Departamento de Arquitectura de Computadores

E.T.S. Ingeniería Informática

TESIS DOCTORAL

Clasificación tisular en GPU: aceleración y optimizaciones

Presentada por:

Antonio Ruiz Sánchez

Dirigida por:

Manuel Ujaldón Martínez

Málaga, 2015

Publicaciones y Divulgación Científica

AUTOR: Antonio Ruiz Sánchez
http://orcid.org/0000-0001-5997-9792

EDITA: Publicaciones y Divulgación Científica.
Universidad de Málaga

UNIVERSIDAD
DE MÁLAGA

Dr. D. Manuel Ujaldón Martínez. Profesor Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

CERTIFICA:

Que la memoria titulada "Clasificación tisular en GPU: Aceleración y optimizaciones", ha sido realizada por D. Antonio Ruiz Sánchez bajo mi dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y concluye la Tesis que presenta para optar al grado de Doctor por la Universidad de Málaga.

Málaga, a 4 de junio de 2015

Dr. D. Manuel Ujaldón Martínez.
Director de la tesis.

A Indira, y a todos quienes la aman

Agradecimientos

Esta tesis es el resultado del equilibrio de un cúmulo de circunstancias y sentimientos tan enfrentados como irracionales. Durante todo este tiempo he vivido muchos momentos de ilusión, de trabajo intenso, de abandonos, de aprendizaje, de decepción, de segundas oportunidades, etc. Sin embargo, tengo que dar las gracias a cada uno de estos momentos, que junto con el apoyo incondicional de mis más allegados, me han hecho ser mejor y crecer en lo personal y profesional.

En primer lugar, quisiera expresar mi enorme gratitud a mi director de tesis, Manuel Ujaldón, por confiar y creer en mí incondicionalmente, por ser el mejor ejemplo del trabajo y el sobresfuerzo, y por transmitirme ánimos e ilusión incluso cuando ya daba por hecho que esta tesis quedaría en un cajón y nunca llegaría a ver la luz. Aún estoy más agradecido por sentir tu confianza incluso después de tomar la decisión de desvincularme de la investigación a tiempo completo. Por todo ello, esta tesis te pertenece, y me gustaría que la sintieras como un éxito de tu dirección y dedicación, a pesar de todas las tempestades ajenas a tu voluntad que has encontrado en el camino. Además, me gustaría resaltar y agradecer la oportunidad que me brindaste para trabajar en el área de las GPUs y la Supercomputación, tengo que reconocer que no he encontrado ningún tema que me llegue a apasionar tanto después de más de seis años de carrera profesional paralela. Tu enseñanza es un legado para afrontar con éxito mi carrera profesional.

Me gustaría agradecer la hospitalidad de todo el personal del Departamento de Arquitectura de Computadores, desde becarios hasta profesores, sin olvidarme de los técnicos y la secretaria. Especial reconocimiento merece el interés mostrado por mi trabajo y las sugerencias recibidas del profesor Nicolás Guil.

Quisiera hacer extensiva mi gratitud al personal del departamento de Biomedical Informatics de The Ohio State University. A los profesores Umit Catalyurek, Kun Huang y Metin Gurcan, a los ya doctores Tim Hartley, Jun Kong, Lee Cooper, y en especial a Olcay Sertel, quien me acogió durante mi estancia de investigación como a un hermano y con quien mantengo la amistad desde la distancia.

Siento la obligación y necesidad de agradecer con mención individualizada a cada uno de mis compañeros del despacho de becarios (también conocido como corral). La obligación, porque ellos tuvieron que aguantarme en el día a día; la necesidad, porque además hicieron parecer que era un placer compartir tiempo y lugar. A todos ellos, Antonio Muñoz, Francisco Jaime, Ricardo Quislant, Rosa Castillo, Adrián Tineo, Juan Lucena, Sergio Varona, Miguel Ángel Sánchez, Victoria Martín, Javier Ríos, Maximiliano García, Fernando Barranco y Alfredo Martínez. Y a cada uno de mis compañeros de trabajo que se han involucrado de una manera u otra, especialmente a Pedro Antonio por su interés y recomendaciones tras la lectura de esta tesis.

Quisiera también agradecer a cada uno de mis amigos de toda la vida, por el simple hecho de seguir siéndolos como el primer día. En especial a Salva, Escarcena, Chacón, Aroro y Trivi, ellos siempre han estado disponibles para todo.

Por último, pero no por ello menos importante, me gustaría agradecer y dedicar este trabajo a mis padres, mujer e hija. Todos ellos han hecho realidad el llegar hasta aquí, con el apoyo incondicional y el seguimiento desde el silencio para encontrarme con todas las facilidades del mundo. Padres, gracias por vuestro apoyo y esfuerzo por permitirme realizarme en mis estudios y mi vida. Agradezco los consejos sabios que en el momento exacto habéis sabido darme para continuar luchando pese a todas las adversidades, y cuando pareciera que no los valoraba. A mi mujer, Ana Eva, debo agradecer encarecidamente todo su apoyo, anímico y afectivo, pese al tiempo que esta tesis le ha robado. Gracias por darle vida a la persona más importante de nuestras vidas. Indira nos ha enseñado lo que es amar de verdad. Gracias hija por tus ganas de jugar y disfrutar cuando veías que no te dedicaba todo el tiempo que merecías porque lo dedicaba a algo que aún estás lejos de entender. A cada uno de los mencionados en este párrafo, os quiero de una manera muy especial, ahora que soy padre sé de primera mano todo lo que siempre me aman los míos.

Resumen

Desde hace una década, los procesadores gráficos o GPUs vienen ganando protagonismo en la computación de altas prestaciones, contribuyendo a la aceleración de miles de aplicaciones en multitud de áreas de la ciencia. Pero más que esta conquista, lo que ha hecho singular al movimiento GPGPU ha sido la vía para su consecución, ofreciendo tecnología popular, barata y notablemente arropada. Como resultado, la supercomputación está hoy al alcance de cualquier usuario y empresa, democratizando un sector hasta entonces circunscrito a unos pocos centros elitistas.

El auge de las GPUs en los entornos de altas prestaciones ha generado un reto a la comunidad de desarrolladores *software*. Los programadores están habituados a pensar y programar de manera secuencial, y sólo una minoría se atrevía hace 10 años a adentrarse en este mundo. La programación paralela es una tarea compleja que exige otras habilidades y modelo de razonamiento, además de conocer nuevos conceptos *hardware*, algoritmos y herramientas de programación. Poco a poco, esta percepción ha ido cambiando gracias a la aportación de aquellos que, conscientes de la dificultad, quisieron aportar su granito de arena para facilitar esta transición.

El trabajo de esta tesis recoge este espíritu. Planteamos nuevos diseños e implementaciones de algoritmos en el ámbito de la biocomputación para evaluar el rendimiento de las GPUs más destacadas durante la última década, desde equipos con una única GPU hasta supercomputadores de 32 GPUs. En cada uno de los problemas de biocomputación se han analizado todas las características relevantes de la GPU que permiten exprimir su gran potencial, para así presentar de una manera didáctica y rigurosa un estudio pormenorizado de los detalles y técnicas de programación más acordes a cada tipo de algoritmo.

Cronológicamente, la aparición de la arquitectura de cálculo paralelo CUDA para GPUs es un hito de especial importancia en la programación de algoritmos de propósito general en GPUs. Nuestro trabajo comenzó en la era pre-CUDA con una aplicación de detección de círculos basada en la transformada de Hough y un algoritmo de detección del tumor neuroblastoma. Sus implementaciones explotan la GPU desde una

perspectiva más artesanal, empleando un gran abanico de unidades funcionales de la GPU. Para ello fueron necesarios buenos conocimientos del cauce de segmentación gráfico y ciertas dosis de creatividad.

Lo habitual en aquella época era aprovechar casi de forma exclusiva los procesadores de píxeles, al ser los más numerosos y mostrar ya claros indicios de escalabilidad. Entre tanto, nuestro estudio se dedicó a mostrar el potencial de otros recursos menos populares, como los procesadores de vértices, el rasterizador (conversión de polígonos en píxeles) y las unidades de *blending* (mezclado de contenidos en pantalla). Con la irrupción de CUDA, nuestra atención se dirigió a aplicaciones más exigentes, como el registro de imágenes o el cálculo de los momentos de Zernike para caracterizar regiones tisulares. Completamos también nuestro estudio del neuroblastoma, para poder así contrastar las facilidades aportadas por CUDA y sus posibilidades de optimización.

Respecto a las arquitecturas gráficas objeto de nuestro análisis, comenzamos nuestra andadura con modestas GeForce, prosiguiendo con Quadro de gama alta, y concluyendo con Tesla de propósito general, justo donde muchos se iniciaron en el mundo GPGPU para tomar el relevo. La longevidad del algoritmo de detección de tumores nos ha permitido comparar evolutivamente todas estas arquitecturas, el registro de imágenes, ilustrar el beneficio de apoyarse en una popular librería como cuFFT, y los momentos de Zernike, desvelar las exigencias para optimizar el código en generaciones venideras (en nuestro caso, Fermi y Kepler).

La exploración de este amplio abanico de posibilidades, tanto en la vertiente software como en la diversidad de modelos *hardware* que nos han acompañado, desemboca en un sinfín de aportaciones que, además de contribuir a una aceleración de hasta dos órdenes de magnitud en comparación con CPUs de su misma gama, han permitido que el trabajo de esta tesis siente las bases de otras muchas líneas de investigación que han dado crédito y continuidad a nuestro esfuerzo.

Índice general

Índice de figuras

Índice de tablas

Parte I

Introducción

1 Aplicaciones biomédicas sobre arquitecturas gráficas

Las aplicaciones de biomedicina se han consolidado entre las más relevantes y exigentes computacionalmente en el contexto actual de la investigación a escala mundial, haciéndose acreedoras al uso de redes de estaciones de trabajo gráficas, sistemas multiprocesador, o incluso potentes supercomputadores para su eficiente ejecución.

Esta tesis doctoral aborda el empleo de *hardware* alternativo para su eficiente ejecución, más concretamente los procesadores gráficos o GPUs (del inglés, *Graphics Processing Units*). Cuando comenzamos este trabajo ocho años atrás, la GPU era una plataforma emergente, llena de posibilidades para la aceleración de aplicaciones científicas, pero inexplorada en el campo de la biocomputación. Ahora resulta habitual encontrar resultados como los que aquí presentamos, pero nuestra aportación, pionera en algunos algoritmos relevantes del procesamiento de imágenes biomédicas, fue anterior a casi todos ellos y debe contemplarse en esa perspectiva. La fecha de publicación de nuestros artículos delata el coste de oportunidad de los hallazgos aquí recopilados, los lugares donde éstos se publicaron, y la relevancia de los mismos.

Uno de los mayores retos de esta memoria consiste en reflejar el marco temporal en que trabajamos con cada algoritmo y generación de GPU. La efervescencia del binomio biocomputación-GPU ha permitido desde entonces progresar en muchos aspectos a partir de nuestras contribuciones, sirviendo éstas de pilares a los programadores que entonces se iniciaban con CUDA. Ha sido así cómo nuestras publicaciones han venido acumulando referencias por parte de otros colegas, aunque si son leídas ahora, su contenido pueda parecer algo simplista. Un ejemplo lo tenemos en uno de nuestros primeros artículos, publicado en el ICS de 2008 [30], cuyos kernels CUDA sentaron las bases de muchos desarrollos posteriores. Este artículo ha sido recientemente galardonado por la ACM como uno de los 25 más influyentes publicados en

sus primeros 25 años de historia [1].

Volviendo a los orígenes de la GPU como acelerador de aplicaciones, la aparición de CUDA en el preciso momento en el que comenzamos esta tesis nos permitió aprovechar sus propiedades de paralelismo SIMD (Simple Instrucción Múltiples Datos), procesamiento vectorial y elevado ancho de banda. Dichas cualidades la señalaban como la candidata ideal para la aceleración de códigos en el contexto de la bioinformática actual, y dentro de ella, aquí nos decantamos por el análisis biomédico de imágenes procedentes de distintos ámbitos de investigación con los que hemos trabajado. Cabe citar entre ellos el James Cancer Hospital vinculado a Ohio State University (Estados Unidos).

La potencia de la GPU ha venido duplicándose cada seis meses desde aquellos orígenes, lo que representa un ritmo tres veces superior al progreso de la CPU en su famosa Ley de Moore. Esto ha situado a la GPU cada vez más en el punto de mira de la ciencia, y su popularidad en el entorno de los videojuegos ha permitido a los científicos disfrutar también de costes irrisorios para procesar aplicaciones de muy diversa índole, algo sin precedentes en la historia de la computación de altas prestaciones o HPC (del inglés, *High Performance Computing*). Como consecuencia, el ratio rendimiento/coste de las GPUs ha sido cada vez más atractivo, y la eclosión de resultados ha permitido disminuir el esfuerzo de implementación. En la actualidad pueden encontrarse multitud de librerías y herramientas que suavizan la curva de aprendizaje, bastante dura cuando nos iniciábamos en la implementación de *kernels* CUDA para nuestros algoritmos.

El contenido de este capítulo está estructurado de la siguiente manera: La Sección 1.1 sitúa las GPUs como plataforma objeto de nuestros esfuerzos de aceleración de aplicaciones biomédicas a lo largo de esta tesis. La Sección 1.2 justifica la razón de este trabajo junto con sus objetivos, realizando además un repaso general del contenido de todo el documento. Finalmente, la Sección 1.3 define la infraestructura *hardware* empleada durante la fase de evaluación experimental para cuantificar las aceleraciones logradas.

1.1. Biomedicina y supercomputación con GPUs

Las aplicaciones biomédicas están cobrando una importancia capital tanto para la comunidad científica como para el bienestar público. En particular, aquellas relacio-

[1] Un número especial de la ACM publicado para conmemorar el 25 aniversario del ICS y titulado "ICS Retrospective" incluye un memorándum de 3 páginas que resume la relevancia científica otorgada por la comunidad HPC a este artículo.

nadas con el procesamiento de imágenes están surgiendo como una nueva oportunidad para la innovación en el punto de encuentro entre la medicina y la informática. Estas aplicaciones son un reto por varias razones. En particular, las aplicaciones de análisis de imágenes biomédicas proporcionan muchas oportunidades para la innovación, desde las preocupaciones prácticas en la rutina de trabajo diaria de un médico hasta las dificultades en el desempeño centrada en el uso de arquitecturas disruptivas para lograr el máximo rendimiento.

La tecnología ha multiplicado últimamente su impacto en la investigación biomédica con imágenes. La reciente disponibilidad de escáneres digitales de muy alta resolución ha hecho que la investigación sobre el análisis de imágenes patológicas resulte más atractiva, al permitir el uso de herramientas de análisis para reducir el tiempo de evaluación que los especialistas dedican a cada imagen escaneada. Esto reduce además la variación en la toma de decisiones entre diferentes especialistas e instituciones, fomentando la reproducibilidad experimental.

Tras las mejoras en la definición de imágenes, el reto se traslada al ámbito computacional, donde el gran volumen de datos aportado por las imágenes biomédicas las sitúan como una de las áreas más prolíficas y representativas del fenómeno recientemente acuñado como *big-data*. Una imagen sin comprimir de las que hemos empleado en nuestros estudios puede fácilmente alcanzar los 30 gigabytes para una lámina de tejido, y algunos conjuntos de datos típicos llegan fácilmente a escalas del terabyte. La tecnología de almacenamiento se encuentra mucho más preparada para dar respuesta a este reto que la requerida para su procesamiento. Almacenar datos es cada vez más barato y escalable, pero el coste y el tiempo computacional no ofrecían una alternativa similar por el lado del procesamiento. Hasta que entraron en escena las GPUs y pudimos ambicionar estar a la altura. Esta tesis doctoral surgió de esa ambición, y para focalizarnos decidimos centrarnos en las técnicas para la detección de diversos tipos de cáncer, fundamentalmente infantil como el neuroblastoma.

Numerosos estudios de investigación sobre los diferentes tipos de cáncer se han llevado a cabo para desarrollar métodos computacionales dentro de este ámbito [53, 27, 39, 44, 63, 70, 73]. La mayoría de estos enfoques se prueban sólo sobre secciones de la imágenes elegidas de forma más o menos aleatoria, mientras que otros [62] han extendido el procesamiento a la imagen completa, pero sin estudiar la carga computacional. Uno de los resultados que aportaba técnicas novedosas de paralelización [13] no se enfocaba estrictamente a lograr mejoras de rendimiento, por lo que el tiempo de procesamiento para una imagen relativamente pequeña tomó casi media hora en un multiprocesador de 16 nodos, lo que resulta todavía poco práctico para la aplicación clínica.

Sin embargo, de forma coetánea con aquellos resultados se empezaba a ver la fuer-

za de las GPUs en aplicaciones científicas de muy diversa índole, como la minería de datos [25], la segmentación y clasificación de imágenes [28], los métodos numéricos para cálculos de elementos finitos utilizados en simulaciones interactivas en 3D [80], y simulaciones de la energía nuclear, dispersión de gases y calor resplandeciente [83], por citar sólo algunas de ellas. Eran los tiempos de mediados de la década pasada, cuando las GPUs basaban su polivalencia en la programación de sombreadores *(shaders)* y su fuerza en el paralelismo de datos, originando los primeros procesadores *multi-core* que pronto pasarían a ser *many-core*. Owens [58] resume de forma magistral estos primeros resultados y las técnicas que se utilizaron para programar aquellas GPUs. Desde entonces ha pasado una década, y los fabricantes de GPU, principalmente Nvidia, han respondido a la gran aceptación y el uso de las GPU de propósito general. El eje central de esta innovación es CUDA (Compute Unified Device Architecture), un nuevo paradigma que lleva de la mano la arquitectura y su programación para transformar la GPU en un coprocesador de propósito general, GPGPU *(General Purpose Graphics Processing Units)*, que ayude a la CPU cuando el paralelismo de datos es aplicable, y al tiempo la mantenga ocupada con los códigos en los que el paralelismo de tareas es preferible. Aquí tiene su origen el concepto de computación heterogénea tan popular en nuestros días.

Las primeras plataformas que aparecen en el mercado con el nacimiento de CUDA son Tesla en Nvidia y FireStream en AMD. AMD propugna un API diferente primero (CTM, *Close-To-Metal*) y un estándar abierto después (OpenCL), sin haber alcanzado las cotas de popularidad de CUDA hasta la fecha. Posteriormente, entraría también en escena Intel con Larrabee, MIC, y más recientemente, Xeon Phi. De esta manera, el movimiento GPGPU gana credibilidad en la comunidad científica y termina conquistando el territorio de la supercomputación a costes asequibles, estableciéndose un ecosistema de positiva realimentación en el que los científicos apadrinan cada vez más estas plataformas y los fabricantes mejoran sus prestaciones para propósito general: Precisión de punto flotante, consumo, acceso a memoria, ...

1.2. Motivación y estructura del documento

1.2.1. Motivación y objetivos

Este trabajo se centra en aprovechar las excelentes credenciales de las arquitecturas gráficas para supercomputación. La plena disponibilidad y continua innovación de las GPUs está garantizada gracias a la masiva orientación de estos productos a un mercado de consumo en el que el precio constituye la variable más sensible de cara a la competitividad comercial del producto. La feroz competencia del sector y la

creciente demanda por parte del usuario representan claros avales para los próximos años, garantizando su sostenibilidad. De hecho, la industria del videojuego facturó en 2013 más de setenta mil millones de euros, más que el sector del cine y la música juntos. Nuestra intención aquí es aprovechar la capacidad de desarrollo e innovación de esta industria para trasladarla a fines más científicos dentro del ámbito biomédico.

Las perspectivas de futuro de las GPUs son además excelentes dada su extraordinaria escalabilidad, precisamente cuando los procesadores convencionales (CPUs) muestran claros síntomas de agotamiento debido a la cada vez más compleja reducción de la anchura de puerta del transistor de silicio. Así, las GPU comerciales para PC se encuentran posicionadas en 2015 en torno a los 10.000 GFLOPS de potencia bruta de cálculo, a unos precios en el que las CPU de este mismo mercado ofrecen tasas en torno a los 100 GFLOPS. Son dos órdenes de magnitud, aunque alcanzar el techo de rendimiento en una GPU resulta bastante más desafiante.

Especialmente estratégica para los fines de este trabajo resulta la posición vanguardista que ocupa la memoria de vídeo en el contexto de la plataforma gráfica. Desde principio de este siglo, esta memoria ha mejorado significativamente, situándose dos generaciones por encima de la memoria principal del PC. Así, mientras la memoria DRAM se encuentra en 2015 en fase de transición desde DDR-3 a DDR-4, la memoria de vídeo lo hace desde GDDR-5 a GDDR-6. Esta mayor velocidad se complementa con un extraordinario ancho de banda en torno a los 700 Gbytes/s., resultado de una anchura superior, en torno a las 384 líneas, que conectan los chips de memoria con la GPU gracias a su disposición semicircular en la placa de circuito impreso de la tarjeta gráfica. Por su parte, la CPU tan sólo dispone de 128 bits en configuración de doble canal, alcanzando un máximo de 256 bits en los casos de cuádruple canal, para dejar el ancho de banda por debajo de los 30 Gbytes/s.

Si algo tienen en común las aplicaciones biomédicas es el uso extensivo de imágenes, tanto en su número elevado como en su gran resolución, y para ellas, una memoria de gran tamaño y velocidad representa un enorme bastión para acelerar su ejecución.

1.2.2. Estructura del documento

La presente memoria está estructurada en cuatro partes desglosadas en diferentes capítulos.

La primera parte presenta la información y conceptos básicos necesarios para entender el resto del documento. El Capítulo 1 presenta las razones para usar las GPUs como arquitecturas de propósito general en ámbitos biomédicos. Además, se describen la motivación y objetivos de todo el trabajo de investigación realizado, al igual que la infraestructura *hardware* utilizada durante la fase de evaluación experimental.

El Capítulo 2 presenta en detalle las GPUs, desde las primeras con más de 10 años hasta las más recientes que hemos utilizado, basadas en la generación Kepler de Nvidia.

La segunda parte está relacionada con la programación gráfica más clásica basada en la renderización como si se tratara de videojuegos o animaciones gráficas. Tanto la arquitectura gráfica como las instrucciones son equivalentes a las utilizadas en las animaciones gráficas, de modo que se delega en la pericia de los desarrolladores la transformación del problema computacional en otro ligado al cauce de segmentación gráfico.

El Capítulo 3 implementa sobre esta perspectiva clásica un algoritmo para la detección de formas circulares en imágenes. Durante el transcurso de este capítulo se llevan a cabo, desde un puesto de vista didáctico, numerosas optimizaciones gráficas de la transformada de Hough para la detección de círculos. Se utiliza para ello la arquitectura gráfica convencional, con elementos como los procesadores de vértices y píxeles, el rasterizador y las unidades de *blending*, que son evaluados desde el punto de vista del rendimiento y la precisión.

La tercera parte resuelve diferentes problemas sobre plataformas gráficas basadas en CUDA *(Compute Unified Device Architecture)*. Desarrollado por la empresa Nvidia, fabricante de más del 75 % de las GPUs actuales, CUDA define tanto el modelo de programación como la arquitectura subyacente y el interfaz para la programación de aplicaciones de propósito general *(API, Application Program Interface)*. Las prestaciones y facilidad de uso ofrecidas por CUDA han ido en aumento desde su aparición, y los capítulos de esta parte analizarán todas ellas desde la perspectiva de nuestras aplicaciones biomédicas. El Capítulo 4 emplea CUDA para desarrollar un algoritmo de detección del tumor neuroblastoma. Este capítulo servirá de introducción a una versión inicial de CUDA, puesto que se implementó durante la transición entre la programación con Cg *(C for graphics)* y la programación con CUDA para aplicaciones de propósito general. Además, aprovechando el momento del cambio, se evaluará una interesante comparativa entre las diferentes metodologías de acceso a la GPU, desde equipos con una sola GPU hasta supercomputadores dotados de 16 nodos con 2 GPUs cada uno. El Capítulo 5 ilustra la potencia de cálculo de las primitivas proporcionadas en la librería de CUDA. En concreto, se emplea el cálculo de la transformada rápida de Fourier (FFT) en una aplicación de registro de imágenes. El Capítulo 6 aprovecha las últimas novedades de la arquitectura gráfica Kepler para implementar el algoritmo de los momentos de Zernike, muy útil para el reconocimiento de patrones en imágenes biomédicas, entre otras muchas aplicaciones. Este tramo final aprovecha ya las últimas prestaciones aparecidas en las GPUs de tercera generación correspondientes al bienio 2013-2014.

La cuarta parte presenta las principales conclusiones y aportaciones de la investigación de esta tesis. La Sección 7.1 describe las conclusiones generales obtenidas de cada uno de los trabajos de investigación realizados para cada Capítulo. La Sección 7.2 muestra las diferentes publicaciones fruto del trabajo de esta tesis. Finalmente, la Sección 7.3 describe brevemente las líneas futuras de investigación que nacen de esta tesis.

1.3. Descripción de los sistemas empleados durante los experimentos

Para evaluar las implementaciones presentadas en los siguientes capítulos se han utilizado plataformas de muy variado coste y complejidad, todas ellas con una arquitectura híbrida CPU-GPU, y representativas de la época en la que se abordaron dichas implementaciones. A continuación se describen en orden cronológico cada una de ellas.

Características del procesador	CPU 2007	GPU 2007
Fabricante	Intel	Nvidia
Modelo	Core 2 Duo	GeForce 8800 GTX
Arquitectura	E6400 Conroe	G80
Frecuencia	2.13 GHz	575/1350 MHz
Potencia bruta de procesamiento	10 GFLOPS	520 GFLOPs
Tipo de memoria	DDR2	GDDR3
Tamaño	4 GBytes	768 MBytes
Ancho del bus	2 x 64 bits	384 bits
Frecuencia	2 x 333 MHz	2 x 900 MHz
Ancho de banda	10.8 GB/s	86.4 GB/s

Tabla 1.1: Características de la CPU y GPU del PC utilizado en 2007.

1.3.1. PC típico de 2007 con una CPU y GPU

La primera configuración y más simple trata de un PC del 2007 compuesto de una única CPU y GPU. En la Tabla 1.1 se detallan las especificaciones de sus procesadores. La GPU con arquitectura G80 es la primera que incorpora la tecnología CUDA y que, aunque mantiene la compatibilidad con los algoritmos de propósito general GPGPU diseñados con Cg, proporciona un entorno más amigable al desarrollador. En los capítulos siguientes se analiza una comparativa para esta GPU entre la programación con Cg y con CUDA para los mismos algoritmos.

1.3.2. Supercomputador BALE del Ohio Supercomputer Center

La segunda configuración de la misma época pero más enfocada a supercomputación es el clúster BALE del centro estadounidense Ohio Supercomputer Center. BALE está compuesto por un total de 71 nodos Linux, pero para nuestro propósito aquí lo más interesante se encuentra en los 16 nodos de visualización incorporados, cada uno de ellos equipados con dos CPUs de doble núcleo AMD Opteron 2218 y dos GPUs Nvidia Quadro FX 5600. La red de interconexión entre los nodos es Infiniband. La Figura 1.1 ilustra la arquitectura de los nodos de visualización, y la Tabla 1.2 resume las especificaciones de los dos procesadores, tanto CPU como GPU.

Cada nodo de BALE es una placa base con doble zócalo (*dual-socket*) dotada de sendos procesadores Opteron X2 2218 de doble núcleo a una frecuencia de 2.6 GHz. Cada núcleo en el sistema tiene un par de cachés L1 gemelas para datos e instrucciones, con capacidad para 64 KB y asociatividad en conjuntos de 2 vías. La caché L2 de 1 MB y también asociativa de dos vías no es compartida por los núcleos, pero sí mantiene la coherencia de caché. Cada zócalo proporciona su propio controlador de memoria DDR2 a 667 MHz de doble canal al igual que un enlace *HyperTransport* para acceder a la caché y memoria de los otros zócalos, ofreciendo un ancho de banda de 10.6 GB/s para un total de 21.3 GB/s para cada nodo de 8 GB de memoria principal. El pico de rendimiento para una aritmética de doble precisión es de 4.4 GFLOPS por núcleo, 8.8 GFLOPS por zócalo y 17.6 GFLOPS por nodo. En simple precisión, el rendimiento conseguido por nodo asciende a 35.2 GFLOPS, proporcionando un total para todo el conjunto de nodos de visualización de 563.2 GFLOPs.

Características	CPU de AMD	GPU de Nvidia
Placa base	ASUS KFN32-D SLI	Quadro FX 5600
Modelo de procesador	Opteron X2 2218	G80
Velocidad del procesador	2.6 GHz	600/1350 MHz
Número de zócalos	2	2
Número de núcleos	2	128
Rendimiento pico	2 x 8.8 GFLOPS	2 x 330 GFLOPS
Tamaño memoria	8 GBytes	2 x 1.5 GBytes
Ancho del bus	2 x 64 bits	2 x 384 bits
Frecuencia memoria	667 MHz	1600 MHz
Ancho de banda	2 x 10.8 GB/s	2 x 76.8 GB/s

Tabla 1.2: Características *hardware* de las CPU y GPUs usadas en cada nodo del supercomputador BALE. Los GFLOPS se calculan para aritmética de punto flotante de simple precisión (32 bits).

Respecto a las prestaciones gráficas de cada nodo, tiene dos tarjetas gráficas Nvidia Quadro FX 5600 basadas en la arquitectura G80. En un entorno gráfico, la ar-

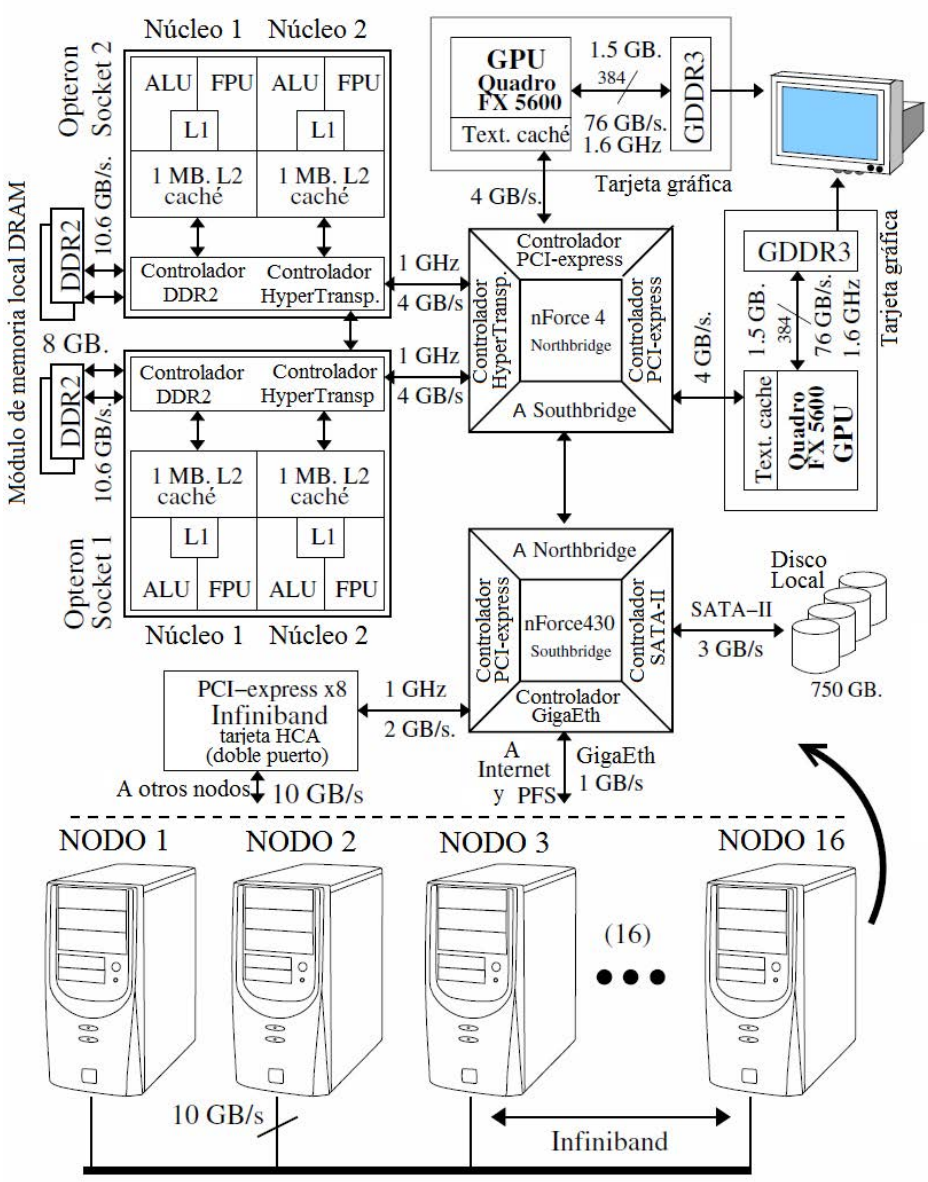


Figura 1.1: El supercomputador BALE.

quitectura G80 puede verse como un cauce de segmentación gráfico de 4 fases para sombreadores *(shaders)*, texturas, rasterizado y coloreado. Como arquitectura paralela, la G80 es un procesador SIMD (*Single Instruction Multiple Data*) compuesto de 128 núcleos, accesible a través de la interfaz proporcionada por CUDA.

1.3.3. Servidor de computación YUCA

La última y más moderna configuración la aporta el servidor de computación Yuca del Departamento de Arquitectura de Computadores de la Universidad de Málaga (ver Tabla 1.3). Las dos GPUs empleadas son diferentes: una con arquitectura Fermi (vigente en el trienio 2010-2012) y otra con arquitectura Kepler (2013-2015). La GPU Fermi permite situar la referencia de los resultados experimentales en una arquitectura de segunda generación, de modo que se pueda cuantificar la rémora con respecto a la tercera generación de multiprocesadores CUDA SMX, incluso antes de aplicar las nuevas técnicas y características introducidas por Kepler.

Procesador	GPU Fermi (Nvidia)	GPU Kepler (Nvidia)
Unidades	1	1
Modelo comercial	Tesla C2075	Tesla K20c
Números de núcleos @ frecuencia	448 @ 1.15 GHz	2496 @ 0.71 GHz
Threads activos por núcleo	48	64
Rendimiento pico	1.03 TFLOPS	3.52 TFLOPS
Frecuencia de la memoria	2x 1566 MHz	2x 2600 MHz
Ancho de banda del bus	384	320
Ancho de banda de memoria	148 GB/s	208 GB/s
Tamaño de memoria y tipo	6 GB de GDDR5	5 GB de GDDR5
Bus a/desde CPU	PCI-e x16 2.0	PCI-e x16 2.0

Tabla 1.3: Características de las GPUs del servidor de computación Yuca.

2 Arquitecturas gráficas

La popularidad alcanzada por las GPUs como procesadores gráficos programables se debe fundamentalmente a su bajo coste y al gran rendimiento alcanzado en muchas aplicaciones de propósito general. Sin embargo, la naturaleza de las arquitecturas gráficas es bien diferente respecto a la actual evolución de estos procesadores dentro de la computación de altas prestaciones.

Esta introducción a la arquitectura de la GPU se desarrolla en orden cronológico para conocer su origen y metamorfosis, lo que nos permitirá entender mucho mejor el modelo de programación implementado y su adecuación a ciertos tipos de algoritmos.

Hemos estructurado este capítulo de la siguiente forma: La Sección 2.1 presenta la arquitectura gráfica en sus orígenes y describe sus unidades funcionales. La Sección 2.2 muestra el cambio de arquitectura gráfica para habilitar las tecnologías y características específicas de una evolución hacia la computación de propósito general. La Sección 2.3 introduce las características del modelo de programación para las arquitecturas gráficas actuales. Finalmente, la Sección 2.4 recoge la evolución de las características implementadas para mejorar el rendimiento de arquitecturas gráficas cuando éstas se orientan a propósito general.

2.1. Cauce de segmentación gráfico

La naturaleza de la propia arquitectura gráfica justifica que el diagrama de bloques en sus orígenes mantuviera una composición y distribución de componentes común independientemente del fabricante. Los elementos que constituían una tarjeta gráfica de mediados de los noventa siguen ahí, tan sólo levemente retocados, y son los siguientes:

- El **procesador gráfico (GPU)**, encargado de mover los vértices iniciales y agruparlos en polígonos, para posteriormente rasterizarlos y transformarlos en píxeles aplicando texturas y colores, que finalmente se mezclarán con los objetos ya existentes en la escena. En el argot gráfico, este proceso recibe el nombre de **renderización**, y sigue vigente en esencia, aunque las transformaciones de vértices y píxeles son programables mediante sombreadores *(shaders)* (2001), cuya polivalencia y unificación originaron CUDA (2006). La Figura 2.1 ilustra el proceso en su conjunto.

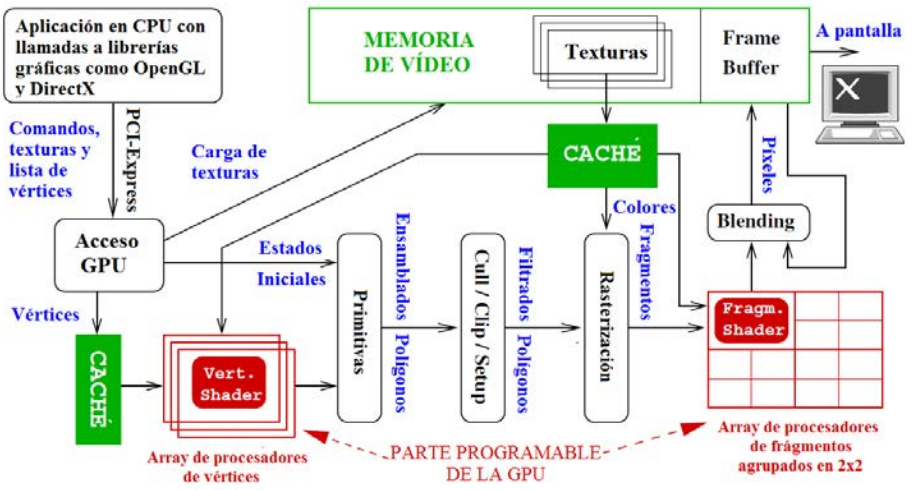


Figura 2.1: Detalle del cauce de segmentación gráfico.

- La **memoria de vídeo (VRAM)**, que aloja todos los datos necesarios para llevar a cabo la renderización, desde los vértices y sus atributos al comienzo del cauce de segmentación, hasta su presentación final en el *frame buffer*, que no es más que una representación interna de lo que vemos en pantalla. Además de estos dos elementos, se alojan en memoria de vídeo los mapas de texturas y el *z-buffer*[1] o *buffer* de profundidad.

- Los **procesadores de señal, luz y color (RAMDAC)**, cuya principal misión es convertir la información digital almacenada en el *frame buffer* en señales aptas para los dispositivos externos de vídeo, analógica para VGA o digital en fechas más recientes para DVI o HDMI.

- Los **conectores externos**, que permiten obtener el resultado de la información

[1] El *buffer* de profundidad que permite distinguir la región visible y oculta de cada objeto según la tercera dimensión espacial (plano Z).

procesada en diferentes fuentes o formatos. Entre ellos podemos citar como más populares DVI, VGA, HDMI, Super-VHS, HD-TV y vídeo compuesto.

- La **BIOS de vídeo**. Es el *firmware* de configuración de la tarjeta gráfica. Esta función ha ido delegándose hacia los *drivers* de la tarjeta gráfica, existiendo utilidades que permiten manipular la configuración desde del propio sistema operativo (SO).

- Los **buses de comunicación**, siendo el bus de memoria de vídeo y el bus que conecta el zócalo gráfico con la CPU (AGP o PCI-Express) los dos más relevantes. El primero comunica la GPU con su memoria de vídeo, y el segundo con el procesador central o CPU, normalmente a través del puente norte, el chip de la placa base que implementa sus controladores más importantes.

- Los **disipadores** encargados de refrigerar todo el conjunto y mantener una temperatura de trabajo adecuada. Suelen ser los más aparatosos y los que ocultan el resto de elementos ya mencionados.

2.1.1. Unidades Funcionales de la GPU

El elemento principal y diferenciador, además de ser el más complejo en cuanto a su arquitectura, es la GPU. De hecho, su protagonismo lleva al lenguaje de la calle a decir que una tarjeta gráfica es de Nvidia o ATI, cuando en realidad sólo lo es la GPU. Nvidia no comercializa GPUs, aunque sí ensambla unas pocas para donarlas a sus GPU Education Centers y GPU Research Centers. El Departamento de Arquitectura de Computadores de la Universidad de Málaga posee ambas distinciones desde hace unos años, lo que le ha permitido beneficiarse de multitud de donaciones, y así las tarjetas gráficas que hemos utilizado en esta tesis proceden directamente de la firma estadounidense.

La complejidad de la GPU ha crecido sobremanera, alcanzando e incluso, en algunos casos, superando a los procesadores de propósito general. Sus elementos más importantes son los procesadores de vértices y píxeles, responsables en buena parte de su versatilidad.

Cada unidad funcional trata la información recibida y la devuelve procesada a la siguiente etapa del cauce de segmentación gráfico. A continuación desglosamos de forma más pormenorizada cada una de ellas.

Vértices

Los vértices son la base de la representación de la geometría de los objetos y contienen múltiple información en forma de atributos:

- Coordenadas tridimensionales. Cada vértice representa su posición por un vector de cuatro componentes. Las dos primeras (x, y) representan la posición en el espacio de dos dimensiones, la tercera z la profundidad o la posición que ocupa en el *z-buffer*, y la última, w, es un factor que permite normalizar el vector $(x, y, z, w) = (x/w, y/w, z/w, 1)$.

- Color. Los vértices pueden presentar dos tipos de colores, el real y el reflectante para los efectos de luz. Ambos tipos son definidos por otras cuatro componentes: rojo, verde, azul y alfa. Sigue el modelo de color RGB con un canal adicional, alfa, para controlar el grado de transparencia u opacidad de cada píxel.

Primitivas

Una primitiva es una forma geométrica simple, desde un punto hasta un polígono. El tipo de primitiva más usado en computación es el triángulo por su versatilidad y rendimiento.

El uso de vértices permite formar diferentes formas de primitivas. En la Sección 3.2 hay un ejemplo práctico del uso de diferentes primitivas y su descripción detallada.

Agrupación de vértices

Esta tarea, además de asociar los vértices con las primitivas seleccionadas, permite generar más vértices con el fin de aumentar el nivel de detalle de la superficie. Para ello, la GPU necesita las posiciones de los vértices y sus vectores normales.

Existen varios métodos para realizar esta tarea, el más simple consiste en aumentar el número de vértices hasta llegar a la resolución deseada, con la desventaja de tener los objetos más distantes con más detalle de lo apreciable y los cercanos pixelados. El método ideal consiste en aumentar el número de vértices conforme disminuye la distancia al punto de vista (*z-buffer*).

Procesado de vértices

El procesado de vértices es uno de los elementos más importantes de la GPU. Este procesador recibe los vértices y los trata en una de las dos modalidades permitidas: fija y programable (*shader* de vértices).

La funcionalidad fija es la única permitida desde los orígenes de las tarjetas gráficas, cuya misión principal es la de efectuar transformaciones basadas en traslaciones, desplazamientos, giros y cambios de escala sobre el conjunto de los vértices.

La funcionalidad programable permite procesar los vértices con programas específicos, denominados *shaders*, que permiten expresar ricos efectos en un lenguaje definido como tal. Este lenguaje ha evolucionado desde HLSL *(High Level Shading Lenguage)* hasta Cg (C for graphics), que puede considerarse una antesala de CUDA con un nivel de abstracción superior.

Clippling, *culling* y rasterización

Los vértices tratados necesitan ser acotados con diversas técnicas para representarlos en un plano bidimensional con unas dimensiones predeterminadas.

La operación *clipping* descarta los polígonos que quedan fuera del ángulo de visión definido por el punto de vista y el plano de la imagen. Define por tanto los límites del espacio bidimensional a representar. La operación *culling* descarta los vértices de cada objeto que quedan ocultos tras su cara visible. Ambas operaciones persiguen filtrar el número de vértices que tienen relevancia para componer la imagen, aliviando la carga computacional de las tareas que les suceden.

La etapa de rasterización convierte cada punto, línea o polígono 3D en una matriz 2D de puntos donde se guarda la información acerca del color y la profundidad para cada uno de los puntos que lo conforman. Para ello entra en juego la resolución de pantalla, ubicándose los vértices en ella mediante un proceso de interpolación. Como resultado, la lista de vértices de entrada se ha convertido en una matriz cuyas dimensiones corresponden a las del *frame buffer*.

