UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Paralelización de Algoritmos para Reconstrucción Rápida de Imagen de Resonancia Magnética a partir de Espacio K Submuestreado

Autor:

Elisa Moya Sáez

Tutores:

Carlos Alberola López y Federico Simmross Wattenberg

Valladolid, 21 de Febrero de 2019

Título:	**Paralelización de Algoritmos para Reconstrucción Rápida de Imagen de Resonancia Magnética a partir de Espacio K Submuestreado**
Autor:	**Elisa Moya Sáez**
Tutores:	**Carlos Alberola López y Federico Simmross Wattenberg**
Departamento:	**TSCIT**

TRIBUNAL

Presidente:	**Juan Ignacio Asensio Pérez**
Secretario:	**J. P. Casaseca de la Higuera**
Vocal:	**Luis Miguel San José Revuelta**

Fecha:

Calificación:

RESUMEN

La imagen de resonancia magnética es una potente técnica de imagen médica muy utilizada en la práctica clínica debido a que es capaz de medir las propiedades de los tejidos de forma no invasiva. El principal inconveniente es que el tiempo necesario para adquirir imágenes de alta calidad es elevado. Con el objetivo de agilizar dicho proceso se han presentado en la literatura numerosas técnicas prometedoras como la adquisición de espacios K submuestreados o la utilización de trayectorias no cartesianas. El problema de estas adquisiciones es que trasladan la complejidad a la reconstrucción de la imagen, lo que da lugar a algoritmos con una alta carga computacional. No obstante, dada la naturaleza paralela de los algoritmos de procesado de imagen, las unidades de procesamiento gráfico (GPU), usadas como dispositivos informáticos de uso general, permiten superar esta necesidad computacional, obteniéndose implementaciones más eficientes de dichos algoritmos.

En este Trabajo Fin de Máster se implementan dos algoritmos ampliamente utilizados en la reconstrucción de imagen de resonancia magnética. Estos son, el algoritmo de optimización NESTA, utilizado para la reconstrucción de MRI a partir de espacios K submuestreados, y el algoritmo de la transformada de Fourier no uniforme (NUFFT), utilizado en adquisiciones no cartesianas. Las implementaciones planteadas son programadas mediante el lenguaje de programación OpenCL, ya que tiene la ventaja de permitir implementaciones independientes del tipo de dispositivo y del fabricante del mismo. Concretamente, se realiza especial énfasis en la utilización de GPUs. Al mismo tiempo, los desarrollos se integran en el *framework* OpenCLIPER, el cual facilita la implementación de algoritmos de procesado de imagen en OpenCL, para poder ser utilizados en *pipelines* más complejos de reconstrucción de MRI.

Finalmente, se analizan las imágenes de resonancia magnética reconstruidas para validar ambos algoritmos y demostrar su utilidad en la reconstrucción de MRI. Además, se evalúa el rendimiento de los algoritmos en términos de tiempo de ejecución. Por un lado, este análisis demuestra la mejora obtenida en la implementación planteada del algoritmo NESTA en comparación con la implementación en MATLAB. Por otro lado, aunque con el algoritmo NUFFT no se reducen los tiempos, se plantean posibles optimizaciones del código para trabajos futuros.

PALABRAS CLAVE

Reconstrucción de MRI, NUFFT, NESTA, Espacio K, GPU, OpenCL, OpenCLIPER

ABSTRACT

Magnetic resonance imaging (MRI) is a powerful medical imaging technique widely spread among the clinical routine due to it ability to measure tissue properties non-invasivaly. However, the main drawbacks of this technique are the long times required to acquire good quality images. This is why many research groups are aiming at speeding up the overall acquisition process while maintaining certain image quality standard. During the past decade, several promising MRI acquisition techniques were presented to the literature (i.e., k-space subsampling or the implementation of non-cartesian trajectories). Nonetheless, these new imaging techniques, while decreasing the acquisition time, considerably increase the computation time required to reconstruct the images. In order to overcome this new funnel on the MRI process, we can take advantage of the parallel nature of the images and speed up the image reconstruction through the usage of graphical processing units (GPUs). GPUs, used as general purpose processing devices, can overcome the reconstruction funnel with the implementation of more efficient reconstruction algorithms.

In this Master's Thesis, we implement two algorithms commonly used in the reconstruction of MRI images. Firstly, the optimization algorithm so called NESTA used in k-space subsampling reconstructions. Secondly, the non-uniform fast Fourier transform (NUFFT) used in non-cartesian acquisitions. The proposed algorithm implementations are programmed in OpenCL since it allows functional coding independent of the device and the manufacturer. Especially, we emphasize the use of GPUs. At the same time, we assamble both, the NESTA and the NUFFT codes in the OpenCLIPER framework to allow its usage in complex MRI reconstruction pipelines.

Finally, we analyze the reconstructed MRI images to validate both reconstruction algorithms and demonstrate their feasibility in MRI post-processing. Further, we assess the algorithms' throughput regarding realization time. On one hand, the proposed NESTA implementation improves (i.e., achieve shorter times) MATLAB's code. On the other hand, our NUFFT implementation does no achieve shorter times. Besides that, we propose further coding optimizations as future line of research.

KEYWORDS

MRI reconstruction, NUFFT, NESTA, K-space, GPU, OpenCL, OpenCLIPER

Agradecimientos

En primer lugar, quiero expresar mi gratitud a mis tutores Carlos Alberola López y Federico Simmross Wattenberg por darme la posibilidad de realizar este Trabajo Fin de Máster y por su esfuerzo y tiempo dedicado a guiarme en el transcurso del mismo.

Así mismo, me gustaría agradecer a todos los compañeros del Laboratorio de Procesado de Imagen su amabilidad y ayuda. Muy especialmente a Elena, Santi, Óscar y Dani.

Por último, dar las gracias a mis amigos y familia, que me han apoyado siempre, y sobre todo en los momentos difíciles.

Índice general

Capítulo 1

Introducción

La imagen de resonancia magnética (MRI, de las siglas en inglés *magnetic resonance imaging*) es una potente técnica de diagnóstico que permite obtener imágenes de las características físicas y químicas de un objeto midiendo las señales procedentes de la técnica de resonancia magnética nuclear (NMR, de las siglas en ingles *nuclear magnetic resonance*) [1]. La formación de las imágenes se realiza gracias a los principios de codificación espacial de la información que permiten detectar el objeto a partir de las señales excitadas de resonancia magnética (MR) [1].

El potencial de esta técnica reside en su capacidad de proporcionar imágenes con mucha información sobre los tejidos que componen la anatomía de manera no invasiva. En comparación con otras técnicas tomográficas como PET (tomografía por emisión de positrones) o SPECT (tomografía computerizada por emisión de fotón único), en la MRI no es necesario inyectar isótopos radiactivos en el objeto para generar la señal de MR [1]. Por otro lado, dado que la MRI opera en el rango de la radiofrecuencia (RF), no usa radiación ionizante evitando así los efectos perjudiciales para la salud que esta podría tener sobre el paciente. Todas esta ventajas, convierten a la MRI en una modalidad de imagen médica versátil y, como consecuencia, muy utilizada en al práctica clínica.

No obstante, la MRI también presenta limitaciones importantes. Entre otras, cabe destacar que el proceso de adquisición de señales es inherentemente lento, así como que es una técnica altamente sensible a la presencia de artefactos. En particular, el movimiento introducido tanto por la actividad cardiaca como por la respiración del paciente se traducen en artefactos en las imágenes obtenidas.

1.1 Motivaciones

El desafío, por lo tanto, está en agilizar el proceso de adquisición de las imágenes, cumpliendo con los correspondientes requisitos de calidad. Para ello, se han realizado varios avances desde el inicio clínico de la MRI. Además de la mejora del *hardware*, se han realizado esfuerzos en términos de procesamiento que han dado lugar a algoritmos con mayor carga computacional. Estos se han centrado en secuencias rápidas [2], imágenes paralelas (PI) [3, 4], técnicas de aceleración espacio-temporales [5, 6], procedimientos de muestreo compresivo (CS) [7, 8] y métodos de bajo rango. Desde entonces, se han desarrollado una gran cantidad de algoritmos de reconstrucción de las imágenes para combinar PI con CS y agregar estimación y compensación del movimiento, con el fin de obtener mejores reconstrucciones.

Por otro lado, también con el objetivo de reducir los tiempos de adquisición y minimizar los artefactos, es común utilizar trayectorias no cartesianas para recorrer el espacio K. La principal desventaja de estos métodos es que dificultan la reconstrucción de los datos por no encontrarse estos en una rejilla cartesiana, ya que no se puede aplicar una simple transformada inversa rápida de Fourier (iFFT). Debido a esto, se hace necesario realizar tareas previas adicionales para remuestrear los datos (localizados en frecuencias espaciales arbitrarias) dentro de una rejilla cartesiana. Este método que se conoce como transformada de Fourier no uniforme (NUFFT), conlleva un mayor número de operaciones a realizar y, por consiguiente, una mayor carga computacional.

En general, todos los algoritmos utilizados en la reconstrucción de MRI implican un gran coste computacional, hasta el punto de que la potencia de procesamiento es generalmente el cuello de botella para el desarrollo de los mismos. No obstante, una gran cantidad de las operaciones que se realizan sobre las imágenes son de naturaleza paralela, por lo que las unidades de procesamiento gráfico (GPU), usadas como dispositivos informáticos de uso general, permiten superar esta necesidad computacional, obteniéndose implementaciones más eficientes de dichos algoritmos.

Es cierto que, actualmente, ya existen implementaciones de gran parte de estos algoritmos que obtienen muy buenas prestaciones en cuanto a tiempos de ejecución haciendo uso de plata-

formas de GPUs. No obstante, la mayoría de las soluciones existentes se encuentran desarrolladas en CUDA, lo cual limita su utilización en GPUs de la marca NVIDIA. Debido a esto, creemos que sería interesante disponer de implementaciones de estos algoritmos que puedan ser ejecutadas independientemente del tipo de dispositivo (CPU, GPU, DSP, etc.), así como de la marca concreta del fabricante del mismo (AMD, Intel, NVIDIA, etc.).

1.1.1 Objetivos

El objetivo perseguido con la elaboración de este trabajo es, por lo expuesto en la sección anterior, **la implementación de algoritmos descritos en el estado del arte para la reconstrucción de MRI a partir de espacios K submuestreados mediante técnicas de computación paralela independientes del dispositivo**, para ello se utiliza el lenguaje de programación OpenCL [9]. La elección del lenguaje está motivada por el hecho de que actualmente no existen implementaciones de estos algoritmos en OpenCL, el cual tiene la ventaja de ser un estándar abierto y libre, de forma que su funcionalidad no está limitada a un fabricante concreto de *hardware*. A pesar de la independencia de la implementación planteada con el tipo de dispositivo, en este TFM se realiza especial énfasis en los dispositivos de tipo GPU. Asimismo, dichos algoritmos se desarrollarán en el *framework* OpenCLIPER con un doble objetivo, dotar al *framework* de mayor potencial y aplicabilidad dentro de la comunidad y que los algoritmos puedan ser combinados con otros ya existentes en el *framework* para ser integrados en varios *pipelines* más complejos de reconstrucción de MRI.

En concreto, dos son los algoritmos que se implementan en este TFM:

- **Algoritmo de optimización iterativo NESTA** [10], utilizado para la reconstrucción de MRI a partir un espacio K submuestreado.
- **Algoritmo que implementa la NUFFT**, junto con la programación de procedimientos de visita del espacio K no cartesiano.

1.1.2 Fases y Métodos

Para alcanzar los objetivos planteados, las labores a desarrollar se han llevado a cabo en dos etapas:

I. **Etapa de formación**, centrada en el estudio de los conceptos básicos necesarios para la elaboración de este trabajo. Ha sido realizada de acuerdo a las siguientes fases:

   a) Estudio de los fundamentos de las técnicas de programación paralela en general, y más concretamente en el lenguaje de programación OpenCL [9]. Así mismo, se profundizó en el uso del *framework* OpenCLIPER [11] para facilitar la tarea de programación en GPU.

   b) Estudio de las principales técnicas utilizadas en al actualidad para reconstrucción rápida de MRI.

   c) Comprensión del algoritmo de optimización NESTA [10] a partir del código disponible en MATLAB.

   d) Comprensión del problema de Transformada de Fourier No Uniforme y de las posibles formas de abordar dicho problema. Tras un estudio general de mismo, se profundizó en la implementación realizada en C++ y CUDA en la librería publicada gpuNUFFT [12].

II. **Etapa de diseño, implementación y validación**, llevada a cabo en las siguientes fases:

   a) Programación del algoritmo de optimización NESTA [10] en GPU, utilizando la API (Interfaz de Programación de Aplicaciones) y el lenguaje de programación de OpenCL. Además, se hizo uso del *framework* OpenCLIPER [11] que proporciona herramientas para facilitar las tareas asociadas a este tipo de programación.

   b) Identificación de las limitaciones en la implementación del algoritmo NESTA [10] e introducción de las mejoras necesarias.

   c) Implementación del algoritmo NUFFT en OpenCL, también mediante la utilización del *framework* OpenCLIPER [11].

   d) Validación de las imágenes resultantes obtenidas mediante ambos algoritmos.

   e) Análisis de las prestaciones de los algoritmos implementados en cuanto a tiempos de ejecución.

1.1.3 Medios materiales

Para la realización de este Trabajo Fin de Máster, será necesario el acceso a las herramientas *software* y *hardware* que se detallan a continuación:

Software

- MATLAB R2017b [13]: lenguaje de programación técnico de alto nivel y entorno de desarrollo integrado para el desarrollo de algoritmos, visualización y análisis de datos, y computación numérica.
- KDevelop [14]: entorno integrado de desarrollo de *software*.
- LaTeX [15] : sistema de composición de textos, orientado a la creación de documentos escritos.

Hardware

- PC de sobremesa con las siguientes características:
  - Procesador 8xIntel® Core™ i7-4790 3.60 GHz.
  - 16 GB de memoria RAM.
  - Disco duro de 500 GB de capacidad.
  - GPU AMD Radeon R9 200 Series (Hawaii) con 4 GB de memoria RAM
  - *Device* con versión OpenCL C 1.2
  - *Driver* con versión 375.39
- Acceso a servidores de cálculo disponibles en el Laboratorio de Procesamiento de Imagen (LPI) de la Universidad de Valladolid (UVa).

En cuanto a los datos utilizados para la validación de las implementaciones, se utiliza un de espacio K cartesiano 2D (160x160), 20 *coils* y 16 *frames* para la validación del algoritmo NESTA [10] y un espacio K no cartesiano 3D (272x32x1560) y 20 *coils* para la validación del algoritmo NUFFT. Ambos están disponibles dentro el grupo en el que se realiza el trabajo, el LPI.

Finalmente, cabe destacar que el trabajo se realizó en el Laboratorio 25 de la Escuela Técnica Superior de Ingenieros de Telecomunicación (ETSIT) de la UVa.

1.2 Estructura del documento

El resto del documento se estructura según se detalla a continuación:

En el **capítulo 2** se plantean los principios básicos de la MRI, así como los fundamentos para la reconstrucción rápida de imagen a partir de espacios K submuestreados. Concretamente, se explicará la utilidad en reconstrucción de MRI del algoritmo de optimización NESTA [10]. Además, se detallarán los aspectos más relevantes acerca de las distintas trayectorias (cartesianas y no cartesianas) que se pueden utilizar para recorrer el espacio K y se explicará cuales son los pasos a seguir y las distintas posibilidades para la aplicación de la NUFFT en el caso de adquisiciones no cartesianas.

En el **capítulo 3** se describe con mayor detalle el método utilizado para la programación de los algoritmos implementados. En este caso, consistirá en la introducción de programación en GPU, concretamente en OpenCL. Para ello, en este capítulo se comenzará describiendo las diferencias entre CPU y GPU, remarcando las ventajas e inconvenientes de la programación heterogénea. Posteriormente, se describirá la API y el lenguaje de programación de OpenCL [9], así como el *framework* de apoyo OpenCLIPER [11] destinado a facilitar las tareas asociadas a la programación en este lenguaje.

En el **capítulo 4** se presenta una visión global de la metodología seguida en términos de análisis, diseño e implementación. Para ello, este capítulo se organiza en dos secciones dedicadas a los dos algoritmos implementados. Para cada uno se realizará un análisis del algoritmo concreto a implementar, se indicarán las decisiones de diseño tomadas y se describirán los detalles de implementación en términos del *framework* en el que están integrados los desarrollos.

En el **capítulo 5** se muestran los resultados obtenidos. Concretamente, se analizan las imágenes obtenidas con los distintos métodos implementados, así como las medidas de rendimiento de los mismos.

Finalmente, en el **capítulo 6** se recogen las principales conclusiones extraídas con la elaboración de este Trabajo Fin de Máster y se plantean las posibles líneas de trabajo futuro surgidas.

Adicionalmente, se añade el **apéndice A** en el que se detalla con mayor precisión las funcionalidades ofrecidas por el *framework* OpenCLIPER [11] para la implementación de algoritmos de procesado y reconstrucción de imagen médica. Concretamente, se realizará especial énfasis en las clases usadas para la implementación de los algoritmos planteados en este proyecto.

Capítulo 2

MRI: fundamentos y reconstrucción

En este capítulo, se plantean los principios básicos de la MRI, así como los fundamentos de la reconstrucción rápida de imagen a partir de espacios K submuestreados. Concretamente, se explicará la utilidad en reconstrucción de MRI del algoritmo de optimización NESTA [10]. Además, se detallarán los aspectos más relevantes acerca de las distintas trayectorias (cartesianas y no cartesianas) que se pueden utilizar para recorrer el espacio K y se explicará cuales son los pasos a seguir y las distintas posibilidades para la aplicación de la NUFFT en el caso de adquisiciones no cartesianas.

2.1 Fundamentos de la MRI

La MRI es una modalidad versátil de imagen médica que permite obtener imágenes muy diferentes de la misma localización anatómica en función del protocolo de adquisición de datos utilizado. De esta forma, permite extraer medidas sobre la anatomía, la microestructura de los tejidos, el flujo, la deformación o la perfusión [16]. Además, tomando las precauciones oportunas, es una modalidad inocua, debido a la ausencia de radiación ionizante. Todas estas ventajas han hecho que esta técnica sea cada vez más popular en multitud de diagnósticos clínicos.

No obstante, la obtención de las imágenes es un proceso complejo, ya que en primer lugar es necesario basarse en principios físicos para generar las señales y, posteriormente, manipular y procesar la señal para formar la imagen. De forma más especifica, las fases necesarias hasta ello son:

1. Generación de señal.
2. Detección de señal.
3. Manipulación de la señal una vez ha sido detectada.
4. Procesamiento.

Estas etapas se resumen en el siguiente hilo principal:

$$\boldsymbol{\mu} \rightarrow \boldsymbol{M} \rightarrow \boldsymbol{M_{xy}} \rightarrow S(t) \rightarrow S(\boldsymbol{k}) \rightarrow I(\boldsymbol{x}) \tag{2.1}$$

Donde, $\boldsymbol{\mu}$ es el momento magnético, el cual al ser expuesto a un campo magnético estático da lugar a una magnetización neta $\boldsymbol{M}$. Está se trasfiere a las componentes transversales $\boldsymbol{M_{xy}}$ mediante los pulsos de RF. $S(t)$ es la señal eléctrica detectada en las boninas receptoras de acuerdo con la ley de la inducción de Faraday. $S(\boldsymbol{k})$ es la señal en el espacio k, obtenida gracias a la codificación espacial mediante el uso de gradientes del campo magnético. Y, finalmente, $I(\boldsymbol{x})$ es la imagen deseada obtenida mediante el proceso de reconstrucción [1].

