



UNIVERSIDAD DE MURCIA

ESCUELA INTERNACIONAL DE DOCTORADO

Desarrollo de sistemas de tiempo real mediante
el uso de aceleradores gráficos y específicos
aplicados a la óptica visual humana

D. Juan Mompéan Esteban
2020

Desarrollo de sistemas de tiempo real mediante el uso de aceleradores gráficos y específicos aplicados a la óptica visual humana

Autor

Juan Mompéan Esteban

Directores

Juan Luis Aragón Alcaraz

Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia

Pablo Artal Soriano

Laboratorio de Óptica, Departamento de Física, Universidad de Murcia



Doctorado en Informática



UNIVERSIDAD DE
MURCIA



Preámbulo

Esta Tesis viene motivada por la necesidad de procesar grandes cantidades de datos en un corto espacio de tiempo. El tipo de datos sobre los que se ha trabajado presenta en la mayoría de los casos una estructura uniforme y un procesamiento repetitivo, aunque no siempre es así. Por ello la paralelización del procesamiento se ha llevado principalmente mediante procesadores gráficos (GPUs) que obtienen un muy buen rendimiento ante este tipo de algoritmos, y en algunos casos con aceleradores específicos implementados en una FPGA. Uno de los campos en los que se ha aplicado la paralelización del procesado ha sido en la óptica adaptativa. En ésta, el procesamiento de imágenes o de otros datos en tiempo real es de una gran utilidad ya que permite llevar a cabo experimentos dinámicos mucho más complejos que los experimentos que se realizan de forma estática. Sin embargo, procesar esa gran cantidad de datos con suficiente velocidad siempre ha sido un reto. El procesamiento de dichos datos en la CPU no suele obtener el rendimiento suficiente como para que los sistemas funcionen en tiempo real. Además, el uso de las GPUs (o aceleradores específicos) permite que las CPUs dispongan de más recursos para gestionar la multitud de dispositivos que suelen tener que controlar en este tipo de experimentos: cámaras, lentes optoelectrónicas, motores, iluminación, pantallas, entre otros. En resumen, esta Tesis ha servido para superar alguno de los retos que presenta la óptica adaptativa sirviendo, además, para el desarrollo de nuevos e interesantes experimentos en este campo.

Agradecimientos

Este trabajo se ha llevado a cabo en el seno de dos departamentos de la Universidad de Murcia, el Laboratorio de Óptica (LOUM), perteneciente al Departamento de Física, y el Departamento de Ingeniería y Tecnología de Computadores (DITEC). No se entiende sin esa sinergia lo que hemos conseguido, sin el apoyo constante de todos los miembros del LOUM que me han ayudado en todo lo que he necesitado sobre óptica y el ojo humano. Dicho sea de paso, el LOUM ha sido el lugar en el que he desarrollado durante estos años mi investigación y con ellos he compartido muchas horas de trabajo, innumerables cafés y numerosas comidas.

El nexo de unión de este trabajo ha sido Juan Luis Aragón por parte de DITEC y Pablo Artal por parte del LOUM. Ellos han sido mis directores en este periplo que ahora llega a su fin y a ellos les debo el haberme dado la oportunidad de llevar a cabo grandes ideas y de afrontar retos que no habrían sido posibles de otra manera. A Juan Luis me gustaría agradecerle su tenacidad y perseverancia, pues gracias a ella la calidad del trabajo que hemos llevado a cabo ha sido excelente. Sin olvidar, por supuesto, las incontables horas que ha dedicado a revisar artículos, resúmenes, y esta Tesis, entre otras cosas. Pablo es una fuente inagotable de innovadoras ideas que nos han llevado a explorar áreas apasionantes y a conseguir resultados que sin ser fáciles de conseguir sí han sido muy satisfactorios. Me gustaría agradecerle su ánimo siempre positivo ante las dificultades que, inevitablemente, siempre surgen cuando se llevan a cabo experimentos originales.

A mi mujer Rosa, y a mi familia no puedo más que agradecerles su comprensión y sobre todo su apoyo incondicional, sin el cuál esta Tesis se habría hecho mucho más larga y dificultosa.

Durante estos años en el LOUM he tenido la oportunidad de trabajar con muchas personas que, de una forma u otra, han contribuido a los resultados que se presentan en esta Tesis. A Carmen Martínez le debo un especial agradecimiento por su constante trabajo

que nos hace mucho más fácil el día a día al resto de compañeros. De Harilaos Ginis he aprendido mucho, su experiencia en ámbitos muy diversos es siempre enriquecedora y divertida. Alexandros Pennos y Dimitrios Christaras han sido dos grandes profesionales pero también dos amigos. Con Consuelo Robles he tenido la suerte de trabajar en muchos experimentos, siempre aprendiendo y disfrutando con una buena amiga. A Silvestre Manzanera quiero agradecerle su predisposición a ayudar y su buen criterio en las innumerables consultas sobre óptica que le he hecho a lo largo de estos años. De Adrián Gambín he aprendido mucho, me gustaría agradecerle sus ganas de hacer un buen trabajo y mejorar continuamente el software que hacemos. De Pedro M. Prieto he aprendido casi todo lo que sé sobre el sensor Hartmann-Shack, gracias por tu paciencia y por tu tiempo. Con Vahid Pourreza he trabajado muy estrechamente en un sistema especialmente importante para mí, en el que se ha llevado al límite la librería de Hartmann-Shack que se ha desarrollado en esta Tesis. Todos los esfuerzos que hemos hecho para que ese sistema funcione lo mejor posible, y lo haga rápido, estoy seguro que pronto nos darán resultados muy interesantes.

Pero son tantas las personas con las que he tenido la suerte de trabajar a lo largo de este tiempo que no me gustaría olvidarme de nadie. Mi más sincero agradecimiento a: Emmanuel Chirre, Esther Berrio, Rahul Dutta, Josua Fernández, Juanma Bueno, Eloy Villegas, Dibyendu Pusti, Fran Ávila, Raúl Duarte, Daniel Sola, Alba Paniagua, Lucía Hervella, Antonio Benito, Augusto Arias, Francisco Javier Roca, Juan Tabernero, Rosa Martínez, Benjamin Klaus Lochocki, Alfonso Jiménez, Ireneusz Grulkowski, Alberto de Castro, Lucie Sawides, Yiwei Chen, Nikolai Suchkov.

Por último, el trabajo desarrollado en esta Tesis no habría sido posible sin el apoyo económico de numerosas becas y subvenciones, entre las que se encuentran: European Research Council Advanced Grant ERC-2013-AdG-339228, SEIDI FIS2013-41237-R, Fundación Séneca (19897/GERM/15), TIN2015-66972-C5-3-R, RTI2018-098156-B-C53, FIS2016-76163-R, PID2019-105684RB-I00.

*A mi mujer Rosa, y a mis padres Juan y Ana,
sin los cuales no habría podido llevar a cabo este trabajo.*

*Lo que puede hacerse o no puede hacerse depende, (...),
de cómo se piensa. Si algo no puede hacerse no hay que dejar de hacerlo
sino cambiar de forma de pensar.*

Santiago Posteguillo.

Resumen

Esta Tesis viene motivada por la necesidad de procesar grandes cantidades de datos en un corto espacio de tiempo. El tipo de datos sobre los que se ha trabajado presenta, en la mayoría de los casos, una estructura uniforme y un procesamiento repetitivo, aunque no siempre es así. Por ello, la paralelización del procesamiento se ha realizado principalmente mediante procesadores gráficos (GPUs) que obtienen un muy buen rendimiento ante este tipo de algoritmos y, en algunos casos, con aceleradores específicos implementados en una FPGA. El trabajo realizado se ha centrado en la paralelización de algoritmos utilizados en diferentes aplicaciones de la óptica.

Esta Tesis se divide en seis capítulos: introducción, seguimiento pupilar de alto rendimiento, gafas de corrección automática y en tiempo real de la presbicia, procesado de imágenes de Hartmann-Shack en tiempo real, paralelización del procesado de imágenes de OCT, y conclusiones y perspectivas.

En cuanto al seguimiento pupilar se han implementado y paralelizado varios algoritmos con el fin de obtener una versión de alto rendimiento. Además, se ha caracterizado un conjunto de imágenes de pupilas con el que se ha llevado a cabo un análisis de la precisión y rendimiento de los diferentes algoritmos implementados. En la implementación GPGPU se han llevado a cabo numerosas optimizaciones para mejorar el rendimiento, consiguiéndose velocidades de procesamiento superiores a 1000 imágenes por segundo en algunas configuraciones.

Partiendo de los avances conseguidos en el capítulo sobre seguimiento pupilar se han desarrollado las gafas de corrección automática y en tiempo real de la presbicia. Este dispositivo es completamente autónomo, siendo controlado únicamente a través de un teléfono móvil inteligente. Dicho teléfono es el encargado de procesar las imágenes de los ojos y, a partir de los datos obtenidos, calcular la corrección que se debe aplicar en las lentes optoelectrónicas. Todo el dispositivo se alimenta de la batería del teléfono móvil y de una pequeña batería externa utilizada para la iluminación infrarroja. Además, se ha llevado a cabo un experimento con 8 sujetos présbitas que han mejorado su agudeza

visual en 0.3 de media en escala decimal sobre un test situado a 35cm de distancia. Por otra parte, se ha realizado una implementación en OpenCL optimizada para FPGAs con el propósito de explorar la idoneidad de estas plataformas para controlar este dispositivo. De forma que se ha medido tanto su consumo como su rendimiento, y se ha comparado con el obtenido por el teléfono inteligente.

En el capítulo relativo a las imágenes de Hartmann-Shack se han paralelizado y optimizado tres algoritmos de procesamiento de éstas para su ejecución en procesadores gráficos. Además, se ha desarrollado, paralelizado y optimizado un novedoso algoritmo de seguimiento pupilar en imágenes de Hartmann-Shack. Asimismo, se ha caracterizado tanto su precisión como su rendimiento, mediante el uso de imágenes generadas mediante simulaciones. Se han obtenido resultados significativos tanto en precisión como en rendimiento. Si bien se han comparado una gran variedad de configuraciones y resoluciones, con algunas de las imágenes utilizadas se consiguen aceleraciones superiores a 100x entre la versión secuencial y la versión paralela GPGPU.

Por último, se ha paralelizado y optimizado el procesamiento de imágenes obtenidas mediante un sistema OCT (Tomografía de Coherencia Óptica). Los sistemas OCT generan grandes cantidades de datos en cada medida, pudiendo alcanzar fácilmente los 2GB según el sistema de adquisición utilizado. Estos datos requieren un procesamiento muy intensivo en cómputo para obtener las imágenes finales. Es por ello que la paralelización de este procesamiento mediante el uso de procesadores gráficos ha arrojado resultados con aceleraciones que alcanzan hasta 8x.

Summary

This Thesis is motivated by the need to process big amounts of data in a short time. The structure of the data to be processed is, in most of the cases, uniform and repetitive, although it is not always the case. Therefore, the parallelization of the processing tasks has been mainly carried out by means of graphics processors (GPUs), which achieve a very high performance for this kind of algorithms, and in some other cases we have implemented the image processing as an specific accelerator in a FPGA. The work accomplished in this Thesis is focused on the parallelization of different compute-intensive processing algorithms used in Optics. In particular, we target the following applications: high performance pupil tracking, automatic and real time presbyopia correction glasses, real time processing of Hartmann-Shack images, and the parallelization of OCT image processing.

Regarding pupil tracking several algorithms have been implemented and parallelized to obtain a high-performance version. Furthermore, a set of images has been characterized and used to perform an accuracy and performance analysis of the different implemented algorithms. Multiple optimizations have been applied to the GPGPU implementation to improve its performance. As a result, pupil tracking at speeds higher than 1000 images/-second have been achieved in some configurations.

Using the tools developed for pupil tracking, an automatic and real time presbyopia correction glasses have been developed. This wearable device is completely autonomous and controlled with a smartphone. The smartphone processes the images of the eyes and, from the obtained data, it calculates the correction that should be applied by the optoelectronic lenses. The whole device is powered by the smartphone battery and another small battery used to power the infrared illumination. To validate the results, a test with 8 subjects has been carried out, obtaining an average improvement of 0.3 in their visual acuity in decimal scale using a test placed at 35cm. Furthermore, a specific OpenCL implementation for FPGAs has been created and optimized with the purpose of testing the suitability of such platforms to control this wearable device. To do so, their power con-

sumption and performance have been measured and compared with the numbers obtained by the smartphone.

Regarding the real-time processing of Hartmann-Shack images, three algorithms have been parallelized and optimized for their execution in graphics processors. A novel pupil tracking algorithm for Hartmann-Shack images has been developed, parallelized and optimized. Both accuracy and performance have been characterized, using H-S images generated by simulations, obtaining good results in both metrics. A wide variety of configurations and resolutions have been tested, and some of them have resulted in a significant speedup of up to 100x when comparing the sequential implementation and the parallel GPGPU one.

Finally, the processing of OCT (Optical Coherence Tomography) images has been parallelized and optimized. OCT systems generate a great amount of data on each measurement, easily reaching more than 2GB of data per volume, as it is the case of the OCT system that we have employed. This huge amount of data requires a highly computing-intensive processing to generate the final 3D images of the eyes. Our parallel implementation of the processing algorithms by means of graphics processors has resulted in speedups of up to 8x.

Índice general

1	Introducción	1
1.1	Computación de propósito general en unidades de procesamiento gráfico	1
1.2	Seguimiento pupilar en el ojo humano	6
1.3	Presbicia y su corrección	8
1.4	Sensor Hartmann-Shack para medir las aberraciones del ojo humano en tiempo real	10
1.5	Tomografía de Coherencia Óptica (OCT)	17
1.6	Estructura de la Tesis	19
2	Seguimiento pupilar de alto rendimiento	23
2.1	Introducción	23
2.2	Sistema de captura de imágenes	26
2.3	Algoritmos, paralelización y optimizaciones	28
2.3.1	Preprocesado	29
2.3.1.1	Paralelización de la fase de pre-procesado	30
2.3.1.1.1	Implementación mediante las NPP	32
2.3.1.1.2	Implementación especializada	33
2.3.1.1.3	Uso de filtros separables	36
2.3.1.1.4	Uso de memoria compartida	39
2.3.1.1.5	Uso de kernels fusionados	41
2.3.1.1.6	Desenrollado de bucles	41
2.3.1.1.7	Optimización del tipo de datos de las imágenes	43
2.3.1.1.8	Eliminación de la interpolación de los reflejos corneales	43
2.3.2	Optimización de la copia de datos entre la CPU y la GPU	44

2.3.3	Ajuste a un círculo o elipse mediante RANSAC (RANdom SAMple Consensus)	45
2.3.3.1	Paralelización del algoritmo RANSAC para GPU	47
2.3.3.1.1	Eliminación del cálculo de los números aleatorios	49
2.3.3.1.2	Kernels fusionados	49
2.3.3.1.3	Streams	49
2.3.3.1.4	Fast math	49
2.3.3.1.5	Reducción de las comunicaciones entre la GPU y el host	50
2.3.4	Umbralización y etiquetado	50
2.3.5	Starburst paralelo	53
2.3.6	Detección de bordes de Canny y Transformada de Hough	58
2.3.7	Implementación OpenCL	61
2.4	Resultados	62
2.5	Sistemas ópticos reales dónde se está utilizando esta implementación paralela del seguimiento pupilar	72
3	Gafas de corrección automática y en tiempo real de la presbicia	75
3.1	Introducción	75
3.2	Antecedentes y estado del arte	76
3.3	Exploración de las posibilidades de distintas implementaciones	80
3.3.1	Implementación en un SoC móvil (smartphone)	81
3.3.1.1	Accesos a memoria vectorizados	82
3.3.1.2	Memoria local	82
3.3.1.3	Operaciones matemáticas nativas	83
3.3.1.4	Tipos de datos especializados	83
3.3.1.5	Fusión de kernels	83
3.3.2	Implementación en un ordenador de sobremesa	84
3.3.3	Implementación en una FPGA	84
3.3.3.1	Memoria local	85
3.3.3.2	Desenrollado de bucles	85
3.3.3.3	Segmentación	86
3.3.3.4	Particionado de arrays	86
3.3.3.5	Max Memory Ports	87

3.4	Diseño y creación de un prototipo	87
3.4.1	Prueba de concepto en una mesa óptica	88
3.4.2	Prueba de concepto portátil	92
3.4.3	Control de las gafas	95
3.4.4	Comportamiento del sistema	99
3.5	Análisis de rendimiento, eficiencia y comportamiento	102
3.5.1	Metodología	102
3.5.2	Rendimiento	102
3.5.3	Eficiencia	104
3.5.4	Experimentos con sujetos	106
4	Procesado de imágenes de Hartmann-Shack de alto rendimiento	113
4.1	Introducción	113
4.2	Paralelización del procesado de imágenes de H-S	115
4.2.1	Seguimiento pupilar	116
4.2.2	Algoritmo PAPYCS	118
4.2.2.1	Búsqueda de los centroides	118
4.2.2.2	Etapas de descarte	124
4.2.2.3	Malla de búsqueda dinámica	124
4.2.2.4	Cálculo de la aberración	125
4.2.3	Algoritmo <i>Reichenbach</i>	127
4.3	Paralelización y optimización	128
4.3.1	Implementación paralela mediante CUDA	129
4.3.1.1	Seguimiento pupilar	129
4.3.1.2	Paralelización y optimizaciones en el algoritmo PAPYCS	130
4.3.1.2.1	Instrucciones shuffle:	133
4.3.1.2.2	Accesos a memoria vectorizados:	133
4.3.1.3	Malla de búsqueda dinámica	133
4.3.1.3.1	Precálculo de los factoriales	134
4.3.1.3.2	Memoria compartida	134
4.3.1.4	Paralelización y optimizaciones en el algoritmo de Reichen-	
	bach	134
4.3.1.4.1	Memoria compartida	135
4.3.1.5	Etapas de descarte	135

4.3.1.6	Cálculo de la aberración	136
4.3.1.7	Mapa de frente de onda	136
4.3.1.7.1	Memoria compartida	137
4.3.1.7.2	Pinned memory	138
4.3.1.7.3	Tipo de datos <i>half</i>	138
4.3.1.7.4	Desenrollado de bucles	139
4.3.1.7.5	Pitched memory	139
4.4	Resultados	140
4.4.1	Metodología de evaluación	140
4.4.2	Rendimiento de los algoritmos paralelos	141
4.4.3	Precisión alcanzada por los diferentes algoritmos	148
4.5	Sistemas ópticos reales donde se está utilizando esta implementación paralela de procesamiento de imágenes de sensores de Hartmann-Shack	151
5	Paralelización del procesamiento de imágenes de OCT	155
5.1	Introducción	155
5.2	Estado del arte	158
5.3	Comparación de implementaciones del procesamiento de imágenes de OCT para varias plataformas	160
5.3.1	Implementación LabVIEW	160
5.3.2	Implementación MATLAB	162
5.3.3	Implementación C++	163
5.3.4	Implementación CUDA	165
5.3.5	Implementación OpenCL	168
5.4	Sistemas implementando este software	169
6	Conclusiones y perspectivas	173
6.1	Conclusiones	173
6.2	Contribuciones de la Tesis	176
6.3	Perspectivas	181
	Bibliografía	185

Índice de figuras

1.1	Organización lógica de los hilos en CUDA. Fuente NVIDIA [6].	4
1.2	Ejemplo de imagen de unos ojos iluminados con luz infrarroja.	6
1.3	Rango de acomodación en función de la edad [8].	9
1.4	(a) Frente de onda aberrado mediante astigmatismo vertical (6º coeficiente de Zernike). (b) Frente de onda aberrado mediante desenfoque (4º coeficiente de Zernike). (c) Frente de onda aberrado mediante astigmatismo oblicuo (5º coeficiente de Zernike). (d) Frente de onda aberrado mediante coma vertical (7º coeficiente de Zernike). (e) Frente de onda aberrado mediante coma horizontal (8º coeficiente de Zernike).	12
1.5	(a) Diagrama de un sensor de frentes de onda H-S registrando un frente de onda plano. (b) Diagrama de un sensor de frentes de onda H-S registrando un frente de onda aberrado. (c) Imagen de los puntos de H-S de un frente de onda perfecto. Arriba se muestra todo el ojo y abajo una parte del mismo ampliada. (d) Imagen de los puntos de H-S de un frente de onda aberrado. Arriba se muestra todo el ojo y abajo una parte del mismo ampliada. . . .	15
1.6	Corte transversal de una córnea y un cristalino con cataratas tomado <i>in-vivo</i> con el sistema OCT del LOUM.	18
2.1	(a) Esquema del sistema óptico utilizado para tomar las imágenes del conjunto de pruebas y para realizar pruebas en tiempo real. (b) Imagen frontal del sistema óptico donde se muestran con precisión los LEDS infrarrojos utilizados para iluminar el ojo.	27
2.2	Histograma que muestra los tamaños de las pupilas presentes en el conjunto de imágenes evaluadas.	28

2.3	(a) Imagen original en la que se pueden ver los reflejos corneales en el interior de la pupila. (b) Resultado de aplicar la transformación <i>tophat</i> (sombrero de copa), donde se destacan los reflejos corneales. (c) Resultado de umbralizar y dilatar la imagen: solamente quedan visibles los reflejos corneales y algunos otros reflejos menores. (d) Resultado de eliminar los reflejos corneales reemplazando sus valores por nuevos valores interpolados con sus vecinos.	30
2.4	Interpolación de los reflejos corneales.	36
2.5	Muestra el cambio que se produce en la paralelización de las operaciones de procesamiento de imágenes al introducir los filtros separables.	37
2.6	Esquema de alto nivel del funcionamiento del algoritmo RANSAC en su implementación paralela.	48
2.7	(a) Imagen de un ojo umbralizada mostrando la pupila en el centro y algunas pestañas, párpados y artefactos. (b) Imagen etiquetada con cada <i>blob</i> detectado en un color diferente. (c) Área etiquetada de mayor tamaño. (d) Se ha detectado el borde del área seleccionada.	53
2.8	(a) Se lanzan rayos en todas direcciones desde el centro (punto azul). (b) Se lanzan rayos desde los puntos encontrados en la primera etapa. (c) Se lanzan más rayos desde los puntos encontrados en la primera etapa. (d) Con todos los puntos encontrados se ejecuta RANSAC y se encuentra un círculo/pupila con centro en el punto azul. (e) Se repite el proceso desde el nuevo centro y algoritmo converge.	56
2.9	Diagrama de Gantt que representa la ejecución del procesamiento usando la técnica de pipelining mediante streams.	58
2.10	(a) Bordes detectados por el algoritmo de Canny. (b) Votos Obtenidos por la transformada de Hough. Se muestran en escala logarítmica para facilitar la visualización.	60
2.11	Resultados de precisión y rendimiento de varias configuraciones del algoritmo Starburst paralelo.	64
2.12	Resultados de precisión con dos tipos de ajuste pupilar (círculo o elipse) para el algoritmo Starburst paralelo. Y resultados de rendimiento de cada uno de esos ajustes utilizando una cantidad variable de <i>streams</i> CUDA . . .	67

2.13	Aceleración obtenida por la implementación CUDA usando 1 stream en comparación con la implementación CPU. Se muestra la aceleración obtenida para cada una de las optimizaciones aplicadas a la implementación GPU.	68
2.14	Resultados de precisión y de rendimiento para el algoritmo de umbralización y etiquetado para varias configuraciones.	69
2.15	Resultados de precisión y de rendimiento para el algoritmo de Canny y Hough para varias configuraciones.	70
2.16	Comparativa de los resultados de precisión y rendimiento de las mejores configuraciones de cada algoritmo.	71
2.17	Comparativa del tiempo que cada algoritmo emplea en cada una de las partes que lo componen.	72
2.18	(a) Captura de pantalla del programa utilizado para el experimento de agudeza visual automática. (b) Posición y tamaño de una pupila cuyo seguimiento se ha llevado a cabo en tiempo real con la implementación GPGPU desarrollada en esta Tesis.	74
3.1	Comparación de la visión que generan unas gafas bifocales (a la izquierda) y la que generan las Gafas de Corrección Automática de la Presbicia (a la derecha). Mientras que las bifocales no enfocan correctamente todas las distancias las Gafas de Corrección Automática de la Presbicia sí lo hacen. .	78
3.2	Sistema inicial con el prototipo sobre una mesa óptica.	88
3.3	Esquema del sistema utilizado para corregir la presbicia incluyendo un camino para medir las aberraciones utilizando un sensor Hartmann-Shack. ¹	90
3.4	(a) Imagen de dos ojos artificiales tomada con la cámara encargada del seguimiento pupilar a través de las primeras lentes utilizadas (Optotune EL-10-30-TC). (b) Imagen de dos ojos reales tomada con la cámara encargada del seguimiento pupilar a través de las segundas lentes utilizadas (Optotune EL-16-40-TC).	91
3.5	(a) Imagen del primer prototipo de las gafas para la corrección automática de la presbicia portátil. (b) Imagen del prototipo actual de las Gafas de Corrección Automática de la Presbicia portátil.	93
3.6	Imagen del prototipo de las gafas con todos los componentes montados. . .	94

3.7	(a) Captura de pantalla de la app de control con la corrección activada. (b) Captura de pantalla de las opciones de configuración de la app de control.	96
3.8	Diagrama de control de las Gafas de Corrección Automática de la Presbicia.	97
3.9	Información del ojo usada para calcular la dirección de mirada del sujeto. (x_c, y_c) es el centro del ojo, (x_f, y_f) es el centro de la pupila cuando el sujeto está mirando lejos, (x_n, y_n) es el centro de la pupila en la imagen cuando el sujeto está mirando cerca, (x_r, y_r) es el centro real de la pupila cuando el sujeto está mirando cerca.	98
3.10	Aparecen dos ojos mirando un estímulo cercano. La rotación de los ojos (α y β), que ha sido previamente calculada, se utiliza para calcular la distancia a la que se encuentra el estímulo.	99
3.11	Esquema que muestra los tiempos de respuesta de los diversos componentes del sistema bajo dos configuraciones para el control de las lentes.	101
3.12	Ejemplo de caso de uso de un sujeto mirando alternativamente lejos y cerca. La línea verde muestra el movimiento de las pupilas y la violeta las dioptrías aplicadas por las lentes.	101
3.13	Precisión de la medida del ángulo de giro de un ojo artificial, comparación entre los grados medidos y los esperados.	102
3.14	Comparación de rendimiento de los 5 dispositivos probados con 3 resoluciones diferentes.	104
3.15	Detalle del tiempo empleado por cada dispositivo para tres resoluciones en terminar cada una de las etapas del procesado. El tiempo de <i>Copia</i> es el tiempo que se tarda en copiar la imagen, si es necesario, así como los resultados a cualquier dato intermedio. El <i>Preprocesado</i> es el tiempo empleado en aplicar las funciones de procesamiento de imágenes. Mientras que <i>Puntos</i> es el tiempo que se tarda en buscar los puntos en el borde de la pupila. Y <i>Ajuste</i> hace referencia al tiempo empleado en ajustar una elipse a los puntos y seleccionar la mejor.	105
3.16	Comparación del consumo energético de cada dispositivo para procesar una imagen con tres resoluciones diferentes.	107
3.17	Test de agudeza visual de cerca utilizado durante las pruebas con sujetos. .	108
3.18	Agudeza visual de los sujetos antes y después de activar la corrección de la presbicia.	109

-
- 3.19 Agudeza visual medida para 8 sujetos usando 3 correcciones diferentes y a 6 distancias. En violeta se muestra la agudeza visual medida utilizando la corrección para lejos, mientras que en verde se muestran los resultados obtenidos con la corrección para cerca (2.5D), finalmente en azul se muestra agudeza utilizando las gafas auto-acomodativas. La agudeza visual se muestra en el eje y en escala decimal, y en el eje x se presentan las 6 distancias a las que se ha hecho el test. 110
- 4.1 Imagen tomada con un sensor de frentes de onda de Hartmann-Shack. Se muestran los dos ojos de un sujeto fusionados utilizando un prisma. . . . 116
- 4.2 (a) Imagen original sobre la que se realiza el seguimiento. (b) Se aplica un filtro de Gauss 2D a la imagen. (c) Se umbraliza la imagen utilizando un algoritmo automático para determinar el umbral. (d) Se aplica una operación morfológica de cierre sobre la imagen. (e) Se detectan los bordes en la imagen. (f) Se ajusta una elipse a los bordes detectados en la imagen. 119
- 4.3 (a) Ventana inicial en la que se realiza la búsqueda delimitada por el cuadrado rojo y centro geométrico señalado por el punto rojo. (b) Ventana de 22x22 píxeles con el centro de masas señalado por el punto rojo. (c) Ventana de 20x20 píxeles con el centro de masas señalado por el punto rojo. (d) Ventana de 16x16 píxeles con el centro de masas señalado por el punto rojo. (e) Ventana de 10x10 píxeles con el centro de masas señalado por el punto rojo. (f) Ventana de 3x3 píxeles con el centro de masas señalado por el punto rojo. 122
- 4.4 (a) Comparación de dos imágenes de Hartmann-Shack tomadas con el mismo sistema. En rojo se muestra la imagen de referencia, mientras que en azul se muestra la imagen del ojo de un sujeto. Se puede apreciar el desplazamiento generado entre los puntos de distinto color. (b) Zona central ampliada en la que se aprecia el desplazamiento de los centroides entre la imagen de referencia (en rojo) y la imagen del ojo de un sujeto (en azul). . 123
-

4.5	(a) Distribución de las microlentes de acuerdo a la distribución de las mismas en la imagen de referencia. Los cuadrados azules delimitan el área de cada microlente, los puntos verdes señalan los centroides que están dentro de las microlentes, y los puntos rojos los que están fuera. (b) Distribución de las áreas desplazadas una vez se ha ejecutado el algoritmo de malla. En azul se delimitan las áreas a analizar en la imagen procesada, los puntos morados indican el centro del centroide en la imagen de referencia y los puntos verdes indican el centroide de los puntos en la imagen procesada.	126
4.6	Esquema que muestra cómo se lleva a cabo el cómputo del mapa del frente de onda de la aberración (mapa de la derecha) a partir de los mapas de cada aberración concreta (mapas de la izquierda) y los coeficientes de Zernike (en el centro).	137
4.7	Aceleración de la implementación paralela sobre la secuencial de la etapa de búsqueda de los centroides y del total del proceso en función del número de puntos procesados (tamaño de la imagen fijo a 2560x2560 píxeles). . . .	142
4.8	Aceleración de la implementación paralela sobre la secuencial de la etapa de búsqueda de los centroides y del total del proceso en función del tamaño de la imagen (número de microlentes fijo a 250).	143
4.9	Tiempo de ejecución en la GPU desglosado por etapas en función del número de puntos procesados (tamaño de la imagen fijo a 2560x2560 píxeles). <i>P</i> para <i>PaPyCS</i> , <i>R</i> para Reichenbach y <i>C</i> para Centro de Masa	144
4.10	Tiempo de ejecución en la CPU desglosado por etapas en función del número de puntos procesados (tamaño de la imagen fijo a 2560x2560 píxeles). <i>P</i> para <i>PaPyCS</i> , <i>R</i> para Reichenbach y <i>C</i> para Centro de Masa	145
4.11	Tiempo de ejecución en la GPU desglosado por etapas en función del tamaño de la imagen (número de microlentes fijo a 250). <i>P</i> para <i>PAPYCS</i> , <i>R</i> para Reichenbach y <i>C</i> para Centro de Masa	146
4.12	Tiempo de ejecución en la CPU desglosado por etapas en función del tamaño de la imagen (número de microlentes fijo a 250). <i>P</i> para <i>PAPYCS</i> , <i>R</i> para Reichenbach y <i>C</i> para Centro de Masa	146
4.13	Aceleración del cómputo del mapa de frente de onda en función del tamaño de imagen para varias cantidades de coeficientes de Zernike.	147
4.14	Velocidad en imágenes/s del cómputo del mapa de frente de onda en función del tamaño de imagen para varias cantidades de coeficientes de Zernike.	148

4.15	Comparación del RMS de la aberración medida de los dos métodos y el teórico. La línea gris muestra el valor teórico.	149
4.16	Las barras (eje Y izquierdo) muestran el porcentaje de imágenes con un RMSE más pequeño que 1. Las líneas (eje Y derecho) muestran el RMSE medio para las mismas imágenes.	149
4.17	Las barras (eje y izquierdo) muestra el número de centroides detectados con un error menor que 1 píxel. Las líneas (eje y derecho) muestra el error medio en píxeles.	150
4.18	Desenfoque medido (en micras) en una serie de imágenes de un sistema H-S real en condiciones estables.	151
4.19	(a) Sistema binocular Hartmann-Shack que permite la visión directa de estímulos. (b) Programa de control del sistema de Hartmann-Shack mostrado en (a).	153
4.20	Medida del desenfoque de un sujeto que mira alternativamente a un estímulo en cerca y lejos.	154
5.1	Ejemplo de imagen frontal de un ojo sano generada a partir de un volumen del mismo.	156
5.2	Imagen real y esquema del sistema OCT instalado en el LOUM [84].	157
5.3	(a) Captura de pantalla de la ejecución del procesamiento de datos de OCT visualizada en el NVIDIA Visual Profiler en la que se aprecia el solapamiento de las copias de memoria y la ejecución de los kernels. (b) Captura de pantalla de la ejecución del procesamiento de datos de OCT visualizada en el NVIDIA Visual Profiler en la que se muestra la ejecución de las diferentes tareas en diferentes streams.	167
5.4	Tiempo de ejecución de las distintas operaciones que se llevan a cabo para el procesamiento de los datos de OCT. Los resultados de MATLAB y C++ se han obtenido en un procesador AMD 1800x, mientras que los resultados de CUDA y OpenCL se han obtenido en una GPU NVIDIA 1080 Ti. Los tiempos de la implementación C++ son una estimación debido a limitaciones de la medida. Lo mismo ocurren con los tiempos de la implementación CUDA.	170
5.5	Captura de pantalla del programa que se utiliza para procesar los datos de OCT.	171

Índice de tablas

2.1	Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento usando las NPP.	33
2.2	Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación especializada de los mismos.	34
2.3	Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación con kernels separables.	38
2.4	Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la fase de preprocesamiento para la implementación con memoria compartida.	39
2.5	Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación con kernels fusionados.	41
2.6	Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación con desenrollado de bucles.	43
2.7	Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación con uint8.	44
2.8	Tabla que muestra los tiempos de ejecución del algoritmo RANSAC con las distintas optimizaciones aplicadas.	50

3.1	Tiempos de ejecución para las etapas de preprocesamiento y de búsqueda de puntos del algoritmo de seguimiento pupilar. Dichos tiempos de ejecución y la aceleración obtenida al implementar varias optimizaciones se muestran tanto para la placa Zedboard como para la más potente Alveo u250.	87
3.2	Equivalencias entre el tamaño de la fuente Times New Roman y la escala de agudezas visuales utilizadas para generar el test de agudeza visual de cerca.	107
4.1	Tabla que muestra algunos resultados de rendimiento de varias implementaciones de un kernel utilizado durante el cálculo de los mapas de frente de onda en la GPU.	138
5.1	Rendimiento obtenido en diferentes trabajos procesando datos obtenidos en sistemas OCT utilizando diferentes configuraciones y hardware. Si bien el procesado llevado a cabo no es el mismo en todos los casos.	159
5.2	Tiempo de ejecución de las versiones secuenciales y paralelas del procesado de datos OCT en LabVIEW para dos resoluciones.	162
5.3	Tiempo de ejecución del procesado de datos OCT en MATLAB para dos resoluciones.	163
5.4	Tiempo de ejecución del procesado de datos OCT en C++ para varias configuraciones y dos resoluciones. Tiempos obtenidos en un procesador AMD Ryzen 7 1800X.	164
5.5	Tiempo de ejecución del procesado de datos OCT en CUDA para varias configuraciones y dos resoluciones.	166
5.6	Tiempo de ejecución del procesado de datos OCT en OpenCL para varias configuraciones y dos resoluciones.	169

Índice de Códigos

2.1	Código CUDA de la implementación especializada de un filtro de erosión. .	34
2.2	Código CUDA de la implementación con filtros separables del kernel de erosión.	38
2.3	Código CUDA de la implementación con memoria compartida del filtro de erosión.	40
2.4	Código CUDA implementando la optimización de desenrollado de bucles en el filtro de erosión.	42
3.1	Ejemplo de particionado de arrays	86

1 Introducción

1.1 Computación de propósito general en unidades de procesamiento gráfico

El término Computación de propósito general en unidades de procesamiento gráfico (General-Purpose computing on Graphics Processing Units, GPGPU por sus siglas en inglés) fue acuñado por Mark Harris en 2002 [1]. Sin embargo, la primera implementación GPGPU de la que se tiene constancia data de 1978 [2]. Aunque no fue hasta principios de los 2000 cuando el hardware tuvo la capacidad suficiente como para ser una herramienta realmente atractiva para realizar computación general o GPGPU. Y no sería hasta que el software también ofreció un acceso más sencillo a esos recursos cuando finalmente se convirtió en una herramienta accesible a una mayor cantidad de usuarios. El lanzamiento de CUDA [3] en 2007, y el de OpenCL [4] en 2008 marcaron un punto de inflexión en el desarrollo de esta tecnología. Desde entonces, la cantidad de aplicaciones que hacen uso de GPGPU para acelerar su velocidad no hace sino incrementarse. Hoy en día, son muchos los centros de supercomputación en los que una gran cantidad de cómputo se ejecuta en procesadores gráficos. Asimismo, multitud de programas son capaces de aprovechar las GPUs (integradas o dedicadas) para reducir el tiempo de ejecución de algunas de sus operaciones, por ejemplo, Adobe Photoshop.

Las características que hacen de las GPUs dispositivos tan atractivos para la computación de alto rendimiento son principalmente dos: una gran capacidad de cómputo, muy superior a la de otros dispositivos; y un gran ancho de banda. Normalmente, los aceleradores se comparan con las CPUs porque es a lo que suelen sustituir. En este caso, si se compara con una CPU, las dos características principales de las GPUs muestran una clara superioridad con respecto a las CPUs. Una CPU AMD EPYC de segunda generación (introducida en 2019) tiene una capacidad de cálculo de más de 4 TFLOPS en precisión simple, aunque varía en función del modelo. Mientras que una GPU AMD Radeon VII

(también introducida en 2019) alcanza los 13.8 TFLOPS. Asimismo, el ancho de banda de la memoria de dicho procesador gráfico es de 1TB/s gracias al uso de memoria HBM (High Bandwidth Memory), el cuál es muy alto en comparación con los 200GB/s que alcanzan las CPU AMD EPYC de segunda generación.

Sin embargo, para alcanzar esas cotas de rendimiento las GPUs se configuran de una forma muy diferente a una CPU. Si en una CPU se busca aumentar al máximo la velocidad de cada núcleo, en las GPUs se suele preferir aumentar la cantidad de núcleos (o elementos de procesamiento). Si bien, evidentemente la velocidad de cada uno de esos núcleos también es importante. Asimismo, las CPUs también han ido incrementando la cantidad de núcleos disponibles, aunque se podría decir que hasta recientemente contaban con cantidades “modestas” de núcleos. Pero con la llegada de las CPUs basadas en la nueva arquitectura Zen desarrollada por AMD, y de algunos procesadores basados en núcleos ARM, la cantidad de núcleos presentes en las CPUs está aumentando drásticamente. Los procesadores EPYC de AMD cuentan con hasta 64 núcleos físicos, la misma cantidad que las CPUs Graviton2 desarrolladas por Annapurna Labs (Amazon). Además, Marvell ha anunciado que este año lanzará sus CPUs ThunderX3 con 96 núcleos y 384 hilos. A pesar de la mayor cantidad de núcleos, las CPUs siguen manteniendo las mismas funcionalidades que las diferencian de las GPUs, como la coherencia de la memoria, que no está disponible en todos los casos en una GPU. Y una sobrecarga mucho mayor en la gestión de los hilos. Así, las GPUs modernas cuentan con miles de procesadores que pueden ejecutar el mismo kernel con millones de hilos, de hecho, es recomendable utilizar una cantidad de hilos muy superior a la cantidad de núcleos presentes en el hardware para ocultar las latencias de la memoria y mejorar el rendimiento. Sin embargo, en una CPU lo óptimo suele ser utilizar tantos hilos software como físicos haya en el procesador.

De acuerdo a la clasificación introducida por *Flynn* [5], las GPUs son dispositivos *SIMD* (*Single Instruction stream, Multiple Data streams*), esto es, el paralelismo de los mismos se basa en ejecutar una misma instrucción con múltiples datos. Por contraposición, el modelo de las CPUs modernas es el *MIMD* (*Multiple Instruction streams, Multiple Data streams*), donde tenemos varias núcleos ejecutando instrucciones diferentes sobre datos diferentes. Si bien, la mayoría de CPUs modernas incorporan instrucciones vectoriales que sí se clasificarían como *SIMD*, aunque la denominación general de dichos procesadores sigue siendo *MIMD*. La arquitectura *SIMD* puede, potencialmente, obtener un rendimiento mayor que la arquitectura *MIMD* debido al mayor paralelismo que potencialmente se puede conseguir. Sin embargo, ese mayor rendimiento solo se obtiene en determinados

algoritmos (ej., los comúnmente denominados masivamente paralelos) y con una correcta programación.

El paradigma de programación de una GPU es, por tanto, muy diferente al paradigma clásico utilizado en una CPU. La granularidad utilizada es muy pequeña, siendo el trabajo desarrollado por cada hilo generalmente muy escaso. En ocasiones, aunque no suele ser lo óptimo, un hilo tan solo realiza una carga de memoria, una operación aritmética y un almacenamiento a memoria. Evidentemente, para que ese modelo funcione la sobrecarga de lanzar un hilo debe ser prácticamente nula. Además, se deben lanzar muchos hilos para poder acometer todo el trabajo. A pesar de la gran cantidad de recursos disponibles en una GPU estos siguen siendo limitados y es por ello que los hilos se agrupan en dos niveles lógicos. La figura 1.1 muestra la organización de los mismos tal y como la describe NVIDIA [6]. El más bajo es el bloque (por la terminología en CUDA) o grupo de trabajo (en OpenCL). Los bloques pueden contener una cantidad variable de hilos, si bien la cantidad óptima varía según la plataforma. Las GPUs de sobremesa suelen soportar un máximo de 1024. Aunque si se trata de un kernel complejo puede ser necesario reducir el número de hilos. Los hilos de un bloque se organizan en tres dimensiones x, y, z que pueden configurarse para adaptarse a la lógica del algoritmo. El segundo nivel lógico de organización de los hilos es el grid (en CUDA) o NDRange (en OpenCL), el cuál también se organiza en tres dimensiones (x, y, z) a semejanza de los bloques, generando una visión homogénea de la distribución global de los hilos. Un Grid se compone de tantos bloques de hilos como se indique en su configuración. Y en lo que se refiere a las limitaciones de cantidad suelen ser más grandes que en los bloques, aunque también varían bastante de una plataforma a otra.

Esa es la división lógica expuesta al programador que ofrecen CUDA y OpenCL, pero internamente la granularidad de ejecución es menor, organizándose generalmente la ejecución en grupos de 32 o 64 hilos (*warps* en CUDA, *wavefront* en GPUs AMD). Si bien, cada hilo puede ejecutarse de manera independiente y seguir caminos de ejecución totalmente distintos del resto de su *warp*, ese comportamiento reduce drásticamente el rendimiento. Por lo tanto, durante la optimización de la implementación GPGPU es importante tener en cuenta esta granularidad si el algoritmo requiere la ejecución condicional de alguno de sus cálculos.

De esta forma, para hacer uso de esa gran cantidad de hilos es necesario modificar la forma en la que se programan los algoritmos. Si en una paralelización clásica en CPU se reparte el trabajo entre los hilos al más alto nivel posible, normalmente se asigna una

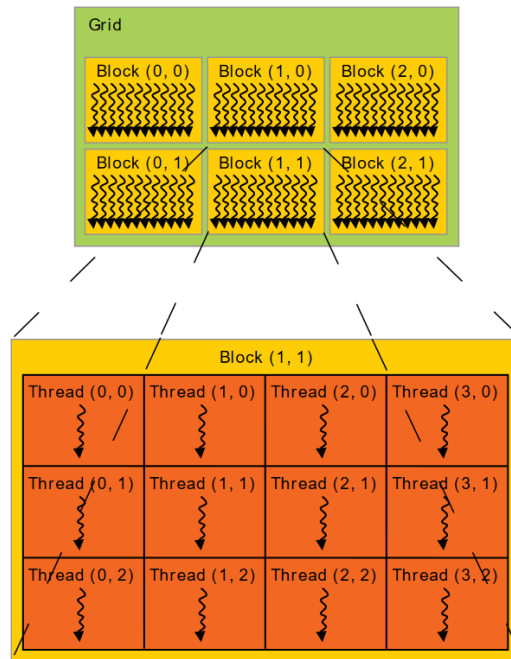


Figura 1.1: Organización lógica de los hilos en CUDA. Fuente NVIDIA [6].

cantidad variable de iteraciones de un bucle a cada hilo, en una GPU se asigna a cada hilo una o unas pocas iteraciones del bucle más interior del algoritmo. De forma que en la paralelización GPU en muchas ocasiones se desenrollan totalmente los bucles y se prescinde de ellos por completo. Por supuesto, para llevar a cabo dicha paralelización no debe haber dependencias entre las iteraciones de los bucles. Si las hubiera, se deberían explorar otras soluciones para ver si es posible eliminar esas dependencias. Si bien, algunos algoritmos se adaptan muy bien a la arquitectura de las GPUs, como muchas funciones de procesamiento de imágenes o de álgebra lineal, y de hecho, su programación es en ocasiones más sencilla en su versión paralela que en su versión secuencial. En otras ocasiones, la paralelización con un grano tan fino es muy compleja y requiere modificaciones de gran calado respecto a las implementaciones secuenciales, como por ejemplo, para llevar a cabo una ordenación de un vector.

Las principales diferencias entre un modelo secuencial de programación y la programación paralela en GPUs suelen ser la sincronización del trabajo y los datos entre los hilos, así como la distribución del trabajo. Es muy importante que la distribución del trabajo sea homogénea para evitar que ciertos hilos tengan que hacer mucho más trabajo que el resto, puesto que esto resultaría en un bajo aprovechamiento de los recursos. El correcto

balanceo y distribución del trabajo también incluye la correcta asignación de los datos que debe procesar cada hilo, siendo crucial el comprobar que ningún hilo accede a datos fuera de la memoria reservada. Debido a que los hilos se lanzan en bloques, en muchas ocasiones es imposible lanzar una cantidad de hilos exactamente igual a la cantidad de datos que se van acceder. En ese caso la aproximación más común es lanzar más hilos que datos y hacer que cada hilo compruebe si se encuentra en la zona válida o fuera de ella. Otra aproximación es lanzar menos hilos que datos a procesar e iterar a través de ellos, pero esto sigue generando el mismo problema al no ser la cantidad de hilos divisor de la cantidad de datos. La sincronización del trabajo y los datos entre los hilos es muy compleja en las GPUs, puesto que es posible sincronizar los hilos y los datos dentro de un bloque pero no entre bloques. Además, las sincronizaciones suelen tener un coste elevado y, por tanto, su uso debe ser lo más reducido posible.

A pesar de la gran potencia de cómputo de las GPUs una paralelización pobre que no tenga en cuenta las particularidades de la arquitectura subyacente puede resultar en un rendimiento peor que la propia versión secuencial. Incluso en ocasiones una paralelización correcta pero que no aprovecha todas las posibilidades de una GPU, como memoria compartida o instrucciones especiales, puede rendir peor que la versión secuencial.

Esta Tesis se ha centrado en el uso del paralelismo masivo y de las GPUs para aumentar la velocidad de ejecución de aplicaciones que requieren funcionar en tiempo real pero que, debido a la enorme carga de trabajo que conllevan, no pueden conseguirlo en una CPU. Por otro lado, en una de las propuestas que se detalla en el capítulo 5 hemos evaluado el uso de aceleradores específicos, como una FPGA. Las FPGAs son una alternativa a las GPUs para llevar a cabo procesamiento masivamente paralelos. Aunque cada una cuenta con unas fortalezas y debilidades, que las hacen más o menos eficientes según el problema que se trate de solucionar con ellas. También hay que tener en cuenta que los algoritmos utilizados por estas aplicaciones presentan ciertas características que permiten que una aproximación masivamente paralela consiga acelerar su ejecución. De otra forma, los resultados obtenidos por las implementaciones ejecutados sobre los procesadores gráficos no habrían obtenido unas mejoras tan significativas respecto a las implementaciones secuenciales. Aunque en algunos casos ha sido necesario modificar la aproximación secuencial para obtener un buen rendimiento con la aproximación paralela.

1.2 Seguimiento pupilar en el ojo humano

El seguimiento pupilar del ojo humano ha sido una de las aplicaciones aceleradas mediante el uso de GPUs. El seguimiento pupilar consiste en la localización a lo largo del tiempo de la posición de la pupila y la medición de sus características (tamaño, posición, inclinación) con precisión. Para ello se pueden utilizar cámaras de diferentes tipos y prestaciones, si bien lo más habitual es utilizar cámaras sensibles a la luz infrarroja. Esto es debido a que en la mayoría de ojos la iluminación infrarroja suele ofrecer un buen contraste entre la pupila y el iris. Asimismo, los algoritmos utilizados están optimizados para el seguimiento en imágenes tomadas con ese tipo de luz. La figura 1.2 muestra un ejemplo de ojos humanos cuando se iluminan con luz infrarroja. Se puede ver el gran contraste que se genera entre la pupila y el iris. Aunque ese contraste varía de una persona a otra dependiendo de su color de iris.

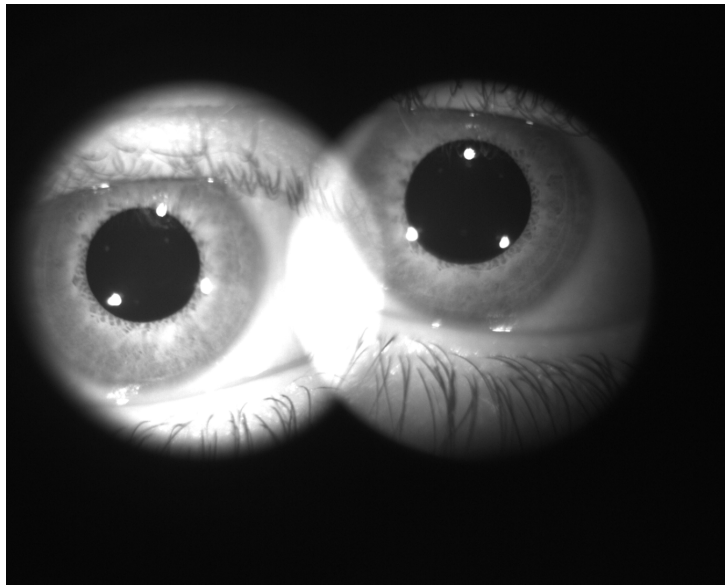


Figura 1.2: Ejemplo de imagen de unos ojos iluminados con luz infrarroja.

El seguimiento pupilar es una herramienta muy útil en multitud de escenarios. Se utiliza en marketing, usabilidad, videojuegos, apoyo a personas con limitaciones de movilidad, etc. En investigación el seguimiento pupilar tiene interesantes aplicaciones. Por ejemplo, el estudio de los *sacádicos*, movimientos del ojo que se producen a muy alta velocidad. El seguimiento de la pupila también puede utilizarse para determinar de una forma objetiva la agudeza visual de una persona [7]. Otras aplicaciones incluyen, determinar tiempos

de reacción ante diferentes tipos de estímulos, o en un sistema de óptica adaptativa con un bucle cerrado desplazar la posición de la corrección siguiendo los movimientos de la pupila.

Si bien los algoritmos públicamente disponibles suelen basarse en el uso de imágenes de infrarrojo aunque hay algunos que utilizan imágenes a color. Las soluciones propuestas suelen tener muchos puntos en común dado que su objetivo es el mismo. Así, algunas se basan en la umbralización, tratando de aislar a la pupila del iris por su diferencia de intensidad. Otros algoritmos utilizan funciones de detección de bordes, las cuáles suelen encontrar con bastante éxito el borde de la pupila, pero su complejidad es mayor. El objetivo de todos ellos es encontrar la posición y tamaño de la pupila y para ello la mayoría optan por encontrar primero los puntos del borde de la pupila y realizar un ajuste a los mismos posteriormente. Aunque algunos algoritmos buscan directamente círculos como al aplicar una transformada de Hough circular para la búsqueda de la pupila.

Cuando se trata de buscar la pupila es fundamental tener en cuenta que las pupilas humanas son elípticas, si bien se aproximan a círculos en media. En ciertas ocasiones también pueden presentar formas irregulares. La mayoría de los algoritmos de seguimiento de pupilas utilizan ajustes circulares o elípticos. Los ajustes elípticos dan un mejor resultado en la mayoría de los casos ofreciendo mayor precisión y estabilidad que los circulares por lo que, a pesar de su mayor coste computacional, suelen ser los preferidos.

Si bien el seguimiento de la pupila también presenta muchos retos para cumplir su tarea con éxito, los algoritmos utilizados para ello plantean nuevos retos computacionales puesto que en muchos casos conllevan una carga computacional muy elevada que se traslada en altos tiempos de ejecución. Esto supone un problema para las aplicaciones que, por sus características, requieren tener información de la pupila en tiempo real. O bien supone largos tiempos de espera en los que el procesado se lleva a cabo a posteriori debido a la gran cantidad de información que se procesa ya que por regla general las imágenes son de alta calidad y las operaciones de procesamiento de imágenes que se utilizan tienen tiempos de ejecución elevados.

Una buena iluminación del ojo es necesaria para la localización precisa de la pupila, así, es importante usar una fuente de luz suficientemente luminosa y homogénea. Las fuentes de luz utilizadas para conseguir esos objetivos pueden llegar a ser muy voluminosas cuando se intenta hacer seguimiento de la pupila a muy alta velocidad, pues se reduce el tiempo de exposición y es necesario captar suficiente luz para formar una imagen sin ruido. Otra cuestión relevante a tener en cuenta es la localización de la fuente de iluminación, para

evitar el efecto “ojos rojos”. Además, suele ser conveniente evitar localizaciones frontales al ojo y es preferible una iluminación más oblicua para desplazar los reflejos corneales fuera de la pupila. Los reflejos corneales se presentan como un punto blanco, que evidentemente de encontrarse dentro de la pupila, dificulta el proceso de localización de la misma.

La captura de las imágenes del ojo se puede llevar a cabo mediante cámaras muy diversas en función del objetivo que se persiga. En esta Tesis se han utilizado principalmente cámaras de alta velocidad que nos permiten obtener mucha información del ojo. La cual se puede utilizar para analizar movimientos oculares de alta frecuencia, tales como los sacádicos o para reducir el ruido de las medidas suavizándolas gracias a toda la información disponible. Por otra parte, cuando hablamos de las cámaras no podemos olvidar el campo de visión, que determina el área que se va a monitorizar. Se trata de un factor clave a la hora de establecer la precisión y rango del sistema: si se utiliza un área demasiado pequeña la precisión del sistema será mayor debido a que habrá más píxeles en la zona del ojo. Mientras que si se utiliza un área relativamente grande la precisión bajará. Por otro lado, un área muy pequeña limita en gran medida el rango de movimientos del ojo, así como las diferencias naturales entre distintos sujetos. Encontrar el equilibrio entre precisión y rango de movimiento/alineación suele ser fundamental para que estos sistemas de seguimiento pupilar puedan usarse eficientemente. También se han utilizado cámaras de prestaciones más modestas, con una menor velocidad de captura pero de un tamaño muy reducido. Esas cámaras han sido utilizadas en las Gafas de corrección automática de la Presbicia en las cuales era imprescindible dar con una solución de reducidas dimensiones.

1.3 Presbicia y su corrección

La presbicia es la pérdida de capacidad del ojo humano para enfocar objetos asociada al envejecimiento. Si bien la pérdida de rango de acomodación se produce desde edades muy tempranas, sus efectos no empiezan a ser patentes hasta los 40 años, momento en el cual algunas personas empiezan a tener dificultades para enfocar objetos cercanos y para leer textos de cerca. Esta degradación la sufren todas las personas sin excepción y con una progresión bastante similar en líneas generales. Es por tanto un problema muy generalizado y con grandes implicaciones debido a las limitaciones que impone a los que lo padecen. La figura 1.3 muestra cómo la capacidad de acomodación del ojo humano va disminuyendo con la edad hasta desaparecer casi totalmente [8].

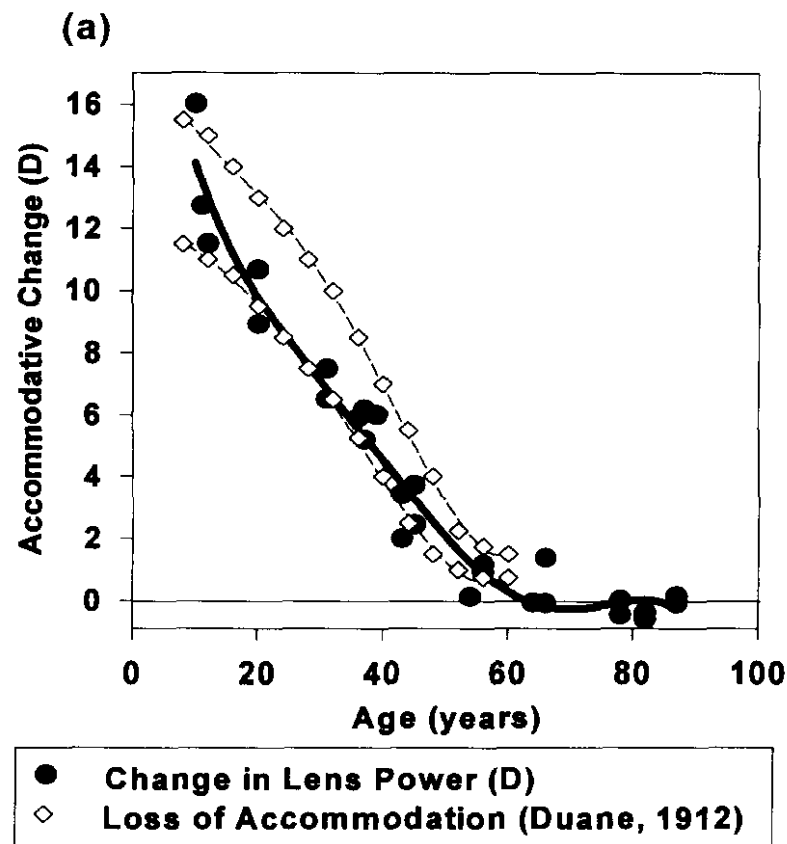


Figura 1.3: Rango de acomodación en función de la edad [8].

Por suerte, contamos con multitud de soluciones para combatir los efectos de la presbicia. El más simple, pero no menos efectivo son las gafas de cerca. Éstas son ampliamente utilizadas para leer textos, como libros, periódicos o ver el móvil. Además, en el mercado se encuentran multitud de opciones adicionales, desde gafas multifocales hasta lentes intraoculares. Sin embargo, todas esas soluciones son estáticas y, por tanto, no devuelven al ojo su funcionalidad original. Lo cual sí es posible lograr con las Gafas de corrección automática de la Presbicia que hemos desarrollado, que más allá de su tamaño o autonomía impuestas por las limitaciones técnicas, son capaces de proporcionar la misma experiencia visual que generaría un ojo sin presbicia.

Sin embargo, el desarrollo de unas gafas autónomas, portátiles y automáticas capaces de corregir la presbicia en tiempo real presenta numerosos retos. El funcionamiento de las gafas debe ser completamente autónomo para proporcionar una buena experiencia a los

usuarios, no debe requerir ningún tipo de interacción consciente. Para lograrlo es necesario realizar el seguimiento de las pupilas del ojo, lo cuál tiene un alto coste computacional. Por tanto, se necesita un dispositivo suficientemente potente como para realizar el seguimiento, pero a la vez este debe ser portátil y estar alimentado por una batería. Además, para realizar el seguimiento pupilar es necesario incluir dos cámaras para capturar las imágenes de los ojos. También es necesario añadir iluminación infrarroja y baterías para alimentarla. Todo este hardware adicional añade un peso y volumen significativo al sistema, lo cual hace más complejo el desarrollo de un soporte que lo integre todo correctamente y a la vez sea suficientemente cómodo para los usuarios. Por otra parte, conseguir una experiencia transparente para el usuario que haga que se olvide de que lleva unas gafas optoelectrónicas y simplemente se concentre en lo que está mirando no es sencillo. Requiere tiempos de reacción rápidos y un funcionamiento suave sin cambios abruptos en la potencia aplicada por las lentes. En definitiva, el desarrollo de unas Gafas de corrección automática de la Presbicia es un reto con muchas ramificaciones que deben ser debidamente consideradas para obtener el equilibrio adecuado entre las prestaciones que queremos obtener y las contraprestaciones que algunas de las soluciones adoptadas pueden añadir.

El capítulo 3 comprende todo lo relacionado con las Gafas de corrección automática de la Presbicia. En él se expone detalladamente la problemática que ha motivado este trabajo y los algoritmos implementados y evaluados. De la misma manera, se presentan los resultados obtenidos, incluyendo las pruebas realizadas con sujetos présbitas.

1.4 Sensor Hartmann-Shack para medir las aberraciones del ojo humano en tiempo real

Antes de hablar sobre el sensor de frentes de onda Hartmann-Shack es necesario entender qué son los frentes de onda y las aberraciones ópticas. Se denomina frente de onda al lugar geométrico que ocupan los puntos de una onda en un instante concreto. Las aberraciones ópticas se definen como la diferencia entre el frente de onda perfecto (plano o esférico) para un sistema óptico ideal y el frente de onda deformado de un sistema óptico real. Las aberraciones ópticas (desenfoque, astigmatismo, coma, etc.) causan desviaciones en los rayos de luz, de forma que en lugar de converger en un solo punto convergen en un área mayor generando una imagen desenfocada. La medición de frentes de onda, o lo que es lo mismo, la medición de aberraciones ópticas se lleva a cabo de forma rutinaria

en muchos campos (astronomía, microscopía, comunicaciones) para diferentes finalidades (medición de la calidad óptica, calibración de instrumento o diseño óptico) [9]. Además, la medición de frentes de onda es fundamental en los sistemas de óptica adaptativa (AO), ampliamente utilizados en astronomía y ciencias de la visión. La óptica adaptativa intenta corregir dinámicamente las aberraciones que fluctúan en un sistema en *tiempo real*, utilizando para ello un sistema de corrección (espejo deformable o un modulador de cristal líquido) cuya forma o su distribución del índice de refracción se pueda modificar punto a punto para modificar el frente de onda [10]. Debido a la dinámica de los frentes de onda, un sistema de óptica requiere realizar mediciones de los frentes de onda a muy alta velocidad, de forma que se trata de un proceso con altos requerimientos de procesamiento. Teniendo en cuenta que la medición de frentes de onda conlleva procesamientos de imágenes muy costosos en lo que a tiempo de ejecución se refiere y que el procesamiento de imágenes es por regla general una tarea intrínsecamente paralela, la Computación de propósito general en unidades de procesamiento gráfico (GPGPU por sus siglas en inglés) es un candidato perfecto para obtener altas velocidades de procesamiento.

El sistema más común para describir frentes de onda en sistemas ópticos son los polinomios de Zernike [11], los cuáles son ortogonales entre sí en el disco unidad. Las aberraciones que se presentan más comúnmente en el ojo humano son las representadas por polinomios de órdenes más bajos. Concretamente, el desenfoque, el astigmatismo y el coma son aberraciones muy comunes en humanos, así, la figura 1.4 muestra los mapas de los frentes de onda de los polinomios de Zernike que representan a cada una de dichas aberraciones.

Las aberraciones ópticas se utilizan para representar las imperfecciones del ojo humano, siendo las más comunes la miopía, el astigmatismo o la hipermetropía. Dada la prevalencia de estas imperfecciones en la población, más incluso en los últimos años con el notable incremento de la miopía [12], es muy importante ser capaces de medirlas con precisión. Para ello disponemos de diversos métodos entre los que se encuentra el sensor Hartmann-Shack que permite realizar una medida objetiva y de alta precisión de las aberraciones del ojo.

