Proyecto de Sistemas Informáticos.
Curso 2006-2007.

Uso de hardware gráfico para la aceleración de métodos algebraicos de reconstrucción

http://gpd.sip.ucm.es/enrique/gpusart/

Autores:
David García Soriano
Enrique Martín Martín
David Romero Laorden

Director del proyecto:
Christian Tenllado van der Reijden

Facultad de Informática.
Universidad Complutense de Madrid.

Uso de hardware gráfico para la aceleración de métodos algebraicos de reconstrucción.

Memoria del proyecto de fin de carrera presentado por David García Soriano, Enrique Martín Martín y David Romero Laorden en la Universidad Complutense de Madrid, y realizado bajo la dirección de Christian Tenllado van der Reijden.

Madrid, 5 de septiembre de 2008.

David García Soriano

Enrique Martín Martín

David Romero Laorden

Resumen

El objetivo de este proyecto ha sido implementar un algoritmo de reconstrucción algebraica (en particular SART, *Simultaneous Algebraic Reconstruction Technique*) y acelerar su ejecución utilizando los recursos paralelos que ponen a disposición del programador las unidades de procesamiento gráfico de hoy en día. Para ello, primero se han desarrollado dos variantes del algoritmo que sólo utilizan la CPU, en las que nos hemos basado para posteriormente portar a la GPU las partes susceptibles de obtener una mejora substancial de velocidad. Por último, se ha analizado en ambas variantes la calidad de las reconstrucciones y el *speedup* obtenido respecto a las implementaciones en CPU, así como el impacto en la calidad de los diversos parámetros configurables del algoritmo.

Summary

The aim of our project has been the implementation of an algebraic reconstruction algorithm (known as SART, *Simultaneous Algebraic Reconstruction Technique*) and improvement of its running time by making use of the massively parallel computational resources which today's graphics processing units provide the programmer with. To this end, two variants of the algorithm have first been developed which are executed entirely on the CPU, and upon which we have based our subsequent porting to the GPU of those parts having the potential to enjoy a substantial increase in speed. Finally, the quality of both variants' reconstructions has been analyzed, along with the speedup obtained over CPU implementations and the impact that various customizable parameters of the algorithm have on image quality.

Lista de palabras clave

Técnicas de reconstrucción algebraica, GPGPU, reconstrucción tridimensional, tomografía computerizada, CT, CT de rayos paralelos, ART, SART, hardware gráfico, CUDA.

Keyword list

Algebraic reconstruction techniques, GPGPU, three-dimensional reconstruction, computed tomography, CT, parallel-beam CT, ART, SART, graphics hardware, CUDA.

Agradecimientos

Queremos mostrar nuestra gratitud a Christian Tenllado van der Reijden y a Manuel Prieto Matías por habernos dado la oportunidad de elaborar este proyecto y por la ayuda que nos han proporcionado.

También queremos aprovechar esta ocasión para agradecer a Carlos Óscar Sánchez Sorzano, del CNB, la documentación amablemente cedida y el alto grado de interés mostrado por nuestros resultados.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este documento tiene el objetivo de dar una visión a distintos niveles sobre nuestro proyecto de *Sistemas Informáticos*, pudiéndose encontrar en él información sobre los distintos aspectos que lo forman. En este capítulo se pretende introducir brevemente cuáles son nuestros objetivos, las herramientas que hemos utilizado y la estructura del resto de la memoria.

El área de aplicación del proyecto es el de la **Tomografía Computerizada**, cuyo propósito es la reconstrucción de la estructura tridimensional interna de un objeto a partir de una serie de proyecciones bidimensionales digitalizadas obtenidas al iluminarlo desde diversas direcciones. Las proyecciones medidas contienen lo que se conoce como la *"transformada de rayos X"* del objeto (esto será precisado más adelante, pero la idea a tener en cuenta es que cada valor medido en un punto de la imagen depende de todos los puntos del objeto que ha atravesado un rayo, influyendo más los puntos con mayor densidad). Estas imágenes son procesadas por un computador, y la salida del proceso es una modelización del volumen que concuerde lo mejor posible con los datos experimentales (idealmente, una función real que a cada punto del espacio le asigne, por ejemplo, la densidad del objeto en él). Un ejemplo de reconstrucción del antígeno T se muestra en las figuras 1.1, 1.2 y 1.3.

Las técnicas tomográficas tienen muy diversos campos de aplicación [KS88,

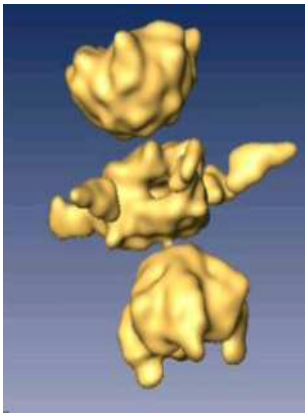

FIGURA 1.1: Representación de la isosuperficie del antígeno T. Fuente: [Sor02]

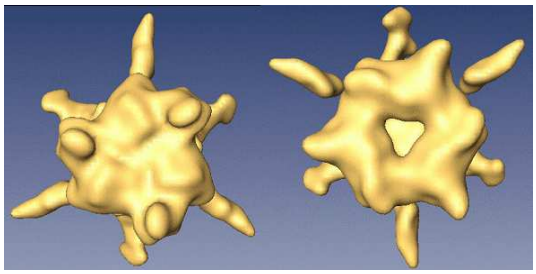

FIGURA 1.2: Representación de la superficie del antígeno T, vista desde la parte inferior y superior respectivamente. Fuente: [Sor02]

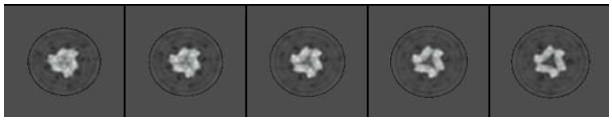

FIGURA 1.3: Algunos planos de la reconstrucción del antígeno T tomados desde arriba. Fuente: [Sor02]

Wil00]; citaremos tres de ellas a modo de ejemplo (y sin pretender en modo alguno cubrir todas las posibilidades):

- En medicina son indispensables como herramienta de diagnóstico en hospitales, al permitir la observación de los órganos internos de los pacientes con una precisión sin precedentes.
- En biología molecular se utilizan para la determinación experimental de la estructura de complejos macromoleculares.
- En astronomía se aplican a la determinación de la distribución de brillo

estelar sobre la esfera celeste.

Cada una de las técnicas de obtención de los datos necesarios (principalmente rayos X, resonancia magnética nuclear y microscopía electrónica de transmisión) tiene una serie de limitaciones (rango de tamaños del objeto, máxima resolución alcanzable, nivel de ruido...), y por tanto los algoritmos de reconstrucción se adaptan en función del método empleado para la adquisición de datos, con el fin de tratar de superar estas limitaciones.

Nosotros nos hemos centrado en los llamados métodos algebraicos de reconstrucción, de naturaleza iterativa. En general, no nos preocupamos por la naturaleza exacta de los objetos a reconstruir, ni de los detalles del proceso físico de obtención de datos; en su lugar hemos abordado el problema de reconstrucción desde un punto de vista matemático, teniendo en cuenta el significado real de los datos de entrada y la presencia de ruido en las proyecciones medidas. Sin embargo, se han utilizado datos proporcionados por el Centro Nacional de Biotecnología (**CNB-CSIC**), y la calidad de los resultados ha sido comparada con los producidos por la suite de aplicaciones **Xmipp** (***X**-Windows-based **M**icroscopy **I**mage **P**rocessing **P**ackage*, paquete de procesamiento de imágenes de microscopía basado en X-Windows), desarrollado en el CNB y de libre distribución, bajo licencia GPL [SMVM$^+$04, MMM$^+$96].

La principal desventaja de las implementaciones en software de estos algoritmos es que se trata de un proceso computacionalmente intensivo. Debido al gran número de proyecciones necesario para obtener reconstrucciones de alta calidad, que junto a la necesidad de conseguir la mayor resolución posible hace crecer demasiado el volumen de datos a tratar, el proceso carece de la velocidad que sería deseable en muchas aplicaciones. Como ejemplo, el programa de reconstrucción de Xmipp tarda unos 24 minutos en reconstruir un objeto a partir de 1000 imágenes de 128×128 píxeles en una CPU de alta gama (Intel Core duo 6600 a 2.4 Ghz).

Este problema se acentúa particularmente el caso de los algoritmos iterativos, cuyo uso es preferible sobre otras alternativas (las que trabajan en

el espacio de Fourier) bajo ciertas condiciones (por ejemplo cuando las proyecciones no están distribuidas uniformemente en ángulo o no cubren todas las orientaciones). Esto provoca que la velocidad a la cual la reconstrucción puede ser obtenida se vea limitada y es lo que ha impedido su uso rutinario en aplicaciones clínicas en tiempo real. Existen, por tanto, dos vías por las que afrontar esta situación: disminuir la complejidad computacional global por medio de las técnicas algorítmicas y/o utilizar hardware especializado de alto rendimiento.

Hay cierto número de implementaciones en hardware de estos algoritmos disponibles comercialmente, bien usando circuitos integrados de aplicación específica (ASIC) o bien mediante FPGAs (***F**ield **P**rogrammable **G**ate **A**rrays*). Este enfoque, sin embargo, sufre de una serie de importantes desventajas, entre las que se encuentran su elevado coste y su falta de flexibilidad (el hecho de que el algoritmo esté ya implementado en hardware hace muy difícil modificarlo o generalizarlo, y supone un obstáculo para desarrollos posteriores). Así pues, es deseable buscar otra solución si pretendemos que su uso sea práctico, tanto para investigadores como para clínicos.

Aunque hoy en día las nuevas CPUs han alcanzado el suficiente poder para ser competitivas en el terreno de la reconstrucción bidimensional, no se puede afirmar lo mismo en lo referente a las reconstrucciones tridimensionales. Afortunadamente, existe una solución intermedia entre una implementación software en CPU y el uso de hardware específico: el empleo de hardware gráfico programable. Trataremos de explicar brevemente a continuación por qué esta idea es aplicable a nuestro problema.

Los procesadores gráficos (***G**raphical **P**rocessing **U**nits*, GPUs) son procesadores diseñados para el renderizado de gráficos, con el objetivo de realizar la mayoría de las operaciones necesarias en hardware, aligerando así la carga de trabajo del procesador central. Esta es una funcionalidad en principio muy específica, pero con el tiempo los fabricantes han ido añadiendo posibilidades de programación con el fin de permitir ciertos efectos especiales, principal-

mente motivados por la industria de los videojuegos. Actualmente se pueden considerar procesadores totalmente programables, si bien con características de rendimiento significativamente distintas a las de las CPUs (por lo general los mecanismos de caching resultan menos efectivos, y carecen de técnicas de predicción de saltos o ejecución especulativa), pero con una ventaja muy importante para nosotros: dado que el proceso de renderizado tiene un alto grado de paralelismo, las GPUs ofrecen una gran poder de cómputo para aplicaciones paralelas y computacionalmente intensivas.

Igualmente importante es el hecho de que los procesadores gráficos programables ofrecen una relación entre precio y capacidad de cómputo inigualada; actualmente es posible adquirir GPUs muy potentes para uso doméstico en ordenadores personales.

Todo esto ha impulsado enormemente el desarrollo de las técnicas conocidas como GPGPU (***G****eneral* ***P****urpose Computation on* ***G****raphical* ***P****rocessing* ***U****nits*, o computación de propósito general en GPUs) durante los últimos años. Estos métodos han sido aplicados con éxito en una gran variedad de campos (procesamiento de imágenes y señales, visión por computador, codificación/decodificación de vídeo, reconocimiento de voz...), y son especialmente exitosos en aquellas aplicaciones que se encajen dentro del modelo de *Stream Processing* (que detallaremos más adelante, en la sección 3.3.1), y al que los algoritmos iterativos de reconstrución se ajustan razonablemente bien.

Por tanto, el objetivo de nuestro proyecto es conseguir una aceleración significativa del proceso de reconstrucción tomográfica, sin sacrificar por ello la precisión obtenida, mediante técnicas de GPGPU. En particular, nos centraremos en las variantes del método conocido como ***A****lgebraic* ***R****econstruction* ***T****echnique* (ART), que actúa de forma iterativa, partiendo de una estimación inicial del volumen y procediendo a corregirlo de acuerdo a las diferencias entre las proyecciones que se calcularían con esa estimación y las proyecciones medidas. Como ya hemos mencionado, el programa de reconstrucción de Xmipp es nuestra principal referencia con la que comparar resultados y tiempos de

ejecución.

La formulación matemática del problema de tomografía, así como los distintos métodos de reconstrucción algebraica y las técnicas relativas a ellos serán el objeto del segundo capítulo.

En el tercer capítulo se profundiza en el interior de un unidad de procesamiento gráfico, detallando la estructura del *pipeline* gráfico y su funcionamiento y además, analizaremos de manera general la arquitectura utilizada por nosotros (***C****ompute* ***U****nified* ***D****evice* ***A****rchitecture*, CUDA), introducida por NVIDIA para la familia de tarjetas GeForce 8800.

En los capítulos cuarto y quinto se describen las implementaciones llevadas a cabo (en CPU y en GPU, respectivamente), analizando sus ventajas e inconvenientes; y en el sexto capítulo se detallan los resultados obtenidos de manera experimental teniendo en cuenta las pruebas realizadas.

Entre las conclusiones detalladas en el séptimo, y último, capítulo, cabe destacar que nuestra implementation en GPU consigue mejoras de un orden de magnitud en los tiempos de reconstrucción con respecto a las implementaciones existentes en CPU, con los mismos casos de prueba y, lo que es igualmente importante, con una calidad de reconstrucción que no tiene nada que envidiar a la de otros paquetes de software desarrollados con la misma finalidad.

Capítulo 2

Los algoritmos de reconstrucción algebraica

2.1. Introducción a la tomografía

Como ya hemos comentado en el capítulo anterior, existen principalmente dos tipos de algoritmos de reconstrucción: los que trabajan en el dominio de Fourier, relacionando los datos de proyección medidos con la transformada de Fourier bidimensional de sus secciones; y los métodos algebraicos, que consisten en asumir que las secciones del cuerpo forman una matriz de incógnitas, y construir y resolver el sistema de ecuaciones asociado, repitiendo el proceso si es necesario hasta conseguir que la reconstrucción encaje con todas las proyecciones. Los primeros son los más utilizados actualmente en aplicaciones médicas (donde predomina la FBP, *Filtered Backprojection* o retroproyección filtrada), pero para ser aplicados con la exactitud requerida necesitan una distribución uniforme de los ángulos de proyección y una gran cantidad de proyecciones, lo que en muchas situaciones no es posible. Cuando esto ocurre (por ejemplo, cuando faltan proyecciones en algunas orientaciones), los métodos algebraicos son la mejor elección. En este trabajo describimos con detalle únicamente los métodos algebraicos, poniendo énfasis en su aplicación en mi-

croscopía electrónica [1]; el lector interesado puede encontrar información sobre los métodos basados en transformaciones en [KS88].

2.2. Descripción del problema

Vamos a describir con más precisión el problema que queremos resolver. El primer paso es representar el volumen a reconstruir como una función real $f : \mathbb{R}^3 \to \mathbb{R}$ definida en el espacio tridimensional, que se pretende determinar a partir de sus proyecciones bidimensionales. Esta función representa la intensidad de cierta magnitud física dependiente del proceso concreto utilizado para la obtención de las imágenes; por ejemplo, en el caso de la tomografía de rayos X, f representa unos coeficientes de atenuación (opacidad a los rayos X); en el caso de la microscopía electrónica representa el potencial electrostático. En cualquier caso, f está relacionada de manera directa con la distribución de masa en el espacio del objeto considerado. Nuestro objetivo es reconstruir los valores de esta función con la mayor precisión posible. Como es obvio, en la práctica debemos obtener una representación finita de la función que la aproxime adecuadamente.

En el proceso de obtención de imágenes tomográficas, se parte de una fuente desde la que se lanzan múltiples rayos (ver figura 2.1). No nos preocuparemos de momento de la geometría exacta de la distribución de estos rayos; los dos casos más frecuentemente considerados son el uso de proyección de rayos paralelos (el caso más común en microscopía electrónica, y el que ha sido considerado en este trabajo) o cónicos (todos partiendo de un mismo punto). De todas formas la elección concreta no afecta a la discusión que sigue en este apartado, por lo que el lector puede imaginar una serie de rayos paralelos si así lo desea.

Asumiremos, como es usual, que los rayos viajan en línea recta. Para el

[1] El lector que desee comprender con algo de detalle la microscopía electrónica, que constituye la principal aplicación del trabajo, puede consultar el apéndice C.

uso de los métodos algebraicos es esencial conocer la trayectoria exacta de los rayos; en otras condiciones, los resultados normalmente carecerán de significado. (Sin embargo, si los efectos de refracción y difracción existen pero son suficientemente pequeños, es posible [AK84] combinar estos métodos con el uso de técnicas digitales de seguimiento de rayos o *ray-tracing*).

Al llegar los rayos al receptor, la imagen resultante está en función de la "Transformada de rayos X" del objeto, que no es más que la integral de línea de la función f a lo largo de la trayectoria que ha atravesado el rayo. (Estamos asumiendo que los rayos tienen grosor nulo; en caso contrario, tendríamos que tomar realmente la integral de volumen de todo el rayo).

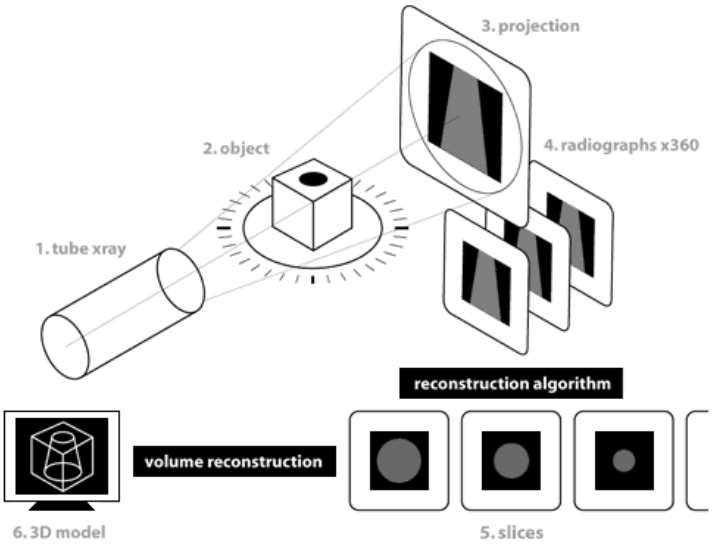

FIGURA 2.1: Obtención de imágenes tomográficas

Por ejemplo, en el caso de los rayos X, la intensidad recolectada en un punto decrece exponencialmente según la integral de línea:

$$I = I_0 \exp\left(-\int_{-\infty}^{\infty} f(p_0 + t \cdot \vec{v})\, \mathrm{d}t\right)$$

(donde p_0 es el origen del rayo y $\vec{v}$ un vector unitario que indica su dirección), pero lo importante es que la relación entre las intensidades medidas y

las originales están en función de estas integrales, en cualquiera de los métodos utilizados. Esta relación es además invertible, por lo que después de cierto preprocesamiento sobre los datos medidos podemos considerar que los datos que tenemos a nuestra disposición son directamente las transformadas de rayos X del volumen. Nótese que, al contrario que en una imagen fotográfica de un objeto, las imágenes resultantes dependen de su interior y no sólo de su frontera.

Naturalmente, una sola proyección no basta para definir la forma de un objeto tridimensional, por lo que es necesario tomar cierto número de imágenes desde distintas direcciones; el objetivo es encontrar una función f que concuerde con todas ellas (y, a ser posible, que la solución sea única, de tal manera que se pueda asegurar que los datos obtenidos son suficientes para una reconstrucción inequívoca). Claramente esto no es posible en la práctica, incluso aunque supusiéramos ausencia de ruido, pero existe un teorema muy importante en tomografía que nos dice qué información podemos obtener a partir de una proyección.

Este resultado se denomina "Teorema de la sección central", y es la base en la que se fundamentan la mayoría de los métodos basados en transformaciones. Establece, en el caso que nos interesa, que la transformada de Fourier de la proyección de un volumen es igual a una sección bidimensional de la transformada de Fourier del volumen entero, tomada según la misma dirección. Esto nos da una condición necesaria y suficiente para conseguir unicidad en la solución en ausencia de ruido: las proyecciones deben ser tales que cubran completamente el espacio de Fourier asociado al volumen. Como es obvio, esto no se puede conseguir con un número finito de proyecciones, pero si se toma un número suficientemente grande que cubra apropiadamente bien el espacio se pueden interpolar los valores a todo el espacio tridimensional con un cierto grado de exactitud. Obsérvese que los métodos basados en una inversión directa de las transformadas de Fourier de las proyecciones necesitan por tanto cubrir la mayor cantidad posible de ángulos de manera uniforme, lo que como ya hemos comentado no siempre es posible y supone una limitación a su apli-

cación.

En cuanto a los ángulos de proyección, dependen directamente del proceso con el que se han obtenido las imágenes, por lo que podría parecer que deben ser conocidos. En realidad, en el campo de la microscopía electrónica esto no es así de sencillo, puesto que las imágenes tomadas pueden corresponder a moléculas distintas, por lo que se necesita realizar otro proceso iterativo en el que se parte de cierta información relativa a los ángulos, se reconstruye el objeto suponiendo que esta información es exacta, y se ajusta de nuevo la estimación angular. En este trabajo nos ocupamos únicamente de la reconstrucción asumiendo que los ángulos son conocidos con exactitud, que es la parte integral del proceso; una posible ampliación (véase el capítulo 7) consistiría en llevar a cabo el proceso entero en la GPU.

La forma usual de especificar la posición y dirección desde las que ha sido obtenida una imagen es especificar una matriz de rotación que transforme la dirección del rayo en el eje Z, y el plano de la imagen (perpendicular a él) en el XY; de este modo el proceso de proyección corresponde simplemente a una proyección vertical (esto es, desechar la tercera coordenada cartesiana). Tradicionalmente, en microscopía electrónica se ha utilizad la representación basada en ángulos de Euler. El origen del sistema de coordenadas se sitúa en el centro del volumen a reconstruir, y se especifica una rotación alrededor de un eje cualquiera que pase por el origen con una terna de ángulos (α, β, γ). Su significado, ilustrado en la figura 2.2, es el siguiente:

- α (ángulo acimutal): indica la rotación realizada en torno al eje Z
- β (ángulo de inclinación): rotación respecto al nuevo eje Y
- γ (ángulo de rotación en el plano): rotación respecto al último Z

Con estas convenciones, la matriz de rotación resultante, dada por el pro-

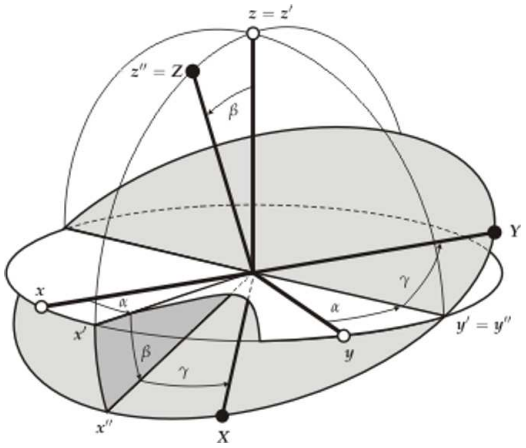

FIGURA 2.2: Ángulos de Euler

ducto de las tres matrices de rotación correspondientes, es

$$\begin{pmatrix} cos(\gamma)cos(\beta)cos(\alpha) - sin(\gamma)sin(\alpha) & cos(\gamma)cos(\beta)sin(\alpha) + sin(\gamma)cos(\beta) & -cos(\gamma)sin(\beta) \\ -sin(\gamma)cos(\beta)cos(\alpha) - cos(\gamma)sin(\beta) & -sin(\gamma)cos(\beta)sin(\alpha) + cos(\alpha)cos(\beta) & sin(\gamma)sin(\beta) \\ sin(\beta)cos(\alpha) & sin(\beta)sin(\alpha) & cos(\beta) \end{pmatrix}$$

En resumen, los datos de entrada para nuestro proceso consisten en una serie de proyecciones que contienen las transformadas de rayos X del objeto, convenientemente normalizadas (con el objetivo de homogeneizar los niveles de densidad de gris de imágenes correspondientes a distintas micrografías), junto con la información de los ángulos de Euler asociados a cada una de ellas; y la salida es una representación de la función f que se pretendía determinar. Cómo exactamente será esta representación, y cuál es el funcionamiento de los métodos algebraicos, es el objeto de las siguientes secciones.

2.3. Representación del volumen

Las técnicas de reconstrucción algebraica forman una familia de algoritmos que utilizan un proceso iterativo de resolución de sistemas de ecuaciones lineales para obtener la reconstrucción buscada. Para describirlos, empezamos definiendo una rejilla de "funciones base" que nos permitan aproximar el vo-

lumen con suficiente precisión, de tal manera que la función que buscamos (la función f comentada en la sección anterior) la podamos escribir, aproximadamente, como una suma finita ponderada de funciones base; de esta manera la salida de nuestro algoritmo será simplemente los coeficientes de esta suma ponderada:

$$f(x) = \sum_{i=1}^{N} f_i \cdot g_i(x)$$

donde $g_i, i \in 1 \dots N$ son las funciones base, cuyo soporte está típicamente limitado a una zona reducida del espacio y son normalmente translaciones unas de otras.

Consideremos que el volumen está dividido en una rejilla compuesta de celdas formadas por cubos de lado unidad (que denominaremos vóxeles, ya que se trata del análogo tridimensional a los píxeles de una imagen). El modelo más simple se basa en suponer que f toma un valor constante en cada una de las celdas; en este caso, podemos usar una función base para cada celda, de tal manera que la función asociada a la celda i tiene valor 1 dentro de ella y fuera en el exterior. Llamaremos vóxeles también a estas funciones base (a lo largo de este trabajo, el contexto dejará claro si nos estamos refiriendo a las celdas en sí o a las funciones base asociadas a ellas). De esta manera, la ecuación anterior se interpreta como que f es constante dentro del vóxel (celda) i, y toma el valor f_i (ver figura 2.3, en la que se muestra el caso análogo bidimensional). Claramente, el uso de un número mayor de vóxeles supondrá una mejor resolución en el resultado obtenido.

A lo largo de esta explicación usaremos vóxeles cuyo lado es el mismo que la distancia entre dos píxeles adyacentes a la imagen. Por su propia naturaleza, el uso de vóxeles como funciones base impide obtener volúmenes con bordes suaves; en la sección 2.6.1.1 examinaremos otras posibilidades, como los *blobs*, que intentan solucionar este problema. Notemos, sin embargo, que la elección de las funciones base no influye en gran medida sobre el algoritmo de reconstrucción; sólo afecta a la manera en que se calculan ciertos coeficientes (las

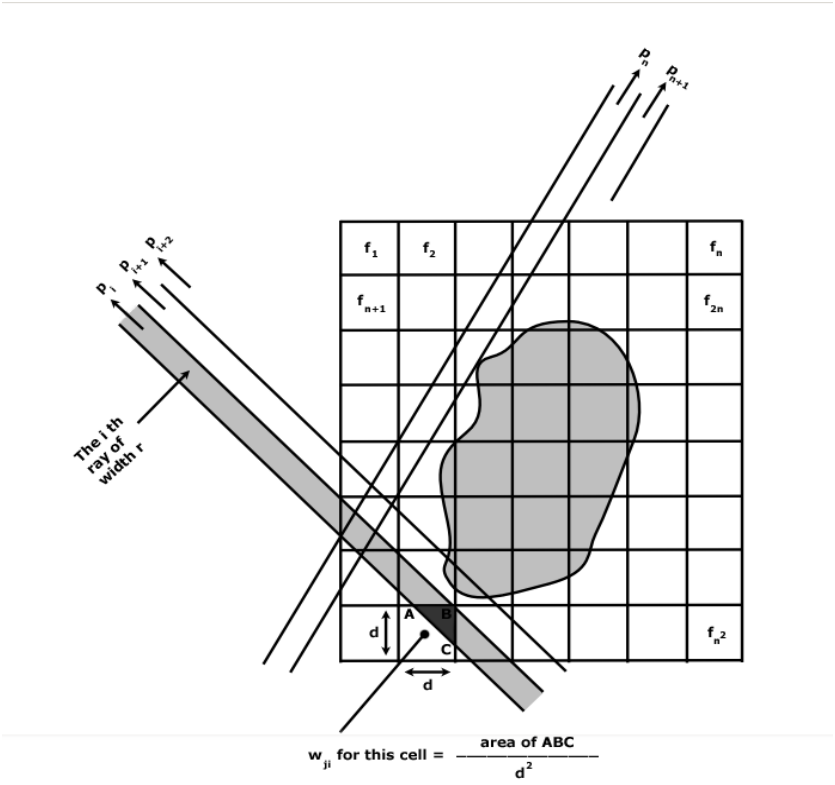

FIGURA 2.3: Representación de la proyección. Fuente: [KS88]

integrales de línea de cada una de las funciones base para cada rayo).

En el proyecto ha sido considerada la utilización tanto de vóxeles como de blobs. En cualquiera de los casos, las funciones base son copias idénticas unas de otras, con la única diferencia de que están desplazadas en el espacio. Los *blobs*, sin embargo, gozan de otra característica deseable de la que carecen los vóxeles, que es su simetría esférica. Este hecho, unido a la observación anterior, permite el precálculo de las integrales de línea de un rayo en función únicamente de su distancia al centro de la celda.

2.4. Formalización del problema

La característica principal del ART es que, una vez definida la rejilla en que se divide el volumen y las funciones base que vamos a utilizar, se plantea un sistema de ecuaciones lineales cuyas incógnitas son los coeficientes asociados a cada celda (el peso de su función base asociada), y cada ecuación representa la medición experimental de la integral de línea de uno de los rayos lanzados. La solución a este sistema será la modelización del objeto reconstruido.

De lo que hemos dicho hasta ahora se desprende que cada celda contribuye al valor de un píxel con una cantidad igual a la integral de la función base a lo largo del rayo. Para el caso de un vóxel, en el que la función es constante dentro de la celda y nula fuera de él, esta integral coincide con la longitud del segmento intersección entre el rayo y el vóxel, multiplicada por el coeficiente correspondiente a esa celda.

Para encontrar el sistema de ecuaciones a resolver, consideremos un volumen formado por N vóxeles y un conjunto de proyecciones formadas por M rayos (que identificaremos con sus píxeles asociados). Establecemos por tanto la siguiente nomenclatura:

- w_{ij} $\quad i \in 1..N, j \in 1..M$: integral de línea de la porción del rayo i-ésimo que corta al vóxel número j,
- f_j $\quad j \in 1..N$: peso de la función base asociada al vóxel j-ésimo.
- p_i $\quad i \in 1..M$: integral del rayo i-ésimo, es decir, el valor obtenido en el píxel i

Entonces la relación que conecta el volumen y las proyecciones obtenidas se resume en:

$$\sum_{j=1}^{N} w_{ij} f_j = p_i \quad i \in 1..M \tag{2.1}$$

Como se puede ver, tenemos una ecuación así para cada rayo, con lo cual

nuestro sistema completo será de la forma

$$\begin{array}{ccccccccccc} w_{11}f_1 & + & w_{12}f_2 & + & w_{13}f_3 & + & \ldots & + & w_{1N}f_N = p_1 \\ w_{21}f_1 & + & w_{22}f_2 & + & w_{23}f_3 & + & \ldots & + & w_{2N}f_N = p_2 \\ w_{31}f_1 & + & w_{32}f_2 & + & w_{33}f_3 & + & \ldots & + & w_{3N}f_N = p_3 \\ \vdots & & & & & & & & \\ w_{M1}f_1 & + & w_{M2}f_2 & + & w_{M3}f_3 & + & \ldots & + & w_{MN}f_N = p_M \end{array} \tag{2.2}$$

que es un sistema lineal de M ecuaciones en N incógnitas. Es importante observar, sin embargo, que la matriz del sistema es muy dispersa, puesto que un rayo sólo puede atravesar un cierto número de celdas del volumen; así pues la gran mayoría de los w_{ij} son cero [2].

Resolver este sistema es sencillo en teoría, pero las grandes dimensiones que tiene hacen impracticable una resolución basada en métodos matriciales convencionales como la eliminación gaussiana. Por ejemplo, para reconstruir un volumen de $64 \times 64 \times 64$ usando 250 proyecciones de 64×64, el sistema de resultante tendría $M = 1024000$ ecuaciones en $N = 262144$ incógnitas. Además, este problema de dimensiones se ve agravado cuando con el aumento del tamaño del volumen a reconstruir, ya que crece proporcionalmente al número de celdas que consideremos (que a su vez es proporcional al cubo de la longitud de un lado del volumen).

Para tratar este problema, los métodos de reconstrucción algebraica adoptan un enfoque iterativo para aproximar la solución del sistema. Nos ocuparemos a continuación de las bases de este enfoque (el método Kaczmarz), así como de los representantes más importantes de esta familia de métodos.

[2] Intuitivamente, el número máximo de vóxeles que puede atravesar un rayo debería variar de manera lineal con la suma de los lados del volumen y no con el número total de celdas en el volumen. Como veremos en el capítulo 4, esto es cierto y además es de gran importancia de cara a la realización de una implementación eficiente.

2.5. El método Kaczmarz

El método Kaczmarz, también llamado método de las proyecciones, es un algoritmo iterativo, propuesto en 1937 por Kaczmarz, para resolver sistemas de ecuaciones lineales como el anterior. Sus características lo hacen especialmente adecuado para nuestras necesidades, debido a sus propiedades de convergencia y a que, como veremos, se puede escribir como un proceso de proyección de un volumen / corrección que se adapta muy naturalmente a nuestro problema.

El algoritmo está basado en considerar que la solución a las ecuaciones 2.2 está dada por un punto en el espacio euclídeo N-dimensional. Cada una de las ecuaciones corresponde por tanto a un hiperplano en $\mathbb{R}^N$, siendo w_i su vector normal. La técnica para resolver el sistema consiste en empezar con una estimación inicial de la solución y proceder a tomar una serie de proyecciones ortogonales sucesivas sobre cada uno de los hiperplanos; cuando se ha realizado una iteración sobre todos ellos, se puede repetir el proceso si es necesario, hasta que se satisfaga algún criterio de convergencia fijado de antemano. Para permitir una mejor visualización de su forma de operar, mostraremos un sencillo ejemplo de dos ecuaciones en dos variables (linealmente independientes, por lo que la solución es única) donde se podrá observar gráficamente el proceso:

$$\begin{aligned} w_{11}f_1 + w_{12}f_2 &= p_1 \\ w_{21}f_1 + w_{22}f_2 &= p_2 \end{aligned} \tag{2.3}$$

Este sistema define dos rectas en el plano, y su solución se halla en la intersección de las mismas.

Se comienza con el establecimiento de una estimación inicial de la solución s_0, que normalmente se toma nula (es decir, se empieza en el origen), aunque si se posee alguna información sobre la naturaleza de la solución se puede generar otra que creamos más cercana a la solución real. Esta solución s_0 se proyecta sobre una de las dos rectas, obteniendo una nueva estimación s_1, que se proyecta sobre la otra recta consiguiendo una tercera estimación s_2. Si el sistema

tiene solución única, ésta está en la intersección de todos los hiperplanos, y repitiendo este proceso de proyecciones sobre las rectas obtenemos una sucesión de puntos que converge a ella (la demostración de la convergencia queda fuera del alcance de este documento, pero puede encontrarse en [Tan71]). Podemos ver la aplicación del método para nuestro pequeño sistema en la figura 2.4.

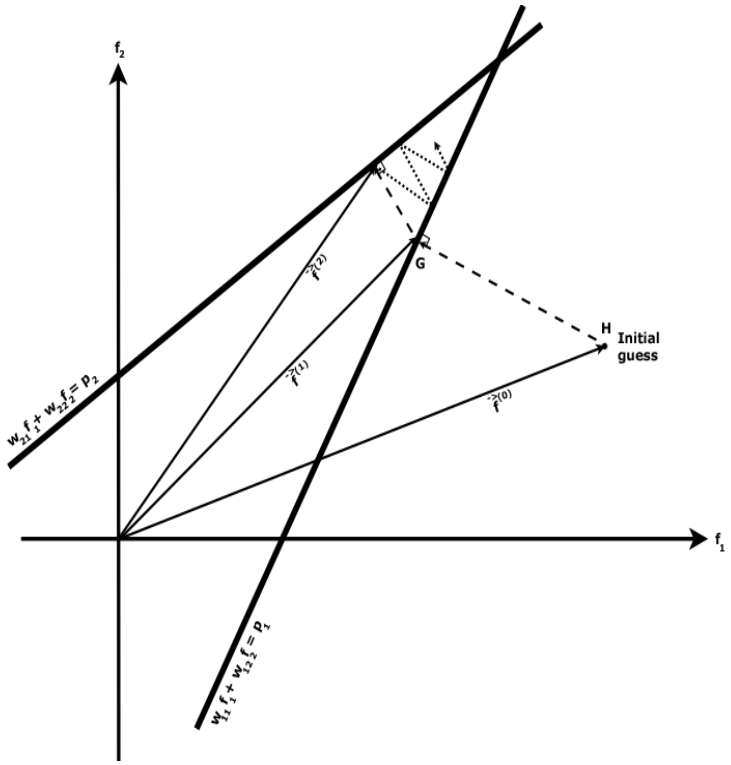

FIGURA 2.4: Representación gráfica del método Kaczmarz. Fuente: [KS88]

Una vez descrito el método general, pasemos a estudiar los factores que afectan a la convergencia del algoritmo. Aunque en general no se obtendrá la solución exacta al cabo de un número finito de proyecciones, la aplicación de un número suficientemente elevado de iteraciones nos lleva a una aproximación a la solución bastante fiel. Existe un caso, sin embargo, en el que la solución se alcanza al cabo de un número finito de pasos, y es importante darse cuenta de ello para estudiar las propiedades de convergencia del algoritmo. Imaginemos que las dos rectas de nuestro ejemplo fueran perpendiculares. Es claro

que entonces alcanzaríamos la solución exacta después de haber realizado sólo dos proyecciones, y es fácil ver que en general si partimos de n hiperplanos ortogonales en un espacio n-dimensional, encontraremos la solución después de haber procesado todos ellos. Por otro lado, si las dos rectas fueran casi paralelas, se necesitaría un gran número de iteraciones para llegar a una aproximación satisfactoria.