2.1.1 Principios físicos: generación y detección de la señal

Los núcleos atómicos que presentan un número atómico o peso atómico impar poseen un momento angular $\boldsymbol{J}$, que se conoce como espín. En el modelo vectorial clásico, esta magnitud es definida como un vector que representa la rotación de los núcleos atómicos alrededor de un determinado eje [1]. Estos núcleos son objetos cargados en rotación, por lo que generarán a su alrededor un campo magnético conocido como momento magnético $\boldsymbol{\mu}$, el cual está relacionado con el espín mediante una constante física característica de cada tipo de núcleo, denominada constante giromagnética (γ) [1].

$$\boldsymbol{\mu} = \gamma \boldsymbol{J} \tag{2.2}$$

Aunque existen varios núcleos atómicos con esta característica, los núcleos de hidrógeno (^{1}H), compuestos por un único protón, tienen especial interés desde el punto de vista de la NMR, ya que están presentes en abundancia en forma de H_2O en los tejidos biológicos. Los núcleos de hidrógeno se caracterizan por una constante giromagnética de 42.58 MHz/T.

Un conjunto de núcleos del mismo tipo forman un sistema de espines, cuya magnetización neta será nula por las cancelaciones mutuas producidas entre ellos, ya que la dirección del momento magnético $\boldsymbol{\mu}$ es aleatoria debido al movimiento térmico aleatorio. No obstante, bajo la aplicación de un campo magnético estático $\boldsymbol{B_0}$ los núcleos de hidrógeno tienden a alinearse con la dirección de este campo, por lo que aparece una magnetización neta $\boldsymbol{M}$ paralela al mismo [1].

Además, debido al campo magnético estático, los momentos magnéticos de los núcleos precesionan a lo largo de la dirección de $\boldsymbol{B_0}$ (típicamente el eje z) a una frecuencia ω_0 conocida como frecuencia de Larmor y dada por [1]:

$$\omega_0 = -\gamma B_0 \tag{2.3}$$

El movimiento de precesión de un espín sometido a un campo magnético estático se puede ver en la Figura 2.1 [17].

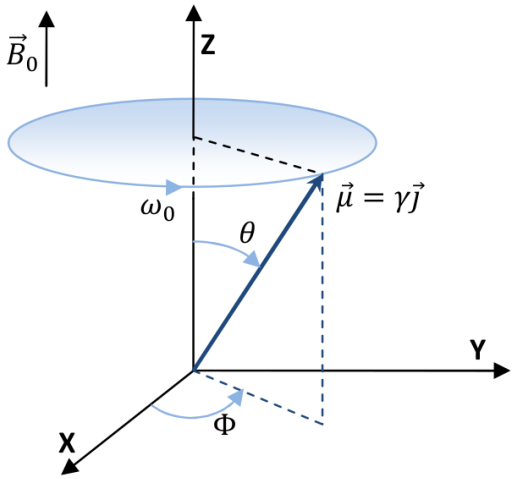

Figura 2.1: Movimiento de precesión de un espín sometido a un campo magnético estático $\boldsymbol{B_0}$ [17].

El vector de magnetización neta puede ser descompuesto en sus tres componentes de la forma $\boldsymbol{M} = [M_x, M_y, M_z]$. Bajo la presencia de un campo magnético estático $\boldsymbol{B_0}$ en la dirección longitudinal z, las componentes transversales (M_x^0 y M_y^0) serán nulas, ya que la precesión de los momentos magnéticos tienen fases aleatorias. No obstante, la componente longitudinal será no nula.

Para que las componentes transversales del vector de magnetización neta sean no nulas, se requiere la aplicación de un segundo campo magnético en la dirección transversal que denominaremos $\boldsymbol{B_1}(t)$, nótese que este segundo campo no será estático sino dependiente del tiempo. Además, es necesario que este campo rote a una frecuencia igual a la frecuencia de Larmor, de forma que los espines entren en resonancia y se pueda producir una transferencia eficiente de energía [18]. La frecuencia de precesión de los núcleos de hidrógeno bajo la presencia de un campo magnético estático de $B_0 = 3\text{T}$ es aproximadamente 128 MHz, dentro del rango de la radiofrecuencia (RF), por lo que este campo se conoce normalmente como pulso de RF.

La aplicación de este segundo campo provocará que el vector de magnetización neta del sistema de espines $\boldsymbol{M}$ se mueva respecto a la posición de equilibro, paralela a $\boldsymbol{B_0}$, siguiendo una trayectoria espiral. La variación con el tiempo de $\boldsymbol{M}$ bajo la aplicación del pulso de RF, es descrita mediante la *ecuación de Bloch* [1]:

$$\frac{d\boldsymbol{M}}{dt} = \gamma \boldsymbol{M} \times \boldsymbol{B} - \frac{M_x\vec{i} + M_y\vec{j}}{T_2} - \frac{(M_z - M_z^0)\vec{k}}{T_1} \tag{2.4}$$

Donde M_z^0 es el valor de equilibrio de $\boldsymbol{M}$ bajo la presencia únicamente del campo magnético estático $\boldsymbol{B_0}$, mientras que T_1 y T_2 son constantes temporales que caracterizan el proceso de relajación del sistema de espines.

Después a de la aplicación del pulso de RF, los espines liberan la energía absorbida y el vector de magnetización neta vuelve gradualmente a la posición de equilibrio en un proceso conocido como relajación [1]. Esta energía liberada es captada por las bobinas receptoras en forma de voltaje. El proceso de relajación caracteriza por tres efectos:

- **Precesión libre:** movimiento de precesión del vector de magnetización neta $\boldsymbol{M}$ alrededor del campo magnético estático $\boldsymbol{B_0}$.
- **Relajación longitudinal o espín-red:** debida a un intercambio de energía entre los espines y el ambiente circundante. Provoca la recuperación de la magnetización longitudinal.
- **Relajación transversal o espín-espín:** debida a la perdida de coherencia de fase de la magnetización del sistema de espines. Da lugar a la perdida de las componentes transversales del vector de magnetización neta.

La relajación longitudinal queda caracterizada en cada tejido por T_1 (tiempo de relajación longitudinal), mientras que la relajación transversal queda caracterizada por T_2 (tiempo de relajación transversal). Estos parámetros, junto con la densidad de protones, son propios de cada tejido y el valor de los píxeles de la imagen de resonancia magnética depende de ellos.

2.1.2 Formación de la imagen de MR

Hasta ahora se ha explicado como generar la señal de resonancia magnética, pero para poder obtener la señal en el espacio K (y generar la imagen), es necesario localizar espacialmente la información, para lo que se utilizan los gradientes de campo magnético. Estos gradientes permiten, en primer lugar, la excitación selectiva, procedimiento para seleccionar únicamente un determinado *slice* generando una señal que corresponda exclusivamente a dicho *slice*. Por otro lado, se utilizan los gradientes para crear dos métodos de codificación espacial:

- **Codificación de frecuencia:** con la aplicación de gradientes de codificación en frecuencia en la dirección x (G_x), se consigue que la tasa de precesión de los espines excitados dependa linealmente con la localización espacial en esa dirección [1].
- **Codificación de fase:** considerando un gradiente aplicado en la dirección y (G_y), se consigue que pasado un tiempo el vector de magnetización neta tenga un desfase dependiente de la posición en esa dirección [1].

La señal codificada espacialmente detectada por las bobinas receptoras tiene la forma de una integral de Fourier. Es decir, consiste en el espacio K de la imagen. Para obtener datos discretos de la señal es necesario muestrear las distintas frecuencias espaciales siguiendo una

trayectoria en el espacio K. Tradicionalmente el patrón de muestreo del espacio K era diseñado para cumplir con el criterio de Nyquist, el cual depende de la resolución y del *field of view* (FOV) de la imagen. Esto es debido a que la violación del criterio de Nyquist causa artefactos en las reconstrucciones lineales [19]. Aunque existe libertad para el diseño de las trayectorias en el espacio K, la trayectoria más popular consiste en líneas en una rejilla cartesiana, ya que en ese caso (y si se ha cumplido con el criterio de Nyquist en el muestreo), la reconstrucción es simple aplicando la inversa de la transformada rápida de Fourier (iFFT) [19].

No obstante, la aplicación de todo este proceso es lenta debido a limitaciones físicas relacionadas con la velocidad a la que pueden variar los gradientes. Esto se traduce en que el tiempo necesario para adquirir las imágenes sea elevado. Para mitigar este problema, con el paso del tiempo se han realizado numerosos esfuerzos orientados a acelerar el proceso de adquisición sin degradar la calidad de la imagen. Entre las posibles soluciones cabe destacar el uso de secuencias rápidas, como por ejemplo EPI (*echo-planar imaging*) [2], la adquisición de espacios K submuestreados bajo la idea de la posible redundancia de los datos de MRI, o la utilización de trayectorias no cartesianas que recorran de forma más eficiente el espacio K y, por consiguiente, sean menos susceptibles a la presencia de artefactos permitiendo así un mayor submuestreo.

2.2 Reconstrucción de MRI a partir de espacios K submuestreados

Como se ha mencionado, una de las formas de acelerar el tiempo de adquisición de la imagen de MR es reducir el número de muestras adquiridas en el espacio K, esto es adquirir espacios K submuestreados [16]. Estas técnicas se basan en la idea de que los datos adquiridos en el espacio K contienen información redundante. La redundancia puede ser creada mediante el diseño, por ejemplo usando diferentes bobinas receptoras; o puede extraerse de las propiedades del modelo de la señal [19]. En esta línea de han desarrollado distintos métodos, tanto de propósito general como para el caso cardiaco. Entre dichos procedimientos se encuentran:

- **Adquisición de imagen en paralelo:** uno de los objetivos de estas técnicas es reducir el tiempo de adquisición de las imágenes mediante el submuestreo del espacio K y la obtención simultánea de datos a través de múltiples bobinas. El submuestreo reduce los

tiempos, mientras que el uso de las bobinas permite la reconstrucción de la imagen final. Entre ellas destacan las técnicas de SENSE y de GRAPPA, completamente establecidas ya en los escáneres de uso clínico [3, 4].

- **Aprovechamiento de la redundancia temporal de los datos cardiacos:** entre estas técnicas se encuentra BLAST k-t [5]. Su fundamento es la aplicación de un muestreo poco denso en el espacio K-t y la resolución del potencial solapamiento en el espacio recíproco x-f a partir de la correlación de la señal, aprendida ésta por medio de datos de entrenamiento de baja resolución [16].

- **Técnicas de CS:** estas técnicas se basan en el principio de que las señales adquiridas presentan un contenido de información muy redundante, lo que implica que el número de muestras necesario para recuperar prácticamente la totalidad del contenido pueda ser mucho menor que el marcado por el límite de Nyquist. En concreto, CS indica que cuando una imagen puede representarse por un número pequeño de coeficientes distintos de cero en algún dominio transformado, entonces se puede recuperar a partir de un número pequeño de medidas incoherentes.

2.2.1 Algoritmo NESTA aplicado a reconstrucción de MRI

La reconstrucción CS de datos de MRI submuestreados es generalmente planteada como un problema de optimización con restricciones, dado por:

$$\underset{\boldsymbol{m}}{\text{minimize}} \quad ||\boldsymbol{\Phi m}||_{l1} \quad \text{s.t.} \quad ||\boldsymbol{y} - \boldsymbol{E m}||_{l2}^2 < \epsilon \tag{2.5}$$

Dónde, $\boldsymbol{\Phi}$ es una transformación *sparse*, $\boldsymbol{m}$ es la imagen de MRI reconstruida, dispuesta como un vector columna, $\boldsymbol{y}$ son los datos del espacio k submuestreados, dispuestos también como un vector columna. El operador de codificación $\boldsymbol{E}$, consiste en la multiplicación de la imagen en cada bobina por los mapas de sensibilidad (en el caso de adquisiciones *multicoil*), la transformada de Fourier espacial, y la aplicación de una máscara o trayectoria de submuestreo que mantiene solo las posiciones del espacio K requeridas (idéntica para todas las bobinas en adquisiciones *multicoil*). Por último, el umbral ϵ establece el nivel de ruido en la adquisición [20].

Mediante el uso de multiplicadores de Lagrange se puede convertir el problema de optimización con restricciones en un problema de optimización sin restricciones de la siguiente forma:

$$\underset{\boldsymbol{m}}{\text{minimize}} \quad \frac{1}{2}||\boldsymbol{y} - \boldsymbol{E}\boldsymbol{m}||_{l2}^2 + \lambda||\boldsymbol{\Phi}\boldsymbol{m}||_{l1} \tag{2.6}$$

Dónde, la elección del valor del parámetro λ establece el compromiso entre la fidelidad de los datos y lo *sparse* que sea la solución [20]. Para la resolución del problema la transformación *sparse* $\boldsymbol{\Phi}$ se debe fijar de antemano a partir del conocimiento previo sobre la estructura de los datos. Por ejemplo, en adquisiciones dinámicas de MRI, en las que se espera que las imágenes consecutivas sean muy similares, se pueden aplicar diferencias temporales para obtener una representación *sparse*, esto es conocido como variación total temporal (tTV). No obstante, existen otras transformaciones que también ofrecen una representación *sparse* de la imagen. Como por ejemplo, la transformada *Wavelet* o la variación total espacial, tanto en su versión isotrópica como anisotrópica [21].

Uno de los algoritmos más utilizados para abordar este problema es el algoritmo NESTA [10]. Esto es debido a tres ventajas principales que ofrece este método:

- **Velocidad:** NESTA es un algoritmo iterativo en el que cada iteración es descompuesta en tres pasos que involucran pocas operaciones matriz-vector. La rápida velocidad de convergencia de este algoritmo hace que sea muy utilizado para resolver problemas de gran escala.
- **Precisión:** el algoritmo NESTA depende de unos pocos parámetros que mantienen una relación directa con la precisión deseada. Es decir, modificando estos parámetros podemos controlar que precisión deseamos en la solución.
- **Flexibilidad:** este algoritmo es capaz de resolver con la misma eficiencia tanto problemas de minimización L1 como que problemas de minimización de variación total (TV).

Debido a esto, NESTA es el algoritmo elegido para ser implementado en este Trabajo Fin de Máster como método de reconstrucción de MRI.

2.3 Trayectorias no cartesianas

Aunque las trayectorias cartesianas son las más populares como patrón para muestrear el espacio K, existen otras trayectorias que recorren el espacio K de forma no cartesiana. Entre otras, cabe destacar el muestreo mediante líneas radiales (ver Figura 2.2 B) o mediante trayectorias espirales (Figura 2.2 C). Las adquisiciones radiales son menos susceptibles a la presencia de algunos artefactos en la imagen, como los artefactos de movimiento, en comparación con las trayectorias cartesianas. Gracias a esto pueden tener una mayor tasa de submuestreo. Por otro lado, las trayectorias espirales hacen un uso más eficiente de los gradientes en el propio *hardware*, lo que permite obtener tiempos de adquisición menores [19].

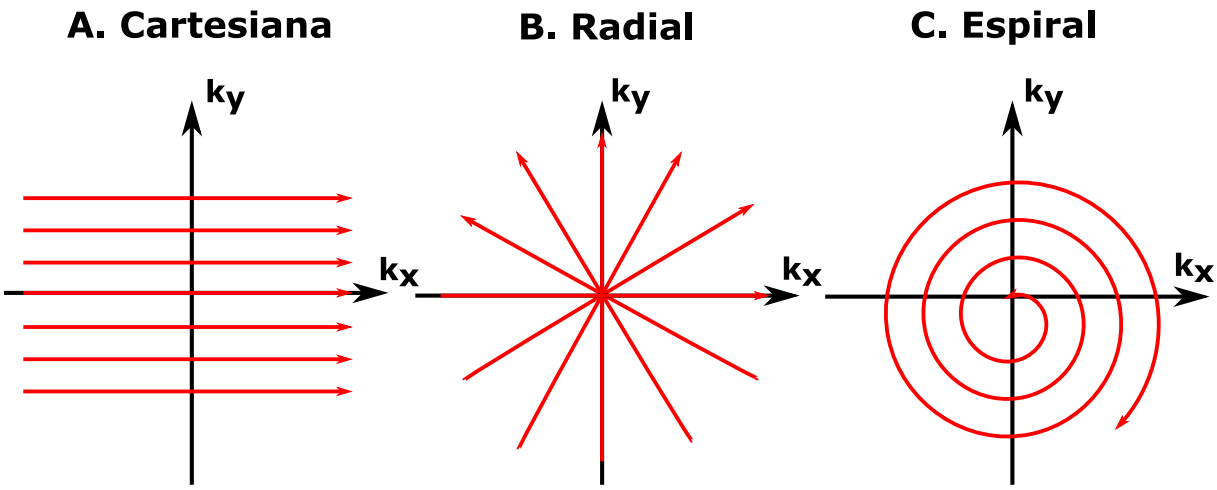

Figura 2.2: Trayectorias comunes (cartesianas y no cartesianas) para recorrer el espacio K.

El problema de usar trayectorias no cartesianas para muestrear el espacio K es que para realizar la reconstrucción de la imagen no se puede aplicar una simple iFFT. De esta forma, la reconstrucción de datos no cartesianos es más compleja y se puede abordar de distintas maneras. La primera posibilidad es adquirir los datos no cartesianos de forma que después se puedan aplicar métodos de reconstrucción con conocimiento a priori, como la reconstrucción con proyecciones [22]. El inconveniente de la aplicación de estos métodos es que es necesario asumir ciertos compromisos en la adquisición de los datos. La segunda opción es demodular punto a punto los datos no cartesianos con una reconstrucción de fase conjugada, el problema en este caso se encuentra en la lentitud del método [22]. Por último, la tercera posibilidad es remuestrear los datos situados en frecuencias espaciales arbitrarias en una rejilla cartesiana de forma previa a

la aplicación de la FFT. Este método se conoce como transformada rápida de Fourier no uniforme (NUFFT). A pesar de que la aplicación de la NUFFT implica tener que realizar tareas adicionales previas a la FFT, es el que mejores prestaciones ofrece, por lo que es el más utilizado.

2.4 La NUFFT: concepto y soluciones existentes

La NUFFT se utiliza cuando los datos adquiridos en el espacio K se encuentran en frecuencias espaciales arbitrarias, por lo que no se puede aplicar una simple iFFT para la reconstrucción de la imagen. De forma general, la principal operación que se realiza, conocida como *gridding*, consiste en remuestrear los datos dentro de una rejilla cartesiana para poder aplicar después la iFFT [22]. Existen distintos métodos para realizar el remuestreo de los datos del espacio K, los principales son:

- **Interpolación basada en la rejilla:** consiste en estimar el valor de cada punto de la rejilla cartesiana, a partir de los datos del espacio K no cartesianos inmediatamente circundantes [22]. Un ejemplo de la aplicación de este método se puede observar en la Figura 2.3 A, dónde para calcular el valor del punto rojo resaltado de la rejilla se utiliza el valor de los cuatro puntos circundantes resaltados en verdes.
- **Interpolación basada en los datos:** se suma la contribución de cada dato del espacio K no cartesiano en los puntos circundantes de la rejilla cartesiana en los que influye. Para ello, cada muestra es convolucionada con un *kernel* cuya forma es elegida estratégicamente para extender cada punto a los vecinos de la rejilla [22]. En este caso la aplicación del método se puede observar en la Figura 2.3 B, donde se realiza una expansión del punto resaltado en verde mediante el kernel de convolución representado en línea discontinua.
- **Interpoladores óptimos:** similar al anterior, pero en este caso cada punto del espacio K no cartesiano es convolucionado por un *kernel* de convolución distinto para conseguir una alta fidelidad en la reconstrucción.

