Máster universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Optimización de una librería de alto rendimiento multiplataforma para procesamiento de imagen

Ricardo Oliva García

Director: Pedro Javier Herrera Caro

Ingeniería de Software (31105109)
Universidad Nacional de Educación a Distancia

Curso: 2017 / 2018
Convocatoria: Junio 2018

Máster universitario de Investigación en Ingeniería de Software y Sistemas informáticos

Optimización de una librería de alto rendimiento multiplataforma para procesamiento de imagen

Ingeniería de Software (31105109)

Tipo de trabajo B

Ricardo Oliva García
Director: Pedro Javier Herrera Caro

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 18/06/2018

Quién suscribe:

Autor(a):Ricardo Oliva García
D.N.I/N.I.E/Pasaporte.:45865266T

Hace constar que es la autor(a) del trabajo:

Optimización de una librería de alto rendimiento multiplataforma para procesamiento de imagen

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo.

Ricardo

IMPRESO TFDM05_AUTOR
AUTORIZACIÓN DE PUBLICACIÓN
CON FINES ACADÉMICOS

ETS de Ingeniería Informática

Impreso TFdM05_Autor. Autorización de publicación y difusión del TFdM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Resumen

Este Trabajo Fin de Máster propone la optimización de determinadas operaciones sobre una librería de procesamiento de imagen, que permite tanto en labores de investigación como en avance tecnológico, el desarrollo simple y rápido de aplicaciones relacionadas con visión por computador.

Hoy en día, este tipo de aplicaciones se utilizan en multitud de campos. La idea es proveer de una librería que facilite el desarrollo tanto en investigación como en aplicaciones. La complejidad de este trabajo radica en optimizar las funciones principales de procesamiento de imagen en múltiples plataformas Hardware, de modo que sin necesidad de programar a bajo nivel, ni para un dispositivo específico se pueda obtener alto rendimiento. Para la elaboración se utilizaron herramientas específicas de múltiples arquitecturas, como pueden ser juegos de instrucciones SIMD en dispositivos INTEL (SSE, SSE2, etc.), o en arquitecturas ARM (NEON), además de utilizar el lenguaje de OpenCL para facilitar el desarrollo en paralelo en múltiples dispositivos.

La intención final del trabajo es que pueda incorporarse en distintas plataformas, disponiendo de funcionalidad en modo grafo para la ejecución en sistemas autónomos, además de dar un rendimiento óptimo y tener un consumo reducido de recursos.

La elección de las distintas funciones para optimizar se escogen en base al consumo computacional de las mismas, inclusive si se requiere su uso de forma constante, como puede ser la corrección de distorsión. Durante la elaboración del trabajo se realizan comparaciones con la librería OpenCV debido a su alto rendimiento y a que es un estándar de uso en el desarrollo de aplicaciones que contengan procesamiento de imagen. Se presentan múltiples optimizaciones tanto de operaciones específicas como generales, en este caso se escogen las siguientes: convoluciones, filtro de caja, filtro bilateral y correcciones de distorsión.

Palabras clave: Procesamiento de imagen, convolución, filtros, segmentación, calibración, diseño.

Abstract

This Master's degree final Project purpose is optimizations over an image processing library, it's allows in research and IT, fast develop of applications related with computer vision.

Nowadays, these applications it's used in a lot of fields. The main idea is to provide an easy library, that permits develop in research and applications. The principal trouble at this job is optimize the main functions at image processing, so without develop at low level, can have high performance. To work out these troubles, uses specifics tools such SIMD instructions at Intel devices, or NEON in ARM, also uses standar languages to facilitate parallel develop in different devices.

The final objective of this job is to incorporate in multiple platform, providing graph mode functionality in order to execute in autonomous systems, also gives high perform, and have low resources consumption.

The choice of the different functions to optimize are chosen based on the computational consumption, or in the high use of them, such as distortion correction. During work elaboration, it's realice comparations between OpenCV and the mentioned library, OpenCV is chosen in base to it's high perfom, and because is an develop standar in computer vision applications. Multiple optimizations are presented, both specific and general, the optimizations chosen are the next: standar convolutions, box filters, bilateral filter and distortion corrections.

Keywords: Image processing, convolution, filters, segmentation, calibration, design.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El avance en tecnología demanda aplicaciones en tiempo real, y la comodidad de usuario es uno de los factores más importantes a la hora de poner a su disposición un producto. Esto requiere implementaciones rápidas de algoritmos, además, surge la necesidad de crear múltiples versiones para la ejecución en distintos dispositivos, dado que en la actualidad se dispone de distintas herramientas de uso tecnológico diario (móviles, PCs, PDAs, ...), por ello se demandan librerías que contengan la implementación de algoritmos y su rápida ejecución para distintos dispositivos.

La visión por computador está presente en múltiples ámbitos, desde la fotografía móvil, hasta la extracción de datos de imágenes espaciales, médicas, etc. La demanda de los usuarios implica un incremento de la calidad de resultados, además de un aumento considerable de la velocidad de ejecución de las aplicaciones.

Existen librerías que pretenden reducir el coste computacional de los algoritmos, pero en dispositivos de Hardware limitado, esta tarea no es trivial, lo que implica que el desarrollo de aplicaciones con necesidad de consumir un alto porcentaje de recursos se elaboren en las propias compañías de desarrollo de la arquitectura o el dispositivo, es decir, las empresas desarrollan las aplicaciones del dispositivo, impidiendo a otras empresas no fabricantes de Hardware proveer aplicaciones con alto consumo computacional.

La finalidad de este trabajo es el desarrollo de una herramienta sencilla de instalar en cualquier dispositivo, desde PCs, móviles, microcontroladores, etc. Esta tarea implica realizar un diseño complejo que permita incluir múltiples tecnologías, y permitir dependiendo del dispositivo el uso de operaciones específicas. Se utiliza una amplia gama de herramientas para optimizar el código, desde instrucciones específicas de cada una de las arquitecturas, hasta el uso de lenguajes estándar para aprovechar el paralelismo de los dispositivos [1]. Las arquitecturas sobre las que se hace más hincapié son NVIDIA, INTEL, ARM y ATI. Para poder realizar la compilación en múltiples arquitecturas es necesario configurar cada una de estas acorde al dispositivo y las librerías que este disponga, por ello se utiliza una herramienta de compilación para configurar de modo automático las opciones del compilador (CMake) [2].

Una de las causas de utilizar una librería de desarrollo es la comodidad y ahorro de tiempo de implementación. Esta tarea se pone en práctica para la elaboración de algoritmos, pero en el caso de tener limitaciones de tiempo o Hardware, no es factible el uso de una librería externa, ya que no necesariamente disponen de una implementación en código nativo para arquitecturas concretas. Si se dispone de una herramienta que facilite este trabajo se puede dedicar mayor tiempo a labores de investigación, u optimización matemática de los algoritmos principales, obviando la preocupación de las implementaciones a más bajo nivel, lo que por lo general se obtienen mejores resultados en la fase de desarrollo.

La optimización de determinadas funciones no implica únicamente la adaptación para distintas arquitecturas, es necesario realizar profundas investigaciones para obtener mejoras matemáticas o diseñar formas distintas de abordar el problema llegando a la misma solución o similar, pero consumiendo menor cantidad de recursos. Este trabajo se centra en determinadas funciones computacionalmente complejas en el ámbito de visión artificial, para investigar en técnicas de optimización algorítmica, con el propósito de obtener un aumento en rendimiento sin la necesidad de realizar optimizaciones para Hardware dedicado, además de acelerar los algoritmos a posteriori con este tipo de técnicas.

La motivación de desarrollo de esta librería viene dada por el uso actual de la fotografía computacional, el campo de la imagen está presente en grandes campos de estudio, siendo compleja la ejecución simple y rápida de las técnicas que presenta. Múltiples aplicaciones que contemplan el tratamiento de imagen disponen únicamente de la versión para un sistema operativo, difiriendo los tiempos de ejecución conforme a la arquitectura del procesador.

El objetivo principal del trabajo es subsanar la problemática de tener que desarrollar código específico para cada plataforma disponible, permitiendo la divulgación y uso de programas en distintos dispositivos sin necesidad de dedicar grandes esfuerzos en disponer de portabilidad, ya que en sí el desarrollo de aplicaciones, creación de interfaces gráficas, conexiones con dispositivos, etc; supone un trabajo complejo.

Se pretende mejorar los tiempos de ejecución de librerías del estado del arte actual, sin añadir complejidad extra de desarrollo, teniendo como objetivo principal los dispositivos móviles, debido a que el Software de cámara es uno de los principales motivos de venta y aún las prestaciones Hardware de la que disponen no contienen suficiente potencia para ejecutar algoritmos rápidos, debido al consumo de batería.

Se proponen los siguientes retos:

- Permitir la instalación en múltiples plataformas.
- Realizar la mejora de distintos algoritmos sin ser dicha mejora dependiente del Hardware.
- Disminuir los tiempos de cómputo en distintas arquitecturas.
- Reducir la complejidad computacional aprovechando las arquitecturas de GPU en los móviles, ofreciendo mejor usabilidad en técnicas de visión.

Este documento se organiza como sigue:

- Capítulo 2: presenta un estado del arte de las librerías actuales en el ámbito de la fotografía computacional, además de una comparación entre ellas, y una serie de conceptos básicos introductorios de la materia a abordar.
- Capítulo 3: este capítulo se centra en el desarrollo de la solución propuesta, planteando el diseño de software, y las implementaciones de mayor aporte en investigación explicando las técnicas de desarrollo utilizadas para optimizar los algoritmos.
- Capítulo 4: la comparativa frente a las distintas librerías del estado del arte se presentan en este capítulo, además de las mejoras que han sucedido durante el transcurso del trabajo.
- Capítulo 5: se presentan las conclusiones y posibles líneas futuras de este trabajo.

Capítulo 2

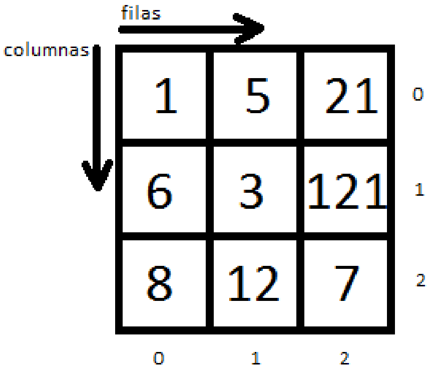
Fundamentación teórica y Estado del arte

Este capítulo aborda a modo introductorio una serie de conceptos necesarios para el seguimiento del trabajo y la comprensión de los distintos métodos desarrollados, dando a conocer conceptos básicos de visión por computador, además de presentar las diferentes librerías existentes en el estado del arte y algunas comparaciones entre las mismas.

2.1. Fundamentos teóricos

El campo de estudio de visión por computador alude a la comprensión de alto nivel de imágenes tanto digitales como analógicas. Las imágenes visualizadas en una pantalla digital son internamente representadas de forma numérica como una matriz 2D, en la que cada posición contempla el color, descrito por uno o varios canales (Gris, RGB, ...), en la figura 2.1 se observa como se indexa y se accede a una matriz, que a diferencia de las coordenadas cartesianas, se utiliza como origen la esquina superior izquierda.

El estudio en procesamiento de imagen [3], se aplica en multitud de campos como se ha explicado anteriormente, permite el análisis de imágenes, como de la secuencia de las mismas (vídeos). La temática a abordar presenta multitud de algoritmos, es de peculiar atención debido al interés social la reconstrucción 3D [4] de escenas grabadas con cámaras 2D, esto se debe a la naturaleza del ser humano ya que visualiza en 3D.



1	5	21	0
6	3	121	1
8	12	7	2
0	1	2	

Figura 2.1 Representación de una imagen a modo numérico

La complejidad y amplitud del campo de estudio define dentro del mismo una serie de módulos, que separan las distintas temáticas abordadas en el tratamiento de imagen, en los que cabe destacar los siguientes:

- Espacios de color.
- Calibrado y corrección de imagen.
- Umbralizado.
- Operaciones de filtrado.
- Segmentación.
- Flujo óptico.
- Estructura del movimiento.

La memoria redactada incorpora la optimización tanto matemática, como a nivel de Hardware de algunas de las técnicas que se incluyen en los módulos nombrados.

Este capítulo prosigue con el resumen de cada uno de los módulos, con el objetivo de facilitar la comprensión de las técnicas que se abordan en el trabajo, para su optimización o mejora, además de proveer una serie de conceptos necesarios para el tratamiento de imágenes.

2.1.1. Espacios de color

Los espacios de color son de gran importancia en el ámbito de la imagen, estos ofrecen tanto una representación visual como analítica de la matriz de píxeles, es decir, no sólo ayuda a su correcta visualización, si no además permiten la adaptación de dicha representación facilitando matemáticamente las operaciones con imágenes. Existen múltiples espacios adaptados para las distintas situaciones, dando cada uno, un manejo de los datos exclusivo para la aplicación de distintas técnicas sobre la imagen [5].

Las pantallas utilizadas para digitalizar una imagen representan el color de forma distinta, dependiendo de la electrónica que dispongan, al igual que las impresoras.

El espacio de color utilizado para la representación visual es el RGB, esto se debe a la franja visual que abarca el ojo humano para detectar las frecuencias de cada uno de los colores, en este caso la representación de la matriz contiene 3 valores por cada posición, correspondientes a cada uno de los colores, de forma analítica no es usual el trabajo con imágenes en RGB, debido a la complejidad añadida subyacente del trabajo con 3 valores en lugar de 1 (escala de grises) por cada píxel de la imagen.

A nivel analítico, un espacio de color de uso común y curioso por sus cualidades es el HSV, ya que representa las características visuales del ojo humano, por ejemplo, numéricamente en el espacio de color RGB, es complicado encontrar la similitud entre distintos colores, a diferencia que ocurre con HSV. Este se divide en 3 canales, brillo, matiz y saturación, teniendo el matiz la componente de color, y dicha representa numéricamente lo que el ojo humano observa como colores similares, utilizando los otros dos canales como representación de luminosidad y dureza del color. En la figura 2.2 se observa el paso de una imagen RGB a HSV con sus canales reflejados por separado para reflejar el aporte de cada uno.

2.1.2. Calibrado y corrección de imagen

Las cámaras presentan una parte importante en el mundo de la visión por computador. Estas se utilizan desde antaño para la captura de escenas reales, con el fin de obtener un almacenamiento ya sea físico o digital. Las cámaras compuestas por lentes tienen una serie de características que no siempre favorecen la captura de la imagen. Esta sección aborda los inconvenientes producidos por la construcción de lentes, ya sea debido al precio o calidad de las mismas, a este campo de estudio se le conoce como calibración de cámara [6].

El calibrado de cámaras se utiliza para determinar los parámetros intrínsecos y extrínsecos de la cámara, los parámetros intrínsecos determinan la relación entre la unidad de la cámara y las unidades del mundo real, estos parámetros se definen como se observa en la ecuación 2.1. Por otro lado, están los parámetros extrínsecos de la cámara que definen dado un sistema de

Figura 2.2 Conversión de color RGB a HSV, arriba se presentan los canales por separado H-S-V consecutivamente y abajo la imagen RGB entrante

coordenadas los movimientos realizados de rotación y traslación entre múltiples imágenes.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

donde f_x es el tamaño de la focal en milímetros en el eje x, f_y es el tamaño de la focal en milímetros en el eje y, y c_x, c_y los centros de la imagen en los ejes x e y respectivamente.

Las aberraciones más comunes en el trato con lentes son cromáticas y geométricas, siendo estas últimas de particular atención. Se contemplan en las aberraciones geométricas dos factores, el radial y el tangencial, los parámetros que definen estas aberraciones se pueden calcular utilizando como base un ajedrez [7] del cual se conocen las líneas paralelas que separan las casillas como se puede observar en la figura 2.3. En la ecuación 2.2 se observan los parámetros que deben hallarse para la corrección radial,

$$\begin{aligned} x_c &= x(1+k_1r^2+k_2r^4+k_3r^6) \\ y_c &= x(1+k_1r^2+k_2r^4+k_3r^6) \end{aligned} \tag{2.2}$$

donde los factores k_i son conocidos como parámetros de distorsión y r representa el radio de corrección.