Los ojos humanos presentan con bastante asiduidad otros problemas que empeoran la calidad de la visión (cataratas, sequedad, etc.), aunque la imperfección más común son las aberraciones y acompañan a la mayoría de las personas durante toda su vida. Es por tanto muy importante medir con precisión las aberraciones del ojo humano, y para ello surge la aberrometría óptica, que ha tenido mucha importancia en el mundo de la óptica por razones obvias. Durante las últimas dos décadas, las aplicaciones de la oftalmología

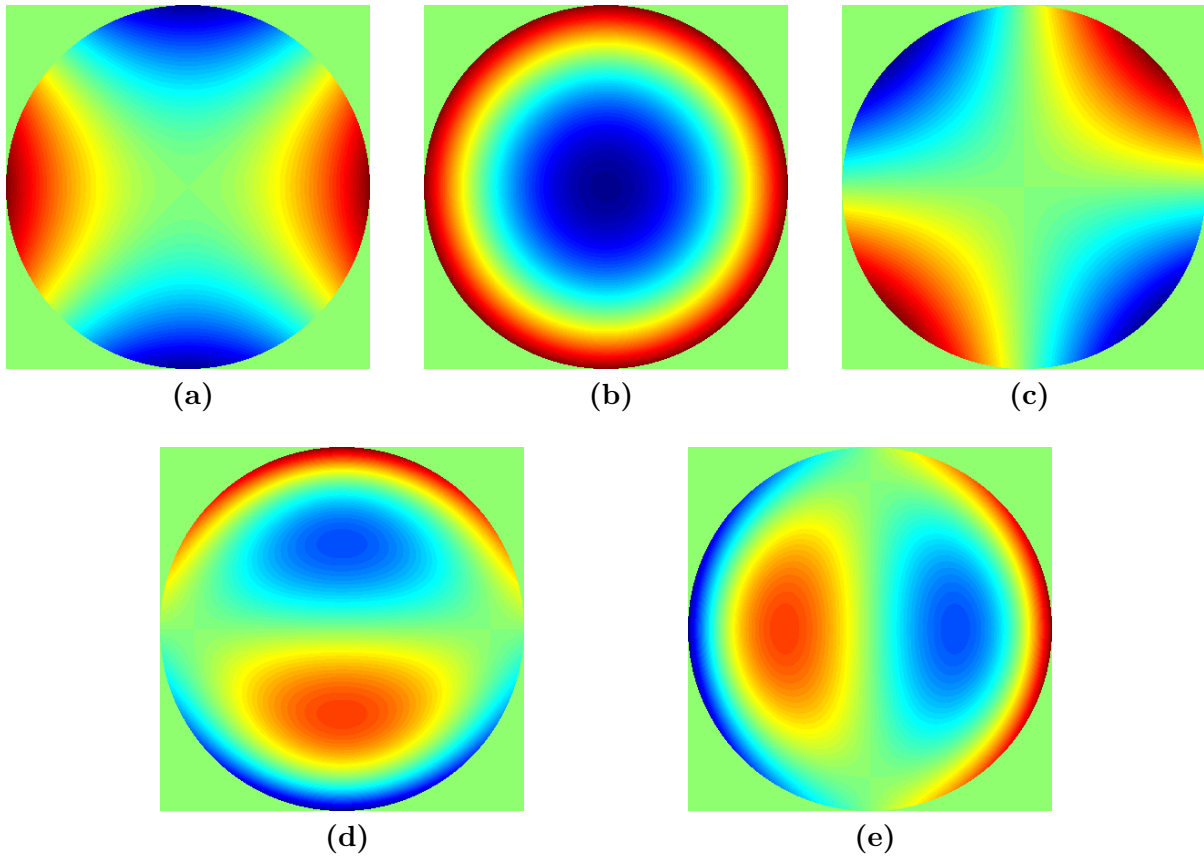


Figura 1.4: (a) Frente de onda aberrado mediante astigmatismo vertical (6º coeficiente de Zernike). (b) Frente de onda aberrado mediante desenfoque (4º coeficiente de Zernike). (c) Frente de onda aberrado mediante astigmatismo oblicuo (5º coeficiente de Zernike). (d) Frente de onda aberrado mediante coma vertical (7º coeficiente de Zernike). (e) Frente de onda aberrado mediante coma horizontal (8º coeficiente de Zernike).

y las ciencias de la visión han servido de catalizador para potenciar el desarrollo de sensores de frente de onda [13]. En el mercado se pueden adquirir aberrómetros para uso clínico, estando la mayoría basados en el principio de Hartmann-Shack, además de muchos otros que se utilizan en investigación y prototipado, como el desarrollado por *Emmanuel Chirre et al.* [14] que destaca por su binocularidad y por permitir la visión directa de los estímulos. Ese sistema es el que se ha utilizado durante el desarrollo de la paralelización de los algoritmos en la GPU para llevar a cabo las pruebas y el primero que utilizó la versión final para poder procesar los datos en tiempo real. Anteriormente, los datos obtenidos por este sistema eran procesados a posteriori debido a que la velocidad de procesamiento no permitía el procesamiento en tiempo real. Además, el tiempo de ejecución del procesamiento era muy elevado.

La aplicación más inmediata de la medición de aberraciones oculares es el diagnóstico y prescripción de la corrección (o refracción) del sujeto, pero también se utilizan en los sistemas de óptica adaptativa para aplicaciones oculares [15, 16]. La óptica adaptativa consiste en la modificación de los frentes de onda para, por regla general, compensar las aberraciones presentes en el sistema. Aunque se puede llevar a cabo cualquier tipo de modificación, pudiéndose inducir nuevas aberraciones no presentes en el sistema. El sistema puede ser un telescopio, un láser o un ojo humano. Esta Tesis se centra en aplicaciones relativas al ojo humano. Para la modificación de los frentes de onda se pueden utilizar diferentes elementos entre los que se encuentran las lentes optoelectrónicas y los espejos deformables. La óptica adaptativa es capaz de deformar de infinitas formas los frentes de onda, abriendo un abanico de aplicaciones ilimitado. Sin embargo, las aplicaciones más interesantes son las que hacen uso de un bucle cerrado (*closed loop*), en el cuál se miden las aberraciones y se modifica la superficie del elemento corrector en cada iteración. Contar con un sistema de medición de las aberraciones de alto rendimiento es fundamental para que un bucle cerrado funcione al máximo de sus posibilidades. Por lo tanto, los desarrollos presentados en esta Tesis son muy importantes para todas las aplicaciones de óptica adaptativa que podrían beneficiarse de una implementación de alto rendimiento, que sólo requiere de un procesador gráfico.

Es importante tener en cuenta que si bien la óptica adaptativa se utiliza en otros campos como telescopios, microscopios o lentes de cámaras no son sistemas directamente comparables. Esto es debido al hecho de que el ojo es un sistema vivo, dinámico y frágil. Hay diversos factores que pueden reducir la calidad de las imágenes de Hartmann-Shack en un ojo humano, tales como reflejos corneales (que degradan significativamente la imagen),

o las cataratas, debido a las cuales la luminosidad de las distintas áreas puede presentar variaciones importantes. Además, la luminosidad también puede variar significativamente debido a la calidad de la lágrima y otros factores.

El sensor de frentes de onda Hartmann-Shack (sensor H-S), descrito en 1971 por Shack *et al.* [17], y a su vez basado en el sensor propuesto por Hartmann en 1900 [18], es el método más ampliamente utilizado para la medición de aberraciones hoy en día. El sensor H-S consiste de un array de microlentes, cada una de las cuales muestrea la apertura de un sistema óptico. Además un sensor H-S incluye un sensor de imagen que detecta el patrón de puntos generado (normalmente llamado imagen de H-S). En un sistema ideal sin aberraciones ópticas cada microlente enfoca el frente de onda muestreado en su distancia focal y un patrón regular es detectado por el sensor de imagen. La figura 1.5a ilustra el funcionamiento básico de un sensor H-S. En concreto, se observa cómo un frente de onda plano, al pasar por las microlentes del H-S, converge en puntos perfectamente enfocados y alineados entre sí, formando la imagen que se muestra en 1.5c. Por otro lado, en un sistema real el frente de onda presenta irregularidades que causan cambios locales en la inclinación sobre cada microlente, de tal forma que se detecta un patrón distorsionado y desplazado. La figura 1.5b ilustra un caso en el que un frente de onda aberrado pasa a través de unas microlentes de H-S y converge de forma no ordenada, tal y como corresponde a su aberración. La figura 1.5d ejemplifica cómo sería la imagen de H-S de un frente de onda aberrado que contiene además un reflejo corneal. El desplazamiento de cada punto con respecto a su posición ideal (la posición obtenida con un frente de onda perfecto) se relaciona con la inclinación local del frente de onda, que a su vez está relacionado con la derivada local del frente de onda. Por lo tanto, la medición de frentes de onda con sensores H-S consiste en tomar una imagen del patrón de puntos, calcular la posición de cada punto (p.e., calcular su centro de masas), calcular el desplazamiento de cada punto en x e y , y finalmente, integrar esas derivadas locales para obtener la forma del frente de onda o mapa de la aberración.

Para procesar una imagen de Hartmann-Shack de un ojo humano y obtener los valores de los coeficientes de Zernike de dicho ojo es necesario llevar a cabo varios pasos. En primer lugar es necesario disponer de una imagen de referencia en la que se prescinde del ojo humano y solamente se miden las aberraciones del sistema óptico. Toda lente presenta pequeñas aberraciones que podrían distorsionar el resultado si no se tienen en cuenta. Una vez obtenida la imagen a procesar es necesario conocer la posición del ojo en la imagen. Si bien cuando se utilizan sensores H-S para medir lentes artificiales la posición de las

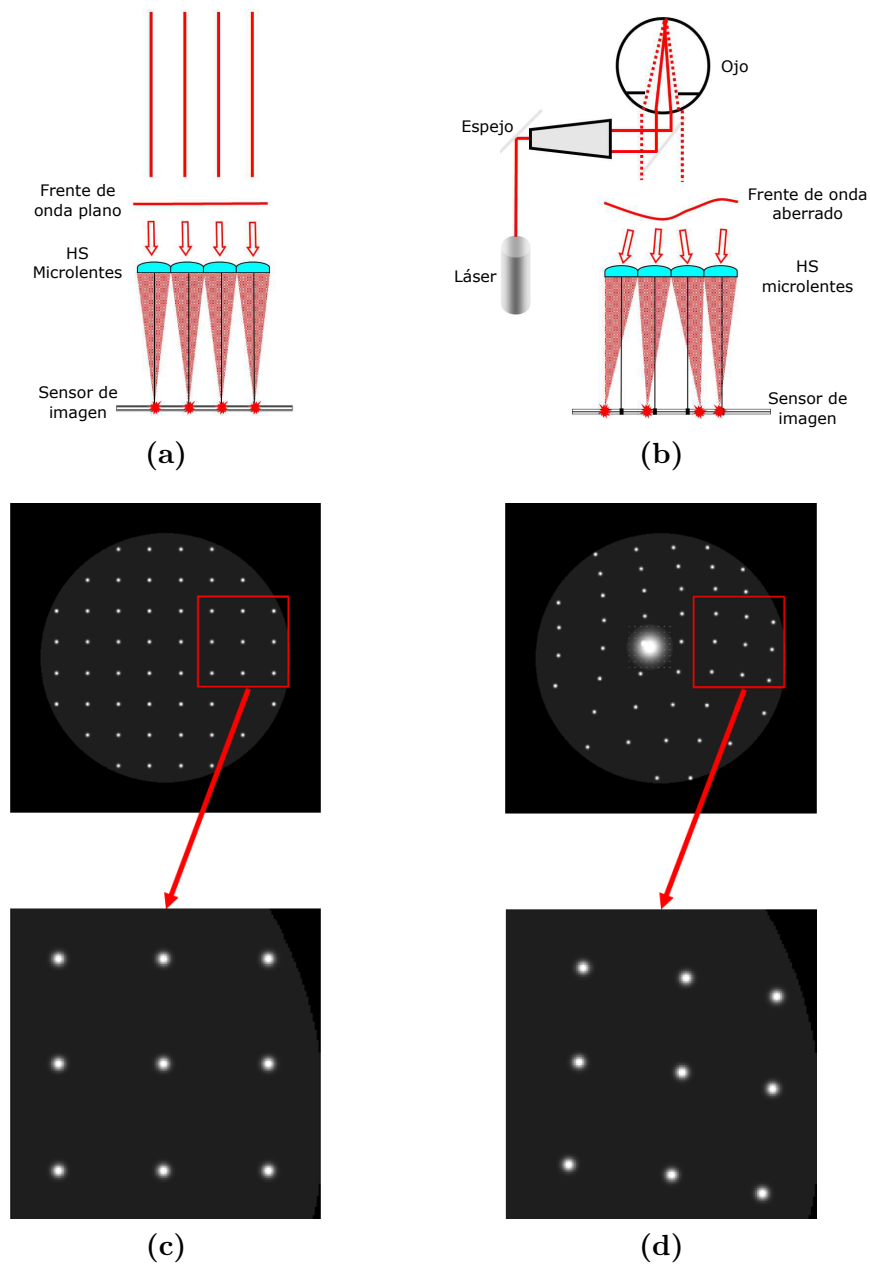


Figura 1.5: (a) Diagrama de un sensor de frentes de onda H-S registrando un frente de onda plano. (b) Diagrama de un sensor de frentes de onda H-S registrando un frente de onda aberrado (las imágenes a) y b) son imágenes creadas por Pedro M. Prieto, se reproducen aquí con permiso del autor). (c) Imagen de los puntos de H-S de un frente de onda perfecto. Arriba se muestra todo el ojo y abajo una parte del mismo ampliada. (d) Imagen de los puntos de H-S de un frente de onda aberrado. Arriba se muestra todo el ojo y abajo una parte del mismo ampliada.

mismas es estable e incluso puede cubrir completamente el sensor de imagen, cuando se miden ojos humanos en vivo es necesario llevar a cabo un seguimiento de la pupila debido a los continuos movimiento oculares. Una vez conocida la posición de la pupila se localizan los centroides de los puntos focalizados en cada microlente. Las posiciones de los mismos son sustraídas de las posiciones de sus equivalentes en la imagen de referencia. De esta forma se obtiene el desplazamiento de los puntos al medir el ojo. Con esa información se lleva a cabo la resolución de un sistema de ecuaciones sobre determinado que implica a los polinomios de Zernike y a los desplazamientos medidos. Como resultado se obtienen los valores de los coeficientes de Zernike.

El procesamiento de las imágenes de Hartmann-Shack en tiempo real requiere la realización de numerosas operaciones de procesamiento de imágenes de un elevado coste computacional. Esto hace que sea un candidato ideal para su paralelización con GPUs. Además, el procesamiento de las imágenes de Hartmann-Shack en tiempo real es fundamental para el funcionamiento de aplicaciones muy interesantes como bucles cerrados de óptica adaptativa en humanos. O de otro tipo de experimentos que reaccionen ante cambios en la refracción del sujeto. Ésta ha sido la motivación de la paralelización llevada a cabo que ha conseguido acelerar en gran medida el procesamiento y permitir que determinadas aplicaciones puedan funcionar en tiempo real.

Un reto adicional asociado al procesamiento de las imágenes de Hartmann-Shack es la localización de la pupila. En este caso pueden utilizarse dos métodos. Es posible añadir una segunda cámara al sistema e iluminación infrarroja para tomar imágenes directas del ojo además de imágenes del sensor Hartmann-Shack y llevar a cabo el seguimiento de la pupila en las imágenes directas. Sin embargo, eso añade una complejidad significativa al sistema, tanto hardware como software, debiendo ser necesario alinear ambas cámaras para que las áreas abarcadas por ambas coincidan. Por otro lado, añadir una segunda cámara e iluminación infrarroja reduce la calidad de la imagen del sensor de Hartmann-Shack reduciendo la cantidad de luz que éste recibe y añadiendo ruido. Por otra parte, si se requiere una sincronización total entre ambas cámaras será necesario añadir mecanismos software, y tal vez hardware, que lo garanticen. Por lo tanto, si no se requiere la información extra que aporta la imagen de infrarrojo es posible llevar a cabo el seguimiento de la pupila en las propias imágenes de Hartmann-Shack. Para ello se ha desarrollado y paralelizado un nuevo algoritmo que lo realiza, y como resultado devuelve una elipse que se ajusta a la pupila. Además, para analizar las imágenes de Hartmann-Shack se requiere un centro desde el que procesarlas. En ocasiones se utiliza una posición estática suminis-

trada manualmente por el operario o se utilizan rudimentarios algoritmos como el centro de gravedad de la imagen. Algunos de los sistemas utilizados en el LOUM permiten una amplia movilidad a los ojos de los sujetos, por lo que una solución estática no es válida y el centro de gravedad presenta limitaciones de precisión obvias en imágenes que no son homogéneas y que pueden tener reflejos. Es por todo ello que el algoritmo desarrollado es una herramienta fundamental en los experimentos que permiten (y en ocasiones propician) la movilidad de los ojos del sujeto. Incluso en experimentos que se realizan de forma estática es habitual que el sujeto se desplace, especialmente si se llevan a cabo durante prolongados periodos de tiempo. Y también en ese caso realizar el seguimiento de sus ojos en tiempo real facilita el desarrollo del experimento y permite obtener unos resultados más precisos.

Un análisis detallado de los algoritmos implementados y paralelizados para el procesamiento de imágenes de sensores Hartmann-Shack se presenta en el capítulo 4. Haciendo especial énfasis en las optimizaciones aplicadas a la implementación en CUDA y los excelentes resultados obtenidos tanto en rendimiento como en precisión.

1.5 Tomografía de Coherencia Óptica (OCT)

Las tomografías de coherencia óptica (OCT, por sus siglas en inglés) son una herramienta con multitud de aplicaciones médicas debido a que, de una forma no invasiva, permiten tomar imágenes de zonas no superficiales del cuerpo humano. Son especialmente útiles en oftalmología al permitir ver con gran precisión el estado de la retina y del cristalino. La figura 1.6 muestra un corte transversal de un ojo humano tomado *in-vivo* utilizando el sistema OCT del LOUM. En él se ve un cristalino con cataratas en el que se aprecian diferentes estructuras que bloquean y disipan el paso de la luz. Mientras que en un ojo sano se vería una estructura más uniforme y más transparente.

Los sistemas OCT son un invento relativamente reciente, el término OCT fue utilizado por primera vez en 1991 por Fujimoto *et al.* [19]. A pesar de ello, la evolución ha sido constante desde su creación y los sistemas han aumentado de forma significativa sus capacidades a la par de los avances técnicos en los láseres, sistemas de adquisición, motores y ordenadores. Hoy en día estos sistemas son capaces de capturar del orden de más de 1 GB/s de datos. Además, esos datos deben ser transformados para obtener la imagen final, así que se requiere una gran capacidad de cómputo para su procesamiento. Dicho proce-

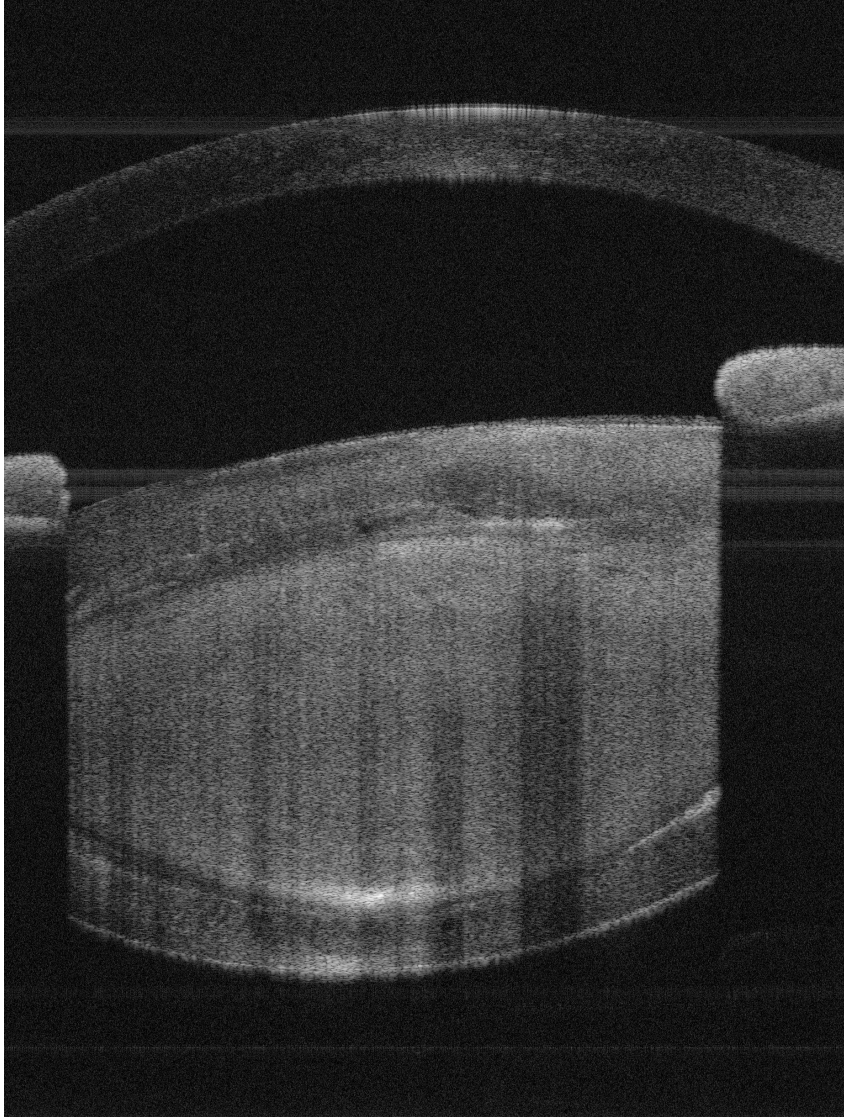


Figura 1.6: Corte transversal de una córnea y un cristalino con cataratas tomado *in-vivo* con el sistema OCT del LOUM.

samiento incluye el cálculo de la transformada rápida de Fourier y de otras operaciones igualmente costosas en tiempo de ejecución.

Un volumen de OCT puede tener un tamaño muy variable, pero si se trata de una medida de alta resolución con un sistema moderno puede llegar a ocupar varios Gigabytes. Un volumen de OCT está compuesto por muchas medidas completamente independientes que pueden procesarse en paralelo. De hecho, muchas de las operaciones que se aplican a los datos son independientes a nivel de vóxel (equivalente a un píxel en un volumen 3D). Por tanto, el uso de procesadores gráficos (GPUs) para el procesamiento de los datos es *a priori* adecuado para este tipo de datos. Las GPUs cuentan con un elevado ancho de banda de memoria y una gran capacidad de cómputo si se comparan con otros tipos de procesadores comunes. Especialmente si se comparan con las CPUs, que están más optimizadas para ejecuciones con divergencias y accesos a datos no tan continuos como las GPUs.

En el capítulo 5 se muestra y explica como se ha paralelizado y optimizado el procesamiento de datos de OCT mediante diferentes lenguajes de programación. Además, se han comparado los tiempos de ejecución de CPUs y GPUs con versiones paralelas en los dos casos. Conseguir procesar los datos de OCT en tiempo real sigue siendo un reto hoy en día, en gran medida debido a que a la vez que aumenta la capacidad de cómputo aumenta la capacidad para generar datos.

1.6 Estructura de la Tesis

Esta Tesis se divide en 5 capítulos que se describen a continuación:

1. **Introducción:** En la introducción se presentan los temas en los que se ha trabajado en esta Tesis, a saber: GPGPU (General-purpose computing on graphics processing units); seguimiento pupilar de alta velocidad mediante GPUs; Gafas de corrección automática de la Presbicia; implementación GPU del procesamiento de imágenes de Hartmann-Shack; y, paralelización del procesamiento de volúmenes de OCT. Asimismo, se exponen las razones por las que estas temáticas resultan interesantes para su estudio. Y, especialmente, su idoneidad para ser paralelizadas y ejecutadas en procesadores gráficos.
2. **Seguimiento pupilar de alta velocidad mediante GPUs:** Este capítulo versa sobre la problemática de rendimiento que presentan algunos algoritmos de segui-

miento pupilar, y cómo se puede solventar mediante la paralelización y el uso de GPUs. Todo ello sin olvidar la importancia de mantener una alta precisión en la búsqueda de la pupila. Para obtener un visión amplia de las posibilidades en cuanto a rendimiento y precisión del seguimiento pupilar se han paralelizado y caracterizado tres algoritmos diferentes.

3. **Gafas de corrección automática de la Presbicia:** Utilizando parte del trabajo desarrollado en el capítulo anterior se ha creado un dispositivo portátil capaz de corregir en tiempo real y de forma dinámica la presbicia. Ello ha requerido la integración de diferentes componentes electrónicos como: cámaras, iluminación infrarroja, lentes optoelectrónicas, baterías, y un teléfono móvil equipado con una GPU de gama media/alta para el procesamiento de los datos. Así como un importante trabajo de integración para controlar adecuadamente todos los componentes. Además, se ha llevado a cabo una comparación de rendimiento y eficiencia de varios dispositivos: dos GPUs de SoCs móviles, dos FPGAs y una GPU de sobremesa.
 4. **Implementación GPU de procesamiento de imágenes de Hartmann-Shack:** Las imágenes de Hartmann-Shack se utilizan para medir las aberraciones ópticas del ojo humano, entre otras muchas aplicaciones. Además, su procesamiento conlleva la realización de costosas operaciones de procesamiento de imágenes, convirtiéndolo en un buen candidato para la paralelización en GPUs que se presenta en este capítulo. También se realiza un estudio y comparación de la precisión de tres algoritmos diferentes.
 5. **Paralelización de procesamiento de imágenes de OCT:** Los sistemas OCT permiten visualizar de forma no invasiva partes no superficiales del ojo humano, como el cristalino y la retina. Estos sistemas cuentan además con una muy alta calidad, lo que implica que se generen una gran cantidad de datos y estos requieran una gran cantidad de cómputo. Es por esto que se han empleado GPUs para llevar a cabo la paralelización del procesamiento. También se ha paralelizado y optimizado una versión para CPUs y se ha comparado el rendimiento obtenido por ambas versiones, así como con otras implementaciones en MATLAB y LabView.
 6. **Conclusiones y perspectivas:** Finalmente se muestran las conclusiones del trabajo desarrollado, recordando los resultados más importantes obtenidos. Además,
-

se exponen posibles mejoras o nuevas investigaciones que se podrían llevar a cabo gracias a lo aprendido de los resultados de esta Tesis.

2 Seguimiento pupilar de alto rendimiento

2.1 Introducción

Dada la popularidad actual de las GPUs como coprocesadores de alto rendimiento y la cantidad de aplicaciones que tiene el seguimiento pupilar no es de extrañar que sea fácil encontrar entre la literatura existente trabajos que abordan el seguimiento pupilar utilizando procesadores gráficos. En concreto, Mulligan fue el primero en desarrollar una implementación GPU para la aceleración de un algoritmo de seguimiento pupilar [20]. Para ello utilizó un lenguaje de alto nivel desarrollado por la NASA conocido como QuIP orientado al procesamiento de imágenes en GPUs. El algoritmo que aceleró es una idea simple pero efectiva que utiliza una umbralización para determinar si un píxel pertenece a la pupila o no. A continuación, sobre una Región de Interés (ROI) de 192x192 píxeles calcula el centro de gravedad inverso de los píxeles que se detectaron como parte de la pupila durante la umbralización. De forma que los píxeles de menor brillo pesan más que los píxeles de mayor brillo. Finalmente, se ejecuta una comprobación para determinar si la ROI contiene o no una pupila. La velocidad obtenida por esta implementación es muy alta teniendo en cuenta el hardware en el que se ejecutaba, obteniendo una velocidad de procesamiento de 250 imágenes por segundo, eso sí, con imágenes de baja resolución.

Otra investigación publicada por Borovikov utiliza una implementación GPU para el seguimiento de pupila, pero en esta ocasión utiliza imágenes a color en lugar de imágenes en blanco y negro con luz infrarroja [21]. Un detector de blobs personalizado se utiliza para buscar la pupila mediante un método iterativo que asume que la pupila es un blob negro situado cerca del centro de la imagen. El método iterativo calcula el centro de masa inverso y, si el desplazamiento con respecto al centro anterior es menor que un cierto umbral, la búsqueda termina. Otra implementación GPU, publicada con posterioridad a la nuestra,

es la presentada por Du Plessis y Blignau [22], su desarrollo es capaz de alcanzar una velocidad de 200 imágenes por segundo utilizando una GPU de gama media. Utiliza para ello una implementación programada con shaders en HLSL (High Level Shader Language), en lugar de utilizar kernels de CUDA u OpenCL. Asimismo, el algoritmo de búsqueda se basa en una umbralización, similar a la que se presenta en esta Tesis. Sin embargo, debido a las limitaciones del hardware utilizado la velocidad obtenida no es muy alta a pesar de la baja resolución de las imágenes procesadas.

Evidentemente, en la literatura hay muchos más algoritmos de seguimiento pupilar que no utilizan GPUs. Un algoritmo conocido por su buena precisión para detectar una pupila en condiciones de iluminación no óptimas es el Starburst [23]. Este algoritmo presenta una carga computacional significativa, si bien la primera parte del mismo implica varias operaciones de procesamiento de imágenes que conllevan una carga computacional alta, la segunda parte del mismo es más moderada en sus exigencias computacionales. En primer lugar, busca los reflejos corneales que suponen una fuente de errores para la segunda parte del algoritmo. En segundo lugar, comienza una búsqueda iterativa de la pupila. Partiendo de un centro inicial, analiza el gradiente de una serie de rayos que avanzan hacia los bordes de la imagen en busca de un cambio brusco en el mismo que supere un cierto umbral. Los puntos encontrados con cambios bruscos en el gradiente son considerados puntos en el borde de la pupila. En esta primera búsqueda se obtiene una primera aproximación de la pupila que se irá refinando en los siguientes pasos. De este modo, el procedimiento se repite a partir de los puntos encontrados pero en dirección contraria y esta vez en forma de cono. Con todos los puntos encontrados se calcula el centro de masa de los mismos que es utilizado como centro de partida en la siguiente iteración. El proceso se detiene cuando el centro de masa se estabiliza, entonces se aplica un ajuste a un círculo mediante RANSAC (Random Sample Consensus) [24] con los puntos del borde de la pupila encontrados. El algoritmo RANSAC se detalla en profundidad en la sección 2.3.3. Por otra parte, el algoritmo Starburst que se ha utilizado en la comparativa de rendimiento y precisión se detalla en la sección 2.3.5.

Otro algoritmo utiliza el corte de grafos, que consiste en una técnica de minimización de la energía a lo largo de un contorno en un imagen, para encontrar el borde de la pupila [25]. Este algoritmo busca el "corte" que maximiza el flujo a lo largo del mismo. Los píxeles se codifican como nodos y los ejes identifican las diferencias con sus vecinos. Para evitar que las pestañas se identifiquen como bordes de la pupila se ejecuta una búsqueda de las mismas antes de comenzar. Una investigación similar utiliza los contornos activos

(o snakes) para encontrar el borde de la pupila [26]. Utilizando una región de interés y una umbralización se obtiene una estimación inicial de la posición de la pupila. A partir de dicha estimación se ejecuta el algoritmo de contornos activos que es capaz de encontrar con una muy alta precisión el borde de la pupila, aunque sus resultados dependen del uso de imágenes con alto contraste.

Mientras que los algoritmos citados obtienen buenos resultados, hay alternativas más simples que generan resultados suficientemente aceptables, según la precisión y/o aplicación en consideración. La pupila suele tener unos bordes bien definidos en imágenes infrarrojas, por lo tanto una combinación del algoritmo de detección de bordes de Canny [27] y la transformada de Hough [28] para círculos ha sido usada para la búsqueda de pupilas [29]. Si bien en el trabajo de Masek la segmentación se lleva a cabo para detectar el iris el mismo algoritmo es aplicable a la búsqueda de pupila. Asimismo, la versión de Masek se centra en evaluar la precisión de su implementación y no incluye una versión GPU de la misma. Este ha sido otro de los algoritmos que se ha implementado y paralelizado y que más adelante se detalla en profundidad en la sección 2.3.6.

Otro algoritmo utiliza la detección de blobs [30], en este caso se utiliza una umbralización para seleccionar la pupila en la zona central de la imagen. Sin embargo, este método puede tener problemas en imágenes con pupilas parcialmente ocluidas por párpados y pestañas.

Un método para realizar seguimiento pupilar más exótico y sin duda original es el propuesto por Ohtani y Ebisawa [31]. Su propuesta consiste en utilizar dos juegos de LEDs para iluminar la pupila situados a dos distancias de la cámara, de forma que los LEDs más cercanos generen el efecto ojos rojos y los más lejanos no. Si se coordina adecuadamente el sistema de iluminación (que funciona de forma intermitente, lejos y cerca alternativamente) con la captura de imágenes se obtienen pares de imágenes en los que las pupilas aparecen en diferentes colores (blanco y negro). La sustracción de ambas imágenes genera como resultado una imagen negra con dos puntos brillantes que corresponden a las pupilas. Si bien esta solución simplifica en gran medida el procesamiento aumenta considerablemente la complejidad física del sistema, haciéndolo poco versátil para muchas aplicaciones de óptica visual humana. Además, añade limitaciones a la velocidad de captura o de movimiento de los ojos.

Por otra parte, el seguimiento pupilar tiene una aplicación potencial en aquellos sistemas que actualmente requieren que el ojo se encuentre estático. Dichos sistemas suelen emplear elementos de fijación que disminuyan los movimientos del sujeto, tales como improntas

dentales, estas son incómodos, lo cuál reduce el tiempo durante el cuál se pueden llevar a cabo experimentos. Actualmente, los aberrómetros permiten cierta libertad de movimiento utilizando seguimiento pupilar para determinar la posición del ojo en tiempo real [14]. Las aplicaciones de óptica adaptativa que incluyen elementos correctivos como espejos deformables o moduladores de luz espaciales tienen requerimientos más restrictivos en lo que a la posición de la pupila se refiere. Un sistema de seguimiento pupilar preciso y rápido permitiría a estos sistemas una mayor libertad de movimientos y una mayor comodidad para el sujeto. La simulación visual con óptica adaptativa [32] es un aplicación que se beneficiaría ampliamente de un seguimiento pupilar preciso de alto rendimiento.

Esta investigación viene por tanto a tratar de resolver una necesidad técnica de otras áreas que hasta ahora no se había abordado desde un punto de vista del alto rendimiento y la paralelización con aceleradores gráficos.

Con el propósito de obtener un visión amplia de diferentes algoritmos, y poder juzgar con precisión cuál de ellos obtiene una mayor velocidad y precisión se han implementado tres algoritmos. Para cada uno de ellos se ha programado una versión secuencial en C++ y una versión paralela para GPU en CUDA. Así, los tres algoritmos implementados son:

- Umbralización y etiquetado, detallado en la sección 2.3.4
- Starburst, detallado en la sección 2.3.5.
- Detección de bordes de Canny y Transformada de Hough, detallado en la sección 2.3.6.

2.2 Sistema de captura de imágenes

Para capturar las imágenes que se han utilizado en nuestra infraestructura de evaluación para determinar la precisión y rendimiento de cada uno de los 3 algoritmos paralelizados, se ha utilizado iluminación infrarroja y una cámara enfocando al ojo. El sistema utilizado para la captura de las imágenes fue el utilizado para otra investigación relacionada con la localización de la posición acromática de la pupila y el primer reflejo de Purkinje en una población normal [33]. Durante dicho estudio se tomaron imágenes de la pupila de los sujetos medidos utilizando un anillo de LEDs de iluminación infrarroja.

La figura 2.1a muestra un esquema del sistema óptico utilizado para tomar las imágenes de pupila, y la figura 2.1b muestra una vista frontal del mismo. La cámara se sitúa al

fondo y el anillo de LEDs infrarrojos al frente, lugar en el que el sujeto sitúa su ojo para llevar a cabo las medidas.

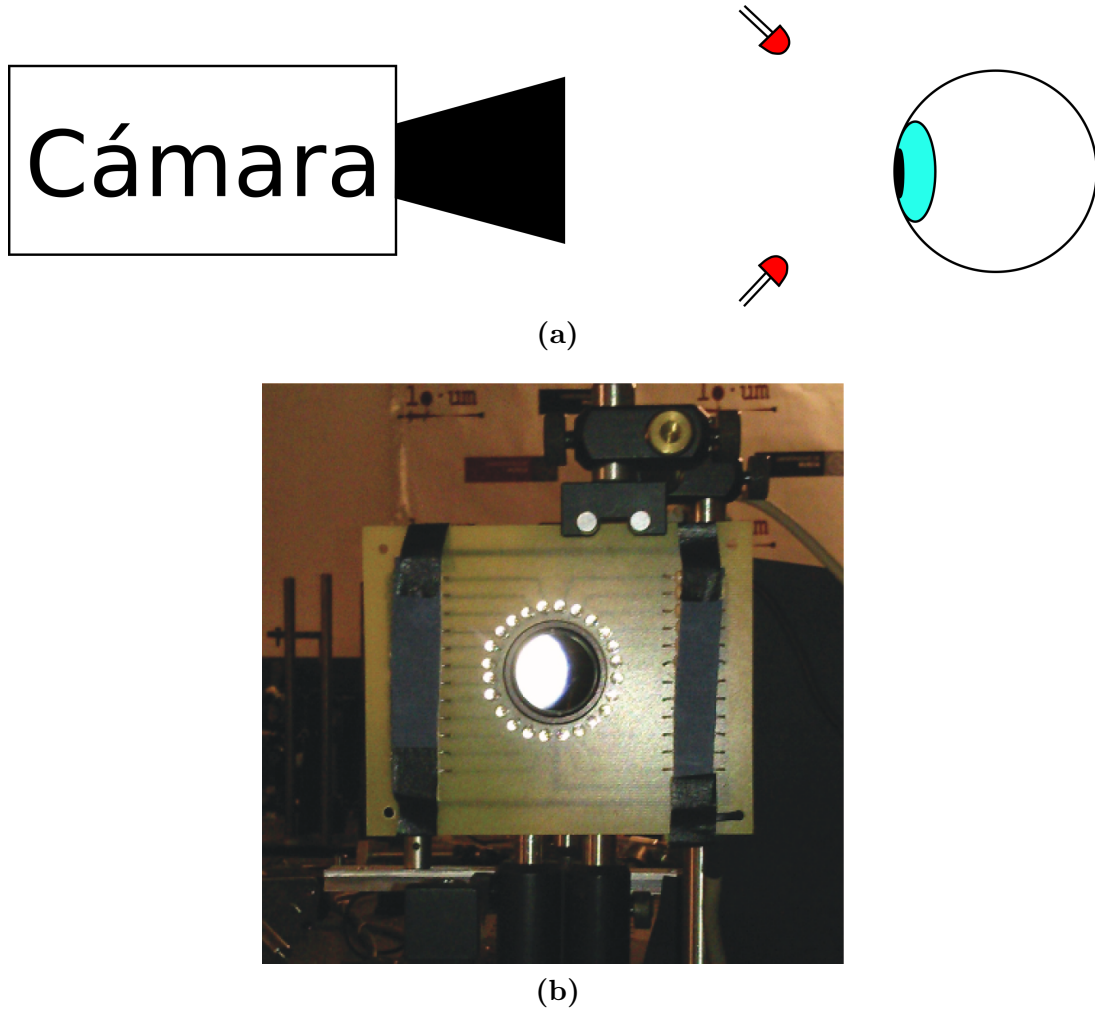


Figura 2.1: (a) Esquema del sistema óptico utilizado para tomar las imágenes del conjunto de pruebas y para realizar pruebas en tiempo real. (b) Imagen frontal del sistema óptico donde se muestran con precisión los LEDs infrarrojos utilizados para iluminar el ojo.

Durante la investigación desarrollada por Manzanera *et al.* [33] se generó un conjunto de 964 imágenes de 1280x1024 píxeles tomadas con el sistema de la figura 2.1. Las imágenes corresponden a los ojos de 51 sujetos diferentes, incluyendo una amplia variedad de radios de pupila que van desde los 1.66 a los 4.28mm, con una media de 3.21mm. La figura 2.2 muestra un histograma con los diámetros de las pupilas en el que se puede observar la variedad de los mismos. Asimismo, las imágenes presentan cierta variabilidad en lo que a

iluminación se refiere y, evidentemente, tienen una amplia variedad de formas y colores propia de un conjunto tan amplio de muestras. Para poder procesar de forma sistemática los datos y obtener resultados cuantitativos se han caracterizado manualmente las pupilas de todas las imágenes. Se ha desarrollado un programa en el que un usuario experto marca cinco puntos en el borde de la pupila tras lo cuál se ajusta una elipse a esos cinco puntos. Tras el ajuste se lleva a cabo una inspección visual y si se considera bueno se almacena y se carga la siguiente imagen para repetir el proceso. Fruto de los ajustes manuales se obtiene la posición y tamaño de la pupila para cada imagen.

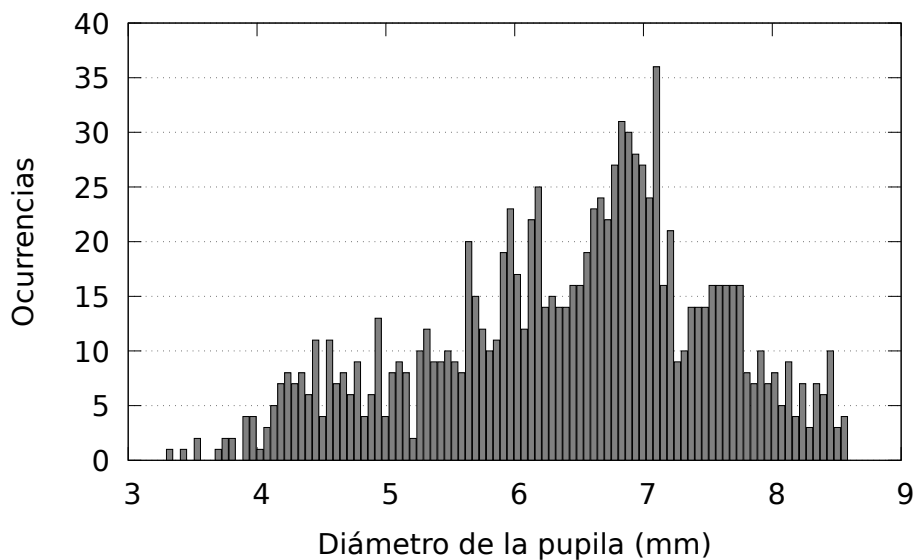


Figura 2.2: Histograma que muestra los tamaños de las pupilas presentes en el conjunto de imágenes evaluadas.

2.3 Algoritmos, paralelización y optimizaciones

En esta investigación se han implementado y paralelizado tres algoritmos, a saber: Starburst; umbralización más etiquetado; y detección de bordes y transformada de Hough. El algoritmo de umbralización y etiquetado es una modificación del propuesto por Rankin *et al.* [30]. En lugar de seleccionar el blob central se selecciona el blob más grande. Además, se prueban varios umbrales y se selecciona el que ofrece un mejor resultado.

2.3.1 Preprocesado

El preprocesado es una etapa fundamental para obtener una imagen que pueda ser procesada adecuadamente por los algoritmos una vez que se han eliminado aquellos elementos que podrían generar errores. La necesidad de esta etapa de preprocesado surge por la iluminación utilizada: los LEDs infrarrojos que se han usado generan reflejos corneales que son visibles en las imágenes que se capturan, como muestra la figura 2.3a.

Los reflejos corneales presentan un gran contraste con la pupila y suponen un problema cuando se procesan las imágenes en busca de la pupila. El algoritmo Starburst se ve afectado especialmente con este tipo de reflejos ya que tiende a confundirlos fácilmente con puntos del borde de la pupila. De ahí la importancia de eliminarlos antes de procesar la imagen.

Para eliminar los reflejos corneales de las imágenes se ha utilizado un método desarrollado por Aydi *et al.* [34]. Dicho método consta de los pasos que se ilustran en la figura 2.3. Primero se aplica una transformada "Tophat" (2.3b), después se umbraliza y se dilata el resultado (2.3c) y, finalmente, se interpola el valor de los píxeles umbralizados como se muestra en la figura 2.3d. La transformada Tophat consigue dar más relevancia a los reflejos corneales y eliminar prácticamente el resto de información presente en la imagen. Sin embargo, es la umbralización la que permite aislar e identificar los reflejos corneales de forma unívoca. Asimismo, la dilatación de las zonas umbralizadas permite ampliar el área que se interpola evitando que queden píxeles de los reflejos sin interpolar. Finalmente, para reemplazar los píxeles mediante la interpolación se calculan sus nuevos valores utilizando los valores de los píxeles vecinos. El resultado final de esta primera etapa de eliminación de reflejos corneales se muestra en la figura 2.3d. Es la misma imagen de la pupila original pero con los reflejos corneales completamente reemplazados por nuevos valores similares al resto de la pupila. Como se muestra más adelante, algunos de los procesados aplicados a las imágenes en estos pasos tienen un alto coste computacional, sin embargo, es posible optimizarlos para reducir sustancialmente su tiempo de ejecución.

Una vez eliminados los reflejos corneales es necesario aplicar un suavizado a la imagen para eliminar las estructuras que generan cambios bruscos en la imagen, especialmente las pestañas, dado que esas estructuras pueden mostrar patrones similares a los del borde de la pupila al procesarse. Se han probado dos métodos para suavizar las imágenes: un filtro 2D de Gauss y un filtro de la mediana.

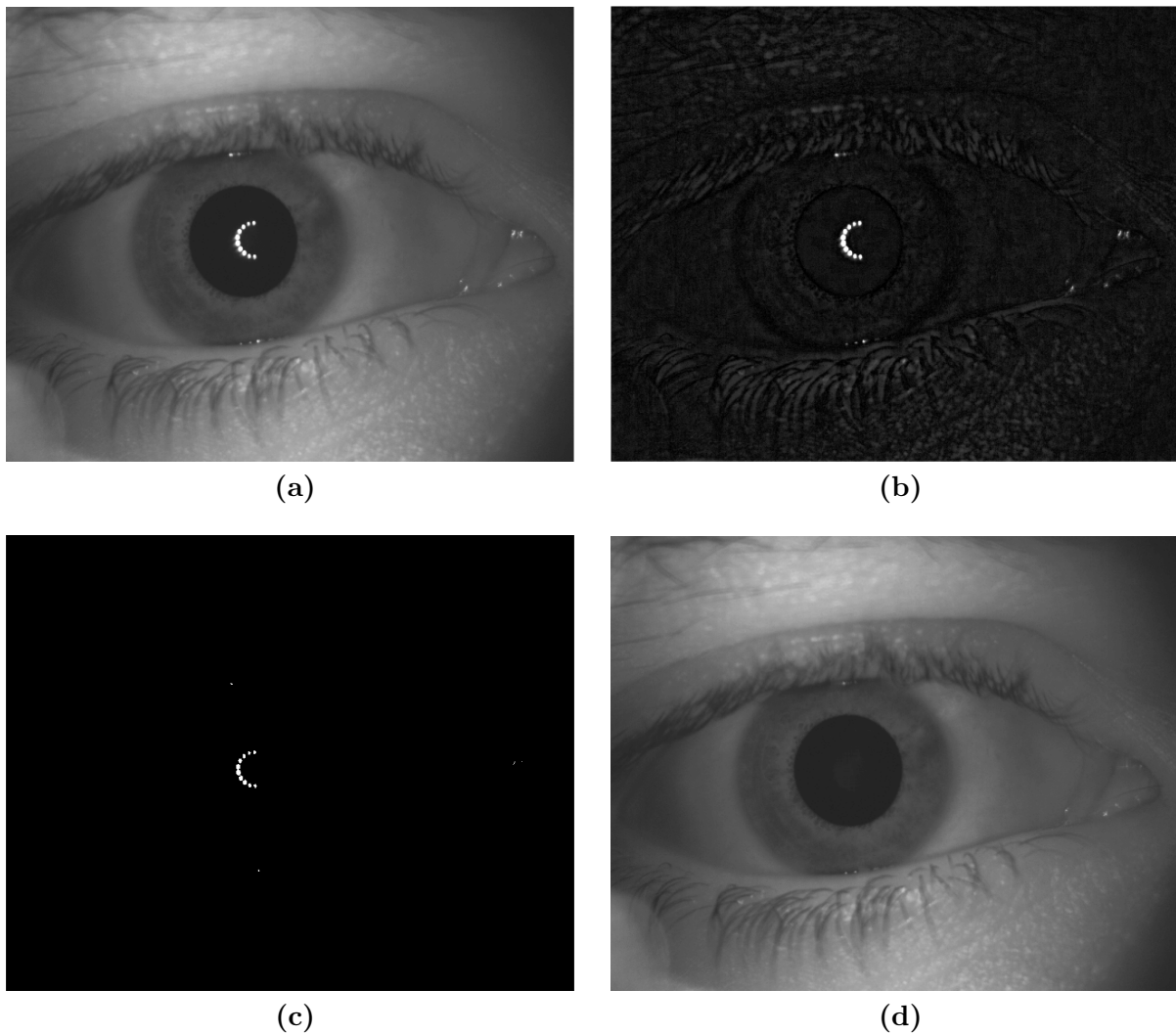


Figura 2.3: (a) Imagen original en la que se pueden ver los reflejos corneales en el interior de la pupila. (b) Resultado de aplicar la transformación *tophat* (sombrero de copa), donde se destacan los reflejos corneales. (c) Resultado de umbralizar y dilatar la imagen: solamente quedan visibles los reflejos corneales y algunos otros reflejos menores. (d) Resultado de eliminar los reflejos corneales reemplazando sus valores por nuevos valores interpolados con sus vecinos.

2.3.1.1 Paralelización de la fase de pre-procesado

Si bien las operaciones de procesamiento de imágenes suelen presentar buenas oportunidades de paralelización es necesario llevar a cabo optimizaciones especializadas si se quiere obtener el máximo rendimiento. Algunas de las operaciones paralelizadas son: una operación morfológica de apertura (una operación morfológica de erosión y otra de dilata-

ción), una sustracción de imágenes, una umbralización, una dilatación y la interpolación de los reflejos corneales. La siguiente fórmula describe las operaciones llevadas a cabo:

$$\text{interpolación}(((f - f \circ b) > \text{umbral}) \oplus b)$$

En primer lugar se realiza la transformada Tophat ($f - f \circ b$). Para ello se lleva a cabo una operación morfológica de apertura ($\circ$). A continuación, se sustrae a la imagen original el resultado de la apertura. Con estas primeras operaciones se consigue prácticamente eliminar todos los elementos de la imagen a excepción de los reflejos corneales. Después se lleva a cabo una umbralización de forma que sólo deberían quedar los reflejos corneales. Tras ello, se aplica una operación morfológica de dilatación ($\oplus$) para aumentar las zonas detectadas como reflejos y evitar que no se reconozca correctamente el borde de los reflejos corneales. Finalmente se lleva a cabo la interpolación en la que se sustituyen los valores de los píxeles detectados como reflejos por nuevos valores interpolados a partir de los píxeles vecinos en la imagen.

Para reducir el ruido de la imagen y suavizar los bordes de algunas estructuras del ojo como las pestañas se lleva a cabo la operación de suavizado con un filtro de Gauss o un filtro de mediana.

A continuación, se detallan las optimizaciones aplicadas a la etapa de preprocesado. Haciendo especial hincapié en la necesidad de las mismas y las mejoras conseguidas con cada una de ellas:

1. Implementación mediante las NPP: La implementación base hace uso de la librería NVIDIA Performance Primitives (NPP). La cual incluye una serie de funciones de procesamiento de imágenes optimizadas para su ejecución en GPUs.
2. Implementación especializada: esta implementación reemplaza las funciones de la librería NPP por funciones especializadas que al ser más específicas consiguen aumentar el rendimiento.
3. Uso de filtros separables: Los filtros separables permite reducir significativamente la carga computacional que se tiene que llevar a cabo en algunas funciones de procesamiento de imágenes. Reduciendo como resultado el tiempo de ejecución.
4. Uso de memoria compartida: La memoria compartida ofrece un ancho de banda muy alto y una latencia muy baja, especialmente cuando se la compara con la memoria global. De esta manera su correcto uso permite aumentar el rendimiento de los kernels.

5. Uso de kernels fusionados: la técnica de fusionado de kernels es capaz de evitar cargas y almacenamientos a memoria global al unificar varios kernels y eliminar sincronizaciones redundantes.
6. Desenrollado de bucles: El desenrollado de bucles permite al compilador realizar optimizaciones más agresivas y una planificación de la ejecución de las instrucciones más óptima.
7. Tipos de datos: La utilización de tipos de los mejores tipos de datos en cada momento permite reducir los tiempos de ejecución y reducir la cantidad de datos que es necesario cargar y almacenar en memoria global.
8. Eliminación de la interpolación de los reflejos corneales: La interpolación de los reflejos corneales es una operación compleja y, que en última instancia sólo sirve para evitar que se detecten bordes falsos alrededor de los reflejos. Por lo tanto, basta con ignorar esas áreas, sin necesidades de reemplazar los píxeles que las ocupan por otros interpolados.

2.3.1.1.1 Implementación mediante las NPP

La paralelización y optimización de los kernels del preprocesado se ha llevado a cabo de una forma iterativa partiendo de una versión inicial más simple hacia una versión más compleja y rápida. En primer lugar se han utilizado las funciones de la librería **NVIDIA Performance Primitives (NPP)** para llevar a cabo las operaciones de procesado de imágenes. Todas salvo la interpolación que al ser muy específica no está implementada en las NPP. Si bien la sección 2.4 muestra los resultados globales obtenidos por los 3 algoritmos paralelizados, en esta sección se muestran los tiempos de ejecución de ciertas funciones (kernels) optimizadas y algunos parámetros de rendimiento de las mismas para comprender en mayor profundidad el funcionamiento de dichas optimizaciones. La tabla 2.1 muestra algunos datos de rendimiento de los kernels NPP utilizados en esta implementación, siendo el número más importante el que indica el tiempo de ejecución de cada kernel. Los kernels más lentos son los de erosión y dilatación, así como el filtro de Gauss, mientras que la substracción y la umbralización son comparativamente mucho más rápidos (varios órdenes de magnitud). Los resultados de la tabla se han obtenido con un procesador gráfico NVIDIA GTX 1080 Ti.

	Memoria Global (GB/s)	Datos transferidos (MiB)	Tiempo de ejecución (ms)	Detención por dependencia de memoria(%)	IPC	Peticiones de carga global
Erosión	2.40	21.75	10.80	93.61	0.75	124.215.296
Dilatación	2.40	21.75	10.80	93.61	0.75	124.215.296
Gauss	6.91	7.57	1.06	60.04	0.605	7.045.120
Substracción	252.88	3.59	0.012	77.71	1.14	81.920
Umbralización	209.14	2.32	0.007	55.96	1.54	61.440

Tabla 2.1: Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento usando las NPP.

2.3.1.1.2 Implementación especializada

Tras esta primera aproximación se llevó a cabo una implementación especializada de las diferentes operaciones de procesamiento de imágenes que permitieron un mayor rendimiento reduciendo copias de memoria innecesarias y kernels menos genéricos y más ajustados a la operación a realizar. En el código 2.1 se muestra la función utilizada para llevar a cabo la operación de erosión, el cuál es muy similar al de la operación de dilatación y en menor medida al filtro de Gauss. Cada hilo calcula el área a procesar y accede a todos los datos dentro de esa área. Esta implementación es funcional pero, como se verá más adelante, tiene mucho margen de optimización. Utilizando las herramientas de *profiling* que ofrece NVIDIA para CUDA se han medido algunos de los parámetros más significativos para comprender cuáles son los cuellos de botella de estos kernels. Un resumen de esos datos para cada kernel se muestra en la tabla 2.2. En ella se observa como los kernels que se encargan de la erosión, dilatación y filtro de Gauss obtienen una velocidad de acceso a memoria global muy pobre, aunque superior a la obtenida con la implementación de las NPP. Además pasan casi todo el tiempo esperando a recibir datos de memoria y, por lo tanto, su IPC (Instructions per Cycle) es también muy bajo. Otro dato interesante es la cantidad de datos transferidos desde memoria global que apenas supera los 10MB. Esto incluye tanto las lecturas como las escrituras y se debe al buen rendimiento que obtienen las cachés para estas operaciones que consiguen absorber prácticamente todos los accesos a memoria. Para comparar los tiempos de ejecución de los kernels de umbralización y substracción con los de la implementación NPP es necesario

tener en cuenta que en la implementación NPP se han utilizado imágenes con un tipo de datos más optimizado (`uint8`) frente a los enteros utilizados en este momento. Se ha optado por hacerlo así para obtener el máximo rendimiento posible de la versión NPP, mientras que más adelante se mostrará el efecto que tiene utilizar el tipo de datos `uint8` en la implementación especializada.

	Memoria Global (GB/s)	Datos transferidos (MiB)	Tiempo de ejecución (ms)	Detención por dependencia de memoria(%)	IPC	Peticiones de carga global
Erosión	4.77	11.77	2.94	94.92	0.59	58.872.640
Dilatación	5.02	12.16	2.94	94.89	0.59	58.872.640
Gauss	13.79	10.69	0.87	92.1	0.51	8.223.312
Substracción	325.15	15.28	0.046	90.57	0.65	327.680
Umbralización	311.2	10.01	0.031	85.73	0.83	163.840
Interpolación	162.25	15.73	0.095	77.58	0.97	743.040

Tabla 2.2: Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación especializada de los mismos.

Código 2.1: Código CUDA de la implementación especializada de un filtro de erosión.

```

1 __global__ void Erosion(int * sr , int * dst, int width, int height, int ↵
    ↵ radius) {
2     int x = blockIdx.x * blockDim.x + threadIdx.x;
3     int y = blockIdx.y * blockDim.y + threadIdx.y;
4     if (y >= height || x >= width) {
5         return;
6     }
7     int start_i = max(y - radius, 0);
8     int end_i = min(height - 1, y + radius);
9     int start_j = max(x - radius, 0);
10    int end_j = min(width - 1, x + radius);
11    int value = 255;
12    for (int i = start_i; i <= end_i; i++) {
13        for (int j = start_j; j <= end_j; j++) {

```

```
14         value = min(value, src[i * width + j]);  
15     }  
16 }  
17 dst[y * width + x] = value;  
18 }
```

Las operaciones de erosión, dilatación y filtro de Gauss presentan un patrón de procesamiento muy similar en el que solo varía la operación que sea aplicada, de forma que se han paralelizado de la misma forma. En la versión inicial la paralelización se lleva a cabo utilizando un hilo CUDA por cada píxel de salida (las imágenes tienen 1280x1024 píxeles), de forma que cada hilo procesa k^2 píxeles, siendo k el tamaño del filtro aplicado. Las operaciones de erosión y dilatación simplemente calculan el mínimo y el máximo dentro del área analizada por el filtro, mientras que el filtro de Gauss pesa el valor de cada píxel dentro del filtro con unos pesos previamente definidos. Por tanto, el filtro de Gauss tiene que cargar simultáneamente la información de la imagen y de los pesos del filtro a aplicar para obtener el resultado.

Por otra parte, las operaciones de substracción y umbralización son muy simples y la implementación inicial, en la que se lanza un hilo por cada píxel de salida, ya obtiene muy buenos resultados por lo que no ha sido necesario llevar a cabo tantas optimizaciones como en el resto de kernels.

Finalmente, la operación de interpolación es algo más compleja. También se lanza un hilo CUDA por cada píxel de salida pero la mayoría de ellos no realizan trabajo efectivo debido a que caen en zonas sin reflejos corneales detectados. Para ello, todos los hilos comprueban al inicio si el píxel que se les ha asignado ha sido clasificado como un píxel perteneciente a un reflejo. Si se trata de un reflejo, se lleva a cabo la interpolación y la sustitución del valor del mismo. Para interpolar el nuevo valor se buscan los cuatro vecinos más próximos en horizontal y vertical que estén fuera del reflejo y se pesan sus valores con la distancia de los mismos al píxel reemplazado, obteniendo así su nuevo valor. La figura 2.4 muestra cómo se lleva a cabo la interpolación del valor de un píxel.

Además del filtro de Gauss también se ha explorado la opción de utilizar un filtro de la mediana 2D. Ambos consiguen un resultado similar suavizando las imágenes y eliminando el ruido que pueda haber en ellas. Si bien el filtro de la mediana se utiliza habitualmente para eliminar ruido en imágenes, en nuestro caso estamos interesados tanto en su capacidad de eliminar ruido como de suavizar las imágenes. Para la paralelización

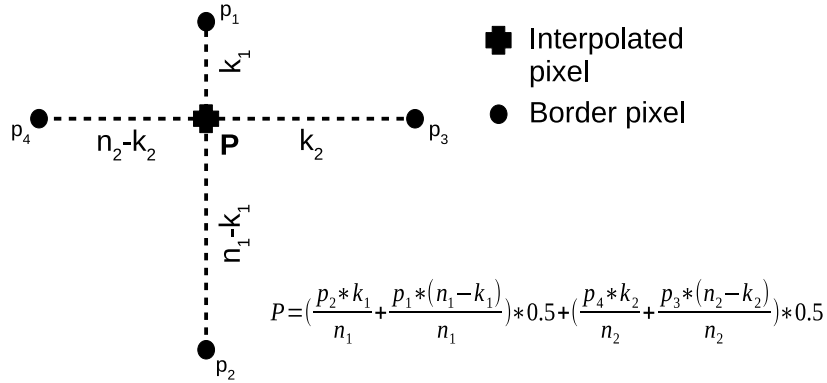


Figura 2.4: Interpolación de los reflejos corneales.

del filtro de la mediana se ha utilizado un hilo CUDA por cada píxel de salida, si bien cada hilo tiene que realizar bastante trabajo. Cada hilo cuenta con un histograma de 256 posiciones que utiliza como contador de los valores de los píxeles de la ventana procesada. Finalmente, escanea el histograma para determinar el valor de la mediana de la ventana y obtener el valor del píxel de salida. El filtro de la mediana presenta una complejidad y carga computacional mayor que el filtro de Gauss a la vez que menores oportunidades de optimización por lo que, como se verá en la sección de resultados, su rendimiento es menor.

2.3.1.1.3 Uso de filtros separables

Tras una primera implementación de los kernels se lleva a cabo una nueva iteración en la optimización de los mismos, utilizando una propiedad de los filtros aplicados en las operaciones de erosión, dilatación y el filtro de Gauss: todos ellos son filtros separables. Los filtros separables son los filtros de 2 dimensiones que se pueden dividir en dos filtros de 1 dimensión y que al multiplicarlos son equivalentes al original. A continuación se muestra un ejemplo de una matriz separable en dos filtros:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

Sustituyendo filtros de 2D por filtros de 1D se obtiene una reducción significativa de la

carga computacional. Para una imagen de $N * M$ píxeles y un filtro de tamaño k supone pasar de una complejidad $O(N * M * k^2)$ a otra de $O(N * M * 2 * k)$. La figura 2.5 muestra el cambio en la implementación de la paralelización. Al dividir el procesamiento en dos filtros de 1D es necesario dividir la implementación en dos kernels CUDA para que el segundo filtro de 1D pueda acceder a los resultados del primero. De forma que el primer kernel lanza un hilo por cada píxel que haya en la imagen y cada hilo procesa k píxeles, este kernel trabaja en horizontal. Mientras que el segundo kernel, de igual manera, lanza un hilo por cada píxel que haya en el resultado del filtro anterior y cada hilo procesa k píxeles, pero en este segundo paso el kernel trabaja en vertical. De esta forma, el segundo kernel aglutina los resultados de varios hilos del primer kernel y abarca todo el área del filtro original.

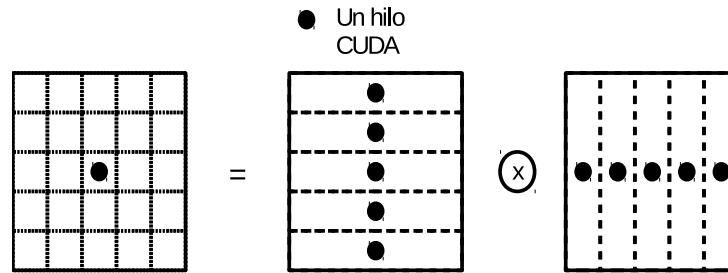


Figura 2.5: Muestra el cambio que se produce en la paralelización de las operaciones de procesamiento de imágenes al introducir los filtros separables.

El código 2.2 muestra la implementación del kernel de erosión con filtros separables. Ahora la ejecución se divide en dos kernels, uno trabaja en horizontal y el siguiente en vertical. Los datos muestran cómo el trabajo a realizar se ha reducido drásticamente. Asimismo cada uno de los kernels también es más sencillo que el kernel anterior, lo cuál facilita la optimización de los mismos.

La tabla 2.3 refleja la mejora de rendimiento que esta optimización ha supuesto para los kernels sobre los que se ha aplicado. Las detenciones por dependencias de datos han disminuido significativamente y el ancho de banda obtenido ha subido de forma muy importante. Como consecuencia el tiempo de ejecución también ha disminuido bastante. Asimismo, se puede apreciar que la cantidad de datos accedidos en memoria global es mayor ahora, esto se debe a que los datos de la tabla muestran los resultados agregados (o ponderados, según corresponda) de los dos kernels en que se ha dividido cada operación.

De forma que ahora cada operación morfológica necesita cargar y almacenar dos veces las imágenes.

	Memoria Global (GB/s)	Datos transferidos (MiB)	Tiempo de ejecución (ms)	Detención por dependencia de memoria(%)	IPC	Peticiones de carga global
Erosión	82.4	20.08	0.26	59.7	1.13	6.260.672
Dilatación	79.9	20.08	0.26	60.1	1.13	6.260.672
Gauss	56.15	20.11	0.29	89.3	0.6	3.283.072

Tabla 2.3: Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación con kernels separables.

Código 2.2: Código CUDA de la implementación con filtros separables del kernel de erosión.

```

1 __global__ void ErosionStep1(int * sr , int * dst, int width, int height, ↵
    ↵ int radius) {
2     ...
3     int start_j = max(x - radius, 0);
4     int end_j = min(width - 1, x + radius);
5     int value = 255;
6     for (int j = start_j; j <= end_j; j++) {
7         value = min(value, src[y * width + j]);
8     }
9     dst[y * width + x] = value;
10 }
11 __global__ void ErosionStep2(int * sr , int * dst, int width, int height, ↵
    ↵ int radius) {
12     ...
13     int start_i = max(y - radius, 0);
14     int end_i = min(height - 1, y + radius);
15     int value = 255;
16     for (int i = start_i; i <= end_i; i++) {
17         value = min(value, src[i * width + x]);
18     }

```

```

19     dst[y * width + x] = value;
20 }

```

2.3.1.1.4 Uso de memoria compartida

A partir de la optimización previa es fácil aprovechar la simplicidad de los nuevos accesos a memoria para, utilizando la memoria compartida, reducir los accesos repetidos a memoria global. En cada uno de los filtros los hilos contiguos en la dirección del filtro acceden repetidamente a los valores de los mismos píxeles en memoria global. Para evitar esos accesos se ha optado por cargar en memoria compartida bloques de memoria global organizados en *tiles* que se corresponden con los bloques de hilos de CUDA. A los datos cargados se les añade relleno (padding) si se encuentran en los bordes de la imagen para homogeneizar el procesamiento posterior. La tabla 2.4 muestra cómo el tiempo de ejecución disminuye significativamente tras aplicar esta optimización, disminuyendo también el porcentaje de detenciones por dependencias de memoria. Asimismo, el IPC se incrementa notablemente y el número de accesos a memoria global disminuye en un orden de magnitud.

