UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Departamento de Electrónica e Computación

Tesis doctoral

Técnicas para la segmentación y visualización eficiente de imagen médica 3D: explotando la arquitectura de la GPU

Presentada por:

Julián Lamas-Rodríguez

Dirigida por:

Prof. Dr. Francisco Argüello

Prof. Dra. Dora Blanco Heras

Junio de 2014

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Francisco Argüello, Profesor Titular de Universidad del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

Dora B. Heras, Profesora Titular de Universidad del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

HACEN CONSTAR:

Que la memoria titulada **Técnicas para la segmentación y visualización eficiente de imagen médica 3D: explotando la arquitectura de la GPU** ha sido realizada por **D. Julián Lamas-Rodríguez** bajo nuestra dirección en el Departamento de Electrónica e Computación de la Universidad de Santiago de Compostela, y constituye la Tesis que presenta para obtar al título de Doctor.

Santiago de Compostela, junio de 2014

Francisco Argüello
Codirector de la tesis

Dora Blanco Heras
Codirectora de la tesis

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

A mis padres y mis hermanos

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Agradecimientos

Resulta difícil encontrar las palabras adecuadas para expresar todo mi agradecimiento a las personas sin las cuales esta Tesis de Doctorado no hubiera sido posible, pero aquí va un pequeño intento.

En primer lugar, quiero agradecer a mis directores de tesis, el Prof. Dr. Francisco Argüello y la Prof. Dra. Dora B. Heras, por todo su esfuerzo y dedicación durante todos estos años, en los momentos buenos y en los malos. Ellos han sido mi guía durante mis primeros pasos (y los inevitables tropiezos) en el mundo de la investigación. Han compartido mis éxitos y mis fracasos, y siempre he podido contar con su apoyo. También quiero agradecer a la Dra. Montserrat Bóo, que participó en la dirección al principio del desarrollo de esta tesis, y cuya labor sin duda fue decisiva para establecer los contactos necesarios que me permitieron realizar varias estancias de investigación en el extranjero.

Agradezco al Grupo de Arquitectura de Computadores, al Departamento de Electrónica y Computación y al Centro Singular de Investigación en Tecnoloxías da Información (CITIUS) por haberme acojido en un ambiente agradable e inspirador. A punto de cumplir su segundo año de vida a fecha de escribir esta tesis, quisiera desearle al CITIUS una carrera llena de merecidos éxitos.

Mi agradecimiento al Dr.-Ing. Oliver Burgert, a Daniella Wellein y a Silvia Born, del Innovation Center Computer Assisted Surgery (ICCAS), de la Universidad de Leipzig, por haberme recibido durante mi primera estancia en el extranjero y por su colaboración y ayuda durante mi trabajo con la herramienta de visualización VolV. También agradezco al Prof. Dr. Dr. h. c. Peter Deuflhard por su invitación al Zuse-Insitut Berlin (ZIB), y al Dr.-Ing. Stefan Zachow y los miembros del Grupo de Planificación Médica del Departamento de Visualización Médica del ZIB por haberme acogido como si fuese uno más, ayudándome en el desarrollo y la integración en Amira de las herramientas de segmentación presentadas en esta tesis.

La financiación de esta investigación ha sido posible gracias al proyecto 08TIC001206PR de la Xunta de Galicia, el proyecto TIN 2010-17545 del Ministerio de Ciencia e Innovación y las ayudas para la consolidación y estructuración de unidades de investigación competitivas 2009/022, 2009/074, 2010/28, 2012/151 y 2012/323 de la Xunta de Galicia. Agradezco al Ministerio de Ciencia e Innovación la financiación de mi beca-contrato FPI-MICINN. Aprovecho para resaltar la importancia de financiar públicamente la investigación. Me resulta descorazonador ver cómo, amparándose en la manida excusa de la crisis, se están cerrando numerosas vías de investigación por falta de financiación, en muchos casos echando por tierra años de trabajo y truncando de raíz carreras profesionales, lo que resulta en un desastre tanto personal como social. Desde aquí insto a nuestros políticos y gobernantes a asumir la responsabilidad de su cargo y fomentar la ciencia y la investigación como motor económico del país.

Quiero mandar un agradecimiento muy especial a todos mis compañeros y amigos con los que tuve el inmenso honor de conocer, y que me aceptaron y me hicieron compañía durante este viaje. Algunos, por azares de la vida, han tenido que marcharse lejos, pero de todos ellos guardo un gran recuerdo. También quiero agradecer a mis amigos de Coruña, quienes, a pesar de no poder verlos más que contados fines de semana, jamás se olvidaron de mí.

Por último, mi agradecimiento más especial a mi familia. Sin el apoyo, la paciencia y la inspiración de mis padres y mis hermanos, no hubiese llegado hasta el final. Ellos creyeron en mí, y seguirán creyendo. Gracias de todo corazón.

Berlín, junio de 2014

Índice general

Prefacio

El 8 de noviembre de 1895 Wilhelm Conrad Röntgen detectó radiación electromagnética en el rango de longitud de onda hoy conocido como rayos X o rayos Röntgen, descubrimiento que le sirvió para obtener el Premio Nobel de Física en 1901. Desde entonces, la capacidad de utilizar los rayos X para visualizar el interior de objetos opacos ha dado lugar a multitud de aplicaciones, entre las que se encuentra, de forma notable, la radiografía médica. Hoy en día, la obtención y el procesamiento de imagen médica, ya sea mediante rayos X o cualquier otra modalidad, se ha convertido en una parte casi imprescindible del diagnóstico médico, el estudio y tratamiento de enfermedades, y la planificación quirúrgica [58].

Las principales tareas asociadas al procesado de imagen médica se pueden clasificar en cuatro categorías: filtrado, registro, segmentación y visualización. Entre las operaciones de filtrado son comunes los procesos para aumentar el ratio de la relación señal-ruido de la imagen y para resaltar las estructuras de interés. Por registro de la imagen se entiende un amplio grupo de operaciones que se utilizan para combinar imágenes del mismo sujeto tomadas desde diferentes posiciones o usando diferentes modalidades, alinear secuencias temporales de imágenes para compensar el movimiento del sujeto, realizar intervenciones guiadas por imagen, y alinear imágenes de múltiples sujetos en los estudios de seguimiento de enfermedades [81]. La segmentación es el proceso de particionar la imagen en el conjunto de regiones que la componen, con el objetivo de que las regiones representen áreas de interés [158]. En el contexto de la imagen médica, la segmentación se utiliza habitualmente para aislar e identificar órganos, vasos sanguíneos, tumores y huesos. Por último, la visualización médica trata fundamentalmente con los procesos necesarios para representar, explorar y analizar los conjuntos de datos adquiridos mediante las diferentes modalidades disponibles [141].

En lo que respecta a la adquisición de imágenes médicas, existe una amplia variedad de modalidades y técnicas. Como ya hemos avanzado, los rayos X permitieron obtener las

primeras imágenes que ofrecían información del interior del cuerpo humano sin necesidad de practicar ninguna disección o cirugía, y constituyen todavía hoy una de las modalidades más comúnmente utilizadas en el campo de la salud, con reconocidas variantes como las mamografías, las angiografías y las fluoroscopias. La introducción en 1968 de la tomografía computacional (en inglés CT, *computational tomography*) por rayos X [83] permitió construir representaciones volumétricas de los objetos observados, pues hasta entonces las imágenes de rayos X estaban limitadas a ser únicamente proyecciones 2D. En las imágenes CT se pueden localizar de forma bastante precisa órganos y tejidos no solo en altura y anchura sino también en profundidad. Además, la tomografía computacional es dos órdenes de magnitud más sensible que las imágenes de rayos X, lo que permite identificar tejidos blandos, como el hígado y el páncreas, que no son distinguibles en las imágenes de rayos X convencionales.

La segunda modalidad de tomografía más extendida es la imagen de resonancia magnética (en inglés MRI, *magnetic resonance imaging*) [38]. La modalidad MRI permite reconstruir imágenes volumétricas a partir de la detección de la señal de radio-frecuencia emitida por los núcleos de los átomos de hidrógeno (presentes en cualquier tejido que contenga moléculas de agua) tras ser excitados dentro de un campo magnético oscilatorio. A diferencia de la tomografía computacional por rayos X, la resonancia magnética no produce radiación ionizante, por lo que el riesgo para la salud del sujeto en observación es mucho menor. Las imágenes MRI proporcionan un contraste mucho más alto en los tejidos blandos, por lo que se utilizan con frecuencia en los campos de neuroimagen (para discriminar las materias blanca y gris del cerebro) y para el diagnóstico de articulaciones, como la rodilla o el hombro. Sin embargo, el hecho de que una misma estructura presente variaciones de contraste superiores al 10 % dentro de la misma imagen representa un problema para el análisis asistido por ordenador, aunque existen soluciones para tratar de corregir este tipo de heterogeneidades de forma algorítmica [141].

Otras modalidades para la adquisición de imágenes médicas se basan en el uso de ultrasonidos, esto es, ondas de sonido de alta frecuencia que penetran los tejidos y reflejan los cambios entre tejidos adyacentes, y en la emisión de positrones (*positron emission tomography* o PET) o de rayos gamma (*single-photon emission computed tomography* o SPECT), que se utilizan para seguir la traza de una sustancia radiofarmacológica introducida dentro del cuerpo.

De las tareas asociadas al procesado de imagen médica, el trabajo de esta tesis se centra en la realización en GPU en tiempo real de las operaciones necesarias para la segmentación y

visualización 3D de conjuntos de datos volumétricos. Como ya hemos señalado, las modalidades de tomografía computacional por rayos X y por resonancia magnética generan imágenes en 3D (esto es, volúmenes), por lo que la mayor parte de los datos usados en este trabajo pertenecen a una de estas dos modalidades.

Las imágenes médicas tridimensionales se suelen almacenar y representar como una pila de imágenes. Cada imagen representa una capa o lámina de la parte del sujeto escaneada y está compuesta de píxeles (del inglés *picture elements*) individuales. Estos píxeles se organizan en una malla bidimensional, donde la distancia entre dos píxeles suele ser constante en cada dirección. Los datos volumétricos combinan cada una de las imágenes bidimensionales en una malla 3D, y cada uno de los datos o elementos de la malla pasa a denominarse vóxel (del inglés *volume element*).

El procesado de imágenes médicas implica la realización de tareas computacionalmente costosas, no solo por su complejidad, sino también por el gran número de elementos a procesar, especialmente en el caso de imágenes volumétricas. Es deseable que los resultados de estas operaciones sean obtenidos de la forma más veloz posible para obtener diagnósticos rápidos que beneficien tanto a médicos como a pacientes, siempre manteniendo los requisitos de robustez, precisión y reproducibilidad [141]. El crecimiento continuo en capacidad computacional de los ordenadores facilita que esto sea posible, pero incluso así todavía queda el reto adicional de proporcionar plataformas de computación que ofrezcan la mayor eficiencia posible al menor coste.

Los chips de procesamiento gráfico (en inglés GPU, *graphics processing unit*) incluidos en las tarjetas gráficas de consumo son hoy en día procesadores paralelos masivos capaces de ejecutar miles de hilos simultáneamente. No se puede negar que la evolución de las GPU ha venido impulsada por la demanda de gráficos cada vez más realistas dentro de la industria de los videojuegos. Dado que el proceso de renderizado se realiza de forma similar para cada uno de los píxeles que componen la imagen final, las GPU han adaptado su diseño al paradigma de computación paralela basado en ejecutar instrucciones que operen de forma simultánea sobre múltiples datos (en inglés SIMD, *single instruction multiple data*).

El rendimiento de los algoritmos en GPU depende de cómo pueden adaptarse al paradigma SIMD. Muchos de los algoritmos de procesado de imágenes médicas son paralelos intrínsecamente, ya que consisten en realizar las mismas tareas de manera más o menos independiente sobre distintos píxeles. Además, en los últimos años, la evolución de los modelos de programación en GPU y de las herramientas disponibles ha facilitado la implementación de este tipo

de algoritmos en tarjetas gráficas. En este sentido, la aparición de CUDA (en inglés, *compute unified device architecture*) para las tarjetas gráficas de NVIDIA y Brook+ para las tarjetas gráficas de ATI/AMD supuso un claro punto de inflexión en el campo de la computación de propósito general en GPU [126, 133]. Así, aunque desde principios de la década del 2000 se observa un crecimiento constante del número de publicaciones relacionadas con el procesado de imágenes médicas en GPU, es en 2008 cuando este crecimiento experimenta un drástico empuje [61]. La popularidad del uso de diversos aceleradores *hardware* ha propiciado también el desarrollo de diferentes estándares para programación de propósito general multiplataforma, como OpenCL, que, al soportar lenguajes como C y C++, ofrecen también una facilidad de programación similar a la de CUDA. Sin embargo, los resultados muestran que, aunque su portabilidad entre plataformas es plausible, al utilizar estos estándares el rendimiento pico que podría alcanzarse con CUDA no está garantizado.

El diseño de aplicaciones que aspiren a obtener el máximo rendimiento de la GPU debe tener en cuenta diversos factores: cómo dividir las operaciones entre los hilos de ejecución de la GPU, cómo almacenar los datos en los diferentes espacios en los que se divide su jerarquía de memoria, cuándo y cómo sincronizar los diferentes hilos, etc. Aunque gracias a las herramientas disponibles es posible implementar en la GPU en poco tiempo un importante número de algoritmos, maximizar su eficiencia exige al programador conocer en profundidad la arquitectura. Desde la aparición de las arquitecturas paralelas se han ido recopilando una serie de "recetas", conocidas como patrones de diseño paralelo, con el objetivo de explotar este tipo de arquitecturas [117]. Sin embargo, en el caso de la GPU no basta solo con aplicar los patrones ya conocidos, sino que también es necesario plantear y utilizar nuevos patrones y técnicas que permitan sacar el máximo partido de las características particulares del *hardware* gráfico.

Resumen de la Tesis de Doctorado

Objetivo y ámbito de estudio

El objetivo del trabajo realizado en esta Tesis de Doctorado es proponer soluciones eficientes para el procesado de imagen médica en tarjetas gráficas (GPU). En particular, el trabajo se ha centrado en las tareas de segmentación y visualización. Ambas tareas son bastante amplias, y en la literatura es posible encontrar multitud de soluciones diferentes para cada una de ellas. Es por esto que hemos seleccionado una serie de algoritmos cuya efectividad ya está demostrada y hemos aplicado diversas de técnicas para implementarlos en GPU tratando de maximizar el rendimiento.

En lo que respecta a la segmentación, dada su relevancia en el campo de visión por computador, existe una amplia gama de métodos con gran variedad de posibles aplicaciones [190, 193]. En el campo de la imagen médica la segmentación es esencial para la localización de tumores, la medición del volumen de los tejidos y el estudio de estructuras anatómicas. La segmentación es también importante para la visualización, pues permite mostrar de forma selectiva solo ciertos objetos contenidos dentro de un volumen más amplio. Hay numerosas técnicas que permiten llevar a cabo esta tarea y, para esta tesis, hemos escogido la técnica de crecimiento de regiones basada en el conjunto de nivel.

Los métodos del conjunto de nivel presentan una serie de características adecuadas para su implementación en GPU; sin embargo, para extraer el máximo rendimiento una implementación directa no es suficiente. Hemos utilizado múltiples estrategias para adaptar estos algoritmos de segmentación y maximizar el rendimiento de la GPU, que han supuesto importantes modificaciones sobre los algoritmos originales. Así, hemos dividido el esquema de ejecución en bloques, de forma que restringimos la computación a solo determinadas regiones de interés durante cada etapa de ejecución (lo que ha supuesto también establecer métodos

para detectar dichas regiones de interés). Aunque esto ha implicado añadir más complejidad a los algoritmos originales, los valores de aceleración conseguidos avalan esta metodología.

La visualización médica es otra tarea con un amplio número de soluciones en la literatura [141, 188]. En el mercado pueden encontrarse aplicaciones orientadas a ofrecer diversos esquemas de visualización de imágenes 2D y 3D, ya sea enfocadas hacia la investigación o hacia el campo clínico [1, 9, 20, 23, 25, 138]. Dado que las GPU nacieron desde un principio como procesadores gráficos, se han convertido en la plataforma estándar *de facto* para visualizar volúmenes. En esta tesis revisamos algunas de las herramientas disponibles para esta tarea, con especial énfasis en Amira [1, 165] y VolV [138, 141], con las que hemos trabajado directamente y en las que hemos introducido algunos de los métodos desarrollados.

Un problema interesante es, dada la tendencia actual a trabajar con conjuntos de datos cada vez más grandes y más precisos, cómo procesar y visualizar dicha información con las restricciones de memoria en las tarjetas gráficas actuales, especialmente en el caso en que un especialista disponga solo de un sencillo equipo basado en un PC de sobremesa que incorpore una GPU. Es por ello que hemos complementado el estudio de la visualización de volúmenes con la compresión de datos en tarjetas gráficas de consumo de NVIDIA programadas en CUDA. Demostramos cómo se puede utilizar Amira para implementar nuestras soluciones de segmentación, y cómo se puede aumentar el *pipeline* de visualización de VolV para dar soporte a la visualización de volúmenes comprimidos en GPU manteniendo una tasa de refresco estable e interactiva sin penalizar la calidad visual.

Un esquema aplicado comúnmente para resolver problemas en paralelo consiste en dividir el conjunto de operaciones (y, en ocasiones, también el conjunto de datos sobre el que se opera) en bloques independientes. Este esquema obedece al patrón de descomposición geométrica y sus variantes, como el conocido “divide y vencerás” [117], y resulta especialmente adecuado para su ejecución en GPU. Sin embargo, algunos algoritmos ejecutan sus operaciones en pasos sucesivos que presentan dependencias que hacen que la división en bloques no sea evidente. Para solucionar este problema, presentamos el patrón de diseño paralelo denominado “divide y fusiona” [30, 89]. Hemos aplicado con éxito este patrón en la resolución de sistemas de ecuaciones tridiagonales y el cálculo de la transformada *wavelet* en GPU. Además, la resolución de sistemas de ecuaciones tridiagonales forma parte del núcleo de uno de los métodos de segmentación basados en conjuntos de nivel cuya implementación hemos abordado en esta tesis.

Por último, las soluciones desarrolladas en esta Tesis de Doctorado están orientadas a su ejecución en tarjetas gráficas de consumo. Por tanto, quedan fuera de este estudio la implementación de algoritmos en sistemas heterogéneos multi-GPU o en *clusters* con nodos GPU.

Estructura de la tesis

Esta tesis consta de cinco capítulos. El capítulo 1 es una introducción a los conceptos teóricos que sirven de base para este trabajo. Empezamos ofreciendo una perspectiva histórica de los eventos más relevantes que han conducido al desarrollo de las GPU, para luego examinar más en detalle cómo es su arquitectura, con especial énfasis en la arquitectura CUDA. A continuación, describimos los fundamentos teóricos detrás de los métodos del conjunto de nivel (que tienen especial importancia para entender los capítulos 3 y 4) y de la transformada *wavelet* (que utilizamos en los capítulos 2 y 5). Finalizamos el capítulo presentando brevemente las medidas de calidad que hemos empleado a lo largo de este trabajo y los recursos que hemos utilizado.

En el capítulo 2 examinamos las principales técnicas para explotar el paralelismo de la GPU, como el patrón de paralelismo de bucle, el patrón de divergencia/unión, el patrón de descomposición geométrica y el patrón "divide y vencerás". A continuación, introducimos el patrón de diseño paralelo "divide y fusiona", y describimos cómo se puede optimizar la resolución de sistemas tridiagonales de ecuaciones lineales en GPU reestructurando las operaciones de cuatro conocidos algoritmos. Aplicamos también esta metodología al cálculo de la transformada *wavelet*, donde examinamos tres variedades diferentes de la transformada de aplicación común.

En los capítulos 3 y 4 abordamos dos algoritmos de segmentación de imágenes y volúmenes basados en métodos del conjunto de nivel. En el capítulo 3 tratamos el método de segmentación basado en el esquema de división del operador aditivo. Se trata de una variante de los métodos de segmentación clásicos donde cada iteración del algoritmo requiere resolver sistemas tridiagonales de ecuaciones lineales. En este capítulo explicamos cómo hemos aplicado la técnica "divide y fusiona" comentada en el capítulo 2 para implementar este método de segmentación en GPU. Por otra parte, en el capítulo 4 presentamos dos propuestas de implementación del método de segmentación rápida de dos ciclos, y explicamos cómo hemos modificado el flujo de ejecución del algoritmo para poder explotar la arquitectura de la GPU.

El capítulo 5 está centrado en las herramientas de visualización y procesado de volúmenes. Explicamos cómo hemos utilizado Amira para implementar las herramientas de segmentación descritas en los capítulos previos, y cómo hemos extendido VolV para ofrecer soporte a la visualización de volúmenes comprimidos en GPU.

Por último, en el capítulo de las conclusiones hacemos un repaso a todo el trabajo realizado en la tesis y proponemos varias líneas de trabajo futuro.

Principales contribuciones

Las principales contribuciones de esta Tesis de Doctorado son:

– Presentamos un nuevo patrón de paralelismo en el contexto de la computación en GPU, denominado "divide y fusiona". Demostramos cómo el uso de este patrón permite resolver en paralelo y de forma eficiente algoritmos multinivel donde no es posible repartir de forma trivial las operaciones a realizar en bloques independientes. Aplicamos este patrón para implementar en paralelo la resolución de sistemas tridiagonales de ecuaciones lineales y acelerar el cálculo de la transformada *wavelet* [30, 100, 101, 104, 144].

– Presentamos una implementación en GPU del algoritmo de segmentación basado en el esquema de división del operador aditivo [184]. Utilizamos esta técnica en el dominio tridimensional para la segmentación de datos volumétricos. Dividimos el dominio en regiones de tamaño fijo y restringimos las computaciones únicamente a las regiones de interés. Mantenemos simultáneamente varias listas de regiones ordenadas por su adyacencia. Optimizamos la resolución de los sistemas de ecuaciones generados durante la ejecución del algoritmo aplicando la técnica de "divide y fusiona".

– Proponemos dos implementaciones en GPU del algoritmo de segmentación rápida de dos ciclos. Nuestra primera propuesta divide el dominio tridimensional en regiones y utiliza una lista de regiones activas de forma similar a nuestra implementación del esquema de división del operador aditivo. La segunda propuesta utiliza un criterio más restrictivo para identificar las regiones activas, con lo que consigue una mejor eficiencia. Los cálculos para la evolución del frente de segmentación se realizan en la memoria compartida de la GPU, y nuestro método es capaz de detectar y corregir las inconsistencias que se generan al procesar las regiones en paralelo [102, 103, 105].

– Presentamos una solución para la visualización en GPU de datos volumétricos comprimidos. El volumen de entrada se procesa mediante una transformada *wavelet* que genera múltiples niveles de resolución, y se codifica eliminando las componentes que menos información aportan. Nuestra solución acopla el proceso de descompresión y renderizado, que se ejecuta de forma íntegra en la tarjeta gráfica, minimizando así la comunicación entre la CPU y la GPU. Se divide el volumen en particiones, que se descomprimen y renderizan de una en una a un nivel de resolución que depende de la distancia a la cámara [99].

Publicaciones

Los artículos publicados durante la realización de esta Tesis de Doctorado se encuentran recogidos en el siguiente listado de publicaciones, así como los capítulos de la tesis relacionados:

[30] Francisco Argüello, Dora B. Heras, Montserrat Bóo y Julián Lamas-Rodríguez. *The split-and-merge method in general purpose computation on GPUs*. Parallel computing, vol. 38, págs. 277–288, 2012. [Capítulos 1, 2 y 3]

[102] Julián Lamas-Rodríguez, Dora B. Heras, Francisco Argüello, Dagmar Kainmueller, Stefan Zachow y Montserrat Bóo. *GPU-accelerated level-set segmentation*. Journal of Real Time-Image Processing, págs. 1–15, 2013. [Capítulo 4]

[144] Pablo Quesada-Barriuso, Julián Lamas-Rodríguez, Dora B. Heras, Montserrat Bóo y Francisco Argüello. *Selecting the best tridiagonal system solver projected on multicore CPU and GPU platforms*. International Conference on Parallel and Distributed Processing Techniques and Applications, págs. 839–845. Las Vegas (EE.UU.), 2011. [Capítulos 1, 2 y 3]

[100] Julián Lamas-Rodríguez, Francisco Argüello, Dora B. Heras y Montserrat Bóo. *Memory hierarchy optimization for large tridiagonal system solvers on GPU*. 10th IEEE International Symposium on Parallel and Distributed Processing with Applications, págs. 87–94. Leganés (España), 2012. [Capítulos 1, 2 y 3]

[103] Julián Lamas-Rodríguez, Dora B. Heras, Franscisco Argüello, Stefan Zachow y Dagmar Kainmueller. *Partitioning and mapping a fast level-set algorithm on the GPU*. 7th

IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications. Berlín (Alemania), 2013. [Capítulo 4]

[99] Julián Lamas-Rodríguez, Francisco Argüello y Dora B. Heras. *A GPU-based multiresolution pipeline for compressed volume rendering*. International Conference on Parallel and Distributed Processing Techniques and Applications, págs. 523–529. Las Vegas (EE.UU.), 2013. [Capítulo 5]

[101] Julián Lamas-Rodríguez, Montserrat Bóo, Dora B. Heras y Francisco Argüello. *Proyección de algoritmos de resolución de sistemas tridiagonales en la tarjeta gráfica*. XX Jornadas de Paralelismo, págs. 349–354. A Coruña (España), 2009. [Capítulos 1, 2 y 3]

[105] Julián Lamas-Rodríguez, Pablo Quesada-Barriuso, Francisco Argüello, Dora B. Heras y Montserrat Bóo. *Proyección del método de segmentación del conjunto de nivel en GPU*. XXIII Jornadas de Paralelismo, págs. 273–278. Elche (España), 2012. [Capítulo 4]

Capítulo 1

Introducción a la segmentación y visualización en GPU

1.1. Introducción

El trabajo llevado a cabo en esta Tesis de Doctorado se centró en el desarrollo de soluciones eficientes en GPU para segmentar y visualizar datos volumétricos correspondientes a imágenes médicas. La intención de este capítulo es servir de introducción a los principales conceptos teóricos utilizados en las soluciones que presentaremos a lo largo de los siguientes capítulos.

En la sección 1.2 se realiza un repaso histórico al desarrollo de la computación de propósito general en GPU y se describe la arquitectura de las GPU actuales y el modelo de programación utilizado. En la sección 1.3 se tratan los fundamentos teóricos de los conjuntos de nivel, los métodos locales y los modelos de contorno activo; la sección finaliza con una revisión bibliográfica de la segmentación en GPU utilizando conjuntos de nivel. La sección 1.4 sirve de introducción para el análisis de señales utilizando *wavelets*, que luego será de especial interés en el capítulo dedicado a la visualización de volúmenes comprimidos. La sección 1.5 se centra en explicar los principales conceptos detrás del renderizado de volúmenes propiamente dicho. La sección 1.6 presenta los métodos utilizados a lo largo de este trabajo para valorar la calidad de los resultados obtenidos. Por último, la sección 1.7 detalla las características de las GPU y los conjuntos de datos utilizados en este trabajo.

1.2. Introducción a la computación en GPU

Una unidad de procesamiento gráfico (en inglés GPU, *graphics processing unit*) es una pieza de circuitería electrónica diseñada para acelerar los cálculos necesarios para generar y manipular las imágenes que luego son visualizadas en una pantalla.

NVIDIA popularizó el término GPU en 1999, cuando comenzó a comercializar la GeForce 256 como "la primera GPU". Sin embargo, el desarrollo de las primeras unidades de procesamiento gráfico se remonta a casi veinte años antes. En esta sección haremos un breve repaso histórico del desarrollo de la GPU desde la década de 1980, describiremos la arquitectura de las GPU actuales (en particular, la arquitectura CUDA), con especial énfasis en el uso de las tarjetas gráficas para computación de propósito general, y, por último, enumeraremos algunas de las alternativas a CUDA más conocidas. Para el lector interesado en conocer más detalles sobre el desarrollo de la computación de propósito general en GPU, recomendamos la lectura de [8, 93, 152].

1.2.1. Historia de la computación de gráficos tridimensionales

Durante la década de 1980 la compañía Silicon Graphics popularizó los gráficos tridimensionales en una amplia variedad de mercados, que incluían desde la industria militar hasta la industria del entretenimiento. En 1992 Silicon Graphics publicó la librería OpenGL con el objetivo de ofrecer una plataforma estandarizada e independiente para escribir aplicaciones que utilizasen gráficos en 3D. Sin embargo, estos productos solo se encontraban al alcance de empresas, instituciones o usuarios muy especializados.

A mediados de los años 90, la demanda de aplicaciones con gráficos 3D se vio acelerada debido a dos importantes hitos. Por un lado, la aparición de videojuegos que ofrecían entornos 3D renderizados en tiempo real inició la carrera por generar gráficos cada vez más realistas e inmersivos. Por otro lado, compañías como NVIDIA, ATI Technologies y 3dfx Interactive empezaron a comercializar tarjetas aceleradoras gráficas con un coste que podía ser asumido por el consumidor medio. Ambos hitos cimentaron los pilares del desarrollo de la tecnología 3D hasta la fecha de hoy.

En 1996, aprovechando la caída en coste del precio de la memoria DRAM, la compañía 3dfx Interactive entró en el mercado de consumo con su tarjeta gráfica Voodoo Graphics. Esta tarjeta estaba destinada únicamente a proveer aceleración de gráficos tridimensionales. Comparada con las GPU actuales, se trataba de un *hardware* más sencillo que incluía una unidad

de mapeo de textura (que permitía añadir detalles o colores a un modelo tridimensional) y un *framebuffer* (una zona de memoria dedicada específicamente a almacenar la información de color de cada uno de los píxeles en pantalla).

Este nuevo e incipiente mercado, al que no tardaron en sumarse NVIDIA y ATI, vio la salida de sucesivas tarjetas aceleradoras 3D que iban poco a poco incrementando sus prestaciones (velocidad de reloj, tamaño de memoria de vídeo, número de unidades de textura, etc). Sin embargo, no sería hasta 1999 cuando la comercialización por parte de NVIDIA de la GeForce 256 supuso el primer paso en la evolución de la arquitectura GPU tal y como la conocemos hoy en día. A diferencia de otras tarjetas aceleradoras 3D de la misma época, el *hardware* de NVIDIA incorporaba por primera vez en su propio *chip* la arquitectura necesaria para realizar operaciones de transformación, recorte e iluminación (en inglés T&L, *transform, clipping, and lighting*), que hasta el momento eran realizadas únicamente por la CPU. La transformación es el proceso de producir una imagen bidimensional a partir de una escena tridimensional; el recorte implica dibujar solo aquellas partes de la escena que son visibles en la imagen cuando termina el proceso de renderizado; por último, la iluminación es la tarea de modificar el color de las superficies en la escena de acuerdo a las fuentes de luz presentes.

Dado que las operaciones de T&L eran parte integral del *pipeline* gráfico que había estandarizado OpenGL, la GeForce 256 marcó el inicio de una progresión natural para implementar más partes de dicho *pipeline* en el *hardware* del procesador gráfico. En 2001 NVIDIA sacó al mercado la serie GeForce 3 con las primeras GPU que implementaban el nuevo estándar DirectX 8.0 de Microsoft. Este estándar permitió a los desarrolladores tener acceso al conjunto de instrucciones utilizadas por los procesadores encargados de realizar operaciones y modificar las posiciones de los vértices y el color final de cada píxel, es decir, los *vertex shaders* y los *fragment shaders*.

A partir de este momento, el diseño de las GPU evolucionó hacia un *pipeline* unificado, donde las diferentes etapas se ejecutan en procesadores no especializados. Esta tendencia vino obligada por la necesidad de balancear la carga entre las diferentes etapas de procesado en la GPU, con la ventaja de tener un único diseño de procesador, en lugar de diferentes tipos de procesadores encargados de realizar tareas diferentes. Precisamente, ha sido esta tendencia lo que también ha permitido utilizar esta arquitectura para computación numérica más general y, de esta forma, la investigación sobre el uso de *hardware* gráfico ha podido expandirse al campo de las aplicaciones de propósito general, como veremos en la sección 1.2.3.

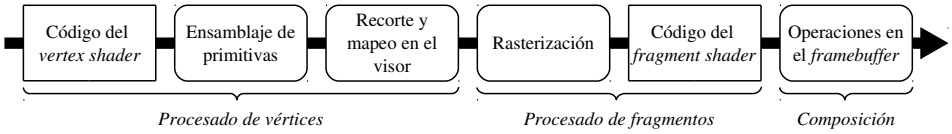


Figura 1.1: Esquema del *pipeline* programable de una GPU.

1.2.2. Pipeline gráfico

La GPU está diseñada para generar imágenes a partir de escenas teseladas (es decir, escenas compuestas de polígonos planos) de forma rápida y eficiente. Todos los dispositivos gráficos 3D implementan este proceso como una secuencia de pasos fijos, que se muestra en la figura 1.1. Esta secuencia de pasos constituye el *pipeline* gráfico.

El *pipeline* procesa una lista ordenada de vértices (con sus coordenadas y otros atributos) y produce a la salida una imagen bidimensional de la escena virtual almacenada en la memoria de vídeo. Las tres principales etapas del *pipeline* son el procesado de vértices, el procesado de fragmentos y la composición. En las siguientes secciones examinaremos brevemente cada una de esas etapas. Para un examen más en detalle de cada etapa, recomendamos la lectura de [62].

Procesado de vértices

En esta etapa, también denominada procesado de geometría, se realizan transformaciones lineales sobre los vértices de entrada. Estas transformaciones lineales incluyen operaciones de rotación, traslación y escalado en el dominio espacial 3D.

En las GPU con *pipeline* fijo (esto es, T&L), el procesado de vértices incluía el cálculo de la iluminación local para cada vértice. Durante el procesado de fragmento, los valores de iluminación eran interpolados de cada vértice, en una técnica conocida como sombreado suave o *goureaud*. Con la aparición de las primeras GPU con *shaders* programables estas operaciones eran realizadas por el procesador de vértices, o *vertex shader*. Las GPU actuales ya no incluyen procesadores específicos para estas tareas.

El código de los *vertex shaders* permite modificar las coordenadas y los atributos de los vértices. El ensamblaje de primitivas agrupa los vértices para construir primitivas geométricas (esto es, puntos, líneas o triángulos). Por último, se recortan aquellas primitivas que caen fuera del visor y se eliminan las que no superen los tests de oclusión. Las primitivas restantes se

mapean en el visor (es decir, se les asigna una coordenada dentro del espacio de coordenadas del visor), para pasar a la siguiente etapa del *pipeline*.

Procesado de fragmentos

Esta etapa recoge las primitivas geométricas generadas en la etapa anterior y realiza un proceso de rasterización, de tal forma que cada primitiva se transforma en un conjunto de fragmentos. Cada fragmento se corresponde con un píxel en la imagen que se obtiene al final del *pipeline*, aunque el color final de cada píxel puede estar determinado por más de un fragmento. Durante la rasterización se calculan los valores de los atributos de cada fragmento interpolando los atributos de cada primitiva. Estos valores servirán de datos de entrada a los *fragment shaders*.

El código de los *fragment shaders* ejecuta la mayor parte de las operaciones tradicionalmente asociadas al proceso de rasterización en un *pipeline* fijo (T&L). Este código calcula el color final y, opcionalmente, el valor de profundidad, de cada fragmento. Para ello, puede acceder a los valores de las texturas almacenadas en la memoria de la GPU y realizar operaciones de filtrado.

Dado que durante esta etapa se obtiene el color final de cada fragmento, es necesario calcular la iluminación local asociada al mismo. Esta técnica se conoce como sombreado *phong*.

Composición

Se trata de la última etapa del *pipeline* gráfico. Se aplican varios tests para determinar qué fragmentos deben ser descartados (por ejemplo, debido a la oclusión de unos fragmentos sobre otros) y cuáles deben ser mostrados en pantalla. Para cada fragmento, se decide cómo combinar su color con el color del píxel correspondiente almacenado en el *framebuffer*.

1.2.3. Computación de propósito general en la GPU

El desarrollo de arquitecturas paralelas es un campo de investigación activo dentro de la computación de altas prestaciones (en inglés HPC, *high performance computing*), con logros tan importantes como el desarrollo de procesadores de múltiples núcleos y la computación en *clusters*, *grids*, y más recientemente, *clouds*. Sin embargo, solo a partir de la década pasada se

empezaron a considerar las tarjetas gráficas como una arquitectura válida para la realización de computación de propósito general.

Como ya se ha comentado, en el año 2001 las GPU iniciaron una evolución desde aceleradores específicos para gráficos con un *pipeline* fijo para llegar a convertirse en procesadores vectoriales programables con una potencia de computación superior a las CPU multinúcleo [112]. Durante el renderizado de gráficos, estos procesadores ejecutan un gran número de operaciones aritméticas de punto flotante sobre datos completamente independientes. Los algoritmos de renderizado son muy eficientes aprovechando el ancho de banda de la memoria y no son muy sensibles a la latencia de memoria; esto explica por qué el diseño de las GPU hace especial énfasis en incrementar la capacidad computacional en punto flotante y añadir funciones gráficas fijas, a la vez que se trata de ampliar el ancho de banda de la memoria sin afectar a su latencia.

Durante los años iniciales de la computación de propósito general en procesadores gráficos (en inglés GPGPU, *general-purpose computing on graphics processing units*) [8], el desarrollo de aplicaciones requería acceder a las funciones de programación gráfica del interfaz de programación (en inglés API, *application programming interface*) para manejar los núcleos del procesador. Para facilitar la implementación de algoritmos paralelos en la GPU, en 2004 el Laboratorio de Gráficos por Computador de la Universidad de Stanford presenta una extensión del lenguaje C denominada Brook. Esta extensión utiliza las API gráficas OpenGL y DirectX de forma transparente para el programador, quien solo debe preocuparse de conocer la nueva sintaxis y nuevas funciones para mover los datos entre la CPU y la GPU.

En 2007 NVIDIA presenta CUDA (*compute unified device architecture*) [16], una arquitectura específicamente diseñada para facilitar la implementación de tareas con grandes requisitos de computación en la GPU. Esta nueva arquitectura iba acompañada de su propia API y un compilador. Los programadores usan *C for CUDA*, una versión de C con varias extensiones para usar las características de la GPU. A diferencia de Brook, CUDA está soportado por el propio *driver* de la tarjeta gráfica, lo que significa que las operaciones en CUDA se traducen directamente a operaciones en la GPU, sin utilizar una API gráfica como capa intermedia. También en 2007 AMD presenta Brook+, una versión de Brook optimizada para su propia línea de GPU. Sin embargo, al ofrecer un entorno más familiar la adopción de CUDA fue mucho más rápida. Con el tiempo, CUDA ha sido integrado con otros lenguajes de programación como C++, Fortran y Python.

En 2008 Apple publica, tras la aprobación del grupo Khronos, del que forman parte empresas como AMD, IBM, Intel y NVIDIA, la primera especificación de OpenCL (en inglés *open computing language*), un entorno para la programación de aplicaciones que puedan ser ejecutadas en múltiples arquitecturas, incluyendo las GPU. OpenCL se compone de una serie de cabeceras y una librería, con funciones invocadas por el programador. Tras descartar Brook+, AMD decide soportar OpenCL en su lugar. Por otra parte, NVIDIA también ofrece soporte a OpenCL a través de su plataforma CUDA.

A pesar de todas las herramientas desarrolladas para facilitar la programación de propósito general en GPU, la implementación directa de algoritmos paralelos suele ofrecer resultados pobres, por lo que es habitual realizar tareas de optimización sobre la gestión de los datos y de la memoria para adaptar los algoritmos a las características específicas de la arquitectura.

1.2.4. Arquitectura CUDA

Una GPU programable consta de varios procesadores multinúcleo capaces de ejecutar cientos de hilos de forma concurrente. Las GPU modernas ofrecen una gran capacidad de computación y un gran ancho de banda de memoria.

En esta sección, presentamos la arquitectura CUDA introducida por NVIDIA en 2007 y que sigue siendo utilizada en las GPU diseñadas hoy en día por esta compañía. Desde que se presentó esta arquitectura, NVIDIA ha ido realizando mejoras incrementales, que se corresponden con las generaciones Tesla, Fermi, Kepler y Maxwell. Cada generación tiene asociada una capacidad computacional, que se identifica con los códigos de versión. Por ejemplo, una GPU con capacidad computacional 1.0 utilizaría arquitectura Tesla, con capacidad 2.1 utilizaría una arquitectura Fermi, y con 3.5 su arquitectura sería Kepler. Cada nueva generación ha supuesto cambios importantes en el diseño de las GPU, por lo que presentamos las diferentes arquitecturas de forma separada, haciendo énfasis en cada caso en las principales características y mejoras. En la guía de programación de CUDA [4] puede encontrarse una descripción más detallada de la arquitectura.

Capacidad computacional 1.x

La figura 1.2 muestra un esquema simplificado de la arquitectura Tesla, donde se han representado los componentes más importantes de la GPU desde el punto de vista de la computación de propósito general. Puede observarse que la arquitectura está organizada en un conjunto de multiprocesadores, denominados *streaming multiprocessors* (SM), cada uno

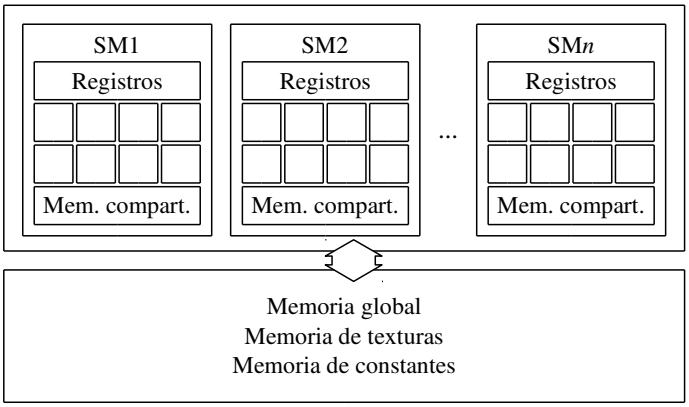


Figura 1.2: Esquema simplificado de la arquictura Tesla.

compuesto de 8 núcleos, denominados *streaming processors* (SP) o, simplemente, CUDA *cores*. Además, cada multiprocesador contiene su propio espacio de registros y un espacio de memoria de 16 kB, denominado *memoria compartida*. Ambos espacios son privados para cada multiprocesador, y su contenido queda invalidado entre la ejecución de un programa CUDA y el siguiente.

Fuera del chip del procesador, se encuentra la memoria DRAM de la GPU, accesible para todos los multiprocesadores. Esta memoria contiene varios espacios, denominados memoria global, memoria de texturas y memoria de constantes. El espacio de memoria global es el de uso más común; es donde la CPU almacena los datos con los que opera la GPU y desde donde la CPU lee los resultados generados por la GPU. Sin embargo, el acceso a esta memoria es hasta 100 veces más lento que la memoria compartida. Los espacios de memoria de texturas y de memoria de constantes aparecen como espacios separados, aunque compartan los mismos bancos de memoria que la memoria global, porque la GPU accede a ambos a través de la caché de texturas y la caché de constantes, respectivamente. No existe una caché intermedia para los accesos a memoria global.

Las primeras tarjetas con arquitectura CUDA que aparecieron en el mercado (serie 8800 Ultra) tienen una capacidad computacional 1.0. La serie 9000 introduce las GPU con capacidad computacional 1.1, capaces de solapar la ejecución de un *kernel* con la transmisión de datos entre la CPU y la GPU (en la mayoría de los modelos). La serie GT 200 apareció en el mercado con una capacidad computacional 1.2. Esta serie aumentaba el número de hilos que

cada multiprocesador podía estar ejecutando de forma concurrente y reducía los problemas de coalescencia de memoria global y conflictos de bancos de memoria compartida presentes en anteriores modelos. La gama más alta de la serie GT 200 tiene capacidad computacional 1.3, y añade soporte para el cálculo en coma flotante de doble precisión.

Capacidad computacional 2.x

Las primeras GPU con capacidad computacional 2.0 (series GTX 300, 400 y 500) incorporan una nueva arquitectura, de nombre en clave Fermi, con importantes cambios sobre las generaciones anteriores. La figura 1.3 muestra un esquema simplificado de esta arquitectura. Aparte del evidente aumento en el número de núcleos por multiprocesador (que pasa a ser 32), la jerarquía de memoria incorpora dos niveles de caché: el nivel L1, que es privado para cada multiprocesador y comparte *hardware* con la memoria compartida (el programador puede ahora configurar 16 kB de memoria compartida y 32 kB de caché L1, o viceversa); y el nivel L2, que, con un tamaño de 768 kB, consiste en una caché unificada compartida entre todos los multiprocesadores.

En la arquitectura Fermi, todos los accesos a memoria global se realizan (por defecto) a través de los dos niveles de caché, lo que supone una importante mejora de rendimiento respecto a arquitecturas previas. Los problemas de coalescencia de memoria se minimizan, aunque a cambio los requisitos de alineamiento de los datos se vuelven más estrictos para un uso óptimo de las cachés. El tamaño de línea de la caché L2 es de 128 bytes; en el caso de programas algunos que realizan accesos dispersos a memoria es, a veces, conveniente desactivar el uso de esta caché.

Algunas GPU más orientadas al mercado del videojuegos (como es el caso de la GTX 460 y la GTX 560) presentan una capacidad computacional 2.1, con 48 núcleos por multiprocesador. Este aumento del número de núcleos tiene como contrapartida la reducción del número de unidades para realizar operaciones en coma flotante de doble precisión. Por otra parte, se incrementa la capacidad de emitir más instrucciones en paralelo por ciclo de ejecución, lo que permite explotar mejor el paralelismo a nivel de instrucción.

Capacidad computacional 3.x

Mientras que el objetivo en las arquitecturas anteriores estaba centrado en incrementar la potencia de cálculo, con la comercialización de las primeras GPU con la arquitectura Ke-

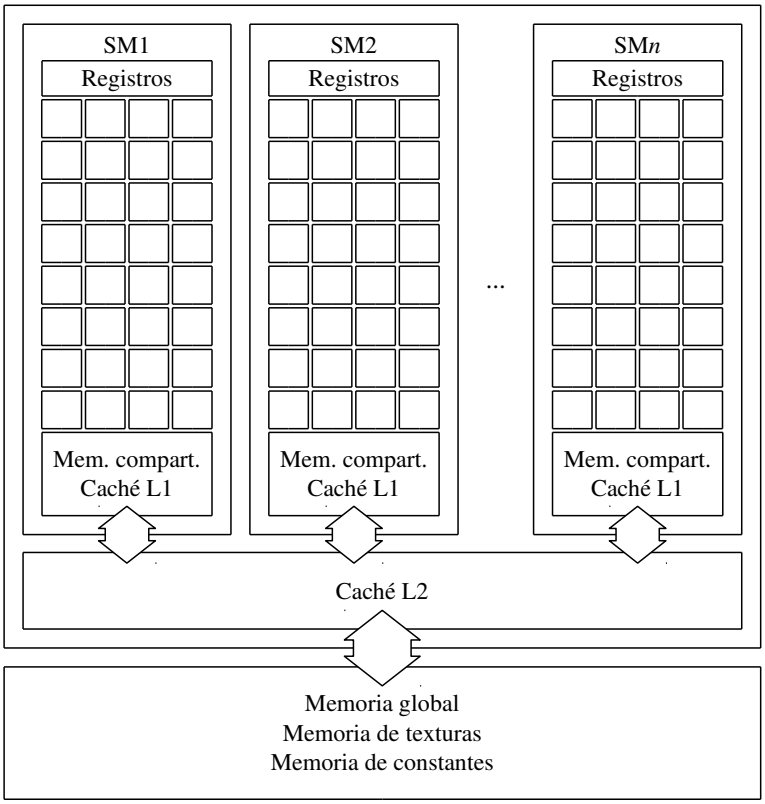


Figura 1.3: Esquema simplificado de la arquictura Fermi.

pler (capacidad computacional 3.0 y 3.5) NVIDIA hace más énfasis en mejorar la eficiencia reduciendo el consumo.

La figura 1.4 muestra un esquema de esta arquitectura, cuyos multiprocesadores (ahora denominados SMX) han sido rediseñados para ofrecer un mejor rendimiento por vatio. Se incrementa el número de núcleos así como el tamaño del espacio de registros, pero como se puede observar en la figura, la jerarquía de memoria se mantiene sin cambios significativos. El espacio dedicado a memoria compartida y a la caché L1 sigue siendo el mismo, aunque se añade la posibilidad de compartirlo a partes iguales. El tamaño de la caché L2 deja de ser fijo, y los mejores modelos incorporan una caché L2 de 1,5 MB. Además, las GPU con capacidad computacional 3.5 añaden una caché de solo lectura de 48 kB (compartida cada

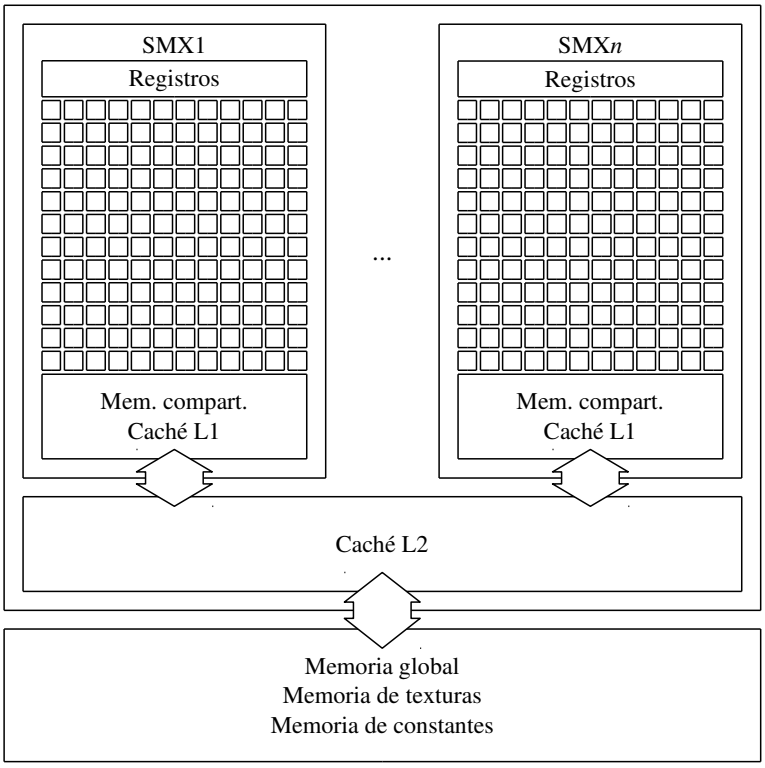


Figura 1.4: Esquema simplificado de la arquictura Kepler.

tres SMX), que puede es utilizada para datos de memoria global que solo son accedidos para lectura durante la ejecución de código CUDA.

1.2.5. Modelo de programación CUDA

El modelo de programación utilizado en CUDA se denomina SPMD (*simple program multiple data*). Cada función CUDA (denominada *kernel*) ejecuta miles de hilos sobre un conjunto de datos. Los hilos se agrupan formando bloques, y estos a su vez se agrupan formando una malla. Tanto los bloques como la malla pueden tener una, dos o tres dimensiones. El programador puede configurar libremente el número de hilos por bloque y el número de bloques invocados por un *kernel*, siempre dentro de los límites de la arquitectura. Como única

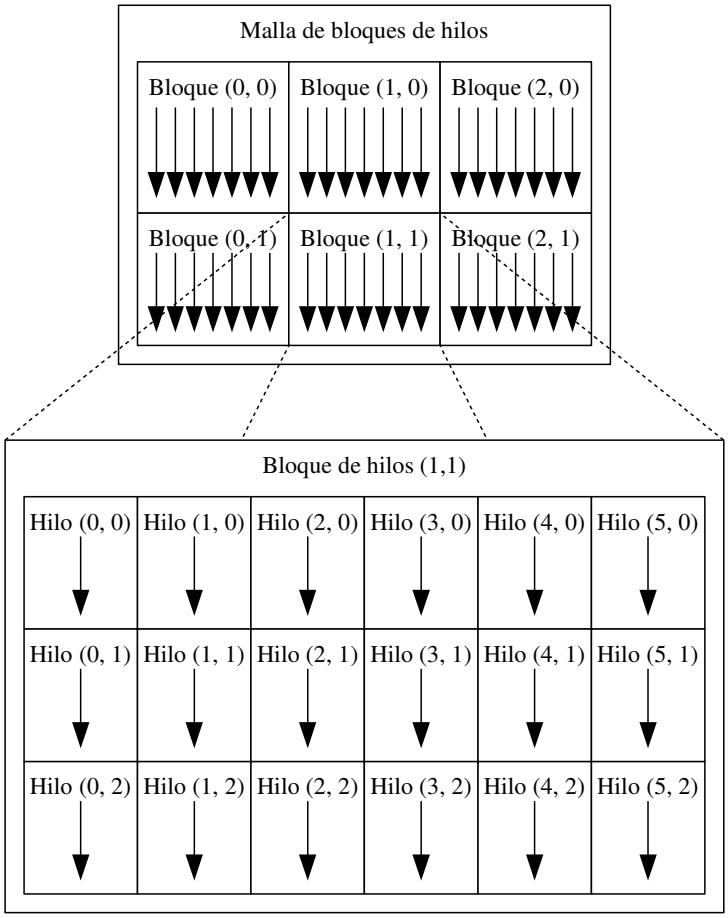


Figura 1.5: Malla de 3×2 bloques de hilos con 6×3 hilos en cada bloque.

restricción, los bloques de hilos deben operar siempre de forma independiente. Aunque este modelo de programación está orientado hacia un nivel de paralelismo de grano fino, también puede ser usado para implementar soluciones de grano más grueso. En la figura 1.5 puede verse un ejemplo de configuración.

El flujo de trabajo de un programa que utiliza la GPU está controlado por la CPU. Normalmente, el proceso comienza cargando los datos en la memoria de la GPU. En determinadas secciones del código la CPU ordenará la ejecución de uno o varios *kernels*, que se ejecutarán

en la GPU. La CPU puede esperar a que la ejecución de los *kernels* finalice, y descargar los resultados de la memoria de la GPU, o seguir ejecutando código y realizar luego una descarga asíncrona.

Cuando se ejecuta un *kernel*, cada bloque de hilos es asignado a un multiprocesador y ejecutado en paralelo. Un multiprocesador puede ejecutar más de un bloque de hilos a la vez, dependiendo de los recursos requeridos por cada bloque. Si se lanzan más bloques de hilos que multiprocesadores disponibles en la GPU, algunos de los bloques de hilos tendrán que esperar a que un multiprocesador termine su trabajo. Esta es la razón fundamental por la que el trabajo realizado por cada bloque de hilos debe ser independiente de otros bloques de hilos. Aunque esto impide que dos bloques de hilos puedan trabajar de forma sincronizada (por ejemplo, compartiendo datos entre ellos), por otro lado hace posible que los programas en CUDA puedan escalar con cada nueva generación de arquitecturas.

La comunicación entre hilos de un mismo bloque puede realizarse estableciendo barreras de sincronización (que solo afectan a los hilos del bloque) y utilizando la memoria global o, preferiblemente, la memoria compartida.

La guía de programación de CUDA [4] contiene un extenso manual sobre el modelo de programación para las GPU de NVIDIA. En la guía de mejores prácticas [3] ofrece numerosos consejos para evaluar y optimizar el rendimiento de los programas en CUDA.

1.2.6. Alternativas a CUDA

CUDA solo es ejecutable oficialmente en *hardware* de NVIDIA. Las dos principales alternativas a CUDA para desarrollar aplicaciones de propósito general en GPU son OpenCL y DirectCompute.

OpenCL [13] es un estándar abierto soportado por NVIDIA, AMD y otras empresas que forman el grupo Khronos para el desarrollo de aplicaciones que realicen tareas de computación intensivas sobre dispositivos como CPU, GPU, o cualquier otro dispositivo para el que exista un *driver* OpenCL. Este estándar plantea una arquitectura virtual similar a la utilizada por CUDA, pero su utilización es algo más compleja. El carácter abierto de OpenCL y la facilidad para portar código entre diferentes arquitecturas convierten a OpenCL en una de las alternativas a CUDA más atractivas.

DirectCompute [10, 11] es la alternativa de Microsoft a CUDA y OpenCL. Dado que forma parte de la API de DirectX, ofrece un entorno de desarrollo familiar para los desa-

rrolladores habituados a trabajar con esta API. Sin embargo, al igual que otros productos de Microsoft, solo está disponible para sistemas operativos Windows.

Por otra parte, existen diferentes API que permiten explotar el paralelismo disponible en las CPU multinúcleo que predominan hoy en día en el mercado. Con OpenMP (*open multi-processing*) [14], por ejemplo, el programador puede utilizar directivas del compilador para dividir la ejecución de bucles y otras tareas de forma automática entre los núcleos disponibles. Con pthreads [39] es posible crear aplicaciones multihilo, aunque la gestión y sincronización de los hilos queda en manos del programador. Aunque pthreads está orientada a entornos Linux, existen versiones de esta librería para Windows [19].

1.3. Segmentación basada en conjuntos de nivel

Los métodos del conjunto de nivel (en inglés, *level-set methods*) [132] son técnicas numéricas para analizar el movimiento de interfaces (que pueden ser curvas o superficies *n*-dimensionales) de forma implícita. Los métodos del conjunto de nivel tienen un gran número de aplicaciones, que incluyen los campos de la física, la química, la mecánica de fluidos, la combustión, la visión por computador y el procesado de imágenes [157].

A diferencia de los métodos explícitos, donde la interfaz está parametrizada, los métodos del conjunto de nivel utilizan una función, denominada función del conjunto de nivel, de forma que la interfaz está definida por aquellos puntos donde la función de conjunto de nivel toma un valor determinado. Al centrarse en el estudio de la evolución en el tiempo de esta función, los cambios en la topología de la interfaz no requieren un tratamiento especial. El uso de métodos del conjunto de nivel para la segmentación de imágenes y volúmenes, donde tanto factores extrínsecos (como pueden ser las intensidades o la textura de la imagen) como intrínsecos (por ejemplo, la curvatura de las regiones segmentadas) determinan el crecimiento de la interfaz, ha demostrado ser una técnica efectiva [185].

En esta sección presentamos los fundamentos matemáticos de los métodos del conjunto de nivel, las principales características de los métodos locales del conjunto de nivel, y los modelos de contorno activo basados en conjuntos de nivel. Por último, haremos una breve revisión de los métodos de segmentación basados en conjuntos de nivel implementados en GPU que han sido publicados en la literatura.

1.3.1. Fundamentos de los conjuntos de nivel

Sea $u(\mathbf{x},t) : \mathbb{R}^{n+1} \rightarrow \mathbb{R}$, con $\mathbf{x} \in \mathbb{R}^n$, una función Lipschitz continua. El conjunto de nivel k de la función u define de forma implícita la hipersuperficie cerrada Γ:

$$\Gamma(t) = \{\mathbf{x} \in \mathbb{R}^n \mid u(\mathbf{x},t) = k\}. \tag{1.1}$$

La hipersuperficie, denominada frente, curva o interfaz, encierra una región cerrada (a veces múltiplemente conectada) denominada Ω. Los métodos del conjunto de nivel computan y analizan el movimiento de Γ y, en consecuencia, la deformación de Ω. La propagación del frente está asociada a la evolución de una función u [157]. Normalmente se suele escoger como u la función de distancia a Γ, con las siguientes propiedades [130]:

$$u(\mathbf{x},t) < 0 \quad \text{if } \mathbf{x} \in \Omega \tag{1.2}$$

$$u(\mathbf{x},t) > 0 \quad \text{if } \mathbf{x} \notin \Omega \tag{1.3}$$

$$u(\mathbf{x},t) = 0 \quad \text{if } \mathbf{x} \in d\Omega = \Gamma(t). \tag{1.4}$$

Tomemos el frente asociado a $k = 0$. Sea entonces $\mathbf{x}(t)$ la posición en cada instante t de una partícula situada en el frente. El valor de la función del conjunto de nivel para cualquier posible punto en la ruta definida por $\mathbf{x}(t)$ en cualquier debe ser cero en todo momento:

$$u(\mathbf{x}(t),t) = 0. \tag{1.5}$$

Para estudiar la variación de u en el tiempo es necesario calcular su derivada. Si se deriva la ecuación anterior y se aplica la regla de la cadena, se obtiene la siguiente ecuación parcial diferencial de primer orden [132]:

$$\frac{du(\mathbf{x}(t),t)}{dt} - \nabla u(\mathbf{x}(t),t) \cdot \frac{d\mathbf{x}(t)}{dt} = 0. \tag{1.6}$$

Sea F la función de velocidad en la dirección del vector normal $\mathbf{n}$, es decir, $F = \frac{d\mathbf{x}(t)}{dt} \cdot \mathbf{n}$, donde $\mathbf{n} = \frac{\nabla u}{|\nabla u|}$. La ecuación de evolución puede reescribirse entonces como:

$$\frac{du(\mathbf{x}(t),t)}{dt} - F|\nabla u| = 0, \tag{1.7}$$

que es la ecuación del conjunto de nivel originalmente definida en [132]. La ecuación de actualización utilizada normalmente en los métodos del conjunto de nivel se define como:

$$u(\mathbf{x},t+\Delta t) = u(\mathbf{x},t) + \Delta t F|\nabla u(\mathbf{x},t)|. \tag{1.8}$$

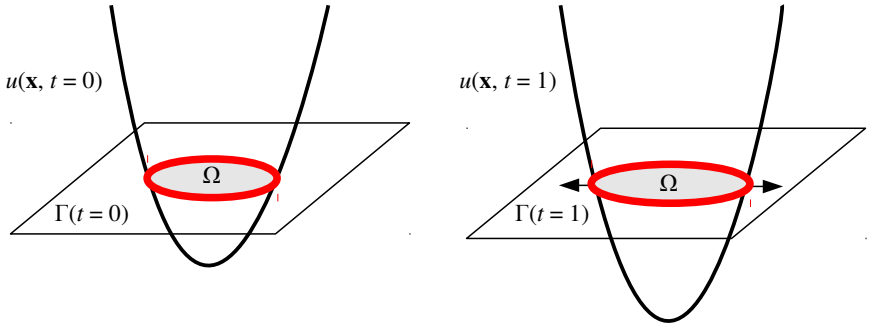


Figura 1.6: Estado de la función del conjunto de nivel en dos instantes de tiempo diferentes.

La figura 1.6 muestra la evolución del frente sobre una superficie 2D en dos instantes de tiempo distintos. En este caso, el frente Γ es una curva bidimensional delimitada por el conjunto de nivel cero de u en un espacio tridimensional. El frente aparece representado por una línea gruesa, y la región encerrada por el frente, Ω, se muestra sombreada. A la izquierda en la figura, el frente está en su estado inicial $t = 0$. En el instante $t = 1$, la función del conjunto de nivel ha cambiado, delimitando una nueva región. En este ejemplo, el frente ha crecido hacia afuera.

1.3.2. Métodos locales del conjunto de nivel

El cálculo de la evolución de la función del conjunto de nivel u en todo el dominio D, tal y como aparece en la ecuación (1.8), no es computacionalmente eficiente. Por esta razón, se han desarrollado los métodos locales del conjunto de nivel, que restringen las computaciones únicamente a aquellas regiones que sean relevantes dentro del dominio. Los métodos locales del conjunto de nivel asumen que solo los puntos cercanos al frente (es decir, aquellos donde $u \simeq 0$) son de interés (este es el caso, por ejemplo, de los algoritmos de segmentación basados en métodos del conjunto de nivel).

Las dos estrategias para implementar métodos locales del conjunto de nivel que aparecen con más frecuencia en la literatura son el esquema de banda estrecha y el de campo disperso [27, 186]. En estos métodos la evolución de u, que necesariamente es definida como una

función de distancia, está limitada a un "tubo" que rodea el frente, y la ecuación de actualización del conjunto de nivel en (1.8) se reescribe de la siguiente forma:

$$u(\mathbf{x},t+\Delta t) = \begin{cases} u(\mathbf{x},t) & \text{if } |u(\mathbf{x},t)| > r \\ u(\mathbf{x},t) + \Delta t F |\nabla u(\mathbf{x},t)| & \text{if } |u(\mathbf{x}),t| \leq r, \end{cases} \tag{1.9}$$

donde r es el radio del tubo que delimita el tamaño del dominio activo.

No es necesario recalcular la posición del tubo en cada iteración del proceso de evolución del conjunto de nivel. Las técnicas de banda estrecha solo actualizan el dominio activo cuando el frente llega al borde del tubo. Las técnicas de campo disperso actualizan el dominio activo más a menudo, cada pocas iteraciones, y el tubo que rodea al frente es más estrecho y contiene solo un pequeño conjunto de posiciones [186, 136]. En cualquier caso, ambas estrategias reducen la complejidad computacional asociada a actualizar la función del conjunto de nivel de $O(n^3)$ a aproximadamente $O(n^2)$ en un dominio 3D con una sección transversal de tamaño n.

Otras técnicas analizan la evolución del conjunto de nivel utilizando *quadtrees* y *octrees* [168, 169]. Una malla *quadtree* en $\mathbb{R}^2$ (o una malla *octree* en el caso de $\mathbb{R}^3$) es una estructura jerárquica compuesta de celdas organizadas en L niveles. La celda raíz, que cubre todo el dominio, está en el nivel 0, y cada celda en el nivel l puede contener hasta 2^2 subceldas más pequeñas (o 2^3 subceldas en dominios 3D) en el nivel $l+1$.

En los métodos basados en *quadtrees* y *octrees* el dominio está dividido en una malla de celdas. La construcción de esta malla es un proceso de refinamiento cuyo objetivo es situar las celdas más pequeñas cerca del frente Γ. Los valores de u se calculan y almacenan solo en los vértices de dicha malla, que se reconstruye en cada iteración del algoritmo. El uso de una malla adaptativa dificulta el empleo de esquemas de diferencias finitas de orden mayor [178], por lo que estos métodos se apoyan en esquemas semi-lagrangianos, como el CIR [53, 167], para resolver la ecuación (1.8). La complejidad computacional de cada paso en estos métodos es de $O(m \log m)$, donde m es el número de elementos en Γ.

Los métodos de malla de bloques dispersa [37] utilizan bloques pequeños de tamaño fijo para representar la banda estrecha que rodea a Γ. A diferencia de otros métodos de banda estrecha, estos métodos no almacenan todo el dominio de u en memoria, sino solo aquellos bloques atravesados por Γ, es decir, los bloques activos. En particular, el método de lista ordenada de regiones (en inglés STL, *Sorted Tile List*) [178, 87] utiliza una lista de regiones activas ordenada por sus coordenadas. Durante la evolución del frente se añaden y eliminan regiones del dominio activo. Aunque la lista debe ser reconstruida frecuentemente, cada pa-

so del algoritmo tiene una complejidad computacional de $O(p)$, donde p es el número de regiones activas.

Un problema común a los métodos locales del conjunto de nivel es la reinicialización: cada vez que el dominio activo se actualiza, los valores de u que en iteraciones previas estaban localizados fuera del dominio activo deben ser recalculados para asegurar que u sigue siendo una función de distancia dentro del nuevo dominio activo. Hay varias aproximaciones que tratan de resolver este problema sin tener que recalcular la función de distancia, y que van desde realizar un seguimiento del gradiente de u a utilizar aproximaciones iterativas que convergen en la función de distancia al conjunto de nivel cero.

1.3.3. Modelos de contorno activo

Algunos de los métodos más conocidos que estudian la modificación de un frente se basan en la minimización de una función de energía. En estos métodos, se hace evolucionar el frente tanto por la influencia de fuerzas internas como externas, con el objetivo de identificar los bordes de una región o un objeto dentro de una imagen o un volumen.

Los primeros modelos asociados a la minimización de una función de energía fueron los modelos explícitos de contorno activo (o *snakes*) [42, 63]. En estos modelos, el frente Γ en cada instante de tiempo t es una curva (o superficie) parametrizada definida como $\Gamma(q) : [0,1] \rightarrow \mathbb{R}^n$ cuya energía asociada viene dada por la siguiente ecuación:

$$E(\Gamma) = \alpha \int_0^1 \left| \frac{d\Gamma}{dt} \right|^2 dq + \beta \int_0^1 \left| \frac{d^2\Gamma}{dt^2} \right|^2 dq - \gamma \int_0^1 |\nabla v(\Gamma(q))| \, dq, \tag{1.10}$$

donde $v : D \rightarrow \mathbb{R}$ es el volumen escalar n-dimensional sobre el que se quiere realizar el proceso de segmentación, α y β son parámetros que permiten controlar la suavidad del frente (energía interna) y, por último, γ es un parámetro que permite controlar la fuerza que atrae el frente hacia la región a segmentar en la imagen o el volumen (energía externa). Resolver el problema de *snakes* se reduce a encontrar una curva Γ que, para unos determinados valores de α, β y γ, minimice el valor de energía E.

Estos modelos presentan problemas a la hora de manejar cambios topológicos. Para solucionar estas limitaciones, se desarrollaron los modelos implícitos de contorno activo, basados en los métodos del conjunto de nivel. El primer modelo desarrollado fue el modelo geométrico [42, 114], que utiliza la siguiente ecuación diferencial parcial:

$$\frac{\partial u}{\partial t} = g(\mathbf{x}) |\nabla u| \mathrm{div} \left(\frac{\nabla u}{|\nabla u|} \right) + k\, g(\mathbf{x}) \, |\nabla u|, \tag{1.11}$$

donde k es una constante real y $g : \mathbb{R}^n \to [0,1)$ es la función de parada que detiene el avance del frente cuando se acerca a un borde de la región a segmentar.

Puesto que la curvatura $\kappa = \text{div}\left(\frac{\nabla u}{|\nabla u|}\right)$, la ecuación del modelo geométrico puede reescribirse como:

$$\frac{\partial u}{\partial t} = g(\mathbf{x})\,(k+\kappa)|\nabla u|. \tag{1.12}$$

Así, dada la siguiente equivalencia:

$$\frac{\partial u}{\partial t} = (k+\kappa)|\nabla u| \qquad \equiv \qquad \frac{\partial \Gamma}{\partial t} = (k+\kappa)\,\mathbf{n}, \tag{1.13}$$

Podemos concluir que en el modelo geométrico la evolución del frente está determinada por dos términos. El término $k\,\mathbf{n}$ es similar a una "fuerza globo" [48] que empuja el frente hacia adentro (o hacia afuera), mientras que el término $\kappa\,\mathbf{n}$ acorta la longitud del frente a la vez que lo suaviza. En resumen, $(k+\kappa)$ actúa de forma similar a la fuerza interna utilizada en el modelo de *snakes*.

Como función de parada es habitual utilizar el estimador de bordes de la ecuación de difusión anisotrópica propuesta por Perona y Malik [137]. Sea v_σ el resultado de aplicar un filtrado gaussiano con varianza σ sobre el volumen v:

$$v_\sigma(\mathbf{x}) = v * f_\sigma = v * \left(\frac{1}{(\sqrt{2\pi}\sigma)^n} e^{-\frac{\sum_{i=0}^{n} x_i^2}{2\sigma^2}}\right), \tag{1.14}$$

donde $*$ denota el operador de convolución. Este filtrado reduce el número de máximos y mínimos locales en los valores de intensidades del volumen, de forma que solo los bordes más visibles de las estructuras del volumen se mantienen. La función de parada se define como:

$$g(\mathbf{x}) = \frac{1}{1+\left(\frac{|\nabla v_\sigma(\mathbf{x})|}{\lambda}\right)^2}. \tag{1.15}$$

Esta ecuación garantiza la función de parada tendrá valores cercanos a 0 en aquellos vóxeles donde el valor de gradiente $|\nabla v_\sigma(\mathbf{x})|$ sea más alto, y cercanos a 1 cuando el gradiente tenga valores muy pequeños. El frente tenderá a detenerse en aquellas zonas con el valor de gradiente más alto.

λ es un parámetro que permite ajustar cómo se distribuyen los valores de la función de parada. La figura 1.7 muestra una visualización de los valores de la función de parada para

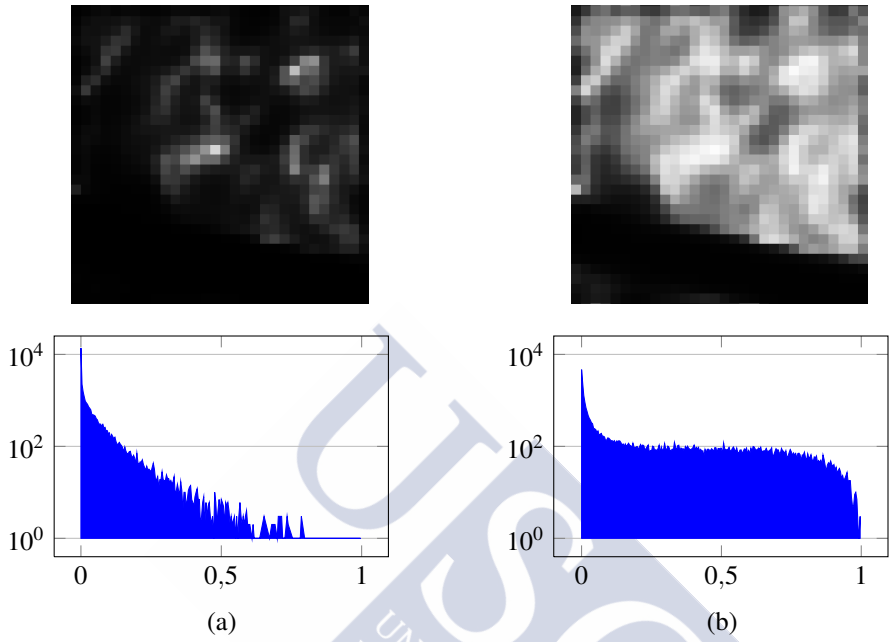


Figura 1.7: Visualización e histograma de la función de parada para (a) $\lambda = 1$ y (b) $\lambda = 5$.

$\lambda = 1$ y $\lambda = 5$ en un volumen (del que solo se muestra una lámina bidimensional). Debajo de cada imagen se muestra el histograma de la función de parada correspondiente a cada valor de λ. Puede observarse que para un valor de λ pequeño los valores de la función de parada tienden a acumularse cerca del 0, pero al incrementar este valor los valores se distribuyen más homogéneamente dentro del rango $[0,1)$.

Al modelo geométrico implícito le seguiría, años más tarde, el desarrollo del modelo geodésico [43, 91, 163]. En este modelo, la ecuación de evolución del conjunto de nivel se define como:

$$\frac{\partial u}{\partial t} = |\nabla u| \mathrm{div}\left(g(\mathbf{x}) \frac{\nabla u}{|\nabla u|} \right). \tag{1.16}$$

Al igual que antes, podemos reescribir la ecuación (1.16) de la siguiente forma:

$$\frac{\partial u}{\partial t} = g(\mathbf{x}) |\nabla u| \mathrm{div}\left(\frac{\nabla u}{|\nabla u|} \right) + \nabla g(\mathbf{x}) \cdot \nabla u, \tag{1.17}$$

por lo que el modelo geodésico es similar al modelo geométrico presentado en la ecuación (1.11) con la excepción del término $\nabla g \cdot \nabla u$. Este producto escalar incrementa la atrac-

ción del frente hacia los bordes presentes en la región a segmentar. A diferencia del modelo geométrico, el modelo geodésico es capaz de detectar bordes con valores de gradiente poco homogéneos e, incluso, con pequeños saltos.

El modelo geodésico puede incorporar también un término similar a la "fuerza globo" para acelerar la velocidad de convergencia y evitar que la evolución del frente se detenga en mínimos locales. En este caso, la ecuación del modelo quedaría como:

$$\frac{\partial u}{\partial t} = |\nabla u| \mathrm{div}\left(g(\mathbf{x})\frac{\nabla u}{|\nabla u|}\right) + k\, g(\mathbf{x})|\nabla u|. \tag{1.18}$$

En el capítulo 3 presentaremos una implementación de un algoritmo de segmentación en GPU que utiliza el modelo geodésico.

1.3.4. Segmentación mediante métodos del conjunto de nivel en GPU

La primera implementación en GPU de segmentación utilizando un método del conjunto de nivel fue propuesta en [150]. Esta implementación discretiza el cálculo de la evolución del conjunto de nivel mediante diferencias finitas en una malla uniforme. Por otra parte, el cálculo del gradiente se realiza utilizando un esquema *upwind* [132] implementado como operaciones de textura ejecutadas en el *buffer* de la tarjeta.

Años más tarde se desarrolla en [111] una de las implementaciones en GPU del método del conjunto de nivel más referenciadas en la literatura. Esta solución, que se puede clasificar dentro de los métodos del conjunto de nivel de banda estrecha, divide el volumen en regiones que son procesadas en paralelo. Además, para dar soporte a esta solución, los autores desarrollaron una implementación *ad hoc* de memoria virtual en la GPU, mucho antes de que tecnologías como CUDA y OpenCL fuesen de uso extendido. Más tarde, en [94] se ampliaría este trabajo para dar soporte a volúmenes grandes añadiendo un sistema de *caching*.

En [143] se presenta una implementación en GPU para el seguimiento de cuerpos en vídeos. Este trabajo es una extensión de [127] y utiliza métodos de conjuntos de nivel para calcular flujos ópticos y flujos de masas en la GPU. Aunque hay más literatura relacionada al respecto, esta solución ya se aleja de la segmentación y se adentra en el campo del *tracking*, que no es de interés en este trabajo.

En [159] se presenta una de las primeras implementaciones en CUDA de métodos de segmentación basados en conjuntos de nivel. En particular, este trabajo estaba enfocado a la identificación automática de bancos de peces a partir de imágenes acústicas del fondo marino. Este tipo de imágenes presenta bordes especialmente borrosos, y los autores desarrollan un método

del conjunto de nivel a partir de la minimización del funcional de energía definido en [131], utilizando el término de parada propuesto en [45] basado en las técnicas de segmentación de Mumford-Shah [44, 124]. Los autores afirman que este modelo es más ventajoso cuando se pretende segmentar bordes borrosos o cuya intensidad varía suavemente. Posteriormente, en [160, 161] se amplía esta implementación para dar soporte a segmentación multidominio.

Al mismo tiempo, en [88] se presenta otra solución en CUDA orientada a segmentar neuronas utilizando métodos del conjunto de nivel. Esta solución combina varias técnicas para tratar de detectar el centro y la dirección del axón de cada neurona y realizar una segmentación bidimensional de su membrana. La segmentación está basada en la técnica de lazo activo [179], donde dos frentes se desplazan e interactúan entre sí manteniendo siempre una topología de lazo. La implementación en CUDA está basada en el trabajo de [111], aunque utiliza operaciones atómicas en la GPU para gestionar el dominio activo.

Por otra parte, en [78] se propone una solución programada en CUDA y orientada a segmentar volúmenes grandes usando un clúster de GPU. En este trabajo, la ecuación del conjunto de nivel se calcula mediante una aplicación del método LBM *(Lattice Boltzmann Model)* [191], que realiza el cálculo de la curvatura de forma implícita. Aunque este método se traslada de manera natural a la forma que tiene de procesar la GPU, los autores reconocen que tiene un coste mucho mayor en consumo de memoria que otras soluciones en la literatura. Más tarde, en [32] también se utilizaría el método LBM en GPU para ofrecer una solución orientada a segmentar bordes borrosos.