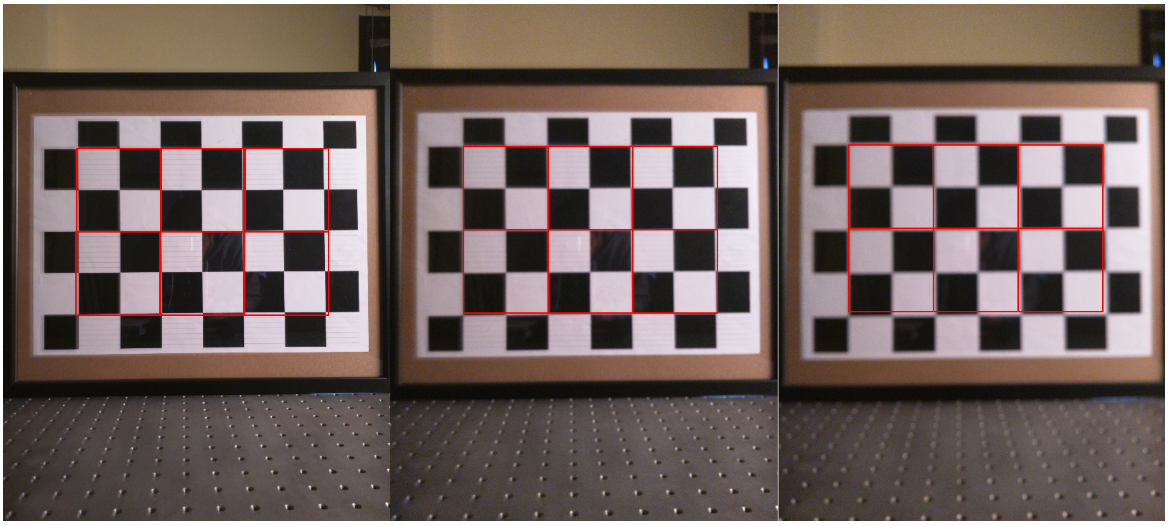


Figura 2.3 Representación de distorsiones geométricas de una misma imagen con distintos enfoques

2.1.3. Umbralizado

En el trabajo rutinario con imágenes surge la necesidad de modificar los colores de la imagen para separar las partes de una imagen que sean objetivo de las que no, a estos métodos se les conoce como umbralizado, existen métodos en los que el umbral se escoge tanto de forma manual como automática [8]. En la figura 2.4 se observa una imagen en escala de grises a la izquierda, y a su derecha la imagen aplicando un umbral de 50, dejando a blanco los valores que superan dicho umbral y a negro los que no.

2.1.4. Operaciones de filtrado

Las técnicas de filtrado permiten operar en una imagen de forma espacial, es decir, permite utilizar la información de un píxeles con sus vecinos, este tipo de operaciones son muy demandadas por cualquier tipo de aplicación, programa o procesamiento básico de

Figura 2.4 Imagen umbralizada con valor 50

imagen. Para dar una contribución desde un píxel local a sus vecinos se utiliza una máscara de tamaño θ, esta se superpone por cada píxel de la imagen, dando como resultado la suma de la multiplicación de cada valor de la máscara por su píxel correspondiente como se puede observar en la ecuación 2.3.

$$I(x,y) * f = \sum_{i=0}^{\theta} \sum_{j=0}^{\theta} I(x+i-\theta/2, y+j-\theta/2) f(i,j) \tag{2.3}$$

donde θ es el tamaño de ventana y x e y las posiciones de un píxel concreto. Las convoluciones son objeto de estudio y optimización dada su complejidad computacional.

2.1.5. Segmentación

Las técnicas para la división de imágenes en múltiples objetos se conocen como segmentación, este proceso trata de realizar una descomposición “lógica” desde la perspectiva humana, tratando de obtener las distintas partes de una imagen separadas por un mismo con color.

Estas técnicas ya hacen uso de algunas comentadas anteriormente como las de umbralizado y filtrado, y se aplican en múltiples campos, teniendo especial importancia en medicina.

Un ejemplo de algoritmo que realiza la separación de una imagen por agrupamiento de píxeles es la técnica “Superpixels” [9], realizando esta una segmentación por vecindad de color.

2.1.6. Flujo óptico

Al igual que el tratamiento de imagen, en el vídeo el seguimiento de los objetos es un tema de atención, al patrón de movimiento de un objeto en la cámara o de la cámara sobre un objeto se le conoce como flujo óptico.

Como sucede en el módulo de segmentación, el campo de flujo óptico también hace uso de técnicas anteriormente descritas para realizar el seguimiento de un objeto en el vídeo, es de especial utilidad cuando se ha realizado reconocimiento de objetos, seguir los mismos para conocer su movimiento [10], de esta manera se puede conocer la posición de los objetos, inclusive conocer su forma.

2.1.7. Estructura del movimiento

Uno de los principales problemas de la imagen digital es la falta de conocimiento de la información tridimensional. En la captura de una imagen no es posible la extracción de formas 3D, a diferencia del ojo humano que utiliza la visión binocular para conocer la forma de los objetos a raíz de distintos puntos de vista como se observa en la figura 2.5.

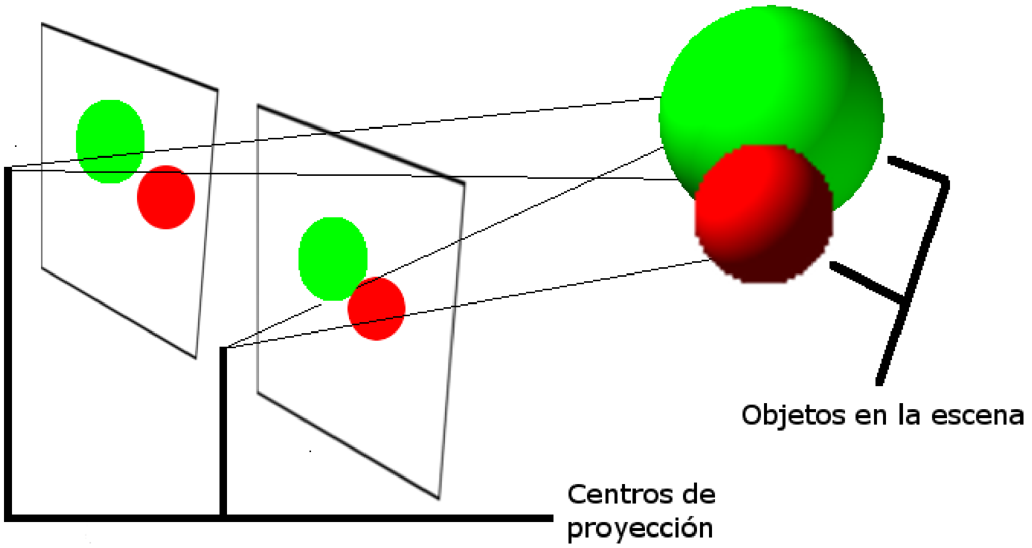


Figura 2.5 Visión binocular

Para la extracción de información 3D es necesario el uso de múltiples imágenes. Tras la captura de las imágenes, la problemática principal consiste en la extracción de los parámetros extrínsecos de la cámara, para ello es necesario reconocer las mismas características en las distintas imágenes, pudiendo así extraer la información necesaria [11]. Conociendo la correspondencia de puntos entre la secuencia provista se puede extraer la información 3D utilizando distintos algoritmos, pudiendo ser uno de ellos la triangulación [12]. En la figura 2.6 se observa la correspondencia necesaria para dos imágenes vista desde distintos puntos de vista.

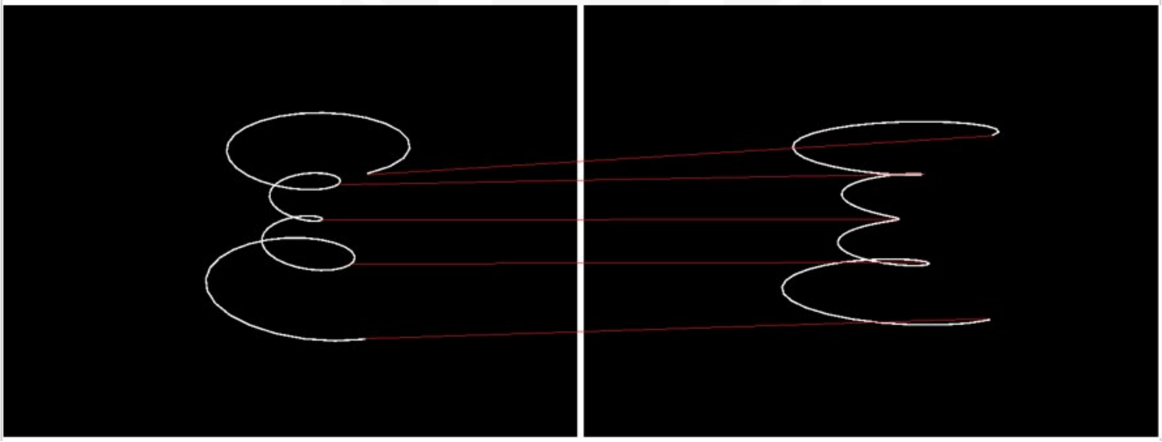


Figura 2.6 Correspondencias entre dos imágenes con distinto punto de vista

2.2. Estado del arte

Las librerías son herramientas comúnmente utilizadas para agilizar el desarrollo tanto de algoritmos como de aplicaciones. Existen múltiples librerías con funciones especializadas en procesamiento de imagen implementadas, lo que implica un alto índice de investigación en este campo. El objetivo principal del uso de las librerías es la implementación de nuevas ideas basándose en herramientas ya existentes y comunes en el ámbito de estudio, normalmente se procura que su uso sea sencillo, la simplicidad de uso depende del lenguaje a utilizar, e inclusive del rendimiento que se exija de dichas funcionalidades.

Uno de los lenguajes más utilizados en el ámbito de la investigación para procesamiento de imágenes es MATLAB [13]. Este lenguaje permite prototipar algoritmos de forma muy simple, y dispone de una amplia gama tanto de operaciones de bajo nivel, como pueden ser convoluciones, umbrales, operaciones matriciales, como algoritmos de más alto nivel como Canny, KLT [14], etc. Para el análisis de imagen es necesario instalar el paquete *"Image Processing Toolbox"*, destacan entre las características principales de este paquete las operaciones de segmentación, morfológicas, estadísticas, mediciones y procesamiento de imágenes 3D, además de operaciones más simples como se ha nombrado previamente. Su uso está extendido a nivel mundial, desde universidades hasta empresas utilizan este tipo de

Software, que es actualizado periódicamente. Por otro lado, pese a todas sus ventajas, tiene licencia de pago, y para poder utilizar este tipo de Software hacen falta equipos con una gran cantidad de recursos, además está limitado a un único sistema operativo (Windows). En este lenguaje, al igual que en otros similares no se pueden elaborar aplicaciones a nivel de usuario ya que no están diseñados para este fin, además los tiempos de ejecución son elevados. Otro tipo de Software similar pero de uso gratuito es Octave, en este caso se permite instalar en múltiples sistemas operativos, pero los tiempos de ejecución frente a MATLAB aumentan considerablemente, como se puede observar en la figura 2.7.

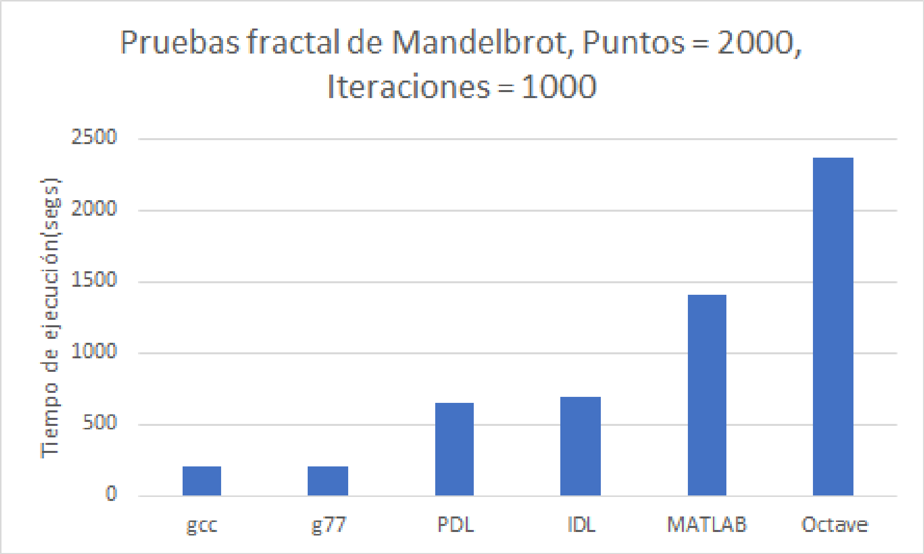


Figura 2.7 Pruebas de comparación del fractal de Mandelbrot en múltiples lenguajes
Fuente: http://freesoftwaremagazine.com/articles/cool_fractals_with_perl_pdl_a_benchmark/

Existen también otros tipos de librerías como FastCV [15], esta librería es específica para una arquitectura, y optimizada para un cierto tipo de procesadores, en este caso Snapdragon. Esta se utiliza para obtener alto rendimiento en móviles cuya arquitectura de procesador es ARM. Su uso principal se da en aplicaciones que obtengan los datos de la cámara y se post-procesen para mostrarlo al consumidor. Para el consumo de esta librería es necesario realizar la instalación de un SDK provisto por Qualcomm, y permite realizar la ejecución de operaciones en múltiples dispositivos de esta arquitectura, desde su DSP [16], hasta el procesador o la GPU. Esta librería permite utilizar determinadas funciones ya implementadas por los desarrolladores del equipo, produce una gran mejora respecto a implementaciones básicas o complejas de determinados algoritmos, no contiene una gran cantidad de algoritmos,

Tabla 2.1 Comparación OpenCV con FastCV

Función	OpenCV	FastCV	FastCV Snapdragon
NCC	1.0x	9.0x	23.1x
Multiplicación punto a punto 128x4	1.0x	4.0x	10.0x
Convert YUV420	1.0x	1.4x	1.3x
Sobel	1.0x	1.8x	7.8x
Mediana 3x3	1.0x	3.8x	51.9x
Gaussiano 3x3	1.0x	2.6x	4.1x
Gaussiano 5x5	1.0x	1.4x	2.9x
Umbral	1.0x	0.7x	9.7x
Integrar imagen	1.0x	1.1x	1.3x
Esquinas de Harris	1.0x	2.8x	8.6x
Dilatación	1.0x	1.4x	15.0x
Erosión	1.0x	1.3x	15.0x
Corrección de perspectiva	1.0x	21.5x	37.8x
Flujo óptico LK	1.0x	2.0x	14.3x

y las implementaciones son específicas para un determinado procesador.

OpenCV [17] es una librería abierta de desarrollo, conocida mundialmente, por su facilidad de uso, y su gran extensión. Esta librería está disponible en múltiples lenguajes de programación como C/C++, Python y Rust, su implementación principal se encuentra en C/C++, y contiene multitud de ficheros de configuración para poder compilar en distintas arquitecturas y sistemas operativos, se puede ejecutar el código tanto en CPU como en GPU si se dispone de CUDA u OpenCL, tiene una buena gestión de memoria, además de bajos tiempos de ejecución frente al procesamiento primitivo de imágenes. Permite realizar una gran cantidad de operaciones debido a la envergadura que contiene, se puede además contribuir al desarrollo de un modo sencillo, ya que usa un repositorio público en el que se dispone todo el código de la librería. Es totalmente configurable dependiendo de la arquitectura y el sistema, para intentar aprovecharse de los recursos ya instalados con el objetivo de obtener un mejor rendimiento. La generalización de este tipo de herramientas hace que en determinados tipos de arquitecturas la velocidad no sea suficiente, como se ve en la tabla 2.1.

