
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE ELECTRÓNICA E INFORMÁTICA

**OPTIMIZACIÓN DE FLUJOS DE PROCESAMIENTO
PARA APLICACIÓN DE MEDICIÓN BASADA EN
VISIÓN EN TIEMPO REAL MEDIANTE APRENDIZAJE
PROFUNDO Y NVIDIA DEEPSTREAM SOBRE GPU
EMBEBIDA**

Tesis de Grado presentada por

Augusto Winkler

como requisito parcial para optar al título de

Ingeniero en Informática

Director de Tesis
Dr. Jorge Portilla

Concepción, 2026.



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: Optimización de flujos de procesamiento para aplicación de medición basada en visión en tiempo real mediante aprendizaje profundo y NVIDIA DeepStream sobre GPU embebida

Nombre del candidato(a): Paul Augusto Winkler Sáez

Carrera / Grado: Ingeniería en Informática

Campus: Sede Concepción Departamento: Departamento de Electrónica e Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, JORGE ALEXIS PORTILLA GÓMEZ, en mi calidad de profesor guía/director del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (marcar una opción):

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 20 de julio del 2026

Firma: _____

JORGE ALEXIS
PORTILLA
GOMEZ

Firmado digitalmente
por JORGE ALEXIS
PORTILLA GOMEZ
Fecha: 2026.07.20
09:30:59 -0400

Estudiante o Candidato(a):

Fecha: 18 de julio del 2026

Firma: _____

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

CONTENIDO

ÍNDICE DE FIGURAS	v
ÍNDICE DE TABLAS	vii
NOMENCLATURA	viii
RESUMEN	1
ABSTRACT	2
1. INTRODUCCIÓN	1
1.1. Motivación y Contexto	1
1.2. Planteamiento del problema	3
1.3. Propuesta de la solución	3
1.4. Objetivos	4
1.4.1. Objetivo general	4
1.4.2. Objetivos específicos	4
1.5. Contribuciones	4
1.6. Alcances y limitaciones	5
1.7. Organización de la Tesis	6
2. MARCO TEÓRICO	7
2.1. Aprendizaje profundo para regresión densa en visión por computador	7
2.1.1. Redes neuronales y regresión de variables continuas	7
2.1.2. Redes neuronales convolucionales para imágenes	8
2.1.3. Fundamentos de U-Net y Attention U-Net	9
2.2. Computación acelerada por GPU	9
2.2.1. Arquitectura general de GPU	10
2.2.2. Modelo de programación CUDA	11
2.2.3. Gestión de memoria en GPU	12
2.2.4. GPUs embebidas y plataformas Jetson	14
2.3. Optimización de inferencia en redes neuronales	15
2.3.1. Entrenamiento e inferencia	15

2.3.2.	Formatos de grafo como representación intermedia	15
2.3.3.	Optimización de inferencia mediante TensorRT	16
2.4.	Integración de flujos de inferencia en tiempo real	18
2.4.1.	Imagen, video y streaming	19
2.4.2.	GStreamer para la construcción de pipelines	19
2.4.3.	NVIDIA DeepStream	21
3.	METODOLOGÍA	24
3.1.	Contexto general del sistema	24
3.1.1.	Contexto de aplicación	24
3.1.2.	Entorno de operación	27
3.1.3.	Conjuntos de datos	27
3.1.4.	Modelos preentrenados	29
3.1.5.	Consideraciones transversales al pipeline	29
3.2.	Pipeline de referencia en Python	30
3.2.1.	Decisiones de diseño	31
3.2.2.	Decodificación	32
3.2.3.	Preprocesamiento	32
3.2.4.	Inferencia	35
3.2.5.	Postprocesamiento y visualización	35
3.3.	Configuración de TensorRT	36
3.3.1.	Construcción de engines TensorRT	37
3.3.2.	Protocolo de evaluación de los engines TensorRT	38
3.4.	Pipeline optimizado con NVIDIA DeepStream	39
3.4.1.	Arquitectura general del pipeline	40
3.4.2.	Decodificación y multiplexación	41
3.4.3.	Preprocesamiento acelerado	41
3.4.4.	Inferencia	43
3.4.5.	Postprocesamiento y visualización	44
3.4.6.	Modos de operación	49
3.5.	Evaluación comparativa de rendimiento	49
3.5.1.	Principios de medición	49
3.5.2.	Definición de métricas	50
3.5.3.	Fenómenos estructurales que condicionan las mediciones	53
3.5.4.	Configuración experimental	54
4.	RESULTADOS	55
4.1.	Evaluación de los engines TensorRT	55
4.1.1.	Propiedades estructurales de los modelos	55
4.1.2.	Precisión de las predicciones	55
4.1.3.	Rendimiento computacional de los engines	56
4.1.4.	Selección de precisión para DeepStream	58
4.2.	Validación cualitativa del pipeline	59

4.3. Latencia	59
4.3.1. Latencia por etapa	59
4.3.2. Latencia end-to-end	61
4.4. Throughput	62
4.5. Uso de memoria	63
4.6. Análisis de resultados	64
5. CONCLUSIONES	66
REFERENCIAS	69

Índice de figuras

2.1. Diagrama de comunicación entre host y device para GPU integrada (izquierda) y GPU discreta (derecha).	11
2.2. Representación de un grafo computacional para un modelo de machine learning basado en una red residual simple a través de operaciones de convolución (conv), funciones de activación (ReLU) y pooling (MaxPool).	16
2.3. Flujo de una TensorRT Application, desde un modelo base hasta la ejecución de inferencia optimizada.	17
3.1. Esquema del arreglo óptico BEMI para capturar la radiación emitida por las partículas de hollín.	26
3.2. Bseline Python implementado para la inferencia de temperatura de hollín.	31
3.3. Frame original decodificado desde el video de entrada en el baseline Python, previo a cualquier procesamiento (2048×1536 px).	32
3.4. Recorte de la ROI y rotación 90° antihoraria en el baseline Python, aplicados mediante slicing con índices fijos, transposición de ejes y rebanado inverso.	33
3.5. Tensor de entrada del baseline Python tras reescalado mediante interpolación bilineal para el proceso posterior de normalización y estandarización por canal.	34
3.6. Salida del modelo, máscara de Otsu y mapa de temperatura resultante en el baseline Python.	36
3.7. Visualización final del baseline Python: panel T_s -RGB con colorbars y valor medio de temperatura (izquierda), y perfil centerline $T(z)$ (derecha).	36
3.8. Pipeline DeepStream implementado para la inferencia de temperatura de hollín.	40
3.9. Frame original decodificado por <code>nvv4l2decoder</code> en el pipeline DeepStream, sobre superficies GPU NV12 y previo a cualquier transformación geométrica o de formato (2048×1536 px).	41
3.10. Secuencia de transformaciones geométricas previas a la construcción del tensor de entrada en el pipeline DeepStream.	42
3.11. Tensor de entrada al modelo en el pipeline DeepStream, tras reordenamiento de canales, normalización y estandarización.	43
3.12. Salida del modelo, máscara de Otsu y mapa de temperatura resultante en el pipeline DeepStream.	45

3.13. Panel de colormap completo en el pipeline DeepStream: los cuatro sub-paneles T_s , R, G, B con sus etiquetas de canal, ambas colorbars (INFERNO y VIRIDIS) y la temperatura media dentro de la máscara.	46
3.14. Gráfico centerline $T(z)$ del pipeline DeepStream, construido íntegramente en GPU, mostrando el perfil de temperatura sobre el eje radial central de la llama.	46
3.15. Canvas final del pipeline DeepStream, compuesto por el panel de colormap y el gráfico centerline lado a lado (1280×512 px).	47
4.1. Campos de temperatura predichos por el modelo base y mapas de diferencia ΔT para distintas precisiones numéricas. Todas las subfiguras comparten la misma escala de temperatura y diferencia.	57
4.2. Salida visual del pipeline DeepStream para las tres condiciones de llama evaluadas.	60

Índice de tablas

3.1. Condiciones de llama simuladas con CoFlame.	28
4.1. Propiedades de los modelos U-Net y Attention U-Net según nivel de cuantización.	56
4.2. Resultados de Regression Accuracy (RMSE en K) bajo distintas condiciones de llama y niveles de cuantización.	56
4.3. Resultados de Regression Closeness (%) para distintos valores de tolerancia absoluta (<i>atol</i>).	58
4.4. Latencia de inferencia (ms), aislada del preprocesamiento y la transferencia CPU-GPU, para los modelos base y engines TRT.	58
4.5. Throughput de inferencia (FPS), derivado de la latencia mediana de la Tabla 4.4.	59
4.6. Latencia por etapa (ms) para ambos pipelines y arquitecturas.	61
4.7. Latencia end-to-end (ms) para ambos pipelines y arquitecturas.	62
4.8. Throughput (fps) para ambos pipelines y arquitecturas.	63
4.9. Uso de memoria neta del sistema (MB).	63
4.10. Síntesis de resultados comparativos entre el baseline Python y el pipeline DeepStream.	64

Nomenclatura

ANN	Artificial Neural Network
API	Application Programming Interface
ARM	Advanced RISC Machine
BEMI	Broadband Emission
CLAD	Canonical Laminar Axisymmetric Diffusion
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDA	Digital Differential Analyzer
DMA	Direct Memory Access
DNN	Deep Neural Network
DPB	Decoded Picture Buffer
EMI	Emission
EOS	End Of Stream
FIFO	First In, First Out
GIL	Global Interpreter Lock
GPC	Graphics Processing Cluster
GPGPU	General-Purpose Computing on GPUs
GPU	Graphics Processing Unit
IoT	Internet of Things
JIT	Just-In-Time
MAE	Modulated Absorption/Emission
NAL	Network Abstraction Layer
NVDEC	NVIDIA Decoder
NVENC	NVIDIA Encoder
ONNX	Open Neural Network Exchange
PCIe	Peripheral Component Interconnect Express
PTS	Presentation TimeStamp
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RMSE	Root Mean Square Error
ROI	Region of Interest
SDK	Software Development Kit
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
SoC	System-on-Chip
TRT	TensorRT

UDP	User Datagram Protocol
VIC	Video Image Compositor
VRAM	Video Random Access Memory

RESUMEN

Este trabajo aborda la optimización de flujos de procesamiento de visión por computador en tiempo real sobre GPU embebida, mediante el diseño, implementación y evaluación comparativa de dos configuraciones de pipeline: una implementación de referencia en Python con ejecución predominantemente en CPU, y un pipeline optimizado construido sobre el SDK NVIDIA DeepStream con ejecución íntegramente acelerada en GPU, desplegado sobre la plataforma embebida NVIDIA Jetson AGX Orin 64GB. Como caso de aplicación experimental se emplea la estimación de la temperatura del hollín en llamas laminares a partir de imágenes RGB mediante las arquitecturas U-Net y Attention U-Net, previamente entrenadas y validadas en trabajos anteriores del grupo de investigación.

La construcción del pipeline optimizado se apoya en dos bloques metodológicos. Primero, se replica y actualiza un flujo de selección de precisión mediante TensorRT, comparando engines cuantizados en FP32, FP16 e INT8 bajo métricas de fidelidad numérica y de rendimiento. Los resultados confirman que INT8 introduce degradaciones espaciales incompatibles con una tarea de regresión térmica continua, mientras que FP16 preserva la fidelidad del modelo base con reducciones de latencia superiores a un orden de magnitud, por lo que se adopta como configuración única para el pipeline optimizado. Segundo, se diseña e implementa el pipeline DeepStream propiamente dicho, integrando decodificación de video por hardware, preprocesamiento acelerado en GPU y un postprocesamiento personalizado desarrollado íntegramente en kernels CUDA propios, incluyendo segmentación adaptativa por el método de Otsu y un renderizador de texto y geometría vectorial en GPU para el perfil de temperatura centerline, dado que la arquitectura estándar de DeepStream, orientada a casos de uso convencionales de detección y segmentación de objetos, no provee soporte nativo para este tipo de visualización científica.

La evaluación comparativa entre ambos pipelines revela un panorama matizado antes que una superioridad uniforme. En throughput bajo régimen de operación continua, el pipeline DeepStream supera al baseline Python por factores de $5.9\times$ a $6.6\times$, con una reducción del uso de memoria neta superior al 92%. En latencia end-to-end aislada por frame, en cambio, DeepStream exhibe una latencia mayor, aunque esta resulta atribuible en su totalidad a las dificultades de aislar por sonda un decodificador de hardware cuyo diseño presupone operación en pipeline.

Este trabajo demuestra la viabilidad práctica de reingeniería completa de un pipeline de inferencia sobre hardware embebido, manteniendo los datos íntegramente en memoria GPU desde la decodificación hasta la transmisión del resultado, y aporta evidencia cuantitativa reproducible, un protocolo de medición riguroso para sistemas de video en tiempo real, y lineamientos de diseño transferibles a otras tareas de regresión visual continua sobre plataformas de cómputo perimetral, más allá del caso de aplicación específico de pirometría de hollín evaluado en esta memoria.

ABSTRACT

This work addresses the optimization of real-time computer vision processing pipelines on embedded GPU, through the design, implementation, and comparative evaluation of two pipeline configurations: a reference implementation in Python with predominantly CPU-based execution, and an optimized pipeline built on the NVIDIA DeepStream SDK with execution fully accelerated on GPU, deployed on the NVIDIA Jetson AGX Orin 64GB embedded platform. As an experimental application case, the estimation of soot temperature in laminar flames from RGB images is employed, using the U-Net and Attention U-Net architectures, previously trained and validated in prior work by the research group.

The construction of the optimized pipeline relies on two methodological blocks. First, a precision selection workflow via TensorRT is replicated and updated, comparing engines quantized in FP32, FP16, and INT8 under numerical fidelity and performance metrics. The results confirm that INT8 introduces spatial degradations incompatible with a continuous thermal regression task, whereas FP16 preserves the fidelity of the base model with latency reductions exceeding one order of magnitude, and is therefore adopted as the sole configuration for the optimized pipeline. Second, the DeepStream pipeline itself is designed and implemented, integrating hardware video decoding, GPU-accelerated preprocessing, and a custom postprocessing stage developed entirely in proprietary CUDA kernels, including adaptive segmentation via Otsu’s method and a GPU-based text and vector geometry renderer for the centerline temperature profile, given that DeepStream’s standard architecture, oriented toward conventional object detection and segmentation use cases, provides no native support for this type of scientific visualization.

The comparative evaluation between both pipelines reveals a nuanced picture rather than a uniform superiority. In throughput under continuous operation, the DeepStream pipeline outperforms the Python baseline by factors of $5.9\times$ to $6.6\times$, with a reduction in net memory usage exceeding 92%. In per-frame isolated end-to-end latency, by contrast, DeepStream exhibits higher latency, although this is fully attributable to the difficulty of probe-isolating a hardware decoder whose design presupposes pipelined operation.

This work demonstrates the practical feasibility of fully re-engineering an inference pipeline on embedded hardware, keeping data entirely in GPU memory from decoding through result transmission, and provides reproducible quantitative evidence, a rigorous measurement protocol for real-time video systems, and design guidelines transferable to other continuous visual regression tasks on edge computing platforms, beyond the specific soot pyrometry application case evaluated in this thesis.

INTRODUCCIÓN

Este documento aborda el problema de la optimización de un pipeline de análisis óptico de llamas en tiempo casi real sobre plataformas GPU (*Graphic Processing Unit*) embebidas, utilizando como caso de aplicación la estimación de la temperatura del hollín en llamas laminares a partir de imágenes RGB mediante redes neuronales profundas (DNN, *Deep Neural Networks*). El trabajo se sitúa en la intersección entre la optimización de sistemas de visión por computador, la computación acelerada y el diagnóstico óptico asistido por aprendizaje profundo, con énfasis en la implementación práctica sobre hardware NVIDIA y en el uso del SDK (*Software Development Kit*) DeepStream como plataforma de integración.

El presente capítulo introduce el contexto general de la investigación, exponiendo la motivación que origina el trabajo y los antecedentes técnicos y científicos que lo sustentan. A partir de este marco, se plantea el problema abordado, se describe la propuesta de solución adoptada y se definen los objetivos y contribuciones del estudio. Finalmente, se establecen los alcances y limitaciones que delimitan esta primera etapa del proyecto, proporcionando una visión clara del enfoque y del rol que cumple esta memoria dentro del desarrollo global del trabajo de título.

1.1. Motivación y Contexto

En las últimas décadas, el aprendizaje profundo se ha consolidado como una herramienta central para el análisis de sistemas complejos mediante visión por computador, permitiendo abordar problemas donde la formulación explícita de modelos resulta limitada o impracticable [1]. Sin embargo, el poder expresivo de las DNN conlleva un aumento considerable en su complejidad computacional, lo que plantea desafíos relevantes no solo durante el entrenamiento, sino especialmente en la etapa de inferencia. En aplicaciones que requieren tiempos de respuesta acotados, ejecución continua o despliegue en hardware con recursos limitados, la eficiencia de la inferencia se convierte en un factor crítico para la viabilidad práctica de estos sistemas.

Este desafío se amplifica cuando se considera el pipeline completo de procesamiento, más allá de la inferencia aislada. En la práctica, un sistema de análisis de video basado en deep learning comprende múltiples etapas: adquisición de datos, preprocesamiento, inferencia y postprocesamiento, cada una de las cuales puede constituir un cuello de botella significativo. Estudios recientes han evidenciado que, a medida que el rendimiento de los aceleradores de deep learning mejora con las sucesivas generaciones de hardware, las etapas de preprocesamiento y las transferencias de datos entre CPU (Central Processing Unit) y GPU se convierten en los factores limitantes del sistema, desplazando el foco de la optimización desde la inferencia hacia el pipeline en su conjunto [2]. Este fenómeno es especialmente pronunciado en plataformas embebidas, donde los recursos de memoria y ancho de banda son más restrictivos, y donde el uso de implementaciones genéricas en Python o de configuraciones típicas de los frameworks de entrenamiento conduce habitualmente a una subutili-

zación del hardware disponible, con transferencias innecesarias entre CPU y GPU, sincronizaciones implícitas y etapas de preprocesamiento ejecutadas íntegramente en CPU [2].

La relevancia de este problema se hace particularmente evidente en aplicaciones de visión por computador sobre plataformas de borde (*edge computing*), donde los datos suelen adquirirse de manera continua y deben ser procesados con latencias predecibles, sin posibilidad de delegar el cómputo a servidores remotos. En estos escenarios, el rendimiento global del sistema no depende únicamente de la precisión del modelo ni de la velocidad de inferencia aislada, sino de cuán eficientemente se articulan todas las etapas del pipeline y de cómo se aprovechan los recursos heterogéneos del hardware disponible, incluyendo unidades de decodificación de video, motores de procesamiento de imágenes y aceleradores de inferencia [3].

Con el objetivo de cerrar esta brecha, han surgido plataformas especializadas para el desarrollo de pipelines de análisis de video en tiempo real sobre GPU. Entre ellas, el SDK DeepStream de NVIDIA se ha consolidado como una solución integral para el despliegue de aplicaciones de visión por computador asistida por IA sobre hardware NVIDIA, incluyendo plataformas embebidas de la familia Jetson, al integrar de forma nativa la decodificación de video por hardware, el preprocesamiento acelerado, la inferencia mediante TensorRT (TRT) y el postprocesamiento y transmisión de resultados en tiempo real, todo dentro de un único pipeline completamente acelerado en GPU basado en el framework multimedia GStreamer [4].

Adoptar y evaluar rigurosamente este tipo de plataformas requiere, sin embargo, un caso de aplicación real que exponga las dificultades genuinas de un sistema de inferencia en producción: un modelo de deep learning ya entrenado y validado, un flujo de datos continuo con requisitos de latencia, y una arquitectura de despliegue heterogénea donde decodificación, preprocesamiento, inferencia y postprocesamiento deben coexistir eficientemente. El diagnóstico óptico de llamas mediante pirometría de hollín constituye un caso de aplicación idóneo para este propósito. La estimación de la temperatura del hollín es una variable termodinámica central en sistemas de combustión, con incidencia directa en la liberación de calor, la eficiencia del proceso y la formación de contaminantes; su reconstrucción a partir de imágenes RGB, sin embargo, constituye un problema inverso mal condicionado, dado que la radiación observada corresponde a una proyección óptica integrada del volumen de la llama, afectada por fenómenos de autoabsorción. Trabajos previos han demostrado que arquitecturas de redes neuronales convolucionales (CNN, *Convolutional Neural Networks*) como U-Net y Attention U-Net pueden aprender esta relación no lineal directamente desde datos sintéticos y experimentales, superando tanto métodos numéricos clásicos como enfoques basados en redes neuronales más simples [5].

Sin embargo, la implementación práctica de estos modelos introduce desafíos que van más allá de la precisión de la estimación, y que son representativos del problema sistémico descrito al inicio de esta sección: en implementaciones orientadas principalmente a la precisión del modelo y la validación metodológica, las etapas de preprocesamiento y postprocesamiento suelen acumular latencias que dificultan o imposibilitan la operación en tiempo real sobre plataformas embebidas. Este trabajo toma la pirometría de hollín como caso de aplicación representativo para abordar un problema de naturaleza más general: la optimización de pipelines de visión por computador en tiempo real sobre hardware GPU embebido, mediante el diseño, implementación y evaluación comparativa de dos configuraciones de procesamiento de flujos de video. Este enfoque da continuidad, además, a investigaciones previas desarrolladas en la Universidad Técnica Federico Santa María [1, 5, 6, 7, 8], que han explorado tanto el uso de arquitecturas CNN para pirometría de hollín como la optimización de inferencia mediante TensorRT, y sobre las cuales este trabajo se apoya como antecedente directo y punto de partida técnico.

Sobre esta base, la presente memoria busca consolidar un flujo de procesamiento eficiente, reproducible y operable en tiempo casi real, contribuyendo tanto al avance metodológico en la optimización de pipelines de visión por computador sobre hardware embebido, como a la comprensión práctica de

los desafíos de ingeniería asociados a dicha optimización, empleando el diagnóstico óptico de llamas como escenario de validación concreto y de relevancia científica propia.

1.2. Planteamiento del problema

Como se señaló en la sección anterior, las ineficiencias asociadas a implementaciones genéricas, en principio marginales en sistemas con GPU de alto rendimiento y recursos abundantes, se vuelven determinantes en plataformas embebidas, donde memoria, ancho de banda y potencia de cómputo son limitados y compartidos entre múltiples etapas del pipeline.

La superación de estas limitaciones requiere una reingeniería del pipeline que traslade el cómputo a la GPU de forma integral, eliminando los cuellos de botella asociados a transferencias y sincronizaciones entre procesadores, y aprovechando capacidades de hardware especializado para decodificación de video, motores de preprocesamiento de imágenes y aceleradores de inferencia, disponibles en plataformas como Jetson. Esto implica, sin embargo, un cambio de paradigma de implementación, desde scripts Python de propósito general hacia pipelines multimedia basados en GStreamer y plugins especializados como los provistos por DeepStream, lo que introduce su propia complejidad de diseño, construcción y verificación: la extensión de plugins mediante código de bajo nivel para casos de uso no convencionales, la gestión de buffers y metadatos a través de múltiples etapas asíncronas, y la necesidad de establecer protocolos de medición que permitan comparar de forma rigurosa el rendimiento de un sistema así construido contra una implementación de referencia.

Para desarrollar, verificar y evaluar esta reingeniería sobre un caso concreto y no meramente sintético, este trabajo adopta como escenario de aplicación la pirometría de hollín en llamas laminares, introducida en la sección anterior. Este escenario resulta apropiado para el propósito metodológico del trabajo por dos razones. Primero, involucra un pipeline de procesamiento representativo del problema general: adquisición continua de video, preprocesamiento no trivial de la imagen, inferencia mediante un modelo de regresión densa (arquitecturas U-Net y Attention U-Net) y postprocesamiento para la reconstrucción y visualización del resultado. Segundo, dispone de una línea de trabajo previa sobre la cual construir, ya que, como se mencionó anteriormente, la capacidad de dichas arquitecturas para aproximar la relación entre observaciones ópticas y temperatura del hollín ya ha sido validada en estudios anteriores [1].

El problema central que aborda esta memoria no radica, por tanto, en la capacidad de los modelos de deep learning para estimar campos de temperatura, sino en cómo diseñar, implementar y evaluar un pipeline de procesamiento de video completo que opere de manera eficiente, confiable y continua sobre hardware embebido, utilizando como caso de aplicación la pirometría de hollín y como punto de referencia una implementación en Python que sirva de baseline reproducible para la comparación.

1.3. Propuesta de la solución

Para abordar el problema descrito, este trabajo propone el diseño e implementación comparativa de dos configuraciones de pipeline para el análisis de flujos de video orientado a pirometría de hollín en llamas laminares.

La primera configuración corresponde a un **pipeline de referencia (baseline)** implementado en Python, con ejecución predominantemente en CPU. Este pipeline articula las etapas de lectura de frames, preprocesamiento, inferencia mediante PyTorch y postprocesamiento de manera secuencial, representando una implementación directa y reproducible del flujo de trabajo de análisis desarrollado en trabajos previos [1, 5, 8]. Su construcción y estabilización constituyen por sí mismas una contribución relevante, ya que permiten establecer un punto de comparación bien definido y confiable.

La segunda configuración corresponde a un **pipeline optimizado** construido sobre el SDK NVIDIA DeepStream, con ejecución íntegramente acelerada en GPU sobre una plataforma Jetson embebida. En este pipeline, todas las etapas se ejecutan de forma encadenada dentro del framework GStreamer, manteniendo los datos en memoria GPU y eliminando gran parte de las transferencias innecesarias entre procesadores. Los resultados pueden ser transmitidos en tiempo real mediante un sink UDP (User Datagram Protocol), habilitando su integración con sistemas externos de visualización o registro, o bien almacenados de forma persistente en disco mediante un sink de archivo.

En ambos casos se utilizan los mismos modelos CNN U-Net y Attention U-Net ya entrenados y validados en trabajos previos, lo que permite centrar el análisis en el comportamiento del pipeline como sistema completo, más allá de las propiedades del modelo aislado. La comparación entre ambas configuraciones se realiza en términos de latencia end-to-end, throughput, uso de memoria, estabilidad de la ejecución y coherencia de los resultados de estimación de temperatura.

El énfasis de este trabajo recae en la implementación, evaluación y validación del proceso de optimización del pipeline completo sobre hardware especializado, analizando el impacto de cada etapa en el desempeño global del sistema y estableciendo una base técnica generalizable más allá del caso de aplicación específico de pirometría de hollín para la construcción de pipelines de visión por computador en tiempo real sobre plataformas GPU embebidas.

1.4. Objetivos

1.4.1. Objetivo general

Diseñar, implementar y evaluar comparativamente un pipeline optimizado de procesamiento de video en tiempo casi real sobre GPU embebida, empleando como caso de aplicación la estimación de la temperatura del hollín en llamas laminares a partir de imágenes RGB mediante DNN.

1.4.2. Objetivos específicos

- Construir y estabilizar un pipeline de referencia (baseline) en Python para el análisis de flujos de video de llamas laminares, articulando las etapas de preprocesamiento, inferencia y postprocesamiento de manera reproducible.
- Implementar un pipeline optimizado sobre el SDK NVIDIA DeepStream en plataforma Jetson embebida, integrando decodificación de video por hardware, preprocesamiento acelerado en GPU, inferencia mediante TensorRT y postprocesamiento y transmisión de resultados en tiempo real.
- Evaluar y comparar el desempeño de ambas configuraciones en términos de latencia end-to-end, throughput, uso de memoria y coherencia de los resultados, considerando las restricciones propias de plataformas embebidas.
- Identificar las limitaciones actuales del enfoque propuesto y delinear líneas de trabajo futuro orientadas a la generalización del pipeline optimizado hacia otros dominios de aplicación en visión por computador en tiempo real.

1.5. Contribuciones

Las principales contribuciones de esta memoria son las siguientes:

- El diseño e implementación de un pipeline completo sobre NVIDIA DeepStream en plataforma Jetson embebida, que integra de forma nativa todas las etapas del procesamiento de video desde la decodificación hasta la transmisión de resultados, con ejecución íntegramente acelerada en GPU, incluyendo la extensión de plugins DeepStream mediante kernels CUDA (Compute Unified Device Architecture) propios para casos de uso no convencionales dentro de dicho ecosistema.
- La construcción y estabilización de un pipeline de referencia en Python, que consolida y extiende implementaciones previas [8] en un flujo de trabajo reproducible y documentado, sirviendo como línea base metodológicamente rigurosa para la evaluación comparativa.
- Un análisis comparativo sistemático del desempeño entre ambos pipelines, considerando métricas relevantes para aplicaciones de procesamiento de video en tiempo casi real sobre hardware embebido, junto con un protocolo de medición diseñado específicamente para aislar y atribuir correctamente estos fenómenos en un pipeline con etapas paralelas y asíncronas.
- Un conjunto documentado de decisiones de ingeniería, hallazgos experimentales y limitaciones estructurales encontradas durante el desarrollo de un pipeline no convencional sobre DeepStream, de utilidad para trabajos futuros que busquen adaptar dicho framework a casos de uso fuera de su ámbito de diseño original (detección y segmentación de objetos).
- La validación de la estimación de temperatura del hollín en llamas laminares mediante CNN como caso de aplicación concreto para la optimización de pipelines propuesta, dando continuidad a investigaciones previas sobre pirometría de hollín asistida por deep learning [1, 5, 8].

El código desarrollado para las implementaciones descritas, incluyendo el pipeline DeepStream, el baseline Python y los scripts de evaluación de latencia, throughput y memoria, está disponible públicamente en un repositorio de acceso abierto [9].

1.6. Alcances y limitaciones

El presente trabajo se desarrolla bajo un conjunto de alcances y limitaciones que delimitan su enfoque, condicionan el diseño metodológico y definen su aplicabilidad en escenarios prácticos. Dado que el foco de esta memoria es la optimización del pipeline de procesamiento, varios de estos alcances corresponden a las condiciones bajo las cuales se evaluó dicho pipeline, más que a limitaciones del enfoque de optimización propuesto.

En primer lugar, la evaluación experimental utiliza como caso de aplicación la estimación de la temperatura del hollín en llamas laminares, caracterizadas por su estabilidad, simetría y reproducibilidad.

En cuanto a la instrumentación, el trabajo se limita al uso de cámaras RGB convencionales, sin incorporar sensores hiperspectrales, cámaras multiespectrales ni sistemas ópticos especializados. Las secuencias de video utilizadas en la evaluación del sistema fueron capturadas previamente en condiciones de laboratorio controladas, lo que permite centrar el análisis en el comportamiento del pipeline de procesamiento. Quedan fuera del alcance técnicas que dependan de información espectral de alta resolución o configuraciones experimentales complejas.

Desde el punto de vista del modelado, se asume el uso de arquitecturas CNN previamente entrenadas, en particular U-Net y Attention U-Net, cuya definición, entrenamiento y validación funcional han sido abordados en trabajos previos [5, 1]. El alcance de esta memoria no incluye el diseño de nuevas arquitecturas ni el reentrenamiento de los modelos. El foco se sitúa en el comportamiento del pipeline completo durante la ejecución, verificando que los modelos optimizados conserven coherencia con respecto a las implementaciones de referencia.

En términos de implementación computacional, el pipeline optimizado se desarrolla específicamente para la plataforma NVIDIA Jetson y el ecosistema DeepStream. No se analizan otras herramientas de inferencia ni plataformas de hardware distintas. La evaluación comparativa se realiza entre el pipeline Python en CPU y el pipeline DeepStream en GPU Jetson, sin extenderse a comparaciones con otras familias de hardware acelerado.

Respecto a la evaluación de desempeño, las métricas consideradas se centran en latencia end-to-end, throughput, uso de memoria y coherencia de los resultados de estimación de temperatura. No se aborda en profundidad la evaluación de robustez frente a variaciones experimentales extremas ni condiciones fuera del dominio de entrenamiento, las cuales se reconocen como limitaciones inherentes a los modelos basados en datos.

Finalmente, se reconoce que aspectos como la ampliación del conjunto de métricas, la evaluación bajo condiciones dinámicas de llama, la incorporación de nuevas arquitecturas de modelo y la integración con sistemas de control o monitoreo en línea quedan fuera del alcance de esta memoria y se proponen como líneas de trabajo futuro.