Procesado de píxeles o fragmentos

El procesador de píxeles o fragmentos transforma los píxeles atendiendo a la información que les acompañan en forma de atributos. Un buen ejemplo de estos atributos son las coordenadas para acceder al mapa de texturas que se quiere utilizar. Al igual que el procesador de vértices, éste también dispone de una funcionalidad fija

que puede enriquecerse sobremanera mediante una versión programable (*shader* de píxeles).

Sin embargo, la versión programable es también menos eficiente en la mayoría de los casos. Los *shaders* de píxeles tienen un mayor impacto sobre las aplicaciones gráficas que los *shaders* de vértices, ya que por norma general hay muchos más píxeles que vértices. Esto provoca que en una GPU los procesadores para píxeles sean también más numerosos, en proporción aproximada de 3 a 1 cuando había un número fijo de ellos. Ahora, los procesadores están unificados y pueden asignarse en tiempo real a computar la información de los vértices o los píxeles según la complejidad de la escena en cada vertiente. Ampliando esta funcionalidad, llegamos al *Streaming Processor* (SP) que forma parte de la GPU *many-core* visible en el paradigma CUDA.

La principales tareas de la funcionalidad fija son el plegado y el mezclado de las texturas. En el plegado de texturas se recubren los polígonos con las texturas responsables del aspecto exterior. El mezclado de texturas permite ir acoplando texturas diferentes sobre los objetos.

Test alfa y de profundidad

Los píxeles que genera su procesador disponen todavía de la posibilidad de ser tratados en una última fase. Aunque el grueso del trabajo ya está realizado, los test alfa y de profundidad permiten añadir ciertos retoques.

El test alfa permite fijar los píxeles según indiquen su transparencia o canal alfa.

El test de profundidad consiste en un análisis final de la profundidad de los píxeles para asignarlos o descartarlos.

2.2. Arquitectura gráfica Nvidia CUDA G80

La computación de propósito general en procesadores tuvo un pronunciando crecimiento gracias a una arquitectura que cambió la relación entre el cauce de segmentación gráfico programable a nivel lógico y el procesador físico. En 2006, surgió una nueva arquitectura de GPU basada en la idea de unificar los procesadores de vértices y píxeles, de manera que las unidades de procesamiento pudiesen computar todas ellas.

Desde un punto de vista gráfico, dicha transformación tiene como objetivo reducir el desequilibrio de carga entre los procesadores de vértices y de píxeles. Debido a este desequilibrio, muchas unidades funcionales estaban ociosas durante grandes periodos de tiempo. En la arquitectura unificada, sólo hay un tipo de unidad de procesamiento

que es capaz de ejecutar operaciones de vértices y píxeles. De esta manera, el cauce de segmentación se vuelve circular en torno a la unidad de procesamiento para cada tipo de operación, y la potencia de cálculo que no requieran los vértices en un momento dado pueden aprovecharla los píxeles (y viceversa).

En el ámbito GPGPU, la arquitectura unificada ofrece al programador más unidades de procesamiento y una mayor funcionalidad. Sin embargo, la revolución en este sentido viene por parte de la arquitectura y nuevo paradigma de programación CUDA de Nvidia. El propósito de CUDA es crear una implementación para la arquitectura unificada basada en el rendimiento y en la facilidad de programación, abstrayendo los detalles del cauce de segmentación que son patrimonio exclusivo del entorno gráfico.

La primera GPU que implementó la arquitectura unificada con las directrices de CUDA fue la G80 (ver Figura 2.2). Sobre ella, Nvidia ha ido incorporando nuevas funcionalidades agrupadas en forma de generaciones: Fermi es la segunda generación (2010), Kepler la tercera (2012), Maxwell la cuarta (2014) y Pascal la quinta (2016).

2.3. Posibilidades de programación

El rendimiento de los algoritmos ejecutados en GPU depende de lo bien que exploten el paralelismo, la jerarquía de memoria, el ancho de banda y los GFLOPS. Su implementación en CUDA requiere una descomposición en hilos que son ejecutados en un multiprocesador masivamente paralelo, que en el caso de la G80 está compuesto de 128 núcleos (ver fila central en la Figura 2.2). La información se almacena en cachés L1 y L2 y en memoria de vídeo (ver zona inferior en esa misma figura), siendo más rápida aquella que se encuentra más cercana a la unidad de procesamiento. Las memorias cachés explotan mejor la localidad espacial, y éstas son mucho más grandes en las CPUs. En cambio, la localidad temporal beneficia a la GPU, ya que el diseño de la arquitectura y al modelo de programación inspiran el paradigma productor/consumidor. En el estado del arte del 2014 para las GPUs, el ancho de banda máximo está en torno a a los 280 GB/s en comparación con los 20 GB/s para las CPUs. Esta diferencia se fundamenta básicamente en el ancho del bus de datos (384 bits descompuestos en seis particiones de 64 bits tal y como se muestra en la Figura 2.2). La capacidad de la GPU para operaciones de punto flotante de simple precisión rebasa los 6000 GFLOPS, en contraste con los cerca de 100 GFLOPS que proporcionan las CPUs de su misma fecha.

La combinación de las mejores características de ambos procesadores conforma una plataforma de computación tan versátil como eficiente, y que requiere un conocimiento explícito del modelo de programación CUDA para que las aplicaciones puedan

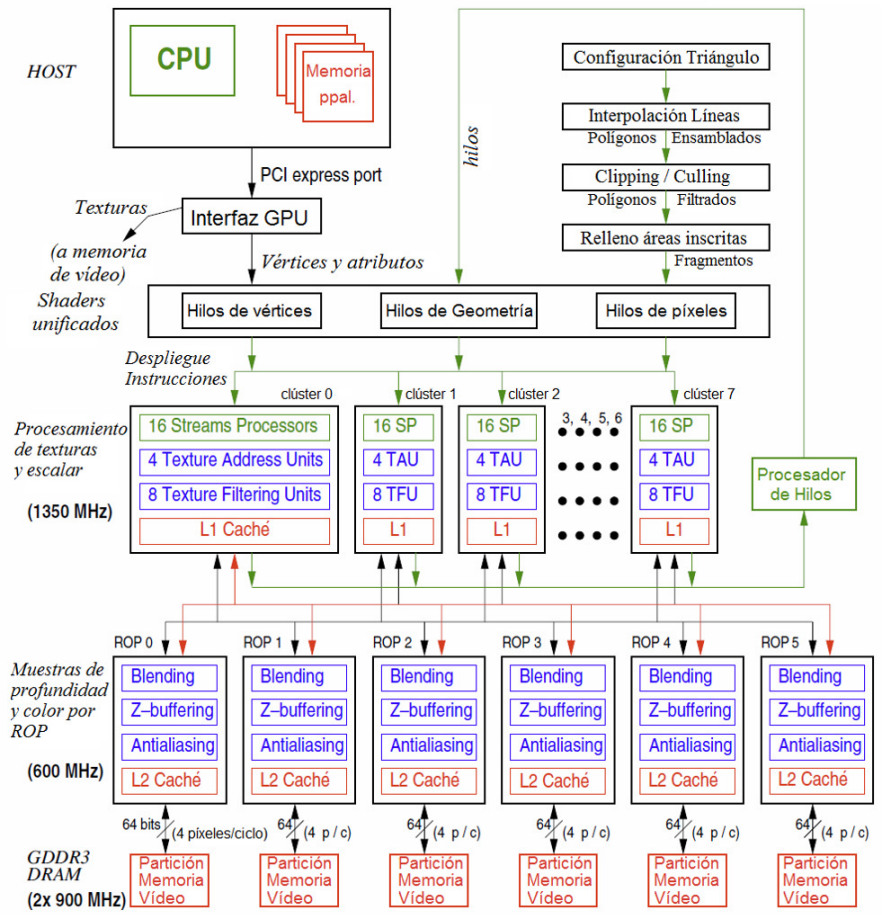


Figura 2.2: Diagrama de bloques de la arquitectura Nvidia G80.

explotar al máximo todo lo que puede ofrecer una GPU actual.

2.3.1. El modelo de programación CUDA

En el paradigma CUDA, la GPU se compone de un conjunto de procesadores estructurados en multiprocesadores simétricos del tipo SIMT (Single Instruction Multiple Threads). Cada multiprocesador tiene una sola unidad de control que comparten todos sus procesadores, junto con una pequeña memoria caché visible al programador, la memoria compartida que enseguida veremos.

La interfaz de programación CUDA [19] consiste en un conjunto de funciones proporcionadas por una librería que puede considerarse una extensión de un lenguaje

de programación de propósito general, que normalmente es C, y en menor medida, Fortran.

El compilador CUDA genera un ejecutable para la GPU que desde el punto de vista de la CPU se percibe como un coprocesador multinúcleo ajeno.

2.3.2. Memoria y registros

Cada GPU dispone de diferentes espacios de memoria (ver Figura 2.3). La memoria global o de vídeo es el único espacio accesible por todos los multiprocesadores, además de la de mayor tamaño y menor velocidad de acceso. Por otro lado, cada multiprocesador tiene su propio espacio de memoria privada denominada memoria compartida, siendo bastante más pequeña pero mucho más rápida que la memoria global.

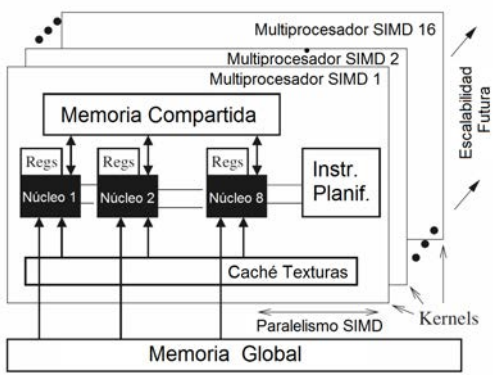


Figura 2.3: La interfaz *hardware* de CUDA para la GPU.

En CUDA, todos los hilos tienen acceso a cualquier espacio de memoria dentro de su jerarquía, pero como es de esperar el rendimiento incrementa conforme se hace un uso más intensivo de la memoria compartida, cuyo uso es explícito por parte del programador.

2.3.3. Paradigma de programación

La programación de propósito general en GPUs fue diseñada inicialmente para aprovechar el flujo de trabajo del cauce de segmentación gráfico, consistiendo cada iteración de procesamiento en una pasada de renderización. Sin embargo, la interfaz *hardware* de CUDA (ver Figura 2.3) plantea ocultar todas estas connotaciones y complejidad, ofreciendo la nueva visión de un programa ramificado en multitud de hilos

que se ejecutan de forma concurrente. Los principales elementos que hacen posible esta nueva perspectiva son los siguientes:

- El **warp** es una colección de hilos que se ejecuta simultáneamente en uno de los multiprocesadores de la GPU. El tamaño de *warp* ha permanecido constante en 32 hilos desde sus orígenes, constituyendo la unidad de planificación del trabajo. El programa puede verse descompuesto horizontalmente en instrucciones y verticalmente en *warps*, pero a diferencia de una máquina SIMD *(Single Instruction Multiple Data)* no progresa de forma síncrona en cada una de sus instrucciones, sino que los 32 carriles (hilos) que engloba cada *warp* puede avanzar con distinta velocidad, encontrándose en un mismo punto del tiempo en distintos grados de madurez dentro del programa. Esto permite ocultar la latencia de los *warps* que se encuentren atascados (por ejemplo, esperando un dato de la memoria) mediante otros *warps* que sí pueden progresar en ese mismo instante, al encontrarse en un punto de ejecución diferente. La GPU establece además un mecanismo instantáneo para la conmutación de contexto entre *warps*, lo que constituye una de las claves para la aceleración en GPU: La máquina difícilmente detiene su ejecución útil si, como se espera, el programador define un código descompuesto en multitud de hilos (y *warps*).

- El **bloque** es un grupo de hilos que son mapeados en un único multiprocesador. Dado que cada multiprocesador dispone de múltiples núcleos y un espacio de memoria compartida, los hilos dentro de un mismo bloque comparten eficientemente dicha memoria. Todos los hilos dentro de un mismo bloque se alojan dentro de un mismo multiprocesador compartiendo los recursos de manera equitativa y con una identificación única que permite procesar diferentes conjuntos de datos en una filosofía tipo SIMD, pero que en realidad se define como SIMT (Single Instruction Multiple Thread) por el matiz ya explicado para la ejecución de warps. Esto obliga al programador a vigilar las dependencias de datos entre *warps* de un mismo bloque, ya que si un *warp* lee un dato que escribe otro *warp* de su bloque en una instrucción anterior, deberá colocar una barrera de sincronización explícita (*synchthreads()* en CUDA). Los bloques, en cambio, no pueden sincronizarse, ya que por definición constituyen particiones disjuntas e independientes del espacio de computación (normalmente, iteraciones *forall* o completamente paralelas de un lazo).

- El **kernel** es la porción de código que va a ser ejecutada por cada hilo y que se diferencia en los datos procesados utilizando su identificador unívoco de hilo y bloque para acceder a los mismos, normalmente mediante expresiones que compondrán los índices de acceso a los vectores de datos involucrados dentro de los lazos.

- El **grid** es la composición de los diferentes bloques que son equitativamente distribuidos y planificados entre todos los multiprocesadores. Engloba toda la carga computacional definida por el programador para cada *kernel*, estructurada según una taxonomía de bloques, *warps* por bloques, e hilos por *warp*.

Con todos estos elementos del modelo de programación, el diseño de la aplicación debe declarar explícitamente el número de bloques, el número de hilos y controlar la distribución de la carga de trabajo y sus puntos de sincronización para cumplir con las limitaciones de paralelismo tales como las dependencias de control y datos.

2.4. Evolución de las características y funcionalidades en la arquitectura CUDA

Las plataformas gráficas evolucionan a una velocidad inusitada, habiéndose desarrollado cuatro generaciones desde el nacimiento de CUDA a finales de 2006 hasta 2014: Tesla (2008), Fermi (2010), Kepler (2012) y Maxwell (2014) [54]. Este trabajo hace un recorrido por cada una de ellas de las presentes en el periodo de trabajo de esta tesis, durante el cual Maxwell aún no estaba presente [56]. Los puntos más destacados de la evolución de la arquitectura desde el punto de vista *hardware* y *software* desde Tesla hasta Kepler se exponen a continuación.

Fermi amplió el número de núcleos de GPU hasta los 512, y los de doble precisión hasta los 256. También introdujo las cachés L1 y L2 (la primera de ellas configurable en tamaño junto a la memoria compartida, para una capacidad conjunta de 64 KB.), extendió la cobertura ECC en la corrección de errores para la memoria DRAM, mejoró los cambios de contexto y agilizó las operaciones atómicas.

Kepler, por su parte, presentó un nuevo multiprocesador SMX con 192 núcleos para computación entera y de simple precisión, y 64 núcleos para doble precisión. Inicialmente se comercializaron versiones con 13 y 14 SMXs (K20 y K20X, respectivamente), ampliándose la gama en Noviembre de 2013 a la versión de 15 SMXs (K40, dotada de 2880 núcleos). SMX incorpora dos grandes novedades: Paralelismo dinámico y planificación Hyper-Q[2]. Antes de detenernos en ellas, la Tabla 2.1 realiza una comparativa entre Fermi y Kepler en base a los parámetros que mejor definen las prestaciones de CUDA. El aumento del número de *warps* y bloques que se pueden ejecutar simultáneamente en la nueva arquitectura constituyen la vía de mejora para las aplicaciones regulares y masivamente paralelas. Si éstas carecen de alguna de estas

[2]Una versión preliminar de Kepler, comercializada como K10, no incluía ninguna de estas dos prestaciones.

Generación de GPU	Fermi	Kepler
Modelo hardware	GF100	GK110
Hilos por warp	32	32
Máximo número de warps por multiprocesador	48	64
Bloques activos por multiprocesador	8	16
Máximo tamaño de bloque (en hilos)	1024	1024
Máximo número de hilos por multiprocesador	1536	2048
Máximo número de registros por hilo	63	255
Dimensión máxima de la malla de hilos	$2^{16}-1$	$2^{32}-1$
Paralelismo dinámico	No	Sí
Hyper-Q	No	Sí

Tabla 2.1: Principales rasgos CUDA de las GPU basadas en las generaciones Fermi y Kepler.

cualidades, el paralelismo dinámico y la planificación Hyper-Q ofrecen gran potencial como alternativa en computación irregular, aunque eso sí, de forma más exigente para el programador.

2.4.1. Paralelismo dinámico

En un sistema híbrido CPU-GPU, la ejecución eficiente de aplicaciones con elevado grado de paralelismo depende en gran medida de la versatilidad de los mecanismos que permitan distribuir el trabajo entre ambos aprovechando las mejores cualidades de cada uno de ellos. Hasta 2013, CUDA postulaba a la GPU como un coprocesador esclavo de la CPU que recibía sus encargos y trataba de acelerarlos, pero con una autonomía bastante limitada. El paralelismo dinámico permite a la GPU lanzar sus propios *kernels*, crear los eventos e hilos necesarios para controlar las dependencias, sincronizar los resultados y controlar la planificación de tareas, todo ello sin intervención alguna de la CPU, que ahora puede dedicarse de forma más eficiente a sus propias tareas. Por su parte, la GPU aporta un procesamiento más directo de bucles anidados y algoritmos recursivos, y en general, una computación más natural de código dinámico y estructuras de datos irregulares.

Por ejemplo, ahora es posible determinar en tiempo de ejecución el número de hilos encargados de procesar los nuevos *kernels* creados, pudiendo establecerse una configuración inicial con un paralelismo más conservador que evite cálculos innecesarios en zonas livianas, para aumentar gradualmente el paralelismo en aquellas zonas que vayan requiriendo una computación más exigente.

2.4.2. Hyper-Q

En el modelo CUDA, la CPU lanza los *kernels* sobre la GPU de forma secuencial, esto es, toda la GPU se dedica a procesar el primer *kernel* y hasta que éste no concluye no se inicia el segundo, y así sucesivamente. En los casos en que los *kernels* sean independientes y no quiera articularse esta barrera implícita de sincronización, puede utilizarse el concepto de *streams* para agrupar los *kernels* dependientes en un mismo *stream* y separar los *kernels* independientes en distintos *streams*.

La búsqueda de un planificador óptimo que administre la carga de trabajo en GPU cuando ésta procede de diferentes *streams* es uno de los retos más difíciles de su arquitectura. Fermi permite una ejecución concurrente de hasta 16 *streams*, pero la existencia de una sola cola de trabajo obliga a multiplexar los *streams*, y por tanto, a su serialización. Aunque esta dependencia puede ser aliviada en una primera fase reordenando los *kernels* de cada *stream*, la tarea empieza a ser complicada y el rendimiento disminuye conforme la complejidad de los programas aumenta. Hyper-Q habilita hasta 32 colas de trabajo entre el *host* y el distribuidor de trabajo de CUDA en GPU, dotando de gran flexibilidad al conjunto para lograr grandes mejoras sin modificar la implementación. Ahora, cada *stream* se gestiona desde su propia cola *hardware* de trabajo, sin interferir las dependencias con otros *streams*, que pueden proceder del mismo programa CUDA u otros ubicados en diferentes procesos MPI o hilos POSIX (más conocidos como *p-threads*).

De esta forma, la concurrencia es natural y no requiere preprocesamiento. Además, este mecanismo resulta más potente a medida que aumentamos el número de núcleos de cada multiprocesador, erigiéndose en uno de los pilares para la escalabilidad de las futuras generaciones de GPUs.

Parte II

GPGPU clásica: Programando gráficos

3 Detección de Patrones

La biología celular dispone de multitud de aplicaciones resueltas a través de un análisis detallado de imágenes previamente procesadas con alguna tinción química que resalte la estructura de los tejidos. Un ejemplo es la regeneración esquelética objeto de este capítulo, donde se pretende descubrir el microambiente que mejor induce la expresión de proteínas de matriz esquelética, responsables de diferenciar entre cartílago y hueso inducido a partir de células madres mesenquimales pluripotentes inyectadas en la región objeto de una lesión ósea.

La evaluación del fenotipo de las imágenes tratadas con células para controlar los cambios producidos en el organismo se lleva a cabo con el recuento de células en cada tipo de tejido: cartílago, hueso y fibra. Este proceso pretende ser semi-automático para que el experto en la materia pueda apoyar su criterio en unos parámetros computables. De otra manera, el procedimiento se torna tedioso, sensible al observador y tremendamente costoso. Entre las técnicas de reconocimiento de patrones, que pueden ser clasificadas en dos familias dependiendo de si se emplea un algoritmo local o global, se ha seleccionado la transformada de Hough por ser una técnica de reconocida robustez, incluso con la presencia de solapamiento de formas y ruido.

En este capítulo, se propone la implementación en GPU de la transformada de Hough para la detección de círculos junto con una serie de optimizaciones que permiten mejorar el rendimiento y maximizar el uso de las unidades funcionales del cauce de segmentación clásico de las arquitecturas gráficas.

Esta implementación pone de manifiesto el potencial que ofrecen los procesadores de vértices, en desuso en una programación de propósito general en GPU, para problemas con unas determinadas características. Con el beneficio adicional de situarse en la fase inicial del cauce de segmentación de la GPU, permitiendo en una misma

fase de renderizado aprovechar otras unidades funcionales como los procesadores de píxeles y las unidades de *blending* (mezclado).

El contenido de este capítulo se estructura de la siguiente forma. La Sección 3.1 describe el algoritmo completo para la detección de círculos así como el preprocesado necesario. La Sección 3.2 presenta el rasterizador de la GPU, que va a jugar un papel fundamental en las optimizaciones de la implementación en GPU de la transformada de Hough para detectar círculos (THC). Esta transformada se describe en la Sección 3.3, mientras que la Sección 3.4 analiza el impacto de las características de la textura usada como estructura de datos de entrada y salida. La Sección 3.5 estudia los parámetros que afectan a las diferentes implementaciones, mientras que la Sección 3.6 evalúa la precisión. Finalmente, las Secciones 3.7 y 3.8 presentan una serie de optimizaciones para GPU y CPU, respectivamente, cuyos resultados se exponen en la Sección 3.9. Finalmente, la Sección 3.10 resume las principales contribuciones de este capítulo.

3.1. El algoritmo

La transformada Hough [5] es un método ampliamente utilizado para detectar círculos y contornos de diversas formas en el campo del procesamiento de imágenes. En la transformada Hough se parte de un objeto o figura objetivo y una imagen donde ha de realizarse la búsqueda del objeto. Posteriormente, se analizan todas las coincidencias del objeto en la imagen, que son caracterizadas como votos en un espacio de parámetros que representa la probabilidad de que el objeto exista en la imagen en una localización particular.

Una tarea tan sencilla para el ojo humano resulta dificultosa y compleja por no disponer a priori de un patrón predefinido para todos los contornos en una imagen. La complejidad y el uso de memoria [24] se disparan a tal nivel que convierte dicho problema en fiel candidato para estudiar su paralelización. La automatización de dicha tarea se ha abordado desde diversas arquitecturas paralelas, como los sistemas multiprocesadores con memoria distribuida [26, 43, 78], computadores piramidales [3], máquinas SIMD [45], hardware de propósito específico [12] y arquitecturas reconfigurables [59, 61]. De carácter más reciente, la GPU ha sido igualmente empleada por su gran aceptación para la resolución de problemas exigentes computacionalmente [58].

Aunque el reconocimiento de círculos recae en la transformada Hough, previamente se aplica un preprocesado para aislar los puntos de interés o contornos en la imagen, eliminando así el ruido de las imágenes de entrada.

3.1.1. Detección de contornos: Operador Canny

Los métodos de detección de contornos utilizan los operadores gradiente como herramienta básica de procesado [65]. Entre todas las posibilidades, el operador Canny [14] está consolidado como una gran alternativa por sus excelentes resultados en imágenes con escala de grises; para imágenes en color, se requiere una conversión inicial aplicando el operador luminancia.

El operador Canny es un procedimiento con varias fases que se detallan a continuación:

1. Primera convolución. Los contornos de la imagen son suavizados al aplicar el operador gaussiano que controla la cantidad de detalle en los contornos a la vez que elimina el ruido.
2. Segunda convolución. El operador Sobel aplicado en una imagen bidimensional y basado en la primera derivada, detecta los contornos gracias al gradiente de la intensidad de cada píxel.
3. Eliminación de los no-máximos. Los contornos detectados cuya intensidad no coincide con los máximos locales del gradiente se eliminan para disponer de contornos limpios.
4. Histéresis de doble umbral. Los valores de intensidad considerados en la imagen final son acotados a través de dos umbrales. Con dos umbrales, T1 >T2, los contornos aceptados serán aquellos que dispongan de algún píxel con un valor de intensidad por encima del umbral T2 y los demás valores por encima de T1.

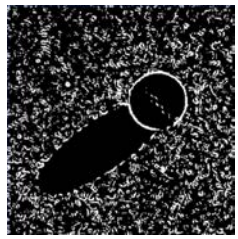
3.1.2. Detección de patrones: La transformada Hough

Una vez aplicado el filtro de Canny, los puntos o píxeles que describen los contornos forman los datos de entrada para la transformada Hough. Cada punto genera la figura a detectar como votos en un espacio de parámetros en el cual los máximos de la acumulación de todos los votos desvelan las coordenadas de las figuras detectadas (ver Figura 3.1). En el caso particular de la transformada Hough para la detección de círculos [43], en el que se centra este capítulo, cada punto genera una superficie tridimensional que describe la siguiente ecuación:

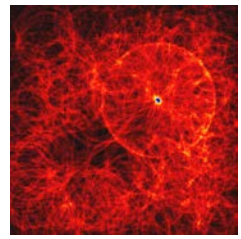
$$(x-a)^2 + (y-b)^2 = r^2 \tag{3.1}$$

Donde (x, y) son las coordenadas de cada punto, y (a, b) y r representan el centro del círculo y el radio, respectivamente. Basando la detección en círculos con un radio

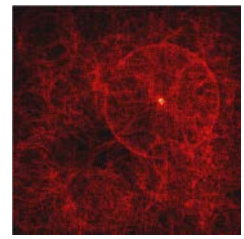
(a) Imagen de muestra



(b) Detección de contorno



(c) Salida con GL_POINTS y $\delta t = 0{,}01$



(d) Salida con GL_POINTS y $\delta t = 0{,}1$

Figura 3.1: (a) Imagen de muestra. (b) Detección de contornos y entrada a la transformada Hough después de aplicar el operador Luminancia y el filtro Canny. (c-d) Salida de la GPU donde el *frame buffer* acumula todos los votos del espacio parametrizado bidimensional.

predefinido r, el proceso global empieza con una transformación entre un espacio de coordenadas (x, y) y el espacio de parámetros (a, b) discretizado, donde cada punto se convierte en el centro de una circunferencia de radio r que incrementa el número de votos por cada celda que pasa en el espacio de parámetros (a, b). Desglosando la ecuación 3.1 en cada eje, la conversión entre ambos espacios vendría definida por el siguiente par de ecuaciones:

$$\left.\begin{array}{l} a = x - r * cos(t) \\ b = y - r * sin(t) \end{array}\right\} t \in [0, 2 * \pi] \tag{3.2}$$

Con la descripción dada, es necesario en una fase inicial decidir la discretización para el ángulo t y el radio r de acuerdo a las características de la imagen. Conforme aumente la granularidad del ángulo, la carga computacional y la precisión aumentan, así como el tiempo de ejecución y la calidad de la detección de los círculos que son detectados en el espacio de parámetros como aquellos puntos votados con un valor superior a un umbral determinado.

La disminución del paso de ángulo, δt, no siempre implica una mejor detección de círculos. Conforme disminuye la distancia entre votos, el peso porcentual de cada voto disminuye al igual que se distribuyen de una forma difuminada alrededor del centro real del círculo, dificultando la localización de un único punto como máximo local (ver Figura 3.1c con un δt superior a la Figura 3.1d).

Existen métodos basados en la información del gradiente para la THC con unas exigencias más relajadas para la carga computacional y la memoria [21] pero que desafortunadamente sufren de imprecisión conforme el ruido es mayor [20]. El algoritmo

clásico es menos sensible al ruido a costa de una mayor carga computacional, y el objetivo de este trabajo es desarrollar una implementación mucho más eficiente en GPU que permita aprovechar la robustez del algoritmo en tiempos de ejecución razonables.

3.2. Impacto del rasterizador de la GPU

Los datos de entrada para el cauce de segmentación gráfico consisten en un conjunto de comandos, texturas y vértices que son mandados desde la CPU a través del bus PCI-express. Haciendo uso de la librería gráfica OpenGL, los comandos inicializan y modifican estados, configuran renderizados y referencian vértices y píxeles; los vértices y los estados fluyen en la GPU a través de una secuencia de fases del cauce de segmentación (ver Figura 2.1):

1. El procesador de vértices permite la programación de cada vértice a través de un programa dedicado, denominado *shader* o *sombreador* de vértices.

2. Los vértices son agrupados en primitivas que pueden definir diferentes formas, bien puntos, líneas o triángulos. Los bloques de *Cull*, *Clip* y *Setup* optimizan las operaciones de cada primitiva, eliminando caras no visibles, recortando la parte de la imagen no mostrada y desarrollando la configuración de las ecuaciones para los planos y contornos, respectivamente.

3. El rasterizador expande los datos hacia unos valores interpolados para generar todos los píxeles que cubren cada primitiva, dando un conjunto de fragmentos como resultado [49].

4. Los píxeles candidatos (fragmentos) son aceptados por el procesador de fragmentos donde se aplica otro *shader* para realizar las transformaciones oportunas a cada píxel.

5. Los fragmentos resultantes son utilizados para escribir los resultados finales en el *frame buffer* (un array bidimensional de píxeles presentados en pantalla) o en una textura 2D como alternativa de almacenamiento.

La lista de primitivas que puede emplear el rasterizador de la GPU para interpolar líneas y completar áreas está resumida en la Tabla 3.1 para el caso de OpenGL, con su funcionalidad mostrada en la Figura 3.2. Para el propósito de implementar la transformada Hough en GPU, las primitivas con relleno de área son descartadas y, por tanto, la selección se limita entre GL_POINTS y GL_LINES. GL_POINTS genera

píxeles desde vértices de entrada aislados, mientras que GL_LINES habilita el proceso de interpolación con la unión de los vértices a través de líneas rectas. La librería gráfica ofrece un total de seis métodos de interpolación [49], entre los cuales hemos seleccionado el método lineal por su excelente combinación entre una gran precisión y una baja complejidad computacional.

Primitiva en OpenGL	Funcionalidad
GL_POINTS	Cada vértice se representa de manera aislada.
GL_LINES	Cada par de vértices forma una línea recta.
GL_LINE_STRIP	Los vértices forman una secuencia de segmentos.
GL_LINE_LOOP	Une el primer y último vértice en GL_LINE_STRIP.
GL_TRIANGLES	Cada tres vértices consecutivos forman un triángulo.
GL_TRIANGLE_STRIP	Similar a GL_TRIANGLES pero con triángulos contiguos.
GL_TRIANGLE_FAN	Todos los triángulos formados comparten el primer vértice.
GL_QUADS	Cada cuatro vértices consecutivos forman un cuadrilátero.
GL_QUAD_STRIP	Igual que GL_QUADS, pero con cuadriláteros contiguos.
GL_POLYGON	Un conjunto de N vértices forman un polígono convexo.

Tabla 3.1: Lista de las primitivas OpenGL para unir vértices.

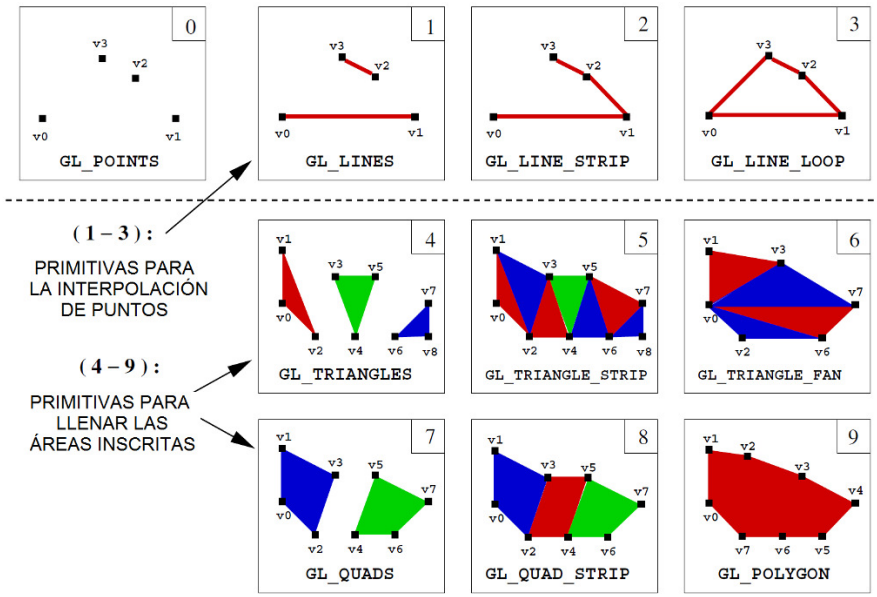


Figura 3.2: Primitivas OpenGL disponibles para que el rasterizador de la GPU haga la interpolación de una lista de vértices.

El procesamiento en GPU basado en líneas puede llevarse a cabo con más paralelismo debido a la gran cantidad de procesadores de píxeles disponibles. OpenVidia ya reveló en un análisis [22] que el procesamiento es aproximadamente tres veces más

rápido cuando se usan las primitivas de líneas frente a las de puntos.

3.3. Implementación en GPU

La transformación computacional del algoritmo de la THC para mapear el algoritmo en GPU se ha desarrollado con OpenGL como librería gráfica y Cg para la programación de *shaders*. El proceso se puede descomponer en tres o cuatro fases según se considere la intervención de la CPU (ver Figura 3.3):

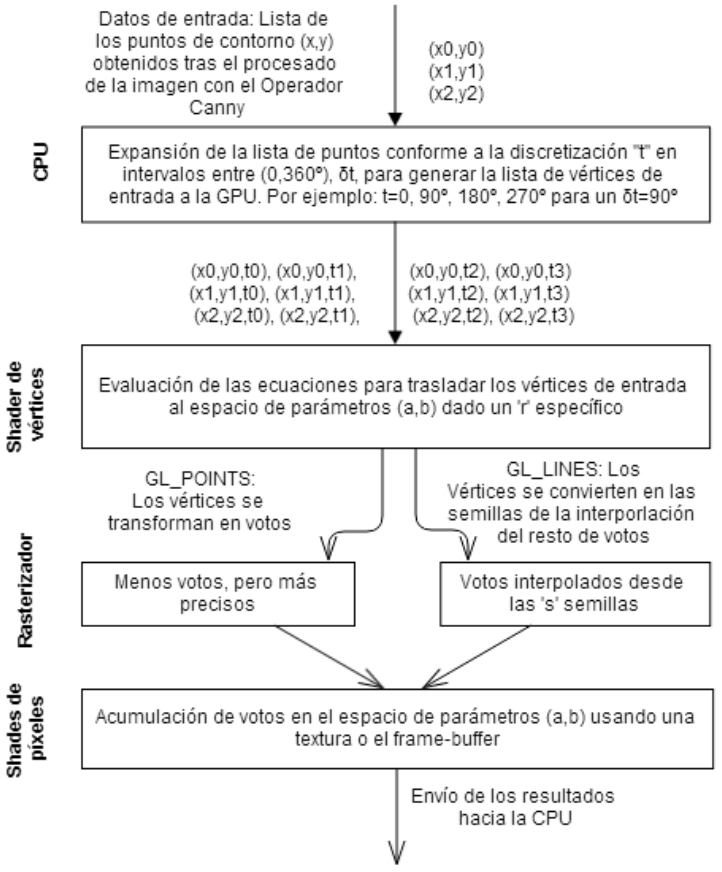


Figura 3.3: Descripción de la transformada de Hough en GPU.

1. El procesador de vértices traslada el espacio de la imagen hacia un espacio parametrizado, transformando las posiciones (x, y, t) en votos (a, b).

2. Aplicando la primitiva GL_LINES, el rasterizador genera el espacio de votos completo a través de un pequeño número de s votos llamados semillas. El resultado consiste en un continuo intervalo interpolado en el rango $[0, 2\pi]$ procedente de la previa discretización, con una área pequeña de error producida por el hecho de que cada voto en el espacio es una aproximación a través un polígono de s lados (ver el área rayada en la Figura 3.4). En este trabajo se realiza un análisis analítico para elaborar las fórmulas que determinan el mínimo valor s que garantiza que el máximo error se mantenga por debajo del umbral de la distancia de píxel en el *frame buffer*, y se verifican empíricamente los resultados para un conjunto variado de imágenes. Los resultados finales de votado indican que empezar con un valor de $s = 8$ es aceptable en cuanto a consistencia.

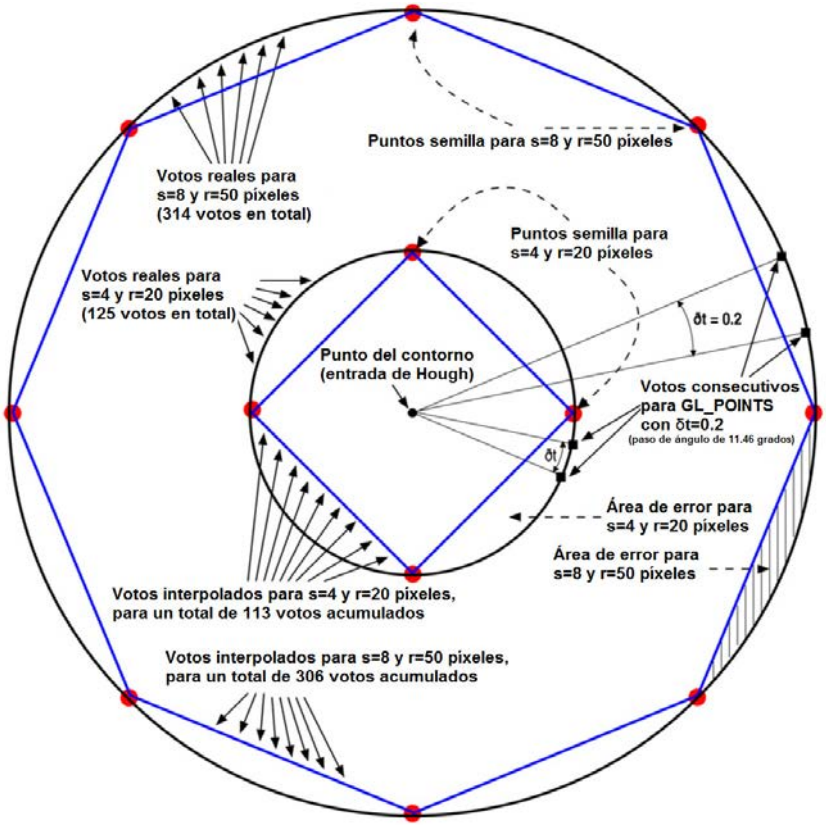


Figura 3.4: Proceso de interpolación sobre un único punto del contorno de los votos generados por el rasterizador de la GPU.

Para el caso de la primitiva GL_POINT, el rasterizador no interviene en el proceso y, por tanto, no existe la interpolación de votos; los únicos votos en el espacio de parámetros son los creados durante la discretización, con su precisa

localización y separados por un paso de ángulo determinado δt.

3. El conjunto de votos de cada punto de contorno se almacena en una textura 2D, cuyas dimensiones dependen de la discretización del espacio de parámetros. Los votos se acumulan por un *shader* de píxeles como si se tratara de la reducción de una textura 3D (la tercera componente es el ángulo) hacía una textura 2D final que se almacena en el *frame buffer* (ver Figura 3.5). Las extensiones de OpenGL usadas para un manejo eficiente de todo el proceso son las de renderizado a textura y los objetos de *frame buffer* FBO (ver Figura 3.6).

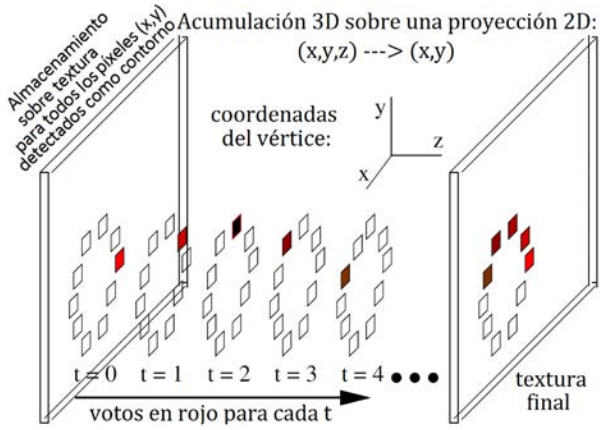


Figura 3.5: Proceso de acumulación de votos sobre una textura gráfica.

Conviene destacar que la resolución de la THC en GPU se va a llevar a cabo en el procesador de vértices por las apropiadas características para este problema, mientras que la tendencia de la programación de propósito general en GPU aboga principalmente por explotar el procesador de píxeles al estar dotado de un mayor paralelismo y cercanía a la memoria de vídeo.

3.4. Discretización y resolución de la textura

De acuerdo a la Ecuación 3.2, todos los votos por cada punto de entrada (x, y) describen un círculo en el espacio de parámetros (a, b) después de calcular todas las posibles instancias del ángulo t comprendido entre 0 y π (ver Figura 3.7a). El número de votos emitidos dependen de δt (ver columnas 2 y 3 en Tabla 3.2), donde la fidelidad del proceso de votado recae en la estructura de datos que almacena todos los votos para el espacio de parámetros (a, b). Mientras que en la CPU esta estructura es una matriz bidimensional, en la GPU se emplea una textura 2D. En ambos casos, una

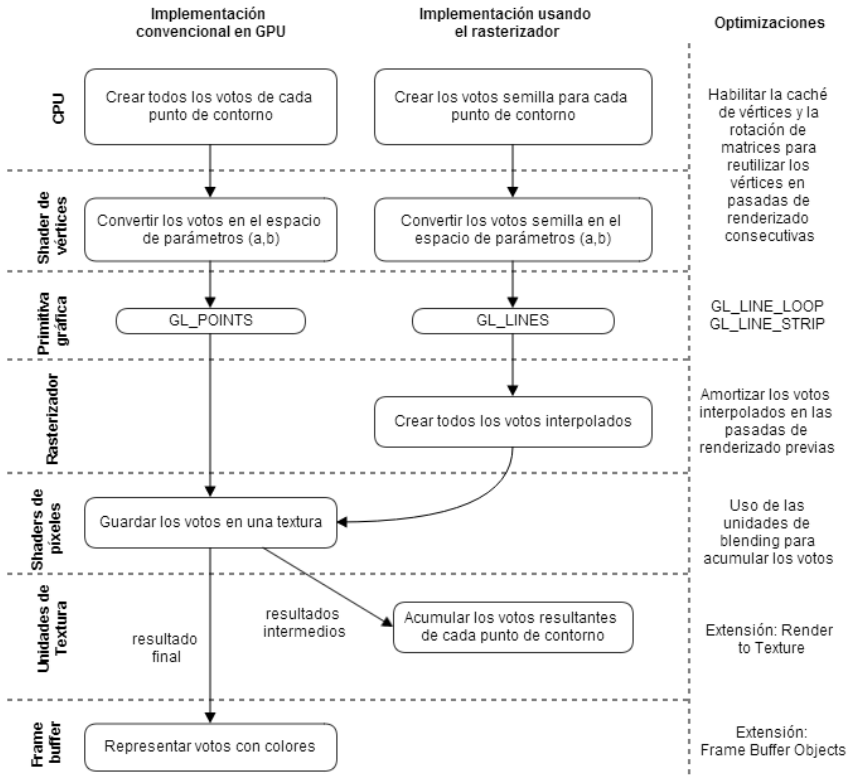


Figura 3.6: Unidades funcionales del cauce de segmentación gráfico empleadas para la implementación de la THC en GPU.

estructura mayor implica una mayor precisión y carga computacional. En este sentido, aparecen los algoritmos multiresolución [4] para optimizar la relación carga de trabajo/precisión.

Radio	GL_POINTS (δt)		GL_LINES (s)			
	δt óptimo: ($\frac{1}{r}$)	votos: ($2\pi r$)	Para s superiores, más votos: $(2 \cdot r \cdot s \cdot \sqrt{1-\cos^2(\frac{\pi}{s})})$ s= 4	s = 8	s = 16	s = 32
r = 10	δt = 0.10	62	56 (90.4 %)	61 (98.3 %)	62 (100 %)	62 (100 %)
r = 20	δt = 0.05	125	113 (90.4 %)	122 (97.6 %)	124 (99.2 %)	125 (100 %)
r = 50	δt = 0.02	314	282 (89.8 %)	306 (97.4 %)	312 (99.3 %)	313 (99.6 %)
r = 100	δt = 0.01	628	565 (89.9 %)	612 (97.4 %)	624 (99.3 %)	627 (99.8 %)

Tabla 3.2: Número de votos para las diferentes implementaciones en GPU.

