
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

# Algoritmos de imagen y sonido digital con restricciones de tiempo real

Tesis Doctoral

Presentada por:
Francisco Javier Alventosa Rueda
Dirigida por:
Pedro Alonso Jordá
Adolfo Martínez Usó

Valencia, 24 de noviembre de 2021

# Resumen


Introducción En la actualidad, cada vez existen más y más tareas que necesitamos exportar y automatizar en dispositivos portables de bajo consumo que se alimentan de baterías, en los cuales es imprescindible realizar un uso "óptimo" de la energía disponible con la finalidad de no drenarlas rápidamente.

## Parte I: Filtros de señales de audio digital

En esta sección "optimizamos" las implementaciones de diferentes filtros, tanto generales como específicos, para aplicaciones de sonido digital diseñados e implantados en plataformas basadas en las arquitecturas ARM®. Como filtros generales, trabajamos con los filtros FIR, IIR y Parallel IIR, siendo este tipo de filtros implementados a bajo nivel con instrucciones vectoriales **NEON®**. Finalmente, se implementa un filtro de separación de señales conocido como "Beamforming", el cual plantea después de su estudio, la problemática de realizar una factorización QR de una matriz relativamente grande en tiempo real, lo cual nos lleva a desarrollar diferentes técnicas de "aceleración" de los cálculos de la misma.

## Parte II: Rellenado de mapa de profundidad de una escena

En la sección de imagen por computador, describimos el proceso de rellenado de un mapa de profundidad de una escena capturada a partir del uso de la imagen RGB y de un mapa de profundidad disperso donde únicamente tenemos valores de profundidad en los bordes de los objetos que componen la escena. Estos algoritmos de "rellenado" del mapa de profundidad, también han sido diseñados e implantados en dispositivos basados en la arquitectura ARM®.

# Resum

INTRODUCCIÓ Actualment, cada vegada existixen més i més tasques que tenen la necessitat d'exportar i automatitzar a dispositius portables de baix consum que s'alimenten amb bateríes, als quals és imprescindible realitzar un ús "óptim" de l'energia disponible amb la finalitat de no drenar-les ràpidament.

## Part I: Filtres de senyals d'àudio digital

En aquesta secció "optimitzarem" les implementacions de diferents filtres, tant generals com específics, empreats a aplicacions de so digital disenyats e implantats a plataformes basades a les arquitectures ARM®. Com a filtres generals, treballem amb els filtres FIR, IIR y Parallel IIR, sent aquests tipus de filtres implementats a baix nivell amb instruccions vectorials **NEON®**. Finalment, s'implementa un filtro de separació de senyals conegut com "Beamforming", el qual planteja després del seu estudi, la problemàtica de realitzar una factorització QR d'una matriu relativament gran en temps real, i açò ens porta a desenvolupar diferents tècniques "d'acceleració" dels càlculs de la mateixa.

## Part II: Emplenat del mapa de profunditat d'una escena

A la secció d'image per computador, descrivim el procés d'emplenat d'un mapa de profunditat d'una escena capturada fent servir l'image RGB i un mapa de profunditat dispers on únicament tenim valors de profunditat als bordes dels objetes que composen l'escena. Aquests algoritmes "d'emplenat" del mapa de profunditat, també han sigut disenyats e implantats a dispositius basats en l'arquitectura ARM®.

# Abstract

INTRODUCTION Currently, there are more and more tasks that we need to export and automate in low-consumption mobile devices that are powered by batteries, in which it is essential to make an "optimum" use of the available energy in order to do not drain them quickly.

## Part I: Filters of digital audio signals

In this section we "optimize" the implementations of different filters, both general and specific, for digital sound applications designed and implemented on platforms based on the ARM®. As general filters, we work with the FIR, IIR and Parallel IIR filters, these types of filters being implemented at a low level with **NEON®** vector instructions. Finally, a signal separation filter known as "Beamforming" is implemented, which set out after its study, the problem of performing a QR factorization of a relatively large matrix in real time, which leads us to develop different techniques of "acceleration" of the calculations of it.

## Part II: Filling the depth map of a scene

In the computer image section, we describe the process of filling in a depth map of a captured scene using RGB image and a sparse depth map where we only have depth values at the edges of the objects that make up the scene. These depth map "filling" algorithms have also been designed and implemented in devices based on the ARM® architecture.

# Agradecimientos

Primero de todo quiero agradecer a mi mujer Tamara Reig, por toda la paciencia que ha tenido conmigo durante todos estos años de duro trabajo hasta haber podido finalizar esta tesis y presentarla; así como en estos últimos meses de redacción de la misma, para mi hijo Hugo por las horas que papá no te ha podido dedicar, pero no te preocupes que las recuperaremos en los próximos meses.

También agradecer al doctor D. Antonio Manuel Vidal Maciá por darme la oportunidad de formar parte de su grupo de investigación INCO2, incorporarme al mismo durante mis estudios de master, así como haberme asignado como director de tesis a D. Pedro Alonso Jordá, el cual ha sido mi mayor apoyo en conjunto con D. Adolfo Martínez Usó. Gracias al esfuerzo de ellos dos y al trabajo desinteresado que han realizado conmigo, he podido alcanzar el sueño de poder obtener el grado de Doctor en Informática.

Dar las gracias a la empresa photonicSENS® por haberme dado la oportunidad de integrarme en su proyecto novedoso y que me ha formado profesionalmente y como persona, además de haberme apoyado y ayudado a descubrir que soy capaz de realizar otras tareas de ingeniería además de las de desarrollador software.

Agradecer también el apoyo de mis padres, hermana y familia durante estos largos años de formación, hasta por fin alcanzar la meta.

Finalmente, dar las gracias también a todos y cada uno de mis profesores durante todos estos años de estudio y formación en la Universidad, así como a todos mis compañeros de promoción, pues de todos ellos he obtenido aprendizajes y vivencias que me han enriquecido profesionalmente y me han ayudado en el desarrollo de mi personalidad.

GRACIAS A TODOS!

# Índice general

# Capítulo 1

# Introducción

Un gran porcentaje de los dispositivos electrónicos fabricados en la actualidad como *smartphones*, *tablets*, *smart-TV*, etc., incorporan un procesador **ARM®**. Estos procesadores no solo proporcionan un gran ratio ***GFlop/Watt*** sino que, además, son los encargados de mover infinidad de aplicaciones, entre ellas el reconocimiento facial o la realidad virtual y aumentada. Dichas aplicaciones a menudo necesitan de la estimación de la profundidad de los objetos para su correcto funcionamiento.

## 1.1 Visión general

Los cambios producidos en las tecnologías digitales y el gran avance de estas en los últimos años nos afectan a todos los ciudadanos sin excepción. Los medios masivos de comunicación como los portales de noticias web, las redes sociales, los trámites burocráticos, transacciones económicas, etc., están en nuestro consumo diario. La línea que separa nuestro *mundo real* de nuestro *mundo online* se vuelve cada vez más delgada y, con ello, pasa muchas veces desapercibida la infraestructura que hay detrás para hacer operativas nuestras costumbres diarias y, muy especialmente relacionado con esta tesis, la inmensa capacidad de cómputo que demandamos para desarrollar esta automatización de las actividades cotidianas.

Esta influencia cada vez mayor de la tecnología en nuestras costumbres produce un aumento considerable en el número de los elementos electrónicos de los que hacemos uso diariamente, como son los *smartphones*, las *tablets*, las *tvs*, los *smart watchs*, etc., produciéndose un incremento considerable en la demanda de cómputo que requerimos de ellos para llevar a cabo cada vez tareas más complejas.

La problemática surge, además, por el hecho de que este incremento de capacidad de cómputo sobre todos estos dispositivos electrónicos lleva también aparejado un incremento en el consumo energético. Por si esto fuera poco, muchos de estos dispositivos electrónicos son alimentados por baterías, por lo que tienen una capacidad de almacenar energía "muy limitada". Como bien es conocido en el campo de la electrónica de consumo, la principal problemática de la industria actual es el diseño y fabricación de baterías con una mayor capacidad de almacenamiento, sin que por ello se vea incrementado en demasía el tamaño y peso de los dispositivos. Todo lo cual hace, más que necesario, imprescindible, el

desarrollo de aplicaciones que hagan un uso óptimo de dicha energía.

Las problemáticas planteadas añaden la necesidad de crear una arquitectura que tenga en cuenta estas premisas, es decir, el "optimizar" el ratio ***GFlop/watt***[1] de energía consumido. De esta nueva necesidad surge la empresa ARM®, la cual se dedica al desarrollo de arquitecturas computacionales "optimizadas" para ser incorporadas en dispositivos que precisan de un consumo reducido, especialmente los que se alimentan de baterías con la finalidad de no drenarlas rápidamente, sin renunciar a una potencia de cómputo adecuada a las necesidades de la sociedad actual. La característica principal de las arquitecturas desarrolladas por la empresa ARM® es el extraordinario ratio ***GFlop/watt*** que desarrollan. Proporcionando un consumo energético módico aparejado a una capacidad de cómputo nada desdeñable, las arquitecturas ARM® se han popularizado de tal forma que cada vez son más los fabricantes que las licencian para el diseño y fabricación de sus nuevas unidades computacionales, que lanzan anualmente al mercado para un sin fin de productos de electrónica de consumo.

Además de esta característica tan importante en los dispositivos que funcionan a batería como es el ratio de **operaciones aritméticas realizadas por segundo/consumo energético**, es decir, el ratio ***FLOPS/watt***, en los últimos años también han proliferado las unidades computacionales de sistemas embebidos. Este tipo de unidades incorporan dentro de un mismo *chip* físico diferentes unidades computacionales tales como CPU, GPU ó DSP[2], conectadas a una misma memoria principal compartida, característica que les permite un intercambio de información entre las diferentes unidades presentes de forma muy eficiente, pues se reduce el tiempo de intercambio de la información haciendo uso de buses de comunicación más cortos y rápidos.

Gracias a los avances en *hardware* embebido que se están llevando a cabo en los últimos tiempos, se ha desarrollado esta tesis doctoral realizando un desarrollo de algoritmos tanto de audio digital (1D), como de imagen digital (2D), aprovechando las características de los mismo, para poder ejecutar dichas aplicaciones en dispositivos móviles que incorporan este tipo de unidades embebidas.

A continuación, presentaremos las dos líneas de investigación desarrolladas en esta tesis y el estado del arte en la materia. Las dos líneas de investigación son: a) el desarrollo de filtros para señales unidimensionales 1D, como son las señales de audio digital y, por otra parte, b) el tratamiento de imagen digital 2D, enfocado al cálculo del mapa de profundidad [28] de una escena 2D capturada por una cámara compuesta por una óptica "especial"[3].

## 1.2 Introducción 1D - Procesamiento de señales sonoras

Un filtro es un dispositivo que, aplicado a un señal de entrada, nos permite "limpiarla" y/o mejorar ciertas características seleccionadas de la misma. Existen dos tipos de filtros de señales, los filtros analógicos y los filtros digitales. Los filtros analógicos son aquellos que se componen de circuitos

[1] **GFlop: Número de operaciones en coma flotante. Watt: Vatios de consumo energético.**

[2] Un **DSP** (Digital Signal Processor) o en español, procesador de señales digitales es una unidad de cómputo "optimizada" para tareas que requieran realizar operaciones a muy alta velocidad.

[3] Óptica compuesta por un conjunto de lentes capaz de calcular la distancia a la que se encuentran los objetos que componen la escena capturada.

electrónicos compuestos por resistencias, condensadores, amplificadores y otros elementos que sean necesarios para realizar la funcionalidad deseada. En cambio, los filtros digitales realizan operaciones matemáticas sobre los datos de entrada de la señal muestreada, bien sea haciendo uso de una unidad computacional genérica como una CPU, o bien haciendo uso de unidades computacionales específicas como un DSP o una FPGA[4].

Una forma de ver un filtro es como el dispositivo capaz de discriminar aquello que pasa a través de él. Como ejemplo ilustrativo, podemos poner el de un filtro de aire, que es capaz de dejar pasar a través de él únicamente el aire, separando las partículas de polvo u otras sustancias que este pueda contener.

En esta tesis nos centraremos en los filtros digitales, en los cuales el filtrado es un proceso que modifica el espectro de frecuencias de una señal o, lo que es lo mismo, la distribución de amplitudes para cada una de las frecuencias que componen la señal. Este tipo de filtros es muy común en el procesamiento de señales especialmente en el campo de las Telecomunicaciones, y han dado lugar a un gran número de investigaciones en la materia al poseer propiedades que los hacen adaptables a un gran número de aplicaciones. En el momento de desarrollo de esta tesis no se encontraron trabajos similares a éste utilizando instrucciones vectoriales NEON® de los procesadores ARM® para un novedoso *chip*, como era en el momento el modelo TK1 desarrollado por la compañía NVIDIA®. Este hecho fundamentó el desarrollo de este trabajo con la finalidad de explorar el rendimiento de las unidades de cálculo vectorial presente en los dispositivos ARM®, y analizando el número máximo de filtros junto con el número de coeficientes de los mismos que podían ser aplicados en un lapso de tiempo acotado.

El Capítulo 3, que forma parte de esta primera parte de la tesis, se basa en el algoritmo del *Beamforming*. Este algoritmo posee como idea fundamental en la cual se basa, el empleo de patrones de dirección de las señales, permitiendo asegurar la recepción de señales provenientes de una ubicación dada, y atenuando el resto de señales presentes en el medio y que no aportan nada a la señal que se desea recibir. Como hemos indicado, es un tipo de filtrado espacial, pues es capaz de detectar la dirección de procedencia de las distintas señales sonoras y "apuntar" en la dirección de la señal que interesa "limpiar" atenuando el resto.

Si bien el algoritmo del *Beamforming* está muy extendido en el campo de las telecomunicaciones y posee diferentes variantes del mismo, las cuales han dado lugar a un gran número de investigaciones, estas siempre habían sido desarrolladas sin tener restricciones de cómputo y haciendo uso de supercomputadores con una ingente capacidad de cómputo y memoria y sin, además, restricciones de consumo energético. El interés de llevar este tipo de algoritmo a equipos móviles de bajo consumo se sustenta en la idea de poder disponer de algoritmo de "separación" de señales en dispositivos móviles de bajo consumo de forma que puedan ser utilizados por los usuarios en cualquier lugar y momento. Un ejemplo gráfico de la utilidad de estos algoritmos en dispositivos móviles consistiría en que los participantes a una ponencia pudieran escuchar de forma "clara" y "limpia" al ponente, aunque tuviéramos otros sonidos de fondo o gente hablando a nuestro alrededor.

[4]Una **FPGA** (Field-programable gate array, o en español Matriz de puertas lógicas programables en el campo, es un dispositivo *hardware* de cálculo programable por un lenguaje de descripción especializado.

Después del desarrollo de los trabajos propuestos en esta parte de la tesis se fueron actualizando los resultados con los nuevos NVIDIA® Jetson® que se lanzaban al mercado, como son los TX1, TX2, Xavier, etc., que nos ha permitido actualizar algunas gráficas y análisis de resultados de esta parte de la tesis.

## 1.3 Introducción 2D - Procesamiento de imágenes

En imagen por computador, el mapa de profundidad de una escena es una matriz 2-dimensional (2D) que almacena un valor de profundidad (distancia) para cada uno de los píxeles que forman la escena entre la lente de la cámara y los diferentes objetos que forman la imagen. En la actualidad, la industria demanda cada vez más estos mapas de profundidad para desarrollar un gran número de aplicaciones, como puede ser el reconocimiento facial [54], las aplicaciones de realidad aumentada [22] y otras aplicaciones de tratamiento de imagen como por ejemplo el efecto *bokeh* [42].

En el mercado compiten diferentes tecnologías capaces de obtener el mapa de profundidad de una escena, como son: Luz estructurada (*Structure Light*, iPhoneX y superiores), Tiempo de vuelo (*Time of Flight*, Kinect), imagen estéreo (RealSense D435 de Intel), cámaras de campo de luz (Plenóptic cameras, Lytro) y el Lidar (iPhone y IPad de última generación) [67]. Si bien todas ellas son capaces de obtener el mapa de profundidad, no es menos cierto que la mayoría de ellas precisan de más de un elemento electrónico o cámara para obtenerlo, aumentando así el dispendio en tecnología y energía consumida.

Cada una de estas tecnologías es capaz de obtener el mapa de profundidad de la escena con mayor o menor acierto dependiendo del tipo de escena y las condiciones de la misma, en base a sus virtudes o defectos. Es posible para todas ellas y dada una escena de carácter general, obtener un mapa de profundidad de una calidad aceptable dependiendo de la aplicación a la que lo queramos enfocar, si bien suelen perder bastante precisión para distancias muy cercanas, a excepción de la tecnología plenóptica.

A raíz de la necesidad de reducir el número de elementos que componen estos sistemas para el cálculo de profundidad de la escena y con la finalidad de reducir costes de producción, ha sido desarrollada por la empresa photonicSENS® una novedosa cámara que, basada en una única lente y un único disparo de cámara, es capaz de extraer la información de profundidad a la que se encuentran los diferentes elementos que componen la escena. Por si esto fuera poco, dicha tecnología también es capaz de medir con precisión distancias muy cercanas, en el orden de milímetros, proporcionándole esta característica ventajas competitiva claves respecto a la competencia, pues la habilita, por ejemplo, para entrar a formar parte de procesos de fabricación que precisan de una precisión milimétrica.

## 1.4 Objetivos de la tesis

La realización de la presente tesis nace como respuesta a la necesidad de optimizar y llevar a dispositivos móviles dos aplicaciones muy demandadas y presentes en la sociedad actual. La primera

aplicación se sustenta en el desarrollo de implementaciones de filtros digitales de sonido con restricciones de tiempo real y energía consumida. La segunda aplicación nace de la necesidad de desarrollar una tecnología "pasiva" capaz de obtener el mapa de profundidad de una escena capturada por dicha tecnología.

En base al estado del arte en la materia mencionado en los dos apartados anteriores e identificadas las necesidades y oportunidades para realizar propuestas tecnológicas originales, se han elaborado los objetivos que se enuncian a continuación:

- Adaptar algoritmos existentes para ser eficientes en consumo energético y ser implantados principalmente en dispositivos portables que funcionen a batería.
- Desarrollar algoritmos de computación heterogénea capaces de obtener el máximo partido a las nuevas arquitecturas computacionales embebidas de los dispositivos móviles actuales.
- Vectorización *Ad-Hoc* y manual de algoritmos genéricos de filtrado de señales de audio empleando el conjunto de instrucciones vectoriales de los dispositivos embebidos ARM® comercializadas bajo la nomenclatura de NEON®.
- Utilizar diferentes implementaciones de librerías de álgebra lineal densa y dispersa, con la finalidad de optimizar los algoritmos desarrollados en número de operaciones y tiempo computacional.
- Adaptar algoritmos existentes y hacerlos portables a diferentes plataformas *hardware*, haciendo uso del standard de programación multiplataforma **OpenCL**.
- Implementar un algoritmo del estado del arte en la separación de señales de sonido digital como es el *Beamforming* para ser adaptado a las restricciones de computo y memoria de los dispositivos embebidos actuales.
- Desarrollar e implementar algoritmos de imagen digital capaces de realizar el "rellenado" de un mapa de profundidad incompleto de una escena capturada por una cámara novedosa de la empresa photonicSENS®. Estos algoritmos deben ser capaces de correr en dispositivos móviles que funcionen con arquitecturas Qualcomm® y en tiempo real (30 *frames* por segundo).

## 1.5 Entorno de investigación del doctorando durante el desarrollo de la presente tesis

En esta sección haremos una pequeña presentación de los entornos de trabajo en los que se ha desarrollado esta tesis: el grupo de investigación de la Universidad Politécnica de Valencia y la empresa photonicSENS®.

El grupo de investigación con el que se trabajó en la primera parte de la tesis está formado por investigadores de la Universidad Politécnica de Valencia pertenecientes al Departamento de Sistemas Informáticos y Computación y al Departamento de Comunicaciones, además de investigadores

pertenecientes a otras universidades que desarrollan parte de su actividad con este grupo. Aunque la formación de sus miembros es diferente y complementaria (titulaciones en Informática, Física y Matemáticas, principalmente) los investigadores del grupo tienen mayoritariamente especialización en ciencias de la computación. El objetivo principal del grupo es la solución de problemas numéricos complejos, en particular aquellos que pueden ser resueltos por computadores paralelos y técnicas de computación de alto rendimiento, así como el modelado y optimización de sistemas computacionales en paralelo. La investigación de este grupo se organiza en torno a tres campos que se encuentran relacionadas entre sí: Computación y Arquitecturas de Alto Rendimiento aplicadas a problemas de Álgebra Lineal Numérica, Computación Paralela Heterogénea y Computación de Alto Rendimiento aplicada a problemas de ingeniería, en general, y de comunicaciones, en particular.

Por su parte, photonicSENS® es una empresa pionera en el desarrollo de un producto capaz de proporcionar información 3D de la escena con un único disparo de cámara. El nacimiento de esta empresa y de su tecnología viene derivada de la necesidad que tiene el ser humano de percibir la profundidad de las escenas en las capturas de imágenes hechas con los dispositivos móviles que poseen. Esta tecnología pretende impulsar la existencia de nuevas aplicaciones de inmersión para el consumidor, como aplicaciones personalizadas de AR (*Augmented Reality*) y VR (*Virtual Reality*), nuevas aplicaciones de navegación y mapeo y nuevas formas de editar contenido digital. La empresa photonicSENS® es pionera en un módulo de detección de profundidad 3D de lente simple para permitir la experiencia de usuario móvil de próxima generación al recrear la capacidad del ojo humano para juzgar con precisión la distancia, permitiendo el desarrollo e implantación de un gran número de aplicaciones que requieran de información 3D para funcionar.

El trabajo del doctorando durante esta tesis ha estado vinculado, en una primera fase, al grupo de investigación de la Universidad Politécnica de Valencia, el relacionado con la temática de filtrado para señales de audio (Capítulos 2 y 3). En una segunda fase (Capítulos 4, 5 y 6), el doctorando se ha encontrado trabajando como desarrollador de software en la empresa photonicSENS®, en la cual ha desarrollado los algoritmos necesarios para la obtención del mapa de profundidad de la escena capturada por la novedosa cámara ApiCam®, además de participar activamente en la redacción y defensa de diferentes patentes europeas que permitan salvaguardar las ventajas competitivas de los productos desarrollados por la empresa. Existe una relación manifiesta entre ambas partes de la presente tesis pues, aunque en una parte se trabaja con algoritmos para señales de audio y en la otra con imagen digital, los algoritmos 2D son una extensión de los algoritmos 1D, y el trabajo desarrollado por el doctorando en ambos escenarios es similar dado que en ambos casos se han desarrollado algoritmos heterogéneos, haciendo uso de librerías de álgebra lineal y optimizando *kernels* computacionales básicos, y en ambos casos se ha empleado la arquitectura ARM®, presente en muchos de los dispositivos electrónicos de consumo actuales.

## 1.6 Estructura de la tesis

La presente tesis se enmarca dentro del campo de la computación paralela, más concretamente de la computación heterogénea; los algoritmos desarrollados persiguen el objetivo de "extraer" el máximo potencial de los diferentes elementos de cómputo presentes en los dispositivos *hardware* utilizados, haciendo especial énfasis en el *hardware* dedicado para dispositivos portables, como son las arquitecturas Jetson® de NVIDIA® o Snapdragon® de Qualcomm®.

Después de la introducción en la materia llevada a cabo en las anteriores secciones, la tesis se divide en dos partes bien diferenciadas. En el primer capítulo de la primera parte (Capítulo 2) se exponen filtros para señales de sonido digitales comunes en el campo de las telecomunicaciones, como son los filtros FIR, IIR y PIIR. El desarrollo de los algoritmos para la paralelización del cómputo de estos filtros se realiza a bajo nivel, empleando las instrucciones para cálculos vectoriales NEON®. El siguiente capítulo expone el diseño, implementación y paralelización de un algoritmo propio, el algoritmo del *Beamforming*. Este algoritmo es capaz de separar diferentes señales sonoras que han sido mezcladas en un canal de comunicación. El diseño del algoritmo ha sido pensado para que pueda ser implantado y utilizado en dispositivos móviles con una baja capacidad de cómputo comparada con la de un supercomputador actual.

Seguidamente, en la segunda parte de la tesis, aparecen tres capítulos. En el primero de ellos, el Capítulo 4, se exponen y presentan dos patentes desarrolladas por la empresa photonicSENS® con la colaboración fundamental del doctorando, que son la base tecnológica y algorítmica en la que se basan los algoritmos desarrollados para el rellenado del mapa de profundidad de una escena. Los algoritmos son presentados con detalle en el capítulo que le sigue, el Capítulo 5. Para finalizar con esta parte de algoritmos de tratamiento de imagen digital, se muestra, en el Capítulo 6, la cámara miniaturizada patentada por la empresa photonicSENS®, así como una comparativa de los resultados de los mapas de profundidad obtenidos por esta nueva tecnología comparada con las tecnologías del estado del arte en la materia.

La tesis termina con un capítulo de conclusiones y líneas de investigación abiertas para el futuro.

# Capítulo 2

# Filtros para aplicaciones de sonido digital

Los filtros de señales de audio digital pueden ser clasificados en dos grupos: aquellos que presentan una respuesta al impulso finita o **FIR**, y aquellos con una respuesta infinita al impulso o **IIR** y su subvariante **PIIR** (Parallel **IIR**). En este capítulo ofrecemos una implementación eficiente de bajo nivel de ambos tipos de filtro para procesadores de tipo ARM®.

## 2.1 Introducción a los filtros para señales digitales

Existen dos grupos principales en los que podemos clasificar un filtro según la manera en la que se implementan, lo que, a su vez, depende de la aplicación que le vayamos a dar: los **filtros analógicos** y los **filtros digitales**. Los filtros analógicos pueden ser clasificados, a su vez, como **pasivos** o **activos**. Los **filtros analógicos pasivos** son aquellos en los que, para su diseño, se han empleado elementos pasivos tales como *resistencias*, *capacidades* e *inductancias* como, por ejemplo, los filtros de tipo RLC. Por su parte, los **filtros analógicos activos** son aquellos que incorporan algún elemento activo para su correcto funcionamiento como pueden ser, por ejemplo, los amplificadores operacionales (filtros OPAMP).

Los filtros digitales, que son los tratados en esta tesis, son sistemas lineales invariantes en el tiempo (**LTI**–Linear Time Invariant). Siendo $x$ el vector que contiene la señal de entrada, $b$ el vector que contiene los coeficientes del filtro e $y$ el vector que contiene la señal de salida, $x$ e $y$ son de longitud $N$, mientras que $b$ es de longitud $M$ (tamaño del filtro), siendo $n$ la muestra i-ésima, donde $0 \leq n \leq N$ y $m$ el coeficiente i-ésimo, donde $0 \leq m \leq M$. Los filtros de este tipo modifican el espectro en frecuencia de las señales de entrada $x(n)$, según la respuesta a la frecuencia en la que hayan sido configurados $b(n)$ (conocida popularmente en este campo como **función de transferencia**), dando lugar a una salida con un espectro $y(n) = x(n) * b(m)$. En cierto modo, el vector de coeficientes del filtro $b(m)$ actúa como una *función de ponderación* o *función de conformación espectral* para las diferentes frecuencias

que componen la señal de entrada $x(n)$.

Los sistemas **LTI** son clasificados en dos grandes grupos: los filtros **FIR** (Finite Impulse Response), que se caracterizan por ser sistemas no recursivos, es decir, para el cálculo del siguiente valor de la señal de salida únicamente se tiene en cuenta el valor actual de la señal de entrada, en otras palabras, no existe retroalimentación con los valores anteriores de la señal; y los filtros **IIR** (Infinite Impulse Response), que sí poseen retroalimentación para el cálculo de la señal de salida en base a los valores de salida anteriormente calculados, además de tener en cuenta los valores de entrada de la señal sin procesar.

En esta tesis nos centramos en los filtros digitales anteriormente nombrados, además de trabajar sobre una modificación de los filtros **IIR** que permite paralelizar el cálculo de los mismos y, de esta forma, poder mejorar las prestaciones. Esta versión la denominaremos **PIIR** (*Parallel* **IIR**).

## 2.2 Tipos de filtro

En los siguientes apartados se explican con detalle los Filtros **LTI** caracterizados por una salida, que es combinación lineal de la entrada, y por la existencia de unos coeficientes que son invariables en el tiempo. La aportación de la tesis en este capítulo consiste en la implementación por software de filtros digitales eficientes sobre dispositivos de bajo consumo. En nuestro caso, estos dispositivos están representados por el procesador **ARM®** y, en particular, las implementaciones a las que ha dado lugar este trabajo están basadas en la utilización de instrucciones vectoriales **NEON®**.

### 2.2.1 Filtros tipo FIR

Como ya hemos expuesto con anterioridad, un filtro **FIR** es aquel que tiene una respuesta al impulso finita y que no tienen retroalimentación de valores de salida anteriores (son sistemas no recursivos).

Un filtro **FIR** de orden $M$ se describe mediante la ecuación en diferencias siguiente:

$$y(n) = x(n) * b_0 + x(n-1) * b_1 + x(n-2) * b_2 + ... + x(n-(M-1)) * b_{M-1},$$

donde la secuencia $[b_k]_{k=0}^{M-1} = b[k]_{k=0}^{M-1}$ está compuesta por los coeficientes del filtro $b$, $x$ e $y$ son, respectivamente, las señales de entrada y de salida. Utilizando esta ecuación de diferencias podemos obtener la función de transferencia del filtro $F(z)$ en el dominio $Z$ (*transformada* **z**).

$$F(z) = \sum_{k=0}^{M-1} b[k] z^{-k}. \tag{2.1}$$

En este tipo de filtros digitales no existe retroalimentación. Además, la respuesta al impulso de este filtro es finita, pues si la señal de entrada alcanza el valor 0 y se mantiene en este valor durante un periodo de $M$ muestras consecutivas, la salida de este filtro también será 0. Algunas de las características más importantes de este tipo de filtros son su estabilidad (tienen el valor 0 en el plano complejo) o su

fácil comprensión e implementación.

En la Figura 2.1 se muestra de forma gráfica la interpretación de un filtro **FIR** de orden $M$ que se extrae directamente de la ecuación en diferencias (2.1). A este modo de implementación se le conoce con el nombre de *implementación directa* de un filtro **FIR**. Como se puede observar en dicha figura, la composición del filtro se basa fundamentalmente en una línea de elementos de retardo, multiplicadores y sumadores en un número igual al orden del filtro ($M$).

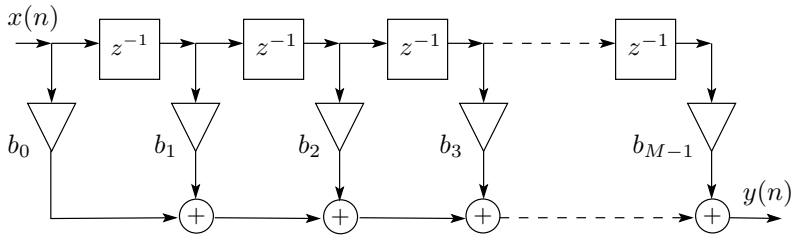



Figura 2.1: Arquitectura de un filtro de tipo **FIR**.

Se puede llevar a cabo una ligera variación de esta estructura, conocida como *estructura directa transpuesta*. Esta se deriva de la implementación directa, a la cual, únicamente se le tiene que modificar el orden de los elementos de retardo de la señal de entrada invirtiendo el orden de los coeficientes del filtro $b$, tal como muestra la Figura 2.2. Esta estructura invertida suele ser la forma predeterminada para implementar filtros tipo **FIR**, debido a que esta estructura invertida elimina la necesidad de introducir una etapa de segmentación adicional para implementar el árbol de sumadores en *cascada*, que se convierte en obligatoria cuando se trata de conseguir la mayor tasa de transferencia (*throughput*) para el filtro.

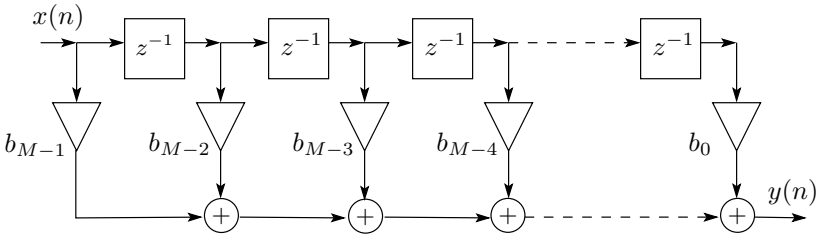



Figura 2.2: Arquitectura de un filtro de tipo **FIR** invertido.

En las Figuras 2.1 y 2.2 se muestran dos de los esquemas más empleados para la representación de un filtro **FIR** pero, en realidad, existen muchas estructuras equivalentes para el diseño de este tipo de filtro que realizan la misma función de transferencia. En la implementación mediante software de un filtro **FIR** hay que tener en cuenta que los computadores operan con aritmética de precisión finita, por lo que podemos obtener resultados diferentes dependiendo de qué estructura implementemos. Es

por esto que debemos seleccionar una estructura que minimice los efectos sobre la *cuantización*[1] de las señales.

Debido a lo expuesto con anterioridad, en la literatura sobre este tipo de filtros, se han propuesto diferentes estructuras, y en cada una de ellas se persiguen diferentes compromisos entre precisión, velocidad y tamaño. Además de las mencionadas anteriormente en las Figuras 2.1 y 2.2, otras estructuras utilizadas frecuentemente son, por ejemplo, las conocidas como *estructuras en celosía*, la *descomposición paralela de secciones de primer y segundo orden*, etc [52].

### 2.2.2 Filtros tipo IIR

Otro tipo de filtros típicos para el tratamiento de señales digitales son los filtros **IIR** (Infinite Impulse Response) de respuesta infinita al impulso. Este tipo de filtro tiene un estructura recursiva, recibiendo retroalimentación de los valores calculados con anterioridad para el calculo del valor actual. Es por ello, que además de los coeficientes de filtro $b$ para los datos de entrada $x$, se tiene otros coeficientes de filtro $a$ para los datos de salida $y$ anteriores, pues como hemos comentado es un filtro retroalimentado. Su ecuación en diferencias puede expresarse como:

$$y[n] = \sum_{k=0}^{L-1} x[n-k] * b[k] + \sum_{k=1}^{L-1} y[n-k]a[k]. \tag{2.2}$$

Por tanto, la función de transferencia para los filtros **IIR** en función de la *transformada* **z** tiene la forma:

$$F(z) = \frac{\sum_{k=0}^{L-1} b[k]z^{-k}}{1 - \sum_{k=1}^{L-1} a[k]z^{-k}}. \tag{2.3}$$

Este tipo de filtro se caracteriza porque, una vez es *"excitada"* la señal, nunca vuelve la misma a un estado de *"reposo"* total, es decir, nunca vuelve a tener el valor 0. De igual modo que en los filtros **FIR**, la implementación directa de los filtros **IIR** puede ser obtenida de la ecuación en diferencias (2.2), tal y como se muestra en la Figura 2.3. A esta configuración de filtro **IIR** se le conoce como *Estructura Directa I*. Cabe recalcar que un filtro **IIR** puede interpretarse como una *cascada* de dos subsistemas compuestos por el numerador y el denominador de la ecuación (2.3): $N(z) \cdot \frac{1}{D(z)}$.

Una variante de la estructura anterior (con un número de retardos igual al orden del filtro seleccionado) se obtiene con el simple hecho de invertir ambos subsistemas de la siguiente forma: $\frac{1}{D(z)} \cdot N(z)$. Esta segunda estructura, representada en la Figura 2.4 y conocida como *Estructura Directa II*, es muy interesante pues, además de ser *canónica*[2], reduce el número de sumadores y es fácil obtener una implementación eficiente.

Como hemos mencionado, existen diferentes estructuras de filtros **IIR**. En esta tesis se ha analizado

[1] La "cuantización" de una señal es el proceso posterior al muestreo, mediante el cuál, se le asigna un valor discreto, usualmente digital binario, a la muestra.

[2] Una estructura se dice que es "canónica" si el número de retardos presentes en el diagrama de bloques coincide con el orden del sistema.

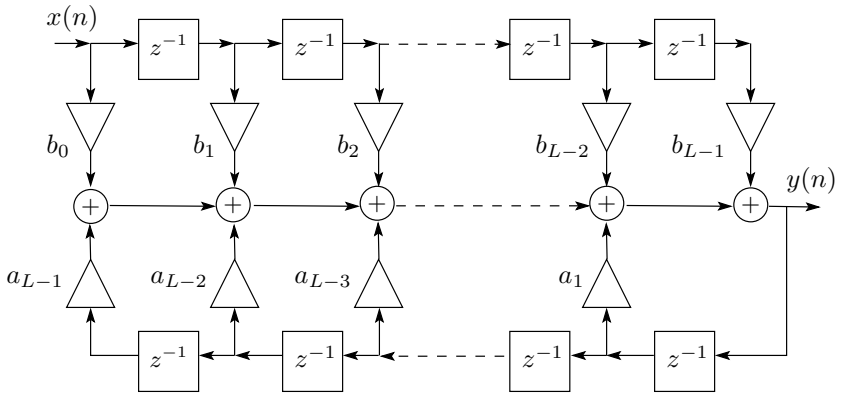



Figura 2.3: Arquitectura de un filtro de tipo **IIR** de la forma *Estructura Directa I*.

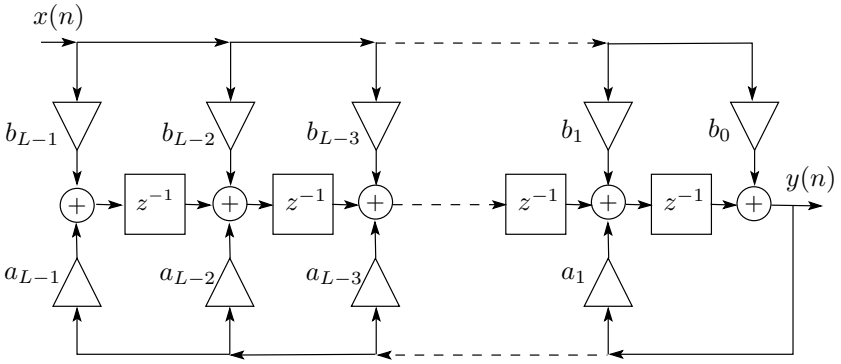



Figura 2.4: Arquitectura de un filtro de tipo **IIR** de la forma *Estructura Directa II*.

una *estructura generalizada* donde la relación de entrada-salida corresponde a la ecuación (2.2) y su representación funcional es la mostrada en Figura 2.5.

Además, tal y como ocurre en los filtros **FIR**, los filtros **IIR** también se pueden implementar utilizando estructuras en cascada y en paralelo de secciones de orden reducido, empleando estructuras *Lattice*, estructuras *Wave* y otras [52]. Las ideas y algoritmos desarrollados en esta tesis para este tipo de filtro pueden aplicarse a estas otras estructuras.

### 2.2.3 Filtros tipo PIIR

Para finalizar con la descripción de los filtros se analizan los *filtros paralelos*. Los filtros de tipo **IIR** de un orden elevado pueden fácilmente hacerse inestables a causa de los ajustes de retardo y distorsión, que pueden alterar los polos y ceros, lo que hace que los filtros sean inestables. Los filtros **IIR** paralelos, en particular los de segundo orden, poseen una menor sensibilidad a la cuantificación de coeficientes y un mejor rendimiento frente al ruido en comparación con los filtros de forma directa **IIR**. Para las

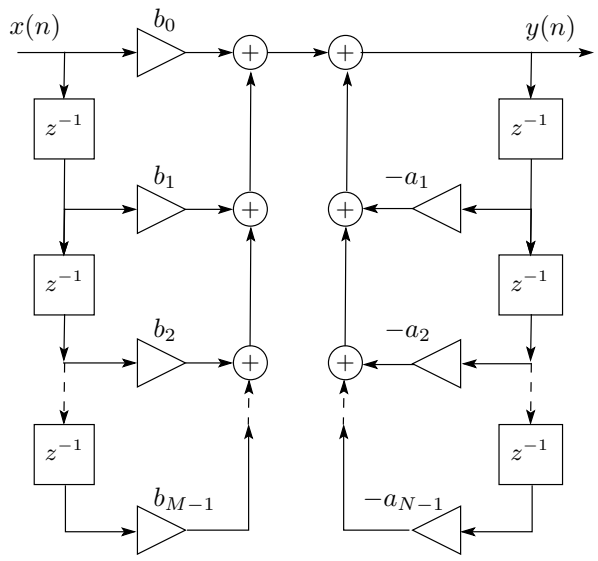



Figura 2.5: Arquitectura de un filtro de tipo **IIR** de la forma *Estructura Generalizada*.

aplicaciones donde se desea emplear un filtro de tipo **PIIR**, primero se diseña su correspondiente filtro **IIR** equivalente y luego se descompone en fracciones simples, agrupando los términos de primer y segundo orden entre sí con objeto de tener coeficientes reales. En la estructura propuesta, a diferencia de la estructura general presentada en [51], únicamente se presenta un coeficiente $d_0$ sin demora. La forma general del filtro paralelo consiste en un conjunto de secciones paralelas de segundo orden y un camino opcional de tipo **FIR** (la parte **FIR** se reduce a una sola ruta de señal) [57]. Para los filtros de tipo **PIIR** se utiliza la siguiente forma de representación:

$$H(z^{-1}) = \sum_{k=1}^{K} \frac{b_{k,0} + b_{k,1} z^{-1}}{1 + a_{k,1} + a_{k,2} z^{-2}} + d_0,$$

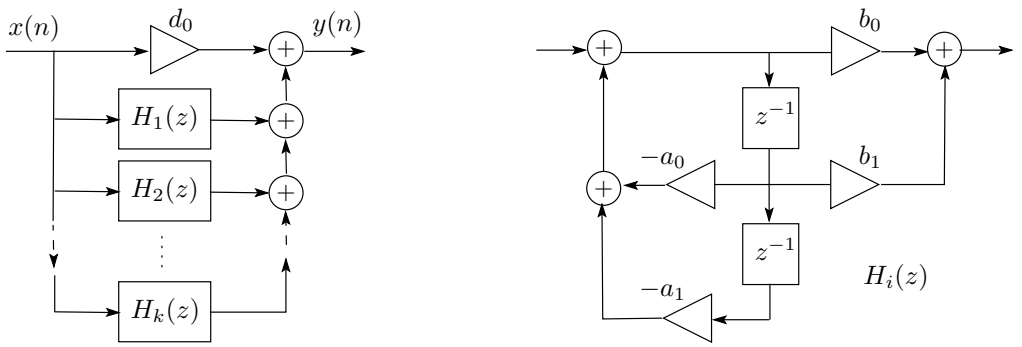
donde $K$ es el número de secciones de segundo orden.

En la Figura 2.6 está representado el filtro **PIIR**. La estructura general se muestra en la Figura 2.6a y en la Figura 2.6b una sección del filtro **PIIR** de segundo orden $H_i(z)$, para $i = 1, \ldots, k$.

### 2.2.4 Comparación entre filtros FIR e IIR

A la hora de elegir un tipo de filtro digital u otro se debe tener en cuenta la facilidad de los métodos de diseño y la flexibilidad para cambiar entre diferentes estructuras de implementación. En la actualidad se diseñan filtros **FIR** e **IIR** utilizando diferentes técnicas como la *ventana de Kaiser* para filtros **FIR** [38] o los *filtros de paso bajo de Butterworth* de tipo **IIR** [61].

Haciendo uso de un software avanzado y amigable, como puede ser la herramienta Matlab, se puede llevar a cabo un estudio preliminar experimental para determinar el mejor tipo y estructura de filtro




(a) Arquitectura del filtro.

(b) Una sección de segundo orden $H_i(z)$ del filtro.

Figura 2.6: Arquitectura de un filtro de tipo **PIIR**.

para una aplicación dada (analizando la cuantización de la entrada/salida, los coeficientes del filtro, los operadores de respuesta, etc.) y buscando el compromiso más adecuado entre las propiedades que son incompatibles entre sí.

Tanto los filtros **FIR** como los **IIR**, tienen ventajas e inconvenientes dependiendo de la aplicación a la que van destinados. Como conclusión general acerca de qué tipo de filtro es el más adecuado podemos indicar que, en rasgos generales, los filtros **IIR** son normalmente más eficientes que los **FIR** a la hora de conseguir ciertas características de calidad de la señal procesada para un mismo orden de filtro. Esta ventaja de los **IIR** viene derivada de la retroalimentación (tiene en cuenta las salidas anteriores a la hora calcular la actual) por lo que es capaz de operar en los *polos*[3] de la función de transferencia de un sistema mientras que los filtros **FIR** únicamente poseen *ceros*[4]. En ciertas aplicaciones, por ejemplo, para conseguir imitar el comportamiento de un filtro **IIR** de orden cuatro es necesario construir un filtro **FIR** de orden mucho mayor [58].

Algunas de las ventajas del uso de filtros **IIR** en lugar de filtros **FIR** son:

1. En la implementación de la función de transferencia los filtros **IIR** necesitan una menor cantidad de memoria y un menor número de instrucciones que los filtros **FIR**.

2. Los filtros **IIR** son diseñados mediante el cálculo de polos y ceros en el plano complejo. Esta característica permite que puedan ser implementadas funciones de transferencia que serían imposibles de implementar con los filtros **FIR**.

3. Es posible mover un filtro **IIR** a un modelo analítico.

Aunque también existen ciertas ventajas en el uso de filtros **IIR** frente a filtros **FIR**, se deben tener algunas consideraciones a la hora de utilizar los primeros:

[3] Valores que hacen cero el numerador de una fracción.
[4] Valores que hacen cero el denominador de una fracción.

1. Es necesario buscar la estabilidad del filtro durante el diseño de un filtro tipo **IIR**, pues estos no son necesariamente estables por definición.

2. Generalmente se tiene distorsión de fase con un filtro de tipo **IIR** al ser de fase no lineal.

3. Se debe contemplar el desbordamiento operacional en un filtro **IIR**, pues se basan en sumas de productos de una suma infinita.

Además de todas las anteriores ventajas y desventajas, hay que añadir aquellas que dependen de la implementación. En los apartados siguientes mostraremos cómo contribuye a esta dicotomía entre ambas estructuras fundamentales de filtro una implementación eficiente en dispositivos de bajo consumo.

## 2.3 Implementación de los filtros digitales

Con el objetivo de establecer una comparativa y *speed-up* homogéneo entre las diferentes implementaciones de los diferentes filtros, se ha decidido aislar dicha ganancia de las particularidades intrínsecas de los filtros. Es por ello que todas las pruebas con los diferentes filtros han sido realizadas con coeficientes aleatorios y sin generar filtros equivalentes para las diferentes estructuras **FIR**, **IIR** y **PIIR** utilizadas, es decir, se ha asignado en cada experimento el mismo orden a cada uno de los diferentes tipos de filtros sin atender a la necesidad de que, como hemos afirmado con anterioridad, un filtro **FIR** equivalente a uno **IIR** puede ser de un orden de hasta 100 veces superior en algunos casos.

Como hemos visto en el capítulo introductorio, los procesadores actuales disponen de varias unidades de procesamiento vectorial de tipo SIMD (Simple Instruction Multiple Data). Estas unidades de computo especial permiten realizar una misma operación sobre un conjunto de datos, generalmente múltiplos de 8 bits. Estas unidades pueden tener diferentes tamaños (128 bits, 256 bits, etc.) por lo que, dependiendo del tamaño de dichos registros vectoriales y del tipo de dato seleccionado para operar, serán capaces de aplicar la operación a un número de datos distinto. Por ejemplo, si disponemos de unidades vectoriales con registros de 128 bits, como es el caso del procesador ARM® Cortex-A15 de NVIDIA del *Jetson TK1*, podremos operar con cuatro datos en caso de utilizar datos reales de simple precisión (32 bits por dato), mientras que en caso de operar con datos reales de doble precisión (64 bits por dato), podremos operar únicamente sobre dos datos en un mismo ciclo. Para poder programar dichas unidades vectoriales los procesadores actuales poseen un juego de instrucciones "especial" catalogadas como SIMD. Este juego de instrucciones permite realizar una misma operación sobre un único conjunto de datos, bien sean de tipo entero o real. En el caso de los procesadores ARM®, este juego de instrucciones se le conoce con el nombre de **NEON®**.

### 2.3.1 Tecnología ARM® NEON®

La tecnología ARM® **NEON®** consiste en una unidad de procesamiento de datos SIMD que, junto a un juego de instrucciones avanzado para utilizarla, está presente en las series Cortex-A y las

series Cortex-R de procesadores de arquitectura ARM®. Esta tecnología fue introducida por primera vez en las arquitecturas ARM® versión 7 (ARMv7 y ARMv7-R) como una extensión a su juego de instrucciones escalares. En cambio, en las arquitecturas ARMv8, el juego de instrucciones SIMD se encuentra completamente integrado y ya no es una extensión como en las ARMv7.

Existen dos tipos de unidades para cálculos vectoriales en los procesadores ARM®. Por un lado, están las unidades vectoriales específicas de este tipo de procesadores, las unidades **NEON®** y, por otro lado, se encuentran las unidades genéricas para este tipo de operaciones o VFP (Vector Floating Point). En términos generales, las unidades **NEON®** procesan cálculos de una forma mucho más rápida que las unidades VFP pero, en cambio, las unidades **NEON®** no son capaces de realizar algunas operaciones matemáticas que sí es posible realizar con las unidades VFP como, por ejemplo, el cálculo de la raíz cuadrada de un número (función `sqrt`).

Las unidades **NEON®** de los procesadores ARMv7 y ARMv7-R, a diferencia de las unidades VFP, no cumplen completamente con el estándar IEEE-754 [44], por lo que un compilador no generará código haciendo uso de estas instrucciones a menos que sea especificado explícitamente por el programador que no se está interesado en el cumplimiento pleno del estándar anterior. Esto puede ser realizado de dos formas diferentes:

1. utilizando funciones intrínsecas para forzar el uso de **NEON®**, también llamadas instrucciones o funciones **NEON®**, o
2. delegar el trabajo en el compilador. En particular, el compilador GCC dispone de la opción `-mfpu=neon` para forzar la generación de código optimizado que haga uso de las **NEON®**.

