ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

IN

ARCHITECTURES AND PLATFORMS FOR ARTIFICIAL INTELLIGENCE

A Motion Gated Intrusion Detection Pipeline Using NVIDIA DeepStream on Jetson

Supervisor:
Prof. Francesco Conti

Candidate:
Giacomo Caroli

Co-supervisor:
Luca Bompani

Session 5th
Academic Year 2024/2025

Abstract

Edge AI video analytics is an increasingly active research and deployment area, driven by growing demand for privacy-preserving, locally operated surveillance solutions with advanced alerting capabilities. Deploying such systems on embedded hardware introduces non-trivial challenges: cost and power constraints, the need to sustain multiple concurrent streams, and the computational demands of modern detection models must all be addressed locally.

This thesis presents a real-time multi-stream video intrusion detection pipeline deployed entirely on the NVIDIA Jetson Orin Nano Super via the DeepStream SDK, designed to fully exploit the platform's dedicated hardware accelerators. A motion-based pre-filtering stage is proposed based on the MOG2 background subtractor, implemented as a native zero-copy GStreamer plugin via NVIDIA VPI, which selectively suppresses frames carrying no relevant motion before they reach the inference engine.

Compared with a baseline Python implementation, the DeepStream pipeline achieves speedups of up to $4\times$ without MOG2 and $3.3\times$ with it.

This work characterizes YOLOv8 and YOLOv11 across four model sizes, four to ten parallel streams, and three precision modes (FP32, FP16, INT8) on the VIRAT dataset. Evaluation covers throughput, mAP@50, and *Time To First Detection*, a metric introduced in this work to quantify the latency in frames between an object's appearance and its first valid detection.

Results show that INT8 yields a significant speedup at the cost of severe accuracy degradation. FP16, leveraging the Tensor Cores of the Jetson platform, proves simultaneously faster and more accurate than FP32 and is the clear deployment choice. The gated configuration outperforms its ungated variant below 50% motion density and sustains eight parallel streams with YOLO11-M FP16 well above the IEC 62676-4 minimum requirement of 12.5 fps/stream for video surveillance systems, demonstrating that a fully local, privacy-preserving deployment is feasible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

If one were asked to identify the most important among the five human senses, vision would likely be the most widely accepted answer. Vision represents the primary means through which humans perceive space, interpret events occurring in their surroundings, and anticipate future outcomes based on the observed dynamics of the physical world.

When it comes to property security systems, the analogy with human perception still holds. Modern solutions encompass a broad range of sensing devices, including motion detectors, magnetic contact sensors, radar-based instruments, optical barriers, and ultrasonic systems. These technologies are frequently deployed in parallel and, much like the human senses, each contributes partial and heterogeneous information addressing specific aspects of environmental monitoring. Exploiting their complementary strengths is what ultimately enhances the robustness and reliability of the overall system.

Within such multi-sensor security architectures, however, not all signals carry the same semantic weight in the decision-making process. While non-visual sensors are effective at triggering alerts, it is visual feedback that often provides the final layer of validation, enabling contextual interpretation and informed response. Video-based systems can confirm the presence, nature, and dynamics of an event, transforming low-level sensor activations into meaningful situational awareness. For this reason, visual perception has progressively assumed a central role in mod-

ern security solutions, positioning video surveillance not merely as an additional sensor, but as the core element around which high-level decisions are formed.

1.1 Problem statement

Recent years have witnessed remarkable progress on both the hardware and software fronts of video-based sensing [1]. On the hardware side, high-resolution imaging sensors capable of delivering detailed, reliable output even in challenging lighting conditions have become widely accessible, reaching consumer and entry-level product tiers at competitive price points. Object recognition software has followed a parallel trajectory: deep learning models for detection and classification have advanced to a point where, on specific large-scale benchmarks such as ImageNet, their performance is comparable to or even surpasses reported human accuracy under the same evaluation protocol [2]. Integrating state-of-the-art recognition capabilities into consumer products has, however, proven considerably harder. The computational requirements of modern models are substantial, and meeting them within the cost and form factor constraints of a mass-market camera device remains an open challenge [3].

To bridge this gap, many manufacturers have adopted a hybrid model [4]: lightweight processing runs locally on the device, while more demanding analysis is offloaded to cloud servers and delivered back to the user as a subscription service. From a detection performance standpoint, this approach can work well, as cloud datacenters provide computational resources far exceeding what any embedded device can offer. It does, however, carry a set of structural drawbacks that are particularly relevant in a security context. Privacy is an increasingly prominent concern [5]: routing video streams from private premises to external infrastructure means that sensitive footage is processed outside the owner's direct control, and the growing awareness around data sovereignty makes local computation an increasingly valued property. Beyond privacy, subscription fees accumulate significantly over the lifetime of an installation, and the reliance on a remote backend introduces a hard dependency on network availability: a temporary outage or a

spike in latency can directly impair the ability to detect and report an intrusion. Vendor lock-in compounds these issues further: cloud-integrated products typically bind the user to a single ecosystem, limiting interoperability and creating long-term continuity risks, particularly with smaller or emerging manufacturers.

Local processing addresses all of these concerns at once [6]. Performing inference directly on the premises eliminates the exposure of footage to external infrastructure, removes the dependency on network connectivity, and frees the user from subscription costs and vendor commitments. Achieving this, however, is not straightforward. The computational demands of real-time multi-stream inference are non-trivial for a resource-constrained edge device, and managing multiple concurrent video streams efficiently requires intelligent solutions that go beyond brute-force processing. Cost must remain a central consideration at every level: both the initial hardware investment and the ongoing operational expenditure, primarily driven by energy consumption, must be kept within bounds that make the approach genuinely viable as an alternative to the solutions currently available on the market.

1.2 Research Objective

This thesis addresses the design and evaluation of a real-time, multi-stream video intrusion detection system deployed entirely on an embedded edge platform. The target scenario is perimeter protection for residential and small commercial environments, where a set of cameras continuously monitors well-defined areas of private interest. The broader goal is to show that a self-contained, locally deployed solution can represent a practically viable approach to perimeter surveillance, delivering satisfactory detection performance within the cost and power constraints of an embedded platform, without any dependency on external infrastructure.

A key observation about the target scenario shapes the approach taken in this work. Cameras deployed for perimeter surveillance observe largely static scenes for the vast majority of their operational time: a driveway, a fence line, an access point. Events of interest are sporadic, and in their absence the scene carries no in-

formation relevant to the detection task. This characteristic is further reinforced by regulatory guidelines: the European Data Protection Supervisor recommends that surveillance cameras be positioned to minimize the capture of public or shared areas, focusing instead on well-defined regions of private interest [7]. Such constraints naturally result in visually simple, predictable scenes with limited background variability. Rather than processing every incoming frame at full inference cost regardless of content, this work investigates whether a lightweight motion-based pre-filtering stage can be used to focus computational resources where they are actually needed, and to what extent this allows the system to scale to a higher number of concurrent streams, accommodate more capable models, or operate at lower latency, without incurring an unacceptable cost in detection accuracy.

1.3 Contributions

In pursuit of the research objective outlined above, this thesis proposes the following contributions.

A real-time multi-stream video analytics pipeline on embedded edge hardware. A fully functional DeepStream-based pipeline is designed, implemented, and evaluated on the NVIDIA Jetson Orin Nano, supporting N concurrent streams with hardware-accelerated decoding and batched TensorRT inference, instrumented for per-stage latency measurement.

A custom motion gate GStreamer plugin. A novel GStreamer plugin integrating NVIDIA VPI-accelerated MOG2 background subtraction directly into the DeepStream pipeline, operating on NVMM-backed buffers via zero-copy wrapping, with per-stream independent background models and an inline frame drop mechanism.

A comparative baseline analysis. A set of Python-based reference pipelines, from a sequential CPU baseline to progressively GPU-offloaded multi-threaded

variants, is evaluated alongside the DeepStream implementation to quantify framework overhead and identify the conditions under which hardware-native pipelines outperform naive implementations.

A systematic experimental evaluation of motion-gated inference. The impact of the motion gate is characterized across combinations of stream count (4, 6, 8, 10), inference precision (FP32, FP16, INT8), model variant (YOLOv8, YOLOv11), and motion density (5% to 100%), quantifying throughput gains, latency effects, and detection accuracy tradeoffs.

1.4 Thesis Outline

This work is organised as follows. Chapter 2 provides the background on the hardware platform, the software stack, and the detection models used in this work. Chapter 3 describes the design and implementation of the DeepStream pipeline and the motion gate plugin. Chapter 4 details the experimental setup, including the dataset, the baseline pipelines, and the evaluation metrics. Chapter 5 presents and discusses the experimental results. Chapter 6 draws conclusions and outlines future directions. Complete numerical results for all experimental configurations are reported in the Appendix A.

Chapter 2

Background

2.1 Edge AI Platforms

The deployment of neural network inference at the edge has driven the emergence of a distinct category of hardware platforms that integrate general-purpose processing with dedicated AI acceleration on a single module or board. As of late 2025, the accessible end of this market is occupied by a small number of representative options. Table 2.1 summarises the platforms considered during the preparation of this work, characterised along the axes most relevant to the target application: AI performance in TOPS, acquisition cost, cost efficiency expressed as USD per TOPS, and system power draw under sustained inference load.

Table 2.1: Accessible edge AI platforms compared (Q4 2025).

Platform	**AI Perf. (TOPS)**	**Price (USD)**	**USD/TOPS**	**Power (W)**
Google Coral Dev Board (4GB)[†]	4	\$150	37.5	$\sim$4
RPi 5 + AI HAT+ (26 TOPS)[‡]	26	\$190	7.3	$\leq$25
NVIDIA Jetson Orin Nano Super	67	\$249	3.7	7–25

[†] End-of-life; included for reference only.

[‡] Combined price of Raspberry Pi 5 (\$80) and AI HAT+ module (\$110).

Prices are indicative as of Q4 2025 and may have varied over the course of this work.

On the single axis of cost-per-TOPS, the Jetson Orin Nano Super is already the most efficient option in the comparison. Three additional criteria, more di-

rectly tied to the requirements of a multi-stream video analytics workload, further consolidate this choice.

Memory. The Jetson Orin Nano Super provides 8 GB of unified LPDDR5 accessible to the CPU, GPU, and all on-chip accelerators within a single address space. The Hailo-8, mounted on the Raspberry Pi AI HAT+, integrates only on-chip SRAM with no external DRAM, relying on the host Raspberry Pi memory via PCIe for any data that does not fit on-chip, an architecture that introduces transfer overhead and limits the effective memory available for large model weights. The Coral Dev Board, while offering up to 4 GB of LPDDR4, remains constrained relative to the unified memory pool of the Jetson.

Hardware-accelerated video decoding. The Jetson integrates a dedicated hardware decode engine capable of handling multiple concurrent 1080p streams without consuming CPU or GPU resources. The Raspberry Pi 5 offers only partial hardware decode support, with notable codec limitations relative to the target deployment scenario. The Coral platform provides no video decode capability.

Ecosystem maturity. The Jetson platform benefits from a mature ecosystem for video analytics workloads, backed by a production-grade proprietary SDK specifically designed for multi-stream AI pipeline development. No comparable integrated framework exists for the Coral or Hailo ecosystems; the Coral platform, additionally, had reached end-of-life status at the time of this work.

The NVIDIA Jetson Orin Nano Super is therefore selected as the target platform, and its hardware characteristics are described in detail in the following section.

Note: at the conclusion of this work (March 2026), the Raspberry Pi AI HAT+ 2 has become available, featuring the Hailo-10H accelerator at 40 TOPS (INT4) with 8 GB of on-module LPDDR4. This platform was not available during the development of this work and no comparative evaluation was conducted, but it represents a potentially relevant alternative that warrants future investigation.

2.2 The NVIDIA Jetson Orin Nano

The NVIDIA Jetson Orin Nano [8] is a compact system-on-module (SOM) designed for edge AI applications. It belongs to the Jetson Orin family, NVIDIA's current generation of embedded AI platforms built around the Ampere GPU architecture.

The module used in this work is the 8GB variant of the Jetson Orin Nano Super, a drop-in performance upgrade over the original Jetson Orin Nano that delivers 67 TOPS of AI performance against 40 TOPS of its predecessor. This improvement is achieved through firmware and software optimisations on the same hardware platform, within the same form factor, memory configuration, and power envelope, at a list price of $249. This combination of cost, power efficiency, and retained hardware acceleration capability makes it a compelling entry point into the Orin family for the class of applications addressed in this work. Its main technical specifications are summarised in Table 2.2.

Throughout this thesis, the Jetson Orin Nano and Jetson Orin Nano Super designations are used interchangeably; all measurements and results presented refer to the Super variant.

Table 2.2: NVIDIA Jetson Orin Nano Super 8GB – Technical Specifications

Component	**Specification**
AI Performance	67 TOPS (INT8)
GPU	NVIDIA Ampere, 1024 CUDA cores, 32 Tensor Cores
CPU	6-core ARM Cortex-A78AE v8.2 64-bit, 1.5MB L2 + 4MB L3
Memory	8GB 128-bit LPDDR5, 102 GB/s
Video Decode	Up to 11x 1080p30 or 5x 1080p60 (H.265)
Power Envelope	7W – 25W
Form Factor	$103 \times 90.5 \times 34.77$ mm

The module runs on NVIDIA JetPack SDK [9] version `6.2`, which represents the latest release compatible with the Jetson Orin Nano at the time of this work, and provides the complete software stack including CUDA [10], cuDNN [11], TensorRT [12], VPI [13], and DeepStream [14]. The DeepStream SDK version

7.1 and VPI version 3.2, both included in the JetPack distribution, constitute the primary development frameworks adopted in this thesis. This specific combination was selected on the basis of hardware compatibility and the availability of reference documentation and examples relevant to the development carried out.

2.2.1 Hardware Acceleration Engines

Beyond the general-purpose CPU and GPU, the Jetson Orin Nano integrates a set of dedicated hardware engines that are central to the design of efficient video analytics pipelines, as illustrated in Figure 2.1.

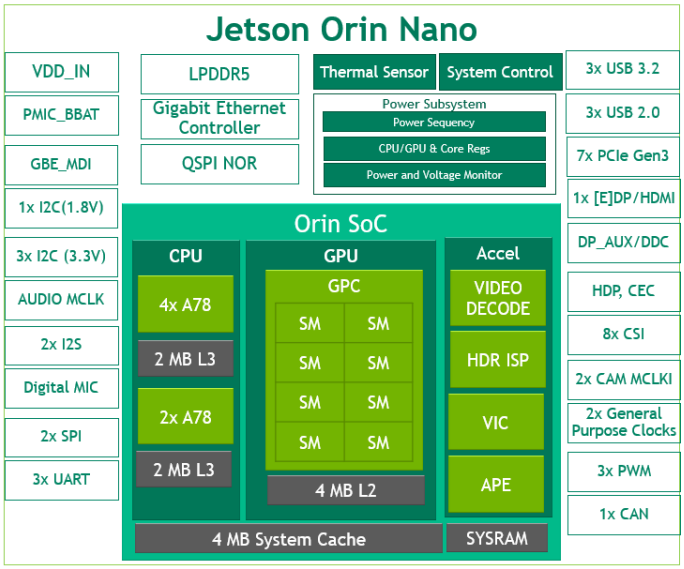

Figure 2.1: Block diagram of Jetson Orin Nano.
Image courtesy of NVIDIA Corporation, developer.nvidia.com.

NVDEC. The NVIDIA Video Decoder (NVDEC) is a fixed-function engine capable of decoding compressed video streams without consuming GPU or CPU resources. On the Orin Nano it supports up to 1x 4K60, 2x 4K30, 5x 1080p60, or 11x 1080p30 concurrent H.265 streams. Decoded frames are written directly into NVIDIA Memory Management (NVMM)-backed surfaces, making them immediately available to downstream GPU processing without any intermediate copy.

In a multi-stream pipeline, offloading decoding to NVDEC is essential to preserve the GPU compute budget for inference.

VIC. The Video Image Compositor (VIC) is a hardware engine dedicated to 2D image operations such as format conversion, scaling, and compositing. In a DeepStream pipeline, VIC handles color space conversions and resolution scaling between pipeline stages without CPU or GPU involvement, resolving format negotiation between elements at negligible computational cost.

NVMM and the unified memory architecture. A defining characteristic of the Jetson platform is its unified memory architecture: CPU and GPU share the same physical LPDDR5 memory pool, interconnected through a high-bandwidth fabric. NVIDIA Memory Management (NVMM) is the buffer management layer built on top of this architecture. NVMM surfaces are allocated once and can be accessed by any hardware engine (CPU, GPU, NVDEC, VIC) without copying data between separate memory pools. A video frame decoded by NVDEC, scaled by VIC, analyzed by VPI, and consumed by the GPU inference engine never leaves device memory, as long as all pipeline stages operate within the NVMM domain. This zero-copy property is the foundation on which efficient video analytics pipelines on Jetson are built, and a prerequisite for achieving the throughput figures that make multi-stream real-time inference feasible on hardware of this class.

2.3 NVIDIA Vision Programming Interface (VPI)

NVIDIA VPI is a software library that provides a unified API for computer vision and image processing algorithms across the heterogeneous hardware backends available on NVIDIA embedded and discrete platforms. Its primary purpose is to abstract the complexity of accessing dedicated hardware accelerators, such as VIC, PVA, and OFA, which would otherwise require low-level, device-specific APIs, while also providing a consistent interface to more general-purpose back-

ends such as CUDA and CPU. An architectural overview of VPI is reported in Figure 2.2.

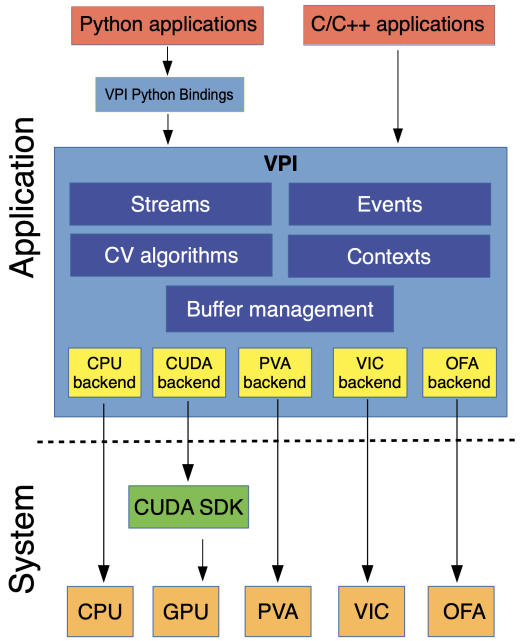

Figure 2.2: VPI architecture diagram.
Image courtesy of NVIDIA Corporation,
`https://docs.nvidia.com/vpi/architecture.html`.

A central concept in VPI is the *stream*, an ordered queue of operations submitted for asynchronous execution on a given backend. Operations are enqueued on a stream via submission calls and executed in order, potentially overlapping with work on other streams or on the host. Explicit synchronization points allow the application to wait for completion before consuming results. This model enables efficient pipelining of heterogeneous workloads: for instance, the GPU can perform inference on one frame while VIC preprocesses the next, and the CPU handles metadata and control logic, all concurrently.

VPI implements a variety of computer vision algorithms, including background subtraction, optical flow, stereo disparity, and image statistics among others, each available on one or more of the supported backends. The same algorithm can be executed on CPU, CUDA, VIC, or PVA, where an implementation exists

for that backend, by selecting the appropriate flag at submission time, without changing the surrounding application code. This flexibility allows the processing pipeline to be configured to exploit the full installed computing capacity of the target device, and makes it straightforward to benchmark or switch between backends during development.

A key property of VPI on Jetson platforms is zero-copy memory interoperability. VPI images can be created as lightweight wrappers around existing NVMM-backed buffers via DMA-BUF file descriptors, allowing frames already resident in device memory to be fed directly into VPI algorithms without any data copy. The version of VPI available on the Jetson Orin Nano at the time of this work is 3.2, which supports the CUDA and VIC backends relevant to this thesis.

2.4 GStreamer and NVIDIA DeepStream

2.4.1 GStreamer

GStreamer [15] is an open-source multimedia framework for constructing processing pipelines by linking modular, reusable components. Originally developed for audio and video playback, it has evolved into a general-purpose dataflow framework widely adopted in embedded systems, broadcast infrastructure, and, more recently, AI-driven video analytics. Its design is built around a small set of abstractions that, combined, allow arbitrarily complex processing graphs to be assembled from independent, interchangeable building blocks.

Elements. The fundamental unit of processing in GStreamer is the *element*. Each element encapsulates a single, well-defined processing function: reading data from a file, decoding a compressed video stream, converting a pixel format, or writing output to a display. Elements are implemented as shared libraries and registered with the GStreamer plugin system, making them discoverable and instantiable at runtime without recompiling the application. Three categories of elements exist: *source* elements, which produce data and expose only output pads; *sink* elements, which consume data and expose only input pads; and *filter* ele-

ments, which transform data flowing through them and expose both input and output pads.

Pads, caps negotiation, and dynamic linking. Elements communicate through *pads*, which are the typed connection points through which data enters and leaves an element. A *source pad* produces data; a *sink pad* consumes it. Each pad is associated with a set of *capabilities* (caps), which describe the media types and formats the pad can handle. Before data begins to flow, GStreamer performs *caps negotiation*: adjacent pads exchange their capability sets and agree on a common format, ensuring type safety at the data flow level.

Not all pads are available at pipeline construction time. Some elements expose *dynamic pads*, created only at runtime once the format of the incoming data is known. A notable example is `qtdemux`, which cannot expose its output pads until it has parsed the container header and identified the elementary streams it contains. GStreamer handles this through the `pad-added` signal: the application registers a callback that performs the downstream link dynamically at the moment the pad becomes available. This mechanism is essential for handling real-world media containers whose internal structure is not known a priori.

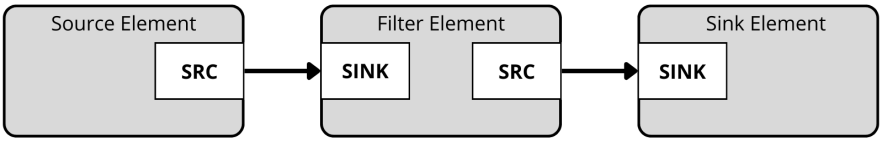

Figure 2.3: A minimal GStreamer pipeline composed of a source, a filter, and a sink element. Each connection is established between the source pad (SRC) of the upstream element and the sink pad (SINK) of the downstream element.

Buffers and the data model. Data flows through the pipeline in the form of *GstBuffers*, reference-counted containers that hold a reference to a memory region containing media data, along with associated metadata such as presentation timestamps and format descriptors. The reference-counting model allows buffers

to be passed between elements without copying the underlying data: an element receiving a buffer holds a reference, processes it, and releases it when done, at which point the memory may be reused. This design is central to achieving high throughput in performance-sensitive pipelines.