	Memoria Global (GB/s)	Datos transferidos (MiB)	Tiempo de ejecución (ms)	Detención por dependencia de memoria (%)	IPC	Peticiones de carga global
Erosión	176.6	20.47	0.121	25.84	2.65	376.320
Dilatación	176.3	20.08	0.121	25.79	2.69	376.320
Gauss	228.8	19.97	0.094	27.2	3.06	399.744

Tabla 2.4: Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la fase de preprocesamiento para la implementación con memoria compartida.

El código 2.3 muestra el kernel que se encarga de realizar el primer paso del filtro de erosión utilizando memoria compartida. Como se puede ver, los hilos de cada bloque cargan conjuntamente los datos que van a procesar, consiguiendo casi eliminar las cargas de datos redundantes que ahora se limitan a las áreas frontera entre los bloques. El procesamiento se ha hecho más genérico que en la etapa anterior, de forma que ahora todos

los hilos ejecutan la misma cantidad de iteraciones del bucle, lo que es importante para siguientes optimizaciones.

Código 2.3: Código CUDA de la implementación con memoria compartida del filtro de erosión.

```

1 __global__ void ErosionStep1Shared(const int * src, int * dst, const int ↵
    ↵ width, const int height, const int stride, const int tile_w, const ↵
    ↵ int tile_h, const int radius) {
2     extern __shared__ int smem[];
3     int tx = threadIdx.x;
4     int ty = threadIdx.y;
5     int bx = blockIdx.x;
6     int by = blockIdx.y;
7     int x = bx * tile_w + tx - radius;
8     int y = by * tile_h + ty;
9     if (x >= width || x < 0 || y >= height) {
10         shsrc[ty * blockDim.x + tx] = 255;
11     } else {
12         shsrc[ty * blockDim.x + tx] = src[y * width + x];
13     }
14     __syncthreads();
15     if (x < (bx * tile_w) || x >= ((bx + 1) * tile_w)) {
16         return;
17     }
18     T * smem_thread = (T*)&smem[ty * blockDim.x + tx - radius];
19     T val = smem_thread[0];
20     for (int xx = 1; xx <= 2 * radius; xx++) {
21         val = min(val, smem_thread[xx]);
22     }
23     if (x < width && x >= 0 && y < height) {
24         dst[y * stride + x] = val;
25     }
26 }

```

2.3.1.1.5 Uso de kernels fusionados

A continuación se muestra otra optimización que consiste en fusionar tres kernels para evitar almacenamientos y cargas de datos de forma consecutiva en diferentes kernels. Como resultado se evitan todos esos accesos a memoria y se utilizan datos que ya están disponibles en los registros de cada hilo. Se han fusionado las operaciones de substracción y umbralización con la segunda etapa de la operación de dilatación. La tabla 2.5 muestra cómo el tiempo para el kernel de la dilatación fusionada es 22us (18%) superior al de la dilatación sin fusionar pero es 55us (28%) inferior al tiempo que lleva realizar las operaciones por separado.

	Memoria Global (GB/s)	Datos transferidos (MiB)	Tiempo de ejecución (ms)	Detención por dependencia de memoria(%)	IPC	Peticiones de carga global
Dilatación Fusionada	192.01	36.82	0.143	39.95	2.29	545.280

Tabla 2.5: Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación con kernels fusionados.

2.3.1.1.6 Desenrollado de bucles

Otra optimización que cobra más importancia cuando se hace uso de filtros separables es el desenrollado de bucles. El desenrollado de bucles consiste en eliminar o reducir el número de iteraciones de los bucles (por regla general bucles `for`) y reemplazarlos por las operaciones que el mismo ejecutaría durante sus iteraciones. Esto presenta algunas ventajas, la principal es que el compilador puede llevar a cabo una planificación de las operaciones de carga de datos más inteligente anticipándose con más tiempo a su utilización y reduciendo los tiempos de espera. En nuestro caso hemos optado por desenrollar el bucle que procesa el filtro de k píxeles para cada hilo. Sin embargo, el desenrollado de bucles presenta un inconveniente: la legibilidad del código y su mantenibilidad se ven significativamente reducidas. Por suerte, es posible utilizar el pragma `unroll` para indicar al compilador que se desea desenrollar un bucle, aunque para esto el número de iteraciones

del mismo debe ser conocido en tiempo de compilación. Es fácil, por tanto, establecer una cantidad fija de iteraciones y el código sigue manteniendo el mismo aspecto que antes de aplicar la optimización. Sin embargo, se ha eliminado toda la flexibilidad del mismo. Antes se podía utilizar cualquier tamaño para los filtros y ahora solo se puede utilizar el seleccionado en tiempo de compilación. Para eliminar parcialmente esa limitación se han utilizado plantillas de C++ que nos permite parametrizar un parámetro que se puede etiquetar como constante (`const`) en tiempo de compilación. Así que a la plantilla se le incluye como parámetro el radio del filtro, de forma que en tiempo de compilación se pueden incluir varias llamadas al kernel con distintos valores para el parámetro de la plantilla. Y en tiempo de ejecución se puede decidir cuál utilizar, recuperando parte de la flexibilidad perdida. El código 2.4 muestra una parte del kernel utilizado para implementar esta optimización. Destaca la parametrización con plantillas y el pragma `unroll` que se puede utilizar gracias al procesamiento genérico del bucle `for`.

Código 2.4: Código CUDA implementando la optimización de desenrollado de bucles en el filtro de erosión.

```
1 template<const int radius> __global__ void ErosionStep1SharedUnrolled(  
2     ↪ const int * src, int * dst, const int width, const int height, ↪  
3     ↪ const int stride, const int tile_w, const int tile_h) {  
4     /* Se cargan los datos */  
5     ...  
6     /* Se procesan los datos */  
7     #pragma unroll  
8     for (int xx = 1; xx <= 2 * radius; xx++) {  
9         val = min(val, smem_thread[xx]);  
10    }  
11    if (x < width && x >= 0 && y < height) {  
12        dst[y * stride + x] = val;  
13    }  
14 }
```

En la tabla 2.6 se muestran los nuevos números de rendimiento obtenidos tras esta optimización. El tiempo de ejecución se ha vuelto a reducir y la velocidad de acceso a memoria se ha vuelto a incrementar. Sin embargo, el IPC se ha reducido y el número de detenciones por dependencias de memoria se ha incrementado. Ello es debido a que al desenrollar el bucle se han eliminado un 30% de las instrucciones y por tanto es posible

que disminuya el IPC y el tiempo de ejecución a la vez. Por otra parte, al eliminar una gran cantidad de instrucciones dedicadas a comprobar los límites del bucle aumenta el porcentaje de detenciones de la ejecución por dependencias de memoria.

	Memoria Global (GB/s)	Datos transferidos (MiB)	Tiempo de ejecución (ms)	Detención por dependencia de memoria(%)	IPC	Peticiones de carga global
Erosión	195.5	19.89	0.102	36.2	2.11	376.320
Dilatación	205.2	20.03	0.102	37.4	2.1	376.320
Dilatación Fusionada	223.2	25.37	0.119	44.9	1.86	545.280
Gauss	271.4	20.1	0.078	43.5	2.62	399.744

Tabla 2.6: Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación con desenrollado de bucles.

2.3.1.1.7 Optimización del tipo de datos de las imágenes

También se ha optimizado el tipo de datos utilizado para almacenar las imágenes, en esta aplicación las imágenes utilizadas son imágenes en escala de grises cuyos píxeles pueden tomar valores entre 0 y 255. Por lo cuál basta con utilizar 8 bits (`uint8_t`) por píxel para almacenar su valor, sin embargo, se venían utilizando `ints` de 32 bits para almacenar cada píxel. Esto permite optimizar desde la copia de las imágenes de la CPU a la GPU hasta la carga de datos durante las operaciones. Aunque es cierto que al utilizar tipos de datos tan pequeños lo normal es que se reduzca la velocidad de las transferencias entre memoria, este efecto es menor que las ganancias generadas. La tabla 2.7 muestra cómo, efectivamente, se ha reducido la velocidad de transferencia de memoria global pero el tiempo de ejecución ha vuelto a disminuir a la vez que las detenciones por dependencias. Además, el IPC ha aumentado significativamente para todas las operaciones.

2.3.1.1.8 Eliminación de la interpolación de los reflejos corneales

Después de las optimizaciones aplicadas, el peso de la operación de interpolación sobre el total del tiempo de ejecución se ha incrementado significativamente. Su optimización es

	Memoria Global (GB/s)	Datos transferidos (MiB)	Tiempo de ejecución (ms)	Detención por dependencia de memoria(%)	IPC	Peticiones de carga global
Erosión	57.8	4.98	0.091	26.4	3.01	376.320
Dilatación	58.5	4.80	0.091	25.4	3.03	376.320
Dilatación Fusionada	73.6	6.57	0.097	25.9	2.97	545.280
Gauss	78.5	5.00	0.066	30.5	3.22	399.744
Interpolación	47.81	3.94	0.085	68.76	1.32	901.147

Tabla 2.7: Tabla que muestra la velocidad de acceso a memoria de cada uno de los kernels utilizados en la etapa de preprocesamiento para la implementación con uint8.

compleja debido a su patrón de accesos a memoria poco uniforme y su difícil paralelización con mucho trabajo distribuido entre unos pocos hilos. Sin embargo, es posible solucionar este problema prescindiendo completamente de la interpolación de los reflejos corneales. Durante la búsqueda de los puntos del borde en el algoritmo *Starburst* la imagen sin los reflejos se procesaba con normalidad, sin embargo, se puede asumir que las zonas con reflejos no contienen bordes de la pupila o que si los contienen no es posible localizarlos con precisión. Por lo que se puede eliminar la etapa de interpolación y en la etapa de búsqueda comprobar si los píxeles pertenecen o no a reflejos, lo cuál resulta en una ejecución más rápida.

2.3.2 Optimización de la copia de datos entre la CPU y la GPU

La copia de datos entre la CPU y la GPU supone en algunas de las implementaciones una gran parte del tiempo de ejecución, es por ello que se ha optimizado para reducirla al máximo. Cuando se programa una copia de datos entre la CPU y la GPU se producen varias operaciones. En primer lugar la zona de memoria accedida por la CPU debe ser *pinned memory*, es decir, su dirección de memoria es fija y no puede ser movida a otra localización (se debe reservar utilizando la función `cudaMallocHost()`). Al contrario de lo que ocurre con la memoria virtual paginada cuya dirección física puede variar en cualquier momento. Por tanto, si los datos a copiar no se encuentran en un área de memoria *fijada o pinned* éstos deben copiarse a una zona con esa propiedad. Una vez los datos se encuentran

en el área adecuada se inicia la copia entre la CPU y la GPU utilizando la conexión PCIe, la cuál tiene un ancho de banda teórico de 16GB/s en cada sentido pero el máximo que se alcanza en la realidad son unos 12GB/s. Por lo tanto, si se utiliza un área de memoria *pinned* para almacenar las imágenes en la CPU antes de copiarlas es posible obtener una velocidad de copia de 12GB/S. Mientras que si se utiliza una zona de memoria virtual paginable la velocidad de copia se reduce hasta los 6.6GB/s. Como resultado de utilizar memoria *pinned* se reduce el tiempo de las copias a prácticamente la mitad.

La **memoria *pinned*** es muy útil para aumentar la velocidad de copia, pero no se debe obviar el hecho de que si se hace un uso excesivo de ella se pueden generar problemas de agotamiento de la memoria disponible. La memoria paginada permite al sistema operativo una gestión mucho más flexible de la memoria disponible e incluso utilizar almacenamiento secundario para expandir la memoria disponible en situaciones en las que la memoria principal se llene. Sin embargo, la memoria *pinned* elimina dicha flexibilidad y no es posible utilizarla en escenarios dónde la memoria principal puede agotarse.

2.3.3 Ajuste a un círculo o elipse mediante RANSAC (RANDOM Sample Consensus)

RANSAC (RANDOM Sample Consensus) es un método para realizar ajustes óptimos a una serie de datos utilizando una pequeña muestra de los mismos y, sobre todo, es un método que puede distinguir el "ruido" entre los datos auténticos, siempre y cuando la proporción ruido/señal sea razonablemente baja [24]. En nuestro caso se ha utilizado para llevar a cabo el ajuste de un círculo o de una elipse a los puntos encontrados en el borde de la pupila. Gracias a la capacidad de RANSAC de descartar los puntos erróneos es posible evitar problemas con puntos detectados fuera de la pupila o con oclusiones pequeñas de la pupila por parte del párpado.

El algoritmo RANSAC basa su funcionamiento en la selección aleatoria de un subconjunto de los datos a los que se realiza un ajuste al modelo elegido. Después, ese ajuste se compara con el conjunto de los datos y se obtiene la cantidad de datos que "votan" por ese ajuste. De esta forma, si se elige al subconjunto adecuado de puntos se puede eliminar por completo el ruido de los datos. El algoritmo 1 muestra el funcionamiento detallado del algoritmo RANSAC realizando un ajuste a un círculo o a una elipse. Así, en el más alto nivel vemos un bucle que se ejecuta hasta que se encuentre un ajuste con muchos votos o hasta que se alcance el número máximo de iteraciones. Dentro de dicho bucle se lleva

a cabo la selección de los puntos aleatorios para cada ajuste. A continuación, se lleva a cabo el ajuste sobre los puntos seleccionados. Finalmente se cuenta la cantidad de votos de dicho ajuste y se compara con el mejor ajuste obtenido hasta el momento.

En los experimentos se han realizado un máximo de 1024 ajustes con 5 puntos seleccionados aleatoriamente para cada ajuste. Se han utilizado esas cantidades por obtener buenos resultados en las pruebas realizadas durante la validación del algoritmo. Asimismo, para que un punto "vote" a un ajuste se ha utilizado una distancia máxima de 2 píxeles. Este margen dependerá en gran medida del tamaño de píxel de cada sistema.

Algoritmo 1 Pseudocódigo del algoritmo RANSAC

```

1: procedure RANSAC(Puntos, NumPuntos)
2:   numPuntosPorAjuste  $\leftarrow$  5
3:   numDeAjustes  $\leftarrow$  1024
4:   porcentajeDeFinalización  $\leftarrow$  0.9
5:   margenVotos  $\leftarrow$  2
6:   mejorAjuste  $\leftarrow$  []
7:   maxNumVotos  $\leftarrow$  0
8:   ultimoPorcentaje  $\leftarrow$  0
9:   iter  $\leftarrow$  0
10:  while ultimoPorcentaje < porcentajeDeFinalización && iter <
    numDeAjustes do
11:    puntosSelec  $\leftarrow$  selecPuntosAleatorios(Puntos, numPuntosPorAjuste)
12:    ajuste  $\leftarrow$  ajustaPuntos(puntosSelec)
13:    votos  $\leftarrow$  cuentaVotos(Puntos, ajuste, margenVotos)
14:    if votos > maxVotos then
15:      maxVotos  $\leftarrow$  votos
16:      mejorAjuste  $\leftarrow$  ajuste
17:    end if
18:    ultimoPorcentaje  $\leftarrow$  votos/NumPuntos
19:    iter  $\leftarrow$  iter + 1
20:  end while
21:  return mejorAjuste
22: end procedure

```

2.3.3.1 Paralelización del algoritmo RANSAC para GPU

RANSAC normalmente se ejecuta de forma iterativa, pudiendo incluir una condición de final o un número fijo de iteraciones. Al paralelizarlo hemos optado por utilizar un número fijo de iteraciones/ajustes. Se ha dividido el algoritmo en varias etapas que emulan a las que se ejecutan en su versión iterativa. La figura 2.6 muestra un esquema de alto nivel del funcionamiento del algoritmo RANSAC en su versión paralela. En primer lugar se seleccionan conjuntos de puntos aleatoriamente que son los que posteriormente se utilizarán para hacer un ajuste a un círculo o a una elipse. En concreto se han utilizado 5 puntos para realizar cada ajuste y se han realizado 1024 ajustes. Para seleccionar los puntos se generan los índices de los mismos aleatoriamente utilizando la librería *cuRAND* proporcionada por *NVIDIA*. A continuación, se lleva a cabo el ajuste a un círculo (o a una elipse) para cada uno de los 1024 conjuntos de 5 puntos aleatoriamente generados. En el caso del círculo se utiliza el ajuste desarrollado por Taubin [35]. Tras ello, es necesario calcular la distancia de cada punto detectado en el borde de la pupila al círculo (o elipse) ajustados. Y finalmente, se suman los votos de cada ajuste y se selecciona el ajuste con más votos.

La etapa que lleva a cabo los ajustes a los círculos (o las elipses) se ha paralelizado lanzando un hilo por cada ajuste. Mientras que la etapa que calcula los votos de cada ajuste permite una paralelización mucho mayor: se lanza un hilo por cada punto del borde de la pupila para cada ajuste realizado. El trabajo de cada hilo consiste en calcular la distancia entre un círculo (o elipse) y un punto, para determinar si esa distancia está por debajo del límite establecido para contar un voto. A continuación, se lleva a cabo una reducción de los votos en otro kernel sumándolos todos y obteniendo el valor total para cada ajuste. Finalmente, otro kernel busca el ajuste con la mayor cantidad de votos. Tras ello, es necesario copiar el resultado al *host*. En primer lugar, se copia el índice del mejor ajuste y, a continuación, se copia el círculo (o la elipse).

Se han aplicado cinco optimizaciones al algoritmo RANSAC:

1. Eliminación del cálculo de los números aleatorios: generar números aleatorios es una operación muy costosa en lo que a coste computacional se refiere. Sin embargo, es posible prescindir de cierta aleatoriedad en nuestra aplicación.
2. Fusión de kernels: la principal ventaja de fusionar kernels es evitar cargas y almacenamientos necesarios para sincronizar los datos entre kernels que de otra forma se ejecutan independientemente.

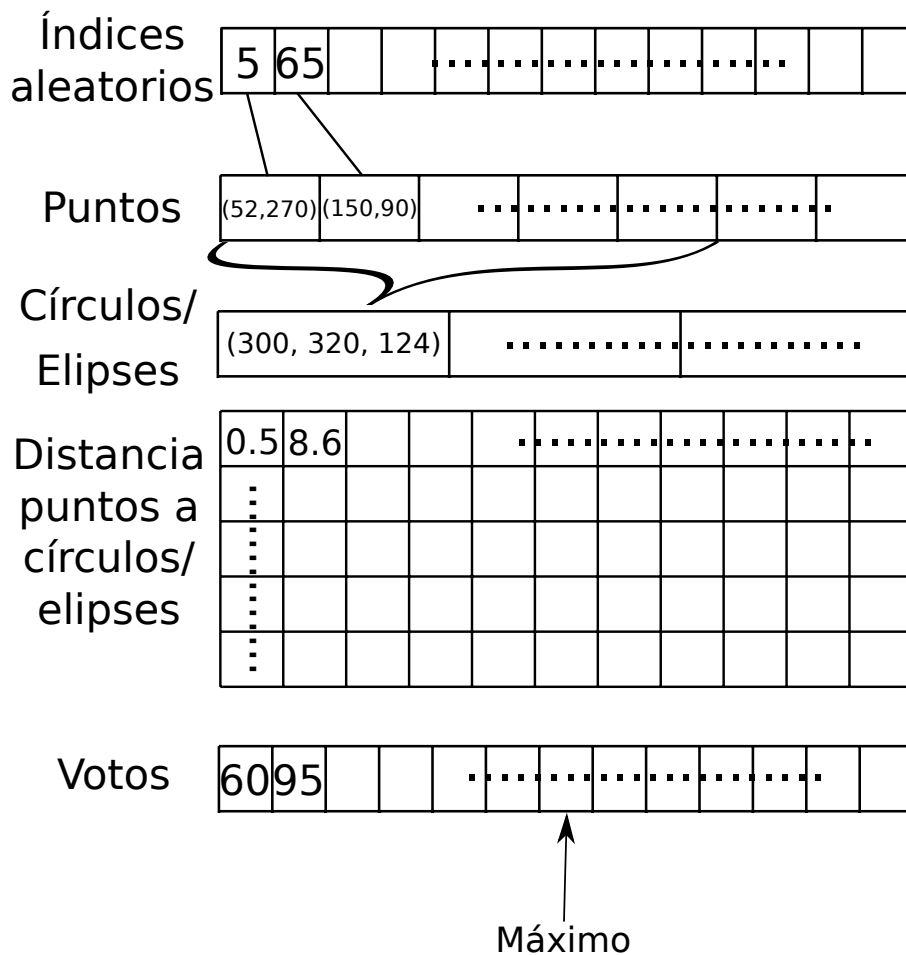


Figura 2.6: Esquema de alto nivel del funcionamiento del algoritmo RANSAC en su implementación paralela.

3. Uso de streams: los streams de CUDA permiten solapar ejecución de kernels y copia de datos. Así como ocultar la sobrecarga del lanzamiento de los kernels.
4. Fast math: algunas operaciones matemáticas cuenta en CUDA con equivalentes que ofrecen un mayor rendimiento a cambio de una menor precisión. El uso de dichas operaciones ha permitido mejorar el rendimiento de algunos kernels.
5. Reducción de las comunicaciones entre la GPU y el host: enviar datos entre la GPU y el host es una operación muy "lenta", especialmente cuando se envían pocos datos y se requiere una sincronización de la operación. Es por ello que reduciendo la cantidad de estas operaciones se puede reducir el tiempo de ejecución.

2.3.3.1.1 Eliminación del cálculo de los números aleatorios

En primer lugar, se ha eliminado el cálculo de los números aleatorios de cada iteración. Los números aleatorios son necesarios para seleccionar los puntos que cada ajuste a un círculo (o una elipse) va a utilizar. Sin embargo, es posible evitar el cálculo de los mismos en cada iteración. Para ello se generan los números aleatorios una única vez al inicio, generando valores entre 0 y 1. Después, en cada iteración se escalan los números aleatorios para que se correspondan con el número total de puntos encontrados. La aleatoriedad sigue garantizada ya que en cada iteración se encontrarán puntos diferentes. Además, se realiza una gran cantidad de ajustes que cubre un gran número de combinaciones aportando suficiente variedad. Esta optimización consigue la mayor reducción del tiempo de ejecución de todas las implementadas, reduciéndolo en más de un 90%. El tiempo de ejecución de esta y las otras optimizaciones del algoritmo RANSAC se muestra en la tabla 2.8.

2.3.3.1.2 Kernels fusionados

A continuación, se ha fusionado el kernel que calcula la distancia entre cada punto y cada ajuste junto al kernel que lleva a cabo la reducción de los votos obtenidos por cada ajuste. De esta forma se evitan almacenamientos y cargas innecesarias a memoria global. Obteniendo una pequeña reducción en el tiempo de ejecución como se ve en la tabla 2.8.

2.3.3.1.3 Streams

Seguidamente se implementa el uso de streams CUDA para agilizar la ejecución de los distintos kernels. Este algoritmo conlleva la ejecución de muchos kernels pequeños y varias copias de memoria de la GPU al *host* incrementando la sobrecarga que pueden generar los lanzamientos de los kernels y las llamadas a operaciones de transferencia de datos. Si bien la mejora obtenida no es muy significativa, es importante implementar esta optimización para que una vez se integre este algoritmo con los otros todos sean capaces de lanzar su trabajo en *streams* reduciendo las sobrecargas o habilitando la ejecución paralela de varios *kernels* y/o copias de datos si las operaciones a ejecutar lo permiten.

2.3.3.1.4 Fast math

Además, se han sustituido operaciones matemáticas de un alto coste computacional (como la `sqrt`) por otras más rápidas (como `__frsqrt_rn`). Asimismo, se ha habilitado

la optimización de las operaciones matemáticas mediante el flag `-use_fast_math`. Como resultado se obtiene una pequeña reducción en el tiempo de ejecución debido al poco peso de las operaciones optimizadas en el tiempo de ejecución global.

2.3.3.1.5 Reducción de las comunicaciones entre la GPU y el host

Finalmente, se han reducido las comunicaciones entre la GPU y el *host*. En la etapa final de la ejecución, el algoritmo RANSAC calcula la posición del mejor ajuste a un círculo (o elipse) y a continuación se copia dicho círculo. Ello conlleva dos copias y sincronizaciones de la GPU al *host*: una para conocer la posición del círculo(o elipse) y otra para copiar el círculo (o elipse). Sin embargo, si el kernel que se encarga de buscar la posición del mejor ajuste copia dicho círculo (o elipse) a una posición conocida se puede evitar copiar el índice del mejor ajuste y copiar directamente el círculo (o elipse). Ésto ha obtenido unos resultados muy buenos reduciendo significativamente el tiempo de ejecución.

Implementación	Tiempo de ejecución (ms)
Inicial	3.926ms
Precálculo de números aleatorios	0.126ms
Fusión de kernels	0.091ms
Streams	0.090ms
Fast math	0.089ms
Reducción comunicaciones entre GPU y host	0.085ms

Tabla 2.8: Tabla que muestra los tiempos de ejecución del algoritmo RANSAC con las distintas optimizaciones aplicadas.

También se ha intentado optimizar la tarea utilizando las funciones `shuffle`, `ballot` y `popc` como se hizo en la implementación probada en una NVIDIA 980 GTX [36]. Sin embargo, en esta ocasión no se ha obtenido ninguna mejora del rendimiento sobre la versión que utiliza la reducción con operaciones atómicas.

2.3.4 Umbralización y etiquetado

El algoritmo de umbralización y etiquetado se basa en una aproximación sencilla al seguimiento de pupila que, a pesar de su simplicidad, es utilizada por muchas aplicaciones debido a su efectividad en determinados sistemas. Se utiliza el preprocesado descrito en la subsección 2.3.1 para eliminar los reflejos corneales antes de empezar con la ejecución de

la umbralización y etiquetado. Se utiliza la versión optimizada pero sin eliminar el paso final de la interpolación de los reflejos corneales.

Debido a la complejidad para seleccionar el umbral adecuado en un conjunto amplio de imágenes, hemos optado por un método iterativo que prueba varios umbrales. Es posible utilizar métodos automáticos de selección de umbrales que funcionan bien en muchos casos pero incluso esos métodos fallan a veces en imágenes con una variabilidad tan alta. Por ello, se ha decidido probar varias configuraciones para el bucle iterativo para medir su rendimiento y precisión. En concreto, se prueban varios valores para el incremento en cada iteración del bucle.

El algoritmo 2 muestra detalladamente el funcionamiento del algoritmo de umbralización y etiquetado con unos valores de inicio, fin y de incremento de ejemplo. El pseudocódigo que muestra el algoritmo permite entender rápidamente el funcionamiento iterativo del algoritmo; la umbralización y etiquetado de la imagen que se lleva en cada iteración y la selección del mejor ajuste de la pupila utilizando el número de votos como criterio para ello.

Algoritmo 2 Pseudocódigo del algoritmo de Umbralización y Etiquetado

```

1: procedure BÚSQUEDAPUPILA(Imagen)
2:   umbral  $\leftarrow$  20
3:   paso  $\leftarrow$  20
4:   mejorPupila  $\leftarrow$  []
5:   Imagen  $\leftarrow$  Preprocesado(Imagen)
6:   while umbral < 100 do
7:     ImagenUmbralizada  $\leftarrow$  Umbralizar(Imagen, umbral)
8:     ImagenEtiquetada  $\leftarrow$  Etiquetar(ImagenUmbralizada)
9:     EtiquetaSeleccionada  $\leftarrow$  buscarEtiqueta(ImagenEtiquetada)
10:    ImagenSeleccionada  $\leftarrow$  selecEtiqueta(ImagenEtiquetada, EtiquetaSeleccionada)
11:    Bordes  $\leftarrow$  buscaBordes(ImagenSeleccionada)
12:    NuevaPupila  $\leftarrow$  RANSAC(Bordes)
13:    if votos(NuevaPupila) > votos(mejorPupila) then
14:      mejorPupila  $\leftarrow$  NuevaPupila
15:    end if
16:    umbral  $\leftarrow$  umbral + paso
17:  end while
18:  return mejorPupila
19: end procedure

```

La umbralización se ha paralelizado utilizando una implementación directa en la que

cada hilo procesa un píxel de entrada y de salida y lleva a cabo la comparación. Esta implementación, a pesar de su sencillez, obtiene un rendimiento muy bueno y su peso sobre el tiempo total de ejecución es muy bajo. Se muestra un ejemplo del resultado de esta etapa en la figura 2.7a.

La etapa del etiquetado es significativamente más compleja de paralelizar de forma eficiente y por ello se ha optado por utilizar una implementación de alto rendimiento desarrollada por Chen et al. [37]. Como resultado de la operación de etiquetado se obtiene una imagen similar a la umbralizada pero con valores distintos para cada *blob* de la imagen, (en la figura 2.7b se puede ver un ejemplo). Después se selecciona la etiqueta más grande de la imagen, para ello se calcula el histograma de la misma y, a continuación, se busca el mayor valor en el mismo. El histograma se calcula utilizando operaciones atómicas, cada hilo comprueba el valor de un píxel y si contiene una etiqueta incrementa el valor de la misma en el histograma. A continuación, mediante la librería *thrust* se obtiene el máximo del histograma para a continuación extraer los bordes de dicha etiqueta. En la figura 2.7c se muestra destacada el área etiquetada de mayor tamaño de la imagen. Durante la extracción de los bordes se lanza un hilo por cada píxel de la imagen que comprueba si se trata de un borde o no y, en caso afirmativo, añade el punto a una lista de puntos del borde. Para generar la lista de puntos también se utiliza una operación atómica que mantiene un contador de cuántos puntos se han añadido a la lista. La figura 2.7d muestra el resultado final donde se muestra el borde del área seleccionada.

Normalmente los puntos del borde encontrado pertenecerán en su totalidad al borde de la pupila, pero en ocasiones habrá bordes de otros elementos del ojo. Por ello se utiliza el algoritmo RANSAC que permite obtener un ajuste a la pupila más realista que si se realizara un ajuste utilizando todos los puntos detectados. Una explicación detallada del algoritmo RANSAC, su implementación y optimizaciones se encuentra en la sección 2.3.3.

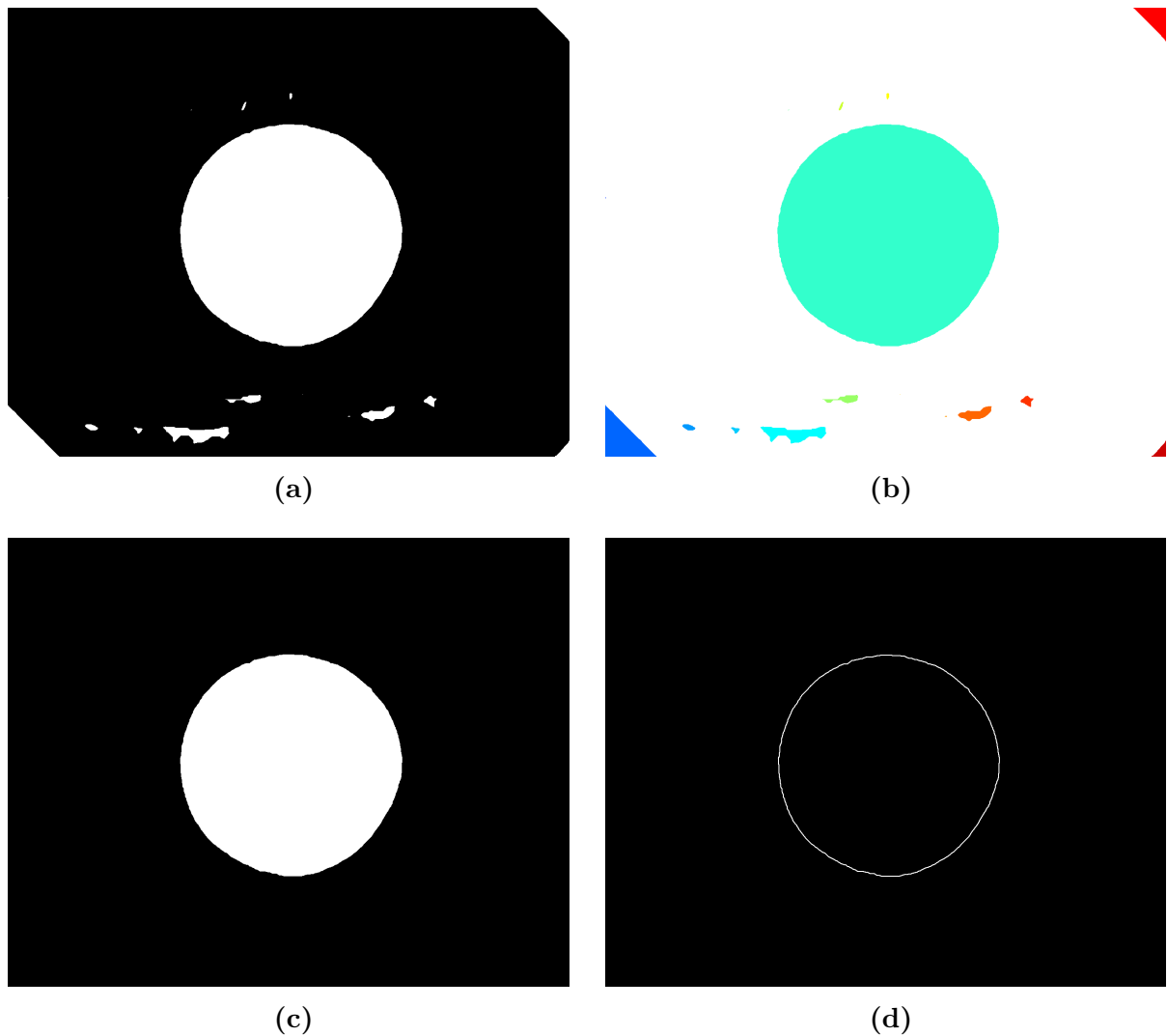


Figura 2.7: (a) Imagen de un ojo umbralizada mostrando la pupila en el centro y algunas pestañas, párpados y artefactos. (b) Imagen etiquetada con cada *blob* detectado en un color diferente. (c) Área etiquetada de mayor tamaño. (d) Se ha detectado el borde del área seleccionada.

2.3.5 Starburst paralelo

El algoritmo original Starburst, desarrollado por Li et al. [23], es un algoritmo secuencial que utiliza un método iterativo para converger al centro de la pupila y encontrar puntos en el borde de la misma. Este algoritmo, como casi todos los desarrollados para la búsqueda de pupila en imágenes tomadas con iluminación infrarroja, basa su funcionamiento en el

contraste que se genera entre la pupila (oscura) y el iris (claro). El algoritmo Starburst espera que el cambio de intensidad (gradiente) entre los píxeles del borde de la pupila sea el mayor de toda la imagen. Si bien, es razonable esperar que haya otros lugares de la imagen donde se encuentren gradientes de valor similar pero en menor cantidad que en el borde de la pupila. Dado que es previsible encontrar algo de ruido se emplea el método RANSAC 2.3.3 para reducir su efecto en el resultado.

El algoritmo 3 muestra un esquema de alto nivel del funcionamiento del algoritmo Starburst secuencial. Tal y como se ve en dicho algoritmo, antes de empezar se eliminan los reflejos corneales, lo cuál se lleva a cabo mediante el método expuesto en la sección 2.3.1. Una vez eliminados los reflejos corneales comienza la búsqueda de la pupila propiamente dicha, la cuál parte de una posición inicial, que puede ser el centro de la imagen o la posición de la pupila en el *frame* anterior. Desde esa posición se analizan los píxeles en una serie de líneas distribuidas de forma uniforme en 360° que, a modo de rayos, recorren la imagen desde el centro hasta los bordes de la misma. Mientras se recorren los píxeles de los rayos se va calculando su gradiente y se compara su valor con un umbral previamente establecido. Si el valor del gradiente es mayor que el umbral, se detiene la búsqueda en ese rayo y se almacena el resultado. El objetivo de esta búsqueda es encontrar puntos en el borde de la pupila: puntos en los cuales el gradiente suele presentar un valor elevado al haber un cambio significativo entre el negro de la pupila y el gris del iris. En la figura 2.8 se puede ver visualmente el método explicado.

En una segunda etapa se realiza un proceso similar. En esta ocasión, los puntos de inicio son los puntos encontrados en la etapa anterior. Además, ahora no se lanzan rayos a 360° sino solamente en un cono de 90° en dirección al centro inicial. Este segundo paso tiene la labor de reforzar los resultados del primero, de forma que los puntos detectados previamente en el borde de la pupila permiten encontrar nuevos puntos en el borde de la pupila con muy alta probabilidad, como se ve en la figura 2.8b. Fíjese que incluso los puntos encontrados fuera de la pupila en la primera etapa pueden encontrar puntos en el borde de la pupila en la segunda etapa, como se muestra en la figura 2.8c. Una vez terminada la segunda fase se fusionan los puntos de las dos fases y se ejecuta el algoritmo RANSAC mostrado en la sección 2.3.3. Como resultado se obtiene el mejor ajuste de un círculo/elipse a los puntos encontrados, se compara la posición del mismo (la figura 2.8d muestra la posición del ajuste en azul) con el centro inicial, y si la distancia entre los dos es menor que un umbral se determina que el algoritmo ha convergido, en caso negativo se ejecuta una nueva iteración. La figura 2.8e muestra el resultado de ejecutar otra iteración,

como se puede ver en esta ocasión el centro del ajuste casi no se ha movido y se concluye que el algoritmo ha convergido.

Algoritmo 3 Pseudocódigo del algoritmo Starburst secuencial

```

1: procedure BÚSQUEDAPUPILA(Imagen, CentroPrevio)
2:   Iteración  $\leftarrow$  0
3:   Imagen  $\leftarrow$  Preprocesado(Imagen)
4:   Distancia  $\leftarrow$   $\infty$ 
5:   while Iteración < 10 & Distancia > 10 do
6:     Puntos1  $\leftarrow$  BuscaPuntos1(Imagen, CentroPrevio)
7:     Puntos2  $\leftarrow$  BuscaPuntos2(Imagen, CentroPrevio, Puntos1)
8:     PuntosUnidos  $\leftarrow$  UnirPuntos(Puntos1, Puntos2)
9:     Pupila  $\leftarrow$  RANSAC(PuntosUnidos)
10:    CentroNuevo  $\leftarrow$  Centro(Pupila)
11:    Distancia  $\leftarrow$  Distancia(CentroPrevio, CentroNuevo)
12:    CentroPrevio  $\leftarrow$  CentroNuevo
13:  end while
14:  return Pupila
15: end procedure

```

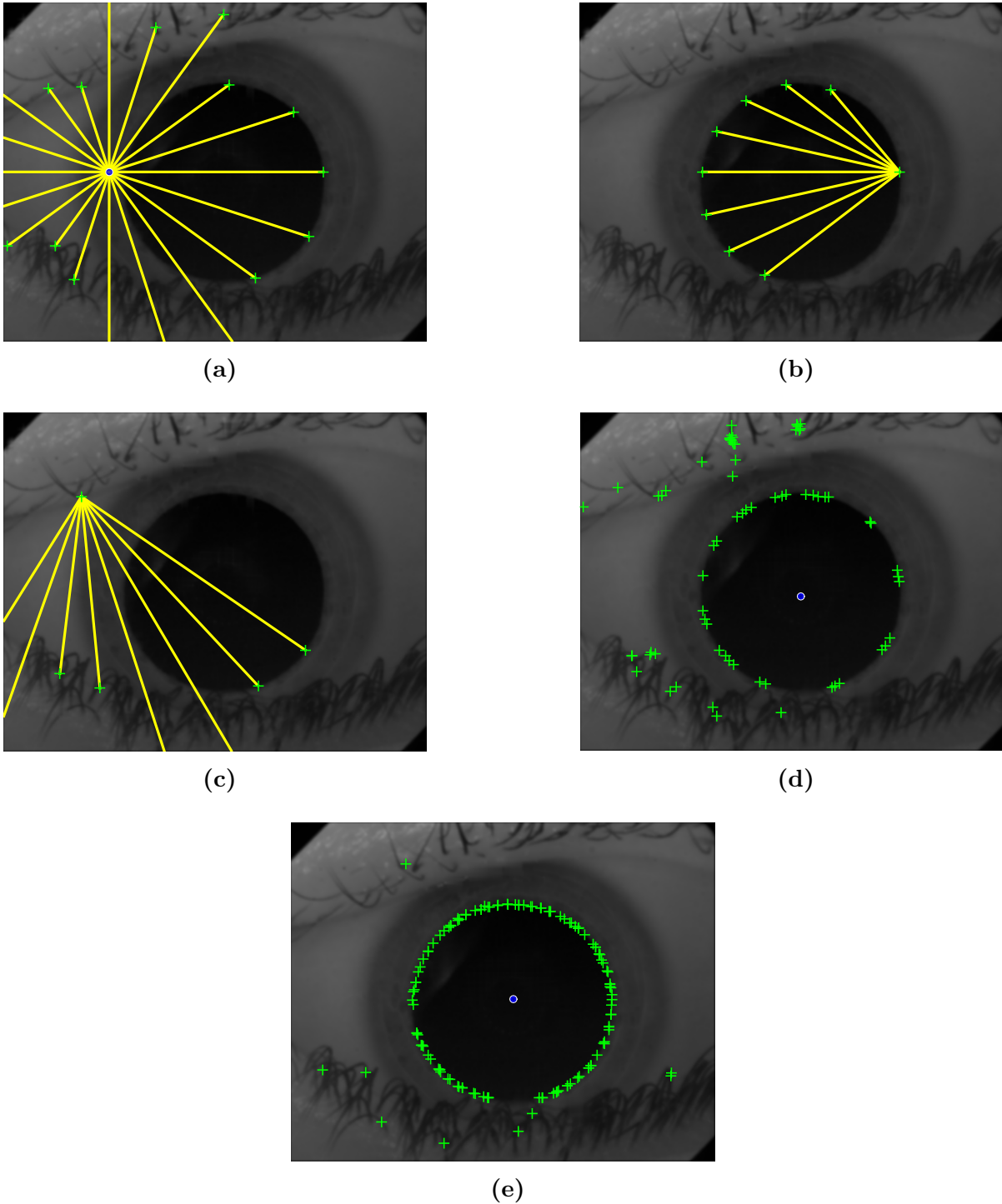


Figura 2.8: (a) Se lanzan rayos en todas direcciones desde el centro (punto azul). (b) Se lanzan rayos desde los puntos encontrados en la primera etapa. (c) Se lanzan más rayos desde los puntos encontrados en la primera etapa. (d) Con todos los puntos encontrados se ejecuta RANSAC y se encuentra un círculo/pupila con centro en el punto azul. (e) Se repite el proceso desde el nuevo centro y algoritmo converge.

Al optimizar el algoritmo Starburst el tiempo de ejecución del mismo ha disminuido drásticamente, hasta tal punto que el peso de la computación es comparable al de la copia de los datos de la CPU a la GPU (como muestra la figura 2.17). Es por ello que se ha planteado una nueva optimización que se ha denominado procesado con *pipelining*, y que consiste en el procesado en solapado/paralelo de varias imágenes. Además, se han utilizado *streams* de CUDA para permitir la ejecución en paralelo del procesado de diferentes imágenes. Al usar varios *streams* es posible solapar la copia de datos con el procesamiento de los mismos y, como resultado, ocultar el tiempo que se emplea en copiar los datos. De esta forma es posible reducir drásticamente el tiempo de ejecución global en casos como éste, en el que la computación y la copia tienen un coste similar. La figura 2.9 muestra un esquema de Gantt con un ejemplo de la concurrencia entre las copias de memoria y la ejecución de los kernels. Si bien los kernels también pueden ejecutarse en paralelo el beneficio obtenido suele ser menor debido a que requiere que haya recursos hardware disponibles. Además, se ve como dos copias se pueden realizar también en paralelo si son en direcciones opuestas, esto es, una del host a la GPU y otra de la GPU al host. Aunque, en esta aplicación concreta la cantidad de datos copiada de la GPU al host es muy pequeña, y el tiempo que se emplea en llevarla a cabo también lo es en comparación con el resto de operaciones.

Esta optimización es de especial interés para operaciones de postprocesado en las que se quieran analizar grandes cantidades de datos capturados con anterioridad. En ese caso, se dispone de todas las imágenes y no es necesario esperar a que la cámara las capture. También se puede utilizar en aplicaciones de tiempo real que no requieran actuar después de procesar cada imagen o que admitan un cierto retraso entre la captura y la actuación. Aunque este último caso no es muy común ya que es más complejo interpretar la respuesta que la actuación está generando.

Para llevar a cabo la implementación con *pipelining* ha sido necesario mover el control del bucle principal del algoritmo *Starburst* a la GPU. De forma que desde el host solamente se ejecutan tres llamadas para procesar cada imagen: la copia de la imagen a la GPU, el lanzamiento del kernel de procesamiento y la copia del resultado al host. El kernel que se lanza desde el host es el que se encarga de ejecutar y sincronizar todos los kernels. Lanzar kernels desde la GPU es más lento que hacerlo desde el host y sólo es beneficioso en casos en los que permite un uso más eficiente de los recursos. Normalmente ajustando la cantidad de hilos que se lanzan al trabajo disponible, que no es conocido previamente. O como en este caso, porque al mover esa tarea a la GPU se pueden procesar varias imágenes

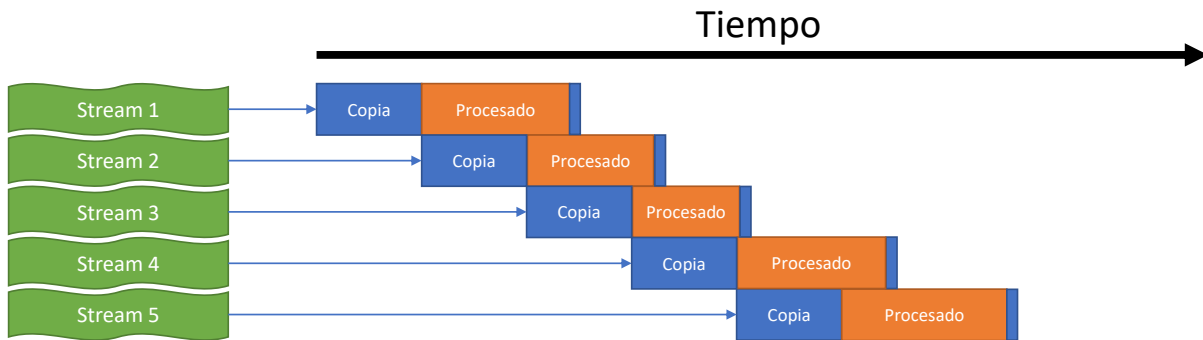


Figura 2.9: Diagrama de Gantt que representa la ejecución del procesamiento usando la técnica de pipelining mediante streams.

simultáneamente porque se han eliminado las sincronizaciones en el host. Sin embargo, si se compara el tiempo de procesamiento de una sola imagen de esta implementación y la implementación que se controla desde el host se observará una caída del rendimiento. Dicha caída del rendimiento sólo se ve compensada una vez que se aumenta el paralelismo aumentando la cantidad de streams que se ejecutan en paralelo.

2.3.6 Detección de bordes de Canny y Transformada de Hough

El tercer algoritmo implementado y paralelizado consiste en la utilización de dos algoritmos ampliamente utilizados en el procesamiento de imágenes: la detección de bordes de Canny [27] y la transformada de Hough [28].

El algoritmo 4 muestra un esquema de alto nivel del funcionamiento de este algoritmo. En primer lugar, se somete la imagen a la etapa de preprocesado para eliminar los reflejos y reducir el ruido en la misma. A continuación, se procesa la imagen con el método de detección de bordes de Canny que es capaz de detectar con alta precisión los bordes presentes en la imagen. En este caso nos interesa detectar el borde entre la pupila y el iris. Aunque también aparecerán en las imágenes los bordes de las pestañas y los párpados, así como los de otros elementos de menor tamaño. El método de Canny consta de varios pasos: suavizado de la imagen; cálculo del gradiente de la imagen; supresión no máxima del gradiente para eliminar los bordes débiles; umbralización doble (bajo y alto); y finalmente, un paso de histéresis.

Algoritmo 4 Pseudocódigo del algoritmo de detección de bordes de Canny y Transformada de Hough

```

1: procedure BÚSQUEDAPUPILA(Imagen)
2:   Imagen  $\leftarrow$  Preprocesado(Imagen)
3:   Bordes  $\leftarrow$  Canny(Imagen)
4:   MejorPupila  $\leftarrow$  []
5:   radio  $\leftarrow$  1
6:   while radio < 5 do
7:     Pupila  $\leftarrow$  Hough(Bordes)
8:     if votos(Pupila) > votos(MejorPupila) then
9:       MejorPupila  $\leftarrow$  Pupila
10:    end if
11:    radio  $\leftarrow$  radio + 1
12:  end while
13:  return MejorPupila
14: end procedure

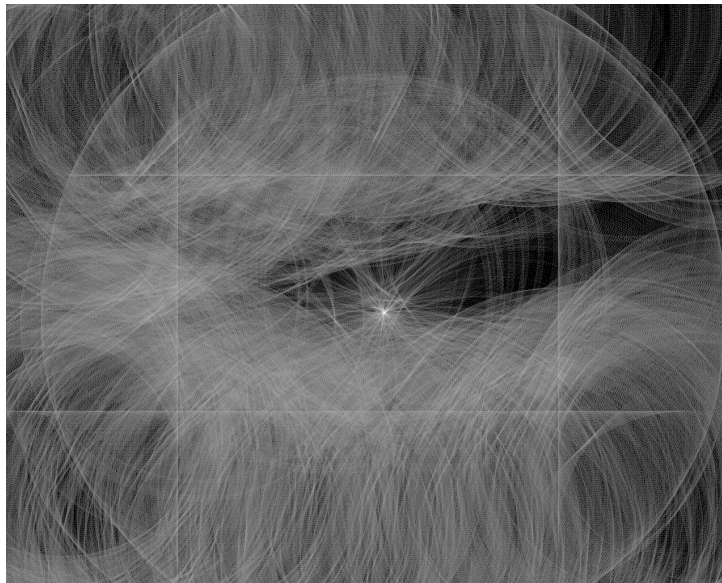
```

El suavizado de la imagen se lleva a cabo en x e y por separado, aplicándose para ello un filtro de Gauss. Por otra parte, para la paralelización del mismo se lanzan tantos hilos CUDA como píxeles haya en la imagen, encargándose cada hilo de procesar un píxel de salida. Para el cálculo del gradiente se repite la misma estrategia de paralelización, lanzando un hilo CUDA por cada píxel de salida, calculando cada hilo el valor del gradiente en su posición y la dirección del mismo. La supresión no máxima vuelve a seguir el mismo patrón de paralelización y lanza un hilo CUDA por cada píxel de salida. En este caso, la tarea es algo más compleja pues cada hilo debe tener en cuenta la dirección del gradiente antes de comprobar si se trata de un eje débil o no. A continuación se lleva a cabo la umbralización doble, que una vez más utiliza la misma estrategia de paralelización lanzando un hilo CUDA por cada píxel de salida. Cada hilo comprueba si su píxel es un borde o no y, en caso de serlo, lo clasifica en borde débil o borde fuerte. Finalmente, la etapa de histéresis determina si los bordes débiles deben ser descartados o conservados. Para ello se tiene en cuenta si alguno de sus ocho vecinos es un borde fuerte, situación en la que se considera que es un borde fuerte. En este caso también se emplea un hilo CUDA por cada píxel de salida. Una vez obtenida la imagen con los bordes se almacenan en un vector las posiciones de las mismas con la intención de optimizar la siguiente fase. Como

resultado se obtienen los bordes de la imagen tal y como se muestra en la figura 2.10a.



(a)



(b)

Figura 2.10: (a) Bordes detectados por el algoritmo de Canny. (b) Votos Obtenidos por la transformada de Hough. Se muestran en escala logarítmica para facilitar la visualización.

En último lugar se ejecuta la transformada de Hough, la cuál se debe ejecutar con varios

radios, aunque en el algoritmo 4 se muestra como un proceso iterativo, en la versión paralela el cálculo de todos los radios se realiza en paralelo. La entrada de la transformada de Hough es el vector con las posiciones de los bordes detectados. A partir de ellos se generan votos en una serie de posiciones que se encuentran a un radio de distancia. La cantidad de votos que genera cada posición del borde dependerá de la precisión seleccionada para la transformada de Hough. Una vez se ha completada la etapa de "votación" se busca el ajuste con más votos que es el seleccionado como la mejor pupila. En la figura 2.10b se muestra un ejemplo del resultado (logarítmico para facilitar la visualización) de la transformada de Hough, se puede ver claramente como hay un punto brillante en el centro que se corresponde en este caso con el centro de la pupila.

2.3.7 Implementación OpenCL

Si bien en casi toda esta Tesis se ha utilizado CUDA como lenguaje de programación para paralelizar el código que se ejecuta en las GPUs también se ha utilizado OpenCL puntualmente [4]. En concreto, se ha implementado una versión OpenCL del algoritmo Starburst utilizando el filtro de Gauss y el ajuste de una elipse. La principal ventaja de OpenCL sobre CUDA es la posibilidad de ejecutar el mismo código sobre distintos dispositivos (CPU, GPU, ...) y sobre dispositivos de distintos fabricantes. Eso otorga una gran flexibilidad a las implementaciones en OpenCL que pueden ejecutarse sobre el mejor hardware disponible en todo momento. Sin embargo, el soporte, la cantidad de librerías disponibles y la cantidad de herramientas de desarrollo disponibles para CUDA es superior a las disponibles para OpenCL.

La implementación OpenCL desarrollada ha sido optimizada de forma idéntica a la implementación CUDA: uso de memoria compartida, filtros separables, fusión de kernels, tipos de datos optimizados, supresión de la interpolación de los reflejos corneales, *pinned memory*, supresión del cálculo de números aleatorios en cada iteración, uso de colas de OpenCL y reducción de copias de memoria entre la GPU y la CPU. Sin embargo, algunas optimizaciones no se han podido aplicar, como el desenrollado de bucles mediante plantillas por la ausencia de soporte para plantillas en OpenCL.

Como resultado se obtiene un rendimiento algo inferior: si la versión CUDA con un stream y el ajuste a una elipse consigue procesar 847 imágenes/s, la versión OpenCL con la misma configuración consigue procesar 588 imágenes/s.

2.4 Resultados

La evaluación se ha centrado en medir el rendimiento y la precisión de las diferentes implementaciones desarrollados con el objetivo de determinar cuál de ellas ofrece la mejor relación rendimiento/precisión. A pesar de que la definición de la mejor relación dependerá de los requerimientos de uso de cada aplicación concreta es posible establecer un criterios generales que ofrezcan un resultado válido para la mayoría de aplicaciones.

Para medir la precisión se ha empleado una fórmula que tiene en cuenta el error en la posición y en el tamaño de la pupila detectada. Ambos errores se calculan en relación al tamaño de la pupila de cada imagen, pues no es lo mismo un error de 0.1mm en una pupila de 2mm de diámetro que en una de 8mm de diámetro. El error viene definido por la ecuación 2.1.

$$Error = \text{máximo}\left(\frac{Error\ posición}{Radio\ de\ la\ pupila}, \frac{Error\ Radio}{Radio\ de\ la\ pupila}\right) \quad (2.1)$$

Así, cuando se muestren los resultados de precisión se hará de forma acumulativa, mostrando sucesivamente el porcentaje de las imágenes que se han procesado con errores que van desde el 1% hasta el 20%.

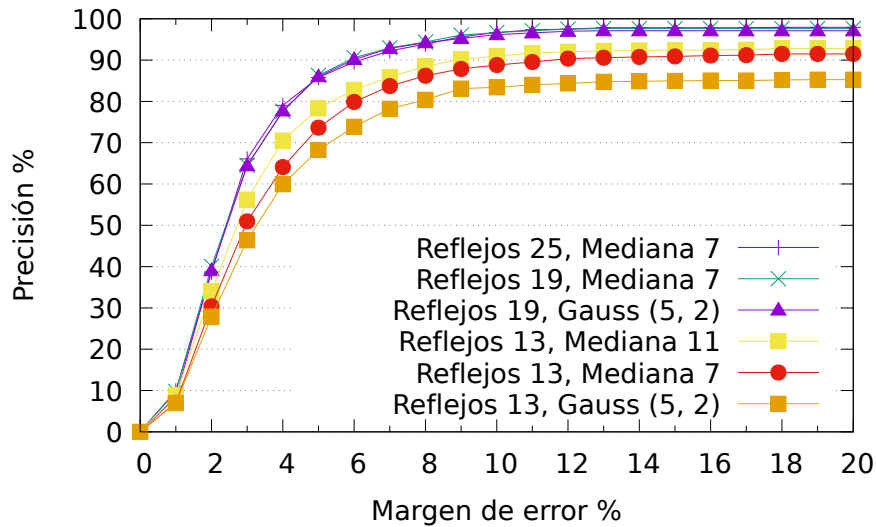
Por otra parte, es importante destacar que los resultados de rendimiento incluyen en todo momento el tiempo de copia de las imágenes de la CPU a la GPU, puesto que en una situación real ese tiempo no se puede obviar. Para obtener los resultados de rendimiento se han procesado las imágenes presentadas en la sección 2.2. El tiempo de ejecución se acumula mientras dura el procesado y finalmente se calcula la media del tiempo de ejecución.

Para obtener los datos de rendimiento en la GPU se ha utilizado un procesador gráfico moderno de altas prestaciones, en concreto, se ha empleado una **NVIDIA GTX 1080 Ti**, que cuenta con 3584 núcleos CUDA y 11GB de memoria que proporcionan 484GB/s de ancho de banda. Se trata por tanto de un procesador gráfico muy potente, si bien las series 2000 y 3000 de NVIDIA presentan un rendimiento sensiblemente superior. Asimismo, los tests se han ejecutado con una versión muy moderna de CUDA, la versión 10.1, lo cuál garantiza que se está obteniendo el mejor rendimiento posible con las herramientas utilizadas. En cuanto a la CPU se ha utilizado una CPU Intel i5-4690 (que trabaja hasta 3.9GHz) y cuenta con 4 núcleos / 4 hilos, además, cuenta con 16GB de RAM DDR3 a 1600MHz. Y el compilador utilizado ha sido gcc-7 con los flags -O3 y -fopenmp.

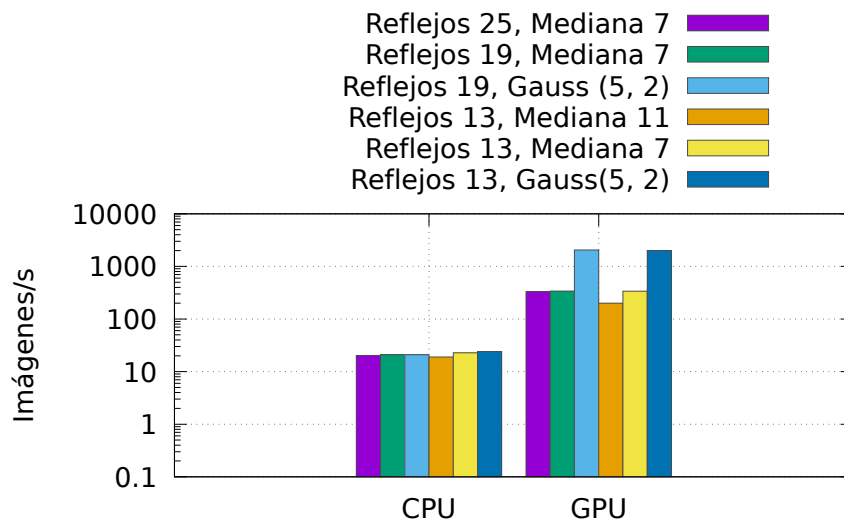
La figura 2.11 muestra los resultados de las distintas ejecuciones del algoritmo *Starburst*. Para este algoritmo se ha evaluado, en primer lugar, su rendimiento tanto en la CPU como en la GPU. Además, se ha evaluado su rendimiento y precisión con varias configuraciones del preprocesamiento, al cuál es muy sensible. Así en la figura 2.11a se muestra la precisión obtenida por cada una de las configuraciones que se agrupan de la siguiente forma:

- Mediana:
 - Reflejos 25x25, Mediana 7x7
 - Reflejos 19x19, Mediana 7x7
 - Reflejos 13x13, Mediana 11x11
 - Reflejos 13x13, Mediana 7x7
- Gauss
 - Reflejos 19x19, Gauss 5x5
 - Reflejos 13x13, Gauss 5x5

Hay dos tipos de preprocesado, como se explica en la sección 2.3.1. Uno utiliza un filtro de la mediana para el suavizado y el otro utiliza un filtro de Gauss. Ambos consiguen muy buenos resultados en lo que a precisión se refiere en algunas de sus configuraciones, sin embargo, las diferencias son mayores en términos de rendimiento. Por otro lado, en la figura 2.11b se muestra el rendimiento obtenido por cada una de las configuraciones. El rendimiento obtenido por las implementaciones secuenciales en la CPU es muy discreto, especialmente cuando se compara con los resultados obtenidos en la GPU. Todas las configuraciones superan las 200 imágenes/s, pero las configuraciones que incluyen el filtro de Gauss llegan hasta las 2000 imágenes/s. Aunque esas velocidades solo se logran utilizando varios *streams* de CUDA para copiar y procesar varias imágenes a la vez, como se muestra en la figura 2.12b. A pesar de ello, el resultado es extremadamente bueno y la velocidad obtenida es más que suficiente para nuestras aplicaciones. Hay que tener en cuenta, que las cámaras no suelen alcanzar velocidades tan altas o no tienen suficiente sensibilidad para formar imágenes correctamente tan rápido. Aunque sí es útil tener mucha velocidad si se quiere hacer un postprocesado de datos ya capturados o en situaciones en las que el seguimiento pupilar no es la única tarea a realizar en tiempo real.



(a)



(b)

Figura 2.11: Resultados de precisión y rendimiento de varias configuraciones del algoritmo Starburst paralelo.

En la figura 2.11 se recogen los resultados obtenidos por varias configuraciones del preprocesado, donde se observa que el tamaño del filtro utilizado para eliminar los reflejos corneales es crítico a la hora de obtener buenos resultados. Todas las configuraciones que utilizan el filtro de 13x13 píxeles obtienen resultados significativamente peores que las que utilizan filtros de 19x19 o 25x25 píxeles. Así, tanto el filtro de la mediana como el de

Gauss ofrecen una precisión muy buena con tamaños grandes para el filtro de eliminación de los reflejos.

Por otra parte, estos resultados utilizan un círculo para llevar a cabo el ajuste de los puntos detectados en el borde de la pupila. Sin embargo, las pupilas humanas presentan formas muy dispares, si bien por regla general se asemejan a un círculo, una mejor aproximación es una elipse. Éstas abarcan un rango mayor de formas y permiten por tanto unos ajustes más precisos de las pupilas. Aunque debido a su mayor complejidad presentan un mayor coste computacional a la hora de realizar el ajuste de los puntos y ejecutar el algoritmo RANSAC.

La figura 2.12a muestra una comparación de la precisión obtenida por el ajuste de la elipse y del círculo con el algoritmo *Starburst*. Los resultados muestran que el ajuste de la elipse ofrece una pequeña mejora en la precisión que será tanto mayor en los sujetos con pupilas más elípticas. Además, es necesario tener en cuenta que en medidas en sujetos con pupilas muy elípticas un ajuste a un círculo puede generar un ruido muy perjudicial para la medida si el ajuste "salta" de un extremo a otro de la pupila. Si bien el error de precisión puede ser más o menos bajo, los constantes cambios pueden hacer imposibles algunas medidas. El problema queda totalmente resuelto con el ajuste a una elipse. Como se ha dicho anteriormente, el coste computacional de calcular el ajuste a una elipse es más elevado que el coste del ajuste a un círculo. La figura 2.12b muestra una comparación del rendimiento de los dos ajustes. Claramente, el rendimiento obtenido por el ajuste a un círculo es superior.