En [145, 146] se describe con detalle una solución CUDA para segmentar imágenes MRI y CT del cuerpo humano basándose en métodos del conjunto de nivel. En este caso el cálculo de la función de conjunto de nivel es similar a la usada en [111], aunque, en lugar de trabajar con regiones, utiliza una lista para mantener exactamente qué elementos pertenecen al dominio activo. La lista se actualiza y mantiene en GPU sin recurrir al uso de operaciones atómicas, por lo que no presenta las pérdidas de rendimiento que ocurren en [88]. Otra contribución importante es la capacidad de detectar qué elementos han cambiado de valor en cada paso de ejecución, y así poder descartar del dominio activo aquellos que ya se hayan estabilizado. Los autores afirman que esta es la primera implementación en GPU de segmentación de conjuntos de nivel con una complejidad lineal en tiempo de procesamiento $O(n)$ y complejidad logarítmica en número de pasos $O(\log n)$, donde n es el tamaño del volumen.

En [173] se presenta una solución en OpenCL para realizar segmentaciones multiobjeto. El reto de este tipo de soluciones es actualizar simultáneamente múltiples funciones de con-

juntos de nivel, cada una de ellas configurada para segmentar un objeto determinado, con la condición de que al final de la segmentación no puede haber solapamientos entre los objetos.

En [87] puede encontrarse una de las implementaciones más recientes de conjuntos de nivel en GPU, aplicada al ámbito de reconstrucción de superficies. Se trata de una reimplementación del método STL *(Sorted Tile List)* publicado por los mismos autores en [97] donde la ecuación de conjunto de nivel se discretiza mediante un esquema *upwind*. En [139] se presenta una implementación en OpenCL basada en el método de segmentación de [111], aunque utilizando una aproximación global en lugar de banda estrecha. Esta solución se caracteriza por hacer uso de un esquema de pasos adaptativos con el objetivo de mejorar la estabilidad de la evolución del conjunto de nivel.

Es posible encontrar también en la literatura implementaciones que modifican la ecuación de conjunto de nivel para realizar tareas de segmentación muy específicas, como por ejemplo, segmentación de nódulos [153] o de órganos completos [172].

En el capítulo 3 presentaremos una implementación de banda estrecha en GPU de un algoritmo de segmentación basado en el conjunto de nivel que utiliza el método AOS. En el capítulo 4 presentaremos dos propuestas de implementación en GPU del algoritmo de segmentación FTC, que utiliza un método disperso del conjunto de nivel.

1.4. Wavelets

En el campo de la teoría de señal, una *wavelet* es una oscilación de corta duración y con forma de onda. Su amplitud empieza siendo cero, y luego se incrementa hasta volver a ser cero. La convolución de una *wavelet* con otras señales es la base de la transformada *wavelet*, que permite extraer información de la señal que inicialmente no era evidente en el dominio del tiempo.

En esta sección presentamos algunos de los principios teóricos fundamentales de la transformada *wavelet* y otras transformadas derivadas. La teoría detrás de la transformada *wavelet* es muy amplia y un estudio detallado está fuera de los objetivos de esta tesis. Para una exploración más en profundidad recomendamos la lectura de [116].

1.4.1. Transformada wavelet continua ortonormal

Sea $\mathbf{L^2}(\mathbb{R})$ un espacio de funciones de energía finita en $\mathbb{R}$ que verifica:

$$\forall f \in \mathbf{L^2}(\mathbb{R}), \quad \|f\|_2 = \left(\int_{-\infty}^{+\infty} |f(t)|^2 \, dt \right)^{\frac{1}{2}} < +\infty. \tag{1.19}$$

Una *wavelet* es una función $\psi \in \mathbf{L^2}(\mathbb{R})$ cuya media es cero:

$$\int_{-\infty}^{+\infty} \psi(t) \, dt = 0, \tag{1.20}$$

La función ψ está normalizada ($\|\psi\|_2 = 1$) y centrada en $t = 0$. A partir de esta función, que se denomina función base o *wavelet* madre, es posible obtener una familia de funciones átomo de tiempo-frecuencia tras aplicar escalados y traslaciones de acuerdo a los parámetros s y u:

$$\psi_{u,s} = \frac{1}{\sqrt{s}} \psi \left(\frac{t-u}{s} \right). \tag{1.21}$$

Estos átomos están también normalizados ($\|\psi_{u,s}\|_2 = 1$). La transformada *wavelet* continua de una función $f \in \mathbf{L^2}(\mathbb{R})$ a escala s y en la posición u se define como:

$$W_\psi f(u,s) = \langle f, \psi_{u,s} \rangle = \int_{-\infty}^{+\infty} f(t) \psi_{u,s}^* \, dt = \int_{-\infty}^{+\infty} f(t) \frac{1}{\sqrt{s}} \psi^* \left(\frac{t-u}{s} \right) dt, \tag{1.22}$$

donde ψ^* denota el conjugado de ψ. La ecuación anterior puede reescribirse como el resultado de una convolución:

$$W_\psi f(u,s) = \int_{-\infty}^{+\infty} f(t) \frac{1}{\sqrt{s}} \psi^* \left(\frac{t-u}{s} \right) dt = f * \bar{\psi}_s(u), \tag{1.23}$$

donde $*$ denota el operador de convolución y:

$$\bar{\psi}_s(t) = \frac{1}{\sqrt{s}} \psi^* \left(\frac{-t}{s} \right). \tag{1.24}$$

La señal transformada proporciona información sobre el tiempo y la frecuencia. El parámetro de traslación t está relacionado con la posición de la función ψ cuando se desplaza a lo largo de la función f, por lo que se obtiene información sobre la correspondencia de f con la *wavelet* en cada instante de tiempo. El parámetro de variación de escala s expande o comprime la función; un valor alto está asociado a las bajas frecuencias y proporciona información oculta en la función, mientras que un valor pequeño está asociado a las altas frecuencias y proporciona una información global de la función.

1.4.2. Análisis multirresolución de Mallat

El análisis multirresolución de Mallat permite adaptar una señal a diferentes niveles de resolución (o diferentes escalas). Esta descomposición tiene multitud de aplicaciones en procesos de reducción de ruido, de compresión y de visión por computador. Por ejemplo, en el capítulo 5 mostraremos un esquema multirresolución piramidal que permite visualizar volúmenes a baja resolución e incrementar de forma selectiva la resolución de determinadas zonas.

Función de escala

Una aproximación de una función $f \in \mathbf{L^2}(\mathbb{R})$ a una resolución 2^{-j}, con $j \in \mathbb{Z}$, es una proyección ortogonal f_j en el subespacio $\mathbf{V}_j \subset \mathbf{L^2}(\mathbb{R})$ que minimiza $\|f - f_j\|$. Denotamos esta proyección como $P_{\mathbf{V}_j} f$.

Una secuencia $\{\mathbf{V}_j\}_{j\in\mathbb{Z}}$ de subespacios cerrados de $\mathbf{L^2}(\mathbb{R})$ es una aproximación multirresolución si verifica las siguientes propiedades:

– $\mathbf{V}_j$ es invariante a cualquier traslación proporcional a 2^j:

$$\forall (j,k) \in \mathbb{Z}^2,\ f(t) \in \mathbf{V}_j \Leftrightarrow f(t - 2^j k) \in \mathbf{V}_j. \tag{1.25}$$

– Una aproximación a una resolución 2^{-j} contiene la información necesaria para calcular una aproximación a una resolución 2^{-j-1}:

$$\forall j \in \mathbb{Z},\ \mathbf{V}_{j+1} \subset \mathbf{V}_j. \tag{1.26}$$

– Incrementar la escala en 2 en $\mathbf{V}_j$ incrementa los detalles por 2:

$$\forall j \in \mathbb{Z},\ f(t) \in \mathbf{V}_j \Leftrightarrow f\left(\frac{t}{2}\right) \in \mathbf{V}_{j+1}. \tag{1.27}$$

– Cuando la resolución tiende a 0, se pierden todos los detalles:

$$\lim_{j\to+\infty} \mathbf{V}_j = \bigcap_{j=-\infty}^{+\infty} \mathbf{V}_j = \{0\}, \tag{1.28}$$

y, además:

$$\lim_{j\to+\infty} \|P_{\mathbf{V}_j}\| = 0. \tag{1.29}$$

– Cuando la resolución tiende a $+\infty$, la señal de aproximación converge en la original:

$$\lim_{j\to-\infty} \mathbf{V}_j = clausura\left(\bigcup_{j=-\infty}^{+\infty} \mathbf{V}_j\right) = \mathbf{L^2}(\mathbb{R}), \tag{1.30}$$

y, por tanto:

$$\lim_{j\to-\infty} \|f - P_{\mathbf{V}_j}\| = 0. \tag{1.31}$$

Para calcular la proyección $P_{\mathbf{V}_j}$ es necesario encontrar una base ortonormal de $\mathbf{V}_j$. Esta base ortonormal se construye escalando y trasladando una única función ϕ, denominada *función de escala*. Esta función tiene una media de uno:

$$\int_{-\infty}^{+\infty} \phi(t)\, dt = 1. \tag{1.32}$$

Las dilataciones y traslaciones de ϕ se construyen de la siguiente forma:

$$\phi_{j,n}(t) = \frac{1}{\sqrt{2^j}}\phi\left(\frac{t-2^j n}{2^j}\right). \tag{1.33}$$

La familia $\{\phi_{j,n}\}_{n\in\mathbb{Z}}$ es una base ortonormal de $\mathbf{V}_j$ para todo $j \in \mathbb{Z}$.

La proyección ortogonal de f en $\mathbf{V}_j$ se calcula como:

$$P_{\mathbf{V}_j} = \sum_{n=-\infty}^{+\infty} a_j[n]\, \phi_{j,n}. \tag{1.34}$$

Cada valor de la secuencia $a_j[n]$ es una aproximación discreta de f calculada a partir de un filtrado paso bajo muestreado en intervalos de 2^j:

$$a_j[n] = \langle f, \phi_{j,n}\rangle = \int_{-\infty}^{+\infty} f(t)\, \frac{1}{\sqrt{2^j}}\, \phi^*\left(\frac{t-2^j n}{2^j}\right) dt = f * \bar{\phi}_j(2^j n), \tag{1.35}$$

donde $*$ denota el operador de convolución y:

$$\bar{\phi}_j(t) = \frac{1}{\sqrt{2^j}}\, \phi^*\left(\frac{t}{2^j}\right). \tag{1.36}$$

Función *wavelet*

Sea $\mathbf{W}_j$ el complemento ortogonal de $\mathbf{V}_j$ en $\mathbf{V}_{j+1}$, es decir:

$$\mathbf{V}_{j-1} = \mathbf{V}_j \oplus \mathbf{W}_j, \tag{1.37}$$

donde $\oplus$ es el operador de suma directa. La proyección ortogonal de f en $\mathbf{V}_{j-1}$ puede descomponerse como la suma de las proyecciones ortogonales en $\mathbf{V}_j$ y $\mathbf{W}_j$:

$$P_{\mathbf{V}_{j-1}} = P_{\mathbf{V}_j} + P_{\mathbf{W}_j}. \tag{1.38}$$

El complemento $P_{\mathbf{W}_j}$ contiene los "detalles" de f que aparecen en la escala 2^{j-1} pero que no están presentes en la escala 2^j. Al igual que antes, para calcular la proyección $P_{\mathbf{W}_j}$ es necesario encontrar una base ortonormal de $\mathbf{W}_j$. En este caso, la base ortonormal se construye escalando y trasladando una única función ψ, la *función wavelet*:

$$\psi_{j,n}(t) = \frac{1}{\sqrt{2^j}} \psi \left(\frac{t - 2^j n}{2^j} \right). \tag{1.39}$$

Para cualquier escala 2^j, la familia de dilataciones $\{\psi_{j,n}\}_{n \in \mathbb{Z}}$, es una base ortonormal de $\mathbf{W}_j$. Además, $\{\psi_{j,n}\}_{(j,n) \in \mathbb{Z}^2}$ es una base ortonormal de $\mathbf{L^2}(\mathbb{R})$.

La proyección ortogonal de f en $\mathbf{W}_j$ se calcula como:

$$P_{\mathbf{W}_j} = \sum_{n=-\infty}^{+\infty} d_j[n] \, \psi_{j,n}. \tag{1.40}$$

$d_j[n]$ es la transformada *wavelet* de f que contiene los detalles en la resolución 2^{-j}:

$$d_j[n] = \langle f, \psi_{j,n} \rangle = \int_{-\infty}^{+\infty} f(t) \frac{1}{\sqrt{2^j}} \psi^* \left(\frac{t - 2^j n}{2^j} \right) dt = f * \bar{\psi}_j(2^j n), \tag{1.41}$$

donde $*$ denota el operador de convolución y:

$$\bar{\psi}_j(t) = \frac{1}{\sqrt{2^j}} \psi^* \left(\frac{t}{2^j} \right). \tag{1.42}$$

Por último, la suma de todos los detalles en todas las resoluciones 2^{-j} que van desde $+\infty$ a 0, o de forma equivalente, de todas las escalas 2^j que van desde 0 a $+\infty$, da lugar a la función original:

$$f = \sum_{j=-\infty}^{+\infty} P_{\mathbf{W}_j} f = \sum_{j=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} d_j[n] \, \psi_{j,n}. \tag{1.43}$$

Filtros espejo conjugados

Por la propiedad descrita en la ecuación (1.26), se verifica que:

$$\frac{1}{\sqrt{2}} \phi \left(\frac{t}{2} \right) \in \mathbf{V}_1 \subset \mathbf{V}_0. \tag{1.44}$$

Dado que $\{\phi(t-n)\}_{n\in\mathbb{Z}}$ es una base ortonormal de $\mathbf{V}_0$, es posible descomponer:

$$\frac{1}{\sqrt{2}}\,\phi\left(\frac{t}{2}\right) = \sum_{n=-\infty}^{+\infty} h[n]\,\phi(t-n), \tag{1.45}$$

donde:

$$h[n] = \left\langle \frac{1}{\sqrt{2}}\,\phi\left(\frac{t}{2}\right), \phi(t-n) \right\rangle. \tag{1.46}$$

Por otra parte, dado que $\psi\left(\frac{1}{2}\right) \in \mathbf{W}_1 \subset \mathbf{V}_0$, es posible descomponer:

$$\frac{1}{\sqrt{2}}\,\psi\left(\frac{t}{2}\right) = \sum_{n=-\infty}^{+\infty} g[n]\,\psi(t-n), \tag{1.47}$$

donde:

$$g[n] = \left\langle \frac{1}{\sqrt{2}}\,\psi\left(\frac{t}{2}\right), \phi(t-n) \right\rangle. \tag{1.48}$$

Las secuencias $h[n]$ y $g[n]$ pueden interpretarse como filtros discretos, denominados *filtros espejo conjugados*. A continuación mostraremos cómo se utilizan ambos filtros para realizar el análisis multirresolución.

Bancos de filtrado

La descomposición de una señal en las secuencias $a_j[n]$ y $d_j[n]$ de coeficientes *wavelet* se suele realizar aplicando sobre la señal operaciones sucesivas de filtrado de paso bajo y de paso alto entre las que se intercalan operaciones de submuestreo.

El análisis comienza con una señal $f \in \mathbf{V}_0$. Por tanto, los coeficientes $d_j[n]$ son diferentes de cero solo para valores de $j > 0$. En cada paso del análisis, la proyección $P_{\mathbf{V}_j} f$ se descompone en una aproximación $P_{\mathbf{V}_{j+1}} f$ más los coeficientes de la proyección $P_{\mathbf{W}_{j+1}} f$. De esta forma, el proceso de descomposición se realiza aplicando las ecuaciones:

$$a_{j+1}[p] = \sum_{n=-\infty}^{+\infty} h[n-2p]\,a_j[n] \tag{1.49}$$

$$d_{j+1}[p] = \sum_{n=-\infty}^{+\infty} g[n-2p]\,a_j[n] \tag{1.50}$$

Por otra parte, el proceso de síntesis o reconstrucción se realiza con la siguiente ecuación:

$$a_j[p] = \sum_{n=-\infty}^{+\infty} h[p-2n]\,a_{j+1}[n] + \sum_{n=-\infty}^{+\infty} g[p-2n]\,d_{j+1}[n] \tag{1.51}$$

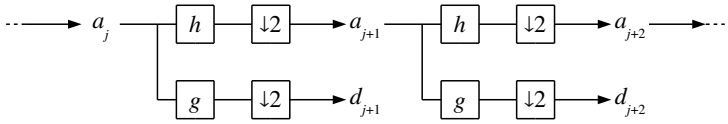


Figura 1.8: Descomposición *wavelet* en árbol de Mallat de dos niveles.

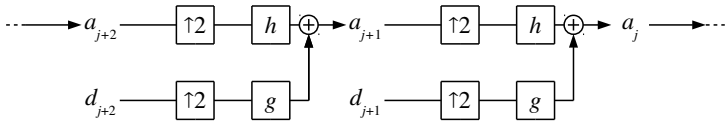


Figura 1.9: Reconstrucción de la señal original en dos niveles.

La figura 1.8 muestra dos niveles de descomposición de una señal en coeficientes *wavelet*. Este proceso da lugar a un esquema conocido como árbol de Mallat. La figura 1.9 muestra el proceso de reconstrucción de la señal a partir de los coeficientes.

Los coeficientes *wavelet* iniciales de $f \in \mathbf{V}_0$ se calculan como $a_0[n] = \langle f(t), \phi(t-n) \rangle$, de forma que $a_0[n]$ es una media local de f entorno a n, pero no es estrictamente igual a $f(n)$. En la mayor parte de los escenarios la señal analógica original es capturada por un dispositivo de resolución finita, a partir del cual se obtiene una señal discreta. En general es posible utilizar esta señal discreta como señal inicial a_0.

1.4.3. Lifting

En los campos de procesado de señales y compresión de datos la reconstrucción perfecta de una señal a la que se le ha aplicado una transformada *wavelet* tiene una importancia crítica. Se dice que un algoritmo *wavelet* permite una reconstrucción perfecta cuando al aplicar la transformada *wavelet* inversa sobre el resultado de una transformada *wavelet* se obtiene exactamente la señal original.

La técnica de *lifting* [55] es una modificación sobre los filtros *wavelet* de reconstrucción perfecta que mejora algunas de sus propiedades. Al igual que antes, el proceso de análisis se realiza encadenando múltiples pasos de descomposición de la señal entre los que se intercalan operaciones de submuestreo y de filtrado. La figura 1.10 muestra un esquema del desarrollo de cada uno de los pasos. Los filtros $\bar{h}[n] = \delta[n]$ y $\bar{g}[n] = \delta[n+1]$, donde δ es la función

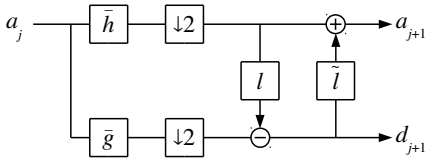


Figura 1.10: Descomposición *wavelet* con *lifting*.

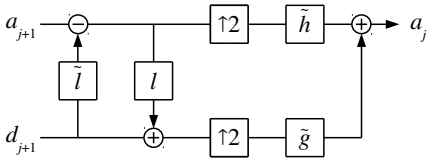


Figura 1.11: Reconstrucción de la señal original usando *lifting*.

delta de Dirac, se utilizan para desplazar la señal a, que luego es submuestreada. El filtro l realiza un filtrado de paso alto consistente en predecir para cada muestra impar de la señal el correspondiente valor de la muestra par, de forma que la diferencia entre la predicción y el valor real se almacena en la señal d. Las diferencias serán más pequeñas cuanto mejor sea la predicción realizada por l. El filtro $\tilde{l}$ realiza un filtrado de paso bajo consistente en actualizar las muestras impares a partir de cada una de las predicciones realizadas. El objetivo de este filtro es mejorar las predicciones en el siguiente paso de descomposición.

El proceso de síntesis se completa también en varios pasos complementarios al proceso de análisis. La figura 1.11 ilustra cada uno de estos pasos. En este caso los filtros l y $\tilde{l}$ permiten reconstruir las muestras impares y pares de la señal original, respectivamente, a partir de los valores de a y d. Los filtros $\tilde{h}[n] = \delta[n]$ y $\tilde{g}[n] = \delta[n-1]$ se utilizan para alinear las muestras tras aplicar un supermuestreo y así obtener la señal original.

1.4.4. Bases wavelet bidimensionales y tridimensionales

Una imagen bidimensional $f(x,y)$ puede aproximarse a una resolución 2^{-j} proyectando dicha imagen en el subespacio vectorial $\mathbf{V}_j^2 \subset \mathbf{L^2}(\mathbb{R}^2)$. Denotamos esa proyección como $P_{\mathbf{V}_j^2} f$.

Sea $\{\mathbf{V}_j\}_{j\in\mathbb{Z}}$ una aproximación multirresolución de $\mathbf{L^2}(\mathbb{R})$. Una aproximación multirresolución bidimensional separable se calcula como:

$$\mathbf{V}_j^2 = \mathbf{V}_j \otimes \mathbf{V}_j, \tag{1.52}$$

donde $\otimes$ es el producto tensorial. Sea ϕ la función de escala tal que $\{\phi_{j,n}\}_{n\in\mathbb{Z}}$ es una base ortogonal de $\mathbf{V}_j$. Entonces, por la ecuación (1.52), $\{\phi_{j,n,m}(x,y)\}_{(n,m)\in\mathbb{Z}^2}$ es una base ortogonal de $\mathbf{V}_j^2$, donde las diferentes traslaciones y dilataciones de $\phi_{j,n,m}$ se construyen como:

$$\phi_{j,n,m}(x,y) = \phi_{j,n}(x)\,\phi_{j,m}(y) = \frac{1}{2^j}\,\phi\left(\frac{x-2^j n}{2^j}\right)\phi\left(\frac{y-2^j m}{2^j}\right). \tag{1.53}$$

Sea el subespacio de detalle $\mathbf{W}_j^2$ igual al complemento ortogonal de $\mathbf{V}_j^2$ en $\mathbf{V}_{j-1}^2$, esto es, $\mathbf{V}_{j-1}^2$ es un espacio de mayor resolución que puede construirse a partir de la siguiente suma directa:

$$\mathbf{V}_{j-1}^2 = \mathbf{V}_j^2 \oplus \mathbf{W}_j^2. \tag{1.54}$$

Sea ψ la función *wavelet* correspondiente a la función de escala ϕ que genera una base ortonormal de $\mathbf{L^2}(\mathbb{R})$. Definimos tres funciones *wavelet* bidimensionales:

$$\psi^1(x,y) = \phi(x)\psi(y), \tag{1.55}$$

$$\psi^2(x,y) = \psi(x)\phi(y), \tag{1.56}$$

$$\psi^3(x,y) = \psi(x)\psi(y). \tag{1.57}$$

La familia de *wavelets* $\{\psi^1_{j,n,m}, \psi^2_{j,n,m}, \psi^3_{j,n,m}\}_{(j,n,m)\in\mathbb{Z}^3}$ es una base ortonormal de $\mathbf{W}_j^2$, donde:

$$\psi^k_{j,n,m}(x,y) = \frac{1}{2^j}\,\psi^k\left(\frac{x-2^j n}{2^j}, \frac{y-2^j m}{2^j}\right) \quad \text{para } 1 \le k \le 3. \tag{1.58}$$

El esquema de descomposición basado en bancos de filtrado puede extenderse a las dos dimensiones. Para todas las escalas $2^j \ge 1$, denotamos:

$$a_j[n,m] = \langle f, \phi_{j,n,m}\rangle \tag{1.59}$$

$$d_j^k[n,m] = \langle f, \psi^k_{j,n,m}\rangle \quad \text{para } 1 \le k \le 3. \tag{1.60}$$

La imagen discreta original puede usarse como señal de entrada $a_0[n,m] = \langle f, \phi_{0,n,m}\rangle$.

Para cada paso de descomposición, el proceso se realiza en dos etapas, como se muestra en la figura 1.12. En primer lugar, la imagen se procesa por filas, aplicando una transformación

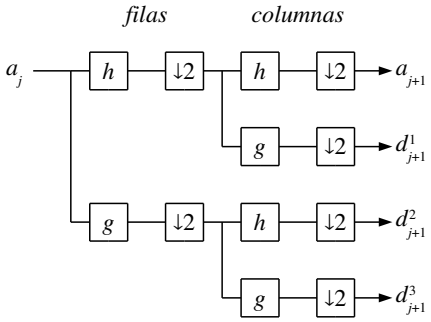


Figura 1.12: Esquema de un paso de descomposición *wavelet* para una señal bidimensional.

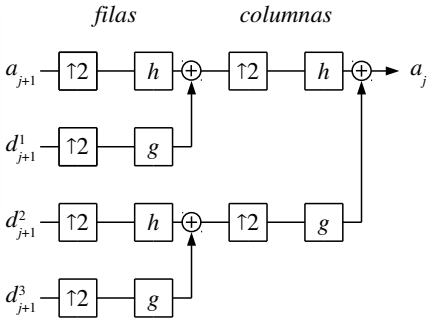


Figura 1.13: Reconstrucción de la señal bidimensional original.

unidimensional a cada fila. A continuación, sobre el resultado obtenido se realiza una transformación unidimensional, esta vez por columnas. La figura 1.13 muestra un esquema de cada paso del proceso de reconstrucción, que ejecuta operaciones similares a la descomposición, pero en orden inverso.

Los esquemas de descomposición y reconstrucción pueden ser extendidos de forma similar a las tres dimensiones, realizando para cada paso la transformación separada por filas, por columnas y, por último, por capas.

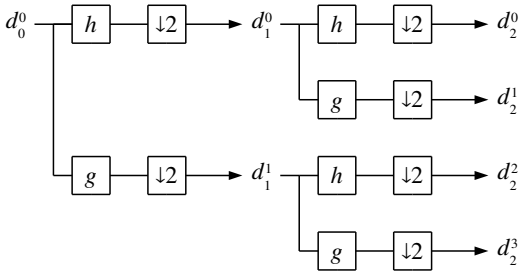


Figura 1.14: Esquema de los dos primeros pasos de descomposición de una transformada paquete *wavelet*.

1.4.5. Transformada paquete wavelet

La transformada paquete *wavelet* [49] es una generalización de la transformada *wavelet* vista en las secciones anteriores: además de dividir los subespacios de aproximación $\mathbf{V}_j$ para construir los subespacios de detalles $\mathbf{W}_j$ y las bases *wavelet*, se dividen también los subespacios $\mathbf{W}_j$ para generar nuevas bases.

La figura 1.14 muestra de forma esquemática el proceso de descomposición de una señal en una transformada paquete *wavelet*. Al igual que antes, a partir de la señal original se generan dos nuevas señales aplicando los filtros de aproximación y de detalles. Sin embargo, en este caso sobre ambas señales se vuelve a aplicar a un proceso de filtrado que genera cuatro nuevas señales. El proceso continua repitiéndose de forma recursiva, y puede verse que este tipo de transformada genera al final un árbol binario.

Las transformadas paquete *wavelet* tienen aplicaciones en la reducción de ruido y la compresión de datos [120, 176], donde se utilizan para calcular la base más representativa de los datos relativa a una función de coste particular.

1.5. Renderizado de volúmenes

Las técnicas tradicionales de gráficos por computador renderizan superficies tridimensionales utilizando mallas poligonales, NURBS (en inglés *nonuninform rational B-spline*) o subdivisión de superficies. En el campo de renderizado de volúmenes, existe una amplia variedad de técnicas para generar imágenes a partir de datos escalares en tres dimensiones. Este tipo de datos es común en el campo de la medicina, como resultado de técnicas como la to-

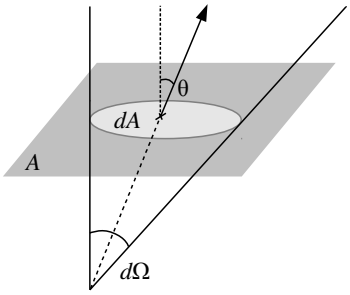


Figura 1.15: Representación de un rayo de luz atravesando un plano.

mografía computerizada (en inglés CT, *computerized tomography*) y la resonancia magnética (en inglés MRI, *magnetic resonance imaging*).

En esta sección repasamos los fundamentos teóricos que sustentan el renderizado de volúmenes y presentamos la técnica de renderizado basada en el mapeo de texturas tridimensionales, que utilizaremos en nuestra solución para compresión y visualización de volúmenes propuesta en el capítulo 5. Puede encontrarse una descripción más exhaustiva en [62].

1.5.1. Fundamentos teóricos

El renderizado de volúmenes está basado en los fundamentos físicos detrás de la óptica geométrica, donde se asume que la luz se propaga en línea recta hasta que interactúa con el medio que atraviesa. Hay tres tipos de interacción: emisión, donde el material atravesado emite luz y, por tanto, incrementa la energía radiativa; absorción, donde el material reduce la energía radiativa; y dispersión, donde la luz cambia su dirección de propagación (lo que puede tener como efecto que la cantidad de energía incremente o disminuya).

La energía de un rayo de luz se describe mediante su radiancia, que se define como la energía radiativa Q por unidad de área A, por ángulo sólido Ω y por unidad de tiempo t:

$$I = \frac{dQ}{dA_{\perp}\, d\Omega\, dt}, \tag{1.61}$$

donde $A_{\perp} = A\cos\theta$, siendo θ el ángulo entre la dirección de la luz y el vector normal a la superficie A. La figura 1.15 muestra esquemáticamente el modelo físico utilizado para calcular la radiancia.

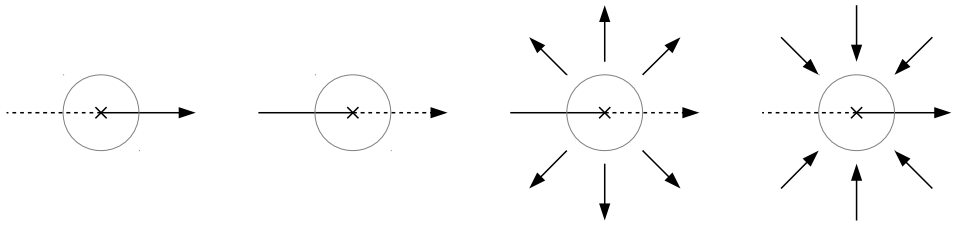


Figura 1.16: Tipos de interacción entre la luz y el medio participante. De izquierda a derecha: emisión, absorción, dispersión hacia afuera y dispersión hacia adentro.

La radiancia de un rayo de luz a lo largo del vacío se mantiene constante, y solo cambia cuando interacciona con un material como consecuencia de un efecto de emisión, absorción o dispersión. Se distinguen, además, dos tipos de dispersión: dispersión hacia afuera, donde el rayo de luz en la dirección actual pierde energía debido a que se dispersa en otras direcciones, y dispersión hacia adentro, donde el rayo de luz gana energía cuando se suman a la dirección actual otros rayos cuyas direcciones eran diferentes. La figura 1.16 muestra esquemáticamente los cuatro tipos de interacción.

De la combinación de todos los efectos de las posibles interacciones entre la luz y el medio surge la siguiente ecuación de transferencia de luz:

$$\vec{\omega} \cdot \nabla I(\mathbf{x}, \vec{\omega}) = -\chi I(\mathbf{x}, \vec{\omega}) + \eta, \tag{1.62}$$

donde $\vec{\omega}$ es la dirección del rayo de luz. El término a la izquierda de la ecuación representa la variación de la radiancia en la dirección del rayo de luz para un punto $\mathbf{x}$ del espacio. El coeficiente χ define la absorción total del medio y se calcula como:

$$\chi = \kappa + \sigma, \tag{1.63}$$

donde κ es el coeficiente de absorción verdadera (por ejemplo, debido a la conversión de luz en calor) y σ es el coeficiente de dispersión hacia afuera. El coeficiente η determina la emisión total del medio y se calcula como:

$$\eta = q + j, \tag{1.64}$$

donde q representa la emisión (por ejemplo, debido a excitación térmica) y j es el coeficiente de dispersión hacia adentro. Todos estos coeficientes dependen tanto de la posición $\mathbf{x}$ como de la dirección del rayo $\vec{\omega}$, pero por simplificar la notación no se incluyen en las ecuaciones. Los

términos κ, σ, y q se corresponden con propiedades ópticas del medio material, y se asignan directamente a través de una función de transferencia (véase sección 1.5.3). Sin embargo, el coeficiente j es calculado en cada punto $\mathbf{x}$ a partir de las propiedades del material y teniendo en cuenta todas las posibles direcciones donde hubiera luz incidente en dicho punto.

Dado que la solución de la ecuación de transporte de la luz es computacionalmente costosa, se suelen utilizar modelos simplificados. En el renderizado de volúmenes el modelo más común es el de emisión-absorción, donde el material puede emitir luz y absorber luz incidente, pero los efectos de dispersión y de luz incidente son ignorados. En este modelo, la ecuación de transporte definida en (1.62) se reescribe como:

$$\vec{\omega} \cdot \nabla I(\mathbf{x}, \vec{\omega}) = -\kappa(\mathbf{x}, \vec{\omega}) I(\mathbf{x}, \vec{\omega}) + q(\mathbf{x}, \vec{\omega}), \tag{1.65}$$

y se denomina ecuación de renderizado de volúmenes. Cuando se considera un único rayo de longitud s, la ecuación correspondiente es:

$$\frac{dI(s)}{ds} = -\kappa(s) I(s) + q(s). \tag{1.66}$$

Esta ecuación diferencial puede resolverse integrando en la dirección del rayo desde una posición s_0 hasta una posición final D, obteniendo así la integral de renderizado de volúmenes:

$$I(D) = I_0 e^{-\int_{s_0}^{D} \kappa(t)\,dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} \kappa(t)\,dt}\,ds, \tag{1.67}$$

donde I_0 representa la luz que llega desde el fondo de la escena y entra en el volumen en la posición s_0, e $I(D)$ es la luz que abandona el volumen en la posición D y llega finalmente a la cámara. El primer sumando de la ecuación representa la atenuación de la luz de fondo que produce el material del volumen; el segundo sumando representa la suma de todas las contribuciones en el interior del volumen atenuadas por el propio material del volumen a lo largo del rayo. El término de atenuación que aparece en ambos sumandos se denomina transparencia:

$$T(s_1, s_2) = e^{-\int_{s_1}^{s_2} \kappa(t)\,dt}, \tag{1.68}$$

por lo que se puede reescribir la integral de renderizado de volúmenes como:

$$I(D) = I_0 T(s_0, D) + \int_{s_0}^{D} q(s) T(s, D)\,ds. \tag{1.69}$$

El principal objetivo del renderizado de volúmenes es calcular la integral definida en la ecuación (1.69). Dado que esta integral no se puede evaluar de forma analítica se utilizan

métodos numéricos para tratar de encontrar una aproximación lo más cercana posible a la solución. Así, el dominio de integración se divide en n intervalos equidistantes con las posiciones $s_0 < s_1 < \cdots < s_{n-1} < s_n$, donde s_0 es el punto de inicio del dominio de integración y $s_n = D$ es el punto final. De esta forma, la radiancia en un punto s_i se calcula como:

$$I(s_i) = I(s_{i-1})\,T(s_{i-1}, s_i) + \int_{s_{i-1}}^{s_i} q(s)\,T(s, s_i)\,ds. \tag{1.70}$$

Sea $T_i = T(s_{i-1}, s_i)$ la transparencia del intervalo i-ésimo, y $c_i = \int_{s_{i-1}}^{s_i} q(s)\,T(s, s_i)\,ds$ la contribución de color (o contribución de radiancia). La radiancia en el punto final D se puede calcular como:

$$I(D) = I(s_n) = I(s_{n-1})\,T_n + c_n = (I(s_{n-2})\,T_{n-1} + c_{n-1})\,T_n + c_n = \ldots \tag{1.71}$$

Es decir:

$$I(D) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j, \text{ con } c_0 = I(s_0). \tag{1.72}$$

La transparencia y la contribución de color se pueden aproximar como:

$$\begin{aligned} T_i &\approx e^{-\kappa(s_i)\,\Delta x}, \\ c_i &\approx q(s_i)\,\Delta x, \end{aligned} \tag{1.73}$$

donde $\Delta x = (D - s_0)/n$ es la longitud de cada intervalo. De esta forma, a partir de la ecuación (1.72) la integral de renderizado se puede aproximar como una suma de Riemann de segmentos constantes. Esta es la aproximación que se emplea en el mapeo de texturas, que es la técnica utilizada en nuestra solución para el renderizado de volúmenes.

1.5.2. Mapeo de texturas tridimensionales

La GPU solo soporta el renderizado de primitivas gráficas compuestas de puntos, líneas o polígonos, por lo que renderizar cualquier objeto volumétrico requiere descomponer el volumen en estas primitivas [62]. La técnica de mapeo de texturas tridimensionales se basa en la idea de representar un volumen escalar discreto como una pila de láminas bidimensionales; si se utiliza un número lo suficientemente elevado de láminas semitransparentes extraídas del volumen, entonces es posible visualizar datos tridimensionales. Las láminas están compuestas de polígonos construidos con las primitivas soportadas por la GPU. Estas láminas constituirán

la *geometría proxy*, que determina la forma del dominio de los datos, habitualmente un prisma que delimita los límites de cada lámina, pero no determina la forma del objeto contenido en los datos.

El conjunto de datos que contiene el objeto volumétrico está estructurado como una malla uniforme que es posible almacenar en la memoria de la GPU como una textura tridimensional. Cada elemento de la textura (o téxel) almacena un valor escalar. Las propiedades ópticas de emisión y absorción de cada punto del volumen se calculan a partir de dicho valor escalar utilizando una función de transferencia. La textura se mapea en las láminas, trasladando las coordenadas de los vértices dentro de la geometría *proxy* a coordenadas de textura.

Las dos operaciones más importantes que intervienen en este proceso son la *interpolación* y la *composición*. La operación de interpolación se utiliza para obtener valores (o muestras) de los datos almacenados en posiciones que no estaban previamente definidas en el conjunto de datos original (en general, muestras entre vóxeles adyacentes), y obtener así el valor final de cada fragmento (es decir, sus propiedades ópticas). La operación de composición aproxima la integral de renderizado calculando de forma iterativa la ecuación (1.72) para obtener el renderizado final en pantalla como resultado de combinar todas las contribuciones de cada uno de los fragmentos de las láminas de la geometría *proxy*.

Interpolación

El proceso de interpolación permite abordar el problema de reconstrucción de un volumen escalar discreto en un espacio tridimensional continuo (esto es, la escena de renderizado). La interpolación se realiza para cada una de las dimensiones de forma combinada en un proceso de filtrado que se puede describir como un producto de tensores:

$$h(x,y,z) = h_x(x) \cdot h_y(y) \cdot h_z(z), \tag{1.74}$$

donde h_x, h_y y h_z son filtros de un solo parámetro aplicados en las direcciones x, y y z. De esta forma, la interpolación trilineal (en tres dimensiones) se realiza a partir de una secuencia de interpolaciones lineales.

La figura 1.17 muestra el proceso de interpolación lineal, bilineal y trilineal para una función escalar discreta f definida en una malla donde la distancia entre dos muestras adyacentes es 1. La interpolación lineal entre dos puntos $\mathbf{a}$ y $\mathbf{b}$ se calcula como:

$$f(\mathbf{p}) = (1-x)\, f(\mathbf{a}) + x\, f(\mathbf{b}), \tag{1.75}$$

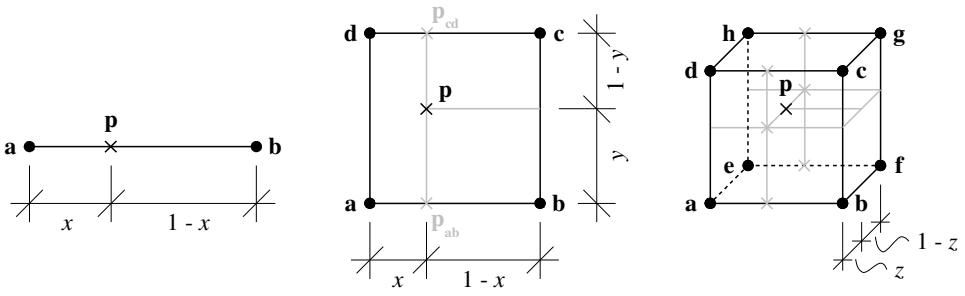


Figura 1.17: Diferentes tipos de interpolación en una, dos y tres dimensiones. De izquierda a derecha: lineal, bilineal y trilineal.

donde $f(\mathbf{a})$ y $f(\mathbf{b})$ son los valores de la función en los puntos **a** y **b** y x es la distancia entre uno de los puntos y **p**, el punto donde se interpola el valor de f. Para el caso de interpolación bilineal, el valor de f se calcula a partir de las interpolaciones lineales para los puntos $\mathbf{p_{ab}}$ y $\mathbf{p_{cd}}$:

$$f(\mathbf{p}) = (1-y)\, f(\mathbf{p_{ab}}) + y\, f(\mathbf{p_{cd}}), \tag{1.76}$$

donde y es la distancia entre uno de los puntos anteriores y el punto **p**. De forma análoga, la interpolación trilineal calcula el valor de f en el caso tridimensional:

$$f(\mathbf{p}) = (1-z)\, f(\mathbf{p_{abcd}}) + z\, f(\mathbf{p_{efgh}}), \tag{1.77}$$

donde z es la distancia entre $\mathbf{p_{abcd}}$ y el punto p, y los valores de $f(\mathbf{p_{abcd}})$ y $f(\mathbf{p_{efgh}})$ se calculan a partir de la interpolación bilineal de las dos caras del cubo. Estos cálculos pueden extenderse para una malla uniforme donde las distancias entre las muestras no sean homogéneas.

Este tipo de interpolación es sencillo y rápido, y por lo general la tarjeta gráfica ofrece soporte directo y de forma transparente a este tipo de operaciones para texturas uni-, bi- y tridimensionales, por lo que se utiliza ampliamente para el renderizado de volúmenes.

Composición

La otra operación fundamental para renderizado de volúmenes es la composición. Este proceso es la base para aproximar la integral de renderizado de volúmenes (ecuación (1.72)) de forma iterativa. Aunque hay dos tipos de esquemas básicos de composición, de delante

hacia atrás y de atrás hacia delante, en esta sección describiremos el segundo esquema, que es el utilizado por nuestra solución de renderizado de volúmenes (véase capítulo 5).

En el esquema de composición de atrás hacia delante se entiende que los rayos de luz llegan desde el fondo, atraviesan el volumen y terminan en la cámara, donde se recoge su valor de luminancia. Las ecuaciones que describen este esquema están definidas para $i = 0, \ldots, n$:

$$\hat{C}_i = \hat{C}_{i-1}(1-\alpha_i) + C_i, \tag{1.78}$$

$$\hat{T}_i = \hat{T}_{i-1}(1-\alpha_i), \tag{1.79}$$

con la siguiente inicialización:

$$\hat{C}_0 = C_0, \tag{1.80}$$

$$\hat{T}_0 = 1-\alpha_0, \tag{1.81}$$

donde C_i y $1-\alpha_i$ son, respectivamente, el color y la transparencia[1] del nodo i-ésimo proporcionado por la función de transferencia (véase más adelante), y que aproximan los valores de c_i y T_i definidos en la ecuación (1.73).

La figura 1.18 muestra cómo funciona el esquema de composición de atrás hacia adelante para un único rayo de luz. Cuando el rayo atraviesa el volumen, se recogen a intervalos regulares los valores de color y transparencia. Por último, cuando el rayo llega a la pantalla, se calcula el color y la transparencia final para el píxel correspondiente a partir de los valores recogidos a lo largo del rayo.

En las soluciones de renderizado de volúmenes con mapeo de texturas la GPU realiza el proceso de composición de forma transparente al programador. Cada fragmento tiene asociado un valor de color C_i y de transparencia T_i, ambos determinados por la función de transferencia a partir del valor interpolado sobre el conjunto de datos original para la coordenada correspondiente al fragmento. Las láminas de la geometría *proxy* se dibujan de atrás hacia adelante sobre el *framebuffer*, de forma que cada fragmento contribuye con su valor de color (y en función de su valor de transparencia) al contenido previamente almacenado en dicho *buffer*. Por último, el contenido del *framebuffer* se traslada a la pantalla.

1.5.3. Función de transferencia

En el campo de la visualización científica los conjuntos de datos volumétricos contienen valores abstractos que representan alguna propiedad del volumen que varía con el espacio. La

[1] A α_i se le suele denominar opacidad.

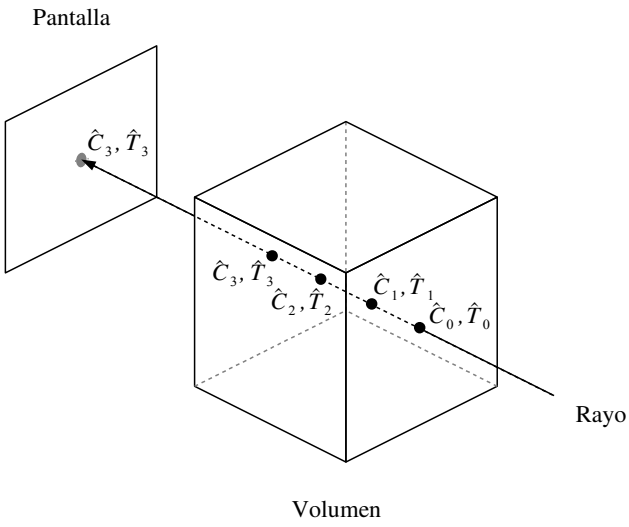


Figura 1.18: Composición de atrás hacia adelante para un único rayo de luz.

función de transferencia asigna propiedades ópticas a estos datos. La selección de la función de transferencia a utilizar depende de las diferentes estructuras que se quieran identificar en la imagen.

Aunque las implementaciones más complejas de la función de transferencia permiten automatizar tareas como la identificación de patrones, en el caso más sencillo una función de transferencia asigna un valor de color y de transparencia a cada posible valor de los datos del volumen. En general, muchas soluciones de renderizado de volúmenes permiten al usuario modificar en tiempo real la función de transferencia, ajustando las curvas de color y de transparencia, y visualizando inmediatamente los resultados por pantalla.

La función de transferencia suele implementarse como una tabla de búsqueda de tamaño fijo que se almacena como textura unidimensional en la memoria de la GPU. El valor de la función de transferencia se calcula dentro de un *fragment shader*. Para cada fragmento se obtiene el valor interpolado del conjunto de datos en el punto correspondiente al fragmento. El *fragment shader* utiliza este valor como índice para localizar en la tabla de la función de transferencia los valores correspondientes de color y transparencia.

1.6. Medidas de calidad

Para poder evaluar de forma cuantitativa la calidad de los resultados obtenidos por los algoritmos de segmentación y de compresión de volúmenes utilizados en visualización científica se hace uso de una serie de coeficientes o índices establecidos desde hace años en la literatura. En esta sección presentamos los índices de calidad que emplearemos en los siguientes capítulos para medir la calidad de los métodos de segmentación y compresión presentados en los siguientes capítulos.

1.6.1. Coeficiente Dice

El coeficiente Dice [59] es un índice que se aplica comúnmente para comparar resultados obtenidos por un método de segmentación con un resultado de referencia (típicamente denominado *ground truth*). Sean dos regiones segmentadas Ω_1 y Ω_2, una de ellas la obtenida por el algoritmo de segmentación y la otra la segmentación de referencia. El valor del coeficiente se calcula como:

$$d(\Omega_1,\Omega_2) = \frac{2\cdot|\Omega_1\cap\Omega_2|}{|\Omega_1|+|\Omega_2|}, \tag{1.82}$$

donde $|\cdot|$ es el tamaño de la región en píxeles o vóxeles.

El coeficiente Dice toma valores entre 0 y 1. Se trata de un caso especial del estadístico *kappa* (κ) [192]. Los valores de $\kappa > 0{,}6$ suelen indicar un acuerdo sustancial entre dos muestras [107]. Sin embargo, lo habitual es encontrar en la literatura valores del coeficiente Dice iguales o superiores a 0,9 como representativos de una calidad de segmentación deseable.

En los capítulos 3 y 4 utilizaremos este coeficiente para evaluar los resultados obtenidos por los métodos de segmentación presentados.

1.6.2. Error cuadrático medio y relación señal a ruido de pico

El error cuadrático medio (en inglés MSE, *mean squared error*) es una de las medidas de referencia para evaluar el rendimiento en el campo de procesamiento de señal y, por tanto, también en el campo de procesamiento de imágenes y volúmenes [182]. El MSE describe la similaridad/fidelidad entre dos señales a partir del grado de error o distorsión entre las mismas.

Sea **x** la señal original e **y** la señal resultado de procesar los valores contenidos en **x**. Si **x** e **y** son dos señales discretas de longitud finita n, entonces el valor de MSE se calcula como:

$$MSE(\mathbf{x},\mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2. \tag{1.83}$$

Así, en el campo de la compresión de imágenes y volúmenes, donde la compresión implica una cierta pérdida de información, la señal **x** hace referencia al volumen original y la señal **y** es el resultado de comprimir el volumen original y luego restaurarlo.

En la literatura es habitual convertir el MSE en relación señal a ruido de pico (en inglés PSNR, *peak signal-to-noise ratio*), mediante la siguiente fórmula:

$$PSNR(\mathbf{x},\mathbf{y}) = 10\log_{10}\frac{L^2}{MSE(\mathbf{x},\mathbf{y})}, \tag{1.84}$$

donde L es el rango dinámico de los valores de intensidad posibles para cada píxel o vóxel de la imagen o el volumen, respectivamente. Si el rango dinámico está limitado por b bits por cada píxel o vóxel, entonces $L = 2^b - 1$. El resultado de calcular el PSNR se expresa en la escala logarítmica de decibelios (dB).

Aunque el PSNR no ofrece más información que el MSE, permite comparar los resultados obtenidos utilizando imágenes o volúmenes con valores de intensidad de diferentes rangos dinámicos. En general, valores de PSNR de entre 60 y 80 dB se suelen considerar aceptables.

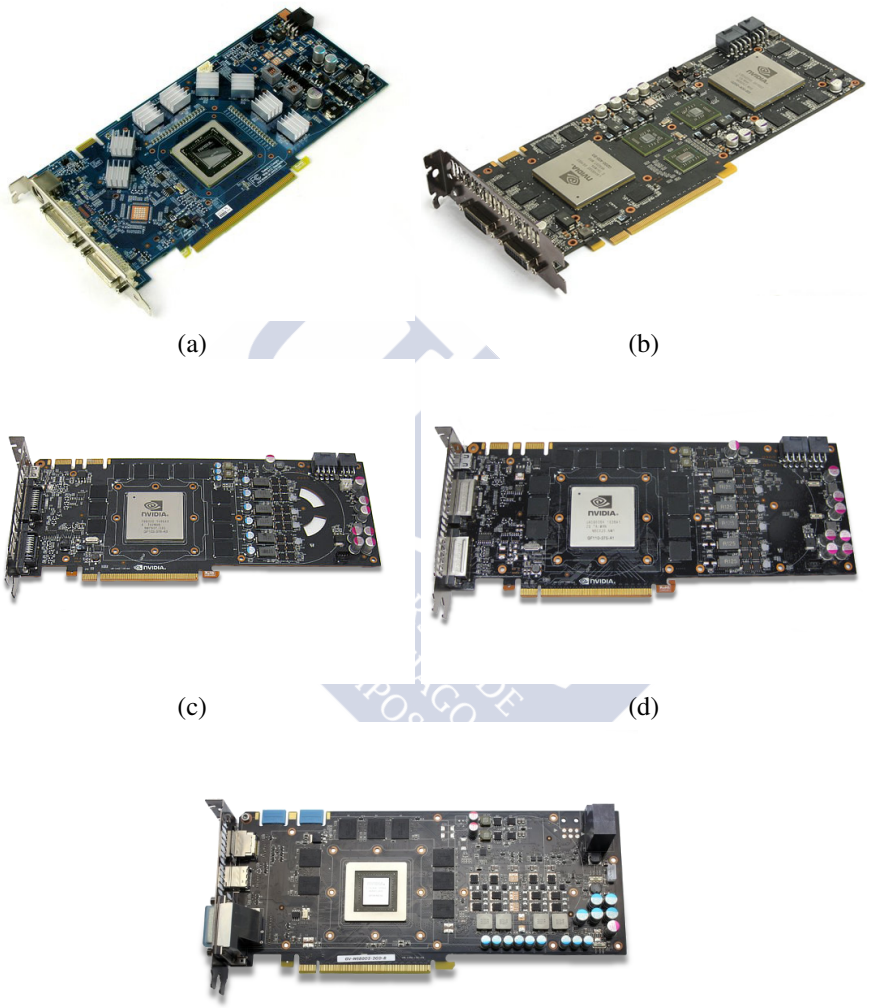
En el capítulo 5 presentaremos un esquema de visualización de volúmenes comprimidos en GPU y utilizaremos el error cuadrático medio y la relación señal a ruido pico para evaluar las pérdidas de información debidas a la compresión.

1.7. Recursos utilizados en este trabajo

En los siguientes capítulos presentaremos varias implementaciones de segmentación y visualización de volúmenes que han sido desarrolladas para esta Tesis de Doctorado. El trabajo cubre varios años de desarrollo, a lo largo de los cuales se han sucedido diversas generaciones de GPU. En esta sección presentamos las principales características de las GPU y los conjuntos de datos utilizados para evaluar el rendimiento y validar nuestras implementaciones.

1.7.1. Modelos de GPU

La figura 1.19 muestra fotografías de las cinco GPU utilizadas en este trabajo. En la tabla 1.1 se muestran sus principales características.



(a) (b)

(c) (d)

(e)

Figura 1.19: Fotografías de las GPU empleadas en este trabajo. De izquierda a derecha y de arriba a abajo: (a) 9800 GT, (b) GTX 295, (c) GTX 480, (d) GTX 580 y (e) GTX 680.

	9800 GT	GTX 295	GTX 480	GTX 580	GTX 680
Arquitectura	G92	GT200	GF100	GF110	GK104
Capacidad computacional	1.1	1.3	2.0	2.0	3.0
Núm. multiprocesadores	14	30	15	8	8
Núm. total núcleos	112	240	480	512	1536
Máx. hilos / bloque	512	512	1024	1024	1024
Máx. hilos / multiproc.	768	768	1536	1536	2048
Máx. bloques / multiproc.	8	8	8	8	16
Mem. compart. / Caché L1	16 kB	16 kB	48 kB	48 kB	48 kB
Caché L2	–	–	768 kB	768 kB	512 kB
Memoria	1024 MB	896 MB	1536 MB	1536 MB	2048 MB
Ancho de banda	70,4 GB/s	111,9 GB/s	177 GB/s	192,4 GB/s	192,2 GB/s
GFLOPS	705	895	1345	1581	3090

Tabla 1.1: Características de las GPU usadas durante este trabajo.

La 9800 GT es una GPU que pertenece a la serie 9 de GeForce, y es similar a la 8800 GT, la primera GPU con arquitectura CUDA (véase sección 1.2.4), pero utiliza un tamaño de puerta más pequeño (de 65 nm a 55 nm).

La GTX 295 pertenece a la serie 200, la primera serie que introdujo la arquitectura Tesla. Comparada con la 9800 GT, este modelo presenta una mejora destacable en casi todas las características. Además, la GTX 295 es una GPU dual que contiene dos chips separados, cada uno con una potencia similar a la GTX 280. Para obtener las prestaciones totales de la tarjeta hay que multiplicar por dos las características que aparecen en la tabla (exceptuando la memoria compartida por multiprocesador).

La GTX 480 pertenece a la serie 400, con arquitectura Fermi. Esta arquitectura introduce una jerarquía de cachés de dos niveles. El primer nivel de caché está compartido con el espacio dedicado a la memoria compartida, que se ha incrementado de 16 a 48 kB. El número de núcleos y el ancho de banda a memoria también experimentan una mejora importante respecto a la arquitectura previa.

La GTX 580 pertenece a la serie 500, una evolución dentro de la arquitectura Fermi. Aunque el número de multiprocesadores se reduce casi a la mitad, el número de núcleos total aumenta de forma significativa, y el ancho de banda y rendimiento en GFLOPS son superiores.

Por último, la GTX 680 pertenece a la serie 600, con la arquitectura Kepler, que incorpora importantes cambios respecto a sus antecesoras. El número de núcleos de esta GPU se multi-

plica por dos respecto a la 580, con lo que se consigue doblar el rendimiento pico en GFLOPS de la tarjeta (aunque manteniendo el mismo ancho de banda en los accesos a memoria).

1.7.2. Conjuntos de datos

En este trabajo hemos utilizado los siguientes conjuntos de datos:

- Un volumen artificial que contiene un cubo en su interior, para el cual generamos varias versiones escaladas a diferente tamaño. Los vóxeles en el interior del cubo tienen el valor de gris más alto, y el resto de vóxeles que pertenecen al fondo tienen un valor de cero. Hemos denominado a este conjunto de datos `cube`, y se muestra en la figura 1.20.

- Un volumen con datos reales extraídos de un volumen CT de una rodilla. Los vóxeles del tejido óseo presentan valores de gris muy bajos, similares a los del color de fondo del volumen. El tejido cartilaginoso que rodea algunas partes del hueso, así como algunas fibras musculares, presentan también valores de gris similares. Hemos denominado a este conjunto de datos `knee`, y se muestra en la figura 1.21.

- Dos volúmenes descargados de la BrainWeb Simulated Brain Database [2, 46, 50, 95, 96]. Esta base de datos contiene un conjunto de volúmenes de datos MRI realistas producidos con un simulador MRI y que también permite generar simulaciones personalizadas configurando una gran variedad de parámetros. Con el nombre de `brainweb-1` denominamos a un volumen de tamaño $181 \times 217 \times 181$ perteneciente a un modelo anatómico del cerebro sin lesiones, modalidad T1, grosor de lámina de 1 mm, sin ruido y con un 20 % de no homogeneidad en los valores de gris. Con `brainweb-2` denominamos a un volumen de tamaño $256 \times 256 \times 181$ que está disponible para descarga desde la misma *web* y que fue generado a partir de 20 modelos diferentes de cerebros normales. Ambos conjuntos de datos se muestran en la figura 1.22.

- Un conjunto de cuatro volúmenes de vasos sanguíneos con alto contraste que presentaban casos apreciables de aneurismas. Aunque los volúmenes son de diferentes tamaños, todos utilizan 2 bytes por vóxel. Denominamos a este conjunto de datos `vessels`. Los cuatro volúmenes se muestran en la figura 1.23.

- Un volumen de un modelo de plástico de una cabeza humana obtenido a partir de una tomografía computerizada. Denominamos a este conjunto de datos `modelhead` (figura 1.24).

Volumen	Dimensiones	Bytes / vóxel	Tamaño
knee	$281 \times 389 \times 104$	2	22 MB
brainweb-1	$181 \times 217 \times 181$	1	7 MB
brainweb-2	$256 \times 256 \times 181$	2	23 MB
vessels-1	$160 \times 161 \times 226$	2	11 MB
vessels-2	$181 \times 176 \times 204$	2	12 MB
vessels-3	$146 \times 174 \times 255$	2	12 MB
vessels-4	$182 \times 133 \times 119$	2	5 MB
modelhead	$512 \times 512 \times 348$	2	174 MB
realhead	$160 \times 512 \times 512$	2	80 MB
oasis-n	$176 \times 208 \times 176$	2	12 MB
chest	$512 \times 512 \times 355$	1	89 MB
kidneys	$512 \times 512 \times 256$	1	64 MB
abdomen	$484 \times 364 \times 164$	1	27 MB

Tabla 1.2: Propiedades de los volúmenes usados durante este trabajo.

– Un volumen obtenido a partir de una imagen MRI de una cabeza humana. Denominamos a este conjunto de datos `realhead` (figura 1.25).

– Cuatro volúmenes MRI de cerebros obtenidos de la base de datos Oasis [12]. Todos los volúmenes, aunque se corresponden a cerebros diferentes, presentan las mismas características de tamaño y tipo de datos, y aparecen con la denominación `oasis`, seguida de un número que permite identificarlos. La figura 1.26 muestra el volumen etiquetado como `oasis-1`.

– Un conjunto de volúmenes obtenidos de la base de datos de Osirix [15]. Estos volúmenes se corresponden con tomografías computerizadas de varias partes del cuerpo humano, concretamente, pecho, abdomen y cintura. Los volúmenes aparecen etiquetados en la base de datos como AGECANOMIX, MECANIX y MACOESSIX. En este trabajo, aparecen con las denominaciones `chest`, `kidneys` y `abdomen`. La figura 1.27 muestra los tres volúmenes.

La tabla 1.2 muestra el las dimensiones, el número de bytes por vóxel, y el tamaño total de los conjuntos de datos utilizados.

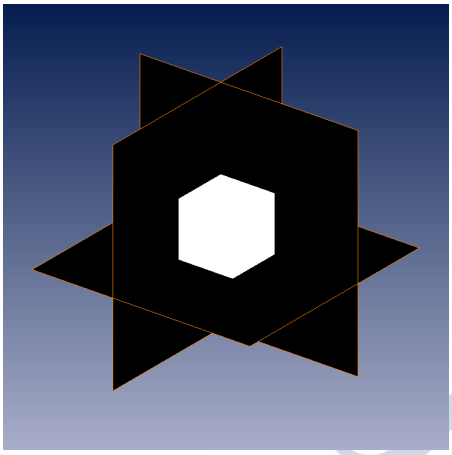


Figura 1.20: Láminas ortogonales obtenidas a partir del volumen `cube`.

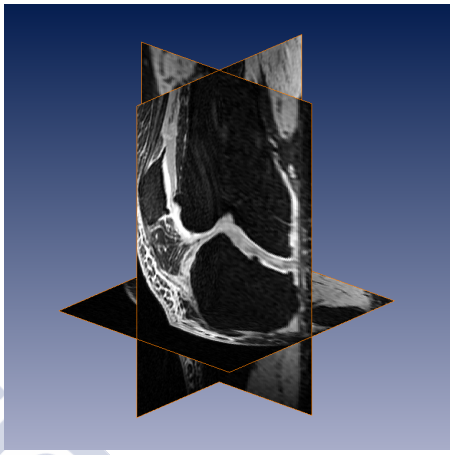
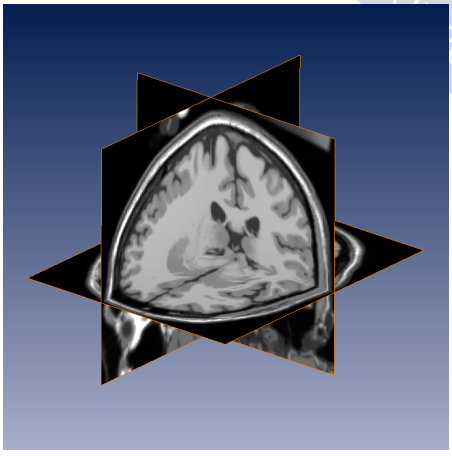
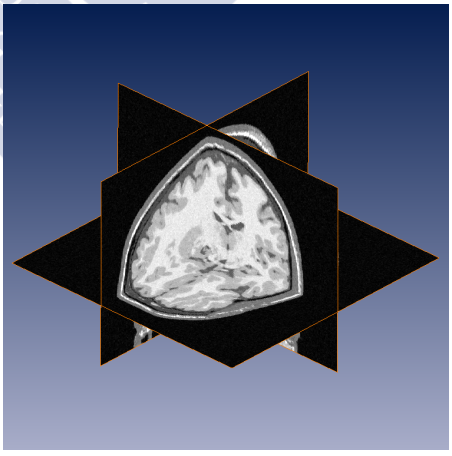


Figura 1.21: Láminas ortogonales obtenidas a partir del volumen `knee`.



(a)



(b)

Figura 1.22: Láminas ortogonales obtenidas a partir de los volúmenes (a) `brainweb-1` y (b) `brainweb-2`.

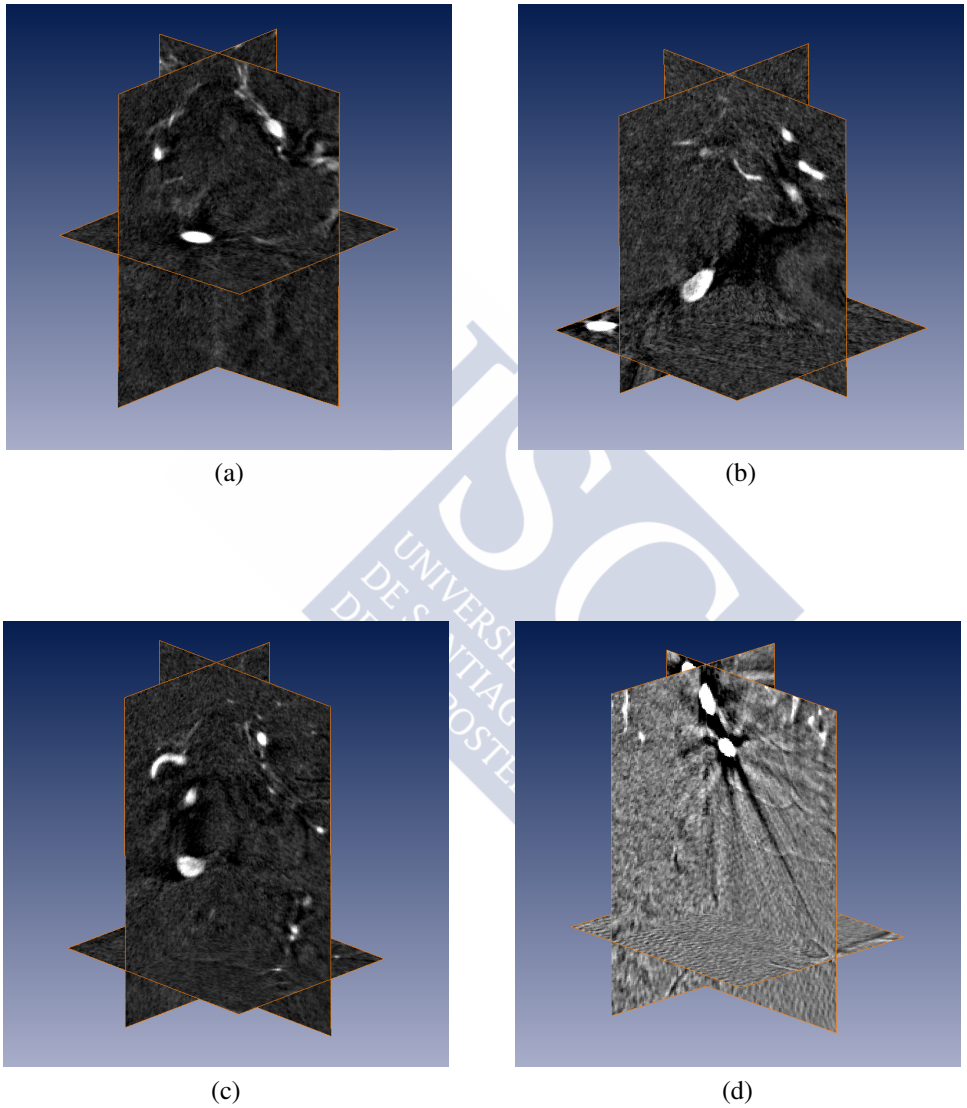


Figura 1.23: Láminas ortogonales obtenidas a partir de los volúmenes (a) `vessels-1`, (b) `vessels-2`, (c) `vessels-3` y (d) `vessels-4`.

Figura 1.24: Láminas ortogonales obtenidas a partir del volumen `modelhead`.

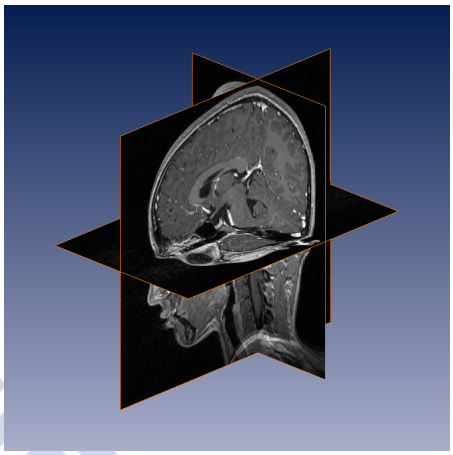


Figura 1.25: Láminas ortogonales obtenidas a partir del volumen `realhead`.

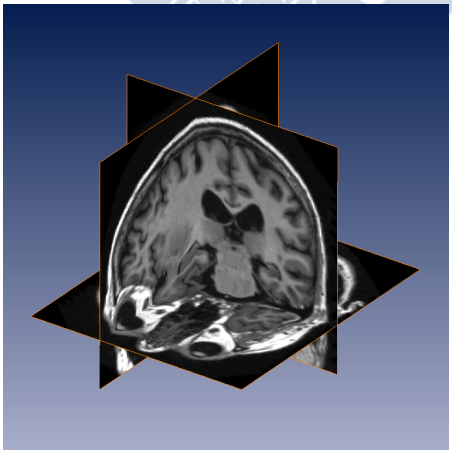
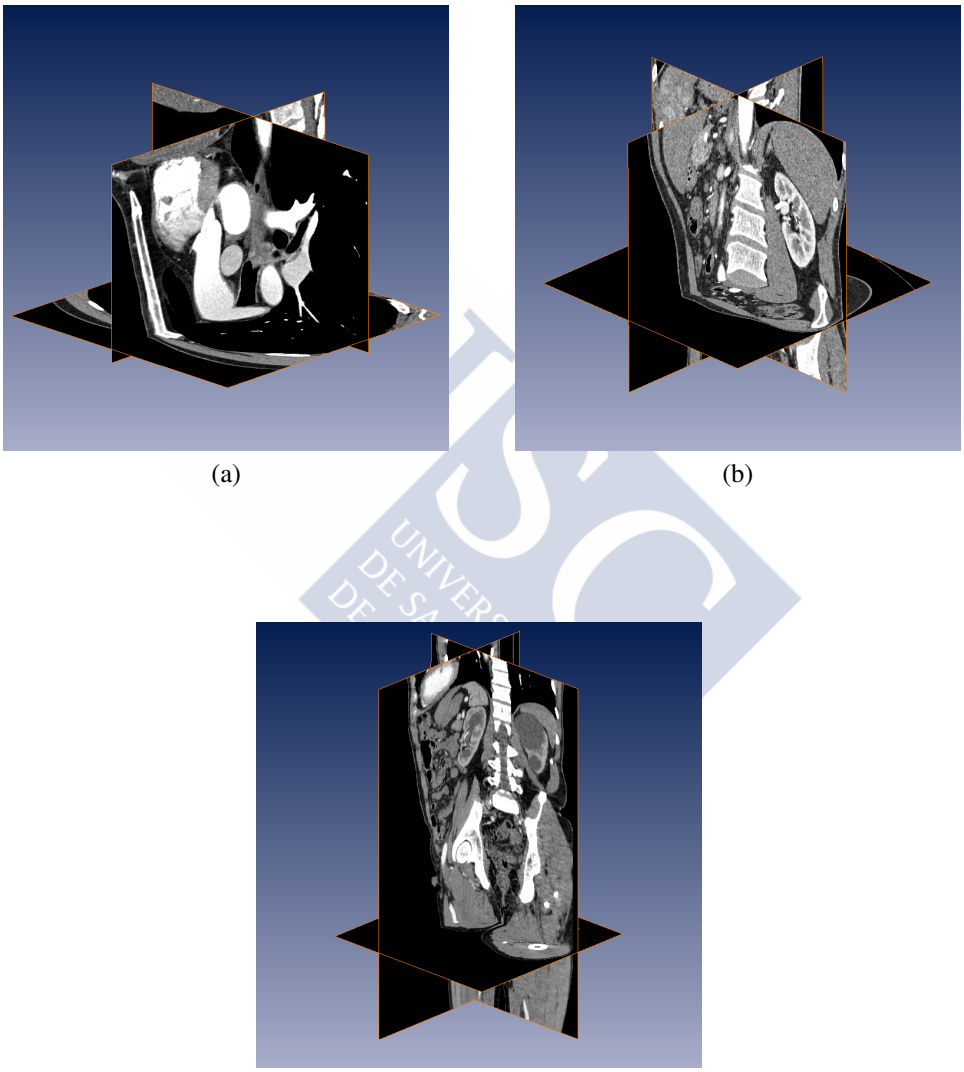


Figura 1.26: Láminas ortogonales obtenidas a partir del volumen `oasis-1`.



(a) (b)

(c)

Figura 1.27: Láminas ortogonales obtenidas a partir de los volúmenes (a) `chest`, (b) `kidneys` y (c) `abdomen`.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Capítulo 2

Técnicas generales para la programación en GPU

2.1. Introducción

En este capítulo presentamos una serie de técnicas básicas para la explotación del paralelismo en las tarjetas gráficas. De entre estas técnicas, haremos especial énfasis en la técnica "divide y fusiona", una extensión del patrón de paralelismo "divide y vencerás" que hemos adaptado para su aplicación en GPU. Mostraremos cómo este patrón permite implementar de forma eficiente métodos de resolución de sistemas tridiagonales y acelerar el cálculo de la transformada *wavelet*.

En el campo de la programación paralela, la tarea de un programador consiste en identificar la concurrencia presente en el problema que se pretende resolver, estructurar el algoritmo para que la concurrencia pueda ser explotada, e implementar la solución utilizando un entorno de programación adecuado. La concurrencia está presente en un problema computacional cuando este puede descomponerse en subproblemas que pueden ser resueltos al mismo tiempo y de forma segura [117].

En el ámbito de desarrollo de *software*, los patrones de diseño proporcionan soluciones reutilizables a problemas recurrentes. En el caso más específico de la programación paralela también se han desarrollado una serie de patrones paralelos que son de especial relevancia para los programadores que desarrollan aplicaciones de propósito general [17, 18, 117].

En este capítulo introducimos algunos de los patrones de diseño paralelos más adecuados para la programación en GPU. Hemos utilizado estos patrones para el desarrollo de las soluciones presentadas en este capítulo y los siguientes. El uso de patrones paralelos nos ha permitido realizar implementaciones óptimas de los problemas abordados, aplicando estrategias de optimización específicas de la arquitecturas GPU (reducción del número de hilos, uso eficiente de la jerarquía de memoria, etc). Como ya hemos avanzado, en este capítulo nos centraremos en el patrón paralelo "divide y fusiona", que resuelve el problema de dividir en bloques independientes las operaciones de aquellos algoritmos multinivel donde las dependencias no permiten realizar una división directa.

Este capítulo está dividido en las siguientes secciones. La sección 2.2 enumera algunos patrones paralelos para el desarrollo de aplicaciones en GPU. En la sección 2.3 presentamos el patrón paralelo "divide y fusiona", y las secciones 2.5 y 2.6 contienen aplicaciones de este patrón en el contexto de la resolución de sistemas tridiagonales de ecuaciones lineales y la transformada *wavelet*. Por último, la sección 2.7 finaliza con las conclusiones.

2.2. Patrones de paralelismo en la programación en GPU

Esta sección describe brevemente patrones de paralelismo usados comúnmente en la programación en GPU. Para una descripción más completa y detallada de cada uno de los patrones, pueden consultarse las referencias [51, 117].

2.2.1. Patrón de paralelismo de bucle

Una gran mayoría de los programas utilizados en aplicaciones científicas y de ingeniería están basadas en bucles, es decir, pueden ser expresados mediante estructuras iterativas. Los bucles contienen porciones de código que se repiten múltiples veces durante la ejecución del programa.

Podemos clasificar los bucles en dos grandes grupos atendiendo a la condición de si las operaciones de una iteración dependen de valores calculados en iteraciones previas o no. Estas dependencias entre iteraciones complican la implementación en paralelo del algoritmo, por lo que es deseable eliminar tantas como sea posible. Cuando no es posible eliminarlas, el bucle suele dividirse en varios segmentos, cuyas tareas sí es posible paralelizar, que se van ejecutando en un orden estricto.

Cuando no existen dependencias entre las iteraciones de un bucle, el trabajo de paralelización consiste en decidir cómo repartir las distintas iteraciones entre los procesadores disponibles. Esta partición debe hacerse tratando de minimizar la comunicación entre los procesadores y maximizando el uso de los recursos. En el caso de las GPU, este patrón proporciona un buen rendimiento cuando las diferentes iteraciones pueden repartirse entre bloques de hilos de forma que todos los multiprocesadores permanezcan ocupados, y las tareas que se realizan en cada iteración son computacionalmente intensivas.

2.2.2. Patrón de divergencia/unión

Este patrón es habitual en multitud de programas donde solo tareas muy específicas son paralelizables. En estas aplicaciones, el código es ejecutado en serie hasta que llega al inicio de una sección paralela. A partir de esta sección, el trabajo se distribuye (o "diverge") entre los procesadores disponibles, que ejecutan las tareas en paralelo y de forma independiente. Cuando finalizan todas las tareas, los diferentes hilos de trabajo convergen (o "se unen") de nuevo en un único hilo que continúa su ejecución en serie.

Este patrón está indicado para aquellas aplicaciones donde el número de tareas concurrentes varía durante la ejecución del programa. Un ejemplo paradigmático es la exploración en paralelo de una estructura en forma árbol, donde el análisis de un nodo puede dar lugar al inicio de nuevos hilos que profundicen en la estructura. Las relaciones que se establecen entre este tipo de tareas impiden la utilización de estructuras de control sencillas, como los bucles paralelos.

En GPU, y más concretamente en CUDA, desde la aparición de la arquitectura Kepler la implementación del patrón de divergencia/unión se lleva a cabo mediante lo que se conoce como "paralelismo dinámico", que permite realizar llamadas a *kernels* desde el interior de otro *kernel*, añadiendo nuevos hilos y bloques a los iniciales. En arquitecturas previas, la implementación directa de este patrón no era posible, ya que el número de hilos y bloques debía permanecer constante durante la ejecución del *kernel*. No obstante, como aproximación a este patrón, algunas soluciones, aprovechando los mecanismos de sincronización disponibles, coordinan la ejecución de los hilos dentro de un bloque de forma que algunos de ellos permanecen ociosos hasta que son necesarios. Aunque los hilos ociosos ocupan recursos y pueden limitar el rendimiento global, no consumen tiempo de ejecución y permiten la utilización de la memoria compartida.

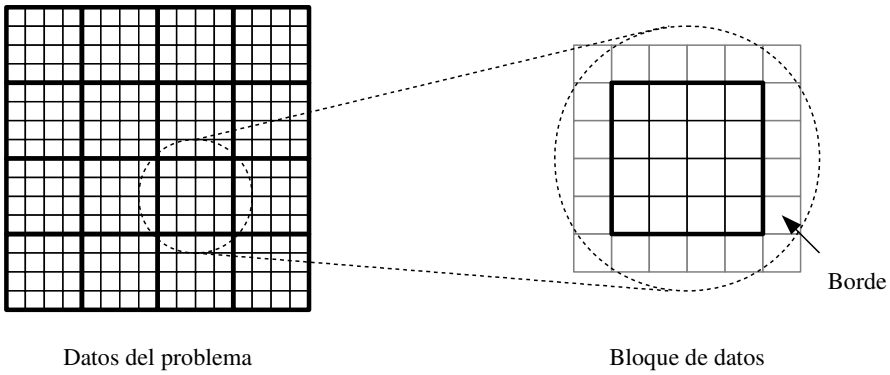


Figura 2.1: Ejemplo de descomposición geométrica para bloques de tamaño 4×4 con un borde de grosor 1.

2.2.3. Patrón de descomposición geométrica

Muchos problemas pueden plantearse como una malla de tareas a resolver o de datos a procesar que puede tener una, dos o tres dimensiones. El modelo de programación CUDA soporta mallas de bloques de hilos de hasta tres dimensiones, por lo que en muchos casos es posible mapear de forma intuitiva los diferentes bloques de tareas o datos en los que se puede dividir el problema a bloques de hilos.