La interpolación basada en la rejilla tiene la ventaja de ser fácil de implementar, pero no es eficiente en términos de relación señal a ruido (SNR). Esta desventaja es solventada con la interpolación basada en los datos. Por otro lado, los interpoladores óptimos permiten realizar

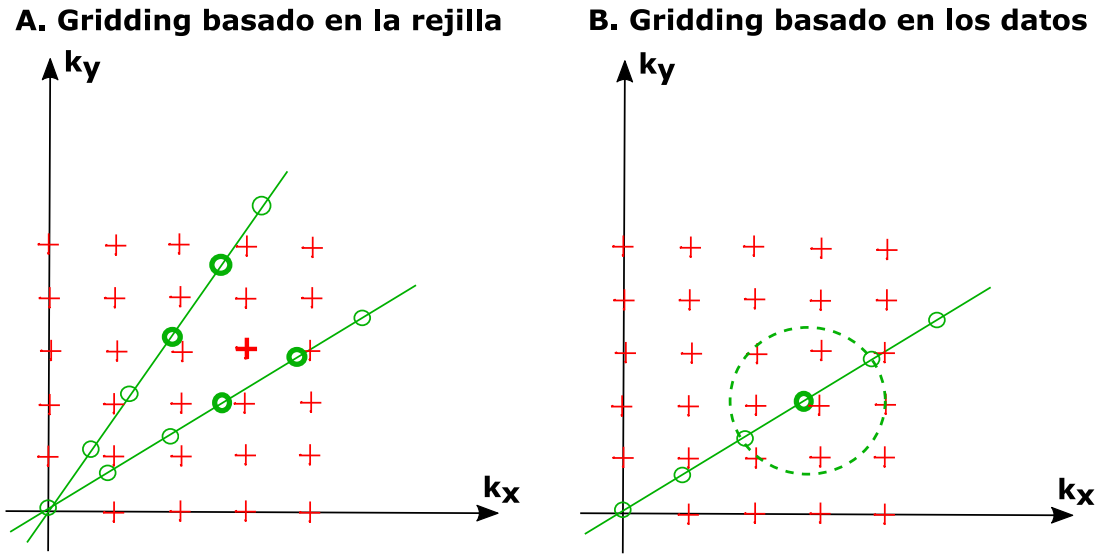

Figura 2.3: Métodos de realizar la interpolación de datos no cartesianos a una rejilla cartesiana [22].

reconstrucciones de alta fidelidad, pero suponen un aumento considerable de la carga computacional. Debido a estas ventajas e inconvenientes, es común adoptar una solución de compromiso con la interpolación basada en los datos [22].

Si nos centramos en la interpolación basada en los datos (por ser la más utilizada), tras la construcción del espacio K cartesiano se podría realizar la reconstrucción directamente mediante la iFFT. No obstante, en ese caso la imagen reconstruida presentará una serie de artefactos (*blurring* y *aliasing*) debido al patrón de las muestras en el espacio K y estará apodizada debido a la multiplicación de la misma por la transformada inversa del *kernel* de convolución [22]. Esto da lugar a que se tengan que realizar más operaciones adicionales al *gridding*. De forma específica, la aplicación de la NUFFT consiste en los siguientes pasos:

1. **Compensación de la densidad:** en las trayectorias no cartesianas el patrón de submuestreo no es uniforme por todo el espacio K. Debido a esto, el primer paso es ponderar el espacio K para compensar la mayor densidad de muestreo en unas zonas en comparación con otras. La ponderación está determinada por la función de compensación de densidad (unos pesos específicos) que depende de la trayectoria utilizada.

2. **Gridding:** cada punto no cartesiano es convolucionado con un *kernel* de convolución finito, y el resultado es muestreado y sumado en la rejilla cartesiana. En la Figura 2.4 se

representa un ejemplo de la aplicación de este método con un punto de una trayectoria no cartesiana. Los aspectos más importantes a tener en cuenta en este paso son la elección del *kernel* de convolución y la densidad de la rejilla de reconstrucción. Es necesario que exista un sobremuestreo de la rejilla cartesiana con respecto al espacio imagen para eliminar los artefactos de *aliasing* debido a los lóbulos secundarios de la iFFT del *kernel* de convolución. Se define el factor de sobremuestreo como el ratio entre el tamaño de la rejilla cartesiana y el tamaño de la imagen.

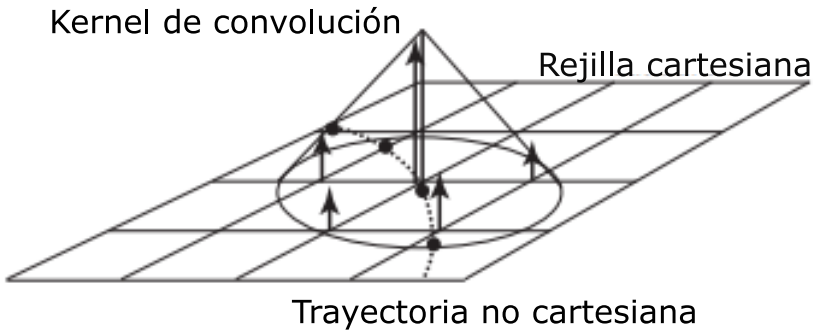

Figura 2.4: *Gridding* basado en los datos para construir la rejilla cartesiana [22].

3. **iFFT:** se aplica la transformada inversa de Fourier para pasar la rejilla del espacio K cartesiano al espacio imagen.

4. **Recorte de la imagen:** debido al factor de sobremuestreo de la rejilla. Este factor de sobremuestreo da lugar, al hacer la iFFT, a una imagen del doble de puntos al deseado.

5. **Deapodización:** corrección que se realiza ponderando la imagen por la inversa de la función de apodización, esto es, la iFFT del *kernel* de convolución.

Los pasos mencionados se corresponden con la aplicación del operador $NUFFT^H$, utilizado para pasar de los datos en el espacio K a los datos en el espacio imagen, esto es la reconstrucción de la imagen.

Capítulo 3

Fundamentos de la computación heterogénea CPU-GPU

En este capítulo se pretende proporcionar una visión general de los principios básicos en los que se basa la computación heterogénea. Concretamente, se realizará especial énfasis en la programación con propósito general en GPU, así como en los lenguajes de programación existentes actualmente para llevarlo a cabo. Por último, se describe con más detalle el lenguaje de programación OpenCL, dado que ha sido el utilizado para llevar a cabo la implementación que en este trabajo se plantea.

3.1 Introducción a la computación Heterogénea

Actualmente, los entornos de computación se presentan cada vez más heterogéneos. La presencia en las arquitecturas de ordenadores de microprocesadores multinúcleo, unidades de procesamiento central (CPUs), procesadores digitales de señal (DSPs), dispositivos lógicos programables (FPGAs) y unidades de procesamiento gráfico (GPUs) amplían las oportunidades de programación para los desarrolladores de software [23]. Debido a esto, aparece el concepto de computación heterogénea, que hace referencia a sistemas que usan más de un tipo procesador. Son sistemas que ganan en rendimiento no por añadir mayor número de procesadores del mismo tipo, sino por añadir procesadores distintos, ya que normalmente los distintos tipos de procesadores incorporan capacidades de procesado especializadas para realizar tareas particulares. Es importante permitir que los desarrolladores de software aprovechen al máximo estas plataformas de procesamiento heterogéneas. No obstante, crear aplicaciones para una gama amplia de

arquitecturas es un desafío para los desarrolladores, que tienen encontrar la concurrencia en su problema y expresarla a nivel de *software* de forma que el programa resultante tenga el rendimiento deseado [9].

Generalmente, las aplicaciones se caracterizan por poseer una combinación variada en cuanto a las características de la carga de trabajo. Por otro lado, no hay ninguna arquitectura que sea la mejor para ejecutar todas las clases de cargas de trabajo. Por ejemplo, las tareas de control intensivo tienden a ejecutarse más rápido en una CPU, mientras que las tareas de uso intensivo de datos independientes tienden a ejecutarse más rápidamente en arquitecturas vectoriales como GPUs, donde la misma operación se aplica a múltiples elementos de datos al mismo tiempo. Esto da lugar, a que la utilización de procesadores de distinto tipo pueda aumentar las prestaciones.

Surge en este contexto la computación paralela, como forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, operando sobre el principio de que problemas grandes, a menudo se pueden dividir en otros más pequeños, que pueden ser resueltos simultáneamente (en paralelo). Tanto las CPUs, que progresan aumentando el número de núcleos, como las GPUs, que han evolucionado desde dispositivos de renderizado a procesadores paralelos programables, tienden a aumentar el paralelismo como principal vía para mejorar el rendimiento [9]. Dentro de este paradigma se encuentra la computación acelerada por GPU, que hace uso de una GPU, utilizada como un dispositivo informático de propósito general, junto a una CPU, para acelerar el funcionamiento de las aplicaciones con elevada carga computacional.

3.2 Computación paralela en GPU

En esencia, la CPU y la GPU son similares: circuitos integrados que realizan cálculos matemáticos con números binarios. No obstante, su diseño es distinto. Debido a esto, se pueden utilizar para la ejecución de distintas partes de la aplicación. Como se puede observar en la Figura 3.1, la CPU se compone de uno o pocos núcleos muy complejos, por lo que se utiliza para el procesamiento en serie de operaciones con una elevada cantidad de flujos de control, mientras que la GPU se compone de cientos o miles de núcleos sencillos que permiten la ejecución de operaciones con flujos de control simples pero con un elevado grado de paralelismo [24]. Por otro lado, en la CPU cualquier operación toma típicamente del orden de 20 ciclos entre entrar y salir

de la pipeline, mientras que en la GPU una operación puede tardar miles de ciclos de principio a fin. Como consecuencia, la latencia es menor en la CPU, pero en la GPU el paralelismo proporciona un mayor *throughput* [25].

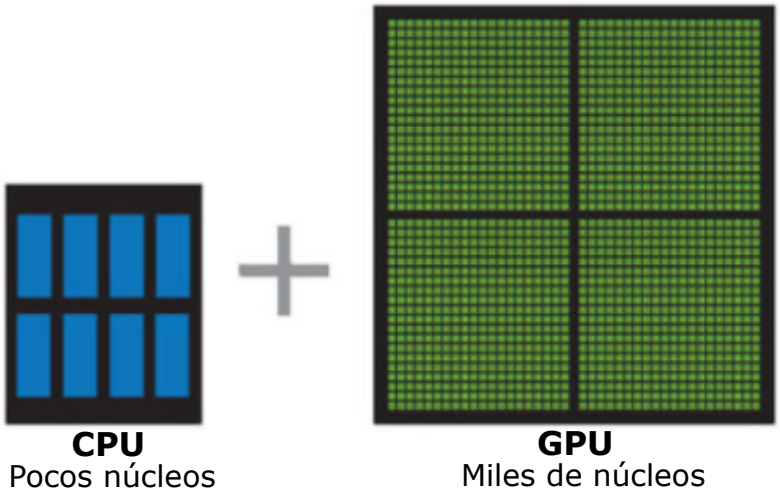

Figura 3.1: Diseño de una CPU vs. GPU [24].

La computación acelerada por GPU permite asignar a la GPU el trabajo de las partes de la aplicación donde la computación es más intensiva, mientras que el resto del código se ejecuta en la CPU. Desde la perspectiva del usuario,las aplicaciones se ejecutan de forma mucho más rápida.

3.2.1 Lenguajes para la computación en GPU

Existen principalmente dos lenguajes para la programación en GPU con propósito general (GPGPU). Estos son CUDA, propietario de NVIDIA, y OpenCL, estándar abierto de Khronos Group [26]. La principal diferencia entre CUDA y OpenCL radica en el hecho de que dado que el primero es propietario, solo está disponible en tarjetas gráficas de la marca comercial NVIDIA. No obstante, OpenCL presenta una ventaja importante con respecto a CUDA y otros lenguajes de programación de GPUs propietarios, ya que, al ser un estándar abierto, se pueden eliminar las restricciones de *hardware* pudiendo ser utilizado en un entorno multiplataforma.

3.3 El lenguaje de programación OpenCL

OpenCL es un *framework* de computación heterogénea gestionado por el consorcio tecnológico Khronos Group [9]. Permite el desarrollo de aplicaciones con distintos niveles de paralelismo que aprovechen el potencial de las plataformas de procesamiento heterogéneas como CPUs, GPUs y otros procesadores. Para ello, cuenta con una API y un lenguaje de programación (OpenCL C). Como ya se ha mencionado, las principal ventaja de OpenCL en comparación con otros lenguajes de programación de GPU, está en la independencia del código generado con el fabricante del *hardware* subyacente sobre el que se ejecute, permitiendo el desarrollo de *software* portable [9].

Por otro lado, el principal problema de OpenCL es el entorno de trabajo complejo inherente a la programación asíncrona. Este es uno de los motivos por el que en este trabajo se decide usar como apoyo el *framework* OpenCLIPER, cuyo objetivo es simplificar las tareas de programación asociada a OpenCL para permitir que los desarrolladores se centren en el desarrollo de algoritmos propiamente dicho. Una descripción más detallada de las propiedades y funcionalidades de OpenCLIPER se puede encontrar en el Apéndice A.

3.3.1 Arquitectura de OpenCL

La especificación de OpenCL define la siguiente jerarquía de *modelos* [9]:

- **Modelo de plataforma:** la plataforma esta formada por un *Host* (típicamente la CPU), encargado de gobernar todo el sistema, por lo que está conectado a uno o más dispositivos de OpenCL. Cada dispositivo se divide en uno o más *Compute Units (CUs)* que a su vez se dividen en uno o más *Processing Elements (PEs)*. Los cálculos en un dispositivo se producen dentro de los PEs [9]. La organización de un sistema OpenCL según el modelo de plataforma se puede observar en la Figura 3.2.

- **Modelo de ejecución:** en la ejecución de un programa en OpenCL se distinguen dos partes: los *kernels*, que son ejecutados por los dispositivos, y el *programa host*, que es ejecutado por el *host* y se encarga de gestionar la ejecución de los *kernels* [9]. Para ello, el

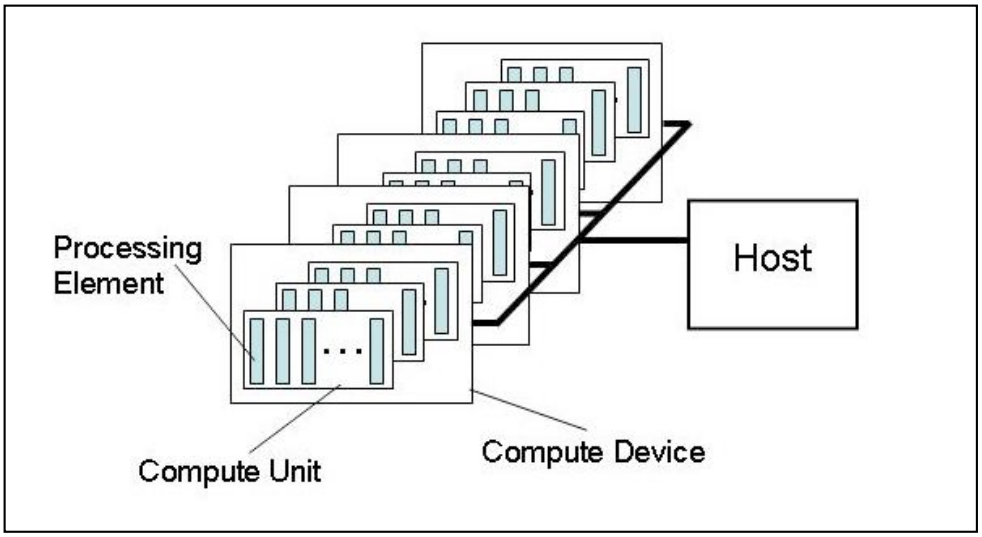

Figura 3.2: Modelo de plataforma de OpenCL [9].

host define un contexto (creado y manipulado con funciones de la API de OpenCL) que incluye los siguientes recursos:

- **Dispositivos:** colección de dispositivos de OpenCL que ejecutan los *kernels*.
- ***Kernels*:** funciones escritas en lenguaje OpenCL que se ejecutan en los dispositivos.
- **Cola de comandos:** estructuras de datos para coordinar la ejecución de los *kernels* en cada dispositivo. Es necesario destacar que deberá existir una cola de comandos por cada dispositivo utilizado.
- **Objetos de programa:** conjunto de *kernels* de un mismo fichero fuente.
- **Objetos de memoria:** contienen los datos que pueden ser operados por instancias de un *kernel*. Son visibles tanto por el *host* como por los dispositivos. En OpenCL los objetos de memoria se pueden definir como *buffers* (unidimensionales), imágenes (bidimensionales) o volúmenes (tridimensionales).

Cuando un kernel es enviado desde el *host* para ejecutarse en el dispositivo se define un espacio de índices, de forma que una instancia del kernel (*work-item*) es ejecutada para cada índice de ese espacio. Cada *work-item* ejecuta el mismo código pero la ruta de ejecución y los datos operados pueden variar. Por otro lado, los *work-items* están organizados

en *work-groups*. Todos los *work-items* de un *work-group* dado se ejecutan simultáneamente en los *processing elements* de una única *compute unit* [9].

- **Modelo de memoria:** los *work-items* que ejecutan instancias de un kernel tienen acceso a cuatro regiones de memoria [9]. Estas regiones, representadas en la Figura 3.3, son:

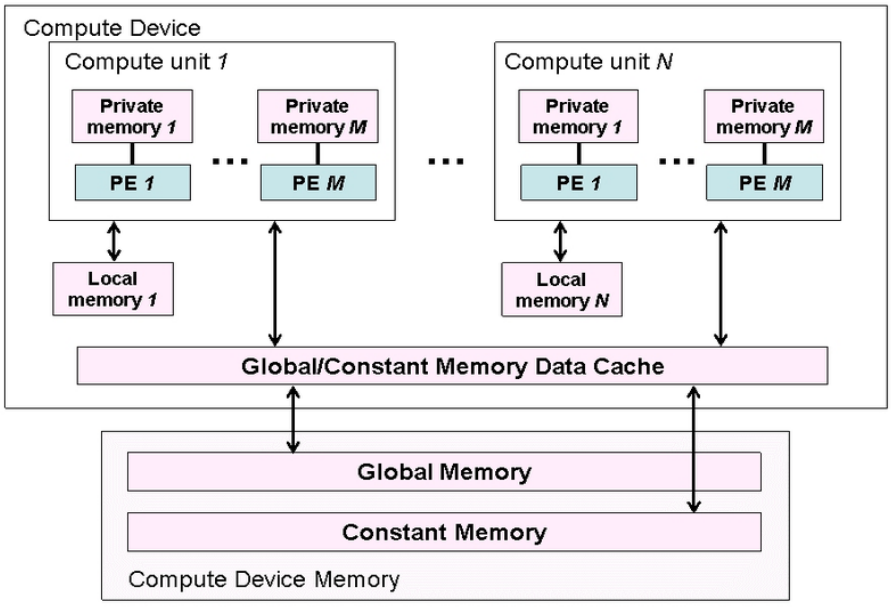

Figura 3.3: Modelo de memoria de OpenCL [9].