Con la primera opción todo el esfuerzo de vectorización del código recae en el programador y debe ser este el que diseñe e implemente el nuevo algoritmo para el cálculo de los valores de forma vectorizada haciendo uso de las unidades **NEON®** disponibles. Para ello, se debe hacer uso explícito de las instrucciones **NEON®** seleccionando en cada caso las que mejor desempeñen el trabajo definido por el algoritmo previo.

Con la segunda opción es posible que las nuevas versiones de compiladores GCC no generen el código vectorial para unidades **NEON®** únicamente especificando la opción de compilación `-mfpu=neon` cuando se opera con datos en coma flotante y, entonces, sea necesario forzar la auto-vectorización que dé lugar a la introducción de instrucciones **NEON®** por parte del compilador añadiendo la opción de compilación `-funsafe-math-optimizations`.

La tecnología **NEON®**, entre otras cosas, intenta mejorar la experiencia multimedia de los usuarios acelerando la codificación y decodificación de audios y vídeos, interfaces de usuarios, gráficos 2D y 3D para juegos, etc. La tecnología **NEON®** puede también acelerar otros tipos de aplicaciones, como pueden ser las aplicaciones de *aprendizaje profundo* (*deep learning*).

### 2.3.2 NVIDIA *Jetson TK1*

Para llevar a cabo la experimentación de los filtros digitales implementados se empleó una placa de desarrollo con procesador embebido de bajo consumo de la compañía NVIDIA, en concreto, una placa

**Algoritmo 1:** Implementation of FIR filter processing using 16-bit integers

**Entrada: X**, **B**, **DESP**
**Salida: Y**

```
1: int32x4_t acum, int32x2_t res; int16_t *px; //Auxiliary variables
2: for i=0 to xysize:  //Iterates through the input and output data
3:   acum = vmovq_n_s32(0); px=x+i //Initialize auxiliary variables
4:   for j=0 to bsize: //Iterates vector B
5:     //Multiply and accumulate current inputs with vector B coefficents
6:     acum = vmlal_s16(acum, b[j], vld1_s16(px)); px += 4;
7:   res = vpadd_s32(vget_high_s32(acum),vget_low_s32(acum)); //Add pairwise
8:   res = vrshr_n_s32(res, DESP);  //Shift values of inter, DESP positions
9:   for j=0 to 1: y[i] += (int16_t)vget_lane_s32(res, j); //Store result
```

de desarrollo llamada *Jetson TK1* que incorpora el chip TK1 (Tegra K1). Este procesador embebido dispone de un procesador Quad-Core Cortex-A15 a 2,32 GHz y de una GPU de NVIDIA con 192 cores y arquitectura **Kepler**.

Los procesadores ARM® Cortex-A15 disponen de registros vectoriales de 128-bits de capacidad que son capaces de operar simultáneamente con hasta dos datos de 64-bits, cuatro de 32-bits, ocho de 16-bits ó 16 de 8-bits. En este estudio se han empleado los datos enteros de 32-bits (`INT32`) y 16-bits (`INT16`) y los reales de 32-bits (`FLO32`) o simple precisión con tres tipos de filtros digitales.

### 2.3.3 Implementación

Básicamente, la implementación de los filtros utiliza dos vectores principales: `x`, el cuál contiene las muestra de audio de la entrada; e `y`, el cuál contiene las muestras de audio de la salida una vez el filtro ha sido aplicado. Además, el vector `b` contiene los valores de los coeficientes del filtro a aplicar sobre los datos de entrada almacenados en el vector `x`. En el caso de los filtros de tipo **IIR** y **PIIR**, en ambos se tienen, además, el vector `a`, el cuál incluye los valores de los coeficientes que multiplican a los valores previos de salidas anteriores del vector `y`. En el caso del filtro **PIIR**, este también tiene dos vectores auxiliares para cálculos, como son los vectores `v1` y `v2`.

Los valores de los vectores `x`, `y`, `a`, `b`, `v1` y `v2` son almacenados en posiciones consecutivas de memoria para poder así obtener la máxima ganancia de rendimiento con las instrucciones SIMD. Estas permiten computar más de una operación a la vez, es decir, una misma operación sobre un conjunto de datos (el número de datos sobre los que opera es variable dependiendo del tipo de datos empleado).

En la implementación realizada en este trabajo, utilizamos tres registros vectoriales, dos de ellos son de solo lectura, mientras que el tercero es usado para escritura y acumular el resultado. Además, se declaran los vectores `a`, `b`, `v1` y `v2` con tipos de datos vectoriales para albergar los datos a operar. Por ejemplo, si trabajamos con datos enteros de 16-bits, el tipo de dato vectorial (tipo de datos **NEON®**) es `int16x4_t`, mientras que para datos reales de 32-bits, este tipo es el `float32x4_t`.

Cuando trabajamos con datos enteros los registros de acumulación empleados para cada tipo de dato es de un orden de magnitud superior, es decir, cuando trabajamos con datos enteros de 16-bits

**Algoritmo 2:** Implementation of IIR filter processing using a data type of 16-bit integer

**Entrada:** **X**, **B**, **A**, **DESP**
**Salida:** **Y**

```
1: int32x4_t acum, int32x2_t res; int16_t *px, *py; //Auxiliary variables
2: for i=0 to xysize:  //Iterates through the input and outut data
3:   acum = vmovq_n_s32(0); py=(y-asize*4)+i+1 //Initialize auxiliary variables
4:   for j=0 to asize: //Iterates vector A
5:     //Multiply and accumulate previous outputs with vector A coefficents
6:     acum = vmlal_s16(acum, a[j], vld1_s16(py)); py += 4;
7:   res = vpadd_s32(vget_high_s32(acum),vget_low_s32(acum)); //Add pairwise
8:   res = vrshr_n_s32(res, DESP);  //Shift values of inter, DESP positions
9:   for j=0 to 1: y[i] -= (int16_t)vget_lane_s32(res, j); //Store result
10:  acum = vmovq_n_s32(0); px=x+i //Initialize auxiliary variables
11:  for j=0 to bsize: //Iterates vector B
12:    //Multiply and accumulate current inputs with vector B coefficents
13:    acum = vmlal_s16(acum, b[j], vld1_s16(py)); px += 4;
14:  res = vpadd_s32(vget_high_s32(acum),vget_low_s32(acum)); //Add pairwise
15:  res = vrshr_n_s32(res, DESP);  //Shift values of inter, DESP positions
16:  for j=0 to 1: y[i] += (int16_t)vget_lane_s32(res, j); //Store result
```

**Algoritmo 3:** Implementation of PIIR filter processing using a data type of 16-bit integer

**Entrada:** **X**, **B1**, **A1**, **B2**, **A2**, **DESP**
**Salida:** **Y**

```
1: int32x4_t acum, int16x4 v0; int32x2_t res; //Auxiliary variables
2: for i=0 to xysize:  //Iterates through the input and outut data
3:   Y[i] = 0; acum = vmovq_n_s32(0); //Initialize auxiliary variables
4:   for j=0 to absize:  //Iterates through the vectors of coefficients
5:     v0 = vmov_n_s16(x[i]); //Load the same value (x[i]) in all positions
6:     v0 = vmls_s16(v0, a1[j], v1[j]); //multiply a1 & v1 & substract to v0
7:     v0 = vmls_s16(v0, a2[j], v2[j]); //multiply a2 & v2 & substract to v0
8:     acum = vmlal_s16(acum, b1[j], v0); //multiply b1 & v0 & accum. to acum
9:     acum = vmlal_s16(acum, b2[j], v1[j]); //multiply b2 & v1 & accum. to acum
10:    vst1_s16(&v2[j],v1[j]);  //Copy the value of v1[j] to v2[j]
11:    vst1_s16(&v1[j],v0);  //Copy the value of v0 to v1[j]
12:  end for
13:  res = vpadd_s32(vget_high_s32(acum),vget_low_s32(acum)); //Add pairwise
14:  res = vrshr_n_s32(res, DESP);  //Shift values of inter, DESP positions
15:  for j=0 to 1: y[i] += (int16_t)vget_lane_s32(res, j); //Store result
16:  y[i] += FIR*x[i]  //FIR is a constant value
```

estos registros de acumulación son de 32-bits, mientras que cuando trabajamos con enteros de 32-bits los acumuladores son de 64-bits. Esto se realiza así para evitar desbordamientos y la perdida de información cuando operamos con datos de tipo entero. Una vez operados los datos, se realiza el *shift* (desplazamiento $DESP$) correspondiente para quedarnos con el valor deseado, pues anteriormente se ha realizado el proceso inverso a la hora de convertir los valores reales a enteros (esto hay que ajustarlo entre la precisión del tipo de datos y los valores reales originales de los coeficientes de filtros y de los valores de entrada del mismo).

Para la inicialización de estos registros vectoriales usamos las instrucciones SIMD `vmovq_n_s32()` y `vmovq_n_s64()` que inicializan a cero las entradas respectivamente. A modo de ejemplo, la instrucción intrínseca `vmovq_n_f32()` es utilizada para inicializar a ceros los registros vectoriales reales de 32-bits.

Dependiendo de si trabajamos con datos enteros de 16-bits o de 32-bits, o con datos reales de 32-bits, los datos de entrada son cargados de memoria en registros vectoriales para operar con ellos haciendo uso de las instrucciones vectoriales de carga `vld1_s16()`, `vld1_s32()` y `vld1_f32()`, respectivamente.

Para las multiplicaciones y sumas, son utilizadas las instrucciones `vmlal_s16()`, `vmlal_s32()` y `vmlaq_f32()`, para cada uno de los tipos de dato correspondientes. Si los cálculos realizados se han llevado a cabo con el tipo de dato de entrada entero de 16-bits, entonces, se emplea la instrucción `vmovn_s32()` para convertir los datos intermedios de las operaciones de 32-bits a un resultado de enteros de 16-bits.

Antes de llevar a cabo las multiplicaciones, los bits que representan los valores de los coeficientes deben ser desplazados a la izquierda, de esta manera el valor del coeficiente queda multiplicado por un factor de escala [32] para convertir el número real en entero. Después de la multiplicación, se realiza un desplazamiento a la derecha del mismo orden. Esta operación de desplazamiento tiene por objeto incrementar la precisión de la multiplicación. El Algoritmo 1, el Algoritmo 2 y el Algoritmo 3, respectivamente, detallan las instrucciones vectorizadas según nuestra implementación basada en instrucciones **NEON®** para los tres tipos de filtro. Para interpretar mejor los algoritmos se muestra en la Tabla 2.1 una selección de instrucciones **NEON®** utilizada en los algoritmos con su descripción.

Para sumar parejas de datos almacenados en registros vectoriales se emplean dos instrucciones **NEON®**: `vpadd_lane_s32()` y `vpadd_lane_f32()` para enteros de 16-bit y reales de 32-bit, respectivamente. Por el contrario, no se ha utilizado ninguna instrucción **NEON®** para trabajar con enteros de 32-bits. En ambos casos, tanto con enteros de 16-bits como con reales de 32-bits, se deben cargar los datos en el registro vectorial como parte alta y baja empleando las instrucciones vectoriales para tal cometido. Se han empleado las instrucciones `vget_high_s32()` y `vget_low_s32()` para enteros de 16-bits y las instrucciones `vget_high_f32()` y `vget_low_f32()` para reales de 32-bits, para poder así preparar los datos correctos y operar con las instrucciones vectoriales de suma por parejas.

Finalmente, se han obtenido y sumado los valores intermedios guardados en registros vectoriales utilizando las instrucciones **NEON®** `vgetq_lane_s32()` y `vgetq_lane_f32()` para datos intermedios vectoriales enteros de 32-bits y reales de 32-bits, respectivamente. Para valores enteros de 64-bits se ha empleado la instrucción `vgetq_lane_s64()`.

## 2.4 Evaluación del rendimiento

Hemos evaluado nuestras implementaciones vectorizadas basadas en instrucciones **NEON®** para las tres estructuras de filtro descritas en los apartados anteriores: **FIR**, **IIR** y **PIIR**. Se han utilizado los tamaños de filtro de 52 y 256 coeficientes para los tres casos, pues son dos de los tamaños de coeficientes de filtro más utilizados en procesamiento de señales sonoras. Todos los códigos fueron compilados con el compilador de C de GNU utilizando las opciones `-O3`, `-mfpu=neon`, and `-march=armv7`. La Figura 2.7

Tabla 2.1: Instrucciones **NEON®** empleadas con su descripción.

| Instruction | Description |
|---|---|
| `vmovq_n_s32(value)` | Initialize `int32x4_t` vector registers |
| `vmovq_n_s64(value)` | Initialize `int64x2_t` vector registers |
| `vmovq_n_f32(value)` | Initialize `flo32x4_t` vector registers |
| `vmov_n_s16(value)` | Initialize `int16x4_t` vector registers |
| `vmov_n_s32(value)` | Initialize `int32x2_t` vector registers |
| `vld1_s16(*data)` | Load `int16x4_t` vector registers |
| `vld1q_s32(*data)` | Load `int32x4_t` vector registers |
| `vld1q_f32(*data)` | Load `flo32x4_t` vector registers |
| `vmls_s16(dest,data1,data2)` | Vector multiply subtract, `int16x4_t` dest. vector registers |
| `vmls_s32(dest,data1,data2)` | Vector multiply subtract, `int32x2_t` dest. vector registers |
| `vmlsq_f32(dest,data1,data2)` | Vector multiply subtract, `flo32x4_t` dest. vector registers |
| `vmlal_s16(dest,data1,data2)` | Vector multiply accumulate long, `int32x4_t` dest. vector registers |
| `vmlal_s32(dest,data1,data2)` | Vector multiply accumulate long, `int64x2_t` dest. vector registers |
| `vmlaq_f32(dest,data1,data2)` | Vector multiply accumulate, `flo32x4_t` dest. vector registers |
| `vst1_s16(dest,source)` | Store a single vector into memory, `*int16_t[4]` dest. registers |
| `vst1q_s32(dest,source)` | Store a single vector Store a single, `*int32_t[4]` dest. registers |
| `vst1q_f32(dest,source)` | Store a single vector into memory, `*flo32_t[4]` dest. registers |
| `vget_high_s32(reg)` | Splitting vector registers, `int32x2_t` return vector registers |
| `vget_high_f32(reg)` | Splitting vector registers, `flo32x2_t` return vector registers |
| `vget_low_s32(reg)` | Splitting vector registers, `int32x2_t` return vector registers |
| `vget_low_f32(reg)` | Splitting vector registers, `flo32x2_t` return vector registers |
| `vpadd_s32(data1,data2)` | Pairwise add, `int32x2_t` dest. vector registers |
| `vpadd_f32(data1,data2)` | Pairwise add, `flo32x2_t` dest. vector registers |
| `vget_lane_s32(reg,pos)` | Extract lanes from a vector, `int32_t` dest. registers |
| `vget_lane_s64(reg,pos)` | Extract lanes from a vector, `int64_t` dest. registers |
| `vget_lane_f32(reg,pos)` | Extract lanes from a vector, `flo32_t` dest. registers |

muestra el tiempo de ejecución requerido por las tres implementaciones con diferentes tipos de dato: de tipo entero de 16-bits (`INT16`) y de 32-bits (`INT32`), y de tipo real en coma flotante de 32 bits (`FLO32`).

Dado que las prestaciones mejores se obtienen con `INT16` y `FLO32`, vamos a comparar nuestra implementación basada en funciones intrínsecas **NEON®** para estos dos tipos de datos con una implementación que no las utiliza pero que ha sido compilada, sin embargo, con las opciones de auto-vectorización del compilador. Esta versión se obtiene utilizando la opción `-ftree-vectorize`. El *speed-up* obtenido por la versión basada en **NEON®** se muestra en la Figura 2.8. En el caso particular del tipo de dato `INT16` se obtiene un gran beneficio comparado con la utilización del tipo de dato `FLO32`. También con `INT16` se consigue mucho más beneficio sobre la versión que utiliza la auto-vectorización. Cabe destacar el incremento de velocidad obtenido para la estructura de filtro **FIR**, que está por encima de 5 y 6 veces la velocidad de la versión auto-vectorizada, respectivamente, para filtros de 52 y 256 coeficientes.

Sea $t_{\text{buff}}$ el tiempo transcurrido desde que el buffer de entrada de muestras está disponible hasta que el siguiente buffer de entrada de muestras está disponible. Asumiendo que podemos ejecutar $F$ operaciones de filtrado simultáneamente, el valor $t_{\text{proc}}$ representa el tiempo de procesamiento transcurrido desde que los $F$ datos del buffer de entrada están disponibles hasta que los $F$ datos del buffer de salida

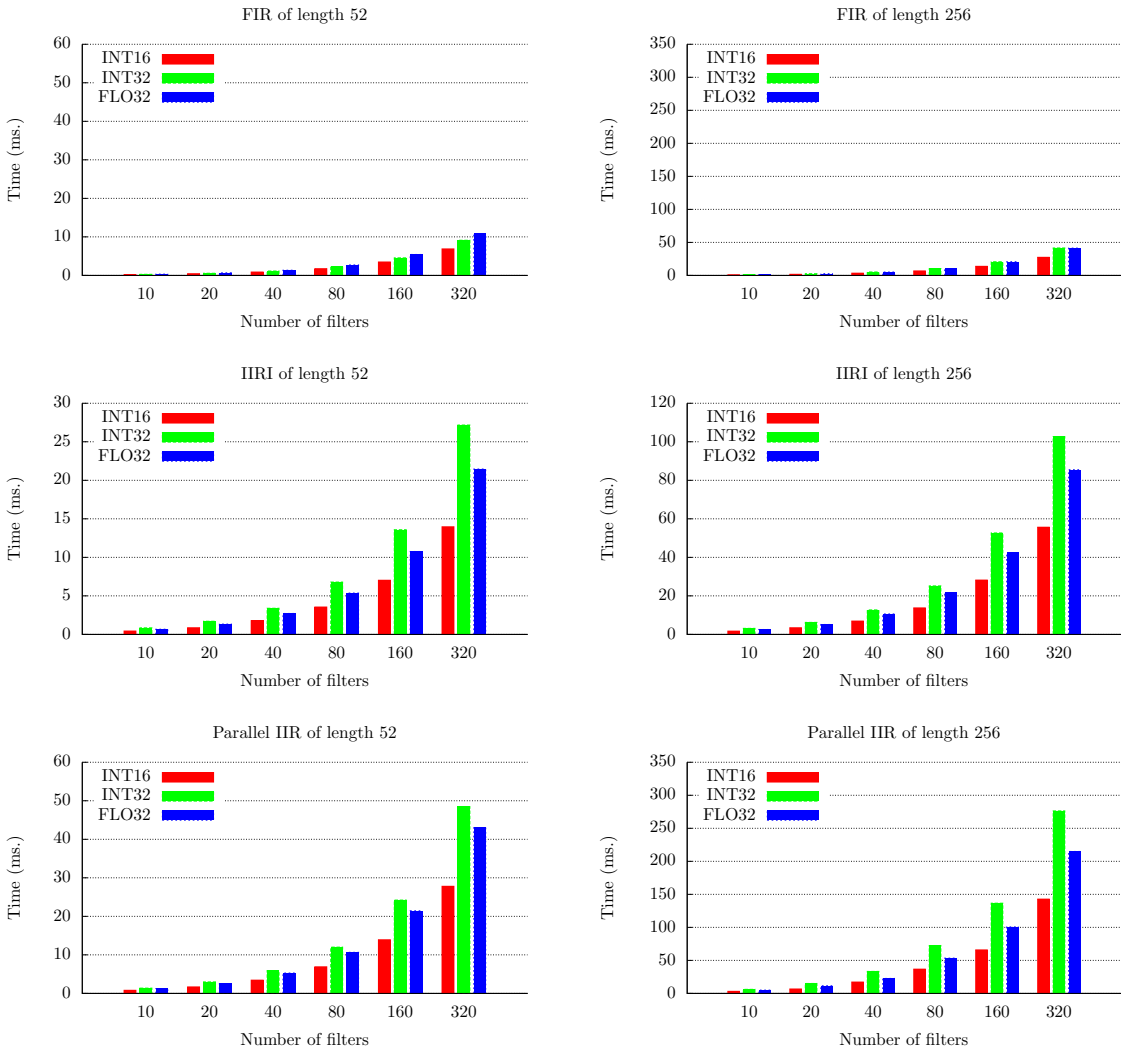



Figura 2.7: Tiempo de ejecución con respecto al tipo de datos utilizado.

han sido totalmente procesados. La operación de filtrado trabaja dentro del umbral de tiempo real definido como $t_{\text{proc}} < t_{\text{buff}}$. El objetivo del siguiente experimento es el de averiguar cuál es el máximo número de operaciones de filtrado, $F_{\text{máx}}$, que pueden ser ejecutadas en tiempo real en el banco de pruebas utilizado para los experimentos. Con objeto de obtener este valor ejecutamos la implementación incrementando el número de filtros calculados comparando el tiempo de ejecución $t_{\text{proc}}$ con el umbral $t_{\text{buff}}$. La Figura 2.9 muestra la evolución del valor $t_{\text{proc}}$ a medida que el número de operaciones de filtrado crece, para tipos de dato `INT16` y `FLO32`. Se puede observar que el máximo número de filtros de tipo **IIR** con 256 coeficientes que pueden ser procesados con restricciones de tiempo real es de 80 para `FLO32` y de 125 para `INT16`, gracias a la utilización de instrucciones **NEON®**. Esto representa

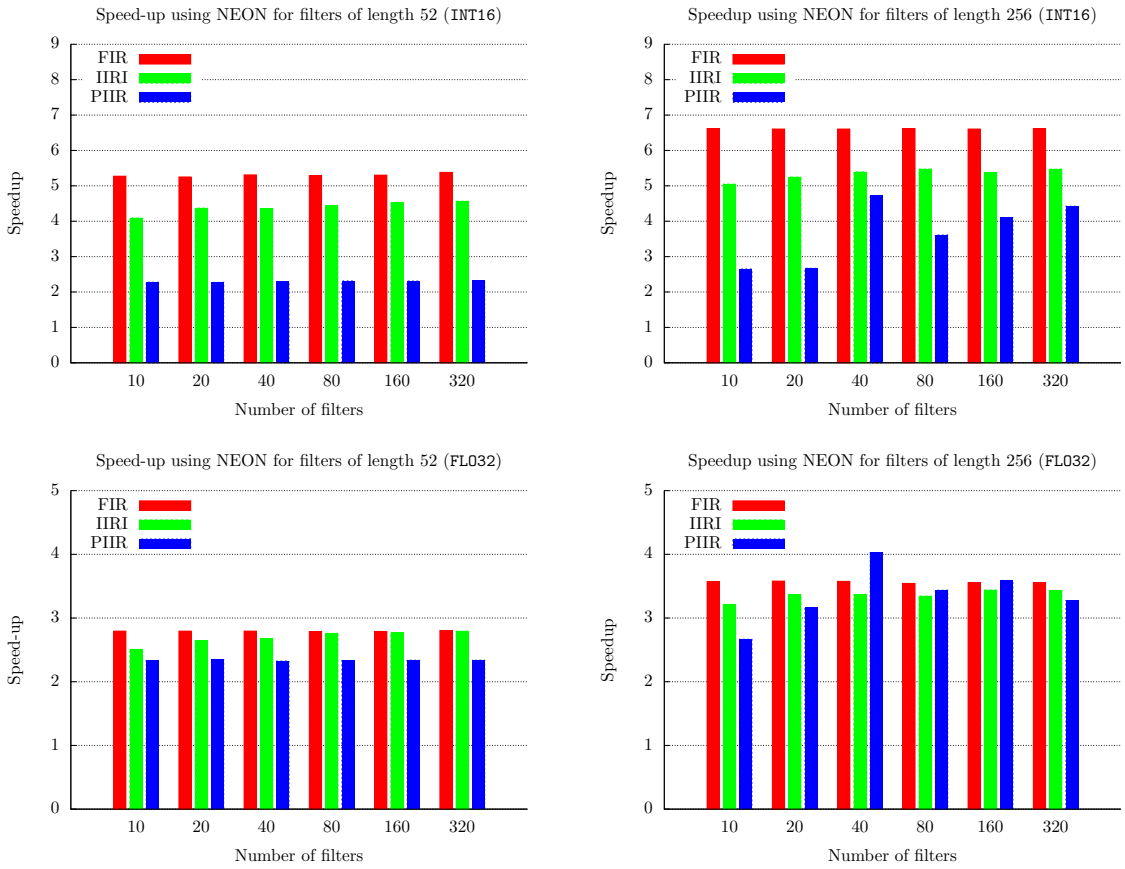



Figura 2.8: *speed-up* de la utilización de funciones **NEON®** con respecto a la versión auto-vectorizada.

un incremento 4× en comparación con la implementación auto-vectorizada. Respecto de los filtros de tipo **FIR**, estos presentan una mejor vectorización dado que no poseen un bucle de retroalimentación. Consecuentemente, la arquitectura ARM® es capaz de manejar el cálculo de más filtros **FIR** que **IIR**. Por ejemplo, para filtros de longitud 256 se pueden calcular 170 operaciones de filtrado en tiempo real de tipo `FLO32` y 260 de tipo `INT16`, es decir, que obtenemos el mismo *speed-up* de cuatro utilizando instrucciones **NEON®** con respecto a la versión auto-vectorizada.

Finalmente, hemos incrementado el aprovechamiento de la arquitectura ARM® subyacente incorporando a todas las implementaciones, es decir para los tres tipos de filtro y tipos de datos utilizados, directivas OpenMP. Según muestra, como ejemplo, la Figura 2.10) para filtros **IIR** con tipo de dato `INT16`, la escalabilidad obtenida es razonable.

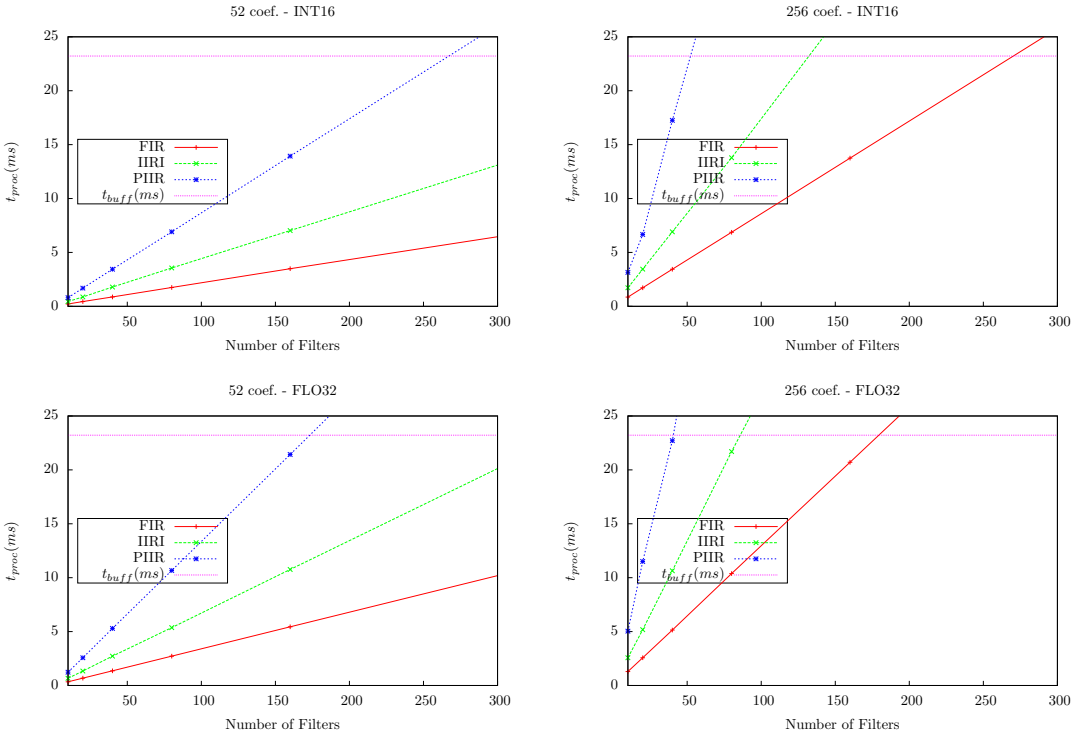



Figura 2.9: Evolución del tiempo de procesamiento necesario para computar en tiempo real.

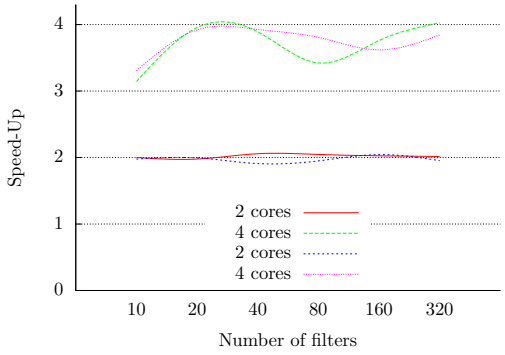



Figura 2.10: *speed-up* con filtros **IIR** sobre tipo `INT16` utilizando dos y cuatro cores del procesador ARM®.

## 2.5 Conclusiones

En este capitulo se ha implementado versiones optimizadas de varios filtros digitales comúnmente utilizados en el tratamiento de señales de audio digital, a saber, filtros de tipo **FIR**, **IIR** y **PIIR**. Para realizar la optimización de los filtros se han empleando instrucciones vectoriales SIMD intrínsecas de la arquitectura ARM®, es decir, instrucciones **NEON®**. Las pruebas experimentales se han realizado en el dispositivo de NVIDIA *Jetson TK1* el cual dispone de una CPU ARM® Cortex-A15 de cuatro cores. El uso de estas instrucciones permite alcanzar más de cuatro veces la velocidad de procesamiento que en el caso de no utilizarlas, incluso utilizando las opciones de compilación adecuadas para forzar al compilador a auto-vectorizar el código.

Se ha trabajado, además, con tres tipos de datos diferentes: `INT16`, `INT32` y `FLO32`. Los filtros utilizan el tipo de datos real, pero en algunos casos particulares, los números reales son convertidos a números enteros con el objetivo de operar más rápidamente con ellos. Aunque existe cierta perdida de precisión al operar con aritmética entera, en muchas aplicaciones de sonido digital esta pérdida es asumible a cambio de una respuesta en tiempo real. De los tres tipos de datos empleados, `INT32` es el más costoso desde un punto de vista computacional. Con el que mejor rendimiento se alcanza es con el tipo de dato `INT16`. Esta conclusión es independientemente del tamaño del filtro.

La evaluación experimental en los procesadores ARM® Cortex-A15 muestra que el código vectorizado puede manejar hasta 80 filtros del tipo **IIR** compuestos por 256 coeficientes cada uno y computarse todos ellos en condiciones de tiempo real. En cuanto a los filtros **FIR**, debemos resaltar que la vectorización funciona mucho mejor que en los filtros **IIR**, produciendo el doble de rendimiento.

Además, se ha realizado una paralelización de los tres tipos de filtro estudiados, ofreciendo un buen *speed-up* gracias a la independencia entre operaciones sobre los datos que integran los filtros. De esta forma hemos podido aprovechar los cuatro cores del procesador ARM® de nuestro dispositivo de pruebas.

# Capítulo 3

# El Algoritmo de *Beamforming*

El término *Beamforming* es empleado en ingeniería de comunicaciones, acústica y procesado de la señal para denotar el filtrado espacial de señales que permite determinar su dirección de origen en presencia de ruido e interferencias. En este capítulo estudiamos una versión del algoritmo de *Beamforming*, conocido como QR-LCMV (Linearly Constrained Minimum Variance), cuya operación principal es la factorización QR de una matriz que relaciona las entradas y las salidas del sistema.

## 3.1 Introducción

El algoritmo de *Beamforming* es un algoritmo usado frecuentemente en el procesamiento de señal digital cuyo propósito es la separación de diferentes fuentes de sonido que se han emitido a la vez y que, por tanto, se han mezclado las unas con las otras en el canal de comunicación. Un ejemplo típico se da cuando varias personas hablan a la vez dentro de una habitación. El objetivo principal del algoritmo de *Beamforming* consiste en separar estas diferentes fuentes de sonido y tratar de extraer las señales originales de cada fuente sin las interferencias acústicas originadas por el resto de las fuentes.

Por otro lado, en el campo de la computación de altas prestaciones (*High Performance Computing* (HPC)) la meta no siempre consiste en mejorar el rendimiento de aplicaciones con un alto coste computacional, sino que también se persigue mejorar el rendimiento de aplicaciones que, si bien no poseen un coste computacional elevado, sí necesitan ejecutarse bajo restricciones de tiempo. Además, en los últimos años, en el campo del HPC también ha crecido la necesidad de optimizar el consumo energético [6], bien utilizando una implementación consciente del consumo energético o bien desarrollando código optimizando para dispositivos de bajo consumo. El algoritmo de *Beamforming* requiere ser ejecutado en tiempo real pero, además, es útil que sea ejecutado en dispositivos "pequeños", fácilmente movibles y de bajo consumo tales como, por ejemplo, los teléfonos móviles. Con este tipo de dispositivos se consigue una alta movilidad pero se tiene el inconveniente de poseer una capacidad computacional reducida (aunque esta capacidad mejore significativamente con el paso de los años). Además, presentan la importante restricción de funcionar a batería, por lo que la reducción del consumo energético es

vital para alargar las horas de uso del dispositivo. Por tanto, aunque se trata de un algoritmo muy conocido que, además, ha recibido mucha atención en los últimos años, en la actualidad sigue siendo un reto el hecho de poder ejecutarlo bajo estas condiciones: en tiempo real y en dispositivos de bajo consumo.

En la actualidad, una gran parte de los procesadores de bajo consumo para dispositivos móviles están basados en la arquitectura ARM®. Es más, algunos diseños contienen varios núcleos con arquitectura ARM® junto a una GPU en el mismo chip formando un *System on a Chip* (SoC). En base a este tipo de procesadores se diseñan dispositivos dedicados a tareas específicas, lo que se conoce como un *sistema embebido* o *empotrado*. La compañía NVIDIA posee la familia de procesadores SoC *Tegra*, entre los que se encuentran, por ejemplo, los procesadores *Tegra K1*, *Tegra X1* y *Tegra X2* que, a su vez, son utilizados en los sistemas embebidos *Jetson TK1*, *Jetson TX1* y *Jetson TX2*, respectivamente. Estos procesadores se pueden encontrar instalados en *tablets* u otros dispositivos móviles, tanto bajo el sistema operativo Android (NVIDIA *Shield*) como en Linux (NVIDIA *Jetson*). Integrados dentro del mismo chip, tanto la CPU como la GPU comparten la misma memoria física y, por tanto, el mismo espacio de direccionamiento. El acceso compartido a los datos posibilita el intercambio de datos CPU-GPU de manera muy rápida al verse reducida la latencia de acceso a memoria. Esto permite realizar un uso optimizado de la capacidad computacional de los dos dispositivos en conjunto y alcanzar así un mejor rendimiento y un mayor ahorro de energía.

Nuestro objetivo es el de alcanzar el tiempo real para el algoritmo de *Beamforming* en equipos embebidos de bajo consumo con CPU y GPU. En particular se ha implementado una versión del algoritmo de *Beamforming* conocida como QR-LCMV y presentada en [41]. En la implementación se han empleado diferentes librerías optimizadas para obtener un alto rendimiento en el cálculo y resolución de problemas de álgebra lineal (BLAS y LAPACK). Se ha hecho uso, en particular, de las librerías OPENBLAS [73] y PLASMA [21], que constituyen implementaciones optimizadas y paralelizadas de BLAS/LAPACK para extraer el máximo rendimiento posible de las *CPU's multicore* actuales. Para hacer uso de la GPU se ha trabajado con la librería CUBLAS [49], la cual es una implementación eficiente propietaria de NVIDIA para sus GPU's. Finalmente, también se ha experimentado con la librería MAGMA [63], la cual ofrece una implementación de las librerías BLAS y LAPACK mixta, es decir, haciendo uso de CPU y GPU simultáneamente en los algoritmos de las funciones implementadas.

Como se podrá ver en los primeros resultados del capítulo, ninguna de estas librerías aprovecha de forma completamente satisfactoria las capacidades computacionales del *Software Development Kit* (SDK) NVIDIA Jetson. Este es el motivo por el que, después de unas primeras pruebas, hemos tomado la determinación de explorar nuestras propias implementaciones del algoritmo de *Beamforming QR-LCMV* con la finalidad de aprovechar al máximo las capacidades computacionales de este *kit* de desarrollo, teniendo en cuenta sus peculiaridades durante la implementación de los distintos algoritmos.

En las tres secciones siguientes se presentan, respectivamente, el fundamento matemático del algoritmo, su implementación y un análisis de resultados de esta implementación que utiliza librerías estándar. En el Apartado 3.5 presentamos la primera contribución importante basada en una actualización rápida de la factorización QR. La segunda aportación importante de este capítulo es una

implementación basada en una estructura de *pipeline*, y se muestra en el Apartado 3.6. Los resultados más destacados que se han ido desarrollando a los largo del capítulo se presentan en una de las plataformas más actuales y representativas de los SoC, el *Jetson AGX Xavier* (Apartado 3.7). Por último, el capítulo cierra con una serie de conclusiones.

## 3.2 Fundamentos matemáticos del Algoritmo de *Beamforming*

La Figura 3.1 muestra gráficamente el problema que soluciona la aplicación del algoritmo de *Beamforming*. La figura muestra un sistema compuesto por dos altavoces que emiten cada uno una señal diferente al otro y un array de tres micrófonos que capturan las señales de sonido emitidas por las fuentes (altavoces). Tanto los altavoces como los micrófonos están separados espacialmente, situados en localizaciones distintas de una habitación dada. En este sistema se requiere que el número de micrófonos (receptores de señales) sea al menos igual al número de altavoces (fuentes de señales), aunque para una mejor calidad de filtrado, es recomendable que el número de receptores de señal sea mayor al número de fuentes.

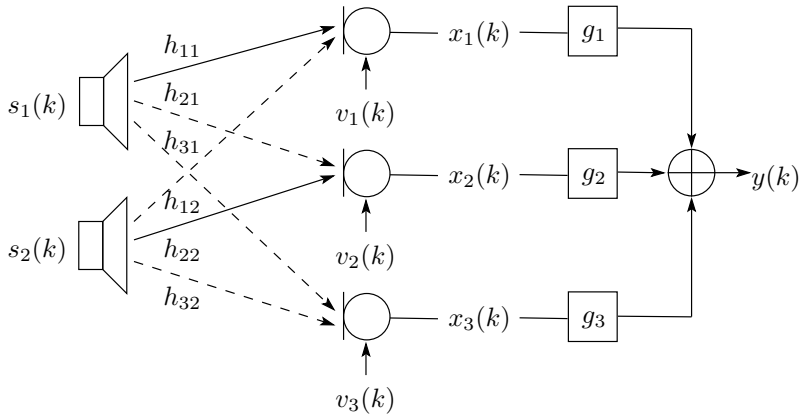



Figura 3.1: Modelo del algoritmo de *Beamforming* para un sistema compuesto por dos altavoces y tres micrófonos.

Formalmente, el modelo de *Beamforming* se puede describir como sigue: Cada $m$, para $m \in \{1, 2\}$, representa a cada uno de los altavoces del modelo. Con $n$, para $n \in \{1, 2, 3\}$, se representan los micrófonos localizados en diferentes puntos de la habitación. La variable $h_{nm}$ representa el canal de comunicación existente entre el altavoz $m$ y el micrófono $n$. El array $g_z$, para $z \in \{1, 2, 3\}$, representa el filtro que se ha de aplicar al micrófono $z$ con la finalidad de separar la señal del resto de señales emitidas y entremezcladas en el canal de comunicación. Finalmente, el array $y(k)$ representa la señal de salida como resultado de aplicar el filtro $g$ a la señal capturada por los micrófonos. Generalizando el sistema podemos afirmar que, siendo $s_m(k)$, $m = 1, \ldots, M$, la señal emitida por $M$ altavoces, el objetivo es

desarrollar $N$ filtros $g_n$, $n = 1, ..., N$, siendo $N$ el número de micrófonos en el sistema, que permiten reconstruir la señal original una vez eliminado el ruido y las reverberaciones de la sala. Para finalizar la descripción, mencionar que se utilizan los canales de respuesta de la habitación, representados como $h_{nm}$, para los valores $n$ y $m$ explicados anteriormente. Estos canales son modelados previamente midiendo la respuesta impulsional mediante un experimento.

La salida del micrófono $n$ viene dada por:

$$x_n(k) = \sum_{m=1}^{M} \sum_{j=1}^{L_h} h_{nm}(j) s_m(k-j) + v_n(k),$$

donde $L_h$ es la longitud de la respuesta al impulso más larga de todos los canales acústicos $h_{nm}$, y $v_n(k)$ es la señal de ruido. Por simplicidad en el modelo y para más claridad no consideraremos el termino de ruido de aquí en adelante. También por claridad y eficiencia computacional se ha reescrito la forma de la señal de salida de cada micrófono de la siguiente forma

$$x_n(k) = \sum_{m=1}^{M} h_{nm}^T s_m(k),$$

donde $s_m(k)$ es el vector columna definido como

$$s_m(k) = \begin{bmatrix} s_m(k) & s_m(k-1) & \dots & s_m(k-L_h+1) \end{bmatrix}^T,$$

y $h_{nm} \in \mathbb{R}^{L_h \times 1}$ es el vector de canal acústico del altavoz $m$ al micrófono $n$.

Considerando ahora el problema de la reconstrucción de la señal fuente $s_m(k)$ a partir de las observaciones grabadas $x_n(k)$, el filtro de *Beamforming* $g_n$ tiene que ser diseñado para que la señal de salida $y(k)$ sea una buena estimación de $s_m(k)$, esto es, $y(k) = \hat{s}_m(k-\tau)$ con mínimo error. Dada la máxima longitud de $L_g$ pasos para cada uno de los $N$ filtros $g_n$, el ancho de banda de la señal de salida de *Beamforming* se puede expresar como

$$y(k) = \sum_{n=1}^{N} g_n^T x_n(k),$$

donde $g_n \in \mathbb{R}^{L_g \times 1}$ es un vector que contiene los valores ordenados de los filtros de *Beamforming* $g_n$, y $x_n(k) = \begin{bmatrix} x_n(k) & x(k-1) & \dots & x_n(k-L_g+1) \end{bmatrix}^T$.

El algoritmo de *Beamforming QR-LCMV* en particular posee filtros con la siguiente forma:

$$g^{LCMV} = \hat{R}_x^{-1} H_{:m} [H_{:m}^T \hat{R}_x^{-1} H_{:m}]^{-1} u_m, \tag{3.1}$$

donde el vector solución $g^{LCMV}$ está formado por la concatenación de los $N$ filtros $g_n$, esto es, $g^{LCMV} = \begin{bmatrix} g_1^T & \dots & g_N^T \end{bmatrix}^T$ y la matriz $H_{:m} \in \mathbb{R}^{(NL_g) \times (L_g + L_h - 1)}$ es la partición de la matriz de canales de

respuesta al impulso en la forma de *Sylvester*[1] que solo incluye las respuestas al impulso de la fuente $m$ a los $N$ micrófonos usados. La matriz $\hat{R}_x$ es la matriz de correlación de las señales grabadas, y $u_m$ es el vector establecido a ceros excepto para una entrada, la cual es puesta a uno en la posición adecuada con objeto de compensar el retardo de la respuesta al impulso de la habitación.

La implementación del algoritmo *Beamforming QR-LCMV* ha sido propuesta con objeto de obtener eficiencia y precisión, por esta razón está principalmente basada en la factorización QR de la matriz, un método numéricamente estable para resolver problemas de mínimos cuadrados debido a la utilización de transformaciones ortogonales. En primer lugar, formamos la matriz $X \in \mathbb{R}^{NL_g \times K}$,

$$X = \frac{1}{\sqrt{K}} \begin{pmatrix} x_1(k) & x_1(k+1) & \cdots & x_1(k+K-1) \\ x_2(k) & x_2(k+1) & \cdots & x_2(k+K-1) \\ \vdots & \vdots & \ddots & \vdots \\ x_N(k) & x_N(k+1) & \cdots & x_N(k+K-1) \end{pmatrix} \tag{3.2}$$

donde $K(< NL_g)$ es el número de muestras empleadas. El algoritmo calcula la factorización QR de $X^T$, esto es, $X^T = QR$, donde $Q$ es una matriz ortogonal y $R$ es una matriz triangular superior [29]. Para poder emplear directamente las rutinas de la librería de álgebra lineal densa LAPACK hemos construido directamente la matriz $X^T$ con almacenamiento de elementos por columnas (*column-major order*). Una vez obtenida la matriz $X$, la matriz $\hat{R}_x$ (3.1) puede ser calculada de la siguiente forma

$$\hat{R}_x = XX^T = R^T Q^T QR = R^T R.$$

Ahora definimos por conveniencia la matriz $W = \hat{R}_x^{-1} H_{:m}$, de manera que el filtro $g^{LCMV}$ (3.1) del algoritmo de *Beamforming QR-LCMV* puede expresarse como

$$g^{LCMV} = W[H_{:m}^T W]^{-1} u_m.$$

Se define la matriz denominada $Z$, la cual es la solución al sistema lineal

$$R^T Z = H_{:m},$$

donde hemos utilizado la factorización QR de la matriz $X$, y teniendo que

$$W = \hat{R}_x^{-1} H_{:m} = (R^T R)^{-1} H_{:m} = R^{-1} Z,$$

donde claramente la matriz $W$ es la solución del sistema lineal $RW = Z$. El camino para obtener los

[1] La forma de *Sylvester* de una matriz para dos o más series (polinomios) permite representar algebraicamente la convolución de dos señales digitales [23]. Esta forma también puede verse como una matriz de bloques *Toeplitz*, mediante la cual también se puede representar la convolución de dos o más señales [66].

filtros procesados resolviendo el sistema lineal

$$Ab_m = u_m, \tag{3.3}$$

donde $A = H_{:m}^T W = H_{:m}^T R^{-1} Z = Z^T Z$. También aquí la solución del sistema lineal (3.3) se obtiene de la factorización QR, en este caso, de la matriz $Z$. Si $Z = Q'R'$ es la factorización QR de la matriz $Z$, entonces el vector $b_m$ puede ser calculado resolviendo los siguientes sistemas lineales triangulares:

$$\begin{aligned} R'^T y &= u_m, \\ R' b_m &= y. \end{aligned}$$

Finalmente, es fácil ver que el algoritmo de *Beamforming* puede ser calculado a través de estos últimos objetos obtenidos: $R$, $Z$ y $b_m$, de la forma siguiente:

$$g^{LCMV} = R^{-1} Z b_m, \tag{3.4}$$

utilizando un producto matriz-vector y la resolución de un sistema de ecuaciones lineales triangular.

## 3.3 Implementación del algoritmo de *Beamforming QR-LCMV*

El Algoritmo de *Beamforming QR-LCMV* derivado en la sección previa puede ser descrito con facilidad a un nivel alto de abstracción a través de cuatro pasos secuenciales (Algoritmo 4). El Algoritmo 4 tiene un coste computacional dominado prácticamente por el coste de la factorización QR de la matriz $X$ (3.2).

**Algoritmo 4:** Pseudocódigo del algoritmo de *Beamforming QR-LCMV*

**Entrada:** $Xmicro$, $N$, $L_g$, $tam$
**Salida:** $Y$

1: Construcción de la matriz $X$ (Algoritmo 5);
2: Cálculo de la factorización QR de la matriz $X$;
3: Cálculo de los filtros ($G$) (Algoritmo 6);
4: Aplicar el filtro (o *Beamformer*[2]) (vector $G$) para obtener la salida (vector $Y$);

La matriz $X$ (3.2) tiene una estructura rectangular caracterizada por un número de filas mucho mayor que el número de columnas. El número de filas y de columnas es una función de la longitud del filtro $L_g$ y del número de micrófonos $N$. Esta matriz es construida tal como muestra el Algoritmo 5. Para la construcción de $X$ se emplean los datos de entrada almacenados en una matriz llamada $Xmicro$, la cual contiene los valores correspondientes a la respuesta al impulso de la habitación donde se han

[2]El filtro que implementa el algoritmo de *Beamforming* también es llamado *Beamformer*.

capturado los datos, es decir, representan las características físicas de los canales de transmisión en la habitación dada ($H$). El parámetro entero *tam* representa el tamaño de la matriz $Xmicro$.

**Algoritmo 5:** Construcción de la matriz $X$

**Entrada:** $Xmicro$, $N$, $L_g$, $tam$
**Salida:** $X$

```
1: pXmicro → Xmicro + L_g − 1; {El símbolo → indica asignación de puntero}
2: pmatrixX → X;
3: for i = 0 to N − 1 do
4:    pXmicroaux = pXmicro;
5:    for k = 0 to L_g − 1 do
6:       for j = 0 to k − 1 do
7:          *pmatrixX = *(pXmicroaux+j);
8:          *pmatrixX++;
9:          pXmicroaux−−;
10:      end for
11:      pXmicro += tam;
12:   end for
13: end for
```

El segundo paso del Algoritmo 4 se ha resuelto con la llamada a la rutina de la librería de álgebra lineal densa LAPACK. Utilizando el paquete de instalación OpenBLAS [73] para dispositivos ARM®, se consigue una implementación de las librerías BLAS/LAPACK optimizadas para este tipo de arquitecturas, tanto en su modo secuencial (empleando un único núcleo o procesador), como en su versión paralela (donde se emplean más de un núcleo o procesador). Además, existen algunas librerías, como MAGMA, que poseen implementaciones de las rutinas de álgebra lineal densa contenidas en LAPACK, que no solo hacen uso de los núcleos de la CPU sino también de la GPU.