Muchos de estos algoritmos tienen un patrón de comunicaciones en el que, para computar el resultado correspondiente a un bloque, se necesita no solamente el dato contenido en dicho bloque si no también los datos contenidos en bloques vecinos. Este es el caso, por ejemplo, de los algoritmos de tipo "autómata celular" y de los basados en "diferencias finitas". En estos casos es necesario replicar datos no locales al bloque, lo que da lugar al solapamiento de datos entre bloques. A esta réplica se le denomina borde "fantasma" o *apron* [140].

La figura 2.1 muestra un ejemplo de uso de este patrón. Una malla de datos de dos dimensiones se ha divido en bloques de tamaño 4×4. Estos bloques podrían, por ejemplo, ser asignados de forma intuitiva a bloques de hilos en GPU de tamaño 4×4. A la derecha, se muestra un bloque de datos desde el punto de vista de su bloque de hilos asociado. Debido al patrón de accesos del algoritmo, además del correspondiente bloque de datos, cada bloque de hilos debe almacenar y acceder a un borde que contiene los datos inmediatamente adyacentes al bloque.

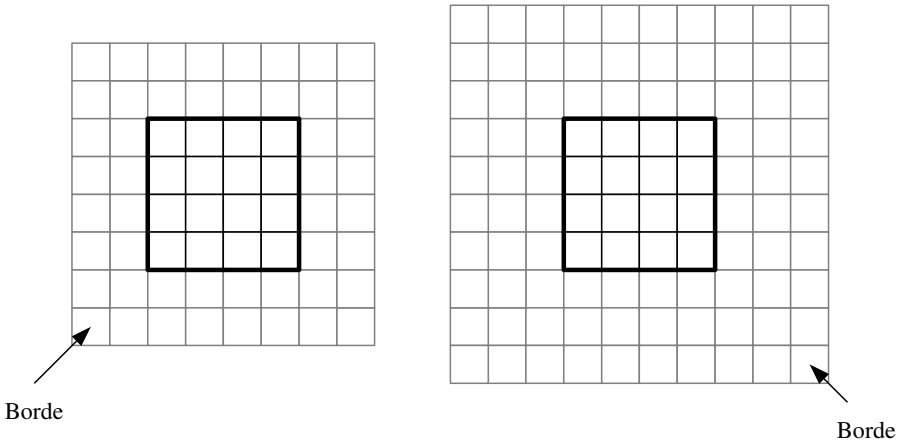


Figura 2.2: Ejemplos de un bloques de tamaño 4×4 rodeados por un borde de grosor 2 y 3.

Este patrón de accesos puede generalizarse si consideramos algoritmos donde para computar el resultado correspodiente a un dato se necesitan datos localizados a distancia de 1, 2 o más del dato original. Las operaciones de filtrado y convolución son un claro ejemplo donde se produce este tipo de situación. En estos casos, para dar soporte a este tipo de operaciones es necesario ampliar el tamaño del borde. La figura 2.2 muestra ejemplos de bloque de datos con bordes de grosor 2 y 3. Cuando las distancias son más grandes, el tamaño del borde se vuelve rápidamente intratable, es decir, la sobrecarga asociada a gestionar el borde es muy superior al coste de gestionar los propios datos del bloque. Para resolver este problema, en la sección 2.3 presentamos el patrón "divide y fusiona".

Otro punto que es importante considerar es el paralelismo a nivel de instrucción y cómo puede ser explotado: es más eficiente ejecutar varias operaciones seguidas y, luego acceder a memoria, que intercalar los accesos a memoria entre las operaciones. En general, debe alcanzarse un equilibrio entre el número de datos procesados por cada hilo y la cantidad de recursos necesarios por el bloque de hilos, ya que esto puede repercutir negativamente en el número de bloques que puede ejecutar la GPU en paralelo.

2.2.4. Patrón "divide y vencerás"

La estrategia de "divide y vencerás", utilizada en multitud de algoritmos secuenciales, consiste en dividir el problema en subproblemas más pequeños, y por último juntar las soluciones de cada subproblema para calcular la solución final. En algunos casos los subproblemas pueden ser resueltos directamente, por lo que en este sentido el patrón sería similar al de descomposición geométrica. En otros casos es necesario aplicar de nuevo a cada subproblema la estrategia de "divide y vencerás", lo que da lugar a una estructura recursiva.

La mayoría de los algoritmos recursivos pueden ser representados utilizando un modelo iterativo, lo que facilita su implementación en GPU. La arquitectura Fermi soporta llamadas recursivas en el código de los *kernels*, que se implementan de forma similar a como lo hacen en la arquitectura CPU. Así, cada llamada recursiva implica añadir nuevos parámetros y variables locales a la pila de llamadas, que se almacena en memoria global. Aunque existe una caché que optimiza el rendimiento de esta memoria, la latencia de acceso a estos datos siempre será superior que la latencia de los registros. Por tanto, el uso de soluciones iterativas proporcionará, por lo general, siempre mejores resultados que la utilización de llamadas recursivas en términos de rendimiento, con la ventaja añadida de compatibilidad con un rango mayor de GPU (en el caso de NVIDIA, solo las GPU con capacidad computacional 2.0 o superior soportan funciones recursivas).

2.3. Patrón de paralelismo "divide y fusiona"

En esta sección describimos el patrón de paralelismo "divide y fusiona". Este patrón presenta características similares a los patrones de descomposición geométrica y "divide y vencerás". De nuevo, se busca dividir el problema en subproblemas que puedan ser resueltos en paralelo, ya sea de forma directa o mediante algún método recursivo. Algunos problemas presentan un esquema de dependencias que dificulta la partición del flujo de trabajo en bloques que se puedan procesar de forma independiente y en paralelo. Este esquema es bastante común en multitud de algoritmos multinivel, como se verá en las siguientes secciones. El patrón "divide y fusiona" permite resolver este tipo de problemas de forma eficiente. Esta técnica fue propuesta inicialmente en [89] en el contexto del diseño de arquitecturas de aplicación específica [121]; en [30] presentamos una aplicación de esta técnica a la programación en GPU, que tratamos a continuación.

2.3.1. Contexto de aplicación

Muchos algoritmos que operan sobre vectores o matrices leen datos de memoria, los procesan, y almacenan los resultados parciales que se generan en las operaciones intermedias en sus posiciones originales. El cálculo de los resultados requiere a menudo no solo los datos que son reescritos en la misma posición, sino también aquellos datos localizados en posiciones cercanas [142]. Algunos de estos algoritmos tienen un esquema de computación multinivel, es decir, están estructurados en un conjunto de etapas donde se calculan resultados parciales hasta llegar a la solución final. En estos algoritmos, la computación de los sucesivos niveles necesita con frecuencia datos de entrada de posiciones cada vez más distantes, o se utilizan secuencias de datos muestreados de niveles previos. Así ocurre, por ejemplo, en la transformada ortogonal *wavelet* [115], y en algunos algoritmos de resolución de sistemas tridiagonales de ecuaciones lineales [29].

En los algoritmos multinivel el resultado parcial i-ésimo de un determinado nivel (x'_i) se obtiene a partir de los datos de entrada del nivel previo (x_i):

$$x'_i = f(\dots, x_{i-2a}, x_{i-a}, x_i, x_{i+a}, x_{i+2a}, \dots), \tag{2.1}$$

donde a es un parámetro que se incrementa en cada nivel y que determina la distancia donde se localiza cada dato. En la práctica, este parámetro suele crecer como una potencia de dos. Esto significa que las primeras etapas del algoritmo tienen la siguiente forma:

$$x'_i = f(\dots, x_{i-2}, x_{i-1}, x_i, x_{i+1}, x_{i+2}, \dots), \tag{2.2}$$

$$x''_i = f(\dots, x'_{i-4}, x'_{i-2}, x'_i, x'_{i+2}, x'_{i+4}, \dots), \tag{2.3}$$

$$x'''_i = f(\dots, x''_{i-8}, x''_{i-4}, x''_i, x''_{i+4}, x''_{i+8}, \dots). \tag{2.4}$$

La figura 2.3 (a) muestra un grafo de dependencias para un algoritmo multinivel. En este caso los niveles están ordenados de abajo a arriba; los cuadrados representan los diferentes datos que se procesan y las líneas que interconectan los datos representan dependencias que aparecen debido a las operaciones que se realizan en cada nivel. La figura 2.3 (b) muestra una representación simplificada del mismo grafo; usaremos este tipo de representación de forma habitual a lo largo de este capítulo.

Los algoritmos multinivel pueden ser clasificados atendiendo a dos criterios diferentes: si pueden ser o no computados *in place*, o si realizan una reducción (en inglés *decimation*) en cada nuevo nivel. En los algoritmos *in place* los resultados parciales pueden ser escritos

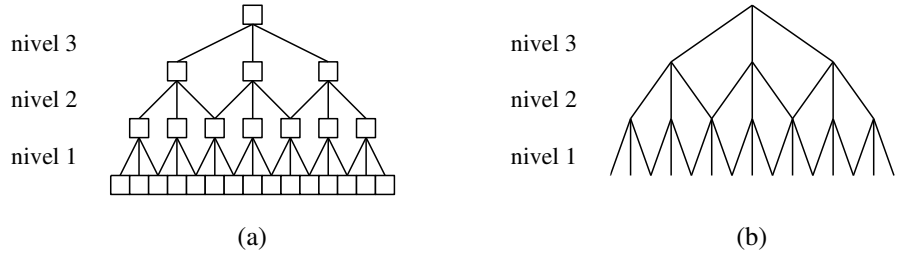


Figura 2.3: Algoritmo multinivel del que se muestra (a) el grafo de dependencias y (b) una versión simplificada del mismo grafo.

en las mismas posiciones que los datos de entrada tan pronto como son computados. En los algoritmos que no verifican esta condición los datos originales no pueden ser sobrescritos hasta que el cálculo de todos los resultados parciales ha terminado. Por tanto, estos algoritmos requieren el doble de memoria que los algoritmos *in place*. El hecho de que un algoritmo sea o no *in place* depende no solo de la naturaleza del algoritmo, sino también de cómo se organizan sus operaciones. En el caso de la programación de GPU utilizando CUDA consideramos que un algoritmo es *in place* si lo es siguiendo el modelo CUDA, es decir, si el algoritmo se puede computar *in place* usando miles de hilos que se ejecutan en paralelo en la GPU. Por ejemplo, de acuerdo a este criterio, un algoritmo que realiza el siguiente cálculo:

$$x'_i = f(x_{i-1}, x_i, x_{i+1}), \quad \text{para } i = 0, 1, 2, 3, \ldots \tag{2.5}$$

No es *in place* porque el dato de entrada x_i es necesario para obtener los resultados x'_{i-1}, x'_i y x'_{i+1}, que se calculan en hilos diferentes. Sin embargo, un algoritmo que realiza el siguiente cálculo:

$$x'_i = f(x_{i-1}, x_i, x_{i+1}), \quad \text{para } i = 0, 2, 4, 6, \ldots, \tag{2.6}$$

es *in place* dado que el dato de entrada x_i es necesario solo para el hilo que calcula el resultado x'_i.

Por otra parte, los algoritmos sin reducción se caracterizan por mantener constante el número de datos y cálculos en todos los niveles. En algoritmos con reducción se produce un nuevo muestreo de la secuencia de datos en cada nuevo nivel con una distancia entre muestras diferentes. Normalmente la distancia se incrementa en cada nuevo nivel en un factor de dos, es decir, que solo la mitad de los datos de un determinado nivel progresan hacia el siguiente.

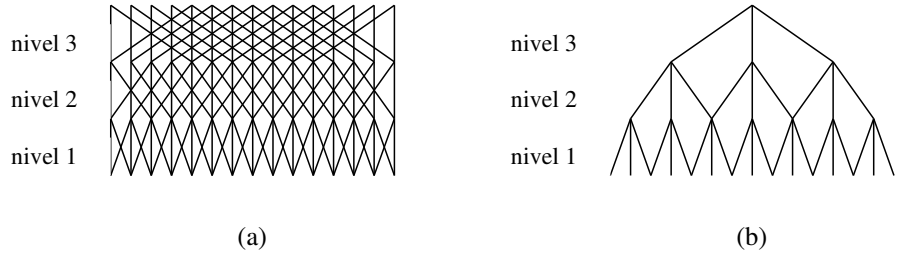


Figura 2.4: Grafos de dependencias para (a) un algoritmo multinivel no *in place* y sin reducción y (b) un algoritmo multinivel *in place* con reducción.

Si hay un número suficiente de niveles, la reducción continuará hasta que solo queda un único dato, por lo que el algoritmo tendrá una estructura de pirámide.

En la figura 2.4 se muestran los grafos de dependencias dos algoritmos diferentes. En la subfigura (a) el grafo de dependencias se corresponde con un algoritmo no *in place* y sin reducción que realiza el cálculo indicado en la ecuación (2.5). En la subfigura (b) el grafo de dependencias se corresponde con un algoritmo *in place* con reducción que realiza el cálculo de la ecuación (2.6).

2.3.2. Motivaciones para utilizar el patrón en GPU

Los principales motivos que determinan el uso de este patrón de paralelismo "divide y fusiona" son:

– Algunos algoritmos procesan un conjunto de datos sobre el que realizan varios niveles de procesamiento, reutilizando en cada nivel un nuevo muestreo de los datos, lo que incrementa la distancia entre los datos operados normalmente en un factor de dos. En estos casos el patrón de descomposición geométrica no es efectivo.

– En estos algoritmos, las operaciones de cada nivel son independientes entre sí, lo que invita a utilizar una aproximación concurrente para su implementación. Sin embargo, los diferentes niveles que componen el algoritmo deben resolverse en un orden estricto.

– Una división eficiente de las tareas de estos algoritmos debe ser capaz de ejecutar de forma eficiente varios niveles al mismo tiempo. Esta división será eficiente si evita replicar recursos y operaciones entre las tareas realizadas por los distintos procesos.

– En las GPU con arquitectura CUDA, la división de tareas en subtareas más pequeñas que pueden ser resueltas de forma independiente permite aprovechar la jerarquía de memoria y hacer uso de la memoria compartida, que ofrece una latencia mucho menor que la memoria global.

2.3.3. Descripción del patrón

En los esquemas de computación que dividen el flujo de trabajo en bloques, ya sea para ser procesados secuencialmente o por múltiples procesadores en paralelo, el cálculo de los datos cercanos a los bordes del bloque plantea ciertas dificultades. Dependiendo del tipo de arquitecturas, las técnicas con solapamiento, como la replicación de datos cercanos a los bordes del bloque, y las técnicas sin solapamiento, basadas en comunicaciones adicionales, pueden dar lugar a soluciones eficientes [72].

En algunos casos, la aplicación directa de estas técnicas en algoritmos multinivel puede ser muy costosa en términos de memoria y de comunicaciones entre los procesadores. Este problema fue estudiado en [89] para el cálculo de la transformada *wavelet*, donde, o bien los bloques contienen suficientes datos solapados para poder ser computados sin necesidad de comunicación, o bien es necesario comunicar datos después de que cada nivel haya sido computado. La primera aproximación, con solapamiento, requiere que bloques adyacentes compartan los datos en los bordes del bloque. Dado que cada bloque tienen que calcular sus propios resultados parciales para múltiples niveles, este solapamiento puede ser bastante elevado. Por ejemplo, en el caso de la transformada *wavelet* o de la resolución de sistemas tridiagonales, el solapamiento crece de forma exponencial al incrementar el nivel. En la segunda aproximación, sin solapamiento, los datos de entrada no están solapados, con lo que los requisitos de memoria son menores; sin embargo, los datos en los bordes deben ser intercambiados en cada nuevo nivel de descomposición.

En [89] se presenta una técnica para el posprocesado de datos en los bordes de los bloques denominada “divide y fusiona” (en inglés SM, *split and merge*) en el contexto de diseño de arquitecturas para calcular la transformada *wavelet*. La clave de esta técnica es que los resultados parciales calculados en cada nivel del algoritmo se pueden volver a almacenar en sus posiciones originales y la transformación puede continuar en cualquier momento siempre que se conserven estos resultados parciales. La idea está basada en el método de “solapamiento y suma” (en inglés OA, *overlap-add*) [80]. La figura 2.5 muestra un ejemplo de aplicación del método “divide y fusiona” a un algoritmo multinivel. Inicialmente, las operaciones se reparten

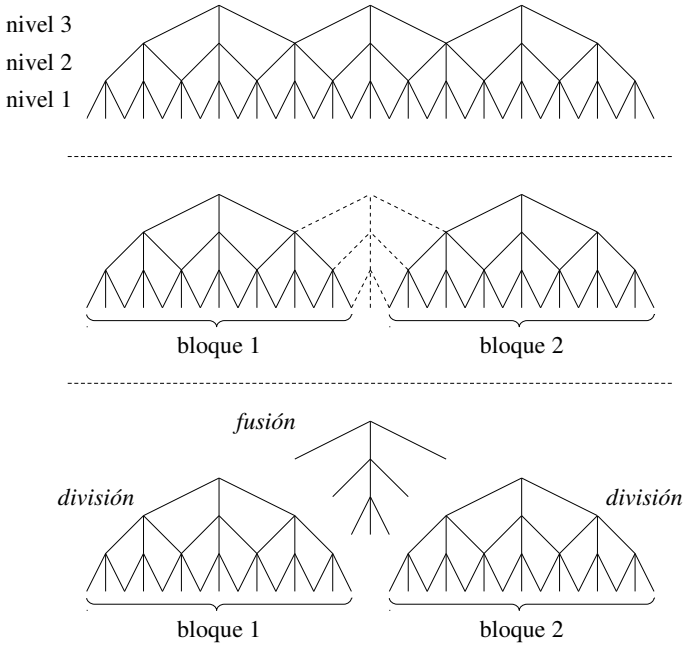


Figura 2.5: Aplicación del método "divide y fusiona" a un algoritmo multinivel.

en bloques, que pueden ser ejecutados de forma independiente durante la fase de *división* utilizando exclusivamente los datos contenidos dentro de cada bloque. Los datos que no pueden ser procesados durante esta fase debido a las dependencias entre las operaciones posponen su procesamiento a la fase de *fusión*, que se ejecuta tras la fase de división y completa la computación.

En algoritmos donde el número de niveles es alto puede ser más eficiente aplicar el método de "divide y fusiona" más de una vez. En este caso, cada fase de división y de fusión cubrirá algunos de los niveles del algoritmo, como se muestra en la figura 2.6 para un algoritmo con forma de pirámide. En esta figura los niveles aparecen ordenados de abajo a arriba, y cada trapecio y triángulo representan una sección de división y fusión, respectivamente. Cada sección contiene un conjunto de operaciones que se ejecutan de forma independiente, y el ancho del trapecio o el triángulo en cada nivel es proporcional al número de operaciones que se realizan en ese nivel. El número de niveles que se cubren en cada fase es un parámetro que puede ser ajustado para minimizar el tiempo de computación y, en general, dependerá del número de

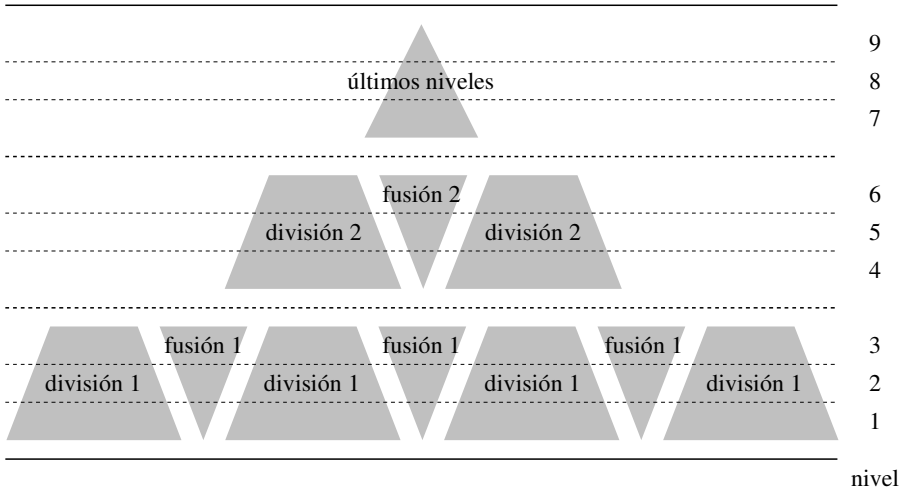


Figura 2.6: Aplicación sucesiva del método "divide y fusiona" a un algoritmo con un gran número de niveles en forma de pirámide.

operaciones y accesos a memoria que se realizan en cada nivel del algoritmo. Para cada fase de división y fusión los datos que se incluyen en cada bloque deben ser seleccionados para maximizar el número de resultados parciales que se pueden calcular dentro del bloque, a la vez que se debe evitar cargar datos que no van a ser utilizados.

En el ámbito de la programación en GPU utilizando CUDA, una implementación directa de los algoritmos multinivel suele seguir un esquema en el que cada nivel de computación se resuelve mediante una llamada a un *kernel*, de forma que habrá el mismo número de llamadas que de niveles. El problema de este tipo de implementación es que, debido a las dependencias entre las operaciones, todos los datos de entrada y todos los resultados parciales deben permanecer en la memoria global. El método "divide y fusiona" permite utilizar de forma eficiente los espacios de memoria de la GPU: cada bloque puede copiar su porción de datos a la memoria compartida, que es más rápida que la memoria global, procesar de forma independiente la división de varios niveles, y escribir los resultados parciales en la memoria global. Los datos se transfieren entre la memoria global y la compartida solo al principio y al final de cada fase de división. Todas las operaciones intermedias se realizan con los datos almacenados en la memoria compartida de cada bloque de hilos. La fase de fusión posterior,

aunque suele requerir menos operaciones que la fase de división, también puede calcularse de esta forma.

2.4. Algunas consideraciones sobre la granularidad

No existe un consenso en la literatura sobre a qué debe responder el término granularidad. En el campo de la programación paralela [117], es habitual definir la granularidad como una medida cualitativa de la proporción del tiempo dedicado en una aplicación a tareas de comunicación frente a tareas de computación real [134, 164]. En cierto modo, esta definición responde a la idea intuitiva que asocia la granularidad al "tamaño" de las tareas que se resuelven en paralelo; así, se habla de paralelismo de grano fino cuando cada una de las tareas individuales requiere un tiempo corto de ejecución, y paralelismo de grano grueso cuando cada tarea individual necesita un tiempo de ejecución considerable para ser completada. Las aplicaciones con paralelismo de grano fino ejecutan, por lo general, un gran número de tareas y sus estrategias de optimización buscan minimizar la sobrecarga asociada a la comunicación y sincronización entre tareas. Por otra parte, las aplicaciones con paralelismo de grano grueso ejecutan pocas tareas que, en ocasiones, pueden presentar problemas de balanceo de carga que penalicen el rendimiento.

En el campo de la programación en GPU la granularidad suele corresponderse con los diferentes niveles que componen la malla de hilos invocada por una aplicación paralela. Así, el nivel más grueso de granularidad se corresponde con la granularidad "de rama" (o granularidad de bloque), esto es, con los hilos que forman parte de un mismo bloque; granularidad de *warp*, esto es, grupos de hilos para los cuales la GPU ejecuta la misma instrucción; y, en el nivel más fino, granularidad de hilo, asociado a las tareas que tiene que ejecutar cada hilo [93, 133]. Esta gama de granularidades convierte a la GPU en una arquitectura muy versátil capaz de soportar diferentes niveles de granularidad, es decir, diferentes formas de dividir y repartir las tareas asociadas al problema que se quiere resolver, aunque por lo general tiende a favorecerse el paralelismo de grano fino [51].

Los patrones de paralelismo examinados en este capítulo muestran diferentes formas de organizar el flujo de trabajo para resolver un determinado problema utilizando una arquitectura paralela, como puede ser un procesador multinúcleo o una GPU. Sin embargo, establecer el tamaño de cada una de las tareas que se han de ejecutar en paralelo requiere adoptar un compromiso entre la naturaleza del problema (por ejemplo, no todos los problemas pueden

dividirse en tareas pequeñas) y las capacidades de la arquitectura (por ejemplo, en la GPU se puede utilizar paralelismo de grano fino), y, por tanto, debe decidirse para cada problema en particular.

2.5. Aplicación en la resolución de sistemas tridiagonales de ecuaciones lineales

Esta sección estudia el problema de la resolución de grandes sistemas tridiagonales de ecuaciones lineales y la aplicación de diversas técnicas de optimización para una implementación eficiente en GPU.

2.5.1. Descripción del problema

La resolución de sistemas tridiagonales de ecuaciones lineales es una tarea que aparece con frecuencia en el campo de la computación científica. Los sistemas tridiagonales están presentes en problemas relacionados con simulaciones físicas, ecuaciones diferenciales parciales, etc. En el capítulo 3 analizaremos un algoritmo de segmentación basado en conjuntos de nivel que requiere la resolución de este tipo de sistemas de ecuaciones.

El objetivo es resolver un sistema de n ecuaciones lineales $A\mathbf{x} = \mathbf{d}$, donde A es una matriz tridiagonal:

$$\begin{pmatrix} b_1 & c_1 & & 0 \\ a_2 & b_2 & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ 0 & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix}. \tag{2.7}$$

Denotaremos cada ecuación del sistema como:

$$E_i \equiv a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \tag{2.8}$$

para $i = 1, \ldots, n$, donde $x_0 = x_{n+1} = 0$.

En este capítulo presentamos varias propuestas de resolución de sistemas tridiagonales de ecuaciones lineales en la GPU. Estos algoritmos han sido escogidos porque su flujo de trabajo está compuesto de pasos independientes, lo que los hace más adecuados para su paralelización en GPU.

2.5.2. Trabajos relacionados

Uno de los algoritmos más conocidos para resolver sistemas tridiagonales es el algoritmo de Thomas [175]. Se trata de una versión simplificada de la eliminación gaussiana sin pivote para resolver sistemas de n ecuaciones de diagonal dominante en $O(n)$ pasos realizando una descomposición LU.

El algoritmo de Thomas proporciona una solución eficiente, pero es completamente secuencial y no se puede adaptar bien a las arquitecturas paralelas y vectoriales, para las que se desarrollaron nuevos algoritmos. Uno de los primeros algoritmos paralelos para la resolución de sistemas tridiagonales fue descrito en [40, 82], y se conoce comúnmente como reducción cíclica. Más adelante, se presentó el algoritmo de doblamiento recursivo en [166] como una versión paralela de la descomposición LU. Ambos algoritmos paralelos presentan un nivel de paralelismo de grano fino, donde cada procesador realiza cálculos sobre una sola ecuación del sistema. Dado que este nivel de paralelismo no es alcanzable por todas las arquitecturas paralelas, existen otras soluciones de paralelismo más grueso, como los métodos de partición presentados en [123, 180], o el método de Bondeli [36], que utiliza un planteamiento basado en el patrón de paralelismo "divide y vencerás".

La resolución de sistemas tridiagonales en la GPU es objeto de reciente estudio en la literatura. El primer algoritmo de resolución de sistemas tridiagonales implementado en GPU fue publicado en [90], al que pronto siguió una implementación en CUDA [155]. En los últimos años ha habido importantes avances en este campo: el rendimiento de varios algoritmos paralelos fue analizado en [189], donde se presentaron diversas variantes híbridas, esto es, implementaciones que combinan el uso de la CPU y la GPU. Al mismo tiempo, se presentó una nueva implementación de reducción cíclica en [70] para resolver el problema de Poisson en mallas de precisión mixta. En [56] se describe una implementación en GPU de reducción cíclica que utiliza una técnica de empaquetado de registros. Todas estas propuestas fueron diseñadas para situaciones donde cientos o miles de pequeños sistemas tridiagonales deben ser resueltos en paralelo. En [28] se explora la resolución de sistemas de gran tamaño (entre uno y diez millones de ecuaciones), proporcionando una solución híbrida que combina el uso de CPU y GPU. Otro algoritmo, consistente en dividir el sistema a resolver en sistemas más pequeños, es descrito en [57].

2.5.3. Reducción cíclica

En esta sección describimos el método de reducción cíclica para la resolución de sistemas tridiagonales de ecuaciones lineales y analizamos tres propuestas para su implementación en GPU.

Descripción

El método de reducción cíclica es ampliamente utilizado en computadores paralelos [29]. Se trata de un algoritmo multinivel, *in place* y con reducción que se desarrolla en dos fases: reducción hacia adelante, que a cada nivel procesa el sistema de entrada para generar un nuevo sistema con la mitad de incógnitas; y la fase de sustitución hacia atrás, que vuelve sobre los pasos de la reducción para calcular los valores de las incógnitas.

La fase de reducción hacia adelante se completa en $\lfloor \log_2 n \rfloor$ niveles de procesado, o pasos, donde n es el número de ecuaciones del sistema. En cada nivel $s = 0, \ldots, \lfloor \log_2 n \rfloor - 1$ se generan nuevas ecuaciones de la forma:

$$E_i^s \equiv a_i^s x_{i-2^s} + b_i^s x_i + c_i^s x_{i+2^s} = d_i^s, \tag{2.9}$$

donde $i = 2^s, \ldots, n$, con paso 2^s, y $x_{i \pm 2^s} = 0$ si $i \pm 2^s \notin [1, n]$. Los resultados obtenidos en el nivel $s-1$ se utilizan para los cálculos en el nivel s. Más específicamente, la ecuación E_i^s se calcula como:

$$E_i^s := \alpha_i^{s-1} E_{i-2^{s-1}}^{s-1} + E_i^{s-1} + \beta_i^{s-1} E_{i+2^{s-1}}^{s-1}, \tag{2.10}$$

con $\alpha_i^{s-1} = -a_i^{s-1} / b_{i-2^{s-1}}^{s-1}$ y $\beta_i^{s-1} = -c_i^{s-1} / b_{i+2^{s-1}}^{s-1}$.

Con el primer operando de la ecuación (2.10) se elimina la incógnita $x_{i-2^{s-1}}$ a la vez que se añade el nuevo término con x_{i-2^s}. De forma similar, con el último operando se elimina la incógnita $x_{i+2^{s-1}}$ a la vez que se añade la incógnita x_{i+2^s}. Como consecuencia, a cada nivel del algoritmo el número de ecuaciones se reduce en un factor de dos.

En el último nivel de la reducción hacia adelante se obtiene una única ecuación, E_i^n, con una única incógnita x_i, con $i = 2^{\lfloor \log_2 n \rfloor}$. En este punto comienza la fase de sustitución hacia atrás, donde para cada nivel $s = \lfloor \log_2 n \rfloor, \ldots, 0$ la ecuación $i = 2^s, \ldots, n$, con paso 2^{s+1}, se formula como:

$$x_i = \frac{d_i^s - a_i^s \cdot x_{i-2^s} - c_i^s \cdot x_{i+2^s}}{b_i^s}. \tag{2.11}$$

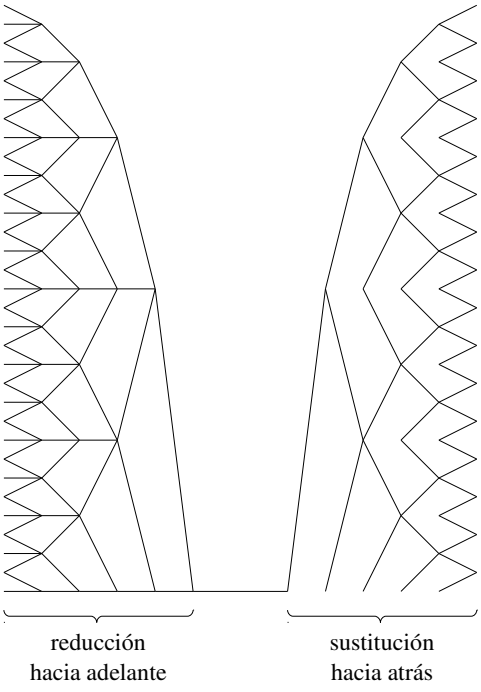


Figura 2.7: Grafo de dependencias del método de reducción cíclica para un sistema tridiagonal de 32 ecuaciones.

La figura 2.7 muestra el esquema de resolución del algoritmo de reducción cíclica para un sistema tridiagonal de 32 ecuaciones. En este caso, los diferentes niveles necesarios para la resolución de este algoritmo aparecen ordenados de izquierda a derecha. En la primera fase, reducción hacia adelante, la operación descrita en la ecuación (2.10) genera un grafo de dependencias piramidal, como el mostrado en la figura 2.3. En la segunda fase, sustitución hacia atrás, las operaciones de la ecuación (2.11) generan una estructura de dependencias diferente.

Implementación

La implementación en GPU del algoritmo de reducción cíclica almacena las ecuaciones del sistema en cinco vectores en memoria global: tres contienen los coeficientes de las

1: **para** $s = 0 \rightarrow \lfloor \log_2 n \rfloor - 1$:
2: < ejecutar un nivel de reducción ($\lfloor n/2^s \rfloor$ ecuaciones, en mem. global) >

3: **para** $s = \lfloor \log_2 n \rfloor \rightarrow 0$:
4: < ejecutar un nivel de sustitución ($\lfloor n/2^{s+1} \rfloor$ incógnitas, en mem. global) >

Figura 2.8: Pseudocódigo de una implementación directa en GPU del algoritmo de reducción cíclica.

diagonales de la matriz, uno almacena los coeficientes del término dependiente, y el último almacena las soluciones.

En la figura 2.8 se muestra el pseudocódigo de una implementación directa en GPU de este algoritmo. Las llamadas a los *kernels* que ejecutan en paralelo un nivel de las operaciones de reducción hacia adelante y sustitución hacia atrás aparecen indicadas con los signos "menor que" y "mayor que". En el patrón de dependencias (figura 2.7) se observa que, a cada nuevo nivel, los accesos a memoria se realizan a datos más distantes. Este patrón de accesos a la memoria global de la GPU da lugar a problemas de coalescencia que repercuten negativamente en el rendimiento. La distancia entre accesos a la memoria global se dobla en cada nivel de la fase de reducción hacia adelante, lo que disminuye la localidad espacial de estos accesos e incrementa el número de fallos caché en tarjetas con arquitectura Fermi o posterior (en GPU más antiguas se incrementa la tasa de serialización de los accesos).

Es posible reordenar las posiciones donde se almacenan los resultados parciales en memoria, de forma que la localidad espacial de los accesos se mantenga alta en todos los niveles. Esta estrategia de reordenación fue propuesta en [70] para la resolución de sistemas tridiagonales pequeños que pueden ser almacenados en la memoria compartida de la GPU. En este caso, la estrategia de reordenación minimiza los conflictos en los accesos a los bancos de memoria compartida, pero el esquema puede extenderse también a la memoria global.

La figura 2.9 muestra las dependencias presentes en el primer nivel de la reducción hacia adelante de un sistema de ocho ecuaciones que hemos numerado del 1 al 8. Al separar las dependencias, se observa que ya en este primer nivel los accesos a memoria no se realizan de forma consecutiva. Si reordenamos las ecuaciones de forma que las pares e impares (representados con diferentes tonos de gris) queden agrupadas, los accesos en el primer nivel se realizan a posiciones consecutivas de memoria. Al aplicar esta reordenación en todos los niveles, se mantiene la localidad espacial de los accesos.

La figura 2.10 muestra el pseudocódigo donde se aplica la estrategia de reordenación. En primer lugar, antes de iniciar la fase de reducción hacia adelante, se reordenan las ecuacio-

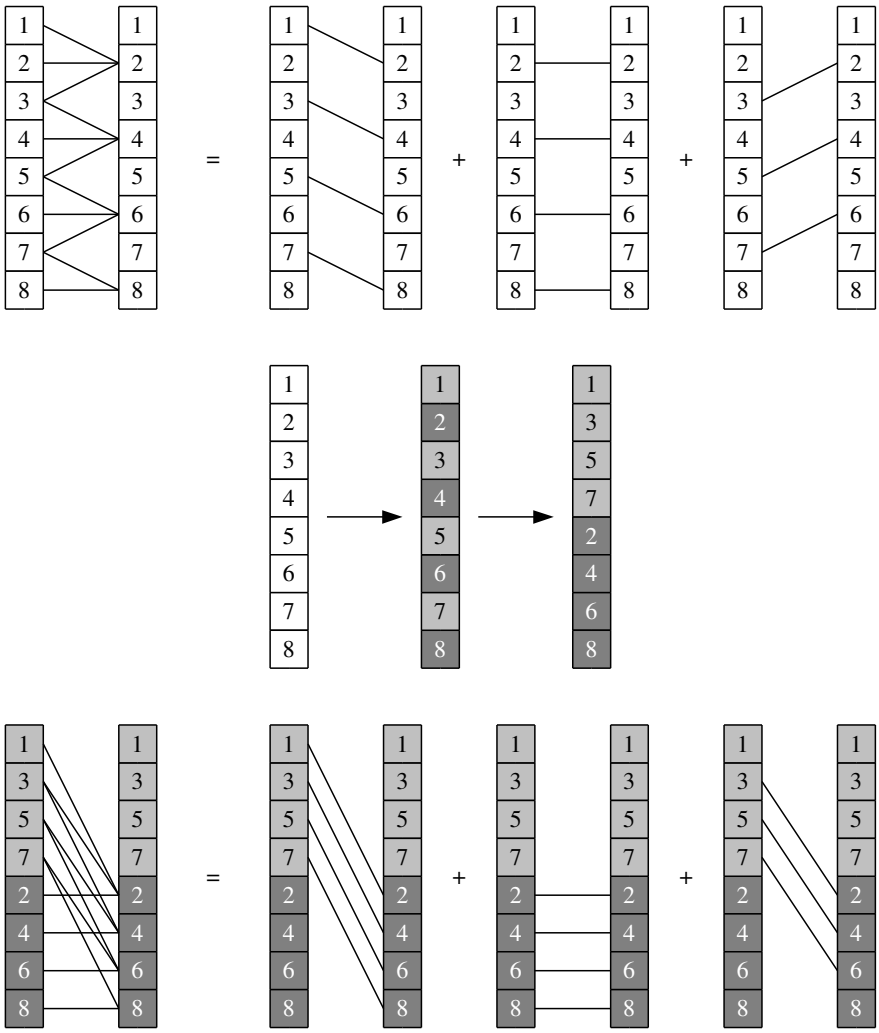


Figura 2.9: Dependencias en la aplicación del primer paso de reducción cíclica a un sistema de 8 ecuaciones, antes y después de reordenarlo.

1: < reordenar n ecuaciones (en mem. global) >

2: **para** $s = 0 \rightarrow \lfloor \log_2 n \rfloor - 1$:
3: < ejecutar un nivel de reducción con reordenación ($\lfloor n/2^s \rfloor$ ecuaciones, en mem. global) >

4: **para** $s = \lfloor \log_2 n \rfloor \rightarrow 0$:
5: < ejecutar un nivel de sustitución con reordenación ($\lfloor n/2^{s+1} \rfloor$ incógnitas, en mem. global) >

6: < restaurar orden original de las n incógnitas (en mem. global) >

Figura 2.10: Pseudocódigo de una implementación con reordenación en GPU del algoritmo de reducción cíclica.

nes agrupándolas en pares e impares. Durante la reducción hacia adelante, la reordenación se repite para las nuevas ecuaciones que se generan en cada paso. Estas nuevas ecuaciones se almacenan en nuevas posiciones en memoria con la ordenación adecuada y, como consecuencia, se incrementa la longitud del vector de ecuaciones. No obstante, la distancia entre los accesos a las ecuaciones es constante en todos los niveles. Durante la sustitución hacia atrás, se mantiene la localidad espacial de los accesos. El valor de cada incógnita se calcula a partir de los valores de otras incógnitas y de las ecuaciones almacenadas en el vector extendido. Un vector auxiliar de índices generado durante la fase de reducción hacia adelante permite conocer la posición de cada ecuación dentro del vector extendido. Por último, tras ejecutar el último paso de la sustitución hacia atrás, se aplica una última reordenación para restaurar el orden original de las incógnitas.

La figura 2.11 muestra un esquema de la implementación de reducción cíclica con reordenación para un sistema tridiagonal de 32 ecuaciones. Por sencillez, la figura no muestra las reordenaciones inicial y final, ni los vectores extendidos.

La estrategia de reordenación mejora la eficiencia de los accesos, pero el esquema de dependencias no permite repartir eficientemente el flujo de trabajo en bloques que puedan ser procesados de forma independiente en la memoria compartida. La estrategia de "divide y fusiona", descrita en la sección 2.3, soluciona este problema.

La figura 2.12 muestra el pseudocódigo de la implementación en GPU de reducción cíclica donde la técnica "divide y fusiona" se aplica a los primeros niveles de la etapa de reducción hacia adelante. En cada iteración de esta etapa se resuelven en GPU m niveles alternando una fase de división (que se ejecuta en la memoria compartida) y una fase de fusión (que se ejecuta en memoria global). Los últimos niveles de la etapa de reducción se ejecutan en memoria global. A continuación, comienza el proceso de resolución hacia atrás. Los primeros

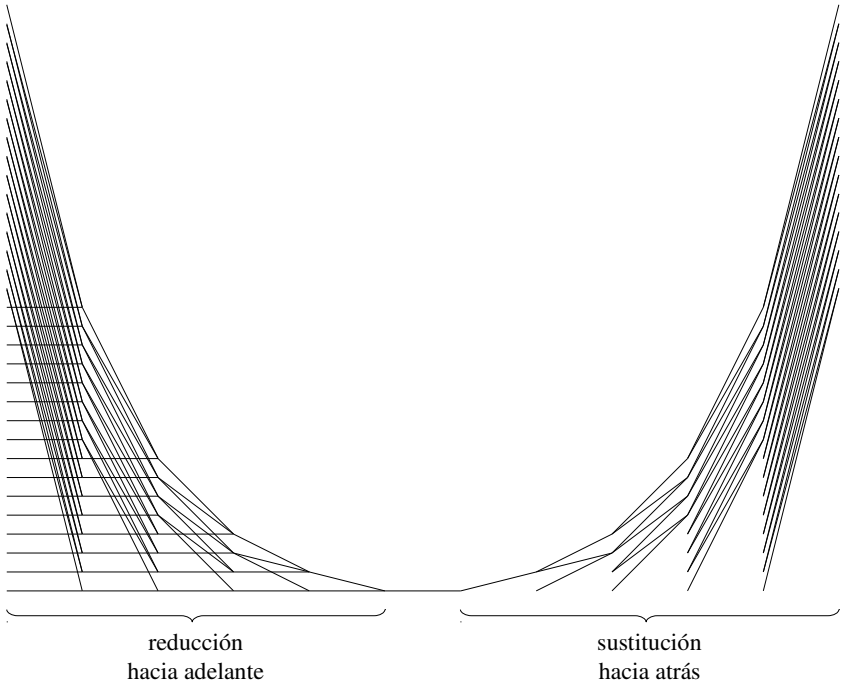


Figura 2.11: Grafo de dependencias del método de reducción cíclica para un sistema tridiagonal de 32 ecuaciones tras aplicar una estrategia de reordenación.

1: **para** $s = 0 \rightarrow \lfloor \log_2 n \rfloor - 1$ **paso** m:
2: < ejecutar m niveles de reducción (fase de división, en mem. compartida) >
3: < ejecutar m niveles de reducción (fase de fusión, en mem. global) >

4: **para** cada nivel s restante:
5: < ejecutar un nivel de reducción ($\lfloor n/2^s \rfloor$ ecuaciones, en mem. global) >

6: **para** varios niveles s:
7: < ejecutar un nivel de sustitución ($\lfloor n/2^{s+1} \rfloor$ incógnitas, en mem. global) >

8: < ejecutar los restantes niveles de sustitución (en mem. compartida) >

Figura 2.12: Pseudocódigo de una implementación en GPU del algoritmo de reducción cíclica aplicando la estrategia de "divide y fusiona".

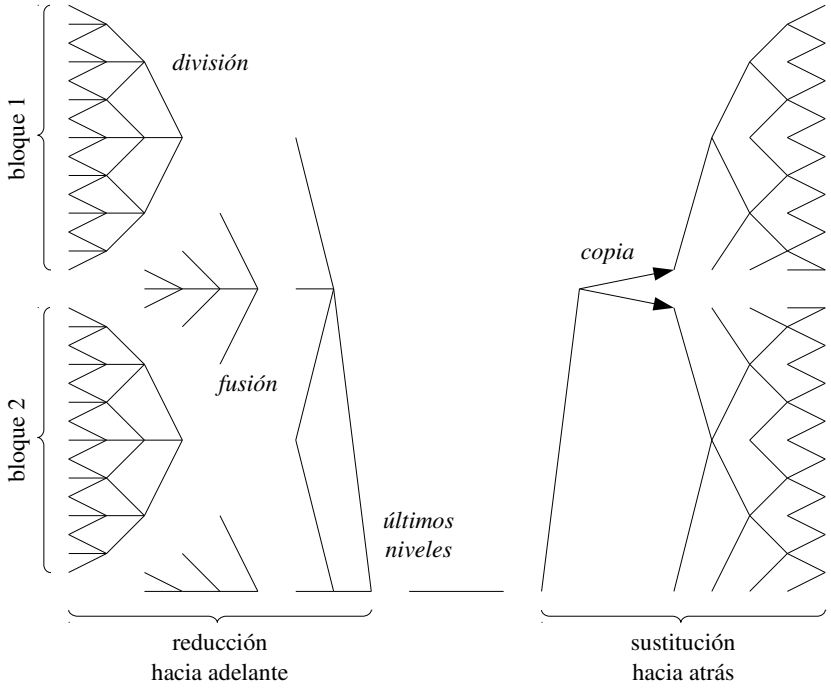


Figura 2.13: Grafo de dependencias del método de reducción cíclica utilizando el esquema "divide y fusiona".

niveles se resuelven de la misma manera que la implementación directa. A partir de cierto punto, sin embargo, es posible repartir el trabajo en bloques independientes y resolver los niveles restantes en la memoria compartida.

La figura 2.13 un ejemplo de aplicación de la aproximación "divide y fusiona" para la resolución de un sistema tridiagonal de 32 ecuaciones utilizando reducción cíclica. Al principio, todas las ecuaciones están almacenadas en memoria global (la primera columna de la figura). Durante la fase de división, cada bloque de hilos copia una porción del sistema de ecuaciones a su espacio de memoria compartida. En este ejemplo, hemos configurado una malla de dos bloques de 15 hilos. Dentro del espacio de memoria compartida, cada bloque ejecuta en paralelo tantos niveles de reducción hacia adelante como sea posible. En el ejemplo, la división ejecuta tres niveles de reducción. Una vez hecha la reducción, las ecuaciones transformadas se vuelven a copiar en memoria global.

Algunas ecuaciones no pueden ser calculadas durante la fase de división, y se procesan durante la fase posterior de fusión. Cuando la fase de fusión termina, la reducción hacia adelante lleva completados tantos niveles como aquellos que se han podido ejecutar durante la fase de división. En nuestro ejemplo, durante la fusión dos hilos transforman las ecuaciones que habían quedado intactas en la primera división, de forma que al final se han ejecutado tres niveles de reducción hacia adelante sobre todas las ecuaciones. Aunque esta fase podría ejecutarse en memoria compartida, nuestros experimentos demostraron que es más eficiente hacerlo en memoria global.

Dado que las dependencias de este algoritmo se estructuran formando una pirámide completa, una vez que finalizan las fases de división y fusión, los últimos niveles se calculan de la forma habitual. La fase de sustitución hacia atrás puede ser resuelta en bloques de trabajo independientes copiando los valores de las incógnitas allí donde sea necesario, como en el ejemplo que se muestra en la figura.

2.5.4. Doblamiento recursivo

En esta sección describimos el método de doblamiento recursivo para la resolución de sistemas tridiagonales de ecuaciones lineales y analizamos dos propuestas para su implementación en GPU.

Descripción

El algoritmo de doblamiento recursivo [29] para resolver sistemas tridiagonales de ecuaciones lineales tiene dos fases: eliminación y sustitución. Durante la fase de eliminación, el sistema se transforma de forma que las diagonales superior e inferior se desplazan hacia fuera. Esta fase se repite hasta que se obtiene un sistema con una única diagonal equivalente al original, que se resuelve durante la fase de sustitución.

La figura 2.14 muestra el estado de la matriz del sistema tridiagonal $A\mathbf{x} = \mathbf{d}$ después de aplicar el primer paso de eliminación. Las diagonales superior e inferior se han desplazado hacia afuera, dejando una diagonal de separación entre ellas y la diagonal principal. Además, los coeficientes a_2^1 y c_{n-1}^1 son ahora iguales a cero. Durante los siguientes pasos de la fase de eliminación, las diagonales externas se desplazarán hacia afuera hasta que solo quede la diagonal principal.

La ejecución de la fase de eliminación es similar a la fase de reducción hacia adelante del algoritmo de reducción cíclica, con la diferencia de que esta vez el número de ecuaciones

$$A^1 = \begin{pmatrix} b_1^1 & 0 & c_1^1 & & & 0 \\ 0 & b_2^1 & 0 & c_2^1 & & \\ a_3^1 & 0 & b_3^1 & 0 & \ddots & \\ & a_4^1 & \ddots & \ddots & \ddots & c_{n-2}^1 \\ & & \ddots & \ddots & \ddots & 0 \\ 0 & & & a_n^1 & 0 & b_n^1 \end{pmatrix}$$

Figura 2.14: Estado de la matriz tras aplicar el primer paso de eliminación de doblamiento recursivo.

no se reduce en cada paso. La fase se completa en $\lceil \log_2 n \rceil$ pasos, y en cada paso se generan n nuevas ecuaciones. Cada ecuación del tipo (2.9) se genera utilizando también la operación descrita en (2.10), pero esta vez con $i = 1, \ldots, n$ para todos los pasos (y asumiendo de nuevo $x_{i\pm 2^s} = 0$ si $i \pm 2^s \notin [1,n]$).

Al final de la fase de eliminación se obtiene un sistema diagonal. Este sistema se resuelve de forma trivial durante la fase de sustitución aplicando:

$$x_i = \frac{d_i^s}{b_i^s}. \tag{2.12}$$

La figura 2.15 muestra el grafo de dependencias para la resolución de un sistema de 32 ecuaciones usando doblamiento recursivo, donde los diferentes niveles de computación aparecen ordenados de izquierda a derecha. Durante la fase de eliminación el flujo de trabajo en cada nivel sigue un esquema de mariposa, donde el número de operaciones se mantiene constante en todos los niveles. La fase de sustitución consta de un único nivel en el que se calcula la solución del sistema.

Implementación

El esquema de almacenamiento de esta implementación utiliza cinco vectores localizados en la memoria global: las diagonales de la matriz, el término dependiente y la solución. Debido al patrón de dependencias de este algoritmo, las computaciones no se pueden realizar sobre esos mismos vectores (esto es, no se pueden realizar *in place*), así que hacen falta además cuatro vectores adicionales para almacenar las transformaciones efectuadas en las diagonales y el término dependiente.

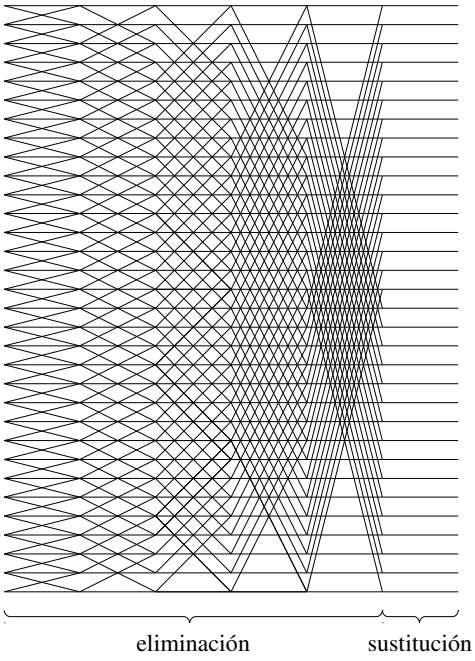


Figura 2.15: Grafo de dependencias del método de doblamiento recursivo para un sistema tridiagonal de 32 ecuaciones.

1: **para** $s = 0 \rightarrow \lceil \log_2 n \rceil - 1$:
2: < ejecutar un nivel de eliminación (n ecuaciones, en mem. global) >

3: < ejecutar resolución (n incógnitas, en mem. global) >

Figura 2.16: Pseudocódigo de una implementación directa en GPU del algoritmo de doblamiento recursivo.

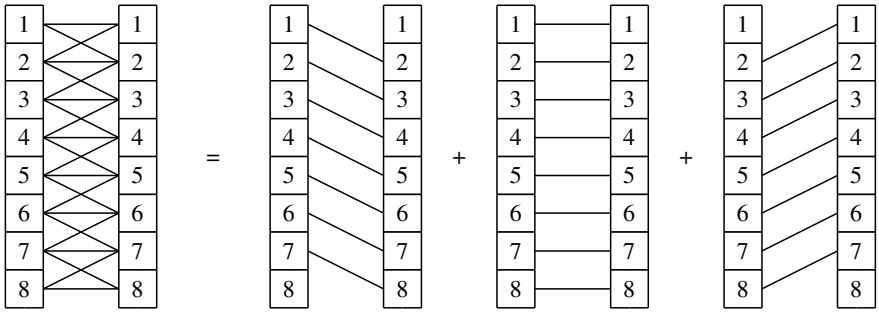


Figura 2.17: Dependencias en la aplicación del primer nivel de doblamiento recursivo a un sistema de 8 ecuaciones.

La figura 2.16 muestra el pseudocódigo de una implementación directa de este algoritmo en GPU, donde las llamadas a los *kernels* que ejecutan las fases de eliminación y resolución aparecen entre los símbolos "menor que" y "mayor que". El *kernel* de eliminación ejecuta un único nivel de la etapa de eliminación sobre todas las ecuaciones del sistema. Un bucle invoca este *kernel* hasta completar todos los niveles. Al finalizar, se realiza una única llamada al *kernel* de resolución, que calcula la solución del sistema.

El patrón de accesos a memoria global no muestra los problemas de coalescencia que aparecen en la reducción cíclica (véase sección 2.5.3). El motivo es que, para cada nivel del doblamiento recursivo, los accesos de los hilos se realizan a posiciones consecutivas; como ejemplo, la figura 2.17 muestra el patrón de accesos para el primer nivel de la fase de eliminación en un sistema tridiagonal de 8 ecuaciones.

Como en la reducción cíclica, una implementación directa no puede utilizar la memoria compartida ya que las dependencias entre datos impiden que los cálculos puedan distribuirse de forma independiente entre los procesadores. Al igual que antes, la estrategia de "divide y fusiona", descrita en la sección 2.3, ofrece una solución a este problema.

La figura 2.18 muestra el pseudocódigo de la implementación en GPU de doblamiento recursivo aplicando la técnica "divide y fusiona". Los primeros niveles de la etapa de eliminación se resuelven en GPU alternando una fase de división (que se ejecuta en la memoria compartida) y una fase de fusión (que se ejecuta en memoria global). Los últimos niveles de la etapa de reducción se ejecutan en memoria global, que se resuelve, al igual que antes, en una única llamada a un *kernel*.

1: **para** $s = 0 \rightarrow \lceil \log_2 n \rceil - 1$ **paso** m:
2: < ejecutar m niveles de eliminación (fase de división, en mem. compartida) >
3: < ejecutar m niveles de eliminación (fase de fusión, en mem. global) >

4: **para** cada nivel s restante:
5: < ejecutar un nivel de eliminación (n ecuaciones, en mem. global) >

6: < ejecutar resolución (n incógnitas, en mem. global) >

Figura 2.18: Pseudocódigo de una implementación en GPU del algoritmo de doblamiento recursivo aplicando la estrategia de "divide y fusión".

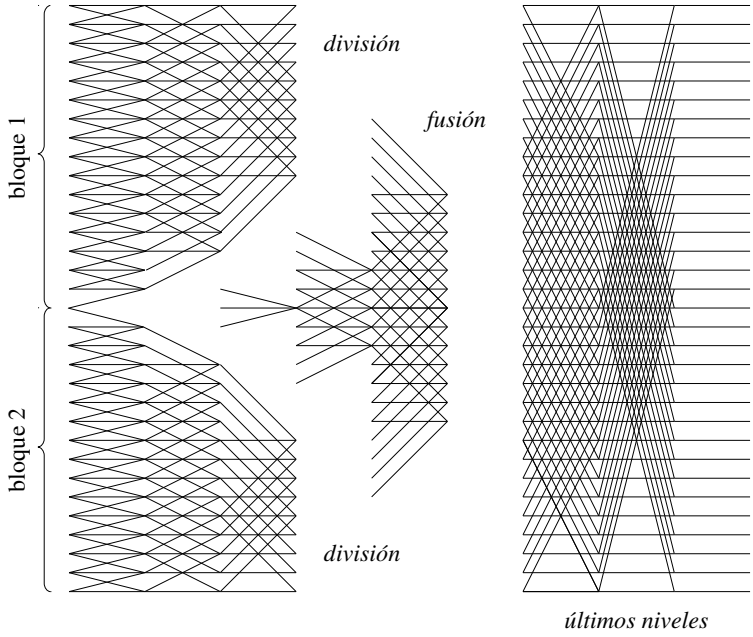


Figura 2.19: Grafo de dependencias del método de doblamiento recursivo para un sistema tridiagonal de 32 ecuaciones tras aplicar la estrategia de "división y fusión".

La figura 2.19 muestra el esquema de subdivisión de este algoritmo para un sistema tridiagonal de 32 ecuaciones. De forma similar al esquema de reducción cíclica, durante la fase de división las ecuaciones se dividen en bloques, que son copiadas por los bloques de hilos en la memoria compartida. Este ejemplo utiliza dos bloques de 16 hilos. Debido al limitado tamaño del bloque, el número de ecuaciones que se puede transformar dentro del bloque se reduce a cada paso de la fase de eliminación. En este caso, solo tres pasos de eliminación se pueden ejecutar en memoria compartida.

Una vez que se ha completado la división, la fase de fusión transforma las ecuaciones que no pudieron ser procesadas previamente en memoria compartida debido a dependencias irresolubles. En el ejemplo la fase de fusión ejecuta sucesivamente dos pasos de la fase de eliminación. El proceso podría continuar con otra fase de división y fusión, pero esto resultaría en accesos a datos no consecutivos en memoria global con una baja localidad espacial que afectaría negativamente al rendimiento. Por tanto, los últimos niveles del algoritmo se resuelven siguiendo el método estándar de resolución de sistemas tridiagonales mediante doblamiento recursivo.

2.5.5. Método de Bondeli

En esta sección describimos el método de Bondeli para la resolución de sistemas tridiagonales de ecuaciones lineales y analizamos su implementación en GPU.

Descripción

El método "divide y vencerás" de Bondeli [36] se basa en dividir el sistema en varios subsistemas de idéntico tamaño que pueden ser resueltos de forma concurrente. El método comienza dividiendo el sistema en bloques:

$$\begin{pmatrix} B_1 & C_1 & 0 & \cdots & 0 & 0 \\ A_2 & B_2 & C_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_p & B_p \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \end{pmatrix} = \begin{pmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \\ \vdots \\ \mathbf{d}_p \end{pmatrix}. \tag{2.13}$$

Cada término A_i, B_i y C_i, con $i = 1, \ldots, p$, representa un bloque de la matriz de tamaño $k \times k$, y cada vector $\mathbf{x}_i$ y $\mathbf{d}_i$ contiene k elementos, donde $k = n/p$ para p un entero positivo arbitrario tal que $p > 1$.

A continuación se ejecutan tres fases:

1. Se resuelven los siguientes $3p-2$ sistemas tridiagonales:

$$\begin{cases} B_i\mathbf{y}_i = \mathbf{d}_i & \text{para } i = 1,\ldots,p \\ B_1\mathbf{z}_1 = \mathbf{e}_k & \\ B_i\mathbf{z}_{2i-2} = \mathbf{e}_1;\ B_i\mathbf{z}_{2i-1} = \mathbf{e}_k & \text{para } i = 2,\ldots,p-1 \\ B_p\mathbf{z}_{2p-2} = \mathbf{e}_1, & \end{cases} \tag{2.14}$$

donde $\mathbf{y}_i$ (con $i = 1,\ldots,p$) y $\mathbf{z}_j$ (con $j = 1,\ldots,2p-2$) son vectores de incógnitas de tamaño k, y $\mathbf{e}_1^t = (1,0,\ldots,0)$ y $\mathbf{e}_k^t = (0,\ldots,0,1)$ son también vectores de tamaño k.

2. De los resultados obtenidos en pasos previos, se genera y resuelve un nuevo sistema tridiagonal $\mathscr{H}\vec{\alpha} = \vec{\beta}$ de tamaño $2p-2$:

$$\begin{pmatrix} s_1 & t_1 & & \\ r_2 & s_2 & \ddots & \\ & \ddots & \ddots & t_{2p-3} \\ & & r_{2p-2} & s_{2p-2} \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{2p-2} \end{pmatrix} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{2p-2} \end{pmatrix}, \tag{2.15}$$

donde

$$s_i = \begin{cases} \mathbf{e}_k^t\mathbf{z}_i & \text{si } i \text{ es impar} \\ \mathbf{e}_1^t\mathbf{z}_i & \text{si } i \text{ es par} \end{cases} \qquad \text{para } i = 1,\ldots,2p-2, \tag{2.16}$$

$$r_i = \begin{cases} \mathbf{e}_k^t\mathbf{z}_{i-1} & \text{si } i \text{ es impar} \\ 1/c_{ik/2} & \text{si } i \text{ es par} \end{cases} \qquad \text{para } i = 2,\ldots,2p-2, \tag{2.17}$$

$$t_i = \begin{cases} 1/b_{k(i+1)/2+1} & \text{si } i \text{ es impar} \\ \mathbf{e}_1^t\mathbf{z}_{i+1} & \text{si } i \text{ es par} \end{cases} \qquad \text{para } i = 1,\ldots,2p-3, \tag{2.18}$$

$$u_i = \begin{cases} -\mathbf{e}_k^t\mathbf{y}_{(i+1)/2} & \text{si } i \text{ es impar} \\ -\mathbf{e}_1^t\mathbf{y}_{i/2+1} & \text{si } i \text{ es par} \end{cases} \qquad \text{para } i = 1,\ldots,2p-2. \tag{2.19}$$

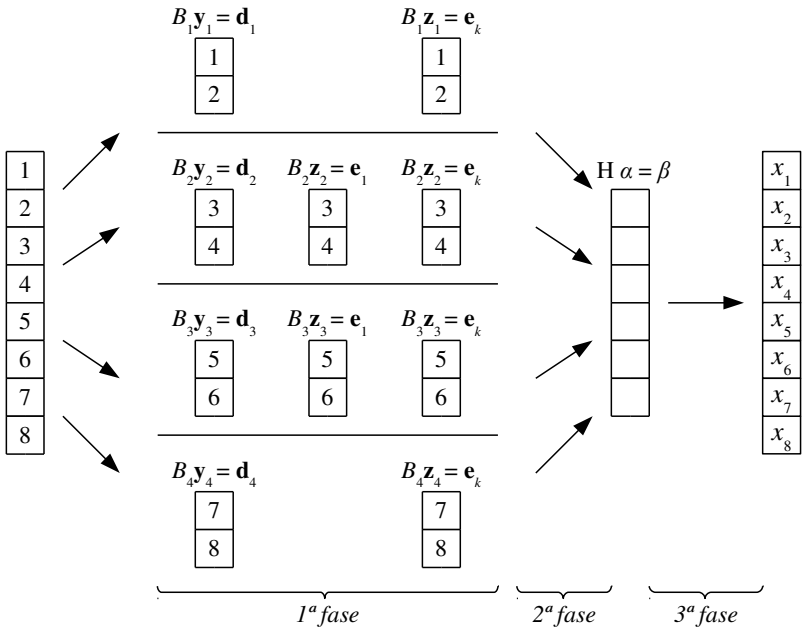


Figura 2.20: Dependencias en la resolución mediante el algoritmo de Bondeli de un sistema de 8 ecuaciones.

3. A partir del vector solución de $\mathscr{H}\vec{\alpha} = \vec{\beta}$ con tamaño $2p-2$ y de los resultados obtenidos en el primer paso, los valores de $\mathbf{x}_i$ se calculan como:

$$\mathbf{x}_i = \begin{cases} \mathbf{y}_i + \vec{\alpha}_i \mathbf{z}_i & \text{para } i = 1 \\ \mathbf{y}_i + \vec{\alpha}_{2i-2}\mathbf{z}_{2i-2} + \vec{\alpha}_{2i-1}\mathbf{z}_{2i-1} & \text{para } i = 2, \ldots, p-1 \\ \mathbf{y}_i + \vec{\alpha}_{2i-2}\mathbf{z}_{2i-2} & \text{para } i = p. \end{cases} \tag{2.20}$$

La figura 2.20 muestra cada una de las fases del método de Bondeli para un sistema de 8 ecuaciones dividido en 4 bloques de tamaño 2. Cada cuadrado representa una ecuación del sistema tridiagonal (1^a fase), una ecuación de los sistemas intermedios (2^a fase) o el cálculo de una incógnita (3^a fase). Durante la 1^a fase el sistema original se divide en bloques y se generan subsistemas de los que se obtienen los valores $\mathbf{y}_i$ y $\mathbf{z}_j$. En la 2^a fase se genera el sistema $\mathscr{H}\vec{\alpha} = \vec{\beta}$, que en este ejemplo tiene 6 ecuaciones. Por último, en la 3^a fase las incógnitas $\mathbf{x}_i$

1: < resolver sistemas del tipo $B_i\mathbf{y}_i = \mathbf{d}_i$ (p sistemas, en mem. compartida) >
2: < resolver sistemas del tipo $B_i\mathbf{z}_j = \mathbf{e}_1$ y $B_i\mathbf{z}_j = \mathbf{e}_k$ ($2p-2$ sistemas, en mem. compartida) >

3: < construir sistema del tipo $\mathscr{H}\vec{\alpha} = \vec{\beta}$ ($2p-2$ ecuaciones, en mem. global) >
4: resolver sistema del tipo $\mathscr{H}\vec{\alpha} = \vec{\beta}$

5: < calcular los valores de las incógnitas $\mathbf{x}_i$ (n incógnitas, en mem. global) >

Figura 2.21: Pseudocódigo de una implementación en GPU del método de Bondeli.

del sistema original se calculan a partir de la solución $\vec{\alpha}$ del sistema intermedio y los valores de $\mathbf{z}_j$.

Implementación

El esquema de almacenamiento de esta implementación consiste en cinco vectores que almacenan las tres diagonales, el término independiente y las soluciones. Además, los valores de los vectores $\mathbf{y}_i$ y $\mathbf{z}_j$ usados en (2.14) se almacenan en dos vectores separados, y el sistema de ecuaciones intermedio $\mathscr{H}\vec{\alpha} = \vec{\beta}$ descrito en (2.15) se almacena en otros cinco vectores, de forma similar al sistema original.

La figura 2.21 muestra el pseudocódigo de la implementación del método de Bondeli en GPU. Las diferentes etapas que componen el método de Bondeli se han implementado en varios *kernels* debido a las dependencias existentes entre las operaciones de este algoritmo (estas llamadas aparecen entre signos "menor que" y "mayor que").

La primera etapa, que consiste en la resolución de un gran número de subsistemas definidos en la ecuación (2.14), se ejecuta en dos *kernels* separados. El primero computa los sistemas del tipo $B_i\mathbf{y}_i = \mathbf{d}_i$, y el segundo los del tipo $B_i\mathbf{z}_j = \mathbf{e}_1$ y $B_i\mathbf{z}_j = \mathbf{e}_k$. Ambos tipos de sistemas son tridiagonales y pueden ser resueltos usando los algoritmos de reducción cíclica o doblamiento recursivo. En nuestras pruebas la reducción cíclica ofreció los mejores resultados. Los sistemas son lo suficientemente pequeños como para ser resueltos en paralelo dentro de la memoria compartida: cada bloque de hilos carga los valores de B_i en su espacio de memoria compartida, resuelve el subsistema y almacena las soluciones en los vectores $\mathbf{y}_i$ y $\mathbf{z}_j$ en la memoria global.

La segunda etapa consiste en la construcción y la resolución de un sistema de ecuaciones intermedio con la forma $\mathscr{H}\vec{\alpha} = \vec{\beta}$, a partir de las ecuaciones (2.17)-(2.19), que son computadas en un único *kernel* que se ejecuta en memoria global. El sistema generado es también

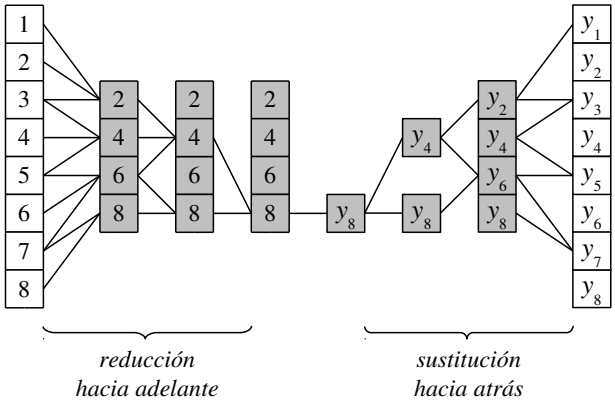


Figura 2.22: Dependencias para resolver un subsistema de 8 ecuaciones del tipo $B_i\mathbf{y}_i = \mathbf{d}_i$ usando reducción cíclica durante el método de Bondeli.

tridiagonal y puede ser resuelto usando uno de los algoritmos previos. Debido a sus satisfactorios resultados, escogimos el algoritmo de reducción cíclica para obtener la solución $\vec{\alpha}$ dentro del espacio de memoria global.

Por último, la tercera etapa consiste en la aplicación directa de la ecuación (2.20) para calcular los valores de las incógnitas $\mathbf{x}_i$ del sistema original. Esta operación se ejecuta en único *kernel* en la memoria global.

La figura 2.22 muestra el proceso de resolver un subsistema de 8 ecuaciones del tipo $B_i\mathbf{y}_i = \mathbf{d}_i$ en memoria compartida. Las ecuaciones del subsistema se leen de memoria global, pero solo las ecuaciones transformadas se copian a memoria compartida (donde las ecuaciones se han representado como cuadrados de fondo gris). Esto reduce el número de hilos necesarios para resolver cada subsistema así como los requisitos de memoria compartida en un factor de dos. Así, más bloques de hilos pueden ejecutarse en paralelo y, por tanto, más subsistemas pueden resolverse concurrentemente. En nuestro ejemplo, cada bloque de 4 hilos resuelve un subsistema de 8 ecuaciones.

2.5.6. Método de partición de Wang

En esta sección describimos el método de Wang para la resolución de sistemas tridiagonales de ecuaciones lineales y analizamos su implementación en GPU.

Descripción

El método paralelo para resolver un sistema tridiagonal de ecuaciones lineales presentado por Wang [180], conocido como método de partición, también está basado en la noción de "divide y vencerás". En este método el sistema se divide en subsistemas donde las diagonales superior e inferior se eliminan aplicando transformaciones elementales de fila. El proceso genera nuevos elementos en la matriz, denominados "elementos de relleno", que desaparecen en las últimas fases del algoritmo, cuando la matriz es diagonalizada.

Para un entero positivo arbitrario $p > 1$, asumiendo que n es divisible entre p y $k = n/p > 1$, este algoritmo tiene las siguientes fases:

1. División de la matriz A en $p \times p$ bloques de tamaño $k \times k$.

2. Aplicación de transformaciones elementales de fila en cada uno de los p bloques para eliminar la diagonal inferior. Este proceso genera "elementos de relleno" en la columna más a la derecha de todos los bloques diagonales inferiores:

$$f_{ik+1} := a_{ik+1}, \qquad \text{para } i = 1,2,\ldots,p-1, \tag{2.21}$$

y para $j = 2,\ldots,k$:

$$a_{ik+j} := a_{ik+j}/b_{ik+j-1}, \qquad \text{para } i = 0,1,\ldots,p-1, \tag{2.22}$$

$$b_{ik+j} := b_{ik+j} - a_{ik+j} \cdot c_{ik+j-1}, \qquad \text{para } i = 0,1,\ldots,p-1, \tag{2.23}$$

$$d_{ik+j} := d_{ik+j} - a_{ik+j} \cdot d_{ik+j-1}, \qquad \text{para } i = 0,1,\ldots,p-1, \tag{2.24}$$

$$f_{ik+j} := -a_{ik+j} \cdot f_{ik+j-1}, \qquad \text{para } i = 1,2,\ldots,p-1. \tag{2.25}$$

3. Aplicación de transformaciones elementales de fila en cada uno de los p bloques para eliminar la diagonal superior. Ahora, los "elementos de relleno" aparecen en la columna más a la derecha de los bloques diagonales superiores:

$$g_{ik-1} := c_{ik-1}, \qquad \text{para } i = 1,2,\ldots,p, \tag{2.26}$$

y para $j = k-1,k-2,\ldots,2$:

$$c_{ik+j-1} := c_{ik+j-1}/b_{ik+j}, \qquad \text{para } i = 0,1,\ldots,p-1, \tag{2.27}$$

$$g_{ik+j-1} := -c_{ik+j-1} \cdot g_{ik+j}, \qquad \text{para } i = 0,1,\ldots,p-1, \tag{2.28}$$

$$d_{ik+j-1} := d_{ik+j-1} - c_{ik+j-1} \cdot d_{ik+j}, \qquad \text{para } i = 0,1,\ldots,p-1, \tag{2.29}$$

$$f_{ik+j-1} := f_{ik+j-1} - c_{ik+j-1} \cdot f_{ik+j}, \qquad \text{para } i = 1,2,\ldots,p-1, \tag{2.30}$$

y también:

$$c_{ik} := c_{ik}/a_{ik+j}, \qquad \text{para } i = 1,2,\ldots,p-1, \tag{2.31}$$

$$b_{ik} := b_{ik} - c_{ik} \cdot f_{ik+1}, \qquad \text{para } i = 1,2,\ldots,p-1, \tag{2.32}$$

$$g_{ik} := -c_{ik} \cdot g_{ik+1}, \qquad \text{para } i = 1,2,\ldots,p-1, \tag{2.33}$$

$$d_{ik} := d_{ik} - c_{ik} \cdot d_{ik+1}, \qquad \text{para } i = 1,2,\ldots,p-1. \tag{2.34}$$

Los pasos (2.31) a (2.34) no se ejecutan si $k = 2$.

4. Eliminación de los "elementos de relleno" de la ik-ésima columna bajo la diagonal principal para $i = 1,\ldots,p-1$. Al final de este paso, la matriz tiene una forma triangular. Para completar este proceso, es necesario usar un vector temporal $\mathbf{t}$ de tamaño $k+1$:

   Para $i = 1,\ldots,p-1$:

$$t_j := g_{ik+j-1}, \qquad \text{para } j = 1,2,\ldots,k, \tag{2.35}$$

$$t_{k+1} := b_{(i+1)k}, \tag{2.36}$$

$$f_{ik+j} := f_{ik+j}/b_{ik}, \qquad \text{para } j = 1,2,\ldots,k, \tag{2.37}$$

$$t_{j+1} := t_{j+1} - f_{ik+j} \cdot t_1, \qquad \text{para } j = 1,2,\ldots,k, \tag{2.38}$$

$$d_{ik+j} := d_{ik+j} - f_{ik+j} \cdot d_{ik}, \qquad \text{para } j = 1,2,\ldots,k, \tag{2.39}$$

$$g_{ik+j} := t_{j+1}, \qquad \text{para } j = 1,\ldots,k-1, \tag{2.40}$$

$$b_{(i+1)k} := t_{k+1}. \tag{2.41}$$

5. Eliminación los "elementos de relleno" de la ik-ésima columna sobre la diagonal principal para $i = p,\ldots,1$. Al final de este paso, la matriz es diagonal:

   Para $i = p-1,\ldots,1$:

$$g_{ik+j-1} := g_{ik+j-1}/b_{(i+1)k}, \qquad \text{para } j = 1,\ldots,k, \tag{2.42}$$

$$d_{ik+j-1} := d_{ik+j-1} - g_{ik+j-1} \cdot d_{(i+1)k}, \qquad \text{para } j = 1,\ldots,k, \tag{2.43}$$

   y también:

$$g_j := g_j/b_k, \qquad \text{para } j = 1,\ldots,k-1, \tag{2.44}$$

$$d_j := d_j - g_j \cdot d_k, \qquad \text{para } j = 1,\ldots,k-1. \tag{2.45}$$

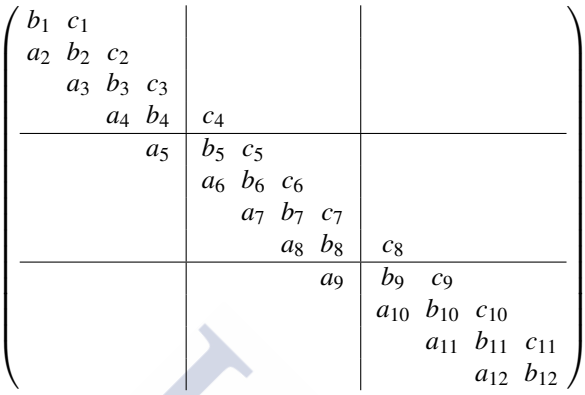


Figura 2.23: División de la matriz de un sistema tridiagonal de 12 ecuaciones en bloques de tamaño 4×4.

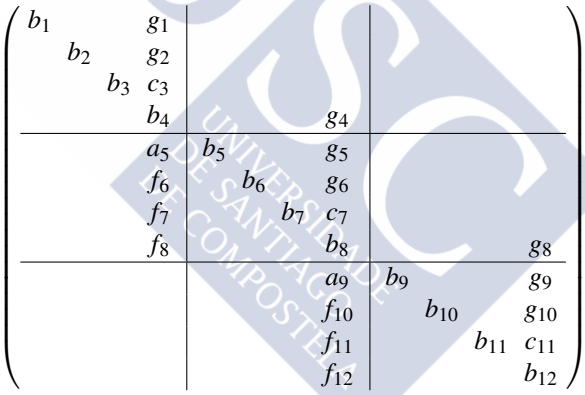


Figura 2.24: Estado de la matriz de un sistema de 12 ecuaciones después de que se hayan eliminado las diagonales superior e inferior.

6. Por último, calcular la solución como:

$$x_i = \frac{d_i}{b_i}, \qquad \text{para } i = 1, \ldots, n. \tag{2.46}$$

La figura 2.23 muestra una matriz A de un sistema tridiagonal de 12 ecuaciones dividida en bloques de 4×4 después de aplicar la 1ª fase del algoritmo. Durante la 2ª y 3ª fase, se realizan transformaciones de fila para eliminar las diagonales superior e inferior. Como resultado, aparecen varios "elementos de relleno" en las columnas más a la derecha de los

```
1: < ejecutar operaciones (2.21)-(2.25) (p hilos, en mem. global) >

2: < ejecutar operaciones (2.26)-(2.30) (n hilos, en mem. global) >
3: < ejecutar operaciones (2.31)-(2.34) (n hilos, en mem. global) >

4: < ejecutar operaciones (2.35)-(2.38) (n hilos, en mem. global) >
5: para i = 1 → p:
6:       < ejecutar operación (2.39) (k hilos, en mem. global) >
7: < ejecutar operaciones (2.40) y (2.41) (n hilos, en mem. global) >

8: < ejecutar operaciones (2.42) y (2.44) (n hilos, en mem. global) >
9: para i = 1 → p:
10:      < ejecutar operaciones (2.43) y (2.45) (k hilos, en mem. global) >

11: < calcular solución mediante operación (2.46) (n hilos, en mem. global) >
```

Figura 2.25: Pseudocódigo de una implementación en GPU del método de Wang.

bloques diagonales superiores e inferiores, tal y como se muestra en la figura 2.24. Durante las siguientes fases, se eliminan estos "elementos de relleno" hasta obtener un sistema de ecuaciones con una sola diagonal, cuya solución es trivial.

