MÁSTER EN TECNOLOGÍAS Y APLICACIONES EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

UNIVERSIDAD DE ALMERÍA

ESCUELA SUPERIOR DE INGENIERÍA

Aceleración de la Estimación del Flujo Óptico y su Aplicación al Códec de Vídeo MCDWT

Curso 2019/2020

Alumno/a: David Béjar Cáceres

Director/es:

Vicente González Ruiz

Aceleración de la Estimación del Flujo Óptico y su Aplicación al Códec de Vídeo MCDWT

David Béjar Cáceres

10 de febrero de 2020

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación

El presente trabajo viene motivado por la utilización del cálculo del flujo óptico en el códec MCDWT (Motion Compensated Discrete Wavelet Transform). Básicamente, MCDWT es una transformada basada en una DWT (Discrete Wavelet Transform) de tipo "2D+t", donde el dominio "2D" (espacial) es una DWT-2D estándar (que explota la redundancia en las imágenes del vídeo) y el dominio "t" es transformado por una MCTF (Motion Compensated Temporal Filtering), que elimina la redundancia temporal entre las imágenes del vídeo por medio de la Compensación de Movimiento (MC, Motion Compensation). El resultado es un conjunto de sub-bandas de de diferentes frecuencias espaciales y temporales. Una característica especial de MCDWT es que la información de movimiento no se transmite desde el codificador al descodificador. Esto es posible porque para estimar el movimiento en el codificador, solo se usa la información que es accesible por el descodificador [36].

Para la etapa MC, MCDWT utiliza actualmente la estimación del flujo óptico denso. El flujo óptico de imagen se refiere a la velocidad aparente correspondiente al desplazamiento observado de patrones de intensidad en sucesivas secuencias de imágenes [41]. El problema de estimación de flujo óptico es el núcleo de la visión computacional y se aplica a varios problemas, tales como el reconocimiento de formas, la edición de vídeo, la compresión de vídeo, los algoritmos de conducción autónoma, etc. [62]. El cálculo de flujo óptico de imagen no tiene una sola solución óptima (se dice por ello *ill-possed*). Por otra parte, existen varios métodos que se pueden agrupar en dos categorías, los métodos globales (como por ejemplo el de Horn/Schunck) y los métodos locales (como el de Lucas/Kanade, Farneback) usado por OpenCV [48].

Tradicionalmente, la estimación del flujo óptico en imágenes es un cálculo que consume muchos recursos de hardware y tiempo de procesamiento, lo que dificulta su utilización en diferentes aplicaciones. Además, el tiempo podría ser afectado por el tipo de secuencias de imágenes o la resolución (cantidad de píxeles) que se requieran procesar. Por tanto, encontrar formas de acelerar el cálculo es crucial en MCDWT.

En el segundo capítulo de este trabajo se mostrará una revisión bibliográfica de los diferentes autores e investigaciones más relevantes en los últimos años, además se presentará el estado del arte en estudios de flujo óptico de imagen, técnicas más utilizadas, limitaciones y estudios que actualmente se están

realizando. En el capítulo tercero se describirá los materiales, tecnologías y dispositivos para este trabajo con una pequeña descripción de cada uno. Además se detalla con mayor detenimiento la forma de conducir los experimentos, los parámetros de las repeticiones, guardado de datos en ficheros de salida y posteriormente cómo se cargan los ficheros para generar gráficas y realizar la discusión de los resultados. En el cuarto capítulo se realiza un resumen de la implementación con extractos del código más relevante y la forma de incluirlo al códec MCDWT de una manera transparente sin afectar al funcionamiento anterior, además las implementaciones para otra plataforma como es Android. El quinto capítulo presenta los resultados obtenidos en la aceleración del flujo óptico, los experimentos realizados, así como gráficas para un mejor análisis y discusión, comparando la calidad del flujo óptico.

1.2. Objetivos

En este TFM se propone implementar el cálculo de la estimación del flujo óptico de imagen para ser aplicado al códec de vídeo MCDWT. Más concretamente, se pretende:

1. Estudiar el cálculo de la estimación del flujo óptico de imagen sobre diferentes implementaciones en CPU [71] y GPU [70], identificando además las bibliotecas del estado de arte.
2. Estudiar la idoneidad de las bibliotecas (licencias, compatibilidad de hardware y software, capacidades multiplataforma).
3. Instalar y configurar las bibliotecas dentro del entorno de desarrollo (Linux).
4. Identificar limitaciones en el cálculo de la estimación de flujo óptico en diferentes plataformas.

1.3. Fases y cronograma asociado

A continuación se presenta una descripción del desarrollo temporal de este proyecto (véase la Figura 1.1):

1. ESTUDIO: Estudio de transformadas, transformadas discretas, codificación de sub-bandas y filtrado por paso de sub-bandas y flujo óptico de imagen (60 h).
2. IDENTIFICACIÓN: Identificación y selección de bibliotecas disponibles para cálculo óptico de imagen en el estado de arte actual (40 h).
3. INSTALACIÓN: Instalación y configuración de las bibliotecas en el entorno de desarrollo (30h).
4. DESARROLLO: Desarrollo para el cálculo de la estimación del flujo óptico de imagen (50h).
5. IMPLEMENTACIÓN: Implementación del cálculo de la estimación del flujo óptimo al códec MCDWT (50h).
6. EVALUACIÓN: Evaluación y pruebas del códec MCDWT (20h).

7. IMPLANTACIÓN: Identificar limitaciones o incompatibilidades de hardware y software en otros dispositivos/plataformas de las bibliotecas usadas (20h).

8. REDACCIÓN: Redacción y correcciones de la memoria previas a la presentación del TFM (30h).

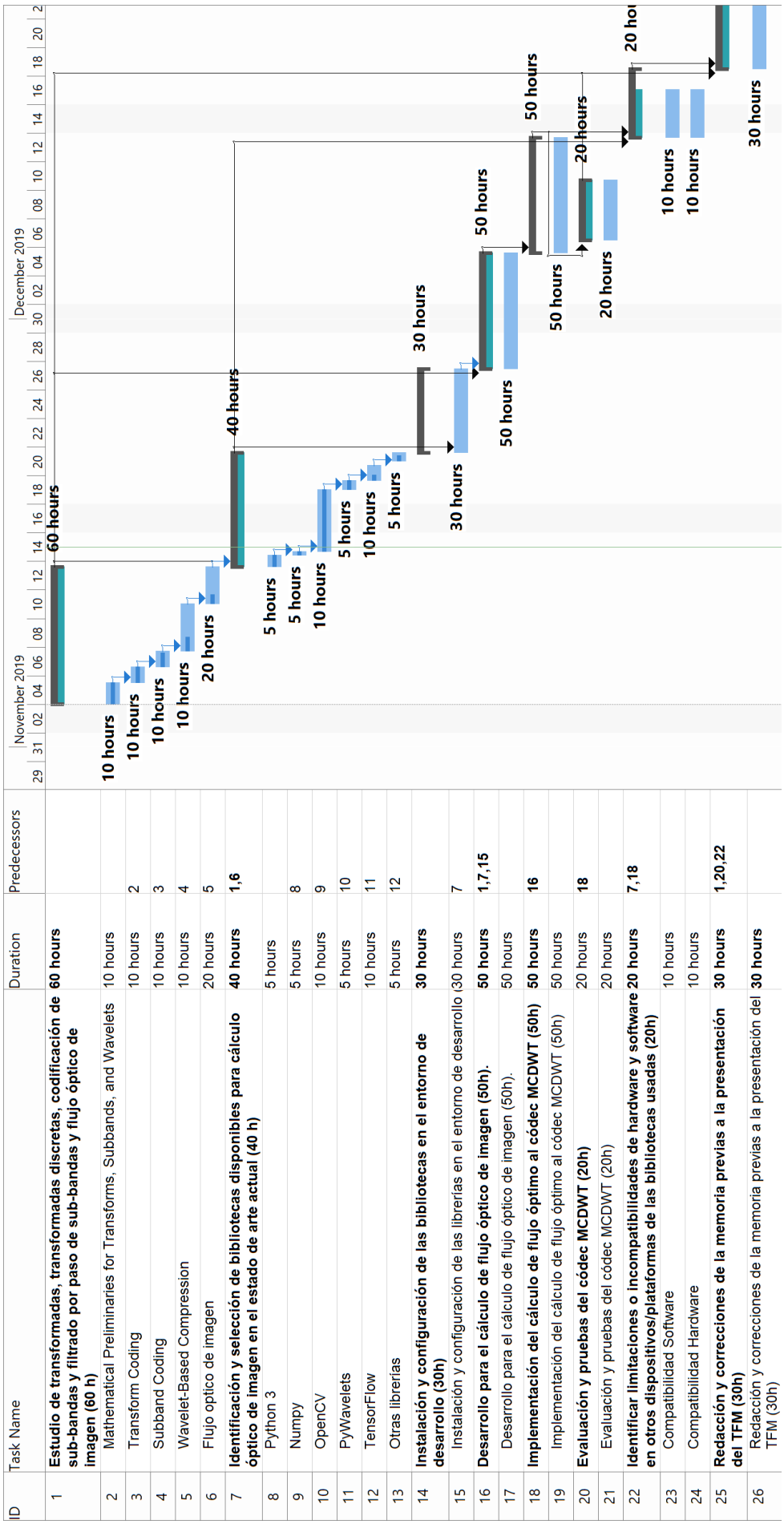


Figura 1.1: Tareas del proyecto y su temporización.

Capítulo 2

Revisión bibliográfica

2.1. Computación de altas prestaciones

La computación de altas prestaciones está motivada por la creciente demanda de velocidad para realizar cálculos cada vez más complejos en tiempos razonables, o tareas que históricamente han requerido de muchos recursos o tiempo para resolver, en diversos campos de ingeniería, simulación, ciencia, etc. A su vez, se divide en dos grupos: aplicaciones de alta productividad HTC y computación de altas prestaciones HPC (High Performance Computing). Las aplicaciones HPC tienen como objetivo aumentar el número de ejecuciones por unidad de tiempo y el rendimiento vienen dado por el número de trabajos ejecutados por unidad de tiempo. Estos jobs pueden tener tres modelos (véase la Figura 2.1), que pueden estar sincronizados cuando acaban al mismo instante. Si los jobs termina en diferentes instantes se los denomina de modelo asíncrono, o pueden tener un modelo maestro/esclavo cuando un trabajo está encargado de sincronizar al resto [27].

Por otro lado, tenemos a las aplicaciones de alto rendimiento HPC, que tienen como objetivo reducir el tiempo de las aplicaciones paralelas. Algunas de las aplicaciones HPC es para el estudio de dinámica de partículas a escala microscópica, o a escala macroscópica [65].

Actualmente la computación de altas prestaciones es sinónimo de paralelismo, es decir, el aumento de procesadores que realizan el cálculo. En general se requieren más procesadores para [27]:

- Mediante más procesadores, resolver tareas en menor tiempo.
- Mediante más memoria resolver problemas con mayor precisión.
- Resolver problemas reales con modelos matemáticos más complejos.

Por tanto, es crucial disponer de formas que aceleren la computación de cálculos complejos. La figura 2.2 demuestra la necesidad de disponer de formas para acelerar ciertos cálculos complejos:

2.1.1. Computación paralela

La industria ha dado un salto hacia la computación paralela en los recientes años. Por ejemplo, desde el 2010 casi todos los computadores destinados al consumo de las masas ya cuentan con procesadores

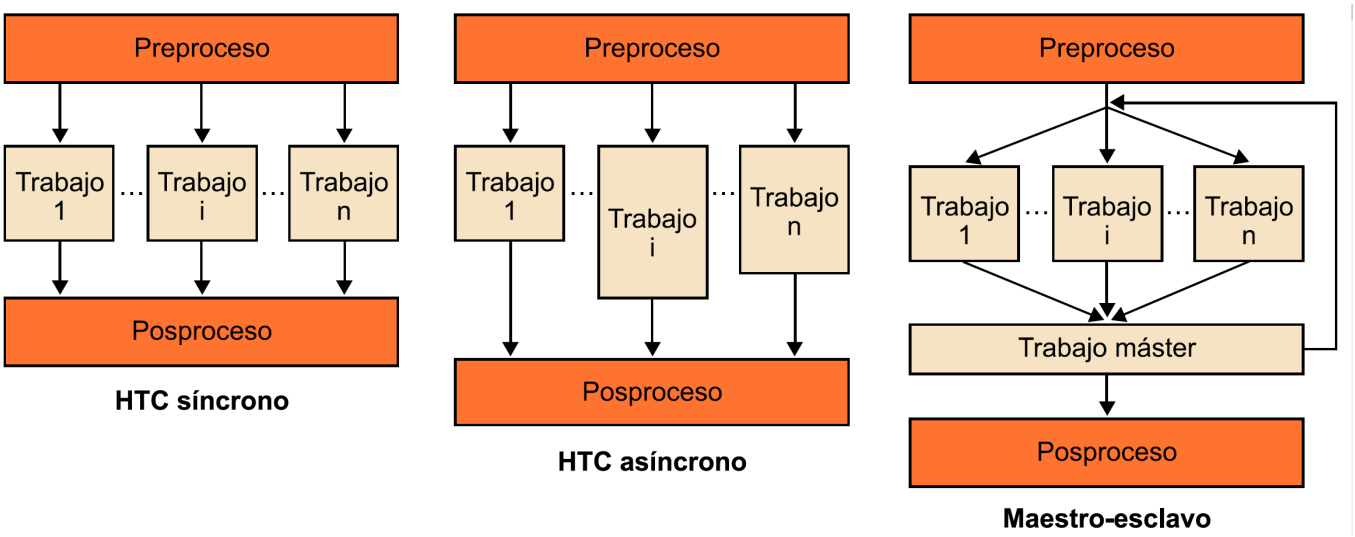


Figura 2.1: Modelos aplicaciones de alta productividad HTC [27].

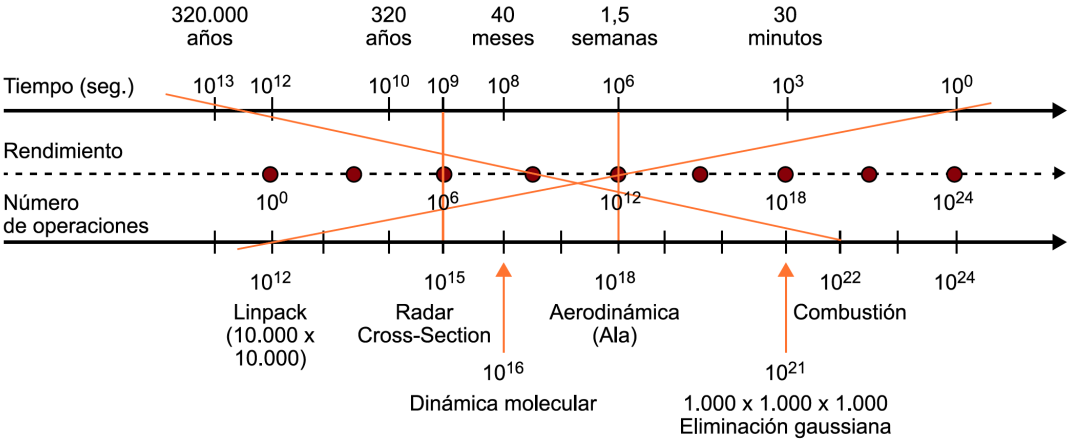


Figura 2.2: Necesidad de la computación de altas prestaciones para cálculos complejos [27].