La figura 2.12b también muestra el rendimiento obtenido con la utilización de varios streams. El solapamiento de la computación y de las copias entre la CPU y la GPU ha permitido un mayor aprovechamiento de los recursos de la GPU incrementando de una forma muy importante el rendimiento. Una vez más se ha utilizado el algoritmo *Starburst* para medir el rendimiento de esta optimización. Teniendo en cuenta que cada imagen ocupa 5MB y que la velocidad de copia entre la CPU y la GPU utilizando *pinned memory* es de unos 12GB/s, el rendimiento máximo que se puede alcanzar, si se consigue ocultar totalmente la computación y solo se tiene en cuenta el tiempo de copia de las imágenes, es de 2450 imágenes/s. Lo cuál está muy cerca de los números obtenidos por la implementación que realiza el ajuste a un círculo, que supera ligeramente las **2000 imágenes/s** cuando se emplean 8 *streams*. El ajuste a una elipse presenta una mejora similar al del círculo pero su rendimiento siempre es menor debido a la mayor carga de trabajo. Por otra parte, el rendimiento aumenta de forma significativa al pasar de 1 *stream*

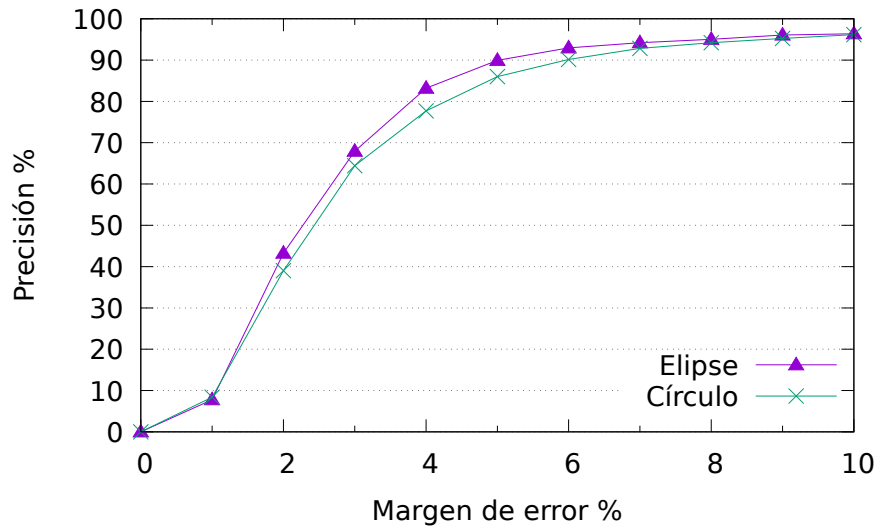
a 4 y ligeramente al aumentar hasta los 8, momento en el cuál el rendimiento ya se ha estabilizado y deja de mejorar debido a la saturación de los recursos disponibles.

También se puede observar en la figura 2.12b que el rendimiento con un solo stream es menor que el obtenido para la implementación sin streams. En la que el control se lleva a cabo desde el host. Esto es debido al mayor coste de lanzar los kernels directamente desde la GPU. Que se ve compensado tan pronto como se aumenta la cantidad de streams ejecutados en paralelo.

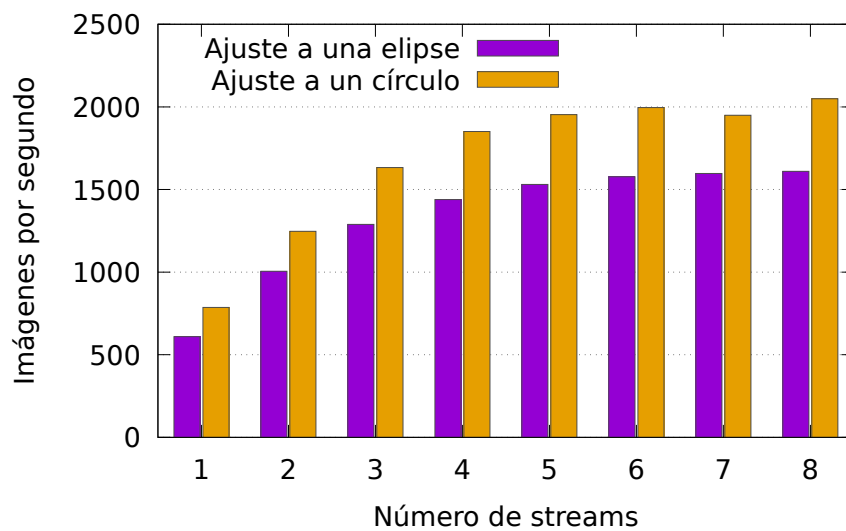
Finalmente, en lo que respecta a los resultados específicos del algoritmo Starburst en la figura 2.13 se muestra la aceleración obtenida tras aplicar cada una de las **optimizaciones** expuestas en las secciones 2.3.1.1, 2.3.2, 2.3.3 a la implementación CUDA. En dicha figura se muestran las 13 optimizaciones implementadas, si bien la ganancia obtenida varía mucho de unas a otras. El resultado final es una aceleración de casi **46x** con respecto a la versión CPU. La implementación utilizada para obtener estos resultados ha sido la que ejecuta los kernels desde el host aplicando las optimizaciones de forma sucesiva y acumulativa. Estos resultados refuerzan los mostrados en las secciones de las optimizaciones, en las que se muestra como se va reduciendo el tiempo de ejecución tras implementarlas.

Un análisis similar al realizado sobre el algoritmo Starburst se ha realizado con los otros dos algoritmos probados. La figura 2.14a muestra la precisión obtenida por el algoritmo de **Umbralización y Etiquetado**, mientras que la figura 2.14b muestra el rendimiento del mismo. Para evaluar convenientemente el algoritmo de Umbralización y Etiquetado se ha variado el tamaño del paso utilizado en las iteraciones del mismo, parámetro que controla la cantidad de umbrales que se prueban y que afecta significativamente a la carga computacional del algoritmo, y por tanto, a su rendimiento y precisión. En lo que a precisión se refiere los mejores resultados se han obtenido con valores de paso de 5 y de 10, mientras que el tamaño de paso de 40 es mucho peor que el resto. Por otra parte, el rendimiento sigue una tendencia inversa a la precisión. Evidentemente, las configuraciones que realizan menos iteraciones obtienen una mayor velocidad y, al contrario, las que ejecutan más iteraciones. La clave está en obtener un buen compromiso entre rendimiento y precisión. Como se observa en la figura, la configuración con un paso de tamaño 20 obtiene dicho buen compromiso entre precisión y velocidad, superando las **100 imágenes/s**. Y una precisión superior al 60% con un margen de error del 5% y una precisión superior al 81% con un margen de error del 10%.

Restan por analizar los resultados obtenidos por el algoritmo que emplea la **detección de bordes de Canny y la Transformada de Hough**. La figura 2.15 muestra los



(a)



(b)

Figura 2.12: Resultados de precisión con dos tipos de ajuste pupilar (círculo o elipse) para el algoritmo Starburst paralelo. Y resultados de rendimiento de cada uno de esos ajustes utilizando una cantidad variable de *streams* CUDA

resultados de precisión y rendimiento de este algoritmo. De igual manera que en los otros dos algoritmos se han probado varias configuraciones. En este caso se ha variado la cantidad de muestras que se toman para cada transformada de Hough. En la figura 2.15b se muestra el rendimiento, que en este caso no es tan elevado como en los otros dos algoritmos debido a la enorme carga computacional que conlleva éste, obteniéndose

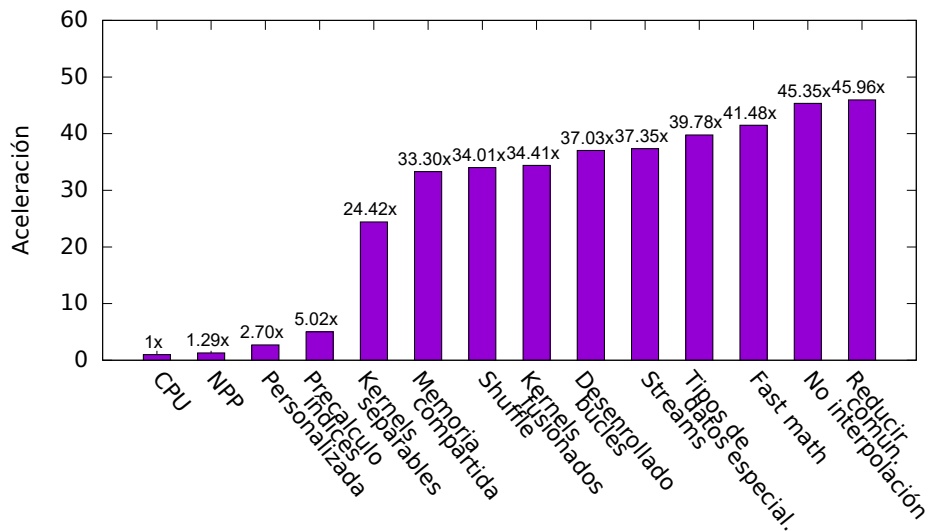


Figura 2.13: Aceleración obtenida por la implementación CUDA usando 1 stream en comparación con la implementación CPU. Se muestra la aceleración obtenida para cada una de las optimizaciones aplicadas a la implementación GPU.

velocidades de unas **30-40 imágenes/s**. En lo que a precisión se refiere los resultados que se muestran en la figura 2.15a son excelentes, especialmente para las configuraciones que tienen una mayor densidad de muestreo que obtienen una precisión de hasta el 96% de las pupilas detectadas correctamente para un margen de error del 10%.

Por último, la figura 2.16 muestra una comparativa entre los tres algoritmos. Dado que cada algoritmo se ha ejecutado con diferentes configuraciones se ha seleccionado la mejor configuración para cada uno, teniendo en cuenta tanto su rendimiento como su precisión: Starburst con filtro de 19x19 para eliminar los reflejos y filtro de Gauss de 5x5; Detección de bordes de Canny y Transformada de Hough con paso de 3.6°; Umbralización y Etiquetado con tamaño de paso de 20. En la figura 2.16a se puede ver que en lo que a precisión se refiere tanto Starburst como la Transformada de Hough ofrecen muy buenos resultados, con más del 96% de las imágenes procesadas con un error por debajo del 10%. Mientras que el algoritmo de Umbralización y Etiquetado es significativamente peor en este apartado, consiguiendo detectar un 81% de las pupilas con un error inferior al 10%. Por otra parte, la figura 2.16b muestra el rendimiento de las tres configuraciones elegidas, se observa que el algoritmo Starburst presenta un rendimiento muy superior a los otros dos algoritmos y, en especial, es muy superior al rendimiento obtenido por la Transformada de Hough, mientras que el Starburst supera las 2000 imágenes/s, la Transformada de Hough

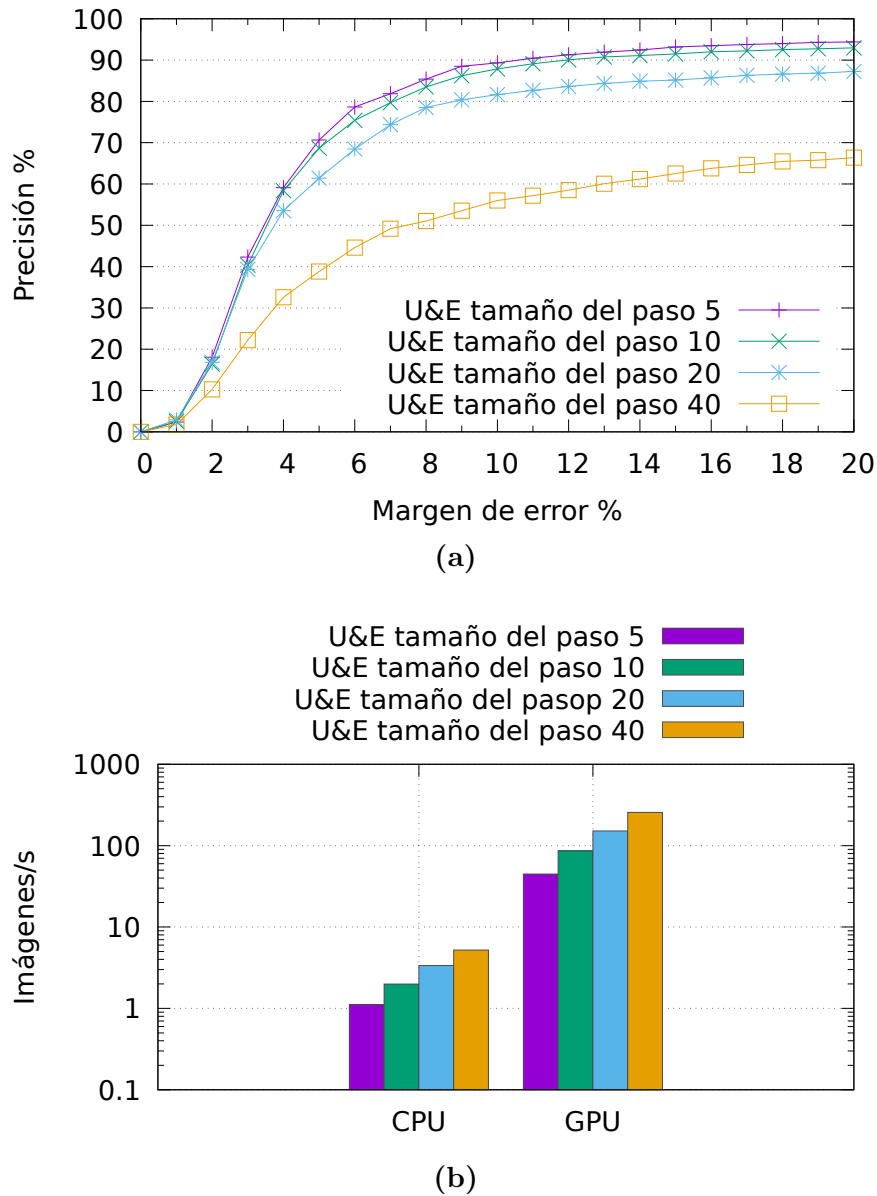
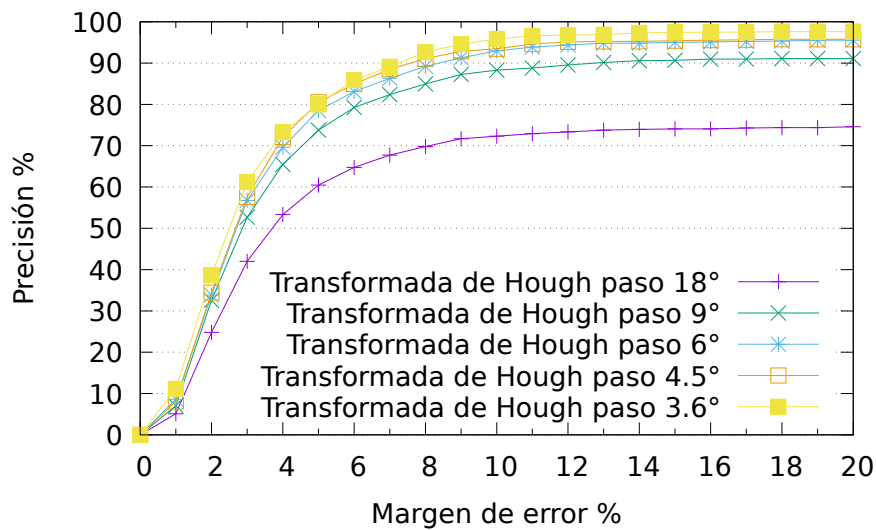
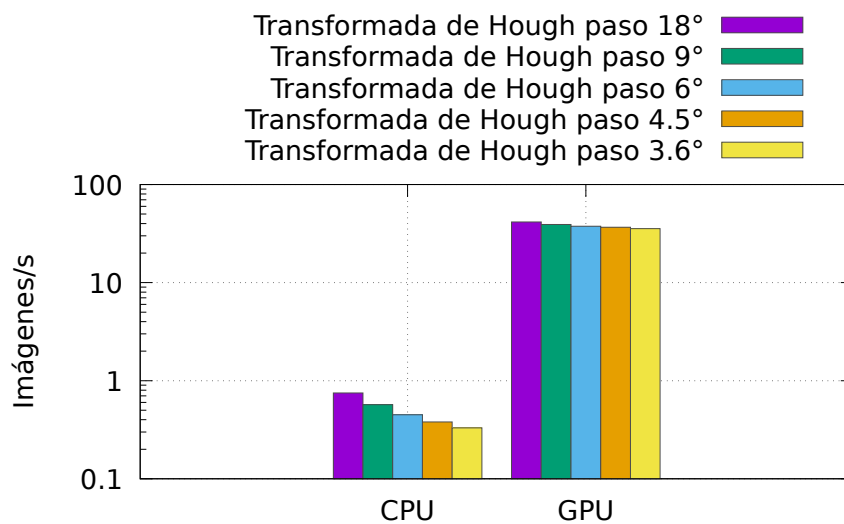


Figura 2.14: Resultados de precisión y de rendimiento para el algoritmo de umbralización y etiquetado para varias configuraciones.

se queda en 30 imágenes/s, y la Umbralización y etiquetado consigue 150 imágenes/s. Por lo tanto, una vez se tiene en cuenta tanto el rendimiento como la precisión **el algoritmo Starburst ofrece sin duda el mejor compromiso** entre los tres. Es por tanto este algoritmo el que hemos utilizado en los diferentes sistemas en los que se ha utilizado la implementación presentada en este capítulo.



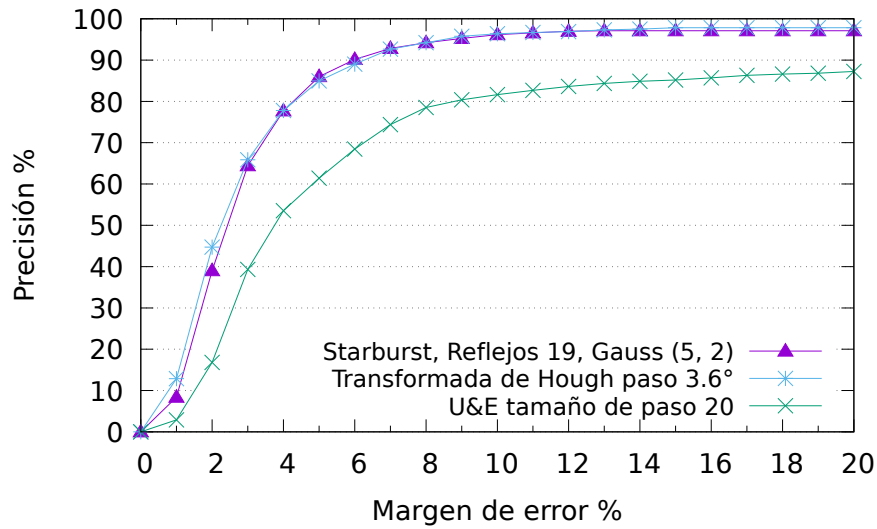
(a)



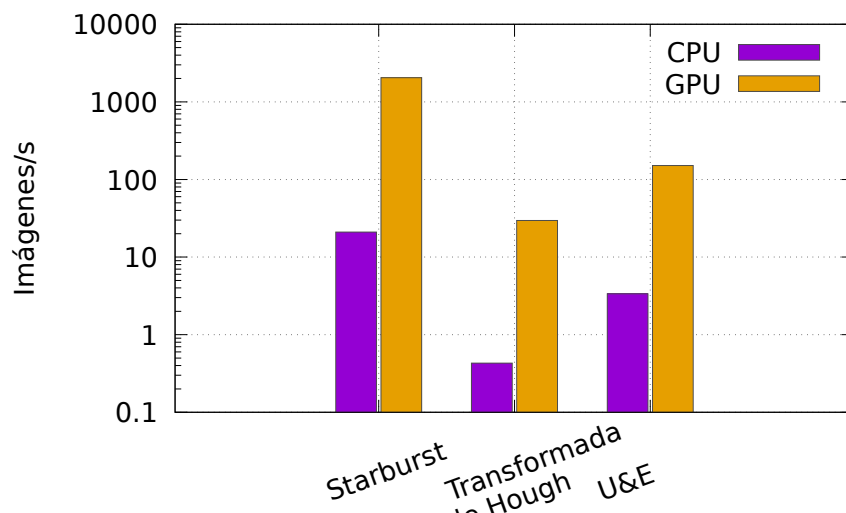
(b)

Figura 2.15: Resultados de precisión y de rendimiento para el algoritmo de Canny y Hough para varias configuraciones.

Con el objetivo de mostrar una visión más detallada del funcionamiento de cada algoritmo se ha añadido la figura 2.17 en la que se desglosan los tiempos de ejecución de cada algoritmo (para el Starburst se trata de la implementación en la que el control se lleva a cabo desde el host). En ella se puede ver el peso que tiene cada etapa para cada algoritmo. Destaca el Starburst por el escaso tiempo que emplea en la etapa de búsqueda



(a)



(b)

Figura 2.16: Comparativa de los resultados de precisión y rendimiento de las mejores configuraciones de cada algoritmo.

de puntos, que contrasta con el algoritmo que utiliza la detección de bordes de Canny y la transformada de Hough y el que usa la Umbralización y Etiquetado. Ambos emplean un tiempo mucho mayor en sus etapas de búsqueda y en las etapas previas. En el caso de la detección de bordes de Canny y la Transformada de Hough, el tiempo de copia y preprocesado es despreciable si se compara con el resto de la ejecución. Mientras que en

el caso de la Umbralización y Etiquetado es la etapa de etiquetado la que emplea aproximadamente el 50% del tiempo de ejecución. Para hacer esta comparación se han utilizado las mismas configuraciones que las usadas en la figura 2.16.

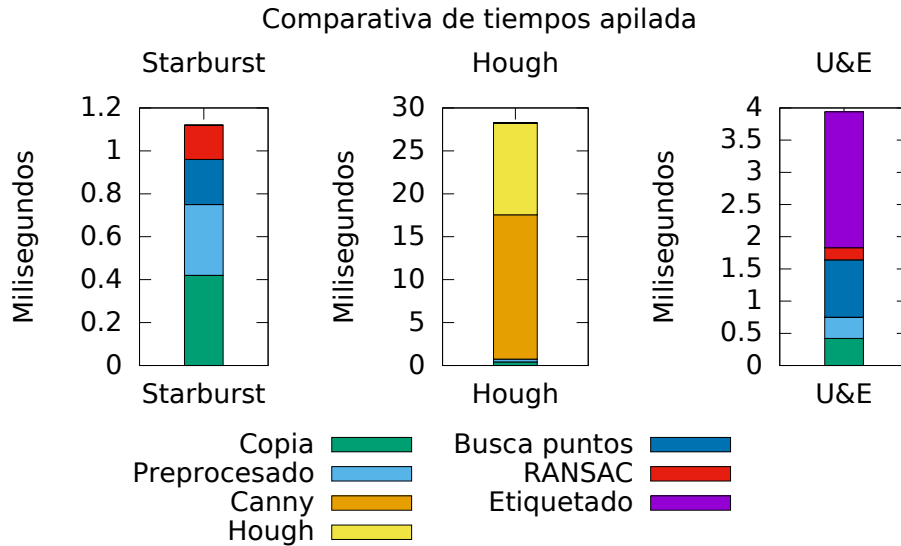


Figura 2.17: Comparativa del tiempo que cada algoritmo emplea en cada una de las partes que lo componen.

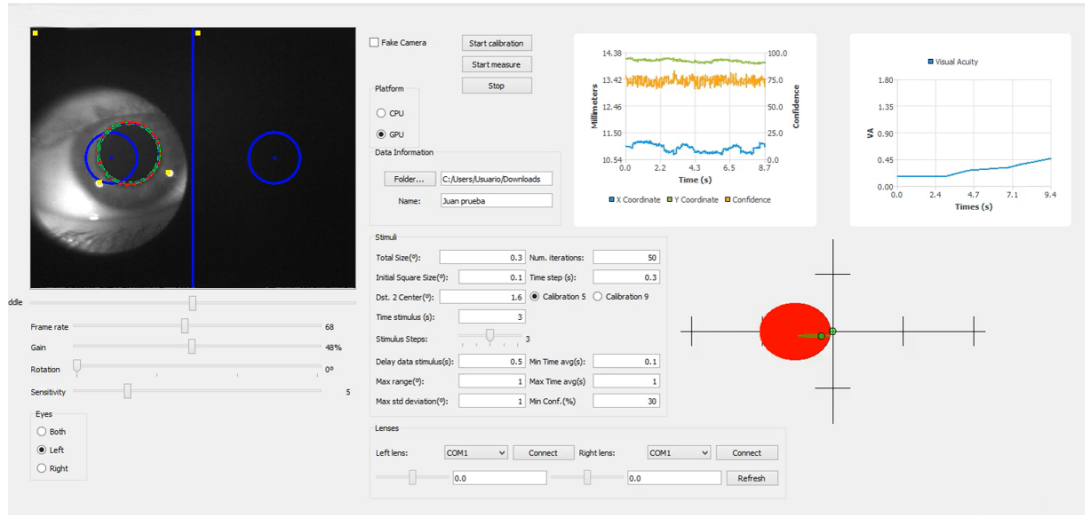
2.5 Sistemas ópticos reales dónde se está utilizando esta implementación paralela del seguimiento pupilar

Después de seleccionar la implementación GPU del algoritmo *Starburst* como la idónea para llevar a cabo la tarea de la búsqueda de la pupila dicha implementación se ha utilizado en varios experimentos. Se utilizó en un experimento [38] para realizar el seguimiento de una pupila mientras se medía el efecto *Stiles-Crawford* [39], que consiste en la diferencia de percepción en la intensidad de un estímulo según el lugar de la pupila en el que se presente el mismo.

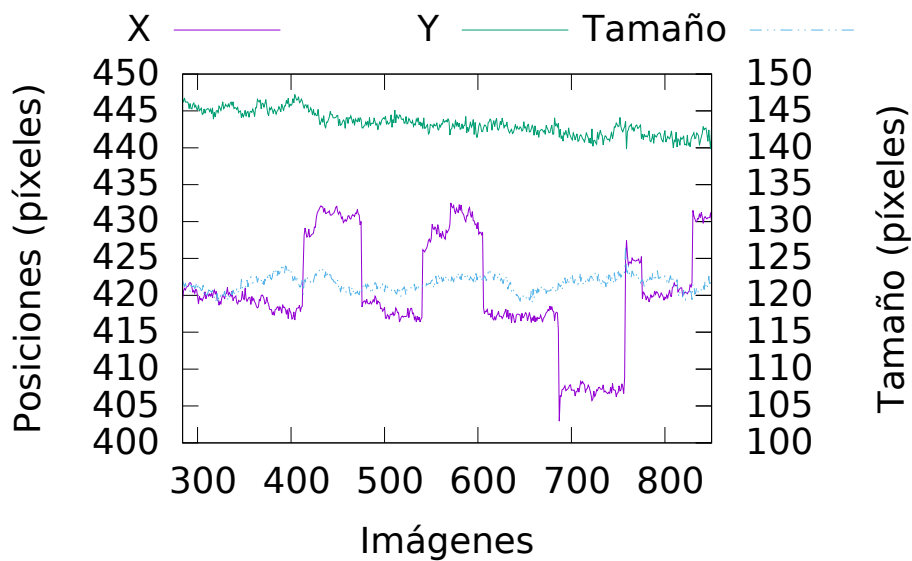
Otro experimento que actualmente utiliza esta implementación de alto rendimiento de seguimiento pupilar determina la agudeza visual del sujeto midiendo la respuesta de la pupila ante la presentación de varios estímulos [7]. En la figura 2.18b se muestra la posición y el tamaño de una pupila en una medida realizada durante dicho experimento.

Asimismo, en la figura 2.18a se muestra una captura de pantalla del programa utilizado para llevar a cabo dicho experimento.

Por otra parte, una implementación MATLAB desarrollada a partir de los resultados obtenidos con esta investigación se ha utilizado para seguir la posición de la pupila mientras se medía el desplazamiento del cristalino [40]. Asimismo, otra implementación MATLAB se ha utilizado para caracterizar el tamaño de la pupila bajo condiciones de visión normales [41].



(a)



(b)

Figura 2.18: (a) Captura de pantalla del programa utilizado para el experimento de agudeza visual automática. (b) Posición y tamaño de una pupila cuyo seguimiento se ha llevado a cabo en tiempo real con la implementación GPGPU desarrollada en esta Tesis.

3 Gafas de corrección automática y en tiempo real de la presbicia

3.1 Introducción

Los avances que se podrían llegar a alcanzar en el caso de que este dispositivo se desarrollase tendrían un gran impacto en la sociedad. La magnitud de lo que estas gafas conseguirían se puede comprender si se tiene en cuenta que toda persona mayor de 50 años se vería beneficiada por el uso de las mismas. La presbicia impone limitaciones y una degradación de la calidad de vida de los que la padecen que no hace sino verse incrementada en los últimos tiempos ante el aumento de dispositivos tales como los teléfonos móviles que requieren que los miremos a cortas distancias. Asimismo, el aumento de los trabajos que requieren el uso de ordenadores también contribuye a la necesidad de desarrollar una solución más completa que las que hay actualmente disponibles.

Por otra parte, la tendencia actual nos indica que la cantidad de tiempo que pasamos mirando pantallas cercanas no hace más que incrementarse. Aumentando la necesidad de soluciones más completas y satisfactorias a la presbicia. Si bien, la miopía también se está incrementando de forma paralela. En el año 2000 alrededor del 20% de la población mundial era miope y se estima que el 50% lo será en el 2050 [42]. La miopía en cierto modo suple las carencias de un présbita, esto es, una persona miope es capaz de enfocar objetos cercanos mientras que no puede hacerlo con los lejanos. Sin embargo, el problema a solucionar es el mismo que en el caso de los emétopes, es necesario suplir dinámicamente la falta de capacidad para enfocar correctamente objetos a una determinada distancia.

Unas Gafas para la Corrección Automática de la Presbicia supondrían una revolución para muchas personas. Sin embargo, desarrollar un sistema que cumpla las expectativas de los présbitas es una difícil tarea debido a los diferentes desafíos que se presentan en varios de los elementos fundamentales de dichas gafas. Este capítulo se centra en los

desafíos de rendimiento y eficiencia del procesamiento de las imágenes. Dicho procesamiento es una parte crucial para el control de las gafas. Pero otros elementos mecánicos como las lentes optoelectrónicas aún necesitan evolucionar para ofrecer una mejor experiencia.

3.2 Antecedentes y estado del arte

La presbicia es una degradación de las propiedades ópticas del ojo humano que se desarrolla a lo largo de muchos años y presenta bastante variabilidad entre diferentes personas. Los niños, que aún presentan una capacidad de acomodación intacta y en su pico de rendimiento son capaces de desarrollar una potencia de más de 10 dioptrías con su cristalino con facilidad, pueden enfocar objetos cercanos que se encuentren a 10cm [43]. Sin embargo, dicha capacidad de acomodación se reduce con la edad hasta alcanzar los cincuenta años [44]. Aunque otros estudios más recientes han medido cambios significativos en la dureza del cristalino hasta la edad de 60 años, por lo que la presbicia continuaría su desarrollo hasta esa edad [45].

A pesar de que la presbicia se empieza a desarrollar a edades tempranas y se va incrementando lentamente a lo largo de la vida de la persona, sus efectos no suelen ser apreciables hasta que se alcanzan los cuarenta años. Momento en el cual se empiezan a experimentar dificultades para leer textos o para ver imágenes cercanas. Antes de esa edad, el rango acomodativo se va reduciendo significativamente con lo que se pierde la capacidad de enfocar correctamente objetos muy cercanos. Sin embargo, muy pocas personas necesitan mirar tan de cerca por lo que sus efectos pasan inadvertidos. Cuando se alcanzan los cincuenta años, la presbicia ya se encuentra en un estado avanzado y la mayoría de las personas son incapaces de leer textos de pequeño tamaño debido a que cuando está suficientemente cerca está desenfocado (debido a la presbicia) y cuando se aleja el texto es demasiado pequeño para ser legible. Existe una excepción, como se ha comentado anteriormente: los miopes siguen siendo capaces de enfocar objetos cercanos debido a que su ojo siempre enfoca objetos cercanos incluso cuando se encuentra relajado. Si bien, dependiendo de la miopía de cada persona variará la distancia a la que es capaz de enfocar.

Como ha quedado expuesto, la presbicia es una condición que afecta a todo el mundo a partir de cierta edad. Por lo tanto, hay una serie de dispositivos en el mercado que tratan de solventar la pérdida de capacidad acomodativa del ojo humano. Sin embargo,

las soluciones actualmente a la venta son parciales y estáticas, de forma que presentan algunas desventajas. La solución más básica (y muy ampliamente utilizada) son las denominadas gafas "de cerca", las cuáles enfocan objetos a una distancia fija que puede variar entre diferentes modelos. A su favor tienen que suelen ser bastante económicas, su funcionamiento es muy sencillo y la adaptación a su uso es inmediata. En contra que solo enfocan a una distancia, impidiendo la visión de lejos o a cualquier otra distancia. Una solución más avanzada son las gafas multifocales o progresivas, las cuáles son capaces de enfocar varias distancias en función del área de la lente que se utilice para mirar. La figura 3.1 muestra una comparación entre la visión generada por unas gafas bifocales con dos distancias optimizadas y la visión generada por nuestras Gafas de Corrección Automática de la Presbicia. Como se puede ver en la figura, las gafas bifocales solo presentan una imagen nítida a dos distancias concretas, que suelen ser el infinito y algún punto cercano para el cuál se han diseñado los cristales. Mientras que las Gafas de Corrección Automática de la Presbicia son capaces de enfocar correctamente todas las distancias dentro de su rango de funcionamiento. Por otra parte, las gafas progresivas presentan problemas de adaptación para algunas personas. Tanto es así que algunas personas nunca consiguen adaptarse a ellas debido a las deformaciones presentes en las imágenes que ven a través de las lentes. Otra opción son las lentes intraoculares multifocales, que a diferencia de las gafas utilizan el mismo área para enfocar objetos a distintas distancias. Sin embargo, eso tiene una importante desventaja, y es que la calidad de la imagen de cada una de las distancias enfocadas se ve reducida significativamente. Otras alternativas más exóticas han explorado el uso de una corrección diferente para cada ojo, de forma que un ojo enfoca constantemente a lejos y el otro enfoca a cerca [46]. Por supuesto, la monovisión tiene importantes desventajas, como la pérdida de la percepción de la profundidad. A pesar de ello, existen diversas técnicas para generar la monovisión: lentes intraoculares, lentes de contacto o cirugía refractiva. Otra opción aún más extravagante es el uso de un anillo para reducir el tamaño de la pupila y mejorar la profundidad de foco del ojo. Este procedimiento ha demostrado ser capaz de mejorar la agudeza visual de cerca y de media distancia [47].

En el dispositivo que presentamos, para corregir automáticamente la presbicia se han empleado un par de lentes optoelectrónicas que son capaces de cambiar dinámicamente su potencia o distancia focal. Son estas lentes las que permiten ajustar el enfoque a la distancia requerida por el sujeto en todo momento. Existen diferentes métodos para conseguir este funcionamiento. En este caso, hemos utilizado las lentes ajustables de Op-

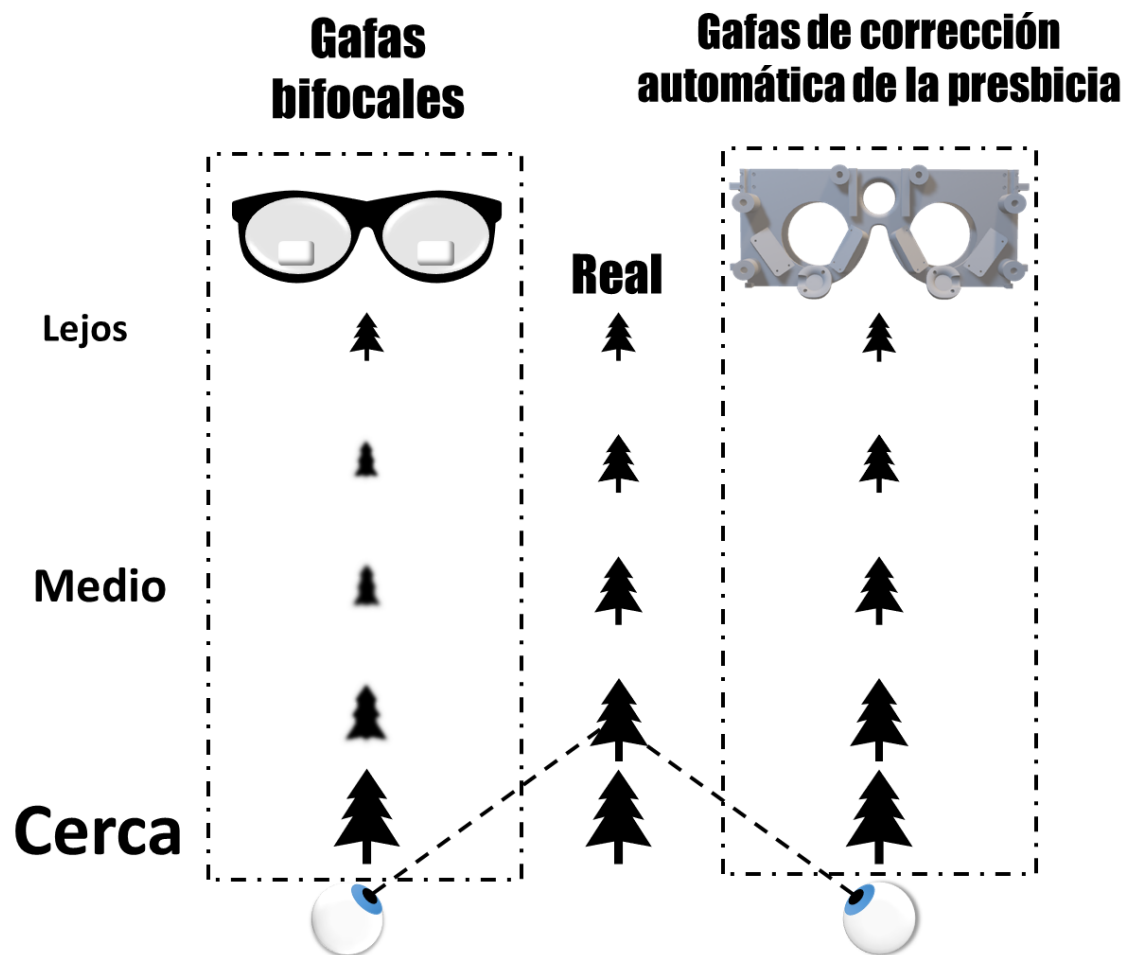


Figura 3.1: Comparación de la visión que generan unas gafas bifocales (a la izquierda) y la que generan las Gafas de Corrección Automática de la Presbicia (a la derecha). Mientras que las bifocales no enfocan correctamente todas las distancias las Gafas de Corrección Automática de la Presbicia sí lo hacen.

totune que disponen de un amplio rango de focalización, que va desde los -100mm hasta los +100mm, es decir, desde -10 a +10 dioptrías [48]. Estas lentes, por supuesto, tienen una gran utilidad en multitud de campos, como visión por computador, microscopía o como en nuestro caso en la corrección refractiva.

Las lentes optoelectrónicas ajustables presentan actualmente muchas aplicaciones potenciales, lo cuál está motivando un fuerte interés por parte de la comunidad científica que está presentando algunas de alternativas para el desarrollo de las mismas. Así *Lin y Chen* han presentado una lente libre de polarizadores que puede ajustar su foco y utiliza una tecnología de cristal líquido. Dispone, además, de una apertura de 6mm [49]. *Hasan et al.* han sido capaces de crear una lente ajustable con una gran apertura de 32mm, y sin sacrificar un rango de actuación razonable de 5.6 dioptrías [50]. Otro trabajo ha mostrado una lente con una apertura de 20mm y un rango de actuación de 3 dioptrías [51, 52]. Los autores de dicho trabajo han creado unas gafas en las que han montado sus lentes ajustables junto a un sensor de distancia que utilizan para controlar la potencia aplicada por las lentes. Asimismo, han propuesto una mejora al diseño de las mismas que implicaría el uso de una cámara y un sistema LiDAR (Light Detection And Ranging) [53] para tratar de paliar las limitaciones de su diseño determinando la distancia a la que está mirando el sujeto. El cuál solo funciona correctamente si el mismo está mirando de frente, mientras que si mira hacia los lados, aunque sea ligeramente, puede generar mediciones erróneas. Incluso hay lentes optoelectrónicas capaces de corregir aberraciones de más alto orden que el desenfoque [54]. Estas lentes cuentan con multitud de actuadores que usados conjuntamente son capaces de deformar la superficie de la lente generando aberraciones de hasta 4º orden. Si bien es cierto que el rango de actuación es limitado para dichas aberraciones.

Por otra parte, muy relacionados con nuestra propuesta encontramos a los *wearables*, que son dispositivos muy populares hoy en día, presentes en todo tipo de aplicaciones, siendo muy utilizados para monitorizar parámetros relativos a la salud o estado físico de las personas. Así, se utilizan para contar pasos, medir el pulso, controlar la diabetes, la calidad y cantidad del sueño, la presión sanguínea, etc. Los *wearables* están diseñados para ser pequeños dispositivos con un muy bajo consumo energético que les permita funcionar con una pequeña batería. Además, su pequeño tamaño permite que las personas los puedan llevar continuamente sin que suponga una molestia. Teniendo en cuenta la importancia actual de los *wearables* y su extendido uso no es de extrañar que se hayan desarrollado chips específicos para ellos. ARM cuenta con una gama de procesadores específicos para

wearables [55]. Otros componentes importantes para estos dispositivos como las pantallas también han visto proliferar una gama específica adaptada a las necesidades de tamaño y consumo de los mismos.

La propuesta de unas gafas dinámicas auto-acomodativas presentada aquí encaja perfectamente en la descripción de *wearables*. Sin embargo, las necesidades de cómputo de estas gafas son significativamente superiores a los de un *wearable* típico. A pesar de ello, sigue siendo necesario que el dispositivo tenga un bajo consumo para que pueda funcionar durante el suficiente tiempo y no se caliente en exceso. Por lo tanto, es necesario que el dispositivo ofrezca una buena eficiencia energética siendo capaz de cumplir los requisitos de cómputo junto a un bajo consumo. Para asegurarnos de que es así, en este capítulo se ha llevado a cabo una evaluación de varias plataformas con el objetivo de determinar cuál ofrece el mejor compromiso de eficiencia y rendimiento.

Una idea similar a la que proponemos ha sido recientemente publicada por *Padmanaban et al.* [56]. Su sistema, llamado *Autofocals*, utiliza componentes comerciales tanto en la parte de seguimiento de pupila como en la parte de las lentes optoelectrónicas. Además, han desarrollado igual que en nuestro caso un soporte personalizado para integrar los distintos componentes. En lo que respecta a las lentes optoelectrónicas han tenido acceso a un modelo que no está disponible públicamente de la misma marca que se ha utilizado en este capítulo, se trata de unas Optotune EL-30-45 que cuentan con una apertura bastante más amplia que modelos anteriores pero un rango de actuación mucho más limitado. Sin embargo, por lo que se puede deducir de su publicación su dispositivo no es portátil que es el mayor desafío de este tipo de dispositivos.

Otro sistema similar ha sido desarrollado por *Mastrangelo et al.* [57] utilizando un método novedoso para llevar a cabo el seguimiento de pupila utilizan un reducido número de LEDs y sensores de luz que permite reducir tanto el tamaño como la energía requerida por el dispositivo para funcionar.

3.3 Exploración de las posibilidades de distintas implementaciones

Después de las implementaciones y el análisis llevado a cabo en el capítulo 2 se determinó que el algoritmo Starburst era un buen candidato para realizar el procesamiento de las imágenes y obtener la posición de la pupila con una buena precisión. Una vez elegido

el algoritmo de seguimiento pupilar, en el presente capítulo se explora el rendimiento y consumo de tres plataformas alternativas para determinar cuál de ellas ofrecía una mejor relación entre esas dos características. Las tres plataformas hardware elegidas son: las GPUs de los smartphones, las GPUs de sobremesa y las FPGAs. Si bien, la implementación para una GPU de sobremesa se ha incluido para enriquecer la comparación y ofrecer un punto de referencia más que como una opción real para gestionar una versión portátil de las gafas de la presbicia.

Por otra parte, por cuestiones de compatibilidad con las diferentes plataformas hardware elegidas la implementación del algoritmo del seguimiento pupilar debía realizarse en OpenCL, ya que se hace uso de GPUs de teléfonos móviles que no soportan CUDA. En el caso de la programación de FPGAs existen multitud de alternativas, entre las cuáles se encuentra OpenCL, lo que lo convierte en el lenguaje idóneo para nuestros propósitos al poderse reutilizar la misma implementación algorítmica en las diferentes plataformas hardware. OpenCL es un lenguaje orientado a las aplicaciones de alto rendimiento, lo cual no encaja totalmente con dispositivos de bajo consumo. Pero también es un lenguaje extraordinariamente flexible y genérico que permite obtener el máximo rendimiento de cualquier dispositivo que lo soporte.

3.3.1 Implementación en un SoC móvil (smartphone)

Actualmente los *smartphones* cuentan con hardware muy avanzado que los convierte en dispositivos capaces de ejecutar cómputo heterogéneo. Todos los *smartphones* cuenta con al menos una CPU y una GPU, pero la mayoría de ellos cuentan además con hardware específico para realizar algunas tareas que hacen uso de cierta inteligencia artificial, codificación y decodificación de vídeo o procesamiento de imágenes. De forma que se ha aprovechado la potencia del hardware disponible para obtener el máximo rendimiento, en concreto haciendo uso de las GPUs de los móviles. Para ello como, se ha comentado con anterioridad, se ha hecho uso de una implementación en OpenCL.

Esta implementación en un smartphone ha sido además la utilizada para llevar a cabo los experimentos con sujetos, es por ello que la CPU del mismo también lleva a cabo tareas importantes, tales como: el control de las cámaras, de las lentes, y la gestión de la interfaz de usuario para permitir configurar y monitorizar el funcionamiento de las gafas. Sin embargo, para la obtención de los resultados de rendimiento que se presentan en la sección 3.5.2 solamente se tiene en cuenta el tiempo y energía empleados en la parte

encargada del procesamiento de las imágenes, obviando el resto de tareas.

Se ha comparado el rendimiento y consumo de dos teléfonos móviles, el Samsung Galaxy S7 que ejecuta Android 7.0 y cuenta con un procesador Samsung Exynos 8890; y el Xiaomi Redmi Note 3 Pro que ejecuta android 6.0 y cuenta con un procesador Qualcomm Snapdragon 650. Para ejecutar el código escrito en C++ y OpenCL en Android se ha utilizado el NDK (Native Development Kit 18b) proporcionado por Google. Si bien el soporte de OpenCL en Android es escaso y depende de cada fabricante, por lo que las herramientas disponibles para el desarrollo no son muchas y presentan incompatibilidades entre distintos fabricantes.

CUDA y OpenCL son dos lenguajes de programación bastante similares en su aproximación al paralelismo. Sin embargo, CUDA dispone de algunas instrucciones especializadas que no están disponibles en OpenCL. Algunas de las optimizaciones aplicadas a la implementación OpenCL en el smartphone y descritas en el siguiente apartado son: el uso de accesos a memoria vectorizados, memoria local, operaciones matemáticas nativas (seno, coseno, ...) y el uso de tipos de precisión reducida (**halfs**) cuando es posible.

3.3.1.1 Accesos a memoria vectorizados

Los accesos a memoria vectorizados hacen uso de tipos de datos empaquetados, tales como: `float2`, `int4`, o `int8`. Estos tipos de datos incluyen tantos números como indica la cifra al final de sus nombres dentro de ellos, que pueden ser accedidos independientemente, con la ventaja de que al acceder a la memoria del dispositivo se generan operaciones de carga de datos de más bytes que permiten maximizar el uso del ancho de banda disponible y, en última instancia, mejorar el rendimiento. Sin embargo, el uso de este tipo de datos puede aumentar la complejidad del procesamiento al hacer necesario evitar accesos ilegales o cargar datos que luego no son procesados. Esta optimización se ha utilizado para optimizar las operaciones de procesamiento de imágenes y para las operaciones de búsqueda de puntos en el borde de la pupila que almacenan los puntos encontrados utilizando el tipo de datos `float2`.

3.3.1.2 Memoria local

En OpenCL la memoria local hace referencia a lo que se conoce como memoria compartida en CUDA. Esta memoria se encuentra muy próxima a las unidades de procesamiento de la GPU y permite obtener un ancho de banda muy superior al de la memoria del dis-

positivo, así como unas latencias inferiores. Sin embargo, la memoria local es un recurso escaso que se puede agotar con facilidad. Esta optimización se ha utilizado para mejorar el rendimiento de las funciones de procesamiento de imágenes, para el algoritmo RANSAC (Random Sample Consensus), y para la comunicación en la etapa de búsqueda de puntos. La memoria local se puede reservar tanto dinámicamente como estáticamente. En esta ocasión se ha reservado dinámicamente para evitar reducir la flexibilidad de las funciones.

3.3.1.3 Operaciones matemáticas nativas

Las operaciones matemáticas nativas son implementaciones optimizadas de las mismas que generalmente sacrifican algo de precisión para obtener grandes beneficios en rendimiento. En concreto, se han utilizado las funciones `native_sin`, `native_cos`, y `native_sqrt` en la etapa de búsqueda de puntos.

3.3.1.4 Tipos de datos especializados

Los tipos de datos especializados se ajustan por regla general mejor a la tarea a realizar que los tipos de datos más genéricos. Así, el tipo de datos `half` se ha utilizado para llevar a cabo la búsqueda de los puntos y en el algoritmo RANSAC. Su uso permite tanto reducir la cantidad de datos transmitida a/desde la memoria del dispositivo como aumentar la cantidad de operaciones que se pueden realizar. Esto es posible debido a que la precisión que ofrece el tipo de datos `half` es suficiente para el problema que aquí se resuelve.

3.3.1.5 Fusión de kernels

La fusión de kernels es una optimización muy interesante que requiere separar la lógica de la implementación. A veces, la lógica del algoritmo lleva a cabo operaciones consecutivas sobre los mismos datos de forma independiente. Sin embargo, si dichas operaciones se aplican a cada píxel de forma independiente, sin interferir en los resultados de los demás píxeles, puede ser interesante realizar las dos operaciones en el mismo kernel. Esto sucede en la última parte de la transformada Top-hat y la umbralización. Primero se lleva a cabo una substracción píxel a píxel y luego una umbralización píxel a píxel. Para ello, después de la substracción, todos los valores se almacenan en memoria de dispositivo y luego se vuelven a cargar para llevar a cabo la umbralización. Al fusionar dichos kernels se evita ese almacenamiento y carga de datos y se elimina, además, la sobrecarga de las llamadas a OpenCL para lanzar otro kernel. Aunque es cierto que estos kernels, por regla general,

son demasiados específicos como para que puedan ser reutilizados en otras ocasiones. Asimismo, también aumenta la complejidad del código y dificulta su mantenimiento.

Por otra parte, se han aplicado algunas otras optimizaciones descritas en el capítulo 2 en la implementación CUDA como: los filtros separables, la reutilización de los números aleatorios, utilizar `uint8_t` para almacenar las imágenes o prescindir de la interpolación de los reflejos corneales.

Para medir el consumo de energía de esta implementación se ha utilizado la app *Trepn Power Profiler* [58] desarrollada por Qualcomm. Esta aplicación permite medir varios parámetros del smartphone, entre los cuáles se encuentra el consumo energético. Además, *Trepn* acepta notificaciones con el estado de nuestra aplicación, lo cual permite enviar una notificación cuando empiece la parte del procesamiento y otra cuando termina. De esa forma se puede determinar con precisión el consumo energético del procesamiento, evitando que la medida sea contaminada con otros datos no deseados. Asimismo, un móvil cuenta con una pantalla que también consume energía y otros procesos que se ejecutan constantemente aumentando el consumo. Para eliminar dichos consumos de la medida *Trepn* permite tomar durante 30 segundos previos a la ejecución del procesamiento una media del consumo energético que se toma como consumo base y que se resta del consumo total obtenido durante el procesamiento.

3.3.2 Implementación en un ordenador de sobremesa

La implementación utilizada en el ordenador de sobremesa, ha sido la misma que la que se ha utilizado en el móvil con muy pequeñas variaciones. El procesador gráfico NVIDIA GTX 980 utilizado no incluye soporte para operaciones con tipos de datos `half` y por lo tanto no se han podido utilizar. También se ha modificado el tamaño de los grupos de hilos de OpenCL (bloques en CUDA). Las GPUs móviles presentan bastante limitaciones en el tamaño del grupo de hilos debido a los limitados recursos de los que disponen, mientras que todos los procesadores gráficos de NVIDIA modernos aceptan bloques de hasta 1024 hilos, y de hecho es necesario utilizar la mayor cantidad posible para conseguir utilizar todos los recursos hardware disponibles en el procesador.

3.3.3 Implementación en una FPGA

La implementación FPGA ha sido probada utilizando una FPGA Xilinx Alveo u250 (en la nube de NIMBIX) y una placa Zedboard, ambas de la marca Xilinx. La cuál

ofrece una amplia variedad de lenguajes de programación que es posible utilizar en sus dispositivos, incluyendo: C/C++, OpenCL y HLS. En este caso, evidentemente, se ha utilizado OpenCL por homogeneidad con las otras implementaciones y por su facilidad de uso. Además, se ha compilado utilizando el entorno de desarrollo Xilinx SDX 2018.3. Aunque se ha utilizado el mismo código OpenCL que en las otras implementaciones se han aplicado optimizaciones específicas o se han modificado algunas optimizaciones para adaptarlas a la FPGA. En concreto, se han añadido **pragmas** específicos de Xilinx a las optimizaciones de memoria local, desenrollado de bucles, pipelining y particionado de arrays. Otras optimizaciones se han llevado a cabo de la misma manera que en el resto de implementaciones. A saber, fusión de los kernels para reducir los accesos a memoria y la sobrecarga de OpenCL; uso de tipos de datos vectorizados; operaciones matemáticas nativas.

3.3.3.1 Memoria local

De igual manera que en las GPUs la memoria local ofrece una baja latencia y alto ancho de banda en las FPGAs. Asimismo, también es un recurso muy escaso. Pero a diferencia de las GPUs, dónde es posible reservar memoria local dinámicamente, en este caso la reserva tiene que ser estática. Ello se debe a que la memoria local se corresponde a bloques físicos en la FPGA que al compilar se asignan a un kernel. Por lo tanto, el tamaño de la memoria local reservada se debe conocer en tiempo de compilación. Dado que se ha utilizado memoria local en las operaciones de procesamiento de imágenes y en el algoritmo RANSAC algunos de sus parámetros deben ser conocidos en tiempo de compilación. En concreto, para las funciones de procesamiento de imágenes es necesario conocer el radio que se va a utilizar. Por lo tanto, se pierde flexibilidad con estas funciones que sólo se podrán ejecutar con la configuración con la que sean compiladas.

3.3.3.2 Desenrollado de bucles

Esta optimización también está presente en CUDA, pero en una FPGA esta optimización implica un comportamiento muy diferente. Si bien en CUDA es el código el que se desenrolla, en una FPGA se genera tanto hardware adicional como sea necesario para ejecutar en paralelo las iteraciones del bucle cuando éste es paralelo. Esta optimización se ha utilizado en la funciones de procesamiento de imágenes que ya contaban con un tamaño estático, el cuál es necesario para llevar a cabo el desenrollado. En concreto, se

aplica al bucle que carga los datos a memoria local y al bucle que procesa los datos. Para desenrollar el bucle se emplea el `pragma __attribute__((opencl_unroll_hint))`. Sin embargo, el desenrollado de bucles es muy exigente en lo que a recursos se refiere, ya que según la cantidad de desenrollado que se use puede incrementar varias veces la cantidad de recursos consumidos por un kernel.

3.3.3.3 Segmentación

La segmentación o *pipelining* permite aumentar la concurrencia en los recursos hardware de la FPGA de la misma forma que lo hace en las CPUs. La segmentación consiste en la división del procesamiento en etapas, tales como: carga de datos, procesamiento, almacenamiento. Si un bucle realiza esas tres etapas puede ejecutar de forma solapada en el tiempo cada una de ellas para tres iteraciones diferentes, de forma que se reduce de forma significativa el tiempo de ejecución sin necesidad de aumentar en exceso los recursos hardware utilizados. Esta optimización se aplica de una forma similar a la de desenrollado de bucles, basta con añadir un `pragma` antes del bucle que se quiere segmentar: `__attribute__((xcl_pipeline_loop))`.

3.3.3.4 Particionado de arrays

La división de los arrays de memoria local permite aumentar el ancho de banda al acceder a sus datos. Consiste en la separación en partes de los arrays de memoria local, de forma que se pueda acceder en paralelo a cada una de las partes a través de diferentes puertos de lectura y escritura. Ésto resulta en un aumento del ancho de banda disponible. Esta optimización es muy útil para funciones con una fuerte presión sobre la memoria como las de procesamiento de imágenes. Se muestra un ejemplo de esta optimización en el código 3.1.

Código 3.1: Ejemplo de particionado de arrays

```
1    local uchar lmem[WIDTH + 2 * EROSION_RADIUS + 1] __attribute__((↵
    ↵ xcl_array_partition(cyclic, 8, 1)));
2    local uchar out_lmem[WIDTH] __attribute__((xcl_array_partition(cyclic↵
    ↵ , 8, 1)));
```

Plataforma	Tiempo sin opt.(ms)	Tiempo con opt.(ms)	Aceleración
Zedboard Proc. Imág.	748	44	17.0x
Zedboard Búsq. Puntos	237	22.1	10.7x
Alveo Proc. Imág.	246.7	14.35	17.2x
Alveo Búsq. Puntos	49.5	5.4	9.2x

Tabla 3.1: Tiempos de ejecución para las etapas de preprocesamiento y de búsqueda de puntos del algoritmo de seguimiento pupilar. Dichos tiempos de ejecución y la aceleración obtenida al implementar varias optimizaciones se muestran tanto para la placa Zedboard como para la más potente Alveo u250.

3.3.3.5 Max Memory Ports

Esta opción, que se puede activar durante la compilación, permite a los kernels de OpenCL utilizar varios puertos para acceder en paralelo a buffers diferentes alojados en la memoria del dispositivo. Sin ella, sólo hay disponible un puerto para acceder a memoria incluso cuando esos accesos están en buffers separados. Evidentemente, esto permite mejorar el ancho de banda de operaciones que necesitan acceder a varios buffers simultáneamente.

La tabla 3.1 muestra los tiempos de ejecución y la aceleración obtenida para las etapas del procesamiento que han sido optimizadas. Dichos tiempos y aceleraciones se desglosan por etapas y para cada una de las dos placas utilizadas. En concreto, la etapa de procesamiento de imágenes inicial obtiene una mejora significativa en su rendimiento gracias a las optimizaciones de *memoria local* + *desenrollado de bucles* + *particionado de arrays* + *Max Memory Ports*. La aceleración conseguida con respecto a la versión sin optimizar es de 17.0x para la placa Zedboard y de 17.2x para la Alveo u250.

Por otra parte, la etapa de búsqueda de puntos ha mejorado su rendimiento gracias a las optimizaciones de *segmentación* + *Max Memory Ports*. Obteniendo en este caso una aceleración de 10.7x y 9.2x para la Zedboard y la Alveo u250 respectivamente.

Para medir el consumo de las implementaciones en las dos FPGAs se ha utilizado el programa Xilinx Vivado que ofrece una estimación del consumo de las mismas.

3.4 Diseño y creación de un prototipo

Si bien la parte más importante para el correcto funcionamiento de las gafas son el seguimiento pupilar y las lentes optoelectrónicas, no se puede obviar que sin una correcta

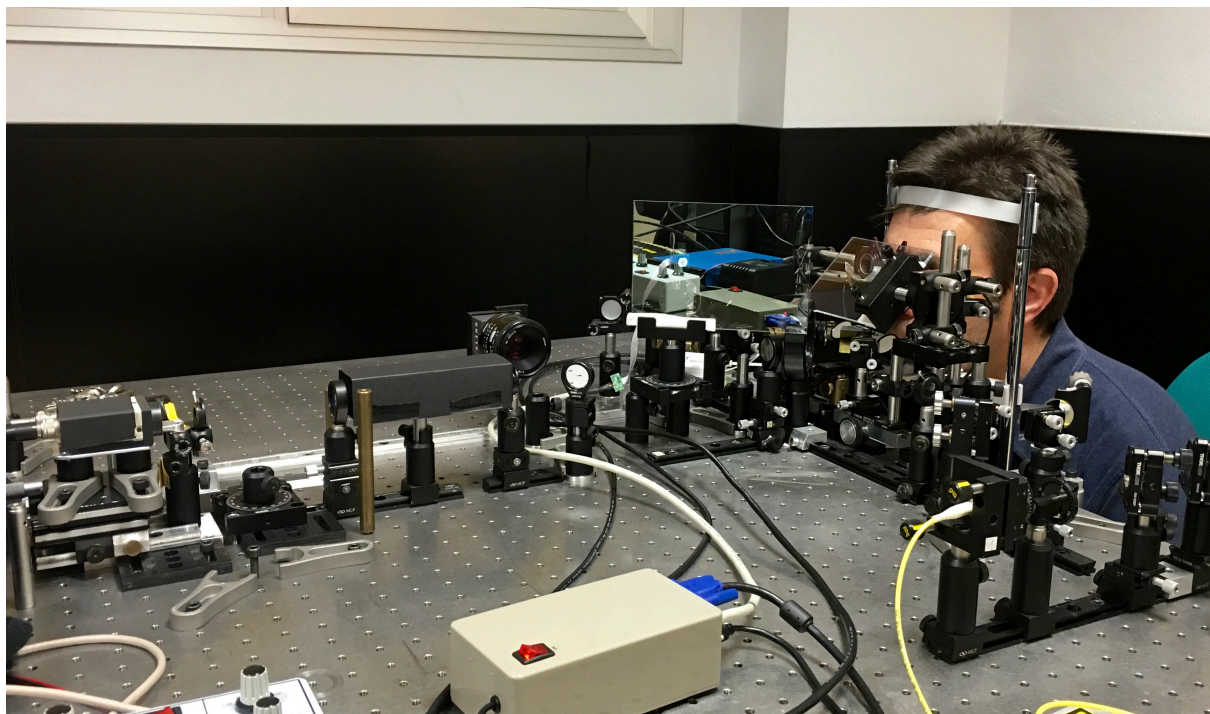


Figura 3.2: Sistema inicial con el prototipo sobre una mesa óptica.

integración nada sería posible. Así, es importante tener en cuenta la comodidad del sujeto, ya que unas gafas incómodas reducirán el tiempo durante el cuál el sujeto esté dispuesto a llevarlas y aumentarán sus movimientos, lo que repercutirá en la calidad de los resultados. Para conseguir que sean cómodas hay dos factores importantes: el peso de las gafas, que viene en su mayor parte determinado por las lentes optoelectrónicas y los cables de las mismas y de las cámaras; y, por supuesto, la forma de las partes que están en contacto con la cara del sujeto.

Además, es muy relevante la posición en la que se colocan las cámaras y los LEDs infrarrojos, el ángulo de iluminación y el ángulo de las cámaras pueden facilitar o dificultar la tarea de los algoritmos de seguimiento pupilar.

3.4.1 Prueba de concepto en una mesa óptica

La primera prueba de concepto realizada sobre la idea de desarrollar unas Gafas de Corrección Automática de la Presbicia y en tiempo real de la presbicia se llevó a cabo en una mesa óptica de grandes dimensiones tal y como se muestra en la figura 3.2.

Para llevar a cabo dichos experimentos, se utilizaron dispositivos que más allá de bus-

car la viabilidad del prototipo como algo portátil y que se pudiera llevar en la cabeza cómodamente buscaban simplemente demostrar que la idea era posible en la práctica, al menos si se obviaban las limitaciones de portabilidad y tamaño/comodidad. Así, las herramientas más importantes de las que se ha hecho uso han sido:

- Una cámara Pixelink D721, la cuál es capaz de capturar hasta 150 imágenes/s con una resolución de 1280x1024 píxeles (o más rápido si se reduce la resolución).
- Un ordenador de sobremesa equipado con un procesador Intel i5-4690 y un procesador gráfico NVIDIA 980 GTX.
- Dos lentes Optotune EL-10-30-TC con una apertura de 10mm y posteriormente dos lentes Optotune EL-16-40-TC con una apertura de 16mm.
- Diversas lentes y espejos necesarias para capturar las imágenes.
- Dos pantallas para mostrar estímulos de cerca y lejos.

Si bien el sistema que se muestra en la figura 3.2 puede parecer extremadamente complejo, lo cierto es que muchos de los componentes dispuestos sobre la mesa óptica pertenecen a otros experimentos que se estaban realizando simultáneamente en la misma [14]. Aprovechando que uno de esos experimentos capturaba las imágenes de los dos ojos y mediante un prisma las aproximaba para poder tomar una imagen en alta resolución de los ojos se reutilizó la óptica del sistema para nuestro nuevo propósito. La figura 3.3 muestra un esquema del sistema, en el se muestran dos caminos ópticos que comparten parte sus elementos. Una parte del sistema toma imágenes de los ojos utilizando luz infrarroja de 800nm para llevar a cabo el seguimiento pupilar, utilizando como iluminación dos LEDs situados junto a los ojos. Mientras que la otra parte del sistema utiliza luz, también infrarroja pero en este caso de 1050nm para medir las aberraciones del ojo utilizando para ello un sensor Hartmann-Shack, del cuál se hablará extensamente en el capítulo 4. Al disponer del sensor Hartmann-Shack se pudo medir el desenfoque del sistema al mismo tiempo que funcionaba el sistema de control de las lentes, aunque el funcionamiento conjunto es extremadamente complejo por las limitaciones de espacio que introduce la pequeña apertura de las lentes y los reflejos que añaden.