Implementación

Nuestra propuesta de implementación en GPU tiene el siguiente esquema de almacenamiento: cinco vectores almacenan las diagonales, el término independiente y las soluciones del sistema original; otros dos vectores del mismo tamaño se usan para almacenar los "elementos de relleno" que se generan durante los pasos intermedios de este algoritmo; por último, un vector almacena los datos temporales necesarios para eliminar los "elementos de relleno". Todos estos vectores están localizados en la memoria global.

Este algoritmo presenta algunas características que complican su implementación de forma eficiente en GPU. La gran cantidad de dependencias entre las operaciones hacen imposible distribuir las operaciones entre bloques de hilos que puedan resolver el sistema de forma independiente. Aunque esto ha sido un punto común a todos los algoritmos analizados hasta ahora, en el caso del algoritmo de Wang hemos tenido que implementar nuestra propuesta en nueve *kernels* diferentes. El hecho de que algunos de ellos se llamen de forma repetida da lugar a una importante sobrecarga. La figura 2.12 muestra el pseudocódigo de la implementación en GPU, con las llamadas a los *kernels* indicadas mediantes los signos "menor que" y "mayor que". Describimos cada etapa de la implementación a continuación.

En la primera etapa dividimos la matriz del sistema en bloques y eliminamos los coeficientes de la diagonal inferior aplicando las operaciones descritas en las ecuaciones (2.21)-(2.25). Estas operaciones deben ser ejecutadas en el orden descrito debido a las dependencias existentes entre ellas. El proceso es realizado por un *kernel* que invoca tantos hilos como bloques en que se ha dividido la matriz, de forma que cada hilo se encarga de transformar un bloque.

En la segunda etapa eliminamos los coeficientes de la diagonal superior, esta vez aplicando las operaciones (2.26)-(2.34). En este caso, debido a las dependencias entre las operaciones, el proceso es realizado por dos *kernels* que se ejecutan consecutivamente, uno para las operaciones (2.26)-(2.30), y el otro para las operaciones (2.31)-(2.34). Cada uno de ellos lanza tantos hilos como ecuaciones en el sistema, de forma que cada bloque de hilos procesa un bloque de la matriz.

En la tercera etapa comienza la eliminación de los "elementos de relleno" generados en las etapas anteriores. Las operaciones (2.35)-(2.38) se ejecutan en un *kernel* que lanza tantos hilos como ecuaciones. La operación (2.39), debido a sus dependencias, debe ser ejecutada secuencialmente, y un *kernel* se encarga de ello. Por último, las operaciones (2.40) y (2.41) se resuelven en un tercer *kernel* donde cada hilo procesa una ecuación del sistema.

En la cuarta etapa se finaliza la eliminación de los "elementos de relleno" y se obtiene una matriz diagonal. Las operaciones (2.42) y (2.44) pueden paralelizarse en un *kernel* que lance tantos hilos como ecuaciones. Sin embargo, las operaciones (2.43) y (2.45), implementadas en otro *kernel*, deben ejecutarse secuencialmente.

Por último, un *kernel* se encarga de calcular la solución sobre el sistema diagonal, donde cada hilo aplica para cada ecuación la operación (2.46).

Todos los datos se leen y escriben en memoria global, ya que el número de operaciones realizado en cada *kernel* no justifica la sobrecarga de copiar datos entre la memoria global y la compartida antes y después de ejecutar las operaciones.

2.5.7. Resultados

En esta sección presentamos las medidas de rendimiento y complejidad obtenidas para las diferentes propuestas de los algoritmos de resolución de sistemas tridiagonales de ecuaciones lineales presentadas en este capítulo.

Metodología

Hemos evaluado nuestras propuestas en una NVIDIA GeForce GTX 295 (véase sección 1.7). Medimos el tiempo de ejecución de los accesos a la memoria global, la memoria compartida, y el tiempo dedicado exclusivamente a la computación. Las medidas de rendimiento fueron obtenidas promediando los resultados de cien iteraciones de cada algoritmo. Para estimar el tiempo de ejecución consumido en accesos a memoria, reemplazamos dichos accesos por accesos a registros, y calculamos el tiempo a partir de la diferencia entre esta versión de la implementación y la original.

Hemos comparado el rendimiento de nuestras implementaciones en GPU con una implementación del método de Bondeli en OpenMP, ya que de los métodos implementados en CPU, este ha sido el que ha ofrecido mejor rendimiento. Esta versión del algoritmo fue compilada con *gcc* en un procesador Intel Core 2 Quad Q9450, con 4 *cores* a 2.66 GHz y 4 GB de RAM, en un sistema operativo Linux.

Aunque el tiempo asociado a la transferencia de datos entre la memoria de la CPU y la memoria de la GPU puede ser ignorado si el algoritmo de resolución forma parte de un proceso más grande ejecutado completamente en CPU [189], al final de esta sección incluimos varios medidas de este tiempo de transferencia.

Medidas de rendimiento

En las esta sección presentamos un análisis detallado de las medidas de rendimiento de cada una de las soluciones planteadas para la resolución de sistemas tridiagonales en GPU.

La tabla 2.1 muestra la complejidad de los algoritmos considerados. La tabla muestra los valores de complejidad en tamaño en memoria, número de accesos a memoria y número de operaciones en función del número de ecuaciones del sistema (n), del tamaño de los bloques (m) cuando se aplica la técnica de "divide y fusiona", y del número de subsistemas (p) en el caso del método de Bondeli. De acuerdo a estos datos, el algoritmo con el menor número de computaciones y el menor número de accesos a memoria es reducción cíclica. Sin embargo, esta información no proporciona una predicción del rendimiento de cada implementación debido a la influencia del patrón de accesos a memoria y del patrón de computación de los algoritmos.

La tabla 2.2 muestra un resumen de las medidas obtenidas para las diferentes implementaciones en GPU de los algoritmos para un sistema tridiagonal de 2^{20} ecuaciones usando aritmética de simple y doble precisión. El rendimiento de las operaciones de doble precisión

Algoritmo	Tamaño en memoria	Accesos mem. global	Accesos mem. comp.	Operaciones aritméticas
Reducción cíclica				
Directa	$5n$	$23n$	-	$17n$
Reordenación	$9n$	$33n$	-	$17n$
Divide y fusiona	$5n$	$12n$	$27n$	$17n$
Doblamiento recursivo				
Directa	$9n$	$16n\log_2 n$	-	$12n\log_2 n$
Divide y fusiona	$9n$	$16n\log_2 n - 8n\log_2 m$	$8n\log_2 m$	$12n\log_2 n$
Método de Bondeli				
Divide y vencerás	$7n+10p$	$40n$	$36n$	$46n$
Método de Wang				
Divide y vencerás	$8n$	$52n$	-	$22n$

Tabla 2.1: Medidas de complejidad para las propuestas de resolución de sistemas tridiagonales en GPU.

Algoritmo	Tiempo (precisión simple)	Aceleración (precisión simple)	Tiempo (precisión doble)	Aceleración (precisión doble)
Método de Bondeli				
OpenMP	0,036574 s	1,0x	0,048030 s	1,0x
Reducción cíclica				
Directa	0,005065 s	7,2x	0,008237 s	5,8x
Reordenación	0,004073 s	9,0x	0,005468 s	8,8x
Divide y fusiona	0,002901 s	12,6x	0,007265 s	6,6x
Doblamiento recursivo				
Directa	0,015189 s	2,4x	0,029848 s	1,6x
Divide y fusiona	0,012833 s	2,8x	0,027462 s	1,7x
Método de Bondeli				
Divide y vencerás	0,005868 s	6,2x	0,018212 s	2,6x
Método de Wang				
Divide y vencerás	0,154060 s	0,2x	0,162722 s	0,3x

Tabla 2.2: Tiempos de ejecución de los métodos de resolución de sistemas tridiagonales implementados en una tarjeta GTX 295 para un sistema de 2^{20} ecuaciones. Se muestran también los valores de aceleración sobre una implementación en OpenMP del método de Bondeli.

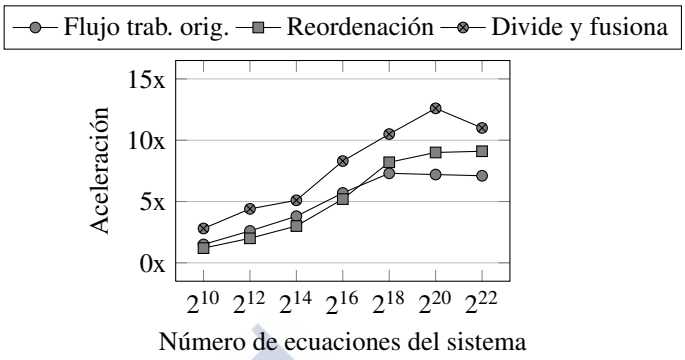


Figura 2.26: Valores de aceleración para diferentes tamaños del sistema de ecuaciones de las tres propuestas de reducción cíclica implementadas en una tarjeta GTX 295.

depende de la arquitectura GPU utilizada, pero para este estudio en particular el análisis en ambos casos es similar, por lo que en adelante solo consideraremos el caso de precisión simple. La primera fila de la tabla muestra el rendimiento de una implementación en OpenMP del método de Bondeli. Utilizamos esta implementación como referencia para medir los valores de aceleración de las implementaciones en GPU. Las tres siguientes filas de la tabla muestran el tiempo consumido en la ejecución de nuestras tres propuestas de reducción cíclica. A continuación, se muestra el tiempo consumido por las dos propuestas de doblamiento recursivo. Las dos últimas filas muestran los tiempos consumidos por las implementaciones de los métodos de Bondeli y Wang.

La implementación directa de reducción cíclica en GPU consigue una aceleración superior a 7x. Al aplicar una estrategia de reordenación para incrementar la localidad espacial de los accesos la aceleración crece hasta 9x. Por último, la propuesta "divide y fusiona" tiene el valor de aceleración más alto, superior a 12x. En términos de tiempo consumido y aceleración, esta última propuesta obtiene los mejores resultados. La implementación directa de doblamiento recursivo alcanza un valor de aceleración de 2,4x, y al aplicar la técnica de "divide y fusiona" este valor se incrementa hasta 2,8x. La siguiente fila muestra el tiempo consumido por la implementación del método de Bondeli, que alcanza una aceleración de 6,2x en GPU respecto a la implementación del mismo algoritmo en OpenMP. La implementación del método de Wang consigue un valor de aceleración insatisfactorio dentro del contexto de la computación en GPU.

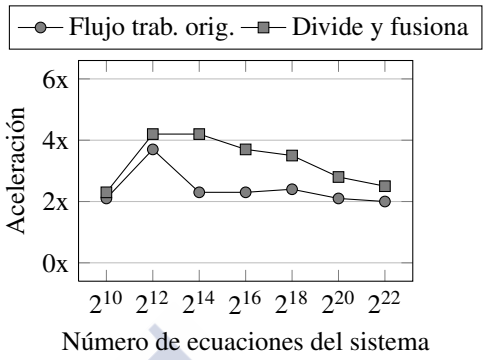


Figura 2.27: Valores de aceleración para diferentes tamaños del sistema de ecuaciones de las dos propuestas de doblamiento recursivo implementadas en una tarjeta GTX 295.

La figura 2.26 muestra las medidas de aceleración obtenidas para las tres propuestas de reducción cíclica con diferentes tamaños del sistema de ecuaciones. Para pequeños sistemas, la aceleración es baja, ya que el *hardware* paralelo no puede ser explotado eficientemente debido al bajo número de computaciones asociado. Para sistemas más grandes la GPU muestra un rendimiento muy bueno en términos de aceleración. Este comportamiento es típico de las GPU, que necesitan procesar miles de datos para explotar al máximo sus capacidades computacionales [93].

Las propuestas basadas en las estrategias de reordenación y "divide y fusiona" se centran en reducir la sobrecarga asociada a los accesos a memoria a la vez que mantienen el mismo número de operaciones. La propuesta basada en reordenación reduce esta sobrecarga aumentando la localidad espacial de los accesos a memoria global. Por otra parte, la propuesta basada en "divide y fusiona" reduce el número de accesos a memoria global haciendo uso del espacio de memoria compartida, que tiene una latencia menor. La versión "divide y fusiona" muestra los valores de aceleración más altos, ya que utilizar el espacio de memoria compartida ofrece mejores resultados que usar el espacio de memoria global.

La figura 2.27 muestra los valores de aceleración para las propuestas de doblamiento recursivo para sistemas de ecuaciones de diferente tamaño. En esta ocasión se observa que los valores de aceleración decrecen a medida que aumentamos el número de ecuaciones del sistema. Al incrementar el tamaño, aumenta también el número de accesos a memoria global, con lo que el rendimiento no escala tan bien como en las propuesta de reducción cíclica. Ambas propuestas realizan el mismo número de computaciones, pero la segunda está diseñada espe-

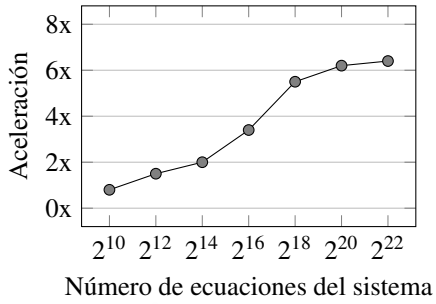


Figura 2.28: Valores de aceleración para diferentes tamaños del sistema de ecuaciones de la propuesta del método de Bondeli implementada en una tarjeta GTX 295.

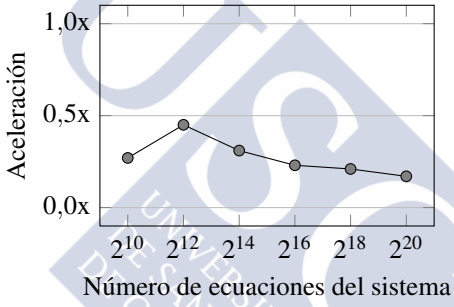


Figura 2.29: Valores de aceleración para diferentes tamaños del sistema de ecuaciones de la propuesta del método de Wang implementada en una tarjeta GTX 295.

cíficamente para reducir la sobrecarga de los accesos a memoria y, en consecuencia, ofrece los valores más altos de aceleración.

Las figuras 2.28 y 2.29 contienen las gráficas de aceleración para las implementaciones de los métodos de Bondeli y Wang, respectivamente. Los bajos e irregulares valores de aceleración muestran que el método de Wang no es adecuado para ser implementado en GPU.

La figura 2.30 muestra el tiempo consumido en la resolución de un sistema tridiagonal de 2^{20} ecuaciones, repartido en accesos a memoria y computación, para las tres propuestas. El acceso a memoria de la propuesta basada en el flujo de trabajo original consume cerca del 90 % del tiempo, mientras que aplicar la estrategia de reordenación reduce esta cantidad a un 80 %. Esto es debido al incremento de la localidad espacial en los accesos a memoria global. Como resultado, se obtienen mejores valores de consumo de ancho de banda, similares a los obtenidos en [189], aunque por debajo de los más de 220 GB/s anunciados por el fa-

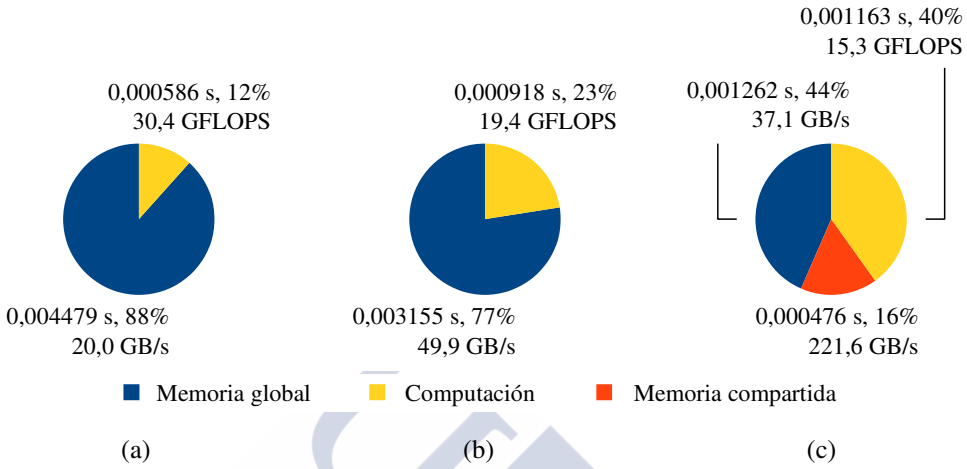


Figura 2.30: Distribución del tiempo de ejecución durante la resolución de un sistema de 2^{20} ecuaciones en una tarjeta GTX 295 para las tres propuestas de reducción cíclica: (a) flujo de trabajo original, (b) reordenación y (c) "divide y fusiona".

bricante. En [28] se aplica una técnica similar, con una mejora de rendimiento prácticamente idéntica. En la propuesta basada en el método de "divide y fusiona" el tiempo necesario para acceder a memoria se reduce al explotar eficientemente el espacio de memoria compartida. En la bibliografía pueden encontrarse otras propuestas enfocadas a explotar este espacio de memoria [70, 189]. Sin embargo, como ya se ha mencionado anteriormente, estas soluciones están enfocadas a resolver en paralelo conjuntos de sistemas tridiagonales pequeños, donde la implementación es prácticamente directa.

Con respecto al tiempo de computación, la estrategia de reordenación y la estrategia "divide y fusiona" son menos eficientes que la propuesta basada en el flujo de trabajo original. La reordenación incrementa este tiempo para poder realizar las operaciones de ordenación de las ecuaciones. En "divide y fusiona" el incremento es algo mayor, debido a los problemas de divergencia que aparecen durante las fases de división: cada bloque de hilos opera sobre un grupo de ecuaciones que se reduce a cada paso, por lo que algunos hilos permanecen ociosos. Los problemas de divergencia no están presentes en las otras propuestas porque los *kernels* implementados solo lanzan los hilos que necesitan. Como el tiempo de computación se incrementa pero el número de operaciones permanece constante, el valor de GFLOPS disminuye.

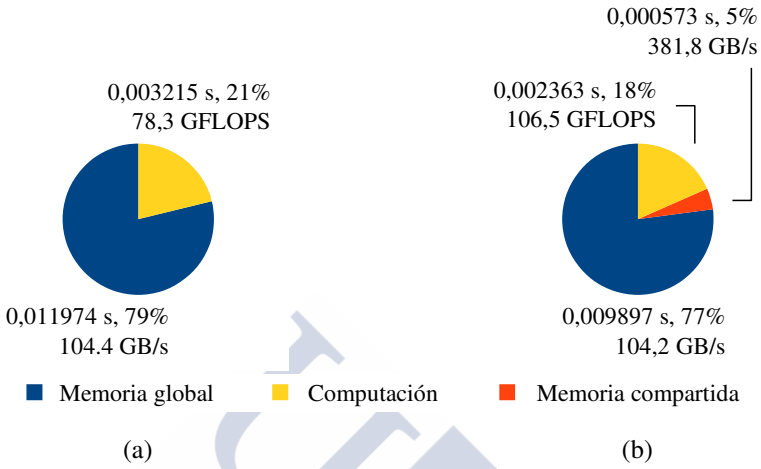


Figura 2.31: Distribución del tiempo de ejecución durante la resolución de un sistema de 2^{20} ecuaciones en una tarjeta GTX 295 para las dos propuestas de doblamiento recursivo: (a) flujo de trabajo original y (b) "divide y fusiona".

En resumen, la propuesta basada en la estrategia de "divide y fusiona" proporciona el mejor rendimiento con una aceleración superior a 12x. Aunque su rendimiento es peor en tiempo de computación, al utilizar el espacio de memoria compartida y reducir sustancialmente la sobrecarga de los accesos a memoria, esta propuesta se convierte en la más aconsejable de las analizadas para implementar la resolución de sistemas tridiagonales de ecuaciones lineales en la GPU.

La figura 2.31 muestra el tiempo consumido en accesos a memoria y computación por las dos propuestas de doblamiento recursivo para resolver un sistema tridiagonal de 2^{20} ecuaciones. Los accesos a memoria consumen la mayor parte del tiempo en ambas propuestas. La propuesta basada en el flujo de trabajo original consume 11 ms en accesos al espacio de memoria global, mientras que la solución basada en "divide y fusiona" necesita cerca de 10 ms para acceder a la memoria global y menos de 1 ms para acceder a la memoria compartida. Como resultado, la versión "divide y fusiona" presenta una ligera mejora en aceleración respecto a la original. El ancho de banda de memoria global es el doble que el conseguido en reducción cíclica ya que se aprovechan mejor las transacciones en memoria. En cuanto al tiempo de computación, los bajos requisitos de la propuesta "divide y vencerás" son debidos

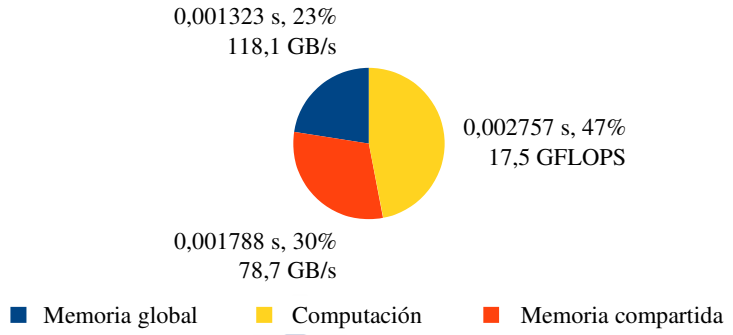


Figura 2.32: Distribución del tiempo de ejecución durante la resolución de un sistema de 2^{20} ecuaciones en una tarjeta GTX 295 para la propuesta de implementación del método de Bondeli.

al bajo número de llamadas a *kernel* presentes en esta propuesta. El número de operaciones es el mismo en ambas propuestas, por lo que el valor de GFLOPS es mayor.

Nuestras pruebas mostraron que la técnica de "divide y fusiona" produce buenos resultados cuando se aplica a los primeros pasos del algoritmo (cuatro pasos en el caso de los problemas analizados). Si la técnica se aplica a más pasos, se produce una fuerte degradación del rendimiento asociada al tiempo de acceso a memoria: los accesos se realizan a datos individuales, ya que cada bloque de hilos utiliza datos que se encuentran en posiciones distantes de memoria, y aunque el patrón de accesos es similar al del algoritmo de reducción cíclica, en este caso el número de accesos es mayor. Por tanto, la técnica de "divide y fusiona" no produce resultados tan positivos como con la reducción cíclica.

En resumen, la propuesta basada en aplicar la estrategia "divide y fusiona" al algoritmo de doblamiento recursivo es la mejor en términos de rendimiento en la GPU, con un valor de aceleración de 2,8x. Aunque se ha explotado el espacio de memoria compartida, no se han obtenidos grandes mejoras debido al patrón de accesos a memoria de la resolución de los sistemas tridiagonales.

La figura 2.32 muestra el gráfico de la distribución de tiempos para la implementación en GPU del método de Bondeli. Se observa que el tiempo dedicado a los accesos a memoria global está equilibrado con el tiempo dedicado a los accesos a memoria compartida. Estos resultados muestran que la jerarquía de memoria ha sido explotada eficientemente, aunque debido a los conflictos de memoria asociados a la solución de los subsistemas en la memoria compartida, no se alcanza el máximo ancho de banda. El tiempo de computación consume la

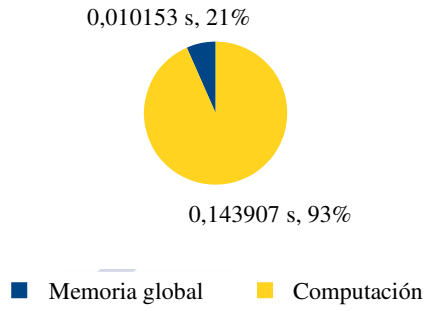


Figura 2.33: Distribución del tiempo de ejecución durante la resolución de un sistema de 2^{20} ecuaciones en una tarjeta GTX 295 para la propuesta de implementación del método de Wang.

mitad del tiempo de ejecución debido a que los requisitos de computación son mayores que en los otros algoritmos (hay un gran número de fases y el flujo de trabajo es más complejo).

Por último, la figura 2.33 muestra la distribución del tiempos para el método de Wang. En este caso, el tiempo de computación es muy alto comparado con los algoritmos anteriores, como resultado del gran número de *kernels* necesarios para implementar el algoritmo. La fragmentación de las computaciones que impide la explotación de la capacidad computacional de la GPU.

En resumen, la estructura del método de Wang no es adecuada para su implementación en GPU. Las dependencias entre las operaciones del algoritmo impiden que se puedan distribuir las tareas a realizar en bloques de hilos independientes de forma eficiente. Los resultados obtenidos son muy inferiores a los de otros algoritmos de resolución de sistemas tridiagonales analizados en este capítulo.

Medidas de precisión

En cuanto a la estabilidad numérica, ninguno de los algoritmos aquí considerados hace uso de pivote. Por tanto, para un sistema tridiagonal de ecuaciones arbitrario, cualquier método podría dar lugar a resultados imprecisos. El método de reducción cíclica es estable para sistemas de diagonal dominante y sistemas simétricos y positivamente definidos [106]. El

Algoritmo	Precisión	CPU		GPU	
		aleatorio	conj. nivel	aleatorio	conj. nivel
Algoritmo de Thomas	simple	0,415	0,182	n/a	n/a
	doble	0,212	0,054	n/a	n/a
Método de Bondeli	simple	0,422	0,190	n/a	n/a
(con alg. de Thomas)	doble	0,215	0,050	n/a	n/a
Reducción cíclica	simple	2,412	1,094	2,453	1,101
	doble	0,509	0,208	0,509	0,208
Doblamiento recursivo	simple	8,734	8,739	8,734	7,667
	doble	1,307	1,307	1,581	1,443
Método de Bondeli	simple	2,566	1,301	2,695	1,279
(con reducción cíclica)	doble	0,556	0,219	0,663	0,237

Tabla 2.3: Medidas de error para un sistema de 2^{20} ecuaciones (los valores deben multiplicarse por 10^{-5} para precisión simple y por 10^{-13} para precisión doble).

método de Bondeli es estable para sistemas de diagonal dominante (o al menos positivamente definidos) [36].

Para determinar la precisión de los algoritmos, tanto en simple como en doble precisión, comparamos nuestros resultados con los obtenidos utilizando la librería de precisión extendida *GNU Multiple Precision Arithmetic Library* [7]. Realizamos dos conjuntos de experimentos: el primero con coeficientes aleatorios en las diagonales, y el segundo utiliza sistemas de diagonal dominante derivados de un método del conjunto de nivel. La tabla 2.3 muestra el error obtenido tanto para las implementaciones en CPU como en GPU. Las medidas de error se calcularon usando la siguiente fórmula:

$$\varepsilon = \frac{1}{n}\sqrt{\sum_i \left(\frac{x_i - x_i'}{x_i}\right)^2}, \tag{2.47}$$

donde n es el tamaño del sistema, y x_i e x_i' son, respectivamente, los valores de las soluciones obtenidas por nuestras implementaciones y por la librería. Cada medida de error en la tabla es el resultado de promediar las medidas de error obtenidas para diez sistemas diferentes.

La tabla no muestra valores de error para el algoritmo de de Thomas en GPU, ya que no se trata de un método adecuado para su ejecución en esa arquitectura. Entre los algoritmos paralelos ejecutados en GPU, nuestros experimentos muestran que el algoritmo de reducción cíclica y el método de Bondeli (combinado con reducción cíclica para resolver los subsistemas) obtienen el menor error, mientras que el mayor error se produce con doblamiento

recursivo. En resumen, la precisión de los resultados es similar tanto en CPU como en GPU, usando simple o doble precisión. Un estudio similar puede encontrarse en [189].

Comparación con otros trabajos

El mayor reto a la hora de afrontar la implementación de una solución en GPU para la resolución de sistemas tridiagonales es cómo distribuir de forma eficiente las tareas entre los bloques de hilos. Para sistemas pequeños (de 1024 ecuaciones o menos), la solución puede ser calculada en memoria compartida y en el espacio de registros sin necesidad de usar resultados intermedios almacenados en memoria global. Así, es posible obtener mejoras sustanciales de rendimiento, como se puede observar en los trabajos comentados en esta sección.

En [189] se propone una aproximación híbrida para resolver múltiples sistemas de hasta 512 ecuaciones. La implementación utiliza reducción cíclica, pero cambia a doblamiento recursivo cuando el nivel de paralelismo es insuficiente para mantener a la GPU ocupada. Con esta aproximación, 512 sistemas de 512 ecuaciones pueden calcularse en 0,4 ms. Nuestra mejor implementación resuelve un sistema de 512^2 ecuaciones, que es una tarea más compleja ya que hay que calcular y recoger resultados intermedios, en 1 ms.

Como ya se mencionó en la sección 2.5.3, la estrategia de reordenación ya había sido aplicada en [70] con el objetivo de reducir los conflictos de bancos de memoria asociados a los accesos a memoria compartida que tienen lugar durante la ejecución de reducción cíclica. Esta implementación resuelve 513 sistemas independientes de 513 ecuaciones en 0,4 ms. Aunque no consigue mejorar el rendimiento conseguido en [189], la implementación es mucho más sencilla. En [28] también se aplica la técnica de reordenación, pero con el objetivo de aumentar la localidad espacial de los accesos a memoria global (aquí la memoria compartida se aprovecha para evitar los accesos repetidos a algunas variables almacenadas en la memoria global). Esta técnica requiere el doble de memoria que otras soluciones, y los autores obtienen resultados análogos a los que presentamos en este capítulo.

En [56] se presenta la técnica de empaquetado de registros, que reduce la comunicación en memoria compartida a base de explotar el espacio de registros y asignar a cada hilo más ecuaciones a procesar. Los autores utilizan esta técnica para implementar en GPU el algoritmo de reducción cíclica, y muestran resultados para la resolución en paralelo de 1024 sistemas de 1024 ecuaciones, con un tiempo de 0,7 ms, así como 4096 sistemas de 4096 ecuaciones, con un tiempo de 23,4 ms, utilizando una GTX 460. Sin embargo, al tratar de aplicar esta técnica para resolver un único sistema de gran tamaño, no conseguimos buenos resultados,

pues continúan los problemas de coalescencia en los accesos a memoria global que penalizan el rendimiento.

En [57] se propone un método multietapa, en el que, mediante la aplicación de doblamiento recursivo, se divide el sistema original en sistemas cada vez más pequeños, hasta que finalmente tienen un tamaño lo suficientemente reducido como para ser resueltos en memoria compartida. Esta solución utiliza además una estrategia de autoajuste para seleccionar los cambios de etapa, y obtiene valores de rendimiento similares a los de [56]. Sin embargo, en el caso extremo de sistemas enormes (2^{22} ecuaciones), los autores informan que el rendimiento de la solución en CPU supera al de GPU.

En el caso de sistemas tridiagonales muy grandes es necesario distribuir las ecuaciones entre bloques de hilos e implementar un mecanismo que combine los resultados parciales obtenidos por estos bloques para obtener la solución global. Podría parecer un problema similar al que se plantea en las aproximaciones para calcular la transformada Fourier rápida (en inglés FFT, *fast Fourier transform*), donde la transformación de una secuencia grande de datos puede resolverse combinando los resultados de FFT de subsecuencias que sean lo suficientemente pequeñas como para caber en la memoria compartida. Normalmente los datos se disponen en una matriz bidimensional y la FFT se calcula a lo largo de las columnas y las filas [74, 128]. Sin embargo, esta técnica no puede se aplicada directamente en la resolución de sistemas tridiagonales, ya que, como hemos visto, la factorización de un sistema tridiagonal en subproblemas que puedan ser resueltos de forma independiente no es tan regular como el caso de la FFT.

Nuestras técnicas permiten resolver eficientemente sistemas grandes de ecuaciones en GPU. Por ejemplo, nuestra propuesta de reducción cíclica basada en la estrategia de "divide y fusiona" ofrece los mejores resultados, con una aceleración superior a 12x sobre una solución paralela implementada en CPU.

Observaciones finales

En esta sección hemos descrito varias técnicas aplicadas a algoritmos de resolución de sistemas tridiagonales de ecuaciones lineales que distribuyen el trabajo entre bloques de hilos, incrementan la localidad espacial de los accesos a memoria, explotan la jerarquía de memoria, etc. A pesar de que hemos seleccionado las técnicas más adecuadas para su implementación en GPU siguiendo el modelo CUDA, las dependencias entre computaciones que limitan la

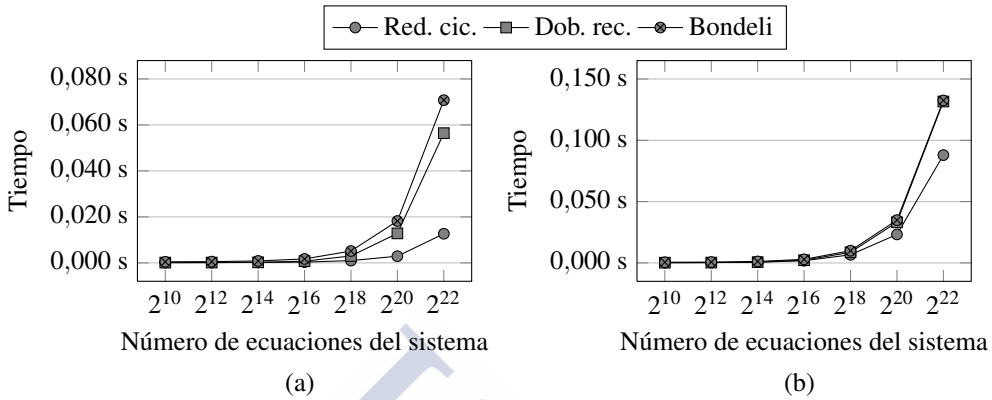


Figura 2.34: Comparación de tiempos de ejecución de varios algoritmos de resolución de sistemas tridiagonales en una tarjeta GTX 295 (a) sin tiempo de transferencia y (b) con tiempo de transferencia.

explotación del paralelismo de grano fino hacen de la resolución de sistemas tridiagonales un problema poco adecuado para su paralelización en GPU.

El algoritmo de reducción cíclica ofrece los mejores resultados de rendimiento gracias a la regularidad de su estructura y la distribución de sus operaciones, que permite utilizar de forma eficiente el espacio de memoria compartida. Entre las tres propuestas presentadas a lo largo de este capítulo para este algoritmo, la versión "divide y fusiona" consigue una aceleración superior a 12x para un sistema de 2^{20} ecuaciones. En esta versión, los datos accedidos están localizados en posiciones distantes en memoria, y esta distancia se incrementa a cada paso del algoritmo. Sin embargo, a pesar de este incremento, el número de accesos a memoria disminuye, lo que resulta en un rendimiento satisfactorio en términos de aceleración.

Los valores de GFLOPS conseguidos para todas las propuestas están lejos del rendimiento pico. Esto se debe principalmente a la baja intensidad aritmética de estos algoritmos, que da lugar a una baja utilización del *hardware* vectorial (esto es especialmente cierto para los pasos intermedios del algoritmo de reducción cíclica). Los resultados también se ven afectados por el alto número de sincronizaciones (dentro de los *kernels* y entre distintas llamadas a *kernels*), y la lógica de control asociada al flujo de trabajo de los algoritmos.

Como ya se ha mencionado anteriormente, se puede compensar el tiempo de transferencia de datos entre la CPU y la GPU si el algoritmo de resolución de sistemas tridiagonales está embebido dentro de una aplicación más grande ejecutada íntegramente en GPU. Sin embargo, una comparación de los mejores resultados considerando este tiempo de transferencia puede

Tamaño	Reducción cíclica	Doblamiento recursivo	Método de Bondeli
2^{10}	0,000105 s	0,000125 s	0,000469 s
2^{12}	0,000137 s	0,000143 s	0,000548 s
2^{14}	0,000200 s	0,000241 s	0,000847 s
2^{16}	0,000345 s	0,000761 s	0,001723 s
2^{18}	0,000992 s	0,003009 s	0,005154 s
2^{20}	0,002901 s	0,012833 s	0,018212 s
2^{22}	0,012666 s	0,056426 s	0,070763 s

(a)

Tamaño	Reducción cíclica	Doblamiento recursivo	Método de Bondeli
2^{10}	0,000192 s	0,000211 s	0,000549 s
2^{12}	0,000291 s	0,000296 s	0,000682 s
2^{14}	0,000619 s	0,000660 s	0,001199 s
2^{16}	0,001806 s	0,002222 s	0,002925 s
2^{18}	0,006642 s	0,008666 s	0,009992 s
2^{20}	0,023248 s	0,033165 s	0,034922 s
2^{22}	0,087945 s	0,131660 s	0,132505 s

(b)

Tabla 2.4: Tiempos de ejecución de varios algoritmos de resolución de sistemas tridiagonales en una tarjeta GTX 295 (a) sin tiempo de transferencia y (b) con tiempo de transferencia.

ser de interés. La tabla 2.4 muestra los tiempos de ejecución de nuestras mejores propuestas: reducción cíclica y doblamiento recursivo, ambos implementados con la técnica "divide y fusiona", y el método de Bondeli. La figura 2.34 muestra estos mismos valores en forma de gráfica. Se puede concluir que el tiempo necesario para transferir datos es prácticamente el mismo para todas las propuestas. Como se puede observar, este tiempo supone una importante sobrecarga sobre el tiempo de ejecución total.

2.6. Aplicación al cálculo de la transformada wavelet

En la sección 1.4 introdujimos algunos de los principales fundamentos teóricos de la teoría detrás de la transformada *wavelet*. En esta sección estudiaremos el problema del cálculo de la transformada en GPU.

Constante	Valor aproximado
α	$-1,586134342$
β	$-0,052980119$
γ	$+0,882911076$
δ	$+0,443506852$

Tabla 2.5: Valores aproximados de las constantes utilizadas en los pasos de *lifting* en la transformada *wavelet* Cohen-Daubechies-Feauveau (9,7).

La transformada *wavelet* es un algoritmo multinivel en el que resulta especialmente efectivo aplicar la estrategia de optimización "divide y fusiona". En esta sección presentamos aplicaciones de esta estrategia a diferentes tipos de transformada (en particular, se analizarán los esquemas basados en *lifting* y los basados en bancos de filtrado). Además, mostraremos cómo el método puede ser generalizado a dos dimensiones.

2.6.1. Transformada Cohen-Daubechies-Feauveau (9,7)

La transformada *wavelet* Cohen-Daubechies-Feauveau (9,7) [47] es ampliamente utilizada en el procesado de señal e imagen. Esta transformada puede realizarse *in place*, utilizando el conocido esquema de *lifting* [55], de forma que los resultados parciales sobrescriben los datos de entrada usados durante el cálculo. Cada nivel en esta transformada incluye cuatro pasos de *lifting*, en cada uno de los cuales se calcula un resultado parcial a partir de los datos en tres posiciones diferentes:

$$x_{i+1} = x_{i+1} + \alpha(x_i + x_{i+2}), \qquad \text{para } i = 0,2,4,\ldots \tag{2.48}$$

$$x_i = x_i + \beta(x_{i+1} + x_{i-1}), \qquad \text{para } i = 0,2,4,\ldots \tag{2.49}$$

$$x_{i+1} = x_{i+1} + \gamma(x_i + x_{i+2}), \qquad \text{para } i = 0,2,4,\ldots \tag{2.50}$$

$$x_i = x_i + \delta(x_{i+1} + x_{i-1}), \qquad \text{para } i = 0,2,4,\ldots \tag{2.51}$$

Las cuatro ecuaciones anteriores se corresponden con los pasos necesarios para ejecutar el primer nivel de la transformada (por tanto, los factores de escala han sido omitidos). Los valores de las constantes α, β, γ y δ están especificados en la tabla 2.5.

Dependiendo de la aplicación, la transformada puede componerse de uno o varios niveles. En el nivel más bajo se procesan todos los datos, pero en los siguientes niveles la cantidad de datos procesados es la mitad que en el nivel previo (una cuarta parte si la secuencia de datos es bidimensional, una octava parte en el caso tridimensional). Este proceso se denomina

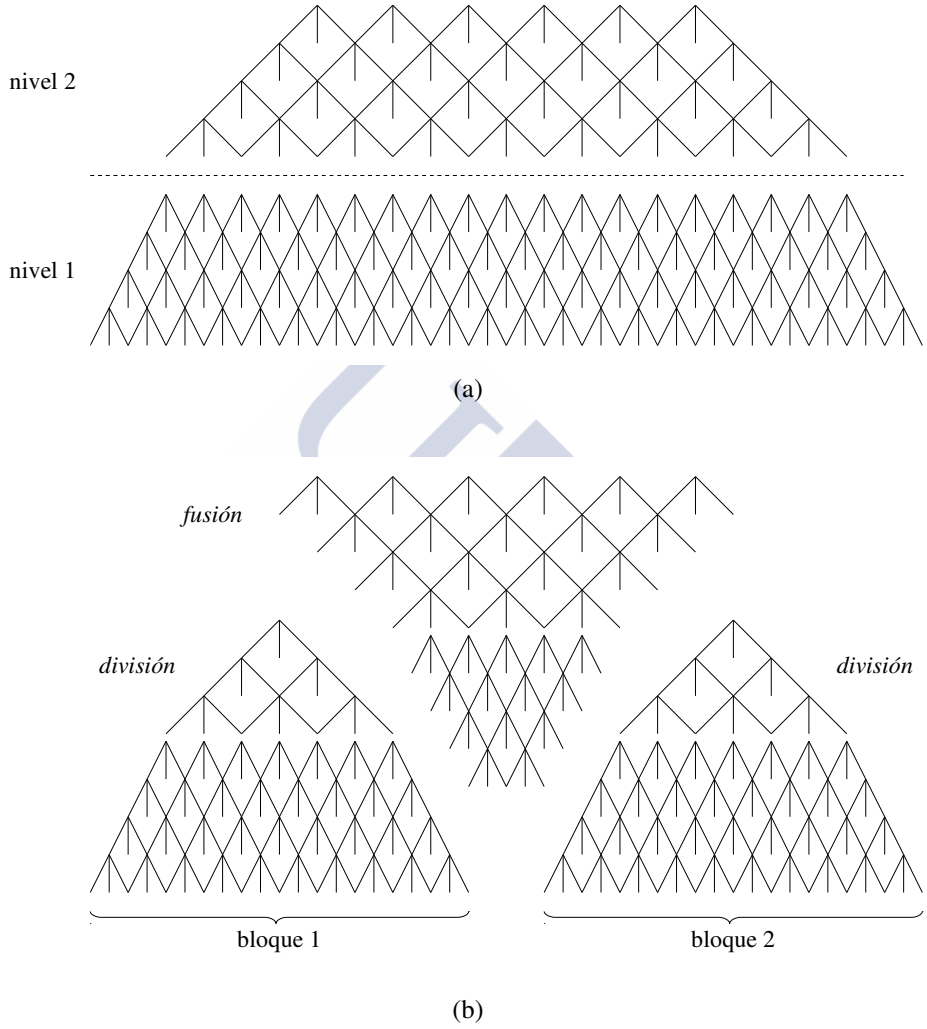


Figura 2.35: Transformada *wavelet* (9,7) de dos niveles con *lifting* donde se muestra (a) el grafo de dependencias y (b) la aplicación del método "divide y fusiona" usando dos bloques.

reducción. Los coeficientes de baja frecuencia se pasan al siguiente nivel, donde serán procesados, y los coeficientes de alta frecuencia se almacenan como resultado. A cada nuevo nivel, el cálculo de los pasos de *lifting* en sucesivos niveles requiere datos de posiciones cada vez más distantes (en las ecuaciones (2.48)-(2.51), $i = 4^k$ en el nivel 2, $i = 8^k$ en el nivel 3, etc).

```
1: para s = 0 → S paso m:
2:     < ejecutar m niveles de transformada (fase de división, en mem. compartida) >
3:     < ejecutar m niveles de transformada (fase de fusión, en mem. compartida) >

4: para cada nivel s restante:
5:     < ejecutar un nivel de transformada (fase de división, en mem. compartida) >
6:     < ejecutar un nivel de transformada (fase de fusión, en mem. compartida) >
```

Figura 2.36: Pseudocódigo de una implementación en GPU de la transformada *wavelet* (9,7) aplicando la estrategia de "divide y fusiona".

La figura 2.35(a) muestra el grafo de dependencias de una transformada *wavelet* (9,7) de dos niveles, donde en cada nivel se realizan cuatro pasos de *lifting*. La aplicación de método "divide y fusiona" en esta transformada se presenta en [89].

Hemos adaptado el método de forma que la transformada puede computarse de forma eficiente en GPU utilizando CUDA. La figura 2.36 contiene el pseudocódigo de la implementación en GPU de la transformada donde se aplica la técnica de "divide y fusiona". Las llamadas a *kernels* aparecen entre los símbolos "menor que" y "mayor que". La transformada se calcula dentro de un bucle donde se alternan las llamadas a los *kernels* que ejecutan las fases de división y de fusión. Ambos *kernels* ejecutan *m* niveles de la transformada sobre bloques de datos que se pueden computar de forma independiente y en memoria compartida. Por último, para completar los *S* niveles de transformada, pueden ser necesarios fases adicionales de división y de fusión.

En la figura 2.35(b) se muestra el grafo de dependencias de la transformada tras aplicar el método "divide y fusiona" usando dos bloques.

2.6.2. Transformada Daubechies D4

La transformada *wavelet* Daubechies D4 [54] utiliza dos filtros de longitud 4, de forma que en cada filtro recoge cuatro datos de entrada y genera dos resultados parciales. A continuación, los filtros se mueven dos posiciones y generan dos nuevos resultados parciales.

Esta transformada se puede calcular utilizando varios pasos de *lifting* como en el caso anterior. Sin embargo, en esta ocasión vamos a utilizar la versión con bancos de filtrado, que se resuelve con el siguiente par de ecuaciones:

$$x_i = h_0 x_i + h_1 x_{i+1} + h_2 x_{i+2} + h_3 x_{i+3}, \qquad \text{para } i = 0,2,4,\ldots \tag{2.52}$$

$$x_{i+1} = h_3 x_i - h_2 x_{i+1} + h_1 x_{i+2} - h_0 x_{i+3}, \qquad \text{para } i = 0,2,4,\ldots \tag{2.53}$$

Constante	Valor
h_0	$\frac{1+\sqrt{3}}{4\sqrt{2}} \approx +0,482962913$
h_1	$\frac{3+\sqrt{3}}{4\sqrt{2}} \approx +0,836516304$
h_2	$\frac{3-\sqrt{3}}{4\sqrt{2}} \approx +0,224143868$
h_3	$\frac{1-\sqrt{3}}{4\sqrt{2}} \approx -0,129409523$

Tabla 2.6: Valores de las constantes utilizadas en la transformada *wavelet* Daubechies D4.

Donde h_0, h_1, h_2 y h_3 son constantes cuyos valores aparecen especificados en la tabla 2.6.

La figura 2.37(a) muestra el grafo de dependencias para una transformada D4 de tres niveles. El esquema de computación de la transformada mediante bancos de filtrado da lugar a un algoritmo que no es *in place*. Cuando la transformada se calcula en paralelo utilizando CUDA, los resultados parciales no pueden sobrescribir los datos de entrada, por lo que se requiere el doble de memoria.

A pesar de que esta transformada *wavelet* sigue una estrategia diferente, la aplicación del método "divide y fusiona" es similar al caso de la *wavelet* (9,7) con *lifting* estudiada en la sección anterior, por lo que omitimos el pseudocódigo. Como particularidad, dado que este algoritmo no es *in place*, hay que tener especial cuidado en no sobrescribir los resultados parciales calculados durante la fase de división que serán luego necesarios durante la fase de fusión. Estos resultados deben transferirse desde la memoria compartida a la memoria global cuando la fase de división termine. La figura 2.37(b) muestra la aplicación del método "divide y fusiona" utilizando dos bloques.

2.6.3. Transformada paquete wavelet

En los algoritmos vistos hasta ahora el número de datos se reduce en cada nivel. Algunas transformadas paquete *wavelet*, por contra, no realizan ninguna reducción, es decir, el número de computaciones se mantiene constante en todos los niveles. Las transformadas paquete *wavelet* se utilizan para calcular la mejor base, esto es, la representación mínima de los datos relativa a una función particular. Este tipo de transformadas tienen aplicaciones en problemas de reducción de ruido y compresión de datos [120, 176].

En las transformadas paquete *wavelet* multinivel la distancia entre las posiciones de los datos de entrada crece exponencialmente (como potencias de dos) en cada nuevo nivel, como

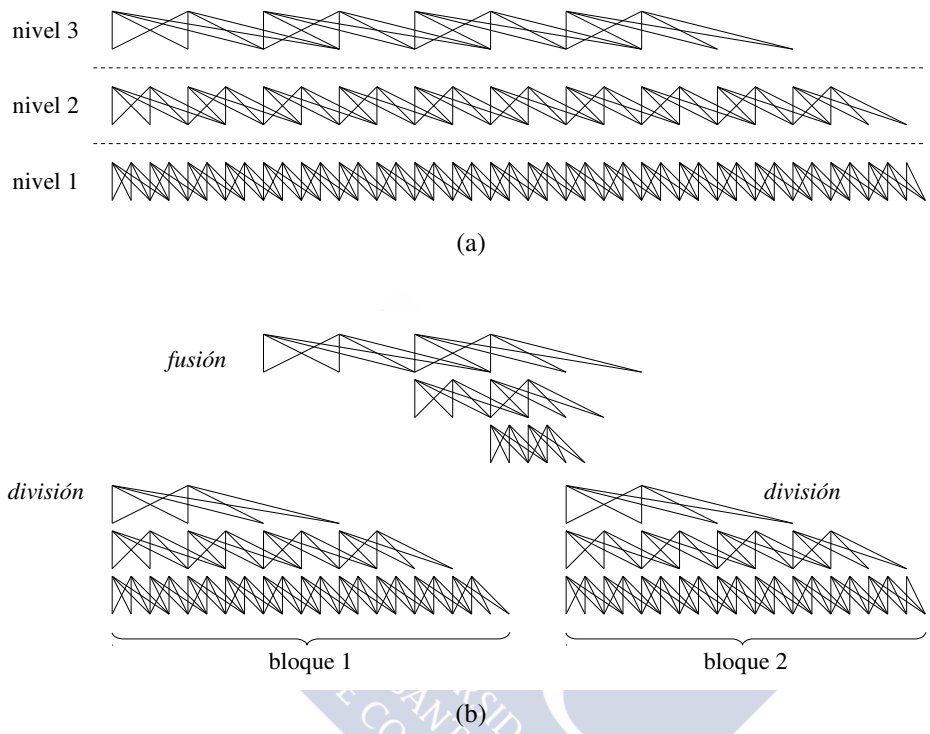


Figura 2.37: Transformada *wavelet* D4 de tres niveles donde se muestra (a) el grafo de dependencias y (b) la aplicación del método "divide y fusiona" usando dos bloques.

también ocurría en las transformada anteriores. En la figura 2.38(a) se muestra el grafo de dependencias de una transformada paquete *wavelet* basada en la D4 con bancos de filtrado.

Aunque la transformada es diferente, la aplicación del método "divide y fusiona" es similar a los casos anteriores, por lo que de nuevo omitimos el pseudocódigo. Dado que este esquema de computación no es *in place*, se aplican las mismas consideraciones antes mencionadas para la implementación de la *wavelet* D4. La figura 2.38(b) muestra un ejemplo de aplicación del método "divide y fusiona" utilizando dos bloques.

2.6.4. Extensión del método "divide y fusiona" a las dos dimensiones

El método de "divide y fusiona" puede extenderse de forma sencilla al caso de dos dimensiones. En esta sección analizamos el rendimiento de múltiples esquemas de partición de

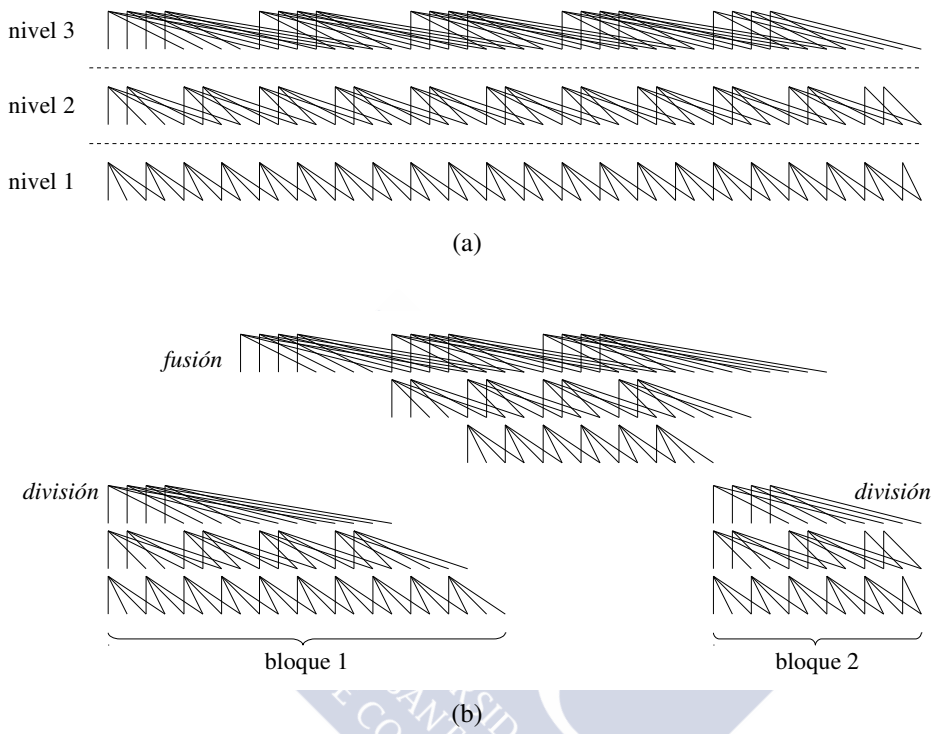


Figura 2.38: Transformada *wavelet* de paquete D4 de dos niveles donde se muestra (a) el grafo de dependencias y (b) la aplicación del método "divide y fusiona" usando dos bloques. Por claridad, solo se muestran la mitad de las dependencias.

una secuencia bidimensional en GPU. Como modelo de algoritmo multinivel bidimensional consideramos la transformada *wavelet* 2D. El esquema básico de computación de la transformada de datos bidimensionales es la descomposición en filas y columnas. Este esquema se ejecuta en dos fases: una fase inicial que calcula la transformada unidimensional de cada una de las filas, y una segunda fase que hace lo mismo con cada una de las columnas. Un esquema alternativo, conocido como descomposición cuadrada, alterna y combina computaciones de filas y columnas. Cuando la transformada 2D se realiza en la GPU el factor más influyente en el rendimiento es la distribución de los cálculos entre los bloques de hilos.

La figura 2.39 muestra tres posibles esquemas de partición de la matriz bidimensional de datos entre los bloques de hilos para una implementación en GPU que utilice memoria global. El primer esquema requiere una fase de partición inicial de las filas en bloques hori-

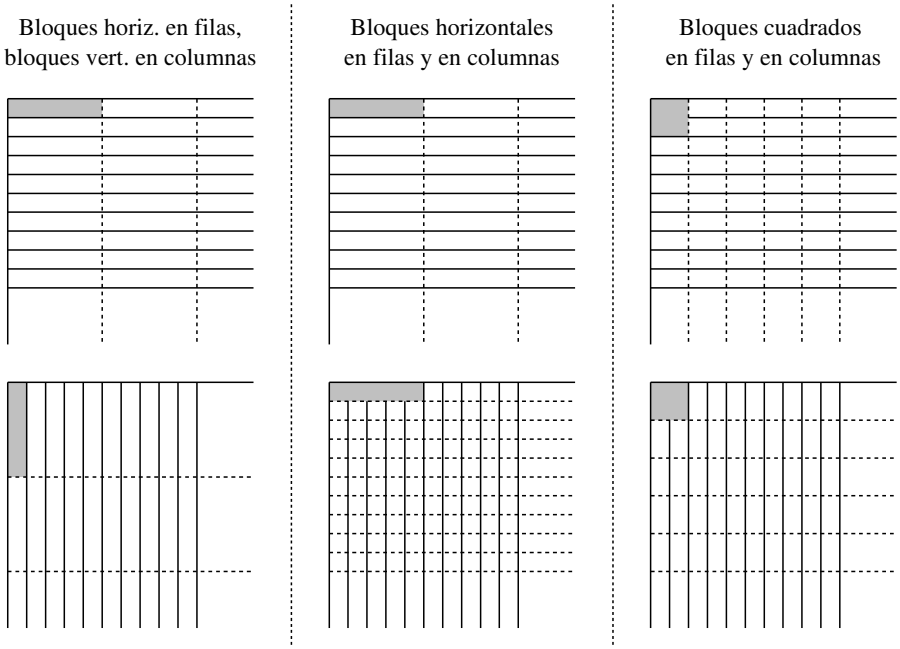


Figura 2.39: Tres esquemas de partición diferentes para una transformada *wavelet* 2D con descomposición en filas y columnas en la GPU utilizando memoria global.

zontales seguida de una segunda fase de partición de las columnas en bloques verticales. Los rectángulos sombreados en la figura muestran cada uno de los bloques. Este particionamiento proporciona un rendimiento bajo ya que los bloques verticales tienen un patrón de accesos a memoria altamente ineficiente.

La segunda aproximación que se muestra en la figura resuelve este problema realizando la partición en bloques de la segunda fase de la misma forma que la primera, esto es, utilizando bloques horizontales también para las columnas. En la fase de transformación de filas, cada bloque ejecutará la transformación parcial de una fila; sin embargo, en la fase de transformación de columnas, cada bloque calculará en paralelo el resultado parcial de un conjunto de columnas. Así, los hilos de cada bloque realizan accesos a memoria manteniendo la coalescencia y minimizando la distancia entre los accesos [129].

Por último, la tercera aproximación utiliza bloques cuadrados del mismo tamaño en ambas fases como compromiso entre facilidad de implementación y eficiencia. Dependiendo

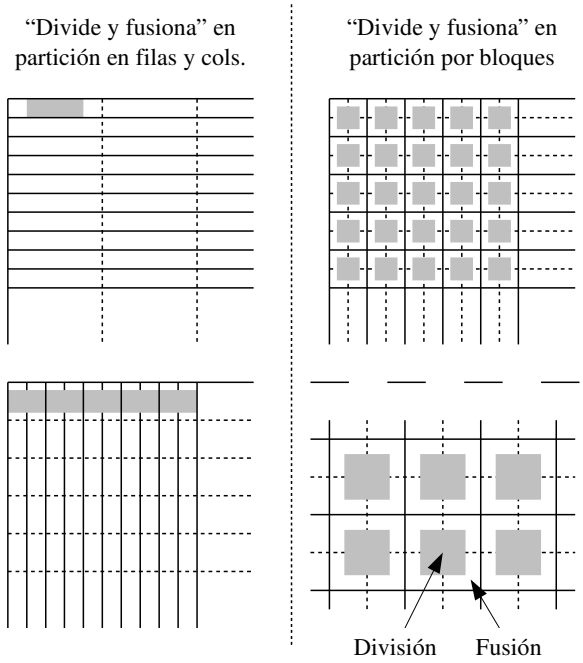


Figura 2.40: Dos esquemas de partición para una transformada *wavelet* 2D en GPU utlizando el método de "divide y fusiona".

del tamaño de los bloques, la eficiencia de este esquema puede ser similar a la del esquema anterior.

La figura 2.40 muestra un ejemplo de aplicación del método "divide y fusiona" para los dos últimos esquemas de descomposición mencionados. En la parte izquierda de la figura el método "divide y fusiona" se aplica a una descomposición en filas y columnas que utiliza bloques horizontales para las filas y bloques cuadrados para las columnas. Durante la transformación de filas las fases de división y fusión se llevan a cabo en memoria compartida al igual que cuando los datos son unidimensionales. La transformación de columnas se realiza en bloques cuadrados, utilizando el esquema "divide y fusiona" en paralelo en múltiples columnas.

En la parte derecha de la figura 2.40 se implementa una descomposición por bloques, donde la matriz de datos bidimensional se parte en bloques cuadrados, de forma que la fase de división procesa los datos en la memoria compartida de cada bloque, transformando filas y

```
1: para s = 0 → S paso m:
2:     < ejecutar m niveles de transformada por bloques (fase de división, en mem. compartida) >
3:     < ejecutar m niveles de transformada por filas (fase de fusión, en mem. compartida) >
4:     < ejecutar m niveles de transformada por columnas (fase de fusión, en mem. compartida) >

5: para cada nivel s restante:
6:     < ejecutar un nivel de transformada por bloques (fase de división, en mem. compartida) >
7:     < ejecutar un nivel de transformada por filas (fase de fusión, en mem. compartida) >
8:     < ejecutar un nivel de transformada por columnas (fase de fusión, en mem. compartida) >
```

Figura 2.41: Pseudocódigo de una implementación en GPU de la transformada *wavelet* bidimensional por bloques aplicando la estrategia de "divide y fusiona".

columnas. A continuación, se calculan dos fases de fusión: la primera completa la transformación de las filas, y la segunda la transformación de las columnas. La figura 2.41 muestra el pseudocódigo de esta implementación.

2.6.5. Resultados

En esta sección presentamos los resultados del implementación del método "divide y fusiona" en la GPU de los diferentes tipos de transformada *wavelet* considerados en este capítulo.

Metodología

Hemos evaluado nuestras implementaciones en tres GPU de NVIDIA diferentes, GT 9800, GTX 295 y GTX 480, cuyas características pueden encontrarse en la sección 1.7. En el caso de la GTX 480, debe tenerse en cuenta que esta arquitectura posee dos niveles de caché, y puede configurarse de forma que cada multiprocesador dedique 16 kB de memoria compartida y 48 KB de caché L1, o viceversa.

Las medidas de rendimiento fueron obtenidas como una media de mil ejecuciones para cada algoritmo. El tiempo asociado con la transferencia de datos entre la CPU y la GPU, que es el mismo para todos los algoritmos, no está incluido en los tiempos de ejecución. Como ya se ha comentado anteriormente, este tiempo puede ser ignorado si los algoritmos forman parte de una aplicación más grande que se ejecuta íntegramente en GPU [189].

Tamaño	Directa	Divide y fusiona	Aceleración
2^{10}	0,121 ms	0,021 ms	4,84x
2^{12}	0,140 ms	0,025 ms	5,60x
2^{14}	0,139 ms	0,033 ms	4,21x
2^{16}	0,196 ms	0,063 ms	3,11x
2^{18}	0,533 ms	0,146 ms	3,65x
2^{20}	1,940 ms	0,420 ms	4,01x

Tabla 2.7: Tiempos de ejecución y valores de aceleración en una una tarjeta GTX 480 para la implementación de la transformada *wavelet* Cohen-Daubechies-Feauveau (9,7) de 8 niveles con *lifting*.

Tamaño	Directa	Divide y fusiona	Aceleración
2^{10}	0,024 ms	0,023 ms	1,04x
2^{12}	0,043 ms	0,032 ms	1,34x
2^{14}	0,055 ms	0,049 ms	1,18x
2^{16}	0,066 ms	0,056 ms	1,12x
2^{18}	0,225 ms	0,128 ms	1,78x
2^{20}	0,864 ms	0,451 ms	1,92x

Tabla 2.8: Tiempos de ejecución y valores de aceleración en una una tarjeta GTX 480 para la implementación de la transformada *wavelet* D4 de 8 niveles con bancos de filtrado.

Tamaño	Directa	Divide y fusiona	Aceleración
2^{10}	0,026 ms	0,019 ms	1,37x
2^{12}	0,028 ms	0,011 ms	1,54x
2^{14}	0,032 ms	0,022 ms	1,45x
2^{16}	0,060 ms	0,040 ms	1,50x
2^{18}	0,211 ms	0,140 ms	1,51x
2^{20}	0,752 ms	0,540 ms	1,40x

Tabla 2.9: Tiempos de ejecución y valores de aceleración en una una tarjeta GTX 480 para la implementación de la transformada paquete *wavelet* D4 de 4 niveles con bancos de filtrado.

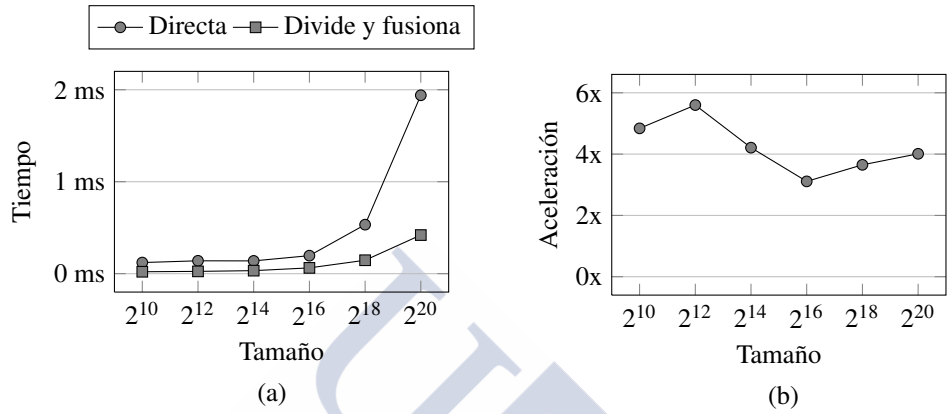


Figura 2.42: Medidas de rendimiento separadas en (a) tiempo consumido y (b) aceleración en una una tarjeta GTX 480 para la implementación de la transformada *wavelet* Cohen-Daubechies-Feauveau (9,7) de 8 niveles con *lifting*.

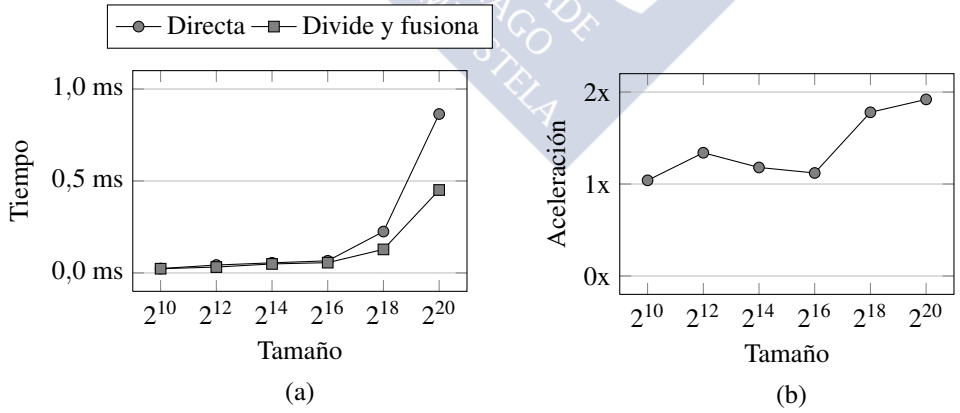


Figura 2.43: Medidas de rendimiento separadas en (a) tiempo consumido y (b) aceleración en una una tarjeta GTX 480 para la implementación de la transformada *wavelet* D4 de 8 niveles con bancos de filtrado.

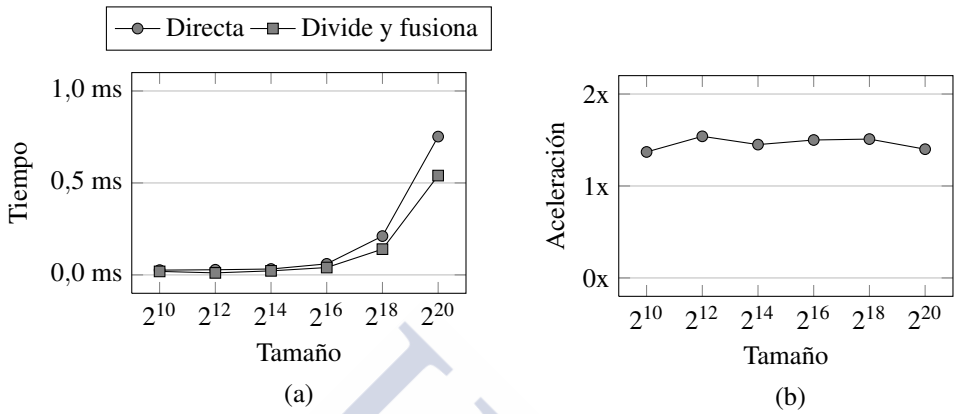


Figura 2.44: Medidas de rendimiento separadas en (a) tiempo consumido y (b) aceleración en una una tarjeta GTX 480 para la implementación de la transformada paquete *wavelet* D4 de 4 niveles con bancos de filtrado.

Medidas de rendimiento

Las tablas 2.7, 2.8 y 2.9 muestran los tiempos de ejecución y los valores de aceleración para las implementaciones en una GTX 480 de los algoritmos descritos. Las figuras 2.42, 2.43 y 2.44 muestran gráficas con los mismos datos, donde comparamos la implementación directa en memoria global con la implementación utilizando el esquema "divide y fusiona" para diferentes tamaños de problema. Mostramos los resultados obtenidos para la transformada *wavelet* (9,7) con *lifting*, la transformada *wavelet* D4 con bancos de filtrado y la transformada paquete *wavelet* D4. Para un problema de tamaño 2^{20} los valores de aceleración obtenidos han sido 4,0x, 1,9x y 1,4x, respectivamente. Como ejemplo, tomando como base la implementación secuencial en CPU (en un núcleo de una Intel Q9450 a 2,66 GHz en Linux, compilada con `gcc` con optimización O3), los valores de aceleración de las versiones con "divide y fusiona" son 35,7x, 25,6x y 55,6x, respectivamente.

Hemos estudiado en detalle los parámetros que pueden ser ajustados en el método de "divide y fusiona" para mejorar la eficiencia de la implementación. La tabla 2.10 muestra, para diferentes valores de los parámetros, los tiempos de ejecución de una transformada *wavelet* (9,7) de 8 niveles de tamaños 2^{20}. Las primeras dos filas de la tabla muestran los tiempos de ejecución de implementaciones directas que utilizan únicamente la memoria global de la GPU. Los valores de aceleración se han calculando tomando como referencia esta implementación directa.

Implementación	N	División	Fusión	H	GT 9800	GTX 295	GTX 470
Global	-	-	-	1	12,63 (1,00x)	4,26 (1,00x)	1,94 (1,00x)
Global	-	-	-	1/2	16,06 (0,79x)	3,95 (1,08x)	1,10 (0,97x)
Divide y fusiona	1	Comp.	Comp.	1	2,75 (4,59x)	0,95 (4,49x)	0,63 (3,11x)
Divide y fusiona	1	Comp.	Comp.	1/2	2,41 (5.25x)	0,81 (5.29x)	0,53 (3,66x)
Divide y fusiona	1	Comp.	Global	1	3,48 (3,63x)	1,37 (3,16x)	0,85 (2,30x)
Divide y fusiona	1	Comp.	Global	1/2	2,86 (4,42x)	1,08 (3,94x)	0,64 (3,03x)
Divide y fusiona	2	Comp.	Comp.	1	2,39 (5.29x)	0,92 (4,62x)	0,53 (3,70x)
Divide y fusiona	2	Comp.	Comp.	1/2	1,70 (7,46x)	0,72 (5.93x)	0,42 (4,60x)
Divide y fusiona	2	Comp.	Global	1	3,55 (3,56x)	1,44 (2,95x)	0,87 (2,24x)
Divide y fusiona	2	Comp.	Global	1/2	2,36 (5.35x)	1,05 (4,04x)	0,58 (3,36x)
Divide y fusiona	4	Comp.	Comp.	1	4,46 (2,83x)	1,61 (2,64x)	0,72 (2,72x)
Divide y fusiona	4	Comp.	Comp.	1/2	2,75 (4,60x)	1,05 (4,07x)	0,49 (3,94x)
Divide y fusiona	4	Comp.	Global	1	5,56 (2,27x)	2,10 (2,03x)	1,27 (1,53x)
Divide y fusiona	4	Comp.	Global	1/2	3,37 (3,75x)	1,46 (2,92x)	0,74 (2,61x)

Tabla 2.10: Tiempos de ejecución en milisegundos y valores de aceleración de las diferentes implementaciones en GPU de la *wavelet* (9,7) para un tamaño de 2^{20}, 8 niveles con *lifting*, y utilizando bloques de 256 hilos.

Como ya se ha explicado, cuando el número de niveles es alto, puede ser más eficiente aplicar el método de "divide y fusiona" varias veces de forma sucesiva. En la tabla 2.10, la columna N especifica el número de niveles cubierto en cada fase de "divide y fusiona". En el caso de la transformada *wavelet* (9,7), cuando $N = 1$ cada fase de división y fusión cubre un solo nivel, por lo que son necesarias ocho fases de división y fusión para completar los ocho niveles de la transformada; en cambio, cuando $N = 2$, cuatro fases de división y fusión de dos niveles cada una son suficientes; y cuando $N = 4$, la transformada se completa en solo dos fases de división y fusión de cuatro niveles cada una. El tiempo de computación depende del número de computaciones realizadas en cada nivel y del número de datos a los que se debe acceder en memoria. Para este caso, el menor tiempo de computación se alcanza cuando $N = 2$.

La fase de fusión realiza las computaciones que no pudieron ser ejecutadas durante la fase de división con los datos contenidos en cada bloque. A menudo la fase de fusión realiza menos operaciones que la fase de división, por lo que puede ser ventajoso ejecutar esta fase en la memoria global. En la tabla 2.10, las columnas tercera y cuarta identifican las fases de división y fusión, respectivamente, dependiendo de si han sido ejecutadas en la memoria

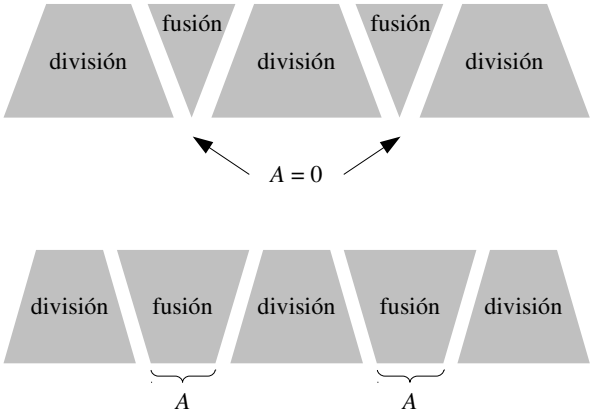


Figura 2.45: Dos posibles tamaños para las fases de división y fusión. La parte superior de la figura muestra el caso donde el ancho de la fase de fusión es $A = 0$. La parte inferior muestra el caso con $A \neq 0$.

global y la memoria compartida. En casi todos estos casos, se consiguen mejores tiempos de computación ejecutando la fase de fusión en la memoria compartida.

En algunos algoritmos multinivel cada etapa de computación genera un número de resultados parciales que es la mitad del número de datos de entrada. Así ocurre, por ejemplo, en la transformada *wavelet* (9,7) con *lifting* y en la reducción cíclica. En una implementación directa de estos algoritmos en GPU, mientras todos los hilos cooperan para acceder a datos de memoria global, solo la mitad de los hilos realizan computaciones. En algunos casos el rendimiento puede incrementarse reduciendo a la mitad el número de hilos, de forma que cada hilo realice el doble de accesos a memoria pero manteniendo, a la vez, el número de operaciones aritméticas ejecutadas. La secuencialización de los accesos a memoria (que de otra forma estaría limitada por el ancho del banda del bus de memoria) puede ser compensada con la ausencia de hilos ociosos. En las tabla 2.10, la columna H indica si el número de hilos es el mismo que en la implementación original ($H = 1$) o si el número de hilos se ha reducido a la mitad ($H = 1/2$). En prácticamente todos los casos se observa una mejora de eficiencia cuando se aplica esta estrategia.

Hasta ahora hemos considerado que la fase de división realiza el máximo número de computaciones que pueden realizarse con los datos contenidos dentro de cada bloque, y que la fase de fusión ejecuta las operaciones restantes. Esta no es la única alternativa, ya que existe

A	GT 9800	GTX 295	GTX 480
0	1,70	0,72	0,42
16	1,97	0,74	0,43
32	2,03	0,71	0,43
64	2,18	0,70	0,41
128	2,35	0,71	0,38

Tabla 2.11: Tiempos de ejecución (en milisegundos) para diferentes anchuras de las fases de división y fusión en la implementación de la trasformada *wavelet* Daubechies (9,7).

Tipo	Filas	Columnas	GT 9800	GTX 295	GTX 480
Global	256×1	1×256	572,05 (1,00x)	132,70 (1,00x)	20,32 (1,00x)
Global	256×1	256×1	44,52 (12,9x)	11,92 (11,1x)	4,94 (4,12x)
Global	16×16	16×16	44,52 (12,8x)	10,80 (12,3x)	5,21 (3,90x)
Divide y fusiona	256×1	16×16	12,47 (45,9x)	4,33 (30,7x)	2,25 (9,03x)
Divide y fusiona	16×16	16×16	14,95 (38,3x)	3,77 (35,2x)	2,43 (8,41x)

Tabla 2.12: Tiempos de ejecución en milisegundos y valores de aceleración para la transformada *wavelet* (9,7) de tamaño 2048×2048 con 4 niveles de *lifting* calculada en GPU utilizando bloques de 256 hilos.

también la posibilidad de mover parte de las computaciones de la fase de división a la fase de fusión. La figura 2.45 muestra cómo el parámetro A ajusta las anchuras de las fases de división y fusión. Los tiempos de ejecución para diferentes valores de A se muestran en la tabla 2.11, donde se puede ver que los tiempos obtenidos son similares en la mayor parte de los casos. Este resultado entra dentro de los esperado, dado que el número de operaciones es siempre el mismo, y lo único que cambia es cómo se distribuyen las operaciones entre las fases de división y fusión. Por ejemplo, para la transformada *wavelet* (9,7) calculada en una GTX 295 con $A = 0$, la sección de división consume 640 μs y la sección de fusión 82 μs, mientras que con $A = 128$, estos tiempos son 491 μs y 220 μs, respectivamente.

La tabla 2.12 muestra los tiempos de ejecución obtenidos para las diferentes implementaciones de la transformada *wavelet* bidimensional con "divide y fusiona". La geometría de cada bloque durante las fases de transformación de filas y columnas aparece especificada en las columnas segunda y tercera, respectivamente. Puede verse que los esquemas de "divide y fusiona" usados ofrecen un rendimiento similar. En la GTX 480 el valor de aceleración es menor que las otras GPU debido a que este modelo dispone de una caché L1 que permite la reutilización de los datos incluso en algoritmos que no hacen uso de la memoria compartida.

wavelet	[174]	[65]	"Divide y fusiona"
D4, EBF, 5 niveles	9,12	-	8,53
D4, EL, 5 niveles	17,9	-	9,67
(9,7), EBF, 5 niveles	16,5	-	16,15
(9,7), EL, 5 niveles	20,7	-	14,37
D4, EBF, 1 nivel	-	4,41	3,77
D4, EL, 1 nivel	-	8,05	6,02

Tabla 2.13: Tiempos de ejecución en milisegundos para transformadas *wavelet* bidimensionales de tamaño 4×10^6 píxeles. Las GPU utilizadas son: 7800 GTX en [174], C870 en [65], y 8800 GTS para "divide y fusiona".