Push-based dataflow and back-pressure. GStreamer supports both push-based and pull-based dataflow models. In the push-based model, which is the one relevant to streaming video analytics, data originates at the source element and is actively pushed downstream: each element processes an incoming buffer and forwards it to the next by pushing it onto its source pad. This model is well suited to live or continuous streams where data must be processed as it arrives.

A direct consequence of the push-based model is that without explicit decoupling, an upstream element cannot push data downstream if the downstream element is not ready to receive it. The upstream element blocks, waiting for the downstream stage to complete its current operation before accepting a new buffer. This back-pressure mechanism propagates through the entire chain: a single slow element stalls every element upstream of it. In the absence of decoupling, the entire pipeline executes on a single streaming thread, with each stage running sequentially in the order data arrives.

Queue elements and multithreading. To introduce concurrency and break the blocking dependency described above, GStreamer provides the `queue` element. A `queue` decouples adjacent pipeline stages by running them on separate threads and maintaining an internal buffer between them. The upstream element pushes data into the queue on its own thread without waiting for the downstream stage to be ready; the downstream element pulls from the queue on its own thread when it is available. This producer-consumer pattern allows stages with different processing times to operate independently, prevents a slow stage from propagating back-pressure upstream, and absorbs transient timing jitter between streams. In a multi-stream pipeline, queue elements are therefore essential for enabling per-stream parallelism and for isolating the latency characteristics of individual processing stages.

Pad probes. GStreamer provides a pad probe API that allows callback functions to be attached to any pad in the pipeline. Probes are invoked synchronously on the streaming thread whenever a buffer passes through the monitored pad, and can read or modify both the buffer data and its attached metadata without interrupting the data flow. This mechanism is commonly used for monitoring, post-processing, and instrumentation purposes, and is exploited in this work for per-stage latency measurement, as described in Section 3.5.

Pipeline construction. A GStreamer pipeline is constructed programmatically by instantiating elements, setting their properties, and linking their pads. The following example illustrates the structure of a simple single-stream playback pipeline:

```
filesrc → qtdemux → h264parse → nvv4l2decoder
        → nvvideoconvert → autovideosink
```

In this pipeline, `filesrc` reads a containerized video file from disk and `qtdemux` demultiplexes the container, extracting the compressed H.264 elementary stream. At this stage, data flows as a raw encoded bitstream: no pixel information is yet available. `h264parse` prepares the bitstream for the decoder, and `nvv4l2decoder` performs hardware-accelerated decoding, producing the first actual video frames as NVMM-backed surfaces. `nvvideoconvert` handles any required format conversion, and `autovideosink` renders the frames to screen. Each arrow represents a pad link established during pipeline construction, subject to caps negotiation. Note that `nvv4l2decoder` and `nvvideoconvert` are DeepStream-specific plugins rather than standard GStreamer elements: they operate on NVMM-backed buffers and leverage the hardware acceleration engines described in Section 2.2.1. The DeepStream SDK and its extensions to the GStreamer model are described in the following subsection.

2.4.2 NVIDIA DeepStream

NVIDIA DeepStream is a streaming analytics SDK built on top of GStreamer, designed specifically for AI-powered video and sensor processing on NVIDIA hardware. It extends the GStreamer model with a set of purpose-built plug-ins, APIs, and conventions that address the specific requirements of intelligent video analytics: hardware-accelerated decoding, batched neural network inference, structured metadata propagation, and multi-stream management.

The NVMM buffer model. DeepStream is designed to operate natively on the unified memory architecture described in Section 2.2.1. Pipeline elements exchange `NvBufSurface`-backed GstBuffers, where `NvBufSurface` encapsulates one or more NVMM-resident frame surfaces along with their attributes (resolution, color format, DMA-BUF file descriptor). As long as all elements in the pipeline declare support for the `memory:NVMM` GStreamer memory feature, no data copy occurs at any stage of the processing chain, from decoding to inference.

NvDsMeta: structured metadata. GStreamer natively supports the attachment of arbitrary metadata to buffers through the `GstMeta` API. DeepStream builds on this mechanism, introducing `NvDsMeta`, a hierarchical metadata system tailored to video analytics workloads. The structure is organized across multiple levels, from batch down to individual detected objects, allowing each stage of the pipeline to read and augment the metadata without accessing the underlying pixel data. For example, inference results produced by `nvinfer` are attached at the object level, populating bounding boxes, class labels, and confidence scores for each detected instance. Custom user metadata can be attached at any level, allowing application-specific data to travel alongside the frame without requiring out-of-band communication between pipeline stages.

nvstreammux. A distinctive element in DeepStream pipelines is `nvstreammux`, the stream multiplexer. It accepts frames from N independent input streams on N sink pads and assembles them into batched `NvBufSurface` buffers suit-

able for batched inference. The batch size is configurable and determines how many frames are processed together in a single inference call. The muxer also handles synchronization between streams, ensuring that frames from different sources are combined into well-formed batches even in the presence of timing differences between streams.

nvinfer. The `nvinfer` plugin is the inference element of DeepStream. It receives batched buffers from the muxer, submits them to a TensorRT engine for inference, and attaches the resulting detections to the `NvDsMeta` structure of each frame. The inference engine is specified through an external configuration file that defines the model path, input dimensions, precision mode, and class labels, decoupling the pipeline structure from the model configuration. `nvinfer` handles internal preprocessing, including letterboxing and normalization, and manages the TensorRT execution context across batches.

Custom plugins. DeepStream's plugin architecture inherits directly from GStreamer's, and all DeepStream plugins are distributed as open-source components, with source code made available by NVIDIA. This includes `gst-dsexample`, a reference template that implements the `GstBaseTransform` interface, an in-place transform element that receives a buffer, performs arbitrary processing on it, and returns it to the pipeline. The template demonstrates the correct patterns for accessing NVMM-backed buffers and reading and writing `NvDsMeta` metadata, and serves as the recommended starting point for implementing custom processing stages. The motion gate plugin developed in this thesis is built directly on top of this template, as described in Section 3.3.

2.5 Motion Detection for Video Surveillance

Motion detection, also referred to as background subtraction or change detection, is a well-established problem in computer vision whose objective is to identify the appearance of new objects or changes in a visual scene with respect

to a previously observed reference state. In the context of video surveillance, this translates into classifying each pixel of an incoming frame as either belonging to the static background or to a foreground region of interest.

The simplest approaches to this problem, such as frame differencing or global thresholding methods like Otsu's algorithm [16], operate by comparing consecutive frames or by thresholding pixel intensity distributions. While computationally minimal, these methods rely on a static or instantaneous reference and lack any adaptive temporal modeling: they cannot distinguish between a genuine foreground event and a gradual illumination change, and are highly sensitive to camera noise and environmental dynamics such as moving vegetation or cast shadows. Their applicability in real-world outdoor surveillance scenarios is therefore limited.

A more robust class of methods models the background as a statistical process rather than a fixed reference. The seminal work of Stauffer and Grimson [17] introduced the Gaussian Mixture Model (GMM) approach to background subtraction: each pixel is modeled independently as a mixture of Gaussian distributions, whose parameters are updated incrementally as new frames arrive. The mixture captures the multimodal nature of real backgrounds: a pixel oscillating between two states due to a waving branch, for instance, is naturally represented as two distinct Gaussian components. Each incoming pixel is then tested against the current model, and classified as background if it is adequately explained by one of the existing components, or as foreground otherwise.

Zivkovic [18, 19] extended this formulation by introducing a per-pixel adaptive selection of the number of Gaussian components, together with a decay mechanism that progressively removes components whose weight falls below a threshold. As a result, each pixel is described by as many Gaussian components as its local dynamics require, without allocating resources for components that no longer contribute to the model. The resulting algorithm, commonly referred to as MOG2, retains all the adaptive properties of the original GMM formulation while reducing memory and computational overhead, making it well suited for use as a lightweight pre-filtering stage in outdoor perimeter monitoring scenarios such as

the one addressed in this work.

2.6 Object Detection Models

Object detection is the computer vision task of simultaneously localizing and classifying all instances of predefined object categories within an image. Unlike image classification, which assigns a single label to the entire input, object detection produces a set of bounding boxes, each associated with a class label and a confidence score. The task is fundamental to video surveillance applications, where the system must not only determine whether an object of interest is present, but also identify its position and extent within the frame.

2.6.1 The YOLO Family

Among the many architectures proposed for real-time object detection, the YOLO (You Only Look Once) family [20] has become one of the most widely adopted, owing to its favourable balance between inference speed and detection accuracy. Unlike two-stage detectors, which first generate region proposals and then classify them, YOLO formulates detection as a single regression problem. A single forward pass of the network produces bounding box coordinates, class probabilities, and confidence scores directly. This design makes YOLO architectures particularly well suited to real-time applications on resource constrained hardware, where minimising inference latency is a primary objective.

In this work, YOLOv8 and YOLOv11 are used as representative off-the-shelf detectors, selected for their availability and ease of deployment. Both follow the same general three stage design composed of a backbone for multi-scale feature extraction, a neck for feature aggregation across scales, and a detection head that produces bounding box and classification predictions at three spatial resolutions targeting small, medium, and large objects respectively. Both models are anchor free and rely on non maximum suppression during inference to remove redundant detections [21].

Despite this shared structure, the two architectures differ in their internal building blocks in ways that are relevant to the deployment context considered in this work. YOLOv8 employs the C2f block as the primary feature extraction unit throughout the backbone. C2f is based on the Cross Stage Partial (CSP) design principle, in which the feature map is divided into two paths. One portion of the features is processed through a sequence of convolutional bottleneck layers while the remaining portion bypasses these transformations. The two branches are subsequently concatenated, which improves gradient flow and reduces redundant computation while maintaining strong feature representation capacity.

YOLOv11 replaces the C2f block with the C3k2 module. This block retains the same general Cross Stage Partial structure but introduces a reduced parameter count. In the medium, large, and extra large model variants, internal bottleneck layers are further replaced by the C3k block, a Cross Stage Partial module containing three convolutional blocks and a sequence of internal bottleneck layers. This modification increases the expressive capacity of the feature extraction stage while maintaining comparable computational depth [21].

A more substantial architectural distinction appears in the neck of the network. Both models include an SPPF (Spatial Pyramid Pooling Fast) block at the end of the backbone, a lightweight variant of spatial pyramid pooling that aggregates contextual information at multiple receptive field sizes through stacked pooling operations. YOLOv11 adds immediately after this stage a C2PSA (Convolutional block with Parallel Spatial Attention) module, which is absent in YOLOv8.

The C2PSA module incorporates a lightweight spatial self attention mechanism that enables the network to capture long range spatial dependencies within the feature map. Such interactions are difficult to model using purely convolutional operations, which typically operate within local receptive fields. In this sense, YOLOv11 can be interpreted as a hybrid architecture that combines convolution based feature extraction with a localised attention mechanism inspired by Transformer models, while YOLOv8 remains a purely convolutional architecture [21].

These architectural differences provide a principled basis for including both

models in the experimental evaluation. They represent two distinct design points within the modern YOLO architecture space, and their comparative performance with respect to the deployment metrics of interest, namely inference throughput and detection accuracy, is therefore treated as an empirical question that the evaluation in Chapter 5 addresses directly.

2.6.2 TensorRT and Model Quantization

Deploying neural network models on embedded hardware requires not only selecting an appropriate neural network architecture, but also optimizing the inference engine for the target platform. NVIDIA TensorRT [12] is a high-performance inference SDK that takes a trained model and produces a platform-specific optimized execution plan, referred to as an *engine*. The optimization process includes layer fusion, kernel auto-tuning, and memory layout optimization, and can yield substantial throughput improvements over a naive framework-level forward pass.

A central feature of TensorRT is support for reduced-precision inference. Three precision modes are relevant to this work:

FP32. Full 32-bit floating point precision. This is the native training precision and produces results identical to the original model, at the cost of higher memory bandwidth and compute requirements.

FP16. Half-precision floating point. On hardware with native FP16 support, including the Tensor Cores present on the Jetson Orin Nano, this mode reduces memory footprint and increases throughput, typically with negligible impact on detection accuracy.

INT8. 8-bit integer quantization. This mode offers the highest throughput and lowest memory footprint, but requires a *calibration* step to determine the optimal quantization parameters. Calibration is performed on a representative dataset: the activations of each layer are profiled on a set of calibration samples, and per-layer scale factors are derived to minimize the quantization error. The quality of the

INT8 engine therefore depends on the representativeness of the calibration data with respect to the deployment scenario.

The tradeoff between precision and throughput is one of the central dimensions of the experimental evaluation in this thesis, and the practical implications of each precision mode on detection accuracy and pipeline performance are analyzed in Chapter 5.

Chapter 3

System Design & Implementation

3.1 Overview

This chapter describes the design and implementation of the video analytics pipeline developed in this work. The system is built entirely on the NVIDIA DeepStream SDK and deployed on the Jetson Orin Nano. The pipeline and all custom components are implemented in C++. It is designed to process N concurrent video streams in real time, performing object detection on each stream via a shared TensorRT inference engine.

The overall architecture follows a multi-branch, single-inference design: each input stream is handled by an independent processing branch, responsible for decoding and optional motion-based filtering; the outputs of all branches converge at a single multiplexer, which assembles batched buffers for inference. This structure is illustrated in Figure 3.1.

The chapter is organized as follows. Section 3.2 describes the pipeline architecture in detail, covering the per-stream processing branches, the multiplexing and inference stage, and the rationale behind the choice of input source. Section 3.3 presents the design and implementation of the motion gate plugin. Section 3.4 describes the inference configuration and the custom output parsing function. Section 3.5 details the latency instrumentation infrastructure.

3.2 Pipeline Architecture

3.2.1 Per-Stream Branch

Each video source corresponds to a dedicated and independent processing branch, whose structure is described in detail in the following paragraphs.

Acquisition stage: real deployment vs. experimental setup. In a production environment, video streams are delivered by IP cameras over the RTSP protocol. For H.264-encoded streams, one could construct an explicit acquisition pipeline using `rtspsrc`, which handles the RTSP connection and retrieves the RTP packetized stream; `rtph264depay`, which extracts the H.264 bitstream from the RTP packets; and `h264parse`, which parses and prepares the bitstream for the downstream decoder. An example acquisition stage would therefore be:

`rtspsrc` → `rtph264depay` → `h264parse`

For H.265-encoded streams, the corresponding H.265-compatible elements would be used instead. In any case, this first stage would then be followed by hardware decoding via `nvv4l2decoder`, which is discussed in detail in the following paragraph.

Alternatively, GStreamer's `uridecodebin` provides a higher-level abstraction that automatically selects and instantiates the appropriate demultiplexing and decoding elements based on the input URI, handling both container parsing and codec negotiation internally. DeepStream additionally provides a purpose-built wrapper called `nvurisrcbin`, which similarly encapsulates the input stage, while adding specializations relevant to production deployments such as automatic RTSP reconnection, file looping, and smart recording. These features are not of primary interest here.

Since this work focuses on the processing of video streams rather than their transport, the acquisition stage is implemented using local H.264-encoded MP4 files to ensure fully reproducible experimental conditions. The pipeline follows the decode chain recommended in the NVIDIA DeepStream best practices documentation [22], instantiating each component explicitly:

filesrc → qtdemux → queue → h264parse → nvv4l2decoder

filesrc reads the file from disk and qtdemux demultiplexes the MP4 container, extracting the compressed H.264 elementary stream. A queue element decouples the two stages, and h264parse prepares the bitstream for hardware decoding. This explicit construction makes each stage of the acquisition chain fully transparent, and maps directly onto a real RTSP deployment where filesrc and qtdemux would be replaced by the corresponding network source and depayloader elements.

Hardware decoding and entry into NVMM. At every stage up to and including the h264parse element, the buffer carries a compressed bitstream: no pixel data is yet available. The transition into the NVMM memory domain occurs at nvv4l2decoder, which interfaces with the NVDEC hardware engine to perform decoding. From this point onwards, each buffer contains an actual video frame, represented as an NVMM-backed NV12 surface allocated in the unified memory space of the Jetson Orin Nano. No CPU or GPU resources are consumed in this process, and the decoded frame is immediately accessible to any downstream hardware engine without data transfer.

Resolution downscaling via VIC. In a standard DeepStream pipeline, decoded frames would proceed directly from the per-stream branch to nvstreammux, which together with nvinfer handles the scaling and letterboxing required to match the input resolution expected by the inference model.

The introduction of the motion gate plugin, however, makes it convenient to anticipate this operation. The resolution of the frame passed to the motion analysis algorithm directly determines its execution time, and operating on a downscaled frame therefore reduces the computational cost of the motion gate without affecting the subsequent inference stage.

In this work, the input sequences have a native resolution of 1920×1080, corresponding to the VIRAT dataset used for evaluation (Section 4.2); a different

source resolution would simply result in a different scaling ratio, without affecting the pipeline structure. Rather than analyzing each frame at full resolution and then scaling it downstream anyway, the pipeline performs the downscaling before the plugin, eliminating the need for any further scaling step afterwards. Importantly, this does not introduce any additional latency: the scaling operation is simply moved earlier in the pipeline, not added on top of it.

To enforce this, an `nvvideoconvert` element followed by a `capsfilter` is inserted before the plugin. As described in Section 2.4.1, GStreamer performs caps negotiation at pipeline startup to verify format compatibility between adjacent elements. By placing an explicit `capsfilter` downstream of `nvvideoconvert`, the desired output resolution is enforced as a hard constraint: GStreamer's negotiation mechanism causes `nvvideoconvert` to configure itself to produce frames at exactly the specified resolution in order to satisfy the downstream caps. The target resolution is set to 640×360, which preserves the 16:9 aspect ratio of the source while matching the width expected by the YOLO model. The remaining height mismatch is resolved via letterboxing before inference, as described in Section 3.4.

The caps constraint is defined as follows:

```
video/x-raw(memory:NVMM), width=640, height=360
```

The `memory:NVMM` declaration ensures that the frame remains in the unified memory domain throughout the branch. `nvvideoconvert` is additionally configured to perform the scaling operation on the VIC hardware engine rather than on the CPU or GPU, via an explicit hardware flag, preserving both compute resources for inference and keeping the operation within the zero-copy memory model. No color space conversion is performed at this stage: the NV12 format produced by the decoder is preserved.

Motion gate plugin. The custom motion gate plugin, built upon the previously introduced `gst-dsexample` template, is inserted after the `capsfilter`, operating on downscaled NVMM-backed frames at the earliest possible point in

the pipeline. Since each per-stream branch instantiates its own independent plugin element, there are N plugin instances running concurrently at runtime. This placement ensures that frames dropped by the motion gate incur no downstream processing cost, as they are discarded before reaching the multiplexer or the inference engine. The design and implementation of the plugin are described in detail in Section 3.3.

Output queue. In addition to the decoupling queue inserted between `qtdemux` and `h264parse` in the acquisition stage, each branch includes a second `queue` element at its output, placed immediately downstream of the motion gate plugin. Its purpose is to decouple the processing speed of the plugin from the consumption rate of the downstream `nvstreammux`: the plugin processes incoming frames at its own pace and pushes those that pass the motion test into the queue, while the muxer dequeues them in FIFO order according to its own timing, without backpressure propagating upstream and stalling the branch.

The queue is left at its default GStreamer configuration: a maximum capacity of 200 buffers, 10MB, or 1 second of data, whichever limit is reached first. When the queue is full, the default behavior is blocking: the upstream element stalls until space becomes available. No frames are discarded at the queue level. These parameters are fully configurable via the `max-size-buffers`, `max-size-bytes`, `max-size-time`, and `leaky` properties, allowing the queue behavior to be tuned for specific deployment requirements.

Per-stream branch summary. Taking stock of the elements described above, each per-stream branch is composed as follows:

$$\texttt{filesrc} \rightarrow \texttt{qtdemux} \rightarrow \texttt{queue} \rightarrow \texttt{h264parse} \rightarrow \texttt{nvv4l2decoder} \rightarrow \texttt{nvvideoconvert} \rightarrow \texttt{capsfilter} \rightarrow \texttt{dsexample} \rightarrow \texttt{queue}$$

`filesrc` simulates an RTSP source by reading a local video file, with the demuxing and decoding stages decoupled by an intermediate queue. Decoded

frames are downscaled via VIC and passed to the motion gate plugin `dsexample`, where they are either forwarded downstream or dropped depending on the detected motion content, as described in Section 3.3. Frames that pass the gate are enqueued in the output queue, which connects each independent branch to the convergence point of the pipeline, where all N parallel branches meet the shared multiplexing and inference stage described in the following section.

3.2.2 Multiplexing and Inference Stage

nvstreammux. The N per-stream branches converge at a single `nvstreammux` element, which collects frames from all branch queues and assembles them into batched `NvBufSurface` buffers. The muxer is configured with a batch size equal to N, matching the number of active streams, and attempts to collect one frame from each branch before forwarding the batch to inference.

In practice, however, the branches may not deliver frames in perfect synchrony. This is an inherent consequence of the decoupled, multi-threaded pipeline design, and is further amplified when the motion gate is active: streams with little or no motion will contribute fewer frames to the queue, potentially leaving the muxer waiting. To bound this waiting time, `nvstreammux` exposes a `batched-push-timeout` parameter, expressed in microseconds. When this timeout expires, the muxer closes and forwards the batch with whatever frames have been collected up to that point, without waiting for the remaining branches. This mechanism prevents a slow or inactive stream from indefinitely stalling the inference stage, at the cost of potentially incomplete batches during periods of low motion activity.

nvinfer. The batched output of `nvstreammux` is passed to `nvinfer`, the DeepStream inference element, which submits each batch to a precompiled TensorRT engine for object detection. The `nvinfer` element is configured via an external configuration file specifying the engine path, input dimensions, precision mode, and class labels. Before inference, `nvinfer` performs internal preprocessing of each frame in the batch, including letterboxing to the model input resolution

of 640×640. Detection results are attached to the `NvDsMeta` structure of each frame and propagated downstream.

Sink. In the benchmarking configuration used for experimental evaluation, the pipeline terminates with a `fakesink` element, which discards all output without any rendering overhead. This ensures that sink-side processing does not contribute to the measured latency or throughput figures. More details in Section 4.1.

In a deployment scenario, this element would be replaced by an appropriate output stage, such as an RTSP re-streaming sink, a recording element, or an on-screen display.

3.2.3 Pipeline Summary

Figure 3.1 provides a simplified view of the pipeline architecture, showing N parallel per-stream branches (illustrated here for $N = 4$) converging at the shared multiplexing and inference stage. The acquisition chain upstream of `nvv4l2decoder`, described in the previous section, is omitted for clarity. Each branch comprises the acquisition, decoding, scaling, and motion filtering elements described above; all branches converge at the shared multiplexing and inference stage. Frames that are dropped by the motion gate never reach `nvstreammux`, and are therefore entirely absent from the inference stage. Figure 3.2 provides the complete view of the pipeline, as generated by the built-in GStreamer pipeline visualization tool, with all elements and pad connections visible.

