact that frame batches were processed strictly according to the speed of the inference thread. As long as every captured batch had to be processed before the next one could be meaningfully used, the overall behaviour of the system became effectively limited by neural network inference latency.

This issue becomes particularly evident on the embedded platform because the camera subsystem continues to produce new frame batches at a fixed rate of approximately 30 FPS, corresponding to one new batch every 33 ms, while the inference thread requires significantly more time to complete a full detection and classification cycle in the lower power modes. Under these conditions, captured frame batches accumulate faster than they can be consumed, and the system behaviour degrades into a queue of batches processed at the speed of the inference thread.

To remove this bottleneck, the execution strategy of the pipeline was redesigned so that frame capture, inference, and debug visualization no longer operate under a strict one-batch-per-inference policy. Instead, inference is performed whenever the inference thread becomes available, always using the most recent frame batch, while the visualization stage displays the newest available frame batch together with the latest available detection and classification results.

The following sections first explain the design rationale behind this choice and then describe the architectural and thread-level modifications introduced in the implementation.

6.2.1 Design Rationale and Perceptual Assumption

The core idea behind the new strategy is that visual responsiveness does not necessarily require performing inference on every captured frame. What matters from the perspective of the final visual output is that the displayed frame stream remains smooth and that the associated detection results remain visually coherent with the scene being shown.

This observation leads to a perceptual assumption: if the temporal difference between the frame currently displayed and the most recent available detection results is limited to only a small number of frames, then the spatial discrepancy between the object and its bounding box will remain negligible for human perception.

In the present system, the timing measurements discussed in Chapter 5 indicate that this offset remains limited. Since the camera subsystem produces a new batch approximately every 33 ms, while the inference thread may require more than one frame interval to complete a full cycle, the visualization stage may display a newer frame together with slightly older inference results.

Under these conditions, the motion of a detected subject between consecutive frames is typically small enough that reusing slightly older bounding boxes and classification results on a newer frame does not create a visually significant mismatch. The result is that the system can maintain a much more fluid visual output while tolerating a limited temporal offset between the displayed frame and the associated inference result.

This reasoning motivates a change in strategy. Instead of forcing inference to follow the camera rate, the pipeline is allowed to infer whenever the inference thread becomes available, while the debug visualization continuously renders the latest frame batch using the most recent detection structure available at that moment.

6.2.2 Architectural Requirements of the New Strategy

Making this strategy possible required several changes to the way data moves through the pipeline.

The first requirement is that frame batches must no longer be stored in a queue. If the system preserved a queue of captured batches, the inference thread would still be forced to process outdated data in arrival order, reintroducing the same backlog problem identified in the previous chapter. The new strategy therefore requires that only the most recent frame batch be retained at any given time, allowing newly captured frames to overwrite older ones before they are consumed by inference if necessary.

The second requirement is that the inference thread must observe the most recent available frame batch as soon as it becomes ready to start a new inference cycle. Instead of relying on a dedicated pending batch prepared specifically for it, the thread must access a shared latest-frame state and begin processing whatever batch is most recently available when it becomes free.

A third requirement concerns the visualization stage. If the debug thread is expected to maintain smooth visual output, it must be able to access the newest frame batch independently of the inference thread. At the same time, it must also be able to access the latest available detection and classification results without waiting for a new inference cycle to complete. This implies that the debug thread must consume two different forms of state: the most recent captured frames and the most recently published debug-visualization structure.

Finally, these modifications introduce additional synchronization constraints. Because the same shared frame batch and detection structures are accessed by different threads at different times, the implementation must ensure that read and write operations remain properly coordinated to avoid data corruption or inconsistent pipeline state.

These architectural requirements define the thread-level modifications described in the following subsections.

6.2.3 Inference Thread Adaptation

The inference thread was modified to no longer process frame batches in arrival order, but instead it always operates on the newest frame batch available when it becomes ready to start a new inference cycle.

This behaviour is implemented through a shared frame publication mechanism based on a global frame identifier and a condition variable. Each time the capture thread finishes preparing a new frame batch, it increments the global frame identifier and notifies the consumer threads through *frame_ready.notify_all()*. The inference thread blocks on this condition variable and wakes up whenever the published frame identifier differs from the last one it processed.

Once awakened, the inference thread reads the current value of *frame_id* and records it as the most recent frame batch it has consumed. Importantly, this mechanism does not preserve intermediate frame batches. If multiple updates occur while the inference thread is still executing a previous cycle, the thread simply observes the most recent identifier when it becomes available again. In this way, the inference stage always works on the newest available batch rather than accumulating outdated work.

Before beginning pre-processing, the inference thread also synchronizes with the capture pipeline at the GPU level through *cudaStreamWaitEvent(stream1, ev_frame_ready, 0)*, ensuring that the batched frames stored are fully ready for device-side use. After this synchronization, the thread executes the usual sequence of operations for detection and classification stages.

Once the classification stage completes, the resulting detection and action structures must be made available to the visualization stage. Because the debug thread may access these results at any moment, a shared detection state was introduced to store the most recently produced visualization structures.

The inference thread updates this shared state through the *debug_snapshot_publish_async* function. This function performs an asynchronous device-to-host copy of the detection structures produced on the GPU and writes them into a host-side double-buffered memory region reserved for visualization. The use of asynchronous memory transfers allows the inference pipeline to continue executing without blocking the CUDA stream used for classification.

Internally, the function alternates between two host buffers and records a CUDA event associated with the copy operation. Once the transfer is scheduled, atomic variables are updated to publish the index of the buffer containing the most recent results and the number of valid detections. In this way, the

visualization thread can safely access the most recently completed detection snapshot without interfering with the inference pipeline.

If the inference stage produces no detections for a given batch, the same mechanism is used to publish an empty snapshot, ensuring that the visualization stage always observes a consistent system state.

Through this mechanism, the inference thread becomes a producer of detection snapshots while the visualization thread acts as a consumer of the latest available snapshot. This decoupling allows the debug visualization to operate independently of the inference execution rate.

6.2.4 Frame Capture Thread Adaptation

The frame capture thread was modified to behave as a continuous producer of the newest available frame batch, independently of the state of the inference thread.

At each capture cycle, the thread attempts to retrieve one frame from each active camera pipeline using *gst_app_sink_try_pull_sample* with a short timeout. The use of a non-blocking timed pull is important, because it prevents the capture stage from being indefinitely stalled by the temporary absence of a frame on one camera stream. If a frame is not available from a given camera within the timeout window, the thread simply continues, preserving the most recent valid image for that camera.

For each camera that successfully provides a new frame, the capture thread performs the usual NVMM-to-device integration steps and writes the resulting BGR image into the correct slice of the batched GPU buffer. Once all available camera frames have been updated, the thread executes a batched undistortion stage, producing the final *d_bgr_undistorted* buffer used by the rest of the pipeline. This extra manipulation stage was added to the thread to undistort the original images that, with the wide field of view of the cameras, caused detection and classification to lose precision.

At that point, the capture thread records the CUDA event on the capture stream, signalling that the newest batched frame buffer is ready for GPU-side consumption. It then increments the global frame identifier through a dedicated function and notifies the waiting consumer threads using *frame_ready.notify_all*.

The important consequence of this design is that the capture thread never waits for the inference thread to process older frame batches. The latest batch simply overwrites the previously published one, and the rest of the system always works with the most recent available data. In this way, the capture stage continues to operate at the maximum rate supported by the camera subsystem, regardless of the slower throughput of the inference stages.

6.2.5 Debug Visualization Thread Adaptation

The debug visualization thread was modified to follow the same asynchronous principle. Rather than waiting for a new inference result before showing the next frame batch, the thread now renders the newest available frames together with the most recently published detection and classification results.

This behaviour relies on two distinct forms of shared state. The first is the current frame batch already stored on the GPU and updated continuously by the capture thread. The second is the latest available detection structure published asynchronously by the inference thread through *debug_snapshot_publish_async*. Because the detection snapshot remains available until a newer one is produced, the visualization stage can reuse the latest available results even if a new inference cycle has not yet completed.

Consequently, the visualization thread is no longer tied to the completion of a specific inference cycle. It can render the current frame batch immediately, using whatever detection and classification data is currently available. If the inference thread has not yet completed a new cycle, the visualization stage simply reuses the latest published results. This is the mechanism that allows the system to display smooth frame progression even when inference cannot keep pace with the incoming camera rate.

The visualization thread was additionally restructured so that its rendering operations are performed directly on the GPU rather than on the CPU. Since this GPU-side reorganization represents a separate implementation topic, it is described in detail in the following section.

6.3 GPU-Based Debug Visualization

In the earlier version of the pipeline, visualization relied on CPU-side frame composition, resizing, and overlay drawing. While functionally correct, this approach introduced additional CPU workload and required repeated transfers of full image data from device memory to host memory before any rendering operation could be performed.

To reduce this overhead, the visualization stage was redesigned so that the main image-processing operations are executed directly on the GPU. Rather than copying full-resolution frames to the host and composing the visualization there, the new implementation performs frame downscaling, side-by-side packing, and bounding-box rendering on device memory. Only the final reduced debug image is then transferred to the CPU for display or further output handling.

The following sections describe the design principle behind the GPU-based visualization stage and the main operations used to implement it.

6.3.1 Design Principle

The key idea behind the redesigned visualization stage is that the frames required for debugging are already available in GPU memory as part of the normal inference pipeline. Once the capture stage has produced the latest batched frame buffer and the inference thread has published the most recent detection results, the visualization stage does not need to reconstruct this state on the CPU. Instead, it can operate directly on the device-resident data already maintained by the rest of the system.

Rather than transferring full-resolution frames to the CPU and then applying resizing, composition, and overlay operations there, these operations are first executed on the GPU. The visualization thread therefore becomes responsible for orchestrating a sequence of lightweight GPU kernels that prepare a compact debug image directly in device memory.

At a high level, the new visualization pipeline is composed of three stages. First, the most recent batched camera frames are converted into a reduced side-by-side image suitable for display. Second, the most recently published detections are copied from the buffer to the GPU and rendered directly onto this reduced image. Third, only the final small visualization frame is transferred back to host memory for presentation.

This strategy is advantageous for two reasons. The first is computational: resizing and box rendering are highly data-parallel operations and therefore naturally suited to GPU execution. The second is architectural: by delaying the device-to-host transfer until after all visualization work has been completed, the amount of transferred data is reduced from a set of full-resolution multi-camera frames to a single reduced debug image.

The resulting debug stage remains fully compatible with the asynchronous pipeline behaviour described in Section 6.2. The visualization thread still consumes the latest available frame batch together with the latest published detection snapshot. The difference is that the visual composition is now produced almost entirely on the GPU, leaving the CPU responsible only for the final lightweight output step.

6.3.2 GPU Frame Composition and Downscaling

The first step of the visualization pipeline consists of generating a compact side-by-side representation of the frames acquired from all cameras. This representation is used as the base image onto which detection and classification results will later be drawn.

In the original CPU-based implementation, this operation required copying each full-resolution frame from GPU memory to host memory and then performing both resizing and image composition using CPU-side libraries. In the GPU-based redesign, these operations are executed directly on the device using a dedicated CUDA kernel.

The visualization thread invokes a kernel that performs two operations simultaneously: it down-samples each camera frame to a reduced resolution and places the resulting image into the correct horizontal position of a single side-by-side output buffer.

The input of the kernel is the batched frame buffer already maintained by the capture stage, where the frames from the different cameras are stored sequentially in device memory. The kernel uses a three-dimensional execution configuration in which the grid's z dimension corresponds to the camera index. In this way, each camera frame can be processed independently in parallel.

For every pixel of the reduced output image, the kernel computes the corresponding location in the original full-resolution frame using a nearest-neighbour mapping strategy. The selected pixel is then copied into the correct position of the side-by-side output image, whose width corresponds to the reduced frame width multiplied by the number of active cameras. The horizontal offset associated with each camera is computed using the camera index, allowing each frame to occupy a contiguous region of the final visualization image.

By combining downscaling and image composition in a single kernel pass, the implementation avoids multiple intermediate buffers and reduces memory bandwidth usage. The resulting output buffer contains a compact visualization image where all camera streams appear side-by-side at reduced resolution, ready for the overlay stage.

6.3.3 GPU Bounding Box Rendering

Once the reduced side-by-side frame has been produced, the visualization pipeline proceeds to render the detection results directly on this image using a second CUDA kernel.

The detections produced by the inference pipeline are first copied to a small device-side buffer used exclusively for visualization. As discussed in Section 6.2, these detections are published as a snapshot when the inference thread completes processing a new batch of frames. The copy to device memory therefore occurs only when a new snapshot becomes available, rather than at every visualization iteration.

After the detection snapshot has been transferred to device memory, the visualization thread launches the kernel responsible for rendering bounding boxes directly onto the side-by-side visualization buffer generated in the previous stage.

Each CUDA block is responsible for drawing a single detection bounding box. Within the block, the threads cooperate to draw the horizontal and vertical edges of the rectangle by iterating over the pixel coordinates along the box boundaries. This design allows multiple detections to be processed in parallel while keeping the kernel structure simple and predictable.

Before drawing the box, the kernel converts the bounding box coordinates from the original full-resolution frame space to the coordinate system of the reduced side-by-side visualization image. This transformation accounts for both the downscaling factor applied to the frame and the horizontal offset associated with the camera index within the composed visualization image.

The colour used to render each bounding box depends on the classification label associated with the detection.

Performing the overlay stage directly on the GPU eliminates the need to reconstruct bounding boxes on the CPU and avoids additional host-side image processing operations. This further reduces CPU load and keeps the visualization pipeline aligned with the device-centric design adopted for the rest of the system.

6.3.4 Final Host Transfer and Display Integration

After frame composition and bounding-box rendering have been completed on the GPU, the final visualization image is transferred to host memory for presentation.

At this stage the visualization buffer already contains the complete debug image at reduced resolution, meaning that only a single device-to-host transfer is required. The amount of transferred data is therefore significantly smaller than in the previous implementation, where full-resolution frames from multiple cameras had to be copied before visualization operations could be applied.

To further reduce overhead, the host-side buffer used to receive the visualization frame is allocated using pinned memory. This allows the transfer to be performed asynchronously, enabling the CUDA stream responsible for visualization to overlap the copy operation with other potential work.

Once the transfer completes, the resulting image is written into the host-side visualization buffer used by the debug interface. From this point onward the frame can be displayed locally or forwarded to other components of the system responsible for output handling.

By restricting host interaction to this final lightweight transfer step, the redesigned visualization stage minimizes CPU involvement while preserving a flexible interface for displaying or exporting debug information.

6.4 Headless Visualization and UDP Streaming

While the GPU-based visualization stage described in the previous section significantly reduced the CPU overhead of debug rendering, the final system design also considered practical deployment scenarios typical of embedded perception systems. In many real-world applications, embedded devices operate without a local graphical interface and must therefore provide diagnostic or visualization outputs through remote monitoring interfaces.

To support this type of deployment, the visualization subsystem was extended with an additional output mode based on UDP video streaming. Instead of displaying frames locally through an OpenCV window, the visualization thread can optionally transmit the generated debug frames over the network to a remote client. This allows developers to monitor the behaviour of the system in real-time without requiring a graphical environment on the embedded device.

The following subsections describe the design motivation, the streaming architecture, and the integration of this mechanism into the GPU-based visualization pipeline.