Comparación con otros trabajos en la literatura

La implementación de la transformada *wavelet* en GPU modernas ha sido explorada en varios trabajos. En [174] se presentan implementaciones de cinco familias de *wavelet* comparando el rendimiento de los esquemas basados en bancos de filtrado y los basados en *lifting*. El análisis está enfocado estrictamente en datos bidimensionales utilizando una estrategia de descomposición en filas y columnas. Se utiliza la API de gráficos 3D OpenGL para organizar los datos en *streams*, que son cargados en la memoria de la GPU como texturas 2D para luego ser procesados por *kernels* implementados como *fragment shader* (implementados en Cg). En el esquema de bancos de filtrado, los *kernels* horizontales leen la mitad superior del espacio de texturas reservado en la memoria de la GPU y escriben en la mitad inferior; de esta forma, la imagen queda dividida en las columnas interiores y los bordes a la derecha y a la izquierda. El filtrado vertical se realiza de forma análoga al horizontal, pero en este caso la imagen es dividida en filas interiores y bordes superior e inferior. El muestreo se realiza especificando áreas de entrada que son el doble de grandes que las de salida. En el esquema basado en *lifting*, cada paso de *lifting* es ejecutado por un *kernel* diferente, y la CPU encadena y serializa las llamadas a estos *kernels* de acuerdo a las dependencias entre datos.

En la tabla 2.13 muestra los tiempos de ejecución obtenidos en [174] para una transformada *wavelet* usando el esquema basado en *lifting* (EL) y el esquema basado en bancos de filtrado (EBF). La tabla incluye los tiempos obtenidos por nuestra implementación, aunque las condiciones de ejecución no han sido exactamente las mismas. En [174] los experimentos fueron realizados en una NVIDIA 7800 GTX (el modelo más potente de la 7ª generación de las GeForce) y el modelo de programación está basado en OpenGL y Cg. Dado que esta generación de GPU no soporta CUDA, la tabla muestra los tiempos obtenidos en una NVIDIA 8800 GTS

(que es segundo modelo más potente de la 8ª generación). Los experimentos han consistido en calcular cinco niveles de transformación en matrices de 4 Mpíxeles (2048×2048). El algoritmo basado en "divide y fusiona" es ligeramente más eficiente en el caso de los bancos de filtrado, y mucho más eficiente que en el caso de *lifting* gracias al ahorro de un gran número de accesos a memoria global.

En [65] se calcula la transformada *wavelet* 2D utilizando una descomposición de filas y columnas con una transposición de matriz como paso intermedio entre la transformación de filas y la de columnas. Esto resuelve el problema de los accesos no coalescentes a memoria global durante la procesado de las columnas. También se lleva a cambo una comparación entre los esquemas de banco de filtros y de *lifting*. La implementación, realizada en CUDA, utiliza un *kernel* por cada paso de *lifting*. La tabla 2.13 muestra los tiempos de ejecución obtenidos por los autores en una NVIDIA Tesla C870 (una GPU de alta gama con el procesador G80). Comparamos estos tiempos de ejecución con los obtenidos por nuestra implementación en una NVIDIA 8800 GTS, que es un modelo de consumo de la misma generación con un rendimiento ligeramente menor. Los experimentos consisten en un nivel de transformación en matrices de tamaño 2048×2048. Nuestra implementación utilizando "divide y fusiona" ofrece un rendimiento mejor.

2.7. Conclusiones

En este capítulo hemos descrito algunos de los patrones de paralelismo más comunes en la implementación de soluciones en GPU, y que han sido utilizados en el desarrollo de este trabajo: paralelismo de bucle, divergencia/unión, descomposición geométrica, y "divide y vencerás". Además, hemos presentado un patrón de paralelismo, denominado "divide y fusiona", que permite implementar de forma eficiente algoritmos multinivel donde las dependencias en las operaciones dificultan una división directa de las tareas para su ejecución en paralelo.

El método "divide y fusiona" permite resolver el problema de la dependencia de datos en los bordes de los bloques en el modelo de programación paralela CUDA. De esta forma, es posible explotar eficientemente los miles de hilos que funcionan en paralelo en la GPU y hacer un uso extensivo de la memoria compartida, que es más rápida que la memoria global.

Realizamos, además, varias propuestas de implementación en GPU de algoritmos de resolución de sistemas tridiagonales de ecuaciones lineales y algoritmos para calcular la transformada *wavelet*. Se trata en ambos casos de algoritmos multinivel donde una división de las

operaciones entre los multiprocesadores de la GPU que maximice la eficiencia no es trivial. Proponemos varias implementaciones utilizando los patrones mencionados, y demostramos que es posible explotar los recursos de la GPU (en particular, la jerarquía de memoria) utilizando el patrón "divide y fusiona" para obtener resultados satisfactorios, incluso en problemas con baja intensidad aritmética.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Capítulo 3

Algoritmo de segmentación basado en el esquema de división del operador aditivo

3.1. Introducción

En este capítulo proponemos una implementación en GPU de un método de segmentación de volúmenes basado en el conjunto de nivel. Este método se caracteriza por aproximar el cálculo la evolución del conjunto de nivel mediante el esquema de división del operador aditivo (en inglés AOS, *additive operator splitting*). Este esquema requiere durante sus pasos intermedios la resolución de varios sistemas tridiagonales, por lo que utilizaremos la técnica de "divide y fusiona", que hemos analizado en el capítulo 2, para proponer una implementación eficiente en GPU de este método de segmentación.

El esquema de división del operador aditivo es un método numérico propuesto inicialmente para el cálculo de la difusión no lineal en el procesado de imágenes [184]. En [183] se demuestra que el método puede ser utilizado para estudiar la evolución del conjunto de nivel con aplicación en la segmentación de imágenes: la evolución del conjunto de nivel se aproxima mediante la resolución de sistemas tridiagonales de ecuaciones lineales que se construyen a partir de las relaciones entre cada elemento de la imagen y sus vecinos. En [71] se presenta una variación del esquema basada en los métodos rápidos de avance y de banda estrecha de conjunto de nivel, que ya habían sido descritos en [156, 157]. Por último, en [92] se describe

una aplicación práctica del método AOS para segmentar la aorta a partir de la segmentación bidimensional de cada lámina de un volumen.

La resolución de los sistemas de ecuaciones es el principal paso para calcular la evolución del conjunto de nivel, pero para construir dichos sistemas de ecuaciones es necesario ejecutar otras operaciones, como el cálculo del gradiente. Hemos implementado todas estas operaciones íntegramente en la GPU. En concreto, hemos escogido una implementación basada en aplicar la técnica de "divide y fusiona" al algoritmo de reducción cíclica para integrarla en esta solución de segmentación en GPU.

La solución AOS que presentamos en este capítulo tiene las siguientes características:

- Consideramos el uso de esta técnica en el dominio tridimensional para la segmentación de datos volumétricos.
- Dividimos el dominio en regiones tridimensionales de tamaño fijo, de forma que la computación puede repartirse de forma natural entre los procesadores de la GPU eficientemente.
- Mantenemos una lista de regiones activas, de forma que la computación se restringe únicamente a aquellas regiones que están más cerca del frente.
- La resolución de los sistemas de ecuaciones se realiza mediante una implementación óptima en GPU del algoritmo de reducción cíclica con la técnica "divide y fusiona".

El resto de este capítulo está organizado como sigue. La sección 3.2 detalla los fundamentos teóricos del esquema AOS. La sección 3.3 presenta nuestra propuesta de implementación en GPU y la sección 3.4 muestra las medidas de rendimiento y calidad. Por último, la sección 3.5 termina el capítulo con las conclusiones.

3.2. Esquema de división del operador aditivo

En el capítulo 1 presentamos dos modelos de contorno activo, el modelo geométrico y el modelo geodésico, obtenidos a partir de la minimización de una función de energía (véase sección 1.3.3). La siguiente ecuación generaliza ambos modelos añadiendo dos nuevas funciones a y b:

$$\frac{\partial u}{\partial t} = a(\mathbf{x})|\nabla u|\mathrm{div}\left(\frac{b(\mathbf{x})}{|\nabla u|}\nabla u\right) + k\,g(\mathbf{x})|\nabla u|, \tag{3.1}$$

donde $u(\mathbf{x},t) : \mathbb{R}^n \to \mathbb{R}$ es la función del conjunto de nivel, $g : \mathbb{R}^n \to [0,1)$ es la función de parada y k es una constante real que determina el peso de la "fuerza globo". Al establecer $a := g$ y $b := 1$ se obtiene el modelo geométrico, mientras que con $a := 1$ y $b := g$ se obtiene el modelo geodésico. Por simplicidad, asumamos que la constante k es igual a cero, de tal forma que la ecuación de evolución del conjunto de nivel queda definida como:

$$\frac{\partial u}{\partial t} = a(\mathbf{x})|\nabla u|\mathrm{div}\left(\frac{b(\mathbf{x})}{|\nabla u|}\nabla u\right). \tag{3.2}$$

Para derivar un algoritmo de esta ecuación, es necesario realizar discretizaciones en el espacio y en el tiempo. Consideramos que un volumen es una malla tridimensional de puntos (o nodos) discretos con distancia $h = 1$ entre dos puntos adyacentes en todas las dimensiones. Por otra parte, la discretización en el tiempo nos lleva a considerar instantes $t_n = n\tau$, donde $n = 0,1,2,\ldots$ y τ es la duración de cada paso de tiempo. Así, u^n_{ijk} denota la aproximación discreta de $u(i\cdot h, j\cdot h, k\cdot h, t_n)$.

En primer lugar, discretizamos el término $\mathrm{div}\left(\frac{b(\mathbf{x})}{|\nabla u|}\nabla u\right)$ considerando $c = \frac{b(\mathbf{x})}{|\nabla u|}$. Dicho término puede ser aproximado de la siguiente forma:

$$\begin{aligned}
\mathrm{div}(c\nabla u) \approx\ & \partial_x\left(c_{ijk}\frac{u_{i+\frac{1}{2},j,k} - u_{i-\frac{1}{2},j,k}}{h}\right) + \partial_y\left(c_{ijk}\frac{u_{i,j+\frac{1}{2},k} - u_{i,j-\frac{1}{2},k}}{h}\right) + \\
& \partial_z\left(c_{ijk}\frac{u_{i,j,k+\frac{1}{2}} - u_{i,j,k-\frac{1}{2}}}{h}\right) \\
\approx\ & c_{i+\frac{1}{2},j,k}\frac{u_{i+1,j,k} - u_{i-1,j,k}}{h^2} + c_{i-\frac{1}{2},j,k}\frac{u_{i+1,j,k} - u_{i-1,j,k}}{h^2} + \\
& c_{i,j+\frac{1}{2},k}\frac{u_{i,j+1,k} - u_{i,j-1,k}}{h^2} + c_{i,j-\frac{1}{2},k}\frac{u_{i,j+1,k} - u_{i,j-1,k}}{h^2} + \\
& c_{i,j,k+\frac{1}{2}}\frac{u_{i,j,k+1} - u_{i,j,k-1}}{h^2} + c_{i,j,k-\frac{1}{2}}\frac{u_{i,j,k+1} - u_{i,j,k-1}}{h^2}.
\end{aligned} \tag{3.3}$$

Los valores $c_{i\pm\frac{1}{2},j,k}$, $c_{i,j\pm\frac{1}{2},k}$ y $c_{i,j,k\pm\frac{1}{2}}$ pueden calcularse por interpolación lineal. Para simplificar la notación, consideramos el volumen u como un vector lineal, de forma que el elemento i-ésimo del vector se corresponde con el punto x_i de la malla, y u^n_i denota la aproximación de $u(x_i,t_n)$. De esta forma, la ecuación 3.2 puede reescribirse con una formulación semi-implícita [183]:

$$u^{n+1}_i = u^n_i + \tau a_i|\nabla u|^n_i \sum_{j\in\mathcal{N}} \frac{2}{\left(\frac{|\nabla u|}{b}\right)^n_i + \left(\frac{|\nabla u|}{b}\right)^n_j}\cdot\frac{u^{n+1}_j - u^{n+1}_i}{h^2}, \tag{3.4}$$

donde $\mathcal{N}$ es el conjunto de puntos vecinos a i. La misma ecuación puede escribirse en notación matricial:

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \tau \sum_{l\in\{x,y,z\}} A_l(\mathbf{u}^n)\mathbf{u}^{n+1} \tag{3.5}$$

donde A_l es la matriz que describe la interacción en la dirección l. Cada elemento de la matriz $A_l(\mathbf{u}^n) = (\hat{a}_{ijl}(\mathbf{u}^n))$ está definido como:

$$\hat{a}_{ijl}(\mathbf{u}^n) := \begin{cases} a_i|\nabla u|_i^n \frac{2}{\left(\frac{|\nabla u|}{b}\right)_i^n + \left(\frac{|\nabla u|}{b}\right)_j^n} & \text{si } j \in \mathcal{N}_l(i), \\ -a_i|\nabla u|_i^n \sum_{m\in\mathcal{N}_l(i)} \frac{2}{\left(\frac{|\nabla u|}{b}\right)_i^n + \left(\frac{|\nabla u|}{b}\right)_m^n} & \text{si } j = i, \\ 0 & \text{en otro caso,} \end{cases} \tag{3.6}$$

donde $\mathcal{N}_l(i)$ es el conjunto de puntos vecinos a i en la dirección $l \in \{x,y,z\}$. La solución $\mathbf{u}^{n+1}$ requiere resolver el sistema de ecuaciones:

$$\left(I - \tau \sum_{l\in\{x,y,z\}} A_l(\mathbf{u}^n)\right)\mathbf{u}^{n+1} = \mathbf{u}^n, \tag{3.7}$$

donde I es la matriz unitaria. Para hallar la solución de este sistema de ecuaciones, el esquema AOS propone, en lugar de resolver el esquema semi-implícito:

$$\mathbf{u}^{n+1} = \left(I - \tau \sum_{l\in\{x,y,z\}} A_l(\mathbf{u}^n)\right)^{-1}\mathbf{u}^n, \tag{3.8}$$

utilizando la siguiente variante:

$$\mathbf{u}^{n+1} = \frac{1}{3}\sum_{l\in\{x,y,z\}} (I - 3\tau A_l(\mathbf{u}^n))^{-1}\,\mathbf{u}^n, \tag{3.9}$$

La principal ventaja de este esquema es que el operador $I - 3\tau A_l(\mathbf{u}^n)$ de la ecuación 3.9 es una matriz tridiagonal con diagonal dominante, lo que da lugar un sistema de ecuaciones que puede ser resuelto en serie aplicando el algoritmo de Thomas [122] u otras técnicas paralelas como reducción cíclica o doblamiento recursivo [29]. Para calcular la solución de la ecuación 3.9 hay que seguir los siguientes pasos:

1. Calcular la evolución en la dirección x resolviendo el sistema tridiagonal $(I - 3\tau A_x(\mathbf{u}^n))\mathbf{u}_x^{n+1} = \mathbf{u}^n$.

2. Calcular la evolución en la dirección y resolviendo el sistema tridiagonal $(I - 3\tau A_y(\mathbf{u}^n))\mathbf{u}_y^{n+1} = \mathbf{u}^n$.

3. Calcular la evolución en la dirección z resolviendo el sistema tridiagonal $(I - 3\tau A_z(\mathbf{u}^n))\mathbf{u}_z^{n+1} = \mathbf{u}^n$.

4. Promediar los tres resultados anteriores: $\mathbf{u}^{n+1} = \frac{1}{3}(\mathbf{u}_x^{n+1} + \mathbf{u}_y^{n+1} + \mathbf{u}_z^{n+1})$.

Es posible integrar el término "fuerza globo" (sección 1.3.3) dentro de la ecuación 3.9 de la siguiente forma:

$$\mathbf{u}^{n+1} = \frac{1}{3} \sum_{l \in \{x,y,z\}} (I - 3\tau A_l(\mathbf{u}^n))^{-1} (\mathbf{u}^n + \tau k g |\nabla \mathbf{u}|^n). \tag{3.10}$$

Por último, aproximaremos el cálculo del gradiente usando un esquema *upwind* [132]:

$$|\nabla u|_i^n \approx \begin{cases} \Big(\begin{array}{l} \max(D^{-x}u_i^n, 0)^2 + \min(D^{+x}u_i^n, 0)^2 + \\ \max(D^{-y}u_i^n, 0)^2 + \min(D^{+y}u_i^n, 0)^2 + \\ \max(D^{-z}u_i^n, 0)^2 + \min(D^{+z}u_i^n, 0)^2 \end{array} \Big)^{\frac{1}{2}} & \text{si } k < 0, \\ \Big(\frac{1}{2} \Big[\begin{array}{lll} (D^{-x}u_i^n)^2 & + & (D^{+x}u_i^n)^2 \\ (D^{-y}u_i^n)^2 & + & (D^{+y}u_i^n)^2 \\ (D^{-z}u_i^n)^2 & + & (D^{+z}u_i^n)^2 \end{array} \Big] \Big)^{\frac{1}{2}} & \text{si } k = 0, \\ \Big(\begin{array}{l} \min(D^{-x}u_i^n, 0)^2 + \max(D^{+x}u_i^n, 0)^2 + \\ \min(D^{-y}u_i^n, 0)^2 + \max(D^{+y}u_i^n, 0)^2 + \\ \min(D^{-z}u_i^n, 0)^2 + \max(D^{+z}u_i^n, 0)^2 \end{array} \Big)^{\frac{1}{2}} & \text{si } k > 0. \end{cases} \tag{3.11}$$

$D^{\pm x}$, $D^{\pm y}$ y $D^{\pm z}$ son aproximaciones de las derivadas espaciales en cada dimensión:

$$
\begin{aligned}
D^{+x}u^n_{i,j,k} &= \begin{cases} u^n_{i+1,j,k} - u^n_{i,j,k} & \text{si } 0 \le i < \bar{a}_x, \\ u^n_{i,j,k} - u^n_{i-1,j,k} & \text{si } i = \bar{a}_x, \end{cases} \\
D^{-x}u^n_{i,j,k} &= \begin{cases} u^n_{i,j,k} - u^n_{i-1,j,k} & \text{si } 0 < i \le \bar{a}_x, \\ u^n_{i+1,j,k} - u^n_{i,j,k} & \text{si } i = 0, \end{cases} \\
\\
D^{+y}u^n_{i,j,k} &= \begin{cases} u^n_{i,j+1,k} - u^n_{i,j,k} & \text{si } 0 \le i < \bar{a}_y, \\ u^n_{i,j,k} - u^n_{i,j-1,k} & \text{si } i = \bar{a}_y, \end{cases} \\
D^{-y}u^n_{i,j,k} &= \begin{cases} u^n_{i,j,k} - u^n_{i,j-1,k} & \text{si } 0 < i \le \bar{a}_y, \\ u^n_{i,j+1,k} - u^n_{i,j,k} & \text{si } i = 0, \end{cases} \\
\\
D^{+z}u^n_{i,j,k} &= \begin{cases} u^n_{i,j,k+1} - u^n_{i,j,k} & \text{si } 0 \le i < \bar{a}_z, \\ u^n_{i,j,k} - u^n_{i,j,k-1} & \text{si } i = \bar{a}_z, \end{cases} \\
D^{-z}u^n_{i,j,k} &= \begin{cases} u^n_{i,j,k} - u^n_{i,j,k-1} & \text{si } 0 < i \le \bar{a}_z, \\ u^n_{i,j,k+1} - u^n_{i,j,k} & \text{si } i = 0, \end{cases}
\end{aligned}
\tag{3.12}
$$

donde $\bar{a}_x$, $\bar{a}_y$ y $\bar{a}_z \in \mathbb{Z}^+$ son el tamaño en cada dimensión del volumen considerado.

3.3. Implementación en GPU

En esta sección presentamos la implementación en GPU de la segmentación AOS. Para optimizar el cálculo de la evolución del frente, utilizamos una estrategia de banda estrecha (véase sección 1.3.2) basada en dividir el volumen en regiones. En cada iteración del algoritmo se genera una lista de regiones activas, es decir, regiones cercanas o atravesadas por el frente. La computación se limita exclusivamente a estas regiones, mientras que en el resto de regiones los valores del conjunto de nivel permanecen constantes.

A grandes rasgos, el algoritmo se ejecuta en dos fases. En la primera fase, que hemos denominado *inicialización*, se establecen los valores iniciales de las estructuras de datos utilizadas durante la ejecución del algoritmo. La segunda fase, *evolución del conjunto de nivel*, ejecuta cada una de las etapas del algoritmo que hacen avanzar o retroceder el frente y realizan el proceso de segmentación.

3.3.1. Inicialización

La *inicialización* es la primera fase de ejecución de nuestra implementación del algoritmo de segmentación AOS. En esta fase se establecen los valores iniciales de las estructuras de datos que luego serán utilizadas en la segunda fase del algoritmo.

La inicialización contempla varias etapas. En primer lugar se calculan los valores iniciales del conjunto de nivel a partir de la posición y el tamaño configurados por el usuario para lo que se conoce como *semilla inicial*. A continuación se calculan los valores de la función de parada a partir de los datos almacenados en el volumen sobre el que se va a realizar la segmentación. Por último, se genera una lista de regiones activas, esto es, una lista con los identificadores de las regiones de tamaño fijo en las que se ha dividido el volumen y que son atravesadas por el frente inicial. Todas estas operaciones se realizan en CPU.

Al final de la inicialización, los datos almacenados en el conjunto de nivel inicial, la función de parada y la lista de regiones activas se copian en la memoria de la GPU.

Inicialización del conjunto de nivel

Antes de iniciar la segmentación, el usuario ha de seleccionar una posición y un tamaño para la semilla inicial. En esa implementación la semilla inicial tiene forma esférica, por lo que el usuario configura dos parámetros: la posición del vóxel dentro del volumen que ocupará su centro y la longitud de su radio (también en vóxeles). A partir de estos parámetros se calcula, para cada vóxel de volumen, su distancia euclidiana a la superficie de la esfera. Para este cálculo, se debe tener en cuenta si el vóxel pertenece o no al volumen de la esfera, y en función de esto el signo de la distancia será uno u otro. Qué signo usar para los vóxeles de fuera y de dentro de la esfera determina la tendencia natural del frente a moverse una vez se inicie el proceso de evolución (véase más adelante).

Obsérvese que, dada su configuración, la superficie de la esfera que el usuario define como semilla inicial atravesará en la mayor parte de los casos el espacio entre dos o más vóxeles[1]. Esto significa que incluso para los vóxeles más cercanos a la esfera, el valor de distancia será diferente de cero, y solo en algunos vóxeles será exactamente igual a cero. Como resultado, en la mayor parte de los casos la posición del frente está determinada no solo por aquellos vóxeles donde el conjunto de nivel toma el valor 0, sino también entre dos o más vóxeles adyacentes y de diferente signo.

[1] Nótese que estamos considerando cada vóxel no como un cubo, sino como un punto dentro del espacio 3D.

Cálculo de los valores de la función de parada

La función de parada determina la velocidad a la que van a cambiar los valores del conjunto de nivel en cada nueva iteración. En este modelo, la función de parada toma valores entre 0 y 1 para todos los vóxeles del volumen. En aquellos puntos donde la función de parada toma valores cercanos a 0 los valores del conjunto de nivel tenderán a quedarse estacionarios.

La selección de una función de parada determina en gran medida cómo va a evolucionar el frente durante el proceso de segmentación. La implementación que se presenta en este trabajo es versátil y soporta cualquier función de parada que verifique las dos siguientes condiciones:

– Tiene un valor entre 0 y 1 para cada vóxel del volumen.

– Es constante en el tiempo.

Normalmente suele escogerse una función de parada cuyos valores dependen del los valores de intensidad de cada vóxel del volumen o del gradiente de dichos valores. En esta implementación nos hemos decantado por una función de parada basada en el gradiente. Utilizamos como función de parada el estimador de bordes empleado en la ecuación de difusión de Perona y Malik [137], que es común en otros trabajos que también utilizan el esquema AOS [92, 183] (véase sección 1.3.3).

Generación de la lista de regiones activas iniciales

La última etapa antes de iniciar la segmentación es determinar qué regiones del volumen están inicialmente activas, ya que será en estas regiones donde se calculará la primera iteración de la evolución del conjunto de nivel. La lista de regiones activas cambiará a medida que el frente evolucione y se añadan o eliminen nuevas regiones del volumen, pero inicialmente esta lista contendrá todas aquellas regiones que son atravesadas por el frente y aquellas que estén más cerca de él, dentro del rango de distancia determinado por el ancho de la banda estrecha.

Este proceso comienza con la identificación de la posición del frente. El volumen se divide en pequeñas regiones de 2^3 vóxeles. Consideramos aquí que el frente atraviesa aquellas regiones que contienen valores del conjunto de nivel de distinto signo. Así, las coordenadas centrales de dichas regiones proporcionan una aproximación de las posiciones que ocupa el frente que es suficiente para construir la lista inicial de regiones activas.

A continuación, el volumen se divide de nuevo en regiones, pero ahora del mismo tamaño que el utilizado durante la evolución del conjunto de nivel, por lo general un múltiplo del

1: **para** $n = 0 \rightarrow máxIters - 1$:
2: **para** $l \in \{x,y,z\}$: $\mathscr{L}_l^n \leftarrow$ <calculaRegionesActivas($\mathbf{u}^n, l$)>
3: $|\nabla \mathbf{u}|_0^n, |\nabla \mathbf{u}|_k^n \leftarrow$ <calculaGradiente($\mathbf{u}^n, \mathscr{L}_x^n, k$)>
4: **para** $l \in \{x,y,z\}$: $A_l(\mathbf{u}^n) \leftarrow$ <calculaMatriz($|\nabla \mathbf{u}|_0^n, |\nabla \mathbf{u}|_k^n, \mathscr{L}_x^n, \mathscr{L}_y^n, \mathscr{L}_z^n, \tau, l$)>
5: **para** $l \in \{x,y,z\}$: $\mathbf{u}_l^{n+1} \leftarrow$ <resuelveSistema($I - 3\tau A_l(\mathbf{u}^n)$)$\mathbf{u}_l^{n+1} = \mathbf{u}^n$)>
6: $\mathbf{u}^{n+1} \leftarrow$ <promedia($\mathbf{u}_x^{n+1}, \mathbf{u}_y^{n+1}, \mathbf{u}_z^{n+1}$)>

Figura 3.1: Pseudocódigo de la implementación del algoritmo de segmentación en GPU.

tamaño 2^3. A cada región se le asigna un identificador único a partir de su posición dentro del volumen, de forma que es posible extraer la posición de una región a partir de su identificador. Dadas las posiciones aproximadas del frente calculadas al principio de este proceso, resulta trivial determinar cuáles de estas regiones son atravesadas por el frente.

Si el tamaño de la región es inferior al ancho de la banda estrecha, es necesario explorar también las regiones vecinas a aquellas identificadas inicialmente como activas. En esta implementación, cualquier región que se encuentre total o parcialmente dentro de la banda estrecha, es decir, aquellas que contengan al menos un vóxel cuya distancia a la posición del frente más cercana sea menor que el rango de distancia permitido por la banda estrecha, se considera también región activa.

3.3.2. Evolución del conjunto de nivel

Una vez que se han establecido la posición inicial del frente, los valores de la función de parada y la lista inicial de regiones activas, y se han inicializado las estructuras de datos utilizadas por el algoritmo, es posible ejecutar el proceso iterativo que modifica los valores del conjunto de nivel y hace avanzar o retroceder el frente.

La figura 3.1 muestra cómo se ha implementado la evolución del conjunto de nivel, con la siguiente notación: n indica la iteración actual; *máxIters* es un parámetro escogido por el usuario que determina el número de iteraciones que se ejecutarán del proceso de segmentación; $\mathbf{u}^n$ contiene los valores del conjunto de nivel en la iteración n; $\mathscr{L}_l^n$ contiene la lista de identificadores de las regiones activas en la dirección l para la iteración n; k y τ son dos parámetros cuyos valores establece el usuario y que controlan la velocidad de evolución del conjunto de nivel; $|\nabla \mathbf{u}|_0^n$ y $|\nabla \mathbf{u}|_k^n$ son, respectivamente, los valores del gradiente del conjunto de nivel $\mathbf{u}^n$ considerando $k = 0$ y k igual al valor designado por el usuario; por último, I y $A_l(\mathbf{u}^n)$ son, respectivamente, la matriz identidad y la matriz del sistema de ecuaciones generado a partir del conjunto de nivel en la iteración n (véase sección 3.2).

El listado contiene el pseudocódigo ejecutado por la CPU, donde las llamadas a código de GPU (implementado usando uno o varios *kernels* en CUDA) se encuentran representadas con los símbolos "menor que" y "mayor que". El proceso contempla varias etapas que se repiten iterativamente. En cada iteración, se comienza calculando las regiones activas, como se puede ver en la línea 3.1.2 de la figura. Se obtienen tres listas iguales, pero ordenadas en las tres posibles direcciones del espacio x, y y z, esto es, $\mathscr{L}_x^n$ contendrá la lista de regiones activas considerando x la dimensión más rápida, y $\mathscr{L}_y^n$ y $\mathscr{L}_z^n$ contendrán las mismas regiones considerando, respectivamente, y y z las dimensiones más rápidas.

Una vez calculadas las regiones activas para la iteración actual, se procede a calcular el gradiente de $\mathbf{u}^n$ considerando dos posibles valores para k (línea 3.1.3 del listado). A partir de estos gradientes, y teniendo en cuenta las listas de regiones activas en las tres direcciones del espacio y el valor de τ, se construyen tres sistemas de ecuaciones diferentes (línea 3.1.4); cada sistema tiene en cuenta las relaciones entre un vóxel y sus vecinos en cada una de las direcciones del espacio.

Los sistemas se resuelven por separado (línea 3.1.5), generando tres soluciones parciales que se almacenan en $\mathbf{u}_x^{n+1}$, $\mathbf{u}_y^{n+1}$ y $\mathbf{u}_z^{n+1}$. Cada iteración termina promediando los valores de las soluciones parciales para calcular el nuevo valor del conjunto de nivel $\mathbf{u}^{n+1}$ (línea 3.1.6).

Todas las operaciones se ejecutan en el espacio de memoria global de la GPU, excepto la etapa de resolución de los sistemas de ecuaciones, que, mediante la técnica "divide y fusiona", resuelve parcialmente las ecuaciones en el espacio de memoria compartida.

Cálculo de las regiones activas

La primera etapa de cada iteración del proceso de segmentación es el cálculo de las regiones activas. La generación de esta lista se realiza en GPU aplicando la técnica de compactación de datos (en inglés *stream compaction*) [79]. Este proceso se realiza en dos pasos. En el primero, denominado *exploración*, se identifican las regiones activas. Se parte de la lista de regiones activas de la iteración anterior, un *kernel* lanza tantos hilos como regiones haya en esa lista, y para cada región, recorre los valores de la función del conjunto de nivel en cada uno de sus vóxeles. En este caso, la región se considera activa si contiene valores de diferente signo, lo que significa que la región es atravesada por el frente. Como ya se ha explicado, la función del conjunto de nivel es una función de distancia al frente, y los vóxeles que pertenecen al interior del volumen de segmentación tienen un signo diferente a los vóxeles fuera de dicho volumen.

Un vector temporal de enteros, del mismo tamaño que el número de regiones e inicializado a cero, se utiliza para marcar qué regiones están activas. Cuando un hilo determina que su región asignada está activa, la marca en el vector escribiendo un uno, y hace los mismo para todas las regiones activas que estén dentro del rango de distancias determinado por el ancho de la banda estrecha. A continuación se realiza una operación de suma de prefijos exclusiva sobre el vector temporal utilizando una función de la librería *thrust* [24]. Con esta suma de prefijos, se obtienen las posiciones que deben ocupar los identificadores de las regiones activas dentro de la lista de regiones activas.

En el segundo paso, *dispersión*, se genera la nueva lista de regiones activas. A partir de los índices calculados en la suma de prefijos, un nuevo *kernel* recorre el vector de enteros, restaurando sus valores a cero (de forma que el vector pueda volver a ser utilizado en posteriores iteraciones del algoritmo) y almacenando, para las regiones activas, sus identificadores en las posiciones indicadas.

Los identificadores de las regiones activas están calculados a partir de su posición dentro del volumen, y cada región tiene tres posibles identificadores utilizando en cada caso x, y o z como la dimensión más rápida. Por tanto, cada identificador se calcula como:

$$id_x(x,y,z) = x + y \cdot \bar{a}_x + z \cdot \bar{a}_x \cdot \bar{a}_y, \tag{3.13}$$

$$id_y(x,y,z) = y + x \cdot \bar{a}_y + z \cdot \bar{a}_x \cdot \bar{a}_y, \tag{3.14}$$

$$id_z(x,y,z) = z + x \cdot \bar{a}_z + y \cdot \bar{a}_x \cdot \bar{a}_z, \tag{3.15}$$

donde $\bar{a}_x$, $\bar{a}_y$ y $\bar{a}_z$ denotan el tamaño del volumen en las dimensiones x, y y z, respectivamente.

Para el cálculo de regiones activas es suficiente con uno de los identificadores, pero durante el resto del algoritmo se utilizarán también los otros dos, por lo que es necesario generar tres listas de identificadores de regiones activas diferentes. El proceso de compactación genera una lista donde los identificadores tienen x como la dimensión más rápida. A partir de esta lista se generan dos nuevas listas con identificadores donde y y z son, respectivamente, las dimensiones más rápidas. Estas listas deben estar ordenadas de forma que dos elementos adyacentes en la lista sean vecinos en la dimensión más rápida. De nuevo, este proceso de ordenación se realiza mediante una llamada a la librería *thrust*.

La figura 3.2 muestra de forma esquemática un ejemplo de generación de listas de regiones activas para una imagen bidimensional. En este caso, la imagen se divide en 4×4 regiones del mismo tamaño, a cada una de las cuales se les asigna dos identificadores. La figura muestra el estado del vector de enteros tras explorar e identificar las regiones activas (que se muestran

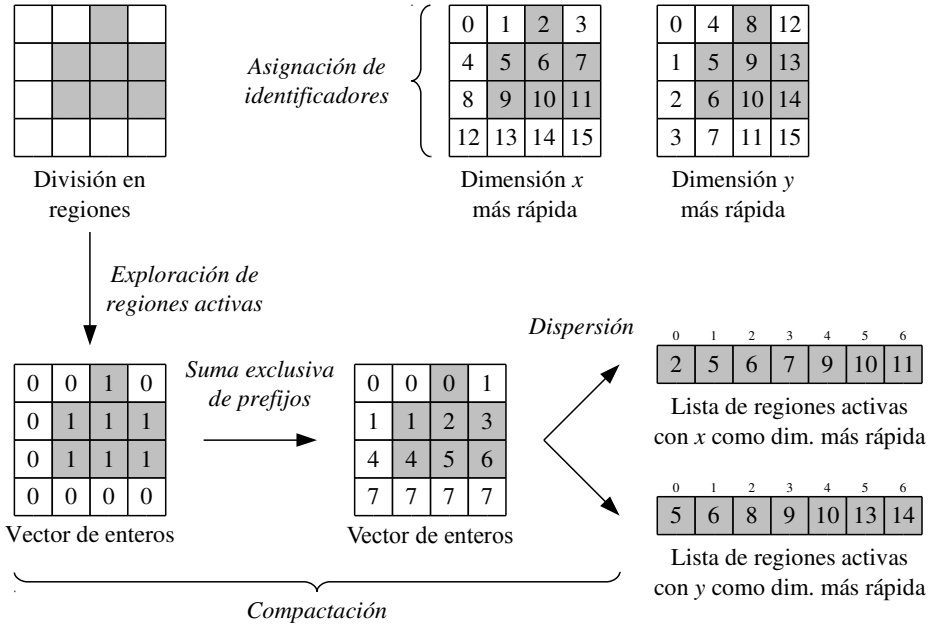


Figura 3.2: Identificación de regiones activas en la GPU.

con un fondo gris), y tras realizar la suma de prefijos. A partir de este cálculo, se obtienen las dos listas de regiones activas. Aunque los identificadores de ambas listas son diferentes, referencian las mismas regiones.

Cálculo del gradiente

Tras determinar las regiones activas en el volumen, se calcula el gradiente de los valores del conjunto de nivel en dichas regiones. Como se muestra en el pseudocódigo de la figura 3.1, es necesario calcular dos gradientes diferentes, atendiendo al valor del parámetro k configurado por el usuario y con un valor de k igual a cero.

El cálculo del gradiente se realiza en un *kernel* que implementa las operaciones indicadas en la ecuación (3.11). El *kernel* invoca tantos bloques de hilos como regiones activas haya en el volumen, de forma que cada hilo calcula el gradiente de un único vóxel.

Construcción de los sistemas de ecuaciones

Como ya se ha descrito en la sección 3.2, el proceso de segmentación requiere resolver tres sistemas lineales de ecuaciones a partir de los cuales se calculan las soluciones parciales $\mathbf{u}_l^{n+1}$ en las tres direcciones del espacio $l \in \{x, y, z\}$. Durante esta etapa, se construyen dichos sistemas de ecuaciones, con la forma:

$$\mathbf{u}_l^{n+1} = B_l^{-1}\mathbf{d}. \tag{3.16}$$

Los valores de la matriz del sistema $B_l(\mathbf{u}^n) = (\hat{b}_{ijl}(\mathbf{u}^n))$ y el vector $\mathbf{d}(\mathbf{u}^n) = (\hat{d}_i(\mathbf{u}^n))$ se generan a partir de la ecuación (3.10), que establece las relaciones entre cada vóxel y sus vecinos y, además, introduce el término de la "fuerza globo". Los coeficientes de la matriz B_l se calculan como:

$$\hat{b}_{ijl}(\mathbf{u}^n) := \begin{cases} -3 \cdot \tau \cdot \hat{a}_{ijl}(\mathbf{u}^n) & \text{si } j \in \mathcal{N}_l(i), \\ 1 - 3 \cdot \tau \cdot \hat{a}_{ijl}(\mathbf{u}^n) & \text{si } j = i, \\ 0 & \text{en otro caso,} \end{cases} \tag{3.17}$$

donde los valores $\hat{a}_{ijl}(\mathbf{u}^n)$ aparecen definidos en la ecuación (3.6) (usando los valores de gradiente $|\nabla \mathbf{u}^n|$ calculados en la etapa previa con $k = 0$) y $\mathcal{N}_l(i)$ es el conjunto de puntos vecinos a i en la dirección l. Cada elemento del vector $\mathbf{d}$ se construye como:

$$\hat{d}_i(\mathbf{u}^n) := u_i^n + \tau \cdot |\nabla u|_i^n \cdot k \cdot g_i, \tag{3.18}$$

con el gradiente $|\nabla \mathbf{u}^n|$ ya calculado previamente, en este caso con k igual al valor determinado por el usuario.

El proceso de construcción de cada sistema se ha implementado en varios pasos. En primer lugar, un *kernel* invoca tantos bloques de hilos como regiones activas, y cada hilo es asignado a un vóxel de una región activa del volumen. Por tanto, los sistemas solo tendrán tantas ecuaciones como vóxeles haya en las regiones activas. Los hilos calculan los valores de los coeficientes de B_l y $\mathbf{d}$. Al final de la ejecución del *kernel*, se han generado cuatro vectores temporales, tres con los coeficientes de las diagonales de B_l y uno con los del vector $\mathbf{d}$.

Los coeficientes de los vectores temporales aparecen ordenados de acuerdo a cómo han sido asignados a los hilos de ejecución, por lo que el sistema resultante no tiene forma tridiagonal. Para establecer la forma tridiagonal del sistema, es necesario reordenarlos según la dirección que se esté procesando, es decir, atendiendo a los identificadores de cada elemento en el volumen (usando x como dimensión más rápida si los coeficientes pertenecen a B_x, o

bien y o z como dimensiones más rápidas si los coeficientes pertenecen a B_y o B_z, respectivamente). Este último paso se ha implementado usando las funciones de la librería *thrust*.

La figura 3.3 ilustra el proceso de construcción de los sistemas de ecuaciones para un caso bidimensional. En la parte superior, una imagen de tamaño 6×6 píxeles ha sido dividida en regiones de 2×2 píxeles. Las regiones activas aparecen resaltadas en gris. Se generan dos sistemas de ecuaciones, en la dirección x y en la dirección y. Dado que el número de regiones activas es 4, cada sistema tiene $4 \times 4 = 16$ ecuaciones. La figura muestra las matrices de cada sistema, donde cada ecuación está identificada por la coordenada (x,y), con notación abreviada xy, de su elemento correspondiente en la imagen. Los coeficientes resaltados en gris se corresponden por los calculados por el *kernel*, el resto de coeficientes son cero. Como se puede observar, los sistemas de ecuaciones generados son dispersos. Tras reordenar las ecuaciones (esto es, tanto los coeficientes en B_l como los elementos de $\mathbf{d}$), se obtienen sistemas de ecuaciones tridiagonales equivalentes a los originales.

Resolución de los sistemas de ecuaciones

En esta etapa se resuelven los sistemas de ecuaciones tridiagonales generados en la etapa anterior. La resolución se lleva a cabo utilizando la implementación en GPU del algoritmo de reducción cíclica con la técnica de "divide y fusiona", descrita en el capítulo 2.

Los coeficientes de los sistemas de ecuaciones se copian en unos vectores temporales. Dado que la implementación actual de reducción cíclica solo soporta vectores cuyo tamaño es potencia de 2, los vectores se inicializan previamente con ceros en el caso de las diagonales superior e inferior de la matriz y del término dependiente y unos en el caso de de la diagonal central y luego se sobrescriben con el contenido de los sistemas de ecuaciones. El efecto es que los sistemas originales se amplían con ecuaciones vacías hasta la cantidad potencia de 2 más cercana. El resultado de estas ecuaciones no afecta al resto de las ecuaciones y es ignorado.

Cálculo de la media de las soluciones parciales

La última etapa de cada iteración del proceso de segmentación consiste en promediar los resultados parciales obtenidos de los sistemas de ecuaciones tridiagonales construidos en las etapas previas.

La implementación es directa. Un *kernel* invoca tantos hilos como vóxeles activos haya en el volumen. Cada hilo recoge los valores de la solución parcial $\mathbf{u}_x^{n+1}$ para el vóxel que tiene

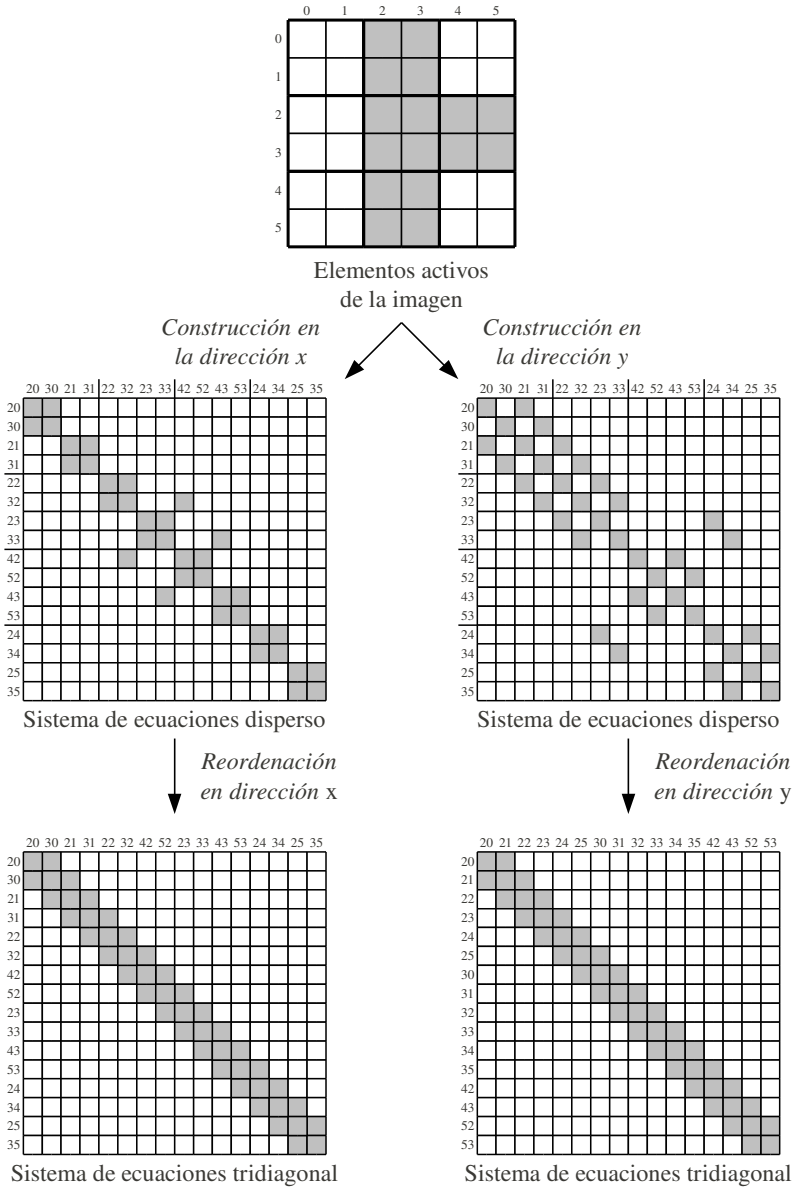


Figura 3.3: Construcción de los sistemas de ecuaciones en la GPU.

asignado en la dimensión x, lo divide entre 3, y lo almacena en la posición correspondiente del vector que almacena los contenidos de $\mathbf{u}^{n+1}$. El proceso se repite para las dimensiones y y z, pero esta vez sumando la solución parcial normalizada sobre el contenido ya existente en $\mathbf{u}^{n+1}$, de forma que al final este vector contendrá el promedio de las soluciones parciales.

Observaciones finales

Los vectores que almacenan los valores del conjunto de nivel $\mathbf{u}$, las dos versiones del gradiente $\nabla\mathbf{u}$ con $k = 0$ y k igual al valor establecido por el usuario, la función de parada g, y las funciones a y b se almacenan en la GPU como *surfaces*. Las *surfaces*, que pueden tener hasta tres dimensiones, son una estructura de datos especial de CUDA que utiliza la caché de texturas de la GPU. Esto nos permite explotar mejor la localidad espacial de los datos en las tres dimensiones y, además, libera a la caché de la memoria global de cargar estos datos.

La implementación final intercala la construcción de los sistemas de ecuaciones con su resolución en cada dirección del espacio. De esta forma, se pueden reutilizar los vectores de datos para almacenar y resolver los sistemas, lo que reduce el consumo de memoria.

3.4. Resultados

En esta sección presentamos las medidas de rendimiento y de calidad obtenidas para la implementación del algoritmo de segmentación presentado en este capítulo.

3.4.1. Metodología

Ejecutamos nuestra implementación en GPU del algoritmo de segmentación en una tarjeta gráfica NVIDIA GeForce GTX 680, cuyas características pueden consultarse en la sección 1.7. La versión del algoritmo implementada en CPU fue compilada con *gcc* y ejecutada en un Intel Core 2 con cuatro *cores* a 2,6 GHz y 6 GB de RAM, utilizando un sistema operativo Linux. Hemos evaluado el rendimiento de nuestra solución midiendo los tiempos de ejecución en GPU y tomando medidas de aceleración respecto a una implementación OpenMP del mismo algoritmo segmentación que hemos preparado.

Datos	Dimensiones	Bytes por vóxel	Tamaño
`cube`	n/a	1	n/a
`knee`	$281 \times 389 \times 104$	2	21,7 MB

Tabla 3.1: Conjuntos de datos usados en los experimentos.

3.4.2. Conjuntos de datos

Para tomar los resultados presentados en este capítulo hemos utilizado los siguientes conjuntos de datos:

– Un volumen artificial que contiene un cubo en su interior, para el cual generamos varias versiones escaladas a diferente tamaño. Los vóxeles en el interior del cubo tienen el valor de gris más alto, y el resto de vóxeles que pertenecen al fondo tienen un valor de cero. Este volumen representa un ejemplo sencillo de segmentar, y lo hemos utilizado para evaluar el rendimiento del algoritmo, que hemos comparado con la versión en CPU. Hemos denominado a este conjunto de datos `cube`.

– Un volumen con datos reales extraídos de un volumen CT de una rodilla. En este caso, hemos tratado de segmentar el tejido óseo del fémur de las demás estructuras presentes en el volumen. Los vóxeles de este tejido presentan valores de gris muy bajos, similares a los del fondo del volumen. El tejido cartilaginoso que rodea algunas partes del hueso, así como algunas fibras musculares, presentan también valores de gris similares. Hemos denominado a este conjunto de datos `knee`.

La tabla 3.1 muestra el tamaño y las dimensiones de ambos conjuntos de datos. La figura 3.4 muestra una lámina bidimensional extraída del volumen `cube`, y un ejemplo de segmentación realizado con la implementación en GPU, donde el volumen de segmentación se visualiza en diferentes instantes de la ejecución del algoritmo usando una isosuperficie. Como se puede observar, la segmentación se inicia dibujando una semilla esférica inicial que rodea completamente el cubo. A cada iteración, la esfera disminuye su tamaño hasta que adquiere la forma del cubo.

La figura 3.5 muestra una lámina extraída del volumen `knee`, junto con un ejemplo de segmentación. En este caso, la segmentación se inicia colocando una semilla inicial dentro del espacio ocupado por el hueso, para luego hacerla crecer en cada iteración. A medida que crece, la semilla adquiere la forma de la articulación.

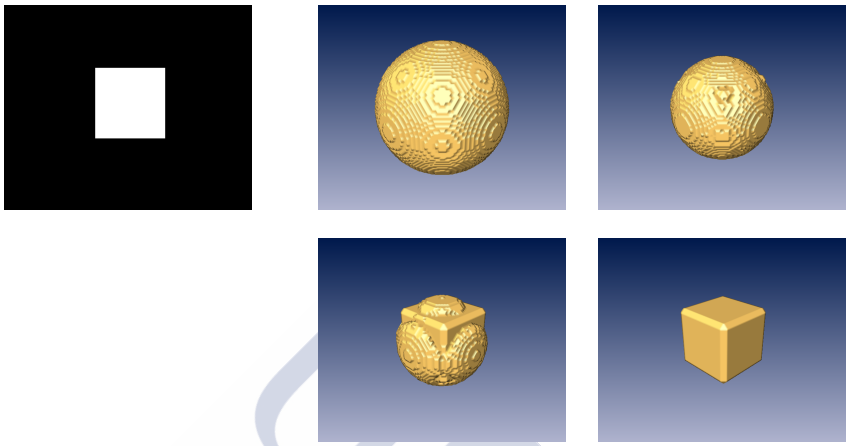


Figura 3.4: Visualización de los resultados de segmentación del volumen `cube`. A la izquierda se muestran los datos originales. A la derecha se muestran cuatro pasos (ordenados de izquierda a derecha y de arriba a abajo) del proceso de segmentación.

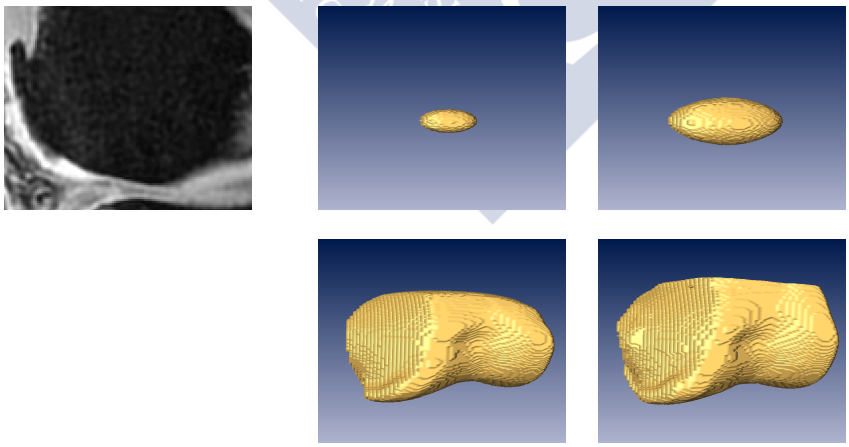


Figura 3.5: Visualización de los resultados de segmentación del volumen `knee`. A la izquierda se muestran los datos originales. A la derecha se muestran cuatro pasos (ordenados de izquierda a derecha y de arriba a abajo) del proceso de segmentación.

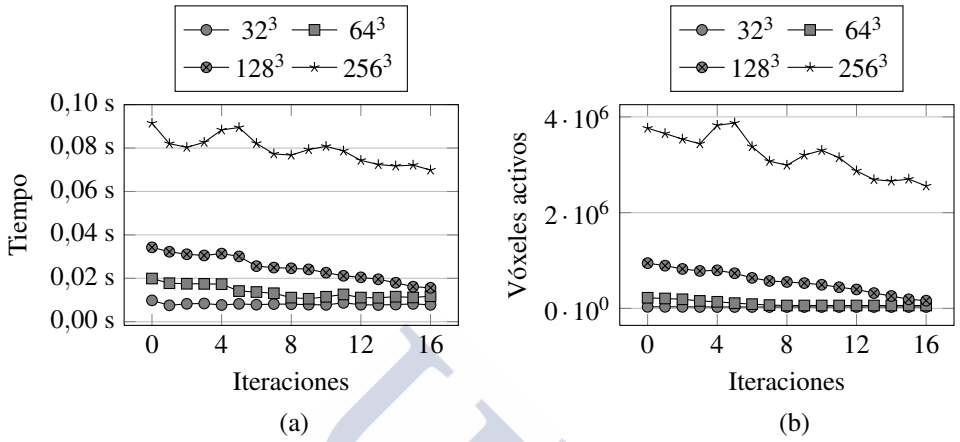


Figura 3.6: Segmentación en una tarjeta GTX 680 de versiones de `cube` con tamaños 32^3, 64^3, 128^3 y 256^3. Se muestran (a) el tiempo de ejecución de cada iteración y (b) el número de vóxeles activos en cada iteración.

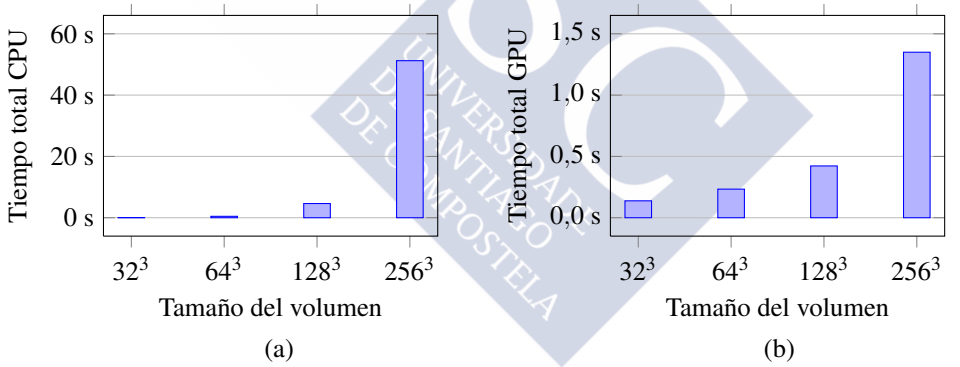


Figura 3.7: Tiempos totales para segmentar versiones de `cube` con tamaños 32^3, 64^3, 128^3 y 256^3. Se muestran (a) el tiempo de ejecución en CPU con OpenMP y (b) el tiempo de ejecución en una tarjeta GTX 680.

3.4.3. Medidas de rendimiento

En primer lugar, hemos medido el rendimiento de nuestra solución en GPU tomando tiempos en cada iteración del algoritmo. La figura 3.6 muestra el tiempo necesario para ejecutar cada iteración para los diferentes tamaños del volumen artificial `cube`. Se muestra también el número de vóxeles activos en cada iteración. Se observa que existe una fuerte correlación

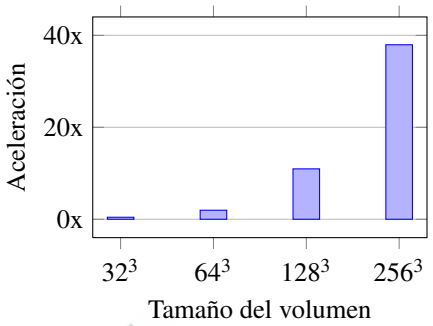


Figura 3.8: Valores de aceleración de la segmentación en una tarjeta GTX 680 respecto a la segmentación en OpenMP usando versiones de `cube` con tamaños 32^3, 64^3, 128^3 y 256^3.

Tamaño	CPU	GPU	Aceleración
32^3	0,060310 s	0,137818 s	0,44x
64^3	0,460466 s	0,233180 s	1,97x
128^3	4,628223 s	0,422348 s	10,96x
256^3	51,230437 s	1,349871 s	37,95x

Tabla 3.2: Tiempos de ejecución y valores de aceleración comparando la implementación en OpenMP y en una tarjeta GTX 680 usando versiones de `cube` de diferentes tamaños.

entre el tiempo de ejecución y el número de elementos activos en cada iteración. Nuestra solución es sensible al tamaño de los datos a procesar.

La figura 3.7 muestra el tiempo total de segmentación para diferentes tamaños de `cube` usando la implementación en GPU y la implementación en OpenMP. La figura 3.8 muestra los valores de aceleración de la implementación en GPU respecto a la CPU, calculados a partir de los tiempos anteriores. Se observa que, para tamaños pequeños del volumen, los valores de aceleración no son significativos. En cambio, a medida que crece el tamaño del volumen la implementación en GPU es más eficiente, llegando a alcanzar valores de aceleración cercanos a 40x. Estos datos aparecen también reflejados en la tabla 3.2.

Con el objetivo de generar un perfil de rendimiento de la implementación en GPU del algoritmo de segmentación, tomamos mediciones del tiempo de ejecución de las distintas etapas del algoritmo ejecutadas en la GPU (véase sección 3.3.2). La tabla 3.3 contiene estas mediciones para una versión del conjunto de datos `cube` con un tamaño de 128^3. Los tiempos mostrados se han calculado sumando el tiempo consumido para cada etapa en todas las itera-

Etapa	Tiempo	Porcentaje
Cálculo de las regiones activas	0,056 s	14,0 %
Cálculo del gradiente	0,011 s	2,8 %
Construcción de los sistemas de ecuaciones	0,252 s	62,5 %
Resolución de los sistemas de ecuaciones	0,074 s	18,4 %
Media de las soluciones parciales	0,009 s	2,3 %

Tabla 3.3: Tiempo dedicado a cada etapa de la segmentación en una tarjeta GTX 680 para una versión de `cube` de tamaño 128^3.

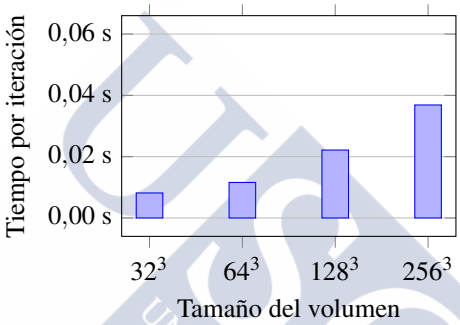


Figura 3.9: Tiempo de ejecución medio por iteración para la segmentación de volúmenes de diferentes tamaños extraídos del conjunto de datos `knee` en una tarjeta GTX 680.

Tamaño	Iteraciones	Tiempo medio
32^3	11	0,008 s
64^3	43	0,012 s
128^3	100	0,022 s
256^3	173	0,037 s

Tabla 3.4: Resultados de segmentar en una tarjeta GTX 680 volúmenes de diferentes tamaños extraídos del conjunto de datos `knee`.

ciones de la ejecución. Además del tiempo, se muestra también el porcentaje que cada etapa consume del tiempo total, y se observa que más del 60 % del tiempo se dedica a la construcción de los sistemas de ecuaciones. Un análisis más detallado de esta etapa mostró que más de la mitad de este tiempo se consumía durante las ordenaciones. Sin embargo, este proceso ya se encuentra optimizado al realizarse mediante llamadas al API de *thrust*.

La figura 3.9 muestra el rendimiento de la implementación en GPU segmentando regiones de interés de diferentes tamaños extraídas del volumen `knee`. La tabla 3.4 contiene los datos

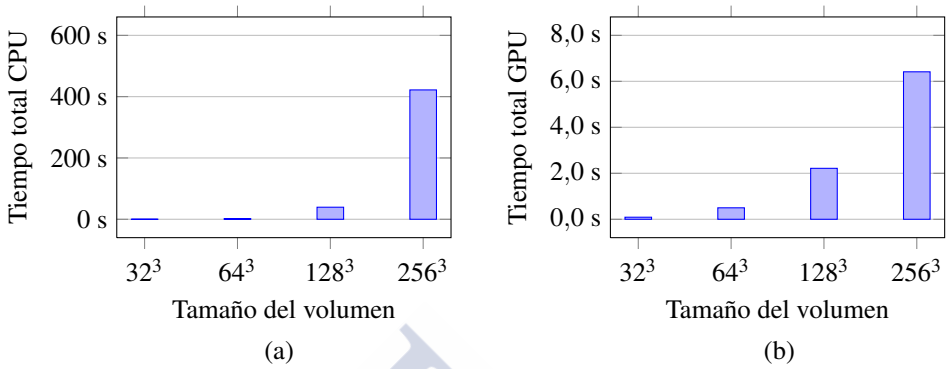


Figura 3.10: Tiempos totales para segmentar versiones de `knee` con tamaños 32^3, 64^3, 128^3 y 256^3. Se muestran (a) el tiempo de ejecución en CPU con OpenMP y (b) el tiempo de ejecución en una tarjeta GTX 680.

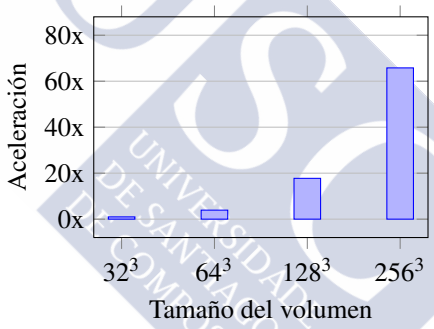


Figura 3.11: Valores de aceleración de la segmentación en una tarjeta GTX 680 respecto a la segmentación en OpenMP usando versiones de `knee` con tamaños 32^3, 64^3, 128^3 y 256^3.

usados para generar las gráficas. Para cada tamaño, se muestran el número de iteraciones ejecutadas para completar la segmentación, la duración media de cada iteración y el tiempo total para completar la segmentación. La segmentación se realizó tras ajustar los parámetros de forma que el resultado obtenido fuese la segmentación del tejido óseo del fémur dentro de la región de interés. En este caso la segmentación es más compleja que en el caso del volumen `cube`, y el tiempo necesario para completar la segmentación es mayor.

La figura 3.10 muestra el tiempo total para completar la segmetnación del conjunto de datos `knee` en la implementación en OpenMP y en la implementación en GPU, y la figura 3.11 recoge los valores de aceleración asociados. Al igual que en el caso del conjunto de datos `cube`, los mejores tiempos de aceleración se consiguen cuanto mayor sea el tamaño de

Tamaño	CPU	GPU	Aceleración
32^3	0,089109 s	0,089705 s	0,99x
64^3	1,953362 s	0,498364 s	3,92x
128^3	39,324378 s	2,215935 s	17,75x
256^3	421,951764 s	6,413728 s	65,79x

Tabla 3.5: Tiempos de ejecución y valores de aceleración comparando la implementación en OpenMP y en una tarjeta GTX 680 usando versiones de `knee` de diferentes tamaños.

la región de interés a segmentar. En el mejor caso se alcanza un valor de aceleración superior a 65x. La tabla 3.5 refleja estos datos con más detalle.

3.4.4. Medidas de calidad

Para evaluar la calidad de las segmentaciones y validar nuestra implementación hemos utilizado el coeficiente Dice (véase sección 1.6). En esta ocasión, disponíamos para el volumen `knee` de una segmentación de referencia realizada por expertos. La segmentación de referencia separaba entre el tejido óseo y el cartilaginoso. Nuestro interés consistía principalmente en segmentar el tejido óseo del fémur. En nuestras pruebas, tras ajustar convenientemente los parámetros de segmentación, conseguimos un índice Dice con un valor cercano al 85 % tras segmentar el tejido óseo de la región más cercana a la rodilla, sin realizar ninguna clase de preprocesado sobre el volumen.

3.5. Conclusiones

En este capítulo hemos presentado una solución en GPU para realizar la segmentación de volúmenes tridimensionales basada en conjuntos de nivel y que utiliza la técnica AOS. Esta técnica aproxima el cálculo de la evolución del conjunto de nivel mediante la resolución de sistemas tridiagonales de ecuaciones lineales.

Nuestra implementación utiliza el algoritmo de resolución de sistemas tridiagonales basado en reducción cíclica con "divide y fusiona", presentado en el capítulo 2. La evolución del conjunto de nivel se calcula íntegramente en la tarjeta gráfica, utilizando tanto *kernels* propios como llamadas a funciones del API de *thrust*.

Hemos comparado el rendimiento de nuestra solución con una implementación del algoritmo en OpenMP. Hemos comprobado que la solución en GPU es aproximadamente 40 veces

más rápida que la implementación en OpenMP, especialmente en volúmenes grandes. Hemos comparado los resultados de segmentación obtenidos con otras segmentaciones de referencia, obteniendo valores del índice Dice cercanos al 85 %.

Capítulo 4

Algoritmo de segmentación rápida de dos ciclos

4.1. Introducción

En el capítulo 3 presentamos una implementación en GPU del algoritmo de segmentación basado en el esquema de operador aditivo. Esta implementación utilizaba el patrón de paralelismo "divide y fusiona" (desarrollado en el capítulo 2) para resolver los sistemas tridiagonales de ecuaciones lineales que se generaban durante los pasos intermedios del algoritmo. En este capítulo presentamos dos propuestas de implementación en GPU del algoritmo de segmentación rápida de dos ciclos (en inglés FTC, *fast two cycle*) [162], que, al igual que el algoritmo de segmentación AOS, también está basado en métodos de conjuntos de nivel (la sección 1.3 describe los principales fundamentos teóricos de este tipo de métodos).

El método FTC se caracteriza fundamentalmente por utilizar solo operaciones enteras para actualizar el conjunto de nivel. Se trata, pues, de un algoritmo más atractivo para su adaptación en GPU de cara al objetivo de proporcionar una solución interactiva para la segmentación de imágenes médicas sin pérdidas significativas de calidad. Completar una segmentación de la forma más rápida posible no solo tiene interés para la mejora de la velocidad de diagnóstico: la selección de valores de los parámetros utilizados en un proceso de segmentación suele realizarse siguiendo un proceso de ensayo y error; poder visualizar de forma inmediata el resultado de una segmentación permite al usuario corregir los valores de los parámetros de forma intuitiva para obtener el resultado deseado.

La implementación secuencial del método de segmentación FTC presentada en [162] se apoya en el uso de listas enlazadas que no se pueden usar de forma eficiente en GPU. En su lugar, nuestra implementación modifica la estructura del algoritmo original mediante el patrón de paralelismo de descomposición geométrica (visto en el capítulo 2), lo que nos permite explotar las características del *hardware* gráfico. Así, nuestras principales contribuciones son:

- Dividimos el dominio computacional en regiones 3D del mismo tamaño que pueden ser almacenadas en memoria compartida y procesadas eficientemente por diferentes bloques de hilos. Esta división requiere tener en cuenta las zonas de transición entre regiones vecinas.

- Modificamos la estructura iterativa del algoritmo original para adaptar el número de iteraciones en el algoritmo a las características de la GPU, de forma que los multiprocesadores puedan calcular varias iteraciones en cada región. Esto nos permite explotar el espacio de memoria compartida, de menor latencia que la memoria global.

- Al procesar cada región de forma independiente, se generan inconsistencias entre las diferentes regiones. Proponemos una solución para detectar y resolver las inconsistencias en memoria global.

- Mantenemos una lista de regiones activas que calculan la evolución del conjunto de nivel solo en las regiones donde una porción del frente esté presente. Nuestra segunda implementación restringe esta lista solo a aquellas regiones donde el frente aún no se ha estabilizado.

Hasta donde hemos podido averiguar, la única implementación en GPU del método de segmentación FTC está descrita en [177]. Sin embargo, está enfocada a datos bidimensionales y solo analiza un único paso de uno de los ciclos del algoritmo. Nuestra solución está diseñada para datos tridimensionales e implementa todas las operaciones del método FTC, lo que ha requerido realizar cambios algorítmicos relevantes.

El resto de este capítulo está organizado como sigue. La sección 4.2 describe el método de segmentación FTC y la sección 4.3 examina nuestras dos propuestas para su implementación en GPU. La sección 4.4 analiza los resultados de rendimiento y calidad obtenidos por nuestras dos propuestas y los compara con otros trabajos recientes de segmentación en GPU. Por último, la sección 4.5 concluye este capítulo.

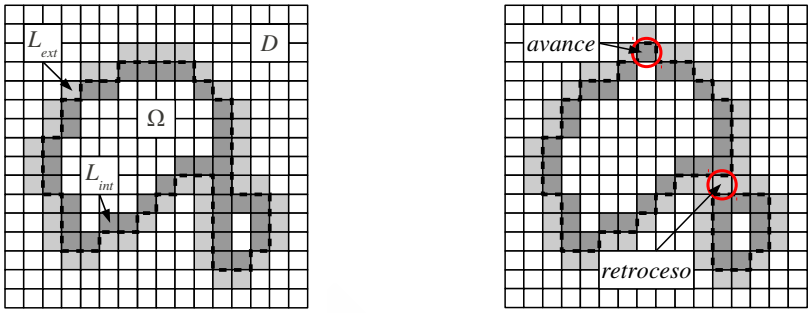


Figura 4.1: Representación del frente y del conjunto de nivel en el método de segmentación FTC sobre un espacio bidimensional discreto.

4.2. Método de segmentación rápida de dos ciclos

El método de segmentación rápida de dos ciclos (FTC) descrito en [162] es una aproximación de un método del conjunto de nivel disperso. Este método analiza el movimiento de un frente representado implícitamente en un espacio discreto y uniforme. Denotamos este espacio discreto y uniforme como D. Para la región $\Omega \subseteq D$ encerrada por el frente, se definen dos listas de puntos vecinos:

$$\begin{aligned} L_{int} &= \{\mathbf{x} \mid \mathbf{x} \in \Omega \text{ y } \exists \mathbf{y} \in \mathcal{N}(\mathbf{x}) \text{ tal que } \mathbf{y} \notin \Omega\}, \\ L_{ext} &= \{\mathbf{x} \mid \mathbf{x} \notin \Omega \text{ y } \exists \mathbf{y} \in \mathcal{N}(\mathbf{x}) \text{ tal que } \mathbf{y} \in \Omega\}, \end{aligned} \tag{4.1}$$

donde $\mathcal{N}(\mathbf{x}) = \{\mathbf{y} \in D \mid \sum_{k=1}^{n} |y_k - x_k| = 1\}$ es el conjunto discreto de puntos vecinos a $\mathbf{x}$. El algoritmo se puede generalizar para cualquier tipo de selección de vecinos.

La figura 4.1 (a) muestra un ejemplo de un espacio bidimensional discreto donde el frente (representado mediante una línea discontinua) está localizado entre el conjunto de puntos definidos por L_{int} y L_{ext} (en gris oscuro y gris claro, respectivamente).

El método FTC define la función del conjunto de nivel u como una función entera que puede tomar valores del conjunto $\{-3,-1,+1,+3\}$. Considerando los *puntos internos* como aquellos puntos dentro de Ω pero no en L_{int} y, por otra parte, los *puntos externos* como aque-

llos puntos fuera de Ω pero no en L_{ext}, la función u es una aproximación local de la función de distancia:

$$u(\mathbf{x}) = \begin{cases} +3 & \text{si } \mathbf{x} \text{ es un punto externo,} \\ +1 & \text{si } \mathbf{x} \in L_{ext}, \\ -1 & \text{si } \mathbf{x} \in L_{int}, \\ -3 & \text{si } \mathbf{x} \text{ es un punto interno.} \end{cases} \tag{4.2}$$

El frente avanza hacia el exterior o retrocede hacia el interior añadiendo y eliminando puntos $\mathbf{x}$ de las listas L_{int} y L_{ext}, actualizando el valor de $u(\mathbf{x})$ y actualizando también los puntos vecinos correspondientes. Para ello, el algoritmo define dos procedimientos: *avance*, que hace avanzar el frente colocando un punto de L_{ext} en L_{int}, y *retroceso*, que hace retroceder el frente colocando un punto de L_{int} en L_{ext}. Ambos procedimientos actualizan los puntos vecinos para mantener los valores de la función del conjunto de nivel consistentes con la posición del frente.

En la figura 4.1 (b) se muestra el resultado de aplicar ambos procedimientos en el espacio discreto de ejemplo. En la parte superior, el frente se ha movido hacia afuera tras aplicar un *avance* en el punto señalado. En la parte inferior, el frente se ha movido hacia adentro como resultado de un *retroceso* que, además, divide a Ω en dos regiones.

La figura 4.2 muestra el flujo de trabajo del método de segmentación FTC en pseudocódigo. El algoritmo tiene una estructura iterativa, donde cada iteración se divide en dos partes denominadas primer y segundo ciclo. Ambos ciclos ejecutan la misma secuencia de pasos, pero con una función de velocidad F diferente. La función de velocidad del primer ciclo es F_1, y se define a partir del problema a tratar; en el contexto de la segmentación, por ejemplo, los valores de esta función pueden estar basados en los valores de intensidad de los vóxeles del volumen o en los valores de gradiente. El segundo ciclo utiliza F_2, que sirve de término de regularización de suavizado (más adelante veremos que esta función realiza un cálculo aproximado de la curvatura media del frente). Así, al intercalar los dos ciclos, el método FTC aproxima la evolución del frente como si la función de velocidad fuese una sola $F = F_1 + F_2$. En ambos ciclos el avance o retroceso del frente viene determinado por el signo de esta función: el frente avanza si el signo es positivo y retrocede si el signo es negativo.

Analizando el pseudocódigo, al principio del algoritmo (línea 1) se inicializan las estructuras para almacenar los valores de u, F_1, F_2 y las listas de puntos L_{int} y L_{ext}. Comienza entonces el primer ciclo (líneas 2–16), que es repetido durante N_1 iteraciones o hasta que la condición

1: Inicializar u, F_1, F_2 y las listas L_{int} y L_{ext}

2: **para** $i = 1 \rightarrow N_1$: ▷ Primer ciclo
3: **para todo** $\mathbf{x} \in L_{ext}$: ▷ Evolución hacia afuera
4: **si** $F_1(\mathbf{x}) > 0$:
5: avance($\mathbf{x}$)
6: **para todo** $\mathbf{x} \in L_{int}$: ▷ Elimina puntos de L_{int}
7: **si** $\forall \mathbf{y} \in \mathcal{N}(\mathbf{x}), u(\mathbf{y}) < 0$:
8: elimina $\mathbf{x}$ de L_{int} y $u(\mathbf{x}) \leftarrow +3$
9: **para todo** $\mathbf{x} \in L_{int}$: ▷ Evolución hacia adentro
10: **si** $F_1(\mathbf{x}) < 0$:
11: retroceso($\mathbf{x}$)
12: **para todo** $\mathbf{x} \in L_{ext}$: ▷ Elimina puntos de L_{ext}
13: **si** $\forall \mathbf{y} \in \mathcal{N}(\mathbf{x}), u(\mathbf{y}) > 0$:
14: elimina $\mathbf{x}$ de L_{ext} y $u(\mathbf{x}) \leftarrow -3$
15: **si** condición de parada satisfecha:
16: fin del bucle

17: **para** $i = 1 \rightarrow N_2$: ▷ Segundo ciclo
18: **para todo** $\mathbf{x} \in L_{ext}$: ▷ Evolución hacia afuera
19: **si** $F_2(\mathbf{x}) > 0$:
20: avance($\mathbf{x}$)
21: **para todo** $\mathbf{x} \in L_{int}$: ▷ Elimina puntos en L_{int}
22: **si** $\forall \mathbf{y} \in \mathcal{N}(\mathbf{x}), u(\mathbf{y}) < 0$:
23: elimina $\mathbf{x}$ de L_{int} y $u(\mathbf{x}) \leftarrow +3$
24: **para todo** $\mathbf{x} \in L_{int}$: ▷ Evolución hacia adentro
25: **si** $F_2(\mathbf{x}) < 0$:
26: retroceso($\mathbf{x}$)
27: **para todo** $\mathbf{x} \in L_{ext}$: ▷ Elimina puntos en L_{ext}
28: **si** $\forall \mathbf{y} \in \mathcal{N}(\mathbf{x}), u(\mathbf{y}) > 0$:
29: elimina $\mathbf{x}$ de L_{ext} y $u(\mathbf{x}) \leftarrow -3$

30: **si** condición de parada **no** satisfecha:
31: repetir de nuevo desde el primer ciclo

Figura 4.2: Pseudocódigo del método de segmentación FTC.

de parada sea satisfecha. N_1 es uno de los parámetros configurables del algoritmo. Cada iteración del primer ciclo realiza los siguientes pasos: en primer lugar, para cada punto de L_{ext} donde F_1 tenga un valor positivo se ejecuta un *avance* (líneas 3–5), lo que mueve el frente un punto hacia afuera; como resultado, algunos puntos que estaban en L_{int} se convierten en puntos internos, por lo que se eliminan de la lista (líneas 6–8). Entonces, para cada punto en L_{int} donde F_1 tenga un valor negativo, se realiza un *retroceso* moviendo el frente hacia adentro (líneas 9-11). De forma similar que antes, los puntos en L_{ext} que son ahora puntos externos se eliminan de la lista (líneas 12–14).

El último paso del primer ciclo consiste en comprobar la condición de parada (línea 15). Esta condición se considera satisfecha si, y solo si:

$$\begin{aligned} &\forall \mathbf{x} \in L_{ext}, \;\; F_1(\mathbf{x}) \leq 0 \\ &\forall \mathbf{x} \in L_{int}, \;\; F_1(\mathbf{x}) \geq 0, \end{aligned} \tag{4.3}$$

o si se alcanza un número máximo previamente especificado de iteraciones.

La segmentación continúa con el segundo ciclo (líneas 17–29), un proceso iterativo que se repite N_2 veces. N_2 es otro parámetro configurable por el usuario, y normalmente suele escogerse un valor mucho menor que N_1. Los pasos del segundo ciclo son similares a los del primero, aunque en este caso la función de velocidad es F_2. Como ya avanzamos, esta función sustituye al término de regularización de suavizado presente en otros métodos del conjunto de nivel para controlar la evolución del frente en base a su curvatura. Como el cálculo de la curvatura es costoso, en el método FTC el término de regularización se aproxima realizando un suavizado gaussiano basándose en una técnica previamente utilizada en [118, 119]. Sea G la función gaussiana:

$$G(\mathbf{x}) = \frac{1}{2\pi\sigma^2} e^{-\frac{|\mathbf{x}|^2}{2\sigma^2}}, \tag{4.4}$$

donde el valor de la varianza σ es proporcionado por el usuario. La función de velocidad F_2 calculada para todos los puntos de L_{ext} se define como:

$$F_2(\mathbf{x}) = \begin{cases} +1 & \text{si } G*H(-u)(\mathbf{x}) > \frac{1}{2}, \\ 0 & \text{en otro caso.} \end{cases} \tag{4.5}$$

En cambio, para todos los puntos de L_{int} se define como:

$$F_2(\mathbf{x}) = \begin{cases} -1 & \text{si } G*H(-u)(\mathbf{x}) < \frac{1}{2}, \\ 0 & \text{en otro caso,} \end{cases} \tag{4.6}$$

donde $*$ denota el operador de convolución y H es una función Heaviside tal que:

$$H(u)(\mathbf{x}) = \begin{cases} 0 & \text{si } \mathbf{x} \notin \Omega, \\ 1 & \text{si } \mathbf{x} \in \Omega. \end{cases} \tag{4.7}$$

En el contexto de la segmentación, el objetivo del método FTC es localizar regiones dentro de imágenes y volúmenes. Solo las posiciones más cercanas al frente son de interés, por lo que el dominio activo es básicamente una banda de unos pocos vóxeles de grosor. El método utiliza dos funciones de velocidad, por lo que su ecuación de actualización del conjunto de nivel podría escribirse como:

$$u(\mathbf{x}, t+1) = \begin{cases} h(F_1, u(\mathbf{x}, t)) & \text{para el primer ciclo,} \\ h(F_2, u(\mathbf{x}, t)) & \text{para el segundo ciclo,} \end{cases} \tag{4.8}$$

donde h transforma los valores de $u(\mathbf{x}, t)$ en $u(\mathbf{x}, t+1)$ para hacer avanzar o retroceder el frente de acuerdo al valor de la función de velocidad en cada punto x.