- **Memoria global:** unión de todas las memorias RAM de los dispositivos (CPU o GPU) seleccionados para trabajar. Permite el acceso de lectura y escritura a todos los *work-items* de todos los *work-group* [9].

- **Memoria constante:** región de la memoria global declarada como de solo lectura durante la ejecución de un kernel [9]. Es necesario destacar, que algunos dispositivos pueden tener una memoria separada dedicada a este fin.

- **Memoria local:** región de memoria compartida por todos los *work-items* de un único *work-group* [9].

- **Memoria privada:** formada por el conjunto de registros conectados a una Unidad Aritmético Lógica (ALU) concreta. Esta zona de memoria solo es accesible para un único *work-item* [9].

- **Modelo de programación:** OpenCL soporta dos modelos de programación paralela:

  - **Paralelización de datos:** la secuencia de instrucciones es aplicada de forma paralela a los múltiples elementos de un objeto de memoria. El espacio de índices asociado al modelo de ejecución define los *work-items* y como los datos se asignan a estos. En una programación estrictamente paralela de datos se tendría una asociación de uno a uno entre cada *work-item* y cada elemento de un objeto de memoria. No obstante, OpenCL permite una implementación más relajada en la que no es necesaria una paralelización tan estricta [9].

  - **Paralelización de tareas:** existen varias tareas concurrentes, donde cada tarea ejecuta una única instancia de un *kernel* independientemente del espacio de índices. Las tareas concurrentes pueden ejecutar diferentes *kernels*. No obstante, este tipo de paralelización presenta algunas restricciones importantes. Por un lado, no todos los dispositivos la soportan, y por otro lado, aunque el dispositivo la soporte, esta paralelización no se podrá realizar dentro de una única *compute unit* [9].

3.3.2 Estructura de un programa en OpenCL

La estrategia general que se sigue en un programa en OpenCL es la siguiente [26]:

1. Obtener las plataformas existentes y escoger una.
2. Obtener los dispositivos disponibles en la plataforma y elegir uno o varios.
3. Crear un contexto con los dispositivos escogidos.
4. Asociar una cola de comandos a cada dispositivo del contexto.
5. Crear un objeto de programa a partir del fichero con extensión *.cl* que contiene el código fuente del *kernel*. Esto es debido a que en OpenCL los *kernels* se compilan al vuelo.

6. Compilar el programa para los dispositivos del contexto.
7. Cargar los datos de entrada en la memoria del *host*.
8. Copiar los datos a objetos de memoria (*buffers*) del contexto de OpenCL y reservar un objeto de memoria para los datos de salida.
9. Crear uno o varios objetos *kernel* a partir del objeto de programa compilado.
10. Apuntar los parámetros del *kernel* a los objetos de memoria.
11. Lanzar el *kernel*.
12. Esperar a que el *kernel* termine de ejecutarse.
13. Copiar el objeto de memoria de salida a la memoria del *host*.
14. Liberar los recursos previamente reservados.

Capítulo 4

Análisis, Diseño e Implementación de los algoritmos

En todo problema relativo a la ingeniería de *software* es necesario abordar las fases de análisis, diseño e implementación. En este capítulo se abordarán estas fases. Para ello, se realizará un análisis de de las soluciones propuestas para los algoritmos, que se tomarán como punto de partida para nuestra implementación. Así mismo, se presentarán detalles específicos de diseño e implementación.

4.1 Reconstrucción de MRI: algoritmo NESTA

La reconstrucción de la imagen de MRI se puede realizar, siguiendo la teoría de CS, a partir de un espacio K submuestreado. Además, se considera el caso particular de reconstrucción de MRI dinámica (varios *frames*) y multicoil (se utilizan los datos de varias bobinas). Para obtener la imagen reconstruida se resuelve el problema:

$$\underset{\boldsymbol{m}}{\text{minimize}} \quad \frac{1}{2}||\boldsymbol{y} - \boldsymbol{E}\boldsymbol{m}||_{l2}^2 + \lambda||\boldsymbol{\Phi}\boldsymbol{m}||_{l1} \tag{4.1}$$

Este problema de optimización puede ser resulto mediante el algoritmo NESTA [10]. Este algoritmo, basado en el método de Nesterov, minimiza la función iterativamente estimando tres secuencias $\boldsymbol{x_k}$, $\boldsymbol{y_k}$ y $\boldsymbol{z_k}$. La secuencia $\boldsymbol{x_k}$ se corresponde con la señal que queremos estimar, la cual se obtiene a partir de una ponderación de las otras dos. Para obtener las otras dos secuencias ($\boldsymbol{y_k}$ y $\boldsymbol{z_k}$) se resuelven dos problemas de optimización. La convergencia del algoritmo

depende de dos valores escalares α_k y τ_k. Esto se puede observar en el pseudocódigo del método de Nesterov, representado en la Figura 4.1.

Initialize x_0. For $k \geq 0$,

1. Compute $\nabla f(x_k)$.

2. Compute y_k:

$$y_k = \text{argmin}_{x \in Q_p} \frac{L}{2}\|x - x_k\|_{\ell_2}^2 + \langle \nabla f(x_k), x - x_k \rangle.$$

3. Compute z_k:

$$z_k = \text{argmin}_{x \in Q_p} \frac{L}{\sigma_p} p_p(x) + \sum_{i=0}^{k} \alpha_i \langle \nabla f(x_i), x - x_i \rangle.$$

4. Update x_k:

$$x_k = \tau_k z_k + (1 - \tau_k) y_k.$$

Stop when a given criterion is valid.

Figura 4.1: Pseudocódigo del método de Nesterov [10].

El algoritmo NESTA, disponible en MATLAB, se muestra resumido en el diagrama de la Figura 4.2, en el cual se incluyen algunos detalles relacionados con la programación que se va a emplear para el mismo. Para realizar la optimización, NESTA hace uso de la aplicación de los operadores y sus adjuntos, ya que la obtención del gradiente se puede realizar fácilmente a partir de los adjuntos de los operadores. En concreto, necesitamos el operador de codificación $\boldsymbol{E}$ y su adjunto $\boldsymbol{E^H}$, y el operador *sparse* $\boldsymbol{\Phi}$ y su adjunto $\boldsymbol{\Phi^H}$.

4.1.1 Operador de codificación

El operador de codificación $\boldsymbol{E}$ permite pasar de los datos en el espacio imagen ($\boldsymbol{m}$) a los datos submuestreados en el espacio K ($\boldsymbol{y}$) [27]. La formulación matricial del problema es:

$$\boldsymbol{y} = \boldsymbol{E}\boldsymbol{m} = \begin{bmatrix} \boldsymbol{KFS_1} \\ ... \\ \boldsymbol{KFS_{Nc}} \end{bmatrix} \boldsymbol{m} \tag{4.2}$$

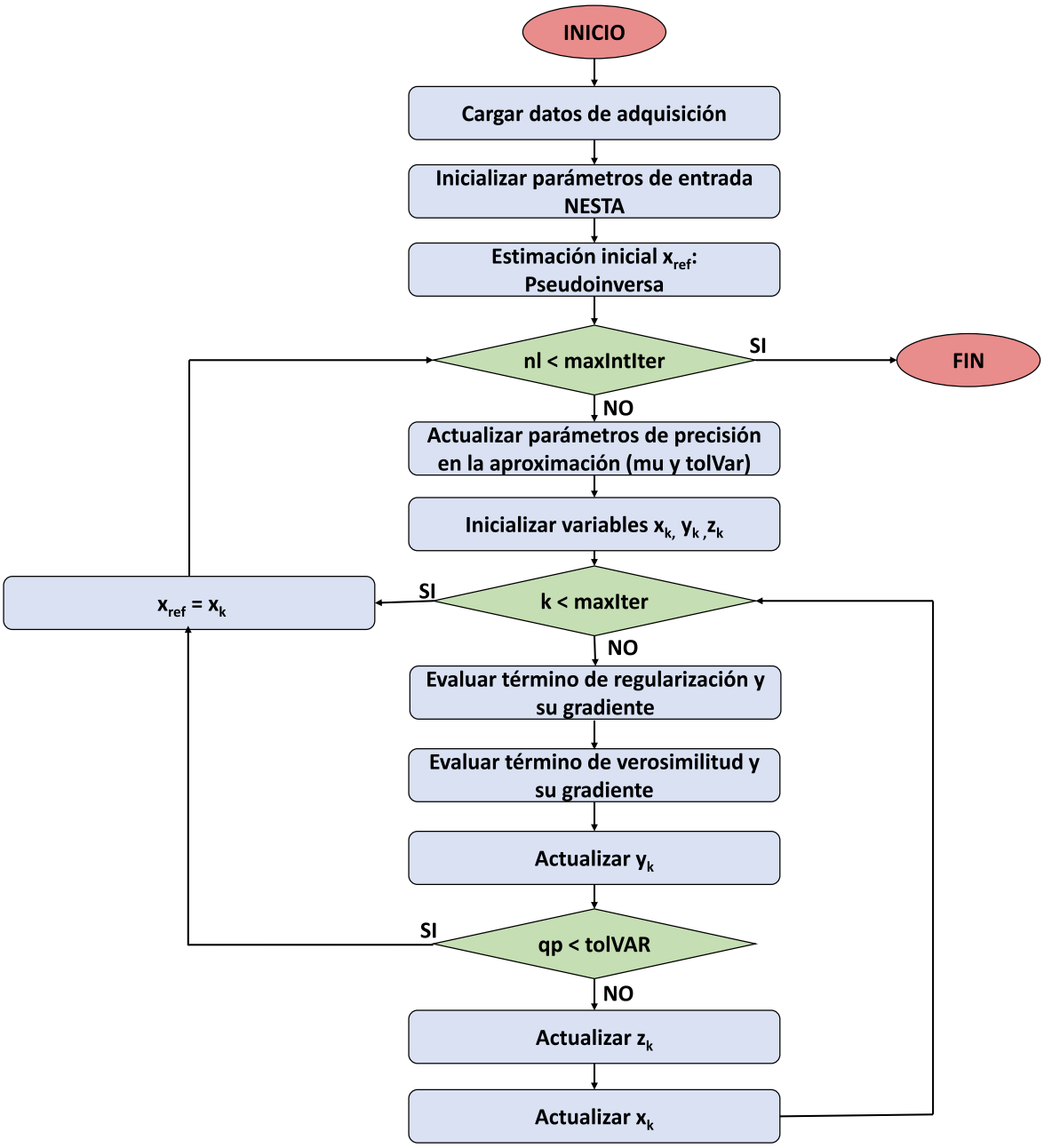

Figura 4.2: Diagrama de flujo del algoritmo NESTA.

Es decir, la aplicación del operador, representado en la Figura 4.3, se puede resumir en tres pasos:

1. Multiplicación punto a punto por los mapas de sensibilidad de cada bobina $\boldsymbol{S_c}$.
2. Transformada de Fourier espacial $\boldsymbol{F}$.
3. Aplicación de una máscara o trayectoria de submuestreo (idéntica para todos las bobinas) que mantiene solo las posiciones del espacio K requeridas $\boldsymbol{K}$.

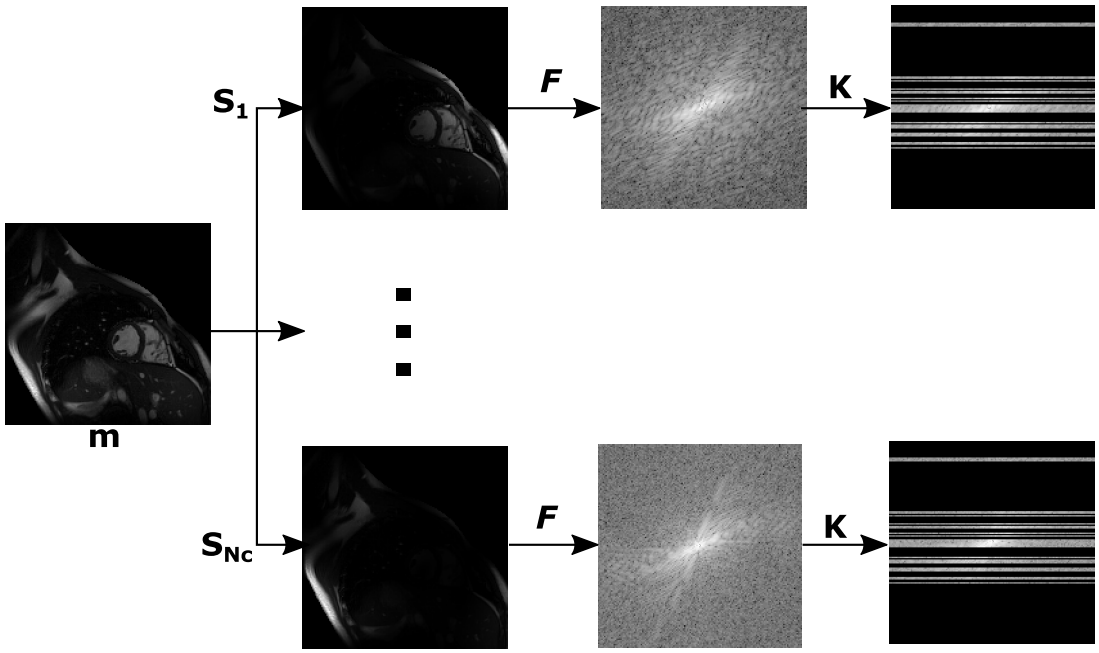

Figura 4.3: Aplicación del operador de codificación. Dónde $\boldsymbol{m}$ es la imagen reconstruida, $\boldsymbol{S_c}$ los mapas de sensibilidad, $\boldsymbol{F}$ la transformada de Fourier y $\boldsymbol{K}$ el patrón de submuestreo.

Por el contrario, el operador de codificación adjunto $\boldsymbol{E^H}$ realiza la operación opuesta, es decir, pasar los datos submuestreados en el espacio K a los datos en el espacio imagen [27] En notación matricial tenemos que:

$$\boldsymbol{m} = \boldsymbol{E^H y} = \begin{bmatrix} \boldsymbol{S_1^H F^H} \\ ... \\ \boldsymbol{S_{N_c}^H F^H} \end{bmatrix} \boldsymbol{y} \tag{4.3}$$

De forma equivalente a la notación matricial, en este caso la aplicación del operador adjunto se puede realizar en tres pasos:

1. Transformada inversa de Fourier $\boldsymbol{F^H}$.

2. Multiplicación punto a punto por el transpuesto conjugado de los mapas de sensibilidad de cada bobina $\boldsymbol{S^H}$.

3. Sumar las imágenes resultantes de todos las bobinas.

4.1.2 Operador sparse

El operador *sparse* $\boldsymbol{\Phi}$ se fija para conseguir realizar una transformación sobre la imagen que permita representarla con un bajo número de coeficientes. Dado que vamos a realizar la reconstrucción de adquisiciones dinámicas de MRI, una buena solución es utilizar como operador *sparse* diferencias temporales de la imagen en *frames* consecutivos. Esta operación es conocida como variación total temporal (tTV) [27]. Si consideramos una imagen dinámica con cinco *frames* temporales $\boldsymbol{m_i}$, la aplicación del operador tTV de forma matricial sería:

$$\boldsymbol{z = \Phi m} = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{m_1} \\ \boldsymbol{m_2} \\ \boldsymbol{m_3} \\ \boldsymbol{m_4} \\ \boldsymbol{m_5} \end{bmatrix} = \begin{bmatrix} \boldsymbol{m_2 - m_1} \\ \boldsymbol{m_3 - m_2} \\ \boldsymbol{m_4 - m_3} \\ \boldsymbol{m_5 - m_4} \end{bmatrix} \tag{4.4}$$

En este caso, el adjunto de este operador coincidirá con la transpuesta de la matriz (dado que es real). Gráficamente, se puede observar en la Figura 4.4 que la aplicación del operador tTV sobre la imagen dinámica da lugar a una transformación *sparse*, ya que el resultado tiene pocos coeficientes distintos de cero. Concretamente, entre dos *frames* consecutivos solo tendremos valores no nulos dónde se ha producido movimiento.

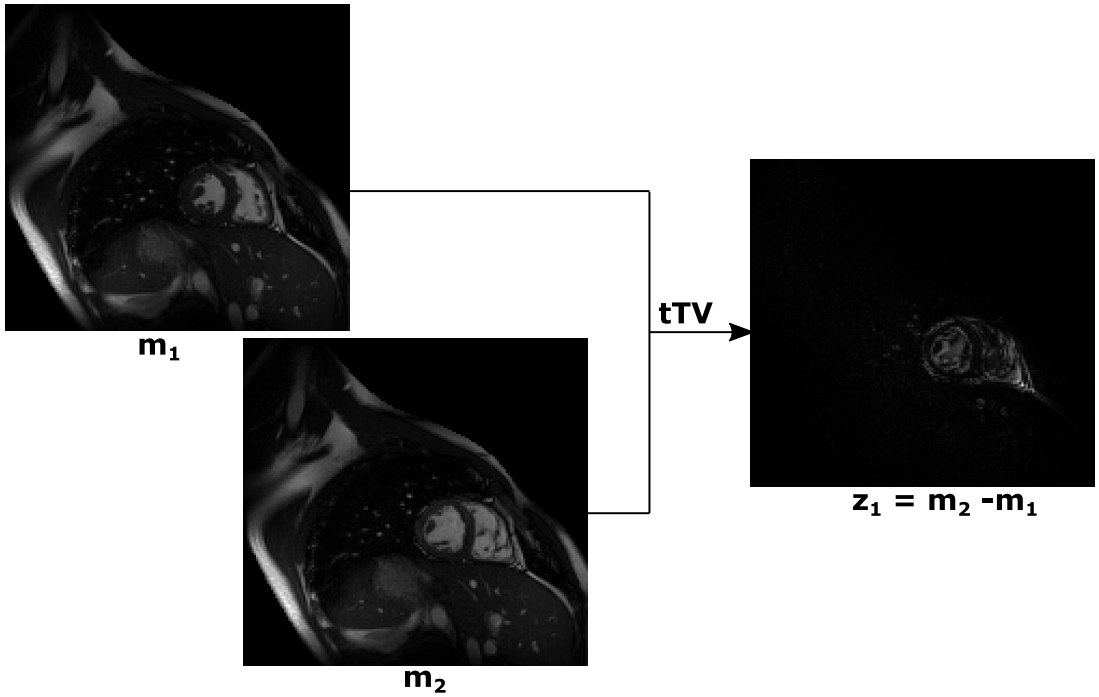

Figura 4.4: Aplicación del operador *sparse* ***tTV***. Dónde ***m*1** es el primer *frame* de la imagen reconstruida, ***m*2** es el segundo *frame* de la imagen reconstruida, y ***z*1** es el resultado de la aplicación del operador.

4.1.3 Diagrama de clases

En esta sección se va a presentar, utilizando notación UML (Unified Modeling Language), el diagrama de clases diseñado y que posteriormente servirá como punto de partida para la implementación del algoritmo. Para diseñar el diagrama de clases es necesario tener en cuenta que la implementación se integrará en el *framework* OpenCLIPER, el cual ya ofrece una estructura de clases con numerosas funcionalidades para facilitar las tareas al programador, por lo que el diagrama de clases diseñado se adapta a la estructura de clases de OpenCLIPER (más detalles sobre este *framework* se pueden encontrar en el apéndice A).

En la Figura 4.5 se ilustra el diagrama de clases simplificado, el cual incluye tanto las clases como las relaciones entre ellas. Por simplicidad, muestra solo clases, sin atributos ni métodos. Como se puede observar, el diseño realizado esta compuesto por siete clases:

- **NestaUp**: clase principal que implementa el algoritmo. Tiene como entradas el espacio K, los mapas de sensibilidad de las bobinas y la máscara de submuestreo. La salida es la imagen reconstruida.