3.2.4 Source Configuration and Reproducibility

The choice of local video files as input sources, rather than live RTSP streams, is motivated exclusively by the need for experimental reproducibility. All benchmarking experiments are conducted on the same set of video sequences from the VIRAT dataset [23], described in detail in Section 4.2. Using local files guarantees that every experimental run processes exactly the same frames in the same order, making results directly comparable across pipeline configurations. In particular,

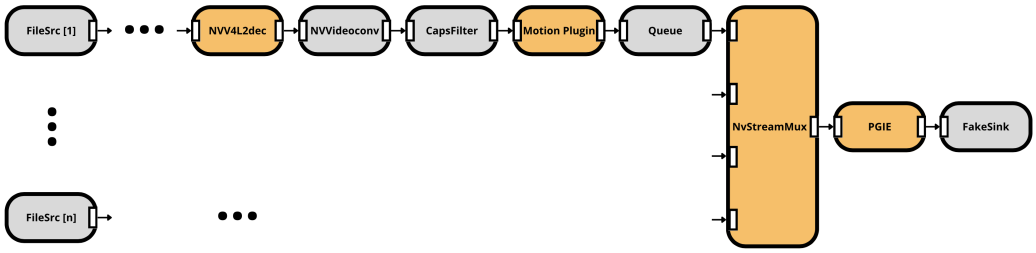

Figure 3.1: Simplified view of the pipeline for $N = 4$ concurrent streams. The acquisition stage upstream of `nvv4l2decoder` is omitted for clarity.

the comparison between the motion-gated and non-gated pipeline configurations, which is the central experimental question of this work, requires that both configurations process identical input, which is only guaranteed with file-based sources.

3.3 Motion Gate Plugin

3.3.1 Design Rationale

As established in Section 1.2, the target deployment scenario is characterized by largely static scenes in which events of interest occur sporadically. Submitting every incoming frame to the inference engine regardless of its content therefore represents a significant waste of computational resources. The motion gate plugin exploits this characteristic by selectively dropping frames that carry no relevant motion, with the goal of reducing the number of frames reaching `nvstreammux` and consequently the load on the inference engine.

The plugin operates directly on NVMM-backed buffers, without introducing data copies or transfers outside the unified memory domain. To achieve this, the implementation relies on NVIDIA VPI, which provides native support for NVMM memory interoperability, as described in Section 2.3. Beyond memory compatibility, VPI offers a hardware-optimized implementation of MOG2 for Jetson platforms, and its backend abstraction layer makes it straightforward to evaluate different execution targets, such as CUDA or CPU, by changing a single configura-

Figure 3.2: DeepStream pipeline architecture used in the experimental setup. The diagram was automatically generated from the running pipeline using the GStreamer pipeline graph utility and rendered with Graphviz `dot`. N parallel per-stream branches converge at `nvstreammux`, where frames are batched before the shared inference stage.

tion flag, without modifying the surrounding application code.

3.3.2 GStreamer Plugin Structure

Like any other element in a GStreamer pipeline, the motion gate plugin exposes a sink pad and a source pad, and participates in caps negotiation with adjacent elements. What distinguishes it from the pre-built elements used elsewhere in the pipeline is that its processing logic is implemented from scratch, starting from the `gst-dsexample` template and adapting it entirely to the motion gating purpose.

The plugin is structured around a set of lifecycle callbacks that the GStreamer framework invokes at well-defined points during pipeline execution:

init. Invoked once per element instantiation. Initializes all instance variables to their default values.

start / stop. Invoked when the pipeline transitions into and out of the playing state. Used to allocate and release all VPI resources required by the processing logic, as detailed in the following subsection.

transform_ip. The core processing function, invoked once per incoming buffer on the streaming thread. Contains the motion analysis logic and the frame drop decision.

Each instantiation of the element in the pipeline creates an independent GObject instance with its own private state. As a result, the N plugin instances running concurrently in the multi-stream pipeline maintain fully independent state, including separate VPI resources and background models, with no shared mutable data between streams.

3.3.3 Initialization and VPI Resource Allocation

All VPI resources are allocated in the `start` callback, once per instance, when the pipeline enters the playing state. Since frames arrive at the plugin al-

ready downscaled to the target resolution, no format conversion between hardware engines is required. A VPI stream is created targeting either the CUDA or CPU backend depending on the chosen configuration. Should independent scheduling of different operations be desired, separate streams can be used for background subtraction and image statistics respectively, decoupling their execution.

Alongside the stream, the MOG2 background subtractor payload is initialized for the configured processing resolution and input format, together with the auxiliary image buffers and arrays required by the subsequent processing steps, including the foreground mask. Pre-allocating these structures at startup and reusing them across frames avoids per-frame allocation overhead on the streaming thread. All resources are released in the `stop` callback when the pipeline is torn down.

3.3.4 Zero-Copy Buffer Access via DMA-BUF

The `transform_ip` function receives a `GstBuffer` for each incoming frame. The buffer is mapped to retrieve a pointer to the underlying `NvBufSurface`, the top-level DeepStream structure for NVMM-backed surfaces. The `NvBufSurface` holds an array of `NvBufSurfaceParams` entries, one per frame in the batch; each entry exposes a DMA-BUF file descriptor through which the NVMM memory region is accessible to any compatible hardware engine.

Rather than copying the frame data, the plugin constructs a `VPIImage` that wraps the existing memory region by referencing this file descriptor directly. Since the wrapper depends on the DMA-BUF descriptor of the incoming frame, which is only available at runtime, it cannot be pre-allocated at startup. On the first frame, a `VPIImage` wrapper is therefore created via `vpiImageCreateWrapper`; on all subsequent frames, the same object is reused via `vpiImageSetWrapper`, updating the memory reference without allocating a new object. This contrasts with resources such as the foreground mask, which are VPI-owned buffers with known dimensions and can therefore be pre-allocated in `start`.

3.3.5 MOG2 Background Subtraction via VPI

With the current frame available as a `VPIImage`, the background subtraction step is submitted to the VPI stream. The MOG2 algorithm updates the per-pixel Gaussian mixture model incrementally and produces a foreground mask: a single-channel `U8` image in which foreground pixels are set to 255 and background pixels to 0.

Shadow detection is enabled in the algorithm configuration. By default, MOG2 assigns a value of 127 to pixels identified as shadows, distinguishing them from both background (0) and foreground (255). In this implementation, shadow pixels are explicitly remapped to 0, keeping the mask strictly binary. This is not equivalent to disabling shadow detection entirely: with shadow detection disabled, shadow pixels would be misclassified as foreground and assigned value 255, inflating the motion pixel count. By enabling detection and remapping shadows to 0, they are correctly identified and excluded from the foreground mask, without contributing to the gating decision.

The background model is updated at each frame with a configurable learning rate, which governs the speed at which the model adapts to gradual illumination changes and slow scene variations.

3.3.6 Foreground Pixel Counting and Frame Drop Decision

Once the foreground mask is available, pixel-level statistics are computed via `vpiSubmitImageStats` to count the number of pixels carrying value 255, representing detected motion. This count provides a scalar measure of the motion content of the current frame.

The count is compared against a threshold. If it falls below the threshold, the frame is dropped by returning `GST_BASE_TRANSFORM_FLOW_DROPPED`, which signals GStreamer to discard the buffer without forwarding it downstream. If it meets or exceeds the threshold, `GST_FLOW_OK` is returned and the buffer proceeds to the output queue.

The appropriate threshold depends on the deployment scenario. Scenes with

large objects or close-range fields of view produce high pixel counts for relevant events, allowing a higher threshold; distant scenes or small objects require lower thresholds to avoid missing events, at the cost of increased sensitivity to noise and background model errors. The threshold is therefore a deployment-time parameter that must be calibrated to the specific installation.

In the experimental evaluation presented in Chapter 5, the frame drop mechanism was adapted to simulate controlled motion densities in a reproducible manner, as described in the experimental setup chapter.

3.4 Inference Configuration

3.4.1 TensorRT Engine and nvinfer Configuration

Both `nvstreammux` and `nvinfer` are configured via external YAML files, which decouple the pipeline structure from the model and inference parameters and allow different experimental configurations to be applied without modifying the pipeline code.

The inference engine is specified as a precompiled TensorRT `.engine` file. While `nvinfer` supports providing an ONNX model directly and compiling the engine at runtime, this option was not adopted in this work. Precompiling the engine offers two practical advantages: it eliminates compilation time at pipeline startup, and it allows the same engine file to be shared across multiple experimental runs that differ only in parameters unrelated to the model or precision, such as the motion density, without redundant recompilation. The engine generation process is described in Section 4.3.

The precision mode is set via the `network-mode` parameter, which accepts the values 0 (FP32), 1 (INT8), and 2 (FP16). The input dimensions are declared as `3;640;640` to match the expected square input of the YOLO model. As described in Section 3.2, frames arrive at the muxer at a resolution of 640×360; the remaining dimension mismatch is resolved internally via symmetric letterboxing, controlled by the `maintain-aspect-ratio` and `symmetric-padding`

flags in the configuration file.

Of particular relevance to the custom parsing architecture is the `output-tensormeta` flag, which instructs `nvinfer` to attach the raw output tensor to the `NvDsMeta` structure, making it directly accessible to the custom parser.

3.4.2 Custom Output Parsing

DeepStream's built-in parsing functions are not compatible with the output format of YOLOv8 and YOLOv11, which requires a custom implementation. The parser is compiled as a shared library and registered in the `nvinfer` configuration via `custom-lib-path` and `parse-bbox-func-name`.

The raw output tensor produced by the TensorRT engine has shape $[84 \times N_{\text{pred}}]$, where 84 corresponds to 4 bounding box parameters plus 80 per-class confidence scores, and N_{pred} is the total number of prediction slots generated by the detection head. Both YOLOv8 and YOLOv11 use a three-scale detection head with strides 8, 16, and 32. For a 640×640 input, this produces grids of 80×80, 40×40, and 20×20 respectively, for a total of 8400 prediction slots. Table 3.1 illustrates the tensor layout.

Table 3.1: YOLO output tensor layout ($84 \times N_{pred}$).

	Slot 0	**Slot 1**	**Slot 2**	**…**	**Slot** $N-1$
Row 0 (c_x)				…	
Row 1 (c_y)				…	
Row 2 (w)				…	
Row 3 (h)				…	
Row 4 (class 0)				…	
⋮			⋮		
Row 83 (class 79)				…	

Each column corresponds to one prediction slot.

The parser is implemented as a CUDA kernel that processes all prediction slots in parallel. For each slot, the kernel identifies the class with the highest confidence score and compares it against the threshold defined in the `nvinfer` configuration

file. Slots whose maximum score falls below the threshold are discarded. For the remaining slots, the bounding box is converted from center-width-height format to corner coordinates and clamped to the network input dimensions. The results are collected into a `NvDsInferParseObjectInfo` array and forwarded to DeepStream's metadata infrastructure for downstream processing. Separate parser entry points are provided for YOLOv8 and YOLOv11.

3.5 Latency Instrumentation

3.5.1 CustomLatencyMeta

To measure per-frame latency across the pipeline stages, a custom metadata structure is attached to each frame as it enters the processing chain. The structure, referred to as `CustomLatencyMeta`, is allocated as a `NvDsUserMeta` object and attached at the frame level within the `NvDsMeta` hierarchy. It carries two identifiers: the stream identifier and a per-stream absolute frame sequence number, along with five nanosecond-resolution timestamps corresponding to the measurement points described in the following subsection.

The explicit tracking of both identifiers is necessary for the correct reconstruction of per-frame metrics at analysis time. The stream identifier disambiguates frames arriving from different sources within the same batch. The absolute frame sequence number is required because the motion gate plugin drops frames before they reach the multiplexer: from the perspective of all downstream elements, the sequence of received frames is no longer contiguous, and any frame counter maintained downstream would reflect only the frames that passed the gate. By attaching the absolute sequence number at the decoder output, before any drop can occur, each frame that eventually reaches the sink carries an unambiguous reference to its position in the original input stream, allowing latency measurements and detection results to be correctly aligned in post-processing.

The `CustomLatencyMeta` structure travels alongside the frame buffer through the entire pipeline, without requiring any out-of-band communication be-

tween stages.

3.5.2 Probe Placement

Timestamps are recorded by GStreamer pad probes attached to the source pads of five elements, each marking the moment a buffer exits that stage. The probe placement is illustrated in Figure 3.3. The five measurement points are:

1. **NVDEC** — source pad of `nvv4l2decoder`: marks the completion of hardware decoding. This is the earliest point at which a discrete frame exists in the pipeline: upstream of the decoder, the buffer carries a compressed bitstream with no notion of individual frames, making it impossible to attach per-frame metadata before this stage. The `CustomLatencyMeta` structure is therefore created and attached here, at the first moment a frame identity can be established.

2. **MOG2** — source pad of `dsexample`: marks the completion of motion analysis and the frame drop decision.

3. **Queue** — source pad of the output `queue`: marks the moment the frame exits the per-stream branch and is ready to be consumed by the multiplexer.

4. **MUX** — source pad of `nvstreammux`: marks the completion of batch assembly.

5. **PGIE** — source pad of `nvinfer`: marks the completion of inference and the attachment of detection results to `NvDsMeta`.

Each probe reads the current monotonic clock time and writes it into the corresponding field of the `CustomLatencyMeta` structure. A final probe placed on the sink element triggers the computation of inter-stage intervals and writes the results to the output files. The four intervals derived from the five timestamps are: pre-filtering time (MOG2 − NVDEC), queue residence time (Queue − MOG2), muxer assembly time (MUX − Queue), and inference time (PGIE − MUX). The total end-to-end latency is the sum of all segments.

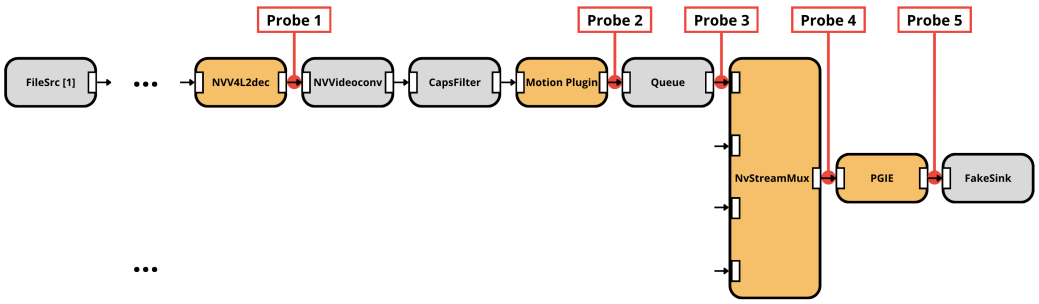

Figure 3.3: Pipeline architecture for $N = 4$ concurrent streams, with latency measurement probes shown at their respective positions. The acquisition stage upstream of `nvv4l2decoder` is omitted for clarity.

It should be noted that the first interval, here referred to as pre-filtering time, encompasses not only the MOG2 background subtraction itself but also the preceding resolution downscaling performed by `nvvideoconvert` via the VIC engine. These two operations are measured as a single block, as the scaling stage is directly preparatory to the motion analysis and cannot be meaningfully separated from it in this instrumentation scheme. The contribution of the scaling step to this interval is expected to be negligible: VIC-accelerated rescaling on the Jetson Orin Nano operates in the order of 0.5 ms [24], well below the typical execution time of the MOG2 algorithm.

3.5.3 Output Files

At the end of each experimental run, three output files are produced.

`latency.csv` contains one row per processed frame, reporting the stream identifier, absolute frame sequence number, and the duration in milliseconds of each inter-stage interval together with the total end-to-end latency.

`predictions_results.csv` contains one row per detected object, reporting the stream identifier, frame identifier, object identifier, class label, bounding box coordinates, and detection confidence score.

`metadata.json` contains run-level summary statistics: the number of ac-

tive streams, total run time, per-stream FPS, and aggregate pipeline FPS. The file also records the warmup configuration, which excludes an initial segment of frames from throughput calculations to avoid including pipeline startup transients in the measurements. The warmup mechanism is discussed in detail in Section 5.7.2.

Chapter 4

Experimental Setup

4.1 Platform Configuration

To ensure that performance measurements reflect the maximum sustainable throughput of the platform, the Jetson Orin Nano Super was configured according to the guidelines provided in the NVIDIA DeepStream performance documentation [25].

All experimental runs were conducted in a standard desktop environment. Pipeline output is not rendered at any point: the pipeline terminates with a dedicated `fakesink` element immediately after `nvinfer`, as described in Section 3.2, ensuring that no resources are consumed by display or rendering operations. Each batch of experiments was conducted on a freshly rebooted system with no background applications running, so as to eliminate interference from unrelated processes on the measured metrics.

The platform was set to its maximum power mode via:

```
sudo nvpmodel -m 2
sudo jetson_clocks
```

The `nvpmodel` command sets the power mode to MAXN, which unlocks the full CPU and GPU clock budget available on the Jetson Orin Nano Super [26]. The `jetson_clocks` command subsequently fixes all clocks at their maximum

frequency, preventing dynamic frequency scaling from introducing variability in the measurements.

Finally, the batch size of both `nvstreammux` and `nvinfer` was set equal to the number of active input streams in each experimental configuration. This ensures that, under nominal conditions, the muxer can assemble a complete batch before forwarding it to inference. In practice, as discussed in Section 3.2, the `batched-push-timeout` mechanism may cause incomplete batches to be forwarded when one or more streams are inactive or motion-gated.

4.2 Dataset

4.2.1 VIRAT Video Dataset

The VIRAT Ground dataset [23] is a large-scale collection of surveillance video sequences recorded by fixed cameras in outdoor public environments, including parking lots, walkways, and open areas. The dataset comprises 11 scenes, each captured by stationary HD cameras and encoded in H.264. Each sequence is accompanied by ground truth annotation files in a structured format. The object-level annotation file (`.viratdata.objects.txt`) provides, for each annotated object, the following fields:

```
object_id, duration, current_frame, x, y, w, h,
                    object_type
```

The dataset defines five object classes (types): person (1), car (2), vehicle excluding passenger cars (3), carried object (4), and bike (5). Not all sequences in the dataset are accompanied by complete annotation files; only sequences for which the full annotation set was available were considered as candidates for evaluation.

4.2.2 Scene Selection

The selection of the evaluation sequence was guided by three criteria motivated by the target deployment scenario of this work. First, the camera viewpoint should be consistent with a realistic domestic or small commercial installation: scenes recorded from highly elevated positions at large distances from the subjects were excluded, as the resulting object scale would not be representative of a perimeter surveillance context. Second, the scene should not be excessively crowded: sequences featuring large numbers of simultaneously present vehicles or pedestrians were considered unrepresentative of the low-traffic scenarios targeted by this work. Third, the sequence should contain instances of both persons and vehicles in order to enable evaluation across both object classes of interest.

Applying these criteria to the available annotated sequences, the VIRAT _S _000002 sequence was selected. The sequence features a moderately trafficked street scene recorded at 1920×1080 resolution and 30 fps, from a viewpoint consistent with a realistic perimeter installation, with both pedestrians and passenger cars present. The first 3600 frames, corresponding to 120 seconds of footage, were used for evaluation: this duration was sufficient to include representative instances of all relevant object classes, without extending the sequence beyond what was necessary for the evaluation purposes of this work.

4.2.3 Ground Truth Preprocessing

The original VIRAT annotation format was converted to match the output format of the detection pipeline. The conversion involved three operations. First, the `duration` field, which is not relevant to per-frame evaluation, was removed. Second, objects of classes 3, 4, and 5 were discarded, retaining only persons and cars, which are the sole object classes present in the selected sequence. Third, the class identifiers were remapped to align with the COCO class indices used by the YOLO models: VIRAT class 1 (person) was remapped to class 0, and VIRAT class 2 (car) was retained as class 2.

The resulting per-frame ground truth format is:

```
FrameID; ObjID; ClassID; BBox_x; BBox_y; BBox_w; BBox_h
```

Bounding box coordinates were additionally scaled from the native annotation resolution of 1920×1080 to the pipeline output resolution of 640×360, to enable direct comparison with the predicted bounding boxes produced by the detection pipeline.

4.3 Model Preparation

4.3.1 ONNX Export

All models are distributed by Ultralytics in PyTorch format (`.pt`). A dedicated Python script automates the preparation pipeline: for each model variant, the corresponding `.pt` weights are downloaded via the Ultralytics Python API and subsequently exported to the ONNX intermediate representation using the same API. The export is performed with a fixed input size of 640×640 and with dynamic shapes disabled.

For FP32 and FP16 engines, a separate ONNX graph is exported for each combination of model variant and batch size. For INT8, the FP32 ONNX is reused directly, since quantization is applied during the TensorRT compilation step rather than at export time.

4.3.2 TensorRT Engine Compilation

FP32 and FP16 engines are compiled from the corresponding ONNX graphs using `trtexec` [27], the TensorRT command-line compilation tool. For FP32, the graph is compiled without additional flags; for FP16, the `--fp16` flag is passed to enable reduced-precision inference. In both cases, `trtexec` performs layer fusion, kernel selection, and platform-specific optimization for the Jetson Orin Nano, producing a serialized `.engine` file that is specific to the target hardware and TensorRT version.

INT8 engines require an additional calibration step to determine the quantization parameters for each layer. This is handled by `polygraphy` [28], the NVIDIA graph inspection and conversion toolkit, which wraps the TensorRT calibration API and accepts a custom Python data loader to supply representative input samples during the calibration process.

4.3.3 INT8 Calibration Dataset

The calibration dataset consists of 500 frames extracted from the sequence `VIRAT_S_000101`, a clip from the same VIRAT scene location as the evaluation sequence but recorded from a different camera angle and at a different point in time. The selected segment was chosen to include a representative variety of scene content: an initial portion showing the road in a predominantly static configuration, followed by segments in which both pedestrians and vehicles are present. Using a clip from the same scene family as the evaluation sequence, but from a distinct viewpoint and temporal segment, ensures that the calibration data is visually relevant without being identical to the evaluation input.

4.4 Baseline Python Pipelines

To provide a reference point for evaluating the performance of the DeepStream pipeline, a set of baseline implementations was developed in Python. The underlying question is whether a DeepStream-based approach offers a measurable advantage over a conventional Python implementation in terms of throughput, and under what conditions this advantage becomes significant.

All baselines follow the same logical flow: frames are read from a local video file, downscaled to 640×360 for motion analysis, optionally processed by a MOG2 background subtractor with foreground pixel counting, batched, preprocessed via letterboxing to 640×640, and submitted to inference using the same precompiled TensorRT engine used in the DeepStream pipeline. No motion-based frame dropping is applied: all frames are forwarded to inference, simulating a worst-case scenario in which the hardware is fully occupied at all times.

Five variants are implemented, in order of increasing GPU utilization. The first is a purely sequential implementation, in which all streams are processed one after the other in a single thread, serving as a naïve lower bound. The second introduces a producer–consumer multithreaded architecture, with one producer thread per stream handling decoding and motion analysis in parallel, and a single consumer thread assembling batches for inference. Each producer pushes frames into a bounded FIFO queue with a maximum capacity of 200 frames, consistent with the default queue configuration used in the DeepStream pipeline (Section 3.2), so as to prevent unbounded memory growth under sustained load. The remaining three variants progressively offload computation to the GPU via CUDA: the third offloads only the pre-MOG2 resize operation; the fourth additionally offloads the letterboxing preprocessing (LB) prior to inference; the fifth extends GPU execution to the MOG2 computation itself using the OpenCV CUDA backend, so that resize, letterboxing, and background subtraction all execute on the GPU.