6.4.1 Design Motivation and Deployment Considerations

In the earlier development stages of the system, debug visualization was performed locally using OpenCV windows displayed on the host system. While this approach is convenient during development, it becomes less suitable in real embedded deployment scenarios.

Edge devices such as the Jetson Xavier NX are frequently deployed in headless configurations, where no graphical display environment is available. In such cases, relying on local rendering introduces unnecessary dependencies on graphical subsystems and increases the CPU workload of the embedded platform.

A more suitable approach consists of transmitting the debug visualization output to a remote machine where it can be displayed or recorded. This design allows the embedded device to focus on perception tasks while providing developers with real-time feedback during testing and demonstrations.

For this reason, the debug visualization subsystem was extended with a network streaming backend capable of transmitting the generated visualization frames using a lightweight UDP transport mechanism. This solution enables remote monitoring of the perception pipeline while maintaining compatibility with headless deployment environments.

6.4.2 UDP Streaming Architecture

The streaming functionality is implemented using the *GStreamer* multimedia framework. A dedicated component named *GstUdpStreamer* manages the lifecycle of the streaming pipeline and provides the application-side interface through which visualization frames generated by the GPU pipeline are forwarded to the network transmission path.

When streaming is enabled, the debug visualization thread produces visualization frames independently from the rest of the perception pipeline. Each frame is passed to GstUdpStreamer, which reuses a pre-initialized GStreamer pipeline and injects the incoming images into its entry point. The overall streaming architecture is summarized in Figure 13.

Inside the pipeline, the first group of stages is responsible for frame injection and preparation. The appsrc element acts as the application-controlled input of the GStreamer chain, allowing raw visualization frames to be pushed into the pipeline. The subsequent videorate and nvvidconv elements regulate the frame flow and convert the images into a format compatible with the Jetson multimedia subsystem.

The prepared frames are then encoded by the nvv4l2h264enc element, which uses the dedicated H.264 hardware encoder available on the Jetson platform. This stage performs the computationally expensive video compression task without relying on the general-purpose processing resources used by the perception pipeline.

After encoding, the resulting bitstream passes through the final packetization and transmission stages. The h264parse element parses the encoded stream, mpegtsmux encapsulates it into an MPEG transport stream, and udpsink transmits the resulting data over the network to the remote receiver through a UDP socket.

This organization allows the visualization subsystem to generate frames continuously while delegating the streaming operations to a fixed multimedia pipeline managed by GstUdpStreamer. Using the hardware encoder ensures that the additional computational load introduced by streaming remains minimal and does not interfere with the execution of the perception pipeline.

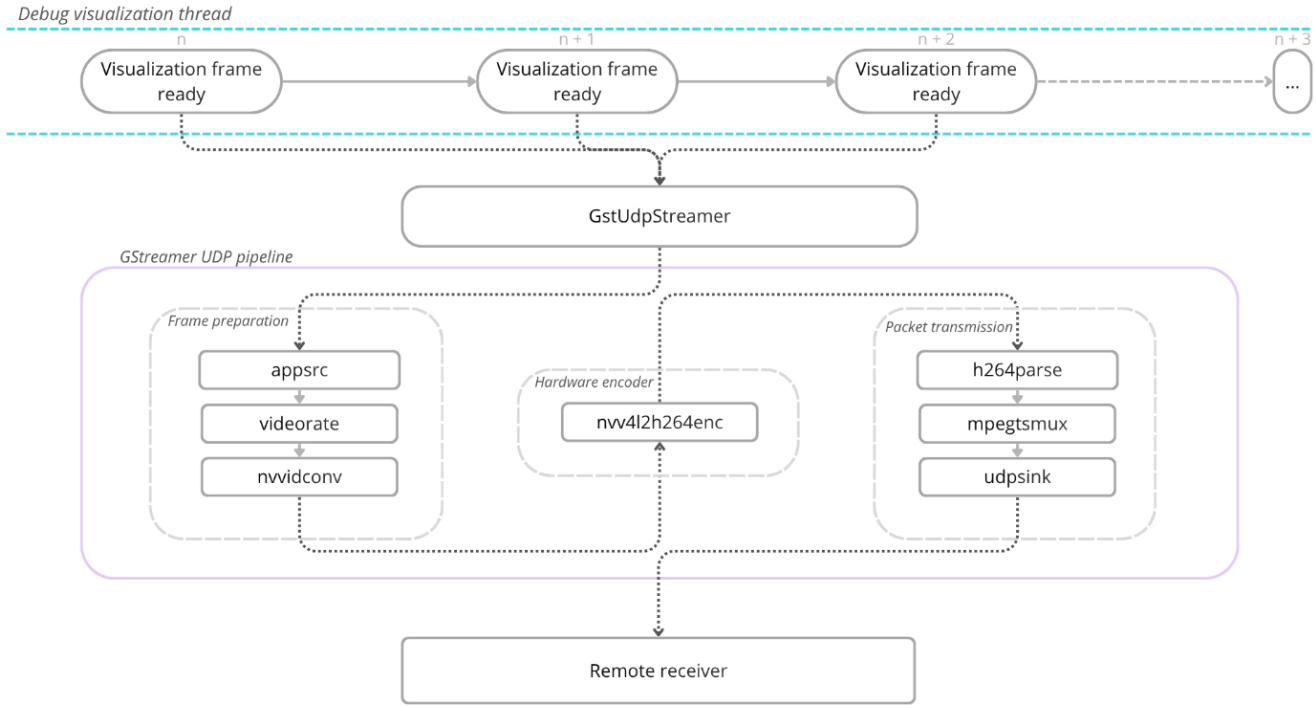

Figure 13 - Data flow conceptual diagram of the UDP streamer pipeline, from producer to receiver.

6.4.3 Integration with the GPU Visualization Pipeline

The UDP streaming functionality is integrated directly into the existing debug visualization thread. At runtime, the thread selects the appropriate visualization backend depending on the compilation configuration. A simple compile-time flag determines whether the visualization output is rendered locally using an OpenCV window or transmitted through the UDP streaming backend.

In both cases, the visualization thread performs the same sequence of operations and executes the GPU-based visualization pipeline described in the previous sections.

When the UDP streaming mode is active, the resulting frame is copied from device memory into a pinned host buffer using an asynchronous CUDA transfer. Once the transfer completes, the frame is forwarded to the GStreamer pipeline through the streaming component, which immediately injects it into the encoder and network transmission stages.

Because the streaming backend only replaces the final visualization output stage, the asynchronous pipeline behaviour introduced in Section 6.2 remains unchanged. The debug visualization thread continues to operate on the most recent frame batch and the latest available detection results, ensuring that the visual feedback remains responsive even when inference latency fluctuates.

7. Final Embedded Validation

7.1 Validation Setup

The purpose of this chapter is to validate the behaviour of the final version of the embedded perception pipeline after the optimizations introduced in Chapter 6. The evaluation was conducted using the same experimental conditions adopted for the baseline embedded implementation described in Chapter 5. Maintaining identical testing conditions allows the results obtained for the final pipeline to be compared directly with the measurements previously collected on the Jetson Xavier NX platform.

All experiments were performed with three cameras active simultaneously, corresponding to the intended operational configuration of the system. This enables an objective comparison between the baseline and optimized versions of the pipeline under realistic working conditions.

The measurements were collected under the three power modes supported by the Jetson Xavier NX platform:

- **10 W mode;**
- **15 W mode;**
- **20 W mode.**

These operating modes impose different limits on the CPU and GPU frequencies available to the system. Evaluating the system across all available power modes makes it possible to observe how the final pipeline responds to different energy-performance configurations.

Execution times were measured using the same profiling methodology adopted in the earlier stages of the project. NVTX instrumentation was used to mark the execution of the main pipeline stages, while CUDA event timing was employed to measure the duration of the corresponding processing steps. Using the same measurement approach ensures that the results obtained in this chapter remain directly comparable with those reported for the baseline embedded system.

A relevant difference with respect to the earlier experiments concerns the debug visualization mechanism. In the final validated configuration, the visualization thread operates in headless mode, meaning that the Jetson device does not display the output locally. Instead, the generated visualization frames are transmitted over the network using the UDP streaming mechanism introduced in Chapter 6.

To visualize the transmitted video stream on the development workstation, the *FFplay* utility was used. *FFplay* is a lightweight media player provided by the *FFmpeg* framework that can receive and decode streaming video over network protocols such as UDP. In the current setup, *FFplay* was used to display the debug visualization stream sent by the Jetson device during testing.

7.2 Quantitative Performance of the Final Pipeline

This section presents the quantitative measurements obtained for the final version of the embedded perception pipeline.

The stages reported in the following tables correspond to the main components of the perception pipeline. Reporting their execution time makes it possible to characterize the performance of the final pipeline and to provide a direct numerical comparison with the baseline Jetson implementation described in Chapter 5.

7.2.1 Pipeline Stage Execution Times

Table 4 reports the median execution time of the main pipeline stages when operating with three cameras active simultaneously.

Table 4 - Final embedded pipeline stages median latencies.

Stage	*10 W*	*15 W*	*20 W*
YOLO Pre-processing	0.039 ms	0.042 ms	0.039 ms
YOLO Inference	48.689 ms	19.344 ms	17.943 ms
YOLO Post-processing	0.249 ms	0.272 ms	0.251 ms
Classifier Pre-processing	0.128 ms	0.129 ms	0.120 ms
Classifier Inference	16.846 ms	6.302 ms	4.413 ms
Classifier Post-processing	0.042 ms	0.045 ms	0.043 ms
Debug Visualization	12 ms	5 ms	5 ms

The results show that the inference stages of the neural networks remain the dominant computational components of the pipeline across all evaluated power modes. In particular, the YOLO detection model accounts for the largest portion of the processing time, followed by the classification network.

7.2.2 Inference Throughput and System Power Consumption

In addition to the execution time of the individual pipeline stages, the overall processing throughput of the inference pipeline can be computed. This value expresses how many complete detection and classification updates can be produced per second by the inference pipeline. It is computed as the inverse of the total inference-thread time:

$$\text{Inference FPS} = \frac{1}{T_{\text{inference}}} \tag{24}$$

where $T_{\text{inference}}$includes the time required to pre-process the frames, execute the detection and classification networks, and perform the corresponding post-processing stages.

During the experiments, the average power consumption of the Jetson device was also monitored.

Table 5 - Overall latency, achievable inference frame rate and average power consumption of the final embedded pipeline

Power Mode	*Inference Time*	*Inference FPS*	*Average Power*
10 W	65 ms	15.15 FPS	11.11 W
15 W	26.13 ms	38.27 FPS	12.32 W
20 W	22.81 ms	43.84 FPS	13.54 W

Since the visualization thread operates asynchronously, the reported throughput represents the computational capacity of the perception pipeline independently from the visualization stage.

As expected, the achievable throughput increases as power modes increase, reflecting the increased GPU and CPU frequencies of the embedded platform. The difference between the 15 W and 20 W configurations is smaller than the gap observed between the 10 W and 15 W modes.

7.3 Qualitative System Behaviour

While the previous section presented quantitative measurements obtained through profiling tools, some aspects of the system behaviour cannot be captured purely through numerical metrics. In particular, the perceived smoothness of the visualization output and the apparent alignment between video frames and detection results are inherently qualitative characteristics.

For this reason, the behaviour of the final system was also evaluated through direct observation during the testing sessions. These qualitative observations provide additional insight into how the pipeline behaves during real-time operation.

7.3.1 Observed Behaviour of the Final Pipeline

During testing, the visualization stream generated by the system showed consistent behaviour across the evaluated power modes. The video output produced through the UDP streaming interface appeared stable and responsive during continuous operation.

In the 15 W and 20 W power modes, the visual output appeared very similar, with only minimal differences observed between the two configurations. In both cases the video stream appeared fluid, with the detection bounding boxes well aligned with the detected subjects during observation.

In the 10 W power mode, the system still maintained stable operation, but the perceived smoothness of the visualization stream was slightly reduced. The video output did not appear fully equivalent to a 30 FPS stream, and occasional instances of slightly misaligned bounding boxes were observed during rapid motion.

These qualitative observations provide an indication of the perceived responsiveness of the system under the different power modes. A detailed analysis of the relationship between these observations and the quantitative performance results will be discussed in Chapter 8.

7.4 Comparison with Baseline Embedded Implementation

The measurements reported in this chapter allow the behaviour of the final optimized pipeline to be compared with the baseline embedded implementation presented in Chapter 5. Since both systems were evaluated under the same experimental conditions and using the same three-camera configuration, the results can be directly compared.

To make this comparison more explicit, additional derived throughput indicators are reported together with the execution times of the main stages. The first is the inference throughput, introduced previously, which represents how many complete semantic updates can be produced per second by the inference pipeline.

The second is the detection-to-visibility throughput, which represents how many times per second a newly computed inference result can become visible in the debug output. This quantity is computed as:

$$\text{Detection-to-Visibility Throughput} = \frac{1}{T_{\text{inference}} + T_{\text{debug}}} \tag{25}$$

where T_{debug} is the time required by the visualization stage to render and publish the corresponding output after the inference result has been produced.

In the final implementation, a third quantity can also be considered, namely the **visualization refresh capability**, defined as:

$$\text{Visualization Refresh Capability} = \frac{1}{T_{\text{debug}}} \tag{26}$$

This value represents the maximum refresh rate that the visualization stage could sustain if considered in isolation. This metric is meaningful only for the final implementation, where the asynchronous organization introduced in Chapter 6 decouples frame display from the production of new semantic results. In the baseline implementation, visualization remained part of the serial pipeline progression, so the practical visible update rate coincided with the detection-to-visibility throughput.

Taken together, these indicators describe different aspects of system behaviour. The inference throughput measures the production rate of semantic information, the detection-to-visibility throughput measures how rapidly a newly produced result can appear in the visible output, and the visualization refresh capability measures the standalone speed of the visualization subsystem in the final asynchronous design.

Table 6 summarizes the key performance indicators measured for the two versions of the Jetson pipeline.

Table 6 - Comparison between original Jetson pipeline and final pipeline components latency.

Stage	*Base 10W*	*Final 10W*	*Base 15W*	*Final 15W*	*Base 20W*	*Final 20W*
YOLO Inference	49.89 ms	48.69 ms	20.94 ms	19.34 ms	18.94 ms	17.94 ms
Classifier Inference	16.85 ms	16.85 ms	7.30 ms	6.30 ms	6.39 ms	4.41 ms
Debug Visualization	23 ms	12 ms	13 ms	5 ms	10 ms	5 ms
Inference FPS	15	15	35	38	40	44
Detect-to-Vis FPS	11	13	24	32	28	36
Visualization Refresh Capability	11	83	24	200	28	200

The comparison shows that the execution time of the neural network inference stages remains very similar between the two Jetson implementations. This is consistent with the adopted optimization strategy, since the neural network models themselves were not modified and therefore continue to define the dominant computational cost of the semantic pipeline.

The most evident numerical variation concerns the debug visualization stage, whose execution time is substantially reduced in all power modes. This improvement directly increases the rate at which a newly computed detection result can become visible, as reflected by the higher detection-to-visibility throughput measured in the final implementation.

In the baseline implementation, the visualization refresh rate effectively coincided with the detection-to-visibility throughput, because the system followed a serial progression in which each visible update depended on the completion of the full inference cycle. In the final implementation, instead, the asynchronous organization separates these two quantities.

The broader implications of this result are analysed in the following chapter.