Nótese que la ecuación de actualización no considera el gradiente u. El frente evoluciona sin necesidad de resolver ecuaciones diferenciales parciales, en contraste con los métodos del conjunto de nivel descritos en el capítulo 1 (véase sección 1.3). La evolución está estrictamente controlada por los valores de las distintas funciones de velocidad, por lo que los valores de u no son más que etiquetas que identifican las posiciones dentro del frente, y dentro y fuera de la región Ω (de forma similar a los campos de etiquetas de otros métodos de campo disperso [186]).

Aunque el método FTC está restringido a volúmenes de puntos discretos y uniformemente muestreados, conserva características de los métodos del conjunto de nivel que son relevantes para la segmentación de imágenes y volúmenes, como, por ejemplo, el manejo automático de cambios topológicos. El volumen de segmentación converge hacia la región esperada siempre que se verifiquen las condiciones establecidas en [162]. Además, el método no requiere reinicializar la función del conjunto de nivel.

4.3. Implementación en GPU

En esta sección proponemos dos implementaciones en GPU método de segmentación FTC. Como ya hemos comentado, la implementación original del FTC utiliza dos listas enlazadas que contienen las posiciones donde está localizado el frente. Cuando el frente evoluciona, estas listas son modificadas con el estado más actual del frente. Dado que las listas

enlazadas no pueden ser implementadas eficientemente en GPU, nuestras propuestas utilizan una aproximación diferente: particionar el dominio en regiones del mismo tamaño que puedan ser almacenadas en memoria compartida y procesadas por los bloques de hilos. La representación del dominio activo no es tan precisa como en el algoritmo original, pero a cambio la evolución del frente puede ser calculada en paralelo para cada una de las regiones activas (regiones que contienen el frente) y no es necesario actualizar la lista de elementos activos en cada iteración.

Los cambios propuestos para implementar el método FTC en GPU requieren una alteración de la estructura original del algoritmo. En nuestra primera propuesta modificamos el número de iteraciones del primer y del segundo ciclo para poder ejecutar el algoritmo en GPU a la vez que se evitan computaciones que no implican una evolución significativa del frente. La segunda propuesta difiere de la primera en que realiza una selección más restrictiva de las regiones activas, lo que reduce el tamaño del dominio activo y, a la vez, incrementa el rendimiento sin pérdidas significativas de calidad. Ambas propuestas utilizan un paralelismo de grano fino, ya que cada hilo opera sobre un único vóxel. A lo largo de esta sección explicaremos los detalles de implementación.

Nuestras dos propuestas se ejecutan de dos fases. La primera de ellas, la fase de *inicialización*, se establecen los valores iniciales de las estructuras de datos utilizadas durante la ejecución del algoritmo. En la segunda fase, *evolución del conjunto de nivel*, se ejecutan los pasos necesarios para modificar las posiciones del frente y realizar la segmentación. Dado que la segunda propuesta está desarrollada a partir de la primera, explicaremos primero la primera propuesta, y luego enumeraremos las características incorporadas a la segunda propuesta y que la diferencian de la primera.

4.3.1. Inicialización

La *inicialización* es la primera fase de ambas propuestas de implementación en GPU del método de segmentación FTC. Durante esta fase se establecen los valores iniciales de las estructuras de datos utilizadas durante la ejecución del algoritmo.

El proceso de inicialización se lleva a cabo en dos etapas. En primer lugar se establece la posición y el tamaño de la *semilla inicial*, lo que permite determinar los valores iniciales de u. A continuación, se calcula la lista de coordenadas de las regiones activas, que luego será actualizada a medida que evolucione el frente.

Inicialización del conjunto de nivel

La inicialización del conjunto de nivel comienza una vez que el usuario ha configurado una posición y un tamaño para la semilla inicial. En esta implementación la semilla es una esfera, por lo que el usuario debe seleccionar el vóxel que será el centro de la esfera y la longitud de su radio en vóxeles. La semilla puede generarse de forma que esté completamente dentro de la región que se desea segmentar, por lo que durante la fase de evolución tenderá a expandirse hasta abarcar dicha región. Alternativamente, la semilla podría encerrar completamente la región a segmentar, por lo que durante la evolución tendería a encogerse hasta adoptar la forma de la región.

Con los datos de la posición y radio iniciales de la semilla, el conjunto de nivel se inicializa con los valores permitidos, de forma que los vóxeles en el interior y más cercanos a la esfera se correspondan con los puntos internos, y los vóxeles en el exterior de la esfera se correspondan con los puntos externos (véase sección 4.2). Al final de este proceso se obtiene se obtiene una región Ω cerrada que conformará el volumen inicial de segmentación. Aunque en esta implementación no hacemos uso de las listas L_{int} y L_{ext}, los valores utilizados para representar los vóxeles en el frente, los puntos internos y los externos se definen de igual modo que en la ecuación (4.2).

Esta etapa de la inicialización se realiza en CPU. Una vez se han obtenido los valores iniciales de u, los datos se almacenan en la memoria de la GPU, para poder proceder con la siguiente etapa de la inicialización.

Generación de la lista de regiones activas iniciales

Una vez que los valores iniciales de u están almacenados en la memoria de la GPU se construye la lista inicial de coordenadas activas. En lugar de almacenar las coordenadas de cada región, asignamos a cada región un identificador único calculado a partir de dichas coordenadas, de forma que se pueden extraer las coordenadas de una región a partir de su identificador, y viceversa.

La generación de la lista de regiones activas se realiza en la memoria global de la GPU aplicando un proceso de compactación [79], de forma similar a la implementación en GPU del método de segmentación AOS desarrollado en el capítulo 3 (véase sección 3.3.2). La compactación tiene dos pasos: *exploración* y *dispersión*. Durante la *exploración* se identifican las regiones activas. Un *kernel* lanza tantos hilos como regiones en que ha sido dividido el volumen. Cada hilo se asigna a una región y comprueba los valores de u en los puntos corres-

pondientes a esa región. En este caso, una región es considerada activa si, y solo si, al menos uno de sus puntos pertenece al borde exterior del volumen de segmentación Ω, denotado como L_{ext} en el método FTC original:

$$\exists \mathbf{x} \in \text{Región}(u), \mathbf{x} \in L_{ext}. \tag{4.9}$$

Utilizamos un vector temporal de enteros, del mismo tamaño que el número de regiones pero inicializado a cero, para indicar qué regiones están activas y cuáles no. Cuando un hilo detecta que su región es activa, la marca almacenando un uno en la posición del vector correspondiente a su región asignada. Cuando se han marcado todas las regiones activas, se realiza una suma exclusiva de prefijos sobre el vector para obtener las posiciones que debe ocupar cada identificador de cada región activa dentro de la lista de regiones activas. Esta operación se realiza en la GPU mediante una función de la librería *thrust* [24].

En el paso de *dispersión* se genera la nueva lista de regiones activas. Un nuevo *kernel* procesa el vector de enteros, restaurando sus valores a cero (de forma que el vector pueda volver a ser utilizado en posteriores iteraciones del algoritmo) y almacenando, para cada región activa, su identificador en la posición calculada.

4.3.2. Evolución del conjunto de nivel

Una vez que se han inicializado los valores de u y se ha construido la lista inicial de regiones activas puede comenzar el proceso iterativo de segmentación. Aunque el núcleo computacional del algoritmo está implementado en código GPU, la CPU está a cargo de gestionar el flujo de trabajo y realizar las llamadas a los *kernels* implementados. Las principales tareas computacionales asignadas a la GPU consisten en actualizar el conjunto de nivel y evaluar la condición de parada.

La figura 4.3 muestra el pseudocódigo ejecutado por la CPU de nuestra implementación del método FTC, con la siguiente notación: *condParada* es una variable que almacena si la condición de parada ha sido o no satisfecha; n indica la iteración actual del bucle del primer ciclo, y N_0 es el número máximo de iteraciones de dicho ciclo; N_1 y N_2 son, respectivamente, el número de iteraciones que se ejecutarán del primer y segundo ciclo dentro de la GPU. Las llamadas a código de la GPU (implementado usando uno o varios *kernels* en CUDA) se encuentran representadas con los símbolos “menor que” y “mayor que”.

Como se puede observar, el código de CPU ya no itera sobre cada uno de los pasos individuales de los dos ciclos del algoritmo, ya que este proceso se realiza internamente en la GPU.

```
1:  condParada ← falso

2:  para n = 1 → N0:
3:      <ejecutar primer ciclo (N1 iters. en mem. compartida)>
4:      <identificar regiones activas (en mem. global)>

5:      condParada ← <comprobar condición de parada (en mem. global)>
6:      si condParada:
7:          fin del bucle

8:  <ejecutar segundo ciclo (N2 iters. en mem. compartida)>
9:  <identificar regiones activas (en mem. global)>

10: si no condParada:
11:     repetir de nuevo desde la línea 1
```

Figura 4.3: Pseudocódigo de la primera propuesta de implementación del método de segmentación FTC en GPU.

Ambos ciclos han sido implementados en dos *kernels* que ejecutan, respectivamente, las N_1 y N_2 iteraciones que se corresponden con cada ciclo (líneas 3 y 8 del pseudocódigo). Se ha incluido, además, un nuevo bucle (línea 2) que ejecuta varias veces el *kernel* del primer ciclo en cada iteración principal del algoritmo. Un nuevo parámetro, N_0, cuyo valor es especificado por el usuario, determina el número de veces que el *kernel* del primer ciclo es ejecutado. Después de cada llamada a dicho *kernel*, se actualiza la lista de regiones activas.

La introducción de un nuevo bucle respecto al método original FTC está motivada por la división en regiones del dominio. Cada región es procesada de forma independiente y en paralelo, por lo que el frente no puede moverse de una región a otra adyacente hasta que se hayan recalculado las regiones activas y se hayan resuelto las posibles inconsistencias entre regiones (de las que hablaremos más adelante). Dado que el tamaño de estas regiones no es demasiado grande (en nuestro caso, hemos usado regiones de tamaño 8^3), ejecutar dentro del *kernel* del primer ciclo un número alto de iteraciones (en nuestro caso, más de 8) no tiene ningún efecto (el frente no puede avanzar o retroceder más de 8 veces). La solución que hemos implementado utiliza valores de N_1 inferiores o iguales a 8, e introduce las llamadas de las líneas 3 y 4 del pseudocódigo de la figura 4.3 en un nuevo bucle, para que el dominio activo se actualice una vez que el frente no pueda seguir avanzando.

Durante el bucle del ciclo uno, tras recalcular la lista de regiones activas, se verifica la condición de parada. Si esta condición ha sido satisfecha, el bucle terminaría sin completar las N_0 iteraciones. En cualquier caso, ya sea porque se han completado las N_0 iteraciones, o

porque se verifica la condición de parada, a continuación se ejecuta en GPU el segundo ciclo, y se vuelve a calcular el dominio activo. Nótese que, aunque hemos introducido un nuevo parámetro N_0 y un nuevo bucle para el primer ciclo, no existe un parámetro similar para el segundo ciclo. Aunque este ciclo se ejecuta de forma similar al primero, con el dominio dividido en regiones que se procesan en paralelo y de forma independiente, N_2 suele tomar valores muy pequeños. Así, incluso para el restringido tamaño de las regiones, una única ejecución del *kernel* del segundo ciclo por cada iteración del algoritmo es suficiente para obtener el efecto deseado.

Los parámetros N_0, N_1 y N_2 permiten al usuario determinar la influencia de los dos ciclos que componen este algoritmo en la evolución del frente. Aunque a diferencia de la versión original nuestra implementación del método FTC el parámetro N_1 tiene un rango de valores muy restringido, el usuario puede utilizar en su lugar N_0 y ajustar la influencia de forma similar. Valores demasiado altos de N_0 pueden provocar "derrames" en la segmentación, es decir, que el frente se mueva hacia regiones no deseadas. Por otra parte, valores demasiado altos de N_2 disminuyen la velocidad a la que evoluciona el frente, y en algunos casos pueden hacer que permanezca estático.

Al final del algoritmo se comprueba si la condición de parada había sido satisfecha durante la ejecución del ciclo uno. Si no es así, el proceso se repite de desde el principio, ejecutando una nueva iteración global del algoritmo.

Ejecución del primer y del segundo ciclo

Dado que tanto el primer ciclo como el segundo ejecutan los mismos pasos y, por tanto, su implementación es similar, los analizamos conjuntamente en esta sección.

Ambos ciclos solo pueden ejecutarse una vez que se ha calculado la lista de regiones activas para la iteración actual. Cada ciclo ha sido implementado en su propio *kernel*. Estos *kernels* invocan tantos bloques de hilos como regiones activas, y cada bloque de hilos contiene tantos hilos como vóxeles en la región, de forma que cada hilo procesa un único vóxel. Cada bloque de hilos se asigna a una región activa, y dado que los bloques son tridimensionales, cada hilo se asigna de manera directa a cada uno de los vóxeles de la región a partir de sus coordenadas.

El primer paso, antes de modificar los valores de u, es copiar el contenido de cada región activa en la memoria compartida. Para cada bloque de hilos se reserva una matriz tridimensional en la memoria compartida que almacenará los elementos de la región asignada y también

```
1: para i = 0 → tamaño_c − 1 paso tamaño_b:

2:     i_c ← i + i_t
3:     i_g ← i_r + i_c

4:     si i_c e i_g están dentro de los límites:
5:         u_c(i_c) ← u_g(i_g)
```

Figura 4.4: Pseudocódigo de la carga de datos en memoria compartida.

todos aquellos pertenecientes a la zona de solapamiento. Esta zona de solapamiento contiene los vóxeles que rodean a la región asignada y que pertenecen a regiones adyacentes, y es necesaria ya que durante la evolución del frente los hilos tienen que acceder la información de su vóxel correspondiente y también la de sus vecinos. En el primer ciclo esta zona de solapamiento tiene un grosor de un vóxel, ya que para procesar cada vóxel, incluyendo los que se encuentran en el borde de la región, es necesario la información de sus vecinos adyacentes. En el segundo ciclo la región es más grande, ya que para realizar el suavizado solo con la información de los vóxeles inmediatamente adyacentes no es suficiente; cada vóxel necesita la información de los vóxeles a su alrededor de acuerdo al tamaño del filtro gaussiano. En ambos casos los vóxeles que se encuentran en esta zona de solapamiento no se modifican durante toda la ejecución del ciclo.

La figura 4.4 muestra el pseudocódigo que ejecuta cada hilo de la GPU para cargar los datos una región de u, inicialmente almacenada en la memoria global, a la memoria compartida. Por simplicidad, el pseudocódigo muestra el proceso para una sola dimensión, pero nuestra implementación soporta datos tridimensionales. Hemos utilizado la siguiente notación: $tamaño_c$ es una constante con el tamaño del espacio reservado en memoria compartida para almacenar los datos de la región y de la zona de solapamiento; $tamaño_b$ contiene el número de hilos por bloque; i_c es el índice en memoria compartida donde se almacenará el dato; i_t es la coordenada del hilo dentro del bloque; i_g es el índice de memoria global de donde se cargará el dato; i_r es la coordenada de la región asignada al bloque de hilos; u_c y u_g representan, respectivamente, el espacio de memoria compartida y memoria global que intervienen en este proceso, donde los datos originales de u están almacenados en u_g.

Como se puede ver en el pseudocódigo, el proceso de carga se ejecuta dentro de un bucle (línea 1). Las coordenadas/índices i_c e i_g se calculan a partir del índice i del bucle y de la coordenada del hilo i_t (líneas 2 y 3). Antes de leer o escribir en memoria, se comprueba que las coordenadas estén dentro de los límites para evitar accesos no válidos a memoria

(línea 4). El volumen puede tener un tamaño no múltiplo del tamaño de la región, por lo que algunas regiones pueden caer parcialmente fuera del volumen; en estos casos, se asigna un valor especial a los vóxeles correspondientes en memoria compartida.

La figura 4.5 muestra cómo se ejecuta el proceso de carga en memoria compartida para un ejemplo en 2D. En este ejemplo, un bloque de 4×4 hilos carga datos de una región de tamaño 4×4 con una zona de solapamiento de un vóxel de grosor. La figura muestra la región en memoria global (izquierda) y cómo se copian a la memoria compartida (derecha) en cuatro pasos. Los hilos se representan mediante cuadrados pequeños. En el primer paso, todos los hilos del bloque cargan los elementos de la esquina superior izquierda de la región, incluyendo también la zona de solapamiento. En el segundo paso, los hilos se desplazan a la derecha, lo que deja a algunos hilos ociosos ya que no tienen ningún dato que copiar. En el tercer paso, los hilos se vuelven a desplazar, esta vez abajo y a la izquierda; de nuevo, algunos hilos estarán ociosos por no estar asignados a ningún dato. Por último, en el cuarto paso los hilos se desplazan hacia la derecha y copian los últimos datos. Aunque este ejemplo muestra el proceso para datos bidimensionales, recordemos que nuestra implementación opera con datos tridimensionales. La principal ventaja de este método es que se puede extender para cualquier posible configuración de tamaño de región, bloque de hilos y zona de solapamiento.

Una vez que se han cargado los datos en memoria compartida, el siguiente paso consiste en corregir las posibles que inconsistencias que hayan aparecido en iteraciones previas del algoritmo. Las inconsistencias surgen como consecuencia de procesar el conjunto de nivel en regiones independientes: el frente crece en cada región de forma independiente, sin tener en cuenta el estado de las regiones adyacentes, lo que da lugar a discrepancias entre ellas. Las inconsistencias se identifican cuando no se verifica el siguiente par de condiciones:

– Todo vóxel perteneciente a L_{int} solo puede tener como vecinos adyacentes otros vóxeles de L_{int}, de L_{ext} o del interior de Ω.

– Todo vóxel perteneciente a L_{ext} solo puede tener como vecinos adyacentes otros vóxeles de L_{ext}, de L_{int} o del exterior de Ω.

Dado que las inconsistencias aparecen fundamentalmente en los vóxeles que se encuentran en el borde de la región, durante el proceso de identificación cada hilo asignado a un vóxel de este tipo comprobará si las condiciones se siguen cumpliendo dado el valor de su vóxel y el valor de los vóxeles adyacentes que se encuentren en la zona de solapamiento (por tanto, fuera de la región).

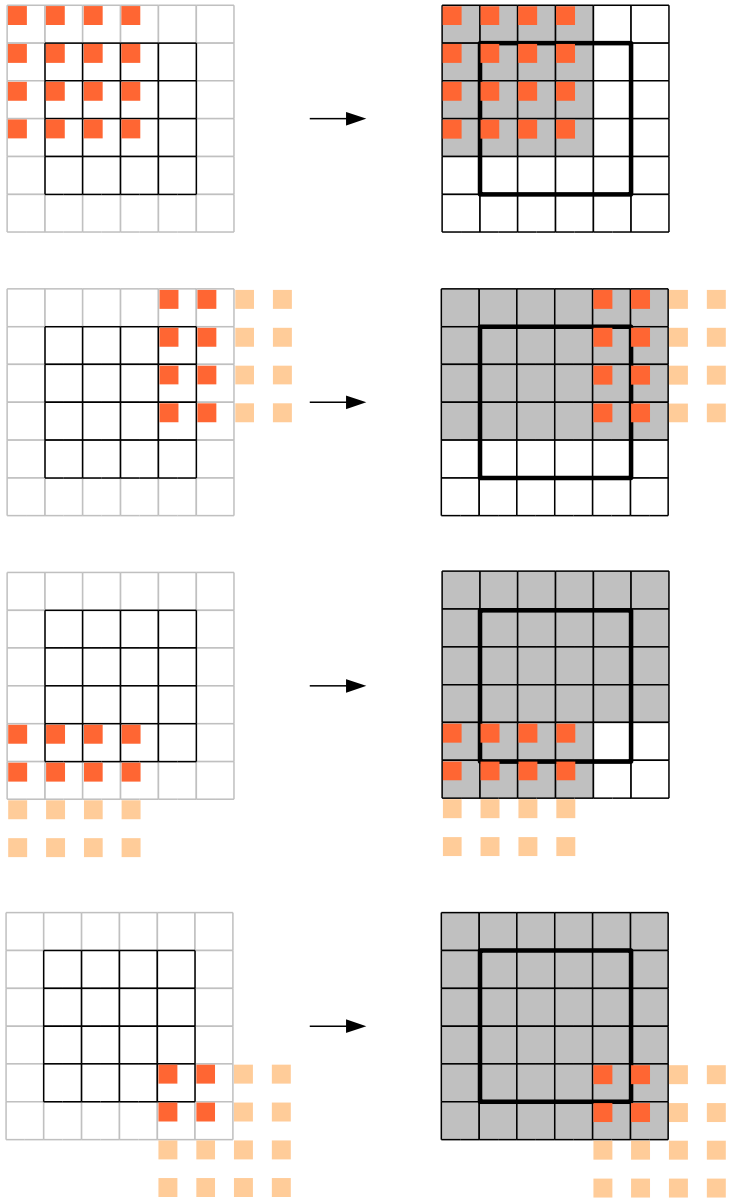


Figura 4.5: Copia de datos de una región bidimensional de memoria global a memoria compartida.

```
1: para todo i_l ∈ Límite(u_c) en paralelo:
2:     t_l ← u_c(i_l)
3:     t_s ← u_c(i_s)

4:     si t_l = +3 y t_s = −1:
5:         u_c(i_l) ← +1
6:     si t_l = −3 y t_s = +1:
7:         u_c(i_l) ← −1
```

Figura 4.6: Pseudocódigo de la corrección de inconsistencias en memoria compartida.

En la figura 4.6 se muestra el pseudocódigo para corregir las inconsistencias entre regiones. Se ha utilizado la siguiente notación: u_c es la matriz que almacena en memoria compartida la región asignada al bloque de hilos y la zona de solapamiento; i_l es la coordenada de un vóxel en el límite de la región almacenada en memoria compartida; i_s es la coordenada del vóxel adyacente a i_l en la zona de solapamiento; por último, t_l y t_s son dos variables temporales que almacenan, respectivamente, los valores de u_c en las coordenadas i_l e i_s.

El código ha sido diseñado para tratar el caso especial en que se añade una nueva región a la lista de regiones activas. En este caso, el frente (definido en aquellos puntos donde $u = \pm 1$), se encuentra entre esta región y la adyacente; como se trata de una nueva región que todavía no ha sido procesada, no contiene los valores correctos de u. El código procesa en paralelo los vóxeles que se encuentran en el límite de la región. En primer lugar, se inicializan las variables t_l y t_s con los valores de u_c en el límite de la región y en la zona de solapamiento adyacente, respectivamente (líneas 2 y 3). Entonces, se comprueba si i_l se corresponde con un punto externo al volumen encerrado por el conjunto de nivel (según los valores establecidos en la ecuación (4.2)) y si i_s es un vóxel de L_{int} (línea 4); en tal caso, hay una inconsistencia, y el valor de u_c en i_l se corrige para formar parte de L_{ext} y mantener la consistencia del frente (línea 7). Este sería el caso de un avance del frente que se traduce en incorporar una nueva región al dominio activo, pero un retroceso también puede incorporar nuevas regiones al dominio activo que anteriormente se encontraban en el interior del volumen encerrado por el frente. La segunda condición evalúa este posible caso y, en esencia, se trata de una comprobación similar (línea 6).

La figura 4.7 muestra un ejemplo de corrección de inconsistencias. En la parte superior se puede ver el estado de dos regiones bidimensionales adyacentes, además de los vóxeles vecinos. El frente, que al igual que en la versión de CPU está definido por vóxeles que pertenecen a L_{int} (en gris oscuro) y L_{ext} (en gris claro), atraviesa la región de la izquierda (la línea

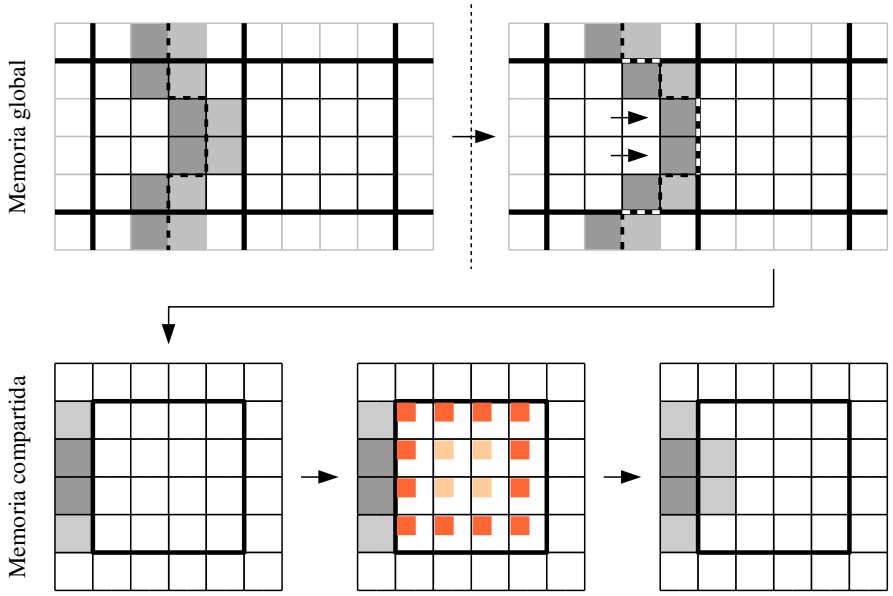


Figura 4.7: Ejemplo de corrección de inconsistencias.

punteada representa la posición exacta del frente). A la derecha puede verse cómo el frente avanza tras ejecutar los pasos necesarios para calcular su evolución. El frente se encuentra ahora entre dos regiones, pero solo la región de la izquierda pudo ser actualizada correctamente, por ser parte del dominio activo. En la siguiente iteración del algoritmo, la región de la derecha se convierte en región activa, y es procesada en la memoria compartida. La parte inferior de la figura muestra este proceso más en detalle. La región, junto con la zona de solapamiento, ha sido cargada en la memoria compartida. Los hilos asignados a los vóxeles en el límite de la región ejecutan el proceso de detección de inconsistencias, y los demás hilos permanecen ociosos. Las inconsistencias son detectadas y corregidas, de forma que el frente vuelve a verificar las condiciones de consistencia mencionadas.

Una vez que se han resuelto las inconsistencias en los límites de la región, se calcula la evolución del conjunto de nivel de forma iterativa. La figura 4.8 muestra el pseudocódigo de este proceso, con la siguiente notación: u_c representa el contenido parcial de u almacenado en memoria compartida, y u_g el contenido almacenado en memoria global; i_c son las coordenadas del vóxel procesado por el hilo en la memoria compartida; e i_g son las coordenadas

```
1: para todo i_c ∈ Región(u_c) en paralelo:
2:     para i = 1 → N_*:                    ▷ N_* puede ser N_1 o N_2
3:         f ← F_*(i_g)                     ▷ F_* puede ser F_1 o F_2

4:         avance(u_c, f)
5:         retroceso(u_c, f)

6:     u_g(i_g) ← u_c(i_c)
```

Figura 4.8: Pseudocógido del algoritmo de evolución del conjunto de nivel en GPU.

```
1: si u_c(i_c) = +1 y f > 0:
2:     u_c(i_c) ← −1

3: si u_c(i_c) = +3 y ∃j_c ∈ η(i_s), u_c(j_c) = −1:
4:     u_c(i_s) ← +1

5: si u_c(i_s) = −1 y ∄j_c ∈ η(i_s), u_c(j_c) = +1 y ∄j_c ∈ η(i_s), u_c(j_c) = +3:
6:     u_c(i_c) ← −3
```

Figura 4.9: Pseudocódigo de la operación de avance en GPU.

correspondientes en memoria global; N_1 y N_2 son el número de iteraciones del primer y del segundo ciclo; y F_1 y F_2 se corresponden con la función de velocidad en el primer y en el segundo ciclo, respectivamente.

Cada hilo inicia un proceso iterativo (línea 2) que comienza calculando el valor de la función de velocidad para su vóxel asignado (línea 3). A continuación se ejecuta el código necesario para actualizar el valor del conjunto de nivel. Estas operaciones se realizan en varios pasos, y aparecen denotadas como *avance* y *retroceso* (líneas 4 y 5). Por último, el contenido en memoria compartida de u_c se almacena en la memoria global u_g (línea 6). Todos estos pasos solo se ejecutan para los vóxeles que pertenecen al interior de la región, la zona de solapamiento permanece inalterada y no se copia en memoria global.

La figura 4.9 muestra el pseudocódigo con los pasos necesarios para hacer avanzar el frente en el vóxel asociado al hilo en función del valor de la función de velocidad en dicho vóxel. Este código es ejecutado por todos los hilos del cada bloque de hilos. Hemos utilizado la siguiente notación: u_c es el contenido del conjunto de nivel en la memoria compartida; f es el valor de la función de velocidad en el vóxel procesado; por último, i_c es la coordenada del vóxel procesado en memoria compartida, y j_c es la coordenada del vóxel adyacente.

1: **si** $u_c(i_c) = -1$ **y** $f < 0$:
2: $\quad u_c(i_c) \leftarrow +1$

3: **si** $u_c(i_c) = -3$ **y** $\exists j_c \in \eta(i_c), u_c(j_c) = +1$:
4: $\quad u_c(i_c) \leftarrow -1$

5: **si** $u_c(i_c) = +1$ **y** $\nexists j_c \in \eta(i_c), u_c(j_c) = -1$ **y** $\nexists j_c \in \eta(i_c), u_c(j_c) = -3$:
6: $\quad u_c(i_c) \leftarrow +3$

Figura 4.10: Pseudocódigo de la operación de retroceso en GPU.

En primer lugar, el hilo comprueba si su vóxel asignado pertenece a L_{ext} (línea 1 en el pseudocódigo). La implementación en GPU no almacena dos listas con los puntos de L_{int} y L_{ext}, pero teniendo el valor de u en dicho punto y considerando la ecuación (4.2), la comprobación es trivial. Si es el caso, y la función de velocidad tiene signo positivo en ese punto, se cambia el valor del conjunto de nivel en ese punto para que pase a pertenecer a L_{int} (línea 2). Es decir, el frente se mueve efectivamente hacia el exterior.

A continuación, debe realizarse una serie de comprobaciones (líneas 3 y 5) entre los vóxeles vecinos para mantener la consistencia del frente. Esta serie de comprobaciones tiene como resultado que los vóxeles adyacentes a i_c que eran puntos externos ($u = +3$) son añadidos a L_{ext} (su valor de u es cambiado a tal efecto en la línea 4), y que los posibles puntos de L_{int} que ahora resulten redundantes se convierten ahora en puntos internos ($u = -3$ en la línea 6). Para que todo el proceso se realice de forma coherente, entre cada paso es necesario colocar barreras de sincronización que bloqueen a todos los hilos del bloque hasta que el paso haya sido realizado.

Las operaciones que se realizan durante el paso de *retroceso* son bastante parecidas a las del paso de *avance*, como se muestra en el pseudocódigo de la figura 4.10. Hemos utilizado la misma notación que antes. Cada hilo comprueba si su vóxel asignado pertenece a L_{int}, y en tal caso si el valor de la función de velocidad es de signo negativo (línea 1). Si es así, el vóxel cambia su valor para pasar a formar parte de L_{ext}, lo que significa que el frente retrocede una posición (línea 2). Por último, al igual que antes, se realizan una serie de comprobaciones con el objetivo de mantener la consistencia del frente (líneas 3–6).

La figura 4.11 muestra una única iteración de la evolución del frente para dos casos, avance y retroceso, en una región de tamaño 4×4 con una zona de solapamiento de un vóxel de grosor. El estado inicial de la región en memoria compartida se muestra en la parte superior de la figura, donde el frente se encuentra representado por los vóxeles de L_{int} en gris oscuro

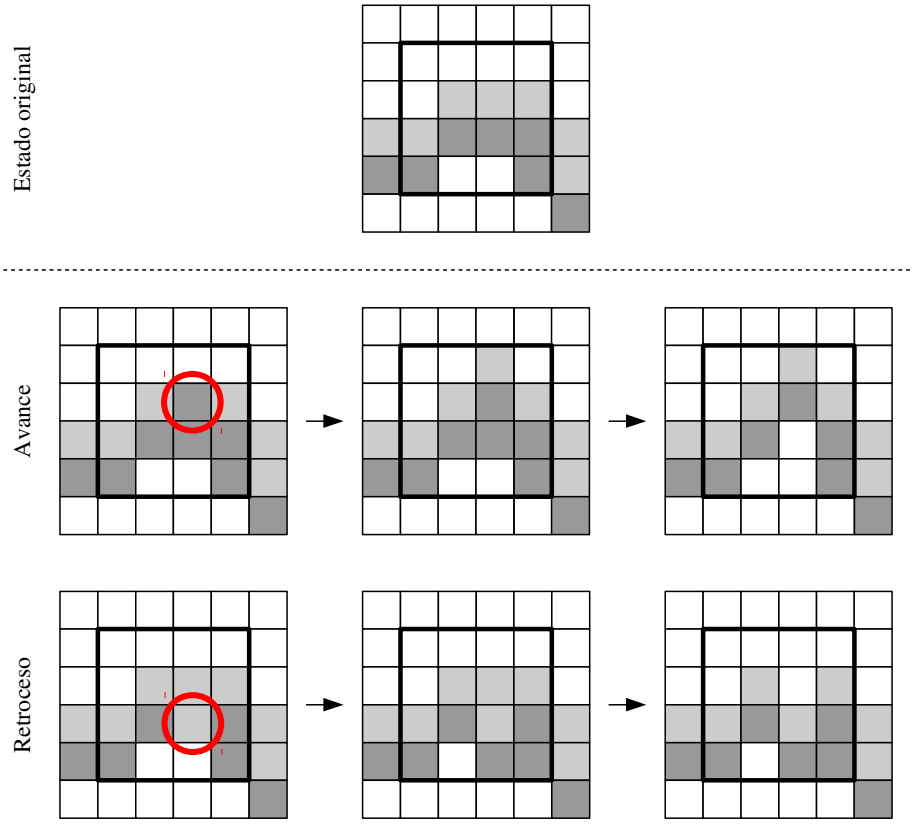


Figura 4.11: Evolución del frente en la memoria compartida de la GPU.

y de L_{ext} en gris claro. En el caso del avance, el vóxel señalado pasa a forma parte de L_{int}, e inmediatamente después, los vóxeles adyacentes externos se añaden a L_{ext}. Por último, se corrigen los vóxeles redundantes. En la parte inferior de la figura se muestra un ejemplo de retroceso. El vóxel señalado pasa a formar parte de L_{ext}, y los vóxeles internos adyacentes se añaden a L_{int}. Al igual que antes, el proceso termina corrigiendo los vóxeles redundantes.

Identificación de las regiones activas

El proceso de construcción de la lista de regiones activas se realiza por compactación, de forma similar a como se hace durante la inicialización (véase sección 4.3.1). Una vez que se ha

ejecutado el código del primer y del segundo ciclo y se ha hecho avanzar o retroceder el frente, antes de que termine la ejecución de los los *kernels*, un hilo en cada bloque de hilos realiza el proceso de exploración, comprobando si la región asignada al bloque y las adyacentes pueden ser consideradas activas para la próxima iteración del algoritmo. La región actual se considera activa si contiene al menos un vóxel que pertenezca a L_{ext}:

$$\exists \mathbf{x} \in \text{Región}(u_c), \mathbf{x} \in L_{ext}. \tag{4.10}$$

Además, una región adyacente se considera activa si en el límite de la región actual contiene al menos un vóxel de L_{ext}.

Los regiones activas se marcan en un vector temporal de enteros, sobre el que luego se realiza una suma exclusiva de prefijos, donde se obtiene la posición de las coordenadas de cada región activa en la nueva lista de regiones activas. Durante el paso de dispersión se construye esta nueva lista, almacenando cada coordenada en la posición indicada.

Comprobación de la condición de parada

Después de que se ejecute el *kernel* del primer ciclo y se actualice la lista de regiones activas, se comprueba si se verifica la condición de parada, definida en la ecuación (4.3). Este proceso se resuelve en una llamada a un *kernel*, que lanza tantos hilos como regiones activas. Cada hilo evalúa la condición de parada para cada vóxel de su región asignada. Para ello, se calcula el valor de F_1 para todos los elementos que pertenecen al frente (tanto en L_{int} como en L_{ext}). Además, las posibles inconsistencias que se encuentren como consecuencia de haber actualizado la lista de regiones activas deben ser corregidas antes de comprobar la condición de parada.

Como resultado del *kernel*, y tras realizar un proceso de compactación, se obtiene una lista con las regiones activas donde no se verifica la condición de parada. Si la lista contiene al menos un elemento, significa que la condición de parada no está satisfecha y el proceso de segmentación debe continuar (siempre y cuando no se haya alcanzado el número máximo de iteraciones).

4.3.3. Diferencias entre la primera propuesta y la segunda

Hasta ahora se han explicado las características comunes a las dos propuestas de implementación en GPU del método FTC. Nuestra primera propuesta está implementada de acuerdo

1: **ejecutar** varias iteraciones hasta satisfacer la condición de parada:
2: ejecutar líneas 2–9 del pseudocódigo de la figura 4.3
(usando la condición más restrictiva de la ecuación 4.11)

3: recalcular la lista de regiones activas
(usando la condición menos restrictiva de la ecuación 4.10)

4: **ejecutar** una iteración:
5: ejecutar líneas 2–9 del pseudocódigo de la figura 4.3

Figura 4.12: Pseudocódigo de la segunda propuesta de implementación del método de segmentación FTC en GPU.

con el pseudocódigo de la figura 4.3, que ha sido analizado a lo largo de las secciones previas. Nuestra segunda propuesta es una modificación de la primera y utiliza un criterio diferente para seleccionar el conjunto de regiones activas. El objetivo es reducir el tamaño del dominio activo eliminando aquellas regiones donde el frente ya se ha estabilizado. Así se evita lanzar bloques de hilos que hagan cálculos sobre regiones donde el frente ya no puede avanzar ni retroceder. La segunda propuesta requiere una iteración final procesando todo el frente.

Durante el proceso de segmentación, es habitual que el frente crezca en tamaño y complejidad, lo que se traduce en un incremento en el número de regiones activas. Sin embargo, solo una porción del frente evoluciona en cualquier instante del tiempo, ya que hay partes del frente que se estabilizan (esto es, convergen) mucho antes de que la segmentación finalice. En la primera propuesta, las regiones atravesadas por cualquier porción del frente, esté estabilizada o no, se consideran activas, aunque realmente no requieran ningún cálculo adicional (véase ecuación (4.10)). En nuestra segunda propuesta, consideramos una región activa si, y solo si, verifica la siguiente condición:

$$\forall \mathbf{x} \in \text{Región}(u_c), \begin{cases} F_1(\mathbf{x}) > 0 & \text{si } \mathbf{x} \in L_{ext}, \\ F_1(\mathbf{x}) < 0 & \text{si } \mathbf{x} \in L_{int}. \end{cases} \tag{4.11}$$

Nótese que esta comprobación es la misma que la condición de parada especificada en la ecuación (4.3).

La figura 4.12 muestra el flujo de trabajo de la segunda propuesta. Como puede observarse, se ejecuta el mismo algoritmo que en la primera propuesta, pero cambiando el criterio para identificar regiones activas. Al final del algoritmo (líneas 4–5), se aplica una iteración adicional, aunque sobre un dominio activo calculado usando la condición menos restrictiva, es decir, sobre un dominio activo que contiene todas la regiones atravesadas por el frente. Esto

implica que el proceso de regularización se realiza en todos los puntos del frente, al igual que se hace en el método FTC original. Como veremos en la sección 4.4, el coste computacional de esta operación es mayor que el coste de todas las iteraciones previas. Sin embargo, los resultados muestran que incluso así el rendimiento de la segunda propuesta es mejor que el de la primera, sin que ello implique una pérdida significativa de calidad.

4.4. Resultados

En esta sección presentamos los resultados de rendimiento y calidad obtenidos para nuestras dos propuestas de implementación del método de segmentación FTC.

4.4.1. Metodología

Realizamos nuestras pruebas en una NVIDIA GeForce GTX 580, cuyas características pueden encontrarse en la sección 1.7). La versión original del método FTC fue evaluada en un Intel Core i7 con 4 *cores* a una frecuencia de reloj de 2,8 GHz (que puede alcanzar los 3,7 GHz para tareas computacionales intensivas que solo requieran un hilo de ejecución) y 8 GB de RAM [85]. Cada *core* tiene cachés L1 de 64 kB separadas para datos y para código, y una caché L2 unificada de 256 kB por core. La arquitectura suma una caché adicional L3 de 8 MB compartida entre los cuatro cores.

Dado que el principal objetivo de estas implementaciones era conseguir tiempos de ejecución cercanos al tiempo real, comparamos nuestras implementaciones en CUDA del método FTC con una implementación en OpenMP del método original, compilada con *gcc* en un sistema operativo Linux. Para este fin, usamos la misma función de velocidad que utilizan los autores del método FTC utilizan en [162] para segmentar volúmenes:

$$F_1(\mathbf{x}) = \begin{cases} +1 & \text{si } V(\mathbf{x}) \in [V_1, V_2], \\ -1 & \text{en otro caso,} \end{cases} \tag{4.12}$$

donde V es el volumen sobre el que se va a realizar la segmentación, y $[V_1, V_2]$ es el rango de intensidades a segmentar. En [111] se utiliza también una función de velocidad similar.

Evaluamos el rendimiento midiendo los tiempos de ejecución y los valores de aceleración de nuestras propuestas, y comparamos nuestros resultados con la ejecución en CPU del método FTC. Para evaluar la calidad de nuestros resultados, utilizamos el coeficiente Dice (véase sección 1.6).

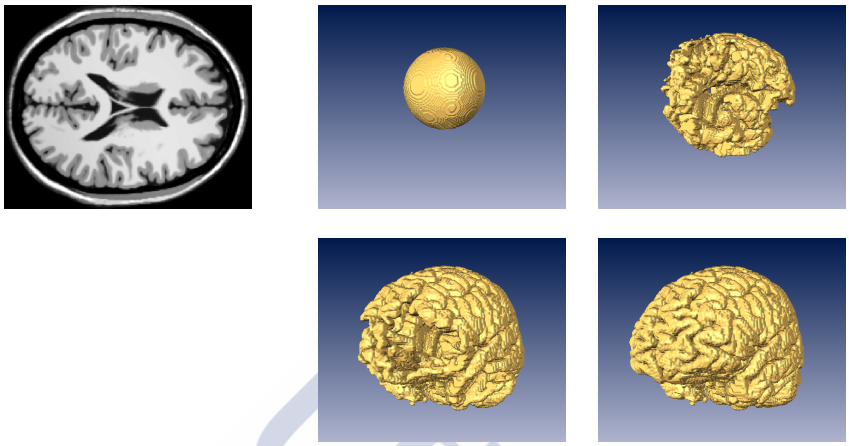


Figura 4.13: Visualización de los resultados de segmentación del volumen `brainweb-1`. A la izquierda se muestran los datos originales. A la derecha se muestran cuatro pasos (ordenados de izquierda a derecha y de arriba a abajo) del proceso de segmentación.

4.4.2. Conjuntos de datos

Presentamos resultados de nuestras dos propuestas de implementación en GPU del método de segmentación FTC para los conjuntos de datos `brainweb-1`, que se corresponde con una imagen MRI del cerebro, `vessels`, un conjunto de imágenes con vasos sanguíneos de alto contraste, tres imágenes CT (`chest`, `kidneys` y `abdomen`) de la base de datos de Osirix, y cuatro imágenes MRI del cerebro obtenidas de la base de datos Oasis (denominadas `oasis-n`). Pueden encontrarse más detalles de estos conjuntos de datos en la sección 1.7.

Las figuras 4.13–4.18 muestran, en cada caso, una lámina bidimensional de una selección de los volúmenes usados en nuestras pruebas, y un ejemplo de segmentación realizado con la implementación en GPU. El volumen de segmentación se visualiza en diferentes estados del proceso utilizando una isosuperficie. El proceso de segmentación comienza con una semilla esférica situada en el interior del volumen (excepto en el caso de 4.16, donde se utilizaron dos semillas para segmentar los dos riñones de forma simultánea). A cada iteración, la esfera incrementa su tamaño y adquiere la forma de la región segmentada.

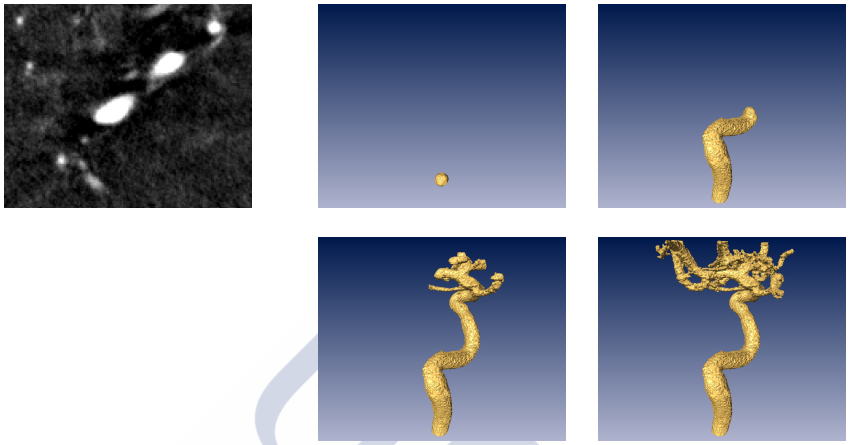


Figura 4.14: Visualización de los resultados de segmentación del volumen `vessels-1`. A la izquierda se muestran los datos originales. A la derecha se muestran cuatro pasos (ordenados de izquierda a derecha y de arriba a abajo) del proceso de segmentación.

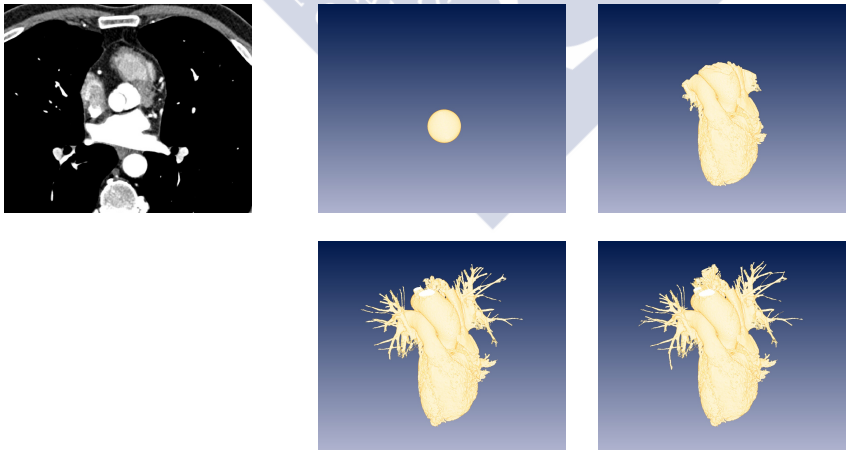


Figura 4.15: Visualización de los resultados de segmentación del volumen `chest`. A la izquierda se muestran los datos originales. A la derecha se muestran cuatro pasos (ordenados de izquierda a derecha y de arriba a abajo) del proceso de segmentación.

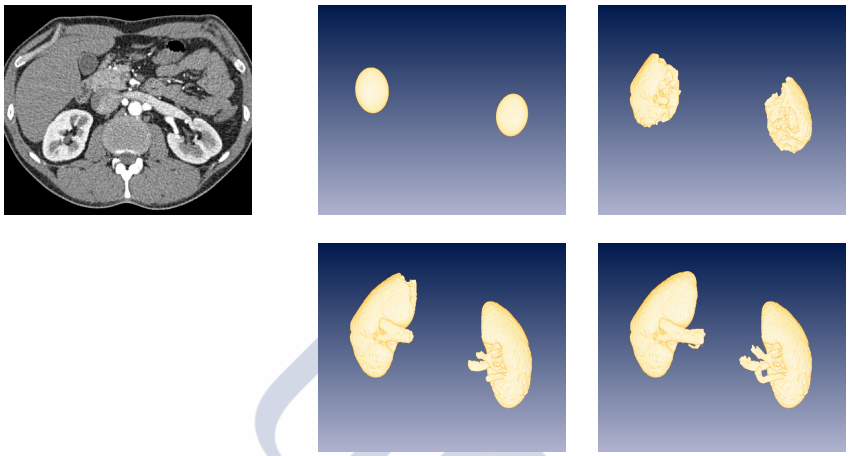


Figura 4.16: Visualización de los resultados de segmentación del volumen `kidneys`. A la izquierda se muestran los datos originales. A la derecha se muestran cuatro pasos (ordenados de izquierda a derecha y de arriba a abajo) del proceso de segmentación.

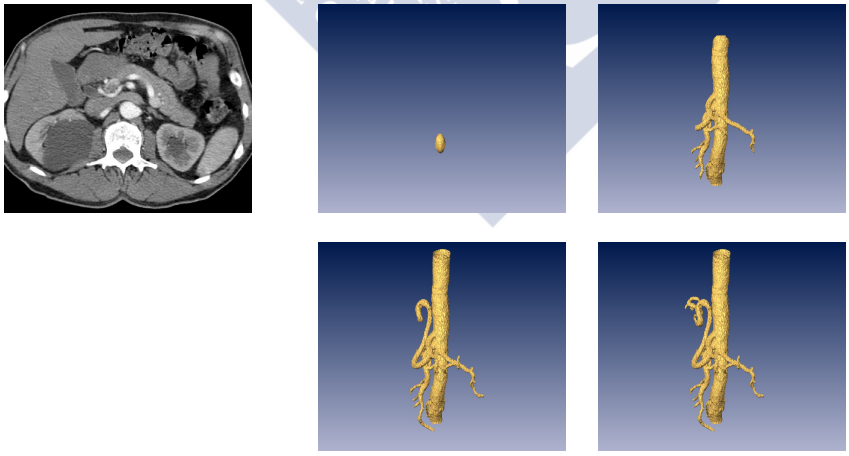


Figura 4.17: Visualización de los resultados de segmentación del volumen `abdomen`. A la izquierda se muestran los datos originales. A la derecha se muestran cuatro pasos (ordenados de izquierda a derecha y de arriba a abajo) del proceso de segmentación.

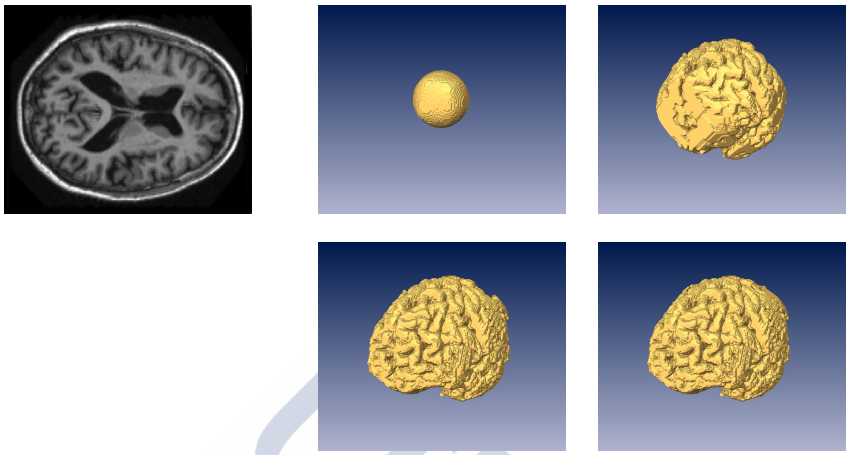


Figura 4.18: Visualización de los resultados de segmentación del volumen `oasis-1`. A la izquierda se muestran los datos originales. A la derecha se muestran cuatro pasos (ordenados de izquierda a derecha y de arriba a abajo) del proceso de segmentación.

Volumen	Posición de la semilla	Radio de la semilla	Rango de intensidades
`brainweb-1`	(100, 103, 90)	40	(120, 160)
`vessels-1`	(126, 134, 26)	10	(600, 6000)
`vessels-2`	(25, 157, 17)	10	(233, 7000)
`vessels-3`	(118, 150, 15)	5	(460, 6000)
`vessels-4`	(164, 122, 23)	4	(0, 10800)
`chest`	(264, 229, 224)	50	(145, 245)
`kidneys`	(117, 307, 159)	30	(200, 243)
`abdomen`	(233, 171, 116)	9	(200, 250)
`oasis-n`	(88, 104, 88)	30	(2300, 3300)

Tabla 4.1: Parámetros comunes de segmentación para los volúmenes usados en nuestras pruebas .

4.4.3. Medidas de rendimiento

La tabla 4.1 muestra los parámetros de segmentación usados en nuestros experimentos que son comunes a las implementaciones en GPU y en CPU del método FTC: la posición y el tamaño de la semilla inicial, y el rango de intensidades utilizado para realizar la segmentación. Para los cuatro volúmenes `oasis-n`, al tratarse de conjuntos de datos muy similares, utilizamos los mismos parámetros en todos los casos.

La tabla 4.2 detalla los valores de los parámetros específicos de cada una de las tres implementaciones del método FTC para los diferentes casos de estudio considerados en esta sección, junto con los resultados obtenidos en términos de tiempo y aceleración. N_0 es el número de veces que el *kernel* del primer ciclo es invocado en cada iteración completa del algoritmo. N_1 y N_2 representan, respectivamente, el número de iteraciones que se realizan dentro del primer ciclo y del segundo (implementación en CPU) o dentro del *kernel* del primer y del segundo ciclo ciclo (implementación en GPU). N_k es el tamaño del filtro de suavizado, y σ es el término de varianza, ambos utilizados durante el proceso de regularización del segundo ciclo. Los valores de los parámetros fueron escogidos a partir de los utilizados en [162], buscando en cada caso el mejor resultado de segmentación.

Las medidas de tiempo y aceleración que aparecen en la misma tabla demuestran que nuestras propuestas son adecuadas para acelerar el proceso de segmentación sin necesidad de recurrir a costosos computadores de altas prestaciones. La primera propuesta de implementación en CUDA es, incluso en el caso de volúmenes de segmentación pequeños (`vessels`), casi tan rápida como la implementación en OpenMP. La segunda propuesta es entre 3 y 10 veces más rápida que la implementación en OpenMP, por lo que se observa que reducir el dominio activo proporciona mejores resultados de rendimiento. Las figuras 4.19–4.21 muestran de forma gráfica las medidas de tiempo y aceleración detalladas en la tabla.

Para estudiar la influencia del tamaño del dominio activo y del número de iteraciones en el tiempo de ejecución realizamos experimentos adicionales sobre los conjunto de datos `brainweb-1` y `vessels-1`. La figura 4.22 muestra el número de regiones activas cada vez que se actualiza el dominio activo para las dos propuestas de implementación en GPU del método de segmentación FTC. En el caso de `brainweb-1` la segmentación comienza con cerca de 500 regiones activas, y en la primera propuesta este número llega a ser cerca de 4000 hacia el final de la segmentación. Se observan algunas caídas cada vez que se ejecuta el segundo ciclo, ya que la regularización tiene el efecto de encoger ligeramente el volumen de segmentación. En la segunda propuesta este comportamiento es diferente: el número de regiones activas crece hasta cerca de las 1500 y luego experimenta un descenso hasta casi el final del algoritmo, con algunos picos después de que se ejecute el segundo ciclo (este efecto se explicará más adelante). En las última iteraciones de la segunda propuesta se ejecutan todos los pasos del primer y del segundo ciclo en todas las regiones atravesadas por el frente, por lo que el número de regiones activas es el mismo que el número de regiones atravesadas por el frente.

Volumen		N_0	N_1	N_2	N_k	σ	Tiempo	Aceleración
brainweb-1	CPU	7	4	2	1	2	1,608 s	1,0x
	GPU 1ª prop.	5	6	3	3	2	0,957 s	1,7x
	GPU 2ª prop.	5	6	3	3	2	0,565 s	2,8x
vessels-1	CPU	7	4	3	3	1	1,234 s	1,0x
	GPU 1ª prop.	6	6	3	3	1	0,977 s	1,3x
	GPU 2ª prop.	6	6	3	3	1	0,222 s	5,6x
vessels-2	CPU	7	4	3	3	1	1,012 s	1,0x
	GPU 1ª prop.	5	6	3	3	1	0,789 s	1,3x
	GPU 2ª prop.	5	6	3	3	1	0,205 s	4,9x
vessels-3	CPU	7	4	3	3	1	1,252 s	1,0x
	GPU 1ª prop.	5	6	3	3	1	1,447 s	0,9x
	GPU 2ª prop.	5	6	3	3	1	0,240 s	5,2x
vessels-4	CPU	7	4	3	3	1	0,266 s	1,0x
	GPU 1ª prop.	5	6	3	3	1	0,310 s	0,9x
	GPU 2ª prop.	5	6	3	3	1	0,094 s	2,8x
chest	CPU	11	4	1	3	1	18,504 s	1,0x
	GPU 1ª prop.	9	5	1	3	1	11,871 s	1,6x
	GPU 2ª prop.	9	5	1	3	1	3,020 s	6,1x
kidneys	CPU	8	4	3	3	2	4,107 s	1,0x
	GPU 1ª prop.	6	5	3	3	2	0,600 s	6,8x
	GPU 2ª prop.	6	5	3	3	2	0,389 s	10,6x
abdomen	CPU	12	4	0	3	1	1,628 s	1,0x
	GPU 1ª prop.	10	5	0	3	1	0,725 s	2,2x
	GPU 2ª prop.	10	5	0	3	1	0,266 s	6,1x
oasis-1	CPU	11	4	1	3	1	3,330 s	1,0x
	GPU 1ª prop.	9	5	1	3	1	1,111 s	3,0x
	GPU 2ª prop.	9	5	1	3	1	0,597 s	5,6x
oasis-2	CPU	4	4	4	3	1	5,842 s	1,0x
	GPU 1ª prop.	3	5	4	3	1	0,867 s	6,7x
	GPU 2ª prop.	3	5	4	3	1	0,664 s	8,8x
oasis-3	CPU	8	4	3	3	1	5,447 s	1,0x
	GPU 1ª prop.	6	5	3	3	1	0,663 s	8,2x
	GPU 2ª prop.	6	5	3	3	1	0,654 s	8,3x
oasis-4	CPU	9	4	2	3	1	4,683 s	1,0x
	GPU 1ª prop.	7	5	2	3	1	1,041 s	4,5x
	GPU 2ª prop.	7	5	2	3	1	0,626 s	7,5x

Tabla 4.2: Parámetros de segmentación y valores de rendimiento de las implementaciones en OpenMP y en una tarjeta GTX 580 obtenidos para los diferentes conjuntos de datos utilizados en nuestros experimentos.

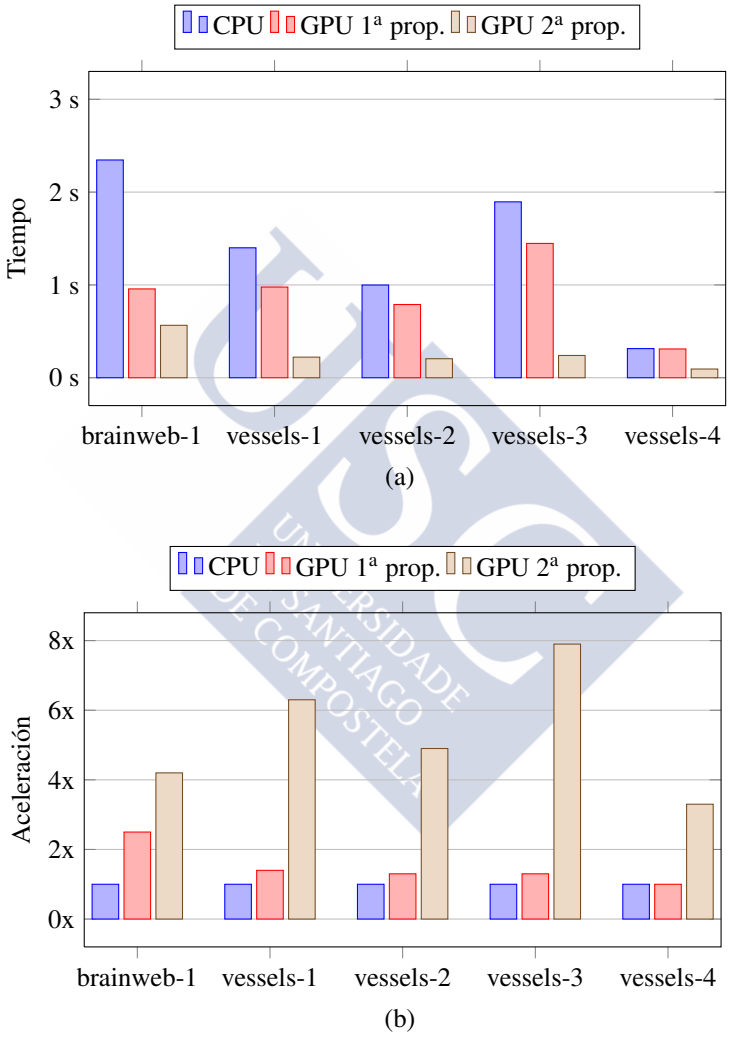


Figura 4.19: Medidas de rendimiento en OpenMP y en una tarjeta GTX 580 de las diferentes implementaciones del método de segmentación FTC para los volúmenes `brainweb-1` y `vessels-n`. Se muestran (a) el tiempo consumido y (b) los valores de aceleración.

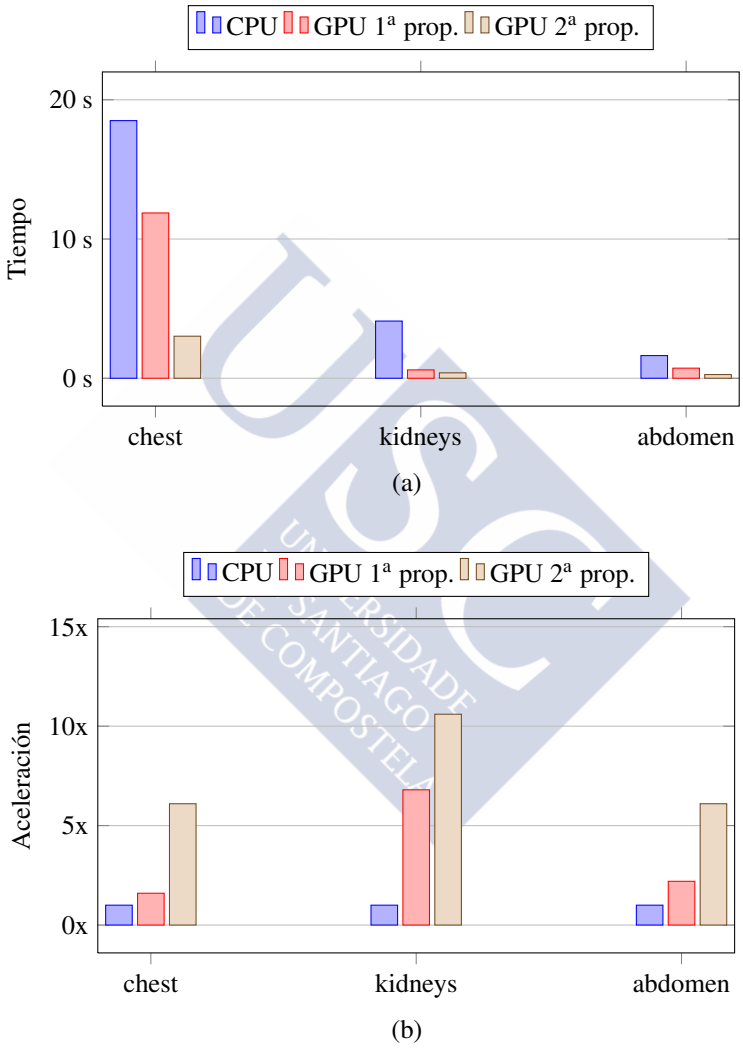


Figura 4.20: Medidas de rendimiento en OpenMP y en una tarjeta GTX 580 de las diferentes implementaciones del método de segmentación FTC para los volúmenes `chest`, `kidneys` y `abdomen`. Se muestran (a) el tiempo consumido y (b) los valores de aceleración.

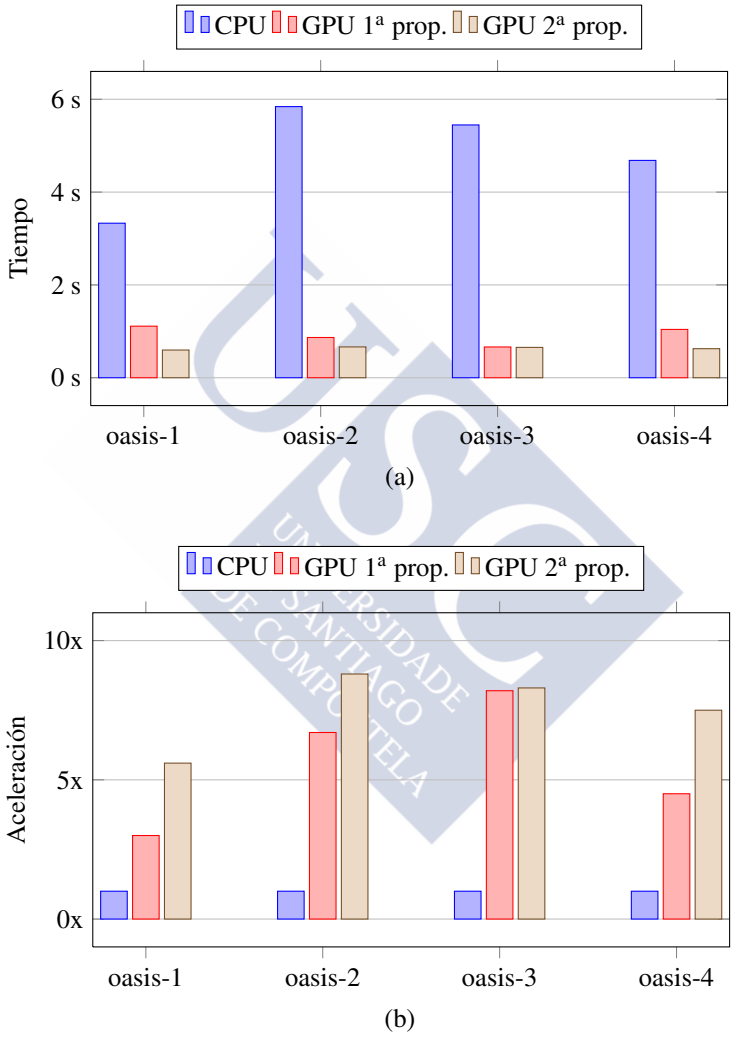


Figura 4.21: Medidas de rendimiento en OpenMP y en una tarjeta GTX 580 de las diferentes implementaciones del método de segmentación FTC para los volúmenes `oasis-n`. Se muestran (a) el tiempo consumido y (b) los valores de aceleración.

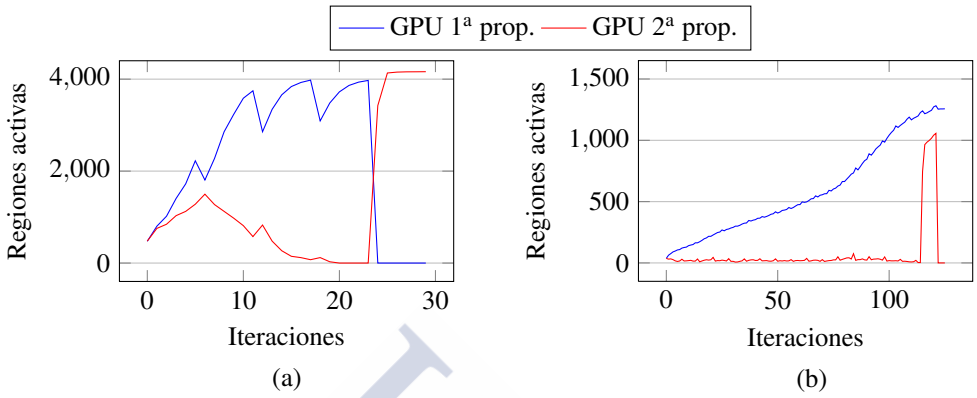


Figura 4.22: Regiones activas durante la segmentación de los conjutnos de datos (a) `brainweb-1` y (b) `vessels-1` con las dos propuestas de implementación en GPU del método FTC.

El método de segmentación FTC utiliza dos funciones de velocidad. Durante el primer ciclo se utiliza la función F_1, que en los casos estudiados en esta sección hace evolucionar el frente en función de los valores de intensidad de los vóxeles del volumen. En el segundo ciclo la función de velocidad utilizada, F_2, no considera las intensidades del volumen a segmentar, sino la forma del frente. Siguiendo el flujo de trabajo del método FTC descrito en el pseudocódigo de la figura 4.2, los avances y retrocesos durante el primer ciclo se producen solo en aquellos vóxeles del frente que no verifican la condición de parada (ecuación (4.3)), es decir, vóxeles que aún no se han estabilizado. Sin embargo, en el segundo ciclo los avances y retrocesos pueden producirse en cualquier vóxel del frente, estabilizado o no, dependiendo del signo de F_2. Por tanto, durante el segundo ciclo es habitual que algunos vóxeles estabilizados cambien de valor, dejando de verificar la condición de parada. Desde el punto de vista de nuestra segunda propuesta de implementación, esto significa que la región sigue siendo activa (y posiblemente también las adyacentes). Esto explica que tras cada segundo ciclo el número de regiones activas aumente visiblemente para la segunda propuesta, como se puede apreciar en la figura 4.22.

La figura también muestra que la variación en el número de regiones activas depende fundamentalmente del volumen a segmentar. En el caso del conjunto de datos `vessels-1`, el número de regiones activas es mucho menor que en el caso anterior. Sin embargo, el comportamiento es similar. En la primera propuesta el número de regiones activas crece de forma casi constante, desde cerca de 100 regiones activas al principio hasta llegar a casi 1300, con

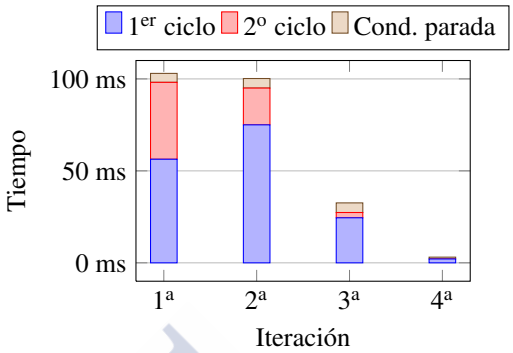


Figura 4.23: Tiempo de computación en una tarjeta GTX 580 para cada iteración de la segunda propuesta de implementación del método FTC para segmentar el conjunto de datos `brainweb-1`.

pequeñas caídas debidas al segundo ciclo. En la segunda propuesta el número de regiones activas a lo largo de la ejecución es claramente inferior, ya que son pocas las regiones donde el frente evoluciona. Además, dado que en ambas propuestas el dominio activo es diferente, no es raro que el número de iteraciones para completar la segmentación sea diferente: en este caso la segunda propuesta termina unas iteraciones antes que la primera, y al igual que antes, las últimas iteraciones el dominio activo abarca todas las regiones atravesadas por el frente.

La figura 4.23 muestra una gráfica acumulativa del tiempo de computación para la segmentación de `brainweb-1` distribuido entre el primer ciclo, el segundo ciclo, y la comprobación de la condición de parada para la segunda propuesta de implementación en GPU del método FTC. La gráfica muestra los tiempos para las cuatro primeras iteraciones globales del algoritmo, y no incluye la iteración final. Se observa que prácticamente todo el tiempo de computación se distribuye entre la ejecución del primer ciclo y la del segundo, aunque el primer ciclo requiere mas tiempo porque se ejecuta más veces en cada iteración global. El tiempo dedicado a actualizar la lista de regiones activas es muy pequeño y no se representa en la gráfica. Si incluimos la iteración final, donde la evolución se calcula para todo el frente y no solo en las regiones activas, casi un 60 % del tiempo necesario para segmentar el conjunto de datos `brainweb-1` se consume en esta iteración.

La tabla 4.3 muestra el rendimiento de la segunda propuesta de implementación en GPU del método FTC segmentando regiones de interés de diferentes tamaños extraídas del volumen `vessels-1`. Para cada tamaño, se muestran el número de iteraciones globales ejecutadas para completar la segmentación, la duración media de cada iteración y el tiempo total para

Tamaño	Iteraciones	Media	Total
32^3	1	0,007 s	0,007 s
64^3	2	0,010 s	0,020 s
128^3	8	0,010 s	0,084 s
256^3	15	0,018 s	0,268 s
512^3	15	0,028 s	0,423 s

Tabla 4.3: Resultados de segmentar en una tarjeta GTX 580 volúmenes de diferentes tamaños extraídos del conjunto de datos `vessels-1` usando la segunda propuesta de implementación del método FTC.

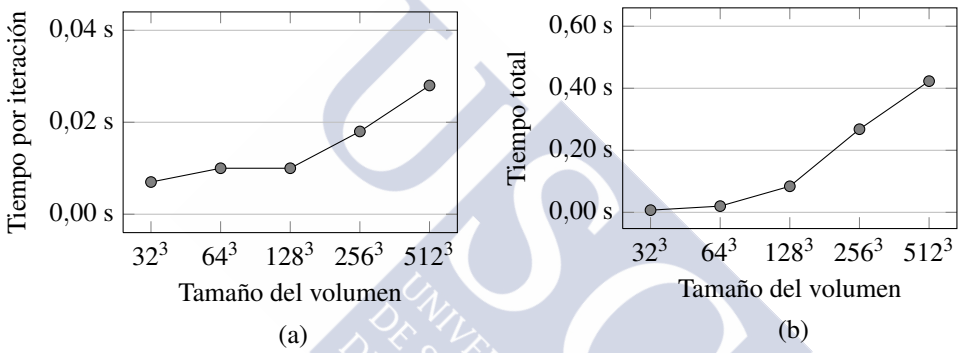


Figura 4.24: Rendimiento de la segmentación una tarjeta GTX 580 de volúmenes de diferentes tamaños extraídos del conjunto de datos `vessels-1`. Se muestran (a) el tiempo de ejecución de cada iteración y (b) el tiempo total para completar la segmentación.

completar la segmentación. La figura 4.24 muestra estos mismos datos en forma de gráfica. En todos los casos se ajustaron los parámetros para tratar de conseguir una segmentación lo más correcta posible de los vasos sanguíneos en el interior del volumen. Aunque el tiempo de segmentación depende de la región del volumen que se pretende segmentar, en este caso el tiempo se duplica cada vez que aumentamos el tamaño del volumen un múltiplo de ocho.

4.4.4. Medidas de calidad

Para evaluar la calidad de las segmentaciones y validar nuestra implementación hemos utilizado el coeficiente Dice (véase sección 1.6). Tomamos medidas de calidad para el conjunto de datos `brainweb-1`, del que existe una segmentación de referencia que distingue entre varios tipos de tejidos. En nuestro caso, nuestra segmentación pretendía segmentar la materia gris y la materia blanca del cerebro. La implementación en CPU del método FTC proporcionó

unos resultados de segmentación con un valor del índice Dice de 95 %. Nuestras dos propuestas en GPU sobre el mismo volumen consiguieron un valor del 96 %, por lo que podemos decir que ambas implementaciones obtienen resultados similares a la versión secuencial de referencia del método FTC.

4.4.5. Comparación con otros trabajos

La comparación de nuestros resultados con otros trabajos relacionados con los métodos de segmentación del conjunto de nivel no es una tarea directa. En general, los métodos del conjunto de nivel tienen una gran variedad de aplicaciones, e incluso en el campo de la segmentación, hay una gran cantidad de soluciones específicas con las que no es posible compararse directamente. Además, no siempre es posible acceder a las mismas bases de datos que se utilizan en otros trabajos, ni todas las soluciones están centradas en optimizar el tiempo de computación.

En [145] se describe una de las primeras implementaciones de un método de segmentación del conjunto de nivel en CUDA. Esta implementación está basada en el trabajo previo propuesto en [108, 109, 110, 111] y, de forma similar a nuestro trabajo, también mantiene un dominio activo, aunque en este caso está formado por vóxeles cercanos a las partes del frente que todavía no han estabilizado. Los autores afirman que gestionar este dominio activo consume un 77 % del tiempo total de computación, y presentan también resultados para segmentar la materia gris y blanca de una imagen obtenida de la BrainWeb Database con un tamaño de 256^3. Su solución requiere 7 s para completar la segmentación en una NVIDIA GTX 280 [5].

Nuestras propuestas de implementación realizan la gestión del dominio activo por regiones, con lo que el tiempo de gestión es despreciable. Para comparar el rendimiento con el trabajo publicado en [145], hemos evaluado nuestras implementaciones en una NVIDIA GTX 295 [6], que tiene dos núcleos, cada uno con características similares al único núcleo GTX 280 (aunque con una frecuencia de reloj y de memoria ligeramente superiores). En esta comparativa hemos utilizados solo uno de los núcleos de la GTX 295. Nuestra segunda propuesta necesitó 1,1 s para segmentar un volumen similar de la BrainWeb Database de tamaño de $256 \times 256 \times 181$. Podemos concluir, por tanto, que nuestra aproximación puede ser más de seis veces más rápida que la utilizada en [145].

En [87] se presenta una aproximación más reciente realizada en CUDA aplicada a la reconstrucción de superficies. En esta solución el dominio se divide en regiones y se utiliza una lista ordenada de regiones activas, basándose en la implementación presentada en [178]. Los

autores muestran resultados del rendimiento de su solución reduciendo el conocido conjunto de datos Stanford Dragon (que puede obtenerse del Stanford 3D Scanning Repository [21]) reconstruido en una malla de 512^3 vóxeles. Este proceso consume 10 s en una NVIDIA GTX 280. Repetimos la misma prueba en una NVIDIA GTX 295, y obtuvimos que segunda propuesta requiere 2,9 s en completar la misma tarea. Por las razones ya expuestas, nuestra implementación es más de tres veces más rápida.

4.5. Conclusiones

En este capítulo hemos presentado dos propuestas de implementación en GPU del método de segmentación FTC. Estas propuestas se caracterizan por evitar las computaciones innecesarias e incrementar el rendimiento sin que esto suponga una pérdida de calidad respecto al método original.

En ambas propuestas dividimos el dominio computacional en regiones del mismo tamaño que pueden ser almacenadas en memoria compartida y que los bloques de hilos de la GPU pueden procesar en paralelo de forma eficiente. Esta división requiere tener en cuenta la zona de solapamiento de cada región y corregir las inconsistencias entre regiones adyacentes. Modificamos la estructura iterativa del algoritmo para poder explotar mejor las características de la GPU, y utilizamos una lista de regiones activas para evitar realizar cálculos en regiones donde el conjunto de nivel ya se ha estabilizado. Para este fin, en nuestra primera propuesta la lista contiene todas las regiones atravesadas por el frente en cualquier instante de tiempo. En nuestra segunda propuesta utilizamos un criterio más restrictivo para reducir el tamaño del dominio activo.