Sin embargo, por el momento no se dispone de una versión optimizada de la librería MAGMA para dispositivos con CPU basada en la arquitectura ARM®. Es por ese motivo que se ha reimplementado la rutina de LAPACK para el cálculo de la factorización QR empleando rutinas de la librería CUBLAS, a la vez que se emplean rutinas de CPU optimizadas. Esta estrategia es muy similar a la empleada por la librería MAGMA para procesadores multinúcleo y GPU.

El tercer paso del Algoritmo 4 consiste en el cálculo del filtro de *Beamforming* $G$ ($g^{LCMV}$ (3.4)), cuya descripción detallada la podemos observar en el Algoritmo 6. El último paso consiste en operaciones necesarias para la obtención del resultado final.

## 3.4 Análisis de resultados previos del algoritmo de *Beamforming*

En este primer trabajo con el algoritmo de *Beamforming* se empleó el kit de desarrollo NVIDIA *Jetson TK1* [47, 48]. Este kit de desarrollo es una plataforma hardware que incorpora un sistema operativo basado en linux Ubuntu y que integra el procesador *Tegra K1*. Este tipo de CPU está diseñado

**Algoritmo 6:** Cálculo del filtro de *Beamforming* $G$

**Entrada:** rows, cols, lda, l, $R$, $H$
**Salida:** $G$ { $Z = R \cdot H$; }

1: `memcpy`(Z,$H$,$n \cdot l$);
2: `strsm`(L,U,T,$N$,cols,l,alpha,$R$,lda,Z,cols); { $A = Z \cdot Z'$;}
3: `memcpy`(A,Z,cols· l);
4: `calculateQR`(cols,l,cols,A); { B[] = zeros; B[lg] = 1; }
5: `spotrs`(U,l,right,A,cols,B,l,info);
6: `sgemv`($N$,cols,l,alpha,Z,cols,B,right,zero,$G$,right);
7: `strsv`(U,$N$,$N$,cols,$R$,rows,$G$,right);

para dispositivos móviles y de bajo consumo como teléfonos móviles o tablets. El chip *Tegra K1* que integra el NVIDIA *Jetson TK1* posee un procesador de cuatro núcleos ARM® Cortex-A15 [5] y una GPU basada en arquitectura NVIDIA Kepler con 192 cores. El NVIDIA *Jetson TK1* viene provisto de 2 GB de memoria RAM, que es compartida por la CPU y la GPU. Además de esto, dispone de una amplia variedad de interfaces de conexión: USB 3.0, mini-PCIE, GigaEthernet, SATA3, HDMI..., que hacen este dispositivo apto para una gran variedad de aplicaciones.

Para este test experimental se ha optado por el uso de un sistema basado en 3 micrófonos y 2 altavoces, tal como muestra la Figura 3.1. Se ha trabajado con filtros de dos longitudes, 319 y 1499, las cuales representan la longitud inferior y superior, respectivamente, entre las que se encuentran las longitudes más usuales en este tipo de sistemas enfocados al procesamiento de sonido digital para *Beamforming* en una sala.

Los resultados experimentales proporcionados en esta sección utilizan una implementación propia de este algoritmo en la que utilizamos las herramientas (bibliotecas) más o menos estándar existentes que pueden utilizarse como paso previo a soluciones dedicadas que hemos desarrollado en el marco de esta tesis. Los dos primeros tests de rendimiento realizados se han focalizado en comparar las diferentes versiones de las librerías BLAS y LAPACK optimizadas para el NVIDIA *Jetson TK1*. Las versiones optimizadas de estas librerías las podemos encontrar, por ejemplo, en los paquetes ATLAS y OPENBLAS. La Tabla 3.1 muestra el tiempo de ejecución empleado por estas dos librerías, respectivamente, cuando las utilizamos empleando un único núcleo computacional. Como se puede observar, la optimización de las librerías BLAS y LAPACK realizada por el paquete OPENBLAS es ligeramente mejor que la realizada por el paquete ATLAS para el hardware empleado.

La Tabla 3.1 muestra el tiempo de ejecución para tres partes bien diferenciadas del algoritmo de *Beamforming* que se corresponden con las partes mostradas en el Algoritmo 4, teniendo en cuenta que los dos últimos pasos del Algoritmo 4 se presentan como uno solo ("Filtro $G$").

Analizando el coste computacional de cada una de las partes del algoritmo de *Beamforming* se puede observar que el tiempo de ejecución consumido por la operación del cálculo de la factorización QR de la matriz $X$ es la operación más costosa, con diferencia, teniendo un coste superior al 85 % del tiempo total necesario para el cálculo del algoritmo. Esta disparidad en el coste computacional de cada una de las partes del algoritmo nos indica que el esfuerzo para la optimización del cómputo de

Tabla 3.1: Resultados del algoritmo de *Beamforming* empleando las librerías ATLAS y OPENBLAS con 1 único núcleo computacional.

| ATLAS | | | | |
|---|---|---|---|---|
| | $L_g = 319$ | | $L_g = 1499$ | |
| | Tiempo (seg.) | Porcentaje | Tiempo (seg.) | Porcentaje |
| Matriz $X$ | 0.022 | 1 % | 0.44 | 0 % |
| QR de la Matriz $X$ | 2.327 | 85 % | 224.03 | 86 % |
| Filtro $G$ | 0.373 | 14 % | 35.57 | 14 % |
| Tiempo total | 2.722 | 100 % | 260.04 | 100 % |
| OPENBLAS | | | | |
| | $L_g = 319$ | | $L_g = 1499$ | |
| | Tiempo (seg.) | Porcentaje | Tiempo (seg.) | Porcentaje |
| Matriz $X$ | 0.015 | 1 % | 0.28 | 0 % |
| QR de la Matriz $X$ | 2.049 | 85 % | 191.19 | 86 % |
| Filtro $G$ | 0.348 | 14 % | 30.91 | 14 % |
| Tiempo total | 2.413 | 100 % | 222.37 | 100 % |

este algoritmo debe ir dirigido claramente a la optimización de la factorización de la matriz $X$.

El siguiente paso natural que hemos seguido para minimizar el tiempo computacional de este algoritmo de *Beamforming* ha sido el de emplear la capacidad multi-núcleo de los procesadores ARM®. En el caso del NVIDIA *Jetson TK1* se ha realizado esta prueba empleando los 4 cores Cortex-A15 de los que dispone. Las librerías OPENBLAS y PLASMA ofrecen implementaciones multi-núcleo para las rutinas del BLAS y LAPACK que permiten realizar la factorización QR, entre otras operaciones matemáticas, de forma paralela. La Tabla 3.2 muestra el tiempo de ejecución de forma similar a las tablas anteriores para las librerías OPENBLAS y PLASMA empleando todos los núcleos de CPU disponibles. De igual modo que sucede empleando ATLAS y OPENBLAS con un único núcleo computacional empleado, el paquete OPENBLAS obtiene un mejor rendimiento que PLASMA para el cálculo de la factorización QR multi-núcleo, que como hemos visto anteriormente, es la operación más costosa.

Seguidamente, nos propusimos utilizar la GPU embebida dentro del mismo chip *Tegra K1* del *Jetson*. Primeramente, utilizamos una librería muy extendida para computación heterogénea como es MAGMA. La implementación de esta librería incorpora el uso de la GPU en el cálculo de diversas funciones incluidas en BLAS/LAPACK. En unas pruebas preliminares observamos que el paquete MAGMA no ofrece una implementación del todo óptima en varias de las funciones de BLAS/LAPACK, por lo que decidimos llevar a cabo nuestra propia implementación de aquellas funciones utilizadas en el algoritmo con objeto de aprovechar toda la capacidad de cómputo del *Tegra K1*. Siguiendo la idea inspirada por MAGMA, realizamos nuestra propia implementación obteniendo unos algoritmos híbridos que se ejecutaban, parte en CPU y parte en GPU. Para ello, hicimos uso de la librería OPENBLAS paralela para utilizar los 4 cores ARM® y de la librería CUBLAS para utilizar la GPU. La Tabla 3.3 muestra la mejora de prestaciones obtenida por nuestra implementación utilizando CUBLAS en comparación con la obtenida por la librería MAGMA. Comprando estos resultados con los obtenidos

Tabla 3.2: Resultados del algoritmo de *Beamforming* empleando las librerías OPENBLAS y PLASMA con 4 núcleos computacionales.

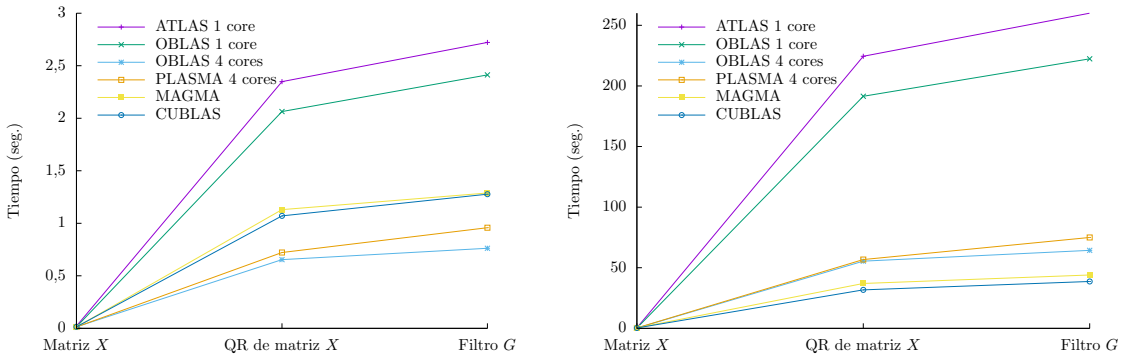
| OPENBLAS | | | | |
|---|---|---|---|---|
| | $L_g = 319$ | | $L_g = 1499$ | |
| | Tiempo (seg.) | Porcentaje | Tiempo (seg.) | Porcentaje |
| Matriz $X$ | 0.015 | 2 % | 0.28 | 0 % |
| QR de la Matriz $X$ | 0.639 | 84 % | 55.19 | 86 % |
| Filtro $G$ | 0.108 | 14 % | 8.91 | 14 % |
| Tiempo total | 0.763 | 100 % | 64.37 | 100 % |
| PLASMA | | | | |
| | $L_g = 319$ | | $L_g = 1499$ | |
| | Tiempo (seg.) | Porcentaje | Tiempo (seg.) | Porcentaje |
| Matriz $X$ | 0.013 | 1 % | 0.32 | 0 % |
| QR de la Matriz $X$ | 0.709 | 74 % | 56.48 | 75 % |
| Filtro $G$ | 0.236 | 25 % | 18.20 | 24 % |
| Tiempo total | 0.958 | 100 % | 75.00 | 100 % |

Tabla 3.3: Resultados del algoritmo de *Beamforming* empleando las librerías OPENBLAS con 4 núcleos computacionales junto con MAGMA o con nuestra implementación basada en CUBLAS.

| OPENBLAS+MAGMA | | | | |
|---|---|---|---|---|
| | $L_g = 319$ | | $L_g = 1499$ | |
| | Tiempo (seg.) | Porcentaje | Tiempo (seg.) | Porcentaje |
| Matriz $X$ | 0.013 | 1 % | 0.40 | 1 % |
| QR de la Matriz $X$ | 1.117 | 87 % | 36.65 | 83 % |
| Filtro $G$ | 0.158 | 12 % | 6.99 | 16 % |
| Total time | 1.289 | 100 % | 44.03 | 100 % |
| OPENBLAS+CUBLAS | | | | |
| | $L_g = 319$ | | $L_g = 1499$ | |
| | Tiempo (seg.) | Porcentaje | Tiempo (seg.) | Porcentaje |
| Matriz $X$ | 0.015 | 1 % | 0.30 | 1 % |
| QR de la Matriz $X$ | 1.056 | 83 % | 31.45 | 81 % |
| Filtro $G$ | 0.206 | 16 % | 6.93 | 18 % |
| Total time | 1.277 | 100 % | 38.68 | 100 % |

haciendo uso únicamente de la capacidad de computo de la CPU (Tabla 3.2) observamos que no necesariamente es mejor utilizar la GPU siempre. Para tamaños de filtro pequeños ($L_g = 319$) es mejor hacer uso de la CPU únicamente, mientras que, para tamaños de filtro grandes ($L_g = 1499$), es mejor utilizar también la GPU.

En la Figura 3.2 se resumen todos los resultados en cuanto a rendimiento computacional del algoritmo de *Beamforming* obtenidos con cada una de las diferentes librerías para los dos tamaños de filtro utilizados a lo largo de este estudio, $L_g = 319$ y $L_g = 1499$.




(a) $L_g = 319$. (b) $L_g = 1499$

Figura 3.2: Tiempo de ejecución de cada parte del algoritmo de *Beamforming* según la librería empleada.

Además, en la Figura 3.3 se representa el porcentaje de mejora por pares de librerías para ambos tamaños de filtro. Por ejemplo, utilizando la librería OPENBLAS con 1 core (OBLAS1) se obtiene un mejor tiempo de ejecución que utilizando la librería ATLAS con 1 core (ATLAS1). Esto indican las dos primeras barras, morada y verde, para filtros de longitud $L_g = 319$ y $L_g = 1499$, respectivamente. En el segundo par de barras de la misma figura podemos ver que el comportamiento en términos de tiempo de ejecución al utilizar los 4 núcleos computacionales de la CPU es estable para ambos tamaños del filtro, pues se obtiene en ambos casos un aprovechamiento relativo de las unidades computacionales en torno al 70 % de eficiencia. La comparación entre las librerías PLASMA y OPENBLAS cuando se utilizan los 4 cores de la CPU se observa en el tercer par de barras, dato que ya habíamos comentado en anteriores resultados. El cuarto par de barras muestra que se puede obtener un mejor comportamiento para aquellos algoritmos basados en la librería CUBLAS frente a las implementaciones basadas en la librería MAGMA, mejora más pronunciada cuanto más grande es el tamaño de filtro. Finalmente, las dos últimas barras denotan lo que ya habíamos observado en la Tabla 3.3. El valor negativo para filtros de longitud pequeña ($L_g = 319$) indica que, en este caso, es mejor no utilizar la GPU. Este dato no deja de ser sorprendente en un sistema SoC, donde la CPU y la GPU comparten el mismo chip y espacio de direccionamiento físico, es decir, la memoria RAM. En entornos de trabajo con una GPU unida al host mediante un PCI la "distancia" CPU-GPU es lógicamente mayor que en dispositivos embebidos y este comportamiento es el que cabe esperar. Sin embargo, los resultados experimentales nos hacen ver que la jerarquía de memoria de este tipo de SoC tiene un impacto significativo que es necesario tener en cuenta en la implementación y que, por tanto, la experimentación es indispensable para encontrar la longitud de filtro para el que sí es rentable utilizar la GPU. Para longitudes de filtro suficientemente grandes como, por ejemplo, $L_g = 1499$, la mejora porcentual llega a ser de alrededor de un 40 % utilizando la GPU en la computación.

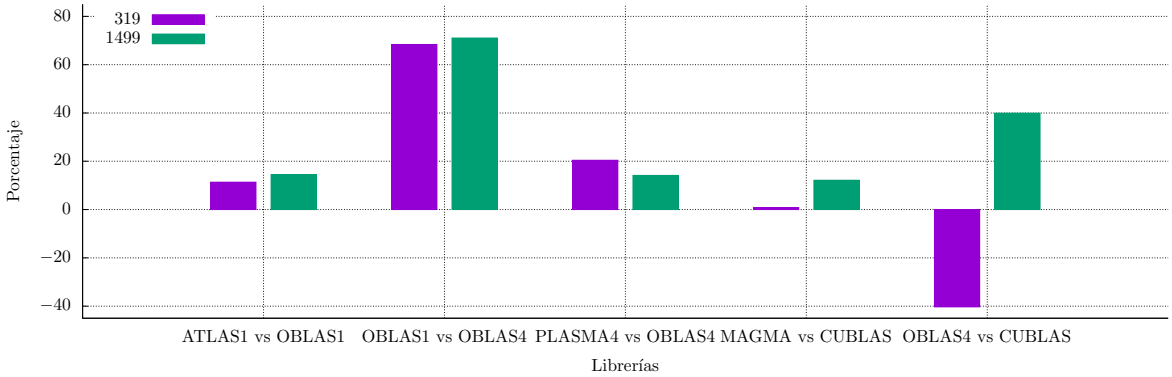



Figura 3.3: Comparación entre pares de librerías para el algoritmo de *Beamforming QR-LCMV*.

El algoritmo de *Beamforming QR-LCMV* ofrece ventajas significativas frente a otros algoritmos en cuanto a precisión debido a la estabilidad numérica de la factorización QR [29]. Sin embargo, el alto coste computacional de esta factorización limita la utilización de este algoritmo bajo restricciones de tiempo real (alrrededor de 30 ms por muestra) teniendo que descartar su uso en aplicaciones típicas de sonido digital a no ser que se utilicen equipos de altas prestaciones o se renuncie a la utilización de longitudes de filtro moderadas, lo que repercute en el objetivo principal de separación de fuentes. Este problema constituye un reto aún más ambicioso cuando queremos llevar este algoritmo a entornos de trabajo de bajas prestaciones computacionales donde prima la movilidad y, por tanto, un bajo consumo energético. La combinación de procesadores SoC que incluyen tanto una CPU multicore basada en arquitectura ARM® como una GPU, como es el caso del NVIDIA *Jetson TK1*, ha abierto la puerta a la posibilidad de alcanzar el objetivo propuesto de llevar la potencia del algoritmo *Beamforming QR-LCMV* a entornos computacionales de bajo consumo, con ejecución en tiempo real y con longitudes de filtro suficientes como para obtener las señales fuente originales limpias.

Sin embargo, lo que hemos observado hasta ahora es que las librerías existentes no están completamente adaptadas a un entorno como este y que, por tanto, no son capaces de extraer toda la capacidad del hardware existente, indispensable para este tipo de aplicaciones. Hemos demostrado, a través de los experimentos realizados y de las implementaciones particulares llevadas a cabo, por un lado, que es posible llegar al objetivo de extraer más rendimiento de este hardware, pero también, por otro lado, que conseguir este objetivo no es trivial. Además de combinar apropiadamente las librerías a nuestro alcance, se necesita profundizar sobre la forma en que los algoritmos existentes realizan, en particular, la factorización QR para adaptarla a este caso particular donde, periódicamente para cada muestra o grupo de muestras, es necesario recalcular la factorización.

## 3.5 Actualización rápida de la factorización QR

En algunas ocasiones, como sucede particularmente en el caso de la aplicación que estamos tratando en este capítulo, donde se lleva a cabo la factorización QR de una matriz rectangular que representa el sistema, sucede que sólo una parte de esta matriz rectangular (a menudo pequeña) varía de una muestra a la siguiente. En esta tesis proponemos aprovechar este hecho con objeto de reutilizar cálculos realizados en el procesamiento de una muestra para ahorrar tiempo de ejecución en el procesamiento de la muestra siguiente. Estos ahorros pueden ser significativos, incluso determinantes, en aplicaciones de tiempo real que se ejecutan a baja potencia computacional en dispositivos de bajo consumo. El algoritmo que proponemos para realizar la factorización QR es paralelo y funciona por tareas utilizando OpenMP. Además de explotar la capacidad multinúcleo de la CPU el algoritmo propuesto está habilitado para, en presencia de una GPU, descargar algunas tareas costosas sobre este dispositivo.

### 3.5.1 La factorización QR

Por simplicidad en la exposición, denotaremos mediante $A \in \mathbb{R}^{m \times n}$ la matriz $X^T$ (3.2) de la que queremos obtener su factorización QR, $A = QR$, donde $Q$ es una matriz ortogonal de tamaño $m \times m$ y $R$ es una matriz triangular superior de tamaño $m \times n$. El coste computacional de esta factorización es de $2n^2(m - n/3)$ flops utilizando reflexiones de Householder [29]. Se han propuesto muchos algoritmos para mejorar el cálculo de la factorización QR pues, cualquier nueva arquitectura o configuración hardware, ha dado lugar a la exploración de nuevos métodos para calcular eficientemente esta factorización, al igual que otras operaciones incluidas en BLAS/LAPACK. Así pues, podemos encontrar un gran número de versiones optimizadas para procesadores secuenciales que mejoran el rendimiento en procesadores con una organización jerárquica del sistema de memoria para procesadores multinúcleo [12, 30], e incluso también aceleradoras como GPU's [4, 37].

En lo que respecta a la optimización de la factorización QR para procesadores multinúcleo, un enfoque que ha tenido bastante éxito en los últimos años se basa en la división del problema computacional en subtareas relacionadas entre sí mediante una jerarquía de dependencias. Estas subtareas son programadas dinámicamente para su ejecución según estas relaciones de dependencia que se han establecido previamente se van cumpliendo. Puede decirse que este enfoque de paralelización, orientado a memoria compartida, ha "tomado prestada" la idea de las técnicas de programación "fuera de orden" (*out-of-order*) implementadas a bajo nivel en arquitecturas super-escalares secuenciales [56]. Un esfuerzo en esta dirección es el proyecto SuperMatrix [13–15], cuyo sistema de ejecución o *runtime* pone en cola las "tareas" que conforman toda la operación de álgebra lineal completa que se desea llevar a cabo construyendo un Grafo Acíclico Dirigido (*Direct Acyclic Graph* (DAG)), codificando sus dependencias y ejecutando las tareas a medida que se satisfacen dichas dependencias. SuperMatrix contiene implementaciones, por ejemplo, de la descomposición LU con *pivotamiento incremental* y un algoritmo por bloques estrechamente relacionado con el anterior para la factorización QR. Ambos algoritmos están basados en un diseño original para la "computación fuera del núcleo" (*out-of-core*) [36]. Esta es

también la idea existente detrás del proyecto PLASMA [11, 12] que proporciona la implementación de muchas rutinas BLAS/LAPACK de manera similar a SuperMatrix. También tenemos el modelo de programación OmpSs [1], que dota al programador de la capacidad de expresar dependencias entre tareas y un *runtime* para la ejecución de un DAG. Estas propuestas tienen en común la misma motivación: la notación algorítmica regular utilizada para escribir algoritmos en los que las tareas y demás operaciones se organizan en bucles limita la capacidad de expresar la naturaleza "desordenada" de la ejecución de muchas tareas. Como resultado de toda esta experiencia previa y de los buenos resultados obtenidos sale a la luz OpenMP 4.0, que incorpora la capacidad de describir dependencias entre suboperaciones o tareas de una forma sencilla.

Con todo lo anterior, hemos propuesto en esta tesis un algoritmo para la factorización QR inspirado en [56] pero, mientras en dicho trabajo se utilizan herramientas más complejas como OmpSs, nosotros lo hemos reescrito utilizando OpenMP 4.0, lo que ha dado lugar a un algoritmo más sencillo. Además, hemos incorporado en el algoritmo el uso de la GPU de manera que, dependiendo del tipo de tarea, esta se ejecutará en un core de la CPU o en la GPU. Esta técnica permite aprovechar las características propias de un SoC como el *Jetson TK1*.

### 3.5.2 El problema de la actualización QR

Denotaremos mediante $k = 0, 1, \ldots$ el espacio de iteración para el problema de muestreo donde el entero $k$ representa el orden de la muestra en el instante $k$ y $A^{(k)}$ la matriz del sistema en dicho instante. El procesamiento de una muestra implica la factorización QR de la matriz del sistema $A^{(k)}$. El algoritmo propuesto se desarrolla por bloques. A estos bloques los denominaremos "*tiles*", dado que así han sido llamados en la literatura en la que nos hemos inspirado para realizar este trabajo. Supongamos, entonces, que la matriz $A^{(k)}$ está dividida en estos bloques cuadrados, denominados *tiles*, de tamaño $t_s \times t_s$ (Figura 3.4). La matriz $A^{(k)}$ se "actualiza" con nuevos datos para formar la matriz del sistema en la siguiente iteración $A^{(k+1)}$ de la siguiente manera:

1. Se eliminan las filas "más antiguas", que son las primeras $t_s$ filas.

2. Un conjunto de $t_s$ filas, que provienen de la señal muestreada, se adjuntan al final (parte de abajo) de la matriz del sistema.

En cada iteración $k$ tenemos que calcular la factorización QR de la matriz del sistema $A^{(k)} = Q_k R_k$, teniendo en cuenta que no es necesario formar explícitamente el factor ortogonal $Q_k$, dado que no es necesario para resolver el sistema. Es necesario tener esto en cuenta pues así se puede ahorrar espacio de almacenamiento y también tiempo de computación.

Con todo lo anterior, hemos propuesto un algoritmo por bloques (o por *tiles*) que permite calcular la factorización QR de la matriz del sistema perteneciente a la iteración $k + 1$ ($A^{(k+1)}$) reutilizando datos calculados para factorizar la matriz del sistema en la iteración anterior ($A^{(k)}$). Para llevar a cabo esto nuestra propuesta consiste en trabajar con un tipo diferente de matriz ($J$) que hemos denominado *dentada* (en inglés *jagged*) y cuya forma puede apreciarse en la Figura 3.5.

$$A^{(k)} = \begin{array}{|c|c|c|}\hline A_{-1,0} & A_{-1,1} & A_{-1,2} \\ \hline A_{0,0} & A_{0,1} & A_{0,2} \\ \hline A_{1,0} & A_{1,1} & A_{1,2} \\ \hline A_{2,0} & A_{2,1} & A_{2,2} \\ \hline \end{array} \quad \Longrightarrow \quad A^{(k+1)} = \begin{array}{|c|c|c|}\hline A_{0,0} & A_{0,1} & A_{0,2} \\ \hline A_{1,0} & A_{1,1} & A_{1,2} \\ \hline A_{2,0} & A_{2,1} & A_{2,2} \\ \hline A_{3,0} & A_{3,1} & A_{3,2} \\ \hline \end{array}$$

Figura 3.4: Actualización de la matriz del sistema $A^{(k)}$ en la iteración $k$ para obtener la matriz $A^{(k+1)}$. Cada uno de los subcuadrados en los que se dividen las matrices mostradas en esta figura corresponden a un "*tile*".

$$J^{(k)} = \begin{array}{|c|c|c|}\hline J_{-1,0} & J_{-1,1} & J_{-1,2} \\ \hline J_{0,0} & J_{0,1} & J_{0,2} \\ \hline J_{1,0} & J_{1,1} & J_{1,2} \\ \hline J_{2,0} & J_{2,1} & J_{2,2} \\ \hline \end{array} \quad \Rightarrow \quad J^{(k+1)} = \begin{array}{|c|c|c|}\hline J_{0,0} & J_{0,1} & J_{0,2} \\ \hline J_{1,0} & J_{1,1} & J_{1,2} \\ \hline J_{2,0} & J_{2,1} & J_{2,2} \\ \hline J_{3,0} & J_{3,1} & J_{3,2} \\ \hline \end{array}$$

Figura 3.5: Actualización de la matriz del sistema $J^{(k)}$ en la iteración $k$ para obtener la matriz $J^{(k+1)}$.

Cada fila de *tiles* de $J^{(k)}$ es el factor triangular superior obtenido de la factorización QR de la fila correspondiente de *tiles* de $A^{(k)}$. Para formar $J^{(k+1)}$ a partir de $J^{(k)}$, existe un paso intermedio que consiste en realizar la factorización QR de las nuevas $t_s$ filas (las últimas filas de $A^{(k+1)}$ representadas por $A_{3,0}, A_{3,1}, A_{3,2}$) para obtener las últimas $t_s$ filas de $J^{(k+1)}$, es decir, $J_{3,0}, J_{3,1}, J_{3,2}$. El objetivo es claro: calcular la factorización QR de la matriz *dentada* $J^{(k+1)}$ tiene menos coste computacional que calcular la factorización QR de la matriz completa $A^{(k+1)}$.

### 3.5.3 El algoritmo de factorización QR por *tiles*

Los elementos de cada *tile* son almacenados en posiciones contiguas en memoria. Este aspecto es clave para que los algoritmos basados en esta idea consigan buenas prestaciones. También implica que, si la matriz a factorizar se almacena, como es costumbre, por columnas, es decir, que los elementos de

cada columna se almacenan en posiciones consecutivas de memoria[3], se necesita una transformación previa para reorganizar la matriz como una colección de *tiles*. Sin embargo, la matriz $A$ se construye progresivamente en base a cada nueva muestra entrante, por lo que no es necesaria una transformación completa de la misma, sino que se pueden ir adaptando los datos a medida que se adquieren.

Tanto en la discusión siguiente como en los resultados experimentales utilizamos una matriz que consta de $4 \times 3$ *tiles*, como la que se muestra en la Figura 3.4 y en la Figura 3.5. Se toma este esquema de mallado como referencia dado que parece razonable que en el algoritmo de *Beamforming* se actualice la matriz del sistema con un 25 % de nuevas filas. Esta simplificación no es óbice para que nuestra propuesta pueda extenderse a cualquier otra disposición de malla de *tiles*. Además, hemos asumido que el tamaño del problema es múltiplo del orden del *tile*. Tampoco esta asunción supone una restricción estricta dado que el problema físico subyacente abordado en nuestra propuesta permite cierto grado de libertad en la elección del número de muestras que conforman el tamaño de *tile*.

El Algoritmo 7 (`QRTiled`) muestra los pasos necesarios para realizar la factorización QR de una matriz estructurada en $M \times N$ *tiles* de la forma que se muestra en la Figura 3.4. Este algoritmo utiliza cuatro tipos diferentes de tareas, las ya identificadas y utilizadas en [56]. Su descripción es la siguiente (por simplicidad, solo especificamos las *tiles* de $A$ involucradas en el cálculo como parámetros de entrada, omitiendo el resto parámetros):

1. `D_QR (A(k,k))`: Calcula la factorización QR de la *tile* $A_{k,k}$. Esta operación devuelve un factor triangular superior sobreescribiendo en esta misma *tile* pivote.

2. `D_QT (A(k,k),A(k,j))`: Aplica el factor $Q^T$ obtenido de la factorización QR del bloque diagonal $A_{k,k}$ a la *tile* $A_{k,j}$ (los reflectores de Householder de $Q^T$ están almacenados en la parte triangular inferior de $A_{k,k}$).

3. `TD_QR (A(k,k),A(i,k))`: Calcula la factorización QR de la matriz $\begin{pmatrix} A_{k,k} \\ A_{i,k} \end{pmatrix}$, donde $A_{k,k}$ es triangular superior, por lo que la *tile* $A_{i,k}$ queda rellenada completamente a ceros.

4. `TD_QT (A(i,k),A(k,j),A(i,j))`: Aplica el factor $Q^T$ obtenido por la operación `TD_QR` dentro de la misma fila $i$ de bloques. Esta operación modifica las *tiles* $A_{k,j}$ y $A_{i,j}$. La operación también lee la *tile* $A_{i,k}$ dado que, aunque implícitamente solo contiene ceros, en realidad contiene los reflectores de Householder que representan implícitamente el factor $Q^T$.

Para factorizar matrices *dentadas* (Figura 3.5), es decir, matrices formadas por la composición de porciones triangularizadas, hemos agregado los siguientes dos tipos de tarea:

1. `TD_QR_T (J(0,0),J(i,0))`: Esta operación es equivalente a `TD_QR` cuando la matriz a reducir es $A_{k,k}$ y $J_{i,k}$, es decir, implicando que el factor inferior es triangular superior.

[3]Este tipo de almacenamiento (*column-wise* o *column major order*) es el requerido por las rutinas de los paquetes BLAS y LAPACK.

**Algoritmo 7:** `QRTiled (A)`: factoriza la matriz rectangular $A$ particionada en *tiles*.

```
1: for ( k=0; k<N; k++ ) do
2:   D_QR ( A(k,k) )
3:   for ( j=k+1; j<N; j++ ) do
4:     D_QT ( A(k,k), A(k,j) )
5:   end for
6:   for ( i=k+1; i<M; i++ ) do
7:     TD_QR ( A(k,k), A(i,k) )
8:     for ( j=k+1; j<N; j++ ) do
9:       TD_QT ( A(i,k), A(k,j), A(i,j) )
10:    end for
11:  end for
12: end for
```

**Algoritmo 8:** `QRTiledJagged (J)`: Factorización QR de la matriz de la forma $J^{(k)}$ (Figura 3.6).

```
1: D_QR ( J(M-1,0) )
2: for ( j=1; j<N; j++ ) do
3:   D_QT ( J(M-1,j), J(M-1) )
4: end for
5: for ( i=1; i<M; i++ ) do
6:   TD_QR_T ( J(0,0), J(i,0) )
7:   for ( j=1; j<N; j++ ) do
8:     TD_QT_T ( J(i,0), J(0,j), J(i,j) )
9:   end for
10: end for
11: QRTiled ( J(1:M,1:N) ); /* Algoritmo 7 */
```

$$J^{(k)} \Rightarrow \hat{J}^{(k)} = \begin{array}{|c|c|c|} \hline J_{0,0} & J_{0,1} & J_{0,2} \\ \hline J_{1,0} & J_{1,1} & J_{1,2} \\ \hline J_{2,0} & J_{2,1} & J_{2,2} \\ \hline A_{3,0} & A_{3,1} & A_{3,2} \\ \hline \end{array} \Rightarrow J^{(k+1)}$$

Figura 3.6: Matriz intermedia (denotada como $\hat{J}^{(k)}$) entre las matrices *dentadas* $J^{(k)}$ y $J^{(k+1)}$.

2. `TD_QT_T (J(i,0),J(0,j),J(i,j))`: Como en el caso anterior, esta operación es equivalente a `TD_QT` cuando el factor $Q^T$ se generó con `TD_QR_T`.

El algoritmo para la factorización QR de una matriz *dentada* tiene la forma que se muestra en el

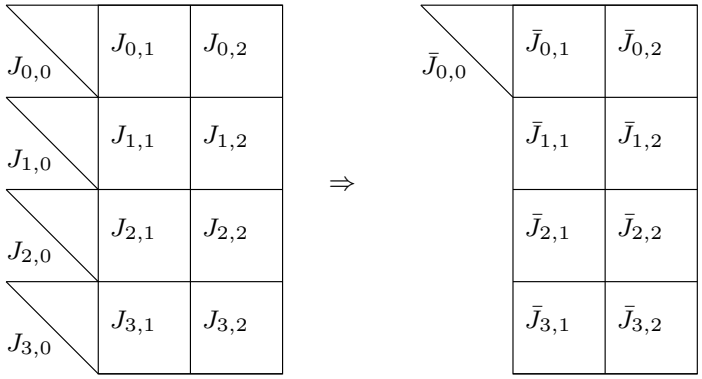



Figura 3.7: Reducción de los bloques triangulares superiores de la primera columna de la matriz $\hat{J}^{(k+1)}$ a la forma triangular superior.

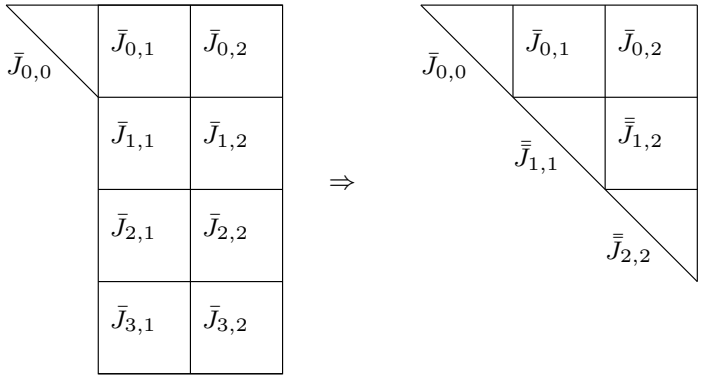



Figura 3.8: Reducción de la matriz $\bar{J}_{1:M,1:N}$ a la forma triangular superior.

Algoritmo 8 (`QRTiledJagged`). Este algoritmo incluye la factorización QR de las $t_s$ filas inferiores de la matriz $\hat{J}^{(k)}$ (Figura 3.6) en las líneas 1-4. Las instrucciones en las líneas 5-10 realizan la factorización QR de la primera columna de bloque de la matriz $J^{(k+1)}$. Esta operación se representa en la Figura 3.7. Finalmente, la submatriz $\bar{J}_{1:M,1:N}$ se reduce a forma triangular superior usando la rutina `QRTiled` (Algoritmo 7), como se ilustra en la Figura 3.8.

### 3.5.4 Modelo de coste para el algoritmo de factorización QR por *tiles*

Esta sección analiza el coste computacional en número de operaciones de coma flotante por segundo (flops) de la factorización QR de una matriz *dentada*, es decir, el coste de la rutina `QRTiledJagged` (Algoritmo 8). Un punto interesante que demuestra el modelo de costes que vamos a ofrecer es que los ahorros al operar con una matriz *dentada* en comparación con el coste de operar con la original son mayores que la proporción de ceros en una matriz *dentada* con respecto al tamaño de la matriz.

Considérense la Figura 3.6 y la Figura 3.7, el coste $c_1$ de obtener las matrices $J_{3,0}$, $J_{3,1}$ y $J_{3,2}$ para

las nuevas filas representadas por los bloques $A_{3,0}$, $A_{3,1}$ y $A_{3,2}$, se puede aproximar como:

$$c_1 = \sum_{i=1}^{t_s-1} (3(t_s - i + 1) + 4(t_s - i + 1)(n - i)) \approx 2nt_s^2 + \frac{3}{2}t_s^2 - \frac{2}{3}t_s^3 \approx 2t_s^2\left(n - \frac{t_s}{3}\right) flops.$$

La reducción de una matriz *dentada* a la forma triangular superior se divide en dos pasos. El coste del primero ($c_2$), que consiste en la reducción de la primera columna de *tiles* a la forma triangular superior (Figura 3.7), se puede aproximar como

$$\begin{aligned} c_2 = \sum_{i=1}^{t_s} (3(i+1) + 4(i+1)(n-i))\left(\frac{m}{t_s} - 1\right) &\approx \left(2nt_s^2 + \frac{3}{2}t_s^2 - \frac{4}{3}t_s^3\right)\left(\frac{m}{t_s} - 1\right) \\ &\approx \left(2t_s^2\left(n - \frac{2}{3}t_s\right)\right)\left(\frac{m}{t_s} - 1\right) flops. \end{aligned}$$

El coste del segundo paso ($c_3$), representado por la Figura 3.8, tiene la siguiente forma

$$c_3 = 2(n - t_s)^2\left((m - t_s) - \frac{n - t_s}{3}\right) = 2n^2\left(m - \frac{n}{3}\right) - \left(4t_s mn + \frac{4}{3}t_s^3 - 2t_s^2 m - 2t_s^2 n\right) flops.$$

Fusionando los tres costes y operando con el resultado obtenemos una aproximación $T_J$ del coste teórico de ejecutar la rutina `QRTiledJagged` (Algoritmo 8) sobre la matriz $\hat{J}^{(k)}$ que permite obtener la factorización QR de la matriz $J^{(k+1)}$,

$$\begin{aligned} T_J &= c_1 + c_2 + c_3 \\ &= \left(2nt_s^2 - \frac{2}{3}t_s^3\right) + \left(2nt_s^2 - \frac{4}{3}t_s^3\right)\left(\frac{m}{t_s} - 1\right) + 2n^2\left(m - \frac{n}{3}\right) - \left(4t_s mn + \frac{4}{3}t_s^3 - 2t_s^2 m - 2t_s^2 n\right) \\ &\leq \left(2nt_s^2 - \frac{2}{3}t_s^3\right)\left(\frac{m}{t_s}\right) + 2n^2\left(m - \frac{n}{3}\right) - \left(4t_s mn + \frac{4}{3}t_s^3 - 2t_s^2 m - 2t_s 2n\right) \\ &= 2n^2\left(m - \frac{n}{3}\right) - \left(\frac{4}{3}t_s^3 - \left(\frac{4}{3}m + 2n\right)t_s^2 + 2mnt_s\right) \\ &= T_A - T_S, \end{aligned}$$

donde $T_A$ denota el coste (en flops) de realizar la factorización QR de una matriz $A$ de tamaño $m \times n$ como la de la Figura 3.4 y $T_S$ denota el ahorro obtenido al trabajar con una matriz *dentada* como la de la Figura 3.6.

La aceleración ($S$) de la factorización QR de la matriz $\hat{J}^{(k)}$ (Figura 3.6) en comparación con la factorización QR de la matriz $A^{(k+1)}$ (Figura 3.4) se puede aproximar como

$$\begin{aligned} S &= \frac{T_A}{T_J} = \frac{T_A}{T_A - T_S} = \frac{2n^2(m - \frac{n}{3})}{2n^2(m - \frac{n}{3}) - (\frac{4}{3}t_s^3 - (\frac{4}{3}m + 2n)t_s^2 + 2mnt_s)} \\ &= \frac{1}{1 - (\frac{2}{3}t_s^3 - (\frac{2}{3}m + n)t_s^2 + mnt_s)/(n^2(m - \frac{n}{3}))}. \end{aligned}$$

Tabla 3.4: Ejemplos numéricos de aceleración $S$ obtenida cuando se trabaja con matrices *dentadas* de la forma mostrada en la Figura 3.6 con respecto a matrices rectangulares de la forma mostrada en la Figura 3.4.

| $m \times n \backslash t_s$ | 1 | 2 | 8 | 32 | 80 | 160 | 320 | 480 | 640 | 960 |
|---|---|---|---|---|---|---|---|---|---|---|
| $1280 \times 960$ | 1.00 | 1.00 | 1.01 | 1.04 | 1.11 | 1.21 | 1.35 | – | – | – |
| $2560 \times 1920$ | 1.00 | 1.00 | 1.01 | 1.02 | 1.06 | 1.11 | 1.21 | – | 1.35 | – |
| $3840 \times 2880$ | 1.00 | 1.00 | 1.00 | 1.01 | 1.04 | 1.07 | 1.14 | 1.21 | – | 1.35 |

Como la aceleración $S$ no varía linealmente con $t_s$, necesitamos usar ejemplos numéricos para ver el impacto de trabajar con matrices *dentadas* en lugar de las originales. La Tabla 3.4 muestra ejemplos numéricos para varios valores del tamaño de bloque $t_s$ y tres tamaños de matriz que pueden servir para hacerse una idea de la reducción en el coste computacional que puede alcanzarse con la modificación introducida. El tamaño de *tile* ($t_s$) elegido es siempre divisor entero de las dimensiones de la matriz, lo que permite trabajar con mallas rectangulares de tamaño $M \times N$, para $M$ y $N$ enteros. Por ejemplo, para un tamaño de $1280 \times 960$ tenemos mallas de $1280 \times 960$ con $t_s = 1$, de $640 \times 480$ con $t_s = 2$, de $160 \times 120$ con $t_s = 8$, de $40 \times 30$ con $t_s = 32$, ... Evidentemente, para $t_s = 1$ la matriz no es *dentada* sino rectangular, por tanto, también tenemos el mismo coste computacional, luego $S = 1$. A medida que $t_s$ es mayor, también lo es el número de 0's, por tanto, también crece $S$, obteniéndose la mayor aceleración ($S = 1{,}35$) con mallas de $4 \times 3$. Vemos aquí un aspecto interesante, y es que los ahorros obtenidos gracias al uso de matrices *dentadas* es mayor que la proporción de ceros introducidos en la matriz del sistema, para obtener esa forma *dentada*, con respecto al tamaño de la matriz. Ese es el dato que ofrecen los ejemplos numéricos, por ejemplo, en el caso de matrices particionadas en mallas de $4 \times 3$ *tiles*, es decir, para el tamaño $1280 \times 960$ con $t_s = 320$, para el tamaño de $2560 \times 1920$ con $t_s = 640$ y para el tamaño de $3840 \times 2880$ con $t_s = 960$.

### 3.5.5 El algoritmo de factorización QR por *tiles* híbrido

Una versión multiproceso del algoritmo de factorización QR por *tiles* basada en OpenMP resulta bastante sencilla paralelizando los dos bucles indexados por `j` en el Algoritmo 7 o en el Algoritmo 8. Sin embargo, existe un enfoque más eficiente basado en ejecutar las operaciones que se pueden ejecutar simultáneamente "fuera de orden", independientemente de si pertenecen a la misma iteración del bucle más externo o no. Con este fin, el algoritmo se puede formular como una colección de operaciones sobre *tiles* que están relacionadas por un conjunto de dependencias. En nuestra formulación, utilizamos OpenMP 4.0. Se puede obtener un diseño orientado a tareas para la factorización QR paralela por *tiles* introduciendo en la misma versión secuencial (Algoritmo 7 para matrices rectangulares) las directivas de tarea (`task`) de OpenMP tal como ya se muestra en el Algoritmo 9. En estas anotaciones de OpenMP, la cláusula `depend` permite especificar las dependencias que existen entre tareas. En tiempo de ejecución, el *runtime* de OpenMP programa las operaciones para ser ejecutadas de acuerdo con un DAG que representa estas relaciones de dependencia. Un ejemplo de este DAG para factorizar una

**Algoritmo 9:** `PQRTiled`: Versión paralela de la rutina `QRTiled (A)` (Algoritmo 7).

```
1: #pragma omp parallel
2: #pragma omp single private(i,j,k)
3: for (k = 0; k <N; k++) do
4:   #pragma omp task depend( inout: A(k,k) )
5:   D_QR ( A(k,k) )
6:   for (j = k+1; j <N; j++) do
7:     #pragma omp task depend( in: A(k,k) ) depend( inout: A(k,j) )
8:     D_QT (A(k,k), A(k,j))
9:   end for
10:  for (i = k+1; i <M; i++) do
11:    #pragma omp task depend( inout: A(k,k), A(i,k) )
12:    TD_QR (A(k,k), A(i,k))
13:    for (j = k+1; j <N; j++) do
14:      #pragma omp task depend( in: A(i,k) ) depend( inout: A(k,j), A(i,j) )
15:      TD_QT (A(i,k), A(k,j), A(i,j))
16:    end for
17:  end for
18: end for
```

matriz de $4 \times 3$ *tiles* se ilustra en la Figura 3.9, donde distinguimos cada tipo de tarea con un color y/o trazo diferente. Además, los números identifican los índices (`i`, `j` o `k`) de la *tile* principal que está siendo modificada por la tarea. Después de los dos puntos, si existe, puede encontrarse un número que denota la iteración en la que se ha modificado la *tile*, en caso de que esa *tile* sea modificada en diferentes iteraciones.

Hemos utilizado la misma estrategia para implementar la versión multiproceso del algoritmo de factorización QR por *tiles* que funciona con matrices de la forma $\hat{J}^{(k)}$ (Figura 3.6) anotando el Algoritmo 8 con directivas OpenMP del tipo `task+depend` para obtener el Algoritmo 10. El DAG correspondiente a la rutina `PQRTiledJagged` es muy similar, tal como puede apreciarse en la Figura 3.10, donde cada color o trazo diferente corresponde a un tipo de tarea.

Estos algoritmos paralelos han sido mejorados añadiendo la capacidad de usar una GPU junto a los cores CPU del sistema. Existen diferentes opciones a la hora de programar que una tarea sea ejecutada por la CPU o por la GPU. En nuestro caso, hemos decidido que las tareas de tipo `D_QR` y `TD_QR` siempre se ejecuten en CPU, mientras que las tareas de tipo `D_QT` y `TD_QT` se ejecuten en la GPU. De manera análoga, en el caso del Algoritmo 10, cualquier tarea de tipo `TD_QR_T` es ejecutada por la CPU mientras que las tareas de tipo `TD_QT_T` se cargan todas en la GPU.

En el lado de la CPU hemos utilizado OpenBLAS para los núcleos computacionales de BLAS/-LAPACK. Las tareas ejecutadas por la GPU se han subdividido en rutinas BLAS más pequeñas para que se puedan realizar fácilmente a través de rutinas de CUBLAS. Cuando el algoritmo encuentra una tarea de tipo `D_QT` o `TD_QT` las *tiles* involucradas en el cálculo se descargan a la GPU y esta realiza el cálculo, devolviendo los resultados a la CPU después. Esto no resulta difícil de implementar utilizando la *memoria unificada* de CUDA. La *memoria unificada* constituye un espacio de direcciones de memoria único al que puede acceder directamente tanto la CPU como la GPU. Este espacio de direc-

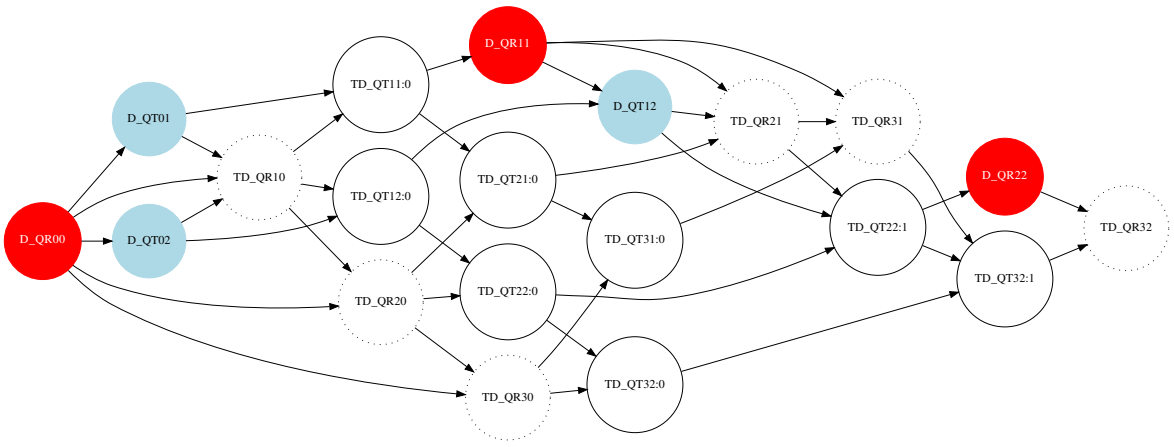



Figura 3.9: Grafo Acíclico Dirigido (DAG) para la factorización QR por *tiles* (rutina `QRTiled`, Algoritmo 9) de una matriz particionada en $4 \times 3$ *tiles* como la de la Figura 3.4.