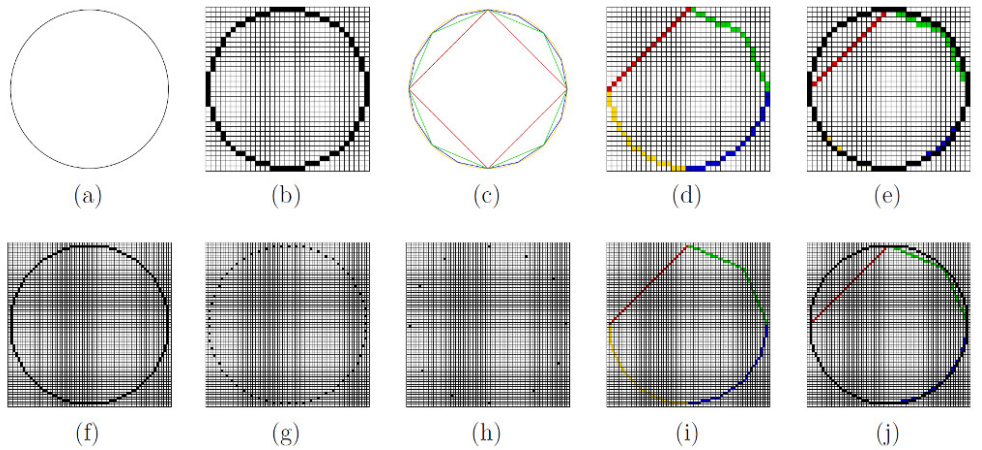
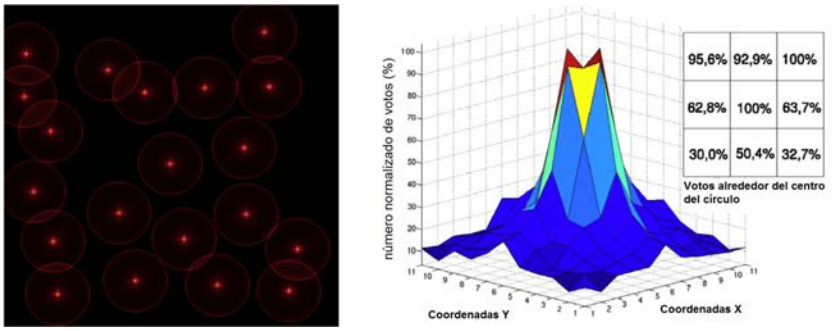


Figura 3.7: Espacio parametrizado de la transformada Hough como resultado del rasterizador de la GPU sobre una textura. (a-e) son los casos de baja resolución; (f-j) corresponden con los casos de alta resolución. La interpolación de los píxeles depende de la primitiva seleccionada, GL_POINTS o GL_LINES, al igual que el número de vértices procesados como función de s o δt y comportándose como semillas del polígono representado.

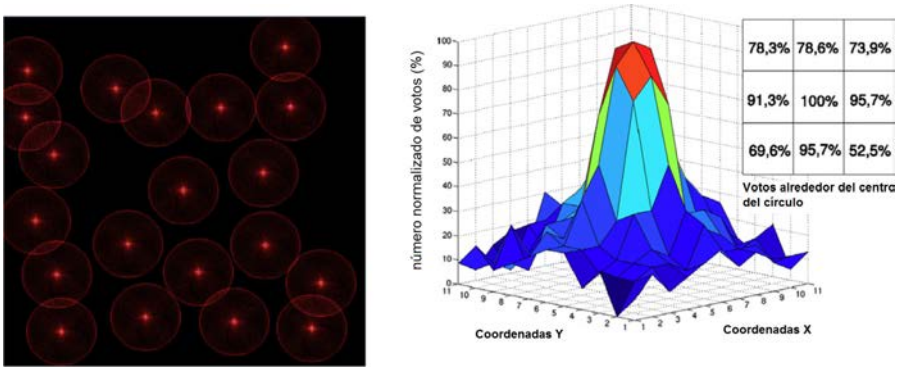
3.4.1. Análisis de resultados

La salida a pantalla de la tarjeta gráfica es un dominio espacial finito que ofrece la posibilidad de mostrar la distribución de votos de una manera gráfica e intuitiva (ver imágenes de la izquierda en Figura 3.8) que permite a los usuario el seguimiento en las aplicaciones de reconocimiento de círculos. Sin embargo, cada operación interna se implementa a través de los búferes internos que ofrece la extensión *Frame Buffer Object* (FBO) de OpenGL para renderizar los resultados en la memoria de vídeo en lugar del *frame-buffer* convencional. De esta forma, las limitaciones asociadas al *hardware* gráfico (resolución máxima de pantalla) y al *software* (tamaño máximo de textura) son remplazadas por las existentes en las texturas 2D del FBO, cuyo tamaño únicamente depende del grado de discretización.

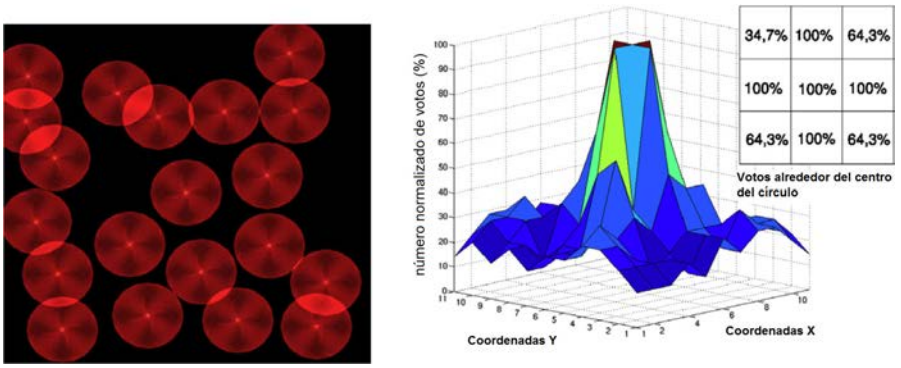
Para obtener una representación más precisa de los resultados, la información alrededor de las coordenadas detectadas como círculos en el espacio de parámetros se transmite hacia CPU para que se obtenga una representación en forma de malla (ver parte derecha de la Figura 3.8).



(a) Espacio de parámetros ideal. GL_POINTS con $\delta t = 0.02$.



(b) Reducción de la carga computacional de la GPU. GL_POINTS con $\delta t = 0.1$.



(c) Uso del rasterizador de la GPU. GL_LINES con s = 16.

Figura 3.8: Salidas de las diferentes estrategias implementadas para la THC usando una imagen de 1024x1024 píxeles que contiene 20 círculos con un radio de 50 píxeles. Las imágenes de la izquierda reflejan el potencial del *frame-buffer* a la hora de mostrar una salida visual que pueden orientar a los usuarios en las aplicaciones de reconocimiento semi-automático de círculos. Las gráficas de la parte derecha representan el espacio parametrizado alrededor de las coordenadas del centro de un círculo detectado; además una matriz de 3 x 3 muestra el porcentaje de los votos acumulados en los ocho píxeles vecinos al centro del círculo (el máximo valor de votado es de 100 %).

3.4.2. Déficit de votos

Dada la discretización en GPU orientada hacia una textura bidimensional formada por píxeles, cuando un círculo tiene un radio de 10 píxeles, su perímetro, $2\pi r$, estará formado por $2\pi 10 = 62{,}83$ píxeles. Este valor será redondeado a 63 o truncado hacia 62 píxeles, pero además el perímetro en ningún caso medirá el valor exacto proporcionado por la fórmula porque la distancia entre dos píxeles consecutivos difiere dependiendo de la alineación (horizontal o vertical) y, en caso de que estén situados en zonas del contorno donde forman una diagonal, aún estarían más lejanos y la contribución al perímetro sería mayor. Este déficit de votos encuentra su caso peor para la búsqueda de grandes círculos en resoluciones de textura más pobres. En la práctica, como una resolución mayor ofrece más píxeles a cada círculo, aumentar el tamaño del círculo o de la resolución de la textura ofrecen resultados similares en este sentido. Por ejemplo, el radio de la Figura 3.7b contiene 16 píxeles y el círculo completo está compuesto por un total de 92, cuando el valor teórico es de $2\pi 16 = 100{,}53$ píxeles (déficit del 8 %); por otro lado, el radio de la Figura 3.7f es de 32 píxeles y el círculo está compuesto por 184, que difiere del valor teórico $2\pi 32 = 201{,}06$ píxeles en un 8 %. El déficit esperado en el proceso de discretización puede ser moderado con la activación del modo antialiasing que suaviza los contornos con el uso del rasterizador.

Para situaciones que requieran texturas de alta resolución, pueden aplicarse estrategias multiresolución para acelerar los tiempos de ejecución. La ejecución se daría desde un nivel inicial de baja resolución hasta una resolución final fijada, de modo que en cada resolución se retiene información valiosa para lograr la detección. En este sentido, la implementación que planteamos para la transformada Hough en GPU es perfectamente válida para poder elegir entre relajar la carga computacional o lograr un mayor grado de precisión.

3.4.3. Efectos colaterales

La resolución de la textura afecta a otros aspectos de los métodos que usan las primitivas GL_POINTS y GL_LINES para calcular los votos en GPU:

1. Una baja resolución produce una deficiencia de votos. Se puede aprovechar la discretización de la textura en píxeles para seleccionar un incremento de ángulo, δt, que sólo genere votos en píxeles diferentes de acuerdo a la Ecuación 3.2, para así relajar la carga computacional sin afectar a la fidelidad de los resultados.

2. Una gran discretización para el ángulo puede provocar la pérdida de votos cuando se usa GL_POINTS. Para una combinación sensata entre buena precisión y

baja carga computacional, se puede querer mantener la resolución de la textura mientras se incrementa la discretización del ángulo para producir puntos más dispersos en la textura bajo la primitiva GL_POINTS (ver Figura 3.7g y 3.7h para incrementos de δt de 0.1 y 0.5). La Sección 3.5 comprueba que la discretización puede ser más agresiva en círculos pequeños.

3. Un bajo número de semillas para GL_LINES repercute en la precisión de los votos. La Figura 3.7c. muestra este efecto para una serie de semillas s = 32, 16, 8, 4; las líneas generadas por el proceso de interpolación en GL_LINES recuperan los votos situados entre las semillas calculadas. Dicha recuperación es menos precisa conforme disminuye el número de semillas. El papel de s es analizado en la Sección 3.5 desde el punto de vista de la precisión, y la Figura 3.9 muestra, en color negro, el área de error para la serie s = 4, 8, 16, 32. Las resoluciones más pequeñas afectan de igual manera tanto a los círculos procesados como a los interpolados.

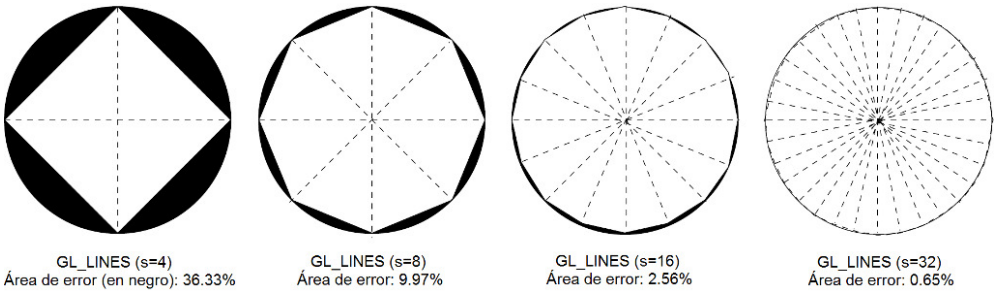


Figura 3.9: Reducción del área de error al aumentar s en un factor 4x desde su valor inicial hasta un valor de 32. El área de error se calcula dividiendo el área circular, πr^2, por el área del polígono inscrito, $sr^2 \sin(\frac{\pi}{s}) \cos(\frac{\pi}{s})$.

3.5. Parámetros clave

Desde una perspectiva cuantitativa, los parámetros determinantes para las implementaciones en GPU usando las primitivas GL_POINTS y GL_LINES son el incremento del ángulo δt y el número de vértices por círculo, respectivamente. En la Sección 3.6 se hace un estudio desde un punto de vista cualitativo.

3.5.1. El papel de δt

Para la selección de un δt óptimo para la transformada Hough a través de la primitiva GL_POINTS, tiene que existir una correspondencia unívoca entre los votos generados por la Ecuación 3.2 y la posición de los puntos del contorno. Con una resolución suficiente para garantizar esta condición, el número de votos bajo una cierta discretización δt es de $\frac{2\pi}{\delta t}$, y el número de píxeles en el espacio de discretización para un círculo de radio r debería ser de $2\pi r$. El valor calculado bajo estas premisas es de $\delta t = \frac{1}{r}$, que revela una relación inversamente proporcional entre el tamaño de los círculos a detectar y la discretización del ángulo. De esta forma, la resolución del ángulo y la carga computacional se intensifican con el tamaño del círculo a detectar por la transformada Hough. El lado izquierdo de la Tabla 3.2 muestra algunos valores de δt con el número de votos procesadores para círculos de radio r = 10, 20, 50, 100.

3.5.2. El papel de s

La parte derecha de la Tabla 3.2 muestra la influencia positiva del incremento de s para el beneficio del recuento de votos cuando los círculos en el espacio de parámetros son creados a través de interpolación con la primitiva GL_LINES. Aunque el número exacto de votos de la circunferencia debe ser el tamaño de la circunferencia $2\pi r$, el número real para los círculos interpolados corresponde con el sumatorio del tamaño de los lados de los s polígonos regulares que describen la circunferencia,

$$\text{Perímetro} = (\text{Número de lados}) \cdot (\text{Tamaño de cada lado}) = s \cdot 2y \tag{3.3}$$

La Figura 3.10 representa el caso particular de $s = 8$, donde el tamaño de cada lado, $2y$, puede calcularse mediante la siguiente expresión:

$$\begin{gathered} r^2 = y^2 + x^2 \Rightarrow r^2 = y^2 + (r \cdot cos(\frac{\pi}{s}))^2 \Rightarrow \\ y = \sqrt{r^2 - (r \cdot cos(\frac{\pi}{s}))^2} \Rightarrow y = r \cdot \sqrt{1 - cos^2(\frac{\pi}{s})} \end{gathered} \tag{3.4}$$

Con s como el número de lados y $2y$ como tamaño de cada lado, el perímetro del polígono proporciona la siguiente cantidad de votos:

$$\#\text{votos} = \text{Perímetro polígono} = s \cdot 2y = 2 \cdot r \cdot s \cdot \sqrt{1 - cos^2(\frac{\pi}{s})} \tag{3.5}$$

Esta expresión refleja el déficit en el número de votos. Un aumento del déficit al incrementar el tamaño de la circunferencia puede ser compensado con el incremento del número de semillas, s (ver Figura 3.9). Al igual que en el análisis de δt, la resolución seleccionada deber ser suficiente para minimizar el déficit debido a las diferentes longitudes por la localización entre píxeles vecinos (vertical, horizontal y diagonal).

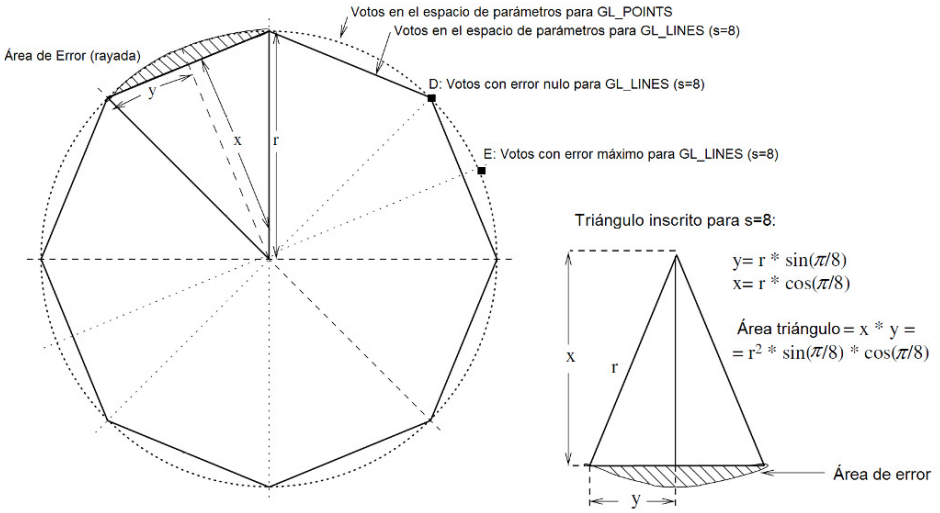


Figura 3.10: Cálculo de error producido cuando la transformada Hough se lleva a cabo a través de la primitiva GL_LINES para el caso particular de s=8.

3.6. Precisión de la interpolación

Como GL_POINTS garantiza una correcta localización de cada punto en el espacio de parámetros, el número máximo de votos que una textura puede alojar supone la restricción más importante para seleccionar el δt óptimo. Sin embargo, aunque s repercute en el número de votos usando GL_LINES, el proceso de interpolación proporciona una cantidad que difiere del valor teórico y que se acentúa con un número menor de semillas. El resto de esta sección analiza la precisión del rasterizador en el proceso de interpolación de votos.

Por otro lado, la resolución de la textura del espacio de parámetros supone otra componente para la precisión como en cualquier proceso de discretización del dominio espacial. Sin embargo, para el estudio de la precisión se asume que la resolución de la textura es óptima, dicho valor se puede cuantificar como el error entre la diferencia de las áreas descritas por la población ideal de votos y los votos reales (ver Tabla 3.3).

Radio - Error medio	s = 4	s = 8	s = 16	s = 32
r = 10 píxeles	2.018	0.511	0.096	0.032
r = 20 píxeles	4.036	1.022	0.192	0.064
r = 50 píxeles	10.090	2.557	0.480	0.160
r = 100 píxeles	20.180	5.114	0.960	0.321
Error medio por voto como porcentaje de r	20.18 %	5.11 %	0.96 %	0.32 %

Tabla 3.3: Tamaño medio del error por voto (en píxeles) para un radio r de un posible centro del círculo cuando la votación en el espacio de parámetros se realiza con GL_LINES con un número concreto de semillas, s.

3.6.1. Estimación del error

El error total para la fracción perteneciente a un polígono de los s formados para generar el círculo corresponde a la diferencia entre el área de dicha porción del círculo y la recogida por el polígono formado (ver Figura 3.10),

$$\begin{aligned}\text{Error total} = \text{Error área} &= (\text{área sector circular}) - (\text{área triángulo}) \\ &= \frac{\pi \cdot r^2}{s} - r^2 \cdot sin(\frac{\pi}{s}) \cdot \cos(\frac{\pi}{s}) = r^2 \cdot (\frac{\pi}{s} - \sin(\frac{\pi}{s}) \cdot \cos(\frac{\pi}{s}))\end{aligned} \tag{3.6}$$

El error por voto es nulo justo en las semillas (como D en Figura 3.10), y alcanza su máximo valor en el punto equidistante entre dos semillas contiguas (como E en la Figura 3.10); dicho valor máximo se puede cuantificar como la diferencia entre el radio del círculo y la apotema de los polígonos generados usando s semillas,

$$r - r \cdot \cos(\frac{\pi}{s}) \tag{3.7}$$

Para obtener el error medio por cada voto procesado, o lo que es lo mismo, la altura media del área subrayada en la Figura 3.10, el área del error tiene que ser dividida por el tamaño del lado $2y$:

$$\begin{aligned}\text{Error medio por voto} = \frac{\text{área de error}}{2 \cdot y} &= \frac{\frac{\pi \cdot r^2}{s} - (r^2 \cdot \sin(\frac{\pi}{s}) \cdot \cos(\frac{\pi}{s}))}{2 \cdot r \cdot \sin(\frac{\pi}{s})} \\ = r \cdot \frac{\frac{\pi}{s} - (\sin(\frac{\pi}{s}) \cdot \cos(\frac{\pi}{s}))}{2 \cdot \sin(\frac{\pi}{s})} &= r \cdot (\frac{\pi}{2 \cdot s \cdot \sin(\frac{\pi}{s})} - \frac{\cos(\frac{\pi}{s})}{2}) \\ &= \frac{r}{2} \cdot (\frac{\pi}{s \cdot \sin(\frac{\pi}{s})} - \cos(\frac{\pi}{s}))\end{aligned} \tag{3.8}$$

3.6.2. Cota del error

Como muestra la Figura 3.9, el número de semillas aplicadas en el rasterizador juega un papel fundamental para reducir el error en la interpolación de votos a través de líneas rectas (polígonos circunscritos dentro de la circunferencia). Este comportamiento se define mejor en la gráfica que acompaña a la Tabla 3.4, donde se representa el error máximo por voto frente al número de semillas, mostrando un decremento exponencial conforme aumenta el radio del círculo.

Un umbral interesante aparece cuando el error máximo supera el valor correspondiente a la distancia de un píxel (ver Tabla 3.4 - línea horizontal sobre la gráfica). Por encima de este punto, no merece la pena incrementar el número de semillas porque el proceso de interpolación está enfocado hacia una representación en pantalla con la limitación del tamaño de píxel real. Puesto que un aumento del número de semillas repercute en la carga computacional de la GPU sin lograr unos resultados más precisos, este punto representa el valor óptimo para s cuando se aplica el rasterizador. Dicho valor, acotado por un error máximo de un píxel, se obtiene mediante la siguiente expresión:

$$r - r \cdot \cos(\frac{\pi}{s}) < 1 \Rightarrow (1 - \cos(\frac{\pi}{s})) \cdot r < 1 \Rightarrow cos(\frac{\pi}{s}) > \frac{r-1}{s} \qquad (3.9)$$

La Tabla 3.4 muestra el número de semillas que cumplen la condición anterior para diferentes tamaños de círculo. El número de semillas empleadas por el rasterizador es siempre lo suficientemente bajo para garantizar un gran rendimiento computacional de la transformada Hough en términos de tiempo de ejecución, tal y como se verá en la Sección 3.9.

3.7. Optimizaciones en GPU

Una desventaja de la implementación básica usando texturas bidimensionales para almacenar los votos es que todos los votos coincidentes que proceden de diferentes círculos o polígonos cuentan como un único voto. Este suceso es una consecuencia del empleo de primitivas básicas dentro de la misma pasada de renderizado. Para evitar esta pérdida de votos, el proceso de renderizado debe descomponerse en varias fases de renderizado, donde cada una de ellas sólo contiene puntos de contornos que se van a solapar en el espacio de parámetros. De esta forma, los s polígonos que representan el espacio de votado se renderizan alineados en una malla 2D, estando sus centros separados por $2r + 1$ píxeles, tanto en la componente horizontal como vertical (ver Figura 3.11a). Esta implementación hace uso de la primitiva GL_LINE_LOOP y

Tamaño radio	Min. semillas
r = 20 píxeles	s = 10
r = 50 píxeles	s = 16
r = 100 píxeles	s = 23
r = 200 píxeles	s = 32

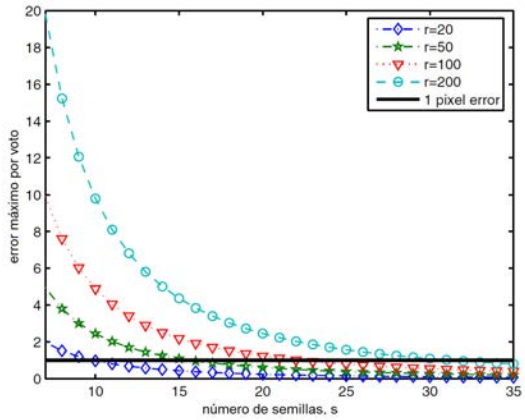


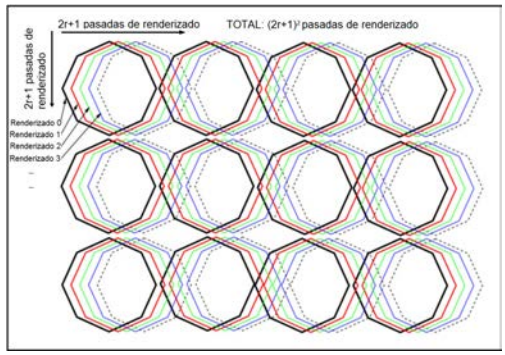
Tabla 3.4: Número mínimo de semillas, s, para mantener el error máximo por debajo de la distancia de un píxel para diferentes tamaños de radio. La gráfica de la derecha representa el error en píxeles cuando se incrementa s para todos los radios.

precisa de $(2*r+1)^2$ pasadas de renderizado, consiguiendo un rendimiento mediocre para radios de gran tamaño.

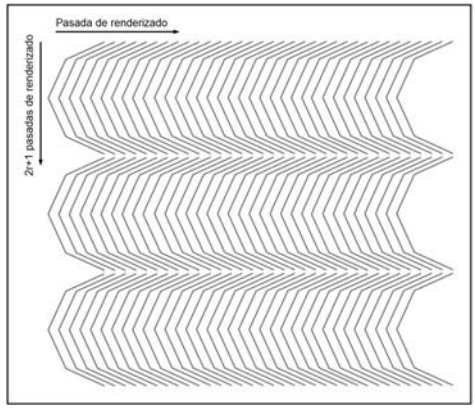
Estrategia implementada	Número de pasadas de renderizado	Estimación del rendimiento	Precisión de la GPU GeForce 7	GeForce 8
Caso base	$(2r+1)^2$	Deficiente	Excelente	Excelente
Agrupando semicírculos	$2(2r+1)$	Razonable	Excelente	Excelente
Juntar canales de color	$\frac{2(2r+1)}{3}$	Bueno	Bueno	Excelente
Mezclado de votos	1	Excelente	Deficiente	Razonable

Tabla 3.5: Conjunto de optimizaciones empleadas con el rasterizador de la GPU en términos de precisión y eficiencia.

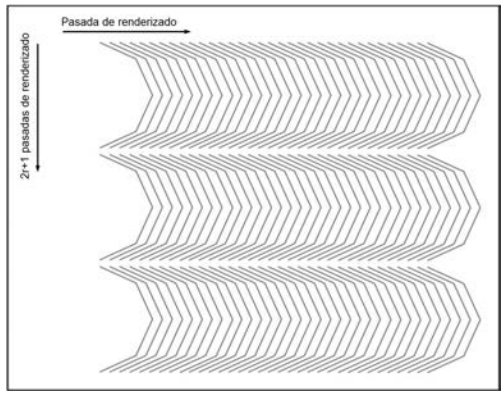
La Tabla 3.5 muestra la relación entre los votos perdidos y el factor de aceleración. El número de pasadas de renderizado de la segunda columna representa el peor caso, ya que en alguna de las fases de renderizado es posible que no exista solapamiento entre los diferentes puntos de contorno. Esta reducción será mayor para imágenes con pocos puntos, y podrían aplicarse una serie de optimizaciones en GPU con la aplicación previa de una fase de preprocesamiento sobre los contornos. Sin embargo, este capítulo estudia las vías de optimización para el peor caso como fiel indicador de la complejidad computacional de la implementación GPU.



(a) GL_LINE_LOOP (s = 8).



(b) Agrupando semicírculos (primera mitad).



(c) Agrupando semicírculos (segunda mitad).

Figura 3.11: Planos de renderizado empleados en las distintas optimizaciones en GPU. (a) Polígonos de ocho caras alineados en una malla 2D cuyos centros están separados en $2r + 1$ píxeles para evitar la colisión de votos. (b) y (c) Todos los círculos "C" se renderizan en una primera pasada, mientras que los círculos "D" lo hacen en una segunda pasada adicional.

3.7.1. Agrupando semicírculos

La gran cantidad de pasadas de renderizado que requiere la implementación básica puede ser reducida a $2 \cdot (2r + 1)$ si los círculos son divididos en semicírculos y son procesados como se muestra en las Figuras 3.11b y 3.11c sobre el eje horizontal. Todos los semicírculos “C"son procesados en una primera pasada de renderizado y, posteriormente, se procede de igual manera con los semicírculos “D". Aún así, el eje vertical sigue requiriendo $2r + 1$ pasadas, pero ahora cada plano de renderizado acumula más votos y la implementación es más eficiente en GPU.

La primitiva de interpolación empleada en esta implementación y en las sucesivas que de ella se derivan es GL_LINES_STRIP.

3.7.2. Uso de la caché de vértices y la rotación de matrices

Los procesadores de vértices pueden beneficiarse de los datos procesados en las pasadas previas de renderizado si hacen uso de la caché de vértices y de la rotación de matrices para explotar las simetrías y la reutilización de los datos.

Un ejemplo se muestra en la Figura 3.11c, que puede obtenerse a través de una rotación de todos los votos en un ángulo de 180 grados respecto a la pasada de renderizado anterior mostrada en la Figura 3.11b. De este modo, se evita toda la generación de vértices en GPU al igual que su comunicación con la CPU. La implementación de esta idea se lleva a cabo creando, en primer lugar, una lista de vértices en la CPU y, posteriormente, renderizando los vértices con *DrawElements* en lugar de mediante los bloques típicos GL_BEGIN-GL_END. La reorganización de los vértices consume tal cantidad de tiempo que no es posible amortizarla con la contribución de la caché de vértices debido a su reducido tamaño.

3.7.3. Mezcla de los canales de color

Las pasadas de renderizado pueden reducirse a la tercera parte si se aprovechan los diferentes canales de color RGB para proyectar diferentes planos y finalmente fusionarlos en uno sólo. En este caso, la colisión de votos se evita, los procesadores de píxeles realizan la acumulación de votos para cada canal y la CPU se encarga de realizar la fusión de los tres canales. El resultado es un renderizado triple con una ejecución más rápida en GPU.

3.7.4. Blending de votos

Como última optimización, existe la posibilidad de evitar la colisión de los votos para el proceso con una única pasada de renderizado a costa de sacrificar precisión en favor del rendimiento. En este caso, los s polígonos no se alinean y el número de colisiones en el espacio de parámetros (a, b) es indeterminado, lo que resulta en un número de votos perdidos que no puede cuantificarse. Sin embargo, el proceso de acumulación de votos puede delegarse a la unidad de *blending* (mezclado), donde el proceso puede llevarse a cabo automáticamente.

Estadísticamente, los votos perdidos son uniformemente distribuidos a través de todo el espacio de parámetros, con especial enfoque en aquellas zonas que reciben más votos. La dificultad de esta implementación recae en la selección que hay que realizar de antemano sobre el peso que va a tener en la escala de colores cada voto para la mezcla de la unidad de *blending*. Un valor de peso muy alto puede provocar una saturación temprana de los canales de color y, por consiguiente, una alta tasa de círculos detectados (y para los valores extremos incluso una desprotección frente al ruido); para valores de peso muy bajo, la GPU podría perder mucha precisión, ya que las unidades de *blending* trabajan en aritmética de punto flotante de 32 bits. En cualquier caso, el parámetro alfa determina el peso real de cada voto.

3.8. Optimizaciones en CPU

La implementación clásica en CPU de la transformada de Hough se ha optimizado en vertientes diferentes a las empleadas para la arquitectura gráfica:

- Basadas en variantes *software* conocidas por ser más rápidas que la versión clásica.
- Aprovechando las recientes ventajas *hardware* en la arquitectura de la CPU, como la tecnología *hyperthreading* y los procesadores multi-núcleo.

3.8.1. Software

La primera variante implementada en CPU hace uso de una tabla precalculada con los valores de las funciones seno y coseno para un radio determinado r y el conjunto de ángulos t en el rango $[0, 2\pi]$ (ver Ecuación 3.1). Con estos valores, los cálculos $r \cdot cos(t)$ y $r \cdot sin(t)$ sólo tienen que consultarse con una indirección de memoria

(a través de las coordenadas (x, y) se obtienen los valores (a, b) del espacio de parámetros). Con esta optimización, el valor de ganancia conseguido asciende a 4x, que además puede ser fácilmente llevado a la implementación en GPU almacenando dichos valores como constantes o registros internos dentro del *shader* de vértices.

La segunda optimización usa la técnica del gradiente [2], donde para cada punto del contorno (x, y) se añade o substrae un vector de magnitud r siguiendo la orientación del vector gradiente asociado al punto en cuestión. La complejidad computacional disminuye extraordinariamente, ya que cada punto de contorno se evalúa con una operación simple con dos funciones trigonométricas, un producto y dos sumas, evitando así todos los votos correspondientes y consiguiendo un factor de ganancia de 40x. Como inconveniente, la precisión de la estimación del gradiente disminuye conforme la tasa de error aumenta en la imagen [38]. De esta forma, la implementación clásica es menos sensible al ruido que usando la técnica del gradiente a costa de una carga computacional mayor, y en este capítulo el principal objetivo es abordar la complejidad de un algoritmo más robusto desarrollando una implementación más rápida en GPU.

3.8.2. Hardware

La Tabla 3.6 muestra los tiempos de ejecución en CPU para dos plataformas diferentes en distintos años. Un Intel Pentium 4 del 2006 con frecuencia de reloj 3.2 GHz y un Intel Core 2 Duo a 2.13 GHz del 2008 (ver Tabla 1.1). La plataforma del 2006 ofrece *hyperthreading*, permitiendo la ejecución simulada de un proceso multi-hilo al replicar las unidades funcionales correspondientes al *front-end* del cauce de segmentación de la CPU. La arquitectura del 2008 permite el procesamiento de doble núcleo, siendo el proceso multi-hilo real por disponer de dos procesadores completos replicados.

Los tiempos mostrados para mejorar la transformada de Hough clásica hacen uso de la libreria *p-threads* para la descomposición del algoritmo en dos hilos. La ganancia del doble núcleo se sitúa entre 1.72x y 1.97x. Para el *hyperthreading*, esta ganancia empieza en 1.04x y alcanza su máximo en 1.40x. En ambos casos, la ganancia mínima/máxima se corresponde con la primera/última fila, sugiriendo que la ganancia crece con la carga computacional.

Sería necesario pensar en una CPU compuesta por más de 100 núcleos para conseguir el rendimiento equivalente con una única GPU de bajo coste. El diseño de los procesados multi-núcleo constituyen una tendencia sólida para la futura escalabilidad de la CPU. Sin embargo, en términos de escalabilidad, la GPU atesora un mejor comportamiento. La ejecución de la THC entre una Geforce 7950 GX2 (un modelo

representativo del año 2006) y la GPU usada en 2008 está en un rango entre 15x y 20x de rendimiento superior en esta última, dependiendo del tamaño de la imagen de entrada.

Imagen	δt	CPU 2008. ¿Doble núcleo?		CPU 2006. ¿Hyper-threading?	
		No	Sí	No	Sí
c20r20	0.10	266.7	154.7 (1.72x)	278.82	268.71 (1.04x)
	0.05	528.7	274.2 (1.85x)	536.24	500.91 (1.07x)
c100r20	0.10	375.3	202.3 (1.86x)	387.58	342.37 (1.13x)
	0.05	745.8	381.6 (1.86x)	752.68	651.65 (1.16x)
c500r20	0.10	888.2	465.7 (1.91x)	959.96	738.03 (1.30x)
	0.05	17770.6	888.3 (1.91x)	1852.08	1418.97 (1.31x)
c20r50	0.10	303.9	168.8 (1.80x)	322.11	294.95 (1.09x)
	0.02	1535.3	753.8 (1.92x)	1502.92	1375.95 (1.09x)
c100r50	0.10	565.7	304.4 (1.86x)	639.79	520.49 (1.23x)
	0.02	2871.0	1407.5 (1.93x)	2885.70	2360.50 (1.22x)
c500r50	0.10	1771.3	905.3 (1.96x)	1890.46	1454.53 (1.30x)
	0.02	9002.1	4425.2 (1.93x)	9399.24	6862.74 (1.37x)
c20r100	0.10	359.8	203.9 (1.76x)	400.64	377.27 (1.06x)
	0.01	3620.9	1978.9 (1.83x)	3670.00	3229.37 (1.14x)
c100r100	0.10	680.6	368.4 (1.85x)	752.99	621.61 (1.21x)
	0.01	6822.4	3408.2 (1.95x)	7061.74	5603.95 (1.26x)
c500r100	0.10	2919.0	1525.1 (1.92x)	3332.09	2387.94 (1.40x)
	0.01	29190.5	14805.2 (1.97x)	31975.06	23169.91 (1.38x)

Tabla 3.6: Tiempos de ejecución (en milisegundos) de la CPU para diferentes imágenes de entrada y pasos de ángulo (δt) bajo dos diferentes plataformas: una CPU Intel Core 2 Duo del 2008 con un único hilo de ejecución y con su versión doble núcleo de la aplicación descompuesta en 2 hilos, y un Pentium 4 del 2006 sin *hyperthreading* pero que se habilita para la aplicación descompuesta en 2 hilos.

3.9. Resultados experimentales

Para el análisis cuantitativo y cualitativo de nuestro algoritmo de la THC se ha empleado una tarjeta gráfica GeForce 8800 GTX con una imagen de 1024 x 1024 píxeles que contiene 20 círculos aleatoriamente distribuidos con un tamaño de radio de 50 píxeles.

3.9.1. Análisis cuantitativo

La Tabla 3.7 proporciona los tiempos de ejecución para diferentes estrategias de computación relacionadas con los resultados mostrados en la Figura 3.8.

Estrategia de computación	Tiempo de ejecución (factor de aceleración)	Ganancia (% reduc.)	Resultado mostrado en	Distribución de votos
La CPU procesa los 314 votos para cada punto	1464.40 ms. (tiempo ref.)	- -	Fig. 3.8a	Caso base
La CPU procesa solo 62 votos para cada punto	292.41 ms. (5.00x)	5.00x (80 %)	Fig. 3.8b	Menos densa
La GPU procesa los 314 votos (GL_POINTS con δt = 0.02)	123.02 ms. (11.90x)	2.37x (58 %)	Fig. 3.8a	Caso base
La GPU procesa solo 62 votos GL_POINTS con δt = 0.01)	47.74 ms. (30.67x)	2.57x (61 %)	Fig. 3.8b	Menos densa
Rasterizador de la GPU con 6 semillas (296 votos interpol.)	35.68 ms. (41.04x)	1.33x (25 %)	Fig. 3.8c	Más precisa

Tabla 3.7: Tiempos de ejecución (en milisegundos) de la Transformada Hough para círculos de tamaño 50 píxeles de radio bajo diferentes estrategias.

Incluso después de considerar la comunicación entre CPU y GPU (8.85 ms. en todos los casos), el tiempo total en CPU se ha acelerado un factor 10 usando la GPU y el mismo número de votos (bien 62 o 314).

La versión de GPU con rasterizador (GL_LINES) emplea 35.68 ms. para 16 semillas, obteniendo una ganancia adicional de 3.44x frente a la versión que utiliza GL_POINTS. Gracias a la contribución del rasterizador, la carga computacional se reduce un 97.5 % y 71.0 % frente a la versión en CPU, respectivamente, lo que revierte en una mejora del rendimiento de 41.04x.

3.9.2. Análisis cualitativo

La Figura 3.8a describe la representación correcta de un espacio de parámetros para la transformada Hough, de forma que 314 votos se han evaluados para cada punto de contorno detectado (en la GPU, equivale al uso de GL_POINTS con un δt óptimo de 0.02 para un radio de 50 píxeles según la relación $\delta t = \frac{1}{r}$ argumentada en la Sección 3.5). El resultado es similar en CPU cuando se evalúa la misma cantidad de votos.

Por otro lado, la Figura 3.8b muestra el espacio de parámetros equivalente al uso de GL_POINTS con $\delta t = 0{,}1$ y, por tanto, evaluando sólo 62 votos por punto de contorno. Los votos están más dispersos alrededor de los centros de los círculos, que aparecen como menos definidos que el caso inicial.

La Figura 3.8c corresponde con el espacio de parámetros obtenido con GL_LINES y s = 16, de forma que se usa el rasterizador con el número de semillas que mantiene el máximo error bajo los límites del tamaño del píxel conforme a lo indicado en la

Imagen	Número de círculos	Radio	Puntos de contorno	Solapamiento de círculos
c20r20	20	20	2512	No
c100r20	100	20	12566	No
c500r20	500	20	62831	Sí
c20r50	20	50	6280	No
c100r50	100	50	31400	Sí
c500r50	500	50	157000	Sí
c20r100	20	100	12560	Sí
c100r100	100	100	62800	Sí
c500r100	500	100	314000	Sí

Tabla 3.8: Conjunto de imágenes empleadas en las pruebas experimentales. Todas las imágenes son de una resolución de 1024 x 1024 píxeles. El campo "Puntos de contorno" representa una estimación de la carga de trabajo exigida para procesar la Transformada Hough.

Sección 3.6.2 (valores mostrados en la Tabla 3.4). Como la Ecuación 3.5 indica, se han evaluado 16 votos y 296 han sido interpolados, resultando en un total de 312 votos y un déficit de 2. Conforme aumenta el número de votos generados por el rasterizador, la imagen resulta más brillante y se percibe un "efecto polígono" en la superficie descrita por la densidad de votos distribuida alrededor de los centros. Sin embargo, observando con detenimiento alrededor de los centros de los círculos (gráfica 3D a la derecha), los votos con GL_LINES se han concentrado mejor en las coordenadas del centro en comparación con los resultados obtenidos con GL_POINTS, describiendo así una localización más precisa del centro. En general, el rasterizador ayuda a refinar la gráfica para los votos distribuidos alrededor del centro del círculo, incluso cuando cuatro píxeles vecinos alcanzan el número máximo de votos simultáneamente. Una cantidad superior de semillas desempeña un papel importante en la definición exacta del centro del círculo, siendo la gráfica más afilada tanto en el centro como en su entorno de vecindad más próximo.

3.9.3. Rendimiento de las técnicas de optimización

La demostración de la efectividad de las técnicas empleadas se lleva a cabo con un conjunto de experimentos en ordenadores personales convencionales (ver las características en la Tabla 1.1). Estos experimentos se han programado con OpenGL 2.0 y el lenguaje Cg de programación de *shaders*.

Sobre imágenes reales, es necesario aplicar un preprocesado que incluya el operador luminancia y un detector de contornos como Canny (ver Figura 3.1). Sin embargo, para evitar dicho preprocesado y aislar el estudio hacia la transformada Hough y la detección de los círculos, se ha trabajado con un conjunto de imágenes en blanco y

negro formadas por una cantidad diferente de círculos y de tamaños de radio (ver Tabla 3.8 para las características de las imágenes y la Figura 3.12 para entender el procesado de las imágenes y sus salidas bajo diferentes versiones del código).

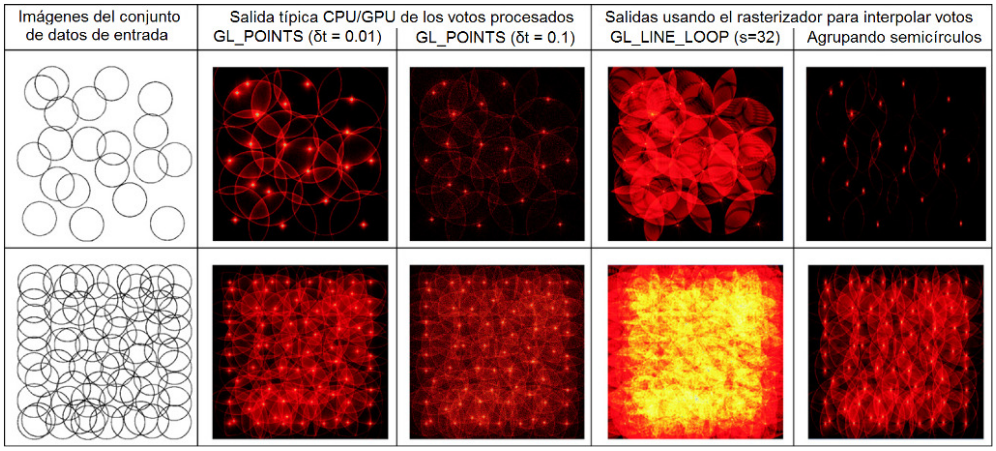


Figura 3.12: Entradas y salidas de datos de la THC para las imágenes representativas c20r100 (arriba) y c100r100 (abajo). Avanzando hacia la derecha, la carga computacional se reduce y el rasterizador está más implicado. Las imágenes de la última columna muestran colores más oscuros debido a que el peso de cada voto se ha infraponderado para evitar la saturación de los colores y permitir la comparación con intensidades similares en GL_POINTS.