Medimos el rendimiento de nuestras implementaciones en una NVIDIA GTX 580 y lo comparamos con una versión implementada en OpenMP del método FTC y ejecutada en CPU. Obtuvimos valores de aceleración entre 3x y 10x en los conjuntos de datos utilizados en nuestros experimentos. Nuestros resultados son competitivos comparados a otros métodos de segmentación basados en conjuntos de nivel implementados en GPU.

Nuestras propuestas hacen un uso intensivo del espacio de memoria compartida, que tiene una latencia similar al espacio de registros y mucho menor que cualquier otro espacio de memoria disponible en la GPU o en la CPU. La nueva arquitectura Maxwell comercializada por NVIDIA incrementa el espacio de memoria compartida por multiprocesador de 48 kB a 64 kB (aunque cada bloque de hilos continúa limitado a utilizar como máximo 48 kB). Todo

parece indicar que futuras generaciones de GPU dispondrán de mayor espacio de memoria compartida, lo que permitiría a nuestras propuestas utilizar regiones de mayor tamaño y, por tanto, mejorar el rendimiento actual al reducir el número de operaciones que se ejecutan en la memoria global.

Es posible considerar también una implementación alternativa que podría repartir las tareas de segmentación entre la CPU y la GPU. Esta aproximación requeriría necesariamente compartir datos entre las memorias de la CPU y la GPU debido a las dependencias entre regiones. Sin embargo, esta comunicación se haría a través del bus PCI, que tiene un ancho de banda mucho menor al de la memoria de la GPU, y sería muy probablemente uno de los cuellos de botella de la aplicación. Nuestra solución también podría portarse también a arquitecturas multi-CPU.

La aproximación que hemos seguido en nuestras propuestas sigue aproximadamente los pasos de particionado del problema en tareas independientes, resolución de cada tarea en el espacio de memoria compartida, y resolución de dependencias detectando y corrigiendo inconsistencias. Esta estrategia puede ser aplicada a otros problemas que tengan un patrón similar de dependencias; para poder hacerlo, los datos deben ser divisibles en porciones donde unos pocos pasos de la solución puedan ser procesados de forma independiente, y la implementación debe ser capaz de detectar y solucionar las posibles discrepancias que puedan aparecer.

Capítulo 5

Compresión y visualización

5.1. Introducción

En este capítulo presentamos las herramientas de visualización y procesado de imagen que hemos utilizado para visualizar los resultados de los algoritmos de segmentación que hemos desarrollado durante el resto de la tesis (véanse capítulos 3 y 4).

De entre todas las herramientas comerciales de procesado y visualización de imagen médica hemos seleccionado dos que están siendo utilizadas en el campo de la imagen médica tanto para investigación como con propósitos comerciales, Amira y VolV. Ambas herramientas proporcionan diversos modos de visualización de imágenes volumétricas así como mecanismos para añadir nuevas funcionalidades.

La herramienta Amira ha sido desarrollada por el Departamento de Visualización y Análisis de Datos del *Zuse-Institut Berlin* (ZIB), con el que hemos mantenido colaboración investigadora durante el desarrollo de esta tesis y que ha conducido a introducir los algoritmos de segmentación desarrollados en los capítulos 3 y 4 como módulos de Amira. Por otra parte, el desarrollo de VolV ha corrido a cargo de la Universidad de Tübingen y el *Innovation Center of Computer Assisted Surgery* (ICCAS) de la Universidad de Leipzig, con el que también hemos mantenido colaboración.

Adicionalmente, en este capítulo presentamos un esquema de visualización en tiempo real que, como principal novedad sobre los esquemas disponibles en las herramientas mencionadas, permite la visualización multirresolución de volúmenes comprimidos de tal forma que que toda el trabajo de descompresión es ejecutado en la GPU. Hemos implementado esta solución de visualización en CUDA como un módulo integrado en la herramienta VolV.

La herramienta de visualización multinivel presentada en este capítulo utiliza la técnica de *bricking*, una aproximación basada en el principio de "divide y vencerás" (véase sección 2.2.4), que consiste en dividir el volumen en particiones (o *bricks*) lo suficientemente pequeñas como para caber en la memoria de la GPU. Las particiones se cargan en un único *buffer* de textura 3D, que es reutilizado cada vez que se procesa una nueva partición. La visualización final se consigue utilizando la técnica de mapeo de texturas (o corte de texturas), que, junto con *ray casting*, es uno de los métodos más extendidos para renderizar datos volumétricos. En esta técnica de renderizado el contenido del *buffer* 3D se mapea en una geometría compuesta de polígonos planos que constituyen láminas translúcidas orientas siempre hacia la cámara; es decir, el objeto volumétrico se corta en láminas que se renderizan siempre paralelas al plano de la imagen.

La solución propuesta emplea también la transformada *wavelet* para comprimir el volumen a una forma compacta y jerarquizada. Existe una amplia colección de transformadas *wavelet* (en el capítulo 2 hemos utilizado algunos ejemplos); para esta implementación hemos seleccionado la *wavelet* Haar por ser computacionalmente sencilla y muy efectiva a la hora de realizar una reconstrucción rápida del volumen [76, 84]. La gran mayoría de los coeficientes de esta transformada se calculan como diferencias entre distintos valores del volumen de datos original. Esto significa que estos coeficientes son de pequeña magnitud, o incluso cero, y que, por tanto, pueden ser ignorados sin una pérdida significativa de información. El esquema de codificación utilizado, basado en [84], se beneficia de esta característica para obtener una forma más compacta del volumen comprimido.

En lo que se refiere al esquema de visualización multirresolución las principales aportaciones son:

– Acoplamos los procesos de descompresión y renderizado en la GPU dividiendo el volumen en particiones que son procesadas una a una. Esta técnica aprovecha la alta tasa de transferencia del bus de memoria GPU (192 GB/s en una NVIDIA GTX 580).

– El esquema de compresión utiliza una transformada *wavelet*, lo que nos permite seleccionar niveles de resolución diferentes a la hora de descomprimir cada partición y reducir así la carga de trabajo para aquellas particiones que menos información aportan al resultado final.

– El proceso de renderizado se realiza utilizando la técnica de mapeo de texturas, que ha sido adaptada a las múltiples resoluciones soportadas por nuestro método.

– Implementamos varias etapas de nuestra solución en CUDA, y medimos su efectividad comparando el rendimiento con implementaciones realizadas en CPU, para las cuales obtuvimos altos valores de aceleración. La solución completa ofrece un tasa de *frames* por segundo estable e interactiva independiente del tamaño del visor, con una buena tasa de compresión y una buena calidad visual.

El resto del capítulo está organizado como sigue. La sección 5.2 presenta herramientas comerciales y libres para la visualización de volúmenes y, en particular, las utilizadas durante este trabajo. La sección 5.3 presenta una breve revisión bibliográfica de las principales contribuciones relacionadas con la visualización de volúmenes comprimidos, y la sección 5.4 examina el diseño de nuestra solución. La sección 5.5 presenta los resultados experimentales obtenidos y compara nuestra implementación con otras soluciones similares. Por último, la sección 5.6 concluye el capítulo comentando nuestras principales contribuciones.

5.2. Herramientas de procesado y visualización

Desde que la visualización científica se convirtió en un campo independiente de investigación en 1980, se han desarrollado numerosas soluciones para la visualización de datos [165]. Algunas de estas soluciones están enfocadas a aplicaciones muy específicas, como puede ser el caso del renderizado de volúmenes para imagen médica. Sin embargo, el campo de visualización es tan complejo que actualmente ya no es factible desarrollar una aplicación desde cero [141]. Es por eso que lo habitual es recurrir a soluciones de carácter general que contienen diferentes módulos o APIs con funcionalidades básicas para el manejo de ficheros, el procesado de imágenes (filtrado, segmentación, etc), y la visualización, que pueden ser combinadas y extendidas de múltiples formas.

Hay una amplia gama de herramientas para el procesado y la visualización de imágenes y volúmenes, tanto comerciales como abiertas. Una comparativa exhaustiva está fuera del ámbito de este trabajo, pero para más información, puede referirse a [34, 141, 187]. La mayoría de estas herramientas se caracterizan por ofrecer un diseño modular y orientado a objetos y, en algunos casos, pueden ser integradas dentro de aplicaciones más grandes. Además, suelen seguir una metáfora de flujo de datos: cada módulo tiene una serie de entradas y salidas que pueden interconectarse (la salida de un módulo puede estar conectada con la entrada de otro módulo). Algunos módulos carecen de entrada alguna y simplemente cargan datos de un fichero. Otros carecen de salida, pero permiten almacenar los datos en disco o explorar en 2D o

3D los resultados obtenidos. Así, una solución completa de visualización se compone de una red de módulos conectados entre sí que puede incluir estructuras de control para implementar saltos, ramas y bucles.

En esta sección presentamos algunas herramientas conocidas para la visualización de volúmenes. Hacemos especial énfasis en Amira y VolV, que hemos utilizado para implementar los algoritmos de segmentación y visualización presentados en este trabajo.

5.2.1. Ejemplos de herramientas

El desarrollo de aplicaciones para el procesado y la visualización de imágenes puede estar soportado a múltiples niveles. Así, las herramientas se pueden clasificar en tres categorías:

- Kits de desarrollo: consisten básicamente en librerías que ofrecen soporte para tareas específicas, y por tanto, son fácilmente adaptables a cualquier aplicación.
- Entornos de desarrollo: proporcionan una aplicación completa para desarrollar nuevas soluciones.
- *Software* extensible: aplicaciones listas para usar que se pueden extender con código personalizado.

Los límites entre estas categorías no son precisos: los entornos de desarrollo y las aplicaciones extensibles suelen estar basadas en kits de desarrollo ya existentes, o incluyen sus propios kits; por otra parte, algunos kits suelen venir acompañados de aplicaciones de muestra que pueden extenderse con código personalizado.

A continuación describimos las principales características de algunas de las herramientas de visualización más destacadas:

- **VTK:** El *visualization toolkit* [23] es un kit de desarrollo de *software* de visualización que proporciona, entre otros servicios, cálculo de isosuperficies, suavizado y reducción de mallas, renderizado de volúmenes, y visualización de tensores. VTK soporta una gran variedad de formatos de ficheros, como VRML, Open Inventor y 3D Studio. Además proporciona una serie de clases para visualizar y explorar datos 3D utilizando OpenGL. Las funciones incluidas en VTK son bastante generales, y por tanto, no presentan el mejor rendimiento ni están orientadas a aplicaciones médicas. Sin embargo, al ser de código abierto, VTK es utilizado como base para otras librerías y aplicaciones más especializadas, como es el caso de VolV y MeVisLab.

– **SciRun:** Se trata de un entorno de desarrollo para la simulación y visualización creado en la Universidad de Utah (Estados Unidos) [20]. SciRun contiene módulos específicos para explorar datos volumétricos de origen médico y es especialmente versátil a la hora de ejecutar simulaciones y mostrar cómo los valores de los parámetros afectan a los resultados de la simulación.

– **MeVisLab:** Es una aplicación desarrollada por el *Center for Medical Diagnosis Systems and Visualization* (Alemania) como resultado de varios años de trabajo por parte de un gran grupo de investigación [9]. MeVisLab ofrece una funcionalidad y una interfaz muy similar a la de Amira, en la que es posible realizar "programación visual" o implementar soluciones más avanzadas utilizando Python y Javascript. Hay disponible una versión gratuita que incorpora la funcionalidad más básica.

– **Voreen:** Es una plataforma de desarrollo abierta para la visualización de volúmenes, mantenida por la Universidad de Münster (Alemania), y que se distribuye bajo una licencia GPL [25]. Consiste en una librería implementada en C++ y OpenGL, que puede ser integrada como parte de una aplicación existente, y una aplicación Qt, que puede ser ejecutada de forma independiente. La plataforma soporta varios formatos de fichero (incluyendo imágenes DICOM), y proporciona varios módulos para la visualización de datos (renderizado de volúmenes, isosuperficies, etc). La arquitectura de la plataforma es modular y permite programar visualmente soluciones de visualización.

El resto de esta sección está dedicado a Amira y VolV. Ambas herramientas se utilizan a nivel de investigación, aunque Amira, además, tiene también una versión comercial. Amira y VolV fueron las herramientas seleccionadas para realizar gran parte del trabajo de esta tesis, y durante este periodo hemos podido contar con acceso a sus equipos de desarrollo.

5.2.2. Amira

Amira [1, 165] es una plataforma *software* extensible para la visualización científica, procesado y análisis de datos 3D y 4D. El desarrollo de esta herramienta es llevado a cabo por el *FEI Visualization Sciences Group* (Francia) y el *Zuse-Institut Berlin* (Alemania). Amira es capaz de trabajar con múltiples formatos de datos (incluyendo el formato DICOM) y ofrece también una amplia variedad de técnicas de visualización.

Aunque el código de Amira es cerrado (ya que se trata de una aplicación comercial), su arquitectura es modular y fácilmente extensible. Cada módulo de Amira se implementa

como una librería que se carga de forma dinámica cuando el usuario selecciona una de las herramientas asociadas al módulo. De esta forma, es posible añadir nuevas funcionalidades a Amira sin necesidad de disponer del código fuente ni de recompilarlo. Amira incorpora de serie una amplia variedad de módulos para el análisis de datos y la visualización en numerosos campos, incluyendo el de imagen médica.

El trabajo en Amira sigue un esquema de flujo de datos mediante "programación visual". Cuando Amira carga un fichero de datos, estos aparecen en la interfaz de usuario como un objeto dentro de un *pool*. A este tipo de objetos es posible conectar uno o más módulos que realizan un procesamiento y generan a su vez nuevos resultados que se almacenan como nuevos objetos dentro del *pool*. El usuario puede continuar encadenando módulos para generar un procesamiento en cascada. Amira incorpora un lenguaje de *scripting* basado en Tcl [22] que permite acceder y modificar todas las propiedades de los objetos del *pool*, especialmente cuando se desea procesar grandes volúmenes de datos de forma automatizada. A diferencia de otras aplicaciones que utilizan esquemas de trabajo similares, los módulos de Amira suelen contener un gran número de funcionalidades.

Hemos implementado en Amira los dos algoritmos de segmentación presentados en este trabajo: el algoritmo basado en el esquema de operador aditivo (descrito en el capítulo 3) y el algoritmo de segmentación rápida de dos ciclos (descrito en el capítulo 4). Nuestras implementaciones en GPU han sido añadidas a Amira como librerías que soportan los tipos de datos usados internamente por Amira, y los resultados producidos pueden ser visualizados utilizando los módulos de visualización que incorpora Amira. Esto permite modificar los parámetros de una segmentación y visualizar en poco tiempo los resultados obtenidos. Nuestras implementaciones han sido incorporadas al repositorio de ZIBAmira [26], la versión de Amira desarrollada en el *Zuse-Institut Berlin*.

En las siguientes secciones explicamos el funcionamiento de ambas implementaciones en Amira.

Implementación de la segmentación AOS en Amira

La figura 5.1 muestra los dos módulos de los que consta la herramienta para la segmentación basada en el esquema de operador aditivo (AOS). El primero, `AOSStop`, recibe como entrada un volumen de datos uniforme, y genera los valores de la función de parada. En esta implementación, la función de parada está basada en el gradiente del volumen, tal y como se describe en la sección 3.3.1. El usuario puede configurar los valores de los parámetros σ y λ,

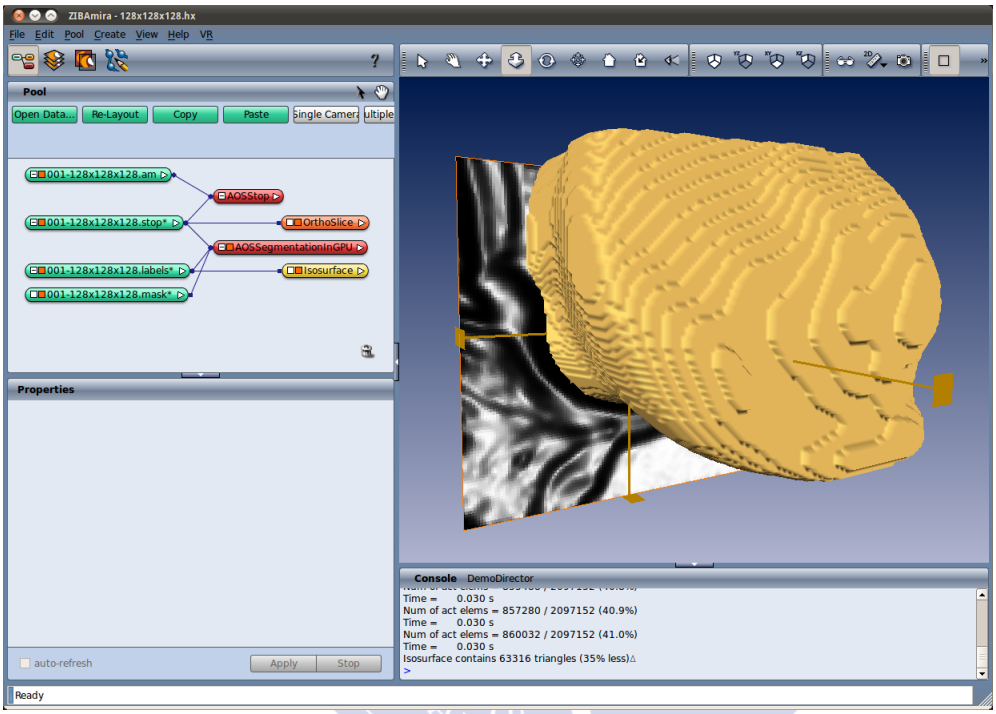


Figura 5.1: Captura de pantalla de Amira mostrando los módulos para la segmentación AOS.

y decidir si desea normalizar o no los resultados, de forma que los valores se extienden a lo largo del rango $[0, 1]$. La figura 5.2 muestra los parámetros que se pueden configurar de este módulo.

El segundo módulo, `AOSSegmentationInGPU`, realiza el proceso de segmentación en GPU. Recibe como entrada los valores de la función de parada almacenados en un volumen de datos distribuidos de forma uniforme. Naturalmente, el volumen generado por `AOSStop` sirve como función de parada, pero es posible desarrollar módulos alternativos que calculen funciones de parada basadas en otras propiedades de la imagen. El usuario configura la posición y el tamaño de la esfera que hará las veces de semilla inicial, el modelo de crecimiento (geodésico o geométrico), si desea que el volumen de segmentación se expanda o se contraiga, el número de iteraciones y los valores de los parámetros k y τ. Además, habilitando la opción de *auto-refresh* el usuario puede visualizar interactivamente el proceso de segmentación, ya

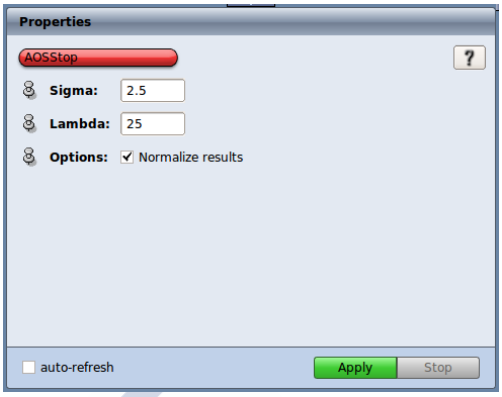


Figura 5.2: Configuración del módulo `AOSStop` en Amira.

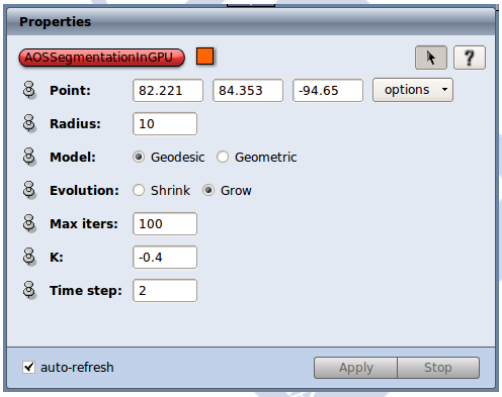


Figura 5.3: Configuración del módulo `AOSSegmentationInGPU` en Amira.

que esta se recalcula cada vez que se modifica un parámetro. La figura 5.3 muestra un ejemplo de configuración de este módulo.

La salida del módulo `AOSSegmentationInGPU` ofrece dos resultados de cada proceso de segmentación: un volumen que contiene los valores finales de la función del conjunto de nivel tras aplicar el proceso de segmentación, y un volumen de etiquetas que indica qué vóxeles pertenecen al interior de la región segmentada.

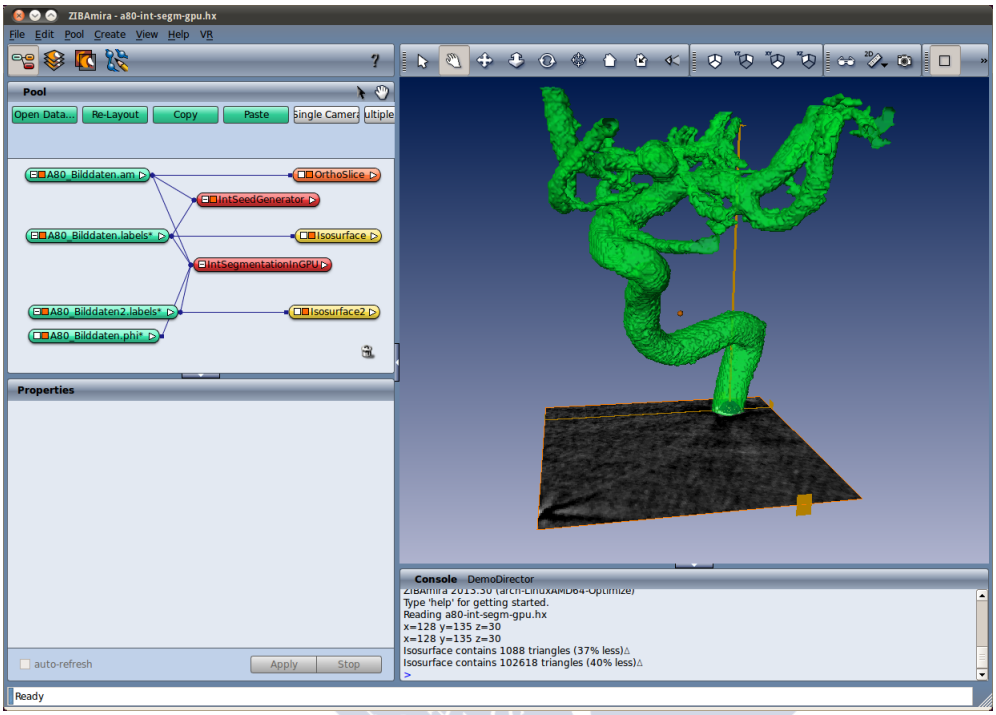


Figura 5.4: Captura de pantalla de Amira mostrando los módulos para la segmentación rápida de dos ciclos.

Implementación de la segmentación rápida de dos ciclos en Amira

La herramienta de segmentación rápida de dos ciclos también consta de dos módulos en Amira, que se muestran en la figura 5.4. El primero, `IntSeedGenerator`, recibe como entrada un volumen de datos uniforme, y genera un volumen inicial de segmentación que incluye entre una y cinco semillas con forma esférica. La figura 5.5 muestra los parámetros que se pueden configurar en este módulo desde Amira. El usuario selecciona para cada una de las cinco semillas su posición y su radio.

El segundo módulo, `IntSegmentationInGPU`, realiza el proceso de segmentación en GPU. Recibe como entrada el volumen a segmentar y el volumen inicial de segmentación generado por `IntSeedGenerator`. Dado que este volumen inicial tiene un formato fácil de modificar en Amira, pues se trata de un volumen de etiquetas, es posible utilizar otras herramientas para personalizar la forma inicial del volumen de segmentación. El usuario configura

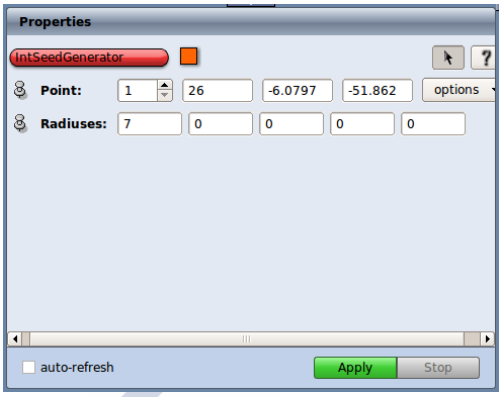


Figura 5.5: Configuración del módulo `IntSeedGenerator` en Amira.

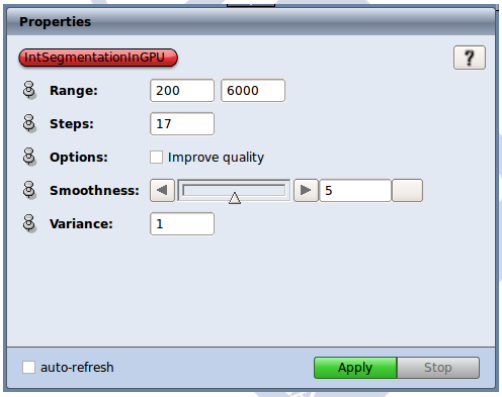


Figura 5.6: Configuración del módulo `InSegmentationInGPU` en Amira.

además el rango de valores de intensidad a segmentar de la imagen, el número máximo de iteraciones globales, y elige entre una segmentación rápida o de más calidad, la influencia de la regularización por suavizado y la varianza para construir el filtro de suavizado. Habilitando la opción de *auto-refresh* el usuario puede visualizar interactivamente el proceso de segmentación. La figura 5.6 muestra un ejemplo de configuración en Amira.

El módulo introducido en Amira sigue el esquema de nuestra segunda propuesta de implementación del método FTC (véase sección 4.3.3), pero el usuario puede elegir si desea realizar una última iteración final sobre todo el frente activando la casilla que permite mejorar la calidad. La selección de la influencia de la regularización por suavizado proporciona al usuario un

método transparente de configurar el número de iteraciones que se ejecutarán del primer y del segundo ciclo; cuando se incrementa esta influencia, se reduce el número de iteraciones del primer ciclo y se decrementa el número de iteraciones del segundo; por el contrario, reducir la influencia tiene el efecto de incrementar el número de iteraciones del primer ciclo y reducir las del segundo.

La salida del módulo `IntSegmentationInGPU` ofrece dos resultados de cada proceso de segmentación: un volumen que contiene los valores finales de la función del conjunto de nivel tras aplicar el proceso de segmentación, y un volumen de etiquetas que indica qué vóxeles pertenecen al interior de la región segmentada. Este volumen de etiquetas podría utilizarse como semilla inicial para otro proceso de segmentación que se realice con este módulo.

5.2.3. VolV

VolV *(VOLume Visualization)* [138, 141] es una plataforma para la visualización de datos médicos desarrollada en la Universidad de Tübingen (Alemania). VolV soporta una gran variedad de formatos de datos, incluyendo imágenes DICOM (el formato estándar para almacenar y transmitir información de imagen médica) e implementa numerosos métodos para visualizar datos.

El código de VolV es abierto y su arquitectura es modular, lo que permite integrar de forma sencilla nuevos módulos que amplíen su funcionalidad. El módulo principal de VolV ofrece soporte para las características básicas de visualización médica: gestión de datos, interacción con el usuario, etc. Alrededor del módulo principal hay ya desarrollados módulos que añaden funcionalidades para el procesamiento y la segmentación de imágenes, la visualización de escenas, la planificación quirúrgica, etc. Las herramientas de visualización de VolV están basadas en OpenGL y VTK (descrito más adelante).

La figura 5.7 muestra una captura de la aplicación. En la parte de la izquierda presenta una visualización 3D de tres láminas ortogonales paralelos a los planos sagital, frontal y transverso que muestran datos del volumen `modelhead`. En el centro se presentan las mismas láminas en 2D. A la derecha es posible seleccionar los posibles conjuntos de datos cargados por VolV y configurar algunas propiedades de visualización.

La solución de visualización de volúmenes comprimidos presentada en este capítulo funciona como una aplicación independiente que utiliza algunas de las funciones de VolV.

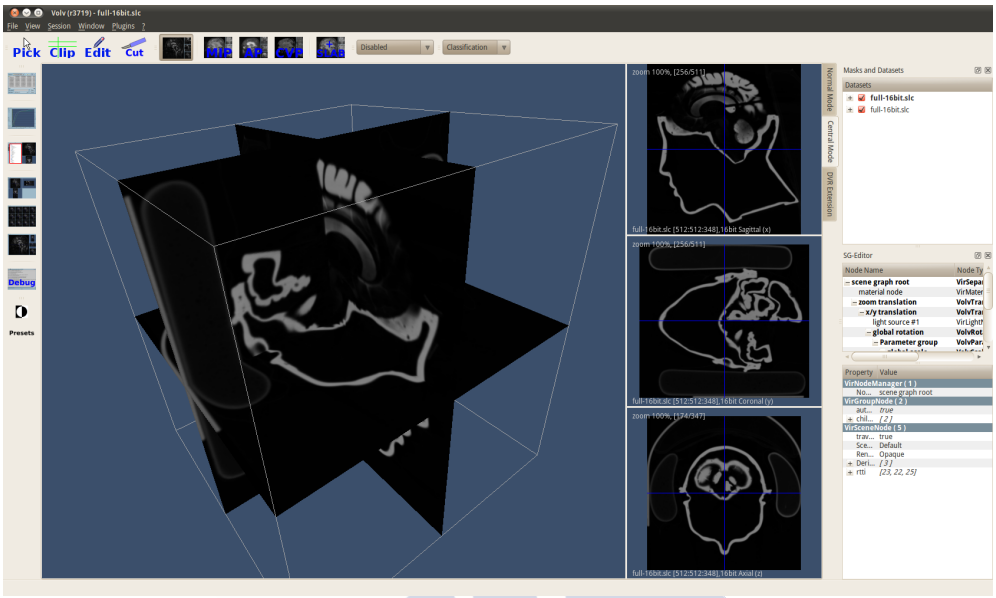


Figura 5.7: Visualización en VolV de tres láminas ortogonales del volumen *modelhead*.

5.3. Trabajos relacionados con la visualización multirresolución

En los últimos años los métodos de adquisición de datos han experimentado mejoras significativas y, como resultado, el tamaño de los conjuntos de datos obtenidos se ha visto incrementado. Dado los escasos recursos de memoria disponibles y el limitado ancho de banda de los canales de comunicación, tanto entre componentes *hardware* de una misma máquina como a través de una red de comunicaciones, la visualización de estos conjuntos de datos supone un reto. La tendencia actual indica que este problema va a seguir existiendo en el futuro próximo.

Los algoritmos de visualización multirresolución proporcionan una solución a este problema. Estos algoritmos permiten visualizar un volumen con diferentes niveles de detalle, por lo que no es necesario disponer de todos los datos para poder iniciar la visualización. En multitud de ocasiones la resolución del conjunto de datos es superior a la resolución del visor, y renderizar el volumen con un gran nivel de detalle supone un gasto innecesario de recursos. En general, solo cuando se realiza una ampliación sobre una región de interés del volumen, es interesante mostrar el máximo nivel de detalle. Los algoritmos multirresolución

funcionan de forma adaptativa, es decir, son capaces de mostrar distintas partes de un mismo volumen a diferente resolución, dependiendo de cuál sea la región de interés en cada momento —normalmente, la más cercana a la cámara—, y cargan los datos del volumen solo cuando son necesarios.

Hay dos aproximaciones generales para descomponer el volumen y mostrar diferentes niveles de detalle: descomposición jerárquica y descomposición plana. En la descomposición jerárquica es habitual utilizar un *octree*, esto es, un árbol donde cada uno de sus nodos tiene ocho nodos hijo. El *octree* se construye de abajo hacia arriba; se comienza dividiendo el volumen en pequeños subvolúmenes y a cada uno de ellos se asocia a los nodos hoja del árbol. A continuación, los subvolúmenes adyacentes se agrupan en grupos de ocho, y se muestrean a la mitad de resolución para obtener los nodos intermedios del árbol. El proceso se repite hasta que se alcanza un único nodo, el nodo raíz, que contiene una representación completa del volumen a la resolución más baja. Durante la visualización, en cada *frame* se explora el árbol y se selecciona un conjunto de nodos a renderizar en función de su distancia a la cámara y la memoria disponible. La descomposición jerárquica se aplica en multitud de soluciones de visualización de datos volumétricos [67, 69, 76, 98, 113, 135, 170, 171] y también en el contexto de visualización de nubes de puntos [68, 73, 151].

En la descomposición plana también se parte el volumen en subvolúmenes más pequeños, pero los diferentes niveles de resolución se calculan para cada uno de los subvolúmenes por separado; esto es, las particiones de más baja resolución no se combinan para dar lugar a nuevas particiones. Dado que cada partición se procesa por separado, este esquema es especialmente adecuado para su implementación en paralelo. En la literatura es posible encontrar soluciones de visualización multirresolución en la que se utiliza la descomposición plana [31, 84, 125, 154], que hemos utilizado también en este trabajo.

En lo que a estrictamente técnicas de visualización de datos volumétricos se refiere, las dos soluciones de renderizado más utilizadas son el *ray casting* y el mapeo de texturas [62]. La idea básica del *ray casting* es evaluar la integral de renderizado (véase sección 1.5) a lo largo de los rayos atravesados por la cámara [149]. Para cada píxel en la imagen final se lanza un único rayo dentro del volumen, y se toman muestras de los datos en posiciones discretas del rayo. En la figura 5.8 puede verse un esquema de esta aproximación. La implementación más común en GPU utiliza una lámina poligonal sencilla que se sitúa delante de la cámara, y un *fragment shader* se encarga de ejecutar, para cada píxel, la toma de muestras a lo largo del rayo

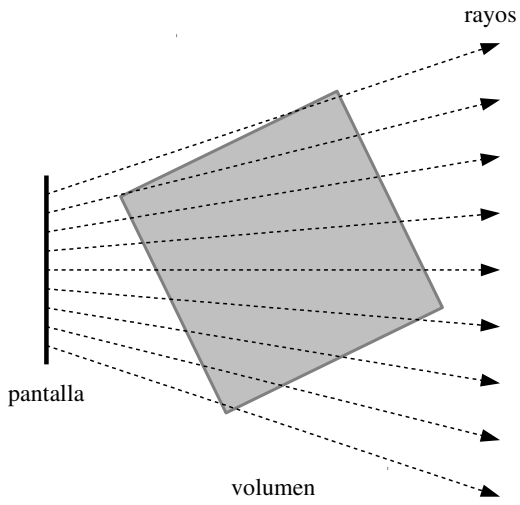


Figura 5.8: Esquema de visualización de un volumen usando *ray casting*.

y calcular el valor del píxel. Es bastante habitual encontrar esta técnica en la literatura [67, 69, 76, 84, 113, 135, 170, 171].

La técnica de mapeo de texturas [66] utiliza una aproximación diferente para calcular la integral de renderizado. Durante la visualización, se genera una serie de láminas poligonales paralelas al plano de la cámara, y se mapea la textura 3D del volumen en esas láminas poligonales. De esta forma, se aprovecha la potencia del *hardware* en la GPU para realizar operaciones de *blending* para calcular el valor final de cada píxel, en lugar de utilizar un *fragment shader*. Esta es la aproximación que utilizamos en nuestra solución (véase sección 5.4), y también está presente en otros trabajos de la literatura [35, 76, 98].

En lo que se refiere a la construcción de una representación compacta de los datos, que suele llevar asociada una reducción de la cantidad de información respecto a la imagen original, en la literatura se han implementado una amplia variedad de esquemas de codificación y decodificación. En el campo de visualización de volúmenes, estas soluciones suelen ser asimétricas, es decir, el conjunto de datos original se compacta en un proceso previo que se ejecuta una sola vez, mientras que la descompresión y la visualización son procesos que se ejecutan en tiempo real [64]. Los métodos más comunes para la compresión suelen implicar el uso de la transformada *wavelet* [76, 84], la cuantización de vectores [64, 67, 135, 154],

tensores multiescala [170, 171], o el muestreo adaptativo de los datos originales [181]. Para una revisión más detallada de soluciones en GPU para la visualización de volúmenes comprimidos, véase [147, 148, 188].

Nuestra implementación de visualización multirresolución está basada en la transformada *wavelet*. Los coeficientes de menor resolución se calculan mediante un esquema de interpolación lineal consistente con el utilizado por el *hardware* de la GPU (véase sección 1.5.2), lo que reduce el número de artefactos cuando se producen cambios de resolución así como la aparición de discontinuidades entre bloques [77]. Además de la transformada *wavelet*, la compresión se completa con un esquema de codificación similar a [84] (véase sección 5.4.1), que permite realizar un acceso rápido a cualquier vóxel del volumen durante la etapa de descompresión.

Muchas de las soluciones para la visualización de volúmenes comprimidos que se pueden encontrar en la literatura se apoyan en almacenar el volumen comprimido en un espacio diferente a la GPU (como puede ser la memoria RAM de la CPU o el disco duro) [69, 76, 113, 170]. Estas técnicas funcionan transfiriendo porciones descomprimidas del volumen a la memoria de la GPU antes de que pueda ser visualizado. Su rendimiento está limitado por la tasa de transferencia del bus PCI (por ejemplo, en el caso de PCI Express 2.0 esta tasa es de 8 GB/s). Otras soluciones utilizan la GPU para ejecutar el proceso de descompresión [67, 135, 154, 171]. Nuestra solución almacena el volumen en su forma comprimida en la memoria de la GPU e implementa el proceso de descompresión en CUDA.

5.4. Método propuesto para la visualización multirresolución de volúmenes en GPU

Nuestra solución se compone de dos fases: preprocesado y visualización. La fase de preprocesado se ejecuta en CPU y genera el volumen comprimido a partir del conjunto de datos original. La fase de visualización se ejecuta en GPU y muestra en pantalla la reconstrucción de un volumen a diferentes niveles de resolución dependiendo de la distancia a la cámara.

La figura 5.9 muestra las diferentes estructuras de datos utilizadas en esta implementación. El volumen original es dividido en *bloques unitarios* de 16^3 vóxeles. Estos bloques se agrupan en *particiones*, que pueden contener uno o más bloques. En la figura, se muestra un ejemplo donde cada partición contiene 2^3 bloques. Los vóxeles de cada bloque se agrupan en *celdas*,

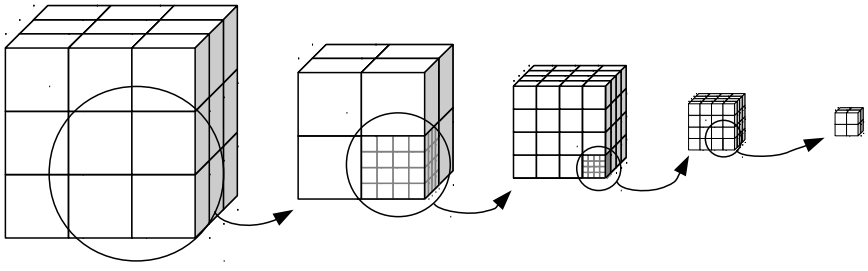


Figura 5.9: Estructuras de datos usadas en este trabajo.

de forma que cada celda contiene 4^3 vóxeles. Por último, una *porción* consistirá en un grupo de 2^3 vóxeles.

Durante la fase de preprocesado, nuestro algoritmo de compresión divide el volumen en bloques y celdas. La transformada *wavelet* se aplica a cada bloque, y el procesado de los datos durante este paso se hace porción a porción. Durante la fase de visualización se considera que el volumen está dividido en particiones, que se procesan de forma individual. A lo largo de esta sección describiremos en detalle las diferentes etapas para completar las fases de compresión y visualización.

5.4.1. Fase de preprocesado

La fase de preprocesado tiene lugar cuando el usuario selecciona un conjunto de datos volumétrico para su compresión. La figura 5.10 muestra las etapas que se ejecutan durante esta fase y los datos que se generan en cada una de ellas. En primer lugar, se aplica una transformada *wavelet* a los datos volumétricos. Después, se realiza un proceso de cuantización para reducir los valores de los coeficientes obtenidos en la transformada, de forma que aquellos que tengan un valor cercano a cero queden anulados. Por último, el proceso de codificación genera el volumen comprimido, que es almacenado en disco. Todos estos pasos que se describen a continuación se ejecutan en la CPU.

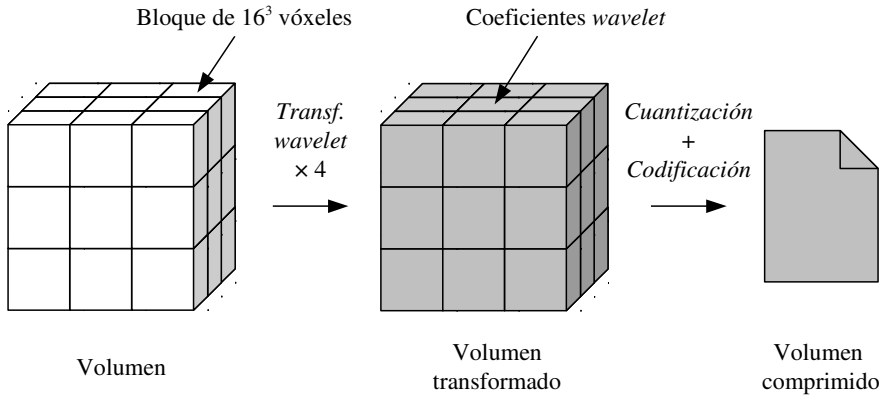


Figura 5.10: Flujo de trabajo durante el preprocesado del volumen para generar la versión comprimida.

Transformada *wavelet*

Este paso aplica la transformada *wavelet* a bloques de 16^3 vóxeles utilizando el filtro Haar. El proceso de la transformada se realiza por bandas: el bloque de datos a transformar forma la banda inicial, y tras aplicar la transformada se generan ocho nuevas bandas (o subbandas) que suman en total el mismo tamaño que la banda original.

La aplicación de una transformada *wavelet* a un conjunto de datos tridimensional normalmente ha de ser realizada en varios pasos, de forma que el proceso de filtrado se computa primero por filas, luego por columnas y, por último, por láminas (véase capítulo 1). Sin embargo, en el caso del filtrado Haar es posible reducir este paso a uno solo, y computar la transformación directamente en las tres dimensiones del espacio. Dada una banda de vóxeles, el proceso de generación de las subbandas como resultado de la transformada comienza dividiendo la banda en porciones de $2^3 = 8$ vóxeles (el mismo tamaño que el filtro). Sobre cada

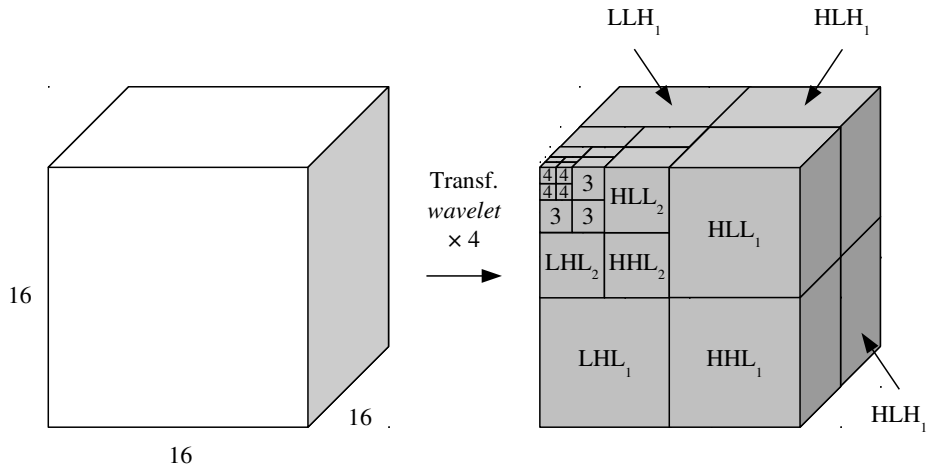


Figura 5.11: Aplicación de una transformada *wavelet* de 4 niveles a un bloque de 16^3 vóxeles.

porción se aplican las operaciones derivadas del filtro Haar, y que tienen como resultado la generación de 8 coeficientes *wavelet*, siguiendo las operaciones indicadas a continuación:

$$\begin{aligned}
c_{lll} &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8, \\
c_{llh} &= c_1 + c_2 + c_3 + c_4 - c_5 - c_6 - c_7 - c_8, \\
c_{lhl} &= c_1 + c_2 - c_3 - c_4 + c_5 + c_6 - c_7 - c_8, \\
c_{lhh} &= c_1 + c_2 - c_3 - c_4 - c_5 - c_6 + c_7 + c_8, \\
c_{hll} &= c_1 - c_2 + c_3 - c_4 + c_5 - c_6 + c_7 - c_8, \\
c_{hlh} &= c_1 - c_2 + c_3 - c_4 - c_5 + c_6 - c_7 + c_8, \\
c_{hhl} &= c_1 - c_2 - c_3 + c_4 + c_5 - c_6 - c_7 + c_8, \\
c_{hhh} &= c_1 - c_2 - c_3 + c_4 - c_5 + c_6 + c_7 - c_8.
\end{aligned} \tag{5.1}$$

El coeficiente c_{lll} se calcula a partir de la suma de todos los vóxeles de la porción, y se denomina coeficiente medio; el resto de coeficientes se denominan coeficientes de detalle. Los coeficientes de cada subbanda se agrupan para generar la banda transformada; esto facilita tanto la aplicación de una nueva transformada sobre una de las subbandas, como el proceso de compresión.

La figura 5.11 muestra cómo se generan los coeficientes de una transformada *wavelet* aplicada a un bloque de 16^3 vóxeles. El bloque inicial a la izquierda de la figura forma una banda única, y tras aplicar la transformada una vez se generan 8 subbandas de 4^3 coeficientes cada una. Estas subbandas se etiquetan con nombres que van desde LLL hasta HHH, donde la subbanda LLL contiene los coeficientes medios y las demás subbandas contienen los coeficientes de detalle. En la figura la transformada se aplica hasta cuatro veces de forma recursiva, siempre sobre la subbanda LLL, para generar nuevos niveles de transformada. Calcular varios niveles de transformada nos permite maximizar el número de coeficientes de detalle en el volumen transformado, que tienen valores más pequeños, de modo que una importante proporción de ellos puede ser eliminada sin afectar significativamente a la reconstrucción del volumen. Además, dado que las subbandas LLL almacenan los valores medios, podemos asociar cada nivel de transformada con un nivel de resolución diferente, lo que nos permite soportar un esquema de visualización con multirresolución, como se explicará más adelante.

Normalmente, cada uno de los coeficientes calculados en la transformada Haar es normalizado, es decir, su valor se divide entre el tamaño del filtro (8 en el caso tridimensional). Nuestra solución podría aplicar este esquema de normalización, con el coste de asumir pérdidas de precisión asociadas a trabajar con coeficientes en punto flotante. Sin embargo, para evitar cualquier posible pérdida de información durante esta fase, no normalizamos los coeficientes. Esto significa que la magnitud de los coeficientes (especialmente los de la banda de baja frecuencia LLL) crece cada vez que se aplica la transformada, pero los valores de los coeficientes siguen siendo enteros durante todo el proceso y el volumen original puede ser reconstruido tal cual a partir del volumen transformado.

Cuantización

La cuantización es una técnica de compresión con pérdidas que reduce el rango de valores del conjunto de datos comprimido [75]. La cuantización de una señal es un proceso por lo general irreversible y que da lugar a pérdidas de información.

Hay una amplia variedad de técnicas de cuantización [52], aunque básicamente se pueden agrupar en dos grandes grupos: cuantización uniforme, también llamada de “memoria cero” o “sin memoria”, donde cada muestra de la señal se cuantiza de forma independiente, y no uniforme, donde la cuantización de cada muestra depende de la función de distribución de la señal.

En nuestra implementación hemos escogido una cuantización uniforme consistente en eliminar los bits menos significativos de los coeficientes obtenidos en la transformada *wavelet* previa. Esta cuantización reduce la magnitud de los coeficientes y anula aquellos con valores próximos a cero. Esta forma sencilla de redondeo nos ha permitido acoplar las fases de cuantización y codificación y ejecutarlas como si fuesen una sola.

Codificación

Esta etapa convierte los datos volumétricos obtenidos de la transformada *wavelet* y de la cuantización en la forma final comprimida del volumen. La implementación de esta etapa está basada en el esquema de codificación utilizado en [84].

La figura 5.12 muestra en la parte superior un bloque unitario de 16^3 vóxeles dividido en celdas de 4^3 vóxeles (véase también figura 5.9). Cada celda está identificada unívocamente por un índice. En la parte inferior se muestran las principales estructuras que se almacenan en el interior del volumen comprimido. Para cada bloque el vector de tablas de etiquetas de celda almacena una tabla de etiquetas. Cada tabla contiene una etiqueta de 2 bytes para cada celda en el bloque. El byte más significativo almacena la longitud en bytes de los coeficientes de la celda (o cero en caso de tratarse de una celda nula, esto es, una celda que solo contiene ceros); el byte menos significativo almacena el índice del mapa de significado asociado a dicha celda. El vector de mapas de significado contiene un mapa de bits para cada celda no nula del volumen. Este mapa de bits identifica qué coeficientes de las celdas son nulos y cuáles no. Por último, cuatro vectores (que no se muestran en la figura) almacenan todos los coeficientes no nulos del volumen transformado. Soportamos coeficientes de entre uno y cuatro bytes, y cada coeficiente se almacena en uno u otro vector dependiendo de su longitud en bytes.

El proceso de codificación se realiza siguiendo los siguientes pasos:

1. El volumen se divide en bloques del mismo tamaño. A su vez, cada bloque se divide en celdas, a las que se les asigna un índice único de acuerdo a su posición dentro del bloque.

2. Para cada bloque se recorre el contenido de todas las celdas y se generan las etiquetas de celda. Como ya hemos comentado, la etiqueta se compone de dos bytes. Si la celda correspondiente contiene solo ceros, el primer byte será un cero. En caso contrario, el byte almacenará la longitud en bytes del coeficiente más alto almacenado en la celda.

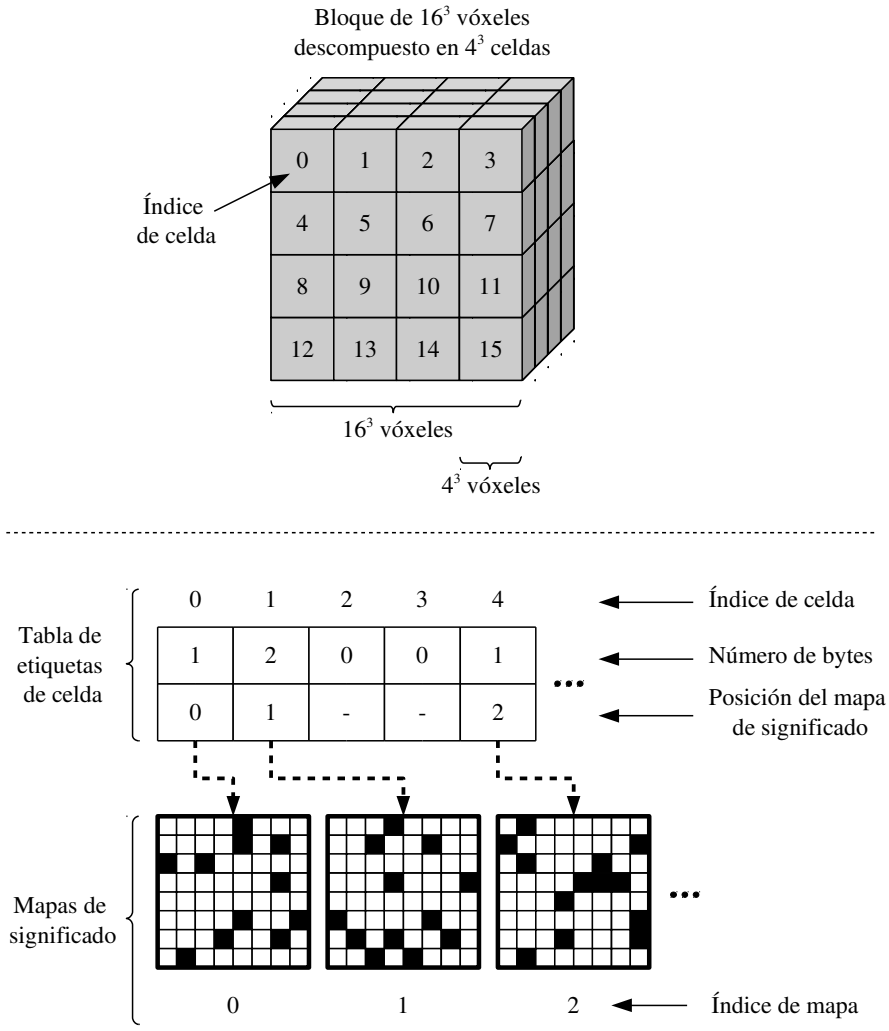


Figura 5.12: Codificación de los datos volumétricos. Para cada bloque unitario, se genera una tabla de etiquetas, con tantas etiquetas como celdas en el bloque. Las etiquetas correspondientes a celdas no nulas almacenan el índice del mapa de significado asociado a la celda.

3. Se recorre el contenido de la celda una vez más para generar su mapa de significado. Para los coeficientes cuyo valor sea cero se marcará un cero en el mapa. Para los coeficientes cuyo valor sea distinto de cero se marcará un uno, y el coeficiente se almacenará en uno de los cuatro vectores de coeficientes no nulos atendiendo a la longitud de bytes establecida en el paso anterior. Si la celda solo contiene ceros, no tendrá mapa de significado asociado.

4. La etiqueta de la celda se actualiza con la posición del mapa en el vector de mapas de significado y se almacena en la tabla de etiquetas en la posición indicada por el índice de celda.

Nuestra solución de codificación incrementa la flexibilidad de la implementación presentada en [84], que utilizaba un único byte para cada etiqueta de celda y, por ello, está limitada a bloques de 4^3 celdas. Al utilizar dos bytes para almacenar cada etiqueta de celda podemos soportar bloques más grandes con un mayor número de celdas y, por tanto, incrementar el número de posibles niveles de resolución (ya que el número máximo de transformadas *wavelet* que se pueden aplicar de forma recursiva a cada bloque está limitado por su propio tamaño). Además, nuestra solución soporta la codificación de coeficientes de hasta cuatro bytes de longitud, lo que significa que podemos almacenar coeficientes con valores de gran magnitud, e incluso coeficientes no normalizados sin cuantizar (aunque esto resultaría en un ratio de compresión bajo).

Además de los datos comprimidos con las estructuras comentadas, nuestra solución genera también un mapa de bits con el mismo tamaño que el número de bloques en que se ha dividido el volumen. Este mapa de bits almacena para cada bloque un cero o un uno dependiendo de si el bloque es nulo (es decir, contiene solo ceros) o no. Los bloques nulos se consideran vacíos, por lo que no son procesados durante la fase de visualización.

5.4.2. Visualización

Durante la fase de visualización se reconstruye en la GPU el volumen comprimido y se renderiza en pantalla. La figura 5.13 muestra las diferentes etapas que componen esta fase. El volumen se reconstruye de forma iterativa: cada iteración comienza seleccionando una partición del volumen para su reconstrucción a un determinado nivel de resolución. Esta reconstrucción implica realizar las etapas de decodificación y transformada inversa, que están

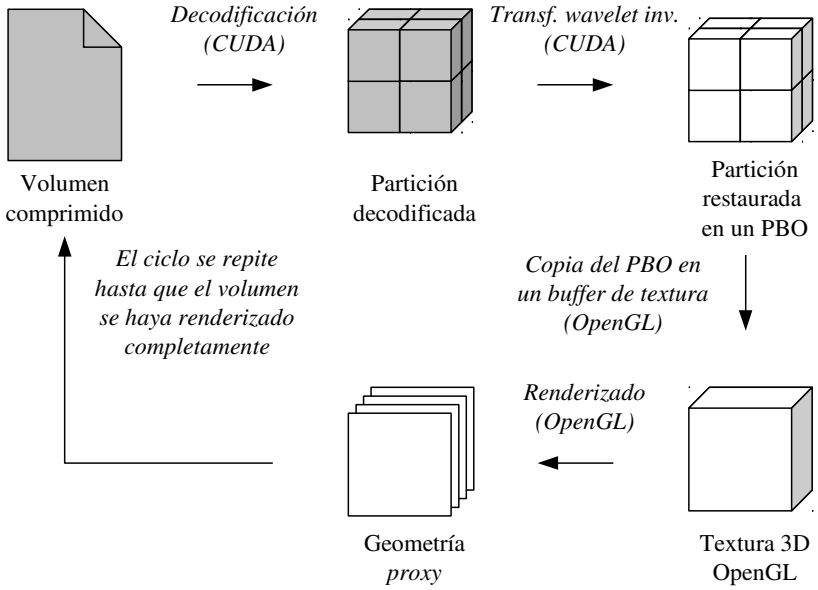


Figura 5.13: Flujo de trabajo durante la visualización del volumen.

implementadas como *kernels* CUDA y, por tanto, son ejecutadas en la GPU. Los datos restaurados de la partición se almacenan en un OpenGL Pixel Buffer Object (PBO), un espacio de memoria de la GPU que puede ser destinado a almacenar datos de textura, y a continuación se copian en un *buffer* de texturas para que puedan ser mapeados en las láminas poligonales paralelas al plano de la cámara que componen la geometría *proxy*. Estas operaciones, incluyendo el renderizado final, se han implementado mediante el API de OpenGL. El proceso continúa con otra partición hasta que la totalidad del volumen es reconstruida y renderizada.

A lo largo de esta sección describiremos cómo la CPU dirige el flujo del trabajo del algoritmo, seleccionando una partición del volumen para su procesado y renderizado, y describiremos la implementación en GPU de cada una de las etapas que participan en este proceso.

División en particiones

Como ya se ha comentado, el proceso de visualización se realiza sobre las diferentes particiones del volumen, reconstruyendo una partición de cada vez. Antes de iniciar el proceso, se recorren los bloques de cada una de las particiones para determinar si hay particiones nulas

(es decir, particiones en las que todos los bloques son nulos). Las particiones nulas se corresponden con espacios vacíos en el volumen, por lo que no necesitan ningún procesamiento durante esta etapa. Además, en las etapas de decodificación y de la transformada inversa los bloques nulos que se encuentren en medio de particiones no nulas tampoco son procesados.

La CPU dirige el proceso de visualización: para cada *frame*, establece en primer lugar en qué orden se van a reconstruir y renderizar las particiones de acuerdo a su posición respecto a la cámara. Las particiones se reconstruyen de atrás hacia adelante, es decir, empezando por las más alejadas de la cámara y terminando con las más cercanas, con objeto de garantizar una correcta composición de las particiones. Además, para cada partición se determina un nivel de resolución, de forma que las particiones más cercanas a la cámara se reconstruyen al nivel más alto posible, y las más alejadas y que, por tanto, menos contribuyen al resultado final se reconstruyen al nivel más bajo. De esta forma, se optimiza el proceso de reconstrucción sin afectar significativamente a la calidad de visualización.

El proceso de decodificación y la transformada *wavelet* inversa se ejecutan en dos *kernels* CUDA llamados desde la CPU. Los datos restaurados se almacenan en un PBO, ya que puede ser accedido tanto por funciones CUDA como por funciones OpenGL. Para completar la visualización, la CPU realiza una llamada OpenGL para copiar los datos del PBO en un *buffer* de texturas. Por último, la CPU ordena la construcción de una geometría *proxy* realizando múltiples llamadas OpenGL. Esta geometría contiene las láminas poligonales en las que se mapea la textura generada a partir de los datos restaurados de la partición.

Decodificación

El proceso de decodificación es ejecutado por un *kernel* en la GPU. Este *kernel* lee los datos comprimidos de la porción seleccionada, que están almacenados en memoria global, para su reconstrucción y escribe los datos decodificados en un espacio de memoria global que luego será utilizado por la etapa de la transformada *wavelet* inversa.

El *kernel* de decodificación asigna cada bloque de hilos a un bloque de datos no nulo dentro de la partición, y cada hilo procesa una de las celdas del bloque. En primer lugar, cada hilo identifica si su celda es o no nula en función de la etiqueta de celda almacenada en la tabla de etiquetas de celda del bloque correspondiente. Si la celda es no nula, se inicia la reconstrucción de los datos: el hilo explora el mapa de significado asociado a la celda, y carga los vóxeles no nulos de los vectores de coeficientes del volumen comprimido y los escribe en el espacio de memoria destinado a los contenidos de la celda. Si, por el contrario, la celda es

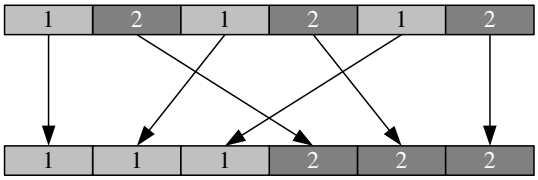


Figura 5.14: Almacenamiento lineal en memoria global de una partición que contiene dos bloques diferentes.

nula, el hilo solo escribirá ceros para invalidar el contenido que estuviera almacenado en el *buffer* como resultado de operaciones previas.

Durante esta etapa se realiza también el proceso inverso a la cuantización, esto es, se añaden los bits menos significativos que han sido eliminados durante la compresión del volumen. Dado que no es posible recuperar los valores originales de los bits eliminados, se añaden ceros en su lugar. Esto significa que el volumen reconstruido diferirá (en mayor o menor medida dependiendo del nivel de cuantización aplicado) del volumen original. Sin embargo, como mostraremos en la sección 5.5, ajustando el nivel de cuantización es posible conseguir buenos niveles de compresión sin afectar severamente la calidad de visualización.

Para incrementar la localidad espacial de los accesos a memoria, los datos decodificados de un mismo bloque se almacenan contiguamente. La figura 5.14 ilustra este proceso para una partición compuesta de dos bloques. En la parte superior de la figura se muestra cómo se intercalan los datos de los diferentes bloques en memoria si se almacenan atendiendo únicamente a su posición dentro de la partición. Nuestro algoritmo reordena los datos de los bloques para incrementar la localidad espacial con el objetivo de mejorar el rendimiento de los accesos a memoria.

Transformada wavelet inversa

Una vez que el *kernel* de decodificación ha recuperado los coeficientes de la partición procesada en memoria global, otro *kernel* ejecuta la transformada inversa en GPU para restaurar el contenido de la partición. Este *kernel* asigna un bloque de hilos a cada bloque de datos no nulo de la partición, y cada hilo procesa una porción del tamaño 2^3 (véase figura 5.9 con las estructuras de datos utilizadas en este capítulo).

La transformada inversa es también un proceso recursivo: se comienza por el nivel más bajo, y en cada iteración se transforma un nuevo nivel. El número de hilos ociosos varía

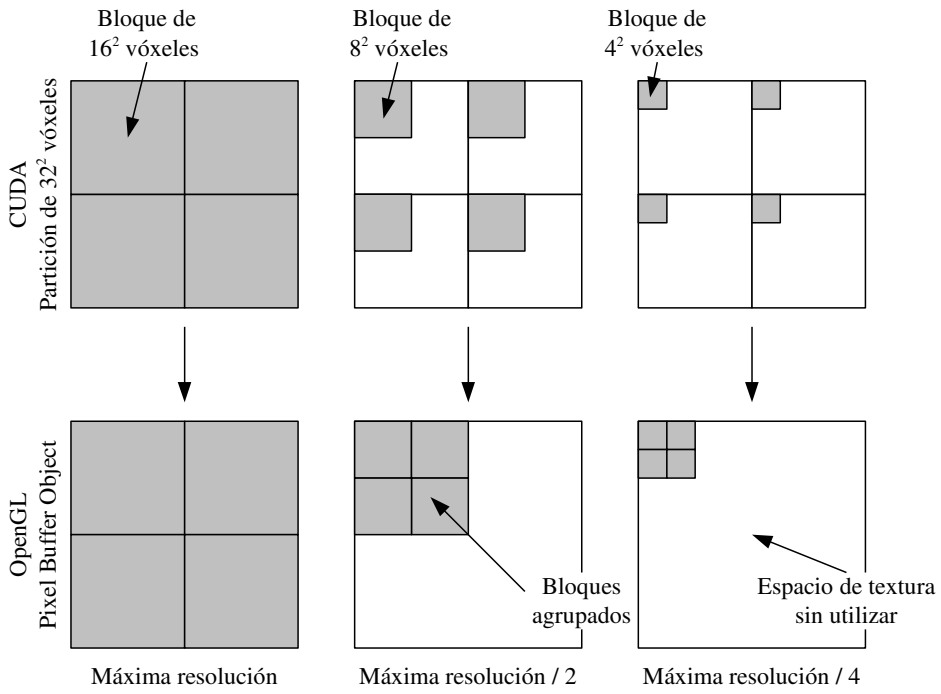


Figura 5.15: Almacenamiento de datos desde memoria compartida en el PBO para diferentes niveles de resolución.

a lo largo de este proceso; al incrementar de nivel aumenta el número de coeficientes que participan en la transformada, y por tanto el número de hilos. Los resultados de transformar un nivel se almacenan en la memoria compartida, de forma que estén disponibles para calcular el siguiente nivel de resolución. Este proceso se repite hasta que se alcanza el nivel de resolución establecido para la partición, momento en que los valores restaurados se normalizan (con el objetivo de que su magnitud sea la misma que la de los vóxeles del volumen original) y se copian desde la memoria compartida al PBO. En el caso de procesar el nivel más alto de resolución, los coeficientes se almacenan directamente en el PBO, reduciendo así el número de transacciones en memoria.

Cuando se almacenan los bloques de datos restaurados en el PBO, sus posiciones se determinan a partir del identificador del bloque de hilos y del nivel de resolución de la partición. Para niveles de resolución bajos, los datos generados por cada bloque de hilos se agrupan con el objetivo de evitar que los datos queden dispersos en el PBO. La figura 5.15 muestra el caso

bidimensional de una partición de 32^2 vóxeles, pero puede extenderse al caso tridimensional. Cada partición se compone de cuatro bloques. Cuando se trabaja a la máxima resolución, cada bloque tiene su tamaño máximo (en este caso, 16^2 vóxeles). En este caso, el almacenamiento de los bloques en el PBO es trivial. Cuando se disminuye la resolución a la mitad, como resultado de la transformada inversa el tamaño de cada bloque pasa a ser la mitad en cada dimensión, es decir, cada bloque pasa a tener 8^2 vóxeles. El tamaño total de los bloques es inferior al PBO, por lo que sus vóxeles deben agruparse para evitar que queden dispersos en el PBO. Solo de esta forma el contenido del PBO será válido para generar la textura en la siguiente etapa. Lo mismo ocurre cuando la resolución se reduce a la cuarta parte de su nivel máximo, como se muestra en la figura.

Copia de datos

Una vez se han restaurado los valores de los vóxeles de la partición al nivel de resolución seleccionado, se procede a copiar sus valores a un *buffer* de textura para que pueda iniciarse el renderizado.

El proceso de copia se realiza con dos llamadas al API de OpenGL. En primer lugar, se activa el *buffer* para su lectura con una llamada a `glBindBuffer`. A continuación, con una llamada a `glTexSubImage3D` se realiza la copia. Esta función, a diferencia de `glTexImage3D`, nos permite reutilizar el mismo *buffer* de textura. Tanto el PBO como el *buffer* tienen el mismo tamaño, es decir, el tamaño necesario para almacenar el contenido de la partición sin importar el nivel de resolución, y el contenido del *buffer* se sobrescribe siempre en su totalidad. Como los datos del PBO ya se almacenan de forma agrupada, no es necesario realizar ninguna operación adicional.

Dependiendo del nivel de resolución, la textura puede no llenar completamente el espacio disponible dentro del *buffer* de texturas: en el nivel de resolución más alto se utiliza completamente el espacio habilitado, pero en los niveles más bajos solo se requiere una porción de dicho espacio. Esto implica que, durante la construcción de la geometría *proxy*, las coordenadas de textura asignadas a cada vértice deben ajustarse al tamaño real de la textura (no al tamaño habilitado en el *buffer*) de acuerdo al nivel de resolución de la partición.

Renderizado

La etapa de renderizado se inicia una vez que se ha generado la textura correspondiente a la partición actual. Durante esta etapa se construye la geometría *proxy*, compuesta por una serie

de láminas poligonales que se renderizan perpendiculares a la línea de visión de la cámara. Dado que la posición de los vértices y la configuración de las láminas depende de la posición de la cámara, es necesario realizar este proceso cada vez que se dibuje un nuevo *frame*.

Esta etapa se desarrolla en varios pasos que detallaremos a continuación:

1. Se determina la posición en la escena del volumen que encierra a las láminas. Este volumen consiste en un prisma situado en el centro de la escena cuyos vértices tienen coordenadas fijas. Aunque el prisma no es renderizado, se utiliza para delimitar la extensión de las láminas. Denominamos a esta estructura como *volumen proxy*.

2. Se determina la posición en la escena del volumen que encierra a la partición. A partir de la posición de la partición en el volumen de datos y de su tamaño, y de la posición y el tamaño del volumen *proxy*, se identifican las coordenadas correspondientes de los vértices de este segundo prisma, cuya función es determinar la extensión de las láminas que pertenecen a la partición. Denominamos a esta estructura como *partición proxy*.

3. Se determina la distancia entre las láminas poligonales que compondrán la geometría *proxy* en función del número de láminas (que es un parámetro configurable de nuestra solución) y el tamaño del volumen *proxy*.

4. Para cada lámina se determinan sus puntos de intersección con la partición *proxy*. Estos puntos de intersección definen los vértices de la lámina. Las láminas que están fuera de la partición actual no tendrán ningún punto de intersección, por lo que no son dibujadas.

5. Se calculan las coordenadas del vértice central de la lámina, es decir, aquel que se encuentra en el centro de todos los vértices de la lámina. Se genera una lista de los vértices ordenada alrededor del vértice central y se eliminan los vértices repetidos.

6. Se dibuja la lámina como un "abanico de triángulos" alrededor del vértice central. Utilizando el vértice central como vértice común, los demás vértices se agrupan de dos en dos para definir los tres vértices de cada triángulo. A partir de la posición de cada vértice dentro de la partición *proxy* y del nivel de resolución, se determina la coordenada de textura asociada. El proceso continúa con la siguiente lámina.

La figura 5.16 muestra un ejemplo de partición *proxy* (representada mediante un prisma de aristas discontinuas) que encierra una serie de láminas poligonales situadas en paralelo al plano de visión de la cámara. Se puede apreciar que las láminas están limitadas al tamaño de

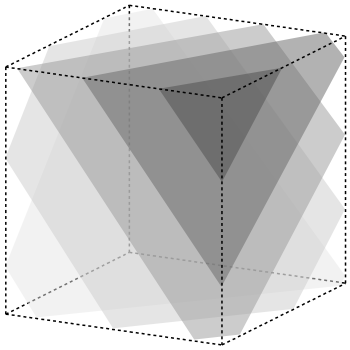


Figura 5.16: Ejemplo de construcción de la geometría *proxy* de una partición.

la partición, y que todos los vértices de las láminas se sitúan en las aristas del prisma. Aunque hemos implementado el cálculo de las coordenadas de los vértices en CPU, construimos la geometría mediante llamadas al API de OpenGL, por lo que el renderizado final se realiza íntegramente en GPU.

Nuestra implementación extiende el módulo de renderizado directo de volúmenes incluido en VolV y añade soporte para visualizar volúmenes comprimidos. Hemos implementado el código necesario para visualizar las diferentes particiones en las que se divide el volumen. También hemos incluido el código para descomprimir en la GPU cada una de las particiones, con las funcionalidades descritas hasta ahora. Gracias a VolV hemos podido centrar nuestros esfuerzos en el desarrollo de la compresión y la visualización, mientras que para la gestión de datos y de la interfaz utilizamos la API que ofrece VolV.

5.5. Resultados

En esta sección presentamos las medidas de rendimiento y de calidad obtenidas con el algoritmo de visualización de volúmenes comprimidos en GPU presentado en este capítulo.

5.5.1. Metodología

Realizamos nuestras pruebas en una NVIDIA GeForce GTX 580, cuyas características aparecen reflejadas en la sección 1.7. El espacio de memoria fue configurado a 16 kB por

Volumen	Dimensiones	Bytes/vóxel	Tamaño
`brainweb-2`	$256 \times 256 \times 181$	2	23 MB
`modelhead`	$512 \times 512 \times 348$	2	174 MB
`realhead`	$160 \times 512 \times 512$	2	80 MB
`knee`	$281 \times 389 \times 104$	2	22 MB
`vessels-1`	$160 \times 161 \times 226$	2	11 MB

Tabla 5.1: Conjuntos de datos utilizados para la visualización de volúmenes comprimidos.

multiprocesador, con 32 kB de caché L1. La GPU estaba montada en una máquina con un procesador Intel Core 2 Quad Q9450 de cuatro *cores* a 2,6 GHz y 6 GB de RAM.

Para realizar una evaluación de calidad de nuestro algoritmo de decodificación tomamos, para diferentes ratios de compresión, medidas del error cuadrático medio (en inglés MSE, *mean squared error*) y de la relación señal a ruido de pico (en inglés PSNR, *peak signal-to-noise ratio*), definidas en la sección 1.6. Para la evaluación de rendimiento medimos el tiempo de ejecución de los *kernels* de decodificación y transformada *wavelet* inversa implementados en CUDA. También medimos el rendimiento de la solución completa de visualización y la tasa de refresco alcanzada en *frames* por segundo (FPS).

5.5.2. Conjuntos de datos

En nuestros experimentos para medir el rendimiento y la calidad de nuestro algoritmo de visualización de volúmenes utilizamos los conjuntos de datos `brainweb-2`, una imagen MRI del cerebro, `modelhead`, una imagen CT de una cabeza hecha de plástico, `realhead`, una imagen MRI de una cabeza humana. También utilizamos el resultado de segmentar el volumen `knee` utilizando el algoritmo de segmentación AOS desarrollado en el capítulo 3, así como el resultado de segmentar el volumen `vessels-1` con el algoritmo de segmentación rápida de dos ciclos desarrollado en el capítulo 4. La tabla 5.1 muestra las dimensiones y el tamaño de los conjuntos de datos utilizados (pueden consultarse más detalles en la sección 1.7).

5.5.3. Medidas de calidad y requisitos de almacenamiento

Medimos la calidad de nuestra implementación de decodificación en los conjuntos de datos `brainweb-2` y `modelhead`. Puede considerarse a estos dos conjuntos de datos como instancias representativas de volúmenes obtenidos a partir de materiales orgánicos e inorgánicos, respectivamente.

Número de bits	Tamaño comprimido	Ratio de compresión	MSE	PSNR
		`brainweb-2`		
0	25,52 MB	1 : 0,89	0,00	∞ dB
1	25,10 MB	1 : 0,90	0,45	99,77 dB
2	18,92 MB	1 : 1,20	0,72	97,75 dB
3	16,81 MB	1 : 1,35	2,28	92,76 dB
4	14,85 MB	1 : 1,52	8,84	86,86 dB
5	13,25 MB	1 : 1,71	35,34	80,85 dB
6	11,41 MB	1 : 1,98	137,92	74,93 dB
7	8,63 MB	1 : 2,62	509,06	69,26 dB
8	5,57 MB	1 : 4,06	1614,11	64,25 dB
9	2,89 MB	1 : 7,83	3488,04	60,90 dB
10	1,81 MB	1 : 12,50	6628,75	58,12 dB
11	1,15 MB	1 : 19,67	12135,79	55,49 dB
		`modelhead`		
0	111,82 MB	1 : 1,56	0,00	∞ dB
1	89,08 MB	1 : 1,95	0,45	99,79 dB
2	64,35 MB	1 : 2,70	0,63	98,31 dB
3	45,87 MB	1 : 3,79	1,34	95,05 dB
4	33,02 MB	1 : 5,27	3,25	91,21 dB
5	23,95 MB	1 : 7,27	8,58	87,00 dB
6	17,74 MB	1 : 9,81	23,81	82,56 dB
7	13,09 MB	1 : 13,29	66,71	78,09 dB
8	9,57 MB	1 : 18,18	187,64	73,60 dB
9	7,08 MB	1 : 24,58	504,59	69,30 dB
10	5,29 MB	1 : 32,89	1320,25	65,12 dB
11	3,82 MB	1 : 45,55	3117,75	61,39 dB

Tabla 5.2: Medidas de calidad para la compresión de los conjuntos de datos `brainweb-2` y `modelhead` utilizando diferentes niveles de cuantización.

La tabla 5.2 muestra los valores de ratio de compresión, error cuadrático medio y relación señal a ruido de pico para los dos conjuntos de datos mencionados con diferentes niveles de cuantización. Cada nivel de cuantización equivale a truncar un determinado número de bits menos significativos, indicados en la primera columna de la tabla. Este truncamiento se aplica sobre los coeficientes obtenidos durante la etapa de la transformada *wavelet* (véase sección 5.4.1). En general, un valor de PSNR por encima de 60 dB se considera de calidad aceptable. En nuestros experimentos no hemos notado que el nivel de cuantización tenga una influencia significativa sobre el rendimiento.

La sección 5.4.1 detalla las diferentes estructuras de datos utilizadas para almacenar el volumen comprimido, que aparecen también en la figura 5.12. En particular:

– Las etiquetas de celda se almacenan en vectores de enteros de 2 bytes.

– Para cada celda no nula se genera un mapa de significado. Los índices de los mapas de significado, los propios mapas y las posiciones de los coeficientes asociados a los mapas se almacenan en vectores de enteros de 8 bytes.

– Cuatro vectores de bytes se utilizan para almacenar los coeficientes no nulos.

– Por último, un vector de punteros de 8 bytes almacena los índices de los bloques no nulos. Durante la visualización, los bloques nulos son ignorados.

Así, para el conjunto de datos `modelhead`, cuyo tamaño original es 174 MB, considerando un tamaño de partición de 128^3, bloques de 16^3 coeficientes y celdas de tamaño 4^3, con un nivel de cuantización de 8 bits, el tamaño requerido para almacenar el volumen comprimido es de 9,6 MB. Se puede reducir este tamaño si se selecciona un nivel de cuantización más alto, pero será a costa de perder calidad en la imagen visualizada.

5.5.4. Medidas de rendimiento

Para evaluar el rendimiento de esta solución nos centramos en la implementación en GPU de las diferentes etapas que componen el algoritmo de visualización. Medimos al rendimiento y la aceleración de los *kernels* de decodificación y transformada *wavelet* inversa en comparación con sus respectivas implementaciones en CPU. También medimos el rendimiento de la solución de visualización completa y tomamos medidas de los tiempos de ejecución de cada etapa y de la tasa de *frames* por segundo (FPS) conseguida para múltiples configuraciones.

		64^3	128^3	256^3	512^3
Decodificación	GPU	0,000076	0,000296	0,002153	0,013647
	CPU	0,003173	0,024237	0,207878	1,626087
	Aceleración	41,6x	81,9x	96,5x	119,2x
Transformada inversa	GPU	0,000033	0,000204	0,001459	0,011550
	CPU	0,009205	0,069018	0,557935	4,434827
	Aceleración	280,3x	338,9x	382,5x	384,0x

Tabla 5.3: Tiempos de ejecución en segundos y medidas de aceleración de los *kernels* de decodificación y transformada inversa ejecutados en una tarjeta GTX 580 respecto a una implementación en CPU para volúmenes de diferentes tamaños construidos a partir del conjuntos de datos `realhead` y considerando un nivel de cuantización 0 (sin compresión).

Figura 5.17: Visualización de los conjuntos de datos utilizados. De izquierda a derecha: `brainweb-2`, `modelhead` y `realhead`.