En la tabla 2.1 se observa la comparación entre las librerías anteriormente nombradas (FastCV y OpenCV), además se pueden observar casi la totalidad de operaciones que tiene implementadas FastCV, mientras que las categorías de la librería OpenCV se pueden visualizar en la tabla 2.2, ahí se reflejan a grandes rasgos las funcionalidades que ofrece esta librería.

Tabla 2.2 Categorías de algoritmos en la librería OpenCV

Estructuras básicas y operaciones
Procesamiento de imagen (Filtros, transformaciones geométricas, ...)
Interfaces gráficas de alto nivel
Análisis de vídeo
Calibración de cámaras y reconstrucción 3D
Detección de características
Detección de objetos
Sistemas de aprendizaje automático
Agrupamientos
Algoritmos implementados en GPU
Fotografía computacional
Algoritmos implementados en OpenCL
Superresolución
Visualizador 3D

OpenCV es una librería robusta de alto rendimiento, pero no está preparada para ofrecer alto rendimiento en procesadores de bajo poder computacional, lo que hace necesario en este tipo de dispositivos un desarrollo específico sin hacer uso de esta u otras librerías.

Existen otras librerías o estándares con el mismo propósito, OpenVX [18] es un estándar que permite optimizar para múltiples plataformas con el fin también de ahorrar en consumo, este tiene ciertas restricciones en cuanto a programación, y un tiempo de desarrollo elevado. Una de las características de OpenVX es que tiene una ejecución basada en grafo, de este modo se aprovecha la memoria que se crea de forma intermedia y así se evita el exceso de uso de memoria, además permite tener definido el algoritmo de modo que cada vez que hay una ejecución ya está creado todo el preámbulo del algoritmo e interconectado con las funciones, esto es especialmente útil para algoritmos definidos que se ejecuten siempre del mismo modo, como puede ser en robótica, reconocimiento de objetos, etc. Este lenguaje tiene interés dado que los fabricantes hacen sus propias versiones lo que en cierto modo aporta un alto rendimiento en distintas arquitecturas, el periodo de desarrollo de los algoritmos es extenso, mayoritariamente dispone de funciones básicas en procesamiento de imagen, y su instalación es costosa.

También está OpenGL como estándar para trabajos con imágenes, buscando el alto rendimiento en gráficos, se utiliza para el desarrollo de videojuegos, está pensado para el uso de GPU, y lo relacionado con la interacción con las pantallas, este estándar se elabora a

Tabla 2.3 Comparación OpenCV-CImg

Arquitectura	Operación	CImg	OpenCV
Haswell (x64)	Erosión	1.0x	957.1x
	Dilatación	1.0x	1043.4x
	Desenfoque Gaussiano	1.0x	61.7x
ARM (x86)	Erosión	1.0x	81.2x
	Dilatación	1.0x	81.3x
	Desenfoque Gaussiano	1.0x	32.17x

partir de primitivas geométricas, que permiten dibujar gráficos.

Otra librería que cabe destacar es ImageR [19], es una librería escrita en C++, que contiene funciones básicas de procesamiento de imagen, ofrece funciones que permiten trabajar con imágenes de forma sencilla, como puede ser la carga, guardado, muestra de imágenes, transformaciones típicas (interpolaciones, redimensionados, rotaciones, etc), así como algunas operaciones más complejas como la transformada rápida de Fourier, operaciones morfológicas, eliminación de ruido, etc. Se utiliza en el entorno R, un software libre al igual que Octave, y también es multiplataforma, pero presenta las mismas limitaciones de velocidad. La librería está basada en CImg [20] una librería escrita en C++, para utilizarla en dicho lenguaje, ImageR utiliza el código de esta librería elaborando la interfaz necesaria para ser utilizada desde R. CImg es sencilla de compilar e instalar en múltiples dispositivos ya que solo se necesita un compilador de C++, esto implica que no está optimizado para distintos dispositivos. Se hace una comparación de algunas funcionalidades de esta librería contra la de OpenCV como se puede observar en la tabla 2.3; esta comparación se ha realizado en una arquitectura Haswell x64, con los mismos medidores de tiempo para no dar lugar a error, además se ha probado en un procesador ARM de una Raspberry PI 2 B+. Se observa en la tabla anteriormente nombrada una ventaja considerable de OpenCV respecto de la librería CImg, esto se debe a las optimizaciones de memoria pertinentes y el uso de operaciones específicas.

Cabe destacar la multitud de librerías que abordan el desarrollo de técnicas de visión por computador, entre ellas, se encuentra VXL [21], esta comienza su desarrollo en el año 1997, pensada para el ámbito de investigación a raíz de su rendimiento, es de compilación costosa, y no ofrece ninguna ventaja sobre las anteriormente nombradas, siendo equiparable por tamaño y número de operaciones a OpenCV. En la tabla 2.4 se muestra una comparación de algunas operaciones, al igual que la tabla 2.3, se observa el desenfoque gaussiano, siendo este más rápido que la librería CImg, pero aún de este modo OpenCV sigue venciendo en rendimiento

Tabla 2.4 Comparación OpenCV-VXL

Operaciones	OpenCV	VxL
Canny	103.5x	1.0x
Desenfoque Gaussiano (3x3)	57.8x	1.0x

de forma considerable, en la comparación con VxL no se especifíca arquitectura debido a que se ha realizado en un procesador Intel común debido a su dificultad de compilación para distintas plataformas, además de no apreciar optimizaciones específicas de Hardware. Una característica interesante en esta librería es que dispone de una versión de desarrollo llamada VxD, y añade utilidades de trabajo con grafos, y desarrollo y manejo de geometría de curvas, dicha implementación no es común de encontrar en las distintas librerías. El código de ambas versiones está disponible tanto de VxL como de VxD, teniendo ambos gran complejidad de comprensión, teniendo escasez de comentarios, e inclusive con distintas funcionalidades sin implementar. El mayor problema del que dispone esta librería es la cantidad ínfima de documentación, como la dificultad de instalación, y las optimizaciones de las que dispone no son equiparables a OpenCV, como se ha podido observar.

El objetivo de este trabajo es el diseño de una librería que presente las ventajas de cada una de las librerías nombradas anteriormente. Haciendo uso de técnicas nombradas, como el desarrollo a modo de grafo de OpenVX, y la implementación en distintas arquitecturas como OpenCV, se pretende obtener una ejecución en múltiples dispositivos y aprovechar por separado el rendimiento en cada uno de ellos mediante técnicas de optimización de algoritmos, como modificaciones de los mismos. Se pretende que el código sea de simple desarrollo, y conlleve un corto tiempo de aprendizaje, ofreciendo las ventajas que proveen otras librerías, procurando disminuir los tiempos de desarrollo de las librerías nombradas anteriormente, excepto de las librerías específicas.

Se puede observar en el estudio que OpenCV muestra cierta ventaja sobre librerías generales sin necesidad de que el fabricante provea librerías específicas para su Hardware, por este motivo se escoge como librería principal de comparación, ya que además de proveer una amplia gama de operaciones, dispone de rápidas ejecuciones de en las distintas plataformas y dispositivos estudiados.

Capítulo 3

Diseño y optimización de la librería

Este capítulo de divide en dos secciones principales en las que se incluyen el diseño de la librería y por otro lado el estudio sobre la optimización de distintos algoritmos referentes al campo de estudio, en ambas se realiza una descripción detallada de las metodologías utilizadas.

3.1. Diseño de la librería

El presente trabajo realiza la implementación de una librería para procesamiento de imagen. Previo al desarrollo de un proyecto de Software, es necesario diseñar una estructura del mismo, y realizar este diseño a conciencia para que pueda ser fácilmente modificado ampliando su contenido y que a su vez cumpla los requisitos funcionales que se deseen obtener.

El fin de la librería además del rendimiento computacional es la comodidad de uso para el cliente; la librería se desarrolla en C++, con la intención de tener una librería orientada a objetos sin hacer uso de paradigmas más antiguos. Para abordar la funcionalidad deseada de permitir la ejecución en múltiples plataformas sin necesidad de especificar dicha plataforma se hace uso del patrón de diseño estrategia [22], este patrón permite encapsular los objetos de modo que es sencillo cambiarlos para diferentes plataformas, y además da simplicidad a la hora de añadir nuevas clases con algoritmos.

La implementación contiene 3 clases principales tal y como se muestra en la figura 3.1, una encargada de gestionar los datos, otra interfaz de operaciones en la que se desarrollan los distintos algoritmos, y una clase grafo la cual se alimenta de operaciones para ejecutarlos a modo de grafo.

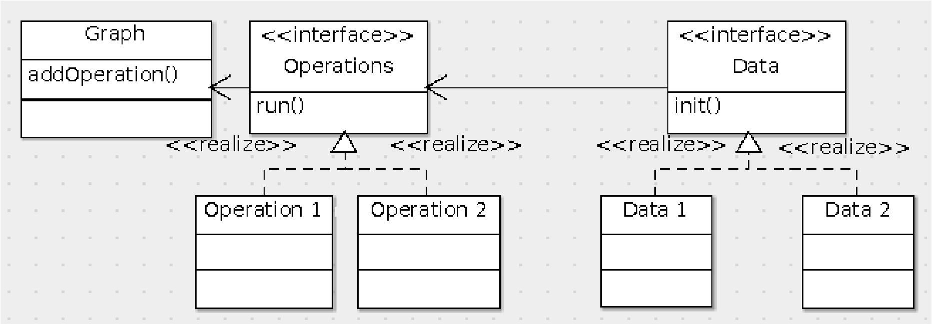


Figura 3.1 Diseño de la estructura básica de la librería

Los patrones de diseño con cierto nivel de abstracción implican que el nuevo desarrollo no modifique el antiguo, lo que hace que sea robusto y seguro. Para tener la seguridad necesaria, la librería cuenta con un núcleo que contiene las funciones triviales con los tipos de datos comunes que se implementan, este se encarga de la gestión de memoria, accesos, conversiones, etc.

Este diseño contempla las operaciones de grafo, puesto que cada una de las operaciones inicializa los datos necesarios para la ejecución en su constructor, mientras que se obliga a sobrecargar la función “*run()*” para de esta manera ejecutarse de forma cómoda en grafo. Como se explica previamente la funcionalidad de grafo esta diseñada con el propósito de abordar sistemas autónomos en los que sea necesario ejecutar de forma consecutiva el mismo conjunto de operaciones, por lo que cada operación se llama del siguiente modo:“*myimagefunction(input, args,output).run()*”, de esta forma el grafo contendrá las operaciones pertinentes pudiendo estas ser llamadas con la función “*run()*”.

Cada una de las operaciones contempla las distintas arquitecturas en la que es posible ejecutar las funciones, en el caso de no contemplarse alguna arquitectura concreta se ejecuta el código optimizado matemáticamente, obteniendo este a priori rendimiento superior frente a otras implementaciones del algoritmo, pero sin hacer uso de operaciones estándares, aunque cabe destacar que multitud de procesadores actualmente contemplan una serie de operaciones comunes para la optimización, por otro lado los avances en los compiladores permiten analizar el código y realizar una determinada serie de optimizaciones. La nomenclatura de uso es el nombre de la función a optimizar seguido de la arquitectura, en este caso las elegidas son ARM, INTEL, ATI y NVIDIA, dentro de cada una de ellas además se subdivide en las que contienen código nativo de las arquitecturas y OpenCL también optimizado por separado conforme la arquitectura.

Existen varios ficheros realizando una interfaz de las operaciones más habituales utilizadas a nivel nativo en cada uno de los procesadores, de manera que la llamada a estas es más cómoda, también incluidas en el Core.

El tener tanto un fichero de Core como las implementaciones comunes para distinto Hardware, permite al desarrollador cuando programe algoritmos de visión de más alto nivel que necesiten de operaciones básicas despreocuparse de la arquitectura en la que se vaya a ejecutar, ya que el sistema está pensado para que las funciones básicas dispongan de su implementación en distinto Hardware, al igual que las clases heredadas de la clase "*Operations*", que disponen de las diferentes arquitecturas, las operaciones de suma, multiplicación de matrices, etc; se sobrecargan para evitar la llamada específica a cada una de estas. Con el transcurso de desarrollo se pretende obviar la arquitectura siempre y cuando la inclusión de nuevos algoritmos utilice la mezcla de algoritmos de más bajo nivel, es decir, conforme se amplie la librería en cuestión las funciones de alto nivel harán uso de otras de más bajo nivel, con lo que no será necesario implementar en todas las arquitecturas los nuevos algoritmos.

Como se nombra anteriormente en la introducción, se compila conforme a la arquitectura con lo que la librería tras ser compilada no contempla todo el código, de esta manera es más ligera, para cuando se trata de dispositivos con poca memoria es bastante útil, ya que los objetos estarán cargados en memoria RAM y si contienen el código y los atributos de todas las arquitecturas estos ocupan mayor cantidad de memoria, por eso se divide en distintos ficheros conforme a las arquitecturas, y se utilizan directivas de compilación para evitar la inclusión de todo el código.

La estructura tiene varios paquetes conforme al área de visión al que pertenecen, se tienen las funciones de procesamiento de imágenes comunes, separadas de las técnicas de vídeo, también las funcionalidades de más alto nivel se separan, como los algoritmos de flujo óptico o de transformaciones de cámara.

También existen varios ficheros para la compilación puesto que la estructura de ficheros depende del sistema operativo, por ello se debe contemplar dividir la compilación conforme al sistema operativo, la compilación se encarga de buscar los recursos y requisitos del sistema para tomar la opción adecuada y producir un código optimizado conforme la arquitectura.

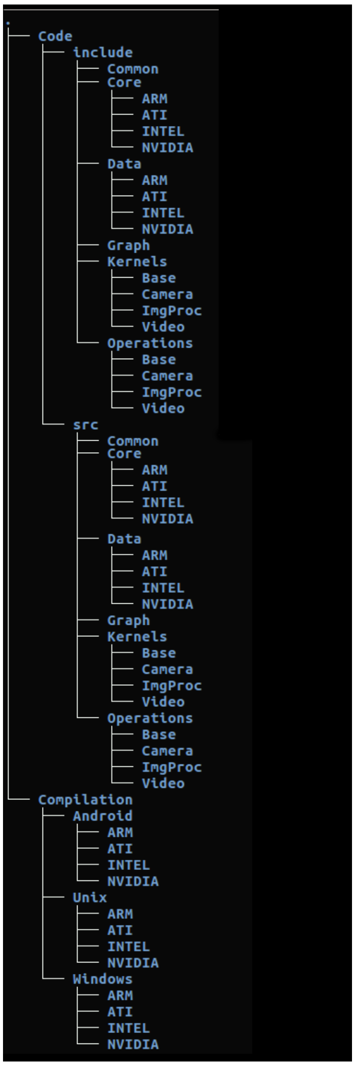
En la figura 3.2 se muestra el árbol de directorios a grandes rasgos de la librería. Como se puede observar de la raíz cuelgan dos directorios, uno que contiene los ficheros de compilación, teniendo los distintos sistemas operativos en los que se va a compilar, e interno a cada uno de ellos se tienen cada una de las arquitecturas. Por otro lado se tiene el código que tiene separada la parte de las cabeceras del fuente, e internamente de cada una de ellas se encuentra la estructura que se muestra en la figura 3.1, además de una carpeta que contiene los kernels de OpenCL, más la estructura de "Core" y la "Common" que contiene la interfaz de interconexión entre los diferentes módulos.

Como se puede observar en la figura 3.2, tanto en los ficheros de código como en las cabeceras se tienen las carpetas: Common, Core, Data, Graph, Kernels y Operations. Cada una de estas carpetas dispone de distintas funcionalidades, es decir, se realiza una separación lógica de las mismas para así disponer de una estructura organizada, y de fácil búsqueda en la misma. La funcionalidad de cada una de las carpetas es la siguiente:

- Common: esta carpeta contiene los ficheros inherentes a la interfaz de llamada a las funciones, no contempla la subdivisión de carpetas conforme las arquitecturas, puesto que dicha interfaz de llamada a las funciones implementadas debe ser igual para todas las arquitecturas, ya que así se aprovecha la portabilidad del sistema. Además esta carpeta contiene la interfaz de elección de plataforma, creación de contexto y cola de comandos de OpenCL.