**Algoritmo 10:** `PQRTiledJagged`: Versión paralela de la rutina `QRTiledJagged (J)` (Algoritmo 8).

```
1: #pragma omp parallel
2: #pragma omp single private(i,j,k)
3: {
4: #pragma omp task depend( inout: J(M-1,0) )
5: D_QR ( J(M-1,0) )
6: for (j = 1; j <N; j++) do
7:   #pragma omp task depend( in: J(M-1,0) ) depend( inout: J(M-1,j) )
8:   D_QT ( J(M-1,j), J(M-1) )
9: end for
10: for (i = 1; i <M; i++) do
11:   #pragma omp task depend( inout: J(0,0), J(i,0) )
12:   TD_QR_T (J(0,0), J(i,0))
13:   for (j = 1; j <N; j++) do
14:     #pragma omp task depend( in: J(i,0) ) depend( inout: J(0,j), J(i,j) )
15:     TD_QT_T (J(i,0), J(0,j), J(i,j))
16:   end for
17: end for
18: PQRTiled (J(1:M,1:N)); /* Algoritmo 9 */
19: }
```

ciones único es reservado mediante la rutina `cudaMallocManaged` y facilita la tarea del programador ya que evita tener que realizar transferencias explícitas de datos entre CPU y GPU. Según nuestra experiencia, la gestión de este tipo de memoria no era del todo eficiente al principio, sin embargo, en las GPU NVIDIA de tipo *Pascal*, la funcionalidad de la memoria unificada se mejoró significativamente. Esta tecnología, además, permite la *sobre-suscripción* de memoria, es decir, poder realizar cálculos fuera de la GPU de forma transparente para el usuario. Los cálculos se ejecutan al mismo tiempo

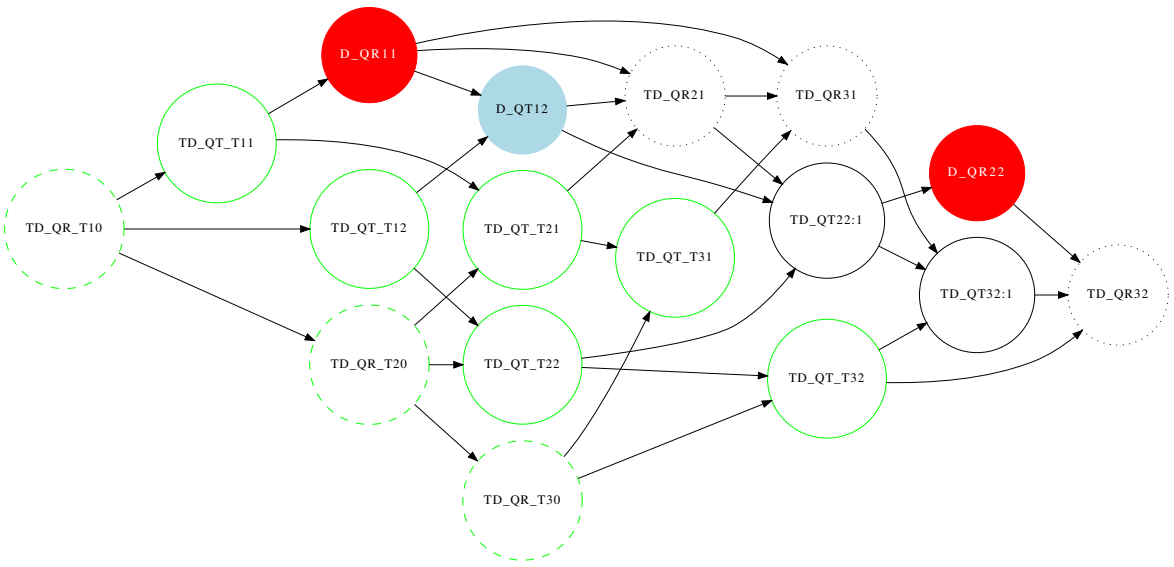



Figura 3.10: Grafo Acíclico Dirigido (DAG) para la factorización QR por *tiles* (rutina `QRTiledJagged`, Algoritmo 10) de una matriz *dentada* particionada en $4 \times 3$ *tiles* como la de la Figura 3.5.

en CPU y GPU. Con la introducción de la tecnología *Hyper-Q* [69] en la arquitectura *Kepler*, una sola GPU puede ejecutar kernels lanzados desde diferentes hilos o subprocesos de CPU, mientras que anteriormente estos kernels eran serializados para poder hacer uso de la GPU. Ahora, varios kernels de GPU que provienen de diferentes subprocesos de CPU se pueden ejecutar simultáneamente dentro de la GPU si hay suficientes recursos. La factorización QR híbrida propuesta aquí hace uso de todas las características mencionadas.

### 3.5.6 Resultados experimentales de la factorización QR

Los experimentos de esta parte del trabajo han sido realizados sobre un kit de desarrollo más avanzado que el NVIDIA *Jetson TK1*: el kit NVIDIA *Jetson TX2*. Esta plataforma cuenta con un Quad ARM® Cortex-A57 y una GPU NVIDIA Pascal con 256 núcleos CUDA. Compilamos el código de la CPU con el compilador GNU gcc 7.5.0, una versión compatible con OpenMP 4.5. Observamos aquí que se necesita una versión $\geq 4.9$ para usar la cláusula `depend` introducida en OpenMP 4.0. Para el código de GPU, usamos CUDA 10.2 y la biblioteca CUBLAS proporcionada para BLAS.

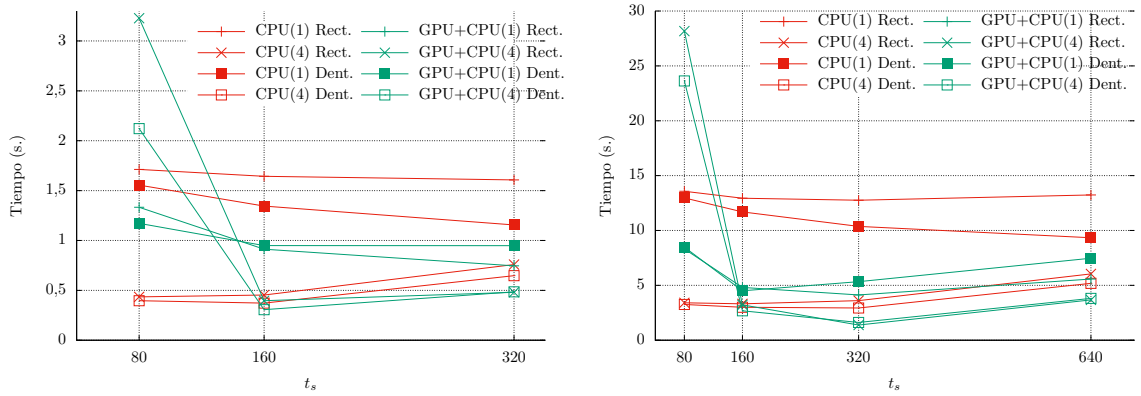
Los resultados de nuestro primer experimento se obtuvieron en un solo núcleo ARM® Cortex-A57. El propósito del experimento es demostrar los beneficios de usar una matriz *dentada* en comparación con la matriz rectangular regular. La Tabla 3.5 muestra el tiempo en segundos para realizar la factorización QR de la matriz $A^{(k)}$ (columna QR($A^{(k)}$)) y la factorización de la matriz *dentada* $J^{(k)}$ (columna QR($J^{(k)}$)). También se compara la aceleración $S$ lograda. Mostramos los resultados en matrices de dos tamaños de problema y dos tamaños de *tile* diferentes ($t_s$) para cada tamaño de problema. La imple-

Tabla 3.5: Tiempo en segundos para la factorización QR de la matriz $A$ (Figura 3.4) comparado con el coste de factorización de la matriz $\hat{J}$ (Figura 3.6) para los tamaños de matrices $1280 \times 960$ y $2560 \times 1920$ variando el tamaño de bloque $b_s$ y el tamaño de *tile* para un *Jetson TX2*.

| $b_s$ | $t_s$ | $m \times n = 1280 \times 960$ | | | $t_s$ | $m \times n = 2560 \times 1920$ | | |
|---|---|---|---|---|---|---|---|---|
| | | $QR(A^{k+1})$ | $QR(\hat{J}^{k+1})$ | S | | $QR(A^{k+1})$ | $QR(\hat{J}^{k+1})$ | S |
| 2 | 160 | 1.660 | 1.373 | 1.21 | 320 | 13.089 | 10.67 | 1.23 |
| 4 | | 1.644 | 1.354 | 1.21 | | 12.943 | 10.52 | 1.23 |
| **5** | | 1.648 | 1.349 | 1.22 | | **12.921** | **10.50** | **1.23** |
| **8** | | **1.637** | **1.342** | **1.22** | | 12.925 | 10.50 | 1.23 |
| 10 | | 1.654 | 1.361 | 1.22 | | 12.988 | 10.56 | 1.23 |
| 16 | | 1.671 | 1.377 | 1.21 | | 13.016 | 10.60 | 1.23 |
| 20 | | 1.676 | 1.385 | 1.21 | | 12.972 | 10.57 | 1.23 |
| 32 | | 1.718 | 1.422 | 1.21 | | 13.126 | 10.72 | 1.22 |
| 40 | | 1.744 | 1.444 | 1.21 | | 13.252 | 10.84 | 1.22 |
| 80 | | 1.889 | 1.581 | 1.19 | | 13.837 | 11.42 | 1.21 |
| 2 | 320 | 1.637 | 1.191 | 1.37 | 640 | 13.542 | 9.796 | 1.38 |
| **4** | | **1.624** | **1.173** | **1.38** | | 12.977 | 9.347 | 1.39 |
| 5 | | 1.631 | 1.178 | 1.38 | | 12.911 | 9.306 | 1.39 |
| **8** | | 1.628 | 1.175 | 1.39 | | **12.840** | **9.223** | **1.39** |
| 10 | | 1.631 | 1.178 | 1.38 | | 12.850 | 9.230 | 1.39 |
| 16 | | 1.636 | 1.185 | 1.38 | | 12.950 | 9.283 | 1.40 |
| 20 | | 1.635 | 1.188 | 1.38 | | 13.165 | 9.412 | 1.40 |
| 32 | | 1.663 | 1.212 | 1.37 | | 13.172 | 9.456 | 1.39 |
| 40 | | 1.682 | 1.227 | 1.37 | | 13.205 | 9.502 | 1.39 |
| 80 | | 1.775 | 1.318 | 1.35 | | 13.455 | 9.785 | 1.38 |

mentación de las tareas está parametrizada por un tamaño de bloque algorítmico ($b_s$), presente en las versiones optimizadas de las rutinas de BLAS, de modo que diferentes tamaños de bloque dan como resultado diferentes tiempos de ejecución en función del hardware subyacente. Tanto el tamaño de bloque $b_s$ como el tamaño de *tile* $t_s$ afectan el rendimiento del algoritmo. Existe cierta libertad para seleccionar el tamaño de bloque $b_s$ siempre que sea un divisor entero de $t_s$. En la exposición realizada hasta ahora, y para mayor claridad, se ha asumido que el tamaño de *tile* es igual al número de filas a actualizar pero, en la implementación real el tamaño de *tile* puede no ser un divisor entero del número de filas a actualizar. La Tabla 3.5 muestra que el mejor tiempo de computación para realizar la factorización QR de las dos matrices, es decir, QR($A^{(k)}$) y QR($J^{(k)}$), se obtiene siempre para la misma combinación de valores ($t_s$,$b_s$). En estos casos, la aceleración de QR($J^{(k)}$) con respecto a QR($A^{(k)}$) está en el rango $\approx [1, 38, 1, 39]$. Para los siguientes experimentos se ha elegido $b_s = 8$ por tratarse de un valor para el que se obtienen tiempos cercanos al óptimo en todos los casos. Estos números son ligeramente superiores a los obtenidos por el modelo de coste mostrado en la Tabla 3.4.

Las gráficas de la Figura 3.11 muestran el tiempo de ejecución para los dos tamaños de problema abordados: $1280 \times 960$ y $2560 \times 1920$. Para cada tamaño de problema se han probado distintos tamaños de *tile*, lo que da lugar a mallas de $4 \times 3$, $8 \times 6$ y $16 \times 12$ (también de $32 \times 24$ para el tamaño $2560 \times 1920$). Las líneas rojas representan tiempo en CPU mientras que las verdes representan la




(a) $1280 \times 960$. (b) $2560 \times 1920$.

Figura 3.11: Tiempo de ejecución del algoritmo paralelo de factorización QR por *tiles* en CPU e híbrido (CPU+GPU) con 1 y 4 cores para matrices rectangulares y *dentadas*.

utilización adicional de la GPU. Se puede ver que, en general, la utilización de la GPU aporta una clara ventaja si solo se utiliza un núcleo de la CPU. Si se utilizan los 4 núcleos, esta ventaja es menor, siendo incluso muy contraproducente si el tamaño de *tile* es pequeño ($t_s = 80$). Otro aspecto a valorar es el mayor paralelismo intrínseco existente en mallas con un número mayor de *tiles*, algo lógico, aunque no siempre se puede elegir el tamaño de *tile* ya que este está condicionado por la velocidad de muestreo y otras variables asociadas al problema de *Beamforming* particular. También observamos la ventaja de utilizar matrices *dentadas* con respecto a las rectangulares, una ventaja que, sin embargo, disminuye con la utilización de más núcleos y de la GPU. Esto último se debe a que el algoritmo de resolución para matrices rectangulares aprovecha mejor este dispositivo. Se puede concluir que existe una gran variabilidad en los parámetros y las alternativas existentes y que, aunque se aprecia una ventaja general en la utilización de más núcleos y de la GPU, es necesario estudiar la combinación adecuada para cada problema con objeto de obtener las mejores prestaciones.

## 3.6 Un *pipeline* para la actualización QR en el procesamiento de señales digitales

Las aplicaciones en tiempo real se caracterizan por realizar el mismo cálculo repetidamente sobre los nuevos datos que llegan. A veces, los datos, que con frecuencia se representan mediante una *matriz de sistema*, cambian de una iteración a la siguiente solo en una pequeña parte. Hasta ahora hemos sido capaces de aprovechar este hecho para acelerar los cálculos, en particular, actualizando la factorización QR de dicha matriz del sistema, un cálculo que es costoso en comparación con otro tipo de factorizaciones pero que tiene a su favor la estabilidad y precisión en los resultados. La idea consistía en trabajar sobre una matriz modificada, denominada *dentada*, en lugar de trabajar en la matriz del

sistema original. Con esta simple idea demostramos que es posible incrementar el rendimiento en un factor cercano a 1,45 para los tamaños de problema que hemos estado manejando en este trabajo, es decir, cuando los datos están representados por una matriz particionada en una malla o *grid* de $4 \times 3$ *tiles* y el 50 % de los datos (25 % filas descartadas y 25 % nuevas filas) cambia de una iteración a la siguiente. El algoritmo paralelo propuesto divide el trabajo en un conjunto de tareas que se ejecutan sobre las *tiles*, tareas que tienen una relación de dependencia entre sí y que cuando se cumplen dichas dependencias se ejecutan, probablemente, de manera concurrente con otras tareas. A todo lo anterior hemos añadido la capacidad de utilizar una GPU, si está presente en el dispositivo, que utilizamos para ejecutar determinado tipo de tareas, caracterizadas por su alto coste computacional y por estar muy optimizadas para GPU.

No obstante, aunque la posibilidad de ejecutar el algoritmo de *Beamforming* en tiempo real está más cerca, aún existe una posible mejora en la actualización de la QR que permite aproximarnos todavía más a este objetivo. En este caso, vamos a aprovechar la posibilidad que ofrece este tipo de aplicación de sonido de admitir un cierto retardo en la respuesta. Este retardo permitirá reorganizar los cálculos sobre un esquema en forma de "tubería" o *pipeline*. La idea que perseguimos con el *pipeline* que proponemos en este apartado es la de avanzar algunos cálculos parciales con el objeto de conseguir que el tiempo final requerido para actualizar la factorización QR sea muy reducido.

### 3.6.1 Descripción del *pipeline* para actualizar la factorización QR de una matriz

Una de las principales limitaciones que podemos encontrar en los algoritmos anteriores es el bajo número de procesadores o núcleos que se pueden utilizar simultáneamente para resolver el problema. La factorización QR en el algoritmo de *Beamforming* es una operación repetida que se lleva a cabo de forma recurrente sobre matrices modificadas con datos de entrada. La estructura de *pipeline* de tareas que proponemos trata de aprovechar mejor la posible existencia de un mayor número de cores. Además, al reutilizar los datos calculados en etapas anteriores de *pipeline*, se reduce el tiempo total necesario para calcular la forma triangular superior ($R$) de la matriz a procesar.

Para entender la idea, supongamos primero que tenemos una matriz *dentada* que representa la matriz del sistema ($J$) y el factor triangular superior ($R_J$) de su factorización QR (Figura 3.12a). Dado un conjunto de filas nuevas representadas por una matriz de $1 \times 3$ *tiles* ($\hat{W}$) (Figura 3.12b), el primer paso del algoritmo es calcular la factorización QR para obtener $W$. Luego, la submatriz de $1 \times 3$ *tiles* en la parte superior ($S$) se descarta y la nueva submatriz *dentada* ($W$) se agrega a la parte inferior, tal como se muestra en la Figura 3.12c. Finalmente, se calcula la factorización QR de la nueva matriz $J'$ para obtener el factor triangular $R_{J'}$. Como se pudo comprobar en la Tabla 3.5, el uso de matrices *dentadas* en lugar de las matrices rectangulares originales, permite acelerar los cálculos hasta $1,39\times$ veces.

Una vez identificadas cada una de las suboperaciones necesarias para actualizar la factorización QR de una matriz dada, podemos ver que existen algunas de estas suboperaciones repetidas en el

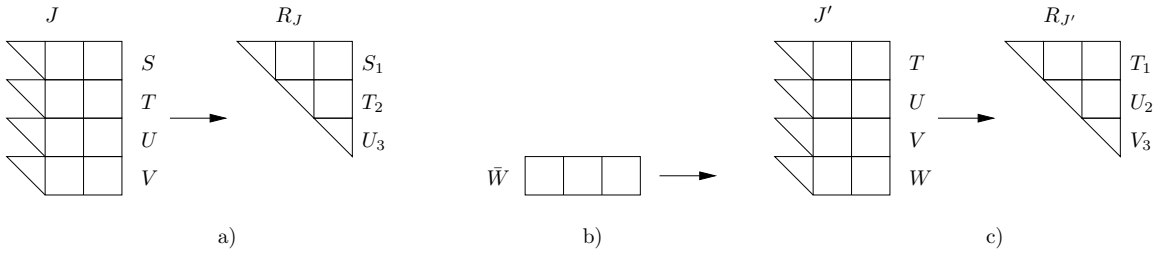



Figura 3.12: Actualización de la factorización QR de una matriz *dentada*.

proceso, por lo que podemos reutilizar y/o ejecutar con antelación estas operaciones con la finalidad de reducir el tiempo total de cómputo. Este hecho, junto con la disponibilidad para utilizar subprocesos concurrentes, motiva la construcción de la estructura de *pipeline* propuesta. La Figura 3.13 muestra el proceso completo cuando la aplicación recibe un nuevo grupo de $t_s$ filas (matrices $\bar{V}$, $\bar{W}$, $\bar{X}$ e $\bar{Y}$). Cada fila de matrices representa las etapas seguidas para reducir una matriz *dentada* a una forma triangular superior utilizando la factorización QR. Por ejemplo, la segunda matriz en la primera fila representa la matriz *dentada* del sistema $J$ de la Figura 3.12a. La Etapa 2 consiste en reducir la submatriz formada por los bloques $S$ y $T$ a la formada por los bloques $\bar{S}_1$ y $\bar{T}_2$. En la siguiente etapa, la submatriz formada por los bloques $\bar{S}_1$, $\bar{T}_2$ y $U$ se reduce a la formada por los bloques $\bar{\bar{S}}_1$, $\bar{\bar{T}}_2$ y $\bar{\bar{U}}_3$. Finalmente, la última etapa produce el factor triangular superior buscado $R_J = \begin{pmatrix} S_1 \\ T_2 \\ U_3 \end{pmatrix}$ de la Figura 3.12a. De la misma manera, la segunda fila de matrices representa la nueva matriz del sistema $J'$ (Figura 3.12c) resultante de descartar el bloque $S$ de la matriz $J$ y añadir la matriz $W$, que es el resultado de reducir el nuevo grupo de filas entrante $\bar{W}$ (Figura 3.12b) a un trapecio rectángulo por su factorización QR. Por tanto, en la última etapa obtendremos $R_{J'}$ (Figura 3.12c). El resto de las filas de la Figura 3.13 representan las siguientes matrices del sistema en forma *dentada*, es decir, matrices formadas cuando se descarta el primer bloque y se agrega uno nuevo.

El objetivo perseguido es el de obtener la factorización QR de la matriz $J'$ (Figura 3.12), es decir, $R_{J'}$, lo más rápido posible. Esto se puede lograr si el factor triangular superior $\begin{pmatrix} \bar{\bar{T}}_1^T \\ \bar{\bar{U}}_2^T \\ \bar{\bar{V}}_3^T \end{pmatrix}^T$ de la factorización QR de la submatriz cuadrada superior de $J'$, es decir, $\begin{pmatrix} T \\ U \\ V \end{pmatrix}$, ya se ha calculado cuando la matriz $W$ está disponible. Un factor clave que se puede observar en la Figura 3.13, y es útil para el diseño del *pipeline*, es que esas etapas enmarcadas operan con la misma submatriz $W$ y las tres pueden ejecutarse simultáneamente, pues son independientes. Hemos utilizado este hecho para diseñar una estructura de *pipeline* como la que se muestra en la Figura 3.14. La figura muestra cuatro pasos de ejecución del *pipeline* en cada línea horizontal. El Paso 1 representa una estado inicial en el cual la salida

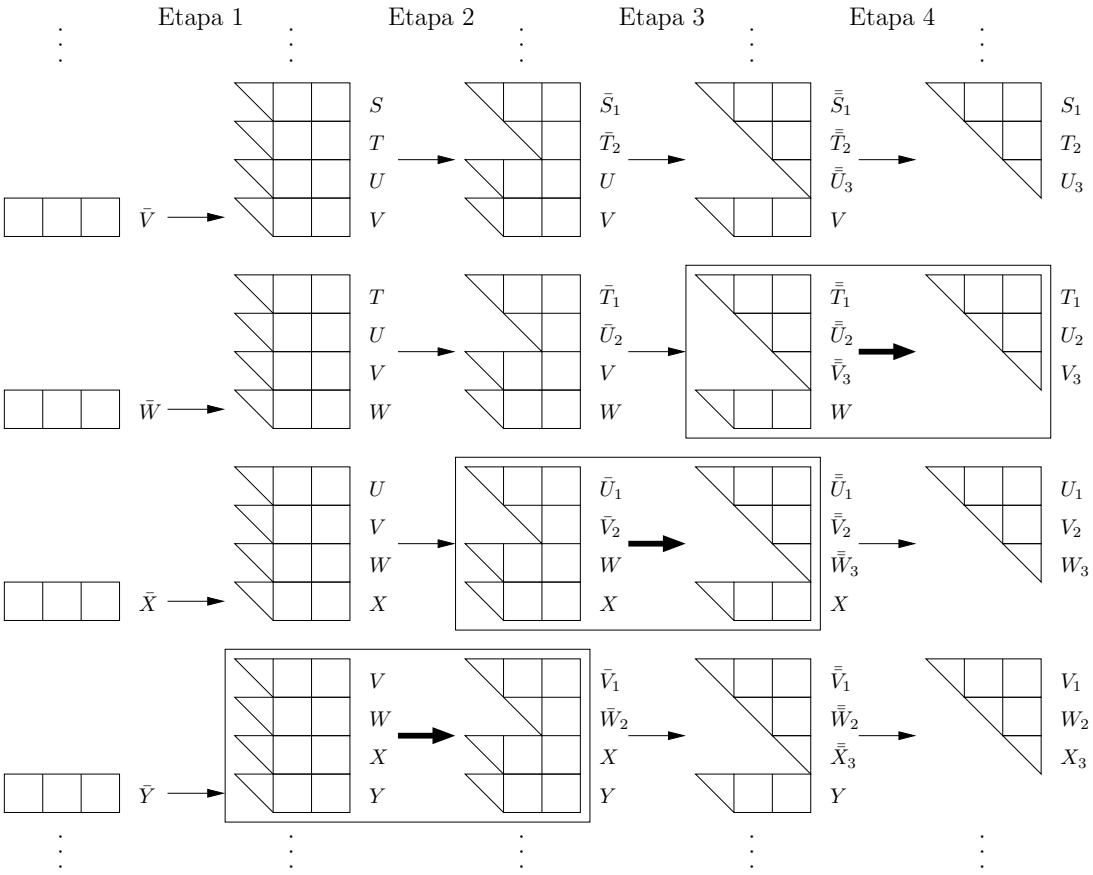



Figura 3.13: Reducción a la forma triangular superior mediante la actualización de la factorización QR de la matriz del sistema $J$ (en forma *dentada*), en diferentes etapas del algoritmo.

del *pipeline* es la matriz $R_J$ (Figura 3.12a). Supongamos ahora que todas las matrices representadas en las etapas del *pipeline* en el Paso 1 han sido calculadas previamente, entonces, el Paso 2 representa la operación en la cual la Etapa 1 del *pipeline* transmite el factor $W$ a las siguientes tres etapas. Los cálculos realmente se llevan a cabo en el tercer paso, donde las matrices construidas en el Paso 2 se reducen mediante una factorización QR a la matriz triangular (o trapezoide) representada dentro de los círculos en negrita. Estas reducciones son precisamente las representadas en las etapas enmarcadas en la Figura 3.13. En el último paso, todas las etapas pasan su factor a la etapa siguiente. En este paso, la última etapa emite su factor triangular a la salida, es decir, $R_{J'}$ (Figura 3.12). Obsérvese que el Paso 1 y el Paso 4 son realmente iguales pero operando con datos diferentes.

Visto el funcionamiento del *pipeline*, mostramos a continuación los costes computacionales. Los costes del *pipeline* se corresponden con los de la etapa más costosa que, en este caso, es la Etapa 4. De acuerdo con Golub y Van Loan [29], aplicar una reflexión de *Householder* para anular las $\mu - 1$ últimas componentes de la primera columna de una matriz rectangular de tamaño $\mu \times \nu$ tiene un

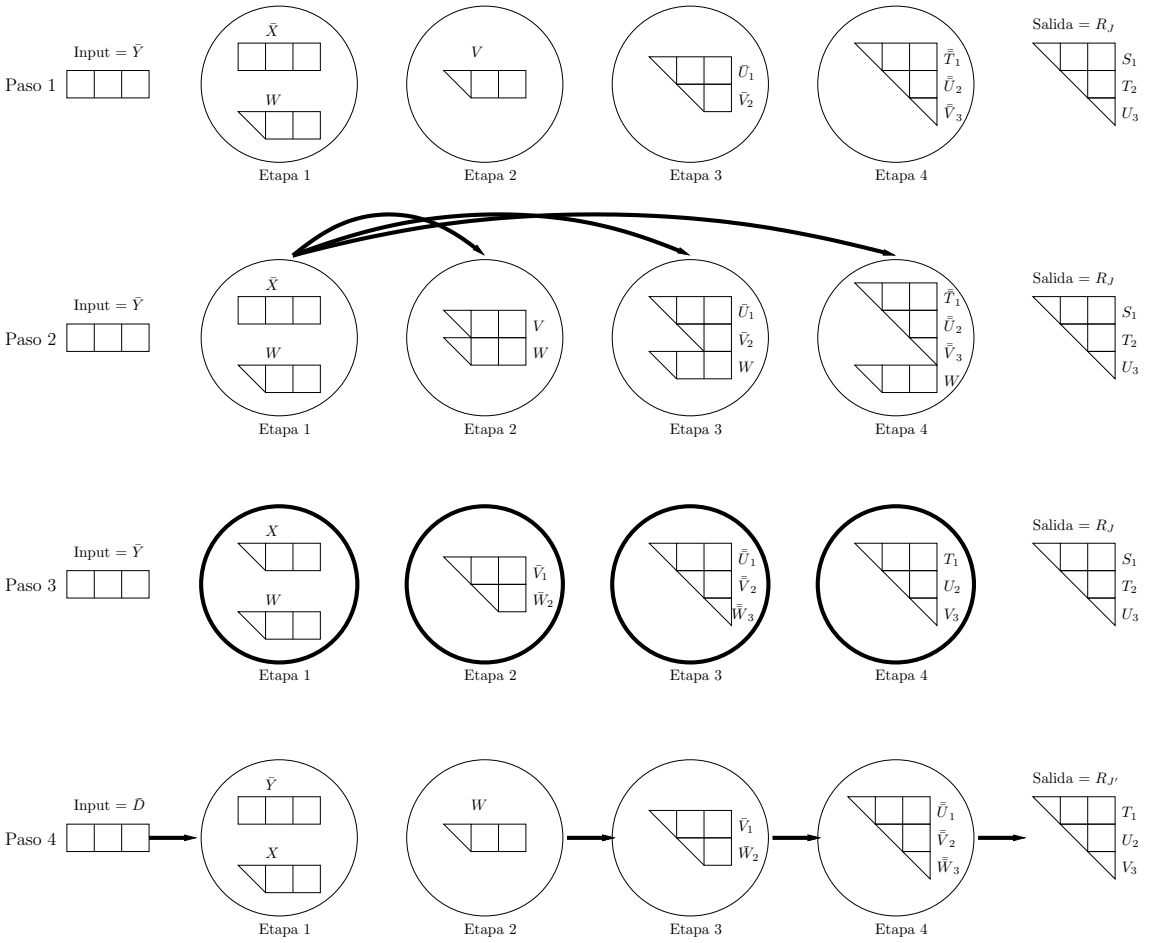



Figura 3.14: *Pipeline* para la actualización QR.

coste computacional de $6\mu + 4\mu\nu$ flops. Entonces, si las triangularizaciones se llevan a cabo mediante reflexiones de *Householder*, el coste computacional de ejecutar la Etapa 4 del *pipeline* (Figura 3.14) se puede aproximar haciendo uso de los siguientes dos términos,

$$\sum_{i=1}^{t_s} 6(i+1) + 4(n-i)(i+1) + \sum_{i=1}^{m} 6(t_s+1) + 4(t_s+1)(m-i). \tag{3.5}$$

El primer sumatorio representa el coste de poner a cero la *tile* triangular inferior de tamaño $t_s \times t_s$, mientras que el segundo sumatorio representa el coste de poner a cero el bloque inferior rectangular restante $t_s \times (n - t_s)$. Después de algunas operaciones aritméticas, el número total de flops se puede aproximar como $2t_s n^2 + 2t_s^3 - 2nt_s^2$ flops.

Para obtener el coste total de la operación, debemos sumar a la ecuación (3.5) el coste de la triangularización del factor $\bar{W}$, que es de $2t_s^2 n$ flops (este coste se puede deducir de la misma manera

de (3.5)). Si tenemos en cuenta la relación entre el número de filas y de columnas de la malla de $4 \times 3$ *tiles*, tenemos los siguientes costes a comparar:

- $54t_s^3$ flops para reducir una matriz rectangular de la forma que se muestra en la Figura 3.4 ($A^{(k)}$) a triangular superior,
- $40t_s^3$ flops para reducir la matriz *dentada* $J^{(k)}$ a triangular superior según la Figura 3.5, y
- $20t_s^3$ flops para reducir una matriz de la forma que se muestra en la Etapa 4 de la Figura 3.14 a la forma triangular superior ($R_{J'}$), incluido el cálculo de $W$ también.

Un algoritmo paralelo para calcular las factorizaciones realizadas en cada una de las etapas del *pipeline* se puede derivar fácilmente del Algoritmo 8 (`QRTiledJagged`). En particular, el algoritmo para realizar la reducción en la Etapa 4, la de mayor coste, da como resultado un DAG como el representado en la Figura 3.15. (Al igual que se ha hecho en el DAG de la Figura 3.9, no se han representado los pasos a realizar para obtener la matriz trapezoide $W$ a partir de $\bar{W}$.) Claramente, el número de tareas es muy bajo en comparación con el DAG anterior, lo que está en relación con la reducción tan importante que se ha obtenido en el coste computacional. También es cierto que el número de tareas que se pueden ejecutar al mismo tiempo se ha reducido a solo dos, sin embargo, esta pérdida de paralelismo se compensa con el aumento en el número de tareas simultáneas que se necesitan para construir el *pipeline*.

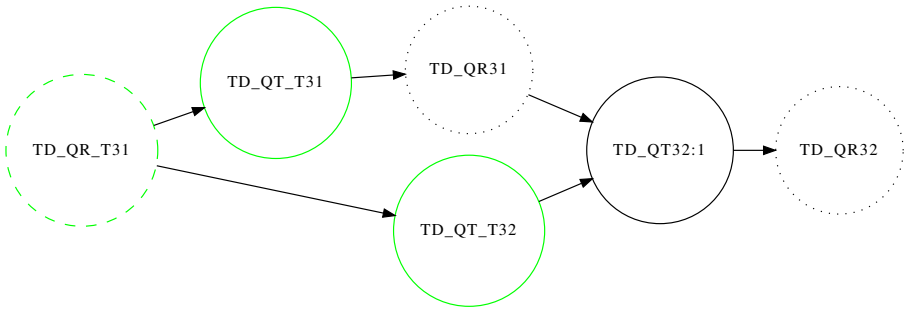



Figura 3.15: DAG de tareas del algoritmo de factorización QR de la matriz representada en la Etapa 4 del *pipeline* de la Figura 3.14.

#### 3.6.1.1 Implementación mediante OpenMP

El *pipeline* se ha implementado de dos maneras distintas. En la primera hemos utilizado C++17 con tareas OpenMP (`task`) y colas de un único productor (*single-producer*)/un único consumidor (*single-consumer*) que conectan las etapas. La implementación que utiliza OpenMP crea tareas para ejecutar cada una de las etapas del *pipeline* y crea una cola *Single-Producer/Single-Consumer* (SPSC) para comunicar matrices entre ellas. La Figura 3.16 muestra el esquema principal del *pipeline* implementado, mientras que el Listado 3.1 muestra una versión simplificada de la implementación en OpenMP. Cabe

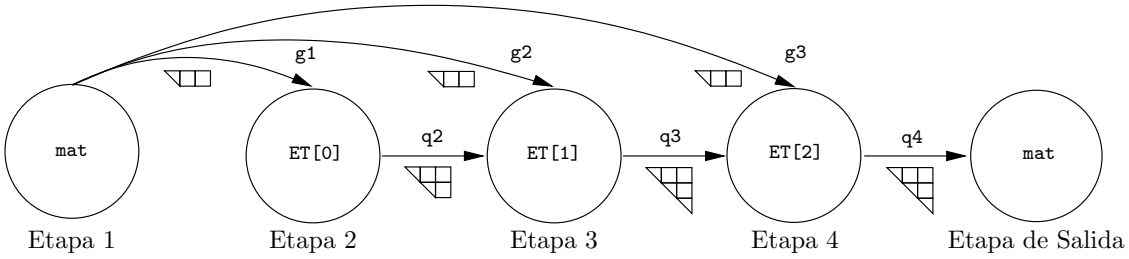



Figura 3.16: Esquema de implementación del *pipeline* de OpenMP.

señalar que las operaciones relacionadas en cada una de las etapas se ejecutan en forma de bucle, por lo que emulan la naturaleza del flujo del patrón del *pipeline*. En particular, una iteración de la Etapa 1 lee el conjunto de entrada de $t_s$ filas, calcula su factorización QR y transmite el resultado a las Etapas 2-4, introduciéndolo en las colas `g1`, `g2` y `g3`. Una iteración de las Etapas 2–4 extrae el bloque de la cola y lo agrega al final de la matriz de estado de etapa (`ET`). Finalmente, las Etapas 2-4 reducen la matriz de estado mediante la factorización QR, introduciendo el factor triangular superior resultante en la cola de salida correspondiente (`q2`, `q3` y `q4` para las Etapas 2, 3 y 4, respectivamente) y extraen/copian el nuevo factor triangular superior (trapezoide) entrante de la cola de entrada `q2`/`q3` para la Etapa 3/Etapa 4 de la matriz de estado de etapa. Este proceso se repite una vez por matriz de entrada y etapa. Denotamos esta versión del *pipeline* como Pipe-OMP-Seq.

Mientras que las etapas del *pipeline* de Pipe-OMP-Seq ejecutan el algoritmo QR por *tiles* en secuencial, hemos desarrollado una versión mejorada que ejecuta el mismo algoritmo pero en paralelo. Específicamente, esta implementación construye un DAG utilizando la directiva `task` de OpenMP y la cláusula `depend`, que encapsula las dependencias de cada factorización QR realizada en las etapas del *pipeline*. Dada la naturaleza paralela anidada de esta implementación, cada tarea OpenMP que ejecuta una etapa del *pipeline* debe utilizar la directiva `taskgroup` de manera que las subtareas que realizan la factorización QR queden "envueltas" y circunscritas a la etapa. Al hacerlo, la tarea de la etapa espera hasta que se completen todas las subtareas relacionadas con la factorización QR antes de pasar a la siguiente iteración con una nueva matriz entrante. Denotamos esta implementación del pipeline como Pipe-OMP-Par.

#### 3.6.1.2 Implementación basada en patrones

Para la segunda manera de implementar el *pipeline* hemos utilizado una herramienta software distinta y más avanzada: la *Interfaz de Patrón Paralelo Reutilizable Genérico* o *Generic Parallel Pattern Interface* (GrPPI) [18], una interfaz de patrones paralelos para aplicaciones C++. Específicamente, GrPPI aprovecha las características modernas de C++, conceptos de metaprogramación y programación genérica para actuar como una interfaz unificada entre OpenMP, *threads* C++ y modelos de programación paralela de Intel *Threading Building Blocks* (TBB), ocultando la complejidad existente detrás del uso de mecanismos de concurrencia nativos. La versión alternativa propuesta para el algoritmo presentado en esta parte hace uso del patrón de tipo *pipeline* GrPPI.

Listing 3.1: Implementación del *pipeline* en OpenMP.

```
#pragma omp parallel num_threads(5)
{
 #pragma omp single nowait
 { // Estado 1
  #pragma omp task shared(gen,g1,g2,g3)
  { for (;;) {
     auto mat = gen();
     g1.push(mat);
     ...
     if (!mat) break;
    }
  }
  // Estado 2
  #pragma omp task shared(ET,qr,g1,q2)
  { for (;;) {
     auto mat = g1.pop();
     if (!mat) break;
     ET[0].copy_rows(*mat, 1, 0, 1);
     q2.push( qr(ET[0], 0) );
     ET[0].copy_rows(*mat, 0, 0, 1);
    }
  }
  // Estado 3
  #pragma omp task shared(ET,qr,g2,q3)
  { ... }
  // Estado 4
  #pragma omp task shared(ET,qr,g3,q4)
  { ... }
  // Output stage
  #pragma omp task shared(q4,cons)
  { ... }
  #pragma omp taskwait
 }
}
```

El Listado 3.2 muestra la interfaz de tipo *pipeline* de GrPPI. Como puede verse, la interfaz recibe la política de ejecución (`exec_policy`), que puede ser cualquiera de las interfaces de programación admitidas (OpenMP, ISO C++ *thread* o Intel TBB), y las funciones (`stages`) relacionadas con las etapas del pipeline por referencia universal (`&&`). La interfaz de C++ utiliza *plantillas* (`templates`), lo que la hace más flexible y reutilizable para cualquier tipo de dato. Obsérvese, por ejemplo, el uso de plantillas variable (*variadic templates*) de C++11 [43], lo que permite que un *pipeline* tenga un número arbitrario de etapas al recibir una colección de funciones pasadas como argumentos. Dependiendo de la política de ejecución elegida, el *pipeline* realiza diferentes acciones. Sin embargo, en aras de la simplicidad, limitamos la ejecución del *pipeline* GrPPI a solo OpenMP, donde se utilizan tareas individuales para articular las etapas del *pipeline*.

Para el desarrollo de esta versión se han adoptado ciertas modificaciones en el *pipeline* original. Dado que el DAG presentado para el *pipeline* Pipe-OMP-Seq en la Figura 3.16 no representa exactamente un patrón de tipo *pipeline*, el DAG original ha sido "aplanado" para poder ser "introducido" en un

Listing 3.2: Interfaz del *pipeline* GrPPI.

```
template <typename E, typename ... S>
void pipeline(E exec_policy, S && ... stages);
```

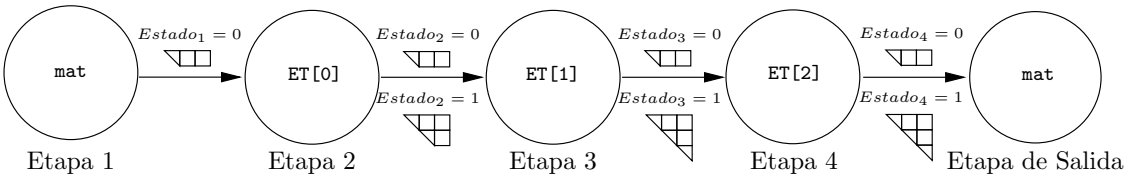



Figura 3.17: Implementación del *pipeline* de GrPPI.

*pipeline* GrPPI. Este proceso de "aplanamiento" ha conducido a la creación de etapas *biestado* para poder imitar el comportamiento del *pipeline* OpenMP. Las etapas biestado pueden representarse como muestra la Figura 3.17 y su implementación puede apreciarse en el Listado 3.3. Durante el Estado 0, el conjunto de $t_s$ filas de $(W)$ es propagado a lo largo de todo el *pipeline*. Esto es, la Etapa 1 lee el conjunto de las $t_s$ filas, calcula su factorización QR y emite el trapecio rectángulo (factor triangular superior $R$) resultante a la Etapa 2. Simultáneamente, la Etapa 2 recibe el factor triangular superior, lo copia localmente en su matriz de estado interna y lo retransmite hacia delante, esto es, a la Etapa 3. Lo mismo sucede en las Etapas 3 y 4. Durante el Estado 1, el resto de operaciones son llevadas a cabo en las Etapas 2-4. Estas etapas realizan la factorización QR de su matriz de estado interna, copian el factor triangular/trapezoide recibido como argumento de su etapa anterior y envían el factor resultante a la siguiente etapa. De cara al análisis posterior llamaremos a esta implementación Pipe-GrPPI-Seq.

Hemos desarrollado la versión GrPPI de menara similar a la implementación OpenMP, que ejecuta el algoritmo de factorización QR por *tiles* a nivel de etapa utilizando una aproximación basada en paralelismo de tareas. Este diseño aprovecha el paralelismo de OpenMP de manera nativa y utilizando el mismo mecanismo presentado en la sección anterior. Nos referiremos a esta versión en lo sucesivo como Pipe-GrPPI-Par.

Con todo, aunque esta versión requirió la transformación de las etapas de uno a dos estados, el alto nivel que proporciona la interfaz basada en patrones de GrPPI simplificó el diseño de la aplicación. Por ejemplo, a diferencia de la implementación de OpenMP, la interfaz GrPPI oculta las colas de comunicación utilizadas para transferir elementos (matrices) entre etapas, así como los bucles principales utilizados en las etapas del *pipeline*. Además, el código final ofrece un diseño compacto, capaz de reducir el número de líneas de código, a la vez que lo hace más robusto gracias a la encapsulación de los mecanismos de concurrencia.

### 3.6.2 Resultados experimentales

En esta sección realizamos una evaluación experimental del algoritmo de factorización QR implementado mediante un diseño en *pipeline*, utilizando los modelos de programación OpenMP y GrPPI.

Listing 3.3: Implementación del *pipeline* en GrPPI.

```
int state[5] = {0,0,0,0,0};
parallel_execution_omp exec_omp{},
grppi::pipeline(
 exec_omp, // Execution mode
 // Estado 1
 [&]() -> optional<Matrix> {
  if (state[0]++ < max_it * 2) {
     return gen(state[0]);
  else
     return {};
 },
 // Estado 2
 [&] (auto mat) {
  switch (state[1]++ % 2) {
   case 0:
    ET[0].copy_rows(mat, 1, 0, 1);
    return mat;
   case 1:
    auto mat_aux= ET[0];
    auto mat_qr= qr(ET[0], 0);
    ET[0].copy_rows(mat_aux, 0, 1, 1);
    return mat_qr;
  }
 },
 // Estado 3
 [&] (auto mat)
 { ... },
 // Estado 4
 [&] (auto mat)
 { ... },
 // Output stage
 [&] (auto mat)
 { ... }
}
```

Esta evaluación la hemos realizado en este contexto hardware/software:

- *Hardware*: Un servidor equipado con 2 procesadores Intel Xeon Ivy Bridge E5-2695 que suman un total de 24 cores corriendo a 2.70GHz, 30MB de caché L3 y 128GB de memoria RAM DDR3. El SO es Linux Ubuntu 16.04.4 LTS con kernel 4.4.0-109.
- *Software*: Para evaluar las prestaciones del *pipeline* OpenMP hemos utilizado el compilador de Intel `icc` v17.0.4 (Intel Parallel Studio XE 2018). Para la implementación basada en patrones hemos utilizado GrPPI v0.4s en combinación con el compilador `gcc` de GNU v6.3.0, que soporta C++17. En ambas versiones utilizamos la opción del compilador `-O3`. Los kernels secuenciales de las librerías BLAS y LAPACK utilizados para ejecutar el algoritmo de descomposición QR por *tiles* fueron las que pueden encontrarse en la *Math Kernel Library* del Intel Parallel Studio XE 2018.

El trabajo de diseño e implementación fue llevado a cabo, en un principio, en este contexto (que llama-

Tabla 3.6: Tiempo en segundos de calcular la factorización QR de una matriz rectangular, una matriz *dentada* y de ejecutar la Etapa 4 del *pipeline* para diferentes tamaños de matriz en el *Xeon*.

| $m \times n$ | Tamaño de *tile* | *Rectangular* | *Dentada* | Pipe-OMP-Seq | Pipe-GrPPI-Seq |
|---|---|---|---|---|---|
| $1280 \times 960$ | 320 | 0.074 s. | 0.064 s. | 0.023 s. | 0.023 s. |
| $2560 \times 1920$ | 640 | 0.379 s. | 0.307 s. | 0.129 s. | 0.117 s. |
| $3840 \times 2880$ | 960 | 1.184 s. | 0.934 s. | 0.363 s. | 0.368 s. |

remos *Xeon*) por comodidad y disponibilidad de herramientas software optimizadas. En el momento de realizar este trabajo no disponíamos de ningún dispositivo de bajo consumo con más de 4 cores iguales y nos interesaba contar con más cantidad para realizar el estudio. Más adelante se ofrecen estos mismos resultados en un dispositivo de bajo consumo como los tratados en el resto de la tesis: el *Jetson AGX Xavier*, que dispone de 8 cores y una GPU.

### 3.6.3 Evaluación del *pipeline* con etapas secuenciales

En un primer paso evaluamos las implementaciones del *pipeline* utilizando el algoritmo secuencial de factorización QR (Pipe-OMP-Seq y Pipe-GrPPI-Seq) con respecto a un algoritmo por *tiles*, y lo mismo con una matriz *dentada*. La Tabla 3.6 muestra una comparación en términos de tiempo de ejecución para tres tamaños de problema diferentes con su respectivo tamaño de *tile*. Los núcleos computacionales básicos que operan sobre cualquier *tile* son rutinas por bloques para las cuales, en todos los casos, el tamaño de bloque elegido $b_s$ se fijó a 40. La columna etiquetada como *Rectangular* muestra el tiempo de ejecución de la factorización QR con un algoritmo por *tiles* (Algoritmo 7) para una matriz cuadrada (Figura 3.4) [56]. La columna etiquetada como *Dentada* muestra el tiempo de ejecución requerido para obtener la factorización QR (Algoritmo 8) de una matriz del tipo que se muestra en la Figura 3.5. La última columna muestra el tiempo obtenido con el *pipeline* propuesto, es decir, este es el tiempo para reducir una matriz de la forma que muestra la Etapa 4 en el Paso 2 según la Figura 3.14 a triangular superior. Tomamos como referencia el tiempo de evaluar la factorización QR realizada en la Etapa 4 ya que es la más costosa y actúa como cuello de botella del *pipeline*. Se puede observar que todos los tiempos obtenidos son coherentes con los cálculos realizados. El uso del *pipeline* permite acelerar el cálculo más de $\times 2$ el tiempo para matrices de la forma *dentada*, y cerca de $\times 3$ el tiempo para la factorización QR de matrices rectangulares.

Para extender este análisis, la Figura 3.18 muestra el tiempo de ejecución de la factorización QR usando el algoritmo por *tiles* sobre matrices de forma *Rectangular* y *Dentada* con diferentes tamaños de problema y número de hilos. Como puede verse, el mejor tiempo se obtiene utilizando 4 hilos. Ir más allá de 4 hilos no aporta ninguna mejora en este caso dado el tamaño de la malla, que está fijado a $4 \times 3$ *tiles*, donde, según el DAG (Figura 3.15), apenas se pueden procesar más de tres *tiles* en paralelo.