Por tanto, la convergencia del método depende en gran medida de los ángulos que formen los hiperplanos. En teoría sería posible realizar una ortogonalización previa (usando por ejemplo el conocido método de Gram-Schmidt), pero tal procedimiento, aparte de no ser computacionalmente factible (tiene complejidad $O(N^3)$, la misma que el método de Gauss que decidimos evitar), tiende a potenciar el efecto del siempre presente ruido. Por todo esto, la opción más práctica, disponible en varias implementaciones (incluida la proporcionada en Xmipp y la nuestra), es ordenar las imágenes antes de tratarlas para intentar evitar que proyecciones con información muy parecida (tomadas desde direcciones muy cercanas) sean tratadas consecutivamente, pues esto supondría proyectar sobre dos hiperplanos casi paralelos.

En cuanto a la presencia de ruido, es inevitable en escenarios de reconstrucción reales, bien sea por errores en la estimación de los ángulos, en la normalización de las imágenes, o en las propias mediciones. Esto, unido al hecho de que los sistemas que se suelen utilizar en la práctica están sobredeterminados (hay más ecuaciones linealmente independientes que incógnitas), implica que el sistema será en realidad inconsistente. En estos casos el método Kaczmarz no converge, sino que se alcanza un patrón casi periódico de oscilaciones torno a una vecindad donde estaría la solución real sin ruido. Si la magnitud del ruido no es excesiva, esta vecindad es aceptable y nos lleva a buenas reconstrucciones, pero cuando el ruido es alto los resultados de la proyección son impredecibles. Un ejemplo de esta oscilación en torno a la solución puede observarse en la figura 2.5.

Queda por considerar el caso de que el sistema sea compatible indeter-

minado. En nuestro ejemplo bidimensional, esto correspondería a que las dos rectas fueran de hecho la misma. Entonces todos los puntos de la recta serían soluciones válidas, y al aplicar el método simplemente proyectaríamos una vez y nos quedaríamos con ese resultado, obteniendo por tanto la solución más cercana al punto de partida. También se puede demostrar [Tan71] que este resultado es cierto en general: el algoritmo converge a la solución más cercana a la estimación inicial (en el sentido de la norma euclídea n-dimensional, que coincide con la que se obtiene al tomar el error cuadrático medio).

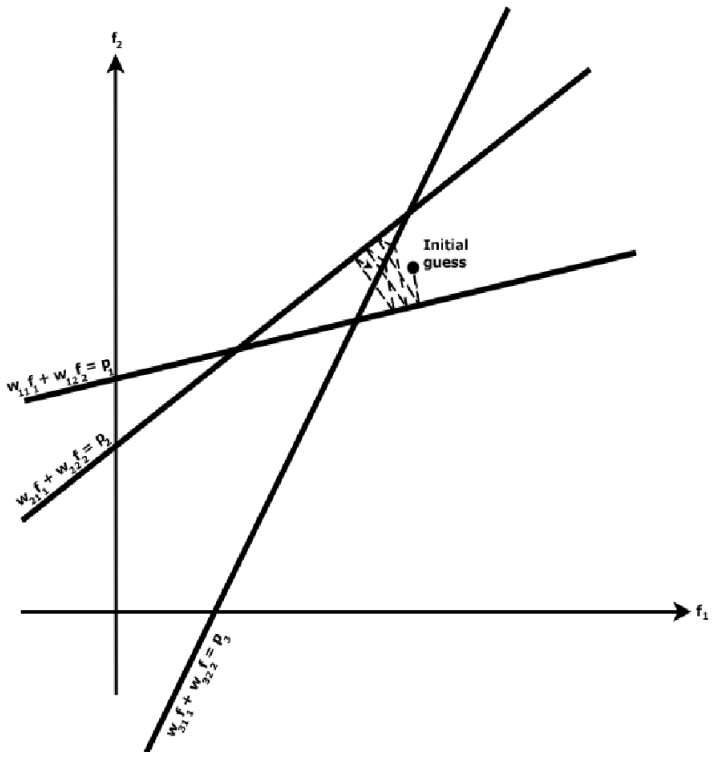

FIGURA 2.5: Oscilación en el método Kaczmarz. Fuente: [KS88]

Es sencillo obtener la expresión matemática correspondiente al proceso. Comenzamos con una solución inicial $f_1^{(0)}, f_2^{(0)}, f_3^{(0)}, \ldots f_N^{(0)}$ que vectorialmente denotaremos $\vec{f}^{(0)}$ (donde el superíndice denota el número de proyecciones ya hechas). Esta solución inicial es proyectada sobre el hiperplano que define la primera ecuación de (2.2), llegando a $\vec{f}^{(1)}$. $\vec{f}^{(1)}$ es entonces proyectada sobre

el hiperplano definido por la segunda ecuación de (2.2), resultado $\vec{f}^{(2)}$, y así sucesivamente. Como $\frac{w_i}{\langle w_i, w_i \rangle}$ es el vector unitario perpendicular al i-ésimo hiperplano, tenemos que

$$\vec{f}^{(i)} = \vec{f}^{(i-1)} + \lambda \frac{w_i}{\langle w_i, w_i \rangle}$$

donde λ es la distancia de $\vec{f}^{(i-1)}$ a su proyección ortogonal sobre el hiperplano (que será $\vec{f}^{(i)}$). Dado que $\vec{f}^{(i)}$ tiene que satisfacer la ecuación del hiperplano

$$\langle \vec{f}^{(i)}, w_i \rangle = p_i \tag{2.4}$$

sustituyendo la ecuación anterior en esta relación obtenemos

$$\lambda = p_i - \langle \vec{f}^{(i-1)}, w_i \rangle$$

por lo que finalmente resulta

$$\vec{f}^{(i)} = \vec{f}^{(i-1)} - \frac{\langle \vec{f}^{(i-1)}, w_i \rangle - p_i}{\langle w_i, w_i \rangle} \cdot w_i \tag{2.5}$$

Por último, notaremos que una implementación ingenua de esta ecuación llevaría a un algoritmo de complejidad en tiempo $\mathrm{O}(NM)$, que es demasiado lento en la práctica. Sin embargo, esta complejidad puede reducirse considerablemente si tenemos en cuenta, como ya observamos, que la mayor parte de los w_{ij} son cero. Comentaremos esto con más detalle en el capítulo de implementación (sección 4.4).

2.5.1. Fases del método. Proyección y backproyección

Existe otra manera de interpretar la ecuación (2.5), más ilustrativa para nuestros propósitos. Se puede ver que para calcular la estimación i-ésima se ha de aplicar una corrección a la solución que se tenía; el valor de la corrección depende a su vez de la diferencia entre la proyección teórica (dada por el producto escalar del numerador) y la proyección medida (p_i). Se puede escribir (2.5) de una forma algo diferente para poner énfasis en la variación que sufre

cada vóxel:

$$f_j^{(i)} = f_j^{(i-1)} + \frac{p_i - q_i}{\sum_{k=1}^{N} w_{ik}^2} w_{ij} \quad \forall j \in 1 \dots M \tag{2.6}$$

donde

$$q_i = \langle \vec{f}^{(i-1)}, w_i \rangle = \sum_{k=1}^{M} f_k^{(i-1)} w_{ik} \tag{2.7}$$

es decir, q_i es la integral de línea del rayo i-ésimo atravesando el volumen actual y p_i la integral de ese mismo rayo atravesando el volumen real; se toma la diferencia y se distribuye en la estimación del volumen según un factor de normalización dado por $\frac{w_{ij}}{\sum_{k=1}^{N} w_{ik}^2}$.

Visto de esta forma, el método puede separarse en tres fases:

1. Proyección: calcula q_i, la suma que daría el rayo i-ésimo al atravesar el volumen actual

2. Calcular la corrección: se halla la diferencia entre la suma del rayo real y la suma del rayo actual, y se normaliza esa diferencia.

3. Backproyección: se aplica dicha corrección a cada uno de los vóxeles para obtener $f(i)$.

Esta división del método en tres fases refleja la naturaleza del proceso (al fin y al cabo las imágenes medidas no son más que proyecciones, y lo que hacemos es compararlas con las que obtendríamos si el volumen concidiera con nuestra estimación, y corregir en función del resultado), y además nos será muy útil posteriormente, cuando hablemos más detalladamente de la implementación del algoritmo.

En cuanto a los costes computacionales, baste decir por ahora tanto la proyección como la backproyección tienen complejidad $\mathrm{O}(n^3)$ en tiempo para una imagen si tenemos proyecciones de n^2 rayos y un volumen de $n \times n \times n$; en la práctica, sin embargo, alrededor del 90 % del tiempo de ejecución se debe

al proceso de proyección (porque es el que tiene que calcular los rayos que intersecan a cada vóxel y los coeficientes; esta información ya está almacenada para el paso de backproyección). Esto es cierto tanto para las implementaciones en CPU como para la nuestra en GPU.

2.6. ART

Algebraic Reconstruction Techniques (ART) es el nombre dado tanto a un algoritmo de reconstrucción como a toda una familia de variantes suyas, sobre algunas de las cuales nos ocuparemos en las secciones siguientes. El algoritmo básico del ART es simplemente el método Kaczmarz, ya descrito, aumentado con un factor de relajación, cuyo uso sirve para hacerlo más tolerante al ruido y que describiremos en breve. Antes, sin embargo, examinaremos algunas de las facilidades que permiten los métodos algebraicos en general (y de ART en particular) de las que carecen los métodos basados en transformaciones. Estas derivan del hecho de que estos algoritmos permiten incluir fácilmente restricciones especiales, como podrían ser por ejemplo:

1. Información sobre el soporte o la forma del objeto (la región de espacio no vacía): sólo actualizamos aquellos valores dentro del soporte, reduciendo así el número de variables. Esta reducción de variables podría también incrementarse si, por ejemplo, supiéramos con seguridad que el objeto tiene alguna simetría.

2. Información sobre la función a reconstruir: si sabemos por ejemplo que sólo toma valores no negativos, podemos forzar esta restricción poniendo a 0 todos los valores que, una vez calculados, resulten ser negativos.

3. Información sobre el volumen que ocupa el objeto (o la masa, si conocemos su densidad media): si sabemos que el volumen total no puede ser mayor que V, en cada paso ponemos a cero todos los valores calculados que no estén entre los V mayores.

Además, como ya hemos visto, el método converge a la solución que más se acerque a la estimación inicial, por lo que podemos sacar partido de esto

empezando con una estimación no nula, de acuerdo a la información de que dispongamos.

2.6.1. Factores del modelo que afectan a la calidad de la reconstrucción

Antes de terminar esta introducción al método ART en general, dedicaremos unas líneas a introducir algunos factores que pueden afectar a la calidad de las reconstrucciones. Estos son principalmente dos: la distribución de densidad dentro de los vóxeles y el establecimiento de un factor de relajación para las correcciones.

En los algoritmos de reconstrucción basados en ART es normal que estos dos factores sean parámetros definidos por el usuario. Como es el usuario quien puede tener acceso a información sobre la naturaleza del objeto a reconstruir y las características (tamaño, nivel de ruido...) de las proyecciones, se le ofrece la posibilidad de ajustarlos y realizar pruebas con diversas combinaciones para encontrar las que satisfagan sus necesidades específicas.

2.6.1.1. Funciones base

Hasta ahora hemos considerado, como aproximación, que la distribución de la función a determinar es constante dentro de los vóxeles, y ocupa todo el volumen de éste. Por eso para calcular la integral del rayo i-ésimo en el vóxel j-ésimo decíamos que solo dependía de la distancia que dicho rayo recorría dentro de ese vóxel. Aunque este modelo de densidad da lugar a buenas reconstrucciones y no es muy complicado de calcular (aparte de ser el más utilizado por los paquetes de software que trabajan con volúmenes) existen infinidad de opciones.

En algunas implementaciones simplemente se sustituyen los pesos (integrales de un rayo en un vóxel) por 0 ó 1, (dependiendo de si el rayo corta al vóxel dado o no), o por alguna otra aproximación que trate alcanzar un buen

compromiso entre calidad de la reconstrucción y velocidad de cálculo de la integral de rayo. Esto facilita los cómputos, pero da lugar a reconstrucciones con una gran cantidad de lo que se conoce como *"salt and pepper noise"*, que surge de las inconsistencias que se introducen en el sistema de ecuaciones al sustituir los w_{ij} reales por aproximaciones no demasiado buenas. La introducción de un factor de relajación (que comentaremos en la siguiente sección) ayuda a eliminar este ruido y a producir reconstrucciones más suaves.

Un modelo más avanzado trata de obtener volúmenes de bordes suaves (lo cual se dificulta con el uso de vóxeles, puesto que por su propia definición son discontinuas en los bordes). Entre las soluciones propuestas, están las funciones de Kaisser-Bessel modificadas, comúnmente conocidas como *blobs* [MHC98]. El valor de estas funciones a una distancia r del centro del *blob* está dado por:

$$b_n^m(r) = \begin{cases} \dfrac{\left(\sqrt{1-\left(\frac{r}{a}\right)^2}\right)^m I_m\left(\alpha\sqrt{1-\left(\frac{r}{a}\right)^2}\right)}{I_m(\alpha)} & \text{si } 0 \leq r \leq a \\ 0 & \text{en otro caso} \end{cases}$$

donde a es el radio del *blob*, α es un factor de suavizado, m es el orden de la función de Bessel (el *blob* resulta ser derivable hasta orden m), n es la dimensión del espacio en que se define (en nuestro caso $n = 3$ puesto que nos encontramos en $\mathbb{R}^3$) e $I_m(x)$ es la función de Bessel de primera especie de orden m. El nombre de Kaisser se debe a que la función de Bessel se encuentra multiplicada por una ventana de Kaisser que la limita en el espacio. La figura 2.6 representa el corte de algunos *blobs* representativos, con los parámetros más utilizados.

El uso de *blobs* tiene además la ventaja añadida de que presentan simetría esférica, por lo que las integrales de línea sólo dependen de la distancia al centro del *blob* y pueden ser precalculadas en una tabla en la que se use esta distancia como índice, ahorrando tiempo de cálculo durante la proyección (tal enfoque no es posible con el uso de vóxeles). En particular, en el caso de las funciones de Bessel de orden 2 ($m = 2$), que son las utilizadas en nuestra

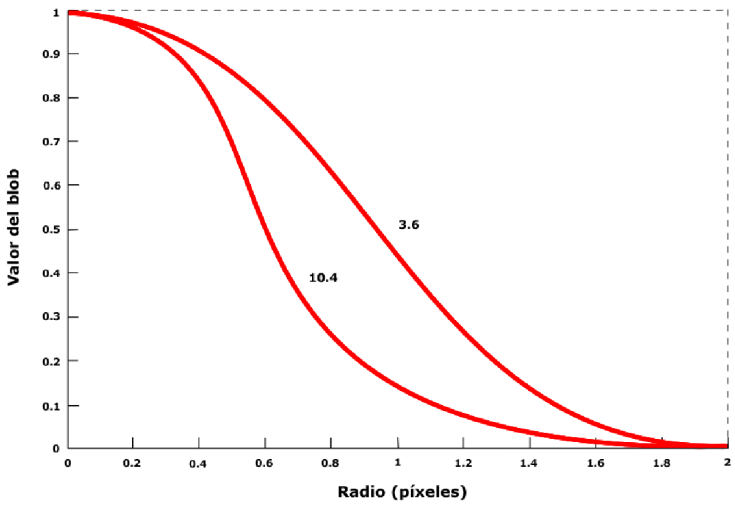

FIGURA 2.6: Perfil transversal de dos *blobs* tridimensionales de parámetros $r = 2$, $m = 2$ y $\alpha = 3,6,\ 10,4$

implementación con *blobs*, la proyección (integral de línea) a una distancia $s \in \mathbb{R}$ del centro está dada por:

$$p(s) = \begin{cases} \frac{2a}{\alpha} t \frac{\frac{3}{\alpha^2 t+1}\sinh\alpha\sqrt{t} - \frac{3}{\alpha\sqrt{t}}\cosh\alpha\sqrt{t}}{I_2(\alpha)} & \text{si } 0 \leq s \leq a \\ \\ 0 & \text{en otro caso} \end{cases} \tag{2.8}$$

donde $t = 1 - \left(\frac{s}{a}\right)^2$ y $I_n(x)$ es la función de Bessel de orden n (de primera especie). Estas funciones se definen por:

$$I_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{n+2k}}{k!\ \Gamma(n+k+1)}$$

donde $\Gamma()$ es la función gamma de Euler.

Este modelo esférico puede ampliarse colocando más de una esfera dentro de cada vóxel, posiblemente permitiendo su solapamiento. Un ejemplo de esto es el uso de una estructura cúbica centrada en el cuerpo (BCC), que es la que

emplea el paquete Xmipp desarrollado en el CNB. Aquí, se coloca un *blob* en cada uno de los vértices del vóxel, además de uno en el centro.

2.6.1.2. Factor de relajación

En sistemas que varían tratando de converger a valor, es usual introducir un factor de relajación. Lo que se pretende es disminuir la influencia del ruido, para lo que se introduce un factor cuyo efecto es dar más peso a los valores pasados que al último calculado. En nuestro caso se corresponde con, en vez de hacer la proyección completa hacia ese hiperplano, recorrer únicamente una fracción dada. Se muestra en la figura 2.7 el resultado de realizar una relajación de manera continua con $\lambda = 0{,}5$; la línea azul indica la señal que cambia y la roja la actualización de los valores [3] . Se aprecia que la variación se realiza al principio de manera gradual, de tal manera que se consigue una convergencia exponencial al nuevo valor. Si el pulso resultara ser espurio y volviera rápidamente a su valor inicial, la estimación habría variado poco y se recuperaría el valor verdadero en poco tiempo. Esto aumenta la estabilidad del algoritmo, reduciendo el impacto del ruido en la reconstrucción. Sin embargo, esta mejora en la calidad se hace a expensas de la velocidad de convergencia, que como es obvio se ve disminuida.

Fijándonos en la ecuación (2.6), vemos que este factor afectaría de la forma:

$$f_j^{(i)} = f_j^{(i-1)} + \lambda \frac{p_i - q_i}{\sum_{k=1}^{N} w_{ik}^2} w_{ij} \qquad \forall j \in 1..M \tag{2.9}$$

Es posible demostrar que la condición necesaria y suficiente para que el método Karcmacz, usando un factor de relajación, converja, es que éste tenga un valor comprendido en el intervalo $(0, 2)$. Intuitivamente esto era de esperar, puesto que cualquier valor en este intervalo acercará la solución al hiperplano,

[3] Con más precisión: si la línea azul es la gráfica de la función $w(x)$, la roja es la gráfica de $y(x)$, y el cambio se produce en $x = 0$, se ha dibujado la solución de la ecuación diferencial $y\prime(x) = \lambda(w(x) - y(x))$, que en nuestro caso ($w(x) = cte = w$) está dada por $y(x) = w + (y(0) - w)\, exp(-\lambda x)$.

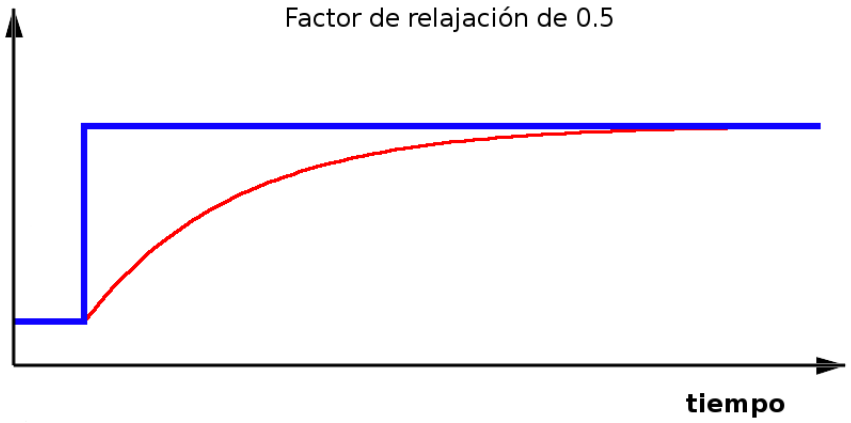

FIGURA 2.7: Efecto del factor de relajación

mientras que cualquier otro valor dejará la nueva estimación a una distancia mayor o igual que la anterior. En efecto, para $\lambda = 0$ el punto permanecería inalterado, mientras que para $\lambda = 2$ lo que se obtiene es la reflexión de ese punto sobre el hiperplano, con lo que su distancia a él permanece constante y no se ha avanzado en la obtención de una mejor aproximación. Es fácil ver que cualquier valor en (0, 2) resulta en una nueva solución más cercana al hiperplano, y cualquier valor superior a 2 o inferior a 0 lo aleja.

2.7. SIRT

Simultaneous Iterative Reconstruction Technique (SIRT) es una variación del ART que normalmente consigue mejores reconstrucciones a costa de una convergencia más lenta. Para ello se usa alguna distribución de densidad sencilla y para cada rayo se calcula la corrección que se le aplicaría a cada vóxel. Pero en vez de aplicarla inmediatamente después de tratar ese rayo, se recorren todos los demás calculando las correcciones asociadas a ellos. Una vez todos han sido procesados, se calcula la media de las correcciones para cada vóxel, que es la corrección que se aplica entonces. Cuando se termina esta primera

iteración, se vuelve a comenzar con el primer rayo para repetir el proceso.

2.8. SART

Simultaneous Algebraic Reconstruction Technique (SART) es un algoritmo perteneciente a la familia del ART creado por A.H Andersen y A.C. Kak [AK84] en 1984. El objetivo es tratar de combinar lo mejor de los dos métodos descritos anteriormente: las propiedades supresoras de ruido de SIRT y la rápida convergencia de ART.

Como método de reconstrucción algebraica, actúa aplicando iteraciones de proyecciones y backproyecciones al volumen hasta llegar a una solución. Pero a diferencia de ART, que aplica la corrección por cada rayo, y de SIRT, que las aplica sólo tras haber procesado todos los rayos, SART calcula las correcciones utilizando todos los rayos de una determinada proyección. Una vez que los ha procesado continúa con la backproyección, aplicando a cada vóxel una media ponderada de las correcciones obtenidas por cada rayo. Una descripción de este algoritmo es:

El algoritmo SART

```
Inicializar volumen
Hasta alcanzar la convergencia
    1) Seleccionar una proyección P
    2) Proyectar del volumen
          Calcular la suma de cada uno de los rayos de la proyección P,
          dando lugar a los valores de los píxeles.
    3) Calcular la correción
          Para cada píxel, calcular la diferencia de los valores reales
          de la proyección P y la suma de los rayos calculada.
    4) Backproyectar la corrección
          Distribuir las correciones entre todos los vóxeles del volumen.
```

La expresión matemática utilizada para la correción es ligeramente distinta a la del ART:

$$f_j^{(k)} = f_j^{(k-1)} + \lambda \frac{\sum_{i=1}^{M} \left(\frac{p_i - \sum_{n=1}^{N} w_{in} f_n^{(k-1)}}{\sum_{n=1}^{N} w_{in}} \right) w_{ij}}{\sum_{i=1}^{M} w_{ij}} \tag{2.10}$$

donde N es el número de vóxeles del volumen, M es el número de rayos en la proyección actual, p_i es la integral medida del rayo i-ésimo, $f_j^{(k)}$ es el valor del vóxel j después de haber tratado k proyecciones, w_{ij} es la integral del rayo i-ésimo en el vóxel j, y λ es el factor de relajación.

Como puede verse, el interior del sumatorio del numerador calcula la corrección para el vóxel j debido al rayo i. El numerador entero calcula la suma ponderada de todas esas correcciones para todos los rayos de una proyección P, dando mayor peso a los rayos con los que hay mayor intersección, y el denominador calcula la suma de todas las integrales de rayo de los rayos de la proyección P que han pasado por el vóxel j. Así pues, la corrección que se le aplica al vóxel j es una media ponderada de todas las correcciones debidas a los rayos de la proyección P, efectuando una relajación con el factor λ.

Como en este método las sumas de rayos se calculan para todos los rayos de una proyección, tiene sentido proponer varias formas de hacerlo. Hay básicamente dos: proyección orientada a píxel y proyección orientada a vóxel. En la proyección orientada a píxel se parte del píxel y se recorre cada rayo calculando qué vóxeles atraviesa y con qué integral. Sumando todas ellas se obtiene la suma del rayo, es decir, el valor del píxel. En la proyección orientada a vóxel se procede de una manera diferente, se recorren los vóxeles calculando qué píxeles afectarían y cuanto. Para efectuar este procedimiento lo usual es usar una técnica conocida como *splatting* (véase sección 5.2.2). Se divide el

volumen en franjas y cada una se trata por separado, como si cada una fuese un volumen en sí. Cada franja nos dará un valor en cada píxel, así que para conseguir el valor final lo que tendremos que hacer es sumar dichos valores de todas las franjas.

Una de las características más sorprendentes del SART es que proporciona reconstrucciones de gran calidad con solo una iteración (esto es, con solo una pasada a través de todas las proyecciones).

Capítulo 3

Unidades de procesamiento gráfico (GPUs)

3.1. Generalidades

Una unidad de procesamiento gráfico (GPU) es un microprocesador dedicado específicamente al renderizado de gráficos; es decir, el procesamiento de una descripción de alto nivel de una escena para obtener una imagen bidimensional rasterizada, liberando a la CPU de gran parte de la carga de trabajo del proceso y acelerándolo significativamente. Su desarrollo ha sido fuertemente impulsado por la industria de los videojuegos (de la que emana la necesidad de generar imágenes renderizadas de calidad cinematográfica en tiempo real), y como resultado hoy en día están ampliamente disponibles en ordenadores personales, estaciones de trabajo y videoconsolas.

Lo importante para nuestros propósitos es que el proceso de renderizado puede ser paralelizado muy eficientemente, y por tanto la estructura de las GPUs es especialmente adecuada para aplicaciones altamente paralelas y computacionalmente intensivas; como veremos, para tales aplicaciones es mucho más efectiva una implementación en una GPU programable que en una CPU. Por este motivo, dedicaremos este capítulo a estudiar la arquitectura de las GPUs actuales. Empezaremos mostrando el funcionamiento general del

pipeline gráfico, así como las ventajas e inconvenientes que se derivan de su uso para tareas distintas de para las que fueron originalmente diseñadas.

3.2. *Pipeline* gráfico

La entrada al *pipeline* gráfico es, típicamente, una descripción en alto nivel de una escena tridimensional generada por una aplicación que se ejecuta en la CPU. Esta escena suele estar representada por una serie de polígonos, con información asociada sobre colores y materiales. El resultado es una imagen bidimensional rasterizada y almacenada en el *framebuffer*. "Rasterizada" quiere decir simplemente que el resultado es una matriz que contiene el color final de cada píxel individual, y por tanto ya está lista para ser mostrada directamente.

La estructura básica del proceso suele concentrase en torno a tres grandes etapas conceptuales (ver figura 3.1):

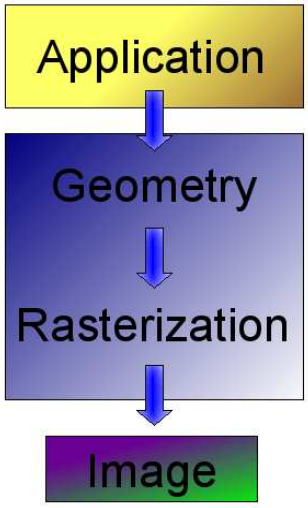

FIGURA 3.1: *Pipeline* gráfico. Las tres etapas principales

1. **Aplicación**: es el programa que se ejecuta en CPU, con comandos referidos al subsistema gráfico de la GPU. Normalmente accede a la GPU mediante APIs de alto nivel como pueden ser OpenGL o DirectX.

2. **Geometría**: calcula qué es lo que se va a mostrar, cómo y dónde. En términos más precisos, se encarga de proyectar los triángulos en la pantalla y calcular atributos como iluminación, color y orientación de las superficies. Todo esto se lleva a cabo mediante operaciones por polígono, como transformaciones de coordenadas, iluminación, generación de texturas de coordenadas y recorte. Este proceso se puede llevar a cabo en paralelo para cada uno de los triángulos.

3. **Rasterización**: se calculan los píxeles (fragmentos) de la imagen resultante y se procesan usando mapeado de texturas, interpolación de colores, visibilidad y mezclado de transparencias. Aquí se opera por píxeles independientemente.

Cada una de estas fases se divide a su vez en varias etapas, y cada una de ellas actúa de manera independiente y simultánea, procesando el resultado de la anterior y enlazando la salida con la entrada a la siguiente etapa. El rendimiento de la unidad gráfica de procesamiento está principalmente determinado por la latencia de la etapa más lenta.

La estructura básica del *pipeline* de renderizado es la que se observa en la figura 3.2. En las secciones siguientes se profundizará en cada etapa de las que se acaban de mencionar.

3.2.1. Etapa geométrica

La escena está definida por medio de vértices, normales, colores y propiedades de iluminación de los materiales, que se concreta mediante primitivas proporcionadas por la API gráfica. Cada elemento de la escena puede estar definido en un marco local de coordenadas distinto (*local space*); el cometido de la etapa geométrica es llevar a cabo una serie de transformaciones de coordenadas cuyo resultado final sean píxeles del puerto de vista (que es donde se mostrará la imagen). Las distintas fases de esta etapa, que pueden verse en la figura 3.3, son:

1. **Modelado y vista**: los datos geométricos de los objetos de entrada, que

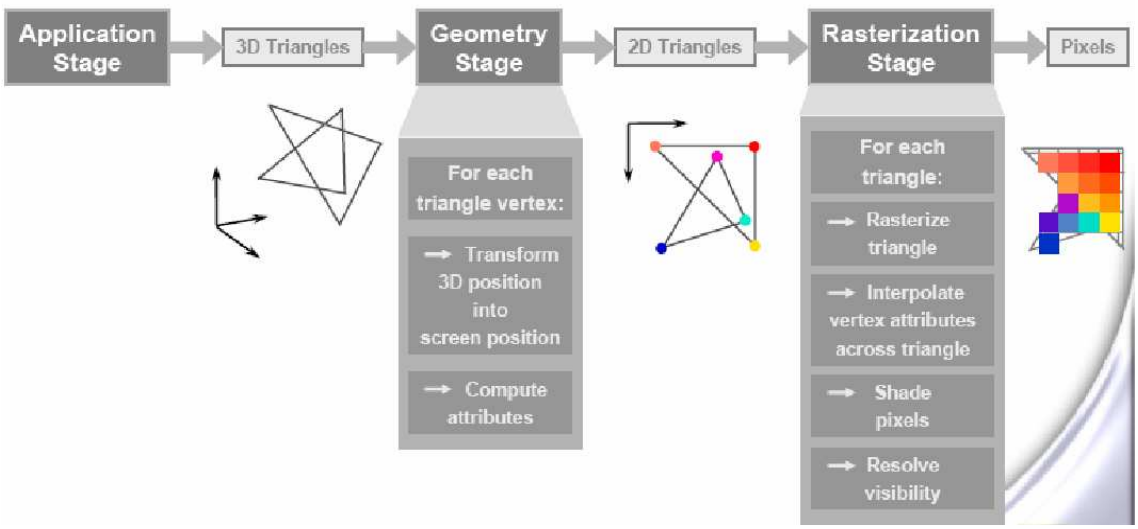

FIGURA 3.2: Estructura básica del *pipeline* gráfico

están inicialmente cada uno enmarcados en un sistema de coordenadas propio (*local space*), se transforman en "coordenadas del mundo" (*world coordinates*), que es un sistema universal de coordenadas en el espacio tridimensional en el que se enmarcan todos los objetos; el resultado de esta transformación (denominada modelado) se convierte a su vez a un nuevo marco de coordenadas (*eye coordinates*) que representa la escena vista desde la cámara (en función de su posición y orientación). Esta división en dos pasos es más bien de carácter conceptual, puesto que lo usual es que ambos se realicen en uno, que se reduce a aplicar una transformación afín dada por la **matriz de modelado y vista**, que combina ambos efectos.

2. **Iluminación por vértice**: se aplican modelos de iluminación a los objetos en función de las posiciones y características de las fuentes de luz (que pueden tratarse de luz ambiente, focos omnidireccionales, focos cónicos...), propiedades de los materiales (la proporción de luz ambiente, difusa y especular que reflejan), normales a las superficies y posición de la cámara. Estos modelos son sólo aproximados; una simulación realista de los procesos físicos de iluminación necesitaría de técnicas de *ray-tracing*

y carecería de la velocidad necesaria para su realización en tiempo real. Es necesario realizar esta fase antes de la de proyección, puesto que en las siguientes se distorsionará la forma de los objetos (hasta ahora sólo se han aplicado isometrías).

3. **Proyección**: se utiliza una aplicación proyectiva para realizar la transformación de proyección (la forma exacta puede variar, dependiendo de si se usa proyección paralela o perspectiva). El volumen de vista (la parte de la escena susceptible de ser dibujada) se transforma en un cubo canónico centrado en el origen para obtener las coordenadas de recorte (*clip coordinates*). Nótese que lo que se mostrará en pantalla es solamente el contenido de uno de los planos de este cubo (aunque con un factor de escalas horizontales y verticales distinto), pero todavía no se puede desechar el resto de la información para poder determinar qué objetos están ocultos por otros más cercanos.

4. **Clipping (recorte)**: se recorta la imagen para eliminar todo lo que resulte estar fuera del cubo canónico. Esta operación está altamente optimizada puesto que no depende de parámetros externos (las dimensiones de este cubo están fijadas de antemano).

5. **División perspectiva**: Esta transformación convierte las coordenadas de recorte a coordenadas del dispositivo; aquí es donde se obtienen las coordenadas bidimensionales de la proyección de la escena en un plano.

6. **Transformación del puerto de vista:** se transforman las coordenadas de dispositivo en coordenadas del puerto de vista para mostrar la imagen resultante por pantalla; el resultado es una serie de triángulos al que se debe aplicar ahora el proceso de rasterización, por lo que con esta fase se concluye la etapa de geometría.

3.2.2. Etapa de rasterización

La rasterización es el proceso de conversión de un gráfico vectorial en una imagen de mapa de bits. Durante el rasterizado se convierten los trazados

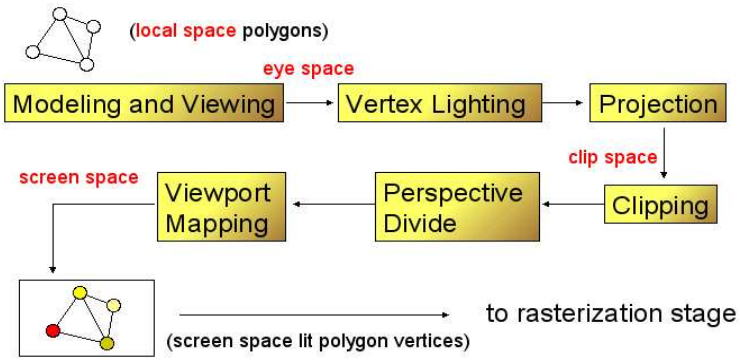

FIGURA 3.3: *Geometry Stage*. Etapas

de los gráficos en píxeles. Las opciones de rasterizado definidas determinan el tamaño y otras características de los píxeles resultantes. Por ello, en esta etapa se tratan operaciones por píxel: se asignan colores a los píxeles en el *framebuffer* (proceso conocido como *scan conversion* o *rasterization*), como se esquematiza en la figura 3.4. Normalmente los colores del interior de un polígono se interpolan de acuerdo a los colores de sus vértices (ya determinados en la etapa previa).

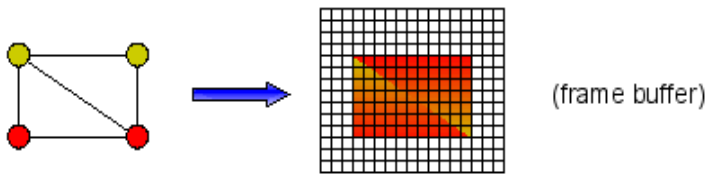

FIGURA 3.4: *Rasterization Stage*

En este contexto se conoce como "primitiva" a una forma geométrica básica como puede ser un círculo, un triángulo o un cuadrado. Las principales fases englobadas dentro de la etapa de rasterización son: *sampling* (convertir una primitiva en fragmentos), interpolación y texturas (iluminación, texturización, z valores...), combinaciones de colores, *fogging* y varios tipos de test

de píxeles (alpha, visibilidad).