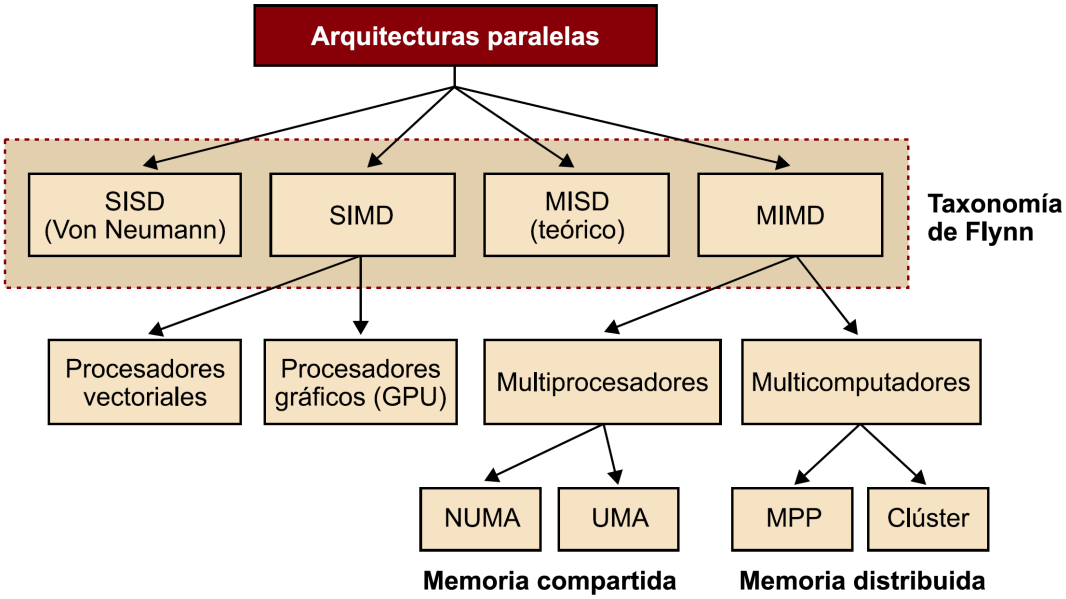


Figura 2.3: Arquitecturas paralelas [27].

multi-core, y desde la introducción del dual-core en laptops de gama baja, hasta estaciones de trabajo con 8 a 16 núcleos que tenemos hoy en día, se observa que la computación paralela no queda relegada a supercomputadores [59].

Por décadas los fabricantes han logrado mayor rendimiento aumentando las velocidades de la CPU a frecuencias que llagan hasta los 4 GHz/s, y aunque siempre ha sido fuente confiable de aumento de rendimiento, los fabricantes han tenido que ver alternativas para aumentar el rendimiento. El aumentar la cantidad de núcleos han dado origen a un incremento enrome en el rendimiento de supercomputadores y para los equipos de consumo personal [59].

La computación paralela normalmente suele utilizar la taxonomía de Flynn [34] para clasificar la arquitectura de computadores. Además, las arquitecturas paralelas pueden ser clasificadas de otras formas. El siguiente gráfico incluye un breve resumen de las clasificaciones de arquitecturas paralelas:

SIMD (Single Instruction, Multiple Data): En estos sistemas una misma instrucción se aplica a múltiples datos, como disponer de una única unidad de control con múltiples ALU. Entre los más populares están los procesadores vectoriales, como el uso de instrucciones NEON en plataformas móviles [44].

SIMD gráficos: Los procesadores gráficos funcionan tradicionalmente mediante pipelines de procesamiento por etapas muy especializadas y en un orden establecido, ocurriendo que cada pipeline presenta salidas que se requieren para la siguiente pipeline. Gracias a esta implementación de pipeline, el procesador gráfico puede ejecutar varias operaciones en paralelo. Aunque estos pipelines están prensados para trabajar con imágenes y renderizado, los procesadores gráficos actuales están orientados a propósitos generales que se discutirán en la siguiente sección [27].

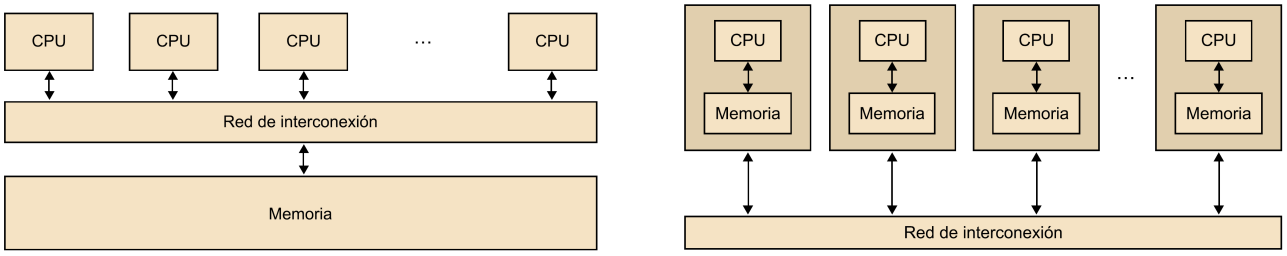
MIMD (Multiple Instruction, Mulitple Data): Los sistemas MIMD son unidades de procesamiento multi-núcleo independientes entre ellos, con sus propias unidades de control y ALU propia. Por lo general, estos sistemas son asíncronos, y cada parte del sistema opera a su propio tiempo (no existe un reloj

global y puede que no tengan relación entre los tiempos de los diferentes procesadores a menos que el programador los indique). Entre estos sistemas se diferencian dos tipos, los de memoria compartida y memoria distribuida. Entre los más conocidas están OpenMP [29] y MPI para memoria distribuida [21].

OpenMP admite programación paralela de memoria compartida multiplataforma en C/C ++ o Fortran. Define un modelo portátil y escalable, con una interfaz simple y flexible para desarrollar aplicaciones paralelas en plataformas desde el escritorio, dispositivos móviles hasta supercomputadores mediante pragmas [64].

En programación distribuida posiblemente la más famosa y usada es MPI. Este estándar de paso de mensajes permite aprovechar los procesadores en local, hasta escalar a cluster con varios o cientos de computadores [24]. MPI es simplemente un estándar, aunque existen implementaciones, entre ellas OpenMPI o MPICH.

El proyecto Open MPI es una implementación de interfaz de paso de mensajes de código abierto desarrollada y mantenida por un consorcio de socios académicos, de investigación y de la industria. Open MPI, por lo tanto, puede combinar la experiencia, las tecnologías y los recursos de toda la comunidad de informática de alto rendimiento para crear la mejor biblioteca MPI disponible. Open MPI ofrece ventajas para proveedores de sistemas y software, desarrolladores de aplicaciones e investigadores en ciencias de la computación [57].



(a) Arquitectura memoria compartida (ej. OpenMP) (b) Arquitectura memoria compartida (ej. OpenMPI)

Figura 2.4: Arquitecturas de memorias paralelas [27].

2.1.2. Computación en GPUs

En comparación a los procesadores tradicionales, el concepto de computación de propósito general en GPUs (GPGPU) es bastante nuevo. Mientras en los años 80s y principios de los 90s se dio un crecimiento muy importante en la demandas de procesamiento gráfico, debido a sistemas operativos gráficos y la industria de videojuegos, se empezó a desarrollar un marcado de procesadores especiales. Estos aceleradores de display ayudaban al hardware tradicional a las operaciones gráficas de la interfaz de usuario en los nuevos sistemas operativos. Al mismo tiempo, la industria empezó a desarrollar varios de estos procesadores especiales para aplicaciones de científicas y de visualización. Fue entonces cuando Silicon Graphics en 1992 libera la biblioteca OpenGL [47] destinada específicamente a procesamiento gráfico [59].

Desde el punto de vista de la computación paralela, durante el 2001 se presentó el mayor avance: Nvidia liberó 3 tarjetas gráficas, las primeras que implementaron el estándar DirectX 8.0 [19]. Este

estándar permitió por primera vez a los desarrolladores interactuar directamente con con las computaciones realizadas en la GPU. A principios de los 2000s, las tarjetas gráficas estaban diseñadas para producir color con cada pixel usando aritmética programable conocida como shaders. En general, estos pixels shaders usaban un sistema de coordenadas (x,y) en la pantalla, que con información adicional podrían presentar color. Esta información adicional podría ser color o texturas, y podía ser controlado por el programador, pero pronto se dieron cuenta que estos "colores" podrían ser cualquier dato. Así que los programadores creaban los pixel shaders para realizar cualquier computación arbitraria con estos datos extra para que la GPU la ejecutara. En esencia, se engañó a la tarjeta gráfica para correr cualquier tipo de cálculo general [59].

Debido a la capacidad de las GPUs para realizar operaciones paralelas para gráficos 3D, pronto se hizo evidente que estos procesadores también podrían usarse para realizar tareas informáticas de alto rendimiento. Como tal, se ha desarrollado un nuevo campo de investigación que ha desarrollado técnicas para explotar las tarjetas gráficas con fines de computación científica. En vista de la naturaleza de las GPUs ideadas para procesamiento gráfico, renderización, es muy importante tener en cuenta que algunos términos comunes en el ámbito gráfico pueden traducirse como las siguientes [28]:

- Texturas, pueden considerase como Arrays.
- Programas en la GPU, que pasan a ser Kernels que se ejecutan en la GPU.
- Renderización, que pasa a ser la ejecución del flujo del programa.

2.1.3. CUDA

CUDA es una especificación desarrollada por Nvidia para la plataforma de productos de computación gráfica GPU, que incluye las especificaciones de arquitectura y un modelo de programación asociado. Se desarrolló para aumentar la productividad en el desarrollo de aplicaciones de propósito general, desde el punto de vista del programador, el sistema está compuesto por una CPU tradicional y uno o más dispositivos GPUs [27].

CUDA pertenece al modelo SIMD (Simple Instruction, Multiple Data), que aprovecha el paralelismo a nivel de datos. En realidad Nvidia desarrolló su propia arquitectura SIMT (Single Instruction, Multiple Thread) que trabaja con Warps (véase la Figura 2.5), mientras SIMD ejecuta un solo "thread" de registros vectoriales sobre datos de memoria contiguos, CUDA utiliza multiples "threads" con instrucciones vectoriales y se pueden ejecutar sobre datos arbitrarios, es decir, no es necesario que estén en espacios de memoria contiguos. CUDA ejecuta sobre bloques "warps" de 32 threads que permiten acceder a sus propios registros de direcciones diferentes. Toda esta funcionalidad de CUDA se ejecuta de forma transparente al programador. [16].

Un programa en CUDA, consiste en algunas fases que pueden ser ejecutadas en secuencia por el procesador principal CPU, o bien en el dispositivo GPU. Se busca que el código donde necesita mayor paralelismo sea ejecutado en la GPU y el resto en el procesador principal. Esto requiere que exista un control explícito sobre la GPU por parte el programador, ya que la GPU maneja su propia memoria y organiza los flujos que serán ejecutados por el kernel [27].

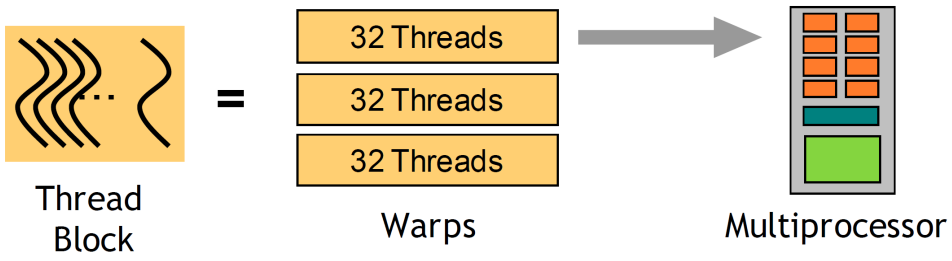


Figura 2.5: Arquitecturas paralelas [42].

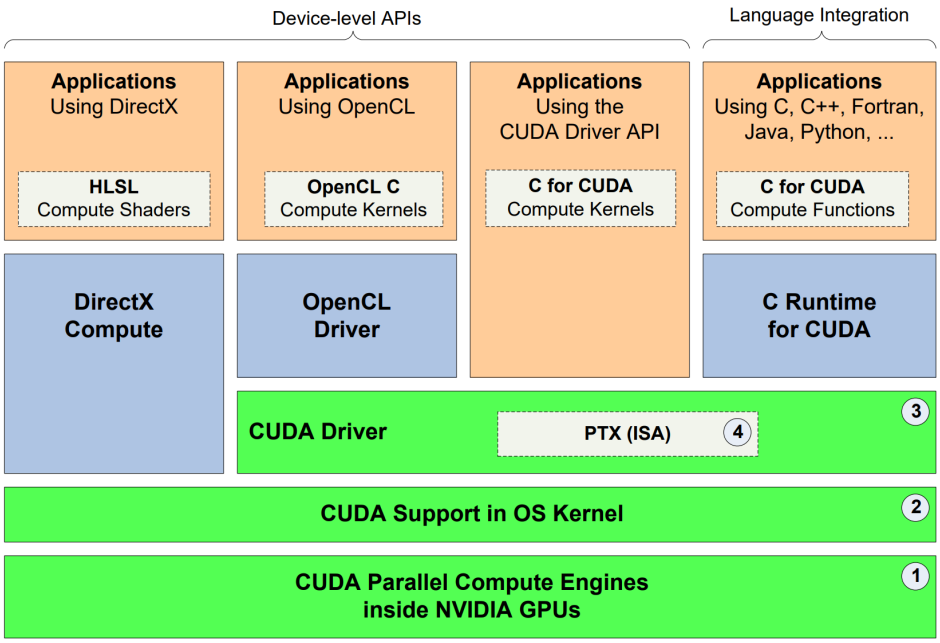


Figura 2.6: Arquitectura CUDA [53].

La arquitectura CUDA está compuesta por cientos o mieles de núcleos que pueden procesar datos enteros o punto flotante. Es gracias a esta cantidad de núcleos que permite altos niveles de paralelismo. Además, el ecosistema CUDA consiste en una serie de componentes y bibliotecas en el stack (véase la Figura 2.6) [50]:

- Motores de computación paralelos dentro de las GPUs Nvidia.
- Soporte a nivel de kernel en el sistema operativo para soporte de configuración, inicialización, etc.
- Driver que provee una API a los desarrolladores.
- PTX Set de instrucciones de la arquitectura (ISA) para computación paralela de kernels y funciones.
- PTX Set de instrucciones de la arquitectura (ISA) para computación paralela de kernels y funciones.

CUDA además tiene el desafío es desarrollar un software de aplicación que escale de manera transparente su paralelismo para aprovechar el creciente número de núcleos de procesador, de la misma manera que las aplicaciones de gráficos en 3D escalan de manera transparente su paralelismo a muchas GPU con muchos números de núcleos. CUDA sobrelleva este problema mientras mantiene una baja curva de

Tabla 2.1: Características CPUs vs GPUs [42].

CPUs	GPUs
Bajas latencias de acceso a los registros	Alto paralelismo de datos, alto rendimiento
Controla la lógica de ejecución	Más transistores dedicados a la computación
Decenas de Threads	Miles de threads

aprendizaje para los programadores que estén familiarizados con lenguajes como C/C++, al disponer de varias abstracciones para los grupos de threads, memoria compartida y barreras de sincronización [50].

Escalabilidad CUDA: Estos niveles de abstracción brinda paralelismo a nidel de datos y threads muy granular, permitiendo al programador dividir los problemas en su-problemas que pueden ser resueltos independientemente en paralelo con bloques de threads. Esta descomposición de los problemas provee al lenguaje de expresividad para que los threads interactúen entre ellos, dando origen a la escabilidad. Es este modelo de programación escalable que permite a la arquitectura CUDA abarcar un gran margen de resolución de problemas, simplemente escalando el número de procesos y particiones de memoria, desde las GPUs de gama baja hasta las tarjetas profesionales de alto rendimiento utilizadas en Clusters [50].

CUDA MPI: Además de la escalabilidad e interconexión que permite CUDA entre sus GPUs, también brinda la posibilidad de conectar mediante MPI (Message Passing Interface). MPI es compatible con CUDA y OpenACC, que fueron diseñados para correr en paralelo en un sólo computador o en los nodos de un clúster [52]. Además, el ecosistema cuenta con una abstracción CUDA-Aware MPI, que utiliza "Unified Virtual Addressing" (UVA), abstrayendo la capa de memoria entre los nodos del cluster. El programador no se tendrá que preocupar por el nodo, dispositivo e incluso MPI podrá transferir mensajes desde este espacio de memoria unificado [51].

2.1.4. OpenCL computación heterogénea

OpenCL es una interfaz estándar, abierta y de código libre multiplataforma en C++ pensada para plataformas heterogénea. La principal motivación fue simplificar al desarrollador la programación portable y eficiente en creciente cantidad de plataformas heterogéneas, CPU multi-core o GPUs [27]. OpenCL mejora en gran medida la velocidad y la capacidad de respuesta de un amplio espectro de aplicaciones en numerosas categorías de mercado, incluidos títulos de juegos y entretenimiento, software científico y médico, herramientas creativas profesionales, procesamiento de visión y capacitación e inferencia de redes neuronales [1].