Each variant is evaluated in two configurations: with and without the MOG2 stage, for a total of ten test runs. All runs use the same input sequence from the VIRAT dataset, with a warmup period of 500 frames excluded from measurements. All tests are conducted on the Jetson Orin Nano Super configured in MAXN mode (`nvpmodel -m 2`, `jetson_clocks`) on a freshly rebooted system.

The comparison against DeepStream is conducted on the least demanding configuration: YOLOv8n, batch size 4, INT8 precision. This represents the most favorable scenario for the Python pipelines, minimizing inference cost and therefore maximizing the relative weight of any framework-level advantages. If DeepStream outperforms all Python variants even under these conditions, the result generalizes by construction to heavier model configurations. The quantitative comparison is presented in Chapter 5.

4.5 Experimental Configurations

The experimental evaluation proceeds along a structured path, starting from a comparison with alternative implementations and progressively narrowing the

focus toward the contributions of this work: the design and systematic characterisation of the DeepStream-based multi-stream pipeline, and the integration and evaluation of the motion-gated inference plugin under varying scene conditions.

4.5.1 DeepStream vs Python Baselines

Before engaging with the full evaluation, a preliminary comparison establishes whether the DeepStream implementation offers a meaningful advantage over a conventional Python-based approach. As detailed in Section 4.4, five Python pipeline variants of increasing complexity are evaluated against the DeepStream pipeline on the least demanding inference configuration: YOLOv8n, batch size 4, INT8 precision. The results of this comparison are reported in Chapter 5.

4.5.2 Reference Baseline: NO_MOG2

The DeepStream pipeline without any motion gate plugin serves as the primary reference configuration, referred to as **NO_MOG2**. All frames from all streams are submitted to inference unconditionally; resolution downscaling to 640×360 is still performed upstream of `nvstreammux`, so that the comparison with gated configurations is not confounded by preprocessing differences.

This configuration is evaluated across all 96 inference engine combinations: two model families (YOLOv8 and YOLOv11), four size variants (nano, small, medium, large), three precision modes (FP32, FP16, INT8), and four batch sizes (4, 6, 8, 10). The extra-large variant was excluded due to memory constraints at the largest batch sizes. These 96 configurations provide the throughput reference against which all subsequent configurations are compared. Detection accuracy (mAP@50) is computed on this configuration only: when a frame reaches the inference engine its output is independent of the upstream pipeline, so the per-model accuracy figures characterise the models themselves and serve as a fixed reference for the subsequent tradeoff analysis.

4.5.3 Execution Backend for Motion Analysis

A non-trivial design decision concerns the choice of VPI backend for the MOG2 algorithm. The initial intuition was to execute background subtraction on the CPU: in the DeepStream pipeline the CPU is largely idle during inference, its role being limited to scheduling, metadata handling and pipeline coordination, while the GPU is heavily loaded by the inference engine. Offloading MOG2 to the CPU would in principle leave the GPU fully dedicated to inference, avoiding resource contention.

However, the Jetson platform introduces considerations that make this choice less straightforward. Both backends operate within the same unified memory architecture, but differ in how they interact with the NVMM memory domain and in the computational throughput they can bring to bear on the background subtraction workload. The net effect of choosing one backend over the other is therefore not predictable from first principles alone, and requires empirical evaluation.

To resolve this question empirically, the two backends are compared at three operating points spanning the full motion density range. At 100% motion density, **MOG2_CPU_NODROP** is compared against **MOG2_CUDA_NODROP**; at 50%, **MOG2_CPU_50** is compared against **MOG2_CUDA_50**; and at 5%, **MOG2_CPU_5** is compared against **MOG2_CUDA_5**, all defined as part of the broader experimental setup in Section 4.5.4.

These three points bracket and sample the inference load spectrum: at 100% the GPU is maximally occupied by inference, maximising the potential benefit of CPU offloading; at 5% the inference load is minimal, making the comparison more sensitive to the raw cost of the MOG2 computation itself. If the CUDA backend outperforms CPU at all three operating points and across the 96 model configurations, the conclusion extends to the entire operating range, and all subsequent analysis is conducted exclusively on the CUDA backend.

4.5.4 Motion Gating Tradeoff

The central question of this work is whether motion-based frame filtering improves pipeline throughput, to what extent, and under what conditions. This is addressed by comparing the following configurations across all 96 inference engine combinations:

- **NO_MOG2**: throughput without any plugin, the upper bound against which the cost of adding the plugin is measured.
- **MOG2_CUDA_NODROP**: plugin active on CUDA backend, no frames dropped. Quantifies the overhead of the plugin in isolation, without any associated filtering benefit.
- **MOG2_CUDA**_k, $k \in \{5, 10, 25, 50, 75\}$: gated configurations with simulated motion densities, where k is the percentage of frames forwarded to inference per stream.

Since the natural motion content of the input video is fixed and not directly controllable, the frame drop decision in the gated configurations is governed by a deterministic pseudo-random mechanism with a fixed global seed, as described in Section 5.7. The N concurrent plugin instances draw from the same sequence on their respective streaming threads; since they run in parallel, their calls interleave, producing decorrelated drop patterns across streams. This approximates a deployment scenario in which different cameras observe motion independently, while ensuring full reproducibility across runs. It is however a simplification of realistic motion dynamics, in which events tend to be temporally clustered; the implications of this approximation are discussed in Section 5.7.3.

Data are collected across the full motion density range without applying any minimum throughput filter, in order to characterise the complete functional relationship between motion density and pipeline throughput. The deployment viability criterion used throughout this work is introduced in Section 4.5.6 below.

4.5.5 Real-Condition Validation

As a final step, the pipeline is evaluated on the unmodified VIRAT sequence with the actual pixel-count-based drop decision active (**MOG2_CUDA**), using the empirically calibrated threshold of 500 pixels on the 640×360 foreground mask. Unlike the simulated configurations, this run exercises the full pipeline under conditions close to a real deployment, including the natural variability of motion across frames and the resulting non-uniform drop patterns.

In this configuration, a periodic pass-through mechanism is also active: regardless of the motion decision, one frame every ten matched as background is unconditionally forwarded to inference. This provides a low-cost background check that ensures the system does not miss slowly developing events during extended periods of apparent stillness, at the cost of a small additional inference load.

A relevant limitation of this configuration concerns the correlation between streams. In all experimental runs, the same video file is replicated across all N input channels. In the simulated motion configurations this choice has no practical consequence, since the drop decision is governed by a per-stream pseudo-random mechanism that is independent of the video content. Here, however, the drop decision is driven by the actual foreground pixel count, which is identical for all streams at every frame. As a result, all channels pass or drop frames simultaneously, producing a fully correlated drop pattern across streams. This correlation has a dual effect on pipeline behaviour: it may benefit batch assembly in `nvstreammux`, since frames from all streams tend to arrive together and the `batched-push-timeout` mechanism is less likely to be triggered, but it does not replicate the decorrelated behaviour expected in a real multi-camera installation where different cameras observe independent portions of the scene. This configuration should therefore be interpreted as a pseudo-real stress test rather than a faithful simulation of operational conditions, and its results are discussed accordingly in Chapter 5.

This configuration is used to evaluate detection delay metrics, capturing the effect of motion gating on the timeliness with which objects are first detected, and complementing the throughput analysis with a qualitative assessment of the

operational impact of the gate.

4.5.6 Deployment Viability Criterion

In this work, 12.5 fps per stream is adopted as the deployment viability threshold, consistent with the minimum frame rates recommended by IEC 62676-4 [29]. As this represents the most stringent suggested minimum in the standard's application guidelines, it provides a conservative and broadly applicable criterion for evaluating pipeline deployment viability.

4.6 Evaluation Metrics

This section defines and motivates the three metrics used to evaluate the pipeline across the experimental configurations described in Section 4.5. The primary metric is throughput, expressed as frames per second per stream, which directly captures the pipeline's ability to sustain real-time multi-stream inference under varying load conditions. Detection accuracy is assessed through mAP@50, providing a model-level characterisation that is independent of the pipeline configuration. Finally, Time To First Detection (TTFD) is introduced as an operational latency metric specific to the gated pipeline, quantifying the delay between an object's first appearance in the scene and its first valid detection by the system.

4.6.1 Throughput

The primary performance indicator of the pipeline is **throughput**, expressed in frames per second per stream (fps/stream). This quantity measures the sustainable rate at which the pipeline processes and analyses video frames from a single source, averaged over the duration of an experimental run after excluding the warmup period of 500 frames. Throughput is computed from the run-level summary recorded in `metadata.json` at the end of each run, as the ratio of total processed frames to total elapsed pipeline time.

It is important to note that this measurement is a **proxy** for the true end-to-end operational throughput, for two distinct reasons.

The first concerns the measurement boundary within the pipeline. As described in Section 3.5, it is structurally impossible to attach per-frame metadata before the output of the hardware decoder (**NVDEC**), since upstream of that stage the pipeline operates on a compressed bitstream with no notion of individual frame identity. The first point at which a discrete frame can be tracked is therefore the NVDEC source pad, which is also the earliest feasible measurement boundary. The throughput figure consequently captures the rate at which frames are processed from the decoder output onward, implicitly assuming that the upstream decoding stage operates without becoming a bottleneck, an assumption that is reasonable for the hardware-accelerated NVDEC engine on the Jetson Orin Nano Super, but is not directly verified by the measurement itself.

The second reason concerns the input modality. In all experimental runs, video is read from local MP4 files on the device storage rather than from live RTSP streams. A real deployment would introduce additional sources of variability: network jitter, retransmission delays, and buffering effects at the RTSP ingestion layer. Local file reading provides a more controlled and reproducible input, but systematically underestimates the latency that a deployed system would experience at the stream ingestion boundary. The throughput figures reported in this work should therefore be interpreted as pipeline-internal performance characterisations rather than end-to-end latency guarantees for a networked deployment.

Within these constraints, fps/stream remains the most informative and directly actionable metric for the purposes of this work, as it determines whether the pipeline can sustain the minimum detection rate required by the deployment context. The 12.5 fps/stream deployment viability criterion defined in Section 4.5.6 is used throughout the analysis.

4.6.2 Detection Accuracy: mAP@50

Detection accuracy is evaluated using the **mean Average Precision at IoU threshold 0.50** (mAP@50), a standard metric in the object detection literature [30].

The metric provides a joint measure of precision and recall across all detected objects, and is well suited to characterising the tradeoff between false positives and false negatives in a class-conditional manner.

Intersection over Union. The geometrical criterion used to determine whether a predicted bounding box constitutes a valid detection of a ground-truth object is the *Intersection over Union* (IoU), defined for two axis-aligned boxes A and B as:

$$\mathrm{IoU}(A, B) = \frac{\mathrm{area}(A \cap B)}{\mathrm{area}(A \cup B)} = \frac{\mathrm{area}(A \cap B)}{\mathrm{area}(A) + \mathrm{area}(B) - \mathrm{area}(A \cap B)}. \tag{4.1}$$

A predicted box is considered a true positive (TP) with respect to a ground-truth box if and only if $\mathrm{IoU} \geq 0.50$; otherwise it contributes a false positive (FP). Each ground-truth box that receives no matching prediction is a false negative (FN).

Per-frame matching. When multiple predicted boxes and multiple ground-truth objects are present in the same frame, a one-to-one assignment is required to avoid matching the same ground-truth object multiple times. Following the standard evaluation protocol used in object detection benchmarks such as Pascal VOC and COCO [30, 31], predictions are matched to ground-truth boxes using a greedy strategy based on confidence scores.

For each class, predicted boxes are first sorted in descending order of confidence. Each prediction is then compared with all ground-truth boxes of the same class in the frame, and the ground-truth box with the highest Intersection over Union (IoU) is identified. If the maximum IoU is greater than or equal to the threshold (0.50 in this work) and the ground-truth box has not already been matched to a higher-confidence prediction, the detection is counted as a true positive (TP) and the ground-truth box is marked as assigned.

Otherwise, the prediction is counted as a false positive (FP). Ground-truth boxes that remain unmatched after processing all predictions are counted as false negatives (FN). This procedure ensures a one-to-one correspondence between predictions and ground-truth instances while penalising duplicate detections of the same object.

Precision–Recall curve. For each class c, all predictions over the entire sequence are sorted by decreasing confidence score. Processing them in this order, each prediction is classified as TP or FP according to the matching procedure above, and the cumulative precision and recall are computed after each prediction:

$$\mathrm{Precision}(k) = \frac{\mathrm{TP}(k)}{\mathrm{TP}(k) + \mathrm{FP}(k)}, \qquad \mathrm{Recall}(k) = \frac{\mathrm{TP}(k)}{N_{\mathrm{GT}}^{c}}, \tag{4.2}$$

where k is the index in the sorted prediction list, $\mathrm{TP}(k)$ and $\mathrm{FP}(k)$ are the cumulative counts up to rank k, and N_{GT}^{c} is the total number of ground-truth instances of class c in the sequence.

Average Precision. The Average Precision (AP) for class c is the area under the precision–recall curve, computed using the 101-point interpolation method standardised by the COCO benchmark [31]:

$$\mathrm{AP}_c = \frac{1}{101} \sum_{r \in \{0, 0.01, \ldots, 1.0\}} \max_{k \,:\, \mathrm{Recall}(k) \geq r} \mathrm{Precision}(k), \tag{4.3}$$

where the maximum is taken over all ranks k such that $\mathrm{Recall}(k) \geq r$, effectively computing the interpolated precision envelope.

Mean Average Precision. The final mAP@50 is the arithmetic mean of AP values over all evaluated classes $\mathcal{C}$:

$$\mathrm{mAP@50} = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \mathrm{AP}_c. \tag{4.4}$$

In this work, $\mathcal{C} = \{\textit{person}, \textit{car}\}$, so mAP@50 is the mean of $\mathrm{AP}_{\mathrm{person}}$ and $\mathrm{AP}_{\mathrm{car}}$. Both AP values are reported separately alongside mAP@50 to enable class-level interpretation, as discussed in Section 5.2.

As noted in Section 4.5, detection accuracy is evaluated exclusively on the NO_MOG2 configuration. When a frame reaches the inference engine, its output is entirely determined by the model weights and the frame content; the upstream pipeline configuration has no influence on the detection results. The mAP@50

figures therefore characterise the inference models themselves and generalise to all pipeline configurations considered in this work.

4.6.3 Time To First Detection

Motivation

The throughput and accuracy metrics characterise the pipeline's behaviour under steady-state conditions: how fast it can process frames, and how accurately it classifies what it sees. Neither of these metrics, however, captures the *responsiveness* of the system to the arrival of a new object in the scene, a property of particular operational relevance in a security surveillance context, where the speed of the first alert may determine the utility of the system.

In the gated pipeline, this responsiveness is structurally constrained. The motion gate forwards a frame to inference only when the foreground pixel count on the 640×360 mask exceeds the empirically calibrated threshold of 500 pixels. An object entering the scene slowly, appearing at distance, or partially occluded, may not immediately generate sufficient foreground activity to trigger the gate. During the interval between the object's appearance and the first frame that crosses the threshold, the object is invisible to the inference engine regardless of the model's detection capability. This introduces a potential detection latency that is absent in the ungated pipeline, where every frame is unconditionally submitted to inference.

The magnitude of this latency depends on two intertwined factors. The first is the MOG2 calibration: a lower threshold increases responsiveness at the cost of passing more frames to inference, while a higher threshold reduces inference load but risks delaying detection of spatially small or slowly moving objects, or even failing to recognise them altogether during prolonged low-activity periods. The second factor is the capability of the downstream inference model: even when a frame passes the gate, a compact or heavily quantized model may fail to produce a confident detection until the object is sufficiently prominent in the frame. The combination of these two factors makes detection latency a composite property of the full pipeline, not attributable to either component in isolation.

To quantify this effect, the **Time To First Detection** (TTFD) metric is introduced. The concept of measuring the temporal gap between an object's first appearance and the system's first response is present in the surveillance and anomaly detection literature, though specific definitions, granularity, and matching criteria vary across works. The formulation adopted here is tailored to the characteristics of this pipeline: it operates at frame granularity, uses IoU-based bounding box matching for per-object identification, and is designed to isolate the gate's contribution by comparing the gated and ungated pipelines at fixed model.

A deliberate choice in this formulation concerns the unit of measurement. The TTFD is expressed in frames rather than wall-clock time. Converting frame counts to milliseconds would require dividing by the pipeline throughput at the time of detection, which is itself a function of the model, the batch size, the number of active streams, and the instantaneous queue state. In a real deployment, additional sources of latency, such as network ingestion, RTSP buffering, and stochastic scheduling, would further decouple the frame-level delay from the experienced response time in ways that cannot be characterised from the available data. A frame-count representation is therefore more informative and more honest: it quantifies the detection gap in terms of the pipeline's own temporal reference, independently of the operating conditions. If an approximation of the wall-clock delay is required for a specific deployment scenario, it can be obtained by dividing the TTFD by the fps/stream figure for the configuration of interest, as reported in Section 5.2; this estimate should however be interpreted as a lower bound on the true response time, since it assumes the pipeline is running at its measured peak throughput without additional sources of latency.

Definition

For a ground-truth object o of class c whose first annotation in the dataset occurs at frame $f_{\mathrm{GT}}(o)$, the TTFD is defined as the number of frames elapsed between $f_{\mathrm{GT}}(o)$ and the first frame at which the pipeline produces a valid detection of o:

$$\mathrm{TTFD}(o) = \max\big(0,\ f_{\mathrm{det}}(o) - f_{\mathrm{GT}}(o)\big), \tag{4.5}$$

where $f_{\mathrm{det}}(o)$ is the smallest frame index $f \geq f_{\mathrm{GT}}(o)$ for which the pipeline produces at least one predicted bounding box of class c that satisfies the matching criterion with the ground-truth box of o at frame f. The lower clamp to zero accounts for the conservative labelling convention of the VIRAT dataset: ground-truth boxes are introduced only once an object is sufficiently visible and well-localised in the frame, whereas the pipeline may produce a valid detection before the corresponding ground-truth annotation appears, yielding a raw value $f_{\mathrm{det}}(o) - f_{\mathrm{GT}}(o) < 0$ that has no meaningful interpretation in this context. The measured TTFD is therefore a lower bound on the true detection latency. If no valid detection of o is found over the entire analysed sequence, the object is marked as *missed* and excluded from the distributional statistics.

Per-Frame Matching

The matching criterion at each candidate frame f is evaluated as follows. Let g_f^o denote the ground-truth box of object o at frame f, and let $P_f^c = \{p_1, \ldots, p_k\}$ denote the set of predicted boxes of class c at frame f. The matching is computed via the Hungarian algorithm applied to the cost matrix $C \in \mathbb{R}^{1\times k}$ with $C_{1j} = 1 - \mathrm{IoU}(g_f^o,\, p_j)$. Since the ground-truth set for a single object at a single frame is always a singleton, the assignment reduces to identifying the predicted box with maximum IoU overlap with g_f^o; a valid detection of o at frame f exists if and only if this maximum exceeds the threshold of 0.50.

An important property of this formulation is that the matching is performed *independently for each ground-truth object*. A predicted bounding box that spatially covers two nearby objects simultaneously, a configuration that arises when pedestrians walk in close proximity, can satisfy the IoU criterion with respect to both objects in the same frame, contributing a valid detection for each. This design choice is intentional: the TTFD asks whether the system has generated any alert in the spatial region of each object, rather than whether it has individually resolved each instance. It does, however, imply that the TTFD is not sensitive

to instance-level localisation quality when objects are heavily overlapping; this limitation is noted in the results discussion.

Aggregation and Comparison

Per-object TTFD values are aggregated by class and reported as minimum, mean, maximum, and number of missed objects. The comparison between the gated (MOG2_CUDA) and ungated (NO_MOG2) configurations at fixed model is expressed as the delta:

$$\Delta\mathrm{TTFD}(o) = \mathrm{TTFD}_{\mathrm{MOG2}}(o) - \mathrm{TTFD}_{\mathrm{NO\_MOG2}}(o), \tag{4.6}$$

which isolates the net latency contribution of the gate. A value $\Delta\mathrm{TTFD}(o) = 0$ indicates that the presence of the gate does not delay the first detection of object o relative to the ungated baseline; a positive value quantifies the additional delay introduced by the gate for that object.

The analysis is conducted on a single stream, as all streams share identical video content and are therefore fully correlated by construction. The implications of this design choice for the interpretation of the results are discussed in Section 5.6.

Chapter 5

Results & Discussion

5.1 Python Baselines vs DeepStream

Table 5.1 reports the throughput of the five Python baseline variants and the DeepStream pipeline on the reference configuration: YOLOv8n, batch size 4, INT8 precision, 4 concurrent streams. Each Python variant is evaluated with and without the MOG2 stage; the DeepStream figures correspond to MOG2_CPU_NODROP, MOG2_CUDA_NODROP, and NO_MOG2 respectively. A detailed description of each pipeline variant is provided in Section 4.4.

Table 5.1: Throughput comparison between Python baseline variants and DeepStream on YOLOv8n, batch 4, INT8, 4 streams.

Pipeline	**With MOG2**	**Without MOG2**
Python v1 – Sequential	9.94	10.90
Python v2 – Parallel	14.26	15.24
Python v3 – Resize on CUDA	14.23	15.22
Python v4 – Resize + LB on CUDA	12.37	13.83
Python v5 – Resize + LB + MOG2 on CUDA	12.83	13.88
DeepStream – MOG2_CPU_NODROP	21.17	–
DeepStream – MOG2_CUDA_NODROP	47.12	–
DeepStream – NO_MOG2	–	61.00

Values in fps per stream.

Several observations emerge from these results. As expected, the presence of the MOG2 stage consistently reduces throughput across all Python variants, though the impact is modest. Among all variants, the purely parallel producer–consumer pipeline (v2) achieves the highest throughput in both configurations. Somewhat counterintuitively, progressively offloading more computation to the GPU does not improve performance: variants v3, v4, and v5 are all slower than v2, suggesting that the overhead introduced by frame copies and CUDA synchronization points outweighs the computational benefit of GPU execution in this context.

The comparison with the DeepStream pipeline is unambiguous. Setting aside the motion stage, the best Python result (v2, without MOG2) is outperformed by NO_MOG2 by more than 4×. Introducing MOG2, the fairest counterpart to the Python variants is MOG2_CPU_NODROP, which executes background subtraction on the CPU as variants v1 through v4 do, yet it already outperforms the best Python result with MOG2 by nearly 1.5×. What is more, while offloading MOG2 to the GPU yields no benefit in Python, DeepStream achieves a 2.2× speedup by moving MOG2 to the CUDA backend over its own CPU counterpart, a gap attributable to its zero-copy NVMM buffer management. Against the best Python result with MOG2, MOG2_CUDA_NODROP exceeds 3×.