La Tabla 3.10 recoge los tiempos de ejecución para todas las imágenes y técnicas de optimización empleadas. En general, la versión con GL_POINTS acelera el tiempo de CPU en un orden de magnitud, y el rasterizador obtiene su mejor beneficio conforme el radio es mayor, donde el aumento del tamaño del círculo implica un mayor número de votos interpolados (ver Tabla 3.9).

La versión básica del rasterizador (GL_LINE_LOOP) alcanza ganancias estables alrededor de 14x, 35x y 70x para tamaños constantes de radio de 20, 100 y 500 píxeles, respectivamente. La técnica *Blending* de votos, siendo la ejecución más rápida, ofrece ganancias inestables en un amplio rango entre 35x y un impresionante 358x. El empleo de GL_LINE_LOOP no se beneficia del incremento de la carga de trabajo con el mismo tamaño de radio por el apabullante número de pasadas de renderizado (ver Tabla 3.5). El *Blending* de votos tiene un buen comportamiento al escalar el número de círculos a procesar, pero el factor de aceleración sufre un decremento debido a la generosa contribución de la memoria caché de la CPU para grandes cantidades de datos con un mismo tamaño de radio.

La Tabla 3.11 amplía los tiempos de ejecución para diferentes valores de δt y

s. El decremento de δt implica un incremento proporcional en el número de votos a procesar por punto de contorno, y en este escenario la GPU muestra un procesamiento mucho más rápido, especialmente en imágenes que contienen pocos círculos y radios pequeños. Por otro lado, el decremento de s tiene dos consecuencias para los círculos que son aproximados por polígonos con pocas caras: se interpolan más votos, sobrecargando el rasterizador, y se procesan menos votos, reduciendo la carga del procesador de vértices. En cargas de trabajo pequeñas (imágenes con 20 círculos), el decremento de s pasa desapercibido en el tiempo de ejecución, lo que sugiere un rasterizado infrautilizado. En cargas de trabajo intensas (subconjunto de imágenes que contienen 500 círculos), el tiempo de ejecución se alivia alrededor de 1/3 cuando el número de semillas es la mitad.

Votos emitidos por GL_POINTS									
Radio	Imágenes representativas del conjunto de datos de entrada	δt = 0.01		δt = 0.02		δt = 0.05		δt = 0.1	
20	c20r20, c100r20, c500r20	628		314		125		63	
50	c20r50, c100r50, c500r50	628		314		125		63	
100	c20r100, c100r100, c500r100	628		314		125		63	
Votos emitidos e interpolados por el rasterizador									
Radio	Imágenes representativas del	s = 4		s = 8		s = 16		s = 32	
	conjunto de datos de entrada	em.	in.	em.	in.	em.	in.	em.	in.
20	c20r20, c100r20, c500r20	4	109	8	114	16	108	32	93
50	c20r50, c100r50, c500r50	4	287	8	298	16	296	32	281
100	c20r100, c100r100, c500r100	4	561	8	604	16	608	32	595

Tabla 3.9: Número de votos emitidos (em.) e interpolados (in.) dependiendo de la implementación en GPU y el radio de los círculos (en píxeles) que va a ser detectado por la transformada de Hough.

3.10. Conclusiones

El cauce de segmentación gráfico de una GPU puede aprovecharse en toda su extensión según la naturaleza del problema a resolver. Este capítulo presenta una implementación alternativa de la transformada de Hough aprovechando el conjunto de la tarjeta gráfica como una extraordinaria combinación de bajo coste y gran rendimiento.

Nuestras implementaciones y optimizaciones de la THC contribuyen con un recorrido sobre las diferentes unidades funcionales de la GPU, desde una versión básica que sólo aprovecha el procesador de vértices para procesar los votos y el procesador de píxeles para acumularlos, hasta versiones más avanzadas donde (1) el rasterizador entra en juego para interpolar el espectro completo de votos, y (2) las unidades

Casos de optimización Hardware					
Imagen	CPU	GPU, GL_POINTS		GL_LINE_LOOP	
	Tiempo	Tiempo	Aceleración	Tiempo	Aceleración
c20r20	528.7	64.7	8.17x	38.8	13.62x
c100r20	745.8	89.0	8.37x	50.8	14.68x
c500r20	1770.6	23.5	7.51x	117.3	15.09x
c20r50	1535.3	130.0	11.81x	44.4	34.57x
c100r50	2871.0	303.0	9.47x	78.2	36.71x
c500r50	9002.1	1084.0	8.30x	237.4	37.91
c20r100	3620.9	293.0	**12.35x**	53.5	67.68x
c100r100	6822.4	684.0	9.97x	94.8	71.96x
c500r100	29190.5	3737.0	7.81x	396.8	**73.56x**
Versiones con técnicas de optimización					
Imagen	Agrupando semicírculos		Mezcla de canales		Blending de votos	
	Tiempo	Aceleración	Tiempo	Aceleración	Tiempo	Aceleración
c20r20	19.2	27.53x	11.8	44.80x	2.0	264.35x
c100r20	24.6	30.31x	15.1	49.39x	9.8	76.10x
c500r20	69.4	25.51x	58.6	30.21x	48.2	36.73x
c20r50	46.0	33.37x	28.5	53.87x	5.0	307.06x
c100r50	62.9	45.64x	37.2	77.17x	24.8	115.76x
c500r50	164.4	54.75x	143.6	62.68x	117.7	76.48x
c20r100	92.3	39.22x	57.6	62.86x	10.1	**358.50x**
c100r100	108.2	62.82x	67.1	101.67x	34.4	198.32x
c500r100	304.2	**95.95x**	254.9	**114.51x**	213.8	136.53x

Tabla 3.10: Tiempos de ejecución (en milisegundos) y factores de aceleración de todas las optimizaciones desarrolladas para la transformada de Hough. Las versiones de CPU y GL_POINTS se han ejecutado para un ángulo δt tal que la cantidad de votos emitidos es similar a las versiones con rasterizador. Los tiempos del rasterizador se muestran para un valor s de 32 semillas. Los factores de aceleración mínimos se han subrayado y los máximos se han resaltado.

de *blending* se utilizan para almacenar los votos de cada textura de una manera más eficiente.

Para la versión que emplea el rasterizador del procesador de vértices haciendo uso de la primitiva GL_LINES, hemos desarrollado una fórmula para determinar el mínimo número de semillas que garantiza que el error máximo siempre estará por debajo del umbral de la distancia de un píxel y que nos permite ahorrar en tiempo de ejecución sin deterioro en la calidad de los resultados. La fórmula es una función de dos parámetros que influyen negativamente en la precisión del proceso de votación: tamaño superior del radio para la detección de círculos, y resolución inferior para las texturas usadas en la acumulación de votos en GPU.

En una cantidad similar de votos, el conjunto completo de optimizaciones en GPU

Imagen	Tiempo CPU		GL_POINTS	GL_LINE_LOOP	*Blending*
c20r20	266.7		44.6	29.9 (4)	1.9 (4)
		($\delta t = 0.1$)		30.7 (8)	1.9 (8)
	528.7		**64.7**	34.1 (16)	2.0 (16)
		($\delta t = 0.05$)		**38.8** (32)	**2.0** (32)
c100r20	375.3		63.2	32.4 (4)	8.0 (4)
		($\delta t = 0.1$)		35.9 (8)	8.0 (8)
	745.8		**89.0**	39.9 (16)	8.5 (16)
		($\delta t = 0.05$)		**50.8** (32)	**9.8** (32)
c500r20	888.2		14.0	45.4 (4)	17.2 (4)
		($\delta t = 0.1$)		54.9 (8)	23.2 (8)
	1770.6		**23.5**	74.0 (16)	32.7 (16)
		($\delta t = 0.05$)		**117.3** (32)	**48.2** (32)
c20r50	303.9		51.2	31.6 (4)	4.8 (4)
		($\delta t = 0.1$)		33.4 (8)	4.8 (8)
	1535.3		**130.0**	35.5 (16)	4.9 (16)
		($\delta t = 0.02$)		**44.4** (32)	**5.0** (32)
c100r50	565.7		80.0	37.0 (4)	20.2 (4)
		($\delta t = 0.1$)		46.4 (8)	20.3 (8)
	2871.0		**303.0**	56.3 (16)	21.6 (16)
		($\delta t = 0.02$)		**78.2** (32)	**24.8** (32)
c500r50	1771.3		256.0	64.7 (4)	43.1 (4)
		($\delta t = 0.1$)		81.4 (8)	54.7 (8)
	9002.1		**1084.0**	133.9 (16)	77.9 (16)
		($\delta t = 0.02$)		**237.4** (32)	**117.7** (32)
c20r100	359.8		59.9	34.5 (4)	9.7 (4)
		($\delta t = 0.1$)		35.1 (8)	9.7 (8)
	3620.9		**293.0**	40.0 (16)	9.9 (16)
		($\delta t = 0.01$)		**53.5** (32)	**10.1** (32)
c100r100	680.6		97.0	39.2 (4)	30.9 (4)
		($\delta t = 0.1$)		47.7 (8)	31.0 (8)
	6822.4		**684.0**	61.5 (16)	32.0 (16)
		($\delta t = 0.01$)		**94.8** (32)	**34.4** (32)
c500r20	2919.0		403.0	87.8 (4)	105.6 (4)
		($\delta t = 0.1$)		121.3 (8)	109.6 (8)
	29190.5		**3737.0**	216.2 (16)	145.7 (16)
		($\delta t = 0.01$)		**396.8** (32)	**213.8** (32)

Tabla 3.11: Tiempos de ejecución (en milisegundos) para diferentes estrategias y carga de trabajo de la THC: los tiempos de CPU y GL_POINTS evalúan dos pasos de ángulo diferentes, δt, y GL_LINE_LOOP y *Blending* evalúan diferentes cantidades de semillas (proporcionadas entre paréntesis). Los valores con un número similar de votos procesados son resaltados para cada imagen con el objetivo de hacer una comparación lo más justa posible entre estrategias.

desarrollado en este capítulo acelera el rendimiento de CPU en un factor entre 25x y 358x, dependiendo del tamaño del radio y el número de círculos que contienen las imágenes de muestra.

Finalmente, con el objetivo de realizar una comparativa entre CPU y GPU en términos de rendimiento y escalabilidad, hemos llevado a cabo una serie de optimizaciones en CPU. Con la ayuda de tablas trigonométricas precalculadas, hemos obtenido un factor de mejora de 4x para ambas versiones. Si empleamos técnicas de gradiente, el rendimiento obtiene una ganancia de 40x a costa de aumentar la sensibilidad al ruido. En este aspecto, la GPU cosecha ejecuciones más rápidas, y siempre con la robustez del algoritmo clásico de la THC.

La escalabilidad se ha analizado sobre la misma familia de plataformas *hardware* entre 2006 y 2008. La GPU GeForce 8800 obtiene un rendimiento entre 10 y 20 veces superior a la GPU GeForce 7950 GX2, mientras que la CPU Intel Core 2 Duo reduce los tiempos de ejecución en aproximadamente la mitad respecto al Pentium 4.

Parte III

GPGPU contemporánea

4 Clasificación celular de los tumores neuroblásticos

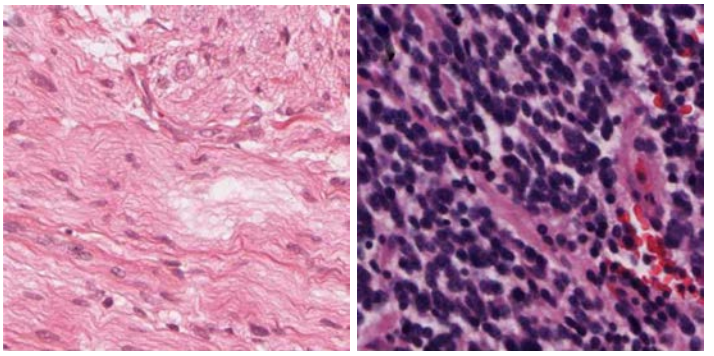
4.1. Análisis de imagen histopatológico

El neuroblastoma es un cáncer agresivo del sistema nervioso que afecta principalmente a los niños. El pronóstico de la enfermedad depende en gran medida del examen histopatológico de los tejidos seccionados y tintados con hematoxilina y eoxina (H & E), realizado visualmente con un microscopio por patólogos expertos. El sistema de clasificación actual se basa en algunas características morfológicas de la histología del tejido tales como el grado de desarrollo del stroma Schwannian, el grado de diferenciación y la mitosis y el índice karryorhexis. Dicho análisis junto con otros indicadores como los factores genéticos y la edad del paciente puede desembocar en un pronóstico favorable o desfavorable.

Sin embargo, el examen cuantitativo de las muestras de tejido depende frecuentemente de variaciones entre los expertos y de un sesgo muestral. Por tanto, dicho método propenso a errores puede ocasionar la planificación de un tratamiento inadecuado para los pacientes. Para solucionar este inconveniente, existen métodos de análisis cuantitativos por ordenador que son más objetivos y precisos a la hora de extraer las características diagnósticas que desde el punto de vista del ser humano sería imposible apreciar [13, 27, 70].

Este capítulo va a tratar la clasificación del grado de desarrollo del stroma de Schwannian, uno de los indicadores más importantes de malignidad según el Sistema Internacional de Clasificación del Neuroblastoma (INCS). El INCS requiere distinguir entre estroma pobre y rico de los tejidos, lo que vamos a abordar como un problema

de clasificación de patrones de los tejidos. La Figura 4.1 muestra regiones pequeñas de tejido rico y pobre en estroma. La textura del tabique del estroma (ver Figura 4.1a) está bastante diferenciada respecto a los neutrófilos (ver Figura 4.1b). Las estructuras de fibrina del estroma con forma de pelo muestran patrones organizados localmente en direcciones concretas, mientras que el neutrófilo sigue una estructura de malla. Esta información se ha extraído de las características de la textura a través de estadísticas de co-ocurrencia y patrones binarios locales (*Local Binary Patter*, LBP). Una descripción más detallada de todo el análisis de la imagen para la clasificación del desarrollo del estroma puede encontrarse en [70].



(a) Tejido rico en estroma. (b) Tejido pobre en estroma.

Figura 4.1: Muestras de imágenes de neuroblastoma.

Debido a las grandes resoluciones de las imágenes procesadas (hasta más de 100K x 100K, con tamaños superiores a 30 gigabytes), el conjunto completo de imágenes seccionadas se divide en grupos más pequeños que no se solapan, y al que se le aplica el análisis de imagen rutinario de forma independiente. Posteriormente, los resultados de cada imagen se fusionan para obtener el mapa de clasificación correspondiente, que consiste en un conjunto de etiquetas de clasificación para cada imagen (estroma pobre o rico). La resolución del mapa de clasificación dependerá de la resolución seleccionada para los grupos de imágenes resultantes de la visión, de forma que las imágenes de menor resolución originan un mapa de clasificación de mayor resolución, y viceversa. La Sección 4.1.1 resume el análisis de imagen propuesto para cada imagen.

4.1.1. El algoritmo

Tal y como ya se ha mencionado, el objetivo es la diferenciación en las texturas entre el tejido rico en estroma y el tejido pobre en estroma. Para ello se ha creado un vector de características compuesto por estadísticos de co-ocurrencia y LBPs para

cada imagen. La clasificación en este espacio de características se finaliza de forma supervisada. La Figura 4.2 presenta el flujo de trabajo del algoritmo de análisis de imagen para la clasificación del desarrollo del estroma en imágenes de neuroblastoma tintadas con hematoxilina y eoxina. Dicho análisis se compone de cuatro fases que se definen a continuación.

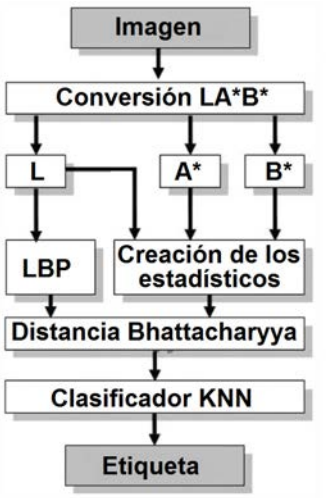


Figura 4.2: Diagrama de flujo del algoritmo de clasificación del estroma.

Transformación del espacio de colores. Para obtener una mejor representación de la información de la intensidad y de los colores de manera independiente, el espacio de colores LA*B* proporciona un espacio de colores perceptiblemente uniforme. La uniformidad perceptible implica que un cambio de color de la misma proporción siempre se manifiesta visualmente con la misma importancia [60]. En el espacio de colores LA*B*, el canal L corresponde a la luminancia o iluminación y los canales A* y B* corresponden a dos dimensiones opuestas de color (los colores opuestos son aquellos que son percibidos de esta manera por el ojo humano). La información de iluminación y de colores se separa para que el procesamiento de la textura sea más apropiado para el reconocimiento de patrones a posteriori.

Características de co-ocurrencia. Los estadísticos de co-ocurrencia obtenidos de las matrices de co-ocurrencia facilitan cuatro características por canal: contraste, correlación, homogeneidad y energía. Dichas características se tipifican como las características de Haralick [77]. Las matrices de co-ocurrencia miden la frecuencia con la que un píxel con una intensidad i aparece en una relación espacial específica con un píxel de intensidad j (ver Figura 4.3).

El tamaño de la matriz de co-ocurrencia tiene un gran impacto en la carga computacional, pero una influencia relativa marginal en la precisión del algoritmo de clasificación. Este hecho ha sido validado después de realizar algunas comparativas de precisión de la clasificación sobre el conjunto de datos de entrenamiento para tamaños de la matriz de co-ocurrencia comprendidos entre 4 y 64 (ver Sección 4.1.2). Por lo tanto, a partir de ahora nos quedaremos con el tamaño de matriz más pequeño, 4 x 4, que será común a toda nuestra fase experimental salvo que explícitamente indiquemos otra cosa.

El pequeño tamaño de la matriz de co-ocurrencia es apropiado para esta aplicación, puesto que se extrae adicionalmente información complementaria de tres canales: la luminacia (L), y los canales de color (A* y B*). Por otro lado, las imágenes de los tejidos teñidos con H & E son de una gran calidad y un contraste excepcional. De esta forma, dado que las estructuras en las imágenes están destinadas a ser tan perceptibles como sea posible, se aprovecha el alto contraste para reducir el tamaño de la matriz de co-ocurrencia, y consecuentemente, los tiempos de ejecución.

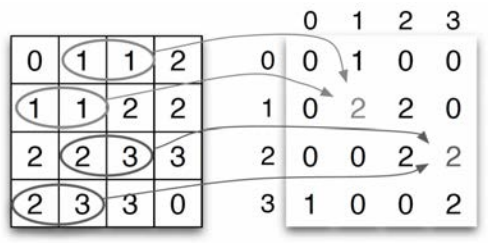


Figura 4.3: Cálculo de la matriz de co-ocurrencia (derecha) para una imagen de tamaño 4x4 en la que se muestra las intensidades de cada píxel.

Operador LBP. El patrón binario local es un potente método de construcción de características de texturas para capturar la presencia de micro-patrones en las imágenes. Se emplea en multitud de ámbitos, desde el reconocimiento de expresiones faciales [82] hasta la recuperación de imágenes basadas en el contenido [74]. El operador LBP convencional se aplica a cada píxel examinando los ocho píxeles vecinos para comprobar si la intensidad es mayor que la del píxel central. Los resultados generan un número binario $b_0b_1b_2b_3b_4b_5b_6b_7$ donde $b_i = 0$ si la intensidad del i-ésimo vecino es menor o igual que p y 1 en cualquier otro caso (ver Figura 4.4).

Ojala y colaboradores publicaron las características del LBP invariantes a la rotación con el uso de píxeles vecinos muestreados circularmente [57]. En lugar de usar un núcleo rectangular, ellos proponen el uso de un núcleo circular alrededor de cada píxel para incrementar el grado de libertad de la rotación invariante. Por tanto, presen-

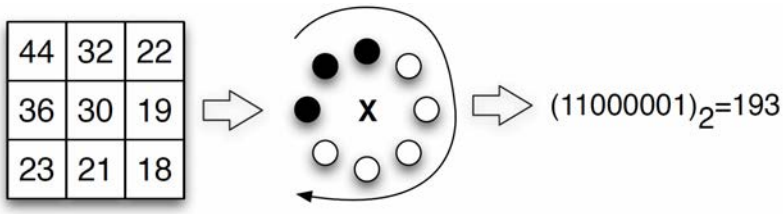


Figura 4.4: Operador LBP sobre una imagen de tamaño de 3x3.

taron patrones uniformes que contienen como mucho dos transiciones de bit de 0 a 1 o viceversa en su cadena circular. Dichos patrones uniformes se denominan foco brillante, área plana, punto negro y varios bordes de curvatura positiva y negativa. Según esta potente representación invariante a la rotación y cualquier cambio de intensidad local o global, las características LBP se han procesado con un radio de un píxel para generar el patrón circular con ocho muestras alrededor de cada píxel.

Histogramas. Dado que los valores de la característica LBP están comprendidos entre 0 y 255, se construye un histograma con 256 intervalos para la imagen completa donde cada intervalo acumula el número de píxeles que tienen un valor de característica LBP. Siguiendo el criterio de uniformidad LPB en [57], los intervalos se reducen en diez clases canónicas normalizadas entre 0 y 1 que forman las componentes de un vector de 10 dimensiones. A continuación se mide la distancia de Bhattacharyya [67] entre el vector de características LBP y el vector (1,1,1,1,1,1,1,1,1,1) para componer la característica LBP uniforme empleada para la clasificación del estroma.

4.1.2. Estudio de la validación de los resultados de clasificación

La evaluación de la clasificación del estroma se ha desarrollado con un conjunto de entrenamiento formado por 250 de imágenes ricas en estroma y otras 250 pobres en estroma que han sido seleccionadas específicamente para sus categorías desde imágenes completas de neuroblastoma. El algoritmo de clasificación usado es el KNN [40], un método supervisado que mapea las muestras de entrenamiento en un espacio de características multi-dimensional y asigna a la muestra probada la etiqueta o clase más frecuente entre las k muestras más cercanas. La principal razón para seleccionar KNN es su rápido comportamiento adaptativo en diferentes conjuntos de datos, ya que usa información local y no depende de probabilidades previas, de modo que proporciona estimaciones precisas de la distribución real de los datos. Para los experimentos se seleccionaron dos metodologías diferentes para seleccionar los datos, una de ellas a través de validación cruzada y la otra de forma aleatoria. En el experimento de valida-

ción cruzada, cada vez se toma una muestra de prueba que se clasifica con el conjunto de entrenamiento del resto de las muestras. El proceso se repite con una planificación *round-robin* hasta que toda las muestras se toman como muestra de prueba al menos una vez. En el experimento con selección aleatoria, cada vez se eligen 50 muestras de prueba y el entrenamiento se realiza con las 450 muestras restantes. Este proceso se repite 20 veces y la precisión de la clasificación final se calcula como la media de las precisiones de clasificación en cada iteración. Todos los experimentos usan matrices de co-ocurrencia de diferente tamaño: 4 x 4, 8 x 8 y 16 x 16.

La Tabla 4.1 muestra la precisión de la clasificación obtenida en cada experimento. Como se puede percibir en la tabla, la precisión de clasificación no cambia drásticamente al variar el tamaño de la matriz de co-ocurrencia. Aunque las matrices de co-ocurrencia de 16 x 16 producen los mejores resultados en términos de precisión, desde un punto de vista de la relación entre eficiencia y precisión, parecen más interesantes las matrices de co-ocurrencia de tamaño 4 x 4. Los efectos de cambiar el tamaño de la matriz de co-ocurrencia serán discutidos en la Sección 4.2.2.

Tamaño matriz de co-ocurrencia	Validación cruzada (%)	Selección aleatoria (%)
4 x 4	98.6 ± 11.8	98.76 ± 1.5
8 x 8	98.6 ± 11.8	98.86 ± 1.6
16 x 16	99.4 ± 7.7	99.59 ± 0.8

Tabla 4.1: Precisión de clasificación para diferentes tamaños de la matriz de co-ocurrencia.

4.2. Implementación pre-CUDA en GPU: *Programación gráfica clásica*

Desde la existencia de las tarjetas gráficas, las GPUs han constituido una opción viable para procesar operaciones de gran intensidad aritmética. Sin embargo, la transformación de cualquier proceso en un código apto para la GPU no es un proceso trivial desde una perspectiva de la programación gráfica clásica, sin existencia de lenguajes de programación y librerías específicos para tal propósito, y donde el rendimiento del algoritmo tiene una gran dependencia de cómo se consiga explotar el paralelismo y el ancho de banda de la memoria.

En general, en una programación gráfica clásica, la GPU acepta una imagen como conjunto de datos de entrada, la transforma a través de una secuencia de etapas, algunas de ellas programables (*shaders*), y devuelve un conjunto de datos como imagen de salida (píxeles rasterizados) escrita en el *frame buffer* (pantalla). La Figura 2.1 recoge

los recursos disponibles para tal propósito en un PC contemporáneo equipado con una GPU.

A continuación ilustraremos cómo llevar a cabo este proceso para el caso concreto de nuestro análisis de imágenes biomédicas descrito en la Figura 4.2.

4.2.1. Detalles de implementación en GPU

Conversión RGB a LA*B*

La conversión de espacio de color desde RGB a LA*B es una operación típica sobre las imágenes de entrada. Como los datos de entrada también coinciden con el atributo de color de cada píxel, podemos aprovechar la vectorización presente en la GPU para implementar esta conversión. De esa forma, la fórmula para pasar de RGB a LA*B* se lleva a cabo en un *shader* de píxeles con un rendimiento óptimo tanto en el acceso a datos como en el paralelismo, maximizando el ancho de anda y los GFLOPS, respectivamente.

Procesamiento de las características estadísticas

A partir de la imagen de entrada en el formato LA*B*, se procesan en cada uno de los nuevos canales de color las siguientes características: contraste, correlación, homogeneidad y energía. El cálculo de la matriz de co-ocurrencia en esta fase supone una gran desafío para la GPU, donde la transmisión de los datos no puede hacer uso de almacenamiento temporal. Una alternativa para solventar este problema es la de aplicar un operador de reducción en sucesivos pasos de renderizado, donde la matriz se reduce progresivamente en un factor 2 x 2 ó 4 x 4, almacenando los resultados intermedios en pequeñas texturas. Sin embargo, los resultados experimentales revelan que el procesamiento no es lo suficientemente pesado como para amortizar el coste de dividir los renderizados, a pesar de que la intensidad aritmética parece suficiente. En su lugar, se decidió calcular cada uno de los elementos de la matriz de co-ocurrencia como una operación de filtro a través de consultas de oclusión [8]. La matriz resultante se almacena en la memoria de vídeo y posteriormente se envía a CPU, donde se calculan las características finales.

Operador LBP, histograma y distancia final

El operador LBP se ejecuta en la GPU como una operación de convolución con los píxeles vecinos. La implementación es directa y muy rápida cuando se lleva a

cabo con un *shader* de píxeles. El reto para conseguir un gran rendimiento recae en el histograma posterior. Pese a que el histograma se añadió como función intrínseca de OpenGL (GL_Histogram), no conocemos que exista un *hardware* específico dedicado a realizar esta operación en la GPU. La GPU proporciona el mejor rendimiento con histogramas de baja granularidad, resoluciones altas de imágenes e incrementando el número de componentes de color hasta cuatro canales. Pero en nuestro caso, los histogramas son de 256 intervalos y un único canal de color (luminancia), de forma que la CPU ejecuta este histograma más rápido, y consecuentemente la GPU no se usa en esta fase. La reducción posterior de todos los intervalos en sólo diez y el posterior cálculo de la distancia de Bhattacharyya también se realizan en la CPU para reducir la sobrecarga de comunicación y los costes de sincronización entre ambos procesadores.

Comparativa global CPU-GPU

Nuestra estrategia de distribución de procesos refleja el procesamiento ideal para cada tipo de procesador: La GPU, para acceso continuo a un flujo de datos independiente en el sentido Una Instrucción Múltiples Datos (SIMD); la CPU, para bucles pequeños, valores filtrados y una amplia reutilización de datos. La conversión LA*B* es un proceso de barrido sobre un área de memoria sin reutilización de datos. Por otro lado, las características estadísticas requieren la creación de una matriz de co-ocurrencia que muestra una gran localidad de acceso. Por tanto, la conversión LA*B* es un proceso adecuado para la GPU mientras que el cálculo de las características estadísticas se trata mejor con la CPU. El operador final LBP y los cálculos de los histogramas requiere combinar CPU y GPU para obtener un rendimiento óptimo.

4.2.2. Estudio experimental

Configuración experimental

La efectividad de las técnicas se pone de manifiesto en sucesivos experimentos sobre una GPU Nvidia Geforce 8800 sobre el sistema descrito en la Sección 1.3.1. Las especificaciones del *hardware* en detalle se pueden consultar en la Tabla 1.1 y en la Figura 2.2.

Desde el punto de vista de la configuración *software*, la GPU usa el entorno de OpenGL y Cg en su versión 1.4, y para la CPU se usan las herramientas Matlab 7.1 y Microsoft Visual Studio 2005 8.0 C++ con las extensiones multimedia habilitadas directamente a través de la capa HAL *(Hardware Abstraction Layer)* sin ninguna librería intermedia. La implementación original del algoritmo se desarrolla en Matlab por ser un lenguaje de más alto nivel y más accesible y rápido para definir y desarrollar

prototipos, aunque con tiempos de ejecución elevados. Posteriormente, los mismos algoritmos se implementan en C++ con OpenGL y Cg.

Una imagen completa típica de neuroblastoma tiene una resolución de hasta 120 x 80 K, sin embargo se suelen descomponer para procesarlas como subimágenes. La descomposición resulta en imágenes RGB de 8-bits por color en formato TIFF con una resolución de 1020 x 916, tal y como muestra el ejemplo de la Figura 4.1.

Tiempos de ejecución

Los tiempos de ejecución de nuestras tres implementaciones con los tres tamaños diferentes de la matriz de co-ocurrencia se muestran en la Tabla 4.2. Las dos primeras columnas muestran el rendimiento cuando se usa la CPU: la primera columna corresponde al código interpretado en Matlab, y la segunda al binario compilado en C++. Ya que ambos código sólo utilizan la CPU, y en beneficio de una presentación simplificada, se han omitido los prefijos CPU de las menciones al código Matlab y C++ en el resto del artículo. La columna GPU trata la implementación en la que ambos procesadores trabajan de forma cooperativa. Los tiempos de lectura de las imágenes no se han incluido porque pueden evitarse casi en su totalidad usando la extensión NV_fence de OpenGL, que implementa un mecanismo asíncrono para la comunicación entre CPU y GPU, permitiendo que la GPU esté procesando una imagen mientras que la siguiente se transfiere a través del bus PCI-express. De esta manera, podemos ocultar la mayor parte de la latencia de entrada y salida.

Etapa de procesamiento	Tamaño de la matriz de co-ocurrencia	CPU Matlab	CPU C++	GPU OpenGL+Cg
Conversión LA*B*		3185.3(52.7 %)	614.8 (72.1 %)	0.5 (2.7 %)
Estadísticos	16x16	2081.4(34.5 %)	26.2 (3.1 %)	160.8 (96.9 %)
	8x8	2097.4(34.6 %)	27.4 (3.2 %)	44.2 (89.5 %)
	4x4	2081.8(34.5 %)	28.9 (3.4 %)	13.6 (72.3 %)
LBP		771.8(12.8 %)	208.8 (24.5 %)	4.7 (25.0 %)
Total	16x16	6038.5	849.8	166.0
	8x8	6054.5	851.0	49.4
	4x4	6038.9	852.5	18.8

Tabla 4.2: Tiempos de ejecución (en milisegundos) para una imagen de tamaño 1020 x 916 píxeles sobre un PC de sobremesa del año 2007. El % en el paréntesis indica el peso porcentual de cada paso sobre el tiempo total.

Para la extracción de las características estadísticas (ver Sección 4.2.1), cada elemento de la matriz de co-ocurrencia se obtiene como un filtro que requiere una única pasada de renderizado en la GPU. Este procedimiento perjudica de forma severa el rendimiento y, por tanto, el nivel de discretización de la matriz de co-ocurrencia juega

un papel fundamental para la eficiencia de la GPU, mientras que la influencia en la CPU es marginal. Aunque la CPU obtiene resultados más rápidos con matrices de 16 x 16, el empleo de matrices de 8 x 8 ó 4 x 4 conducen a unos resultados de clasificación similares (ver Tabla 4.1) y la computación en GPU es mucho más rápida.

En general, el tiempo de ejecución para la aplicación del análisis de las imágenes patológicas se puede reducir en un factor 321x (matriz de co-ocurrencia 4 x 4 en Tabla 4.3), donde un factor de 7x proviene de la traslación del código Matlab a C++ dentro de la CPU y el resto corresponde a la contribución de la GPU. Mientras que la primera mejora se mantiene constante, la segunda es sensible al tamaño de la matriz de co-ocurrencia y la forma en que se procesa en la GPU. Para simplificar el estudio, de aquí en adelante todos los resultados se refieren a una matriz de co-ocurrencia de 4 x 4, ya que se obtiene la mejor aceleración sin perder precisión en los resultados de clasificación.

Tamaño de la matriz de co-ocurrencia	Matlab/C++	C++/GPU	Matlab/GPU
16 x 16	7.1x	5.1x	36.4x
8 x 8	7.1x	17.2x	122.6x
4 x 4	7.0x	45.3x	321.2x

Tabla 4.3: Factor de aceleración para diferentes matrices de co-ocurrencia sobre una imagen de tamaño 1020 x 916 píxeles bajo diferentes plataformas.

Las Tablas 4.4 y 4.5 y la Figura 4.5 muestran cómo la mejora en CPU no se mantiene constante para cada tarea cuando el tamaño de la imagen va incrementándose: las ganancias se sustentan en el cálculo de las características estadísticas, donde LA*B* y LBP reducen la ganancia como penalización por interpretar el código Matlab en tiempo de ejecución, hecho éste que se reduce cuanto más datos por instrucción surgen del programa. Además, el código Matlab se comporta mejor trabajando con la memoria de forma masiva por el buen uso de la memoria virtual, mientras que el código C++ resulta más efectivo en imágenes pequeñas, haciendo un uso más eficiente de la memoria caché.

Las Figuras 4.6 y 4.7 muestran la ganancia adicional de la GPU sobre el código Matlab y C++. El cómputo de la conversión de color LA*B* es de nuevo la tarea con un impacto de rendimiento mayor, ya que constituye un operador *streaming* típico mientras retiene la mayor parte de la carga de trabajo (ver Tabla 4.5). Este papel de liderazgo se mantiene también en la ganancia total, aunque las características estadísticas no rindan mejor en GPU que en C++ para tamaños de imagen muy pequeños, donde el cómputo no amortiza el coste de mover los datos hacia la CPU. Por último, los factores de aceleración se mantienen para imágenes grandes (1 x 1 K y 2 x 2 K).

Finalmente, la Tabla 4.4 muestra hasta qué punto el peso computacional de cada

Etapa de procesamiento	En la CPU: C++/Matlab	Entre procesadores: CPU/GPU	Global: Matlab/GPU
Conversión LA*B*	5.9x - 5.2x	69.2x - 1409.6x	406.1x - 7391.3x
Estadísticos	122.2x - 90.0x	0.2x - 2.1x	21.8x - 192.1x
LBP	8.3x - 3.9x	4.2x - 38.3x	34.6x - 148.3x
Total	13.3x - 7.6x	2.6x - 46.3x	33.4x - 350.9x

Tabla 4.4: Influencia de la carga de trabajo en las ganancias de rendimiento para cada una de las tareas involucradas en el análisis del neuroblastoma. Las ganancias se muestran para imágenes de 128 x 128. El tamaño de las matrices de co-ocurrencia es de 4 x 4 en todos los casos.

Equipamiento *hardware* del año 2006	Desaceleración relativa (%)
CPU Mobile Intel Core Duo @ 2.16GHz	45.2
CPU Desktop Intel Pentium 4 @ 3.4GHz	34.1
CPU Desktop Mac Pro Intel Xeon @ 2.66GHz	12.0
GPU Nvidia GeForce	96.5

Tabla 4.5: Comparativa de la escalabilidad del *hardware* desde el lado de la CPU y la GPU. La última columna resume la desaceleración en comparación con el *hardware* del año 2007 mostrado en la Tabla 4.2. El tamaño de la imagen es de 1020 x 960 píxeles.

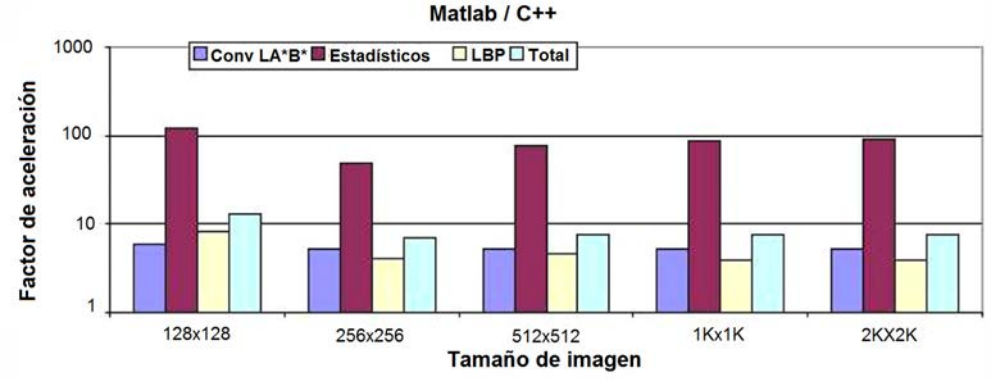


Figura 4.5: Factores de aceleración entre la versión C++ y Matlab para diferentes tamaños de imagen (en píxeles).

tarea depende de la implementación real. En la CPU, la carga principal viene dada por la conversión del espacio de color, mientras que en la GPU las características estadísticas componen la mayoría del tiempo de ejecución. Como se muestra en la última columna, la implementación en GPU proporciona una aceleración de más de dos órdenes de magnitud para imágenes de 2 x 2 K.

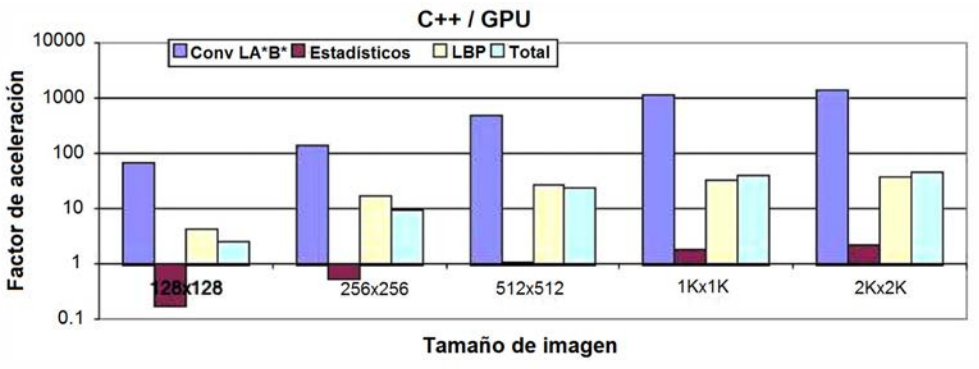


Figura 4.6: Factores de aceleración entre la versión GPU y C++ para diferentes tamaños de imagen (en píxeles).

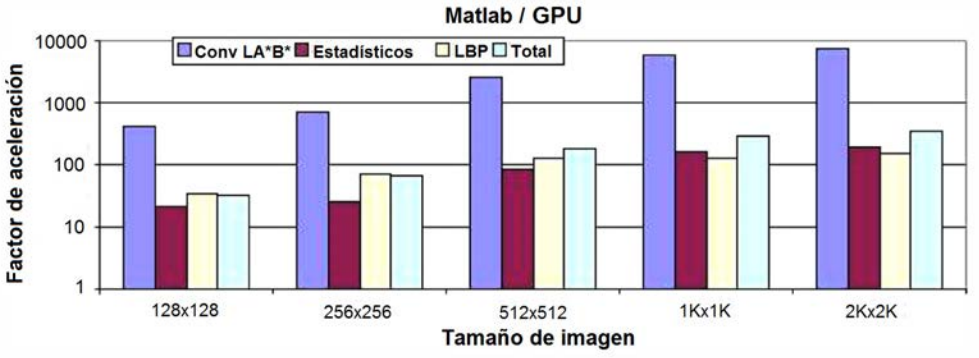


Figura 4.7: Factores de aceleración entre la versión GPU y Matlab para diferentes tamaños de imagen (en píxeles).

Variación del rendimiento en función del hardware. En CPU se ha ejecutado el código en tres equipos diferentes del año 2006: una CPU portátil, un Pentium 4 convencional y un Xeon de gama alta. Por parte de la GPU, el código se ha ejecutado en una GeForce 7950 GX2 en representación del *hardware* gráfico del 2006. Los resultados se muestran en la Tabla 4.5, donde se puede apreciar que la mejora de rendimiento de la GPU es mucho mayor que en todas las CPUs. Mientras que el modelo de GPU del año 2007 consigue un factor de ganancia sobre 2x en comparación con el modelo de 2006, la CPU del 2007 obtiene apenas una aceleración de un 1.1x frente a un modelo de gama alta del 2006, y sólo un 1.3x aproximadamente sobre la versión portátil del 2006.

Validación numérica. Para juzgar la similitud de los valores procesados en diferentes plataformas, ejecutamos el código de análisis de imágenes en cada plataforma,

obteniendo un vector de características para 500 imágenes de nuestro conjunto de imágenes de entrenamiento. La Tabla 4.6 muestra el valor medio y la desviación estándar de las diferencias entre los vectores de 13 características sobre las 500 imágenes. Las diferencias están alrededor de un 3 % con un valor menor de 0.02 en promedio, siendo estas pequeñas variaciones inapreciables en la precisión de la clasificación final (ver Tabla 4.1).

Versiones comparadas	Diferencia media	Desviación estándar	Media del error relativo máximo
Matlab/C++	0.00014 - 0.012	0.00018 - 0.01	2.00
C++/GPU	0.00065 - 0.021	0.00043 - 0.05	3.46
Matlab/GPU	0.0015 - 0.017	0.00075 - 0.05	

Tabla 4.6: Precisión de los valores de salida para diferentes plataformas *hardware*. El rango de variación corresponde a un test sobre 500 imágenes para las 12 características estadísticas junto con el operador LBP (13 valores en total).

Manejo de la memoria y tamaño de las imágenes. La capacidad de la memoria DRAM se ha duplicado cada año y medio, en línea con la predicción de la Ley de Moore. Con el tamaño actual de 1 GB de memoria de vídeo y el formato de color SRGB de 8 bits usado en la GPU para representar las imágenes como texturas, nuestra aplicación se podría ejecutar completamente en un solo núcleo del año 2012. Hasta entonces había que emplear alguna lógica de particionado porque la GPU no utilizaba memoria virtual. Un particionado en imágenes más pequeñas puede beneficiarse de las cachés de la GPU de manera similar a cómo los compiladores aplican las transformaciones de bloques en el ámbito de la CPU, con la distinción que la memoria caché de la GPU es mucho más pequeña y la memoria DRAM mucho más rápida.

La Tabla 4.7 muestra los factores de aceleración conseguidos usando diferentes tamaños para el particionado y para una matriz de co-ocurrencia de tamaño 4 x 4 para la extracción de las características estadísticas. El ancho de banda determina un particionado máximo en imágenes de 128 x 128 píxeles, lo que supone el límite crítico donde el tiempo de inicialización se amortiza por la velocidad de la comunicación. Por otro lado, la capacidad de memoria y la programación del *software* determinan el tamaño máximo de las imágenes, que en la práctica fue de 8 x 8 K para la GPU usando OpenGL 2.1.4, 4 x 4 K para la CPU usando C++, y 2 x 2 K cuando se usa Matlab de 32 bits. La Figura 4.7 muestra que el rendimiento mejora con el tamaño de las imágenes, reflejando que los tamaños de 1 x 1 K y 2 x 2 K son prioritarios de cara al factor de aceleración.