Como se ha comentado el sistema de la mesa óptica también contaba con dos pantallas que se utilizan para mostrar estímulos cerca y lejos, están situadas a 0.4m y 2.7m del sujeto respectivamente. Dichas pantallas se sitúan visualmente alineadas delante del

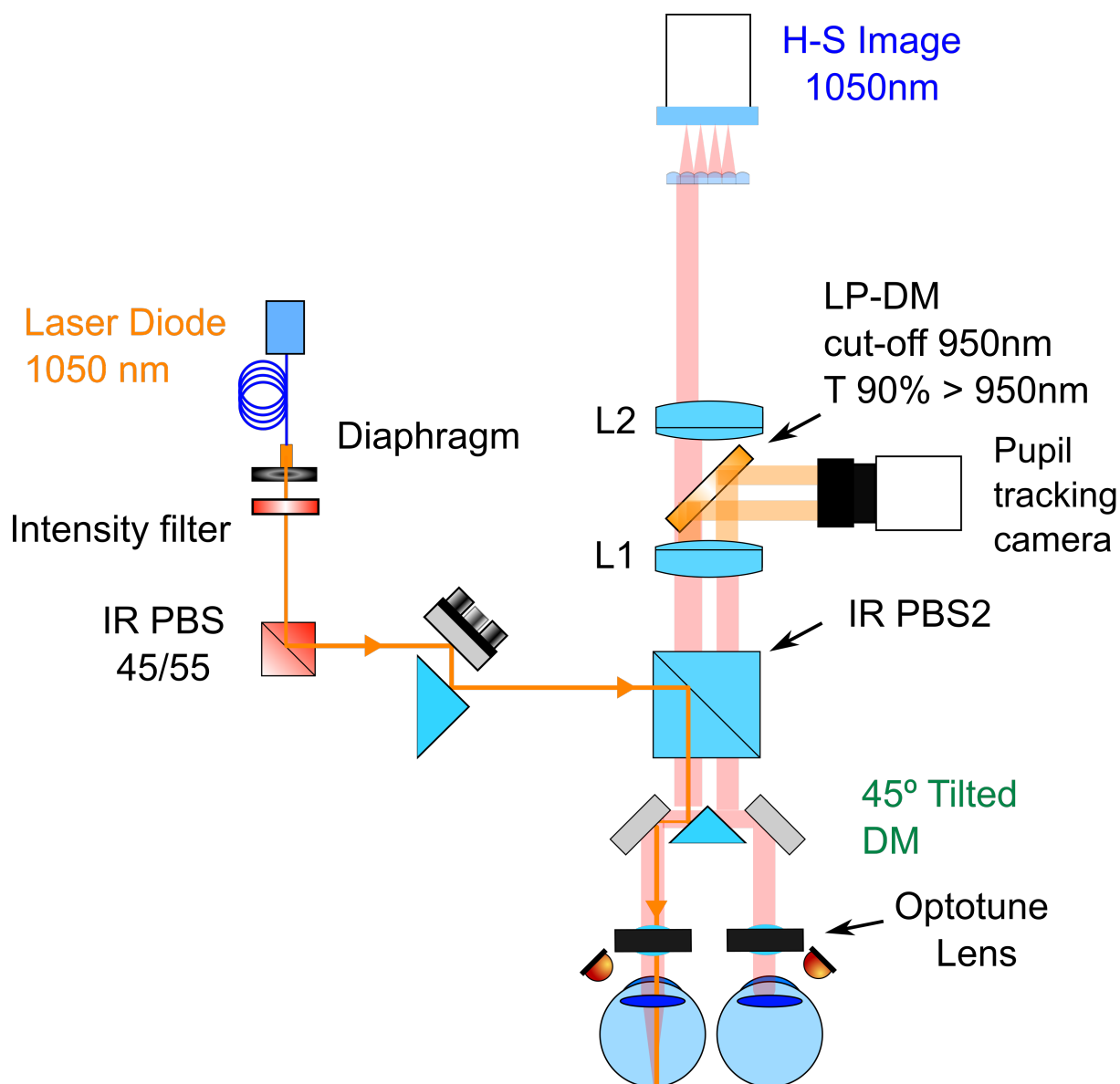


Figura 3.3: Esquema del sistema utilizado para corregir la presbicia incluyendo un camino para medir las aberraciones utilizando un sensor Hartmann-Shack.¹

sujeto, aunque la que se encuentra a 0.4m está en realidad a la derecha y lo que el sujeto ve es su reflejo en un divisor de haz. Las pantallas además se encienden alternativamente, de forma que nunca están encendidas simultáneamente pues eso generaría confusión al sujeto.

¹Imagen originalmente creada por Emmanuel Chirre [14] y modificada para añadir las lentes optoelectrónicas al sistema. Se ha obtenido permiso expreso del autor para su reproducción.

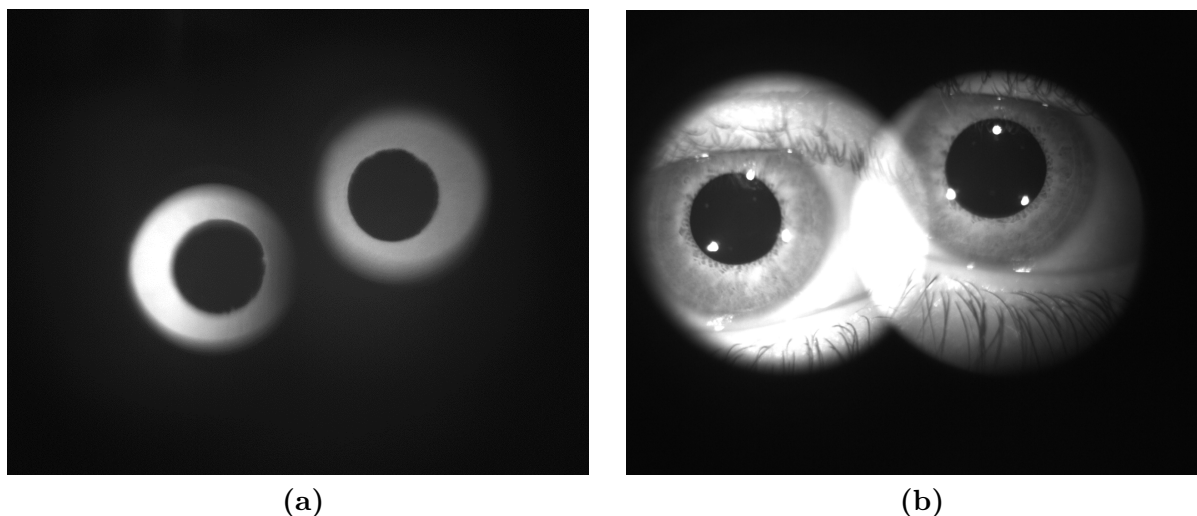


Figura 3.4: (a) Imagen de dos ojos artificiales tomada con la cámara encargada del seguimiento pupilar a través de las primeras lentes utilizadas (Optotune EL-10-30-TC). (b) Imagen de dos ojos reales tomada con la cámara encargada del seguimiento pupilar a través de las segundas lentes utilizadas (Optotune EL-16-40-TC).

El sujeto se sienta delante de la mesa óptica y su cabeza descansa sobre una mentonera, lo cuál otorga suficiente estabilidad para llevar a cabo el experimento. Además, dado que cada persona tiene una interdistancia distinta entre sus pupilas es necesario ajustar el sistema para adaptarlo a la distancia correcta. Así, es necesario mover las lentes para alinear sus centros con los centros de los ojos del sujeto y es necesario ajustar el sistema de espejos más próximo a los ojos para optimizar la posición de los ojos en la cámara. La figura 3.4a ilustra cómo de difícil era la alineación de los sujetos con las primeras lentes optoelectrónicas utilizadas. Mientras que la figura 3.4b muestra que las segundas lentes utilizadas facilitaron en gran medida la tarea. Sirvan además dichas figuras para ilustrar la imagen que se obtiene gracias al sistema de espejos, prisma y lentes.

Al reemplazar las lentes EL-10-30-TC con las EL-16-40-TC también se mejoró la pieza que se utilizaba para sujetar las lentes incluyendo tres LEDs de iluminación infrarroja cuyos reflejos corneales se pueden apreciar en la figura 3.4b. El nuevo diseño permitió mejorar significativamente la uniformidad y cantidad de la iluminación con respecto a la situación anterior dónde un sólo LED situado más lateralmente tenía que iluminar cada ojo.

Fruto de los resultados obtenidos mediante este primer prototipo, que si bien tenía importantes limitaciones físicas permitió demostrar la utilidad de la idea, se realizaron

varias comunicaciones [59, 60, 61].

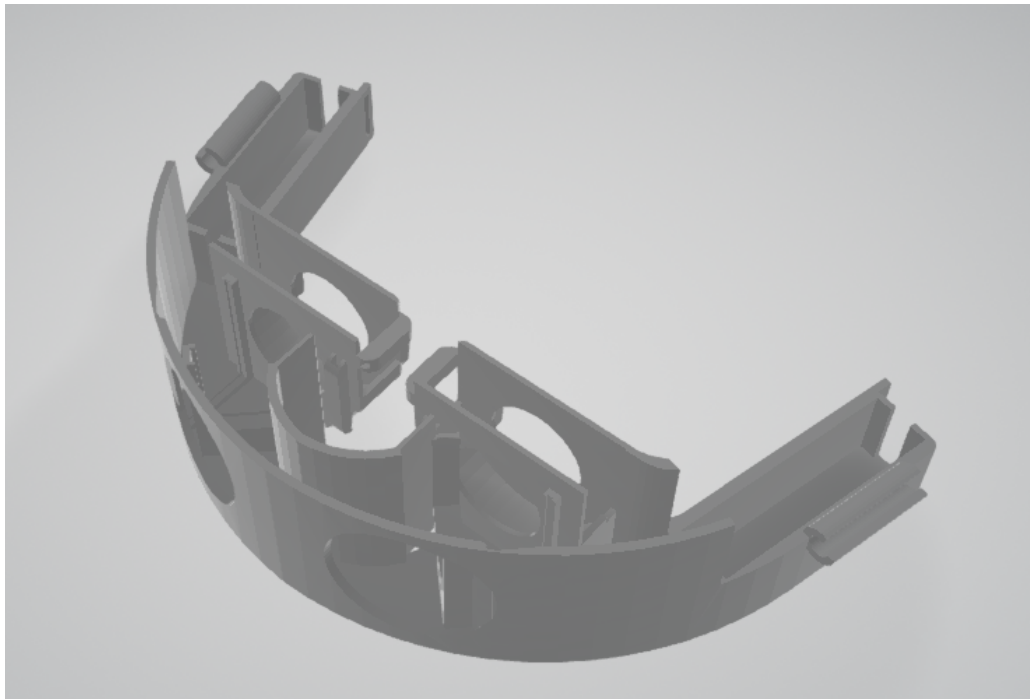
3.4.2 Prueba de concepto portátil

Si bien el desarrollo del sistema sobre una mesa óptica planteaba muchos retos y posibles soluciones, la integración de todos los componentes en unas gafas portátiles conlleva una dificultad mucho mayor. La necesidad de reducir el tamaño de casi todos los componentes y reubicarlos para adaptarse al nuevo soporte ha requerido de constantes iteraciones en el diseño del soporte y en el reemplazo de las cámaras.

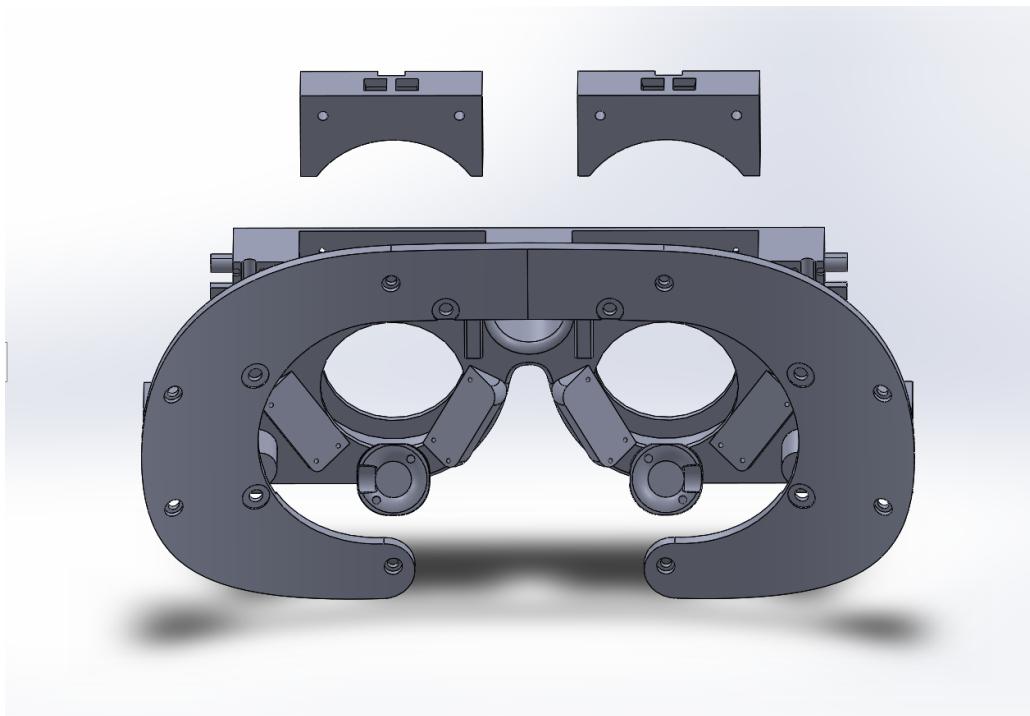
Una primera versión de las gafas portátiles impresa con una BQ Witbox 2 incluía dos webcam Microsoft Lifecam HD 3000, a las cuáles se les retiró los plásticos exteriores dejando sólo la placa dónde va el sensor y el cable. Estas cámaras se situaban a los lados de las lentes, tomando imagen gracias a un espejo caliente situado delante de las lentes optoelectrónicas. La figura 3.5a muestra el soporte 3D diseñado para el prototipo inicial, que contaba con un voluminoso espacio para alojar los controladores de las lentes situados a los dos lados de la cabeza que actuaban como las patillas de unas gafas. La iluminación infrarroja estaba compuesta por cuatro LEDs que se situaban delante de cada ojo en la parte interior del soporte. Este primer diseño no era muy cómodo debido a los bordes del mismo y su gran tamaño.

Tras muchas iteraciones del diseño y el uso de la impresora Form 2 de Formlabs que utiliza resina y tecnología láser para realizar una impresión de muy alta calidad se ha desarrollado el prototipo actual que se muestra en la figura 3.5b. Además, esta nueva versión cuenta con unas nuevas cámaras USB renovadas que son más pequeñas y se pueden colocar mirando directamente al ojo. Lo cuál permite prescindir del espejo caliente. Dichas cámaras son fabricadas por MISUMI² que está especializado en el diseño y fabricación de cámaras en miniatura para diferentes propósitos. Asimismo, los leds infrarrojos que eran muy difíciles de sostener anteriormente se han incorporado a una pequeña placa que cuenta con perforaciones para facilitar su sujeción. Por otra parte, los controladores USB de las lentes que eran muy voluminosos se han extraído de su carcasa de plástico reduciendo drásticamente su tamaño y permitiendo una mayor integración en el soporte. Ahora se sitúan a los lados de las gafas quedando al descubierto. Además, como se aprecia en la figura 3.6 se ha añadido una espuma a la parte frontal que está en contacto con la cara del sujeto para mejorar la comodidad de las gafas, que se sujetan con una cinta a la

²<http://www.misumi.com.tw/>



(a)



(b)

Figura 3.5: (a) Imagen del primer prototipo de las gafas para la corrección automática de la presbicia portátil. (b) Imagen del prototipo actual de las Gafas de Corrección Automática de la Presbicia portátil.



Figura 3.6: Imagen del prototipo de las gafas con todos los componentes montados.

cabeza.

La integración de los componentes conseguida ha mejorado ampliamente respecto al primer diseño en tamaño y comodidad, pero siguen habiendo 4 cables USB que salen de las gafas hacia el teléfono móvil que las controla. Estos cables son algo incómodos, además, inicialmente se utilizó un hub USB que a pesar de tener un tamaño comedido no dejaba de ser demasiado grande para lograr que las gafas fueran cómodas de llevar. Por ello, se ha utilizado finalmente un micro hub USB con 4 puertos para reemplazar al hub original. El hub utilizado tiene unas dimensiones extremadamente reducidas de 16x13mm que han dificultado las soldaduras. De hecho, el correcto funcionamiento del mismo sólo se ha logrado de forma intermitente debido a la complejidad de conseguir soldaduras y cables de la calidad necesaria para una conexión USB. Con el micro hub USB se han eliminado todos los conectores USB y se ha reducido en gran medida la longitud de los cables USB. De esa forma, el hub USB consigue reemplazar los 4 cables USB que salían de las gafas por uno solo, haciendo mucho más cómodo el uso de las mismas y reduciendo su peso.

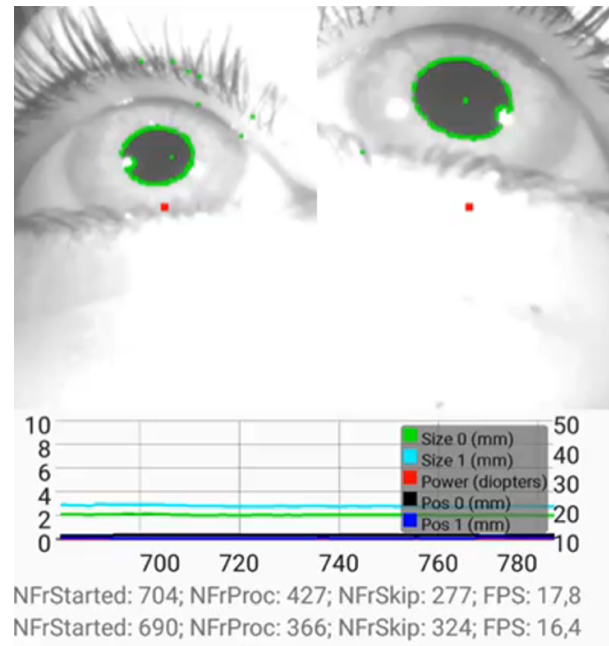
Para proveer de energía a los componentes de las Gafas de Corrección Automática de la Presbicia se ha utilizado la energía que provee el móvil encargado del procesamiento de

las imágenes y control de las gafas. El móvil provee la energía utilizada por las cámaras y las lentes optoelectrónicas. Mientras que una pequeña batería situada tras la espuma que apoya sobre la cara del sujeto aporta la energía consumida por los LEDs infrarrojos. El móvil utilizado para controlar las gafas y en los experimentos realizados con sujetos ha sido un Samsung Galaxy S7.

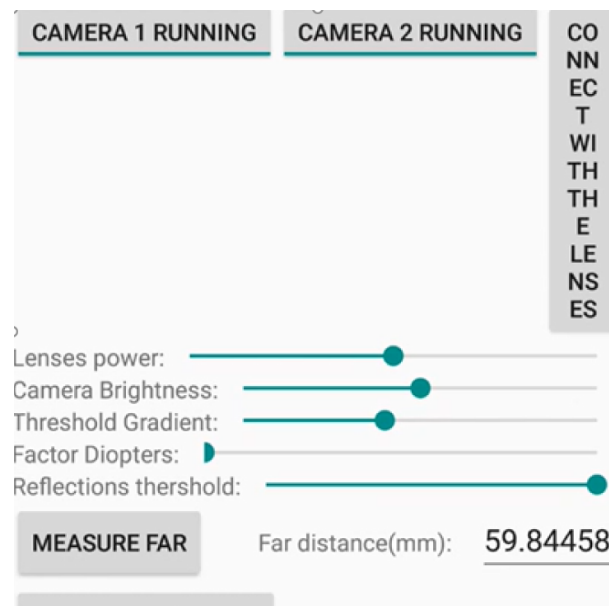
3.4.3 Control de las gafas

Una característica del sistema inicial de la mesa óptica era el uso del tamaño de la pupila para determinar dónde está mirando el sujeto. El tamaño de la pupila varía en función de la distancia a la que esté mirando el sujeto. Así, si mira un objeto lejano, el tamaño de su pupila será mayor que si mira un objeto cercano. Aunque la variación del tamaño de la pupila es muy diferente de un sujeto a otro y además se ve influenciado por otros factores: cantidad de luz ambiental, atención que se presta a algo, drogas, etc. Dicha variabilidad hace necesario llevar a cabo una calibración antes de empezar el experimento. Durante la calibración se pide al sujeto que mire cerca y lejos alternativamente mientras se monitorizan sus pupilas. A partir de la calibración es posible establecer el tamaño de la pupila para cada uno de los estímulos mostrados e incluso mediante una interpolación calcular valores intermedios. Otro problema del uso del tamaño de la pupila es la velocidad a la que se producen los cambios en el mismo, el cuál suele cambiar relativamente despacio si se compara con la velocidad de convergencia. Por lo tanto, si se utiliza como principal factor para determinar la distancia a la que está mirando el sujeto la respuesta del sistema se vuelve algo lenta y la satisfacción del sujeto con el sistema se degrada. A pesar de ello, en condiciones de luz controladas funciona muy bien y sirvió para controlar la versión inicial del sistema con mucho éxito. Para superar las limitaciones del control por medio del tamaño de las pupilas, el mismo se ha sustituido por el control mediante la posición de las mismas. Cuando se mira un objeto cercano los ojos convergen (se acercan), mientras que cuando se mira un objeto lejano estos divergen (se alejan). Por lo tanto, el uso de la vergencia es una herramienta perfecta para determinar la distancia a la que el sujeto está mirando. Además, la vergencia es muy rápida y es independiente de cualquier otro factor externo. Sin embargo, es cierto que algunas personas sufren insuficiencia de convergencia u otras afecciones como el estrabismo que impedirían funcionar correctamente a nuestro sistema. A pesar de ello, esta solución es la mejor de las estudiadas y la que se ha utilizado.

Como ya se ha mencionado, todos los componentes de los gafas son controlados y



(a)



(b)

Figura 3.7: (a) Captura de pantalla de la app de control con la corrección activada. (b) Captura de pantalla de las opciones de configuración de la app de control.

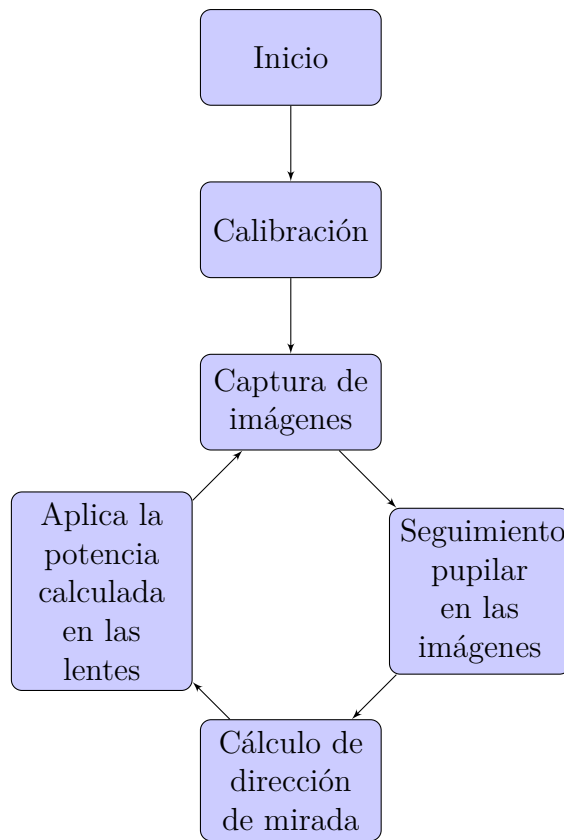


Figura 3.8: Diagrama de control de las Gafas de Corrección Automática de la Presbicia.

alimentados por un Samsung Galaxy S7, a excepción de los LEDs infrarrojos, que usan una batería. De tal forma que las imágenes son enviadas al móvil y procesadas en él para después actuar sobre las lentes cambiando su potencia si es necesario. Además, todos los parámetros se ajustan en la misma aplicación. La figura 3.7 muestra dos capturas de pantalla dónde se ven las partes más importantes de la app de control. En primer lugar se muestra la previsualización de los ojos y el seguimiento pupilar. Y en segundo lugar se muestran algunas de las opciones de configuración disponibles. Por otra parte, un diagrama mostrando el funcionamiento básico del algoritmo de control se muestra en la figura 3.8.

Así, lo primero que hace el sujeto es la calibración, la cuál es muy importante, pero también muy sencilla. Se le pide al sujeto que mire un objeto lejano enfrente de él o ella mientras se almacenan las posiciones de sus pupilas. Además, la calibración es muy

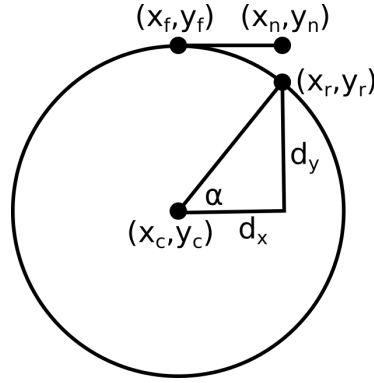


Figura 3.9: Información del ojo usada para calcular la dirección de mirada del sujeto. (x_c, y_c) es el centro del ojo, (x_f, y_f) es el centro de la pupila cuando el sujeto está mirando lejos, (x_n, y_n) es el centro de la pupila en la imagen cuando el sujeto está mirando cerca, (x_r, y_r) es el centro real de la pupila cuando el sujeto está mirando cerca.

rápida, puesto que solo hacen falta unas pocas imágenes para determinar con precisión la posición. Inmediatamente después de la calibración, el sistema se inicia y activa las lentes comenzando a aplicar la corrección. A partir de ese momento el móvil está continuamente capturando y procesando imágenes, por lo que se ha añadido un procedimiento de descarte de imágenes en caso de que la anterior aún no se haya terminado de procesar. Asimismo, la corrección aplicada se recalcula constantemente después de cada nueva imagen recibida.

Para calcular la dirección de mirada del sujeto se han utilizado algunas funciones trigonométricas y la información obtenida durante la calibración. En primer lugar se calcula la posición de la pupila en el ojo. Para ello se calcula la intersección entre el ojo y posición de la pupila en la imagen de la cámara. La figura 3.9 muestra los datos que se han utilizado y calculado en el ojo, donde (x_f, y_f) es la posición de la pupila cuando el sujeto está mirando lejos durante la calibración. Mientras que (x_n, y_n) es la posición de la pupila en la imagen de la cámara cuando el sujeto mira cerca. (x_c, y_c) es el centro del ojo y (x_r, y_r) es la posición real de la pupila mientras que mira un estímulo cercano. Finalmente, *alfa* es la rotación actual del ojo. Así, la posición real de la pupila se calcula usando la fórmula:

$$d_y = \sqrt{d_x^2 + r^2} \quad (3.1)$$

y la rotación del ojo:

$$\alpha = \text{atan}\left(\frac{d_y}{d_x}\right); \quad (3.2)$$

Una vez calculada la rotación de los dos ojos se puede calcular la distancia a la que el

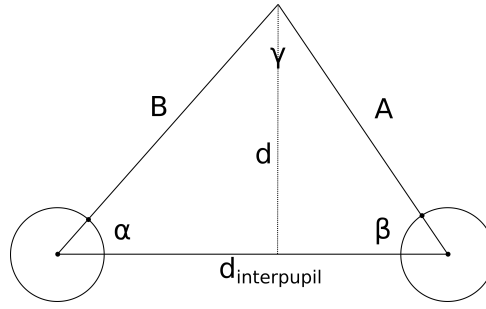


Figura 3.10: Aparecen dos ojos mirando un estímulo cercano. La rotación de los ojos (α y β), que ha sido previamente calculada, se utiliza para calcular la distancia a la que se encuentra el estímulo.

sujeto está mirando. La figura 3.10 muestra la información utilizada para calcular dicha distancia: la rotación de los dos ojos y el IPD (distancia entre las pupilas cuando se mira lejos). La interdistancia pupilar se calcula durante la calibración.

Primero se calculan A y B:

$$A = d_{interpupil} * \frac{\sin(\alpha)}{\sin(\gamma)} \quad (3.3)$$

$$B = d_{interpupil} * \frac{\sin(\beta)}{\sin(\gamma)} \quad (3.4)$$

y finalmente la distancia del estímulo:

$$d = \sin(\alpha) * B; \text{ o } d = \sin(\beta) * A \quad (3.5)$$

3.4.4 Comportamiento del sistema

Los movimientos del ojo humano se pueden producir a gran velocidad, lo que implica que el sistema de control de las lentes puede generar cambios muy bruscos en la potencia aplicada por las lentes. Sin embargo, los ojos humanos no se comportan de esa forma en lo que respecta a la acomodación, haciendo que si se generan cambios muy bruscos en la potencia óptica de las lentes, los sujetos puedan sentirse incómodos. El tiempo medio de reacción en humanos desde que se muestra un estímulo cercano hasta que la acomodación empieza se ha medido en 0.37 segundos [62]. Además, la acomodación tarda una media de 0.25 segundos para un cambio de 2 dioptrías. Así que para evitar la sensación de brusquedad se suaviza el cambio, generando varios escalones intermedios a una velocidad

de 50 dioptrías por segundo. A esta velocidad, el ejemplo de dos dioptrías tardaría 40ms en completar la transición. Eso sigue siendo bastante más rápido que el comportamiento medido por *Campbell y Westheimer* pero es suficientemente lento para evitar la sensación desagradable que producen los cambios bruscos. Además, hay que tener en cuenta que los movimientos acomodativos en un ambiente real son voluntarios y, por tanto, es de esperar que los tiempos de reacción sean mucho menores.

La figura 3.11 muestra un ejemplo de una respuesta típica del sistema cuando se detecta la acomodación. Al inicio se incluye el tiempo que el sistema tarda en capturar y procesar una imagen puesto que este genera un retraso en la reacción. Asimismo, se ha incluido el tiempo de reacción que según las especificaciones tiene la lente Optotune EL-16-40-TC, que es de 7ms desde que se solicita la nueva potencia hasta que efectivamente se aplica. Después se tiene en cuenta el suavizado aplicado en el cambio de la potencia óptica (región violeta en la figura) por los motivos explicados en los párrafos anteriores, que en este ejemplo ilustrativo son 80ms ya que el cambio aplicado son 4 dioptrías. Finalmente, se ha añadido un período de 40ms que es el tiempo que la lente tarda en estabilizarse totalmente.

La figura 3.11 muestra como se produciría el cambio en la potencia de la lente si no se aplicara el suavizado de la misma (línea verde). El cambio sería mucho más rápido, pero la sensación del sujeto sería incómoda.

La figura 3.12 muestra un ejemplo en el que se pidió a un sujeto que mirara alternativamente lejos y cerca. Cuando los ojos estaban más cerca (corresponde con un valor más alto de la distancia en la gráfica) mayor era la potencia aplicada, mientras que cuando los ojos estaban más lejos menor era la potencia aplicada. Esta figura sirve para mostrar un caso de uso práctico de las gafas y su desempeño ante tal situación. En la actividad cotidiana en multitud de ocasiones se cambia la distancia focal alternativamente, por ejemplo, delante de un ordenador mirando alternativamente a la pantalla y al teclado.

Por otra parte, se ha llevado a cabo un experimento con ojos artificiales para comprobar que el ángulo de giro de los ojos calculado durante las convergencias era correcto. Para ello se han empleado dos ojos artificiales situados sobre unas monturas de rotación con una precisión de 2 grados. Con ese sistema se han probado varias posiciones para cada ojo, incluyendo posiciones con los dos ojos en el mismo ángulo y con diferentes ángulos. En la figura 3.13 se recogen algunos de los resultados mostrando la buena relación obtenida entre los ángulos esperados y los ángulos medidos.

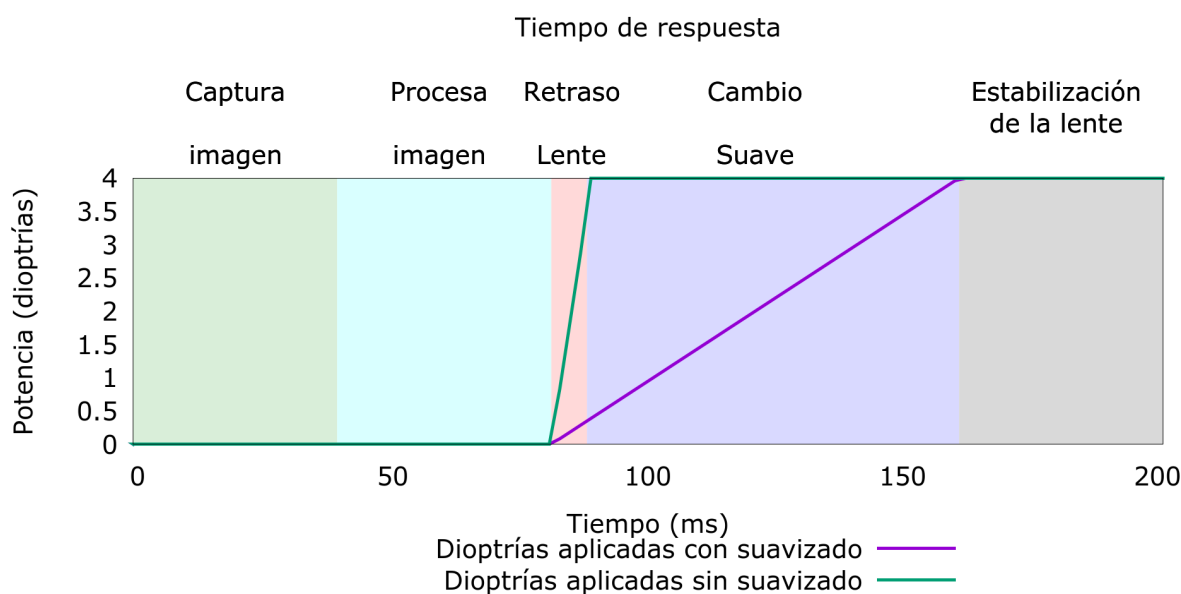


Figura 3.11: Esquema que muestra los tiempos de respuesta de los diversos componentes del sistema bajo dos configuraciones para el control de las lentes.

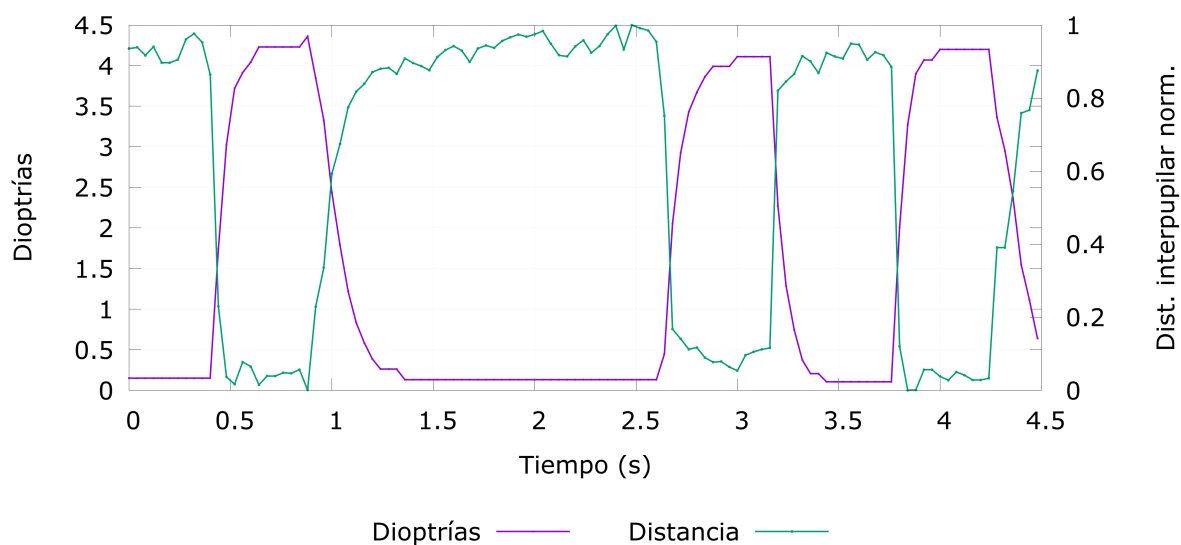


Figura 3.12: Ejemplo de caso de uso de un sujeto mirando alternativamente lejos y cerca. La línea verde muestra el movimiento de las pupilas y la violeta las dioptrías aplicadas por las lentes.

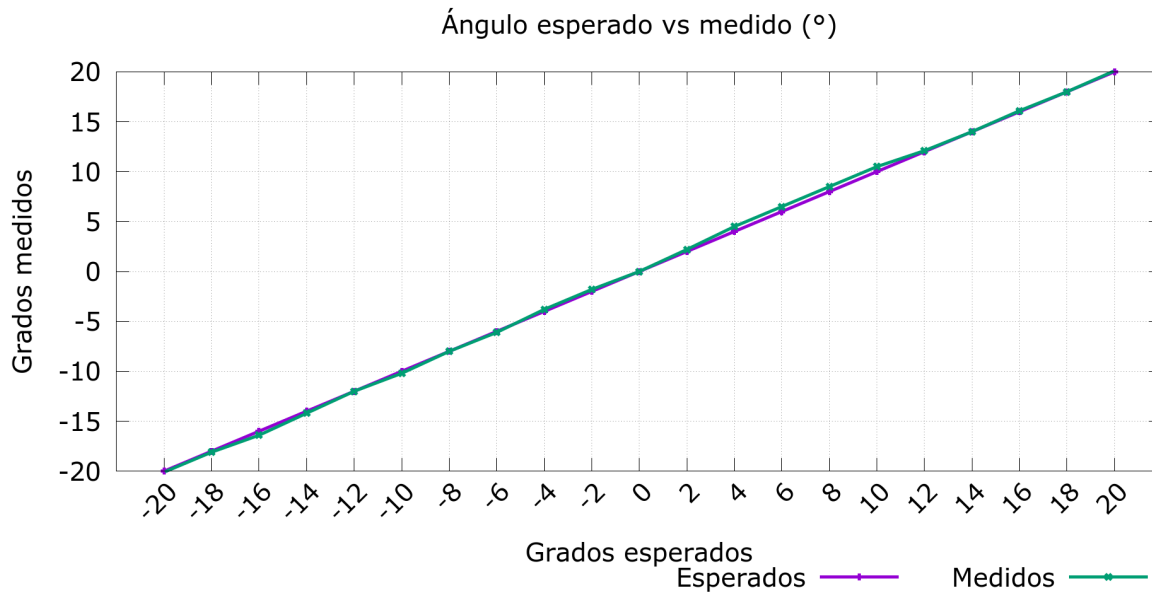


Figura 3.13: Precisión de la medida del ángulo de giro de un ojo artificial, comparación entre los grados medidos y los esperados.

3.5 Análisis de rendimiento, eficiencia y comportamiento

3.5.1 Metodología

Para analizar el rendimiento de las tres plataformas hardware consideradas, se ha utilizado un vídeo que contiene 251 imágenes grabado con las Gafas de Corrección Automática de la Presbicia. El vídeo se grabó a una resolución de 1280x960 píxeles para tener la resolución más alta posible, pero para obtener resultados más cercanos a lo que es realista procesar en tiempo real se redujo su resolución a tamaños de 640x480 y 320x240 píxeles. En el apartado de resultados 3.5.2 se mostrará el rendimiento de las plataformas hardware evaluadas con las tres resoluciones, a excepción de las FPGAs en las que solo fue posible compilar el algoritmo ajustado para procesar las imágenes con la menor resolución, esto es, la de 320x240 píxeles.

3.5.2 Rendimiento

La figura 3.14 muestra el rendimiento obtenido por los cinco dispositivos probados con tres resoluciones. El rendimiento de la implementación para FPGA puede parecer pobre, pero no es sorprendente debido a las optimizaciones básicas que se han aplicado

a dicha implementación. Tal y como se ha comprobado en investigaciones previas las implementaciones OpenCL diseñadas para GPUs pueden tener un rendimiento no muy bueno cuando se ejecutan en FPGAs. *Zohouri et al.* [63] han comparado el rendimiento que obtienen kernels OpenCL optimizados para GPUs y otros optimizados para FPGAs y han encontrado grandes diferencias entre los dos. En sus pruebas ejecutaron la suite de benchmarks Rodinia y observaron que solamente cuando se aplican optimizaciones avanzadas para FPGAs el rendimiento era similar al obtenido por la implementación GPU.

El rendimiento obtenido por la GPU de sobremesa (NVIDIA 980 GTX) es ciertamente sobresaliente. Aunque también es el que cabía esperar debido a su enorme capacidad de cómputo y el enorme ancho de banda si se compara con las otras plataformas probadas. A pesar de ello, la GPU de sobremesa es una opción inviable para controlar las Gafas de Corrección Automática de la Presbicia debido a su gran tamaño y elevado consumo. Pero ofrece una visión más amplia de los resultados y enriquece la comparación.

Finalmente, los resultados de rendimiento obtenidos por el Samsung Exynos 8890 y el Qualcomm Snapdragon 650 son similares, siendo el Exynos ligeramente más lento que el Snapdragon. Además, el tiempo de procesamiento de las GPUs móviles no se incrementa al mismo ritmo que el tamaño de las imágenes. Esto es debido a dos factores, la sobrecarga de las llamadas a las funciones de OpenCL que se mantiene constante y tiene un peso significativo; y a un mejor uso de los recursos hardware disponibles al aumentar el tamaño de las imágenes. Si se observa con detenimiento la figura 3.14 se ve que solamente las imágenes de 320x240 píxeles se procesan suficientemente rápido como para alcanzar el procesamiento en tiempo real requerido por la aplicación.³

La figura 3.15 ofrece una visión mucho más clara de como se distribuye el tiempo de ejecución entre las distintas fases de la búsqueda. Además, en esta gráfica se han incluido los tiempos de ejecución de la implementación FPGA sin optimizar, para mostrar que a pesar de no haber utilizado optimizaciones avanzadas para FPGAs el rendimiento ha mejorado mucho gracias a las optimizaciones aplicadas. La fase de preprocesado de las imágenes que se ejecuta al inicio del algoritmo ha obtenido una aceleración de hasta 17x, mientras que la fase de búsqueda de puntos en el borde de la pupila se ha acelerado hasta un 10x. Por supuesto, las implementaciones que se ejecutan en las GPUs siguen mostrando un rendimiento algo superior a las implementaciones que se ejecutan en las FPGAs. Sin

³Se considera tiempo real cuando se procesan al menos 24 imágenes por segundos, esto es, el procesamiento de cada imagen lleva 42ms o menos.

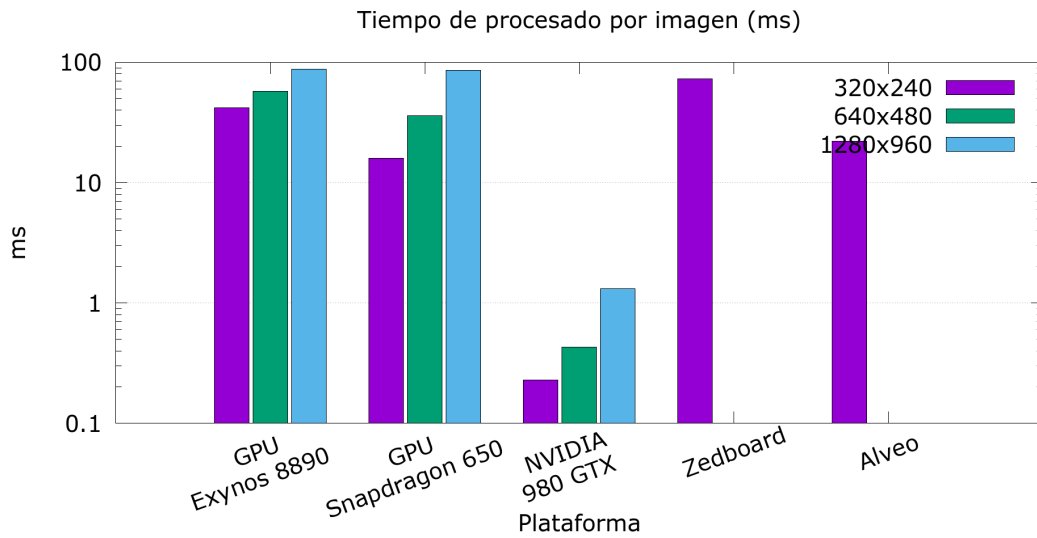


Figura 3.14: Comparación de rendimiento de los 5 dispositivos probados con 3 resoluciones diferentes.

embargo, tras las optimizaciones, el rendimiento de las FPGAs es comparable al obtenido por las GPUs.

3.5.3 Eficiencia

La eficiencia es un factor clave para las Gafas de Corrección Automática de la Presbicia, pues uno de sus requisitos es que estén alimentadas por una batería y que tengan una autonomía razonable. Así, se ha analizado el consumo energético de los cinco dispositivos probados, cuyos resultados se muestran en la figura 3.16. Para ello la figura muestra los Julios consumidos por cada dispositivo para procesar imágenes en cada una de las tres resoluciones consideradas. Nótese que nuestra métrica de eficiencia energética es rendimiento/vatio (u operaciones por segundo por vatio), y puesto que el rendimiento de nuestra aplicación viene determinado por las imágenes procesadas por segundo (fps) a las que es capaz de operar, resulta que: $\text{rendimiento/vatio} = \text{imágenes/segundo/vatio} = \text{imágenes/julio}$. Es por ello que en la figura 3.15 se muestran los julios requeridos por cada imagen procesada (julios/imagen) como una forma más clara de representar la eficiencia energética de cada plataforma.

El coste energético de procesar una imagen varía desde los 0.016J para una imagen de 320x240 píxeles para la GPU del SoC Exynos 8890 hasta los 0.43J de la Alveo u250 para

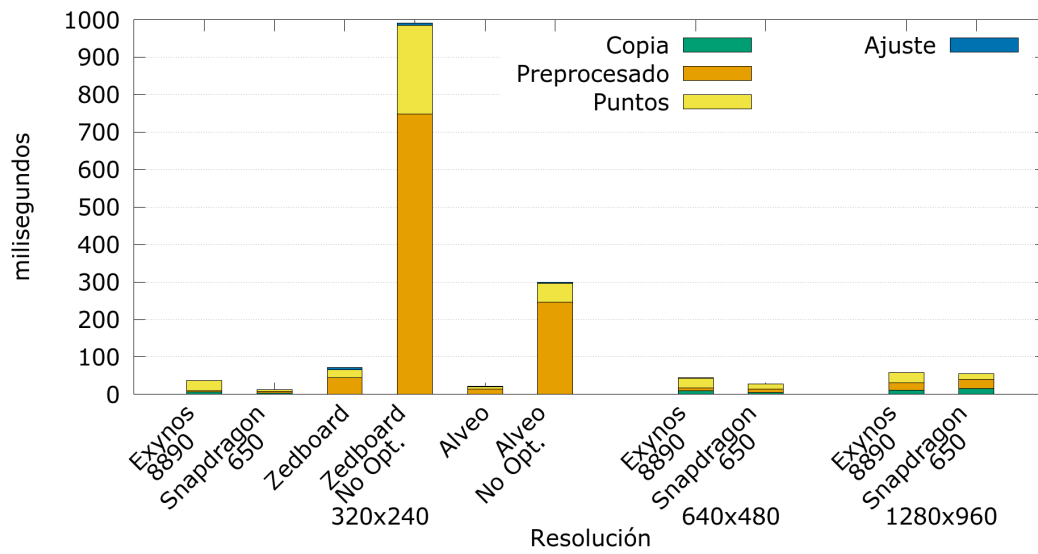


Figura 3.15: Detalle del tiempo empleado por cada dispositivo para tres resoluciones en terminar cada una de las etapas del procesado. El tiempo de *Copia* es el tiempo que se tarda en copiar la imagen, si es necesario, así como los resultados a cualquier dato intermedio. El *Preprocesado* es el tiempo empleado en aplicar las funciones de procesamiento de imágenes. Mientras que *Puntos* es el tiempo que se tarda en buscar los puntos en el borde de la pupila. Y *Ajuste* hace referencia al tiempo empleado en ajustar una elipse a los puntos y seleccionar la mejor.

una imagen de la misma resolución. Así mismo, la cantidad de energía utilizada aumenta al incrementarse el tamaño de las imágenes como era de esperar. Se puede ver como para las tres plataformas en las que se han podido procesar imágenes de varias resoluciones el consumo sigue la tendencia descrita. Al igual que en el caso del rendimiento las FPGAs muestran una eficiencia baja debido a una implementación que no está completamente optimizada como exponían *Zohouri et al.* [63]. La potencia disipada por las FPGAs es relativamente elevada, siendo 6.6w en el caso de la placa Zedboard y 26.4w en el caso de la Alveo u250. Si bien se ha llevado a cabo un importante esfuerzo para optimizar la implementación que se ejecuta sobre las FPGAs. Compilando con distintas cantidades de desenrollado de bucles, o habilitando más puertos de acceso a memoria. También aplicando optimizaciones específicas para FPGAs, a pesar de ello, el rendimiento y eficiencia obtenidos siguen estando por debajo de lo que cabría esperar. Por otra parte, la GPU de sobremesa es un caso especial en lo que respecta a eficiencia, como se vio en la figura 3.14 su rendimiento es muy bueno, pero su potencia disipada es muy elevada, acercándose a los 100 vatios variando según el tamaño de la imagen. El resultado es que la eficiencia obtenida es muy buena y es, de hecho, comparable a la obtenida por las GPUs móviles.

En lo que respecta a las GPUs móviles ambas obtienen buenos resultados, ligeramente mejores en el caso del SoC Exynos. Evidentemente las GPUs móviles están diseñadas para ser energéticamente eficientes debido a que se alimentan de una batería, por lo que este resultado era de esperar. Teniendo en cuenta que las Gafas de Corrección Automática de la Presbicia requieren un bajo consumo las GPUs móviles resultan una elección adecuada a la vista de los resultados obtenidos.

3.5.4 Experimentos con sujetos

Para realizar los experimentos con sujetos se ha desarrollado un test de agudeza visual de cerca. Los test de agudeza visual se presentan de muchas formas, siendo el más ampliamente utilizado la gráfica de Snellen que presenta letras de varios tamaños. Aunque también los hay que utilizan textos completos. En esta caso se ha optado por incluir fragmentos de texto de diferentes tamaños. Para ello se ha partido de los tamaños de letra utilizados por *Sanders et al.* [64] incluyendo agudezas visuales que van desde 0.16 hasta 1.0. La tabla 3.2 muestra las equivalencias de las agudezas visuales utilizadas en varias escalas y su equivalente en la fuente utilizada. El test desarrollado se muestra en la figura 3.17.

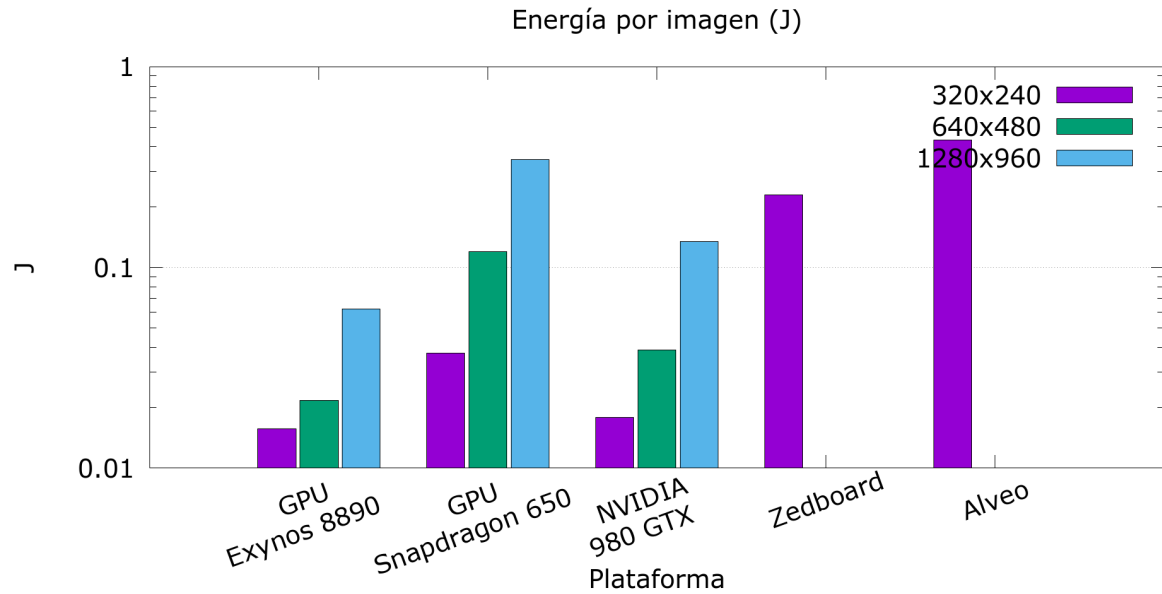


Figura 3.16: Comparación del consumo energético de cada dispositivo para procesar una imagen con tres resoluciones diferentes.

Decimal	ETDRS	Jaeger	Tamaño de la fuente
0.16	20/125	J12	14
0.18	20/114	J11	13
0.2	20/100	J10	12
0.25	20/80	J9	9.5
0.32	20/63	J8	7.5
0.4	20/50	J6	6
0.5	20/40	J5	4.5
0.63	20/32	J4	3.5
0.8	20/25	J2	3
1.0	20/20	J1	2.5

Tabla 3.2: Equivalencias entre el tamaño de la fuente Times New Roman y la escala de agudezas visuales utilizadas para generar el test de agudeza visual de cerca.

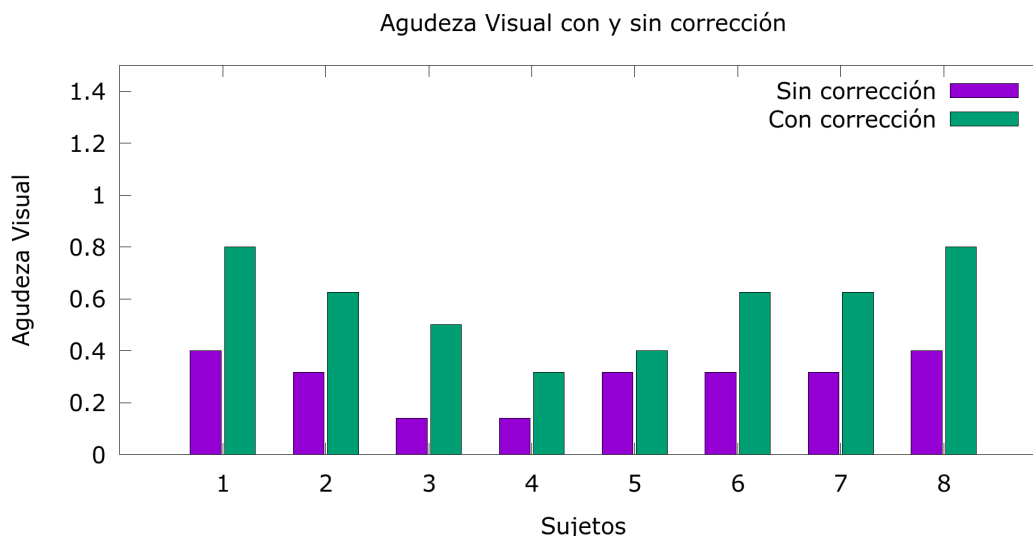


Figura 3.18: Agudeza visual de los sujetos antes y después de activar la corrección de la presbicia.

Todos los tests de agudeza visual se situaron en la misma posición, siendo el sujeto el que se desplazaba a la distancia correspondiente en cada caso. De esta forma también se puso en práctica la movilidad de las Gafas de Corrección Automática de la Presbicia que permite al sujeto desplazarse mientras las usa. Las tres correcciones aplicadas fueron:

- **Corrección de lejos:** Se midió la refracción de los sujetos y se aplicó la corrección correspondiente con lentes de prueba. Generando la mejor visión posible para lejos pero mala visión de cerca.
- **Corrección de cerca:** De la misma manera se aplicó la corrección correspondiente para obtener una potencia total (corrección + ojo) de 2.5D. Generando una buena visión a unos 40cm pero mala visión más cerca o más lejos.
- **Gafas de Corrección Automática de la Presbicia:** En este caso el sujeto lleva puestas las gafas auto-acomodativas que se encargan de generar la mejor corrección para la distancia a la que esté mirando el sujeto.

En la figura 3.19 se muestran los resultados obtenidos por los sujetos en el segundo experimento. En la línea violeta se muestra la agudeza visual obtenida con la corrección de lejos, como es de esperar con esta corrección la agudeza medida es mejor para las distancias más lejanas y peor para las más cercanas. La línea verde muestra el resultado

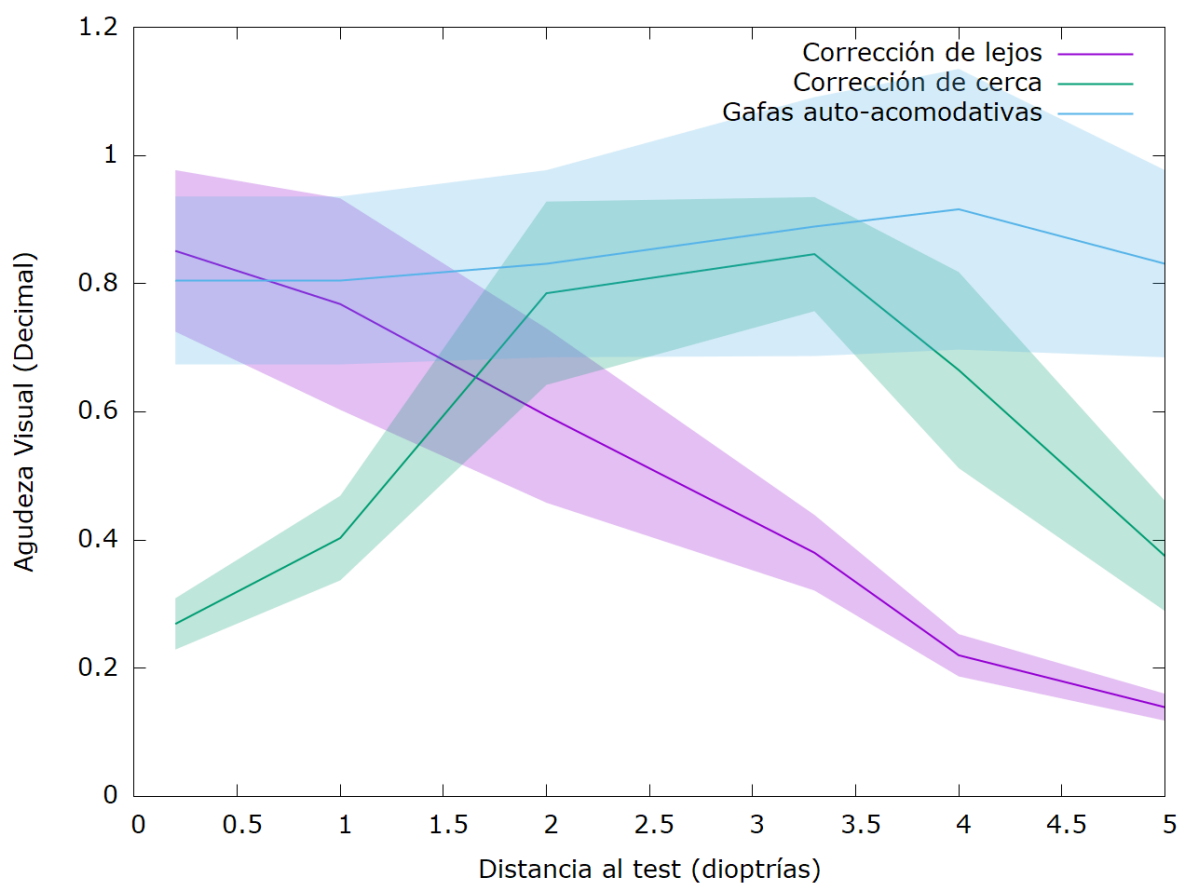


Figura 3.19: Agudeza visual medida para 8 sujetos usando 3 correcciones diferentes y a 6 distancias. En violeta se muestra la agudeza visual medida utilizando la corrección para lejos, mientras que en verde se muestran los resultados obtenidos con la corrección para cerca (2.5D), finalmente en azul se muestra agudeza utilizando las gafas auto-acomodativas. La agudeza visual se muestra en el eje y en escala decimal, y en el eje x se presentan las 6 distancias a las que se ha hecho el test.

de la corrección para cerca, en concreto se empleó una corrección de 2.5D, con lo que los mejores resultado se han obtenido para las distancias de 50cm y 30cm, 2D y 3.3D respectivamente. Mientras que el resto de distancias obtienen una agudeza bastante pobre con esta corrección debido a lo inapropiado de esta corrección tanto para lejos como para muy cerca. Finalmente, la línea azul presenta los resultados obtenidos por la corrección con las gafas auto-acomodativas, en este caso se puede ver como la agudeza visual se mantiene muy estable en todas las distancias. Este es el resultado que cabía esperar ya que el rango de acomodación de las gafas auto-acomodativas comprende todas las distancias que se han medido en este experimento.

En resumen, las gafas acomodativas han obtenido un resultado significativamente superior a las correcciones estáticas. Pero para cuantificarlo se ha llevado a cabo una prueba t de Student para muestras pareadas, debido a que el comportamiento es diferente para cada distancia se han comparado sólo algunas de ellas. Así, hay una diferencia significativa en la agudeza visual medida a 0.5m con la corrección estática de lejos ($M=0.59$, $SD=0.06$) y con las gafas auto-acomodativas ($M=0.78$, $SD=0.14$) con condiciones $t(7)=-4.3658$, $p=0.003291$. Se han hecho dos pruebas t de Student más, en este caso comparando la corrección estática de cerca y las gafas auto-acomodativas a 5m y 0.2m. Para la distancia de 5m se ha medido una diferencia significativa entre la corrección estática de cerca ($M=0.27$, $SD=0.04$) y las gafas auto-acomodativas ($M=0.80$, $SD=0.13$) con condiciones $t(7)=-9.4305$, $p=0.0000314$. Para 0.2m también se ha medido una diferencia significativa entre la corrección estática de cerca ($M=0.37$, $SD=0.09$) y las gafas auto-acomodativas ($M=0.83$, $SD=0.15$) con condiciones $t(7)=-8.994$, $p=0.00004284$.

4 Procesado de imágenes de Hartmann-Shack de alto rendimiento

4.1 Introducción

El sensor de frentes de onda Hartmann-Shack se ha utilizado ampliamente en óptica desde hace mucho años y se han publicado numerosos trabajos sobre la aceleración del procesado de las imágenes de H-S, especialmente sobre mediciones de sistemas ópticos artificiales, tales como los telescopios.

Algunos trabajos previos han conseguido obtener tiempos de procesamiento muy bajos solamente utilizando la CPU, como el desarrollado por Mocci *et al.* [65]. Sin embargo, su implementación está orientada a la calibración de láseres, dónde es posible utilizar imágenes de menor resolución que en el ojo y algoritmos más simples para buscar los centroides de los puntos. Además, dado que su sistema es estático no es necesario utilizar una solución de seguimiento pupilar como en el caso del ojo humano. Yu and Zhang [66, 67] han desarrollado otra implementación CPU que consigue alcanzar 110 imágenes por segundo, aunque si sólo se tiene en cuenta el procesamiento puede alcanzar hasta 162 imágenes por segundo en imágenes de 512x512 píxeles con 193 microlentes. Al igual que en el trabajo anterior, se trata de imágenes estáticas en las que no es necesario llevar a cabo un seguimiento pupilar y su resolución no es muy alta.

La primera implementación GPGPU para la búsquedas de los centroides y cálculo de los Zernikes en imágenes de Hartmann-Shack fue desarrollada por Marichal-Hernández *et al.* [68]. Sin embargo, esta implementación fue desarrollada para telescopios (nuevamente se trata de imágenes estáticas y no de ojos humanos), consiguiendo una aceleración de 10x para la etapa de búsqueda de centroides y de 2x para la etapa de la reconstrucción del frente de onda.

Otra propuesta que obtiene una alta velocidad de procesamiento ha utilizado una FPGA

(Field Programmable Gate Array) y un ASIC (Application Specific Integrated Circuit) para conseguir procesar 830 imágenes por segundo [69]. Sin embargo, la precisión estaba limitada por la simplicidad del algoritmo de detección de centroides y la cantidad limitada de patrones utilizados durante el entrenamiento de la red neuronal. En dicha implementación se busca el punto más brillante dentro de cada microlente para encontrar su centroide y calcular su desplazamiento. La red neuronal se utiliza para relacionar los desplazamientos encontrados con unos desplazamientos que se utilizan previamente para entrenarla. Usando la FPGA y el ASIC han sido capaces de desarrollar un sistema autónomo que, como se ha mencionado, es capaz de medir la aberración del frente de onda a 830 imágenes por segundo. Por otra parte, una vez más esta implementación está orientada al procesamiento de sistemas ópticos artificiales tales como telescopios. Y, por lo tanto la precisión y robustez no son suficientes para medir las aberraciones de un sistema óptico dinámico y vivo como el del ojo humano.

Reichenbach *et al.* han desarrollado otra propuesta muy interesante utilizando la combinación de una FPGA, una CPU y una GPU para procesar las imágenes de H-S [70]. Aunque esta investigación también estaba enfocada en el procesamiento de imágenes de H-S de sistemas ópticos artificiales, particularmente en telescopios. A pesar de ello, su método para buscar los centroides es muy interesante y el rendimiento obtenido muy bueno. Su algoritmo consta de dos fases: en una primera fase los centroides potenciales se buscan seleccionando los píxeles más brillantes en un área de 9x9 píxeles; en la segunda fase los centroides finales se seleccionan llevando a cabo una correlación entre unos centroides iniciales y los anteriormente encontrados. La FPGA se utiliza para la búsqueda del centroide de cada punto, mientras que la CPU se utiliza para correlacionarlos. Finalmente la GPU se usa para recuperar la aberración del frente de onda. Con esta implementación han conseguido procesar hasta 294 imágenes por segundo con una configuración de 100x100 microlentes en imágenes de 1024x1024 píxeles. Esta es la implementación más rápida que hemos encontrado en la literatura. Nuevamente esta implementación está orientada a sistemas ópticos artificiales. Aún así, debido a que es el trabajo más reciente y con mejor rendimiento de los encontrados lo hemos seleccionado para poder compararnos tanto en rendimiento como en precisión con nuestro algoritmo PAPYCS.