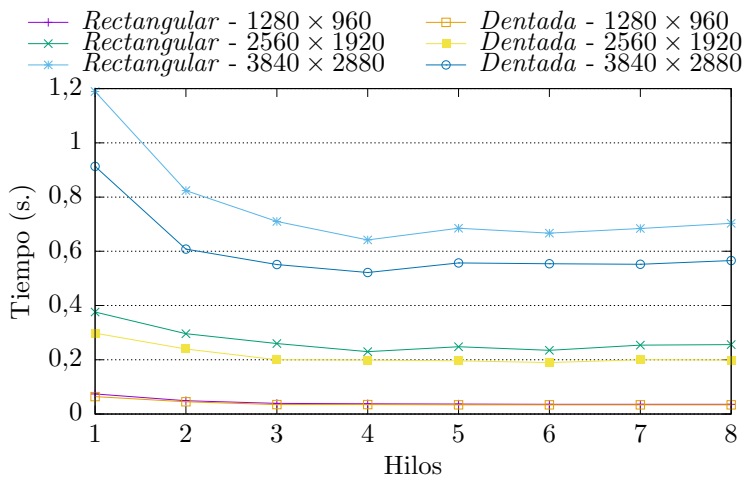


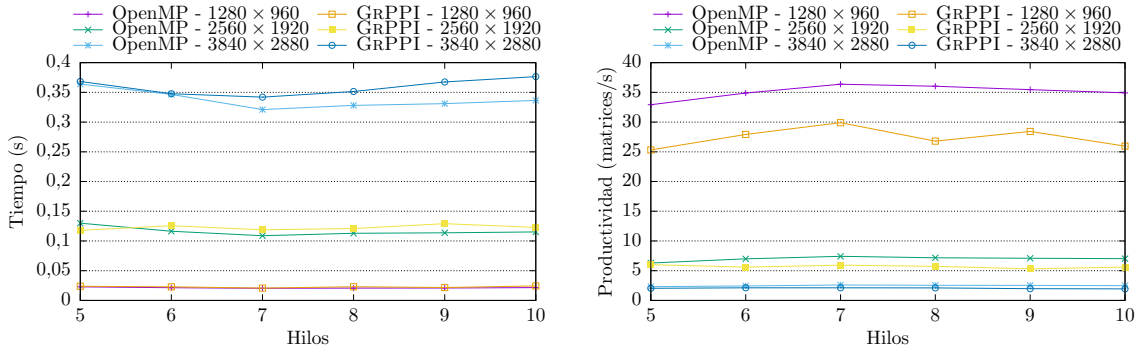

Figura 3.18: Tiempo de ejecución para calcular la factorización QR utilizando el Algoritmo 7 (*Rectangular*) y el Algoritmo 8 (*Dentada*) variando tamaño de matriz y número de hilos.

### 3.6.4 Evaluación del *pipeline* con etapas paralelas

En esta sección evaluamos el tiempo de ejecución de la factorización QR realizada en la Etapa 4 y el rendimiento de las implementaciones Pipe-OMP-Par y Pipe-GrPPI-Par con un número creciente de hilos de 5 a 10. En primer lugar, conviene tener en cuenta que el uso de 5 hilos se corresponde con la versión secuencial ya que el *pipeline* está compuesto por 5 etapas, por lo que no quedan hilos adicionales para la ejecución paralela de la factorización QR. Además, se ha seleccionado como límite superior 10 hilos debido a la malla fija de $4 \times 3$ *tiles* lo que permite que el algoritmo por *tiles* pueda procesar solo hasta tres *tiles* en paralelo. Como se puede ver en la Figura 3.19a, el tiempo de factorización QR en la Etapa 4 permanece casi constante, independientemente del número de hilos adicionales que quedan para procesar subtareas. Es poco notable la disminución del tiempo de ejecución para 7 hilos en todos los tamaños de problema. Esto se debe a que el algoritmo por *tiles* en esta etapa no puede aprovechar más de 2 subprocesos; por lo tanto, si siempre se dedican 5 subprocesos a ejecutar las etapas del *pipeline* y más de 2 subprocesos no aportan beneficios en cuanto a rendimiento, el mejor tiempo de ejecución se obtiene con 7 hilos. En cualquier caso, si comparamos estos resultados con los que se muestran en la Figura 3.18, para el tiempo de ejecución de la factorización QR utilizando el algoritmo por *tiles* sobre matrices rectangulares (*Rectangular*) y (*Dentada*), podemos observar fácilmente que los algoritmos basados en *pipeline* siempre ofrecen mejores aceleraciones. Concretamente, se reduce en al menos 50 % el tiempo de ejecución de ambas factorizaciones QR en matrices rectangulares y *dentadas* para su mejor configuración usando 4 hilos.

Por otro lado, la Figura 3.19b muestra el rendimiento del *pipeline* en la etapa *consumidor* (Etapa 5), es decir, la proporción de factores triangulares superiores (matrices) entregados por segundo. Como se puede ver, las cifras también son casi constantes con respecto al número de hilos. Como ocurre con el tiempo de ejecución de la factorización QR y por las mismas razones mencionadas anteriormente, el

mejor rendimiento se obtiene cuando se utilizan 7 hilos. Además, también notamos una ralentización del rendimiento del *pipeline* GrPPI en todos los tamaños de problema. Relacionamos esta desaceleración con las modificaciones introducidas en la aplicación para encajar el DAG en un *pipeline* con etapas biestado para que coincida con la interfaz de tipo *pipeline* de GrPPI, ofreciendo un menor rendimiento con respecto a la versión OpenMP. Aunque la ralentización es considerable, los beneficios de GrPPI de usar patrones de alto nivel, es decir, menos líneas de código, mejor portabilidad y productividad, pueden compensar el uso de la interfaz en algunos casos.




(a) Tiempo de factorización QR en la Etapa 4.

(b) Productividad del *pipeline*.

Figura 3.19: Tiempo de factorización QR en la Etapa 4 y productividad total de las versiones Pipe-OMP-Par y Pipe-GrPPI-Par con respecto a la variación del número de hilos en el *Xeon*.

## 3.7 El Algoritmo de *Beamforming* en el NVIDIA *Jetson AGX Xavier*

En esta sección se ofrece un análisis experimental conclusivo de los algoritmos desarrollados en este capítulo en una plataforma actual representativa de los dispositivos de bajo consumo que hemos tenido siempre bajo la lupa en todo el desarrollo. Se trata del SoC NVIDIA *Jetson AGX Xavier* (Xavier), el cuál consta de una CPU de 64 bits ARM® de 8 núcleos v8.2, con 8 MB L2 y 4 MB L3 (codename: "Carmel"). Además, cuenta con una GPU Volta de 512 núcleos con núcleos Tensor. En cuanto a memoria, el *Jetson AGX Xavier* cuenta con 32 GB 256-Bit LPDDR4x (137 GB/s) y una memoria eMMC 5.1 de 32 GB para almacenamiento.

Lo primero que hemos hecho ha sido estudiar el tamaño de bloque algorítmico $b_s$ óptimo. La Tabla 3.7 muestra este estudio. Como puede observarse, esta tabla es análoga a la Tabla 3.5 en la que se mostraron estos mismos resultados para el NVIDIA *Jetson TX2*. Al igual que en aquel caso, se muestra el tiempo de factorización para una matriz *rectangular* ($A^{(k)}$, Figura 3.4) y para una matriz *dentada* ($J^{(k)}$, Figura 3.5), ofreciendo así mismo la aceleración $S$ lograda. Respecto de esto último, vemos que el incremento de velocidad es del mismo orden que en aquella ocasión, siendo este resultado bastante lógico. En la Tabla 3.7 solo hemos mostrado estructuras de malla tipo $4 \times 3$, que es el caso

que nos ocupa. Aunque no se muestran, también obtuvimos resultados para tamaños $640 \times 480$ y $1920 \times 1440$ siendo el mejor tamaño de bloque para ambos $b_s = 8$, valor que también fue el utilizado para el *Jetson TX2*.

Tabla 3.7: Tiempo en segundos para la factorización QR de la matriz $A$ (Figura 3.4) comparado con el coste de factorización de la matriz $\hat{J}$ (Figura 3.6) para los tamaños de matrices $1280 \times 960$ y $2560 \times 1920$ variando el tamaño de bloque para un *Jetson AGX Xavier*.

| $b_s$ | $m \times n = 1280 \times 960$ | | $t_s = 320$ | $m \times n = 2560 \times 1920$ | | $t_s = 640$ |
|---|---|---|---|---|---|---|
| | $QR(A^{k+1})$ | $QR(\hat{J}^{k+1})$ | S | $QR(A^{k+1})$ | $QR(\hat{J}^{k+1})$ | S |
| 2 | 0,860 | 0,644 | 1,34 | 6,803 | 4,923 | 1,38 |
| 4 | 0,865 | 0,647 | 1,34 | 6,782 | 4,911 | 1,38 |
| 5 | 0,865 | 0,650 | 1,33 | 6,786 | 4,899 | 1,39 |
| **8** | **0,857** | **0,643** | **1,33** | 6,747 | 4,860 | 1,39 |
| 10 | 0,870 | 0,660 | 1,32 | 6,740 | 4,876 | 1,38 |
| **16** | 0,877 | 0,667 | 1,31 | **6,711** | **4,845** | **1,39** |
| 20 | 0,884 | 0,680 | 1,30 | 6,875 | 4,957 | 1,39 |
| 32 | 0,895 | 0,704 | 1,27 | 8,962 | 5,879 | 1,52 |
| 40 | 0,917 | 0,731 | 1,25 | 8,856 | 6,094 | 1,45 |
| 64 | 1,211 | 0,891 | 1,36 | 7,181 | 5,331 | 1,35 |
| 80 | 1,227 | 0,961 | 1,28 | 7,254 | 5,446 | 1,33 |
| 160 | 1,149 | 1,018 | 1,13 | 7,792 | 6,040 | 1,29 |

Una vez estudiado el tamaño de bloque pasamos a mostrar la reducción en tiempo de ejecución debida a la utilización de varios cores del *Jetson AGX Xavier*. La Figura 3.20 muestra estos resultados. Se hace patente que, al igual que sucedía con el *Xeon*, no es posible escalar más allá de 3 cores por el bajo paralelismo intrínseco que existe en la configuración de $4 \times 3$ *tiles*. Dado que el *Jetson AGX Xavier* posee 8 cores, se hace más necesario aquí, o cambiar a una configuración de malla con más *tiles* o utilizar un *pipeline* con el propósito de aprovechar este recurso.

La introducción de la GPU en el cómputo ofrece unos resultados interesantes aunque muy relacionados con los obtenidos en el *Jetson TX2*. En este caso hemos utilizado únicamente dos tamaños. Para matrices de $1280 \times 960$ (Figura 3.21) encontramos que el uso de la GPU no es indicado para tamaños de *tile* pequeños, tal como se aprecia en la Figura 3.21a con $t_s = 80$ y en la Figura 3.21b con $t_s = 160$. En el segundo caso, solo se obtiene ventaja con uno o dos hilos. Al igual que sucedía con el *Jetson TX2*, la ventaja de utilizar matrices *dentadas* disminuye con la utilización de la GPU debido a que el Algoritmo 7 aprovecha mejor este recurso. Esta ventaja también disminuye con el aumento del número de cores, aunque en menor medida, de manera que la utilización de matrices *dentadas* sigue siendo una idea interesante en el caso de utilizar dispositivos similares que no dispongan de una GPU. Conclusiones parecidas pueden obtenerse en el caso de matrices más grandes como, por ejemplo, de tamaño $2560 \times 1920$ (Figura 3.22). En este caso se ha mostrado una malla con más *tiles* (Figura 3.22a), que posee un grado de paralelismo mayor, y donde se aprecia que la utilización de la GPU resulta claramente desaconsejable. Existe, pues, una relación inversa entre el tamaño de *tile* y el número de hilos, de manera que *tiles* pequeñas con mucho hilos resultan contraproducentes en combinación con

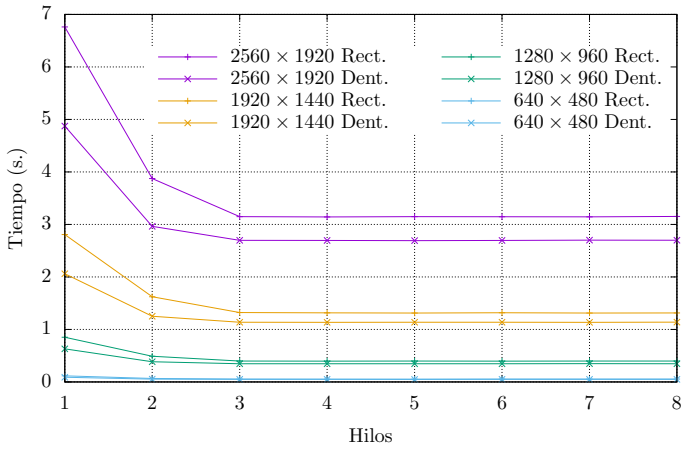



Figura 3.20: Tiempo de ejecución para calcular la factorización QR utilizando el Algoritmo 7 (*Rectangular*) y el Algoritmo 8 (*Dentada*) variando tamaño de matriz y número de hilos en el *Jetson AGX Xavier*.

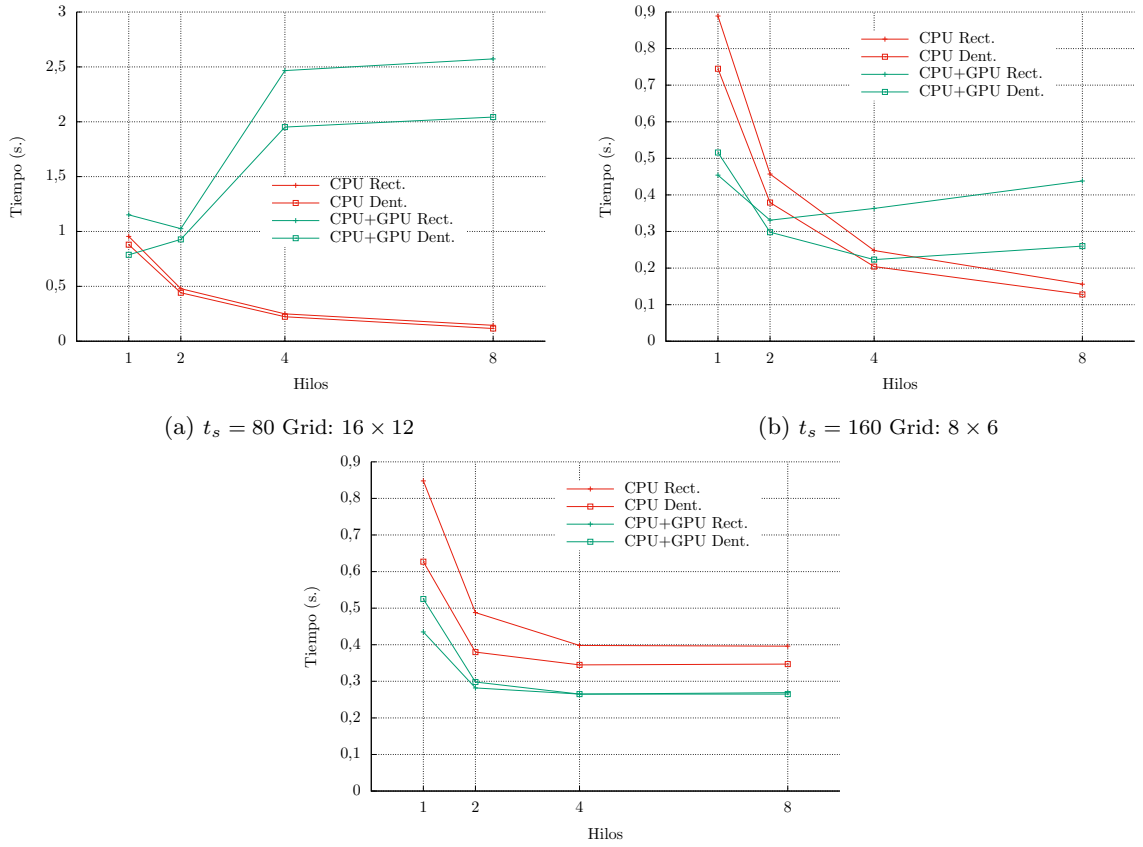
Tabla 3.8: Tiempo en segundos de calcular la factorización QR de una matriz rectangular, una matriz *dentada* y de ejecutar la Etapa 4 del *pipeline* para diferentes tamaños de matrices el *Jetson AGX Xavier*.

| $m \times n$ | Tamaño de *tile* | *Rectangular* | *Dentada* | PIPE-OMP-SEQ | PIPE-GRPPI-SEQ |
|---|---|---|---|---|---|
| $640 \times 480$ | 160 | 0.117 s. | 0.090 s. | 0.032 s. | 0.029 s. |
| $1280 \times 960$ | 320 | 0.852 s. | 0.631 s. | 0.212 s. | 0.207 s. |
| $1920 \times 1440$ | 480 | 2.808 s. | 2.060 s. | 0.681 s. | 0.671 s. |
| $2560 \times 1920$ | 640 | 6.761 s. | 4.875 s. | 1.591 s. | 1.586 s. |

la GPU. Para cada tamaño de problema y dispositivo es necesario realizar un estudio experimental previo que ofrezca la mejor combinación. Los demás aspectos que pueden apreciarse en estas gráficas no hacen más que reafirmar lo ya observado hasta el momento.

El mismo estudio realizado anteriormente para el *pipeline* se ha llevado a cabo ahora con el *Jetson AGX Xavier*. La Tabla 3.8, análoga a la Tabla 3.6, muestra el tiempo de factorización QR con una matriz rectangular, *dentada* y de solo la Etapa 4 del *pipeline*. Se observan reducciones de tiempo significativas también en este caso. La comparación entre la versión OpenMP y la basada en patrones (GRPPI) arroja resultados similares a los obtenidos con otros equipos.

Así como la Tabla 3.8 muestra tiempos con etapas secuenciales, la Figura 3.23a muestra el tiempo utilizado en la Etapa 4 para varios threads, de manera análoga a la Figura 3.19a, es decir, variando el tamaño de la matriz pero siempre sobre una malla de $4 \times 3$ *tiles*. Hay que tener en cuenta que el grafo de tareas que se ejecuta en esta etapa es pequeño y también su grado de concurrencia por lo que el incremento de velocidad no puede ser mejor. Sin embargo, desde el punto de vista de lo



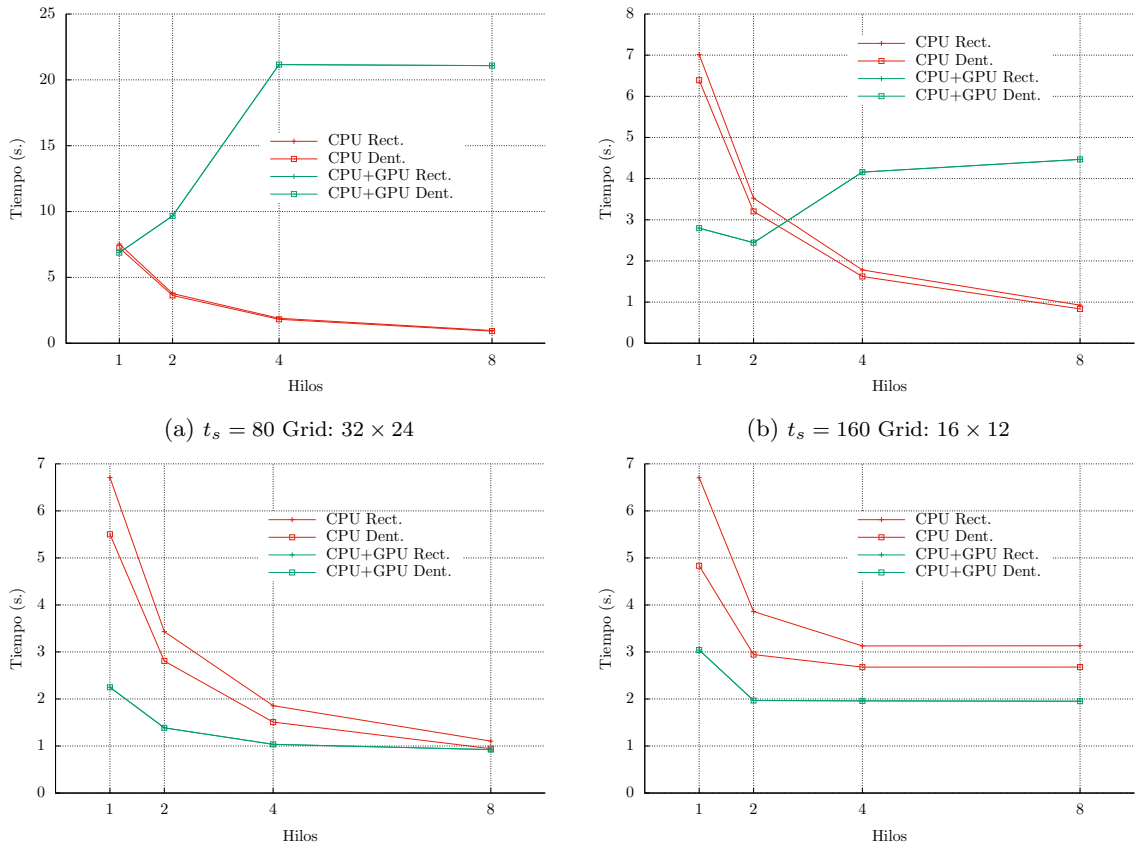

(a) $t_s = 80$ Grid: $16 \times 12$

(b) $t_s = 160$ Grid: $8 \times 6$

(c) $t_s = 320$ Grid: $4 \times 3$

Figura 3.21: Tiempo de ejecución para calcular la factorización QR utilizando el Algoritmo 7 (*Rectangular*) y el Algoritmo 8 (*Dentada*) para un tamaño de matriz de $1280 \times 960$ en CPU y CPU+GPU en función del número de hilos en el *Jetson AGX Xavier*.

que es la productividad del *pipeline* (Figura 3.23c) los resultados son buenos, tal como mostraba la Figura 3.19b para el *Xeon*. Las Figuras 3.23b y 3.23d muestran, respectivamente, el tiempo de ejecución y la productividad del *pipeline* utilizando también la GPU. Se aprecia que, para este tipo de mallas de $4 \times 3$ *tiles*, la utilización de la GPU aporta un valor importante. Aunque la diferencia de tiempo entre CPU y GPU para, por ejemplo, matrices pequeñas de tamaño $640 \times 480$ no se aprecia debido a la escala de tiempos utilizada, las gráficas de productividad permiten observar un notable incremento en la cantidad de matrices procesadas por unidad de tiempo.




(a) $t_s = 80$ Grid: $32 \times 24$

(b) $t_s = 160$ Grid: $16 \times 12$

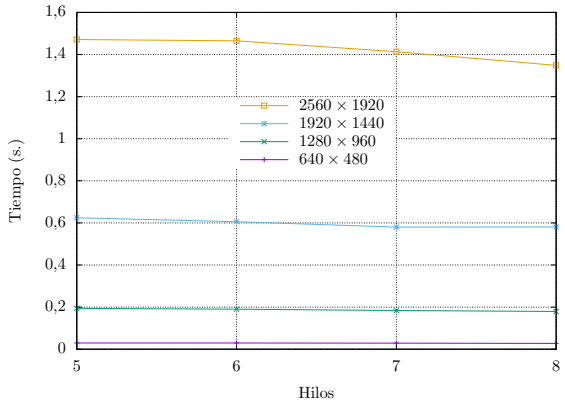
(c) $t_s = 320$ Grid: $8 \times 6$

(d) $t_s = 640$ Grid: $4 \times 3$

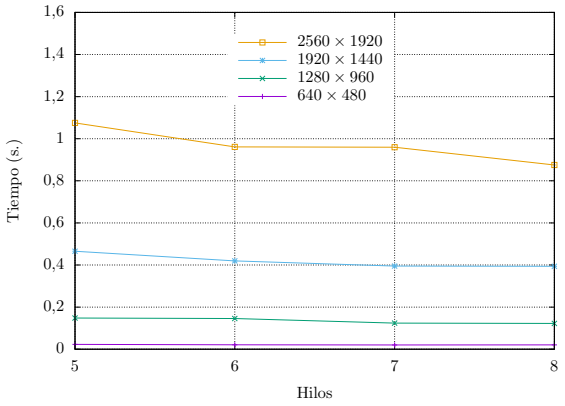
Figura 3.22: Tiempo de ejecución para calcular la factorización QR utilizando el Algoritmo 7 (*Rectangular*) y el Algoritmo 8 (*Dentada*) para un tamaño de matriz de $2560 \times 1920$ en CPU y CPU+GPU en función del número de hilos en el *Jetson AGX Xavier*.

## 3.8 Conclusiones

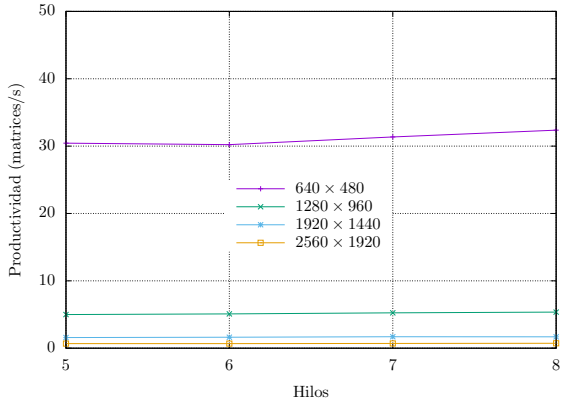
La factorización QR de una matriz rectangular constituye un núcleo computacional básico utilizado en muchas aplicaciones científicas. La aplicación que se ha abordado en este capítulo, el algoritmo de *Beamforming*, se define mediante una matriz rectangular que cambia rápidamente (en tiempo real) en base a un conjunto dado de señales de entrada que se obtienen de muestrear una señal de sonido digital. La factorización de esta matriz, que se requiere para construir las señales de salida correspondientes, es la tarea que consume más tiempo de las que forman parte del algoritmo y debe calcularse periódicamente y de manera rápida para poder realizar un seguimiento de las entradas. Primero, hemos demostrado que es posible acelerar esta factorización mediante la utilización de una matriz especial que hemos denominado *dentada*. Su particular estructura, que mantiene una serie de



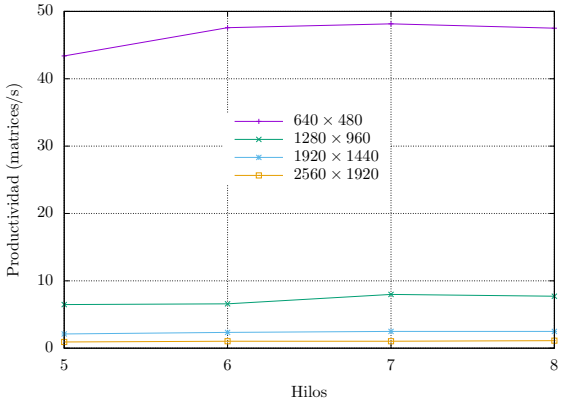

(a) Tiempo de la Etapa 4 en CPU.




(b) Tiempo de la Etapa 4 en CPU+GPU.




(c) Productividad del *pipeline* en CPU.




(d) Productividad del *pipeline* en CPU+GPU.

Figura 3.23: Tiempo de factorización QR en la Etapa 4 y productividad del *pipeline* en el *Jetson AGX Xavier* con CPU solo y con GPU añadida.

ceros, permite ahorrar operaciones. Segundo, hemos demostrado que organizar las operaciones en forma de *pipeline* permite adelantar operaciones de la factorización y, por tanto, obtenerla con menos coste computacional con cada nueva entrada de datos.

La idea básica del *pipeline* ha sido planteada como un DAG especial, muy cercano a un pipeline (cuasi-pipeline), que, actualmente, se puede implementar con la herramienta de programación OpenMP gracias a la cláusula `depend`. Es más, mediante la directiva `taskgroup` es posible inducir paralelismo dentro de cada una de las etapas del *pipeline*. Sin embargo, con el fin de utilizar herramientas de programación de un nivel superior a OpenMP, hemos "aplanado" el *pipeline* anterior o cuasi-pipeline para derivar un *pipeline* real. Esta nueva estructura de *pipeline* real se puede implementar fácilmente usando la herramienta GrPPI, lo que aporta ventajas en la codificación como, por ejemplo, evitar la construcción explícita de colas de datos entre las etapas. Comparando las dos herramientas podemos considerar que ambas se encuentran casi al mismo nivel de rendimiento. La utilización del *pipeline*,

además, permite aprovechar más el paralelismo acelerando 2 y 3 veces el algoritmo. Aunque este mecanismo induce un retardo en el cálculo, este es admisible en este tipo de aplicaciones.

Solo queda decir que en este capítulo se ha abordado el problema particular en el que la matriz del sistema forma una malla de $4 \times 3$ *tiles*. Esta forma surge cuando la matriz del sistema cambia con la adición de un 25 % de filas "nuevas" en sustitución de otro tanto de filas "antiguas" en cada paso. Sin embargo, la propuesta puede extenderse fácilmente a un caso más general en el que el porcentaje de filas sea diferente. Tal como hemos mostrado en algunos experimentos, si el número de filas a actualizar es menor del 50 %, obtenemos un grado de paralelismo mayor pudiendo aprovechar así una mayor cantidad de cores. sin embargo, hay que observar que, si el tamaño de *tile* se reduce, la utilización de la GPU puede ser contraproducente.

# Capítulo 4

# Estimación de profundidad basado en hardware especial

> Atrás hemos dejado el desarrollo de filtros para señales unidimensionales. Una imagen digital viene típicamente representada como una señal bidimensional, estructuralmente una matriz numérica.

Esta segunda parte de la tesis está basada en las siguientes patentes para cámaras plenópticas que son propiedad intelectual de la empresa photonicSENS®:

1. DEVICE AND METHOD FOR OBTAINING DEPTH INFORMATION FROM A SCENE [10]).
2. *LIGHT-FIELD* OPTICAL IMAGE SYSTEM WITH DUAL MODE [3]).

En ambas patentes ha participado activamente el doctorando, siendo además autor de [3]. La Patente [3] se presenta como contribución principal de esta parte de la tesis.

## 4.1 Introducción

Dentro de las diferentes formas de trabajo con las que podemos mostrar la geometría de una escena tridimensional encontramos que ésta puede representarse mediante una lista de coordenadas 3D del mundo real o bien, equivalentemente, a partir de una imagen 2D y de un mapa de profundidad. Una imagen de profundidad es básicamente una matriz que devuelve la profundidad en cada una de sus posiciones, estando estas posiciones (filas y columnas) relacionadas con las coordenadas de la imagen [77].

Las principales tecnologías capaces de obtener el mapa de profundidad de una escena son:

1. las cámaras de tiempo de vuelo (*Time Of Flight (ToF)*),
2. las cámaras de luz estructurada (*Structured Light*), y
3. las cámaras plenópticas o de campo de luz (*plenoptic* o *light-field*).

Figura 4.1: Ejemplo de cámara 3D, en este caso la **Intel Realsense D435**. A partir de dos imágenes, izquierda y derecha separadas por un *baseline*, se reconstruye un mapa de profundidad (centro) donde los colores más claros representan mayor profundidad, siendo el color blanco zonas indefinidas.

4. las cámaras stereo.

5. la tecnología Lidar.

Con cualquiera de las anteriores tecnologías que son explicadas con mayor detalle en [67], la representación de una escena tridimensional devuelve la mayor parte de las veces una visión incompleta de la misma, ya sea por oclusiones[1], reflejos o zonas de la escena difíciles de resolver por la falta de textura. Por lo tanto, se hace necesario un proceso final de rellenado de estas zonas incompletas por medio de algoritmos de procesamiento de imagen o redes neuronales entrenadas para dicho fin. Como caso práctico de este problema podemos tomar la familia de productos Realsense® de Intel®. La Intel® relasense® D435, utiliza dos cámaras para poder calcular la profundidad y poder proporcionar entonces un mapa de profundidad de la escena. En la Figura 4.1 se puede apreciar un mapa de profundidad (imagen central) de la escena capturado con un dispositivo Realsense, correspondiente al par estéreo formado por la imagen de la izquierda y la imagen de la derecha. Como se aprecia en el mapa de profundidad, existen zonas en blanco (sin valor) donde el dispositivo ha sido incapaz de estimar la profundidad para estos píxeles por el efecto de las oclusiones. En estas zonas se emplean diferentes algoritmos de rellenado del mapa de profundidad que implementa la Realsense, de tal forma que obtengamos un valor de profundidad para la totalidad de los píxeles de la imagen.

A parte de las oclusiones, otra problemática que nos encontramos a la hora de obtener un mapa de profundidad completo son las zonas sin textura, donde los algoritmos son incapaces de encontrar correspondencias entre los puntos de las imágenes estéreo. Para evitar grandes zonas sin valor de profundidad se pueden emplear soluciones activas complementarias como la luz estructurada o como la proyección de patrones de puntos distribuidos sobre la imagen, que es la solución que implementa la Realsense. Esto nos permite obtener información "extra" que ayuda al algoritmo de cálculo de profundidad a encontrar las correspondencias entre píxeles del par estéreo.

[1] El objeto de interés puede estar parcialmente oculto tras otro objeto en una imagen.

El proceso de rellenado completo (reconstrucción densa [17]) de una escena es una de las necesidades más demandadas en dispositivos móviles durante los últimos años (para aplicaciones fotográficas, juegos, etc.). Sin embargo, esta necesidad es una tarea computacionalmente muy costosa y que en muchos casos se debe realizar en todos los *frames* de una secuencia de vídeo. Por lo tanto, estamos frente a uno de los campos de investigación más importantes y prolíficos de nuestros días, el cual necesita apoyarse en el cálculo masivo paralelo que aportan las GPU de los dispositivos móviles actuales. Esta programación GPU (o en general heterogénea para aprovechar toda la potencia de los dispositivos actuales) puede realizarse a través de lenguajes de programación multiplataforma como *OpenCL* (*Open Computing Language*).

Los algoritmos para la extracción del mapa de profundidad de una escena presentados en esta tesis están descritos en la patente europea [10] y en la patente depositada [3]. El doctorando ha participado activamente en todas las funciones y subprocesos de dichos algoritmos, siendo autor principal de la parte de rellenado completo de la escena. En definitiva, dichas patentes describen todo el proceso de interpretación de los datos procedentes de una cámara de campo de luz y cómo convertir dicha información en valores de profundidad para todos los píxeles de una escena.

### 4.1.1 Organización, estructura y definiciones

Esta segunda parte de la tesis se estructura de la siguiente manera. Los siguientes párrafos del presente capítulo describen los elementos que componen una cámara plenóptica, así como el funcionamiento de la misma. Posteriormente, se describe un sistema dual óptico (dos modos de funcionamiento) capaz de capturar tanto una imagen de campo de luz, como una imagen de resolución total 2D. Finalmente, se describen los algoritmos desarrollados para el cálculo de profundidades de una forma computacionalmente óptima.

En el Capítulo 5 se describe con detalle el algoritmo de rellenado de los valores de profundidad faltantes, basado en una estrategia de K vecinos más cercanos (*K-nearest neighbords*, KNN) a partir de los valores de profundidad de los píxeles de la escena que sí tengan valor asignado. Dicho algoritmo es uno de los algoritmos más empleados en imagen por ordenador para clasificación de clases sin etiqueta y que nosotros vamos a utilizar para proporcionar valores de profundidad a los píxeles de la escena que no los tengan, es decir, para hacer el rellenado completo (proceso conocido como *filling*).

Finalmente, en el Capítulo 6 se realiza un análisis de los resultados obtenidos por los diferentes algoritmos presentados en esta parte de la tesis, sus problemáticas y las soluciones propuestas, así como los trabajos futuros para la continuación del desarrollo de esta investigación.

#### Algunas definiciones previas

1. **Cámara plenóptica:** Dispositivo capaz de capturar no sólo la información espacial de la escena, sino también la dirección de llegada de los rayos de luz entrantes.

2. **Sistema multi-vista:** Sistema capaz de capturar una escena desde diferentes puntos de vista.

Una cámara **plenóptica** puede considerarse un sistema multi-vista. Un sistema **estéreo** o **multi-estéreo** de cámaras también es considerado un sistema **multi-vista**.

3. **Campo de luz (*light-field* o LF):** Estructura de cuatro dimensiones (cinco si es un LF con color) que contiene la información espacial y angular de la luz capturada por los píxeles debajo de las microlentes en una cámara **plenóptica**.

4. **Conjunto de microlentes (*Microlens array - MLA*):** Conjunto de pequeñas lentes (microlentes) que abarcan un número determinado y regular de píxeles del sensor.

5. **Vista plenóptica:** Imagen bidimensional (2D) formada a partir de un subconjunto de píxeles de la estructura de *light-field*. Este subconjunto de píxeles no es continuo en el sensor, sino que se eligen ciertos píxeles del sensor debajo de cada una de las microlentes.

6. **Profundidad:** Distancia entre el plano de un punto del objeto de la escena y el plano principal de la cámara, ambos planos son perpendiculares a los ejes ópticos de cada vista.

7. **Mapa de profundidad:** Matriz bidimensional (2D) en la cual se han calculado valores de profundidad de los objetos de la escena (distancia a la cual se encuentran), para algunos (disperso) o todos (denso) los píxeles que forman la escena.

8. **Disparidad:** Distancia entre dos (o más) proyecciones de un mismo punto de la escena a cada vista plenóptica de la cámara.

9. ***Baseline*:** Distancia entre los centros de dos (o más) cámaras. En caso de una configuración estéreo (o multi-estéreo), comparten un mismo plano con sus ejes ópticos paralelos a la escena.

### 4.1.2 Descripción de la tecnología

Una vez descrita la finalidad de los algoritmos que se defienden en la presente tesis, es importante describir la tecnología sobre la que éstos han sido implementados y que forma el *apparatus* de las patentes anteriores.

Las cámaras de campo de luz o cámaras plenópticas[2] son dispositivos de captura de imágenes. Ideadas por Lippmann en 1907, las cámaras plenópticas no únicamente capturan la información espacial de la escena, sino también la información angular (la dirección) de los rayos de luz que inciden en la cámara.

Una cámara plenóptica consta básicamente de una lente de campo que realiza la función de proyectar la imagen de la escena 3D sobre un punto más o menos próximo a las microlentes que se encuentran sobre el sensor. Las microlentes junto al sensor de imagen, típicamente CMOS, tienen la capacidad intrínseca de registrar la información espacio-direccional. En definitiva, los rayos de luz pasan por la

[2]La formación de campos de luz se puede llevar a cabo mediante diferentes configuraciones que forman en definitiva una matriz de imágenes, en nuestro caso el campo de luz como resultado de salida de una cámara plenóptica hará que utilicemos ambos los conceptos de "campo de luz" y "cámara plenóptica" indistintamente.

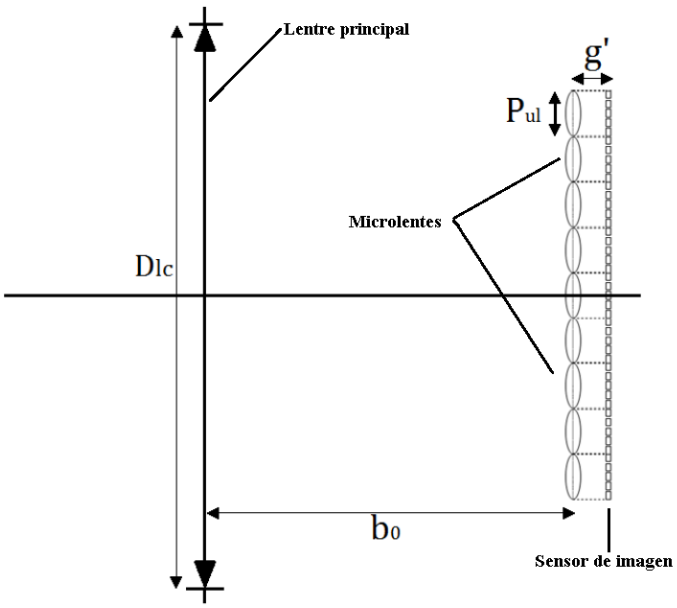



Figura 4.2: Vista esquemática bidimensional de una cámara plenóptica.

lente de campo y las microlentes e impactan sobre el sensor. La Figura 4.2 muestra un esquema general de una cámara plenóptica. Sobre la figura se definen algunas de las distintas características:

1. **Dlc**: Diafragma de apertura de la lente de campo.

2. $\mathbf{b}_0$: distancia de la lente de campo al array de microlentes.

3. $g\prime$: distancia del array de microlentes al sensor.

4. $P_{ul}$: *pitch* de microlentes (radio de curvatura de la esfera de cada microlente).

Un "campo de luz" (*light-field*) se representa como una función cuatro[3] dimensional *LF(px, py, lx, ly)*, donde *px* y *py* registran las coordenadas angulares o direccionales de los rayos y *lx* y *ly* representan la posición espacial de los rayos. Esta información angular no es "gratis", pues la obtención de dicha información lleva aparejada la pérdida de resolución lateral, que pasa a ser la resolución de la matriz de microlentes (*lx* y *ly*), siendo esta resolución mucho menor que la resolución del sensor de imagen de la cámara empleada. Un ejemplo comercial de cámara plenóptica es la cámara *Lytro* [33]. Esta cámara, en su primera generación, tiene un sensor de alrededor de 11 megapíxeles, produciendo una resolución angular de $11\times11$ y resolución espacial de menos de 0.15 megapíxeles. La segunda generación de dicha cámara tiene un sensor de 40 megapíxeles y, a pesar de este tamaño de sensor, resulta en solo

[3]Seguiremos generalizando para el ejemplo de sensor monocromo, aunque un LF tendría cinco dimensiones con el color.

4 megapíxeles de resolución espacial (con la codificación del fabricante) debido a la compensación de resolución angular-espacial.

Este inconveniente tiene difícil aceptación en el mundo de los megapíxeles en el que vive el mercado de la telefonía móvil, el cual se ha convertido en una dura batalla por proporcionar al usuario la mejor cámara profesional (con la mayor resolución lateral de la escena inclusive), dentro de los dispositivos móviles. Es por ello que, a la vista de la dificultad de introducir la tecnología plenóptica en este mercado tan goloso, se ha trabajado en la Patente [3], la cual permite, partiendo de una única cámara, obtener dos modos de funcionamiento radicalmente distintos en términos de resolución y estimación de profundidad, esto es, proporcionándonos las bondades de ambos tipos de cámaras, cámaras plenópticas y cámaras digitales convencionales. Así, en [3] se proporciona un mecanismo capaz de cambiar entre un modo de campo de luz (o modo de profundidad) y un modo de resolución completa (o modo de imagen 2D), compartiendo ambos modos un mismo eje óptico. Cuando la cámara está trabajando en su modo de resolución total, no es posible calcular la profundidad de los objetos del mundo real, como sí se puede hacer con la cámara trabajando en modo campo de luz.

### 4.1.3 Trabajos relacionados

Como ya se ha dicho, un *light-field* es una matriz 4-dimensional que contiene la información espacial $(lx, ly)$ y la dirección de procedencia de los rayos de luz $(px, py)$ (coordenada angular) para cada uno de los píxeles de la escena. En las siguientes subsecciones se repasan brevemente los trabajos de cálculo de un mapa de profundidad (*depth map*) con esta matriz 4D $(lx, ly, px, py)$ así como los trabajos relacionados con los procesos que implican dicho cálculo.

#### Estimación de profundidad en *Light-fields*

Durante los últimos años, la investigación para la estimación de profundidad a partir de un LF de entrada ha experimentado un crecimiento espectacular. Básicamente, las soluciones que encontramos en la literatura para dicha estimación parten de las características particulares de un LF. Así, podemos encontrar trabajos que se basan en *photo-consistency* [74], imagen epipolar (EPI) [39], *depth-from-focus* (DFF) [53], *depth-from-defocus* (DFD) [62] y métodos basados en aprendizaje [75].

En el artículo [72] podemos encontrar un repaso histórico a la evolución en el campo de las cámaras de campo de luz durante más de 20 años de investigación.

#### Aceleración del procesado de *Light-fields*

Muchos de los trabajos donde se ha usado la GPU sobre LFs han sido trabajos en los que se renderizaba una imagen o se calculaba de manera muy simple una estimación de la profundidad [26]. Sin embargo, en los últimos años, están proliferando los trabajos que emplean la GPU de dispositivos móviles para la estimación de profundidad en LFs con una significativa aceleración en el cálculo de dicha estimación [35]. Además, ciertas tareas del procesado de imágenes ya se llevan a cabo por hardware dedicado dentro de los procesadores [71].

Con la finalidad de "acelerar" el computo de los diferentes algoritmos y que estos sean portables a diferentes plataformas *hardware*, nace *OpenCL* (*Open Computing Language*, en español lenguaje de computación abierto) es un lenguaje de programación estándar y multiplataforma. *OpenCL*, junto a una interfaz de programación, permite crear aplicaciones para el procesado paralelo sobre sistemas heterogéneos (CPUs, GPUs y procesadores digitales de señal DSP) [50], siendo además un lenguaje de programación "traducible" directamente a diferentes compañías de fabricación de unidades computacionales, sin tener que realizar cambio alguno en el código.

**Procesamiento de LFs en dispositivos móviles**

Desde hace unos años, la industria de la telefonía móvil se ha visto en la necesidad de incorporar la información de profundidad *depth map* de las escenas capturadas por las cámaras para desarrollar aplicaciones muy demandadas por los usuarios, como por ejemplo el efecto *bokeh* [25] en el modo retrato o aplicaciones de realidad virtual, aumentada o mixta [64] y, es por ello, que han surgido tecnologías que compiten por ofrecer este contenido tridimensional. Las cámaras plenópticas se enfrentan actualmente a dos grandes retos para su incorporación a dispositivos móviles, por un lado se deben desarrollar unos algoritmos extremadamente eficientes y en constante desarrollo para procesar los LFs y, por otro lado, el reto es todavía mayor cuando se trata de integrar dicha tecnología "dentro" de los dispositivos móviles [34, 65].

## 4.2 Formación de un LF mediante un sistema dual óptico

En esta sección se resume la Patente [3] de un sistema micro-electro-mecánico (MEMS, del inglés *microelectromechanical systems*). En un sistema compuesto por una lente principal (que puede estar formada por una o varias lentes), un array de microlentes y un sensor de imagen (típicamente un sensor CMOS), se patentan dos actuadores: un primer actuador permite intercambiar entre dos modos de configuración, produciendo un desplazamiento relativo entre el sensor de imagen y el array de microlentes (distancia $g\prime$) para intercambiar entre dos configuraciones posibles, configuración de campo de luz (*light-field*) y la configuración 2D de una cámara digital convencional. Un segundo actuador permite auto-enfocar la imagen en el modo de configuración 2D de forma mecánica.

En conjunto, la Patente [3] comprende un primer actuador configurado para causar un desplazamiento relativo que modifica la distancia entre el sensor de imagen y la matriz de microlentes. Esto permite cambiar entre dos configuraciones ópticas diferentes, una configuración de campo de luz en la que el sensor de imagen y la matriz de microlentes están separados por una primera distancia ($g_p\prime$) y una configuración de imagen 2D en la que el sensor de imagen y la matriz de microlentes están separados por una segunda distancia ($g_c\prime$), inferior a la primera distancia y próxima a 0, que permite al sensor de imagen evitar el efecto de las microlentes.

La Figura 4.3 representa con un diagrama de bloques los elementos principales de un sistema óptico tal y como se concibe en la invención de la Patente [3]. Además de la lente de campo, el array de microlentes y el sensor de imagen, el sistema de imagen óptica de campo de luz de la invención

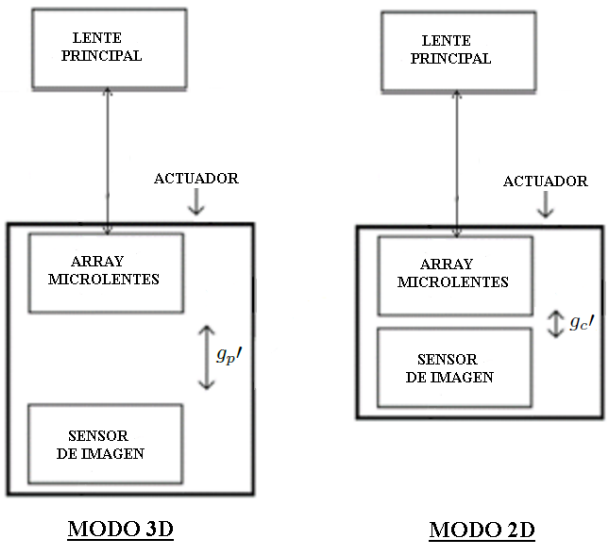



Figura 4.3: Vista esquemática de una cámara plenóptica con actuador para modificar la distancia $g\prime$.

incluye un primer actuador para modificar la distancia $g\prime$, permitiendo cambiar el comportamiento de la cámara entre dos modos de funcionamiento: el modo 3D (*configuración plenóptica*) y el modo 2D (*configuración de imágenes 2D* convencionales).

En la **configuración plenóptica**, el sensor de imagen y el array de microlentes están separados por una primera distancia $g_p\prime$ que permite al sensor de imagen capturar una imagen de campo de luz, trabajando así como una cámara plenóptica (ver Figura 4.3 - izquierda), siendo esta distancia $g_p\prime$ normalmente cercana a la distancia focal de cada una de las lentes pequeñas del array de microlentes. El rango de distancias de los objetos de la escena que se pueden resolver en esta configuración depende de la focal de la lente principal y del *baseline*. Así, en una cámara diseñada para telefonía móvil, esta distancia focal será pequeña (unos pocos milímetros) y por lo tanto el rango de distancias que se pueden resolver será muy corto también (hasta 1 metro aproximadamente). Dicha distancia focal será mayor en dispositivos de mayor tamaño y con mayores ópticas, resolviendo así escenas con distancias mayores.

En una **configuración de imágenes 2D** (o modo de resolución completa) el sensor de imagen y el array de microlentes están separados por una segunda distancia $g_c\prime$, inferior a la distancia de la configuración plenóptica (muy próxima a 0), que permite al sensor de imagen evitar el efecto de campo de luz y actuar como una cámara convencional (ver Figura 4.3 - derecha).

Nótese que el sistema de la citada Patente [3] permite:

1. una configuración para la adquisición de una imagen plenóptica o una imagen 2D estándar en un solo disparo de cámara, es decir, una configuración donde la cámara puede funcionar como una cámara digital 2D estándar o como una cámara plenóptica; o

2. una configuración para la adquisición de una imagen plenóptica y otra imagen 2D estándar si se efectúan 2 disparos de cámara, pudiendo ser el primero con el array de microlentes alejado del sensor una distancia $g_p\prime$ igual a la distancia focal de las microlentes y el segundo con el array de microlentes pegado al sensor, es decir, con una distancia $g_c\prime$ muy próxima a 0, o viceversa. Los algoritmos que se aplican a la parte plenóptica se explican en la Sección 4.3.