8. Scalability and System-Level Analysis

This chapter analyses the behaviour of the final embedded perception pipeline based on the measurements presented in Chapter 7. While the previous chapter focused on reporting quantitative and qualitative validation results, the goal of this chapter is to interpret those results and examine the system from a broader perspective.

The analysis explores several aspects of the system behaviour, including the distribution of computational workload across the pipeline, the scalability of the architecture with respect to camera count and detection complexity, and the differences between the embedded deployment and the original desktop implementation.

In addition to performance considerations, the chapter also discusses practical deployment constraints and limitations related to the trained models and available datasets.

8.1 Final Embedded Performance Analysis

The validation results presented in Chapter 7 allow the behaviour of the final embedded perception pipeline to be interpreted in greater detail.

The following sections analyse the main aspects of the system behaviour observed during testing, including the distribution of computational cost within the inference thread, the effect of the asynchronous pipeline design on responsiveness, and the practical implications of the different power modes supported by the Jetson Xavier NX platform.

8.1.1 Distribution of Inference Thread Latency

The validation results presented in Chapter 7 make it possible to examine the final distribution of latency within the inference thread of the embedded perception pipeline. As already observed during the analysis of the baseline embedded implementation, the execution of the neural network models remains the dominant component of the total processing time required to produce a new set of classified detections.

This outcome was expected. The detection and classification models adopted in the system were fixed by the project requirements and were not modified during the optimization process. The architectural interventions introduced in Chapter 6 therefore did not aim to reduce the intrinsic latency of the neural networks themselves, but to reduce the surrounding overheads and improve the behaviour of the pipeline around them. To provide a clearer view of the final computational distribution, Table 7 reports the approximate percentage contribution of the stages belonging to the inference thread across the three Jetson power modes.

Table 7 - Approximative percentages of latency contribution of each inference thread stage.

Component	***10 W***	***15 W***	***20 W***
YOLO inference	~74%	~74%	~79%
Classifier inference	~25%	~24%	~19%
Other stages	~1%	~2%	~2%

Although the absolute execution times vary across the three Jetson power modes, the relative distribution remains highly consistent. In all configurations, the detection network accounts for the largest portion of the inference-thread latency, while the classification network represents the second most significant contribution. The remaining stages, including pre-processing and post-processing, account only for a very small fraction of the total time.

This result is particularly relevant from an engineering perspective. It shows that, once the major architectural inefficiencies were removed, the inference thread became almost entirely bounded by neural network execution. The final latency distribution therefore confirms that the optimized pipeline has reached a structurally efficient organization: the overhead introduced by the surrounding software components is minimal, and the remaining performance bound is primarily determined by the execution speed of the neural network models themselves.

8.1.2 Effect of the Asynchronous Visualization Strategy

A central result of the final embedded implementation concerns the effect of the asynchronous execution strategy introduced in Chapter 6. In the baseline version of the Jetson pipeline, the visualization stage remained tied to the sequential production of new inference results, so the perceived visual feedback was limited by the total time required to complete the full processing cycle, including detection, classification, and debug rendering. This caused the displayed output to inherit the latency of the slowest stages of the pipeline.

In the final version, this behaviour changes. The debug visualization thread no longer waits for a new inference result before updating the displayed frame. Instead, it continuously uses the most recent batch of camera frames available in GPU memory and overlays the most recent published detection and classification results. Therefore, the frequency at which the video stream is visually refreshed is no longer constrained by the inference-thread completion time.

This distinction is important. In the optimized system, two update rates coexist:

- the **frame refresh rate**, determined by how quickly the visualization thread can receive and display the latest captured frames;
- the **inference-result refresh rate**, determined by how quickly the inference thread can produce a new set of detections and classifications.

Because the visualization path was reduced to a much smaller cost than the inference path, the displayed video stream can now be updated at the rate of the arrival frequency of the captured frames, while the bounding boxes and action labels are updated whenever a new inference result becomes available. The practical effect is that the user perceives a fluid video stream even though the semantic annotations may be refreshed at a lower rate.

This behaviour explains the qualitative observations reported in Chapter 7. In the 15 W and 20 W modes, the inference latency is sufficiently low that the annotation updates occur frequently enough to remain visually well aligned with the displayed frames during normal motion. In the 10 W mode, the delay between consecutive inference results becomes more noticeable, and this occasionally appears as slightly stale bounding boxes during faster motion. However, even in this configuration, the asynchronous design prevents the accumulation of a growing backlog. Instead of progressively increasing latency, the system simply skips intermediate frame batches when necessary and resumes processing from the newest available state.

This characteristic is one of the main reasons why the final pipeline provides substantially better responsiveness than the baseline embedded implementation. The optimization does not eliminate the neural-network cost, but it prevents that cost from propagating directly to the perceived smoothness of the visual output.

8.1.3 Bounded Temporal Mismatch Between Frames and Annotations

The asynchronous strategy introduced in the final embedded pipeline improves the perceived smoothness of the output by decoupling frame refresh from inference completion. This design, however, also introduces a deliberate temporal mismatch: the displayed frame may be newer than the detection and classification result drawn on top of it.

In the present system, this mismatch remains bounded by construction. The visualization thread always uses the most recent captured frame batch, while the inference thread always processes the most recent frame batch available when it becomes ready to start a new cycle. Older pending batches are never stored in a queue and therefore cannot accumulate delay over time. If inference temporarily falls behind acquisition, intermediate frame batches are simply discarded and processing resumes from the newest available data.

This property is central to the behaviour of the final system. In a sequential queued design, every delay would be propagated to the next batch, progressively increasing the distance between the displayed

scene and the semantic information associated with it. In the adopted strategy, this cumulative drift is removed. The system may occasionally skip some frame batches, but it does not continue operating on increasingly outdated data. This means that the temporal offset between frames and annotations remains controlled rather than unbounded.

This aspect is important not only in the lowest power mode, but across all operating conditions. The execution times discussed in Chapter 7 are median values, and therefore represent the expected behaviour of the system rather than a strict constant execution time. In practice, individual inference cycles may still take longer than the median because of normal timing variability. For this reason, even in the faster power modes, a design that attempted to process every frame batch in strict order would still risk occasional backlog formation. The adopted strategy avoids this condition entirely: if a slower inference cycle occurs, the system does not accumulate pending work, but simply resumes from the latest available frame batch at the next cycle. In this sense, frame discarding is not only a mechanism to preserve responsiveness at 10 W, but a general safeguard that guarantees stable behaviour in every configuration.

The practical consequence is that the system now exhibits predictable latency characteristics. The refresh rate of the visual stream and the refresh rate of the semantic annotations are no longer rigidly identical, but their relationship remains bounded and stable. Most importantly, the mismatch cannot grow indefinitely with time or with temporary slowdowns. This provides a predictable worst-case behaviour that was not guaranteed in the previous strictly coupled version of the pipeline.

From the application perspective, this bounded mismatch remains visually acceptable because of the spatial scale of the observed targets. The detection box generally encloses the whole body of a person and therefore provides a margin that is significantly wider than the small displacement typically occurring between two consecutive annotation updates. As a result, even when the box is not refreshed on every displayed frame, the subject usually remains visually covered by the previously estimated region. This explains why the final output can appear coherent even though the annotation is not perfectly synchronized with every single displayed frame.

The most critical situations are naturally those involving faster motion or narrower visible body profiles, for example when the subject is observed from the side. In such cases, the effective width of the body inside the image is smaller, and a given physical displacement is more likely to expose portions of the body outside the previous bounding box. Even in these cases, however, the system does not accumulate progressive delay: once a new inference result is available, the semantic overlay is immediately refreshed using the newest available frame batch. The mismatch therefore remains limited in time and does not degrade continuously during operation.

For these reasons, the asynchronous strategy should be interpreted as a controlled compromise. The system accepts a limited and bounded temporal mismatch in exchange for a substantial improvement in responsiveness and stability. Within the operating conditions considered in this work, this trade-off proved to be visually effective and numerically consistent with the measured timings.

8.1.4 Overall Interpretation of the Final Embedded Optimization

Taken together, the results discussed throughout Section 8.1 define the effective outcome of the final embedded optimization. The main result is not that the Jetson platform became able to execute the semantic models substantially faster than before, but that the overall pipeline was reorganized so that the unavoidable cost of inference no longer dictates the behaviour of the whole visible system as rigidly as in the baseline implementation.

After the architectural changes introduced in Chapter 6, the dominant share of the inference-thread latency is still due to the neural networks themselves. This confirms that the optimized system did not remove the computational bottleneck, but rather reduced the surrounding overhead until the remaining limit became almost entirely model-bound. From an engineering perspective, this is an important result: once pre-processing, post-processing, and synchronization no longer contribute significantly to serial latency, the system can be considered structurally efficient with respect to the fixed models it was required to deploy.

The practical consequence of the asynchronous organization is that the pipeline no longer exposes this fixed inference cost in the same way as the baseline implementation. By decoupling visualization from inference completion and by always operating on the most recent available state, the final pipeline avoids progressive backlog growth and preserves a current and visually fluid output even when semantic updates arrive at a lower rate than displayed frames.

This trade-off remains acceptable for the target application scenario. In the higher Jetson power modes, the semantic update frequency is sufficient to preserve good apparent alignment between the displayed frames and the associated annotations, while even in the 10 W configuration the degradation remains bounded and manifests primarily as occasional annotation staleness during faster motion rather than as a collapse of the full pipeline behaviour.

In this sense, the comparison with the baseline implementation presented in Section 7.4 acquires its full meaning. The final version should not be interpreted simply as a pipeline with a faster debug stage, but as a system in which the temporal relationship between semantic computation and visible feedback has been fundamentally restructured. In the baseline design, inference throughput, visible update rate, and practical responsiveness remained tightly coupled. In the final design, these quantities become partially independent: semantic production remains limited by model cost, while visible refresh is allowed to evolve according to the lighter visualization path. This is the key architectural achievement of the final embedded iteration.

The broader implication is that, under fixed-model embedded constraints, meaningful performance gains do not necessarily require reducing neural-network latency itself. They can also arise from reorganizing the pipeline so that the bottleneck is isolated, its effects are bounded, and its impact on the overall user-visible behaviour is reduced. The final implementation validated in this thesis demonstrates exactly this point.

8.2 Detection-Count Scalability

8.2.1 Motivation and Relation to the Previous Redesign

One of the main structural limitations identified in the baseline architecture was the direct dependence of frame latency on the number of detected individuals present in the scene. As discussed in Chapter 4, the original pipeline executed the classification stage sequentially, meaning that each detected person triggered an additional cycle of classification pre-processing, inference and post-processing. The redesign introduced there replaced this behaviour with a batched classification stage in which all detected regions are aggregated into a single input and processed within one inference execution.

The effectiveness of this redesign was first validated in the x86 development environment. The purpose of the present section is to verify that the same scalability property is preserved after deployment on the Jetson Xavier NX, ensuring that the embedded implementation does not reintroduce a dependency on scene complexity.

8.2.2 Embedded Validation Under Varying Detection Count

To evaluate detection-count scalability on the embedded target, the final pipeline was tested under a fixed three-camera configuration while varying the number of people visible in the scene. All measurements reported in this section were collected in 15 W mode under consistent runtime conditions and are intended to highlight the effect of detection count on the classification stage.

In order to isolate the behaviour of the batched classification pipeline, Table 8 reports only the components directly associated with the classification path, including batched pre-processing, inference, and post-processing, together with their combined contribution to the overall classification latency.

Table 8 - Classification pipeline latency under varying detection count, 3 active cameras 15W

Detected people	Cls pre-process [ms]	Cls inf. [ms]	Cls post-process [ms]	Total [ms]
1	0.143	6.36	0.039	6.54
2	0.152	6.90	0.047	7.01
3	0.136	7.33	0.045	7.51
10	0.129	6.40	0.045	6.55
15	0.128	7.06	0.042	7.21

The results show that the latency of the classification stage remains confined within a narrow range despite the increase in the number of detected individuals. The measured values fluctuate slightly across the different test conditions, but no monotonic growth with respect to detection count is observed.

A limited degree of timing variation is still visible across the reported measurements, with differences on the order of a few milliseconds between the tested conditions. However, these fluctuations do not follow the number of detections and therefore cannot be interpreted as a residual detection-scaling effect. Rather, they are more coherently explained as part of the normal runtime variability of the embedded execution environment, which occasionally produces slightly slower inference cycles even under otherwise comparable operating conditions.

This behaviour is fundamentally different from that of the original sequential implementation. In the baseline architecture, each additional detection would have introduced an additional classification inference, resulting in a clear increase in latency as the number of detected persons grew. In the embedded measurements reported here, this dependency is no longer present. The classification stage maintains a nearly constant execution time regardless of whether one or fifteen individuals are detected in the scene.

The absence of detection-dependent latency growth indicates that the classification stage is effectively executed as a batched operation on the embedded platform, preserving the execution model introduced during the redesign phase.

8.2.3 Independence from the Distribution of Detections Across Cameras

The collected measurements did not show any meaningful variation in the latency of the batched classification stage as a function of this distribution. For a fixed total number of detections, the execution time remained substantially unchanged regardless of whether the detected subjects were localized in one camera or distributed across several cameras.

This result further confirms the execution model introduced by the redesign. The classification stage does not operate through separate per-camera inference paths, but on a single aggregated set of detected regions, which is processed as one batched input. As a consequence, the temporal cost of the classification stage is determined by the total batched execution rather than by the origin of the detections within the multi-camera configuration.

This observation reinforces the conclusion of the present section. In the final embedded pipeline, the classification latency is no longer governed by scene complexity in the form of either detection count or spatial distribution across cameras.

8.2.4 Interpretation and Implications

From a system-level perspective, this confirms that the batching strategy introduced in Chapter 4 remains effective after the cross-architecture transition to Jetson Xavier NX. The classification stage preserves its intended execution structure, as scene complexity does not affect latency through the same mechanism that constrained the baseline pipeline. The embedded deployment therefore retains one of the key scalability benefits of the redesign rather than only mimic its functional output.

The classification workload, which previously represented a source of input-dependent latency variability, is now bounded by the cost of a single batched execution. Variations in the number of detected instances therefore affect only the semantic content of the output, without altering the temporal structure of the inference process. The remaining fluctuations observed in the measured timings are not driven by scene complexity, but by the general execution variability of the embedded platform.

This result confirms that the architectural modifications introduced in the earlier stages of the project are not limited to the development environment, but remain valid under the constraints of the embedded platform. The system therefore preserves a stable and predictable classification behaviour even in scenarios characterized by higher scene complexity.

8.3 Camera-Count Scalability

8.3.1 Motivation and Analytical Goal

While the previous section showed that the final embedded implementation preserves the intended scalability with respect to the number of detections, the effect of the number of active camera streams must be analysed separately. This distinction is important because the two forms of scalability refer to fundamentally different aspects of the workload. Detection-count scalability concerns the dependence of the classification stage on scene content, whereas camera-count scalability concerns the amount of concurrent visual data that the overall system must acquire, process and render.

For the target application, the proposed monitoring system is intrinsically multi-camera and the final embedded implementation was explicitly designed to support simultaneous processing of multiple input streams. For this reason, verifying how the latency evolves as the number of active cameras increases is not only a matter of performance characterization, but also of architectural validation.

The analytical goal of this section aims to determine how the main processing stages scale as the number of active camera pipelines increases and interpret the observed trend.