Existen dos operaciones principales:

- **Selección de fragmentos**: genera un fragmento por cada píxel que se interseca con la primitiva.
- **Asignación de fragmentos**: establece las propiedades de la primitiva (color, profundidad...) a cada fragmento.

3.3. Computación de Propósito General en GPUs (GPGPU)

Después de haber introducido el funcionamiento básico de una GPU moderna, examinaremos los desarrollos recientes que han permitido su aplicación a otros campos.

El uso de GPUs para computación de propósito general (GPGPU) es un área activa de investigación bastante reciente. Se han utilizado con éxito para muy diversos campos, tanto en aplicaciones de ingeniería (visión por computador, codificación de vídeo, procesamiento de señales, imágenes, voz...) como en el ámbito científico (dinámica de fluidos, predicción meteorológica, bioinformática...). Inicialmente, sin embargo, las GPUs no eran programables, siendo por tanto su funcionalidad bastante limitada; en seguida veremos que sus posibilidades de programación han estado creciendo constantemente en los últimos años. Por otra parte, tradicionalmente se han representado los colores como valores entre 0 y 1 con sólo 8 bits de precisión, lo que es inaceptable para un uso científico por lo que el primer paso fue aumentar el rango y la precisión de los cómputos (actualmente se soporta coma flotante de 32 bits, lo que todavía es insuficiente para algunas aplicaciones).

En el año 2000, la necesidad de permitir a los programadores la inclusión de efectos especiales más realistas llevó a que se les añadiera posibilidades de

programación; se empezó permitiendo el uso de *vertex shaders* programables (que permiten procesar los vértices antes de la etapa de rasterización, alterando atributos como color o posición de cada vértice), y poco después siguieron los *fragment shaders* (que se ejecutan después de la etapa de rasterizado y permiten cambiar el color de cada píxel para, por ejemplo, utilizar un modelo de iluminación distinto). A principios del 2003 se desarrollaron los primeros lenguajes de programación de alto nivel (como Cg).

Sin embargo, seguía habiendo limitaciones bastante restrictivas sobre los patrones de acceso a memoria admisibles (básicamente, era muy costoso "esparcir" datos en direcciones de memoria no consecutivas) y en el control de flujo (no se permitían instrucciones condicionales). Ahora, sin embargo, podemos considerar que las GPUs actuales son totalmente programables, aunque con unas características de rendimiento que difieren significativamente de las de las CPUs.

Hasta hace poco, seguía habiendo un problema, sin embargo: el programador debía remodelar sus algoritmos y estructuras de datos para que encajaran en conceptos gráficos, que para la mayoría de las aplicaciones es bastante poco natural. Para remediarlo, en febrero de 2007 NVIDIA desarrolló una nueva tecnología conocida como **CUDA** (*Compute Unified Device Archiceture*, Arquitectura de Dispositivo Unificada para Cómputo), que funciona con la serie GeForce 8 y permite la programación de la GPU de una manera general siguiendo el modelo de ***Stream Processing***, que detallaremos a continuación. Esto permite al programador acceder a los recursos masivamente paralelos que ofrecen las GPUs, utilizando un lenguaje de programación que es una extensión de C.

3.3.1. *Stream Processing*

Stream processing es un paradigma relativamente nuevo, pero que ha tenido bastante éxito, que permite procesamiento de datos paralelo, consiguiendo una eficiencia nunca antes alcanzada con un mínimo esfuerzo. En ciertas apli-

caciones que requieren operaciones masivas sobre vectores, puede incrementar en varios órdenes de magnitud el rendimiento de una CPU convencional.

Dados un conjunto de datos de entrada y salida llamados ***streams***, el modelo está principalmente basado en la definición de una serie de operaciones de cómputo intensivo, llamadas ***kernel functions***, que se aplican a cada elemento dentro del *stream*. El paradigma que mejores resultados ha dado es el ***Uniform Streaming***, en el que en cada momento se aplica un único *kernel* a todos los datos (si es necesario utilizar varios *kernels* para diversas tareas, se aplican secuencialmente, pero en cada momento sólo uno de ellos está activo). La ejecución de los *kernels* debe poder llevarse a cabo en cualquier orden sin que el resultado varíe. Este paradigma está íntimamente relacionado con las arquitecturas SIMD (*Single Instruction - Multiple Data*), y es capaz de simplificar las interconexiones y conseguir grandes incrementos del rendimiento y un modelo sencillo de programacion, permitiendo un desarrollo en alto nivel y al mismo tiempo que se obtiene un alto rendimiento en hardware.

El *Stream Processing* es en realidad un compromiso, que funciona muy bien para las aplicaciones con un modelo de datos céntrico (como las tradicionales aplicaciones de procesamiento digital de señales), pero carece de la misma eficacia en aplicaciones más generales que requieren cuyos accesos a los datos son más impredecibles (como las bases de datos). Es decir, se intercambia la flexibilidad por un mayor rendimiento en un cierto tipo de aplicaciones (análogamente a lo que ocurre con el hardware de las GPUs).

Una gran ventaja del modelo de *Stream Programming* reside en el hecho de que un *kernel* mantiene un uso de los datos independiente (respecto a otros hilos ejecutando el mismo *kernel*) y local. Aunque no suelen estar expuestas en el modelo de programación, las operaciones de entrada / salida están mucho más avanzadas en procesadores de *streams*. Esto se debe a que están interconectadas entre ellas de manera que la estructura del chip puede ayudar a ocultar latencias. La definición explícita de los datos se hace en el *kernel*, que espera tener unas entradas y salidas bien definidas. Tener por tanto cada

bloque de computación independiente y bien definido permite incrementar la eficiencia de la cache y del bus de memoria.

La localidad de los datos es también explícita en el *kernel*. Este concepto, llamado *kernel locality*, identifica todos los valores con una corta vida con una sola invocación de un *kernel*. Todos los valores temporales se suponen locales a cada llamada a un *kernel*, y de esta manera el hardware o el software puede alojar los datos en los registros, a los que se accede de una manera rápida y eficiente. Este hecho está estrechamente relacionado con el grado de paralelismo que puede ser explotado.

El paralelismo entre dos instancias de *kernel* es equiparable al paralelismo entre hilos. En cada *kernel* es posible utilizar paralelismo a nivel de instrucción y en algunas ocasiones, es también posible explotar paralelismo a nivel de tarea (por ejemplo, en la solapación de operaciones de entrada/salida). Es muy fácil tener miles de instancias de *kernel* pero es imposible tener la misma cantidad de hilos ejecutándose al mismo tiempo; el hecho de tener un número tan elevado de *kernels* en ejecución permite ocultar las latencias cambiando el hilo que se ejecuta cuando sea necesario.

3.3.2. Características de rendimiento

Como hemos visto, por su propia naturaleza el proceso de renderizado es muy paralelo (se puede operar independientemente sobre cada uno de los vértices y cada uno de los fragmentos), por lo que las GPUs están especializadas en tareas de cómputo intensivo y paralelo. Es importante entender las razones por las GPUs ofrecen una mayor potencia de cómputo y bajo qué circunstancias se aprovechan más eficientemente; examinaremos aquí los principales motivos.

Las CPUs actuales dedican gran parte de los recursos al control de flujo (predicción dinámica de saltos, reordenamiento dinámico de instrucciones, ejecución especulativa...) y cacheo de datos (*data caching*). Todas estas son

técnicas deseables para una unidad central de proceso cuya carga de trabajo consistirá en ejecutar, en modo multitarea, un cierto número de aplicaciones con diversas características, pero son impracticables o menos efectivas en el caso de las GPUs. Por ejemplo, el cacheo de datos es mucho menos efectivo, porque al tratar cantidades enormes de datos simultáneamente, el tamaño del conjunto de trabajo es demasiado grande, por lo que los *hit ratios* que se consiguen (alrededor del 80 %) no son comparables los que estamos acostumbrados en CPUs (cercanas al 99.8 %). Además, las tareas para las que se han dedicado las GPUs tradicionalmente no necesitaban mecanismos de control de flujo (pues se ejecuta el mismo conjunto de operaciones para todos los elementos). El no incluir estas técnicas supone una menor penalización por control y permite además concentrarse en aumentar los recursos puramente computacionales; esta es la razón por la que GPUs fácilmente adquiribles ofrecen cifras de rendimiento en coma flotante mucho mayores que las de las CPUs domésticas (actualmente alrededor de 300 GFlops; compárese con los $\sim$ 50 GFlops de las últimas CPUs de Intel); posteriormente incluiremos una comparativa cuantitativa más detallada.

Las características que una aplicación debería tener para permitir un uso óptimo de las tecnologías de GPGPU son por tanto:

1. Trabajar con enormes cantidades de datos de entrada/salida.

2. El mismo trozo de código se puede ejecutar a la vez sobre un gran número de elementos de datos; el orden en el que se ejecute en cada elemento no debe afectar al resultado final.

3. Alta intensidad aritmética (relación entre instrucciones aritméticas ejecutadas y accesos a memoria lanzados); esto permite que el *thread scheduler* elimine las latencias de memoria parando la ejecución de un hilo y lanzando otro en su lugar.

4. Dependencia mínima diferentes elementos de los datos.

3.3.2.1. Ventajas e inconvenientes de la GPGPU

Ya hemos visto algunas de las razones por las que el desarrollo en GPGPU puede ser ventajoso. Prueba de ello es que las GPUs están sufriendo una rápida evolución (tres veces más rápida de lo predicho por la Ley de Moore). Además son de bajo coste, debido fundamentalmente a la alta demanda por parte de los videojuegos. Cabe también destacar, que poseen unas muy altas prestaciones y capacidad programable, lo que permite explotarlas en un uso no referente a gráficos.

Pese a que las ventajas del uso de GPUs para ciertas aplicaciones están claras, no faltan las críticas, generalmente derivadas de la inconveniencia de usar un procesador para fines completamente diferentes a lo que se pensaba al diseñarlos. Un argumento común es la falta de continuidad de las arquitecturas usadas. Debido a la rápida evolución del hardware gráfico (que puede ser una ventaja como hemos comentado), implementaciones de algoritmos que funcionaban óptimamente, en un modelo de GPU, dejan de hacerlo, o lo hacen subóptimamente en un modelo posterior. Otra crítica es la dificultad que engloba, debido fundamentalmente a los conceptos de bajo nivel. Además hay que conocer que la aplicación tiene que ajustarse al modelo de *Stream Processing*, que se hacen necesarios entornos de depuraciuón adecuados y que no siempre es factible o conveniente la portabilidad de las soluciones.

3.3.2.2. Comparativa GPU-CPU

Esta sección pretende mostrar cuantitativamente las diferencias en cuanto a potencia de cómputo, ancho de banda e integración de transistores entre diversas GPUs y CPUs. Los datos se corresponden al *Supercomputing 2006*, que es el Congreso Internacional de Computación de Alto Rendimiento, Redes, Almacenamiento y Análisis, que se celebra anualmente en el mes de Noviembre desde el año 1988 y se pueden observar en la tabla 3.1.

Potencia de cómputo:

- Unidad SSE Intel Pentium 4 → 14.8 GFLOPS.

CPU / GPU	**Computación**	**Ancho de banda**	**Precio**
Intel Core2 Duo 3.0GHz	48 GFLOPS (peak)	21 GB/s (peak)	874$ (chip)
Nvidia GeForce 8800 GTX	**330** GFLOPS (obs)	**55.2** GB/s (obs)	599$ (tarjeta)

TABLA 3.1: Evolución de las GPUs frente a las CPUs

- Intel Pentium 4 Dual 3.0 GHz (2005) → 26.4 GFLOPS.
- ATI X800 (2004) → 63 GFLOPS.
- Nvidia GeForce 7800GTX (2005) → 165 GFLOPS = 850*2 (DDR, MHz) *4 (rgba) * 24 (pip).

Ancho de banda a memoria:

- Chipset Intel Pentium 4 → 6 GB/s.
- Intel® 955X Express Chipset (2006) → 10.7 GB/s.
- NV40 (2004) → 35.2 GB/s = 550*2 (DDR, MHz) * 256 (bits) = 1,1 (GHz) * 32 (B).
- G71 (2005) → 51.4 GB/s.
- G80 (2006) → 86.4 GB/s = 900*2 (DDR, MHz) * 384 (bits) = 1,8 (GHz) * 48 (B).
- G80 (2007) *overclocked* → 96 GB/s = 1000*2 (DDR, MHz) * 384 (bits) = 2 (GHz) * 48 (B).

Integración de transistores:

- Intel Pentium 4 Extreme Edition 840 (2005) → 230 millones
- Intel Core 2 Duo E6600 (2006) → 291 millones
- Nvidia GeForce 7800 GTX (2005) → 302 millones
- Nvidia GeForce 8800 GTX (2006) → **681 millones**

Rápida evolución en paralelismo:

- NV40 (2004) → 12 *pipelines* simultáneos (FP)
- G70 (2005) → 24 *pipelines* simultáneos (FP)
- G80 (2006) → 128 *streaming processors* (VP+FP)

3.4. CUDA

3.4.1. Introducción a CUDA

3.4.1.1. Evolución de las GPUs

En cuestión de pocos años, la unidades programables de procesamiento gráfico han visto incrementada su capacidad de computación de manera vertiginosa. Con la aparición de múltiples cores dirigidos por un alto ancho de banda, hoy en día las GPUs ofrecen multitud de recursos para procesamiento gráfico y no gráfico.

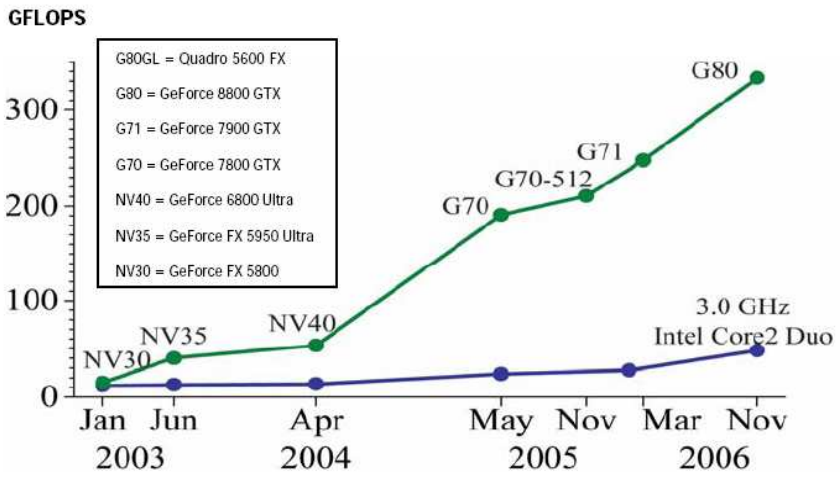

FIGURA 3.5: Gráfico de la evolución de las operaciones en punto flotante en CPU y GPU. Fuente: [NVI07]

La principal razón que hay detrás de está evolución es que la GPU está

especializada para un cálculo intensivo, con alto grado de paralelismo, y por tanto está diseñada tal que más transistores están dedicados al procesamiento de datos que al cacheo de datos y su control. Más específicamente, la GPU está especialmente bien situada para resolver problemas expresados como computaciones de datos paralelos con alta intensidad aritmética.

Sin embargo, acceder a todo este poder de computación que engloba una unidad de procesamiento gráfico y utilizarlo de la manera adecuada para aplicaciones no gráficas es un problema delicado:

1. Tradicionalmente la GPU sólo puede ser programada a través de una API gráfica, que impone una alta curva de aprendizaje para los que se inician y es inadecuada para aplicaciones no gráficas.

2. La memoria DRAM de una GPU puede ser leída desde cualquier parte de la memoria de manera general, pero hasta hace muy poco no podía ser escrita en cualquier parte, eliminando la flexibilidad disponible en CPU.

3. En algunas aplicaciones el cuello de botella está en el limitado ancho de banda de la memoria.

3.4.1.2. Una nueva arquitectura

CUDA es el acrónimo de *Compute Unified Device Arquitecture* y es una nueva arquitectura software y hardware creada para afrontar y administrar los cómputos en la GPU como un dispositivo de cómputo de datos en paralelo, sin necesidad de tener que mapearlos por medio de una API gráfica. Esta arquitectura está disponible a partir de los modelos Nvidia Geforce 8800. La estructura sofware de CUDA está compuesta por varias capas como se observa en la figura 3.6:

- Un controlador hardware.

- Una interfaz de programación de la aplicación (API) y su entorno de ejecución asociado.

- Dos librerías matemáticas de alto nivel de uso común: CUFFT (*Fast Fourier Transform*) y CUBLAS (*Basic Linear Algebra Subroutines*).

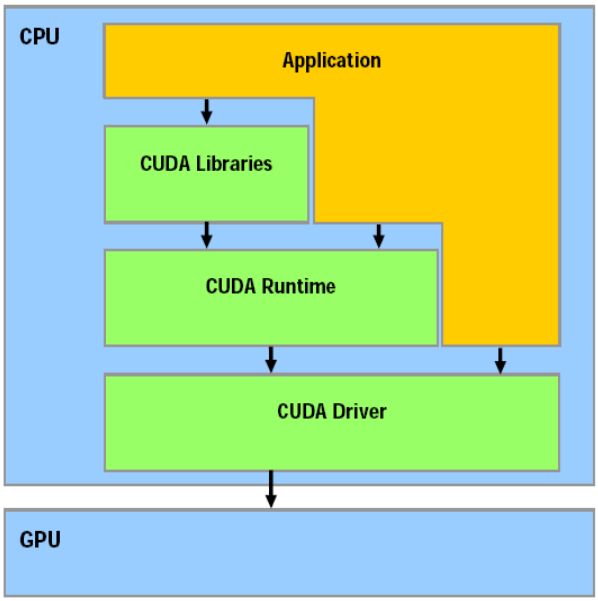

FIGURA 3.6: Capas software de CUDA. Fuente: [NVI07]

La API de CUDA se corresponde con una extensión del lenguaje de programación C, para facilitar su aprendizaje y su incorporación a otros proyectos.

CUDA proporciona memoria DRAM direccionada para facilitar la flexibilidad de programación: operaciones de lectura (*Gather*) y escritura (*Scatter*) en memoria (Ver figura 3.7). Esto significa que se tiene la posibilidad de leer y escribir datos desde cualquier punto de la memoria, como ocurre en una CPU.

CUDA proporciona una cache de datos paralela o memoria compartida en chip con un tiempos de acceso tanto para lectura como escritura muy rápidos, ya que los hilos (*threads*) la utilizan para compartir datos entre ellos.

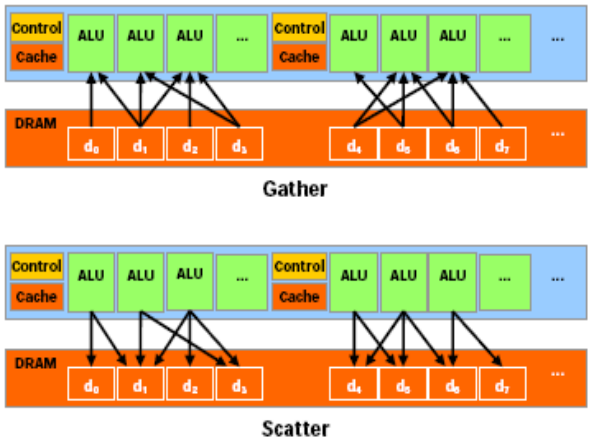

FIGURA 3.7: Operaciones de lectura y escritura en memoria en CUDA. Fuente: [NVI07]

3.4.2. Modelo de programación

Programando mediante CUDA, la GPU se puede ver como un *dispositivo de cómputo* capaz de ejecutar un alto número de hilos en paralelo. Opera como un coprocesador de la CPU, que en este contexto se conoce como ***host***. Concretamente, una porción de código de una aplicación que se ejecuta muchas veces, independiente de los tipos de datos que acepta, puede ser englobada en una función que es ejecutada en el dispositivo con varios hilos diferentes. Cada función compilada en el conjunto de instrucciones del dispositivo y su programa resultante recibe el nombre de ***kernel***.

Tanto el *host* como el dispositivo mantienen su propia memoria DRAM, referidas como *host memory* y *device memory* respectivamente. Cualquiera puede copiar datos desde su propia memoria a la otra a través de una API optimizada que utiliza los motores de alto rendimiento del dispositivo encargado de el Acceso Directo a Memoria (DMA).

3.4.3. *Thread Batching*

La colección de hilos que ejecuta un *kernel* está organizada en forma de rejilla (*grid*) de bloques de hilos, como puede verse en la figura 3.8.

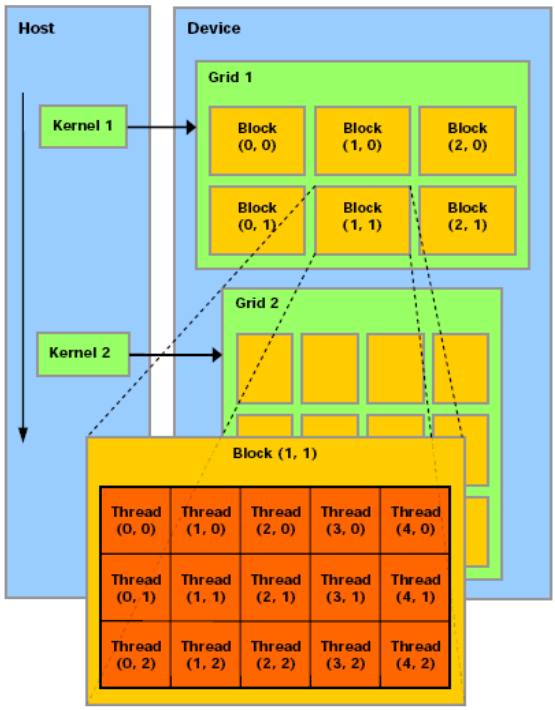

FIGURA 3.8: *Thread Batching*. El *host* invoca una serie de *kernels* al dispositivo. Cada *kernel* se ejecuta como una colección de hilos organizados en forma de rejilla. Fuente: [NVI07]

3.4.3.1. Bloques de hilos

Un bloque de hilos es una colección de hilos que pueden cooperar juntos, compartiendo datos eficientemente entre ellos, a través de memoria compartida muy rápida y sincronizando su ejecución para coordinar los accesos a memoria. En particular, se pueden especificar puntos de sincronización (barreras) en cada *kernel*, donde los hilos de un bloque se suspenden hasta que alcanzan el punto de sincronización.

Cada hilo está identificado por su *thread id*, que es el número (identificador) de hilo dentro del bloque. Para facilitar el direccionamiento por medio del identificador del hilo, una aplicación puede especificar un bloque como un array de dos o tres dimensiones de tamaño variable y identificar a cada hilo usando dos o tres indexaciones.

3.4.3.2. Rejillas de hilos

Existe un número máximo de hilos que un bloque puede contener. Sin embargo, los bloques que ejecutan el mismo *kernel* pueden englobarse juntos en una rejilla de bloques, para que, de este modo, el número total de hilos que pueden ser lanzados por un único *kernel* sea mucho mayor. Esto reduce la cooperacion entre hilos, ya que los hilos en diferentes bloques de un mismo *grid* no pueden sincronizarse ni comunicarse entre ellos. El dispositivo puede lanzar todos los bloques de una rejilla secuencialmente si tiene pocas capacidades de paralelismo, o en paralelo si tiene muchas, y en general se suele usar una combinación de ambas.

Cada bloque está identificado por su *block id*, que es el número de bloque dentro del *grid*. Al igual que ocurría con el caso de los hilos, para facilitar el direccionamiento a través del identificador de bloque se puede especificar un *grid* como un array bidimensional de tamaño arbitrario e identificar cada bloque usando dos indexaciones.

3.4.3.3. Modelo de memoria

Un hilo que se ejecuta en el dispositivo tiene sólo acceso a la memoria DRAM de éste y a través de los diferentes espacios de memoria como se observa en la figura 3.9.

Más específicamente:

1. Lectura y escritura por hilo en registros (*registers*).
2. Lectura y escritura por hilo en memoria local (*local memory*).
3. Lectura y escritura por bloque en memoria compartida (*shared memory*).
4. Lectura y escritura por grid en memoria global (*global memory*).
5. Sólo lectura por grid en memoria constante (*constant memory*).
6. Sólo lectura por grid en memoria de textura (*texture memory*).

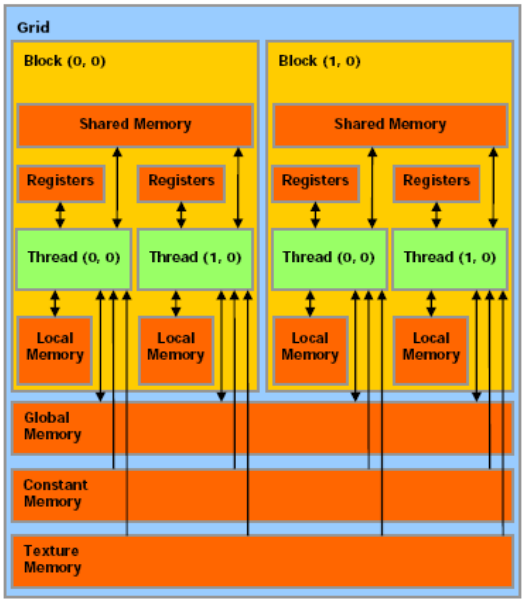

FIGURA 3.9: Modelo de memoria. Cada hilo tiene acceso a DRAM y a espacios de memoria de distintos ámbitos. Fuente: [NVI07]

Tanto el espacio de memoria global como el constante y el de textura pueden leerse y escribirse desde y por el *host*, y permanecen inalterables en las llamadas de *kernel* de la misma aplicación. La memoria de textura ofrece además diferentes modos de direccionamiento, como por ejemplo filtros de datos en diferentes aplicaciones.

3.4.4. Implementación hardware

El dispositivo está implementado como un conjunto de multiprocesadores donde cada uno de ellos posee una arquitectura de tipo SIMD (*Single Instruction, Multiple Data*) como se puede ver en la figura 3.10. En cualquier ciclo de reloj, cada procesador del multiprocesador ejecuta la misma intrucción pero trabaja con datos diferentes.

Concretamente, cada multiprocesador tiene integrado lo siguiente:

1. Un conjunto de 32 registros por procesador.

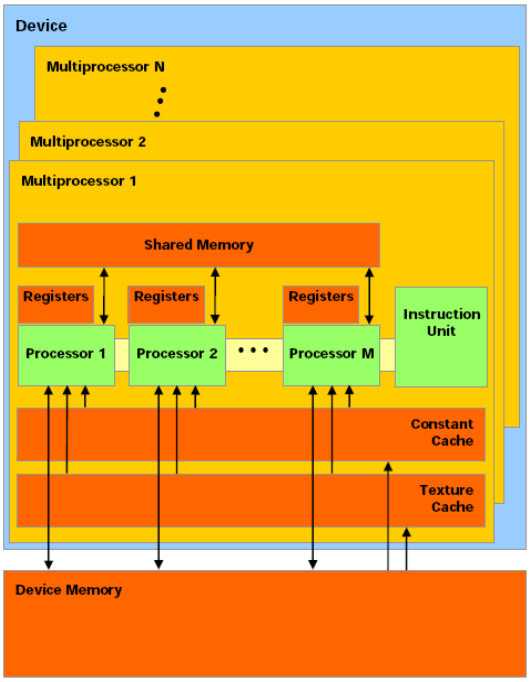

FIGURA 3.10: Modelo hardware de CUDA. Fuente: [NVI07]

2. Una cache de datos paralela (memoria compartida) compartida entre todos los procesadores.

3. Una cache constante de sólo lectura compartida por todos los procesadores y que incrementa las velocidades de lectura desde el espacio de memoria constante.

4. Una cache de textura de sólo lectura compartida por todos los procesadores y que incrementa las velocidades de lectura desde el espacio de memoria de textura.

3.4.4.1. Modelo de ejecución

Un *grid* de bloques de hilos es ejecutado en el dispositivo, ejecutando uno o más bloques en cada multiprocesador, usando rodajas de tiempo (*slices*). Cada bloque se divide en grupos SIMD de hilos llamados *warps*.

Cada uno de estos *warps* contiene el mismo número de hilos que determinan su tamaño (*warp size*) y es ejecutado por cada multiprocesador por medio de un método en el cual periódicamente se cambia de *warp* para maximizar el uso de los multiprocesadores de los recursos computacionales. De esta tarea se encarga el manejador de hilos (*thread scheduler*).

El método por el que un bloque de hilos se divide en *warps* es siempre el mismo: cada *warp* contiene hilos consecutivos, empezando con el hilo número 0 en el primer *warp* e incrementando el índice de hilo en los sucesivos.

Un bloque se procesa en un sólo multiprocesador; de ahí que el espacio de memoria compartida resida en el chip, para tener tiempos de acceso más rapidos. Un multiprocesador puede procesar varios bloques concurrentemente, repartiendo entre ellos el conjunto de registros y la memoria compartida. De hecho, el número de registros disponibles por hilo es igual a:

$$\frac{R}{B \cdot ceil(T, 32)}$$

donde R es el número total de registros por multiprocesador, B el número de bloques activos a la vez en el multiprocesador, T el número de hilos por bloque y $ceil(T, 32)$ es T redondeado al alza al múltiplo más cercano de 32.

El orden de ejecución de un *warp* dentro de un bloque está indefinido, aunque puede ser sincronizado para coordinar los accesos a memoria global o compartida. Si la instrucción ejecutada por los hilos de un *warp* escribe en el mismo lugar de memoria global o compartida, el orden de las escrituras no estará definido.

3.4.5. Serie Nvidia GeForce 8800

La serie GeForce 8800 de Nvidia, y concretamente la **GeForce 8800 GTX** utilizada, tiene como principales características, las siguientes:

1. El número máximo de hilos por bloque es 512.

2. El tamaño máximo de cada dimensión de un *grid* de bloques es 65535.
3. El número de multiprocesadores es de 16 a 675 MHz.
4. La cantidad de memoria de dispositivo es de 768 MB.
5. La cantidad de memoria compartida disponible por multiprocesador es de 16 KB dividida en 16 bancos de memoria.
6. La cantidad de memoria constante disponible es de 64 KB con una cache de 8 KB por cada multiprocesador.
7. El número de registros disponibles por multiprocesador es de 8192.
8. La cache para texturas unidimensionales es de 8 KB por multiprocesador.
9. El tamaño máximo del *warp* es de 32 hilos.

Por último, cabe destacar que cada multiprocesador está compuesto por ocho procesadores corriendo al doble de las frecuencias de reloj y a cada multiprocesador le está permitido ejecutar los 32 hilos de un *warp* en dos ciclos de reloj.

3.4.6. Factores influyentes en el rendimiento

Para conseguir que una aplicación aproveche al máximo los recursos que ofrece la GPU mediante CUDA hay que tener en cuenta multitud de factores. En esta sección citaremos los más importantes, pero una exposición más detallada se puede encontrar en [NVI07]. Los factores que influyen en el rendimiento pueden catalogarse básicamente en tres grupos: instrucciones, memoria y tamaños de bloque; cada uno con mayor o menor importancia dependiendo de las características de la aplicación.

3.4.6.1. Instrucciones aritméticas y de control

Para ejecutar una instrucción de un *warp* es necesario leer los operandos de cada *thread* del *warp*, ejecutar la instrucción y por último escribir el resultado de cada *thread* del *warp*. Dependiendo del tipo de la instrucción se

consumirá un distinto número de ciclos en lanzarla. Los ciclos consumidos por las principales instrucciones artiméticas son:

- La suma y multiplicación en punto flotante, suma de enteros, operaciones de bits, comparaciones, mínimo, máximo y conversiones de tipo consumen 4 ciclos.
- La raíz cuadrada y la multiplicación de enteros consumen 16 ciclos.
- Las operaciones de división y módulo en enteros son muy costosas y deben evitarse, convirtiéndolas en operaciones de bits cuando sea posible (el compilador lo realiza automáticamente cuando el dividendo es un literal).
- La raíz cuadrada en punto flotante requiere 32 ciclos, al igual que *sin*, *cos* y *exp*.
- La división en punto flotante consume 36 ciclos.

Existe una colección de funciones aritméticas que son más rápidas que las estándar, pero a costa de relajar el error cometido. Como norma general se llaman de la misma manera que las normales, pero añadiendo __ delante, como por ejemplo __*sin(x)* o __*log(x)*. Una lista completa de estas instrucciones y sus errores se puede consultar en [NVI07].

Las instrucciones de control (*if, switch, while, for, do*) pueden tener un gran impacto en el rendimiento, ya que pueden hacer que los hilos de un mismo *warp* sigan distintos caminos. Cuando ocurre esto las instrucciones son serializadas, es decir, se ejecutan primero las de un camino y después las del otro, inhibiendo la escritura de resultados a los hilos debidamente. Esto hace que el número total de instrucciones totales del *warp* aumente, empeorando el rendimiento. En ocasiones el compilador puede desenrollar bucles o usar ejecución predicada (*branch predication*) para evitar esta divergencia y no empeorar el rendimiento.

3.4.6.2. Instrucciones de acceso a memoria

Cada instrucción de acceso a memoria necesita 4 ciclos para ser lanzada. Aparte de esto, hay que tener en cuenta el número de ciclos (posiblemente mucho mayor) de latencia de memoria. Estos valores son muy variables dependiendo del tipo de memoria a la que se esté accediendo.

- Un acceso a la memoria global requiere entre 400 y 600 ciclos. Esta memoria no es cacheable y admite transferencia de un máximo de 128 bits por instrucción.

- La memoria constante es cacheable, y en caso de fallo se necesitan los ciclos de un acceso a memoria global. En caso de acierto se necesitan los ciclos de una acceso a registro por cada posición distinta referenciada en el *warp*.

- La memoria de textura funciona de manera similar a la memoria constante en cuanto a ciclos de acceso, aunque mejora la localidad espacial y es más versátil en cuanto a los patrones de acceso.

- La memoria compartida reside en el chip y por ello es muy rápida. Los tiempos de acceso son los mismos que los de un registro siempre que no haya conflictos de bancos, en cuyo caso se escalará con el número de conflictos encontrados. La memoria compartida está dividida en 16 bancos de la siguiente forma: la primera palabra de 32 bits pertenece al banco 0, la segunda al 1, la tercera al 2... y así cíclicamente. Cuando varios hilos de un *warp* acceden al mismo banco forman un conflicto, y esos accesos se deben serializar. De ahí que el número de ciclos requeridos en un acceso se escale con el número de conflictos, ya que se debe dividir todos los accesos en subgrupos libres de conflictos y servir uno detrás de otro.

- Los registros pueden ser accedidos sin ningún coste extra de ciclos salvo que existan dependencias *read-after-write*.

El ancho de banda entre la tarjeta y su memoria es mucho mayor que el ancho de banda entre memoria principal y la memoria de la tarjeta gráfica.

Es por esta razón que se debe utilizar la memoria de la tarjeta lo más posible, y transferir los resultados sólo una vez se ha efectuado todo el procesamiento. También es conveniente empaquetar las posibles transferencias en bloques grandes para amortizar la sobrecarga producida.

3.4.6.3. Tamaño de bloque

El tamaño de bloque ha de elegirse para que sea tal que maximize la utilización de los recursos. En particular, han de permitir que exista un mínimo de 2 bloques por multiprocesador, para que pueda cambiar entre uno y otro cuando uno esté esperando por algún motivo (acceso a memoria, sincronización...). Para que estos bloques puedan coexistir en el multiprocesador también es necesario que la memoria compartida usada por cada uno no supere la máxima del multiprocesador. También es interesante, sobre todo cuando hay un número elevado de bloques, que el número de hilos por bloque sea múltiplo del tamaño máximo de *warp* para no desperdiciar recursos en *warps* poco poblados.

La ocupación se define como el número de *warps* que se ejecutan concurrentemente en los multiprocesadores. Este valor es muy importante, ya que si se alcanza el máximo se consigue que el multiprocesador siempre tenga otro *warp* que ejecutar cuando uno de ellos se quede en espera por alguna razón. Para conseguir este valor máximo es necesario ajustar el tamaño de bloque teniendo en cuenta que se precisa que existan registros y memoria compartida suficientes para todos los *warps* concurrentes. Si la configuración de ejecución lanza bloques cuya necesidad de registros o memoria compartida excede los máximos del multiprocesador, dichos hilos no serán lanzados durante la ejecución. NVIDIA ha puesto a disposición de los desarrolladores [NVI] el *CUDA Occupancy Calculator*, una hoja de cálculo con la que medir de manera sencilla la ocupación alcanzada para cada *kernel* a partir del número de registros y la memoria compartida que usa, así como el número de hilos de ese mismo *kernel* que se lanzan por bloque.

En general, un tamaño de bloque de 64 es el mínimo aceptable para aplicaciones con un gran número de bloques concurrentes. Un valor entre 192 y 256 es normalmente el mejor y permite un aprovechamiento óptimo de los recursos.

