Universidad de Almería

Tesis Doctoral

Reconstrucción tomográfica ultrarrápida en procesadores multicore

José Ignacio Agulleiro Baldó

Director de la tesis: José Jesús Fernández Rodríguez

Almería, 2 de marzo de 2011

A mis padres

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Agradecimientos

Al (1) Programa FPU (Formación del Profesorado Universitario) del Ministerio de Educación, (2) al Plan Nacional de I+D+i del Ministerio de Ciencia e Innovación (proyectos TIN2005-00447 y TIN2008-01117), (3) al Plan Andaluz de Investigación (Proyecto de excelencia P06-TIC-01426), (4) al programa de proyectos intramurales especiales del Consejo Superior de Investigaciones Científicas (CSIC, PIE200920I075) y (5) al Sexto Programa Marco de la Unión Europea (Red de excelencia 3DEM, LSHG-CT-2004-502828). La financiación de estos programas y planes ha hecho posible el trabajo desarrollado en esta tesis.

Al Ministerio de Educación por financiar parte de mi estancia en el Centro Nacional de Biotecnología (CSIC) a través de las ayudas para estancias breves que concede a los becarios FPU. La estancia se extendió desde el 28 de septiembre de 2009 hasta el 30 de junio de 2010, y la ayuda recibida fue para los meses de mayo y junio.

Al Departamento de Estructura de Macromoléculas del Centro Nacional de Biotecnología (CSIC) por haberme dado acceso a su cluster 'Crunchy', donde se realizaron buena parte de los experimentos de esta tesis, en concreto aquellos en los que se usó el procesador E5405.

Al grupo "Supercomputación: algoritmos" de la Universidad de Almería por haberme proporcionado la infraestructura computacional necesaria para el desarrollo de esta tesis, con especial mención a José Antonio Martínez García y José Manuel Molero Pérez, encargados de la gestión y mantenimiento de dicha infraestructura.

Huyo de ser prolijo, pero no quiero olvidar a nadie. Venid conmigo, que debéis estar aquí:

Inma, que hiciste posible mi contrato inicial en el Departamento de Arquitectura de Computadores y Electrónica de la Universidad de Almería.

Ester, que te fijaste en mí cuando eras mi profesora y me animaste a emprender la tarea de esta tesis. Has sido siempre asilo tutelar y consejera personal.

José Jesús (Jose), que confiaste en las ideas que yo sugerí, me diste libertad a la hora de investigar y me atendiste siempre que te lo requerí, sin dilación ni objeción alguna. Ojalá todos los jefes que me queden por tener sean como tú.

Ricardo, Marina, Mari Ángeles, Michele (Bambini) y las otras personas que conocí durante los meses que pasé en el CNB. Y muy especialmente Elías, que me ayudaste mucho más de lo que crees.

Juani, con quien compartí despacho y rutina de trabajo diaria. Y Juan. Con vosotros siempre es un placer hablar.

Eduardo, Sergi y Vicen, amigos de todas las horas.

Verónica, que caminas a mi lado y haces pequeña la distancia.

Mi hermana María del Mar, a quien más quiero en este mundo. Tú sabes darme buenos consejos, me enseñas cosas valiosas aun siendo menor que yo, me escuchas, me aguantas y me permites pensar en voz alta. No podría expresar con palabras lo que significas para mí.

Mis padres y maestros, José María y Encarnita, que me habéis dado los valores y educación que tengo, os habéis desvivido por mí y habéis sido siempre incondicionales en vuestra ayuda. Papá y mamá, este libro está dedicado a vosotros. Os quiero.

Y tantos otros que en mi memoria siempre estáis.

José Ignacio Agulleiro Baldó
Almería, febrero de 2011

Prólogo

La tomografía electrónica posibilita la visualización de la estructura tridimensional (3D) de especímenes biológicos a resolución molecular. Para alcanzar dichos niveles de resolución es necesario que se tomen grandes imágenes de proyección de los especímenes, lo cual permitirá obtener reconstrucciones en 3D de calidad. El proceso de reconstrucción a esta escala es muy exigente en cuanto a recursos computacionales se refiere y requiere un tiempo considerable.

El procedimiento investigado en esta tesis tiene como objetivo el cálculo rápido de reconstrucciones tomográficas. Esta rapidez es especialmente necesaria en sistemas de tomografía de tiempo real, concebidos para que el usuario adquiera imágenes del espécimen y obtenga la estructura 3D del mismo tan pronto como sea posible. De esta forma se podrá valorar la calidad de la muestra sin pérdida de tiempo y decidir si es conveniente tomar más imágenes de ella. La rapidez también se precisa en sistemas interactivos de procesamiento y visualización de imágenes, donde lo deseable es que el usuario no tenga que esperar demasiado para recibir el resultado de la reconstrucción. Tradicionalmente se han empleado computadores paralelos o clusters de computadores para acelerar el cálculo de la reconstrucción 3D, si bien en la actualidad las tarjetas gráficas (GPUs, *Graphics Processing Units*) están siendo explotadas enormemente debido a su excelente relación rendimiento/coste. No obstante, el inconveniente reside en la necesidad de ese hardware específico (GPU) que ha de estar disponible en el computador y que ha de poseer unas determinadas características para ser empleado en procesos de cálculo de propósito general.

Nuestro procedimiento se basa en la optimización de código y el aprovechamiento de todas las capacidades de los potentes computadores estándar actuales —e.g. varios núcleos de computación en un mismo procesador físico, procesamiento vectorial, rápidas memorias caché— con el fin de proporcionar métodos rápidos de reconstrucción 3D que sean competitivos con las implementaciones en GPUs y no exijan ningún hardware especial, lo cual supone una gran ventaja de cara a la distribución del software. Por otro lado, y puesto que no es precisa la gestión de hardware alguno —cluster o GPU—, nuestro procedimiento facilitará la instalación y uso del software en laboratorios científicos (de biología o medicina, por ejemplo) donde generalmente los investigadores no son expertos en computadores.

En resumen, el objetivo general de esta tesis es la producción de algoritmos de reconstrucción tomográfica muy rápidos que puedan ser utilizados incluso en entornos de tomografía de tiempo real y que puedan correr en computadores estándar sin necesidad de ningún hardware especial, de modo que se facilite su distribución y su uso en los laboratorios científicos de tomografía. Para conseguir este objetivo principal nos serviremos de un objetivo puente: aprovechar al

máximo las características de los actuales procesadores multicore basados en la arquitectura x86.

Esta tesis se encuentra dividida en seis capítulos. El primero de ellos es una introducción a la tomografía electrónica, donde se explicará en qué consiste esta técnica, se presentarán los algoritmos de reconstrucción más empleados en el campo, WBP y SIRT, y se hará un repaso de las diferentes plataformas computacionales que se han usado en tomografía electrónica. Los capítulos 2, 3 y 4 detallan las distintas optimizaciones aplicadas a WBP y SIRT. El 2 se ocupa de las básicas, el 3 se centra en el procesamiento vectorial y el 4 trata el aprovechamiento de los núcleos de un procesador multicore y la optimización de la E/S a disco. El capítulo 5 muestra y analiza los resultados experimentales obtenidos y, por último, el 6 expone las conclusiones que se desprenden de la investigación llevada a cabo en esta tesis y apunta nuevas y futuras vías de optimización aplicables al proceso de reconstrucción 3D.

Capítulo 1

La tomografía electrónica

La tomografía es una técnica empleada en diversas ciencias para hallar la estructura 3D de un objeto. Por ejemplo, en geología puede ser utilizada para desvelar la de un mineral; en ciencia de materiales, para encontrar imperfecciones en un metal; en biología, para conocer la estructura de células o virus. Existen diferentes tipos de tomografías (electrónica, de rayos X, etc.), pero los procedimientos para obtener la estructura 3D son básicamente los mismos y únicamente varía el instrumento de adquisición de imágenes. Dependiendo de las características del objeto que se quiera examinar, se elegirá una u otra. En tomografía electrónica se usa el microscopio electrónico de transmisión para estudiar especímenes biológicos.

Este capítulo comienza con una introducción a la tomografía electrónica. Seguidamente serán presentados los procedimientos de reconstrucción más empleados en el campo: WBP y SIRT. Por último, haremos un repaso de las distintas plataformas de computación usadas en tomografía electrónica.

1.1. Introducción a la tomografía electrónica

La microscopía electrónica es una poderosa herramienta utilizada en la biología moderna que se sirve del microscopio electrónico y de sofisticadas técnicas de procesamiento de imagen y de reconstrucción tridimensional (3D) para generar la estructura 3D de especímenes biológicos. El funcionamiento del microscopio electrónico es similar al del óptico, si bien en lugar de luz visible se emplean electrones acelerados en el vacío, pues se comportan igual que la luz [27]. La información estructural obtenida es de vital importancia para comprender la función biológica de los especímenes.

Existen diversas geometrías de adquisición de datos, entre las que encontramos las de eje único de giro, las de doble eje y las cuasi-cónicas. Independientemente de la geometría empleada, el resultado es un conjunto de imágenes obtenidas mediante el microscopio electrónico que serán combinadas por las técnicas de reconstrucción para derivar la estructura 3D del espécimen. Tanto los procesos de adquisición de imágenes como los de reconstrucción 3D han de ser robustos en el tratamiento del ruido y han de proporcionar niveles de resolución adecuados, todo ello para permitir una correcta interpretación de las características estructurales. Los métodos de reconstrucción suelen ser compu-

tacionalmente intensivos y precisan tiempos de procesamiento considerables en computadores convencionales.

En microscopía electrónica existen distintas formas de abordar la determinación de la estructura 3D de los especímenes biológicos. La adopción de una u otra depende de la naturaleza del espécimen —e.g. su tamaño, su organización en forma de agregados o no—. Para especímenes grandes y/o complejos se suele recurrir a la tomografía electrónica (TE) [44]. Sin embargo, cuando el espécimen es relativamente pequeño la metodología de reconstrucción se conoce como análisis de partículas individuales (SPA, *Single Particle Analysis*) [45].

En tomografía electrónica una sola muestra de un espécimen es expuesta al microscopio electrónico y se obtienen vistas de la misma a diversos ángulos de inclinación siguiendo geometrías de eje único de giro[1] (Figura 1.1) o de doble eje. El resultado de la adquisición de datos es un conjunto de 60 a 200 imágenes de proyección con tamaños típicos de 512×512, 1024×1024 o 2048×2048 píxeles. La determinación de la estructura 3D implica combinar ese lote de imágenes mediante algoritmos de reconstrucción para generar volúmenes que pueden llegar a ocupar varios gigabytes. La tomografía electrónica está jugando un papel muy importante en biología celular [35, 65, 66]. Los especímenes susceptibles de ser estudiados en TE suelen tener tamaños del orden de micras, situándose en el rango subcelular. Ejemplos son mitocondrias, dendritas, retículo endoplasmático o virus de gran tamaño. Esta técnica es directamente comparable a la tomografía axial computerizada utilizada en medicina.

En el análisis de partículas individuales, las micrografías adquiridas con el microscopio contienen muchas ocurrencias de un mismo espécimen a distintas vistas (Figura 1.2). Las ocurrencias relacionadas con una misma vista se agrupan y se alinean, generándose decenas de miles de imágenes, cuyos tamaños pueden ir desde 64×64 a 512×512 píxeles, dependiendo del estudio que se esté llevando a cabo. Una vez alineadas, las imágenes son promediadas y se obtiene una imagen representativa de cada vista. La aplicación de algoritmos de reconstrucción sobre estas imágenes para la determinación de la estructura 3D origina volúmenes del orden de decenas o cientos de megabytes. Los especímenes susceptibles de estas técnicas se sitúan en el rango macromolecular, siendo sus tamaños inferiores a 100 nanómetros (e.g. proteínas).

Las tendencias actuales en microscopía electrónica hacen uso de técnicas de criomicroscopía con objeto de preservar los detalles estructurales del espécimen. En criomicroscopía las muestras biológicas se congelan y se exponen al microscopio empleando muy baja dosis electrónica. Como resultado, las imágenes obtenidas presentan muy bajo contraste y son extremadamente ruidosas. Ante estas condiciones, es de vital importancia el desarrollo de algoritmos de reconstrucción que presenten un comportamiento robusto frente al ruido.

Como hemos visto, la naturaleza del espécimen impone el procedimiento que se ha de seguir a la hora de adquirir las imágenes con el microscopio. No obstante, los algoritmos de reconstrucción son independientes de dicho procedimiento, aunque nosotros los enfocaremos a tomografía electrónica. El algoritmo estándar de reconstrucción en el campo de la tomografía de especímenes biológicos es *Weighted Backprojection* (WBP) [44], y es objeto de estudio en esta tesis. La relevancia de este algoritmo viene determinada por su relativa simplicidad computacional. Sin embargo, tiene grandes desventajas. Por un lado, los resul-

[1] En esta tesis nos centramos sólo en este tipo de geometría.

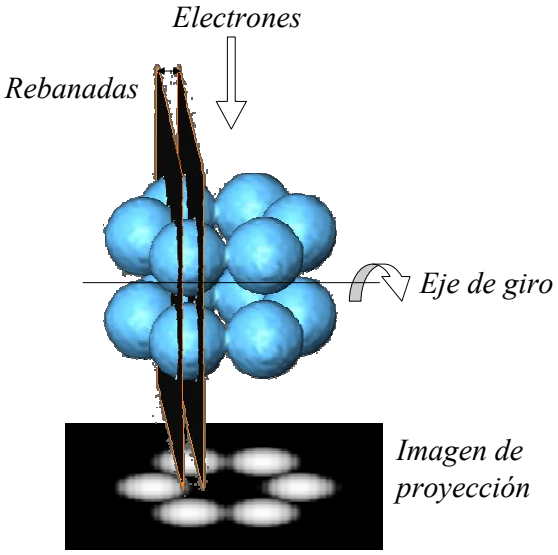

Figura 1.1: Geometría de adquisición de eje único de giro. El microscopio emite electrones siempre en la misma dirección y sentido según indica la flecha mientras el espécimen es girado en torno al eje en rangos de $[-60°, +60°]$ o $[-70°, +70°]$ con pequeños incrementos de 1 o 2 grados. A cada ángulo se obtiene una imagen de proyección distinta, siendo el conjunto de todas ellas necesario para generar la reconstrucción 3D. En esta geometría en particular, el espécimen puede considerarse formado por rebanadas perpendiculares al eje. Cada imagen de proyección contiene información acerca de todas ellas.

tados pueden presentar fuertes artefactos debido a la imposibilidad física del microscopio de obtener todas las posibles vistas del espécimen. Por otro, es muy sensible al ruido. Los métodos iterativos [52] constituyen una de las principales alternativas a WBP, pues son capaces de proporcionar soluciones más suaves, lo cual los hace muy atractivos en situaciones con alto nivel de ruido. A pesar de sus virtudes, aún no han sido aplicados de forma extensiva en reconstrucción 3D a causa de su alto coste computacional. El otro algoritmo que se estudiará en esta tesis, *Simultaneous Iterative Reconstruction Technique* (SIRT) [61], pertenece a la categoría de métodos de reconstrucción iterativos.

1.1.1. Adquisición de datos detallada en TE

En este apartado vamos a analizar de forma más detallada la adquisición de datos vista en la Figura 1.1. Si bien en TE es el espécimen el que se mueve mientras el microscopio queda fijo, aquí supondremos que ocurre al contrario, lo cual no afecta a los datos que se obtienen del espécimen, pero el proceso es más sencillo de entender. Este proceso aparece ilustrado en la Figura 1.3. El detector es el dispositivo del microscopio que recibe los rayos de electrones (en la figura los rayos tienen una flecha que va hacia el detector porque le devuelven información) e irá rotando alrededor del espécimen para obtener las imágenes de proyección. Nótese que en la Figura 1.3 el objeto representa únicamente a una rebanada y no al espécimen completo. Por tanto, la función g es la proyección

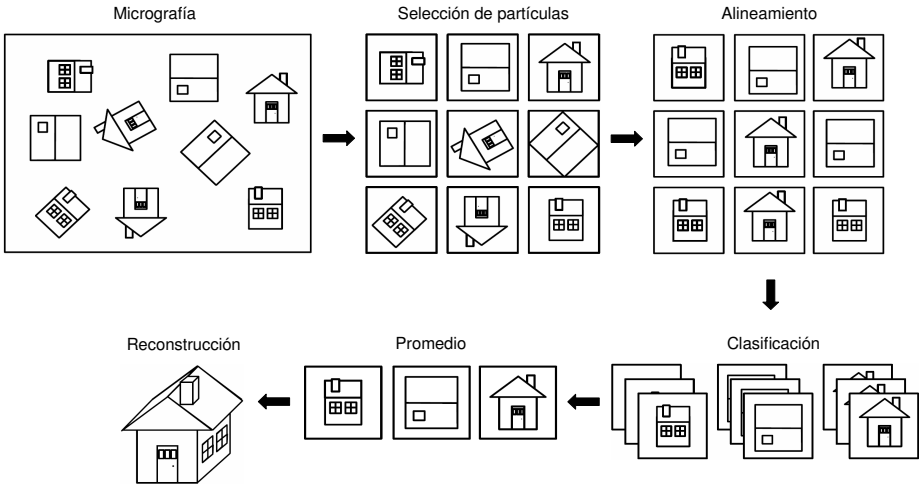

Figura 1.2: Análisis de partículas individuales. Como se aprecia en la figura, se tiene una micrografía con ocurrencias a distintas vistas (planta, alzado y perfil) de un mismo espécimen (una casa). Las ocurrencias (o partículas) se seleccionan y se alinean. Luego son clasificadas y promediadas para obtener una imagen representativa de cada vista. El proceso culmina al aplicar los algoritmos de reconstrucción a estas imágenes representativas. Figura reproducida de [81].

unidimensional (1D) de la rebanada en cuestión. El proceso se repetirá para las otras rebanadas, constituyendo la unión de todas las proyecciones 1D una imagen de proyección. θ es el ángulo que forma el detector con el eje X y es al que se toman las proyecciones. Como vemos, el espécimen siempre se sitúa en el centro de los ejes cartesianos y el giro se hace con respecto al origen de coordenadas.

Las rebanadas las debemos considerar formadas por un conjunto de puntos, cada uno de ellos con un nivel de densidad asociado[2]. $f(x, y)$ es una representación en niveles de densidad del objeto y nos proporciona el valor de densidad para cada punto (x, y). Esta función es la que a nosotros nos interesa conocer. $g(r, \theta)$ constituye la proyección de $f(x, y)$ a un ángulo θ y almacena la suma de las densidades recorridas por los rayos de electrones (información devuelta al detector), no conociéndose ni la posición de dichas densidades en el objeto ni su cantidad. Como consecuencia, una única proyección es insuficiente para obtener la distribución de densidades en el objeto de interés. Tal y como muestra la Figura 1.3, dependiendo de la suma que devuelva cada rayo, para una posición r, g devolverá mayor o menor densidad.

El resultado del proceso de adquisición descrito es un conjunto de imágenes de proyección. Cada una ha sido tomada a un ángulo distinto y contiene información sobre todas las rebanadas del espécimen. En particular, de cada rebanada almacena una proyección 1D. Téngase en cuenta que cada vez que situamos el detector a un determinado ángulo, obtiene proyecciones 1D de todas las rebanadas al mismo tiempo, las cuales forman la imagen de proyección a ese ángulo. Esto es sencillo de entender si imaginamos la existencia de varios

[2]Nótese que al hacer esto estamos discretizando el objeto.

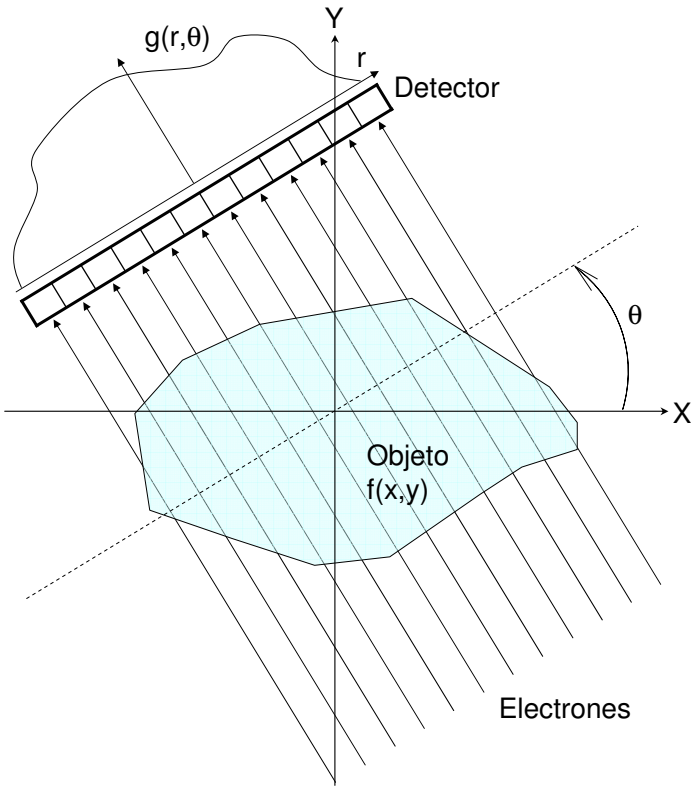

Figura 1.3: Geometría de adquisición detallada. ¿Qué ocurre en cada rebanada? Dado un ángulo θ, los rayos de electrones atraviesan el objeto y le devuelven al detector la suma de las densidades recorridas. Cada posición r del detector guarda una suma, pero no se conocen cuántas densidades la componen ni la posición en el objeto de las mismas. Se ha transformado así una imagen bidimensional en una unidimensional.

detectores —uno por rebanada— trabajando al unísono. Los algoritmos de reconstrucción procesan una a una las rebanadas del espécimen. Puesto que los datos referentes a una rebanada están esparcidos entre todas las imágenes de proyección, es mucho más óptimo juntarlos para procesarlos. Esta estructura de datos recibe el nombre de sinograma y se obtiene según indica la Figura 1.4.

Al final, hemos convertido cada rebanada del espécimen en un sinograma. Esta operación recibe el nombre de proyección y se modela matemáticamente mediante la transformada de Radon discreta:

$$G_i = \sum_{j=1}^{m} A_{i,j} R_j^\star \qquad 1 \leq i \leq n \tag{1.1}$$

La Ecuación 1.1 calcula la contribución de cada píxel j de la rebanada $R^\star$ al valor de cada posición i del sinograma G, lo cual viene determinado por la matriz de coeficientes (o pesos) A. Cuando la contribución sea nula, la matriz almacenará un 0 para el par (i, j) correspondiente. $n = n_x n_y$ y $m = m_x m_y$ son las dimensiones del sinograma y la rebanada, respectivamente. n_x viene limitado

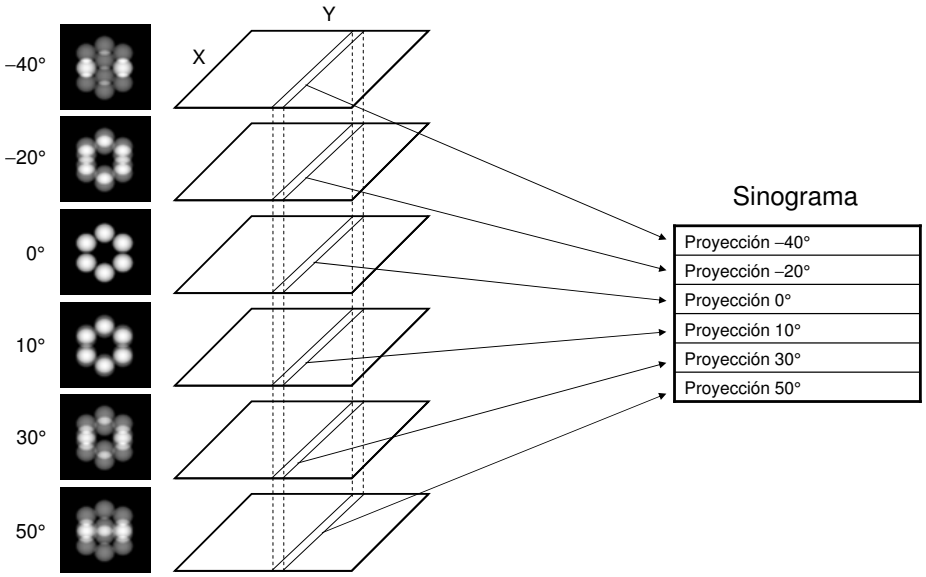

Figura 1.4: Obtención de un sinograma. Todas las imágenes de proyección se apilan y las proyecciones 1D (o, simplemente, proyecciones) que pertenecen a la misma rebanada (aquellas entre las líneas verticales punteadas) se agrupan en un sinograma. Este proceso se repite para cada rebanada, por lo que tendremos tantos sinogramas como rebanadas.

por la longitud del detector y n_y es igual al número de ángulos empleado en el proceso de adquisición. En general, se cumplirá que $m_x = m_y = n_x$[3].

La matriz A se calcula como explicamos a continuación. Primero hay que conocer si el píxel R_j se corresponde con la posición G_i. Tanto el sinograma como la rebanada son imágenes en 2D, por lo que i y j son posiciones absolutas en dichas imágenes (debemos considerar que las filas están colocadas consecutivamente). En realidad, cada i da lugar a una coordenada (θ, r), al igual que cada j es en realidad un par (x, y). Así pues, para hallar la correspondencia dado un ángulo θ se aplica la Ecuación 1.2 [18].

$$r = xcos\theta + ysen\theta \tag{1.2}$$

Si r no está en el intervalo $[1, n_x]$, entonces el píxel R_j no contribuye a G_i, y $A_{i,j} = 0$. Si obtenemos un valor dentro del citado intervalo, pueden darse dos situaciones: el valor es entero o es real. En el primero de los casos, la correspondencia es directa, por lo que $A_{i,j} = 1$. En el segundo, nos encontramos entre dos posiciones del sinograma. Por ejemplo, si $r = 4{,}3$, estamos entre 4 y 5. Aquí se aplica una interpolación, de manera que R_j contribuirá con mayor intensidad a la posición más cercana a r y menos a la más lejana. Así, en 4 la contribución será de $A_{i,j} = 5 - 4{,}3 = 0{,}7$, mientras que en 5 valdrá $A_{i+1,j} = 0{,}3$.

El problema de la reconstrucción puede formularse como sigue: dado un sinograma, ¿cuál es la distribución de densidad de la rebanada con la que se

[3]Tanto m_x como m_y se pueden reducir durante la reconstrucción para focalizarla.

corresponde? De ello se encargan los algoritmos de reconstrucción, que pasamos a ver en el siguiente apartado.

1.2. Los algoritmos de reconstrucción WBP y SIRT

En este apartado explicaremos el funcionamiento de los algoritmos *Weighted Backprojection* (WBP) y *Simultaneous Iterative Reconstruction Technique* (SIRT). Como ya hemos apuntado, WBP constituye actualmente el estándar en tomografía electrónica por su simplicidad computacional, si bien es muy sensible al ruido. Esto provoca que las reconstrucciones 3D no tengan una calidad muy buena. SIRT es uno de los métodos iterativos más comunes. Los volúmenes 3D generados por este algoritmo tienen una alta calidad, pero en su contra está la gran cantidad de tiempo que requiere.

Tal y como hemos visto, el espécimen del que deseamos conocer la estructura 3D se divide en rebanadas. De cada una de ellas se obtiene un conjunto de proyecciones 1D tomadas a distintos ángulos que se agrupan en una estructura de datos llamada sinograma. Las rebanadas son reconstruidas individualmente a partir del sinograma correspondiente mediante un algoritmo de reconstrucción y, una vez que todas han sido procesadas, se apilan para crear la estructura 3D. Por tanto, podemos reducir la explicación del problema de la reconstrucción a una única rebanada y extenderlo al resto, pues el procedimiento es siempre el mismo.

1.2.1. Weighted Backprojection

Para llevar a cabo la reconstrucción, backprojection distribuye los niveles de densidad presentes en las proyecciones 1D de un sinograma sobre los mismos puntos de la rebanada en los que incidieron los rayos del microscopio cuando dichas proyecciones se obtuvieron, es decir, efectúa la operación inversa a la realizada por el microscopio (sería como cambiarle el sentido a las flechas de los electrones en la Figura 1.3). Al ir repitiendo este proceso con cada proyección 1D del sinograma, la densidad en cada punto de la rebanada se va reforzando y, por tanto, ésta es reconstruida. Una vez que todas las rebanadas han sido reconstruidas, éstas se apilarán y darán lugar al volumen 3D. Para entender el procedimiento, podríamos pensar en un cilindro opaco que cortamos en rebanadas. Dichas rebanadas son reveladas por separado y luego vueltas a poner juntas, permitiendo entonces que el interior del cilindro pueda ser observado. El proceso de reconstrucción de una rebanada se puede apreciar en la Figura 1.5. Obsérvese que cuantas más proyecciones 1D se usen, mejor será la reconstrucción, pero nunca el resultado final será exactamente igual a la imagen original. Esto se debe al emborronamiento típico que introduce backprojection, efecto que puede atenuarse aplicando un filtrado al sinograma antes de proceder con la reconstrucción y del que hablaremos más adelante.

El algoritmo backprojection realiza la operación de retroproyección, es decir, la "inversa" de la proyección (Ecuación 1.1), quedando modelado matemáticamente de la siguiente manera:

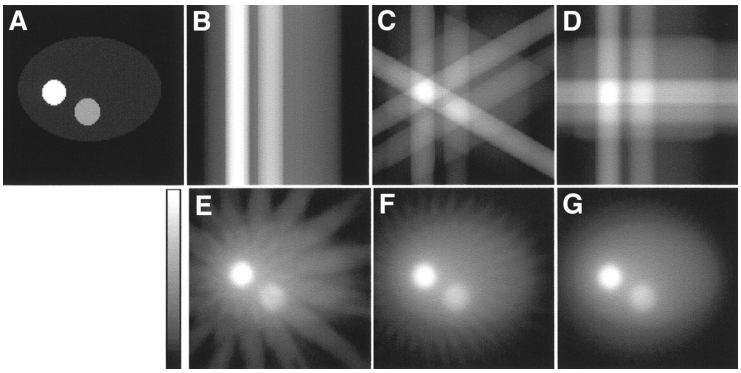

Figura 1.5: Imagen reconstruida con backprojection. No se aplica filtrado. La imagen A representa la rebanada de la que se tomaron las proyecciones, mientras que las imágenes B-G son las reconstrucciones de A al emplear 1, 3, 4, 16, 32 y 64 proyecciones, respectivamente. Tal y como se muestra, cuantas más se usen, mejor será la reconstrucción. Figura tomada de [18].

$$R_j = \sum_{i=1}^{n} B_{j,i} G_i \qquad 1 \leq j \leq m \tag{1.3}$$

En realidad, no se lleva a cabo la inversa de la proyección, pues es imposible asignar a los píxeles de la rebanada sus valores originales de densidad, sino que reciben una suma global. En la Ecuación 1.3 esto queda reflejado al escribir R en vez de $R^\star$, o sea, lo obtenido no es igual a lo original, si bien cuantas más proyecciones tenga el sinograma G, mejor será la aproximación. B es la traspuesta de la matriz A que aparecía en la Ecuación 1.1. Las demás variables tienen el mismo significado en ambas ecuaciones. Backprojection suele ir acompañado de una operación de filtrado previa a la reconstrucción que se aplica a los sinogramas. En la Ecuación 1.3 puede suponerse que G está filtrado.

La implementación de WBP se muestra en el Algoritmo 1.1. El bucle más externo (línea 1) va seleccionando las rebanadas una a una para reconstruirlas. Recordemos que la reconstrucción de una rebanada se hace a partir del sinograma relacionado con ella, y que un sinograma está formado por cierto número de proyecciones 1D, cada una tomada a un ángulo distinto. Dichas proyecciones se procesan de modo secuencial (línea 4). Por otro lado, los bucles en las líneas 5 y 6 se encargan de recorrer todos los píxeles de la rebanada actual. Antes de retroproyectar una posición del sinograma (línea 10), es necesario conocer la correspondencia entre la coordenada (x, y) que se esté tratando y dicha posición. Ese cálculo (línea 7) coincide con la Ecuación 1.2. Puesto que el resultado no tiene por qué ser entero, debemos interpolar (líneas 8, 9 y 10) tal y como se detalló al explicar la Ecuación 1.1, salvo que ahora los pesos multiplican al sinograma y no a la rebanada.

Filtrado en backprojection

El grave inconveniente de backprojection es el emborronamiento que tienen las imágenes reconstruidas [77], efecto que se produce debido a que todos los

Algoritmo 1.1: Implementación de WBP

```
for s in Nslices {
   /* Reconstrucción de una rebanada */
   set_to_zero(slice[s])
   for a in Nangulos
      for y in Nfilas
         for x in Ncols {
            rf = projected_point(x, y, a)
            r = (int)rf
            weight = rf - r
            slice[s][y][x] = slice[s][y][x] + ...
                sino[s][a][r+1]*weight + ...
                sino[s][a][r]*(1-weight)
         }
}
```

píxeles de la rebanada por los que pasa un rayo reciben la suma total de niveles de densidad que se obtuvo durante la adquisición de las proyecciones y no la suya propia. Por ejemplo, si durante la proyección un rayo atravesó tres píxeles de una de las rebanadas del espécimen en las que los niveles de densidad fueron 5, 2 y 7, en la retroproyección cada uno de los tres píxeles recibirá como densidad la suma, es decir, 14. La Figura 1.6 muestra el efecto del emborronamiento. Dicho efecto no se puede eliminar completamente, pero sí es posible atenuarlo filtrando las proyecciones adquiridas del espécimen con un filtro paso alto, de manera que resaltemos los perfiles y contornos de las imágenes. Cuando se utiliza filtrado en backprojection, éste recibe el nombre Weighted Backprojection (WBP) o Filtered Backprojection (FBP). El filtrado siempre se suele aplicar, ya que si no es así, las imágenes tendrán una calidad pobre que imposibilitará el hacer un examen acertado de las mismas.

1.2.2. Simultaneous Iterative Reconstruction Technique

SIRT consiste en la aplicación reiterada de los operadores de proyección (Ecuación 1.1) y retroproyección[4] (Ecuación 1.3) durante un número n de iteraciones junto con un cálculo de error que se lleva a cabo entre ambos operadores. A lo largo de todo el proceso, las matrices A y B permanecen invariables, pues su contenido depende de factores geométricos y no de los datos almacenados en sinogramas y rebanadas. SIRT comienza aplicando el operador de proyección a una rebanada inicial, a la que llamaremos $R0$. Dicha rebanada puede haber sido inicializada con ciertos valores, si bien lo común es que todos sus píxeles sean 0. El resultado de este paso será un sinograma $G1$ que se restará del sinograma experimental —es decir, el obtenido con el microscopio—, generándose el sinograma de error $G1'$. A partir de él, el operador de retroproyección creará la rebanada correspondiente a la primera iteración de SIRT, $R1$, que supondrá la entrada para el operador de proyección en la segunda iteración, refinándose de esto modo la reconstrucción de la rebanada. El proceso descrito se repetirá

[4]En SIRT no se aplica el filtrado.

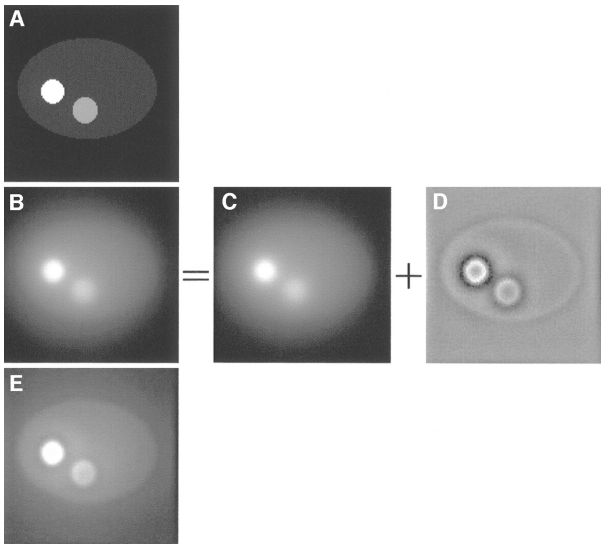

Figura 1.6: Filtrado en WBP. Se presenta la reconstrucción filtrada de la Figura 1.5. En este caso se ha filtrado después de reconstruir la imagen. Es también posible hacerlo así, pero es menos eficiente, pues normalmente los sinogramas serán más pequeños que las imágenes a las que dan lugar. A es la imagen original, B es el resultado sin filtrar, que se puede descomponer en bajas (C) y altas (D) frecuencias. E es el resultado de aplicar el filtro paso alto. Como se aprecia, el resultado no es igual al original, si bien tiene mejor calidad que la reconstrucción no filtrada. Figura tomada de [18].

hasta que se alcance la última iteración[5]. Si el número de iteraciones fuera 1, el algoritmo habría terminado con la obtención de $R1$. En esta explicación hemos supuesto que solamente se reconstruye una rebanada, pero en realidad este proceso se aplica a todas las rebanadas de forma secuencial. La Figura 1.7 detalla el funcionamiento de SIRT.