8.3.2 Embedded Measurements Under Increasing Camera Count

To evaluate camera-count scalability, the pipeline was profiled under three operating conditions, with one, two, and three active camera streams respectively. The measurements reported here were collected under the same general runtime configuration and are intended to capture the trend induced by the increase in the number of active pipelines, rather than to restate the absolute final timings of Chapter 7.

For compactness, the lighter pre-processing and post-processing components are aggregated as a single "Other kernels" column, while the heavier stages are reported individually.

Table 9 - System components latency under increasing number of active cameras, 15W mode.

Active cameras	***YOLO inference [ms]***	***ActionCls inference [ms]***	***Other kernels [ms]***	***DebugVisualization [ms]***
1	8.71	3.29	0.56	5.16
2	16.27	4.79	0.61	14.25
3	25.16	6.30	0.92	23.79

To make the trend more explicit,

Table 10 reports the relative scaling factors with respect to the one-camera case.

Table 10 - Latency scaling with respect to the single camera configuration, 15W mode.

Active cameras	***YOLO inference***	***ActionCls inference***	***Other kernels***	***DebugVisualization***
1	1.00×	1.00×	1.00×	1.00×
2	1.87×	1.45×	1.08×	2.76×
3	3.35×	2.52×	1.63×	4.61×

The results show a clear increase in latency as the number of active camera streams grows. The effect is particularly evident for the heaviest stages of the pipeline, YOLO inference, Action Classification inference and Debug Visualization, while the lighter kernels remain comparatively less affected.

A first important observation is that the increase is not uniform across all stages. The heavier components are the most sensitive to the number of active cameras, whereas the aggregated lighter kernels remain limited in absolute value and show a noticeably weaker growth. This point is relevant

because it already suggests that the observed trend is not simply the consequence of an explicit per-camera serial software structure reappearing identically throughout the entire pipeline.

8.3.3 Why the Observed Trend Does Not Indicate Batching Failure

At first sight, the trend reported in Table 9 could suggest that the introduction of multiple camera streams reintroduces a form of seriality similar to the one previously removed from the classification path. However, a closer analysis shows that this interpretation is not consistent with the observed execution behaviour.

Section 8.2 showed that the classification stage does not exhibit latency growth with respect to the number of detections. This confirms that the redesign introduced in Chapter 4 effectively removed the original sequential per-detection loop and that the embedded implementation preserves this property. Moreover, the validation reported in Section 8.2.3 showed that the classification latency is not affected by how detections are distributed across the active cameras. For a fixed total number of detected individuals, concentrating them in a single camera or distributing them across multiple cameras does not produce meaningful timing differences.

These two observations are essential. If the increase in classification latency observed in Table 9 were due to the classifier being executed separately for each camera, then the execution time should depend on how detections are partitioned across cameras. This is not the case. In the implemented pipeline, all detections produced by the YOLO stage are aggregated into a single shared structure and passed to the classifier as one batched input. The classification stage therefore remains structurally batched even when multiple cameras are active.

The same reasoning also helps interpret the behaviour of the YOLO stage. The increase in latency with camera count should not be read as evidence that the detection logic has reverted to a naive per-camera serial implementation. Rather, it indicates that the batched multi-camera workload associated with additional active streams is being executed under progressively less favourable system conditions. In other words, a batched implementation removes explicit software seriality, but it does not guarantee constant latency when the size of the live multi-camera workload itself increases.

To better understand this point, it is useful to recall the producer–consumer structure of the final pipeline described in Chapter 6. The frame capture thread acts as a producer of synchronized multi-camera frames, the inference thread consumes this frame batch and produces detection and classification results, and the debug visualization thread consumes both frames and inference outputs to generate the displayed stream. These components operate asynchronously and are logically decoupled: each stage runs as fast as it can on the newest available data, and when no new data is available it simply remains idle until the next input becomes ready.

Despite this functional decoupling, the latency increase observed when the number of cameras grows appears in all the main stages: frame capture, detection, classification and visualization. Since these stages play different roles in the producer–consumer chain and are not executed as a single serial block,

a common scaling trend across all of them cannot be explained by a local software seriality introduced in a specific stage. Instead, the fact that all independent components exhibit a similar trend strongly suggests the presence of a shared system-level factor affecting the execution time of all major stages.

The data therefore does not show that batching stopped working. It shows that batching alone is not sufficient to maintain near-constant execution time once the full concurrent workload of the multi-camera system becomes too demanding for the available embedded resources.

8.3.4 System-Level Interpretation and Architectural Implications

The most coherent interpretation of the measured trend is therefore a system-level one. As the number of active cameras increases, the Jetson Xavier NX is required to sustain a progressively heavier runtime context, including simultaneous camera acquisition pipelines, GPU-side pre-processing, neural inference, post-processing and debug visualization. Even though these activities are organized through separate execution paths and CUDA streams, they still rely on the same finite hardware resources of the embedded platform.

Under these conditions, increasing the number of active streams does not simply enlarge the semantic input of the pipeline. It also increases the degree of concurrent resource contention under which the main stages execute. In the final architecture, the frame capture thread, the inference thread and the debug visualization thread operate asynchronously and are no longer forced to wait for each other before starting their respective computations. This improves pipeline efficiency and removes unnecessary synchronization, but it also means that the most computationally intensive stages can now be executed at the same time. When multiple heavy operations, such as neural inference and visualization rendering, require GPU resources concurrently, they must share the same finite computational units and memory bandwidth. As a result, some operations are forced to wait for resources to become available, leading to an increase in their effective execution time.

This explains why the growth is most evident in the heavier components: stages such as YOLO inference, Action Classification inference and Debug Visualization occupy hardware resources for longer intervals and are therefore more exposed to contention effects than the lighter auxiliary kernels. The lighter pre-processing and post-processing kernels remain comparatively short and therefore contribute less to the overall scaling behaviour.

This interpretation is also consistent with the broader profiling evidence discussed throughout the chapter. The classification stage remains correctly batched, yet its latency still increases with camera count. The same happens to the detection and visualization stages, even though their internal roles in the producer–consumer structure of the pipeline are different. Such a common trend across distinct components is much more coherently explained by a shared system-level limitation than by independent regressions of the individual software stages.

For this reason, the observed camera-count trend should not be interpreted as a failure of the redesign, but as the manifestation of a practical scalability limit of the target hardware platform. The batching

strategy remains beneficial and necessary: without it, the pipeline would have reintroduced explicit per-camera or per-detection serial execution, resulting in an even less favourable latency profile. What the present measurements show is that, on the Jetson Xavier NX, the available resource headroom is not sufficient to strongly amortize the added workload associated with multiple simultaneous camera pipelines.

The final implication is therefore that the architecture successfully supports the intended multi-camera execution model and preserves its batched logic across deployment. On the other hand, the embedded platform reaches a contention-dominated operating regime in which adding further streams leads to a marked increase in latency, particularly in the heaviest stages. This identifies a concrete system-level limitation of the deployed solution and provides a grounded explanation for the scalability behaviour observed in practice. At the same time, this observation provides useful insight for future developments, indicating that further scalability improvements would require either more powerful hardware resources or additional architectural optimizations aimed at reducing resource contention under multi-stream operation.

8.4 Comparison with Desktop Execution

8.4.1 Comparison Methodology

After analysing the scalability behaviour of the embedded implementation, it is useful to compare the deployed system with the original desktop development platform. This comparison makes it possible to place the embedded results in context and to evaluate the practical trade-off between absolute performance and constrained-power execution.

The desktop and embedded versions of the system share the same logical pipeline, the same neural network models and the same batched execution strategy introduced in the previous chapters. The main difference lies in the hardware environment in which the pipeline is executed. On the desktop side, the system runs on a high-performance discrete GPU platform, the NVIDIA RTX 2080, while on the embedded side it runs on the Jetson Xavier NX under the power modes supported by the device. This allows the comparison to isolate the effect of the target hardware characteristics on the execution behaviour of the pipeline.

For the most direct platform-to-platform comparison, the one-camera configuration represents the most appropriate reference case. In this condition, both systems execute the same processing chain with only one active camera, making the comparison more directly representative of the intrinsic hardware gap between the two platforms. The multi-camera embedded results are then considered separately to show how the Jetson behaviour changes as the number of active streams increases.

All measurements reported in this section are based on median execution times extracted from NVTX profiling. To keep the comparison compact and readable, the reported timings are grouped into four categories: YOLO inference, Action Classification inference, Other kernels and total latency. The Other

kernels category includes the auxiliary pre-processing and post-processing stages, which individually remain small but collectively contribute to the full execution cost of the pipeline.

In addition to latency and throughput, power consumption is also considered in the analysis, since the desktop and embedded platforms are designed for different operating regimes.

8.4.2 Single-Camera Performance Comparison

Table 11 reports the measured performance of the desktop platform and of the Jetson Xavier NX in the one-camera configuration. This represents the most direct comparison between the two systems, since it minimizes the influence of multi-camera contention and allows the observed difference to be attributed more directly to the capabilities of the underlying hardware.

Table 11 - Comparison between the various Jetson modes and Desktop inference latency, one active camera.

Platform	***YOLO inf. [ms]***	***Cls inf. [ms]***	***Other kernels [ms]***	***Total [ms]***	***Working FPS***
Desktop	1.12	0.35	0.12	1.67	628.93
Jetson 10W	14.24	6.93	0.29	21.45	46.61
Jetson 15W	6.14	2.68	0.34	9.16	109.16
Jetson 20W	5.79	1.99	0.33	8.11	123.35

To make the relative difference more explicit,

Table 12 reports the slowdown factors of the Jetson configurations with respect to the desktop platform.

Table 12 - Slowdown factor of the various Jetson modes, compared to the Desktop latency.

Platform	**YOLO factor**	**Cls factor**	**Total factor**
Jetson 10W	12.73×	19.69×	12.84×
Jetson 15W	5.49×	7.61×	5.49×
Jetson 20W	5.18×	5.65×	4.86×

The results show a clear performance gap between the desktop platform and the embedded device, but they also indicate that the gap remains bounded within a range that is meaningful for practical

deployment. In the one-camera configuration, the Jetson Xavier NX in 20 W mode is approximately five times slower than the desktop platform in terms of total latency, while the 15 W mode remains in a similar range. The 10 W mode, as expected, shows a substantially larger difference.

The same trend is visible in the individual stages. YOLO inference, which remains the heaviest component of the pipeline, shows the largest absolute cost on both platforms, but its slowdown on the embedded target is still contained within roughly one order of magnitude. The Action Classification stage shows a stronger penalty in the lowest Jetson power mode, while the gap becomes much smaller in the higher power configurations. The Other kernels remain comparatively limited in absolute value on both platforms, confirming that the main cross-platform difference is driven primarily by the heavier computational stages.

A relevant outcome of this comparison is that the Jetson Xavier NX does not simply behave as an extremely slow version of the desktop platform. In the higher power modes, the embedded device remains capable of executing the full pipeline at well above real-time rates even in the one-camera configuration. This is an important result, because it shows that the final optimized implementation preserves a meaningful portion of the desktop behaviour despite operating under a far tighter hardware and power budget.

8.4.3 Power Consumption and Efficiency Comparison

The comparison presented in the previous section considered only execution time. While latency and throughput are fundamental metrics, they do not fully describe the behaviour of an embedded system, whose design is strongly constrained by power consumption and thermal limits. For this reason, it is necessary to extend the comparison by including power consumption and energy efficiency among the evaluation metrics.

Table 13 - Comparison of total latency and efficiency between Jetson and Desktop, one active camera. reports the semantic pipeline latency, throughput and average power consumption for the desktop platform and the Jetson Xavier NX in the one-camera configuration. From these values, the energy efficiency of the system can be expressed in terms of frames per second per watt.

Table 13 - Comparison of total latency and efficiency between Jetson and Desktop, one active camera.

Platform	*Total [ms]*	*FPS*	*Avg Power [W]*	*FPS/W*
Desktop	1.63	613	200	3.06
Jetson Xavier NX 10W	21.45	46.61	6.47	7.20
Jetson Xavier NX 15W	9.16	109.16	7.77	14.05
Jetson Xavier NX 20W	8.11	123.35	9.10	13.55

From a purely performance-oriented perspective, the desktop platform clearly outperforms the embedded system, achieving a significantly lower latency and a much higher throughput. However, when power consumption is taken into account, the comparison changes considerably. The desktop system operates under a power budget that is approximately one order of magnitude higher than that of the embedded platform, resulting in a much lower performance-per-watt ratio.

The Jetson Xavier NX achieves a substantially higher energy efficiency in all power modes. In particular, the 15 W mode provides the highest efficiency in the one-camera configuration, indicating that this operating point represents the best balance between performance and power consumption for this workload. Increasing the power budget to 20 W further improves raw performance, but the efficiency gain becomes smaller, indicating diminishing returns as the system approaches its hardware limits.

This comparison highlights a fundamental difference between desktop and embedded platforms. Desktop systems are designed to maximize absolute performance, while embedded systems are designed to maximize performance within a constrained power envelope. For real-time applications deployed in power-limited environments, energy efficiency becomes as important as raw performance.

8.4.4 Multi-Camera Workload on Embedded Platform: Performance Degradation and Power Trade-off

The comparisons presented in the previous sections were based on the one-camera configuration, which represents the most favourable operating condition for the embedded platform. In this configuration, the Jetson Xavier NX achieves high energy efficiency and real-time performance within a

relatively small power envelope. However, the target deployment scenario of the system involves multiple simultaneous camera streams, and it is therefore necessary to analyse how the embedded platform behaves as the workload increases.

Table 14 reports the semantic latency, throughput, power consumption and energy efficiency of the Jetson Xavier NX for different numbers of active cameras and power modes.

Table 14 - Performance and efficiency comparison of the various Jetson power modes, under multi-camera workload.

Power mode	***Active cams***	***Total [ms]***	***FPS***	***Power [W]***	***FPS/W***
10W	1	21.45	46.61	6.47	7.20
10W	2	38.07	26.27	10.32	2.54
10W	3	66.00	15.15	11.11	1.36
15W	1	9.16	109.16	7.77	14.05
15W	2	12.87	77.68	11.46	6.78
15W	3	26.13	38.27	12.33	3.10
20W	1	8.11	123.35	9.10	13.55
20W	2	11.31	88.41	12.55	7.04
20W	3	22.80	43.86	13.54	3.24

From these results, a clear trend can be observed. As the number of active cameras increases, the achievable frame rate decreases accordingly. At the same time, the average power consumption of the system increases, but not proportionally to the increase in execution time. As a result, the overall energy efficiency of the system, expressed in frames per second per watt, decreases as the workload increases.

This behaviour represents the practical trade-off of the embedded deployment. In the single-camera configuration, the system operates in a highly efficient regime, achieving a high throughput within a relatively low power envelope. As additional camera streams are activated, the system is required to sustain a heavier concurrent workload, which leads to longer execution times and higher power consumption. The combined effect of these two factors results in a progressive reduction in efficiency.

Another important observation concerns the optimal power mode. In the single-camera configuration, the 15 W mode provides the highest efficiency, while in the two- and three-camera configurations the 20 W mode becomes slightly more efficient due to the higher available compute resources. This indicates that the optimal operating point of the system is not fixed, but depends on the workload intensity and on the number of active camera streams.

These results confirm the interpretation introduced in Section 8.3. The increase in latency observed in multi-camera operation is not caused by a structural limitation of the pipeline, but by the fact that the embedded platform progressively enters a resource contention regime as the workload increases. In this regime, the system consumes more power while delivering less throughput per watt, highlighting the practical scalability limits of the embedded hardware.