Además, se ha estudiado el efecto de un particionado en imágenes no cuadradas. Cuando la dimensión horizontal de las imágenes es mayor, la CPU es 2.7 % más rápida usando Matlab, lo que refleja que el índice de los *arrays* sigue un orden basado en las

Tamaño de imagen	Matlab/C++	C++/GPU	Matlab/GPU
128 x 128	13.3x	2.6x	33.4x
256 x 256	7.0x	9.1x	64.2x
512 x 512	7.5x	24.1x	182.1x
1024 x 1024	7.6x	439.4x	297.7x
2048 x 2048	7.6x	46.3x	350.9x

Tabla 4.7: Factores de aceleración obtenidos para una matriz de co-ocurrencia de 4x4 sobre diferentes tamaños de imagen (expresados en píxeles).

columnas en la implementación de Matlab que se beneficia de un patrón de localidad a la hora de realizar las peticiones de las líneas de caché. Sin embargo, la GPU no obtiene ningún beneficio usando imágenes no cuadradas.

Comparativa global. Para componer el cómputo de todas las imágenes de la muestra inicial, el tiempo total de ejecución para la clasificación de la región del estroma para una imagen de 50 x 50 K alcanza las 4 horas y 45 minutos en Matlab, 37 minutos para la ejecución C++ en CPU y sólo 145 segundos para la implementación más rápida en GPU. Considerando que aproximadamente 600 pacientes son diagnosticados con neuroblastoma al año y que cada tumor lleva consigo aproximadamente 5-6 biopsias recogidas del paciente, el tiempo total de procesamiento de estas imágenes sería de 21 meses usando Matlab en una CPU de doble núcleo frente a unos 5.3 días para la versión GPU descrita en este capítulo. La Sección 4.2.1 resume la mejor estrategia de distribución de la carga entre los procesadores: la GPU procesa la conversión LA*B* y el operador LBP, mientras que la CPU procesa las sumas de las 12 características estadísticas. En una CPU de doble núcleo programada con POSIX threads, uno de los núcleos puede trabajar en el algoritmo y en interactuar con la GPU, mientras el otro núcleo se encarga de cargar la siguiente imagen a procesar. De esta forma se ocultaría la latencia de entrada/salida para el proceso completo. En general, una vez que la conversión LA*B* ha terminado, el operador LBP y las características estadísticas son independientes y pueden ser procesadas en paralelo en la GPU y el primer núcleo de la CPU, respectivamente. Mientras tanto, el segundo núcleo puede encargarse de cargar el resto de las imágenes.

4.3. Implementación del código de análisis de imagen en CUDA

La implementación CUDA se ha llevado a cabo con su ciclo de desarrollo típico. En primer lugar, el código se compila con las opciones de compilación que generan un informe sobre el aprovechamiento de los recursos hardware de cada *kernel* (regis-

tros requeridos por cada hilo y memoria compartida ocupada por cada bloque). Dicho informe analizado en detalle proporciona el número de hilos y bloques que son necesarios lanzar en la ejecución para conseguir la máxima ocupación dentro de cada multiprocesador. Sí aún así no hay posibilidad de obtener una eficiencia satisfactoria, el código debe revisarse para reducir la presión sobre la memoria.

Debido al gran rendimiento que la GPU proporciona con operaciones en punto flotante, los accesos a memoria suelen ser los cuellos de botella del rendimiento en diversas partes de la aplicación. La imagen de entrada (1K x 1K x 3 bytes) es bastante más grande que el tamaño de la memoria compartida (16 KB), así que la prioridad está en las estructuras de datos como las matrices de co-ocurrencia (fase 2) y los histogramas parciales (fase 4). Sin embargo, en la fase 1, aunque el procesamiento es un barrido sobre todos los píxeles, el tiempo de ejecución es menor cuando se usa la memoria compartida (2.32 ms contra 2.77 ms - ver Tabla 4.8). Además, en la fase 3, los píxeles de entrada se trasladan a memoria compartida porque el cálculo del operador LBP reutiliza mucho los datos.

Para ilustrar la progresión de una implementación CUDA de una manera didáctica y detallada, se van a ilustrar las diferentes optimizaciones que hemos llevado a cabo en cada fase.

4.3.1. Detalles de implementación en CUDA

Fase 1: Conversión de color

Como punto de partida se emplean tipos de datos float3 de 24 bits para cada canal de color. Sin embargo, usando un mecanismo de relleno para que el ancho de datos sea de 32 bits se puede mejorar el rendimiento en todas las optimizaciones en las que entra en escena la memoria compartida (ya que la salida de un banco de esta memoria es de 32 bits). La intención es eliminar la famosa coalescencia de datos, que para esta fase permite ahorrar un 35 % del tiempo de computación, aunque a costa de aumentar el tiempo de comunicación (ver Tabla 4.8). A continuación puede observarse cómo un tipo de datos uchar de 8 bits es suficiente para la precisión de la aplicación. Tal y como se espera, el tiempo de comunicación se reduce aproximadamente a la cuarta parte.

El uso de las opciones de compilación que informan sobre el uso de los registros y la memoria en CUDA indican que el *kernel* CUDA de la conversión de color hace uso de 13 registros y 1064 bytes de memoria compartida, permitiendo una ocupación máxima del 75 % del multiprocesador para una ejecución en la que el bloque tenga entre 176 y 192 hilos. Sin embargo, tomamos la decisión de seleccionar 256 hilos, lo

Fase	Descripción / Optimizaciones	Tiempo de ejecución (ms.)		
		Comun.	Comput.	Total
1:	Versión base: un float3 para cada canal de color	8.49	3.71	12.20
Conversión	Coalescencia (insertando el canal alfa) con float3	10.79	2.44	13.23
de color	Remplazo de float3 por uchar (256 hilos/bloque)	2.98	2.77	5.75
RGB a	**Uso de memoria compartida (256 hilos/bloque)**	**2.98**	**2.32**	**5.30**
LA*B*	Utilizando entre 169 y 192 hilos/bloque	2.98	2.43	5.41
2:	Versión base: memoria global		15.40	15.40
Parámetros	Memoria compartida para las matrices de co-ocurrencia		4.48	4.48
estadísticos	**Conflictos resueltos en bancos de mem. compartida**		**2.58**	**2.58**
3:	Versión base: Hilos especiales sobre los bordes *(halos)*		2.29	2.29
Operador	**Bloques de 16 x 16 hilos, computación en 14 x 14**		**1.82**	**1.82**
LBP	Bloques de 8 x 8 hilos, computación en 6 x 6		2.31	2.31
4:	Versión base: memoria global	4.02	2.08	6.10
Histograma	Memoria compartida para histogramas locales	0.31	0.61	0.92
	Conflictos entre warps resueltos en bancos de mem.	**0.31**	**0.59**	**0.90**
Tiempo	Versión base	12.51	13.48	35.99
total	Con memoria compartida	3.29	9.23	12.52
GPU	**Versión óptima**	**3.29**	**7.31**	**10.60**
Tiempo total CPU: (1:) 880.31 ms + (2:) 43.24 ms + (3:) 156.28 ms + (4:) 7.49 ms = **1087.32 ms.**				

Tabla 4.8: Principales optimizaciones CUDA en la aplicación de análisis de imagen. Los tiempos de ejecución corresponden a una solo imagen de 1K x 1K píxeles en una única GPU. La fase 1 muestra los tiempos para la comunicación entre CPU y CPU, y la computación real en GPU. Las fases 2 y 3 sólo computación (no existe comunicación). La fase 4 muestra los tiempos para la comunicación entre GPU y CPU después de la computación real en GPU.

que supone un canje entre la ocupación (desde 75 % hasta 67 %) para obtener un mejor balanceo de la carga, ya que 256 es múltiplo de 32 (número máximo de hilos por *warp*), y a su vez es divisor de 768 (número máximo de hilos por multiprocesador) y 1024 (número máximo de píxeles por imagen). El resultado muestra que el tiempo de ejecución mejora ligeramente (ver Tabla 4.8). Desafortunadamente, el rendimiento máximo está limitado porque cada hilo necesita 11 registros, provocando que se usen menos de los 768 hilos de ocupación máxima por multiprocesador. El tiempo de ejecución óptimo para esta fase es de 2.98 ms. para la transferencia de los píxeles, y de 2.32 ms. para computar la conversión de color, tal y como se refleja en la Tabla 4.8.

Fase 2: Características estadísticas

Este *kernel* necesita 9 registros y 4132 bytes de memoria compartida, de manera que se pueden alojar hasta 3 bloques de 256 hilos en paralelo. Empleando 768 hilos por bloque podemos subir el factor de ocupación los recursos hardware de la GPU

hasta el 100 %. Los píxeles se distribuyen equitativamente entre los hilos para procesar simultáneamente las matrices de co-ocurrencia. Finalmente, los resultados parciales se acumulan a través del operador de reducción.

Dentro de esta fase, el resto consiste en procesar las matrices de co-ocurrencia evitando los conflictos de acceso a los 16 bancos de memoria. Con una matriz de 256 hilos distribuida en 16 x 16, el despliegue más sencillo de hilos sería el de forzar a los 32 hilos de un *warp* el acceso a sólo 8 bancos de memoria compartida, cuyo rendimiento estaría mermado por dicha limitación de paralelismo. Actuando de manera inteligente, el acceso de los hilos activos se puede distribuir de manera que ningún hilo tenga que esperar para acceder a su banco de memoria (ver Figura 4.8). Esta compleja optimización resuelve todos los conflictos en el acceso a memoria, reduciendo el tiempo de ejecución a 2.58 ms. desde 4.48 ms. Sin el uso de memoria compartida, una implementación base tardaría 15.40 ms (ver Tabla 4.8).

Fase 3: Operador LBP

La implementación del operador LBP conlleva la aplicación de una máscara de convolución de tamaño 3 x 3 píxeles seguida de una conversión de binario a decimal (ver Figura 4.4). Cada hilo necesita 10 registros y cada bloque de hilos usa 296 bytes de memoria compartida. Debido a las características del uso de la memoria, se puede alcanzar una ocupación de 100 % con una distribución de 256 hilos desplegados en cuadrículas de 16 x 16. Cada hilo lee un píxel desde memoria global y lo almacena en la estructura de datos de memoria compartida. Los hilos situados en los bordes de las cuadrículas no pueden realizar procesamiento alguno puesto que no tienen acceso a todos sus vecinos más cercanos (ver Figura 4.9). El operador LBP para las regiones de los bordes se calcula con el siguiente bloque de hilos, para ello es necesario que haya solapamiento entre bloques de dos filas y dos columnas. Esta estrategia de cómputo sufre las consecuencias de un 23 % de ciclos ociosos y del correspondiente acceso a memoria redundante, a costa de conseguir un procesamiento más homogéneo y que los hilo estén menos sobrecargados. En comparación con la versión heterogénea, donde no existen ciclos ociosos y accesos redundantes a memoria, la versión homogénea es un 25 % más rápida, llevando el mejor tiempo de ejecución hasta los 1.82 ms.

Dado que el procesamiento del operador LBP es muy regular, existe la posibilidad de seleccionar diferentes tamaños para el despliegue de hilos, siempre y cuando la distribución se haga en cuadrículas simétricas. Por tanto, se ha investigado el efecto de las diferentes relaciones de hilos por bloque, en particular con los tamaños 20 x 20 (2.02 ms.), 18 x 18 (1.90 ms.), 16 x 16 (1.82 ms. - óptimo), 14 x 14 (1.89 ms.), 12 x 12 (2.00 ms.), 10 x 10 (2.16 ms.), y 8 x 8 (2.31 ms. - ver en Tabla 4.8). Un

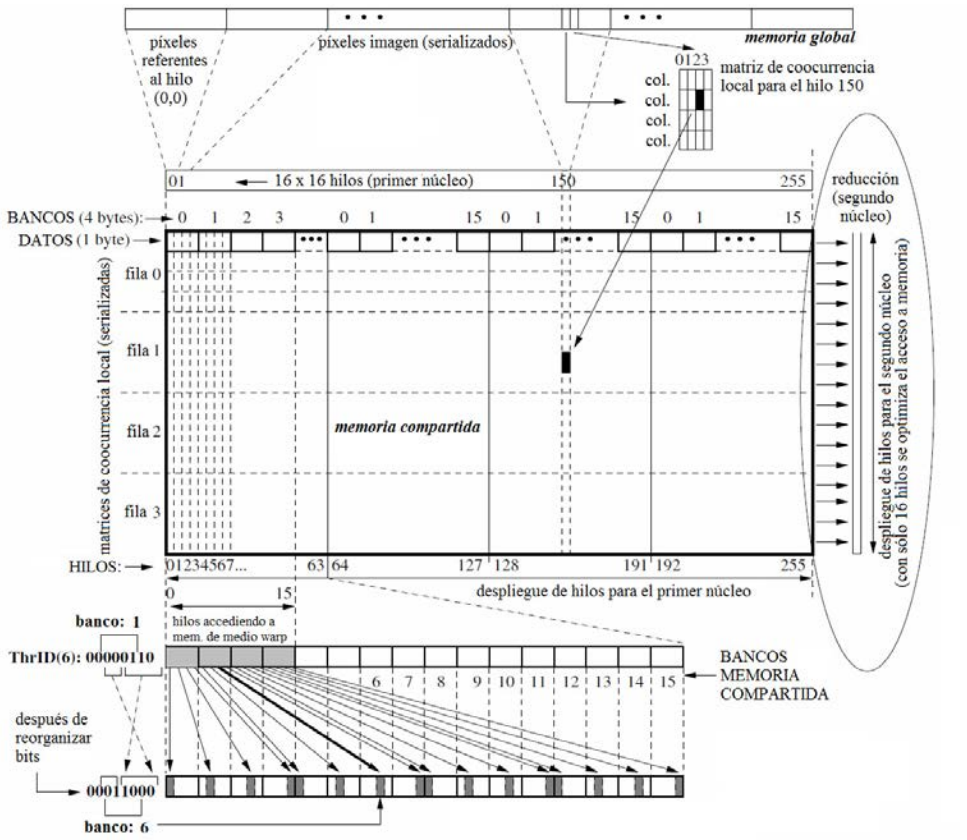


Figura 4.8: Matrices locales de co-ocurrencia en CUDA. La asignación de bancos en memoria compartida a cada hilo evita conflictos de memoria al procesar las matrices de co-ocurrencia.

tamaño de cuadrícula de 16 x 16 es en la mayoría de los casos una buena elección para maximizar el rendimiento en CUDA. El estudio sobre los diferentes tamaños sólo pretende cuantificar la penalización que conlleva una mala elección de tamaño.

Fase 4: Histograma

La implementación de esta fase se basa en el núcleo del histograma incluido en la librería CUDA [64], donde la reducción global se delega a la CPU. Esta solución se ha considerado acertada porque el histograma sólo se computa una vez por imagen y no implica un gran tiempo de ejecución. El tiempo de ejecución para esta última fase es de 0.9 ms.

La Tabla 4.8 resume todas las optimizaciones desarrolladas en cuanto a CUDA se

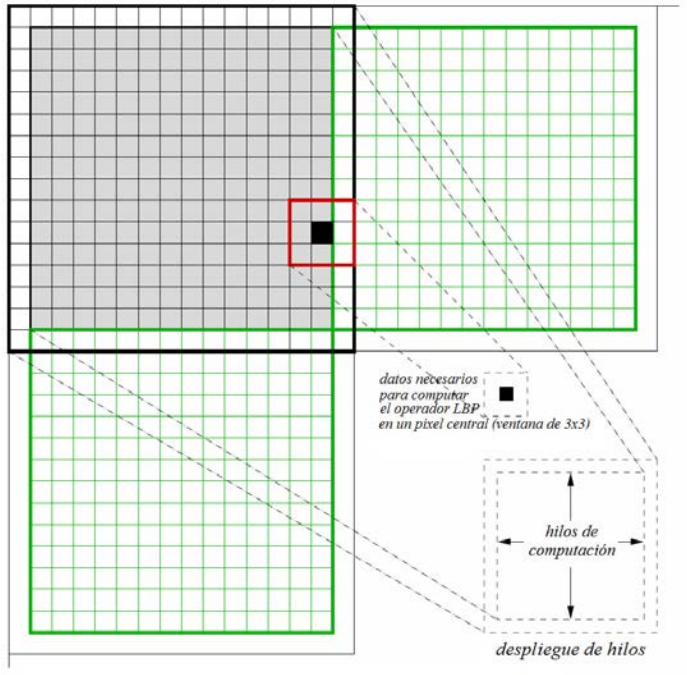


Figura 4.9: Operador LBP en CUDA. La memoria empleada por cada bloque es superior a la cantidad de hilos para poder computar las convoluciones en todos los bloques de forma homogénea.

refiere, desglosando por fases y tiempos de ejecución. Dado que el modelo de programación CUDA para la versión empleada no permite la ejecución de *kernels* concurrentes entre multiprocesadores, las cuatro fases se ejecutan secuencialmente y la GPU se explota en su completitud para cada fase independientemente. En general, la aplicación de CUDA para esta aplicación de análisis de imágenes permite la reducción de los tiempos de ejecución en un factor entre 3 y 5 respecto a los tiempos obtenidos en la versión Cg (ver Tabla 4.9 para los tiempos sobre una imagen completa), con un factor adicional de 3x cuando se habilita la memoria compartida, añadiendo además un 20 % extra cuando se solventan los conflictos en los accesos a los bancos de memoria.

4.3.2. Datacutter

La paralelización del análisis de imagen que hemos abordado en este capítulo ha contado con la ayuda del *middleware* Datacutter. Datacutter [6] ha sido desarrollado

	En CPU		En GPU	
Tamaño imagen	Matlab	C++	Cg	CUDA
Pequeño	2h 57' 29"	43' 40"	1' 02"	27", 14"
Mediano	6h 25' 45"	1h 34' 51"	2' 08"	58", 32"
Grande	11h 39' 28"	2h 51' 23"	3' 54"	1' 47", 58"

Tabla 4.9: Tiempos de ejecución de la aplicación de análisis de imagen para diferentes métodos de programación y plataformas *hardware*. Los tiempos no incluyen la sobrecarga por entrada/salida y descompresión. La matriz de co-ocurrencia es de tamaño 4x4. El tamaño de la imagen es de 1K x 1K píxeles.

por el grupo de investigación con el que hemos colaborado en The Ohio State University durante nuestras estancias de investigación. Se trata de un entorno de trabajo basado en componentes que proporciona un sistema de flujo de datos de grano grueso, permitiendo combinar paralelismo de tareas y de datos. Las aplicaciones se descomponen en tarea secuenciales (filtros) con dependencia de datos entre ellas. El flujo de datos entre los filtros se lleva a cabo a través de una distribución lógica en búferes no bloqueantes. El almacenamiento en búferes permite un solapamiento sencillo de la comunicación de datos y la computación que permite ocultar las latencias de disco y de intercomunicación.

El entorno de ejecución soporta una ejecución eficiente de filtros sobre clústeres de computación heterogéneos, multi-zócalo y multi-núcleo. Dicho entorno ejecuta todos los pasos necesarios para instanciar cada filtro en el equipo deseado para conectar todos los extremos lógicos y llamar a las interfaces de los filtros para procesar el trabajo. Cada filtro se ejecuta en un hilo POSIX separado, facilitando a la CPU el solapamiento de comunicación de entrada y salida. El intercambio de datos entre dos filtros en el mismo equipo se realiza con copia de zonas de memoria o con operaciones de punteros, mientras que por otro lado los sockets TCP o las comunicaciones MPI (con el fin de aprovechar las interconexiones de gran ancho de banda y baja latencia como Infiniband) se emplean para la comunicación de filtros entre diferentes equipos.

Nuestra aplicación de análisis de imagen se divide en tres fases sencillas, cada una de ellas implementada con un filtro: (1) un lector de las imágenes TIFF que leen los binarios TIFF, (2) un descompresor de imágenes TIFF a formato RGB y (3) el análisis de imagen en sí. Un diseño con un filtro de cada tipo completa la tarea para una imagen completa, mientras que la réplica de dichos filtros permite una rápida paralelización en aras a una ejecución más eficiente.

La Figura 4.10 representa el diseño general del sistema. Uno o más nodos de lectura leen el binario de la imagen TIFF desde disco y distribuyen la información hacia el descompresor TIFF. Uno o más filtros de descompresión TIFF pueden coexistir

en el mismo nodo y básicamente cogen el binario TIFF leído y lo transforman en una imagen RGBA apropiada. La imagen descomprimida se escribe en la entrada del filtro analizador de imagen en el mismo nodo. Cuando dicho filtro emplea la GPU, explotar todo su potencial depende de la habilidad de los múltiples filtros descompresores a la hora de proporcionar datos. Alojar los nodos de descompresión y de análisis en el mismo nodo tiene el beneficio de ahorrar en ancho de banda, dejando la imagen resultante en el mismo lugar (solamente se transfiere el puntero de memoria).

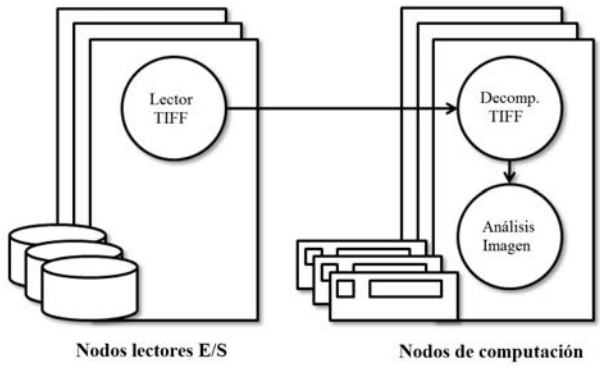


Figura 4.10: Estructura del DataCutter para la aplicación de análisis de imágenes.

El modelo de programación de DataCutter basado en componentes nos permite fácilmente desarrollar y desplegar aplicaciones de análisis de imágenes que utilizan la GPU como co-procesador. Simplemente remplazando el filtro de análisis en C++ por uno específico desarrollado para la GPU, podemos rápidamente desarrollar un código paralelo que se ejecuta eficientemente en un clúster multi-zócalo, multi-núcleo o multi-GPU. El diseño, los filtros de lectura y descompresión, y el sistema entero de paralelización son reutilizables.

4.3.3. Resultados experimentales

Nuestros experimentos se ha llevado a cabo en el clúster BALE del Ohio Supercomputing Center (ver Sección 1.3.2), usando los 16 nodos de visualización y seis nodos adicionales como nodos de lectura, de forma que el ancho de banda conseguido evita la creación de cuellos de botella de entrada/salida. Cualquier clúster de producción dedicado al análisis de imagen con un gran rendimiento gracias a su configuración de GPUs y una rápida interconexión suele proporcionar un gran ancho de banda para entrada/salida.

La Tabla 4.10 resume las características de las tres imágenes patológicas digitalizadas que hemos utilizado a lo largo de nuestros experimentos.

Nombre	Resolución en píxeles	Número de subimágenes de 1Kx1K
Pequeña	32,980 x 66,426	33 x 65 = 2,145
Mediana	76,543 x 63,024	75 x 62 = 4,659
Grande	109,110 x 80,828	107 x 79 = 8,453

Tabla 4.10: Propiedades de las imágenes usadas en los experimentos.

El primer conjunto de experimentos muestra el rendimiento de un único nodo con CPU y GPU para la diversas implementaciones del algoritmo de análisis de imagen. La Figura 4.11 recoge los tiempos de ejecución y el tiempo de sobrecarga de cada implementación para la imagen pequeña. Las cuatro primeras barras corresponden a implementaciones exclusivas en CPU, mientras que las seis restantes introducen una o dos GPUs a las pruebas. Las barras etiquetadas con el prefijo DC son aquellas implementadas con el *middleware* DataCutter, y las no etiquetadas corresponden a implementaciones serializadas básicas.

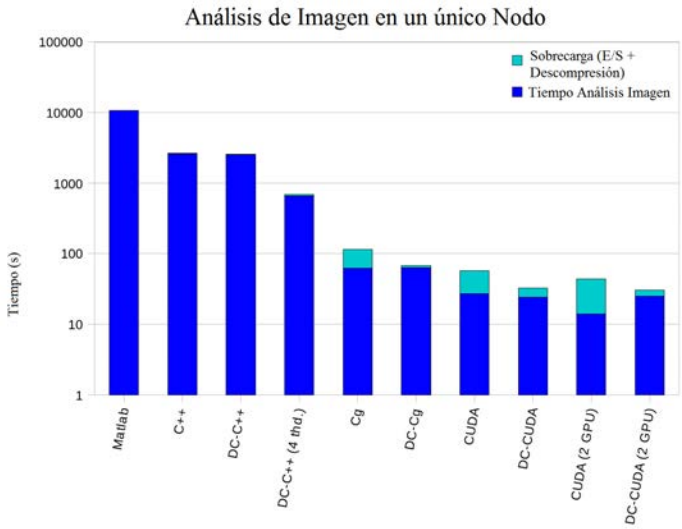


Figura 4.11: Comparativa de los tiempos de ejecución para todas las implementaciones del código de análisis de imágenes sobre un único nodo cuando la imagen es pequeña.

El tiempo de ejecución en la Figura 4.11 (representado como la porción oscura en la parte baja de la barra) corresponde única y exclusivamente al análisis real de la imagen, mientras que la sobrecarga (representada en la parte superior de la barra en un color más claro) se debe a las operaciones de disco (entrada/salida), la descompresión TIFF, la llamada remota de ejecución y las latencias de red. En este experimento, lo más reseñable es el factor de aceleración de orden 3x entre las versiones basadas

en CPU y la versiones basadas en GPU. Aunque el movimiento desde Matlab hacia C++ representan un gran aumento de rendimiento, en CPU sigue siendo más lento que en GPU. Adicionalmente, la versiones con DataCutter del algoritmo de análisis de imágenes basadas en GPU recortan el tiempo de ejecución final para la imagen completa gracias a que la naturaleza desacoplada y multi-hilo de DataCutter permite solapar el análisis real de la imagen con la descompresión TIFF y la entrada/salida a disco. Desafortunadamente, la implementación en CUDA es lo suficientemente rápida para que la descompresión TIFF se convierta en el cuello de botella cuando se emplean dos GPUs. En este caso, la espera de las GPUs para recibir las imágenes impide que las dos GPUs se aprovechen al 100 %. Dado que cuatro hilos de C++ muestran una clara ventaja en los resultados respecto al uso de un único hilo, de aquí en adelante los resultados supondrán la versión de DataCutter con cuatro hilos de análisis de imagen por nodo.

La Figura 4.12 refleja la comparativa de rendimiento de las implementaciones basadas en GPU de la rutina de análisis para las tres imágenes. La idea principal que se puede extraer de esta gráfica es que hay una relación lineal entre el tiempo de ejecución y el tamaño total de la imagen analizada en todas las implementaciones. Desafortunadamente, incluso para las imágenes más grandes, las sobrecargas asociadas entorpecen el uso de dos GPUs. Por esta razón, a partir de ahora no mostraremos dichas variantes en nuestros experimentos.

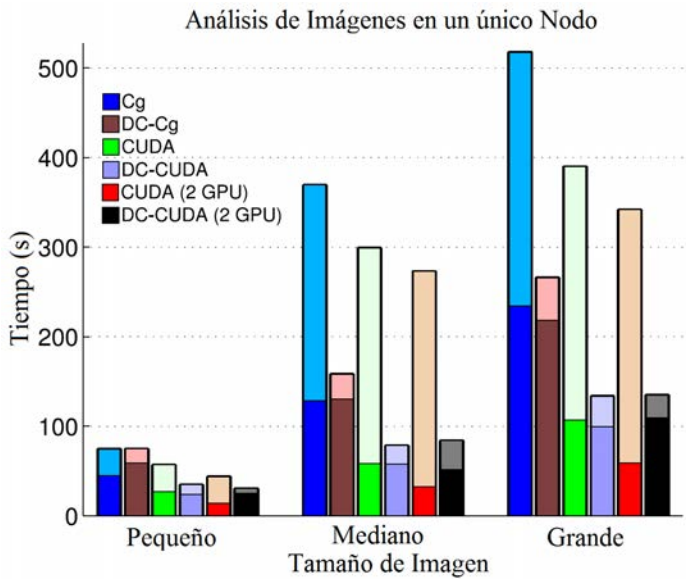


Figura 4.12: Comparativa de tiempos de ejecución de las implementaciones en GPU y DataCutter sobre un único nodo para los tres tamaños de imagen.

La Figura 4.13 muestra la escalabilidad de nuestra solución respecto al número de nodos. El número de nodos involucrados en el análisis de las imágenes son 1, 2, 4, 8, 12 y 16, y se representan por barras en la figura de izquierda a derecha, seis en cada grupo de color. Al igual que en la Figura 4.12, la porción inferior de color más oscuro de cada barra corresponde al tiempo de análisis de la imagen, mientras que la porción superior de color claro representa la sobrecarga añadida. Este tipo de procesamiento de análisis de imágenes escala muy bien, de manera que los tiempos de ejecución decrecen casi linealmente con el número de nodos. En este sentido, el tiempo total del análisis de la imagen para la implementación DataCutter/CUDA cuando la ejecución se lanza sobre 16 nodos es de cuatro segundos por debajo para la imagen pequeña, siete segundos por debajo para la imagen mediana, y en torno a 11 segundos para la imagen grande. En comparación con el tiempo de ejecución de un único nodo con Matlab sobre tres horas para la imagen pequeña y aproximadamente 12 horas para la imagen grande, la mejora representa un notable incremento de la productividad. Con la intención de aumentar la legibilidad de las gráficas, hemos acortado su representación para un único nodo con C++. Sus valores son 629.42 segundos para el análisis de la imagen y 29.7 segundos para la sobrecarga. Además, dado que el interés principal de este trabajo son los resultados de la GPU y que los resultados en C++ se muestran bien a escala en el peor caso (debido a que provoca la sobrecarga proporcional más pequeña), se ha eliminado dicho tiempo de las figuras que muestran los resultados para las imágenes mediana y grande.

La Figura 4.14 muestra el factor de aceleración logrado al aumentar el número de nodos. Tal y como se percibe en las figuras, existe una relación aproximadamente lineal debido a que las imágenes pueden descomprimirse y procesarse de forma completamente independiente en cada nodo. Sin embargo, un tiempo de ejecución tan pequeño para las implementaciones basadas en GPU provoca que las diferentes sobrecargas (configuración remota del arranque, red y latencias de la descompresión TIFF) empiecen a tener un peso porcentual importante en el tiempo total y de una magnitud comparable. En el caso de 16 nodos, la implementación CUDA necesita al menos 1.80 segundos de computación para procesar 2,145 de las subimágenes resultantes de la descomposición de la imagen original. Sin embargo, pese a la disposición de tres filtros para la descompresión TIFF, sólo el tiempo de descompresión para la asignación de nodos de 135 imágenes puede oscilar entre 0.3 y 1.3 segundos.

El factor de aceleración puede ser prácticamente lineal en un entorno de producción si asumimos que la invocación del proceso remoto sólo se da en una ocasión para un conjunto de imágenes analizado. En estas circunstancias, las únicas sobrecargas se deberían al sistema de entrada/salida y a las latencias de red.

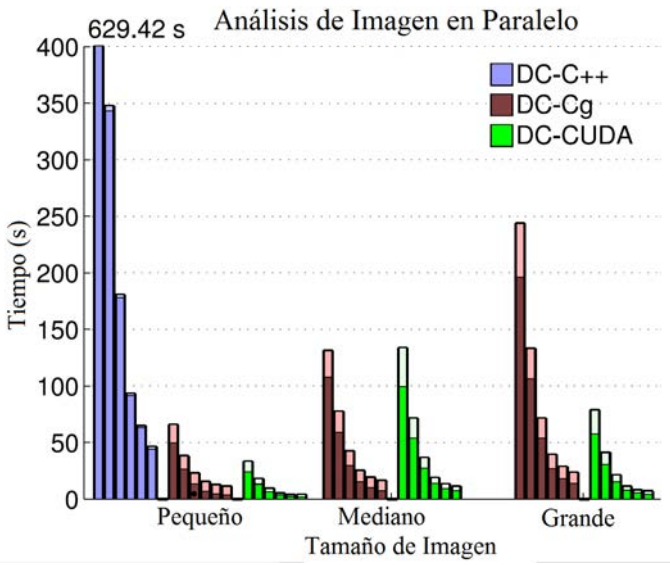


Figura 4.13: Tiempos de ejecución en paralelo de C++, Cg y CUDA basados en la implementación con DataCutter para las tres imágenes y variando el número de nodos desde 1 hasta 16.

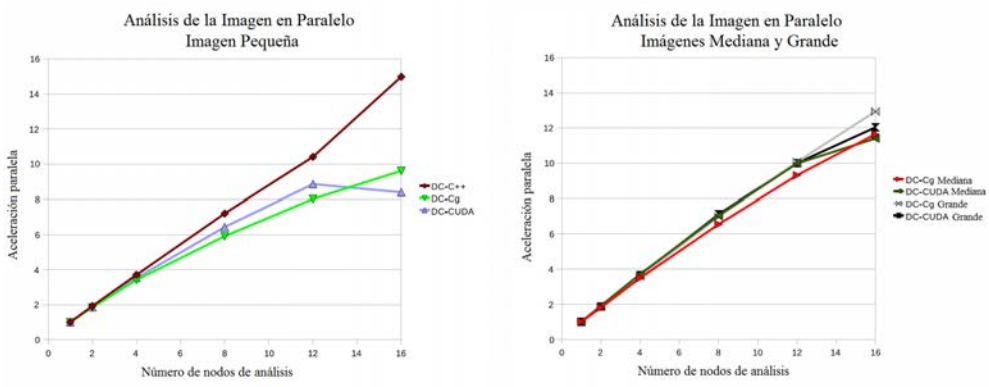


Figura 4.14: Resultados del factor de aceleración al paralelizar la aplicación. Dentro de las versiones con DataCutter, a la izquierda se muestra la versión de una CPU asistida por dos GPUs para la imagen pequeña, y a la derecha la misma versión para las imágenes mediana y grande.

4.4. Resumen de resultados

Este capítulo presenta la evaluación del rendimiento para nuestro análisis biomédico de imágenes en diferentes tecnologías y plataformas de CPUs y GPUs. Partien-

do desde la tecnología más clásica para programar las GPUs con la librería gráfica OpenGL y una plataforma híbrida sencilla de una única CPU y GPU, se empiezan a obtener claras evidencias de que el uso de la GPU para este tipo de análisis biomédicos permite ahorrar grandes cantidades de tiempo, aunque no por igual en todas las fases de computación. El diseño de la aplicación biomédica comienza en un lenguaje de alto nivel interpretado (Matlab) para definir un prototipo que posteriormente se traslada al lenguaje de programación C++ con uso exclusivo de CPU y otra versión con uso de GPU. Tras un análisis de las operaciones más adecuadas para cada tipo de procesador, el primer intento de plataforma híbrida consigue tiempos de ejecución alrededor de 45 veces más rápidos respecto a la versión C++ ejecutada exclusivamente en CPU, y 321x más rápidos frente a la versión prototipo en Matlab. La GPU es especialmente ventajosa cuando las operaciones a realizar requieren un gran ancho de banda, mientras que otras tareas como las matrices de co-ocurrencia son más apropiadas para la CPU.

Un entorno más complejo compuesto de un clúster cooperativo de CPUs y GPU, y empleando la tecnología CUDA y DataCutter para paralelizar la computación internamente y entre los nodos, supone una sólida plataforma multiprocesador cooperativa y heterogénea en la que se puede explotar paralelismo a múltiples niveles. Concretamente, hasta cuatro niveles de granularidad de la arquitectura *hardware* y la aplicación *software*: (1) multi-nodo (usando DataCutter para el particionado de datos entre nodos), (2) SMP y a nivel de hilo (usando DataCutter para utilizar todos los recursos dentro de cada nodo y de cada procesador), (3) SIMD (usando CUDA para llenar todos los multiprocesadores de las GPU), y finalmente, (4) ILP (Paralelismo a nivel de instrucción), configurando los bloques de hilos dentro de la GPU de manera que siempre exista trabajo en la cola de procesos.

Los resultados experimentales muestran el éxito de las técnicas empleadas, comenzando con el decremento de los tiempos de ejecución en un solo nodo CPU/GPU por el uso de diferentes optimizaciones intra-nodo. La ganancia se extiende al paralelismo inter-nodo para una ejecución escalable del sistema multiprocesador. En el análisis del conjunto de imágenes más grandes, para la configuración de clúster de 16 nodos, la implementación más simple de GPU DataCutter-CUDA es 31.3 veces más rápida que su implementación serie en CUDA. Al introducir dos GPUs por nodo, el tiempo de procesamiento de cada nodo cae por debajo del minuto, de manera que si se excluye el tiempo de entrada y salida y la descompresión de imágenes, queda demostrado su potencial. Además, cuando se emplea DataCutter para solapar los tiempos de computación con los de entrada y salida y descompresión, la GPU permanece con carga de trabajo constante. El resultado de esta aproximación llega a acelerar los tiempos en un factor de 12.94 sobre 16 nodos.

5 Registro no rígido de imágenes microscópicas

La caracterización de fenotipos asociados a genotipos específicos es fundamental para esclarecer las actividades de los genes y su interacción. La morfología de la estructura celular y tisular son aspectos del fenotipo que proporcionan la información necesaria para conocer procesos biológicos esenciales, tales como la iniciación del cáncer en el microentorno del tumor y la formación de redes neuronales en la corteza del cerebro. Sin embargo, las técnicas actuales que obtienen en la información tridimensional magnificada desde las muestras biológicas son bastante limitadas. Para este propósito suele emplearse un microscopio de fluorescencia multifotón y cofocal que necesita marcadores fluorescentes y tiene un campo de visión limitado. Por tanto, la reconstrucción a través de las imágenes 2D obtenidas de la sección tisular y el microscopio óptico representa una buena alternativa para recopilar información 3D. Dicha tarea se basa en el emparejamiento (*matching*) de imágenes 2D de secciones finas empleando la técnica de registro de imágenes [10, 11, 15, 17, 18, 23, 29, 32, 34, 35, 41, 42, 46, 50, 51, 66, 68, 69, 72, 79, 81].

La operación clave para el registro de imágenes en este escenario consiste en compensar la distorsión entre imágenes de secciones consecutivas que se produce tras el corte transversal. Estos cortes son tan delicados como finos (3 a 5 μm). El proceso de preparación (por ejemplo, seccionado, tintado y sellado) puede añadir una variedad de deformaciones no rígidas entre las que se encuentran la flexión, la erosión, la elongación o el desgarro. A resoluciones microscópicas, una mínima deformación es bastante apreciable, y en no pocas ocasiones la precisión es esencial para el éxito del proceso. Con el objetivo de compensar estas deformaciones, el uso de un registro no rígido es esencial, y su éxito depende de la exactitud de las características seleccionadas para emparejar todo el conjunto de imágenes. En aras a fomentar la precisión, la

intensidad de los píxeles se encuentra en permanente evaluación, lo que desemboca en un proceso muy costoso computacionalmente, sobre todo si se emplean métricas de comparación ya consolidadas como la Información Mutua (IM).

En este capítulo se desarrolla un método de computación de alto rendimiento para generar un emparejamiento preciso de los rasgos o características (*features*) de las imágenes entre todas las muestras obtenidas a partir del microscopio. Los retos que deben abordarse para seleccionar y correlar el conjunto de imágenes son muy diversos. Destacan entre ellos los siguientes:

1. **Entorno rico en características.** La calidad de la textura obtenida a través de las imágenes microscópicas representa una oportunidad única para seleccionar una amplia gama de características sobre las que luego cimentar un robusto emparejamiento. La detección tradicional de estas características, por ejemplo, para la localización de las esquinas, genera un abundante número de parámetros que normalmente perjudican el realismo del emparejamiento final.

2. **Imágenes a gran escala.** Los escáneres actuales de alta resolución pueden generar imágenes con resoluciones de 0.46 μm/píxel (con objetivos de 40x) e incluso superiores. Así, un desplazamiento de 1 mm. supone más de 2,000 píxeles. Por tanto, sin una buena inicialización, el emparejamiento de características se convierte en una tarea inviable.

3. **Elevada carga computacional.** Un método habitual para el registro de imágenes de alta resolución consiste en aplicar un registro multiescala donde se reduce la resolución para poder procesar la transformación no rígida, y una vez obtenida esta transformación, se aplica las imágenes de mayor resolución. Este proceso no es trivial porque hay que realizar la transformación de las imágenes flotantes en cada escala. Por ejemplo, en este capítulo las imágenes tratadas tienen una resolución de hasta 23 K x 62 K píxeles y, con un factor de escala de dos, la transformación de las imágenes involucra en torno a 12 K x 31 K píxeles como antesala de la etapa final.

Para abordar estos retos, se ha desarrollado una alternativa de alta computación para el registro de imágenes, compuesta de un registro rígido rápido para la inicialización, y un algoritmo eficiente y paralelizable para el registro no rígido implementado en procesadores gráficos (GPUs). Dicho enfoque tiene las siguientes ventajas:

1. **Implementación en GPU.** La idoneidad de la GPU para nuestro propósito se pone de manifiesto en diversos estudios [9, 58] que subrayan su paralelismo masivo, escalabilidad y bajo coste, rasgos todos ellos muy deseables para el

registro de imágenes. Adicionalmente, CUDA mejora esta percepción al ofrecer un modelo de programación alternativo que no requiere conocimientos sobre renderizado o gráficos y permite transformar la tecnología de la GPU en un procesador paralelo en cualquier PC.

2. **Registro rígido rápido para la inicialización.** El algoritmo de registro no rígido se inicializa con un algoritmo de registro rápido. Este algoritmo usa regiones anatómicas destacadas (por ejemplo, vasos sanguíneos) como características de alto nivel, y la transformación rígida se lleva a cabo con un esquema de votado sobre el espacio de transformación euclídea. Este procedimiento es altamente eficiente y preciso para imágenes histológicas, con la ventaja de poderse ajustar tanto las rotaciones como las traslaciones de forma arbitraria. Esto nos permite comenzar desde un gran punto de partida, reduciendo significativamente el espacio de búsqueda del emparejamiento objetivo.

3. **Selección de características.** Las características para conseguir un emparejamiento preciso en el método de registro no rígido se seleccionan en base a la complejidad de las proximidades, más que en base a su geometría. De esta forma, a la vez que se reduce la carga computacional, se ofrece al usuario una distribución más uniforme de las características.

4. **Correlación cruzada normalizada (NCC)** [1] **rápida para un emparejamiento preciso.** La búsqueda precisa de características se basa en el método NCC entre los mosaicos vecinos en cada imagen. El cálculo de NCC puede implementarse eficientemente con la transformada rápida de Fourier (FTT), obteniendo ejecuciones más rápidas que métodos como MI. Por otro lado, NCC tiene una interpretación intuitiva que simplifica la selección de los parámetros empleados como umbrales para discriminar entre buenos y malos emparejamientos.

5. **Transformación simple de la salida.** Para un emparejamiento preciso, los parámetros de la transformación euclídea procedentes de la inicialización rígida se usan para localizar y transformar los vecinos correspondientes, así se evita aplicar la costosa transformación rígida sobre la imagen completa.

6. **Paralelización para el emparejamiento.** El proceso de emparejamiento preciso es altamente paralelo, prestándose a ser ejecutado en múltiples núcleos, procesadores o clústeres de computación.

La implementación en GPU lleva a cabo la parte del algoritmo más exigente computacionalmente: el cálculo de la correlación cruzada para un emparejamiento

[1] *Normalized Cross Correlation.*

de imágenes preciso, lo que supone hasta un 60 % del tiempo de ejecución total. Los resultados de la ejecución del algoritmo se exponen en sus versiones serie y paralela, tanto en CPU como en GPU, y haciendo un recorrido por diversos parámetros para estudiar la escalabilidad y eficiencia (ver Tabla 5.2).

El conjunto de imágenes para las pruebas experimentales (ver Tabla 5.3) procede de dos proyectos de fenotipo cuantitativo:

- El primer proyecto es un estudio morfométrico basado en el gen retinoblastoma (un supresor de tumores bien conocido) del desarrollo de la placenta de los ratones. En este estudio, se obtienen tres placentas de control y mutantes con el gen Rb eliminado. Cada muestra se ha seccionado a 3 μm y cada sección se ha tintado usando hematoxilina estándar y eosina. Las secciones tintadas se digitalizan usando un escáner de alta resolución Aperio ScanScope con un objetivo de 20x que produce una resolución de 0.46 μm/píxeles. Las seis muestras constituyen más de 3,000 imágenes con las dimensiones típicas de 16 K x 16 K píxeles, ocupando más de tres terabytes de datos sin comprimir.
- El segundo proyecto es parte de un estudio reciente sobre el microentorno en los tumores del cáncer de mama en ratones. Las imágenes de este estudio son normalmente de 23 K x 62 K píxeles, y alcanzan un tamaño en torno a cuatro gigabytes sin comprimir.