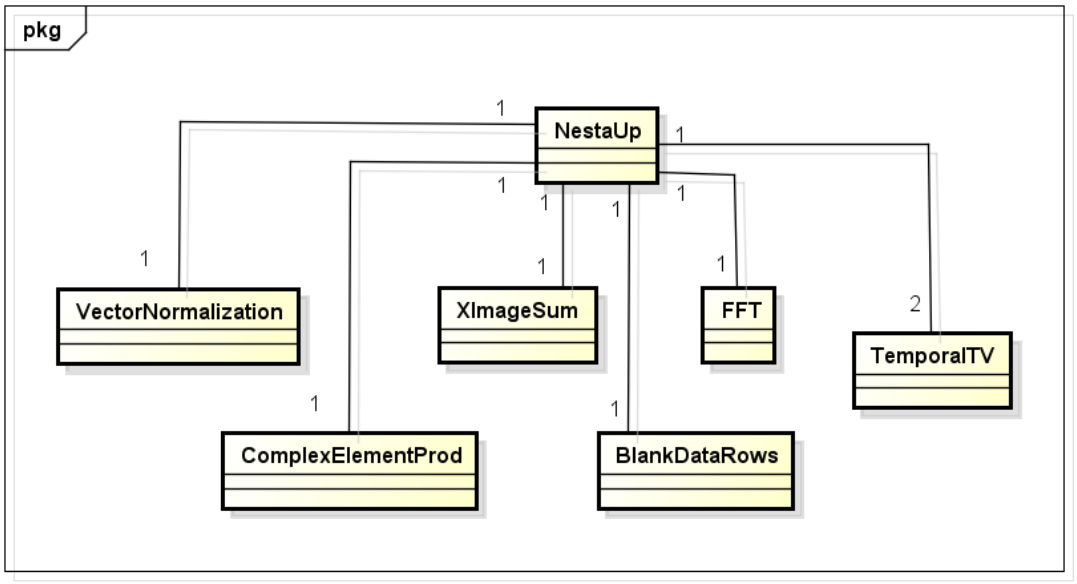

Figura 4.5: Diagrama de clases UML del algoritmo NESTA diseñado.

- **FFT**: realización de la FFT o iFFT.

- **ComplexElementProd**: multiplicación compleja punto a punto de dos *arrays* de datos. Debe permitir también realizar la multiplicación con el transpuesto conjugado de uno de los *arrays*. Las entradas son las imágenes de cada bobina y los mapas de sensibilidad, mientras que la salida es la multiplicación de ambas.

- **XImageSum**: adición de las imágenes capturadas por un grupo de bobinas al mismo tiempo (mismo *frame*). La entrada es la imagen de cada bobina (ya multiplicada por los mapas de sensibilidad) y la salida es la suma de todas ellas.

- **BlankDataRows**: aplicación de la máscara de submuestreo, la cual consistirá en inicializar a 0 las líneas submuestreadas. Las entradas son el espacio K no submuestreado y la máscara de submuestreo, mientras que la salida es el espacio K submuestreado.

- **TemporalTV**: aplicación del operador variación total temporal o su adjunto.

- **VectorNormalization**: normalización de un *array* de datos complejos. Debe normalizar cada valor del *array* por el máximo entre una constante dada o el módulo de ese elemento del *array*.

4.1.4 Implementación sobre OpenCLIPER

El objetivo perseguido con el desarrollo del algoritmo es disponer de una implementación para cualquier tipo de dispositivo (independientemente del fabricante del mismo) y, en especial, los dispositivos de tipo GPU. Dicha implementación se integrará en el *framework* OpenCLIPER, el cual facilita las tareas de programación asociadas a OpenCL. Además, dado que este *framework* está destinado al procesamiento y la reconstrucción de imágenes médicas, el algoritmo aquí implementado se podría combinar con otros implementados en el mismo *framework*, como la compensación de movimiento en imágenes dinámicas de MRI, para dar lugar a *pipelines* más complejos de reconstrucción.

Según la estructura de clases de OpenCLIPER (ver apéndice A) las clases que hemos diseñado se implementarán como clases derivadas de la clase `Process`. Además, los *process* `FFT`, `ComplexElementProd`, `XImageSum` y `BlankDataRows` son funcionalidades ya incorporadas en el *framework* por lo que directamente se hace uso de ellos en la implementación del algoritmo. En concreto, se crea una nueva clase principal `NestaUp` que se encarga de realizar las operaciones necesarias en el algoritmo. La clase `NestaUp`, como todos los *process* en OpenCLIPER, consta de cuatro métodos públicos:

- `NestaUp()`: constructor de la clase.
- `init()`: para realizar las tareas de inicialización de los *process* necesarios en el algoritmo.
- `launch()`: para realizar las tareas de computación, propiamente dicho, del algoritmo. La mayor parte de estas tareas consisten en operaciones con matrices y/o vectores, las cuales se realizan con apoyo de la librería CLBlas [28] para cálculo matricial en OpenCL. Las funciones utilizadas de esta librería son:
  - **clblasCcopy()**: copia los elementos de un vector en precisión simple a otro.
  - **clblasScnrm2()**: computa la norma euclídea de un vector que contiene elementos complejos en precisión simple.
  - **clblasCdotc()**: producto escalar de dos vectores que contienen elementos complejos en precisión simple, conjugando el primer vector.
  - **clblasCaxpy()**: escalado de un vector de elementos complejos en precisión simple por una constante compleja y suma de los elementos de un segundo vector.

  - **clblasCsscal()**: escalado de un vector de elementos complejos en precisión simple por una constante real también en precisión simple.
  - **clblasScasum()**: suma de valores absolutos de un vector que contiene elementos complejos en precisión simple.

- `~NestaUp()`: destructor de la clase.

Como se ha indicado en la descripción del algoritmo NESTA, es necesario definir los operadores que caracterizan el problema de reconstrucción de MRI y sus adjuntos. Estos operadores se implementan mediante métodos privados de la clase `NestaUp`. Tenemos, por lo tanto cuatro métodos privados para estos operadores:

- `operatorA()`: operador de codificación.
- `operatorAt()`: operador de codificación adjunto.
- `operatorU()`: operador *sparse*, es decir, tTV.
- `operatorUt()`: operador *sparse* adjunto.

Desde estos métodos se realiza el lanzamiento de otros *process*, en los cuales a su vez se ejecutan *kernels*, escritos en lenguaje OpenCL, en la GPU. De estos *process* se crean instancias reutilizadas en los distintos métodos y en cada una de la llamada a esos métodos. Gráficamente, en la Figura 4.6 se muestra la relación entre las clases `Process` utilizadas y las instancias creadas de las mismas, así como los *process* llamados en cada método.

Adicionalmente, en la clase `NestaUp` se crea un quinto método privado `myNormest()`. Este método estima la norma espectral de un operador utilizando el método de la potencia y será utilizado para estimar la norma del operador *sparse* necesario en el algoritmo NESTA.

Por último, conviene destacar que los datos de entrada al algoritmo son el espacio K cartesiano submuestreadp creado como un objeto de la clase `KData`, los mapas se sensibilidad de las bobinas cargados como un objeto de la clase `SensitivityMapsData` y la máscara de submuestreo como un objeto de la clase `SamplingMasksData`. La salida es la imagen reconstruida para cada *frame* almacenado en un objeto de la clase `XData`.

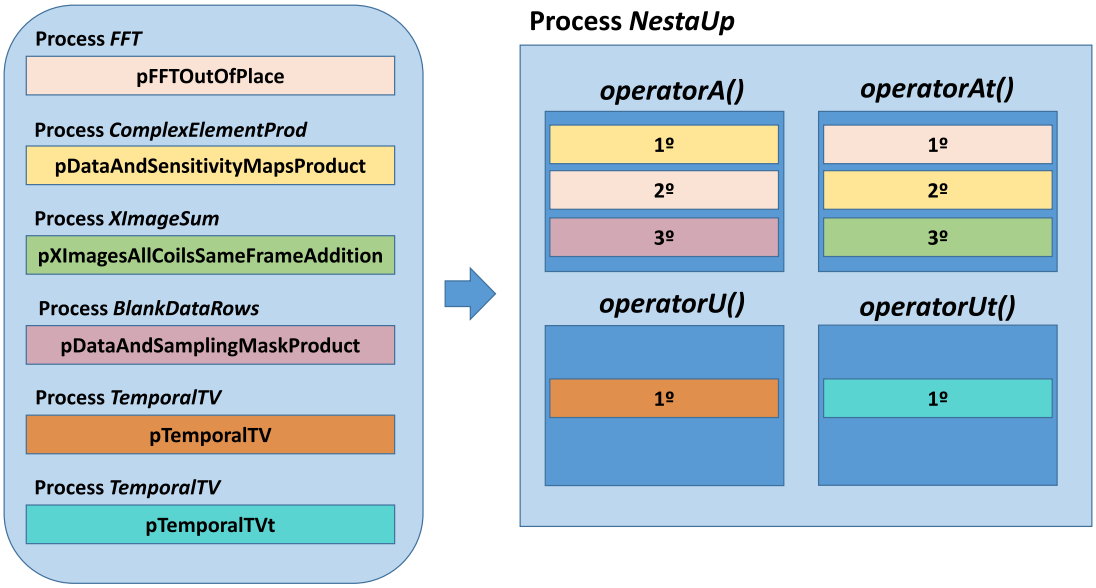

Figura 4.6: Instancias de *process* creadas en `NestaUp` y orden en el que son lanzados dichos *process* en los métodos.

4.2 Algoritmo NUFFT

Este algoritmo realiza el remuestreo de datos (situados en frecuencias espaciales arbitrarias) en una cuadrícula cartesiana, con el fin de soportar el uso de la iFFT. Esto es útil en el caso de adquisiciones de datos no uniformes en el espacio K, por ejemplo, mediante el uso de trayectorias radiales o espirales. Basándonos en la librería gpuNUFFT [12], programada en C++ y CUDA, se realiza un implementación de este algoritmo. De forma específica, la implementación realizada permite pasar datos en 3D de espacios K no cartesianos al espacio imagen. Como se explicó en el capitulo 2, los pasos necesarios para la aplicación de la NUFFT son:

1. **Compensación de la densidad.**

2. **Gridding.**

3. **iFFT.**

4. **Recorte de la imagen.**

5. **Deapodización.**

En el algoritmo se considera que los puntos del espacio K no cartesiano se muestrean en el rango [-0.5, 0.5] en cada dimensión (k_x, k_y y k_z), y que la imagen resultante tiene resolución unitaria en el rango [-N/2, N/2-1]. Donde N es el ancho de la imagen en píxeles.

El enfoque que se realiza en el algoritmo es dividir la rejilla cartesiana del espacio K en regiones más pequeñas denominadas sectores, cuyo ancho es determinado por el parámetro *sector_width*. El objetivo perseguido es poder trabajar con estos sectores en paralelo. A modo de ejemplo, en la Figura 4.7 se muestra una división en cuatro sectores de una rejilla cartesiana 2D en el rango [0, 0.5]. Este enfoque tiene el problema de cómo considerar correctamente los puntos del espacio K no cartesiano que se encuentran en el borde del sector. Para ello, cada sector se expande en la mitad del ancho del *kernel* (W) en cada dirección del espacio K. Este nuevo ancho de sector se denomina *sector_pad_width* [12]. Las zonas de solapamiento entre sectores se encuentran representadas en la Figura 4.7 en color gris oscuro.

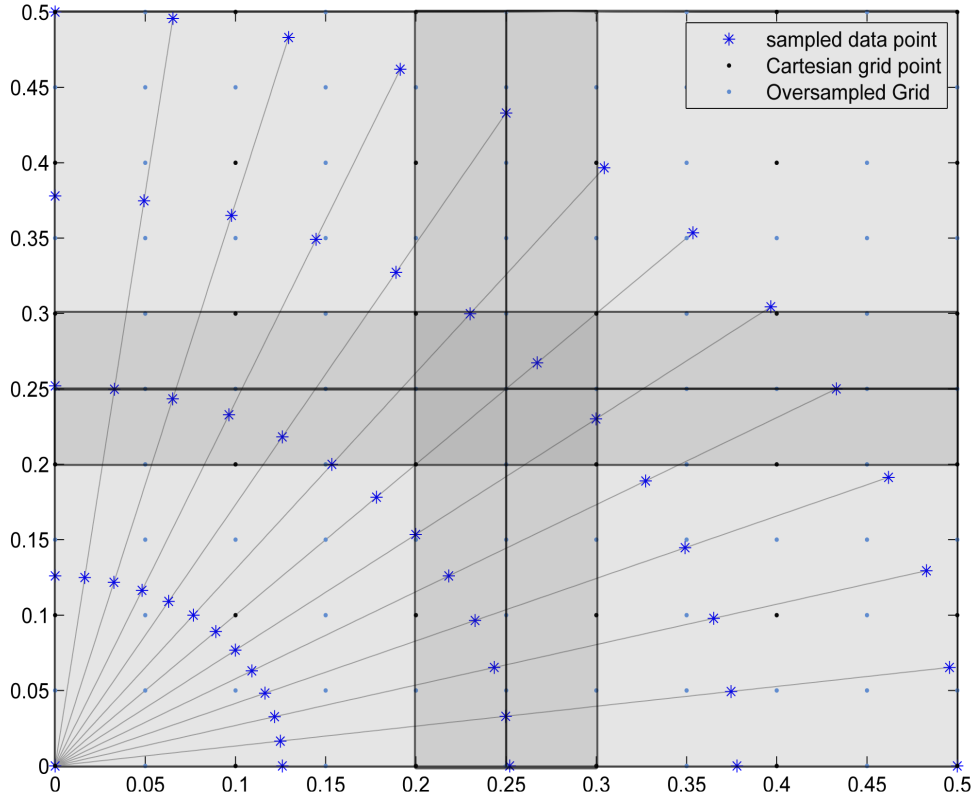

Figura 4.7: División de la rejilla cartesiana en cuatro sectores, cada uno con un ancho de sector de 6 unidades de la rejilla. Las regiones resaltadas en gris oscuro, de 3 unidades de rejilla (ancho del *kernel*), indican los puntos de la rejilla que están influenciados por sectores vecinos debido a la superposición [12].

Para aplicar el algoritmo de esta manera, cada punto de datos no cartesiano debe asignarse a su sector correspondiente. Es necesario destacar que este mapeo se debe realizar una única vez por trayectoria. Es decir, una vez la trayectoria ha sido mapeada se podría utilizar este paso preliminar para la aplicación de la NUFFT a cualquier espacio K adquirido con la misma trayectoria. Así mismo, el *kernel* de convolución puede ser precomputado y almacenado en una tabla de consulta. El producto de la densidad del kernel K_d y el ancho de mismo W determina el número de entradas necesarias en la tabla. Lo mismo ocurre con la función de deapodización.

La realización del *gridding* implica que se evalúe a que puntos de la rejilla cartesiana afecta cada muestra del espacio K no uniforme. Para ello, se calcula la distancia cuadrática en las direcciones x, y y z desde cada punto no cartesiano a la rejilla cartesiana. Posteriormente, se selecciona la entrada correspondiente en la tabla de consulta del *kernel* de convolución. El valor del espacio K no cartesiano se pondera con el valor del *kernel*, y el resultado se acumula en el punto correspondiente de la rejilla. En el caso de que el punto de datos caiga fuera del borde de la rejilla, entonces este valor se acumula en el lado opuesto de la misma.

De forma general, el algoritmo explicado a lo largo de esta sección se muestra resumido en el diagrama de flujo de la Figura 4.8, en el cual se incluyen algunos detalles relacionados con la programación que se va a emplear para el mismo.

4.2.1 Kernel de convolución

Uno de los aspectos clave para reducir los tiempos de computo del algoritmo NUFFT es la elección adecuada del *kernel* de convolución y la utilización del mínimo factor de sobremuestreo necesario [12]. Esto es debido a que estos dos parámetros influyen notablemente en el paso de la convolución, que es el más costoso en todo el algoritmo. No obstante, la elección de estos parámetros también influirá en la precisión y calidad de la imagen final, por lo que deben satisfacer determinados requisitos para cumplir con dichos objetivos.

Siguiendo la implementación de la librería gpuNUFFT [12], la función de interpolación propuesta es la función de Kaiser-Bessel descrita en la ecuación 4.5.

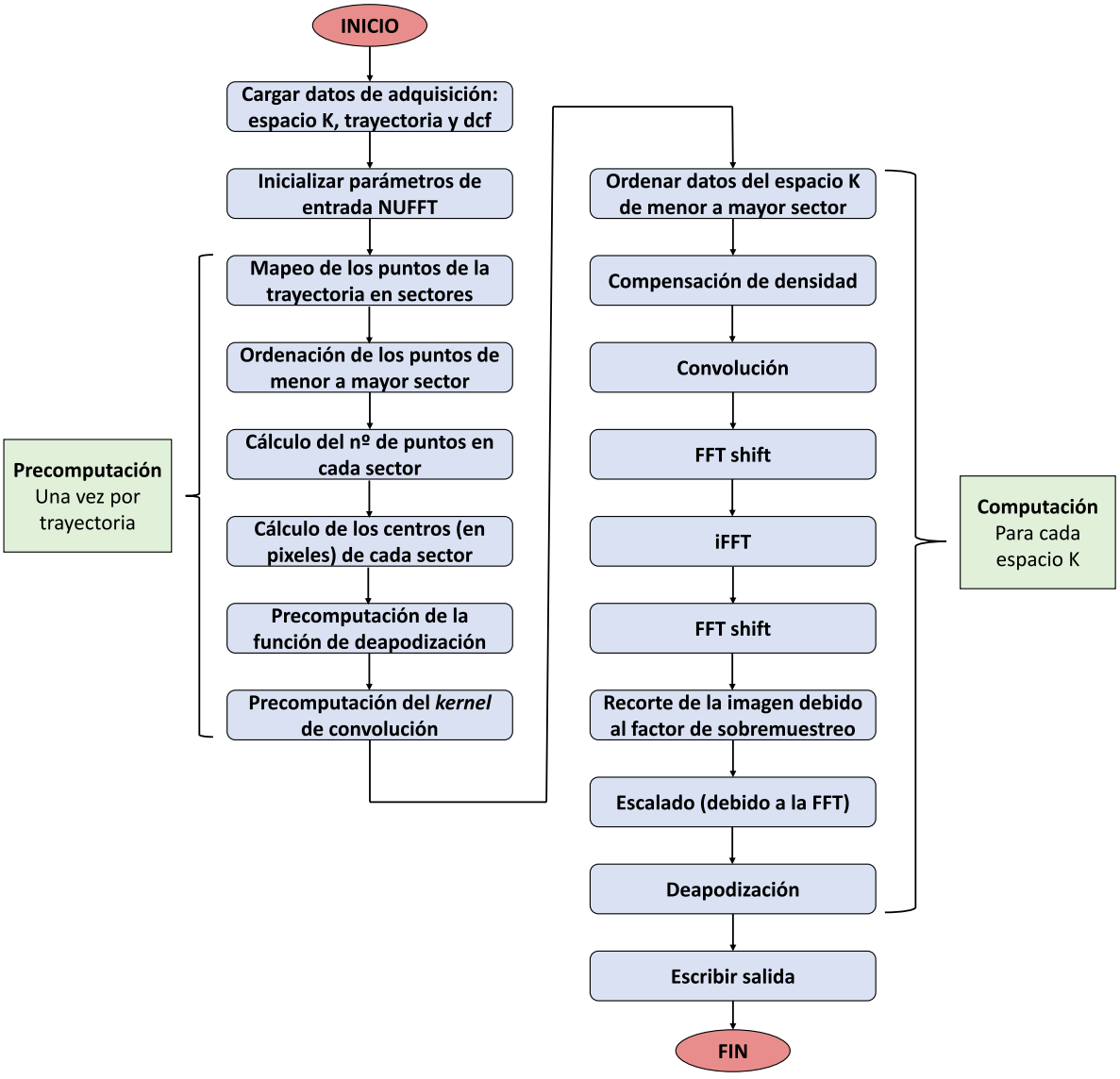

Figura 4.8: Diagrama de flujo del algoritmo NUFFT.