Puesto que WBP se basa exclusivamente en ejecutar una única vez la retroproyección, podemos imaginar la diferencia de tiempo entre ambos algoritmos. Teniendo en cuenta que un número común de iteraciones es 30, SIRT requerirá $2 \times 30 = 60$ veces más tiempo que WBP[6].

El Algoritmo 1.2 muestra la implementación de SIRT. Tal y como se aprecia, se encuentra dividida en los tres módulos que componen SIRT: proyección, cálculo de error y retroproyección, los cuales son ejecutados en este mismo orden. El bucle más externo (línea 1) va seleccionando una a una las rebanadas para su reconstrucción. La línea 4 marca la cantidad de iteraciones que se utilizarán. El módulo de proyección se corresponde con la transformada de Radon vista en la Ecuación 1.1, es decir, la creación de un sinograma a partir de una rebanada. En este código la matriz A no se almacena ya calculada debido a la gran cantidad de memoria que puede llegar a ocupar, por lo que sus coeficientes se hallan cuando son requeridos. La línea 11 es la aplicación de la Ecuación 1.2,

[5] A mayor número de iteraciones, mejor calidad tendrá la reconstrucción, pero también aumentará el tiempo de ejecución.

[6] El 2 se debe a la aplicación de proyección y retroproyección.

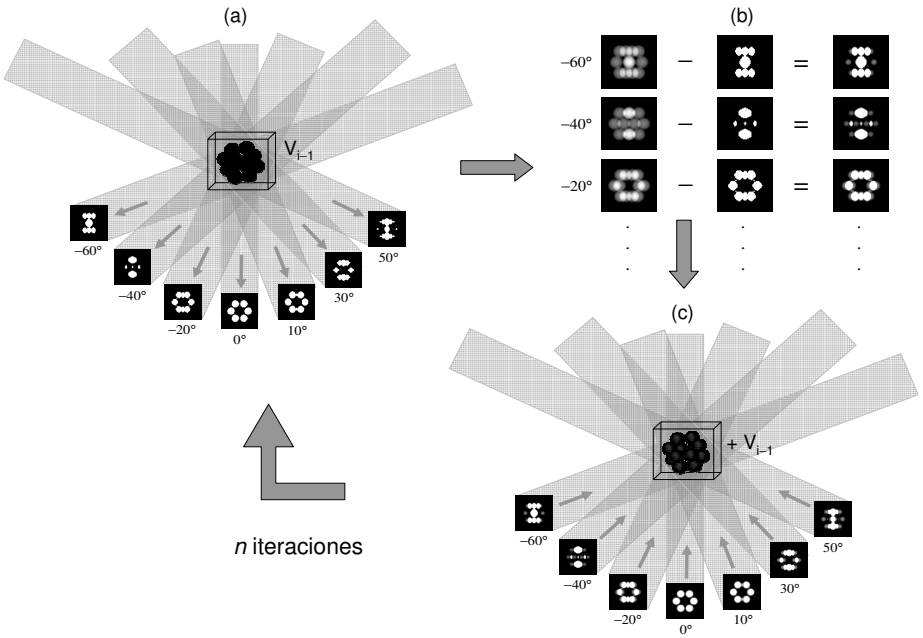

Figura 1.7: Cómo funciona SIRT. El algoritmo SIRT se divide en tres módulos distintos, que son el de proyección (a), el de cálculo de error (b) y el de retroproyección (c), que equivale a backprojection sin filtrado. Los tres se repiten durante un número n de iteraciones. Se observa que la reconstrucción en (c) es mejor que la de partida (a). Si $i \in [1, n]$ y representa la iteración actual, V_{i-1} denota el volumen reconstruido en la iteración anterior. $+V_{i-1}$ indica que el volumen obtenido en la iteración anterior es tenido en cuenta para reconstruir el volumen de la iteración actual. En general, el volumen se inicializa a 0, es decir, $V_0 = 0$. Nótese que en esta figura los volúmenes se generan a partir de imágenes de proyección y no sinogramas. Este hecho no afecta al funcionamiento de SIRT, ya que imágenes de proyección y sinogramas son dos maneras distintas de guardar una misma información (i.e. las proyecciones de las rebanadas).

y las líneas 12 y 13 calculan los pesos necesarios para llevar a cabo la interpolación. Por su parte, las líneas 14 y 15 realizan la proyección de la rebanada sobre el sinograma. El módulo para el cálculo de error simplemente resta del sinograma experimental el sinograma generado por la proyección, dando lugar a un sinograma de error[7]. Dicho sinograma es la entrada para el módulo de retroproyección, cuyo funcionamiento es idéntico al del Algoritmo 1.1.

1.3. Computación de altas prestaciones en tomografía electrónica

En este apartado generalizaremos nuestra exposición hablando de microscopía electrónica, pero todo lo que comentemos será también aplicable a la TE.

[7] $gfactor$ es sólo una matriz de pesos que únicamente dependen de factores geométricos, por lo que una vez calculados, no varían.

Algoritmo 1.2: Implementación de SIRT

```
for s in Nslices {
   /* Reconstrucción de una rebanada */
   set_to_zero(slice[s])
   for i in Niteraciones {
      set_to_zero(sino)

      /* Módulo de proyección */
      for a in Nangulos
         for y in Nfilas
            for x in Ncols {
               rf = projected_point(x,y,a)
               r = (int)rf
               weight = rf - r
               sino[a][r] = sino[a][r] + ...
                   slice[s][y][x]*(1-weight)
               sino[a][r+1] = sino[a][r+1] + ...
                   slice[s][y][x]*weight
            }

      /* Módulo para el cálculo del error */
      for a in Nangulos
         for r in Ncols {
            sino_err[a][r] = sino_exp[s][a][r] - sino[a][r]
            sino_err[a][r] = sino_err[a][r] * gfactor[a][r]
         }

      /* Módulo de retroproyección */
      for a in Nangulos
         for y in Nfilas
            for x in Ncols {
               rf = projected_point(x,y,a)
               r = (int)rf
               weight = rf - r
               slice[s][y][x] = slice[s][y][x] + ...
                   sino_err[a][r+1]*weight + ...
                   sino_err[a][r]*(1-weight)
            }
   }
}
```

La computación de altas prestaciones (HPC, *High Performance Computing*) ha sido aplicada ampliamente para satisfacer los requisitos computacionales de la microscopía electrónica y se espera que en el futuro siga teniendo igual o más importancia [31, 76]. A lo largo del tiempo se han empleado diferentes plataformas de HPC en microscopía electrónica. Durante la década de los 90, los grandes supercomputadores de centros institucionales fueron los protagonistas. Terminando dicha década comenzaron a tomar fuerza los clusters de computadoras, pues su bajo coste e increíble rendimiento hicieron posible que cada grupo de investigación tuviera uno propio. En los últimos años nuevas plataformas y tecnologías computacionales han irrumpido dentro del mundo de la HPC, entre las que destacamos las unidades de procesamiento gráfico (GPUs, *Graphics Processing Units*) y los procesadores multicore. El uso de HPC ha sido crucial en importantes investigaciones y descubrimientos dentro de la biología [42, 59, 90].

Todas las plataformas empleadas en microscopía electrónica se basan en el uso de más de un procesador, y se pueden separar en dos grandes grupos dependiendo de la organización de la memoria y el esquema de interconexión [55, 82] (Figura 1.8). Por un lado tenemos los computadores de memoria compartida, también llamados multiprocesadores simétricos (SMPs, *Symmetric Multi-Processors*), formados por un número relativamente pequeño de procesadores que comparten una memoria central. Por otro lado están los computadores de memoria distribuida, que consisten en nodos individuales compuestos por uno o más procesadores, los cuales comparten una memoria. Todos los nodos están conectados a través de una red. Una derivación de este segundo grupo son los computadores de memoria distribuida y compartida (DSM, *Distributed-Shared Memory*), que permiten que las memorias locales de los nodos estén disponibles para el resto de nodos. Aquí el acceso a memoria es no uniforme (NUMA, *Non-Uniform Memory Access*), ya que la latencia de los accesos dependerá de la localización física de la memoria.

Fundamentalmente, los multiprocesadores —y, por ende, la HPC— se usan para conseguir un paralelismo real. Así, un problema será dividido en un conjunto de tareas que se ejecutarán cada una en un procesador distinto y realizarán una porción del trabajo total minimizando la cantidad inicial de tiempo requerida. Normalmente, esas tareas necesitarán comunicarse entre ellas para intercambiar datos o por motivos de sincronización. En los computadores de memoria compartida estas comunicaciones se realizan escribiendo y leyendo de la memoria que comparten. Tanto OpenMP [23] como los Pthreads [20] se pueden usar para programar esta clase de computadores. En cambio, en plataformas de memoria distribuida se emplea el paso de mensajes, siendo MPI (*Message-Passing Interface*) la herramienta utilizada [78]. Las comunicaciones entre las diferentes tareas pueden llegar a suponer un cuello de botella importante en los sistemas de memoria distribuida, sobre todo en aquellos donde el número de procesadores conectados a la red es muy grande. Para reducir la latencia se usan redes con gran ancho de banda, tales como Ethernet, Myrinet o InfiniBand, las cuales pueden llegar a soportar transferencias de datos de hasta 10 gigabits por segundo.

En los apartados siguientes haremos un breve repaso de las diferentes plataformas de HPC que se utilizan y se han utilizado en microscopía electrónica y citaremos trabajos que hayan hecho uso de las mismas. Dichos trabajos serán específicamente de tomografía electrónica.

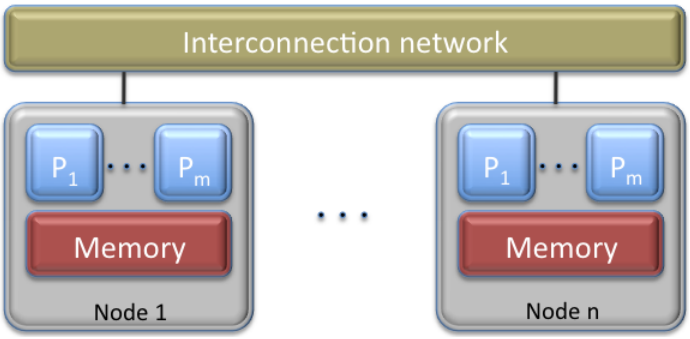

Figura 1.8: Arquitecturas multiprocesador. Cada uno de los nodos es un SMP, por lo que los procesadores del mismo comparten una memoria. Los diferentes nodos se encuentran interconectados a través de una red. Si los nodos sólo tienen acceso a su memoria, se trata de un computador de memoria distribuida. Si, por el contrario, todos los nodos pueden acceder a todas las memorias, se trata de un DSM. Un único nodo representaría a un supercomputador pequeño, mientras que el esquema global es la arquitectura implementada en los grandes supercomputadores y en los clusters, si bien en estos últimos no se suele seguir el modelo DSM. Figura reproducida de [31].

1.3.1. Supercomputadores

Los supercomputadores pueden ser plataformas de memoria compartida o distribuida, aunque en este último caso la arquitectura DSM es la más extendida (Figura 1.8). Si es de memoria compartida, el número de procesadores de los que se dispondrá no será demasiado alto —como mucho 64 o 128—. Un ejemplo es el Sun Sunfire. Si se quiere construir un gran supercomputador con muchos procesadores, es necesario que la plataforma sea de memoria distribuida y así la red de interconexión pueda soportar las demandas de comunicación de los procesadores, cuyo número puede ascender a varios miles. Ejemplos son el Cray XT5, el IBM BlueGene o el Intel Paragon. Normalmente, los centros institucionales de HPC proporcionan acceso a los supercomputadores, los cuales son utilizados por muchos científicos.

En [75] se hacía uso de supercomputadores de memoria distribuida para acelerar el proceso de reconstrucción. Debido a la latencia de las comunicaciones, en este tipo de sistemas las rebanadas suelen empaquetarse en lotes para ser enviadas a los distintos nodos. Sin embargo, en este trabajo, que seguía el paradigma maestro/esclavo, la asignación se llevaba a cabo rebanada a rebanada. Así, cuando un esclavo terminaba la reconstrucción de una, el maestro le asignaba otra.

1.3.2. Clusters

Los clusters de computadores han marcado una tendencia importante en HPC [29, 55]. Un cluster está formado por computadores independientes interconectados a través de una red, y proporciona un gran rendimiento a bajo coste, lo que ha permitido que grupos y centros individuales puedan tener uno propio dedicado exclusivamente a su investigación. Estas máquinas tienen una

alta escalabilidad (i.e. se pueden conectar fácilmente más computadores a la red) y flexibilidad (i.e. pueden usarse para resolver cualquier tipo de problema). Cuando los grupos de investigación cuentan con su propio cluster, los resultados pueden obtenerse más rápidamente que en los supercomputadores, ya que en ellos podría suceder que tuviéramos que esperar mucho tiempo en las colas de trabajo. Normalmente, podemos encontrar dos niveles de paralelismo en los clusters. El primero se refiere a la cantidad de computadores conectados a la red y el otro, al número de procesadores dentro de cada computador. Así, el cluster se puede contemplar globalmente como una arquitectura de memoria distribuida, estando configurado cada computador independiente como un SMP (Figura 1.8).

Aprovechando las características que ofrecen los clusters se han escrito paquetes de software para tomografía electrónica, entre los que destacamos Spider, Priism, Imod y UCSF tomography [91]. En [86] se presentó una estrategia híbrida para llevar a cabo la reconstrucción mediante algoritmos iterativos en la que se utilizaba MPI para las comunicaciones entre tareas corriendo en los computadores conectados a la red y OpenMP para el uso de threads en cada computador individual. En los clusters las comunicaciones pueden suponer un cuello de botella. En reconstrucción tomográfica esto ocurre cuando se emplean otras funciones base en lugar de vóxeles (e.g. blobs) que se extienden más allá de una rebanada [32, 34]. En [32, 34, 47, 67] se han implementado técnicas que tratan de ocultar la latencia de las comunicaciones en este tipo de máquinas.

1.3.3. Procesamiento vectorial

Una instrucción vectorial sigue el modelo de computación SIMD (*Single Instruction, Multiple Data*), en el que una única instrucción es capaz de realizar la misma operación sobre varios elementos de datos del mismo tipo a la misma vez. El procesamiento vectorial fue la base de muchos supercomputadores del pasado (e.g. Cray Y-MP), y hoy día ha resurgido gracias a que los procesadores comerciales actuales incorporan pequeñas unidades vectoriales. En el Capítulo 3 de esta tesis se trata en profundidad el procesamiento vectorial.

1.3.4. Computadores multicore

La arquitectura de computadores ha apostado en los últimos años por la inclusión de múltiples núcleos de computación dentro de un único procesador en vez de seguir aumentando la frecuencia de reloj. Sin embargo, la arquitectura de los núcleos no se ha visto simplificada en absoluto con respecto a los procesadores de generaciones previas. Así, características como la superescalaridad, la supersegmentación, el multithreading o el procesamiento vectorial se continúan incorporando [55]. Los procesadores multicore pertenecen a la categoría de los SMPs (Figura 1.9). Actualmente, los hay con dos, cuatro o seis núcleos, y dicha cantidad se espera que se incremente en el futuro. Los computadores multicore se pueden usar para construir clusters. En el Capítulo 4 de esta tesis se trata en profundidad el aprovechamiento de los procesadores multicore mediante programación paralela.

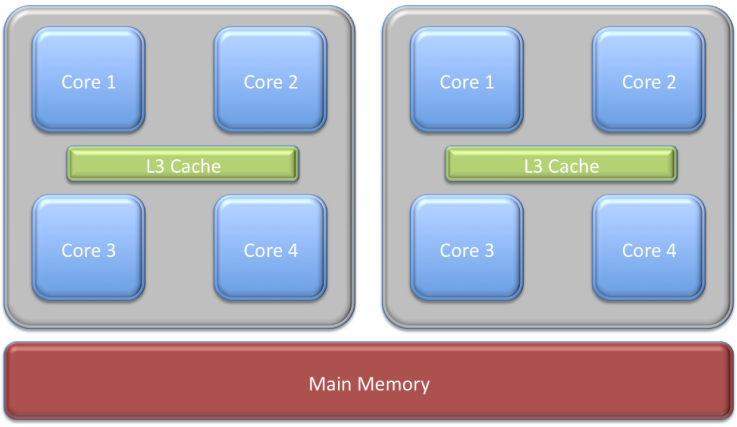

Figura 1.9: Computadores multicore. Se muestra un sistema formado por dos procesadores multicore, cada uno con cuatro núcleos. Un procesador dispone de varios niveles de caché. El nivel 3 (L3), si existe, es común a todos los núcleos de un mismo procesador, mientras que el 1 (L1) y el 2 (L2), que no aparecen en esta figura, suelen ser privados. La memoria principal del computador es compartida por todos los procesadores y núcleos. Figura reproducida de [31].

1.3.5. Unidades de procesamiento gráfico

Las unidades de procesamiento gráfico (GPUs) (Figura 1.10) han emergido recientemente como plataformas de computación de propósito general con masivas capacidades de paralelismo que proporcionan un rendimiento muy alto en computación científica a un precio incomparable [62]. Una GPU está formada por unidades de multiprocesamiento (SMs, *Streaming Multiprocessors*), cada una compuesta por cierta cantidad de núcleos, los cuales reciben el nombre de procesadores escalares (SPs, *Scalar Processors*). Todos los SPs en un SM comparten recursos, en concreto registros y una memoria local, la cual permite que los threads corran en paralelo sin necesidad de comunicarse con el exterior. A su vez, los SMs comparten una memoria global, cuyo tiempo de acceso es ostensiblemente mayor al de la memoria local. Las tarjetas más modernas de NVIDIA tienen 32 SPs y 16 SMs, por lo que en total disponen de 512 núcleos. Una novedad destacable en dichas tarjetas es la caché que se ha incluido entre la memoria global y los SMs. Para optimizar el rendimiento de las GPUs, el programador ha de tener en cuenta dos aspectos importantes: el balanceo de la carga de trabajo entre los distintos núcleos y el acceso a los datos a través de la jerarquía de memoria.

Los trabajos pioneros con GPUs en tomografía electrónica [22, 21] redujeron extraordinariamente el tiempo de ejecución requerido por los algoritmos de reconstrucción. Debido a ello, atrajeron el interés de la comunidad científica por estos dispositivos, que se han usado extensivamente desde entonces. Por ejemplo, en [83] se presentó una aproximación matricial para WBP que superaba a la implementación estándar de ese algoritmo. De forma breve, en este trabajo se convertía el problema de la reconstrucción en un producto matriz dispersa-vector implementado eficientemente en la GPU. Otro ejemplo es [88], que se centraba en los algoritmos iterativos. Se presentaba aquí un enfoque que po-

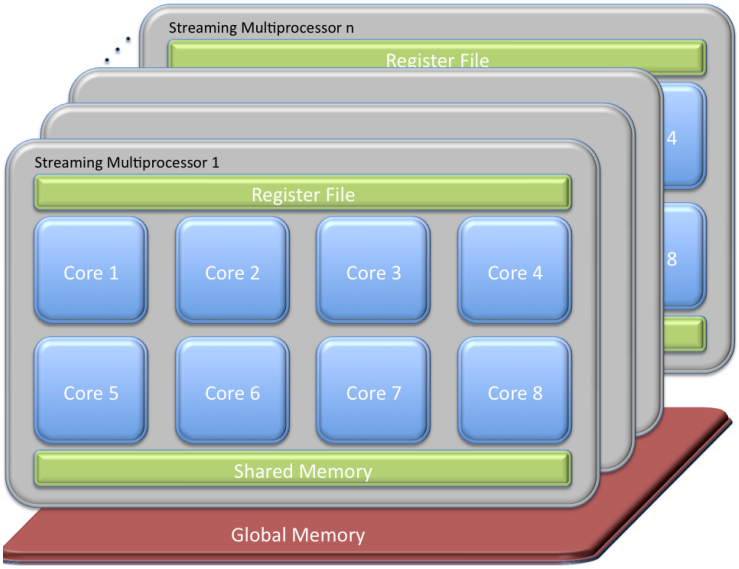

Figura 1.10: Arquitectura de una GPU. Las GPUs disponen de cientos de núcleos capaces de ejecutar conjuntamente miles de threads de computación. Los núcleos se encuentran distribuidos entre varias unidades de multiprocesamiento (*streaming multiprocessors*). Existe una memoria global compartida por todas estas unidades, llamada memoria del dispositivo. Figura reproducida de [31].

sibilitaba la obtención de grandes reconstrucciones 3D en cuestión de minutos gracias a optimizaciones de bajo nivel que minimizaban latencias y aprovechaban los canales RGBA de la GPU. También hay programas comerciales (*Xplore 3D* de la compañía FEI o *Inspect-X* de Nikon Metrology) y de uso público (*IMOD* o *Priism*) para tomografía electrónica que permiten utilizar la tarjeta gráfica si ésta admite computación de propósito general.

1.3.6. Computación distribuida

La computación distribuida se desarrolló para poder aprovechar la potencia de cálculo de recursos computacionales heterogéneos dispersados en un laboratorio, un centro, un campus o incluso geográficamente entre ciudades, países o continentes con el fin de resolver problemas complejos (Figura 1.11). Un ejemplo son los grids, donde las necesidades de computación y almacenamiento se distribuyen entre máquinas separadas geográficamente [41, 19, 87]. Entre ellos, cabe destacar el grid europeo (*European Grid Initiative*, http://www.egi.eu/) o el americano (*American Teragrid*, http://www.teragrid.org/), que son capaces de proporcionar petaflops de computación y petabytes de almacenamiento.

Un ejemplo de tomografía electrónica sobre grids es el proyecto Telescience. La implementación de WBP realizada permite la distribución de las distintas rebanadas que forman el volumen 3D entre los diferentes recursos computacionales dentro del grid para su reconstrucción [74, 63]. Por otro lado, en [33] se demostró que es posible encontrar el lote de carga de trabajo óptimo para un determinado problema y grid. Sin embargo, este trabajo también confirmaba

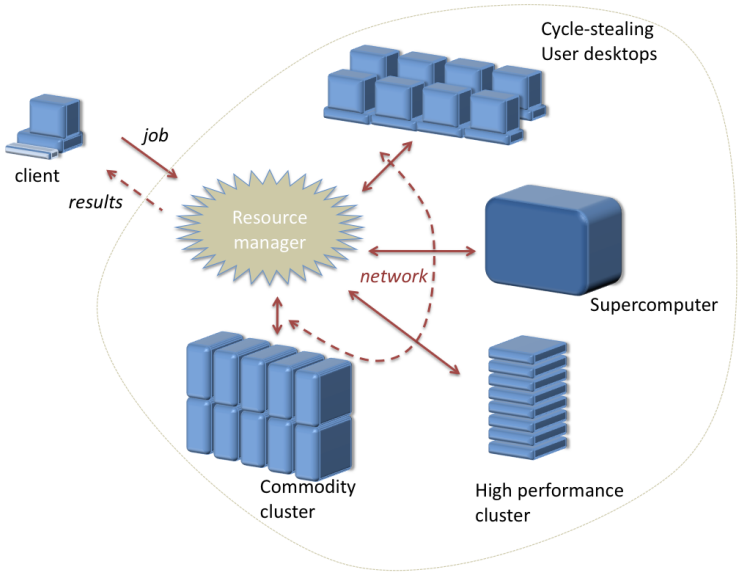

Figura 1.11: Computación distribuida. El sistema distribuido comprende un conjunto de recursos computacionales heterogéneos que pueden estar separados geográficamente. El usuario manda el trabajo al sistema, cuyas capas de gestión analizan el trabajo y la disponibilidad de recursos, y lo envían a la plataforma adecuada. El usuario recibirá los resultados cuando estos estén disponibles. Figura reproducida de [31].

que para ello era preciso tener en cuenta ciertos parámetros que de otra forma impedirían la explotación del grid. En [15] se desarrolló una aplicación que permitía establecer dichos parámetros de una manera sencilla para el usuario.

Capítulo 2

Optimizaciones básicas

Este capítulo abarca la primera fase de optimización de los algoritmos WBP y SIRT, y aúna un conjunto de optimizaciones aplicadas a nivel de núcleos individuales. Con ellas pretendemos refinar los algoritmos de reconstrucción y prepararlos para la aplicación de optimizaciones más avanzadas.

Como veremos en el Capítulo 5, nuestras optimizaciones básicas pueden contribuir a acelerar significativamente el código original. Comprenden un uso eficiente de la memoria caché, el aprovechamiento de la simetría de las imágenes, la utilización de regiones de interés, el empleo de la librería FFTW (*Fastest Fourier Transform in the West*) para la eliminación de ruido en WBP y una serie de optimizaciones a las que hemos llamado generales —e.g. desenrolle de bucles—.

2.1. Uso eficiente de la memoria caché

La Figura 2.1 muestra la forma original de acceder a sinogramas y rebanadas que se usaba tanto en la proyección como en la retroproyección de una rebanada. Se puede observar que se toma la primera proyección del sinograma y se recorren una a una todas las filas de la rebanada, repitiéndose este proceso para las demás proyecciones. Este modo de acceso es ineficiente, pues las filas de la rebanada no se mantienen en caché y cada vez que una es visitada de nuevo, hay que volver a cargarla en caché.

Para evitar el problema y aprovechar al máximo la memoria caché, tanto los sinogramas como las rebanadas se dividen en bloques durante el proceso de reconstrucción. La Figura 2.2 muestra el procedimiento implementado, que es similar a la técnica de *blocking* tan usada en computación científica [71, 85]. El objetivo de dicha técnica es minimizar el intercambio de información con la memoria principal al reutilizar los datos que hay en caché ampliamente. Así, se lleva a cabo una partición de los datos que se han de procesar en pequeños bloques que quepan en la caché y el código se reorganiza para que opere con un bloque tanto como sea posible antes de cambiar a otro. Se han empleado técnicas similares a ésta en estudios pertenecientes a los campos de procesamiento de imagen y tomografía [26, 68, 69, 79].

Tal y como se muestra en la Figura 2.2, nuestro mecanismo de *blocking* consiste en reconstruir la rebanada en pasos de R filas mientras el sinograma se va

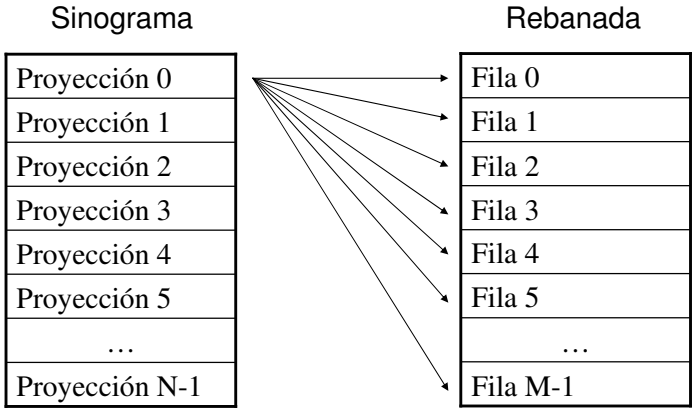

Figura 2.1: Acceso original a sinogramas y rebanadas.

leyendo en bloques de P proyecciones. Para ello fue necesario cambiar el orden de los bucles, siendo ahora el primero el que recorre las filas de la rebanada y viniendo después el que visita las proyecciones del sinograma. Mediante este procedimiento, el bloque de R filas se mantiene en caché hasta que se procesa completamente, momento en el que se producirá el cambio de bloque. Los parámetros R y P son configurables, por lo que el mecanismo se puede ajustar a cualquier tamaño de caché. De acuerdo con nuestra experiencia, R y P deberían tomar valores pequeños que sean potencias de 2 (2, 4, 8, 16), pues se adaptan bien a los tamaños de caché actuales (entre 512KB y 12MB) (consultar Apéndice B). Esta recomendación debe tomarse como una norma general, pero si se quiere configurar el mecanismo de una forma más precisa, es necesario conocer cuánta caché hay instalada y poseer ciertos conocimientos sobre el funcionamiento de una memoria caché, aparte de que el tamaño de rebanadas y sinogramas influye de modo crucial. Por ejemplo, si disponemos de poca caché y las imágenes son grandes, R y P deberían tomar valores pequeños. En cambio, si las imágenes son también pequeñas, R y P podrían ser aumentados. Tampoco es buena idea darle a estos parámetros valores para que ocupen por completo la caché, pues podría haber filas distintas —o proyecciones— que compartiesen la misma línea de caché.

Del párrafo anterior se deduce que la configuración de las variables R y P es compleja y se realiza a muy bajo nivel, por lo que no es apta para el usuario normal. Por esta razón, el mecanismo de caché también admite ser configurado mediante la selección de un bloque en KB (por ejemplo, 128 o 384) para el *blocking*. El tamaño de dicho bloque representa la cantidad de caché que se ocupará. De esta forma, el usuario solamente habrá de centrarse en elegir un bloque cuyo tamaño no supere al de la caché instalada. Internamente, el bloque se divide en dos partes —en general, desiguales—, convirtiéndose una de ellas en un número de filas y la otra, en un número de proyecciones, siendo dichos números los valores que tomarán R y P. Los tamaños de las partes se eligen teniendo en cuenta la relación existente entre los tamaño de rebanadas y sinogramas. Por ejemplo, si el tamaño de un sinograma representa el 10 % del tamaño de una rebanada, se dedicará un 10 % del bloque para albergar proyecciones, y el 90 % restante se empleará para filas. Por defecto, el mecanismo

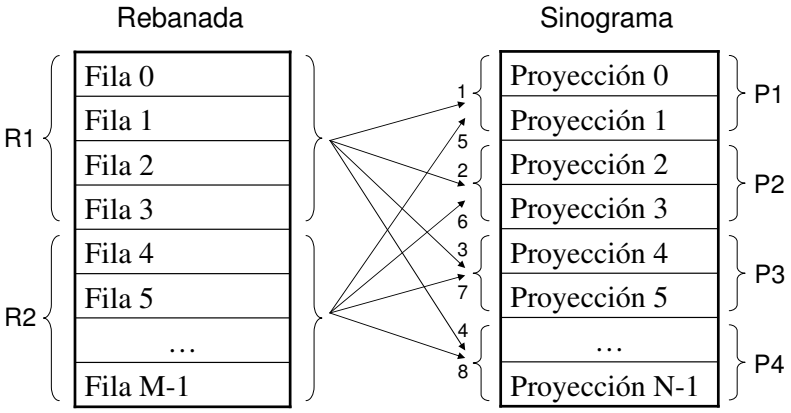

Figura 2.2: Procedimiento diseñado para reducir la tasa de fallos de caché. La rebanada se reconstruye en pasos de R filas, mientras que el sinograma es leído en bloques de P proyecciones. El orden de procesamiento viene indicado por 1, 2, 3, 4, etc. Así, R1 se procesa en primer lugar con P1, luego con P2, después con P3 y, por último, con P4. Esto mismo se repite para R2. Si bien en esta figura R es divisor del número de filas y P, del número de proyecciones, esto no tiene por qué suceder. Por ejemplo, si tuviéramos 8 filas y R fuese igual a 3, se tendrían tres bloques: dos de 3 filas y uno de 2.

detectará la cantidad de caché instalada y elegirá automáticamente un bloque óptimo para ejecutar el *blocking*. Bloques de tamaño 1/6 o 1/8 de la caché son adecuados, ya que minimizamos las posibles colisiones que puedan ocurrir y dejamos espacio para otras estructuras de datos y para otros threads o procesos que pueda haber en el sistema.

En resumen, tenemos tres formas de configurar el mecanismo de caché: a través de los parámetros R y P, seleccionando un bloque en KB para ejecutar el blocking o dejando que el mecanismo elija un bloque óptimo en función de la cantidad de caché instalada en el procesador, que es el funcionamiento por defecto.

El procedimiento desarrollado en esta tesis para aprovechar eficientemente la memoria caché del procesador permitió acelerar la ejecución de los algoritmos de reconstrucción WBP y SIRT, aparte de evitar que el speedup decayera cuando se manejaban imágenes grandes, ya que fue posible dividirlas en pequeños bloques y así reconstruirlas en caché.

2.2. La simetría de las imágenes

Esta optimización reduce el número de operaciones que requiere la reconstrucción de un volumen al aprovechar la simetría existente en la proyección y retroproyección de cada rebanada. En ambos procedimientos es necesario calcular la correspondencia entre los píxeles de la rebanada y los de la proyección 1D que en ese momento se esté procesando. Para ello, es necesario aplicar la fórmula $r = xcos\theta + ysen\theta$ a cada uno de los píxeles de la rebanada en cuestión. Esta fórmula es costosa por incluir operaciones con senos y cosenos. No obstante, si nos fijamos, por ejemplo, en el píxel D de la Figura 2.3, veremos que se

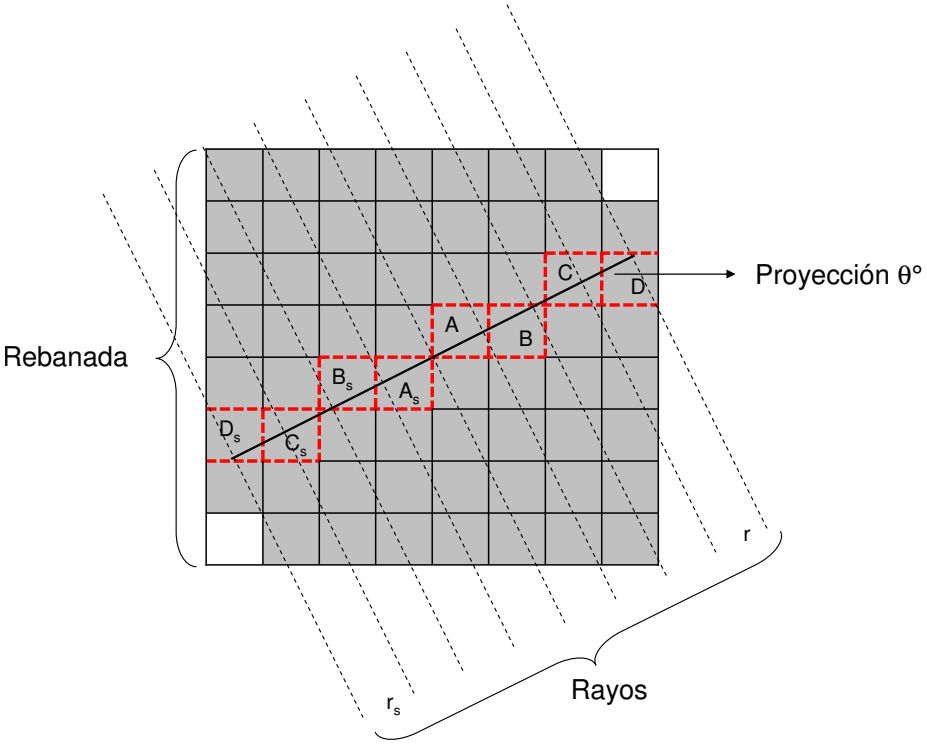

Figura 2.3: Simetría y límites de proyección/retroproyección. Los puntos simétricos llevan el subíndice *s*. Los puntos blancos no están afectados por la proyección 1D de ángulo θ.

corresponde con el píxel r de la proyección, y el simétrico de D, representado por D_s, se corresponde con el simétrico de r, simbolizado por r_s. Esta simetría se cumple para el resto de los píxeles de la rebanada, por lo que solamente será necesario aplicar la fórmula anterior a los de la mitad superior, pues las correspondencias de los de la mitad inferior se hallarán por simetría, lo cual requiere mucho menos tiempo. En la práctica, los puntos simétricos se colocan seguidos para un mejor aprovechamiento de la caché, si bien esto no aparece ilustrado en la Figura 2.3 por claridad.