Mauch y Reger [72, 71] también han utilizado FPGAs para obtener una alta integración con la cámara y ser capaces de procesar 905 imágenes por segundo con una resolución de 224x224 píxeles. Además, emplean un algoritmo dinámico para procesar grandes aberraciones. Primero detectan los puntos utilizando un algoritmo de etiquetado

de componentes conectados para después reordenarlos para relacionar sus posiciones con las de los puntos de referencia.

Cuando se procesan imágenes de H-S del ojo humano es imprescindible conocer la posición y el tamaño de la pupila del sujeto para poder procesarla. Además, su posición y tamaño variarán en la mayoría de sistemas a lo largo de la medida, por lo cuál es necesario conocer su posición para cada imagen procesada y no sólo para la primera. Por otra parte, es posible utilizar otra cámara para tomar imágenes en infrarrojo del ojo y llevar a cabo un seguimiento pupilar en esa imagen. Sin embargo, ello añade complejidad al sistema óptico (requiriendo un perfecto alineamiento con la cámara de Hartmann-Shack) y también más complejidad de procesamiento. Por suerte, es posible realizar el seguimiento pupilar directamente sobre las imágenes de H-S. Entorno a ese propósito se han propuesto varias soluciones, Meimon *et al.* [73] propone integrar los valores de los píxeles sobre cada microlente, umbralizarlos y ajustar una elipse a los puntos del borde. Este método ha sido caracterizado con una gran precisión, pero no se ha desarrollado una versión de alto rendimiento del mismo. Otro método ha sido desarrollado por Arines *et al.* [74], en este caso se umbraliza la imagen de H-S y se calcula el centroide de la imagen resultante. Sin embargo, este método presenta dos problemas importantes: en primer lugar, no es capaz de medir el radio de la pupila; y en segundo lugar, no funciona correctamente con pupilas que se encuentran parcialmente fuera de la imagen o que están parcialmente ocultas por el párpado. Finalmente, de Castro *et al.* [75] han desarrollado otro método de sistema pupilar utilizando las posiciones de los puntos detectados y su brillo. Y han desarrollado un sistema de bucle cerrado para probar las ventajas de utilizar el seguimiento pupilar en el mismo.

4.2 Paralelización del procesamiento de imágenes de H-S

El algoritmo utilizado para procesar las imágenes de Hartmann-Shack se puede dividir en dos etapas totalmente diferenciadas, a saber: seguimiento pupilar y procesamiento de Hartmann-Shack. En las subsecciones 4.2.1 y 4.2.2 se describe el funcionamiento de los algoritmos, mientras que la paralelización y optimización de los mismos se explica posteriormente en la sección 4.3.

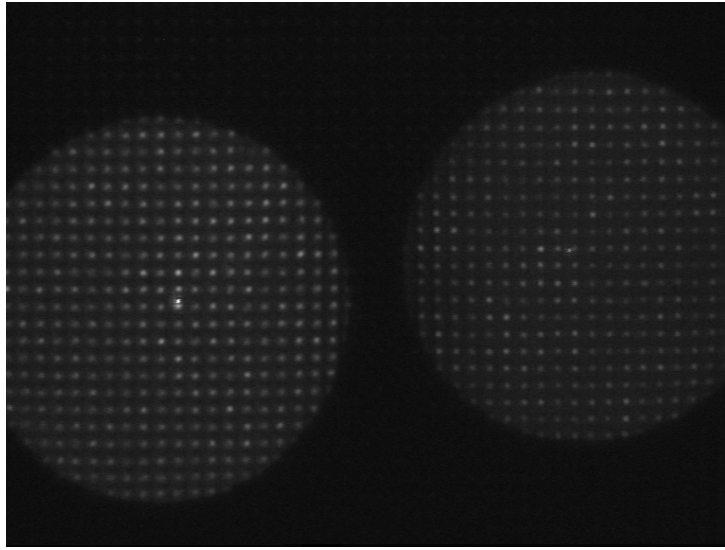


Figura 4.1: Imagen tomada con un sensor de frentes de onda de Hartmann-Shack. Se muestran los dos ojos de un sujeto fusionados utilizando un prisma.

4.2.1 Seguimiento pupilar

Como ya se ha mencionado con anterioridad, determinar la posición y tamaño de la pupila del sujeto con precisión es crucial para poder procesar correctamente las imágenes de Hartmann-Shack. Las imágenes de Hartmann-Shack en ojos humanos suelen presentar un aspecto similar a la que se muestra en la figura 4.1. Dicha imagen se tomó con el sistema descrito en [14], y en ella se pueden ver dos ojos que han sido *desplazados* mediante un prisma para que cupieran en el sensor de la cámara. Para realizar el seguimiento de la pupila se utiliza la gran diferencia de luminosidad entre la zona de la pupila y el resto de la imagen.

En este caso la imagen es binocular, por lo que se divide por la mitad y cada mitad se procesa de forma independiente. De forma que las imágenes se procesan internamente como si fueran monoculares, abstrayendo al procesamiento de las particularidades de este sistema. Esto hace que el algoritmo de seguimiento tenga un ámbito de aplicación mayor, puesto que la mayoría de los sistemas de Hartmann-Shack son monoculares.

La figura 4.2 muestra el proceso utilizado para llevar a cabo el seguimiento pupilar. A continuación se describen los pasos que se emplean:

1. En primer lugar en 4.2a se muestra la imagen original que se corresponde con la mitad izquierda de la imagen mostrada en la figura 4.1.
-

2. El primer procesado que se aplica a la imagen es un filtro de Gauss 2D para suavizar la imagen, reduciendo de esa forma los picos y distribuyendo más uniformemente tanto la parte de la pupila como la parte del fondo. El resultado se muestra en 4.2b.
 3. A continuación se lleva a cabo una umbralización para separar la pupila del fondo. Para determinar el umbral a utilizar se ha usado el algoritmo de Otsu [76]. De esta forma, la umbralización es capaz de adaptarse a los cambios de luminosidad que típicamente se pueden producir en las imágenes de H-S entre sujetos o incluso durante una misma medida. El resultado de la umbralización se muestra en 4.2c. Sin embargo, como se puede apreciar en la figura, el resultado presenta algunas áreas en el borde de la pupila que han sido erróneamente detectadas como fondo. Ello se debe a que la pupila suele presentar menos luminosidad en sus bordes que en el centro.
 4. Para reducir este efecto indeseado se aplica una operación morfológica de cierre, que permite eliminar esos espacios siempre y cuando tengan un tamaño reducido. El resultado se muestra en la figura 4.2d. Dependiendo de la configuración del sistema de Hartmann-Shack, este paso puede requerir ajustar el radio de la operación morfológica para cerrar con éxito los espacios vacíos entre áreas umbralizadas.
 5. A continuación se aplica una detección de bordes que busca las zonas de frontera entre la pupila umbralizada y el fondo, comparando cada pixel que se encuentra en la zona umbralizada con sus 8 vecinos y, si alguno pertenece al fondo, se determina que ese punto está en el borde de la pupila. La figura 4.2e muestra los bordes encontrados en la imagen procesada.
 6. Por último, se aplica el método RANSAC (Random Sample Consensus) [24] con ajuste a una elipse. Dicho método permite filtrar de una forma muy eficaz los puntos que se encuentran en el borde de la pupila de los que no. Lo cuál es muy conveniente en imágenes con reflejos o con una umbralización pobre. Con tal propósito se seleccionan aleatoriamente conjuntos de 5 puntos de entre todos los puntos del borde detectados. Sobre cada uno de esos conjuntos de puntos se aplica un ajuste a una elipse y después se comprueba la distancia del resto de puntos a cada uno de esos ajustes. Finalmente, se selecciona la elipse con una mayor cantidad de puntos del borde cercanos a ella. La figura 4.2f muestra la elipse seleccionada para la imagen procesada. Se ha utilizado el ajuste a una *elipse* en lugar del ajuste a un
-

círculo debido a la mayor precisión que es posible obtener con la elipse tal y como se ha comprobado en el capítulo 2. Además, las imágenes de Hartmann-Shack pueden presentar importantes deformaciones globales según la configuración del mismo y las aberraciones del sujeto a medir.

4.2.2 Algoritmo PaPyCS

Las imágenes de Hartmann-Shack presentan una estructura bastante homogénea, aunque dependiendo de las aberraciones del ojo que se midan puede haber variaciones significativas en la misma. El procesado siempre sigue un patrón similar: se buscan los centroides de las microlentes, se calculan los desplazamientos de los mismos con respecto a su posición de referencia y se calculan los valores de los coeficientes. Sin embargo, las imágenes suelen presentar imperfecciones o casos extremos que hay que tener en cuenta. El algoritmo 5 muestra como se han abordado los posibles problemas que pueden aparecer durante el procesado en el algoritmo PAPYCS (Parallel Pyramidal Centroid Search). El algoritmo PAPYCS, propuesto en esta Tesis, realiza una búsqueda piramidal y paralelizada de los centroides de la imagen del sensor H-S. Estos problemas y sus soluciones se detallan más adelante en esta sección.

4.2.2.1 Búsqueda de los centroides

En primer lugar, vamos a centrarnos en la tarea más importante del procesamiento, que no es otra que localizar los centroides de los puntos de cada microlente. Para ello hemos utilizado un algoritmo *piramidal* por su forma de converger. Este algoritmo toma un centro y un tamaño inicial a partir del cuál procesa el área abarcada por una microlente. El centro inicial viene determinado por la posición de dicha microlente en la imagen de referencia y el tamaño inicial por el tamaño de la microlente en el sensor de la cámara. En primer lugar se procesa la imagen de referencia, la cuál se asume que tiene una aberración pequeña y, por tanto, una distribución de los puntos focalizados similar a la que generaría un frente de onda sin aberraciones. De esta forma, es posible procesarla partiendo de los centros iniciales que generaría un frente de onda sin aberraciones.

Tanto la imagen de referencia como la imagen *procesada* se analizan de la misma forma, la cuál se describe a continuación. En la figura 4.3a se muestra un ejemplo del área inicial de búsqueda y el centro inicial de una microlente en una sección de una imagen de H-S.

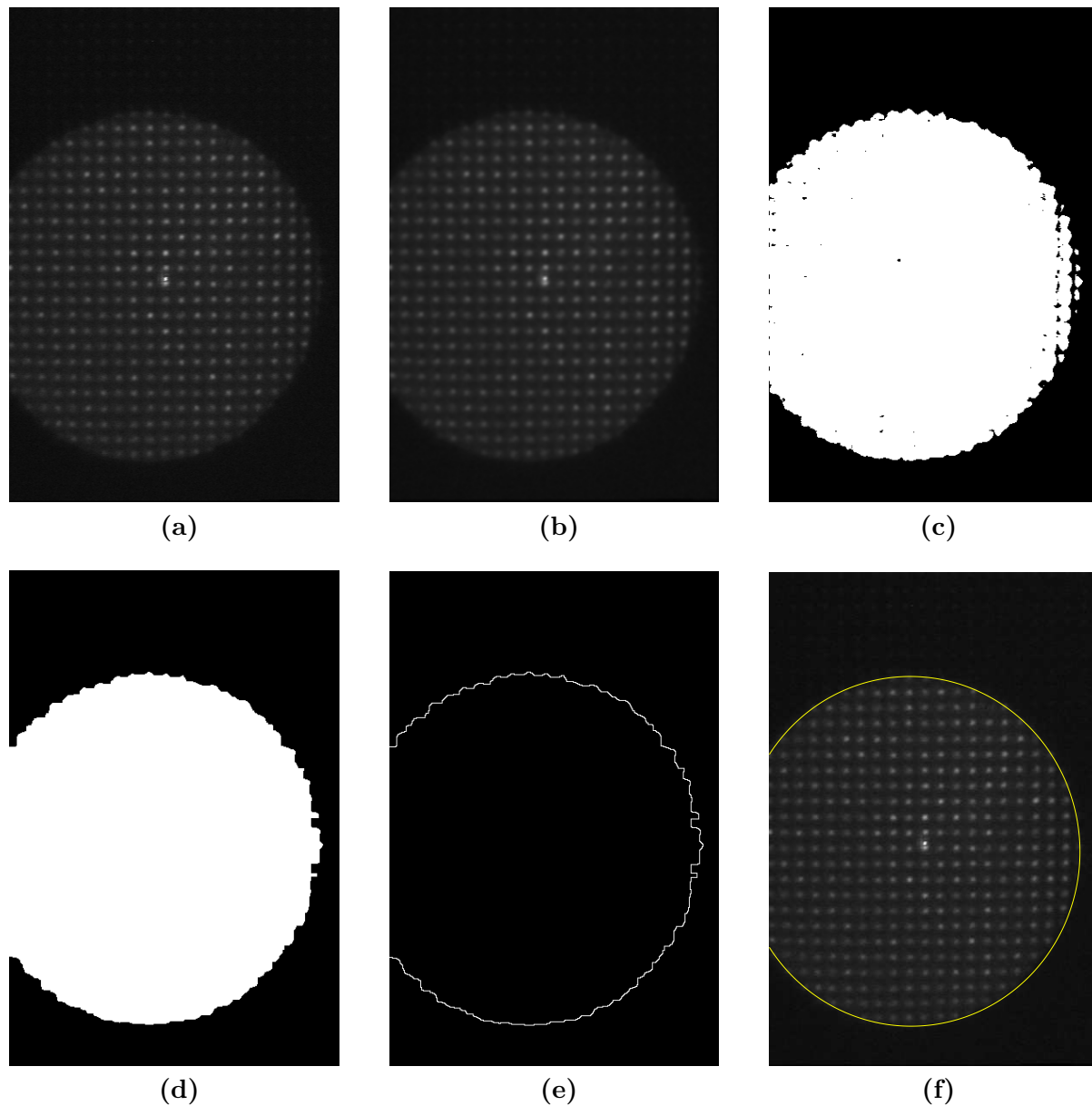


Figura 4.2: (a) Imagen original sobre la que se realiza el seguimiento. (b) Se aplica un filtro de Gauss 2D a la imagen. (c) Se umbraliza la imagen utilizando un algoritmo automático para determinar el umbral. (d) Se aplica una operación morfológica de cierre sobre la imagen. (e) Se detectan los bordes en la imagen. (f) Se ajusta una elipse a los bordes detectados en la imagen.

Algoritmo 5 Pseudocódigo del algoritmo PAPYCS

```

1: procedure PAPYCS(IMedida, IReferencia, Radio)
2:   Pupila  $\leftarrow$  BuscaPupila(IMedida)
3:   CentroidesRef  $\leftarrow$  BuscaCentroides(IReferencia, CentroidesPerfectos, Pupila, Radio)
4:   DesplazamientoCentro  $\leftarrow$  CalculaDesplCentro(IReferencia, IMedida)
5:   CentroidesRef  $\leftarrow$  CentroidesRef + DesplazamientoCentro
6:   CentroidesMed  $\leftarrow$  BuscaCentroides(IMedida, CentroidesRef, Pupila, Radio)
7:   [CentroidesMed, CentroidesRef]  $\leftarrow$  FiltraCentroides(CentroidesMed, CentroidesRef)
8:   if MallaDinámica then
9:     ZernikesDesenfoque  $\leftarrow$  CalculaCoeficientes(CentroidesRef, CentroidesMed)
10:    ZernikesEscalados  $\leftarrow$  EscalaCoeficientes(ZernikesDesenfoque)
11:    Desplazamientos  $\leftarrow$  CalculaDesplazamientos(ZernikesEscalados)
12:    CentroidesRef  $\leftarrow$  CentroidesRef + Desplazamientos
13:    CentroidesMed  $\leftarrow$  BuscaCentroides(IMedida, CentroidesRef, Pupila, Radio)
14:    [CentroidesMed, CentroidesRef]  $\leftarrow$  FiltraCentroides(CentroidesMed, CentroidesRef)
15:   end if
16:   Zernikes  $\leftarrow$  CalculaCoeficientes(CentroidesRef, CentroidesMed)
17:   return Zernikes
18: end procedure

```

Para procesar el área, el algoritmo calcula el valor mínimo de todos los píxeles y calcula el centro de masas sustrayendo el valor mínimo a todos los píxeles, dicha sustracción favorece una más rápida convergencia hacia el área más brillante. En la figura 4.3b se muestra el área inicial con su centro de masa señalado con un punto rojo. Se puede ver como el centro ha convergido hacia el punto de luz formado por la microlente con respecto al centro inicial. A continuación se centra el área de análisis en el nuevo centro, se reduce su tamaño en un píxel y se repite el procesado. En la figura 4.3c se muestra el centro de masas calculado tras tres iteraciones. Asimismo, las figuras 4.3d y 4.3e muestran el centro de masas calculado tras siete y trece iteraciones respectivamente, ya situado prácticamente sobre su posición final. El proceso se repite hasta un tamaño de 3x3 píxeles, cuyo resultado se muestra en la figura 4.3f. Como se puede observar la cantidad de trabajo a realizar disminuye con cada iteración, concentrándose la mayor parte del mismo en la primeras iteraciones, que son, además, las más relevantes pues es donde se producen mayores desplazamientos del centro de masa. Aunque distintas configuraciones de H-S (distancia focal y tamaño de microlente) pueden generar puntos focalizados de distintos tamaños que pueden variar la necesidad de iteraciones a llevar a cabo para converger al centro. Este procesado se lleva a cabo para cada una de las microlentes procesadas. La cantidad de microlentes

a procesar es muy variable entre diferentes sistemas, en las pruebas realizadas hemos utilizado imágenes que contienen desde 250 hasta 2000 microlentes.

Sin embargo, en ocasiones las imágenes procesadas y la imagen de referencia se encuentran desplazadas, y estos desplazamientos pueden ser muy significativos. En la figura 4.4 se muestra un ejemplo de un caso en el que el desplazamiento de la imagen procesada con respecto a la de referencia es exactamente media microlente. En rojo se muestra la imagen de referencia, y en azul la procesada. Como se puede ver, algunos puntos azules están situados entre cuatro de los rojos. Por lo tanto, en este caso se produce la situación tanto en vertical como en horizontal. Esto supone un gran problema, ya que la convergencia del algoritmo piramidal podría generar resultados erróneos. Pudiendo darse dos situaciones igualmente negativas para el resultado: el algoritmo piramidal podría converger al área que se encuentra entre dos puntos, no encontrando el centroide de ninguno de los dos. O algunos puntos podrían converger a la derecha (o arriba) y otros a la izquierda (o abajo) generando un aberración errónea. Un problema similar ocurre si el desplazamiento es incluso superior al de este ejemplo, pues al no poder establecer la relación entre las microlentes de referencia y las microlentes procesadas correctamente el resultado sería erróneo.

Para solventar este problema se opta por decidir que puntos están más cercanos de media en vertical y horizontal. Para ello, antes de procesar la imagen completa se procesa una cruz de puntos centrada en el centro de la pupila, compuesta por siete puntos en vertical y siete puntos en horizontal. Estos siete puntos se procesan utilizando el algoritmo piramidal igual que durante el resto del procesado. Sin embargo, una vez obtenidas las coordenadas de sus centroides se calcula la posición mediana de las mismas. A partir de los puntos analizados en vertical se obtiene la mediana de su coordenada X, de forma que si cuatro de los puntos convergieron a la derecha y tres a la izquierda, la mediana determinará que el procesado debería tomar los puntos de la derecha como los correctos. Lo mismo sucede en horizontal, se calcula la mediana de la coordenada Y determinando que puntos son los correctos. Una vez calculado el desplazamiento entre las posiciones de inicio de referencia y las medianas de las procesadas se aplica dicho desplazamiento a todos los puntos de inicio. Ahora, se vuelve a ejecutar desde esos nuevos puntos de inicio el algoritmo piramidal, evitando que se produzcan situaciones de indecisión y errores. Este sistema se puede denominar un sistema de votación en el que la posición con más votos es la elegida como correcta de un modo similar a como funciona el algoritmo RANSAC.

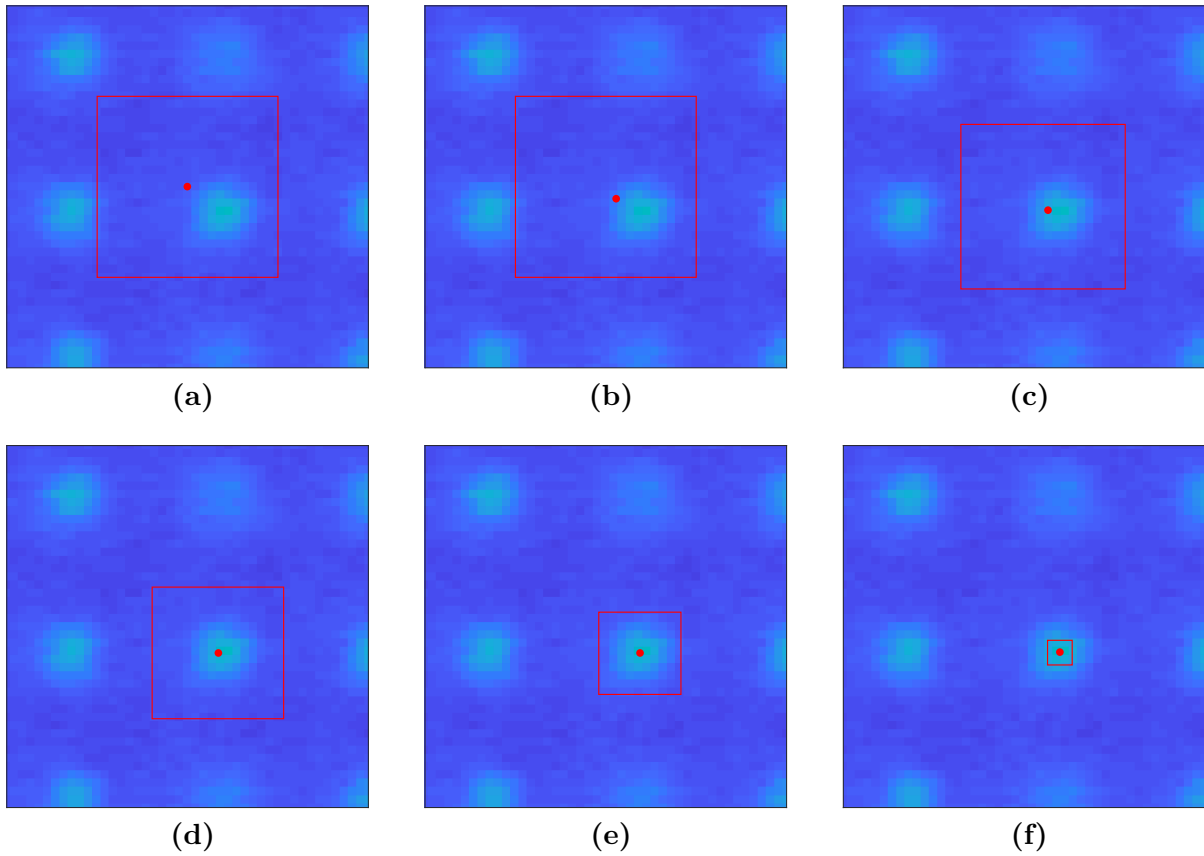
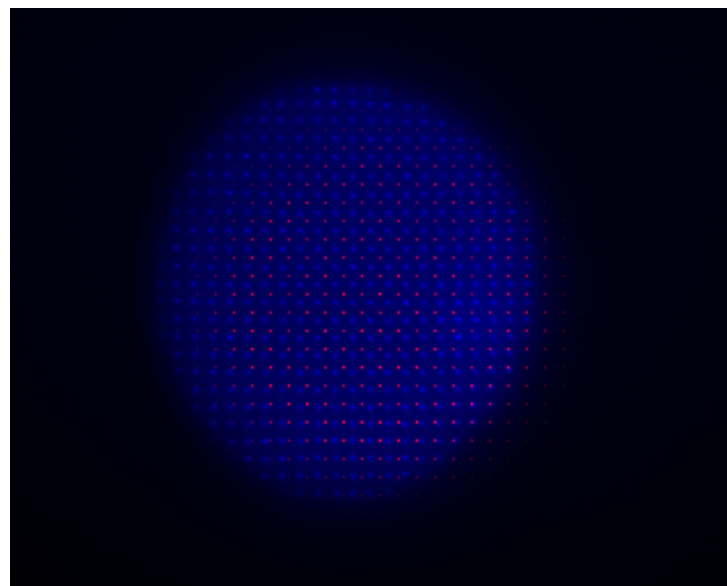
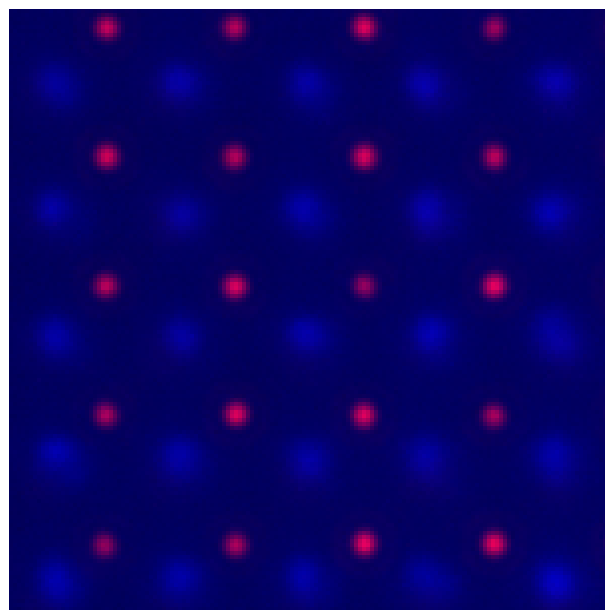


Figura 4.3: (a) Ventana inicial en la que se realiza la búsqueda delimitada por el cuadrado rojo y centro geométrico señalado por el punto rojo. (b) Ventana de 22x22 píxeles con el centro de masas señalado por el punto rojo. (c) Ventana de 20x20 píxeles con el centro de masas señalado por el punto rojo. (d) Ventana de 16x16 píxeles con el centro de masas señalado por el punto rojo. (e) Ventana de 10x10 píxeles con el centro de masas señalado por el punto rojo. (f) Ventana de 3x3 píxeles con el centro de masas señalado por el punto rojo.



(a)



(b)

Figura 4.4: (a) Comparación de dos imágenes de Hartmann-Shack tomadas con el mismo sistema. En rojo se muestra la imagen de referencia, mientras que en azul se muestra la imagen del ojo de un sujeto. Se puede apreciar el desplazamiento generado entre los puntos de distinto color. (b) Zona central ampliada en la que se aprecia el desplazamiento de los centroides entre la imagen de referencia (en rojo) y la imagen del ojo de un sujeto (en azul).

4.2.2.2 Etapa de descarte

Otro problema frecuente en las imágenes de sensores Hartmann-Shack son los reflejos corneales. En ocasiones, según el ángulo de incidencia del láser en el ojo, se pueden producir reflejos que incidan directamente sobre el sensor y hagan imposible encontrar el centroide de algunas microlentes correctamente. Si esto no se evita, la aberración calculada es errónea y la medida queda invalidada. Idealmente es un problema que se debe resolver desplazando el ojo o el sistema, pero en ocasiones es imposible solucionarlo completamente o no se percibe hasta que la medida ya se ha completado. Otras situaciones también pueden reducir la calidad de la imagen hasta hacer imposible la detección de los centroides, como es el caso de la rotura de la lágrima que recubre el ojo, la cuál puede reducir muy significativamente el contraste de un punto hasta hacer imposible determinar su centroide con precisión. Por tanto, es necesario desarrollar una solución que lo mitigue. Para tratar de reducir la incidencia de estas situaciones de deterioro en la calidad del resultado se han establecido tres reglas que un centroide debe cumplir para no ser descartado:

1. Se comprueba si el centroide detectado está saturado (podría tratarse de un reflejo) o es muy poco brillante (podría haberse roto la lágrima). Con tal fin se compara el valor del píxel del punto central con la media de sus ocho vecinos. Para ser un centroide correcto el valor debería ser significativamente mayor que el de sus vecinos.
2. Se compara la posición de cada centroide con sus vecinos. Un centroide válido tendrá sus vecinos a una distancia similar a la distancia ideal entre microlentes mientras que uno erróneo estará o muy cerca o muy lejos de sus vecinos.
3. Cualquier centroide que no tenga centroides vecinos en ninguna dirección es descartado, porque es imposible comparar su posición con sus vecinos y porque el tener vecinos erróneos hace más posible que también se trate de un centroide erróneo.

4.2.2.3 Malla de búsqueda dinámica

Por otra parte, el rango de aberraciones que puede presentar un ojo humano es bastante amplio, y de hecho, es bastante común que presente valores muy altos de miopía. Sin embargo, debido a las limitaciones físicas de un sensor de Hartmann-Shack valores muy altos de miopía o hipermetropía pueden ocasionar desplazamientos muy grandes de los

centroides en la imagen de H-S (el desenfoque causa una expansión o concentración de los puntos alrededor del centro). Si dicho desplazamiento es mayor que el tamaño de media microlente, el centroide se moverá al área de otra microlente y generará errores irrecurables en el procesado. Para evitar este problema se ha implementado un método que permite adaptarse a aberraciones grandes, lo hemos denominado **mallá de búsqueda dinámica**. La clave del método es aprovechar que los desplazamientos de los centroides son menores más cerca del centro de la pupila. De forma que inicialmente se procesa una pupila más pequeña que la final y se obtienen los coeficientes de Zernike de la misma. Sin embargo, es necesario reducir la cantidad de coeficientes que se obtienen al procesar la pupila debido a la reducción en la cantidad de centroides obtenidos, ya que no es posible obtener más coeficientes que centroides se procesan si se quieren obtener buenos resultados [77]. Solamente se obtienen los coeficientes relativos al desenfoque y el astigmatismo que son además los que provocan los grandes desplazamientos de los centroides. Una vez obtenidos se redimensionan al tamaño de la pupila deseada para obtener todos los coeficientes de Zernike y a partir de ellos se desplazan los centroides de referencia a las posiciones que ocuparían con dicha aberración. De esta forma se puede realizar un nuevo procesado con la pupila completa y evitar que los centroides se salgan de las áreas destinadas a cada microlente. La figura 4.5 muestra un ejemplo ilustrativo. En ella se compara el método de la mallá de búsqueda dinámica (derecha) frente al mecanismo de búsqueda estático convencional (izquierda). La figura 4.5a muestra como ante una gran aberración algunos centroides se salen del área procesada para su microlente correspondiente. Mientras que la versión dinámica en la figura 4.5b es capaz de anticipar con suficiente precisión dónde se van a encontrar los puntos y analizar las áreas adecuadas para cada microlente.

4.2.2.4 Cálculo de la aberración

Finalmente, una vez se han filtrado los centroides y se ha ejecutado la etapa de la mallá dinámica, si está habilitada, se calculan los coeficientes de Zernike. Para calcular el valor de los coeficientes de Zernike se utilizan las derivadas de los polinomios de Zernike y los valores de los desplazamientos de los centroides de la imagen medida con respecto a la imagen de referencia. Una vez obtenidos los desplazamientos el valor de los coeficientes se puede obtener mediante un ajuste de mínimos cuadrados del sistema de derivadas-desplazamientos o mediante el cálculo de la inversa de las derivadas que es a su vez multiplicado por los desplazamientos. El uso de la inversa suele ofrecer un mejor

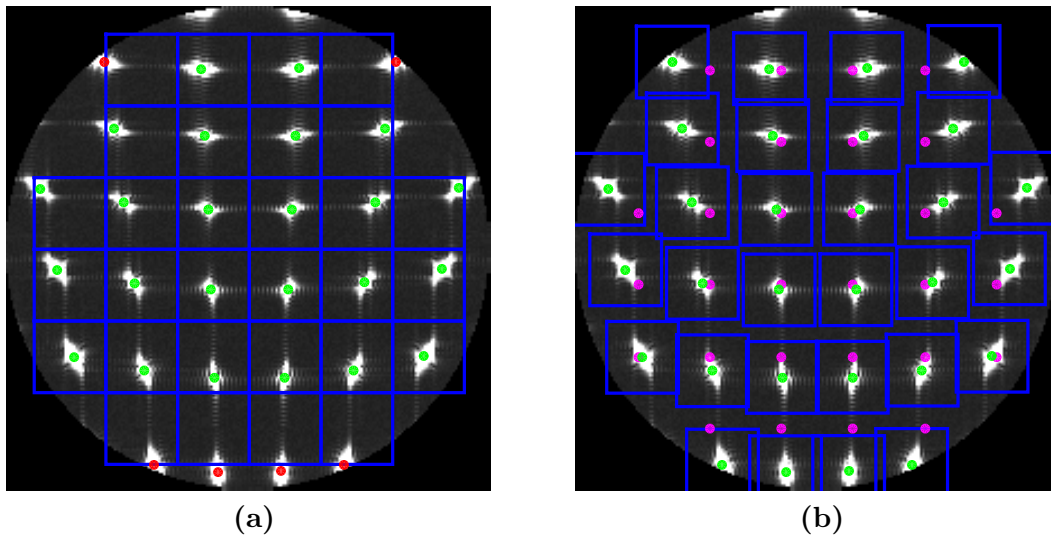


Figura 4.5: (a) Distribución de las microlentes de acuerdo a la distribución de las mismas en la imagen de referencia. Los cuadrados azules delimitan el área de cada microlente, los puntos verdes señalan los centroides que están dentro de las microlentes, y los puntos rojos los que están fuera. (b) Distribución de las áreas desplazadas una vez se ha ejecutado el algoritmo de malla. En azul se delimitan las áreas a analizar en la imagen procesada, los puntos morados indican el centro del centroide en la imagen de referencia y los puntos verdes indican el centroide de los puntos en la imagen procesada.

rendimiento, aunque algunas implementaciones del ajuste de mínimos cuadrados pueden ofrecer una mayor estabilidad del resultado.

Las ecuaciones 4.1, 4.2, y 4.3 muestran la definición de los polinomios de Zernike radiales. Donde los términos pares se definen como:

$$Z_n^m(\rho, \varphi) = R_n^m(\rho) \cos(m\varphi) \quad (4.1)$$

mientras que los impares como:

$$Z_n^{-m}(\rho, \varphi) = R_n^m(\rho) \sin(m\varphi) \quad (4.2)$$

donde n denota el orden del polinomio radial y m es la frecuencia azimutal, cuyo valor depende de n ($m = \pm n, \pm(n-2), \pm(n-4)...$) finalmente los polinomios radiales R_n^m se definen como:

$$R_n^m(\rho) = \sum_{k=0}^{\frac{n-m}{2}} \frac{(-1)^k (n-k)!}{k! (\frac{n+m}{2} - k)! (\frac{n-m}{2} - k)!} \rho^{n-2k} \quad (4.3)$$

4.2.3 Algoritmo Reichenbach

Se ha comparado el rendimiento y la precisión obtenida por el algoritmo PAPYCS con la obtenida por el algoritmo desarrollado por Reichenbach *et al.* [70]. Dicho algoritmo utiliza un método dinámico para encontrar los centroides correspondientes a cada microlente. Un esquema de alto nivel de la aproximación de Reichenbach se muestra en el algoritmo 6, y sus diversas etapas se detallan a continuación:

1. En primer lugar lleva a cabo una búsqueda de candidatos a centroides de microlentes. Para ello examina áreas de 9x9 píxeles buscando su pico y comprobando si éste se encuentra por encima de un umbral. Se examina toda la imagen en busca de posibles candidatos, de forma que si alguna microlente se encuentra saturada se podrían encontrar varios candidatos contiguos. Además, una vez aceptado un píxel como candidato se calcula el centro de masa del área de 9x9 píxeles alrededor del mismo.
2. Partiendo de un conjunto inicial de centroides, se estima la posición de los demás. En nuestro caso, como conjunto inicial hemos tomado los centroides centrales. Seleccionando de la lista de candidatos los que se encuentran más cerca del centro de

la pupila.

3. Después se van seleccionando parejas de centroides *frontera* y, a partir de la distancia entre ellos, se estima dónde estaría el siguiente centroide. Con esa estimación se busca en la lista de candidatos el más cercano que se toma como el correcto.
4. Una vez se han seleccionado todos los centroides se lleva a cabo el cálculo de la aberración del frente de onda mediante un ajuste por mínimos cuadrados de la matriz de derivadas de los polinomios de Zernike igual que se hacía en el algoritmo PAPYCS.
5. Finalmente se computa el mapa del frente de onda. Si bien este paso es opcional en el artículo original [70] el sistema implementaba un espejo deformable que se utilizaba para corregir el frente de onda, por lo cuál este paso era fundamental.

Algoritmo 6 Pseudocódigo del algoritmo de Reichenbach

```

1: procedure REICHENBACH(IMedida, IReferencia, Radio)
2:   Pupila  $\leftarrow$  BuscaPupila(IMedida)
3:   CandidatosRef  $\leftarrow$  BuscaCandidatosCentroides(IReferencia, Pupila, Radio)
4:   CandidatosMed  $\leftarrow$  BuscaCandidatosCentroides(IMedida, Pupila, Radio)
5:   [CentroidesRef, CentroidesMed]  $\leftarrow$  BuscaCentroidesIniciales(
       CandidatosMed, CandidatosRef, Pupila)
6:   NumIteraciones  $\leftarrow$  Radio/TamMicrolente
7:   for i  $\leftarrow$  0, NumIteraciones do
8:     [CentroidesRef, CentroidesMed]  $\leftarrow$  Correlaciona(CentroidesRef,
       CentroidesMed, CandidatosRef, CandidatosMed, Pupila, Radio)
9:   end for
10:  Zernikes  $\leftarrow$  CalculaCoeeficientes(CentroidesRef, CentroidesMed)
11:  MapaFrenteDeOnda  $\leftarrow$  CalculaMapa(Zernikes) ▷ Opcional
12:  return [Zernikes, MapaFrenteDeOnda]
13: end procedure

```

4.3 Paralelización y optimización

Se han desarrollado dos implementaciones para el algoritmo PAPYCS y el algoritmo Reichenbach, una secuencial en C++ y otra paralela utilizando la GPU en CUDA. La implementación en C++ hace uso, además, de la librería de código abierto OpenCV [78]

para las funciones de procesamiento de imagen (a saber: filtro de Gauss, umbralización, cierre morfológico), utilizados en el seguimiento pupilar. Se ha optado por la librería abierta OpenCV por estar altamente optimizada, hasta el punto de utilizar instrucciones vectoriales si están disponibles en el procesador. También se ha empleado la librería de código abierto Eigen [79] para llevar a cabo el ajuste final de los polinomios de Zernike.

4.3.1 Implementación paralela mediante CUDA

4.3.1.1 Seguimiento pupilar

La etapa de seguimiento pupilar consta de varias etapas, las iniciales implican un pre-procesado de la imagen para eliminar ruidos y facilitar su posterior análisis en busca de la pupila:

1. En primer lugar se aplica un filtro de Gauss que se paraleliza de una forma similar a la presentada en el capítulo 2.
2. En segundo lugar se lleva a cabo una umbralización, para la cuál se calcula el valor del umbral a aplicar dinámicamente. Utilizando para ello el algoritmo de Otsu [76], que analiza la distribución de los valores del histograma de la imagen para determinar el umbral adecuado automáticamente. Mientras que la etapa de análisis del histograma es bastante liviana, el cálculo del histograma requiere bastante más cómputo. La paralelización del histograma se ha llevado a cabo en dos pasos. Primero se realiza una reducción a nivel de bloque utilizando operaciones atómicas en memoria compartida en la que cada bloque de hilos de CUDA genera un histograma de salida. En segundo lugar se lleva a cabo una reducción de los histogramas generados en el primer paso, de forma que se lanzan 256 hilos y cada uno lleva a cabo la reducción de una posición del histograma. Una vez calculado el histograma se ejecuta el análisis del histograma con el método de Otsu. Para ello se lanzan 256 hilos CUDA, en primer lugar calculan en paralelo el resultado del producto escalar entre el histograma y un vector $[0:255]$. A continuación, cada uno calcula el valor de la métrica final para cada una de las posiciones del histograma y mediante una operación atómica en memoria compartida se obtiene el valor máximo de la misma y su umbral correspondiente.

Una vez obtenido el umbral el proceso de umbralización se ha paralelizado lanzando un hilo por cada píxel de la imagen que se encarga de cargar el valor de entrada,

compararlo con el umbral y escribir el resultado para el mismo.

3. A continuación, se lleva a cabo una operación morfológica de cierre para como se ha descrito anteriormente tratar de minimizar los huecos dentro de la pupila sin umbralizar. Para llevar a cabo esta operación se requiere el cómputo de una operación de dilatación y otra operación de erosión, las cuáles también se han utilizado en el capítulo 2 y cuya optimización se ha descrito extensamente en el mismo.
4. Para terminar, queda determinar el borde de la pupila. Para ello se comprueban todos los píxeles de la imagen y, en caso de que no se trate de un píxel del fondo, se comprueban sus 8 vecinos. Así, si algunos de sus vecinos es un píxel del fondo se marca dicho píxel como un píxel del borde de la pupila. La paralelización de esta etapa se ha llevado a cabo lanzando un hilo CUDA por cada píxel de la imagen que se encarga de realizar las comprobaciones. En caso de que se trate de un píxel del borde de la pupila se incrementa un contador de píxeles del borde mediante una operación atómica y se almacena la posición del píxel en la lista de píxeles del borde.
5. Finalmente, se procesan los puntos encontrados con el algoritmo RANSAC, la paralelización del mismo se ha explicado detalladamente en el capítulo 2.

Algunas de las optimizaciones aplicadas a las operaciones paralelizadas descritas hasta ahora, tales como las operaciones de erosión y dilatación, o el filtro de Gauss, y el algoritmo RANSAC se han descrito en detalle en el capítulo 2. Basta, por tanto, mostrar aquí una enumeración de las mismas: kernels separables, memoria compartida, desenrollado de bucles automático utilizando plantillas de C++, optimización del ancho de banda de la memoria, *fast math*, precálculo de números aleatorios, optimización de las comunicaciones dispositivo-anfitrión.

4.3.1.2 Paralelización y optimizaciones en el algoritmo PaPyCS

Se han realizado tres implementaciones de la búsqueda de los centroides de los puntos, como se ha comentado con anterioridad: PAPYCS, centro de masa simple, y propuesta de Reichenbach. En primer lugar se describe la paralelización de *PaPyCS*, este algoritmo divide la imagen en tantas subsecciones como microlentes haya en el H-S, siendo éste el primer nivel de paralelismo, puesto que cada microlente se procesa de forma independiente.

Además, el procesamiento de cada microlente también se ejecuta en paralelo. En concreto se lanzan 32 hilos por cada microlente, que se corresponde con el tamaño de un *warp*. En el algoritmo 7 se muestra la paralelización a alto nivel de PAPYCS. Se pueden procesar una cantidad variable de puntos por cada bloque de hilos CUDA, mientras que siempre se lanza 32 hilos por cada punto. La cantidad total de hilos y bloques en ejecución dependerá de la cantidad de microlentes a procesar, asegurando una buena escalabilidad para diferentes sistemas con diferentes cantidades de microlentes. Por otra parte, el algoritmo 8 muestra la paralelización del procesamiento de una microlente. En este caso se muestra como los hilos de cada *warp* colaboran para calcular el centro de masa en cada iteración, al final de la cuál comparten sus resultados para obtener el centro de masa. Este algoritmo cuenta con dos bucles, el bucle superior ejecuta una cantidad fija de iteraciones, esto es, tantas veces como se establezca en la configuración. Normalmente el número de iteraciones dependerá del tamaño de la microlente en el sensor de la cámara, a mayor tamaño más iteraciones y viceversa. El bucle interior se ejecuta una cantidad variable de veces, con cada iteración del bucle superior la cantidad de iteraciones del bucle interior se reduce en una unidad. Como se muestra en el algoritmo 8 el acceso a los datos se ejecuta en tres fases. Esto se debe a que la fase central utiliza operaciones de carga de datos vectoriales y es necesario asegurar el alineamiento de dichas operaciones en la primera y tercera fase del procesado. También se puede observar que el nivel de paralelismo disminuye con cada iteración al disminuir la cantidad de trabajo a realizar, de forma que cada vez más hilos se quedan ociosos, pero también hay menos trabajo a realizar en cada iteración.

El algoritmo de centro de masa simple utiliza el mismo código que PAPYCS con la única diferencia que solo ejecuta la primera iteración, frente a las n iteraciones que realiza PAPYCS.

Algoritmo 7 PAPYCS pseudo-código de alto nivel de la búsqueda de centroides.

Imagen: imagen de Hartmann-Shack

Centroides: Vector con las coordenadas de cada centroide

NumCentroides: Número de centroides

NumPasos: Número de pasos para la búsqueda

$block \leftarrow (32 * CENTROIDES_POR_BLOQUE, 1, 1)$

$grid \leftarrow (NumCentroides / CENTROIDES_POR_BLOQUE, 1, 1)$

$BuscaCentroides <<< grid, block >>> (Imagen, Centroides, NumCentroides, NumPasos)$

A continuación se describen dos optimizaciones específicas aplicadas al algoritmo 8.

Algoritmo 8 Kernel del algoritmo PAPYCS.

```

1: Imagen: imagen de Hartmann-Shack
2: Centroides: Vector con las coordenadas de cada punto
3: NumCentroides: Número de puntos
4: NumPasos: Número de pasos para la búsqueda
5:  $CentroActual \leftarrow CalcularCentro(Centroides, ThreadId)$ 
6: for  $i \leftarrow 0, NumPasos - 1$  do
7:    $TamPiramide \leftarrow TamLente - i$ 
8:    $CentroParcial \leftarrow (0, 0)$ 
9:    $Minimo \leftarrow inf$ 
10:   $y \leftarrow CentroActual.y - \frac{TamPiramide}{2} + ThreadIdx$ 
11:  while  $y < CentroActual.y + \frac{TamPiramide}{2}$  do
12:     $[CentroParcial, Mnimo] += InicioCentroDeMasaParcialSinAlinear(Imagen,$ 
     $CentroActual, y, TamPiramide)$ 
13:     $[CentroParcial, Mnimo] += CentroDeMasaParcialVectorizado(Imagen, Cen-$ 
     $troActual, y, TamPiramide)$ 
14:     $[CentroParcial, Mnimo] += FinCentroDeMasaParcialSinAlinear(Imagen,$ 
     $CentroActual, y, TamPiramide)$ 
15:     $y \leftarrow y + 32$ 
16:  end while
17:   $CentroActual \leftarrow ShuffleCentroDeMasa(CentroParcial, Mnimo)$ 
18: end for
19:  $GuardaCentro(Centroides, CentroActual)$ 

```

4.3.1.2.1 Instrucciones `shuffle`:

Las instrucciones `shuffle` permiten una comunicación muy rápida entre hilos del mismo *warp*, así que son perfectas para optimizar el procesamiento de las microlentes, donde cada microlente es procesada por un solo *warp*. Se emplean para compartir el resultado parcial del centro de masa, se comparten cuatro parámetros: la suma de la coordenada x multiplicada por la intensidad de cada píxel, la suma de la coordenada y multiplicada por la intensidad de cada píxel, la suma total de las intensidades de los píxeles y la intensidad mínima. Finalmente, el hilo 0 comunica el resultado global al resto. Además, para mejorar el rendimiento las instrucciones `shuffle` se han entrelazado. La reducción mediante instrucciones `shuffle` se lleva a cabo en un bucle con dependencias entre las iteraciones. Cuando se llevan a cabo varias reducciones de forma consecutiva, como en este caso con las posiciones x e y , la suma de intensidad y la intensidad mínima, se pueden fusionar los bucles de las distintas reducciones y aumentar el paralelismo a nivel de instrucción del código. Evitando o reduciendo las detenciones por dependencias de datos.

4.3.1.2.2 Accesos a memoria vectorizados:

Los accesos a memoria vectorizados son muy útiles en diversas situaciones pero más aún cuando se trabaja con tipos de datos de tamaño muy reducido como es el caso de las imágenes en las que cada píxel tan sólo ocupa 8 bits. Se ha dividido el procesamiento de cada fila de píxeles de la microlente en tres etapas para en la etapa central asegurar que los píxeles están alineados en memoria y se puede acceder a ellos con operaciones vectoriales. Se ha optado por cargar 4 píxeles en cada operación como un buen compromiso entre aumentar la tasa de datos utilizados de cada carga de memoria y evitar que las etapas de alineamiento supongan una gran parte del procesado.

4.3.1.3 Malla de búsqueda dinámica

Para paralelizar el mecanismo de malla de búsqueda dinámica descrito anteriormente se ha hecho uso de la librería CUBLAS, que implementa operaciones de álgebra lineal. Se computa una matriz con las derivadas de los polinomios de Zernike para el tamaño de pupila deseado. El cómputo de la llamada matriz B requiere de una gran cantidad de cómputo, incluyendo el cálculo de números factoriales, exponenciales y numerosos productos y divisiones de números usando `floats`. Es por ello que se ha llevado a cabo

una profunda paralelización de este cálculo, empleando $1024 * numCoeficientes$ hilos CUDA para el cálculo de la matriz, de forma que el aumento del número de coeficientes utilizados no lastre el rendimiento. Cada bloque de 1024 hilos se encarga de calcular los valores de la matriz correspondientes a un coeficiente, realizando gran parte del trabajo de forma independiente, solamente comparten sus resultados utilizando un array de memoria compartida. Algunas optimizaciones aplicadas a esta implementación son:

4.3.1.3.1 Precálculo de los factoriales

El rango de valores de números factoriales que utiliza el algoritmo viene dado por el orden de polinomios de Zernike que se esté utilizando para resolver el sistema de ecuaciones. Por lo que es posible realizar el cálculo de los mismos al inicio de la ejecución, almacenar el resultado y posteriormente cargarlos de memoria. Evitando así un gran número de operaciones para factoriales grandes.

4.3.1.3.2 Memoria compartida

Si bien el precálculo de los factoriales es una mejora respecto a su cálculo constante el acceso a sus valores en memoria global sigue suponiendo un gran coste en tiempo de ejecución, especialmente si se tiene en cuenta que se usan de forma muy intensiva. Así que se ha optado por cargar el array de los valores factoriales en un array de memoria compartida. Esta operación se repite una vez por cada bloque pero permite reducir los accesos a memoria global y las latencias de dichos accesos.

Finalmente, la matriz B se multiplica por otra que contiene los valores de los coeficientes de Zernike escalados al mismo tamaño de pupila. Con lo que se obtienen los desplazamientos que cada punto dentro de cada microlente para la aberración utilizada. Finalmente, basta con sumar los desplazamientos a la posición de cada centroide en la imagen de referencia para obtener la nueva malla de búsqueda.

4.3.1.4 Paralelización y optimizaciones en el algoritmo de Reichenbach

El algoritmo desarrollado por Reichenbach *et al.* [70] implica un análisis inicial de la imagen muy minucioso, sin embargo, es posible paralelizar dicho análisis eficientemente. En primer lugar, el algoritmo analiza todos los píxeles de la imagen comparándolos con sus vecinos, buscando picos que almacena como potenciales centroides de puntos. Esta etapa se ha paralelizado empleando un hilo por cada píxel de la imagen. Dicho hilo tiene que realizar

bastante trabajo ya que tiene que analizar los vecinos de cada píxel para determinar si se trata de un posible punto y además calcular su centro de masas en tal caso. Además, para evitar accesos repetidos a memoria, cada hilo realiza a la vez la comprobación de si se trata de un pico y el cálculo del centro de masa de dicho píxel. Ello permite evitar cargas de datos repetidas, aunque genera cálculos que serían prescindibles si el píxel resulta no ser un pico. Además, se ha implementado una optimización especialmente enfocada a los accesos a memoria.

4.3.1.4.1 Memoria compartida

Se ha utilizado la técnica conocida como *tiling* que consiste en procesar los datos por bloques que caben en la memoria compartida y que son accedidos de forma repetida. Durante el análisis inicial de la imagen en busca de candidatos a puntos se generan muchos accesos a las mismas posiciones de memoria desde hilos contiguos, de forma que el uso de *tiling* con memoria compartida permite reducir latencias y aumentar la velocidad de acceso a memoria en esta operación.

Una vez generada la lista de candidatos a puntos se genera una malla inicial de 5x5 puntos en el centro de la pupila, y se seleccionan los candidatos más próximos a las localizaciones ideales. Se lanzan 32 hilos por cada punto en la malla inicial para buscar el más cercano de los candidatos. Una vez completada la malla inicial comienza una búsqueda iterativa en dos pasos del resto de puntos. Primero se comprueba qué posiciones es posible interpolar a partir de la información disponible. Para ello se lanza un hilo por cada microlente del sistema. Cada uno de los hilos comprueba si todavía no ha sido asignado, en cuyo caso comprueba si es posible estimar su posición a partir de sus vecinos. En caso afirmativo estima su posición y la añade a una lista de búsqueda. En una segunda etapa se mide la distancia entre cada posición estimada y cada candidato a centroide y se selecciona el más cercano, siempre y cuando se encuentre alguno dentro del rango máximo. Este proceso se repite mientras el área abarcada por los puntos sea menor que el tamaño de la pupila procesada.

4.3.1.5 Etapa de descarte

La etapa de descarte consta de tres criterios como se ha descrito anteriormente. El primero que se aplica es el de saturación o poco brillo. Se lanza un hilo por cada microlente, y éste calcula la media de sus ocho píxeles vecinos. Si la diferencia entre el píxel central y

sus vecinos es menor de 15^1 se descarta, si no se mantiene. El segundo criterio aplicado es la distancia con sus vecinos, para comprobar este es necesario disponer los puntos en una estructura 2D que permite llevar a cabo la comparación. E igual que con el primer criterio, si los puntos son aceptados se mantienen y si no se descartan. Finalmente, se comprueba si algún punto ha quedado totalmente aislado, en cuyo caso también se descarta.

4.3.1.6 Cálculo de la aberración

Para el cálculo de la aberración se ha utilizado la librería CUBLAS. Durante las pruebas previas de rendimiento y precisión se han comparado la función `gels`, que ejecuta un ajuste por mínimos cuadrados, y la multiplicación de la inversa de la matriz que contiene las derivadas de los polinomios de Zernike por los desplazamientos de los puntos. Si bien el ajuste por mínimos cuadrados ofrece una mayor precisión que el uso de la inversa, su rendimiento es significativamente menor. Así que, para obtener los resultados de rendimiento y precisión se ha utilizado el método de la inversa.

4.3.1.7 Mapa de frente de onda

El cálculo del mapa de frente de onda es una tarea que requiere un gran cantidad de cómputo, que además se incrementa de forma lineal con el número de coeficientes de Zernike utilizados y de forma cuadrática con el tamaño del mapa. Para su paralelización hemos optado por lanzar un hilo por cada píxel de salida, con lo cuál ese hilo tendrá más trabajo cuanto mayor sea el número de coeficientes de Zernike utilizados. En el ámbito de este estudio eso es razonable ya que la cantidad de coeficientes nunca será muy elevada, pero si se diera el caso sería necesario paralelizar también esa dimensión. Lo cuál se podría lograr fácilmente utilizando operaciones atómicas para combinar los resultados parciales de cada hilo. La figura 4.6 muestra un esquema del cómputo del mapa de frente de onda final. A la izquierda se muestran los mapas correspondientes a cada coeficiente de Zernike, en el centro el vector de coeficientes que representa a la aberración que se usa para pesarlos. El resultado de cada multiplicación se acumula y se obtiene la imagen final, que se muestra a la derecha.

El cálculo del mapa de frente de onda correspondiente a cada coeficiente de Zernike se precalcula al inicio de la ejecución y se reutiliza después, por lo tanto su tiempo de

¹Es un valor obtenido mediante experimentación que puede variar para diferentes de configuraciones de sensores de Hartmann-Shack

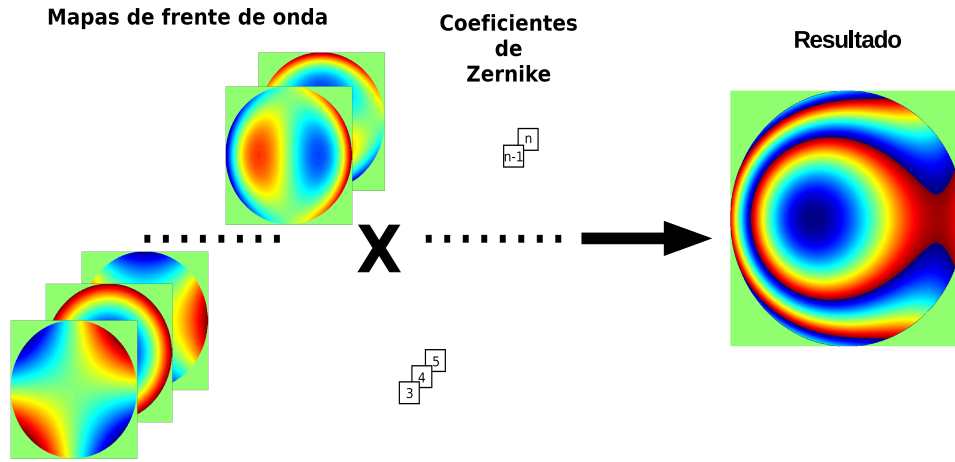


Figura 4.6: Esquema que muestra cómo se lleva a cabo el cómputo del mapa del frente de onda de la aberración (mapa de la derecha) a partir de los mapas de cada aberración concreta (mapas de la izquierda) y los coeficientes de Zernike (en el centro).

ejecución se ha omitido de los resultados.

La implementación *naïve* del cálculo del mapa de frente de onda ofrece un rendimiento muy bueno. Tanto es así que si analizamos el kernel que concentra casi todo el trabajo necesario para el cálculo del mapa de frente de onda vemos que la paralelización conseguida es muy buena. La tabla 4.1 muestra una comparación entre la implementación *naïve* de este kernel y el mismo con las distintas optimizaciones aplicadas. Se ha utilizado la configuración con una mayor carga de trabajo de las utilizadas para obtener esos resultados (imágenes de 2000x2000 píxeles con 100 coeficientes de Zernike). En lo que a la implementación *naïve* se refiere, se puede ver cómo se ha conseguido un ancho de banda con la memoria global bastante alto (261 GB/s), aunque el IPC conseguido no es muy bueno. La proporción de cómputo/accesos a memoria es muy baja y eso lastra el IPC.

4.3.1.7.1 Memoria compartida

Los valores de los coeficientes de Zernike son accedidos por todos los hilos del kernel, por lo tanto hay una gran cantidad de accesos duplicados a ellos. Es posible optimizar dichos accesos con el uso de memoria compartida, descongestionando además la memoria global muy necesaria en este kernel para cargar los datos de los mapas de frente de onda. Cada bloque de hilos en CUDA carga una copia de los coeficientes de Zernike a su propia

	BW Memoria Global (GB/s)	Datos transferidos (MiB)	Tiempo de ejecución (ms)	Detención por dependencia de memoria(%)	IPC
<i>Naïve</i>	261.13	1553.77	6.45	32.67	0.305
Memoria compartida	326.80	1549.44	4.72	30.23	0.390
Half	233.46	786.28	3.41	41.93	0.610
Desenrollado de bucles	280.60	783.93	2.80	69.28	0.589
Pitched memory	332.87	783.99	2.35	66.65	0.700

Tabla 4.1: Tabla que muestra algunos resultados de rendimiento de varias implementaciones de un kernel utilizado durante el cálculo de los mapas de frente de onda en la GPU.

zona de memoria compartida, y posteriormente, accede a dicha memoria para obtener los valores de las coeficientes. Como se puede ver en la tabla 4.1 al aplicar esta optimización se consigue aumentar el ancho de banda de memoria en los accesos a memoria global por encima de los 300 GB/s. Así como incrementar el IPC y reducir el tiempo de ejecución significativamente.

4.3.1.7.2 Pinned memory

Esta optimización se ha utilizado para incrementar el ancho de banda en la copia de la imagen resultado de la GPU al anfitrión. Al igual que ya se explicó en el caso del seguimiento pupilar 2.3.2, la *pinned memory* permite utilizar páginas de memoria bloqueadas en la CPU y programar copias entre el dispositivo y el anfitrión sin utilizar áreas temporales de memoria reservadas por el driver. Debido al alto grado de optimización conseguido, el tiempo de ejecución del cómputo es muy reducido y el peso de la copia del resultado es significativo, por lo que esta optimización es una gran herramienta para reducir dicho peso.

4.3.1.7.3 Tipo de datos *half*

El cálculo del mapa de frente de onda no requiere una gran precisión. Ya que cuando se utiliza para visualización no hay diferencias apreciables entre distintas precisiones y cuando se utiliza para corregir los elementos correctores suelen disponer de un rango limitado de posibles valores. Por lo tanto, es posible utilizar el tipo de datos *half* para

almacenar el mapa de frente de onda de cada coeficiente en lugar de `float`. Aún así, las operaciones se han llevado a cabo utilizando `floats` por los escasos recursos hardware de los que dispone el procesador gráfico para llevar a cabo las operaciones con `half`. Un procesador gráfico con soporte completo para operaciones con tipos de datos `half` debería ser capaz de procesar el doble de operaciones que utilizando `float`, y por tanto, podría potencialmente ofrecer una mejora de rendimiento significativa. El utilizar el tipo de datos `half` ha permitido reducir significativamente el tamaño de los datos que tiene que acceder el kernel, como se muestra en la tabla 4.1 la cantidad de datos accedidos desde memoria global desciende desde más de 1500MB hasta casi la mitad con unos 780MB. Asimismo, el tiempo de ejecución también se ve reducido y el IPC aumenta. Si bien el ancho de banda de los accesos a memoria global cae, se ve compensado por la menor cantidad de datos accedidos.

4.3.1.7.4 Desenrollado de bucles

Otra optimización se centra en el desenrollado del bucle que se encarga de multiplicar cada frente de onda de cada coeficiente de Zernike por el valor de dicho coeficiente en la aberración introducida. Este bucle en nuestras pruebas ejecuta entre 10 y 100 iteraciones, y en cada una de ellas se carga un dato de memoria global y otro de memoria compartida que posteriormente se multiplican. El desenrollado permite prescindir de muchas operaciones innecesarias y optimizar las operaciones y las cargas de datos. La tabla 4.1 muestra que tras esta optimización se consigue reducir el tiempo de ejecución significativamente y aumentar el ancho de banda de los accesos a memoria. Sin embargo, el IPC disminuye ligeramente. Ésto se debe a que al desenrollar el bucle el número de instrucciones ejecutadas por el kernel disminuye en un 20%. Por lo tanto, es posible disminuir el tiempo de ejecución manteniendo el mismo IPC. Además, debido a que al desenrollar el bucle es posible evitar las dependencias entre las operaciones de cómputo se incrementa el porcentaje de detenciones por dependencias de datos.

4.3.1.7.5 Pitched memory

La *pitched memory* permite alinear los datos en memoria para optimizar los accesos a los mismos. Se ha utilizado para almacenar el mapa del frente de onda de cada coeficiente de Zernike, de forma que el comienzo de cada línea de las imágenes está alineado. Debido a que el cómputo de los mapas de frente de onda presenta una gran cantidad de accesos a memoria global esta optimización ofrece unos resultados muy positivos. En la tabla 4.1

se muestra que se consigue mejorar el ancho de banda de los accesos a memoria global, mejorando el tiempo de ejecución y el IPC.

4.4 Resultados

4.4.1 Metodología de evaluación

Medir la precisión del procesado en imágenes de H-S reales es complejo debido a la imposibilidad de determinar con suficiente precisión el valor real de lo que se está midiendo. Incluso si se mide una lente que se sabe que es de 1 dioptría de desenfoque y se asume que está perfectamente alineada se sigue cometiendo un error al obviar que ninguna lente es perfecta y siempre contiene alguna aberración, además de la dificultad de alinearla perfectamente. Así que, para medir correctamente la precisión y el rendimiento de los algoritmos implementados, se ha optado por generar unos conjuntos de imágenes artificiales en las que las aberraciones son conocidas. Sin embargo, las imágenes artificiales no incluyen muchos elementos que sí incluyen las imágenes reales, así que se han añadido algunos elementos a las mismas para obtener unas imágenes más cercanas a la realidad. Se han añadido reflejos que simulan los reflejos corneales, con “mapas de brillo” obtenidos de imágenes de H-S reales, se han simulado los cambios de brillo que por múltiples razones se observan en imágenes reales (rotura de lágrima, iluminación no uniforme, inclinación no óptima,...). También se ha añadido el reflejo de luz de la retina que abarca toda la pupila en las imágenes de Hartmann-Shack de un ojo humano. Finalmente, se ha añadido ruido blanco a toda la imagen. Además, para generar unas imágenes mas realistas en las aberraciones simuladas se ha ponderado cada coeficiente de Zernike con el peso que tiene en una población normal de sujetos, como los medidos por Castejón-Mochón *et al.* [80].

Para evaluar la precisión se han generado imágenes con diferentes aberraciones, abarcando desde imágenes que presentan ligeras aberraciones (con un valor de 0.5 RMS, Root Mean Square) hasta imágenes que presentan grandes aberraciones (con un valor de 6.0 RMS).

Para evaluar el rendimiento se han generado imágenes variando dos parámetros, el tamaño de la imagen y la cantidad de microlentes del sensor H-S. De forma que se ha generado un conjunto en el que las imágenes tienen 250, 500, 1100 ó 2000 microlentes con un tamaño constante de 2560x2560 píxeles. El área que ocupa cada microlente varía según se va aumentando el número de microlentes, de forma que el área total ocupada

por las microlentes siempre es la misma. Se ha generado otro conjunto de imágenes en el que se varía el tamaño de las mismas, a saber: 640x460, 1280x1280 y 2560x2560 píxeles, fijándose el número de microlentes en 250. El porcentaje de la imagen ocupado por las microlentes se mantiene constante, por lo que el tamaño total de cada microlente va en aumento con la resolución de la imagen.