3.4.7. Recomendaciones para el rendimiento

El principal factor para conseguir un buen rendimiento en la GPU mediante CUDA es el tipo de algoritmo que se quiere portar. Para conseguir el máximo aprovechamiento de los recursos, el algoritmo debe de ser de alta intensidad aritmética, es decir, que tenga un elevado número de instrucciones aritméticas por cada instrucción de acceso a memoria. Esto hará que el tiempo de espera debido a accesos a memoria pueda ser ocultado por la ejecución de instrucciones aritméticas.

Aparte de esto, es muy interesante minimizar el número de puntos donde el flujo de ejecución diverge en los hilos de un mismo *warp*, ya que esto hará que se serialicen los diferentes caminos. Si no se pueden evitar estos puntos de divergencia, lo más interesante es conseguir dividir los hilos de tal manera que todos los de un mismo *warp* sigan el mismo camino, eligiendo para ello una ordenación adecuada de los hilos dentro de los bloques.

Los accesos a memoria son las operaciones más costosas, así que son un punto muy importante a la hora de optimizar. Hay que minimizar el uso de la memoria principal (CPU) y la memoria global (GPU), ya que son las más lentas. Lo más recomendable es cargar en la memoria compartida los datos que necesitan los hilos de un bloque y que operen con ellos allí. Así se amortizará la transferencia si varios hilos acceden a ellos, o si se accede varias veces al mismo. Cuando hayan terminado de tratar esos datos, se moverán los resultados de vuelta a memoria global. También es interesante tratar de empaquetar datos de tal manera que en una operación de transferencia se muevan el máximo tamaño de datos posible. La memoria de texturas debe ser utilizada cuando la localidad espacial o temporal de los accesos a memoria efectuados por el

algoritmo sea alta, ya que permitirá que los datos estén en caché la mayoría de las veces que sean pedidos.

Hay que utilizar las instrucciones más rápidas siempre que sea posible, ya sea usando las operaciones de bits (para multiplicaciones y módulo de enteros, por ejemplo) o las funciones aritméticas rápidas si el error cometido es aceptable. Utilizar datos de tipo *float* (simple precisión) siempre que sea posible también es una buena elección, ya que son los datos que más rápido trata la GPU al ser los que se usan en las diversas etapas del *pipeline* gráfico. (En realidad por ahora es la *única* opción, ya que los datos de doble precisión se convierten a *floats* por el compilador de NVIDIA, pero esta situación va a cambiar en el futuro próximo).

Por último, se debe tratar de reducir al máximo el número de registros y memoria compartida que utiliza cada *thread*, ya que cuanto menor sea este número más fácil será conseguir la máxima ocupación de los multiprocesadores. Utilizando la hoja de cálculo de [NVI] es posible encontrar los tamaños de bloque que maximizan la ocupación y probarlos para ver cuál es el que mejor comportamiento obtiene (maximizar la ocupación no siempre significa minimizar el tiempo de cómputo).

3.4.8. Limitaciones

Concluiremos comentando brevemente las limitaciones específicas de CUDA (dejando aparte las intrínsecas al uso de GPUs para computación general, ya expuestas con anterioridad).

1. Sólo se soporta aritmética de precisión simple. Esto puede ser muy problemático para aplicaciones científicas que requieran un alto grado de exactitud, pero afortunadamente NVIDIA ha anunciado que próximamente se permitirá el uso de aritmética de doble precisión.

2. No sigue en su totalidad las convenciones del estándar de coma flotante IEEE-754. Las diferencias están principalmente en que no se garantizan

las mismas cotas de precisión, y en el comportamiento de los NaNs y números denormalizados.

3. Faltan mecanismos de sincronización. El único mecanismo soportado es la sincronización (por medio de una barrera) entre todos los hilos de un mismo bloque; y no hay mecanismo alguno que permita sincronización entre hilos de distintos bloques.

4. No se permiten llamadas recursivas. Esto es así porque, actualmente, todas las llamadas a funciones que se ejecuten en la GPU son extendidas en línea (*inlined*) por el compilador.

Capítulo 4

Implementación en CPU

Código fuente en `http://gpd.sip.ucm.es/enrique/gpusart/`

Una vez explicados los métodos de reconstrucción algebraica y las unidades de procesamiento gráfico (GPUs), continuaremos exponiendo los detalles de la implementación. El algoritmo elegido es el SART. La razón es doble: por un lado, vimos que SART constituye una mejora tanto con respecto a ART como a SIRT, en términos de tolerancia al ruido y rapidez de convergencia; por otro lado, el ART básico no posee un elevado grado de paralelismo (dado que se basa en una actualización de la estimación del volumen rayo por rayo), y como ya vimos esto es una condición indispensable para obtener un buen rendimiento de una implementación en GPU.

Esta parte se ha dividido en dos: la implementación del algoritmo SART para que se ejecute sólo en la CPU y el posterior proceso de adaptación a la GPU (que es el tema del siguiente capítulo). Este orden fue el que seguimos al elaborar el proyecto, ya que una versión que se ejecute íntegramente en la CPU es más sencilla de construir, a la vez que nos permite familiarizarnos el método, centrarnos en los aspectos puramente algorítmicos del problema y obtener resultados que nos permitan evaluar la calidad obtenida (dado que no tiene sentido tratar de acelerar un programa que no diera como resultado reconstrucciones de la exactitud deseada). Además, una versión en CPU es el punto de partida para el proceso de migración de partes a la GPU, y ayuda

mucho en aspectos tan difíciles en GPU como la depuración (ya que tener una implementación correcta del algoritmo facilita la comparación de resultados).

4.1. Formato de los archivos

Durante el proceso de la reconstrucción es necesario tratar con diversos tipos de archivos. Uno es el fichero de selección, donde se almacenan los nombres de las proyecciones que se usarán como entrada en el algoritmo. Las mencionadas proyecciones también son un tipo de archivo que es necesario tratar, al igual que el volumen reconstruido final. Con el fin de maximizar la utilidad del programa que elaboremos, necesitamos que estos formatos sean lo más estándar posible, ya que facilitará cualquier conexión con otros paquetes de software existentes actualmente y en el futuro y también será de gran utilidad a cualquiera que desee continuar extendiendo la implementación. Por esta razón nos hemos decantado por utilizar el mismo formato que usa Xmipp, la suite de aplicaciones para microscopía electrónica del CNB. Estos formatos, a su vez, son los mismos que los usados por otra importante suite de aplicaciones con la misma finalidad, SPIDER (***S**ystem for **P**rocessing **I**mage **D**ata from **E**lectron microscopy and **R**elated fields*, sistema de procesamiento de imágenes de microscopía electrónica y campos afines).

Estos formatos son:

- **Archivo de selección de proyecciones:** es un sencillo archivo de texto ASCII, con extensión *.sel*, donde cada línea contiene la ruta de una fichero de proyección. Al lado de ésta aparece el estado, que está representado con un 1 si está activo o un -1 si está descartado.

- **Archivo de proyección y volumen:** son los archivos con extensión *.xmp* o *.vol* y están formados por una cabecera de 1 kb seguida por los valores correspondientes, representados en *floats*. La cabecera contiene la información importante de la proyección como son: el número de rodajas (*slices*), columnas y filas; el tipo de archivo concreto del que

se trata; información estadística de los valores; los ángulos de Euler; desplazamientos y factor de escalado, etc. A la cabecera le siguen los valores de de la proyección/volumen codificados en *floats* de 4 bytes en formato IEEE 754. Estos valores siguen el mismo orden de bytes que la arquitectura donde se originaron (*little endian* o *big endian*). De máxima importancia para nosotros es la información relativa a los ángulos de Euler, que contienen los ángulos de una rotación respecto de un eje que pasa por el centro del volumen (que se toma como origen del centro de coordenadas). Esta rotación lleva el sistema universal de coordenadas a un sistema local en el que el eje Z es la dirección de proyección, y el centro de la imagen es la proyección del centro del volumen.

Para tratar estos archivos hemos optado por utilizar la interfaz de clases contenida en Xmipp, ya que es una implementación sobradamente probada (actualmente está en su versión 1.1), a la vez que facilitaría su incorporación a la suite.

4.2. Supuestos previos

Antes de comenzar con la implementación hay que abordar una serie de elecciones y suposiciones. Unas tratan sobre el procedimiento con el que se han tomado las proyecciones, y otras son decisiones tomadas acerca del modelo de volumen que se va a reconstruir. En nuestro caso se han tomado las siguientes, que son las mismas para todas las versiones implementadas tanto en CPU como en GPU:

- **Rayos de grosor nulo.** Esto justifica el uso de la transformada de rayos X, y es una aproximación físicamente realista. Además, se evitan complicaciones ya que un rayo, en una determinada fracción de su trayectoria, solo afectará a un vóxel.
- **Trayectorias rectilíneas.**
- **Haz de rayos paralelos.** Las proyecciones que tomamos para reconstruir el volumen han sido creadas con una sistema de proyecciones para-

lelas. Esto simplifica la reconstrucción ya que para una proyección dada todos los rayos que inciden tienen la misma dirección.

- **Proyección ortogonal.** El plano de proyección es perpendicular a la dirección de los rayos.

- **Rejilla tridimensional como distribución en el espacio de las funciones base.** Las funciones base cuyos coeficientes hemos de calcular se encuentran distribuidas en el espacio siguiendo una rejilla equiespaciada en cada una de tres direcciones ortogonales entre sí. Cada punto de la rejilla es el centro de un vóxel. La distancia entre vóxeles se toma para que sea igual que la distancia entre píxeles de la proyección. Si este valor fuese mucho menor habría demasiado pocos rayos pasando por cada vóxel, ya que éstos no tienen grosor. Esto llevaría a una reconstrucción artificialmente costosa y que no mejoraría la calidad, ya que la información para la corrección del coeficiente de la función base asociada al vóxel sería insuficiente. Análogamente, no estará justificado tomar tamaños de píxel mucho menores que los de los vóxeles, pues la reconstrucción en todo caso no podría tener la exactitud suficiente (dado que, dentro del vóxel, el valor de la función a reconstruir depende de un único parámetro). Esto justifica que la longitud del lado de un vóxel de la rejilla y la longitud del lado de un píxel de la imagen deban tomarse al menos del mismo orden de magnitud.

- **Los ángulos de las proyecciones son conocidos.** Esto es usualmente cierto en aplicaciones médicas, aunque en microscopía electrónica hay que realizar un proceso de estimación de ángulos. En este caso, durante el proceso de reconstrucción se suponen estos valores conocidos con exactitud, y posteriormente se realiza una nueva estimación de los ángulos, por lo que esto no resta validez a nuestra implementación, que sigue siendo necesaria.

Una vez determinadas todas las suposiciones del modelo y las proyecciones y los formatos de archivos utilizados podemos pasar a la implementación en sí.

4.3. Esquema genérico

Daremos aquí el esquema de una posible implementación del algoritmo SART (puede ser útil consultar de nuevo la expresión matemática del método, contenida en la ecuación 2.10 de la sección 2.8). Como se puede apreciar, en ella se ha evitado dar demasiados detalles específicos (como puede ser por ejemplo cómo recorrer los rayos que atraviesan a cada vóxel). La razón es que así el pseudocódigo mostrado es, por un lado, suficientemente preciso como para facilitar la comprensión del algoritmo y las estructuras que se utilizan; y a la vez suficientemente ambiguo como para dar lugar a diversas implementaciones basadas en el mismo esquema pero que difieren enormemente en cuanto a su eficiencia.

Las funciones `proyecta`, `corrige` y `backproyecta` se muestran en las figuras 4.2, 4.3 y 4.4 respectivamente. Las estructuras de datos utilizadas son las siguientes:

- `volumen`: matriz tridimensional que almacena los pesos de la función base asociada a cada vóxel.
- `proyCalculada`: matriz bidimensional que contiene la transformada de rayos X del volumen actual en una dirección determinada.
- `factNormPixel`: factores de normalización por píxel. Se corresponden con el denominador del sumatorio del numerador de la ecuación 2.10.
- `correccion`: contendrá las diferencias normalizadas (los coeficientes izquierdos del sumatorio del numerador de la ecuación 2.10).

4.4. Problemas de la implementación inmediata

En vista del pseudocódigo anterior, puede elaborarse una implementación del SART ingenua en la que para implementar, por ejemplo, los pasos descritos con "para cada vóxel que interseque a r" se realiza un recorrido sobre todos

```
/******************** SART ****************************************
 * Reconstruccion utilizando el algoritmo del SART                 *
 *                                                                 *
 *   Entrada: vector que contiene la informacion de todas las proyecciones *
 *   Salida: representacion del volumen reconstruido               *
 *****************************************************************/
SART(proyecciones) {
  // Crea un volumen de un tamano adecuado y una estimacion inicial
  volumen = inicializa(proyecciones.tamanos);

  // Si es necesario, ordena las proyecciones para aumentar la velocidad de convergencia
  proyecciones.ordena();

  while (no se satisfaga el criterio de convergencia)
    // Realiza una iteracion del algoritmo
    for (int i = 0; i < proyecciones.size(); ++i) { // para cada proyeccion medida
      <proyCalculada, factNormPixel> = proyecta(proyecciones[i].info, volumen);
      corr = correccion(proyecciones[i].datos, proyCalculada, factNormPixel);
      volumen = backproyecta(proyecciones[i].info, volumen, corr, factorRelajacion);
    }

  return volumen;
}
```

FIGURA 4.1: Pseudocódigo del SART

```
/************** Proyeccion ************************************
 * Calcula la proyeccion de la estimacion actual del volumen      *
 * segun la direccion de los rayos.                               *
 *                                                                *
 *   Entrada: informacion de proyeccion (tamano, angulos de Euler...)  *
 *   Salida:  proyeccion del volumen actual (proyCalculada)       *
 *            factor de normalizacion de cada pixel (factNormPixel)  *
 ******************************************************************/
proyecta(infoProyeccion, volumen) {
  // Factor de normalizacion de cada rayo, inicializado a 0
  factNormPixel[] = { 0 };

  // Proyeccion del volumen actual en cada pixel, inicializado a 0
  proyCalculada[] = { 0 };

  for (cada rayo r en infoProyeccion)
    for (cada voxel v en volumen al que interseque r) {
      integral = integralRayoEnVoxel(r, v);
      proyCalculada[r] += volumen[v] * integral;
      factNormPixel[r] += integral;
    }

  return <proyCalculada, factNormPixel>;
}
```

FIGURA 4.2: Pseudocódigo de la proyección

```
/***************** Correccion ******************************
 * Calcula las correcciones por cada pixel, segun la diferencia entre *
 * la proyeccion medida y la calculada                                *
 *                                                                    *
 *   Entrada: proyeccion medida (proyMedida),                         *
 *            proyeccion calculada (proyCalculada),                   *
 *            factores de normalizacion (factNormPixel)               *
 *   Salida:  array de correcciones (corr)                            *
 **********************************************************/
correccion(proyMedida, proyCalculada, factNormPixel) {
  // Correccion por cada pixel (rayo)
  double[] corr;

  for (cada pixel p en proyMedida)
    corr[p] = (proyMedida[p] - proyCalculada[p]) / factNormPixel[p];
  return corr;
}
```

FIGURA 4.3: Pseudocódigo de la corrección

```
/*************************************************
 * Distribuye las correcciones en el volumen     *
 *   Entrada:                                    *
 *     estimacion actual del volumen (volumen),  *
 *     array de correccion (corr)                *
 *     factor de relajacion (lambda)             *
 *   Salida: nuevo volumen                       *
 *************************************************/
backproyecta(infoProyeccion, volumen, corr, lambda) {
  for (cada voxel v en volumen) {
    double factNormVoxel = 0, sumaCorrecciones = 0;
    for (cada rayo r de infoProyeccion que corte a v) {
      double integral = integralRayoEnVoxel(info.direccionRayos, r, v);
      factNormVoxel += integral;
      sumaCorrecciones += corr[r] * integral;
    }

    volumen[v] += lambda * sumaCorrecciones / factNormVoxel;
  }

  return volumen;
}
```

FIGURA 4.4: Pseudocódigo de la backproyección

los vóxeles y se comprueba la intersección. Sin embargo, esta implementación, aunque fácil de llevar a cabo, tiene un coste computacional inaceptable. De hecho, su complejidad temporal es de $\theta(voxeles \times rayos)$, dado que tanto la proyección como la backproyección hacen un trabajo constante por cada par (rayo, vóxel).

Para simplificar los análisis, y dado que los tamaños de las imágenes y de los volúmenes no deberían ser dispares, nosotros supondremos que las dimensiones del volumen a reconstruir y de las proyecciones son todas iguales a N. En este caso, el coste en tiempo del algoritmo ingenuo está en $\theta(N^5)$. Un simple cálculo nos muestra que éste es un coste desorbitado si queremos ser capaces de reconstuir volúmenes de cientos de unidades de lado a partir de cientos de imágenes, como suele ser el caso en la práctica.

Sin embargo, este alto coste no es inevitable. La razón está en un hecho al que ya se ha apuntado con anterioridad, y es que un rayo sólo puede atravesar un número reducido de vóxeles, mucho menor que N^3, así que no es necesario comprobar su intersección con todos ellos. Por esta razón, la mayoría de las integrales de rayo por vóxel son nulas, con lo cual la mayor parte del tiempo se desperdicia comprobando intersecciones vacías. Dado un rayo cualquiera, el número máximo de vóxeles que atraviesa es $3N - 2$. Intuitivamente este resultado es bastante plausible (parece que el caso peor está cuando el rayo "está muy cerca de atravesar una diagonal del cubo"), y de hecho no es difícil hallar una demostración, que daremos a continuación.

Consideremos un volumen formado por $X \times Y \times Z$ cubos unitarios, de tal manera que las esquinas de cada uno de estos cubos (vóxeles) se corresponde con un punto $(x, y, z) \in \mathbb{R}^3$ de coordenadas enteras y tal que $0 \leq x < X$, $0 \leq y < Y$, $0 \leq z < Z$. Sea I el punto de intersección de un rayo con el volumen, y v su dirección, de tal manera que la posición de un punto del rayo está dada por $I + \vec{v} \cdot t$ al variar $t \geq 0$. Supongamos que $v_x \geq 0$, $v_y \geq 0$, $v_z \geq 0$ (esto es, el rayo se mueve en dirección positiva en cada uno de los ejes; como veremos enseguida esto no supone una pérdida de generalidad). Considérese ahora la función $f(p) = \lfloor p_x \rfloor + \lfloor p_y \rfloor + \lfloor p_z \rfloor$, donde $\lfloor\rfloor$ denota la función parte entera inferior; entonces $f(p + \vec{v} \cdot t)$ es una función creciente, y su valor está comprendido entre 0 y $X + Y + Z - 3$ para todo punto de la trayectoria del

rayo (salvo el punto de salida del cubo considerado). Además, este valor sólo varía cuando cambia la parte entera de alguna de las coordenadas, lo que equivale a que el rayo corte a un nuevo vóxel (obviamente distinto de todos los anteriores). Por tanto el número de vóxeles que ha atravesado el rayo está acotado por $f(voxeldesalida) - f(voxeldeentrada) + 1 = X + Y + Z - 2$. En el caso de no ser $v_x \geq 0$, $v_y \geq 0$, $v_z \geq 0$, es fácil que el mismo argumento anterior sigue siendo válido si se aplica a la función $f(p) = sgn(v_x) \cdot \lfloor p.x \rfloor + sgn(v_y) \cdot \lfloor p.y \rfloor + sgn(v_z) \cdot \lfloor p.z \rfloor$, donde sgn denota la función signo ($+1$ para positivos, 0 para el 0 y -1 para negativos). Por tanto queda probada nuestra afirmación.

Otra observación interesante, que será necesaria para algunas de las implementaciones, es que con las suposiciones del apartado anterior el máximo número de rayos que puede atravesar un vóxel es de 4. Esto se puede ver notando que la mayor distancia posible entre dos puntos de un mismo vóxel es $\frac{\sqrt{3}}{2} \simeq 0{,}866$. Es fácil convencerse de que dada la proyección de un rayo, sólo hay 4 posibles píxeles cuyo centro diste menos que este valor de ella (dependiendo de la posición donde haya quedado la proyección dentro del píxel correspondiente).

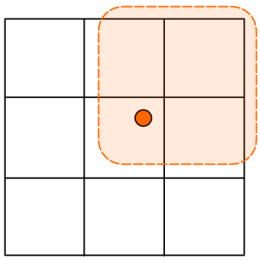

FIGURA 4.5: Se muestran los 4 posibles píxeles cuyos rayos pueden cortar al vóxel cuyo centro se proyecta en el punto marcado en rojo.

Con estas dos consideraciones ya estamos en condiciones de abordar un algoritmo cúbico, tarea que como veremos en la siguiente sección puede enfocarse de dos maneras.

4.5. Proyección

Como ya vimos al tratar el tema del algoritmo del SART, existen dos formas de calcular las proyecciones del volumen de una manera eficiente. Una está dirigida por rayos (*ray-driven*) y la otra está dirigida por vóxeles (*voxel-driven*). Aunque en principio las dos tienen un coste de $\theta(N^3)$ (tanto en tiempo como en espacio), cada uno tiene características que los hacen ser mejores dependiendo de en qué procesador correrán y qué funciones base y distribuciones se usarán.

4.5.1. Proyección dirigida por rayos

En esta forma de proyección se parte del rayo que llega al píxel i-ésimo de la imágen y se recorre para averiguar qué vóxeles atraviesa. La dirección del rayo se obtiene como la tercera columna de la matriz de rotación de Euler que permite pasar del sistema local de coordenadas (que está rotado para que el eje Z coincida con la dirección de proyección) al sistema de coordenadas universal.

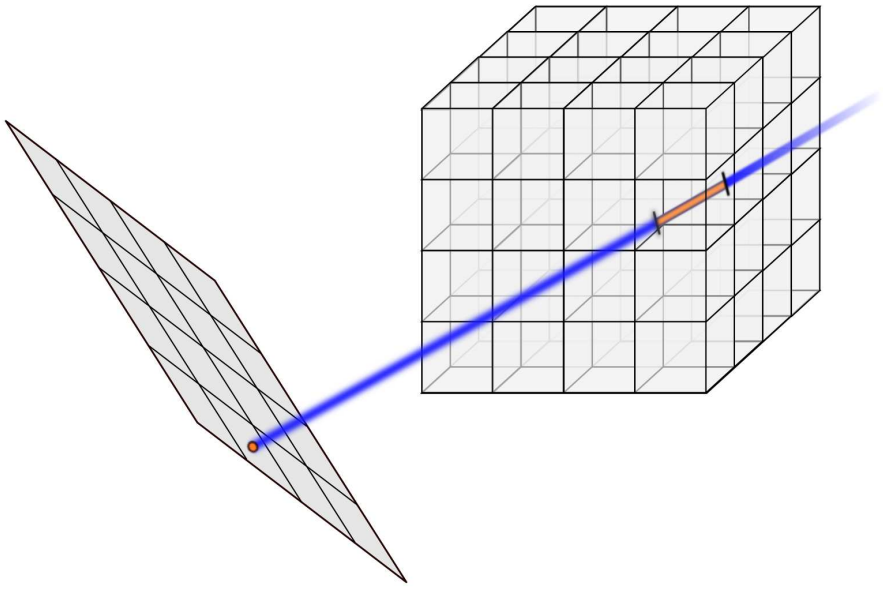

FIGURA 4.6: Proyección por rayos.

Como se ha comentado antes, este recorrido ha de ser algo más "astuto" que probar con todos los vóxeles, y de hecho se puede conseguir con coste constante por cada uno de vóxeles que atraviesa (y por tanto lineal con las dimensiones del volumen). Supongamos que los centros de los vóxeles están dados por $(x, y, z) \in \mathbb{N}^3 \mid 0 \leq x < X, 0 \leq y < Y, 0 \leq y < Z$ (y por tanto el paralelepípedo que contiene al volumen es el de esquinas $(-\frac{1}{2}, -\frac{1}{2}, -\frac{1}{2})$ y $(X - \frac{1}{2}, Y - \frac{1}{2}, Z - \frac{1}{2})$ y cuyos lados son paralelos a los ejes coordenados. La idea es simplemente cortar (*clip*) el rayo con este paralelepípedo, hallar el primer vóxel que atraviesa y, a partir de ahí, calcular en cada momento la menor longitud que debe recorrer el rayo para pasar al siguiente vóxel (o, equivalentemente, que alguna de las coordenadas cambie al siguiente múltiplo de $\frac{1}{2}$), hasta que se llegue al punto opuesto de la intersección. Este proceso de clipping se realiza utilizando una generalización a tres dimensiones del eficiente algoritmo de "Cohen-Sutherland", que se detallará a continuación. De las observaciones de la sección anterior se desprende que el coste de este algoritmo está en $\theta(N^3)$ en tiempo y espacio.

4.5.1.1. Algoritmo de Cohen-Sutherland generalizado a tres dimensiones

El algoritmo utilizado para la operación de clipping es el clásico Cohen-Sutherland, que en su forma bidimensional puede encontrarse en numerosas obras de informática gráfica (consúltese, por ejemplo, [FSH01]). Describiremos aquí directamente el caso tridimensional, en el que se parte de un segmento y un paralelepípedo cuyos lados están alineados con los ejes cartesianos y se trata de eliminar las partes del segmento que estén fuera del paralelepípedo (al que denotaremos por P).

El primer paso es calcular un código para cada uno de los dos extremos del segmento. Este código, para un punto Q determinado, indica en cuál de las 27 regiones externas a P en que las caras de P dividen al espacio está Q. Estos códigos se almacenan en un entero de 6 bits, de tal manera que cada uno de los bits indica, si está activo, que una de las coordenadas del punto es demasiado alta o demasiado baja, lo que corresponde a cada uno de los 6

```
/*************************************************************************
 * Implementacion del algoritmo de Cohen-Sutherland extendido para la    *
 * interseccion de una recta y un paralelepipedo con sus lados alineados con los ejes *
 * Devuelve true si queda algo visible                                   *
 * El segmento es r1->r2                                                 *
 *************************************************************************/
clipsegment(punto& r1, punto& r2, paralelepipedo p) {
  int c1 = codigo(r1, p), c2 = codigo(r2, p);
  if (c1 & c2)   return false; // trivial reject
  if (!c1 && !c2) return true; // trivial accept

  for (int i = 0; i < 6; ++i) { // para cada lado del paralelepipedo
    if      (bit i de c1 esta a 1) corta la parte correspondiente a r1 con el lado i;
    else if (bit i de c2 esta a 1) corta la parte correspondiente a r2 con el lado i;
  }

  punto pmedio = (r1 + r2) / 2;
  return <dentro(pmedio, v), r1, r2>;
}
```

FIGURA 4.7: Algoritmo de Cohen-Sutherland en 3D

pares de regiones. (Ver figura 4.8).

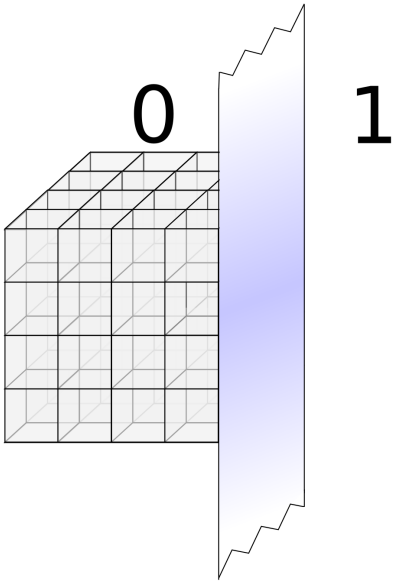

FIGURA 4.8: Cohen-Sutherland. Detalle de una de las seis regiones posibles

Se procede entonces a la realización de dos tests destinados a comprobar rápidamente si ambos puntos están dentro de P (*trivial accept*, en cuyo caso no es necesario el clipping) o ambos están en una misma región externa a P (*trivial reject*, en cuyo caso no hay intersección). El primer caso se da cuando ambos códigos son cero; el segundo cuando ambos tienen el mismo bit a 1. Si ninguno de los tests arroja un resultado concluyente, se procede a comprobar cada uno de los bits de los códigos que estén a uno y a cortar el segmento por el lado correspondiente a esa cara. Al final, comprobamos que el punto medio del segmento esté dentro de P; si es así, hay intersección con P y se devuelven sus los extremos; si no, se notifica que no hay intersección. Por último, el paso de hacer clip de un segmento a hacer clip de una línea (que es lo que necesitamos) es trivial; basta con extender el segmento lo suficiente como para asegurarse de que ambos extremos están fuera de P (y de hecho se puede simplificar ligeramente el código resultante porque en esta situación no se puede dar el *trivial accept*).

4.5.1.2. Pseudocódigo de la proyección por rayos

El pseudocódigo de este tipo de proyección queda como se ilustra en la figura 4.9.

Para terminar, destacaremos que este tipo de proyección tiene la ventaja de que, al ir avanzando vóxel a vóxel, resulta inmediato calcular la distancia que recorre el rayo dentro de esos vóxeles. Si la función base es tal que permite calcular su valor con este dato, es conveniente este tipo de proyección. En particular éste es el caso cuando se utilizan vóxeles como funciones base, ya que la integral del rayo en una celda no es más que la distancia recorrida multiplicada por un coeficiente; y por ello es por lo que la hemos elegido para este tipo de proyección. Como desventaja, este método posee un paralelismo máximo de solo N^2, es decir, solo podemos tratar en paralelo cada uno de los rayos (puesto que dentro del procesamiento de cada rayo, es preciso recorrer los vóxeles por los que pasa secuencialmente). En la práctica, para llevar a cabo una implementación paralela surgen problemas de concurrencia, pero no son excesivos ya que por un vóxel solo pasan un número muy pequeño de rayos. Discutiremos cómo solucionarnos en el capítulo de implementación en GPU.

4.5.2. Proyección dirigida por vóxeles

En este caso lo que se hace es recorrer todos los vóxeles del volumen y, para cada uno de ellos, calcular su proyección en el plano. Esto se consigue proyectando el centro del vóxel, qué normalmente caerá en algún píxel de la proyección, y luego comprobando los píxeles vecinos para ver si también contribuye a ellos. La proyección de un punto se obtiene simplemente premultiplicándolo por la matriz de Euler de paso del sistema universal al local. Esta matriz es la inversa de la que utilizamos para la proyección por rayos, y de hecho en este caso coincide con su traspuesta al tratarse de una matriz ortonormal, puesto que la transformación realizada es una isometría. Ahora, si el soporte de la función base es suficientemente pequeño (como es el caso si se usan vóxeles o *blobs* con el radio de la esfera circunscrita), los únicos rayos que pueden contribuir con una integral no nula son los

```
proyeccionPorRayos(infoProyeccion, volumen) {
  factNormPixel[] = { 0 };
  proyCalculada[] = { 0 };
  rayosPorVoxel[] = { vector vacio };

  // Extremos del paralelepipedo en el que estamos reconstruyendo
  paralelepipedo p(−0.5, −0.5, −0.5, volumen.Xdim − 0.5, volumen.Ydim − 0.5,
      volumen.Zdim − 0.5);

  for (cada rayo r en infoProyeccion) {
    // Corta p con la recta (no segmento) r
    <hayinterseccion, p, q> = clipline(r, volumen);
    if (hayinterseccion) {          // el rayo corta de p a q
      longitud = distancia(p, q); // longitud de la interseccion
      while (longitud > 0) {
        t = minima longitud a recorrer para llegar al siguiente voxel;
        v = voxel comprendido entre p y p + t * direccion normalizada de r;
        integral = integral de linea de la funcion base en v segun r;

        rayosPorVoxel[v].push_back(<r, integral>);
        normFactorPixel[r] += integral;
        proyCalculada[r] += volumen[v] * integral;

        longitud −= t;
      }
    }

  return <proyCalculada, factNormPixel, rayosPorVoxel>;
}
```

FIGURA 4.9: Pseudocódigo de la proyección dirigida por rayos

correspondientes a los 4 píxeles más cercanos a la proyección del centro del vóxel considerado (ver figura 4.5). Estos píxeles vecinos se pueden obtener simplemente comprobando la posición relativa entre esta proyección y el centro de ese píxel (si, por ejemplo, la proyección cae a la derecha y abajo, los vecinos candidatos se obtienen en la dirección creciente de X y decreciente de Y).

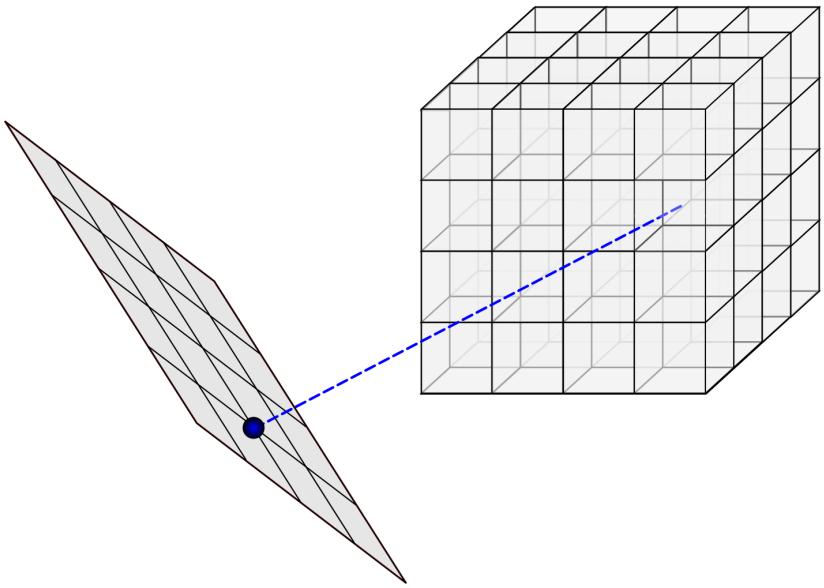

FIGURA 4.10: Proyección por vóxeles. Se proyecta el centro de un vóxel, que no tiene por qué caer en el centro de un píxel.

Una vez que se ha realizado esta tarea con todos los vóxeles, lo único que hace falta para calcular la imagen proyección es sumar todas las contribuciones que ha tenido cada píxel. Como el trabajo que se realiza por vóxel es constante (proyectar el centro, comprobar un número fijo de píxeles vecinos) el cálculo de las contribuciones tiene un coste de $O(N^3)$.

En este caso la información que tenemos al procesar cada vóxel no es la distancia que recorre el rayo dentro de él, sino la distancia del rayo al centro del vóxel (que es la distancia entre el centro del vóxel proyectado y el centro del píxel). En estas circunstancias, para acelerar el proceso conviene utilizar una función base cuya integral de rayo pueda ser calculada a partir de esa

```
proyeccionPorVoxels(infoProyeccion, volumen) {
  factNormPixel[] = { 0 };
  proyCalculada[] = { 0 };
  rayosPorVoxel[] = { vector vacio };

  for (cada voxel v en volumen) {
    proy = proyecta(v, infoProyeccion.EulerMatrix); // proyecta el centro del voxel
    for (cada uno de los 4 pixeles p mas cercanos a proy) {
      r = rayo que pasa por el centro de p;
      if (r interseca al soporte de la funcion base de v) {
        integral = integral de linea de la funcion base en v segun r;
        rayosPorVoxel[v].push_back(<r, integral>);
        normFactorPixel[r] += integral;
        proyCalculada[r] += volumen[v] * integral;
      }
    }
  }

  return <proyCalculada, factNormPixel, rayosPorVoxel>;
}
```

FIGURA 4.11: Pseudocódigo de la proyección dirigida por vóxeles

distancia. Para ello es necesario que la función base tenga una simetría esférica, para que no importe el lugar exacto del vóxel por el que ha pasado el rayo. Funciones base que se ajustan muy bien a estos requerimientos son las esferas macizas, esferas con densidad gaussiana o los blobs. En nuestra versión de CPU usando proyección dirigida por vóxeles nos hemos decantado por utilizar esferas en las que la densidad disminuye con la distancia al centro, de tal forma que la integral de un rayo a distancia r está dada por $c - r^2$. Tales esferas existen, aunque este modelo no tiene por qué ser el más apropiado para las aplicaciones reales, pero hay que tener en mente que nuestro objetivo final es una implementación en GPU (en la que utilizamos *blobs* al proyectar por vóxeles), y para comprobar la corrección del algoritmo el uso de unas funciones bases u otras no afecta prácticamente en nada. En principio, el paralelismo que se consigue con este tipo de proyección es de N^3, ya que cada vóxel es independiente de los demás para el proceso de la proyección. Pero hay que notar que un rayo pasa por un número elevado de vóxeles (máximo teórico $3N - 2$, en un phantom típico de $N = 64$ la media está en alrededor de 80) así que existirán problemas de concurrencia si se busca una implementación paralela, aunque pueden mitigarse en gran medida mediante *splatting*, que comentaremos en el siguiente capítulo (sección 5.2.2).

4.6. Cálculo de la corrección

El proceso de determinar la corrección correspondiente a cada píxel es el más sencillo de todos, ya que solo consta de una resta y una división por píxel. Por comodidad se utiliza una estructura de datos, `factNormPixel`, que se calcula durante la fase de proyección y donde se almacena la integral de línea para cada rayo atravesando el volumen. Con esta estructura resulta muy sencillo implementar el paso de corrección.