Más formalmente, asumiendo que el centro de un plano es el punto $(0, 0)$, si un punto (x, y) de dicho plano se proyecta sobre un punto $r = xcos\theta + ysen\theta$ de una proyección 1D tomada a θ grados, entonces se verifica que el punto $(-x, -y)$ del plano se proyecta en un punto simétrico $r_s = -r$.

Podemos optimizar aún más el cómputo de correspondencias si tenemos en cuenta que, para un mismo ángulo, éstas son iguales para todas las rebanadas, ya que dependen exclusivamente de las coordenadas del píxel y del ángulo. Así pues, podríamos tomar cada ángulo, calcular las correspondencias de los píxeles de la mitad superior y almacenar los resultados para reutilizarlos. Sin embargo, el consumo de memoria podría llegar a ser elevado. Supongamos, por ejemplo, que tenemos 140 ángulos y rebanadas de tamaño 1024×1024. Habría que aplicar

la fórmula a 512K[1] elementos, por lo que tendríamos 512K resultados por ángulo, ocupando cada uno 4 bytes. En total, necesitaríamos 280MB. Si bien en este caso el consumo no es excesivo, la cifra sí es muy superior a la caché instalada en los procesadores actuales. Al tratarse de datos usados con mucha frecuencia, estarían constantemente entrando y saliendo de la caché, lo cual degradaría el rendimiento de la reconstrucción.

Ya que precalcular todas las correspondencias puede acarrearnos más problemas que beneficios, podemos hallar solamente un subconjunto y luego, mediante operaciones sencillas, derivar el resto. Para cada ángulo y cada fila de la mitad superior de una rebanada vamos a calcular una correspondencia inicial, es decir, aplicamos $r = ysen\theta$. Estas correspondencias iniciales las guardaremos en una matriz de nombre ci, a la que se accede $ci[ángulo][y]$. El siguiente paso que damos es precalcular los cosenos de todos los ángulos, que quedan almacenados en un array de nombre cos. Para obtener el coseno de un ángulo basta escribir $cos[ángulo]$. Por tanto, la expresión que ahora nos proporciona las correspondencias es $r = ci[ángulo][y] + xcos[ángulo]$. La multiplicación no es preciso llevarla a cabo, ya que al ir recorriendo las filas elemento a elemento, es suficiente con obtener $r = ci[ángulo][y]$ y luego ir sumando $cos[ángulo]$ según avanzamos en x. Esta es una técnica similar a la "optimización de la variable de inducción en bucles" empleada por las estrategias de optimización de los compiladores [30, 85]. En comparación con el ejemplo del párrafo anterior, este nuevo enfoque necesita aproximadamente 280KB de memoria.

Para que la simetría pueda llevarse a cabo es preciso que la altura de las rebanadas sea un número par. Si bien esta situación es la normal, no siempre se dará. Cuando la altura sea impar, nosotros añadiremos una fila adicional.

2.3. Definición de regiones de interés

Una región de interés (ROI, *Region Of Interest*) es el área de la imagen que va a ser reconstruida y, por tanto, nosotros sólo estamos interesados en dicha área. En cada rebanada existe un conjunto de puntos que no se halla afectado por los procesos de proyección y retroproyección. Dada una proyección a θ grados, ese conjunto es el mismo para todas las rebanadas que componen el volumen, pero varía al cambiar el ángulo de inclinación de la proyección. Esto implica que para cada ángulo tendremos un conjunto de puntos distinto, aunque aplicable a todas las rebanadas. No visitar esos puntos durante el proceso de reconstrucción resulta beneficioso, pues ahorramos tiempo de computación.

La Figura 2.3 muestra un ejemplo donde los puntos blancos de la rebanada no forman parte del ROI, ya que no están afectados por los rayos correspondientes a la proyección 1D representada. Es por ello que no se tendrán en cuenta durante la reconstrucción.

Nosotros precalculamos para cada ángulo los límites de la reconstrucción, y únicamente reconstruimos lo que hay dentro de dichos límites. Sin este precálculo, sería preciso comprobar cada píxel de las rebanadas para saber si hemos de incluirlo o no. Aprovechándonos de la simetría vista en el apartado anterior, solamente sería preciso realizar el precálculo teniendo en cuenta las filas de la mitad superior de las rebanadas, pues los límites de los de la mitad inferior son simétricos (Figura 2.3). Para cada ángulo y cada fila de la mitad superior,

[1] Aquí 1K equivale a 1024.

vamos a tener una estructura que nos dirá cuál es la coordenada x inicial y cuál la final. Además, a dicha estructura podemos añadirle la correspondencia inicial para ese ángulo y esa fila. La estructura es la siguiente:

```
struct ROI
{
   int x_inicial;  /* Coordenada x inicial */
   int x_final;    /* Coordenada y final */
   float r;        /* Correspondencia para x_inicial */
};
```

De esta forma, ya sea en la proyección o retroproyección, dado un ángulo y una fila sabremos en qué coordenada x empezar y en cuál terminar, aparte de cuál es la correspondencia inicial que debemos tomar.

2.4. La librería FFTW

Tal y como se vio en el Capítulo 1, las imágenes reconstruidas mediante el algoritmo backprojection se encuentran emborronadas debido a la aplicación implícita de un filtrado paso bajo. Este efecto puede compensarse realizando la operación inversa, esto es, un filtrado paso alto. De esta manera, los perfiles y contornos de las imágenes quedarán resaltados, lo cual facilitará el análisis posterior del volumen 3D generado.

La transformada de Fourier es necesaria para llevar a cabo la operación de filtrado y nosotros decidimos valernos de la librería FFTW (*Fastest Fourier Transform in the West*) [46] para ello, ya que es muy rápida al estar altamente optimizada y usar procesamiento vectorial internamente. Una de las principales ventajas del empleo de la FFTW reside en que ésta se basa en el precálculo de los coeficientes involucrados en el cómputo de la FFT. Así se obtiene el mayor beneficio posible cuando se aplica de forma repetida con los mismos parámetros de tamaño. Este es justo el caso del filtrado paso alto de las proyecciones en tomografía. Por ejemplo, para una reconstrucción a partir de 1024 sinogramas de 1024 píxeles y 140 ángulos se precisarían $1024 \times 140 = 143360$ transformadas de Fourier 1D de tamaño 1024.

2.5. Optimizaciones generales

Una amplia gama de optimizaciones se aplica como complemento a las anteriores con el objetivo de acelerar aún más el código de los algoritmos de reconstrucción [30, 49, 50, 53, 70, 71, 72, 85]. Entre ellas destacamos (1) un incremento del paralelismo a nivel de instrucción, (2) precálculos de datos frecuentemente utilizados a lo largo de todo el proceso de reconstrucción —e.g. cosenos de ángulos—, (3) el empleo de funciones *inline*, (4) sustitución de divisiones y multiplicaciones con potencias de dos por desplazamientos, (5) sustitución de algunas divisiones en bucles por multiplicaciones, (6) desenrolle de bucles y (7) eliminación de condicionales.

Debido a la generalidad de todas estas optimizaciones, podemos considerarlas como un conjunto de buenas prácticas de programación que puede ser administrado a cualquier tipo de programa, sea cual sea su naturaleza.

Capítulo 3

Procesamiento vectorial con instrucciones SSE

Es intención de este capítulo hacer una introducción al procesamiento vectorial utilizando las instrucciones SSE (*Streaming SIMD Extensions*) presentes en cualquiera de los procesadores que hoy día comercializan tanto Intel como AMD. Existen cuatro posibles maneras de emplearlas en nuestros programas, que se explicarán acompañadas de ejemplos para facilitar su comprensión, viendo las virtudes y desventajas de cada una de ellas. Por tanto, no se pretende entrar en complejos y profundos detalles de arquitectura, sino exponer de forma práctica y sencilla qué posibilidades vectoriales posee cualquiera de los ordenadores con los que usualmente trabajamos y cómo podemos aprovecharlas, siempre desde el punto de vista del programador.

Con todo este conocimiento previo se desea proporcionar la base necesaria para entender la vectorización de los algoritmos WBP y SIRT, que será presentada dentro de este capítulo también.

3.1. Introducción

El procesamiento vectorial consiste en realizar una misma operación sobre varios elementos de datos de igual tipo a la vez, y no es un concepto nuevo en la arquitectura de computadores [38]. De hecho, durante los años 80 y 90, los procesadores vectoriales disfrutaron de gran fama y fueron la base de muchos supercomputadores, los cuales tuvieron un uso eminentemente científico [55]. Sin embargo, debido al progresivo incremento en rendimiento de los procesadores de propósito general, empezaron a perder la importancia que hasta entonces habían tenido. Un ejemplo claro de aquellos supercomputadores fue el Cray Y-MP modelo D, cuya versión más avanzada contaba con 8 procesadores vectoriales.

Hoy día el procesamiento vectorial ha resurgido, pues los mismos computadores de propósito general que desbancaron a aquellos supercomputadores incorporan pequeñas unidades vectoriales y un amplio conjunto de instrucciones para poder aprovecharlas, usualmente conocidas como SIMD (*Single Instruction, Multiple Data*) [1, 43, 50, 54]. Ejemplos son MMX y SSE, introducidas por Intel e incorporadas luego por AMD, 3DNow! de AMD o AltiVec de Motorola. Además, también las videoconsolas y las tarjetas gráficas actuales se apoyan

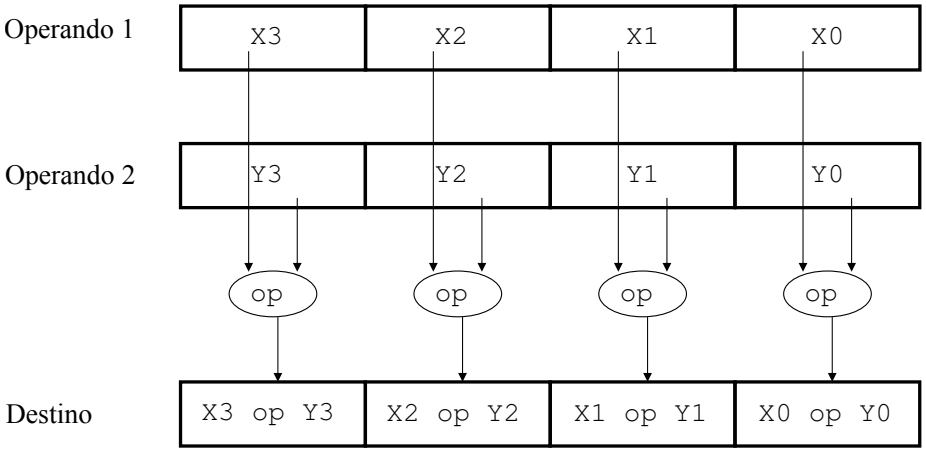

Figura 3.1: Funcionamiento típico de una instrucción SIMD. La operación ejecutada está indicada con 'op', y podría ser una suma, una división, una raíz cuadrada, etc. Un procesador que no dispusiera de unidades vectoriales estaría obligado a realizar las cuatro operaciones de forma secuencial, consumiendo un tiempo mayor. Aunque se muestren operandos de cuatro elementos, nada impide que sean más grandes. Por otro lado, los elementos pueden ser números enteros o flotantes, y su precisión puede variar. Por ejemplo, suponiendo operandos de 256 bits, podríamos trabajar con 4 enteros de 64 bits, 8 de 32, 16 de 16 o 32 de 8. Tanto la longitud de los operandos como la precisión de los elementos son características propias de cada arquitectura, y los detalles de la misma los decidirá el fabricante del procesador.

enormemente en el procesamiento vectorial para poder alcanzar altas cotas de rendimiento [62].

3.2. Las instrucciones SSE

Hasta el Pentium, él incluido, todos los procesadores de Intel habían sido puramente SISD (*Single Instruction, Single Data*), aunque esto cambió con la llegada del Pentium MMX en 1997, que fue el primero en dotar a la arquitectura x86 de instrucciones SIMD (*Single Instruction, Multiple Data*). Una instrucción SIMD, en contraposición a una SISD, es capaz de realizar de una sola vez la misma operación sobre varios elementos de datos empaquetados, tal y como se aprecia en la Figura 3.1.

El problema de las instrucciones MMX radicaba en que solamente trabajaban con enteros y no con flotantes, aparte de tener que compartir con la unidad de punto flotante (FPU) x87 los registros para los cálculos, lo cual provocaba que la mezcla de instrucciones MMX con instrucciones de la FPU fuera algo realmente no deseable. No tardó AMD en aprovechar este handicap, pues en 1998 introdujo su K6-2 y lo dotó con una mejora a la tecnología MMX: las instrucciones 3DNow!, las cuales ya sí operaban con números flotantes. Mientras tanto, los laboratorios de Intel no habían estado parados, y en 1999 la compañía dio salida a su Pentium III, que aumentaba el repertorio de instrucciones

de la arquitectura x86 con las extensiones SSE. Originariamente diseñadas para manejar flotantes en simple precisión (32 bits), fueron modernizadas para que también pudieran tratar enteros y flotantes en doble precisión (64 bits), ampliación ésta que recibió el nombre de SSE2 y, si bien ha habido otras (SSE3, SSSE3 y SSE4), podemos afirmar que ella ha sido la más relevante. Para evitar confusiones, a partir de ahora utilizaremos el término SSE para referirnos a las extensiones SSE en general, mientras que con SSE1 nos referiremos a la primera versión aparecida con el Pentium III.

En un principio las instrucciones SSE fueron creadas para acelerar aplicaciones multimedia de procesamiento de vídeo, imagen y sonido, pero rápidamente su uso se extendió para acelerar cualquier tipo de aplicación, y en especial las científicas [2, 3, 13, 14, 24, 25, 28, 48].

3.2.1. Interfaz con el programador

Las instrucciones SSE suponen la conexión entre el programador y las distintas unidades vectoriales del procesador. El repertorio de instrucciones SSE es bastante amplio y puede ser consultado en [58]. Las SSE operan sobre vectores —registros o zonas de memoria— que tienen una longitud de 128 bits. En total, el programador dispone de 8 registros SSE, ampliados a 16 en los procesadores con extensiones de 64 bits y solamente usables cuando funcionan en este modo, que reciben el nombre de XMMn, donde n es un número de 0 a 15. En cada registro se pueden empaquetar:

- ✓ Cuatro flotantes en simple precisión (4 x 32 bits)
- ✓ Dos flotantes en doble precisión (2 x 64 bits)
- ✓ Dieciséis enteros de 8 bits (16 x 8 bits)
- ✓ Ocho enteros de 16 bits (8 x 16 bits)
- ✓ Cuatro enteros de 32 bits (4 x 32 bits)
- ✓ Dos enteros de 64 bits (2 x 64 bits)

3.2.2. Consideraciones antes de programar

Antes de empezar a (re)escribir un algoritmo empleando las instrucciones SSE, hay una serie de cuestiones que es importante tratar, o de lo contrario es probable que se enfoque la vectorización de un modo poco apropiado, lo que nos llevaría a un desaprovechamiento de los recursos del procesador y a una pérdida de nuestro tiempo. En los apartados siguientes examinamos estas cuestiones brevemente.

Cuándo y qué vectorizar

En nuestro contexto, la optimización es el proceso a través del cual el código de una aplicación se escribe de manera distinta o especial para conseguir que dicha aplicación se ejecute más rápido en el mismo computador. Este proceso es siempre a medida, pues dependerá de la propia aplicación y del procesador, es decir, no todas las estrategias que utilicemos tienen por qué ser válidas con otras

aplicaciones o con otros procesadores. Por ejemplo, un buen aprovechamiento de la caché puede ser muy beneficioso en un programa de tratamiento de imágenes que haga un uso intensivo de matrices, mientras que en otro que maneje grandes bases de datos almacenadas en medios magnéticos sería más conveniente mejorar el acceso a disco.

La vectorización es una estrategia más de optimización que los programadores tienen a su disposición y, por tanto, no siempre será adecuado emplearla. Tendremos que examinar cuidadosamente nuestro programa y decidir si puede verse recompensado. En general, si trabaja sobre grandes cantidades de datos realizando repetidamente las mismas operaciones, entonces es seguro que va a merecer la pena. Según [57], ejemplos de buenos candidatos para ser escritos con instrucciones SIMD son:

- Filtros y algoritmos de compresión de voz.
- Rutinas de visualización de vídeo.
- Rutinas de renderizado.
- Gráficos y audio en 3D.
- Algoritmos de procesamiento de imagen y vídeo.
- Aplicaciones científicas.

Por otro lado, carece de sentido intentar optimizar cada parte de nuestro programa, pues esto desembocaría en una inútil pérdida de tiempo. Pensemos que no todas consumen la misma cantidad de tiempo, y debería ser tarea del programador encontrar cuál o cuáles son las más lentas (*hotspots* o puntos calientes), centrando sus esfuerzos en mejorarlas sólo a ellas. Para esto pueden venir muy bien herramientas de análisis del rendimiento como VTune.

Cómo saber si el procesador soporta las extensiones SSE

Una vez analizada la aplicación y decidido que la vectorización es factible, deberíamos verificar que el computador con el que se va a trabajar soporta las extensiones SSE. Para ello disponemos de la instrucción CPUID, que nos devuelve gran cantidad de información acerca del procesador. Sin embargo, este método es algo complejo, ya que tendríamos que programar una rutina en ensamblador. Afortunadamente, existen otras maneras que podemos utilizar para conocer rápidamente la información deseada. En Windows tenemos el programa CPU-Z, que se puede descargar desde *www.cpuid.com*, y en Linux podemos escribir en la consola *cat /proc/cpuinfo*.

Sería muy raro que el procesador no soportase las SSE1, pues desde la aparición del Pentium III todos los nuevos modelos de la familia x86 las han incorporado. Distinto es el caso de las SSE2, SSE3, SSSE3 y SSE4, que estarán o no presentes dependiendo de cuán moderno sea. Además, los procesadores de AMD no implementan las SSSE3 ni las SSE4 de Intel.

El compilador

Hemos estado hablando continuamente de instrucciones SSE y quizá se tenga la sensación de que la única vía para emplearlas en un programa es haciendo

uso del lenguaje ensamblador. En el pasado, cuando se querían utilizar características especiales de los procesadores y optimizar al máximo, ésta era la única manera. Sin embargo, ahora existen compiladores optimizadores que son capaces de hacer esto automáticamente y que también ofrecen herramientas a los programadores para facilitarles esta tarea en el caso de que el grado de optimización alcanzado por el compilador no sea satisfactorio. Ejemplos son el tan usado GCC en plataformas Linux o el ICC de Intel, del que nos ocuparemos más adelantes en este mismo capítulo.

Alineamiento de la memoria

Cuando se programa con las instrucciones SSE es sumamente recomendable alinear las regiones de memoria a 16 bytes, es decir, la dirección base o inicial de la zona de memoria habrá de ser múltiplo de 16. A tal efecto, el compilador ICC de Intel proporciona las funciones *_mm_malloc*() y *_mm_free*() para la gestión de la memoria dinámica y la palabra clave *___declspec* para la estática.

Aunque la memoria no esté alineada es aún posible seguir utilizando las SSE, pero tendremos que usar instrucciones de acceso a memoria no alineada, lo que degradará en gran medida el rendimiento.

Disposición de los datos en memoria (*data layout*)

Es importante que los datos con los que se va a trabajar estén todos agrupados en posiciones contiguas de memoria, y si éste no fuera el caso, deberían realizarse las transformaciones necesarias. Con esto conseguiremos aprovechar la caché del procesador y, al mismo tiempo, que las instrucciones SSE trabajen a pleno rendimiento. Por ejemplo, si quisiéramos sumar las columnas de una matriz, el proceso de carga de los elementos en registros SSE sería muy lento, pues al no estar en posiciones contiguas, tendríamos que ir cargando uno por uno en lugar de leer varios a la vez, aparte de que cada acceso a memoria supondría un fallo de caché. Calcular la traspuesta de la matriz solucionaría el problema.

Selección del tipo de dato

Especialmente cuidadoso hay que ser a la hora de elegir el tipo de dato con el que operar, pues hacerlo sin analizar qué precisión es la necesaria podría provocar que obtuviéramos menos aceleración de la posible. Si nos fijamos, por ejemplo, en los flotantes que las instrucciones SSE pueden manejar, veremos que en un registro XMM podemos guardar dos en doble precisión o cuatro en simple precisión. Si no necesitamos la doble precisión, sería bastante más sensato utilizar la simple, pues estaríamos manejando los números de cuatro en cuatro en lugar de hacerlo de dos en dos, con la consiguiente ganancia en velocidad.

Compatibilidad entre la FPU x87 y las instrucciones SSE en punto flotante

Supongamos que tenemos un antiguo programa que realiza operaciones con flotantes y hemos visto la posibilidad de reescribirlo con instrucciones SSE. Lo hacemos y estamos muy satisfechos porque hemos conseguido un buen factor de aceleración. Sin embargo, al comparar los resultados de los cálculos, vemos que son algo distintos a los que obteníamos antes, así que pensamos que nos estamos

equivocando en algo. Ocurre que, si bien tanto la FPU como las SSE trabajan con flotantes de 32 y 64 bits, éstas lo hacen en modo nativo, mientras que aquélla trabaja internamente con 80 bits, así que ha de hacer una conversión al principio de los cálculos y al final. En general, las diferencias serán muy pequeñas, y si no es así, entonces sí es probable que hayamos cometido algún error.

3.3. El compilador de C/C++ de Intel

Hemos ofrecido hasta aquí una visión general de las instrucciones SSE, explicando qué son y mostrando su filosofía de trabajo, pero nada sabemos sobre cómo utilizarlas en nuestras aplicaciones. Un camino podría ser escoger un ensamblador que las reconozca —el NASM sería una opción acertada— y escribir nuestro programa. No existiría un problema insalvable si no las soportase, pues podríamos codificar directamente las instrucciones en código hexadecimal indicando que son datos dentro de la sección de texto (código). Sin embargo, la realidad nos dice que el campo de acción de estos métodos sería reducidísimo —por ejemplo, rutinas para sistemas empotrados donde los requisitos de memoria suelen ser muy restrictivos— y no serían recomendables ni para aplicaciones exageradamente simples.

Como ya hemos apuntado, hoy día existen compiladores optimizadores capaces de generar código que aproveche las características especiales que pueda tener un procesador y, además, le proporciona al programador herramientas para que sea él quien escriba el código a mano si es que no ha quedado satisfecho. Uno de los que citábamos era el ICC (*Intel C++ Compiler*), y lo hemos elegido en lugar del GCC porque, en general, se obtienen mejores resultados en cuanto a velocidad, aparte de que la documentación y el soporte proporcionados son amplios y de gran calidad. Hay que resaltar que el ICC es un producto creado por Intel para sacar el máximo partido a sus procesadores y no a los de la competencia, léase AMD, pero esto no significa que no podamos usarlo con ellos.

La Figura 3.2 muestra las diferentes técnicas de programación que el ICC pone a nuestra disposición para que podamos escribir aplicaciones con las extensiones SSE, las cuales iremos describiendo acompañadas de ejemplos. Tal y como se indica, la más sencilla de usar es la vectorización automática, aunque es a la vez la que menos control otorga al programador. En el otro extremo se encuentra el ensamblador en línea, la más compleja de todas y la que más inconvenientes tiene, pero, a cambio, proporciona un control total al programador sobre el código. Téngase en cuenta que más control no tiene por qué corresponderse siempre con mayor velocidad.

3.3.1. Vectorización automática

Con este método, escribimos nuestros programas de manera normal, pero le pasamos el argumento $-x\{K|N|P|T\}$ al compilador, dejando así la vectorización del código en sus manos. K es para Pentium III, N para Pentium 4, P para Core y T para Core 2, y resulta necesario especificar el tipo de procesador porque no todos soportan las mismas extensiones SSE. Por ejemplo, el Pentium III soporta la primera versión de las SSE, pero ninguna de las ampliaciones posteriores. Además, distintos procesadores requieren de distintas estrategias de optimización, aunque algunas puedan ser comunes a todos. En realidad, con $-x$ estamos

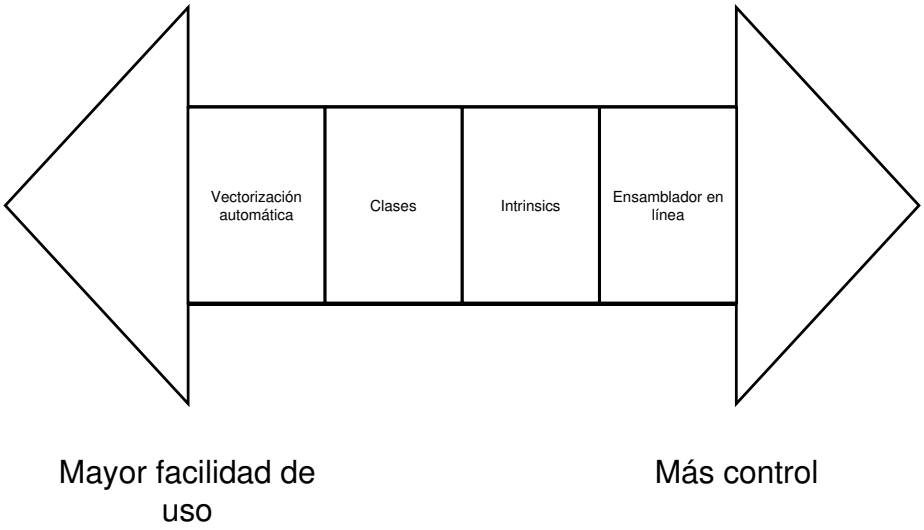

Figura 3.2: Métodos para escribir un programa con instrucciones SSE.

Algoritmo 3.1: Suma de dos vectores secuencial

```
1 void sumavect_normal(float *c, float *a, float *b)
2 {
3     int i;
4
5     for (i=0; i < ARRAY_SZ; i++)
6         c[i] = a[i] + b[i];
7 }
```

obligando al ICC a que genere código especializado para correr en el procesador que le indiquemos, siendo una consecuencia de esto la vectorización. Cuando el compilador consigue vectorizar un bucle avisa por pantalla escribiendo *LOOP WAS VECTORIZED*.

Se puede colegir, pues, que la relación esfuerzo/beneficio de este procedimiento es de un diez absoluto, pues el programador no ha de dedicar ni un minuto de su tiempo a la vectorización, si bien existe la posibilidad de ayudar al compilador en esta tarea introduciendo una serie de directivas en el código que lo puedan orientar mejor [50]. No obstante, hemos de decir que esta sencillez es, a la vez, lo mejor y lo peor de este método, pues en muchas ocasiones el análisis del código que lleve a cabo el compilador no será tan bueno como el que pueda hacer un programador, aunque, la verdad sea dicha, funciona muy bien con bucles no demasiado complejos. Además, si nuestro objetivo es aprender a optimizar algoritmos haciendo uso de las instrucciones SSE, no es ésta una buena forma, ya que poco sabremos acerca de lo que ocurre.

Ya que nada especial ha de hacerse con el código, aprovechamos ahora para presentar en el Algoritmo 3.1 la función en C que iremos reescribiendo con clases, *intrinsics* y ensamblador en línea. Su cometido es sumar los vectores a y b, y guardar el resultado en c. Vamos trabajar con flotantes de simple precisión.

Algoritmo 3.2: Suma de dos vectores empleando clases

```
1 #include <fvec.h>
2
3 // Asumimos que los arrays están alineados a 16
4 // y que ARRAY_SZ es una constante múltiplo de 4
5
6 void sumavect_clases(float *c, float *a, float *b)
7 {
8     int i;
9     F32vec4 *a4 = (F32vec4 *)a;
10    F32vec4 *b4 = (F32vec4 *)b;
11    F32vec4 *c4 = (F32vec4 *)c;
12
13    for (i=0; i < ARRAY_SZ / 4; i++)
14        c4[i] = a4[i] + b4[i];
15 }
```

3.3.2. Clases

El compilador de Intel viene con un conjunto de clases que podemos usar para vectorizar nuestros programas. Dependiendo de nuestras necesidades, hemos de incluir alguno de los siguientes ficheros de cabecera:

- *ivec.h*, que proporciona soporte para las instrucciones MMX
- *fvec.h*, que proporciona soporte para las instrucciones SSE1
- *dvec.h*, que proporciona soporte para las instrucciones SSE2

Esta técnica de programación adolece de un importante problema, pues no tenemos acceso a todas las instrucciones y, por tanto, no siempre podremos expresar aquello que queremos. Además, no están implementadas ni SSE3 ni SSSE3 ni SSE4.

La función que suma dos vectores escrita con clases se puede ver en el Algoritmo 3.2. Como vemos, el código se parece mucho al original, lo cual lo hará fácil de seguir, depurar y mantener, siendo ésta una de las grandes ventajas de las clases. Si nos fijamos en las líneas 9, 10 y 11 vemos que aparece el tipo de dato *F32vec4*, que sería el equivalente a un registro XMM en el que se guardan cuatro flotantes en simple precisión. Al realizar esas asignaciones, cada posición de $a4$, $b4$ y $c4$ hará referencia a cuatro posiciones de a, b y c, respectivamente. Así, acceder a $b4[i]$ sería hacerlo al mismo tiempo a $b[i]$, $b[i+1]$, $b[i+2]$ y $b[i+3]$. La sentencia $c4[i] = a4[i] + b4[i]$ en la línea 14 sigue la filosofía de operación que explicamos en la Figura 3.1, y como se están procesando cuatro flotantes por iteración en vez de uno, el tope del bucle ha de dividirse entre cuatro.

3.3.3. Intrinsics

En este método, cada instrucción SSE es representada por una función o *intrinsic*. Los programas se escriben utilizando esas funciones y después el com-

Algoritmo 3.3: Suma de dos vectores empleando intrinsics

```
1 #include <xmmintrin.h>
2
3 // Asumimos que los arrays están alineados a 16
4 // y que ARRAY_SZ es una constante múltiplo de 4
5
6 void sumavect_intrinsics(float *c, float *a, float *b)
7 {
8     int i;
9     __m128 a4, b4, c4;
10
11    for (i=0; i < ARRAY_SZ; i+=4)
12    {
13        a4 = _mm_load_ps(a+i);
14        b4 = _mm_load_ps(b+i);
15        c4 = _mm_add_ps(a4, b4);
16        _mm_store_ps(c+i, c4);
17    }
18 }
```

pilador las reemplazará por las instrucciones correspondientes. Para usar las *intrinsics* es necesario incluir alguno de los siguientes ficheros de cabecera:

- *mmintrin.h*, que proporciona soporte para las instrucciones MMX
- *xmmintrin.h*, que proporciona soporte para las instrucciones SSE1
- *emmintrin.h*, que proporciona soporte para las instrucciones SSE2
- *pmmintrin.h*, que proporciona soporte para las instrucciones SSE3
- *tmmintrin.h*, que proporciona soporte para las instrucciones SSSE3
- *smmintrin.h* y *nmmintrin.h*, que proporcionan soporte para las instrucciones SSE4

La gran ventaja de las *intrinsics* es que tenemos noción y un control más o menos amplio de las instrucciones que estamos empleando sin necesidad de bajar al ensamblador, pero la asignación de registros o la emisión de instrucciones no son todavía competencia nuestra. Por otro lado, todo el repertorio de instrucciones SSE está disponible para el programador.

La función que suma dos vectores usando intrinsics podemos verla en el Algoritmo 3.3. Si nos fijamos en el cuerpo del bucle, el código comienza a alejarse del original. Hay que especificar de una en una las operaciones que se han de llevar a cabo, lo cual nos va recordando un poco al lenguaje ensamblador, aunque aún sigue siendo muy claro y legible. En la línea 9 aparecen declaradas tres variables como *__m128*. Este tipo de dato sería como el *F32vec4* que utilizábamos en las clases, y, como dijimos, es el equivalente a un registro XMM que guarda cuatro flotantes en simple precisión. Puesto que en cada iteración se procesan cuatro flotantes, el contador del bucle se ha de incrementar de cuatro en cuatro.

Algoritmo 3.4: Suma de dos vectores empleando ensamblador en línea

```
1  // Asumimos que los arrays están alineados a 16
2  // y que ARRAY_SZ es una constante múltiplo de 4
3
4  void sumavect_asm(float *c, float *a, float *b)
5  {
6      __asm {
7          mov     eax, a
8          mov     ebx, b
9          mov     ecx, c
10         mov     edx, (ARRAY_SZ*4)-16
11
12     suma:
13         movaps  xmm0, xmmword ptr [eax+edx]
14         addps   xmm0, xmmword ptr [ebx+edx]
15         movaps  xmmword ptr [ecx+edx], xmm0
16         sub     edx, 16
17         jns     suma
18 }
```

3.3.4. Ensamblador en línea

Escribimos directamente en ensamblador aquellas partes del código que nos interesan y luego las insertamos dentro del código C. La desventaja es que nuestro código perderá portabilidad y será más difícil de escribir y mantener. A pesar de todo, no debemos olvidar lo útil que puede llegar a ser este método de programación en algunas situaciones debido al control total sobre las instrucciones que nos ofrece. Por ejemplo, imaginemos que hemos programado un algoritmo utilizando los tres métodos anteriores, pero el rendimiento obtenido todavía no es el deseado. Podríamos inspeccionar el código ensamblador generado por el compilador y ver si hay posibilidades de mejora cambiando o reordenando las instrucciones para conseguir una emisión más eficiente. También podría ocurrir que los requisitos de memoria fueran muy restrictivos e impusieran un espacio limitado para el código, por lo que probablemente no nos quedaría otro remedio que escribirlo directamente en ensamblador para aprovechar cada byte al máximo. Desde luego, tareas como las anteriores distan mucho de ser sencillas y se necesitarían conocimientos profundos de la arquitectura del procesador. No son caminos especialmente adecuados para el neófito, pues podría ocurrir que nuestro código ensamblador fuese más lento u ocupase más que el del compilador.

La función que suma dos vectores escrita en ensamblador aparece en el Algoritmo 3.4. Las líneas 7, 8 y 9 cargan las direcciones de los arrays a, b y c en los registros eax, ebx y ecx, respectivamente. El registro edx se utiliza a la vez como índice para acceder a los arrays y como contador del bucle. En la línea 13 cargamos en xmm0 cuatro flotantes de a, los sumamos con cuatro de b en la línea 14 y, en la 15, guardamos el resultado en c, repitiéndose esto hasta que todos los números hayan sido procesados. Nótese en la línea 16 que el índice es disminuido de 16 en 16. La razón es que el lenguaje ensamblador, al menos el x86, no interpreta los contenidos de las posiciones de memoria, cosa que sí hace C. Así, él no ve 4 flotantes, sino los 16 bytes que ocupan.

3.3.5. Consideraciones finales

Qué método de los cuatro anteriores se debe usar es la pregunta lógica que toca hacerse en este momento, y ciertamente no hay una respuesta concreta, es decir, no podemos coronar a ninguno como el mejor. Dejando a un lado los gustos y las preferencias de cada uno, depende de la situación. ¿Hemos conseguido el rendimiento buscado con la vectorización automática? Si la respuesta es afirmativa, sería poco inteligente intentar reprogramar el algoritmo con alguno de los otros métodos, ya que podríamos dedicar nuestro tiempo a tareas mucho más útiles. Por el contrario, si la respuesta es negativa, entonces deberíamos probar con las clases o las *intrinsics* y ver qué ocurre. Programar directamente en ensamblador tendría que ser nuestro último recurso, pues podríamos sorprendernos al descubrir que nuestro código artesano es igual de rápido o, lo que es peor, más lento que el generado por el compilador, el cual puede conocer y aplicar muchas estrategias de optimización que al programador sin demasiada experiencia se le pueden escapar.

Elijamos el método que elijamos, lo que sí habremos de hacer en primera instancia es analizar el algoritmo que tenemos intención de vectorizar y comprobar si es un buen candidato. En el caso de que no lo sea, tendremos que pensar en cómo podemos modificarlo, y, a partir de ahí, iremos realizando pruebas hasta alcanzar una aceleración que nos satisfaga.