$$C(k_x) = \frac{C}{W} I_0 \left(\beta \sqrt{1 - (2Gk_x/W)^2} \right) \tag{4.5}$$

Dónde, I_0 es la función de Bessel modificada de primera especie y orden cero. Mientras que β es el parámetro que define la forma de la función y, como se muestra en la ecuación 4.6, depende del ancho del kernel W y del ratio de sobremuestreo α.

$$\beta = \pi \sqrt{\frac{W^2}{\alpha^2} \left(\alpha - \frac{1}{2} \right)^2 - 0{,}8} \tag{4.6}$$

4.2.2 Diagrama de clases

Igual que en el caso anterior, la implementación del algoritmo se integrará en la estructura de clases del *framework* OpenCLIPER (ver apéndice A). En la Figura 4.9 se muestra el diagrama de clases diseñado para este algoritmo en notación UML. Por simplicidad, muestra solo clases, sin atributos ni métodos. Como se puede observar, el diseño realizado consta de nueve clases:

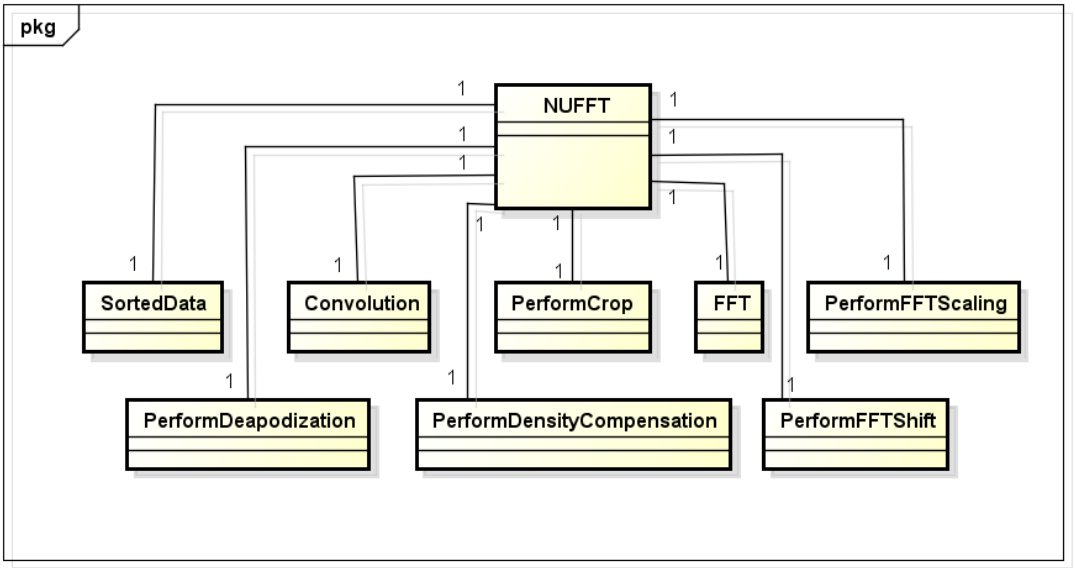

Figura 4.9: Diagrama de clases UML del algoritmo NUFFT diseñado.

- **NUFFT**: clase principal que implementa el algoritmo. Tiene como entradas el espacio K no cartesiano, la trayectoria de adquisición de los datos y la función de compensación de densidad. La salida es la imagen reconstruida.
- **SortedData**: ordenación de los datos del espacio K no cartesiano de acuerdo con el sector al que pertenecen. La entrada es el espacio K no cartesiano, y la salida ofrecida son los mismos datos ordenados de menor a mayor sector.
- **PerformDensityCompensation**: escalado de los datos no cartesianos por la función de compensación de densidad.
- **Convolution**: interpola los datos del espacio K no cartesianos dentro de la rejilla cartesiana, mediante la convolución con un kernel.
- **PerformFFTShift**: desplazamiento de la componente de continua (frecuencia cero) al centro de la imagen.
- **FFT**: realiza la iFFT de los datos en al rejilla cartesiana. Devuelve los datos en el espacio imagen.
- **PerformCrop**: recorte de los datos en el espacio imagen debido al factor de sobremuestreo de la rejilla cartesiana.
- **PerformFFTScaling**: escalado de los datos en el espacio imagen debido a la aplicación de la iFFT.
- **PerformDeapodization**: corrección de los datos en el espacio imagen debido a la convolución del espacio K no cartesiano con el kernel de convolución. Se realiza mediante un escalado de los datos con la función de deapodización.

4.2.3 Implementación sobre OpenCLIPER

El operador NUFFT y su adjunto son comúnmente utilizados en algoritmos iterativos de reconstrucción, por lo que la implementación realizada se integra en el *framework* OpenCLIPER con el objetivo de que en el futuro este desarrollo se pueda utilizar dentro de alguno de los algoritmos de reconstrucción implementados en él.

Según la estructura de clases de OpenCLIPER (ver Apéndice A) las clases que se han diseñado se implementarán como clases derivadas de la clase `Process`. En particular, se crea una nueva clase principal `NUFFT` que se encarga de gobernar el orden de todas las operaciones necesarias en el algoritmo. La clase `NUFFT`, como todos los *process* en OpenCLIPER, consta de cuatro métodos públicos:

- `NUFFT()`: constructor de la clase.
- `init()`: es un método destinado a las tareas de inicialización de los *process* y objetos de memoria necesarios en el algoritmo. Además, para este algoritmo también se realizan las tareas de precomputación, es decir, las tareas que son necesarias realizar una única vez por trayectoria. En la Figura 4.8 se puede observar en qué consisten estas tareas.
- `launch()`: método para realizar las tareas de computación, propiamente dicho, del algoritmo. Es decir, en este método se realizan las tareas que son necesarias realizar para cada espacio K no cartesiano.
- `~NUFFT()`: destructor de la clase.

El orden que se sigue en la ejecución del algoritmo sería crear un objeto de tipo `NUFFT`, llamar al método `init()` una única vez para realizar las tareas de precomputación, llamar al método `launch()` para reconstruir el espacio K capturado por cada bobina y, por último, liberar los recursos. Es necesario destacar que las tareas de precomputación se ejecutan siempre en CPU, mientras que las tareas de computación se ejecutan en GPU.

Adicionalmente, en la clase `NUFFT`, se crean los métodos privadas `kernel()` y `i0()`, destinados a precomputar el *kernel* de convolución. Y los métodos `precomputeDeapodization()`, `calculateDeapodization()` y `calculateDeapodizationValue()` destinados a precomputar la función de deapodización. Todos estos métodos privados, dado que son de precomputación, son utilizados en el método `init()`.

Por último, conviene destacar que los datos de entrada al algoritmo son el espacio K no cartesiano creado como un objeto de la clase `KData` y la trayectoria de adquisición cargada como un objeto de la clase `Trajectories`; la salida es el volumen 3D reconstruido almacenado en un objeto de la clase `XData`.

Capítulo 5

Resultados

En este capítulo se analizan los resultados obtenidos, tanto en cuanto a validación de las imágenes de resonancia magnética resultantes de los dos algoritmos bajo estudio (NESTA y NUFFT), como en cuanto a evaluación de las medidas de rendimiento de los algoritmos implementados en comparación con otras implementaciones de los mismos.

5.1 Resultados del algoritmo NESTA

En esta sección se muestran los resultados obtenidos de la ejecución del algoritmo NESTA implementado. La ejecución del algoritmo se realiza con un espacio K cartesiano 2D (160 x 160), 20 *coils* y 16 *frames*. Este espacio K cuenta con un factor de aceleración de 6 en la dirección de codificación de fase. Así mismo, se dispone de los mapas de sensibilidad de los 20 *coils* y la máscara de submuestreo de la adquisición.

Los valores de los parámetros utilizados de entrada al NESTA se muestran en la Tabla 5.1. Es necesario destacar que el valor de estos parámetros influirá en la correcta convergencia del algoritmo y, por consiguiente, en el tiempo de ejecución del mismo. Una breve explicación del significado de estos parámetros se indica también en la Tabla 5.1.

Con estos datos de entrada y fijados los parámetros según se indica en la Tabla 5.1 se obtienen las imágenes mostradas en la Figura 5.1. Las imágenes de la izquierda ($\boldsymbol{m_0}$) se corresponden con dos *frames* distintos (en sistole y diastole) de la estimación inicial realizada mediante la pseudoinversa ($\boldsymbol{E^H y}$), la cual es utilizada como semilla en el algoritmo. Las imágenes de la

Parámetro	Valor	Explicación
lambda	3E-3	Peso de regularización
muf	1E-4	Precisión en la aproximación
La	1	Constante de Lipschitz del término cuadrático
tolVar	1E-4	Tolerancia para el criterio de parada
maxIter	3000	Máximo número de iteraciones del bucle interior
maxIntIter	6	Número de pasos de aproximación (bucle exterior)

Tabla 5.1: Valores de los parámetros de entrada al NESTA utilizados.

derecha ($\boldsymbol{m_{NESTA}}$) son los mismos *frames*, pero de las imágenes finales reconstruidas mediante el algoritmo. Como se puede observar este algoritmo de optimización permite realizar una muy buena reconstrucción de las imágenes, las cuales se obtienen prácticamente libres de artefactos. Adicionalmente, para demostrar la corrección de la implementación, se calcula el índice de similitud estructural (SSIM) entre la imagen obtenida mediante nuestra implementación y la obtenida con MATLAB. Este parámetro se basa en el cálculo de tres términos, a saber, el término de luminancia, el término de contraste y el término estructural. De forma que el índice es una combinación multiplicativa de los tres términos [29]. El índice es calculado *frame* a *frame* para cada imagen 2D resultante, obteniéndose un valor medio entre todos los *frames* de 0.9688 y una varianza de 3.1020E-06. Dado que el máximo valor de este parámetros es 1, consideramos que es un valor razonable teniendo en cuenta posibles errores de precisión numérica.

Por otro lado, para evaluar la correcta convergencia del algoritmo con los parámetros fijados en la Tabla 5.1, en la Figura 5.2 se muestra la curva de convergencia del algoritmo en escala logarítmica. En esta imagen se puede ver la reducción de la función de coste a medida que aumenta el número de iteraciones, hasta llegar a un punto en el que el valor de esta función se estabiliza. Como se puede observar, se consigue reducir la función de coste en un orden de magnitud.

Hasta ahora, el análisis se ha realizado con unos parámetros de entrada al NESTA que funcionan razonablemente bien, aunque muchos de estos parámetros se deben ajustar empíricamente para cada problema concreto. Uno de los parámetros de mayor importancia es el peso de la regularización λ. Para ilustrar la importancia de este parámetro en la correcta convergencia del algoritmo, así como en la imagen obtenida, en la Figura 5.3 se muestran dos *frames* distintos (en sistole y diastole) de las imágenes finales reconstruidas mediante el algoritmo para tres valores de

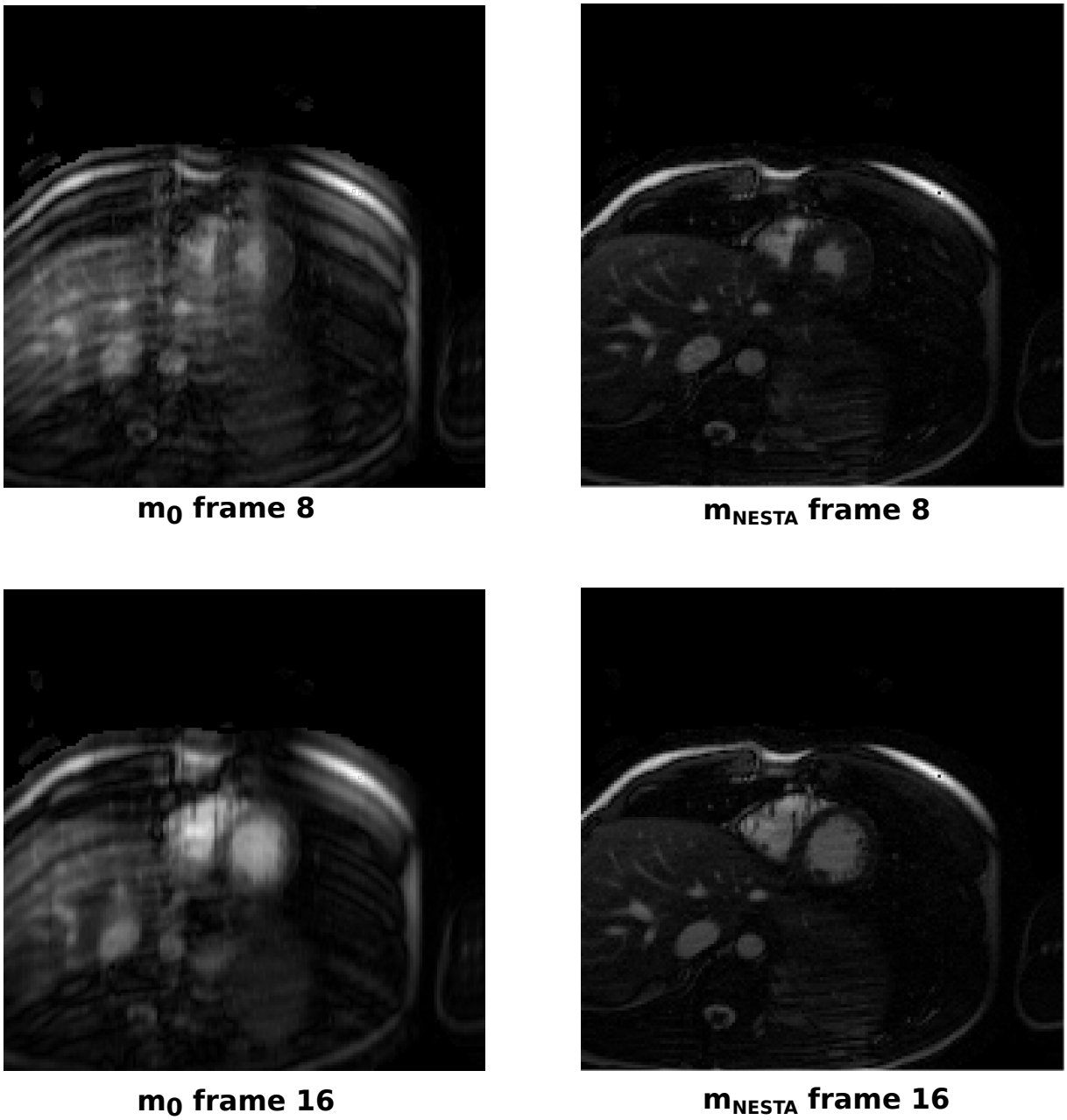

Figura 5.1: a la izquierda, dos *frames* (*frame* 8 y *frame* 16) de la estimación inicial mediante la pseudo-inversa, utilizada como semilla en el algoritmo. A la derecha, mismos *frames* en la imagen reconstruida obtenida con el algoritmo NESTA.

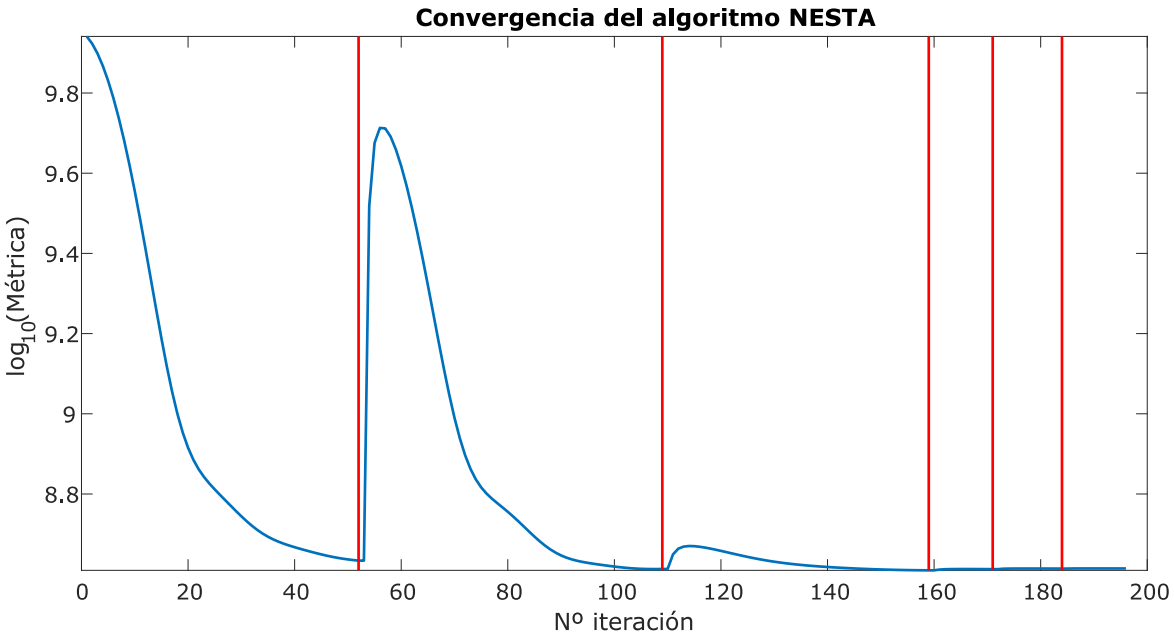

Figura 5.2: Evolución de la métrica en escala logarítmica con el número de iteraciones del algoritmo. Las líneas verticales rojas separan las optimizaciones con un menor valor del parámetro de precisión en la aproximación (mu).

λ. Se puede observar que en el caso de no incorporar regularización ($\lambda = 0$), la reconstrucción de la imagen no se realiza correctamente. En caso contrario, con un término de regularización muy grande ($\lambda = 3$), la imagen se encuentra sobrerregularizada, por lo que no se aprecia variación temporal en la secuencia reconstruida. Esto indica que es necesario realizar un ajuste fino del parámetro para conseguir un compromiso entre la fidelidad a los datos y la regularización de la solución. Experimentalmente se ajusta este parámetro al valor $\lambda = 3E - 3$ para obtener una buena solución.