- Core: esta carpeta se subdivide en las distintas arquitecturas elegidas a desarrollar, ya que contiene las interfaces de llamada a operaciones de la arquitectura concreta de forma sencilla, se separa debido a que cada arquitectura tiene su función específica para determinadas operaciones. Esta interfaz se desarrolla con el fin de obtener llamadas a funciones específicas del mismo modo para las diferentes arquitecturas sin necesidad de tener siempre la documentación disponible, además de estandarizar las llamadas dentro de la librería.

- Data: esta carpeta contiene los distintos nombres de arquitectura debido a que los tipos de dato en las diferentes arquitecturas difieren entre sí, el manejo de datos se realiza por arquitectura dado que el alineado de memoria, gestión y tamaños de tipos de dato



```
.
├── Code
│   ├── include
│   │   ├── Common
│   │   ├── Core
│   │   │   ├── ARM
│   │   │   ├── ATI
│   │   │   ├── INTEL
│   │   │   └── NVIDIA
│   │   ├── Data
│   │   │   ├── ARM
│   │   │   ├── ATI
│   │   │   ├── INTEL
│   │   │   └── NVIDIA
│   │   ├── Graph
│   │   ├── Kernels
│   │   │   ├── Base
│   │   │   ├── Camera
│   │   │   ├── ImgProc
│   │   │   └── Video
│   │   └── Operations
│   │       ├── Base
│   │       ├── Camera
│   │       ├── ImgProc
│   │       └── Video
│   └── src
│       ├── Common
│       ├── Core
│       │   ├── ARM
│       │   ├── ATI
│       │   ├── INTEL
│       │   └── NVIDIA
│       ├── Data
│       │   ├── ARM
│       │   ├── ATI
│       │   ├── INTEL
│       │   └── NVIDIA
│       ├── Graph
│       ├── Kernels
│       │   ├── Base
│       │   ├── Camera
│       │   ├── ImgProc
│       │   └── Video
│       └── Operations
│           ├── Base
│           ├── Camera
│           ├── ImgProc
│           └── Video
└── Compilation
    ├── Android
    │   ├── ARM
    │   ├── ATI
    │   ├── INTEL
    │   └── NVIDIA
    ├── Unix
    │   ├── ARM
    │   ├── ATI
    │   ├── INTEL
    │   └── NVIDIA
    └── Windows
        ├── ARM
        ├── ATI
        ├── INTEL
        └── NVIDIA
```

Figura 3.2 Árbol de diseño de la librería

no son necesariamente iguales en las distintas arquitecturas.

- Graph: Al igual que la carpeta Common, la estructura de grafo es común para las distintas arquitecturas, ya que se encarga de inicializar la memoria y ejecutar por separado tras tener el resto de operaciones implementadas.

- Kernels: Esta estructura contiene las carpetas que implementan las diferentes operaciones contempladas en la librería de visión, en cada una de las carpetas de operación también se incluyen las distintas arquitecturas, ya que aún conteniendo el código de un lenguaje estándar, las implementaciones son distintas conforme las arquitecturas para poder obtener un mayor rendimiento.

- Operations: Esta tiene la misma estructura que la carpeta Kernels, en Operations, se encuentran las distintas implementaciones de las operaciones de visión, ya sean funciones nativas o desarrolladas en OpenCL.

Cabe destacar en una de las funcionalidades de Common, concretamente la creación de contexto de la aplicación y de OpenCL, se realizan en una clase Singleton [23], para poder utilizar dicha instancia en toda la aplicación, aprovechando su creación principal para agilizar la ejecución de operaciones, la creación de contexto permite generar un gestor de memoria, y recursos del sistema, intentando aprovechar al máximo dichos recursos implicando un incremento de rendimiento considerable. Por otro lado este gestor de recursos permite la asignación de un número máximo de hilos a utilizar, en el caso de utilizar la aplicación junto a otras que también necesiten el uso de gran cantidad de recursos, no se utilizan la totalidad de unidades de cómputo.

La librería está documentada con Doxygen [24], de esta manera se genera la documentación en HTML de forma automática, permitiendo así comodidad a la hora de búsqueda de información y paso de argumentos.

Tras la elaboración del diseño se prosigue con la implementación de la librería, buscando técnicas de optimización, sin contemplar a priori las arquitecturas, de este modo se busca primero la metodología correcta, de modo que cuando sea necesario realizar implementaciones específicas se hagan sobre un código ya optimizado, y utilizar este para arquitecturas que no se contemplen en la librería.

3.2. Optimizaciones destacadas

En este Trabajo Fin de Máster cabe destacar la serie de investigaciones realizadas para optimizar las distintas funcionalidades previstas en la librería, esta sección contiene las de mayor avance y modificación sobre los algoritmos originales, realizando una descripción detallada de los avances y la versión final de los algoritmos.

3.2.1. Convolución

Como se observa en el Capítulo 2, la operación de convolución es ampliamente utilizada en el campo de visión por computador, y esta tiene alta complejidad computacional (O(n^4)), como se observa en la ecuación 2.3.

Una primera optimización sobre este algoritmo implica realizar la convolución de forma separable, esto implica aplicar la convolución de dos Kernels unidimensionales en lugar de uno bidimensional realizando luego su suma. En la ecuación 3.1 se observa las operaciones a realizar por píxel de forma separable.

$$I(x,y) * f = \sum_{i=0}^{\theta} I(x+i-\theta/2,y)f(i) + \sum_{j=0}^{\theta} I(x,y+j-\theta/2)f(j)^T \tag{3.1}$$

utilizando la misma nomenclatura que en la ecuación 2.3, se puede observar que la complejidad computacional disminuye.

Para el uso de esta técnica es necesario que los filtros de convolución sean separables, esto implica que la convolución 2D se pueda producir por la convolución de dos kernels unidimensionales. Un filtro 2D es separable cuando el rango de su matriz [25] es 1, debido a que se busca que los vectores sean linealmente dependientes, esto quiere decir que las columnas son múltiplos de las otras columnas. La obtención de los filtros unidimensionales tras conocer el rango, se realiza utilizando descomposición por valores singulares (SVD), en este caso se tiene un único valor singular distinto de 0. Para extraer los valores de la convolución se escogen las primeras columnas de U y V resultantes del SVD, conocidos como vectores-singulares-izquierdos y vectores-singulares-derechos respectivamente, en el Algoritmo 1 se observa la implementación de una función que retorna los filtros 1D en caso de ser separable la convolución que se recibe por argumento.

```
Data: Convolución 2D
Result: FiltroVertical, FiltroHorizontal
rangomatriz = rank(Convolución 2D) // Se calcula el rango de la matriz
    entrante
if rangomatriz == 1 then
    u,s,v = svd(Convolución 2D);
    // Se empieza el índice en 0, y se accede por [columna, fila]
        donde los : devuelven todos los valores de la posición
        correspondiente
    // La función sqrt ejecuta la raíz cuadrada
    // s es un vector que contiene los valores de la diagonal
        ordenados de mayor a menor
    FiltroVertical = u[:, 0] * sqrt(s[0]);
    FiltroHorizontal = v[:, 0] * sqrt(s[0]);
```

Algoritmo 1: Extracción de convoluciones unidimensionales de un kernel 2D

La técnica de convolución separable es de gran ahorro en el caso de que los filtros tengan rango 1, pero estos no contemplan una mejora de todos los casos posibles. La complejidad computacional de esta implementación es de O(n^3), esto supone una ventaja respecto a la implementación tradicional.

Otro modo de realizar la implementación de una convolución se puede dar en el dominio de frecuencias, este dominio surge de la transformación de dominio de intensidad, siendo este último el que trata las imágenes con valores de color dentro de las matrices, y el dominio de frecuencias representa la imagen espacial en función de senos y cosenos, donde cada punto contiene un valor de frecuencia que se encuentra en la imagen de intensidad. Una convolución en dominio transformado se efectúa realizando una multiplicación punto a punto entre el Kernel en dominio de frecuencias y la imagen en el mismo dominio. La ecuación 3.2 describe el cómputo que se debe realizar por cada píxel para transformar de dominio de intensidad a dominio de frecuencias, conocido como transformada de Fourier.

$$F(u,v) = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} I(x,y) e^{-2\pi i (\frac{uv}{M} + \frac{uv}{N})} \tag{3.2}$$

donde M y N son las filas y columnas de la imagen respectivamente, y las posiciones u y v las resultantes. Se puede observar la complejidad de dicha función, esto implica la necesidad de transformar a dominio de frecuencias para convolucionar, considerando M y N las filas y las columnas de una imagen I, el número de operaciones de lectura escritura que se debe hacer para realizar una convolución pasando por dominio de frecuencias y volviendo a dominio espacial, teniendo en cuenta que la transformación desde Fourier a imagen de intensidad conlleva el mismo número de operaciones, los accesos a memoria son los siguientes: $2M^2N^2+MN$, suponiendo el uso de una convolución no separable en dominio de intensidad se obtiene como número de operaciones: $MNmn$, donde m y n es el tamaño de la convolución, por lo que para obtener optimización realizando una convolución en dominio transformado se debe resolver la inecuación 3.3, viendo su resultado en la ecuación 3.4, cabe destacar que la ecuación referenciada supone un tamaño de convolución cuadrada.

$$2M^2N^2+MN < MNmn \tag{3.3}$$

$$m > \sqrt{2MN+1} \tag{3.4}$$

Por otro lado, existe la FFT (*Fast Fourier Transform*) [26] , en la que la precisión del resultado difiere de la original, pero es totalmente válida para realizar convoluciones en imágenes, esta tiene un orden de complejidad de O(N $log(N)$), esto implica que para filtros de un determinado tamaño es viable el uso del método obteniendo cierta aceleración.

La utilidad principal de las convoluciones consiste en suavizar, realzar, u obtener altas frecuencias en la imagen, para ello el tamaño de filtros que se suele utilizar es inferior a 5x5, por ello el uso de la transformada de Fourier no se hace viable. Como el objetivo de la librería es aprovechar los recursos computacionales de múltiples plataformas, se deben tener en cuenta los dispositivos de bajo rendimiento, por esta razón se investiga en un algoritmo que minimice el consumo de recursos. El algoritmo propuesto para cumplir los requisitos se conoce como convoluciones de Winograd [27].

Este algoritmo consiste en calcular el número de operaciones mínimas para calcular mxn salidas en un filtro de tamaño rxs, esto se denota como $F(mxn,rxs)$. En la ecuación 3.5 se puede observar el número de multiplicaciones que conlleva realizar un filtro con esta técnica.

$$\mu(F(mxn,rxs)) = (m+r-1)(n+s-1) \tag{3.5}$$

Este algoritmo se utiliza para kernels cuadrados, realizando la convolución del siguiente modo:

$$Y = A^T \left[[GgG^T] \odot [B^T dB] \right] A \tag{3.6}$$

Donde $\odot$ es la multiplicación punto, g es el filtro y d es un recuadro de la imagen de tamaño $(m+r-1)x(m+r-1)$ y las matrices restantes se definen a continuación para una convolución F(2x2, 3x3), es decir 4 salidas utilizando una convolución de 3x3:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 1 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{3.7}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 1 \end{bmatrix}$$

Para la generación automática de las matrices además de los valores m y r es necesario dar los coeficientes de un polinomio de grado $m+r-2$, estos coeficientes se obtienen del teorema *Chinese Remainder Theorem* [28], en los algoritmos 2 y 3 se observa el pseudocódigo para la generación de las matrices.

Si se calcula el polinomio para un F(2x2, 3x3), se obtienen los índices $(0, 1, -1)$, al alimentar el algoritmo provisto en 2, se obtienen los resultados reflejados en la ecuación 3.7. Esta técnica es computacionalmente eficiente, debido que para un filtro 3x3 con una salida de 4 valores se realizan 6 multiplicaciones por cada valor de la imagen, frente a 9 que se realizan en una convolución convencional, además tiene la ventaja a diferencia de las convoluciones separables de poder tener cualquier tipo de coeficientes en los kernels, sin la necesidad de que sean separables, y es sencillo de calcular para filtros de pequeño tamaño.

```
Data: Polinomio, m, r
Result: Bt, G, At
tileSize = m + r - 1;
AuxMatrix = MatrixZeros[tileSize, tileSize];
// Se inicializa una matriz de 0
AuxMatrix[tileSize -1, tileSize - 1] = 1;
for $i \leftarrow 0$ to $tileSize - 1$ do
    AuxMatriz[i, i] = 1;
    for $j \leftarrow 0$ to $tileSize - 1$ do
        if i != j then
            AuxMatriz[i, i] *= (Polinomio[i] - Polinomio[j]);
        else
            AuxMatriz[i, i] *= 1;
        end
    end
end
AuxMatrixCopy = AuxMatrix;
AuxMatrix[0, 0] = Abs(AuxMatrix[0, 0]);
AtMatrix = Matrixzeros[m, tileSize];
AtMatrix[m -1, tileSize -1] = 1;
for $i \leftarrow 0$ to $m - 1$ do
    for $j \leftarrow 0$ to $tileSize - 1$ do
        // La función power calcula la potencia entre dos números
        AtMatrix[i, i] = power(Polinomio[i],j);
    end
end
// Se realiza la matriz traspuesta
AtMatrix = AtMatrix.transpose();
```

Algoritmo 2: Algoritmo de cálculo del filtro mínimo para extraer las matrices de la convolución Winograd

```
Data: Polinomio, m, r
Result: Bt, G, At
GMatrix = AtMatrix / AuxMatriz.transpose();
BMatrix = MatrixZeros[tileSize, tileSize];
Polinomios = CrearPolinomios[tileSize - 1];
// Se crean los polinomios de forma paramétrica con x, ej:
   Polinomio[i] = (x - 1) * (x + 1) * ...
for $i \leftarrow 0$ to $tileSize - 1$ do
    Polinomios[i] = 1 for $j \leftarrow 0$ to $tileSize - 1$ do
        if i != j then
            Polinomios[i] *= x - Polinomios[j] ;
        else
            Polinomios[i] *= i;
        end
    end
end
// En esta extensión se asume que se tienen las variables creadas
   anteriormente
Baux = MatrixZeros[tileSize, tileSize];
for $i \leftarrow 0$ to $tileSize - 1$ do
    for $j \leftarrow 0$ to $tileSize - 1$ do
        // Polinomios(i,j) sustituye el valor j en el polinomio i
           dando la imagen j en el polinomio i
        Baux[i, j] = Polinomios(i, j) / AuxMatrixCopy[i, i];
    end
end
Baux = Baux.transpose();
BauxMult = MatrixZeros[tileSize, tileSize];
BauxMult[:tileSize - 1, :tileSize - 1] = Identity(tileSize - 1);
for $i \leftarrow 0$ to $tileSize - 1$ do
    BauxMult[tileSize - 1, i] = power(-Polinomio(i),tileSize -1);
end
BMatrix = (Baux * BauxMult).transpose()
```

Algoritmo 3: Extensión del algoritmo 2 para el cálculo de la matriz B

Otra de las investigaciones realizadas implica la ejecución de los filtros conocidos como filtros de caja, estos kernels consisten en una convolución que contempla una matriz rellena con valores 1, y finalmente se hace una normalización por el tamaño de la matriz, esto aplica un efecto de suavizado a la imagen como se observa en la figura 3.3.

Figura 3.3 Filtro de caja de tamaño 7 x 7

Una primera aproximación investigada consiste en el desplazamiento del kernel con reaprovechamiento de memoria, es decir, se efectua un recorrido de la imagen almacenando los datos intermedios, de manera que para el cálculo de el primer píxel es necesario recorrer el tamaño de la matriz de convolución, pero al estar contenida con unos, se almacena la acumulación de la primera columna y de la primera fila para en las siguientes iteraciones evitar el recorrido intermedio. En la figura 3.4 se muestra una descripción gráfica del proceso, se puede observar en color rojo los nuevos datos a recorrer y acumular y en amarillo los valores que se deben restar. Esta técnica en la práctica es más rápida que la convolución separable, debido a que no surge la necesidad de recorrer dos veces la imagen por separado y luego acumular, esta técnica se nombra a partir de ahora *ventana deslizante*.