3.4. Vectorización de WBP y SIRT

En este apartado vamos a explicar detalladamente la implementación vectorial de los algoritmos WBP y SIRT desarrollada en esta tesis. De entre los cuatro métodos de programación que acabamos de ver, nosotros decidimos seleccionar las *intrinsics* por ser el que mayor flexibilidad y control nos ofrecía sin necesidad de bajar al ensamblador. No obstante, para hacer más sencilla la comprensión de los algoritmos, aquí presentaremos un pseudocódigo más descriptivo que el código real.

Recordemos que el núcleo de backprojection es el proceso de retroproyección y que el de SIRT está constituido además por el proceso inverso —el de proyección— más un cálculo de error. Por tanto, casi todo lo que digamos acerca de WBP será también aplicable a SIRT.

3.4.1. Vectorización de WBP

Tal y como se mostró en el Capítulo 1, el algoritmo WBP está compuesto por dos módulos que se ejecutan de manera secuencial:

$$\boxed{WBP = [Filtrado] + Retroproyección}$$

El filtrado es opcional y por eso aparece entre corchetes, si bien su uso es altamente recomendable para atenuar el emborronamiento de las imágenes. Para dicho filtrado, nosotros hemos empleado la librería FFTW (*Fastest Fourier Transform in the West*) [46], la cual está muy optimizada, ya que, entre otras razones, también se sirve del procesamiento vectorial.

El proceso de retroproyección es el módulo más importante y, en resumen, distribuye los niveles de densidad de las imágenes de proyección sobre las distintas rebanadas que componen el espécimen. Para llevar a cabo su vectorización

barajamos dos estrategias diferentes, aunque finalmente descartamos una de ellas debido a la pobre aceleración conseguida. A ésta no le dedicaremos mucho espacio en esta tesis, pero creemos conveniente comentarla sucintamente. A partir de ahora, para que las explicaciones sean más sencillas, reduciremos el problema de la reconstrucción a una única proyección unidimensional, si bien todo lo que digamos será aplicable a las demás proyecciones del sinograma.

Mediante el enfoque fallido se pretendían procesar cuatro píxeles consecutivos de una misma rebanada al mismo tiempo. Recordemos que los píxeles de las imágenes, ya sean sinogramas o rebanadas, son números flotantes en simple precisión de 32 bits y, por tanto, podemos guardar cuatro en un registro vectorial XMM. Durante la reconstrucción es necesario calcular con qué posiciones de la proyección unidimensional se corresponden los píxeles de la rebanada. El inconveniente de este cálculo es que no puede ser vectorizado y debe realizarse de manera secuencial. Cuando la proyección 1D es tomada a $\pm 0°$ o $\pm 90°$, cuatro píxeles consecutivos de la rebanada se van a corresponder con cuatro píxeles consecutivos de la proyección 1D, pero esto no se cumplirá para el resto de ángulos (Figura 3.3). Por consiguiente, al orientar la resolución del problema tal y como hemos apuntado, se hacía inevitable hallar las correspondencias de los cuatro píxeles de la rebanada por separado, lo que provocaba que el factor de aceleración obtenido fuera inexistente con respecto a la versión secuencial. Es por ello que resolvimos abandonar este planteamiento.

La estrategia con la que finalmente decidimos quedarnos persigue procesar cuatro rebanadas a la misma vez en lugar de intentar acelerar el procesamiento de una sola, que como ya hemos visto no es una buena solución. En teoría, esto significa que si en un tiempo t la versión secuencial reconstruye una rebanada, la versión vectorizada reconstruirá cuatro. En el Capítulo 1 explicábamos que para averiguar con qué píxel r de una proyección 1D tomada a un ángulo θ se corresponde cierto píxel (x, y) de una rebanada teníamos que aplicar la siguiente fórmula:

$$r = xcos\theta + ysen\theta$$

Así pues, únicamente es preciso conocer el ángulo al que se tomó la proyección 1D y las coordenadas del píxel de la rebanada para encontrar la correspondencia. Esto supone que, dado un ángulo, todos los píxeles de todas las rebanadas situados en las mismas coordenadas van a estar asociados a una misma posición r, si bien el contenido de dicha posición será distinto en cada caso al pertenecer a diferentes partes del espécimen. Por ejemplo, si el píxel $(13, 34)$ de cierta rebanada se corresponde con la posición 28 de una proyección 1D que se tomó a 30°, los píxeles $(13, 34)$ de las otras rebanadas del espécimen se corresponderán también con la posición 28 de sus respectivas proyecciones 1D tomadas a 30°. Recuérdese que de cada rebanada se toma siempre el mismo número de proyecciones 1D, cada una de ellas a un ángulo distinto, pero invariable. Por ejemplo, si de un espécimen dividido en dieciséis rebanadas se toman de la primera cuatro proyecciones 1D a 0, 10, 20 y 30 grados, de las quince restantes se tomarán también cuatro proyecciones 1D a esos mismos ángulos.

El hecho que acabamos de comentar puede ser aprovechado para suprimir el problema que antes se nos hubo presentado, si bien aparece uno nuevo: ahora podemos calcular un índice r que valga para varios píxeles, pero dichos píxeles estarán todos en rebanadas distintas. Por otro lado, aunque el valor del índice r es compartido, no lo es el contenido de la posición de memoria, ya que nos

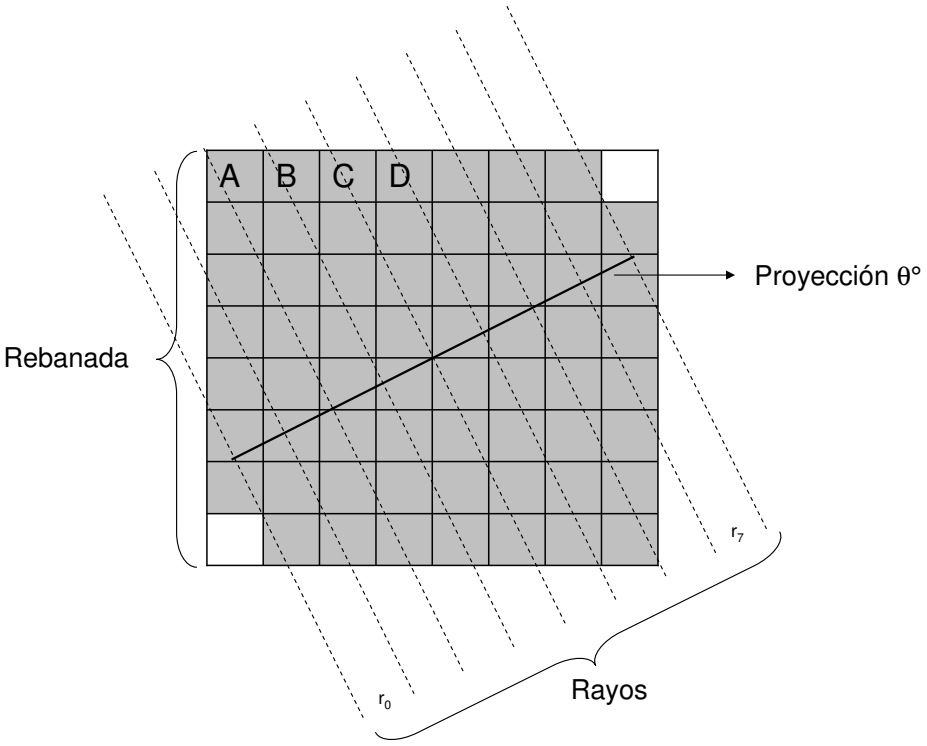

Figura 3.3: Dada una proyección 1D tomada a un ángulo θ cualquiera distinto de $\pm 0°$ y $\pm 90°$, los píxeles A, B, C y D de la rebanada no se corresponden directamente con cuatro píxeles consecutivos de la proyección, por lo que habrá que calcular las posiciones de una en una.

estamos refiriendo a proyecciones diferentes. Por tanto, ni los píxeles de las rebanadas en los que estamos interesados están consecutivos en memoria ni tampoco los píxeles de índice r asociados a ellos, lo cual es un impedimento, pues las instrucciones SSE precisan de datos contiguos en memoria.

Para resolver este nuevo problema, y puesto que en un registro vectorial XMM es posible guardar hasta cuatro flotantes en simple precisión, podríamos agrupar las rebanadas de cuatro en cuatro de forma que los píxeles que comparten coordenada estuvieran consecutivos. También agruparíamos los sinogramas de cuatro en cuatro disponiendo las proyecciones 1D tomadas a un mismo ángulo para que índices iguales ocupasen posiciones contiguas de memoria.

Este esquema de distribución de datos puede observarse en la Figura 3.4. A la izquierda, las rebanadas están representadas por matrices bidimensionales 3×3, y las proyecciones 1D por vectores de 3 elementos. Todos los píxeles $(1, 0)$ de las rebanadas se corresponden con el índice $r = 0$, apreciándose que ni los píxeles de las rebanadas en los que estamos interesados ni los píxeles de índice 0 asociados a ellos están colocados consecutivamente. Con la transformación de la derecha, agrupamos los píxeles para conseguir que sí lo estén. Ahora, con $x = 1$ y $r = 0$ queremos decir que accedemos a las posiciones de la rebanada SSE 4, 5, 6 y 7, y a las posiciones 0, 1, 2 y 3 de la proyección SSE, respectivamente. Un sinograma SSE será aquel cuyas filas sean todas proyecciones SSE.

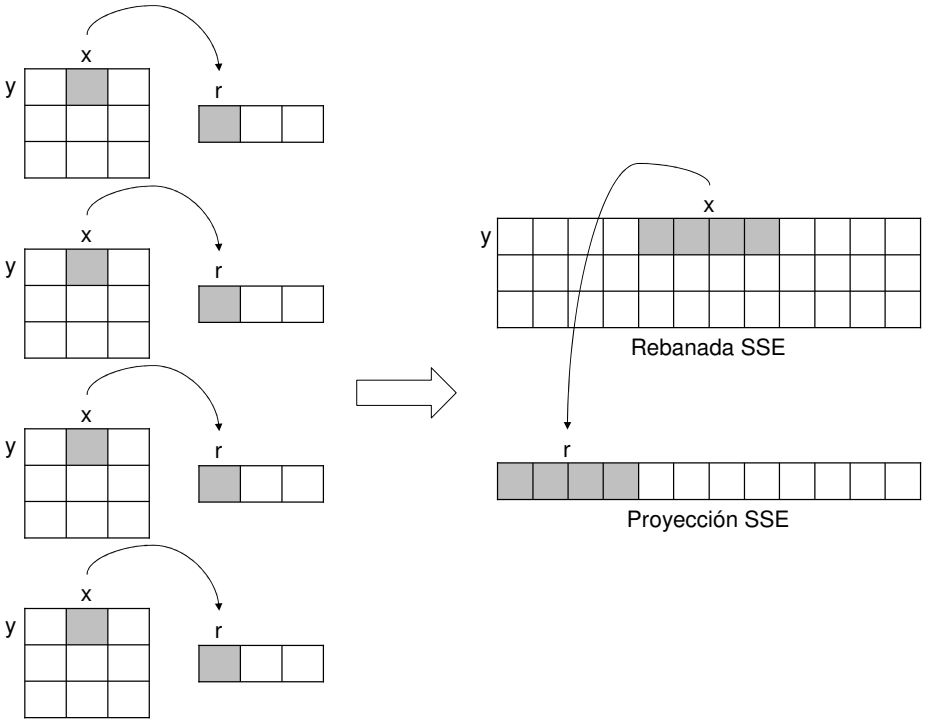

Figura 3.4: Distribución de datos para hacer posible la reconstrucción de cuatro rebanadas a la vez mediante procesamiento vectorial.

Al estar los píxeles distribuidos de esta forma, es posible leerlos, operar con ellos y almacenarlos de cuatro en cuatro mediante instrucciones SSE. Una vez reconstruidas las cuatro rebanadas habría que convertir la rebanada SSE en las cuatro rebanadas originales. Todas estas transformaciones consumen un tiempo adicional, pero éste es despreciable en comparación con el usado por backprojection. En el caso de que el número de rebanadas no sea un múltiplo de cuatro, se empleará un relleno o *padding* hasta alcanzar el múltiplo de cuatro más cercano. Por ejemplo, si se tuvieran 301 rebanadas, se añadirían 3 más para llegar a 304. En ningún caso esto implica un incremento del tiempo de ejecución, pues no hay diferencia entre procesar una rebanada o cuatro debido a que se reconstruyen a la vez gracias al procesamiento vectorial.

La implementación vectorial de WBP se muestra en el Algoritmo 3.5 e ilustra todo lo que acabamos de explicar, si bien el filtrado opcional se ha omitido por simplicidad. La línea 4 representa la distribución de datos de sinogramas ($Pack$), mientras que la 18 descompone una rebanada SSE en las cuatro que dieron lugar a ella ($Unpack$). Nótese que no se hace referencia a la creación de la rebanada SSE, cosa que sí es exhibida en la Figura 3.4. Téngase en cuenta que todas las rebanadas se inicializan a cero antes de ser reconstruidas (línea 7), por lo que realizar el agrupamiento es menos eficiente que reservar una región de memoria que albergue a la rebanada SSE y poner dicha región a cero. Por otro lado, en la línea 14 aparece el operador $SSE()$, el cual indica que las operaciones afectadas por él serán efectuadas mediante procesamiento vectorial

Algoritmo 3.5: Implementación vectorial de WBP

```
1  for s in Nslices by 4 {
2     /* Creamos un sinograma SSE agrupando los sinogramas
3        s, s+1, s+2 y s+3 */
4     sinoSSE = Pack(sino[s], ... ,sino[s+3])
5
6     /* Reconstrucción de 4 rebanadas a la vez */
7     set_to_zero(sliceSSE)
8     for a in Nangulos
9        for y in Nfilas
10          for x in Ncols {
11             rf = projected_point(x, y, a)
12             r = (int)rf
13             weight = rf - r
14             sliceSSE[y][x*4] = SSE(sliceSSE[y][x] + ...
                   sinoSSE[a][r+1]*weight + ...
                   sinoSSE[a][r]*(1-weight))
15          }
16
17    /* Extraemos las cuatro rebanadas reconstruidas */
18    slice[s], ... ,slice[s+3]=Unpack(sliceSSE)
19 }
```

y los índices de acceso más internos serán multiplicados por cuatro. Así, si $r = 4$, no se accede a la posición 4, sino a la 20. Siguiendo con este ejemplo, la variable *weight* multiplicará a $sinoSSE[a][20]$, $sinoSSE[a][21]$, $sinoSSE[a][22]$ y $sinoSSE[a][23]$ en un único paso. Ni el índice a ni el y se verán afectados por el operador $SSE()$, ya que no son los más internos.

3.4.2. Vectorización de SIRT

Tal y como vimos en el Capítulo 1, el algoritmo SIRT está compuesto por tres módulos que se ejecutan de manera secuencial durante una determinada cantidad de iteraciones:

$$\boxed{SIRT = (Proyección + Cálculo\ de\ error + Retroproyección) \times n}$$

Una vez completada la vectorización de backprojection, la de SIRT no es excesivamente compleja. Recordemos que la proyección es el proceso inverso a la retroproyección, es decir, a partir de rebanadas obtenemos sinogramas. De nuevo, dispondremos las imágenes tal y como indicamos en la Figura 3.4 y operaremos según la filosofía de trabajo de las instrucciones SSE (Figura 3.1).

Para poder vectorizar el cálculo de error tendremos agrupados los sinogramas experimentales también de cuatro en cuatro. Cada sinograma SSE producido por el módulo de proyección será restado mediante procesamiento vectorial a su correspondiente sinograma SSE experimental. Los píxeles de los sinogramas se leerán, restarán y almacenarán en grupos de cuatro. Puesto que cada grupo contiene píxeles de cuatro sinogramas distintos, estarán siendo restados en paralelo.

La retroproyección es exactamente igual que en backprojection, salvo que ahora se toman como entrada para generar las rebanadas del volumen 3D los sinogramas devueltos por el módulo de cálculo de error. Todo este proceso se volverá a repetir durante n iteraciones, pero, a diferencia del algoritmo secuencial, el tiempo de ejecución teórico de SIRT será cuatro veces menor debido al procesamiento vectorial.

La implementación vectorial de SIRT se muestra en el Algoritmo 3.6. Como se aprecia, se encuentra dividida en los tres módulos que acabamos de comentar: proyección, cálculo de error y retroproyección. Todos ellos han sido vectorizados, lo cual es revelado por el empleo del operador $SSE()$, cuyo funcionamiento es calcado al descrito en WBP. Son destacables la línea 4 en la que se lleva a cabo la distribución de datos de los sinogramas experimentales ($Pack$) y la 40, donde se deshace el agrupamiento de las cuatro rebanadas que formaban la rebanada SSE ($Unpack$). En realidad, al igual que sucedía en backprojection, el agrupamiento no se efectúa debido a que es menos eficiente que reservar una zona de memoria para la rebanada SSE y luego ir inicializando dicha zona a cero con el fin de reutilizarla (línea 7).

Algoritmo 3.6: Implementación vectorial de SIRT

```
1  for s in Nslices by 4 {
2     /* Creamos un sinograma SSE agrupando los sinogramas
3        experimentales s, s+1, s+2 y s+3 */
4     sinoSSE_exp = Pack(sino_exp[s], ... ,sino_exp[s+3])
5
6     /* Reconstrucción de 4 rebanadas a la vez */
7     set_to_zero(sliceSSE)
8     for i in Niteraciones {
9        set_to_zero(sinoSSE)
10       /* Módulo de proyección */
11       for a in Nangulos
12          for y in Nfilas
13             for x in Ncols {
14                rf = projected_point(x,y,a)
15                r = (int)rf
16                weight = rf - r
17                sinoSSE[a][r*4] = SSE(sinoSSE[a][r] + ...
                      sliceSSE[y][x]*(1-weight))
18                sinoSSE[a][(r+1)*4] = SSE(sinoSSE[a][r+1] ...
                      + sliceSSE[y][x]*weight)
19             }
20
21       /* Módulo para el cálculo del error */
22       for a in Nangulos
23          for r in Ncols {
24             sinoSSE_err[a][r*4] = SSE(sinoSSE_exp[a][r] - ...
                   sinoSSE[a][r])
25             sinoSSE_err[a][r*4] = SSE(sinoSSE_err[a][r] * ...
                   gfactor[a][r])
26          }
27
28       /* Módulo de retroproyección */
29       for a in Nangulos
30          for y in Nfilas
31             for x in Ncols {
32                rf = projected_point(x,y,a)
33                r = (int)rf
34                weight = rf - r
35                sliceSSE[y][x*4] = SSE(sliceSSE[y][x] + ...
                      sinoSSE_err[a][r+1]*weight + ...
                      sinoSSE_err[a][r]*(1-weight))
36             }
37    }
38
39    /* Extraemos las cuatro rebanadas reconstruidas */
40    slice[s], ... , slice[s+3]=Unpack(sliceSSE)
41 }
```

Capítulo 4

Multithreading y optimización de la E/S

Mediante el multithreading como técnica de programación es posible la paralelización de aplicaciones que corren en procesadores multicore. En este capítulo examinaremos cómo valernos de dicha técnica para acelerar la reconstrucción tomográfica. Gracias a ella podremos dividir una reconstrucción en lotes de trabajo más pequeños que serán procesados simultáneamente por los diferentes núcleos existentes. Además, tal y como veremos, el multithreading también nos ayudará a incrementar la eficiencia del sistema de E/S.

El capítulo comienza exponiendo de forma breve qué son y cómo nacieron los procesadores multicore. Luego se hace una pequeña introducción al multithreading. Por último, se analizan los diferentes enfoques con threads que se han diseñado en esta tesis para atacar el problema de la reconstrucción tomográfica.

4.1. El nacimiento de los procesadores multicore

Un procesador multicore es aquel que posee más de un núcleo, que es la parte encargada de la ejecución de las instrucciones máquina. Son considerados multiprocesadores homogéneos y debido a la existencia de varios núcleos, podrán ejecutar paralelamente las instrucciones de varios programas distintos, lo cual debe hacerse en los procesadores con un único núcleo de forma alternada en el tiempo. No fue hasta el año 2005 cuando Intel y AMD sacaron al mercado los primeros procesadores multicore, si bien su nacimiento se debe más a un problema tecnológico al que no se encontró una solución factible que a un avance buscado [12].

Corrían los primeros años del 2000 e Intel seguía apostando por aumentar la frecuencia de reloj de sus Pentium 4, mientras que AMD había decidido invertir su tiempo en el desarrollo de un microprocesador con más de un núcleo de ejecución, lo cual fue anunciado por la compañía en el año 2003. Así las cosas, en el 2004 Intel se topó con un obstáculo difícil de superar, pues si bien podía continuar incrementando la velocidad de sus procesadores, no era capaz de elaborar un sistema de refrigeración que pudiera disipar todo el calor generado por estos que fuera barato y silencioso, aparte de que el consumo de energía de estos procesadores era cada vez mayor [16]. En ese momento Intel se percató de

que AMD le llevaba una ventaja considerable y de que tenía que pensar en un nuevo diseño viable si no quería verse enteramente desplazado por su rival en la carrera de los microprocesadores.

En una lucha contra el tiempo y en tan solo nueve meses, en mayo del 2005 Intel dio salida a su Pentium D, cuyo diseño consistía en dos procesadores Pentium 4 unidos en un mismo chip. Por su lado, y en esa misma fecha, AMD ya tenía disponible su Athlon 64 X2, el cual poseía dos núcleos de ejecución. Sin embargo, existía una gran diferencia entre ambos procesadores: el de Intel fue creado rápida y precipitadamente aprovechando lo que se tenía en aquel preciso momento, mientras que el de AMD había sido concebido desde el principio como un procesador multicore nativo. Esto significaba que AMD se encontraba bastante por delante de Intel, si bien éste había conseguido no quedarse completamente atrás.

De esta forma nacieron los procesadores multicore, tan populares y utilizados actualmente, cuya arquitectura no se ha visto en absoluto simplificada con respecto a la de los procesadores de generaciones previas. Así pues, potentes características como la superescalaridad, la supersegmentación o el procesamiento vectorial todavía permanecen [55]. Los diseños de aquellos primeros procesadores multicore se han ido mejorando y actualizando, y hoy día existen procesadores con cuatro núcleos —Intel Core 2 Quad o AMD Phenom FX— e incluso con seis —Intel Core i7 Extreme Edition 980X o AMD Phenom II X6—. El número de núcleos continuará creciendo con el paso del tiempo, pero la inclusión de este tipo de procesadores en el mercado entraña también un riesgo que antaño no existía [12]. Antes la velocidad de los procesadores se incrementaba con cada nueva generación, lo que directamente beneficiaba a los programas ya existentes. Dicha velocidad ahora también se aumenta, pero a un ritmo muy inferior, lo que implica que si se quiere sacar el máximo partido a los procesadores multicore es necesario aplicar técnicas especiales de programación y reescribir las antiguas aplicaciones para que sean capaces de aprovechar la potencia de cálculo de estas modernas máquinas, con el consiguiente gasto económico que ello conlleva.

4.2. Multithreading

Debemos distinguir entre el multithreading en hardware y en software. El primero de los casos alude a la habilidad que tiene un procesador para comenzar a ejecutar instrucciones de otro thread cuando el que está activo no puede continuar —por ejemplo, porque necesita un dato que no está en caché—. En este contexto, un thread puede ser un proceso o un thread propiamente dicho. En el segundo de los casos, se trata de una técnica de programación que permite la división de una aplicación en threads, y es a esta acepción a la que nos referimos en este capítulo.

Un thread —o hilo— puede contemplarse como un conjunto de instrucciones que realizan una determinada acción [51]. Por ejemplo, un thread podría ser una función en C que ordena un *array* de enteros. Todos los programas tienen al menos un thread, que será el único que haya si el programador no crea ninguno más. En C, el thread principal que siempre existe es aquel que ejecuta la función `main()`. Una vez nacen, los threads dependen del proceso desde el cual se crearon, pero su ejecución es autónoma. Esto significa que pueden correr de

forma paralela a dicho proceso, pero si éste termina, todos los threads activos subordinados a él acabarán automáticamente también.

Los threads son útiles tanto en aplicaciones que realizan distintas tareas susceptibles de ser ejecutadas concurrentemente como en otras donde una misma tarea pueda ser particionada en lotes de trabajo más pequeños. Un ejemplo de las primeras podría ser un servidor de descarga de ficheros que crea un thread por cada petición que recibe para poder seguir a la escucha, mientras que un ejemplo de las segundas sería el problema de la reconstrucción tomográfica que abordamos en esta tesis. Dependiendo de si están ejecutándose en un sistema multiprocesador o monoprocesador, los threads correrán en paralelismo real o ficticio.

Una aplicación que se ejecuta como un conjunto de threads deberá ser capaz de manejar la simultaneidad de acciones que se suceden, por lo que hay varias cuestiones importantes que habremos de tener en cuenta [51]:

- La **sincronización**, que es el mecanismo a través del cual varios threads coordinan sus actividades. Por ejemplo, si un thread necesita la salida de otro para continuar, deberá esperar hasta que este último acabe.

- Los **recursos no compartidos**. Tanto en un computador como en un programa existen recursos cuyo uso no puede compartirse por más de un thread al mismo tiempo. Imaginemos, por ejemplo, un conjunto de threads que han de modificar una estructura de datos global para ir actualizando el resultado de sus acciones. Si en un momento dado más de un thread quiere alterar dicha estructura, solamente uno de ellos podrá hacerlo, teniendo los demás que esperar.

- El **balanceo de carga** [40], que hace referencia a la distribución de trabajo entre los threads, la cual debería ser por regla general equitativa, si bien existen situaciones en las que es más adecuado que unos threads tengan más carga que otros. Por ejemplo, supongamos un procesador multicore con uno de sus núcleos sobrecargado. Si nuestra aplicación crea varios threads y uno de ellos cae en dicho núcleo, no debería tener la misma carga de trabajo que los demás.

- La **escalabilidad**, que apunta a la capacidad de una aplicación para aprovechar todos los núcleos de un procesador multicore. Es decir, si una aplicación se diseñó para exprimir al máximo cuatro núcleos y se ejecuta en una máquina con ocho, ¿los aprovechará igual de bien?

Otro aspecto importante en el multithreading es la optimización de código previa a la paralelización de una aplicación [51]. La búsqueda de más rendimiento siempre debe comenzar sustentada por un código secuencial óptimo. La primera y más importante optimización que se ha de acometer es a nivel algorítmico, es decir, realizar cambios en el diseño del algoritmo para que éste produzca los mismos resultados en menos tiempo, tarea que no suele ser ni rápida ni sencilla. Sin embargo, cuando una aplicación es dividida en threads se nos presenta una buena ocasión para volver a visitar el código y probar, por ejemplo, nuevas estructuras de datos y técnicas que incrementen su rapidez. Por otro lado, un código secuencial siempre será más sencillo de escribir, depurar y analizar. Como comentamos en el Capítulo 3, un punto de inicio ideal en la optimización lo constituyen los puntos calientes de nuestra aplicación, pues es

mejorándolos a ellos cuando la reducción de tiempo se hará más evidente. No obstante, los puntos fríos (*cold spots*) también suponen una buena oportunidad en el multithreading. Dichos puntos son las partes de un programa donde los recursos no están siendo aprovechados tanto como podrían estarlo. Ejemplos son las zonas de sincronización o la E/S. El tiempo que conllevan esas operaciones podría ser utilizado para solapar otras pertenecientes a otros hilos.

En el mundo Unix, la opción más extendida para programar con threads es la librería *Pthreads* (POSIX Threads) [20], que es la que nosotros hemos usado en nuestro programa. Una alternativa muy interesante es *OpenMP* [23]. Se trata de un conjunto de directivas que se insertan en el código fuente para guiar al compilador, dejando en sus manos la divisón en threads de la aplicación. Si bien facilita enormemente la tarea del programador, no es tan flexible como los *Pthreads*.

4.3. Multithreading con WBP y SIRT

Nosotros hemos utilizado el multithreading para acelerar la reconstrucción tomográfica de volúmenes 3D en procesadores multicore desde tres ópticas diferentes. Dichas ópticas tienen en común que las distintas rebanadas que componen un volumen se distribuyen entre tantos threads de procesamiento como cores —o núcleos— haya disponibles. Para reconstruir las rebanadas, cada thread ejecuta un algoritmo —WBP o SIRT— en el que se aúnan las optimizaciones básicas del Capítulo 2 y la vectorización del Capítulo 3. Con la inclusión de los threads perseguimos reducir el tiempo requerido por una reconstrucción en un factor igual al número de núcleos del procesador. Las diferencias entre las tres ópticas atañen a cómo se realiza la distribución de la carga de trabajo —rebanadas— y al modo de llevar a cabo la E/S a disco. La Figura 4.1 muestra un esquema de las mismas.

4.3.1. Asignación estática de carga

En este primer diseño (Figuras 4.1A y 4.2) la carga de trabajo se distribuye entre los threads de forma estática, esto es, antes de que comiencen su ejecución reciben todos la misma cantidad de rebanadas para reconstruir y dicha cantidad no cambia a lo largo de toda la duración del programa. Así, si tenemos 64 rebanadas y 4 threads, cada uno recibirá un total de 16. Debido a nuestro enfoque vectorial, es preciso que el número de rebanadas a reconstruir sea un múltiplo de cuatro, lo cual no siempre sucede. Como ya comentamos en el Capítulo 3, en estos casos se aplicará un *padding*. También podría ocurrir que la cantidad de rebanadas no fuera divisible entre el número de threads. En situaciones de este tipo, se hará el reparto más justo posible. Por ejemplo, si un volumen está compuesto por 64 rebanadas y se lanzan 5 threads, todos reconstruirán 12, salvo uno que procesará 16. Téngase en cuenta que la unidad mínima de carga de trabajo está compuesta por cuatro rebanadas debido al empleo de las instrucciones SSE.

Hasta ahora nos hemos centrado en el procesamiento de las imágenes y no hemos dicho nada acerca de su lectura y escritura en disco. Es esta una fase importante de la reconstrucción, pues para volúmenes grandes la E/S puede suponer una fracción significativa del tiempo total, por lo que es recomendable

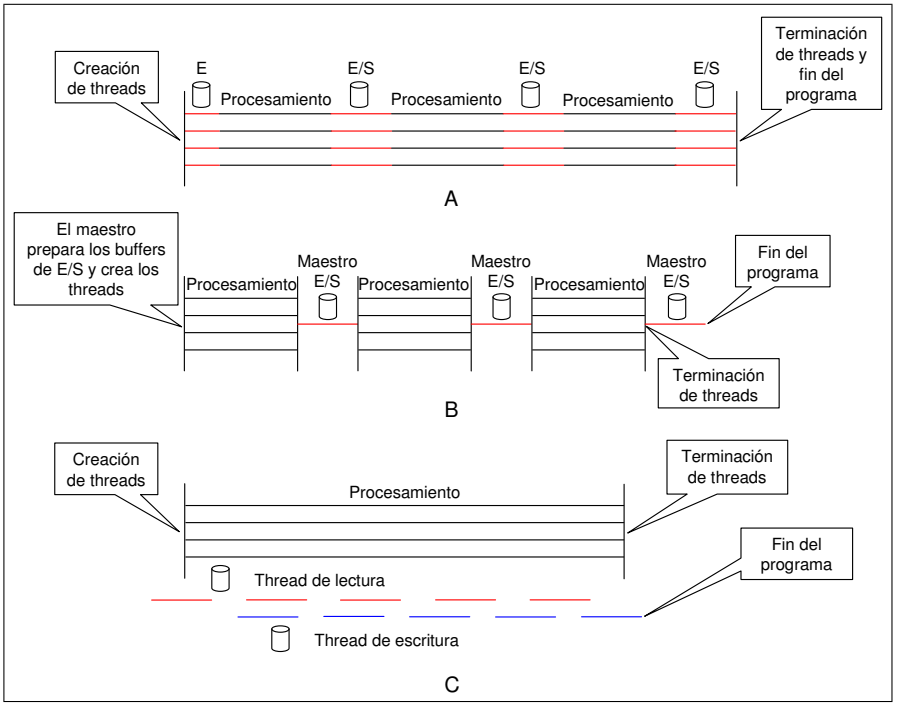

Figura 4.1: Estrategias usando multithreading. En cada una hay representados cuatro threads mediante líneas negras horizontales y paralelas. En A la asignación de carga es estática. Obsérvese que antes de comenzar a reconstruir, los threads llenan sus buffers de entrada. En B la asignación de carga es dinámica. Un hilo maestro prepara los buffers de E/S y crea los threads trabajadores. El maestro también efectúa la E/S cuando los buffers se llenan o vacían. En C la asignación de carga es como en B, pero el maestro es sustituido por dos hilos de E/S que se ejecutan cada cierto tiempo de forma concurrente con los trabajadores. El thread de lectura empieza antes su ejecución para llenar el buffer de entrada y acaba también antes al no haber ya más sinogramas. El thread de escritura comienza a escribir una vez que se han reconstruido algunas rebanadas y termina al volcar en disco el último lote de rebanadas.

su optimización. Una reconstrucción se lleva a cabo de la siguiente manera: se van leyendo de disco los sinogramas, a partir de los cuales son reconstruidas las rebanadas haciendo uso del algoritmo elegido, y éstas se escriben en disco. La aplicación original de la que nosotros partimos cargaba los sinogramas de uno en uno y escribía las rebanadas también de una en una, lo cual se traducía en un uso ineficiente del sistema de E/S debido a la gran cantidad de accesos a disco realizados.

Nosotros proponemos el uso de buffers de E/S para optimizar las lecturas y escrituras de disco. Un buffer de E/S es una región de memoria, estructurada como muestra la Figura 4.3, que tiene capacidad para albergar una cierta cantidad de sinogramas o rebanadas. Llamamos buffer de entrada o de lectura a un buffer que guarda sinogramas, y llamamos buffer de salida o de escritura a un buffer que almacena rebanadas. Si antes se leía un único sinograma, se procesa-

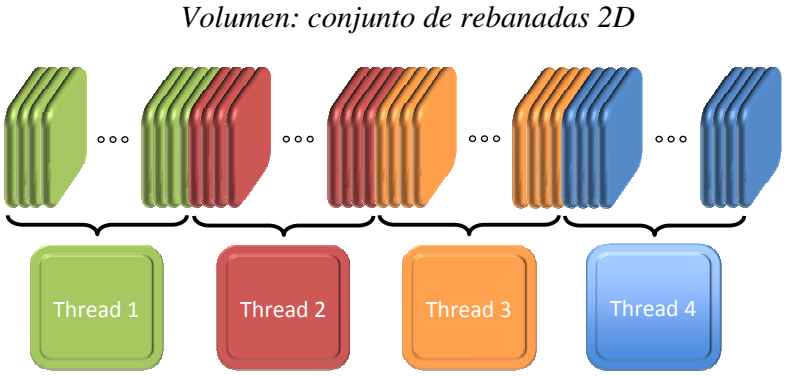

Figura 4.2: Asignación estática de carga. El volumen se reparte a partes iguales entre los threads, que se ejecutan en paralelo y reconstruyen sus rebanadas de cuatro en cuatro gracias al procesamiento vectorial. Se tienen, pues, dos niveles de paralelismo: threads e instrucciones SSE. Aunque en la figura se muestren sólo cuatro threads, se crearán tantos como núcleos haya disponibles.

ba y se escribía la rebanada resultante, ahora la E/S se hace a nivel de buffers. Esto significa que cada thread llenará su buffer de entrada con sinogramas y según los procesa, irá escribiendo las rebanadas reconstruidas en su buffer de salida. Cuando el buffer de lectura se queda vacío, se vuelve a llenar, al igual que cuando el buffer de escritura se completa, se vuelca en disco.

El uso de los buffers de E/S implica que el número de accesos a disco se ve reducido en gran medida, habiendo menos cuanto más grandes sean los buffers. Así, si un determinado volumen está compuesto por 64 rebanadas, sin los buffers de E/S se requerirán 128 accesos (64 para leer los sinogramas y otros 64 para escribir las rebanadas), mientras que si usamos, pongamos por caso, buffers de lectura de tamaño 32 y buffers de escritura de tamaño 16, solamente se necesitarán 6.