En primer lugar, presentamos las medidas de aceleración conseguidas por nuestros *kernels* de decodificación y transformada inversa. La tabla 5.3 muestra los resultados obtenidos para volúmenes de diferente tamaño extraídos del conjunto de datos `realhead` y comprimidos sin ninguna pérdida. Para ambos *kernels* el valor de aceleración se incrementa con el tamaño del volumen, ya que las capacidades computacionales de la GPU se aprovechan mejor cuando los datos a procesar son más numerosos y más *cores* procesan información simultáneamente [93].

También evaluamos el rendimiento del proceso completo de visualización en GPU. La figura 5.17 muestra un ejemplo de visualización de cada uno de los conjuntos de datos utilizando nuestra solución de visualización implementada en VolV. La tabla 5.4 muestra el rendimiento de cada etapa de nuestra implementación cuando se utiliza un nivel de cuantización de 8 bits para generar los volúmenes comprimidos y se ejecuta la visualización con diferentes

Dim. part.	Núm. part.	Decod. (CUDA)	T. inv. (CUDA)	Copia (OpenGL)	Render. (OpenGL)	Tiempo por part.	Tiempo por *frame*	FPS
				`brainweb-2`				
64^3	48	0,000058	0,000024	0,000171	0,000985	0,001	0,059	12
128^3	8	0,000159	0,000112	0,000345	0,001686	0,002	0,018	43
256^3	1	0,001037	0,000919	0,001266	0,003657	0,007	0,007	119
512^3	1	0,001453	0,001144	0,007114	0,003618	0,013	0,013	52
				`modelhead`				
64^3	384	0,000047	0,000016	0,000172	0,000592	0,001	0,317	2
128^3	48	0,000141	0,000099	0,000345	0,000993	0,002	0,076	9
256^3	8	0,000694	0,000628	0,001259	0,001721	0,004	0,034	22
512^3	1	0,005616	0,004877	0,007337	0,003519	0,021	0,021	35
				`realhead`				
64^3	192	0,000049	0,000017	0,000171	0,000587	0,001	0,158	4
128^3	32	0,000116	0,000080	0,000345	0,001033	0,002	0,050	15
256^3	4	0,000825	0,000664	0,001266	0,001949	0,005	0,019	41
512^3	1	0,003216	0,002243	0,007249	0,003573	0,016	0,016	44

Tabla 5.4: Tiempos de ejecución en segundos y FPS en una tarjeta GTX 580 para cada una de las etapas de visualización de conjuntos de datos comprimidos con una cuantización de 8 bits.

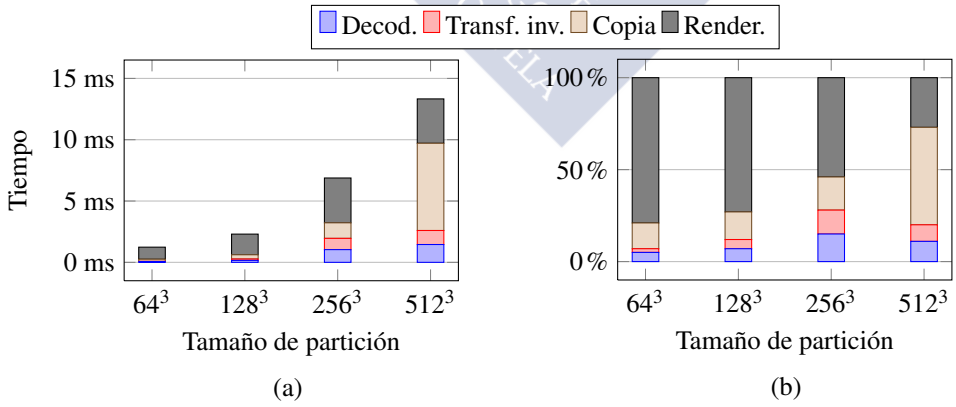


Figura 5.18: Tiempos de ejecución en una tarjeta GTX 580 de cada una de las etapas necesarias para procesar una partición durante la visualización del conjunto de datos `brainweb-2`, en (a) valores absolutos y (b) porcentaje sobre el total.

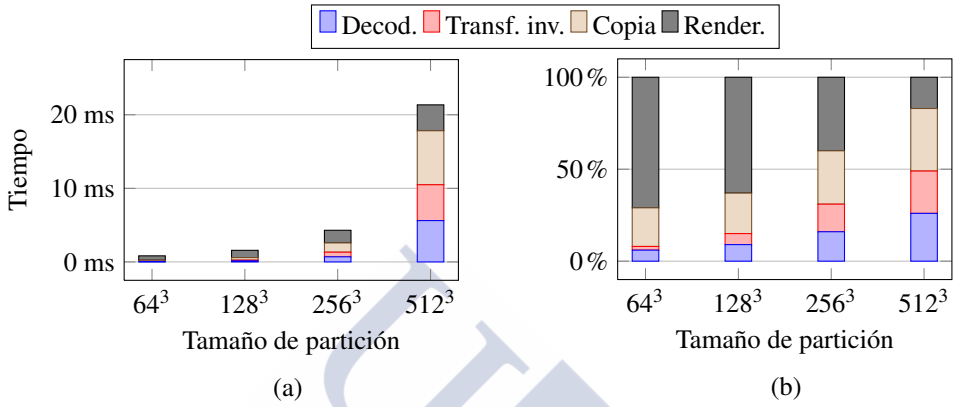


Figura 5.19: Tiempos de ejecución en una tarjeta GTX 580 de cada una de las etapas necesarias para procesar una partición durante la visualización del conjunto de datos `modelhead`, en (a) valores absolutos y (b) porcentaje sobre el total.

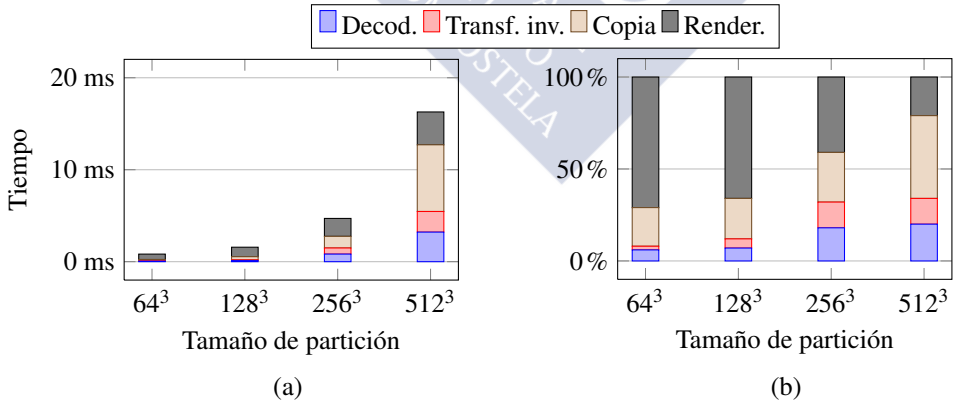


Figura 5.20: Tiempos de ejecución en una tarjeta GTX 580 de cada una de las etapas necesarias para procesar una partición durante la visualización del conjunto de datos `realhead`, en (a) valores absolutos y (b) porcentaje sobre el total.

tamaños de partición. Las tres últimas columnas de la tabla contienen el tiempo para procesar una partición, el tiempo para procesar todas las particiones en cada *frame*, y la tasa de *frames* por segundo medida.

Se observa que el tiempo necesario para ejecutar cada etapa crece con el tamaño de partición. En prácticamente todos los casos, el tiempo necesario para descomprimir una partición del volumen (correspondiente a las etapas de decodificación y transformada inversa) es inferior al tiempo necesario para copiar los datos de la partición en la memoria de texturas y realizar el renderizado. Por otra parte, dependiendo de la naturaleza del volumen a procesar, los tiempos de descompresión presentan importantes variaciones, ya que el rendimiento de ambos algoritmos se ve afectado por el número de bloques con datos no nulos presentes en cada partición. Los tiempos de copia y renderizado solo están influenciados por el tamaño de la partición. En general, incrementar el tamaño de la partición aumenta el tiempo de procesamiento requerido, pero simultáneamente reduce el número de particiones presentes en el volumen. Como se observa en la tabla, utilizar particiones pequeñas no proporciona un buen rendimiento, porque el número de particiones se vuelve demasiado elevado, y utilizar particiones demasiado grandes puede perjudicar también el rendimiento ya que los tiempos de procesamiento crecen de forma exponencial.

Nótese que el tiempo de decodificación para `realhead` con un tamaño de partición de 512^3 que aparece en la tabla 5.4 es inferior al tiempo de decodificación mostrado en la tabla 5.3 para el mismo tamaño de partición. La razón de esta discrepancia radica en el nivel de cuantización utilizado en cada caso. Para la tabla 5.3 el nivel de cuantización utilizado es cero (compresión sin pérdidas), con lo que la mayor parte de las celdas son no nulas. Al aplicar una cuantización de 8 bits (compresión con pérdidas), como se hace para obtener los datos de la tabla 5.4, se eliminan los coeficientes con los valores más bajos, correspondientes con el ruido de fondo, por lo que el número de celdas nulas se incrementa. Como se ha explicado en la sección 5.4.2, las celdas nulas no requieren ningún tipo de procesamiento, por lo que el tiempo de decodificación es menor.

Las figuras 5.18, 5.19 y 5.20 recogen los tiempos de la tabla para cada etapa de la visualización de los diferentes conjuntos de datos y los muestran en forma de gráfica. Se puede observar más claramente cómo el tiempo de copia y renderizado es superior al tiempo de decodificación y transformada inversa en casi todos los casos.

Por último, evaluamos el rendimiento de nuestra solución para la visualización de volúmenes resultado de aplicar las herramientas de segmentación analizadas en los capítulos 3

Dim. part.	Núm. part.	Decod. (CUDA)	T. inv. (CUDA)	Copia (OpenGL)	Render. (OpenGL)	Tiempo part.	Tiempo *frame*	FPS
			Tamaño del volumen:		$32 \times 32 \times 32$			
32^3	1/1	0,000063	0,000012	0,000116	0,002383	0,003	0,003	400
64^3	1/1	0,000062	0,000012	0,000173	0,002813	0,003	0,003	269
128^3	1/1	0,000042	0,000010	0,000344	0,002639	0,003	0,003	268
256^3	1/1	0,000048	0,000012	0,001253	0,003673	0,005	0,005	158
512^3	1/1	0,000048	0,000012	0,007222	0,003600	0,011	0,011	58
			Tamaño del volumen:		$64 \times 64 \times 64$			
32^3	8/8	0,000057	0,000012	0,000115	0,001579	0,002	0,014	56
64^3	1/1	0,000072	0,000017	0,000174	0,003004	0,003	0,003	256
128^3	1/1	0,000064	0,000016	0,000345	0,002901	0,003	0,003	247
256^3	1/1	0,000064	0,000016	0,001256	0,003668	0,005	0,005	155
512^3	1/1	0,000064	0,000016	0,007210	0,003623	0,011	0,011	58
			Tamaño del volumen:		$128 \times 128 \times 128$			
32^3	29/64	0,000047	0,000011	0,000115	0,000853	0,001	0,030	24
64^3	8/8	0,000064	0,000014	0,000172	0,001513	0,002	0,014	55
128^3	1/1	0,000079	0,000051	0,000344	0,003200	0,004	0,004	224
256^3	1/1	0,000070	0,000056	0,001258	0,003715	0,005	0,005	153
512^3	1/1	0,000070	0,000053	0,007173	0,003841	0,011	0,011	57
			Tamaño del volumen:		$256 \times 256 \times 256$			
32^3	115/512	0,000052	0,000011	0,000115	0,000483	0,001	0,076	8
64^3	29/64	0,000054	0,000015	0,000172	0,000860	0,001	0,032	22
128^3	8/8	0,000067	0,000033	0,000344	0,001593	0,002	0,016	47
256^3	1/1	0,000240	0,000226	0,001256	0,003824	0,006	0,006	145
512^3	1/1	0,000240	0,000219	0,007160	0,003766	0,011	0,011	56
			Tamaño del volumen:		$512 \times 512 \times 512$			
32^3	566/4096	0,000048	0,000009	0,000114	0,000292	0,000	0,262	2
64^3	118/512	0,000054	0,000017	0,000171	0,000479	0,001	0,085	7
128^3	29/64	0,000062	0,000026	0,000345	0,000890	0,001	0,038	18
256^3	8/8	0,000159	0,000118	0,001255	0,001747	0,003	0,026	27
512^3	1/1	0,001053	0,000807	0,007146	0,003800	0,013	0,013	52

Tabla 5.5: Tiempos de ejecución en segundos y FPS en una tarjeta GTX 580 para cada una de las etapas de la visualización de volúmenes obtenidos a partir de la segmentación del conjunto de datos `knee` escalados a diferentes tamaños y comprimidos con una cuantización de 8 bits.

Dim. part.	Núm. part.	Decod. (CUDA)	T. inv. (CUDA)	Copia (OpenGL)	Render. (OpenGL)	Tiempo part.	Tiempo *frame*	FPS
			Tamaño del volumen:		$32 \times 32 \times 32$			
32^3	1/1	0,000051	0,000012	0,000119	0,002424	0,003	0,003	309
64^3	1/1	0,000054	0,000012	0,000173	0,002635	0,003	0,003	292
128^3	1/1	0,000065	0,000012	0,000345	0,002640	0,003	0,003	355
256^3	1/1	0,000052	0,000012	0,001253	0,003678	0,005	0,005	153
512^3	1/1	0,000052	0,000012	0,007219	0,003576	0,011	0,011	58
			Tamaño del volumen:		$64 \times 64 \times 64$			
32^3	7/8	0,000062	0,000012	0,000116	0,001533	0,002	0,012	66
64^3	1/1	0,000066	0,000017	0,000173	0,002760	0,003	0,003	277
128^3	1/1	0,000065	0,000017	0,000344	0,003099	0,004	0,004	236
256^3	1/1	0,000066	0,000017	0,001250	0,003220	0,005	0,005	185
512^3	1/1	0,000062	0,000016	0,007234	0,003201	0,011	0,011	60
			Tamaño del volumen:		$128 \times 128 \times 128$			
32^3	29/64	0,000047	0,000011	0,000115	0,000844	0,001	0,030	23
64^3	7/8	0,000058	0,000014	0,000172	0,001529	0,002	0,012	63
128^3	1/1	0,000090	0,000052	0,000345	0,002978	0,003	0,003	237
256^3	1/1	0,000101	0,000059	0,001252	0,003590	0,005	0,005	155
512^3	1/1	0,000094	0,000054	0,007240	0,003226	0,011	0,011	59
			Tamaño del volumen:		$256 \times 256 \times 256$			
32^3	125/512	0,000044	0,000010	0,000115	0,000483	0,001	0,081	8
64^3	31/64	0,000053	0,000013	0,000172	0,000856	0,001	0,034	21
128^3	7/8	0,000061	0,000029	0,000344	0,001544	0,002	0,014	56
256^3	1/1	0,000183	0,000184	0,001257	0,003716	0,005	0,005	148
512^3	1/1	0,000214	0,000204	0,007243	0,003632	0,011	0,011	57
			Tamaño del volumen:		$512 \times 512 \times 512$			
32^3	531/4096	0,000045	0,000009	0,000115	0,000307	0,000	0,253	2
64^3	129/512	0,000052	0,000012	0,000171	0,000476	0,001	0,092	7
128^3	31/64	0,000062	0,000026	0,000345	0,000874	0,001	0,040	18
256^3	7/8	0,000141	0,000104	0,001249	0,001682	0,003	0,022	33
512^3	1/1	0,000810	0,000657	0,007178	0,003697	0,012	0,012	53

Tabla 5.6: Tiempos de ejecución en segundos y FPS en una tarjeta GTX 580 para cada una de las etapas de la visualización de volúmenes obtenidos a partir de la segmentación del conjunto de datos `vessels-1` escalados a diferentes tamaños y comprimidos con una cuantización de 8 bits.

Figura 5.21: Visualización de los conjuntos de datos segmentados. De izquierda a derecha: `knee` y `vessels-1`.

y 4. Concretamente, utilizamos resultados de segmentación obtenidos con el método de segmentación AOS para el conjunto de datos `knee` y resultados obtenidos con el método de segmentación FTC para el conjunto de datos `vessels-1`. En ambos casos, utilizamos volúmenes con tamaños entre 32^3 y 512^3 con el objetivo de estudiar cómo escala el rendimiento de nuestra aplicación de visualización. La figura 5.21 contiene ejemplos de visualización de los conjuntos de datos segmentados.

Las tablas 5.5 y 5.6 muestran el rendimiento obtenido durante la visualización de los volúmenes segmentados, que hemos comprimido utilizando un nivel de cuantización de 8 bits. Para estas pruebas hemos considerado también diferentes configuraciones del tamaño de partición. Dado que los volúmenes segmentados contienen grandes regiones completamente vacías, la segunda columna de ambas tablas muestra el número de particiones con datos frente al total de particiones en que se ha dividido el volumen. Las particiones sin datos no se procesan durante la visualización. El resto de las columnas siguen el esquema de la tabla 5.4.

Se observa que los tiempos de decodificación y transformada inversa varían ligeramente al incrementar el tamaño del volumen, debido al aumento de la cantidad de datos no nulos que deben ser procesados. Los tiempos requeridos por las etapa de copia y renderizado solo se ven afectados por el tamaño de partición utilizado. En todos los casos se obtienen buenos valores de *frames* por segundo. En general, cuando el volumen es pequeño, compensa utilizar tamaños de partición pequeños. En el caso de volúmenes grandes es necesario aumentar el tamaño de la partición si se quiere obtener el mejor rendimiento.

Las figuras 5.23 y 5.22 presentan gráficas con los tiempos de cada etapa para todos los tamaños de volumen y todas las configuraciones de partición analizadas. Dado que el número

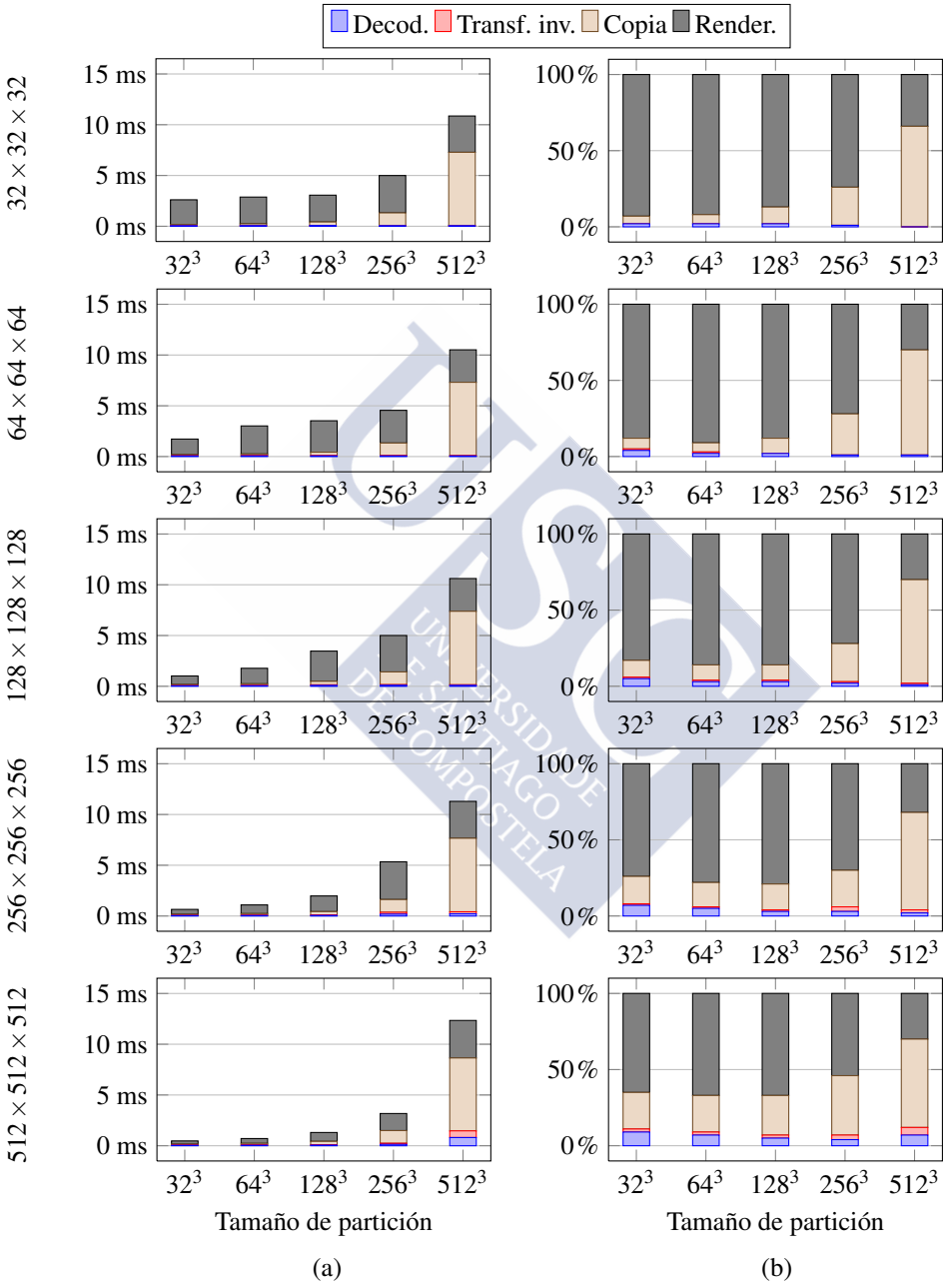


Figura 5.22: Tiempos en una tarjeta GTX 580 para procesar una partición durante la visualización del volumen segmentado `vessels-1` escalado a diferentes tamaños, en (a) valores absolutos y (b) porcentaje sobre el total.

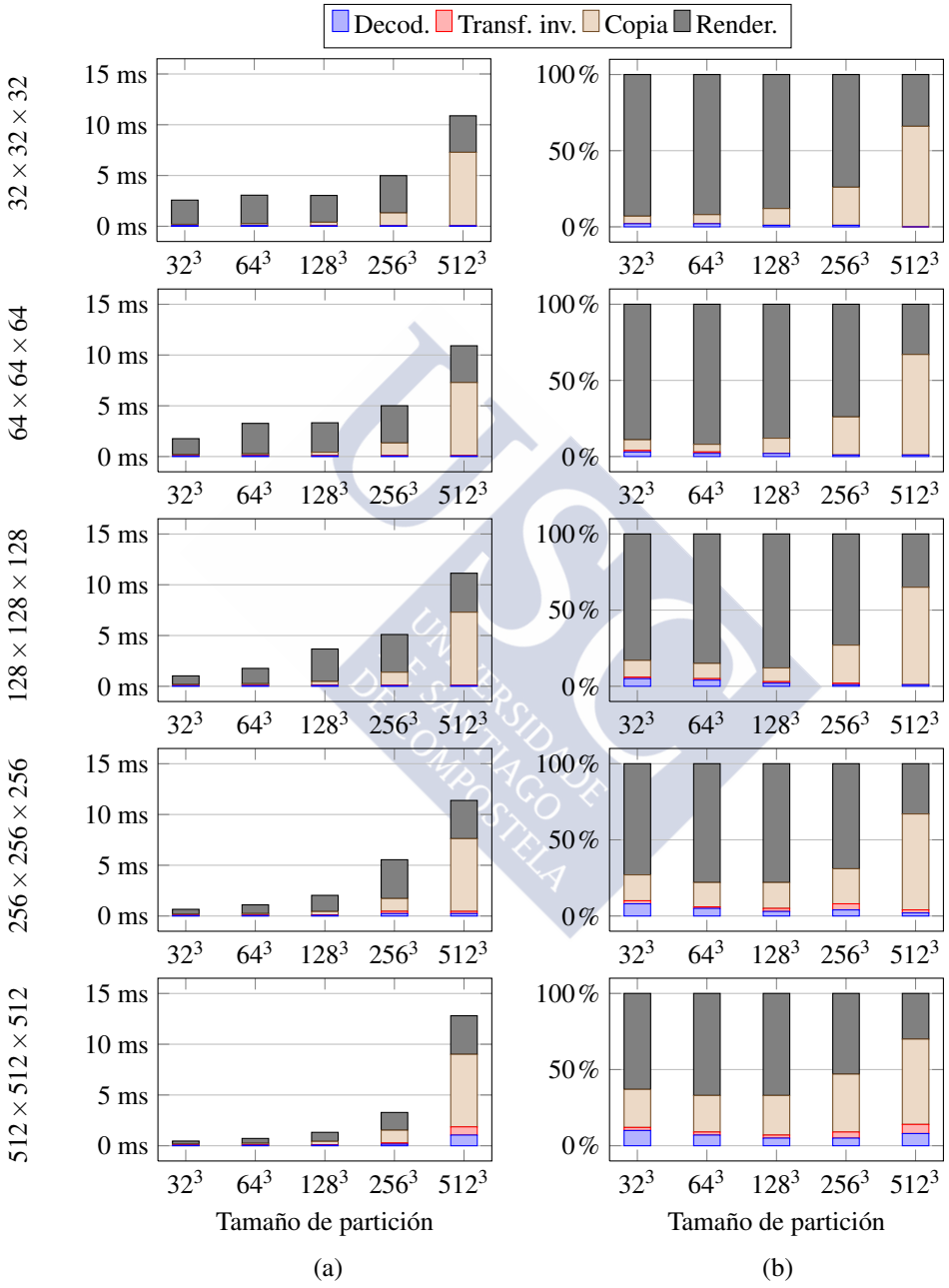


Figura 5.23: Tiempos en una tarjeta GTX 580 para procesar una partición durante la visualización del volumen segmentado `knee` escalado a diferentes tamaños, en (a) valores absolutos y (b) porcentaje sobre el total.

de datos a procesar es muy bajo, se puede observar más claramente que la mayor parte del tiempo se consume en las etapas de copia y renderizado.

Comparación con otros trabajos

Hasta donde tenemos conocimiento, esta es la primera implementación en GPU del esquema de descompresión basado en [84]. Los autores afirman que su solución requiere, en el mejor de los casos, casi 10 segundos para reconstruir un volumen de tamaño 512^3 en la CPU. Este tiempo incluye los pasos de decodificación y transformada inversa. Para una partición del mismo tamaño, la tabla 5.4 muestra un rendimiento de entre 15 y 20 ms para ambos pasos.

Nuestra implementación de la transformada *wavelet* inversa se compara favorablemente con otras implementaciones en la GPU presentes en la literatura. En un trabajo reciente [33], se midió el rendimiento de la transformada *wavelet* rápida en tres dimensiones procesando vídeo a diferentes resoluciones. Esta solución realiza una transformada de un solo nivel aplicando la *wavelet* D4 [54], y los autores presentan sus resultados midiendo el tiempo de procesado para un vídeo de 64 *frames*. Su implementación requiere 6,8 ms para procesar un vídeo de 512^2 píxeles/*frame*, y 13,4 ms en el caso de un vídeo de 1024^2 píxeles/*frame*. Para comparar estos resultados, hemos medido el rendimiento de nuestro *kernel* de transformada inversa para un único nivel en lugar de cuatro: para procesar una partición de 256^3 vóxeles, que es exactamente el mismo tamaño que el vídeo de 64 *frames* de 512^2 píxeles, se requiere 1 ms en nuestra solución. Para una partición de 512^3 vóxeles, que es el mismo tamaño que 64 *frames* de vídeo de 1024^2 píxeles, el tiempo requerido es 7 ms.

El rendimiento del algoritmo completo de visualización en GPU también es competitivo con soluciones similares de la literatura. En [64] se presenta un esquema basado en la transformada Karhunen-Loève [60], que identifica las componentes principales del volumen a partir del cálculo de autovectores. La compresión se realiza en CPU aplicando una cuantización de vectores que conserva los valores de los coeficientes de la transformada en aquellos bloques que contienen los bordes más relevantes. La visualización se realiza en dos pasos de renderizado: el primero descomprime varias láminas de los datos, y el segundo dibuja el volumen. Los autores muestran resultados para un volumen de tamaño 512^3, que se renderiza con una tasa de refresco entre 6 y 11 FPS, dependiendo del tamaño del visor. Para un volumen de tamaño similar (`modelhead`), nuestra solución alcanza los 28 FPS sin que el tamaño del visor afecte a la tasa de refresco.

Por último, en [41] se presenta una solución basada en el algoritmo de compresión de texturas S3 (también conocido como DXT [86]), que divide las texturas en bloques de 8 píxeles y transforma cada bloque en un entero de 32 bits. Este formato de compresión fue diseñado originalmente para ser implementado en *hardware* gráfico, por lo que ofrece buenos resultados de rendimiento en GPU y está implementado en la mayoría de los *drivers* OpenGL. En [41] se utiliza el algoritmo S3 para visualizar conjuntos de datos 3D que varían en el tiempo. La reconstrucción de los datos comprimidos está programada en un *shader*, y los canales RGB de la textura se aprovechan para comprimir hasta 3 *frames* de vídeo 3D. Los autores muestran resultados para un volumen de tamaño $400 \times 600 \times 400$ visualizado a 35 FPS. Aunque este rendimiento es algo mayor que el de nuestra solución, nuestro esquema de compresión obtiene mejores resultados en términos de calidad, con un mayor PSNR para ratios de compresión similares a los usados por los autores.

5.6. Conclusiones

En este capítulo hemos presentado algunas de las herramientas más conocidas para la visualización y procesado de de conjuntos de datos volumétricos. Hemos seleccionado Amira y VolV para visualizar los resultados de los algoritmos de segmentación desarrollados en los capítulos 3 y 4.

Hemos presentado también una solución GPU para la descompresión y visualización multinivel de conjuntos de datos volumétricos. En esta solución la GPU almacena una versión comprimida y jerarquizada del volumen original. Durante la visualización, el volumen comprimido es dividido en particiones que se reconstruyen de una en una. Cada partición es descomprimida hasta un nivel de resolución que depende de su distancia a la cámara. La descompresión implica una etapa de decodificación y una etapa en la que se aplica la transformada *wavelet* inversa. En la solución propuesta la comunicación entre la CPU y la GPU es mínima, por lo que se evita el cuello de botella del bus PCI. Dado que aplicamos cuatro niveles de transformada *wavelet*, nuestra aproximación soporta hasta cuatro niveles diferentes de resolución (cinco incluyendo el nivel de resolución original).

La visualización se lleva a cabo usando la técnica de mapeo de texturas. Los datos de la partición descomprimida se copian en un *buffer* de textura OpenGL y se mapean en una geometría *proxy* compuesta de múltiples láminas poligonales. La GPU dibuja la geometría combinando las láminas para generar la imagen final.

Con los conjuntos de datos probados en esta tesis obtuvimos resultados competitivos con otras implementaciones recientes de visualización de volúmenes comprimidos en GPU. La solución que proponemos es capaz de visualizar volúmenes comprimidos con tamaños entre 256^3 y 512^3 vóxeles con una tasa de refresco entre 30 y 60 FPS, un valor de PSNR mayor que 60 y un ratio de compresión entre 1:4 y 1:18.

La implementación actual utiliza la distancia de cada partición a la cámara como criterio para decidir a qué resolución se procesa y visualiza la región correspondiente del volumen. En la práctica, el usuario podría señalar una región de interés para su visualización a mayor resolución, y el algoritmo mostraría dicha región aumentando la resolución.

Hoy en día, dada la tendencia actual a incrementar el tamaño de los conjuntos de datos, el procesamiento de volúmenes comprimidos en GPU se ha convertido en una necesidad prácticamente ineludible. Evitar el cuello de botella que suponen las comunicaciones entre la CPU y la GPU almacenando el volumen comprimido en la memoria de la GPU tiene aplicaciones en aquellos entornos donde se manejen simultáneamente múltiples volúmenes, incluso si cada uno de estos volúmenes pueden ser almacenado individualmente de forma descomprimida en la memoria de la GPU. Nuestra solución obtiene buenos valores de PSNR y, por tanto, podría ser aplicable al campo de la visualización médica, donde las soluciones de compresión sin pérdidas son siempre preferibles.

Conclusiones y trabajo futuro

El trabajo de investigación llevado a cabo en esta Tesis de Doctorado ha estado orientado hacia la segmentación y la visualización multirresolución de imágenes médicas en GPU. En el campo médico acelerar las tareas de procesado de datos puede contribuir a obtener diagnósticos en menor tiempo, beneficiando tanto a médicos como a pacientes. Las soluciones que hemos propuesto soluciones son accesibles a un amplio rango de usuarios, ya que pueden ejecutarse de forma eficiente en equipos que disponen de una GPU de consumo y que no requieren una gran inversión. En este apartado resumimos las principales contribuciones de este trabajo y proponemos futuras líneas de investigación.

Las soluciones presentadas en esta tesis han sido desarrolladas específicamente para la arquitectura GPU. Esta arquitectura se organiza en multiprocesadores, cada uno de los cuales consiste a su vez en múltiples *cores*. La jerarquía de memoria se organiza en varios espacios. En el espacio de memoria global se almacenan datos que son accesibles para todos los multiprocesadores. Cada multiprocesador dispone de un espacio de memoria privado conocido como memoria compartida, que presenta un tiempo de acceso mucho menor que el de memoria global. El modelo de programación CUDA, empleado en las implementaciones presentadas en esta tesis, divide el flujo de trabajo en hilos de ejecución, que se agrupan en bloques. Cuando se ejecuta una función CUDA (denominada *kernel*), cada bloque se asigna a un multiprocesador. Por su parte, el multiprocesador asigna al bloque de hilos una porción de su espacio de registros y de su espacio de memoria compartida de acuerdo a sus necesidades. Cada bloque de hilos opera de forma independiente a los demás, y no es posible sincronizar de manera eficiente bloques de hilos entre sí, por lo cual la independencia entre bloques es crítica desde el punto de vista del rendimiento.

El trabajo desarrollado en esta Tesis de Doctorado puede resumirse en los siguientes puntos:

– Presentamos un nuevo patrón de paralelismo, que denominamos "divide y fusiona", diseñado para su aplicación en GPU. Este patrón está enfocado a algoritmos multinivel. En este tipo de algoritmos no es posible realizar un reparto trivial de las operaciones entre bloques de hilos debido a las dependencias entre computaciones de diferentes bloques, ya que los resultados parciales generados por un bloque requieren datos de entrada calculados por un bloque diferente. Típicamente, la implementación directa en CUDA de los algoritmos multinivel suele resolver cada nivel en una llamada a un *kernel*, almacenando los resultados en el espacio de memoria global entre una llamada y la siguiente. La aplicación del patrón "divide y fusiona" permite resolver varios niveles de computación en una sola llamada a *kernel* y posibilita una mejor explotación de la jerarquía de memoria de la GPU, al poder aprovechar el espacio de memoria compartida.

  El patrón "divide y fusiona" establece dos fases para completar cada uno de los niveles de computación del algoritmo multinivel. Estas dos fases, división y fusión, cada una de las cuales se computa en un *kernel* ejecutado en memoria compartida, se alternan cíclicamente. Durante la fase de división las operaciones se reparten en bloques de hilos, cada uno de los cuales resuelve parcialmente varios niveles del algoritmo utilizando únicamente los datos contenidos dentro de cada bloque. Durante la fase de fusión se completan aquellas operaciones que, debido a las dependencias entre datos, no pudieron ser procesadas durante la fase de división previa. Cada bloque de hilos opera únicamente con los datos de entrada de que dispone, con lo cual puede ejecutar sus operaciones de forma eficiente en memoria compartida, que ofrece un mayor rendimiento que la memoria global.

  Hemos utilizado el patrón "divide y fusiona" para incrementar la eficiencia en GPU de dos ejemplos de algoritmos multinivel: la resolución de sistemas tridiagonales de ecuaciones lineales y el cálculo de la transformada *wavelet*. En el caso de los sistemas tridiagonales, hemos analizado cuatro algoritmos de resolución paralela: reducción cíclica, doblamiento recursivo, el algoritmo "divide y vencerás" de Bondeli y el método de Wang. Presentamos varias propuestas de implementación tratando de maximizar el rendimiento de los accesos a memoria. Demostramos que la técnica "divide y fusiona" nos permite desarrollar en GPU los valores de aceleración más altos en comparación con una implementación optimizada en OpenMP. En el mejor caso, el algoritmo de reducción cíclica, hemos conseguido para un sistema tridiagonal de 2^{20} ecuaciones un

valor de aceleración de 12,6x en una tarjeta GTX 295 con respecto a una implementación optimizada del método de Bondeli en OpenMP.

En lo que respecta a la transformada *wavelet*, implementamos en GPU tres variantes diferentes, representativas del amplio espectro de transformadas *wavelet* que se pueden encontrar en la bibliografía: Cohen-Daubechies-Feauveau (9,7) con *lifting*, Daubechies D4 con bancos de filtrado y la transformada paquete D4. A pesar de que los grafos de dependencias así como las operaciones ejecutadas por las tres transformadas son diferentes, la aplicación del patrón "divide y fusiona" es similar en los tres casos. También analizamos cómo el patrón puede aplicarse para el cálculo de las transformadas bidimensionales, donde la transformada se ejecuta en dos fases, primero por filas y luego por columnas. Para este caso, considerando una transformada (9,7) bidimensional de tamaño 2048×2048 con 8 niveles de *lifting*, conseguimos en una tarjeta GTX 480 valores de aceleración cercanos a 9,0x respecto a una implementación directa que utilice únicamente memoria global, mientras que con respecto a la implementación en CPU llegamos a obtener aceleraciones de hasta 55,6x.

– En lo que respecta a la segmentación de volúmenes, presentamos una solución en GPU basada en el conjunto de nivel que utiliza el esquema de operador aditivo. En esta solución la evolución del conjunto de nivel está dirigida por una función de parada cuyos valores se determinan antes de iniciar la segmentación. El proceso de segmentación comienza con la selección de una semilla inicial configurada por el usuario, a partir de la cual se inicializa el conjunto de nivel. A continuación, se ejecuta un proceso iterativo, donde se modifican los valores del conjunto del nivel hasta que el frente que delimita el volumen de segmentación adquiere la forma deseada.

  Nuestra propuesta de implementación sigue un esquema de banda estrecha: dividimos el volumen de datos en regiones de tamaño fijo, y en cada iteración restringimos la actualización del conjunto de nivel solo a aquellas regiones que se encuentran más cercanas al frente, con el objetivo de reducir el número de computaciones necesarias para calcular su evolución. Al principio de cada iteración nuestra propuesta genera en GPU tres listas de regiones activas mediante la técnica de compactación de datos. Estas listas contienen los identificadores de las regiones donde se calcularán los valores del conjunto de nivel. A partir de cada lista se construye un sistema de ecuaciones tridiagonal, que se resuelve en GPU mediante el algoritmo de reducción cíclica y utilizando el patrón "divide y fusiona". De esta forma, implementamos de forma eficiente (y haciendo uso

de la memoria compartida) uno de los pasos más costosos del algoritmo debido al gran número de ecuaciones involucradas (que, dependiendo del tamaño de región utilizado, puede ser de varios millones). Como último paso, se promedian los resultados de los tres sistemas de ecuaciones y se actualiza el valor del conjunto de nivel en las regiones activas. La solución que hemos propuesto es, para un volumen de tamaño 256^3, entre 40 y 65 veces más rápida en una tarjeta GTX 680 que una implementación en OpenMP, y ofrece unos resultados de segmentación similares a otras segmentaciones de referencia, con un valor de índice Dice del 85 %.

– Siguiendo con la exploración de los métodos de segmentación en GPU basados en el conjunto de nivel, presentamos dos propuestas de implementación en GPU del método de segmentación rápida. Ambas propuestas siguen también un esquema de banda estrecha donde el conjunto de nivel se divide en regiones de tamaño fijo, en este caso con el objetivo de asignar a cada bloque de hilos una región. Ajustando el tamaño de las regiones, mostramos cómo el cálculo de la evolución del frente de segmentación puede realizarse de forma eficiente en el espacio de memoria compartida.

  El proceso de segmentación de ambas propuestas de implementación presenta características comunes. La segmentación se resuelve de forma iterativa, donde cada bloque de hilos ejecuta en paralelo y de forma independiente a los demás el proceso de segmentación en su región asignada. Cada bloque de hilos ejecuta varias iteraciones de este proceso y dentro de su espacio asignado de memoria compartida, haciendo avanzar o retroceder el frente uno o varios pasos. Cuando el proceso termina, los datos de cada región se copian en memoria global, se calcula la nueva lista de regiones activas de acuerdo a la posición actualizada del frente y, si es necesario, se ejecutan más iteraciones de segmentación. Al procesar el conjunto de nivel de esta forma, es decir, con el frente creciendo de manera independiente en cada región, resulta inevitable que surjan inconsistencias entre regiones adyacentes. Por eso, antes de cada ciclo de iteraciones, los bloques de hilos realizan una revisión de sus regiones asignadas con objeto de localizar las inconsistencias y corregirlas.

  La primera propuesta de implementación del método de segmentación utiliza una lista de regiones activas construida a partir de la posición del frente. La segunda propuesta hace uso de un criterio más restrictivo, estableciendo como activas solo aquellas regiones donde el frente no se ha estabilizado y aún puede evolucionar. En este segundo caso se obtienen los mejores resultados, ya que se reduce el tamaño del dominio activo

(y por tanto, del número de computaciones), y se hace un uso eficiente de la jerarquía de memoria de la GPU. En particular, hemos obtenido para el volumen `kidneys` de tamaño 256^3 y en una tarjeta GTX 580 valores de aceleración de hasta 10,6x frente a una implementación en OpenMP del mismo algoritmo, con resultados de segmentación similares en un 95 %, según el índice Dice, a la implementación de referencia. Ambas propuestas de implementación han sido integradas en la herramienta de procesado de imágenes Amira.

– Por último, en lo que respecta a la visualización de volúmenes, presentamos una solución en GPU basada en la compresión multinivel y la visualización a múltiples niveles de resolución. El proceso de compresión se realiza como una etapa de preprocesado. El volumen original se divide en bloques de tamaño fijo, y sobre cada bloque se aplica una transformada *wavelet*, que permite identificar las componentes más importantes a la vez que se establecen varios niveles de resolución. Tras un proceso de cuantización donde se eliminan las componentes que menos información significativa aportan, reduciendo así el espacio requerido para almacenar la información del volumen, se codifican las componentes restantes siguiendo un esquema de codificación que permite restaurar de forma eficiente el valor original de cualquier vóxel del volumen.

  El volumen comprimido se almacena en la memoria de la GPU. El proceso de descompresión se ha implementado en CUDA y se ejecuta simultáneamente junto con el renderizado en OpenGL. Durante la visualización, se considera una división del volumen en particiones, que se reconstruyen y renderizan una a una hasta un determinado nivel de resolución que depende de su distancia a la cámara. Los bloques asociados a cada partición se decodifican en GPU siguiendo el proceso inverso a la codificación. Los valores restaurados se almacenan en un *buffer* para generar la textura 3D que se visualizará durante el renderizado.

  Todo el proceso descompresión y renderizado se ejecuta íntegramente en la GPU utilizando tanto funciones CUDA como OpenGL. Nuestra solución, que ha sido integrada dentro de la herramienta VolV, permite visualizar volúmenes comprimidos de tamaño entre 256^3 y 512^3 vóxeles con una tasa de refresco entre 30 y 60 *frames* por segundo en una tarjeta GTX 580, con una tasa de compresión entre 1:4 y 1:18 manteniendo una buena calidad de visionado.

En líneas generales, para obtener la máxima eficiencia en GPU hemos aplicado una serie de técnicas que enumeramos a continuación:

– Nuestras soluciones se caracterizan por organizar el dominio computacional en bloques que pueden procesarse de forma independiente. Esto nos ha permitido hacer un uso intensivo del espacio de memoria compartida, cuya latencia de acceso es menor que la de memoria global. Como ya hemos visto, la aplicación del patrón "divide y fusiona" a la ejecución de algoritmos multinivel contribuye a este objetivo, ya que permite resolver varios niveles de computación en una sola llamada a un *kernel* que se ejecuta en memoria compartida.

  La organización del dominio computacional en bloques obliga a intercalar fases intermedias que resuelvan operaciones pendientes entre datos de diferentes bloques (como ocurre al aplicar el patrón "divide y fusiona"), o a corregir posibles inconsistencias entre los datos de los bloques (como ocurre en nuestra propuesta de segmentación rápida de dos ciclos). La eficiencia conseguida mediante esta técnica aumenta al reducir el número de sincronizaciones necesarias y optimizar el uso de la jerarquía de memoria.

– Hemos organizado los datos en memoria tratando de maximizar las localidades espacial y temporal con el objetivo de reducir el número de fallos caché y aumentar la coalescencia de los accesos. Las soluciones que trabajan con datos volumétricos hacen uso además de la memoria de texturas, que permite explotar la localidad de los datos en tres dimensiones. De esta forma, reducimos también el número de accesos a la memoria global.

– En las soluciones de segmentación hemos reducido el dominio computacional de forma que solo se ejecutan los cálculos estrictamente necesarios, es decir, aquellos asociados al dominio activo. Demostramos que la sobrecarga asociada a mantener actualizado el dominio activo compensa con respecto a resolver el problema en todo el dominio. De forma similar, nuestra solución de visualización descomprime para cada partición solo los niveles de resolución necesarios para visualizar correctamente el volumen.

– Hemos utilizado la técnica conocida como compactación de datos (en inglés, *stream compaction*), que permite generar en paralelo y de forma eficiente listas compactas de elementos a partir de mapas binarios. Durante la compactación se calcula una suma de prefijos sobre el mapa binario. Como resultado de dicha suma se obtienen las posiciones

que deben ocupar dentro de la lista los índices de cada uno de los elementos activos en el mapa. Aunque es posible procesar el mapa binario de forma secuencial y realizando menos operaciones para conseguir el mismo resultado, la compactación de datos se adapta mucho mejor al esquema de procesamiento paralelo de la GPU. Hemos aplicado esta técnica en los esquemas de segmentación desarrollados en esta tesis para generar las listas de regiones activas dentro de cada dominio.

– Finalmente, nuestras soluciones minimizan la comunicación entre la CPU y la GPU, ejecutando la mayor parte de los cálculos en la tarjeta gráfica. De esta forma, se evita el cuello de botella del bus PCI. Aunque el coste asociado al tiempo de transferencia de los datos entre la CPU y la GPU al principio y al final de la computación supone una penalización, las implementaciones propuestas pueden ser integradas dentro de una aplicación más grande, minimizando este coste adicional.

El trabajo de esta Tesis de Doctorado ha estado centrado en la arquitectura GPU, pero los métodos desarrollados se pueden aplicar en otro tipo de arquitecturas de características similares. Por ejemplo, en arquitecturas de tipo *cluster*, que presentan un esquema de memoria distribuida donde cada nodo realiza computaciones en paralelo de forma independiente a los demás y la comunicación se realiza a través de una red, las tareas pueden repartirse entre los nodos de forma similar a cómo se reparten los cálculos entre los bloques de hilos de la GPU en nuestras soluciones. Por otra parte, estrategias similares pueden usarse en arquitecturas de acceso uniforme a memoria (en inglés UMA, *uniform memory access*), siempre y cuando se realicen las adaptaciones necesarias para evitar accesos a memoria innecesarios (por ejemplo, no es necesario ejecutar operaciones como la copia de datos entre memoria global y compartida, ya que el espacio de memoria es único). En este tipo de arquitecturas, al repartir las tareas entre procesadores de forma que cada procesador opere siempre sobre el mismo conjunto de datos, se puede conseguir un buen aprovechamiento de la memoria caché.

El trabajo realizado deja abiertas varias líneas de investigación que pueden explorarse en el futuro:

– La gran mayoría de las soluciones que hemos propuesto, especialmente aquellas relacionadas con los métodos de segmentación, hacen un uso intensivo del espacio de memoria compartida. En el momento de escribir esta tesis, NVIDIA ha comenzado a comercializar modelos de GPU con la nueva arquitectura Maxwell, en cuyo diseño se

han incluido 64 kB por multiprocesador dedicados exclusivamente a memoria compartida. Todo parece indicar que la tendencia futura es aumentar este espacio de memoria y darle más importancia dentro de la jerarquía.

– Otro campo abierto a la exploración es la posibilidad de reimplementar las soluciones propuestas en este trabajo repartiendo las tareas entre la CPU y la GPU. Las aproximaciones híbridas requieren necesariamente compartir datos entre las memorias de ambos procesadores y, en muchas ocasiones, aumentar la comunicación a través del bus PCI. En este trabajo hemos evitado este tipo de soluciones, ya que el bus PCI suele ser un cuello de botella, pero las constantes mejoras que está experimentando este bus (con el lanzamiento inminente de la cuarta generación del bus PCI Express) pueden hacer más atractivo el estudio de soluciones híbridas en el futuro. Por otra parte, una de las últimas tendencias entre los fabricantes de *hardware* consiste en juntar en un mismo chip *cores* de CPU y GPU. Las soluciones híbridas podrían beneficiarse de esta nueva arquitectura heterogénea, conocida como APU (del inglés *accelerated processing unit*).

Bibliografía

[1] Amira. http://www.vsg3d.com/amira/overview/. Accedido en febrero de 2014.

[2] BrainWeb simulated brain database. http://www.bic.mni.mcgill.ca/brainweb/. Accedido en noviembre de 2013.

[3] CUDA C best practices guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. Accedido en septiembre de 2013.

[4] CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accedido en septiembre de 2013.

[5] GeForce GTX 280 specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-280/specifications. Accedido septiembre de 2013.

[6] GeForce GTX 295 specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-295/specifications. Accedido septiembre de 2013.

[7] GNU Multiple Precision Arithmetic Library. http://gmplib.org/. Accedido en agosto de 2013.

[8] GPGPU. http://gpgpu.org/. Accedido en octubre de 2013.

[9] MeVisLab. http://www.mevislab.de/. Accedido en febrero de 2014.

[10] MSDN: Compute Shader Overview. http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx. Accedido en septiembre de 2013.

[11] NVIDIA: Direct Compute. https://developer.nvidia.com/directcompute. Accedido en septiembre de 2013.

[12] Oasis. http://www.oasis-brains.org/. Accedido en noviembre de 2013.

[13] OpenCL. https://www.khronos.org/opencl/. Accedido en septiembre de 2013.

[14] OpenMP. http://openmp.org/. Accedido en septiembre de 2013.

[15] Osirix Viewer – Datasets. http://www.osirix-viewer.com/datasets/. Accedido en noviembre de 2013.

[16] Parallel programming and computing platform, CUDA, NVIDIA. http://www.nvidia.com/object/cuda_home_new.html. Accedido en octubre de 2013.

[17] Parallel programming patterns. https://wiki.engr.illinois.edu/display/ppp/Home. Accedido en agosto de 2013.

[18] A pattern language for parallel programming. http://parlab.eecs.berkeley.edu/wiki/patterns/patterns. Accedido en agosto de 2013.

[19] POSIX threads for Win32. http://www.sourceware.org/pthreads-win32/. Accedido en septiembre de 2013.

[20] SCIRun. http://www.sci.utah.edu/cibc-software/scirun.html. Accedido en febrero de 2014.

[21] The Stanford 3D scanning repository. `http://graphics.stanford.edu/data/3Dscanrep/`. Accedido en noviembre de 2013.

[22] Tcl SourceForge Project. `http://tcl.sourceforge.net/`. Accedido en febrero de 2014.

[23] The Visualization Toolkit. `http://www.vtk.org/`. Accedido en febrero de 2014.

[24] Thrust. `http://code.google.com/p/thrust/`. Accedido en marzo 2012.

[25] Voreen - Volume Rendering Engine. `http://www.voreen.org/`. Accedido en febrero de 2014.

[26] ZIBAmira. `http://amira.zib.de/`. Accedido en febrero de 2014.

[27] D. Adalsteinsson y J. A. Sethian. A fast level set method for propagating interfaces. *Journal of Computational Physics*, 118(2):269–277, 1995.

[28] P. Alfaro, P. Igounet y P. Ezzatti. Resolución de matrices tri-diagonales utilizando una tarjeta gráfica (GPU) de escritorio. *Mecánica Computacional*, 29(30):2951–2967, 2010.

[29] S. Allmann, T. Rauber y G. Runger. Cyclic reduction on distributed shared memory machines. En *Euromicro Workshop on Parallel and Distributed Processing*, págs. 290–297. IEEE, 2001.

[30] F. Argüello, D. B. Heras, M. Bóo y J. Lamas-Rodríguez. The split-and-merge method in general purpose computation on GPUs. *Parallel Computing*, 38:277–288, 2012.

[31] C. Bajaj, I. Ihm y S. Park. 3D RGB image compression for interactive applications. *Transactions on Graphics*, 20(1):10–38, 2001.

[32] S. Balla-Arabé, B. Wang y X. Gao. Level set region based image segmentation using lattice Boltzmann method. En *International Conference on Computational Intelligence and Security*, págs. 1159–1163. IEEE, 2011.

[33] G. Bernabe, G. D. Guerrero y J. Fernandez. CUDA and OpenCL implementations of 3D fast wavelet transform. En *Latin American Symposium on Circuits ans Systems*, págs. 1–4. IEEE, 2012.

[34] I. Bitter, R. Van Uitert, I. Wolf, L. Ibanez y J.-M. Kuhnigk. Comparison of four freely available frameworks for image processing and visualization that use ITK. *Transactions on Visualization and Computer Graphics*, 13(3):483–493, 2007.

[35] I. Boada, I. Navazo y R. Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, 2001.

[36] S. Bondeli. Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations. En *Joint International Conference on Vector and Parallel Processing*, págs. 419–434. Springer, 1990.

[37] R. E. Bridson. *Computational aspects of dynamic surfaces*. Tesis doctoral, Standford University, 2003.

[38] S. C. Bushong. *Magnetic resonance imaging*. Elsevier Health Sciences, 2003.

[39] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, 1997.

[40] B. L. Buzbee, G. H. Golub y C. W. Nielson. On direct methods for solving Poisson's equations. *Journal on Numerical Analysis*, 7(4):627–656, 1970.

[41] Y. Cao, L. Xiao y H. Wang. Hardware-accelerated volume rendering based on DXT compressed datasets. En *Conference on Audio Language and Image Processing*, págs. 523–527. IEEE, 2010.

[42] V. Caselles, F. Catté, T. Coll y F. Dibos. A geometric model for active contours in image processing. *Numerische Mathematik*, 66(1):1–31, 1993.

[43] V. Caselles, R. Kimmel y G. Sapiro. Geodesic active contours. *International Journal of Computer Vision*, 22(1):61–79, 1997.

[44] T. F. Chan y L. A. Vese. Active contours without edges. *Transactions on Image Processing*, 10(2):266–277, 2001.

[45] T. F. Chan y L. A. Vese. A level set algorithm for minimizing the Mumford-Shah functional in image processing. En *Workshop on Variational and Level Set Methods in Computer Vision*, págs. 161–168. IEEE, 2001.

[46] C. A. Cocosco, V. Kollokian, R. K.-S. Kwan y A. C. Evans. BrainWeb: Online interface to a 3D MRI simulated brain database. *NeuroImage*, 5(4):425, 1997.

[47] A. Cohen, I. Daubechies y J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 45(5):485–560, 1992.

[48] L. D. Cohen. On active contour models and balloons. *Graphical Models and Image Processing: Image Understanding*, 53(2):211–218, 1991.

[49] R. R. Coifman, Y. Meyer y V. Wickerhauser. Wavelets analysis and signal processing. En *Wavelets and Their Applications*, págs. 153–178. Jones and Bartlett, 1992.

[50] D.L. Collins, A.P. Zijdenbos, V. Kollokian, J.G. Sled, N.J. Kabani, C.J. Holmes y A.C. Evans. Design and construction of a realistic digital brain phantom. *Transactions on Medical Imaging*, 17(3):463–468, 1998.

[51] S. Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Elsevier, 2013.

[52] P. C. Cosman, R. M. Gray y M. Vetterli. Vector quantization of image subbands: a survey. *Transactions on Image Processing*, 5(2):202–225, 1996.

[53] R. Courant, E. Isaacson y M. Rees. On the solution of nonlinear hyperbolic differential equations by finite differences. *Communications on Pure and Applied Mathematics*, 5(3):243–255, 1952.

[54] I. Daubechies. *Ten lectures on wavelets*. Society for Industrial and Applied Mathematics, 1992.

[55] I. Daubechies y W. Sweldens. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis and Applications*, 4(3):247–269, 1998.

[56] A. Davidson y J. D. Owens. Register packing for cyclic reduction: a case study. En *Workshop on General Purpose Processing on Graphics Processing Units*, págs. 4:1–4:6. ACM, 2011.

[57] A. Davidson, Y. Zhang y J. D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. En *International Parallel and Distributed Processing Symposium*, págs. 956–965. IEEE, 2011.

[58] T. M. Deserno. Fundamentals of biomedical image processing. En *Biomedical Image Processing*. Springer, 2011.

[59] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[60] R. D. Dony. Karhunen-Loève transform. En *The Transform and Data Compression Handbook*. CRC Press, 2004.

[61] A. Eklund, P. Dufort, D. Forsberg y S. M. LaConte. Medical image processing on the GPU – past, present and future. *Medical Image Analysis*, 17(8):1073–1094, 2013.

[62] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama y D. Weiskopf. *Real-time volume graphics*. A. K. Peters, 2006.

[63] H. Eviatar y R. L. Somorjai. A fast, simple active contour algorithm for biomedical images. *Pattern Recognition Letters*, 17(9):969–974, 1996.

[64] N. Fout y K.-L. Ma. Transform coding for hardware-accelerated volume rendering. *Transactions on Visualization and Computer Graphics*, 13(6):1600–1607, 2007.

[65] J. Franco, G. Bernabe, J. Fernandez, M. E. Acacio y M. Ujaldon. The GPU on the 2D wavelet transform. Survey and contributions. En *International Workshop on State-of-the-Art in Scientific and Parallel Computing*, págs. 173–183. Springer, 2010.

[66] A. V. Gelder y K. Kim. Direct volume rendering with shading via three-dimensional textures. En *Symposium on Volume Visualization*, págs. 23–30, 98. IEEE, 1996.

[67] E. Gobbetti, J. A. Iglesias Guitián y F. Marton. COVRA: a compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks. *Computer Graphics Forum*, 31(3pt4):1315–1324, 2012.

[68] E. Gobbetti y F. Marton. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(6):815–826, 2004.

[69] E. Gobbetti, F. Marton y J. A. Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7–9):797–806, 2008.

[70] D. Göddeke y R. Strzodka. Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid. *Transactions on Parallel and Distributed Systems*, 22(1):22–32, 2011.

[71] R. Goldenberg, R. Kimmel, E. Rivlin y M. Rudzsky. Fast geodesic active contours. *Transactions on Image Processing*, 10(10):1467–1475, 2001.

[72] P. González, J. C. Cabaleiro y T. F. Pena. Parallel computation of wavelet transforms using the lifting scheme. *The Journal of Supercomputing*, 18(2):141–152, 2001.

[73] P. Goswami, F. Erol, R. Mukhi, R. Pajarola y E. Gobbetti. An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *The Visual Computer*, 29(1):69–83, 2013.

[74] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, Smith. B. y J. Manferdelli. High performance discrete Fourier transforms on graphics processors. En *International Conference on High Performance Computing, Networking, Storage and Analysis*, págs. 2:1–2:12. ACM, IEEE, 2008.

[75] R. M. Gray y D. L. Neuhoff. Quantization. *Transactions on Information Theory*, 44(6):2325–2383, 1998.

[76] S. Guthe y W. Strasser. Advanced techniques for high-quality multi-resolution volume rendering. *Computers & Graphics*, 28(1):51–58, 2004.

[77] S. Guthe, M. Wand, J. Gonser y W. Strasser. Interactive rendering of large volume data sets. En *Visualization*, págs. 53–60. IEEE, 2002.

[78] A. Hagan y Y. Zhao. Parallel 3D image segmentation of large data sets on a GPU cluster. *Advances in Visual Computing*, 5876:960–969, 2009.

[79] M. Harris, S. Sengupta y J. D. Owens. Parallel prefix sum (scan) with CUDA. En *GPU Gems 3*. Pearson Education, 2007.

[80] M. H. Hayes. Overlap-add. En *Schaum's Outline of Theory and Problems of Digital Signal Processing*. McGraw-Hill, 1999.

[81] D. L. G. Hill, P. G. Batchelor, M. Holden y D. J. Hawkes. Medical image registration. *Physics in Medicine and Biology*, 46(3):R1, 2001.

[82] R. W. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM*, 12(1):95–113, 1965.

[83] G. N. Hounsfield. Apparatus for examining a body by radiation such as X or gamma radiation, 1976. Patente EE.UU. 3944833.

[84] I. Ihm y S. Park. Wavelet-based 3D compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18(1):3–15, 1999.

[85] Intel Corporation, Santa Clara, California, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide*, May 2011.

[86] K. I. Iourcha, K. S. Nayak y Z. Hong. System and method for fixed-rate block-based image compression with inferred pixel values, 1999. Patente EE.UU. 5956431.

[87] A. C. Jalba, W. J. van der Laan y J. B. T. M. Roerdink. Fast sparse level sets on graphics hardware. *Transactions on Visualization and Computer Graphics*, 19(1):30–44, 2013.

[88] W.-K. Jeong, J. Beyer, M. Hadwiger, A. Vazquez, H. Pfister y R. T. Whitaker. Scalable and interactive segmentation and visualization of neural processes in EM datasets. *Transactions on Visualization and Computer Graphics*, 15(6):1505–1514, 2009.

[89] W. Jiang y A. Ortega. Lifting factorization-based discrete wavelet transform architecture design. *Transactions on Circuits and Systems for Video Technology*, 11(5):651–657, 2001.

[90] M. Kass, A. Lefohn y J. D. Owens. Interactive depth of field using simulated diffusion. Informe técnico, Pixar Animation Studios, 2006.

[91] S. Kichenassamy, A. Kumar, P. Olver, A. Tannenbaum y A. Yezzi Jr. Conformal curvature flows: from phase transitions to active vision. *Archive for Rational Mechanics and Analysis*, 134(3):275–301, 1996.

[92] K. J. Kirchberg, A. Wimmer y C. H. Lorenz. Modeling the human aorta for MR-driven real-time virtual endoscopy. En *Medical Image Computing and Computer-Assisted Intervention*, págs. 470–477. Springer, 2006.

[93] David B. Kirk y Wen-mei W. Hwu. *Programming Massively Parallel Processors: a Hands-on Approach*. Elsevier, 2010.

[94] O. Klar. Interactive GPU based segmentation of large medical volume data with level sets. En *Central European Seminar on Computer Graphics*, 2006.

[95] R. K.-S. Kwan, A. C. Evans y G. B. Pike. An extensible MRI simulator for post-processing evaluation. En *International Conference on Visualization in Medical Computing*, págs. 135–140. Springer, 1996.

[96] R. K.-S. Kwan, A.C. Evans y G.B. Pike. MRI simulation-based evaluation of image-processing and classification methods. *Transactions on Medical Imaging*, 18(11):1805–1897, 1999.

[97] Wladimir J. Laan, A. C. Jalba y J. B. Roerdink. A memory and computation efficient sparse level-set method. *Journal of Scientific Computing*, 46:243–264, 2011.

[98] E. LaMar, B. Hamann y K. I. Joy. Multiresolution techniques for interactive texture-based volume visualization. En *Visualization*, págs. 355–361. IEEE, 1999.

[99] J. Lamas-Rodríguez, F. Argüello y D. B. Heras. A GPU-based multiresolution pipeline for compressed volume rendering. En *International Conference on Parallel and Distributed Processing Techniques and Applications*, págs. 523–529. World Academy of Science, 2013.

[100] J. Lamas-Rodríguez, F. Argüello, D. B. Heras y M. Bóo. Memory hierarchy optimization for large tridiagonal system solvers on GPU. En *International Symposium on Parallel and Distributed Processing with Applications*, págs. 87–94. IEEE, 2012.

[101] J. Lamas-Rodríguez, M. Bóo, Dora B. Heras y F. Argüello. Proyección de algoritmos de resolución de sistemas tridiagonales en la tarjeta gráfica. En *XX Jornadas de Paralelismo*, págs. 349–354. Sociedad de Arquitectura y Tecnología de Computadores, 2009.

[102] J. Lamas-Rodríguez, D. B. Heras, F. Argüello, D. Kainmueller, S. Zachow y M. Bóo. GPU-accelerated level-set segmentation. *Journal of Real Time-Image Processing*, págs. 1–15, 2013.

[103] J. Lamas-Rodríguez, D. B. Heras, F. Argüello, S. Zachow y D. Kainmueller. Partitioning and mapping a fast level-set algorithm on the GPU. En *Conference on*

Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications. IEEE, 2013.

[104] J. Lamas-Rodríguez, D. B. Heras, M. Bóo y F. Argüello. Tridiagonal system solvers. Informe técnico, University of Santiago de Compostela, 2011.

[105] J. Lamas-Rodríguez, P. Quesada-Barriuso, F. Argüello, D. B. Heras y M. Bóo. Proyección del método de segmentación del conjunto de nivel en GPU. En *XXIII Jornadas de Paralelismo*, págs. 273–278. Sociedad de Arquitectura y Tecnología de Computadores, 2012.

[106] J. J. Lambiotte y R. G. Voit. The solution of tridiagonal linear systems on the CDC STAR-100 computer. *Transactions on Mathematical Software*, 1(4):308–329, 1975.

[107] J. R. Landis y G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[108] A. E. Lefohn, J. E. Cates y R. T. Whitaker. Interactive, GPU-based level sets for 3D brain tumor segmentation. Informe técnico, University of Utah, School of Computing, 2003.

[109] A. E. Lefohn, J. E. Cates y R. T. Whitaker. Interactive, GPU-based level sets for 3D segmentation. En *Medical Image Computing and Computer Assisted Intervention*, págs. 564–572. Springer, 2003.

[110] A. E. Lefohn, J. M. Kniss, C. D. Hansen y R. T. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. En *Visualization*, págs. 11–18. IEEE, 2003.

[111] A. E. Lefohn, J. M. Kniss, C. D. Hansen y R. T. Whitaker. A streaming narrow-band algorithm: interactive computation and visualization of level sets. *Transactions on Visualization and Computer Graphics*, 10(4):422–433, 2004.

[112] E. Lindholm, J. Nickolls, S. Oberman y J. Montrym. NVIDIA Tesla: a unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[113] B. Liu, G. J. Clapworthy, F. Dong y E. C. Prakash. Octree rasterization: accelerating high-quality out-of-core GPU volume rendering. *Transactions on Visualization and Computer Graphics*, 19(10), 2013.

[114] R. Malladi, J. A. Sethian y B. C. Vemuri. Shape modeling with front propagation: a level set approach. *Transactions on Pattern Analysis and Machine Intelligence*, 17(2):158–175, 1995.

[115] S. G. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, 1989.

[116] S. G. Mallat. *A wavelet tour on signal processing*. Academic Press, 2008.

[117] T. G. Mattson, B. A. Sanders y B. L. Massingill. *Patterns for parallel programming*. Addison-Wesley, 2004.

[118] B. Merriman, J. Bence y S. Osher. *Diffusion generated motion by mean curvature*. University of California, Los Angeles, 1992.

[119] B. Merriman, J. Bence y S. Osher. Motion of multiple junctions: a level set approach. *Journal of Computational Physics*, 112:334–363, 1994.

[120] F. G. Meyer, A. Z. Averbuch y J.-O. Stromberg. Fast adaptive wavelet packet image compression. *Transactions on Image Processing*, 9(5):792–800, 2000.

[121] U. Meyer-Baese. *Digital signal processing using field programmable gate arrays*. Springer, 2007.

[122] K. W. Morton y D. F. Mayers. *Numerical solution of partial differential equations*. Cambridge University Press, 2005.

[123] S. M. Müller y D. Sheerer. A method to parallelize tridiagonal solvers. *Parallel Computing*, 17(2–3):181–188, 1991.

[124] D. Mumford y J. Shah. Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics*, 42(5):577–685, 1989.

[125] K. G. Nguyen y D. Saupe. Rapid high quality compression of volume data for visualization. 20(3):49–57, 2001.

[126] J. Nickolls, I. Buck, M. Garland y K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[127] M. Niethammer, P. A. Vela y A. Tannenbaum. Geometric observers for dynamically evolving curves. En *Conference on Decision and Control*, págs. 6071–6077. IEEE, 2005.

[128] A. Nukada, Y. Ogata, T. Endo y S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. En *International Conference on High Performance Computing, Networking, Storage and Analysis*, págs. 5:1–5:11. ACM, IEEE, 2008.

[129] NVIDIA, Santa Clara, California, USA. *CUDA C best practices guide (version 4.0)*, 2011.

[130] S. Osher. Level set methods. En *Geometric level set methods in imaging, vision, and graphics*. Springer, 2003.

[131] S. Osher y R. Fedkiw. *Level set methods and dynamic implicit surfaces*. Springer, 2002.

[132] S. Osher y J. A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[133] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone y J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[134] M.A. Palis. The granularity metric for fine-grain real-time scheduling. *Transactions on Computers*, 54(12):1572–1583, 2005.

[135] R. Parys y G. Knittel. Giga-voxel rendering from compressed data on a display wall. *Journal of WSCG*, 17(1–3):73–80, 2009.

[136] D. Peng, B. Merriman, S. Osher, H. Zhao y M. Kang. A PDE-based fast local level set method. *Journal of Computational Physics*, 155(2):410–438, 1999.

[137] P. Perona y J. Malik. Scale-space and edge detection using anisotropic diffusion. *Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, 1990.

[138] M. Pfeifle, S. Born, J. Fischer, F. Duffner, J. Hoffmann y D. Bartz. VolV – Eine OpenSource-Plattform für die medizinische Visualisierung. En *Annual Meeting of the German Society for Computer and Robot-Assisted Surgery*. German Society for Computer and Robot-Assisted Surgery, 2007.

[139] M. J. Piggott, P. Vallotton, J. A. Taylor y T. P. Bednarz. Accelerated implementation of level set based segmentation. *Australia and New Zealand Industrial and Applied Mathematics Journal*, 54:C327–C344, 2013.

[140] V. Podlozhnyuk. Image convolution with CUDA. NVIDIA Whitepaper, 2007.

[141] B. Preim y D. Bartz. *Visualization in medicine: theory, algorithms and applications*. Elsevier, 2007.

[142] W. H. Press, B. P. Flannery, S. A. Teukolsky y W. T. Vetterling. *Numerical recipes in C: the art of scientific computing*. Cambridge University Press, 1992.

[143] G. Pryor, T. ur Rehman, S. Lankton, P. A. Vela y A. Tannenbaum. Fast optimal mass transport for dynamic active contour tracking on the GPU. En *Conference on Decision and Control*, págs. 2681–2688. IEEE, 2007.

[144] P. Quesada-Barriuso, J. Lamas-Rodríguez, D. B. Heras, M. Bóo y F. Argüello. Selecting the best tridiagonal system solver projected on multi-core CPU and GPU platforms. En *International Conference on Parallel and Distributed Processing Techniques and Applications*, págs. 839–845. World Academy of Science, 2011.

[145] M. Roberts, J. Packer, M. C. Sousa y J. R. Mitchell. A work-efficient GPU algorithm for level set segmentation. En *Conference on High Performance Graphics*, págs. 123–132. Eurographics Association, 2010.

[146] M. Roberts, M. C. Sousa y J. R. Mitchell. Level set segmentation of volume data, 2010. Patente EE.UU. 12/924,325.

[147] M. B. Rodríguez, E. Gobbetti, J. A. Iglesias Guitián, M. Makhinya, F. Marton, R. Pajarola y S. K. Suter. A survey of compressed GPU-based direct volume rendering. En *Eurographics*, págs. 117–136. The Eurographics Association, 2013.

[148] M. B. Rodríguez, E. Gobbetti, J. A. Iglesias Guitián, M. Makhinya, F. Marton, R. Pajarola y S.K. Suter. State-of-the-art in compressed GPU-based direct volume rendering. *Computer Graphics Forum*, 2014.

[149] S. D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, 1982.

[150] M. Rumpf y R. Strzodka. Level set segmentation in graphics hardware. En *International Conference on Image Processing*, volumen 3, págs. 1103–1106. IEEE, 2001.

[151] S. Rusinkiewicz y M. Levoy. QSplat: a multiresolution point rendering system for large meshes. En *Conference on Computer Graphics and Interactive Techniques*, págs. 343–352. SIGGRAPH, 2000.

[152] J. Sanders y E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. NVIDIA, 2010.

[153] J. S. Schildkraut, N. Prosser, A. Savakis, J. Gomez, D. Nazareth, A. K. Singh y H. K. Malhotra. Level-set segmentation of pulmonary nodules in megavolt electronic portal images using a CT prior. *Medical Physics*, 37:5703, 2010.

[154] J. Schneider y R. Westermann. Compression domain volume rendering. En *Visualization*. IEEE, 2003.

[155] S. Sengupta, M. Harris, Y. Zhang y J. D. Owens. Scan primitives for GPU computing. En *Symposium on Graphics Hardware*, págs. 97–106. ACM, The Eurographics Association, 2007.

[156] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[157] J. A. Sethian. *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*. Cambridge University Press, 1999.

[158] L. Shapiro y G. Stockman. Image segmentation. En *Computer Vision*. Prentice Hall, 2001.

[159] O. Sharma y F. Anton. CUDA based level set method for 3D reconstruction of fishes from large acoustic data. En *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. Eurographics Association, ACM, SIGGRAPH, 2009.

[160] O. Sharma, Q. Zhang, F. Anton y C. Bajaj. Multi-domain, higher order level set scheme for 3D image segmentation on the GPU. En *Conference on Computer Vision and Pattern Recognition*, págs. 2211–2216. IEEE, 2010.

[161] O. Sharma, Q. Zhang, F. Anton y C. Bajaj. Fast streaming 3D level set segmentation on the GPU for smooth multi-phase segmentation. *Transactions on Computational Science XIII*, págs. 72–91, 2011.

[162] Y. Shi y W. C. Karl. A real-time algorithm for the approximation of level-set-based curve evolution. *Transactions on Image Processing*, 17(5):645–656, 2008.

[163] K. Siddiqi, Y. B. Lauziere, A. Tannenbaum y S. W. Zucker. Area and length minimizing flows for shape segmentation. *Transactions on Image Processing*, 7(3):433–443, 1998.

[164] S. Spacey, W. Luk, P. H. J. Kelly y D. Kuhn. Improving communication latency with the write-only architecture. *Journal of Parallel and Distributed Computing*, 72:1617–1627, 2012.

[165] D. Stalling, M. Westerhoff y H.-C. Hege. Amira: a highly interactive system for visual data analysis. En *The visualization handbook*. Elsevier, 2005.

[166] H. S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20(1):27–38, 1973.

[167] J. Strain. Semi-Lagrangian methods for level set equations. *Journal of Computational Physics*, 151(2):498–533, 1999.

[168] J. Strain. Tree methods for moving interfaces. *Journal of Computational Physics*, 151(2):616–648, 1999.

[169] J. Strain. A fast modular semi-Lagrangian method for moving interfaces. *Journal of Computational Physics*, 161(2):512–536, 2000.

[170] S. K. Suter, J. A. Iglesias Guitian, F. Marton, M. Agus, A. Elsener, C. P. E. Zollikofer, M. Gopi, E. Gobbetti y R. Pajarola. Interactive multiscale tensor reconstruction for multiresolution volume visualization. *Transactions on Visualization and Computer Graphics*, 17(12):2135–2143, 2011.

[171] S. K. Suter, M. Makhynia y R. Pajarola. TAMRESH – Tensor approximation multiresolution hierarchy for interactive volume visualization. En *Computer Graphics Forum*, volumen 32, págs. 151–160. The Eurographics Association, 2013.

[172] G. Tao, A. Singh y L. Bidaut. Liver segmentation from registered multiphase CT data sets with EM clustering and GVF level set. En *SPIE Medical Imaging*, págs. 76230V–76230V. International Society for Optics and Photonics, 2010.

[173] R. Taylor, M. Kazhdan y B. Lucas. Multi-Object Geodesic Active Contours (MOGAC): a parallel sparse-field algorithm for image segmentation. Informe técnico, Johns Hopkins Whiting School of Engineering, 2012.

[174] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel y F. Tirado. Parallel implementation of the 2D discrete wavelet transform on graphics processing units: filter bank versus lifting. *Transactions on Parallel and Distributed Systems*, 19(3):299–310, 2008.

[175] L. H. Thomas. Elliptic problems in linear difference equations over a network. Informe técnico, Columbia University, 1949.

[176] P. E. Tikkanen. Nonlinear wavelet and wavelet packet denoising of electrocardiogram signal. *Biological Cybernetics*, 80(4):259–267, 1999.

[177] G. J. Tornai y G. Cserey. 2D and 3D level-set algorithms on GPU. En *International Workshop on Cellular Nanoscale and their Applications*, págs. 1–5. IEEE, 2010.

[178] W. J. van der Laan, A. C. Jalba y J. B. T. M. Roerdink. A memory and computation efficient sparse level-set method. *Journal of Scientific Computing*, 46(2):1–22, 2011.

[179] A. Vazquez-Reina, E. Miller y H. Pfister. Multiphase geometric couplings for the segmentation of neural processes. En *Conference on Computer Vision and Pattern Recognition*, págs. 2020–2027. IEEE, 2009.

[180] H. H. Wang. A parallel method for tridiagonal equation. *Transactions on Mathematical Software*, 7(2):170–183, 1981.

[181] Y.-S. Wang, C. Wang, T.-Y. Lee y K.-L. Ma. Feature-preserving volume data reduction and focus+context visualization. *Transactions on Visualization and Computer Graphics*, 17(2):171–181, 2011.

[182] Z. Wang y A. C. Bovik. Mean squared error: love it or leave it? A new look at signal fidelity measures. *Signal Processing Magazine*, 26(1):98–117, 2009.

[183] J. Weickert y G. Kühne. Fast methods for implicit active contour models. En *Geometric Level Set Methods in Imaging, Vision y Graphics*. Springer, 2003.

[184] J. Weickert, B. M. ter Haar Romeny y M. A. Viergever. Efficient and reliable schemes for nonlinear diffusion filtering. *Transactions on Image Processing*, 7(3):398–410, 1998.

[185] R. T. Whitaker. Volumetric deformable models: active blobs. En *Visualization in Biomedical Computing*, volume 2359, págs. 122–134. SPIE, 1994.

[186] R. T. Whitaker. A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision*, 29(3):203–231, 1998.

[187] I. Wolf. Toolkits and software for developing biomedical image processing and analysis applications. En *Biomedical Image Processing*. Springer, 2011.

[188] Q. Zhang, R. Eagleson y T. M. Peters. Volume visualization: a technical overview with a focus on medical applications. *Journal of Digital Imaging*, 24(4):640–664, 2011.

[189] Y. Zhang, J. Cohen y J. D. Owens. Fast tridiagonal solvers on the GPU. En *Symposium on Principles and Practice of Parallel Computing*, págs. 127–136. ACM, 2010.

[190] F. Zhao y X. Xie. An overview of interactive medical image segmentation. *Annals of the BMVA*, 7:1–22, 2013.

[191] Y. Zhao. Lattice Boltzmann based PDE solver on the GPU. *The Visual Computer*, 24:323–333, 2008.

[192] A. P. Zijdenbos, B. M. Dawant, R. A. Mangolin y A. C. Palmer. Morphometric analysis of white matter lesions in MR images: method and validation. *Transactions on Medical Imaging*, 13(4):716–724, 1994.

[193] T. Zuva, O. O. Oludayo, S. O. Ojo y S. M. Ngwira. Image segmentation, available techniques, developments and open issues. *Canadian Journal on Image Processing and Computer Vision*, 2(3):20–29, 2011.

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Índice de figuras

Índice de tablas

USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA