



UNIVERSITAT  
POLITÈCNICA  
DE VALÈNCIA



DEPARTAMENTO DE SISTEMAS  
INFORMÁTICOS Y COMPUTACIÓN

Paralelización del algoritmo de Minimización de la  
Diferencia Total Ponderada para la reconstrucción de  
imagen en tomografías (WTDM-STF)

Héctor Martínez Navarro

Máster en Computación Paralela y Distribuida

Septiembre de 2015

**Tutor:**

Vicent E. Vidal Gimeno

**Co-tutor:**

Esther Durá Martínez







# Agradecimientos

Gracias a Bienve, Claudia, Manolo y Roberto por su constante apoyo, siendo ellos tan distintos y a la vez idénticos a mí.

Gracias a Vicent Vidal por ofrecerme este trabajo de fin de máster, cuyo tema es preferencia personal mía.

Gracias a Esther Durá, co-tutora y profesora de la Universidad de Valencia, por la ayuda ofrecida.

Gracias a Miguel Lozano Ibáñez, que me animó a cursar este máster.

Gracias a Lorena Calabuig y Joan Sanchis, excelentes compañeros.

Gracias a Rosa Sánchez y Héctor Barreiro, padrinos del proceso de elaboración de este trabajo y excelentes amigos, personas e investigadores.

Gracias a mis compañeros de trabajo del grupo LSYM, al *Departament d'Informàtica* de la Universidad de Valencia y al profesorado del Máster en Computación Paralela y Distribuida de la Universidad Politécnica de Valencia (incluyendo a Pedro López y Elvira Baydal) por el apoyo y la comprensión ofrecida.

Por último, gracias a mi madre por enseñarme a aprender de crío, y a Ignacio García Fernández por enseñarme a aprender de adulto.



# Resumen

El diagnóstico basado en imagen es una herramienta fundamental en medicina, siendo un ejemplo muy representativo la Tomografía Axial Computerizada (TAC), una técnica en constante evolución en la que se combina medicina, física y computación.

El TAC se basa en registrar el nivel de densidad de rayos X captados por una serie de sensores bajo la incidencia de un conjunto de proyecciones efectuadas sobre el cuerpo a diagnosticar. Según los distintos materiales y formas de las que consta el cuerpo a diagnosticar, se obtendrán un conjunto de datos basado en la densidad de radiación que ha sido capaz de atravesar al cuerpo.

En base a los valores de radiación registrados se puede reconstruir la imagen del objeto sometido al diagnóstico. Tradicionalmente se han empleado modelos analíticos, debido al bajo coste computacional. El aspecto negativo fundamental de dichos métodos radica en la exigencia de una colección de datos bastante completa, lo que implica someter al cuerpo a diagnosticar a gran cantidad de radiación, siendo potencialmente peligroso para la salud [1].

Los modelos algebraicos representan una alternativa interesante. Implican mayor coste computacional que puede satisfacerse con las prestaciones ofrecidas por la tecnología actual. El gran beneficio respecto a los algoritmos basados en modelos analíticos reside en requerir menor número de proyecciones en el paciente, representando menor riesgo en la salud.

Se ha seleccionado para la realización del presente trabajo el algoritmo de Minimización de la Diferencia Total Ponderada con Filtro de Umbral Suave (WTDM-STF), un método algebraico de reconstrucción de imagen en tomografías. Data de 2014, y los autores lo caracterizan como una opción ventajosa respecto a otros algoritmos, pues requiere menos iteraciones para lograr una buena calidad en la imagen reconstruida [2]. En el presente trabajo se documentan, analizan y comparan una serie de implementaciones del algoritmo, tratando de alcanzar en ellas el menor coste computacional y temporal posible, usando como herramienta clave el paralelismo ofrecido por los dispositivos de procesamiento gráfico (*Graphics Processing Unit*, GPU), usándolos como procesadores *manycore* de propósito general (*General Purpose computing on GPU*, GPGPU).





# Abstract

Image diagnosis is essential to detect medical abnormalities or diseases, being a representative example X-ray computed tomographies (also called CT scan). It is a constantly evolving research field, combining computer and medical science and physics.

CT scans register X-ray density by strategically disposed sensors. X-ray projections radiates the analyzed body and sensors produce a data collection defined by the geometrical forms and materials that compose the diagnosing body, based on the radiation that could go across the body.

Traditionally are analytical methods the most used to use this radiation registered data to reconstruct the image that shows the internal composition of the diagnosed body. These methods imply low computational costs, but require a complete data collection, involving a high radiation with potential danger to the patient [1].

This is the reason why algebraic models are explored during last years. They require a higher computational cost, but are not as strict as analytical models. High quality reconstructed images can be produced with few projections. It is an advantage to prevent radiation risks to the patient.

The Weighted Total Difference Minimization algorithm with Soft Threshold Filtering (WTDM-STF) is chosen in this thesis as a recent algebraic method for few-view computed tomography. Presented in 2014, it is depicted by the authors as an evolution towards past algorithms, because of the low number of iterations required to acquire a good quality reconstructed image [2]. This work tries to document, analyze and compare different implementations of this algorithm, trying to reach the minimal computational and timing costs. A key tool for this will be parallel computing, using graphics processing units (GPUs) as general purpose manycore processors (*General Purpose computing on GPU*, GPGPU).



# Tabla de Contenidos

<b>1. Introducción</b>	<b>1</b>
1.1 Motivación	1
1.2 Objetivos	2
1.3 Estado del arte	2
1.4 Estructura del documento	5
<b>2. Arquitecturas hardware empleadas en sistemas de computación de altas prestaciones (HPC)</b>	<b>7</b>
2.1 Taxonomía de Flynn	7
2.2 Supercomputadores	8
2.3 Clusters	8
2.4 Procesadores multicore	9
2.5 Procesadores manycore	9
2.6 Equipo utilizado	11
<b>3. Bibliotecas software para sistemas de computación de altas prestaciones (HPC)</b>	<b>13</b>
3.1 Intel MKL	14
3.2 CUDA	14
3.3 CUBLAS	16
3.4 CUSPARSE	16
<b>4. Algoritmo WTDM-STF</b>	<b>19</b>
4.1 SART	21
4.2 Filtro de umbral suave (STF)	23
4.3 Acelerador (ACCEL)	24
<b>5. Detalles de la implementación</b>	<b>27</b>
5.1 Versión MATLAB	27

5.2	Versión MKL de Intel	28
5.3	Versión CUBLAS	29
5.4	Versión CUBLAS con uso de objeto textura	31
5.5	Versión CUBLAS con CUSPARSE	32
<b>6.</b>	<b>Pruebas y resultados</b>	<b>35</b>
6.1	Diseño de las pruebas	35
6.2	Resultados obtenidos y discusión	37
6.2.1	El algoritmo WTDM-STF en relación a sus predecesores	38
6.2.2	Las versiones implementadas del algoritmo	39
6.2.3	El número de iteraciones realizadas	42
6.2.4	La cantidad de proyecciones empleada	43
6.2.5	El tamaño de la imagen a reconstruir	45
<b>7.</b>	<b>Conclusiones y trabajo futuro</b>	<b>49</b>
7.1	Conclusiones	49
7.2	Trabajo futuro	50
<b>8.</b>	<b>Bibliografía</b>	<b>53</b>

# Lista de Ilustraciones

Ilustración 1: Esquema simplificado de la secuencia de escaneo	4
Ilustración 2: Arquitectura del sistema de diagnóstico por imagen para tomografías computerizadas de Hounsfield	4
Ilustración 3: Tomografía del cráneo. En la mitad derecha se observa un tumor y la hemorragia provocada	4
Ilustración 4: Arquitectura genérica de las GPU NVIDIA Tesla	10
Ilustración 5: Función saxpy secuencial (izq.) y en kernel en CUDA (der.)	15
Ilustración 6: IZQ, método para hallar las intersecciones de los rayos basada en píxeles (Joseph); DER, método para hallar las intersecciones de los rayos basada en planos (Siddon)	22
Ilustración 7: Cambio del formato del vector u para aplicar el filtro	23
Ilustración 8: Gestión de píxeles en el kernel de filtrado	30
Ilustración 9: FORBILD y su descripción	36
Ilustración 10: Leyenda del fantoma FORBILD	37



# Lista de Ecuaciones

Ecuación 1: Total variation _____	19
Ecuación 2: Total difference _____	20
Ecuación 3: Estructura del algoritmo WTDM-STF _____	20
Ecuación 4: Sistema lineal en la reconstrucción de imagen _____	21
Ecuación 5: Inicialización de variables del algoritmo WTDM-STF _____	22
Ecuación 6: Actualización del vector u en cada resolución de la formula SART _____	22
Ecuación 7: Filtro de umbral suave _____	24
Ecuación 8: Acelerador _____	25
Ecuación 9: Fórmula SART y los términos que no varían durante las iteraciones _____	28
Ecuación 10: Vector Z1 de tamaño N, sumatorio de las columnas _____	29
Ecuación 11: Vector Z2 de tamaño M, sumatorio de las filas _____	29
Ecuación 12: Prototipo de la función del kernel de filtro usando un array de tipo float ____	31
Ecuación 13: Prototipo de la función del kernel de filtro usando una textura _____	31





# Introducción

## 1.1 Motivación

El diagnóstico por imagen es una herramienta fundamental en medicina, siendo la Tomografía Axial Computerizada (TAC) uno de los métodos más extendidos. Se aplica incluso a ámbitos propios de la industria, ya que permite visualizar la composición interna del sujeto u objeto sometido a examen.

El TAC se basa en proyectar radiación desde distintos puntos hacia el paciente. Dicha radiación consta de rayos X, los cuales son percibidos por sensores que registran para cada una de las proyecciones la densidad de radiación captada, siendo ésta afectada por el material y geometría del sujeto a examen. La colección de datos registrada por los sensores permite reconstruir mediante modelos matemáticos una imagen que permite visualizar la estructura interna del elemento sometido a diagnóstico.

Existen dos tipos de reconstrucción de imagen. Uno de ellos es el modelo analítico, cuya mayor ventaja es la bajo coste computacional que implica. Su mayor hándicap reside en la exigencia de una colección de datos bastante completa, que está estrechamente relacionada con la necesidad de someter al sujeto a mayor número de proyecciones y, por lo tanto, a mayor radiación, lo que supone un riesgo para la salud del mismo [1].

Es por este motivo por el que se están investigando alternativas basadas en el modelo algebraico, pues converge en una reconstrucción de imagen de alta calidad con un limitado número de proyecciones, aunque se requiere un coste computacional más elevado.

El algoritmo de Minimización de la Diferencia Total Ponderada con Filtro de Umbral Suave (*Weighted Total Difference Minimization Algorithm with Soft-Threshold Filtering*, WTDM-STF) se basa en el modelo algebraico [2]. Se trata de un algoritmo ite-

rativo que representa una evolución respecto a otros algoritmos, proponiendo una alternativa que requiere menor número iteraciones para obtener unos resultados de calidad equiparable.

Dicho coste computacional podría ser resolverse de diversas formas, aunque la más conveniente está relacionada con la que menor coste temporal y económico implique, de lo que deriva la idea de emplear dispositivos de procesamiento gráfico (*Graphics Processing Unit, GPU*) para llevar a cabo los cálculos. Debido al enorme paralelismo que ofrecen al ser empleadas como procesadores *manycore*, las GPUs permiten realizar múltiples operaciones simultáneamente, siendo muy efectivas para la resolución de problemas algebraicos.

## 1.2 Objetivos

El objetivo del presente trabajo es diseñar, implementar y evaluar diversas implementaciones del algoritmo WTDM-STF para GPU. Este objetivo genérico y principal se deja descomponer en los siguientes puntos:

- Comprensión y familiarización con el algoritmo.
- Implementación básica del algoritmo.
- Implementación de diversas variantes del algoritmo para GPU.
- Ejecución de pruebas sobre las distintas implementaciones.
- Análisis de las pruebas y elaboración de conclusiones.

## 1.3 Estado del arte

Fue en 1917 cuando Johann Radon propuso en un artículo científico el uso de un conjunto integral de líneas atravesando un cuerpo para determinar su composición y distribución de material que lo conforma. En 1956 Bracewell usó dichas técnicas en el campo de la radioastronomía, pero con escasa repercusión y en un tema totalmente ajeno a la medicina.