El contenido de este capítulo se estructura de la siguiente manera. Desde la Sección 5.1 hasta la Sección 5.3 se resume el método empleado para el registro de imágenes, incluyendo la revisión del algoritmo rápido de registro rígido en la Sección 5.1, una descripción de la selección de características, el emparejamiento preciso de imágenes en la Sección 5.2, y finalmente un repaso de la transformación de la imagen en la Sección 5.3. Las Secciones 5.4 y 5.5 presentan un resumen de la arquitectura GPU y una descripción de nuestra implementación. Los resultados experimentales se muestran en la Sección 5.6, que posteriormente son analizados en la Sección 5.7.

5.1. Registro de imágenes para la inicialización

Aunque el desarrollo del algoritmo del registro rígido queda fuera de nuestra jurisdicción, resulta interesante conocer algunos pormenores del mismo para luego asimilar las virtudes de la versión de alto rendimiento que hemos llevado a cabo. La inicialización proporcionada por el registro rígido reduce el área de búsqueda empleada para el emparejamiento preciso en el registro rígido, y este hecho provoca una reducción importante de la carga computacional.

La base del algoritmo de registro rígido reside en el emparejamiento de características de alto nivel, como regiones que corresponden a estructuras anatómicas específicas como los vasos sanguíneos o los conductos mamarios. El empleo de características de alto nivel presenta una multitud de ventajas respecto a los métodos que usan características más primitivas como puede ser la detección de esquinas.

En primer lugar, las características de alto nivel son fácilmente segmentables porque corresponden a grandes regiones continuas de píxeles con similitudes cromáticas. Todo lo que se necesita para la extracción es una simple clasificación por color de los píxeles y una secuencia de operaciones morfológicas [71]. Ambas operaciones disponen de implementaciones optimizadas en las librerías comunes de procesamiento de imágenes. Además, a menudo esta extracción puede realizarse con menor muestreo sin perder fidelidad. Por ejemplo, para el cálculo del registro rígido hemos empleado factores de magnificación 5x sobre imágenes 20x. Puesto que el registro rígido únicamente contribuye en la fase de inicialización, donde aún se trabaja con las imágenes magnificadas originales, la pérdida de información por relajar el muestreo no afecta a los resultados finales.

En segundo lugar, el número de características de alto nivel se limita habitualmente incluso en imágenes microscópicas para mantener el número de posibles emparejamientos. Finalmente, estas características puede emparejarse a través de parámetros como la forma y el tamaño, que son descriptores globales. Por lo tanto, el proceso no está limitado a búsquedas locales, pudiendo acomodar imágenes con un rango completo de desalineamiento. El uso de características como la forma y el tamaño como criterio para el emparejamiento reduce además la ambigüedad.

La cuestión fundamental para cualquier plan de búsqueda de características está en la detección de las discordancias. Con esta finalidad se han desarrollado dos planteamientos. Cualquier par de características de emparejamiento pueden generar una transformación rígida especificada por la rotación del ángulo θ y la transformación T si las distancias dentro de la imagen son consistentes. Esto puede ser concebido como gran parte de las transformaciones basadas en los emparejamientos aproximados que deben concentrase alrededor de los parámetros verdaderos en el espacio euclídeo de transformación. Por tanto, la selección de la transformación rígida óptima se puede obtener como el punto de mayor coincidencia dentro de un proceso de votado.

La Figura 5.1 muestra un ejemplo de este proceso. En el caso donde pueda reducirse el número total de características presentes en la imagen (por ejemplo, menos de 10), los resultados del proceso de votado pueden ser menos fiable. Por esta razón, se ha desarrollado un enfoque teórico-gráfico basado en el mismo principio para la búsqueda de características [66].

Este algoritmo tiene tiempos de ejecución razonables incluso en implementacio-

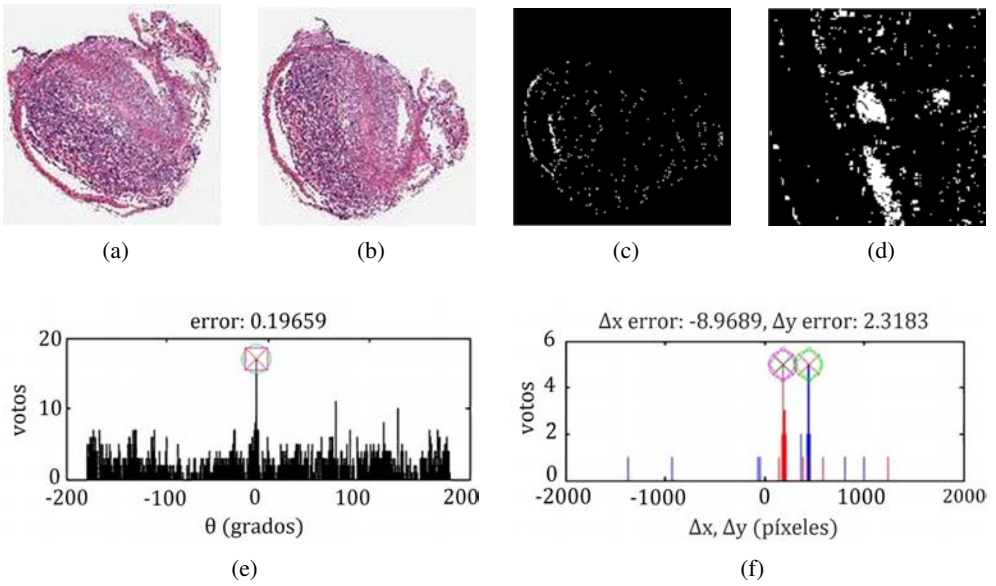


Figura 5.1: Registro rígido rápido usando características de alto nivel.

nes modestas. Usando Matlab, el parámetro rígido estimado para el ejemplo de la Figura 5.1 se calculó en menos de cuatro segundos en un sistema de características similares a los descritos en la Sección 5.6.

5.2. Extracción y búsqueda de características

Una vez descrita la inicialización rígida, toma el relevo el procesamiento del registro no rígido. La corrección de la distorsión no rígida para obtener la precisión necesaria en el fenotipo cuantitativo exige el establecimiento de un gran número de correspondencias precisas. Estas correspondencias deben ser además distribuidas equitativamente a través de las áreas de la imagen que son de interés para obtener una calidad uniforme. Estas consideraciones se remiten a la extracción de características y al emparejamiento, donde los parámetros rígidos de inicialización y el muestreo identifican las regiones correspondientes y son comparadas usando NCC.

5.2.1. Extracción de características

La primera cuestión en la extracción de características es la selección de características no ambiguas que resultan candidatas en emparejamientos más precisos y

específicos. Esto es especialmente importante en el registro no rígido, ya que usar la colección de emparejamientos para interferir en algo sobre la calidad de un emparejamiento individual es difícil debido a la libertad y menor escala de la distorsión no rígida. En este sentido, los emparejamientos en el registro no rígido son de naturaleza local: la única información disponible para juzgar la calidad procede de la vecindad de la característica.

En nuestro caso, los mosaicos seleccionados son de un contenido relevante, una mezcla de diferentes tejidos o tejido y fondo que elaboran una apariencia característica y son candidatos a la generación de emparejamientos muy específicos. Y para asegurar dicho procedimiento, los mosaicos seleccionados tienen una varianza que supera un umbral mínimo establecido. De esta forma, la característica de cualquier punto p con coordenadas $\begin{bmatrix} x \\ y \end{bmatrix}$ y centrado en la ventana W x W píxeles, debe cumplir

$$\frac{1}{W^2 - 1} \sum_{i,j} (t(i,j) - \bar{t})^2 \geq \sigma^2 \tag{5.1}$$

donde t es el patrón, una representación en escala de grises de la ventana de píxeles centrada en p con valor medio $\bar{t}$, y σ^2 es el umbral de la varianza. Existe la posibilidad de que este umbral pueda ser superado, dando lugar a un resultado ambiguo (considera una plantilla pequeña con la mitad superior blanca y la mitad inferior negra dentro de una plantilla similar pero de mayor tamaño), aunque este tipo de casos apenas se presentan en la práctica.

Para mantener una cantidad razonable de características y poder tratar la distribución de características, las características se han muestreado sobre el espacio de las imágenes con un mosaico de W x W. Por ejemplo, en las imágenes de la placenta de 16 K x 16 K, normalmente se crearían mosaicos en el rango de 150-350 píxeles para generar un total de 2025-11236 características posibles, aunque la gran mayoría de las características se descartan por no alcanzar el umbral de la varianza. Con un conjunto de características identificadas en una imagen, sus correspondencias únicas en la imagen siguiente son fáciles de determinar.

5.2.2. Búsqueda de características

Para las características de cualquier punto seleccionado $p1$ con coordenadas $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$ en la primera imagen, en primer lugar se selecciona una ventana de píxeles BxB centrada en $p1$. Esta ventana se transforma a escala de grises y se rota por un ángulo θ obtenido de la inicialización del registro rígido. Posteriormente, el área central

de $W_1 x W_1$ píxeles se usa como plantilla $p1$ para identificar $p2$, lo que sería el punto correspondiente $p1$ en la segunda imagen. El cálculo de B depende de θ y W_1, abarcando suficiente espacio para acomodar el patrón.

La coordenada $p2$ se puede estimar usando la siguiente expresión

$$p2' = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \mathbf{T} \tag{5.2}$$

donde $\mathbf{T}$ corresponde con el vector de traslación obtenido de la inicialización del registro rígido. Un patrón de $W_2 x W_2$ píxeles centrado en $p2'$ y designado como ventana de búsqueda se toma de la segunda imagen. El NCC se procesa entre la plantilla y la ventana de búsqueda, y el centro del área que se corresponde con el mayor valor NCC se pone en $p2$. Si este valor máximo sobrepasa un umbral (normalmente 0.8 o superior), el emparejamiento se considera correcto y $p1$ y $p2$ se guardan como correspondencias.

La Figura 5.2 refleja este proceso. La selección de W_1 y W_2 se basan en la severidad de la deformación y en la capacidad computacional. Empíricamente se ha fijado $W_2 = 2W_1$. Una deformación superior necesita una tamaño de ventana de búsqueda W_2 superior. Los resultados experimentales se han evaluado con diferentes tamaños de W_1 y W_2 (ver Sección 5.6), con algunos de ellos favorables en CPU y otros en GPU (ver Tabla 5.2).

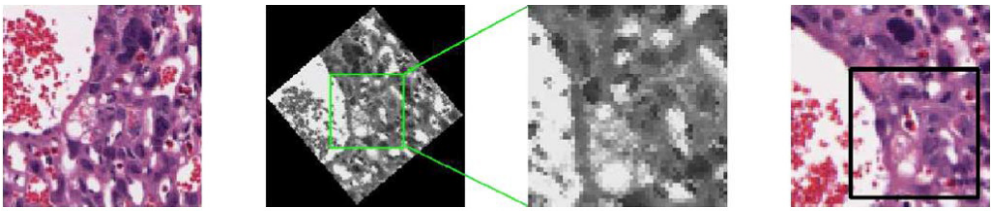


Figura 5.2: Proceso de emparejamiento de características. (a) Imagen inicial. (b) Imagen inicial rotada por el ángulo obtenido en el registro rígido. (c) Región seleccionada dentro de la primera imagen (parche patrón o ventana de características de $W_1 x W_1$ píxeles). (d) Región de búsqueda dentro de la segunda imagen (ventana de búsqueda de $W_2 x W_2$ píxeles).

Entre las medidas de similitud comúnmente utilizadas para la información de la intensidad y que son diferentes a NCC están la raíz cuadrada de la suma de las diferencias (SSD) y MI. SSD no es una buena elección para las imágenes microscópicas porque el contenido tiende a ser discreto (por ejemplo, los límites entre el núcleo celular, el citoplasma y la membrana celular). MI se usa habitualmente como una métrica

en estrategias de búsqueda por gradiente, pero el histograma conjunto provoca que la búsqueda exhaustiva sea muy costosa computacionalmente. Por tanto, se ha seleccionado NCC porque, además de su robustez en la identificación de similitudes, es altamente eficiente cuando se implementa con la FFT. La robustez es un parámetro clave en la aplicación, ya que resulta muy frecuente encontrar variaciones de intensidad debido a que las secciones no son uniformes en grosor y tintado. Por otro lado, los valores de NCC tienen una interpretación tan intuitiva que la selección de los parámetros umbrales es relativamente fácil.

5.2.3. Procesamiento de NCC

La gran cantidad de características que existen dentro de un conjunto de datos convencional hace que el procesamiento eficiente de NCC sea una parte fundamental. Además, más que usar una estrategia de búsqueda, NCC se procesa entre el patrón y las ventanas de búsqueda dentro de todos los desplazamientos posibles con el fin de evitar problemas de mínimos locales.

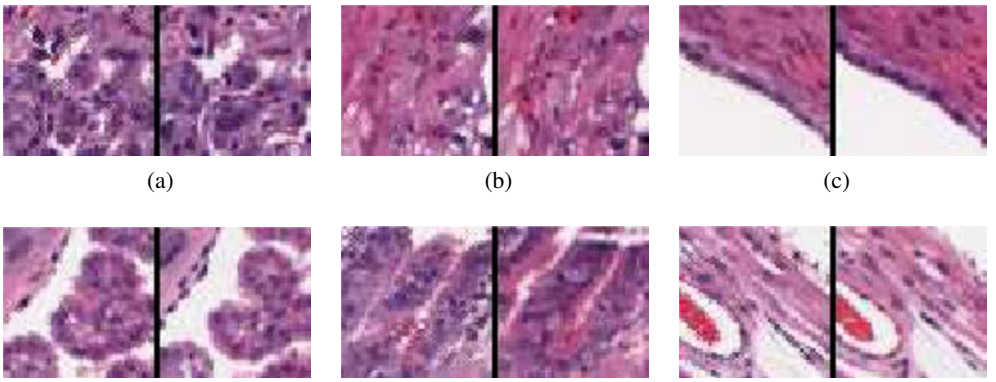
Dado un patrón t de tamaño $W_1 x W_1$ con media $\overline{t}$, y ventana de búsqueda s con tamaño $W_2 x W_2$, $W_2 > W_1$, el NCC entre t y s es el cociente de la covarianza y las varianzas individuales:

$$\rho(u,v) = \sum_{x,y} \frac{\{t(x-u,y-v)-\overline{t}\}\{s(x,y)-\overline{s}_{u,v}\}}{(\{t(x-u,y-u)-\overline{t})\}^2\{s(x,y)-\overline{s}_u,v\}^2)^{\frac{1}{2}}} \tag{5.3}$$

donde $\overline{s}_{u,v}$ es la media de la parte de la ventana de búsqueda que se solapa con el patrón en un desplazamiento (u,v).

Para calcular los factores de normalización en el denominador se usa el método de suma acumulada [47]. Este método evita los costosos cálculos locales de la media y la varianza de la ventana de búsqueda para la región solapada cuando el patrón se desplaza a través de las posiciones $(W_1 + W_2 - 1)^2$, reduciendo el número de operaciones de $3W_2^2(W_1 - W_2 - 1)^2$ a aproximadamente $3W_1^2$.

La correlación cruzada no normalizada del numerador se calcula a través del teorema de convolución de la Transforma de Fourier Discreta (DFT,) que relaciona el producto del espectro de la DFT con la convolución circular en el dominio espacial. Respecto a las correlaciones cruzadas, en este capítulo es de especial interés la correlación convencional, por lo que s y t se rellenan con ceros hasta el tamaño W_1+W_2-1 antes de aplicar la transformada para asegurar que en el resultado no existen porciones de solapamiento circular. La Figura 5.3 muestra algunos ejemplos seleccionados aleatoriamente de las regiones emparejadas.



(a) (b) (c)

(d) (e) (f)

Figura 5.3: Áreas seleccionadas aleatoriamente correspondiente al emparejamiento de características entre dos imágenes.

5.3. Transformación de la imagen

La colección de puntos de correspondencia generados por el procesado de emparejamiento proporciona la información necesaria para calcular las reglas de transformación de una imagen en conformación con la otra. En la práctica se usan una variedad de reglas de mapeo no rígido, diferenciadas por la carga computacional, la robustez con las correspondencias erróneas, y la existencia de formas invertidas. La implementación de alto rendimiento del proceso de transformación no es una tarea trivial y no ha sido contemplada en este capítulo.

5.3.1. La transformación polinomial

En la selección de un tipo de transformación, se ha buscado aquella que sea capaz de corregir distorsiones complejas, robusta al emparejamiento de errores, que admita formas invertidas y que sea computacionalmente eficiente.

De los tipos de mapeo no rígidos de uso común, tales como la interpolación *Thin-Plate Spline*, las medias ponderadas locales, el mapeo polinomial, y las variaciones definidas a intervalos, se ha seleccionado el mapeo polinomial. *Thin-Plate Spline* proporciona una asignación mínima de energía, siendo atractiva para los problemas que afectan a la deformación física real, pero un emparejamiento perfecto en los lugares de correspondencia puede potencialmente ocasionar una gran distorsión en otras áreas y un error de mapeo excesivo en caso de que exista una correspondencia errónea. La escasez de una forma inversa implica que la imagen transformada debe ser calcula-

da en una dirección progresiva, probablemente dejando huecos en el resultado de la transformación. Existen métodos como la búsqueda de gradiente que pueden utilizarse para paliar este problema, pero a costa de un incremento de la carga computacional que puede ser astronómica cuando se aplica para la transformación en cada píxel en una imagen de un gigapíxel. Los métodos basados en núcleos, como la media ponderada local, requieren una distribución uniforme de las correspondencias y, dada la heterogeneidad de las características de los tejidos, no puede ser garantizada.

La deformación polinómica admite una forma inversa, es rápida de aplicar, y ha sido capaz de corregir experimentalmente la distorsión encontrada en las imágenes seccionadas de forma satisfactoria. Los parámetros de la deformación polinómica pueden calcularse usando mínimos cuadrados o variaciones de mínimos cuadrados que puedan mitigar los efectos de los errores de emparejamiento. El mapeo afín ofrece beneficios similares pero está más limitado en las deformaciones que puede representar.

En nuestro algoritmo, se ha empleado polinomios de segundo grado. En concreto, para un punto (x, y) en la imagen de referencia, la coordenada (x', y') de su correspondencia en la segunda imagen es

$$\begin{cases} x' = a_1x^2 + b_1xy + c_1y^2 + d_1x + e_1y + f_1, \\ y' = a_2x^2 + b_2xy + c_2y^2 + d_2x + e_2y + f_2, \end{cases} \tag{5.4}$$

Como cada par de puntos de correspondencia viene dado por dos ecuaciones, se necesitan al menos seis pares de puntos de correspondencia para resolver los coeficientes de la ecuación 5.4. En la práctica, el número de puntos de correspondencia es mucho mayor.

5.3.2. Reconstrucción 3D

En aplicaciones de reconstrucción 3D, donde tenemos que registrar una secuencia de imágenes, el proceso de emparejamiento se aplica sucesivamente y en orden a cada par de imágenes de la secuencia. Las imágenes se transforman a partir de un extremo de la secuencia y, en cada fase, las transformaciones previas del par de imágenes se propagan con el fin de lograr un conjunto coherente de imágenes transformadas.

El flujo de trabajo para todo el proceso de registro se expone en la Figura 5.4. Por otro lado, los resultados de la reconstrucción de 50 secciones de una placenta mutante se muestran en la Figura 5.6. Debido a las limitaciones del software de presentación a la hora de manejar grandes conjuntos de imágenes, sólo podemos representar una porción de la reconstrucción.

Fase computacional	CPU	GPU
Dos FFTs	68 %	74 %
Multiplicación por puntos	3 %	2 %
Una FFT inversa	29 %	24 %

Tabla 5.1: Peso porcentual medio de cada una de las fases antes y después de migrar el algoritmo hacia la GPU.

5.4. Registro de la imagen en GPU

El flujo de trabajo del algoritmo de registro se recoge en la Figura 5.4. Desde el punto de vista del rendimiento, el conjunto de Transformadas Rápidas de Fourier (FFTs) empleadas para procesar la NCC en un emparejamiento preciso del registro no rígido, es la parte más importante por ocupar la mayor parte del tiempo de ejecución. Por ejemplo, en nuestro experimento de registro de un par de imágenes de 23 K x 62 K píxeles, más del 60 % del tiempo de ejecución se dedica al procesamiento de la NCC.

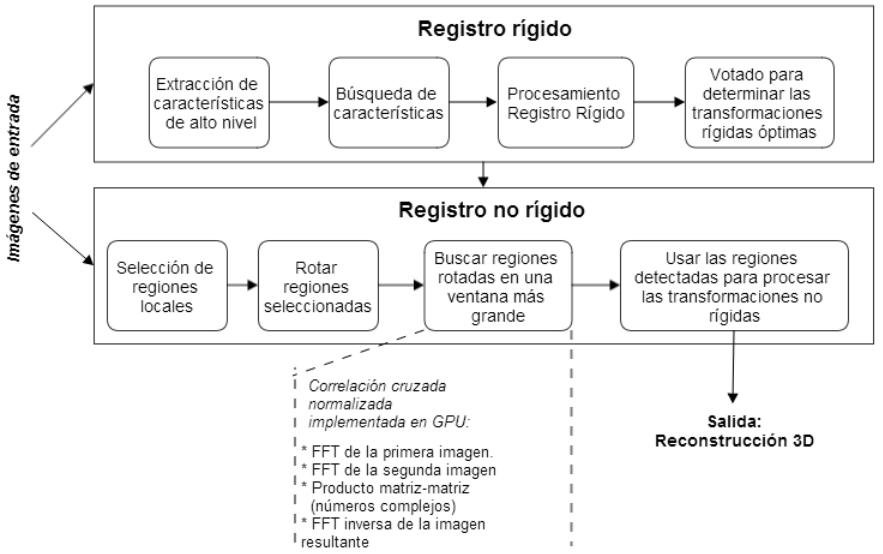


Figura 5.4: Flujo de trabajo del algoritmo de registro que consta de dos fases: el registro rígido y no rígido. Cada caja representa una operación independiente que puede ser directamente paralelizada. La fase computacionalmente más exigente se ejecuta en la GPU para conseguir una ejecución mucho más rápida.

Este proceso se ha optimizado con su implementación en GPU (ver Sección 5.4.1), incluyendo la FFT y la FFT inversa. La multiplicación por puntos del espectro de la

FFT que se necesita entre la FFT y su inversa ha sido también implementada en la GPU para conservar el movimiento de datos entre los procesadores y así aprovechar una mayor intensidad aritmética frente a la computación en la CPU. La Tabla 5.1 representa el peso porcentual de las operaciones procesadas en cada plataforma.

Las partes restantes del algoritmo de registro, entre las cuales se incluye el votado, los cálculos de la varianza y las transformaciones simples (por ejemplo, las rotaciones) no se han beneficiado de una aceleración significativa en GPU debido a tres razones principales:

1. El tiempo de ejecución en CPU ya era insignificante.

2. El envío del código y los datos emplea demasiado tiempo, y también la recepción una vez han sido procesados. Hay que tener en cuenta que debemos utilizar el bus de memoria, el enlace *hypertransport* y el controlador PCI-express (ver Figura 1.1). Este coste computacional difícilmente puede amortizarse durante la computación posterior en GPU a pesar de su gran tasa de GFLOPS.

3. La mayoría de las operaciones contienen sentencias condicionales y no son aritméticamente intensivas, para lo cual la CPU es más apropiada. De esta forma, nuestra plataforma habilita los dos procesadores para que se distribuya la ejecución del algoritmo completo según las mejores propiedades de cada uno de ellos.

5.4.1. Correlación cruzada normalizada (NCC) con CUDA

La NCC puede implementarse de manera eficiente en la GPU utilizando el modelo de programación de CUDA. Nuestra estrategia de cálculo se basa en el teorema donde las convoluciones circulares en el espacio geométrico ascienden a la multiplicación por puntos en el espacio discreto de la frecuencia. De este modo, usando la librería CUFFT [55] como una implementación eficiente de la FFT y su inversa, la correlación basada en Fourier puede ser más eficiente que la implementación directa en el dominio espacial, permitiendo además aprovechar la potencia de la representación en punto flotante y el paralelismo de la GPU sin necesidad de desarrollar una implementación personalizada.

La FFT es un algoritmo altamente paralelizable al que se le puede aplicar la filosofía "divide y vencerás" para el cálculo de la Transformada Discreta de Fourier de una o múltiples señales. El teorema de la convolución incumbe a la imagen (ventana de búsqueda) y al núcleo de la convolución (ventana del patrón) que comparten los mismos tamaños. En casos como el nuestro, donde la imagen es mayor que el tamaño

del núcleo, el núcleo debe expandirse al tamaño de la imagen como se muestra en la Figura 5.5. Además, la convolución convencional requiere que las ventanas de búsqueda y del patrón sean rellenadas con ceros en los bordes inferiores y de la derecha, tal y como ya anticipamos en la Sección 5.2.3.

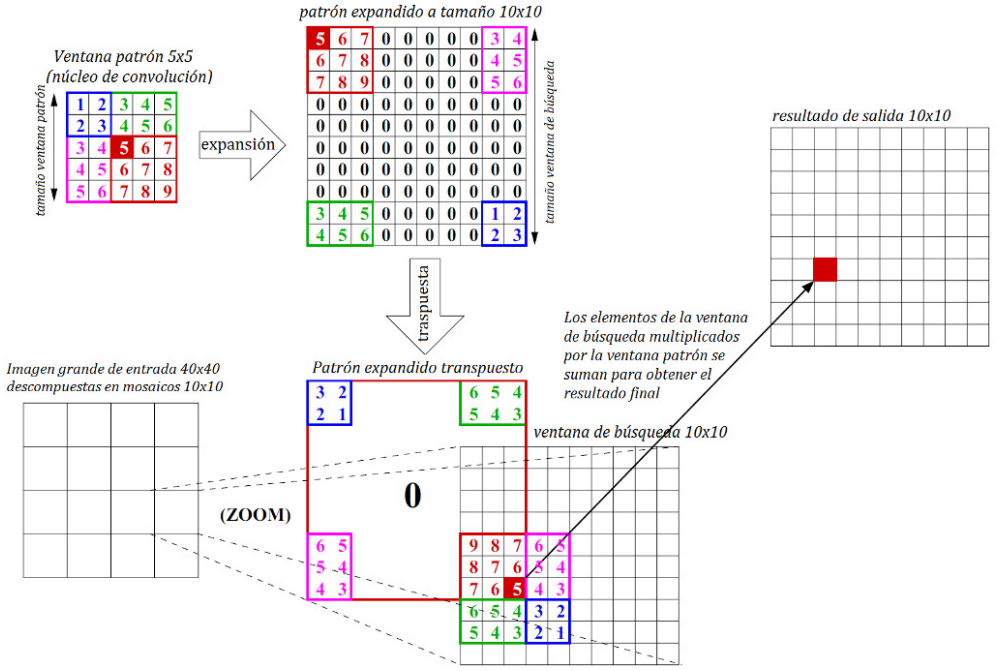


Figura 5.5: Correlación cruzada normalizada basada en la FFT. La ventana patrón tiene que expandirse al tamaño de la ventana de búsqueda para que la convolución con el núcleo sea equivalente a la del núcleo inicial. El ejemplo muestra el caso donde una imagen grande de 40 x 40 píxeles se descompone en 4 x 4 mosaicos. La ventana patrón es de 5 x 5 píxeles, la mitad de la ventana de búsqueda en cada dimensión al igual que en el algoritmo de registro.

Las dos dimensiones de la FFT son parámetros fundamentales en CUDA para optimizar el rendimiento. Cuando la ventana de búsqueda y el patrón son múltiplos de una potencia de dos o de un número primo pequeño, el uso de memoria generado por el algoritmo CUDA minimiza los conflictos de acceso a la memoria compartida y el rendimiento aumenta. Para la implementación en C++ se ha empleado la librería FFTW [33], una de las más populares y eficientes librerías para la FFT en CPU, y de esa forma poder realizar una comparación justa con los resultados de la GPU. La FFTW también favorece el uso de determinadas dimensiones, en este caso el óptimo se alcanza cuando la suma de los tamaños de la ventana de búsqueda y del patrón menos uno es potencia de dos. El conjunto de pruebas creado para la CPU y GPU ha

sido cuidadosamente seleccionado para cumplir los casos óptimos en cada plataforma. La Tabla 5.2 recoge todos los tamaños seleccionados para la evaluación experimental y analiza la idoneidad para cada tipo de procesador.

Imagen de entrada:	Placenta: 16K x 16K			Mamaria: 23K x 62K		
Tamaño de ventana:	Pequeño	Mediano	Grande	Pequeño	Mediano	Grande
Ventana de plantilla (en píxeles)	171	250	342	342	500	683
Ventana de búsqueda (en píxeles)	342	500	683	683	1000	1366
Juntas (plantilla+búsqueda-1)	512	749	1024	1024	1499	2048
CPU óptima (librería FFTW)	Sí	No	Sí	Sí	No	Sí
GPU óptima (librería CUFFT)	Sí	[2]	Sí	Sí	No	Sí

Tabla 5.2: Diferentes tamaños de las ventanas de búsqueda y características en los algoritmos de registro (en píxeles).

Para los casos donde no se cumplen las reglas previas, las librerías FFTW y CUFFT proporcionan un mecanismo simple de configuración que llaman "plan" y que especifica por completo el plan óptimo de ejecución para un tamaño y tipo de datos de la FFT en particular. La ventaja de este "plan" es, que una vez que ha sido creado por el usuario, la librería almacena en un fichero los estados necesarios para ejecutar el plan en varias veces, así consigue evitar la penalización de la preparación minuciosa de transformaciones en tiempo de ejecución. Por ejemplo, con una ventana de planilla de tamaño 350 x 350 píxeles y un tamaño de ventana de búsqueda de 700 x 700 píxeles, la librería FFTW consume alrededor de 0.7 s, mientras que el "plan" de computación sólo consume 0.32 s con una penalización inevitable de 6 segundos para preprocesar dicho plan (coste que puede amortizarse posteriormente cargando el plan en tiempo de ejecución sobre la subsiguientes transformaciones 2D del mismo tamaño).

5.5. Escenario de pruebas

5.5.1. Datos de entrada

Para evaluar nuestras implementaciones y optimizaciones, el algoritmo de registro se aplica a una serie de imágenes microscópicas tomadas de las secciones consecutivas de dos muestras bien diferentes:

1. La placenta de un ratón para un estudio morfométrico en el ámbito del gen retinoblastoma.

[2] Este hueco es parcialmente favorable para la GPU porque 749 es múltiplo de siete, un número primo pequeño.

2. Las glándulas mamarias para estudiar el microentorno del tumor del cáncer de mama [79].

La Tabla 5.3 recoge los detalles de los conjuntos de imágenes empleados. El objetivo en ambos casos es la reconstrucción 3D de los tejidos para su estudio microanatómico.

Campo de estudio	Área de investigación y objetivos biomédicos	Zona del ratón	Carga computacional	Tamaño de imagen (píxs.)	Número de secciones
Genética	Estudio funcional de un gen	Placenta	Media	16K x 16K	100
Oncología	Tumor de cáncer de mama	Mamas	Alta	23K x 62K	4

Tabla 5.3: Conjunto de imágenes de entrada empleadas como datos de entrada para el algoritmo de registro.

5.5.2. Plataforma hardware

La aplicación que procesa el registro automático ha sido implementada en el sistema descrito en la Sección 1.3.2, en un nodo GPU de visualización donde las características de un procesador de doble núcleo AMD Opteron 2218 han sido combinadas con una GPU de doble zócalo Nvidia Quadro FX 5600 (ver Figura 1.1). La CPU va acompañada de 4 GB de memoria DRAM DDR2 a 667 MHz, mientras que la GPU dispone de 1.5 GB de memoria DRAM GDDR3 a 1600 MHz (ver resto de características en la Tabla 1.2). El sistema en conjunto tiene 7 GB de memoria, un disco duro de 750 GB SATA II a 7200 RPM con 16 MB de memoria caché y una tarjeta InfiniBand para la comunicación exterior.

En los experimentos no se ha considerado el tiempo de lectura de los archivos de las imágenes. En cualquier caso, este tiempo se puede ocultar parcialmente solapando las comunicaciones de entrada/salida con el procesamiento en GPU gracias a que la comunicación entre ambos procesadores es asíncrona.

5.5.3. Software

La implementación en GPU se ha desarrollado con la versión 1.1 de CUDA, y para aplicar el paralelismo hacia las dos GPUs hemos empleado la librería *p-threads* (POSIX threads).

En la CPU hemos utilizado el compilador de Microsoft Visual Studio 2005 8.0. Además, una implementación previa en Matlab 7.1 ha servido como herramienta para

el prototipado y validación de los resultados, a la vez que como tiempo de ejecución de referencia.

5.6. Resultados experimentales

Tal y como refleja la Tabla 5.3, una numerosa cantidad de experimentos se han llevado a cabo sobre 100 imágenes de conjunto de la placenta y sobre cuatro imágenes para el conjunto de las mamas.

5.6.1. Resultados del registro de imágenes

Una evaluación directa de la calidad de los algoritmos para el registro de imágenes microscópicas es una tarea complicada debido a dos razones principales: (1) la falta de una referencia ideal y (2) la validación del algoritmo no atañe al contenido de este capítulo. La demostración que se ha realizado sobre el registro de algunas muestras ha sido visual tal y como se muestra en la Figura 5.6.

En otro estudio que no hemos presentado aquí se aprecia que la discrepancia entre las imágenes queda reducida a diez píxeles, siendo esta cantidad insignificante en comparación con el tamaño de los datos tratados. Dicha discrepancia está dentro de lo esperado por la diferencia morfológica entre las imágenes. Además de comparar los resultados del registro rígido y no rígido de imágenes, se han analizado las posibilidades para que el método de registro sea útil para el objetivo de la reconstrucción 3D.

Como se demuestra en la Figura 5.6f, las imágenes registradas se apilan y se muestran usando una presentación volumétrica. Las secciones cruzadas virtuales generadas demuestran la necesidad del registro no rígido para este caso. Es evidente que los resultados del registro no rígido dirigen la reconstrucción suave de las estructuras microscópicas mientras que el registro rígido dirige los límites sobresalientes de la estructura.

5.6.2. Caracterización de la carga de trabajo

Hay que destacar que el tiempo de ejecución para cada imagen dentro del mismo conjunto experimenta variaciones debido al contenido y a que, como consecuencia de ello, el número de características procesadas es diferente. Como se describe en la Sección 5.2.1, la varianza se calcula en una ventana de 200 x 200 píxeles para conservar solamente los puntos que son representativos. Este procedimiento puede provocar que

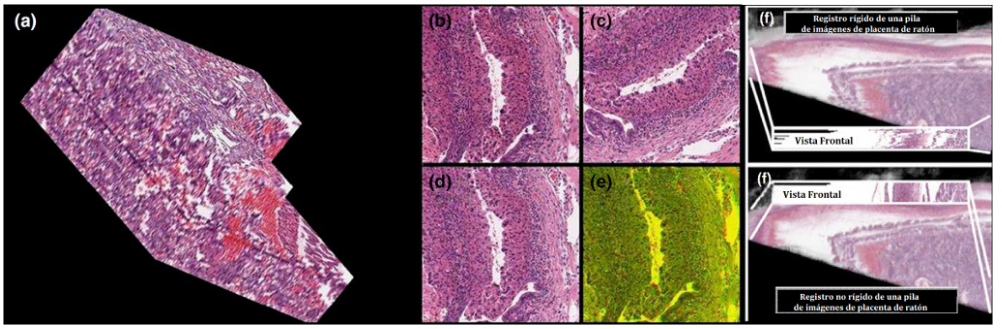


Figura 5.6: (a) Ejemplo de reconstrucción 3D de la placenta de ratón. Dado que las imágenes son de grandes dimensiones, sólo se muestra una fracción de la reconstrucción correspondiente a 30 secciones. (b-e) Registro de las imágenes de glándulas mamarias: (b) área de 1000 x 1000 píxeles de la imagen de referencias; (c) el correspondiente área de 1000 x 1000 píxeles de la imagen flotante; (d) el área de la imagen flotante después de la transformación no rígida; (e) solapamiento de las dos imágenes. (f) imágenes registradas (arriba las imágenes del registro rígido y abajo las del registro no rígido) apiladas y presentadas usando un renderizado en volumen. La vistas frontales son las secciones cruzadas virtuales generadas tras el apilamiento 3D.

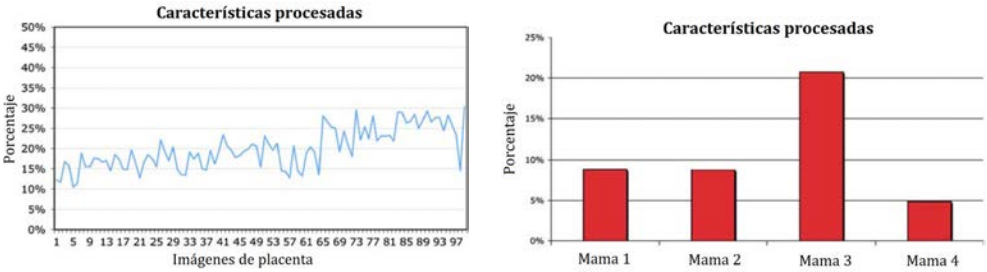
imágenes de diferente tamaño generen una carga computacional diferente basada en el contenido (cuanto más homogénea es la imagen, menor es la carga computacional). La Tabla 5.4 resume el número de características extraídas para cada imagen de entrada perteneciente al conjunto de datos mamario, así como el tiempo de ejecución y total para completar el algoritmo de registro en una CPU Opteron.

	Número de características extraídas			Carga de trabajo (en segundos)	
Tamaño ventana: (patrón, búsqueda)	Pequeño (342,683)	Medio (500,1000)	Grande (683,1366)	Tiempo de ejecución con E/S	Tiempo de ejecución sin E/S
Mama 1	1196	655	384	650.86	558.54 (85 %)
Mama 2	1048	568	312	497.83	414.17 (83 %)
Mama 3	3119	1528	854	1320.01	1192.69 (90 %)
Mama 4	690	322	168	463.77	340.62 (73 %)

Tabla 5.4: Tamaños de las ventanas de búsqueda y características empleadas en los algoritmos de registro (en píxeles).

El porcentaje de características procesadas oscila entre 4 % y 30 % del área total de la imagen, variando ligeramente estos valores según el tamaño de la ventana (ver Tabla 5.2). Sin embargo, se puede considerar que los porcentajes son estables para cada imagen si se selecciona el tamaño más pequeño de imagen como la más representativa (resolución de búsqueda mayor). Bajo esta hipótesis, la Figura 5.7 proporciona

los detalles sobre el porcentaje de las características procesadas para el conjunto de imágenes de la placenta y de las mamas. El mínimo porcentaje para las imágenes de la placenta ocurre en la imagen 5 con un 10.48 %, mientras que el máximo se da en la imagen 99 con un 30.38 %, y una medida total de 19.88 %. En las imágenes de las glándulas mamarias el mínimo porcentaje es 4.82 % para la imagen 4, con un máximo de 20.71 % para la imagen 3, y una media de 10.77 %. De acuerdo a nuestra definición de característica, el conjunto de imágenes de la placenta contiene aproximadamente el doble de información relevante, mientras que el conjunto de las glándulas mamarias representa una matriz con una tasa superior de dispersión incluso con un tamaño superior de imágenes.



(a) Conjunto de imágenes de la placenta.

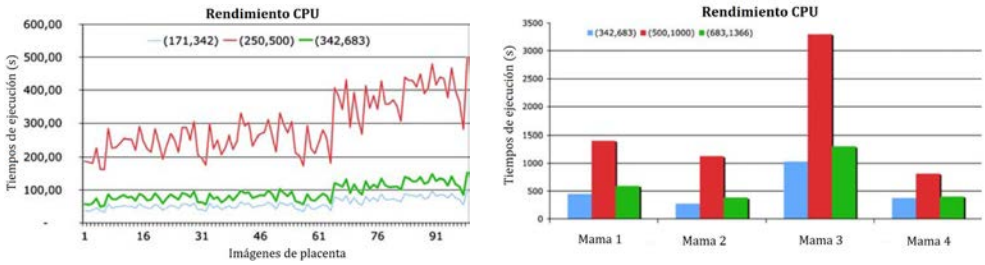
(b) Conjunto de imágenes mamarias.

Figura 5.7: Porcentaje de características procesadas por imagen en cada conjunto de imágenes de entrada. El tamaño para la ventana de búsqueda y de plantilla es el pequeño por ser el más representativo.

5.6.3. Tiempos de ejecución en CPU

La Figura 5.8 muestra los tiempos de ejecución para el algoritmo de registro representado en la Figura 5.4 cuando se procesa completamente en la CPU usando la librería FFTW. Los resultados para el conjunto de imágenes de la placenta y las glándulas mamarias se muestran a la izquierda y derecha, respectivamente. Dentro de cada caso, hemos realizado las pruebas con tres patrones y ventanas de búsqueda diferentes (ver Tabla 5.2): pequeñas (azul, más a la izquierda), mediana (rojo, céntrico) y grande (verde, más a la derecha). Según los consejos proporcionados por la librería FFTW, los tamaños pequeño y grande cumplen las condiciones óptimas, mientras que el tamaño mediano rompe todas las reglas. Este hecho ralentiza la ejecución, con un tiempo medio para el caso de la placenta de 294.57 segundos para el tamaño mediano, 57.97 segundos para el tamaño pequeño y 91.33 segundos para el tamaño grande. El resultado supone un incremento del 57 % cuando se duplica el tamaño de la ventana dentro de las condiciones óptimas y un 222 % cuando no se cumplen dichas condiciones.

El comportamiento para las imágenes mamarias es similar, aunque los incrementos se reducen al 26 % y 147 % respectivamente, con tiempos de ejecución medios de 530.41 segundos (tamaño pequeño), 1660.91 segundos (mediano) y 669.96 segundos (grande).



(a) Conjunto de imágenes de la placenta.

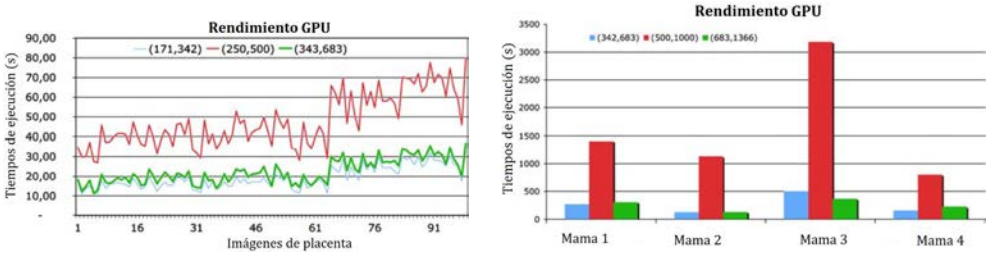
(b) Conjunto de imágenes mamarias.

Figura 5.8: Tiempos de ejecución en la CPU Opteron para el algoritmo de registro sobre un par de imágenes de diferentes conjuntos de muestra y tamaños de ventana. El primer par de números de la leyenda corresponde con el tamaño pequeño de ventana de búsqueda y plantilla, respectivamente. Los dos pares siguientes se corresponden con el tamaño mediano y grande, respectivamente.

5.6.4. Tiempos de ejecución en GPU

La Figura 5.9 muestra los tiempos de ejecución para el algoritmo de registro cuando la GPU asiste a la CPU procesando la correlación cruzada basada en FFT con CUDA. La gráfica de la izquierda es el resultado para el conjunto de imágenes de la placenta, y a la derecha están los resultados para las muestras mamarias, diferenciándose con leyendas los distintos tamaños de ventana (ver Tabla 5.2). En esta ocasión, los tamaños pequeño y grande cumplen todas las condiciones impuestas por la librería CUFFT, y además el tamaño mediano de 794 píxeles satisface la condición de ser múltiplo de un número primo pequeño (en este caso, 7). Sin embargo, la sobrecarga es aún apreciable. Los tiempos medios para la placenta son de 19.27 segundos (pequeño), 47.80 segundos (mediano) y 22.22 segundos (grande), y la disminución es del 15 % cuando el tamaño de ventana asciende al doble dentro de las condiciones óptimas, y un 115 % adicional fuera de dichas condiciones.

Para las imágenes mamarias, los tamaños grandes consiguen mejorar ligeramente el rendimiento de los pequeños, y la sobrecarga fuera de las condiciones óptimas (tamaño mediano) alcanza el mayor valor: 531 %.



(a) Conjunto de imágenes de la placenta. (b) Conjunto de imágenes mamarias.

Figura 5.9: Tiempos de ejecución en la GPU Quadro para el algoritmo de registro sobre un par de imágenes de diferentes conjuntos de muestra y tamaños de ventana. El primer par de números de la leyenda corresponde con el tamaño pequeño de ventana (de búsqueda y plantilla), y los siguientes, con el tamaño mediano y grande.