En último lugar, para intentar evaluar la precisión en un sistema real, se ha tomado una serie de imágenes de un ojo artificial con un sensor H-S manteniendo todos los elementos del mismo constantes. Esta serie de imágenes permite evaluar la estabilidad del algoritmo ante imágenes muy similares en las que solo varía el ruido de fondo.

El rendimiento se ha medido utilizando un procesador gráfico NVIDIA GTX 1080 Ti que cuenta con 3584 núcleos CUDA, se ha compilado utilizando CUDA 10.1 y se ha utilizado la librería cuBLAS para el ajuste de los polinomios de Zernike. La versión secuencial implementada en C++ se ha ejecutado en un procesador Intel i5-4690 (turbo hasta 3.9Ghz) que cuenta con 4 núcleos físicos y 4 hilos, utilizando el compilador gcc V7. La librería OpenCV (v3.3.1) [78] se ha utilizado en la versión secuencial para las operaciones de procesamiento de imágenes (filtro de Gauss, umbralización y cierre morfológico). Además, la versión secuencial utiliza la librería Eigen [79] para realizar el ajuste de los polinomios de Zernike.

4.4.2 Rendimiento de los algoritmos paralelos

Las imágenes de Hartmann-Shack tienen dos parámetros fundamentales que afectan al rendimiento de forma significativa, su tamaño y el número de microlentes del sensor. Por ello se han analizado estos dos parámetros de forma independiente. La figura 4.7 muestra la aceleración obtenida por la versión GPU sobre la versión secuencial para cada uno de los tres algoritmos. Además, se desglosan por separado las aceleraciones de la parte de búsqueda de los centroides y la aceleración total. Para esta medida se ha utilizado una imagen de 2560x2560 píxeles y la cantidad de spots se ha variado de 250 a 2000. Se puede ver cómo la aceleración aumenta al aumentar el número de microlentes que añaden más trabajo y permiten un paralelismo mayor en la GPU mientras que la versión secuencial aumenta su tiempo de ejecución en mayor medida. Además, se puede observar cómo el algoritmo PAPYCS ha conseguido una aceleración significativamente mayor que la de los otros algoritmos. Ello se debe a la mayor facilidad que presenta el algoritmo para su paralelización y a la mayor cantidad de cómputo que conlleva el mismo.

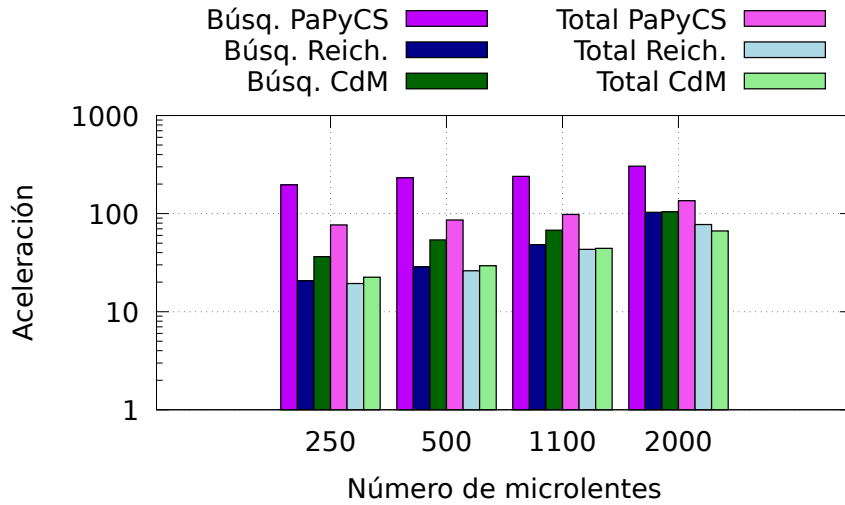


Figura 4.7: Aceleración de la implementación paralela sobre la secuencial de la etapa de búsqueda de los centroides y del total del proceso en función del número de puntos procesados (tamaño de la imagen fijo a 2560x2560 píxeles).

Por otra parte, la figura 4.8 también muestra la aceleración de la versión GPU con respecto a la versión secuencial para cada algoritmo pero esta vez se compara la aceleración con respecto a un tamaño de imagen variable, mientras que la cantidad de microlentes se mantiene fija. En este caso se observa que la etapa de búsqueda de centroides mejora su aceleración al pasar de imágenes de 640x640 píxeles a imágenes de 1280x1280 píxeles y vuelve a mejorar al pasar a imágenes de 2560x2560 píxeles. A pesar de que la aceleración aumenta en la etapa de búsqueda de centroides no lo hace para todos los algoritmos en lo que al tiempo total se refiere. Para aumentar el rendimiento de la etapa de búsqueda en imágenes con microlentes muy grandes se podría incrementar la cantidad de hilos que se lanzan para procesar cada microlente, sin embargo, una configuración con tantos píxeles por microlente es muy poco común en los sistemas ópticos. A pesar de ello, la aceleración conseguida en la etapa de búsqueda de centroides para el algoritmo PAPYCS es superior a 100x. Mientras que el algoritmo de Reichenbach consigue una aceleración de 21x y el algoritmo de Centro de Masas obtiene una aceleración de 39x también para la etapa de búsqueda de centroides.

En la figura 4.9 se muestra el tiempo empleado por cada una de las etapas del procesamiento para los tres algoritmos implementados en función del número de microlentes de la imagen, manteniendo el tamaño de la misma fijo en 2560x2560 píxeles. Las partes del procesamiento

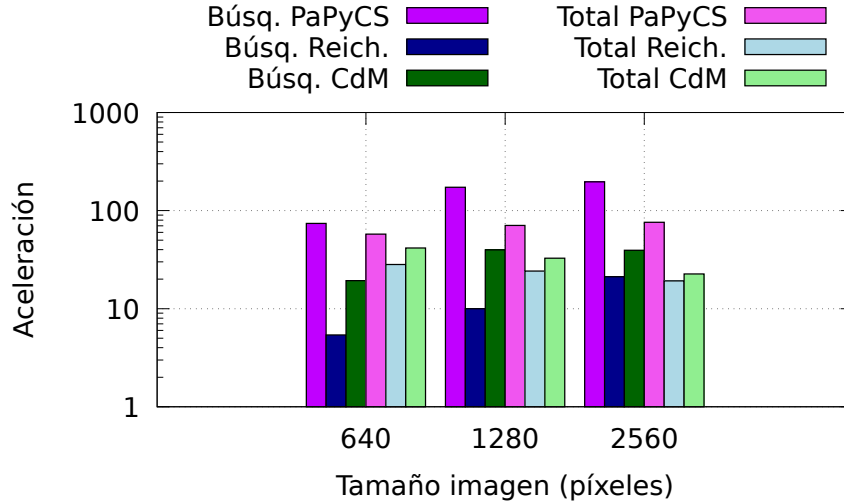


Figura 4.8: Aceleración de la implementación paralela sobre la secuencial de la etapa de búsqueda de los centroides y del total del proceso en función del tamaño de la imagen (número de microlentes fijo a 250).

compartidas por los tres algoritmos tienen el mismo tiempo de ejecución, mientras que la etapa de búsqueda varía. La única etapa común que ve su tiempo de ejecución afectado por la cantidad de microlentes es el ajuste de los coeficientes de Zernike. Cuanto mayor es su número mayor es el tamaño de la matriz de derivadas de los polinomios de Zernike y mayor tiempo se tarda en procesarlas.

En cuanto a la etapa de búsqueda, tanto el algoritmo PAPYCS como el del centro de masa muestran muy buena escalabilidad con el número de microlentes debido a que cuánto mayor sea el número de lentes más hilos se lanzan y el trabajo a realizar permanece bastante constante debido a que el área procesada es de igual tamaño. No ocurre lo mismo con el algoritmo de Reichenbach que, debido a su ejecución iterativa, un mayor número de lentes implica más iteraciones, y por tanto, un mayor tiempo de ejecución. Así, el tiempo de búsqueda pasa de 0.86ms para 250 microlentes a 2.54ms para 2000 microlentes para el algoritmo de Reichenbach. Mientras que para el algoritmo PAPYCS pasa de 1.76ms a 1.67ms para 250 y 2000 microlentes respectivamente.

La figura 4.10 muestra el tiempo que tarda la CPU en ejecutar cada una de las etapas del procesamiento para los tres algoritmos y cuatro configuraciones de microlentes distintas. Si bien el reparto del cómputo entre las distintas etapas varía significativamente de una configuración, a otra en las más densas (esto es, con un mayor número de microlentes)

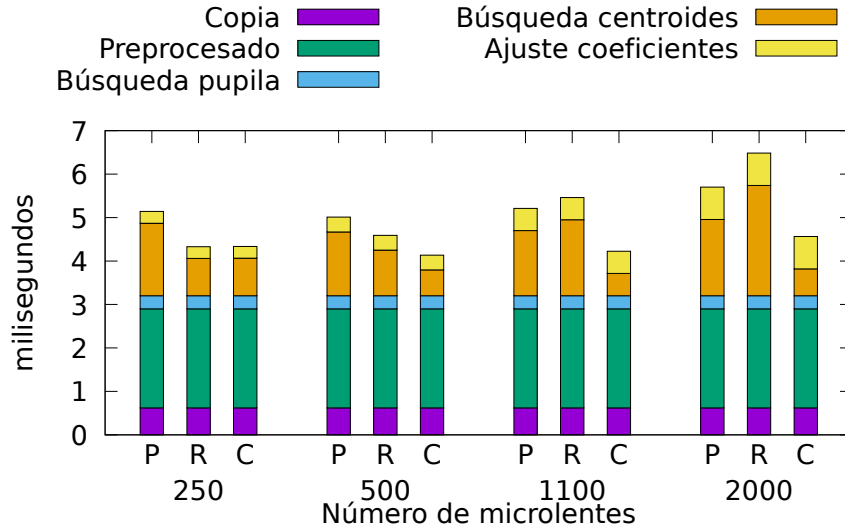


Figura 4.9: Tiempo de ejecución en la GPU desglosado por etapas en función del número de puntos procesados (tamaño de la imagen fijo a 2560x2560 píxeles). *P* para *PaPyCS*, *R* para Reichenbach y *C* para Centro de Masa

los tres algoritmos emplean la mayor parte del tiempo en la detección de los centroides y el cálculo de la aberración. Asimismo, el algoritmo PAPYCS conlleva la mayor carga de cómputo y como consecuencia el mayor tiempo de ejecución.

De igual forma la figura 4.11 muestra el tiempo empleado por cada etapa del procesamiento para los tres algoritmos cuando se varía el tamaño de la imagen, manteniendo el número de microlentes en 250. En este caso se ven afectadas todas las etapas del procesamiento excepto la que realiza el ajuste de los coeficientes de Zernike. La fase de copia aumenta por el aumento evidente de datos al copiar imágenes más grandes, mientras que las fases de preprocesado y la búsqueda de pupila incrementan sus tiempos de ejecución por el mayor trabajo que deben realizar. Finalmente, la etapa de búsqueda escala muy bien en el caso del algoritmo de Reichenbach debido a que solamente tiene más trabajo en su primera etapa de búsqueda de candidatos. La cuál, además, se beneficia del uso de la memoria compartida para mitigar el incremento de los datos procesados. Su tiempo de ejecución para etapa de búsqueda pasa de 0.56ms a 0.84ms para imágenes de 640x640 píxeles y 2560x2560 píxeles respectivamente. Mientras que el algoritmo PAPYCS y el centro de masa sí ven su tiempo de ejecución subir significativamente con el tamaño de la imagen debido a que no se incrementa el paralelismo al aumentar el tamaño y sí que aumenta el trabajo a realizar. El tiempo de la etapa de búsqueda pasa de 0.31ms para imágenes de 640x640

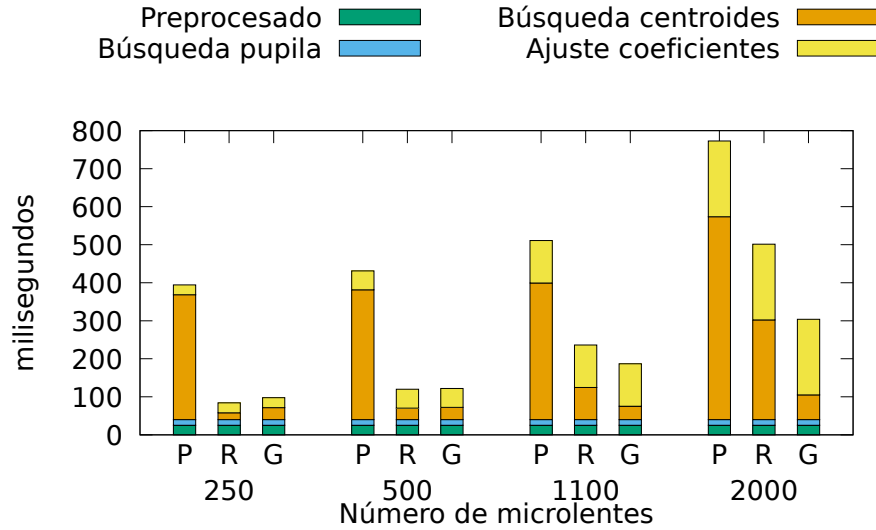


Figura 4.10: Tiempo de ejecución en la CPU desglosado por etapas en función del número de puntos procesados (tamaño de la imagen fijo a 2560x2560 píxeles). *P* para *PaPyCS*, *R* para Reichenbach y *C* para Centro de Masa

píxeles a 1.67ms para imágenes de 2560x2560 píxeles para el algoritmo PAPYCS. Pero como se ha comentado anteriormente es muy poco común que se utilice una configuración de características similares en sistemas ópticos reales.

Igualmente, la figura 4.12 muestra el tiempo de ejecución desglosado por etapas de cada uno de los algoritmos para tres tamaños de imagen diferentes en la CPU. En este caso se incrementa sobre todo el tiempo de preprocesamiento y de búsqueda de la pupila en todos los algoritmos. El tiempo de preprocesamiento pasa de 1.32ms a 25.2ms para 640x640 píxeles y 2560x2560 píxeles respectivamente. Y el tiempo de búsqueda de la pupila pasa de 2.72ms a 15.0ms para los mismos tamaños de imagen. Se incrementa también de forma muy importante el tiempo empleado en la etapa de búsqueda de centroides por parte del algoritmo PAPYCS y el del Centro de Masas, que crece de forma lineal con el aumento de trabajo que se multiplica por 4 al pasar de un tamaño al siguiente. También aumenta el tiempo que emplea el algoritmo de Reichenbach en la búsqueda de centroides en una proporción similar, pero de forma mucho más moderada en términos absolutos. El tiempo de ejecución de la etapa de búsqueda de centroides para el algoritmo PAPYCS pasa de 22.9ms a 328.2ms, para imágenes de 640x640 y 2560x2560 píxeles respectivamente. Mientras que sólo pasa de 4.1ms a 31.5ms para el algoritmo de Reichenbach con imágenes de los mismos tamaños.

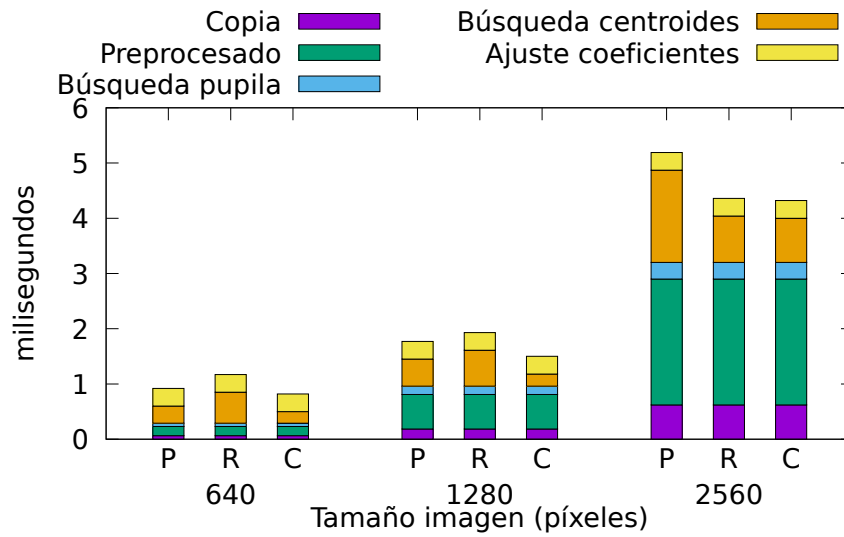


Figura 4.11: Tiempo de ejecución en la GPU desglosado por etapas en función del tamaño de la imagen (número de microlentes fijo a 250). *P* para PAPYCS, *R* para Reichenbach y *C* para Centro de Masa

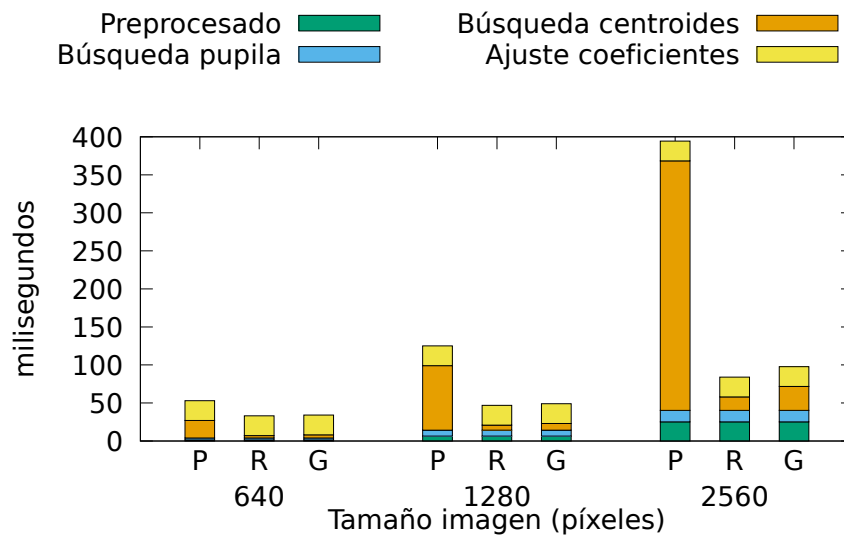


Figura 4.12: Tiempo de ejecución en la CPU desglosado por etapas en función del tamaño de la imagen (número de microlentes fijo a 250). *P* para PAPYCS, *R* para Reichenbach y *C* para Centro de Masa

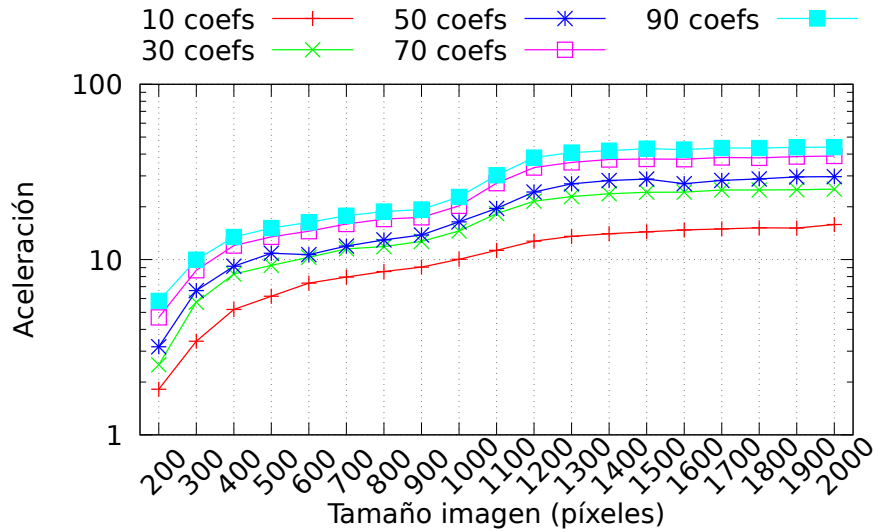


Figura 4.13: Aceleración del cómputo del mapa de frente de onda en función del tamaño de imagen para varias cantidades de coeficientes de Zernike.

En lo que respecta al cálculo del mapa de frente de onda la figura 4.13 muestra la aceleración conseguida por la versión GPU respecto de la CPU en función del tamaño de imagen para cinco cantidades de coeficientes de Zernike. Cuando se calcula el mapa de un frente de onda éste se puede calcular para diferentes tamaños, que pueden variar en función de la resolución del elemento corrector que se esté utilizando. Asimismo, el mapa de un frente de onda se puede calcular para distintas cantidades de coeficientes de Zernike, la cuál dependerá de la cantidad de polinomios de Zernike que se hayan utilizado para procesar la imagen de H-S. La implementación paralela muestra una buena escalabilidad tanto al aumentar el tamaño de la imagen como al aumentar el número de coeficientes, consiguiendo aceleraciones de hasta 40x en las configuraciones más exigentes en cuanto a cantidad de cómputo.

En la figura 4.14 se muestran los resultados de rendimiento expresados en imágenes por segundo en escala logarítmica para las mismas configuraciones que en la figura anterior. Además, los resultados se muestran para la implementación CUDA y para la implementación secuencial CPU en C++. Como es de esperar, el rendimiento obtenido es mayor para imágenes más pequeñas y una menor cantidad de coeficientes de Zernike puesto que es menor la cantidad de trabajo a realizar. Destacan las 200 imágenes/s que es capaz de procesar la versión CUDA para la configuración de 2000x2000 píxeles con 90 coeficientes.

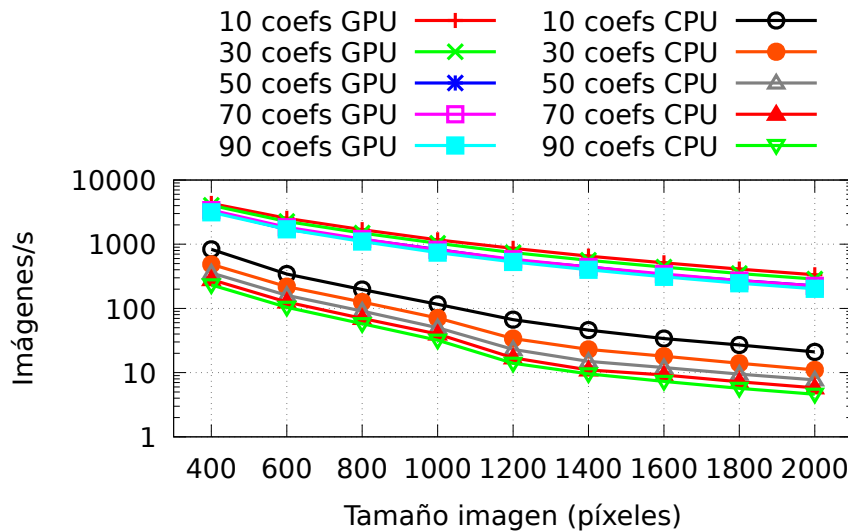


Figura 4.14: Velocidad en imágenes/s del cómputo del mapa de frente de onda en función del tamaño de imagen para varias cantidades de coeficientes de Zernike.

4.4.3 Precisión alcanzada por los diferentes algoritmos

La figura 4.15 muestra el valor del RMS (Root Mean Square) de la aberración medida en una serie de imágenes con varios valores de aberración. Si la medida fuera perfecta se obtendría un RMS igual al de la imagen y en la gráfica aparecería como la línea negra. Sin embargo, se puede ver que hay errores, generados tanto por la aberración como por los reflejos introducidos, que hacen que el RMS medido difiera del esperado para algunas imágenes. Se observa que el algoritmo PAPYCS presenta unos resultados mucho mejores que el algoritmo de Reichenbach que suele fallar en imágenes que presentan reflejos corneales.

La figura 4.16 muestra con más detalle la precisión obtenida por los algoritmos al procesar las imágenes de prueba. En esta gráfica se muestra el porcentaje de imágenes que cada algoritmo consigue procesar con un error menor que 1 en el RMS. Asimismo, se presenta el error medio obtenido por cada algoritmo. Así, se puede ver como el algoritmo PAPYCS obtiene unos resultados mucho mejores, siendo capaz de procesar todas o casi todas las imágenes con RMS menor que 5 correctamente. Mientras que el método de Reichenbach presenta más problemas con algunas imágenes, especialmente las que contienen reflejos y obtiene una precisión menor.

La figura 4.17 muestra la precisión de las medidas desde otro punto de vista, en concreto, muestra el error cometido por cada método calculando el centroide de cada punto. Los

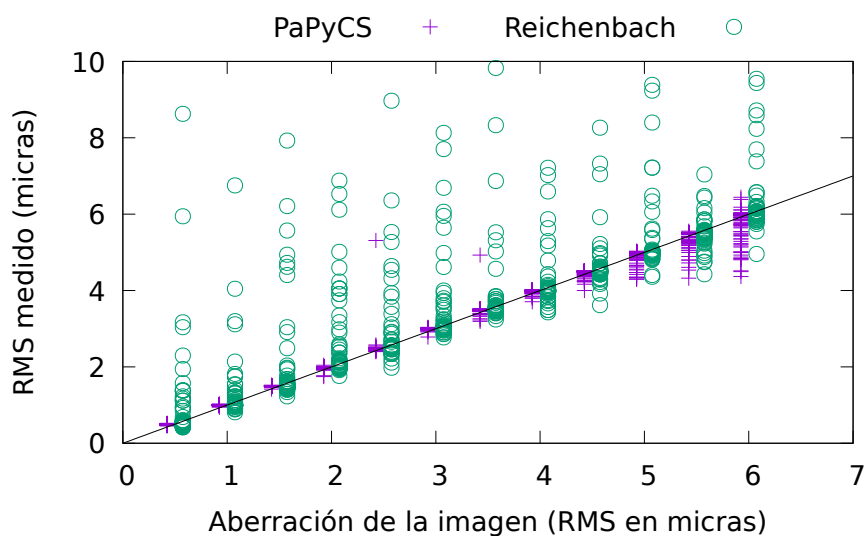


Figura 4.15: Comparación del RMS de la aberración medida de los dos métodos y el teórico. La línea gris muestra el valor teórico.

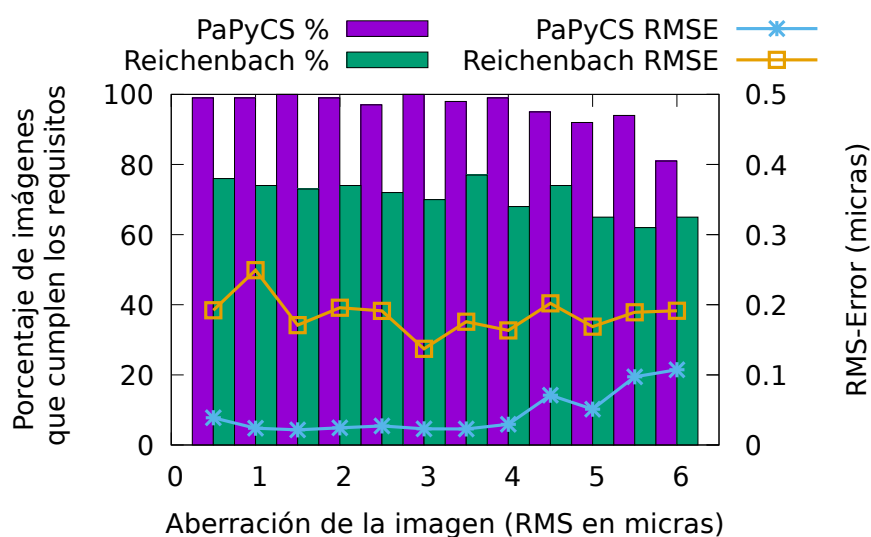


Figura 4.16: Las barras (eje Y izquierdo) muestran el porcentaje de imágenes con un RMSE más pequeño que 1. Las líneas (eje Y derecho) muestran el RMSE medio para las mismas imágenes.

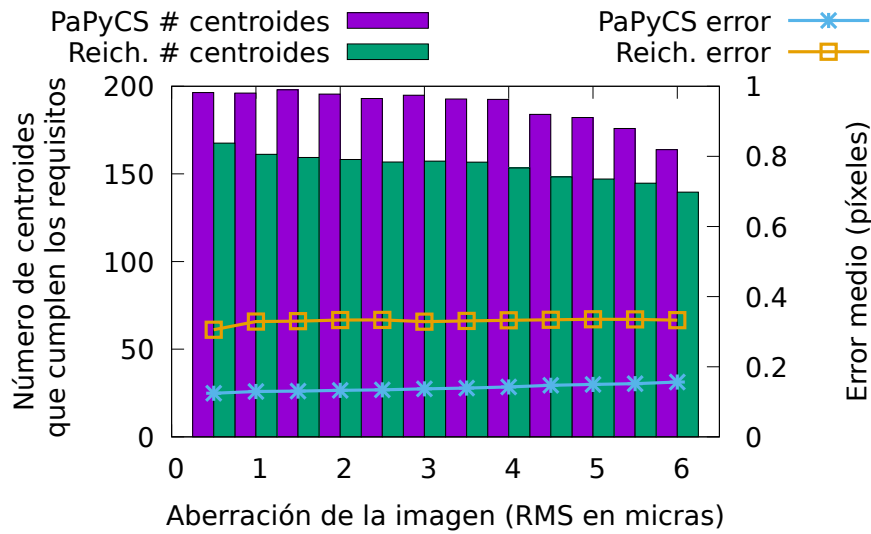


Figura 4.17: Las barras (eje y izquierdo) muestra el número de centroides detectados con un error menor que 1 píxel. Las líneas (eje y derecho) muestra el error medio en píxeles.

errores en el cálculo de los centroides se trasladan directamente a errores en el cálculo de las aberraciones. Por lo tanto, estos resultados están conectados con los mostrados en la figura 4.16 y se observa una tendencia similar, con el algoritmo PAPYCS obteniendo una mayor precisión que el algoritmo de Reichenbach. El error medio en el cálculo del centroide se sitúa por debajo de 0.2 píxeles para PAPYCS y por debajo de 0.4 píxeles para Reichenbach.

Finalmente, la figura 4.18 muestra el valor del desenfoque medido en una serie de imágenes de un ojo artificial. En este caso se ha utilizado un sistema real para tratar de obtener unos resultados más realistas, sin embargo, como es imposible tener una referencia de la aberración teórica con la que compararse se ha optado por utilizarlo para medir la estabilidad de la medida que también es un factor importante. Al tratarse de un ojo artificial todas las oscilaciones presentes en la imagen deben proceder de variaciones que pueda tener el sensor de la cámara y variaciones del láser, el resto de elementos permanecen constantes. Dichas variaciones son, en principio, muy pequeñas y apenas deben afectar a la medida. A pesar de ello se puede ver como el método de Reichenbach es más sensible a esas pequeñas variaciones que PAPYCS, que obtiene valores muy similares para el desenfoque en toda la serie. En cualquier caso, es importante tener en cuenta la escala de la figura y lo pequeñas que son las variaciones medidas.

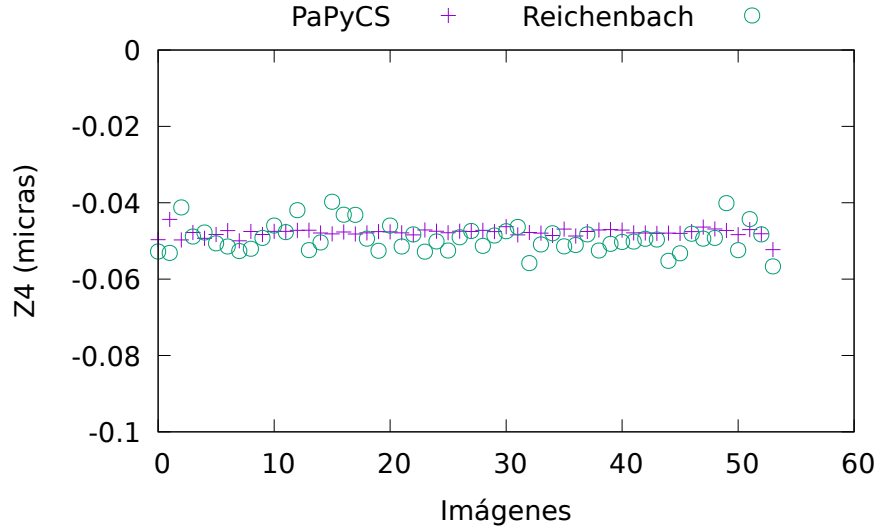


Figura 4.18: Desenfoque medido (en micras) en una serie de imágenes de un sistema H-S real en condiciones estables.

4.5 Sistemas ópticos reales donde se está utilizando esta implementación paralela de procesamiento de imágenes de sensores de Hartmann-Shack

Fruto de esta investigación se ha desarrollado una librería que ofrece varias alternativas para procesar imágenes de sensores Hartmann-Shack. Dicha librería incluye la mayoría de las opciones aquí presentadas: versión secuencial en CPU y versión paralela en GPU; algoritmo PAPYCS y algoritmo centro de masa; y seguimiento pupilar. Además, incluye una versión paralela en CPU utilizando OpenMP. También incluye métodos alternativos para realizar el seguimiento pupilar, orientados específicamente a imágenes de lentes artificiales que presentan características distintas de las de los ojos humanos. La variedad de opciones disponibles en la librería pretende obtener las mejores prestaciones de cada sistema, en concreto, se intenta optimizar el mismo disponga o no de una GPU compatible con CUDA.

Además, para reducir la complejidad de uso de la librería y abstraerla de la implementación subyacente en CUDA se ha diseñado una interfaz completamente independiente de la misma. Así, los datos de entrada al procesado se pasan utilizando objetos de imágenes de OpenCV. Si bien esto supone una pequeña pérdida de rendimiento si se compara con

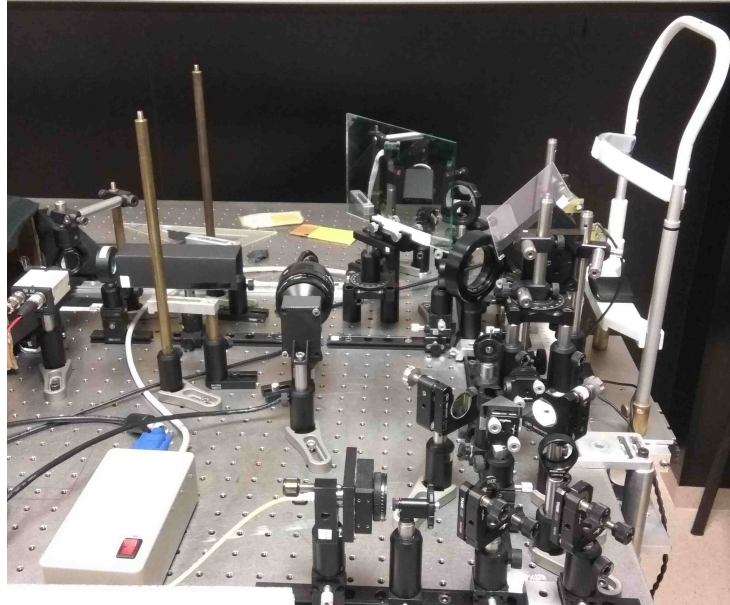
una implementación que haga uso de *pinned memory* también supone una facilidad de uso muy superior. Por otra parte, el resultado del procesamiento incluye información sobre los centroides detectados que se puede utilizar para ofrecer una visión más completa del funcionamiento del sistema al operador si así se desea.

Actualmente la librería desarrollada se está utilizando en 5 sistemas ópticos. En 4 de ellos se utiliza la versión paralelizada en CUDA que permite procesar las imágenes en tiempo real y en 1 de ellos se utiliza la versión CPU por carecer de una GPU que soporte CUDA.

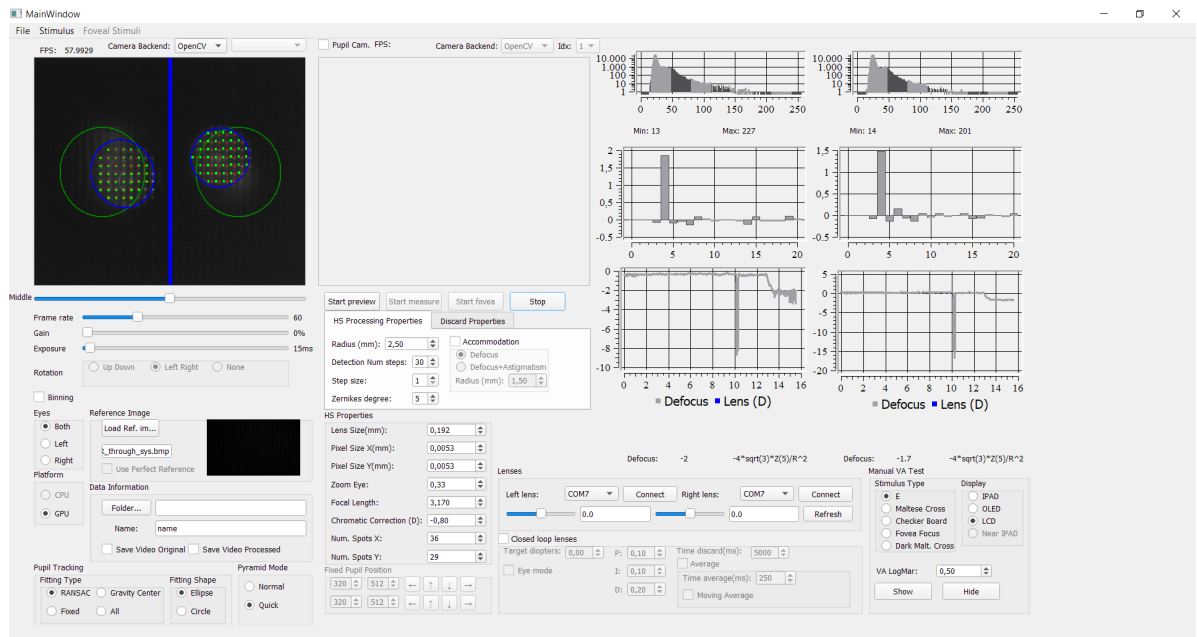
En la figura 4.19a se muestra uno de los sistemas Hartmann-Shack que utilizan la librería desarrollada para procesar en tiempo real las imágenes de la cámara. Este sistema en concreto es además capaz de procesar en paralelo imágenes de pupila también en la GPU en tiempo real. Se utilizó con dicho propósito para medir las aberraciones de los ojos del sujeto mientras se controlaban unas lentes optoelectrónicas con la información del seguimiento de pupila [59]. Este sistema también se ha utilizado para llevar a cabo experimentos relativos a la miopía [81].

Por otra parte, la figura 4.19b muestra el programa que se usa para controlar dicho sistema Hartmann-Shack. Este programa hace uso de la librería desarrollada en esta Tesis para procesamiento de imágenes de sensores Hartmann-Shack. La cámara que se usa actualmente en dicho sistema alcanza una velocidad de 60 imágenes por segundo con una resolución de 1280x1024 píxeles. Por tanto, la velocidad máxima de este sistema es 60 imágenes por segundo, lo cuál es una velocidad muy alta si se tiene en cuenta la longitud de onda del láser utilizado (1050nm) y la configuración abierta del sistema.

Asimismo, la figura 4.20 muestra un ejemplo de la acomodación de un sujeto que mira alternativamente a dos estímulos, uno situado a más de 2 metros y otro situado a 50cm. El desenfoque medido corresponde a la potencia que el ojo del sujeto emplea para enfocar el estímulo cercano. Además, en esa figura se muestra el radio de la pupila del sujeto en todo momento. Para obtener dicho radio se ha empleado el algoritmo de seguimiento pupilar sobre imágenes de Hartmann-Shack desarrollado en esta Tesis. Análogamente los datos de acomodación se han obtenido con la implementación GPGPU desarrollado en esta Tesis.



(a)



(b)

Figura 4.19: (a) Sistema binocular Hartmann-Shack que permite la visión directa de estímulos.
(b) Programa de control del sistema de Hartmann-Shack mostrado en (a).

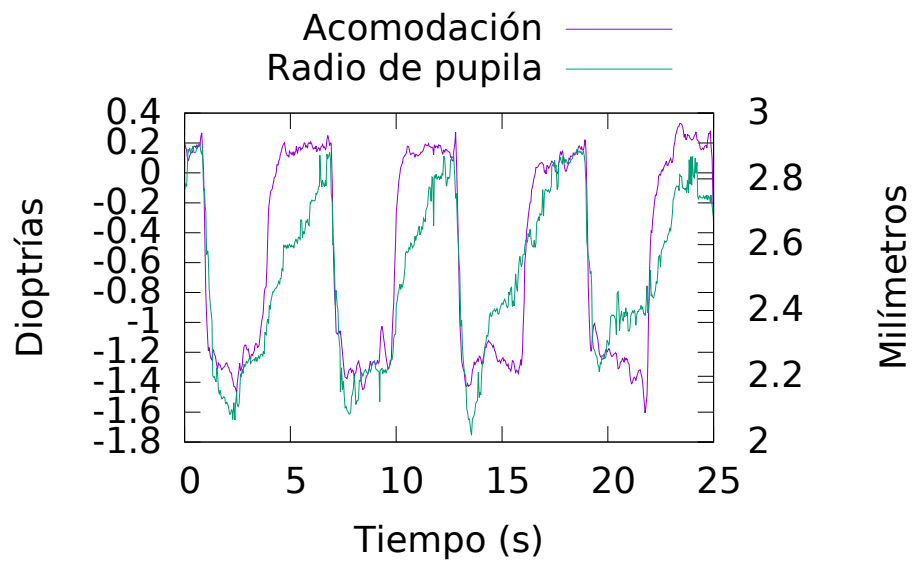


Figura 4.20: Medida del desenfoque de un sujeto que mira alternativamente a un estímulo en cerca y lejos.

5 Paralelización del procesamiento de imágenes de OCT

5.1 Introducción

El OCT por sus siglas en inglés o tomografía de coherencia óptica es una tecnología extremadamente útil en muchos campos y particularmente interesante para el estudio de las cataratas *in-vivo* que es el uso que se le ha dado. Este tipo de tomografía consigue profundizar bastante en el tejido y permite obtener imágenes de muy alta resolución en un tiempo relativamente pequeño. Además, en el caso del OCT desarrollado conjuntamente por el LOUM e investigadores de la Universidad Nicolás Copérnico de Torún (Polonia), las imágenes obtenidas abarcan desde la córnea del ojo hasta el cristalino del mismo [82, 83, 84]. Aunque también se ha empleado una variación del OCT para obtener imágenes de los cuerpos presentes en el humor vítreo [85, 86, 87].

El interés del OCT para el estudio de las cataratas radica en su capacidad no solo de tomar imágenes con precisión sino volúmenes, ofreciendo una vista detallada de toda la catarata. Esto permite identificar con precisión el tipo y estado de catarata ante el que nos encontramos, así como medir parámetros de la misma que de otra forma no sería posible.

La figura 5.1 muestra un ejemplo de una imagen de un ojo humano. Para generar dicha imagen se capturó un volumen de un ojo y a partir de dicho volumen se calculó su proyección frontal. Si bien dicha figura muestra una visión muy amplia de un ojo incluyendo ambos párpados y pestañas, es posible reducir el tamaño del área escaneada para aumentar la resolución de la imagen final. El sistema permite capturar una cantidad limitada de datos aunque es posible configurar la densidad de la información según las necesidades, siempre dentro de los límites de configuración de la óptica y hardware utilizados.

Por otra parte, ha sido posible establecer una relación entre la intensidad de la catarata

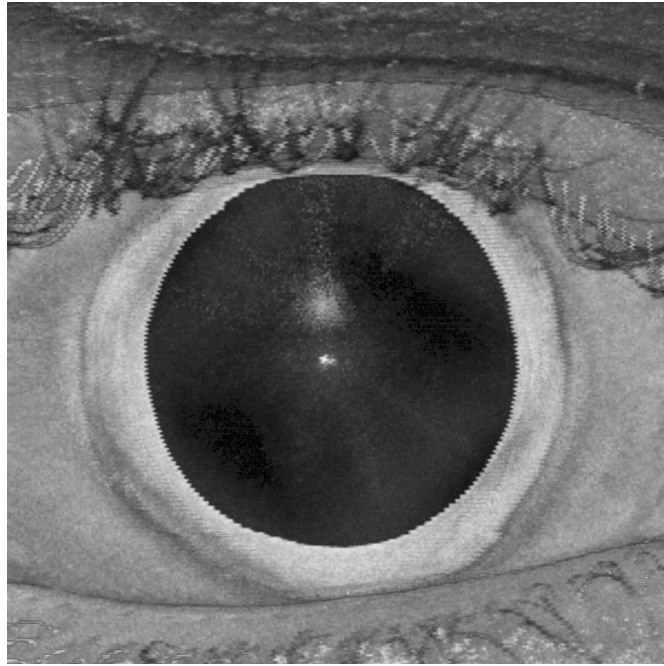


Figura 5.1: Ejemplo de imagen frontal de un ojo sano generada a partir de un volumen del mismo.

en las imágenes y cómo de avanzada se encuentra la misma. Este estudio, llevado a cabo por *de Castro et al.* [84], se centró en las cataratas nucleares, las cuáles como su propio nombre indica se encuentran en el centro o núcleo del cristalino. Para ello se tomaron imágenes con el OCT de pacientes que tenían cataratas en distintas fases que iban a ser sometidos a cirugía para eliminarla. Simultáneamente a la captura de imágenes, un/a oftalmólogo/a revisaba a los pacientes utilizando una lámpara de hendidura (herramienta ampliamente utilizada en oftalmología para la graduación de cataratas) y graduaba la catarata de acuerdo a los parámetros de la escala LOCS III [88]. Posteriormente, se calculó el valor medio de la intensidad de los píxeles en el área central de la imagen y se comparó dicho valor con el obtenido en la escala LOCS III, llevando a cabo una correlación entre ambos.

El OCT es por tanto una herramienta muy útil para el estudio de las cataratas y es por ello una tarea en la que es interesante mejorar su rendimiento. Hay que tener en cuenta que un volumen OCT suele contar con cantidades ingentes de datos. En concreto, un volumen típico generado por nuestro sistema OCT cuenta con 300x300x8192 datos de entrada y hasta 300x300x4096 píxeles de salida. Si se utilizan *floats* para almacenar los datos, el volumen de entrada ocupa unos 2.7GB. Además, el procesamiento de los datos de

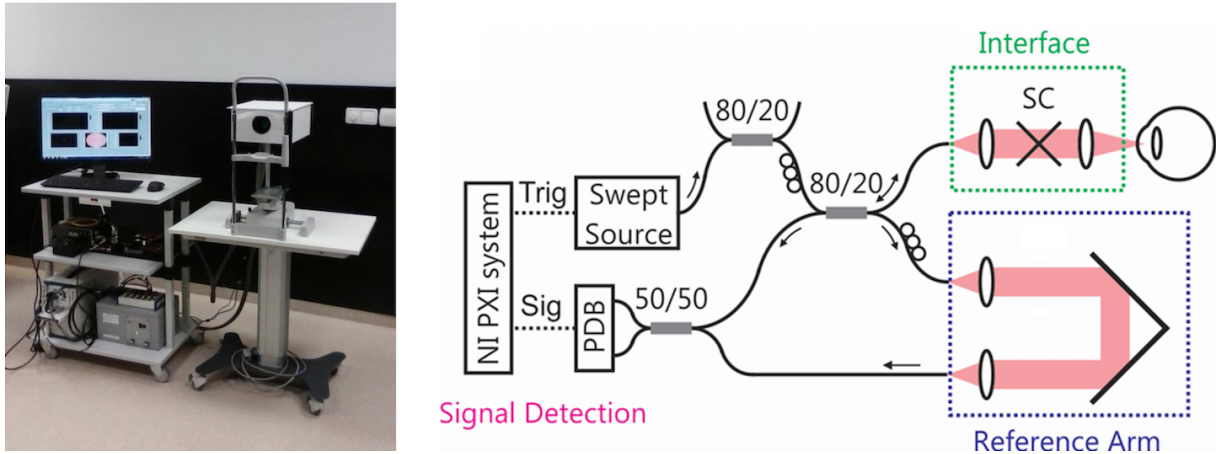


Figura 5.2: Imagen real y esquema del sistema OCT instalado en el LOUM [84].

OCT conlleva varios pasos entre los que se encuentra una transformada rápida de Fourier que requiere una gran cantidad de cómputo.

Por otra parte, la captura de los volúmenes de OCT en sujetos con cataratas se tiene que repetir con frecuencia debido a que los mismos no ven bien y en ocasiones tienen dificultades para mantener la mirada fija durante los 2 segundos que dura la medición. Generando como consecuencia imágenes con movimientos que no son válidas. Si el procesamiento de las imágenes lleva mucho tiempo, el proceso se vuelve tedioso y la capacidad de colaboración y concentración de los sujetos disminuye empeorando la calidad de las medidas.

El funcionamiento del OCT implementado en el LOUM se resume en la figura 5.2. Una fuente de luz que se divide entre el brazo de muestreo y el brazo de referencia y posteriormente la luz se recombina y llega a un fotodetector que envía los resultados al ordenador donde los datos son procesados. Este OCT es conocido como SSOCT (Swept Source OCT) un tipo de OCT en el que la fuente de luz es capaz de cambiar la frecuencia a la que emite dinámicamente. Las distintas frecuencias codifican la profundidad de los datos capturados, mientras que el brazo de muestreo mueve el haz por la muestra codificando las dimensiones X e Y de la misma. Asimismo, sobre los datos obtenidos por el fotodetector es necesario llevar a cabo un intenso procesamiento que incluye una FFT para obtener la imagen final.

Si bien los sistemas SSOCT son extremadamente rápidos si se comparan con los primeros OCTs desarrollados en los que se desplazaba el brazo de referencia para codificar la profundidad su velocidad sigue siendo una limitación. Esto es debido a la gran cantidad

de datos que cada medida de OCT requiere si se quiere obtener una alta resolución. A pesar de que los elementos del sistema funcionan a varios KHz, obtener volúmenes de datos con una alta densidad de muestreo supone muchos retos.

5.2 Estado del arte

Dada la naturaleza masivamente paralela del procesamiento de los datos de los sistemas OCT su procesamiento se ha optimizado en multitud de ocasiones para ser ejecutado en GPUs [89, 90, 91, 92]. Estas implementaciones aprovechan de una forma muy similar las oportunidades de paralelización que ofrecen los datos. Asimismo, obtienen unos resultados excelentes cuando se comparan con los obtenidos por las versiones secuenciales. La mayor capacidad de cómputo y el mayor ancho de banda confieren a las GPUs unas cualidades únicas para este tipo de algoritmos. Aunque los resultados de estos estudios no son directamente comparables debido a diferencias en el hardware utilizado y pequeñas diferencias en el procesamiento aplicado, el rendimiento obtenido por todos ellos es similar. Además, el principal cuello de botella suele encontrarse en la transferencia entre la CPU y la GPU que está limitada por la velocidad del bus PCIe. En esos estudios la versión PCIe utilizada fue la 2.0 que limita la velocidad de transferencia a 8GB/s, aunque en condiciones reales el máximo alcanzable suelen ser entre 5 y 6GB/s.

La tabla 5.1 muestra los procesadores gráficos utilizados en algunos de los artículos citados, las configuraciones que han utilizado y las velocidades obtenidas. Es importante tener en cuenta que hay grandes diferencias entre el procesamiento que se ha llevado a cabo en unos casos y en otros. En el caso de Jian *et al.* [90] los resultados no se copian a la CPU, sino que se muestran directamente en pantalla mediante el uso de OpenGL. Además, en este caso cada píxel sólo ocupa 2 bytes cuando se copia de la CPU a la GPU, aumentando la velocidad de copia en 2x con respecto a otras implementaciones que empleen 4 bytes para almacenar cada píxel. Asimismo, solamente se llevan a cabo dos operaciones en la GPU, el cálculo de la Transformada de Fourier y la selección de una parte del resultado. Esto contrasta con otros procesados de datos de sistemas OCT que llevan a cabo más operaciones sobre los datos, lo cual conlleva una mayor carga computacional.

Autores	GPU	Tamaño	Rendimiento
Zhang et al. [91]	GTX 480	1x4096x2048	1.09 Gpx/s
Jian et al. [90]	GTX 680	200x256x2048	2.44 Gpx/s
Cho et al. [92]	GTX 480	1x2048x512	655.36 Mpx/s
Cheon et al. [89]	GTX 590	1x4096x128	806.60 Mpx/ s

Tabla 5.1: Rendimiento obtenido en diferentes trabajos procesando datos obtenidos en sistemas OCT utilizando diferentes configuraciones y hardware. Si bien el procesamiento llevado a cabo no es el mismo en todos los casos.

También es común que la velocidad de procesamiento de los datos sea superior a la de adquisición. Por ejemplo, Cho *et al.* son capaces de procesar a 655.36Mpx/s pero sólo pueden adquirir a 115.34Mpx/s debido a una limitación en la velocidad del dispositivo de adquisición, en ese caso se trata de una cámara lineal con una velocidad de 70000 líneas/s.

Zhang *et al.* [91] han llevado a cabo una comparativa del rendimiento y precisión de varios algoritmos de procesamiento de OCT. Además, han integrado la implementación desarrollada para GPU en un sistema real obteniendo una velocidad de procesamiento de 122K líneas/s (líneas de 1024 píxeles). Igual que en otros casos, la velocidad está limitada por el sistema de adquisición y no por el sistema de procesamiento. De hecho, para uno de los algoritmos implementados obtuvieron una velocidad superior a 1M líneas/s (líneas de 1024 píxeles).

El objetivo de aumentar el rendimiento del procesamiento de datos de OCT es normalmente o bien ser capaces de procesarlos en tiempo real o simplemente reducir la espera en el postprocesado. Cheon *et al.* han utilizado sus implementaciones para mejorar herramientas utilizadas en microcirugías que requieren una gran precisión y se benefician enormemente de poder ver dónde se sitúan las herramientas y el tejido [89, 93]. También se ha implementado una versión paralela en GPU con CUDA para la monitorización de la vascularización de la retina [94].

El procesamiento de OCT consiste principalmente en calcular la transformada de Fourier de los datos obtenidos. Si bien se llevan a cabo otras operaciones que mejoran la calidad del resultado, estas suelen ser opcionales. Por otra parte, la transformada de Fourier es una herramienta utilizada en muchas áreas, lo que ha motivado que se hayan desarrollado numerosas librerías para llevar a cabo dicho cálculo en múltiples lenguajes de programación. Steinbach y Werner [95] llevaron a cabo una comparativa del rendimiento de las tres librerías utilizadas en esta Tesis: *clFFT* [96], *cuFFT* (CUDA Fast Fourier Transform

library) y *FFTW* [97]. Dichas librerías están programadas en OpenCL, CUDA y C, respectivamente. De una forma similar a los resultados obtenidos en ese estudio, la librería *cuFFT* ha obtenido el mejor rendimiento, seguida de la librería *clFFT* y en último lugar *FFTW*. El rendimiento de estas librerías está evidentemente ligado al hardware sobre el que se utilizan, de forma que las diferencias de rendimiento son más atribuibles a ese factor que a las librerías en sí.

5.3 Comparación de implementaciones del procesamiento de imágenes de OCT para varias plataformas

Se ha llevado a cabo una comparación de varias implementaciones que se han desarrollado con diferentes propósitos para el procesamiento de los datos obtenidos. La primera implementación desarrollada por Lukasz Cwiklinski y después paralelizada para esta Tesis se realizó en LabVIEW como se describe en la sección 5.3.1. Esta fue la primera implementación que se utilizó para procesar las imágenes y su lentitud fue la que motivó el desarrollo de otras. En segundo lugar, Alberto de Castro [84] desarrolló otra implementación utilizando MATLAB para analizar los datos con más facilidad, discutida en la sección 5.3.2. Posteriormente, se han desarrollado tres implementaciones de alto rendimiento utilizando para ello C++, CUDA y OpenCL, descritas en las secciones 5.3.3, 5.3.4 y 5.3.5.

El funcionamiento del procesamiento de las imágenes de OCT presentado en el algoritmo 9 muestra las operaciones que se aplican a los datos. Como se puede ver se llevan a cabo una cantidad importante de operaciones, especialmente si se tiene en cuenta la gran cantidad de datos que se están procesando (2.7GB de datos por cada medida). Si bien algunas operaciones de las aplicadas son indispensables en el procesamiento de cualquier dato de OCT otras son opcionales o específicas de nuestros datos, tales como voltear los datos o seleccionar un área de interés. Aunque, la mayor parte de la carga computacional de la operación se concentra en la transformada rápida de Fourier.

5.3.1 Implementación LabVIEW

Una parte importante del hardware informático utilizado es de la marca National Instruments, la cuál es la desarrolladora de LabVIEW, por lo tanto la elección inicial de

Algoritmo 9 Pseudocódigo del algoritmo de procesado de datos de OCT

```

1: procedure PROCESAR OCT(Datos, indices)
2:    $rScans \leftarrow inicializarVector(numAScans)$ 
3:   for  $i \leftarrow 1$  a  $numAScans$  do
4:      $AScan \leftarrow Datos[i]$ 
5:      $AScan \leftarrow SubstraerFondo(AScan)$ 
6:      $AScan \leftarrow Shaping(AScan)$ 
7:      $AScan \leftarrow Interpolación(AScan)$ 
8:      $fftResultado \leftarrow FFT(AScan)$ 
9:      $resultado \leftarrow Log(fftResultado)$ 
10:     $resultado \leftarrow Voltar(resultado)$ 
11:     $resultado \leftarrow Normalizar(resultado)$ 
12:     $resultado \leftarrow Recortar(resultado)$ 
13:     $rScans[i] \leftarrow resultado$ 
14:   end for
15:   return  $rScans$ 
16: end procedure

```

LabVIEW para controlar el sistema y procesar los datos generados era bastante obvia. National Instruments ofrece librerías y drivers perfectamente integrados en LabVIEW que facilitan la utilización de su hardware. De hecho, se sigue utilizando LabVIEW para la captura de los datos puesto que la misma es muy compleja, incluyendo la sincronización del láser con la captura de datos por parte de una tarjeta conversora analógica/digital. Además, se cuenta con una FPGA que realiza un procesado en baja resolución en tiempo real que permite alinear con precisión a los sujetos en el sistema. Asimismo, para acelerar la escritura de los datos se utiliza un disco duro virtual alojado en memoria RAM.

Sin embargo, el rendimiento de LabVIEW para procesar los datos es muy pobre. Tanto es así que, como se ha comentado anteriormente, supone un problema durante las medidas por la necesidad de comprobar que las mismas se han realizado correctamente y las largas esperas que añade. Dados los problemas de rendimiento se paralelizó la ejecución del procesado utilizando LabVIEW y, aunque se consiguió un muy buen resultado utilizando las herramientas para paralelizar bucles de LabVIEW, el tiempo de ejecución seguía siendo muy elevado.

La tabla 5.2 muestra los tiempos de ejecución de las versiones secuencial y paralela de la implementación LabVIEW. Además, se incluyen dos resoluciones de los datos según se necesite mayor detalle o no. Como se puede ver las versiones paralelas obtienen unas aceleraciones muy buenas teniendo en cuenta que estas pruebas se han ejecutado en un

Configuración	Resolución (300*300*x)	Tiempo de ejecución (s)	Rendimiento
LabVIEW secuencial	8192	312.5	2.36 MPx/s
LabVIEW secuencial	4096	166.7	2.21 MPx/s
LabVIEW paralelo	8192	71.4	10.3 MPx/s
LabVIEW paralelo	4096	38.0	9.7 MPx/s

Tabla 5.2: Tiempo de ejecución de las versiones secuenciales y paralelas del procesamiento de datos OCT en LabVIEW para dos resoluciones.

procesador Intel i7-3610QE que cuenta con 4 núcleos y 8 hilos. La aceleración obtenida es de 4.4x que, teniendo en cuenta el hardware utilizado, es un muy buen resultado e indica que la paralelización en LabVIEW obtiene muy buenos resultados, los cuáles son especialmente destacables dada la simplicidad de su implementación. Para paralelizar un bucle en LabVIEW basta con habilitar la paralelización en la ventana de propiedades del mismo e indicar cuántos hilos se deben utilizar, siempre y cuando no haya dependencias entre los datos. Soporta así mismo la paralelización anidada si hay bucles dentro de bucles. También deja entrever que el procesamiento de los datos de OCT es una tarea con muchos datos independientes y, por tanto, con muy buenas posibilidades de paralelización.

5.3.2 Implementación MATLAB

La implementación MATLAB desarrollada por Alberto de Castro destaca por su simplicidad inherente en un lenguaje como MATLAB en el que todas las variables son matrices y que, por tanto, encaja muy bien con los datos de OCT.

Por otra parte, MATLAB utiliza el paralelismo por defecto para procesar los datos, por lo que obtiene resultados muy buenos si se comparan con los obtenidos por la versión LabVIEW. Así la tabla 5.3 muestra los tiempos de ejecución obtenidos por la implementación MATLAB para las dos resoluciones probadas. Es importante tener en cuenta que las pruebas de esta implementación se han hecho en un ordenador diferente por la imposibilidad de instalar MATLAB en el ordenador que ejecuta LabVIEW y viceversa. El ordenador en el que se ha probado la versión MATLAB está equipado con un procesador AMD 1800x (8 núcleos y 16 hilos) de unas prestaciones significativamente superiores a las del Intel i7-3610QE que se ha utilizado para probar la versión LabVIEW. La diferencia del hardware es relevante para comparar los resultados, ya que la versión MATLAB obtiene una aceleración aproximada de 2x con respecto a la versión LabVIEW, siendo dicha aceleración atribuible a la diferencia de hardware.

Configuración	Resolución (300*300*x)	Tiempo de ejecución (s)	Rendimiento
MATLAB	8192	29.7	24.8 MPx/s
MATLAB	4096	17.3	21.3 MPx/s

Tabla 5.3: Tiempo de ejecución del procesamiento de datos OCT en MATLAB para dos resoluciones.

La figura 5.4 muestra los tiempos de ejecución de cada una de las etapas del procesamiento de la implementación MATLAB para dos resoluciones: 300x300x4096 y 300x300x8192 píxeles. En dicha figura se observa que si bien la etapa con un mayor tiempo de ejecución es la transformada de Fourier, hay otras etapas del procesamiento con tiempos de ejecución muy similares, como la normalización y la interpolación. Un análisis global pone de manifiesto que es necesario optimizar todas las etapas del procesamiento si se quiere obtener una mejora significativa de los tiempos de ejecución.

5.3.3 Implementación C++

La versión C++ utiliza OpenMP para paralelizar los bucles del procesamiento que se ha considerado oportuno. Asimismo, para calcular la transformada rápida de Fourier se ha utilizado la librería FFTW 3 (Fastest Fourier Transform in the West) [97], que es una de las librerías más rápidas disponibles actualmente para el procesamiento de transformadas rápidas de Fourier.

La implementación C++ mimetiza el algoritmo 9 previamente mostrado en su mayor parte en lo que a la estructura de las funciones se refiere. Sin embargo, algunas optimizaciones aplicadas difuminan la lógica del programa ligeramente. Así, algunas operaciones como el *shaping* y la *interpolación* se pueden fusionar en una sola función para procesar cada dato simultáneamente, evitando operaciones innecesarias de almacenamiento y carga de datos a/de RAM.

Para probar el rendimiento de esta implementación se ha utilizado el mismo ordenador que en el caso de la implementación MATLAB. La tabla 5.4 muestra el rendimiento obtenido por las diferentes configuraciones, se ha ejecutado la implementación con varias cantidades de hilos, desde 1 hasta 16. Si se compara con el rendimiento obtenido por MATLAB los resultados son muy buenos, consiguiendo una aceleración de 11.4x y 16.8x para las resoluciones de 8192 y 4096, respectivamente. Si bien la escalabilidad obtenida por la paralelización con OpenMP no es excelente. A pesar de lo cuál los tiempos se

Configuración	Resolución (300*300*x)	Tiempo de ejecución (s)	Rendimiento
C++ secuencial	8192	15.00	48.9 MPx/s
C++ OpenMP 2 hilos	8192	8.25	88.5 MPx/s
C++ OpenMP 4 hilos	8192	4.17	173.5 MPx/s
C++ OpenMP 8 hilos	8192	3.33	216.2 MPx/s
C++ OpenMP 16 hilos	8192	2.60	275.1 MPx/s
C++ secuencial	4096	7.30	50.2 MPx/s
C++ OpenMP 2 hilos	4096	3.70	98.6 MPx/s
C++ OpenMP 4 hilos	4096	1.93	187.1 MPx/s
C++ OpenMP 8 hilos	4096	1.10	323.4 MPx/s
C++ OpenMP 16 hilos	4096	1.03	341.3 MPx/s

Tabla 5.4: Tiempo de ejecución del procesamiento de datos OCT en C++ para varias configuraciones y dos resoluciones. Tiempos obtenidos en un procesador AMD Ryzen 7 1800X.

han reducido mucho y se acercan al objetivo de procesarlos en tiempo real. El volumen de mayor resolución se procesa en tan solo 2.60 segundos, mientras que el de menor resolución se procesa en 1.03 segundos.

Por otra parte, la figura 5.4 muestra los tiempos de ejecución de cada una de las etapas del procesamiento de los datos del OCT. Se muestran los tiempos de ejecución de la versión paralela con 16 hilos por ser la que ha obtenido un mejor rendimiento. En el desglose de los tiempos de ejecución destacan tres etapas por su peso en el tiempo global del procesamiento: el logaritmo, la FFT, y la interpolación. Siendo la interpolación el procesamiento en el que se emplea más tiempo después de llevar a cabo la paralelización.