Aunque OPenCL se asemeja a CUDA, tiene grandes diferencias. Por ejemplo, cuenta con modelo de gestión más complejo, y permite aprovechar el paralelismo a nivel de datos y nivel de tareas. Por otro lado, esta biblioteca no se limita a dispositivos de fabricantes en específico, sino que puede se utilizada por dispositivos de diferentes fabricantes, y puede ejecutarse sobre la CPU o GPU indistintamente de la plataforma.

2.2. Flujo óptico de imagen

El flujo óptico de imagen se refiere a la velocidad aparente correspondiente al desplazamiento observado de patrones de intensidad en sucesivas secuencias de imágenes [41]. El movimiento en fotogramas 2D es la proyección del movimiento tridimensional de objetos, en relación a la vista, en su plano de imagen. Las secuencias de imágenes ordenadas en el tiempo permiten la estimación del movimiento proyectado de la imagen 2D como velocidades de imagen instantáneas o desplazamientos de imagen discretos. Estos desplazamientos se conocen como flujo óptico o velocidad del campo de la imagen [22]. El problema de estimación de flujo óptico es el núcleo de la visión computacional y se aplica a varios problemas, tales como el reconocimiento de formas, la edición de vídeo, la compresión de video, los algoritmos de conducción autónoma, etc [62]. El cálculo de flujo óptico de imagen no tiene una sola solución óptima (se dice por ello ill-posed). Por otra parte, existen varios métodos que se pueden agrupar en dos categorías, los métodos globales (como por ejemplo el de Horn/Schunck) y los métodos locales (como el de Lucas/Kanade) usado por la biblioteca OpenCV [48]. Se sabe que los métodos locales son más robustos bajo ruido, mientras que las técnicas globales producen campos de flujo completamente densos [25].

El flujo óptico ha sido conocido por décadas, pero durante la década de los 80s es donde se idearon algoritmos de estimación para secuencias de imágenes. En concreto, el método de Lucas/Kanade [46] asume que todos los píxeles entre 2 imágenes son causados por el desplazamiento de los valores de cada pixel o cambios de sub-pixel [58]. En el año 2000 surge el método Farneback [20],

2.2.1. Estimación por métodos globales

Los métodos globales devuelven el campo óptico denso, pero son cómputos que requieren de muchos recursos. Entre estos se encuentran Horn-Schunck, basado en la minimización de energía, y TV-L1 [72]. Además, tienen las desventaja de ser mucho más sensibles al ruido. También está el método Farneback usado en este trabajo. Este método aproxima los pixeles vecinos de ambas imágenes en tiempos diferentes usando una función polinomial [20].

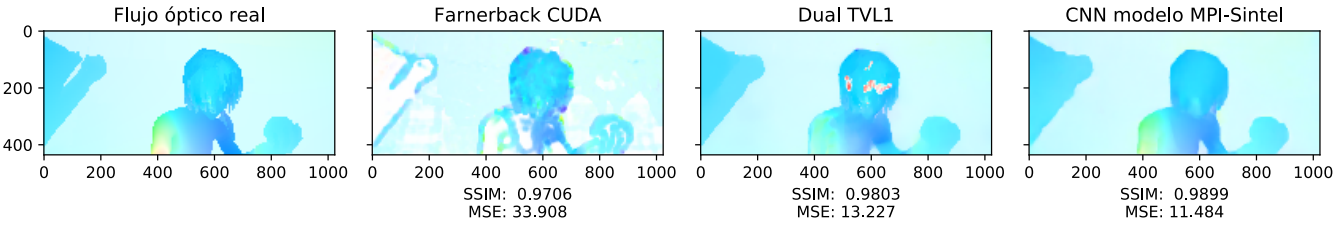
2.2.2. Estimación por métodos locales

Los métodos locales son más robustos bajo ruido. Entre estos están Lucas-Kanade [46], Bigun et al. [23], y su versión espacio variante [49]. Aunque los métodos locales tengan mejor rendimiento y robustez, no brindan los campos de flujo óptico denso [25].

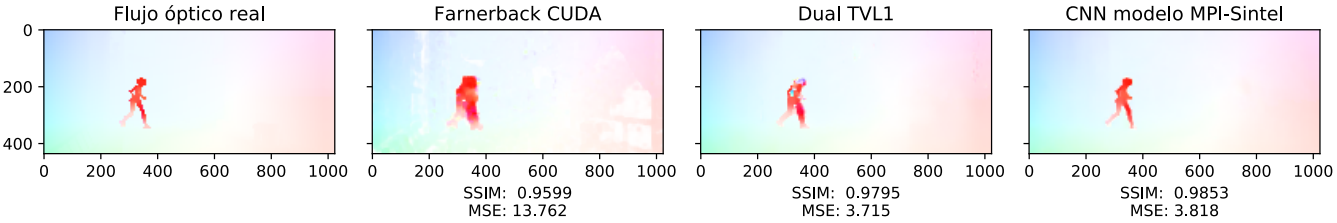
2.2.3. Otros métodos propuestos

Además de los métodos mencionados anteriormente, han ido surgiendo diferentes alternativas como la combinación de técnicas entre métodos locales y globales, y se ha propuesto utilizar una técnica de suavizado entre ambos métodos [25].

Hu, Song y Li proponen un método modificando de grueso a fino para calcular el flujo óptico denso y así reducir el ruido en a estimación. Sin embargo, tiene también sus desventajas cuando existen desplazamientos grandes en las imágenes. Este método, denominado "Ric", tiene una buena precisión en la



(a)



(b)

Figura 2.7: Múltiples métodos para calcular el flujo óptico denso de imagen.

estimación y más tolerante aldesplazamientos grandes en las imágenes. Por otro lado, en los últimos años, han surgido una nueva forma de estimar el flujo óptico de imagen, mediante el uso de redes neuronales se puede predecir el flujo óptico, y que prometen bastante dentro de este campo de estudio. Entre ellas se encuentra Flownet 2.0, el cual demostró que la estimación de flujo óptico de imagen puede ser considerado como un problema de aprendizaje. El hecho de plantearlo como un problema de aprendizaje desde los datos, lo alejó mucho de los métodos tradicionales, y se demostró que se requieren grandes cantidades de datos. Sin embargo, al usar múltiples datasets se mejoró la calidad de las estimaciones considerablemente, y el tiempo de procesamiento era similar al usar los métodos tradicionales, y con muy buena calidad de estimaciones en los datasets Sintel y KITTI [38].

Por otro lado, también existen métodos que hacen uso de redes neuronales convolucionales (CNN), como por ejemplo las propuestas por Gadot Wolf [60] y Bailer et al. [38], que producen una buena calidad de flujo óptico pero requieren un exhaustivo trabajo con los datasets, y son muy lentos para las aplicaciones prácticas. Además, los enfoques basados en parches carecen de la posibilidad de utilizar el contexto más amplio de toda la imagen porque operan en pequeños parches de imagen. Los métodos CNN producen mucho ruido o resultados borrosos, pero puede aplicarse algún tratamiento luego de la estimación para solventar esto [38].

El método FlowNet 2.0 fue revolucionario, sin embargo, requería grandes cantidades de parámetros para su correcto funcionamiento. Han surgido alternativas más eficientes, como PWC-Net que es un método que hace uso de CNN, mucho más liviano que FlowNet2, 17 veces más pequeño y más fácil de entrenar que el anterior, consiguiendo un mejor balance entre precisión y tamaño. PWC-Net permite correr a 35 FPS en el dataset Sintel con una Nvidia Pascal Titan X GPU [63].

2.3. Similaridad entre imágenes

Un aspecto muy importante en los sistemas multimedia es la posibilidad de evaluar la similaridad entre imágenes. Sobretodo, es usado en los códecs de vídeo donde se requiere comprar la calidad entre

(a) Original MSE 0, SSIM: 1.

(b) Bajo Contraste MSE $1401{,}41$ SSIM: $0{,}78$.

(c) Editada MSE $1076{,}53$ SSIM: $0{,}69$.

Figura 2.8: SSIM brinda una métrica más confiable.

la versión comprimida contra la versión en formato binario "raw". Existen varios métodos de evaluación: PSNR, SSIM, MS-SSIM, VQM, VQM-VDF, PEVQ, y PEVQ-S [61].

Las imágenes digitales pueden estar sujetas a varias distorsiones durante la captura, procesamiento, compresión, almacenamiento y transmisión, que puede resultar en la degradación de la calidad visual. Para la evaluación de la calidad de imágenes donde el espectador final es el ser humano, la manera de hacerlo es mediante métodos subjetivos, normalmente estos suelen ser inconvenientes, costosos en recursos de procesamiento y no siempre brindan un buen resultado al ojo humano [68].

2.3.1. Error medio cuadrático

Este tipo de error mide los cuadrados de las diferencias entre 2 imágenes: A menor el valor del MSE, mayor será la similaridad. Un problema con el error cuadrático medio es que depende en gran medida de la escala de la intensidad de la imagen. Esto se puede mitigar usando la métrica PSNR [67]. El MSE responde a

$$\mathrm{MSE} = \frac{1}{n}\sum_{t=1}^{n} e_t^2, \tag{2.1}$$

donde n es el número de pixels y e_t es el error para el t-ésimo pixel.

2.3.2. Índice de similaridad estructural SSIM

Tradicionalemente se han usado métodos subjetivos para cuantificar los errores visibles de las imágenes utilizando varias propiedades conocidas al sistema del ojo humano. El índice de similaridad estructural es una alternativa para comparar imágenes, y obtener la calidad de una imagen en comparación con otras [68].

El índice de similaridad estructural es una métrica objetiva. El índice de similaridad estructural comprende entre los valores -1 a $+1$, donde $+1$ representa la similitud perfecta entre imágenes. Este método ha sido probado ser mejor que el MSE, dando mejores resultados [61, 69, ?]. El SSIM se calcula como

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1) + (2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}, \tag{2.2}$$

donde μ es la media, σ la desviación típica, σ^2 la varianza estadística y C_1 y C_2 son dos costantes que

dependen del rango dinámico de las imágenes.

Se puede observar en la Figura 2.8 que puede existir una gran discrepancia entre el método MSE y SSIM. El índice de similaridad estructural es mucho más preciso, en la mayoría de los casos.

Capítulo 3

Materiales y métodos

3.1. Introducción

En este capítulo se presentan los principales elementos de cómputo y programación usados en este trabajo, que de forma resumida son:

- Lenguaje de programación Python 3 [66] para implementar el código.
- Código C++ con OpenCV 4.1.2 para estimar flujo óptico en diferentes plataformas.
- Sistema Operativo Linux Ubuntu 18 LTS Bionic [45] en la infraestructura OpenStack de la UAL [30], como entorno de desarrollo y test.
- Google Colab [39] y Nvidia Jetson [54] como entornos de ejecución acelerados.
- Dispositivo Android arquitectura ARMv8 para ejecutar flujo óptico.
- Python Jupyter Lab/Notebook [40] como herramienta de desarrollo (especialmente para Colab).
- LaTeX [33] para la redacción de la documentación asociada a este TFM.

3.2. Nvidia Jetson TX2

Los dispositivos Nvidia Jetson son plataformas extensibles que proporcionan rendimiento y eficiencia energética para ejecutar software en máquinas autónomas e inteligencia artificial. Cada Jetson es un SOM (System On Module) con su propio CPU (Central Processing Unit), GPU (Graphics Processing Unit), PMIC (Power Management Integrated Circuit), DRAM (Dynamic Random Access Memory) y almacenamiento flash. Esta plataforma contienen una pila de software dentro del paquete Nsight Developer Tools de Nvidia, donde en el Jetpack viene incluído CUDA (Compute Unified Device Architecture), Tensorflow (deep dearning), OpenCV (visión computacional), OpenGL (renderizado de gráficos), etc. [54].

En concreto, la placa Jetson TX2 es parte de la familia Jetson con una GPU de arquitectura Nvidia Pascal corriendo Ubuntu 18 LTS y la JetPack de Nvidia con todos los drivers y biblioyrvsd, las características se detallan en la Tabla 3.1 [55]:

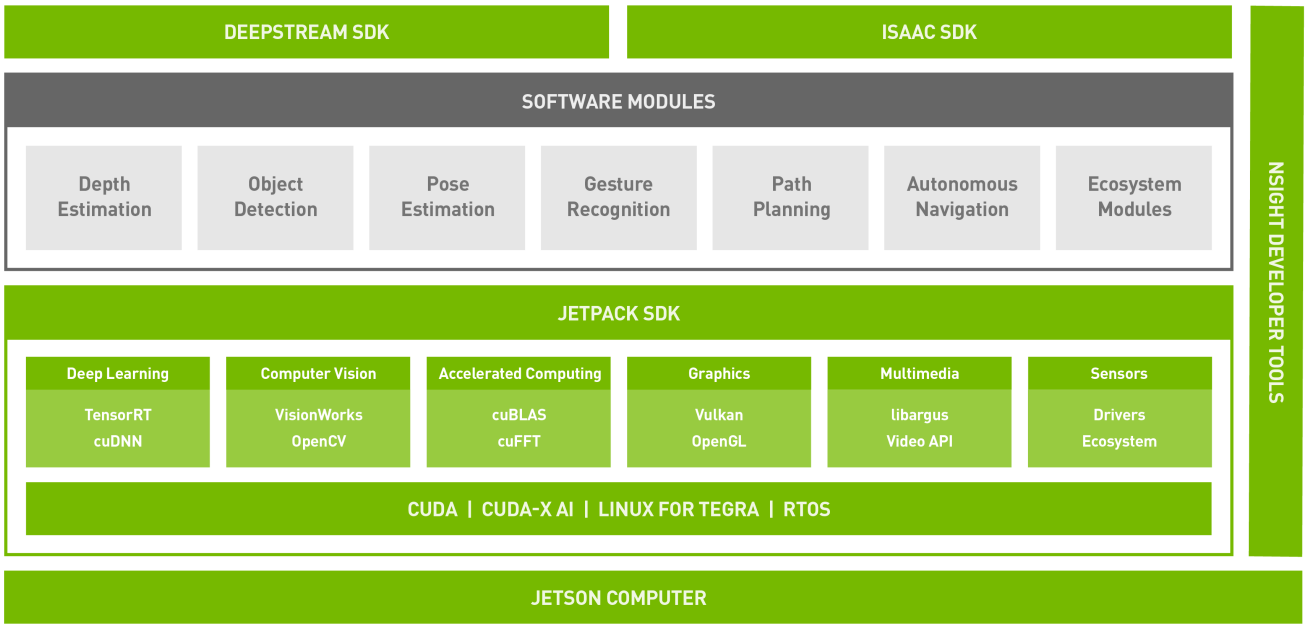


Figura 3.1: Pila de Software Jetson. [54]

Tabla 3.1: Especificaciones de la Nvidia Jetson TX2

	Jetson TX2
GPU	Arquitectura NVIDIA Pascal™ con 256 núcleos NVIDIA CUDA
CPU	CPU de 64 bits Denver 2 de doble núcleo y ARM A57 Complex
Memoria	LPDDR4 de 8 GB y 128 bits
Almcenamiento	eMMC 5.1 de 32 GB
Codificación de Video	3 de 4K a 30 cuadros (HEVC)
Descodificación de video	4x 4K @ 30 (HEVC)
Conectividad	Wi-Fi integrado, Gigabit Ethernet
Cámara	12 vías MIPI CSI-2, D-PHY 1.2 (30 Gbps)
Pantalla	HDMI 2.0/eDP 1.4/2 DSI/2 DP 1.2
UPHY	Gen 2 | 1x4 + 1x1 O 2x1 + 1x2, USB 3.0 + USB 2.0
Tamaño	87 mm x 50 mm
Mecánicas	Conector de 400 pines con placa de transferencia térmica (TTP)

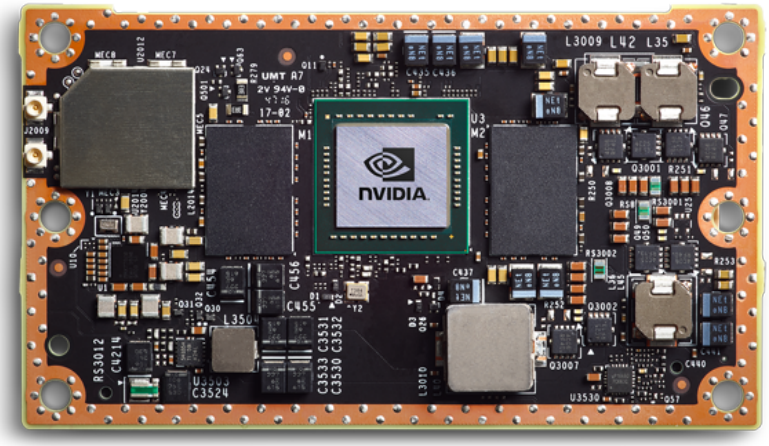


Figura 3.2: Módulo Nvidia Jetson TX2 [55]

3.3. Computación en GPU

Las GPUs son piezas de hardware especializadas para realizar operaciones de naturaleza gráfica. Entre los mayores usos han sido para videojuegos, codificación/descodificación de vídeo y procesamiento de imágenes. Además, en los últimos años con la evolución de las GPUs y el desarrollo de herramientas de software, ha sido posible la computación de propósito general en GPU y diferentes aplicaciones a la inteligencia artificial, visión computacional y procesamiento de otros algoritmos paralelos que antes eran sólo posibles en las CPUs.