A partir de 1957 se comenzó a poner en práctica este concepto relacionándolo con la medicina, y fue por parte A M Cormack, quien estudió los distintos niveles de absorción de radiación según el tejido humano. Parte de su trabajo fueron temas en los que

Radon ya trabajó. Además observó que H A Lorentz supo cómo resolver la reconstrucción de imagen tridimensional en 1902 [3].

En los años 60 y 70 se sucedieron una serie de importantes avances, entre los que destacan Gordon et al. y Herman et al. que describieron las técnicas de reconstrucción algebraicas; Bates y Peters presentaron la reconstrucción con transformadas de Fourier; Shepp y Logan realizaron aportaciones fundamentales a los filtros a incorporar en las reconstrucciones [4].

En 1972 llevó Hounsfield por primera vez la reconstrucción de imagen a la práctica, por lo que comúnmente es aceptado como padre de la tomografía computerizada. El primer diagnóstico por TAC data de 1972 en Londres, detectando un tumor en el lóbulo frontal del paciente, respaldando el scanner de Hounsfield como una herramienta de diagnóstico confiable.

Fue la empresa EMI, que hasta ese momento se había dedicado exclusivamente a música y componentes electrónicos, la que comercializó el invento. Hounsfield revolucionó el diagnóstico por imagen que alcanzó un rápido desarrollo y demanda durante la década de 1970. En 1974 se registraron 60 scanners de EMI, mientras que en 1980 la cifra de dispositivos operativos era mayor a 10000 unidades.

En 1979 Cormack y Hounsfield recibieron el premio Nobel en Medicina por sus sobresalientes aportaciones hechas en relación al diagnóstico por imagen.

Las ilustraciones del documento original [5] de 1973 sirven para ilustrar de una manera sencilla y completa el funcionamiento del dispositivo (*Ilustraciones 1 y 2*). Este mismo documento ya presenta ejemplos de pacientes diagnosticados mediante las imágenes reconstruidas por el sistema (*Ilustración 3*).

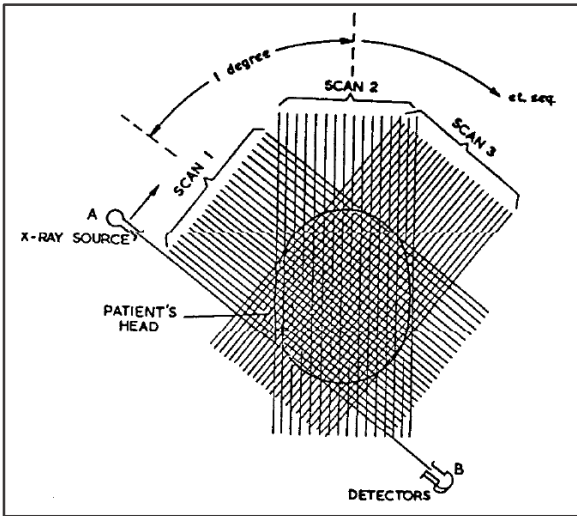


Ilustración 1: Esquema simplificado de la secuencia de escaneo

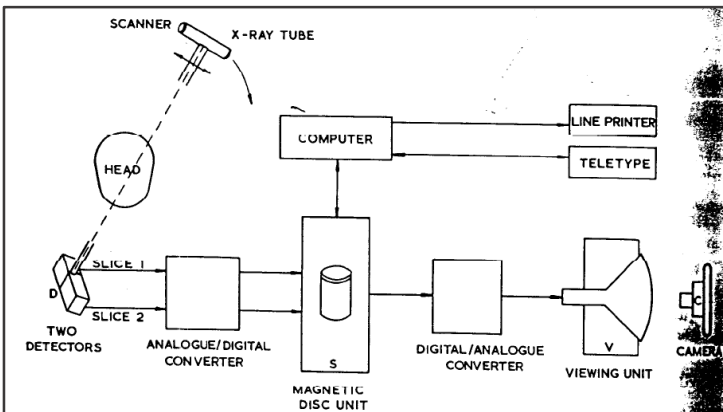


Ilustración 2: Arquitectura del sistema de diagnóstico por imagen para tomografías computerizadas de Hounsfield

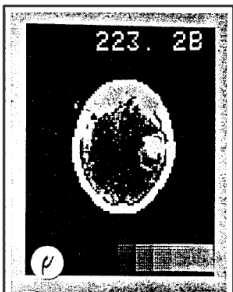


Ilustración 3: Tomografía del cráneo. En la mitad derecha se observa un tumor y la hemorragia provocada

Gordon es, entre otros, quien describe las técnica de reconstrucción algebraica (*Algebraic Reconstruction Technique, ART*) [6], siendo ésta la base del algoritmo WTDM-STF. La reconstrucción algebraica requiere menor número de proyecciones sobre el paciente que la reconstrucción analítica (Radon), aunque mayor coste computacional. A día de hoy se intenta buscar una reconstrucción de imagen rápida y de calidad, pero con el mínimo número de proyecciones necesarias para reducir así la exposición a radiación del paciente.

En los años 90 se extendió el uso de GPUs como co-procesadores gráficos en computadoras personales o de entretenimiento, siendo destinadas a procesar un gran número de píxeles o vértices. Su constante abaratamiento e incremento de capacidad de cómputo (incluyendo cada vez mayor paralelismo en el procesamiento de información) provoca que desde principios del siglo XXI se empleen para usos más generales que requieran operar sobre un gran número de datos [7] [8], de lo que nace el término *General Purpose Computing on GPU* (GPGPU). El desembolso económico de una GPU adecuada para computación de altas prestaciones está al alcance de la mayoría de colectivos interesados en el ámbito.

El presente trabajo consiste en aplicar este modelo de programación a la implementación del algoritmo WTDM-STF, tratando de reducir costes y número de proyecciones empleadas sin renunciar a una buena calidad de imagen.

## 1.4 Estructura del documento

Este documento abarca el contenido del trabajo desarrollado. La estructura del mismo es la siguiente:

- **Capítulo 1. Introducción.** Se expone la motivación que impulsó a la realización del presente trabajo. Además, se detallarán los objetivos de los que consta, así como un breve estado del arte para contextualizar el ámbito que trata. Por último, se expondrá la estructura del documento, definiendo el contenido de cada capítulo.
- **Capítulo 2. Arquitecturas hardware empleadas en sistemas de computación de altas prestaciones.** En este capítulo se exponen las principales

alternativas en lo que arquitecturas de computadores destinados a la computación de altas prestaciones se refiere. Se acaba con una descripción del sistema empleado en las pruebas del trabajo.

- **Capítulo 3. Bibliotecas software para sistemas de computación de altas prestaciones.** Se expondrán una serie de APIs aplicables a la computación de altas prestaciones, algunas de ellas destinadas a la resolución de métodos algebraicos.
- **Capítulo 4. Algoritmo WTDM-STF.** Se describe y contextualiza el algoritmo, analizando cada una de sus partes.
- **Capitulo 5. Detalles de la implementación.** En este capítulo se detallarán las distintas implementaciones desarrolladas del algoritmo.
- **Capítulo 6. Pruebas y resultados.** Cada una de las versiones implementadas se someterán a una serie de pruebas descritas en este apartado. Los resultados obtenidos de la ejecución de estas pruebas se analizarán y se compararán, tratando de obtener conclusiones fundamentadas.
- **Capítulo 7. Conclusiones y trabajo futuro.** En este capítulo se recogen las conclusiones obtenidas de la realización del trabajo, que desembocan en las posibilidades de trabajo futuro.
- **Capítulo 8. Bibliografía.** Listado de referencias.

# Arquitecturas hardware empleadas en sistemas de computación de altas prestaciones (HPC)

## 2.1 Taxonomía de Flynn

Michael J Flynn expuso en su artículo de 1972 *Some Computer Organizations and Their Effectiveness* [9] una clasificación de computadoras según el tipo de procesamiento de datos, proponiendo los siguientes modelos:

- *Single Instruction stream-Single Data*, **SISD**. Es el paradigma más básico y antiguo. Un flujo de datos se procesa secuencialmente. Actualmente se prefieren otras alternativas que exploten el potencial de los procesadores *multicore* y GPUs con los que cuenta cualquier ordenador personal del mercado.
- *Single Instruction stream-Multiple Data*, **SIMD**. En este modelo una sola instrucción se ejecuta sobre distintos datos simultáneamente, produciendo un procesamiento más rápido. Un ejemplo de este esquema se encuentra en los procesadores vectoriales de los supercomputadores antiguos o en las GPUs.
- *Multiple Instructions stream-Single Data*, **MISD**. Este esquema propone la manipulación de un mismo flujo de datos por parte de varios procesadores ejecutando instrucciones distintas. Ejemplos para este paradigma son todos aquellos programas que establezcan un *pipeline*, como la computación gráfica, permitiendo así operar sobre los datos inicialmente en CPU y procesándolos más tarde sobre la GPU.
- *Multiple Instructions stream-Multiple Data*, **MIMD**. Este modelo propone el sistema paralelo más común en la actualidad: distintas unidades de procesamiento procesan distintos flujos de datos de forma independiente.

En base a la taxonomía de Flynn, los distintos tipos de computadores permiten ser clasificados de una manera sencilla según qué tipo de procesamiento paralelo ofrecen. El paralelismo en la resolución de un problema algebraico es crucial para obtener unos costes temporales razonables. A continuación se exponen distintos tipos de máquinas o arquitecturas hardware destinados a la computación de altas prestaciones.

## 2.2 Supercomputadores

A lo largo de la historia se han desarrollado máquinas con capacidad de cómputo mucho mayor que sus coetáneas, destinadas principalmente a institutos de investigación, empresas de alta tecnología o universidades. El supercomputador de mayor éxito es el *Cray 1* [10], de 1976, que disponía de procesador vectorial que le permitía operar con arrays de datos, procesando paralelamente los elementos del array. No obstante, como procesador escalar era también extremadamente rápido. El *Cray 1* alcanzaba un rendimiento de hasta 250 MegaFLOPS (millones de operaciones en coma flotante por segundo), algo insólito para la época, permitiéndole lograr gran éxito comercial. Situando el *Cray 1* como ejemplo de supercomputador, se pueden destacar las siguientes características: precio extremadamente elevado, gran volumen y peso, requiere trabajar en un espacio acondicionado y mantenimiento especializado.

Hoy día los supercomputadores permiten alcanzar un rendimiento de decenas de PetaFLOPS. El supercomputador más potente en el año 2013, el *Tianhe-2*, en China, llega a alcanzar 33.86 PetaFLOPS y consta de 16000 nodos de cómputo interconectados, contando cada uno con 2 procesadores *Intel Ivy Bridge Xeon* y 3 co-procesadores *Intel Xeon Phi*. En total, el supercomputador cuenta con 3.12 millones de núcleos.

En definitiva, este es un enfoque ajeno al presente trabajo, debido a que la motivación es poder resolver un algoritmo del modo que menor coste represente, de modo que, en un contexto realista, el equipo a emplear debe estar al alcance de una clínica que realice diagnóstico por TAC.

## 2.3 Clusters

Un cluster es un conjunto de computadores conectados por una red de alta velocidad, que permiten ser gestionados individualmente o en conjunto. El tamaño del conjunto



de nodos puede variar de las pocas decenas hasta superar el millar. El objetivo fundamental de un cluster es solucionar un problema de la manera más rápida y eficiente posible, repartiendo la carga entre los nodos. Es una alternativa mucho más barata que los supercomputadores, ya que los nodos que conforman un cluster pueden ser computadores convencionales de aplicación genérica. Aunque suponga un desembolso mucho menor que un supercomputador, el coste económico sigue siendo elevado, pues se necesita una red de alta velocidad para conectar los nodos, un área reservada para almacenarlos y asegurar las condiciones necesarias para garantizar un funcionamiento correcto y evitar fallos o averías en la medida de lo posible.

## 2.4 Procesadores multicore

Los procesadores han constado tradicionalmente de un solo núcleo. Rockwell International fabricó en la década de los 80 modelos con dos núcleos, pero no será hasta principios del siglo XXI cuando Intel y AMD, entre otros, comercialicen procesadores multinúcleo o *multicore*. Un procesador *multicore* se diferencia de un procesador de un solo núcleo en que suele contar con 2, 4, 6, 8, 10 o más núcleos como unidades de procesamiento de datos, permitiendo así el paralelismo a nivel de CPU.

El paralelismo ofrecido por los procesadores *multicore* suele depender fuertemente del modo en el que se diseñan e implementan los programas. Los núcleos suelen contar con su propia unidad de memoria cache de nivel 1, pero comparten la cache de nivel 2 y el bus de conexión [11].

A día de hoy es difícil adquirir un ordenador personal cuyo procesador posea un solo núcleo, siendo la mayoría de tipo *multicore*. De modo análogo, el software y sistemas operativos están diseñados en la actualidad para aprovechar el paralelismo ofrecido.

## 2.5 Procesadores manycore

Los procesadores *manycore* son unidades de procesamiento de datos de numerosos núcleos, cuya principal diferencia respecto a los procesadores *multicore* es que suelen ser co-procesadores dedicados al cálculo intensivo o al procesamiento gráfico. Otra gran diferencia es el número de núcleos de los que dispone, ya que contienen varias decenas o cientos. Las GPUs pertenecen a este tipo de procesadores y son el ejemplo

más ilustrativo y extendido. El presente trabajo tiene como objetivo la implementación del algoritmo WTDM-STF en GPU, por lo que es conveniente centrar la atención en la arquitectura de estos dispositivos, así como en el modo en el que se programa software que aproveche su alto rendimiento. Cumple con el modelo *SIMD* según la taxonomía de Flynn, permitiendo procesar múltiples datos de forma paralela sometidos al mismo conjunto de instrucciones.

Aunque en sus comienzos las GPUs fueron dispositivos destinados para acelerar la *pipeline* de gráficos en OpenGL y DirectX, desde hace años son empleados como co-procesadores de propósito general. La biblioteca CUDA, creada por NVIDIA que es fabricante mayoritario de GPUs, permite al desarrollador controlar de forma precisa el paralelismo en la ejecución del código en GPU y manejar los datos en la memoria de la forma más conveniente. En definitiva, el uso de GPUs como co-procesadores de múltiples núcleos es una alternativa en auge hoy en día, debido al altísimo nivel de paralelismo ofrecido, el bajo coste económico que conlleva y el elevado control del desarrollador sobre los mecanismos disponibles. En próximas secciones se expondrá el funcionamiento de CUDA.

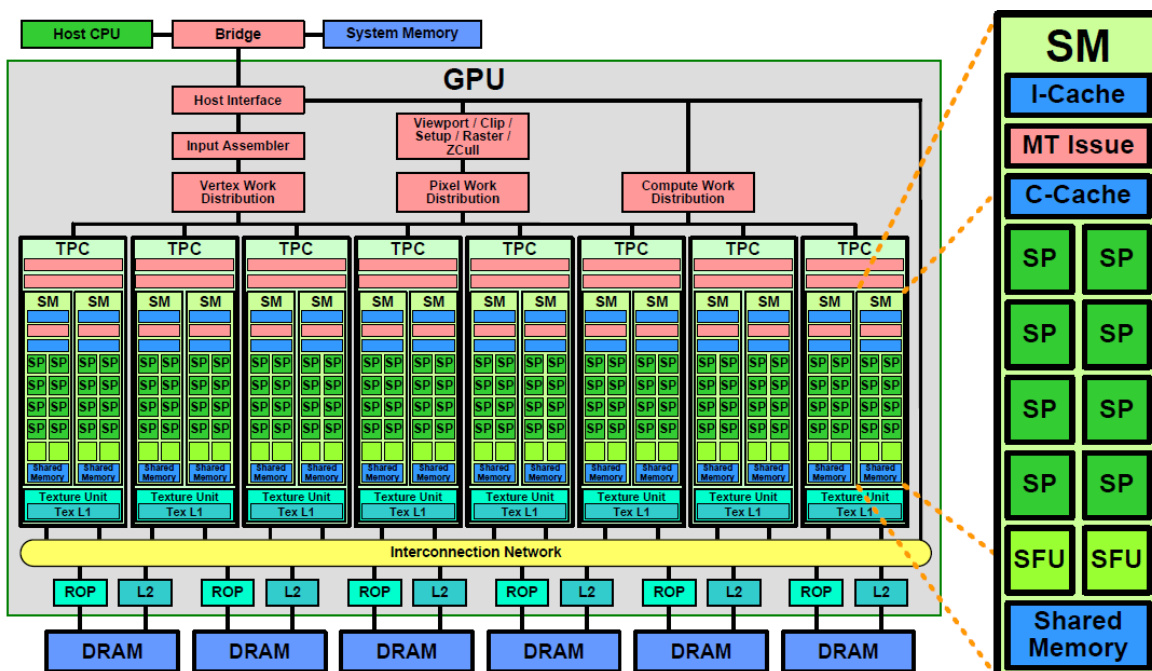


Ilustración 4: Arquitectura genérica de las GPU NVIDIA Tesla

## 2.6 Equipo utilizado

A continuación se describirá el equipo empleado para las pruebas del presente trabajo. Ha sido una decisión lógica, pues la máquina seleccionada es la que cumple de forma más satisfactoria los requisitos. El Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia pone a disposición de este trabajo el servidor `gpu.dsic.upv.es`. Se trata de un servidor que ofrece unas prestaciones excelentes y aptas para la computación de altas prestaciones, pero con un coste razonable, al alcance de organizaciones cuya actividad esté relacionada con la reconstrucción de imagen, como puedan ser centros de investigación, clínicas u hospitales.

Las especificaciones técnicas de la máquina son las siguientes:

CPU:

- **Intel i7-3820 @ 3.6 GHz, 4x cores**

Memoria principal:

- **16 GB**

GPU:

- **2x Tesla K20c**, cada una con:
  - 4.7 GB de memoria
  - `maxBlockSize: 1024, 1024, 64`
  - `maxGridSize: 2147483647, 65535, 65535`
  - `maxThreadsPerBlock: 1024`
  - 13 `StreamMultiProcessors`
  - `sharedMemPerBlock: 49152 bytes`



## Bibliotecas software para sistemas de computación de altas prestaciones (HPC)

Tradicionalmente se ha usado en el ámbito de la computación de altas prestaciones el lenguaje de programación FORTRAN, que fue desarrollado con el objetivo de ser usado en computación científica y análisis numérico. Por otra parte, C/C++ es uno de los lenguajes de propósito general más extendidos. Pese a ser un lenguaje de alto nivel y orientado a objetos, permite la manipulación de datos a muy bajo nivel, por lo que el desarrollador puede programar aplicaciones especialmente adaptadas para un hardware determinado.

En este capítulo se exponen una serie de bibliotecas de altas prestaciones, con el fin de contextualizar las tecnologías estudiadas y usadas en el trabajo. Muchas de ellas se basan en *Basic Linear Algebra Subprograms* (BLAS), una especificación técnica cuya primera implementación es de 1979 para FORTRAN [12]. Cuenta con 3 partes:

- BLAS level 1: operaciones vector – vector.
- BLAS level 2: operaciones matriz – vector.
- BLAS level 3: operaciones matriz – matriz.

También se detallará acerca de CUDA, la biblioteca de NVIDIA para programar sobre GPUs. CUDA puede considerarse una extensión de C/C++. Además tiene soporte de compatibilidad con FORTRAN, pero la modernidad de C/C++, el mayor conocimiento acerca de este lenguaje respecto a otros y trabajar en la línea del DSIC han sido los motivos por los que se ha seleccionado C/C++ como lenguaje de programación para el presente proyecto.

### 3.1 Intel MKL

La *Intel Math Kernel Library* es una biblioteca científica de altas prestaciones que ofrece múltiples funciones matemáticas optimizadas para procesadores *multicore* Intel, de modo que a nivel interno se establece paralelismo en el procesamiento de los datos. Intel proporciona un compilador específico (*icc*) para compilar los programas. Entre sus funciones se encuentran implementaciones de BLAS, de LAPACK y transformadas de Fourier, entre otras [13].

### 3.2 CUDA

CUDA es una biblioteca mediante la cual se puede controlar el procesamiento y gestión de datos en la GPU, proporcionando así una herramienta que propone programar con un elevado grado de paralelismo y libertad. Permite extender C/C++ mediante un reducido conjunto de directivas. Se distinguen los siguientes tipos de subrutinas o funciones en un programa con CUDA:

```
__global__ functionA // indica que la función es un kernel
__host__ functionB // indica que la función es invocada y ejecutada en CPU
__device__ functionC // indica que la función es invocada y ejecutada en GPU
```

Un *kernel* es una función invocada desde la CPU y ejecutada en la GPU. Un *thread* o hilo es la unidad básica de datos en ejecución. Un *kernel* está compuesto de un programa secuencial que es ejecutado por un conjunto de *threads* de forma paralela, según el paradigma SIMD.

Los hilos están organizados por bloques (*blocks*), los cuales se agrupan en una organización mayor llamada *grid*. El programador tiene la capacidad para reorganizar los tamaños de cada uno de estas entidades. Hilos del mismo bloque tienen la capacidad de compartir datos (de acceso especialmente rápido) mediante las variables declaradas como `__shared__`, además de poder activar entre ellos mecanismos de sincronización por barreras con `__syncthreads()`.

Dentro del *kernel*, cada *stream* de datos ha de ser capaz de identificarse a sí mismo, por lo que es necesario consultar un conjunto de palabras reservadas durante la ejecución del *kernel*:

```

vec3 threadIdx    // identifica al hilo dentro de su bloque
vec3 blockIdx    // identifica al bloque dentro del grid
vec3 blockDim    // almacena el número de hilos por bloque
vec3 gridDim     // almacena el número de bloques

```

Obsérvese que las variables representan vectores tridimensionales, debido a que está permitida la creación de estructuras de *datastreams* de 1, 2 o 3 dimensiones. Estos valores son definidos en el momento de invocar un *kernel*. Cuando un *kernel* es invocado desde la CPU, es necesario emplear la siguiente sintaxis:

```
kernel_name << int B, int N >> (parameters);
```

Donde N indica el número de hilos por bloque y B el número de bloques totales.

Supongamos un *kernel* que implementa una función *saxpy* (adición de vectores elemento a elemento), donde N sería el tamaño de los vectores y se desea un número de hilos por bloque igual a 1024. Una llamada típica sería la siguiente:

```
saxpy << ceil( N/1024 ), 1024 >> (parameters);
```

Se muestra a continuación (*ilustración 5*), como ejemplo ilustrativo, un extracto de [14] . Se observa en la mitad derecha un *kernel* que implementa un procedimiento *saxpy* junto con la llamada que lo invoca. Este código permite una ejecución paralela y mucho más rápida que la alternativa secuencial que figura en el lado izquierdo. Bibliotecas orientadas a diversas finalidades, como CUBLAS o PhysX, se basan en CUDA para proporcionar soporte de ejecución en GPU.

<pre> // Serial loop for computing y ← ax+y void saxpy(uint n, float a, float *x, float *y) {     for(uint i = 0; i &lt; n; ++i)         y[i] = a*x[i] + y[i]; }  void serial_sample() {     // Call serial SAXPY function     saxpy(n, 2.0, x, y); } </pre>	<pre> // Parallel kernel for computing y ← ax+y __global__ void saxpy(uint n, float a, float *x, float *y) {     uint i = blockIdx.x*blockDim.x + threadIdx.x;     if( i &lt; n ) y[i] = a*x[i] + y[i]; }  void parallel_sample() {     // Launch parallel SAXPY kernel using     // [n/256] blocks of 256 threads each     saxpy&lt;&lt;&lt;ceil(n/256), 256&gt;&gt;&gt;(n, 2.0, x, y); } </pre>
--	---

Ilustración 5: Función saxpy secuencial (izq.) y en kernel en CUDA (der.)

Por último es necesario distinguir entre dos tipos de memoria: la memoria principal del sistema y la memoria de la GPU. Los datos se encuentran inicialmente en la memoria principal y para su procesamiento deberán llevarse a la memoria de GPU, de donde serán leídos para ser procesados. Más tarde será necesario devolver los resultados a la memoria principal para mostrarlos o almacenarlos. Hay que tener en cuenta que para poder almacenar datos en memoria de GPU se requiere reservar memoria, de forma similar a cómo se haría en memoria principal [15].

```
// reservar memoria en GPU
cudaMalloc( (void **) &d_X, N * sizeof(float) );

// transferir dato de memoria principal a GPU
cudaMemcpy( d_X, X, N * sizeof(float), cudaMemcpyHostToDevice );

// transferir dato de GPU a memoria principal
cudaMemcpy( X, d_X, N * sizeof(float), cudaMemcpyDeviceToHost );

// liberar memoria en GPU
cudaFree(d_X);
```

### 3.3 CUBLAS

CUBLAS es la implementación de BLAS basada en CUDA. Al llamar a una de las funciones de la especificación de BLAS se invoca a una función implementada sobre CUDA.

Está implementado por NVIDIA [16], que asegura un tiempo de ejecución entre 6 y 17 veces menor respecto al de la *Intel Math Kernel Library*.

Los nombres de las funciones de CUBLAS suelen comenzar por `cublas_ [...]`. Para emplear la API es necesario inicializar la biblioteca creando un manejador. Permite el uso de múltiples GPU.

### 3.4 CUSPARSE

CUSPARSE es la alternativa a CUBLAS en caso de que las matrices sean dispersas. También es una implementación de BLAS llevada a cabo por NVIDIA [17]. Es un poco más compleja debido al sistema empleado para almacenar las matrices, pues se



guardan únicamente los valores no nulos y sus índices. NVIDIA afirma unos tiempos hasta 8 veces más rápidos que usando la MKL.

Los nombres de las funciones de CUBLAS suelen comenzar por `cusparse_ [...]`. Al igual que CUBLAS, requiere inicializar la librería creando un manejador.



## Algoritmo WTDM-STF

En este capítulo se estudiará el algoritmo entorno al cual gira este trabajo. Se expondrán los aspectos más importantes del algoritmo, sin entrar a especificar detalles relacionados con una implementación concreta.

Entre los modelos de reconstrucción algebraica de imagen tomográfica, se distinguen dos algoritmos clásicos: el ART (Algebraic Reconstruction Technique) [6] y el SART (Simultaneous Algebraic Reconstruction Technique) [18].

Pese al buen funcionamiento de los modelos de reconstrucción algebraica con un limitado número de proyecciones, el ART y el SART pueden provocar resultados con cierto nivel de artefactos y ruido. Es por ello que investigaciones posteriores buscan alternativas que aporten resultados de mayor calidad, manteniendo limitado el número de proyecciones.

A partir de la década de los 90 se han buscado formas de reducir el ruido y la borrosidad de la imágenes reconstruidas. En 1992 Rudin et al. emplearon la variación total (TV), como término de regularización de imagen [19], empleado inicialmente para el ruido en imágenes estándar, aunque su aplicación fue ampliada a la reconstrucción de imagen en tomografías.

$$TV(u) = \sum_i \sqrt{(D_h u)_i^2 + (D_v u)_i^2}$$

Ecuación 1: Total variation

Donde  $u$  representa el array de la imagen,  $D_h$  representa el gradiente horizontal y  $D_v$  el gradiente vertical.

En 1996, Li y Santosa sugieren emplear la diferencia total (TD), una aproximación computacionalmente eficiente de la TV [20].

$$TD(u) = \|D_h u\|_1 + \|D_v u\|_1$$

Ecuación 2: Total difference

Finalmente en 2010, Yu y Wang desarrollan el algoritmo TDM-STF [21], que usa la TD y ofrece muy buenos resultados. Más tarde se le incorporó un acelerador con el fin de requerir menor número de iteraciones para conseguir buena calidad.

En el artículo donde se presenta el algoritmo WTDM-STF [2], los autores proponen evitar el uso de la TD por no respetar la continuidad de los gradientes en la imagen y, por tanto, perder calidad en los bordes de las figuras representadas. Como alternativa presentan el término WTD (diferencia total ponderada), tratando de corregir la continuidad en los gradientes y mejorar la calidad de los bordes de las figuras.

En próximas secciones se describen las tres fases del algoritmo. El siguiente pseudocódigo muestra la relación entre ellas y el orden propuesto por los autores:

```
for (i=0; i<Niterations; i++)      // Los 3 pasos del algoritmo en cada iteración
{
    SART();           // Fórmula SART que reduce discrepancia entre proyección y original
    STF();           // Filtro de umbral suave para reforzar la continuidad de gradientes
    ACCEL();         // Técnica de aceleración para más rápida convergencia
}
```

Ecuación 3: Estructura del algoritmo WTDM-STF

## 4.1 SART

El punto de partida es un sistema de ecuaciones representado por un producto matriz vector.

$$Au = g$$

Ecuación 4: Sistema lineal en la reconstrucción de imagen

Siendo  $M$  el número de rayos transmitidos, producto del número de emisores y proyecciones,  $g$  es un vector de tamaño  $M$  que representa la radiación emitida en cada disparo de rayos  $X$ .

Siendo  $N$  el número de píxeles en la imagen, obtenido mediante el producto del tamaño en píxeles horizontal por el tamaño vertical, el vector  $u$ , de tamaño  $N$ , es la imagen a reconstruir.

Por último, la variable  $A$ , una matriz dispersa de tamaño  $M \times N$  contiene los datos de la proyección hacia adelante (*forward projection*, como término empleado en la bibliografía).

De modo que, dentro de  $A$ ,  $a_{ij}$  contiene la aportación del rayo  $i$ -ésimo en el píxel  $j$ -ésimo. Hay diversas formas de construir la matriz dispersa  $A$ , entre los métodos más conocidos el de Siddon y el de Joseph.

- El método de Joseph [22] se basa en otros métodos que interpretan cada píxel como un cuadrado y que registran la distancia entre intersecciones. Joseph propone asumir los componentes de la matriz de imagen como un conjunto de valores sobre los que se establece interpolación lineal entre sí. De este modo se logra aumentar la precisión y resolución simplemente incrementando el número de píxeles.
- El método de Siddon [23] propone posteriormente una alternativa más ventajosa ya enfocada a la visualización tridimensional. A diferencia del método de Joseph (entre otros), en el que se asume la existencia de un conjunto de elementos que conforman la matriz de imagen, se propone dividir el espacio en un conjunto orto-

gonal de planos equidistantes entre sí. El resultado ofrece mayor exactitud, facilidad de implementación y eficiencia que otras alternativas. La siguiente ilustración muestra en la parte izquierda la determinación clásica de la trayectoria de un rayo, mientras que en la parte derecha se observa la alternativa de Siddon, basada en discretizar el espacio en planos y hallar en ellos la intersección de los rayos.

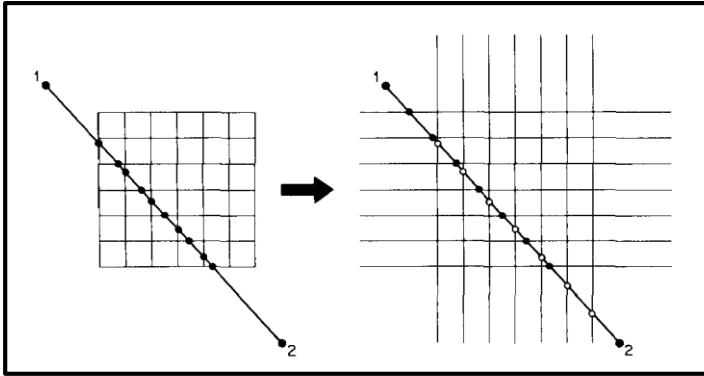


Ilustración 6: IZQ, método para hallar las intersecciones de los rayos basada en píxeles (Joseph); DER, método para hallar las intersecciones de los rayos basada en planos (Siddon)

El algoritmo debe contar con las variables previamente inicializadas.

$$n=0, \mathbf{u}^{(n)}=0, \mathbf{u}_{temp}=0, t_n=1, \alpha=1.0.$$

Ecuación 5: Inicialización de variables del algoritmo WTDM-STF

Donde  $\mathbf{u}_{temp}$  es una variable intermedia que almacena el valor de  $\mathbf{u}$  obtenido en la anterior iteración, mientras que  $\alpha$  es una variable empleada en el STF, que más tarde se estudiará. En cada iteración el vector  $\mathbf{u}$  es actualizado del siguiente modo:

$$\tilde{u}_j^{n+1} = u_j^n - \gamma^n \frac{1}{\sum_{i=1}^M a_{i,j}} \sum_{i=1}^M \frac{a_{i,j}}{\sum_{j=1}^N a_{i,j}} (g_i - \mathbf{A}_i \mathbf{u}^n), j=1,2,\dots,N.$$

Ecuación 6: Actualización del vector  $\mathbf{u}$  en cada resolución de la formula SART

Donde el término  $\gamma^n$  es un parámetro relax.

El coste asimptótico de esta parte del algoritmo es de orden  $O(MN)$ , siendo el cálculo más costoso el producto matriz – vector. Es necesario puntualizar que en caso de emplear el producto matriz – vector implementado para matrices dispersas, el coste de la operación será de orden  $O(NNz)$  siendo NNz el número de elementos no nulos en la matriz. En la fórmula SART aparecería una nueva dimensión a tener en cuenta.

El cálculo de matrices dispersas tiene un coste adicional relacionado al almacenamiento de los datos. Por lo que incorporar el producto matriz – vector para matrices dispersas al algoritmo será más eficiente únicamente si se cumple la siguiente condición:

$$t_{Ab} > t_{initCUSPARSE} + t_{A_{sparse}B}$$

El cumplimiento de la condición depende del tamaño total de la matriz A y de la proporción de elementos no nulos que incluye.

## 4.2 Filtro de umbral suave (STF)

Tras actualizar el vector  $u$  se aplica el filtro de umbral suave sobre dicho vector, que contiene los píxeles de la imagen a reconstruir.

Para poder aplicar el filtro se requiere reestructurar su forma, de manera que el vector de tamaño N, sea interpretado como la imagen reconstruida, es decir, una matriz de H píxeles de alto por W píxeles de ancho. En C/C++ implicaría simplemente acceder a los datos del vector de un modo concreto para tener en cuenta los píxeles vecinos del píxel tratado. En MATLAB lo más sencillo sería emplear la siguiente directiva:

$$u_{reshaped} = reshape(u, W, H);$$

Ilustración 7: Cambio del formato del vector u para aplicar el filtro

Tras reestructurar el vector  $u$  (o establecer un modo de acceder a píxeles vecinos) se procede a aplicar el filtro.

$$\begin{aligned}
u_{i,j}^{(n+1)} &= \frac{1}{4+4\cdot\alpha} (q(\omega, \tilde{u}_{i,j}^{(n+1)}, \tilde{u}_{i+1,j}^{(n+1)}) + q(\omega, \tilde{u}_{i,j}^{(n+1)}, \tilde{u}_{i,j+1}^{(n+1)}) + q(\omega, \tilde{u}_{i,j}^{(n+1)}, \\
&\tilde{u}_{i,j-1}^{(n+1)}) + q(\omega, \tilde{u}_{i,j}^{(n+1)}, \tilde{u}_{i-1,j}^{(n+1)}) + \alpha \cdot ((q(\omega, \tilde{u}_{i,j}^{(n+1)}, \tilde{u}_{i+1,j+1}^{(n+1)}) + q(\omega, \tilde{u}_{i,j}^{(n+1)}, \\
&\tilde{u}_{i+1,j-1}^{(n+1)}) + q(\omega, \tilde{u}_{i,j}^{(n+1)}, \tilde{u}_{i-1,j-1}^{(n+1)}) + q(\omega, \tilde{u}_{i,j}^{(n+1)}, \tilde{u}_{i-1,j+1}^{(n+1)})), \\
q(\omega, y, z) &= \begin{cases} (y+z)/2, & \text{if } |y-z| < \omega \\ y-\omega/2, & \text{if } y-z \geq \omega \\ y+\omega/2, & \text{if } y-z \leq -\omega. \end{cases}
\end{aligned}$$

$$\omega = \max_i |r_i|, \mathbf{r} = \mathbf{A}^* (\mathbf{g} - \mathbf{A}\mathbf{u})$$

Ecuación 7: Filtro de umbral suave

Como se observa en la ecuación del filtro, los bordes de la imagen son corregidos al reforzar el contraste entre píxeles vecinos. El umbral establecido por  $\omega$  representa la *minimización de la diferencia total ponderada* (WTDM).

El valor  $\alpha$  está inicializado inicialmente a 1.0, aunque es modificable para conseguir mejores resultados con el filtro, dependiendo de las características de la imagen a reconstruir, ya que si su valor fuera siempre ése, carecería de sentido. Esta variable añade o resta presencia a los píxeles vecinos en diagonal respecto a los horizontales y verticales.

El coste algorítmico del filtro es de orden  $\mathcal{O}(N)$ , puesto que el filtro se aplica sobre cada uno de los elementos de la imagen (N en total).

### 4.3 Acelerador (ACCEL)

La técnica de aceleración permite converger más rápidamente en los resultados, requiriendo menor número de iteraciones en el algoritmo.

La variable  $\mathbf{u}_{temp}$  (ecuación 8) es almacenada con el resultado de  $\mathbf{u}$  de la iteración  $k$ , pues es necesario para el acelerador de la iteración  $k+1$ .



$$\begin{aligned}
\mathbf{h} &= \mathbf{u}^{(n)}; \\
t_{n+1} &= \frac{1 + \sqrt{1 + 4t_n^2}}{2}; \\
\mathbf{u}^{(n+1)} &= \mathbf{h} + \left(\frac{t_n - 1}{t_{n+1}}\right)(\mathbf{h} - \mathbf{u}_{temp}); \\
\mathbf{u}_{temp} &= \mathbf{h};
\end{aligned}$$

Ecuación 8: Acelerador

El coste asintótico del acelerador es de orden  $O(N)$ , puesto que se aplica a los resultados del vector que contiene la imagen de  $N$  pixels.



## Detalles de la implementación

Tras conocer el algoritmo se deciden realizar varias versiones del mismo usando distintas técnicas. En primer lugar se ha realizado una implementación en MATLAB, como prueba de concepto. Se ha decidido implementar una versión sobre la biblioteca MKL de Intel para poder comparar el rendimiento de las versiones en GPU respecto al de una versión ejecutada en CPU.

Se han hecho 3 implementaciones sobre GPU. Una de ellas basada en CUBLAS, otra similar pero haciendo uso del objeto *textura* para optimizar accesos a memoria. La última implementación sustituye el producto matriz – vector en CUBLAS por una llamada CUSPARSE con la intención de reducir costes, pues la matriz  $A$  es dispersa, ya que no todos los rayos emitidos inciden en todos los píxeles de la imagen a reconstruir.

### 5.1 Versión MATLAB

Esta versión es un prototipo creado para estudiar el algoritmo al detalle. Es la implementación más sencilla, ya que MATLAB es un lenguaje muy flexible, que abstrae al desarrollador del hardware. En ningún momento es necesario preocuparse de la gestión de memoria.

Además, la flexibilidad del lenguaje permite que con poca experiencia se puedan implementar rápidamente algoritmos expresados en notación matemática, como las fórmulas y ecuaciones del capítulo anterior. Por citar un ejemplo, la fórmula SART en MATLAB es posible de implementar en una sola línea, debido a la inmensa cantidad de operaciones disponibles y un tipado débil.

Por otra parte, el desarrollo es cómodo y rápido, pues no requiere ser compilado, con un solo botón comienza la ejecución del código escrito. La depuración es extremadamente sencilla, permitiendo establecer puntos de ruptura en el programa o en las

funciones invocadas para paralizar el estado de ejecución en cualquier momento y comprobar el estado de una variable o realizar alguna operación.

El mayor problema que surge en el desarrollo en MATLAB es el rendimiento. Es realmente bajo, el lenguaje es interpretado y no se ejecuta de forma paralela a menos de que se implementen los mecanismos necesarios. Esta serie de características permiten contemplar a MATLAB como una herramienta muy poderosa de prototipado, que sirve de piedra angular para el resto de versiones.

## 5.2 Versión MKL de Intel

La implementación *multicore* del WTDM-STF se ha llevado a cabo usando la API de la *Intel Math Kernel Library*, en concreto sobre su implementación de BLAS.

A continuación se detallarán algunas cuestiones precisas acerca de la implementación. Retomando la fórmula SART, se distinguen:

- Dos términos invariables que son tomados de los valores iniciales en cada iteración:  $\mathbf{g}$  y  $\mathbf{A}$ , marcados en la ecuación 9 de color azul.
- Dos términos que pueden ser precalculados con anterioridad a las iteraciones sin necesidad de calcularlos una segunda vez, aquellos que en la ecuación 9 están marcados de color naranja.

$$\tilde{u}_j^{n+1} = u_j^n - \gamma^n \frac{1}{\sum_{i=1}^M a_{i,j}} \sum_{i=1}^M a_{i,j} \left( g_i - \mathbf{A}_j \mathbf{u}^n \right), j = 1, 2, \dots, N.$$

Ecuación 9: Fórmula SART y los términos que no varían durante las iteraciones

Por este motivo se definen las variables  $\mathbf{Z1}$  y  $\mathbf{Z2}$ , que corresponden a los términos en color naranja.

$$Z1 = \sum_{i=1}^M a_{ij}$$

Ecuación 10: Vector Z1 de tamaño N, sumatorio de las columnas

$$Z2 = \sum_{j=1}^N a_{ij}$$

Ecuación 11: Vector Z2 de tamaño M, sumatorio de las filas

El resto del algoritmo es una adaptación de la versión en MATLAB usando las primitivas propuestas por BLAS. En caso de requerir operaciones ajenas a BLAS, éstas han sido implementadas usando la librería openMP, que proporciona procesamiento multihilo.

### 5.3 Versión CUBLAS

La versión del algoritmo basada en la *Intel Math Kernel Library* sirve de piedra angular para el resto, ya que la especificación de BLAS no varía, independientemente de su implementación, por lo que las primitivas empleadas son muy similares.

En esta versión se está trabajando con una API basada en CUDA, por lo que será necesario usar las técnicas de programación adecuadas: transferencia de GPU a memoria principal y viceversa, reserva de memoria en GPU y programación según el esquema SIMD.

Todas aquellas operaciones que no están disponibles en la especificación de BLAS se han implementado mediante *kernels*, de manera que su ejecución sea sobre GPU y lo más eficiente posible. El filtro de umbral suave es un ejemplo. Es conveniente comentar algunos puntos acerca de la implementación del filtro

El filtro es un *kernel* que recibe como argumentos el umbral a aplicar ( $\omega$ ), las dimensiones de la imagen, el vector que almacena los píxeles ( $X$ ), un vector del mismo tamaño vacío para guardar la imagen filtrada sin eliminar la anterior ( $newX$ ) y el

valor  $\alpha$  que proporciona menor o mayor grado de participación a los píxeles diagonales respecto al píxel tratado.

El algoritmo no es excesivamente complejo, aunque sí que es necesario conseguir el máximo grado posible de paralelización o el rendimiento proporcionado por las bibliotecas empleadas será lastrado. La GPU de la que se dispone es capaz de crear un máximo de 1024 threads o hilos por bloque, siendo 1024 el cuadrado de 32. El tamaño de threads por *warp* (agrupación de procesamiento hardware basada en hilos) es 32, por lo que se consideró como opción óptima de tamaño de bloque 32x32. Cada bloque operará en un conjunto de 32x32 píxeles (1024 en total), satisfaciendo así la concordancia con el tamaño de hilos por *warp* y máximo número de hilos por bloque.

A simple vista puede resultar trivial el procesamiento del filtro por bloques, pero queda definir ciertos accesos a píxeles ajenos al bloque.

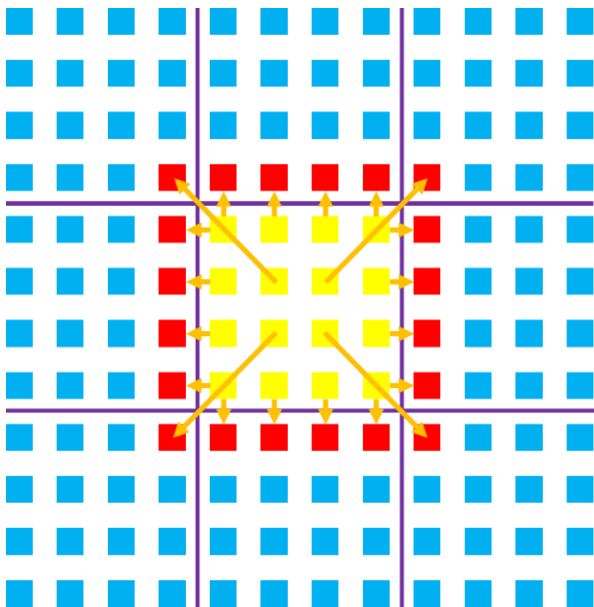


Ilustración 8: Gestión de píxeles en el kernel de filtrado

En una imagen a filtrar (*ilustración 7*), se supone un tamaño de bloque 4x4 y un bloque central para simplificar la aclaración (color amarillo). Cada bloque solamente debe manipular los píxeles que le corresponde para poder lograr el mayor paralelismo y escalabilidad posible. El dilema consiste en poder contar con los píxeles de color rojo,

los adyacentes y externos al bloque que se ejecuta. La estrategia a seguir para lograr el máximo rendimiento es la siguiente:

Se aprovechará la alta velocidad que permite la compartición de datos entre los hilos de un mismo bloque. Entonces, previo al procesado de los píxeles del bloque, se accederá a los píxeles vecinos y se almacenarán en un array de tipo `__shared` de forma conjunta a los píxeles del mismo bloque, para así ser rápidamente accesibles y compartidos por los hilos. En la ilustración se muestra mediante flechas de color naranja qué hilos (simbolizados por el píxel que procesan) deben acceder a qué píxeles adyacentes, además de acceder a sus propios píxeles. Este proceso minimiza y paraleliza los accesos a memoria global de la GPU.

Posteriormente se adapta el algoritmo para que, según el bloque que se procese, se accedan a los datos correctos, evitando accesos a memoria inválidos.

Las siguientes implementaciones del algoritmo y del filtro son evoluciones de ésta, aportando variaciones que buscan una mejora del rendimiento.

## 5.4 Versión CUBLAS con uso de objeto textura

Se parte de la versión del algoritmo anterior. La diferencia radica en el *kernel* que aplica el filtro de umbral suave. A continuación se presentan los prototipos de la función del *kernel*. En la implementación básica en CUBLAS la matriz de imagen es un array de tipo *float*, mientras que en esta implementación se ha creado un objeto de textura

*cudaTextureObject\_t* con el fin de acceder más rápidamente a la matriz que representa la imagen reconstruida (vector *u* reestructurado), aprovechando que durante la ejecución del *kernel* es considerada variable de sólo lectura.

```
__global__ void STfilter(int M, int N, float *X, float *newX, float w, float alpha );
```

Ecuación 12: Prototipo de la función del kernel de filtro usando un array de tipo float

```
__global__ void STfilterTex(int M, int N, cudaTextureObject_t tex, float *newX, float w, float alpha );
```

Ecuación 13: Prototipo de la función del kernel de filtro usando una textura

## 5.5 Versión CUBLAS con CUSPARSE

La matriz  $A$  es una matriz dispersa, por lo que se ha variado la implementación original en CUBLAS de manera que el programa disponga también de la API CUSPARSE. Ésta es empleada para cálculos con matrices dispersas. Hay una única variable con forma de matriz dispersa, la matriz  $A$ , cuya participación en el algoritmo es en un producto matriz-vector. Éste será el único caso en el que una llamada a función CUSPARSE sustituirá la llamada original de CUBLAS.

Hay que tener en cuenta que en este caso no sólo se requiere lanzar el manejador de una API, sino de dos, además de crear estructuras de datos más complejas para almacenar la matriz  $A$ , en las que se guardan, además de los valores no nulos de la matriz, los índices donde se sitúan. El objetivo es evitar costes innecesarios derivados de los cálculos relacionados con elementos nulos. Se ha seleccionado el formato *Compressed Sparse Column* (CSC). Éste formato se basa en almacenar los valores no nulos de la matriz  $A$  en un array. Dos arrays adicionales serán necesarios para almacenar la indexación de dichos elementos. Uno de ellos almacenará para la posición de cada elemento en su propia columna de la matriz, en el supuesto de que  $A$  se descomponga en columnas (de ahí el nombre del formato). Para conocer a qué columna concreta se hace referencia, un vector de tamaño  $M+1$  almacenará la posición en el vector de valores de aquellos valores no nulos que en la matriz son el primer elemento no nulo de la columna. Para ilustrar se adjunta un ejemplo de la documentación de CUSPARSE para el almacenamiento comprimido en columnas [24]:

$$\text{Siendo } A = \begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 \end{bmatrix}$$

El almacenamiento de la matriz en formato CSC sería:

$$\begin{aligned} \text{cscValA} &= [1.0 \ 5.0 \ 4.0 \ 2.0 \ 3.0 \ 9.0 \ 7.0 \ 8.0 \ 6.0] \\ \text{cscRowIndA} &= [0 \ 2 \ 0 \ 1 \ 1 \ 3 \ 2 \ 2 \ 3] \\ \text{cscColPtrA} &= [0 \ 2 \ 4 \ 6 \ 7 \ 9] \end{aligned}$$



Existen otros tipos de formatos [25], como *Coordinate Format* (COO), *Compressed Sparse Row Format* (CSR), híbrido (HYB)... Se ha elegido CSC por la facilidad que supone usarlo debido a que la matriz  $A$  está almacenada por columnas.



# Pruebas y resultados

## 6.1 Diseño de las pruebas

En este capítulo se evaluarán las distintas versiones del algoritmo, con el fin de averiguar qué estrategia es más eficiente computacionalmente. Las implementaciones (a excepción de aquella basada en MATLAB) serán testadas variando principalmente los siguientes parámetros:

- Tamaño de la imagen a reconstruir en píxeles (64x64, 128x128...).
- Número total de proyecciones, lo que incide linealmente en el número de rayos proyectados  $M$  (180, 120, 60). La colección de datos de la tomografía simulada establece una proyección cada 1.0 grados o 0.01745 radianes, con un total de 360 proyecciones equidistantes disponibles. Se seleccionarán subconjuntos más pequeños y se evaluará la calidad de la imagen obtenida. Las proyecciones de dichos subconjuntos deberán ser equidistantes entre sí de manera que si se escogen 60 proyecciones, deberán respetarse intervalos de 6 proyecciones. El número de sensores es constante.
- El número de iteraciones efectuadas en el algoritmo (200, 400, 600...).
- Pasos del algoritmo incluidos (SART, SART con acelerador, completo).
- La implementación (MKL, CUBLAS, CUBLAS y memoria de texturas, CUBLAS y CUSPARSE).

Se analizarán los tiempos de ejecución, el ruido respecto al fantoma original, el aspecto visual de la imagen reconstruida y el coste del algoritmo, contrastándolo con su coste teórico.

En la evaluación de costes temporales se incluirán los costes relacionados con la transferencia de datos entre distintos tipos de memoria y creación o destrucción de manejadores, pues son factores a tener en cuenta en una implementación sobre GPU.

Además, se mostrará la diferencia entre el SART original de 1979 (con o sin acelerador) y el WTDM-STF. Se busca la mejor calidad de imagen con el mínimo número de proyecciones posible.

Se trabajará en la línea de otros trabajos del departamento [26], usando para las pruebas sus colecciones de datos, basadas en simulaciones de tomografía realizadas sobre el fantoma FORBILD [27].

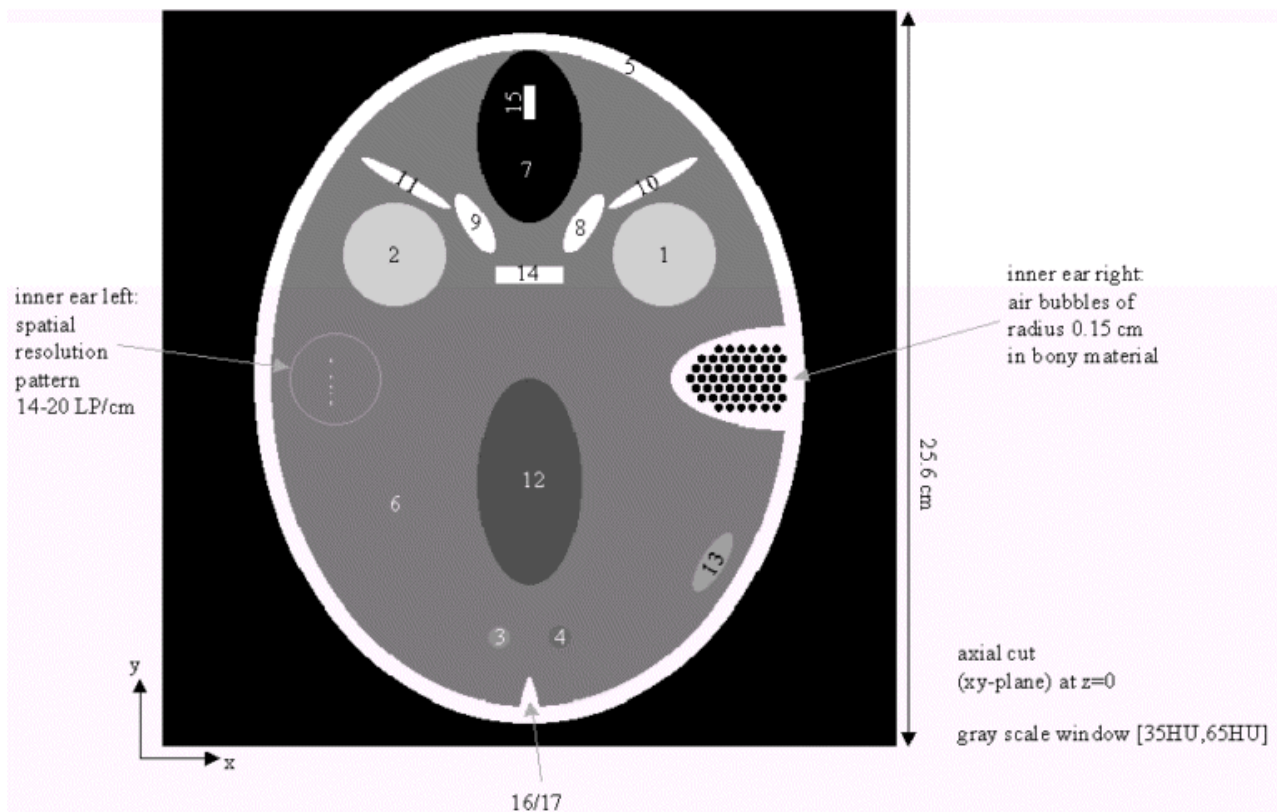


Ilustración 9: FORBILD y su descripción

En el fantoma FORBILD se distinguen una serie de figuras que emulan órganos o partes del cráneo, con el fin de poder evaluar la calidad de la reconstrucción, teniendo en cuenta qué figuras son reconstruidas en un nivel de precisión aceptable.

En las pruebas se empleará una variación del FORBILD sin el oído izquierdo.

La leyenda del FORBILD es la siguiente:

label	geometry	CT-number	anatomical relation
1	sphere	60	eye
2	sphere	60	eye
3	sphere	52.5	
4	sphere	47.5	
5	ellipsoid	800	calotte
6	ellipsoid	50	homogeneous brain matter
7	ellipsoid	-1000	frontal sinus
8	ellipsoid	800	bone surrounding frontal sinus
9	ellipsoid	800	bone surrounding frontal sinus
10	elliptical cylinder	800	bone surrounding frontal sinus
11	elliptical cylinder	800	bone surrounding frontal sinus
12	ellipsoid	45	ventricle
13	ellipsoid	55	subdural hematoma
14	elliptical cylinder	800	bone surrounding frontal sinus
15	elliptical cylinder	800	bone surrounding frontal sinus
16	cone	800	internal occipital protuberance
17	cone	800	internal occipital protuberance

**Table 1:** Description of all geometrical objects except inner ear.

Ilustración 10: Leyenda del fantoma FORBILD

## 6.2 Resultados obtenidos y discusión

De las múltiples pruebas que se han realizado, se han seleccionado aquellos que aporten mayor información a la hora de elaborar conclusiones.

### 6.2.1 El algoritmo WTDM-STF en relación a sus predecesores

Se han realizado muchos esfuerzos con el fin de reproducir los excelentes resultados del algoritmo WTDM obtenidos por sus autores. Se ha programado el lanzamiento del programa que reconstruye la imagen aplicando distintos valores de  $\alpha$  en el filtro y  $\gamma$  en la fórmula SART, dentro de los rangos deseados, y probando las combinaciones posibles por ambos valores.

Si se realiza dentro de cada iteración la secuencia SART-STF-ACCEL se obtienen unos resultados inaceptables sean cuales sean los valores seleccionados para  $\alpha$  y  $\gamma$ . La imagen queda “quemada”, por decirlo de algún modo, pues los contrastes son llevados a extremos que hacen la imagen indistinguible. El único modo que permite implementar el filtro de un modo aceptable es aplicarlo una vez se han finalizado las iteraciones. Dichos resultados suavizan los bordes en caso de hallar los valores adecuados de  $\alpha$  y  $\gamma$  introduciendo ruido en la imagen, contrariamente a lo que se indica en el artículo donde se presenta el WTDM-STF.

Se ha comparado el filtro con otras implementaciones del departamento para comprobar su corrección. Asimismo, se ha intentado contactar con los autores del artículo [2], aunque sin éxito.




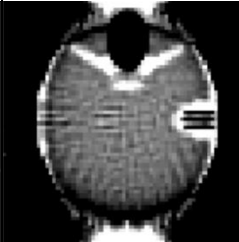
Image 64x64, 200 iterations, 60 projections				
	Original	SART	SART + accelerator	WTDM-STF
Image				
Noise difference	-	Base to compare	-4,02%	1,63%

Tabla 1: Pruebas realizadas comparando el algoritmo WTDM-STF con el SART (con y sin acelerador)

La **tabla 1** expone las diferencias entre el algoritmo SART, el algoritmo SART acelerado y el WTDM-STF con un solo filtrado tras las iteraciones. Los valores de  $\alpha$  y  $\gamma$  no son arbitrarios, son aquellos con los que mejores resultados se obtienen para las

circunstancias determinadas en la prueba. Es evidente que el acelerador hace su función procurando una convergencia más rápida, reduciendo el nivel de ruido al incluirlo. Si bien la versión completa del WTDM-STF aumenta el nivel de ruido, consigue corregir algunos de los artefactos producidos, (obsérvese la parte superior de la figura). Con resignación, se han aceptado adoptar la versión acelerada de SART como el algoritmo con el que se van a lanzar las siguientes pruebas. Esto deriva en desechar la versión del algoritmo que, además de CUBLAS, hace uso de objetos textura, pues dicha característica se aplicaba únicamente en el filtro STF.

### 6.2.2 Las versiones implementadas del algoritmo

Las **tablas 2, 3 y 4** muestran el comportamiento de las distintas implementaciones del algoritmo de reconstrucción de imagen (SART acelerado) ante distintas circunstancias.


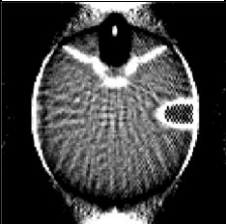
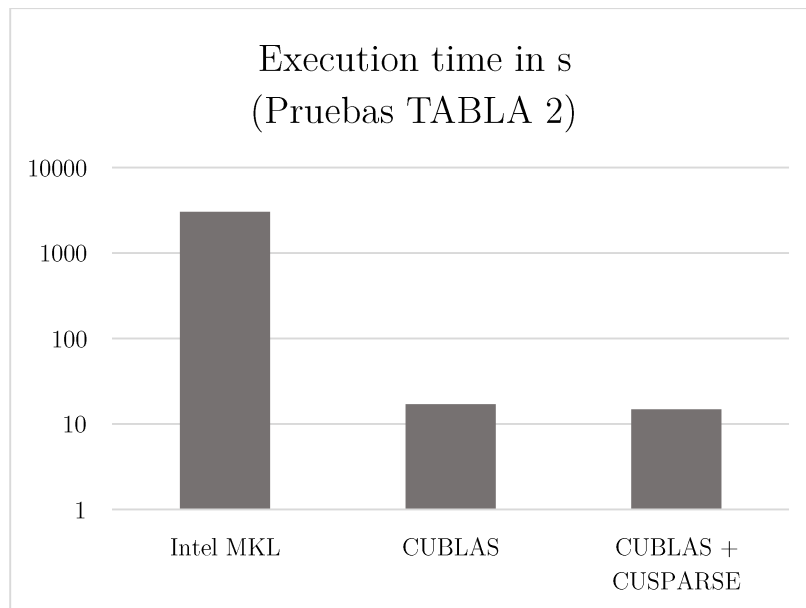
Image 128x128, 200 iterations, 60 projections, SART + accel				
Original	Reconstructed image	MKL	CUBLAS	CUBLAS + CUSPARSE
		Execution time in ms:		
		3057150,651	17113,80816	14912,68301

Tabla 2: Pruebas realizadas comparando las distintas implementaciones realizadas (A)

En la **tabla 2** se ha reconstruido una imagen mediana (128x128 píxeles) con una colección de proyecciones bastante limitada (60 proyecciones, una cada 6 grados de la circunferencia de scanner) y se han ejecutado pocas iteraciones, 200. El resultado es una reconstrucción de una calidad bastante pobre, donde sólo parte de las figuras del fantoma son claramente reconocibles. Pero son los costes lo que interesa de esta prueba, ya que el resultado de imagen es igual en todas las versiones (MKL, CUBLAS, y CUBLAS combinado con CUSPARSE). Los costes evidencian que las versiones basadas en CUDA ofrecen unos costes temporales muy inferiores respecto a los de la versión *multithread* sobre la Intel MKL.

Se han realizado múltiples pruebas incluyendo la versión basada en la Intel MKL, produciendo siempre costes temporales más altos que el resto de alternativas, aunque los resultados siempre ofrecían un valor de varianza algo elevado, probablemente atribuido al uso público del servidor, produciendo retrasos en aquellas tareas de cálculo intensivo sobre CPU.

La siguiente gráfica muestra los resultados obtenidos:



En la misma línea se han realizado más pruebas (**tabla 3 y 4**) enfocadas a comparar el rendimiento de las versiones del algoritmo basadas en CUDA.


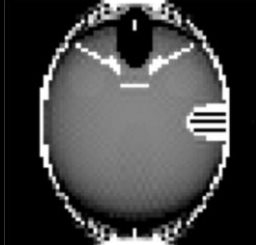
Original	Reconstructed image	CUBLAS	CUBLAS + CUSPARSE
		Execution time in ms:	
		50126,14489	50704,93484

Tabla 3: Pruebas realizadas comparando las distintas implementaciones realizadas (B)



La **tabla 3** no aporta demasiada información acerca de cuál de las versiones basadas en CUDA es más conveniente, pues los costes temporales son muy similares, resultando ligeramente más costosa la versión que realiza el producto matriz-vector con CUSPARSE, la biblioteca especialmente orientada al cálculo con matrices dispersas.

Una nueva prueba comparando las versiones implementadas del algoritmo se representa en la **tabla 4**.

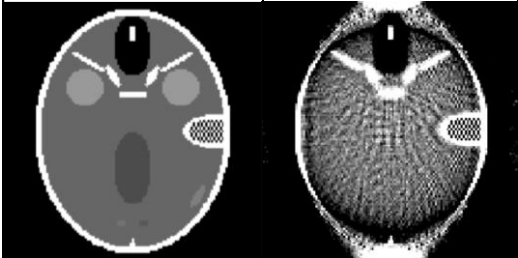
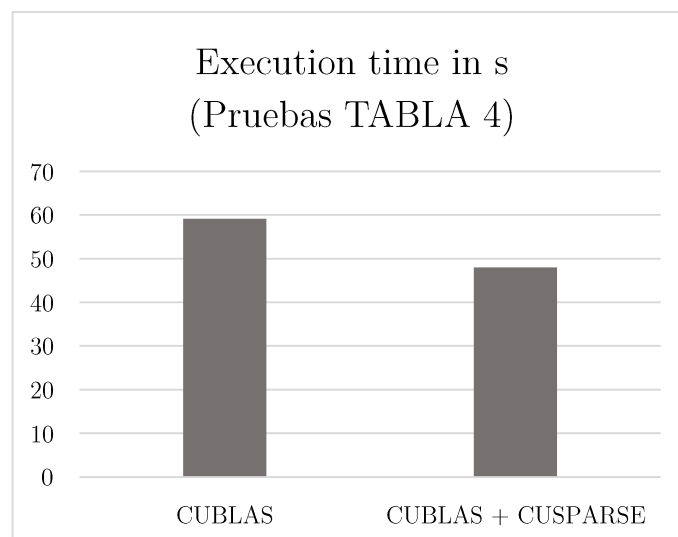
Image 128x128, 800 iterations, 60 projections, SART + accel			
Original	Reconstructed image	CUBLAS	CUBLAS + CUSPARSE
		Execution time in ms:	
		59144,04702	48026,4349

Tabla 4: Pruebas realizadas comparando las distintas implementaciones realizadas (C)

Los resultados quedan representados en la siguiente gráfica:



En este caso se observa que la versión que emplea CUSPARSE en el producto matriz-vector del algoritmo ofrece unos costes claramente inferiores. En esta prueba, la matriz A (tamaño 61500x16384) es tan grande y posee tantos valores nulos como para justificar el despliegue de la biblioteca que optimiza el cálculo con matrices dispersas.

En contraposición, la **tabla 3** hace referencia a pruebas en las que no resulta rentable el uso de la biblioteca CUSPARSE (matriz A de tamaño 184500x4096).

### 6.2.3 El número de iteraciones realizadas

La **tabla 5** muestra la reconstrucción de imagen bajo determinadas circunstancias, variando el número de iteraciones del algoritmo ejecutadas.







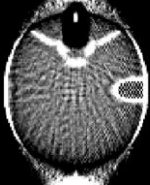



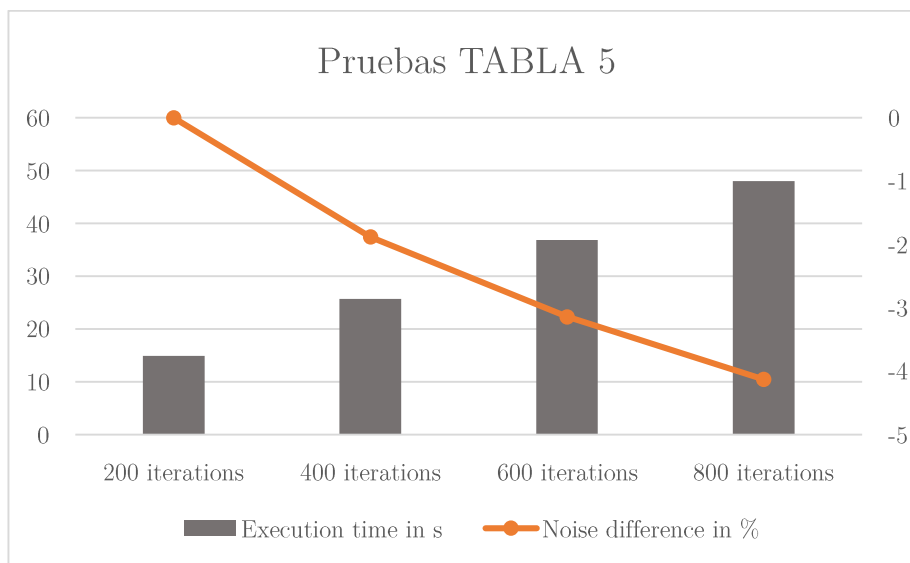
		Image 128x128, 60 projections, SART + accel				
ITERATIONS	Original	200	400	600	800	
Right ear in detail						
Image						
Execution time in ms	-	14912,68301	25678,74885	36852,95296	48026,4349	
Noise difference	-	Base to compare	-1,88%	-3,14%	-4,13%	

Tabla 5: Pruebas realizadas variando el número de iteraciones realizadas

El número de iteraciones incide directamente en la cantidad del ruido eliminada y coste temporal del algoritmo, como se observa en la siguiente gráfica:



Se observa que el coste temporal está linealmente relacionado con el número de iteraciones realizadas, al margen de los costes fijos del algoritmo, como transferencias de datos entre la máquina dispositivo y anfitrión e inicialización de bibliotecas. No es lineal la relación de número de iteraciones respecto al porcentaje de ruido eliminado, más bien se establece una relación logarítmica. Para poder observar de un modo visual la disminución de ruido en cada ejecución, se muestra una vista en detalle de la zona del fantoma que representa el oído derecho.

#### 6.2.4 La cantidad de proyecciones empleada

El número de proyecciones incide directamente en las dimensiones de la matriz  $A$ , debido a que la matriz es de dimensiones  $M \times N$ , donde  $N$  representa el número de píxeles en la imagen y  $M$  corresponde al producto de las proyecciones efectuadas por el número de sensores disponibles, siendo  $A$  una tabla en la que queda reflejada la aportación de cada rayo  $X$  emitido a cada píxel de la imagen. Aumentar el número de proyecciones provoca que aumente linealmente el tamaño de  $M$ , por lo que suceden dos efectos directos:

- Un aumento del número de ecuaciones que componen el sistema  $Au = g$ , lo que permite tener más información a la hora de despejar el vector  $u$  (la imagen a reconstruir), produciendo una mayor calidad de imagen y menor número de

iteraciones del algoritmo. No hay que olvidarse que se pretende usar el mínimo número de proyecciones posible para someter al paciente a la mínima radiación.

- Un aumento del coste computacional de la fórmula SART, pues el coste empírico es de  $O(MN)$ .

La **tabla 6** muestra las pruebas realizadas en la reconstrucción de la imagen variando en cada ejecución el número de proyecciones empleadas de la colección.



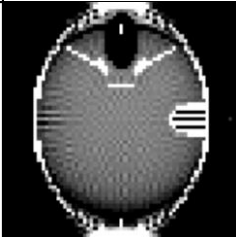

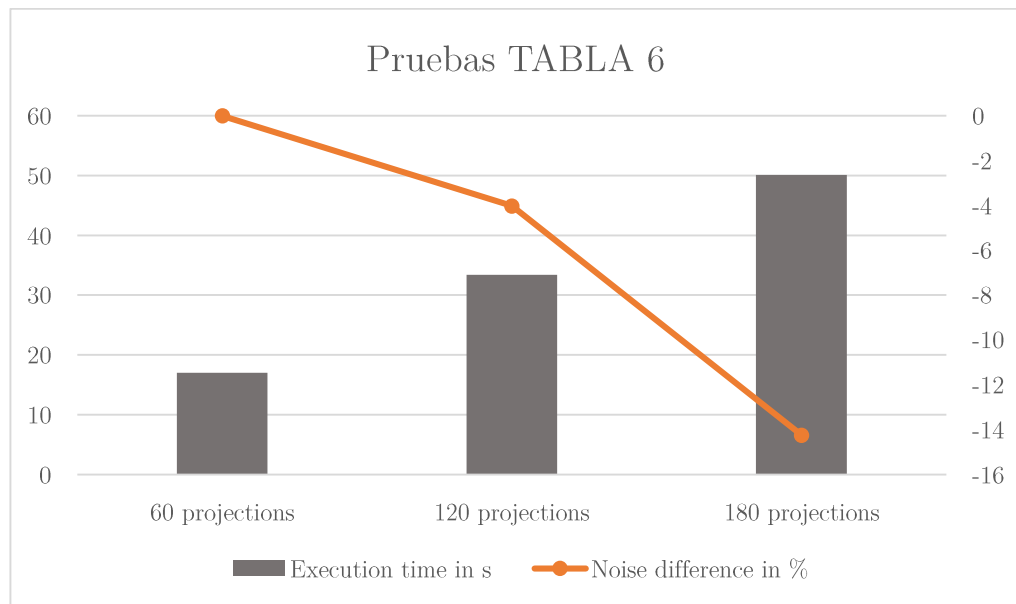
Image 64x64, 400 iterations, SART + accel				
PROJECTIONS	Original	60	120	180
Image				
Execution time in ms	-	17037,26316	33403,74994	50126,14489
Noise difference	-	Base to compare	-4,02%	-14,24%

Tabla 6: Pruebas realizadas variando el número de proyecciones empleadas en la reconstrucción

La siguiente gráfica analiza la tendencia del ruido y los costes temporales en los distintos resultados. Se observa que el nivel de ruido se reduce exponencialmente en reconstrucciones en las que se aumenta en intervalos constantes el número de proyecciones empleados, pues como se observa, triplicando el número de proyecciones se reduce el ruido un 14%. En la reconstrucción basada en 180 proyecciones son fácilmente distinguibles las figuras más sutiles del fantoma.

Es mucho más efectivo reducir el ruido aumentando el número de proyecciones que aumentando el número de iteraciones.

Asimismo se observa que, tal como el coste asintótico  $O(MN)$  deja prever, el coste del algoritmo aumenta linealmente respecto al tamaño de  $M$ .



### 6.2.5 El tamaño de la imagen a reconstruir

Por último, se estudia el comportamiento del algoritmo frente al aumento del tamaño  $N$ , que corresponde al tamaño en píxeles de la imagen.

La **tabla 7** muestra la reconstrucción de imagen con los mismos parámetros para una de tamaño  $64 \times 64$  y otra de  $128 \times 128$ . Las consecuencias de aumentar el tamaño de la imagen a reconstruir son:

- Una mayor carga computacional, según los costes asimptóticos calculados.
- Un aumento del número de ecuaciones del sistema  $Au = g$ , pues el número de variables aumenta ( $N$ ), junto al rango de la matriz  $A$ .

En estas pruebas se ha duplicado el número de píxeles en cada dimensión de la imagen, lo que repercute en que el número de píxeles ( $N$ ) se cuadruplica.



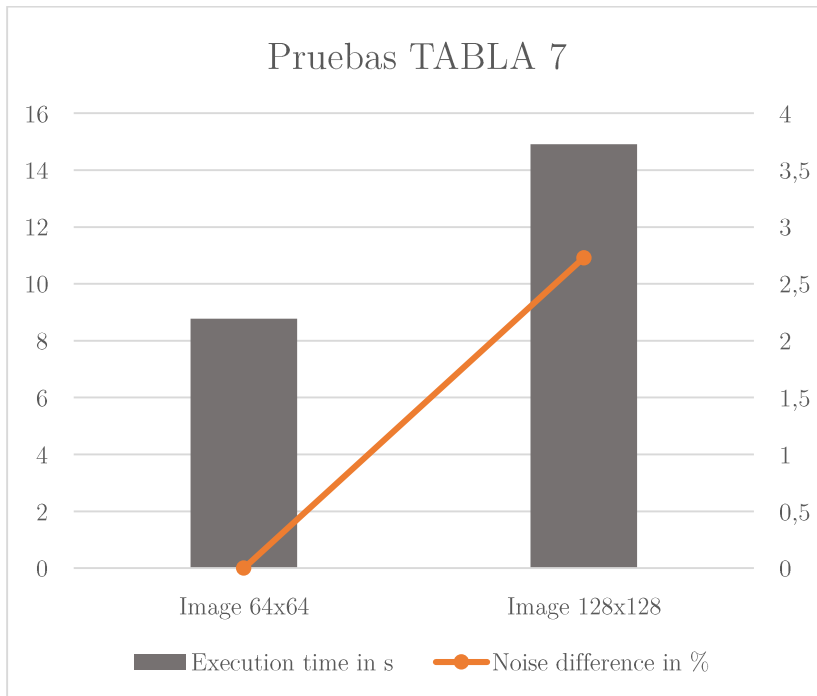
200 iterations, 60 projections, SART + accel		
	64x64	128x128
Image		
Execution time difference	8774,122953	14912,68301
Noise difference	Base to compare	2,73%

Tabla 7: Pruebas realizadas variando el tamaño de la imagen a reconstruir

La siguiente gráfica muestra los resultados:



Se necesitaría más pruebas variando el tamaño de la imagen a reconstruir con el fin de obtener resultados más concluyentes, pero se observa que, cuadruplicando el tamaño de  $N$  y manteniendo invariable el resto de condiciones, los costes temporales no llegan a duplicarse y el ruido aumenta de forma muy moderada. No ha sido un caso aislado, ha ocurrido en más pruebas realizadas empleando distinto número de iteraciones y proyecciones.





# Conclusiones y trabajo futuro

## 7.1 Conclusiones

El objetivo del trabajo es la implementación eficiente del algoritmo WTDM-STF sobre GPU. Se han implementado todos los pasos del algoritmo, así como también un conjunto de pruebas para evaluar el rendimiento y los costes. El único aspecto negativo relacionado es que se ha dejado al margen el filtro de umbral suave del algoritmo al no haber conseguido los excelentes resultados que los autores anuncian, debido principalmente a los escasos detalles que arrojan acerca de su implementación. Tampoco fue posible contactar vía correo electrónico, ya que no respondieron.

Aunque no es sencillo determinar el speed-up concreto alcanzado entre la versión *multicore* (CPU) y las versiones *manycore* (GPU) debido a la enorme variabilidad de tiempos obtenidos en la versión en CPU, sí que se puede afirmar que prácticamente siempre las versiones en GPU son más rápidas (a excepción de algunas pruebas con pocas iteraciones para reconstrucción de imágenes de tamaño 10x10 píxeles). No es extraño observar speed-ups de incluso 200.0 entre la versión que emplea la Intel MKL y la basada en CUBLAS.

También se ha podido observar en los resultados diferencias interesantes entre la implementación puramente en CUBLAS y la implementación que resuelve el producto matriz – vector del algoritmo mediante la biblioteca CUSPARSE, aprovechando que la matriz de dicha operación es de tipo disperso. A partir de cierto tamaño de matriz (imágenes a reconstruir de tamaño 128x128) se obtienen tiempos más rápidos en las versiones que incluyen la operación en CUSPARSE, obteniendo resultados para dicho tamaño de imagen con un speed-up de hasta 1.23.

Hay un momento en el que el número de iteraciones del algoritmo debería ascender con el fin de suplir una disminución en el número de proyecciones de radiación realizadas para someter al paciente a la mínima radiación posible. El objetivo es poder

determinar el número exacto de proyecciones mínimas necesarias para lograr una reconstrucción con un ruido tan reducido que permita visualizar todas y cada una de las figuras representadas en el fantoma. Según las pruebas realizadas, dicho número de proyecciones radica entre 180 y 120 proyecciones (separadas por 2 y 3 grados respectivamente), lo que indica que el ángulo entre proyecciones mínimas realizadas debería ser entre 2 y 3 grados, y un número de iteraciones superior a 600. Estas indicaciones varían para cada uno de los tamaños de imagen, por lo que estas afirmaciones sólo son válidas para imágenes a reconstruir de tamaño 128x128.

Por otra parte, es alentadora la información obtenida en la reconstrucción de imágenes de distintos tamaños, ya que el algoritmo escala realmente bien en dicha dimensión ( $N$ ), pues cuadruplicando el tamaño de imagen, los costes temporales apenas se duplican y el ruido no aumenta demasiado. También anima a seguir trabajando en esta línea averiguar si la correcta y completa implementación del algoritmo WTDM-STF produce unos resultados tan excelentes como se documentan en [2], pues permitiría prescindir de mayor número de proyecciones.

## 7.2 Trabajo futuro

En referencia al algoritmo WTDM-STF, se debería hacer un estudio intensivo relativo de la variable  $\alpha$  del filtro de umbral suave, debido a que en el artículo [2] no se dan detalles acerca de una correcta parametrización, pues únicamente se da un valor inicial y se asegura un excelente comportamiento.

Es necesario ampliar el banco de pruebas a imágenes de mayor tamaño, con el fin de encontrar limitaciones en los distintos sistemas de memoria. Otro aspecto que podría incluirse en las pruebas sería emplear datos provenientes de reconstrucciones originales para basar los resultados en ejemplos reales y no sólo en simulaciones.

Respecto al número de proyecciones requeridas para una reconstrucción de imagen de calidad aceptable, sería recomendable hacer un estudio enfocado especialmente a ello, pues sería útil encontrar el mínimo número de proyecciones necesarias para obtener resultados confiables, siempre teniendo en cuenta los costes admisibles, la colección de proyecciones disponible o la calidad de imagen requerida.

Por otra parte, también sería aconsejable probar nuevas posibilidades en lo que a hardware se refiere. El modelo de GPU empleado en el trabajo es un modelo de altas

prestaciones; sería interesante usar GPUs de un ámbito más cotidiano, buscando hardware que comporte menor coste y valorar su rendimiento. Además sería una buena idea adaptar el algoritmo a la resolución mediante 2 GPUs.



## Bibliografía

- [1] Linet MS, Slovis TL, Miller DL, et al., «Cancer risks associated with external radiation from diagnostic imaging procedures,» *CA Cancer J Clin.*, vol. 62, pp. 75-100, 2012.
- [2] Wei Yu, Li Zeng, «A Novel Weighted Total Difference Based Image Reconstruction Algorithm for Few-View Computed Tomography,» *PLOS ONE*, vol. 9, pp. 1-10, 2014.
- [3] Allan M Cormack, «75 years of Radon Transform,» *Journal of Computer Assisted Tomography*, vol. 16, n° 5, p. 673, 1992.
- [4] Willi A Kalender, «X-ray computed tomography,» *Phys. Med. Biol.*, vol. 51, pp. 29-43, 2006.
- [5] Hounsfield GN, «Computerized transverse axial scanning,» *British Journal of Radiology*, vol. 46, n° 552, pp. 1016-1022, 1973.
- [6] Richard Gordon, Robert Bender, Gabor T. Herman, «Algebraic Reconstruction Techniques (ART) for Three-dimensional Electron Microscopy and X-ray Photography,» *Journal of Theoretical Biology*, vol. 29, pp. 471-481, 1970.
- [7] Govindaraju et al., «Fast Computation of Database Operations using Graphics Processors,» *ACM SIGMOD*, 2004.
- [8] Harris et al., «Physically-Based Visual Simulation on Graphics Hardware,» *Graphics Hardware*, 2002.
- [9] Michael J Flynn, «Some Computer Organizations and Their Effectiveness,» vol. C21, n° 9, 1972.
- [10] James S Kolodzey, «CRAY-1 Computer Technology,» *IEEE Transactions on Components, Hybrids and Manufacturing Technology*, vol. 4, n° 2, pp. 181-187, 1981.

- [11] Lizhe Wang, Jie Tao, Gregor von Laszewski, Holger Marten, «Multicores in Cloud Computing: Research Challenges for Applications,» *Journal of Computers*, vol. 5, n° 6, 2010.
- [12] Lawson CL, Hanson RJ, Kincaid DR, Krogh FT, «Basic Linear Algebra Subprograms for Fortran Usage,» *ACM Transactions on Mathematical Software*, vol. 5, n° 3, pp. 324-325, 1979.
- [13] INTEL Corporation, «Reference Manual for Intel® Math Kernel Library 11.3,» [En línea]. Available: [https://software.intel.com/sites/default/files/managed/c8/36/mklman\\_c\\_11.3.pdf](https://software.intel.com/sites/default/files/managed/c8/36/mklman_c_11.3.pdf).
- [14] Michael Garland, «Sparse Matrix Computations on Manycore GPUs,» *ACM Proceedings of the 45th annual Design Automation Conference*, pp. 2-6, 2008.
- [15] NVIDIA Corporation, «Cuda C Programming Guide,» [En línea]. Available: <http://docs.nvidia.com/cuda/cuda-c-programming-guide>.
- [16] NVIDIA Corporation, «cuBLAS Documentation,» [En línea]. Available: <http://docs.nvidia.com/cuda/cublas/>.
- [17] NVIDIA Corporation, «cuSPARSE Documentation,» [En línea]. Available: <http://docs.nvidia.com/cuda/cuspars/>.
- [18] Andersen AH, Kak AC, «Simultaneous Algebraic Reconstruction Technique (SART): A Superior Implementation of the ART Algorithm,» *Ultrasonic Imaging*, vol. 6, pp. 81-94, 1984.
- [19] Leonid I. Rudin, Stanley Osher, Emad Fatemi, «Nonlinear total variation based noise removal algorithms,» *Physica D*, vol. 60, pp. 259-268, 1992.
- [20] Yuying Li, Fadil Santosa, «A Computational Algorithm for Minimizing Total Variation in Image Restoration,» *IEEE Transactions on Image Processing*, vol. 5, n° 6, pp. 987-996, 1996.
- [21] Hengyong Yu, Ge Wang, «A soft-threshold filtering approach for reconstruction from a limited number of projections,» *Phys. Med. Biol.*, vol. 55, n° 13, p. 3905–3916, 2010.
- [22] Peter M Joseph, «An Improved Algorithm for Reprojecting Rays Through Pixel images,» *IEEE Transactions on Medical Imaging*, vol. 1, n° 3, pp. 192-197, 1982.
- [23] Robert L Siddon, «Fast calculation of the exact radiological path for a three-dimensional CT array,» *Medical Physics*, vol. 12, p. 252, 1985.

- [24] NVIDIA Corporation, «Compressed Sparse Column Format (CSC),» [En línea]. Available: <http://docs.nvidia.com/cuda/cusparse/#compressed-sparse-column-format-csc>.
- [25] NVIDIA Corporation, «cuSPARSE Indexing and Data Formats,» [En línea]. Available: <http://docs.nvidia.com/cuda/cusparse/#matrix-formats>.
- [26] Liubov A. Flores, Vicent Vidal, Patricia Mayo, Francisco Rodenas, Gumersindo Verdú, «CT Image Reconstruction Based on GPUs,» *International Conference on Computational Science*, vol. 18, pp. 1412-1420, 2013.
- [27] Institute of Medical Physics, Erlangen (Germany) and Siemens Healthcare, «Phantom group, Institute of Medical Physics Friedrich-Alexander-University Erlangen-Nürnberg,» 1999. [En línea]. Available: <http://www.imp.uni-erlangen.de/phantoms/>.