8.4.5 Interpretation of the Cross-Platform Trade-off in the Target Application

The comparison presented in the previous sections highlighted the large performance gap between the desktop GPU and the embedded platform in terms of maximum achievable throughput. However, the interpretation of these results must be contextualized within the requirements of the target application considered in this work.

The system developed in this thesis is designed for real-time human detection and action classification in a multi-camera monitoring scenario. In this context, real-time operation does not require extremely high frame rates, but rather a stable throughput in the range of approximately 25–30 frames per second, which is sufficient to ensure temporal continuity and responsiveness in human perception and monitoring tasks.

For this reason, the maximum throughput achievable by the desktop GPU, which can reach several hundreds of frames per second, is not fully exploited in the intended application scenario. When a platform operates far above the required frame rate, a large portion of its computational capacity remains unused, while the power consumption remains close to its nominal operating level. In such conditions, the effective energy efficiency of the system must be evaluated not at maximum throughput, but at the throughput required by the application.

To better illustrate this point, Table 15 reports the effective efficiency of the considered platforms when operating at the frame rates required by the target application. For the desktop platform and for the Jetson operating in 15 W and 20 W modes, the frame rate is capped to 30 FPS, while for the Jetson operating in 10 W mode the achievable frame rate with three cameras is approximately 15 FPS.

Table 15 - Comparison between Jetson modes and Desktop efficiency, under capped application frame rate.

Platform	***Target FPS***	***Power [W]***	***Effective FPS/W***
Desktop RTX 2080	30	200	0.15
Jetson Xavier NX 10W	15	11.1	1.35
Jetson Xavier NX 15W	30	12.3	2.44
Jetson Xavier NX 20W	30	13.5	2.22

From this perspective, the interpretation of the trade-off changes significantly. While the desktop platform achieves much higher absolute performance, its energy efficiency at the operating point required by the application becomes significantly lower than that of the embedded platform. The Jetson Xavier NX, although slower in absolute terms, operates much closer to the required throughput and therefore delivers a substantially higher effective performance per watt in the target deployment scenario.

At the same time, it is important to consider scalability. As shown in Section 8.3, the embedded platform enters a contention-dominated regime as the number of active camera streams increases, which leads to a significant increase in latency. A desktop-class GPU, due to the much larger number of compute units, higher memory bandwidth and higher power budget, can sustain heavier concurrent workloads before reaching a similar contention regime. For this reason, it is reasonable to expect that a desktop platform would scale more favourably in scenarios involving a larger number of simultaneous camera streams or higher resolution inputs.

The comparison therefore highlights a clear trade-off between absolute performance and energy efficiency in the specific context of the developed application. Desktop platforms provide large computational headroom and strong scalability, making them suitable for high-throughput or large-scale multi-stream systems. Embedded platforms, on the other hand, provide sufficient performance for real-time operation while operating within a much lower power envelope, making them particularly suitable for edge deployment scenarios where power consumption, size and thermal constraints are critical factors.

In the context of this work, the Jetson Xavier NX represents a balanced solution that allows real-time multi-camera operation with significantly lower power consumption, while the desktop platform represents the upper bound in terms of achievable performance and scalability.

8.5 Resource Usage and Deployment Trade-offs

The analysis developed in the previous sections focused primarily on execution behaviour, highlighting how the final pipeline scales with respect to detections, camera streams, power modes, and hardware platform. However, a complete evaluation of an embedded AI system cannot rely on latency and throughput alone. Resource usage is equally important, because it determines not only whether the system can be deployed on a given platform, but also how easily the same architecture could be adapted to devices with different memory, storage, or power constraints.

For this reason, the final system must also be analysed from a resource-oriented perspective. The amount of RAM required during execution, the GPU memory occupied by persistent buffers and inference engines, the storage footprint of the deployable package, and the residual headroom left during runtime all contribute to defining the practical class of hardware that can support the pipeline. These aspects are particularly relevant in embedded deployment, where available memory, storage capacity, cooling capability, and power budget are often tightly constrained.

The following sections therefore analyse the resource requirements of the final system, its memory occupancy, the practical meaning of the available power modes, the remaining headroom under concurrent multi-camera operation, and the deployment implications that emerge from these observations.

8.5.1 Resource Requirements of the Final System

Before analysing memory behaviour in detail, it is useful to characterize the final pipeline in terms of the overall resources it requires during execution. This provides an indication of the class of hardware needed to run the system and helps evaluate how easily the same architecture could be deployed on platforms with different memory and storage constraints.

From a main-memory perspective, the final implementation remains within a relatively contained footprint for an embedded multi-camera AI pipeline. Measurements collected during execution showed an overall RAM occupancy of approximately 5 GB in the non-headless configuration and approximately 3 GB in the headless configuration, which represents the reference deployment mode considered in this work. No swap usage was observed during the final tests, indicating that the application remained within physical memory limits and did not enter memory-pressure conditions during operation.

This information is important from a deployment perspective. The observed footprint indicates that an 8 GB memory configuration is sufficient for stable execution of the full pipeline, including multi-camera acquisition, inference and visualization, while still leaving memory available for the operating system and auxiliary services. This suggests that the system does not require high-memory embedded platforms and can operate within the memory range commonly available on mid-range edge devices.

Storage requirements are similarly moderate. The deployable package generated for the final system occupies approximately 250 MB in compressed form and approximately 450 MB once extracted. The serialized TensorRT engines account for approximately 42 MB of this space, while the remaining storage is mainly occupied by shared libraries required by CUDA, TensorRT, OpenCV and GStreamer. The executable itself occupies only a small fraction of the total footprint. This indicates that the deployment size is driven primarily by runtime dependencies rather than by the application code or the neural network models themselves.

On Jetson platforms, CPU and GPU share the same physical memory through a unified memory architecture. As a result, CUDA allocations, TensorRT buffers, multimedia buffers and application memory all contribute to the same system RAM usage. For this reason, GPU memory usage must be interpreted as part of the overall RAM footprint rather than as a separate resource, as would be the case on desktop GPUs with dedicated VRAM.

The explicit CUDA buffers allocated by the application account for approximately 1.07 GB in the three-camera configuration. Further memory is required internally by TensorRT for activation tensors, intermediate feature maps and workspace buffers used by convolution and tensor operations. Additional memory is also used by the multimedia pipeline for camera frame buffers and format

conversion stages. These components together account for a significant portion of the total RAM usage observed during runtime, which is therefore dominated by runtime data rather than by model parameters alone.

Overall, the resource analysis shows that the final system maintains a predictable memory footprint and can operate within the constraints of an embedded platform equipped with approximately 8 GB of shared system memory, while still leaving sufficient headroom for the operating system and auxiliary services.

8.5.2 Memory Occupancy Analysis

To better understand the resource requirements of the final system, a more detailed analysis of memory occupancy was conducted using the headless configuration as reference. This configuration represents the final deployment scenario and shows a total RAM usage of approximately 3 GB during operation, spread across application memory, CUDA allocations, TensorRT buffers and multimedia buffers.

The total memory usage can be conceptually divided into four main components: explicit CUDA buffers allocated by the application, TensorRT runtime memory used for inference, multimedia buffers used by the camera pipelines, and operating system plus application overhead. Among these components, the explicit CUDA buffers account for approximately 1.07 GB in the three-camera configuration. These buffers include frame storage, pre-processing buffers, detection buffers, classifier workspace buffers and visualization buffers, all allocated at initialization and reused throughout execution to ensure maximum runtime speed.

A key characteristic of the implemented allocation strategy is that not all buffers scale with the number of cameras. Buffers associated with frame storage and YOLO detection scale linearly with the number of active cameras, since each camera produces an independent input frame and corresponding detection results. In contrast, classifier-related buffers are allocated according to a fixed maximum batch size representing the worst-case number of detections to be processed in a single iteration. The classifier workspace therefore remains constant regardless of how detections are distributed across cameras, as intended.

This design choice has an important effect on total memory usage. When comparing the explicit CUDA memory allocated in the single-camera and three-camera configurations, the difference is only approximately 25 MB. This relatively small variation indicates that most of the memory footprint is determined by fixed workspace buffers rather than by camera-dependent buffers. As a result, increasing the number of active cameras has only a limited impact on the total memory usage of the system.

Beyond the explicit CUDA allocations, a significant portion of memory is used internally by TensorRT during inference. Although the serialized YOLO and action-classifier engines occupy approximately 42 MB on disk, the runtime memory required during inference is considerably larger. This is because deep neural network inference requires additional memory for intermediate feature maps, activation tensors, convolution workspaces and tensor reformatting buffers. These runtime structures typically occupy

significantly more memory than the network weights themselves and represent one of the main contributors to overall memory usage during execution.

Additional memory is also used by the multimedia subsystem. Each active camera pipeline requires frame buffers for capture, format conversion and synchronization, which are allocated in system memory through the NVMM memory management system. These buffers contribute to the overall RAM usage but do not scale dramatically with the number of cameras due to buffering policies and the use of memory recycling mechanisms in the multimedia pipeline.

Monitoring of system resources also showed moderate utilization of the external memory controller, indicating that the system was not limited by memory bandwidth during operation. This observation is consistent with the hardware characteristics discussed in Chapter 2 and helps close the loop with the measured runtime footprint: although the application allocates a conservative amount of CUDA memory, the observed RAM usage and EMC behaviour indicate that the multi-camera slowdown cannot be explained by memory-capacity or memory-bandwidth saturation.

Overall, the analysis shows that the memory footprint of the system is dominated by runtime data and workspace buffers rather than by model parameters, and that total memory usage remains relatively stable as the number of active cameras increases. This indicates that the performance degradation observed when multiple camera streams are active cannot be attributed to memory capacity limitations and must instead be related to computational load, power constraints and concurrent multimedia processing.

8.5.3 Power Modes and Deployment Operating Points

The results presented in the previous sections showed a clear relationship between power mode, execution latency and overall system throughput, as well as the resulting energy efficiency expressed in frames per second for each watt. These measurements highlighted the trade-off between performance and power consumption that characterizes embedded platforms such as the Jetson device used in this work.

From a deployment perspective, the different power modes can be interpreted as different operating points of the same system, each corresponding to a different balance between latency, throughput and energy consumption. Rather than representing simply different performance levels, these modes allow the system to be tuned according to the requirements of the target application.

If the objective is to minimize latency and maximize throughput, the 20 W mode represents the most suitable configuration, as it provides the shortest inference time and the highest achievable frame rate. This configuration is appropriate in scenarios where maximum responsiveness is required and power consumption is not a primary constraint.

At the opposite end, the 10 W mode minimizes power consumption but significantly increases execution time, especially when multiple cameras are active. This configuration may be suitable in scenarios where energy availability is limited and real-time constraints are less strict.

Between these two extremes, the 15 W mode represents the most balanced operating point for the proposed system. The measurements showed that this configuration allows the system to maintain real-time operation while achieving a significantly better energy efficiency compared to the maximum power configuration. This mode allows the system to sustain the target real-time operating range while keeping power consumption at a lower level, making it the most suitable configuration for continuous operation in a realistic deployment scenario.

This highlights an important aspect of embedded AI system design: the objective is not necessarily to operate the system at maximum performance, but rather to select the operating point that satisfies real-time constraints while minimizing power consumption and thermal stress. In this context, power configuration becomes a deployment parameter that must be selected based on application requirements rather than purely on performance considerations.

8.5.4 Resource Contention and System Saturation

Section 8.3 provided a system-level interpretation of the scalability behaviour observed when increasing the number of active cameras, identifying resource contention on the embedded platform as the most coherent explanation for the measured latency growth. The resource measurements discussed in the present section allow this interpretation to be further examined from a deployment and hardware perspective.

The memory analysis presented in Section 8.5.2 showed that total memory usage remains relatively stable as the number of active cameras increases and that the additional memory required when moving from one to three cameras is limited. This indicates that the observed performance degradation cannot be attributed to insufficient memory capacity. Furthermore, monitoring of the external memory controller showed moderate utilization levels, suggesting that memory bandwidth was not saturated during operation.

At the same time, system monitoring consistently showed high GPU utilization and the presence of power-related limitations when operating in higher power modes under multi-camera load. These observations are consistent with a scenario in which multiple concurrent tasks compete for the same computational resources and power budget, leading to increased execution time due to resource contention rather than to algorithmic inefficiencies or memory limitations.

From a deployment perspective, these results are particularly relevant because they indicate which hardware resources represent the true limiting factors of the system. The analysis shows that the application does not require exceptionally large memory capacity, since memory usage remains within the limits of the platform even under multi-camera operation. Instead, the main constraint is

represented by the available computational throughput and power budget, which determine how well the system can sustain multiple simultaneous processing pipelines.

This observation provides useful guidance for the selection of alternative hardware platforms. For this type of multi-camera real-time AI pipeline, increasing available computational performance or power budget would result in a more significant scalability improvement than increasing memory capacity alone. Additional computational capability directly improves the system's ability to handle multiple concurrent streams.

In this sense, the resource analysis presented in this section supports and reinforces the system-level interpretation discussed in Section 8.3, confirming that the scalability limit observed on the embedded platform is primarily determined by computational resource contention and power constraints rather than by memory-related limitations.

8.6 Model Behaviour and Dataset Limitations

The previous sections focused on the computational and system-level aspects of the proposed pipeline, analysing its latency, scalability, power consumption and resource usage. However, in a real deployment scenario, system performance cannot be evaluated only in terms of execution speed. The reliability of the system also depends on how accurately and consistently the neural network models perform when operating in real-world conditions.

For this reason, it is important to analyse the behaviour of the detection and classification models when deployed in the final system. The objective of this section is not to compare different model architectures or to optimize model accuracy, but rather to evaluate how the selected models behave in the target scenario and to understand how their performance affects the overall reliability of the system.

This analysis is also relevant from a scalability perspective. Understanding the practical behaviour of the models helps determine whether the system could benefit more from larger or more accurate models, or whether the current models already provide sufficient performance for the intended application. In other words, this section helps determine whether some limitations of the system are model-related and therefore if future improvements should focus on model training and dataset quality.

The following subsections therefore analyse the behaviour of the detection model and the classification model separately, highlighting their strengths, limitations and implications for real-world deployment.

8.6.1 Detection Model Behaviour

The person detection stage is based on a YOLO model operating on resized images with a resolution of 256×256 pixels. Despite the relatively small input resolution, the detector showed stable and reliable behaviour during real-world testing. The model was able to consistently detect persons across different scenes, backgrounds and lighting conditions, demonstrating good generalization capability.

The main limitation observed during testing was related to detection accuracy at increasing distances from the camera. Since the input resolution is relatively low, the number of pixels representing a person decreases rapidly as the distance increases, making detection more difficult, especially in the presence of partial occlusions or background colours similar to the subject's clothing.

During testing, the detection behaviour could be qualitatively divided into three distance ranges. In the range between approximately 0 and 3 meters from the camera, detection performance was very reliable. The model was able to detect persons consistently even in the presence of partial occlusions, such as objects positioned between the camera and the subject. Bounding boxes in this range were generally stable and well aligned with the detected person.

In the range between approximately 3 and 5 meters, detection remained generally consistent, although the model could occasionally fail to detect persons that were significantly occluded or partially hidden behind objects. Nevertheless, detection performance in this range was still considered acceptable for the intended application.

Beyond approximately 5 meters, detection performance degraded significantly. Persons that were partially occluded or whose clothing colours were similar to the background were often not detected and bounding box precision decreased. This behaviour is consistent with the reduced spatial resolution available at larger distances when using a 256×256 input image.