Aunque puede variar entre diferentes fabricantes o incluso entre diferentes generaciones, a rasgos generales, se tiene una unidad de procesamiento que brinda un alto nivel de paralelismo. Más concretamente, donde una CPU puede ejecutar cálculos con sus 2 a 32 núcleos, una GPU puede utilizar cientos de núcleos e incuso conectar varios GPUs para disponer de miles de núcleos (véase la Figura 3.3).

Sin embargo, la utilización de la GPU no puede reemplazar a una CPU. La relación que manejan son de master/slave (maestro/esclavo), donde necesariamente se requiere una CPU que controle la ejecución. Cada GPU dispone de su modelos de memoria (cache, memoria global, compartida) y es gestionado mediante la CPU.

3.4. CUDA

CUDA es una plataforma de computación paralela y un modelo de programación desarrollado por Nvidia para la computación general en unidades de procesamiento gráfico (GPU). Con CUDA, los desarrolladores pueden acelerar drásticamente las aplicaciones informáticas al aprovechar el poder de las GPUs [2].

En las aplicaciones aceleradas por GPU, la parte secuencial de la carga de trabajo se ejecuta en la

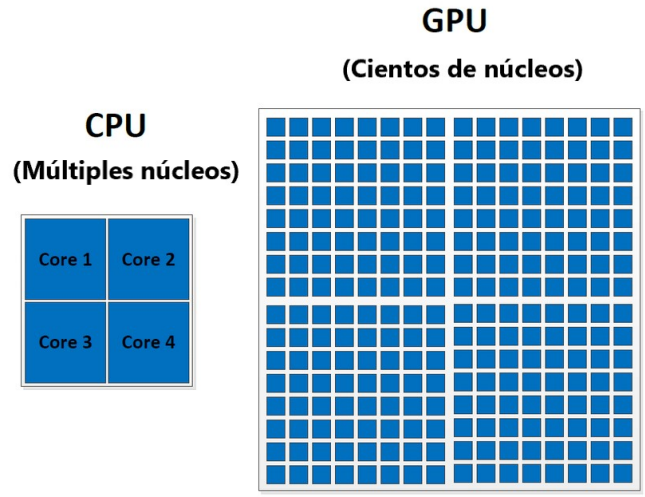


Figura 3.3: Comparación CPU vs GPU en cantidad de núcleos.

CPU (que está optimizada para un rendimiento de subproceso único) y la parte de computación intensiva de la aplicación se ejecuta en miles de núcleos de GPU en paralelo. Al usar CUDA, los desarrolladores programan en lenguajes populares como C, C ++, Fortran, Python y MATLAB, y expresan paralelismo a través de extensiones en forma de algunas palabras clave básicas.

3.5. Arquitectura Instrucción única, múltiples datos SIMD

CUDA saca provecho gracias a la arquitectura SIMT (Single Instruction, Multiple Thread), muy similar a SIMD (Simple Instruction, Multiple Data). Esta arquitectura se extiende de la taxonomía de Flynn. En concreto, mientras SIMD ejecuta un solo "thread" de registros vectoriales sobre datos de memoria contiguos, CUDA utiliza multiples "threads" con instrucciones vectoriales y se pueden ejecutar sobre datos arbitrarios, es decir, no es necesario que estén en espacios de memoria contiguos. CUDA ejecuta sobre bloques "warps" de 32 threads que permiten acceder sus propios registros de direcciones diferentes. Toda esta funcionalidad de CUDA se ejecuta de forma transparente al programador [16].

3.6. OpenCV

OpenCV (Open source Computer Vision library) es una biblioteca de visión y aprendizaje computacional con más de dos mil algoritmos optimizados que pueden ser utilizados para reconocimiento facial, identificación de objetos, análisis de vídeo, rastreo de objetos en movimiento, realidad aumentada, etc. [3].

OpenCV es multi-plataforma. Al ser escrita en lenguaje C y C++, puede ser ejecutada en casi cualquier sistema comercial, donde la mayoría de algoritmos están desarrollados en C++. Pero además, contiene "wrappers" para lenguajes como Python y Java, que permiten llamar a las funciones de la biblioteca. OpenCV corre en sistemas de escritorio (Windows, Linux, Android, MacOS, FreeBSD, OpenBSD) y móviles (Android, Maemo, iOS).

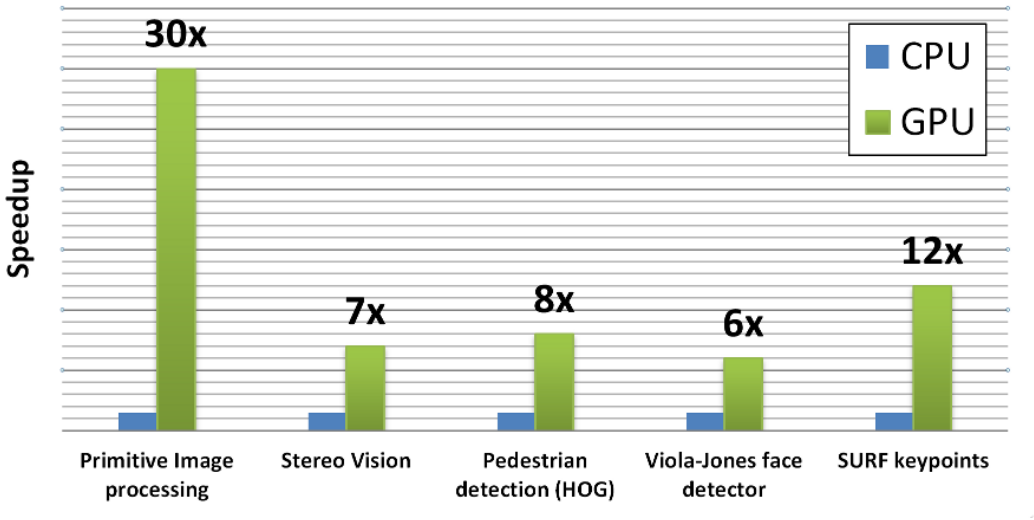


Figura 3.4: Mejora rendimiento OpenCV en CUDA Tesla C2050 versus Core i5-760 2.8Ghz [4].

3.7. OpenCV en CUDA

Desde 2010, OpenCV soporta y saca provecho a las arquitecturas CUDA para realizar cálculos en la GPU [5]. Por otra parte, estos dispositivos se han convertido en suficientemente potentes como para realizar computación de propósito general. Debido a que la visión computacional o la manipulación de imágenes puede sacar provecho de la computación paralela, los algoritmos pueden acelerarse al correr sobre las GPUs [4].

OpenCV promete grandes ganancias en rendimiento y consumo energético al correr sus algoritmos si se utiliza una GPU. Concretamente, las ganancias comprenden desde 6 a 30 veces más rápidos en las pruebas de rendimiento a comparación con la CPU (véase la Figura 3.4).

3.8. Python

Python es un lenguaje interpretado de código abierto, fácil de utilizar y que corre en múltiples plataformas, con un catálogo muy grande de módulos del estándar o de la comunidad [6].

El intérprete de Python se amplía fácilmente con nuevas funciones y tipos de datos implementados en C o C ++ (u otros lenguajes invocables desde C). Python también es adecuado como lenguaje de extensión para aplicaciones personalizables [7]. Es gracias a esta capacidad de extensibilidad de módulos en C++ que Python puede interactuar con OpenCV.

3.9. Jupyter

Jupyter Notebook es una aplicación web de código abierto que le permite crear y compartir documentos que contienen código en vivo, ecuaciones, visualizaciones y texto narrativo. Los usos incluyen: limpieza y transformación de datos, simulación numérica, modelado estadístico, visualización de datos, aprendizaje automático y mucho más [40]. Jupyter Notebooks es utilizado por Google para demos con CNNs (Convulational Neural Networks) y TensorFlow, visualización de imágenes, gráficos generados con Python,

Tabla 3.2: Parámetros compilación OpenCV 4.1.2.

Parámetro	Valor
Versión	4.1.2
Versión CUDA	10.0
CUDA_ARCH_BIN	62
WITH_CUBLAS	True
BUILD_opencv_python3	True
CUDA_FAST_MATH	True
ENABLE_FAST_MATH	True
WITH_CUDNN	True
CUDNN_VERSION	7.3.1
BUILD_opencv_dnn	True
OPENCV_DNN_CUDA	True
OPENCV_DNN_OPENCL	True
WITH_OPENCL	True
BUILD_opencv_cudaflow	True
WITH_PTHREADS_PF	True
WITH_LIBV4L	True
OPENCV_EXTRA_MODULES_PATH	/home/nvidia/opencv_contrib-4.1.2/modules
WITH_OPENGL	True

análisis de big data, o específicamente en este proyecto, para mostrar gráficos con las estimaciones de flujo óptico obtenidas con OpenCV, y leer los resultados mediante la biblioteca Pandas.

3.10. Compilación OpenCV en la Jetson

La capacidad de utilizar las funciones que permiten aprovechar CUDA en la Jetson requieren que OpenCV sea compilada desde el código fuente. Además requiere compilar los wrappers para que las funciones y todas las clases sean accesibles desde Python. En la Tabla 3.2 se muestra los parámetros utilizados para la compilación de OpenCV 4.1.2 + los módulos extras de OpenCV [32].

3.11. Diseño de los experimentos

Uno de los aspectos más importantes para conseguir resultados confiables, tiene que ver con la cantidad de datos generados durante los experimentos. Este trabajo está centrado en la aceleración conseguida para el cálculo de flujo óptico utilizando la GPU, y como recordemos, este cálculo es posiblemente el que mayor poder de procesamiento requiere dentro del códec MCDWT.

Además, al trabajar con multimedia, debe considerarse en que las diferentes resoluciones de las imágenes, la secuencia de imágenes que forman un vídeo, etc., pueden traer una carga gigante al momento de ser procesadas, dependiendo de la resolución o cantidad de píxeles. Por ejemplo, al procesar una secuencia de imágenes con resolución 854x480 (480p estándar) tendrán que procesarse 409920 píxeles, que normalmente corren a 30 fotogramas por segundo, es decir, más de 12 millones de píxeles por segundo. Ese número puede aumentar considerablemente si subimos a la resolución a 1280x720 (720p estándar),

llegando a más de 27 millones píxeles por segundo.

3.11.1. Aspectos a considerar para la evaluación de la aceleración

Como ya se menciona previamente en este trabajo, al trabajar con arquitectura CUDA, se está trabajando con un modelo master/slave. Por tanto, los datos que se procesan por la tarjeta gráfica deben ser copiados primero a la memoria de esta antes de ser procesados, y una vez finalizados, se deben mover nuevamente a la memoria del host para continuar con el flujo normal del programa. Este proceso de copia de datos entre el host y la GPU requiere un tiempo, que dependerá mucho del ancho de banda de memoria en ambos dispositivos, junto a la cantidad de datos que deban ser movidos entre las memorias. En vista de que la información mientras se encuentra en la memoria de la GPU, no puede ser utilizada para nada desde el host o para el funcionamiento del códec, se considera para los experimentos la medición del tiempo que transcurre desde que recibe imágenes el módulo Python, hasta que retornan el flujo óptico denso de imagen desde la GPU. Este será el valor real de aceleración conseguida dentro del funcionamiento del códec. El proceso para calcular el flujo óptico de imagen en la arquitectura GPU con el códec MCDWT es la siguiente:

- Módulo/Función Python recibe un par de imágenes desde el códec MCDWT.
- Reserva de memoria en la GPU (Memory allocation).
- Copia de datos desde la memoria del Host a la GPU.
- Calcular el flujo óptico denso de las imágenes de entrada.
- Copiar el flujo óptico de imagen desde la memoria de la GPU al Host.
- Módulo/Función retorna el flujo óptico.

3.11.2. Secuencias de imágenes, resoluciones e iteraciones

Para evaluar la aceleración conseguida se tomaron dos sets de imágenes y las siguientes resoluciones:

Vídeo 1, Generado por Computador Big Buck Bunny [9]:

- 480 x 854 pixels (480p).
- 720 x 1080 pixels (720p).
- 1080 x 1920 pixels (1080p).

Vídeo 2, Automóviles en movimiento [8]:

- 480 x 854 pixels (480p).
- 720 x 1080 pixels (720p).
- 1080 x 1920 pixels (1080p).

(a) Video generado por computador, Big Buck Bunny [9].

(b) Video 2, vehículos en movimiento [8].

Figura 3.5: Vídeos usados para los experimentos.

Número de iteraciones

Para evaluar la aceleración conseguida por le uso de la GPU se realizan experimentos con 599 fotogramas para cada una de las dos secuencias de imágenes, tanto para CPU y luego para GPU. Con la finalidad de obtener resultados consistentes, se repite el experimento 100 veces. Los experimentos son automatizados por scripts de Python (ssim-error-experiments.py) y miden los tiempos en completarse cada experimento[1].

Lectura y presentación de los datos generados

Los resultados generados con el script python benchOpticalFlow.py, que guarda en archivos de texto ".txt", y que luego serán cargados con un Jupyter Notebook directamente usando pandas (Python Data Analysis Library) [?]. Se crean Dataframes con los que se pueden realizar operaciones estadísticas y mostrar los resultados (véase la Figura 3.6).

[1] los tiempos para cada uno son guardados en archivos de texto ".txt".

```python
# Codigo Python
resuldatosGPU: list = [
  # Video 1 CGI
  "v1_480.mp4.txt",
  "v1_720.mp4.txt",
  "v1_1080.mp4.txt",
  # Video 2 Cars moving
  "v2_480.mp4.txt",
  "v2_720.mp4.txt",
  "v2_1080.mp4.txt",
]
resultadosCPU: list = [
  # Video 1 CGI
  "v1_480_cpu.mp4.txt",
  "v1_720_cpu.mp4.txt",
  "v1_1080_cpu.mp4.txt",
  # Video 2 Cars moving
  "v2_480_cpu.mp4.txt",
  "v2_720_cpu.mp4.txt",
  "v2_1080_cpu.mp4.txt"
]
```

```python
# Codigo Python
    ############ LOADS ALL DATA FROM TEXT FILES ###############
def importTxtFiles(fileNames: list) -> pandas.core.frame.DataFrame:
# Loads first file to create DataFrame
FirstFile = pandas.read_csv(fileNames[0], names = [fileNames[0].split(".")[0]], header=None) df: pandas.core.frame.DataFrame = pandas.DataFrame(FirstFile)

for resultFile in fileNames[1:]:
resultadosArchivo = pandas.read_csv(resultFile, names = [resultFile.split(".")[0]], header=None)
df.insert(df.shape[1], resultFile.split(".")[0], resultadosArchivo, True)
print(resultFile)


return df
```

Figura 3.6: Lectura de resultados rendimiento.

Una vez cargado los datos en pandas, se muestran gráficos y datos estadísticos de los resultados generados. JupyterLab [14] permite la visualización en forma de gráficos interactivos con Plotly [18], o gráficas estáticas que son utilizados en este trabajo. En la Figura 3.7 se puede observar Jupyter Notebook corriendo en la plataforma Colab de Google generado con el notebook VisualizadDatos.ipynb.