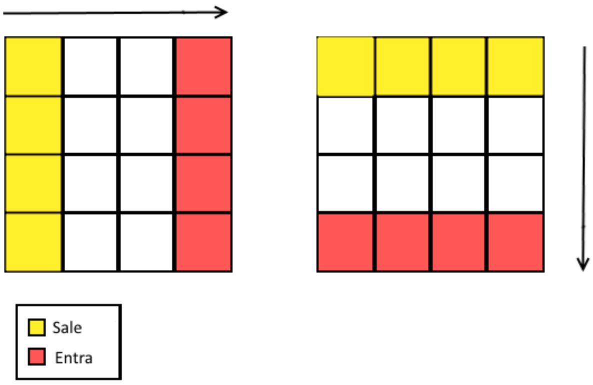


Figura 3.4 Desplazamiento de memoria en el cálculo del filtro de caja

La técnica nombrada anteriormente al igual que las convoluciones separables cuando el tamaño del kernel aumenta se nota la bajada de rendimiento. Para abordar este problema se hace uso del algoritmo *Summed-Area-Table* [29], esta técnica se utiliza para calcular la integral de una imagen, con el objetivo final de hacer el algoritmo de orden O(n^2), es decir, para cualquier tamaño de filtro de caja solo es necesario recorrer una vez la imagen entera, para ello se hace un primer recorrido calculando la integral como se observa en el Algoritmo 4.

```
Data: Imagen
Result: ImagenIntegral
// Se crea una matriz del mismo tamaño que la imagen
$ImagenIntegral$ = $MatrixZeros$[Imagen];
$ImagenIntegral$[0, 0] = $Imagen$[0, 0];
for $i \leftarrow 1$ to $Ancho(Imagen)$ do
    $ImagenIntegral$[0, i] = $ImagenIntegral$[0, i - 1] + $Imagen$[0, i];
end
for $i \leftarrow 1$ to $Alto(Imagen)$ do
    $ImagenIntegral$[i, 0] = $ImagenIntegral$[i - 1, 0] + $Imagen$[i, 0];
end
for $i \leftarrow 1$ to $Ancho(Imagen)$ do
    For$j \leftarrow 1$ to $Alto(Imagen)$ $ImagenIntegral$[i, j] = $ImagenIntegral$[i, j - 1] +
      $ImagenIntegral$[i - 1, j] -
    $ImagenIntegral$[i - 1, j - 1] + $Imagen$[i, j];
end
```

Algoritmo 4: Cómputo de la integral de una imagen

Con la integral de la imagen el cálculo del filtro de tamaño n se observa en la ecuación 3.8.

$$F(x,y) = i(x+n/2, y+n/2) + i(x-n/2, y-n/2) - i(x-n/2, y+n/2) - i(x+n/2, y-n/2) \tag{3.8}$$

donde i es la imagen integral, y F(x,y) la imagen filtrada para el punto (x,y).

Las distintas técnicas abordadas hasta ahora no suponen variaciones notorias en los resultados, por lo que dichas optimizaciones se consideran válidas sin la necesidad de descartarlas debido a decrementos en precisión.

Las convoluciones convencionales sólo tienen en cuenta la información espacial, existen filtros no lineales utilizados en distintas técnicas con intención de reducir ruido y mantener las altas frecuencias en la imagen(bordes), el filtro bilateral [30] se clasifica dentro de los filtros no lineales, este filtro considera las diferencias de color para efectuar las operaciones, esto permite una reducción del ruido de la imagen, en la figura 3.5 se puede observar la aplicación del filtro bilateral a una imagen.

Figura 3.5 Imagen resultante de la aplicación del filtro bilateral, ampliada para apreciar el efecto

El filtro bilateral se define en la ecuación 3.9.

$$\begin{gathered} If_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(||p-q||) G_{\sigma_r}(Ip - Iq) * Iq \\ W_p = \sum_{q \in S} G_{\sigma_s}(||p-q||) G_{\sigma_r}(Ip - Iq) \end{gathered} \tag{3.9}$$

Donde W_p es el factor de normalización, p y q son los píxeles en el espacio de imagen I, Ip e Iq es el valor de intensidad en p y q, σ_r es la desviación típica de la gaussiana que se le aplica a la distancia de color, y σ_s la desviación típica de la gaussiana espacial, S indica el recorrido del tamaño de ventana asignado. Se puede observar en la ecuación 3.9 el cálculo de dos Gaussianas, para el ahorro computacional se modifica la gaussiana espacial sustituyendo la misma por un filtro de caja, observándose en la ecuación 3.10.

$$\begin{gathered} If_p = \frac{1}{W_p}\sum_{q\in N_{\sigma_s}(p)} G_{\sigma_r}(Ip-Iq)\cdot Iq \\ W_p = \sum_{q\in N_{\sigma_s}(p)} G_{\sigma_r}(Ip-Iq) \end{gathered} \tag{3.10}$$

donde N_{σ_s}(p) = $q, ||p-q|| \leq \sigma_s$, $N_{\sigma_s}(p)$ es la vecindad en un cuadrado de tamaño $\sigma_s x \sigma_s$, de este modo la sigma espacial depende del histograma de los vecinos, la implementación con esta reducción se puede observar en el Algoritmo 5.

El cálculo del tamaño de ventana para este algoritmo se calcula del siguiente modo:

$$ws = 2\sigma_s + 1 \tag{3.11}$$

Al igual que en los filtros de caja convencionales, se puede aplicar la técnica de ventanas deslizantes que se refleja en la figura 3.4 aplicada al cálculo de los histogramas, reaprovechando así una parte de los cálculos.

```
Data: Imagen, SigmaS, SigmaR
Result: Filtro Bilateral
Gaussiana = createGaussian(0, 511) ; // El doble de 256 para poder hacer
    el desplazamiento correcto
for Pixel ← 0 to Ancho(Imagen) * Alto(Imagen) do
    // Se calcula el hgistograma para el tamaño de ventana
    for xval ← Pixel − SigmarR to Pixel + SigmaR do
        for yvalue ← Pixel − SigmaR to Pixel + SigmaR do
            Histogram[Imagen[xval + yvalue * Ancho]] ++;
        end
    end
    // Se calcula la normalización, y el valor
    for iter ← 0 to 256 do
        PesosP += Histogram[iter] * Gaussian[255 -Imagen[Pixel]];
        ValorPixel += Histogram[iter] * Gaussian[255 -Imagen[Pixel]] * iter;
    end
    Resultado[Pixel] = ValorPixel / PesosP;
end
```

Algoritmo 5: Filtro Bilateral con optimización de gaussiana espacial

Cabe destacar que aunque esta optimización suponga una ventaja sobre el algoritmo principal, sigue siendo computacionalmente costosa, al igual que la técnica SAT se aplica sobre los filtros de caja, se puede aplicar sobre el cálculo de los histogramas, a esto se le conoce como histogramas integrales [31]. La diferencia respecto al original radica en que por cada posición del píxel se debe guardar el histograma acumulado hasta llegar a la posición de cada píxel, siendo el cálculo principal fácilmente paralelizable, este se muestra en la ecuación 3.8, pero aplicado a los 256 valores del histograma (imagen monocromo). La ventaja de aplicar esta técnica consiste en separar el coste computacional del parámetro σ_s, aplicar la técnica descrita tiene la problemática del consumo de recursos de memoria, debido a que el almacenamiento de un histograma es elevado, la ecuación 3.12 refleja el consumo de memoria del algoritmo en bytes.

$$M = w * h * 256 * dt \tag{3.12}$$

donde w es el ancho de imagen, h el alto, y dt es el tamaño del tipo de dato a utilizar en bytes, el tipo de dato se puede escoger en relacción al tamaño de la imagen, se pueden escoger las variantes de enteros sin signo, y se debe tener en cuenta la siguiente premisa $w * h < 2^{dt*8}$, porque en el caso peor la matriz puede contener el mismo valor de intensidad. Para una imagen FullHD (1920 x 1080) utilizando como tipo de dato un entero sin signo de 4 bytes se tiene un consumo de memoria aproximado de 2GB, esto limita las arquitecturas en las que se puede implementar. Para reducir el coste de memoria se puede agrupar el histograma reduciendo la precisión, es decir, se puede escoger el rango de valores del histograma pudiendo este ser inferior a 256, pero se debe tener en cuenta la pérdida de precisión ya que al dividir el rango el múltiplos de 2 es necesario hacer una agrupación cada n valores, siendo n el múltiplo al que se aplica la división.

Puesto que el algoritmo de histogramas integrales consume una gran cantidad de recursos, se propone una solución de ahorro de memoria, aumentando de forma mínima el costo computacional. Para no aumentar de forma considerable el cómputo se hace almacenamiento de un histograma cada n posiciones, siendo el peor de los casos recorrer el histograma más n x n posiciones para obtener el valor del histograma integral, a mayor n mayor ahorro de memoria y más coste computacional. En la ecuación 3.13 se refleja el cálculo de la nueva memoria a reservar.

$$M = (\frac{w}{n}\frac{h}{n}) * 256 * dt + w * h * dt \tag{3.13}$$

En caso de utilizar un valor n = 10 para una imagen FullHD el consumo de memoria pasa a ser 30MB, pero supone en el peor de los casos un recorrido de 100 valores más por cada ventana posible de (n-1)x(n-1). En la figura 3.6 se muestra gráficamente las posiciones donde se almacenan los histogramas, almacenando el resto sus valores iniciales, los puntos rojos en la figura indican donde se realiza el almacenamiento del histograma, y los puntos azules el peor de los casos de recorrido, ya que se debe recorrer una matriz 9 x 9 para tener la totalidad del histograma.

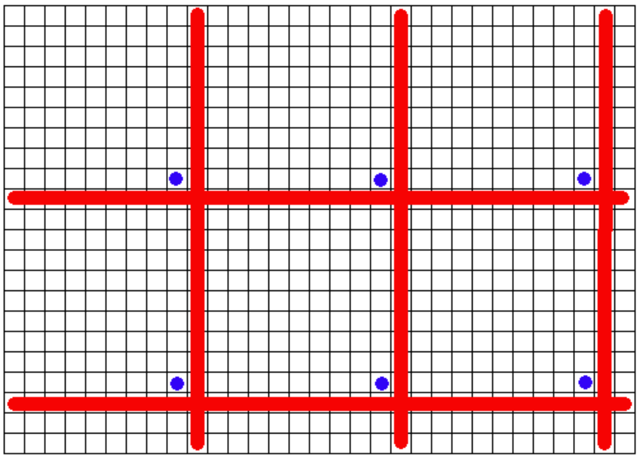


Figura 3.6 Estructura de memoria para el ahorro de recursos con la técnica histogramas integrales, para un tamaño n = 10

El uso de histogramas integrales es una solución compleja para este problema, dado que para reducir el costo computacional de la técnica denominada ventanas deslizantes se necesita que el tamaño de ventana sea superior a 512, debido a que cada histograma contiene 256 valores en caso de no ser agrupados por múltiplos de dos como se ha comentado previamente, y se necesita realizar operaciones con 4 histogramas. La ventana deslizante hace un único recorrido completo del tamaño de ventana al cuadrado, siendo los resultantes del tamaño de ventana ya que sólo es necesario recorrer un lado de la ventana sea vertical u horizontalmente. La técnica SAT supone un gran ahorro computacional para realizar filtros de caja sobre imágenes, en el caso de utilizar histogramas la ventana deslizante supone una gran ventaja sobre este, debido a su consumo de memoria, y la capacidad de ahorro de cálculo.

3.2.2. Calibrado y correción de la imagen

Las cámaras digitales introducen distorsión en la captura de imágenes debido a las deformaciones de la lente, dicha distorsión es principalmente radial, notándose en las esquinas de las imágenes. Para la búsqueda de patrones en imágenes consecutivas, o inclusive en imágenes de distintas cámaras es necesario corregir dicha distorsión para que las imágenes sean lo más parecidas posibles. Este calibrado se realiza por cada lente, en el capítulo 2 se hace referencia a un algoritmo para realizar el calibrado de lentes. El resultado de la calibración es la aplicación de un mapa de coordenadas donde una coordenada (x, y) se convierte en (x', y') tras aplicar el factor de distorsión. El proceso a aplicar para la aplicación radial se refleja en la ecuación 3.14.

$$\begin{gathered}
x \leftarrow (u - c'_x)/f'_x \\
y \leftarrow (v - c'_y)/f'_y \\
[XYW]^T \leftarrow R^{-1} * [xy1]^T \\
x' \leftarrow X/W \\
y' \leftarrow y/W \\
k_r = \frac{1+k_1 r^2+k_2 * r^4+k_3 * r^6}{1+k_4 r^2+k_5 r^4+k^r_6 6} \\
x'' \leftarrow x' k_r + \cancel{2p_1 x' y' + p_2(r^2 + 2x'^2)} \\
y'' \leftarrow y' k_r + \cancel{p_1(r^2 + 2y'^2) + 2p_2 x' y'} \\
map_x(u,v) \leftarrow x'' f_x + c_x \\
map_y(u,v) \leftarrow y'' f_y + c_y
\end{gathered} \tag{3.14}$$

donde las variables con una comilla simple pertecen a una matriz intrínseca de cámara ajustada, o equivale a la matriz original si no se dispone de un ajuste de los parámetros intrínsecos, $[X,Y,W]^T$ son las coordenadas en el mundo real, y $[xy1]^T$ las coordenadas de imagen homogéneas, siendo R la matriz identidad, las partes tachadas en la ecuación pertenecen a distorsión tangencial la cual no se aborda en el estudio, dado que es menos común y con la distorsión radial se corrigen la mayoría de los errores, por último x'' e y'' son las coordenadas corregidas siendo map_x y map_y el mapeo de coordenadas. Para calcular un ajuste de parámetros intrínsecos es necesario una escena estéreo, donde en base a las correspodencias entre las distintas imágenes se puede estimar una matriz intrínseca más ajustada. En la imagen 3.7 se observa cómo se ocasiona la distosión.

Se puede observar en la figura 3.7 un punto P en el mundo real, donde la línea punteada roja es el camino que debe seguir, pero al llegar a la lente, se desvía por la línea punteada azul, situando dicho punto en otro lado del sensor, provocando así el efecto de distorsión radial producido por la lente.

Al contemplar distorsión radial, dicha distorsión es simétrica tanto horizontal como verticalmente, esto es de gran ayuda dado que la primera optimización posible es el cálculo de la distorsión en un único cuadrante de la imagen, realizando las inversiones correspondientes tras tener los coeficientes para hallar las del resto de cuadrantes. Al realizar dicho cambio, se debe contemplar el cambio pasando de acumular el centro a tener el desplazamiento correspondiente. En la ecuación 3.15 se aprecia dicho cambio, donde se refleja la modificación

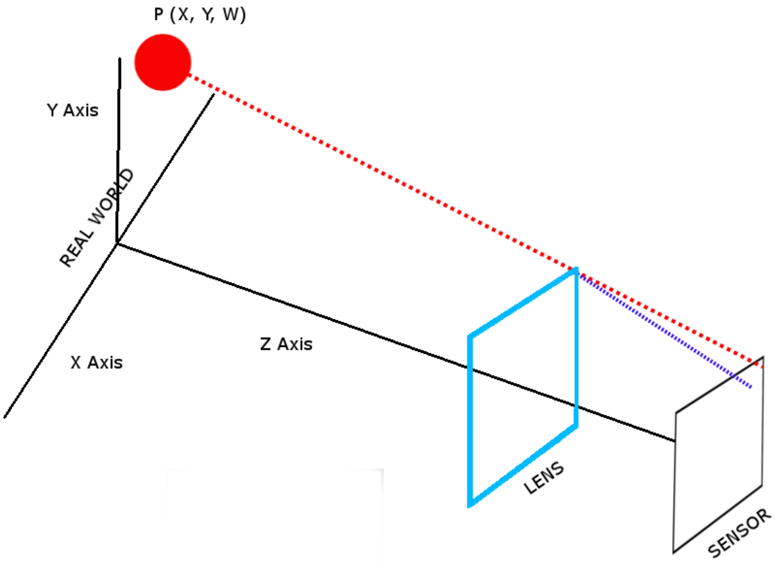


Figura 3.7 Efecto de distorsión radial en una imagen

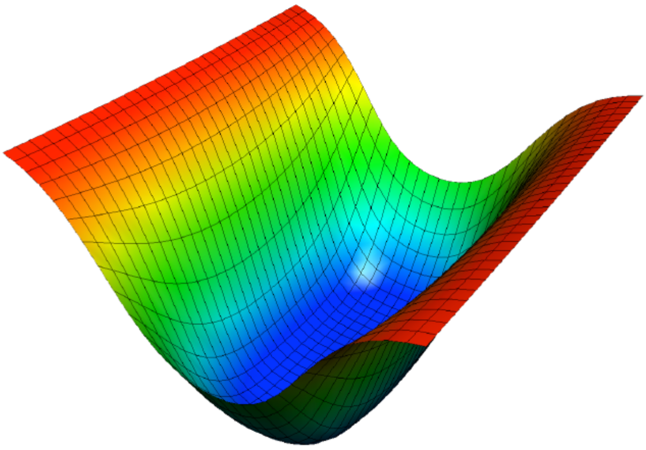


Figura 3.8 Representación del valor k_r para una determinada configuración de imagen

realizada para la optimización.