5.6.5. Comparativa CPU-GPU

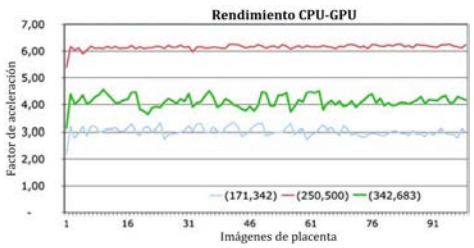
La fila central de la Tabla 5.5 muestra los factores de aceleración media cuando la GPU procesa la correlación cruzada basada en FFT utilizando CUDA. Las ganancias no son estables para los casos que están fuera de las condiciones óptimas, y los resultados más reales se dan en los tamaños pequeño y grande, donde los tamaños de la ventana siguen estrictamente las indicaciones marcadas por las librerías FFTW y CUFFT. En el caso de la placenta, las imágenes pequeñas generan un factor de aceleración de 3.00x mientras que para las grandes se alcanza 4.11x. En el caso de las glándulas mamarias, dichos factores son más modestos: 2.00x y 2.59x, respectivamente.

Imagen de entrada:	Placenta: 16K x 16K			Mamaria: 23K x 62K		
Tamaño de ventana:	Pequeño	Mediano	Grande	Pequeño	Mediano	Grande
(búsqueda, patrón)	(171,342)	(250,500)	(342,683)	(342,693)	(500,1000)	(683,1366)
CPU tiempo ejec.	57.97	294.57	91.33	530.41	1660.91	669.96
GPU tiempo ejec.	19.27	47.80	22.22	264.09	1629.72	257.95
GPU factor acel.	3.00x	6.16x	4.11x	2.00x	1.01x	2.59x
Tiempo 2 GPUs	13.13	26.05	13.66	225.17	837.51	234.62
2 GPU / 1 GPU	1.46x	1.83x	1.62x	1.17x	1.94x	1.09x
2 GPU / 1 CPU	4.41x	11.30x	6.68x	2.57x	1.98x	2.85x

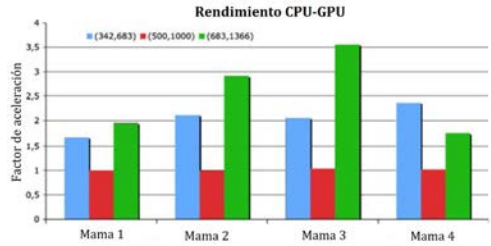
Tabla 5.5: Tiempos de ejecución (en segundos) y factores de aceleración para las diferentes implementaciones desarrolladas para el algoritmo de registro con máximo rendimiento.

La Figura 5.10 demuestra que el factor de mejora en la GPU tiene gran dependencia de la imagen de entrada, especialmente para el conjunto de imágenes mamarias

donde los números son menos consistentes. Adicionalmente, estas ganancias son más inestables conforme aumenta el tamaño de las ventanas. Este efecto se debe a que el contenido de las imágenes más grandes empieza a ser más heterogéneo en grandes búsquedas, mostrando por otro lado disparidades entre las imágenes. La Figura 5.8 corrobora dicho efecto.



(a) Conjunto de imágenes de la placenta.



(b) Conjunto de imágenes mamarias.

Figura 5.10: Comparativa entre los tiempos de ejecución de la CPU y la GPU en términos de factores de aceleración. Cuando el tamaño de las ventanas aumenta, los tiempos de ejecución son más irregulares en (b). El primer par de números de la leyenda corresponde con el tamaño pequeño de ventana (de búsqueda y plantilla), y los siguientes, con el tamaño mediano y grande. (a) El factor medio para la placenta es de 3.00x (pequeño), 6.16x (mediano) y 4.11x (grande). (b) Para las mamas el factor medio es de 2.00x (pequeño), 1.01x (mediano) y 2.59x (grande).

5.6.6. Paralelismo y escalabilidad en la GPU

La popularidad conseguida por la GPU durante la última década se debe a la espectacular escalabilidad de su arquitectura, manteniéndose en su objetivo de duplicar su rendimiento cada seis meses. Además de la tendencia *intra-chip*, otras iniciativas como la tecnología SLI de Nvidia y Crossfire de ATI han surgido para explotar el paralelismo inter-chip (SMP - MultiProcesamiento Simétrico). La iniciativa ha alcanzado un notable éxito dentro de la industria de los videojuegos, pero su impacto sobre la comunidad GPGPU ha sido bastante menor.

Esta sección evalúa el rendimiento del algoritmo de registro sobre un par de GPUs cuando se aplica paralelismo SMP. Nuestras técnicas de programación son extensibles a cualquier número de tarjetas gráficas, ya que los métodos usados para particionar los datos garantizan perfectamente la escalabilidad del problema. Sin embargo, en este ambicioso algoritmo hay que advertir el papel crítico que asume el sistema de entrada/salida: docenas o incluso cientos de GPUs que trabajan en paralelo pueden encontrar un método fácil para distribuir las diferentes ventanas de búsqueda de una

manera eficiente cuando tratan con grandes imágenes, pero el sistema de archivos tendría que ser de tal rendimiento que permitiese leer cada imagen en paralelo con un ancho de banda sostenible y suficiente para conseguir una tasa de procesamiento cercana al Teraflop.

Durante los experimentos, este cuello de botella no ha sido analizado en detalle, pero la Tabla 5.4 cuantifica en las dos últimas columnas el tiempo de ejecución total (incluyendo entrada/salida) y el tiempo de cómputo (excluyendo entrada/salida), revelando así que la interacción con disco es responsable del 10-20 % del tiempo total de ejecución. Este tiempo no se incluye en los análisis posteriores, ya que es el mismo tanto para la CPU como para la diferentes versiones de la implementación en GPU y no es el objeto de este trabajo adentrarnos en el sistema de almacenamiento. Implícitamente se asume que los datos de las imágenes se encuentran en la memoria DRAM o que se pueden recuperar de una manera eficiente utilizando un sistema de archivos en paralelo o en RAID.

Una vez que los datos se encuentran en la CPU, hay dos alternativas básicas para distribuir la carga computacional del algoritmo de registro entre múltiples GPUs: por bloques o cíclica. Para el caso concreto de un par de GPUs, pero sin perder generalidad, la distribución por bloques asigna la mitad superior de la imagen a una GPU y la mitad inferior a la otra. La distribución cíclica es justo al contrario, numera las diferentes imágenes para enviar las impares a una CPU y las pares a la otra. Dado que las características más interesantes de las imágenes tienden a estar concentradas, la distribución por bloques es más congruente y ha sido la seleccionada para nuestros experimentos.

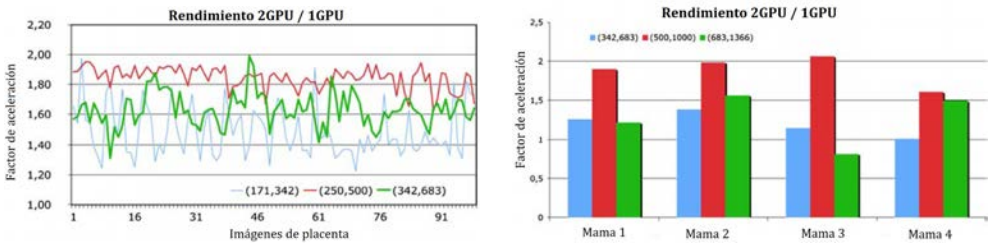
Nuestro método de paralelización funciona de la siguiente forma: se crea un hilo para cada región de la imagen que calcula la varianza en la CPU en cuestión para evaluar si merece la pena procesarla. Si se supera esta prueba, la imagen se envía a una GPU predeterminada para procesar la correlación cruzada normalizada y la búsqueda de características. La Tabla 5.6 resume el número de imágenes procesadas y descartadas en cada GPU dependiendo de la imagen de entrada procedente del conjunto de imágenes mamarias. El desequilibrio de la carga computacional está entre un 2.76 % de la imagen 2 y un 13.33 % de la imagen 4, en sentido creciente conforme es menor el número de imágenes a procesar (tasa de dispersión de la imagen de entrada).

Finalmente, la Figura 5.11 desvela que la ganancia obtenida cuando una segunda GPU entra en juego es muy diversa, empezando con 30-50 % para tamaños pequeños de ventana, continuando con 60 % para tamaños grandes de ventana, y acabando con una escalabilidad óptima (100 %) en tamaños medianos. Estas ganancias son proporcionales a la carga computacional, demostrando que la GPU es más escalable cuanto más puede explotar su intensidad aritmética. O lo que es lo mismo, los GFLOPS no

Imagen de entrada	Procesador	Número de mosaicos		Desequilibrio de la carga de trabajo (%)	Tiempo de ejec. (s.)
		Testeados	Procesados/descartados		
Mama 1	GPU 1	1672	196/1476	4.08	260.41
	GPU 2	1672	188/1484		
Mama 2	GPU 1	1496	158/1338	2.53	101.32
	GPU 2	1496	154/1342		
Mama 3	GPU 1	1872	428/1444	2.76	522.43
	GPU 2	1911	426/1485		
Mama 4	GPU 1	1786	78/1708	13.33	225.37
	GPU 2	1786	90/1696		

Tabla 5.6: Número de mosaicos procesados y descartados para cada imagen en el conjunto de imágenes mamarias en cada GPU bajo la ejecución paralela en dos GPUs.

están limitados por la escasez de datos procedente de un ancho de banda insuficiente entre la memoria de vídeo y la GPU.



(a) Conjunto de imágenes de la placenta.

(b) Conjunto de imágenes mamarias.

Figura 5.11: Escalabilidad de la GPU: factores de mejora cuando se habilita una segunda GPU. El primer par de números de la leyenda corresponde con el tamaño pequeño de ventana (de búsqueda y plantilla), y los siguientes, con el tamaño mediano y grande. (a) El factor medio para la placenta es de 1.46x (pequeño), 1.83x (mediano) y 1.62x (grande). (b) Para las mamas el factor medio es de 1.17x (pequeño), 1.94x (mediano) y 1.09x (grande).

5.7. Conclusiones

Tras un análisis de los experimentos se pueden extraer la siguientes conclusiones:

1. El conjunto de imágenes de la placenta muestra factores de aceleración superiores en la plataforma gráfica gracias a que las imágenes tienen un contenido más relevante, conducente a una mayor carga computacional que explota mejor

la intensidad aritmética y el ancho de banda. Además, un numero pequeño de características procesadas implica una alta presencia de sentencias condicionales en el código, uno de los rasgos más perjudiciales para el rendimiento de la GPU.

2. La mayor escalabilidad se consigue con el conjunto de imágenes de la placenta, y sus ganancias son más estables entre los diferentes tamaños de ventana. La gran disparidad de las imágenes mamarias juegan un papel negativo en la distribución de la carga, introduciendo desbalances e impidiendo que el paralelismo sea completamente explotado.

En conjunto, la GPU consigue un factor de aceleración de 3-4x en la mayoría de los escenarios típicos (texto enmarcado en Tabla 5.5) comparado con la CPU, y un par de GPUs muestra una escalabilidad satisfactoria, aunque ganancias inestables bajo diferentes imágenes y tamaños de ventana.

6 Optimizando los momentos de Zernike sobre Kepler

6.1. Introducción

Este capítulo pretende analizar la arquitectura Kepler de Nvidia, prestando atención a los dos puntales estrella de sus multiprocesadores SMX: paralelismo dinámico y planificación Hyper-Q. Para ello hemos utilizado un algoritmo que computa los momentos de Zernike para la caracterización de imágenes, aplicado en diversas áreas científicas como la biomedicina.

Las funciones de momentos se expresan con cálculos integrales que, aplicadas a imágenes digitales compuestas por píxeles, se transforman al dominio discreto. Aunque la operación realizada puede ser interpretada como una máscara de convolución, los momentos pueden tener atractivas características como la invarianza a la translación de la imagen, al cambio de escala y la rotación [7]. Estas propiedades, junto con la de ortogonalidad, otorgan protagonismo a los momentos de Zernike como descriptores de imagen independientes y mínimamente redundantes [75].

Aunque la compatibilidad de las aplicaciones desarrolladas en CUDA está asegurada en sus nuevas versiones, el rendimiento puede mejorar notablemente si el código se optimiza para una arquitectura concreta, labor que requiere un buen conocimiento de las mejoras software y hardware para tomar las decisiones más acertadas que maximicen el rendimiento. En este sentido, los cambios arquitecturales no son revelados en su completitud por parte de los fabricantes. Nuestro trabajo aquí consiste en aportar nuevos elementos de juicio en base a investigación experimental que permitan guiar al programador en esta ardua labor.

Los contenidos de este capítulo están estructurados de la siguiente forma: La Sección 6.2 describe los momentos de Zernike y el estado del arte de su desarrollo computacional. En la Sección 6.3 mostramos la implementación de partida para los momentos de Zernike con sus peculiaridades para explotar el paralelismo de la GPU. La Sección 6.4 introduce las estrategias a seguir para conseguir un mejor rendimiento con la arquitectura Kepler. Los resultados experimentales y las conclusiones que se derivan de los mismos se exponen en las dos secciones finales.

6.2. Momentos de Zernike

6.2.1. Formulación matemática

Los momentos de Zernike son un conjunto de funciones ortogonales y complejas con unos coeficientes que tienen la propiedad de invarianza a la rotación de la imagen sobre la que se computan. También satisfacen la propiedad de ortogonalidad, esto es, la contribución de cada coeficiente de un momento en particular de la imagen es único, evitando así la redundancia entre ellos. Estas características los señalan como buenos descriptores de imagen. El conjunto de momentos ortogonales de Zernike para una imagen representada por la intensidad de sus píxeles $f(r,\theta)$ con orden p y repetición q se define como sigue [75]:

$$Z_{pq} = \frac{p+1}{\pi} \int_0^{2\pi} \int_0^1 f(r,\theta) V_{pq}^*(r,\theta), r dr d\theta, \tag{6.1}$$

donde $V_{pq}^*(r,\theta)$ son los conjugados complejos de los polinomios de Zernike $V_{pq}(r,\theta)$ cuya representación describen el círculo unidad definido como

$$V_{pq}(r,\theta) = R_{pq}(r) e^{jq\theta} \tag{6.2}$$

siendo p un entero que cumple

$$\begin{aligned} p \geq 0, 0 \leq |q| \leq p, p - |q| = par, j = \sqrt{-1} \\ \theta = \tan^{-1}(y/x), 0 \leq \theta \leq 2\pi \end{aligned} \tag{6.3}$$

Los polinomios radiales $R_{pq}(r)$ son definidos como

$$R_{pq}(r) = \sum_{k=0}^{(p-|q|)/2} (-1)^k \frac{(p-k)!}{k!(\frac{p+|q|}{2}-k)!(\frac{p-|q|}{2}-k)!} r^{p-2k} \tag{6.4}$$

La formulación anterior está expresada en el dominio continuo y en coordenadas polares. La aproximación de la ecuación 6.1 en el dominio discreto para una función imagen $f(x, y)$ de tamaño NxN queda tal que

$$Z_{pq} = \frac{p+1}{\gamma N} \sum_{i=0}^{N-1} \sum_{k=0}^{N-1} f(x_i, y_k) V_{pq}^*(x_i, y_k) \Delta x_i \Delta y_k \tag{6.5}$$

donde $x_i^2 + y_i^2 \leq 1$ y γN es una componente de normalización que corresponde con el número de píxeles dentro del círculo unitario cuyas coordenadas están representadas por

$$x_i = \frac{2i+1-N}{N}, y_k = \frac{2k+1-N}{N} \tag{6.6}$$

Si añadimos a la imagen $f(x, y)$ una rotación con un ángulo α, el momento de Zernike V'_{pq} de la imagen es

$$V'_{pq} = V_{pq} e^{-jq\alpha} \tag{6.7}$$

Como la rotación añadida sólo modifica la fase de los momentos de Zernike, el valor absoluto es invariante a la rotación. Además, lo mismo ocurre cuando las imágenes son modificadas para ser centradas o escaladas, siendo también invariante a las transformaciones lineales. Estas características tan apreciadas en el procesamiento de imágenes hace que los momentos de Zernike sean más adecuados que otros [76], como pueden ser los momentos de Legendre, aunque a cambio de un mayor coste computacional [52].

6.2.2. Técnicas de computación

La Figura 6.1 muestra el pseudocódigo para el cálculo de los momentos de Zernike de un orden n y una repetición m acorde a la ecuación 6.5, y asumiendo un tamaño de imagen de NxN. En la Figura 6.1, la parte real e imaginaria de los momentos de Zernike están expresadas como z_r y z_i, respectivamente.

Los métodos para computar los momentos de Zernike han ido evolucionando con objeto de reducir su complejidad [1]. Desde una perspectiva computacional, los métodos existentes se pueden clasificar en dos grandes grupos: directos e iterativos.

FUNCTION RadialPolynomial(ρ, n, m)
$radial = 0$
for $s = 0$ to $(n–m)/2$

$$c = (-1)^s \frac{(n-s)!}{s!\left(\frac{n+|m|}{2}-s\right)!\left(\frac{n-|m|}{2}-s\right)!}$$

$radial = radial + c * \rho^{n-2s}$
end for
return *radial*

FUNCTION ZernikeMoments(n, m)
$z_r = 0$
$z_i = 0$
$cnt = 0$
for $y = 0$ to $N–1$
for $x = 0$ to $N–1$

$$\rho = \frac{\sqrt{(2x-N+1)^2+(N-1-2y)^2}}{N}$$

if $\rho \le 1$
$radial$ = RadialPolynomial(ρ, n, m);

$$theta = \tan^{-1}\left(\frac{N-1-2y}{2x-N+1}\right)$$

$z_r = z_r + f(x, y)$ * *radial* * cos(m*$theta$)
$z_i = z_i + f(x, y)$ * *radial* * sin(m*$theta$)
$cnt = cnt + 1$
end if
end for
end for
return $\frac{n+1}{cnt}(z_r + jz_i)$

Figura 6.1: Pseudocódigo para el cálculo de los momentos de Zernike.

Métodos directos

Los métodos directos calculan individualmente un momento de Zernike para un orden y una repetición dados, a través de la ecuación 6.5. Es la aproximación más utilizada cuando se precisa de órdenes y repeticiones concretos, tal y como ocurre en la clasificación y segmentación de imágenes, donde sólo ciertos momentos son relevantes para la discriminación [48].

Métodos recursivos

Esta metodología se usa en aplicaciones que necesitan toda la serie de momentos de Zernike para un orden y/o repetición dados, tal y como sucede en los algoritmos de reconstrucción de imágenes. Con estos métodos, el cálculo de momentos aislados no

pueden ser procesados sin la previa realización de un barrido de procesamiento para todos los momentos de un orden o repetición. Los nuevos coeficientes se calculan en base a los obtenidos previamente, y a través de formulas matemáticas que permiten amortizar parte de la computación ya realizada, ya que el punto de partida para cada orden y repetición corresponde con el resultado del coeficiente anterior.

Chong y cols. [16] proponen un particular método recursivo al igual que exponen las técnicas existentes para realizar una comparativa. Además, otros autores han aportado ideas para acelerar el procesamiento de los momentos de Zernike a través de la simetría del espacio de coordenadas polares [36].

6.3. Implementación de los momentos de Zernike

Para la optimización de los momentos de Zernike en la arquitectura Kepler tomaremos como punto de partida la implementación más reciente desarrollada para GPU [48]. Además, de cara a unificar las comparativas con las diferentes versiones de la bibliografía, mediremos los tiempos distinguiendo entre métodos directos y recursivos.

La implementación base defiende el empleo de la metodología directa por ser más apta para la GPU al evitar la dependencia de datos y por ser más eficiente en las aplicaciones donde no se necesitan todos los momentos de un orden o repetición. Además, en su implementación pueden aprovecharse un par de optimizaciones:

- **Paralelismo.** Nuestro algoritmo aplica a cada píxel el mismo conjunto de operaciones de forma independiente, permitiendo un uso natural del paralelismo de datos SIMD utilizado en CUDA.
- **Simetría.** Una de las fases computacionalmente más pesadas de los momentos de Zernike son los cálculos trigonométricos del circulo unidad que se aplican para la transformación del espacio polar a un espacio de coordenadas cartesianas. La simetría de los cuadrantes e incluso de los octantes permite reutilizar muchos de los cálculos [36], logrando aceleraciones de hasta 8x en GPU.

Nuestra implementación en GPU consta de cinco *kernels* que aceptan una imagen de intensidades o escala de grises y devuelven el cálculo de los momentos de Zernike según el método directo. Adaptando el proceso al modelo de programación CUDA, tenemos:

1. **Trigonometría del círculo unidad.** En una primera fase se procesa todo el espacio de coordenadas cartesianas con sus valores trigonométricos (seno y co-

seno) y su distancia, distinguiendo además los píxeles que quedan dentro y fuera del círculo unidad. ésta es la fase que aprovecha la optimización de simetría.

2. **Polinomios de Zernike.** Con las distancias y senos/cosenos procedentes de la fase anterior, se calculan los polinomios de Zernike para cada píxel. El sumatorio de la ecuación 6.5 se realiza sobre un espacio de memoria compartida para evitar accesos reiterados a memoria global de CUDA.

3. **Aplicación a la imagen de entrada.** El resultado del espacio conseguido en la fase anterior se multiplica con la imagen de entrada.

4. **Sumatorio de píxeles.** Se contabiliza la suma de los píxeles comprendidos dentro de círculo unitario a través de un algoritmo de reducción.

5. **Sumatorio de las componentes de cada píxel.** Al igual que en la fase previa, sumamos la componente que cada píxel aporta al momento de Zernike, fusionándola en un único valor mediante un algoritmo de reducción.

Con esta implementación de partida en CUDA, los experimentos son realizados el servidor Yuca descrito en la Sección 1.3.3, la Tabla 1.3 presenta el *hardware* sobre el que realizaremos todos los experimentos. Contaremos con dos GPUs de generaciones diferentes: Una Fermi GF100 que sitúa los tiempos de referencia para la arquitectura antecesora, y otra Kepler GK110 que permite cuantificar las mejoras logradas en los nuevos procesadores SMX con paralelismo dinámico y Hyper-Q.

6.4. Optimizando Zernike sobre Kepler

En esta sección analizaremos las distintas partes del código en las que se pueden aplicar las características de la arquitectura Kepler, aunque como veremos no todas ellas revertirán en mejoras productivas.

6.4.1. Recursividad

Comenzamos describiendo la implementación del método recursivo en GPU, a pesar de que pueda parecer más desafiante que el directo para lograr una ejecución eficiente [48].

Dentro de los métodos recursivos, *q-recursive* es el más actual y eficiente [16] para computar todas las repeticiones de los momentos de Zernike que corresponden a un orden concreto. Los dos primeros momentos calculados corresponden a las dos repeticiones más altas, y a partir de ahí, se obtienen todas las repeticiones progresivamente

más bajas a partir de unas expresiones estáticas que involucran a los dos momentos de repetición inmediatamente superiores.

Esta metodología resulta acertada en CPU cuando el objetivo es computar varios momentos pertenecientes a un mismo orden, aunque para ello es necesario cambiar algunos aspectos de la implementación base que comentamos en la Sección 6.3.

La codificación de este método se va a llevar a cabo siguiendo un proceso iterativo que calcula en cada paso los polinomios de Zernike para una repetición dada a partir de los momentos previamente almacenados. El espacio de memoria aumentará en el mismo factor que el número de repeticiones a calcular, pero la complejidad del algoritmo disminuye y los *kernels* que no dependen de la repetición se pueden amortizar para todas las iteraciones.

Los *kernels* numerados como 1 y 4 en la Sección 6.3 permanecen intactos, mientras los demás sufren los cambios que se detallan a continuación:

- El *kernel* 2 que aplica los polinomios de Zernike a cada píxel debe escindirse en dos, ya que la naturaleza del algoritmo iterativo impide enlazar el procesamiento de los polinomios de Zernike con su aplicación al punto del espacio de coordenadas cartesianas. Ahora tenemos:

  2.1 Polinomios de Zernike en forma recursiva. Procesa los polinomios de Zernike recursivamente. Este *kernel* se ejecuta tantas veces como repeticiones haya para los momentos de Zernike de un orden específico. Cada valor procesado usa su propio espacio de memoria.

  2.2 Aplicación al espacio cartesiano. Los polinomios de Zernike se aplican sobre todo el espacio, junto con las funciones trigonométricas que le anteceden.

- Los *kernels* 2.1, 3 y 5 aumentan su carga de trabajo en el mismo factor que el número de repeticiones, recayendo este trabajo sobre cada hilo, que distingue de forma unívoca la partición del espacio de memoria al que necesita acceder gracias a su identificador de bloque e hilo dentro de éste.

6.4.2. Paralelismo dinámico

El paralelismo dinámico puede aplicarse de varias formas a los momentos de Zernike según la carga de trabajo que se traslade desde la CPU a la GPU. A continuación se exponen diferentes estrategias de paralelismo dinámico que se podrían combinar, y lo que cada una de ellas puede aportar:

1. **Lanzar *kernels* desde la GPU.** Las llamadas de los cinco *kernels* del método directo se trasladan al ámbito de la GPU, de forma que inicialmente se lanza un *kernel* de un único hilo y bloque. Este *kernel* inicial o raíz realiza las llamadas correspondientes al código de los momentos de Zernike desde la GPU al igual que lo hacía la versión convencional desde la CPU.

2. **Calcular una repetición desde cada hilo.** En este caso se aprovecha el paralelismo dinámico para calcular todos los momentos de un orden dado. En la versión convencional y siguiendo el método directo, sería necesario aplicar un bucle que itere tantas veces como repeticiones existan. Aplicando paralelismo dinámico, la implementación sería equivalente a la del punto anterior, con la salvedad de que el *kernel* no tendría un solo hilo, sino uno por cada repetición. Para evitar redundancia en los cálculos, los *kernels* en común para todas las repeticiones serían procesados desde un mismo hilo.

3. **Paralelizar el bucle de los polinomios de Zernike.** El bucle *for* que se requiere en el cálculo de los polinomios de Zernike por el método directo (ver Figura 6.1), es buen candidato para aprovechar el paralelismo dinámico. Cada píxel precisa de dicho cálculo, así que cada uno de ellos lanzará un nuevo *kernel* que procese de forma concurrente el trabajo de ese bucle.

6.4.3. Hyper-Q

El aprovechamiento de Hyper-Q requiere establecer un proceso con varios flujos de ejecución concurrente que no presenten dependencias. En los momentos de Zernike se consigue, al igual que el segundo punto del anterior apartado, cuando necesitamos calcular todas las repeticiones de los momentos de Zernike para un orden específico.

El aprovechamiento de Hyper-Q es transparente al programador. La implementación se lleva a cabo usando *streams* que puedan beneficiarse de las múltiples colas de ejecución concurrente. El aumento del número de colas de 16 a 32 resulta también clave en la nueva arquitectura para aprovechar al máximo el creciente número de procesadores CUDA, ya que ahora, con un número cercano a los tres mil, resulta más probable que un *kernel* sólo pueda ocupar una fracción de éstos. Distribuyendo los *kernels* en *streams* siempre que sea posible, conseguiremos que el remanente de procesadores que haya dejado libre la malla de hilos definida para un primer *kernel* en ejecución, pueda ser aprovechada desde otros procedentes de *streams* adicionales.

6.5. Resultados experimentales

La Tabla 1.3 resume las prestaciones de las GPUs utilizadas durante la evaluación experimental de nuestras optimizaciones.

Hemos empleado un tipo de datos de simple precisión y tomado tamaños de imagen progresivos desde 64x64 hasta 2Kx2K píxeles. El orden máximo de los momentos que se calcula es de 34 debido al límite impuesto por el hardware para el cálculo de los factoriales que aparecen en los polinomios de Zernike.

Momentos de Zernike	GPU	Tamaño de imagen						Mejora	
		64	128	256	512	1024	2048	Mín.	Máx.
$A_{4,*}$	Fermi	0,12	0,17	0,37	1,11	4,08	15,75	0,67x	2,20x
	Kepler	0,12	0,17	0,37	1,11	4,08	15,75		
$A_{8,*}$	Fermi	0,20	0,32	0,74	2,36	8,83	34,71	0,71x	2,35x
	Kepler	0,12	0,17	0,37	1,11	4,08	15,75		
$A_{12,*}$	Fermi	0,29	0,50	1,21	4,08	15,32	60,41	0,71x	2,45x
	Kepler	0,12	0,17	0,37	1,11	4,08	15,75		
$A_{16,*}$	Fermi	0,38	0,72	1,82	6,14	23,49	92,05	0,72x	2,49x
	Kepler	0,12	0,17	0,37	1,11	4,08	15,75		
$A_{20,*}$	Fermi	0,51	0,97	2,50	8,68	33,37	130,93	0,76x	2,54x
	Kepler	0,12	0,17	0,37	1,11	4,08	15,75		
$A_{24,*}$	Fermi	0,61	1,27	3,31	11,76	45,39	176,51	0,76x	2,57x
	Kepler	0,12	0,17	0,37	1,11	4,08	15,75		
$A_{28,*}$	Fermi	0,74	1,59	4,23	15,20	58,20	229,20	0,76x	2,61x
	Kepler	0,12	0,17	0,37	1,11	4,08	15,75		
$A_{32,*}$	Fermi	0,87	2,00	5,27	18,89	73,30	288,76	0,76x	2,64x
	Kepler	0,12	0,17	0,37	1,11	4,08	15,75		
$A_{34,*}$	Fermi	0,95	2,16	5,89	20,96	81,29	319,76	0,78x	2,64x
	Kepler	0,12	0,17	0,37	1,11	4,08	15,75		

Tabla 6.1: Tiempos de ejecución (en milisegundos) para procesar todas las repeticiones de un orden a través del método directo de los momentos de Zernike. Las imágenes de entrada son cuadradas y de dimensiones potencia de dos en un rango entre 64 y 2048 píxeles.

6.5.1. Cambio de arquitectura: SMX

En primer lugar, cuantificamos las mejoras producidas por el cambio de arquitectura sin realizar modificaciones en el código. La Tabla 6.1 recoge los tiempos de ejecución sobre Fermi y Kepler para el algoritmo de partida de la Sección 6.3, a la vez que se comparan con el factor de ganancia.

El factor de mejora para el cambio de plataforma oscila entre 0.67x y 2.64x. La

cota mínima de 0.67x, que supone un aumento de tiempo de ejecución de 1.49x, se produce para el tamaño más pequeño de la imagen, y de igual forma, la aceleración máxima de 2.64x llega cuando la carga de trabajo alcanza las dosis más elevadas.

La GPU como procesador, y su modelo de paralelismo de datos como forma de programación, se lucen más a medida que crecen las imágenes de entrada, lo que explica que la migración a Kepler acelere más conforme aumenta el número de píxeles a procesar. El tamaño de imagen más pequeño consta de $64 \times 64 = 4096$ píxeles que se distribuyen en bloques repartidos equitativamente entre todos los multiprocesadores disponibles. El multiprocesador SM de Fermi puede procesar hasta 2 *warps* de forma concurrente, lo que da un total de 896 píxeles a procesar en nuestra plataforma (2x32x14). El SMX de Kepler llega hasta 4 *warps* para un total de 1664 píxeles a la vez para aprovechar los recursos. Con estos datos, la imagen pequeña de 64 x 64 píxeles aprovecharía sólo el 32.6 % y 18.9 % de los recursos para Fermi y Kepler, respectivamente. En este caso, aunque la arquitectura haya sido mejorada, sus recursos son infrautilizados y es la frecuencia, muy superior en Fermi, la que resulta más determinante en el tiempo de ejecución.

Con un tamaño de 128 x 128 píxeles, la carga de trabajo ya satura a la máxima que puede procesar concurrentemente Fermi, mientras que la GPU Kepler se queda en un 75.7 %. En este tamaño de imagen, el rendimiento tiende a igualarse en ambas arquitecturas, ya que aunque la GPU Kepler no aproveche el 100 % de sus recursos, puede planificar todo el cálculo de una sola vez, mientras que la GPU Fermi necesita una segunda tanda para planificar el remanente de bloques que están a la espera.

A partir de aquí, para imágenes de tamaño superior, la mejora de rendimiento en Kepler aumenta hasta alcanzar su cota máxima con el 100 % de los recursos ocupados en esta nueva arquitectura.

En un análisis vertical, el aumento del orden en los momentos de Zernike supone una ejecución adicional del algoritmo por cada dos órdenes. Esta carga adicional aumenta el tiempo de cómputo a la vez que beneficia ligeramente a la GPU.

6.5.2. Configuración de la carga de trabajo

El incremento del número de núcleos del multiprocesador SMX en un factor de seis respecto a SM predispone a reflexionar sobre si la configuración óptima del tamaño del bloque de hilos puede haber variado respecto a la implementación base. En la GPU Fermi dotada de 32 núcleos, un valor entre 128 y 256 hilos era el óptimo para lograr un factor de ocupación del 100 % mientras no hubiera restricciones respecto al uso de los registros y la memoria compartida. La GPU Kepler, cuyo multiprocesador SMX cuenta con 192 núcleos, merece un análisis más pormenorizado a este respecto.

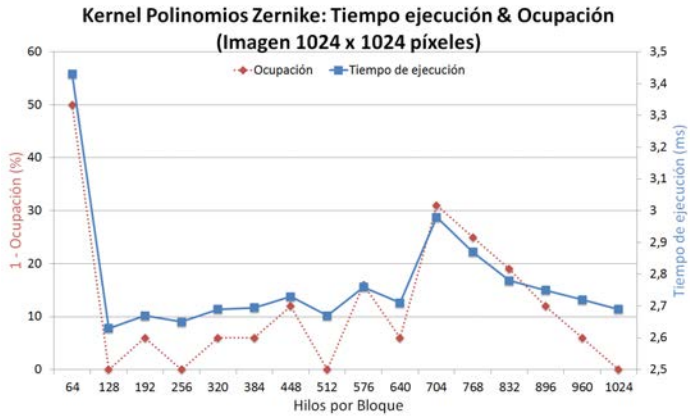


Figura 6.2: Tiempo de ejecución en Kepler para evaluar el rendimiento (escala derecha) en función del tamaño de bloque. Estos valores se contrastan con los teóricos de ocupación para la arquitectura (escala izquierda).

La Figura 6.2 desvela los tiempos de ejecución para el *kernel* que procesa los polinomios de Zernike, siendo éste el más critico en cuanto a recursos. Los tiempos, según la escala de la derecha, se contrastan frente al valor teórico para bloques con diferentes configuraciones de hilos, lo que desemboca en el factor de ocupación según la escala de la izquierda. Aunque teóricamente el mejor rendimiento se consigue para potencias de dos a partir de 128 hilos por bloque, el resultado ha sido levemente mejor para una configuración exacta de 128 hilos, la más baja entre las candidatas. Sin embargo, la mayor divergencia entre la teoría y la práctica se produce cuando el número de hilos por bloque no es suficiente para aprovechar los recursos, o también cuando al añadir un nuevo bloque se sobrepasa el límite de hilos por multiprocesador y, por tanto, hay que sacrificar un bloque en la ejecución concurrente. Ambas zonas se pueden diferenciar:

1. Si el bloque tiene menos de 128 hilos, la limitación para no llegar al número de hilos máximo por multiprocesador la impone el máximo número de bloques permitido. En la GPU Kepler, se pueden ejecutar hasta 16 bloques activos en cada multiprocesador SMX, y la fórmula de la ocupación tiene la siguiente expresión:

$$Ocupaci\acute{o}n = \frac{Hilos * 16}{2048} \tag{6.8}$$

2. Cuando el número de hilos alojados en un multiprocesador se acerca a su valor máximo, la inclusión de un nuevo bloque puede sobrepasar el máximo y, por

tanto, el número permitido de bloques no consigue aprovechar todos los recursos. El peor caso (69 % de ocupación) se da con una configuración de 704 hilos por bloque cuando se aplica la siguiente expresión:

$$Ocupación' = \frac{Hilos * Bloques_{max}}{2048} \tag{6.9}$$

6.5.3. Paralelismo dinámico

Las estrategias de paralelismo dinámico descritas en la Sección 6.4.2 van a permitir conocer cuándo es beneficiosa su aplicación. En la Tabla 6.2 se muestra que la aplicación más sencilla, denominada "Lanzar *kernels* desde la GPU", y la estrategia "Calcular una repetición desde cada hilo" no resultan positivas. Estos resultados se han tomado para momentos con orden y repetición específicos para, además de comparar los resultados con paralelismo dinámico, dar una idea sobre los tiempos de ejecución para momentos individuales.

En el primer caso, los tiempos de ejecución son mayores en un factor entre 1.15x y 2.15x (ver Tabla 6.2a). Esta varianza depende de la carga de trabajo que supone cada llamada a un nuevo *kernel* desde la GPU. Las imágenes de tamaño superior sufren menos penalización al igual que los momentos computacionalmente más exigentes, que son aquellos en los que la diferencia entre el orden y la repetición es mayor.

Para la segunda estrategia, el rendimiento empeora hasta en un factor de 1.4x en el mismo sentido que la estrategia anterior (ver Tabla 6.2b). En este caso, el rendimiento es mejor para las imágenes pequeñas debido a que se procesa un conjunto de momentos que consiguen aprovechar los recursos, mientras que para momentos específicos no sería posible. En resumen, el rendimiento para imágenes pequeñas es sensible a dos aspectos: el propio de lanzar los *kernels* internamente desde GPU, y el que permite paralelizar los cálculos de los momentos que forman parte de la cadena $A_{n,*}$.

La tercera estrategia, denominada "Paralelizar el bucle de los polinomios de Zernike", es muy pretenciosa por su naturaleza dinámica. Sin embargo, las pruebas experimentales dejan ver rápidamente que su rendimiento es catastrófico. La implementación requiere que cada hilo lance un nuevo *kernel*, lo que dispara el tiempo hasta factores 1000x rápidamente. Hemos medido el tiempo que consume una nueva llamada a GPU, resultando entre 5 y 16 $\mu sg.$, mientras que en la CPU consume alrededor de 3 μsg. No parece lógico asumir que el lanzamiento de un *kernel* introduzca mayor sobrecarga cuando se lanza desde circuitería mucho más cercana como la propia GPU, y pensamos que esta rémora se solventará en versiones más maduras de los *drivers*

Momento	Sin P. Dinámico			Con P. Dinámico			Ganancia	
de Zernike	64	256	1024	64	256	1024	Min.	Max.
$A_{0,0}$	0,08	0,10	0,56	0,16	0,20	0,88	0,48x	0,64x
$A_{6,2}$	0,08	0,13	0,93	0,16	0,22	1,26	0,53x	0,74x
$A_{12,0}$	0,09	0,16	1,43	0,16	0,27	1,79	0,58x	0,80x
$A_{25,13}$	0,09	0,17	1,52	0,16	0,27	1,88	0,59x	0,81x
$A_{34,0}$	0,12	0,28	3,07	0,18	0,38	3,51	0,65x	0,88x
$A_{34,18}$	0,10	0,19	1,82	0,16	0,29	2,20	0,60x	0,83x
$A_{34,34}$	0,08	0,10	0,63	0,16	0,20	0,95	0,49x	0,66x

(a) Tiempos de ejecución en milisegundos sin/con paralelismo dinámico.

Todos los momentos	Tamaño de imagen					
para un orden dado	64	128	256	512	1024	2048
$A_{4,*}$	0,73	0,70	0,78	0,70	0,71	0,72
$A_{8,*}$	0,82	0,76	0,71	0,75	0,76	0,76
$A_{12,*}$	0,89	0,82	0,76	0,79	0,79	0,79
$A_{16,*}$	0,91	0,84	0,78	0,80	0,81	0,81
$A_{20,*}$	0,93	0,86	0,81	0,82	0,82	0,82
$A_{24,*}$	0,93	0,87	0,82	0,84	0,84	0,84
$A_{28,*}$	0,95	0,89	0,84	0,85	0,85	0,85
$A_{32,*}$	0,96	0,91	0,85	0,86	0,86	0,86
$A_{34,*}$	0,96	0,90	0,85	0,86	0,86	0,86

(b) Factores de aceleración logrados con paralelismo dinámico.

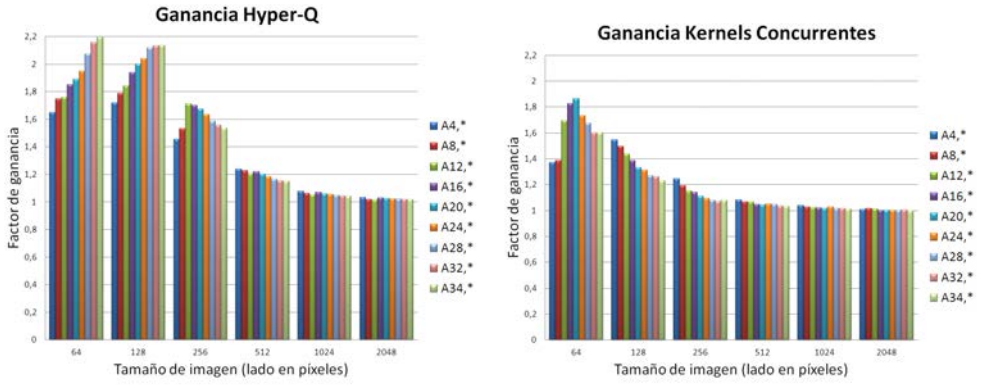
Tabla 6.2: Tiempos de ejecución y factores de aceleración logrados para las diferentes estrategias que explotan el paralelismo dinámico con imágenes cuadradas de diferentes tamaños para momentos de Zernike concretos en el primer caso, y todas las repeticiones de cada orden en el segundo.

y/o implementaciones más maduras de los multiprocesadores SMX.

No obstante, el paralelismo dinámico está orientado a aplicaciones con una filosofía "divide y vencerás" y debe concebirse en la misma línea que la computación con GPUs: Potenciando pocas llamadas a *kernels* con gran cantidad de datos. Otra característica, que limita el ámbito de aplicación del paralelismo dinámico, es la restricción de que cada hilo no puede acceder a la memoria compartida de los *kernels* padres. La información a compartir entre *kernels* padres e hijos queda relegada al uso de la memoria global, con la penalización de rendimiento que esto supone.

6.5.4. Hyper-Q

Para aprovechar Hyper-Q en nuestro algoritmo definiremos un *stream* por cada repetición cuando se persigue calcular todas las repeticiones de un orden. La Figura



(a) Mejoras logradas con Hyper-Q en Kepler.

(b) Mejoras logradas con *Kernels* concurrentes en Fermi.

Figura 6.3: Ganancia obtenida con el uso de *streams* para distintos tamaños de imagen cuando aumentamos el conjunto de momentos de Zernike a procesar.

6.3a compara el factor de ganancia cuando entra en escena Hyper-Q para este supuesto, variando el orden de los momentos de Zernike y el tamaño de la imagen de entrada sobre la que se aplica.

Mientras que la ganancia máxima obtenida es de 2.2x, la ganancia mínima queda a la par con la implementación básica. Esta situación de paridad se produce ya para una carga de trabajo superior a la que abastece a la totalidad de recursos de la arquitectura. Si la imagen es de tal tamaño que la carga de trabajo generada mantiene a la GPU completamente ocupada, no quedan remanentes para ser aprovechados desde *streams* adicionales mediante Hyper-Q, y cada nuevo *stream* acabará serializándose en la cola de trabajo. Por otro lado, en los tamaños de imagen más pequeños, la ganancia aumenta proporcionalmente al número de repeticiones a calcular por cada momento. Este escenario es el ideal para el aprovechamiento de Hyper-Q, ya que la poca carga de trabajo suministrada para cada imagen se compensa dando entrada al procesamiento de otras imágenes en paralelo. Por tanto, el rendimiento máximo se consigue procesando la imagen más pequeña y el momento de Zernike para el orden más alto.

Como el uso de Hyper-Q no supone ningún cambio para el desarrollador y la gestión de colas para procesar los *streams* es transparente, la misma aplicación ejecutada en la arquitectura Fermi nos proporciona una visión de la mejora que supuso la inclusión de *Kernels* Concurrentes. La Figura 6.3b muestra, al igual que se hizo en Kepler, el factor de ganancia obtenido por la incorporación de esta técnica en Fermi. Los valores son similares cualitativamente con la peculiaridad de que, para el tamaño

de imagen inferior donde la ganancia es mayor, la GPU llega a su plena ocupación con órdenes de momento inferiores. Concretamente, el valor máximo 1.86x se consigue para 11 *streams*, situación que corresponde para el cálculo del momento de orden 20 (A_{20}).