From the perspective of the intended application, however, this limitation does not represent a critical issue. The system is designed to monitor the area immediately surrounding a vehicle, where the relevant interaction range is relatively short. A reliable detection range of approximately 5 meters is sufficient for this scenario, as a person located at a greater distance does not likely represent an immediate interaction risk for the monitored vehicle. Furthermore, at such distances, the visual detail available in the image is limited, and even a human observer would have difficulty identifying specific details of the person given the original camera quality.

Overall, the detection model demonstrated sufficient robustness and generalization capability for the intended application and its limitations are primarily related to the input resolution and the physical constraints of the camera setup rather than to model instability. For the considered deployment scenario, the detection stage can therefore be considered reliable within the operational range of interest.

8.6.2 Classification Model Behaviour

The behaviour of the action classification stage required a more careful analysis than the detection stage. During deployment tests, the classifier exhibited unstable predictions in live operation. In particular, the predicted class often changed rapidly over time even when the subject remained approximately still and continued performing the same action. This instability was clearly visible in the live visualization, where the colour associated with the predicted behaviour frequently changed from frame to frame without a corresponding change in the observed human action.

In addition to this temporal instability, the classifier sometimes produced predictions that were not coherent with the observed behaviour. For example, static poses that were clearly represented in the training dataset were occasionally classified as different behaviours, even when the subject remained still and fully visible in the scene. This behaviour suggested that the classifier was not extracting sufficiently robust features to consistently distinguish between the different action classes under real operating conditions.

Since these issues appeared during live deployment, several possible causes had to be considered before attributing the problem to the model itself. In particular, the observed behaviour could potentially be caused by incorrect pre-processing, inference-engine issues, limited model generalization, or insufficient variability and quality of the training dataset.

For this reason, a structured validation process was carried out to isolate the source of the problem and determine whether the observed instability was caused by deployment issues or by intrinsic limitations of the trained model. The results of this investigation are described in the following subsection.

8.6.3 Dataset Limitations and Practical Implications

To identify the cause of the unstable classification behaviour observed during deployment, a series of validation tests were performed to verify the correctness of the pre-processing pipeline, the inference engine and the overall classification path.

The first step consisted in verifying that the pre-processing performed during deployment matched the pre-processing used during model training. As discussed earlier in the thesis, the Edge Impulse exported C++ library was analysed to reconstruct the exact pre-processing sequence. The CUDA pre-processing pipeline implemented in the deployed system was then checked against this reference. The outputs of the deployed pre-processing were compared numerically with the reference tensors produced by Edge Impulse, showing very small numerical differences and visually identical images. This confirmed that the deployment pre-processing pipeline was consistent with the training pre-processing and was not the primary cause of the observed classification errors.

The second step consisted in verifying the correctness of the TensorRT inference engine. To isolate the engine from the pre-processing stage, pre-processed tensors generated directly by Edge Impulse were fed into the TensorRT engine. The classifier produced the expected predictions when using both static and dynamic-batch classifier engines, confirming that the model itself and the inference process were working correctly in this configuration. In particular, the classifier produced consistent results when tested on images from the original dataset.

Once pre-processing and inference-engine correctness were verified, the remaining limitations observed during real-world deployment had to be attributed to the model and to the dataset used during training. A visual inspection of the dataset revealed that it was collected in a very specific environment, with limited variability in terms of subjects, backgrounds, camera viewpoints and lighting conditions. Furthermore, the images used for training were not always perfectly aligned with

the type of bounding boxes produced by the detection stage during deployment, resulting in a distribution mismatch between training data and runtime inputs.

These characteristics suggest that the classifier learned environment-specific visual patterns rather than general behaviour-related features, which explains why it achieved high accuracy on the original dataset but produced unstable predictions when deployed in a different environment. In other words, the model appears to suffer from limited generalization capability, likely due to the limited variability of the training dataset. [59] [60]

It is important to note that, based on the available evidence, it is not possible to determine with certainty whether the main limitation is the model architecture itself or the dataset used during training. It is plausible that a more diverse and representative dataset would significantly improve classifier performance without requiring a more complex model. On the other hand, it is also possible that a more powerful model could extract more robust features even from the existing dataset. Determining which of these two factors is dominant would require additional training experiments that are outside the scope of this work.

From a system design perspective, however, this analysis provides an important insight. The detection stage proved sufficiently reliable for the intended application and represents the most computationally demanding part of the pipeline. The classification stage, on the other hand, is computationally lighter but currently represents the weakest component in terms of reliability. This means that future improvements to the system should primarily focus on improving the action classification stage, either through improved dataset design, improved training procedures or, if necessary, a more robust model architecture. Since the classification stage has a smaller computational cost compared to detection, moderate increases in model complexity could be acceptable from a real-time deployment perspective if they result in significantly more stable and reliable behaviour recognition.

8.7 Comparison with Related Jetson-Based Implementations

The previous sections analysed the proposed system mainly from an internal perspective, discussing its scalability, resource usage, deployment trade-offs, and model behaviour under the conditions considered in this thesis. However, a broader evaluation also requires positioning the obtained results with respect to related embedded implementations reported in the literature.

For this reason, it is useful to introduce a comparison with selected state-of-the-art works addressing real-time visual inference on NVIDIA Jetson platforms. The objective of this section is not to provide an exhaustive survey of all embedded computer vision systems, but rather to identify works that are sufficiently close to the present application to provide a meaningful external reference.

This comparison is relevant to help contextualize the performance achieved by the proposed pipeline with respect to systems operating on similar hardware classes. It also makes it possible to clarify in

which sense the architecture developed in this thesis is similar to, or different from, other Jetson-based real-time implementations already discussed in the literature.

Direct comparison is not entirely straightforward. The systems reported in the literature differ in application domain, camera configuration, network architecture and output objective. Some works focus only on object detection, others combine detection with tracking, and some consider multi-camera acquisition without introducing an additional semantic stage comparable to the action-classification component adopted in this thesis. For this reason, the comparison developed in the following subsections should be interpreted as a structured approximation rather than as a one-to-one benchmark against an identical system.

8.7.1 General Overview of Related Literature

The literature on embedded real-time visual inference on NVIDIA Jetson platforms is broad, but most works focus on a subset of the challenges addressed by the present system. Many published implementations are centred on accelerating a single object-detection network, often from the YOLO family, on an embedded GPU to maximize throughput under constrained power and memory budgets.[61]

This line of work is highly relevant because it provides a useful reference for understanding the practical behaviour of real-time detection models on Jetson-class hardware. In several cases, these studies also consider complete video-based pipelines rather than isolated image inference, meaning that inference is performed while processing a live or recorded stream rather than on already loaded static images. This aspect is particularly important for the present thesis, since system-level behaviour depends not only on network latency, but also on the interaction between inference and the surrounding video-processing pipeline.

A second group of works extends this setting by combining detection with additional stages such as tracking, counting, or event interpretation. These systems are closer to the architecture proposed in this thesis, because they move beyond isolated detection and begin to address multi-stage embedded perception pipelines. However, even in these cases, the semantic structure usually remains simpler than the one considered here, since the additional stages often operate on the detection output through tracking logic rather than through a second neural network dedicated to per-instance semantic classification. [62]

A further subset of the literature addresses multi-camera embedded perception on Jetson platforms.[63] These works are particularly relevant from the perspective of this thesis because they explicitly consider the effect of processing multiple simultaneous visual streams on an embedded device. Nevertheless, the corresponding architectures generally remain centred on a single main recognition stage and therefore do not fully reproduce the combined workload of multi-camera acquisition, object detection, and second-stage activity classification studied in the present work.

Taken together, these observations suggest that the literature contains several systems that are partially comparable to the one developed in this thesis, but that no direct equivalent was identified among the selected peer-reviewed references. The closest works typically match one or two of the following aspects: execution on Jetson hardware, real-time operation on video streams, multi-camera acquisition, or multi-stage perception. However, they do not usually combine all of them within the same embedded pipeline in the specific form considered here.

For this reason, the comparison developed in the next subsection focuses on a small number of representative works chosen for their proximity to the present implementation. The objective is not to claim absolute superiority of the proposed system, but rather to situate its results with respect to the most relevant neighbouring solutions available in the literature.

8.7.2 Quantitative Comparison with Selected Jetson-Based Works

A more concrete comparison can be established by examining a small set of representative Jetson-based implementations and contrasting their main hardware and system characteristics with those of the present work. Since the literature does not report results in a fully standardized way, the comparison cannot be interpreted as a strict benchmark under identical conditions. Nevertheless, it remains useful to position the proposed system with respect to the closest neighbouring implementations identified in the literature.

Table 16 summarizes the selected works in terms of embedded board, available GPU core count when explicitly identifiable, YOLO model, number of cameras, use of a live video stream, power information when reported, and the main performance result emphasized by each work. For the present thesis, the table reports the best measured results obtained in 20 W mode for one, two, and three active cameras.

Table 16 - Comparison between this thesis results and similar state of the art published papers.

Work	***Board***	***# cores***	***YOLO model***	***# cameras***	***Live stream***	***Power***	***Reported performance***
[61]	Jetson AGX Xavier	n.a.	Multiple YOLO-family models; best real-time result with **YOLOR-CSP + DeepSORT**	n.a.	Yes	n.a.	**33.3 FPS** with detection interval = 1; **17 FPS** with detection interval = 0
[62]	Jetson Nano 2 GB	128	**YOLOv5n / YOLOv5s**	1	Yes	n.a.	**YOLOv5n ≈ 15 FPS**, **YOLOv5s ≈ 10 FPS**; reported **mean inference time 112.82 ms/frame** for the selected YOLOv5s configuration

[63]	Jetson Xavier platform	n.a.	**YOLOv3**	4	Yes	n.a.	**15.7 FPS** average processing speed
This thesis	Jetson Xavier NX	384	**YOLOv8s**	1	Yes	**9.10 W** average measured power (20 W mode)	**8.11 ms**, **123.35 FPS**
This thesis	Jetson Xavier NX	384	**YOLOv8s**	2	Yes	**12.55 W** average measured power (20 W mode)	**11.31 ms**, **88.41 FPS**
This thesis	Jetson Xavier NX	384	**YOLOv8s**	3	Yes	**13.54 W** average measured power (20 W mode)	**22.80 ms**, **43.86 FPS**

The comparison in Table 16 highlights two main points. First, the proposed system is positioned within a literature space that already contains embedded YOLO-based real-time pipelines on Jetson hardware, confirming that the present work belongs to an active and relevant line of research. At the same time, the selected references show that the surrounding literature usually focuses on either single-camera detection and tracking on lower-end boards, as in [62], or on multi-camera person recognition with a single main semantic stage, as in [63], or on Jetson-based ADAS-oriented detection and tracking pipelines without the same multi-stage activity-recognition structure considered here, as in [61].

Second, the reported results confirm that direct comparison must be interpreted carefully. The proposed system achieves higher throughput than the selected reference works in the reported one-, two-, and three-camera configurations, even while integrating a multi-camera embedded pipeline with an additional semantic stage beyond detection. However, this should not be read as a claim of universal superiority, because the compared works differ in application domain, model set, tracking logic, camera configuration, and evaluation methodology. In particular, [61] includes detector–tracker combinations designed for ADAS scenarios and reports real-time results within a Deep Stream-based deployment context, [62] is constrained by the much lighter Jetson Nano 2 GB platform, and [63] addresses a four-camera recognition problem with a YOLOv3-based architecture on a Xavier-class device.

For this reason, the most relevant conclusion is not simply that the proposed system reaches high frame rates, but that it does so while combining properties that, in the selected literature, tend to appear only separately. The reviewed works show that Jetson-based real-time detection, tracking, and even multi-camera recognition are individually feasible. The results of this thesis extend that picture by showing that a multi-camera embedded pipeline can also integrate person detection, second-stage activity

classification, and real-time streamed operation within the same system while remaining in a clearly real-time regime under the tested conditions.

9. Conclusions

This thesis addressed the design and implementation of a real-time multi-camera AI-based person activity recognition system on a low-power embedded GPU platform. The work was not limited to the integration of existing neural models into an embedded device, but focused on the broader engineering problem of making the complete perception pipeline operate reliably under strict computational and deployment constraints in an automotive-oriented edge scenario.

Throughout the thesis, the problem was approached at multiple levels. The system was first analysed in its baseline form, then progressively redesigned through batching strategies, kernel adaptations, concurrency-oriented restructuring, cross-architecture deployment work and embedded-specific optimization. The final result was then evaluated not only in terms of latency and throughput, but also with respect to scalability, resource usage, deployment practicality and model behaviour in real conditions.

For this reason, the conclusions of this work do not concern only whether the system runs, but more importantly what was required to make it run, what limits remain on the selected hardware and which directions appear most relevant for future improvements.

9.1 Objectives and Achieved Results

The main objective of this thesis was to determine whether a multi-camera human detection and activity recognition pipeline could be executed in real-time on a low-power embedded GPU platform while preserving stable behaviour under the constraints imposed by embedded deployment.

This question is particularly relevant for automotive-oriented edge systems, where multi-camera perception must be integrated within a variety of operating constraints.

The results obtained throughout the project show that this objective was achieved. The final system was successfully deployed on the NVIDIA Jetson Xavier NX platform and was able to sustain real-time multi-camera operation in the target configuration. The final architecture supports three simultaneous cameras and performs GPU-based person detection and action classification within the power and memory constraints of the target device.

A central result of the work is that real-time behaviour was not achieved through a single isolated optimization, but through the coordinated redesign of the pipeline across several system layers. As discussed throughout the thesis, the baseline implementation already demonstrated functional correctness, but also revealed structural limitations related to sequential execution, per-detection classification cost, synchronization overhead and poor scalability under increasing workload. These issues were progressively addressed through batching, concurrency-oriented restructuring and embedded-specific optimization.

The final performance analysis showed that the pipeline can sustain real-time operation in the intended three-camera scenario, while the broader system-level analysis clarified that the remaining bottlenecks are primarily related to the computational and power limits of the selected embedded platform rather than to unresolved architectural inefficiencies.

Another important outcome of the thesis is methodological. The work demonstrated that deploying deep learning perception pipelines on embedded systems cannot be treated as a model-porting problem. Achieving stable real-time behaviour required coordinated decisions involving batching, memory management, thread-level orchestration, stream-level scheduling and deployment packaging. In this sense, the thesis contributes not only a working prototype, but also a concrete demonstration that architectural design is the determining factor in making embedded multi-camera AI feasible in practice.

The final evaluation also clarified the limits of the obtained solution. The embedded platform proved sufficient to sustain the target application, but it also entered a contention-dominated regime as the workload increased, especially with three active camera streams. Furthermore, while the detection stage showed behaviour consistent with the application requirements, the classification stage revealed limitations primarily related to generalization and dataset quality.

Overall, the thesis demonstrates that real-time multi-camera person activity recognition on a low-power embedded GPU is feasible, but only when the system is designed as a complete embedded architecture compatible with the constraints of practical edge deployment.

9.2 Deployment Considerations, Energy Implications and Practical Lessons

One of the most relevant practical lessons of this work concerns the role of TensorRT engines in the deployment of deep learning models on embedded NVIDIA platforms. TensorRT was not the only possible deployment strategy, since conventional ONNX-based execution would also have been feasible. It was selected here to evaluate how far a hardware-specific optimization path could improve performance in a demanding embedded multi-camera scenario.