4.7. Backproyección

Independientemente de la forma que se haya elegido para la proyección, la backproyección se efectuará de la misma manera. Recogerá la corrección normalizada para cada píxel y la distribuirá convenientemente entre los vóxeles implicados en ese rayo. En este proceso también se necesitan las integrales de los rayos por los vóxeles, pero como estos datos ya se han calculado durante el paso de proyección (de cualquiera de las dos formas) es innecesario volver a calcularlos. Para ello hemos creado una estructura de datos llamada `rayosPorVoxel`, que es una matriz tridimensional donde almacenamos qué rayos han atravesado cada vóxel y con qué integral. Esta estructura se crea durante el proceso de proyección, y su tamaño es de $O(N^3)$ (esto se sigue del hecho de que el máximo número de rayos que pueden atravesar a un vóxel es de 4). Usándola, la backproyección queda muy sencilla ya que para cada vóxel la única tarea a realizar es recorrer la lista y calcular la corrección final que se le aplicará a ese vóxel.

Cabe destacar que el paralelismo de la backproyección sí que es $O(N^3)$ de verdad. Utilizando `rayosPorVoxel`, la corrección que se le aplica a cada vóxel puede hacerse de manera totalmente independiente de las otras. Así no surge ningún problema de concurrencia al acceder a ningún dato, ni el valor del volumen ni la lista de `rayosPorVoxel`.

4.8. Orden de las proyecciones

Uno de los objetivos de nuestra implementación es que disponga de la suficiente flexibilidad como para que se permita intentar acelerar la velocidad de convergencia mediante el ajuste de ciertos parámetros. Aparte de la elección del número de iteraciones a realizar y del factor de relajación, el algoritmo es especialmente sensible al orden en que se toman las proyecciones (como hicimos notar en la exposición del método Kaczmarz, sección 2.5). Por tanto, damos la opción de realizar un preprocesamiento de acuerdo a la geometría de las direcciones en que han sido tomadas las imágenes.

La estrategia óptima sería tomar los hiperplanos correspondientes a cada

sistema de ecuaciones y realizar una ortogonalización. Tal enfoque, sin embargo, no es factible en la práctica (dejando de lado el hecho de que tendería a reforzar los efectos del ruido): por una parte tendríamos que calcular todas las integrales de línea de todos los rayos antes de empezar una iteración propia de SART; y por otra un proceso de ortogonalización completo, aun cuando fuera posible, sería demasiado costoso. Además es preferible, dada la naturaleza del método de reconstrucción, restingirnos a cambios que solamente afecten al orden en que se procesen las proyecciones, y no a la forma del sistema de ecuaciones que se resolverá.

Una idea alternativa sería intentar que los rayos de proyecciones que se traten consecutivamente formen el mayor ángulo posible (intuitivamente, con esto se consigue que la información de las dos proyecciones sea significativamente distinta y se realizen correcciones en distintas posiciones del volumen). Hay estudios [GG94, Sor02] que se basan en esta idea, y es la que hemos adoptado nosotros.

En concreto, el preprocesamiento realizado consiste (dado un parámetro $n \in \mathbb{N}$) en:

1. Seleccionar la primera ($i = 0$) proyección aleatoriamente.
2. Elegir la proyección número $i > 0$ como la que maximize el menor ángulo con las $min(i, n)$ anteriores, de entre todas las restantes.

Normalmente el mejor valor para n (que indica cuántas de las proyecciones anteriores considerar al buscar la siguiente) es $n = 2$, dado que no puede haber más de tres vectores en espacio mutuamente perpendiculares.

Concluiremos esta sección observando que, en la práctica, este preprocesamiento no da lugar a mejoras significativas en la calidad de la reconstrucción cuando se dispone de datos tomados desde cientos de posiciones distintas. Su utilidad se ve limitada, por lo tanto, a los casos en los que sólo se parte de un pequeño número de proyecciones (del orden de las decenas).

Capítulo 5

Implementación en GPU

Código fuente en `http://gpd.sip.ucm.es/enrique/gpusart/`

Dedicaremos este capítulo a explicar brevemente las principales decisiones tomadas al portar nuestra aplicación a la GPU y las optimizaciones realizadas. Dado que los algoritmos siguen siendo básicamente los mismos, y que los detalles de implementación ya fueron expuestos en el capítulo anterior, no mostraremos pseudocódigo. En su lugar nos centraremos en las principales diferencias entre las versiones para CPU y GPU, y en las técnicas específicas que han sido necesario emplear para aumentar el paralelismo.

Uno de los principios que han guiado nuestra implementación es que tratamos de mejorar la *velocidad* de la reconstrucción; otros aspectos como el uso de memoria han sido considerados secundarios. Esto ha llevado a tomar decisiones en favor de la velocidad cuando ha existido un compromiso entre ambos. Una de estos casos es el referente a la estructura `rayosPorVoxel`, que almacena los rayos que intersecan a cada vóxel. Podría considerarse que no es estrictamente necesaria, puesto que para el proceso de backproyección se podría volver a calcular su contenido, y su consumo de memoria es bastante importante (tiene $4N^3$ elementos). Aun así se decidió incluirla, ya que el proceso de proyección (en el que se calculan estos vectores) es el que más tiempo de cómputo requiere (aproximadamente el 85 %). Otros ejemplos se refieren al tratamiento de vectores (que implementamos como arrays de un

tamaño máximo prefijado). Además, en algunas ocasiones hemos hecho los arrays de un tamaño mayor del necesario con el fin de hacer innecesarias las comprobaciones de acceso a fuera de rango y así acelerar el proceso.

La reconstrucción se lleva a cabo íntegramente en la GPU. Aunque en principio pudiera parecer que se obtendría un rendimiento mejor repartiendo el cómputo entre la CPU y la GPU, éste no es el caso en la práctica por una serie de razones. En primer lugar, una versión que se ejecuta solamente en la GPU es ya de por sí aproximadamente unas 20 veces más rápida que en la CPU que hemos utilizado; por tanto, la ganancia de velocidad al usar ambas sería mínima, aproximadamente un 5 % *como máximo*. Más aún, una cooperación entre la CPU y la GPU obligaría a transferir grandes cantidades de memoria entre ambas DRAMs, por lo que las hipotéticas ganancias perderían su valor. Por último, no se puede realizar un paralelismo a nivel de imagen (es decir, reconstrucción a partir de una proyección en la CPU mientras la GPU está tratando otras) sin cambiar la naturaleza del algoritmo, puesto que SART requiere una actualización de la estimación del volumen al *término de cada imagen*.

Por estos motivos, la CPU sólo se encargará de:

1. Leer los parámetros de la línea de comandos.

2. Ordenar las proyecciones si se requiere, de acuerdo al algoritmo expuesto en la sección 4.8.

3. Llamar a unas funciones de inicialización que se ejecutan en la GPU y se encargan de reservar la memoria necesaria en la GPU y almacenar los datos globales (tamaños de las proyecciones).

4. Reservar memoria para el volumen.

5. Leer el contenido de cada una de las proyecciones del disco y transferir la información de cada una de éstas a la GPU. (Nótese que la cantidad de información que se trasfiere es mucho menor que si hubiera que mover el volumen). El proceso se bloquea hasta que la GPU acaba la backproyección correspondiente a esa imagen, y se pasa a la siguiente.

Al final de todo el procedimiento, y sólo al final, la GPU copiará el volumen reconstruido a la dirección indicada (la reservada en el punto 4).

5.1. Estructura general

Los datos se guardan en coma flotante de simple precisión (*float*), dado que es la mayor precisión soportada por CUDA hasta la fecha. En general esto no afecta de manera significativa a la calidad de las reconstrucciones obtenidas (véase el capítulo siguiente, sección 6.1), aunque nos hemos visto forzados a utilizar un valor relativamente elevado ($\sim 10^{-5}$) del ε necesario para permitir la comparación adecuada de valores de coma flotante. Esto se debe a que los *floats* disponen únicamente de 24 bits de mantisa (lo que da una precisión relativa máxima de $\sim 10^{-7}$. El primer paso fue asegurarnos, con nuestra implementación en CPU, de que este cambio no afecta de manera notable a la calidad de reconstrucción.

Las estructuras de datos utilizadas internamente por la GPU son:

- `myvector<inforayo>rayosporvoxel[Zdim][Ydim][Xdim]`: contiene un vector que indica los pares (rayo, integral de línea) de todos los rayos que intersecan a ese vóxel. `inforayo` es una estructura que almacena esta información y está convenientemente alineada en memoria para acelerar los accesos.

- `float proyeccion[Ydim][Xdim]`: inicialmente es el valor de la proyección leída de disco. Después del proceso de proyección, contendrá las diferencias normalizadas entre las proyecciones leídas y las calculadas. (es decir, integramos el paso de corrección en la proyección, ya que se dispone de toda la información necesaria).

- `float volumen[Zdim][Ydim][Xdim]`: estimación del volumen. Se inicializa a 0 y se actualiza después de procesar cada imagen.

- `float2 slice[Zdim][Ydim][Xdim]`: contiene la proyección de cada "rodajada" (*slice*) del volumen. Estas rodajas, que se usan sólo en la proyección dirigida por vóxeles, son necesarias para poder llevar a cabo el *splatting*;

su uso se discutirá en la sección 5.2.2. *float2* es un tipo de datos proporcionado por CUDA que permite manejar pares de *floats* con una mayor eficiencia.

Para que sean posibles accesos a memoria con una sola instrucción, cada elemento de las estructuras que utilizamos tienen tamaño de 4, 8 ó 16 bytes, y están alineados a direcciones múltiplos de 4.

Las dimensiones mostradas son puramente "lógicas"; en realidad en algunos casos se reserva más tamaño del estrictamente necesario con el fin de poder evitar *bounds checking* en ciertas situaciones. Todos estos arrays residen en la memoria *global*, a pesar de su gran latencia. La razón es que, con excepción de la proyección (y sólo para tamaños moderados), no es posible almacenarlas en otros tipos de memoria (*shared* o de texturas) debido a que su tamaño máximo es de 16Kb (entre todos los bloques). Por otro lado, no existe reuso de datos (cada hilo lee o escribe una sola vez en cada posición) ni cooperación entre los hilos, por lo que los patrones de acceso no son los adecuados para que sea posible aprovechar las características de estos tipos de memoria. Por ejemplo, la proyección por rayos accede una sola vez por hilo a cada posición de `proyeccion`. La única posible excepción sería el uso de la memoria de texturas para precalcular las integrales de línea en función de la distancia al centro del vóxel (en los casos donde se usen funciones base con simetría esférica, por ejemplo *blobs*); a esta tabla precalculada se le suele referir con el nombre de *footprint* [MY99]. Sin embargo, veremos posteriormente que el hecho de que el cuello de botella actual de nuestra implementación esté en los accesos a memoria nos permite calcular estas integrales de línea en tiempo real *sin pérdida alguna de velocidad y con mayor precisión* (dado el limitado tamaño que este *footprint* puede tener).

Los vectores (aquí denotados por `myvector`) son arrays de tamaño máximo fijo, ya que no se permite reservar memoria dinámicamente dentro de un *kernel*. Además tener que mover los datos de la memoria cada vez que el tamaño del vector excediera su capacidad supondría una importante pérdida de rendimiento (máxime cuando, como veremos, el cuello de botella de nuestra aplicación son los accesos a memoria). Esto no supone ninguna pérdida de aplicabilidad, pues como se discutió en 4.4, el máximo número de

rayos que puede atravesar a un vóxel dado es de 4. (Si permitiéramos vóxeles cuyo lado fuera algo menor que el de los rayos, este número aumentaría, pero en cualquier caso hay una cota superior no demasiado alta, pues como ya vimos carece de sentido utilizar lados de vóxel demasiado pequeños).

El acceso a cada array se realiza de manera lineal, de tal manera que, por ejemplo, el punto (x, y, z) corresponde a la posción $(Zdim \cdot Ydim \cdot z + Ydim \cdot y + x)$. Es importante, sin embargo, organizar los accesos a memoria de tal manera que hilos del mismo bloque accedan a posiciones consecutivas (desafortunadamente, no es posible conseguir esto para accesos al volumen y a la proyección al mismo tiempo). Con esto se posibilita que los accesos a memoria de hilos ejecutándose concurrentemente se mezclen en uno sólo. Como veremos, esto se consigue indexando los hilos de la manera adecuada.

En cuanto al consumo de memoria, está dado (aproximadamente) por $68n^3 + 4n^2$ bytes, por lo que un simple cálculo muesta que el tamaño máximo de una proyeción cuadrada que podemos manejar es de 226. Se podrían manejar, sin embargo, tamaños considerablemente mayores si eliminamos la estructura de `rayosPorVoxel`, a costa de una penalización en eficiencia.

Dejando de lado las partes de inicialización y finalización y algunas tareas de *bookkeeping* (reserva de memoria, copia del volumen reconstruido a la RAM de la CPU, elección de los mejores parámetros de hilos por bloque), la mayor parte del código a ejecutar en la GPU consiste en una serie de *kernels* (ver capítulo 3). Cada uno de estos *kernels* se ejecuta de manera independiente en un hilo distinto. Los *kernels* existentes son los que se encargan de la proyección (bien con cada hilo siguiendo un rayo o proyectando un vóxel), la backproyección y la suma de *slices* (que es un proceso adicional necesario para una eficiente implementación dirigida por vóxeles).

La información que la CPU transmite a la GPU es:

- Tamaño de las proyecciones y del volumen a reconstruir.
- Dirección de memoria de la CPU donde escribir el volumen.
- Proyección medida a procesar.

- Matrices de Euler de la rotación utilizada.

- Parámetros configurables: factor de relajación, tipo de proyección (*ray-driven* o *voxel-driven*), configuraciones deseadas del número de hilos por bloque a usar en los kernels, etc.

5.2. Proyección y cálculo de las correcciones

Como ya se ha dicho, en la implementación para GPU hacemos los procesos de proyección y cálculo de correcciones al mismo tiempo. Sabemos que existen, en principio, dos enfoques distintos para realizar la proyección (dirigido por rayos y dirigido por vóxeles), y cabe preguntarse cuál es mejor.

La idea principal es que, en el primer caso, cada hilo se encargaría independientemente de seguir la trayectoria de un rayo y calcular los vóxeles que interseca y sus respectivas integrales de línea; en el segundo caso cada hilo se encargaría de proyectar el centro de un vóxel y comprobar los rayos que pueden intersecar.

Podría parecer por tanto que la proyección por vóxeles dispone de mucha mayor paralelismo (hay N^3 vóxeles y sólo N^2 rayos), pero este razonamiento ignora los problemas de concurrencia que pueden surgir. En efecto, por término medio $O(N)$ vóxeles se proyectan en mismo píxel (en realidad esta es sólo una cota superior, pero se alcanzará para la mayoría de las distribuciones "razonables" de ángulos, incluyendo las nuestras, en las que el rayo del centro de la imagen siempre pasa por el centro del volumen). Los hilos correspondientes a estos vóxeles deberían escribir todos ellos en la misma posición de memoria (referente a la proyección), posiblemente de manera concurrente. Debido a la virtual carencia de mecanismos de sincronización en CUDA, esta es una situación a evitar. La única primitiva de sincronización, *__syncthreads()*, bloquea todos los hilos de un mismo bloque hasta que todos hayan llegado a esa barrera. Dejando de lado la ralentización producida por esta espera, no podríamos ni siquiera sincronizar hilos de distintos bloques, por lo que el número de *kernels* que podríamos estar ejecutando de manera independiente en un momento dado se vería

drásticamente reducido (a 512).

Además, sería necesario un proceso de *clipping para cada vóxel*. Por las razones explicadas en el capítulo 3, un algoritmo que requiera una gran cantidad de distinciones de casos puede ralentizar de manera apreciable el rendimiento en la GPU. Esto es así puesto que fuerza a que la ejecución de varios hilos pertenecientes al mismo *warp*, que de otra manera estarían ejecutando la misma instrucción SIMD, sean serializados.

Por estas razones optamos por comenzar con una implementación dirigida por rayos. Sin embargo, como veremos, todas estas dificultades pueden solventarse y su resolución da lugar a una implementación ligeramente superior a la dirigida por rayos (la diferencia es más notable conforme los tamaños de los volúmenes a reconstruir aumentan). Empezaremos describiendo la aproximación dirigida por rayos.

5.2.1. Proyección dirigida por rayos

El primer paso de este tipo de proyección es borrar el contenido del array `rayosPorVoxel`, antes de que los hilos del *kernel* principal realicen la proyección propiamente dicha. Esta operación se realiza trivialmente y de forma totalmente paralela. Lo único que merece la pena comentar es que el tamaño de `rayosPorVoxel` se redondea al siguiente múltiplo de 512 para acelerar este proceso y no tener que comprobar índices (ya que por la simplicidad de dicha operación se puede alcanzar el máximo de 512 hilos por bloque).

Cada uno de los *kernels* empieza realizando un recorte del rayo considerado con el volumen. Se utiliza el mismo algoritmo (Cohen-Sutherland) cuyo pseudocódigo se muestra en la figura 4.5, con la única salvedad de que se ha desenrollado el bucle. En cualquier caso, esta operación de *clipping*, en principio costosa, no resulta en una degradación apreciable del rendimiento, puesto que la mayor parte del tiempo de ejcución con este enfoque se realiza al recorrer el rayo.

Cada rayo escribe la proyección final en una posición distinta (en realidad se almacena la corrección, ya normalizada, pero esto carece de importancia

para la discusión que sigue), por lo que no hay problemas de concurrencia al acceder a la matriz de proyección. Sin embargo, sí surgen al acceder a `rayosPorVoxel`: cada vez que el rayo atraviesa un nuevo vóxel, este hilo debe escribir en `rayosPorVoxel`, y ciertamente puede haber rayos distintos que atraviesen el mismo vóxel. La clave está en en que esto no puede ocurrir para un par de rayos suficientemente lejanos entre sí. En particular, dado que la diagonal del cubo es $\sqrt{3} \simeq 1{,}732 < 2$, es imposible que dos rayos a distancia ≥ 2 corten al mismo vóxel. Esto nos lleva a la siguiente solución: dividamos la imagen de proyección en grupos de 2×2 píxeles, y realicemos la proyección en cuatro fases: en la fase dada por (i, j), $0 \leq i < 2, 0 \leq j < 2$, proyéctense únicamente los rayos tales que $Y \equiv i \pmod 2$, $X \equiv j \pmod 2$ (ver figura 5.1). Este esquema garantiza que no habrá accesos concurrentes, y en cada una de las fases se lanzarán $N^2/4$ *kernels* totalmente independientes.

1	2	1	2
4	3	4	3
1	2	1	2
4	3	4	3

FIGURA 5.1: División en grupos de 2×2 píxeles; en cada fase se procesan los píxeles marcados con el mismo número.

La indexación de estos *kernels* se realiza de manera lineal, en el mismo sentido en el que están ordenados el volumen y `rayosPorVoxel` (direcciones consecutivas corresponden a aumentos en X; direcciones separadas por $Xdim$ vóxeles corresponden a aumentos en Y, y análogamente para Z). Nótese que, puesto que el número de vóxeles que corta un rayo es variable, algunos hilos finalizarán antes que otros, reduciendo la efectividad de la ejecución SIMD. Sin embargo, si el número de hilos por bloque no es muy elevado (como es el caso en la práctica, puesto que el número máximo de hilos por bloque está limitado en función del número de registros que usa el

kernel), los hilos en el mismo bloque seguirán trayectorias parecidas y sus longitudes serán similares. Los accesos a `proyeccion`, por su parte, son obviamente en regiones consecutivas en memoria (ya que cada hilo accede solamente a su posición correspondiente).

Como optimización menor, comentaremos que se obtuvo un aumento del 10 % en velocidad al manejar directamente la memoria asociada a los vectores en lugar de utilizar la interfaz general que habíamos diseñado para ocultar sus detalles de implementación (lo que una vez más pone de manifiesto la dependencia del rendimiento sobre los accesos a memoria). Además, un factor adicional ha sido el uso de las funciones matemáticas rápidas proporcionadas por CUDA.

5.2.2. Proyección dirigida por vóxeles

Hemos notado anteriormente que este enfoque presenta, en principio, mayores problemas de paralelismo. Vamos a ver, sin embargo, que el uso de *splatting*, junto a la misma idea del 2×2 puede aplicarse para resolver estas dificultades. El *splatting* consiste en dividir el volumen en rodajas (*slices*) y para cada una de ellas calcular la imagen proyección. Por ejemplo, en nuestro caso se divide el volumen en N *slices* de $N \times N$ cada uno, calculándose sus proyecciones. Una vez se han obtenido, hace falta un último paso para sumarlas todas y obtener la proyección real de todo el volumen.

El objetivo es realizar este *slicing* de tal manera que se minimicen los problemas de concurrencia, causados por dos vóxeles a los que corta un mismo rayo. La peor situación posible sería que los rayos fueran paralelos a las *slices*; en ese caso, cada rayo atravesaría una gran cantidad de vóxeles pertenecientes a la misma *slice* y no podrían proyectarse independientemente dentro de cada *slice*. Parece natural, por tanto, elegir la dirección del *slicing* de tal forma que el plano de cada rodaja forme el mayor ángulo posible con los rayos (ver figura 5.2). Para realizar este particionamiento de la manera más regular, eficiente y cómoda posible, sólo consideraremos *slices* que sean paralelas a uno de los planos XY, XZ y YZ; seleccionemos entonces el plano que forme un mayor ángulo con los rayos, y sean nuestras *slices*

paralelas a este plano. Veremos a continuación que con esta elección, y aplicando una división de cada *slice* en cuatro grupos dependiendo de la posición de cada vóxel en su grupo de 2×2 análoga a la detallada en la sección anterior, se garantiza que dos de los *kernels* que estén activos en un momento dado proyectarán sus respectivos vóxeles a píxeles distintos.

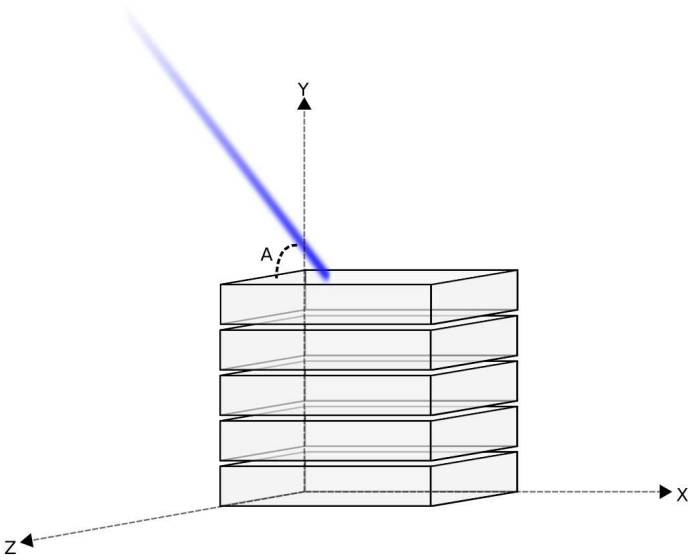

FIGURA 5.2: *Splatting*

Supongamos que la dirección del rayo está dada por el vector unitario $\vec{v} = (x, y, z)$. El ángulo de v con la recta determinada por el eje X es $\arccos|x|$, y análogamente para los otros dos planos. Como $x^2 + y^2 + z^2 = 1$, se tiene que $max(|x|, |y|, |z|) \geq \frac{1}{\sqrt{3}}$, de lo que se sigue que el mayor de los ángulos con los planos XY, XZ y YZ es al menos de $90\,^\circ - \arccos(\frac{1}{\sqrt{3}}) \simeq 35{,}26\,^\circ$, como se aprecia en la figura. (El caso peor se da cuando el rayo es paralelo a una diagonal de los vóxeles). Supongamos, por ejemplo, que las rodajas se hacen an la dirección del eje X. Cada *slice* está dada por $x = k$, $k \in 1..N$. Si seguimos la trayectoria del rayo, cada vez que la parte entera de la coordenada X de un punto suyo cambia de valor, las coordenadas Y y Z han cambiado en menos de una unidad (puesto que $|x| \geq max(|y|, |z|)$). Esto quiere decir que el rayo sólo puede cortar a dos

vóxeles adyacentes de una misma *slice*, lo que justifica el particionamiento en grupos de 2×2.

El último paso necesario es el de reducción: acumular las proyecciones de cada una de las *slices*. Este proceso puede realizarse en $\theta(logN)$ pasos. En el primer paso, N^2 *kernels* suman los valores asociados a cada dos *slices* consecutivas; en el siguiente se suman los pares de valores resultantes (que por tanto acumulan las sumas de cada grupo de 4 consecutivas), y así sucesivamente; ver figura 5.3). El coste de este paso está por tanto en $\theta(N^2logN)$, por lo que no varía la complejidad del proceso global.

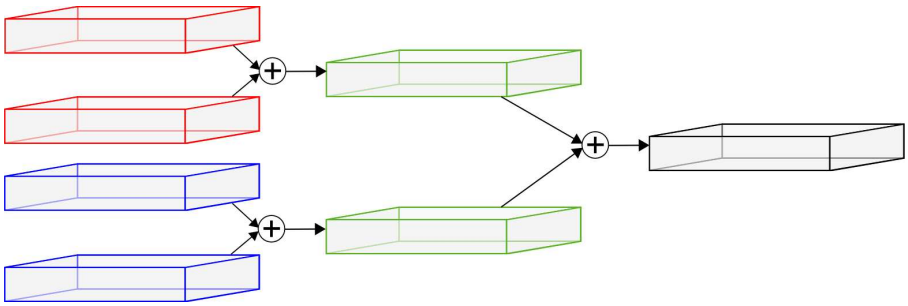

FIGURA 5.3: Paso de reducción

Entre las optimizaciones menores realizadas se cuenta el pasar sólo dos filas de la matriz de Euler (ya que sólo hacen falta esas dos para la proyección, y esto reduce el número de registros que usa cada *kernel* de proyección).

5.3. Backproyección

La operación de backproyección no presenta dificultades debido a su alto paralelismo, ya que si se lanza únicamente un hilo por vóxel, el hilo correspondiente sólo necesita acceder a los valores del volumen y `rayosPorVoxel` correspondientes a ese vóxel. Por esta razón es responsable de sólo el 10 % del tiempo de ejecución total. La única traba que impide que cada hilo se ocupara sólo de un vóxel es la limitación de 65535 bloques de hilos. Por ese motivo existe un parámetro k tal que cada *kernel* se ocupa de k vóxeles. Este valor se calcula en función de las dimensiones del volumen y

de la limitación comentada.

Una optimización para el proceso es organizar los accesos a memoria en grupos de 16 bytes, dado que la GPU es capaz de leerlos de la memoria global en una sola instrucción. Sin embargo, no se observaron mejoras significativas, debido probablemente a que esta optimización fue realizada previamente por el compilador de NVIDIA.

5.4. Consideraciones finales

Cabe destacar que, a pesar del considerable *speedup* conseguido (ver capítulo siguiente), no se llega a alcanzar la ocupación máxima de los multiprocesadores en ninguna de las dos versiones (una ocupación máxima no garantiza un mayor rendimiento final, pero puede ser un buen indicador). Esto sólo ocurre en el paso de proyección (y no para el mucho más sencillo proceso de backproyección, que sí puede alcanzar la máxima ocupación). Este impedimento es debido en gran medida al elevado número de registros que usan los *kernels* de proyección, dado que los registros del multiprocesador deben dividirse entre todos los *threads* que se estén ejecutando de manera concurrente. El control del número de registros es pues de la máxima importancia al programar la GPU: un número muy alto de registros limita el máximo número de hilos que puede haber en un bloque, y si se rebasa este valor el *kernel* no será lanzado. Esto claramente dificulta el paralelismo. Nosotros utilizamos 19 registros para la proyección dirigida por rayos y 26 para la dirigida por vóxeles. La diferencia entre ambas cantidades es debida al hecho de que calculemos en tiempo de real las integrales de línea de los *blobs*, dadas por la ecuación 2.8. Tal procedimiento no ralentiza en absoluto la reconstrucción por vóxeles debido a que estamos limitados por la latencia de memoria.

Se ha realizado un esfuerzo para reorganizar el código con el fin de reducir significativamente el número de registros empleado por ambos *kernels*, pero las propias características del algoritmo hacen imposible una reducción significativa. Esto es así ya que existe una gran cantidad de estructuras distintas que el *kernel* debe tratar: datos y dimensiones del volumen,

proyección leída, `rayosPorVoxel`, matriz de Euler de la rotación, identificador de hilo, posición dentro del bloque de 2×2, parámetros de hilos por bloque y, en el caso de la proyección por vóxeles, dirección de la matriz de *splatting* y orientación según la cual se ha realizado el *slicing*.

Algunos de estos parámetros (en particular la posición dentro del bloque de 2×2 y la dirección de *slicing*) pueden suprimirse simplemente teniendo varias versiones del mismo *kernel*, pero la realización de tales optimizaciones no ha proporcionado resultados positivos.

El compilador de CUDA se encarga de reutilizar los registros de una manera bastante eficiente si los resultados de cálculos previos no se reutilizan (esto se ha comprobado experimentalmente). Por lo tanto, concluimos que simplemente por el hecho de tener que tratar con toda esta información y tener que almacenar algunos resultados intermedios, el número de registros no puede reducirse significativamente por debajo de 19.

Capítulo 6

Resultados experimentales

Una vez implementadas las dos variantes del algoritmo, tanto en CPU como en GPU, llega el momento de obtener y evaluar los resultados. Aunque el principal objetivo del proyecto es conseguir optimizar el tiempo necesario para formar la reconstrucción, hay otros factores importantes. Entre ellos destacan, por su importancia, la medición de la calidad de las reconstrucciones conseguidas y la variación de la misma respecto al factor de relajación λ, aunque también hay otros resultados interesantes. En primer lugar se comenzará analizando la calidad de las reconstrucciones obtenidas, para una vez constatada ésta continuar con el *speedup* y otros resultados.

6.1. Calidad de las reconstrucciones

Aunque el objetivo principal ha sido reducir el tiempo de ejecución, la calidad de las reconstrucciones es un factor importantísimo del proceso. De poco sirve conseguir un *speedup* muy alto si después las reconstrucciones son manifiestamente peores que las que se consiguen con otros programas más lentos.

6.1.1. Calidad visual

Para medir la calidad de las reconstrucciones primero se mostrarán imágenes de las reconstrucciones conseguidas por Xmipp y nuestras dos variantes del

algoritmo SART, junto con su volumen original. Aunque se ha trabajado con varios volúmenes y *phantoms* durante el año, esta primera parte se centra en dos: el antígeno T y el complejo DnaB-DnaC. Estos dos ejemplos de prueba han sido descargados de la página web oficial de Xmipp [Xmi].

A la hora de visualizar los volúmenes se han utilizado dos programas distintos. Uno es el visualizador que forma parte de Xmipp, *xmipp_show*, que nos permite ver el volumen *slice* a *slice* (esto es, se muestran una serie de secciones para cada valor de la coordenada Z). El otro es un famoso visualizador y analizador molecular llamado *UCSF Chimera* [UCS], que representa el volumen en 3 dimensiones posibilitando girarlo con el ratón para observar más fácilmente su estructura espacial.

El volumen original del antígeno T se muestra en la figura 6.1, y la reconstrucción realizada por el programa de Xmipp *xmipp_art* [MHC98] en la figura 6.2. La reconstrucción obtenida con nuestro algoritmo, usando proyección dirigida por rayos y vóxeles como función base, es la mostrada en la figura 6.3; y la resultante de utilizar proyección dirigida por vóxeles y *blobs* como función base se muestra en la figura 6.4. Para los cuatro volúmenes se ha utilizado el visualizador que forma parte de Xmipp. En las tres reconstrucciones se ha utilizado un valor del factor de relajación de 1 (el mejor valor de los probados) y un conjunto de 250 proyecciones de 64×64 píxeles. Además, el programa reconstructor de Xmipp se ha usado activando la opción de utilizar el método SART, para que el método utilizado para obtener las reconstrucciones sea el mismo.

Como se puede observar, ninguna de las dos reconstrucciones llega a conseguir un nivel de detalle tan bueno como el volumen original. La reconstrucción de Xmipp es algo "borrosa" pero tiene poco *salt and pepper noise* (aparición de píxeles de color bastante distinto al de sus vecinos). Respecto a las dos variantes de nuestro algoritmo, con ambas se obtiene una reconstrucción muy parecida. Son más nítidas que las de Xmipp, pero se les observa una mayor cantidad de *salt and pepper noise*, un poco más presente en la reconstrucción que utiliza *blobs* y proyecta por vóxeles. Sin embargo, en ambas se aprecia claramente la forma del antígeno T.

Respecto al complejo DnaB-DnaC, las reconstrucciones son mostradas con el

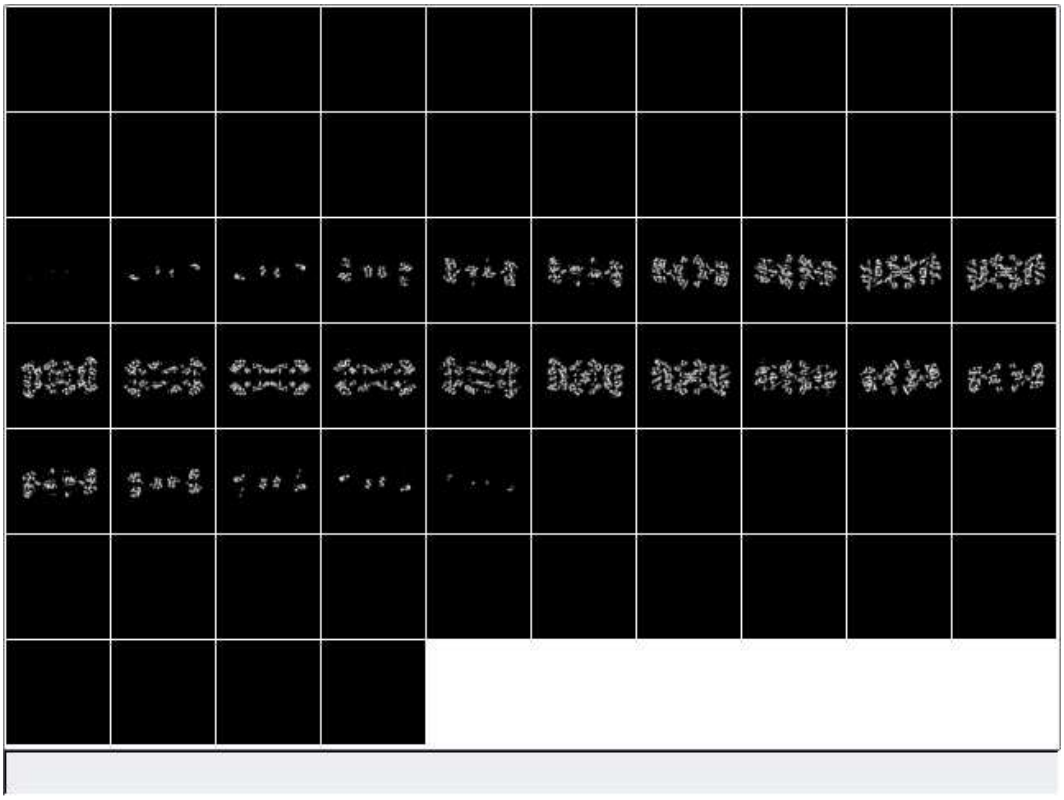

FIGURA 6.1: Volumen original del antígeno T

visualizador molecular *UCSF Chimera*. El volumen original puede observarse en la figura 6.5, y la reconstrucción obtenida con el programa de Xmipp en la figura 6.6. Las reconstrucciones conseguidas con nuestro algoritmo se muestran en las figura 6.7 y 6.8; la primera ha sido producida utilizando proyección dirigida por rayos y vóxeles como funciones base, y la segunda usando proyección dirigida por vóxeles y *blobs* como funciones base. En los tres casos se ha utilizado un valor de 0,01 para el factor de relajación, e igual que en la anterior prueba el reconstructor de Xmipp ha sido utilizado activando el método SART. El conjunto de proyecciones de entrada ha estado formado en todos los casos por 6995 imágenes de 81×81 píxeles. Como se puede observar, las dos reconstrucciones obtenidas con nuestro algoritmo son muy parecidas. Ambas tienen una forma muy similar a la original, diferenciándose principalmente en que la obtenida mediante

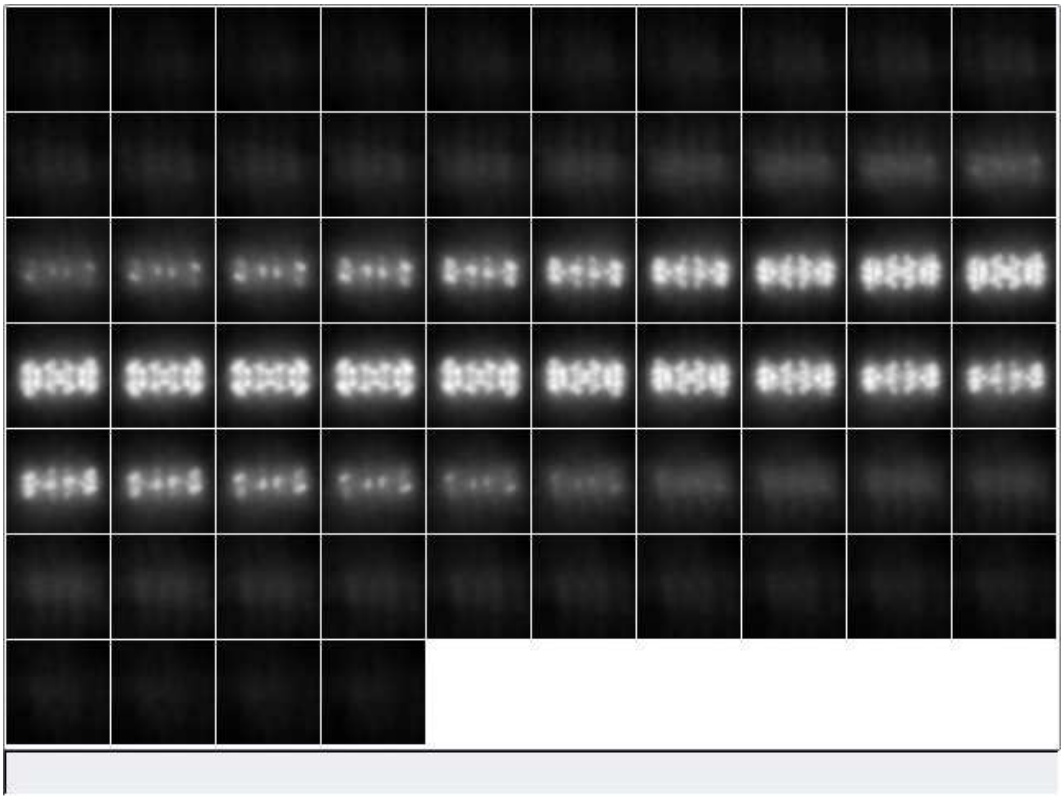

FIGURA 6.2: Antígeno T reconstruido utilizando la suite Xmipp

proyección dirigida por rayos (y que usa vóxeles como función base) tiene unas formas más suaves que la otra, aunque no es tan cercana a la forma del complejo original. En cambio, la conseguida utilizando proyección dirigida por vóxeles y *blobs* como función base tiene una forma más definida pero algunas zonas poco suaves. La reconstrucción generada con el reconstructor de Xmipp es la que formas más suaves tiene, aunque la forma es la que menos se parece al original.