Nótese también que el segundo de los modos de funcionamiento considera tomar dos imágenes (2 *shots*), teniendo tanto la imagen de campo de luz para el cálculo del mapa de profundidad de la escena como la imagen 2D con la resolución lateral completa proporcionada por el sensor de imagen. Por lo tanto, esto nos permite obtener el mapa de profundidad de la escena en esta misma resolución mediante el uso de algoritmos de *upsampling* guiados por la imagen [16, 31, 40], siendo este mapa de profundidad de gran calidad ya que se guía de una imagen 2D a resolución completa que no ha sido *"upsampleada"* (sin *estimar* información por interpolaciones).

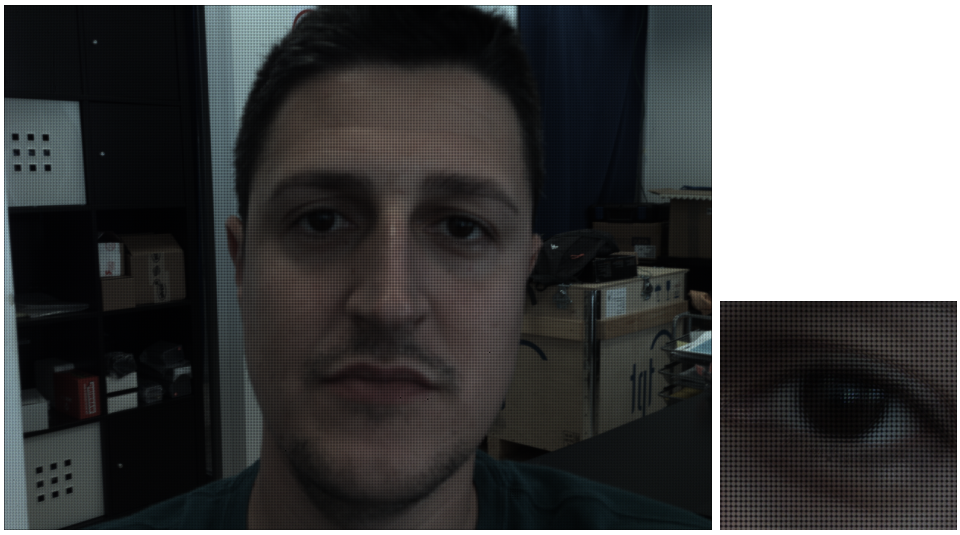
La implementación utilizada en esta tesis se basa en [31] ya que, además de ser una implementación muy conocida en la literatura, principalmente es la que mejor resultados ha proporcionado en nuestro caso. Sin embargo, se han realizado pruebas con otros algoritmos de *upsampling* [16] que se comparan con [31], saliendo ganadores en diferentes apartados de la comparativa. Aunque somos conscientes de estos trabajos, en nuestro caso esto no sucede así y los mejores resultados se obtienen con [31].

## 4.3 Algoritmos de estimación de profundidad

La obtención del mapa de profundidad de las escenas capturadas se realiza mediante la implementación de unos novedosos algoritmos capaces de obtener dicho mapa de profundidad mediante el procesamiento de LFs capturados por una cámara plenóptica (Figura 4.2) o por cualquier otro dispositivo de adquisición de LFs. El método patentado en [10] es muy eficiente computacionalmente, por lo que se puede utilizar para obtener mapas de profundidad en tiempo real incluso en dispositivos móviles de bajo consumo, que incorporan unidades computacionales con ratio eficiente GFlop/Watt. La eficiencia en los algoritmos no solo viene a proporcionar tiempo real a los resultados que se obtienen, sino que se necesitan dichos cálculos eficientes para evitar agotar las baterías rápidamente. Así, estos algoritmos permiten obtener imágenes 3D mediante cámaras plenópticas en dispositivos móviles que procesen vídeo en tiempo real (a 30 fotogramas por segundo o incluso más) mediante la detección de los bordes de los objetos que componen la escena y calculando la profundidad a la que se encuentran estos bordes identificados.

### 4.3.1 Formas de interpretar un *light-field* (LF)

En los siguientes párrafos procederemos a profundizar en la explicación de cómo se puede interpretar la información recogida por el sensor de imagen de nuestra cámara plenóptica, es decir, cómo se puede interpretar el campo de luz o *light-field* (LF).



(a) Raw Completo. (b) Zoom ojo.

Figura 4.4: Ilustra una imagen completa con los valores directamente leídos del sensor (sin tratamiento alguno) conocido como *RAW*.

### Raw image

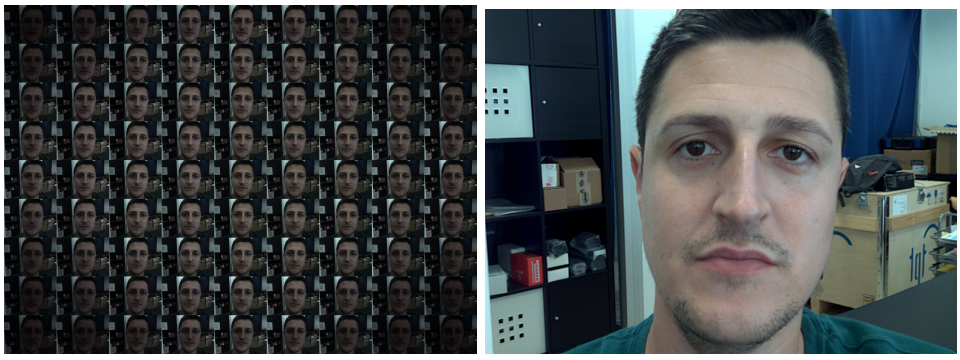
La forma más sencilla de interpretar un LF es lo que se conoce como *raw image* (imagen en bruto, sin tratamiento de ningún tipo), la cual consiste en tomar para cada píxel el valor leído directamente del sensor de imagen.

En la Figura 4.4 podemos apreciar que la *raw image* está compuesta por un conjunto de micro-imágenes, igual al número de microlentes. En este ejemplo, las microimágenes se aprecian circulares porque la apertura óptica de la lente principal lo es, y las microlentes hacen una imagen de dicha apertura óptica. Los bordes en negro de cada microimagen se corresponden con la pérdida de información derivada de un bloqueo de luz por parte del sistema óptico, provocando también que la imagen en su conjunto parezca más oscura. En el zoom mostrado en la Figura 4.4 podemos apreciar con más detalle esta forma circular que muestran cada una de las microlentes que forman el sistema óptico.

### Vistas

Las vistas se forman tomando el mismo píxel debajo de cada una de las microlentes que forman el *array* de microlentes, es decir, el píxel con la misma dirección $(px, py)$. Por tanto, el número de vistas diferentes que vamos a poder obtener es igual al número de píxeles debajo de cada microlente. En la Figura 4.5 (derecha) se muestra una vista ejemplo que se obtiene para la *raw image* anterior, en concreto, se muestra la vista central que se obtiene seleccionando el pixel central de cada una de las microlentes que componen el sistema óptico.

Si comparáramos un par de vistas cualesquiera de las que se pueden apreciar en la Figura 4.5 nos



(a) Multi-vista. (b) Vista Central.

Figura 4.5: Izquierda, se muestra la apertura de la lente de campo y cómo se distribuyen las diferentes $9 \times 9$ vistas a lo largo de ella (vistas procedentes de las diferentes cámaras equivalentes) para cada dirección $(px, py)$. Derecha, se muestra la vista central (es decir, del array de vistas de la izquierda, la imagen en la posición $(5, 5)$) del conjunto de vistas de las diferentes cámaras equivalentes.

daríamos cuenta de que ambas imágenes no son iguales, existe un desplazamiento vertical y/u horizontal (paralaje) entre ambas vistas, dependiendo de qué vistas seleccionásemos. Este desplazamiento sería vertical si ambas vistas se encuentran en la misma columna, horizontal si estuvieran en la misma fila y, tendría ambos paralajes si ambas vistas no se encontrasen ni en la misma fila ni en la misma columna.

**Epipolares**

La representación de un LF con epipolares es la forma de visualización más abstracta del mismo. Sin embargo, es la más importante y empleada por los algoritmos para el cálculo de profundidad de la escena. Una epipolar es una sección 2D de la información presente en un LF. Así, siguiendo la nomenclatura anterior, $(ly, py)$ son las filas del *light-field* donde $ly$ representa la microlente en esa dimensión y $py$ representa la fila del pixel dentro de dicha microlente y $(lx, px)$ son las columnas del LF donde $lx$ representa la microlente en esa dimensión y $px$ representa la columna del pixel dentro de dicha microlente, fijamos una de las tuplas, $(ly, py)$ ó $(lx, px)$ y recorremos al completo la otra tupla no fijada para obtener los datos completos de una epipolar. Si se fija la tupla $(ly, py)$, estaríamos obteniendo una epipolar horizontal, mientras que si fijamos la tupla $(lx, px)$, estaríamos obteniendo una epipolar vertical.

En la Figura 4.6 se busca mostrar de una forma más visual cómo serían cada una de las posibles interpretaciones de un LF. Podemos ver el plano de las microlentes (al frente) y el plano del sensor (al fondo). En este ejemplo, se muestra un plano de 9 microlentes cuadradas donde se ha coloreado la microlente de la esquina superior izquierda en color azul. Cada una de las microlentes se corresponde con una matriz de $3 \times 3$ píxeles del sensor que se encuentra en el plano trasero, configurando un ejemplo de sensor de $9 \times 9$ píxeles (plano trasero al completo).

Por tanto, siguiendo el ejemplo de la Figura 4.6, podemos hacer referencia a las diferentes formas

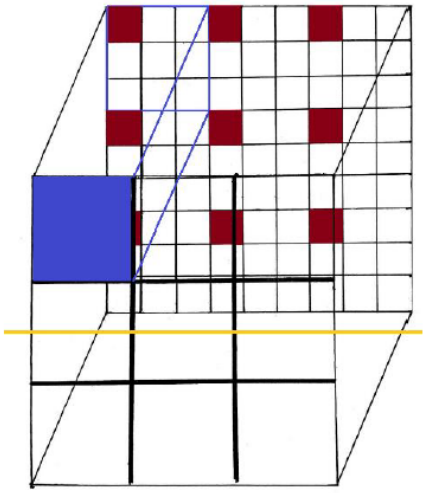


Figura 4.6: Ilustra las tres formas de interpretar el *light-field* (idealización).

de visualización descritas anteriormente:

1. La *raw image* correspondería a tomar todos los píxeles del sensor y mostrar la imagen tal cual ésta es capturada.

2. Una de las *sub-aperture images* o vistas se formaría tomando los píxeles coloreados en color granate (vista superior izquierda de las 9 vistas posibles), es decir, tomar el mismo píxel debajo de cada microlente.

3. Un ejemplo de línea *epipolar* sería la línea amarilla de la figura (epipolar horizontal). Al recorrer en esa dirección el sensor involucraríamos a 3 microlentes, recolectando los tres píxeles por cada una en una misma línea del sensor.

En la Figura 4.7 se puede observar una imagen con las epipolares formadas para la imagen de ejemplo utilizada (centro). Se muestra una epipolar vertical en el margen izquierdo de la imagen, y una epipolar horizontal en el margen inferior de la imagen. Una imagen epipolar horizontal se formará fijando una fila de microlentes del array de microlentes y seleccionando todos los píxeles de cada una de las microlentes en la dirección horizontal que forma dicha fila de microlentes. De forma análoga, para obtener una epipolar vertical se fija una columna de microlentes del array de microlentes y se realiza la misma operación que para las horizontales.

La profundidad de los objetos en una escena 3D puede ser estimada a partir de la pendiente de los bordes que se pueden detectar en cada imagen epipolar. Obviamente, para que esto sea posible, el objeto ha de presentar bordes o texturas que lo posibiliten; si se trata de un objeto o zona sin contraste (totalmente homogénea en sus colores e intensidades) no seríamos capaces de detectar profundidad.

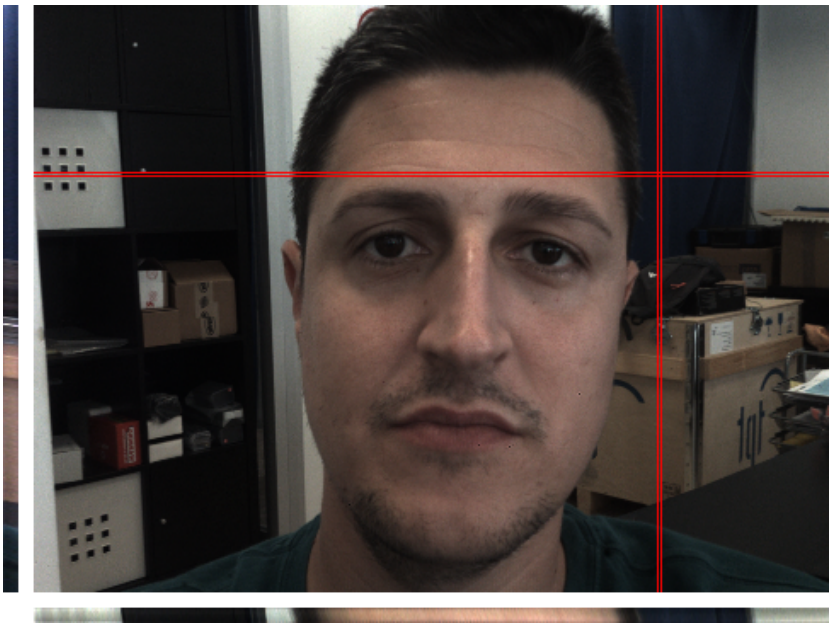
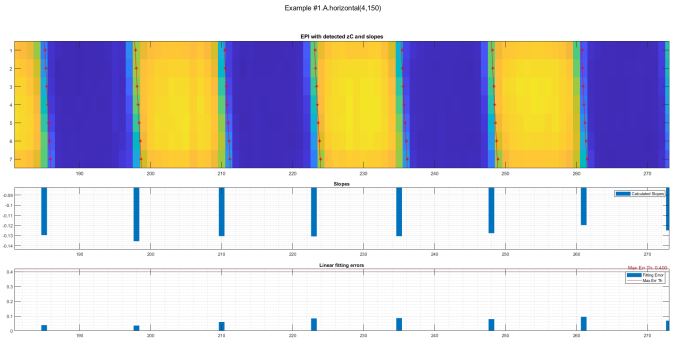


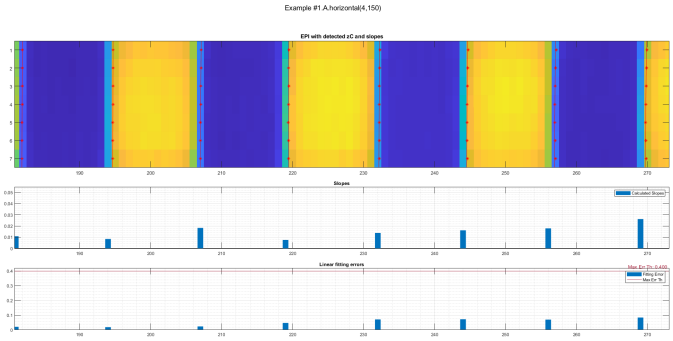
Figura 4.7: Se muestra un ejemplo de epipolar vertical (izquierda de la imagen) y un ejemplo de epipolar horizontal (debajo de la imagen). sobre la imagen se han marcado las epipolares seleccionadas con unas líneas en rojo.

En la Figura 4.8 se muestran tres ejemplos de bordes detectados por la cámara plenóptica diseñada por la empresa photonicSENS®. Los tres bordes que se muestran corresponden al mismo borde detectado en un *target* ejemplo compuesto por una imagen de rayas horizontales negras sobre un fondo blanco. Se aprecian bordes distintos en las 3 imágenes que componen la Figura 4.8, porque estos bordes han sido capturados a diferentes distancias del *target* respecto al plano de la cámara plenóptica. En base a esto, podemos deducir fácilmente por qué las cámaras plenópticas son capaces de detectar la distancia a los objetos. Esto es, primeramente es necesario detectar los bordes dentro de cada epipolar para, seguidamente, obtener el *slope* o pendiente de dicho borde. El valor de esta pendiente nos permitirá obtener directamente la profundidad haciendo uso de una función de conversión pendiente-profundidad, función esta que se calibra para cada cámara.

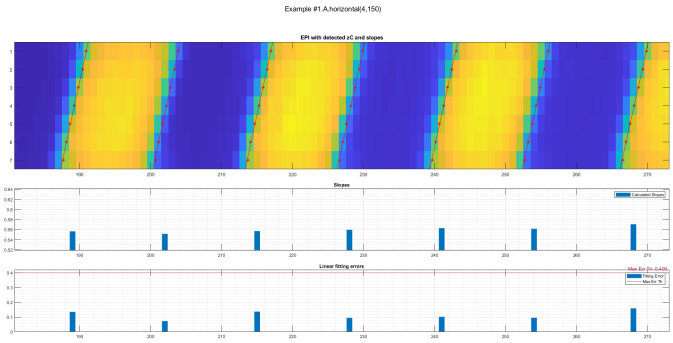
Por lo tanto, la pendiente de las líneas epipolares permite obtener la profundidad de los objetos, ya que son magnitudes directamente relacionadas (el slope es directamente proporcional a la inversa de la distancia [45]). La correspondencia entre el valor de la pendiente y la distancia se puede obtener de manera teórica conociendo los parámetros ópticos de la cámara (focal de la lente, distancia de enfoque, *baseline*...) y también de manera experimental (calibración de la cámara). En general, el $slope = 0$ corresponderá a objetos situados a la distancia de enfoque de la cámara. Los *slopes* negativos y positivos corresponderán a distancias más lejanas y cercanas, respectivamente, a la distancia de enfoque de la lente principal.




(a) Slope Negativo.




(b) Slope 0.




(c) Slope Positivo.

Figura 4.8: Ejemplo de cálculo de slopes.

En la Figura 4.9 se muestra el diagrama de flujo del algoritmo para el cálculo de profundidad en los píxeles de la escena. Los procesos del algoritmo se identifican por el número de su recuadro y se clasifican por colores para diferenciar al proceso al que pertenecen.

En color azul, se engloban los procesos pertenecientes a la Patente [10], procesos que nos proporcionan un mapa de disparidad de la escena "incompleto", pues únicamente proporciona valores de profundidad en los píxeles de los bordes de los objetos que componen la escena.

En color verde, se engloban los diferentes procesos (con más detalle en el recuadro de ampliación) mediante los que se obtiene el valor de disparidad para el resto de píxeles que componen la escena, es decir, para aquellos que no representan un borde, a partir de los valores de disparidad de los píxeles que representan un borde.

Finalmente, en un último proceso, se convierte el valor de *"slope"* o "pendiente" que proporcionan los algoritmos descritos en la Patente [10] a valores de distancia equivalentes para todos y cada uno de los píxeles con información de profundidad.

El método de la Figura 4.9 genera primeramente las imágenes epipolares horizontales (10) y verticales (11) a partir de un LF (00) capturado por una cámara plenóptica. Para cada imagen epipolar (horizontal y vertical) se le aplica un proceso de 3 pasos de los cuales uno es opcional. En el primer paso, que es el opcional, se le aplica un filtro de suavizado, correspondiente a las casillas (20) y (21). En un segundo y tercer paso que van ligados, se aplica la segunda derivada (30) y (31) a las imágenes epipolares, para finalmente aplicar un algoritmo de detección de cruce por 0 (40) y (41) que nos informará de la existencia de un borde en la imagen epipolar. Una vez detectados los bordes, se identifican las líneas epipolares válidas (50, 51) dentro de las imágenes epipolares. A continuación, se calculan las pendientes (60, 61) de estas líneas epipolares válidas y finalmente se combinan ambos mapas de *slopes* horizontal y vertical, para obtener un único mapa disperso de *slopes* como combinación de ambos, donde se tiene un valor de *"slope"* o "pendiente" para cada uno de los píxeles de la escena donde se ha detectado el borde de un objeto.

Resumiendo el proceso para la obtención del mapa de disparidad como pseudo-algoritmo y tomando como ejemplo el caso de las epipolares horizontales (equivalente para las epipolares verticales) sería el siguiente:

1. Para cada imagen epipolar horizontal (10), obtenida para un par de valores fijos: ($py$, $ly$) del *light-field* (00).

    *a*) Aplicar un filtro unidimensional (o superior) a lo largo de la dimensión $lx$ para reducir el ruido, obteniendo una imagen epipolar horizontal filtrada (20).

    *b*) Para cada píxel ($px$, $lx$), calcular la segunda derivada espacial (30) en el píxel ($px$, $lx$) sobre la intensidad de la luz o el contraste de los píxeles a lo largo de la dimensión $lx$.

    *c*) Determinar los bordes (40) del mundo de los objetos analizando las líneas epipolares con precisión de subpíxeles, más específicamente mediante la detección del cruce por cero de las segundas derivadas espaciales.

    *d*) Buscar cada uno de los cruces por cero que están correctamente dispuestos formando una línea epipolar válida (50), descartando líneas epipolares no válidas.

2. Ídem para cada imagen epipolar vertical (11).

3. Para cada línea epipolar válida (50, 51) que se encuentra en las imágenes epipolares horizontales y verticales, los bordes de precisión de subpíxeles se utilizan para determinar la pendiente (60,

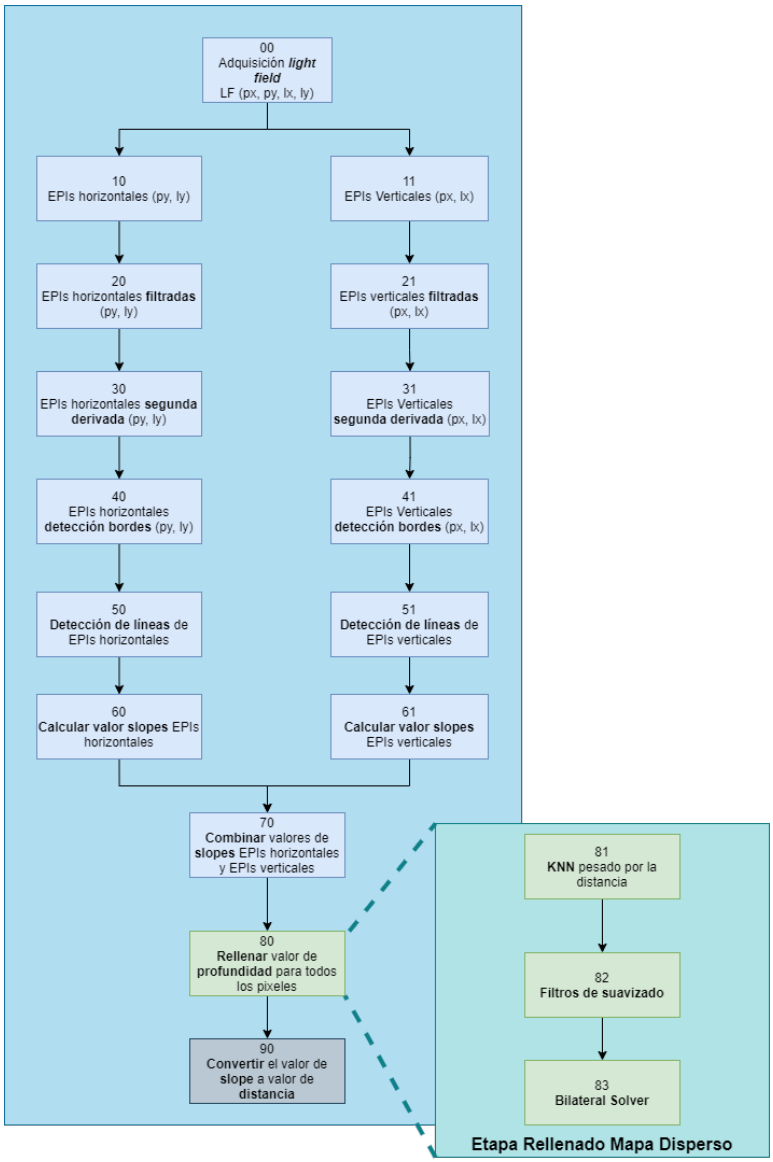



Figura 4.9: Diagrama de flujo del algoritmo para el cálculo de la profundidad de los píxeles de la escena. Los diferentes subprocesos se identifican mediante el número de su rectángulo.

61) de la línea epipolar válida (50, 51) mediante la realización de una técnica de regresión lineal (pero cualquier otra técnica de ajuste a una recta, también podría ser utilizada).

4. Se combinan dos matrices de *slopes*, una matriz de *slopes* horizontal (60) para las imágenes epipolares horizontales y una matriz de *slopes* vertical (61) para las imágenes epipolares verticales.

5. Como último paso para la obtención del mapa de disparidad disperso, las dos matrices de *slopes*

(60) y (61) a partir de las pendientes se combinan en una sola matriz de *slope* que representará el mapa de profundidad disperso (*sparsemap*) de la escena analizada.

6. A la matriz resultado del punto anterior, *sparsemap*, se le aplica algoritmos de rellenado para obtener un mapa denso con valor de *slope* para cada uno de los píxeles de la escena capturada, punto remarcado en verde (80) en la figura y es la contribución desarrollada en la presente tesis.

7. Finalmente, el mapa denso de *slope* obtenido, se le aplica un algoritmo de conversión de *slope* a distancia para obtener el mapa de profundidad denso de la escena.

Los pasos de filtrado para las epipolares horizontales (20) o verticales (21) son filtros de suavizado para los algoritmos de detección de bordes siguientes de segunda derivada y detección de paso por cero, algoritmos éstos que suelen ser muy sensibles al ruido. Este paso puede ser opcional o ser más suave si se utilizan otros algoritmos de detección de bordes más robustos frente al ruido, como por ejemplo el algoritmo de **Canny** [20]. Dependiendo del algoritmo de detección de bordes empleado, los resultados pueden variar en la calidad y cantidad de bordes detectados, así como la eficiencia computacional del algoritmo.

En la Figura 4.9 también se realiza un desglose de los pasos que componen el algoritmo para la obtención de un mapa denso de profundidad de la escena (80). Este paso se divide en tres subprocesos: KNN pesado (81), Filtros de suavizado (82) y *Bilateral Solver* (83). En el Capítulo 5 se explica con más detalle todo este proceso.

# Capítulo 5

# Algoritmo de rellenado para la estimación de profundidad

La idea básica sobre la que se fundamenta el algoritmo de los $k$ vecinos más cercanos KNN, es que cada nueva instancia se va a etiquetar con la clase más frecuente a la que pertenecen sus K vecinos más cercanos.

## 5.1 Introducción

En el presente capítulo se describen los algoritmos de rellenado de un mapa de profundidad disperso desarrollados e implantados en un dispositivo móvil Snapdragon® de la compañía americana Qualcomm®. La técnica de rellenado del mapa de profundidad se desarrolla a partir del mapa disperso que se ha obtenido después de aplicar los algoritmos descritos en la Patente [10] utilizando el diseño de cámara plenóptica descrito en el Capítulo 4 y registrado en la Patente [3] para la adquisición de las imágenes plenópticas. En la Figura 5.1, arriba, podemos observar la imagen RGB capturada y el mapa de profundidad disperso (*sparse map*) proporcionado por los algoritmos descritos en el Capítulo 4; mientras que, abajo, podemos observar la misma imagen RGB con el mapa denso (*dense map*) obtenido aplicando los algoritmos que vamos a describir en el presente capítulo, teniendo únicamente como información de partida el mapa disperso y la imagen RGB mostradas en la imagen superior.

Para el rellenado del mapa de profundidad se ha optado por uno de los algoritmos más extendidos en la literatura de imagen por computador, como es el de los $k$ vecinos más cercanos (KNN). Este algoritmo de aprendizaje automático, muy simple de entender en su funcionamiento y de fácil implementación, es utilizado para dar solución a problemas, tanto de clasificación como de regresión. Estas características han hecho del KNN uno de los algoritmos más famosos de la literatura, siendo hoy en día incluso implementado por *hardware* en algunos de los procesadores de vanguardia.

La implementación del algoritmo del KNN empleada en esta tesis ha sido desarrollada y testeada para las plataformas *embedded* Snapdragon® de la empresa Qualcomm®, teniendo en cuenta la configuración particular de dicho *hardware* para una mejor optimización de recursos. Esta adaptación al

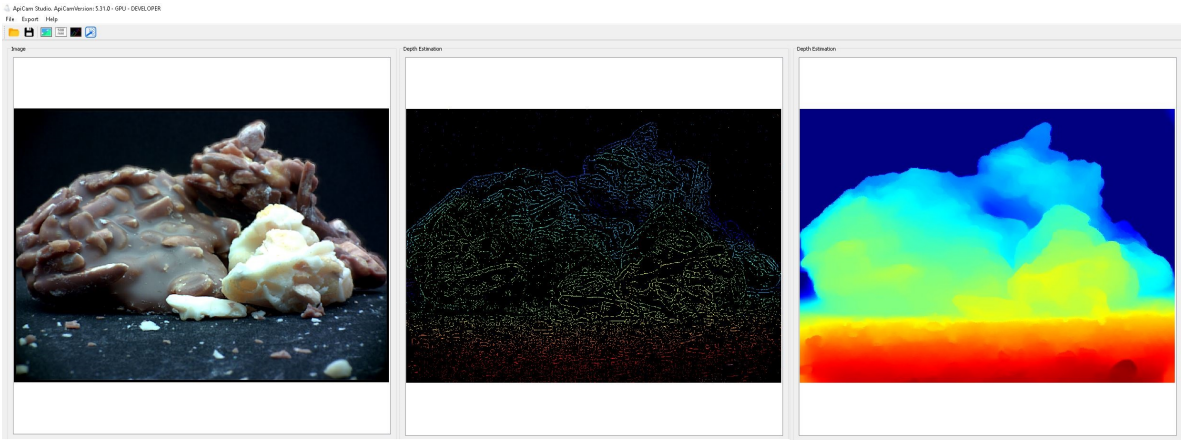


Figura 5.1: Ejemplo de mapa de profundidad disperso (imagen central) y cómo quedaría denso (derecha) después de aplicar los algoritmos de rellenado expuestos en este capítulo. Los objetos más cercanos a la cámara, y por tanto con un valor de profundidad menor, se muestran en color rojo, mientras que los objetos más lejanos a la cámara, y por consiguiente con un valor de profundidad mayor, se muestran en color azul. Las profundidades intermedias se representan con un degradado entre estos dos colores pasando por el naranja, amarillo, verde, etc.

tipo de plataforma empleado consigue minimizar el coste computacional y de memoria del algoritmo.

Es por ello que, de entre todas las estrategias disponibles en la literatura, se ha optado por desarrollar e implementar el algoritmo del KNN basado en *kd-tree* [70]. El algoritmo del *kd-tree* se basa en la estrategia de programación de *ramificación y poda*, permitiéndonos reducir el número de distancias a calcular entre los nodos consulta y los nodos con valor de profundidad, pues es capaz de predecir cuándo una rama del árbol ya no es prometedora para obtener un vecino más cercano que los actuales y, de esta forma, dejar de explorar dicha ramificación. Además, a la hora de mantener los mejores vecinos encontrados hasta el momento, se ha empleado la estructura de programación basada en un *max-heap*, manteniendo siempre en la primera posición del array de vecinos el peor nodo seleccionado hasta el momento (con un valor de distancia mayor al nodo objetivo), de forma que pueda guiar al algoritmo de ramificación y poda en su búsqueda de forma eficiente. Todas estas decisiones en el diseño del algoritmo nos han permitido desarrollar un algoritmo ligero y capaz de proporcionar un *frame rate* aceptable en *streaming* de vídeo, corriendo los algoritmos en las plataformas móviles Snapdragon®.

Por otra parte, y dado que nos encontramos ante un problema de regresión, pues los valores de los píxeles a calcular no tienen porque ser idénticos al de sus vecinos, sino que puede ser un valor próximo. Además, se ha "pesado" la aportación que realiza cada vecino encontrado al valor de profundidad del nuevo nodo. Este "peso" viene calculado de forma inversamente proporcional a la distancia a la que se encuentra cada vecino del nodo objetivo, de forma que cuanto más "cerca" esté el vecino del nodo objetivo, mayor aportación hará su valor de profundidad en el cálculo del valor final que se persigue, y de esta forma se trata de no suavizar los bordes de los objetos que componen la escena.

Para el cálculo de las distancias entre píxeles, se ha tenido en cuenta tanto la componente espacial

($x$ e $y$) del píxel como su componente de color, siendo esta la luminancia en caso de cámara monocromáticas y el RGB en caso de hacer uso de cámaras a color. Estas componentes de distancia, espacial y de color, han sido también pesadas por separado con la finalidad de poder así atribuirle un mayor o menor peso a cada una de las componentes en aras de obtener un mejor resultado final. Esta técnica del pesado nos ha ayudado a obtener un mapa de profundidad mucho más realista, proporcionándonos un mapa denso que representa de la forma más fiel posible la imagen de entrada.

La elección del *kd-tree* como estrategia de búsqueda para los $k$ vecinos más cercanos también nos permite la flexibilidad de no tener que calcular el valor de profundidad para la totalidad de los píxeles de la escena, sino que se puede realizar esta tarea bajo "demanda", es decir, únicamente se calculan los vecinos (y por consiguiente el valor de profundidad) para aquellos píxeles que el usuario o la aplicación requieren, ahorrando de este modo una gran cantidad de cómputo (y por consiguiente energía) en los casos que no sea necesario la obtención de un mapa denso completo de la escena, sino únicamente de algunas zonas o píxeles de la misma.

Para finalizar y con el objetivo de reducir el tiempo de cómputo necesario para la obtención del mapa denso, se ha hecho uso de las técnicas de imagen por computador *downsampling* y *upsampling*, permitiendo de este modo reducir el número de píxeles involucrados en el algoritmo del KNN y que, por tanto, reduce de forma importante sus requerimientos computacionales. El uso de este tipo de técnicas de modificación del tamaño del problema no es "gratis", pues introduce pérdidas en los detalles de la imagen original que pueden ser apreciadas al comparar la imagen original con el mapa denso de la escena calculada, pero a cambio se obtienen unos tiempos de ejecución mucho más competitivos.

Resumiendo, y más allá de las publicaciones en forma de patente, en este capítulo se describe el aporte del doctorando a la solución final implantada, una solución caracterizada por:

- Implantar algoritmos en un producto comercial que funciona sobre una plataforma de bajo consumo disponible en el mercado,
- Varias decenas de kits de evaluación/desarrollo siendo evaluados por grandes marcas (p.e. Apple, Google, HTC, HP, etc.),
- Aportar unas características sin competencia en el mercado[1], y
- La colaboración con la empresa líder en España de reconocimiento facial facePHI®, para la incorporación de nuestra tecnología a sus algoritmos de reconocimiento facial con la finalidad de proporcionarles robustez *anti-spoofing* sin la necesidad de interactuar con el usuario.

Es importante recalcar que, partiendo de la cámara descrita en el Capítulo 4, tomando un único disparo de cámara y sin necesidad de elementos externos, la solución presentada es capaz de reconstruir un mapa de profundidad de la escena como el que se muestra en la Figura 5.1, mientras que las técnicas utilizadas por la competencia precisan de más disparos de cámara y/o de elementos externos activos que les ayudan a estimar la distancia a la cual se encuentran los objetos.

[1] Afirmación válida en la actualidad sobre todo en distancias cercanas a la cámara, dispositivos sin otros elementos *hardware* de apoyo a la estimación (p.e. luz estructurada) y de bajo coste, esto es, un coste asumible para su incorporación, por ejemplo, al mercado de la telefonía móvil.

Si bien el mapa disperso (*sparse map*) puede ser adecuado para algunas aplicaciones, la mayoría de aplicaciones precisan de poder conocer la profundidad a la que se encuentran la totalidad o algunas zonas concretas de la imagen, y no únicamente las zonas en las que tengamos un borde. Es por ello, que se ha precisado del diseño e implementación de un algoritmo capaz de proporcionar profundidad a los píxeles de la escena de los cuales no la tengamos.

### 5.1.1 Caso práctico de aplicación de mapa denso, reconstrucción 3D

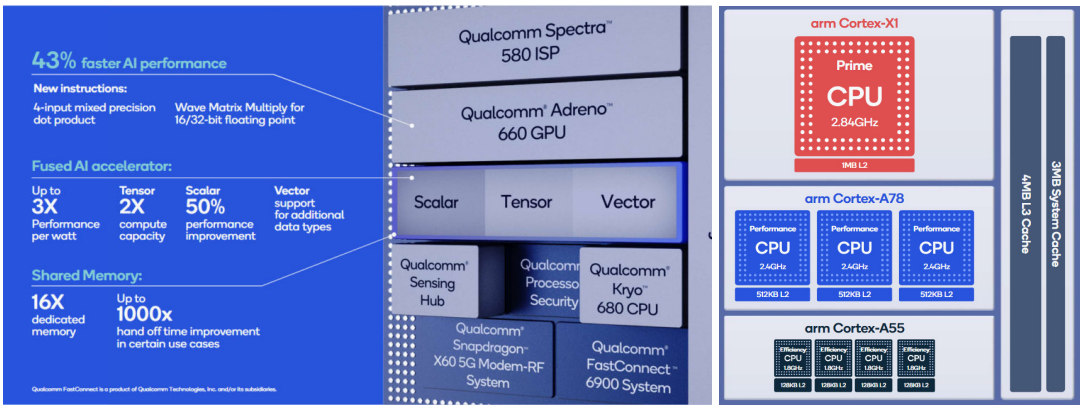
Un ejemplo ilustrativo y claro de la necesidad del mapa denso de profundidad, es el de la colaboración con la empresa facePHI®, empresa dedicada al reconocimiento facial y que actualmente basa sus algoritmos de reconocimiento facial en la imagen RGB de una cámara convencional.

Los algoritmos de reconocimiento basados en la imagen 2D presentan una problemática principal, y es que a veces no son capaces de diferenciar una foto de una cara real, pues no disponen de información de profundidad. Actualmente, esto se suele solucionar con una interacción con el usuario, es decir, le piden al usuario que guiñe un ojo, abra la boca o alguna acción similar aleatoria que les permita constatar que es una persona la que se encuentra validando su identidad, y no es un vídeo reproducido en una tablet o una fotografía. Este tipo de solución, si bien funciona, no es amigable para el usuario (*user friendly*), por lo que a la empresa de reconocimiento le interesa incorporar una fuente de información extra que por un lado hace más amigable la aplicación de su reconocimiento y por otro añade una característica más a su algoritmo que aporta robustez a la identificación. La tecnología necesaria para todo ello la proporciona la empresa photonicSENS®.

La solución propuesta parte de unos puntos claves de la cara, como pueden ser nariz, ojos o boca, entre otros, y la tecnología de photonicSENS® debe ser capaz de proporcionar la información de profundidad a la que se encuentran dichos píxeles (o grupo de píxeles) con la finalidad de verificar que son consistentes con los de una cara real (no una foto) y poder validarlo sin necesidad de interactuar con el usuario. Para ello, ambas empresas se apoyan en una red neuronal de detección de puntos claves de la cara y un mapa de profundidad denso "alrededor" de dichos puntos, para validar que éstos configuran un volumen que encaja con una nariz, una boca, un ojo, etc. Para ello, es imprescindible tener la profundidad de dicho puntos clave, mediante el mapa denso completo o pidiendo puntos bajo demanda. El algoritmo que resuelve el mapa denso o la petición de profundidad de puntos concretos se va a explicar en el actual capítulo, con un par de ligeras variaciones que enumeramos a continuación:

- Se detecta el *bounding box* de la cara en la imagen y únicamente computamos como píxeles con valor de profundidad válidos los incluidos dentro de éste.
- Después de detectar los puntos claves de la cara, nariz, boca, ojos, etc., se crean *sub-boxes* más pequeñas alrededor de ellos, generándose el mapa denso de profundidad solo en estas zonas y poder así determinar que se trata de una cara real.

Otro ejemplo claro de la necesidad de tener un mapa denso de una escena, es poder realizar una reconstrucción 3D de la misma, por ejemplo para tareas de impresión 3D o diseño y amueblado de estancias en casas u oficinas.




(a) SoC Qualcomm 888 (b) CPU Qualcomm Kryo-680

Figura 5.2: Arquitectura SoC 888 de la compañía Americana Qualcomm®.

### 5.1.2 Hardware: Qualcomm Snapdragon

La multinacional americana Qualcomm® es la compañía líder en sistema sobre chip (SoC) para dispositivos móviles. Tiene una amplia gama de ellos, siendo capaces de adaptarse a múltiples necesidades. Los SoC de la compañía se comercializan bajo el nombre comercial Snapdragon®, los cuales se basan en las arquitecturas tipo RISC desarrolladas por la compañía ARM®.

Estos SoC de Qualcomm® disponen de diferentes unidades de procesamiento en un mismo chip, entre las que se encuentran una CPU multi-núcleo, una GPU integrada con la denominación Adreno®, así como uno o varios *Digital Signal Procesing* (DSP) o Procesador de Señales Digitales, incluyendo algunos programables por el usuario, conocidos con la nomenclatura Hexagon®.

Los SoC de la compañía Qualcomm® son muy eficientes, obteniendo un magnífico ratio *GFlop/watt*, convirtiéndolos en la compañía líder del sector de la telefonía móvil, hecho que nos ha hecho decantar por desarrollar nuestros algoritmos de rellenado (*filling*) optimizándolos para el *hardware* de esta compañía en busca de un gran nicho de mercado en el que poder vender las cámaras de profundidad que estamos desarrollando en la empresa photonicSENS®.

Para alcanzar un alto grado de optimización de los algoritmos al *hardware* donde se van a ejecutar, es necesario el conocimiento de los detalles de implementación de dicho *hardware* y así poder explotar al máximo su capacidad computacional. En una empresa líder como Qualcomm® esto no es sencillo, pues únicamente dan soporte y detalles de implementación del *hardware* a aquellas empresas de su ecosistema[2] y siempre bajo la firma de contratos de confidencialidad millonarios que les permita salvaguardar la tecnología que les proporciona ventajas competitivas respecto a la competencia.

En el momento de redactar esta parte de la tesis, la empresa photonicSENS® tiene firmado un

[2] Alrededor de Qualcomm® existen numerosas empresas que actúan, bien como *partners* bien como empresas que Qualcomm® considera estratégicas (interesantes por volumen de compras y/o por potencialidad de clientes futuros que puedan atraer consigo).

acuerdo con la compañía Qualcomm® para disponer de acceso a la documentación clasificada y al soporte de los ingenieros de la multinacional. Sin embargo, esto no ha sido así durante el desarrollo de los algoritmos de rellenado del mapa denso presentados en esta tesis, por lo que se han tenido que desarrollar diferentes implementaciones y analizarlas de forma empírica para poder sacar conclusiones acerca del funcionamiento del *hardware* y, de esta forma, poder obtener un algoritmo eficiente para dicha plataforma. Gracias al acuerdo de colaboración firmado, también se ha tenido acceso en exclusiva, y con anterioridad a la salida al mercado del último procesador y buque insignia de la compañía para el mercado de telefonía móvil en la anualidad 2021, al Snapdragon 888 (ver Figura 5.2), hecho que nos está permitiendo poder realizar la evaluación de nuestros algoritmos en dicha plataforma y optimizarlos para la misma con la inestimable colaboración de sus ingenieros.

## 5.2 Algoritmos de aprendizaje automático

La búsqueda de coincidencias, la comparación de muestras o el reconocimiento de patrones son problemáticas habituales en un gran número de algoritmos informáticos. Tradicionalmente se llevaban a cabo búsquedas de coincidencias exactas de la instancia de entrada en la estructura de datos almacenada como, por ejemplo, la búsqueda de una cadena de texto en un documento o la búsqueda de un determinado valor de un campo en una base de datos. Este tipo de búsquedas exactas no abarcaban la nueva problemática que se presentaba pues, por ejemplo, surgió la necesidad de saber si dos imágenes pertenecían a la misma persona, cuya búsqueda no debía ser exacta pues podían ser dos imágenes totalmente distintas de la misma persona, es decir, con iluminación distinta y cambios de color, oclusiones, etc. Es por ello, que surge la necesidad de plantear las búsquedas por *similitud*, es decir, cuán parecida es una instancia comparada con otra, sin tener la obligación de que ambas sean exactamente la misma instancia. Para ello, se deben definir métricas que permitan medir el valor de *similitud* entre dos instancias para así poder clasificar las instancias por dicho valor.

Dada esta nueva necesidad de aprender de los datos de entrada, apareció el aprendizaje automático. Pues ya no únicamente era necesario ejecutar acciones en base a unas reglas predefinidas para los datos de entrada, sino que se debía ir "aprendiendo de la experiencia" de los datos de entrada que se iban procesando para, de esta forma, poder ejecutar la acción idónea en cada momento.

Existen dos tipos de algoritmos de aprendizaje automático, los algoritmos de aprendizaje automático *supervisados* y los *no supervisados*. Los algoritmos supervisados son aquellos en los que se posee un conjunto de datos de entrenamiento de los cuales se conoce la etiqueta (valor de salida esperado) dada una entrada, es decir, relacionan valores de entrada con los valores de salida esperados. En cambio, en los no supervisados no estamos interesados en relacionar entradas con salidas, sino en agrupar los datos de entrada según una extracción de características relevantes, de forma que queden agrupadas las diferentes entradas en base a dichas características similares, en forma de grupos o *clusters*. Los algoritmos no supervisados se basan en "aprender" estructuras básicas de los datos de entrada para tratar de agruparlos correctamente en base a la similitud de su estructura.

Como se ha mencionado, un algoritmo de aprendizaje automático supervisado está basado en la

existencia de un conjunto de muestras etiquetadas, de forma que pueda aprender una función que produce un resultado esperado cuando se le proporcionan unos nuevos datos sin etiquetar. Por el tipo de aplicación que se desarrolla en este trabajo, con puntos espaciales de los que conocemos su valor junto con otros puntos espaciales pendientes de etiquetar, nos vamos a centrar más en este tipo de algoritmos, los supervisados.

Como analogía de fácil comprensión para poder entender cómo funcionaría un algoritmo de aprendizaje automático supervisado, imagínese por un momento que la aplicación es un niño y nosotros queremos que el niño aprenda a diferenciar un perro de otro tipo de animales (gatos, vacas, cerdos, etc.). Para ello, preparamos un número de muestras significativo de diferentes imágenes de animales, entre las que se encuentren diferentes imágenes que muestren un perro. Cuando vemos una imagen que muestra un perro, nosotros le indicamos al niño que esa imagen se corresponde con la de un perro, en cambio, cuando la imagen mostrada no corresponde a un perro, se lo hacemos saber al niño de igual modo. Una vez ya hemos repetido este procedimiento con un número significativo de imágenes, el niño ya es capaz de diferenciar cuándo una imagen corresponde a un perro y cuándo no. Si ahora al niño se le muestran imágenes nuevas, que nunca ha visto, tanto con perros como sin ellos, debería ser capaz de clasificarlas correctamente. Esto es aprendizaje automático supervisado.

Existen dos tipos de problemas base para los algoritmos de aprendizaje automático, los problemas basados en clasificación (se emplean valores discretos) y los problemas basados en regresión (los cuales emplean valores reales).

Un ejemplo de problema de clasificación sería el descrito anteriormente del niño y las imágenes de perros, pues como salida, el sistema debe devolver que una imagen corresponde a un perro o no corresponde a un perro, donde en una aplicación podríamos asignar el valor 1 para cuando se trata de una imagen de un perro y un 0 cuando no lo es.

En cambio, en un problema de regresión se utilizan valores reales. Un símil representativo podría ser el de estimar el peso óptimo de una persona en función de parámetros biométricos (p.e. altura). De esta forma, el algoritmo de regresión aprendería un modelo de estimación del peso basado en dichos parámetros, y estimaría las nuevas entradas en base al modelo aprendido, siendo el peso un valor real y no una clase discreta.

## 5.3 Algoritmo $k$ vecinos más cercanos

En esta sección vamos a presentar un algoritmo de aprendizaje automático conocido como KNN $k$ *nearest neighbors* (o en español $k$ vecinos más cercanos), el cual es uno de los algoritmos de clasificación básicos y esenciales en las aplicaciones de aprendizaje automático (o del inglés *Machine Learning*). Este algoritmo es un método clasificatorio que se basa en otras instancias "similares" a la hora de clasificar nuevas instancias sin la necesidad de una coincidencia exacta con las instancias ya clasificadas, sino que utiliza la distancia entre instancias como una medida de similaridad. El KNN parte de la suposición base de que si dos instancias se encuentran "cercanas", es porque ambas son muy similares y, por tanto, pertenecerán a la misma clase o poseerán un valor real muy similar. Es decir, este algoritmo se

fundamenta en la idea básica de que un nuevo caso a clasificar va a pertenecer a la clase más común (más votada) entre sus $k$ vecinos más cercanos, o bien, va a poseer un valor real próximo al de sus vecinos.

La popularidad de este algoritmo radica en la simplicidad e intuición de la idea que hay detrás y que, además, puede ser implementado de forma sencilla, tanto para la clasificación de nuevas instancias (valores discretos) como para la predicción de valores continuos (regresión).

En el algoritmo del KNN, la variable $k$ representa la cantidad de instancias vecinas en las que se va a basar el algoritmo para predecir la clase o valor de la nueva instancia. El algoritmo del KNN posee las siguientes características:

- **Supervisado:** partimos de etiquetas o valores de entrenamiento.
- **No paramétrico:** no hace suposiciones explícitas sobre la forma funcional (estructura básica) de los datos, evitando modelar mal la distribución subyacente de los datos.
- **Basado en instancia:** nuestro algoritmo no aprende explícitamente un modelo. En lugar de ello, opta por memorizar las instancias de formación que posteriormente se utilizan como "conocimiento" para la fase de predicción. Concretamente, esto significa que solo cuando se realiza una consulta a nuestro compendio de instancias clasificadas, es decir, cuando le pedimos que asigne una etiqueta o valor a una nueva entrada, el algoritmo utilizará las instancias ya "memorizadas" para dar una respuesta.