En la figura 3.8 se observa el comportamiento del parámetro k_r dados unos parámetros de distorsión determinados y un tamaño de imagen predefinido, de este modo se puede observar de forma aproximada que distorsión se debe corregir, y además se observa la simetría que ofrece.

$$\begin{aligned} u &= k_r x + c_x \\ y &= k_r y + c_y \\ k_r x &= x + \Delta x \rightarrow \Delta x = (k_r - 1)x \\ u &= x + (k_r - 1)x \\ v &= y + (k_r - 1)y \end{aligned} \tag{3.15}$$

Otra optimización posible a aplicar en el recorrido del cuadrante, es evitar barrer todas las posiciones dentro de la imagen, de modo que se recorren los cuadrantes conforme a los valores de los radios dentro de la imagen, para ello se necesita extraer la posición en la que cae la x, y dado un determinado valor de radio, el cálculo de dichas extrapolaciones de coordenadas se refleja en la ecuación 3.16.

$$\begin{aligned} r^2 &= \tfrac{x^2+y^2}{f^2} \\ y &= \sqrt{r^2 f^2 - x^2} \\ y &= 0 \rightarrow x = rf \end{aligned} \tag{3.16}$$

donde r es el radio y f la distancia focal, en la figura 3.9 se observa gráficamente cómo se debe de hacer el recorrido, se puede observar que computacionalmente es más complejo, pero se subsana con el ahorro de recorrido ya que k_r se calcula un menor número de veces.

En la correción de distorsión se pierde gran parte del tiempo realizando la evaluación de $k_r(r^2)$, esto se debe al cálculo de la curva para cada punto, puesto que el cálculo de los puntos en una curva se hace costoso, se puede realizar ahorro de tiempo interpolando dicha curva con múltiples rectas, esto produce una modificación en el resultado del algoritmo debido a la precisión de la interpolación, es necesario realizar el cálculo de los puntos de interpolación, este proceso puede ser parte del calibrado original, ya que dichos algoritmos no tienen la necesidad de hacerse en tiempo real si no en una fase previa. En el algoritmo 6 se presenta el

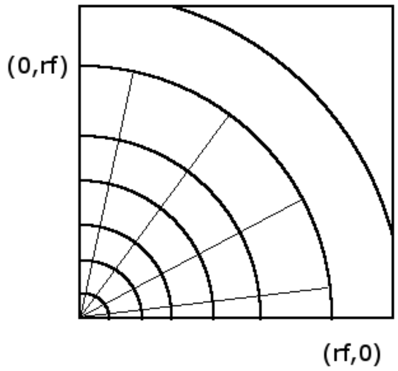


Figura 3.9 Representación de recorrido radial

pseudocódigo para el cálculo de k_r pertenencientes a la interpolación de las líneas.

```
Data: MatrizIntrínseca, ParametrosDistorción, MaxLíneas
Result: ValoresKR
focalInv = 1 / (MatrizIntrinseca[fx] * MatrizIntrinseca[fy]);
maxRad = (power(MatrizIntrinseca[cx], 2) * power(MatrizIntrinseca[cy], 2)) *
  focalInv;
rDelta = maxRad / MaxLineas;
ValoresKR = [] ValoresKR.append(1);
// el primer valor es 1
for r2 ← rDelta to maxRad do
    r2 += rDelta;
    ValoresKR.append((1 + ((KParametrosDistorcion[2]*r2 +
      KParametrosDistorcion[1])*r2 + KParametrosDistorcion[0])*r2) / (1 +
      ((KParametrosDistorcion[5]*r2 + KParametrosDistorcion[4])*r2 +
      KParametrosDistorcion[3])*r2));
    // Se aplica la ecuación para el cálculo de kr
    r2 += rDelta;
end
```

Algoritmo 6: Algoritmo de interpolación de líneas

En la figura 3.10 se observa cómo se realiza la interpolación de una curva con rectas, esta se puede hacer con la precisión que se desee aumentando el número de líneas, su modificación supone cambios a la hora de realizar el mapeo, pero son visualmente inapreciables, siendo la mejora en tiempo considerable.

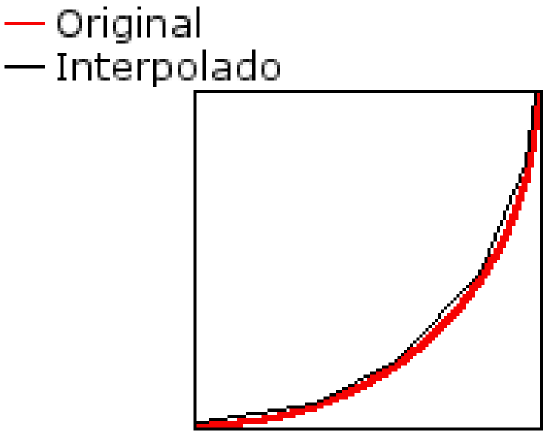


Figura 3.10 Interpolación de una curva con líneas

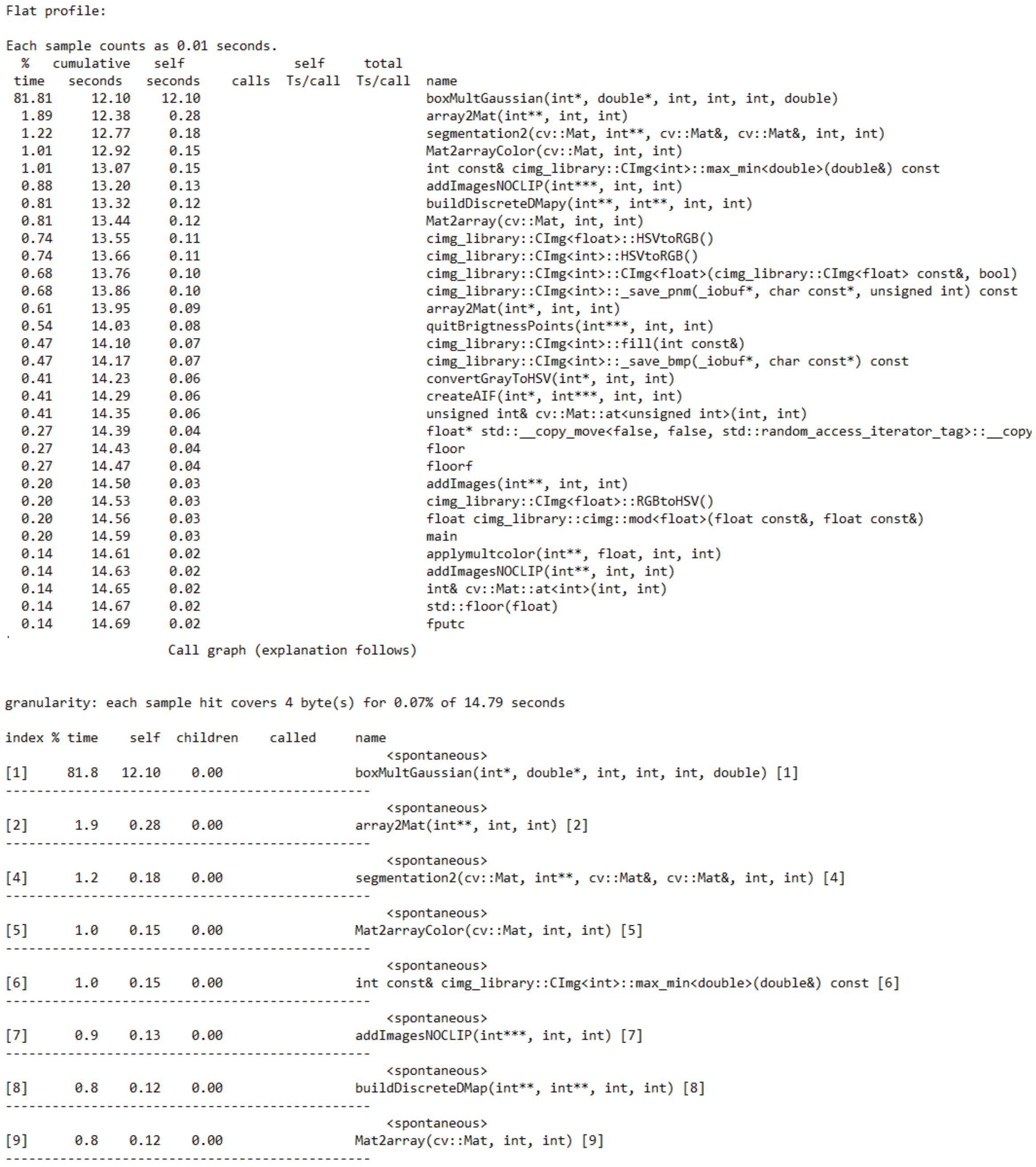
Capítulo 4

Resultados obtenidos

En el capítulo 3 se explican en detalle las optimizaciones realizadas en la librería desarrollada en este trabajo; se presentan las pruebas realizadas en las diferentes arquitecturas, junto con ciertas descripciones de optimización a nivel de Hardware realizadas.

Los tiempos que se muestran en las diferentes gráficas y tablas se han testeado con un número determinado de pruebas en las distintas plataformas, sobre las mismas condiciones, utilizando los mismos medidores de tiempos para comparar la librería respecto de OpenCV. Debido a que el uso de distintos medidores no provee una información certera debido a los posibles problemas de precisión Hardware, se procura para la medida de tiempos que el uso de CPU sea similar. Se ha escogido OpenCV como comparación debido a que es la que más se adapta a las características de la librería desarrollada en el trabajo, además como se observa en el Capítulo 2, es la librería general que mejores tiemops de ejecución ofrece, disponiendo ya esta de una serie de optimizaciones considerables a nivel de código.

Como herramienta para la toma de medidas se ha escogido una de licencia GNU, Gprof [32], el compilador gcc permite pasar una opción de depuración para el uso de esta herramienta, es bastante completa y compleja, realizando un análisis estadístico monitorizando el código para así poder centrarse en las funcionalidades y tiempos de código sin necesidad de medir los preámbulos, también muestra las veces que se realizan las llamadas a las funciones y los porcentajes de tiempo. En el caso del sistema operativo Android, se hace uso del profiler del NDK compatible con gprof [33]. En la figura 4.1 se observa un ejemplo de salida de la ejecución del *profiler* en un sistema UNIX, mostrando de izquierda a derecha el porcentaje total de tiempo, el tiempo de forma acumulativa, el tiempo por separado y la firma de la función que se ejecuta.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 81.81     12.10    12.10                             boxMultGaussian(int*, double*, int, int, int, double)
  1.89     12.38     0.28                             array2Mat(int**, int, int)
  1.22     12.77     0.18                             segmentation2(cv::Mat, int**, cv::Mat&, cv::Mat&, int, int)
  1.01     12.92     0.15                             Mat2arrayColor(cv::Mat, int, int)
  1.01     13.07     0.15                             int const& cimg_library::CImg<int>::max_min<double>(double&) const
  0.88     13.20     0.13                             addImagesNOCLIP(int***, int, int)
  0.81     13.32     0.12                             buildDiscreteDMapy(int**, int**, int, int)
  0.81     13.44     0.12                             Mat2array(cv::Mat, int, int)
  0.74     13.55     0.11                             cimg_library::CImg<float>::HSVtoRGB()
  0.74     13.66     0.11                             cimg_library::CImg<int>::HSVtoRGB()
  0.68     13.76     0.10                             cimg_library::CImg<int>::CImg<float>(cimg_library::CImg<float> const&, bool)
  0.68     13.86     0.10                             cimg_library::CImg<int>::_save_pnm(_iobuf*, char const*, unsigned int) const
  0.61     13.95     0.09                             array2Mat(int*, int, int)
  0.54     14.03     0.08                             quitBrigtnessPoints(int***, int, int)
  0.47     14.10     0.07                             cimg_library::CImg<int>::fill(int const&)
  0.47     14.17     0.07                             cimg_library::CImg<int>::_save_bmp(_iobuf*, char const*) const
  0.41     14.23     0.06                             convertGrayToHSV(int*, int, int)
  0.41     14.29     0.06                             createAIF(int*, int***, int, int)
  0.41     14.35     0.06                             unsigned int& cv::Mat::at<unsigned int>(int, int)
  0.27     14.39     0.04                             float* std::__copy_move<false, false, std::random_access_iterator_tag>::__copy
  0.27     14.43     0.04                             floor
  0.27     14.47     0.04                             floorf
  0.20     14.50     0.03                             addImages(int**, int, int)
  0.20     14.53     0.03                             cimg_library::CImg<float>::RGBtoHSV()
  0.20     14.56     0.03                             float cimg_library::cimg::mod<float>(float const&, float const&)
  0.20     14.59     0.03                             main
  0.14     14.61     0.02                             applymultcolor(int**, float, int, int)
  0.14     14.63     0.02                             addImagesNOCLIP(int**, int, int)
  0.14     14.65     0.02                             int& cv::Mat::at<int>(int, int)
  0.14     14.67     0.02                             std::floor(float)
  0.14     14.69     0.02                             fputc

                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.07% of 14.79 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     81.8   12.10    0.00                 boxMultGaussian(int*, double*, int, int, int, double) [1]
-----------------------------------------------
                                                 <spontaneous>
[2]      1.9    0.28    0.00                 array2Mat(int**, int, int) [2]
-----------------------------------------------
                                                 <spontaneous>
[4]      1.2    0.18    0.00                 segmentation2(cv::Mat, int**, cv::Mat&, cv::Mat&, int, int) [4]
-----------------------------------------------
                                                 <spontaneous>
[5]      1.0    0.15    0.00                 Mat2arrayColor(cv::Mat, int, int) [5]
-----------------------------------------------
                                                 <spontaneous>
[6]      1.0    0.15    0.00                 int const& cimg_library::CImg<int>::max_min<double>(double&) const [6]
-----------------------------------------------
                                                 <spontaneous>
[7]      0.9    0.13    0.00                 addImagesNOCLIP(int***, int, int) [7]
-----------------------------------------------
                                                 <spontaneous>
[8]      0.8    0.12    0.00                 buildDiscreteDMap(int**, int**, int, int) [8]
-----------------------------------------------
                                                 <spontaneous>
[9]      0.8    0.12    0.00                 Mat2array(cv::Mat, int, int) [9]
-----------------------------------------------
```

Figura 4.1 Resultados de ejecución de Gprof

Este capítulo se divide en 3 secciones, dos de ellas contemplando cada una de las optimizaciones nombradas en el capítulo 3, y otro valorando las mejoras en las distintas arquitecturas y haciendo una comparación de las mismas dando a conocer las ventajas y desventajas del trabajo elaborado.

4.1. Convolución

En este trabajo se ha hecho hincapié en las convoluciones, debido a que estas son una de las operaciones más utilizadas en el procesamiento de imagen, se han propuesto varias optimizaciones matemáticas para reducir la complejidad del algoritmo, pero vía Hardware se han utilizado además técnicas más específicas.

Al trabajar con imágenes en blanco y negro, se modifica la localidad de memoria de la imagen ya que en la mayoría de procesadores se disponen de operaciones vectoriales para trabajar con elementos de forma paralela. Para aprovechar dicha capacidad se genera una imagen de tamaño $\frac{(wxh)}{4}$ simulando una imagen de canales RGBA y así, trabajar con los 4 canales de forma simultánea, en estos casos se debe tener en cuenta el acceso espacial entre píxeles, dependiendo de la localidad de memoria, esta técnica de organización es más rápida. Para procesadores Intel con el set de instrucciones intrínsecas [34], se genera memoria alineada aprovechando así el set de instrucciones, la generación de memoria se realiza del siguiente modo:

```
int *matrixAligned = (int*) _aligned_malloc(
                                    width * heigth *
                                    sizeof(int), 32);
__m128i *intrinsic_vector = (__m128i*) out;
```

De esta manera se tiene un vector de memoria alineada para utilizar con instrucciones intrínsecas aprovechando los tipos de dato con lo que las unidades funcionales son capaces de trabajar a la vez, para NEON [35], es necesario copiar la memoria a sus vectores, para este caso se utiliza la instrucción siguiente:

```
uint32x4_t vector = vld1q_u32(matrix[i]);
```

En caso de OpenCL, la versión utilizada es la 2.0 [36], para la creación de imágenes se utliza la función *clCreateImage*, con formato *CL_RGBA*, en este caso se necesita tener bien definidos los tipos de datos, ya que la función necesita la memoria des-entrelazada y consecutiva para poder crear el almacenamiento en los dispositivos correspondientes.

En la figura 4.2 se observa una media de las optimizaciones realizadas de forma global para observar la aceleración matemática provista por los distintos algoritmos en las diferentes plataformas. En la gráfica de dicha figura se encuentran las mejoras propuestas para las convoluciones convencionales, dado el factor de aceleración que se produce es factible evaluar si las convoluciones son separables, mostrando los tiempos de ejecución de la función en la figura 4.3, donde se muestra un aumento de tiempo lineal, esta medida se ha realizado sobre el desarrollo del algoritmo 1.

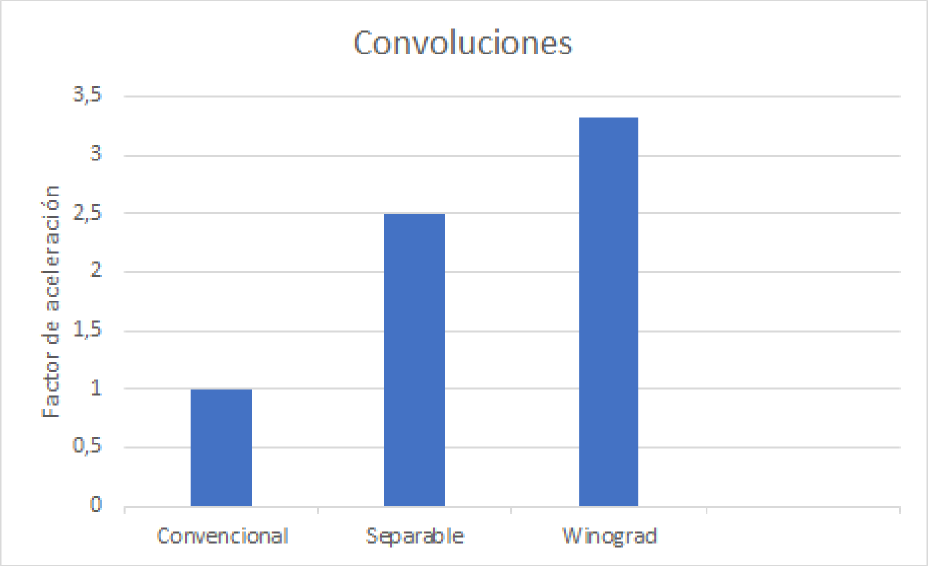


Figura 4.2 Aceleración matemática de las convoluciones

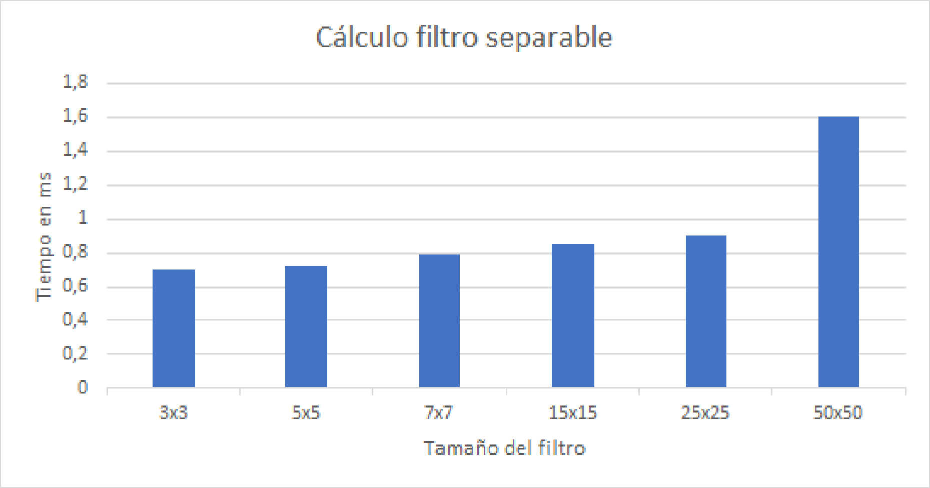


Figura 4.3 Comprobación y generación de filtros separables cuando el rango de la matriz es 1

En la tabla 4.1 se observa la comparación con la librería OpenCV, apreciando en los procesadores de teléfonos el mayor aumento de velocidad, esto viene dado por la especialización

Tabla 4.1 Comparación de las técnicas de optimización aplicadas a convoluciones en diferentes arquitecturas

Arquitecturas	OpenCV	Convencional	Separable	Winograd
Intel	1.0x	1.3x	2.5x	3.3x
ATI	1.0x	1.5x	2.6x	3.0x
NVIDIA	1.0x	1.7x	2.9x	3.2x
SNAPDRAGON	1.0x	2.1x	2.7x	3.7x

de Hardware, debido a que el código no es sólo portable si no específico por dispositivo. Para la implementación de Winograd es necesario realizar multiplicación de matrices, una primera aproximación se ha realizado utilizando una implementación rápida[37], pero debido a los accesos de localidad de memoria y la necesidad de realizar siempre las mismas multiplicaciones de matrices, los tiempos de cómputo eran elevados a raíz de esta pérdida. Para solventar el problema, se hace uso de técnicas de precompilado que preparan una serie de tamaños de filtro para ejecutar con Winograd (3x3 y 5x5), descomponiendo la multiplicación de matrices con valores constantes ya que estos no cambian en el mismo tamaño de convolución, esta técnica supone un gran ahorro debido a la introducción de los valores constantes en la memoria caché.

Los filtros de caja han sido objeto de estudio en el trabajo, su uso directo en aplicación a la imagen tiene el fin de suavizar, para este acción no se hace necesaria la ejecución de grandes tamaños de kernels, pero para cierto tipo de aplicaciones como puede ser el añadir a una imagen con un determinado mapa efecto Bokeh[38], se utilizan grandes tamaños de filtros. Las convoluciones separables en este caso no aumentan el tiempo de cómputo, por ello se han implementado las dos propuestas que se comentan en el capítulo 3. Se propone utilizar la técnica SAT para ejecutar cualquier tamaño de filtro en tiempo lineal, esto no es del todo cierto, la memoria dependiendo del planificador almacena en la memoria de acceso rápido (caché) una determinada cantidad de datos, en los que el acceso entre ellos es bastante rápido, en cambio el movimiento de bloques de la memoria principal a memoria caché es costoso. Esto implica que la técnica SAT no sea totalmente lineal, pero aún así es con diferencia la técnica más rápida, en la figura 4.4 se observa la aceleración media con las distintas técnicas probadas, el tamaño de kernel más grande para estas pruebas ha sido de 20x20, en el caso de aumentar el tamaño la técnica SAT aumenta el factor de aceleración.

Para la implementación en GPU de esta técnica se realiza primero la división por el factor total, teniendo así una imagen con valores flotantes, haciendo a posteriori la suma de los mismos, de este modo se aprovecha la división en el tiempo de cálculo de la integral, además se hace uso de las unidades funcionales de las GPUs en este caso, porque estos

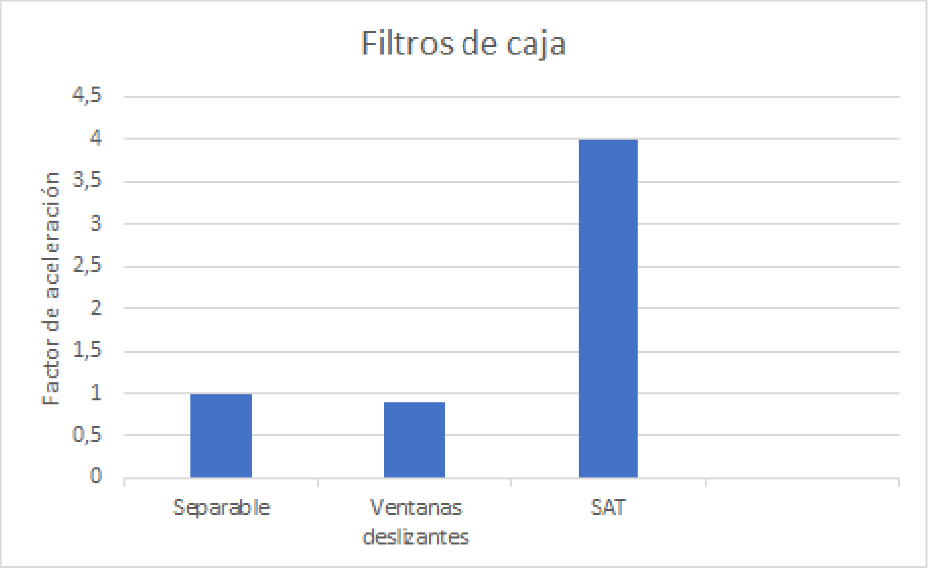


Figura 4.4 Aceleración sobre las optimizaciones propuestas

Tabla 4.2 Comparación de las técnicas de optimización aplicadas a filtros de caja en diferentes arquitecturas

Arquitecturas	OpenCV	Separable	Ventanas deslizantes	SAT
Intel	1.0x	1.2x	1.1x	2.8x
ATI	1.0x	1.3x	1.0x	4.3x
NVIDIA	1.0x	1.5x	1.1x	4.9x
SNAPDRAGON	1.0x	1.9x	1.5x	7.1x

dispositivos están pensandos para el procesamiento de gráficos, con lo que integran más unidades de trabajo con flotantes que con enteros. En la tabla 4.2 se muestra la comparación en las distintas arquitecturas, añadiendo la librería de OpenCV como base.

El uso de filtros no lineales se utiliza en aplicaciones médicas, y otros campos de estudio en los que la imagen se integra, su uso principal es la eliminación de ruido de la imagen sin perder los detalles de la misma. Se barajan varias implementaciones, la primera de ellas supone una modificación del algoritmo principal, siendo esta reflejada en la ecuación 3.10, las siguientes modificaciones son basadas en esta, intentando aprovechar las técnicas comentadas anteriormente a este filtro. A diferencia del SAT, la técnica de histogramas integrales no supone aumento de rendimiento, primero teniendo en cuenta el consumo de memoria que conlleva, en la figura 4.5 se observa la reducción de memoria que se obtiene al utilizar la

Tabla 4.3 Comparación de las técnicas de optimización aplicadas al filtro bilateral en diferentes arquitecturas

Arquitecturas	OpenCV	Eliminación de Gaussiana	Ventanas deslizantes	Histogramas integrales
Intel	1.0x	1.3x	2.1x	0.4x
ATI	1.0x	1.6x	2.7x	0.8x
NVIDIA	1.0x	1.7x	3.3x	0.9x
SNAPDRAGON	1.0x	1.7x	3.5x	0.8x

estructura de memoria presentada en el capítulo 3.

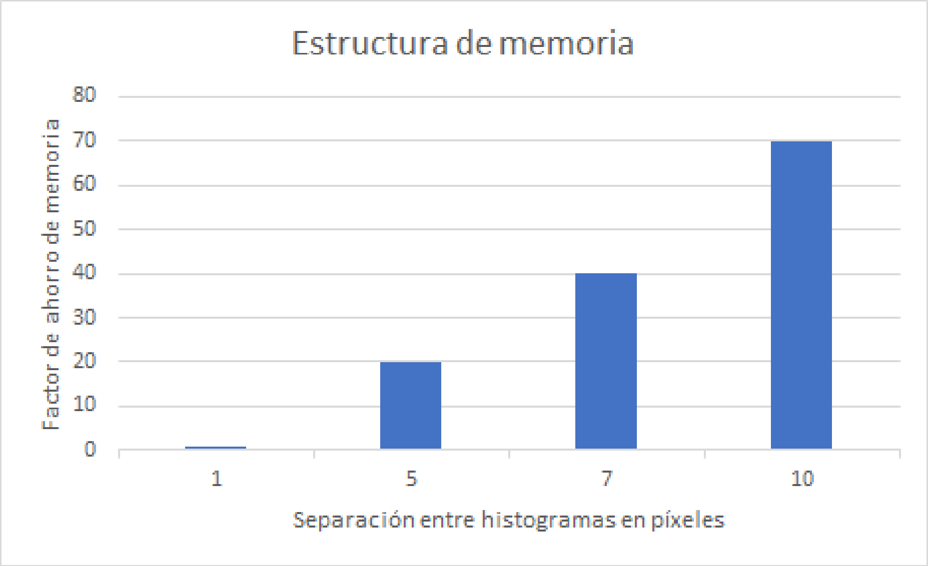


Figura 4.5 Ahorro de memoria en histogramas integrales frente al procesamiento convencional

Al igual que anteriormente, ambas implementaciones probadas utilizan tipos de datos vectoriales, en el caso de los histogramas integrales el tipo de dato a utilizar es de 16 bits en cada arquitectura para aprovechar la capacidad de cálculo, suponiendo que la imagen no debe tener más de 65.535 valores iguales. Para el uso de ventanas deslizantes se aprovechan tanto los desplazamientos verticales como horizontales, en la figura 4.6 se observa la optimización producida por los distintos métodos, reflejándose estos en la tabla 4.3 sobre las distintas arquitecturas.

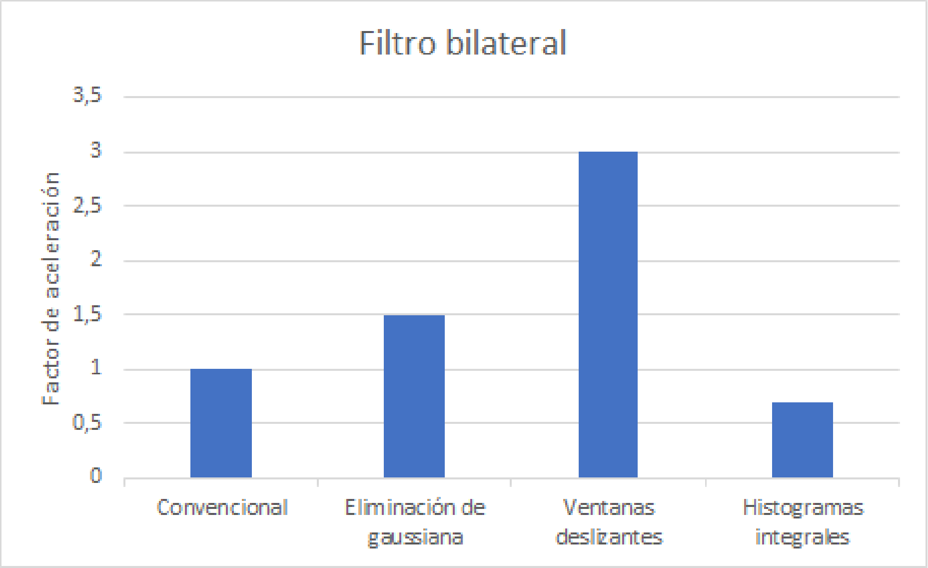


Figura 4.6 Optimizaciones sobre el Filtro Bilateral

4.2. Calibrado y corrección de la imagen

El calibrado de imagen, es una operación de uso continuo, ya sea para vídeo o captura. El procesamiento de imagen que se aplica sobre un frame con distorsión dan lugar a resultados irreales, por ejemplo, la transformada de Hough aplicada sobre una imagen con distorsión puede ocultar información de interés, ya que esta se basa en la obtención de líneas.

En la figura 4.7 se observan las distintas implementaciones realizadas de forma incremental, en este caso se hacen múltiples combinaciones de las optimizaciones ya que no son exclusive como ha sucedido hasta ahora. La naturaleza de este algoritmo es el trabajo con números flotantes, pero las CPUs utilizan menos ciclos de reloj para realizar operaciones en enteros, por este motivo una de las optimizaciones que se aplica es la transformación a entero, para ello se escalan los valores k_r intentando perder la menor precisión posible dentro del rango de los 32 bits. Por otro lado la generación de mapas de coordenadas se multiplica por dos, esto se hace para dejar la parte flotante almacenada en forma del resto de la división, de manera que al hacer el remapeo de coordenadas se obtengan con mayor precisión. Pese a la costosa operación que supone realizar un módulo, la optimización sigue siendo fructífera. La nomenclatura utilizada en la figura 4.7 es la siguiente:

- O1: Implementación común.
- O2: Implementación sin contemplar aberraciones tangenciales.

Tabla 4.4 Comparación de las técnicas de optimización aplicadas a la correción de distorsión en diferentes arquitecturas

Arquitecturas	OpenCV	O1	O2	O3	O4	O5	O6
Intel	1.0x	1.2x	1.4x	2.2x	2.3x	2.7x	3.4x
ATI	1.0x	1.1x	1.5x	2.4x	2.5x	2.7x	2.8x
NVIDIA	1.0x	1.2x	1.7x	2.5x	2.7x	2.9x	3.0x
SNAPDRAGON	1.0x	1.3x	1.9x	2.7x	3.0x	3.1x	3.2x

- O3: Aprovechamiento de simetría.
- O4: Implementación con interpolación de rectas.
- O5: Aprovechamiento del recorrido de los radios para el menor cálculo de valores K_r.
- O6: Optimización de enteros.

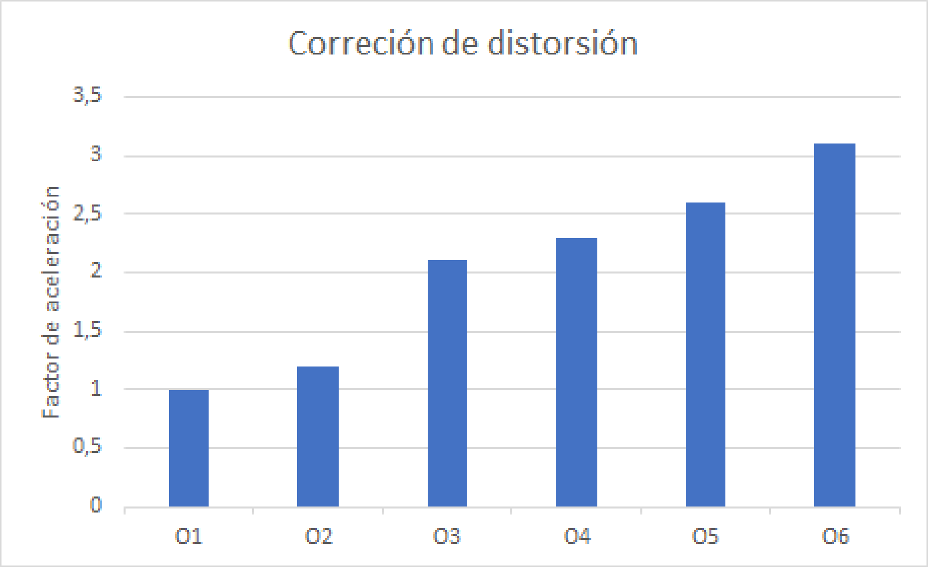


Figura 4.7 Optimizaciones sobre la correción de distorsión

La combinación de las optimizaciones efectúa un mejor rendimiento del algoritmo, aunque cabe destacar que las optimizaciones por separado ya suponen mejora. La optimización a nivel programatical es de mayor efectividad con el uso de enteros, notándose mejor porcentaje de mejora en las CPUs, en la tabla 4.4 se reflejan dichos resultados.

4.3. Comparación

Tras la valoración e implementación de las distintas optimizaciones se observa en las diferentes tablas que las GPUs de dispositivos móviles son las que se proveen de una mayor aceleración frente a las funciones de OpenCV, esto se debe a su novedoso uso, ya que dichas tecnologías son actuales, siendo la implementación de OpenCV más antigua. En cuanto a implementaciones de OpenCL la librería de licencia libre no dispone de una versión para los distintos dispositivos, esto se refleja en las comparaciones. Cabe destacar que las implementaciones que ofrece OpenCV están mayoritariamente pensadas para CPUs, y su gran envergadura hace difícil la elaboración e inclusión de nuevos algoritmos.

Las optimizaciones realizadas para Snapdragon son más estables debido a que la arquitectura es similar entre las diferentes versiones, a diferencia de las gráficas de NVIDIA y ATI. Por otro lado, se observan esos tiempos de ejecución debido a que la transferencia entre CPU y GPU es prácticamente inmediata ya que los dispositivos móviles disponen de una única memoria RAM compartida, en cambio las GPUs añadidas a ordenadores tienen su propia memoria, teniendo que realizar transferencia de datos entre las mismas.