6.1.2. Error cuadrático medio

Aunque comprobar la calidad visual de las reconstrucciones obtenidas es importante, se necesita también un resultado numérico que la apoye. Hay multitud de opciones para medir la calidad de una reconstrucción [SMB+01],

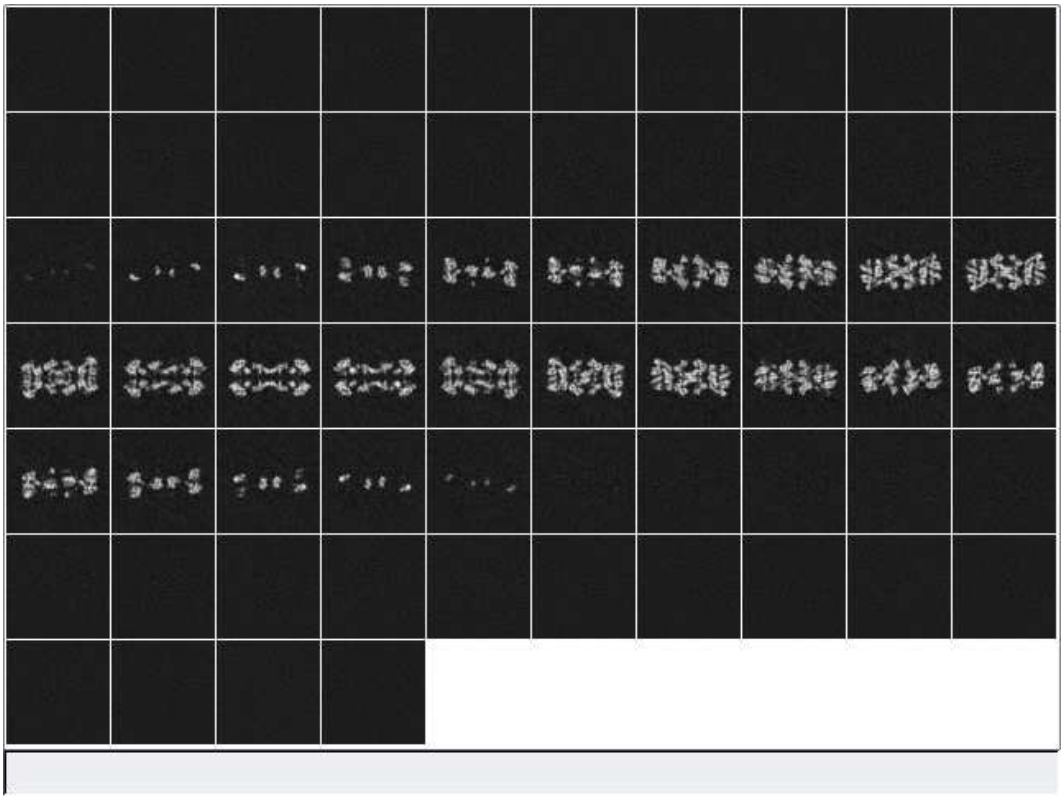

FIGURA 6.3: Antígeno T reconstruido usando proyección dirigida por rayos y vóxeles como función base

como pueden ser el error absoluto medio, la densidad media, el error en las desviaciones típicas... Una de las más utilizadas, y la que se ha elegido en este proyecto, ha sido el error cuadrático medio, pues da más importancia a los errores grandes. Para calcular este error, una posibilidad es utilizar las diferencias entre los valores de las distintas proyecciones de la reconstrucción resultante y lo que debería obtenerse según las proyecciones de entrada, es decir, cómo de bien se ajusta al sistema de ecuaciones original. Esta es una buena opción, y es la única posible cuando no se dispone del volumen original con el que comparar, y por esta razón el programa reconstructor de Xmipp calcula el error cuadrático medio basándose en esto. Pero si se dispone del volumen original, es mejor opción utilizarlo para calcular el error. Ésta ha sido nuestra elección, ya que en todos los casos hemos tenido

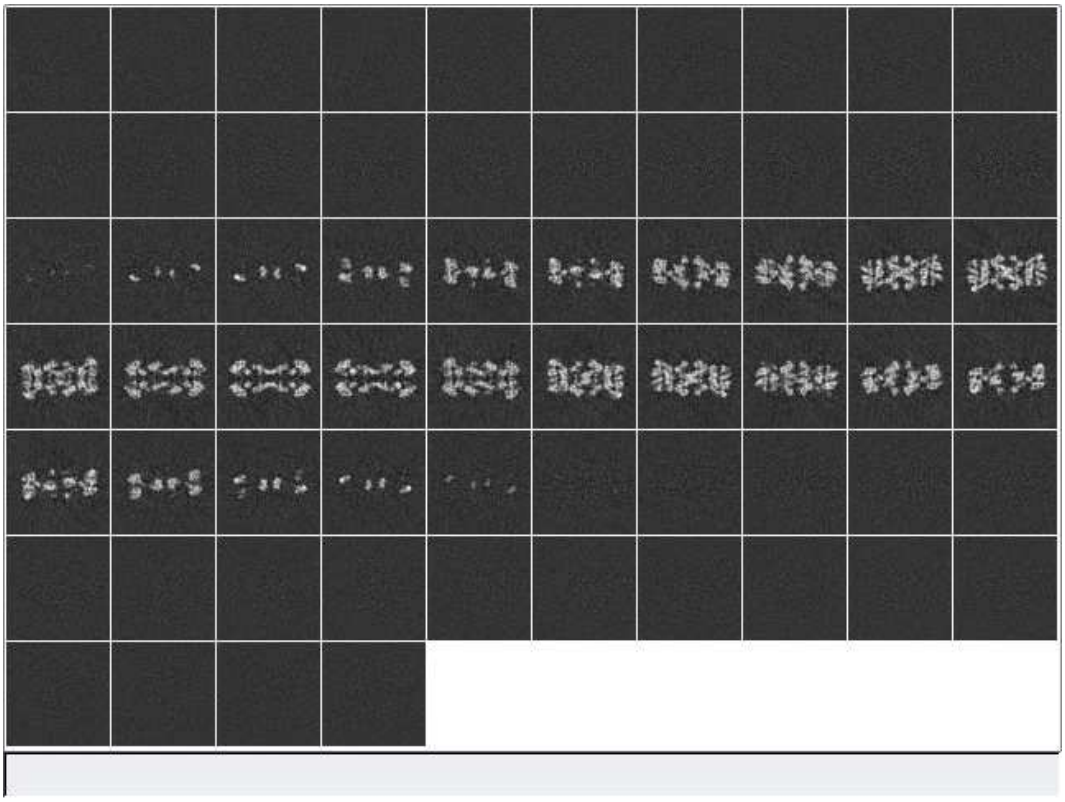

FIGURA 6.4: Antígeno T reconstruido usando proyección dirigida por vóxeles y *blobs* como función base

tenido acceso a dichos volúmenes originales. Para la medición de este error, hemos implementado un pequeño programa de comparación de volúmenes que actúa de la siguiente forma:

1. Normaliza el volumen original para que sus valores estén entre 0 y 1. Este paso no es necesario para comparar distintas reconstrucciones de un mismo volumen; se incluye sólo con el significado de intentar dar un significado "absoluto" a los errores, incluso con distintos volúmenes originales.

2. Busca los valores a y b tales que minimizan el error cuadrático medio tras aplicarle al volumen reconstruido una transformación lineal del tipo $ax + b$.

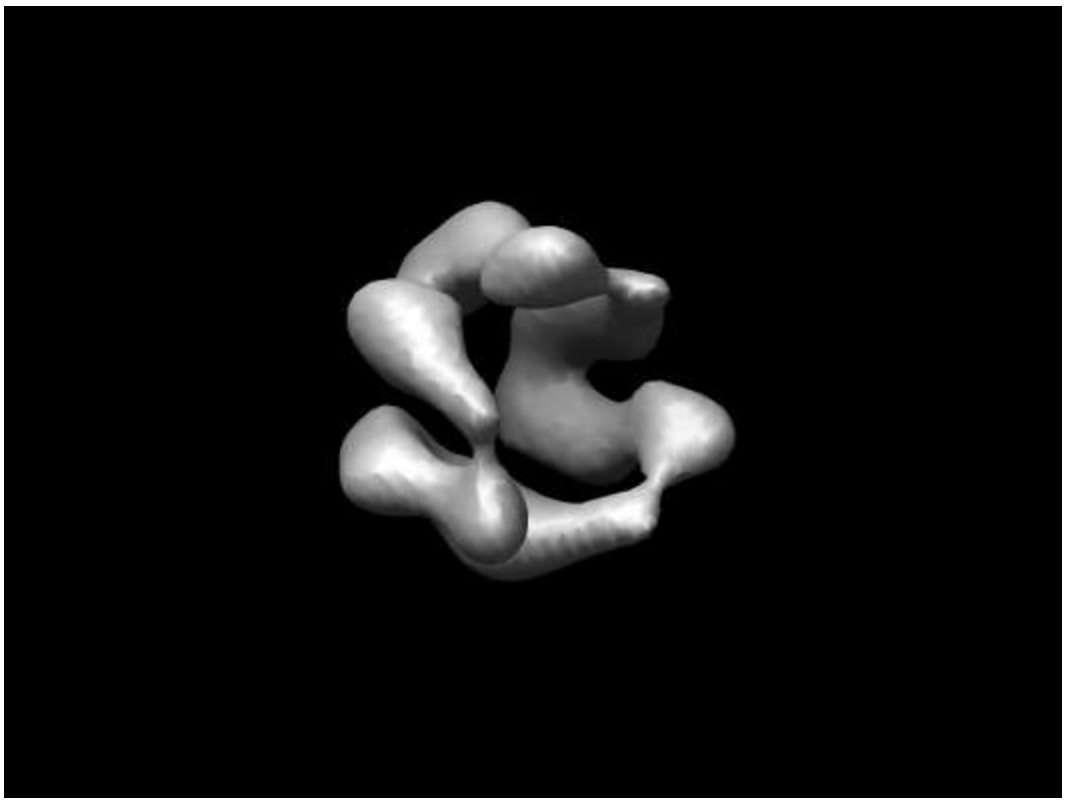

FIGURA 6.5: Volumen original del complejo *DnaB-DnaC*

3. Utilizando esos valores de a y b, calcula el error cuadrático resultante.

La inclusión de esta mejor aproximación lineal al calcular el error cuadrático medio se debe a que un volumen sigue mostrándose igual si se le aplica cualquier transformación lineal. Aunque en principio pueda parecer que no es posible comparar dos volúmenes así si utilizan distintas funciones base, sí es posible. Cuando los dos volúmenes usen la misma función no habrá ningún problema, y si uno usa vóxeles y otro *blobs* habrá que entender este error cuadrático como el error restringiéndose al muestreo de los centros de los vóxeles (ya que tanto los vóxeles como los *blobs* valen 1 en el centro).

Se ha utilizado este método de medición del error para comprobar la calidad de las reconstrucciones de los ejemplos anteriores y otros dos nuevos. Los resultados se pueden ver en la tabla 6.1 para los algoritmos que usan CPU, y

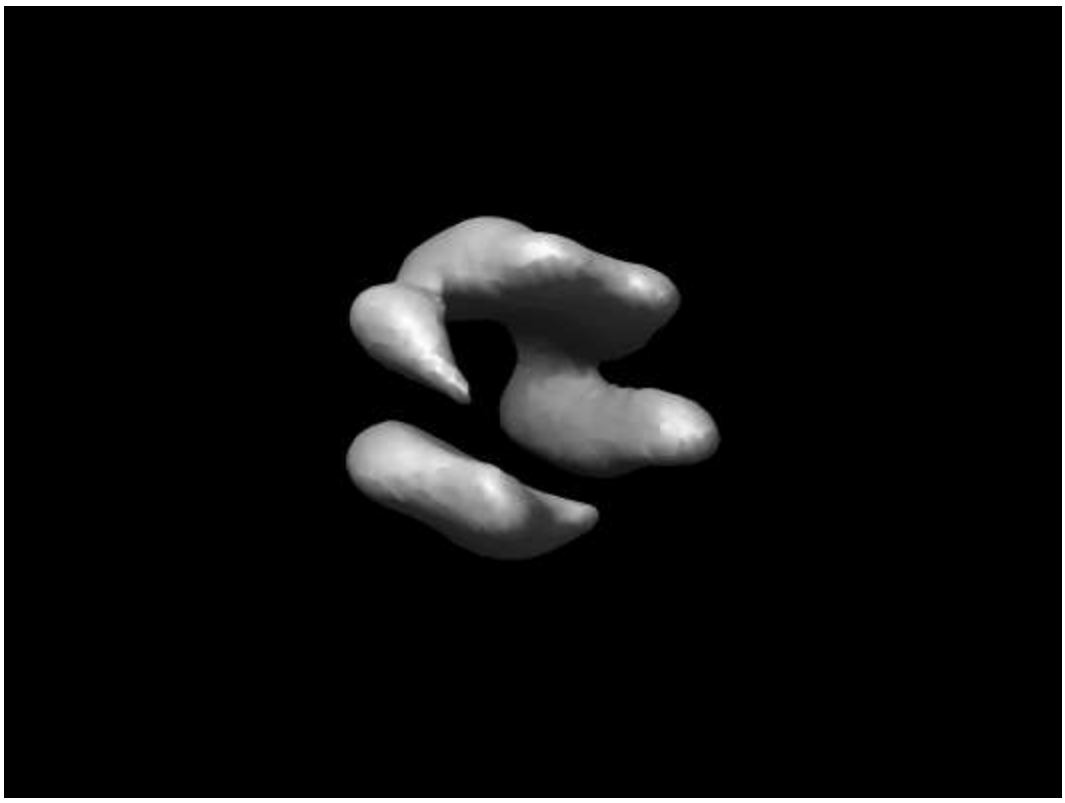

FIGURA 6.6: *DnaB-DnaC* reconstruido utilizando la suite Xmipp

en la tabla 6.2 para los algoritmos que corren sobre la GPU. Como se puede observar, para una misma función base (*blobs* o vóxeles) las versiones que utilizan CPU y GPU cometen un error muy similar. Hay una excepción, que aparece cuando utilizamos vóxeles en el volumen *DnaB-DnaC*. En este caso el error casi se duplica al pasar de la versión de CPU (0,001478) a la versión de GPU (0,002817). Aunque en principio podría parecer que los errores no deberían variar tanto según el procesador utilizado, ya que es el mismo método en los dos, hay que tener en cuenta la precisión con la que se hacen las operaciones. En CPU se usan *doubles*, variables de punto flotante de doble precisión, mientras que en la GPU sólo se pueden utilizar *floats*. Aunque en general esta diferencia suele pasar desapercibida, en reconstrucciones tan intensivas como la de *DnaB-DnaC* (que, recordemos, son 6995 proyecciones de 81×81 píxeles) dicha diferencia

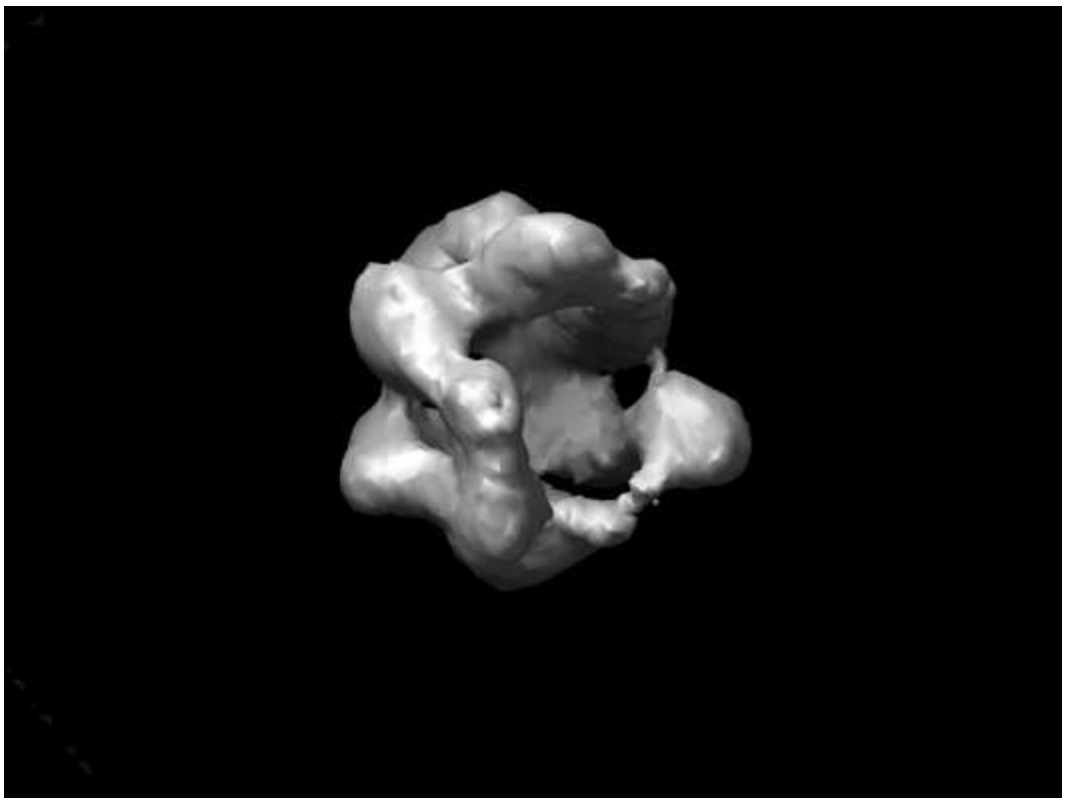

FIGURA 6.7: Complejo DnaB-DnaC reconstruido usando proyección dirigida por rayos y vóxeles como función base

se hace más patente.

También se observa que el error cometido por nuestro algoritmo es en todos los casos inferior al cometido por el reconstructor de Xmipp, salvo en el mismo caso de la reconstrucción de *DnaB-DnaC*. Aunque esto no se percibe en las imágenes del apartado de 6.1.1 (por razones de tamaño no se ha incluido el volumen entero), el error cometido se localiza en las proximidades a las aristas del volumen. Durante la reconstrucción de este volumen se generan una serie de *artefactos* a lo largo de las aristas del volumen que contribuyen con bastante peso al error global. Aunque estas *artificialidades* son poco deseables, que se generen solamente en las aristas y lejos del "volumen" real reconstruido hacen que su importancia sea menor. Más aún, con toda probabilidad son un efecto del algoritmo en sí, unido a la

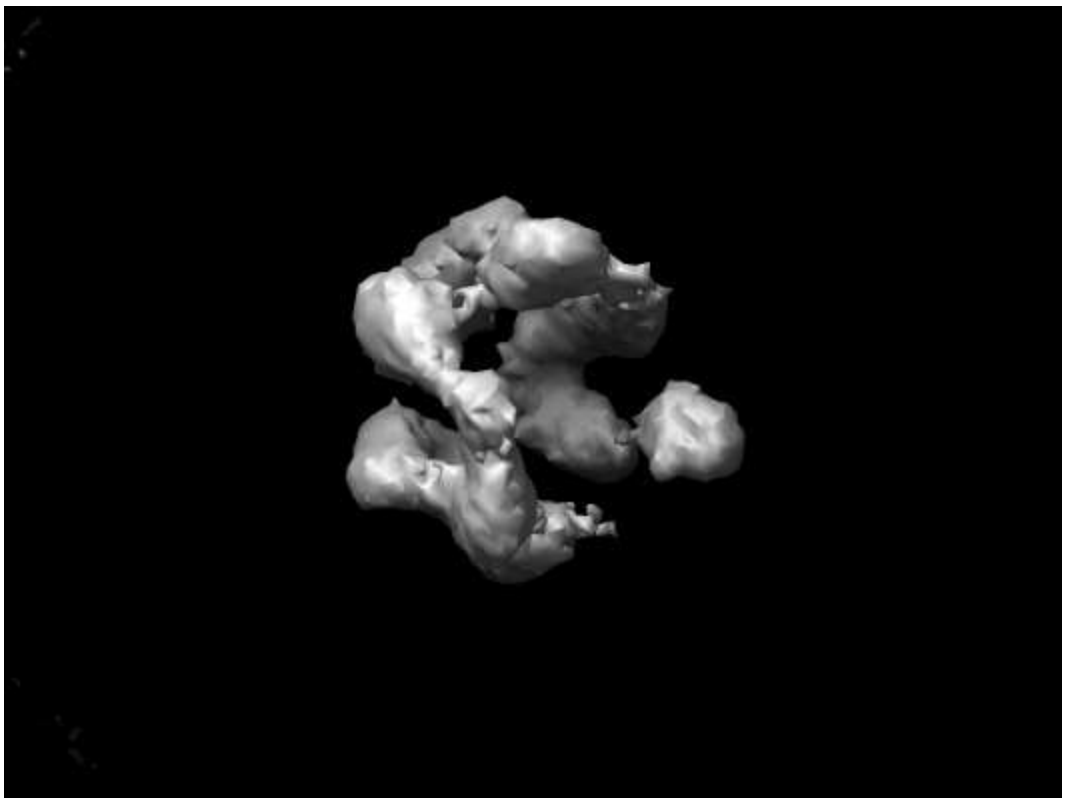

FIGURA 6.8: Complejo DnaB-DnaC reconstruido usando proyección dirigida por vóxeles y *blobs* como función base

distribución de ruido que tienen las proyecciones que se utilizan de entrada (dado que obtenemos el mismo resultado con ambas variantes de proyección en nuestro programa, incluso en la versión de CPU).

6.2. *Speedup*

Una vez demostrado que las reconstrucciones obtenidas con nuestro algoritmo tienen una calidad aceptable, continuaremos exponiendo los resultados relativos al tiempo de ejecución.

Algoritmo / Volumen	xmipp_art	CPU vóxeles	CPU *blobs*
Antígeno T	0.003764	0.003098	0.003222
DnaB-DnaC	0.001168	0.001478	0.001575
TwoRings	0.026991	0.010998	0.018833
cuboloid	0.007541	0.003696	0.002935
sphereblob	0.004791	0.000788	0.000778

TABLA 6.1: Error cuadrático medio para distintas reconstrucciones y algoritmos en CPU

Algoritmo / Volumen	GPU vóxeles	GPU *blobs*
Antígeno T	0.003098	0.003216
DnaB-DnaC	0.002817	0.001582
TwoRings	0.018624	0.018824
cuboloid	0.003647	0.002928
sphereblob	0.000788	0.000780

TABLA 6.2: Error cuadrático medio para distintas reconstrucciones y algoritmos en GPU

6.2.1. Afinación de los algoritmos

Antes de comparar los tiempos de las distintas implementaciones, es necesario afinar cada una para que el tiempo que consuma sea el mínimo. Para los algoritmos que usan la GPU, el paso final de esta afinación se resume en encontrar aquellos valores de tamaños de bloque que hacen que el aprovechamiento de los recursos sea máximo, y, previsiblemente, el tiempo necesitado mínimo. Para los algoritmos que utilizan la CPU la afinación ha sido básicamente permitir al compilador que realice el máximo de optimizaciones que pueda, junto con el ensayo de diversas construcciones que, sin variar el algoritmo, aceleran el programa.

Para encontrar aquellos valores de tamaño de bloque que optimizan el aprovechamiento de los recursos de la GPU, la primera opción es utilizar el *CUDA Occupancy Calculator* mencionado anteriormente. Esta hoja de cálculo muestra qué tamaños de bloque hacen que la ocupación de los

multiprocesadores sea máxima. Como ya se comentó antes, este hecho no implica que sea el valor que consiga minimizar el tiempo, pero es un buen comienzo. Para cubrir todas las posibilidades, se ha hecho un *profiling* de todas las combinaciones posibles de tamaños de bloque para los distintos volúmenes de prueba. Luego se han mostrado esas tablas como gráficas en 3 dimensiones y encontrado los valores que minimizan los tiempos. Como ejemplo, se muestran las gráficas obtenidas para tres volúmenes de distinto tamaño y utilizando las dos formas distintas de calcular la proyección.

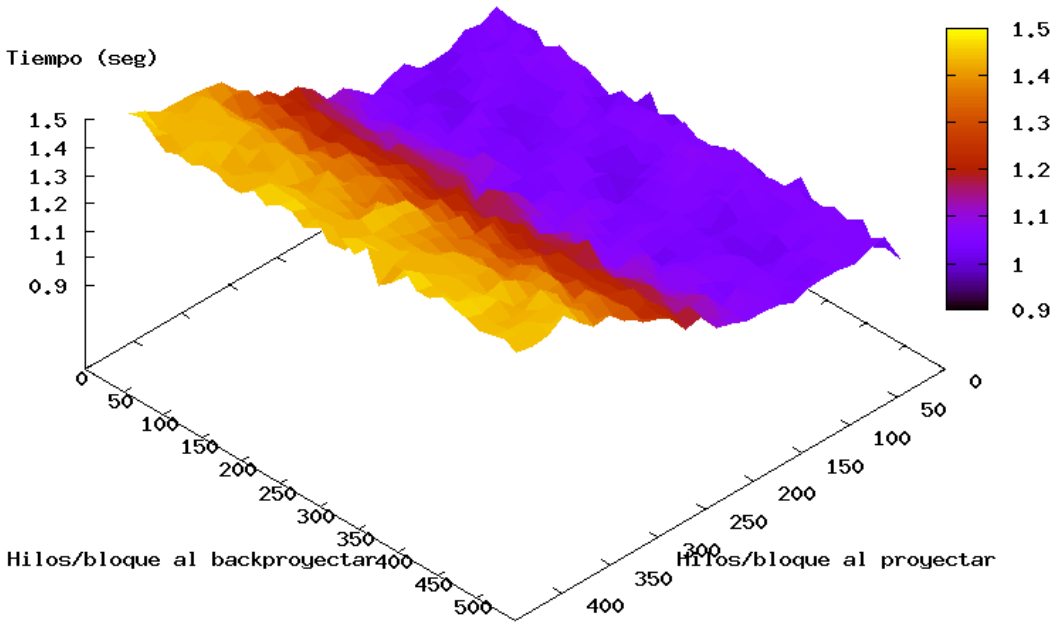

FIGURA 6.9: Gráfica de tiempos de la reconstrucción del Antígeno T proyectando por rayos en GPU

Las gráficas correspondientes al volumen del Antígeno T (250 proyecciones de 64×64) se observan en las figuras 6.9 y 6.10; las correspondientes al complejo DnaB-DnaC (6995 proyecciones de 81×81) en las figuras 6.11 y

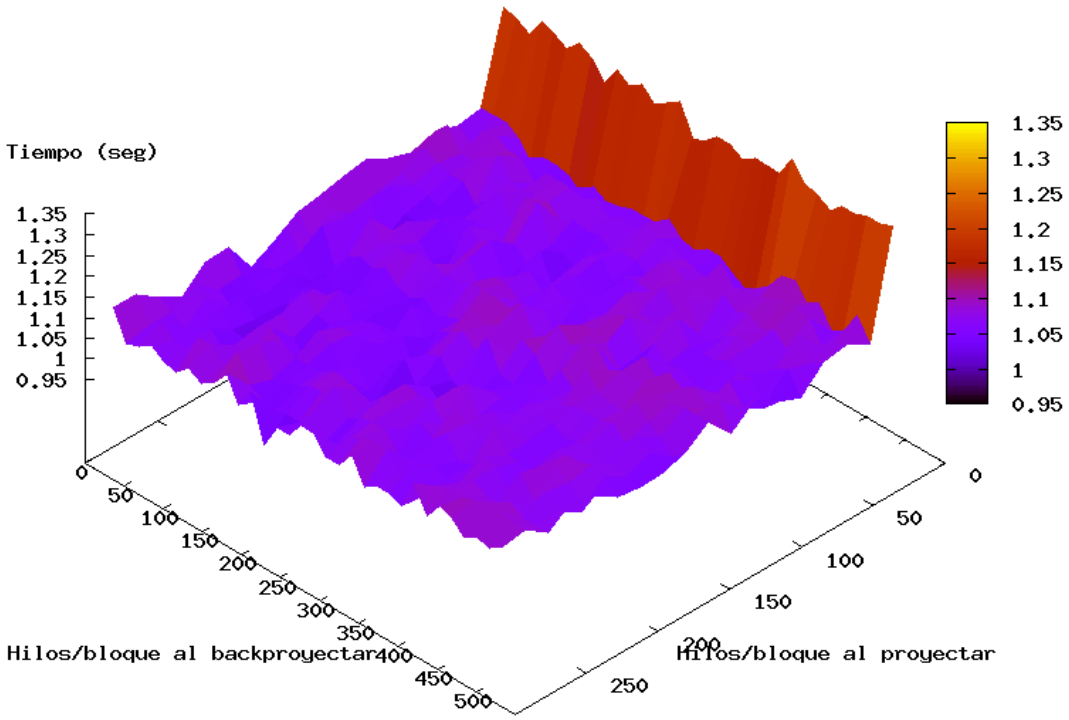

FIGURA 6.10: Gráfica de tiempos de la reconstrucción del Antígeno T proyectando por vóxeles en GPU

6.12; y las correspondientes al *phantom Sphereblob* (480 proyecciones de 192×192) en las figuras 6.13 y 6.14.

Como puede comprobarse en las gráficas, ninguna de ellas encaja con lo que el *CUDA Occupancy calculator* "predecía". Esto es así porque el algoritmo del SART no es lo suficientemente intensivo en operaciones aritméticas como para aprovechar al máximo los multiprocesadores, y las latencias de memoria influyen en gran medida. Además el uso de los datos provenientes de memoria siguen un patrón que raramente reutiliza datos; sólo se leen, se computa un nuevo resultado y se vuelven a escribir. Esto hace que estas latencias sean "virtualmente" imposibles de evitar u ocultar, más allá de un cierto punto.

Aunque en general no se puede encontrar ninguna regla para hallar los

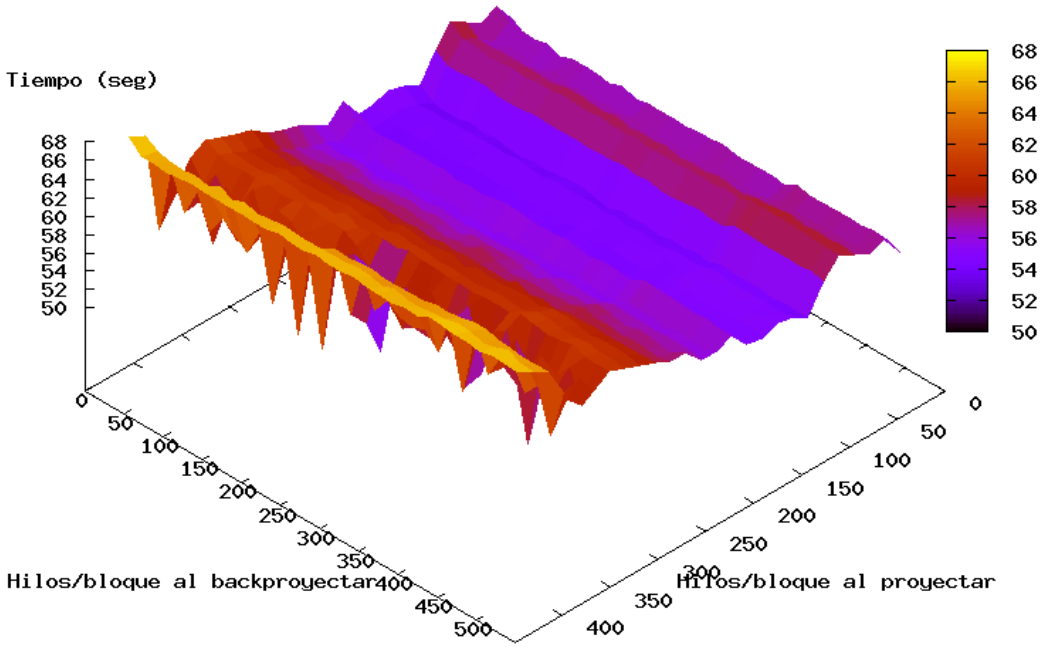

FIGURA 6.11: Gráfica de tiempos de la reconstrucción del complejo *DnaB-DnaC* proyectando por rayos en GPU

valores óptimos, al analizar el *profiling* se han podido determinar ciertos valores que funcionan "razonablemente" bien para los volúmenes tratados dependiendo de sus tamaños. En particular, se ha podido comprobar que la variación del número de hilos por bloque al backproyectar tiene una influencia casi inapreciable en el tiempo total, aunque a veces valores muy pequeños perjudican ligeramente el rendimiento.