5.1.1 Medidas de rendimiento

El rendimiento del algoritmo implementado se evalúa en términos de tiempo de ejecución por iteración. Se toma esta decisión ya que, en principio, el número de iteraciones necesarias para la convergencia el algoritmo depende de los datos, de los parámetros fijados de entrada al NESTA y de la inicialización realizada. De esta forma, el tiempo medio por iteración representa una medida mucho más flexible del rendimiento. Los tiempos de ejecución obtenidos con la implementación planteada ejecutada en GPU se comparan con los obtenidos mediante el código de MATLAB del que se partió. El experimento se realiza en una máquina con procesador 8xIntel®

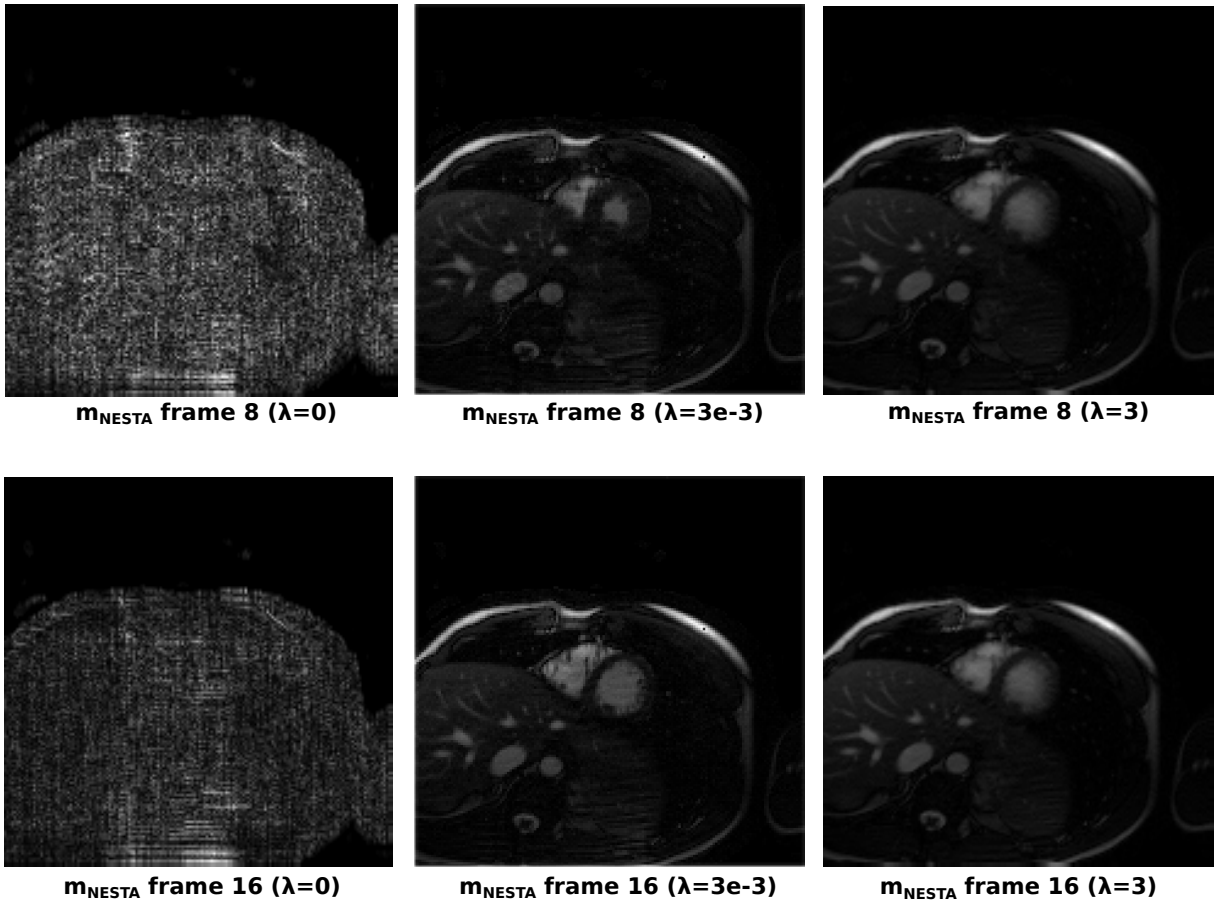

Figura 5.3: Análisis del término de regularización. Dos *frames* (*frame* 8 y *frame* 16) de la imagen reconstruida obtenida con el algoritmo NESTA para disintos valores del parámetro λ.

Core™ i7-4790 3.60 GHz y 16 GB de memoria RAM. Así mismo, en la máquina estaba disponible una GPU AMD *Hawaii* de 4 GB de memoria RAM. Para medir los tiempos de forma fiable, se realizan 100 ejecuciones del algoritmo.

En la Figura 5.4 se muestran los valores obtenidos de tiempo medio por iteración. Como se puede observar, además de cumplir con el objetivo de disponer de una implementación del algoritmo independiente del dispositivo, la implementación desarrollada en OpenCLIPER permite reducir en un orden de magnitud los tiempos obtenidos con respecto a la implementación secuencial en MATLAB.

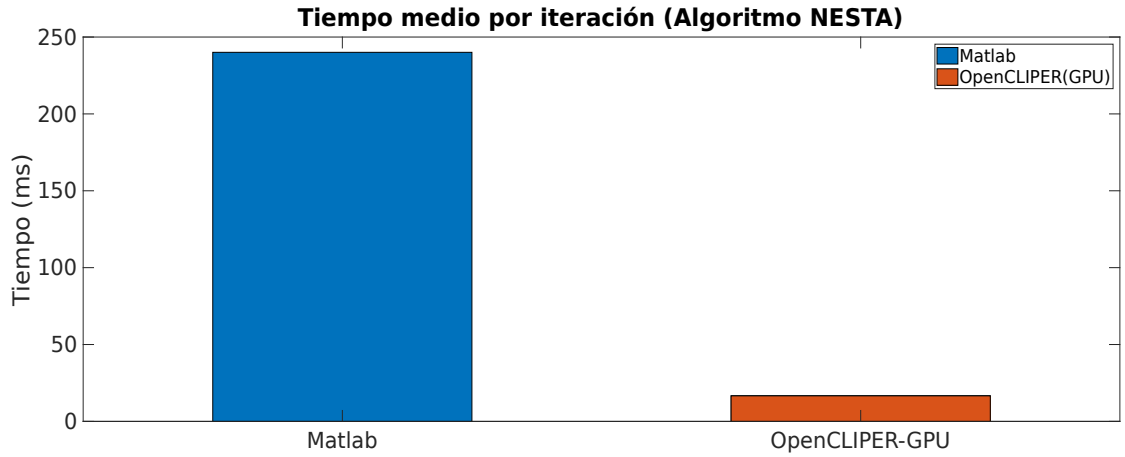

Figura 5.4: Tiempo medio por iteración del algoritmo NESTA. Comparación de la implementación en MATLAB (azul) con la implementación en el *framework* OpenCLIPER (naranja) ejecutada en GPU.

5.2 Resultados del algoritmo NUFFT

En los experimentos de ejecución del algoritmo NUFFT implementado se utiliza un espacio K no cartesiano 3D (272 x 32 x 1560) y 20 *coils*. Además, se dispone de la trayectoria no cartesiana de adquisición de los datos normalizada entre [-0.5 , 0.5] en cada dirección (k_x, k_y y k_z), siendo 0 el centro del espacio K. Esta trayectoria sigue el patrón de una *phyllotaxis* y se encuentra representado en la Figura 5.5.

Los valores de los parámetros utilizados de entrada al algoritmo NUFFT se muestran en la Tabla 5.2. Así mismo, se indica una breve explicación del significado de los mismos.

Parámetro	Valor	Explicación
ovsf	2	Factor de sobremuestreo
kernelSize	4	Radio del kernel de convolución
sectorSize	8	Nº de puntos del sector en cada dimensión
imgDims	[128, 128, 128]	Dimensiones de la imagen resultante

Tabla 5.2: Valores de los parámetros de entrada al algoritmo NUFFT utilizados.

En la imagen 5.6 se muestran los resultados obtenidos con este algoritmo. El espacio K no cartesiano, mediante el *gridding*, es interpolado en una rejilla cartesiana sobremuestreada de dimensiones [256 x 256 x 256]. El corte central de esta rejilla es mostrado en la Figura 5.6 A.

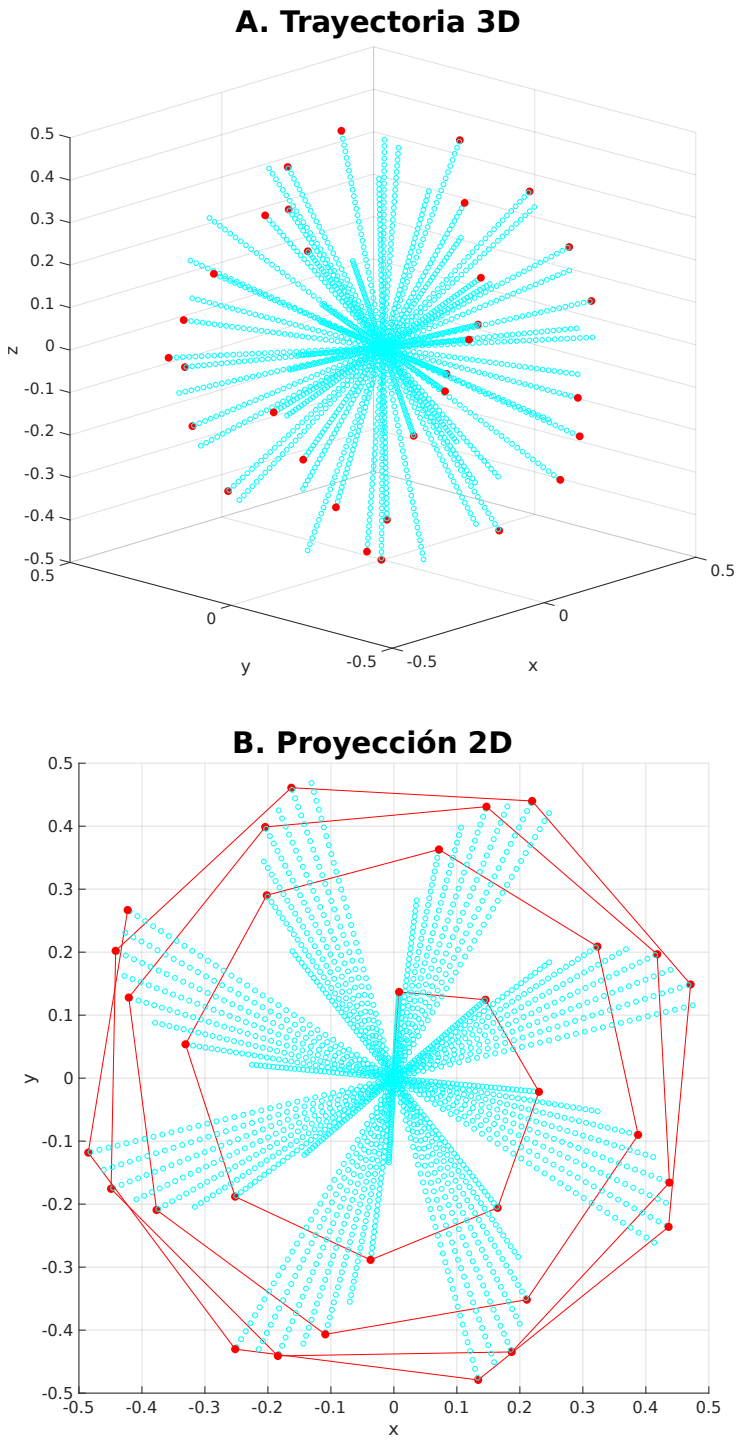

Figura 5.5: Arriba, todos los radios de los rayos que forman la primera espiral espiral, en rojo se encuentran representados el primer punto de los 32 rayos. Abajo, la proyección en 2D, donde los primeros puntos de los 32 rayos que forman la primera espiral han sido unidos por una línea roja.

Por otro lado, el corte central en el plano coronal de la imagen obtenida con los datos de un *coil* tras aplicar el operador $NUFFT^H$ se muestra en la Figura 5.6 B. Por último, una vez el operador es aplicado a los datos de todos los *coils*, dado que no disponemos de los mapas de sensibilidad de dichos *coils*, se realiza una reconstrucción *Sum of Square* (SoS) para obtener una imagen final (ver Figura 5.6 C).

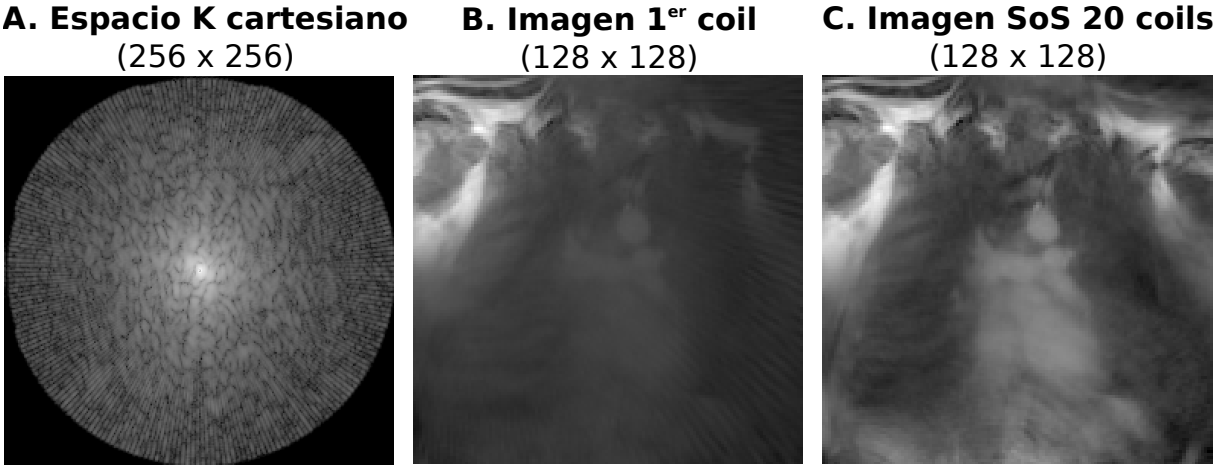

Figura 5.6: Resultados del algoritmo NUFFT implementado. **A**, espacio K cartesiano creado a partir del espacio K no cartesiano. **B**, imagen reconstruida con la aplicación del operador $NUFFT^H$ para el primer *coil*. **C**, imagen reconstruida mediante *Sum of Squares* (SoS) tras la aplicación del operador $NUFFT^H$ a todos los *coils*. Todas las imágenes se corresponden con el corte central del volumen en el plano coronal.

Es cierto que, aparentemente, la imagen resultante está muy artefactada, pero es debido a que estas imágenes se obtienen a partir de un espacio K submuestreado y no se reconstruyen mediante ningún algoritmo de optimización. Es decir, la utilización de un algoritmo de optimización junto con el operador $NUFFT^H$ produciría una mejora considerable en la calidad de la imagen.

Igual que en el caso anterior, para demostrar la corrección de la implementación se calcula el SSIM entre la imagen obtenida mediante nuestra implementación y la obtenida mediante la librería gpuNUFFT [12], ampliamente utilizada en la comunidad científica. El índice es calculado para el volumen 3D resultante, obteniéndose un valor de 0.9970.

5.2.1 Medidas de rendimiento

El rendimiento del algoritmo implementado se evalúa también en términos de tiempo de ejecución. Para ello. se comparan los tiempos de ejecución obtenidos con la implementación planteada ejecutada en GPU con los obtenidos mediante la librería gpuNUFFT. La ejecución del algoritmo implementado se realiza en una máquina con procesador 8xIntel® Core™ i7-4790 3.60 GHz y 16 GB de memoria RAM. Así mismo, en la máquina estaba disponible una GPU AMD *Hawaii* de 4 GB de memoria RAM. Mientras que la ejecución del algoritmo de la gpuNUFFT se realiza en una máquina 8xIntel® Core™ i7-7700 3.60 GHz y 16 GB de memoria RAM, en la cual estaba disponible una GPU NVIDIA GeForce GTX 1050 de 2 GB de memoria RAM. La utilización de dos máquinas distintas se debe a que la librería gpuNUFFT, al estar programada en CUDA, necesita una GPU NVIDIA para ejecutarse. Por último, para medir los tiempos de forma fiable se realizan 100 ejecuciones del algoritmo.

En la Figura 5.7 se muestran los valores de tiempo medio obtenidos en el experimento planteado. En este caso, como se puede observar, el algoritmo implementado en OpenCLIPER obtiene peores prestaciones en cuanto a tiempos de ejecución que las obtenidas mediante la librería publicada gpuNUFFT [12].

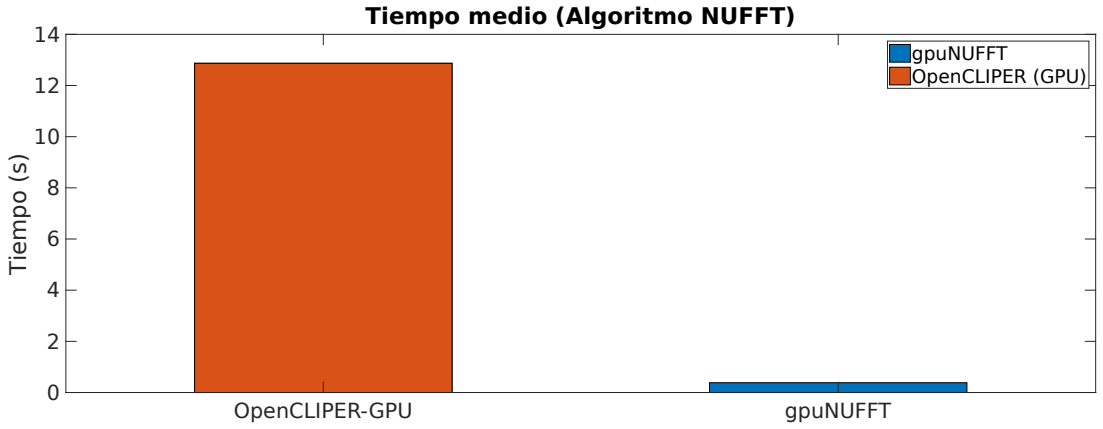

Figura 5.7: Tiempo medio del algoritmo NUFFT. Comparación de la implementación de la librería gpuNUFFT (azul) con la implementación en el *framework* OpenCLIPER (naranja) ejecutada en GPU.

Es necesario tener en cuenta que lo que hemos planteado en este TFM es una aproximación todavía no optimizada del problema, con el objetivo de disponer de una implementación del algoritmo independiente del dispositivo. Además, la librería gpuNUFFT se encuentra ya

muy madura y optimizada. No obstante, tras realizar una estudio detallado del rendimiento del algoritmo mediante el *software* CodeXL [30], hemos podido comprobar que en nuestra implementación existe mucho margen de mejora.

En la Tabla 5.3 se muestra información detallada del tiempo que tarda en ejecutarse cada uno de los *kernel* implementados. Como se puede observar, el *kernel* `convolution` es el que supone más carga computacional, con un 98.57 % del tiempo total. Actualmente, los valores de ocupación obtenidos de este *kernel* son del 40 %, lo que indica que es posible realizar implementaciones más eficientes del mismo. Para ello, en primer lugar sería necesario utilizar contadores para conocer con detalle que partes del *kernel* tardan más en realizarse. Posteriormente, con la información obtenida de los contadores, se podrían realizar optimizaciones haciendo uso de distintas zonas de memoria disponibles en OpenCL (memoria local, memoria privada, etc.), las cuales son más cercanas a los elementos de cálculo y, por consiguiente, más rápidas.