En este primer enfoque los buffers de E/S son privados, es decir, cada thread tiene un buffer propio de lectura y otro de escritura. Uno de los problemas derivados de este diseño es que el consumo de memoria puede llegar a ser muy elevado. Imaginemos, por ejemplo, que empleamos buffers de E/S de 128 entradas para reconstuir un conjunto de datos con 140 imágenes de proyección de tamaño 1024×1024 en las que cada píxel ocupa cuatro bytes. Si lanzamos cuatro threads de procesamiento, los buffers de lectura precisarán de 280MB y los de escritura, de 2GB. Como vemos, la cifra es importante, sin ser excesivo el tamaño de la reconstrucción ni el número de threads.

Otro problema que tiene este diseño es que todos los threads tienen permiso para realizar E/S, lo cual puede provocar que se perjudiquen los unos a los otros. Cuando un thread lee de disco, se suelen leer más datos de los requeridos, que quedan almacenados en la caché de disco por si se solicitasen en un futuro más o menos cercano. Si otro thread obtiene el disco para efectuar E/S, es bastante probable que sobreescriba los datos en caché con los suyos propios. Así, cuando el thread inicial quiera leer de nuevo, no encontrará sus datos en la caché, y estos tendrán que releerse. A su vez, este thread sobreescribirá los datos de los demás, repitiéndose este fenómeno continuamente. Por otro lado, con este diseño

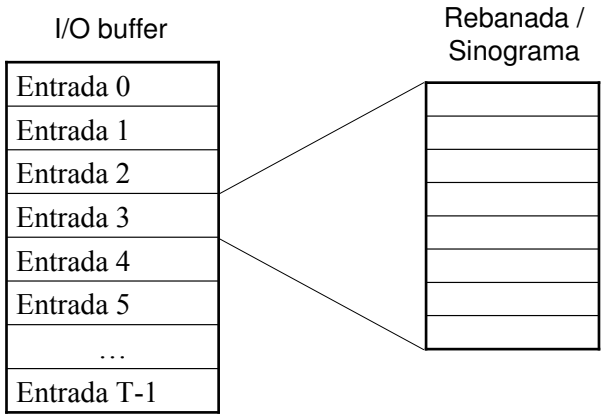

Figura 4.3: Un buffer de E/S puede ser de lectura o escritura. En el primero de los casos, cada entrada guarda un sinograma, mientras que en el segundo, cada entrada almacena una rebanada.

estaríamos obligando al disco a tener que posicionarse una y otra vez, ya que no estaríamos leyendo ni escribiendo de forma secuencial.

Por último, debemos mencionar que si bien la asignación estática de carga es ventajosa debido a que los threads no necesitan comunicarse entre ellos y, por tanto, no hay tiempos muertos generados por las comunicaciones, implica la inexistencia de balanceo de carga. Así pues, si la ejecución de un thread fuera más lenta que la de los demás, provocaría que el tiempo total de la reconstrucción se incrementara. Lo adecuado en tales circunstancias sería que el thread lento procesase menos carga que los rápidos para paliar en la medida de lo posible el retraso.

Los otros enfoques que vamos a analizar tratan de corregir los tres problemas que acabamos de exponer: consumo excesivo de memoria, interferencias en la E/S y ausencia de balanceo de carga.

4.3.2. Asignación dinámica de carga

La gran diferencia entre este diseño (Figura 4.1B) y el anterior es que los buffers de E/S ya no se replican, sino que existe un único buffer de entrada y un único buffer de salida, los cuales son compartidos por todos los threads de procesamiento. Esto provoca que el consumo de memoria se vea reducido drásticamente. Si volvemos al ejemplo que pusimos en el apartado anterior, la memoria que ahora requieren los buffers de E/S es de 70 y 512 megabytes, respectivamente, sin importar el número de threads usado.

Este nuevo enfoque se basa en el paradigma maestro/esclavo. Los esclavos son los threads trabajadores o de procesamiento, y el maestro es un thread que se encarga de la creación de los trabajadores y de gestionar los buffers de E/S —i.e. llenarlos o vaciarlos, según corresponda—.

Antes de nada, el maestro llena el buffer de lectura y prepara el de escritura. Realizadas estas operaciones, crea a los trabajadores y permanece a la escucha. Los trabajadores acceden de forma autónoma al buffer de lectura para asignarse carga de trabajo. En cada acceso cogen cuatro sinogramas para procesarlos

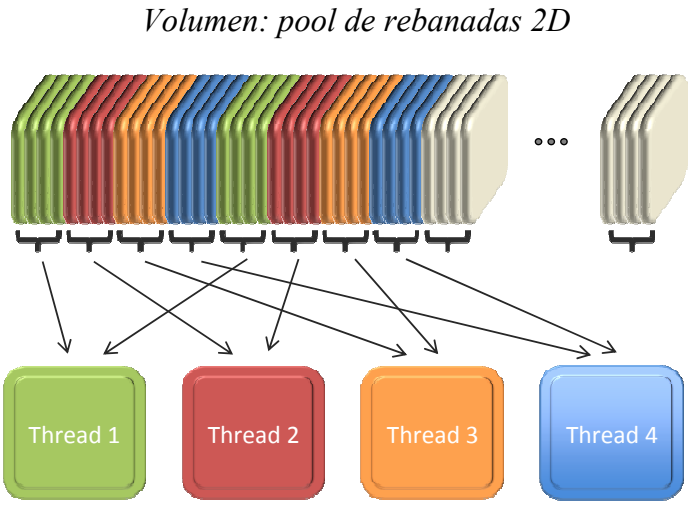

Figura 4.4: Asignación dinámica de carga. El volumen se puede ver como un *pool* o almacén de rebanadas al que los threads acceden de forma autónoma para asignarse carga de trabajo. En cada acceso un thread coge cuatro rebanadas para reconstruirlas mediante procesamiento vectorial. Cuando termina con esas cuatro, accede de nuevo al *pool*. Este diseño permite que los threads más rápidos entren con mayor frecuencia al almacén y procesen más rebanadas, no viéndose retrasados por los lentos.

con instrucciones SSE. Cuando las cuatro rebanadas correspondientes han sido reconstruidas, son depositadas en el buffer de escritura y regresan en busca de trabajo al buffer de lectura. De aquí se deduce que ahora existe un balanceo de carga implícito, pues los threads más rápidos accederán más frecuentemente al buffer de lectura y reconstruirán más rebanadas. Este esquema de funcionamiento se ilustra en la Figura 4.4, si bien aquí se han omitido los buffers de E/S para hacer más sencilla la explicación.

Llegará un punto en que el buffer de entrada se vacíe o el de salida se llene, o ambas situaciones sucedan a la misma vez. Cuando uno de los trabajadores detecte alguno de estos estados, comprobará primero si los demás threads siguen activos o han acabado. Si ninguno está activo, le notificará al maestro la situación y éste preparará los buffers de E/S para que la reconstrucción pueda proseguir. Si ya no hay más rebanadas que reconstruir, avisará a los trabajadores y estos terminarán. Antes de comunicarse con el maestro, es preciso que no haya trabajadores activos. Si los hay, significa que hay sinogramas que aún están siendo procesados, por lo que las entradas que ocupan en el buffer de lectura no podrán ser ocupadas por otros sinogramas.

Como vemos, ahora los threads de procesamiento no tienen permiso para efectuar E/S, ya que avisan al maestro para que lleve a cabo las operaciones de vaciado y llenado de buffers. Por consiguiente, podemos afirmar que las operaciones de E/S están *ordenadas* y, en teoría, deberían desaparecer —o al menos mitigarse— las interferencias que antes podían existir.

Puesto que los buffers de E/S son un recurso compartido, podrían quedar en un estado inconsistente si varios trabajadores accedieran a ellos a la misma vez.

Hay, pues, que limitar su uso a un único thread en un determinado instante, pero hay que evitar que dicho thread permanezca mucho tiempo en la sección crítica si no queremos degradar el rendimiento de la reconstrucción. Para resolver este problema, nosotros obligamos a los trabajadores a que adquieran un mutex antes de acceder a los buffers de E/S. Los niveles de los buffers —cómo de vacíos o llenos están— vienen determinados por unos índices enteros que, además, marcan la entrada del siguiente sinograma a procesar y qué entrada en el buffer de escritura debe ocupar la rebanada resultante. Por tanto, una vez dentro de la sección crítica, el thread hace una copia local de estos índices y los actualiza con el fin de reservarse esas entradas y evitar que los demás threads puedan cogerlas, saliendo inmediatamente después.

Si bien hemos atacado y hemos dado una solución a los tres problemas que se nos planteaban en el diseño con asignación estática de carga, este nuevo enfoque no está exento de inconvenientes. Puesto que todos los threads tienen que haber terminado su procesamiento antes de avisar al maestro para que vuelva a preparar los buffer de E/S, si hay un thread lento, los rápidos estarán parados mientras que él no acabe, por lo que este problema no está del todo resuelto. Por otro lado, durante el tiempo en el que el maestro realiza la E/S, no es posible reconstruir nada. Sin embargo, podríamos intentar solapar la E/S con el procesamiento de las imágenes de manera que ambas operaciones —reconstrucción y E/S— se llevasen a cabo de manera concurrente y no de forma secuencial como ahora ocurre.

4.3.3. Asignación dinámica de carga con E/S asíncrona

Este último enfoque (Figura 4.1C) trata de resolver los dos problemas de los que adolece el diseño que acabamos de analizar: un trabajador lento es aún capaz de retrasar a los rápidos, y durante el tiempo que se realiza E/S no se reconstruye nada.

La asignación dinámica de carga explicada en el apartado anterior e ilustrada en la Figura 4.4 se sigue manteniendo, pero el maestro encargado de gestionar los buffers es sustituido por dos threads independientes que asumen la ejecución de las operaciones de E/S, ocupándose uno de ellos de la lectura de sinogramas y el otro, de la escritura de rebanadas. La E/S se lleva a cabo de modo asíncrono porque los trabajadores no avisan a los threads de E/S para que llenen o vacíen los buffers, sino que estos últimos comprobarán los niveles de los buffers cada cierto tiempo, y volcarán en disco las rebanadas ya reconstruidas y reemplazarán a los sinogramas ya procesados por nuevos.

Ahora la ejecución de los trabajadores no se ve interrumpida por la E/S, ya que ésta se solapa en el tiempo con el procesamiento de las imágenes al realizarse ambas operaciones de forma concurrente. Además, un trabajador lento no retrasará al resto, pues para efectuar el vaciado o llenado de buffers no es preciso que los trabajadores estén inactivos, algo obligatorio en el diseño anterior.

Podría ocurrir que la reconstrucción fuese más rápida que la E/S, por lo que resultaría imposible solapar ésta con aquélla completamente. En tales circunstancias, o los trabajadores no tendrán sinogramas para procesar o no dispondrán de entradas en el buffer de escritura donde poder almacenar las rebanadas que reconstruyen, implicando esto que habrán de esperar hasta que los buffers de

E/S estén preparados de nuevo. Esta situación es mucho más típica en WBP que en SIRT debido a la menor complejidad computacional del primero.

Es importante resaltar que el disco duro no es un recurso al que distintos procesos o threads puedan acceder simultáneamente y, por tanto, la lectura de sinogramas y escritura de rebanadas no puede llevarse a cabo paralelamente. No obstante, este tercer diseño permite el uso de dos discos duros diferentes para que la E/S sí pueda efectuarse en paralelo. De cara al usuario, este mecanismo es totalmente transparente, pues lo único que él ha de proporcionar al programa es la ruta de los ficheros, y bastará con que la ruta del de entrada pertenezca a un disco duro y la del de salida se corresponda con el otro disco.

Capítulo 5

Resultados experimentales

Este capítulo presenta los resultados de la investigación llevada a cabo en esta tesis. Dichos resultados tienen que ver con la aplicación sobre los algoritmos de reconstrucción WBP y SIRT de las optimizaciones comentadas en los capítulos 2, 3 y 4, y los hemos ordenado en cuatro apartados. El primero analiza los relativos al tiempo de reconstrucción, el segundo examina los de E/S, el tercero se centra en el balanceo de carga y el cuarto realiza una comparación entre los tiempos de procesamiento obtenidos por nuestros algoritmos WBP y SIRT, y los ofrecidos por las implementaciones más recientes de estos métodos en GPUs.

En las pruebas efectuadas se emplearon dos computadores diferentes. Uno de ellos estaba basado en un procesador Intel Core 2 Quad Q9550 a 2.83GHz con 4 núcleos y 12MB de caché L2. Además, disponía de 8GB de RAM. El otro estaba formado por dos procesadores Intel Xeon E5405 a 2.00GHz, cada uno con 4 núcleos y 12MB de caché L2. Este computador tenía 16GB de RAM. En el resto de este capítulo nos referiremos a ellos mediante los nombres Q9550 y E5405. Los algoritmos WBP y SIRT, en todas sus versiones y configuraciones, fueron compilados con el compilador de Intel (ICC) usando los parámetros de optimización -O3 y -xT[1]. Todos los experimentos se hicieron en el SO Linux.

5.1. Tiempos de reconstrucción

Para medir los tiempos de reconstrucción de nuestros algoritmos WBP y SIRT utilizamos seis conjuntos de datos distintos. Dos estaban formados por imágenes de proyección de tamaño 512×512 píxeles; otros dos, por imágenes de tamaño 1024×1024, y los dos restantes, por imágenes de tamaño 2048×2048[2]. La diferencia entre conjuntos con imágenes de igual tamaño estriba en la cantidad de las mismas. Así, siempre habrá uno con 70 y otro con 140. Los conjuntos de 70 imágenes se tomaron en el rango $[-70°, +68°]$ con incrementos de 2°, mientras que los de 140 se adquirieron grado a grado dentro del intervalo $[-70°, +69°]$. A la hora de reconstruir cada conjunto de datos, nosotros seleccionamos diferentes grosores para las rebanadas. Para los conjuntos

[1] -O3 es el grado de optimización genérico máximo y -xT produce código especializado para correr en arquitecturas Core 2, que es en la que están basados tanto el Q9550 como el E5405.

[2] En lo que sigue, para referirnos a un conjunto de datos con imágenes de $N \times N$, escribiremos solamente N.

de 512, los grosores fueron 128, 256 y 512, lo que da lugar a volúmenes 3D de 512×128×512, 512×256×512 y 512×512×512, respectivamente[3]. En los de 1024 los grosores fueron 256, 512 y 1024, lo que genera volúmenes de 1024×256×1024, 1024×512×1024 y 1024×1024×1024, respectivamente. Por último, los grosores para los conjuntos de 2048 fueron 256, 512 y 2048. Esto origina volúmenes de 2048×256×2048, 2048×512×2048 y 2048×2048×2048, respectivamente. Debido a sus características (tamaño de las imágenes de proyección, cantidad de las mismas y grosor de la reconstrucción), estos conjuntos de datos son típicos en tomografía electrónica.

Cada uno de los volúmenes 3D anteriores fue reconstruido por los algoritmos WBP y SIRT, aplicándose diferentes grados de optimización y empleando las dos máquinas disponibles (Q9550 y E5405). Cada tiempo de reconstrucción mostrado es la media de cinco ejecuciones, estando todos expresados en segundos. Los resultados de WBP pueden observarse en las tablas 5.1 y 5.3. La primera presenta los tiempos en el Q9550, mientras que la segunda se refiere al E5405. Como se puede apreciar, cada una de ellas se encuentra dividida en otras seis tablas más pequeñas, las cuales se corresponden con los conjuntos de datos comentados en el párrafo anterior. Para cada volumen reconstruido, primero se compara el tiempo de WBP original con el obtenido por las optimizaciones básicas, a las que luego se une el procesamiento vectorial y más tarde el multithreading. En estas tablas, *Original* denota el algoritmo WBP sin optimizar del que partimos[4], *Básicas* es la versión secuencial refinada mediante las optimizaciones básicas[5] y *SSE* añade el procesamiento vectorial. En estos casos, solamente existe un thread de procesamiento. *2T*, *4T* y *8T* indican que se han creado dos, cuatro y ocho threads para aprovechar dos, cuatro y ocho núcleos. En las pruebas nunca se lanzan más threads que núcleos existentes, por lo que en el Q9550 se tendrán cuatro threads como máximo, y en el E5405, ocho. Los speedups mostrados son acumulativos. Así pues, una fila en la que ponga *2T* nos estará informando de que se han creado dos threads, cada uno ejecutando un algoritmo WBP al que se han aplicado las optimizaciones básicas y el procesamiento vectorial. Nótese cómo el tiempo de reconstrucción se decrementa conforme añadimos nuevas optimizaciones, alcanzándose speedups de 80x en el Q9550 y de 160x en el E5405.

Las tablas 5.2 y 5.4 son resúmenes de la 5.1 y la 5.3, respectivamente. Muestran el factor de aceleración que aporta cada optimización por separado, y también el crecimiento global del speedup con respecto a la versión original y a la secuencial optimizada. Tanto en el Q9550 como en el E5405 las optimizaciones básicas proporcionan un speedup superior a 6x. Con las instrucciones SSE conseguimos acelerar los algoritmos en un factor que se acerca a 3.5x, lo cual está muy bien si tenemos en cuenta que 4x es el ideal. Al usar multithreading y hacer la media de los valores de ambas máquinas, obtenemos que con dos núcleos el speedup crece hasta 2x y con cuatro roza 3.8x. En el caso del E5405, con ocho núcleos el speedup es 7.4x. Si acumulamos todos estos speedups individuales, resulta que la combinación optimizaciones básicas-procesamiento vectorial aporta una aceleración global en torno a 21x. Con dos núcleos dicha aceleración se sitúa aproximadamente en 42x, con cuatro en 80x y con ocho en 160x.

[3] Un volumen de tamaño $X \times Y \times Z$ está formado por Z rebanadas de $X \times Y$ píxeles, donde Y denota el grosor de la reconstrucción.

[4] Recordamos que también fue compilado con el ICC usando los modificadores -O3 y -xT.

[5] A veces también la llamaremos "versión secuencial optimizada".

Los resultados de SIRT se presentan en las tablas 5.5 (Q9550) y 5.7 (E5405). Las tablas-resumen de las mismas son la 5.6 y la 5.8, respectivamente. El número de iteraciones seleccionado para SIRT fue 30, que es una cifra común en tomografía electrónica. Teniendo en cuenta que SIRT se basa en la rutina de backprojection (consultar Capítulo 1), el algoritmo fue programado directamente empleando las optimizaciones básicas que ya habíamos aplicado a WBP. Por tanto, podemos suponer que con ellas el speedup sobre una supuesta versión sin optimizar de SIRT sería también en torno a 6x. Además, los resultados de las tablas 5.6 y 5.8 corroborarían este hecho, ya que los speedups mostrados son del mismo orden que los conseguidos en WBP. Más exactamente, son algo superiores debido a la mayor carga computacional de SIRT. Si analizamos ambas tablas en conjunto, veremos que utilizando las instrucciones SSE el factor de aceleración es ligeramente superior a 3.6x. Empleando 2 y 4 núcleos podemos afirmar que el speedup es lineal con el número de núcleos, pues en media logramos 2x y rozamos 4x respectivamente. Al usar ocho núcleos en el E5405, el speedup alcanzado es 7.6x. Si acumulamos todos estos speedups individuales y suponemos que con las optimizaciones básicas conseguiríamos un speedup por encima de 6x, tendríamos que la combinación optimizaciones básicas-procesamiento vectorial nos ofrece un speedup global cercano a 23x. Con dos núcleos dicha aceleración crecería hasta encontrarse en torno a 45x, con cuatro estaría alrededor de 90x y con ocho superaríamos 170x.

En general, podemos decir que los speedups son bastante homogéneos entre máquinas, volúmenes y algoritmos. No obstante, un examen más detallado nos permite observar que en WBP parece haber una tendencia ligeramente al alza con el número de ángulos y también, aunque muy sutil, con el grosor. En cambio, en SIRT los speedups son más estables en este sentido. En ambos algoritmos, se aprecia una tenue caída con el tamaño de las imágenes de proyección.

Para finalizar este apartado, la Figura 5.1 presenta una gráfica con los speedups alcanzados mediante cada optimización por separado teniendo en cuenta las dos máquinas, los dos algoritmos y todos los volúmenes reconstruidos. Se ha construido promediando las primeras filas de las tablas 5.2, 5.4, 5.6 y 5.8. Por ejemplo, para el caso del procesamiento vectorial se ha hecho la media entre todas las celdas *SSE* pertenecientes a esas primeras filas. Se puede observar que con las optimizaciones básicas se consigue un speedup por encima de 6x y con el procesamiento vectorial se logra un factor en torno a 3.5x. Usando dos núcleos llegamos a 2x, utilizando cuatro nos quedamos muy cerca de 4x y al emplear ocho subimos hasta 7.5x. Por otro lado, la Figura 5.2 es una gráfica que muestra los speedups obtenidos al ir acumulando los aportados por cada optimización individualmente. Para hallarlos se tuvieron en cuenta los dos computadores, los dos algoritmos y todos los volúmenes reconstruidos, por lo que se promediaron las diagonales en gris claro de las tablas 5.2, 5.4, 5.6 y 5.8. El speedup gracias a las optimizaciones básicas es igual que antes, es decir, superior a 6x. Al añadir las instrucciones SSE crecemos algo por encima de 20x. Al incorporar el uso de dos, cuatro y ocho núcleos, nos situamos en el entorno de 40x, 80x y 160x, respectivamente. Con speedups de este orden, la reducción del tiempo de reconstrucción es espectacular. Por ejemplo, un algoritmo SIRT sin optimizar ejecutado en el E5405 tardaría en reconstruir el volumen más grande (2048×2048×2048 con 140 ángulos) $197301{,}50 \times 6{,}22 \approx 2$ *semanas*. Ahora, empleando la configuración más rápida (optimizaciones básicas, SSE y ocho núcleos), sólo necesitaría $7421{,}35 \approx 2$ *horas*.

512	128	256	512
Original	30,32	57,99	110,07
Básicas	4,77	9,38	17,57
Speedup	6,36	6,18	6,26
SSE	1,44	2,73	5,32
Speedup	21,06	21,24	20,69
2T	0,73	1,38	2,65
Speedup	41,53	42,02	41,54
4T	0,38	0,76	1,47
Speedup	79,79	76,30	74,88

512	128	256	512
Original	60,49	115,67	220,35
Básicas	9,52	18,62	34,78
Speedup	6,35	6,21	6,34
SSE	2,84	5,32	10,15
Speedup	21,30	21,74	21,71
2T	1,42	2,69	5,02
Speedup	42,60	43,00	43,89
4T	0,75	1,41	2,64
Speedup	80,65	82,04	83,47

1024	256	512	1024
Original	233,97	456,77	872,81
Básicas	37,92	74,27	139,25
Speedup	6,17	6,15	6,27
SSE	11,79	23,37	43,11
Speedup	19,84	19,55	20,25
2T	5,86	11,53	21,13
Speedup	39,93	39,62	41,31
4T	3,37	6,46	11,18
Speedup	69,43	70,71	78,07

1024	256	512	1024
Original	466,68	911,33	1740,64
Básicas	75,04	146,81	275,70
Speedup	6,22	6,21	6,31
SSE	22,36	42,98	78,45
Speedup	20,87	21,20	22,19
2T	10,96	20,74	37,89
Speedup	42,58	43,94	45,94
4T	5,98	11,01	21,05
Speedup	78,04	82,77	82,69

2048	256	512	1024
Original	944,07	1836,03	7022,88
Básicas	150,76	302,74	1130,89
Speedup	6,26	6,06	6,21
SSE	47,96	91,54	336,99
Speedup	19,68	20,06	20,84
2T	23,96	46,21	171,05
Speedup	39,40	39,73	41,06
4T	13,10	24,87	87,32
Speedup	72,07	73,83	80,43

2048	256	512	1024
Original	1883,27	3663,11	13975,17
Básicas	298,32	597,96	2231,86
Speedup	6,31	6,13	6,26
SSE	89,42	172,71	628,37
Speedup	21,06	21,21	22,24
2T	44,92	86,31	315,73
Speedup	41,92	42,44	44,26
4T	26,91	50,07	170,55
Speedup	69,98	73,16	81,94

Tabla 5.1: WBP en el Q9550. Se muestran los conjuntos de datos 512, 1024 y 2048, tanto con 70 ángulos (izquierda) como con 140 (derecha). Los speedups son acumulativos, es decir, primero se tienen en cuenta las optimizaciones básicas (*Básicas*), luego se añade el procesamiento vectorial (*SSE*) y, por último, el multithreading con dos (*2T*) y cuatro threads (*4T*).

WBP Q9550			
Básicas	SSE	2T	4T
6,24	3,36	2,00	3,69
	20,97	6,72	
		41,93	12,40
			77,37

Tabla 5.2: Speedups globales de WBP en el Q9550. La primera fila de la tabla presenta los speedups teniendo en cuenta cada una de las optimizaciones por separado. La diagonal en gris claro muestra el crecimiento del speedup con respecto a la versión original, mientras que la oscura lo hace en relación a la versión secuencial con las optimizaciones básicas. Todos estos speedups se han calculado a partir de los tiempos de la Tabla 5.1 y son promedios que tienen en cuenta a todas las reconstrucciones de dicha tabla. Por ejemplo, 20,97 es el speedup medio que obtendremos con WBP en el Q9550 al combinar las optimizaciones básicas y el procesamiento vectorial.

512	128	256	512
Original	42,82	81,98	155,57
Básicas	6,77	13,24	24,81
Speedup	6,32	6,19	6,27
SSE	2,00	3,80	7,19
Speedup	21,41	21,57	21,64
2T	1,03	1,94	3,68
Speedup	41,57	42,26	42,27
4T	0,52	0,98	1,88
Speedup	82,35	83,65	82,75
8T	0,27	0,52	1,00
Speedup	158,59	157,65	155,57

512	128	256	512
Original	85,40	163,48	311,43
Básicas	13,43	26,30	49,17
Speedup	6,36	6,22	6,33
SSE	3,94	7,45	13,88
Speedup	21,68	21,94	22,44
2T	2,01	3,80	7,07
Speedup	42,49	43,02	44,05
4T	1,02	1,90	3,57
Speedup	83,73	86,04	87,24
8T	0,53	0,99	1,86
Speedup	161,13	165,13	167,44

1024	256	512	1024
Original	330,37	645,63	1234,16
Básicas	53,52	105,49	199,76
Speedup	6,17	6,12	6,18
SSE	15,83	31,35	58,76
Speedup	20,87	20,59	21,00
2T	8,09	16,03	30,06
Speedup	40,84	40,28	41,06
4T	4,14	8,19	15,34
Speedup	79,80	78,83	80,45
8T	2,14	4,24	7,97
Speedup	154,38	152,27	154,85

1024	256	512	1024
Original	659,07	1287,79	2460,71
Básicas	106,25	209,32	394,63
Speedup	6,20	6,15	6,24
SSE	30,72	59,50	111,00
Speedup	21,45	21,64	22,17
2T	15,62	30,32	56,34
Speedup	42,19	42,47	43,68
4T	7,91	15,31	28,68
Speedup	83,32	84,11	85,80
8T	4,21	8,07	14,94
Speedup	156,55	159,58	164,71

2048	256	512	1024
Original	1333,17	2608,58	9857,23
Básicas	213,72	426,64	1601,55
Speedup	6,24	6,11	6,15
SSE	65,11	126,33	465,81
Speedup	20,48	20,65	21,16
2T	32,95	64,48	240,19
Speedup	40,46	40,46	41,04
4T	16,84	32,96	121,92
Speedup	79,17	79,14	80,85
8T	8,41	16,54	62,10
Speedup	158,52	157,71	158,73

2048	256	512	1024
Original	2658,91	5179,14	19651,07
Básicas	424,15	843,73	3156,02
Speedup	6,27	6,14	6,23
SSE	124,42	240,22	883,44
Speedup	21,37	21,56	22,24
2T	62,95	121,68	449,32
Speedup	42,24	42,56	43,74
4T	32,19	62,08	233,83
Speedup	82,60	83,43	84,04
8T	15,90	31,03	124,35
Speedup	167,23	166,91	158,03

Tabla 5.3: WBP en el E5405. Se muestran los conjuntos de datos 512, 1024 y 2048, tanto con 70 (izquierda) como con 140 ángulos (derecha). Los speedups son acumulativos, es decir, primero se tienen en cuenta solamente las optimizaciones básicas (*Básicas*), luego se añade el procesamiento vectorial (*SSE*) y, por último, el multithreading con dos (*2T*), cuatro (*4T*) y 8 threads (*8T*).

WBP E5405				
Básicas	SSE	2T	4T	8T
6,22	3,45	1,96	3,85	7,43
	21,46	6,76		
		42,06	13,28	
			82,62	25,63
				159,44

Tabla 5.4: Speedups globales de WBP en el E5405. La primera fila de la tabla presenta los speedups teniendo en cuenta cada una de las optimizaciones por separado. La diagonal en gris claro muestra el crecimiento del speedup con respecto a la versión original, mientras que la oscura lo hace en relación a la versión secuencial con las optimizaciones básicas. Todos estos speedups se han calculado a partir de los tiempos de la Tabla 5.3 y son promedios que incluyen a todas las reconstrucciones de dicha tabla. Por ejemplo, 3,45 es el speedup medio que obtendremos con WBP en el E5405 al usar instrucciones SSE.

512	128	256	512
Básicas	295,45	585,22	1102,19
SSE	79,79	157,77	297,94
Speedup	3,70	3,71	3,70
2T	40,04	78,86	149,19
Speedup	7,38	7,42	7,39
4T	20,11	40,00	76,04
Speedup	14,69	14,63	14,49

512	128	256	512
Básicas	590,67	1170,37	2204,70
SSE	159,56	315,75	596,40
Speedup	3,70	3,71	3,70
2T	79,77	157,87	298,29
Speedup	7,40	7,41	7,39
4T	40,34	79,90	150,68
Speedup	14,64	14,65	14,63

1024	256	512	1024
Básicas	2338,31	4638,96	8749,40
SSE	644,33	1284,48	2427,68
Speedup	3,63	3,61	3,60
2T	322,35	642,95	1214,71
Speedup	7,25	7,22	7,20
4T	165,83	328,01	617,84
Speedup	14,10	14,14	14,16

1024	256	512	1024
Básicas	4678,15	9279,49	17498,46
SSE	1288,08	2553,72	4817,30
Speedup	3,63	3,63	3,63
2T	644,13	1276,75	2407,79
Speedup	7,26	7,27	7,27
4T	331,62	656,83	1239,69
Speedup	14,11	14,13	14,12

2048	256	512	1024
Básicas	9339,05	18658,64	69811,66
SSE	2602,02	5189,61	19429,18
Speedup	3,59	3,60	3,59
2T	1301,95	2596,91	9716,41
Speedup	7,17	7,18	7,18
4T	674,99	1357,21	5080,08
Speedup	13,84	13,75	13,74

2048	256	512	1024
Básicas	18692,14	37329,24	139587,20
SSE	5201,90	10371,88	38689,48
Speedup	3,59	3,60	3,61
2T	2599,79	5171,56	19346,10
Speedup	7,19	7,22	7,22
4T	1380,33	2731,51	10209,65
Speedup	13,54	13,67	13,67

Tabla 5.5: SIRT en el Q9550. Se muestran los conjuntos de datos 512, 1024 y 2048, tanto con 70 (izquierda) como con 140 ángulos (derecha). Los speedups son acumulativos, es decir, primero se tiene en cuenta el procesamiento vectorial (*SSE*) y luego se añade el multithreading con dos (*2T*) y cuatro threads (*4T*). En SIRT partimos directamente de la versión secuencial optimizada, por lo que no hemos incluido el speedup relativo a las optimizaciones básicas.

SIRT Q9550			
Básicas	SSE	2T	4T
6,24	3,64	2,00	3,89
	22,71	7,28	
		45,43	14,16
			88,36

Tabla 5.6: Speedups globales de SIRT en el Q9550. La primera fila de la tabla presenta los speedups teniendo en cuenta cada una de las optimizaciones por separado. El speedup correspondiente a las optimizaciones básicas es virtual, y se ha tomado de WBP (ver Tabla 5.2). La diagonal en gris claro muestra el crecimiento del speedup con respecto a una supuesta versión sin optimizar, mientras que la oscura lo hace en relación a la versión secuencial con las optimizaciones básicas. Todos estos speedups —salvo los de la diagonal en gris claro— se han calculado a partir de los tiempos de la Tabla 5.5 y son promedios que tienen en cuenta a todas las reconstrucciones de dicha tabla. Por ejemplo, 14,16 es el speedup medio que obtendremos con SIRT en el Q9550 cuando usemos procesamiento vectorial y los cuatro núcleos del procesador.

512	128	256	512
Básicas	417,68	827,47	1558,20
SSE	113,06	223,56	421,93
Speedup	3,69	3,70	3,69
2T	56,54	111,82	211,03
Speedup	7,39	7,40	7,38
4T	28,30	55,95	105,62
Speedup	14,76	14,79	14,75
8T	14,24	28,35	54,05
Speedup	29,33	29,19	28,83

512	128	256	512
Básicas	835,49	1655,31	3116,47
SSE	226,21	447,44	844,64
Speedup	3,69	3,70	3,69
2T	113,17	223,88	422,52
Speedup	7,38	7,39	7,38
4T	56,61	111,97	211,37
Speedup	14,76	14,78	14,74
8T	28,66	56,78	107,05
Speedup	29,15	29,15	29,11

1024	256	512	1024
Básicas	3310,56	6564,33	12372,28
SSE	910,34	1812,14	3418,82
Speedup	3,64	3,62	3,62
2T	456,01	908,41	1714,74
Speedup	7,26	7,23	7,22
4T	228,28	455,61	860,09
Speedup	14,50	14,41	14,38
8T	119,58	236,73	447,01
Speedup	27,69	27,73	27,68

1024	256	512	1024
Básicas	6622,12	13132,02	24754,10
SSE	1821,40	3607,99	6798,68
Speedup	3,64	3,64	3,64
2T	912,10	1805,73	3406,14
Speedup	7,26	7,27	7,27
4T	456,65	904,78	1705,02
Speedup	14,50	14,51	14,52
8T	239,25	473,06	892,81
Speedup	27,68	27,76	27,73

2048	256	512	1024
Básicas	13200,94	26372,74	98646,70
SSE	3665,74	7306,13	27319,60
Speedup	3,60	3,61	3,61
2T	1837,41	3657,19	13701,20
Speedup	7,18	7,21	7,20
4T	922,42	1838,62	6878,55
Speedup	14,31	14,34	14,34
8T	493,52	982,47	3710,51
Speedup	26,75	26,84	26,59

2048	256	512	1024
Básicas	26421,20	52771,40	197301,50
SSE	7323,16	14576,90	54533,76
Speedup	3,61	3,62	3,62
2T	3666,85	7323,99	27280,48
Speedup	7,21	7,21	7,23
4T	1843,53	3671,13	13702,77
Speedup	14,33	14,37	14,40
8T	1014,69	1994,38	7421,35
Speedup	26,04	26,46	26,58

Tabla 5.7: SIRT en el E5405. Se muestran los conjuntos de datos 512, 1024 y 2048, tanto con 70 (izquierda) como con 140 ángulos (derecha). Los speedups son acumulativos, es decir, primero se tiene en cuenta el procesamiento vectorial (*SSE*) y luego se añade el multithreading con dos (*2T*), cuatro (*4T*) y ocho threads (*8T*). En SIRT partimos directamente de la versión secuencial optimizada, por lo que no hemos incluido el speedup relativo a las optimizaciones básicas.

SIRT E5405				
Básicas	SSE	2T	4T	8T
6,22	3,65	2,00	3,99	7,62
	22,70	7,30		
		45,41	14,56	
			90,58	27,81
				173,00

Tabla 5.8: Speedups globales de SIRT en el E5405. La primera fila de la tabla presenta los speedups teniendo en cuenta cada una de las optimizaciones por separado. El speedup correspondiente a las optimizaciones básicas es virtual, y se ha tomado de WBP (ver Tabla 5.4). La diagonal en gris claro muestra el crecimiento del speedup con respecto a una supuesta versión sin optimizar, mientras que la oscura lo hace en relación a la versión secuencial con las optimizaciones básicas. Todos estos speedups —salvo los de la diagonal en gris claro— se han calculado a partir de los tiempos de la Tabla 5.7 y son promedios que incluyen a todas las reconstrucciones de dicha tabla. Por ejemplo, 3,65 es el speedup medio que obtendremos con SIRT en el E5405 al usar únicamente instrucciones SSE.