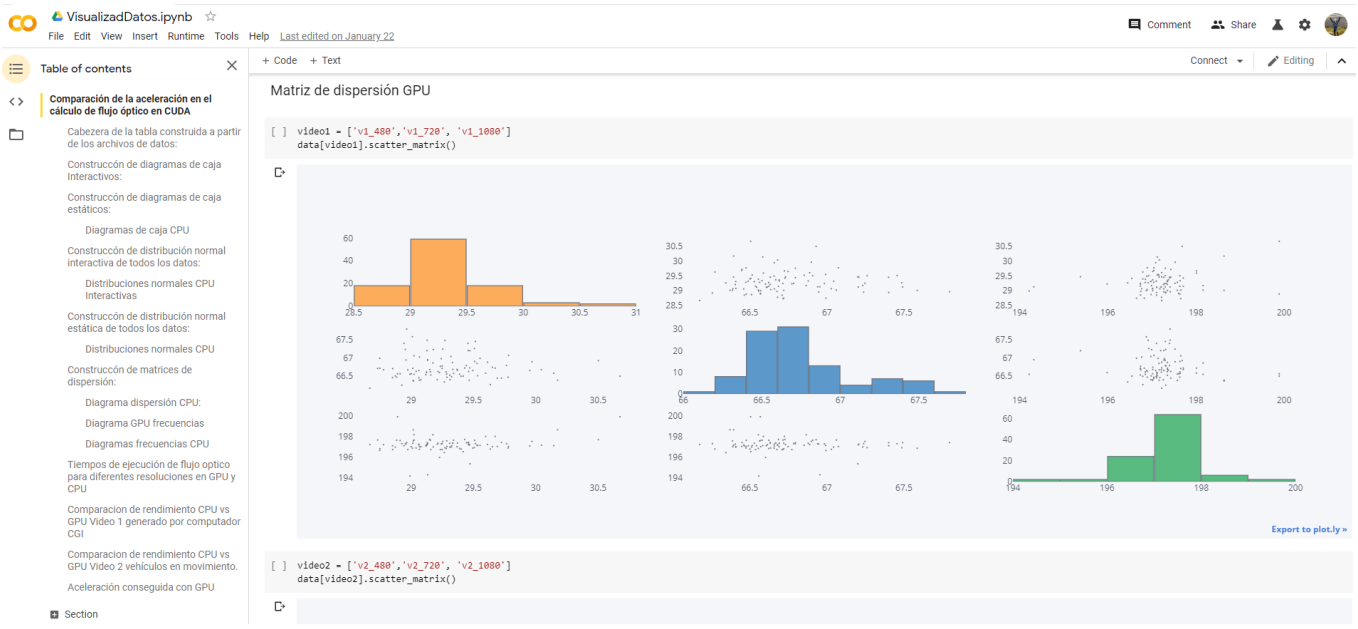


Figura 3.7: Jupyter Notebook con el análisis estadístico y gráficos interactivos en Colab.

3.11.3. Evaluación de la precisión de la estimación de flujo óptico

La precisión de la estimación del flujo óptico de imagen se evalúa considerando el índice de similaridad estructural, o error SSIM, descrito más a profundidad en el capítulo anterior. En concreto, comparamos la similitud del flujo óptico de imagen generado con el real.

Los flujo ópticos de imagen reales existen en varios sets de datos. En concreto, el set que servirá para este trabajo a modo de comparación será MPI-Sintel, un dataset muy completo y relativamente pequeño en tamaño (poco más de 5GB), en comparación de otros sets de datos. MPI-Sintel contiene varias secuencias de imágenes generados por computador, junto con el flujo óptico de imagen real entre cada par de fotogramas. El dataset incluye una codificación a color para el flujo óptico de imagen, que permitirá ver de forma más clara las diferencias entre el flujo real y el conseguido utilizando Fernerback (OpenCV) en la GPU [26].

Para obtener el índice de similaridad estructural SSIM, se prevee usar la siguientes (sub-)secuencias del dataset MPI-Sintel:

- Alley 1: con 50 fotogramas.
- Alley 2: con 50 fotogramas (mucho movimiento).
- Shaman 1: con 50 fotogramas.
- Sleeping 1: 50 fotogramas.
- Temple 3: 50 fotogramas (mucho movimiento).
- Market 5: 50 fotogramas (mucho movimiento).
- Cave 2: 50 fotogramas (mucho movimiento).

Una vez más, mediante un script en Python serán cargados los flujos ópticos reales en formato ".flo" del dataset. Cada flujo óptico viene acompañado de un par de imágenes que dan origen al flujo óptico de imagen. El script toma estas imágenes, realiza el cálculo de (la estimación del) flujo óptico en la GPU, lo transforma al mismo formato usado en la mayoría de bibliografía estándar, y realiza la comparación entre el flujo obtenido con el real.

Para obtener el índice de similitud entre ambos existen varias formas de implementarlo: (1) OpenCV [10], (2) Scikit-Image [13], y (3) Tensorflow API [11]. Este trabajo provee scripts con implementaciones para cada una de estas formas. Sin embargo, para evaluar la similitud se opta por usar OpenCV, por la relevancia que tiene el framework dentro del códec MCDWT, y porque es fácilmente portarlo con C++ a diferentes plataformas, mediante el script ssim-error-flow.py que contiene una función que devuelve el índice de similaridad estructural entre las 2 imágenes a evaluar.

Capítulo 4

Implementación

La implementación tiene como objetivo incluir todo el código de la aceleración del flujo óptico de imagen al códec MCDWT. El código propuesto utiliza Python, pero especificando tipo de variables, "strongly typed", ya que Python actúa simplemente como un Wrapper que llama a funciones en C++, el intérprete de Python no necesariamente utiliza las mismas clases que las funciones en C++. Una ventaja al especificar el tipo de datos es la facilidad con la que se puede portar el código a C++, para poder así portar a otras plataformas.

El cálculo de flujo óptico usando a Python ha sido probado en ambos sistemas operativos, Ubuntu y Windows. Además, para los sistemas que no cuentan con un intérprete Python nativo (como Android), se puede aprovechar compilando OpenCV desde la fuente y pudiendo ser llamado desde Android para calcular el flujo óptico de imagen. En las siguientes secciones se explica con más detenimiento las implementaciones.[1]

4.1. Implementación al códec MCDWT

Al momento de implementar el código acelerado, se ha intentado realizar de la forma más transparente posible, primero incluyendo en la clase una forma de comprobar que el dispositivo tenga capacidades CUDA y sean accesibles las funciones C++ desde los wrappers de OpenCV. Esto es muy importante porque el códec está pensado para ser ejecutado en una gran variedad de plataformas y con diferentes sistemas operativos. Una vez comprobada la compatibilidad del código acelerado, se procede a llamar la función que calcula el flujo óptico en la GPU y se devuelve un flujo óptico al códec para que siga con la ejecución.

4.1.1. Comprobación funcionalidad CUDA

Antes de intentar realizar los cálculos en una GPU CUDA, es necesario comprobar que el sistema es capaz de ejecutar el código que permite generar el flujo óptico de imagen. Para que sea compatible debe cumplir algunos requisitos:

- Disponer de una tarjeta gráfica CUDA.

[1] Sin embargo, nótese que a propósito se han excluido detalles y código que no sea relevante para la redacción de este documento.

```python
# Codigo Python
cuda_enabled = False
cuda_turing_aceleration_sdk = False

try:
  # Check if can allocates memory on the GPU
  cuMat1: cv2.cuda_GpuMat = cv2.cuda_GpuMat()
  # Checks if python wrappers for OpenCV is accessible from Python
  opticalFlowGPUCalculator = cv2.cuda_FarnebackOpticalFlow.create(
    10, 0.5, False, 15, 3, 5, 1.2, 0)
  # If no error is shown, activates the CUDA functionality
  cuda_enabled = True
except:
  print("No CUDA support")
# Codigo Python
```

Figura 4.1: Chequeo del soporte para CUDA (Python).

- Tener instalado los drivers de Nvidia y CUDA.
- Usar una distribución de OpenCV 4.+.
- Disponer de los Wrappers de Python para OpenCV con soporte CUDA.

A continuación se muestra una sección del código donde se comprueba estas capacidades en el sistema. Una vez que lo confirma, se activa para todo el módulo Python. Así, MCDWT lo ejecuta automáticamente y crea la clase del método Farneback que se usará luego (véase la Figura 4.1).

4.1.2. Función Farneback Python/C++

En caso de ejecutarse en un sistema que no tenga CUDA implementado, no utilizará ningún tipo de aceleración, pero se mantiene la compatibilidad para todo tipo de plataforma. Sin embargo, al comprobar si el modo acelerado está activado dentro del módulo, se llama a la función que se encargará de la aceleración. Este proceso se hace de una manera transparente e independiente del resto del códec. En la Figura 4.2 se observa como la función original del codec lo implementaba junto a la nueva versión.

4.1.3. OpticalFlow tarjetas gráficas Turing

Recientemente Nvidia liberó una API NVIDIA Optical Flow SDK para el cálculo de flujo óptico de imagen. Esta API aprovecha las funcionalidades de CUDA y el alto nivel de paralelismo de las GPU modernas y promete aligerar la carga en la CPU y la GPU porque tiene un hardware dedicado encargado de este cálculo (no utiliza los núcleos CUDA).

En este trabajo se deja una implementación en caso el códec se ejecute en este tipo de dispositivos a futuro. Sin embargo, la Jetson utilizada no tiene la arquitectura ni las capacidades de hardware para ejecutarlo. También se incluye el código para que lo ejecute de forma transparente al resto del códec, y que prueba las capacidades del sistema donde se ejecuta y activa la API (véase la Figura 4.3).

```python
# Codigo Python
def opticalFlowCuda(next_y: np.uint8, curr_y: np.uint8) -> cv2.UMat:
  if cuda_turing_aceleration_sdk:
    return opticalFlowCuda_testla(next_y, curr_y)
    # Allocates memory on GPU
  next_y_gpu: cv2.cuda_GpuMat = cv2.cuda_GpuMat()
    # Allocates memory on GPU
  curr_y_gpu: cv2.cuda_GpuMat = cv2.cuda_GpuMat()

  # Moves image to GPU allocated memory
  next_y_gpu.upload(next_y)
  # Moves image to GPU allocated memory
  curr_y_gpu.upload(curr_y_gpu)

  flowGPU: cv2.cuda_GpuMat = opticalFlowGPUCalculator.calc(
    next_y_gpu, curr_y_gpu, None)
  # Copies the optical flow from GPU memory to Host memory
  flow:  cv2.UMat = flowGPU.download()
  return flow
```

```cpp
// Codigo C++
void opticalFlowCuda(cv::UMat next_y, cv::UMat curr_y) {
  // Allocates memory on GPU for frames
  cv::cuda::GpuMat next_y_gpu
  cv::cuda::GpuMat curr_y_gpu
  cv::cuda::GpuMat flujo_gpu(next_y.size(), CV_32FC2);
  // Moves images from host to gpu memory
  next_y_gpu.upload(next_y)
  curr_y_gpu.upload(curr_y)
  //Starts the class to calculate the flow
  cv::cuda::FarnebackOpticalFlow opticalFlowGPUCalculator = cv::cuda::FarnebackOpticalFlow::create(
    int numLevels = 5,
     double pyrScale = 0.5,
     bool fastPyramids = false,
     int winSize = 13,
     int numIters = 10,
     int polyN = 5,
     double polySigma = 1.1,
     int flags = 0);
  opticalFlowGPUCalculator.calc(next_y_gpu, curr_y_gpu, flujo_gpu)
  cv::UMat flow
  flujo_gpu.download(flow)
  return flow.clone();
```

Figura 4.2: Código acelerado en la GPU Python (izquierda) y C++ (derecha).

```python
# Codigo Python
if cuda_enabled:
  try:
    blank_image = np.zeros((500, 500, 3), np.uint8)
    blank_image.fill(200)
    nvof = cv2.cuda_NvidiaOpticalFlow_1_0.create(
      blank_image.shape[1], blank_image.shape[0], 5, False, False, False, 1)
    flow = nvof.calc(cv2.cvtColor(blank_image, cv2.COLOR_BGR2GRAY), cv2.cvtColor(
      blank_image, cv2.COLOR_BGR2GRAY), None)
    cuda_turing_aceleration_sdk = True
  except:
    print("No CUDA Turing GPU")
  pass
  nvof = cv2.cuda_NvidiaOpticalFlow_1_0.create(
    next_y.shape[1], next_y.shape[0], 5, False, False, False, 1)
  flow:  cv2.UMat = nvof.calc(next_y, curr_y, None)
  return flow
```

Figura 4.3: Chequeo de la disponibilidad de hardware Turing (Python).

```
externalNativeBuild {
  cmake {
    arguments "-DANDROID_STL=c++_shared", "-DHAVE_NEON=1", "-DANDROID_ARM_NEON=TRUE", "-
  DANDROID_ABI=arm64-v8a", "-DCMAKE_BUILD_TYPE=Release"
    targets "opencv_jni_shared"
    abiFilters 'arm64-v8a'
    cppFlags "-std=c++17 -fopenmp -O3 -mfloat-abi=softfp -mfpu=neon -flax-vector-conversions"
  }
}

```

Figura 4.4: Parámetros de compilación de OpenCV en Android.

```
// Codigo Kotlin
private val mLoaderCallback = object : BaseLoaderCallback(this) {
  override fun onManagerConnected(status: Int) {
    when (status) {
      LoaderCallbackInterface.SUCCESS -> {
        Log.i(TAG, "OpenCV loaded successfully")
        mOpenCvCameraView!!.enableView()
      }
      else -> {
        super.onManagerConnected(status)
      }
    }
  }
}
```

Figura 4.5: Carga librerías nativas OpenCV desde Kotlin.

4.2. Implementación flujo denso en Android

Finalmente, también se presenta una implementación del cálculo de flujo óptico para el sistema operativo Android [15], que gracias a la JNI (Java Native Interface) permite operar desde la máquina virtual Java JVM (Java Virtual Machine) con bibliotecas escritas en C++/C o ensamblador [17].

Para llamar a las funciones de OpenCV C++ desde la JVM se usa la JNI, y el código de OpenCV se debe compilar como un módulo de Android. La compilación de código C++ en Android requiere el NDK (Native Development Kit) [31]. Así, OpenCV quedará compilado para la arquitectura requerida y una vez compilada la biblioteca y configurada, se puede realizar llamadas desde el lenguaje Kotlin [35] que corre sobre la JVM. En la Figura 4.4 se aprecia los parámetros de compilación del módulo OpenCV para Android.

Una vez configurada la compilación, se procede a compilar el código en Kotlin llamando las funciones de OpenCV. Además, desde la Activity principal de la app (MainActivity.kt) se importan toda todas las clases que serán usadas. El código es muy parecido al presentado en la sección anterior. Primero se carga la biblioteca desde Kotlin (véase la Figura 4.5).

Una vez que se ha cargado correctamente OpenCV, ya puede interactuar con la JVM (Java Virtual Machine). La implementación es muy parecida a la vista en la anterior sección para el resto de plataformas (véase la Figura 4.6).

```
// Codigo Kotlin
class MainActivity : AppCompatActivity(), CameraBridgeViewBase.CvCameraViewListener2 {
  lateinit var prevImage: Mat
  lateinit var curImage:Mat
  lateinit var flowMat:Mat
  lateinit var points: MutableList<Point>
  // Codigo Kotlin

  fun opticalFlowOpenCV(prevImage: Mat, curImage: Mat):Mat{
      val prevImage_gray: Mat = prevImage.gray()
      val curImage_gray: Mat = curImage.gray()
      val flow =  Video.calcOpticalFlowFarneback(prevImage, curImage, flowMat, 0.4, 1, 8, 10, 8,
    1.2, 0)
      return flow
```

Figura 4.6: Llamada al estimador de flujo óptico nativo.

Como se puede ver en dicha figura, la app toma las imágenes desde la cámara y calcula el flujo óptico entre ellas. Por tanto, se puede ver que gracias al uso de código C++ de OpenCV, éste aumenta la cantidad de plataformas soportadas. Nótese que en esta plataforma no se han corrido pruebas de rendimiento, ni otro test. Se lo ha hecho específicamente para comprobar la portabilidad entre las diferentes plataformas y arquitecturas, de acorde con los objetivos de este trabajo.

Capítulo 5

Resultados y discusión

5.1. Tiempos de los experimentos GPU vs CPU

En la Figura 5.1 se muestra los tiempos promedio de calcular el flujo óptico de imagen para 599 experimentos. Se observaron mejoras considerables de los tiempos de ejecución al usar la GPU. Tanto para el primero como para el segundo vídeo. Se observa también que no hay diferencias significativas entre los tiempos de procesado para el vídeo generado por computador "CGI" a comparación con el vídeo capturado con una cámara. Además, el tamaño de los fotogramas es crucial para el tiempo de procesado, al aumentar la cantidad de píxeles a procesar, aumenta considerablemente el tiempo total del experimento.