Aunque el aumento en factor de velocidad en los procesadores móviles sea mayor, los tiempos de ejecución son menores que en las arquitecturas NVIDIA y ATI, no sucediendo lo mismo con los procesadores INTEL. Uno de los factores a tener en cuenta es la libertad que provee el sistema operativo para el control de recursos, en este caso Android permite disponer al usuario de una gran cantidad de recursos a diferencia de los procesadores INTEL.

En cuanto a la calidad de las optimizaciones se aprecia que el caso de los histogramas integrales no supone mejora, esto se debe a las grandes cantidades de memoria que maneja, este tipo de optimizaciones son más eficaces en procesadores vectoriales de alto rendimiento, los que en este trabajo no se ha tenido disposición para realizar las pruebas pertinentes.

Capítulo 5

Conclusiones y líneas futuras

En este capítulo se muestran las conclusiones obtenidas durante el transcurso del trabajo, además de posibles ideas para la continuación de desarrollo del mismo.

5.1. Conclusiones

En los capítulos anteriores, se hace alusión a la metodología aplicada para el desarrollo del trabajo, se pueden observar junto con la revisión del estado del arte las distintas mejoras. La elaboración de una librería es un proceso costoso, el cual debe pasar por una serie de pasos a seguir para obtener un resultado final satisfactorio. Es de vital importancia el realizar un diseño adecuado para la fluidez de desarrollo, y por otro lado dicho diseño no se puede exceder tanto en tiempo, como programaticamente, debido a que al aumentar el nivel de abstracción la gestión de recursos se hace menos eficaz.

Ante el diseño desarrollado se puede decir que la introducción de nuevos algoritmos resulta sencilla, siendo tediosa la labor de implementar en cada una de las arquitecturas por separado, la necesidad de disponer de documentación para el uso de distintas operaciones específicas implica grandes tiempos de búsqueda y ajuste de las mismas. Los lenguajes estándar son de gran utilidad para el ahorro de tiempo en búsqueda de operaciones, aunque se dispone de la dependencia del fabricante para el uso de la misma, no siendo esto posible en todas las arquitecturas.

Las optimizaciones aplicadas son satisfactorias, debido al aumento considerable de tiempos frente a librerías del mercado. El estudio de las distintas técnicas de optimización es un proceso laborioso, en el que surge la necesidad de investigar en cada uno de los campos por separado, realizando pruebas múltiples y observando el comportamiento de cada uno de los algoritmos para ver las posibles mejoras.

La serie de mejoras realizadas no abarca la propuesta inicial del trabajo en su totalidad, esto se debe al tiempo que conlleva el desarrollo de cada una de las técnicas en las distintas plataformas, y la búsqueda de optimizaciones para cada una de las técnicas constituye un trabajo muy complejo. Cabe destacar que la realización de código dependiente de la plataforma ya supone en sí un ahorro de cómputo, aunque este pudiera ser mayor encontrando las mejoras oportunas para los algoritmos.

Se logra por otro lado la inclusión en distintas plataformas, a excepción de disponibilidad para algunas concretas como arquitecturas MALI [39] o MIPS.

En rasgos generales, se ha cumplido con gran parte de los objetivos, debido a que se han desarrollado las condiciones básicas de realizar la implementación en arquitecturas múltiples y una serie de optimizaciones válidas para cualquier dispositivo.

5.2. Líneas futuras

Dada la complejidad del trabajo, surge un amplio abanico de ramas a investigar. Las técnicas de procesamiento en visión por computador contienen una amplia gama de operaciones, no siendo todas ellas abordadas, cabe la posibilidad de realizar optimizaciones sobre los distintos algoritmos de segmentación, valorando el estado del arte de los mismos [40]. También es posible mejorar tanto en velocidad como en calidad, algoritmos de detección de características en múltiples imágenes como el SIFT [41]. En cuanto a algoritmia, en relación con el uso actual de redes neuronales [42], es posible añadir un sistema de ejecución de las mismas, sin introducir la fase de entrenamiento, dado que este es un proceso complejo y no reiterativo.

Por otro lado, cabe la posibilidad de utilizar otros estándares, como pueden ser Halide [43] o Vulkan [44], se caracterizan por una mayor capacidad de portabilidad ante OpenCL, ofreciendo algunos de ellos un desarrollo a más bajo nivel como Vulkan, o a más alto nivel como Halide.

Un campo novedoso de peculiar interés son los procesadores NPU(Neural Processing unit), diseñados para la ejecución de redes neuronales en dispositivos de bajo poder computacional con la intención de ofrecer la ejecución fluida de redes neuronales debido al aumento de aplicaciones con este tipo de técnicas. Las librerías de visión por computador en desarrollo incluyen operaciones de redes neuronales, lo que supone un gran avance y mejora de la librería presentada en el trabajo, pudiendo utilizar las optimizaciones propuestas para las convoluciones.

Bibliografía

[1] J. Tompson and K. Schlachter. An introduction to the opencl programming model. 2012.

[2] A. Cedilnik. Cross-platform software development using cmake. 2013.

[3] Arzoo M. Lakhwani, Krupali H. Shah, and Ankita S. Vaghela et al. Review on basics of computer vision and its applications. 2017.

[4] O. Özyesil, V. Voroninski, R. Basri, and A. Singer. A survey on structure from motion. *CoRR*, 2017.

[5] Noor A. Ibraheem and Pramod K. Mishra Mokhtar M. Hasanand Rafiqul Z. Khan. Understanding color models: A review. *ARPN Journal of Science and Technology*, 2012.

[6] W. Qi, F. Li, and L. Zhenzhong. Review on camera calibration. In *2010 Chinese Control and Decision Conference*, pages 3354–3358, 2010.

[7] J. Chen, K. Benzeroual, and Robert S. Allison. Calibration for high-definition camera rigs with marker chessboard. 2017.

[8] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 62–66, 1979.

[9] M. Van den Bergh, X. Boix, G. Roig, and Luc J. Van Gool. SEEDS: superpixels extracted via energy-driven sampling. *CoRR*, 2013.

[10] H. J. Kim, J. Y. Lee, J. H. Kim, J. B. Kim, and W. Y. Han. Object recognition and pose estimation using klt. In *2012 12th International Conference on Control, Automation and Systems*, pages 214–217, 2012.

[11] R. I. Hartley. In defence of the 8-point algorithm. In *Proceedings of IEEE International Conference on Computer Vision*, 1995.

[12] G. P. Fickel, C. R. Jung, R. Samadani, and T. Malzbender. Stereo matching based on image triangulation for view synthesis. In *2012 19th IEEE International Conference on Image Processing*, pages 2733–2736, 2012.

[13] A. McAndrew. *An Introduction to Digital Image Processing with Matlab*. 2004.

[14] Tomasi, Carlo, and T. Kanade. Detection and tracking of point features, computer science department. 1991.

[15] Qualcomm Computer Vision. FastCV 1.7.1 SDK. https://developer.qualcomm.com/software/fastcv-sdk, 2018. [Online].

[16] L. Codrescu et al. Hexagon dsp: An architecture optimized for mobile multimedia and communications. 2012.

[17] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. Realtime computer vision with opencv, communications of the acm. 2012.

[18] A. Syschikov, B. Sedov, K. Nedovodeev, and S. Pakharev. Visual development environment for openvx. 2017.

[19] ImageR Library. http://dahtah.github.io/imager/. [Online].

[20] CImg Library. http://cimg.eu/. [Online].

[21] VXL Library. http://vxl.sourceforge.net/. [Online].

[22] R. Bala and Kapil K. Kaswan. Strategy design pattern. 2012.

[23] Stencel, Krzysztof, Wegrzynowicz, and Patrycja. Implementation variants of the singleton design pattern. pages 396–406, 11 2008.

[24] Doxygen. http://www.stack.nl/~dimitri/doxygen/. [Online].

[25] G. Strang. *Introduction to Linear Algebra, Fifth Edition*. 2016.

[26] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch. What is the fast fourier transform? *Proceedings of the IEEE*, 55(10):1664–1674, 1967.

[27] A. Lavin. Fast algorithms for convolutional neural networks. *CoRR*, abs/1509.09308, 2015.

[28] X. Li and X. G. Xia. A fast robust chinese remainder theorem based phase unwrapping algorithm. *IEEE Signal Processing Letters*, 15:665–668, 2008.

[29] A. Kasagi, T. Tabaru, and H. Tamura. Fast algorithm using summed area tables with unified layer performing convolution and average pooling. In *2017 IEEE 27th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6, 2017.

[30] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, and Frédo Durand. A gentle introduction to bilateral filtering and its applications.

[31] F. Porikli. Integral histogram: a fast way to extract histograms in cartesian spaces.

[32] Graham, Susan L., Kessler, Peter B., Mckusick, and Marshall K. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.

[33] NDK Profiler. https://developer.oculus.com/documentation/mobilesdk/latest/concepts/mobile-ndk-profiler/. [Online].

[34] Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE,SSE2. [Online].

[35] Neon programmer's guide. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0018a/index.html. [Online].

[36] Opencl specifications. https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_Cxx.html. [Online].

[37] Naohito Nakasato. A fast gemm implementation on the cypress gpu. 38:50–55, 03 2011.

[38] T. Mcgraw. Fast bokeh effects using low-rank linear filters. 31, 05 2014.

[39] Mali graphics processing. https://www.arm.com/products/graphics-and-multimedia/mali-gpu?tab=Resources. [Online].

[40] J. Kuruvilla, D. Sukumaran, A. Sankar, and S. P. Joy. A review on image processing and image segmentation. In *2016 International Conference on Data Mining and Advanced Computing (SAPIENCE)*, pages 198–203, March 2016.

[41] K. Li and S. Zhou. A fast sift feature matching algorithm for image registration. In *2011 International Conference on Multimedia and Signal Processing*, volume 1, pages 89–93, May 2011.

[42] A.A. Chojaczyk, A.P. Teixeira, L.C. Neves, J.B. Cardoso, and C. Guedes Soares. Review and application of artificial neural networks models in reliability analysis of steel structures. *Structural Safety*, 52:78 – 89, 2015.

[43] Halide languaje. http://halide-lang.org/. [Online].

[44] Vulkan. https://www.khronos.org/vulkan/. [Online].