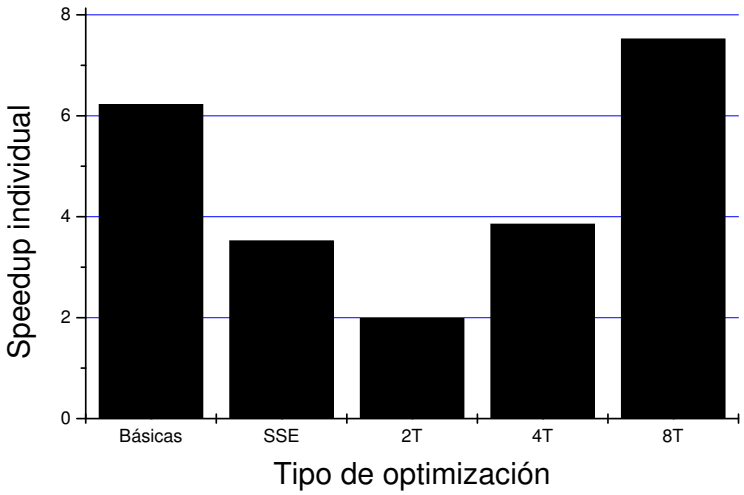

Figura 5.1: Speedups individuales globales. En la gráfica se muestran los speedups conseguidos globalmente con cada optimización por separado. Estos speedups son medias calculadas teniendo en cuenta las dos máquinas (Q9550 y E5405), los dos algoritmos (WBP y SIRT) y todos los volúmenes reconstruidos. Para el caso de *8T* sólo se incluyó el E5405, ya que en el Q9550 disponemos únicamente de cuatro núcleos.

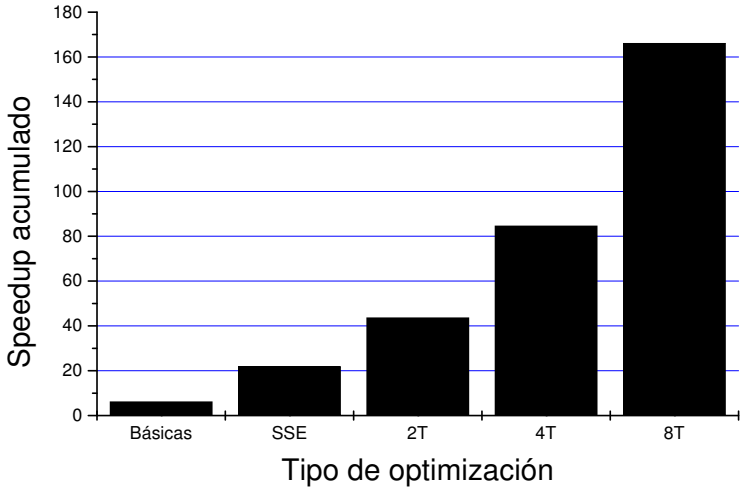

Figura 5.2: Speedups acumulados globales. En la gráfica se muestran los speedups conseguidos globalmente al acumular los resultantes de cada optimización. Estos speedups son medias calculadas teniendo en cuenta las dos máquinas (Q9550 y E5405), los dos algoritmos (WBP y SIRT) y todos los volúmenes reconstruidos. Para el caso de *8T* sólo se incluyó el E5405, ya que en el Q9550 disponemos únicamente de cuatro núcleos.

5.2. Tiempos de E/S

En el apartado anterior hemos analizado los tiempos de reconstrucción, omitiendo en todo momento los de acceso a disco. Recordemos que para llevar a cabo la reconstrucción de un volumen es necesario leer de disco las imágenes de proyección y luego escribir las rebanadas reconstruidas en el mismo. La optimización de la E/S viene motivada por el hecho de que puede llegar a suponer una porción importante del tiempo total del programa[6] cuando los volúmenes ocupan varios gigabytes, lo cual es algo común si las imágenes de proyección tienen una alta resolución.

En el estudio realizado hemos comparado las tres estrategias diseñadas con multithreading que expusimos en el Capítulo 4. Éstas eran asignación estática de carga, asignación dinámica de carga y asignación dinámica de carga con E/S asíncrona. La primera tenía el inconveniente de consumir mucha memoria, pues los buffers de E/S se replicaban tantas veces como threads se creasen. Aparte, todos los threads tenían permiso para acceder al disco, lo cual podía devenir en un uso ineficiente del mismo. La segunda estrategia solventaba estos problemas, pero durante el tiempo que duraba la E/S los threads trabajadores estaban parados. Así, la tercera solapaba la E/S con la reconstrucción de las imágenes. Además, tenía la capacidad de usar dos discos duros, lo que permitía que lectura y escritura de imágenes pudieran correr en paralelo.

Para el estudio de la E/S seleccionamos los conjuntos de datos compuestos por 140 imágenes de proyección con dimensiones 1024×1024 y 2048×2048, cuyos tamaños en disco eran 140MB y 560MB, respectivamente[7]. Para hacer evidentes las diferencias en tiempo entre estrategias, escogimos volúmenes grandes: uno de 1024×1024×1024, otro de 2048×256×2048 y un último de 2048×512×2048. Los dos primeros ocupaban 4GB y el tercero, 8GB[8]. Cada uno se reconstruyó con WBP y con SIRT[9] utilizando las tres estrategias comentadas en el párrafo anterior. Ambos algoritmos se lanzaron en sus configuraciones más rápidas, esto es, con las optimizaciones básicas incluidas, el procesamiento vectorial activado y un thread por cada núcleo existente. Se emplearon buffers de E/S con 16, 32, 64, 128 y 256 entradas. Por ejemplo, 128 quiere decir que el buffer de lectura tenía capacidad para 128 sinogramas y el de escritura, para 128 rebanadas. Las pruebas fueron realizadas en el Q9550 (cuatro núcleos), que además contaba con dos discos duros Serial ATA convencionales a 7200 rpm. Cada prueba se repitió cinco veces y se halló la media. Entre pruebas, la caché de disco de Linux se vació para evitar que los tiempos se falsearan[10]. Estos se expresan en segundos.

Las tablas 5.9, 5.10 y 5.11 presentan los resultados para los volúmenes 1024×1024×1024, 2048×256×2048 y 2048×512×2048, respectivamente. La estrategia que utiliza E/S asíncrona se ejecutó con uno y dos discos duros. $T_{rec.}$ es

[6] El tiempo que pasa desde que se lanza el programa hasta que termina su ejecución, es decir, el *wall time*.

[7] El que tengan tamaños tan reducidos se debe a que están almacenados en modo byte, lo que significa que cada píxel ocupa solamente un byte. Otros modos posibles son el *float* (4 bytes) o el *short* (2 bytes).

[8] Los volúmenes se almacenan en modo *float*.

[9] En SIRT seleccionamos sólo 5 iteraciones, ya que son suficientes para analizar el comportamiento de cada estrategia.

[10] Esto se lleva a cabo mediante el uso de dos comandos: "sync" y "echo 3 >/proc/sys/vm/drop_caches". El primero graba físicamente en el disco los datos que necesiten ser actualizados y el segundo libera la memoria reservada para la caché.

el tiempo de reconstrucción, $T_{E/S}$ es el de E/S y $T_{prog.}$ es el total. Se cumplirá que $T_{prog.} \approx T_{rec.} + T_{E/S}$. Así pues, un descenso del tiempo de E/S repercutirá directamente en el tiempo del programa. Cuando se usa E/S asíncrona, $T_{E/S}$ representa el tiempo de E/S que no se ha podido solapar, que es el que realmente influye en el tiempo total, ya que el resto ha quedado oculto en el procesamiento de las imágenes. *Mem.* es la cantidad de memoria en GB usada. En estas tablas se aprecia que la asignación estática necesita siempre más memoria que las otras. El consumo de memoria crecerá según lo hagan los buffers de E/S. Normalmente, buffers más grandes reducirán el tiempo de E/S, ya que habrá menos accesos a disco.

En general, podemos afirmar que el enfoque con asignación estática es el peor. Como se comprueba en los resultados obtenidos, consume mucha más memoria que los otros enfoques, llegando a ser dicho consumo prohibitivo cuando se usan buffers de E/S grandes. Por ejemplo, en la reconstrucción 2048×512×2048 se precisan más de 5GB con buffers de 256, mientras que con esos mismos buffers las otras estrategias requieren menos de 1,5GB. Si nos centramos en WBP, podemos observar que a igual tamaño de buffer los tiempos de E/S de la estrategia con asignación estática suelen ser mayores que los ofrecidos por las otras. Globalmente en cada volumen esto siempre se cumple, es decir, podemos encontrar un tiempo perteneciente a la asignación dinámica y otro perteneciente a la E/S asíncrona mejores que el más bajo de la asignación estática para el volumen en cuestión. Las diferencias entre tiempos de E/S se acentúan sobre todo al analizar el volumen más grande. Siguiendo con WBP, el enfoque que usa E/S asíncrona resulta el vencedor. Cuando se usa un disco duro, su rendimiento es sólo algo mejor que el proporcionado por la asignación dinámica, pero cuando se emplean dos, se pone de manifiesto toda su potencia y su liderazgo es claro.

Con E/S asíncrona es importante resaltar que existen operaciones de naturaleza secuencial imposibles de solapar con el procesamiento de las imágenes, como la primera carga del buffer de lectura y el último volcado del buffer de escritura. Es por ello que el solapamiento perfecto no existe. Aun así, en SIRT siempre puede conseguirse que el tiempo del programa tienda al tiempo de la reconstrucción. En WBP los resultados son muy buenos, ya que esto también sucede, aunque no de modo tan perfecto. Es destacable el caso del volumen 1024×1024×1024 donde, usando buffers de 256 y dos discos, el tiempo del programa es 28,36 y el de la reconstrucción vale 24,34. Por otro lado, también con E/S asíncrona, se observa que los tiempos de reconstrucción suelen ser ligeramente más altos que con las otras estrategias. Esto es debido a que los núcleos han de ser compartidos entre threads trabajadores y threads de E/S. Aunque estos últimos permanecen dormidos gran parte del tiempo, cuando están despiertos consumen ciclos de CPU. No obstante, este pequeño incremento queda plenamente compensado por la reducción del tiempo de E/S.

El comportamiento de la E/S en SIRT no es igual que en WBP, aunque hay puntos comunes. Los buffers de E/S siguen siendo necesarios si queremos realizar la E/S eficientemente, pero no precisan ser tan grandes como en WBP. Normalmente, con buffers de 64 será suficiente. La asignación estática sigue requiriendo mucha más memoria que las otras estrategias, pero al analizar los tiempos de programa podemos apreciar que son más bajos que los obtenidos con la asignación dinámica y cercanos a los ofrecidos por la E/S asíncrona. Si nos centramos en la asignación dinámica, comprobaremos que al sumar el tiempo de reconstrucción con el de E/S, no siempre obtendremos valores próximos al

tiempo del programa. Esto se debe a la dependencia que existe entre threads a la hora de vaciar y llenar los buffers de E/S, la cual provoca que se generen tiempos muertos en los que hay threads esperando a que otros acaben de procesar el último lote de trabajo que cogieron. Los threads que esperan no pueden seguir trabajando porque el buffer de lectura está vacío, el de escritura está lleno o se dan las dos circunstancias, y hasta que no terminen todos los threads los buffers no se podrán preparar de nuevo. Esta situación puede darse también en WBP, pero las esperas serán mínimas debido a la mayor rapidez de este algoritmo. Un ejemplo sencillo que ilustra este fenómeno es el siguiente. Supongamos buffers de 16 y cuatro threads. El buffer de lectura tendrá capacidad para cuatro lotes de cuatro sinogramas cada uno. Tres threads comienzan su ejecución de manera normal cogiendo un lote, pero el restante no puede porque, pongamos por caso, hay otro programa ocupando el núcleo que le corresponde. Imaginemos que en reconstruir un lote cada thread emplea un segundo y que cuando ha pasado medio, el thread parado arranca. Transcurrido un segundo, a éste aún le queda medio para terminar. Los otros no pueden continuar, ya que no hay más sinogramas en el buffer. Así pues, se genera un tiempo muerto de medio segundo en el que estos threads podrían haber estado trabajando, pero no lo han hecho debido a la dependencia que existe a la hora de llenar y vaciar los buffers.

Debido al mayor tiempo de procesamiento de SIRT, en este algoritmo los tiempos de E/S son menores. Ocurre que cuando nosotros ordenamos una operación de escritura a disco, ésta no se realiza físicamente de forma inmediata, sino que los datos se almacenan en unos buffers de disco[11] que implementa Linux. El sistema operativo vuelca parte del contenido de esos buffers cada cierto tiempo. Como SIRT tiene mucha carga computacional, a Linux le da tiempo de vaciar los buffers entre dos peticiones de escritura y podrá aceptar nuevos datos. WBP, por su rapidez, no deja que Linux haga nuevo espacio, por lo que los buffers se llenan y cuando llega una nueva petición de escritura, se ordena un volcado físico en disco para hacer hueco en el buffer. En resumen, en SIRT existe un solapamiento implícito de la E/S con el procesamiento, mientras que en WBP esto no se da. De ahí que la E/S sea más costosa en SIRT. Podríamos poner un símil para entender mejor la situación. Imaginemos una librería en la que trabajan dos personas, las cuales desean colocar un conjunto de libros en una de las estanterías. Una de ellas será la encargada de ir colocando los libros que la otra le lleva. A la que coloca los libros la llamaremos el dueño de la librería y a la otra, el ayudante. Para que el dueño pueda coger los libros que el ayudante le lleva, ha de tener espacio en las manos. Si no es así, el ayudante tendrá que esperar a que el dueño coloque los libros que ya tiene en su poder antes de darle los últimos que acaba de traer. Es decir, no puede dejárselos en el suelo e ir a por más. Si el conjunto de libros está muy cerca del ayudante (por ejemplo, a un metro), éste irá a por un puñado de libros y volverá con él muy rápido, alcanzándose un punto en el que el dueño se saturará y no podrá aceptar más libros. Por tanto, tendrá que decirle al ayudante que espere. Sin embargo, si la pila de libros se encuentra lejos (por ejemplo, a 100 metros), el ayudante tardará mucho en ir y volver a por ellos, por lo que el dueño tendrá tiempo de sobra para colocar los libros que el ayudante le ha traído mientras éste va a por más. Así, cuando el ayudante regrese, el dueño aceptará sin problemas los nuevos libros, siendo fluida cada entrega y colocación de libros. Si extrapolamos

[11] O caché de disco.

el ejemplo al problema de la reconstrucción, tendremos que el dueño representa al sistema operativo y sus manos son los buffers de disco implementados. La estantería es el disco duro y los libros, los datos a escribir. El ayudante hace el papel de WBP o SIRT, dependiendo de si el conjunto de libros está cerca o lejos, respectivamente. Al igual que cuando los libros están lejos el tiempo de colocarlos se solapa con el de ir a buscarlos, cuando la carga computacional es alta el sistema operativo solapa la E/S con el procesamiento de las imágenes.

Las figuras 5.3 y 5.4 son dos gráficas que muestran el ratio $T_{prog.}/T_{rec.}$ de cada estrategia analizada para WBP y SIRT, respectivamente. Para su construcción se han promediado los ratios de los tres volúmenes analizados. Puesto que el tiempo del programa es aproximadamente la suma del de reconstrucción con el de E/S, el ratio nos permite conocer qué porción del tiempo total pertenece a la E/S. Cuanto más cerca de 1 esté el valor del ratio, más ligera será la E/S. En cambio, cuanto más lejos esté, más pesada resultará. En la Figura 5.3 podemos apreciar que conforme aumentamos el tamaño de los buffers de E/S, el ratio de las estrategias decrece. Este hecho nos indica el efecto beneficioso de los buffers sobre el tiempo de E/S. Solamente hay un caso en el que el ratio sube en vez de bajar, y es en la asignación estática cuando pasamos de 128 a 256. Debido a la cantidad de memoria que precisa esta estrategia, es probable que los buffers no quepan en RAM e intervenga algún mecanismo de *swapping*. Aunque el Q9550 dispone de 8GB, hay que tener en cuenta que esa memoria se comparte entre nuestros buffers y los del sistema operativo. Este mismo fenómeno se aprecia en la gráfica de SIRT (Figura 5.4). Si en la Figura 5.3 comparamos entre estrategias, la que tiene un ratio más bajo es la que utiliza E/S asíncrona, obteniéndose mejores valores cuando se usan dos discos, de forma que con buffers de 128 o 256 el ratio está por debajo de 1,5. Esto significa que la mayor parte de la E/S se está solapando. A continuación tenemos la estrategia con asignación dinámica y luego la que emplea asignación estática. La Figura 5.4 confirma el efecto beneficioso de los buffers, si bien aquí a partir del tamaño 64 no hay ganancia. Como ya hemos explicado, en SIRT hay un solapamiento implícito de la E/S con el procesamiento de las imágenes. Por consiguiente, existe una ayuda adicional que elimina la necesidad de contar con buffers muy grandes. De nuevo, la estrategia que mejores resultados proporciona es la que usa E/S asíncrona, pero esta vez no hay una diferencia relevante entre usar uno o dos discos debido a la mayor carga computacional de SIRT. Se aprecia que el ratio es aproximadamente 1, lo que quiere decir que la porción de E/S sin solapar es mínima y que el tiempo del programa tiende al tiempo de la reconstrucción. Le sigue la que tiene asignación estática, siendo la peor la que utiliza asignación dinámica. En resumen, ambas gráficas nos informan de que para realizar E/S eficiente es preciso el uso de buffers. Tamaños pequeños (16 o 32) deberían ser evitados. En SIRT basta con 64, pero a la luz de los resultados, en WBP podemos seguir aumentando el tamaño, ya que conseguiremos ratios más bajos. La estrategia vencedora es la que emplea E/S asíncrona, siendo muy recomendable el uso de dos discos duros en WBP, mientras que en SIRT es suficiente con uno.

5.3. Análisis del balanceo de carga

En este apartado de nuevo comparamos los tres enfoques con multithreading que explicamos en el Capítulo 4. Cuando se usa la asignación estática de carga,

WBP					
Asignación estática					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	20,50	20,78	20,95	21,02	21,54
$T_{E/S}$	64,94	47,71	38,40	28,50	46,80
$T_{prog.}$	85,72	69,13	59,89	50,72	68,85
Mem.	0,36	0,64	1,20	2,40	4,60
Asignación dinámica					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	22,90	21,41	21,63	21,49	21,00
$T_{E/S}$	91,89	50,61	24,29	21,47	21,69
$T_{prog.}$	115,68	74,04	47,41	44,05	43,44
Mem.	0,14	0,22	0,36	0,64	1,20
E/S asíncrona (1 disco)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	23,01	24,37	24,70	24,56	24,32
$T_{E/S}$	79,25	48,54	23,63	13,14	5,20
$T_{prog.}$	102,82	73,48	48,73	38,23	30,07
Mem.	0,14	0,22	0,36	0,64	1,20
E/S asíncrona (2 discos)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	22,57	24,02	24,58	24,42	24,34
$T_{E/S}$	36,53	22,00	10,83	5,26	3,48
$T_{prog.}$	59,64	46,56	35,95	30,22	28,36
Mem.	0,14	0,22	0,36	0,64	1,20

SIRT					
Asignación estática					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	202,42	202,86	203,67	204,63	206,00
$T_{E/S}$	11,88	8,98	7,80	9,53	48,58
$T_{prog.}$	217,22	216,89	216,16	216,80	255,48
Mem.	0,37	0,65	1,20	2,40	4,60
Asignación dinámica					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	205,33	205,15	204,62	204,99	205,56
$T_{E/S}$	28,44	15,94	14,49	13,36	16,97
$T_{prog.}$	242,32	228,95	227,15	225,40	227,50
Mem.	0,15	0,22	0,37	0,65	1,20
E/S asíncrona (1 disco)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	208,21	209,09	208,90	208,68	209,18
$T_{E/S}$	57,37	6,48	2,89	3,76	4,47
$T_{prog.}$	267,22	216,92	213,07	213,71	214,93
Mem.	0,15	0,22	0,37	0,65	1,20
E/S asíncrona (2 discos)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	208,10	209,61	209,38	209,51	209,66
$T_{E/S}$	53,96	2,37	3,15	3,31	3,54
$T_{prog.}$	263,69	213,25	213,80	214,08	214,49
Mem.	0,15	0,22	0,37	0,65	1,20

Tabla 5.9: Estudio de la E/S del volumen 1024×1024×1024. Se observa que conforme aumentamos el tamaño de los buffers, el tiempo de E/S disminuye. Sin embargo, al usar buffers de 256 en la asignación estática, dicho tiempo crece de forma significativa. Debido a la cantidad de memoria que precisan, es probable que no quepan en RAM e intervenga algún mecanismo de *swapping*. Aunque el Q9550 dispone de 8GB, hay que tener en cuenta que esa memoria se comparte entre nuestros buffers y los del sistema operativo. La estrategia ganadora es la que emplea E/S asíncrona, tanto con uno como con dos discos.

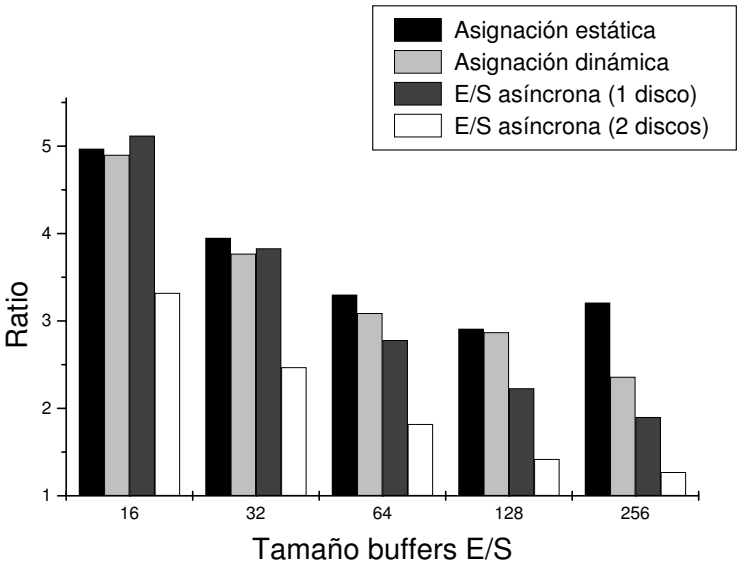

Figura 5.3: Ratio $T_{prog.}/T_{rec.}$ en WBP. Cuanto más cerca de 1 esté el ratio, más ligera será la E/S. Nótese cómo los buffers de E/S contribuyen significativamente a que esto suceda.

WBP					
Asignación estática					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	23,29	22,94	23,07	22,89	23,68
$T_{E/S}$	113,49	80,49	60,00	45,69	47,72
$T_{prog.}$	136,94	103,72	83,36	68,40	72,03
Mem.	0,24	0,44	0,82	1,60	3,10
Asignación dinámica					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	26,31	25,77	25,24	24,80	24,71
$T_{E/S}$	111,15	88,99	78,64	69,76	40,16
$T_{prog.}$	138,59	116,06	105,24	95,51	65,65
Mem.	0,10	0,15	0,24	0,44	0,82
E/S asíncrona (1 disco)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	27,07	28,75	28,02	28,71	28,83
$T_{E/S}$	138,92	114,06	81,18	52,01	36,19
$T_{prog.}$	166,37	143,20	109,59	81,16	65,38
Mem.	0,10	0,15	0,24	0,44	0,82
E/S asíncrona (2 discos)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	27,24	28,80	28,43	29,21	29,44
$T_{E/S}$	87,50	60,91	35,44	20,73	11,69
$T_{prog.}$	115,14	90,10	64,22	50,31	41,51
Mem.	0,10	0,15	0,24	0,44	0,82

SIRT					
Asignación estática					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	230,06	230,21	231,29	232,85	234,89
$T_{E/S}$	20,11	17,60	18,82	20,39	16,78
$T_{prog.}$	253,23	249,94	254,08	256,05	251,72
Mem.	0,26	0,45	0,84	1,60	3,20
Asignación dinámica					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	238,10	237,27	236,27	235,77	235,97
$T_{E/S}$	46,11	35,06	25,68	31,72	22,16
$T_{prog.}$	291,15	278,91	268,31	273,31	262,71
Mem.	0,12	0,16	0,26	0,45	0,84
E/S asíncrona (1 disco)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	234,39	239,75	239,52	239,86	240,15
$T_{E/S}$	86,22	15,74	3,43	2,38	3,12
$T_{prog.}$	321,59	256,38	243,79	243,06	244,10
Mem.	0,12	0,16	0,26	0,45	0,84
E/S asíncrona (2 discos)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	232,91	240,24	240,56	239,74	240,28
$T_{E/S}$	89,69	7,89	2,27	2,44	3,96
$T_{prog.}$	323,52	248,99	243,65	243,01	245,07
Mem.	0,12	0,16	0,26	0,45	0,84

Tabla 5.10: Estudio de la E/S del volumen 2048×256×2048. Se aprecia que al ir aumentando el tamaño de los buffers, el tiempo de E/S disminuye. En WBP el mejor tamaño de buffer es 256, ya que es donde se consiguen los tiempos más bajos. No obstante, en la asignación estática resulta más adecuado un valor de 128, pues se consume menos memoria y el tiempo es algo más bajo. Para todas las estrategias, en SIRT son suficientes buffers de 64.

WBP					
Asignación estática					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	44,00	43,52	43,18	43,87	46,22
$T_{E/S}$	167,95	130,56	105,04	102,17	110,93
$T_{prog.}$	212,73	174,70	147,80	146,40	157,29
Mem.	0,40	0,72	1,40	2,60	5,20
Asignación dinámica					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	49,66	48,25	47,51	47,50	46,97
$T_{E/S}$	165,32	110,13	85,48	79,56	62,57
$T_{prog.}$	218,10	161,75	137,76	129,37	111,24
Mem.	0,16	0,24	0,40	0,72	1,40
E/S asíncrona (1 disco)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	51,54	54,17	54,36	54,19	53,52
$T_{E/S}$	192,81	133,82	78,43	69,72	63,31
$T_{prog.}$	244,97	188,68	133,47	124,57	117,48
Mem.	0,16	0,24	0,40	0,72	1,40
E/S asíncrona (2 discos)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	49,88	53,13	52,98	53,41	54,01
$T_{E/S}$	103,42	70,49	38,17	14,61	12,45
$T_{prog.}$	153,92	124,28	91,83	68,72	67,12
Mem.	0,16	0,24	0,40	0,72	1,40

SIRT					
Asignación estática					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	461,24	461,52	461,99	470,99	464,10
$T_{E/S}$	23,86	22,66	23,91	28,49	68,89
$T_{prog.}$	489,38	488,29	488,71	502,94	535,26
Mem.	0,42	0,74	1,40	2,60	5,20
Asignación dinámica					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	466,03	465,91	464,60	463,79	465,74
$T_{E/S}$	51,89	40,15	35,80	43,26	45,23
$T_{prog.}$	535,05	521,90	515,62	521,06	520,88
Mem.	0,18	0,26	0,42	0,74	1,40
E/S asíncrona (1 disco)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	460,79	470,99	470,67	471,00	470,78
$T_{E/S}$	130,86	7,79	3,07	3,20	4,86
$T_{prog.}$	593,57	480,45	475,41	475,82	477,28
Mem.	0,18	0,26	0,42	0,74	1,40
E/S asíncrona (2 discos)					
Buffer	*16*	*32*	*64*	*128*	*256*
$T_{rec.}$	463,20	470,47	470,16	469,93	471,26
$T_{E/S}$	106,66	3,67	3,20	3,06	4,27
$T_{prog.}$	571,85	475,79	475,00	474,58	477,13
Mem.	0,18	0,26	0,42	0,74	1,40

Tabla 5.11: Estudio de la E/S del volumen 2048×512×2048. En general, al aumentar el tamaño de los buffers, decrece el tiempo de E/S. Un buen tamaño de buffer en WBP es 128, mientras que en SIRT basta con uno de 64. La estrategia ganadora es la que usa E/S asíncrona. Cuando se usan dos discos duros en WBP, las diferencias relativas a los tiempos de E/S con respecto a las demás estrategias son muy relevantes.

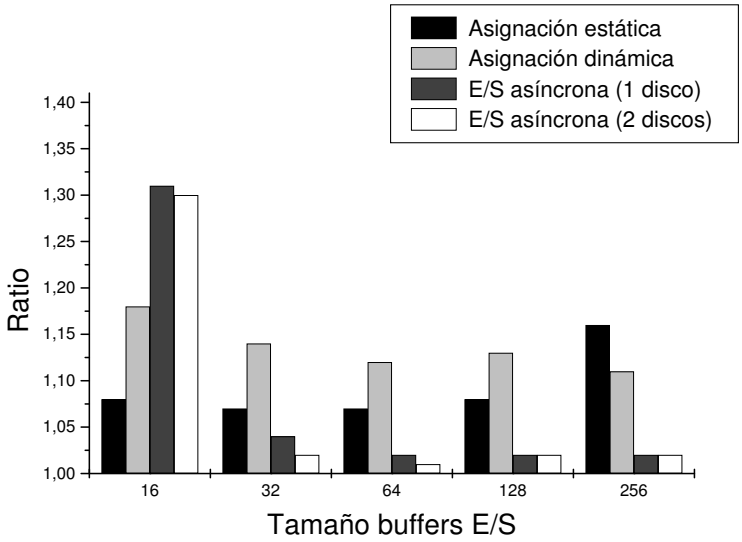

Figura 5.4: Ratio $T_{prog.}/T_{rec.}$ en SIRT. Cuanto más cerca de 1 esté el ratio, más ligera será la E/S. Se puede observar que en SIRT no son necesarios buffers tan grandes como en WBP para obtener buenos ratios.

un thread lento retrasará en gran medida a la reconstrucción debido a que no existe ningún mecanismo que distribuya la carga de forma dinámica en tiempo de ejecución. La asignación dinámica resuelve el problema, pero tiene el inconveniente de que no se puede proceder a la escritura o lectura de los buffers de E/S hasta que todos los threads hayan terminado su reconstrucción en curso. Por tanto, un thread lento seguirá suponiendo un lastre, si bien es cierto que en menor grado. La asignación dinámica de carga con E/S asíncrona le da una solución a este nuevo impedimento, pues elimina la restricción existente para el vaciado o llenado de buffers implementando un mecanismo asíncrono de E/S.

Las tablas 5.12 y 5.13 presentan la reconstrucción con 140 ángulos del volumen 2048×512×2048 usando WBP y SIRT[12], respectivamente. Ambas están divididas en dos tablas más pequeñas, refiriéndose la de la izquierda a una ejecución normal y la de la derecha, a una con sobrecarga. Hemos elegido un volumen con alta carga computacional para reflejar mejor las diferencias entre estrategias. Para los experimentos empleamos el Q9550 y lanzamos los dos algoritmos en sus configuraciones más rápidas, es decir, con las optimizaciones básicas incluidas, el procesamiento vectorial activado y cuatro threads. Cada experimento se repitió cinco veces y se halló la media. Todos los tiempos están en segundos[13]. Las tablas comparan el comportamiento de las diferentes estrategias ante una sobrecarga. En nuestro caso, la sobrecarga consistió en retrasar artificialmente[14] a los threads 0 y 2 con 1 y 2 segundos, respectivamente, cada vez que reconstruían cuatro rebanadas. Como se distingue, los peores resultados

[12]En SIRT seleccionamos sólo 5 iteraciones, ya que son suficientes para examinar los efectos del balanceo.

[13]El tiempo de E/S se atenuó para resaltar el incremento del tiempo de procesamiento a causa de la sobrecarga. Para ello se empleó el fichero de dispositivo /dev/null de Linux a la hora de realizar las escrituras.

[14]Mediante una llamada a la función sleep().

WBP		
Asignación estática		
Thread	Hits	$T_{prog.}$
T0	512	49,95
T1	512	
T2	512	
T3	512	
Asignación dinámica		
Thread	Hits	$T_{prog.}$
T0	512	53,23
T1	512	
T2	512	
T3	512	
E/S asíncrona (1 disco)		
Thread	Hits	$T_{prog.}$
T0	508	54,41
T1	512	
T2	508	
T3	520	

WBP con sobrecarga		
Asignación estática		
Thread	Hits	$T_{prog.}$
T0	512	301,64
T1	512	
T2	512	
T3	512	
Asignación dinámica		
Thread	Hits	$T_{prog.}$
T0	256	91,71
T1	832	
T2	128	
T3	832	
E/S asíncrona (1 disco)		
Thread	Hits	$T_{prog.}$
T0	224	78,53
T1	848	
T2	128	
T3	848	

Tabla 5.12: Balanceo de carga en WBP. *Hits* alude al número de rebanadas reconstruidas. Se puede observar que los tiempos sin sobrecarga son del mismo orden. Sin embargo, al lastrar a los threads 0 y 2, el tiempo del programa crece, viéndose más afectada la estrategia que usa asignación estática. Las otras dos son menos sensibles al balancear automáticamente la carga, pero el mecanismo de E/S asíncrona decanta como ganadora a la estrategia del mismo nombre.

son los obtenidos con la asignación estática de carga, funcionando los otros dos enfoques bastante mejor, pues al no tener fijada de antemano la cantidad de rebanadas que han de reconstruir, los threads van cogiendo trabajo conforme se van quedando libres. Así, los más rápidos procesarán más rebanadas. Entre estos dos últimos enfoques, es menos sensible a la sobrecarga el que utiliza E/S asíncrona debido a la dependencia entre threads rápidos y lentos que existe en el otro enfoque a la hora de llenar y vaciar los buffers de E/S.

Usando la estrategia con E/S asíncrona, en WBP y SIRT se observa que cuando no existe sobrecarga hay threads que reconstruyen unas pocas rebanadas más que otros. Este hecho es normal y también podría producirse en la asignación dinámica. No obstante, con dicho enfoque sucede que antes de proceder al vaciado o llenado de los buffers de E/S, los threads rápidos esperan a los lentos, por lo que estos últimos se reenganchan. En el caso de la asignación dinámica con E/S asíncrona, esta dependencia no existe y, así, los retrasos se acumulan y provocan que los threads más lentos procesen un poco menos de carga. En realidad, este fenómeno es ya el resultado de un balanceo.

5.4. Comparación con GPUs

Las GPUs están revolucionando el campo de la computación de altas prestaciones y, en particular, se han aplicado con gran éxito al problema de la

SIRT		
Asignación estática		
Thread	Hits	$T_{prog.}$
T0	512	472,09
T1	512	
T2	512	
T3	512	
Asignación dinámica		
Thread	Hits	$T_{prog.}$
T0	512	477,49
T1	512	
T2	512	
T3	512	
E/S asíncrona (1 disco)		
Thread	Hits	$T_{prog.}$
T0	516	473,28
T1	504	
T2	504	
T3	524	

SIRT con sobrecarga		
Asignación estática		
Thread	Hits	$T_{prog.}$
T0	512	712,76
T1	512	
T2	512	
T3	512	
Asignación dinámica		
Thread	Hits	$T_{prog.}$
T0	448	572,02
T1	640	
T2	384	
T3	576	
E/S asíncrona (1 disco)		
Thread	Hits	$T_{prog.}$
T0	464	543,57
T1	600	
T2	384	
T3	600	

Tabla 5.13: Balanceo de carga en SIRT. *Hits* alude al número de rebanadas reconstruidas. Se aprecia que los tiempos sin sobrecarga son del mismo orden. Sin embargo, al lastrar a los threads 0 y 2, el tiempo del programa crece, viéndose más afectada la estrategia que usa asignación estática. Las otras son menos sensibles al balancear automáticamente la carga, pero el mecanismo de E/S asíncrona decanta como ganadora a la estrategia del mismo nombre.

reconstrucción 3D en tomografía electrónica [21, 22, 83, 88]. En este apartado comparamos los tiempos de procesamiento obtenidos por nuestros algoritmos WBP y SIRT con los ofrecidos por las implementaciones más recientes de estos métodos en GPUs. Las características de las GPUs con las que comparamos se muestran en la Tabla 5.14. Todas ellas son tarjetas de NVIDIA. Aquellas cuyo nombre comienza con una 'C' son teslas, es decir, tarjetas gráficas pensadas exclusivamente para computación de propósito general que no pueden mostrar gráficos en un monitor. Por comparación con esta tabla, el E5405 es un procesador que Intel introdujo en el último trimestre del 2007 (Q4'07), mientras que el Q9550 data del primer trimestre del año 2008 (Q1'08).