Debido a la última característica del algoritmo, esto es, estar basado en instancias, cabe destacar que la fase de formación mínima de KNN se realiza a cambio de un coste de memoria, ya que debemos almacenar un conjunto de datos de tamaño considerable que se requiere para la clasificación de una observación determinada en la solución. En la práctica, esto no es deseable a nivel de almacenamiento y de tiempo de ejecución, por lo que se utilizan diferentes estrategias de programación para optimizar el número de datos a explorar, como pueden ser los árboles de búsqueda.

### 5.3.1 Elección del valor correcto de $k$

El valor adecuado de $k$ se selecciona de forma empírica, eligiendo el valor que reduce la cantidad de errores que encontramos mientras mantenemos la habilidad del algoritmo para poder hacer las predicciones de forma rápida. Este valor por defecto es de $k = 10$, aunque debemos tener en cuenta los siguientes escenarios:

- A medida que el valor de $k$ se aproxima a 1, las predicciones que realizaría el clasificador se vuelven menos estables, pues, se basa para clasificar la instancia nueva únicamente en el valor del vecino más cercano, el cual puede no representar la clase o valor correcto para la nueva instancia. Además, esta opción genera resultados muy discretos (las transiciones entre regiones del mapa de profundidad denso no quedan suaves) debido a que se basa solo en la muestra más cercana y no promedia con otros vecinos (a modo de *winner-take-all*).

- Si hacemos lo contrario, elegimos un valor para $k$ grande, las predicciones de clasificación se vuelven más estables y precisas, pues estas clasificaciones se basan en una "mayoría" de vecinos significativa. No obstante, si este valor de $k$ es excesivamente grande, las predicciones pueden comenzar a fallar ya que forzamos al algoritmo a buscar un número alto de muestras y puede estar incorporando vecinos muy lejanos (que probablemente sean irrelevantes o erróneos) para llegar a ese número $k$. Además, un valor muy alto para la variable $k$ del algoritmo, conlleva un incremento directamente proporcional del tiempo de cómputo, pudiendo llegar a hacerse el proceso significativamente lento.

Por lo tanto, en la aplicación final se ofrece un valor por defecto probado empíricamente que generalmente devuelve buenos resultados, llegando a un compromiso entre las dos opciones extremas ya citadas pero, en cualquier caso, es un parámetro configurable a nivel de usuario.

### 5.3.2 Tipos de distancias

Para el cálculo de distancias entre dos o más instancias en la búsqueda de los vecinos más cercanos y teniendo en cuenta posiciones espaciales dentro de la imagen podemos utilizar las siguientes medidas de distancia:

#### Distancia Euclídea

Una opción muy recurrida en el mundo de la imagen para "cuantificar" la similitud entre instancias es la distancia euclídea. La distancia euclídea o euclidiana, es la distancia "ordinaria" entre dos puntos de un espacio euclídeo, la cual se deduce a partir del teorema de *Pitágoras*.

Por ejemplo, en un espacio tridimensional (3D), la distancia euclídea entre dos puntos $X$ y $Y$, de coordenadas cartesianas $(x_1, x_2, x_3)$ y $(y_1, y_2, y_3)$, respectivamente, es:

$$D_E(X,Y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2},$$

y generalizando para cualquier dimensión ($n \geq 1$ queda):

$$D_E(X,Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

#### Distancia Manhattan

La distancia de Manhattan entre dos puntos es la suma de las longitudes de las proyecciones del segmento de línea entre dos puntos sobre el sistema de ejes coordenados en un espacio vectorial real n-dimensional con un sistema de coordenadas cartesianas fijo.

El ejemplo para un espacio tridimensional 3D, la distancia de Manhattan entre dos puntos $X$ y $Y$,

de coordenadas cartesianas ($x_1$, $x_2$, $x_3$) y ($y_1$, $y_2$, $y_3$), respectivamente, es:

$$D_M(X,Y) = |x_1 - y_1| + |x_2 - y_2| + |x_3 - y_3|,$$

y generalizando para cualquier dimensión ($n \geq 1$) queda:

$$D_M(X,Y) = \sum_{i=1}^{n} |x_i - y_i|.$$

En nuestro caso, dado que la dimensión de los datos no cambia, es mucho más conveniente utilizar la distancia de Manhattan que la distancia euclídea, ya que igual nos va a separar las instancias por distancias (siendo poco importante el significado del valor absoluto de los valores de distancia) y el tiempo de cómputo es menor en la distancia de Manhattan que en la distancia euclídea.

### 5.3.3 Pasos del algoritmo KNN

El pseudocódigo del algoritmo de los $k$ vecinos más cercanos se muestra en el Algoritmo 11.

**Algoritmo 11:** Algoritmo del **KNN**

```
//Instancias sin clasificar (QP)
//Instancias clasificadas (VP)
//Número de vecinos a buscar (k)
//k vecinos más cercanos (knn)
for qp in QP do
    for vp in VP do
        Calcular la distancia d entre qp y vp.
        Añadir la tupla distancia válida–pixel (d, vp) al array de vecinos del qp actual.
    end for
    Ordenar el array de vecinos del qp actual.
    Elegir las k primeras entradas de la colección ordenada anterior.
    Obtener las etiquetas de las k entradas de la colección con menor valor de distancia.
    Calcular el valor de salida para la instancia de entrada:
    if regresión then
        Calcular la media de los valores de sus k vecinos más cercanos. (También podríamos calcular un
        valor de media ponderada por la distancia u otras técnicas de optimización del valor calculado según
        la naturaleza de la problemática.)
    else
        {Si es clasificación.} Devolver la etiqueta con mayor número de apariciones entre sus k vecinos más
        cercanos.
    end if
end for
```

### 5.3.4 Técnicas de implementación para búsqueda rápida

Cuando trabajamos con grandes volúmenes de datos o con aplicaciones que requieren la respuesta en tiempo real, la búsqueda por similitud requiere enfocar la atención en la eficiencia de las búsquedas

con la finalidad de obtener la respuesta en el menor tiempo posible.

El primer algoritmo que nos viene a la cabeza a la hora de implementar la técnica de los vecinos más cercanos es el algoritmo de fuerza bruta, es decir, calcular la distancia de todas y cada una de las instancias de entrada con todas y cada una de las instancias correctamente clasificadas. En esta técnica, trivial, el coste de búsqueda depende linealmente del número de muestras de entrenamiento almacenadas, pues se realiza una búsqueda exhaustiva. Por consiguiente, esta implementación no es apta para grandes colecciones de datos de entrenamiento ni para aplicaciones que tengan requerimientos de tiempo real, siendo el coste del cálculo de la distancia entre muestras no despreciable, aunque sí es una de las técnicas que mejor funciona y que más fácilmente permite exportar el algoritmo al cálculo masivo en GPUs, especialmente cuando contamos con GPUs de supercomputadores y sin restricciones de consumo de energía.

Sin embargo, debido a su popularidad, el algoritmo de los $k$ vecinos más cercanos ha sido causa de estudio desde hace muchos años y, como consecuencia, se ha ido proponiendo un gran número de variantes encaminadas a reducir el coste computacional de las búsquedas [19, 55]. Principalmente existen dos factores que afectan al coste computacional del algoritmo: el tamaño del conjunto de datos y la dimensionalidad de los mismos.

Dado que en nuestra problemática el tamaño de los datos viene predefinido por el número de píxeles con valor de profundidad detectados por el algoritmo y la dimensionalidad de los datos es muy reducida, nos centramos en algoritmos que realizan las búsquedas de forma eficiente en el contexto del número de instancias clasificadas a explorar, en concreto, seleccionamos el *kd-tree* o árbol $k$-dimensional debido a su eficiencia en la búsqueda de los vecinos con un número de dimensiones reducido como es nuestro caso, además de poseer otras ventajas, como la búsqueda de vecinos bajo demanda una vez el árbol de búsqueda ha sido generado.

El coste computacional de la búsqueda de los $k$ vecinos más cercanos para el algoritmo de *fuerza bruta* es lineal con el número de muestras clasificadas $n$ ($O(n)$). En cambio, el coste de la búsqueda de los vecinos haciendo uso de un *kd-tree* es logarítmico con el número de muestras clasificadas $n$ ($O(\log_2(n))$) [70].

### 5.3.5 *Max Heap*

Un *Max Heap* es un árbol binario completo en el que el valor de cada nodo interno es mayor o igual que los valores de los hijos de ese nodo.

*Mapear* los elementos de un *heap* en un vector es trivial: si un nodo es almacenado en la posición $i$ del vector, entonces su hijo izquierdo se almacena en la posición $2i + 1$ y su hijo derecho se almacena en la posición $2i + 2$.

#### Representación de un *Max Heap*

Un *Max Heap* es un árbol binario completo, que generalmente se representa como un vector de datos. El elemento raíz del árbol será ubicado en la posición 0 del vector. Dado el $i$-ésimo nodo, para

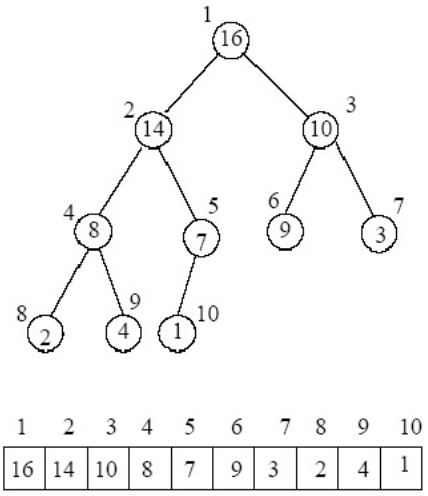



Figura 5.3: Ejemplo de un *Max Heap*.

$i > 0$, los otros nodos relacionados con este serían:

- $(i-1)/2$: Devuelve el nodo padre del nodo $i$-ésimo.
- $2i+1$: Devuelve el nodo hijo izquierdo del nodo $i$-ésimo.
- $2i+2$: Devuelve el nodo hijo derecho del nodo $i$-ésimo.

En la Figura 5.3 podemos ver un ejemplo de un *Max Heap*, tanto en su representación gráfica como árbol binario (arriba), como en su representación en un vector (abajo).

**Operaciones en un *Max Heap***

- `getMax()`: Devuelve el elemento raíz del *Max Heap*. La complejidad temporal de esta operación es $\theta(1)$.
- `extractMax()`: Elimina el elemento máximo del *Max Heap*. La complejidad de tiempo de esta operación es $\theta(\log_2(n))$ ya que esta operación necesita mantener la propiedad del *heap* después de eliminar la raíz.
- `insert()`: Insertar un nuevo nodo en el árbol tiene un coste de $\theta(\log_2(n))$. Agregamos esta nueva clave al final del árbol. Si la nueva clave es más pequeña que su padre, entonces no es necesario que hagamos nada. De lo contrario, necesitamos subir el nodo insertado hasta cumplir la propiedad del *heap*.

Por tanto, utilizando un *Max Heap* en lugar de un array ordenado por completo, pasamos de tener un coste computacional logarítmico con el número de vecinos $k$ en caso promedio $O(\log_2(k))$, en vez $O(k\log_2(k))$ que tendría de coste en caso promedio si tuviésemos un array de vecinos ordenado.

### 5.3.6 *kd-tree*

Los árboles k-dimensionales (*kd-tree*) son una generalización de los árboles binarios de búsqueda (*Binary Search Tree - BST*) que nos permiten trabajar con varias dimensiones de búsqueda. Este tipo de árbol organiza y almacena las instancias de un espacio $k$-dimensional subdividiendo las instancias en cada nivel del árbol en dos subárboles equilibrados. Este algoritmo fue propuesto por *Bentley* en 1975 [8].

La raíz del árbol se determina realizando una división en base a un valor de corte de la primera coordenada, empleando las siguientes coordenadas de forma ordenada en los sucesivos niveles del árbol de búsqueda hasta utilizar todas las coordenadas del problema. Cuando ya han sido utilizadas todas las $k$ dimensiones en el nivel $k$ del árbol, volveremos a utilizar la primera dimensión de nuevo para discriminar en dicho nivel, y así sucesivamente hasta llegar a las hojas del árbol. Por tanto, se puede expresar la coordenada discriminante del nivel $i$ del árbol como: $(i+1)\,mod\,k$.

Durante la fase de búsqueda se recorre el árbol siguiendo un esquema de ramificación y poda para encontrar el vecino más cercano a una muestra dada $x$. El proceso de búsqueda en el *kd-tree* es recursivo. Dada una muestra $x$ y un nodo cualquiera del árbol (que no sea una hoja), se compara la coordenada de $x$, que es discriminante para ese nodo, con el valor de corte que sirvió para construir sus subárboles. Si la instancia del nodo actual del árbol es menor que la primera instancia del *Max Heap*, se inserta este nodo en el *Max Heap* eliminando consiguientemente la instancia cabeza del *heap*; y, a continuación, se comprueba si alguno de los dos hijos, y por tanto sus subárboles correspondientes, son "prometedores", es decir, si pueden contener algún vecino más cercano que el peor vecino encontrado hasta el momento en base al criterio de corte y, si es así, se continúa explorando las ramas prometedoras hasta agotarlas.

Tanto en la fase de construcción del árbol como en la de la búsqueda de los $K$ vecinos, se utilizan las coordenadas de las instancias, por lo que este tipo de árboles se basa en espacios vectoriales. Se hace uso de la recursividad para la búsqueda de los vecinos más cercanos y de la estrategia de ramificación y poda para reducir el número de nodos del árbol a explorar y, por tanto, el cálculo de las distancias entre la instancia de entrada y las posibles instancias vecinas.

Debido a la popularidad y la utilidad de este algoritmo entre la comunidad científica, se han propuesto infinidad de optimizaciones, como pueden ser [9, 24, 59, 60], por citar algunas de las destacadas. Pese al gran número de propuestas de optimización que se han dado, cuando se hace uso de la distancia euclídea en la implementación, el algoritmo original de Bentley sigue siendo la referencia en el estado del arte.

#### Construcción del *kd-tree*

Los *kd-trees* se construyen de forma *top-down*, por lo que antes de proceder a la construcción del *kd-tree* a partir del conjunto de instancias $C$, debemos elegir la coordenada discriminante que mayor varianza presente de entre las posibles, y elegir un hiperplano que divida el conjunto de instancias $C$ en dos subconjuntos. Procederemos del mismo modo de forma recursiva con cada nuevo subconjunto

que obtengamos hasta llegar a las hojas del árbol.

Otro aspecto importante a tener en cuenta a la hora de construir un *kd-tree* es que este se construya de forma balanceada, es decir, que las profundidades de las hojas del árbol sean idénticas en la medida de lo posible. Esto se consigue seleccionando un hiperplano que se sitúe en la mediana de los valores de la coordenada discriminante.

A continuación, describimos los pasos necesarios para la construcción del árbol:

1. Se selecciona como nodo raíz la instancia que representa la mediana de entre el resto de instancias del conjunto $C$ para la coordenada discriminante seleccionada y se almacena como información de dicho nodo del árbol la información de la instancia.

2. Dividimos el conjunto de instancias $C$ en dos subconjuntos, donde los elementos menores o iguales a la mediana seleccionada se ramificarán a partir del nodo que compone el hijo izquierdo del nodo raíz, mientras que los mayores se ubicarán en el nodo derecho.

3. Creamos recursivamente árboles asociados a cada subconjunto hasta que los tamaños de los subconjuntos sean menores o iguales del tamaño prefijado de antemano para los nodos hojas del árbol.

Seguidamente, describimos un ejemplo de la construcción de un *kd-tree* bidimensional $(x, y)$ formado por los puntos de coordenadas cartesianas siguientes: $(8, 1)$, $(7, 2)$, $(2, 3)$, $(5, 4)$, $(9, 6)$ y $(4, 7)$ y apoyamos dicha descripción con la Figura 5.4.

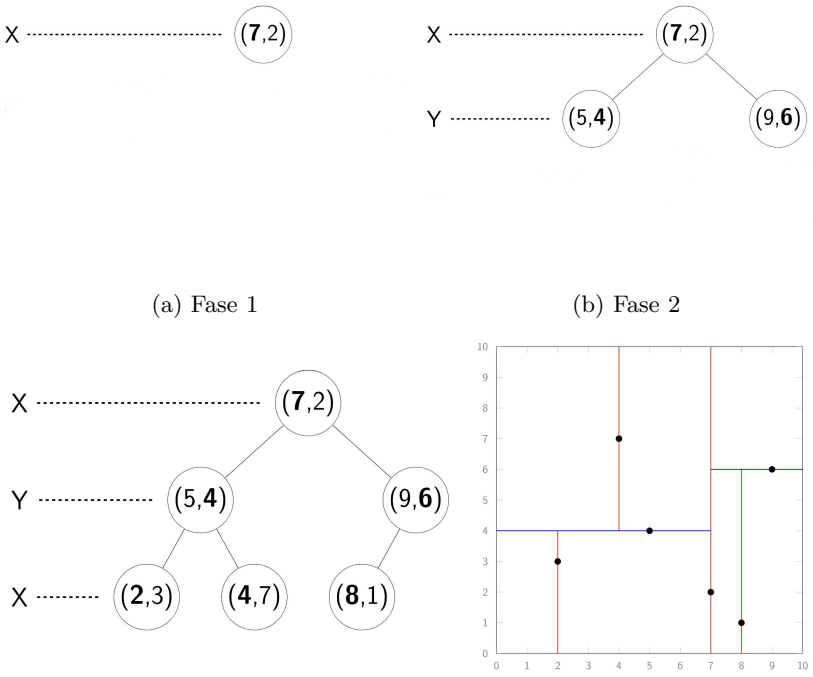
En una primera fase se ordena el vector de puntos anterior por la primera coordenada discriminante, en este caso la coordenada $x$, quedando el vector ordenado de la siguiente forma: $(2, 3)$, $(4, 7)$, $(5, 4)$, $(7, 2)$, $(8, 1)$ y $(9, 6)$. Seleccionamos la mediana de estos datos, que es el punto $(7, 2)$, y lo asignamos como nodo raíz del árbol (como se puede observar en la Figura 5.4 - Fase 1).

En una segunda fase, ordenamos los dos vectores de puntos bidimensionales que nos quedan a la derecha y a la izquierda del nodo raíz seleccionado con anterioridad, quedando en este caso ordenados por la coordenada discriminante $y$. Entonces, quedan los vectores ordenados de la forma siguiente: vector-izquierdo $(2, 3)$, $(5, 4)$ y $(4, 7)$ y vector-derecho $(8, 1)$ y $(9, 6)$. Volvemos a seleccionar las medianas de cada vector y las establecemos como nodos hijos izquierdo y derecho del nodo raíz, respectivamente. En la Figura 5 - Fase 2 podemos observar este paso.

Finalmente, dado que quedan tres vectores unitarios, es decir, que contienen un único punto cada uno, de izquierda a derecha serían: $v_1$: $(2, 3)$, $v_2$: $(4, 7)$ y $v_3$: $(8, 1)$. Los ubicaríamos como hijos izquierdos o derechos de sus nodos padres según corresponda.

En la Figura 5 - Fase 3 podemos observar cómo quedaría el árbol completo a la izquierda y, a la derecha, los puntos que contiene repartidos en el espacio 2D con sus correspondientes líneas divisorias del espacio asignadas por el valor de la mediana. En rojo se observan las líneas divisorias del espacio para la coordenada $x$ y en azul para la coordenada $y$.

El coste computacional de la construcción del árbol de media es lineal más logarítmico con el número de nodos del árbol $n_A$, esto es $\theta(n_A \log_2(n_A))$.




(a) Fase 1

(b) Fase 2

(c) Fase 3

Figura 5.4: Ejemplo de construcción *kd-tree*.

**Búsqueda en el *kd-tree***

Partimos de una instancia $I = (x_1, \ldots, x_d)$, siendo $d$ la dimensión del sistema, y la distancia $d_{nn}(0)$ al peor vecino hasta el momento de los $K$ vecinos que queremos buscar (inicialmente la distancia al peor vecino es $\infty$) y un árbol de búsqueda para buscar el vecino más cercano de la instancia $I$. Procedemos recursivamente de la siguiente forma [27]:

1. Si el nodo actual del árbol de búsqueda no es una hoja:

   *a*) Se compara la instancia $I$ con la instancia contenida en la hoja actual del árbol (es decir, se calcula la distancia de la instancia $I$ con la instancia de la hoja del árbol) y si esta distancia es menor que la distancia al peor de los $k$ vecinos encontrados hasta el momento, se elimina este vecino y se inserta de forma ordenada en el *Max Heap* la instancia de la hoja del árbol.

   *b*) Se compara el valor de la instancia $I$ de la coordenada discriminante actual *coord* con el valor de corte $c$. Si $I_{coord} + d_{nn} \leq c$, se deja de explorar por la rama derecha del nodo actual, pues el hijo derecho del nodo no puede contener un vecino más cercano que los ya encontrados, por lo tanto, se poda dicha rama del árbol. En cambio, si $I_{coord} - d_{nn} \geq c$, no es necesario explorar el hijo izquierdo del nodo y se poda. En caso de que ninguna de ambas

condiciones se cumplan, se exploran ambas ramas del nodo actual en busca de vecinos más cercanos a la instancia consultada.

2. Si el nodo actual del árbol de búsqueda es una hoja, se compara la instancia $I$ con la instancia contenida en la hoja actual del árbol (es decir, se calcula la distancia de la instancia $I$ con la instancia de la hoja del árbol) y si esta distancia es menor que la distancia al peor de los $k$ vecinos encontrados hasta el momento, se elimina este vecino y se inserta de forma ordenada en el *Max Heap* la instancia de la hoja del árbol.

El coste computacional de las búsqueda en el árbol de los $k$ vecinos de media es logarítmico con el número de nodos que forman el árbol $n_A$, esto es $\theta(\log_2(n_A))$.

## 5.4 Análisis de la solución adoptada

Resumiendo la problemática que se nos presenta, a partir de un mapa *sparse*, es decir, un mapa de la imagen con valores de profundidad únicamente en algunos de los píxeles de la escena (concretamente en los bordes debido al algoritmo descrito en la Patente [10]) y la imagen original de la escena, tenemos que reconstruir un mapa de profundidad *denso*, esto es, que posea valores de profundidad para todos y cada uno de los píxeles de la escena. Esta fase de "rellenado" de los píxeles sin valor del mapa de entrada *sparse* tiene más o menos dificultad dependiendo de:

- la cantidad de puntos *sparse* que tengamos de inicio, y
- de cómo están estos distribuidos en el espacio de la imagen.

Como se ha comentado ya, el algoritmo KNN puede ser utilizado tanto para regresión como para clasificación. En la solución adoptada se utiliza el KNN para efectuar una regresión de los valores de los vecinos más cercanos, siendo el promedio de estos valores ponderado por la distancia de cada vecino el valor asignado.

Puesto que la solución implantada contempla imágenes y/o cámara en movimiento, la generación del árbol de búsqueda se realiza para cada uno de los *frames* que nos llegan de la cámara, de forma que por cada *frame* se construye el árbol *k-dimensional* de búsqueda a partir de los valores del mapa disperso recibido, y se destruye una vez se han realizado las pertinentes búsquedas de los $k$ vecinos más cercanos y se ha generado el mapa denso correspondiente a dicho *frame*. Esto es así siempre y cuando la escena se haya movido, pues se ha implementado un detector de movimiento entre *frames* consecutivos capaz de indicarnos los píxeles que se han movido y los que no. Si el número de píxeles que se han movido respecto al *frame* anterior es menor de un *threshold* predefinido, no se recalcula el mapa de profundidad y se utiliza el mismo. En cambio, si el número de píxeles que se han movido es superior al *threshold* anterior pero inferior a un segundo *threshold* predefinido, se recalcula el árbol de búsqueda y únicamente se buscan los $K$ vecinos más cercanos para aquellos píxeles que han variado su valor según el detector de movimiento, y se calculan, en base a ello, sus nuevos valores de profundidad.

Las dimensiones del árbol *k-dimensional* pueden variar entre 3 y 5, utilizando 3 dimensiones cuando trabajamos con imágenes monocromáticas, donde la entrada está formada por las coordenadas espaciales $x$ e $y$ de los píxeles de la imagen y, como tercera dimensión, se contempla de entrada el valor de la luminancia de la imagen a procesar. Sin embargo, cuando trabajamos con cámaras RGB, las coordenadas espaciales son las mismas que en las imágenes monocromáticas pero, en cambio, se tienen 3 coordenadas discriminantes para el color (una por cada componente de color RGB). Finalmente, cabe la posibilidad de utilizar imágenes de color pero transformar las 3 dimensiones a su luminancia equivalente o transformarlas a soluciones mejores a la hora de separar instancias [46], ya que las diferencias en el espacio RGB no tienen por qué ser las más discriminantes entre instancias.

La totalidad de los valores para las coordenadas discriminantes, tanto las espaciales como las de color/intensidad, son normalizados a valores comprendidos en el intervalo $[0, 1]$ con la finalidad de equiparar el peso de las distancias calculadas de cada una de las coordenadas. Además, también se ha procedido a tratar los factores que forman la distancia final en dos grupos que aportan un porcentaje $\alpha$ al peso de dicha distancia (generalmente dicho porcentaje es del 50 %, aunque este parámetro es configurable). Por una parte, se promedian los valores de distancia espaciales y se multiplican por un valor de $\alpha$, y por otro lado, a los valores de distancia del color o luminancia se les aplica un producto con el valor de $(1-\alpha)$. Con ello se consigue equiparar el aporte que realizan las componentes espaciales y las de color a la distancia final entre instancias. A modo de ejemplo, si asumimos 2 puntos, $P1$ de coordenadas $(x_1, y_1)$ con nivel de gris $I_{x_1y_1}$ y $P2$ de coordenadas $(x_2, y_2)$ y nivel de gris $I_{x_2y_2}$, la distancia calculada entre ambos puntos sería:

$$D(P1, P2) = \alpha \frac{\frac{abs(x_1-x_2)}{nCols} + \frac{abs(y_1-y_2)}{nRows}}{2} + (1-\alpha)\frac{abs(I_{x_1y_1}) - abs(I_{x_2y_2})}{nGreyLevels}, \tag{5.1}$$

siendo $nCols/nRows$ el número de columnas/filas de la imagen respectivamente y $nGreyLevels$ el número de niveles de gris de la imagen. La extensión a color sigue el mismo formato pero con tres canales de color en lugar de tener únicamente la luminancia y, obviamente, promediando los canales de color.

En la Figura 5.5 se muestra el diagrama de flujo de la implementación realizada basada en el *kd-tree* y que, junto a los siguientes párrafos donde se dan más detalles, va a servir para explicar la implementación adoptada.

Las instancias que representan a los píxeles con valor de profundidad son almacenadas en un vector o array de valores de píxeles válidos para que se almacenen de forma consecutiva en memoria, pues vamos a utilizarlos en la construcción del árbol y posteriormente en la búsqueda de sus vecinos. Para la construcción del árbol de búsqueda, lo primero que tenemos que hacer es generar el nodo raíz, el cual se obtiene después de ordenar el array por la primera coordenada discriminante y seleccionando como nodo raíz la instancia de la mediana. Para el siguiente nivel del árbol, es decir, para los hijos izquierdo y derecho, se procede a ordenar los dos sub-vectores a la izquierda (hijo izquierdo) y a la derecha (hijo derecho) del nodo padre (la mediana seleccionada en el nivel anterior), ordenando ambos sub-vectores de forma independiente por la segunda coordenada discriminante y volviendo a seleccionar como hijo

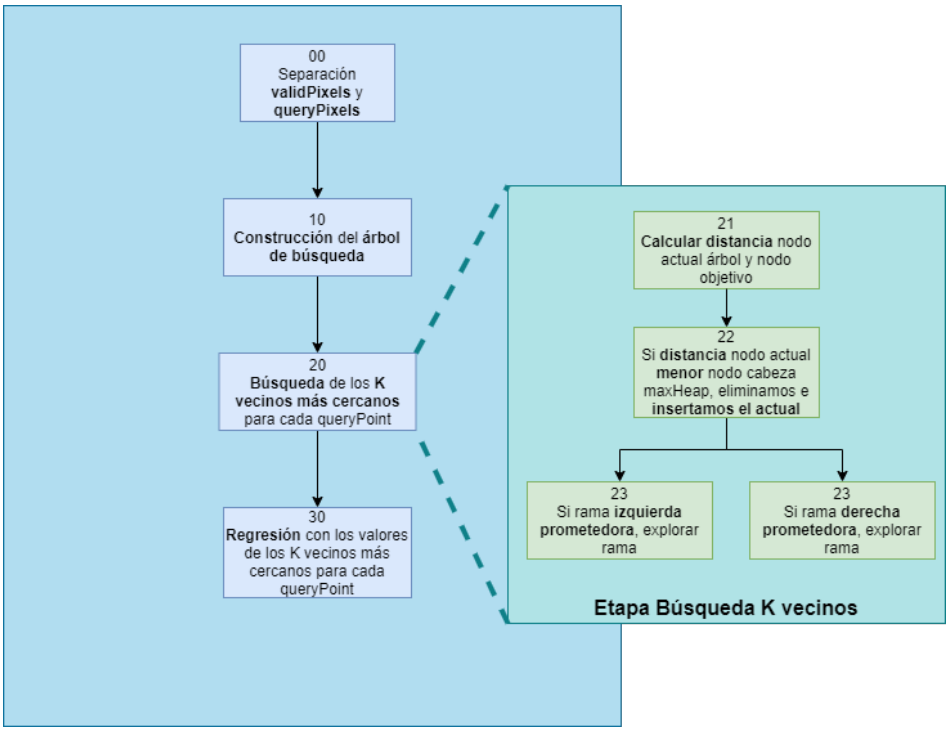



Figura 5.5: Diagrama de flujo algooritmo del KNN implementado.

izquierdo e hijo derecho las medianas obtenidas de ordenar los dos sub-vectores izquierdo y derecho respectivamente. Para el resto de niveles del árbol se procedería de forma idéntica a la descrita.

La elección de la mediana para la división de las muestras no es casual, pues se ha optado por la selección de la mediana persiguiendo generar un árbol de búsqueda balanceado, y así poder obtener unas podas más "potentes" y reducir el tiempo de búsqueda de los vecinos. Además, la generación del árbol se ha realizado mediante punteros al vector de píxeles válidos generado, de forma que no tengamos que replicar datos y producir así un ahorro de memoria.

En cuanto a la búsqueda de los vecinos, y siendo esta totalmente independiente para cada píxel de la imagen sin valor de profundidad, se ha paralelizado el algoritmo de búsqueda haciendo uso de la tecnología *OpenMP* para que se lancen tantos hilos de búsqueda como núcleos computacionales físicos tiene el dispositivo hardware empleado.

Para realizar una mejor estimación del valor de profundidad para cada píxel a partir del valor de profundidad de sus vecinos más cercanos, se ha ponderado el aporte de cada uno de estos por la distancia existente al píxel o instancia que estamos calculando ($W$). El aporte del $i$-ésimo vecino ($W_i$) al cálculo del valor de profundidad para el píxel objetivo es inversamente proporcional a la exponencial del producto de un factor ($F$) (a "configurar") con la distancia a la que se encuentra este vecino ($D_i$,

calculado a partir de la Ecuación 5.1), a saber,

$$W_i = \frac{1}{e^{F \cdot D_i}}.$$

Esta fórmula permite que el aporte de los vecinos más próximos al píxel objetivo sea muy superior que al de los píxeles más lejanos. El factor de distancia a configurar ($F$) que se corresponde con un valor real, nos permite "jugar" con la diferencia de aporte al valor final, entre los vecinos más lejanos y los más cercanos, pues multiplica al valor de distancia obtenido por el vecino.

Finalmente, los pesos se normalizan para que la suma de ellos sea igual a 1:

$$W_i = \frac{W_i}{\sum_i W_i}.$$

Por lo tanto, en función de los $k$ vecinos $V_{x,y}$ encontrados, el nuevo valor de profundidad $d_{x,y}$ calculado sobre una posición vacía $(x, y)$ del mapa disperso será como sigue:

$$d_{x,y} = W_1 \cdot V_{x_1,y_1} + W_2 \cdot V_{x_2,y_2} + \ldots + W_K \cdot V_{x_K,y_K}.$$

Las gráficas de la Figura 5.6 muestran como varían los pesos $W$ en función del factor $F$ para cada distancia $D$.

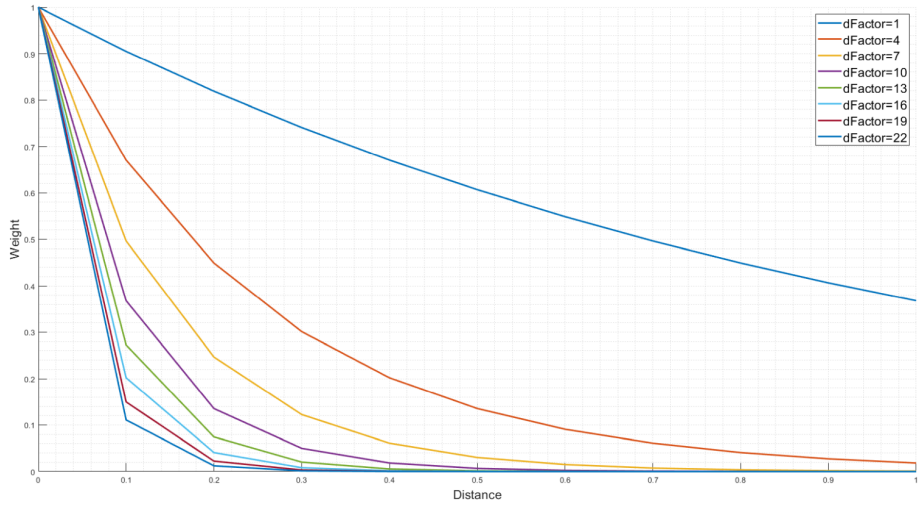



Figura 5.6: Ejemplo de pesos (*weight*) que reciben los vecinos más cercanos en función a su distancia con respecto a la muestra que se evalúa.

## 5.5 Pre-procesos

Con la finalidad de obtener un mapa denso más realista mejorando el resultado que nos proporcionará el algoritmo del *kd-tree* en el resultado visual del mapa de profundidad, se realiza un proceso de eliminación de puntos aislados y de valores anormales del mapa disperso de entrada.

#### 5.5.0.1 Filtro de outliers

Un filtro de outliers[3] puede ser calculado con los valores de estudio provenientes de una ventana de la imagen o de la totalidad de la misma, y se basa en calcular la media o mediana de dichos valores y en base a dicho valor calculado, eliminar los valores que se alejan en demasía de la población de estudio.

En nuestro caso se ha utilizado la media de los valores y se ha implementado el filtro por ventanas "locales" de la imagen, estudiando diferentes valores de ventana como son el de $3 \times 3$, $5 \times 5$, $7 \times 7$, etc. y, eliminando aquellos valores que exceden en un *threshold*[4] predefinido la media de los valores de la ventana correspondiente.

Además de este proceso, también se eliminan los píxeles con valores de profundidad que se encuentran "aislados" dentro de la ventana de procesamiento del filtro, es decir, si dentro de la ventana actual únicamente se encuentra un píxel con valor de profundidad, este se considera como un punto aislado y es eliminado, pues muy probablemente sea un valor de profundidad no válido.

## 5.6 Post-procesos

Con la finalidad de mostrar un resultado más realista y corregir aquellas pequeñas imperfecciones que nos haya podido ocasionar el algoritmo del *kd-tree* en el resultado visual del mapa de profundidad, se realizan diferentes procesos de refinamiento y suavizado del mapa "denso" obtenido.

En este aparatado se detallan los procesos opcionales que se pueden aplicar a la salida de los procesos anteriormente descritos, esto es, opcionalmente para el usuario el mapa denso resultado de la aplicación del algoritmo de los $k$ vecinos más cercanos se puede refinar mediante los post-procesos que se describen a continuación.

### 5.6.1 Filtro de mediana

El filtro de mediana no es sensible a valores extremos presentes en la ventana de filtrado, es por ello que lo empleamos para eliminar este tipo de valores extremos que no han sido eliminados por el filtro de *outliers*. Además, este filtro no se "inventa" ningún valor, pues el valor seleccionado como mediana es un valor que existe en la imagen original. Este tipo de filtros es interesante para respetar los bordes

[3] Un *outlier* o valor "atípico" es un valor que se aleja en demasía de la media o mediana de los valores de estudio o población.

[4] *Threshold*, umbral o límite que si es excedido, se considera el valor como *outlier*.

de los diferentes objetos de la imagen. Reseñar, que este es el filtro aplicado por defecto y que si se aplica este, no se aplica el siguiente filtro de media.

### 5.6.2 Filtro de media

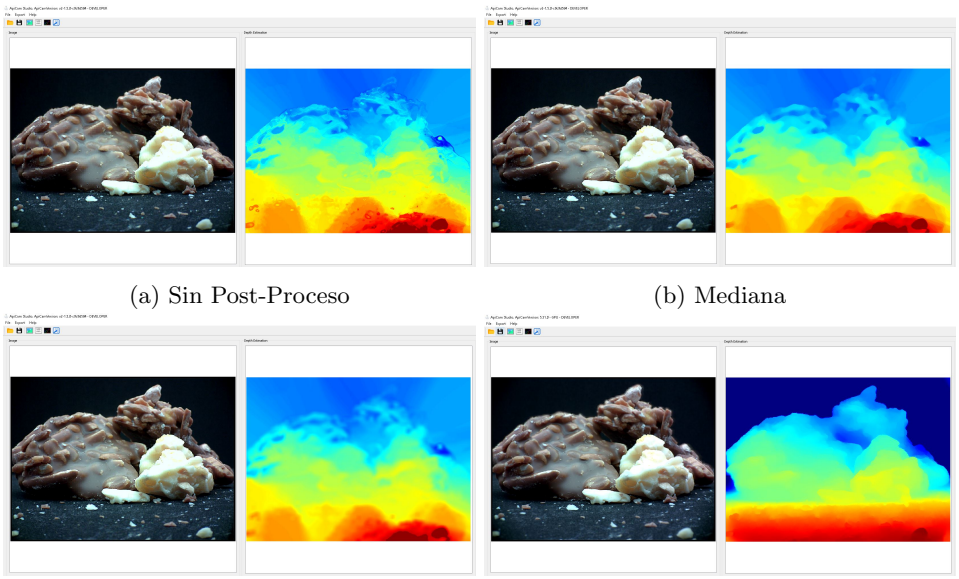
El filtro de media es utilizado para "suavizar" los valores de los diferentes píxeles de la imagen utilizando los valores de los píxeles vecinos. Este tipo de filtro es interesante para el "suavizado" de imágenes, pero por contra las puede emborronar y difuminar los bordes de los diferentes objetos presentes en la misma. Como se ha explicado, si se aplica el filtro de mediana no se aplica el de media y viceversa.

### 5.6.3 *Fast Bilateral Solver*

El algoritmo de *Fast Bilateral Solver* [7] es un algoritmo relativamente novedoso pues fue desarrollado y publicado por los investigadores *J. T. Barron* y *B. Poole* en el año 2016. Es un algoritmo de "suavizado" avanzado que, a diferencia del filtro de media, no emborrona la imagen y permite respetar los bordes de la escena. Es un algoritmo con un coste computacional relativamente "bajo" e igual de flexible que los filtros simples, como el de media o mediana, pero con la precisión de los algoritmos de filtrado más avanzados del dominio. Esta técnica permite competir con los algoritmos de vanguardia en cuanto a calidad de resultados en campos de investigación como el de los mapas de profundidad de una escena, asumiendo como beneficio un uso de recursos computacionales reducido. El principal problema que tiene este algoritmo al "suavizar" la imagen de profundidad es que provoca que las distancias entre los diferentes objetos de la imagen se "acorten" y no se represente fielmente la realidad, aunque visualmente es mucho más ameno de observar y "corrige" o "suaviza" errores de los algoritmos de cálculo de profundidad de los píxeles.

En la Figura 5.7 podemos observar los diferentes mapas de profundidad obtenidos para la misma imagen de ejemplo, aplicando los diferentes algoritmos de refinamiento explicados en esta sección, partiendo del resultado original proporcionado por el algoritmo del *KNN* explicado en la sección anterior. En la imagen *(a)* podemos observar el mapa de profundidad obtenido después de aplicar el algoritmo de los $k$ vecinos más cercanos pesados por la distancia que hemos explicado anteriormente. En las imágenes *(b)* y *(c)* mostramos el resultado de aplicar el filtro de mediana y el filtro de media, respectivamente, con un tamaño de ventana de $3 \times 3$ a la imagen resultado del algoritmo del *KNN*. Finalmente, en la imagen *(d)*, podemos observar el mapa de profundidad obtenido después de aplicar el algoritmo de filtrado *Fast Bilateral Solver* descrito anteriormente.

Como podemos concluir claramente después de los resultados mostrados, para poder obtener un mapa de profundidad de la escena con una "calidad" adecuada para aplicaciones comerciales reales de la actualidad, debemos hacer uso del algoritmo del *Fast Bilateral Solver*, el cual consigue obtener un mapa de profundidad y, por ende, una reconstrucción 3D de la escena muy realista y próxima al de otras tecnologías actuales de la competencia, como puede ser la luz estructurada empleada por *Apple* en sus diferentes dispositivos.



(a) Sin Post-Proceso

(b) Mediana

(c) Media

(d) Bilateral Solver

Figura 5.7: En esta figura se muestran los resultados obtenidos sin aplicar y aplicando los diferentes filtros de post-procesamiento del mapa denso de profundidad.

# Capítulo 6

# Mapa de profundidad y producto final

Veamos ahora la tecnología desarrollada para adquirir las imágenes en las cuales se basan los algoritmos expuestos en esta tesis para la obtención del mapa de profundidad. Además, veremos diferentes aplicaciones así como un estudio comparativo con el estado del arte en la materia.

## 6.1 Introducción

En este capítulo se presentan los resultados de la técnica de rellenado de mapas dispersos descrita en el Capítulo 5, la cual aplica los algoritmos patentados en [10] sobre el hardware Snapdragon® de la compañía Qualcomm®. El resultado final es un producto comercial que es capaz de proporcionar imagen y mapa de profundidad en un solo disparo, así como hacer reenfocado digital de la imagen *a posteriori*, con una cámara para telefonía móvil (u otros usos) desarrollada por la empresa photonicSENS®.

Cabe destacar que el mapa denso de profundidad de la escena descrito en esta tesis se ha presentado a potenciales clientes para la adquisición de la cámara desarrollada por photonicSENS®. Sus resultados se han comparado con las soluciones competidoras, mostrándose en este capítulo cómo dichos resultados son muy favorables para la empresa photonicSENS®.

## 6.2 Presentación del producto

En la Figura 6.1 se muestran dos configuraciones de producto final de la cámara desarrollada por la empresa photonicSENS®. La imagen de más a la izquierda (apiCUBE®) es una solución *plug & play* apta para cualquier tipo de producto final capaz de poder comunicarse a través de las diferentes interfaces de comunicación USB. Este tipo de producto nos hace abarcar un gran abanico de posibilidades en el mercado, pues la interfaz USB es ampliamente adoptada por un sin fin de productos

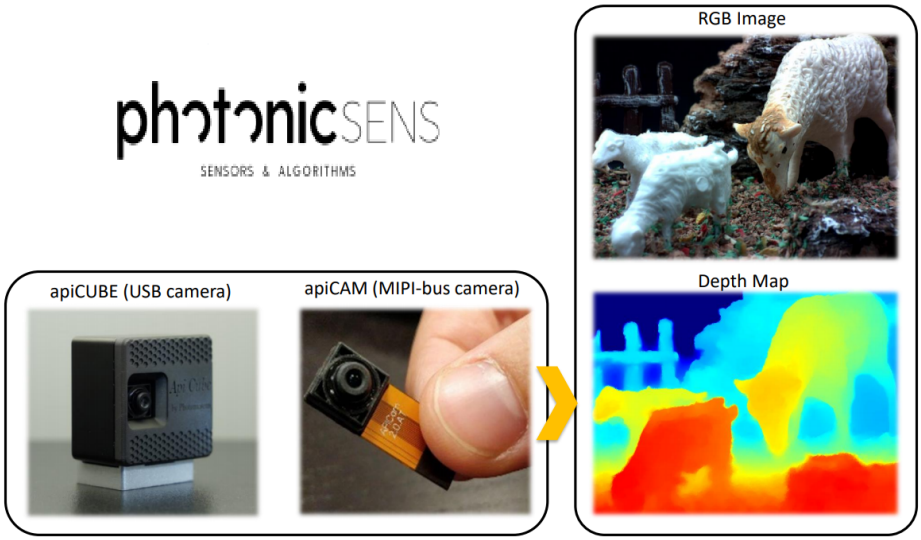



Figura 6.1: En esta imagen se muestran dos productos finales de la cámara desarrollada por la empresa photonicSENS® (izquierda) junto a un ejemplo de imagen y mapa de profundidad obtenido (derecha).

actuales del mercado, como los teléfonos móviles, ordenadores, portátiles, etc. En cambio, la imagen de su derecha (apiCam®) es una solución *ad-Hoc* para su montaje en telefonía móvil, la cual utiliza la interfaz MIPI [2] de comunicación con la cámara.

En la parte derecha de la Figura 6.1 podemos observar una imagen (arriba) capturada por una de las cámaras desarrolladas, junto con el mapa de profundidad (abajo) obtenido después de aplicar los algoritmos de rellenado del mapa de profundidad descritos en esta tesis.

A modo de resumen, la solución tecnológica que aquí se presenta se caracteriza por:

- **Módulo 3D**: formado por una única cámara capaz de proporcionar simultáneamente la imagen de la escena junto con su correspondiente mapa de profundidad sin necesidad de la intervención de elementos activos que se lo faciliten.

- **Rango de medición**: desde unos pocos milímetros del objetivo de la cámara hasta varios metros de distancia a la misma (según la configuración de la cámara).

Finalmente, y dada la versatilidad que nos proporciona la interfaz de comunicación USB (ver detalles en la Figura 6.2), ya se han enviado kits de evaluación a potenciales clientes, tanto bajo la plataforma Windows como Android; y actualmente se está trabajando en hacerlo posible bajo Linux también.

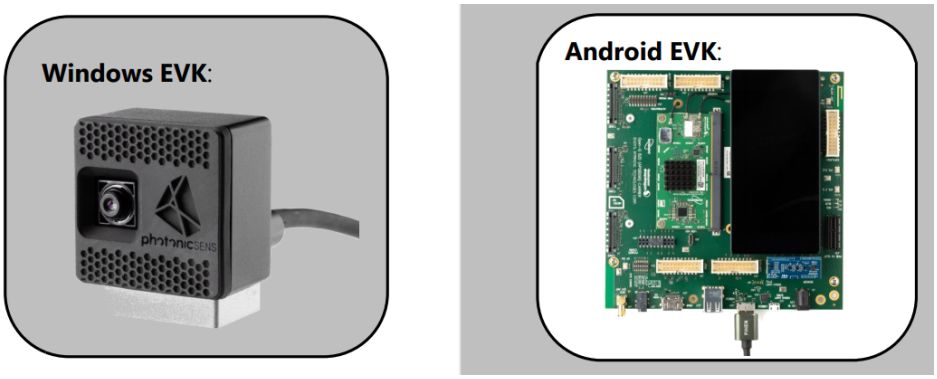



Figura 6.2: En la imagen se muestran dos equipos de evaluación para los sistemas Windows (izquierda) y Android (derecha).

## 6.3 Presentación de los resultados

A la hora de desarrollar un algoritmo para las unidades computacionales de dispositivos "móviles" alimentados por baterías es importante, además del tiempo de cómputo para la obtención del resultado, analizar el número de operaciones en cada una de las unidades computacionales (CPU, GPU, etc.) y el consumo que conlleva conseguir dicho resultado. Una de las principales preocupaciones de los clientes es el consumo energético, siendo este detalle crítico para evitar drenar la batería muy rápidamente. Es por ello que, a la hora de desarrollar los algoritmos presentados en esta tesis, se ha tenido muy en cuenta el uso de los recursos computacionales que utilizan y, por consiguiente, el consumo de la batería que realizan, con la finalidad de buscar un compromiso entre eficiencia computacional y de consumo[1].

Además del ratio *GFlop/Watt* entre la eficiencia computacional y de consumo, también es de suma importancia el uso de memoria principal que se realiza, ya que, por ejemplo, los dispositivos móviles no disponen de la cantidad de memoria que poseen los ordenadores o portátiles actuales. Por lo tanto, es de suma importancia hacer un uso óptimo de la memoria por parte de los diferentes algoritmos desarrollados. Nótese que, en la mayoría de las ocasiones, no se puede hacer uso de la totalidad de los recursos presentes en el dispositivo, memoria y cómputo, pues estos se deben compartir con otras tareas en curso y no bloquear así el dispositivo.

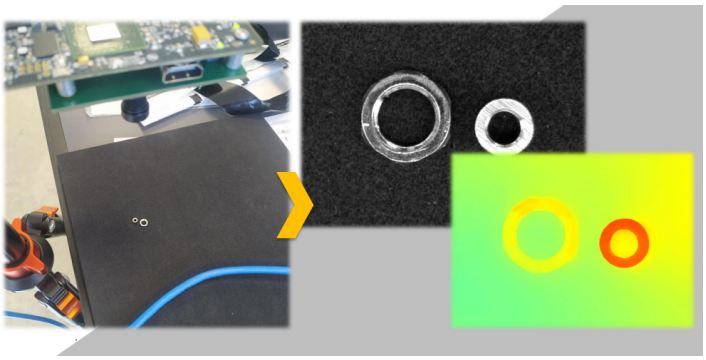
### 6.3.1 Aplicaciones y resultados obtenidos

En este subapartado vamos a presentar diferentes resultados de mapa de profundidad para diferentes configuraciones ópticas de la tecnología desarrollada por la empresa photonicSENS® y comercializada bajo la denominación ApiCam®, presentando posibles aplicaciones que pueden tener en la industria.

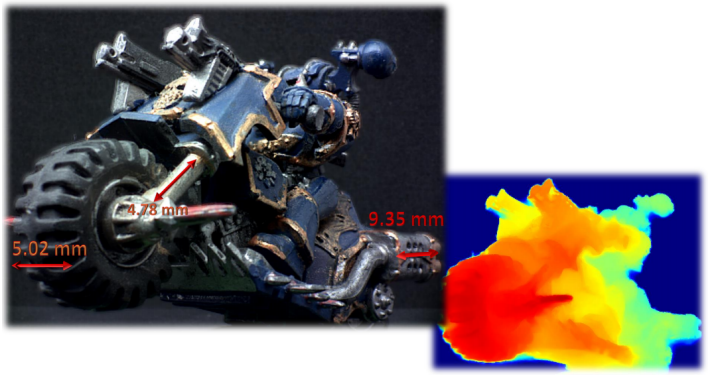
[1] Obviamente el tiempo de cómputo y el consumo para una misma unidad computacional suelen ir aparejados, pero no así cuando se utilizan diferentes unidades computacionales, pues cada una de ellas puede tener un consumo diferente.

### Cámaras para distancias cortas

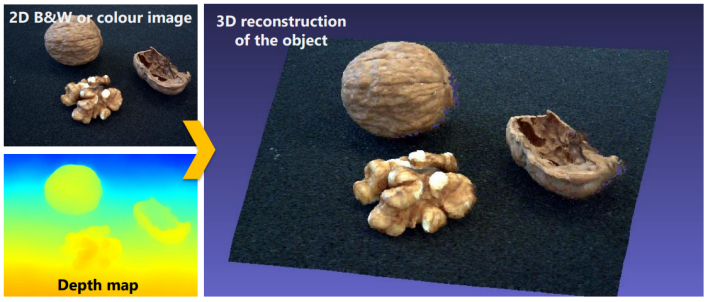
Las primeras configuraciones ópticas presentadas de la cámara ApiCam® son aquellas capaces de adaptarse a aplicaciones que necesiten de una precisión óptima, desde unos pocos milímetros hasta unos pocos centímetros, como pueden ser las aplicaciones de "robótica" de precisión, macro fotografía, reconstrucción 3D de piezas pequeñas o medidas de dimensiones reales de pequeñas piezas, entre otras.



(a) Automatización de procesos




(b) Medidas de distancias

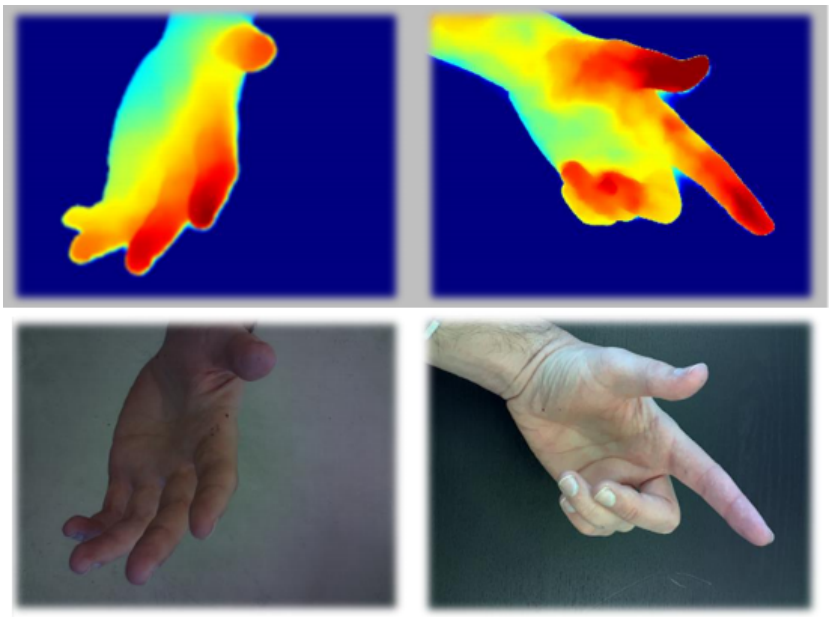



(c) Reconstrucción 3D

Figura 6.3: Ejemplos de tres aplicaciones con cámaras de profundidad capaces de medir distancias cortas con gran precisión.

En la Figura 6.3 se muestran algunos ejemplos de estas aplicaciones junto con el mapa de profundidad obtenido basado en los algoritmos descritos en esta tesis para una configuración óptica desarrollada *ad-hoc* para distancias cortas. Como se pueden apreciar en los diferentes mapas de profundidad obtenidos para las diferentes aplicaciones de ejemplo, son mapas de profundidad que reflejan de una forma muy fiel la realidad de la escena pudiendo competir con el estado del arte en la materia.


(a) Medidas de objetos



(b) Reconocimientos de gestos

Figura 6.4: Ejemplo de medida de una caja de embalaje o reconocimiento de gestos para un kit de testeo de una cámara ApiCam®.

### Cámaras para distancias medias

En este segundo subapartado se presenta una configuración óptica de la cámara ApiCam® capaz de adaptarse a aplicaciones que necesiten de una precisión óptima, desde unos pocos centímetros hasta el metro de distancia. Ejemplos de aplicaciones para este contexto serían la medición de cajas de embalaje o el reconocimiento de gestos, entre otras.

En la Figura 6.4 se muestran unos ejemplos de aplicaciones muy comunes y actuales como puede ser la medida de objetos "medianos" presentes en nuestro entorno o el reconocimiento de gestos para el manejo de diferentes dispositivos digitales, como puede ser el sistema multimedia de un vehículo. En el caso de la imagen de arriba, se realiza la medida de un lado de la caja, pero también se podría medir el volumen y otras características de la misma. En la imagen de abajo, se muestran dos ejemplos de mapas de profundidad para dos gestos realizados con la mano[2].

### Cámaras para distancias largas

En la actualidad se están terminando de implementar y testear nuevos algoritmos que, apoyados en los actuales y haciendo uso de la tecnología que pueda acompañar[3] a la cámara ApiCam®, nos permitan obtener mapas de profundidad precisos para escenas con grandes objetos u objetos lejanos, con la finalidad de abarcar todas las posibilidades y poder posicionar el producto en nuevos mercados.

### Reconocimiento facial, *anti-spoofing*

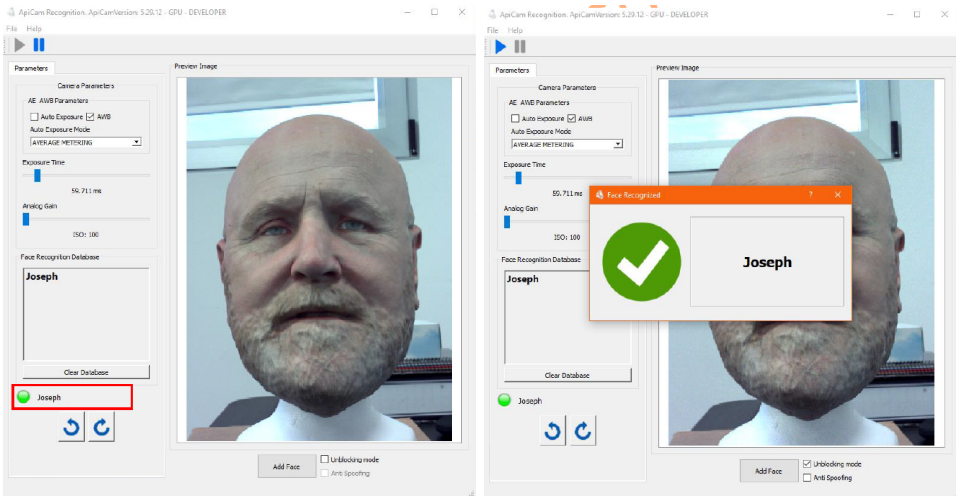
Durante los últimos años, la empresa photonicSENS® ha estado colaborando con diferentes empresas de reconocimiento facial para incorporar la tecnología ApiCam® a sus algoritmos y que, de esta forma, se pueda implementar un *anti-spoofing*[4] facial sin la necesidad de interactuar con el usuario.

Como se puede observar en la Figura 6.5, arriba a la izquierda se muestra una cara a la que vamos a denominar "Joseph" que se encuentra almacenada en el sistema (aparece a la izquierda dentro del recuadro "*Face recognition database*" (Base de datos de reconocimiento facial) con la palabra "Joseph") y por lo tanto se reconoce. Dado que la cara está en el sistema y se reconoce como una cara válida, el sistema reacciona desbloqueando la aplicación (arriba derecha). Finalmente, si activamos la opción "*anti-spoofing*" (podemos ver la opción marcada en la imagen de abajo, justo debajo de donde se muestra la cara), aunque la aplicación reconoce la cara como "Joseph" (recuadro en rojo muestra la luz verde de reconocimiento positivo), no desbloquea el sistema pues se da cuenta de que es una imagen sin profundidad o con cambios de profundidad poco consistentes con lo que se espera del volumen de

[2]Se podría diseñar e implementar una red neuronal de reconocimiento de mapas de profundidad de diferentes gestos "habituales" para el manejo de los diferentes sistemas multimedia presentes en los vehículos actuales.

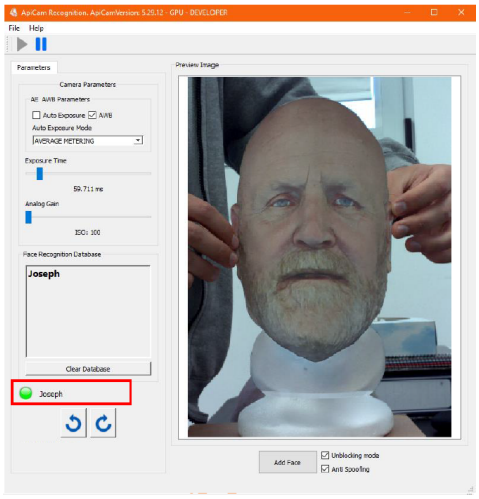
[3]Nótese que, por ejemplo, en un *smartphone* ya es muy habitual encontrar varias cámaras y que, muy probablemente, una cámara como la que se propone iría acompañada de otras cámaras. Estas cámaras tendrían, sin duda, otras funcionalidades, pero sus imágenes son igualmente aprovechables como información extra siendo, además, información muy valiosa por su diferente *baseline* y lo que esto supone para resolver distancias más lejanas.

[4]**Spoofing facial**, consiste en imitar o usar la cara de una persona para hacerse pasar por ella, e intentar engañar a los sistemas de identificación facial.




(a) Cara reconocida como Joseph

(b) Cara reconocida y válida




(c) Cara reconocida como Joseph pero no válida.

Figura 6.5: Reconocimiento facial con sistema pasivo de *anti-spoofing*.

una cara. Por lo tanto, al realizar el reconocimiento facial con la información 3D de la escena capturada la aplicación ofrece una mayor robustez ante intentos de engaño.

### 6.3.2 Comparativa del mapa de profundidad con la competencia

#### Mapa de profundidad de alta resolución

A diferencia de algunas tecnologías con las que compite ApiCam®, como es el *Time of Flight (TOF)*, el mapa de profundidad proporcionado por los algoritmos descritos en la presente tesis son

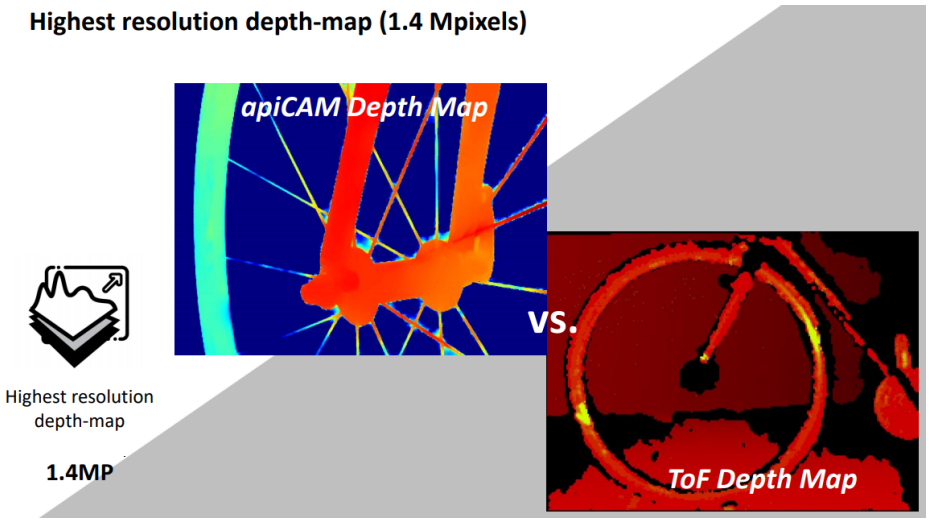



Figura 6.6: Ejemplo de mapa de profundidad de alta resolución de la cámara ApiCam® comparado con el de un TOF.

capaces de conservar pequeños detalles como se puede apreciar en la Figura 6.6, donde el mapa de profundidad de la rueda de una bicicleta proporcionado por la tecnología apiCam®, es capaz de diferenciar la profundidad a la que se encuentran los diferentes radios que componen la rueda, mientras la imagen de profundidad proporcionada por la tecnología TOF no es capaz de capturar estos detalles, proporcionando un mapa de profundidad incompleto e "irreal" de la escena capturada.

### Comparativa de precisión

A continuación vamos a mostrar las gráficas comparativas de los resultados obtenidos a partir del mapa de profundidad denso explicado en esta tesis (Capítulo 5) relativos al porcentaje de error que se obtiene respecto de la distancia real a la que se encuentran los objetos de la escena. Los resultados se muestran en diferentes condiciones y comparándolos con las tecnologías que podemos encontrar actualmente en el mercado en los dispositivos móviles de vanguardia.

En la Figura 6.7 podemos ver una gráfica comparativa de las diferentes tecnologías analizadas en entornos de iluminación interior (la iluminación no es luz natural). Comparando los resultados de profundidad proporcionados por cada tecnología con la distancia real de cada punto de la imagen se obtiene un valor de precisión en base al error porcentual que acarrean a las diferentes distancias estimadas y comparadas. Así, se muestran diferentes tecnologías (ver la leyenda superior izquierda), de arriba a abajo son: *Structure Light* Iphone 11 (cámara frontal), configuración estéreo Iphone 11 (cámara trasera), *Time of Flight* de la compañía PMD[5], 4 configuraciones ópticas distintas de nuestras cámaras ApiCam (H-710, L-575, DL-550 y DIL-470) y por último el *Lidar* del Ipad Pro.

[5] https://pmdtec.com/

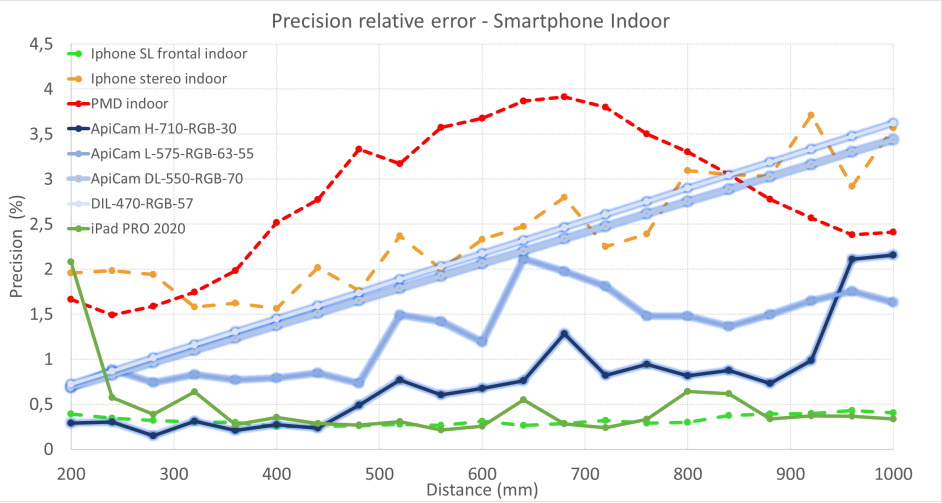



Figura 6.7: Porcentaje de error en escenas interiores para las diferentes tecnologías analizadas. La precisión de las cámaras sería la inversa, es decir, $1/error$.

Como podemos apreciar en la Figura 6.7, el *Structure Light* de la cámara frontal del Iphone y el *Lidar* de la cámara trasera del Ipad Pro, son las dos tecnologías con mejor precisión obteniendo un porcentaje de error de entorno al 0,5 % hasta 1 metro de distancia. Cerca de ellos, se coloca la configuración óptica de la cámara ApiCam H-710, cámara diseñada con una óptica adecuada para abarcar distancias desde unos pocos centímetros hasta el metro de distancia, obteniendo unos resultados de precisión comparables al *Structure Light* del Iphone hasta el medio metro de distancia, empeorándolo en otro medio grado porcentual entre el medio metro y hasta aproximarse al metro de distancia. El *Lidar* del Ipad Pro funciona con esa tasa de error a partir de los 25 centímetros de distancia, pues anteriormente obtiene un porcentaje de error de la estimación de profundidad mayor.

Por otra parte, las ApiCam DL-550 y DIL-470 obtienen un comportamiento lineal con la distancia en cuanto a su tasa de error, pues parte de una tasa de error aproximada del 0,75 % a 20 centímetros hasta alcanzar una tasa de error de 3,5 % al metro de distancia.

Finalmente, resaltar que las dos peores tecnologías de esta comparativa son el *Time of Flight* de PMD y la configuración estéreo de las cámaras traseras del Iphone 11; estas tecnologías presentan un error de estimación de entre el 1,5 % y el 4,0 % en el rango de distancias comprendido entre los 25 centímetros y el metro.

En la Figura 6.8, a diferencia de la Figura 6.7, se compara la tasa de error de las tecnologías en condiciones de iluminación natural o al aire libre. En esta gráfica podemos apreciar que las dos mejores tecnologías en cuanto a precisión son el *Time Of Flight* de Iphone 11 y la ApiCam H-710. En este caso no se ha evaluado el *Lidar* de Ipad Pro, pues al basarse en la proyección y recepción de un patrón de haz lumínico para calcular la distancia a la que se encuentran los objetos, este no funciona correctamente en condiciones de "mucha" iluminación como puede ser en un día soleado al aire libre.

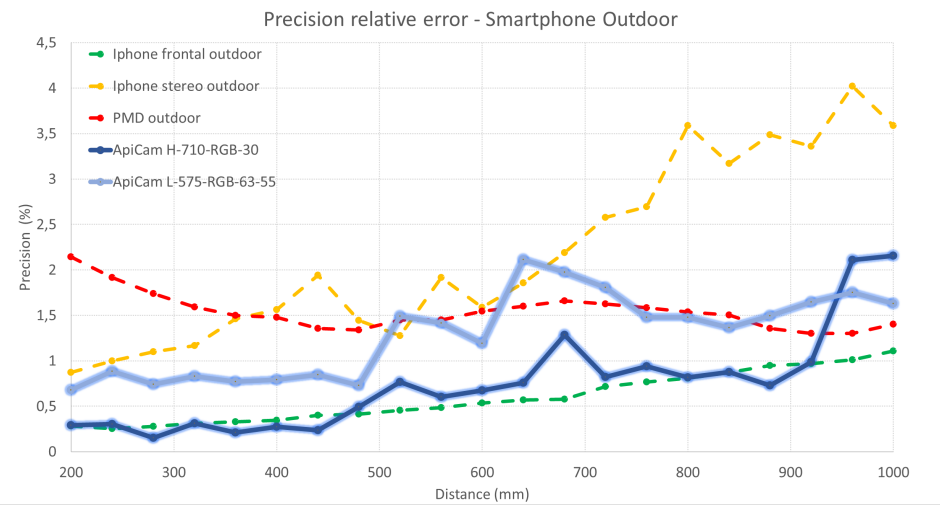



Figura 6.8: Porcentaje de error en escenas exteriores para las diferentes tecnologías analizadas. La precisión de las cámaras sería la inversa, es decir, $1/error$.

Ambas tecnologías poseen una tasa de error próxima al 0,5 % en el rango de distancias comprendidas entre los 25 centímetros y el metro de distancia.

De igual modo que ocurre en las condiciones de iluminación interior (Figura 6.7), las dos peores tecnologías en condiciones de iluminación natural son el *Time Of Flight* de PDM y la visión estéreo de las cámaras trasera del Iphone 11. A diferencia de lo que ocurre en condiciones de iluminación interior, la tecnología de PDM alcanza una tasa de error menor, manteniéndose prácticamente constante con la distancia desde 30 centímetros hasta 1 metro de distancia en un valor de 1,5 % de precisión.

### 6.3.3 Análisis de los resultados

Como ha quedado patente en las gráficas comparativas anteriores, la calidad del mapa de profundidad de la escena proporcionado por los algoritmos descritos en esta tesis (haciendo uso de la tecnología patentada ApiCam® descrita en la Patente [3]) permite posicionar la presente tecnología en el mercado de vanguardia de los dispositivos móviles. La cámara ApiCam® consigue unas tasas de precisión mucho mejores que la competencia para la solución estéreo (dos cámaras), y unos resultados competitivos frente al resto de tecnologías, posicionándose en este último caso muy cerca de la tasa de error de las mejores tecnologías actuales.

Por si esto fuera poco, la mayoría de las tecnologías comparadas con ApiCam®, son tecnologías activas, es decir, que se apoyan en elementos electrónicos activos como puede ser un proyector de puntos infrarrojo para poder hacer las estimaciones de profundidad, hecho que provoca un mayor consumo energético, así como un encarecimiento final de la tecnología al tener que disponer de un mayor número de elementos electrónicos diferentes para tal fin.

La única tecnología aparte de ApiCam® que no necesita elementos activos para calcular el mapa

de profundidad de la escena, es la tecnología estéreo, el resto de tecnologías aquí nombradas precisan de elementos activos para el cálculo del mapa de profundidad, incluida la Realsense de Intel, que si bien se basa principalmente en la tecnología estéreo, algunos de sus modelos hacen uso de elementos activos para rellenar la totalidad del mapa de profundidad.

Resumiendo la comparativa, los resultados presentados manifiestan que los algoritmos desarrollados en esta tesis son implantables en productos comerciales de mercado en telefonía móvil, proporcionando unos resultados de estimación de profundidad muy competitivos frente a los de la competencia. Estos mismos resultados están siendo presentados a potenciales clientes por parte de la empresa photonicSENS®, estando en este momento a punto de cerrar contratos con importantes fabricantes. Además, la tecnología ApiCam® es una tecnología con un precio por unidad muy competitivo debido al menor número de elementos electrónicos y cámaras de las que precisa para poder estimar la profundidad, además del ahorro en consumo energético que esto representa.

# Capítulo 7

# Contribuciones, conclusiones y trabajo futuro

**No son milagros; solo resultados del trabajo duro.**
Maria Petrou (1953 DC - 2012 DC).

**Disfrutar de lo que haces lleva a la perfección en el trabajo.**
Aristotle (384 AC - 322 AC).

## 7.1 Introducción

En este capítulo se presentan las principales conclusiones sobre las contribuciones aportadas por esta tesis. Como bien se ha mencionado en el capítulo de introducción (Capítulo 1), esta tesis se encuentra dividida en dos partes diferenciadas pero muy interrelacionadas entre ellas. Por un lado nos encontramos con los trabajos desarrollados para el filtrado de señales de audio, mientras que por el otro hemos desarrollado algoritmos para imagen por computador, concretamente para el rellenado de un mapa de profundidad disperso de la escena.

En la primera parte de esta tesis se ha trabajado con un conjunto de filtros "simples" y "comunes" para señales de audio como son los filtros FIR e IIR, implementando estos algoritmos comunes en las telecomunicaciones con instrucciones vectoriales NEON®. Como segundo aporte de esta primera parte, se ha desarrollado una implementación eficiente para dispositivos portables del algoritmo de separación de señales *Beamforming* donde, mediante el uso de diferentes librerías, se ha conseguido una implementación propia de la factorización QR de una matriz en paralelo, así como de la actualización de la misma cuando únicamente cambian de una interacción a la siguiente un porcentaje de los datos originales.

En la segunda parte de esta tesis, se presentan metodologías de imagen por computador alrededor de una implementación optimizada para dispositivos de bajo coste/consumo del algoritmo del KNN.

La función primordial de este proceso radica en "rellenar" el mapa de profundidad disperso con el algoritmo del KNN pesado por la distancia. Finalmente, se aplica un suavizado al resultado del mismo empleando el algoritmo del Bilateral Solver, presente en las contribuciones externas de la librería OpenCV. Quedan pendientes algunas mejoras ya pensadas pero no implementadas y testadas que detallamos en la sección trabajos futuros.

## 7.2 Contribuciones

No hay duda de la utilidad del proceso que en esta tesis se describe, no solo como aportación a las técnicas de desarrollo e implementación de algoritmos eficientes para dispositivos que se alimentan de baterías, es decir, optimización del ratio *Gflop/watt*, sino también como base de fundamentos para futuros trabajos que puedan sustentarse en los procesos descritos en ella, e incluso reutilizar parte de las técnicas descritas en estas líneas.

Esta tesis tiene dos partes diferentes pero interrelacionadas bajo el mismo campo de investigación: cálculo masivo paralelo en configuraciones híbridas CPU-GPU en dispositivos portables actuales. Las contribuciones en la primera parte se enumeran en forma de artículos en revista y conferencias mientra que, en la segunda parte, lo hacen en forma de patente.

1. Contribución en el filtrado de señales digitales

   *a*) **Artículo:** Accelerating Multi-Channel Filtering of Audio Signal on ARM Processors, **Autores**: José Antonio Belloch Rodríguez, Francisco Javier Alventosa, Pedro Alonso Jordá, Enrique Salvador Quintana Ortí y Antonio Manuel Vidal Maciá, **Revista:** The Journal of Supercomputing (*Impact Factor:* 2.6 (2020)), **Año:** 2017.

      1) *Vectorización filtros FIR e IIR.* Vectorización "manual" empleando instrucciones vectoriales NEON® (típicas de los procesadores ARM®) de los filtros para procesamiento de señales sonoras FIR, IIR y Parallel IIR, obteniendo mejores resultados que con la auto-vectorización realizada por el compilador *gcc* al código base de los algoritmos de referencia.
      2) *Uso como tipo de datos para operar enteros y reales.* También se ha procedido a experimentar con la conversión a datos enteros de los filtros de señales experimentales para así poder operar con ellos. Se ha constatado una mejora en torno al 30 % en coste computacional, por el hecho de operar con datos enteros en vez de utilizar datos reales.

   *b*) **Artículo:** Fast block QR update in digital signal processing, **Autores**: Francisco Javier Alventosa, Pedro Alonso Jordá, Antonio Manuel Vidal Maciá, María Gemma Piñero Sipan y Enrique Salvador Quintana Ortí, **Revista:** The Journal of Supercomputing (*Impact Factor:* 2.6 (2020)), **Año:** 2019.

   *c*) **Artículo:** A pipeline structure for the block QR update in digital signal processing, **Autores**: Manuel Francisco Dolz Zaragozá, Francisco Javier Alventosa, Pedro Alonso Jordá y

Antonio Manuel Vidal Maciá, **Revista:** The Journal of Supercomputing (*Impact Factor:* 2.6 (2020)), **Año:** 2019.

*d*) **Artículo:** A Pipeline for the QR Update in Digital Signal Processing, **Autores**: Manuel Francisco Dolz Zaragozá, Francisco Javier Alventosa, Pedro Alonso Jordá y Antonio Manuel Vidal Maciá, **Revista:** Computational and Mathematical Methods (*Impact Factor:* 2.43 (2020)), **Año:** 2019.

1) *Análisis del coste por partes del algoritmo del Beamforming.* Se realizado un implementación propia del algoritmo del *Beamforming* con dos objetivos principales. Por una parte el de analizar el coste de cada una de las partes que lo componen, para así poder enfocar nuestros esfuerzos en reducir el tiempo computacinal de la/s mas costosa/s. Por otra parte, en esta primera implementación se hace uso de diferentes implementaciones de las liberías BLAS y LAPACK con la finalidad de determinar qué implementaciones se encuentran mejor optimizadas para los chips ARM®.
2) *Actualización rápida factorización QR.* Implementación del algoritmo para la actualización de la factorización matricial QR de una matriz rectangular (sobredimensionada, es decir, con más sistemas de ecuaciones que incógnitas). Se realizan las implementaciones de la actualización tanto haciendo uso de la CPU únicamente, como del conjunto CPU-GPU.
3) *Pipeline actualización factorización QR.* Primero, se implementa el *pipeline* con tareas de OpenMP, para posteriormente hacer uso de la librería GRPPI de C++ para el diseño de un *pipeline* que realice la actualización de la factorización QR de una matriz rectangular del algoritmo del *Beamforming*.

2. Contribución en la estimación de mapas densos de profundidad

*a*) **Patente:** Light-Field optical image system with dual mode, **Autores**: Francisco Javier Alventosa, Jorge Vicente Blasco, Francisco Ribes, Carles Montoliu, Javier Grandía, Ivan Virgilio Perino y Adolfo Martínez, **Código Patente:** *filed*, PCT/EP2021/062775, **Año:** 2020.

1) *Patente MEMS.* Diseño y redacción patente MEMS [3] para la obtención de una imagen de alta resolución en conjunto con un mapa de profundidad con la misma resolución (Resolución máxima en megapixeles del sensor).
2) *KNN pesado.* Diseño e implementación del algoritmo del KNN pesado adaptado a la problemática del mapa de profundidad disperso.
3) *Suavizado mapa denso.* Prueba y aplicación de diferentes filtros de suavizado para mejorar el aspecto general del mapa denso ante posibles errores. Se prueban diferentes filtros como son: de media, de mediana y el Bilateral Solver para finalmente aplicar uno o algunos de ellos dependiendo de la configuración.

## 7.3 Notas finales y trabajo futuro

La primera parte de la tesis se ha dado por cerrada a la espera de procesadores móviles más potentes y que nos permitan calcular el algoritmo del *Beamforming* en "tiempo real". Para la segunda parte de la tesis sí que se tiene en mente la realización de diferentes trabajos de investigación con la finalidad de mejorar la calidad y prestaciones del mapa denso de profundidad proporcionado por la cámara ApiCam®.

En este apartado vamos a introducir las nuevas ideas planteadas durante el desarrollo de esta tesis y que derivarán en trabajos futuros a desarrollar y explorar con la finalidad de obtener mejores resultados, tanto en la calidad como en la velocidad en la que obtenemos el mapa denso de la escena.

Una primera idea intuitiva que surge después de desarrollar esta tesis es a partir del uso del *kd-tree* como estructura de búsqueda para los $k$-vecinos más cercanos, es desarrollar la construcción y/o la búsqueda de los $k$-vecinos más cercanos en la GPU disponible. En la actualidad, existen muchos trabajos realizados en este contexto, como pueden ser, por nombrar algunos, [68, 76, 78]. La problemática en la que nos encontramos en este caso, es que al utilizar dispositivos de bajo consumo para dispositivos móviles, las GPUs disponibles son GPUs donde se persigue un buen ratio *GFlop*//textitWatt, por lo que son GPUs con una capacidad computacional muy discreta comparada con sus homónimas de los equipos desarrollados para estar conectados a la red eléctrica, pues estos no tienen restricciones de consumo. Es por ello, que estas GPUs "móviles" incorporan un número de cores o núcleos computacionales muy reducido, además de disponer de un número mucho menor de multiprocesadores en los que que agruparían los cores computacionales. Por todo lo dicho anteriormente, se debe realizar un estudio del estado del arte de la materia, *kd-tree* en GPUs, con la finalidad de estudiar y adaptar algunas de las soluciones, o incluso desarrollar una estrategia nueva para este tipo de dispositivos "móviles".

Un segunda idea que se ha estudiado consiste en el desarrollo e implementación de una red neuronal que, a partir del mapa disperso (*sparsemap*), consiga obtener un mapa denso (*densemap*) de la escena. Para ello, el doctorando se ha formado mediante la realización de un curso de redes neuronales de la plataforma **Coursera** titulado de *introducción a las redes neuronales convolucionales* y de 100 horas de duración, con la finalidad de obtener la base para poder así empezar a estudiar y probar diferentes trabajos desarrollados en este ámbito por otros investigadores y, finalmente, poder desarrollar e implementar una red neuronal capaz de realizar dicha tarea. Si bien el curso ya ha sido recibido por el doctorando, aún no se ha introducido en materia del desarrollo e implementación de la red neuronal propia, sino que únicamente se han realizado unas primeras pruebas de diferentes redes neuronales existentes y analizado su funcionamiento y resultado propuesto.

Como tercera idea, se está trabajando en un detector de movimiento más robusto que el implementado actualmente, pues este al tener únicamente en cuenta los cambios de intensidad, provoca que sea muy sensible al ruido. El nuevo detector de movimiento, deberá tener en cuenta además de la intensidad, otras características de la imagen como el color o bordes de los objetos para así poder predecir mejor cuando una escena se ha movido. Este nos indica qué píxeles han cambiado de un *frame* al siguiente, de forma que únicamente calculemos los vecinos más cercanos para aquellos píxeles que

han variado, siempre y cuando el número de píxeles que hayan variado no supere un porcentaje del total de píxeles que forman la escena, en cuyo caso se recalcularía por completo.

Como cuarta idea, en aras de reducir el tiempo de búsqueda de los vecinos, se plantea estudiar la implementación de un *kd-tree* bidireccional, es decir, que a partir de los hijos se pueda también acceder a los padres. De esta forma y dado que los píxeles vecinos espacialmente se encuentran próximos en el array de *query points*, podríamos iniciar la búsqueda de los vecinos más cercanos para un píxel dado a partir del último vecino encontrado del píxel anterior, reduciendo de esta forma el número de vecinos a explorar y, por tanto, de distancias a calcular. Obviamente esta solución plantearía una complicación importante de la lógica del algoritmo de búsqueda de los vecinos más cercanos.

Como quinta idea, se están implementando y testando diferentes algoritmos de detección de *background* con la finalidad de reducir tiempos de cómputo, evitando tener que calcular la profundidad de los píxeles que lo forman. Los píxeles de *background* se etiquetan con una profundidad predefinida como *background* (o indefinida) sin necesidad de aplicarle los algoritmos descritos en la presente tesis y centrando la atención en el *foreground*.

Finalmente, se ha conseguido firmar un acuerdo de colaboración con la empresa líder del sector de los procesadores para telefonía móvil e *internet of things*, Qualcomm®, mediante el cual la empresa photonicSENS® va a tener acceso prioritario a la nueva tecnología desarrollada por esta empresa, sus documentos descriptivos de los nuevos productos, así como cursos de formación y soporte facilitados por ingenieros de Qualcomm®. Haciendo uso de este nuevo acuerdo, se pretende que el doctorando consiga una formación más específica y cualificada en los dispositivos de esta compañía y que, de esta forma, pueda desarrollar nuevos algoritmos que realicen un mejor uso de los dispositivos computacionales disponibles. También se pretende hacer uso del soporte de los ingenieros de la compañía para que se puedan mejorar los algoritmos existentes y el desarrollo de otros nuevos. También se prevé probar los algoritmos en las nuevas unidades computacionales pues, sin prácticamente esfuerzo de ingeniería, se podrá obtener una mejora significativa en cuanto al tiempo computacional necesario para la obtención del mapa de profundidad denso de escena, y todo gracias al avance del *hardware* desarrollado por la empresa.

# Bibliografía


[1] The OmpSs Programming Model. https://pm.bsc.es/ompss. Accessed on May 2017.

[2] MIPI Alliance. Mipi camera serial interface 2 (mipi csi-2). https://www.mipi.org/specifications/csi-2, 2021.

[3] F. J. Alventosa, J. V. Blasco, F. Ribes, C. Montoliu, J. Grandía, I. V. Perino, y A. Martínez. Light-field optical image system with dual mode. 15-12-2020. Patente *filed*, PCT/EP2021/062775.

[4] Robert Andrew y Nicholas Dingle. Implementing qr factorization updating algorithms on gpus. *Parallel Computing*, 40(7):161–172, 2014. ISSN 0167-8191. doi:\url{https://doi.org/10.1016/j.parco.2014.03.003}. URL https://www.sciencedirect.com/science/article/pii/S0167819114000337. 7th Workshop on Parallel Matrix Algorithms and Applications.

[5] ARM. Cortex-a15. url: https://developer.arm.com/ip-products/processors/cortex-a/cortex-a15, 2012.

[6] Arm. Server and cloud leaders collaborate to create china-based green computing consortium. https://www.arm.com/company/news/2016/04/server-and-cloud-leaders-collaborate-to-create-china-based-green-computing-consortium, 2016.

[7] J. T. Barron y B. Poole. The fast bilateral solver. *Semantic Scholar*, 1(1):1–50, 2016.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[9] J. L. Bentley. K-d trees for semidynamic point sets. *Proceedings of the sixth annual symposium on computational geometry*, 1(1):187–197, 1990.

[10] J. V. Blasco, C. Montoliu, I. V. Perino, y A. Martínez. A device and methods for obtaining distance information from views. 18-10-2016. Patente pública, US2020134849A1.

[11] Alfredo Buttari, Julien Langou, Jakub Kurzak, y Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008. doi:10.1002/cpe.1301. URL http://dx.doi.org/10.1002/cpe.1301.

[12] Alfredo Buttari, Julien Langou, Jakub Kurzak, y Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009. ISSN 0167-8191. doi:\url{http://dx.doi.org/10.1016/j.parco.2008.10.002}. URL http://www.sciencedirect.com/science/article/pii/S0167819108001117.

[13] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, y Robert van de Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. En *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, págs. 116–125. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-667-7. doi:10.1145/1248377.1248397. URL http://doi.acm.org/10.1145/1248377.1248397.

[14] Ernie Chan, Field G. Van Zee, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, y Robert De Van Geijn. Satisfying your dependencies with supermatrix. En *Proceedings - 2007 IEEE International Conference on Cluster Computing, CLUSTER 2007*, págs. 91–99. 2007. ISBN 1424413885. doi:10.1109/CLUSTR.2007.4629221.

[15] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, y Robert A. van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. En Siddhartha Chatterjee y Michael L. Scott, eds., *PPOPP*, págs. 123–132. ACM, 2008. ISBN 978-1-59593-795-7. URL http://dblp.uni-trier.de/db/conf/ppopp/ppopp2008.html#ChanZBQQG08.

[16] J. Chen, A. Adams, N. Wadhwa, y S. W. Hasinoff. Bilateral guided upsampling. *ACM Transactions on Graphics*, 1(203):1–8, 2016.

[17] C. Conti, L. Ducla Soares, y P. Nunes. Dense light field coding: A survey. *Journals and Magazines*, 8(1), 2020.

[18] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, y José Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24), 2017. doi:10.1002/cpe.4175. URL https://doi.org/10.1002/cpe.4175.

[19] S. Dhanabal y S. Chandramathi. A review of various k-nearest neighbor query processing techniques. *International Journal of Computer Applications*, 31(7):1–9, 2011.

[20] Lijun Ding y Ardeshir Goshtasby. On the canny edge detector. *Pattern Recognition*, 34(3):721–725, 2001.

[21] J. Dongarra y et all. Plasma users guide. url: http://icl.cs.utk.edu/plasma, 2015.

[22] Ruofei Du, Eric Turner, Maksym Dzitsiuk, Luca Prasso, Ivo Duarte, Jason Dourgarian, Joao Afonso, Jose Pascoal, Josh Gladstone, Nuno Cruces, et al. Depthlab: Real-time 3d interaction with depth maps for mobile augmented reality. En *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, págs. 829–843. 2020.

[23] Guang-Ren Duan. On the solution to the sylvester matrix equation av+bw=evf. *IEEE Transactions on Automatic Control*, 41(4):612–614, 1996. doi:10.1109/9.489286.

[24] A. Duch, V. Estivill-Castro, y C. Martínez. Randomized k-dimensional binary search trees. *Proceedings of the Ninth Annual International Symposium on Algorithms and Computation*, 1(1):199–209, 1998.

[25] S. Dutta. Depth-aware blending of smoothed images for bokeh effect generation. *Journal of Visual Communication and Image Representation*, 77(1):1–10, 2021.

[26] M. Escrivá, J. Blasco, F. Abad, E. Camahort, y R. Vivó. Un sistema de caché para la visualización interactiva de campos de luz de alta resolución. En C. Andújar y J. LLuch, eds., *CEIG 2009: Congreso Español de Informática Gráfica*, tomo 1, págs. 75–84. EG, 2009.

[27] J.H. Friedman, F. Baskett, y L.J. Shusket. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers*, 24(1):1000–1006, 1975.

[28] IGI Global. What is depth map. https://www.igi-global.com/dictionary/depth-map/68286, 2021.

[29] G.H. Golub y C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013. ISBN 9781421407944. URL https://books.google.es/books?id=X5YfsuCWpxMC.

[30] Brian C. Gunter y Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, 2005. URL http://doi.acm.org/10.1145/1055531.1055534.

[31] K. He, J. Sun, y X. Tang. Guided image filtering. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 35(1):1–13, 2013.

[32] S. B. Holgersson. Optimising IIR Filters Using ARM NEON. Malmö University, 2011. Technology and society, Computer science.

[33] Lytro inc. https://www.lytro.com/.

[34] A. Ivan, Williem, y I. K. Park. Light field depth estimation on off-the-shelf mobile gpu. *Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2018.

[35] Andre Ivan, Williem, y In Kyu Park. Light field depth estimation on off-the-shelf mobile gpu. En *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. 2018.

[36] Thierry Joffrain, Enrique S. Quintana-Ortí, y Robert A. van de Geijn. Rapid development of high-performance out-of-core solvers. En *Applied Parallel Computing, State of the Art in*

*Scientific Computing, 7th International Workshop, PARA 2004, Lyngby, Denmark, June 20-23, 2004, Revised Selected Papers*, págs. 413–422. 2004. doi:10.1007/11558958_49. URL http://dx.doi.org/10.1007/11558958_49.

[37] Andrew Kerr, Dan Campbell, y Mark A. Richards. QR decomposition on gpus. En David R. Kaeli y Miriam Leeser, eds., *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2009, Washington, DC, USA, March 8, 2009*, tomo 383 de *ACM International Conference Proceeding Series*, págs. 71–78. ACM, 2009. doi:10.1145/1513895.1513904. URL https://doi.org/10.1145/1513895.1513904.

[38] F. F. Kuo y J. F. Kaiser. *System Analysis by Digital Computer*. New York: Wiley, 1966.

[39] K. Li, J. Zhang, R. Sun, X. Zhang, y J. Gao. Epi-based oriented relation networks for light field depth estimation. *British Machine Vision Conference (BMVC)*, 5(3), 2020.

[40] X. Liu, D. Zhai, R. Chen, X. Ji, D. Zhao, y W. Gao. Depth super-resolution via joint color-guided internal and external regularizations. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 28(4):1–10, 2019.

[41] J. Lorente, G. Pi nero, A.M. Vidal, J.A. Belloch, y A. González. Parallel implementations of beamforming design and filtering for microphone array applications. págs. 501–505. Actas de la Conferencia (ISSN 2076-1465), Barcelona, Spain, 2011.

[42] Chenchi Luo, Yingmao Li, Kaimo Lin, George Chen, Seok-Jun Lee, Jihwan Choi, Youngjun Francis Yoo, y Michael O Polley. Wavelet synthesis net for disparity estimation to synthesize dslr calibre bokeh effect on smartphones. En *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, págs. 2407–2415. 2020.

[43] Microsoft. Puntos suspensivos y plantillas variádicas. https://docs.microsoft.com/es-es/cpp/cpp/ellipses-and-variadic-templates?view=msvc-170, 2021.

[44] Microsoft. Representación iee de punto flotante. https://docs.microsoft.com/es-es/cpp/build/ieee-floating-point-representation?view=msvc-170, 2021.

[45] R. Ng. *Digital Light Field Photography*. Dissertation, Computer Science Department. Stanford University, 2006.

[46] Rang M. H. Nguyen y M. S. Brown. Why you should forget luminance conversion and do something better. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1(1):6770–6758, 2017.

[47] Nvidia. Jetson tk1 development pack 1.0. 2014.

[48] Nvidia. Nvidia presenta el primer superordenador móvil para sistemas integrados. url: https://www.nvidia.es/object/nvidia-jetson-tk1-devkit-mar25-2014-es.html, 2014.

[49] Nvidia. Nvidia basic linear algebra subroutines. url: https://developer.nvidia.com/cublas, 2019.

[50] OpenCL. Opencl library. url: https://www.khronos.org/opencl/, 2021.

[51] AV. Oppenheim, AS. Willsky, y S. Hamid. *processing series*, págs.–. Prentice-Hall, Upper Saddle River, 1997.

[52] A.V. Oppenheim, A.S. Willsky, S.H. Nawab, y G.M. Hernández. *Señales y sistemas*. Prentice Hall Hispanoamericana, 1997.

[53] F. Perez-Nava. Super-resolution in plenoptic cameras by the integration of depth from focus and stereo. *International Conference on Computer Communications and Networks (ICCCN)*, 1(1), 2010.

[54] Stefano Pini, Guido Borghi, Roberto Vezzani, Davide Maltoni, y Rita Cucchiara. A systematic comparison of depth map representations for face recognition. *Sensors*, 21(3):944, 2021.

[55] V. B. Surya Prasath, H. A. Abu Alfeilat, A. B. A. Hassanat, O. Lasassmeh, A. S. Tarawneh, M. B. Alhasanat, y H. S. Eyal Salman. Effects of distance measure choice on knn classi
er performance - a review. *Mary Ann Liebert*, 1(1):1–39, 2019.

[56] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, y Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3):14:1–14:26, 2009. ISSN 0098-3500. doi:10.1145/1527286.1527288. URL http://doi.acm.org/10.1145/1527286.1527288.

[57] L. R. Rabiner y B. Gold. *Theory and application of digital signal processing*. Prentice-Hall, Englewood Cliffs, N.J, 1975.

[58] LR Rabiner, JF Kaiser, O Herrmann, y MT Dolan. Some comparisons between fir and iir digital filters. *Bell System Technical Journal*, 53(2):305–331, 1974.

[59] H. Samet. Foundations of multidimensional and metric data structures. *Morgan Kaufmann*, 1(1), 2006.

[60] C. Silpa-Anan y R. Hartley. Optimised kd-trees for fast image descriptor matching. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR08)*, 1(1):1–8, 2008.

[61] C. S. Burrus T. W. Parks. *Digital Filter Design*. John Wiley and Sons Ltd, 1987.

[62] M. W. Tao, S. Hadap, J. Malik, y R. Ramamoorthi. Depth from combining defocus and correspondence using light-field cameras. *IEEE International Conference on Computer Vision*, 1(1), 2013.

[63] S. Tomov, J. Dongarra, y M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. url: http://icl.cs.utk.edu/magma, 2010.

[64] M. I. Villa. Narrativas inmersivas para comunicadores. realidad virtual, aumentada y mixta en propuestas audiovisuales de ficción y no ficción. *Dialnet*, 1(1):1–6, 2018.

[65] N. Wadhwa, R. Garg, D. E. Jacobs, B. E. Feldman, N. Kanazawa, R. Carroll, Y. Movshovitz-Attias, J. T. Barron nand Y. Pritch, y M. Levoy. Synthetic depth-of-field with a single-camera mobile phone. *ACM Transactions on Graphics*, 37(4), 2018.

[66] M. Wax y T. Kailath. Efficient inversion of toeplitz-block toeplitz matrix. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 31(5):1218–1221, 1983. doi:10.1109/TASSP.1983.1164208.

[67] Weekly-Geekly. Cámaras de profundidad: revolución silenciosa (cuando los robots lo verán) parte 1. https://weekly-geekly-es.imtqy.com/articles/457524/index.html, 2021.

[68] D. Wehr y R. Radkowski. Parallel kd-tree construction on the gpu with an adaptive split and sort strategy. *International Journal of Parallel Programming*, 46(1):1139–1156, 2018.

[69] Florian Wende, Thomas Steinke, y Frank Cordes. Multi-threaded kernel offloading to gpgpu using hyper-q on kepler architecture. Inf. Téc. 14-19, ZIB, Takustr.7, 14195 Berlin, 2014.

[70] Wikipedia. Kd-tree. url: https://es.wikipedia.org/wiki/%C3%81rbol_kd, 2021.

[71] C. T. Wu, L. F. Isikdogan, S. Rao, B. Nayak, T. Gerasimow, A. Sutic, L. Ain-Kendem, y G. Michael. Visionisp: Repurposing the image signal processor for computer vision applications. *IEEE International Conference on Image Processing (ICIP)*, 1(1):4624–4628, 2019.

[72] G. Wu, B. Masia, A. Jarabo, Y. Zhang, L. Wang, Q. Dai, T. chai, y Y. Liu. Light field image processing: An overview. *Journals and Magazines*, 11(7):926–954, 2017.

[73] Xianyi. An optimized blas library. url: http://www.openblas.net, 2019.

[74] C. Yang, Z. Liu, K. Di, C. Hu, Y. Wang, y W. Liang. Improved depth estimation for occlusion scenes using a light-field camera. *Photogrammetric Engineering and Remote Sensing*, 86(7):443–456, 2020.

[75] D. Yang, X. Zhong, D. Gu, X. Peng, G. Yang, y C. Zou. Unsupervised learning of depth estimation, camera motion prediction and dynamic object localization from video. *International Journal of Advanced Robotic Systems*, 1(1), 2020.

[76] S. Zellmann, J. P. Schelze, y U. Lang. Binned k-d tree construction for sparse volume data on multi-core and gpu systems. *Journal and Magazines: IEEE Transactions on Visualization and Computer Graphics*, 27(3):1–12, 2019.

[77] Shuai Zhang, Chong Wang, y Shing-Chow Chan. A new high resolution depth map estimation system using stereo vision and kinect depth sensing. *Journal of Signal Processing Systems*, 79(1):19–31, 2015.

[78] K. Zhou, Q. Hou, R. Wang, y B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):1–11, 2008.