5.2. Aceleración conseguida usando la GPU

Aunque el tiempo de procesado puede ser diferente en los diferentes tamaños de fotogramas, el objetivo final es determinar la cantidad de aceleración que se consigue usando la GPU a comparación de la CPU. En la Figura 5.2 se puede ver la aceleración conseguida para cada vídeo, en los diferentes tamaños de fotogramas.

Al observar la anterior gráfica se puede ver que, el procesamiento de flujo óptico de imagen utilizando la GPU llega a ser más de 4 veces más rápido que al utilizar la CPU. Esta es una aceleración muy significativa, que sin duda mejora mucho el rendimiento del códec MCDWT, la operación de flujo óptico de imagen es la operación que mayor consumo de recursos y tiempo tiene dentro del códec.

Por otro lado, la aceleración conseguida con la GPU disminuye considerablemente mientras el tamaño de fotogramas sube. Este efecto viene dado por un motivo: la cantidad de píxeles que se procesan para el flujo óptico, como ya se describió en el Capítulo 2, entre la menor resolución (480×854) y la de mayor resolución (1080×1920), es de casi 10 veces, y según la arquitectura CUDA, todos estos datos deben primero cargar a memoria del Host, y luego reservar memoria en la GPU. Este proceso de reserva en memoria consume gran cantidad de tiempo en CUDA, y aunque hay muchas formas de optimizarlo, dependerá de la habilidad del programador junto con la compresión del sistema o hardware en el que se trabaja. En concreto, al procesar secuencias con resolución mayores a 720 x 1080, en el hardware de la Jetson el ancho de banda de la GPU llega a unos 59 GB/s, muy baja en comparación a las tarjetas gráficas modernas que llegan a más de 600 GB/s.

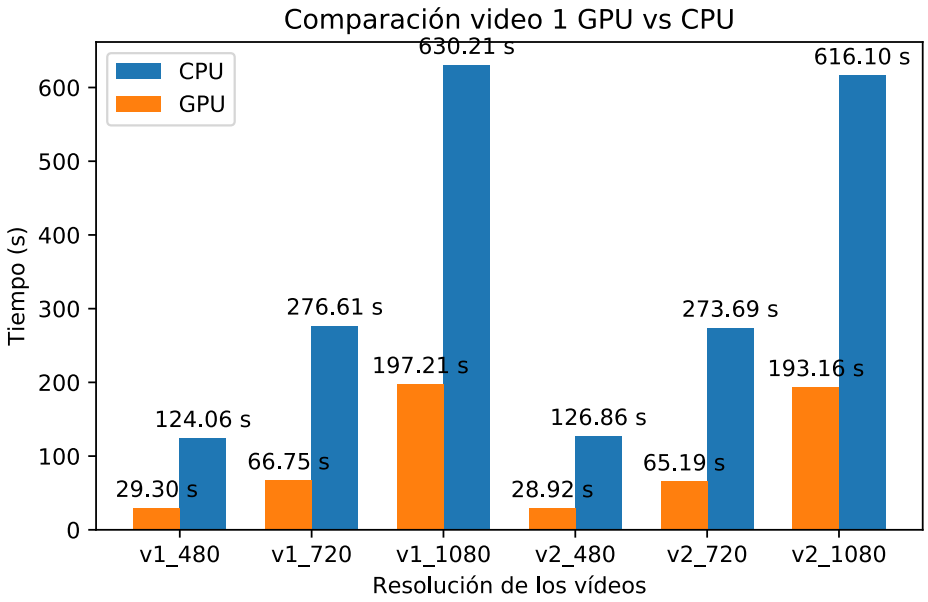
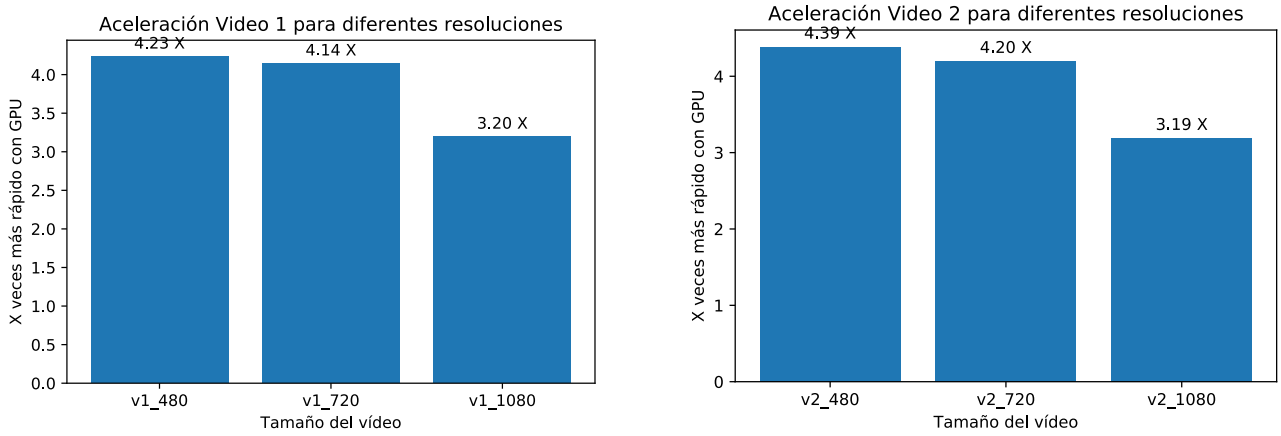


Figura 5.1: Tiempos de ejecución de los vídeos en diferentes resoluciones GPU vs CPU.



(a) Video generado por computador, Big Buck Bunny [9].

(b) Video 2, vehículos en movimiento [8].

Figura 5.2: Aceleración conseguida con GPU para ambos vídeos en diferentes resoluciones

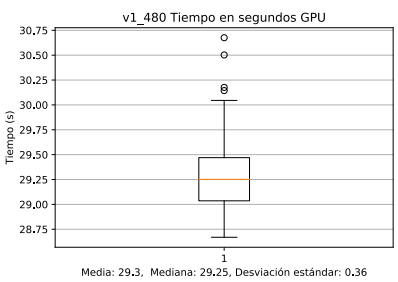
5.3. Resultados de tiempos

Una vez conocida la aceleración conseguida al usar la GPU con OpenCV para el cálculo, existen algunas evidencias más que se pueden extraer de los 100 experimentos en diferentes tamaños de fotogramas en ambos vídeos. Esta sección se analiza algunas medidas estadísticas para entender mejor los resultados.

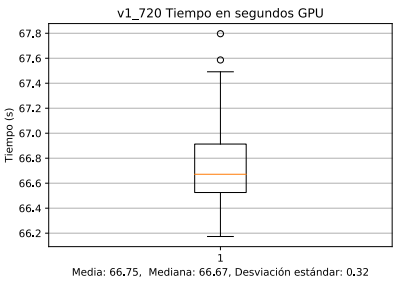
5.3.1. Diagramas de caja video 1 GPU y CPU

Los diagramas de caja son una la representación gráfica, basada en cuartiles, que ayuda a exhibir un conjunto de datos numéricos, la asimetría y los datos extremos con relación a la mediana [43].

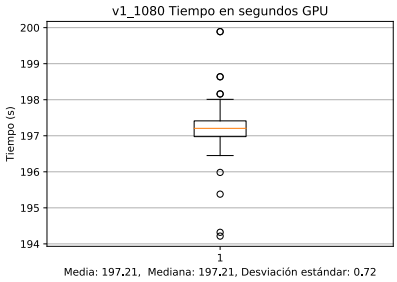
Por otra parte, para este análisis es importante saber la consistencia en el tiempo de procesamiento. Concretamente, en multimedia el tener varianzas pequeñas en los tiempos de ejecución puede llevar a salto



(a) 480x854

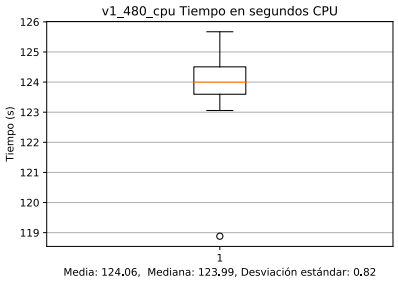


(b) 720x1080

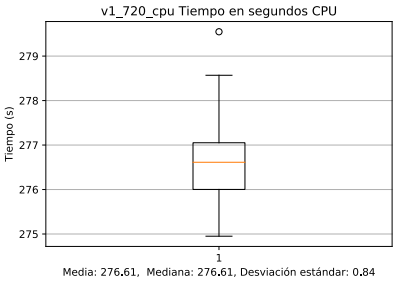


(c) 1080x1920

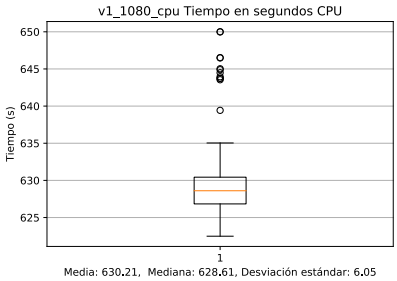
Figura 5.3: Promedia 100 iteraciones mediante la GPU en diferentes resoluciones.



(a) 480x854



(b) 720x1080



(c) 1080x1920

Figura 5.4: Promedio 100 iteraciones mediante la CPU en diferentes resoluciones.

de fotogramas o la sensación de ralentización en la interfaz de usuario. En la siguiente la Figura 5.3 se muestra la distribución de los tiempos totales de las iteraciones para el vídeo 1 en los diferentes tamaños de fotogramas en la GPU, y en la Figura 5.4 en la CPU.

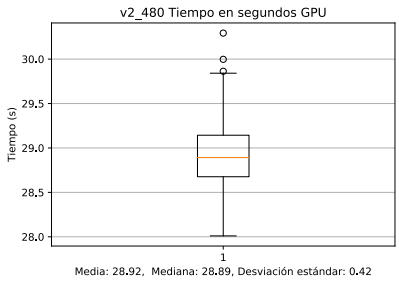
En la Tabla 5.1 se observa la gran diferencia en la simetría de los tiempos de ejecución al usar la GPU (la desviación estándar en la GPU es muy inferior a comparación a la CPU), lo que brinda la ventaja de tener consistencia en los tiempos de ejecución. Otra de las ventajas que existe al procesar en la GPU, es que al ser un dispositivo externo, se encarga específicamente de tareas o procesamiento dedicado, es decir, no será molestado por el resto del sistema. Incluso al contar con su propia memoria, no influyen las aplicaciones o procesos que se ejecuten en el Host.

5.3.2. Diagramas de caja video 2 GPU y CPU

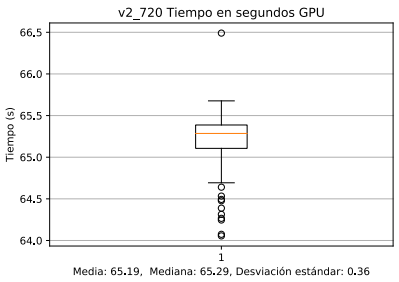
En la Figura 5.5 se observa la distribución de los datos generados del vídeo 2. Los diagramas de caja en el segundo vídeo son muy similares a los del vídeo 1. Y aunque los resultados entre ambos vídeo se parecen, si existe una pequeña variación.

Tabla 5.1: Desviación estándar vídeo 1.

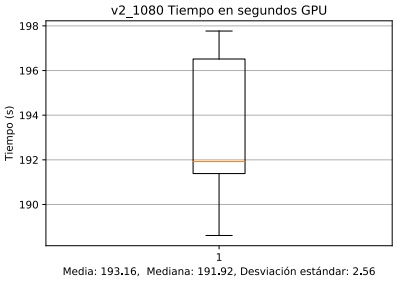
Resolución	Desviación estándar GPU	Desviación estándar CPU
480x854 pixeles	0,36	0,82
720x1080 pixeles	0.32	0,84
1920x1080 pixeles	0.72	6.05



(a) 480x854

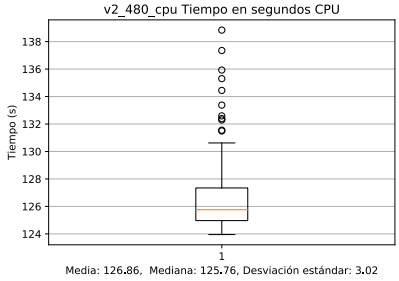


(b) 720x1080

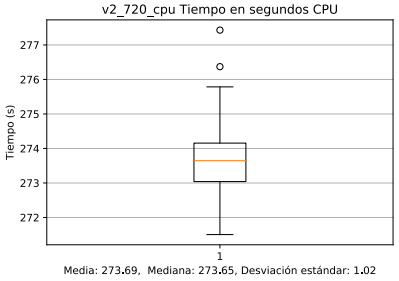


(c) 1080x1920

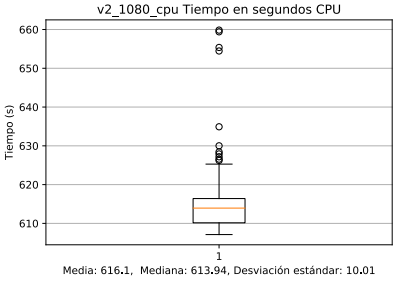
Figura 5.5: Promedio 100 iteraciones mediante la GPU en diferentes resoluciones.



(a) 480x854



(b) 720x1080

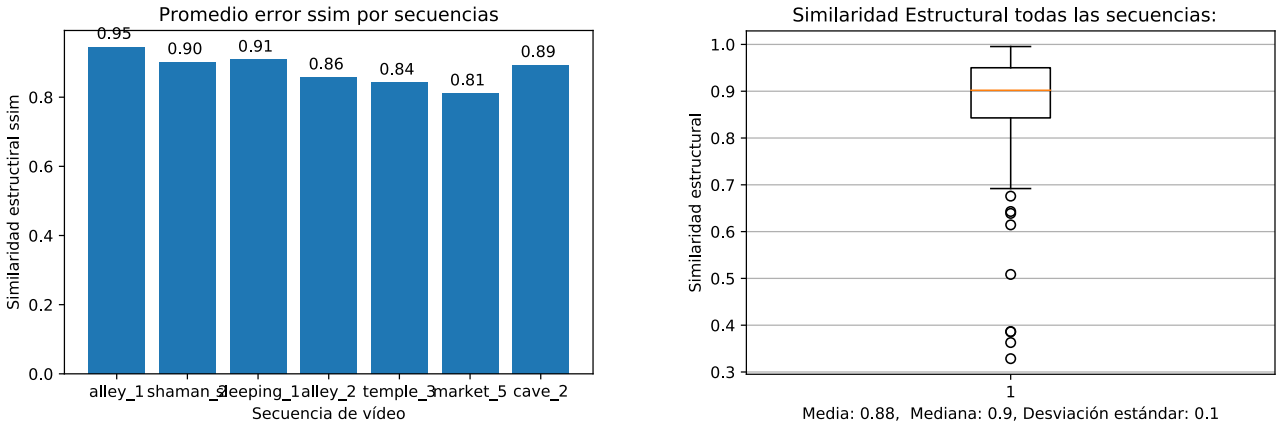


(c) 1080x1920

Figura 5.6: Promedio 100 iteraciones mediante la CPU en diferentes resoluciones.

Tabla 5.2: Desviación estándar vídeo 2.

Resolución	Desviación estándar GPU	Desviación estándar CPU
480x854 pixeles	0,42	3,02
720x1080 pixeles	0,36	1,02
1920x1080 pixeles	2,56	10,01



(a) SSIM promedio de cada secuencia.

(b) SSIM para todas las secuencias.

Figura 5.7: Similaridad estructural del flujo óptico Farneback vs real.

Tabla 5.3: Índice de similaridad estructural promedio de las secuencias.

	Alley 1	Shaman 2	Sleeping 1	Alley 2	Temple 3	Market 5	Cave 2
Desplazamiento	medio	medio	medio	medio-alto	alto	alto	alto
Similaridad estructural	0,95	0,90	0,91	0,86	0,84	0,81	0,89

Como se puede ver en la Tabla 5.2, una vez más la desviación estándar que existe entre la GPU ya la CPU es muy grande. Al usar la GPU para el cálculo del flujo óptico de imagen, se consigue mayor consistencia en los tiempos de ejecución.