En la Tabla 5.15 comparamos con una implementación de backprojection para GPUs presentada en [83] que convierte el problema de la reconstrucción en un producto matriz dispersa-vector implementado eficientemente en la GPU. En este trabajo la tarjeta para los experimentos fue una GTX 295 y se evaluaron diversos conjuntos de datos, de los que cuales hemos seleccionado los cuatro más representativos para analizarlos aquí. Los tamaños de las imágenes de proyección de tres de ellos fueron 1024×1024 píxeles, teniendo uno 60 imágenes, otro 90 y el último 120. Los tres dieron lugar a reconstrucciones de 1024×1024×1024 vóxeles. El cuarto conjunto estaba formado por 61 imágenes con dimensiones 2048×2048 (VV2K), siendo el volumen generado 2048×960×2048. Todos los tiempos de la tabla están en segundos y sólo se ha tenido en cuenta el tiempo de reconstrucción. Nosotros ejecutamos nuestro algoritmo backprojection con

	C1060	GTX 280	GTX 285	GTX 295	C2050
Lanzamiento	Q2'08	Q2'08	Q1'09	Q1'09	Q4'09
Rendimiento (GFlops)	933	933	1062	1788	1030
Ancho banda (GB/s)	102	141	159	224	144
Frecuencia (GHz)	1.3	1.3	1.4	1.21	1.15
Frec. mem. (MHz)	800	1107	1242	999	1500
Memoria (GB)	4	1	2	896×2	2.6
Núcleos	240	240	240	240×2	448

Tabla 5.14: Especificaciones de las GPUs.

	60	*90*	*120*	*VV2K*
GTX 295 [83]	6,19	7,95	9,54	26,05
E5405 (8T)	7,96	10,78	12,11	32,82
Q9550 (4T)	10,83	14,46	18,24	42,75

Tabla 5.15: CPU vs. GPU (backprojection). Se compara nuestra implementación de backprojection con la publicada en [83]. Los tiempos en la GPU son mejores que en el Q9550, pero al emplear el E5405 las diferencias son pequeñas.

la configuración más rápida, esto es, con las optimizaciones básicas, el procesamiento vectorial activado, y cuatro y ocho núcleos, en el Q9550 y en el E5405, respectivamente. Cada una de nuestras ejecuciones fue repetida cinco veces, y se calculó la media.

En la Tabla 5.16 comparamos nuestro SIRT con dos implementaciones distintas de este algoritmo en GPUs. Una es la aparecida en [88], donde la tarjeta empleada fue una GTX 280. La otra se deriva de la implementación matricial de backprojection que hemos comentado antes, si bien este trabajo [84] aún no ha sido publicado[15]. Las tarjetas que se utilizaron en dicho trabajo fueron la GTX 285, la C1060 y la C2050. Los conjuntos de datos usados consistieron en 61 imágenes de proyección de tamaño 356×506 (*Dataset A*), 712×1012 (*Dataset B*) y 1424×2024 (*Dataset C*). Estos conjuntos de datos dieron lugar a volúmenes de 356×148×506, 712×296×1012 y 1424×591×2024 vóxeles, respectivamente. Únicamente se usó una iteración de SIRT. Todos los tiempos están en segundos y sólo se ha tenido en cuenta el tiempo de reconstrucción. Nosotros lanzamos nuestro algoritmo SIRT con la configuración más rápida, repitiendo cada experimento cinco veces y calculando la media después.

La implementación de backprojection de [83] es muy eficiente, de hecho la mejor para GPUs hasta el momento. Los tiempos que proporciona son mejores

[15]Se agradece enormemente a Francisco Vázquez López y a Ester Martín Garzón la cesión de estos resultados sin publicar para poder realizar esta comparación.

	Dataset A	*Dataset B*	*Dataset C*
GTX 280 [88]	2,10	10,66	58,47
C1060 [84]	0,53	4,80	51,28
GTX 285 [84]	0,46	3,90	42,32
C2050 [84]	0,57	3,70	37,91
Q9550 (4T)	0,48	4,03	32,17
E5405 (8T)	0,35	2,94	24,55

Tabla 5.16: CPU vs. GPU (SIRT 1 iteración). Los tiempos del Q9550 son bastante mejores que los de la GTX 280 y del mismo orden que los conseguidos por la otra implementación en las demás tarjetas. El E5405 resulta el vencedor absoluto de la comparativa.

que los nuestros, pero es destacable que las distancias no son grandes, especialmente cuando utilizamos el E5405. En el caso de SIRT, nuestra implementación necesita menos tiempo que las más eficientes para GPUs de la actualidad, lo cual se manifiesta claramente al reconstruir el volumen de mayor tamaño.

Capítulo 6

Conclusiones y trabajo futuro

La mayor parte de los métodos de reconstrucción 3D rápida emplean clusters de computadores o, más recientemente, tarjetas gráficas (GPUs), las cuales han revolucionado el campo de la tomografía por los excelentes factores de aceleración que proporcionan —hasta 100x— [22, 88] y por su insuperable relación rendimiento/coste. Por su rapidez, las GPUs son muy apropiadas para entornos de tomografía en tiempo real. Desde el punto de vista del desarrollador, una desventaja de las GPUs es que su programación aún no está completamente estandarizada, si bien poco a poco se están dando pasos hacia ello. Otra desventaja, que también atañe al usuario, es que existen marcas o modelos de tarjetas que todavía no soportan computación de propósito general. Así pues, el principal impedimento de estos dispositivos es que son componentes hardware *especiales* con los que ha de contar el computador, lo cual puede suponer un problema de cara a la distribución del software. Además, una GPU no es un procesador tan versátil como una CPU, por lo que el problema a resolver ha de poseer ciertas características computacionales que lo habiliten como candidato para ser portado a la GPU.

Del párrafo anterior podríamos concluir que sin un hardware especial —cluster o GPU— parece imposible calcular rápidamente reconstrucciones 3D. No obstante, los modernos procesadores multicore ponen a disposición del usuario una gran potencia de cálculo. A finales de la primera década del 2000, surgió una tendencia hacia la implementación eficiente de programas para sacar partido de esta potencia [39, 56, 60, 89]. Sin embargo, tras el repentino auge de las GPUs, esta línea de investigación fue abandonada por la comunidad científica. Nosotros hemos retomado dicha línea y hemos diseñado un procedimiento de reconstrucción para explotar al máximo los computadores actuales y llegar al nivel de rendimiento de las GPUs, superándolo incluso a tenor de los resultados expuestos en el Capítulo 5. La novedad con respecto a los trabajos previos [39, 56, 60, 89] es que estos estaban enfocados a reconstrucción 2D y no a reconstrucción 3D [89], o se centraban en métodos de reconstrucción muy sencillos sin incluir los métodos iterativos más sofisticados [39, 60], o no aprovechaban los múltiples núcleos de un procesador ni aplicaban optimización de código [56]. Últimamente, la comunidad científica también está comparando el rendimiento

de los computadores multicore con el mostrado por las GPUs para aplicaciones científicas generales, y las conclusiones son similares a las nuestras. Esto es, si se programan eficientemente, la distancia entre los computadores multicore y las GPUs en términos de rendimiento no es tan grande, y puede ser incluso similar [17, 64].

El procedimiento desarrollado en esta tesis utiliza la optimización de código para realizar cambios a nivel algorítmico en los métodos de reconstrucción. Dichos cambios aluden al mecanismo de caché, a la simetría y a las regiones de interés. También se lleva a cabo la sustitución de ciertas operaciones por otras menos costosas. Aquí incluimos la librería FFTW y las optimizaciones generales. El objetivo de todas estas transformaciones es producir algoritmos de reconstrucción secuenciales muy eficientes. Sobre ellos se asientan las versiones paralelas que hacen uso de las características especiales que brindan los modernos procesadores multicore, es decir, el procesamiento vectorial y la existencia de varios núcleos de computación. Aparte, el acceso a disco también ha sido optimizado para que la E/S se solape, en la medida de lo posible, con la reconstrucción de las imágenes. A la luz de los tiempos de ejecución mostrados en el Capítulo 5, con nuestros algoritmos WBP y SIRT es posible generar reconstrucciones tomográficas en computadores convencionales con una rapidez comparable a la de una GPU. Así pues, podrán ser empleados en entornos de tomografía de tiempo real, ya que permitirán al usuario obtener reconstrucciones 3D típicas velozmente. La gran ventaja de nuestro procedimiento es que no precisa de ningún hardware particular, lo cual facilitará su distribución y su uso en laboratorios científicos de tomografía donde los investigadores que allí trabajan no son expertos en administración de sistemas informáticos. Además, merece ser destacado el ahorro económico que supone no tener que comprar potentes tarjetas gráficas ni clusters; sólo es necesario el ordenador de sobremesa o el portátil con el que trabajemos a diario.

Los capítulos 2, 3 y 4 se han dedicado a exponer y detallar las distintas optimizaciones aplicadas a los métodos de reconstrucción WBP y SIRT. El 2 se centraba en las optimizaciones básicas. Atendiendo a los resultados obtenidos en el Capítulo 5, el haber invertido nuestro tiempo en analizar el código original de dichos métodos para buscar posibles modificaciones a nivel algorítmico ha resultado ser muy beneficioso, pues hemos conseguido un speedup por encima de 6x. El Capítulo 3 giraba alrededor del procesamiento vectorial. Aquí se explicaba cómo era posible llevar a cabo la reconstrucción de un volumen procesando las rebanadas de cuatro en cuatro gracias a las instrucciones SSE. Solamente con el uso de estas instrucciones logramos un speedup en torno a 3.5x. Al unir las optimizaciones básicas con el procesamiento vectorial, el speedup global subió hasta 20x. Por último, el Capítulo 4 trataba el aprovechamiento de los múltiples núcleos de un procesador multicore y la mejora de la E/S. De forma resumida, las diferentes rebanadas que componían un volumen 3D se distribuían entre tantos threads de procesamiento como núcleos hubiera disponibles. La distribución de las rebanadas podía ser estática o dinámica. El speedup alcanzado mediante este enfoque resultó ser aproximadamente lineal con el número de núcleos, y así el speedup global creció hasta el entorno de 40x, 80x y 160x, con 2, 4 y 8 núcleos respectivamente.

Un hecho que se deriva de las optimizaciones aplicadas y que queremos resaltar es el uso adecuado de la jerarquía de memoria en todo momento. A tal fin, la reconstrucción de un volumen 3D se divide en partes con tamaños acordes

al nivel de la jerarquía en el que nos encontremos para que su resolución sea eficiente. Así pues, el volumen 3D, que en disco podría ocupar varios gigabytes (e.g. 8GB), se reconstruye por fragmentos (i.e. un subconjunto de rebanadas) en la RAM gracias al buffer de escritura (e.g. 256MB). A su vez, las rebanadas en el buffer son agrupadas en lotes de cuatro y asignadas a los threads, los cuales fraccionan cada lote en pequeños bloques (e.g. 512KB) para que quepan en caché. Por último, los bloques son reconstruidos en grupos de cuatro píxeles aprovechando la capacidad de los registros vectoriales (16 bytes). Hay que tener en cuenta que la reconstrucción de un volumen 3D se hace a partir de un conjunto de sinogramas, al que también se aplica la técnica anterior. La única diferencia es que este conjunto (e.g. 1GB) no se escribe en disco, sino que se lee de él poco a poco mediante el buffer de lectura (e.g. 64MB). De forma breve, los sinogramas en el buffer son combinados en lotes de cuatro y asignados a los threads, los cuales dividen dichos lotes en pequeños bloques (e.g. 128KB) que entran en caché. Estos bloques se procesan de cuatro en cuatro píxeles (16 bytes), realizándose operaciones entre píxeles de las rebanadas y píxeles de los sinogramas.

Aparte de mejorar los tiempos de reconstrucción, en esta tesis también se han cuidado otros aspectos. Nos estamos refiriendo a la optimización del acceso a disco y al balanceo de carga entre threads. Cuando se trabaja con volúmenes grandes que ocupan varios gigabytes, el tiempo de E/S puede ser incluso mayor que el de reconstrucción, por lo que se hace imprescindible optimizar también el acceso a disco. En nuestro caso hemos implementado un mecanismo que solapa la E/S con el procesamiento de las imágenes y usa buffers de E/S para minimizar el número de accesos. Además, es posible utilizar dos discos duros para que lectura y escritura corran en paralelo. Por otro lado, se debe tener en cuenta que en un computador multicore el usuario puede estar ejecutando otros programas a la misma vez que el nuestro, con lo cual algunos núcleos podrían estar sobrecargados. Es por ello que diseñamos una estrategia con asignación dinámica de carga que se adapta a las circunstancias del sistema. Así, los threads que se ejecuten en núcleos sobrecargados reconstruirán menos rebanadas.

Una conclusión importante que se extrae de este trabajo es que, cuando se trata de optimizar una aplicación, siempre es recomendable contar con algún computador cuyas prestaciones sean modestas, pues las ineficiencias en el código tenderán a ocultarse en computadores potentes, mientras que se magnificarán en aquellos con recursos limitados. Por ejemplo, un procesador que disponga de una gran caché será más robusto frente a los fallos de caché provocados por un acceso inadecuado a la jerarquía de memoria que otro con una caché más reducida. De hecho, en el estudio realizado en el Apéndice B, podemos comprobar que las tablas de tiempos del Xeon son mucho más heterogéneas que las del Core 2 debido a su pequeña caché. En reconstrucciones donde el Core 2 no sufre sea cual sea la configuración elegida para el mecanismo de caché, el Xeon se ve más o menos perjudicado según ésta. Debemos asumir que las características que tendrán los ordenadores de los usuarios finales de nuestro programa serán muy variadas, y no sería deseable que ineficiencias no detectadas se presentaran una vez el software ha salido del laboratorio.

Las investigaciones llevadas a cabo en esta tesis han sido publicadas en un total de ocho artículos [4, 5, 6, 7, 8, 9, 10, 11]. En [5, 6] se presentó la vectorización del algoritmo de reconstrucción WBP. En [7, 9] se demostró el efecto beneficioso de las optimizaciones básicas sobre WBP. En [10] se extendieron

las investigaciones anteriores al algoritmo SIRT. [8] exploró la viabilidad de los métodos iterativos de reconstrucción en entornos de tomografía de tiempo real. En [4, 11] se aplicó el multithreading a WBP y a SIRT, constatándose que ambos algoritmos eran ya utilizables para tiempo real. Adicionalmente, en [4] se publicó el software fruto de esta tesis, cuyo nombre es Tomo3D y puede ser descargado desde http://www.cnb.csic.es/~jjfernandez/tomo3d/. La aplicación es de uso público, va acompañada de un manual y está disponible tanto para Linux como para Mac OS X. El programa está compilado estáticamente, por lo que no necesita de ninguna librería externa y puede ser ejecutado al descomprimir el paquete.

En esta tesis se han exprimido al máximo las capacidades y características de los actuales procesadores multicore. Sin embargo, se pueden aprovechar otras tecnologías presentes también en los computadores estándar para acelerar aún más el proceso de reconstrucción. Un ejemplo es la GPU. Ya que tan buenos resultados está dando en tomografía electrónica, podría usarse para realizar computación híbrida CPU-GPU [73, 80], de manera que la GPU interviniera conjuntamente con la CPU en la reconstrucción del volumen 3D. Se podría crear un hilo de ejecución adicional que corriese en la GPU, permitiendo así que este dispositivo funcionase como un procesador adicional que ayuda en la reconstrucción. Por otro lado, Intel ha anunciado que en futuros microprocesadores añadirá un nuevo conjunto de instrucciones vectoriales llamado AVX (*Advanced Vector Extensions*) [37]. La longitud de los registros vectoriales pasará de 128 a 256 bits, lo que posibilitaría el procesamiento de ocho rebanadas al mismo tiempo en lugar de las cuatro actuales. Es también muy probable que la cantidad de núcleos incluidos en los procesadores se vea aumentada. Esto beneficiará directamente a nuestro programa, pues es capaz de detectar cuántos hay y sacar partido de ellos. Además, otras mejoras como memorias cachés más grandes, computadores más rápidos y discos duros más eficientes favorecerán a nuestro programa automáticamente sin necesidad de tocar el código fuente.

Otra línea de trabajo futuro podría ser la aplicación de las optimizaciones analizadas en esta tesis a otros algoritmos de reconstrucción, como por ejemplo ART (*Algebraic Reconstruction Technique*). También otras operaciones costosas en tomografía electrónica —e.g. *denoising*— u otros problemas computacionalmente complejos de otros campos científicos podrían beneficiarse de dichas optimizaciones.

Por último, podríamos adaptar nuestra aplicación para que corriese en clusters de computadores multicore, de forma que cada computador del cluster ejecutase un algoritmo optimizado.

Apéndice A

Publicaciones derivadas de esta tesis

Este apéndice contiene una lista —extraída de la bibliografía— de las publicaciones derivadas a raíz de la investigación llevada a cabo en esta tesis. Los artículos citados están ordenados según hayan sido publicados en revistas internacionales, en congresos internacionales o en congresos nacionales. Dentro de cada grupo, el orden seguido es el del año de publicación (primero los artículos más antiguos).

A.1. Artículos en revistas internacionales

[10] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Vectorization with SIMD extensions speeds up reconstruction in electron tomography. *Journal of Structural Biology*, 170:570–575, 2010.

☞ El índice de impacto de esta revista según el JCR 2009 es a dos años 3,673 y a cinco, 3,925. Esta revista pertenece a la categoría *Biophysics*, ocupando la posición 19/74 (primer tercil).

[4] J.I. Agulleiro and J.J. Fernández. Fast tomographic reconstruction on multicore computers. *Bioinformatics*, 2011. En prensa (doi: 10.1093/bioinformatics/btq692).

☞ El índice de impacto de esta revista según el JCR 2009 es a dos años 4,926 y a cinco, 6,271. Esta revista pertenece a la categoría *Mathematical and Computational Biology*, ocupando la posición 2/29 (primer tercil).

A.2. Artículos en congresos internacionales

[6] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Fast tomographic reconstruction with vectorized backprojection. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 387–390, 2008. Fue seleccionado para ser presentado dentro de la *Best papers session*.

[9] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Ultra-fast tomographic reconstruction with a highly optimized weighted backprojection algorithm. In *18th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2010)*, pages 281–285, 2010.

[11] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Multi-core desktop processors make possible real-time electron tomography. In *19th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2011)*, 2011. En prensa.

☞ El *Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)* es un congreso que figura en la categoría C del ranking CORE (*COmputing Research and Education*).

A.3. Artículos en congresos nacionales

[5] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Fast tomographic reconstruction with vectorized backprojection. In *Actas de las XVIII Jornadas de Paralelismo*, pages 579–586, 2007.

[7] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Ultra-fast tomographic reconstruction with a highly optimized back-projection algorithm. In *Actas de las XX Jornadas de Paralelismo*, pages 75–80, 2009.

[8] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Towards real time iterative tomographic reconstruction. In *I Simposio en Computación Empotrada*, pages 161–168, 2010.

A.4. Capítulos de libro

[36] J.J. Fernández, J.I. Agulleiro, J.R. Bilbao-Castro, A. Martínez, I. García, F.J. Chichón, J. Martín-Benito, and J.L. Carrascosa. Image processing in electron tomography. In A. Méndez-Vilas and J. Díaz, editors, *Microscopy: science, technology, applications and education*, Microscopy book series. Editorial Formatex, 2011. En prensa.

Apéndice B

Los parámetros R y P del mecanismo de caché

Este apéndice comprende un estudio de los parámetros R y P referentes al mecanismo de caché implementado en los algoritmos de reconstrucción WBP y SIRT. Dicho mecanismo fue explicado en el Capítulo 2, y decíamos que existían tres formas de configurarlo, siendo una de ellas a través de los parámetros R y P, cuya función era establecer los tamaños de los bloques en los que se dividían rebanadas y sinogramas para su procesamiento. Realizar la configuración usando estos parámetros es difícil, si bien nosotros apuntábamos que valores pequeños que fueran potencias de 2 —2, 4, 8, 16— daban buenos resultados. El estudio que hacemos en este apéndice justifica esta afirmación.

En nuestro estudio nos hemos centrado en la memoria caché de nivel 2 (L2), pues la de primer nivel (L1) suele ser bastante pequeña y la de tercer nivel (L3) no se incluye en todos los procesadores. De hecho, ninguno de los utilizados aquí disponía de una caché L3. Para las pruebas se eligieron dos computadores distintos. El primero de ellos era un Intel Xeon basado en Pentium 4 con 2GB de RAM y una pequeña caché de 512KB. El segundo era un Intel Core 2 Quad a 2.83GHz con 8GB de RAM y una gran caché de 12MB. Los tamaños de caché de estos procesadores son especialmente adecuados, pues nos permitirán comprobar si nuestro mecanismo escala y se adapta bien a diferentes tamaños. El sistema operativo que había instalado era Linux. Se ejecutó el algoritmo WBP con todas las optimizaciones básicas añadidas (ver Capítulo 2), las instrucciones SSE activadas (ver Capítulo 3) y un solo thread de procesamiento. Los tiempos de E/S no se tuvieron en cuenta. Puesto que SIRT está basado en backprojection, todo lo comentado aquí es aplicable también a este otro algoritmo.

Cada conjunto de datos seleccionado se obtuvo de una mitocondria sintética [34] y consistió en 180 imágenes de proyección tomadas en el rango de ángulos $[-90°, +89°]$ con incrementos de $1°$. Los tamaños de los conjuntos de datos fueron 128, 192, 256, 384, 512, 768 y 1024. Así, el de tamaño 128 estaba formado por 180 imágenes de proyección de 128×128 píxeles, dando como resultado un volumen 3D de $128\times128\times128$ vóxeles, lo cual significa que se tienen 128 rebanadas de tamaño 128×128.

Para cada volumen a reconstruir R tomó todos los divisores de la altura de la rebanada, aunque no se incluyó el propio valor de la altura como divisor debido

a la simetría (ver Capítulo 2). Para cada uno de esos R se tomaron todos los P divisores de 180. Cada ejecución fue repetida 5 veces. Así, para la reconstrucción de 128×128×128, R adquirió los valores 1, 2, 4, 8, 16, 32 y 64, y P fue 1, 2, 3, 4, 5, 6, 9, 10, 12, 15, 18, 20, 30, 36, 45, 60, 90 y 180. Esto nos da un total de $7 \times 18 \times 5 = 630$ ejecuciones solamente para el volumen más pequeño.

La Tabla B1 muestra los tiempos para el Xeon, mientras que la Tabla B2 se corresponde con el Core 2. Todos los tiempos están expresados en segundos. Para cada tamaño se ha resaltado en negrita el mejor tiempo de cada fila y en rojo, el mejor tiempo absoluto. En el pie de cada tabla se han añadido una serie de comentarios para hacer más sencilla la interpretación de los datos presentados. Si observamos las tablas, comprobaremos que siempre hay tiempos óptimos —no tienen por qué ser los mejores— en las coordenadas (R, P) con R y P siendo pequeñas potencias de dos. Si queremos comparar dichos tiempos con los que se obtendrían con la forma original de acceder a sinogramas y rebanadas (ver Capítulo 2), basta con que nos fijemos en la celda inferior izquierda de cada tamaño, es decir, cuando P vale 1 y R toma el valor máximo permitido.

En general, podemos afirmar que el mecanismo de caché es indispensable para que los algoritmos de reconstrucción trabajen de modo eficiente. Por ejemplo, si nos centramos en el tamaño 1024, WBP requirió 597,57 segundos en el Xeon empleando la forma original de acceso a memoria, mientras que con $R = 8$ y $P = 2$ solamente precisó de 222,48 segundos, tiempo muy bueno si tenemos en cuenta que el mejor para este tamaño fue 221,45. El mismo efecto puede ser observado en el Core 2, donde WBP tardó 309,72 segundos usando el acceso original, y al ejecutarse con $R = 8$ y $P = 2$ únicamente consumió 102,75 segundos, tiempo también óptimo si nos fijamos en que el mejor fue 102,46. Este hecho nos confirma que, aunque dispongamos de un moderno procesador con una gran caché, sigue siendo necesaria la inclusión de algún procedimiento que la aproveche adecuadamente. En ambos casos, nuestro método de *blocking* hizo unas tres veces más rápido a WBP. Además, el asignar pequeñas potencias de dos a R y a P resultó ser una decisión acertada, ya que los tiempos conseguidos así están muy próximos a los mejores absolutos.

Por otro lado, el mecanismo de caché se hace tanto más útil cuanto menor sea la cantidad de caché instalada en el procesador y mayores dimensiones tengan las imágenes a reconstruir. Por ejemplo, si nos fijamos en el tamaño 512, en el Core 2 apenas hay variación sean cuales sean los valores asignados a R y P. Sin embargo, en el Xeon tenemos que ser más cuidadosos, pues la obtención de tiempos óptimos depende completamente de una correcta configuración de R y P. Aquí, un ejemplo de buena configuración sería $R = 4$ y $P = 2$, de nuevo potencias de dos pequeñas. Como hemos visto en los ejemplos del párrafo anterior, cuando las imágenes son grandes, el mecanismo de caché se hace imprescindible.

1/2		P 1	P 2	P 3	P 4	P 5	P 6	P 9	P 10	P 12	P 15	P 18	P 20	P 30	P 36	P 45	P 60	P 90	P 180	Ganador
128	R 1	0,48	0,48	0,47	0,47	0,47	0,46	0,46	0,44	**0,44**	0,46	0,47	0,47	0,47	0,47	0,48	0,47	0,47	0,47	P 12
	R 2	0,45	0,46	0,46	0,46	0,46	0,46	0,47	0,46	0,47	0,46	0,47	0,47	0,47	0,46	0,46	**0,45**	0,47	0,49	P 60
	R 4	0,45	0,44	0,45	0,44	0,44	0,44	0,44	0,44	0,44	0,44	0,44	0,44	0,44	0,44	0,44	0,44	**0,43**	0,46	P 90
	R 8	0,43	0,43	0,44	0,44	0,44	0,43	0,44	0,44	0,44	0,44	0,43	0,43	0,43	0,43	0,43	0,43	**0,43**	0,45	P 90
	R 16	**0,43**	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,48	P 1
	R 32	0,43	0,43	0,43	0,43	0,43	0,43	**0,43**	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,43	0,46	P 9
	R 64	0,43	0,43	0,43	0,43	0,42	0,42	0,42	0,43	0,42	0,42	0,42	**0,42**	0,42	0,42	0,42	0,42	0,42	0,49	P 20
192	R 1	1,87	1,87	1,86	1,87	1,86	1,86	1,86	1,86	1,87	1,87	1,86	1,86	1,86	1,86	1,86	1,86	1,87	**1,85**	P 180
	R 2	1,64	1,65	1,64	1,64	1,64	1,63	1,64	1,63	1,64	1,64	**1,63**	1,63	1,63	1,63	1,63	1,63	1,66	1,86	P 18
	R 3	1,56	1,57	1,57	1,56	1,56	1,56	1,56	1,56	1,56	1,56	1,56	1,56	1,56	1,56	1,56	**1,55**	1,58	1,85	P 60
	R 4	1,52	1,52	1,52	1,52	1,52	1,52	1,52	1,52	1,52	1,52	1,52	1,52	1,52	1,52	1,52	**1,51**	1,53	1,82	P 60
	R 6	**1,47**	1,48	1,48	1,48	1,48	1,48	1,48	1,48	1,48	1,48	1,49	1,48	1,48	1,48	1,48	1,48	1,50	1,83	P 1
	R 8	**1,45**	1,46	1,46	1,46	1,46	1,46	1,46	1,46	1,47	1,47	1,47	1,47	1,47	1,47	1,47	1,46	1,48	1,84	P 1
	R 12	**1,43**	1,44	1,44	1,44	1,44	1,44	1,44	1,44	1,44	1,44	1,45	1,45	1,45	1,45	1,45	1,45	1,46	1,86	P 1
	R 16	**1,42**	1,43	1,43	1,43	1,43	1,43	1,43	1,43	1,43	1,43	1,43	1,43	1,44	1,44	1,44	1,44	1,46	1,86	P 1
	R 24	**1,41**	1,42	1,42	1,41	1,42	1,42	1,42	1,42	1,42	1,42	1,42	1,42	1,43	1,43	1,43	1,43	1,45	1,86	P 1
	R 32	1,44	1,44	1,42	1,42	1,42	1,42	1,42	1,42	**1,41**	1,42	1,42	1,42	1,42	1,42	1,42	1,42	1,44	1,88	P 12
	R 48	1,55	1,54	1,48	1,46	1,46	1,45	1,43	1,43	1,42	1,43	1,43	1,42	1,42	1,42	1,42	**1,41**	1,43	1,89	P 60
	R 96	3,21	2,25	1,95	1,82	1,72	1,68	1,57	1,55	1,52	1,49	1,47	1,46	1,43	1,42	1,41	**1,40**	1,42	1,89	P 60
256	R 1	5,61	5,57	5,57	**5,55**	5,57	5,56	5,56	5,55	5,55	5,59	5,61	5,65	5,65	5,66	5,64	5,62	5,61	5,62	P 4
	R 2	**4,43**	4,46	4,47	4,46	4,47	4,46	4,48	4,48	4,47	4,47	4,47	4,46	4,47	4,46	4,46	4,48	4,66	5,54	P 1
	R 4	**3,83**	3,84	3,85	3,85	3,85	3,86	3,86	3,86	3,87	3,88	3,88	3,88	3,88	3,89	3,90	3,99	4,15	5,58	P 1
	R 8	**3,54**	3,56	3,57	3,57	3,57	3,58	3,58	3,59	3,59	3,60	3,60	3,61	3,61	3,62	3,62	3,68	3,83	5,57	P 1
	R 16	**3,41**	3,45	3,44	3,43	3,44	3,43	3,44	3,44	3,45	3,46	3,46	3,46	3,47	3,48	3,53	3,55	3,81	5,52	P 1
	R 32	3,76	3,57	3,57	3,48	3,48	3,45	3,43	3,43	3,43	3,42	3,42	3,42	3,42	3,42	**3,41**	3,42	3,71	5,62	P 45
	R 64	6,36	5,01	4,39	4,13	3,98	3,87	3,68	3,65	3,60	3,54	3,50	3,48	3,42	3,40	**3,40**	3,45	3,72	5,58	P 45
	R 128	9,34	6,25	5,21	4,72	4,41	4,21	3,88	3,81	3,71	3,61	3,55	3,51	3,42	3,39	3,38	**3,37**	3,65	5,51	P 60
384	R 1	**17,29**	17,34	17,36	17,35	17,36	17,36	17,37	17,36	17,36	17,35	17,36	17,35	17,36	17,34	17,36	17,35	17,36	17,36	P 1
	R 2	**14,04**	14,09	14,09	14,09	14,10	14,09	14,10	14,10	14,11	14,11	14,13	14,14	14,21	14,27	14,40	15,01	15,62	17,35	P 1
	R 3	**12,95**	13,00	13,00	13,01	13,01	13,02	13,03	13,04	13,04	13,07	13,08	13,09	13,15	13,22	13,41	13,81	15,02	17,36	P 1
	R 4	**12,40**	12,45	12,46	12,47	12,47	12,48	12,49	12,50	12,51	12,54	12,55	12,57	12,63	12,70	12,83	13,39	14,71	17,32	P 1
	R 6	**11,88**	11,93	11,93	11,94	11,95	11,95	11,97	11,98	11,99	12,02	12,04	12,06	12,13	12,19	12,31	12,89	14,41	17,33	P 1
	R 8	**11,62**	11,66	11,67	11,68	11,68	11,69	11,71	11,72	11,74	11,77	11,80	11,83	11,92	11,95	12,14	12,47	14,41	17,30	P 1
	R 12	**11,37**	11,42	11,43	11,43	11,44	11,45	11,48	11,49	11,51	11,54	11,58	11,60	11,69	11,72	11,94	12,31	14,20	17,32	P 1
	R 16	11,57	11,52	11,42	11,44	11,43	**11,41**	11,46	11,43	11,45	11,50	11,54	11,55	11,60	11,63	11,75	12,30	14,14	17,30	P 6
	R 24	12,77	11,80	11,79	11,63	11,57	11,64	11,57	11,62	11,59	11,58	11,60	11,60	**11,55**	11,59	11,70	12,62	13,99	17,31	P 30
	R 32	15,68	13,41	12,90	12,41	12,34	12,17	11,91	11,88	11,81	11,74	11,69	11,64	11,54	11,54	**11,50**	12,45	13,95	17,26	P 45
	R 48	25,06	17,90	15,76	14,62	13,83	13,37	12,62	12,44	12,24	11,99	11,84	11,77	11,52	11,51	**11,46**	12,25	13,97	17,24	P 45
	R 64	29,33	20,06	17,01	15,52	14,58	13,96	12,92	12,71	12,40	12,10	11,90	11,80	11,52	**11,43**	11,44	12,18	13,90	17,26	P 36
	R 96	31,70	21,12	17,65	15,91	14,88	14,19	13,04	12,82	12,48	12,15	11,92	11,81	11,50	**11,40**	11,42	12,06	13,84	17,28	P 36
	R 192	31,87	21,25	17,76	16,02	14,97	14,27	13,10	12,86	12,51	12,17	11,93	11,83	11,49	11,43	**11,38**	12,15	13,74	17,28	P 45

Tabla B1 (a): Tabla de tiempos del Intel Xeon. Cada ejecución (celda) se repitió 5 veces. Los valores en negrita se corresponden con el mejor tiempo de una fila. Los resaltados en rojo indican el mejor tiempo global para un tamaño. Se muestran los tamaños 128, 192, 256 y 384. Obsérvese que los peores tiempos siempre se consiguen usando el acceso original a sinogramas y rebanadas, mientras que los óptimos se obtienen asignando a R y a P potencias de dos pequeñas. Así, en 384 con R = 192 y P = 1 el tiempo es de unos 32 segundos. En cambio, si tomamos R = 8 y P = 2, el tiempo se reduce hasta quedar por debajo de los 12 segundos.