The table reports both MOG2_CPU_NODROP and MOG2_CUDA_NODROP as a first indication of the CPU vs. CUDA tradeoff, which is examined in depth in Section 5.3.

While the Python best case nominally exceeds the IEC minimum threshold of 12.5 fps/stream, it does so on the lightest possible configuration (4 streams, smallest model, most aggressive quantization), leaving no margin for heavier models or additional streams. The comparison is therefore not competitive, and all subsequent analysis is conducted exclusively on the DeepStream pipeline.

5.2 Reference Throughput and Detection Accuracy

This section characterises the performance of the DeepStream pipeline in its reference configuration (**NO_MOG2**), evaluated across all 96 inference engine combinations. Throughput and detection accuracy are treated as independent dimensions: throughput depends on model size, precision, and number of streams, while accuracy depends solely on the model weights and is invariant to pipeline configuration, as discussed in Section 4.5.2. Complete results are reported in Tables A.1 and A.2 in Appendix A.1.

The NO_MOG2 configuration serves as the natural starting point for this analysis: by removing the motion gate entirely, it represents the fastest achievable pipeline state and establishes an upper bound on throughput against which all gated configurations can be measured. Understanding this baseline is a prerequisite for quantifying the cost of integrating MOG2 and, subsequently, for characterising the tradeoff between motion density and throughput gain, as examined in Section 5.5. Detection accuracy is evaluated here rather than in the gated sections because it is a property of the inference engine alone: when a frame reaches the PGIE block, its detection result is identical regardless of whether a motion gate is present upstream. The accuracy metrics reported here therefore generalise to all pipeline configurations considered in this work.

5.2.1 Throughput

Several patterns emerge from the data in Table A.1. As expected, throughput decreases monotonically with model size and increases with quantization aggressiveness: INT8 configurations are consistently the fastest, followed by FP16 and FP32. Throughput also decreases as the number of streams increases, since the inference engine must process larger batches at each step.

A notable saturation effect is observed across the lightest model configurations. YOLOv8-N, YOLOv8-S, and YOLO11-N in INT8 precision all converge to approximately 61 fps per stream at batch size 4, despite differing in architecture and parameter count, with YOLO11-S INT8 nearly reaching the same ceiling.

More strikingly, the aggregate pipeline throughput across all streams stabilises at approximately 240 fps regardless of batch size: $61 \times 4 \approx 40 \times 6 \approx 30 \times 8 \approx 24 \times 10 \approx 240$ fps total. This value is in the same order of magnitude as the NVDEC throughput ceiling of the Jetson Orin Nano Super for H.264 content, rated at 450 MPix/s [32], which at 1920×1080 corresponds to approximately 217 fps. Whether this cap originates from the video decoding stage, from pipeline scheduling overhead, or from other factors cannot be determined from the available data alone and would require dedicated per-stage profiling. What the data do establish is the existence of a pipeline-level throughput ceiling at approximately 240 fps in the present use case, beyond which further optimisation of the inference engine yields no benefit.

In terms of deployment viability, 19 out of 24 model–precision combinations meet the IEC threshold of 12.5 fps/stream at batch size 4, decreasing to 16, 13, and 10 at batch sizes 6, 8, and 10 respectively.

FP32 large and medium models fall below threshold at all batch sizes. FP32 small models meet the threshold only at the smallest batch sizes, while FP32 nano models remain above threshold across all configurations. INT8 configurations are the most robust, with all eight combinations remaining above threshold up to batch size 6.

A practically significant observation concerns the comparison between FP16 and FP32 configurations. FP16 engines are consistently faster than their FP32 counterparts across all models and batch sizes, with throughput gains that are substantial for larger models and moderate for lighter ones. As a representative example, YOLO11-L FP16 at batch size 4 reaches 14.58 fps/stream against 6.36 fps/stream for FP32, more than doubling the throughput. The implication for deployment is significant: FP16 brings large models within the IEC threshold in configurations where FP32 would fail entirely. As discussed in the following subsection, this throughput advantage is accompanied by a marginal but consistent gain in detection accuracy, making FP16 the precision of choice for deployment across the full model family.

5.2.2 Detection Accuracy

Detection accuracy is evaluated on the reference video VIRAT_S_000002 using mAP@50 as the primary metric, computed separately for the two annotated classes: *person* and *car*. It is important to note that this evaluation is conducted on a single video sequence and should not be interpreted as a generalised benchmark of model capability across arbitrary scenes. Its purpose is to provide a relative characterisation of the models under the specific conditions of the target deployment environment, enabling a meaningful inter-model comparison that is relevant to the use case at hand.

Although mAP was computed independently for each stream and batch size, no meaningful variation was observed in either dimension. Results are therefore reported as single aggregated values per model and precision.

AP_{car}. The AP for the car class is near-perfect across virtually all configurations, ranging from 0.911 to 0.999. This result is consistent with a manual inspection of the reference video: a single vehicle enters the scene prominently and in the foreground, presenting an unambiguous detection target at every frame in which it appears. While vehicle detection remains a relevant capability in surveillance deployments, the specific conditions of this scene make it an insufficiently discriminating metric for inter-model comparison.

AP_{person}. Unlike the car class, person detection provides a meaningful discriminating metric across model configurations. The reference scene, despite being recorded in daylight, presents non-trivial detection challenges: subjects move between foreground and partially occluded areas, spend portions of the sequence in shadow, appear in partial overlap with structural elements, and in some cases enter the scene at significant distance, subtending only a small number of pixels. These conditions are representative of realistic surveillance deployments and make AP_{person} a meaningful proxy for expected field performance.

For FP32 and FP16, the progression across model sizes is consistent and monotonic for both families. Larger models detect persons more reliably, with

YOLO11-M achieving the highest AP_{person} among all FP32 and FP16 configurations at 0.678. A consistent and noteworthy pattern is that FP16 outperforms FP32 in AP_{person} across all eight models, with gains ranging from 0.003 (YOLO11-N) to 0.012 (YOLO11-L). While these differences are small in absolute terms, their consistency across all models and both families suggests a systematic effect rather than measurement noise. On the Jetson Orin Nano Super, TensorRT selects different execution kernels for FP16 engines, leveraging the Tensor Core units with fused multiply-add operations that differ numerically from the FP32 CUDA core path. The resulting accumulation pattern appears to be marginally more favourable for this task. Combined with the throughput advantage discussed above, this makes FP16 strictly preferable to FP32 for all model sizes: it is simultaneously faster and marginally more accurate, with no countervailing disadvantage.

For INT8, the picture is more complex. YOLO11-L and YOLO11-M retain acceptable person detection with AP_{person} of 0.478 and 0.451 respectively, representing a degradation of 0.166 and 0.227 points respectively relative to their FP16 counterparts. YOLO11-S and YOLO11-N suffer a sharp collapse, reaching 0.177 and 0.049 respectively. A qualitatively similar pattern is observed in the YOLOv8 family, where all INT8 models degrade substantially. The relative robustness of the two families, however, is not uniform across model sizes: YOLO11 INT8 outperforms YOLOv8 INT8 at the larger sizes (L and M), whereas at smaller sizes (S and N) this advantage reverses, suggesting that architectural differences in sensitivity to post-training quantization interact with model scale. A notable exception to the expected size progression within the YOLOv8 INT8 family is YOLOv8-S, whose AP_{person} of 0.264 exceeds both YOLOv8-L (0.225) and YOLOv8-M (0.203). This inversion does not follow the pattern observed in any other precision or family and does not admit a straightforward interpretation; it may reflect statistical variability in the quantization process for this specific architecture, but no definitive conclusion can be drawn from the available data.

5.2.3 Precision Mode Selection

The accuracy data reinforce the case for FP16 as the preferred precision for deployment. INT8 may be acceptable for YOLO11-L and YOLO11-M when throughput constraints are severe, but at the cost of a meaningful reduction in person detection quality. For all other INT8 configurations, the accuracy degradation is too severe to be considered viable for a surveillance application where person detection is the primary objective.

5.3 CUDA vs CPU Backend for Motion Analysis

The results of Section 5.1 already provide a first indication: in Table 5.1, MOG2_CUDA_NODROP outperforms MOG2_CPU_NODROP by a factor of 2.2. However, that comparison is limited to a single model configuration and reflects full inference load. A different picture might emerge for heavier models, where the GPU is the primary bottleneck and offloading MOG2 to the CPU could in principle free resources for inference. This section addresses that question systematically across all 96 configurations and three motion density operating points: 100% (NODROP), 50%, and 5%, which bracket and sample the full range of conditions considered in this work. As discussed in Section 3.3, VPI allows the backend to be switched by a single configuration flag, making the two options architecturally symmetric. Complete results are reported in Tables A.3, A.4, and A.5 in Appendix A.2.

5.3.1 Full Load: NODROP

At full inference load, the picture is mixed. The CPU backend outperforms CUDA in 19 out of 96 configurations, concentrated in FP32 large, medium, and small models and in FP16 large and medium models, primarily at batch sizes 4 and 6, with the sole exception of YOLOv8-L FP16, which also favours the CPU at batch size 8. In these cases, offloading MOG2 to the CPU marginally relieves GPU pressure, yielding gains of up to 18.6%; as a representative example,

YOLO11-M FP32 at B4 reaches 7.77 fps/stream on CPU against 6.77 fps/stream on CUDA. However, almost all configurations in which CPU prevails already fall well below the IEC deployment threshold of 12.5 fps/stream regardless of backend choice, and the absolute fps values are too low to be of practical interest. Across the remaining 77 configurations, CUDA outperforms CPU by a mean factor of 1.40. The average plugin overhead relative to NO MOG2 is 28.8% for the CUDA backend and 41.4% for CPU, confirming that even at full inference load the CUDA backend is more efficient overall.

5.3.2 Intermediate Load: 50% Motion Density

At 50% motion density, the CPU advantage observed at full load disappears entirely: CUDA outperforms CPU across all 96 configurations, with a ratio ranging from 1.05× to 2.81× and a mean of 1.73×. Even the marginal gains seen for large FP32 models at NODROP do not survive a halving of the inference load. This result indicates that the regime in which CPU offloading is beneficial is confined to an extreme corner of the operating space that is simultaneously the least practically relevant.

5.3.3 Minimal Load: 5% Motion Density

At 5% motion density, CUDA dominates across all 96 configurations without exception, with a ratio ranging from 1.67× to 2.98× and a mean of 2.53×. The CPU backend saturates at approximately 22 fps per stream for the lightest models regardless of precision; as a representative example, YOLO11-N INT8 at B4 reaches 22.08 fps/stream on CPU against 61.03 fps/stream on CUDA. At this motion density the GPU is largely idle between inference calls, yet the CPU backend still cannot match CUDA throughput. The bottleneck is therefore not GPU contention alone. Beyond the raw computational speed of MOG2, the CPU backend may also incur additional synchronization and memory coherency costs at the CPU-GPU boundary that are absent in the CUDA path.

5.3.4 Backend Selection

The three operating points bracket the full range of motion densities considered in this work. The CPU backend offers a marginal advantage exclusively for large FP32 models at 100% motion density, in configurations that are already below the IEC deployment threshold. As soon as the motion density drops to 50% or below, CUDA is superior across all 96 configurations without exception, with increasingly large margins at lower motion densities. All subsequent analysis is therefore conducted exclusively on the CUDA backend.

5.4 Plugin Overhead

Section 5.3 reported an average throughput reduction of 28.8% for the CUDA backend relative to NO_MOG2 under full motion load. This section examines that overhead in detail: how it distributes across model sizes, precisions, and batch sizes, and what its practical consequence is in terms of deployment viability. The analysis is confined to MOG2_CUDA_NODROP, which isolates the pure cost of the plugin by removing any throughput benefit from frame dropping. Complete results are reported in Table A.6 in Appendix A.3.

5.4.1 Overhead Magnitude

As noted above, the plugin introduces a mean throughput reduction of 28.8% relative to NO_MOG2 across all 96 configurations, with individual values ranging from 15.3% to 38.1%. The overhead is remarkably uniform across model sizes and precisions: mean values by size range from 27.8% for large models to 30.5% for small models, and by precision from 27.0% for FP32 to 30.6% for FP16. This uniformity is noteworthy. One might expect the overhead to be proportionally smaller for inference-bound configurations, where the inference stage dominates pipeline time and MOG2 represents a smaller fraction of the total cost. The data suggest instead that the interaction between the MOG2 stage and the downstream pipeline scheduling introduces a consistent penalty regardless of the inference

engine weight, an effect that, again, would require per-stage latency profiling to fully characterise.

5.4.2 Impact on Deployment Viability

The overhead has a direct and significant impact on the number of configurations meeting the IEC threshold of 12.5 fps/stream. Of the 58 configurations that meet the threshold under NO_MOG2, 14 fall below it once the plugin is introduced, leaving only 44 viable configurations under MOG2_CUDA_NODROP. The affected configurations span all model families and precisions, and include cases where the margin above threshold in NO_MOG2 was narrow: for instance, YOLO11-L FP16 at batch size 4 drops from 14.58 to 11.79 fps/stream, and YOLO v8-L INT8 at batch size 6 drops from 13.91 to 9.90 fps/stream.

5.4.3 Implications

The plugin overhead quantified here represents the break-even point for the motion gate: any gated configuration must recover at least 28.8% throughput on average through frame dropping before it becomes beneficial relative to the ungated pipeline. The following section examines at what motion density this condition is met.

5.5 Motion Gating Tradeoff

With the break-even cost established in Section 5.4, this section addresses the central question of this work: how pipeline throughput evolves as the fraction of frames carrying motion decreases, and at what point the gated pipeline becomes preferable to the ungated reference. The analysis compares **MOG2_CUDA**$\_k$ for $k \in \{5, 10, 25, 50, 75, 100\}$ against **NO_MOG2**, where $k = 100$ corresponds to the MOG2_CUDA_NODROP configuration introduced in Section 4.5.4. Complete results are reported in Tables A.7 through A.10 in Appendix A.4.

5.5.1 Throughput Progression Across Motion Densities

The crossover with respect to NO_MOG2 occurs between $k = 75$ and $k = 50$. At $k = 75$, no configuration exceeds NO_MOG2. At $k = 50$, 38 out of 96 configurations surpass NO_MOG2, with the gains concentrated in inference-heavy configurations where each suppressed frame yields a disproportionately large benefit. The count grows rapidly at lower densities: 86 out of 96 at $k = 25$, 93 out of 96 at $k = 10$, and all 96 at $k = 5$.

The configurations that do not yet meet NO_MOG2 at $k = 25$ and $k = 10$ are exclusively models already operating at the pipeline throughput ceiling. Their differences from NO_MOG2 are at most 0.21 fps/stream, well within run-to-run measurement noise at saturation, and carry no interpretive significance.

The magnitude of the gains at low motion densities is substantial for inference-bound configurations. At $k = 5$, YOLOv8-L FP32 at batch size 4 reaches 27.63 fps/stream against 4.79 fps/stream in NO_MOG2 (5.77×); YOLO11-L FP32 at batch size 4 reaches 33.81 against 6.36 fps/stream (5.32×). The gains are consistently largest for configurations that are furthest from the throughput ceiling in NO_MOG2, where the GPU is most heavily loaded and frame suppression relieves it the most.

5.5.2 Convergence to the Pipeline Throughput Ceiling

An important structural feature of the tradeoff curve is that all configurations converge toward the same pipeline-level throughput ceiling of approximately 61 fps/stream at batch size 4 (and the corresponding values at larger batch sizes, as established in Section 5.2.1). The motion gate accelerates this convergence for configurations that are below the ceiling in NO_MOG2, but cannot push any configuration beyond it.

The rate at which models approach the ceiling as k decreases depends directly on their inference cost. Lightweight models that are already near the ceiling in NO_MOG2 reach it at relatively high motion densities: for example, YOLOv8-N INT8 and YOLO11-N INT8 at batch size 4 reach the ceiling at $k = 25$ and show

no further gain at lower densities. Models with heavier inference loads approach the ceiling more gradually: at $k = 5$, YOLO11-S FP32 reaches 57.00 fps/stream, YOLO11-M FP16 reaches 57.89 fps/stream, and YOLOv8-S FP32 reaches 57.61 fps/stream. These values are approaching the ceiling but have not reached it within the tested range. Whether a further reduction in k below 5% would bring them to the same ceiling cannot be determined from the available data, but the trend is consistent with the ceiling being a pipeline-level constraint shared by all configurations rather than an inference-level one specific to any model.

The most inference-bound models, large FP32 and FP16 variants at large batch sizes, remain well below the ceiling even at $k = 5$: YOLOv8-L FP32 at batch size 10 reaches only 6.76 fps/stream, reflecting the combined cost of heavy inference and large batch assembly at the maximum number of streams. For these configurations, the gate provides the largest absolute gain but the ceiling is still far off at the lowest density tested.

5.5.3 Deployment Viability and IEC Threshold

As motion density decreases from 100% to 5%, the number of viable configurations progressively recovers and then surpasses the ungated reference. As established in Section 5.4.2, introducing the plugin at full motion load reduces viable configurations from 58 (NO_MOG2) to 44 (MOG2_CUDA_NODROP). The count then grows as k decreases: $k = 75$ reaches 48, $k = 50$ reaches 57, $k = 25$ reaches 72, $k = 10$ reaches 86, and $k = 5$ reaches 90. Of the 52 configurations below the IEC threshold at $k = 100$, 46 recover viability by $k = 5$.

The evolution is particularly consequential for the larger models. YOLO11-L FP16 at batch size 4 drops below threshold once the plugin is introduced ($k = 100$: 11.79 fps/stream), crosses it again at $k = 50$ (15.42 fps/stream, against a NO_MOG2 reference of 14.58 fps/stream), and reaches 53.58 fps/stream at $k = 5$. Models that are below threshold even without the plugin, such as large and medium FP32 configurations, remain below threshold until motion density drops sufficiently; YOLOv8-L FP32 at batch size 4 crosses 12.5 fps/stream between $k = 25$ and $k = 10$.

The six configurations that remain below the IEC threshold at $k = 5$ are exclusively large and medium FP32 models at batch sizes 8 and 10, where inference cost and batch assembly overhead are simultaneously at their maximum.

5.5.4 Interpretation

The results establish a clear and monotonic relationship between motion density and pipeline throughput across the full 96-configuration sweep. As motion density decreases, the gate progressively reduces the inference load, shifting the pipeline bottleneck away from the GPU. For configurations far from the ceiling, this bottleneck shift translates into large and growing throughput gains. For configurations near or at the ceiling, the gate contributes to reaching it faster but cannot produce further improvement once it is reached.

From a deployment perspective, the motion gate is a net benefit in any environment where scenes are not in constant motion, the defining characteristic of the perimeter surveillance scenario targeted by this work. Even at $k = 50$, already a high-activity operating point for a fixed surveillance camera, 38 out of 96 configurations surpass the ungated reference. At lower densities, representative of typical conditions where activity is intermittent, the gate enables model and batch size combinations that would otherwise fail the IEC threshold, most notably the larger FP16 models that offer the best accuracy-throughput tradeoff as identified in Section 5.2.

5.6 Real-Condition Validation: Time To First Detection

5.6.1 Experimental Setup

The Time To First Detection metric is evaluated on the reference sequence `VIRAT_S_000002`, the same clip used throughout this work, using the MOG2_CUDA pipeline with the operationally calibrated foreground pixel threshold of

500. As discussed in Section 4.5.5, this threshold was determined through manual calibration and corresponds approximately to the projected pixel area of a pedestrian entering the scene at moderate distance, excluding low-level GMM noise elsewhere in the frame. It therefore represents a physically grounded lower bound on what constitutes a meaningful motion event in the target deployment scenario.

The TTFD evaluation is conducted on 18 out of the 24 model–precision combinations considered in this work. The MOG2_CUDA pipeline is used with the operationally calibrated threshold of 500 foreground pixels and the periodic safe-pass mechanism active, whereby frames that do not trigger the motion gate are nonetheless forwarded to inference with a probability of 10%, providing a background check against slow or subtle events.

The analysis is conducted on a single stream; as all streams share identical video content, results were verified to be consistent across streams and across batch sizes, with minor frame-level fluctuations attributable to the stochastic interaction between the safe-pass mechanism and the per-stream drop pattern. The results are therefore reported as aggregates. Complete per-object, per-model tables are provided in Appendix A.5.

The six excluded combinations did not produce a complete prediction output within the allotted execution time of the automated test pipeline: their throughput, insufficient at all tested batch sizes to meet the IEC 12.5,fps/stream threshold, prevented the run from terminating before timeout. This represents a limitation of the automated evaluation methodology rather than a deliberate experimental choice, and the results should not be interpreted as a generalist characterisation of TTFD across arbitrary scenes or conditions. Being tied to a single video sequence, a single camera viewpoint, and a specific object composition, they are best understood as a first indicative view of the pipeline's responsiveness under the specific conditions of the reference scenario.

The analysis window spans frames 500 to 3600, following the warmup period, and considers the eight annotated ground-truth objects present in this interval: seven persons and one car.

5.6.2 Observed TTFD on the Operational Pipeline

Table A.11 in Appendix A.5 reports the full per-class TTFD statistics for all 18 evaluated model–precision combinations under MOG2_CUDA. The results reveal two clearly distinct regimes.

For FP16 and FP32 models, TTFD values for the person class are moderate and monotonically dependent on model size. Medium models achieve mean latencies of 5 to 7 frames, with maximum values of 33 frames. Small models show mean latencies around 9 to 10 frames and maxima of 35 to 37 frames. Nano models exhibit the highest latencies, with means of 43 to 50 frames and maxima of 200 to 217 frames. These differences are consistent with the detection sensitivity characteristics of each architecture: larger models produce detections earlier in the object's trajectory, while compact models require the subject to be closer and more prominently framed before producing a detection at all.

For INT8 models, the picture is markedly different. Maximum TTFD values reach into the hundreds and even thousands of frames across multiple configurations, with some objects remaining undetected across a significant portion of the analysis window. This behaviour is consistent with the severe $\mathrm{AP}_{\mathrm{person}}$ person degradation documented for INT8 configurations in Section 5.2: a model that struggles to localise persons accurately under any condition will naturally exhibit large and unpredictable detection latencies. The TTFD figures for INT8 configurations are therefore not interpretable as a characterisation of the gate's behaviour, but rather as a reflection of the models' degraded detection capability. They are reported for completeness but excluded from the conclusions on gate performance.

For the car class, all FP16 and FP32 models record a TTFD of zero. The reasons for this are discussed in Section 5.6.4.

5.6.3 Isolating the Gate Contribution

A non-zero TTFD measured on MOG2_CUDA alone is not sufficient evidence that the gate is introducing detection latency. The same delay would be observed even if every frame reached PGIE, provided the model itself requires a number of

frames before generating a confident detection on a given object. To distinguish between gate-induced latency and model-induced latency, the TTFD is compared at fixed model between MOG2_CUDA and NO_MOG2, according to the delta defined in Equation (4.6).