Kernel Name	Device	# of calls	Total Time(ms)	% of Total Time	Avg Time(ms)	Max Time(ms)	Min Time(ms)
convolution	Hawaii	1	12712.7683	98.57	12712.76830	12712.76830	12712.76830
sortedData	Hawaii	1	81.17274	0.63	81.17274	81.17274	81.17274
fft_back	Hawaii	3	71.22504	0.55	23.74168	62.51081	0.93911
performDensityCompensation	Hawaii	1	17.06030	0.13	17.06030	17.06030	17.06030
performFFTShift	Hawaii	2	13.95896	0.11	6.97948	12.40711	1.55185
performCrop	Hawaii	1	0.45644	0.00	0.45644	0.45644	0.45644
performDeapodization	Hawaii	1	0.39630	0.00	0.39630	0.39630	0.39630
performFFTScaling	Hawaii	1	0.30504	0.00	0.30504	0.30504	0.30504

Tabla 5.3: *Kernels* implementados en el algoritmo. Para cada uno se indica el número de llamas y el tiempo de ejecución.

Capítulo 6

Conclusiones

6.1 Conclusiones

En este Trabajo Fin de Máster se ha planteado la implementación de dos algoritmo descritos en el estado del arte para la reconstrucción de MRI, empleando técnicas de computación paralela en GPU. De forma específica, los algoritmos implementados son el algoritmo de reconstrucción NESTA (a partir del código publicado en MATLAB), y el algoritmo NUFFT (a partir de la librería gpuNUFFT desarrollada en C++ y CUDA).

El lenguaje de programación OpenCL para computación en GPU tiene la ventaja de permitir el desarrollo de *software* portable, que pueda ser ejecutado en distintas plataformas y con independencia del fabricante del *hardware*. Debido a estas ventajas, y a que actualmente no existen implementaciones de estos algoritmos en este lenguaje, ha sido el elegido para realizar la implementación. Además, se ha utilizado como apoyo el *framework* OpenCLIPER, lo que permite la integración de estos algoritmos con otros disponibles en este *framework* de procesado y reconstrucción de imagen. A su vez, la inclusión de los algoritmos en el *framework*, permiten dotar a éste de mayor potencial y aplicabilidad dentro de la comunidad.

Tras realizar la implementación de los algoritmos, se han analizado las imágenes de resonancia magnética reconstruidas. Estas imágenes permiten validar el correcto funcionamiento del los algoritmos, a la vez que demuestran la utilidad de los mismos en la reconstrucción de MRI.

A su vez, se ha analizado el rendimiento de las implementaciones planteadas en términos de tiempo de ejecución. Con estas medidas se ha podido comprobar que la arquitectura subyacente

(GPU, CPU, fabricante, etc.) influye notablemente en el rendimiento final obtenido. Los tiempos de ejecución obtenidos con el algoritmo NESTA implementado suponen una mejora razonable con respecto a la implementación secuencial en MATLAB de este algoritmo. Por otro lado, es cierto que la implementación del algoritmo NUFFT adolecen de ciertas limitaciones relacionadas, principalmente, con los tiempos de ejecución en comparación con los obtenidos mediante la librería gpuNUFFT. No obstante, es necesario destacar que lo planteado en el presente trabajo es una aproximación todavía no optimizada del problema, existiendo, por consiguiente, mucho margen de mejora.

6.2 Líneas futuras

Siguiendo el trabajo presentado en esta memoria, podría ser posible añadir ciertas mejoras a las implementaciones, bien para añadir funcionalidad a los algoritmos o bien para reducir los tiempos de ejecución de los mismos.

En cuanto al algoritmo NESTA, se podría añadir funcionalidad con la incorporación de otros operadores *sparse*, como por ejemplo, variación total espacial o la transformada *Wavelet*. Así mismo, dado que en las adquisiciones cartesianas típicamente el submuestreo se realiza únicamente en la dirección de codificación de fase, trabajar en el espacio híbrido permitiría reducir la carga computacional del algoritmo.

En cuanto al algoritmo NUFFT, la implementación planteada es una aproximación no optimizada al problema, por lo que existe mucho espacio para mejorar el rendimiento del mismo. Concretamente, se podrían realizar importantes optimizaciones utilizando distintas zonas de memoria más rápidas. Además, sería deseable generalizarlo para poder ser utilizado en adquisiciones 2D y/o adquisiciones dinámicas. Por otro lado, poder aplicar el algoritmo a los datos de todas las bobinas en paralelo reduciría notablemente los tiempos de ejecución de la reconstrucción.

Además, los algoritmos implementados se pondrían integrar en *pipelines* más complejos de reconstrucción de MRI. El algoritmo NESTA podría utilizarse junto con un algoritmo de compensación de movimiento para mejorar las prestaciones de la reconstrucción. Por otro lado,

el algoritmo NUFFT se podría utilizar como operador de codificación de algún algoritmo de optimización iterativo (por ejemplo, NESTA) para poder realizar mejores reconstrucciones de adquisiciones no cartesianas.

Apéndice A

Overview de OpenCLIPER

OpenCLIPER es un *framework* basado en OpenCL para el procesamiento y la reconstrucción de imágenes médicas [11]. Su objetivo es simplificar las tareas de programación asociadas a OpenCL, para permitir que los desarrolladores se centren en el desarrollo de algoritmos propiamente dicho. De esta forma, OpenCLIPER está diseñado como un conjunto de clases que brindan tres servicios principales al desarrollador:

- **Gestión de los dispositivos informáticos:** en OpenCL el concepto del dispositivo no se maneja automáticamente, como sucede en CUDA. Además, OpenCL introduce el concepto de plataformas para abordar el problema de dar soporte a diferentes proveedores de hardware. Todas estas tareas de gestión, que en OpenCL deberían ser realizadas por el programador, son realizadas por el *framework* de forma transparente.

- **Almacenamiento y manipulación de datos:** otro problema asociado con este tipo de programación se debe al hecho de que el *host* y el dispositivo tienen sus propias zonas de memoria separadas. Por lo tanto, los datos deben transferirse del *host* al dispositivo antes de procesarlos, y del dispositivo al host posteriormente. OpenCLIPER ofrece funcionalidades al programador para facilitar sustancialmente esta tarea.

- **Manejo de algoritmos:** en OpenCL, además, es necesario cargar los *kernels*, compilarlos, verificar posibles errores, realizar un seguimiento de ellos en tiempo de ejecución, etc. OpenCLIPER también ha sido diseñado para facilitar el trabajo de lanzamiento y gestión de *kernels*.

Es decir, el *framework* maneja automáticamente el descubrimiento y la inicialización del dispositivo, las transferencias de datos hacia y desde el dispositivo, la carga del *kernel*, la compilación y el informe de errores. La estructura de clases que presenta el *framework* se muestra en la Figura A.1.

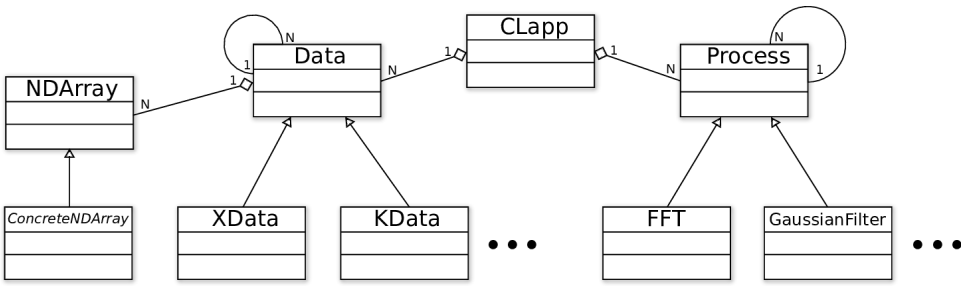

Figura A.1: Diagrama de clases del framework OpenCLIPER [11].

A continuación, se realiza una descripción más detallada de las características de las principales clases, mostradas en la Figura A.1, que dan soporte al *framework*. Estas son:

- **`CLapp:`** es la clase principal del *framework*. Ofrece al usuario una interfaz al dispositivo OpenCL y almacena la información relativa a las plataformas, dispositivos y cola de comandos. Esta clase también es la encargada de la gestión de memoria del dispositivo, así como de las transferencias de datos. Para minimizar el número de transferencias de datos entre el host y el dispositivo, automáticamente mapea los datos en ambas memorias. Así mismo, contiene la lista de los objetos de datos que se procesaran en cada caso específico.

- **`Data:`** es una clase abstracta. Los objetos de sus clases derivadas contienen un conjunto de imágenes, volúmenes o datos n-dimensionales en el caso más general. Cada objeto de tipo derivado de **`Data`** contiene uno o más objetos de tipo **`NDArray`**, que no necesitan ser iguales en tamaños ni dimensiones. Además, soporta tipos de datos tanto reales como complejos. OpenCLIPER proporciona dos especializaciones de propósito general para la clase **`Data`**, llamadas **`XData`** (para datos con una interpretación física directa) y **`KData`** (para datos del espacio K).

- **`NDArray:`** representa una señal, una imagen, un volumen o una estructura de datos n-dimensional. Dado que, en principio, no se conoce el tipo de datos específico en el que se desea almacenar los datos, esta clase es definida como abstracta. Debido a esto, únicamente

se limita a definir los atributos y métodos comunes para todos los tipos de datos posibles, y a la creación de objetos de la verdadera clase contenedora, que es `ConcreteNDArray`.

- **`ConcreteNDArray:`** es la clase que verdaderamente almacena los datos. No está destinada a ser utilizada por los usuarios del *framework*, sino solo para contener datos en bruto y detalles que dependen del tipo de datos máquina.

- **`Process:`** es una clase abstracta de la cual los desarrolladores deberían derivar para implementar sus propios procesos. Representa una interfaz estándar a los algoritmos que procesan los datos, de modo que los desarrolladores no necesitan ningún conocimiento previo sobre sus componentes internos para comenzar a trabajar con ellos. Además, los distintos procesos que pueden formar un algoritmo más complejo se pueden conectar en cascada (uno tras otro) sin que suponga ninguna penalización de rendimiento.

Para entender mejor la arquitectura y el funcionamiento del *framework*, a continuación se muestra un ejemplo, ofrecido en la documentación del mismo, con la implementación de un algoritmo sencillo que permite obtener el negativo de una imagen.

Listado A.1: Ejemplo simple de OpenCLIPER (programa principal) [11].

```
#include <OpenCLIPER/XData.hpp>
#include <OpenCLIPER/processes/examples/Negate.hpp>
#include <iostream>
#include <string>

using namespace OpenCLIPER;
int main(int argc, char *argv[]) {
  // Step 0: get a new OpenCLIPER app
  std::shared_ptr<CLapp> pCLapp = std::make_shared<CLapp>();

  try {
    // Step 1: initialize computing device
    CLapp::PlatformTraits platformTraits;
    CLapp::DeviceTraits deviceTraits;
    pCLapp->init(platformTraits,deviceTraits);

    // Step 2: load OpenCL kernel(s)
    pCLapp->loadKernels("examples/negate.cl");
```

```
    // Step 3: load input data
    std::shared_ptr<Data> pIn(new XData(std::string("Cameraman.tif"), type_index(typeid(
        realType))));

    // Step 4: create output with same size as input
    std::shared_ptr<Data> pOut(new XData((dynamic_pointer_cast<XData>(pIn)), false));

    // Set 5: register input and output in our CL app
    DataHandle inHandle = pCLapp->addData(pIn);
    DataHandle outHandle = pCLapp->addData(pOut);

    // Step 6: create new process bound to our CL app
    // and set its input/output data sets
    std::unique_ptr<Process> pProcess(new Negate(pCLapp));
    pProcess->setInHandle(inHandle);
    pProcess->setOutHandle(outHandle);

    // Step 7: initialize & launch process
    pProcess->init();
    pProcess->launch();

    // Step 8: get data back from computing device
    pCLapp->device2Host(outHandle, SyncSource::BUFFER_ONLY);

    // Step 9: save output data
    auto outputData=dynamic_pointer_cast<XData>(pCLapp->getData(outHandle));
    outputData->save("output.png", SyncSource::BUFFER_ONLY);

    // Step 10: clean up
    pProcess.reset(nullptr);
    pCLapp->delData(inHandle);
    pCLapp->delData(outHandle);
    pCLapp = nullptr;
  } catch (std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
  }
}
```

Listado A.2: Ejemplo simple de OpenCLIPER (cabecera de la clase `Process`) [11].

```
#ifndef INCLUDE_OPENCLIP_NEGATE_HPP_
#define INCLUDE_OPENCLIP_NEGATE_HPP_

#include <OpenCLIPER/CLapp.hpp>
#include <OpenCLIPER/Process.hpp>
```

```
namespace OpenCLIPER {
class Negate : public OpenCLIPER::Process {
  public:
    Negate(std::shared_ptr<OpenCLIPER::CLapp> pCLapp): Process(pCLapp) {};
    void init();
    void launch(bool profilingEnabled = false);
};
} //namespace OpenCLIPER
#endif /* INCLUDE_OPENCLIP_NEGATE_HPP_ */
```

Listado A.3: Ejemplo simple de OpenCLIPER (Implementación de la clase **Process**) [11].

```
#include <OpenCLIPER/processes/examples/Negate.hpp>

namespace OpenCLIPER {
void Negate::init() {
    kernel = getApp()->getKernel("negate_kernel");
    queue = getApp()->getCommandQueue();
}

void Negate::launch(bool profilingEnabled) {
    // Set input and output OpenCL buffers on device memory
    cl::Buffer* pInBuf = getInput()->getNDArray(0)->getDeviceBuffer();
    cl::Buffer* pOutBuf = getOutput()->getNDArray(0)->getDeviceBuffer();

    // Set kernel parameters
    kernel.setArg(0, *pInBuf);
    kernel.setArg(1, *pOutBuf);

    // Set kernel work items size: number of pixels to process is image width x height
    cl::NDRange globalSizes = {NDARRAYWIDTH(getInput()->getNDArray(0)) * NDARRAYHEIGHT(
        getInput()->getNDArray(0))};

    // Execute kernel
    queue.enqueueNDRangeKernel(kernel, cl::NullRange, globalSizes, cl::NDRange(), NULL,
        NULL);
}
}
```

Listado A.4: Ejemplo simple de OpenCLIPER (kernel) [11].

```
  #include <OpenCLIPER/kernels/hostKernelFunctions.h>

__kernel void negate_kernel(__global realType* input, __global realType* output) {
```

```
    int num = get_global_id(0);
    output[num] = (1.0 - input[num]);
}
```

Bibliografía

[1] Z. P. Liang and P. C. Lauterbur. *Principles of Magnetic Resonance Imaging.* IEEE Press, 2000.

[2] M. K. Stehling, R. Turner, and P. Mansfield. Echo-planar imaging: magnetic resonance imaging in a fraction of a second. *Science*, 254(5028):43–50, 1991.

[3] K. P. Pruessmann, M. Weiger, M. B. Scheidegger, and P. Boesiger. SENSE: sensitivity encoding for fast MRI. *Magnetic resonance in medicine*, 42(5):952–962, 1999.

[4] M. A. Griswold, P. M. Jakob, R. M. Heidemann, M. Nittka, V. Jellus, J. Wang, B. Kiefer, and A. Haase. Generalized autocalibrating partially parallel acquisitions (GRAPPA). *Magnetic Resonance in Medicine*, 47(6):1202–1210, 2002.

[5] T. Jeffrey, P. Boesiger, and K. P. Pruessmann. k-t BLAST and k-t SENSE: dynamic mri with high frame rate exploiting spatiotemporal correlations. *Magnetic Resonance in Medicine*, 50(5):1031–1042, 2003.

[6] F. Huang, J. Akao, S. Vijayakumar, G. R. Duensing, and M. Limkeman. k-t GRAPPA: A k-space implementation for dynamic MRI with high reduction factor. *Magnetic Resonance in Medicine*, 54(5):1172–1184, 2005.

[7] M. Lustig, D. Donoho, and J. M. Pauly. Sparse MRI: The application of compressed sensing for rapid MR imaging. *Magnetic Resonance in Medicine*, 58(6):1182–1195, 2007.

[8] U. Gamper, P. Boesiger, and S. Kozerke. Compressed sensing in dynamic MRI. *Magnetic Resonance in Medicine*, 59(2):365–373, 2008.

[9] Khronos Group. *The OpenCL Specification. Versión: 1.2. Revisión: 19*, 2012. `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`.

[10] S. Becker, J. Bobin, and E. J. Candès. NESTA: A fast and accurate first-order method for sparse recovery. *SIAM Journal on Imaging Sciences*, 4(1):1–39, 2011.

[11] F. Simmross-Wattenberg, M. Rodriguez-Cayetano, J. Royuela del Val, E. Martin-Gonzalez, E. Moya-Saez, M. Martin-Fernandez, and C. Alberola-Lopez. Opencliper: an OpenCL-based C++ framework for overhead-reduced medical image processing and reconstruction on heterogeneous devices. *IEEE Journal of Biomedical and Health Informatics*, 2018.

[12] F. Knoll, A. Schwarzl, C. Diwoky, and D. K. Sodickson. gpuNUFFT - an open-source GPU library for 3D gridding with direct matlab interface. page 4297, 2014.

[13] The MathWorks - MATLAB and Simulink for Technical Computing. Website. Último acceso febr. 2019. `http://www.mathworks.com/`.

[14] M. Wolff. KDevelop 4.7. Website. Último acceso febr. 2019. `https://www.kdevelop.org/`.

[15] L. Lamport. LaTeX Project. Website. Último acceso febr. 2019. `https://www.latex-project.org/`.

[16] L. Cordero. *Estimación del tensor de esfuerzo del miocardio. Integración de propiedades físicas del problema e imagen de resonancia magnética multimodal en un modelo estocástico.* PhD thesis, Universidad de Valladolid, 2011.

[17] P. Suetens. *Fundamentals of Medical Imaging.* Cambridge, 2006.

[18] R. A. Poley. Fundamentals physics of MR Imaging. *RadioGraphics*, 25(4):1087–1099, 2005.

[19] M. Lustig, D. L. Donoho, J. M Santos, and J. M. Pauly. Compressed sensing MRI. *IEEE Signal Processing Magazine*, 25(2):72–82, 2008.

[20] J. Royuela-del Val, L. Cordero-Grande, F. Simmross-Wattenberg, M. Martín-Fernández, and C. Alberola-López. Nonrigid groupwise registration for motion estimation and compensation in compressed sensing reconstruction of breath-hold cardiac cine mri. *Magnetic resonance in medicine*, 75(4):1525–1536, 2016.

[21] S. Sanz-Estébanez, E. Moya-Sáez, J. Royuela-del Val, and C. Alberola-López. On the construction of non linear adjoint operators: Application to L1-penalty dynamic image reconstruction. *CASEIB. Ciudad Real, España*, 2018.

[22] J. M. Pauly. Chapter 5: Reconstruction of non-cartesian data. https://users.fmrib.ox.ac.uk/~karla/reading_group/lecture_notes/AdvRecon_Pauly_read.pdf.

[23] B. L. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*. Elsevier, 2013.

[24] Nvidia Corporation. What is GPU-Accelerated computing?. Website. Último acceso jun. 2017. http://www.nvidia.com/object/what-is-gpu-computing.html.

[25] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Preceedings of the IEEE*, 96(5):879–899, 2008.

[26] F. Simmross-Wattenberg. *Introducción a la programación heterogénea CPU-GPU en OpenCL*. Informe técnico, Universidad de Valladolid, 2015.

[27] J. Royuela-del Val. *Software solutions for two computationally intensive problems: reconstruction of dynamic MR and handling of alpha-stable distributions.* PhD thesis, Universidad de Valladolid, 2017.

[28] clBLAS. Website. Último acceso febr. 2019. https://github.com/clMathLibraries/clBLAS.

[29] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, et al. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[30] Advanced Micro Devices, Inc. CodeXL. Website. Último acceso febr. 2019. https://gpuopen.com/compute-product/codexl/.