Hyper-Q ha supuesto en los momentos de Zernike una aceleración máxima de 1.74x respecto a lo ya conseguido con *Kernels* Concurrentes en Fermi.

El análisis anterior engloba conjuntamente la aplicación de Hyper-Q y el uso pleno de los recursos de los multiprocesadores. Para aislar la aceleración correspondiente a Hyper-Q, hemos realizado un experimento con una imagen de 16 x 16 píxeles (que corresponde con nuestro tamaño del bloque de hilos en CUDA). De esta forma, la ejecución convencional de todas las repeticiones de un mismo orden se realiza secuencialmente y sin aprovechar todos los recursos al enviar un único bloque por iteración. Cuando se usa Hyper-Q o *Kernels* concurrentes, el uso de los recursos se incrementa por la ejecución de las distintas repeticiones en paralelo.

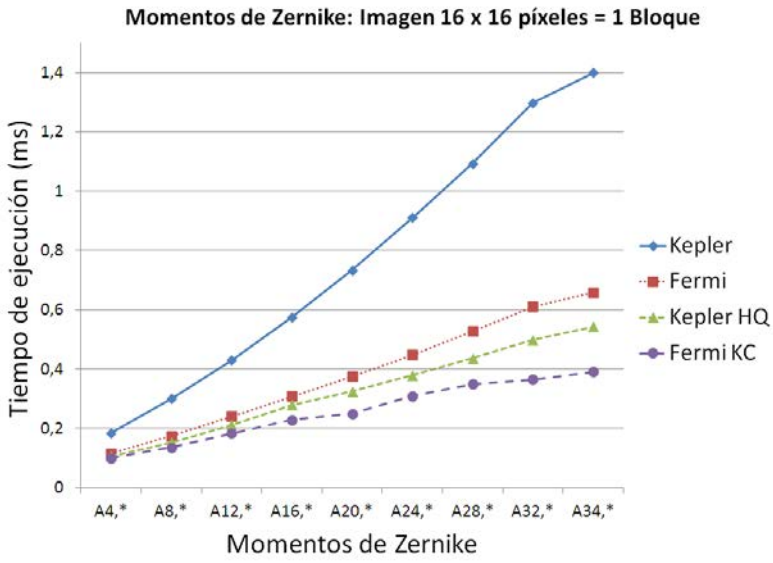


Figura 6.4: Análisis comparativo del beneficio conseguido en la GPU cuando la imagen es de 16 x 16 píxeles, correspondiente a un bloque de hilos CUDA, con objeto de aislar la aceleración atribuida a Hyper-Q y *Kernels* Concurrentes.

La Figura 6.4 muestra los resultados de la comparativa para Fermi y Kepler. Los tiempos cuando no se habilita la paralelización con *streams* favorecen a la GPU Fermi al tratarse de una imagen de entrada muy pequeña, tal y como ya nos ocurrió y explicamos en la Sección 6.5.1 para imágenes de 64x64 píxeles. Cuando se habilita

Hyper-Q en Kepler o *Kernels* Concurrentes en Fermi, la ganancia obtenida es superior en el primer caso (un factor 1.7x), aunque el tiempo de ejecución sigue quedando por encima del obtenido para Fermi. Esto induce a pensar que Hyper-Q resulta un gran complemento para explotar el incremento de núcleos de procesamiento, pero que no aporta tanto a los pilares que despliegan el paralelismo más tradicional de CUDA, léase el que se cimienta sobre los bloques de hilos a nivel intra-multiprocesador y las mallas de bloques concurrentes a nivel inter-multiprocesador. Esta conjetura se consolidará tras la aportación proveniente de la variante recursiva a continuación.

Momentos de Zernike	Tamaño de imagen						Ganancia	
	64	128	256	512	1024	2048	Mínima	Máxima
GPU Fermi								
$A_{4,*}$	0,08	0,12	0,29	0,90	3,31	13,07	1.21	1.50
$A_{8,*}$	0,11	0,17	0,45	1,50	5,78	22,53	1.53	1.9
$A_{12,*}$	0,12	0,23	0,61	2,12	8,12	32,12	1.88	2.35
$A_{16,*}$	0,14	0,28	0,79	2,74	10,58	41,66	2.21	2.67
$A_{20,*}$	0,17	0,34	0,95	3,36	13,13	51,35	2.54	3.07
$A_{24,*}$	0,19	0,39	1,12	3,98	15,59	60,88	2.90	3.26
$A_{28,*}$	0,21	0,45	1,28	4,60	17,97	70,56	3.24	3.56
$A_{32,*}$	0,23	0,50	1,45	5,31	20,41	80,19	3.56	4.02
$A_{34,*}$	0,24	0,52	1,54	5,55	21,64	85,00	3.76	4.12
GPU Kepler								
$A_{4,*}$	0,11	0,12	0,24	0,59	2,01	7,76	0.92	1.62
$A_{8,*}$	0,14	0,16	0,32	0,91	3,27	12,73	1.16	2.05
$A_{12,*}$	0,17	0,20	0,42	1,25	4,54	17,73	1.39	2.34
$A_{16,*}$	0,20	0,24	0,52	1,58	5,82	22,79	1.62	2.58
$A_{20,*}$	0,23	0,29	0,62	1,91	7,11	27,87	1.85	2.88
$A_{24,*}$	0,26	0,32	0,72	2,24	8,38	32,93	2.08	3.06
$A_{28,*}$	0,28	0,36	0,82	2,58	9,66	38,03	2.31	3.42
$A_{32,*}$	0,32	0,41	0,93	2,92	10,95	43,15	2.54	3.52
$A_{34,*}$	0,33	0,44	0,98	3,09	11,59	45,70	2.65	3.66

Tabla 6.3: Tiempos de ejecución (en milisegundos) para imágenes cuadradas de diferente tamaño cuando procesamos todas las repeticiones de un orden específico a través del método *q-recursive* en las GPU Fermi y Kepler. La ganancia de las cuatro últimas columnas se calcula respecto al tiempo obtenido en cada caso mediante el método directo.

6.5.5. Recursividad

Hasta ahora hemos tratado de aprovechar paralelismo dinámico y Hyper-Q para procesar todas las repeticiones de un orden concreto de los momentos de Zernike. ésta es la manera más sencilla de aumentar la cantidad de datos a computar esquivando las dependencias de datos. Sin embargo, la Sección 6.4.1 introdujo algoritmos

que realizan estos mismos cálculos aplicando recursividad. De optar por este camino, tendremos decididamente un algoritmo más eficiente en CPU, pero menores oportunidades de paralelismo en GPU.

La Tabla 6.3 muestra los tiempos de ejecución obtenidos al aplicar el método *q-recursive* en las GPU Fermi y Kepler, contrastándolos en las columnas de ganancia respecto al método directo que computamos anteriormente, donde las oportunidades para desplegar paralelismo son muy superiores.

Los resultados favorecen a la implementación recursiva prácticamente en todos los casos, con la única excepción del orden del momento más bajo para la imagen más pequeña. La ganancia crece conforme aumenta el orden de los momentos y el tamaño de la imagen, y también es superior en Fermi. Dos razones puedes justificar este resultado aparentemente sorprendente: Primero, la implementación recursiva reduce la complejidad de los cálculos y, por tanto, la carga de trabajo exigida a la GPU, lo que perjudica a la plataforma Kepler. Segundo, Fermi parte de un tiempo de referencia bastante más elevado, lo que le otorga un mayor potencial de mejora.

6.5.6. Recursividad frente a Hyper-Q en métodos directos

Las dos subsecciones anteriores tienen como objetivo el cálculo del conjunto de momentos con un orden específico desde dos perspectivas diferentes: Mediante métodos directos que permitan aplicar un mayor grado de paralelismo y Hyper-Q, y utilizando métodos recursivos que sacrifiquen estos mecanismos al introducir dependencias de datos. Una comparativa final entre ambas estrategias determinará si para el cálculo de los momentos de Zernike resulta mejor explotar todos los recursos de la GPU u optar por un algoritmo más hostil al paralelismo pero computacionalmente más eficiente.

La Figura 6.5 muestra la esperada comparativa entre ambas, una suerte de confrontación entre la fuerza bruta y la calidad de la computación. Tanto en Fermi como en Kepler, la implementación recursiva resulta vencedora en GPU, con una tendencia prácticamente lineal respecto al orden del momento y escalable para el rango de órdenes considerado (hasta 34).

Además, los tamaños de imagen más grandes favorecen a la GPU Kepler. Esto se debe a que los multiprocesadores están empleando un mayor número de núcleos en una distribución convencional a través de la malla de hilos en CUDA, y alcanzado el volumen de datos necesario para alimentar a todos los núcleos, la sobrecarga que introduce la gestión más sofisticada de Hyper-Q queda claramente amortizada.

La principal conclusión que podemos extraer de este análisis es que el despliegue

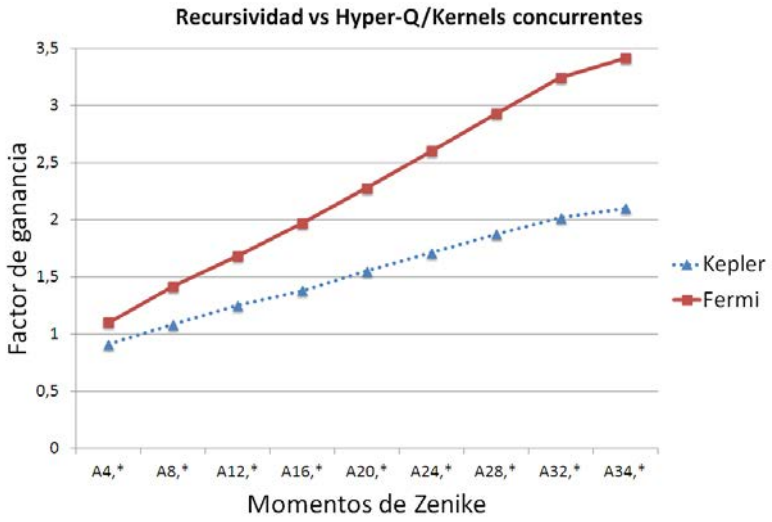


Figura 6.5: Comparativa entre la variante recursiva frente a Hyper-Q en el método directo. Valores superiores a 1 suponen una mejora de la alternativa recursiva.

del paralelismo inherente a los algoritmos debe centrarse en primer lugar en explotar los niveles de hilos por bloque (entre los núcleos del multiprocesador) y bloques concurrentes (entre los multiprocesadores de la GPU), dejando el tercer nivel de kernels concurrentes y su Hyper-Q como baza adicional cuando la carga de trabajo no permita saturar los núcleos de computación disponibles. Esta situación se producirá con asiduidad en el contexto de aplicaciones irregulares en las que el volumen de datos se encuentre repartido en un conjunto ingente de procesos de tal forma que cada uno de ellos involucre a estructuras de datos de tamaño reducido, pongamos inferior al millar de elementos de entrada. También será un supuesto más probable en las futuras generaciones hardware dotadas de un mayor número de núcleos, lo que indica que sólo estamos comenzando a aprovechar el potencial que Hyper-Q esconde como recurso.

6.6. Conclusiones

Este trabajo analiza las posibilidades que las GPUs ofrecen para acelerar el cálculo de los momentos de Zernike, otorgando protagonismo a la arquitectura Kepler y su multiprocesador SMX. El cambio de arquitectura supone un gran avance en rendimiento, resultando una mejora de hasta un 265 % con el mismo código, y muy superior si se aprovechan técnicas como el paralelismo dinámico y Hyper-Q. El paralelismo dinámico no ha resultado beneficioso por la sobrecarga detectada en el lanzamiento

de *kernels* desde la GPU, que esperemos quede solventada en versiones futuras. Por su parte, Hyper-Q consigue aumentar el rendimiento hasta un 220 % respecto a la implementación base en Kepler del método directo, resultando mayor beneficio cuando la carga de trabajo procedente de un solo kernel no satura el número de procesadores disponibles en GPU. Esta ganancia puede mejorarse en códigos paralelos que usan hilos POSIX o MPI.

La versión recursiva de los momentos de Zernike también se mejora en la arquitectura Kepler aunque haya que sacrificar buena parte del paralelismo y Hyper-Q. Comparada con el método directo, sólo resulta peor hasta un orden seis, trasladando a partir de ahí mejoras que alcanzan hasta un 350 %.

En última instancia, el mejor método para el cálculo de los momentos de Zernike está supeditado a las necesidades de nuestra aplicación. En muchas de ellas, como la caracterización de imágenes y texturas, sólo se utilizan unos pocos momentos puntuales (aquellos que presentan mayor poder de discriminación tras un análisis previo). En otras aplicaciones, como las pertenecientes al campo de la compresión y reconstrucción de imágenes, es necesario computar, además del conjunto de repeticiones para un orden dado, un número elevado de órdenes en los que la formulación recursiva puede incluso desplegarse en una segunda dimensión. Este trabajo estudia la situación intermedia que considera el cálculo de todas las repeticiones de un mismo orden, y establece las bases para que en cualquiera de las numerosas aplicaciones de los momentos de Zernike, siempre podamos encontrar una implementación ventajosa en GPU respecto a CPU: Ya sea por la vía directa, gracias a la riqueza de los mecanismos de paralelización de CUDA, o por la vía recursiva, que a pesar de ser una formulación más competitiva en CPU, ha logrado factores de aceleración superiores en GPU. Y todo ello, favorecido por el tamaño de la imagen de entrada y el número de procesadores disponibles, dos factores que tienen mucho recorrido al alza tanto para las aplicaciones software como para las plataformas hardware venideras.

Parte IV

Conclusiones

7 Conclusiones y principales contribuciones

7.1. Conclusiones

A lo largo de esta tesis doctoral hemos analizado en detalle las oportunidades de optimización que han brindado las arquitecturas gráficas de la última década para acelerar diversos algoritmos de clasificación tisular. Esta faceta de la bioinformática ofrece multitud de aplicaciones reales, entre las que se encuentran la detección de tumores cancerígenos, la regeneración de lesiones óseas y la tipificación de la actividad neuronal.

Uno de los mayores retos de nuestro trabajo ha sido evolucionar al compás de la GPU, cuyo dinamismo ha sido vertiginoso incluso antes de la aparición de CUDA.

Así, los primeros capítulos de esta tesis reflejan una GPU monopolizada por los gráficos que no concebía de ninguna manera la computación de algoritmos de propósito general. Aquel programador atrevido que emplease GPGPU necesitaba un conocimiento exhaustivo del cauce de segmentación gráfico y grandes dotes de imaginación y destreza para transformar el algoritmo objetivo en un problema gráfico. Dicha transformación además requería una selección eficiente de las unidades funcionales gráficas empleadas en cada operación.

GPGPU tardó un tiempo en ser aceptada como línea de trabajo emergente en el campo de la computación de altas prestaciones, y se consolidaría definitivamente con la llegada de CUDA, aportando unas librerías y un entorno de programación más cercano a la programación convencional. Dicha tecnología de *shaders* unificados provocó un impulso cualitativo a nivel de rendimiento y cuantitativo a nivel de divulgación. A

partir de este momento, es más sencillo crear los algoritmos en GPU, aunque sigue requiriendo un amplio conocimiento de la arquitectura para explotar el máximo rendimiento como ha quedado de manifiesto a lo largo de esta tesis. Ahora, la utilización de GPGPU es una realidad expandida en la comunidad científica y en algunos segmentos profesionales, y esperamos que esta tendencia continúe en los próximos años para que se siga aprovechando nuestra contribución al impulso inicial.

En el primer acercamiento a una programación GPGPU clásica, los algoritmos biomédicos que elegimos para ilustrar nuestras técnicas han sido fundamentalmente dos: (1) la detección y recuento de patrones circulares para reconocimiento de células, y (2) la detección del tumor neuroblastoma. El primero hace un uso muy amplio y variado de las unidades funciones de la GPU, estudiando en detalle las posibilidades de cada una de ellas según el tipo de operación exigida desde el código. El factor de mejora en este caso oscila entre 25x y 358x dependiendo de los diferentes parámetros del problema. El segundo problema muestra el método más clásico de GPGPU donde toda la potencia de cálculo se centra en el procesador de fragmentos por ser la unidad funcional más potente y versátil. Los tiempos de ejecución obtenidos han sido hasta 45 veces más rápidos.

Una vez establecida la programación GPGPU contemporánea, el segundo de los algoritmos biomédicos anteriores se vuelve a diseñar y a programar para tener una comparativa real propia entre ambas posibilidades de programación de la GPU, y además se añade un algoritmo biomédico de registro para reconstruir imágenes de tejido 3D a través de las imágenes de las secciones tomadas de las muestras. En el primer caso se hace un análisis minucioso de cada característica de CUDA, extendiendo nuestro estudio desde un ordenador convencional hasta un clúster cooperativo de multiprocesadores para estudiar la escalabilidad. La GPGPU contemporánea sale victoriosa respecto a la GPGPU clásica y la escalabilidad entre nodos es prácticamente lineal hasta que el tiempo de lectura de las imágenes sobrecarga el sistema. El segundo problema demuestra que, además de las ventajas anteriores intrínsecas en CUDA, la inclusión de algunas librerías de funciones estándares como los algoritmos de resolución de FFT supone una facilidad extra en el nuevo entorno de programación. El factor de mejora oscila entre 2x y 4x, pudiendo extenderse a dos GPUs de forma lineal en la aceleración conforme aumenta la carga de trabajo.

En la parte final se han analizado las prestaciones más emergentes de la tecnología CUDA con la llegada de la generación Kepler. Para ello nos hemos centrado en el algoritmo del cálculo de los momentos de Zernike como descriptores de imágenes para su posterior clasificación. Mientras que el cambio de arquitectura ha mejorado el rendimiento hasta en un 250 %, las innovaciones generadas por la concurrencia Hyper-Q de los kernels en ejecución y la posibilidad de invocar a éstos desde la propia GPU

(paralelismo dinámico) no han colmado nuestras expectativas, transmitiendo como último mensaje que el modelo de máquina biprocesadora sigue estando plenamente vigente.

A partir de aquí, en la nueva era de la computación heterogénea que acabamos de inaugurar, algunos problemas requerirán más presencia de CPUs y otros de GPUs, y será cada caso concreto el que aconseje comprar un chip procesador con mayor presencia de *multi-cores* complejos o *many-cores* sencillos. Pero lo que queda claro en tiempos modernos es que un programador que no sepa aprovechar las prestaciones de la GPU se encuentra en clara desventaja para sacar partido desde la vertiente *software* al potencial que ofrece la capa *hardware* vigente en la actualidad. A menos que este proceso pueda completarse de forma semiautomática, y no parece que nos encontremos cerca de ese hito cuando la aplicación es mínimamente compleja, como ha sido el caso de los algoritmos estudiados a lo largo de nuestro trabajo.

7.2. Publicaciones relacionadas

La contribución científica de esta tesis tiene el respaldo y la validación de una larga lista de publicaciones internacionales, entre las que citamos, por orden de importancia, las siguientes:

- Seis revistas internacionales, cinco de ellas con índice de impacto JCR.
- Tres capítulos de libro en editoriales norteamericanas.
- Diez congresos, entre los que destacan tres de la IEEE de larga tradición en el ámbito de la bioinformática y uno de la ACM.

Este último congreso, uno de nuestros trabajos más pioneros publicado en la vigésimo segunda edición del International Conference on Supercomputing (ICS'08), es especial para nosotros por haber sido reconocido como uno de los 25 artículos más citados en la larga historia de esta conferencia, una de las grandes referencias en el área de la computación de altas prestaciones a la que nos sentimos más afín.

Pero todos y cada uno de nuestros trabajos ha aportado su granito de arena a la comunidad científica, y aunque nuestra línea de investigación es bastante homogénea y ha mostrado su continuidad en el tiempo, los artículos pueden acotarse en distintos vasos comunicantes que diferenciamos a continuación.

7.2.1. Cálculo de descriptores estadísticos representativos para la caracterización de tumores cancerígenos en imágenes biomédicas y su aceleración en GPU

Revistas

- A. Ruiz, O. Sertel, M. Ujaldón, U. Catalyurek, J. Saltz, M. Gurcan. Stroma Classification for Neuroblastoma on Graphics Processors. *Intl. Journal of Data Mining and Bioinformatics. Ed. Inderscience.* Volumen 3, número 3, páginas 280-298. 2009.

Congresos

- O. Sertel, A. Ruiz, U. Catalyurek, M. Ujaldón, J. Saltz, M. Gurcan. Computationally Efficient Pathologic Image Analysis: Use of GPUs for Classification of Stromal Density. *Archives of Pathology and Laboratory Medicine. 12th Anatomic Pathology Informatics and Imaging Support for Translational Medicine (APIII'07)*, 2007.
- A. Ruiz, O. Sertel, M. Ujaldón, U. Catalyurek, M. Gurcan, J. Saltz. Pathological Image Analysis Using the GPU: Stroma Classification for Neuroblastoma. *Proceedings IEEE Computer Society Press of IEEE Intl. Conference on BioInformatics and BioMedicine (BIBM'07), páginas 78-88*, 2007.
- M. Gurcan, O. Sertel, J. Kong, A. Ruiz, M. Ujaldón, U. Catalyurek, G. Lozanski, H. Shimada, J. Saltz. Computer-assisted Histopathology: Experience with Neuroblastoma and Follicular Lymphoma. *Proceedings Workshop on BioImage Informatics: Biological Imaging, Computer Vision and Data Mining.*, 2008.
- A. Ruiz, J. Kong, M. Ujaldón, K. Boyer, J. Saltz, M. Gurcan. Pathological Image Segmentation for Neuroblastoma Using the GPU. *En Proceedings of IEEE Intl. Symposium on Biomedical Imaging: From Nano to Macro.*, páginas 296-299, 2008.

Capítulos de libro

- U. Catalyurek, T. Hartley, O. Sertel, M. Ujaldón, A. Ruiz, J. Saltz, M. Gurcan. Processing of Large-Scale Biomedical Images on a Cluster of Multi-Core CPUs and GPUs. *Libro Advances in Parallel Computing. High Speed and Large Scale Scientific Computing. IOS Press (capítulo 18).* Páginas 341-364, 2009.

7.2.2. Clasificación automatizada de regiones de hueso y cartílago en imágenes biomédicas para la regeneración ósea y su aceleración en GPU

Congresos

- A. Ruiz, M. Ujaldón, J.A. Andrades, J. Becerra, K. Huang, T. Pan, J. Saltz. The GPU on Biomedical Image Processing for Color and Phenotype Analysis. *Proceedings IEEE Computer Society Press of 7th IEEE Intl. Symposium on BioInformatics & Bioengineering (BIBE'07)*, páginas 1124-1128, 2007.

7.2.3. Técnicas para la detección automática de formas circulares en imágenes celulares. Aceleración utilizando el hardware gráfico de rasterización

Revistas

- M. Ujaldón, A. Ruiz, N. Guil. On the Computation of the Circle Hough Transform by a GPU Rasterizer. *Pattern Recognition Letters.* Volumen 29, número 3, páginas 309-318, 2008.

- A. Ruiz, N. Guil, M. Ujaldón. Recognition of Circular Patterns on GPUs: Performance Analysis and Contributions. *Journal of Parallel and Distributed Computing. Special Issue on General-Purpose Parallel Processing Using GPUs.* Volumen 68, número 10, páginas 1329-1338, 2008.

Congresos

- A. Ruiz, M. Ujaldón. El Procesador Gráfico como Acelerador de la Transformada Hough. *In Proceedings of 2nd Intl. Conference on Science and Technology (ICST'07)*, 2007.

- A. Ruiz, M. Ujaldón, N. Guil. Using Graphics Hardware for Enhancing Edge and Circle Detection. *Lecture Notes in Computer Science of 3rd I. Conference on Pattern Recognition and Image Analysis (IbPRIA'07),* volumen 4478, páginas 234-241, 2007.

7.2.4. Reconstrucción 3D de imágenes biomédicas de alta resolución en plataformas de altas prestaciones: Supercomputadores y clústeres de procesadores *multi-core* y *many-core*

Revistas

- A. Ruiz, M. Ujaldón, L. Cooper, K. Huang. Non-rigid Registration for Large Set of Microscopic Images on Graphics Processors. *Journal of Signal Processing Systems for Signal, Image and Video Technology. Special Issue on Biomedical Imaging,* volumen 55, número 1, páginas 229-250, 2009.

Capítulos de libro

- L. Cooper, K. Huang, A. Ruiz, M. Ujaldón. Scalable Image Registration and 3D Reconstruction at Microscopic Resolution. *Capítulo del libro High Throughput Image Reconstruction and Analysis. ArTech House Publishers,* páginas 173-199, 2009.

7.2.5. Análisis de color y texturas sobre procesadores gráficos

Revistas

- F. Igual, R. Mayo, T. Hartley, U. Catalyurek, A. Ruiz, M. Ujaldón. Color and Texture Analysis on Emerging Parallel Architectures. *Journal High Performance Computing Applications,* volumen 25, número 4, páginas 404-427, 2011.

Congresos

- F. Igual, R. Mayo, T. Hartley, U. Catalyurek, A. Ruiz, M. Ujaldón. Optimizing Co-occurrence Matrices on Graphics Processors Using Sparse Representations. *Lecture Notes in Computer Science of 9th Intl. Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA'08),* 2008.

- T. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, M. Ujaldón. Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores. *Proceedings ACM Press of 22nd ACM International Conference on Supercomputing*, páginas 15-25, 2008.

- A. Ruiz, M. Ujaldón, F. Igual, R. Mayo. BIPGPU: Una Biblioteca para el Procesamiento de Imágenes Optimizada sobre Procesadores Gráficos. *Actas de las XIX Jornadas de Paralelismo,* 2008.

Capítulos de libro

- F. Igual, R. Mayo, U. Catalyurek, T. Hartley, A. Ruiz, M. Ujaldón. Exploring the GPU for Enhancing Parallelism on Color and Texture Analysis. *Libro Advances in Parallel Computing. Parallel Computing: From Multicores and GPUs to Petascale,* capítulo 19, páginas 299-306, 2010.

7.2.6. Análisis del rendimiento de la arquitectura Kepler mediante algoritmos dinámicos e irregulares

Revistas

- A. Ruiz, M. Ujaldón. Exploiting Kepler Capabilities on Zernike Moments. *Annals of Multicore and GPU Programming,* volumen 1, número 1, páginas 27-37, 2014.

7.3. Línea de Investigación Actual y Trabajo Futuro

Esta tesis forma parte de una línea de investigación basada en la aceleración de aplicaciones biomédicas en arquitecturas gráficas. En los comienzos, los algoritmos se desarrollaron bajo OpenGL/Cg y alcanzaron una discreta aceleración respecto a CPU entre 1.45x y 3.43x.

Con la llegada de CUDA y OpenCL, las GPUs aumentaron su posibilidades para procesar algoritmos de propósito general, atrayendo todo el interés de las nuevas aplicaciones biomédicas. En este sentido, se hace un estudio del potencial que ofrecen las GPUs en un conjunto de nodos de supercomputación, donde cada nodo consta de varias CPUs y GPUs. Los resultados ya mostraban cómo la tendencia hacia el uso de GPUs para problemas de esta naturaleza era tan evidente como natural [31].

La mejora de rendimiento analizada en esta tesis tras el empleo de arquitecturas gráficas es sólo el comienzo de un largo camino impulsado por la escalabilidad de la GPU para convertirse en un procesador de referencia para la clasificación de imágenes y su procesamiento en general. En este sentido, este estudio forma parte de un proyecto más ambicioso consistente en la creación de una librería biomédica acelerada

usando GPUs como co-procesadores.

La nueva arquitectura Maxwell [56] y la llegada de alternativas como Xeon Phi [37] genera una vertiente de interés nueva para el procesamiento paralelo muy acorde a lo hasta ahora evaluado para las GPUs, pues ofrecen un modelo de programación similar bajo la iniciativa OpenCL. Las aplicaciones biomédicas se evaluarán en OpenCL en comparación a CUDA, aunque las características más innovadoras no suelen estar implementadas o lo hacen con cierto retraso.

Bibliografía

[1] AL-RAWI, M. Fast zernike moments. *Journal of Real-Time Image Processing 3*, 1 (2008), 89–96.

[2] ANDO, S. Consistent gradient operators. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 22*, 3 (2000), 252–265.

[3] ATIQUZZAMAN, M. Pipelined implementation of the multiresolution Hough transform in a pyramid multiprocessor. *Pattern Recognition Letters 15*, 9 (1994), 841–851.

[4] ATIQUZZAMAN, M. Coarse-to-fine search technique to detect circles in images. *The International Journal of Advanced Manufacturing Technology 15*, 2 (1999), 96–102.

[5] BALLARD, D. Generalizing the Hough transform to detect arbitrary shapes. *Pattern recognition 13*, 2 (1981), 111–122.

[6] BEYNON, M., KURC, T., CATALYUREK, U., CHANG, C., SUSSMAN, A., AND SALTZ, J. Distributed processing of very large datasets with DataCutter. *Parallel Computing 27*, 11 (2001), 1457–1478.

[7] BIN, Y., AND JIA-XIONG, P. Invariance analysis of improved Zernike moments. *Journal of Optics A: Pure and Applied Optics 4*, 6 (2002), 606.

[8] BITTNER, J., WIMMER, M., PIRINGER, H., AND PURGATHOFER, W. Coherent hierarchical culling: Hardware occlusion queries made useful. In *Computer Graphics Forum* (2004), vol. 23, Wiley Online Library, pp. 615–624.

[9] BOTNEN, M., AND UELAND, H. The GPU as a computational resource in medical image processing. *Dept. of Computer and Information Science, Norwegian Univ. of Science and Technology, Tech. Rep* (2004).

[10] BRANDT, R., ROHLFING, T., RYBAK, J., KROFCZIK, S., MAYE, A., WESTERHOFF, M., HEGE, H., AND MENZEL, R. Three-dimensional average-shape atlas of the honeybee brain and its applications. *The Journal of comparative neurology 492*, 1 (2005), 1–19.

[11] BRAUMANN, U., KUSKA, J., EINENKEL, J., HORN, L., LOFFLER, M., AND HOCKEL, M. Three-dimensional reconstruction and quantification of cervical carcinoma invasion fronts from histological serial sections. *Medical Imaging, IEEE Transactions on 24*, 10 (2005), 1286–1307.

[12] BRUGUERA, J., GUIL, N., LANG, T., VILLALBA, J., AND ZAPATA, E. Cordic based parallel/pipelined architecture for the Hough transform. *The Journal of VLSI Signal Processing 12*, 3 (1996), 207–221.

[13] CAMBAZOGLU, B., SERTEL, O., KONG, J., SALTZ, J., GURCAN, M., AND CATALYUREK, U. Efficient processing of pathological images using the grid: Computer-aided prognosis of neuroblastoma. In *Proceedings of the 5th IEEE workshop on Challenges of large applications in distributed environments* (2007), ACM, pp. 35–41.

[14] CANNY, J. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 6 (1986), 679–698.

[15] CAPOWSKI, J. Computer-aided reconstruction of neuron trees from several serial sections. *Computers and Biomedical Research 10*, 6 (1977), 617–629.

[16] CHONG, C.-W., RAVEENDRAN, P., AND MUKUNDAN, R. A comparative analysis of algorithms for fast computation of Zernike moments. *Pattern Recognition 36*, 3 (2003), 731–742.

[17] COOPER, L., HUANG, K., SHARMA, A., MOSALIGANTI, K., AND PAN, T. Registration vs. reconstruction: Building 3-d models from 2-d microscopy images. In *Proceedings of the workshop on multiscale biological imaging, data mining and informatics* (2006), pp. 57–58.

[18] CRUM, W., HARTKENS, T., AND HILL, D. Non-rigid image registration: theory and practice. *British journal of radiology 77*, suppl 2 (2004), S140–S153.

[19] CUDA, N. Parallel Programming and Computing Platform. `http://www.nvidia.es/object/cuda_home_new_es.html`, June 2013.

[20] DAVIES, E. The effect of noise on edge orientation computations. *Pattern recognition letters 6*, 5 (1987), 315–322.

[21] DAVIES, E. A modified Hough scheme for general circle location. *Pattern Recognition Letters 7*, 1 (1988), 37–43.

[22] FUNG, J., AND MANN, S. OpenVIDIA: parallel GPU computer vision. In *Proceedings of the 13th annual ACM international conference on Multimedia* (2005), ACM, pp. 849–852.

[23] GOSHTASBY, A. *2-D and 3-D Image Registration: for Medical, Remote Sensing, and Industrial Applications*. Wiley-Interscience, 2005.

[24] GRIMSON, W., AND HUTTENLOCHER, D. On the sensitivity of the Hough transform for object recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 12*, 3 (1990), 255–274.

[25] GUHA, S., KRISNAN, S., AND VENKATASUBRAMANIAN, S. Data visualization and mining using the gpu. *Tutorial at ACM SIGKDD 5* (2005).

[26] GUIL, N., AND ZAPATA, E. A parallel pipelined hough transform. In *Euro-Par'96 Parallel Processing* (1996), Springer, pp. 131–138.

[27] GURCAN, M., KONG, J., SERTEL, O., CAMBAZOGLU, B., SALTZ, J., AND CATALYUREK, U. Computerized pathological image analysis for neuroblastoma prognosis. In *AMIA Annual Symposium Proceedings* (2007), vol. 2007, American Medical Informatics Association, p. 304.

[28] HADWIGER, M., LANGER, C., SCHARSACH, H., AND BUHLER, K. State of the art report on GPU-based segmentation. *VRVis Research Center, Vienna, Austria, Tech. Rep. TR-VRVIS-2004-17* (2004).

[29] HAJNAL, J., HILL, D., AND HAWKES, D. *Medical image registration*. CRC, 2001.

[30] HARTLEY, T., CATALYUREK, U., RUIZ, A., IGUAL, F., MAYO, R., AND UJALDÓN, M. Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores. In *Proceedings ACM Press* (Isla de Kos (Grecia), Junio 2008), 22nd ACM International Conference on Supercomputing, pp. 15–25.

[31] HARTLEY, T. D., CATALYUREK, U., RUIZ, A., IGUAL, F., MAYO, R., AND UJALDON, M. Biomedical image analysis on a cooperative cluster of GPUs and multicores. In *Proceedings of the 22nd annual international conference on Supercomputing* (2008), ACM, pp. 15–25.

[32] HILL, B., AND BALDOCK, R. A. The constrained distance transform: interactive atlas registration with large deformations through constrained distances. In

Proceedings of DEFORM'06: Workshop on Image Registration in Deformable Environments, Edinburgh (2006), Citeseer, pp. 51–60.

[33] HOME PAGE, F. The FFTW Library. `http://www.fftw.org/`, June 2014.

[34] HUANG, K., COOPER, L., SHARMA, A., AND PAN, T. Fast automatic registration algorithm for large microscopy images. In *Life Science Systems and Applications Workshop, 2006. IEEE/NLM* (2006), IEEE, pp. 1–2.

[35] HUIJSMANS, D., LAMERS, W., LOS, J., AND STRACKEE, J. Toward computerized morphometric facilities: A review of 58 software packages for computer-aided three-dimensional reconstruction, quantification, and picture generation from parallel serial sections. *The Anatomical Record 216*, 4 (2005), 449–470.

[36] HWANG, S.-K., AND KIM, W.-Y. A novel approach to the fast computation of Zernike moments. *Pattern Recognition 39*, 11 (2006), 2065–2076.

[37] INTEL. The Intelő Xeon PhiŹ Coprocessor: Parallel Processing, Unparalleled Discovery. `http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html`, June 2013.

[38] IOANNOU, D., HUDA, W., AND LAINE, A. Circle recognition through a 2D Hough transform and radius histogramming. *Image and vision computing 17*, 1 (1999), 15–26.

[39] JAFARI-KHOUZANI, K., AND SOLTANIAN-ZADEH, H. Multiwavelet grading of pathological images of prostate. *Biomedical Engineering, IEEE Transactions on 50*, 6 (2003), 697–704.

[40] JAIN, A., DUIN, R., AND MAO, J. Statistical pattern recognition: A review. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 22*, 1 (2000), 4–37.

[41] JENETT, A., SCHINDELIN, J., AND HEISENBERG, M. The Virtual Insect Brain protocol: creating and comparing standardized neuroanatomy. *BMC bioinformatics 7*, 1 (2006), 544.

[42] JOHNSON, E., AND CAPOWSKI, J. A system for the three-dimensional reconstruction of biological structures. *Computers and Biomedical Research 16*, 1 (1983), 79–87.

[43] KAVIANPOUR, A., SHOARI, S., AND BAGHERZADEH, N. A new approach for circle detection on multiprocessors. *Journal of Parallel and Distributed Computing 20*, 2 (1994), 256–260.

[44] KONG, J., SHIMADA, H., BOYER, K., SALTZ, J., AND GURCAN, M. Image analysis for automated assessment of grade of neuroblastic differentiation. In *Biomedical Imaging: From Nano to Macro, 2007. ISBI 2007. 4th IEEE International Symposium on* (2007), IEEE, pp. 61–64.

[45] KUMAR, S., RANGANATHAN, N., AND GOLDGOF, D. Parallel algorithms for circle detection in images. *Pattern Recognition 27*, 8 (1994), 1019–1028.

[46] LEVINTHAL, C., AND WARE, R. Three dimensional reconstruction from serial sections.

[47] LEWIS, J. Fast normalized cross-correlation. In *Vision interface* (1995), vol. 10, pp. 120–123.

[48] MARTÍN-REQUENA, M. J., AND UJALDÓN, M. Leveraging Graphics Hardware for an Automatic Classification of Bone Tissue. In *Computational Vision and Medical Image Processing*. Springer, 2011, pp. 209–228.

[49] MARTZ, P. *OpenGL distilled*. Addison-Wesley Professional, 2006.

[50] MOSALIGANTI, K., PAN, T., SHARP, R., RIDGWAY, R., IYENGAR, S., GULACY, A., WENZEL, P., DE BRUIN, A., MACHIRAJU, R., HUANG, K., ET AL. Registration and 3d visualization of large microscopy images. In *Proc. of SPIE Vol* (2006), vol. 6144, pp. 61442V–1.

[51] MOSS, V. The computation of 3-dimensional morphology from serial sections. *European Journal of Cell Biology 48* (1989), 61–64.

[52] MUKUNDAN, R., AND RAMAKRISHNAN, K. Fast computation of Legendre and Zernike moments. *Pattern recognition 28*, 9 (1995), 1433–1442.

[53] NASSER ESGIAR, A., NAGUIB, R., SHARIF, B., BENNETT, M., AND MURRAY, A. Microscopic image analysis for quantitative measurement and feature identification of normal and cancerous colonic mucosa. *Information Technology in Biomedicine, IEEE Transactions on 2*, 3 (1998), 197–203.

[54] NVIDIA. The Kepler architecture. `http://www.nvidia.com/object/nvidia-kepler.html`, June 2013.

[55] NVIDIA. CUFFT Library. `http://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf`, June 2014.

[56] NVIDIA. The Maxwell architecture. `https://developer.nvidia.com/maxwell-compute-architecture`, Apr. 2015.

[57] OJALA, T., PIETIKAINEN, M., AND MAENPAA, T. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 24*, 7 (2002), 971–987.

[58] OWENS, J., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A., AND PURCELL, T. A Survey of General-Purpose Computation on Graphics Hardware. In *Computer graphics forum* (2007), vol. 26, Wiley Online Library, pp. 80–113.

[59] PAN, Y., LI, K., AND HAMDI, M. An improved constant-time algorithm for computing the Radon and Hough transforms on a reconfigurable mesh. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on 29*, 4 (1999), 417–421.

[60] PASCHOS, G. Perceptually uniform color spaces for color texture analysis: an empirical evaluation. *Image Processing, IEEE Transactions on 10*, 6 (2001), 932–937.

[61] PAVEL, S., AND AKL, S. Efficient algorithms for the Hough transform on arrays with reconfigurable optical buses. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International* (1996), IEEE, pp. 697–701.

[62] PETUSHI, S., GARCIA, F., HABER, M., KATSINIS, C., AND TOZEREN, A. Large-scale computations on histology images reveal grade-differentiating parameters for breast cancer. *BMC Medical Imaging 6*, 1 (2006), 14.

[63] PETUSHI, S., KATSINIS, C., COWARD, C., GARCIA, F., AND TOZEREN, A. Automated identification of microstructures on histology slides. In *Biomedical Imaging: Nano to Macro, 2004. IEEE International Symposium on* (2004), IEEE, pp. 424–427.

[64] PODLOZHNYUK, V. Histogram calculation in CUDA. *NVIDIA Corporation, White Paper* (2007).

[65] PRATT, W., ET AL. Digital imaging processing, 1978.

[66] PRESCOTT, J., CLARY, M., WIET, G., PAN, T., AND HUANG, K. Automatic registration of large set of microscopic images using high-level features. In *Biomedical Imaging: Nano to Macro, 2006. 3rd IEEE International Symposium on* (2006), IEEE, pp. 1284–1287.

[67] RAY, S. On a theoretical property of the bhattacharyya coefficient as a feature evaluation criterion. *Pattern recognition letters 9*, 5 (1989), 315–319.

[68] SARMA, S., KERWIN, J., PUELLES, L., SCOTT, M., STRACHAN, T., FENG, G., SHARPE, J., DAVIDSON, D., BALDOCK, R., AND LINDSAY, S. 3D modelling, gene expression mapping and post-mapping image analysis in the developing human brain. *Brain research bulletin 66*, 4 (2005), 449–453.

[69] SCHMITT, O., MODERSITZKI, J., HELDMANN, S., WIRTZ, S., AND FISCHER, B. Image registration of sectioned brains. *International Journal of Computer Vision 73*, 1 (2007), 5–39.

[70] SERTEL, O., KONG, J., SHIMADA, H., CATALYUREK, U., SALTZ, J., AND GURCAN, M. Computer-aided prognosis of neuroblastoma on whole-slide images: Classification of stromal development. *Pattern Recognition 42*, 6 (2009), 1093–1103.

[71] SHARP, R., RIDGWAY, R., MOSALIGANTI, K., WENZEL, P., PAN, T., DE BRUIN, A., MACHIRAJU, R., HUANG, K., LEONE, G., AND SALTZ, J. Volume rendering phenotype differences in mouse placenta microscopy data. *Computing in science & engineering 9*, 1 (2007), 38–47.

[72] STREICHER, J., DONAT, M., STRAUSS, B., SPORLE, R., SCHUGHART, K., AND MULLER, G. Computer-based three-dimensional visualization of developmental gene expression. *nature genetics 25*, 2 (2000), 147–152.

[73] TAHIR, M., AND BOURIDANE, A. Novel round-robin tabu search algorithm for prostate cancer classification and diagnosis using multispectral imagery. *Information Technology in Biomedicine, IEEE Transactions on 10*, 4 (2006), 782–793.

[74] TAKALA, V., AHONEN, T., AND PIETIKÄINEN, M. Block-based methods for image retrieval using local binary patterns. *Image Analysis* (2005), 13–181.

[75] TEAGUE, M. R. Image analysis via the general theory of moments. *J. Opt. Soc. Am 70*, 8 (1980), 920–930.

[76] TEH, C.-H., AND CHIN, R. T. On image analysis by the methods of moments. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 10*, 4 (1988), 496–513.

[77] TUCERYAN, M., AND JAIN, A. Texture analysis. *Handbook of pattern recognition and computer vision 276* (1993).

[78] UNDERHILL, A., ATIQUZZAMAN, M., AND OPHEL, J. Performance of the Hough transform on a distributed memory multiprocessor. *Microprocessors and Microsystems 22*, 7 (1999), 355–362.

[79] WENZEL, P., WU, L., DE BRUIN, A., CHONG, J., CHEN, W., DURESKA, G., SITES, E., PAN, T., SHARMA, A., HUANG, K., ET AL. Rb is critical in a mammalian tissue stem cell population. *Genes & development 21*, 1 (2007), 85–97.

[80] WU, W., AND HENG, P. A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting. *Computer Animation and Virtual Worlds 15*, 3-4 (2004), 219–227.

[81] YOO, T. *Insight into images: principles and practice for segmentation, registration, and image analysis*, vol. 203. AK Peters Ltd, 2004.

[82] ZHAO, G., AND PIETIKAINEN, M. Dynamic texture recognition using local binary patterns with an application to facial expressions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 29*, 6 (2007), 915–928.

[83] ZHAO, Y., HAN, Y., FAN, Z., QIU, F., KUO, Y., KAUFMAN, A., AND MUELLER, K. Visual simulation of heat shimmering and mirage. *Visualization and Computer Graphics, IEEE Transactions on 13*, 1 (2007), 179–189.