Otra ventaja al usar la GPU, es que al momento de distribuir los recursos de todo el sistema, se reduce la carga que lleva la CPU. Los cálculos más complejos pueden realizarse en la GPU, mientras la CPU se encuentra encargada de coordinar en una relación maestro/esclavo.

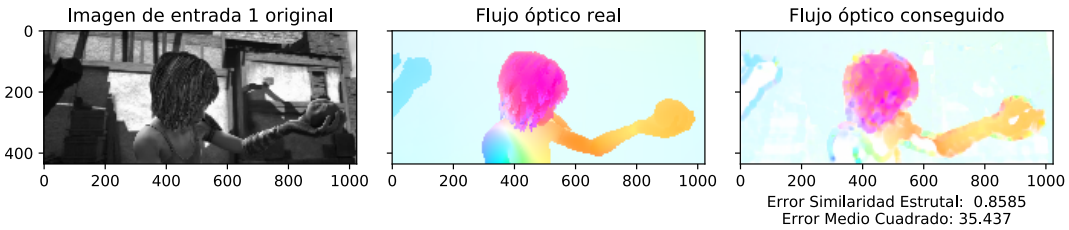
5.4. Similaridad estructural del flujo

Promedio evaluar el flujo óptico de imagen entre el real y el calculado mediante Farneback en las 7 secuencias del Dataset MPI-Sintel se observaron variaciones entre las diferentes secuencias. Ls 4 secuencias que presentan mayor movimiento entre fotorgramas, no son necesariamente las que mayor se asemejan (menor error). En la Figura 5.7 y en la Tabla 5.3 se pueden apreciar los promedios del índice de similaridad estructural SSIM para todas las secuencias.

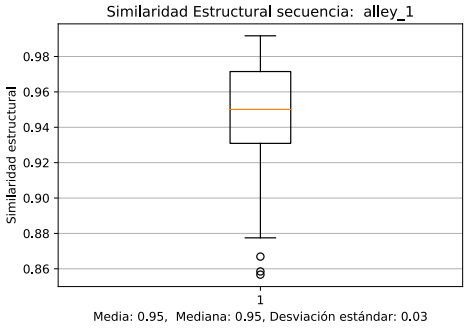
5.4.1. SSIM secuencia alley 1

EN la Figura 5.8 se muestra la similaridad estructural entre el flujo óptico real (Ground truth) vs el flujo denso Farneback CUDA en la secuencia alley 1. Esta secuencia no presenta mayor desplazamiento de los píxeles entre los fotogramas.

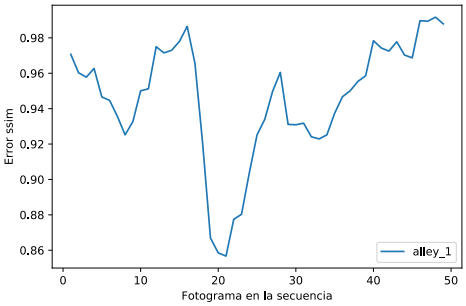
En la Figura 5.8 se aprecian los índices de similaridad estructural durante los 50 fotogramas. La calidad, dada por el índice SSIM de $0{,}95$ dentro del rango -1 a $+1$, es bastante próxima al flujo óptico real, y con una baja desviación estándar de $0{,}03$.



(a) Visualización del flujo coodificada en color.



(b) SSIM de la secuenica con 50 fotogramas.



(c) SSIM de los fotogramas de la secuencia.

Figura 5.8: Similaridad estructural del flujo óptico Farneback GPU alley 1.

5.4.2. SSIM secuencia shaman 2

En la Figura 5.9 se muestra la similaridad estructural entre el flujo óptico real vs el flujo óptico denso Farneback CUDA en la secuencia shaman 1, esta secuencia tampoco cuenta con una gran cantidad de desplazamiento/movimiento de píxeles entre fotogramas.

En la Figura 5.9 se aprecian los índices de similaridad estructural durante los 50 fotogramas. Laa calidad, dada por el índice SSIM de $0{,}9$ dentro del rango -1 a $+1$, es bastante próxima al flujo óptico real, y con una baja desviación estándar de $0{,}08$, aunque más alta que en la primera secuencia.

5.4.3. SSIM secuencia sleeping 1

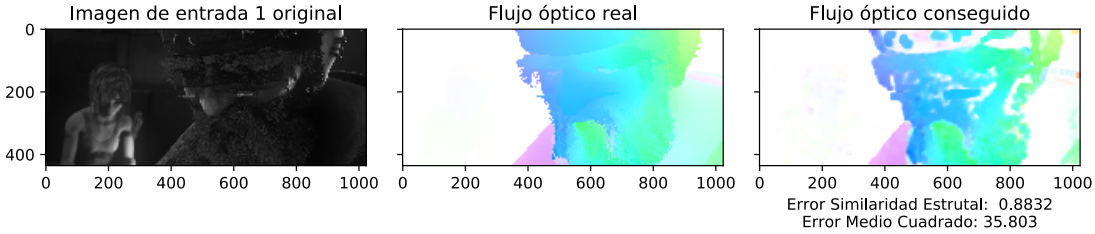
En la Figura 5.10 se muestra la similaridad estructural entre el flujo óptico real vs el flujo óptico denso Farneback CUDA en la secuencia sleeping 1, esta la última secuencia donde el movimiento/desplazamiento de los píxeles no es tan grande entre fotogramas.

En la Figura 5.10 se aprecian los índices de similaridad estructural durante los 50 fotogramas. La calidad, dada por el índice SSIM de $0{,}91$ dentro del rango -1 a $+1$, es bastante próxima al flujo óptico real, y con una baja desviación estándar de $0{,}03$, casi igual a la primera secuencia.

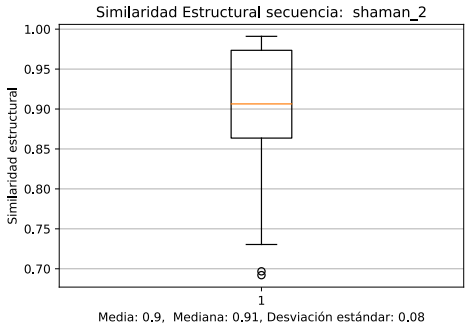
5.4.4. SSIM secuencia alley 2

En la Figura 5.11 se muestra la similaridad estructural entre el flujo óptico real vs el flujo óptico denso Farneback CUDA en la secuencia alley 2. A diferencia de la primera alley 1, esta secuencia presenta bastante movimiento/desplazamiento de los píxeles en porciones específicas de la imagen.

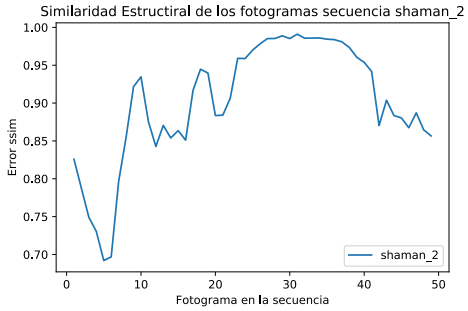
Esta secuencia tiene una media de $0{,}86$ en el índice de similaridad estructural, bastante más baja que las secuencias anteriores. También una desviación estándar mayor con $0{,}9$. Además es la que mayores



(a) Visualización del flujo coodificada en color.

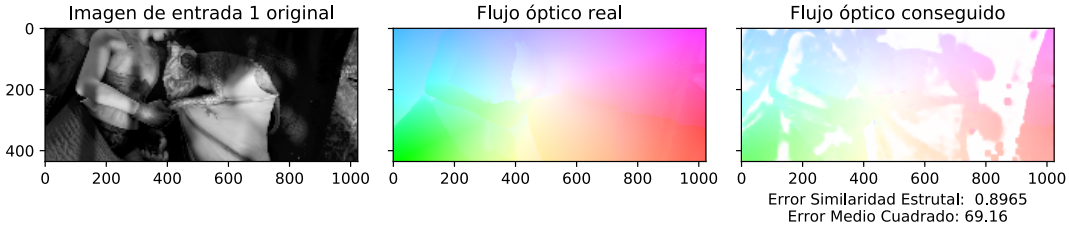


(b) SSIM de la secuenica con 50 fotogramas.

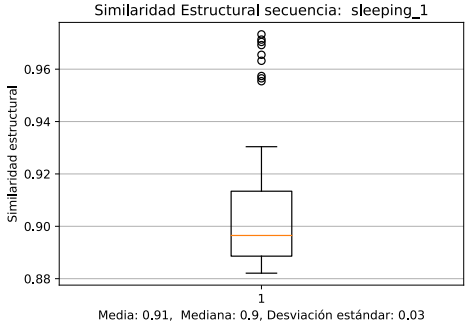


(c) SSIM de los fotogramas de la secuencia.

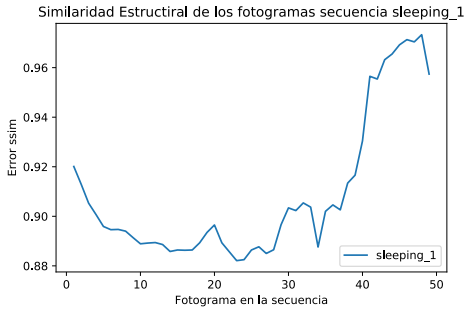
Figura 5.9: Similaridad estructural del flujo óptico Farneback GPU shaman 2.



(a) Visualización del flujo coodificada en color.

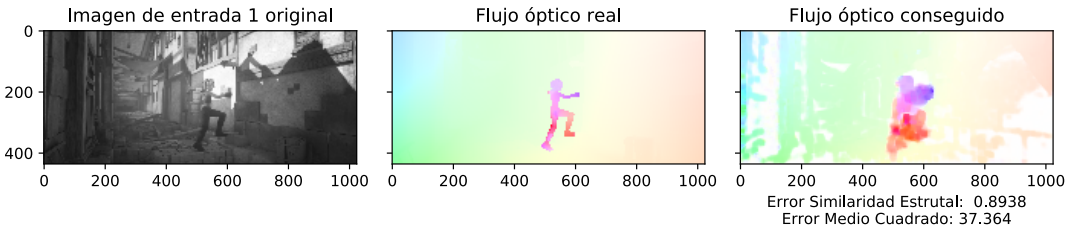


(b) SSIM de la secuenica con 50 fotogramas.

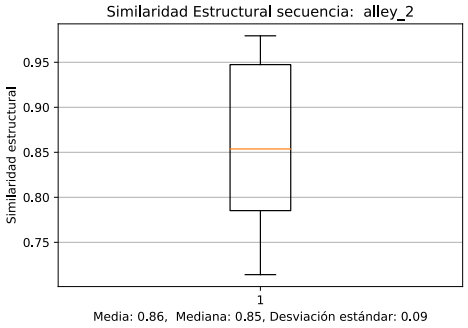


(c) SSIM de los fotogramas de la secuencia.

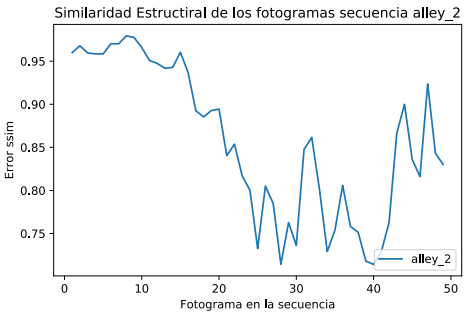
Figura 5.10: Similaridad estructural del flujo óptico Farneback GPU sleeping 1.



(a) Visualización del flujo coodificada en color.



(b) SSIM de la secuenica con 50 fotogramas.



(c) SSIM de los fotogramas de la secuencia.

Figura 5.11: Similaridad estructural del flujo óptico Farneback GPU alley 2.

variaciones existen entre las estimaciones (el cuartil inferior y superior son muy amplios). Claramente la cantidad de desplazamiento/movimiento de píxeles entre los fotogramas afectan a la calidad del cálculo de flujo óptico.

5.4.5. SSIM secuencia temple 3

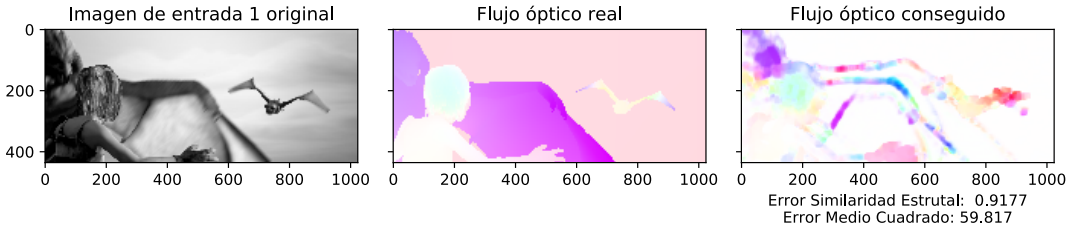
En la Figura 5.12 se muestra la similaridad estructural entre el flujo óptico real vs el flujo óptico denso Farneback CUDA en la secuencia temple 3, esta es la segunda con gran cantidad de movimiento/desplazamiento de píxeles entre fotogramas, esta secuencia presenta objetos en movimiento sobre un fondo con poco o nulo desplazamiento.

Una vez más, esta secuencia con bastante desplazamiento/movimiento entre los fotogramas tienen una menor calidad de estimación, e promedio $0{,}84$. Además la gran cantidad de movimiento en esta secuencia afecta negativamente a la calidad de la estimación.

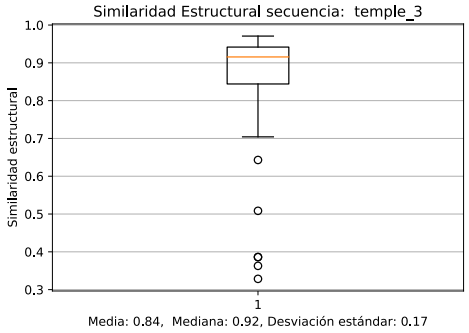
5.4.6. SSIM secuencia market 5

La Figura 5.13 muestra la similaridad estructural entre el flujo óptico real vs el flujo óptico denso Farneback CUDA en la secuencia market 5, esta es otra secuencia que presenta mucho desplazamiento/-movimiento de píxeles entre los fotogramas, existe movimiento en los varios objetos en la imagen, junto con el movimiento de cámara, por este efecto, existe movimiento de fondo en todo momento durante la escena.

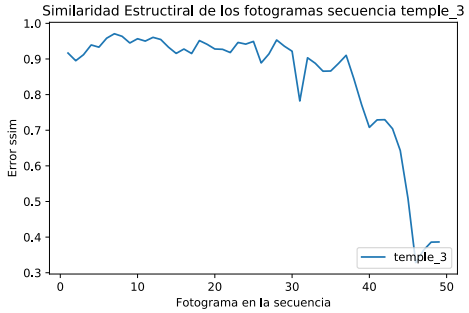
Una vez más se observa que la gran cantidad de desplazamiento de pixeles entre fotogramas afecta a la calidad conseguida en la estimación. De hecho es la secuencia con mayor cantidad de movimiento, y coincide con la menor calidad de estimación entre el resto de secuencias. Se consigue un $0{,}81$ de media,



(a) Visualización del flujo coodificada en color.

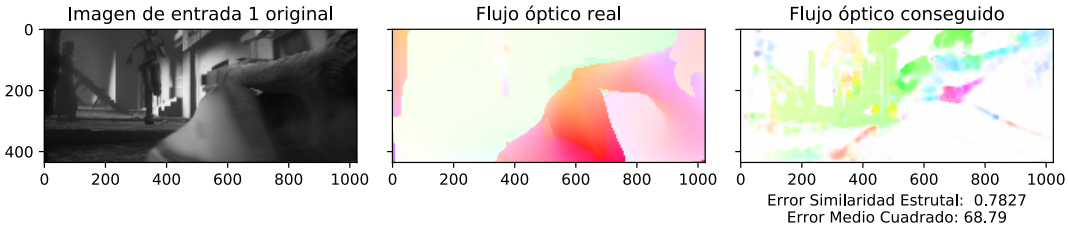


(b) SSIM de la secuenica con 50 fotogramas.