2/2		P 1	P 2	P 3	P 4	P 5	P 6	P 9	P 10	P 12	P 15	P 18	P 20	P 30	P 36	P 45	P 60	P 90	P 180	Ganador
512	R 1	35,46	39,09	39,08	39,09	39,07	35,52	33,01	33,04	32,85	32,98	37,75	39,08	39,04	39,06	39,06	32,97	**32,85**	32,97	P 90
	R 2	**29,20**	29,94	32,35	32,36	32,36	32,39	31,17	29,41	29,37	29,43	29,43	31,39	33,00	33,48	33,91	35,95	33,57	32,92	P 1
	R 4	**27,44**	27,51	27,53	28,15	27,59	27,61	27,67	28,00	29,22	29,33	29,45	29,51	29,29	28,54	29,01	31,72	36,83	35,40	P 1
	R 8	**26,57**	26,65	26,68	26,72	27,33	27,54	27,65	27,71	27,76	27,14	28,04	28,13	28,63	29,02	28,33	29,84	31,80	33,24	P 1
	R 16	**26,92**	27,11	27,17	27,24	27,28	27,14	27,18	27,22	27,32	27,39	27,75	27,85	28,00	28,02	29,62	29,62	31,73	33,12	P 1
	R 32	56,73	41,67	36,61	34,16	32,58	31,63	29,88	29,43	28,90	28,47	28,21	27,96	27,43	**27,40**	27,89	29,35	35,20	39,07	P 36
	R 64	72,86	49,20	41,34	37,39	35,01	33,45	30,81	30,30	29,58	28,81	28,32	28,07	27,48	**27,40**	27,90	29,00	31,42	35,55	P 36
	R 128	75,67	50,47	42,10	37,98	35,43	33,79	31,01	30,46	29,64	28,80	28,28	28,00	**27,31**	27,47	28,34	31,56	36,14	39,06	P 30
	R 256	75,92	50,57	42,16	37,99	35,49	33,83	31,02	30,48	29,65	28,82	28,27	28,01	**27,81**	28,20	29,21	31,51	35,61	39,09	P 30
768	R 1	139,91	139,84	139,75	139,65	139,70	139,73	139,69	139,73	139,73	139,85	139,69	**139,06**	139,62	139,71	139,68	139,73	139,69	139,63	P 20
	R 2	**112,70**	112,91	113,05	113,04	113,14	113,15	113,46	113,60	113,62	113,87	114,16	115,35	119,98	122,72	127,37	133,33	137,04	137,54	P 1
	R 3	**102,94**	103,29	103,49	103,65	103,70	104,00	104,62	104,66	105,01	105,96	106,94	107,88	113,40	115,98	122,77	130,22	136,21	137,43	P 1
	R 4	**98,82**	99,15	99,37	99,53	99,71	99,90	100,61	100,83	101,27	102,08	103,13	103,75	109,44	112,09	119,47	129,10	135,76	137,63	P 1
	R 6	**94,82**	95,19	95,47	95,74	96,02	96,25	97,13	97,43	97,85	98,59	99,70	100,11	105,97	109,43	115,85	127,82	135,57	137,83	P 1
	R 8	**93,30**	93,66	94,05	94,39	94,69	95,07	95,98	96,32	96,69	97,53	98,10	98,36	103,87	107,64	114,74	127,68	135,26	137,95	P 1
	R 12	98,47	96,87	**96,26**	96,55	96,61	96,69	96,95	96,96	97,01	97,23	97,90	97,32	101,02	105,96	113,61	126,23	135,34	137,89	P 3
	R 16	125,19	109,63	105,38	102,78	101,52	100,67	99,13	98,74	98,11	97,48	97,40	**96,73**	99,68	106,19	111,99	126,90	135,49	137,98	P 20
	R 24	198,24	143,63	126,16	117,16	111,87	108,36	102,43	101,25	99,45	97,83	**96,77**	96,84	98,90	104,86	112,11	126,65	135,73	138,29	P 18
	R 32	231,67	159,20	135,30	123,46	116,30	111,66	103,79	102,27	99,96	97,93	96,35	**95,76**	97,98	103,69	112,70	126,61	135,72	138,73	P 20
	R 48	251,00	167,37	140,42	126,70	118,68	113,15	104,36	102,61	100,02	97,54	96,13	**96,09**	102,70	105,46	113,28	124,52	135,99	139,29	P 20
	R 64	252,67	168,29	140,87	127,08	118,87	113,36	104,33	102,52	99,91	97,34	95,96	**95,44**	101,23	107,01	112,64	124,49	136,18	139,42	P 20
	R 96	253,86	168,86	141,44	127,42	119,16	113,50	104,34	102,52	99,80	97,28	95,78	**94,89**	99,85	105,98	113,36	123,82	135,98	139,12	P 20
	R 128	254,77	169,41	141,73	127,57	119,27	113,57	104,32	102,44	99,77	97,05	95,75	**94,62**	100,49	109,15	113,22	124,00	135,99	139,50	P 20
	R 192	255,02	169,56	141,76	127,65	119,26	113,62	104,30	102,42	99,69	97,05	95,81	**94,57**	100,71	107,89	112,57	123,05	135,97	139,52	P 20
	R 384	255,40	169,66	141,71	127,62	119,21	113,52	104,22	102,32	99,60	96,88	95,69	**94,68**	100,20	107,26	113,39	122,27	136,38	139,57	P 20
1024	R 1	287,46	287,31	301,67	294,20	287,92	294,74	287,78	285,40	279,79	**261,95**	279,25	279,77	284,91	269,13	287,03	283,76	288,04	284,61	P 15
	R 2	244,21	244,74	244,95	244,94	245,47	245,50	247,14	248,46	**238,85**	249,36	241,15	243,47	265,49	277,59	280,70	282,49	284,80	287,96	P 12
	R 4	229,00	230,17	**224,22**	232,12	228,57	227,18	225,86	226,68	236,70	237,00	243,71	248,37	263,94	269,59	273,99	296,92	285,37	287,09	P 3
	R 8	**221,45**	222,48	226,01	226,75	227,65	227,56	231,34	230,78	233,15	236,53	237,12	241,57	256,90	271,41	282,41	268,18	282,71	287,93	P 1
	R 16	420,98	322,65	289,36	273,00	262,61	255,83	243,82	240,68	238,19	**236,56**	241,13	237,46	259,44	279,83	287,19	275,49	295,38	287,94	P 15
	R 32	584,80	393,15	330,62	299,03	279,89	267,72	247,61	243,73	238,67	237,09	240,04	**235,92**	256,13	251,92	268,40	262,81	273,15	279,37	P 20
	R 64	595,19	398,25	333,41	300,93	281,41	268,41	247,47	243,18	238,58	**235,87**	238,14	236,93	260,07	258,57	275,81	281,04	292,43	283,09	P 15
	R 128	597,43	398,83	333,18	300,78	280,88	268,17	246,76	243,91	238,71	240,67	**235,60**	235,76	241,19	258,41	250,58	272,34	281,98	284,42	P 18
	R 256	597,36	399,36	333,64	300,91	281,11	268,12	246,65	243,15	237,56	240,55	**231,36**	237,68	255,51	262,90	270,23	275,41	268,12	277,76	P 18
	R 512	597,57	399,24	333,59	300,78	280,98	267,97	246,78	242,99	238,24	241,63	**231,44**	243,14	259,13	252,37	285,68	294,42	293,06	296,37	P 18

Tabla B1 (b): Ahora se muestran los tamaños 512, 768 y 1024. Debido a las mayores dimensiones de las imágenes de esta tabla y a la pequeña caché del Xeon, la configuración del mecanismo de caché es más sensible. Es por ello que los tiempos aquí presentados no son tan homogéneos como los de la Tabla A1 (a). De nuevo, los peores tiempos están relacionados con el acceso original a sinogramas y rebanadas, y los óptimos se adhieren a la regla de las potencias pequeñas de dos. Por ejemplo, en 768 la configuración R = 384 y P = 1 devuelve un tiempo de 255,40 segundos, mientras que seleccionando R = 8 y P = 2 conseguimos 93,66 segundos, tiempo muy próximo al mejor de todos (93,30).

1/2		P 1	P 2	P 3	P 4	P 5	P 6	P 9	P 10	P 12	P 15	P 18	P 20	P 30	P 36	P 45	P 60	P 90	P 180	Ganador
128	R 1	0,197	0,198	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	**0,196**	P 180
	R 2	0,198	0,196	0,196	0,195	**0,195**	0,195	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	P 5
	R 4	0,196	0,196	0,195	0,195	**0,195**	0,195	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	P 5
	R 8	0,198	0,197	0,196	0,195	0,195	**0,195**	0,195	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	P 6
	R 16	0,198	0,197	0,196	0,195	0,195	**0,195**	0,195	0,195	0,195	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	P 6
	R 32	0,198	0,198	0,196	0,195	0,195	**0,195**	0,195	0,195	0,195	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	P 6
	R 64	0,198	0,196	0,195	0,195	0,195	**0,195**	0,195	0,195	0,195	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	0,196	P 6
192	R 1	0,657	0,656	0,656	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	**0,654**	P 180
	R 2	0,655	0,655	**0,653**	0,654	0,654	0,654	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	P 3
	R 3	0,655	0,656	0,653	**0,653**	0,653	0,653	0,654	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,654	P 4
	R 4	0,655	0,655	0,653	0,653	**0,653**	0,653	0,654	0,654	0,655	0,654	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,654	P 5
	R 6	0,658	0,655	0,654	0,653	**0,653**	0,653	0,654	0,654	0,654	0,654	0,655	0,654	0,654	0,655	0,655	0,655	0,655	0,655	P 5
	R 8	0,658	0,657	0,654	0,653	**0,653**	0,653	0,654	0,654	0,654	0,654	0,654	0,654	0,655	0,655	0,655	0,655	0,655	0,654	P 5
	R 12	0,659	0,656	0,654	0,653	0,653	**0,653**	0,654	0,654	0,654	0,654	0,654	0,655	0,654	0,655	0,655	0,655	0,655	0,654	P 6
	R 16	0,658	0,655	0,653	0,653	0,653	**0,653**	0,654	0,654	0,654	0,654	0,654	0,654	0,655	0,655	0,655	0,655	0,655	0,654	P 6
	R 24	0,658	0,655	0,653	0,653	0,652	**0,652**	0,654	0,654	0,654	0,654	0,654	0,654	0,654	0,655	0,655	0,655	0,654	0,654	P 6
	R 32	0,658	0,655	0,653	0,653	0,653	**0,652**	0,654	0,654	0,654	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,655	0,654	P 6
	R 48	0,658	0,655	0,653	0,652	**0,652**	0,652	0,654	0,654	0,654	0,654	0,654	0,654	0,655	0,655	0,655	0,655	0,654	0,655	P 5
	R 96	0,658	0,657	0,653	0,653	0,652	0,652	0,654	0,654	0,654	0,654	0,654	0,654	0,654	0,654	0,655	0,655	0,654	0,654	P 5
256	R 1	1,548	1,550	1,545	1,545	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,543	**1,543**	P 180
	R 2	1,551	1,548	**1,542**	1,544	1,545	1,545	1,545	1,545	1,545	1,544	1,545	1,544	1,544	1,544	1,544	1,544	1,544	1,543	P 3
	R 4	1,552	1,550	1,543	1,544	1,545	1,545	1,545	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,543	1,543	1,543	**1,543**	P 180
	R 8	1,550	1,548	**1,542**	1,543	1,545	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,545	1,544	1,543	1,544	1,543	P 3
	R 16	1,551	1,546	**1,542**	1,543	1,544	1,544	1,545	1,544	1,544	1,544	1,544	1,544	1,544	1,543	1,544	1,543	1,544	1,543	P 3
	R 32	1,550	1,547	**1,541**	1,542	1,544	1,544	1,545	1,544	1,544	1,544	1,544	1,544	1,543	1,544	1,543	1,543	1,543	1,543	P 3
	R 64	1,549	1,548	**1,541**	1,542	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,544	1,543	1,544	1,543	P 3
	R 128	1,563	1,553	1,546	1,546	1,547	1,547	1,546	1,546	1,546	1,545	1,545	1,545	1,544	1,544	1,544	1,544	1,543	**1,543**	P 180
384	R 1	5,268	5,264	5,266	5,266	5,265	5,264	5,263	5,263	5,263	**5,262**	5,265	5,266	5,266	5,266	5,265	5,265	5,266	5,267	P 15
	R 2	5,287	5,273	5,270	5,271	5,270	5,269	5,266	5,266	5,265	5,266	5,266	5,265	5,263	5,262	5,262	5,261	**5,261**	5,264	P 90
	R 3	5,288	5,274	5,270	5,269	5,271	5,269	5,267	5,266	5,265	5,264	5,264	5,264	5,262	5,261	**5,260**	5,262	5,262	5,264	P 45
	R 4	5,288	5,272	5,270	5,270	5,272	5,269	5,268	5,267	5,266	5,266	5,267	5,268	5,267	5,266	5,266	5,265	**5,263**	5,268	P 90
	R 6	5,288	5,272	5,268	5,268	5,268	5,267	5,266	5,265	5,264	5,264	5,264	5,264	5,263	5,264	5,263	5,262	**5,262**	5,264	P 90
	R 8	5,285	5,271	5,269	5,269	5,269	5,268	5,267	5,266	5,267	5,265	5,264	5,264	5,262	5,263	**5,262**	5,263	5,263	5,265	P 45
	R 12	5,283	5,271	5,267	5,267	5,268	5,268	5,265	5,265	5,266	5,264	5,264	5,264	5,262	5,262	5,262	5,263	**5,261**	5,264	P 90
	R 16	5,281	5,265	5,264	5,265	5,266	5,266	5,265	5,264	5,263	**5,262**	5,263	5,263	5,262	5,263	5,263	5,263	5,264	5,265	P 15
	R 24	5,279	5,267	5,266	5,266	5,268	5,267	5,265	5,265	5,265	5,264	5,264	5,263	5,262	5,262	**5,261**	5,262	5,262	5,266	P 45
	R 32	5,279	5,268	5,265	5,265	5,266	5,265	5,265	5,265	5,264	5,262	5,262	5,262	5,262	5,262	**5,261**	5,263	5,262	5,265	P 45
	R 48	5,280	5,267	5,266	5,267	5,269	5,267	5,265	5,265	5,266	5,265	5,265	5,263	5,262	5,261	5,261	**5,260**	5,261	5,265	P 60
	R 64	5,282	5,268	5,265	5,265	5,267	5,266	5,264	5,263	5,262	5,265	5,267	5,266	5,264	5,263	5,262	**5,262**	5,262	5,264	P 60
	R 96	5,316	5,285	5,277	5,276	5,275	5,272	5,269	5,268	5,266	5,264	5,266	5,267	5,264	5,263	5,263	5,265	**5,261**	5,264	P 90
	R 192	5,328	5,292	5,281	5,279	5,278	5,275	5,270	5,269	5,268	5,264	5,263	5,262	5,261	5,261	**5,259**	5,260	5,260	5,263	P 45

Tabla B2 (a): Tabla de tiempos del Intel Core 2 Quad. Cada ejecución (celda) se repitió 5 veces. Los valores en negrita se corresponden con el mejor tiempo de una fila. Los resaltados en rojo indican el mejor tiempo global para un tamaño. Se muestran los tamaños 128, 192, 256 y 384. Debido a que los volúmenes reconstruidos son pequeños y a la gran memoria caché de este procesador, no existen diferencias apreciables en los tiempos de cada tamaño.

2/2		P 1	P 2	P 3	P 4	P 5	P 6	P 9	P 10	P 12	P 15	P 18	P 20	P 30	P 36	P 45	P 60	P 90	P 180	Ganador
512	R 1	12,672	12,676	12,667	12,670	12,664	12,665	**12,660**	12,661	12,661	12,670	12,671	12,665	12,666	12,667	12,669	12,661	12,663	12,663	P 9
	R 2	12,737	12,710	12,691	12,677	12,676	12,670	12,665	12,673	12,667	12,664	12,662	12,663	12,661	12,651	12,650	12,647	**12,647**	12,665	P 90
	R 4	12,704	12,688	12,675	12,660	12,655	12,657	12,646	12,645	12,643	12,641	12,644	12,647	12,638	12,641	12,640	**12,632**	12,634	12,656	P 60
	R 8	12,677	12,667	12,663	12,652	12,650	12,645	12,639	12,639	12,637	12,637	12,633	12,634	12,632	**12,629**	12,630	12,630	12,630	12,657	P 36
	R 16	12,668	12,663	12,666	12,658	12,653	12,647	12,641	12,641	12,640	12,636	12,635	12,640	12,637	12,635	12,632	**12,629**	12,632	12,660	P 60
	R 32	12,669	12,662	12,664	12,664	12,655	12,648	12,650	12,643	12,644	12,637	12,636	12,636	12,638	12,637	12,637	**12,630**	12,633	12,660	P 60
	R 64	12,723	12,702	12,685	12,649	12,646	12,637	12,628	12,626	12,623	12,623	12,625	12,626	12,613	12,617	**12,607**	12,614	12,614	12,643	P 45
	R 128	12,742	12,695	12,675	12,653	12,641	12,634	12,617	12,619	12,614	12,611	12,613	12,612	**12,606**	12,607	12,610	12,610	12,615	12,642	P 30
	R 256	12,812	12,745	12,718	12,694	12,673	12,699	12,644	12,656	12,655	12,632	12,624	12,628	12,624	12,624	12,625	12,626	**12,618**	12,645	P 90
768	R 1	43,476	43,457	43,443	43,411	43,409	43,396	43,399	43,398	43,397	43,390	43,393	43,395	43,386	43,391	43,402	**43,384**	43,406	43,392	P 60
	R 2	43,489	43,486	43,435	43,410	43,403	43,388	43,368	43,363	43,361	43,360	43,353	43,359	43,354	**43,348**	43,351	43,350	43,389	43,401	P 36
	R 3	43,423	43,483	43,415	43,410	43,380	43,382	43,363	43,359	43,348	43,349	43,345	**43,332**	43,345	43,341	43,347	43,346	43,376	43,427	P 20
	R 4	43,393	43,448	43,411	43,384	43,392	43,364	43,352	43,342	43,350	43,336	43,328	43,325	**43,322**	43,325	43,329	43,336	43,362	43,410	P 30
	R 6	43,361	43,442	43,412	43,325	43,299	43,262	43,242	43,244	43,241	43,235	43,230	43,225	**43,222**	43,225	43,229	43,229	43,273	43,312	P 30
	R 8	43,255	43,333	43,320	43,285	43,271	43,261	43,244	43,239	43,257	43,250	43,237	**43,234**	43,236	43,251	43,241	43,238	43,289	43,316	P 20
	R 12	43,252	43,353	43,305	43,291	43,305	43,272	43,260	43,252	43,252	43,242	43,241	**43,239**	43,248	43,249	43,247	43,283	43,300	43,342	P 20
	R 16	43,262	43,357	43,319	43,307	43,294	43,276	43,265	**43,255**	43,255	43,257	43,269	43,272	43,266	43,282	43,289	43,272	43,320	43,345	P 10
	R 24	**43,269**	43,381	43,345	43,334	43,310	43,313	43,283	43,284	43,297	43,301	43,278	43,282	43,274	43,275	43,273	43,273	43,343	43,362	P 1
	R 32	43,287	43,402	43,361	43,333	43,281	43,269	43,249	43,247	43,260	43,242	43,233	43,228	**43,224**	43,234	43,235	43,248	43,280	43,306	P 30
	R 48	43,490	43,484	43,400	43,355	43,343	43,322	43,287	43,280	43,271	43,254	43,259	43,253	43,242	**43,242**	43,243	43,264	43,310	43,309	P 36
	R 64	43,572	43,527	43,416	43,365	43,344	43,308	43,265	43,266	43,260	43,243	43,235	43,237	**43,230**	43,256	43,248	43,257	43,321	43,303	P 30
	R 96	43,576	43,524	43,426	43,397	43,340	43,327	43,279	43,279	43,286	43,269	**43,248**	43,256	43,259	43,271	43,283	43,319	43,372	43,315	P 18
	R 128	43,654	43,589	43,507	43,428	43,390	43,380	43,340	43,336	43,328	43,324	43,326	43,346	43,337	43,352	43,364	43,386	43,410	**43,317**	P 180
	R 192	45,769	45,679	44,185	44,300	43,980	43,669	43,608	43,619	43,642	43,655	43,652	43,629	43,592	43,544	43,541	43,492	43,451	**43,244**	P 180
	R 384	116,973	79,807	67,287	60,966	57,492	54,980	50,918	50,011	48,782	47,537	46,750	46,329	45,125	44,754	44,345	43,954	43,573	**43,264**	P 180
1024	R 1	103,13	103,07	103,04	103,03	103,00	102,95	102,95	102,99	103,00	102,97	102,92	102,96	**102,92**	102,95	102,97	102,94	102,99	102,94	P 30
	R 2	102,96	102,89	102,86	102,85	102,81	102,85	102,80	102,81	102,81	102,82	102,80	**102,74**	102,79	102,74	102,80	102,79	102,94	102,91	P 20
	R 4	102,73	102,85	102,75	102,75	102,70	102,69	102,68	102,70	102,69	102,69	102,70	102,72	**102,68**	102,68	102,74	102,75	102,91	102,91	P 30
	R 8	102,63	102,75	102,68	102,71	102,66	102,64	102,63	102,66	102,68	102,68	102,64	**102,62**	102,64	102,65	102,66	102,71	102,94	102,89	P 20
	R 16	102,58	102,66	102,61	102,59	102,63	102,55	102,57	102,62	102,55	102,58	**102,55**	102,62	102,62	102,58	102,59	102,67	102,93	102,91	P 18
	R 32	102,72	102,71	102,60	102,57	102,54	102,52	102,47	102,51	102,47	102,47	102,46	**102,46**	102,50	102,50	102,53	102,64	103,00	102,89	P 20
	R 64	103,20	102,99	102,79	102,67	102,62	102,63	102,58	102,51	102,51	102,53	**102,50**	102,52	102,56	102,63	102,69	102,84	103,10	102,89	P 18
	R 128	103,91	103,72	103,73	103,64	103,61	103,42	103,36	103,27	103,29	103,30	103,26	103,31	103,38	103,50	103,50	103,54	103,45	**102,92**	P 180
	R 256	258,18	179,99	154,57	141,37	133,85	128,25	119,53	117,65	115,00	112,51	110,62	109,84	107,17	106,18	105,37	104,61	103,84	**102,90**	P 180
	R 512	309,72	206,33	171,05	153,77	142,94	136,03	124,30	122,05	118,56	115,09	112,78	111,60	108,28	107,15	106,03	104,94	104,02	**102,90**	P 180

Tabla B2 (b): Ahora se muestran los tamaños 512, 768 y 1024. Obsérvese que en 768 y 1024 se hace necesario el uso del mecanismo de caché, ya que con el acceso original a sinogramas y rebanadas los tiempos son muy altos. Por ejemplo, en 768 si tomamos R = 384 y P = 1, obtenemos 117 segundos, mientras que si elegimos R = 4 y P = 2, conseguimos 43,5 segundos, tiempo prácticamente igual al marcado en rojo (43,2). Esto demuestra lo imprescindible de contar con un mecanismo de *blocking* aun teniendo un procesador con mucha memoria caché.

Bibliografía

[1] J. Abel, K. Balasubramanian, M. Bargeron, T. Craver, and M. Phlipot. Applications tuning for Streaming SIMD Extensions. *Intel Technology Journal*, 3(2):1–13, 1999.

[2] D. Aberdeen and J. Baxter. General matrix-matrix multiplication using SIMD features of the PIII. In *Euro-Par 2000 Parallel Processing*, pages 980–983, 2000.

[3] D. Aberdeen and J. Baxter. Emmerald: a fast matrix-matrix multiply using Intel's SSE instructions. *Concurrency and Computation: Practice and Experience*, 13:103–119, 2001.

[4] J.I. Agulleiro and J.J. Fernández. Fast tomographic reconstruction on multicore computers. *Bioinformatics*, 2011. En prensa (doi: 10.1093/bioinformatics/btq692).

[5] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Fast tomographic reconstruction with vectorized backprojection. In *Actas de las XVIII Jornadas de Paralelismo*, pages 579–586, 2007.

[6] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Fast tomographic reconstruction with vectorized backprojection. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 387–390, 2008.

[7] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Ultra-fast tomographic reconstruction with a highly optimized back-projection algorithm. In *Actas de las XX Jornadas de Paralelismo*, pages 75–80, 2009.

[8] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Towards real time iterative tomographic reconstruction. In *I Simposio en Computación Empotrada*, pages 161–168, 2010.

[9] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Ultra-fast tomographic reconstruction with a highly optimized weighted backprojection algorithm. In *18th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2010)*, pages 281–285, 2010.

[10] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Vectorization with SIMD extensions speeds up reconstruction in electron tomography. *Journal of Structural Biology*, 170:570–575, 2010.

[11] J.I. Agulleiro, E.M. Garzón, I. García, and J.J. Fernández. Multi-core desktop processors make possible real-time electron tomography. In *19th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2011)*, 2011. En prensa.

[12] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52:56–67, 2009.

[13] G. Bernabé, J.M. García, and J. González. Reducing 3D wavelet transform execution time through the Streaming SIMD Extensions. In *11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 49–56, 2003.

[14] G. Bernabé, J.M. García, and J. González. Reducing 3D fast wavelet transform execution time using blocking and the Streaming SIMD Extensions. *The Journal of VLSI Signal Processing*, 41:209–223, 2005.

[15] J.R. Bilbao-Castro, I. García, and J.J. Fernández. EGEETomo: a user-friendly, fault-tolerant and grid-enabled application for 3D reconstruction in electron tomography. *Bioinformatics*, 23:3391–3393, 2007.

[16] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

[17] R. Bordawekar, U. Bondhugula, and R. Rao. Can CPUs match GPUs on performance with productivity?: Experiences with optimizing a FLOP-intensive application on CPUs and GPU. Research Report RC25033, IBM, 2010.

[18] P.P. Bruyant. Analytic and iterative reconstruction algorithms in SPECT. *The Journal of Nuclear Medicine*, 43:1343–1358, 2002.

[19] K.H. Buetow. Cyberinfrastructure: Empowering a 'third way' in biomedical research. *Science*, 308:821–824, 2005.

[20] D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[21] D. Castaño-Díez, D. Moser, A. Schoenegger, S. Pruggnaller, and A.S. Frangakis. Performance evaluation of image processing algorithms on the GPU. *Journal of Structural Biology*, 164:153–160, 2008.

[22] D. Castaño-Díez, H. Mueller, and A.S. Frangakis. Implementation and performance evaluation of reconstruction algorithms on graphics processors. *Journal of Structural Biology*, 157:288–295, 2007.

[23] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable shared memory parallel programming*. MIT Press, 2007.

[24] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. 2-D wavelet transform enhancement on general-purpose microprocessors: Memory hierarchy and SIMD parallelism exploitation. In *High Performance Computing (HiPC 2002)*, pages 9–21, 2002.

[25] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. Wavelet transform for large scale image processing on modern microprocessors. In *High Performance Computing for Computational Science (VECPAR 2002)*, pages 57–76, 2003.

[26] M. Christiaens, B. De Sutter, K. De Bosschere, J. Van Campenhout, and I. Lemahieu. A fast, cache-aware algorithm for the calculation of radiological paths exploiting subword parallelism. *Journal of Systems Architecture*, 45:781–790, 1999.

[27] FEI Company. An introduction to electron microscopy, 2010.

[28] M.T.F. Cunha, J.C.F. Telles, and F.L.B. Ribeiro. Streaming SIMD Extensions applied to boundary element codes. *Advances in Engineering Software*, 39:888–898, 2008.

[29] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High-performance computing: Clusters, constellations, mpps, and future directions. *Computing in Science and Engineering*, 7(2):51–59, 2005.

[30] K. Dowd and C. Severance. *High Performance Computing*. O'Reilly, second edition, 1998.

[31] J.J. Fernández. High performance computing in structural determination by electron cryomicroscopy. *Journal of Structural Biology*, 164:1–6, 2008.

[32] J.J. Fernández, J.M. Carazo, and I. García. Three-dimensional reconstruction of cellular structures by electron microscope tomography and parallel computing. *Journal of Parallel and Distributed Computing*, 64:285–300, 2004.

[33] J.J. Fernández, I. García, J.M. Carazo, and R. Marabini. Electron tomography of complex biological specimens on the Grid. *Future Generation Computer Systems*, 23:435–446, 2007.

[34] J.J. Fernández, A.F. Lawrence, J. Roca, I. García, M.H. Ellisman, and J.M. Carazo. High-performance electron tomography of complex biological specimens. *Journal of Structural Biology*, 138:6–20, 2002.

[35] J.J. Fernández, C.O.S. Sorzano, R. Marabini, and J.M. Carazo. Image processing and 3-D reconstruction in electron microscopy. *IEEE Signal Processing Magazine*, 23(3):84–94, 2006.

[36] J.J. Fernández, J.I. Agulleiro, J.R. Bilbao-Castro, A. Martínez, I. García, F.J. Chichón, J. Martín-Benito, and J.L. Carrascosa. Image processing in electron tomography. In A. Méndez-Vilas and J. Díaz, editors, *Microscopy: science, technology, applications and education*, Microscopy book series. Editorial Formatex, 2011. En prensa.

[37] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. Technical report, Intel Corporation, October 2009.

[38] M.J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett Publishers, 1995.

[39] E. Fontaine and H.H.S. Lee. Optimizing Katsevich image reconstruction algorithm on multicore processors. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS 2007) - Volume 1*, pages 1–8, 2007.

[40] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley, 1995.

[41] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 2003.

[42] A. Fotin, Y. Cheng, P. Sliz, N. Grigorieff, S.C. Harrison, T. Kirchhausen, and T.Walz. Molecular model for a complete clathrin lattice from electron cryomicroscopy. *Nature*, 432:573–579, 2004.

[43] F. Franchetti, S. Kral, J. Lorenz, and C.W. Ueberhuber. Efficient utilization of SIMD extensions. *Proceedings of the IEEE*, 93:409–425, 2005.

[44] J. Frank. *Electron Tomography.* Plenum Press, New York, 1992.

[45] J. Frank. *Three-Dimensional Electron Microscopy of Macromolecular Assemblies.* Academic Press, 1996.

[46] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93:216–231, 2005.

[47] P.C. Fritzsche, J.J. Fernández, D. Rexachs, I. García, and E. Luque. Analytical performance prediction for iterative reconstruction techniques in electron tomography of biological structures. *International Journal of High Performance Computing Applications*, 24:457–468, 2010.

[48] C. García, R. Lario, M. Prieto, L. Piñuel, and F. Tirado. Vectorization of multigrid codes using SIMD ISA extensions. In *International Parallel and Distributed Processing Symposium*, page 58a, 2003.

[49] R.P. Garg and I. Sharapov. *Techniques for Optimizing Applications: High Performance Computing.* Sun Microsystems, 2001.

[50] R. Gerber, A.J.C. Bik, K.B. Smith, and X. Tian. *The Software Optimization Cookbook. High-Performance Recipes for IA-32 Platforms.* Intel Press, second edition, 2006.

[51] R. Gerber and A. Binstock. *Programming with Hyper-Threading technology.* Intel Press, 2004.

[52] P. Gilbert. Iterative methods for the three-dimensional reconstruction of an object from projections. *Journal of Theoretical Biology*, 36:105–117, 1972.

[53] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes.* SIAM, 2001.

[54] M. Hassaballah, S. Omran, and Y.B. Mahdy. A review of SIMD multimedia extensions and their usage in scientific and engineering applications. *The Computer Journal*, 51:630–649, 2008.

[55] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2007.

[56] I.K. Hong, S.T. Chung, H.K. Kim, Y.B. Kim, Y.D. Son, and Z.H. Cho. Ultra fast symmetry and SIMD-based projection-backprojection (SSP) algorithm for 3-D PET image reconstruction. *IEEE Transactions on Medical Imaging*, 26:789–803, 2007.

[57] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*.

[58] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference*.

[59] W. Jiang, M.L. Baker, J. Jakana, P.R. Weigele, J. King, and W. Chiu. Backbone structure of the infectious ϵ15 virus capsid revealed by electron cryomicroscopy. *Nature*, 451:1130–1134, 2008.

[60] M. Kachelriess, M. Knaup, S. Steckmann, F. Marone, and M. Stampanoni. Hyperfast O(2048^4) image reconstruction for synchrotron-based X-ray tomographic microscopy. In *IEEE Nuclear Science Symposium Conference Record*, pages 3810–3813, 2008.

[61] A.C. Kak and M. Slaney. *Principles of computerized tomographic imaging*. SIAM, 2001.

[62] D. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[63] D. Lee, A.W. Lin, T. Hutton, T. Akiyama, S. Shinji, F.P. Lin, S. Peltier, and M.H. Ellisman. Global telescience featuring IPv6 at iGrid2002. *Future Generation Computer Systems*, 19:1031–1039, 2003.

[64] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38:451–460, 2010.

[65] A.P. Leis, M. Beck, M. Gruska, C. Best, R. Hegerl, W. Baumeister, , and J.W. Leis. Cryo-electron tomography of biological specimens. *IEEE Signal Processing Magazine*, 23(3):95–103, 2006.

[66] V. Lučić, F. Förster, and W. Baumeister. Structural studies by electron tomography: from cells to molecules. *Annual Review of Biochemistry*, 74:833–865, 2005.

[67] J.A. Álvarez, J. Roca, and J.J. Fernández. Multithreaded tomographic reconstruction. In *Lecture Notes in Computer Science*, pages 81–88, 2007.

[68] R.A. Neri-Calderón, S. Alcaraz-Corona, and R.M. Rodríguez-Dagnino. Cache-optimized implementation of the filtered backprojection algorithm on a digital signal processor. *Journal of Electronic Imaging*, 16:043010, 2007.

[69] I.M. Orlov, D.G. Morgan, and R.H. Cheng. Efficient implementation of a filtered back-projection algorithm using a voxel-by-voxel approach. *Journal of Structural Biology*, 154:287–296, 2006.

[70] C. Pancratov, J.M. Kurzer, K.A. Shaw, and M.L. Trawick. Why computer architecture matters. *Computing in Science and Engineering*, 10(3):59–63, 2008.

[71] C. Pancratov, J.M. Kurzer, K.A. Shaw, and M.L. Trawick. Why computer architecture matters: Memory access. *Computing in Science and Engineering*, 10(4):71–75, 2008.

[72] C. Pancratov, J.M. Kurzer, K.A. Shaw, and M.L. Trawick. Why computer architecture matters: Thinking through trade-offs in your code. *Computing in Science and Engineering*, 10(5):74–79, 2008.

[73] S.J. Park, J. Ross, D. Shires, D. Richie, B. Henz, and L. Nguyen. Hybrid core acceleration of UWB SIRE radar signal processing. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):46–57, 2011.

[74] S.T. Peltier, A.W. Lin, D. Lee, S. Mock, S. Lamont, T. Molina, M. Wong, L. Dai, M.E. Martone, and M.H. Ellisman. The Telescience Portal for advanced tomography applications. *Journal of Parallel and Distributed Computing*, 63:539–550, 2003.

[75] G.A. Perkins, C.W. Renken, J.Y. Song, T.G. Frey, S.J. Young, S. Lamont, M.E. Martone, S. Lindsey, and M.H. Ellisman. Electron tomography of large, multicomponent biological structures. *Journal of Structural Biology*, 120:219–227, 1997.

[76] M. Schmeisser, B.C. Heisen, M. Luettich, B. Busche, F. Hauer, T. Koske, K.H. Knauber, and H. Stark. Parallel, distributed and GPU computing technologies in single-particle electron microscopy. *Acta Crystallography section D: Biological Crystallography*, 65:659–671, 2009.

[77] S.W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[78] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The complete reference*. MIT Press, 1998.

[79] B. De Sutter, M. Christiaens, K. De Bosschere, and J. Van Campenhout. On the use of subword parallelism in medical image processing. *Parallel Computing*, 24:1537–1556, 1998.

[80] G. Teodoro, R. Sachetto, O. Sertel, M.N. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In *IEEE International Conference on Cluster Computing and Workshops (CLUSTER 2009)*, pages 1–10, 2009.

[81] P.A. Thuman-Commike. Single particle macromolecular structure determination via electron microscopy. *FEBS Letters*, 505:199–205, 2001.

[82] M. van Heel, B. Gowen, R. Matadeen, E.V. Orlova, R. Finn, T. Pape, D. Cohen, H. Stark, R. Schmidt, M. Schatz, and A. Patwardhan. Single-particle electron cryo-microscopy: towards atomic resolution. *Quarterly Reviews of Biophysics*, 33:307–369, 2000.

[83] F. Vázquez, E.M. Garzón, and J.J. Fernández. A matrix approach to tomographic reconstruction and its implementation on GPUs. *Journal of Structural Biology*, 170:146–151, 2010.

[84] F. Vázquez, E.M. Garzón, and J.J. Fernández. Matrix implementation of simultaneous iterative reconstruction technique (SIRT) on GPUs. En revisión, 2011.

[85] K.R. Wadleigh and I.L. Crawford. *Software Optimization for High Performance Computing*. Prentice Hall PTR, 2000.

[86] X. Wan, F. Zhang, and Z. Liu. Modified simultaneous algebraic reconstruction technique and its parallelization in cryo-electron tomography. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, pages 384–390, 2009.

[87] B. Wilkinson. *Grid computing. Techniques and applications*. CRC Press, 2010.

[88] W. Xu, F. Xu, M. Jones, B. Keszthelyi, J. Sedat, D. Agard, and K. Mueller. High-performance iterative electron tomography reconstruction with long-object compensation using graphics processing units (GPUs). *Journal of Structural Biology*, 171:142–153, 2010.

[89] K. Zeng, E. Bai, and G. Wang. A fast CT reconstruction scheme for a general multi-core PC. *International Journal of Biomedical Imaging*, 2007, 2007. Article ID 29160.

[90] X. Zhang, L. Jin, Q. Fang, W.H. Hui, and Z.H. Zhou. 3.3 Å cryo-EM structure of a nonenveloped virus reveals a priming mechanism for cell entry. *Cell*, 141:472–482, 2010.

[91] S.Q. Zheng, B. Keszthelyi, E. Branlund, J.M. Lyle, M.B. Braunfeld, J.W. Sedat, and D.A. Agard. UCSF tomography: An integrated software suite for real-time electron microscopic tomographic data collection, alignment, and reconstruction. *Journal of Structural Biology*, 157:138–147, 2007.