For all FP16 and FP32 configurations, $\Delta\mathrm{TTFD} = 0$ across all objects and all statistical aggregations. The gate at the 500-pixel threshold introduces no measurable additional detection latency relative to the ungated baseline. This result indicates that the threshold is well-calibrated for the target scenario: at the moment a new object generates sufficient foreground activity to be worth forwarding to inference, the inference engine is already able to detect it, and the gate does not withhold any frame that would have produced an earlier detection. The TTFD values reported for FP16 and FP32 models are therefore entirely attributable to model capability, not to the gating mechanism.

For INT8 configurations, the delta is non-zero for several models and objects, and the interpretation is more complex. These models already exhibit large TTFD values under NO_MOG2, reflecting their degraded detection capability. Under MOG2_CUDA, the delta is further amplified by an interaction between the gate and the model's failure mode: a subject that the model would eventually detect only under specific framing conditions may, at that precise moment, produce insufficient foreground activity to cross the gate threshold, having become partially assimilated into the background model. The safe-pass mechanism mitigates this effect but does not eliminate it, since the one-in-ten passthrough may not coincide with the frames in which the model would have fired. The result is a compounding of two independent failure modes, confirming that INT8 configurations at this quantization level are operationally unsuitable not only from an accuracy standpoint, as established in Section 5.2, but also from a latency standpoint.

5.6.4 Scene Analysis: Warm and Cold Start

The null delta for FP16 and FP32 models, and the zero TTFD for the car class, are explained by the structure of the test scene, which is worth analysing carefully to correctly interpret what the results do and do not establish.

From frame 500 onward, three pedestrians are continuously present and in motion, maintaining the foreground pixel count consistently above the 500-pixel threshold throughout the analysis window. As a consequence, PGIE is invoked on a large majority of frames (motion density approximately 62–63%), and the pipeline remains in an active state for most of the analysis window. The residual static fraction is largely concentrated in the interval immediately preceding the vehicle entry, where no new motion event occurs. Objects 6, 7, 8, and 9, which enter progressively from frame 2544 onward, benefit from this warm pipeline: even if their individual pixel footprint at entry distance would not independently trigger the gate, the gate is already open due to the ongoing motion of the other subjects. Their TTFD therefore reflects only the model's ability to detect them as they enter, not the gate's responsiveness. This is the warm-start condition: the gate contribution to TTFD is structurally zero because the pixel threshold is never the binding constraint.

The sole cold-start event in the scene is the entry of the vehicle at frame 2097. At that moment, no other foreground activity is present and the gate must be triggered by the vehicle alone. The vehicle is immediately detected by all FP16 and FP32 models: the first valid bounding box appears at or before the first annotated ground-truth frame, consistent with the conservative VIRAT labelling convention discussed in Section 4.6.3. The TTFD is clamped to zero, but the raw detection anticipates the annotation, confirming that a sufficiently large object generates enough foreground pixels to cross the 500-pixel threshold as soon as it enters the frame, and that the model fires with high confidence at first occurrence. This is a positive result for the operational calibration at 500 pixels.

The key limitation of these results follows directly from this analysis. The test sequence does not contain any cold-start detection event for the person class: no pedestrian enters the scene in isolation, from distance, without prior motion from other subjects. The null delta for persons therefore does not establish that the gate would introduce no latency in such a scenario; it establishes that, under warm-start conditions, the gate does not add latency. Whether a lone pedestrian entering a cold scene at the boundary of the detection range would experience a

non-zero TTFD at the 500-pixel threshold cannot be determined from these data. This motivates the threshold ablation study described in the following section.

5.6.5 Threshold Ablation: Controlled Cold-Start Simulation

To characterise gate-induced TTFD under controlled cold-start conditions, a targeted ablation experiment is conducted on the vehicle entry event at frame 2097, the only cold-start event available in the reference sequence. The foreground pixel threshold is artificially raised to 1500 pixels, approximately three times the operational value. Under this higher threshold, the vehicle's initial pixel footprint as it enters the frame is insufficient to trigger the gate immediately. This mimics the behaviour expected for a smaller or more distant object under the operational 500-pixel threshold: the object is present and visible, but its foreground contribution does not yet meet the gate criterion. The 1500-pixel threshold is not proposed as an operational setting; it serves as a controlled perturbation that introduces a measurable gate-induced latency on a known cold-start event, allowing the gate's contribution and the safe-pass mechanism's mitigating effect to be isolated and quantified.

Four configurations are compared, all using YOLO11-M FP16, the highest-performing model in AP_{car}, so that the model contribution to TTFD is minimised and any residual latency is more directly attributable to the gate behaviour:

- **NO_MOG2**: ungated baseline, all frames reach PGIE.
- **MOG2_500**: operational threshold, safe-pass active.
- **MOG2_1500**: elevated threshold, safe-pass active.
- **MOG2_1500_nosafepass**: elevated threshold, safe-pass mechanism disabled.

The TTFD results are reported in Table 5.2.

The results confirm the expected behaviour. NO_MOG2 and MOG2_500 detect the vehicle at frame 2097 with zero TTFD, consistent with the main evaluation. Under MOG2_1500 with safe-pass active, the TTFD rises to 11 frames:

Table 5.2: TTFD on the vehicle cold-start event under varying threshold configurations.

Configuration	Detection Frame	TTFD (frames)
NO_MOG2	2097	0
MOG2_500	2097	0
MOG2_1500	2108	11
MOG2_1500_nosafepass	2203	106

Model: YOLO11-M FP16. TTFD expressed in frames.

the gate suppresses the initial frames in which the vehicle's foreground area falls below 1500 pixels, but the safe-pass mechanism forwards one of the first ten frames regardless, enabling detection shortly after entry. Under MOG2_1500 with safe-pass disabled, the TTFD increases to 106 frames: without the periodic passthrough, the pipeline must wait until the vehicle has advanced far enough into the scene to independently generate 1500 foreground pixels. The 95-frame difference between the two elevated-threshold configurations directly quantifies the operational value of the safe-pass mechanism: in this cold-start scenario, it reduces gate-induced detection latency by a factor of approximately ten.

The same four configurations were then evaluated using YOLO11-S INT8, the model with the lowest AP_{car} among the tested configurations. In this case, the TTFD is 132 frames across all four configurations, including NO_MOG2. The gate threshold and the safe-pass mechanism have no measurable effect: the model alone determines the detection frame, requiring the vehicle to be sufficiently large and centred before generating a bounding box that satisfies the IoU criterion. The gate is entirely irrelevant in this regime because the model's capability is the binding constraint before the gate ever becomes one. Notably, 132 frames corresponds precisely to the maximum car TTFD observed for INT8 configurations in the main evaluation of Section 5.6.2, confirming the consistency of the measurement across experimental conditions.

This result provides a clean empirical illustration of the two-regime structure discussed in Section 4.6.3: when the gate is the binding constraint, raising the

threshold increases TTFD and the safe-pass mechanism is the primary mitigating factor; when the model is the binding constraint, threshold and safe-pass are irrelevant and TTFD is determined entirely by inference quality.

Figures 5.1 and 5.2 show the first valid detection frame for each of the four configurations, for YOLO11-M FP16 and YOLO11-S INT8 respectively. Ground-truth and predicted bounding boxes are overlaid on the full-resolution video frame. The two carousels together illustrate the two-regime behaviour: for FP16 (Figure 5.1), the vehicle's position at detection advances progressively across the four panels, confirming that the gate is the binding constraint; for INT8 (Figure 5.2), all four configurations share the same detection frame, confirming that the model alone is the binding constraint regardless of gate settings.

5.6.6 Discussion

The TTFD analysis yields a coherent and internally consistent picture of the pipeline's responsiveness. The central question is whether the gate itself introduces detection latency, over and above what the inference engine alone would produce. Comparing MOG2_CUDA against NO_MOG2, the answer for FP16 and FP32 models is unambiguous: the delta is zero across all evaluated configurations. At the operational threshold of 500 pixels, the gate activates as soon as a relevant motion event occurs, and the model detects the object at the first forwarded frame. The measured TTFD values for these models reflect exclusively the detection capability of the inference engine.

The reference sequence, however, does not contain a genuine cold-start person detection event: all pedestrian objects are already present and in motion when the analysis window opens. This prevents a direct evaluation of gate-induced latency under the most critical condition for a perimeter surveillance system. The threshold ablation addresses this gap by simulating a cold-start scenario on the vehicle entry event. For YOLO11-M FP16, the results confirm that gate-induced latency does exist when the threshold exceeds the object's entry-level pixel footprint, and that the safe-pass mechanism is a practically effective mitigation, reducing TTFD from 106 to 11 frames under a threshold three times the operational value. For

Figure 5.1: Threshold ablation – YOLO11-M FP16. First valid detection frame for each configuration (top-left: NO_MOG2, TTFD = 0; top-right: MOG2_500, TTFD = 0; bottom-left: MOG2_1500, TTFD = 11; bottom-right: MOG2_1500 without safe-pass, TTFD = 106). Ground-truth boxes in green, predicted boxes in blue.

Figure 5.2: Threshold ablation – YOLO11-S INT8. All four configurations produce the same detection frame (frame 2229, TTFD = 132), confirming that the model is the binding constraint regardless of gate settings. Ground-truth boxes in green, predicted boxes in blue.

YOLO11-S INT8, the ablation produces no variation across configurations: the TTFD is identical regardless of gate threshold or safe-pass setting, confirming that the model is the binding constraint and the gate is entirely irrelevant in this regime. Combined with the accuracy degradation documented in Section 5.2, this establishes that INT8 is unsuitable for the target deployment scenario across all relevant performance dimensions.

The threshold ablation provides a partial substitute by reproducing the cold-start condition on a surrogate object, but a dedicated evaluation on a sequence containing isolated pedestrian entries would be a natural extension, given that the person class exhibits different detection sensitivity characteristics as documented in Section 5.2.

5.7 Reproducibility and Experimental Limitations

5.7.1 Reproducibility

All experimental runs in this work are conducted on the same fixed input sequence, `VIRAT_S_000002`, loaded from disk as a local H.264-encoded file. This choice, motivated in Section 4.6.1, ensures full determinism of the input signal across all configurations and runs.

The MOG2 background subtractor, when active, is configured with a foreground pixel threshold of 500 on the 640×360 mask, as defined in Section 4.5.5, and with a learning rate of `0.01`, which corresponds to the default value provided by the VPI library. For configurations with a fixed motion density ($k \in \{5, 10, 25, 50, 75\}$), the frame drop decision is governed by a deterministic pseudo-random generator with a fixed global seed of `180226`, ensuring that the drop pattern is identical across repeated runs of the same configuration.

It should be noted that, over a sequence of 3600 frames, the realized motion density may not converge exactly to the target value k. The pseudo-random mechanism produces the correct density in expectation, but the finite length of the sequence introduces residual variance; the actual fraction of forwarded frames is

therefore an approximation of the nominal target.

All results are reported from single runs. A statistically rigorous characterisation would require multiple repeated runs per configuration and reporting of mean and standard deviation. This was not carried out for practical reasons given the large number of configurations evaluated. However, the platform configuration guidelines described in Section 4.1 were followed consistently, and empirical spot-checks conducted across several configurations at different times showed run-to-run variation in the order of a few fps on the aggregate pipeline throughput (summed across all streams), confirming that the single-run figures are representative and stable.

5.7.2 Warmup Period

All measurements exclude the first 500 frames of each run, defined as the warmup period. During this interval, the pipeline executes normally, including the MOG2 algorithm where applicable, but no metrics are recorded. The dual purpose of this exclusion is to avoid capturing pipeline startup transients in the throughput figures, and to allow the MOG2 background model to reach a stable state before the evaluation window begins. Empirically, the model produces a well-formed foreground mask within approximately 100–200 frames under the learning rate and scene conditions used here; the 500-frame warmup is therefore conservative, chosen to ensure that no residual model instability affects the results.

5.7.3 Stream Correlation and Scope of the Throughput Evaluation

A fundamental constraint of this experimental setup is that the same video file is replicated across all N input channels. In a real multi-camera installation, each camera observes an independent portion of the scene, producing decorrelated motion patterns and therefore decorrelated frame drop decisions across streams. This property cannot be replicated with a single shared source.

For the simulated motion density configurations (MOG2_CUDA_k), this lim-

itation is mitigated by the pseudo-random drop mechanism. Since each plugin instance draws from the same random sequence but runs on a separate thread, their calls interleave non-deterministically, producing drop patterns that are decorrelated across streams over the course of the run. Although this is not a perfect model of independent cameras, it distributes the coincidences between stream activity across the full 3600-frame window, and the resulting throughput figures are a reasonable approximation of what would be observed in a real deployment where different cameras exhibit intermittent, statistically independent motion. The throughput results reported in Sections 5.3 through 5.5 are therefore considered valid for comparative purposes across models and operating points.

For the real-condition validation configuration (MOG2_CUDA), the situation is qualitatively different. Here the drop decision is driven by the actual foreground pixel count, which is identical for all streams at every frame. All channels therefore pass or drop frames simultaneously, producing a fully correlated drop pattern. This correlation fundamentally alters the behaviour of the multiplexer and the batch assembly process, making the aggregate throughput unrepresentative of operational conditions. For this reason, the MOG2_CUDA configuration is not characterised in terms of throughput: the correlated drop pattern would introduce a systematic distortion whose magnitude is scene-dependent and not generalisable. Instead, this configuration is used exclusively for the TTFD evaluation, where per-object detection latency is the quantity of interest and stream correlation does not affect the validity of the measurement.

5.7.4 The Multiplexer Timeout and Its Implications

A non-trivial source of variability in the throughput measurements, and a significant open question for real deployments, is the `batched-push-timeout` parameter of `nvstreammux`. As described in Section 3.2, the muxer assembles batches of fixed size equal to the number of parallel streams, waiting up to a configurable timeout for a frame from each stream before forwarding an incomplete batch to inference.

In a real deployment with live RTSP streams, NVIDIA's guidance [33] rec-

ommends setting the timeout to $1.000.000\,\mu\text{s}/\text{fps}_{\max}$, so that the muxer waits at most one frame interval before dispatching. This prescription, however, is circular in the present context: the objective of this work is precisely to determine the maximum achievable fps, which is not known in advance and depends on the configuration under test. The timeout therefore cannot be optimally tuned per configuration without prior knowledge of the result.

All runs in this work use a fixed timeout of $50.000\,\mu\text{s}$. This value was chosen as a practical compromise given the circular dependency described above: without prior knowledge of the achievable fps, no principled optimal value can be derived.

The relationship between timeout value and throughput is not monotonic and depends on the specific operating conditions. A timeout that is too short causes the muxer to dispatch batches before most streams have contributed a frame, resulting in largely incomplete batches and poor GPU utilisation. A timeout that is too long introduces unnecessary waiting even when all available frames have already arrived, adding idle time between dispatches. The optimal value sits between these extremes and depends on the frame arrival rate of each stream, which is itself a function of the motion density of the scene and the behaviour of the gate. Empirical tests with alternative timeout values confirmed this sensitivity, with aggregate throughput varying meaningfully across the tested range.

More broadly, this reveals a structural limitation of the fixed-timeout batching policy. In a scenario where only one stream is active and the others are idle, the muxer will wait the full timeout duration before dispatching a batch containing a single valid frame out of N slots, wasting the remaining inference capacity. A smarter batching strategy, aware of the instantaneous activity pattern across streams, could adapt dynamically, dispatching as soon as a useful batch can be assembled rather than waiting for a fixed interval. This remains a direction for future work, discussed further in Chapter 6.

Chapter 6

Conclusion & Future Work

This thesis set out to answer a concrete question: whether a real-time, multi-stream video intrusion detection system, deployed entirely on embedded edge hardware, can deliver reliable detection performance within the cost and power constraints of a self-contained local installation, without any dependency on external infrastructure. The experimental results presented in Chapter 5 provide a grounded answer to that question, and this chapter draws together what those results establish, what they leave open, and where the most consequential directions for future work lie.

6.1 DeepStream as the Reference Framework

The comparison against five Python baseline variants, evaluated on the lightest available inference configuration to maximise the baselines' relative advantage, produced an unambiguous outcome. On the ungated pipeline, the best Python result is outperformed by more than $4\times$ in throughput; with the motion gate active, the gap remains above $3\times$.

A significant part of this advantage stems from DeepStream's native access to the dedicated hardware accelerators available on the Jetson platform: video decoding through NVDEC, format conversion and scaling through VIC, and memory management through the NVMM unified domain. Replicating this level of hard-

ware integration in a Python-based pipeline would be considerably more complex, and in some cases not practically achievable.

For the class of applications addressed here, multi-stream real-time inference on constrained hardware, the data presented in this work support DeepStream as the framework of choice. Navigating its adoption in a non-standard pipeline configuration is not straightforward, and the implementation details documented throughout this work are intended to ease that path for analogous development efforts.

6.2 The Motion Gate and the CUDA Backend

One of the core contributions of this work is the motion gate plugin: a custom GStreamer element integrating VPI-accelerated MOG2 background subtraction directly into the DeepStream pipeline, operating on NVMM-backed buffers via zero-copy wrapping, with independent background models per stream and an inline frame drop mechanism.

A design question whose answer was not obvious a priori concerns the execution backend for MOG2. The initial reasoning favoured the CPU: in a DeepStream pipeline, the CPU is largely idle during inference, and offloading background subtraction there would in principle leave the GPU fully dedicated to inference. The experimental data contradict this reasoning at every tested operating point below full motion load. The CUDA backend outperforms the CPU with a mean factor of $1.73\times$ at $k = 50\%$ and $2.53\times$ at $k = 5\%$. The advantage grows precisely as motion density decreases, that is, as the deployment conditions approach those of the target scenario. The explanation lies in the interaction between the unified memory architecture and the synchronisation costs at the CPU–GPU boundary: the CUDA path, operating entirely within the NVMM domain, avoids those costs entirely, and that advantage is most visible when the inference engine is not saturating the GPU. All subsequent analysis was therefore conducted on the CUDA backend.

6.3 Operating Conditions and Gate Calibration

The gate introduces a mean throughput overhead of 28.8% relative to the ungated pipeline at full motion load. This cost is recovered as motion density decreases, with the crossover occurring between $k = 75\%$ and $k = 50\%$. Motion densities above 50% are, in practice, symptomatic of a deployment issue rather than a normal operating condition: they indicate suboptimal camera placement, an insufficiently calibrated MOG2 threshold, or the presence of background dynamics too complex for the GMM model to absorb, such as dense foliage moving in the wind, whose irregular and persistent motion may exceed the adaptive capacity of the background model and require dedicated outside-ROI masking to be excluded from the analysis.

In the conditions that characterise a well-configured perimeter installation, motion density could be expected to reside well below 25%: even at that upper bound, one frame in four, on every stream, carries detected motion, which already represents a conservative and rather demanding operating point for a fixed surveillance camera scenario. Within this range, the gate consistently recovers its overhead and delivers substantial throughput gains across the full model space.

This behaviour, however, depends on calibration. The threshold governing the gate must be set carefully: too tight, and the gate becomes the bottleneck rather than the inference engine, forwarding so few frames that relevant events are missed or delayed; too loose, and the filtering benefit is lost, with the inference load approaching that of the ungated pipeline.

That said, a degree of imprecision in the gate is not only tolerable but, in a two-stage architecture, arguably desirable: the role of the first stage is to produce triggers, including on ambiguous frames, while the inference model is precisely there to confirm or discard them. The inverse does not hold. The downstream model is the final decision stage before a security alert is triggered, and its output admits no equivalent margin. The two components must therefore be calibrated in relation to each other, with the gate tuned for sensitivity and the model selected for precision.

On this point, the periodic safe-pass mechanism is worth noting. By uncondi-

tionally forwarding one frame in ten classified as background, the pipeline maintains a continuous, low-cost inference activity during idle periods, functionally equivalent to operating at $k = 10\%$, at negligible additional cost. As soon as motion is detected on a stream, the gate opens and the full burst of motion frames is forwarded; the safe-pass simply ceases to be the active path. The mechanism works in symbiosis with the gate rather than against it, and its operational cost is entirely justified by the detection robustness it provides.

6.4 Detection Accuracy and Precision Mode

The Time To First Detection analysis confirms that, for FP16 and FP32 models at the operational threshold, the gate introduces no measurable additional detection latency relative to the ungated baseline. The latency values observed for these models reflect exclusively the detection capability of the downstream model, a result that is not trivially guaranteed: it follows from the threshold being well-calibrated for the specific scene, ensuring that no frame carrying a detectable event is suppressed before reaching the inference engine.

The interaction between gate threshold and model sensitivity was further characterised through a targeted ablation on the vehicle cold-start event, which isolated the respective contributions of the two components and quantified the mitigating effect of the safe-pass mechanism.

INT8 configurations produced a markedly different outcome. The degradation is selective in a way that is informative: AP_{car} remains high across most configurations, while AP_{person} degrades severely and without a consistent pattern across model sizes and families, suggesting that architectural differences in sensitivity to post-training quantization interact with model scale in ways that are not uniform. Whether this pattern reflects an intrinsic limitation of INT8 quantization for small, partially occluded objects, or a consequence of the calibration dataset not adequately representing the full distribution of pedestrian appearances in the deployment scene, cannot be determined from the available data alone. What the results do establish is that INT8 configurations, as calibrated in this work, are

not viable for a surveillance application where person detection is the primary objective, and that further investigation into the calibration process would be a necessary prerequisite before reconsidering this precision mode.

FP16 emerges as the precision of choice. Halving the numerical precision reduces memory bandwidth and computational cost, making FP16 engines consistently faster than their FP32 counterparts across all tested models and batch sizes. More interestingly, the same engines also produce marginally higher AP_{person} across every model and both families, a result whose consistency suggests a systematic numerical effect rather than measurement noise. There is no trade-off to manage: FP16 is both faster and more accurate than FP32, and should be the default precision for this class of deployment.

6.5 Model and Configuration Selection

With the operating conditions and precision mode established, the question of model selection follows a natural progression of constraints.

The first constraint is hardware. The NVDEC engine on the Jetson Orin Nano Super is rated for a maximum of 11 concurrent H.265 streams at 1920×1080 resolution and 30 fps. Beyond that limit, a single device is insufficient regardless of the inference configuration, and a second unit would be required. Within that bound, the pipeline has been evaluated up to batch size 10, with margin remaining; the practical stream count for a single-device installation is therefore constrained by NVDEC before it is constrained by inference.

The second constraint is motion. As noted above, all of the throughput gains documented in this work are conditional on the scene not being in constant motion, which is the defining characteristic of the scenario this work targets.

Within the YOLO11 and YOLOv8 families, a consistent pattern emerges across model sizes. On the M, S, and N sizes, YOLO11 simultaneously outperforms YOLOv8 on both AP_{person} and throughput, with accuracy advantages of $+0.060$, $+0.088$, and $+0.080$ respectively, and throughput advantages that grow as motion density decreases. The L size is the exception: YOLOv8-L achieves a $+0.022$-

point advantage on AP_{person}, while YOLO11-L is consistently faster at every operating point.

Given these results, **YOLO11-M FP16** seems to be the most balanced choice for the target scenario. It achieves the highest AP_{person} of the entire comparison at 0.678, above even the larger L models, and its throughput remains above the IEC 12.5 fps/stream threshold up to batch 8 at $k = 25\%$, and across all batch sizes at $k \leq 10\%$. At $k = 5\%$, it reaches 21.4 fps/stream even at batch 10, a figure that is not only above the deployment threshold but can even approach real-time operation on a 10-stream installation.

When throughput is the binding constraint, for instance when a larger number of streams must be supported or when motion conditions are less favourable, **YOLO11-S FP16** is the appropriate alternative. Its AP_{person} of 0.623 remains competitive, and it meets the IEC deployment threshold across all tested batch sizes even at $k = 50\%$, providing consistent headroom in configurations where the M model begins to show margin pressure.

6.6 Future Work

The motion gate plugin as implemented is functionally sound, and its net benefit in realistic scene conditions has been demonstrated. However, the 28.8% overhead it introduces is higher than the equivalent cost observed in the Python baselines, despite the use of VPI acceleration and zero-copy buffer access. This discrepancy suggests that the interaction between the plugin and the DeepStream pipeline scheduling introduces latency that has not been characterised in this work. A dedicated profiling study of the GPU execution context within the `gst-dsexample` plugin would be the natural starting point, and may reveal margins for improvement without altering the functional behaviour of the plugin.

The more structurally significant open problem is the `nvstreammux` element. The multiplexer operates under a fixed-timeout batching policy that is designed for standard pipelines where all streams deliver frames at a uniform rate. The motion gate breaks this assumption: when only a subset of streams are active,

the muxer waits the full timeout before dispatching an incomplete batch, wasting inference capacity in a way that is inherent to its design and cannot be corrected by calibration. The standard element was never intended for a pipeline with upstream frame filtering, and the appropriate solution is the same one adopted for the motion gate itself: a custom plugin, tailored to the specific behaviour of a prefiltered pipeline, that dispatches as soon as a useful batch can be assembled rather than waiting for a fixed interval, and that is capable of filling a batch with frames from a single active stream in order to drain the motion burst accumulated in the upstream queue as quickly as possible. In a real deployment, where different cameras observe independent portions of the scene and produce decorrelated motion patterns, the muxer behaviour is the component most likely to determine whether the pipeline achieves its potential throughput or consistently underperforms it.

Finally, further directions that this work opens, without having explored them, include the definition of inside-ROI regions to focus motion analysis exclusively on areas of interest; multi-stage inference cascades in which the output of the motion gate feeds specialised downstream models for tasks such as licence plate recognition or face detection, applied exclusively to the objects and frames selected by the first stage; and context-aware alert generation based on complex temporal and multi-event rules, which would allow the system to reason beyond individual detections and trigger notifications only when a meaningful behavioural pattern is identified. The throughput headroom that the gate creates is precisely what makes these extensions feasible, and the architecture presented in this thesis is a foundation on which they can be built.

Bibliography

[1] Vassilios Tsakanikas, Tasos Dagiuklas, *"Video Surveillance Systems: Current status and future trends"*. Computers Electrical Engineering 70 (2018): 736-753.

[2] K. He, X. Zhang, S. Ren and J. Sun, *"Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"* 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, Chile, 2015, pp. 1026-1034, doi: 10.1109/ICCV.2015.123.

[3] Syed Muhammad Raza, Syed Murtaza Hussain Abidi, Md Masuduzzaman, Soo Young Shin, *"Lightweight deep learning for visual perception: A survey of models, compression strategies, and edge deployment challenges"* Neurocomputing, Volume 656, 2025, 131357, ISSN 0925-2312. https://www.sciencedirect.com/science/article/pii/S0925231225020296

[4] Linxiao Gong, Hao Yang, Gaoyun Fang, Bobo Ju, Juncen Guo, Xiaoguang Zhu, Xiping Hu, Yan Wang, Peng Sun, Azzedine Boukerche, *"A Survey on Video Analytics in Cloud-Edge-Terminal Collaborative Systems"*. 2025 - https://arxiv.org/abs/2502.06581 https://doi.org/10.48550/arXiv.2502.06581

[5] Obermaier, Johannes, Martin Hutle, *"Analyzing the security and privacy of cloud-based video surveillance systems"* Proceedings of the 2nd ACM international workshop on IoT privacy, trust, and security. 2016.

[6] Ravindran AA, *"Internet-of-Things Edge Computing Systems for Streaming Video Analytics: Trails Behind and the Paths Ahead"* IoT. 2023; 4(4):486-513. https://doi.org/10.3390/iot4040021

[7] The EDPS video-surveillance guidelines https://www.edps.europa.eu/sites/default/files/publication/10-03-17_video-surveillance_guidelines_en.pdf

[8] NVIDIA Jetson Orin Nano™ Super https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/

[9] NVIDIA JetPack™ https://developer.nvidia.com/embedded/jetpack

[10] NVIDIA® CUDA® https://developer.nvidia.com/cuda/toolkit

[11] NVIDIA® CUDA® Deep Neural Network library (cuDNN) https://developer.nvidia.com/cudnn

[12] NVIDIA® TensorRT™ https://developer.nvidia.com/tensorrt

[13] NVIDIA® Vision Programming Interface (VPI) https://docs.nvidia.com/vpi/

[14] NVIDIA DeepStream SDK https://developer.nvidia.com/deepstream-sdk

[15] GStreamer - open source multimedia framework https://gstreamer.freedesktop.org

[16] N. Otsu, *"A Threshold Selection Method from Gray-Level Histograms"* in IEEE Transactions on Systems, Man, and Cybernetics, vol. 9, no. 1, pp. 62-66, Jan. 1979, doi: 10.1109/TSMC.1979.4310076.

[17] Stauffer C, Grimson W., *"Adaptive background mixture models for real-time tracking"* in Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149). IEEE Comput. Soc. Part Vol. 2, 1999.

[18] Zoran Zivkovic, *"Improved Adaptive Gaussian Mixture Model for Background Subtraction"*. (2004) Proceedings - International Conference on Pattern Recognition. 2. 28 - 31 Vol.2. 10.1109/ICPR.2004.1333992.

[19] Zoran Zivkovic, Ferdinand van der Heijden, *"Efficient adaptive density estimation per image pixel for the task of background subtraction"* Pattern Recognition Letters, Volume 27, Issue 7, 2006, Pages 773-780, ISSN 0167-8655, https://doi.org/10.1016/j.patrec.2005.11.005.

[20] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, Conference: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), DOI:10.1109/CVPR.2016.91

[21] Priyanto Hidayatullah, Nurjannah Syakrani, Muhammad Rizqi Sholahuddin, Trisna Gelar, Refdinal Tubagus, *"YOLOv8 to YOLO11: A Comprehensive Architecture In-depth Comparative Review"* 2025 - `https://arxiv.org/abs/2501.13400`

[22] NVIDIA Accelerated Decode on GStreamer: Video Decode Examples `https://docs.nvidia.com/jetson/archives/r38.4/DeveloperGuide/SD/Multimedia/AcceleratedGstreamer.html`

[23] Sangmin Oh, Anthony Hoogs, Amitha Perera, Naresh Cuntoor, Chia-Chih Chen, Jong Taek Lee, Saurajit Mukherjee, J.K. Aggarwal, Hyungtae Lee, Larry Davis, Eran Swears, Xiaoyang Wang, Qiang Ji, Kishore Reddy, Mubarak Shah, Carl Vondrick, Hamed Pirsiavash, Deva Ramanan, Jenny Yuen, Antonio Torralba, Bi Song, Anesco Fong, Amit Roy-Chowdhury, Mita

Desai, *"A Large-scale Benchmark Dataset for Event Recognition in Surveillance Video"*, Proceedings of IEEE Comptuer Vision and Pattern Recognition (CVPR), 2011. https://viratdata.org

[24] NVIDIA® VIC rescale algorithm performance tables https://docs.nvidia.com/vpi/algo_rescale.html

[25] DeepStream application benchmark - system configuration https://docs.nvidia.com/metropolis/deepstream/5.0/dev-guide/index.html#page/DeepStream_Development_Guide/deepstream_performance.html

[26] NVIDIA Jetson Orin Nano MAXN mode setup https://developer.ridgerun.com/wiki/index.php/Exploring_NVIDIA_Jetson_Orin_Nano_Super_Mode_performance_using_Generative_AI

[27] Command-Line Programs: trtexec https://docs.nvidia.com/deeplearning/tensorrt/latest/reference/command-line-programs.html

[28] NVIDIA Polygraphy https://docs.nvidia.com/deeplearning/tensorrt/latest/_static/polygraphy/index.html

[29] *BS EN IEC 62676-4:2025 – Video surveillance systems for use in security applications – Part 4: Application guidelines*. IEC/CENELEC, November 2025.

[30] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. *The Pascal Visual Object Classes (VOC) Challenge.* uInt. J. Comput. Vision 88, 2 (June 2010), 303–338. https://doi.org/10.1007/s11263-009-0275-4

[31] Tsung-Yi Lin and Michael Maire and Serge Belongie and Lubomir Bourdev and Ross Girshick and James Hays and Pietro Perona and Deva Ramanan and

C. Lawrence Zitnick and Piotr Dollár. *Microsoft COCO: Common Objects in Context* European Conference on Computer Vision (ECCV), 2014 https://arxiv.org/abs/1405.0312

[32] Jetson Orin Nano Series Modules Data Sheet https://developer.nvidia.com/embedded/downloads#?search=Data%20Sheet

[33] DeepStream SDK FAQ - How to set parameters reasonably to improve the efficiency of nvstreammux in live mode https://forums.developer.nvidia.com/t/deepstream-sdk-faq/80236/34

Appendix A

Supplementary Results

A.1 Reference Configuration: Complete Results

Table A.1 reports the full throughput of the NO_MOG2 pipeline across all 96 configurations. Table A.2 reports the detection accuracy for all model and precision combinations.

Table A.1: NO_MOG2 throughput (fps/stream) across all 96 configurations.

Model	Precision	B4	B6	B8	B10
YOLO11-L	FP32	6.36	4.34	3.57	2.92
YOLO11-M	FP32	8.30	5.53	4.15	3.68
YOLO11-S	FP32	**18.20**	12.28	9.06	7.29
YOLO11-N	FP32	**33.89**	**22.32**	**17.98**	**14.31**
YOLO11-L	FP16	**14.58**	10.34	7.93	6.26
YOLO11-M	FP16	**18.42**	**12.64**	10.08	7.18
YOLO11-S	FP16	**37.37**	**22.74**	**19.55**	**15.88**
YOLO11-N	FP16	**56.30**	**36.78**	**27.86**	**24.39**
YOLO11-L	INT8	**23.57**	**16.00**	**12.65**	10.15
YOLO11-M	INT8	**29.53**	**20.23**	**15.17**	**12.46**
YOLO11-S	INT8	**54.64**	**35.73**	**28.90**	**24.39**
YOLO11-N	INT8	**60.97**	**40.67**	**30.48**	**24.41**
YOLOv8-L	FP32	4.79	3.66	2.74	2.17
YOLOv8-M	FP32	8.72	5.91	4.46	3.58
YOLOv8-S	FP32	**18.29**	**12.65**	9.95	7.73
YOLOv8-N	FP32	**34.25**	**23.80**	**18.32**	**14.73**
YOLOv8-L	FP16	11.21	7.63	5.85	4.61
YOLOv8-M	FP16	**17.24**	**12.36**	9.34	7.49
YOLOv8-S	FP16	**35.99**	**25.85**	**19.20**	**15.54**
YOLOv8-N	FP16	**59.03**	**36.42**	**30.49**	**24.38**
YOLOv8-L	INT8	**20.46**	**13.91**	11.11	8.78
YOLOv8-M	INT8	**28.62**	**18.68**	**14.53**	**12.26**
YOLOv8-S	INT8	**60.99**	**40.63**	**30.50**	**24.39**
YOLOv8-N	INT8	**61.00**	**40.67**	**30.51**	**24.41**

Note: Values meeting the IEC threshold of 12.5 fps/stream are shown in **bold**.

Table A.2: Detection accuracy (mAP@50) on the reference video VIRAT_S_000002 across all model and precision combinations.

Model	Precision	AP_{person}	AP_{car}	mAP@50
YOLO11-L	FP32	0.6315	0.9985	0.8150
YOLO11-L	FP16	0.6439	0.9985	0.8212
YOLO11-L	INT8	0.4779	0.9993	0.7386
YOLO11-M	FP32	0.6658	0.9993	0.8325
YOLO11-M	FP16	0.6775	0.9993	0.8384
YOLO11-M	INT8	0.4509	0.9991	0.7250
YOLO11-S	FP32	0.6124	0.9990	0.8057
YOLO11-S	FP16	0.6227	0.9990	0.8108
YOLO11-S	INT8	0.1767	0.9109	0.5438
YOLO11-N	FP32	0.4850	0.9992	0.7420
YOLO11-N	FP16	0.4882	0.9992	0.7437
YOLO11-N	INT8	0.0493	0.9116	0.4804
YOLOv8-L	FP32	0.6572	0.9992	0.8282
YOLOv8-L	FP16	0.6657	0.9992	0.8325
YOLOv8-L	INT8	0.2246	0.9993	0.6120
YOLOv8-M	FP32	0.6105	0.9992	0.8048
YOLOv8-M	FP16	0.6174	0.9992	0.8083
YOLOv8-M	INT8	0.2025	0.9992	0.6009
YOLOv8-S	FP32	0.5297	0.9993	0.7645
YOLOv8-S	FP16	0.5350	0.9993	0.7672
YOLOv8-S	INT8	0.2636	0.9992	0.6314
YOLOv8-N	FP32	0.4020	0.9993	0.7006
YOLOv8-N	FP16	0.4079	0.9993	0.7036
YOLOv8-N	INT8	0.1252	0.9448	0.5350

A.2 CUDA vs CPU Backend: Complete Results

Tables A.3, A.4, and A.5 report the full throughput comparison between the CUDA and CPU backends across all 96 model configurations at motion densities of 100%, 50%, and 5% respectively. For each cell, the higher value between the two backends is shown in bold.

Table A.3: Throughput (fps/stream) for MOG2_CUDA_NODROP and MOG2_CPU_NODROP across all 96 configurations.

		B4		B6		B8		B10	
Model	**Prec.**	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU
YOLO11-L	FP32	5.39	**6.11**	3.52	**3.79**	**2.66**	2.43	**2.14**	1.91
YOLO11-M	FP32	6.77	**7.77**	4.30	**4.78**	**3.23**	3.04	**2.62**	2.40
YOLO11-S	FP32	14.18	**14.32**	8.70	**9.17**	**6.67**	6.06	**5.30**	4.71
YOLO11-N	FP32	**24.82**	18.38	**15.32**	12.84	**11.76**	9.67	**9.19**	7.66
YOLO11-L	FP16	11.79	**11.80**	7.34	**7.54**	**5.49**	4.89	**4.37**	3.75
YOLO11-M	FP16	**14.50**	14.23	9.11	**9.25**	**6.82**	6.03	**5.50**	4.71
YOLO11-S	FP16	**27.22**	18.52	**18.01**	13.02	**12.86**	9.58	**10.10**	7.67
YOLO11-N	FP16	**38.85**	20.53	**26.53**	14.99	**19.15**	11.19	**16.23**	8.69
YOLO11-L	INT8	**17.89**	16.41	**11.18**	10.90	**8.37**	7.45	**6.55**	5.79
YOLO11-M	INT8	**22.73**	17.59	**14.12**	12.11	**10.12**	8.92	**8.11**	7.21
YOLO11-S	INT8	**38.10**	20.41	**25.26**	14.88	**18.61**	11.13	**15.13**	8.70
YOLO11-N	INT8	**46.16**	21.14	**31.64**	15.83	**23.99**	11.98	**19.53**	9.41
YOLOv8-L	FP32	4.05	**4.63**	2.64	**2.90**	**2.02**	1.88	**1.61**	1.45
YOLOv8-M	FP32	6.22	**7.38**	4.16	**4.60**	**3.17**	2.96	**2.58**	2.33
YOLOv8-S	FP32	13.97	**14.23**	9.04	**9.20**	**6.70**	5.95	**5.13**	4.58
YOLOv8-N	FP32	**26.18**	18.61	**16.55**	13.04	**12.27**	9.66	**9.42**	7.78
YOLOv8-L	FP16	7.86	**9.10**	4.93	**5.71**	3.79	**3.82**	**3.23**	2.90
YOLOv8-M	FP16	**13.31**	**13.31**	7.72	**8.73**	**6.27**	5.60	**4.64**	4.42
YOLOv8-S	FP16	**27.16**	18.03	**17.19**	12.79	**12.54**	9.53	**9.93**	7.63
YOLOv8-N	FP16	**39.83**	20.53	**25.62**	14.96	**19.54**	11.11	**15.79**	8.71
YOLOv8-L	INT8	**15.24**	15.12	**9.90**	9.72	**7.37**	6.46	**5.80**	5.01
YOLOv8-M	INT8	**22.03**	17.59	**12.96**	11.90	**9.89**	8.68	**7.71**	6.87
YOLOv8-S	INT8	**41.23**	20.54	**27.89**	15.12	**21.12**	11.19	**16.06**	8.72
YOLOv8-N	INT8	**47.12**	21.17	**32.26**	15.91	**24.96**	12.01	**19.75**	9.39

Note: for each CUDA–CPU pair, the higher value is in **bold**.

Table A.4: Throughput (fps/stream) for MOG2_CUDA_50 and MOG2_CPU_50 across all 96 configurations.

		B4		B6		B8		B10	
Model	**Prec.**	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU
YOLO11-L	FP32	**6.90**	6.55	**4.89**	4.21	**3.86**	2.85	**3.36**	2.30
YOLO11-M	FP32	**9.00**	8.24	**6.06**	5.25	**4.72**	3.58	**4.05**	2.91
YOLO11-S	FP32	**19.28**	13.69	**12.19**	9.78	**9.73**	6.88	**8.16**	5.51
YOLO11-N	FP32	**33.24**	18.05	**20.71**	13.00	**16.56**	9.85	**13.39**	7.78
YOLO11-L	FP16	**15.42**	11.93	**10.14**	8.20	**8.08**	5.66	**6.76**	4.50
YOLO11-M	FP16	**19.43**	13.77	**12.67**	9.83	**9.99**	6.79	**8.45**	5.56
YOLO11-S	FP16	**38.21**	18.66	**24.53**	13.35	**17.92**	9.88	**14.64**	7.76
YOLO11-N	FP16	**55.50**	20.67	**36.74**	15.25	**26.71**	11.39	**22.64**	8.85
YOLO11-L	INT8	**24.55**	15.86	**15.61**	11.33	**11.97**	8.36	**9.95**	6.73
YOLO11-M	INT8	**30.00**	17.24	**19.11**	12.38	**14.29**	9.28	**12.02**	7.60
YOLO11-S	INT8	**53.04**	20.58	**34.00**	15.16	**25.54**	11.40	**21.13**	8.87
YOLO11-N	INT8	**59.66**	21.42	**40.30**	15.99	**30.48**	12.08	**24.39**	9.46
YOLOv8-L	FP32	**5.20**	4.97	**3.74**	3.25	**3.09**	2.18	**2.60**	1.75
YOLOv8-M	FP32	**8.55**	7.80	**5.95**	5.04	**4.75**	3.46	**4.01**	2.85
YOLOv8-S	FP32	**19.33**	13.64	**12.51**	9.74	**9.65**	6.83	**7.83**	5.36
YOLOv8-N	FP32	**33.22**	18.36	**21.58**	13.19	**16.58**	9.88	**13.53**	7.87
YOLOv8-L	FP16	**11.12**	9.64	**7.59**	6.38	**6.24**	4.42	**5.26**	3.44
YOLOv8-M	FP16	**18.00**	13.05	**11.79**	9.27	**9.39**	6.38	**7.79**	5.16
YOLOv8-S	FP16	**36.13**	18.23	**23.26**	13.19	**17.45**	9.82	**14.33**	7.66
YOLOv8-N	FP16	**55.54**	20.71	**35.72**	15.24	**27.09**	11.36	**21.72**	8.91
YOLOv8-L	INT8	**20.87**	14.58	**13.63**	10.32	**10.59**	7.44	**8.83**	5.88
YOLOv8-M	INT8	**28.58**	17.21	**17.46**	12.21	**13.67**	9.26	**11.26**	7.56
YOLOv8-S	INT8	**57.37**	20.89	**37.65**	15.24	**27.03**	11.51	**21.46**	8.92
YOLOv8-N	INT8	**60.20**	21.39	**40.36**	16.20	**30.47**	12.19	**24.38**	9.48

Note: for each CUDA–CPU pair, the higher value is in **bold**.

Table A.5: Throughput (fps/stream) for MOG2_CUDA_5 and MOG2_CPU_5 across all 96 configurations.

		B4		B6		B8		B10	
Model	**Prec.**	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU
YOLO11-L	FP32	**33.81**	14.89	**18.36**	9.49	**11.61**	6.39	**10.91**	4.60
YOLO11-M	FP32	**41.42**	16.40	**21.96**	10.71	**13.71**	7.29	**11.07**	5.37
YOLO11-S	FP32	**57.00**	19.58	**35.05**	13.98	**25.24**	10.16	**20.75**	7.84
YOLO11-N	FP32	**60.98**	21.13	**40.68**	15.68	**30.47**	11.90	**24.38**	9.34
YOLO11-L	FP16	**53.58**	18.62	**31.93**	13.15	**23.22**	9.39	**19.60**	7.08
YOLO11-M	FP16	**57.89**	19.66	**36.19**	13.96	**25.90**	10.19	**21.36**	7.85
YOLO11-S	FP16	**60.96**	21.34	**40.63**	15.92	**30.42**	11.82	**24.41**	9.27
YOLO11-N	FP16	**61.02**	21.85	**40.71**	16.66	**30.47**	12.50	**24.41**	9.87
YOLO11-L	INT8	**60.59**	20.30	**39.34**	14.79	**28.75**	10.93	**23.11**	8.51
YOLO11-M	INT8	**60.95**	20.83	**40.55**	15.37	**30.38**	11.47	**24.16**	9.05
YOLO11-S	INT8	**61.02**	21.93	**40.67**	16.67	**30.54**	12.43	**24.41**	9.67
YOLO11-N	INT8	**61.03**	22.08	**40.71**	16.85	**30.54**	12.65	**24.43**	9.90
YOLOv8-L	FP32	**27.63**	13.19	**15.20**	8.19	**8.94**	5.36	**6.76**	3.73
YOLOv8-M	FP32	**39.19**	16.05	**21.76**	10.55	**13.82**	7.25	**11.46**	5.22
YOLOv8-S	FP32	**57.61**	19.58	**35.83**	14.06	**24.83**	10.17	**20.50**	7.75
YOLOv8-N	FP32	**61.01**	21.30	**40.65**	15.79	**30.56**	11.89	**24.38**	9.34
YOLOv8-L	FP16	**45.37**	17.34	**25.66**	11.79	**17.68**	8.32	**15.52**	6.09
YOLOv8-M	FP16	**56.83**	19.20	**34.87**	13.68	**24.28**	9.89	**20.84**	7.61
YOLOv8-S	FP16	**60.98**	21.20	**40.65**	15.80	**30.51**	11.89	**24.43**	9.30
YOLOv8-N	FP16	**61.02**	21.88	**40.68**	16.56	**30.52**	12.43	**24.41**	9.73
YOLOv8-L	INT8	**58.66**	19.94	**37.42**	14.33	**26.86**	10.41	**21.55**	8.02
YOLOv8-M	INT8	**60.95**	20.89	**40.41**	15.23	**30.04**	11.38	**23.83**	9.00
YOLOv8-S	INT8	**61.03**	21.95	**40.68**	16.67	**30.52**	12.47	**24.41**	9.74
YOLOv8-N	INT8	**61.04**	22.16	**40.68**	16.88	**30.53**	12.60	**24.42**	9.83

Note: for each CUDA–CPU pair, the higher value is in **bold**.

A.3 Plugin Overhead: Complete Results

Table A.6 reports the throughput comparison between the NO_MOG2 pipeline and MOG2_CUDA_NODROP across all 96 configurations. Running the gate at 100 % motion density provides no frame-drop benefit, so the table isolates the pure computational cost of integrating the plugin. Values meeting the IEC 12.5 fps/stream threshold are in bold.

Table A.6: Plugin overhead: NO_MOG2 vs. MOG2_CUDA_NODROP (fps/stream).

		B4		B6		B8		B10	
Model	**Prec.**	NO	NODROP	NO	NODROP	NO	NODROP	NO	NODROP
YOLO11-L	FP32	6.36	5.39	4.34	3.52	3.57	2.66	2.92	2.14
YOLO11-M	FP32	8.30	6.77	5.53	4.30	4.15	3.23	3.68	2.62
YOLO11-S	FP32	**18.20**	**14.18**	12.28	8.70	9.06	6.67	7.29	5.30
YOLO11-N	FP32	**33.89**	**24.82**	**22.32**	**15.32**	**17.98**	11.76	**14.31**	9.19
YOLO11-L	FP16	**14.58**	11.79	10.34	7.34	7.93	5.49	6.26	4.37
YOLO11-M	FP16	**18.42**	**14.50**	**12.64**	9.11	10.08	6.82	7.18	5.50
YOLO11-S	FP16	**37.37**	**27.22**	**22.74**	**18.01**	**19.55**	**12.86**	**15.88**	10.10
YOLO11-N	FP16	**56.30**	**38.85**	**36.78**	**26.53**	**27.86**	**19.15**	**24.39**	**16.23**
YOLO11-L	INT8	**23.57**	**17.89**	**16.00**	11.18	**12.65**	8.37	10.15	6.55
YOLO11-M	INT8	**29.53**	**22.73**	**20.23**	**14.12**	**15.17**	10.12	12.46	8.11
YOLO11-S	INT8	**54.64**	**38.10**	**35.73**	**25.26**	**28.90**	**18.61**	**24.39**	**15.13**
YOLO11-N	INT8	**60.97**	**46.16**	**40.67**	**31.64**	**30.48**	**23.99**	**24.41**	**19.53**
YOLOv8-L	FP32	4.79	4.05	3.66	2.64	2.74	2.02	2.17	1.61
YOLOv8-M	FP32	8.72	6.22	5.91	4.16	4.46	3.17	3.58	2.58
YOLOv8-S	FP32	**18.29**	**13.97**	**12.65**	9.04	9.95	6.70	7.73	5.13
YOLOv8-N	FP32	**34.25**	**26.18**	**23.80**	**16.55**	**18.32**	12.27	**14.73**	9.42
YOLOv8-L	FP16	11.21	7.86	7.63	4.93	5.85	3.79	4.61	3.23
YOLOv8-M	FP16	**17.24**	**13.31**	12.36	7.72	9.34	6.27	7.49	4.64
YOLOv8-S	FP16	**35.99**	**27.16**	**25.85**	**17.19**	**19.20**	**12.54**	**15.54**	9.93
YOLOv8-N	FP16	**59.03**	**39.83**	**36.42**	**25.62**	**30.49**	**19.54**	**24.38**	**15.79**
YOLOv8-L	INT8	**20.46**	**15.24**	**13.91**	9.90	11.11	7.37	8.78	5.80
YOLOv8-M	INT8	**28.62**	**22.03**	**18.68**	**12.96**	**14.53**	9.89	12.26	7.71
YOLOv8-S	INT8	**60.99**	**41.23**	**40.63**	**27.89**	**30.50**	**21.12**	**24.39**	**16.06**
YOLOv8-N	INT8	**61.00**	**47.12**	**40.67**	**32.26**	**30.51**	**24.96**	**24.41**	**19.75**

Note: Values meeting the IEC threshold of 12.5 fps/stream are shown in **bold**.

A.4 Motion Gating Tradeoff: Complete Results

Tables A.7–A.10 report throughput for all 96 configurations across NO_MOG2 and MOG2_CUDA at motion densities $k \in \{100, 75, 50, 25, 10, 5\}\,\%$.

Table A.7: Motion gating tradeoff, batch size 4 (fps/stream).

Model	Prec.	NO	k=100	k=75	k=50	k=25	k=10	k=5
YOLO11-L	FP32	6.36	5.39	5.84	†6.90	†11.04	†**21.73**	†**33.81**
YOLO11-M	FP32	8.30	6.77	7.30	†9.00	†**14.79**	†**27.52**	†**41.42**
YOLO11-S	FP32	**18.20**	**14.18**	**15.27**	†**19.28**	†**28.01**	†**46.37**	†**57.00**
YOLO11-N	FP32	**33.89**	**24.82**	**28.11**	**33.24**	†**44.42**	†**60.03**	†**60.98**
YOLO11-L	FP16	**14.58**	11.79	**12.77**	†**15.42**	†**23.25**	†**39.49**	†**53.58**
YOLO11-M	FP16	**18.42**	**14.50**	**15.74**	†**19.43**	†**28.21**	†**46.07**	†**57.89**
YOLO11-S	FP16	**37.37**	**27.22**	**31.88**	†**38.21**	†**52.12**	†**60.84**	†**60.96**
YOLO11-N	FP16	**56.30**	**38.85**	**45.25**	**55.50**	†**60.84**	†**60.98**	†**61.02**
YOLO11-L	INT8	**23.57**	**17.89**	**20.09**	†**24.55**	†**34.29**	†**52.26**	†**60.59**
YOLO11-M	INT8	**29.53**	**22.73**	**24.74**	†**30.00**	†**40.71**	†**58.91**	†**60.95**
YOLO11-S	INT8	**54.64**	**38.10**	**43.79**	**53.04**	†**60.99**	†**61.01**	†**61.02**
YOLO11-N	INT8	**60.97**	**46.16**	**52.61**	**59.66**	**60.76**	†**61.00**	†**61.03**
YOLOv8-L	FP32	4.79	4.05	4.31	†5.20	†8.51	†**17.30**	†**27.63**
YOLOv8-M	FP32	8.72	6.22	7.05	8.55	†**14.13**	†**26.13**	†**39.19**
YOLOv8-S	FP32	**18.29**	**13.97**	**15.24**	†**19.33**	†**28.25**	†**44.66**	†**57.61**
YOLOv8-N	FP32	**34.25**	**26.18**	**28.70**	**33.22**	†**44.86**	†**60.13**	†**61.01**
YOLOv8-L	FP16	11.21	7.86	9.24	11.12	†**17.42**	†**31.71**	†**45.37**
YOLOv8-M	FP16	**17.24**	**13.31**	**14.70**	†**18.00**	†**27.13**	†**43.48**	†**56.83**
YOLOv8-S	FP16	**35.99**	**27.16**	**30.39**	†**36.13**	†**48.73**	†**60.79**	†**60.98**
YOLOv8-N	FP16	**59.03**	**39.83**	**46.13**	**55.54**	†**60.98**	†**60.96**	†**61.02**
YOLOv8-L	INT8	**20.46**	**15.24**	**17.14**	†**20.87**	†**30.95**	†**48.14**	†**58.66**
YOLOv8-M	INT8	**28.62**	**22.03**	**23.40**	**28.58**	†**39.56**	†**56.86**	†**60.95**
YOLOv8-S	INT8	**60.99**	**41.23**	**45.76**	**57.37**	**60.84**	†**61.02**	†**61.03**
YOLOv8-N	INT8	**61.00**	**47.12**	**53.83**	**60.20**	†**61.09**	†**61.04**	†**61.04**

Note: Values in **bold** meet the IEC threshold of 12.5 fps/stream;
† denotes configurations exceeding the NO_MOG2 baseline.

Table A.8: Motion gating tradeoff, batch size 6 (fps/stream).

Model	Prec.	NO	k=100	k=75	k=50	k=25	k=10	k=5
YOLO11-L	FP32	4.34	3.52	4.03	†4.89	†6.51	†10.98	†**18.36**
YOLO11-M	FP32	5.53	4.30	5.02	†6.06	†8.01	†**13.94**	†**21.96**
YOLO11-S	FP32	12.28	8.70	10.18	12.19	†**16.05**	†**25.31**	†**35.05**
YOLO11-N	FP32	**22.32**	**15.32**	**17.10**	**20.71**	†**27.09**	†**37.80**	†**40.68**
YOLO11-L	FP16	10.34	7.34	8.42	10.14	†**13.47**	†**20.96**	†**31.93**
YOLO11-M	FP16	**12.64**	9.11	10.49	†**12.67**	†**16.81**	†**26.12**	†**36.19**
YOLO11-S	FP16	**22.74**	**18.01**	**20.28**	†**24.53**	†**31.93**	†**40.48**	†**40.63**
YOLO11-N	FP16	**36.78**	**26.53**	**30.53**	**36.74**	†**40.60**	†**40.68**	†**40.71**
YOLO11-L	INT8	**16.00**	11.18	**13.01**	**15.61**	†**20.57**	†**31.17**	†**39.34**
YOLO11-M	INT8	**20.23**	**14.12**	**15.83**	**19.11**	†**24.56**	†**35.80**	†**40.55**
YOLO11-S	INT8	**35.73**	**25.26**	**28.13**	**34.00**	†**40.53**	†**40.69**	†**40.67**
YOLO11-N	INT8	**40.67**	**31.64**	**36.63**	**40.30**	†**40.68**	†**40.69**	†**40.71**
YOLOv8-L	FP32	3.66	2.64	3.10	†3.74	†5.02	†9.00	†**15.20**
YOLOv8-M	FP32	5.91	4.16	4.93	†5.95	†8.03	†**13.72**	†**21.76**
YOLOv8-S	FP32	**12.65**	9.04	10.41	**12.51**	†**16.27**	†**25.63**	†**35.83**
YOLOv8-N	FP32	**23.80**	**16.55**	**17.91**	**21.58**	†**28.35**	†**38.86**	†**40.65**
YOLOv8-L	FP16	7.63	4.93	6.22	7.59	†10.03	†**16.39**	†**25.66**
YOLOv8-M	FP16	12.36	7.72	9.82	11.79	†**15.54**	†**24.86**	†**34.87**
YOLOv8-S	FP16	**25.85**	**17.19**	**19.36**	**23.26**	†**30.38**	†**40.14**	†**40.65**
YOLOv8-N	FP16	**36.42**	**25.62**	**29.82**	**35.72**	†**40.63**	†**40.67**	†**40.68**
YOLOv8-L	INT8	**13.91**	9.90	11.20	**13.63**	†**18.09**	†**27.87**	†**37.42**
YOLOv8-M	INT8	**18.68**	**12.96**	**14.52**	**17.46**	†**22.59**	†**33.57**	†**40.41**
YOLOv8-S	INT8	**40.63**	**27.89**	**31.12**	**37.65**	**40.63**	†**40.71**	†**40.68**
YOLOv8-N	INT8	**40.67**	**32.26**	**36.97**	**40.36**	**40.64**	**40.67**	†**40.68**

Note: Values in **bold** meet the IEC threshold of 12.5 fps/stream;
† denotes configurations exceeding the NO_MOG2 baseline.

Table A.9: Motion gating tradeoff, batch size 8 (fps/stream).

Model	Prec.	NO	k=100	k=75	k=50	k=25	k=10	k=5
YOLO11-L	FP32	3.57	2.66	3.06	†3.86	†5.39	†8.07	†11.61
YOLO11-M	FP32	4.15	3.23	3.80	†4.72	†6.25	†9.63	†**13.71**
YOLO11-S	FP32	9.06	6.67	7.82	†9.73	†**13.51**	†**19.79**	†**25.24**
YOLO11-N	FP32	**17.98**	11.76	**13.54**	**16.56**	†**22.59**	†**29.59**	†**30.47**
YOLO11-L	FP16	7.93	5.49	6.39	†8.08	†11.74	†**17.32**	†**23.22**
YOLO11-M	FP16	10.08	6.82	7.94	9.99	†**13.71**	†**20.62**	†**25.90**
YOLO11-S	FP16	**19.55**	**12.86**	**14.55**	**17.92**	†**24.72**	†**30.43**	†**30.42**
YOLO11-N	FP16	**27.86**	**19.15**	**21.82**	**26.71**	†**30.45**	†**30.52**	†**30.47**
YOLO11-L	INT8	**12.65**	8.37	9.72	11.97	†**16.30**	†**23.73**	†**28.75**
YOLO11-M	INT8	**15.17**	10.12	11.69	**14.29**	†**19.40**	†**27.02**	†**30.38**
YOLO11-S	INT8	**28.90**	**18.61**	**21.07**	**25.54**	†**30.44**	†**30.48**	†**30.54**
YOLO11-N	INT8	**30.48**	**23.99**	**27.46**	**30.48**	†**30.51**	**30.46**	†**30.54**
YOLOv8-L	FP32	2.74	2.02	2.38	†3.09	†4.15	†6.03	†8.94
YOLOv8-M	FP32	4.46	3.17	3.77	†4.75	†6.61	†9.45	†**13.82**
YOLOv8-S	FP32	9.95	6.70	7.80	9.65	†**13.15**	†**19.47**	†**24.83**
YOLOv8-N	FP32	**18.32**	12.27	**13.52**	**16.58**	†**22.53**	†**29.27**	†**30.56**
YOLOv8-L	FP16	5.85	3.79	4.95	†6.24	†9.13	†**13.53**	†**17.68**
YOLOv8-M	FP16	9.34	6.27	7.37	†9.39	†**13.14**	†**19.42**	†**24.28**
YOLOv8-S	FP16	**19.20**	**12.54**	**14.38**	**17.45**	†**23.67**	†**30.12**	†**30.51**
YOLOv8-N	FP16	**30.49**	**19.54**	**22.30**	**27.09**	**30.43**	†**30.53**	†**30.52**
YOLOv8-L	INT8	11.11	7.37	8.61	10.59	†**14.62**	†**21.31**	†**26.86**
YOLOv8-M	INT8	**14.53**	9.89	11.21	**13.67**	†**18.54**	†**25.81**	†**30.04**
YOLOv8-S	INT8	**30.50**	**21.12**	**23.16**	**27.03**	**30.49**	†**30.52**	†**30.52**
YOLOv8-N	INT8	**30.51**	**24.96**	**28.19**	**30.47**	**30.50**	†**30.52**	†**30.53**

Note: Values in **bold** meet the IEC threshold of 12.5 fps/stream;
† denotes configurations exceeding the NO_MOG2 baseline.

Table A.10: Motion gating tradeoff, batch size 10 (fps/stream).

Model	Prec.	NO	k=100	k=75	k=50	k=25	k=10	k=5
YOLO11-L	FP32	2.92	2.14	2.53	†3.36	†5.00	†8.09	†10.91
YOLO11-M	FP32	3.68	2.62	3.14	†4.05	†5.64	†8.63	†11.07
YOLO11-S	FP32	7.29	5.30	6.34	†8.16	†11.81	†**17.54**	†**20.75**
YOLO11-N	FP32	**14.31**	9.19	10.73	**13.39**	†**18.71**	†**24.27**	†**24.38**
YOLO11-L	FP16	6.26	4.37	5.21	†6.76	†9.87	†**15.78**	†**19.60**
YOLO11-M	FP16	7.18	5.50	6.51	†8.45	†12.04	†**18.38**	†**21.36**
YOLO11-S	FP16	**15.88**	10.10	11.70	**14.64**	†**20.09**	†**24.23**	†**24.41**
YOLO11-N	FP16	**24.39**	**16.23**	**18.49**	**22.64**	**24.38**	†**24.40**	†**24.41**
YOLO11-L	INT8	10.15	6.55	7.90	9.95	†**13.98**	†**20.77**	†**23.11**
YOLO11-M	INT8	12.46	8.11	9.68	12.02	†**16.63**	†**22.75**	†**24.16**
YOLO11-S	INT8	**24.39**	**15.13**	**17.21**	**21.13**	†**24.46**	†**24.40**	†**24.41**
YOLO11-N	INT8	**24.41**	**19.53**	**22.34**	**24.39**	**24.41**	**24.41**	†**24.43**
YOLOv8-L	FP32	2.17	1.61	1.90	†2.60	†3.83	†5.32	†6.76
YOLOv8-M	FP32	3.58	2.58	3.11	†4.01	†6.14	†9.05	†11.46
YOLOv8-S	FP32	7.73	5.13	6.15	†7.83	†11.53	†**17.19**	†**20.50**
YOLOv8-N	FP32	**14.73**	9.42	10.97	**13.53**	†**19.13**	†**24.02**	†**24.38**
YOLOv8-L	FP16	4.61	3.23	3.89	†5.26	†8.05	†**12.87**	†**15.52**
YOLOv8-M	FP16	7.49	4.64	6.07	†7.79	†11.72	†**17.68**	†**20.84**
YOLOv8-S	FP16	**15.54**	9.93	11.56	**14.33**	†**19.84**	†**24.24**	†**24.43**
YOLOv8-N	FP16	**24.38**	**15.79**	**18.05**	**21.72**	†**24.41**	†**24.41**	†**24.41**
YOLOv8-L	INT8	8.78	5.80	6.89	†8.83	†12.43	†**18.09**	†**21.55**
YOLOv8-M	INT8	12.26	7.71	9.08	11.26	†**15.83**	†**21.96**	†**23.83**
YOLOv8-S	INT8	**24.39**	**16.06**	**18.00**	**21.46**	†**24.47**	†**24.41**	†**24.41**
YOLOv8-N	INT8	**24.41**	**19.75**	**22.28**	**24.38**	**24.40**	†**24.42**	†**24.42**

Note: Values in **bold** meet the IEC threshold of 12.5 fps/stream;
† denotes configurations exceeding the NO_MOG2 baseline.

A.5 Time To First Detection: Complete Results

Tables A.11 and A.12 report TTFD statistics for the person and car classes, evaluated on VIRAT_S_000002 using MOG2_CUDA (500,px threshold, safe-pass active) and NO_MOG2. Results are aggregated across streams; no meaningful variation was observed across streams. Entries marked "—" correspond to configurations excluded from the evaluation (see Section 5.6.1). Values in frames.

Table A.11: TTFD for the person class (7 objects, warm-start) under MOG2_CUDA and NO_MOG2. Det = detected / total. "—": excluded from evaluation (see Section 5.6.1). Values in frames.

			MOG2_CUDA			NO_MOG2		
Model	**Prec.**	**Det**	**Min**	**Mean**	**Max**	**Min**	**Mean**	**Max**
YOLO11-L	FP32				—			
YOLO11-M	FP32				—			
YOLO11-S	FP32	7/7	0	9.3	37	0	9.3	37
YOLO11-N	FP32	7/7	0	43.1	217	0	43.1	217
YOLO11-L	FP16				—			
YOLO11-M	FP16	7/7	0	5.0	33	0	5.0	33
YOLO11-S	FP16	7/7	0	9.0	37	0	9.0	37
YOLO11-N	FP16	7/7	0	43.1	217	0	43.1	217
YOLO11-L	INT8	7/7	7	82.0	294	7	82.0	294
YOLO11-M	INT8	7/7	0	16.9	57	0	16.9	57
YOLO11-S	INT8	7/7	49	336.9	1418	49	322.1	1315
YOLO11-N	INT8	5/7	93	1054.8	2213	93	881.6	2069
YOLOv8-L	FP32				—			
YOLOv8-M	FP32				—			
YOLOv8-S	FP32	7/7	0	10.0	35	0	10.0	35
YOLOv8-N	FP32	7/7	0	50.3	200	0	50.3	200
YOLOv8-L	FP16				—			
YOLOv8-M	FP16	7/7	0	7.3	33	0	7.3	33
YOLOv8-S	FP16	7/7	0	10.0	35	0	10.0	35
YOLOv8-N	FP16	7/7	0	48.6	200	0	48.6	200
YOLOv8-L	INT8	7/7	14	60.4	128	14	60.4	128
YOLOv8-M	INT8	7/7	66	501.0	1334	66	498.1	1315
YOLOv8-S	INT8	7/7	1	182.9	852	1	124.6	444
YOLOv8-N	INT8	7/7	48	372.7	1636	48	323.7	1293

Table A.12: TTFD for the car class (1 object, cold-start, entry at frame 2097) under MOG2_CUDA and NO_MOG2. Det = detected / total. "—": excluded from evaluation (see Section 5.6.1). Values in frames.

			MOG2_CUDA			NO_MOG2		
Model	**Prec.**	**Det**	**Min**	**Mean**	**Max**	**Min**	**Mean**	**Max**
YOLO11-L	FP32				—			
YOLO11-M	FP32				—			
YOLO11-S	FP32	1/1	0	0	0	0	0	0
YOLO11-N	FP32	1/1	0	0	0	0	0	0
YOLO11-L	FP16				—			
YOLO11-M	FP16	1/1	0	0	0	0	0	0
YOLO11-S	FP16	1/1	0	0	0	0	0	0
YOLO11-N	FP16	1/1	0	0	0	0	0	0
YOLO11-L	INT8	1/1	0	0	0	0	0	0
YOLO11-M	INT8	1/1	0	0	0	0	0	0
YOLO11-S	INT8	1/1	132	132	132	132	132	132
YOLO11-N	INT8	1/1	12	12	12	12	12	12
YOLOv8-L	FP32				—			
YOLOv8-M	FP32				—			
YOLOv8-S	FP32	1/1	0	0	0	0	0	0
YOLOv8-N	FP32	1/1	0	0	0	0	0	0
YOLOv8-L	FP16				—			
YOLOv8-M	FP16	1/1	0	0	0	0	0	0
YOLOv8-S	FP16	1/1	0	0	0	0	0	0
YOLOv8-N	FP16	1/1	0	0	0	0	0	0
YOLOv8-L	INT8	1/1	0	0	0	0	0	0
YOLOv8-M	INT8	1/1	0	0	0	0	0	0
YOLOv8-S	INT8	1/1	0	0	0	0	0	0
YOLOv8-N	INT8	1/1	14	14	14	14	14	14

In accordance with the University of Bologna's guidelines on transparency in the use of large language models (LLMs), it is hereby disclosed that the preparation, formatting, and textual editing of this document made use of such tools. The development and validation of the technical content reflect the author's independent understanding and design effort.