6.2.2. Comparación de tiempos

Para calcular el *speedup* obtenido se hemos elegido comparar los tiempos de ejecución de nuestro algoritmo corriendo CPU y GPU. Aunque en un

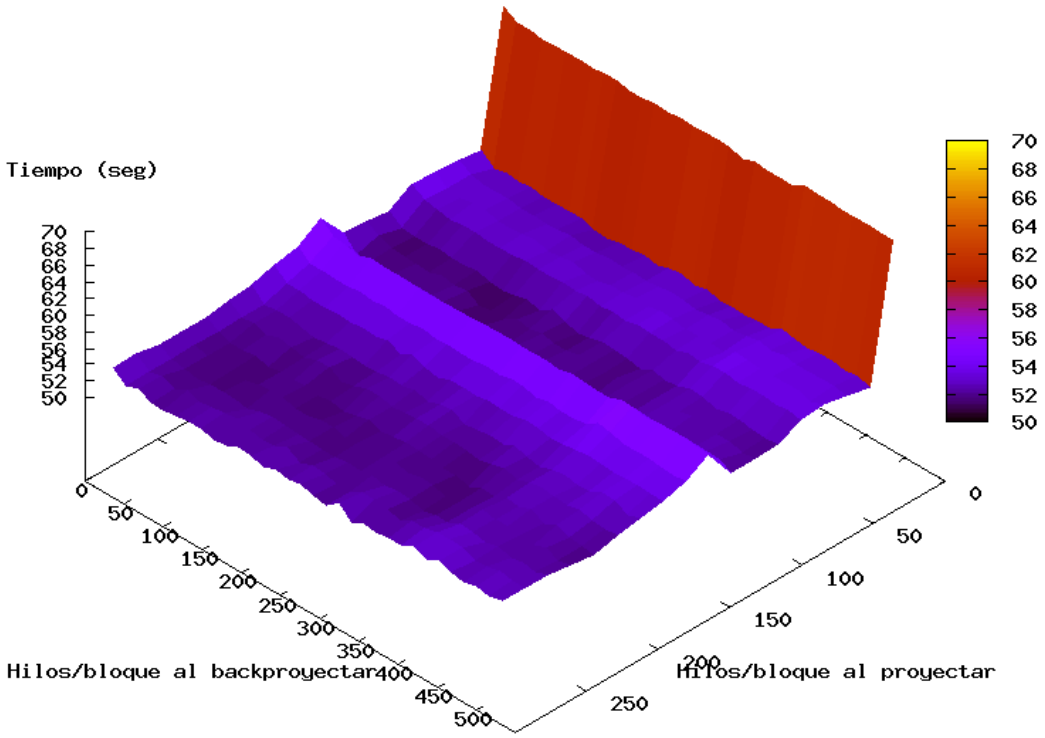

FIGURA 6.12: Gráfica de tiempos de la reconstrucción del complejo *DnaB-DnaC* proyectando por vóxeles en GPU

principio se pensó en comparar los tiempos de ejecución del algoritmo corriendo en GPU y del programa reconstructor de Xmipp, decidimos elegir también nuestro algoritmo de CPU ya que los resultados serían más representativos del factor de ganancia obtenido por el uso de la GPU. El reconstructor de Xmipp actúa de forma diferente al nuestro (en particular siempre realiza la ordenación de proyecciones y además utiliza una distribución espacial algo distinta); mientras que nuestras versiones de CPU y GPU realizan casi exactamente las mismas operaciones. De todas formas, los resultados de *speedup* no se deben tomar en un sentido absoluto, puesto que estamos comparando tiempos de un programa ejecutado enteramente en la CPU con uno en el que la mayor parte se realiza en una GPU

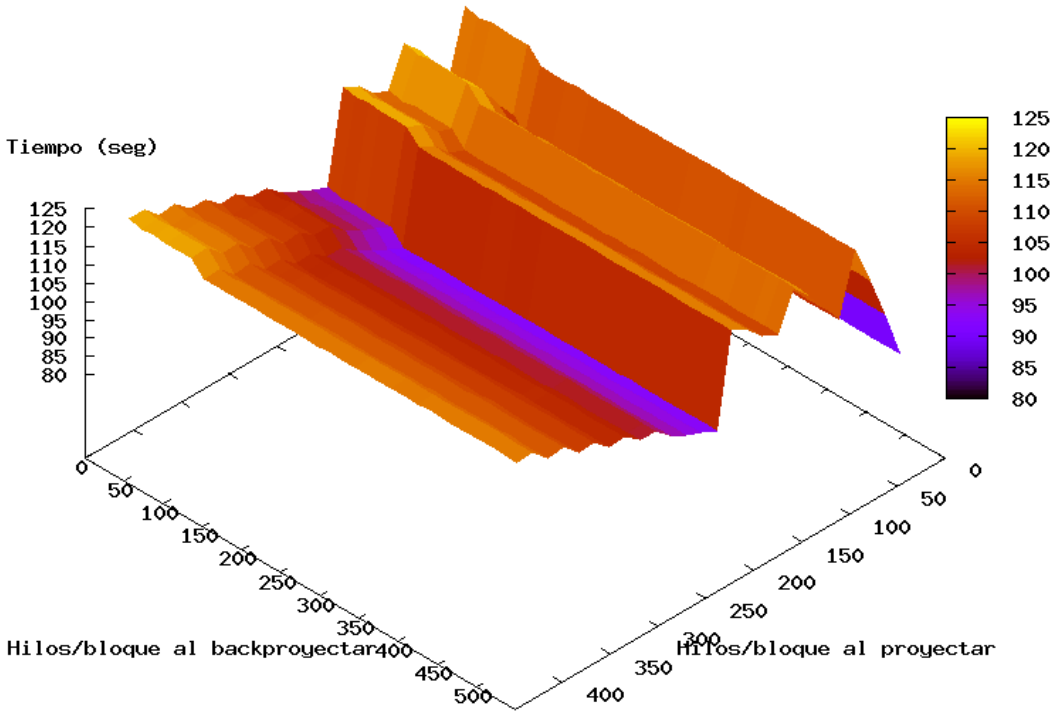

FIGURA 6.13: Gráfica de tiempos de la reconstrucción del *phantom Sphereblob* proyectando por rayos en GPU

independiente; un cambio de cualquiera de éstas nos daría resultados numéricos distintos, aunque presumiblemente las diferencias se reducirían a un factor constante que no afectará a su orden de magnitud. Para realizar una comparación más justa, utilizamos en ambos casos equipos bastante potentes.

Para medir los tiempos de ejecución del algoritmo en CPU, se ha compilado con *g++* utilizando la opción de máxima optimización, *-O3*. La medición de tiempos del algoritmo en GPU fue el propio *profiling*, simplemente seleccionando el tiempo mínimo de todos lo obtenidos para los distintos tamaños de bloque. Dado que los accesos a disco para leer las proyecciones son los mismos en los casos a comparar, decidimos que la medición debía

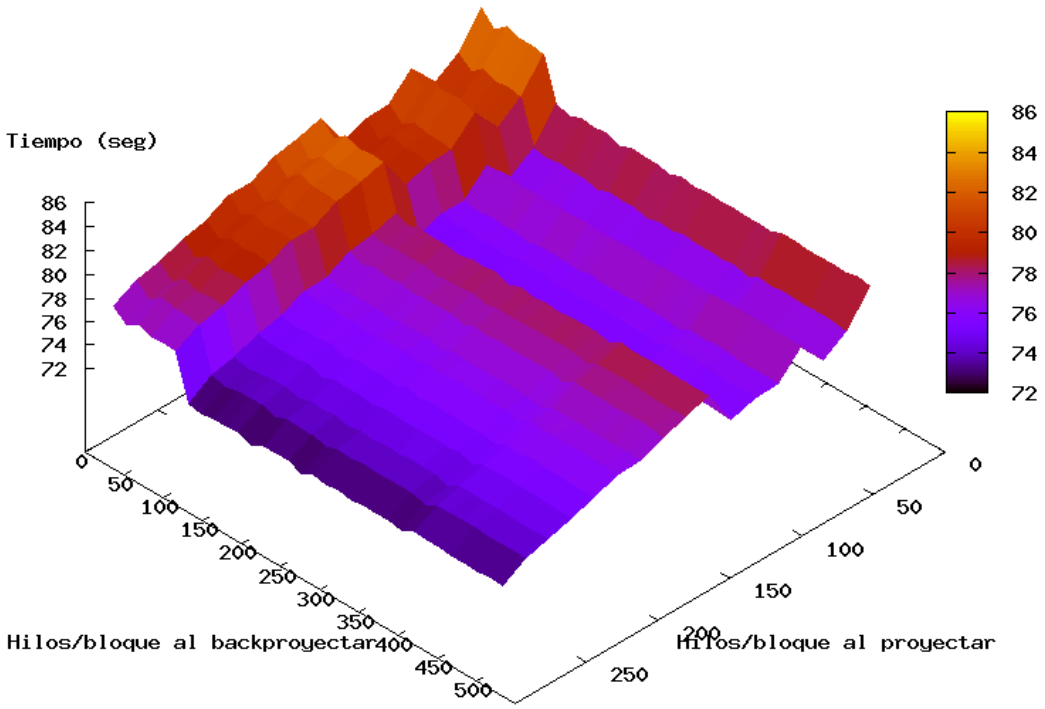

FIGURA 6.14: Gráfica de tiempos de la reconstrucción del *phantom Sphereblob* proyectando por vóxeles en GPU

tener en cuenta solamente el tiempo de usuario, ya que es el tiempo durante el cual se está realizando realmente la reconstrucción. Para evitar interferencias con otros procesos que pudiesen distorsionar las mediciones, éstas se realizaron en un sistema de colas, ya que permite a los procesos tener la propiedad de los recursos durante todo el tiempo que duraba la ejecución. Para la compilación y ejecución de los programas se ha utilizado un sistema del departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid, con las siguientes características:

- Procesador Intel Core2 Duo, E6600 a 2.4 GHz (de doble núcleo)
- 2 Gb de Memoria RAM

- GPU Nvidia 8800 GTX
- g++ (gcc) 4.1.2 20061115
- NVIDIA CUDA Toolkit y NVCC versión 0.8

Algoritmo / Volumen	xmipp_art	CPU ray-driven	CPU voxel-driven
Antígeno T	45.28	23.04	20.19
DnaB-DnaC	2463.19	1334.61	1116.00
TwoRings	113.69	75.16	51.55
cuboloid	820.22	488.19	376.31
sphereblob	2192.79	1680.01	1011.94

TABLA 6.3: Tiempo de usuario (en segundos) para distintas reconstrucciones y algoritmos en CPU

Algoritmo / Volumen	GPU ray-driven	GPU voxel-driven
Antígeno T	0.94	1.00
DnaB-DnaC	50.31	51.11
TwoRings	2.28	2.46
cuboloid	26.36	20.51
sphereblob	83.41	72.25

TABLA 6.4: Tiempo de usuario (en segundos) para distintas reconstrucciones y algoritmos en GPU

Los tiempos obtenidos para los distintas reconstrucciones se pueden observar en las tablas 6.3 y 6.4 para los algoritmos ejecutándose en CPU y GPU respectivamente. Los *speedups* relativos a cada volumen y método de proyección se pueden encontrar en la tabla 6.5. Como dato informativo, se pueden encontrar los *speedups* de los algoritmos en GPU frente al reconstructor de Xmipp en la tabla 6.6. Por último, los detalles de los volúmenes mencionados en las tablas anteriores (así como a lo largo de la memoria) se pueden encontrar en 6.7.

Algoritmo / Volumen	GPU ray-driven	GPU voxel-driven
Antígeno T	24.51	20.19
DnaB-DnaC	26.53	21.84
TwoRings	32.96	20.96
cuboloid	18.52	18.35
sphereblob	20.14	14.01

TABLA 6.5: *Speedup* para distintas reconstrucciones

Algoritmo / Volumen	GPU ray-driven	GPU voxel-driven
Antígeno T	48.17	45.28
DnaB-DnaC	48.96	48.19
TwoRings	49.87	46.22
cuboloid	31.12	39.99
sphereblob	26.29	30.35

TABLA 6.6: *Speedup* frente a Xmipp para distintas reconstrucciones

	Tamaño de proyección	Número de proyecciones
Antígeno T	64×64	250
TwoRings	65×65	600
DnaB-DnaC	81×81	6995
cuboloid	128×128	600
sphereblob	192×192	1000

TABLA 6.7: Detalles de los volúmenes utilizados para la reconstrucciones

Como se puede observar, el *speedup* obtenido está entre 14,01 y 26,53, estando por lo general en torno a 20.

6.3. Parámetros del algoritmo que afectan a la calidad

Existen varios parámetros ajustables del algoritmo que afectan a la calidad de las reconstrucciones obtenidas. Aunque el más importante es el factor de relajación λ, la ordenación de proyecciones y el número de iteraciones efectuadas recorriendo todas las proyecciones también afecta a la calidad de la reconstrucción final. En esta sección se analizará el impacto de cada una de ellos basándonos en nuestra medida de la calidad, el error cuadrático medio.

6.3.1. Factor de relajación λ

Como ya se ha comentado antes, el factor de relajación λ es un valor que amortigua las correcciones que se aplican al volumen, y cuyo valor debe estar comprendido en $(0, 2)$ para garantizar la convergencia; nosotros sólo consideraremos valores en $(0, 1)$. Para ilustrar su impacto, hemos calculado el error cuadrático medio cometido para distintas reconstrucciones variando de factor de relajación, y lo hemos mostrado en gráficas.

Las figuras 6.15 y 6.16 muestran cómo varía la calidad de las reconstrucciones del *phantom cuboloid* cambiando los valores de λ, utilizando como funciones base *blobs* y vóxeles respectivamente. La figura 6.17 muestra la variación de calidad que sufre la reconstrucción del complejo *DnaB-DnaC* utilizando *blobs*, y la figura 6.18 la variación sufrida por la reconstrucción del antígeno T utilizando *blobs*.

Como se observa, no existen valores de λ que funcionen bien con todos las reconstrucciones. Por ejemplo para las reconstrucciones a partir de proyecciones con mucho ruido, como es el caso del complejo *DnaB-DnaC*, los mejores valores son los más bajos. En cambio, para reconstrucciones a partir de proyecciones con muy poco ruido (como el antígeno T) son mejores valores altos. Esto es debido a que cuando las proyecciones tienen poco

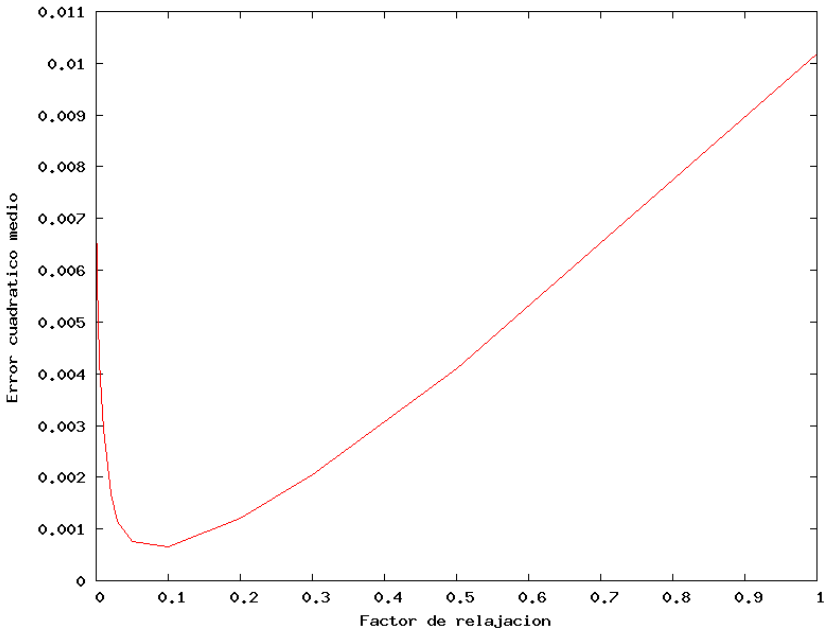

FIGURA 6.15: Calidad del *phantom Cuboloid* utilizando *blobs* y variando el factor de relajación λ

ruido, cada corrección que produzcan acercará al volumen actual hacia el volumen final, con lo cual será recomendable no amortiguarla. En el caso de proyecciones con alto nivel de ruido, dichas correcciones no tendrán tanta información útil, y entonces será recomendable amortiguarlas lo suficiente para evitar que los coeficientes oscilen mucho innecesariamente.

Las figuras 6.15 y 6.16 también reflejan un resultado interesante. Aunque ambas tratan sobre la reconstrucción del mismo volumen utilizando las dos funciones base distintas, se aprecia una similitud entre ellas. A pesar de que la variación al cambiar el valor de λ es diferente, se observa que los valores que mejor resultado dan en ambos casos están muy cercanos. Este comportamiento se aprecia en todos los demás volúmenes probados, con lo que se llega a la conclusión que los valores óptimos de λ están más relacionados con el volumen en sí que con la función base elegida (al menos de entre las dos utilizadas).

En general, para la mayoría de las reconstrucciones, un valor de λ menor que

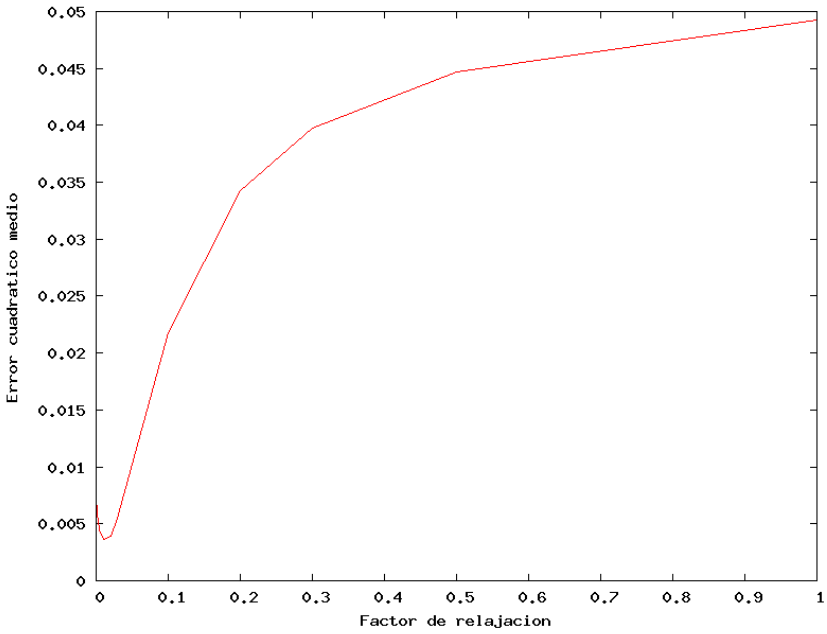

FIGURA 6.16: Calidad del *phantom Cuboloid* utilizando vóxeles y variando el factor de relajación λ

0.1 será lo recomendable.

6.3.2. Número de iteraciones

Otro parámetro que se puede ajustar para intentar conseguir una mayor calidad de la reconstrucción es el número de iteraciones que se aplicarán a todas las proyecciones. El comportamiento por defecto es recorrer todas las proyecciones, y una vez ha terminado con ellas devolver el volumen resultante. Pero nada nos impide volver a repetir el proceso de nuevo empezando con la estimación devuelta, con la esperanza de conseguir que el volumen reconstruido se aproxime mejor al original.

Los resultados de medir la calidad de las reconstrucciones para distinto número de iteraciones y volúmenes se pueden observar en las figuras 6.19, 6.20, 6.21 y 6.22. Las dos primeras son las obtenidas de la reconstrucción del *phantom TwoRings* utilizando las dos funciones base. Las dos restantes son las obtenidas de la reconstrucción *phantom Sphereblob* con las dos funciones

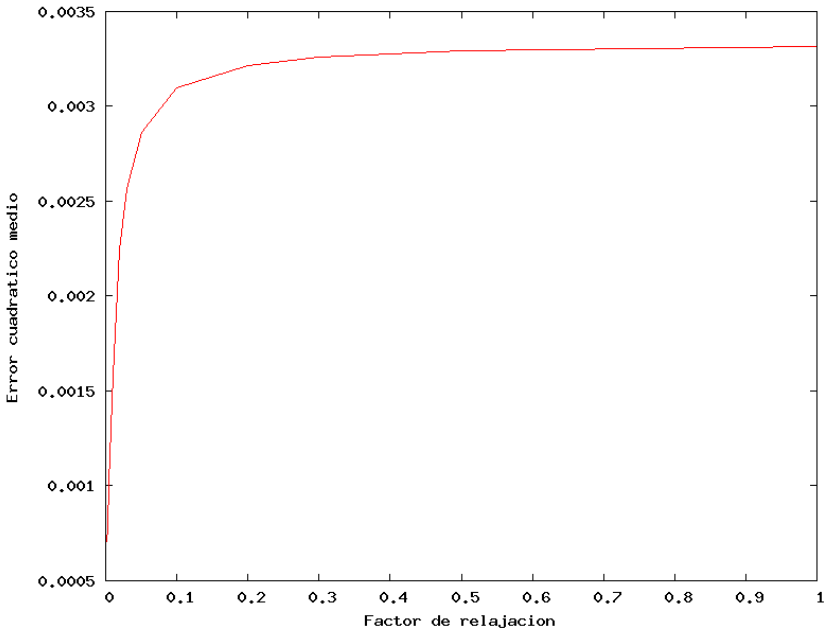

FIGURA 6.17: Calidad del complejo *DnaB-DnaC* utilizando *blobs* y variando el factor de relajación λ

base.

Como se puede observar, no siempre es mejor iterar un número mayor de veces, contrariamente a lo que intuitivamente pueda parecer. Como se explica en [Sor02], iterar demasiadas veces hace que las soluciones que van consiguiéndose incorporen cada vez mayor parte del ruido de las proyecciones (ya que tienden a reducir el error cuadrático medio entre las proyecciones medidas y las calculadas). Es por ello que en la reconstrucción del *phantom TwoRings* iterar más veces da lugar a reconstrucciones peores. En otros casos como el *phantom Sphereblob* iterar más veces mejora la calidad de la reconstrucción (al menos hasta la décima iteración). Teniendo en cuenta el conjunto de reconstrucciones que hemos comprobado, 2 de ellas empeoraban al incrementar el número de iteraciones mientras que 3 mejoraban con dicho aumento. Esto deja entrever que iterar no lleva a mejores resultados en la mayoría de los casos.

Otro resultado interesante que se comprueba observando las gráficas es que

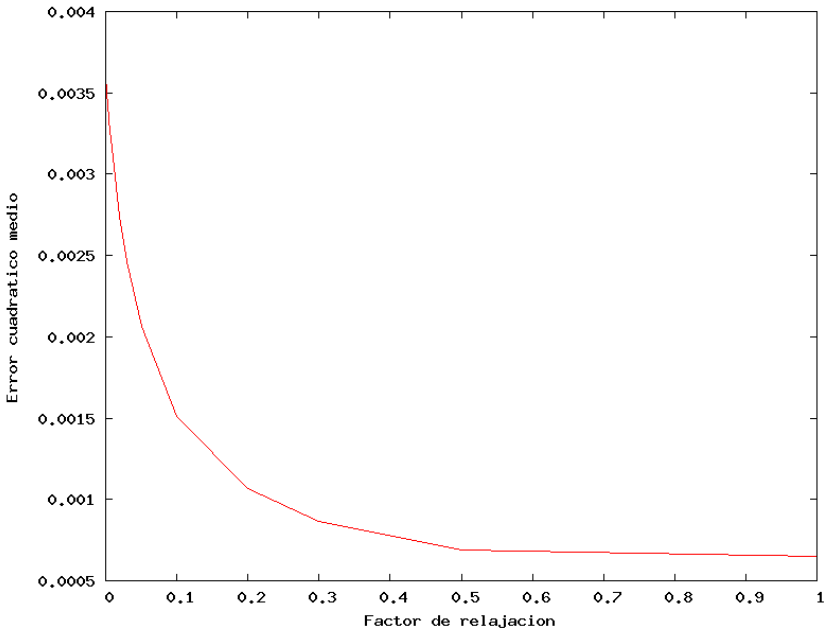

FIGURA 6.18: Calidad del antígeno T utilizando vóxeles y variando el factor de relajación λ

una reconstrucción tiene un comportamiento muy similar respecto a su calidad final al iterar, independientemente de la función base usada. Esto es coherente con lo anteriormente explicado de que el aumento de las iteración puede incorporar ruido a la reconstrucción, y esto no depende de la función base utilizada. Eso es lo que se aprecia en las figuras 6.19, 6.20, 6.21 y 6.22, aunque para las producidas por el *phantom Sphereblob* se observe un pequeño aumento del error (oscilación de la solución) a partir de la cuarta iteración.

6.3.3. Ordenación de las proyecciones

El último aspecto que queda por analizar es el impacto que tiene la ordenación previa de las proyecciones sobre la calidad final. A priori, dicha ordenación debería mejorar la convergencia del algoritmo ya que en cada paso se elegiría una proyección con un ángulo lo más perpendicular posible con las anteriores. Al hacer esto, la información aportada por la actual sería bastante diferente de las anteriores, produciendo una corrección mayor y

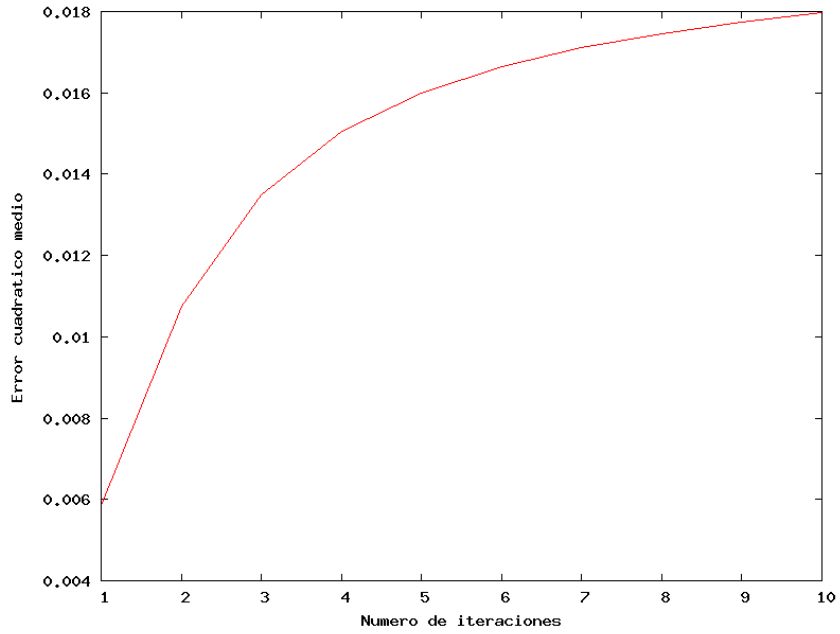

FIGURA 6.19: Calidad del *phantom TwoRings* utilizando *blobs* y variando el número de iteraciones

acercando al volumen actual hacia la solución final.

Para medir este impacto hemos repetido las mediciones del apartado anterior, pero efectuando una ordenación de las proyecciones (opción `--placebo 2` del programa, que ordena las proyecciones tratando de maximizar el ángulo formado entre la actual y las dos inmediatamente anteriores). Las gráficas resultantes se han comparado con las obtenidas sin ninguna ordenación. Un ejemplo de ellas se puede observar en las figuras de esta sección. La figura 6.23 es la producida por el *phantom Sphereblob* ordenando proyecciones, y la producida sin ordenación de puede observar en la figura 6.22 de la sección anterior. La figura 6.24 muestra la gráfica producida ordenando proyecciones en el *phantom TwoRings*, y la figura 6.19 la obtenida sin dicha ordenación. Para la reconstrucción del antígeno T, la diferencia entre ordenar y no ordenar proyecciones se puede apreciar en las figuras 6.26 y 6.25.

Como queda patente en las gráficas mostradas, la ordenación por

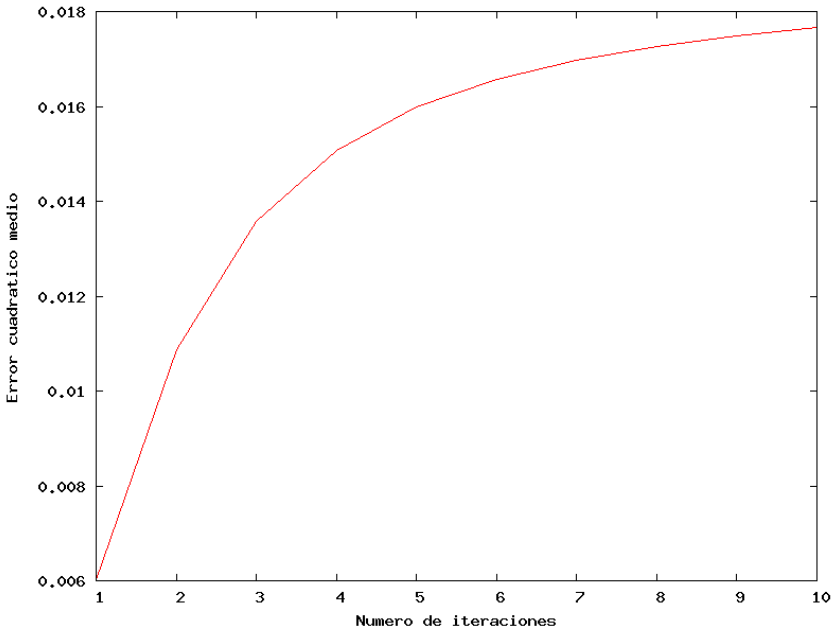

FIGURA 6.20: Calidad del *phantom TwoRings* utilizando vóxeles y variando el número de iteraciones

perpendicularidad de las proyecciones no afecta en absoluto a la calidad de la reconstrucción. Tanto en los ejemplos mostrados como en las demás reconstrucciones probadas, la diferencia entre ordenar y no ordenar es imperceptible, contrariamente a lo que pudiera parecer en un principio.

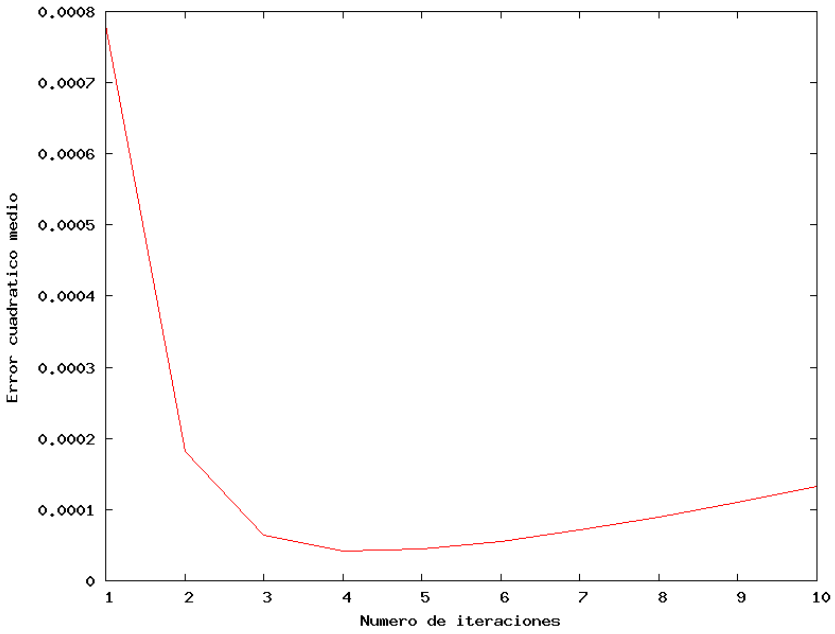

FIGURA 6.21: Calidad del *phantom Sphereblob* utilizando *blobs* y variando el número de iteraciones

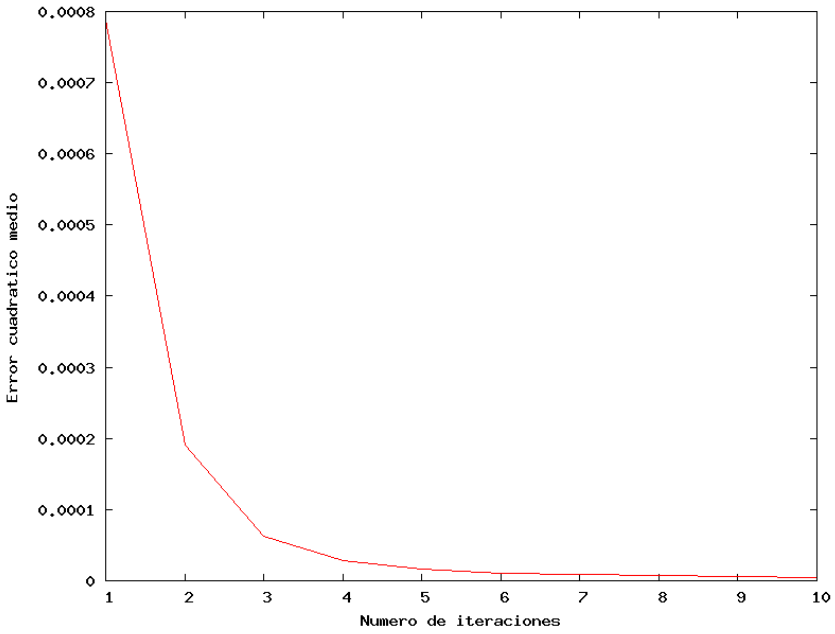

FIGURA 6.22: Calidad del *phantom Sphereblob* utilizando vóxeles y variando el número de iteraciones

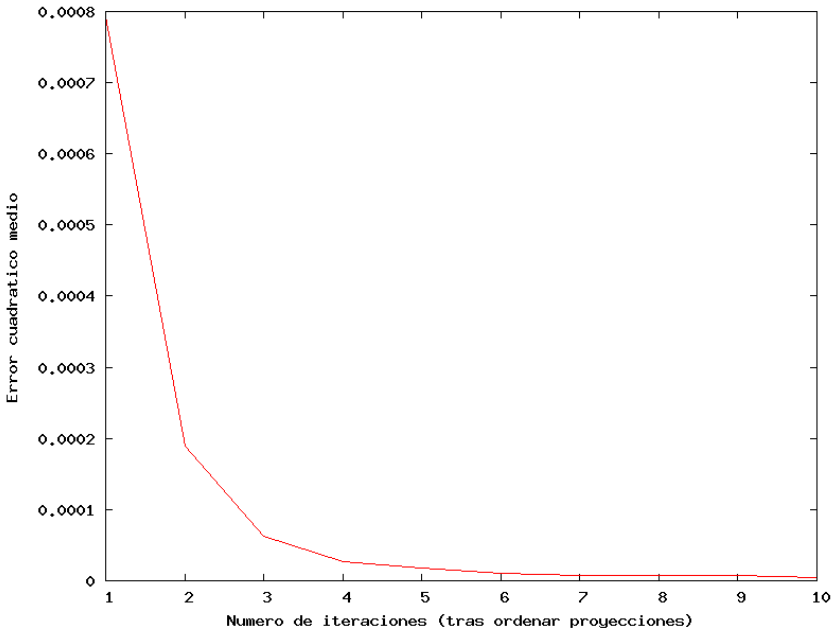

FIGURA 6.23: Calidad del *phantom Sphereblob* utilizando vóxeles, ordenando previamente las proyecciones y variando el número de iteraciones

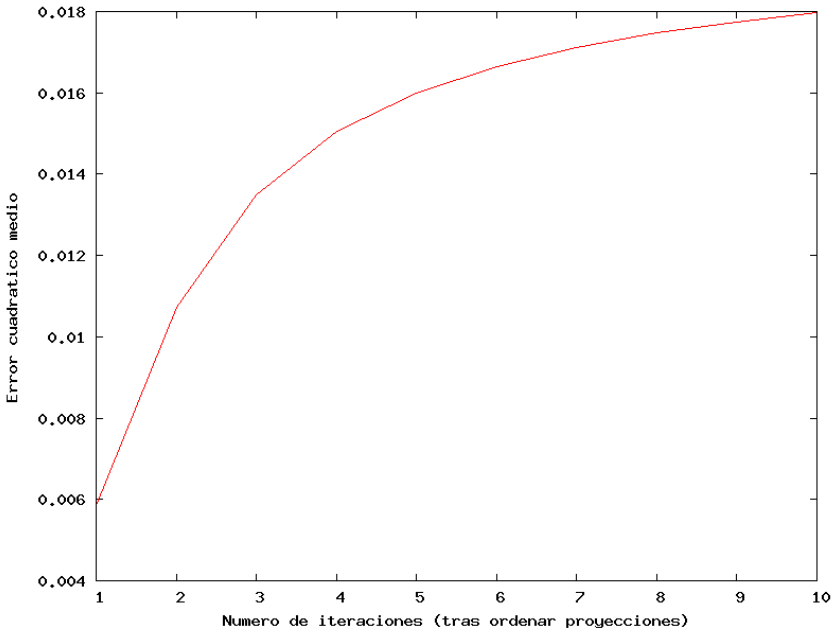

FIGURA 6.24: Calidad del *phantom TwoRings* utilizando *blobs*, ordenando previamente las proyecciones y variando el número de iteraciones

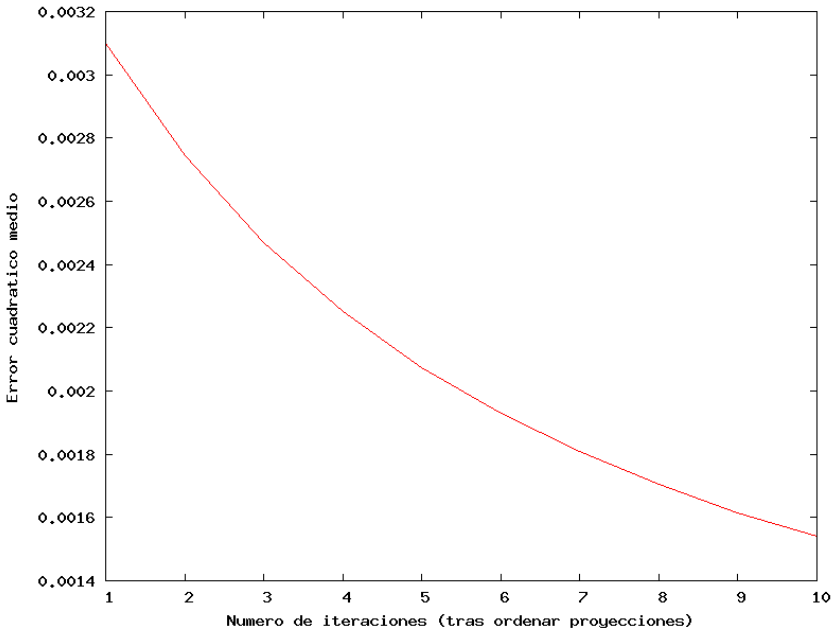

FIGURA 6.25: Calidad del antígeno T utilizando vóxeles, sin ordenar las proyecciones y variando el número de iteraciones

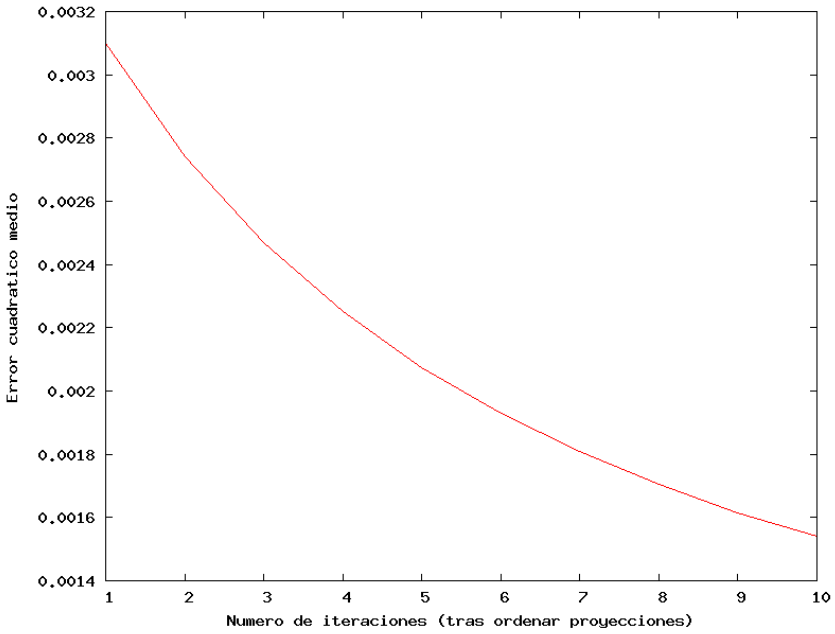

FIGURA 6.26: Calidad del antígeno T utilizando vóxeles, ordenando previamente las proyecciones y variando el número de iteraciones

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

Tanto NVIDIA como ATI están apostando por facilitar la programación de sus tarjetas gráficas, intentando alcanzar nuevas áreas de mercado donde antes no estaban presentes. Para ello están adoptando una arquitectura unificada de sus chips y ofrecen un modelo de ejecución mas sencillo, el *Stream Proccesing*. Este nuevo enfoque facilita en gran manera la utilización de las tarjetas gráficas como coprocesadores, ya que no es necesario conocer todos los detalles arquitectónicos de bajo nivel de los chips ni programar manualmente los diferentes *shaders*. A pesar de la gran utilidad de este enfoque, su desarrollo aún no ha llegado a una fase estable para su utilización. En particular, el sistema que se ha utilizado en este proyecto (CUDA, desarrollado por la empresa NVIDIA) es el primer sistema lanzado de este tipo y aún permanece en fase beta. Durante la realización del proyecto han cambiado los manuales de referencia así como algunas de las características del sistema en sí (latencias de memoria, número de ciclos de las instrucciones, tipos de accesos a memoria, modelo de ejecución...). En los drivers se han encontrado bugs que poco a poco se van solucionando. A pesar de esto, NVIDIA sigue haciendo un gran esfuerzo para conseguir un sistema más potente, y además de arreglar los errores sigue añadiendo nuevas posibilidades (como una intercomunicación de hilos más flexible

mediante el uso de semáforos generales o la adición de soporte para números en coma flotante de doble precisión). Es por esto que, si el mercado responde, este tipo de sistemas está llamado a ser una de las opciones más interesantes a la hora de cubrir una amplia gama de necesidades de computación a un precio muy competitivo.