Al analizar los datos de la figura 5.4 hay que tener en cuenta que para evitar añadir sincronizaciones innecesarias los tiempos de ejecución desglosados no son totalmente precisos. Los 16 hilos ejecutan de forma independiente cada una de las etapas, por lo que los tiempos de ejecución de cada etapa para cada uno de los hilos se suman y se dividen entre 16. Sin embargo, si durante la ejecución alguno de los hilos termina su trabajo antes que el resto, este método genera un pequeño error en los resultados. Ese error se traslada en que la suma de las diferentes etapas es inferior al tiempo total de ejecución contabilizado. Asimismo, también hay que tener en cuenta que la creación y gestión de los hilos añade cierta sobrecarga que no se ha cuantificado.

5.3.4 Implementación CUDA

Al igual que en la implementación C++ en este caso también se mimetiza la lógica del algoritmo 9 y se traslada al código casi en su totalidad. Se han fusionado algunas operaciones como la *interpolación* y el *shaping* para evitar cargas y almacenamientos innecesarios en memoria global.

La implementación CUDA es completamente asíncrona, puesto que todos los kernels y copias de memoria se ejecutan asíncronamente. Esto permite maximizar la utilización de la GPU y solapar cómputo y copias de memoria, que en este caso suponen una parte importante de la ejecución debido al elevado volumen de datos que es necesario copiar. Si bien un volumen de OCT se puede ver como una unidad en realidad, se trata de un conjunto de imágenes 2D totalmente independientes. Aprovechando esa característica los datos se procesan de forma independiente para cada una de esas imágenes, es lo que permite solapar cómputo y copias de datos de la CPU a la GPU. Por supuesto, para poder ejecutar todo ese trabajo asíncronamente se han empleado streams de CUDA.

Por otra parte, para llevar a cabo la transformada rápida de Fourier en CUDA se ha empleado la librería cuFFT desarrollada por NVIDIA. Antes del comienzo de la ejecución se generan planes para la configuración de datos que se va a procesar. En el caso de una resolución de 300x300x8192 para cada imagen 2D se llevan a cabo 300 transformadas rápidas de Fourier de 8192 elementos cada una. La librería cuFFT permite configurar un plan para que se procesen varias transformadas rápidas de Fourier simultáneamente, por lo que todas se pueden llevar a cabo a la vez. Finalmente, se asigna a cada plan el stream que se encarga de procesar esa imagen.

Otra optimización que se ha llevado a cabo es la utilización de memoria bloqueada o *pinned memory*. Esta memoria permite aumentar el ancho de banda entre la CPU y la GPU, puesto que desde esta memoria se pueden programar copias a través de la conexión PCIe directamente. Mientras que si se utiliza memoria paginada es necesario realizar una copia intermedia a memoria bloqueada que reduce el rendimiento. Sin embargo, debido a que la memoria bloqueada es un recurso escaso, y que si se usa en exceso puede reducir el rendimiento global del sistema, es importante encontrar un compromiso en su uso. Especialmente en un caso como este en el que la cantidad de datos procesados es muy grande.

La implementación CUDA logra unos espectaculares resultados de rendimiento. La tabla 5.5 muestra los tiempos de ejecución de esta versión para dos tarjetas gráficas y dos

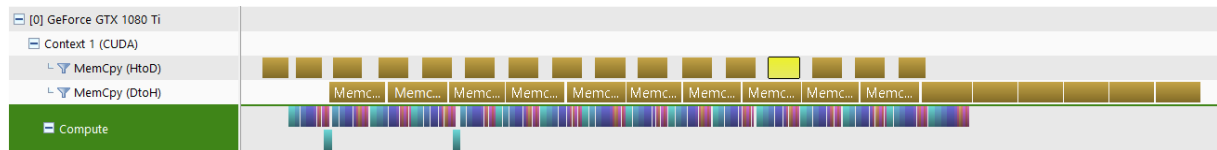
Configuración	Resolución (300*300*x)	Tiempo de ejecución (s)	Rendimiento
NVIDIA GTX 1080 Ti	8192	0.30	2.46 Gpx/s
NVIDIA GTX 1080 Ti	4096	0.16	2.30 Gpx/s
NVIDIA GTX 1050	8192	0.80	696 MPx/s
NVIDIA GTX 1050	4096	0.39	921 MPx/s

Tabla 5.5: Tiempo de ejecución del procesamiento de datos OCT en CUDA para varias configuraciones y dos resoluciones.

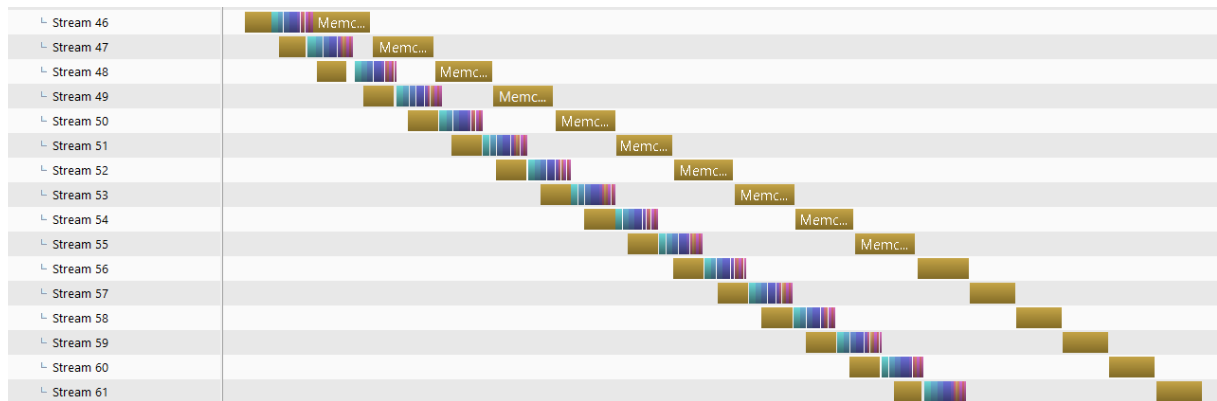
resoluciones. Se ha probado en una GPU de gama alta, la NVIDIA GTX 1080 Ti. Con este procesador gráfico se ha obtenido el mejor rendimiento, siendo capaz de procesar el volumen de 300x300x8192 píxeles en 0.3 segundos, o lo que es lo mismo, procesa a una velocidad de 2.46 Gpx/s. Mientras que el volumen de menor resolución se procesa en 0.16 segundos. Asimismo, también se ha probado en una GPU de gama baja, la NVIDIA GTX 1050 (integrada en un portátil) que también ha obtenido un resultado excelente, superando sobradamente a la versión CPU. En este caso, el volumen de 300x300x8192 píxeles se procesa en 0.8 segundos, y el de menor resolución en 0.39 segundos.

Las figuras 5.3a y 5.3b muestran el detalle de la ejecución del procesamiento de los datos de OCT en el NVIDIA Visual Profiler. En la figura 5.3a se muestra el solapamiento de las copias de memoria entre la GPU y la CPU en ambas direcciones: *DeviceToHost* y *HostToDevice*. Asimismo, también se muestra el solapamiento de las copias de memoria con el procesamiento de los datos. El solapamiento de las operaciones es posible por el uso de *streams* para el procesamiento de datos de forma independiente y el empleo de memoria *pinned* que permite realizar las copias de datos simultáneamente. Por otra parte, en la figura 5.3b se refleja de forma fiel como funciona la ejecución del programa. Se divide el trabajo en bloques de datos independientes que se procesan en streams de CUDA separados. En primer lugar, cada stream lleva a cabo la copia de datos de la CPU a la GPU. A continuación, procesa los datos y, finalmente, copia el resultado de vuelta a la CPU. Al llevar a cabo la misma operación en varios streams se consigue el solapamiento de las diferentes tareas como se muestra en la figura 5.3a.

Por otra parte, la figura 5.4 muestra el detalle del tiempo de ejecución de cada una de las distintas etapas del procesamiento de los datos de OCT. Aunque en el caso de la implementación CUDA estos tiempos al presentarlos como una gráfica apilada resultan confusos. Esto se debe a que en la implementación CUDA tanto las copias como el procesamiento se producen en paralelo y al sumar todos los tiempos se obtiene un total mayor



(a)



(b)

Figura 5.3: (a) Captura de pantalla de la ejecución del procesamiento de datos de OCT visualizada en el NVIDIA Visual Profiler en la que se aprecia el solapamiento de las copias de memoria y la ejecución de los kernels. (b) Captura de pantalla de la ejecución del procesamiento de datos de OCT visualizada en el NVIDIA Visual Profiler en la que se muestra la ejecución de las diferentes tareas en diferentes streams.

que el tiempo real de ejecución. A pesar de ello se ha decidido mostrar los tiempos totales para poder compararlos con el resto de implementaciones que sí reflejan fielmente el total del tiempo de ejecución cuando se muestran apilados. En el caso de la implementación en CUDA la copia de los datos domina el total, aunque se muestra el tiempo necesario para copiar los datos hacia la GPU y copiar los resultados hacia la CPU. Debido a que el procesado de los datos se divide en bloques más pequeños, ambas copias se producen con un gran solapamiento, reduciendo el peso real a un valor más cercano al tiempo de procesado.

5.3.5 Implementación OpenCL

Finalmente se ha desarrollado una implementación OpenCL, ya que aunque la implementación CUDA consigue un muy buen rendimiento presenta una fuerte limitación debido a que solamente es posible utilizarlo si se cuenta con una tarjeta gráfica NVIDIA. Por el contrario, una implementación OpenCL puede ejecutarse en múltiples plataformas que van desde las CPU hasta las FPGAs pasando por las GPUs.

Para llevar a cabo el cálculo de la transformada rápida de Fourier se ha utilizado la librería *clFFT* [96] desarrollada por AMD y cuyo único requisito es el soporte de OpenCL 1.2. El rendimiento obtenido por esta librería es bastante bueno, a pesar de que ya no se actualiza frecuentemente.

La implementación OpenCL ha utilizado una aproximación muy similar a la utilizada por la versión CUDA, obteniendo resultados ligeramente inferiores en lo que a rendimiento se refiere. La diferencia en rendimiento se debe a que la implementación OpenCL no utiliza memoria *pinned* para las copias entre la GPU y la CPU, reduciendo el ancho de banda entre ambas a la mitad. Además, tampoco se solapa la copia de datos y el procesamiento de los mismos, como sí ocurre en la implementación CUDA. Habría sido posible llevar a cabo dichas optimizaciones pero el objetivo de la implementación OpenCL era mostrar su versatilidad probándola en varias plataformas.

La tabla 5.6 muestra los tiempos de ejecución de las distintas GPUs probadas, entre las que se incluyen las que se probaron con CUDA, pero también se incluye una tarjeta de AMD, y una GPU integrada de un procesador Intel que consigue un rendimiento significativo teniendo en cuenta sus limitaciones. Así, la NVIDIA 1080 Ti tarda 1.57 segundos en procesar el volumen de 300x300x8192 píxeles y 0.85 segundos en procesar el de 300x300x4096. Mientras que la Intel HD 630 emplea 7.5 segundos y 2.75 segundos

Configuración	Resolución (300*300*x)	Tiempo de ejecución (s)	Rendimiento
NVIDIA GTX 1080 Ti	8192	1.57	469.6 MP _x /s
NVIDIA GTX 1080 Ti	4096	0.85	433.7 MP _x /s
NVIDIA GTX 1050	8192	3.1	237.8 MP _x /s
NVIDIA GTX 1050	4096	1.67	220.7 MP _x /s
Intel HD 630	8192	7.5	98.3 MP _x /s
Intel HD 630	4096	2.75	134.1 MP _x /s

Tabla 5.6: Tiempo de ejecución del procesamiento de datos OCT en OpenCL para varias configuraciones y dos resoluciones.

para los volúmenes de 300x300x8192 y 300x300x4096 píxeles, respectivamente.

Además, la figura 5.4 muestra el tiempo empleado por cada una de las etapas del procesamiento para la implementación OpenCL. Al igual que ocurría con la implementación CUDA la mayor parte del tiempo se emplea en la copia de los datos lo que limita el rendimiento. La limitación se hace más acusada al no haber implementado el uso de memoria *pinned*, ni solapar las copias de la CPU a la GPU con las copias en la dirección inversa.

5.4 Sistemas implementando este software

Actualmente el software desarrollado en este capítulo se utiliza para procesar los datos obtenidos por el OCT instalado en el LOUM (la figura 5.2 muestra una imagen del mismo). Y ha resultado especialmente útil para una nueva investigación aún sin publicar en la que se tomaban cinco medidas consecutivas de un mismo sujeto y en la que la calidad de las medidas debía ser máxima. Aunque debido a que el ordenador utilizado para llevar a cabo las medidas de OCT no cuenta con una tarjeta gráfica dedicada, el procesamiento se lleva a cabo en la CPU, reduciendo sensiblemente el rendimiento, pero mejorándolo mucho si se compara con la implementación en LabVIEW. La figura 5.5 muestra una imagen del programa que utiliza las mejoras presentadas en este capítulo. Este programa carga los datos, los procesa con la tecnología que el usuario elija (OpenMP, CUDA u OpenCL) y finalmente presenta una vista previa del resultado y guarda las imágenes en disco. Asimismo, es posible modificar algunos de los parámetros más comúnmente utilizados para modificar los volúmenes en este sistema.

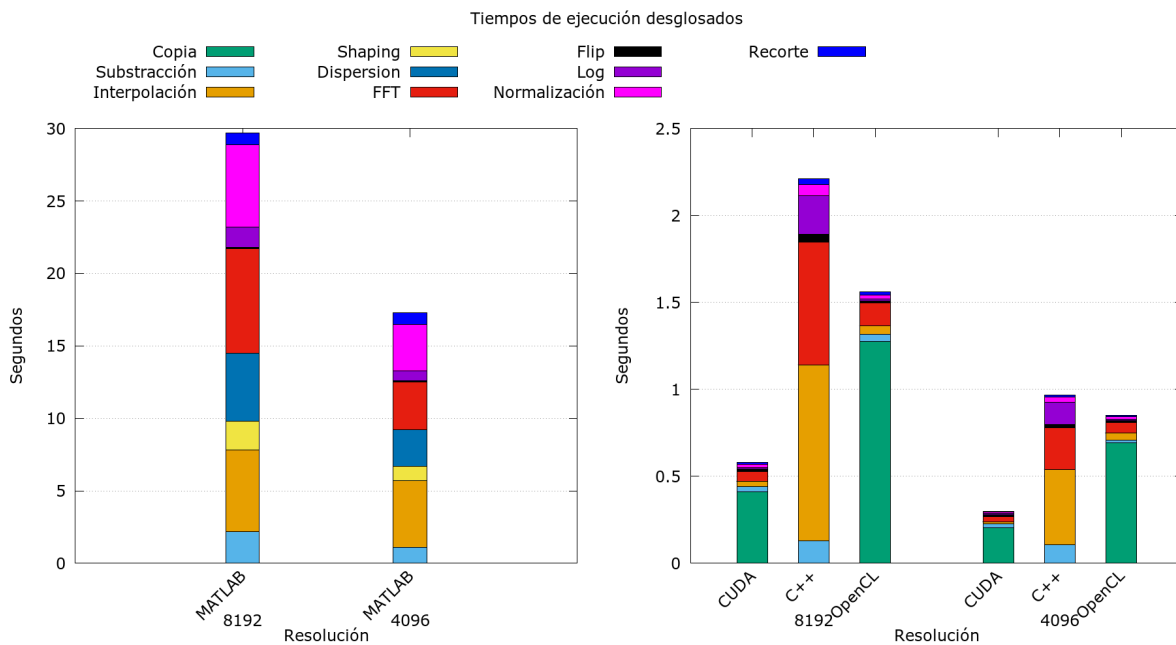


Figura 5.4: Tiempo de ejecución de las distintas operaciones que se llevan a cabo para el procesamiento de los datos de OCT. Los resultados de MATLAB y C++ se han obtenido en un procesador AMD 1800x, mientras que los resultados de CUDA y OpenCL se han obtenido en una GPU NVIDIA 1080 Ti. Los tiempos de la implementación C++ son una estimación debido a limitaciones de la medida. Lo mismo ocurren con los tiempos de la implementación CUDA.^a

^aEl tiempo total de las implementaciones CUDA es menor que el reflejado al mostrar los distintos tiempos apilados debido a que algunas tareas se ejecutan en paralelo mediante el uso de streams.

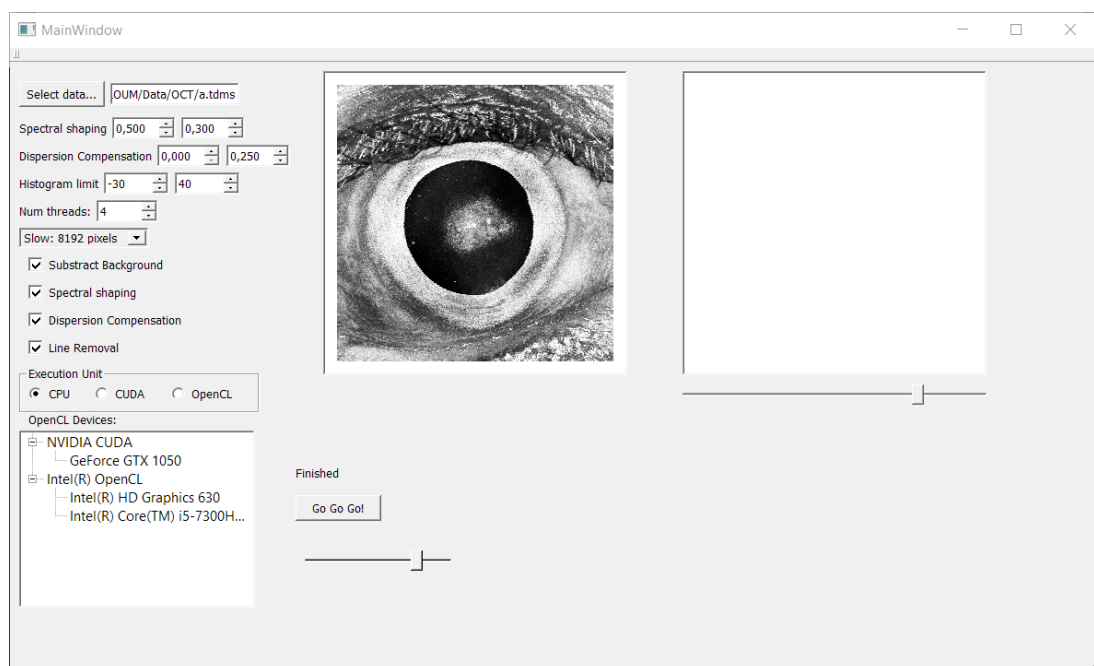


Figura 5.5: Captura de pantalla del programa que se utiliza para procesar los datos de OCT.

6 Conclusiones y perspectivas

6.1 Conclusiones

El procesamiento de imágenes a altas velocidades es un reto debido a la gran cantidad de cómputo y de accesos a memoria que implica. Por otra parte, el procesamiento de imágenes suele presentar un patrón de accesos a memoria muy repetitivo y una ejecución previsible sin muchas divergencias de control en la misma. Esas dos características son claves para que la paralelización orientada a procesadores gráficos obtenga un buen resultado. Por ello, se ha desarrollado una implementación paralela en CUDA que se ejecuta sobre procesadores gráficos obteniendo un gran rendimiento, siendo los resultados más destacables:

1. La paralelización de tres algoritmos de seguimiento pupilar: *Starburst*, Umbralización y Etiquetado, y detección de bordes de Canny y Transformada de Hough.
2. El algoritmo *Starburst* ha obtenido un rendimiento excelente en su versión paralelizada para GPUs, ha conseguido una velocidad de 2049 imágenes/s para el ajuste a un círculo y 1610 imágenes/s para el ajuste a una elipse.
3. Se ha comparado el rendimiento obtenido con una implementación secuencial ejecutada sobre una CPU, constatando una gran mejora con una aceleración superior a 45x para el algoritmo *Starburst*.
4. Se ha caracterizado la precisión de los algoritmos implementados con un conjunto de imágenes, en la que el algoritmo *Starburst* ha obtenido una tasa de éxito superior al 96% para un margen de error del 10%, en la configuración del filtro de Gauss de tamaño 5 y un tamaño de 19 para el filtro de los reflejos corneales.
5. Se han realizado ajustes tanto a un círculo como a una elipse, obteniendo una mayor precisión el ajuste a una elipse debido a que es más fiel a la forma de las pupilas del

ojo humano. En concreto, el ajuste a una elipse consigue una tasa de éxito del 90% con un margen de error del 5% frente a un 86% del ajuste a un círculo.

Los hitos más destacables referentes a las Gafas de Corrección Automática de la Presbicia son:

1. Se ha llevado a cabo una comparativa de rendimiento y eficiencia energética entre varias GPUs de SoCs móviles, dos FPGAs y una GPU de sobremesa. Obteniendo como resultado un rendimiento de 24 imágenes/s para el SoC Samsung Exynos 8890 y una eficiencia energética de 0.016 J/imagen (para imágenes de 320x240 píxeles). Mientras que la FPGA Alveo u250 ha obtenido un rendimiento de 45 imágenes/s y 0.43 J/imagen.
2. Las Gafas de Corrección Automática de la Presbicia son totalmente portátiles, además, funcionan de forma autónoma, dinámica, automática y en tiempo real.
3. Se ha validado el correcto funcionamiento de las Gafas de Corrección Automática de la Presbicia llevando a cabo dos experimentos con sujetos presbítas. En el primer experimento se obtuvo una mejora de la agudeza visual de 0.27 en escala LogMAR entre la visión normal sin las gafas y la visión mejorada con las gafas. Mientras que en el segundo experimento se comparó el rendimiento de las gafas auto-acomodativas con dos correcciones estáticas, una de cerca y otra de lejos. En ambos casos se consiguió igualar el resultado en la distancia óptima de las correcciones estáticas y mejorarlo en el resto de distancias.
4. Las Gafas de Corrección Automática de la Presbicia se han patentado, protegiendo los derechos intelectuales de las mismas.

Se han implementado y paralelizado varios algoritmos de procesamiento de imágenes de Hartmann-Shack. Para la paralelización se ha utilizado CUDA ejecutado sobre GPUs, obteniendo una mejora significativa sobre la implementación secuencial ejecutada sobre una CPU. Los resultados más importantes respecto a los sensores de Hartmann-Shack son:

1. La implementación en CUDA del algoritmo PaPyCS alcanza una velocidad de 175 imágenes/s con una resolución de 2560x2560 píxeles y 2000 microlentes. Obteniendo además una aceleración de 101x respecto a la versión secuencial.
-

2. Igualmente, se ha llevado a cabo un estudio de la precisión de los algoritmos implementados haciendo uso de un conjunto de imágenes representativo de las medidas más comunes. En dicho estudio se ha podido comprobar la superioridad del algoritmo PaPyCS respecto a los otros, tanto por su mayor precisión como por su robustez. Dicho algoritmo ha logrado procesar correctamente hasta el 90% de imágenes con un RMS de 5.
3. Además, se ha creado un algoritmo original de seguimiento pupilar sobre imágenes de Hartmann-Shack que también se ha paralelizado y optimizado en CUDA.

La aplicación de captura de datos del sistema OCT está programada en LabView, por lo que la versión inicial de procesamiento también estaba programada en LabView. Posteriormente, se desarrolló una versión MATLAB, y después, varias versiones de alto rendimiento: C++, CUDA y OpenCL. Asimismo, en las versiones de alto rendimiento se han utilizado tres librerías para calcular la FFT: FFTW, cuFFT y clFFT. Los principales hitos conseguidos en la paralelización y optimización del procesamiento de datos de OCT son:

1. La implementación en C++ ha sido optimizada y paralelizada hasta obtener una muy alta velocidad de 2.6 segundos por volumen de 300x300x8192 píxeles, o lo que es lo mismo, 275 MPx/s. Esto supone una aceleración de 11.4x sobre la versión programada en MATLAB.
 2. La implementación en CUDA obtiene un rendimiento cercano al máximo posible, dado que prácticamente consigue saturar el bus PCIe que une la GPU y la CPU. Esto se consigue mediante el solapamiento tanto de las copias en ambas direcciones (*DeviceToHost* y *HostToDevice*) como del cómputo. El tiempo de procesamiento es de 0.3 segundos para el volumen de 300x300x8192 píxeles, lo cual equivale a 2.46 GPx/s.
 3. También se ha programado una implementación OpenCL muy versátil que puede ejecutarse en cualquier plataforma que soporte OpenCL 1.2 o superior. El rendimiento obtenido por esta versión ha sido bueno, aunque algo inferior al obtenido por la versión CUDA debido a la menor cantidad de optimizaciones aplicadas en este caso. Así, esta implementación emplea 1.57 segundos en procesar el volumen de 300x300x8192 píxeles, lo que equivale a 469 MPx/s. Se ha probado en varias GPUs de distintos vendedores con resultados comparables en función de las capacidades de cada dispositivo.
-

6.2 Contribuciones de la Tesis

En el transcurso de esta Tesis se han obtenido notables resultados que han conducido a generar una serie de importantes contribuciones. Éstas se detallan a continuación:

- **Se han paralelizado y caracterizado tres algoritmos de seguimiento pupilar.** Se han implementado tanto versiones secuenciales utilizando C++ como versiones paralelas utilizando CUDA. Las versiones paralelas han sido optimizadas para maximizar el rendimiento de las mismas. Esas implementaciones han sido caracterizadas utilizando una amplia biblioteca de imágenes obteniendo el rendimiento y la precisión de los tres algoritmos.
 - Como resultado de la paralelización y caracterización de los algoritmos de seguimiento pupila **se ha generado una librería de alto rendimiento de seguimiento pupilar.** Esa librería se utiliza en diversos sistemas ópticos del LOUM.
 - **Se ha creado un prototipo de Gafas de corrección automática de la Presbicia.** Este prototipo realiza un seguimiento pupilar en tiempo real utilizando dos micro cámaras y un teléfono inteligente. Para realizar el seguimiento pupilar se utiliza un implementación OpenCL que se ejecuta en el procesador gráfico del teléfono inteligente. Después, a partir de los datos obtenidos aplica la corrección adecuada a las lentes optoelectrónicas. Todo ello se lleva a cabo en milisegundos de forma que el sujeto no percibe la latencia del sistema.
 - **Se ha implementado una versión para FPGAs del seguimiento pupilar.** Por otra parte, se ha comparado el rendimiento y el consumo de esa implementación con el de la implementación OpenCL y la implementación CUDA. Para llevar a cabo las comparaciones de rendimiento y consumo se han utilizado dos FPGAs, dos teléfonos inteligentes y una GPU de sobremesa.
 - **El prototipo de Gafas de corrección automática de la Presbicia ha sido valido** con pacientes presbitas. Se han llevado a cabo dos estudios con ocho sujetos cada uno. En el primer experimento se ha pedido a los sujetos que leyeran un texto con diferentes tamaños de letra situado a 35cm de sus ojos sin y con las gafas optoelectrónicas. Todos ellos mostraron una mejora significativa cuando usaron las Gafas de corrección automática de la Presbicia. En el segundo experimento se han com-
-

parado dos correcciones estáticas para cerca y lejos con las gafas optoelectrónicas, obteniendo estas últimas una mejora significativa sobre las correcciones estáticas.

- El prototipo de Gafas de corrección automática de la Presbicia ha sido **patentado**, protegiendo la propiedad intelectual generada en el desarrollo de esta Tesis [98].
 - **Se ha desarrollado un algoritmo de seguimiento pupilar para imágenes de sensores Hartmann-Shack.** Éste permite prescindir de la necesidad de tomar imágenes del ojo en infrarrojo en determinadas circunstancias. Asimismo, la paralelización en GPU que se ha llevado a cabo del mismo proporciona una implementación de alto rendimiento para sistemas que requieren funcionar en tiempo real.
 - **Se han paralelizado y caracterizado tres algoritmos para procesamiento de imágenes de sensores Hartmann-Shack.** Se han implementado versiones secuenciales en C++ y versiones paralelas en CUDA de los tres algoritmos. Además, se han llevado a cabo pruebas de rendimiento y precisión de los mismos para determinar su comportamiento con diferentes configuraciones de sensores Hartmann-Shack. Para obtener el máximo rendimiento se han aplicado numerosas optimizaciones en las implementaciones CUDA.
 - Con la implementación para procesamiento de imágenes de **sensores Hartmann-Shack** que obtuvo la mejor combinación de rendimiento y precisión **se ha creado una librería de alto rendimiento y alta precisión.** Esta librería incluye tanto una versión secuencial en C++ como una versión paralela en CUDA para GPUs.
 - **Se ha paralelizado un algoritmo de procesamiento de datos de OCT utilizando CUDA.** Dicha implementación obtiene un rendimiento superior a la velocidad de captura, permitiendo el procesamiento en **tiempo real**. Si bien, actualmente sólo se utiliza para acelerar el posprocesado de los datos.
 - Asimismo, **se ha paralelizado y optimizado el procesado de datos de OCT usando C++ y OpenMP.** Por otra parte, también se ha desarrollado una implementación usando **OpenCL**. Las tres implementaciones han obtenido un **buen rendimiento**, si bien este ha sido **inferior al obtenido por la implementación en CUDA**.
-

Las contribuciones anteriores se han materializado en las siguientes publicaciones:

- J. Mompeán, J. L. Aragón, P. Prieto y P. Artal. “GPU-Accelerated High-Speed Eye Pupil Tracking System”. En: *Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on Computer Architecture and High Performance Computing*. IEEE. Florianópolis, Brazil, 2015, págs. 17-24
 - J. Mompeán, J. L. Aragón, P. M. Prieto y P. Artal. “Robust High-speed Eye Pupil Tracking System on GPUs”. En: *XXVII Jornadas de Paralelismo, 2016 Jornadas SARTECO*. SARTECO. Salamanca, España, 2016
 - J. Mompeán, S. Manzanera, J. L. Aragón y P. Artal. “Presbyopia correction with optoelectronic lenses driven by pupil size”. En: *Investigative Ophthalmology & Visual Science* 57.12 (2016)
 - J. Mompeán, J. L. Aragón, P. M. Prieto y P. Artal. “High-speed Wavefront Sensor Processing with Graphics Accelerators”. En: *XI Workshop on Adaptive Optics for Industry and Medicine, 2018*. Laboratorio de Óptica de la Universidad de Murcia. Murcia, España, 2018
 - J. Mompeán, J. L. Aragón, P. M. Prieto y P. Artal. “Design of an accurate and high-speed binocular pupil tracking system based on GPGPUs”. En: *The Journal of Supercomputing* 74.5 (mayo de 2018), págs. 1836-1862. ISSN: 1573-0484. DOI: 10.1007/s11227-017-2193-5. URL: <https://doi.org/10.1007/s11227-017-2193-5>
 - J. Mompeán, J. L. Aragón, P. M. Prieto y P. Artal. “Cálculo de Mapas de Frente de Onda mediante GPUs para Aplicaciones de Óptica Adaptativa en Tiempo Real”. En: *XXVII Jornadas de Paralelismo*. Vol. 1. 1. SARTECO. 2017, págs. 107-112
 - J. Mompeán, J. L. Aragón, P. M. Prieto y P. Artal. “GPU-based processing of Hartmann-Shack images for accurate and high-speed ocular wavefront sensing”. En: *Future Generation Computer Systems* 91 (2019), págs. 177-190
 - J. Mompeán, J. L. Aragón y P. Artal. “Automatic and Dynamic presbyopia correction”. En: *33rd International Pupil Colloquium, 2019*. Laboratorio de Óptica de la Universidad de Murcia. Murcia, España, 2019
-

Además de las anteriores publicaciones, las diferentes propuestas y herramientas que han surgido gracias a esta Tesis se han (o están) utilizado en otros experimentos realizados en el seno del Laboratorio de Óptica de la UMU y han dado a las siguientes publicaciones:

- H. S. Ginis, A. Pennos, J. Mompeán y P. Artal. “Macular pigment density estimated with Fourier-domain fundus reflectometry”. En: *Investigative Ophthalmology & Visual Science* 57.12 (2016), págs. 5097-5097
- I. Grulkowski, L. Cwiklinski, S. Manzanera, J. Mompeán, P. Artal y M. D. Wojtkowski. “Long-Depth-Range Swept-Source OCT Instrument for Imaging Crystalline Lens Opacities”. En: *Investigative Ophthalmology & Visual Science* 57.12 (2016), págs. 437-437
- C. Robles, P. M. Prieto, J. Mompeán, J. Tabernero y P. Artal. “Crystalline lens position dynamics during mixed saccadic/convergence eye movements”. En: *Investigative Ophthalmology & Visual Science* 58.8 (2017), págs. 2059-2059
- I. Grulkowski, S. Manzanera, L. Cwiklinski, J. Mompeán, A. de Castro, J. M. Marín y P. Artal. “Volumetric macro- and micro-scale assessment of crystalline lens opacities in cataract patients using long-depth-range swept source optical coherence tomography”. En: *Biomed. Opt. Express* 9.8 (ago. de 2018), págs. 3821-3833. DOI: 10.1364/BOE.9.003821. URL: <http://www.osapublishing.org/boe/abstract.cfm?URI=boe-9-8-3821>
- A. de Castro, A. Benito, S. Manzanera, J. Mompeán, B. Cañizares, D. Martínez, J. M. Marín, I. Grulkowski y P. Artal. “Three-Dimensional Cataract Crystalline Lens Imaging With Swept-Source Optical Coherence Tomography Cataract Crystalline Lens Imaging With SS-OCT”. En: *Investigative Ophthalmology & Visual Science* 59.2 (feb. de 2018), págs. 897-903. ISSN: 1552-5783. DOI: 10.1167/iovs.17-23596. eprint: https://iovs.arvojournals.org/arvo/content/_public/journal/iovs/936739/i1552-5783-59-2-897.pdf. URL: <https://doi.org/10.1167/iovs.17-23596>
- I. Grulkowski, A. Rodriguez-Aramendia, D. Ruminski, S. Manzanera, Y. Chen, J. Mompeán, F. D. Douton, J. Pujol, J. Sebag y P. Artal. “Vitreous body imaging with long-range swept-source optical coherence tomography for detection of opacities”. En: *Investigative Ophthalmology & Visual Science* 60.9 (2019), págs. 1291-1291

-
- I. Grulkowski, A. Rodríguez-Aramendía, S. Manzanera, Y. Chen, J. Mompeán, F. Díaz-Doutón, J. Pujol, J. Sebag y P. Artal. “In vivo imaging of vitreous opacities with full-eye-length SS-OCT (Conference Presentation)”. En: vol. 10858. 2019. DOI: 10.1117/12.2509474. URL: <https://doi.org/10.1117/12.2509474>
 - A. Rodríguez-Aramendía, I. Grulkowski, A. Jiménez Villar, S. Manzanera, Y. Chen, J. Mompeán, F. Díaz-Doutón, J. Pujol, J. L. Güell y P. Artal. “Optimization of a SS-OCT with a focus tunable lens for enhanced visualization of ocular opacities”. En: vol. 10867. 2019. DOI: 10.1117/12.2508640. URL: <https://doi.org/10.1117/12.2508640>
 - S. Manzanera, I. Delgado, J. Mompeán y P. Artal. “Pupil diameter under normal viewing condition”. En: *33rd International Pupil Colloquium, 2019*. Laboratorio de Óptica de la Universidad de Murcia. Murcia, España, 2019
 - D. Christaras, H. Ginis, A. Pennos, J. Mompeán y P. Artal. “An optical method to assess the macular pigment density”. En: *Investigative Ophthalmology & Visual Science* 60.9 (2019), págs. 6094-6094
 - D. Christaras, H. Ginis, A. Pennos, J. Mompean y P. Artal. “Objective method for measuring the macular pigment optical density in the eye”. En: *Biomedical optics express* 10.7 (2019), págs. 3572-3583
 - A. Gupta, D. Ruminski, A. J. Villar, R. D. Toledo, S. Manzanera, S. Panezai, J. Mompean, P. Artal e I. Grulkowski. “In vivo SS-OCT imaging of crystalline lens sutures”. En: *Biomed. Opt. Express* 11.10 (oct. de 2020), págs. 5388-5400. DOI: 10.1364/BOE.401254. URL: <http://www.osapublishing.org/boe/abstract.cfm?URI=boe-11-10-5388>
 - D. Christaras, J. Mompeán, P. Artal y H. Ginis. “A portable optical instrument for the measurement of macular pigment optical density”. En: *Investigative Ophthalmology & Visual Science* 61.9 (2020), PB00128-PB00128
 - C. Robles, J. Mompean, P. Prieto y P. Artal. “Objective Eye’s Refraction from Pupil-tracking”. En: *Investigative Ophthalmology & Visual Science* 61.7 (2020), págs. 535-535
-

- A. Gkiala, D. Christaras, J. Mompean, N. Symidalas, A. Livir-Rallatos, P. Artal y H. S. Ginis. “Macular pigment optical density in a healthy population measured with a new optical instrument”. En: *Investigative Ophthalmology & Visual Science* 61.7 (2020), págs. 1856-1856

6.3 Perspectivas

El desarrollo de nuevos dispositivos portátiles con altos requerimientos de cómputo sigue siendo un campo en el que hay muchas oportunidades de investigación. Y en lo relacionado con las aplicaciones de la óptica el desarrollo de dispositivos de medida portátiles aún es un área incipiente. Aunque en ese caso supone dos retos, por un lado, la miniaturización de la óptica y, por otro, la implementación de los algoritmos de procesado en sistemas portátiles que funcionen con una batería.

El seguimiento pupilar, en el que se ha trabajado ampliamente en esta Tesis, puede ser una herramienta de gran utilidad para muchos dispositivos de medición portátiles. Independientemente de las medidas que se lleven a cabo, todo sistema que pretenda tomar una medición de algún parámetro del ojo va a requerir estar correctamente alineado con el mismo. Es por esto que un sistema de seguimiento pupilar puede controlar el alineamiento, aunque sea manual, y funcionar como un disparador de la medida cuando se detecte que la posición es la correcta.

Además, en esta Tesis se ha presentado una solución que utiliza un teléfono móvil para llevar a cabo el procesamiento. Pero el uso de otras plataformas podría mejorar la integración del dispositivo de procesamiento en los dispositivos de medida. Dispositivos como la Raspberry Pi, que entra dentro de la categoría de Single Board Computer (SBC), presentan un mejor equilibrio entre potencia, autonomía y versatilidad que los teléfonos móviles para este tipo de desarrollos. Principalmente por su capacidad para interactuar con motores, sensores y una infinidad de elementos electrónicos necesarios en cualquier tipo de dispositivo de medición portátil. Evidentemente, los SBC presentan limitaciones en cuanto a su potencia, e incluso son algo voluminosos para determinados dispositivos. Pero conseguir un tamaño más reducido manteniendo la versatilidad de los SBC requeriría de costosos desarrollos personalizados.

De hecho, las Gafas de Corrección Automática de la Presbicia podrían mejorar la integración de los componentes de forma significativa si se reemplazara el teléfono inteligente

que se utiliza actualmente por un dispositivo de menor tamaño como una Raspberry Pi Zero. Sin embargo, este tipo de dispositivos suelen contar con una capacidad de cómputo limitada, así como una conectividad igualmente reducida. A pesar de ello, si se consigue optimizar lo suficiente el procesamiento para esa plataforma los beneficios de integración serían notables. Se podrían eliminar todos los cables externos y construir un dispositivo de un tamaño más reducido y cómodo. Del mismo modo, el uso de una FPGA de un tamaño y consumo contenidos pero con suficiente potencia como para alcanzar los requerimientos de procesamiento supondría una gran mejora en la integración del sistema.

En lo que respecta al sensor Hartmann-Shack el trabajo es continuo, la librería desarrollada en esta Tesis recibe mejoras frecuentemente, ya sea para arreglar bugs, añadir nueva funcionalidad o mejorar el rendimiento. En este caso sería especialmente útil el desarrollo de una implementación OpenCL o SYCL. La necesidad de implementaciones de alto rendimiento que se puedan ejecutar en cualquier plataforma es creciente debido al amplio uso que se hace de esta librería y la heterogeneidad de los dispositivos en los que se requiere su uso.

Por otra parte, el desarrollo de un sistema de bucle cerrado utilizando la implementación de alto rendimiento del procesamiento de imágenes de sensores Hartmann-Shack es una aplicación inmediata. Los bucles cerrados con sensores de Hartmann-Shack son sistemas que ya se han desarrollado en el pasado [111], pero no dejan de ser un reto hoy en día. En este caso, el objetivo sería hacerlo a la máxima velocidad posible para poder reaccionar con una reducida latencia a los cambios del ojo. Al mismo tiempo, una elevada resolución de la cámara y un algoritmo muy robusto deben permitir obtener unos resultados precisos y fiables.

Los sistemas de OCT capturan grandes cantidades de datos, y eso limita la velocidad de adquisición en gran medida. Si bien los avances en los sistemas de adquisición son constantes, su velocidad sigue siendo menor que la capacidad de procesamiento. Es por ello que realizar una visualización en tiempo real de las medidas suele suponer un mayor reto para el sistema de adquisición que para el sistema de procesamiento. Sin embargo, la visualización en tiempo real de las medidas es útil en diversos ámbitos como se ha demostrado en algunas publicaciones [94, 93, 89]. Por lo tanto, existe una necesidad de conseguir visualizaciones en tiempo real con la máxima calidad posible.

Una idea en la que se está empezando a trabajar es la adquisición de datos a una menor resolución y, mediante la combinación de una interpolación temporal y espacial, generar unos resultados de mayor resolución. De esta forma se conseguiría aumentar la velocidad

de captura (al reducir la resolución de captura) mientras se mantiene la resolución final gracias a la interpolación. Así, utilizando el sistema OCT del LOUM [82] se podría usar una resolución baja, de 50x50x8192 píxeles, que permitiría alcanzar los 40 volúmenes por segundo, cantidad muy superior a las 0.55 volúmenes por segundo obtenidos con la resolución actual.

Bibliografía

- [1] M. Harris. *A brief history of GPGPU*. <http://web.archive.org/web/20190523155719/http://cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf>. Accedido: 23-05-2019.
- [2] J. N. England. “A system for interactive modeling of physical curved surface objects”. En: *ACM SIGGRAPH Computer Graphics*. Vol. 12. 3. ACM. 1978, págs. 336-340.
- [3] J. Nickolls et al. “Scalable Parallel Programming with CUDA”. En: *Queue* 6.2 (mar. de 2008), págs. 40-53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. URL: <http://doi.acm.org/10.1145/1365490.1365500>.
- [4] J. E. Stone, D. Gohara y G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. En: *Computing in Science Engineering* 12.3 (mayo de 2010), págs. 66-73. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.69.
- [5] M. J. Flynn. “Some computer organizations and their effectiveness”. En: *IEEE transactions on computers* 100.9 (1972), págs. 948-960.
- [6] *CUDA C++ Programming Guida*. Inf. téc. NVIDIA, ago. de 2020. URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
- [7] C. Robles, P. Prieto y P. Artal. “Pupil-tracking-based determination of visual acuity”. En: *Investigative Ophthalmology & Visual Science* 60.9 (2019), págs. 5906-5906.
- [8] A. Glasser y M. C. Campbell. “Presbyopia and the optical changes in the human crystalline lens with age”. En: *Vision research* 38.2 (1998), págs. 209-229.
- [9] M. Carillet et al. “Wavefront sensing: from historical roots to the state-of-the-art”. En: *European Astronomical Society Publications Series* 22 (2006), págs. 165-185.
- [10] R. Duffner. *The adaptive optics revolution: a history*. University of New Mexico Press, 2009.
- [11] Z. von F. “Beugungstheorie des schneidenverfahrens und seiner verbesserten form, der phasenkontrastmethode”. En: *Physica* 1.7-12 (1934), págs. 689-704.

-
- [12] K. M. Williams et al. "Increasing prevalence of myopia in Europe and the impact of education". En: *Ophthalmology* 122.7 (2015), págs. 1489-1497.
 - [13] V. Molebny. "Wavefront Sensors". En: *Handbook of Visual Optics, Vol. II: Instrumentation and Vision Correction*. Ed. por P. Artal. CRC Press, 2017, págs. 17-36.
 - [14] E. Chirre, P. M. Prieto y P. Artal. "Binocular open-view instrument to measure aberrations and pupillary dynamics". En: *Optics letters* 39.16 (2014), págs. 4773-4775.
 - [15] J. Porter et al. *Adaptive optics for vision science: principles, practices, design and applications*. Vol. 171. John Wiley & Sons, 2006.
 - [16] C. Cánovas et al. "Hybrid adaptive-optics visual simulator". En: *Optics letters* 35.2 (2010), págs. 196-198.
 - [17] R. V. Shack y B. C. Platt. "Production and use of a lenticular Hartmann screen". En: *Journal of the Optical Society of America* 61.5 (1971), págs. 656-660.
 - [18] J. Hartmann. "Bemerkungen über den Bau und die Justierung von Spectrographen". En: *Zf Instrumentenk* 20.4 (1900).
 - [19] D. Huang et al. "Optical coherence tomography". En: *science* 254.5035 (1991), págs. 1178-1181.
 - [20] J. B. Mulligan. "A GPU-accelerated Software Eye Tracking System". En: *Proc. of the ACM Symp. on Eye Tracking Research and Applications*. Santa Barbara, CA, USA, 2012, págs. 265-268. ISBN: 978-1-4503-1221-9. DOI: 10.1145/2168556.2168612.
 - [21] I. Borovikov. "GPU-acceleration for Surgical Eye Imaging". En: *Proc. of the 4th SIAM Conference on Mathematics for Industry (MI09)*. San Francisco, CA, USA, 2009.
 - [22] J.-P. D. Plessis y P. Blignaut. "Performance of a simple remote video-based eye tracker with GPU acceleration". En: *Journal of Eye Movement Research* 9.4 (2016). ISSN: 1995-8692. URL: <https://bop.unibe.ch/index.php/JEMR/article/view/2476>.
 - [23] D. Li, D. Winfield y D. J. Parkhurst. "Starburst: A Hybrid Algorithm for Video-based Eye Tracking Combining Feature-based and Model-based Approaches". En: *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) - Workshops*. San Diego, CA, USA, 2005. ISBN: 0-7695-2372-2-3.
-

-
- [24] M. A. Fischler y R. C. Bolles. “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography”. En: *Communications of the ACM* 24.6 (1981), págs. 381-395.
 - [25] H. Mehrabian y P. Hashemi-Tari. “Pupil boundary detection for iris recognition using graph cuts”. En: *Proc. of the Int’l Conf. on Image and Vision Computing New Zealand (IVCNZ)*. 2007, págs. 77-82.
 - [26] A. A. Jarjes, K. Wang y G. J. Mohammed. “GVF snake-based method for accurate pupil contour detection”. En: *Information Technology Journal* 9.8 (2010), págs. 1653-1658.
 - [27] J. Canny. “A computational approach to edge detection”. En: *IEEE Trans. on Pattern Analysis and Machine Intelligence* 8.6 (1986), págs. 679-698.
 - [28] D. H. Ballard. “Generalizing the Hough transform to detect arbitrary shapes”. En: *Pattern Recognition* 13.2 (1981), págs. 111-122.
 - [29] L. Masek et al. “Recognition of human iris patterns for biometric identification”. Tesis de mtría. University of Western Australia, 2003.
 - [30] D. M. Rankin et al. “Dynamic iris biometry: a technique for enhanced identification”. En: *BMC research notes* 3.1 (2010), pág. 182.
 - [31] M. Ohtani e Y. Ebisawa. “Eye-gaze detection based on the pupil detection technique using two light sources and the image difference method”. En: *Engineering in Medicine and Biology Society, 1995., IEEE 17th Annual Conference*. Vol. 2. IEEE. 1995, págs. 1623-1624.
 - [32] C. Schwarz et al. “Binocular adaptive optics vision analyzer with full control over the complex pupil functions”. En: *OSA Optics Letters* 36.24 (2011), págs. 4779-4781.
 - [33] S. Manzanera et al. “Location of Achromatizing Pupil Position and First Purkinje Reflection in a Normal Population”. En: *Investigative Ophthalmology & Visual Science* 56.2 (2015), págs. 962-966.
 - [34] W. Aydi, N. Masmoudi y L. Kamoun. “New corneal reflection removal method used in iris recognition system”. En: *World Academy of Science, Engineering and Technology* 5.5 (2011), págs. 898-902.
-

-
- [35] G. Taubin. "Estimation of planar curves, surfaces, and nonplanar space curves defined by implicit equations with applications to edge and range image segmentation". En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13.11 (1991), págs. 1115-1138.
- [36] J. Mompeán et al. "Design of an accurate and high-speed binocular pupil tracking system based on GPGPUs". En: *The Journal of Supercomputing* 74.5 (mayo de 2018), págs. 1836-1862. ISSN: 1573-0484. DOI: 10.1007/s11227-017-2193-5. URL: <https://doi.org/10.1007/s11227-017-2193-5>.
- [37] P. Chen et al. "Block-run-based connected component labelling algorithm for GPGPU using shared memory". En: *IET Electronics Letters* 47.24 (2011), págs. 1309-1311.
- [38] P. Prieto et al. "Impact of Stiles-Crawford peak decentration with small apertures". En: *Investigative Ophthalmology & Visual Science* 57.12 (2016), págs. 3120-3120.
- [39] W. S. Stiles y B. Crawford. "The luminous efficiency of rays entering the eye pupil at different points". En: *Proceedings of the Royal Society of London. Series B, Containing Papers of a Biological Character* 112.778 (1933), págs. 428-450.
- [40] C. Robles et al. "Crystalline lens position dynamics during mixed saccadic/convergence eye movements". En: *Investigative Ophthalmology & Visual Science* 58.8 (2017), págs. 2059-2059.
- [41] S. Manzanera et al. "Pupil diameter under normal viewing condition". En: *33rd International Pupil Colloquium, 2019*. Laboratorio de Óptica de la Universidad de Murcia. Murcia, España, 2019.
- [42] B. A. Holden et al. "Global prevalence of myopia and high myopia and temporal trends from 2000 through 2050". En: *Ophthalmology* 123.5 (2016), págs. 1036-1042.
- [43] B. Sterner, M. Gellerstedt y A. Sjöström. "The amplitude of accommodation in 6-10-year-old children-not as good as expected!" En: *Ophthalmic and Physiological Optics* 24.3 (2004), págs. 246-251.
- [44] A. Duane. "Studies in monocular and binocular accommodation with their clinical applications". En: *American Journal of Ophthalmology* 5.11 (1922), págs. 865-877.
- [45] K. R. Heys, S. L. Cram y R. J. Truscott. "Massive increase in the stiffness of the human lens nucleus with age: the basis for presbyopia?" En: (2004).
-

-
- [46] S. Jain, I. Arora y D. T. Azar. "Success of monovision in presbyopes: review of the literature and potential applications to refractive surgery". En: *Survey of ophthalmology* 40.6 (1996), págs. 491-499.
- [47] O. Seyeddain et al. "Refractive surgical correction of presbyopia with the AcuFocus small aperture corneal inlay: two-year follow-up". En: *Journal of Refractive Surgery* 26.10 (2010), págs. 707-715.
- [48] M. Blum et al. "Optotune focus tunable lenses and laser speckle reduction based on electroactive polymers". En: *MOEMS and Miniaturized Systems XI*. Ed. por H. Schenk, W. Piyawattanametha y W. Noell. Vol. 8252. International Society for Optics y Photonics. SPIE, 2012, págs. 43-53. DOI: 10.1117/12.902631. URL: <https://doi.org/10.1117/12.902631>.
- [49] Y.-H. Lin y H.-S. Chen. "Electrically tunable-focusing and polarizer-free liquid crystal lenses for ophthalmic applications". En: *Optics Express* 21.8 (2013), págs. 9428-9436.
- [50] N. Hasan et al. "Tunable-focus lens for adaptive eyeglasses". En: *Optics express* 25.2 (2017), págs. 1221-1233.
- [51] J. Jarosz et al. "Experimental optical analysis of an original presbyopia-correcting variable focus lens". En: *Investigative Ophthalmology & Visual Science* 59.9 (2018), págs. 255-255.
- [52] J. Jarosz et al. "Adaptive eyeglasses for presbyopia correction: an original variable-focus technology". En: *Opt. Express* 27.8 (abr. de 2019), págs. 10533-10552. DOI: 10.1364/OE.27.010533. URL: <http://www.opticsexpress.org/abstract.cfm?URI=oe-27-8-10533>.
- [53] J. C. F. Diaz et al. "LiDAR Remote Sensing". En: *Handbook of Satellite Applications*. Ed. por J. N. Pelton, S. Madry y S. Camacho-Lara. Cham: Springer International Publishing, 2017, págs. 929-980. ISBN: 978-3-319-23386-4. DOI: 10.1007/978-3-319-23386-4_44. URL: https://doi.org/10.1007/978-3-319-23386-4_44.
- [54] J. M. Bueno et al. "Wavefront correction in two-photon microscopy with a multi-actuator adaptive lens". En: *Optics express* 26.11 (2018), págs. 14278-14287.
-

-
- [55] J. Yiu. *ARM Cortex-M for Beginners. An Overview of the ARM Cortex-M processor family and comparison*. Inf. téc. ARM, mar. de 2017, pág. 28. URL: https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2057-00-00-00-01-28-35/Cortex_2D00_M-for-Beginners-_2D00_-2017_5F00_EN_5F00_v2.pdf.
- [56] N. Padmanaban, R. Konrad y G. Wetzstein. “Autofocals: gaze-contingent eyeglasses for presbyopes”. En: *ACM SIGGRAPH 2018 Emerging Technologies*. ACM. 2018, pág. 3.
- [57] A. S. Mastrangelo et al. “A Low-Profile Digital Eye-Tracking Oculometer for Smart Eyeglasses”. En: *2018 11th International Conference on Human System Interaction (HSI)*. IEEE. 2018, págs. 506-512.
- [58] Qualcomm Technologies, Inc. *Treppn Power Profiler*. Ver. 6.2. 3 de mar. de 2016. URL: <https://developer.qualcomm.com/software/treppn-power-profiler>.
- [59] J. Mompeán et al. “Presbyopia correction with optoelectronic lenses driven by pupil size”. En: *Investigative Ophthalmology & Visual Science* 57.12 (2016).
- [60] P. Artal. “Adjustable power lenses for vision correction”. En: *Investigative Ophthalmology & Visual Science* 59.9 (2018), págs. 1992-1992.
- [61] J. Mompeán, J. L. Aragón y P. Artal. “Automatic and Dynamic presbyopia correction”. En: *33rd International Pupil Colloquium, 2019*. Laboratorio de Óptica de la Universidad de Murcia. Murcia, España, 2019.
- [62] F. Campbell y G. Westheimer. “Dynamics of accommodation responses of the human eye”. En: *The Journal of physiology* 151.2 (1960), págs. 285-295.
- [63] H. R. Zohouri et al. “Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs”. En: *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2016, págs. 409-420.
- [64] D. R. Sanders y M. L. Sanders. “Near visual acuity for everyday activities with accommodative and monofocal intraocular lenses”. En: *Journal of Refractive Surgery* 23.8 (2007), págs. 747-751.
- [65] J. Mocci et al. “A multi-platform CPU-based architecture for cost-effective adaptive optics systems”. En: (ene. de 2018).
-

-
- [66] Y. Yu e Y. Zhang. “Dual-thread parallel control strategy for ophthalmic adaptive optics”. En: *Chinese Optics Letters* 12.12 (2014), págs. 121202-121202.
- [67] Y. Yu et al. “High-speed adaptive optics for imaging of the living human eye”. En: *Optics express* 23.18 (2015), págs. 23035-23052.
- [68] J. G. Marichal-Hernández, J. M. Rodríguez-Ramos y F. Rosa. “Recuperación de fase de frente de onda atmosférico usando hardware gráfico”. En: *XI Congreso Nacional de Teledetección*. Tenerife, Spain, sep. de 2005, págs. 569-572.
- [69] A. Pichler, P. Raymond y M. Eichhorn. “Fast wavefront sensing using a hardware parallel classifier chip”. En: *Applied Physics B* 115.3 (2014), págs. 325-334.
- [70] M. Reichenbach et al. “Fast image processing for optical metrology utilizing heterogeneous computer architectures”. En: *Computers & Electrical Engineering* 40.4 (2014), págs. 1158-1170.
- [71] S. Mauch y J. Reger. “Real-time implementation of the spiral algorithm for Shack-Hartmann wavefront sensor pattern sorting on an FPGA”. En: *Measurement* 92 (2016), págs. 63-69.
- [72] S. Mauch y J. Reger. “Real-Time Spot Detection and Ordering for a Shack-Hartmann Wavefront Sensor With a Low-Cost FPGA”. En: *IEEE Transactions on Instrumentation and Measurement* 63.10 (2014), págs. 2379-2386.
- [73] S. Meimon et al. “Pupil motion analysis and tracking in ophthalmic systems equipped with wavefront sensing technology”. En: *Applied Optics* 56.9 (2017), págs. D66-D71.
- [74] J. Arines, P. Prado y S. Bará. “Pupil tracking with a Hartmann-Shack wavefront sensor”. En: *Journal of biomedical optics* 15.3 (2010), págs. 036022-036022.
- [75] A. de Castro et al. “Adaptive optics retinal imaging with automatic detection of the pupil and its boundary in real time using Shack-Hartmann images”. En: *Appl. Opt.* 56.24 (ago. de 2017), págs. 6748-6754. DOI: 10.1364/AO.56.006748. URL: <http://ao.osa.org/abstract.cfm?URI=ao-56-24-6748>.
- [76] N. Otsu. “A Threshold Selection Method from Gray-Level Histograms”. En: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), págs. 62-66.
-

-
- [77] G. Yoon. "Wavefront Sensing and Diagnostic Uses". En: *Adaptive Optics for Vision Science*. John Wiley & Sons, Ltd, 2005. Cap. 3, págs. 63-81. ISBN: 9780471914877. DOI: 10.1002/0471914878.ch3. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/0471914878.ch3>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/0471914878.ch3>.
- [78] G. Bradski. "The OpenCV Library". En: *Dr. Dobb's Journal of Software Tools* (2000).
- [79] G. Guennebaud, B. Jacob et al. *Eigen v3*. <http://eigen.tuxfamily.org>. 2010.
- [80] J. F. Castejón-Mochón et al. "Ocular wave-front aberration statistics in a normal young population". En: *Vision research* 42.13 (2002), págs. 1611-1617.
- [81] V. Pourreza Ghouschi, P. Prieto y P. Artal. "Speed of accommodation responses in myopes." En: *Investigative Ophthalmology & Visual Science* 60.9 (2019), págs. 1791-1791.
- [82] I. Grulkowski et al. "Long-Depth-Range Swept-Source OCT Instrument for Imaging Crystalline Lens Opacities". En: *Investigative Ophthalmology & Visual Science* 57.12 (2016), págs. 437-437.
- [83] I. Grulkowski et al. "Volumetric macro- and micro-scale assessment of crystalline lens opacities in cataract patients using long-depth-range swept source optical coherence tomography". En: *Biomed. Opt. Express* 9.8 (ago. de 2018), págs. 3821-3833. DOI: 10.1364/BOE.9.003821. URL: <http://www.osapublishing.org/boe/abstract.cfm?URI=boe-9-8-3821>.
- [84] A. de Castro et al. "Three-Dimensional Cataract Crystalline Lens Imaging With Swept-Source Optical Coherence TomographyCataract Crystalline Lens Imaging With SS-OCT". En: *Investigative Ophthalmology & Visual Science* 59.2 (feb. de 2018), págs. 897-903. ISSN: 1552-5783. DOI: 10.1167/iovs.17-23596. eprint: https://iovs.arvojournals.org/arvo/content/_public/journal/iovs/936739/i1552-5783-59-2-897.pdf. URL: <https://doi.org/10.1167/iovs.17-23596>.
- [85] I. Grulkowski et al. "Vitreous body imaging with long-range swept-source optical coherence tomography for detection of opacities". En: *Investigative Ophthalmology & Visual Science* 60.9 (2019), págs. 1291-1291.
-

-
- [86] I. Grulkowski et al. "In vivo imaging of vitreous opacities with full-eye-length SS-OCT (Conference Presentation)". En: vol. 10858. 2019. DOI: 10.1117/12.2509474. URL: <https://doi.org/10.1117/12.2509474>.
- [87] A. Rodríguez-Aramendía et al. "Optimization of a SS-OCT with a focus tunable lens for enhanced visualization of ocular opacities". En: vol. 10867. 2019. DOI: 10.1117/12.2508640. URL: <https://doi.org/10.1117/12.2508640>.
- [88] L. T. Chylack et al. "The lens opacities classification system III". En: *Archives of ophthalmology* 111.6 (1993), págs. 831-836.
- [89] G. W. Cheon et al. "Accurate real-time depth control for CP-SSOCT distal sensor based handheld microsurgery tools". En: *Biomedical optics express* 6.5 (2015), págs. 1942-1953.
- [90] Y. Jian, K. Wong y M. V. Sarunic. "GPU accelerated OCT processing at megahertz axial scan rate and high resolution video rate volumetric rendering". En: *Optical Coherence Tomography and Coherence Domain Optical Methods in Biomedicine XVII*. Vol. 8571. International Society for Optics y Photonics. 2013, 85710Z.
- [91] K. Zhang y J. U. Kang. "Graphics processing unit accelerated non-uniform fast Fourier transform for ultrahigh-speed, real-time Fourier-domain OCT". En: *Optics express* 18.22 (2010), págs. 23472-23487.
- [92] N. H. Cho et al. "High speed SD-OCT system using GPU accelerated mode for in vivo human eye imaging". En: *Journal of the Optical Society of Korea* 17.1 (2013), págs. 68-72.
- [93] G. W. Cheon et al. "Motorized microforceps with active motion guidance based on common-path SSOCT for epiretinal membranectomy". En: *IEEE/ASME Transactions on Mechatronics* 22.6 (2017), págs. 2440-2448.
- [94] J. Xu et al. "Real-time acquisition and display of flow contrast using speckle variance optical coherence tomography in a graphics processing unit". En: *Journal of biomedical optics* 19.2 (2014), págs. 026001.
- [95] P. Steinbach y M. Werner. "gearshifft—the fft benchmark suite for heterogeneous platforms". En: *International Supercomputing Conference*. Springer. 2017, págs. 199-216.
- [96] AMD. *clFFT*. <https://github.com/clMathLibraries/clFFT>. Accedido: 27-08-2019.
-

-
- [97] M. Frigo y S. G. Johnson. “The Design and Implementation of FFTW3”. En: *Proceedings of the IEEE* 93.2 (2005). Special issue on “Program Generation, Optimization, and Platform Adaptation”, págs. 216-231.
- [98] P. A. Soriano, J. M. ESTEBA y J. L. Aragón. *Optoelectronic binocular instrument for the correction of presbyopia and method for the binocular correction of presbyopia*. US Patent App. 16/325,001. Jun. de 2019.
- [99] J. Mompeán et al. “GPU-Accelerated High-Speed Eye Pupil Tracking System”. En: *Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on Computer Architecture and High Performance Computing*. IEEE. Florianópolis, Brazil, 2015, págs. 17-24.
- [100] J. Mompeán et al. “Robust High-speed Eye Pupil Tracking System on GPUs”. En: *XXVII Jornadas de Paralelismo, 2016 Jornadas SARTECO*. SARTECO. Salamanca, España, 2016.
- [101] J. Mompeán et al. “High-speed Wavefront Sensor Processing with Graphics Accelerators”. En: *XI Workshop on Adaptive Optics for Industry and Medicine, 2018*. Laboratorio de Óptica de la Universidad de Murcia. Murcia, España, 2018.
- [102] J. Mompeán et al. “Cálculo de Mapas de Frente de Onda mediante GPUs para Aplicaciones de Óptica Adaptativa en Tiempo Real”. En: *XXVII Jornadas de Paralelismo*. Vol. 1. 1. SARTECO. 2017, págs. 107-112.
- [103] J. Mompeán et al. “GPU-based processing of Hartmann–Shack images for accurate and high-speed ocular wavefront sensing”. En: *Future Generation Computer Systems* 91 (2019), págs. 177-190.
- [104] H. S. Ginis et al. “Macular pigment density estimated with Fourier-domain fundus reflectometry”. En: *Investigative Ophthalmology & Visual Science* 57.12 (2016), págs. 5097-5097.
- [105] D. Christaras et al. “An optical method to assess the macular pigment density”. En: *Investigative Ophthalmology & Visual Science* 60.9 (2019), págs. 6094-6094.
- [106] D. Christaras et al. “Objective method for measuring the macular pigment optical density in the eye”. En: *Biomedical optics express* 10.7 (2019), págs. 3572-3583.
-

-
- [107] A. Gupta et al. "In vivo SS-OCT imaging of crystalline lens sutures". En: *Biomed. Opt. Express* 11.10 (oct. de 2020), págs. 5388-5400. DOI: 10.1364/BOE.401254. URL: <http://www.osapublishing.org/boe/abstract.cfm?URI=boe-11-10-5388>.
- [108] D. Christaras et al. "A portable optical instrument for the measurement of macular pigment optical density". En: *Investigative Ophthalmology & Visual Science* 61.9 (2020), PB00128-PB00128.
- [109] C. Robles et al. "Objective Eye's Refraction from Pupil-tracking". En: *Investigative Ophthalmology & Visual Science* 61.7 (2020), págs. 535-535.
- [110] A. Gkiala et al. "Macular pigment optical density in a healthy population measured with a new optical instrument". En: *Investigative Ophthalmology & Visual Science* 61.7 (2020), págs. 1856-1856.
- [111] J. Gourlay et al. "A real-time closed-loop liquid crystal adaptive optics system: first results". En: *Optics communications* 137.1-3 (1997), págs. 17-21.
-