From this perspective, TensorRT proved to be a powerful deployment tool. By converting the models into hardware-optimized inference engines, it made it possible to reach very low execution times and exploit the platform more efficiently than a more generic inference path would allow. TensorRT engines therefore represent an effective final acceleration mechanism when the objective is to extract the maximum achievable performance from a specific target platform.

At the same time, the work also showed the limitations of this approach during development. As discussed in the earlier deployment chapters, TensorRT engines are tightly bound to the hardware and software environment in which they are generated. This makes the engine workflow highly effective for final deployment, but less flexible during iterative development and testing. A reasonable practical conclusion emerging from this work is therefore that TensorRT engines are best used as a final-stage optimization tool: development and validation can be carried out using a more flexible inference path,

while engine extraction is introduced once the overall system behaviour has already been stabilized and the objective becomes maximum performance on a specific target.

A second important lesson concerns the interpretation of deployment itself in an automotive-oriented edge context. In this setting, deployment should not be understood only as the transfer of a software pipeline from a development workstation to an embedded target, but also as the integration of that target into a practical on-vehicle computing environment. From this perspective, power consumption is not relevant only as an abstract optimization metric, but as a concrete factor affecting the long-term sustainability of the deployed system, together with thermal behaviour and installation constraints. NVIDIA positions the Jetson Xavier NX precisely as a compact embedded edge module operating in 10 W, 15 W and 20 W power configurations. f

The experiments of this thesis characterized the behaviour of the final system under the available operating modes and measured an average platform power consumption of approximately 11.11 W, 12.32 W and 13.54 W respectively for the considered three-camera configuration. Interpreted in isolation, these values confirm that the platform remains within the expected operating range of a low-power embedded GPU module. However, their practical meaning becomes clearer when they are related to the energy sources typically available in automotive deployment scenarios.

In the case of an electric vehicle traction battery, whose capacity is commonly on the order of several tens of kilowatt-hours, the absolute energy absorbed by the module remains small. For example, considering the 48.4 kWh and 65.4 kWh battery packs offered in the Hyundai Kona Electric range [64], the highest average power measured in this thesis, 13.54 W, would correspond to approximately 0.028% and 0.021% of the total battery capacity per hour of continuous operation, respectively. Even if the module were kept active continuously for 24 hours, the corresponding energy usage would remain below 0.7% of the 48.4 kWh pack and about 0.5% of the 65.4 kWh pack. This suggests that, in a battery-electric vehicle, a perception module of this class is unlikely to be critical when considered only in relation to traction-battery capacity. At the same time, it would still represent a continuously active auxiliary load, whose practical relevance would increase when considered together with the many other electronic subsystems already present on the vehicle.

The situation is different if the same reasoning is applied to a conventional 12 V automotive battery [65]. Taking representative values of 60 Ah and 70 Ah, the nominal stored energy is approximately 720 Wh and 840 Wh, respectively. Under the highest measured average load of 13.54 W, the system would draw about 1.13 A at 12 V. This corresponds to roughly 1.88% of a 60 Ah battery and 1.61% of a 70 Ah battery per hour of continuous operation. In purely theoretical terms, such batteries could sustain the load for about 53 hours and 62 hours, respectively, if fully discharged. However, this is not a realistic operating interpretation: a conventional starter battery is not intended to support prolonged deep discharge at vehicle standstill, because doing so would directly interfere with the energy reserve required for reliable startup and with the normal quiescent loads of the electrical system. For this reason, while the measured module power remains moderate in absolute terms, it should still be considered too high for long-duration always-on operation on a conventional 12 V battery without a more selective duty cycle or an external trigger strategy.

The practical deployment lesson is therefore not simply that the measured power remains “low,” but that its acceptability depends strongly on the operating strategy of the system. Continuous full-rate operation may be compatible with some use cases, whereas prolonged always-on operation in a parked vehicle would require careful consideration of both battery usage and thermal dissipation. Energy consumption, duty cycle and thermal design should therefore be treated as part of the same deployment problem.

A third important lesson concerns the interpretation of scalability on embedded hardware. The analysis carried out in Chapter 8 showed clearly that the proposed system architecture scales correctly at the software level, both with respect to batching and with respect to the intended multi-camera execution model. However, this does not imply that every hardware platform can sustain that scaling equally well in practice. Once the system enters a regime of strong resource contention, the practical scalability of the deployed solution becomes dominated by the characteristics of the platform rather than by the logical structure of the software alone.

This leads to an important conclusion: scalability depends not only on the algorithmic structure of the pipeline, but also on its interaction with the target hardware. The system developed in this work is architecturally scalable, but the Jetson Xavier NX reaches a point where the available computational resources are no longer sufficient to absorb the growth of concurrent workload without significant latency increase.

This observation also suggests a possible direction for hardware-specific adaptation. One way to reduce contention and improve practical scalability would be to reduce the computational pressure generated by the deployed models, for example by selecting lighter engines or less demanding configurations. However, such a choice would introduce an explicit trade-off between resource usage, semantic capability and intrinsic inference latency. The optimal balance would therefore need to be evaluated experimentally on the target hardware rather than assumed a priori.

Another practical aspect concerns the thermal operating conditions of the embedded platform. The experiments of this thesis characterized the behaviour of the system under the available power modes and showed the corresponding trade-off between latency, throughput and energy consumption. However, long-duration thermal validation was not part of the experimental scope of this work. During the performed runs, no critical behaviour was observed, but the board temperature was clearly influenced by workload intensity and selected power mode. For this reason, prolonged operation, especially in the higher power configurations, should be considered with caution in a real deployment scenario. The required cooling and thermal dissipation capability of the final installation should therefore be evaluated together with the selected operating mode. This is particularly relevant if the system is imagined in a continuous on-vehicle monitoring role, where thermal accumulation may become a practical limit even before total battery capacity does.

More generally, this thesis confirmed that the most significant performance gains in embedded AI systems do not necessarily come from replacing the models themselves, but from improving the way information moves through the system and the way resources are coordinated. In this work, the final

improvements were achieved primarily through batching, asynchronous execution, thread-level decoupling and stream-level scheduling rather than through modifications to the neural networks.

Overall, the deployment experience of this work suggests that embedded AI systems should be treated as complete computing systems rather than containers for neural networks. Model execution speed remains important, but the final behaviour of the system depends just as strongly on how data is acquired, transferred, synchronized and consumed under the constraints of the target hardware. In an automotive-oriented edge scenario, the practical success of deployment therefore depends on finding the right balance between model capability, architectural efficiency and hardware-specific limits, while also accounting for energy usage, thermal sustainability and the realistic operating role assigned to the system on the vehicle.

9.3 Possible System Improvements and Future Work

The results obtained in this work show that real-time multi-camera activity recognition on an embedded GPU platform is feasible, but they also highlight several directions in which the system could be improved if the objective were to move from a prototype system to a long-term, real-world deployment. These directions concern not only raw performance, but also the operating strategy of the system in deployment conditions.

A first possible improvement concerns the operating strategy of the system. In the current implementation, the system processes frames continuously at real-time speed, which ensures minimal reaction time but also implies constant power consumption and continuous resource usage. In a practical deployment scenario, however, it may not be necessary to operate continuously at maximum processing speed. A more efficient strategy could be based on multiple operating modes. For example, the system could operate in a low-frequency monitoring mode, processing frames at a reduced rate, and switch to full real-time operation only when a person is first detected in the scene. This approach would reduce average power consumption and thermal load while preserving the ability to react quickly when relevant activity is detected.

A complementary approach could involve an external trigger mechanism. Instead of continuously processing camera frames, the system could remain in an idle state and activate the vision pipeline only when another sensor detects a potential interaction near the vehicle. Such a trigger could be generated by proximity sensors, ultrasonic sensors, radar sensors or other low-power monitoring devices. In this configuration, the embedded GPU system would act as an intelligent perception module that activates when necessary, significantly reducing average power consumption and allowing the system to operate for prolonged periods without thermal or energy-related issues.

In this context, startup time becomes an important parameter. If the neural network engines and GPU buffers are already loaded into memory, the system can begin inference within a very short time after activation. On the other hand, if the engines must be loaded and initialized at activation time, the startup delay would be significantly longer. For this reason, a practical deployment strategy could involve

keeping the system in a low-activity standby mode with engines already loaded in memory, so that the perception pipeline can become operational almost immediately when triggered by an external event and output the first results within milliseconds.

Another important area for improvement concerns the way camera frames reach the processing hardware. In the current implementation, the Jetson board is responsible for handling the camera pipelines and acquiring frames directly. While this approach works, it also means that the embedded platform must continuously manage camera acquisition in addition to running the perception algorithms. In a more advanced system architecture, it would be advantageous if the embedded GPU platform acted primarily as a consumer of already available image frames rather than as the component responsible for acquiring them. For example, frames could be captured and pre-processed by a dedicated camera control unit or image signal processor and then transferred directly to the embedded GPU memory. In such a configuration, the GPU platform could focus primarily on inference and high-level processing, potentially allowing the system to handle a larger number of cameras than in the current architecture.

Hardware evolution also plays an important role in the practical feasibility of systems of this type. The experiments presented in this thesis were conducted on the Jetson Xavier NX platform, which represents a mid-range embedded GPU solution. However, newer embedded platforms, such as the Jetson Orin Nano [66] and related variants, provide significantly higher computational performance while maintaining similar or lower power consumption modes and costs. This means that a system architecture like the one presented in this work could likely support a larger number of cameras, more complex models, or higher input resolutions when deployed on newer embedded GPU platforms without fundamentally changing the software architecture.

More generally, the work presented in this thesis shows that the main architectural principles developed for this system are not strictly tied to a specific device, but rather to a design approach based on batching, asynchronous execution and resource-aware scheduling. For this reason, future improvements should not focus only on individual components of the current implementation, but also on evaluating how the same architectural principles could be deployed on different hardware platforms depending on the performance, power and cost requirements of the final application.

The next section discusses how systems of this type relate to different classes of embedded computing hardware and what trade-offs emerge when moving from GPU-based platforms to other types of accelerators.

9.4 Considerations on Alternative Hardware Platforms

While this work focused on an embedded GPU platform, it is important to note that GPU-based systems are not the only possible solution for embedded AI perception. Different hardware platforms optimize different aspects of the design space, such as computational performance, power efficiency, determinism, flexibility and development complexity. For this reason, the choice of the most

appropriate hardware platform depends strongly on the characteristics of the application and on the structure of the processing pipeline. In an automotive-oriented deployment context, this choice is influenced not only by peak inference capability, but also by how stable the system requirements are expected to remain over time and how much post-deployment flexibility is considered valuable.

In the system presented in this thesis, the workload is not limited to neural network inference alone. The complete pipeline includes image acquisition, geometric transformations, pre-processing, neural inference, post-processing, multi-camera coordination, visualization and streaming. Many of these stages involve custom operations that cannot be easily expressed as part of a single neural network graph. In this context, one of the main advantages of GPU-based embedded platforms is their programmability. A GPU allows both neural inference and custom vision operations to be implemented and optimized on the same device using the same programming model. This makes it possible to design heterogeneous pipelines in which multiple processing stages interact closely and can be optimized together. **NVIDIA** positions Jetson modules precisely as programmable embedded AI platforms spanning multiple power and performance classes, including Xavier NX and newer variants.

This programmability has an important practical implication. During development, validation and early deployment, the perception pipeline may still evolve at multiple levels: models may be retrained, pre-processing may need adjustment, synchronization logic may be refined, and system-level policies may change as testing progresses. In this phase, the ability to update the behaviour of the platform through software remains a major advantage. A programmable embedded GPU is therefore particularly effective when the system is still expected to change, because architectural modifications remain relatively inexpensive compared with more specialized hardware solutions.

Fixed-function AI accelerators and embedded NPUs, such as those integrated in some automotive or industrial system-on-chip platforms, typically offer very high performance per watt for neural network inference.[67] **NXP**, for example, presents its *eIQ Neutron NPU* family [68] as a scalable machine-learning accelerator architecture integrated into embedded processors. These devices are particularly efficient when the application consists mainly of executing one or more neural networks with fixed input sizes and limited post-processing. However, they are generally less flexible when the pipeline includes custom pre-processing, non-standard post-processing, multi-camera synchronization or dynamic execution logic. In such cases, parts of the pipeline often must be executed on the CPU or on separate processing units, which can introduce additional data movement and integration complexity. For applications dominated by a single inference task and strict power constraints, such accelerators may represent a very efficient solution.

FPGA-based [69] platforms represent another possible solution for embedded vision systems. In principle, an FPGA can implement a deeply pipelined vision architecture with very low latency and high energy efficiency, since each processing stage can be mapped directly into dedicated hardware logic. **AMD** presents its *Versal AI Edge* family, including automotive-grade variants, precisely in terms of low-latency edge inference, flexible real-time pre-processing and post-processing and suitability for applications such as automated driving and ADAS. In this sense, FPGA-based or adaptive-SoC solutions

become particularly interesting when the workload is well understood, the latency budget is strict, and the operating energy budget is more critical than development flexibility.

At the same time, these advantages come with important trade-offs. Compared with a programmable GPU, an FPGA-based implementation typically requires significantly longer development time, more complex debugging and validation procedures as well as higher engineering effort whenever the architecture must be revised. For this reason, FPGA-based solutions are often well suited for highly optimized industrial products with stable requirements, but less suited for rapid prototyping, iterative development or research environments in which the pipeline structure may still evolve. In an automotive context, this distinction is especially relevant: if the perception stack is expected to receive frequent refinements or model updates, the flexibility of a software-programmable platform may remain more valuable than the efficiency gains of a more specialized design.

A further step in specialization is represented by **ASIC**-based [70] solutions. An ASIC can push the same logic even further, because the target function is no longer mapped onto reconfigurable hardware but directly embedded into a dedicated chip design. Synopsys describes ASICs as custom-built integrated circuits optimized for predefined functions with maximum efficiency, high performance and low power consumption, while also noting that FPGAs typically have lower non-recurring engineering costs than full-custom ASICs. In practical terms, this means that ASICs can offer the strongest combination of performance, latency and energy efficiency when the application is mature and the target volume is sufficiently high, but they also imply the highest redesign cost, the highest development effort and the lowest post-deployment flexibility.

This point has a direct automotive implication. In a vehicle platform with very stable requirements and sufficiently large production volumes, the high upfront cost of a specialized hardware solution may be amortized across many units. Under those conditions, FPGA-based and especially ASIC-based approaches can become attractive not only because of their efficiency, but also because the cost of specialization is distributed over large-scale production. However, the same specialization makes architectural change significantly more expensive. A programmable GPU platform instead carries a different value proposition: lower specialization, but much higher tolerance to iteration, late-stage adjustment and software-driven evolution of the perception stack. In other words, the most appropriate hardware is not determined only by inference speed, but also by where the product stands along the path from prototype to stable, high-volume deployment.

For completeness, desktop-class GPU platforms represent the reference solution in terms of raw computational performance and development convenience. They provide large computational margins and mature software tools, which makes development and debugging easier. However, their power consumption, size and integration requirements make them unsuitable for any embedded or edge deployment scenarios, as already clear from the very beginning.

From the perspective of this thesis, the most important conclusion is that different hardware platforms are better suited to different workload profiles and different product phases. The choice of hardware platform should not be based only on peak inference performance, but on the overall structure of the

application, including how many processing stages are involved, how often the system architecture is expected to change, how strict the power constraints are, how important deterministic low-latency behaviour is and how much value is assigned to post-deployment update flexibility. The work presented in this thesis shows that, for complex multi-camera AI pipelines that combine neural inference with custom processing and streaming logic, programmable GPU-based embedded platforms represent an effective solution, especially when development flexibility, architectural iteration and software-driven refinement remain central requirements.