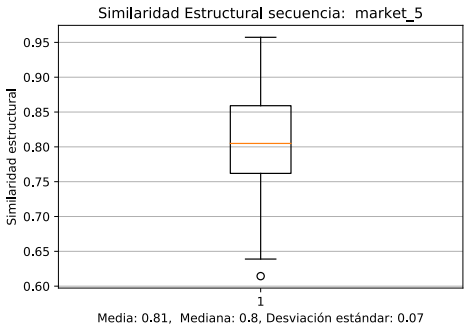


(c) SSIM de los fotogramas de la secuencia.

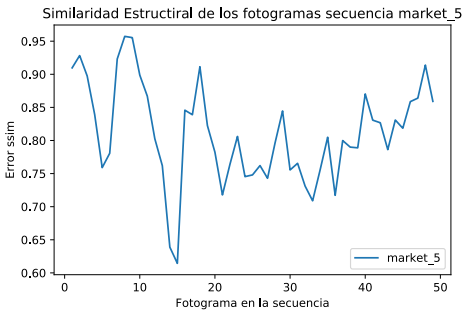
Figura 5.12: Similaridad estructural del flujo óptico Farneback GPU temple 3.



(a) Visualización del flujo coodificada en color.

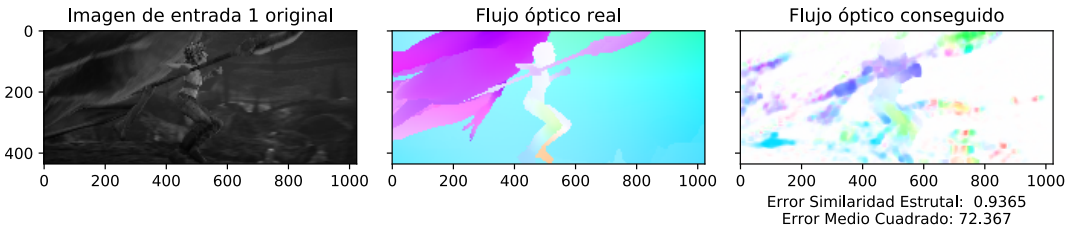


(b) SSIM de la secuenica con 50 fotogramas.

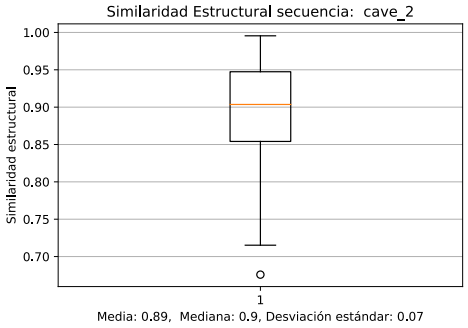


(c) SSIM de los fotogramas de la secuencia.

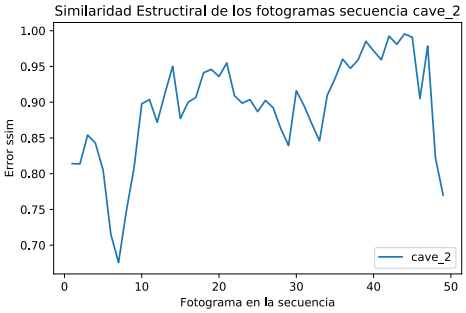
Figura 5.13: Similaridad estructural del flujo óptico Farneback GPU market 5.



(a) Visualización del flujo coodificada en color.



(b) SSIM de la secuenica con 50 fotogramas.



(c) SSIM de los fotogramas de la secuencia.

Figura 5.14: Similaridad estructural del flujo óptico Farneback GPU cave 2.

y desviación estándar $0{,}07$. Existen pocos datos que salen del cuartil inferior.

5.4.7. SSIM secuencia cave 2

En la Figura 5.14 se presenta la similaridad estructural entre el flujo óptico real vs el flujo óptico denso Farneback CUDA en la secuencia cave 2, esta es la última que se utilizó en este trabajo, también es una secuencia que presenta bastante desplazamiento en los objetos, desplazamiento de cámara y del fondo.

Se observa que una vez más que la secuencia tiene menor calidad de estimación en relación al resto de secuencias con menor desplazamiento/movimiento de píxeles, alcanzando un $0{,}89$ de índice de similaridad estructural, aunque con una desviación estándar de $0{,}07$. No presenta muchos valores extremos, los que se presentan están debajo del cuartil inferior.

5.5. Evaluación multi-plataforma

La posibilidad de portar el cálculo de flujo óptico denso de imagen a diferentes plataformas se ha evaluado durante el desarrollo de este trabajo. OpenCV permite la portabilidad, y aunque la mayoría de este trabajo se ha realizado con el lenguaje Python, la implementación en C++ (véase la Figura 4.2) brinda la posibilidad de utilizarlo en otras plataformas. La Jetson corre sobre un sistema GNU/Linux y con arquitectura ARMv8 x64 con una GPU CUDA. El mismo código podrá correr en un sistema Windows y el módulo Python detecta el sistema automáticamente.

Por otro lado, se ha comprobado también que puede realizarse el cálculo de flujo óptico de imagen para otras plataformas. La implementación que se ha propuesto funciona en el sistema operativo Android. Desde Kotlin se llamaron a funciones nativas en C++ de OpenCV que se puede compilar para dispositivos Android.

Finalmente, existe una gran limitación al momento de utilizar la computación acelerada propuesta. La compatibilidad está completamente ligada al uso de CUDA, una tecnología propietaria de Nvidia que no puede ser aprovechada en otras GPUs. Esta es una limitación muy grande sobre todo para plataformas móviles Android, ya que actualmente no existen plataformas móviles Android con GPUs CUDA. Nvidia sólo produce la familia de dispositivos Shield [56], que no están enfocados a computación de altas prestaciones ni cuentan con controladores o librerías CUDA.

Capítulo 6

Conclusiones

1. CUDA puede acelerar el cálculo de flujo denso óptico de imagen entre 300 % y 400 % en la Jetson TX2.

2. El hardware es crítico para el rendimiento deseado. De hecho, tarjetas modernas con más de 1000 núcleos CUDA pueden acelerar aún más los cálculos complejos.

3. El usar procesamiento en la GPU permite bajar la carga que recae sobre la CPU, que pasa a segundo plano a modo de coordinador de las acciones de la GPU en un esquema maestro/esclavo. Pero a su vez, añade complejidad al programador, tener en mete el flujo de datos entre memoria de la CPU y la GPU.

4. OpenCV gracias al código C++ permite compilar a una gran variedad de dispositivos. Gracias a esa flexibilidad se puede correr código desarrollado en este trabajo tanto en Windows, Linux, y Android.

5. El tiempo que toma calcular el flujo óptico denso crece considerablemente con el tamaño de los fotogramas.

6. Existen estudios para mejorar la calidad del flujo óptico de imagen mediante redes neuronales, pero requieren de cantidades enormes de datos correctamente etiquetados y entrenamiento sobre los datos, lo que es imposible de entrenar en la Jetson.

7. CUDA es una arquitectura y framework propietaria, es decir, sólamente se puede sacar provecho al tener el hardware necesario de una compañía, que en la mayoría de casos brinda rendimiento superior a otras tecnologías heterogéneas como OpenCL.

8. La arquitectura CUDA presenta muchas ventajas para el procesamiento de imágenes o el tratamientos de señales, es un entorno muy maduro y con una biblioteca de alto nivel que permite al programador interactuar directamente con la tarjeta gráfica.

Bibliografía

[1] Nvidia cuda. https://developer.nvidia.com/cuda-zone, Accedido en 2019-12-14.

[2] Big buck bunny video. https://peach.blender.org/about/, Accedido en 2020-01-28.

[3] Module: tf | TensorFlow Core r1.15, Accedido en 2020-01-28.

[4] AmirHessam Aminfar, Nami Davoodzadeh, Guillermo Aguilar, and Marko Princevac. Application of optical flow algorithms to laser speckle imaging. *Microvascular research*, 122:52–59, 2019.

[5] Oracle and/or its affiliates. Java native interface overview, Accedido en 2020-02-08.

[6] Brandon Barker. Message passing interface (mpi). In *Workshop: High Performance Computing on Stampede*, volume 262, 2015.

[7] Steven S. Beauchemin and John L. Barron. The computation of optical flow. *ACM computing surveys (CSUR)*, 27(3):433–466, 1995.

[8] Josef Bigün, Goesta H. Granlund, and Johan Wiklund. Multidimensional orientation estimation with applications to texture analysis and optical flow. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (8):775–790, 1991.

[9] Barney Blaise. Message passing interface (mpi), Accedido en 2020-02-09.

[10] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[11] Andrés Bruhn, Joachim Weickert, and Christoph Schnörr. Combining the advantages of local and global optic flow methods. In *Joint Pattern Recognition Symposium*, pages 454–462. Springer, 2002.

[12] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A naturalistic open source movie for optical flow evaluation. In A. Fitzgibbon et al. (Eds.), editor, *European Conf. on Computer Vision (ECCV)*, Part IV, LNCS 7577, pages 611–625. Springer-Verlag, October 2012.

[13] Ivan Castro and Francesc Bernat. *Introducción a la computación de altas prestaciones*. Universitat Oberta de Catalunya, 2014.

[14] Sumohana S Channappayya, Alan C Bovik, Constantine Caramanis, and Robert W Heath. Ssim-optimal linear image restoration. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 765–768. IEEE, 2008.

[15] Darren M Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1566–1573, 2007.

[16] NVIDIA Corporation. Using CUDA Warp-Level Primitives, January Accedido en 2020-01-30.

[17] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[18] Universidad de Almería. UAL Cloud - Universidad de Almería, 2019.

[19] Android Developers. Android ndk, Accedido en 2020-02-08.

[20] Intel Corporation. OpenCV Foundation. NVIDIA Corporation. Advanced Micro Devices. opencv contrib, Accedido en 2020-02-08.

[21] Leslie Lamport et al. The LaTeXProject. https://www.latex-project.org, Accedido en 2019-11-12.

[22] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[23] Kotlin Foundation. Kotlin for android - kotlin programming language, Accedido en 2020-02-08.

[24] Python Software Foundation. Python about. https://www.python.org/about/, Accedido en 2019-12-16.

[25] Python Software Foundation. The python tutorial. https://docs.python.org/3/tutorial/index.html, Accedido en 2019-12-16.

[26] Vicente González-Ruiz. Motion Compensated Discrete Wavelet Transform (MCDWT).

[27] Google, Accedido en 2020-01-30.

[28] The Khronos Group, Jul 2013.

[29] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2462–2470, 2017.

[30] Google Inc. Colaboratory. https://colab.research.google.com, Accedido en 2019-11-12.

[31] Plotly Technologies Inc. Collaborative data science, 2015.

[32] Project Jupyter. Jupyter notebook. https://jupyter.org, Accedido en 2019-11-12.

[33] Project Jupyter, Accedido en 2020-01-30.

[34] Matthias Kramer and Hubert Chanson. Optical flow estimations in aerated spillway flows: Filtering and discussion on sampling parameters. *Experimental Thermal and Fluid Science*, 103:318–328, 2019.

[35] Jeff Larkin. Gpu fundamentals. Nov 2016.

[36] Douglas A Lind, William G Marchal, and Samuel A Wathen. *Estadística aplicada a los negocios y la economía*. McGraw-Hill,, 2005.

[37] Arm Ltd. Simd isas neon.

[38] Canonical Ltd. Ubuntu. https://ubuntu.com, Accedido en 2019-11-12.

[39] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. 1981.

[40] Karol Majek. 4K Road traffic video for object detection and tracking - free, Accedido en 2020-01-28.

[41] MartínAbadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[42] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[43] Tom McReynolds and David Blythe. *Advanced graphics programming using OpenGL*. Elsevier, 2005.

[44] Alexander Mordvintsev and K. Abid. *Optical Flow — OpenCV-Python Tutorials 1 documentation*.

[45] H-H Nagel and A Gehrke. Spatiotemporally adaptive estimation and segmentation of of-fields. In *European Conference on Computer Vision*, pages 86–102. Springer, 1998.

[46] Nvidia. Programming guidecuda toolkit documentation.

[47] Nvidia. An introduction to cuda-aware mpi, Mar 2013.

[48] Nvidia. Mpi solutions for gpus, May 2013.

[49] Nvidia. Nvidia cuda achitecture. Nov 2018.

[50] Nvidia. Nvidia jetson. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems, Accedido en 2019-11-12.

[51] Nvidia. Nvidia jetson. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2, Accedido en 2019-11-12.

[52] Nvidia, Accedido en 2020-02-08.

[53] The Open MPI Project. Open mpi: Open source high performance computing, Accedido en 2020-02-09.

[54] Florian Raudies. Optic flow. *Scholarpedia*, 8(7):30724, Jul 2013.

[55] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming, portable documents*. Addison-Wesley Professional, 2010.

[56] Tal Schuster, Lior Wolf, and David Gadot. Optical flow requires multiple strategies (but only one network). In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4950–4959, 2017.

[57] Jacob Søgaard, Lukáš Krasula, Muhammad Shahid, Dogancan Temel, Kjell Brunnström, and Manzoor Razaak. Applicability of existing objective metrics of perceptual quality for adaptive video streaming. *Electronic Imaging*, 2016(13):1–7, 2016.

[58] Deqing Sun, Xiaodong Yang, Ming-Yu Liu, and Jan Kautz. PWC-Net: CNNs for optical flow using pyramid, warping, and cost volume. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8934–8943, 2018.

[59] Deqing Sun, Xiaodong Yang, Ming-Yu Liu, and Jan Kautz. Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8934–8943, 2018.

[60] OpenCV team. Opencv cuda. https://opencv.org/platforms, Accedido en 2019-12-14.

[61] OpenCV team. Opencv platforms. https://opencv.org/platforms, Accedido en 2019-12-14.

[62] tim.lewis. Openmp.

[63] Fernando G Tinetti. High performance cluster computing: Architectures and systems, vol. 1. *Journal of Computer Science & Technology*, pages 51–52, 2002.

[64] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[65] Guido van Rossum et al. Python. https://www.python.org, Accedido en 2019-11-12.

[66] Todd Veldhuizen. Measures of image quality.

[67] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[68] Zhou Wang and Qiang Li. Information content weighting for perceptual image quality assessment. *IEEE Transactions on image processing*, 20(5):1185–1198, 2010.

[69] Wikipedia. https://en.wikipedia.org/wiki/Graphics_processing_unit, Accedido en 2019-11-12.

[70] Wikipedia. Cpu. https://en.wikipedia.org/wiki/Central_processing_unit, Accedido en 2019-11-12.

[71] Wikipedia, Accedido en 2020-02-09. Page Version ID: 935661907.

[72] Christopher Zach, Thomas Pock, and Horst Bischof. A duality based approach for realtime tv-l 1 optical flow. In *Joint pattern recognition symposium*, pages 214–223. Springer, 2007.

Resumen/Abstract

El flujo óptico de imagen es muy usado en aplicaciones multimedia, visión computacional, codificación/decodificación de vídeo, sin embargo, tradicionalmente es un cálculo que consume muchos recursos de hardware y tiempo. Este trabajo presenta formas de acelerar el flujo óptico denso de imagen entre fotogramas aprovechando la arquitectura CUDA de las tarjetas gráficas de Nvidia mediante OpenCV. También se evalúa la calidad del flujo óptico denso de imagen para varias secuencias, cada una con diferente grado de movimiento de píxeles entre fotogramas, y se diiscutien otras formas de calcular el flujo óptico del estado de arte. Además se evalúa la posibilidad de realizar el mismo cálculo en diferentes plataformas, Windows, Linux, Android gracias a la potabilidad del código Python y C++. Finalmente se integra en el códec MCDWT el código acelerado en la GPU, de una forma trasparente y multi-plataforma, que se activará automáticamente al detectar un sistema CUDA.

⋆ ⋆ ⋆

Optical image flow is widely used in multimedia applications, computer vision, video coding/decoding. However, it is traditionally a calculation that consumes many hardware resources and time. This project presents ways to accelerate the dense optical flow of image between frames taking advantage of the CUDA architecture of Nvidia graphics cards through OpenCV. The quality of the dense optical obtained is also evaluated for several sequences, each one with different degree of pixel movement between frames, also discussing other ways to calculate the optical flow in the state of the art. In addition, the possibility of performing the same calculation on different platforms, Windows, Linux, Android is evaluated thanks to the portability of Python and C ++ code on other platforms. Finally, the accelerated code in the GPU is integrated into the MCDWT codec, in a transparent and multi-platform way, which will be activated automatically if a CUDA system is detected.