El algoritmo del SART que hemos desarrollado y portado a la GPU, a pesar de obtener una ganancia espectacular de velocidad, no aprovecha al máximo los recursos que el dispositivo ofrece; de hecho alcanza una ocupación global de los multiprocesadores en torno al 75 %. Pero es complicado alcanzar un rendimiento mayor debido principalmente a las características del algortmo en sí. A pesar de tener una alta carga aritmética, también requiere un alto número de accesos a memoria. En su gran mayoría, los datos que se leen de memoria tienen una corta vida útil, y amortizar la transferencia mediante la reutilización no es posible. Aparte de esto, aparecen problemas de concurrencia al acceder a memoria distintos hilos que solo se pueden resolver (con las posibilidades actuales de CUDA) separando en el tiempo los hilos candidatos a producir dichos conflictos. Con las nuevas versiones de CUDA y las nuevas posibilidades que anuncia NVIDIA estos problemas puede que dejen de tener una importancia tan crucial. Entonces quizá se puedan desarrollar otras aproximaciones que permitan utilizar al máximo los potentes recursos que nos proporciona la GPU.

Las funciones base son un elemento fundamental en el proceso de la reconstrucción. En nuestro proyecto hemos utilizado dos distintas, vóxeles y *blobs*, pero existen muchas más. La forma en que se calcula la integral de rayo a través de esas funciones base hace que unas u otras sean más propicias para la reconstrucción dependiendo del método de proyección que se haya usado. Para funciones base con simetría esférica como los *blobs*, donde la integral puede calcularse solamente a partir de la distancia del rayo al centro del vóxel, la proyección dirigida por vóxeles es la más adecuada. Si el cálculo de estas integrales de rayo a partir de la distancia al centro no es demasiado costoso, se puede precalcular para distintos valores de la distancia (conocidos como *footprint*) y aproximar o interpolar los resultados cada vez que se requiera este cálculo. Para funciones como los vóxeles, cuyo cálculo

depende de la distancia recorrida dentro del vóxel, la proyección dirigida por rayos es la más idónea ya que trabaja internamente con esta información. Los *blobs* forman una serie de funciones base con interesantes características, pero para que los resultado obtenidos sean apreciablemente mejores que con el uso de vóxeles se debe adoptar una distribución espacial más compleja que la utilizada en este proyecto. Situar un solo *blob* en el centro de cada vóxel hace que sea imposible reconstruir volúmenes donde existan regiones de densidad constante. Para estos casos, una mejor elección es una distribución espacial cúbica centrada en el cuerpo (BCC) como se explica en [Sor02] y como utiliza la suite Xmipp.

De los tres parámetros variables del algoritmo cuyo impacto se ha analizado, dos no han surtido el efecto deseado sobre la calidad final de las reconstrucciones. La iteración a través de las proyecciones, a pesar de las razones esgrimidas en contra para los algoritmos algebraicos de reconstrucción, merecía ser probada. Los resultados han sido negativos, pues a veces mejora la calidad y otras la empeora (si bien no demasiado), aumentando siempre el tiempo requerido para la reconstrucción. La ordenación de las proyecciones siguiendo criterios geométricos de perpendicularidad ha arrojado unos resultados sorprendentes: afectaba mínimamente a la calidad. Aunque en un principio elegir primero las proyecciones que más información nueva aportaban debería hacer que la solución se pareciese más al original, se ha comprobado que para el algoritmo del SART esto no tiene ninguna importancia cuando se dispone de un número suficientemente elevado de proyecciones.

El *speedup* obtenido frente al programa reconstructor de Xmipp (entre 30 y 50 para las pruebas realizadas) hace que sea muy interesante su utilización para la investigación biológica. En las reuniones mantenidas, Carlos Óscar Sánchez Sorzano (miembro del CNB y autor de los programas de reconstrucción de Xmipp) ha expresado su interés en que se continúe este proyecto ampliándolo para que acabe formando parte de la suite de aplicaciones Xmipp. Esto acelerará sustancialmente el proceso de obtener modelos en tres dimensiones de partículas en estudio, ya que la reconstrucción forma parte de dos de las etapas más largas de este proceso:

la estimación de ángulos de las proyecciones y la reconstrucción a partir de proyecciones en sí misma.

7.2. Trabajo futuro

El trabajo que se puede realizar tomando como base este proyecto es muy interesante y de gran utilidad para el campo de la microscopía electrónica de transmisión.

En primer lugar, una posibilidad es añadir *splines* espaciales como función base de los volúmenes. Estas funciones no gozan actualmente de amplio uso en el campo de la microscopía electrónica (debido principalmente a que su uso ralentiza todavía más el proceso de reconstrucción), pero se están empezando a estudiar y sus características son muy interesantes. Con ellas se pueden modelizar volúmenes con bordes suaves, forman una partición de la unidad y además pueden aproximar con exactitud una amplia gama de funciones.

Como se ha comentado en las conclusiones, la estimación de ángulos es una etapa muy lenta dentro del proceso completo de hallar la representación tridimensional de macromolécula a partir de proyecciones en microscopía electrónica. Dicha etapa es un proceso iterativo en el que a partir de algunas pistas sobre la forma de la partícula, se genera una estimación de ángulos y se va refinando progresivamente hasta encontrar una estimación que genere una reconstrucción coherente con las pistas iniciales. Portar partes de este proceso a la GPU para que se ejecutasen en paralelo conseguiría reducir drásticamente el tiempo total necesitado, haciendo que en vez de necesitar varios meses (como es el caso actualmente), se pueda conseguir una reconstrucción del objeto que se estudia en unos pocos días.

Por último, la integración de este proyecto con la suite Xmipp sería un paso muy importante para la difusión del mismo. Así se conseguiría que la comunidad de microscopía electrónica tuviese acceso a las mejoras en tiempo alcanzadas, a la vez que se garantizaría la supervivencia y evolución de este proyecto como parte de uno de los paquetes de software más importantes en este campo. Para proceder a ello lo primero que sería necesario sería

homogeneizar el algoritmo del SART utilizado, ya que en Xmipp usa un método directamente derivado de ART, que ellos denominan Block-ART, y que es diferente al utilizado por [KS88], [MY99] y la mayoría de autores. Una vez hecho esto, la integración sería muy sencilla ya que se ha hecho un esfuerzo en utilizar las misma interfaz que proporciona Xmipp para la entrada y salida de datos.

Apéndice A

Instalación y uso

Este apéndice trata acerca de la instalación y uso de la aplicación. Se ha intentado que dichas tareas sean lo más intuitivas posible para que el usuario no tenga que referirse a este apéndice. De todas formas lo se incluye para ayudar a quien encuentre dificultades.

A.1. Instalación

La instalación del programa es una tarea muy sencilla: sólo hace falta disponer de las fuentes y compilarlas mediante el comando *make*. Antes de este paso es necesario disponer en el sistema de los siguientes componentes:

- Compilador GNU de C++ (*g++*).
- Una tarjeta gráfica compatible con CUDA (NVIDIA GeForce 8800, NVIDIA Quadro FX 5600/4600).
- *CUDA Tookit*. Actualmente la última versión disponible es la 0.8 y se puede descargar de [NVI].
- Xmipp correctamente compilado e instalado.

Una vez que se tienen todos estos componentes en el sistema hay que descomprimir las fuentes en el directorio deseado. Para ello se debe abrir un

terminal y moverse al directorio destino donde queremos instalar el programa. Una vez allí, ejecutamos:

```
tar xvzf gpusart.tgz
```

Esto creará una nueva carpeta en el directorio actual que contendrá las fuentes de las versiones de GPU y CPU, así como el archivo *Makefile* para proceder a la instalación. A continuación debemos efectuar varios cambios al archivo *Makefile* para que refleje las rutas del *CUDA Tookit* y de Xmipp en nuestro sistema. Para esto, abrimos dicho fichero con cualquier editor de texto y modificamos las líneas

```
XMIPPPATH =  /Xmipp-1.1/
CUDAPATH = /usr/local/cuda/
```

para que apunten a los directorios donde están instalados Xmipp y el *CUDA Tookit* respectivamente. Una vez esto hecho, el paso de compilación se reduce a ejecutar el comando

```
make
```

que construirá ambos ejecutables. Estos ejecutables se ubicarán en el mismo directorio que sus respectivas fuentes, tanto el de CPU como el de GPU. Existen varios parámetros que se le pueden pasar a *make* dependiendo de lo que se quiera construir:

sartcpu Compila el ejecutable que solo hace uso de la CPU y lo guarda en *cpu/sart*. Utiliza todas las optimizaciones disponibles.

sartcpu-dbg Compila el ejecutable que solo utiliza la CPU y lo guarda en *cpu/sart*. Compila mostrando todos los mensajes de aviso e introduce la información de depuración en el ejecutable.

sartgpu Compila el ejecutable que utiliza la CPU y la GPU y lo almacena en *gpu/sart*. Usa todas las optimizaciones, tanto en la CPU como en la GPU

sartgpu-emu Igual que lo anterior, salvo que utiliza al opción de emulación para la parte de GPU. Con esta opción el ejecutable resultante (*gpu/sart-emu*) puede correr en sistemas sin tarjeta gráfica compatible con CUDA, ya que emula los *threads* de CUDA mediante *POSIX threads*.

gpu-cubin Genera el archivo *gpu/kernel.cubin*. Este archivo muestra información muy detallada de todos los *kernels* utilizados por la aplicación, como son el número de registros, memoria compartida utilizada...

clean Borra los ejecutables y otros archivos temporales que se generan durante la compilación.

A.2. Uso

La aplicación se ejecuta en modo terminal, y por ello todo lo referente a su uso trata acerca de los distintos parámetros que acepta desde la línea de comandos. Tanto la aplicación que utiliza sólo CPU como la que también utiliza la GPU como coprocesador aceptan los mismos argumentos (salvo dos excepciones). Estos parámetros y su utilidad son:

−?, −−help Muestra el mensaje de ayuda sobre el uso del programa y sale.

−i, −−input Archivo de selección que contiene la ruta de las proyecciones que se utilizarán para calcular la reconstrucción. Este archivo tiene el formato explicado en la sección 4.1. Si se omite este parámetro, el valor por defecto es *img.sel*

−o, −−output Ruta donde se almacenará el volumen reconstruido. Este volumen tiene el formato descrito en la sección 4.1, y su valor por defecto es *./flipas.vol*

−l, −−lambda valor Valor del factor de relajación λ que se aplicará durante la reconstrucción. El rango recomendado para este factor es entre 0 y 0,1. Si se omite, su valor por defecto es 0,01.

−b, −−blob La reconstrucción utiliza funciones de *Kaisser-Bessel* (blobs) como funciones base. Además de esto, se utiliza el método de proyección dirigida por vóxeles.

−v, −−voxel Se utilizan vóxeles como función base para la reconstrucción. Además, se aplica el método de proyección dirigida por rayos.

−n, −−noit Número de iteraciones que realizará a lo largo de todas las proyecciones. El comportamiento por defecto es aplicar sólo una, es decir, se recorren todas las proyecciones y se termina.

−s, −−placebo n Realiza una ordenación previa de las proyecciones maximizando el mínimo ángulo formado con las últimas n proyecciones, empezando por una de ellas al azar. Si este valor es 0 el ordenamiento es totalmente aleatorio.

−−tbproj (Sólo para la aplicación que utiliza GPU): Número de *threads* por bloque para la proyección.

−−tbback (Sólo para la aplicación que utiliza GPU): Número de *threads* por bloque para la backproyección.

Apéndice B

Visualizador de volúmenes Chimera

UCFS Chimera es un sistema interactivo de análisis y visualización de moléculas. Puede leer estructuras y datos moleculares en un gran número de formatos, mostrando dichas estructuras en una gran variedad de representaciones y generando imagenes tridimensionales de alta calidad. Proporciona herramientas que van desde la muestra de mapas de densidad al análisis de datos microscópicos y estructuras de alto nivel.
En este apéndice se describe el funcionamiento de *Chimera* únicamente para el tema que nos ocupa: la visualización tridimensional de volúmenes, detallando las diversas opciones que este fantástico programa nos ofrece.

B.1. Instalación

La instalación del programa es una tarea realmente simple: basta con dirigirse a la página web del programa (véase [UCS]) y descargar el programa disponible en la sección de descargas (tamaño $\sim$ 57MB). Una vez descargado solo hace falta instalarlo. Para ello se se debe abrir un terminal y dar permisos de ejecución al script de instalación:

```
chmod u+x chimera-xxxx.exe
```

Ejecutamos:

```
./chimera-xxx.exe
```

Empezará el proceso de instalación que por defecto creará una nueva carpeta en el directorio (/usr/local/chimera). Tras concluir la instalación basta con añadir /usr/local/chimera/bin al *path* del usuario.

B.2. Uso general

Lanzamos el programa desde un nuevo terminal, ejecutando:

```
chimera
```

El programa se abrirá y tendrá un aspecto parecido al de la figura B.1 que se muestra a continuación (dependiendo de la versión utilizada puede haber pequeñas diferencias).

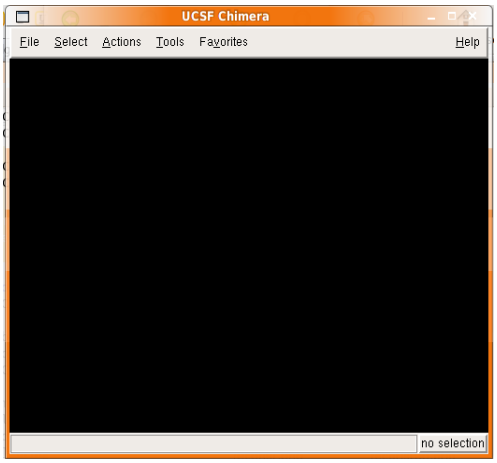

FIGURA B.1: UCFS Chimera, a extensible molecular modeling system. Pantalla principal.

Las principales funciones del menú que proporciona el programa son:

- **File**: En este submenú se puede cargar (*Open*) y buscar por identificador (*Fetch ID*) cualquier tipo de modelo molecular soportado por el programa. Entre las opciones más interesantes se encuentra la posibilidad de capturar imágenes del volumen a alta calidad (*Save Image*) para poder utilizarlas en diversas publicaciones.

- **Select**: Es posible seleccionar, dependiendo del modelo cargado, distintos componentes del mismo; en nuestro caso, al tratarse de volúmenes no es posible seleccionar elementos individuales (*Chains, Atoms, Residues...*).

- **Actions**: Acciones a ejecutar sobre los elementos que se pueden seleccionar. No se usará este submenu para el propósito de este proyecto.

- **Tools**: En este submenú se muestra una gran cantidad de herramientas disponibles según el tipo de modelo molecular que estemos tratando. En nuestro caso particular, se usará el menú *Volume Data*, que es el que contiene todas las herramientas para mostrar y analizar volúmenes. Más adelante se detallarán las principales opciones que nos encontramos.

B.2.1. Visualización de un volumen concreto

Para abrir un volumen, basta con ir a *File* → *Open*, y moverse hasta el directorio en el que tenemos guardado el volumen. Tras seleccionarlo, aparecerá en pantalla una reconstrucción tridimensional del modelo, y además una pantalla adicional llamada (*Volume Viewer*) que se explicará más adelante. Las opciones disponibles sobre el objeto son las siguintes:

- **Rotación**: con el botón izquierdo del ratón es posible rotar el volumen con total libertad.

- **Traslación**: manteniendo presionados ambos botones del ratón, se traslada el modelo por la escena.

- **Escalado**: por medio del botón derecho del ratón se puede aumentar o disminuir el tamaño del objeto.

B.2.1.1. *Display Styles*

Los volúmenes pueden ser mostrados en tres diferentes estilos: **superficie**, **malla** o **sólido**. Están disponibles en el *Volume Viewer* las opciones que permiten cambiar entre unos y otros. Los estilos de superficie y el de malla

son el mismo, excepto que éste último muestra también la triangulación de la superficie. El estilo sólido asigna un grado de transparencia a cada punto del volumen y muestra el volumen de manera semi-translúcida.

El nivel de detalle se controla cambiando la opcion de *Step*, situada en la parte superior derecha de la ventana de *Volume Viewer*. (*Tools* → *Volume Data* → *Volume Viewer*). Un paso de 1 1 1 utiliza todos los datos y muestra el mayor detalle posible.

Adicionalmente, se puede suavizar (*smooth*) la superficie. Ésta técnica mantiene el nivel de detalle pero reduce considerablemente los picos que se generan en la superficie. Ésta opción se encuentra en *Volume Viewer* → *Features* → *Surface and Mesh Options*. Activando la opción de *Surface smoothing iterations*, se utliza un filtro que aplica a cada vértice un valor medio de la suma de sus vértices adyacentes. Varios parámetros controlan el grado de suavizamiento, el número de iteraciones que se aplica el filtro y la máscara utlizada. Un ejemplo se muestra en la figura B.2

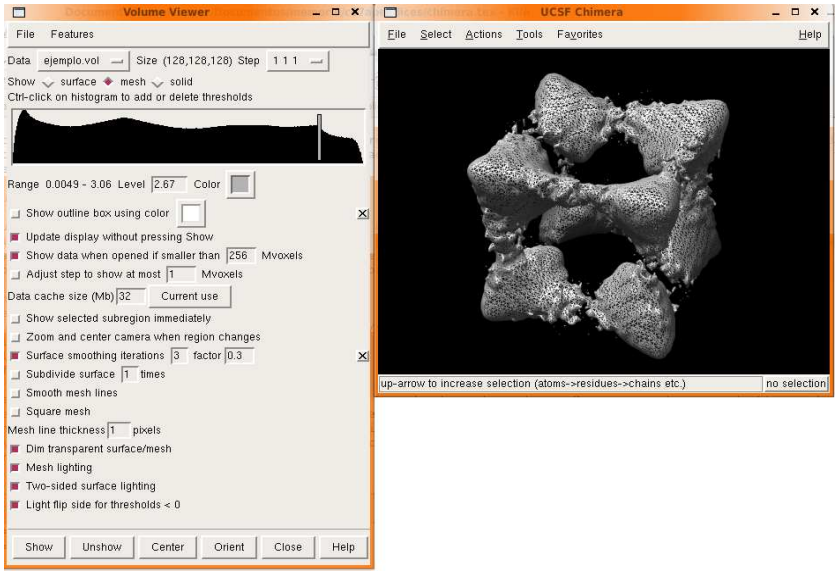

FIGURA B.2: Estilo de malla y Step fijado al máximo detalle (1 1 1).Aplicación de un filtro de suavizado con 3 iteraciones y un factor de media = 0.3

Destacar por último que se pueden añadir varios contornos de nivel. En *Volume Viewer*, si se mantiene presionada la tecla Control y se hace click en el histograma se añade un nuevo nivel. Además es posible aplicar distintos

estilos y colores a los diferentes contornos.

B.2.1.2. Coloreado

Las superficies pueden ser coloreadas de ditintas maneras. Usando la entrada del menú *Tools* → *Volume Data* → *Surface Color* podemos aplicar distintos modelos de color. Como se muestra en la figura B.3, podemos seleccionar que tipo de coloreado se quiera: radial (radius, aplica un coloreado radial), por volumen (todo el volumen pintado del mismo color) o por altura (*height*).

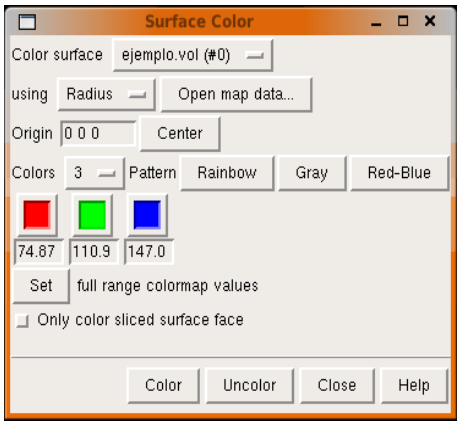

FIGURA B.3: Ventana Surface Color

Una vez seleccionado el tipo de patrón de coloreado, es hora de elegir los colores. Primero se selecciona el número de colores que se va a utilizar (en un rango de 2 a 20 colores). Tras esto, es posible asignarlos manual o automáticamente con combinaciones aleatorias (arcoiris, bicolor, escala de grises). Por último aplicamos el color cliqueando en *Set* y después en *Color*. De esta sencilla manera, se consigue un acabado más impactante y colorista, que hará que los volúmenes deslumbren. Un ejemplo de coloreado se muestra en la figura B.4

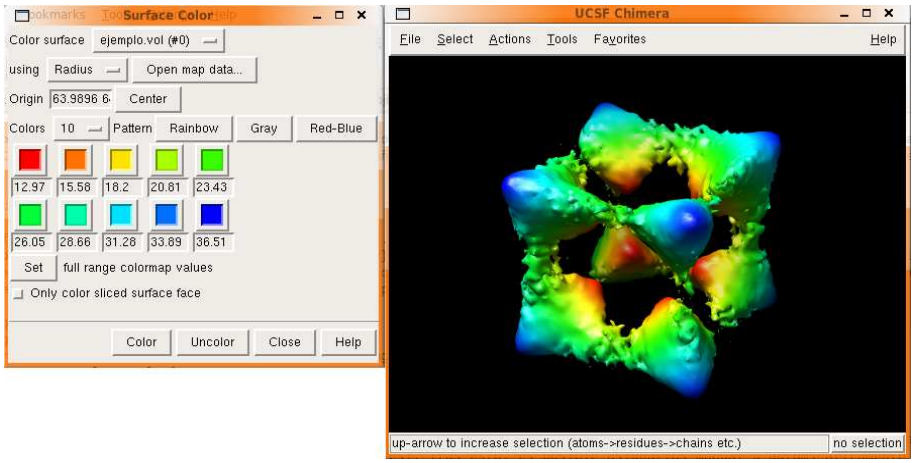

FIGURA B.4: Coloreado efecto radial, con 10 colores y patrón arcoiris.Origen en el centro del volumen

Apéndice C

Introducción a la microscopía electrónica de transmisión

Basado en [Sor02].

La biología nos ha enseñado que desde los seres vivos unicelulares hasta el más complejo de los pluricelulares puede ser estudiado a nivel local por el comportamiento de su constituyente más basico: la célula. La célula presenta toda la funcionalidad de un ser vivo completo (nutrición, reproducción y adaptación al entorno) y, para que todas estas funciones se cumplan, existen mecanismos bioquímicos que permiten a cada célula desempeñar la tarea encomendada. En estos mecanismos intervienen multitud de reacciones químicas en las que las principales protagonistas son moléculas de gran tamaño (macromoléculas) y complejidad normalmente basadas en muy pocos elementos (carbono, oxígeno, nitrógeno e hidrógeno).

Debido a que las moléculas implicadas en estas reacciones son específicas en la función que realizan, el estudio profundo de dichas moléculas permite caracterizar el comportamiento de una célula de forma constitutiva, y en último término de un tejido, órgano u organismo.

Los tipos de macromoléculas que participan en las reacciones químicas que se producen en el interior de la célula se pueden reunir en grandes familias denominadas glúcidos, lípidos, ácidos nucleicos y proteínas. De manera general, se puede decir que los glúcidos constituyen la fuente de energía de la

célula, los lípidos se emplean para la compartimentalización celular, los ácidos nucleicos se usan para la conservar la información necesaria para poder reproducir las proteínas en momentos posteriores y las proteínas protagonizan toda la vida de la célula. Es por ello que el estudio de las proteínas es un pilar fundamental en el análisis de la dinámica celular y ha provocado el nacimiento de distintas disciplinas (biología molecular, bioquímica, biología estructural) que abordan el problema desde diferentes perspectivas.

Se han diseñado diferentes estrategias para estudiar la estructura tridimensional que adopta una proteína en el espacio: predecirla a partir de los datos bioquímicos que se poseen de ella por medio de la resolución de complejas ecuaciones que definen la interacción de los diferentes átomos que componen la macromolécula y visualizar la partícula bajo estudio por medio de diferentes técnicas biofísicas (cristalografía de rayos X, espectroscopía por resonancia magnética nuclear o microscopía electrónica de transmisión).

La microscopía electrónica de transmisión es una técnica estructural de baja-media resolución que es cada vez más utilizada como una herramienta de gran potencial para realizar estudios estructurales a media-alta resolución, debido a que requiere una menor cantidad de proteína, no necesita un estado cristalino de la misma y no impone limitaciones de tamaño a la partícula bajo estudio. Su funcionamiento es similar al de la microscopía óptica aunque, en este caso, el haz de "luz" incidente no se compone de fotones de una determinada longitud de onda sino que son electrones acelerados por una considerable diferencia de potencial. Los electrones se producen por un filamento (cátodo) a alta temperatura en la parte superior del microscopio y son acelerados por una fuerte diferencia de potencial, producida por la presencia cercana de un ánodo en la que se ha practicado un diminuto orificio por el que salen los electrones en dirección a la columna del microscopio. Puesto que los electrones se dispersan por fenómenos de colisión con moléculas de gases, es necesario que se haga un fuerte vacío en el interior de la columna de manera que el haz de electrones alcance la muestra situada en el portamuestras. A lo largo de su trayectoria, dicho haz es focalizado por medio de dos lentes magnéticas (lente condensadora y lente objetivo) de

forma similar a la que las lentes de un microscopio óptico focalizan la luz. Algunos de los electrones del haz son dispersados por la muestra, otros interactúan con los átomos de la misma modificando únicamente su fase y el resto atraviesan la columna sin sufrir alteración alguna. La interacción de los electrones con la muestra será tanto mayor cuanto mayor sea la densidad local del material. En la base de la columna se encuentra el sistema de registro de imagen que puede ser analógico (pantalla fosforescente y una película fotográfica) o digital (formado por una cámara CCD) (ver figura C.1). La imagen captada recibe el nombre de micrografía.

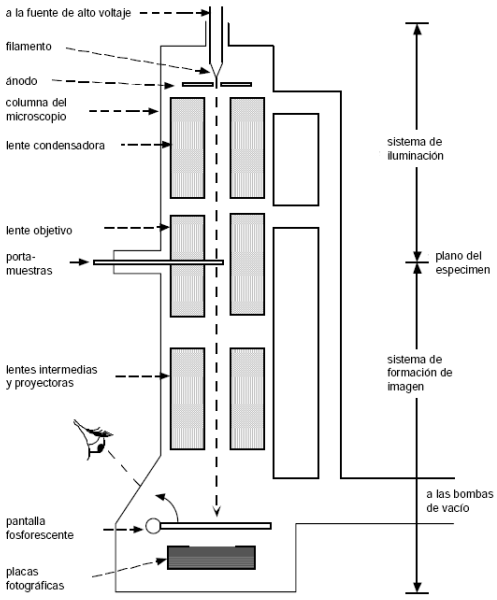

FIGURA C.1: Corte longitudinal del tubo de formación de imagen en un microscopio electrónico de transmisión. Fuente: [Sor02]

El hecho de que la muestra se encuentre en una situación de alto vacío en el interior de la columna impone algunas limitaciones en cuanto a la resolución de la imagen, por lo que se debe emplear algún modo de fijar su situación espacial. Actualmente se emplean principalmente dos técnicas: la tinción negativa y la criomicroscopía . La tinción negativa deposita sobre el espécimen una sal metálica y el contraste en la micrografía viene dado

principalmente por la interacción de los electrones con las moléculas de dicha sal; así pues esta técnica tan sólo proporciona información sobre la superficie exterior de la macromolécula bajo estudio. Además, la sal puede no llegar a recubrir la totalidad de la altura del espécimen, o puede dar lugar a deformaciones de la macromolécula bajo el peso del agente de tinción. La criomicroscopía, congela en un tiempo ultracorto una solución purificada de la proteína en estudio. La rápida congelación evita la formación de cristales de hielo que harían más difícil la interpretación de los resultados. Las ventajas principales de esta técnica son que la proteína se encuentra en un entorno más parecido al natural y que, ahora, sí se dispone de información proveniente del interior de la macromolécula. Como desventaja, el contraste de las imágenes es inferior al estar provocado con átomos de mucho menor número atómico que en el caso de la sal metálica. En los dos casos es necesario añadir una película de carbón que haga la muestra más rígida. Sin embargo, la tecnología ha avanzado en este sentido y actualmente se han conseguido micrografías de criomuestras en las que no se ha utilizado una película de carbón.

Para realizar la reconstrucción tridimensional de partículas aisladas se combinan imágenes de proyección procedentes de multitud de distintas macromoléculas, supuestas todas iguales en forma. No obstante, esto no es sino una aproximación puesto que diferencias en el agente de tinción o de su entorno pueden modificar ligeramente la estructura flexible de la proteína en estudio. Para finalizar, destacar que las micrografías obtenidas por microscopía electrónica se caracterizan por un bajo contraste de la señal frente al ruido, presentando una relación señal a ruido y por una resolución muy por debajo de la teórica impuesta por las limitaciones del microscopio.

Suponiendo que el espécimen es relativamente delgado con respecto a la profundidad de foco del microscopio, la imagen adquirida es una imagen de proyección (como ejemplo, las radiografías convencionales son imágenes de proyección). Entra en juego ahora, la técnica matemática de la tomografía, que resuelve la estructura tridimensional de la macromolécula que generó tales proyecciones.

Las etapas por las que pasa un conjunto de micrografías antes de producir

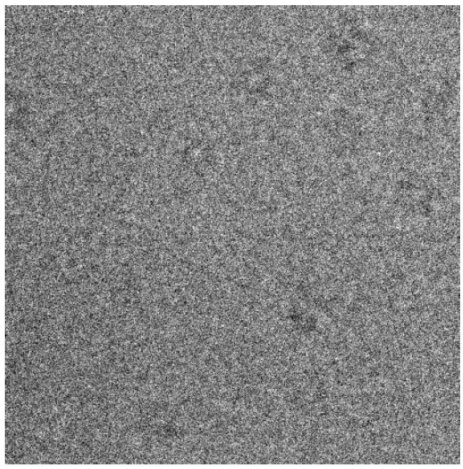

FIGURA C.2: Criomicrografía de ejemplo, sin haber utilizado soporte de carbón. Fuente: [Sor02]

una estructura tridimensional están descritas brevemente a continuación:

1. Suponiendo que la micrografía se encuentra en soporte fotográfico, el primer paso consiste en la digitalización correcta de la misma, de forma que pueda ser tratada por ordenador.

2. Posteriormente se procede a la selección y extracción de aquellas regiones dentro de la micrografía que corresponden a proyecciones de la proteína estudiada.

3. Se realiza a continuación una normalización de las imágenes de proyección con el propósito de homogeneizar los niveles de densidad de gris de imágenes procedentes de distintas micrografías.

4. Un primer paso de alineamiento traslacional y, dependiendo del caso, rotacional, es necesario para centrar todas las imágenes de proyección y colocarlas en un mismo marco geométrico.

5. Ahora se puede enventanar las imágenes para eliminar aquellas zonas en las que con seguridad no hay proyección de la proteína. De este modo se elimina parte del ruido existente.

6. Se prosigue con una clasificación de las imágenes de forma que se detecten posibles diferentes conformaciones y se desechen proyecciones que no correspondan a la partícula estudiada.

7. Se procede a una asignación angular, es decir, la determinación de la dirección desde la que cada imagen ha sido proyectada.

8. Por último se realiza la reconstrucción tridimensional y se visualiza el resultado obtenido.

Bibliografía

[AK84] A. H. Andersen and A. C. Kak. Simultaneous algebraic reconstruction technique (SART). *Ultrason.Img., vol.6: 81-94*, 1984.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[FSH01] Jr. F. S. Hill. *Computer Graphics Using OpenGL*. Prentice Hall International Editions, second edition, 2001.

[GA97] Luis Merino González and Evangelina Santos Aláez. *Álgebra lineal con métodos elementales*. Thomson Paraninfo, 1997.

[GG94] H. Guan and R. Gordon. A projection access order for speedy convergence of ART (algebraic reconstruction technique): a multilevel scheme for computed tomography. *Physics in Medicine and Biology*, (39):2005–2022, 1994.

[KS88] Avinash C. Kak and Malcolm Slaney. *Principles of Computerized Tomographic Imaging*. IEEE Press, 1988.

[MHC98] R. Marabini, G. T. Herman, and J. M. Carazo. 3D reconstruction in electron microscopy using ART with smooth spherically symmetric volume elements (blobs). *Ultramicroscopy*, 72:53–65, 1998.

[MMM+96] R. Marabini, I. M. Masegosa, M. C. San Martín, S. Marco, J. J. Fernández, L. G. de la Fraga, C. Vaquerizo, and J. M. Carazo.

Xmipp: An image processing package for electron microscopy. *Journal of Structural Biology*, 116:237–240, 1996.

[MOS01] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, 2001.

[MY99] Klaus Mueller and Roni Yagel. On the use of graphics hardware to accelerate algebraic reconstruction methods. In *Proceedings of SPIE Medical Imaging Conference*, 1999.

[MYW99] Klaus Mueller, Roni Yagel, and John J. Wheller. Fast implementations of algebraic methods for three-dimensional reconstruction from cone-beam data. *IEEE Transactions on Medical Imaging*, 18(6), 1999.

[NVI] NVIDIA. NVIDIA CUDA homepage. http://developer.nvidia.com/object/cuda.html.

[NVI05] NVIDIA. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. 2005.

[NVI07] NVIDIA. *CUDA Programming Guide 0.8.2*, 2007.

[SMB+01] C. O. S. Sorzano, R. Marabini, N. Boisset, E. Rietzel, R. Schroder, G. T. Herman, and J. M. Carazo. The effect of overabundant projection directions on 3D reconstruction algorithms. *Journal of Structural Biology*, 2001.

[SMVM+04] C. O. S. Sorzano, R. Marabini, J. Velazquez-Muriel, J. R. Bilbao-Castro, S. H. W. Scheres, J. M. Carazo, and A. Pascual-Montana. Xmipp: a new generation of an open-source image processing package for electron microscopy. *Journal of Structural Biology*, 148(2):194–204, 2004.

[Sor02] Carlos Óscar Sánchez Sorzano. *Algoritmos Iterativos de Tomografía Tridimensional en Microscopía Electrónica de Transmisión*. PhD thesis, Universidad de Málaga, 2002.

[Str97] Bjarne Stroustroup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1997.

[Tan71] K. Tanabe. Projection method for solving a singular system. *Numer. Math.*, 17:203–214, 1971.

[UCS] UCSF Chimera home page. http://www.cgl.ucsf.edu/chimera/.

[Wil00] Matthew Willis. Algebraic reconstruction algorithms for remote sensing image enhancement. Master's thesis, Department of Electrical and Computer Engineering, BYU University, 2000.

[XM05] Fang Xu and Klaus Mueller. Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. *IEEE Transactions on Nuclear Science*, 52(3), Junio 2005.

[Xmi] Xmipp wiki. http://xmipp.cnb.csic.es/.