9.5 Final Remarks

This thesis investigated the feasibility of executing a real-time multi-camera person detection and activity recognition pipeline on a low-power embedded GPU platform. The results demonstrated that such a system can be implemented in practice and can sustain real-time operation when the pipeline is designed specifically around the constraints of embedded hardware rather than treated as a simplified version of a desktop system. In this sense, the work provides not only a technical validation of the proposed architecture, but also a concrete indication that automotive-oriented edge perception of this type can be made operational on a compact embedded platform.

One of the main conclusions of this work is that the primary difficulty of deploying artificial intelligence on embedded platforms does not lie only in the computational cost of neural network inference, but in the coordination of multiple heterogeneous processing stages that must operate concurrently under limited hardware resources. Achieving real-time performance therefore required system-level architectural decisions rather than simply selecting faster neural network models.

The experiments also showed that system scalability is not determined only by software design, but also by the available hardware resources. The proposed architecture scales correctly from a structural point of view, but the embedded platform eventually enters a resource contention regime when too many concurrent streams are active. This highlights an important practical consideration: scalability in embedded systems must always be evaluated in relation to the hardware on which the system is deployed and not only in terms of software architecture. In an automotive-oriented context, this also means that deployment feasibility depends on how computational capability, energy usage and thermal sustainability interact within the intended operating role of the system on the vehicle.

Another important observation is that once the computational pipeline is able to sustain real-time performance, the main limitation of the system may shift from computational capability to model reliability and dataset quality. In the presented system, the detection stage showed behaviour consistent with the requirements of the application, while the classification stage revealed limitations mainly related to generalization and dataset variability. This does not invalidate the system design objective of the thesis, which was centred on correct and predictable embedded execution, but it indicates a clear direction for future improvement of the semantic component without changing the architectural foundation of the pipeline.

In conclusion, the main contribution of this thesis is the design and implementation of a complete embedded vision architecture capable of sustaining real-time multi-camera AI processing under realistic deployment constraints. More broadly, this work shows that embedded artificial intelligence should be approached as a system-level engineering problem in which neural networks, data movement, concurrency and hardware characteristics must be designed together. The methodology and architectural principles presented in this work can therefore serve as a reference for the design of similar embedded AI systems where real-time performance must be achieved under strict power, memory and hardware limitations, and where the final hardware choice must balance efficiency, flexibility and practical deployment needs.

References

[1] G. Velez and O. Otaegui, "Embedding vision-based advanced driver assistance systems: A survey," *IET Intelligent Transport Systems*, vol. 11, no. 3, pp. 103–112, Apr. 2017, doi: 10.1049/IET-ITS.2016.0026.

[2] C. Häne *et al.*, "3D visual perception for self-driving cars using a multi-camera system: Calibration, mapping, localization, and obstacle detection," *Image Vis. Comput.*, vol. 68, pp. 14–27, Dec. 2017, doi: 10.1016/j.imavis.2017.07.003.

[3] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, doi: 10.1038/NATURE14539;SUBJMETA.

[4] X. Wang *et al.*, "Empowering Edge Intelligence: A Comprehensive Survey on On-Device AI Models," *ACM Comput. Surv.*, vol. 57, no. 9, p. 1, Mar. 2025, doi: 10.1145/3724420.

[5] "Home page en | Department of Electrical, Computer and Biomedical Engineering." Accessed: Apr. 17, 2026. [Online]. Available: https://iii.dip.unipv.it/en

[6] "Nardò Technical Center | Porsche Engineering | Porsche Engineering." Accessed: Apr. 17, 2026. [Online]. Available: https://www.porscheengineering.com/it/nardo/

[7] "Accelerating innovative mobility solutions | Porsche Engineering." Accessed: Apr. 17, 2026. [Online]. Available: https://www.porscheengineering.com/en/peg/

[8] "CUDA C++ Programming Guide Release 13.2 NVIDIA Corporation," 2026.

[9] "Sentry Mode." Accessed: Apr. 16, 2026. [Online]. Available: https://www.tesla.com/ownersmanual/model3/en_us/GUID-56703182-8191-4DAE-AF07-2FDC0EB64663.html

[10] G. Sreenu and M. A. Saleem Durai, "Intelligent video surveillance: a review through deep learning techniques for crowd analysis," *J. Big Data*, vol. 6, no. 1, Dec. 2019, doi: 10.1186/S40537-019-0212-5.

[11] P. Pareek, · Ankit Thakkar, A. Thakkar, P. Pareek, and A. Thakkar, "A survey on video-based Human Action Recognition: recent updates, datasets, challenges, and applications," *Artificial Intelligence Review 2020 54:3*, vol. 54, no. 3, pp. 2259–2322, Sep. 2020, doi: 10.1007/S10462-020-09904-8.

[12] N. O'Mahony *et al.*, "Deep Learning vs. Traditional Computer Vision," *Advances in Intelligent Systems and Computing*, vol. 943, pp. 128–144, 2020, doi: 10.1007/978-3-030-17795-9_10.

[13] Y. Lecun, L. Eon Bottou, Y. Bengio, and P. H. Abstract|, "Gradient-Based Learning Applied to Document Recognition".

[14] F. Sultana, A. Sufian, and P. Dutta, "A review of object detection models based on convolutional neural network," *Advances in Intelligent Systems and Computing*, vol. 1157, pp. 1–16, 2020, doi: 10.1007/978-981-15-4288-6_1.

[15] "Using CUDA Warp-Level Primitives | NVIDIA Technical Blog." Accessed: Apr. 17, 2026. [Online]. Available: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

[16] "CUDA Best Practices Guide — CUDA C++ Best Practices Guide 13.2 documentation." Accessed: Apr. 17, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[17] "Jetson Xavier Series | NVIDIA." Accessed: Apr. 17, 2026. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/

[18] "Leopard Imaging Inc." Accessed: Apr. 17, 2026. [Online]. Available: https://leopardimaging.com/

[19] "LI-XNX-BOX-GMSL2 - Leopard Imaging Inc." Accessed: Apr. 17, 2026. [Online]. Available: https://leopardimaging.com/product/platform-partners/nvidia/xavier-nx-camera-kits/li-xnx-box-gmsl2/li-xnx-box-gmsl2/

[20] "Artificial Intelligence Architecture | NVIDIA Volta." Accessed: Apr. 17, 2026. [Online]. Available: https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/

[21] Jason. Sanders and Edward. Kandrot, “CUDA by example : an introduction to general-purpose GPU programming,” p. 290, 2011, Accessed: Apr. 17, 2026. [Online]. Available: https://books.google.com/books/about/CUDA_by_Example.html?id=49OmnOmTEtQC

[22] “Power Optimization with NVIDIA Jetson | NVIDIA Technical Blog.” Accessed: Apr. 16, 2026. [Online]. Available: https://developer.nvidia.com/blog/power-optimization-with-nvidia-jetson/

[23] “Explore Ultralytics YOLOv8 - Ultralytics YOLO Docs.” Accessed: Apr. 16, 2026. [Online]. Available: https://docs.ultralytics.com/models/yolov8/

[24] T. Y. Lin *et al.*, “Microsoft COCO: Common Objects in Context,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8693 LNCS, no. PART 5, pp. 740–755, May 2014, doi: 10.1007/978-3-319-10602-1_48.

[25] “COCO - Common Objects in Context.” Accessed: Apr. 16, 2026. [Online]. Available: https://cocodataset.org/#home

[26] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma, “A Review of Yolo Algorithm Developments,” *Procedia Comput. Sci.*, vol. 199, pp. 1066–1073, Jan. 2022, doi: 10.1016/J.PROCS.2022.01.135.

[27] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, Jan. 2018, doi: 10.1109/CVPR.2018.00474.

[28] F. Chollet, “Xception: Deep Learning With Depthwise Separable Convolutions,” 2017.

[29] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” 2018.

[30] “LI-IMX390-GMSL2-200H - Leopard Imaging Inc.” Accessed: Apr. 17, 2026. [Online]. Available: https://leopardimaging.com/product/automotive-cameras/cameras-by-interface/adi-gmsl2-cameras/li-imx390-gmsl2/li-imx390-gmsl2-200h/

[31] H. Barnes, “Pro Windows Subsystem for Linux (WSL): Powerful Tools and Practices for Cross-Platform Development and Collaboration,” *Pro Windows Subsystem for Linux (WSL): Powerful Tools and Practices for Cross-Platform Development and Collaboration*, pp. 1–287, Jun. 2021, doi: 10.1007/978-1-4842-6873-5/COVER.

[32] “GeForce GT 1030 | GeForce.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/gt-1030/

[33] “Developer Guide :: NVIDIA Deep Learning TensorRT Documentation.” Accessed: Apr. 16, 2026. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-861/developer-guide/index.html

[34] “ONNX | Home.” Accessed: Apr. 17, 2026. [Online]. Available: https://onnx.ai/

[35] S. Narang *et al.*, “Mixed Precision Training,” *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, Oct. 2017, Accessed: Apr. 16, 2026. [Online]. Available: https://arxiv.org/pdf/1710.03740

[36] “ONNX 1.22.0 documentation.” Accessed: Apr. 16, 2026. [Online]. Available: https://onnx.ai/onnx/

[37] “Command-Line Programs — NVIDIA TensorRT.” Accessed: Apr. 17, 2026. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/latest/reference/command-line-programs.html

[38] “OpenCV - Open Computer Vision Library.” Accessed: Apr. 17, 2026. [Online]. Available: https://opencv.org/

[39] “Part I - Video for Linux API — The Linux Kernel documentation.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.kernel.org/doc/html/v4.9/media/uapi/v4l/v4l2.html

[40] J. Hosang, R. Benenson, and B. Schiele, “Learning Non-Maximum Suppression,” 2017.

[41] H. Rezatofighi, N. Tsoi, J. Gwak, A. Sadeghian, I. Reid, and S. Savarese, “Generalized Intersection Over Union: A Metric and a Loss for Bounding Box Regression,” 2019.

[42] J. S. Bridle, “Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition,” *Neurocomputing*, pp. 227–236, 1990, doi: 10.1007/978-3-642-76153-9_28.

[43] “GNU make.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.gnu.org/software/make/manual/make.html

[44] “CMake Reference Documentation — CMake 4.3.1 Documentation.” Accessed: Apr. 17, 2026. [Online]. Available: https://cmake.org/cmake/help/latest/index.html

[45] Nvidia, “NVIDIA GeForce RTX 2080 User Guide,” 2080.

[46] “Schede grafiche e laptop GeForce RTX Serie 20 | NVIDIA.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.nvidia.com/it-it/geforce/20-series/

[47] S. M. Nabavinejad, M. Ebrahimi, and S. Reda, “Throughput Maximization of DNN Inference: Batching or Multi-Tenancy?,” Aug. 2023, Accessed: Apr. 16, 2026. [Online]. Available: https://arxiv.org/pdf/2308.13803

[48] “NVIDIA Nsight Deep Learning Designer — NsightDeepLearningDesigner.” Accessed: Apr. 17, 2026. [Online]. Available: https://docs.nvidia.com/nsight-dl-designer/

[49] D. Jaggar, “Arm Architecture And Systems,” *IEEE Micro*, vol. 17, no. 04, pp. 9–11, Jul. 1997, doi: 10.1109/MM.1997.612174.

[50] “NVIDIA JetPack SDK Documentation — JetPack.” Accessed: Apr. 17, 2026. [Online]. Available: https://docs.nvidia.com/jetson/jetpack/index.html

[51] “GMSL2 General User Guide”.

[52] NXP BV, “i.MX 8/RT MIPI DSI/CSI-2,” 2024.

[53] “Docker: Accelerated Container Application Development.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.docker.com/

[54] “Docker Docs.” Accessed: Apr. 17, 2026. [Online]. Available: https://docs.docker.com/

[55] “GStreamer.” Accessed: Apr. 17, 2026. [Online]. Available: https://gstreamer.freedesktop.org/documentation/?gi-language=c

[56] “EGL Interoperability”.

[57] “Multimedia APIs — NVIDIA Jetson Linux Developer Guide.” Accessed: Apr. 17, 2026. [Online]. Available: https://docs.nvidia.com/jetson/archives/r36.5/DeveloperGuide/SD/Multimedia/MultimediaApis.html

[58] “NVIDIA DeepStream SDK API Reference: NvBufSurface Struct Reference | NVIDIA Docs.” Accessed: Apr. 17, 2026. [Online]. Available: https://docs.nvidia.com/metropolis/deepstream/dev-guide/sdk-api/structNvBufSurface.html

[59] C. Rohlfs, “Generalization in Neural Networks: A Broad Survey,” *Neurocomputing*, vol. 611, Sep. 2022, doi: 10.1016/j.neucom.2024.128701.

[60] E. B. Joaquin Quiñonero-candela, M. Sugiyama, A. Schwaighofer, and N. D. Lawrence, “DATASET SHIFT IN MACHINE LEARNING”.

[61] P. Azevedo and V. Santos, “Comparative analysis of multiple YOLO-based target detectors and trackers for ADAS in edge devices,” *Rob. Auton. Syst.*, vol. 171, p. 104558, Jan. 2024, doi: 10.1016/J.ROBOT.2023.104558.

[62] H. Gomes, N. Redinha, N. Lavado, and M. Mendes, “Counting People and Bicycles in Real Time Using YOLO on Jetson Nano,” *Energies 2022, Vol. 15, Page 8816*, vol. 15, no. 23, p. 8816, Nov. 2022, doi: 10.3390/EN15238816.

[63] T. H. Jung, B. Cates, I. K. Choi, S. H. Lee, and J. M. Choi, “Multi-Camera-Based Person Recognition System for Autonomous Tractors,” *Designs 2020, Vol. 4, Page 54*, vol. 4, no. 4, p. 54, Dec. 2020, doi: 10.3390/DESIGNS4040054.

[64] “KONA Electric | SUV Elettrico | Hyundai.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.hyundai.com/it/it/models/nuova-kona-electric.html

[65] “Explore Dynamic AGM Batteries | VARTA Automotive Batteries | VARTA Automotive Batteries.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.varta-automotive.com/en/products/automotive/dynamic-agm

[66] “Jetson AGX Orin for Next-Gen Robotics | NVIDIA.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/

[67] L. Bamberg *et al.*, “eIQ Neutron: Redefining Edge-AI Inference with Integrated NPU and Compiler Innovations,” Sep. 2025, Accessed: Apr. 17, 2026. [Online]. Available: https://arxiv.org/pdf/2509.14388v1

[68] “eIQ® Neutron Neural Processing Unit (NPU) | NXP Semiconductors.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.nxp.com/applications/technologies/ai-and-machine-learning/eiq-neutron-npu:EIQ-NEUTRON-NPU

[69] “What is an FPGA (Field-Programmable Gate Array)? – How it Works | Synopsys.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.synopsys.com/glossary/what-is-field-programmable-gate-array.html

[70] “What is ASIC Design? – How it Works | Synopsys.” Accessed: Apr. 17, 2026. [Online]. Available: https://www.synopsys.com/glossary/what-is-asic-design.html

Ringraziamenti

Prof Leporati, Manu, Marc, grazie di essere sopravvissuti alla lettura.