1.7. Organización de la Tesis

El contenido de esta memoria se organiza de la siguiente manera:

- El Capítulo 2 desarrolla el marco teórico, abordando los fundamentos necesarios para comprender el trabajo realizado, incluyendo los fundamentos del aprendizaje profundo, las arquitecturas CNN, U-Net y de GPU embebida, la optimización de la inferencia mediante TensorRT y el SDK NVIDIA DeepStream como plataforma de integración de pipelines en tiempo real.
- El Capítulo 3 describe la metodología empleada, empezando por describir el entorno de operación y el caso de aplicación de este trabajo, detallando también los conjuntos de datos utilizados y las arquitecturas de los modelos implementados, para luego ahondar en el diseño del pipeline de referencia y la construcción del pipeline DeepStream optimizado, así como las métricas y procedimientos de evaluación considerados.
- El Capítulo 4 presenta los resultados obtenidos a partir de los experimentos realizados, analizando el desempeño comparativo de ambos pipelines en términos de precisión, latencia, throughput y eficiencia computacional.
- El Capítulo 5 expone las conclusiones del trabajo, discutiendo los principales hallazgos, las limitaciones identificadas y posibles líneas de trabajo futuro.

MARCO TEÓRICO

Este capítulo presenta los fundamentos teóricos y técnicos que sustentan el desarrollo de esta tesis. Se introducen los conceptos esenciales asociados al procesamiento de imágenes y video mediante redes DNN y arquitecturas de regresión densa, junto con los principios de aceleración por GPU y los modelos de ejecución subyacentes, con énfasis en CUDA y en las consideraciones arquitectónicas relevantes para aplicaciones de inferencia. A continuación, se describe el rol de TensorRT como inference engine optimizado, así como el uso de formatos de representación intermedia basados en grafos computacionales para desacoplar las etapas de entrenamiento y despliegue. Finalmente, se introducen los fundamentos de construcción de pipelines multimedia mediante GStreamer y NVIDIA DeepStream como framework de integración para pipelines de inferencia en tiempo real. En conjunto, este capítulo establece el marco conceptual necesario para comprender las decisiones metodológicas y técnicas desarrolladas en los capítulos posteriores.

2.1. Aprendizaje profundo para regresión densa en visión por computador

El aprendizaje automático ha emergido como una alternativa a los enfoques físico-analíticos tradicionales para la estimación de campos continuos a partir de observaciones ópticas, en problemas donde la relación entre la señal medida y la variable física de interés es altamente no lineal o carece de una formulación analítica robusta. En estos escenarios, un modelo de deep learning permite reformular la estimación como un proceso de inferencia directa a partir de datos, evitando la resolución explícita de modelos físicos complejos y la inversión numéricamente inestable de ecuaciones integrales [10, 11, 12].

Esta estrategia resulta particularmente atractiva en escenarios donde el modelo físico completo es difícil de formular o resolver de manera robusta, y ha demostrado ser efectiva en múltiples dominios de diagnóstico óptico y regresión de campos físicos [12, 13, 14]. El caso de aplicación evaluado en este trabajo, la pirometría óptica de hollín en llamas laminares, constituye un ejemplo representativo de este tipo de problema, y se retoma con mayor detalle físico en el Capítulo de Metodología.

2.1.1. Redes neuronales y regresión de variables continuas

Las redes neuronales artificiales (ANN, *Artificial Neural Network*) constituyen una familia de modelos computacionales diseñados para aproximar funciones no lineales de alta complejidad mediante la composición de transformaciones simples organizadas en capas. Cada capa aplica una combinación lineal de sus entradas seguida de una función de activación no lineal, lo que permite al modelo representar relaciones altamente no lineales entre las variables de entrada y salida.

En problemas de regresión de campos físicos como el del caso de pirometría del hollín que cubre este trabajo, las redes neuronales se emplean para tareas de regresión densa: la entrada es una imagen RGB de la llama y la salida es un campo bidimensional donde cada elemento representa una estimación local de la temperatura del hollín. A diferencia de los problemas de clasificación, en los cuales la salida pertenece a un conjunto discreto de etiquetas, la regresión densa exige que el modelo produzca un tensor de valores numéricos coherentes con la naturaleza física de la magnitud estimada, preservando continuidad, suavidad y consistencia espacial [12, 10].

La regresión de campos físicos impone desafíos adicionales respecto a la regresión convencional: el modelo debe capturar correlaciones espaciales de múltiples escalas, mantener coherencia entre regiones adyacentes y evitar la introducción de artefactos no físicos en las predicciones. Estas exigencias influyen directamente en la elección de la arquitectura de red, en la representación de los datos y en la definición del objetivo de entrenamiento.

Desde la perspectiva del pipeline de procesamiento, la naturaleza tensorial de la salida tiene implicancias computacionales directas. A diferencia de un modelo que produce un escalar o un vector de pocas dimensiones, un modelo de regresión densa genera por cada frame de entrada un tensor de dimensiones comparables a la imagen original, que debe ser transferido, postprocesado y convertido en una representación útil (como un mapa de calor de temperatura) antes de su visualización o transmisión. Esta característica determina en parte los requisitos de memoria y ancho de banda del pipeline de inferencia, y debe considerarse al diseñar un sistema eficiente sobre plataformas con recursos limitados.

2.1.2. Redes neuronales convolucionales para imágenes

Las CNN constituyen una clase de modelos especialmente diseñados para el procesamiento de datos con estructura espacial, como imágenes. Su eficacia se basa en la explotación explícita de correlaciones locales y en el uso de mecanismos de compartición de parámetros, lo que les permite modelar patrones espaciales complejos de manera eficiente tanto desde el punto de vista representacional como computacional.

El bloque fundamental de una CNN es la operación de convolución, mediante la cual un conjunto de filtros o kernels se desplaza sobre la imagen de entrada realizando productos punto locales. Cada kernel actúa como un detector de patrones específicos (bordes, transiciones suaves de intensidad, estructuras texturales), generando mapas de activación que resaltan la presencia de dichos patrones en distintas posiciones espaciales. A través de la composición de múltiples capas convolucionales, las CNN construyen representaciones jerárquicas de la información, donde las capas iniciales capturan características locales de baja complejidad y las capas más profundas integran información de regiones espaciales más amplias.

En el caso de aplicación de este trabajo para pirometría, las imágenes RGB de llamas contienen información radiométrica distribuida espacialmente, influenciada por emisión térmica, autoabsorción y proyección óptica. Las CNN permiten capturar estas relaciones espaciales de forma implícita, preservando la topología del dominio y evitando la necesidad de describir explícitamente cada uno de los mecanismos físicos subyacentes. En comparación con redes completamente conectadas, las CNN reducen significativamente el número de parámetros entrenables, lo que mejora la eficiencia del entrenamiento y disminuye el riesgo de sobreajuste [1]. Además, operan naturalmente sobre tensores multidimensionales, lo que las hace directamente compatibles con imágenes multicanal como las imágenes RGB.

En el ámbito del diagnóstico de combustión, investigaciones previas han aplicado CNN a la estimación de temperatura y fracción volumétrica de hollín a partir de señales de emisión y absorción, evidenciando que estas redes son capaces de aprender relaciones complejas entre imágenes proyectadas y campos físicos subyacentes [5]. Estos avances han sentado las bases para el uso de arquitecturas

convolucionales más especializadas, como U-Net y sus variantes con mecanismos de atención.

2.1.3. Fundamentos de U-Net y Attention U-Net

La arquitectura U-Net fue introducida originalmente para tareas de segmentación semántica en imágenes biomédicas, pero su estructura la hace igualmente adecuada para problemas de regresión densa en los cuales la salida corresponde a un campo continuo definido en el mismo dominio espacial que la entrada [15]. Esta formulación es directamente aplicable a la pirometría de hollín, donde se busca reconstruir mapas bidimensionales de temperatura a partir de imágenes ópticas proyectadas [10].

U-Net se basa en una arquitectura codificador-decodificador completamente convolucional. El codificador está compuesto por bloques convolucionales seguidos de operaciones de reducción de resolución mediante max-pooling, que disminuyen progresivamente la resolución espacial mientras aumenta la profundidad de las representaciones, permitiendo a la red capturar información contextual de mayor escala. El decodificador invierte este proceso mediante operaciones de upsampling o convoluciones transpuestas, reconstruyendo la salida hasta alcanzar la resolución original de la imagen de entrada. Un elemento distintivo de U-Net son las conexiones de salto (*skip connections*) entre niveles simétricos del codificador y el decodificador, que permiten transferir directamente características de alta resolución espacial, combinando información de bajo nivel (detalles locales, bordes, gradientes suaves) con información de alto nivel (contexto global) y mejorando así la precisión espacial de la reconstrucción.

Las variantes de Attention U-Net incorporan mecanismos de atención que refinan el flujo de información transmitido a través de las conexiones de salto [16]. Las puertas de atención (*attention gates*) modulan las características transmitidas asignando mayores pesos a regiones informativas y suprimiendo aquellas dominadas por ruido o irrelevantes para la estimación, combinando información contextual proveniente del decodificador con detalles locales del codificador. En pirometría óptica, esto se traduce en una mayor sensibilidad a regiones de la imagen asociadas a la presencia de hollín y a zonas térmicamente activas. Estudios previos han mostrado que estas variantes mejoran la estabilidad y precisión de las reconstrucciones, particularmente bajo condiciones de alta variabilidad espacial o información espectral limitada [5].

Desde la perspectiva del pipeline de procesamiento, es relevante destacar que tanto U-Net como Attention U-Net operan de forma independiente sobre cada imagen de entrada, sin mantener estado entre ejecuciones consecutivas. En el contexto de un sistema de análisis en tiempo real sobre un flujo de video, esto implica que el modelo procesa cada frame de manera aislada: la inferencia se ejecuta repetidamente sobre la secuencia de imágenes, produciendo para cada una un tensor de salida bidimensional que representa el mapa de temperatura estimado. Este tensor debe ser posteriormente denormalizado y convertido en una representación visual antes de su transmisión o almacenamiento. La eficiencia con que se ejecuta este ciclo determina en última instancia el throughput y la latencia del sistema completo, motivando las decisiones de diseño del pipeline que se describen en los capítulos siguientes.

2.2. Computación acelerada por GPU

El crecimiento sostenido en la complejidad de los modelos de deep learning, junto con el aumento en la resolución y volumen de los datos procesados, ha impulsado la adopción de arquitecturas de cómputo acelerado como una necesidad práctica más que como una optimización opcional. Las GPUs se han consolidado como la plataforma predominante tanto para el entrenamiento como para la inferencia de DNN, debido a su elevada capacidad de paralelismo y a su eficiencia en la ejecución

de operaciones numéricas intensivas[17].

A diferencia de otras alternativas de hardware especializado, las GPU presentan un equilibrio favorable entre flexibilidad, accesibilidad y rendimiento, lo que ha favorecido su adopción masiva en aplicaciones de aprendizaje automático. En particular, las GPU de Nvidia han establecido un ecosistema ampliamente soportado, que integra hardware, modelos de programación y bibliotecas optimizadas, facilitando su uso en investigación y desarrollo aplicado.

En aplicaciones de pirometría basada en visión computacional, donde la inferencia se realiza a partir de imágenes y puede estar sujeta a restricciones de tiempo casi real, la computación acelerada por GPU resulta especialmente relevante. La posibilidad de ejecutar modelos convolucionales profundos con baja latencia habilita su uso en entornos experimentales, sistemas de monitoreo continuo y plataformas embebidas, donde el rendimiento computacional y la eficiencia energética son factores críticos.

2.2.1. Arquitectura general de GPU

Las GPU presentan una arquitectura fundamentalmente distinta a la de las CPU (Central Processing Unit). Mientras que las CPU están diseñadas para optimizar la ejecución secuencial de tareas heterogéneas con baja latencia, las GPU priorizan el alto rendimiento agregado mediante la ejecución concurrente de un gran número de operaciones simples. Esta diferencia arquitectónica refleja una filosofía de diseño orientada al paralelismo masivo de datos, característica central de las aplicaciones de deep learning.

En las GPU modernas, el chip se organiza jerárquicamente en varios niveles. A alto nivel, la GPU se divide en *Graphics Processing Clusters* (GPC), cada uno de los cuales agrupa múltiples unidades de procesamiento. Dentro de estos clusters se encuentran los *Streaming Multiprocessors* (SM), que constituyen la unidad básica de ejecución paralela. Cada SM integra numerosos núcleos de cómputo, junto con unidades especializadas para operaciones aritméticas, control de flujo y acceso a memoria.

El modelo de ejecución de las GPU se basa en la creación de un gran número de hilos de ejecución (threads), los cuales realizan instancias independientes de una misma secuencia de instrucciones. Estos hilos se agrupan internamente en conjuntos denominados warps, típicamente de 32 hilos, que se ejecutan de manera sincronizada bajo un esquema SIMT (*Single Instruction, Multiple Threads*). Todos los hilos de un warp ejecutan la misma instrucción en cada ciclo de reloj, aunque sobre datos distintos, lo que permite un aprovechamiento eficiente del paralelismo de datos presente en operaciones vectoriales y matriciales.

La jerarquía de memoria de la GPU está diseñada para maximizar el ancho de banda y minimizar los cuellos de botella asociados al movimiento de datos. Esta jerarquía incluye memoria global, memoria compartida a nivel de SM, memoria constante y registros. La memoria compartida, en particular, permite intercambios rápidos entre hilos pertenecientes a un mismo SM, favoreciendo la reutilización de datos y reduciendo accesos costosos a memoria global. En muchas aplicaciones de deep learning, el rendimiento está limitado por el acceso a memoria más que por la capacidad de cómputo puro.

Adicionalmente, las GPU modernas incorporan pipelines altamente optimizados y unidades especializadas para acelerar operaciones específicas, como las multiplicaciones de matrices en precisión reducida, ampliamente utilizadas durante la inferencia de modelos de deep learning. Estas capacidades permiten alcanzar altos niveles de rendimiento con un consumo energético relativamente contenido, especialmente cuando se comparan con soluciones de propósito general.

En conjunto, esta arquitectura orientada al paralelismo masivo, basada en la ejecución concurrente de hilos organizados en warps y en una jerarquía de memoria de alto ancho de banda, convierte a las GPU en una plataforma especialmente adecuada para la ejecución eficiente de modelos convo-

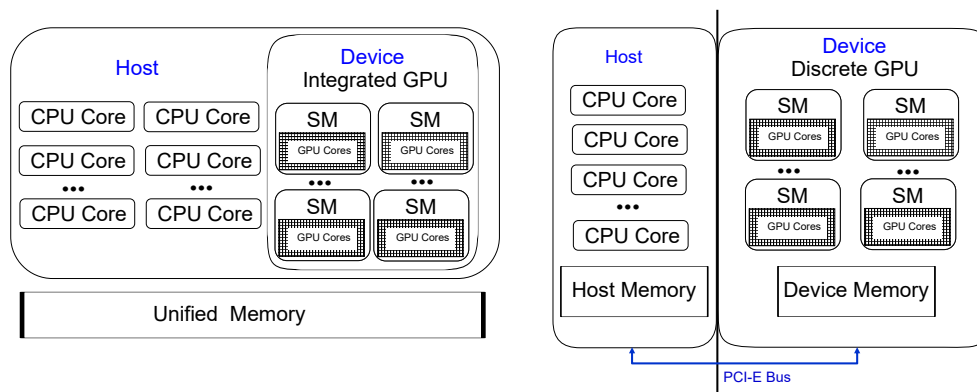


Fig. 2.1: Diagrama de comunicación entre host y device para GPU integrada (izquierda) y GPU discreta (derecha).

lucionales profundos.

2.2.2. Modelo de programación CUDA

CUDA es el modelo de programación desarrollado por NVIDIA para explotar de forma explícita el paralelismo masivo de sus GPU [18]. Este modelo habilita el desarrollo de aplicaciones GPGPU (*General-Purpose Computing on GPUs*), permitiendo que algoritmos altamente paralelizables, como los asociados al deep learning, se ejecuten de manera significativamente más eficiente que en arquitecturas basadas exclusivamente en CPU.

El modelo CUDA concibe el sistema como un entorno de cómputo heterogéneo en el cual coexisten dos dominios claramente diferenciados: el *host*, correspondiente a la CPU y su memoria asociada, y el *device*, correspondiente a la GPU y su jerarquía de memoria. La CPU actúa como coordinadora del flujo de ejecución, gestionando la transferencia de datos y la invocación de funciones paralelas, mientras que la GPU funciona como un coprocesador especializado encargado de las tareas computacionalmente intensivas. Las funciones paralelizables que se ejecutan en la GPU se denominan *kernels*, y su invocación desde el host se conoce como *kernel launch*.

La comunicación entre host y device depende de la arquitectura del sistema. Tal como se ilustra en la Figura 2.1, en sistemas con GPU discretas la GPU se encuentra en un chip separado de la CPU y se comunica a través de buses de interconexión como PCIe (*Peripheral Component Interconnect Express*). En contraste, en plataformas con GPU integradas como NVIDIA Jetson, CPU y GPU residen en un mismo chip, lo que en términos generales reduce la necesidad de transferencias explícitas entre espacios de memoria diferenciados; las implicancias concretas de este esquema de memoria unificada se detallan en la Sección 2.2.4.

Desde el punto de vista lógico, CUDA organiza la ejecución paralela mediante una jerarquía de abstracciones. Cada kernel se ejecuta sobre un gran número de hilos (*threads*), los cuales se agrupan en bloques (*blocks*), y estos, a su vez, se organizan en una grilla (*grid*). Esta estructura permite escalar la ejecución del mismo código sobre miles o millones de hilos de manera transparente para el programador. Aunque todos los hilos ejecutan el mismo código, cada uno opera sobre diferentes elementos de datos, siguiendo el paradigma de paralelismo de datos característico de las GPU. La correspondencia entre esta jerarquía lógica y el hardware es gestionada por el sistema de ejecución de CUDA: los bloques son asignados dinámicamente a los SM, y los hilos dentro de cada bloque se ejecutan agrupados en warps.

Un programa CUDA típico sigue una secuencia de etapas: asignación de memoria en el device, transferencia de datos desde el host al device, ejecución de uno o más kernels, copia de resultados de vuelta al host y liberación de recursos. En una ejecución secuencial, estas etapas se suceden de forma bloqueante, lo que implica que la CPU permanece inactiva mientras espera la finalización de cada operación en la GPU. Para evitar este desperdicio de recursos, CUDA provee mecanismos de ejecución asíncrona basados en *streams*.

Un CUDA stream es una cola ordenada de operaciones GPU (lanzamientos de kernels, transferencias de memoria y eventos de sincronización) que se ejecutan en el orden en que fueron puestas en cola. Las operaciones en streams distintos pueden, en cambio, ejecutarse de manera concurrente, sujeto a la disponibilidad de recursos del hardware [18]. Este mecanismo permite solapar la transferencia de un lote de datos desde el host con la ejecución del kernel sobre el lote anterior, un patrón conocido como *pipeline de doble buffer*, que mantiene la GPU continuamente ocupada y maximiza el throughput del sistema. Para que la transferencia asíncrona sea efectiva, la memoria del host debe estar reservada como *pinned memory* (memoria bloqueada en páginas físicas), lo que permite al hardware de transferencia DMA (*Direct Memory Access*) operar sin intervención de la CPU. La sincronización entre streams puede gestionarse mediante *CUDA events*, que actúan como marcadores de progreso insertables en cualquier punto de la cola de un stream y consultables desde otro stream o desde el host.

En aplicaciones de deep learning, como la inferencia de CNN sobre flujos de video, el uso de streams permite solapar la preparación del siguiente frame con la inferencia sobre el frame actual, reduciendo la latencia efectiva del sistema. En plataformas Jetson, donde la memoria es unificada y las transferencias explícitas son innecesarias, los streams conservan su utilidad para solapar etapas de cómputo interno: preprocesamiento, inferencia y postprocesamiento pueden organizarse en streams separados con dependencias controladas por eventos, maximizando la utilización de los recursos disponibles.

Si bien CUDA ofrece un control detallado sobre el hardware, la programación directa en este modelo resulta compleja. Por esta razón, la mayoría de las aplicaciones de deep learning se implementan mediante frameworks de alto nivel que abstraen los detalles del modelo CUDA. No obstante, el desempeño final del sistema sigue estando fuertemente condicionado por cómo estas abstracciones se traducen internamente en kernels, planificación de ejecución y uso de memoria. Adicionalmente, aspectos clave del comportamiento de las GPU como la planificación interna de tareas, el context switching y ciertos mecanismos de sincronización no están completamente documentados [19], lo que introduce un carácter parcialmente opaco que obliga en muchos casos a recurrir a evaluaciones empíricas.

2.2.3. Gestión de memoria en GPU

La organización de los datos en memoria GPU es uno de los factores que mayor impacto tiene sobre el rendimiento de los kernels CUDA, y resulta particularmente relevante en aplicaciones de visión por computador donde los datos son tensores multidimensionales. Comprender los modelos de memoria disponibles y sus implicancias en el acceso y la transferencia de datos resulta entonces esencial tanto en el diseño de pipelines eficientes sobre plataformas con recursos limitados como en su programación efectiva.

Reservas de memoria

La forma más directa de reservar memoria en el device es mediante `cudaMalloc`, que asigna un bloque contiguo de bytes en la memoria global de la GPU. Este esquema, denominado *memoria lineal*, es apropiado para estructuras unidimensionales o cuando el acceso secuencial a los datos está

garantizado. Sin embargo, cuando se trabaja con arreglos bidimensionales (como imágenes o mapas de características), la alineación de los accesos de memoria se convierte en un factor crítico.

Las GPU alcanzan su máximo ancho de banda de memoria cuando los hilos de un mismo warp acceden a posiciones de memoria contiguas y alineadas a múltiplos de un tamaño de transacción determinado por el hardware (típicamente 32, 64 o 128 bytes, según la arquitectura). Esta propiedad, conocida como *memory coalescing*, permite que los accesos de los 32 hilos de un warp se sirvan en una única transacción de memoria global en lugar de 32 transacciones independientes. Cuando las filas de un arreglo bidimensional no comienzan en direcciones alineadas, el coalescing se rompe, aumentando la cantidad de transacciones requeridas y degradando el rendimiento.

Para resolver este problema, CUDA provee `cudaMallocPitch`, que asigna un arreglo bidimensional añadiendo automáticamente relleno (*padding*) al final de cada fila para garantizar que el inicio de cada fila cumpla los requisitos de alineación del hardware [18]. El valor retornado en el parámetro `pitch` corresponde al ancho efectivo de la fila en bytes, incluyendo el padding, y debe usarse en lugar del ancho lógico para calcular las direcciones de acceso:

$$\text{T* pElement} = (\text{T*})((\text{char*})\text{BaseAddress} + \text{Row} \times \text{pitch}) + \text{Column} \quad (2.2.1)$$

Como consecuencia, el `pitch` puede ser estrictamente mayor que `width_bytes` cuando el hardware añade padding. Las transferencias de datos bidimensionales con `pitch` se realizan mediante `cudaMemcpy2D`, que requiere especificar tanto el `pitch` de origen como el de destino para copiar correctamente cada fila de forma independiente.

Layouts de tensor

Más allá de la alineación de filas individuales, la organización lógica de los datos en un tensor multidimensional tiene un impacto significativo sobre la eficiencia de los kernels de convolución y de las operaciones de preprocesamiento y postprocesamiento. En el contexto del deep learning, los tensores de imagen o de mapas de características se indexan habitualmente con cuatro dimensiones: batch (N), canales (C), altura (H) y ancho (W). El orden en que estas dimensiones se almacenan en memoria, denominado *layout* del tensor, determina cuáles elementos son contiguos en memoria y, por tanto, qué patrones de acceso resultan eficientes para el hardware.

Los dos layouts predominantes son NCHW y NHWC. En el layout **NCHW** (también denominado *channels-first*), los datos se organizan de modo que los píxeles de un mismo canal son contiguos en memoria. Este formato es el utilizado de forma predeterminada por PyTorch y resulta eficiente para ciertos kernels de convolución donde la iteración sobre posiciones espaciales dentro de un canal se realiza de forma secuencial. En el layout **NHWC** (también denominado *channels-last*), los valores de todos los canales para una misma posición espacial (h, w) son contiguos en memoria. Este formato favorece el coalescing cuando múltiples hilos procesan simultáneamente distintas posiciones espaciales de la misma muestra, y es el formato preferido por las Tensor Cores de las GPU NVIDIA modernas para operaciones de convolución [20].

Las fórmulas de indexación para ambos layouts, dado un tensor de dimensiones $[N, C, H, W]$, son:

$$\text{NCHW: } \text{idx}(n, c, h, w) = n \cdot CHW + c \cdot HW + h \cdot W + w \quad (2.2.2)$$

$$\text{NHWC: } \text{idx}(n, c, h, w) = n \cdot HWC + h \cdot WC + w \cdot C + c \quad (2.2.3)$$

La elección del layout no es trivial en el contexto de un pipeline completo, ya que distintas etapas pueden operar con preferencias distintas. Los modelos entrenados en PyTorch utilizan NCHW

internamente, mientras que los plugins de preprocesamiento de DeepStream pueden producir buffers en NHWC, y TensorRT puede requerir o preferir uno u otro dependiendo de las optimizaciones aplicadas. Una inconsistencia de layouts entre etapas consecutivas del pipeline obliga a insertar operaciones de transposición explícita del tensor, cuyo costo computacional puede ser significativo y debe evitarse o minimizarse mediante un diseño cuidadoso de los buffers compartidos entre etapas.

En el caso concreto de este trabajo, la salida del modelo, un mapa de temperatura bidimensional, corresponde a un tensor de dimensiones $[1, 1, H, W]$ para inferencia con batch unitario. Aunque en este caso particular los layouts NCHW y NHWC son equivalentes (la dimensión de canales es 1), el manejo correcto del pitch y de la alineación del buffer de salida resulta igualmente relevante para su transferencia y postprocesamiento eficiente dentro del pipeline DeepStream.

2.2.4. GPUs embebidas y plataformas Jetson

Las GPU embebidas representan una evolución relevante en el ámbito de la computación acelerada, al trasladar capacidades de procesamiento paralelo hacia plataformas compactas, de bajo consumo energético y orientadas al despliegue en *edge computing*. A diferencia de las GPU discretas utilizadas en estaciones de trabajo o servidores, las soluciones embebidas priorizan la integración de componentes, la eficiencia energética y la reducción de latencia, aun cuando ello implique restricciones en términos de recursos absolutos disponibles.

La familia de plataformas NVIDIA Jetson se ha consolidado como una de las soluciones más difundidas para aplicaciones de visión artificial, robótica, sistemas autónomos y análisis de señales en tiempo cercano al real sobre hardware compacto. Estas plataformas adoptan un diseño SoC (*System-on-Chip*), integrando en un único dispositivo núcleos de CPU, GPU y aceleradores adicionales, junto con los subsistemas de memoria y comunicaciones necesarios para su operación autónoma.

Una característica arquitectónica distintiva de las plataformas Jetson, con implicancias directas sobre el diseño del pipeline de este trabajo, es la **memoria unificada**: CPU y GPU comparten el mismo espacio de memoria física, lo que elimina la necesidad de transferencias explícitas entre dominios mediante DMA, presentes en sistemas con GPU discreta. Esto tiene consecuencias prácticas importantes: los buffers de datos pueden ser escritos por la CPU y leídos directamente por kernels CUDA sin copias intermedias, reduciendo la latencia y simplificando el flujo de gestión de memoria. Sin embargo, la memoria unificada también significa que CPU y GPU compiten por el mismo ancho de banda de memoria: aunque no existe una transferencia DMA entre chips distintos, los kernels que acceden a buffers mal alineados experimentan igualmente penalizaciones de rendimiento por ruptura del coalescing, lo que hace aún más relevante la correcta alineación de los buffers y la minimización de accesos innecesarios.

Adicionalmente, las Jetson incluyen motores de hardware dedicados para la decodificación y codificación de video mediante NVDEC (*NVIDIA Decoder*) y NVENC (*NVIDIA Encoder*), respectivamente, que permiten descomprimir y comprimir flujos de video directamente en buffers de la GPU sin intervención de la CPU. Esta capacidad es aprovechada por el SDK DeepStream para construir pipelines de análisis de video completamente acelerados, donde los frames decodificados están disponibles en memoria GPU y pueden ser procesados por etapas sucesivas de preprocesamiento e inferencia sin salir del espacio de memoria del device.

No obstante, el uso de GPU embebidas impone restricciones que condicionan el diseño del sistema. Las plataformas Jetson disponen de memoria total más limitada que las GPU de escritorio (compartida entre CPU y GPU), menor capacidad de cómputo en términos absolutos y presupuestos energéticos acotados. Estas restricciones hacen que la ejecución de modelos complejos sin optimización pueda resultar inviable para operación en tiempo real, y motivan directamente el uso de herramientas como TensorRT [21] para reducir la huella computacional y de memoria de los modelos de inferencia. En este escenario, optimizar no solo la inferencia sino el pipeline completo se

convierte en un requisito para alcanzar el rendimiento necesario en la plataforma objetivo para futuras aplicaciones.

2.3. Optimización de inferencia en redes neuronales

El uso práctico de DNN no se limita al diseño y entrenamiento de modelos con alta precisión, sino que requiere considerar cómo estos modelos serán ejecutados en escenarios reales de despliegue. Cuando la inferencia debe realizarse en hardware con recursos limitados o bajo restricciones estrictas de latencia, consumo energético y uso de memoria, la eficiencia del proceso de inferencia se convierte en un aspecto central del sistema. La optimización de inferencia tiene como objetivo reducir el costo computacional y de memoria asociado a la ejecución de un modelo entrenado, manteniendo una degradación mínima o idealmente nula en la calidad de las estimaciones, mediante la reorganización y adaptación de su ejecución para aprovechar de manera más eficiente las características del hardware objetivo.

2.3.1. Entrenamiento e inferencia

El entrenamiento y la inferencia de una red neuronal corresponden a etapas conceptualmente distintas dentro del ciclo de vida de un modelo de deep learning. Durante el entrenamiento, el objetivo es ajustar los parámetros del modelo a partir de un conjunto de datos representativo, minimizando una función de pérdida mediante algoritmos de optimización iterativa que involucran el cálculo de gradientes y múltiples pasadas sobre grandes volúmenes de datos. Este proceso se realiza habitualmente mediante frameworks de alto nivel como PyTorch [22], TensorFlow [23] o Keras [24], que abstraen los detalles de bajo nivel de la ejecución en GPU.

La inferencia, en contraste, consiste en aplicar un modelo previamente entrenado a nuevas entradas sin modificar sus parámetros. Aunque algorítmicamente más simple, la inferencia está sujeta a restricciones más estrictas de latencia, consumo energético y uso de memoria, especialmente en aplicaciones en tiempo real o sobre plataformas embebidas. Si bien los frameworks de entrenamiento permiten ejecutar inferencia de manera directa, las abstracciones que facilitan el desarrollo pueden introducir sobrecostos computacionales e impedir el uso de primitivas optimizadas propias del hardware objetivo. Por esta razón, en flujos de trabajo modernos se establece una separación entre el entorno de entrenamiento y el entorno de despliegue, dando lugar a procesos de optimización de inferencia cuyo resultado es un *inference engine*: una representación optimizada del modelo que encapsula tanto su estructura como las decisiones necesarias para ejecutarlo de forma eficiente en una plataforma de hardware determinada.

2.3.2. Formatos de grafo como representación intermedia

Los formatos de grafo constituyen una representación estandarizada de modelos de machine learning que facilita su interoperabilidad entre distintos frameworks de entrenamiento y herramientas de optimización de inferencia. En este enfoque, un modelo de red neuronal se describe como un grafo computacional estático en el cual los nodos representan operaciones y las aristas representan los tensores que transportan datos entre ellas, incluyendo los parámetros del modelo ajustados durante el entrenamiento. Esta representación permite analizar la estructura del modelo, identificar dependencias entre operaciones y aplicar transformaciones sistemáticas sin depender del framework original. La Figura 2.2 ilustra un ejemplo de este tipo de representación para un modelo de red residual simple.

Dentro de esta categoría, ONNX (*Open Neural Network Exchange*) [25] es uno de los formatos de grafo más ampliamente utilizados. Define un conjunto estandarizado de operadores y una semánti-

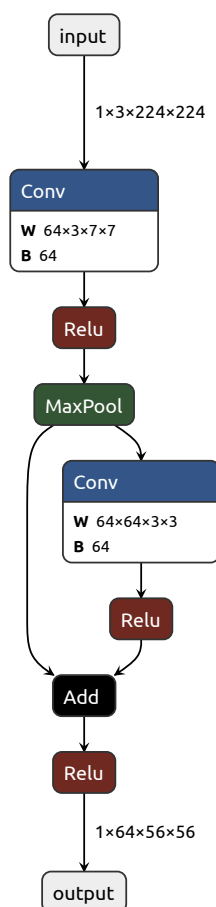


Fig. 2.2: Representación de un grafo computacional para un modelo de machine learning basado en una red residual simple a través de operaciones de convolución (conv), funciones de activación (ReLU) y pooling (MaxPool).

ca común que permite exportar modelos entrenados en frameworks como PyTorch o TensorFlow hacia una representación intermedia independiente del entorno de entrenamiento, habilitando su posterior procesamiento por inference engines especializados. En el contexto de este trabajo, ONNX actúa como el puente entre modelos entrenados en PyTorch y su optimización mediante TensorRT, permitiendo además verificar la consistencia funcional del modelo exportado antes de su despliegue.

2.3.3. Optimización de inferencia mediante TensorRT

TensorRT es un *inference optimizer* desarrollado por NVIDIA, diseñado específicamente para maximizar el rendimiento de la inferencia de DNN en GPUs de esta compañía [26]. A diferencia de los frameworks de entrenamiento, TensorRT se enfoca exclusivamente en la ejecución eficiente de modelos previamente entrenados, aplicando optimizaciones orientadas al hardware objetivo. Existe evidencia empírica sustancial que demuestra que los inference engines generados por TensorRT se encuentran entre las soluciones más eficientes para este propósito, tanto en términos de latencia como de throughput, en una amplia variedad de arquitecturas y escenarios de despliegue [27, 28, 29, 30].

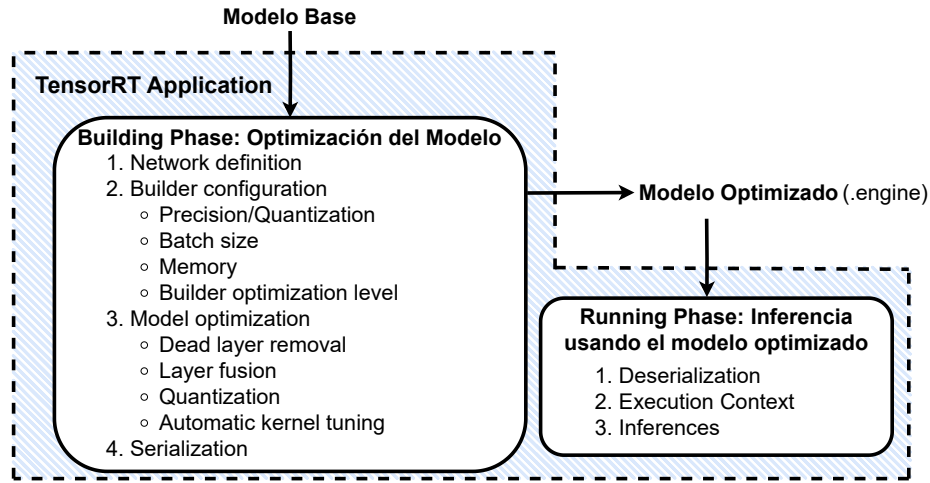


Fig. 2.3: Flujo de una TensorRT Application, desde un modelo base hasta la ejecución de inferencia optimizada.

TensorRT recibe como entrada un modelo base representado en un formato de grafo interoperable como ONNX y produce como salida un *inference engine* optimizado para una arquitectura de GPU específica. El proceso se apoya en características propietarias del hardware NVIDIA, lo que convierte a TensorRT en un software de naturaleza cerrada; no obstante, su funcionamiento general ha sido ampliamente documentado y evaluado de forma empírica en la literatura [8]. TensorRT se distribuye como un SDK que proporciona API (*Application Programming Interfaces*) tanto en C++ como en Python, y su flujo de trabajo se divide en dos etapas bien definidas: la *building phase*, en la cual se optimiza el modelo y se genera el engine, y la *running phase*, en la cual dicho engine se utiliza para ejecutar inferencias. La Figura 2.3 sintetiza este flujo.

Building phase

La *building phase* corresponde al proceso de optimización del modelo base y a la generación del inference engine. TensorRT comienza construyendo una *network definition*: una representación interna de la topología de la red que incluye tipos de capas, parámetros asociados y dimensiones de los tensores de entrada y salida. Esta representación se obtiene habitualmente importando un modelo ONNX mediante el parser integrado en TensorRT [26], aunque también es posible construirla manualmente mediante la API de bajo nivel.

A partir de la *network definition* se establece una *builder configuration* que controla el proceso de optimización. Entre los parámetros más relevantes se encuentran la precisión numérica (fp32, fp16 o int8), el batch size (estático o dinámico, con valores mínimo, óptimo y máximo), los límites de memoria durante la construcción, el nivel de agresividad en la búsqueda de kernels, y la posibilidad de definir dimensiones espaciales dinámicas [31]. La cuantización a int8 requiere además un proceso de calibración automática sobre un conjunto representativo de datos de entrada.

Con estos elementos definidos, TensorRT procede a la *optimización del modelo*, aplicando transformaciones como eliminación de capas inactivas, fusión vertical y horizontal de operaciones para reducir el número de lanzamientos de kernel [29], cuantización de pesos y activaciones, y *automatic kernel tuning*, que consiste en evaluar empíricamente distintas implementaciones de cada operación en el hardware objetivo y seleccionar la más eficiente. Este último proceso introduce una componente

no determinística: el tiempo de construcción, las condiciones del hardware y los procesos en segundo plano pueden producir inference engines con rendimiento ligeramente distinto entre ejecuciones, aun partiendo del mismo modelo base. El resultado final de la building phase es un inference engine serializable en disco (comúnmente denominado *plan*), lo que permite desacoplar completamente la etapa de optimización de la etapa de ejecución.

Running phase

La *running phase* corresponde a la utilización del engine para realizar inferencias. El plan serializado se deserializa en memoria, evitando repetir el costoso proceso de optimización, y a partir de él se crea un *execution context* que encapsula la información necesaria para ejecutar las operaciones del modelo en la GPU, incluyendo la gestión de memoria y la configuración de los kernels. Esta separación entre construcción y ejecución es especialmente relevante en plataformas embebidas, donde los recursos son limitados y el tiempo de inicialización del sistema debe minimizarse.

Durante la ejecución, el engine opera directamente sobre direcciones de memoria en la GPU mediante un mecanismo de *bindings*: un mapa ordenado de punteros a regiones contiguas de memoria global que indica exactamente dónde el engine debe leer las entradas y dónde debe escribir las activaciones de salida [26]. Este mapa debe respetar el mismo orden lógico del grafo compilado para que cada kernel acceda a la región correcta de memoria. La contigüidad de los buffers es un requisito funcional: un tensor cuyos datos no son contiguos en memoria describe sus elementos mediante saltos de stride, lo que impide que los hilos de un mismo warp realicen accesos coalescentes a memoria global. La coalescencia es determinante para el throughput efectivo de la GPU; su ausencia incrementa el número de transacciones necesarias y degrada el rendimiento independientemente de las optimizaciones aplicadas por el builder [18].

Los kernels del engine se lanzan sobre un CUDA stream: una cola FIFO (*First In, First Out*) de ejecución administrada por el scheduler del driver que despacha operaciones a los SM cuando los recursos quedan disponibles. Este modelo de ejecución es inherentemente asíncrono: la CPU no bloquea su flujo lógico mientras la GPU planifica y ejecuta los kernels, lo que permite que ambos procesadores operen en paralelo [18]. Organizar la inferencia sobre un stream dedicado evita interferencias con otras colas de ejecución CUDA presentes en el sistema, maximizando la disponibilidad de los SM para despachar los kernels fusionados del engine. Dado que la GPU no señaliza automáticamente la finalización de los kernels al host, la aplicación debe introducir una sincronización explícita del stream antes de leer los resultados o registrar tiempos de ejecución, delimitando correctamente el intervalo de medición y evitando condiciones de carrera.

El rendimiento observable en la running phase está condicionado por la *occupancy* del SM: la fracción de warps activos respecto al máximo teórico soportado por el multiprocesador. Una occupancy alta permite al scheduler ocultar latencias de memoria o dependencias aritméticas manteniendo otros warps listos para ejecución (*latency hiding*); una occupancy baja deja burbujas de inactividad en los SM que el pipeline externo no puede compensar [18]. La occupancy está determinada por las propiedades de los kernels seleccionados durante la building phase (uso de registros, tamaño de bloque, memoria compartida) y no puede modificarse en runtime. En consecuencia, el rol de la aplicación se limita a proveer datos en el formato correcto, gestionar los streams y sincronizar el host, dejando al plan compilado la responsabilidad de maximizar el paralelismo aritmético por SM.

2.4. Integración de flujos de inferencia en tiempo real

La ejecución de modelos de deep learning sobre imágenes aisladas en entornos de laboratorio constituye solo una parte del problema cuando se considera el despliegue en aplicaciones reales. En escenarios de análisis continuo, como el monitoreo óptico de llamas laminares mediante flujos

de video exigido por el caso de aplicación, el sistema debe procesar una sucesión ininterrumpida de frames produciendo resultados para cada uno dentro de un presupuesto de tiempo acotado. Esto introduce requisitos que trascienden la optimización de la inferencia aislada: la adquisición de datos, su preprocesamiento, la inferencia y el postprocesamiento deben articularse como un pipeline continuo en el que la eficiencia de cada etapa y la ausencia de cuellos de botella entre ellas determinan el rendimiento global del sistema.

En arquitecturas convencionales basadas en Python con ejecución en CPU, una parte significativa de las etapas de preprocesamiento y postprocesamiento se ejecuta en el host, lo que genera transferencias frecuentes de datos entre la memoria del host y la memoria del device y limita la utilización efectiva de la GPU. Las arquitecturas aceleradas buscan mitigar este problema desplazando la mayor cantidad posible de operaciones hacia la GPU y manteniendo los datos en memoria del device a lo largo de todo el pipeline, eliminando las transferencias innecesarias entre procesadores.

Esta sección presenta los fundamentos tecnológicos utilizados para construir dicho pipeline. Se introducen primero los conceptos fundamentales que distinguen imagen, video, stream y procesamiento en tiempo real, seguidos de GStreamer como framework de construcción de pipelines multimedia sobre el que se apoya DeepStream, y finalmente de NVIDIA DeepStream como plataforma de integración acelerada para aplicaciones de análisis de video con IA.

2.4.1. Imagen, video y streaming

En el contexto del procesamiento de imágenes y visión por computador, es fundamental distinguir entre los conceptos de imagen, video, stream y procesamiento en tiempo real, ya que implican diferencias relevantes a nivel conceptual, temporal y de implementación.

Un video puede entenderse como una secuencia ordenada de imágenes (frames) asociadas a información temporal. Técnicamente, un archivo de video está compuesto por un contenedor que organiza uno o más streams junto con metadatos y marcas temporales que permiten reconstruir el orden y la sincronización de la secuencia. El proceso de decodificación transforma los datos codificados del stream (por ejemplo, H.264) en frames de imagen en memoria, respetando el orden temporal indicado por los timestamps.

Desde el punto de vista del consumo de datos, cabe distinguir entre procesamiento offline y procesamiento en tiempo real. En un escenario offline, el stream está completamente definido desde el inicio: el productor puede entregar frames tan rápido como el sistema sea capaz de leerlos, sin presión temporal. En un escenario en tiempo real, el stream se construye progresivamente a medida que los datos son generados por una fuente activa como una cámara, y el sistema debe procesar cada frame dentro de una ventana temporal acotada para evitar acumulación de latencia o pérdida de datos.

A pesar de esta diferencia en la disponibilidad de los datos, ambos modos comparten una noción clave: la existencia de un contrato de stream que define el formato, la resolución, la cadencia y las políticas de manejo de los datos. En el caso offline, este contrato está explícitamente codificado en el contenedor del archivo. En tiempo real, se establece mediante configuración o negociación antes de recibir el primer frame. Este contrato permite que los frames sean interpretados como parte de una misma secuencia coherente, con independencia de si la fuente es un archivo o una cámara.

2.4.2. GStreamer para la construcción de pipelines

GStreamer es un framework de código abierto para el procesamiento de flujos multimedia, diseñado alrededor del concepto de pipeline: una cadena de elementos procesadores conectados entre sí a través de la cual fluyen los datos desde una fuente hasta un destino [32]. Su arquitectura basada en grafos dirigidos, su soporte para paralelismo interno mediante threads gestionados por el

propio framework, y su extensibilidad mediante plugins lo han convertido en la base sobre la que se construyen numerosas aplicaciones de procesamiento de video en tiempo real, incluyendo el SDK NVIDIA DeepStream.

Elementos, pads y caps

El bloque fundamental de GStreamer es el *elemento*. Cada elemento realiza una operación específica sobre el flujo de datos: leer un archivo, demultiplexar un contenedor, decodificar un codec, transformar un formato, ejecutar un procesamiento arbitrario, o escribir hacia un destino [32]. Los elementos se conectan entre sí a través de *pads*: puntos de conexión tipados que pueden ser de tipo *source* (salida de datos desde el elemento) o *sink* (entrada de datos hacia el elemento). Un elemento fuente tiene únicamente source pads; un elemento destino tiene únicamente sink pads; y los elementos de procesamiento intermedio tienen ambos.

Los pads describen los tipos de datos que pueden manejar mediante *capabilities* o *caps*, que especifican propiedades como el tipo de medio, la resolución, la tasa de frames o el formato de píxel. Cuando dos elementos se conectan, GStreamer verifica la compatibilidad de sus caps mediante un proceso de negociación; si los caps no son compatibles, la conexión falla con un error en tiempo de construcción del pipeline [32]. Este mecanismo garantiza la coherencia del tipo de datos a lo largo del pipeline y constituye la base sobre la que DeepStream controla los formatos de buffer entre sus plugins.

Los elementos pueden agruparse en *bins*, que actúan como contenedores lógicos gestionando el estado y el bus de mensajes de sus elementos internos como si fueran una unidad. Un *pipeline* es un tipo especial de bin que actúa como el contenedor de nivel superior de la aplicación, gestionando el reloj global de sincronización y las transiciones de estado de todos los elementos que contiene [32].

Estados y flujo de ejecución

Un pipeline GStreamer puede encontrarse en cuatro estados: **NULL**, **READY**, **PAUSED** y **PLAYING**. Las transiciones son gestionadas automáticamente por el bin de nivel superior, que las propaga desde los elementos sink hacia los elementos source, garantizando que los elementos downstream estén listos para recibir datos antes de que los elementos upstream comiencen a producirlos. El estado **PLAYING** activa el flujo continuo de buffers de datos a través del pipeline.

Pad probes

GStreamer permite insertar *pad probes*: callbacks invocados cada vez que un buffer cruza un pad específico del pipeline. Estas sondas permiten inspeccionar, modificar o interceptar los datos en cualquier punto del grafo sin interrumpir su flujo normal [32], y constituyen el mecanismo principal para inyectar lógica de medición o depuración desde una aplicación externa al pipeline.

Construcción de pipelines

Los pipelines GStreamer pueden construirse mediante dos enfoques. El primero es la herramienta de línea de comandos `gst-launch-1.0`, que interpreta una descripción textual del pipeline en la que los elementos se enumeran separados por el operador `!` y sus propiedades se especifican en línea, ideal para prototipado rápido. El segundo es la API programática en C o Python, que construye el pipeline elemento por elemento con mayor control sobre el ciclo de vida del sistema. En este trabajo se emplearon ambos enfoques: `gst-launch-1.0` para el desarrollo y verificación inicial del pipeline, y la API Python con pad probes para la evaluación cuantitativa del rendimiento.

2.4.3. NVIDIA DeepStream

NVIDIA DeepStream es un SDK para el desarrollo de aplicaciones de análisis de video asistidas por inteligencia artificial sobre hardware NVIDIA [33]. Construido sobre GStreamer, DeepStream lo extiende con un conjunto de plugins especializados que encapsulan operaciones aceleradas por hardware: decodificación de video mediante NVDEC, preprocesamiento de frames en GPU, inferencia mediante TensorRT, seguimiento de objetos y transmisión de resultados. El objetivo central de esta integración es mantener todos los datos dentro del espacio de memoria GPU a lo largo de todas las etapas del pipeline, eliminando las transferencias CPU-GPU entre etapas que constituyen el principal cuello de botella en implementaciones convencionales.

DeepStream está diseñado para su despliegue tanto en plataformas de servidor con GPU discreta como en plataformas embebidas de la familia Jetson. En estas últimas, la memoria unificada entre CPU y GPU simplifica adicionalmente el flujo de datos, aunque persisten las consideraciones de alineación y layout de buffers descritas en las Secciones 2.2.3 y 2.2.4. La configuración de cada plugin se controla mediante archivos de texto estructurados, lo que permite modificar el comportamiento del pipeline sin recompilar código. El comportamiento personalizado se implementa mediante bibliotecas compartidas compiladas (.so) cargadas dinámicamente por los plugins en tiempo de ejecución.

Flujo de procesamiento

El pipeline de DeepStream organiza el procesamiento en un conjunto de etapas sucesivas que transforman los frames de video en resultados de análisis, identificadas por la documentación oficial del SDK como: captura y decodificación (*capturing and decoding*), preprocesamiento y agrupamiento en batches (*preprocessing and batching*), inferencia con IA (*AI inference*), seguimiento de objetos (*tracking*), generación de metadatos para mensajería y analítica (*IoT message brokers and analytics*), y composición de la salida visual (*composing*) [33]. No todas las etapas son obligatorias: el pipeline puede configurarse para omitir o sustituir cualquiera de ellas según los requisitos de la aplicación. En este trabajo se prescinde de las etapas de tracking, mensajería IoT (*Internet of Things*) y composición estándar, que son sustituidas por un postprocesamiento personalizado que opera directamente sobre los tensores de salida de la inferencia; los criterios de esta decisión se desarrollan en el Capítulo de Metodología.

Plugins principales

Los plugins de DeepStream relevantes para el pipeline de este trabajo son los siguientes [33].

nvv4l2decoder: Plugin de decodificación de video por hardware que utiliza el motor NVDEC disponible en plataformas Jetson. Produce frames decodificados directamente en memoria GPU sin intervención de la CPU, eliminando la transferencia que sería necesaria con decodificadores por software.

nvstreammux: Multiplexor de streams que agrupa frames de una o más fuentes en lotes de tamaño configurable, produciendo el buffer de batch que las etapas downstream esperan. Gestiona además la estructura de metadatos `NvDsBatchMeta` que acompaña al buffer a lo largo del pipeline y sirve como contenedor para los resultados generados por los plugins posteriores. Sus parámetros principales `batch-size`, `width` y `height` se configuran como propiedades del elemento en la descripción del pipeline.

nvdspreprocess: Plugin de preprocesamiento configurable que opera sobre buffers en memoria GPU. Permite definir regiones de interés (ROI, *Region Of Interest*) sobre los frames del batch y aplica transformaciones como reescalado, conversión de formato y normalización antes de la inferencia. Su comportamiento se controla mediante un archivo de configuración .txt que especifica las ROIs, el formato de salida del tensor y los parámetros de normalización. Cuando este plugin está activo en el

pipeline, la etapa de preprocesamiento interna de **nvinfer** se desactiva y el tensor preprocesado se transfiere directamente al motor de inferencia. La lógica de preprocesamiento puede personalizarse implementando la interfaz C++ provista por el SDK y compilándola como biblioteca **.so**, como se hizo en este trabajo para la implementación de normalización y estandarización por canal RGB y otras medidas.

nvinfer: Plugin de inferencia que ejecuta un modelo TensorRT sobre el batch de tensores preprocesados. Se configura mediante un archivo **.txt** que especifica la ruta al plan TensorRT serializado, el tipo de red, las dimensiones de entrada y salida, la precisión numérica y otros parámetros de ejecución. Cuando se habilita **output-tensor-meta=1** en su archivo de configuración, el tensor de salida del modelo se adjunta como metadato al buffer del pipeline en lugar de ser procesado por el postprocesador interno estándar, permitiendo que un plugin downstream acceda directamente al tensor crudo para su postprocesamiento personalizado.

nvdspostprocess: Plugin de postprocesamiento extensible que recibe los tensores de salida de **nvinfer** y ejecuta lógica definida por el usuario mediante una biblioteca compartida dinámica. La interfaz de personalización se implementa en C++ extendiendo la clase base provista por el SDK, que expone un *output thread* donde se procesan los tensores de salida para cada frame del batch. El plugin se configura mediante un archivo **YAML** y la ruta a la biblioteca **.so** compilada se especifica como propiedad del elemento en el pipeline.

fakesink / filesink / udpsink: Elementos sink que determinan el destino final del pipeline. **fakesink** descarta los buffers sin procesamiento adicional, útil para evaluaciones de rendimiento donde el objetivo es medir la velocidad del pipeline sin el costo de escritura a disco o red. **filesink** escribe los buffers a un archivo. **udpsink** transmite los datos mediante UDP, habilitando la recepción en tiempo real por parte de un cliente externo para visualización o almacenamiento remoto.

Archivos de configuración

Una característica central de DeepStream es la separación entre la lógica del pipeline y sus parámetros operacionales mediante archivos de configuración. Cada plugin relevante consume uno o más archivos de texto que controlan su comportamiento en tiempo de ejecución:

El archivo de configuración de **nvdspreprocess** especifica en formato **.txt** los parámetros de las ROIs sobre el frame de entrada (coordenadas, dimensiones), el formato y resolución del tensor de salida, y los coeficientes de normalización aplicados canal a canal. Esto permite cambiar la región de interés o los parámetros de normalización sin recompilar ningún componente del pipeline.

El archivo de configuración de **nvinfer** especifica en formato **.txt** la ruta al plan TensorRT serializado (**model-engine-file**), el tipo de red (**network-type**), las dimensiones del tensor de entrada (**infer-dims**), la precisión numérica, el batch size, y si los tensores de salida deben exponerse como metadatos para el postprocesamiento personalizado (**output-tensor-meta**). También puede especificar una biblioteca de parser personalizada para interpretar la salida del modelo en caso de que el formato no corresponda a los tipos estándar soportados por DeepStream.

El archivo de configuración de **nvdspostprocess** en formato **YAML** especifica parámetros que el output thread personalizado puede leer en tiempo de inicialización, como rutas a recursos auxiliares, parámetros del mapa de color para la visualización de temperatura, o umbrales de postprocesamiento.

Extensión mediante C++ y kernels CUDA

La arquitectura de extensión de DeepStream es especialmente pertinente para aplicaciones científicas donde el postprocesamiento estándar (diseñado para modelos de detección y clasificación de objetos) no es aplicable. En el caso de pirometría de hollín, la salida del modelo es un tensor de regresión densa que debe ser denormalizado, enmascarado sobre la región de la llama y convertido en una

representación visual de temperatura, operaciones que no tienen equivalente en los postprocesadores estándar de DeepStream.

La implementación del output thread de `nvdspostprocess` en C++ recibe los tensores de salida de la inferencia como punteros a memoria GPU, lo que permite operar directamente sobre ellos mediante kernels CUDA personalizados sin transferir los datos al host. En el pipeline de este trabajo, el output thread invoca un flujo de kernels CUDA que realiza la denormalización del tensor de temperatura, el cálculo del umbral de Otsu frame a frame sobre el mapa resultante, la generación de la máscara binaria de la región de hollín activo, la superposición del mapa de temperatura con escala de color sobre el frame original en la ROI correspondiente, y el trazado del perfil centerline de temperatura para visualización. Todas estas operaciones se ejecutan íntegramente en GPU, preservando el principio de residencia en memoria del device que caracteriza al pipeline DeepStream.

Metadatos y pad probes en DeepStream

DeepStream gestiona la información asociada a cada frame a través de la estructura `NvDsBatchMeta`, que acompaña al buffer GStreamer a lo largo del pipeline y puede ser extendida con metadatos personalizados por cualquier plugin. Los pad probes de GStreamer permiten acceder a esta estructura desde código externo al pipeline, adjuntando callbacks a los pads de cualquier elemento sin interrumpir el flujo de datos [32].

METODOLOGÍA

Este capítulo describe el conjunto de decisiones metodológicas, implementaciones y configuraciones que conforman el trabajo desarrollado. La exposición se organiza en cuatro bloques principales. El primero establece el contexto general del sistema, incluyendo el caso de aplicación que motiva la evaluación experimental, los conjuntos de datos empleados, los modelos de red neuronal preentrenados que sirven como base para todos los experimentos, y ciertas decisiones de configuración del pipeline compartidas por ambas implementaciones. El segundo bloque describe la construcción del pipeline de referencia en Python. El tercero presenta la réplica de los experimentos de optimización de inferencia con TensorRT, cuyos resultados de precisión justifican la configuración adoptada en el pipeline optimizado. Y el cuarto bloque describe el diseño e implementación del pipeline DeepStream sobre plataforma Jetson.

3.1. Contexto general del sistema

Esta sección establece el marco común sobre el que se construyen los siguientes bloques metodológicos. Se presenta en primer lugar el caso de aplicación que motiva la evaluación experimental del pipeline, seguido del entorno de hardware y software sobre el que se ejecutan todos los experimentos, los conjuntos de datos empleados, los modelos preentrenados utilizados como punto de partida, y las decisiones de configuración compartidas por ambos pipelines. Estos elementos son transversales al proyecto: el baseline Python, el bloque de selección de precisión TensorRT y el pipeline DeepStream operan todos sobre el mismo caso de aplicación, la misma plataforma, los mismos datos y los mismos modelos de base.

3.1.1. Contexto de aplicación

La estimación de la temperatura del hollín en llamas laminares constituye el caso de aplicación sobre el cual se evalúa experimentalmente el pipeline optimizado descrito en este trabajo. Esta subsección resume los fundamentos físicos que motivan dicha elección y las dificultades que hacen de este problema un escenario de prueba representativo para la reingeniería de pipelines de inferencia en tiempo real.

Fundamentos de la pirometría óptica

La pirometría de hollín es una técnica de diagnóstico óptico no intrusiva utilizada para estimar la temperatura en sistemas de combustión donde la radiación térmica emitida por partículas sólidas incandescentes constituye el mecanismo dominante de emisión. En llamas de combustión incompleta de hidrocarburos, la presencia de hollín permite acceder indirectamente a la información térmica

del sistema a través del análisis de la radiación que estas partículas emiten al encontrarse a altas temperaturas.

La temperatura es una de las variables termodinámicas más relevantes en el estudio de la combustión, ya que gobierna tanto la cinética química como los procesos de formación, crecimiento y oxidación del hollín [34]. Asimismo, el campo de temperatura constituye una referencia fundamental para la validación de modelos numéricos y para la comparación entre resultados experimentales y simulaciones [35]. En este contexto, disponer de técnicas confiables para estimar la temperatura del hollín resulta esencial para el estudio de llamas laminares y para el desarrollo de modelos predictivos de combustión [36].

A diferencia de los métodos de contacto, la pirometría óptica permite inferir campos de temperatura sin perturbar el flujo ni alterar la estructura de la llama. Esta característica resulta particularmente importante en sistemas laminares, donde pequeñas perturbaciones pueden modificar de manera significativa la distribución espacial de las propiedades térmicas y químicas. Por esta razón, la pirometría de hollín se ha consolidado como una herramienta clave en estudios fundamentales de combustión y en configuraciones experimentales controladas.

Este trabajo se centra en llamas de difusión laminares axisimétricas canónicas (CLAD, *Canonical Laminar Axisymmetric Diffusion*), un escenario de referencia ampliamente utilizado en estudios fundamentales de combustión por su estabilidad, simetría y reproducibilidad. El hollín presenta una elevada capacidad de absorción y emisión en el espectro visible [37], lo que lo convierte en un marcador óptico natural para la estimación de temperatura mediante cámaras RGB convencionales, aunque a costa de una notable pérdida de resolución espectral respecto a técnicas más especializadas.

Autoabsorción y proyección óptica

Uno de los principales desafíos en la pirometría de hollín en llamas es la presencia de fenómenos radiativos que alteran la señal emitida antes de su detección. Entre estos, destaca la autoabsorción radiativa: la radiación térmica generada por partículas de hollín en regiones internas de la llama puede ser absorbida parcialmente por otras partículas a lo largo del trayecto óptico, de modo que la intensidad registrada por el sensor no depende únicamente de la temperatura local de emisión, sino también de la distribución espacial del hollín y de sus propiedades ópticas [38]. Este efecto es especialmente relevante en llamas de difusión con concentraciones moderadas a altas de hollín, donde las capas externas pueden atenuar y modificar la radiación proveniente del interior, introduciendo una dependencia no local en la señal detectada.

Adicionalmente, tal como se aprecia en la Figura 3.1, la cámara registra una imagen bidimensional que corresponde a la integración de la radiación emitida a lo largo de la línea de visión. En llamas CLAD, esta proyección puede interpretarse como una integración tangencial del campo radiativo, en la cual cada píxel resulta de la superposición de contribuciones provenientes de distintas posiciones radiales dentro de la llama. En consecuencia, la imagen bidimensional constituye una representación comprimida de la información volumétrica del sistema, donde la estructura tridimensional del campo de temperatura queda implícitamente codificada en la intensidad radiométrica proyectada.

La combinación de autoabsorción radiativa y proyección óptica hace que la relación entre los campos internos de temperatura y las imágenes registradas sea altamente no trivial, estableciendo las bases físicas que explican la dificultad de reconstruir información local a partir de mediciones ópticas integradas.

Limitaciones del enfoque físico-analítico

Los métodos tradicionales de pirometría de hollín se basan en la formulación explícita de modelos físicos que describen la emisión, absorción y propagación de la radiación térmica en el interior de la

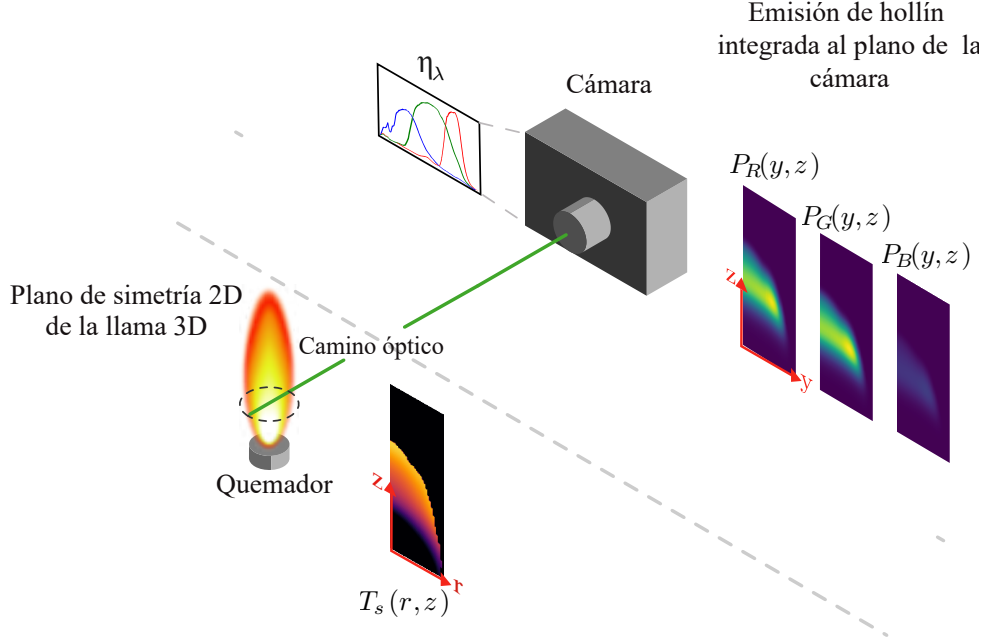


Fig. 3.1: Esquema del arreglo óptico BEMI para capturar la radiación emitida por las partículas de hollín.

llama. En particular, estimaciones precisas de la temperatura requieren mediciones multiespectrales que permitan vincular la radiación registrada con la distribución interna de temperatura mediante la resolución de un problema inverso [35].

Dentro de este marco, técnicas como *Modulated Absorption Emission* (MAE) permiten estimar simultáneamente la temperatura y la fracción volumétrica de hollín mediante la resolución de la ecuación de transferencia radiativa [36], pero requieren configuraciones experimentales complejas, fuentes de iluminación externas, sistemas de sincronización y alineación óptica precisa que limitan su aplicabilidad fuera de entornos de laboratorio.

Como alternativa, los métodos basados únicamente en *Broadband Emission* (BEMI) emplean cámaras digitales para capturar la radiación emitida en múltiples canales espectrales [39, 40], reduciendo costos y complejidad experimental. Sin embargo, al utilizar sensores de banda ancha, la señal corresponde a una integración espectral amplia sobre una proyección de la línea de visión, donde se superponen contribuciones de regiones con diferentes temperaturas. Los efectos de autoabsorción introducen además un acoplamiento no lineal entre capas, que estos métodos tienden a omitir, produciendo subestimaciones de temperatura del orden de decenas de kelvin bajo condiciones de alta carga de hollín [41].

Desde el punto de vista matemático, la estimación de campos locales de temperatura a partir de señales proyectadas constituye un problema inverso inherentemente mal condicionado. Pequeñas perturbaciones en las mediciones (derivadas del ruido del sensor, variaciones experimentales o incertidumbres de calibración) pueden traducirse en desviaciones significativas en la temperatura estimada.

Esta fragilidad de los métodos analíticos, sumada a la complejidad instrumental de los enfoques de mayor precisión, motiva el uso de arquitecturas de deep learning capaces de aproximar la relación

no lineal entre observaciones ópticas y temperatura sin requerir la resolución explícita del problema inverso [1, 5]. Es precisamente esta combinación de características: un modelo de regresión densa no trivial, un problema físico con relevancia científica propia, y un requisito genuino de operación continua sobre flujos de video, la que hace de la pirometría de hollín un caso de aplicación adecuado para evaluar la optimización de pipeline propuesta en este trabajo, más allá de cualquier plataforma de prueba sintética.

3.1.2. Entorno de operación

Todos los experimentos de este trabajo se realizaron sobre la plataforma NVIDIA Jetson AGX Orin 64GB Developer Kit, que constituye tanto el entorno de desarrollo como la plataforma de despliegue objetivo del sistema. Este dispositivo integra una GPU basada en la arquitectura Ampere con 64 GB de memoria unificada LPDDR5 compartida entre CPU y GPU, y un procesador ARM (*Advanced RISC Machine*) de 12 núcleos¹, con soporte nativo para rutas de cómputo en media precisión mediante Tensor Cores y cuantización de 8 bits. La arquitectura de memoria unificada elimina la necesidad de transferencias explícitas entre host y device mediante DMA, aunque preserva las consideraciones de alineación de buffers y coalescing descritas en la Sección 2.2.3.

El entorno de software se configuró sin entornos virtuales Python, dado que la integración con el stack NVIDIA (CUDA Toolkit, TensorRT, DeepStream y OpenCV) requiere acceso directo a las bibliotecas del sistema. Esta decisión garantiza compatibilidad entre todos los componentes del stack y evita conflictos de versiones que pueden surgir al aislar dependencias en entornos virtuales cuando estas dependen de extensiones nativas del sistema operativo. Se verificó que la GPU del dispositivo expusiera soporte operativo para las rutas de precisión reducida utilizadas en la compilación de engines, asegurando que los planes generados por el builder pudieran ejecutarse sin fallos ni inestabilidad numérica en la running phase.

Para minimizar interferencias externas durante las mediciones, se adoptó un esquema de inicialización diferida de los módulos CUDA, activándolos únicamente cuando son invocados por el pipeline, y se evitó ejecutar procesos concurrentes de alto consumo durante las sesiones de medición. El dispositivo se configuró en modo de máximo rendimiento (`MAXN, ID 0`) mediante `sudo nvpmodelΓm 0`, con los relojes de CPU y GPU fijados al máximo mediante `sudo jetson_clocks`, deshabilitando los mecanismos de gestión dinámica de frecuencia que podrían introducir variabilidad en las mediciones [42]. Esta configuración se verificó al inicio de cada sesión de medición mediante `nvpmodelΓq` y `jetson_clocks--show`, reapiando ambos comandos en la configuración inicial del dispositivo o en caso de detectar una desviación respecto al estado esperado.

3.1.3. Conjuntos de datos

El trabajo se apoya en conjuntos de datos simulados y experimentales previamente desarrollados en el contexto de la pirometría de hollín en llamas laminas [1]. Estos conjuntos han sido validados en estudios anteriores tanto para el entrenamiento de redes neuronales como para la evaluación de técnicas de medición óptica de temperatura, y se emplean aquí sin modificaciones como base para el análisis del pipeline de inferencia.

Datos simulados

El conjunto simulado fue generado mediante un marco numérico que combina resultados de simulaciones de llamas laminas con modelos radiativos espectrales, reproduciendo la respuesta de

¹RISC: *Reduced Instruction Set Computing*.

Tabla 3.1: Condiciones de llama simuladas con CoFlame.

ID de llama	C ₂ H ₄ (cm ³ /min)	N ₂ (cm ³ /min)	Air (L/min)	Height (mm)
Yale-32	80.28	170.59	98.36	25.8
Yale-40	100.35	150.52	98.36	32.0
Yale-60	150.52	100.35	98.36	51.0

un sistema de medición óptica real [1]. Las condiciones de llama fueron modeladas con el software CoFlame [43], que permite simular llamas CLAD producidas con un quemador tipo Yale [44].

Se consideraron tres configuraciones Yale-32, Yale-40 y Yale-60, definidas por distintos caudales de combustible C₂H₄ y gas cofluyente N₂ según se indica en la Tabla 3.1. Estas configuraciones, seleccionadas debido a su estabilidad, reproducibilidad y pertinencia para el estudio sistemático de la formación y radiación de hollín, inducen cambios significativos en los gradientes térmicos y en la fracción volumétrica de hollín, produciendo una distribución espacialmente diversa de los campos de temperatura.

La simulación de señales RGB incorpora los parámetros espectrales de un montaje experimental tipo BEMI compuesto por una cámara Basler acA2040-55uc y un lente AF-S Nikkor de 50 mm, generando proyecciones RGB consistentes con la sensibilidad espectral del sensor. Para cada condición de llama se generaron aproximadamente 8 000 ejemplos, alcanzando un total cercano a 24 000 muestras. Cada muestra está compuesta por un par entrada-salida: una imagen RGB simulada $P_{\{R,G,B\}}$ utilizada como input del sistema, y el correspondiente campo 2D de temperatura del hollín T_s , empleado como ground truth de supervisión. El conjunto fue distribuido en particiones de entrenamiento, validación y prueba siguiendo proporciones estándar en regresión densa con aprendizaje supervisado.

Datos experimentales

El conjunto experimental fue obtenido a partir de mediciones realizadas sobre llamas laminares generadas con un quemador Yale bajo condiciones similares a las consideradas en las simulaciones indicadas en la Tabla 3.1. Las mediciones emplean diversas técnicas ópticas de pirometría: MAE de banda angosta, *Emission* (EMI) de dos colores y BEMI, utilizando el mismo sistema óptico considerado en la generación de datos simulados, lo que permite una comparación más directa entre ambos conjuntos.

Para técnicas MAE de un color, se dispone de 100 imágenes con iluminación LED activa y 100 imágenes con LED desactivado por cada condición de llama[45]. Para técnicas EMI de dos colores, se registraron 100 imágenes por longitud de onda y por condición de llama[35]. Para técnicas BEMI, se recolectaron 300 imágenes RGB por cada condición de llama. En todos los casos, las imágenes de una misma condición experimental fueron promediadas para mejorar la relación señal-ruido.

Para los fines de este trabajo, entonces, el conjunto experimental se divide en dos subconjuntos con usos diferenciados. El primero, compuesto por una imagen promediada por condición con su correspondiente campo de temperatura de referencia, se emplea para evaluar la precisión de regresión de los modelos en la Sección 3.3.1 para guiar la selección de precisión de TensorRT. El segundo está compuesto por las 300 imágenes individuales sin ground truth de cada condición de llama, para un total de 900 imágenes, y tiene uso como conjunto de calibración para la cuantización INT8 de los engines TensorRT y como material base para la construcción de los flujos de video descritos en la subsubsección siguiente.

Construcción de los flujos de video

Los pipelines evaluados en este trabajo operan sobre flujos de video en lugar de imágenes individuales, lo que requirió construir secuencias de video a partir de las imágenes experimentales disponibles. Para cada condición de llama, las imágenes BEMI capturadas en laboratorio fueron concatenadas en orden y codificadas como un archivo de video MP4 mediante FFmpeg [46], utilizando el codec H.264 (`libx264`), formato de píxel `yuv420p` compatible con decodificadores de hardware, una tasa de 30 FPS y un factor de calidad `crf 18` que preserva la fidelidad visual de las imágenes originales. La resolución del video resultante corresponde a la resolución nativa de la cámara Basler: 2048×1536 px.

Esta construcción no introduce información temporal real entre frames, pero establece un contrato de stream válido para ambos pipelines: define el formato, la resolución, la cadencia de entrega y el codec de decodificación que tanto el baseline como el pipeline DeepStream esperan recibir en su entrada.

3.1.4. Modelos preentrenados

Los modelos empleados en este trabajo son los desarrollados y validados en investigaciones previas del grupo [5, 1]: una arquitectura U-Net base y una variante Attention U-Net, cuyas propiedades estructurales fueron descritas en el Capítulo 2. Ambos modelos fueron entrenados bajo aprendizaje supervisado sobre el conjunto de datos simulado y validados con datos experimentales, y se utilizan aquí en forma de archivos de pesos `.pth` sin modificar ni reentrenar.

Ambos modelos reciben como entrada tensores BGR en formato NCHW con dimensiones $[1, 3, 128, 32]$ y producen un tensor de salida de dimensiones $[1, 1, 128, 32]$ que representa el mapa denso de temperatura estimada T_s , con activación lineal en la capa final. La U-Net base cuenta con 1 884 631 parámetros entrenables; la Attention U-Net con 8 828 070. Estos archivos de pesos constituyen el punto de partida común para los tres bloques metodológicos siguientes: el pipeline baseline en Python los carga directamente para inferencia mediante PyTorch, mientras que el bloque de construcción de engines los exporta a ONNX como paso previo a la optimización con TensorRT para su uso en el pipeline DeepStream.

3.1.5. Consideraciones transversales al pipeline

Además de los datos y los modelos, ambos pipelines comparten un conjunto de decisiones de configuración y de componentes algorítmicos que condicionan su comportamiento en tiempo de ejecución, y cuya descripción conjunta evita reiterarla por separado en las secciones dedicadas a cada implementación.

Batch size

En todos los experimentos de este trabajo se adopta un batch size de 1 para la evaluación del pipeline en condiciones de streaming, el mismo valor que aparece como primera dimensión de los tensores de entrada y salida de los modelos descritos en la subsección anterior. Esta decisión refleja el escenario de despliegue objetivo: un único flujo de video proveniente de una fuente continua (cámara o archivo procesado como stream) donde los frames deben procesarse individualmente a medida que llegan, sin posibilidad de acumular un lote antes de proceder. Bajo esta configuración, la latencia end-to-end es la métrica relevante, ya que cada frame debe ser procesado y su resultado entregado antes de que el siguiente requiera atención.

El batch size podría incrementarse en dos escenarios distintos. En un sistema de múltiples fuentes simultáneas, el multiplexor del pipeline podría agregar frames de distintas cámaras en un mismo

batch, mejorando la utilización de la GPU. Alternativamente, en un modo de procesamiento offline donde no existe presión temporal de entrega, batches mayores permiten maximizar el throughput a costa de mayor latencia por muestra. Ambos escenarios quedan fuera del alcance de este trabajo pero constituyen extensiones naturales del sistema desarrollado.

Segmentación por umbralización de Otsu

Ambos pipelines emplean el método de Otsu [47] para segmentar la ROI (la llama) del fondo, previo al cálculo de los paneles de visualización. Dado que el algoritmo es idéntico en ambas implementaciones y difiere únicamente en el dispositivo de ejecución (CPU en el pipeline de referencia, GPU en el pipeline optimizado), se describe aquí una única vez su formulación, remitiendo a las secciones correspondientes para el detalle de cada implementación.

El método de Otsu determina, a partir del histograma de intensidades de una imagen, el umbral t^* que maximiza la varianza inter-clase entre dos poblaciones de píxeles: fondo y primer plano, separadas por dicho umbral. Formalmente, sea h_i el número de píxeles en el nivel de intensidad $i \in [0, L - 1]$, con $L = 256$ niveles. Para un umbral candidato t , se definen los pesos de clase

$$w_0(t) = \sum_{i=0}^t p_i, \quad w_1(t) = \sum_{i=t+1}^{L-1} p_i, \quad (3.1.1)$$

donde $p_i = h_i/N$ es la probabilidad del nivel i y N el número total de píxeles, y las medias de clase

$$\mu_0(t) = \frac{1}{w_0(t)} \sum_{i=0}^t i p_i, \quad \mu_1(t) = \frac{1}{w_1(t)} \sum_{i=t+1}^{L-1} i p_i. \quad (3.1.2)$$

La varianza inter-clase para el umbral t se define como

$$\sigma_B^2(t) = w_0(t) w_1(t) (\mu_0(t) - \mu_1(t))^2, \quad (3.1.3)$$

y el umbral óptimo es aquel que maximiza esta cantidad sobre el espacio discreto de niveles posibles:

$$t^* = \arg \max_{t \in [0, L-1]} \sigma_B^2(t). \quad (3.1.4)$$

En este trabajo, el canal de entrada sobre el cual se aplica el método corresponde al canal G del tensor de entrada normalizado al rango $[0, 1]$ y remapeado a $L = 256$ niveles discretos; el umbral resultante se remapea de vuelta al rango normalizado para generar la máscara binaria correspondiente. Las particularidades de cómputo de cada implementación (construcción del histograma, cálculo de t^* y aplicación de la máscara) se describen en las Secciones 3.2.5 y 3.4.5, respectivamente.

3.2. Pipeline de referencia en Python

El pipeline de referencia, denominado en adelante *baseline*, constituye la implementación Python contra la cual se compara el pipeline DeepStream optimizado. Su punto de partida fue el pipeline desarrollado por Aguilera [8] para la evaluación de engines TensorRT, el cual fue revisado, extendido y estabilizado con el objetivo de producir un sistema compartimentado, reproducible y funcionalmente correcto sobre flujos de video.

La Figura 3.2 presenta una visión general del baseline implementado y sirve como referencia para la descripción detallada de sus distintas etapas presentada a continuación.

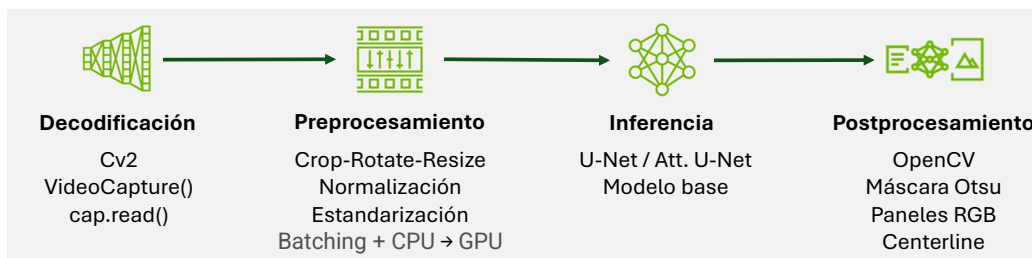


Fig. 3.2: Bslene Python implementado para la inferencia de temperatura de hollín.

3.2.1. Decisiones de diseño

El baseline se caracteriza por una ejecución predominantemente en CPU: la decodificación, el preprocesamiento y el postprocesamiento se realizan íntegramente en el host mediante NumPy [48] y OpenCV [49], siguiendo un modelo de ejecución secuencial donde cada etapa debe completarse antes de que comience la siguiente. La única excepción es la etapa de inferencia, que se ejecuta en GPU mediante PyTorch [22]. Esta decisión se hereda del trabajo de Aguilera y se mantiene por una razón práctica: el mismo baseline es reutilizado en el bloque metodológico siguiente, donde se evalúan engines TensorRT (que requieren GPU) frente a los modelos PyTorch originales. Mantener una infraestructura de inferencia común en ambos casos simplifica la implementación y asegura que las diferencias observadas entre modelos PyTorch y engines TensorRT sean atribuibles exclusivamente a la optimización del engine, y no a diferencias en el entorno de ejecución. Como consecuencia directa de ejecutar la inferencia en GPU, el baseline incorpora dos pasos adicionales ausentes en un diseño puramente secuencial sobre CPU: la preparación del batch (expansión de dimensiones y conversión a tensor mediante `from_numpy`), y la transferencia explícita CPU → GPU del tensor de entrada, ambas agrupadas junto al preprocesamiento.

El baseline está estructurado en módulos con responsabilidades claramente separadas: gestión de argumentos y configuración, carga del modelo, decodificación de video, preprocesamiento, inferencia, postprocesamiento y visualización. Esta compartimentalización facilita el perfilado por etapas y asegura que las optimizaciones aplicadas a una etapa no interfieran con la medición de las demás.

El pipeline itera sobre un directorio de videos MP4, procesando cada archivo de forma secuencial. En la configuración de evaluación adoptada en este trabajo, cada directorio contiene un video por condición de llama Yale, de modo que el sistema procesa tres videos en total. El batch size se fija en 1 en todos los modos de operación orientados a streaming, por las razones expuestas en la Sección 3.1.5. El uso de un batch size mayor conserva sentido únicamente en el modo de evaluación de throughput offline, donde se busca caracterizar la capacidad máxima de procesamiento bajo carga continua sin presión temporal de entrega.

El baseline expone tres modos de operación diferenciados. El modo `simulate` constituye la operación principal: procesa el flujo de video frame a frame, ejecuta el pipeline completo y transmite los resultados visualizados como un stream HTTP mediante un servidor Flask, emulando el rol del `udpsink` en el pipeline DeepStream. Los modos `latency` y `throughput` encapsulan la misma lógica de procesamiento pero sin transmisión externa, equivalentes funcionalmente al uso de `fakesink` en DeepStream, y están destinados a la captura de métricas de rendimiento. Existe además un modo `visualize`, orientado a la generación de figuras de documentación mediante Matplotlib en lugar

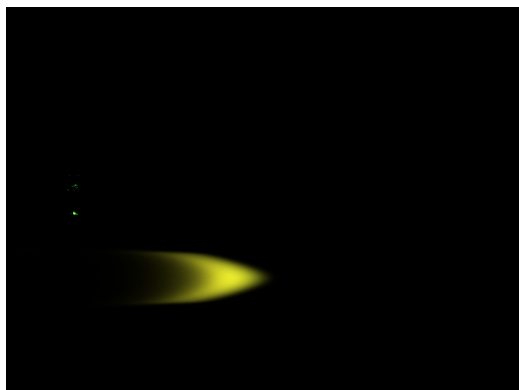


Fig. 3.3: Frame original decodificado desde el video de entrada en el baseline Python, previo a cualquier procesamiento (2048×1536 px).

de OpenCV; al no estar diseñado para evaluación de rendimiento, no se emplea como referencia comparativa en el capítulo de Resultados.

3.2.2. Decodificación

La decodificación de los frames de video se realiza mediante `cv2.VideoCapture` [49], implementada como un generador Python que emite un frame por iteración. Esta estructura permite que el bucle principal consuma frames de forma perezosa (*lazy evaluation*), sin cargar el video completo en memoria. La Figura 3.3 muestra un frame representativo tal como es entregado por el generador, antes de aplicarse cualquiera de las transformaciones descritas en la Sección siguiente.

Durante el desarrollo se evaluó una variante con decodificación en hilo separado y cola FIFO, con el objetivo de desacoplar la decodificación del preprocesamiento y solapar ambas etapas. Esta variante incrementó el throughput al eliminar la serialización entre ambas etapas, aprovechando el paralelismo entre el código nativo de OpenCV (que libera el GIL (*Global Interpreter Lock*) de Python) y el hilo principal. Sin embargo, introduce complejidad de sincronización que dificulta la reproducibilidad de las mediciones y anticipa innecesariamente optimizaciones que son absorbidas de forma más robusta por el pipeline DeepStream, donde la decodificación es trasladada íntegramente a GPU mediante NVDEC. Por coherencia metodológica con el objetivo de establecer una línea base clara, se mantiene la decodificación secuencial simple en el baseline final.

3.2.3. Preprocesamiento

El preprocesamiento transforma cada frame decodificado en el tensor de entrada que el modelo espera: dimensiones $[1, 3, 128, 32]$ en formato NCHW, con valores normalizados y estandarizados. La secuencia de operaciones es la siguiente:

Recorte

Se extrae la ROI de la llama mediante slicing con índices fijos `CENTER_X:BORDER_X` sobre el eje radial. Estos índices son constantes para una geometría de quemador y configuración experimental fijas, y se calculan una única vez fuera del bucle principal.

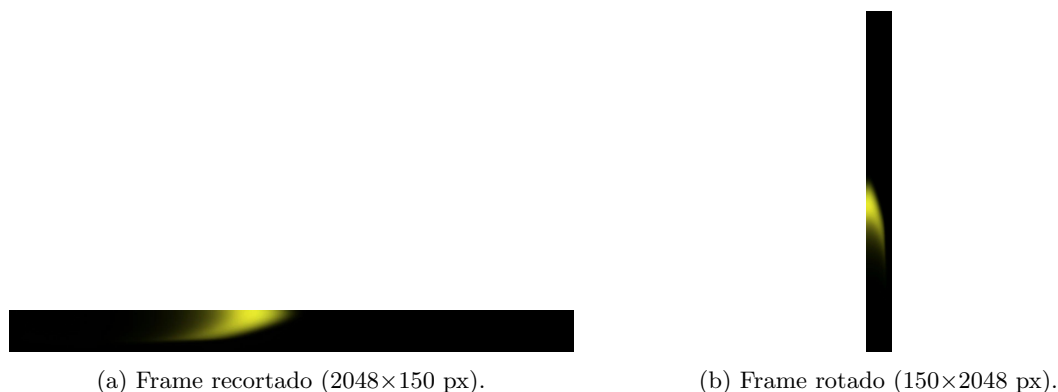


Fig. 3.4: Recorte de la ROI y rotación 90° antihoraria en el baseline Python, aplicados mediante slicing con índices fijos, transposición de ejes y rebanado inverso.

Rotación

La cámara Basler fue dispuesta en orientación horizontal en el montaje experimental, por lo que las imágenes capturadas y, por extensión, los frames del video construido a partir de ellas, presentan la llama en disposición horizontal. Dado que el modelo fue entrenado con imágenes en orientación vertical y no incorpora transformaciones geométricas internas, la rotación debe aplicarse como parte del preprocesamiento. Esta decisión es además pertinente en el contexto de un pipeline en tiempo real: en un sistema de captura continua, cada frame llega con la orientación nativa del sensor, por lo que no es posible aplicar correcciones geométricas de forma offline antes del procesamiento.

Se aplica por tanto una transposición de ejes seguida de un rebanado inverso en el primer eje (`transpose(1,0,2)[:,-1]`), equivalente a una rotación de 90° en sentido antihorario. A diferencia de `np.rot90`, esta operación reordena únicamente los strides del array sin crear una copia en memoria, reduciendo la presión sobre el recolector de basura. La Figura 3.4 ilustra ambas transformaciones sobre un frame representativo: el recorte de la ROI radial y la reorientación vertical resultante.

Reescalado por interpolación bilineal

Los frames recortados deben ser reescalados a la resolución fija $[128 \times 32]$ que el modelo espera. Esta operación fue el principal cuello de botella del pipeline en su versión inicial y concentró el mayor esfuerzo de optimización.

La implementación original empleaba `scipy.interpolate.interp2d` [50], una función de propósito general que reconstruye internamente estructuras de interpolación en cada invocación. Ejecutada tres veces por frame (una por canal de color), consumía aproximadamente 54 ms por frame, representando más del 99 % del tiempo total de preprocesamiento y limitando el throughput del sistema completo. Cabe señalar además que `interp2d` fue declarada obsoleta en SciPy 1.10 y eliminada en la versión 1.14 [51], lo que constituía adicionalmente un riesgo de compatibilidad.

Como primer paso de optimización se sustituyó `interp2d` por `RegularGridInterpolator` [50], también de SciPy, que está diseñada específicamente para grillas regulares y permite separar la construcción del interpolador (realizada una sola vez) de su evaluación por frame. Este cambio redujo el tiempo de interpolación a ≈ 3.5 ms por frame.

El análisis posterior reveló que incluso este costo era evitable. Dado que la grilla de origen (coordenadas del dominio físico) y la grilla de destino (128×32 puntos fijos) permanecen constantes a lo largo de toda la ejecución, los índices base y los pesos de la interpolación bilineal también son



(a) Frame reescalado (128×32 px).

(b) Normalización y estandarización.

Fig. 3.5: Tensor de entrada del baseline Python tras reescalado mediante interpolación bilineal para el proceso posterior de normalización y estandarización por canal.

constantes. Se implementó entonces una versión que precalcula al inicio del programa, mediante `numpy.searchsorted`, los índices de los cuatro vecinos y los pesos asociados para cada punto de la grilla de destino (arrays `IZ`, `IR`, `WZ`, `WR`). Durante la ejecución, la interpolación se reduce a la evaluación directa de la fórmula bilineal vectorizada sobre arrays NumPy:

$$f(z, r) \approx (1 - w_z)(1 - w_r) v_{00} + (1 - w_z) w_r v_{01} + w_z(1 - w_r) v_{10} + w_z w_r v_{11} \quad (3.2.1)$$

donde v_{ij} son los valores en los cuatro vecinos y w_z , w_r los pesos lineales en cada dirección. La Figura 3.5a muestra el tensor resultante de esta etapa para un frame representativo. Esta implementación es matemáticamente equivalente a las dos anteriores, pero elimina completamente el overhead de construcción de objetos interpoladores, reduciendo el tiempo de reescalado a ≈ 0.35 ms por frame. El throughput del sistema mejoró considerablemente tras este cambio, y los picos de latencia anómalos observados previamente en la etapa de transferencia CPU-GPU desaparecieron, confirmando que su causa era la acumulación de latencia en el preprocesamiento que se manifestaba en puntos posteriores del pipeline por efecto de sincronizaciones implícitas de CUDA.

Durante este proceso de optimización se identificó además una inconsistencia entre el dominio de coordenadas de las imágenes experimentales y el dominio de interpolación heredado del pipeline original. El pipeline previo definía el dominio físico de la grilla en unidades métricas mediante un factor de escala y un offset vertical, parámetros calibrados para el montaje experimental original. Al trabajar con imágenes experimentales de geometría y escala conocidas y fijas, estos parámetros resultaban innecesarios y, en la práctica, producían una desalineación entre el dominio físico y el dominio de la grilla de interpolación que situaba los puntos de destino fuera del soporte de los datos. Esta desalineación permanecía oculta en la implementación con `interp2d` gracias a la extrapolación implícita habilitada por `fill_value=None`, pero se manifestaba como resultados nulos o constantes al implementar la interpolación de forma explícita. La solución adoptada fue redefinir ambos dominios en coordenadas de píxel, eliminando los parámetros de escala y asegurando coherencia entre la grilla de origen y la grilla de destino. Las implicancias de esta simplificación y la necesidad de reintroducir la calibración métrica en trabajos futuros orientados a la generalización del sistema se discuten en el Capítulo 5.

Normalización y estandarización

Tras el reescalado, los valores de cada canal se normalizan al rango $[0, 1]$ dividiéndolos por el valor máximo global de la escala de cuantización del sensor ($X_{\max}$), y a continuación se estandarizan restando la media y dividiendo por la desviación estándar de cada canal:

$$x'_c = \frac{x_c / X_{\max} - \mu_c}{\sigma_c}, \quad c \in \{B, G, R\} \quad (3.2.2)$$

Las constantes $X_{\max}$, $\{\mu_c\}$ y $\{\sigma_c\}$ fueron calculadas sobre los tres conjuntos de datos simulados (condiciones Yale-32, Yale-40 y Yale-60), aplicando volteo vertical para reproducir la orientación de entrenamiento. El valor $X_{\max}$ corresponde al máximo global de intensidad observado en el conjunto simulado, y las medias y desviaciones estándar por canal reflejan la distribución radiométrica del hollín en el espectro visible. Estas constantes se usan como valores escalares en el módulo de configuración para evitar recalcularlas en cada ejecución, y se almacenan como arrays NumPy de forma $[3, 1, 1]$ para compatibilidad con broadcasting directo sobre el tensor de entrada. La Figura 3.5b presenta el mismo tensor tras aplicar esta normalización, listo para ser consumido por el modelo.

3.2.4. Inferencia

La inferencia se ejecuta en GPU mediante PyTorch [22] para los modelos `.pth`, y mediante los bindings de TensorRT para los engines optimizados. Ambas rutas comparten la misma infraestructura de transferencia: el tensor de entrada se construye desde el array NumPy preprocesado mediante `torch.from_numpy`, se transfiere con `pin_memory()` y `.to(device, non_blocking=True)`, y la inferencia se ejecuta dentro de un bloque `torch.no_grad()` para deshabilitar el cálculo del grafo de gradientes. Esta arquitectura compartida es la que permite que el baseline sirva indistintamente para los modelos PyTorch del bloque de simulación y para los engines TensorRT del bloque de réplica de Aguilera, sin cambios en la infraestructura de transferencia ni en el postprocesamiento.

Durante el análisis de rendimiento se observó que la etapa de transferencia CPU-GPU presentaba ocasionalmente tiempos anómalos del orden de 110–120 ms intercalados con tiempos normales de ≈ 0.4 ms. El análisis posterior mostró que estos picos no correspondían a la transferencia en sí, sino que reflejaban latencia acumulada en el preprocesamiento que se manifestaba en el momento de la sincronización CUDA. Una vez optimizada la interpolación, los picos desaparecieron completamente.

3.2.5. Postprocesamiento y visualización

El tensor de salida del modelo, de dimensiones $[1, 1, 128, 32]$ y mostrado en la Figura 3.6a, se transfiere al host, se desnormaliza aplicando la estandarización inversa con las constantes μ_{T_s} y σ_{T_s} del conjunto de entrenamiento, y se convierte a Kelvin. Sobre el mapa de temperatura resultante se calcula la máscara de Otsu frame a frame siguiendo la formulación descrita en la Sección 3.1.5, aplicando el umbral resultante al canal G del tensor de entrada para delimitar la región de hollín activo, tal como se observa en la Figura 3.6b. A diferencia de la implementación GPU descrita en la Sección 3.4.5, el baseline recurre directamente a la implementación de Otsu incluida en OpenCV (`cv2.threshold` con el indicador `THRESH_OTSU`), dado que en CPU no existe la restricción de ausencia de primitivas que sí motiva la implementación de kernels propios en el pipeline DeepStream.

La visualización se implementa íntegramente con OpenCV [49] en lugar de Matplotlib. Esta decisión responde a consideraciones de rendimiento en tiempo real: Matplotlib opera sobre un backend de renderización que introduce overhead significativo para operación frame a frame, mientras que OpenCV opera directamente sobre arrays NumPy en memoria con latencia mínima y sin gestión de estado de figura. Para cada frame se generan dos paneles: un mapa de temperatura T_s con escala de

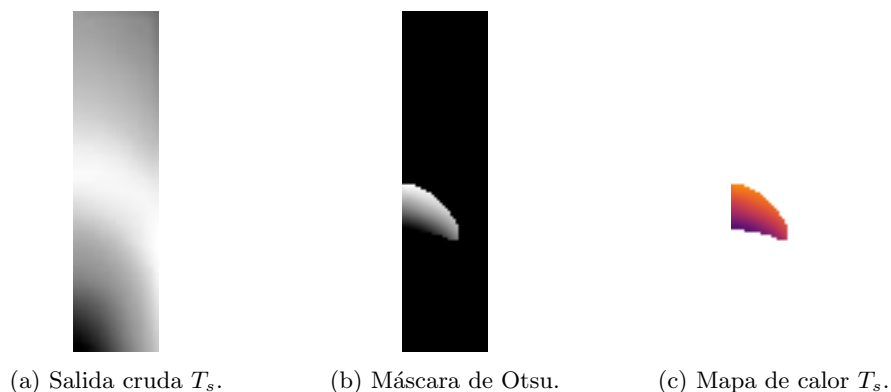


Fig. 3.6: Salida cruda del modelo, máscara de Otsu calculada sobre el canal G del tensor de entrada mediante `cv2.threshold` con el indicador `THRESH_OTSU`, y mapa de temperatura resultante tras desnormalizar y enmascarar la temperatura T_s predicha, en el baseline Python.

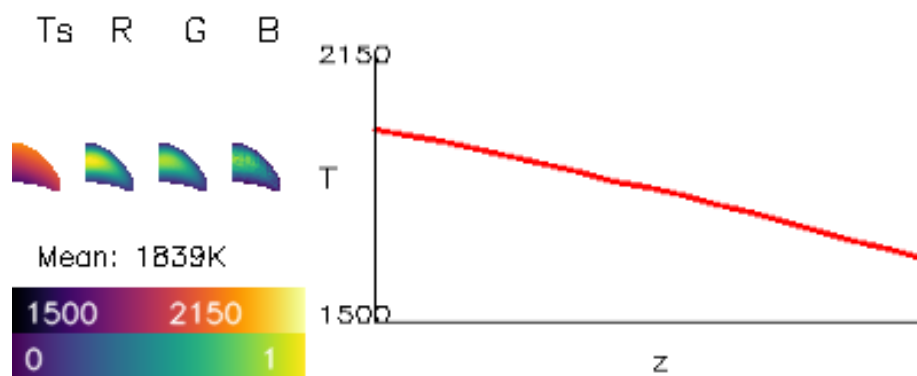


Fig. 3.7: Visualización final del baseline Python: panel T_s -RGB con colorbars y valor medio de temperatura (izquierda), y perfil centerline $T(z)$ (derecha).

color INFERNO superpuesto sobre la región enmascarada, como el que se muestra en la Figura 3.6c, acompañado de los tres canales BGR del tensor de entrada visualizados con escala VIRIDIS para referencia; y un gráfico de perfil centerline que muestra la distribución de temperatura sobre el eje radial central de la llama, trazado directamente sobre un canvas NumPy, tal como se reproduce en la Figura 3.7. Ambos paneles se concatenan horizontalmente y se emiten como frame JPEG al servidor Flask en el modo `simulate`, o se descartan en los modos de evaluación de métricas.

3.3. Configuración de TensorRT

Este bloque metodológico tiene un objetivo acotado y preciso: determinar el nivel de cuantización TensorRT más adecuado para su uso posterior en el pipeline DeepStream. Para ello se replica el flujo experimental desarrollado por Aguilera [8] sobre el caso de pirometría de hollín, evaluando engines compilados en FP32, FP16 e INT8 para ambas arquitecturas de red, con el fin de seleccionar la configuración que mejor equilibre fidelidad numérica y eficiencia computacional bajo las restricciones del sistema objetivo. Dado que el batch size en el pipeline DeepStream es fijo en 1 por las razones

expuestas en la Sección 3.1.5, la evaluación de rendimiento en este bloque se circunscribe también a batch unitario, que es la condición operativa relevante.

3.3.1. Construcción de engines TensorRT

La construcción de los engines comprende dos etapas secuenciales: la exportación del modelo desde PyTorch al formato intermedio ONNX, y la compilación del grafo ONNX hacia un plan TensorRT serializado mediante el builder. Ambas etapas se ejecutan de forma offline, una única vez por modelo y nivel de precisión, produciendo los archivos `.engine` que serán cargados en la running phase tanto en este bloque de evaluación como en el pipeline DeepStream. Los engines se compilaban directamente sobre la plataforma Jetson AGX Orin, garantizando que las tácticas seleccionadas por el builder estén especializadas para la microarquitectura Ampere del dispositivo objetivo y no requieran recompilación en el momento del despliegue.

Exportación a ONNX

Los modelos U-Net y Attention U-Net, almacenados como pesos serializados `.pth`, se exportaron al formato intermedio ONNX siguiendo los principios descritos en la Sección 2.3.3. Para cada modelo se reconstruyó la arquitectura sobre el dispositivo CUDA, se definió un tensor dummy de entrada en formato NCHW con forma (1, 3, 128, 32) y tipo `float32`, y se ejecutó la exportación con batch explícito estático, deshabilitando dimensiones dinámicas para simplificar la compilación y asegurar reproducibilidad. El grafo ONNX resultante fue validado verificando integridad de nodos, tensores y compatibilidad operacional con TensorRT antes de proceder a la compilación.

Configuración del builder

La construcción de los engines se realizó mediante un objeto `EngineBuilder` inicializado a partir del grafo ONNX de cada modelo, con perfil de entrada estático de forma (1, 3, 128, 32). El espacio de trabajo del builder se configuró asignando como límite el total de memoria disponible en la GPU, permitiendo que TensorRT disponga del margen máximo para aplicar optimizaciones de alto nivel. Se estableció nivel de optimización 3, considerado estándar para habilitar la mayor parte de las estrategias de fusión y reescritura del grafo, y una iteración de promediado para el perfilado temporal de capas. Cada engine fue compilado de forma individual por modelo y precisión (FP32, FP16, INT8), almacenándose con nombres específicos para facilitar la trazabilidad.

Calibración INT8

La optimización en precisión INT8 requiere una etapa de calibración estadística que caracterice el comportamiento de las activaciones intermedias del modelo bajo condiciones representativas del dominio de inferencia [52]. Dado que la cuantización de 8 bits restringe la representación numérica a 256 niveles discretos, el proceso debe estimar rangos dinámicos robustos que eviten saturación numérica y minimicen la pérdida de información semántica relevante para la tarea de regresión térmica.

Para este propósito se empleó el segundo subconjunto experimental descrito en la Sección 3.1.3: las 300 imágenes individuales sin ground truth por condición de llama, para el total de 900 imágenes. A diferencia de los conjuntos empleados para entrenamiento o evaluación de precisión, estas imágenes no requieren etiquetas asociadas, ya que su función es describir estadísticamente la distribución de activaciones internas del modelo y no medir su calidad predictiva. La composición balanceada entre las tres condiciones de llama asegura que los tensores intermedios observados durante la calibración reflejen rangos de activación representativos de la variabilidad espectral y espacial del dominio.

Las imágenes fueron preprocesadas de forma homogénea al formato tensorial NCHW esperado por la red, agrupadas en lotes estáticos y almacenadas en un buffer en memoria CPU desde donde son transferidas secuencialmente a GPU conforme el calibrador es consultado por el builder. La caché del builder fue limpiada antes de cada compilación INT8 para evitar la reutilización de perfiles desactualizados, garantizando que la cuantización se ejecute desde un estado limpio y determinista. TensorRT estima las escalas de cuantización por tensor minimizando la divergencia de Kullback-Leibler entre la distribución original de activaciones y su aproximación discreta en 8 bits [52], integrando los factores de escala resultantes directamente en el engine compilado.

3.3.2. Protocolo de evaluación de los engines TensorRT

Esta subsección describe el procedimiento empleado para evaluar los engines construidos según el proceso descrito en la Sección 3.3.1, orientado a contrastar cada nivel de cuantización (FP32, FP16, INT8) contra el modelo base PyTorch en dos dimensiones complementarias: fidelidad numérica de las predicciones respecto del modelo original, y eficiencia computacional bajo las condiciones de despliegue esperadas. Ambas dimensiones son necesarias para una decisión de cuantización informada: una reducción de latencia que sacrifique fidelidad física más allá de un umbral aceptable no constituye una optimización válida para una tarea de regresión térmica donde el error tiene interpretación directa en Kelvin.

Herramientas de instrumentación

La evaluación combina herramientas de inspección estática de los modelos con un protocolo de medición temporal basado en eventos CUDA. Las propiedades estructurales se extraen directamente de los artefactos serializados (grafos ONNX y engines TensorRT) mediante herramientas de inspección específicas, detalladas en la subsubsección siguiente. Las medidas de tiempo, en cambio, se obtienen delimitando la ejecución mediante eventos CUDA insertados antes del lanzamiento del engine al stream y después de su sincronización completa [53], lo que permite aislar el costo de cómputo en GPU del overhead del lenguaje interpretado en el que se orquesta la medición. Para asegurar un régimen estable de ejecución bajo este protocolo, se ejecuta previamente un warm-up equivalente al 10 % del conjunto de evaluación [27], permitiendo que el scheduler de CUDA estabilice sus colas de kernel y que las compilaciones JIT (*Just-In-Time*) diferidas de cuDNN y cuBLAS se completen antes de iniciar las mediciones. Todas las métricas temporales se reportan mediante mediana, percentil 95 (p95) y valor máximo, consistente con el criterio de agregación adoptado en el resto de este trabajo.

El monitoreo de uso de memoria del sistema, empleado en la evaluación comparativa de pipeline completo (Sección 3.5), se realiza mediante `tegrastats` con un intervalo de muestreo de 1000 ms. Esta resolución temporal es adecuada para capturar el consumo sostenido de un pipeline en ejecución continua, pero resulta insuficiente frente a los tiempos de inferencia aislados evaluados en este bloque, del orden de fracciones de milisegundo: cualquier medición de memoria en esta escala no lograría atribuirse específicamente a la inferencia de un modelo o precisión particular, sino al proceso en su conjunto. Por este motivo, el uso de memoria no se reporta como métrica en la evaluación de engines individuales.

Propiedades del modelo

Se caracterizan el número de capas, el número de parámetros y el tamaño en disco de cada modelo y engine. El número de capas se obtiene mediante `Polygraphy` [54], que permite inspeccionar grafos ONNX y engines TensorRT. El número de parámetros de los engines se obtiene mediante `EngineInspector` [55]; para los modelos base se utiliza `ONNX-opcounter` [56]. El tamaño en disco se mide directamente sobre el archivo serializado mediante `os.path.getsize`, excluyendo buffers

temporales de activaciones y workspace que se asignan dinámicamente en runtime.

Métricas de precisión

La precisión de las predicciones se cuantifica mediante el error cuadrático medio (RMSE, *Root Mean Square Error*) sobre el campo bidimensional de temperatura estimada, calculado píxel a píxel respecto del campo de referencia experimental [5]:

$$\text{RMSE} = \sqrt{\frac{1}{m \cdot p} \sum_{i=1}^m \sum_{j=1}^p (Y_{i,j} - \hat{Y}_{i,j})^2} \quad (3.3.1)$$

donde $Y_{i,j}$ es la temperatura de referencia y $\hat{Y}_{i,j}$ la estimada en el píxel (i, j) . El dominio de evaluación se restringe a píxeles con fracción volumétrica de hollín $f_s > 0,1$ ppm y se excluyen diferencias absolutas superiores a ± 100 K, que se concentran en los bordes de la llama donde el ground truth es poco confiable. Este criterio se denomina *Regression Accuracy*.

Como criterio complementario se emplea la *Regression Closeness*: la fracción de píxeles cuya desviación entre la salida del modelo base y la del engine optimizado se mantiene dentro de una tolerancia predefinida, siguiendo la metodología de [57] validada en [8]. El criterio elemento a elemento, implementado mediante `torch.isclose` [58], es:

$$|a - b| \leq atol + rtol \times |b| \quad (3.3.2)$$

donde a es la predicción del modelo base y b la del engine optimizado. Se fija $rtol = 0$ dado que el error físico relevante depende de su magnitud absoluta en Kelvin. La tolerancia $atol$ se define empíricamente como porcentaje del valor máximo de predicciones del modelo base dentro del percentil 90, evaluándose múltiples niveles para caracterizar la robustez de cada configuración de cuantización.

Métricas de rendimiento

La latencia se define como el tiempo de procesamiento de una única entrada, medido según el protocolo de eventos CUDA descrito previamente. Esta medición se circunscribe exclusivamente a la etapa de inferencia: no incluye el preprocesamiento del frame, ajeno al proceso de inferencia en sí, ni las operaciones de preparación de batch (expansión de dimensiones, conversión a tensor) o la transferencia CPU $\rightarrow$ GPU que el baseline Python requiere para construir la entrada al modelo, dado que ambos elementos son artefactos de la arquitectura de memoria discreta del modelo base y no del costo computacional del engine evaluado.

El throughput se deriva directamente de la latencia de inferencia mediana, calculando el número de inferencias por segundo que el engine sostendría en ejecución continua bajo la condición de batch unitario: $\text{FPS} = 1000 / \text{mediana (ms)}$. Esta derivación es válida en este contexto porque la medición de latencia ya aísla el costo puro de la etapa de inferencia sin contaminación de preprocesamiento, batching o transferencia CPU-GPU, de modo que no es necesaria una medición independiente de throughput como la empleada en la evaluación de pipeline completo (Sección 3.5).

3.4. Pipeline optimizado con NVIDIA DeepStream

Este bloque describe el diseño e implementación del pipeline de análisis de video en tiempo real construido sobre el SDK NVIDIA DeepStream [33]. A diferencia del baseline Python, donde

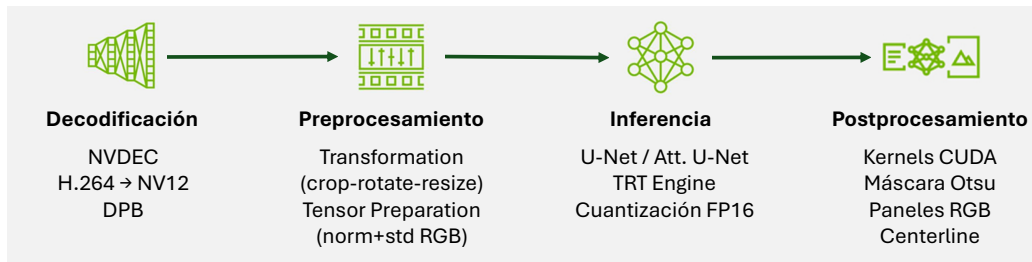


Fig. 3.8: Pipeline DeepStream implementado para la inferencia de temperatura de hollín.

las etapas de decodificación, preprocesamiento, inferencia y postprocesamiento se ejecutan de forma secuencial con intercambios frecuentes entre CPU y GPU para el bloque de inferencia según lo detallado en la Sección 3.2.1, el pipeline DeepStream mantiene los datos íntegramente en el espacio de memoria GPU desde la decodificación hasta la transmisión del resultado, eliminando las transferencias entre procesadores que constituyen el principal cuello de botella del baseline.

El pipeline opera en tres modos de salida intercambiables sin modificar su lógica interna, análogos a los del baseline Python: **fakesink** descarta los buffers procesados y se emplea para la medición de métricas de rendimiento sin costo de escritura; **filesink** almacena el resultado codificado en disco; y **udpsink** transmite el flujo en tiempo real a un receptor externo para visualización o almacenamiento remoto. La descripción del pipeline que sigue corresponde a la configuración con **udpsink**, que constituye el modo de operación principal.

La Figura 3.8 presenta una visión general del pipeline DeepStream implementado y sirve como referencia para la descripción detallada de sus componentes presentada a continuación.

3.4.1. Arquitectura general del pipeline

La implementación del modo de transmisión se construye como una cadena de elementos GStreamer conectados entre sí, según lo descrito en la Sección 2.4.2. La composición del pipeline es la siguiente:

```

filesrc ! qt demux ! h264parse ! nvv4l2decoder !
nvstreammux batch-size=1 width=2048 height=1536 !
nvdspreprocess config-file=config_preprocess.txt !
nvinfer config-file-path=config_infer_primary.txt !
nvdspostprocess postprocesslib-name=libpostprocess_impl.so
               postprocesslib-config-file=config_postprocess.txt !
nvv4l2h264enc bitrate=4000000 ! h264parse !
rtph264pay config-interval=1 pt=96 !
udpsink host=<IP> port=5000 sync=false

```

La ejecución de esta cadena de procesos es gestionada por GStreamer, que se encarga de la negociación de caps entre elementos adyacentes, la propagación de estados y el despacho de buffers entre etapas. Los elementos **nvdspreprocess**, **nvinfer** y **nvdspostprocess** son plugins DeepStream que extienden el modelo GStreamer con lógica acelerada en GPU y gestión de metadatos estructurados mediante **NvDsBatchMeta**.

El comportamiento de estos plugins se controla mediante archivos de configuración en texto plano, que permiten modificar parámetros operativos sin recompilar ningún componente del pipeline. Sin embargo, la flexibilidad que ofrece esta capa de configuración tiene límites precisos: los archivos

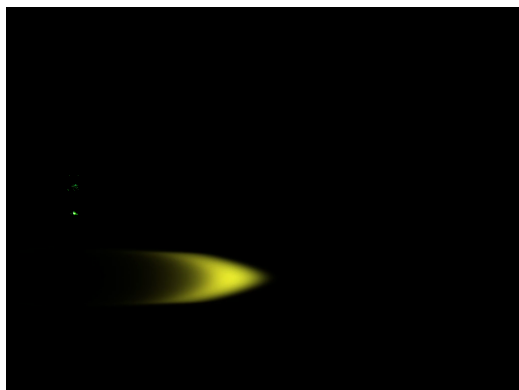


Fig. 3.9: Frame original decodificado por `nvv4l2decoder` en el pipeline DeepStream, sobre superficies GPU NV12 y previo a cualquier transformación geométrica o de formato (2048×1536 px).

`config_preprocess.txt` y `config_infer_primary.txt` exponen únicamente los parámetros previstos por el diseño del plugin, orientados a casos de uso convencionales de visión por computador como detección y segmentación de objetos. Cualquier comportamiento que se desvíe de ese marco, como la construcción de un tensor de entrada con normalización específica del dominio, la lectura del tensor de salida como campo denso de regresión, o la generación de una visualización científica personalizada, requiere extender los plugins mediante código C++ compilado en bibliotecas compartidas. Esta limitación estructural es la que motiva el uso de implementaciones personalizadas, descritas en las subsecciones siguientes.

3.4.2. Decodificación y multiplexación

La decodificación del video de entrada se realiza mediante `nvv4l2decoder`, que utiliza el motor NVDEC disponible en la plataforma Jetson para descomprimir el flujo H.264 directamente en superficies GPU de tipo `NVBUF_MEM_SURFACE_ARRAY`, sin intervención de la CPU [59]. La Figura 3.9 muestra un frame representativo tal como es entregado por este decodificador. Estas superficies son buffers DMA del hardware VIC (*Video Image Compositor*) de la Jetson, no memoria CUDA contigua, lo que tiene implicancias en las etapas posteriores de acceso y escritura.

Los frames decodificados se entregan al multiplexor `nvstreammux`, configurado con `batch-size=1`, `width=2048` y `height=1536`. El multiplexor agrupa los frames en el lote esperado por las etapas de inferencia y adjunta la estructura `NvDsBatchMeta` que acompaña al buffer a lo largo del pipeline.

3.4.3. Preprocesamiento acelerado

El preprocesamiento se implementa mediante el plugin `nvdspreprocess`, que expone dos puntos de extensión en C++: `CustomAsyncTransformation`, responsable de las transformaciones geométricas sobre la superficie de entrada, y `CustomTensorPreparation`, responsable de construir el tensor de entrada al modelo en formato NCHW normalizado.

Para que `nvinfer` consuma el tensor construido por `nvdspreprocess` en lugar de realizar su propio preprocesamiento interno, es necesario declararlo explícitamente mediante `input-tensor-from-meta=1` en su archivo de configuración; de lo contrario, el pipeline opera de inicio a fin sin errores visibles pero el modelo nunca recibe el tensor preprocesado correctamente. Del mismo modo, `output-tensor-meta=1` debe habilitarse para que el tensor de salida de la inferencia sea adjuntado como metadato en el buffer y pueda ser leído por el postprocesador personalizado. Estas directivas representan casos en

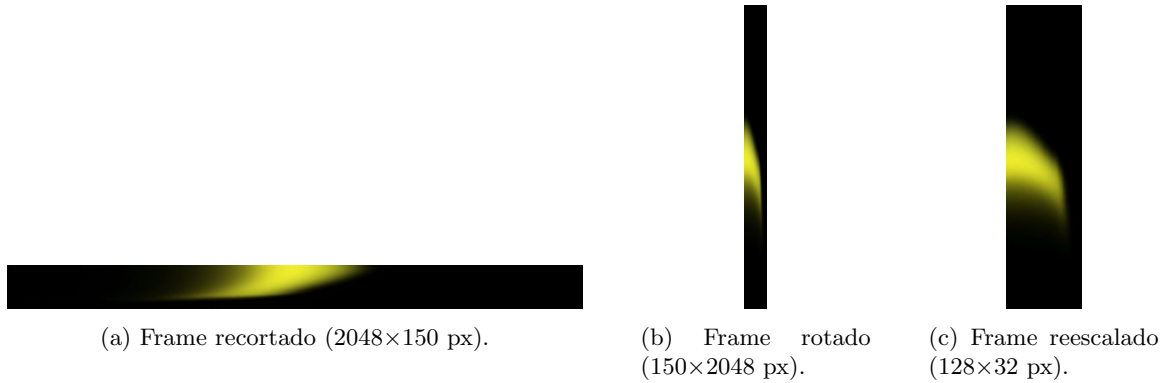


Fig. 3.10: Secuencia de transformaciones geométricas aplicadas en el pipeline DeepStream mediante `NvBufSurfTransform`: recorte, rotación y reescalado hasta la resolución de entrada del modelo.

que el comportamiento requerido existe en el plugin pero debe activarse explícitamente porque no corresponde al modo de operación predeterminado de DeepStream.

Transformaciones geométricas

Las transformaciones geométricas replican la secuencia del baseline Python de recorte, rotación y reescalado ilustrada en la Figura 3.10, pero se ejecutan íntegramente en GPU mediante `NvBufSurfTransform` [60]. La equivalencia matemática entre `NvBufSurfTransform.Rotate90` y la operación `transpose(1,0,2)[:,-1]` del baseline fue verificada numéricamente.

Las superficies intermedias para el recorte y la rotación (`prealloc_crop_surf` y `prealloc_rot_surf`) se pre-alocan una única vez durante la inicialización del plugin y se reutilizan en cada frame, eliminando el costo no determinístico de las allocaciones de memoria GPU por frame que constituirían la principal fuente de varianza en la latencia del preprocesamiento.

Construcción del tensor de entrada

La función `CustomTensorPreparation` construye el tensor de entrada al modelo a partir de la superficie RGBA resultante de las transformaciones geométricas. Para comprender los desafíos encontrados en esta etapa es necesario entender cómo DeepStream concibe el flujo de datos dentro de un pipeline típico.

En el caso de uso convencional de DeepStream para detección o segmentación de objetos en video, el frame original mantiene su utilidad a lo largo de todo el pipeline: la inferencia se ejecuta sobre una región o versión preprocesada del frame, pero el frame original sigue siendo necesario en el postprocesamiento para superponer sobre él los resultados visuales (bounding boxes, máscaras, etiquetas). Para facilitar este acceso posterior, `nvdspreprocess` convierte internamente la superficie de entrada a formato RGBA antes de llamar a `CustomTensorPreparation`, y dispone de la función `prepare_tensor` para construir el tensor float a partir de esa superficie RGBA intermedia. Esta cadena de transformaciones compuesta por decodificación a NV12[61], conversión NV12→RGBA, y conversión RGBA→tensor float, está diseñada para preservar el frame en un formato accesible por las etapas de composición posteriores.

En el contexto de pirometría de hollín, sin embargo, el frame original no tiene utilidad tras el preprocesamiento: la salida visual se construye completamente desde cero en el postprocesamiento a partir del campo de temperatura T_s y los canales RGB del tensor de entrada. La conversión

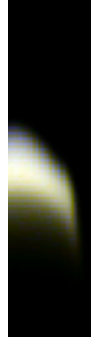


Fig. 3.11: Tensor de entrada al modelo en el pipeline DeepStream, tras reordenamiento de canales, normalización y estandarización.

NV12→RGBA era por tanto un paso heredado de la arquitectura general de DeepStream que no aportaba ningún beneficio en este caso de uso. Además de ser innecesaria, `prepare_tensor` presentaba un error crítico en plataforma Jetson: asumía un stride de fila de $W \times 4$ bytes (128 bytes para $W = 32$), cuando el hardware Jetson alinea las superficies CUDA a fronteras de memoria resultando en un pitch real de 256 bytes por fila. Al leer la superficie con stride incorrecto, la mitad de las filas del tensor se llenaban con bytes de padding (ceros), que al normalizarse producían un valor constante negativo, haciendo que el modelo produjera estimaciones de temperatura de ≈ 3500 K en lugar de los valores esperados de ≈ 2000 K.

La solución adoptada eliminó completamente `prepare_tensor` y la conversión NV12→RGBA, reemplazándolas por un kernel CUDA propio que opera directamente sobre la superficie RGBA resultante de `CustomAsyncTransformation` y construye el tensor en una única pasada. Este kernel lee la superficie respetando el pitch real obtenido de `batch->pitch`, aplica el reordenamiento de canales de RGB a BGR necesario para ser consistente con el convenio del baseline Python, y realiza la normalización y estandarización con las constantes del conjunto de entrenamiento almacenadas en memoria constante CUDA:

$$\text{chw}[c \cdot HW + \text{idx}] = \frac{v_c \cdot X_{\text{máx}}^{-1} - \mu_c}{\sigma_c}, \quad c \in \{B, G, R\} \quad (3.4.1)$$

donde v_c es el valor del canal c leído de la superficie RGBA con el pitch correcto. El resultado de esta etapa, mostrado en la Figura 3.11, corresponde al tensor CHW efectivamente consumido por el modelo. Las constantes $X_{\text{máx}}^{-1}$, $\{\mu_c\}$ y $\{\sigma_c\}$ se copian a memoria constante CUDA una única vez durante la inicialización mediante `cudaMemcpyToSymbol`, eliminando las transferencias de constantes por frame presentes en la implementación previa.

3.4.4. Inferencia

El plugin `nvinfer` ejecuta el engine TensorRT seleccionado en el bloque metodológico anterior sobre el tensor de entrada preprocesado. La configuración del plugin en `config_infer_primary.txt` especifica la ruta al plan serializado, las dimensiones del tensor de entrada (`infer-dims=3;128;32`), la precisión del engine y `output-tensor-meta=1`, que instruye al plugin a adjuntar el tensor de salida como metadato en los buffers GStreamer en lugar de procesarlo con el postprocesador interno estándar.

La activación de `input-tensor-from-meta=1` modifica la jerarquía en la que `nvinfer` adjunta el tensor de salida. Sin este parámetro, el tensor se adjunta directamente en `frame_user_meta_list`.

Con él activo, el tensor se anida dentro de un `NvDsRoiMeta`:

```
frame_user_meta_list
  NVDS_ROI_META
    roi_user_meta_list
      NVDSINFER_TENSOR_OUTPUT_META
```

El `OutputThread` del plugin `nvdspostprocess` fue modificado para recorrer esta jerarquía de dos niveles al buscar el tensor de salida.

3.4.5. Postprocesamiento y visualización

El postprocesamiento se implementa como una biblioteca compartida `.so` cargada dinámicamente por el plugin `nvdspostprocess`. La lógica principal reside en el `OutputThread`, que recibe el buffer `GStreamer` por cada frame del batch, extrae de sus metadatos el tensor de salida T_s (desde `NvDsInferTensorMeta`) y el tensor de entrada CHW (desde `GstNvDsPreProcessBatchMeta`), y los pasa a la función `ThermalRender_RenderFrame`.

Todos los buffers GPU necesarios para el postprocesamiento se pre-allocan una única vez en `ThermalRender_Init` mediante la estructura `GpuBuffers`, eliminando el overhead no determinístico de `cudaMalloc` por frame. Este mismo principio de pre-alocación se extiende a los elementos estáticos de la visualización, que se construyen una única vez durante la inicialización en lugar de recalcularse en cada frame, según se detalla en las subsecciones siguientes.

Segmentación por Otsu

La máscara de Otsu se calcula sobre el canal G del tensor CHW de entrada, siguiendo el mismo criterio que el baseline Python según la formulación general detallada en la Sección 3.1.5: la Figura 3.12a muestra la salida cruda T_s previa a este procesamiento, y la Figura 3.12b la máscara binaria resultante. La implementación consta de tres kernels encadenados, todos ellos GPU-resident: `otsu_histogram_kernel` construye el histograma de 256 niveles del canal G normalizado mediante reducción en memoria compartida para minimizar la contención atómica global; `otsu_threshold_kernel` calcula el umbral óptimo maximizando la varianza inter-clase mediante un scan paralelo (Hillis-Steele [62]) de las sumas acumuladas y una reducción de `argmax`, escribiendo el resultado directamente en memoria de dispositivo; y `apply_threshold_kernel` genera la máscara binaria leyendo dicho umbral por puntero. Ningún valor intermedio de esta cadena transita por el host, por lo que la etapa completa se ejecuta como una secuencia de kernels encadenados en el mismo stream CUDA sin puntos de sincronización explícita.

El mismo criterio de diseño se aplica al cálculo de mínimo y máximo requerido para normalizar los cuatro canales relevantes (G para Otsu, y B, G, R para los paneles de colormap, Sección 3.4.5): `minmax_reduce_kernel` escribe su resultado en un buffer de ocho flotantes (cuatro pares independientes) que los kernels de normalización correspondientes consumen directamente por puntero. Dado que no existe dependencia de datos entre canales, las cuatro reducciones se encolan de forma consecutiva sin necesidad de esperar el resultado de ninguna de ellas antes de lanzar la siguiente.

Construcción de paneles de visualización

Los paneles de temperatura se construyen mediante dos familias de kernels CUDA de colormap. El kernel `normalise_ts_kernel` desnormaliza el tensor T_s aplicando la transformación inversa $T_K = T_s \cdot \sigma_{T_s} + \mu_{T_s}$ y remapea los valores al rango físico $[T_{\min}, T_{\max}]$. El kernel `colormap_ts_kernel` aplica la LUT INFERNO [63] sobre los píxeles de la región enmascarada, asignando blanco a los

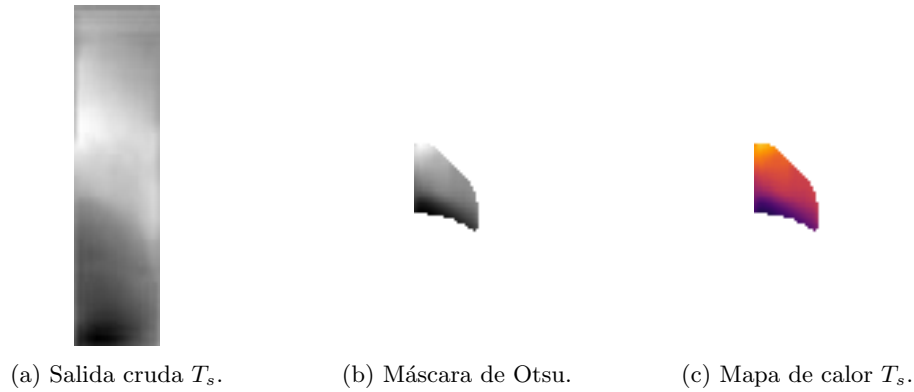


Fig. 3.12: Salida cruda del modelo, máscara de Otsu calculada sobre el canal G del tensor de entrada mediante la cadena de kernels GPU-resident `otsu_histogram_kernel`, `otsu_threshold_kernel` y `apply_threshold_kernel`, y mapa de temperatura resultante tras desnormalizar y enmascarar la temperatura T_s predicha, en el pipeline DeepStream.

píxeles de fondo, tal como se observa en la Figura 3.12c. Análogamente, los tres canales del tensor de entrada, reordenados de BGR a RGB para su presentación visual, se normalizan por mínimo y máximo y se colorean con la LUT VIRIDIS [63] mediante `normalise_channel_kernel` y `colormap_channel_kernel`. Los cuatro paneles resultantes, T_s , R, G, B, cada uno de 128×32 píxeles, se componen mediante `composite_panel_kernel` en un canvas cuadrado de 128×128 píxeles.

Este canvas se escala mediante interpolación bilineal (`scale_bilinear_kernel`) a un cuadrado cuyo lado es igual a la altura disponible para el área de imagen del panel, preservando la proporción 1:1 del canvas fuente y evitando así cualquier deformación anisótropa de los cuatro sub-paneles. El cuadrado resultante se centra horizontalmente sobre un fondo estático mediante `composite_panel_kernel`, dejando el remanente como margen en blanco.

El fondo estático del panel, las etiquetas T_s , R, G, B sobre cada sub-panel, las colorbars de ambas paletas de color (INFERNO para T_s , VIRIDIS para los canales de entrada) junto con sus valores de rango, y el texto fijo asociado a la temperatura media, se construyen una única vez en `ThermalRender_Init` mediante OpenCV en CPU y se transfieren a memoria de dispositivo, dado que ninguno de estos elementos depende del contenido del frame. El único valor dinámico de esta región es la temperatura media dentro de la máscara, calculada por frame mediante `mean_reduce_kernel` (reducción de bloque único análoga a `minmax_reduce_kernel`) y ...estampada sobre el fondo estático mediante el mecanismo de renderizado de texto descrito en la Sección 3.4.5. La Figura 3.13 muestra el panel resultante para un frame representativo.

Gráfico centerline

El perfil de temperatura centerline se construye muestreando la columna central del campo T_s sobre los píxeles de la región enmascarada y trazando la curva $T(z)$. El fondo estático del gráfico (ejes, ticks y etiquetas numéricas del eje de temperatura, y las etiquetas fijas de ambos ejes) se pre-computa una única vez en `ThermalRender_Init`, de forma análoga al fondo del panel de colormap. Por frame, cuatro kernels operan directamente sobre los buffers ya residentes en GPU (`ts_norm` y `mask`), sin transferencia alguna al host: `column_span_kernel` determina, mediante una reducción de bloque único, el rango de filas enmascaradas de la columna muestreada; `draw_z_ticks_kernel` traza las marcas del eje z en las posiciones correspondientes a dicho rango y registra los centros y valores de las etiquetas a renderizar; `draw_curve_kernel` traza la curva $T(z)$ mediante un algoritmo DDA

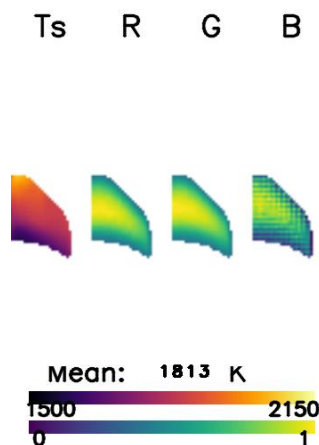


Fig. 3.13: Panel de colormap completo en el pipeline DeepStream: los cuatro sub-panes T_s , R, G, B con sus etiquetas de canal, ambas colorbars (INFERNO y VIRIDIS) y la temperatura media dentro de la máscara.

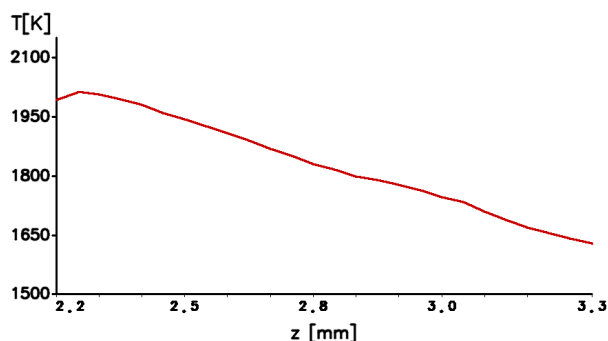


Fig. 3.14: Gráfico centerline $T(z)$ del pipeline DeepStream, construido íntegramente en GPU, mostrando el perfil de temperatura sobre el eje radial central de la llama.

(*Digital Differential Analyzer*) [64] que conecta cada par de filas enmascaradas consecutivas, dado que la pendiente entre puntos consecutivos es arbitraria y puede exceder un píxel de desplazamiento horizontal; y `blit_no_data_kernel` sustituye el gráfico completo por un aviso estático cuando ninguna fila de la columna muestreada se encuentra dentro de la máscara. La Figura 3.14 presenta el resultado de esta construcción para el mismo frame representativo empleado en las figuras anteriores.

Al ejecutarse la totalidad de esta etapa en GPU sobre datos ya residentes en dispositivo, el gráfico centerline resulta contemporáneo al panel de temperatura del mismo frame, sin transferencia device-to-host ni latencia introducida entre ambos elementos de la visualización. Las etiquetas numéricas dinámicas del eje z , cuyo valor depende del rango muestreado en cada frame, se renderizan mediante el mecanismo de texto descrito en la Sección 3.4.5.

Renderizado de texto en GPU

Dado que no existen primitivas nativas de CUDA para el trazado de texto, se implementó un mecanismo de atlas de glifos para los elementos de texto cuyo contenido varía por frame: las etiquetas

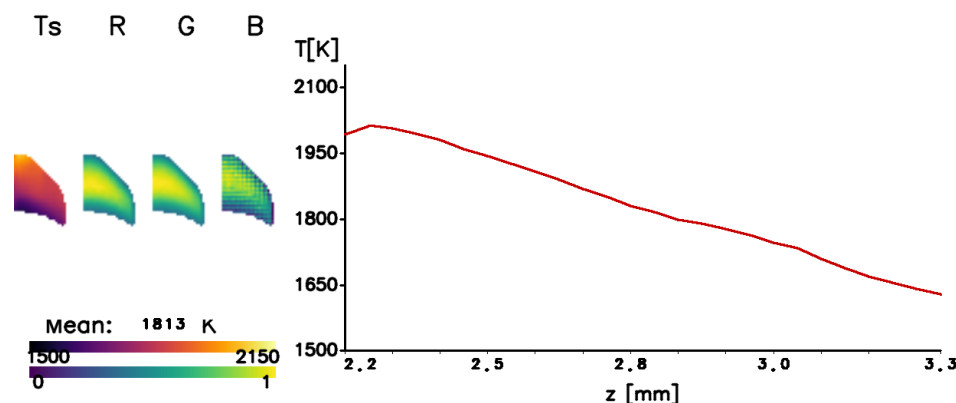


Fig. 3.15: Canvas final del pipeline DeepStream, compuesto por el panel de colormap y el gráfico centerline lado a lado (1280×512 px).

del eje z del centerline (Sección 3.4.5) y la temperatura media del panel de colormap (Sección 3.4.5). Los elementos de texto fijos (etiquetas de ejes, nombres de canal, valores de rango de las colorbars) no requieren este mecanismo, dado que se renderizan directamente con OpenCV como parte de los fondos estáticos descritos en las subsecciones anteriores.

El atlas contiene los glifos de los dígitos 0-9 y el punto decimal, generados mediante OpenCV en `ThermalRender_Init` y almacenados como un arreglo de celdas de tamaño fijo en memoria de dispositivo; el tamaño de celda se deriva de las dimensiones del canvas de salida, por lo que el atlas se regenera automáticamente a la resolución correspondiente ante un cambio de las constantes de dimensión del canvas. La conversión de un valor de punto flotante o entero a la secuencia de códigos de glifo correspondiente se realiza en dispositivo mediante funciones `device` de formateo manual de dígitos, dado que no se dispone de una rutina equivalente a `snprintf` utilizable en el contexto de un kernel. El kernel `draw_glyphs_kernel` estampa dichas secuencias sobre el canvas correspondiente, centrando cada etiqueta horizontalmente respecto a la posición de referencia provista por el kernel que la generó.

Composición, escalado y conversión de formato

Los dos paneles descritos, el panel de colormap (Sección 3.4.5) y el gráfico centerline (Sección 3.4.5), se componen lado a lado en un canvas final mediante `composite_panel_kernel`, como se muestra en la Figura 3.15. El resultado corresponde a una resolución de 1280×512 píxeles, determinada experimentalmente como un balance entre calidad visual y throughput del pipeline².

La composición distingue entre contenido derivado de datos muestreados (los cuatro paneles de colormap, escalados con interpolación bilineal desde su resolución nativa de 128×128) y contenido geométrico sintético (el centerline y los fondos estáticos, generados directamente a su resolución final sin ningún paso de reescalado adicional). Esta distinción evita la introducción de artefactos de bloque característicos del escalado por vecino más cercano sobre elementos vectoriales como ejes, texto y la curva del centerline.

²Se evaluó también una configuración a la resolución nativa del buffer de entrada (2048×1536 píxeles). Esta prueba está relacionada con el experimento de relleno de fondo descrito en la Sección 3.4.5, en tanto ambos exploran el efecto de modificar las dimensiones relativas entre el contenido de visualización y el buffer sobre el que se escribe; a dicha resolución, la corrección de proporción descrita para los paneles de colormap resultaba insuficiente para evitar artefactos de estabilidad entre frames consecutivos, y se optó por revertir a la resolución de 1280×512 antes de continuar con el resto de la implementación.

El canvas final se convierte de BGRA a NV12[61] mediante `bgra_to_nv12_kernel` usando la matriz de transformación BT.601 rango limitado [65], produciendo los planos Y y UV separados requeridos por el codificador de hardware.

Write-back al buffer de video

La escritura del canvas NV12 sobre el buffer de video que gestiona DeepStream refleja directamente la arquitectura in-place del framework. DeepStream no está diseñado para que un plugin de postprocesamiento produzca un buffer de salida de dimensiones distintas al buffer de entrada; en cambio, el buffer decodificado fluye a lo largo de todo el pipeline y el postprocesador lo modifica en el lugar, sobrescribiendo la región correspondiente con el resultado de la visualización. Esta es la arquitectura natural para los casos de uso convencionales de DeepStream, donde la salida es el mismo video de entrada con anotaciones superpuestas; un supuesto que, según se detalla en la Sección 3.4.5, no se ajusta plenamente al caso de uso de este proyecto.

Dado que las superficies decodificadas por `nv412decoder` son buffers DMA de tipo `NVBUF_MEM_SURFACE_ARRAY` no directamente accesibles como memoria CUDA, la ruta de escritura requiere una superficie intermedia `cuda_surf` pre-alocada con `NvBufSurfaceCreate`. Los planos NV12 generados en GPU se copian al buffer mapeado de `cuda_surf` mediante `cudaMemcpy2DAsync` respetando el pitch de alineación del hardware, y el resultado se transfiere al buffer de destino mediante `NvBufSurfTransform`, centrado según las dimensiones del buffer de destino.

Limitación: fondo residual del frame original

El diseño in-place de DeepStream, descrito en la Sección 3.4.5, asume implícitamente que el contenido escrito por un elemento de postprocesamiento ocupa la totalidad (o una anotación superpuesta parcial pero visualmente integrada) del buffer de video sobre el que opera, dado que el caso de uso convencional del framework es la superposición de anotaciones (cajas delimitadoras, máscaras de segmentación, texto) sobre el mismo video de entrada. El presente proyecto se aparta de este supuesto: el buffer decodificado se emplea únicamente como fuente para el preprocesamiento y la inferencia, mientras que el contenido final relevante para el usuario es la visualización sintética descrita en las subsecciones anteriores, de dimensiones (1280×512) sustancialmente menores al buffer de entrada configurado en `nvstreammux` (2048×1536). Como consecuencia, la región del buffer de destino no cubierta por el canvas de visualización conserva el contenido del frame original decodificado, en lugar de un fondo uniforme, produciendo un artefacto visual de video de fondo detrás de la visualización compuesta.

Se evaluó una corrección en tiempo de ejecución consistente en aplicar un relleno de color sólido sobre la totalidad del buffer de destino mediante una segunda transformación `NvBufSurfTransform`, previa al blit del contenido de visualización, utilizando una superficie auxiliar pre-alocada de color uniforme como fuente. Esta aproximación resultó inviable por una combinación de restricciones de hardware y de diseño. El motor de transformación por hardware (VIC) impone un límite de factor de escala de 16× por eje [60]; una superficie fuente de dimensiones mínimas excede ampliamente dicho límite al escalarse al tamaño del buffer de destino, forzando una degradación a transformación por GPU y, en la configuración probada inicialmente, un fallo directo de la operación. Ampliar la superficie fuente lo suficiente para satisfacer esta restricción resolvió el fallo de hardware, pero expuso un segundo problema: al no existir sincronización explícita entre las dos transformaciones `NvBufSurfTransform` consecutivas sobre el mismo buffer de destino, se observaron artefactos de parpadeo atribuibles a un posible solapamiento o reordenamiento de su ejecución en el motor VIC, a la vez que el tiempo de ejecución medido para un mismo lote de frames se incrementó en aproximadamente un 60 % respecto de la configuración sin relleno de fondo. Introducir la sincronización explícita necesaria para eliminar dicho solapamiento habría incrementado aún más este costo, dado

que forzaría al pipeline a esperar la finalización completa de una transformación de pantalla completa antes de iniciar la siguiente, en lugar de permitir su solapamiento con otro trabajo del stream. Ante una penalización de esta magnitud por una mejora meramente estética, sin efecto sobre el contenido informativo de la visualización, se descartó continuar por esta vía.

La alternativa estructural consiste en reducir la resolución configurada en `nvstreammux` para que coincida con las dimensiones del canvas de visualización, eliminando así la región residual en su origen en lugar de enmascararla en cada frame. Esta alternativa es viable porque `nvstreammux` ya reescala el frame decodificado a las dimensiones configuradas como parte de su operación normal de batching, independientemente de la resolución nativa del video de entrada; en consecuencia, reducir dicho valor no introduce una etapa de reescalado adicional, sino que redirige a un destino de menor tamaño una operación que ya se ejecuta en cada frame, con un volumen de datos asociado considerablemente menor. Sin embargo, esta modificación tendría un efecto en cascada sobre el resto del pipeline: tanto la configuración de `nvdspreprocess` como la de `nvinfer` están definidas en función de las dimensiones del frame muxeador 2048×1536, por lo que un cambio en dicho valor requeriría revisar y potencialmente ajustar ambas configuraciones para preservar el comportamiento esperado del modelo de inferencia. Dado que las dimensiones finales de la visualización no constituyen, en la fase experimental actual del proyecto, un valor estable de producto, se optó por no acoplar la configuración de captura de todo el pipeline a una decisión de visualización aún sujeta a cambios, documentando en su lugar el fondo residual como limitación conocida y postergando dicha reconfiguración a una fase posterior de estabilización.

3.4.6. Modos de operación

El pipeline expone tres modos de operación diferenciados por el elemento sink, análogos a los del baseline Python. El modo `fakesink` (`sync=false`) descarta los buffers procesados sin escritura ni codificación, y se emplea para la medición de métricas de rendimiento end-to-end con un costo mínimo de salida. Los modos `filesink` y `udpsink` añaden la etapa de codificación H.264 mediante el codificador de hardware `nvv4l2h264enc` y la encapsulación correspondiente, y constituyen los modos de despliegue operativo del sistema.

Esta distinción entre `fakesink` como métrica baremo y `filesink` y `udpsink` como referencia de despliegue se mantiene consistente a lo largo de las tres métricas de rendimiento evaluadas en la Sección 3.5: latencia, throughput y uso de memoria.

3.5. Evaluación comparativa de rendimiento

Esta sección define el protocolo de evaluación mediante el cual se comparan el baseline Python y el pipeline DeepStream optimizado. El objetivo es caracterizar el rendimiento de cada sistema en términos de latencia, throughput y uso de memoria bajo condiciones controladas y reproducibles, de modo que las diferencias observadas sean atribuibles a las decisiones de diseño de cada pipeline y no a variaciones en el entorno de medición.

3.5.1. Principios de medición

Dos propiedades del sistema que deben medirse y reportarse de forma diferenciada son la latencia y el throughput [66], cuyas definiciones operacionales precisas se detallan en la subsección siguiente. En un pipeline segmentado donde las etapas operan en paralelo, ambas métricas son conceptualmente ortogonales: el throughput en estado estacionario queda determinado por la cadencia de la etapa más lenta, mientras que la latencia depende del tiempo acumulado de todas las etapas para un frame individual. Un sistema puede exhibir alto throughput y, al mismo tiempo, una latencia

considerable por frame; mezclar ambas métricas o derivar una de la otra sin esta distinción produce interpretaciones incorrectas del rendimiento [67].

En ambos pipelines, se utiliza batch size unitario por las razones establecidas en la Sección 3.1.5, lo que implica que el throughput del sistema en modo de streaming queda acotado por la inversa de la latencia de la etapa dominante. Sin embargo, medir el throughput como métrica global independiente sigue siendo relevante, ya que captura efectos de inicialización, colas internas y sincronizaciones que no son visibles en la medición frame a frame de latencia.

3.5.2. Definición de métricas

La comparación entre el pipeline baseline y el pipeline DeepStream se realiza a partir de tres métricas complementarias: latencia end-to-end, que captura el tiempo de respuesta por frame relevante para aplicaciones de monitoreo en tiempo real; throughput, que determina la capacidad de procesamiento sostenido del sistema bajo carga continua; y uso de memoria, que condiciona la viabilidad de despliegue en la plataforma embebida de recursos limitados. Las subsecciones siguientes detallan la definición operacional de cada métrica y el procedimiento de instrumentación empleado en cada pipeline.

Latencia end-to-end

La latencia se define como el tiempo total que tarda un único frame en recorrer el pipeline completo, desde su entrada al primer elemento de procesamiento hasta la disponibilidad de su resultado. Esta medición es end-to-end e incluye todas las etapas: decodificación, preprocesamiento, inferencia y postprocesamiento.

En el baseline Python, la latencia se mide con `time.perf_counter()` delimitando el intervalo desde el inicio de la decodificación del frame hasta la finalización del postprocesamiento y la generación de la visualización. Dado que PyTorch ejecuta operaciones GPU de forma asíncrona, se aplica `torch.cuda.synchronize()` tras la inferencia para asegurar que todas las operaciones en GPU hayan finalizado antes de registrar el tiempo final. No se emplean CUDA streams personalizados, transferencias asíncronas (`non_blocking=True`) ni memoria pinned en la medición de latencia, con el objetivo de aislar el comportamiento secuencial del sistema y obtener una línea base clara y reproducible [53].

Además de la latencia end-to-end, el baseline Python reporta un desglose por etapa: decodificación, preprocesamiento, inferencia y postprocesamiento + visualización. Para efectos de comparación directa con el pipeline DeepStream, la preparación del batch y la transferencia CPU → GPU se agrupan dentro de la etapa de preprocesamiento según lo descrito en la Sección 3.2.1, ya que el pipeline DeepStream, al operar sobre memoria unificada, no posee una etapa de transferencia explícita equivalente. Esta agrupación permite una comparación de cuatro etapas homóloga a la empleada en DeepStream, aunque ambos pipelines difieran en su arquitectura de memoria subyacente: la cifra de "Preprocesamiento" reportada para el baseline Python incluye la preparación del batch, y la cifra de "Inferencia" incluye la transferencia CPU → GPU además del cómputo propiamente tal.

En el pipeline DeepStream, la latencia end-to-end se mide mediante pad probes de GStreamer insertadas en los pads de entrada del elemento `nvv4l2decoder` y de salida del elemento `nvdsppostprocess` [32]. Cada probe registra un timestamp con `time.perf_counter_ns()` en el momento en que el buffer cruza el pad correspondiente. El emparejamiento de timestamps de entrada y salida se realiza mediante colas FIFO independientes por etapa, lo que garantiza la correlación correcta de frames incluso cuando `nvstreammux` reescribe el PTS (*Presentation TimeStamp*) de los buffers al construir el batch. Adicionalmente, se instrumentan las etapas intermedias (`nvdsppreprocess`, `nvinfer`, `nvdsppostprocess`) con pares de probes entrada-salida para obtener la latencia desagregada por

etapa, lo que permite identificar los componentes dominantes del tiempo de procesamiento.

Es importante señalar que las pad probes en Python requieren adquirir el GIL en cada disparo, lo que introduce una penalización en el hilo de streaming de GStreamer. Este overhead es sistemático y se cancela parcialmente en el cálculo de diferencias dentro de la misma etapa, pero reduce el throughput global del pipeline respecto a la ejecución nativa con `gst-launch-1.0`. Por esta razón, las mediciones de latencia por etapa se realizan con el script Python instrumentado, mientras que las mediciones de throughput de referencia se obtienen de forma separada a través de ejecuciones `gst-launch-1.0` sin instrumentación interna, como se indica a continuación.

En ambos pipelines se reportan la mediana (p50) como métrica principal de latencia en estado estacionario, junto con el percentil 95 (p95) y el valor máximo como información complementaria, calculados sobre los frames posteriores al periodo de warm-up (Sección 3.5.4). La mediana representa el comportamiento típico del sistema sin verse afectada por los valores extremos del arranque, mientras que el p95 caracteriza el peor caso práctico y resulta de especial interés para aplicaciones con requisitos de tiempo real. El valor máximo se incluye como referencia adicional.

Estas métricas se presentan para los tres modos de sink descritos en la Sección 3.4.6, utilizando `fakesink` como baremo del rendimiento de procesamiento puro y `filesink/udpsink` como referencia de escenarios de despliegue. En los tres casos, la latencia se mide con `disable-dpb=true` en el elemento `nvv4l2decoder`, ya que esta configuración constituye la técnica que permite aislar el procesamiento frame a frame del pipeline, como se desarrolla en la Sección 3.5.3).

Throughput

El throughput se define como la cantidad de frames procesados por unidad de tiempo bajo carga continua. Se calcula dividiendo el número de frames procesados por el tiempo de ejecución medido en estado estacionario, excluyendo el periodo de warm-up.

En el baseline Python, el throughput se mide con un reloj global que se activa únicamente una vez completado el warm-up: el temporizador se inicia (tras una sincronización CUDA) en el instante en que se alcanza el frame correspondiente al 10% de warm-up, y se detiene al finalizar el último frame, con una sincronización final adicional. De este modo, ni el tiempo ni el conteo de frames del periodo de arranque contaminan el FPS reportado. Se utiliza `pin_memory()` y transferencias `non_blocking=True` para aproximar las condiciones de un escenario de operación real, a diferencia de la medición de latencia.

En el pipeline DeepStream, el throughput de referencia se obtiene ejecutando el pipeline completo con `gst-launch-1.0` sin instrumentación Python, lo que elimina el overhead del GIL y refleja el rendimiento nativo del sistema. La medición se apoya en el mensaje que `gst-launch-1.0` imprime al finalizar la ejecución (**Execution ended after H:MM:SS.nnnnnnnnn**). Este cronómetro se activa únicamente una vez completada la transición del pipeline a estado **PLAYING**; la construcción del pipeline y la transición previa a **PAUSED** quedan estructuralmente fuera de la ventana medida, y para fuentes no-live esta última transición dispara además el mecanismo de *preroll*, en el que se empuja al menos un buffer a través de todo el pipeline antes de que el cronómetro comience a correr.

A diferencia del baseline Python, en el pipeline nativo no es posible descartar frames individuales de warm-up sin reintroducir instrumentación por pad probe (con su correspondiente overhead de GIL), lo que anularía el propósito de esta medición de referencia. Se optó en su lugar por el método de *truncated replications* [68], empleado en simulación de eventos discretos para remover el sesgo de arranque de sistemas de estado estacionario sin necesidad de instrumentación interna: el mismo pipeline se ejecuta dos veces sobre el mismo video, truncado a distintas longitudes mediante el elemento `identity eos-after=N` insertado inmediatamente después del decoder (forzando un EOS (*End Of Stream*) real tras N buffers decodificados), y el throughput de estado estacionario se obtiene

por diferencia:

$$\text{FPS}_{\text{estable}} = \frac{N_{\text{total}} - N_{\text{warmup}}}{T_{\text{total}} - T_{\text{warmup}}}$$

Esta resta cancela cualquier costo fijo de arranque presente en ambas corridas por igual. El procedimiento se repite 10 veces para cada longitud, emparejando cada repetición de la corrida corta con la correspondiente de la corrida larga, y agregando el resultado por promedio y desviación estándar entre pares. Adicionalmente, se calcula el margen de incertidumbre de la diferencia de tiempos mediante propagación de errores estándar ($SE = \sqrt{SE_{\text{warmup}}^2 + SE_{\text{total}}^2}$), lo que permite distinguir si una diferencia observada entre el FPS ingenuo (frames totales sobre tiempo total, sin corrección) y el FPS de estado estacionario constituye evidencia real de overhead de arranque o es indistinguible del ruido de medición.

Se reporta el FPS de estado estacionario como métrica principal del throughput nativo, junto con el FPS ingenuo y su diferencia porcentual respecto al valor estable como información complementaria, lo que permite evaluar en qué medida el costo de arranque del pipeline DeepStream afecta una medición simple no corregida. A diferencia de la latencia, el throughput se mide con el decoder en su configuración nativa (DPB activo), como se justifica en la Sección 3.5.3. Estas métricas se presentan para los tres modos de sink descritos en la Sección 3.4.6, utilizando **fakesink** como baremo del rendimiento de procesamiento puro y **filesink/udpsink** como referencia de escenarios de despliegue.

Uso de memoria

El uso de memoria se evalúa sobre la totalidad de la plataforma Jetson, que en su arquitectura de memoria unificada no distingue entre RAM (*Random Access Memory*) del host y VRAM (*Video Random Access Memory*) del device. Bajo el supuesto de que el dispositivo está dedicado íntegramente a la ejecución del pipeline durante la evaluación, el consumo total del sistema es la métrica principal reportada, ya que representa el footprint operativo real bajo ese supuesto de despliegue. Como referencia complementaria, se reporta también el consumo neto (total menos el consumo del sistema en reposo), útil para aislar el costo atribuible específicamente al pipeline evaluado.

El consumo de memoria se monitorea con **tegrastats** [69] a un intervalo de muestreo de 10001000 1000 ms, que corresponde tanto al valor por defecto de la herramienta [70] como a la granularidad reportada en trabajos previos de caracterización de dispositivos Jetson [71], donde se observa que el propio proceso de monitoreo introduce un overhead no despreciable sobre las mediciones, lo que motiva mantener un intervalo moderado en lugar de maximizar la resolución temporal. El consumo en reposo (*idle*) se estima de forma independiente al inicio de cada una de las 10 repeticiones, promediando 3 muestras consecutivas de **tegrastats** inmediatamente antes de lanzar el proceso bajo medición. Este cálculo por repetición, en lugar de una única estimación global al inicio del bloque de pruebas, permite que el valor neto reportado absorba variaciones del entorno de ejecución entre repeticiones, en lugar de asumir un nivel de reposo constante a lo largo de toda la evaluación.

Para cada configuración evaluada se realizan 10 repeticiones independientes, relanzando tanto **tegrastats** como el proceso bajo medición en cada una. Se reportan el promedio y la desviación estándar, entre repeticiones, tanto del valor máximo como del promedio de RAM observados (total y neto).

En el baseline Python, la memoria se mide bajo la misma configuración de la medición de throughput (batch_size unitario, transferencias con **pin_memory** y **non_blocking=True**, descartando los resultados de inferencia), y no bajo la configuración de latencia, ya que se busca aproximar un escenario de operación continua representativo, en lugar del aislamiento artificial empleado para medir latencia por frame.

En el pipeline DeepStream, el uso de memoria se mide ejecutando el pipeline `gst-launch-1.0` completo sobre el conjunto de evaluación, con el decoder en su configuración nativa (DPB activo), consistente con la medición de throughput y representativa del footprint de memoria bajo operación sostenida real. Los resultados se reportan para los tres modos de sink descritos en la Sección 3.4.6, utilizando `fakesink` como baremo del consumo de memoria del procesamiento puro. Esta distinción es particularmente relevante en esta métrica, ya que el encoder H.264 activo en `filesink` y `udpsink` introduce un footprint adicional que no está presente en `fakesink`.

3.5.3. Fenómenos estructurales que condicionan las mediciones

Las mediciones de latencia en el pipeline DeepStream están afectadas por dos fenómenos inherentes a la arquitectura del sistema que deben comprenderse para interpretar correctamente los resultados, en particular los asociados a la etapa de decodificación.

Decoded Picture Buffer (DPB)

El estándar H.264 permite codificar frames fuera del orden de presentación, utilizando referencias cruzadas entre frames de tipo B y P para maximizar la eficiencia de compresión. Como consecuencia, el decoder debe acumular un número de frames en un buffer interno, el DPB, antes de poder entregar el primero en el orden de presentación correcto [72]. Este mecanismo es estructural al estándar y no impone un límite superior definido en la latencia que introduce: depende del perfil de codificación, del número de frames de referencia declarados en el stream y de la implementación del decoder.

En la medición por pad probe, el intervalo registrado para la etapa de decodificación abarca desde la entrega del NAL (*Network Abstraction Layer*) unit comprimido al plugin hasta la entrega del frame decodificado al elemento siguiente. Este intervalo incluye el tiempo de hardware NVDEC (típicamente estimado en 5–15 ms para H.264 en plataformas Jetson [33]) más el tiempo de retención en el DPB hasta que el frame puede ser entregado en orden. La diferencia entre la latencia observada por pad probe y el tiempo de hardware estimado es atribuible en gran parte a este efecto.

Para aislar el procesamiento frame a frame en la medición de latencia, se emplea de forma estándar la propiedad `disable-dpb=true` del elemento `nvv4l2decoder`, que instruye al decoder a entregar frames inmediatamente sin esperar a completar el buffer de reordenamiento. Esta configuración reduce la latencia de decodificación a costa de introducir riesgo de artefactos visuales si el stream contiene B-frames con reordenamiento significativo; en el contexto de este trabajo, donde los videos fueron codificados con `libx264` y parámetros controlados, este riesgo es acotado, lo que permite adoptar `disable-dpb=true` como configuración estándar de medición para las métricas de latencia (por etapa y end-to-end) reportadas en el siguiente Capítulo, en las tres implementaciones de sink evaluadas. El throughput y el uso de memoria se miden, en cambio, con DPB activo; la justificación de esta distinción se desarrolla en la Sección 3.5.3.

Backpressure del nvstreammux

El elemento `nvv4l2decoder` en Jetson gestiona un pool de buffers de salida de tamaño fijo (típicamente cuatro buffers) que son asignados al pipeline aguas abajo. Cuando `nvstreammux` no puede consumir frames a la misma velocidad a la que el decoder los produce, los frames decodificados permanecen retenidos dentro del pool del decoder hasta que el elemento siguiente los solicita [73]. El tiempo de espera en este pool queda registrado como parte de la latencia de la etapa de decodificación por la pad probe, aunque no corresponde al costo de decodificación en sí sino al tiempo de cola entre etapas.

Este fenómeno explica por qué la latencia medida de la etapa de decodificación es sistemáticamente mayor que el tiempo de hardware NVDEC, y por qué presenta mayor varianza que las

etapas de preprocesamiento e inferencia: la variabilidad en el tiempo de consumo del `nvstreammux` se propaga hacia atrás y se manifiesta como variabilidad en la latencia observada del decoder. En la comparación entre configuraciones de sink, este backpressure se modifica cuando el encoder introduce regulación de flujo aguas abajo, lo que altera indirectamente la distribución de latencias observadas en la decodificación.

Implicancias para el protocolo de medición

Ambos fenómenos descritos explican por qué la latencia de decodificación observada por pad probe combina el costo de hardware NVDEC, el retardo de reordenamiento del DPB y el tiempo de espera por consumo de `nvstreammux`, en lugar de reflejar únicamente el costo de decodificación.

La latencia end-to-end reportada en el siguiente Capítulo, medida con `disable-dpb=true`, aísla por tanto el costo de decodificación por hardware más el backpressure de `nvstreammux`, lo que permite una comparación reproducible y homóloga entre pipelines y sinks. Esta cifra constituye, no obstante, una cota inferior de la latencia de despliegue real en escenarios donde el stream de entrada requiera un DPB activo (por ejemplo, B-frames con múltiples referencias que no admitan entrega inmediata sin riesgo de artefactos); en este trabajo, con streams codificados bajo parámetros controlados que acotan ese riesgo, la cota se considera representativa.

El throughput y el uso de memoria, en cambio, se miden con DPB activo: en estado estacionario el decoder opera con el buffer completamente lleno y entrega frames a la cadencia de la etapa más lenta del pipeline, sin los retardos transitorios del arranque. El posible efecto del backpressure sobre el throughput se controla verificando, durante los experimentos, la ausencia de errores de pool en el bus de GStreamer.

3.5.4. Configuración experimental

Todas las mediciones se repiten 10 veces por configuración evaluada. La agregación entre repeticiones se realiza por *promedio de estadísticos*: cada estadístico relevante se calcula de forma independiente dentro de cada repetición, y el valor final reportado es el promedio de ese estadístico entre las 10 repeticiones, junto con su desviación estándar entre corridas. Este criterio se prefirió por sobre concatenar las muestras de todas las repeticiones en un único arreglo, ya que preserva la variabilidad entre corridas independientes como una medida explícita de incertidumbre, en lugar de diluirla dentro de una única distribución agregada.

Para garantizar un régimen estable de ejecución previo a las mediciones, se aplica un warm-up equivalente al 10% del conjunto de evaluación [27] en ambos pipelines, aunque con mecanismos distintos según la métrica y la naturaleza de cada pipeline. En el baseline Python, el warm-up consiste en descartar los primeros 90 frames de las estadísticas de latencia (por etapa y end-to-end) y de throughput, dentro de cada una de las 10 repeticiones, limpiando el *allocator* de PyTorch (`torch.cuda.empty_cache()`) entre repeticiones. En el pipeline DeepStream, la medición de latencia aplica el mismo criterio de descarte explícito de 90 frames mediante contadores independientes por etapa en las pad probes, ya que la instrumentación permite un descarte frame a frame idéntico al del baseline Python. La medición de throughput nativo, en cambio, al no contar con instrumentación por frame, emplea el método de truncated replications descrito en la Sección 3.5.2 para remover el efecto del arranque sin necesidad de descarte individual. La medición de memoria no aplica ningún mecanismo de warm-up ni descarte, ya que se ejecuta sobre el pipeline completo (900 frames) y su objetivo es caracterizar el consumo total durante una ejecución representativa, incluyendo su fase de inicialización.

RESULTADOS

Este capítulo presenta los resultados experimentales obtenidos al evaluar el pipeline baseline Python y el pipeline DeepStream optimizado según el protocolo definido en la Sección 3.5 sobre el entorno de operación descrito en la Sección 3.1.2. El capítulo se organiza en seis partes: primero se presentan los resultados de la evaluación de engines TensorRT que determinan la configuración de precisión adoptada en el pipeline DeepStream; a continuación se presenta la validación cualitativa del sistema mediante las salidas visuales del pipeline sobre las tres condiciones de llama evaluadas; luego se comparan las métricas de latencia (por etapa y end-to-end), throughput y uso de memoria entre ambos pipelines; y finalmente se analizan de forma integrada los resultados principales.

Como quedó establecido en la Sección 3.2.1, el baseline cuenta con una implementación equivalente a `udpsink`, cuyo resultado no se reporta aquí dado que ya el obtenido en `fakesink` basta para descartar la viabilidad del baseline frente a DeepStream; y aunque existe un modo de escritura a disco mediante `visualize`, este está orientado a la generación de figuras de documentación mediante Matplotlib y no a la evaluación de rendimiento, por lo que se excluye de la comparación.

4.1. Evaluación de los engines TensorRT

Esta sección presenta los resultados de la evaluación descrita en la Sección 3.3.1, en términos de propiedades estructurales de los modelos, precisión de las predicciones y rendimiento computacional.

4.1.1. Propiedades estructurales de los modelos

La Tabla 4.1 resume las propiedades estructurales de los modelos base y engines compilados. La optimización TensorRT reduce sustancialmente el número de capas de 91 a 24–30 en U-Net y de 114 a 46–62 en Attention U-Net, mientras que el número de parámetros permanece prácticamente inalterado. El tamaño en disco se reduce de forma proporcional a la precisión numérica adoptada.

4.1.2. Precisión de las predicciones

La Tabla 4.2 presenta los resultados de Regression Accuracy. Las cuantizaciones FP32 y FP16 preservan la precisión del modelo base en ambas arquitecturas, con variaciones del orden de decenas de Kelvin comparables a las del propio modelo base respecto del ground truth experimental. Las diferencias entre FP32 y FP16 son numéricamente despreciables en todos los casos. La cuantización INT8 muestra comportamiento más variable: aunque en algunos casos el RMSE es comparable al de FP32 y FP16, la exclusión de diferencias superiores a ± 100 K puede subestimar errores localizados en bordes de llama, por lo que un RMSE global reducido en INT8 no garantiza preservación adecuada de la fidelidad térmica en todo el dominio espacial.

Tabla 4.1: Propiedades de los modelos U-Net y Attention U-Net según nivel de cuantización.

Modelo	Capas	Parámetros	Tamaño (MB)
U-Net Base	91	1 884 631	7.25
U-Net TRT FP32	24	1 880 040	7.56
U-Net TRT FP16	25	1 880 040	4.36
U-Net TRT INT8	30	1 880 040	2.34
Att. U-Net Base	114	8 828 070	33.75
Att. U-Net TRT FP32	62	8 823 221	34.62
Att. U-Net TRT FP16	46	8 823 221	18.12
Att. U-Net TRT INT8	47	8 823 221	10.70

Tabla 4.2: Resultados de Regression Accuracy (RMSE en K) bajo distintas condiciones de llama y niveles de cuantización.

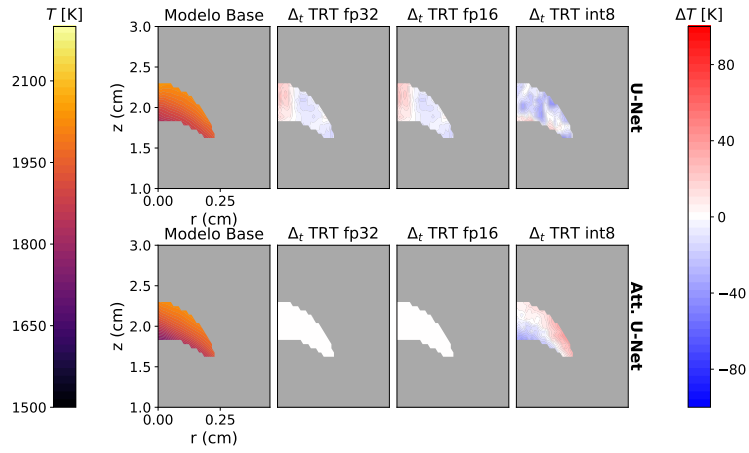
Modelo	Cond. llama	Base	TRT FP32	TRT FP16	TRT INT8
U-Net	Yale-32	15.30	19.44	19.46	20.21
	Yale-40	40.76	39.84	39.61	26.81
	Yale-60	31.62	35.40	35.39	34.02
Att. U-Net	Yale-32	31.00	31.02	31.00	35.86
	Yale-40	38.38	38.40	38.39	36.12
	Yale-60	28.42	28.43	28.43	29.07

La Figura 4.1 presenta los campos de temperatura predichos por el modelo base y los mapas de diferencia absoluta $\Delta T = T_{s,\text{opt}} - T_{s,\text{base}}$ para ambas arquitecturas y todos los niveles de cuantización. Los modelos optimizados en FP32 y FP16 mantienen desviaciones típicamente inferiores a ± 30 K. En Attention U-Net, las diferencias entre FP32 y FP16 son prácticamente imperceptibles a escala visual. La cuantización INT8 introduce degradaciones espaciales marcadas: en U-Net múltiples regiones superan diferencias de decenas de Kelvin, y en Attention U-Net se observan áreas extendidas con errores cercanos o superiores a ± 100 K.

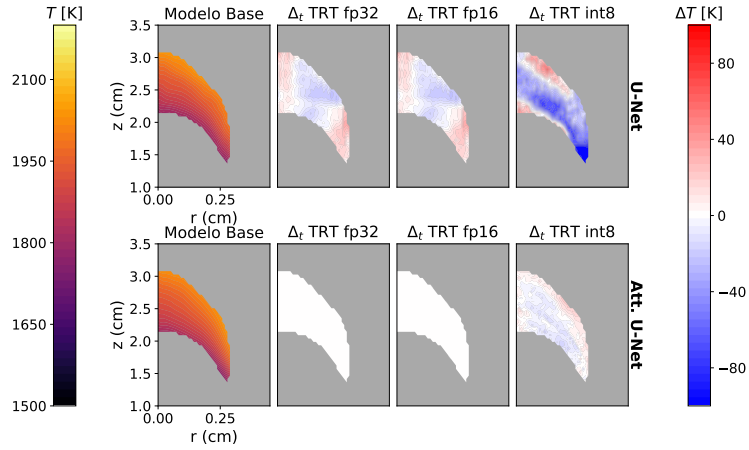
La Tabla 4.3 presenta los resultados de Regression Closeness. Attention U-Net en FP16 mantiene cercanía prácticamente perfecta con el modelo base incluso bajo tolerancias estrictas. U-Net preserva niveles elevados de coincidencia para tolerancias moderadas, lo que es aceptable desde un punto de vista físico. La cuantización INT8 reduce significativamente la cercanía en ambas arquitecturas, particularmente bajo tolerancias bajas, siendo insuficiente para una tarea de regresión térmica continua.

4.1.3. Rendimiento computacional de los engines

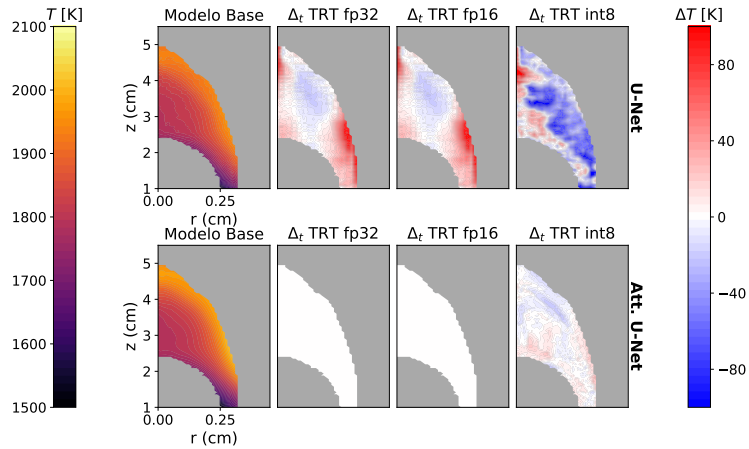
La Tabla 4.4 presenta los resultados de latencia con batch size 1, aislando exclusivamente el costo de la etapa de inferencia. TensorRT FP16 reduce la latencia mediana en un factor de aproximadamente **12.6**× en U-Net (de 6.53 ms a 0.52 ms) y **7.9**× en Att. U-Net (de 8.99 ms a 1.14 ms) respecto del modelo base PyTorch. INT8 introduce una reducción adicional marginal sobre FP16 (0.47 ms en U-Net, 1.01 ms en Att. U-Net), consistente con la expectativa teórica de que la ganancia de rendimiento de INT8 sobre FP16 es menor que la de FP16 sobre FP32, dado que ambas explotan aceleración por Tensor Cores y la diferencia se limita al ancho de los operandos. La dispersión entre mediana y máximo es reducida en todas las configuraciones TensorRT (variaciones inferiores al 30 % del valor mediano), reflejando la estabilidad temporal propia de kernels fijados durante la fase de compilación del engine; el modelo base PyTorch exhibe una dispersión mayor en términos relativos,



(a) Llama experimental de tipo Yale-32



(b) Llama experimental de tipo Yale-40



(c) Llama experimental de tipo Yale-60

Fig. 4.1: Campos de temperatura predichos por el modelo base y mapas de diferencia ΔT para distintas precisiones numéricas. Todas las subfiguras comparten la misma escala de temperatura y diferencia.

Tabla 4.3: Resultados de Regression Closeness (%) para distintos valores de tolerancia absoluta (*atol*).

Modelo	Precisión	<i>atol</i>					
		0.005	0.01	0.1	0.2	0.5	1
U-Net	FP32	11.50 %	23.15 %	85.73 %	95.73 %	99.83 %	100.00 %
	FP16	11.48 %	23.08 %	85.76 %	95.73 %	99.83 %	100.00 %
	INT8	3.82 %	7.57 %	24.92 %	30.24 %	82.12 %	93.39 %
Att. U-Net	FP32	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %	100.00 %
	FP16	99.79 %	99.99 %	100.00 %	100.00 %	100.00 %	100.00 %
	INT8	5.16 %	10.23 %	63.23 %	73.14 %	87.61 %	99.52 %

Tabla 4.4: Latencia de inferencia (ms), aislada del preprocesamiento y la transferencia CPU-GPU, para los modelos base y engines TRT.

Modelo	Configuración	Mediana	p95	Máx.
U-Net	PyTorch Base	6.527	6.685	7.488
	TRT FP32	0.663	0.692	0.855
	TRT FP16	0.516	0.542	0.682
	TRT INT8	0.469	0.507	0.617
Att. U-Net	PyTorch Base	8.993	9.153	10.314
	TRT FP32	1.875	1.905	2.020
	TRT FP16	1.139	1.167	1.320
	TRT INT8	1.012	1.098	1.114

aunque acotada en magnitud absoluta al aislar únicamente el cómputo de inferencia.

La Tabla 4.5 presenta el throughput derivado para cada configuración. TensorRT FP16 eleva el throughput de inferencia de 153.21 FPS a 1937.98 FPS en U-Net (**12.6×**) y de 111.20 FPS a 877.96 FPS en Att. U-Net (**7.9×**), coherente con las reducciones de latencia mediana reportadas en la Tabla 4.4. INT8 introduce una mejora adicional moderada sobre FP16 (2132.20 FPS frente a 1937.98 FPS en U-Net; 988.14 FPS frente a 877.96 FPS en Att. U-Net), insuficiente para justificar su adopción dada la degradación de fidelidad numérica ya establecida en las Tablas 4.2 y 4.3. En ambas arquitecturas, la totalidad de las configuraciones TensorRT supera ampliamente el desempeño de los modelos base.

4.1.4. Selección de precisión para DeepStream

Los resultados de este bloque permiten establecer con claridad la configuración a adoptar en el pipeline DeepStream. La cuantización INT8 queda descartada en ambas arquitecturas por introducir degradaciones espaciales incompatibles con la naturaleza del problema: una tarea de regresión térmica continua donde el error tiene interpretación física directa en Kelvin. Las diferencias entre FP32 y FP16 son numéricamente despreciables tanto en Regression Accuracy como en Regression Closeness, mientras que FP16 ofrece ventajas concretas: menor huella de memoria, acceso a rutas de cómputo aceleradas mediante Tensor Cores, y tiempos de inferencia equivalentes o inferiores a FP32.

En consecuencia, TensorRT FP16 se adopta como la única configuración de engine utilizada en el pipeline DeepStream, sin necesidad de evaluar otras precisiones en ese contexto. Esta decisión está respaldada por evidencia cuantitativa de preservación numérica, por consideraciones de eficiencia de memoria relevantes en la plataforma Jetson, y por la reducción sustancial de latencia y el consecuente

Tabla 4.5: Throughput de inferencia (FPS), derivado de la latencia mediana de la Tabla 4.4.

Modelo	Configuración	Throughput (FPS)
U-Net	PyTorch Base	153.21
	TRT FP32	1508.30
	TRT FP16	1937.98
	TRT INT8	2132.20
Att. U-Net	PyTorch Base	111.20
	TRT FP32	533.33
	TRT FP16	877.96
	TRT INT8	988.14

aumento de throughput respecto al modelo base PyTorch, que habilitan la operación en tiempo cercano al real sobre flujos continuos de video.

4.2. Validación cualitativa del pipeline

La Figura 4.2 presenta la salida visual del pipeline DeepStream para las tres condiciones de llama evaluadas: Yale-32, Yale-40 y Yale-60. Cada subfigura muestra el panel compuesto generado por el postprocesamiento: el mapa de temperatura T_s , los tres canales del tensor de entrada (R, G, B), y el perfil de temperatura centerline sobre el eje radial central de la llama.

Las tres condiciones exhiben la estructura espacial esperada para llamas CLAD: una distribución de temperatura con gradientes suaves en la región central, con valores más elevados en la zona de mayor concentración de hollín y descenso progresivo hacia los bordes. La condición Yale-60, con mayor caudal de combustible, presenta una llama más alta y una región de hollín más extensa, consistente con lo reportado en [5, 1]. El perfil centerline refleja en los tres casos la estratificación axial de temperatura característica de las llamas de difusión laminar.

Estos resultados confirman que el pipeline opera correctamente sobre las tres condiciones sin requerir ningún ajuste de configuración entre ellas. La misma cadena de preprocesamiento, engine TRT FP16 y postprocesamiento produce salidas físicamente coherentes para el rango completo de condiciones evaluadas, validando la generalización del modelo y la robustez del pipeline como sistema integrado.

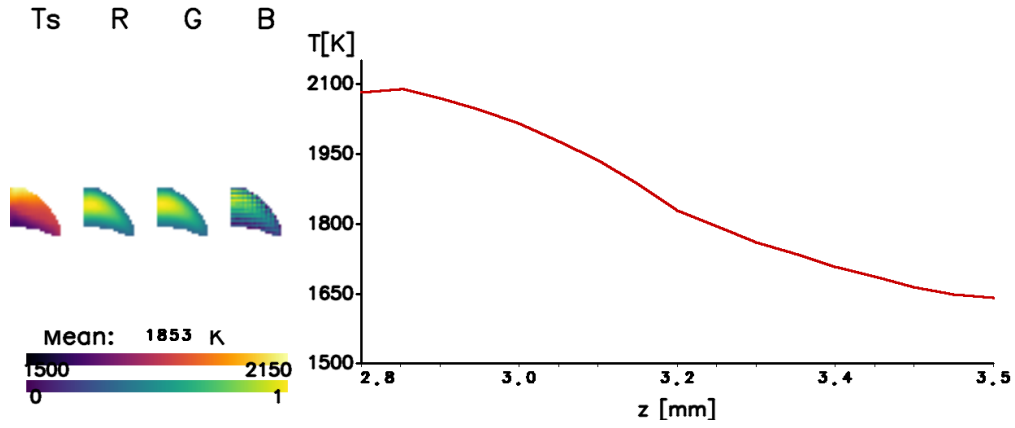
4.3. Latencia

Esta sección presenta la latencia del sistema en dos niveles de granularidad. Primero se descompone la latencia por etapa, lo que permite identificar el peso relativo de cada componente del pipeline. A partir de esa descomposición, se presenta luego la latencia end-to-end, cuya lectura resulta más clara una vez que se ha establecido cuánto pesa cada etapa individual.

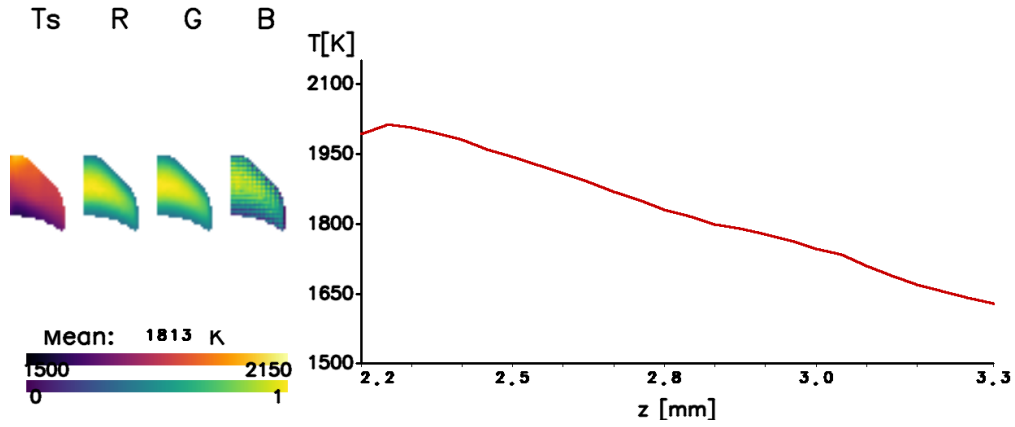
4.3.1. Latencia por etapa

La descomposición por etapa, presentada en la Tabla 4.6, revela un patrón contrastante entre ambos pipelines. En el baseline Python, la inferencia es la etapa dominante (35–43% del tiempo total según el modelo), seguida de la decodificación (7.0 ms) y el postprocesamiento (4.4 ms), con el preprocesamiento como la etapa más liviana (1.1 ms).

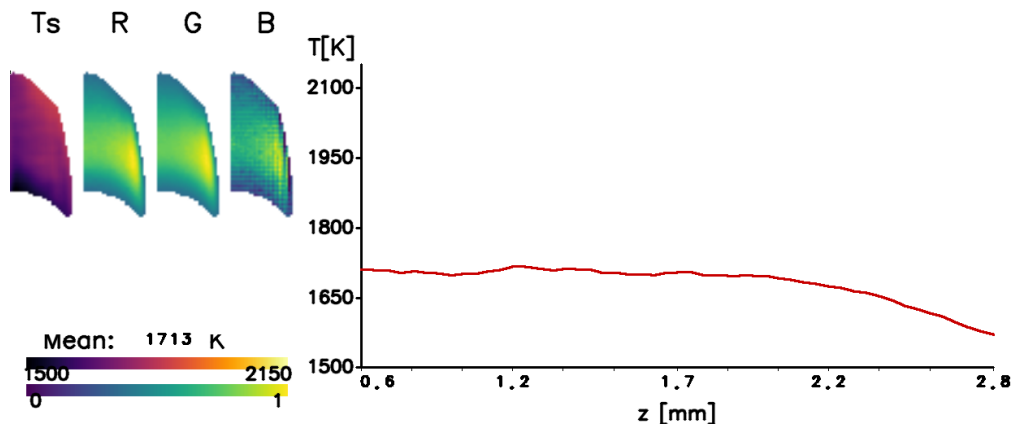
En el pipeline DeepStream, la decodificación es la etapa individual más costosa (39.3–49.0 ms),



(a) Condición Yale-32



(b) Condición Yale-40



(c) Condición Yale-60

Fig. 4.2: Salida visual del pipeline DeepStream para las tres condiciones de llama evaluadas. Las tres condiciones se procesaron con la misma configuración del pipeline, sin ajustes específicos por condición de llama, lo que refleja la capacidad de generalización del modelo [5].

Tabla 4.6: Latencia por etapa (ms) para ambos pipelines y arquitecturas. Para DeepStream, la columna Implementación representa el costo de codificación de video y transmisión/escritura no capturado por sonda individual para **filesink**/**udpsink**.

Pipeline	Sink	Modelo	Decodif.	Preproc.	Inferencia	Postproc.	Implementación
Baseline Python	—	U-Net	7.03 (± 0.03)	1.14 (± 0.01)	6.97 (± 0.03)	4.37 (± 0.01)	—
		Att. U-Net	7.02 (± 0.02)	1.13 (± 0.02)	9.50 (± 0.02)	4.36 (± 0.01)	—
DeepStream	fakesink	U-Net	39.30 (± 0.02)	0.40 (± 0.00)	0.45 (± 0.00)	2.69 (± 0.00)	—
		Att. U-Net	39.30 (± 0.02)	0.40 (± 0.01)	1.07 (± 0.01)	2.69 (± 0.01)	—
	filesink	U-Net	42.93 (± 0.31)	0.40 (± 0.01)	0.45 (± 0.01)	2.72 (± 0.01)	12.84
		Att. U-Net	48.64 (± 0.23)	0.41 (± 0.01)	1.07 (± 0.01)	2.72 (± 0.01)	16.54
	udpsink	U-Net	43.21 (± 0.30)	0.41 (± 0.01)	0.46 (± 0.01)	2.74 (± 0.01)	13.03
		Att. U-Net	48.98 (± 0.50)	0.41 (± 0.01)	1.08 (± 0.02)	2.73 (± 0.02)	16.42

muy por encima del postprocesamiento (2.7 ms), la inferencia TRT FP16 (0.45–1.08 ms) y el preprocesamiento (0.40–0.41 ms). Esta magnitud, aproximadamente el doble de la decodificación del baseline, es consistente con las limitaciones de aislar por sonda el paralelismo interno de NVDEC discutidas en la Sección 3.5.3. Un fenómeno análogo se observa en el propio baseline Python: la suma de las latencias de decodificación medidas por frame es mayor que la que se derivaría de dividir el tiempo total de ejecución por el número de frames procesados en modo throughput, lo que indica que incluso una configuración simple con `cv2.VideoCapture` conserva procesos paralelos subyacentes (E/S, buffering interno del contenedor) no capturados por la medición secuencial ingenua. La latencia de decodificación aislada por sonda, en ambos pipelines, debe interpretarse entonces como una cota superior conservadora del costo real en régimen de streaming continuo, no como su valor exacto.

La comparación de las etapas de cómputo puro en términos de preprocesamiento, inferencia y postprocesamiento confirma la ventaja de la aceleración GPU-resident: el preprocesamiento se reduce en aproximadamente 65 % (de 1.13–1.14 ms a 0.40–0.41 ms), la inferencia se reduce entre 89 % (Att. U-Net) y 94 % (U-Net), y el postprocesamiento se reduce en aproximadamente 38 % (de 4.36–4.37 ms a 2.69–2.74 ms). La brecha entre modelos en la etapa de inferencia (0.45 ms para U-Net frente a 1.07 ms para Att. U-Net) refleja directamente la diferencia de complejidad arquitectónica entre ambos modelos, y es marginal en el contexto del pipeline completo.

La comparación entre modalidades de sink dentro de DeepStream aísla el costo de codificación y transmisión: la etapa de Implementación (ausente en **fakesink**) introduce entre 12.8 y 16.5 ms adicionales para **filesink**/**udpsink**, correspondientes al encoder de hardware (`nv412h264enc`), el muxeo del contenedor y, en el caso de **udpsink**, la transmisión por red. Esta etapa adicional se acompaña de un incremento correlacionado en la decodificación medida (de 39.3 ms en **fakesink** a 42.9–49.0 ms en **filesink**/**udpsink**), consistente con el mismo fenómeno de backpressure descrito en el párrafo anterior, ahora amplificado por la presión adicional que el encoder introduce sobre `nvstreammux`.

4.3.2. Latencia end-to-end

La Tabla 4.7 presenta la latencia end-to-end para ambas arquitecturas y ambos pipelines. Para el pipeline DeepStream, la latencia se reporta con `disable-dpb=true` como protocolo estándar en las tres implementaciones de sink (**fakesink**, **filesink** y **udpsink**), según lo justificado en el capítulo de Metodología.

La latencia end-to-end mediana de DeepStream (42.9–43.5 ms en **fakesink**) resulta aproximadamente el doble que la del baseline Python (19.7–22.2 ms). Esta diferencia es enteramente atribuible a la etapa de decodificación (Tabla 4.6), y no a las etapas de cómputo, todas ellas más rápidas en

Tabla 4.7: Latencia end-to-end (ms) para ambos pipelines y arquitecturas.

Pipeline	Sink	Modelo	Mediana	p95	Máx.
Baseline Python	—	U-Net	19.66 (± 0.09)	20.08 (± 0.29)	42.81 (± 1.77)
		Att. U-Net	22.16 (± 0.04)	22.43 (± 0.03)	44.21 (± 0.97)
DeepStream	fakesink	U-Net	42.93 (± 0.04)	43.42 (± 0.03)	43.72 (± 0.11)
		Att. U-Net	43.53 (± 0.03)	44.05 (± 0.02)	44.29 (± 0.13)
	filesink	U-Net	59.34 (± 0.45)	60.75 (± 2.06)	61.87 (± 2.64)
		Att. U-Net	69.38 (± 0.31)	70.64 (± 1.63)	72.06 (± 1.98)
	udpsink	U-Net	59.85 (± 0.47)	61.08 (± 0.82)	62.07 (± 1.07)
		Att. U-Net	69.62 (± 0.51)	71.35 (± 1.77)	72.32 (± 2.08)

DeepStream. Este resultado no debe interpretarse como que el pipeline DeepStream es más lento en operación real: el costo de decodificación aquí medido corresponde a un frame aislado, mientras que en la operación real (un flujo continuo de frames) ese mismo costo se distribuye y se solapa entre los múltiples frames que el decodificador procesa en paralelo, tal como lo captura el throughput medido en régimen estacionario en la sección siguiente. La latencia aislada por frame es, en este sentido, una cota superior conservadora del costo marginal real por frame en streaming continuo, y no una medida representativa de la capacidad de respuesta del sistema en su escenario de despliegue objetivo.

Un resultado que sí es representativo con independencia del protocolo de aislamiento es la distribución de latencias dentro de cada pipeline. El baseline Python presenta una latencia máxima muy superior a su mediana (42.8 ms en U-Net y 44.2 ms en Att. U-Net frente a medianas de 19.7 y 22.2 ms), reflejando picos esporádicos asociados a la variabilidad de la decodificación OpenCV, el recolector de basura de Python y sincronizaciones CPU-GPU no determinísticas. El pipeline DeepStream, en cambio, exhibe una dispersión mucho más acotada entre mediana y máximo en **fakesink** (43.7–44.3 ms), consistente con su arquitectura de buffers pre-allocados y ausencia de sincronizaciones ad hoc; esta dispersión se amplía en **filesink**/**udpsink** (hasta 2.6 ms de rango intercuartílico), reflejando la variabilidad introducida por la escritura a disco y la transmisión de red.

4.4. Throughput

La Tabla 4.8 presenta los resultados de throughput para ambas arquitecturas y pipelines. Para DeepStream se incluyen, además de **fakesink**, las implementaciones **filesink** y **udpsink** como referencia. A diferencia de la latencia, el throughput se mide con el decoder en su configuración nativa (DPB activo), ya que caracteriza el comportamiento del sistema en estado estacionario y no el costo de un frame individual (Sección 3.5.3).

El pipeline DeepStream (**fakesink**) alcanza un throughput de estado estacionario de 306.68–306.96 fps para ambas arquitecturas, frente a 52.12 fps (U-Net) y 46.73 fps (Att. U-Net) del baseline Python, lo que representa factores de mejora de **5.9×** y **6.6×** respectivamente. La brecha de throughput entre U-Net y Att. U-Net en el baseline (10%) es considerablemente menor que la brecha observada en la latencia de inferencia aislada (36%, Tabla 4.6), lo que sugiere que en el baseline la inferencia no domina completamente el costo total del ciclo, y que otras etapas (decodificación, post-procesamiento) atenúan proporcionalmente el impacto del modelo más costoso sobre el throughput agregado. En DeepStream, la diferencia entre arquitecturas es marginal (306.96 frente a 306.68 fps), confirmando que la etapa de inferencia no es el cuello de botella del pipeline optimizado, con independencia del modelo empleado.

Las modalidades **filesink** (236.0–276.6 fps) y **udpsink** (233.4–274.2 fps) muestran una reducción

Tabla 4.8: Throughput (fps) para ambos pipelines y arquitecturas.

Pipeline	Sink	Modelo	FPS estado estacionario	FPS ingenuo	Diferencia
Baseline Python	—	U-Net	52.12 (± 0.30)	—	—
		Att. U-Net	46.73 (± 0.11)	—	—
DeepStream	fakesink	U-Net	306.96 (± 1.10)	304.16	−0,91 %
		Att. U-Net	306.68 (± 0.61)	301.36	−1,73 %
	filesink	U-Net	276.56 (± 1.62)	273.74	−1,02 %
		Att. U-Net	236.01 (± 1.80)	233.50	−1,06 %
	udpsink	U-Net	274.24 (± 2.66)	271.62	−0,96 %
		Att. U-Net	233.42 (± 1.76)	231.22	−0,94 %

Tabla 4.9: Uso de memoria neta del sistema (MB), calculada como el consumo menos la RAM idle del entorno de ejecución.

Pipeline	Sink	Modelo	RAM máx. neta	RAM prom. neta
Baseline Python	—	U-Net	11 923.70 (± 15.13)	5 005.96 (± 14.45)
		Att. U-Net	11 919.60 (± 190.15)	5 554.81 (± 21.45)
DeepStream	fakesink	U-Net	972.67 (± 28.94)	397.49 (± 119.25)
		Att. U-Net	284.84 (± 3.55)	213.16 (± 2.75)
	filesink	U-Net	972.90 (± 24.42)	420.13 (± 160.86)
		Att. U-Net	362.04 (± 11.98)	261.89 (± 8.16)
	udpsink	U-Net	999.80 (± 21.34)	1 005.33 (± 199.74)
		Att. U-Net	424.34 (± 9.55)	298.36 (± 8.17)

moderada respecto de **fakesink**, consistente con el costo adicional de codificación y transmisión ya cuantificado en la Tabla 4.6; ambas modalidades continúan superando ampliamente el throughput del baseline Python.

El análisis por truncated replications evidencia, en todas las combinaciones de modelo y sink, un costo de arranque estadísticamente significativo: la diferencia entre FPS ingenuo y FPS de estado estacionario oscila entre 0.91 % y 1.73 %, superando en todos los casos el margen de incertidumbre derivado de la propagación del error estándar de las mediciones (0.07–0.30 %). Esta diferencia refleja el costo de inicialización del contexto TRT y las colas de GStreamer, que se amortiza a lo largo de la ejecución y justifica el uso del método de corrección adoptado.

4.5. Uso de memoria

La Tabla 4.9 presenta los resultados de uso de memoria del sistema para ambas arquitecturas y pipelines, medidos con el decoder en su configuración nativa (DPB activo), consistente con la medición de throughput. Se reporta únicamente la RAM neta (consumo menos idle), que es la magnitud comparable entre pipelines, ya que la RAM total observada depende del RAM idle del entorno de ejecución al momento de la medición y no del pipeline en sí.

En términos de consumo promedio neto, el pipeline DeepStream (**fakesink**) reduce drásticamente el uso de memoria respecto del baseline Python: de 5 005.96–5 554.81 MB a 213.16–397.49 MB, una reducción superior al 92 % en ambos modelos. Esta diferencia de orden de magnitud es consistente con el comportamiento del asignador de memoria *caching allocator* de PyTorch sobre la arquitectura de memoria unificada de Jetson, que reserva bloques sustancialmente mayores a los estrictamente requeridos por el modelo, frente a la gestión de buffers pre-alocados y de tamaño fijo del pipeline

Tabla 4.10: Síntesis de resultados comparativos entre el baseline Python y el pipeline DeepStream, descartando los frames tras su procesamiento en modalidad **fakesink** o equivalente. La latencia de DeepStream se mide con **disable-dpb=true**; el throughput y la memoria, con el decodificador en su configuración nativa con DPB activo.

Métrica	Baseline Python		DeepStream	
	U-Net	Att. U-Net	U-Net	Att. U-Net
Latencia mediana (ms)	19.66	22.16	42.93	43.53
Latencia p95 (ms)	20.08	22.43	43.42	44.05
Latencia máxima (ms)	42.81	44.21	43.72	44.29
Throughput (fps)	52.12	46.73	306.96	306.68
Mejora throughput	—	—	5.9×	6.6×
RAM neta prom. (MB)	5 005.96	5 554.81	397.49	213.16
RAM neta máx. (MB)	11 923.70	11 919.60	972.67	284.84

DeepStream.

Dentro de DeepStream, sin embargo, las mediciones de memoria exhiben una inestabilidad notable que no se observa en las métricas de latencia ni throughput. La RAM neta máxima de U-Net (972.67–999.80 MB según sink) resulta considerablemente mayor y más dispersa (desviaciones de hasta ± 199.74 MB) que la de Att. U-Net (284.84–424.34 MB, desviaciones de un orden de magnitud menor), una asimetría que se reproduce de forma consistente en repeticiones adicionales del experimento y que resulta contraintuitiva dado que Att. U-Net es el modelo de mayor complejidad y, en el baseline Python, ambos modelos presentan un consumo de memoria prácticamente idéntico. Esta inestabilidad no afecta las conclusiones de rendimiento del sistema, pues la latencia y el throughput de ambos modelos en DeepStream son consistentes y reproducibles entre repeticiones, pero constituye un fenómeno específico de la medición de memoria que se documenta aquí como observación abierta, sin que los datos disponibles permitan atribuirlo de forma concluyente a una causa particular dentro del alcance de este trabajo.

4.6. Análisis de resultados

La Tabla 4.10 resume, para el escenario **fakesink**, los resultados principales de la comparación entre el baseline Python y el pipeline DeepStream, organizados en tres bloques: latencia, throughput y memoria.

Los resultados presentan un panorama más matizado que una superioridad uniforme de DeepStream sobre el baseline Python, y su correcta interpretación exige distinguir entre las condiciones de medición de cada métrica.

En throughput, la métrica más representativa del régimen de operación continua para el cual el sistema fue diseñado, DeepStream supera al baseline de forma contundente (5.9–6.6×) con un uso de memoria neta muy inferior (reducción superior al 92%). Ambos resultados se miden con el decodificador en su configuración nativa (DPB activo), reflejando el comportamiento sostenido bajo carga continua que constituye el escenario de despliegue objetivo de este trabajo.

En latencia end-to-end aislada por frame, en cambio, DeepStream exhibe una latencia mediana aproximadamente el doble que la del baseline Python, resultado explicado en su totalidad por la etapa de decodificación, siendo la única etapa en la que DeepStream no supera al baseline, mientras que las etapas de cómputo puro (preprocesamiento, inferencia y postprocesamiento) son, todas ellas, entre 38 % y 94 % más rápidas en DeepStream. Este resultado, sin embargo, no debe leerse como una

desventaja real del pipeline optimizado: la latencia de decodificación aislada por sonda sobreestima el costo marginal de un frame en operación continua, ya que en un flujo real ese tiempo se distribuye y se solapa entre los múltiples frames que el decodificador procesa en paralelo, que es precisamente el régimen que capta la métrica de throughput. Que el fenómeno también pudiera identificarse en el propio baseline Python confirma que se trata de una limitación general de medir por sonda individual una etapa de comportamiento inherentemente paralelo, y no una particularidad ni una debilidad del pipeline optimizado.

La magnitud de esta ganancia en las etapas de cómputo puro es consistente con la elección de FP16 como precisión de despliegue: al preservar la fidelidad numérica del modelo base sin las penalizaciones de fidelidad observadas en INT8, FP16 permite capturar la aceleración de Tensor Cores sin comprometer la validez física de la estimación de temperatura, explicando por qué la etapa de inferencia, dominante en el baseline Python, se vuelve marginal frente a la decodificación en el pipeline optimizado.

La factibilidad de sostener, además de `fakesink`, las modalidades `filesink` y `udpsink` con throughputs de 233–277 fps (sin contraparte viable en términos de rendimiento) refuerza que DeepStream constituye la única de las dos arquitecturas evaluadas que resulta viable para escenarios de despliegue real con almacenamiento persistente o transmisión en tiempo real, más allá de la comparación estricta de throughput en modalidad de referencia.

CONCLUSIONES

Este trabajo abordó el diseño, implementación y evaluación comparativa de un pipeline de análisis óptico de llamas en tiempo real sobre plataforma GPU embebida, tomando como caso de aplicación la estimación de temperatura de hollín en llamas laminares a partir de imágenes RGB mediante arquitecturas CNN. El enfoque adoptado sitúa la optimización del pipeline completo como el problema central, y propone una comparación estructurada entre un pipeline de referencia en Python con ejecución predominantemente en CPU y un pipeline optimizado construido sobre NVIDIA DeepStream con ejecución íntegramente acelerada en GPU sobre la plataforma Jetson AGX Orin.

La construcción del pipeline baseline a partir de los trabajos previos requirió una revisión metodológica significativa orientada a producir un sistema estable, compartimentado y representativo de una implementación Python de propósito general para análisis de flujos de video.

El análisis de alternativas de paralelización inicial durante la construcción del baseline secuencial mostró que la decodificación puede beneficiarse del desacoplamiento, pero que la complejidad de sincronización introducida no es coherente con el objetivo de establecer una base reproducible. Esta conclusión conecta directamente con la motivación del pipeline DeepStream: la decodificación acelerada en hardware y el paralelismo integral de las etapas son precisamente los problemas que DeepStream resuelve de forma nativa y robusta.

La réplica de los experimentos de optimización de inferencia de Aguilera sobre el caso de pirometría de hollín permitió establecer con evidencia cuantitativa la configuración de engine TensorRT más adecuada para su integración en el pipeline DeepStream.

Los resultados de Regression Accuracy muestran que las cuantizaciones FP32 y FP16 preservan la precisión del modelo base en ambas arquitecturas evaluadas, con diferencias entre sí que son numéricamente despreciables en todas las condiciones de llama. La cuantización INT8 introduce degradaciones espaciales incompatibles con la naturaleza del problema: al tratarse de una tarea de regresión continua donde el error tiene interpretación física directa en Kelvin, la pérdida de fidelidad numérica no se traduce únicamente en mayor error numérico sino en predicciones físicamente no válidas, especialmente en las regiones de mayor gradiente térmico. Los resultados de Regression Closeness refuerzan esta conclusión, evidenciando que INT8 reduce significativamente la fracción de píxeles dentro de tolerancias aceptables incluso para umbrales moderados.

La comparación entre FP32 y FP16 muestra que ambas configuraciones son prácticamente equivalentes en términos de fidelidad numérica, mientras que FP16 ofrece ventajas concretas en huella de memoria y acceso a rutas de cómputo aceleradas mediante Tensor Cores.

En términos de inferencia aislada, TensorRT FP16 reduce la latencia mediana en un factor de 12.6× en U-Net (de 6.53 ms a 0.52 ms) y de 7.9× en Attention U-Net (de 8.99 ms a 1.14 ms) respecto al modelo base PyTorch, con mejoras de throughput consistentes (153.21 fps a 1937.98 fps y 111.20 fps a 877.96 fps, respectivamente). INT8 introduce una reducción adicional marginal sobre

FP16 en ambas métricas, insuficiente para justificar su adopción dada la degradación de fidelidad numérica ya establecida.

La arquitectura del modelo condiciona además la estabilidad numérica alcanzable tras la optimización. Attention U-Net demostró mayor tolerancia a las transformaciones del grafo computacional, manteniendo equivalencia práctica con el modelo original incluso bajo tolerancias estrictas en FP16, mientras que U-Net clásico exhibió mayor sensibilidad en regiones de alta complejidad térmica. Esta diferencia sugiere que los mecanismos de atención introducen propiedades estructurales que favorecen la robustez frente a la fusión y replanificación de capas durante la building phase, a costa de ciertos retrasos en latencia.

En consecuencia, TensorRT FP16 fue adoptado como la configuración de engine para el pipeline DeepStream, una decisión respaldada por evidencia cuantitativa de preservación numérica, eficiencia de memoria y reducción sustancial de latencia.

La implementación del pipeline sobre NVIDIA DeepStream demostró que es posible construir un sistema de análisis óptico científico en tiempo real sobre plataforma embebida a partir de un framework diseñado originalmente para visión por computador convencional, siempre que se comprendan sus limitaciones estructurales y se aproveche adecuadamente su arquitectura de extensión mediante bibliotecas C++ compiladas.

Los resultados de throughput muestran que el pipeline DeepStream alcanza 306.68–306.96 fps en modo `fakesink` con `gst-launch-1.0` nativo, frente a 46.73–52.12 fps del baseline Python, representando una mejora de un factor de $5.9\times$ a $6.6\times$ según el modelo. La latencia end-to-end mediana del pipeline DeepStream, medida bajo `disable-dpb`, es de 42.93–43.53 ms, de la cual la etapa de decodificación representa la fracción dominante (~39.3 ms de mediana); este valor constituye una cota superior conservadora del costo real en streaming continuo, y no el costo marginal por frame, dado que el paralelismo interno del decodificador NVDEC y el backpressure del `nvstreammux` inflan la medición aislada por sonda individual. El preprocesamiento (0.40–0.41 ms), la inferencia TRT FP16 (0.45–1.08 ms) y el postprocesamiento (2.69–2.74 ms) permanecen, en cambio, en rangos que habilitan holgadamente el procesamiento en tiempo casi real. El uso de memoria neta promedio se reduce, además, en más de un 92 % respecto del baseline (de 5 005.96–5 554.81 MB a 213.16–397.49 MB), reforzando la viabilidad del pipeline DeepStream para operación sostenida sobre plataforma embebida con recursos limitados.

Estos resultados evidencian, a su vez, que la comparación de sistemas de procesamiento de video en tiempo real no admite una métrica única de velocidad, ya que el throughput en régimen estacionario y la latencia aislada por frame responden a preguntas distintas; el primero refleja la capacidad sostenida de procesamiento, mientras que la segunda representa el costo computacional puro de cada etapa del algoritmo. Por ello, la elección de la métrica debe ajustarse al objetivo de la medición, sin que ambas puedan mezclarse en una conclusión única sin pérdida de precisión. Bajo la métrica más representativa del escenario de despliegue objetivo de este trabajo, la operación sostenida en régimen de streaming continuo, DeepStream supera de forma contundente al baseline Python tanto en capacidad de procesamiento como en eficiencia de memoria.

Desde el punto de vista de la ingeniería de sistemas, el desarrollo del pipeline DeepStream produjo varias contribuciones metodológicas relevantes. Entre ellas, la identificación y corrección del error de stride/pitch en el preprocesamiento, donde la función interna `prepare_tensor` asumía un stride de $W \times 4$ bytes cuando el hardware Jetson asigna un pitch de 256 bytes para superficies de ancho 32, ilustra que las implementaciones genéricas de DeepStream no contemplan las particularidades del hardware embebido en escenarios fuera del caso de uso convencional. La solución, un kernel CUDA propio que lee el pitch real desde `batch->pitch`, eliminó simultáneamente el error y una transformación intermedia innecesaria heredada de la arquitectura general de DeepStream, reduciendo la cadena de transformaciones del preprocesamiento.

La conclusión metodológica más relevante en este sentido es que, aunque DeepStream no está diseñado para aceleración de computación científica, su arquitectura de extensión permite adaptarlo a ese propósito. Los puntos de extensión mediante bibliotecas `.so` compiladas en C++ para `nvdspreprocess` y `nvdspostprocess` ofrecen acceso directo a los buffers GPU en el momento preciso en que los datos están disponibles en memoria del device, lo que permite implementar cualquier lógica de procesamiento sin salir del espacio de memoria GPU. Esta capacidad es la que habilita que todas las etapas del postprocesamiento se ejecuten íntegramente mediante kernels CUDA sin transferencias al host.

El trabajo desarrollado establece una base técnica sólida, pero también pone de manifiesto un conjunto de limitaciones que definen líneas de trabajo futuro. La principal de ellas corresponde a la calibración física del preprocesamiento. La simplificación del dominio de interpolación a coordenadas de píxel, adoptada para resolver la inconsistencia entre el dominio físico del pipeline original y el dominio esperado por el modelo, implica que el sistema no reconstruye actualmente las dimensiones físicas reales (en milímetros) del campo de temperatura. Para que el sistema pueda aplicarse a configuraciones experimentales distintas de la evaluada, será necesario reintroducir los parámetros de escala y desplazamiento, además de implementar un procedimiento de calibración asistida que permita estimarlos a partir de la geometría conocida del quemador y la cámara.

Un segundo pendiente relevante corresponde a la validación con ground-truth del pipeline completo operando sobre flujos de video. La comparación de fidelidad numérica reportada para los engines TensorRT sobre los modelos PyTorch base se realizó sobre el subconjunto experimental de imágenes promediadas con su correspondiente campo de temperatura de referencia, permitiendo establecer dicha equivalencia de forma rigurosa. Sin embargo, los flujos de video sobre los que operan efectivamente los pipelines evaluados fueron contruidos a partir del segundo subconjunto experimental, compuesto por imágenes individuales sin ground truth de temperatura. En consecuencia, si bien existe evidencia sólida de que el engine preserva la precisión del modelo base, no se ha validado aún que el pipeline completo (incluyendo las etapas de preprocesamiento y postprocesamiento reimplementadas en CUDA, así como los efectos de la decodificación de video sobre la señal) reproduzca temperaturas físicamente correctas en régimen de streaming continuo. Cerrar esta brecha requeriría la construcción de nuevos conjuntos de flujos de video que incorporen su respectivo ground-truth de temperatura por frame, obtenido mediante técnicas de medición ya establecidas, permitiendo así extender la validación numérica ya realizada a nivel de engine hacia una validación física a nivel de sistema completo.

Otra línea de desarrollo corresponde a la integración del pipeline dentro de un sistema de diagnóstico óptico completo. La solución propuesta constituye el núcleo computacional de una arquitectura de despliegue formada por tres componentes principales: la cámara como fuente de adquisición de datos, la plataforma Jetson como unidad de procesamiento en tiempo real y un equipo externo encargado de la interfaz de usuario y del almacenamiento de la información. La integración de estos componentes en un único software representa el siguiente paso hacia un sistema funcional capaz de operar en entornos experimentales e industriales. En este sentido, las modalidades `filesink` y `udpsink` evaluadas en el capítulo de Resultados, con throughputs sostenidos de 233–277 fps, confirman la viabilidad de dicha integración sin comprometer significativamente la capacidad de procesamiento del sistema.

Finalmente, el flujo metodológico desarrollado presenta un nivel de generalidad que permite extenderlo a otros problemas de regresión continua basados en imágenes sobre plataformas GPU embebidas. Aunque la pirometría de hollín constituye el caso de estudio abordado en esta memoria, la misma arquitectura resulta aplicable a cualquier escenario que requiera inferencia densa en tiempo real con recursos computacionales limitados, incluyendo aplicaciones de monitoreo industrial de procesos físicos continuos y análisis de variables ambientales en sistemas autónomos.

REFERENCIAS

- [1] Jorge Portilla, Juan J. Cruz, Felipe Escudero, Alonso Rodríguez, Rodrigo Demarco, Andrés Fuentes, and Gonzalo Carvajal. Towards low-cost soot pyrometry in laminar flames using broadband emission measurements and artificial neural networks. *Journal of the Energy Institute*, 109:101258, 2023.
- [2] Ahmed Abouelhamayed, Susanne Balle, Deshanand Singh, and Mohamed Abdelfattah. Beyond inference: Performance analysis of dnn server overheads for computer vision. In *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC '24*, page 1–6. ACM, 2024.
- [3] Umair Iqbal, Tim Davies, and Pascal Perez. A review of recent hardware and software advances in gpu-accelerated edge-computing single-board computers (sbc) for computer vision. *Sensors*, 24(15), 2024.
- [4] NVIDIA Corporation. NVIDIA DeepStream SDK. <https://developer.nvidia.com/deepstream-sdk>, 2024.
- [5] J. Portilla, J.J. Cruz, F. Escudero, R. Demarco, A. Fuentes, and G. Carvajal. A generalized neural network for accurate estimation of soot temperature in laminar flames using a single rgb image. *Journal of the Energy Institute*, 119:102001, 2025.
- [6] J. Portilla. *Desarrollo de instrumentación de bajo costo para pirometría de hollín usando cámaras a color y redes neuronales artificiales*. Ph.d. dissertation, Universidad Técnica Federico Santa María, Valparaíso, Chile, 2024.
- [7] Yuxiao Zhou and Kecheng Yang. Exploring TensorRT to improve real-time inference for deep learning. In *IEEE 24th Int Conf on High Performance Computing Communications; 8th Int Conf on Data Science Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud Big Data Systems Application*, pages 2011–2018, 2022.
- [8] Juan Aguilera and Gonzalo Carvajal. Experimental evaluation of neural networks optimized for gpu inference using tensorrt. In *2024 IEEE International Conference on Automation/XXVI Congress of the Chilean Association of Automatic Control (ICA-ACCA)*, pages 1–6, Oct 2024.
- [9] WinklerLaplace. deepstream-soot-pyrometry: Deepstream-based optimized inference pipeline for soot pyrometry on laminar flames. <https://github.com/WinklerLaplace/deepstream-soot-pyrometry>, 2026.
- [10] A. Rodríguez, F. Escudero, J.J. Cruz, G. Carvajal, and A. Fuentes. Retrieving soot volume fraction fields for laminar axisymmetric diffusion flames using convolutional neural networks. *Fuel*, 285:119011, 2021.

- [11] Felipe Escudero, Victor Chernov, Juan J. Cruz, Efraín Magaña, Benjamín Herrmann, and Andrés Fuentes. Robust automatic retrieval of soot volume fraction, temperature and radiation for axisymmetric flames. *Proceedings of the Combustion Institute*, 40(1):105493, 2024.
- [12] Tao Ren, Michael F. Modest, Alexander Fateev, Gavin Sutton, Weijie Zhao, and Florin Rusu. Machine learning applied to retrieval of temperature and concentration distributions from infrared emission measurements. *Applied Energy*, 252:113448, 2019.
- [13] Tao Ren, Hongxu Li, Michael F. Modest, and Changying Zhao. Machine learning applied to the retrieval of three-dimensional scalar fields of laminar flames from hyperspectral measurements. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 279:108047, 2022.
- [14] Hongxu Li, Tao Ren, Xunchen Liu, and Changying Zhao. U-net applied to retrieve two-dimensional temperature and co2 concentration fields of laminar diffusion flames. *Fuel*, 324:124447, 2022.
- [15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.
- [16] Ozan Oktay, Jo Schlemper, Loic Le Folgoc, Matthew Lee, Mattias Heinrich, Kazunari Misawa, Kensaku Mori, Steven McDonagh, Nils Y Hammerla, Bernhard Kainz, Ben Glocker, and Daniel Rueckert. Attention u-net: Learning where to look for the pancreas, 2018.
- [17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [18] NVIDIA. Cuda c programming guide. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, 2024. Accessed: October 2024.
- [19] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:21, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [20] NVIDIA Corporation. NVIDIA cuDNN Documentation. <https://docs.nvidia.com/deeplearning/cudnn/>, 2024.
- [21] NVIDIA. Nvidia tensorrt. <https://developer.nvidia.com/tensorrt>, 2024. Accessed: October 2024.
- [22] A. Paszke, S. Gross, and S. Chintala. Automatic differentiation in pytorch. <https://pytorch.org/>, 2017. Accessed: August 2024.
- [23] M. Abadi. Tensorflow: Large-scale machine learning on heterogeneous systems. <https://www.tensorflow.org/>, 2015. Accessed: August 2024.
- [24] F. Chollet. Keras. <https://github.com/fchollet/keras>, 2015. Accessed: August 2024.
- [25] ONNX Contributors. Open neural network exchange (onnx). <https://github.com/onnx/onnx>, 2024. Accessed: October 2024.
- [26] NVIDIA. Tensorrt 8.6 developer documentation. <https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-861/developer-guide/index.html>, 2023.

- [27] Yuxiao Zhou and Kecheng Yang. Exploring tensorrt to improve real-time inference for deep learning. *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 2011–2018, 2022.
- [28] Omais Shafi, Chinmay Rai, Rijurekha Sen, and Gayathri Ananthanarayanan. Demystifying tensorrt: Characterizing neural network inference engine on nvidia edge devices. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 226–237, 2021.
- [29] Gaurav Verma, Yashi Gupta, Abid M. Malik, and Barbara Chapman. Performance evaluation of deep learning compilers for edge inference. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 858–865, 2021.
- [30] Berk Ulker, Sander Stuijk, Henk Corporaal, and Rob Wijnhoven. Reviewing inference performance of state-of-the-art deep learning frameworks. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '20, page 48–53, New York, NY, USA, 2020. Association for Computing Machinery.
- [31] NVIDIA. Tensorrt IBuilderConfig api documentation. https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/infer/Core/BuilderConfig.html, 2024. Accessed: October 2024.
- [32] GStreamer Community. GStreamer Application Development Manual: Basic Concepts. <https://gstreamer.freedesktop.org/documentation/application-development/introduction/basics.html>, 2024.
- [33] NVIDIA Corporation. DeepStream SDK Developer Guide: Overview. https://docs.nvidia.com/metropolis/deepestream/dev-guide/text/DS_Overview.html, 2024.
- [34] G. Blanquart and H. Pitsch. Analyzing the effects of temperature on soot formation with a joint volume-surface-hydrogen model. *Combustion and Flame*, 156(8):1614–1626, 2009.
- [35] J. J. Cruz, Luis Fernando Figueira da Silva, F. Escudero, F. Cepeda, J. C. Elicer-Cortés, and A. Fuentes. Soot pyrometry by emission measurements at different wavelengths in laminar axisymmetric flames. *Combustion Science and Technology*, 194(8):1643–1660, 2022.
- [36] T.P Jenkins and R.K Hanson. Soot pyrometry using modulated absorption/emission. *Combustion and Flame*, 126(3):1669–1679, 2001.
- [37] Jinyu Zhu, Mun Young Choi, George W. Mulholland, Samuel L. Manzello, Louis A. Gritzo, and Jill Suo-Anttila. Measurement of visible and near-ir optical properties of soot produced from laminar flames. *Proceedings of the Combustion Institute*, 29(2):2367–2374, 2002.
- [38] Fengshan Liu, Jean-Louis Consalvi, and Fatiha Nmira. The importance of accurately modelling soot and radiation coupling in laminar and laboratory-scale turbulent diffusion flames. *Combustion and Flame*, 258:112573, 2023. Special issue and Perspective on the Chemistry and Physics of Carbonaceous Particle Formation.
- [39] A. Guibaud, J.M. Citerne, J.M. Orlac'h, O. Fujita, J.-L. Consalvi, J.L. Torero, and G. Legros. Broadband modulated absorption/emission technique to probe sooting flames: Implementation, validation, and limitations. *Proceedings of the Combustion Institute*, 37(3):3959–3966, 2019.

- [40] Juan José Cruz, Felipe Escudero, Emilio Álvarez, Luís Fernando Figueira da Silva, Gonzalo Carvajal, María Thomsen, and Andrés Fuentes. Three-wavelength broadband soot pyrometry technique for axisymmetric flames. *Opt. Lett.*, 46(11):2654–2657, Jun 2021.
- [41] Nathan J. Kempema and Marshall B. Long. Effect of soot self-absorption on color-ratio pyrometry in laminar coflow diffusion flames. *Opt. Lett.*, 43(5):1103–1106, Mar 2018.
- [42] NVIDIA Corporation. Jetson Orin Nano Series, Jetson Orin NX Series and Jetson AGX Orin Series — Platform Power and Performance, Jetson Linux Developer Guide. <https://docs.nvidia.com/jetson/archives/r36.4.4/DeveloperGuide/SD/PlatformPowerAndPerformance/JetsonOrinNanoSeriesJetsonOrinNxSeriesAndJetsonAgxOrinSeries.html>, 2024.
- [43] Nick A. Eaves, Qingan Zhang, Fengshan Liu, Hongsheng Guo, Seth B. Dworkin, and Murray J. Thomson. Coflame: A refined and validated numerical algorithm for modeling sooting laminar coflow diffusion flames. *Computer Physics Communications*, 207:464–477, 2016.
- [44] J. Gau, D. Das, C. McEnally, D. Giassi, N. Kempema, and M. Long. Yale coflow diffusion flames: Steady flame burner. http://guilford.eng.yale.edu/yalecoflowflames/steady_burner.html, 2014. Accessed: August 2024.
- [45] Jérôme Yon, Juan José Cruz, Felipe Escudero, José Morán, Fengshan Liu, and Andrés Fuentes. Revealing soot maturity based on multi-wavelength absorption/emission measurements in laminar axisymmetric coflow ethylene diffusion flames. *Combustion and Flame*, 227:147–161, 2021.
- [46] FFmpeg Team. FFmpeg. <https://ffmpeg.org>, 2024.
- [47] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979.
- [48] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.
- [49] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. Referencia canónica del proyecto: <https://opencv.org>; documentación técnica: <https://docs.opencv.org/>.
- [50] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3):261–272, 2020.
- [51] SciPy Developers. SciPy 1.10.0 Release Notes. <https://docs.scipy.org/doc/scipy/release/1.10.0-notes.html>, 2023. Sección de deprecaciones (incluye `interp2d`).
- [52] NVIDIA. Post-training quantization using calibration. <https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-861/developer-guide/index.html#enable-int8-calibration>, 2024. Accessed: August 2024.

- [53] Fumio Hamanaka, Takashi Odan, Kenji Kise, and Thiem Van Chu. An exploration of state-of-the-art automation frameworks for fpga-based dnn acceleration. *IEEE Access*, 11:5701–5713, 2023.
- [54] NVIDIA. Polygraphy. <https://docs.nvidia.com/deeplearning/tensorrt/polygraphy/docs/index.html>, 2024. Accessed: August 2024.
- [55] NVIDIA. Tensorrt engineinspector api. https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/infer/Core/EngineInspector.html, 2024. Accessed: August 2024.
- [56] G. Malivenko. onnx-opcounter. <https://github.com/gmalivenko/onnx-opcounter>, 2024. Accessed: August 2024.
- [57] Zhu Chunxiang, Qian Jiacheng, and Binrui Wang. Yolox on embedded device with cctv tensorrt for intelligent multicategories garbage identification and classification. *IEEE Sensors Journal*, 22(16):16522–16532, 2022.
- [58] PyTorch Contributors. torch.isclose. <https://pytorch.org/docs/stable/generated/torch.isclose.html>, 2024. Accessed: August 2024.
- [59] NVIDIA Corporation. Jetson Linux API Reference: Video Decoder (NvVideoDecoder). https://docs.nvidia.com/jetson/l4t-multimedia/group__l4t__mm__nvvideodecoder__group.html, 2024.
- [60] NVIDIA Corporation. DeepStream Plugin Guide: Gst-nvdspreprocess (NvBufSurfTransform). https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_plugin_gst-nvdspreprocess.html, 2024. API reference: https://docs.nvidia.com/metropolis/deepstream/8.0/sdk-api/nvbufsurftransform_8h_source.html.
- [61] FourCC.org. YUV pixel formats — NV12. <https://www.fourcc.org/pixel-format/yuv-nv12/>, 2024.
- [62] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.
- [63] Jamie R. Nuñez, Christopher R. Anderton, and Ryan S. Renslow. Optimizing colormaps with consideration for color vision deficiency to enable accurate interpretation of scientific data. *PLOS ONE*, 13(7):1–14, 08 2018.
- [64] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2nd edition, 1995.
- [65] International Telecommunication Union. Recommendation ITU-R BT.601-7: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios. Recommendation BT.601-7, ITU Radiocommunication Sector (ITU-R), 3 2011.
- [66] RidgeRun. Embedded GStreamer Performance Tuning. https://developer.ridgerun.com/wiki/index.php/Embedded_GStreamer_Performance_Tuning, 2024.
- [67] Saeid Ghafouri, Yuming Ding, Katherine Diaz Chito, Jesús Martínez del Rincón, Niamh O’Connell, and Hans Vandierendonck. Optimizing video analytics inference pipelines: A case study. In *Proceedings of the IEEE/ACM 12th International Conference on Big Data Computing, Applications and Technologies*, BDCAT ’25, New York, NY, USA, 2025. Association for Computing Machinery.

-
- [68] Averill M. Law. *Simulation Modeling and Analysis*. McGraw-Hill Education, New York, NY, 5th edition, 2015.
- [69] NVIDIA. tegrastats utility. https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvlib_docs/index.html, 2024. Accessed: August 2024.
- [70] NVIDIA Corporation. Tegrastats Utility — Jetson Linux Developer Guide. <https://docs.nvidia.com/jetson/archives/r36.4.4/DeveloperGuide/AT/JetsonLinuxDevelopmentTools/TegrastatsUtility.html>, 2024.
- [71] S. K. Prashanthi, Sai Anuroop Kesanapalli, and Yogesh Simmhan. Characterizing the Performance of Accelerated Jetson Edge Devices for Training Deep Learning Models. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 6(3), 2022.
- [72] Gary J. Sullivan. Reducing latency in video encoding and decoding. US Patent 10,805,640 B2, 2020. Assignee: Microsoft Technology Licensing, LLC. Publicada el 13 de octubre de 2020 (prioridad: 30 de junio de 2011). <https://patents.google.com/patent/US10805640B2/en>.
- [73] NVIDIA Corporation. DeepStream Plugin Guide: Gst-nvvideo4linux2. https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_plugin_gst-nvvideo4linux2.html, 2024.