UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR

DE INGENIEROS DE TELECOMUNICACIÓN

# PROYECTO FIN DE CARRERA

INGENIERO TÉCNICO DE TELECOMUNICACIÓN

TELEMÁTICA

# "ESTIMACIÓN DE PARÁMETROS EN RESONANCIA MAGNÉTICA DE DIFUSIÓN SOBRE GPU"

**Autor:** Silvia Pedrón Hermosa

**Tutor:** Federico Simmross Wattenberg, Santiago Aja Fernández

Septiembre de 2015

| | |
|---|---|
| Título: | Estimación de parámetros en Resonancia Magnética de Difusión sobre GPU |
| Autor: | Silvia Pedrón Hermosa |
| Tutores: | Federico Simmross Wattenberg |
| | Santiago Aja Fernández |
| Departamentos: | Ingeniería Telemática |
| | Teoría de la Señal y Comunicaciones |

Miembros del tribunal

| | |
|---|---|
| Presidente: | Carlos Alberola López |
| Secretario: | Federico Simmross Wattenberg |
| Vocal: | Santiago Aja Fernández |
| Suplente: | Marcos Martín Fernández |

Fecha de lectura: 11 de Septiembre de 2015

Calificación:

# ÍNDICE GENERAL

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

# ÍNDICE DE ECUACIONES

# ÍNDICE DE CÓDIGOS

# AGRADECIMIENTOS

Después de tanto tiempo hemos llegado a este punto. Y digo hemos por que no lo hubiera conseguido yo sola.

En primer lugar me gustaría dar las gracias a mis padres. Porque si no fuera por ellos yo me habría echado atrás hace mucho tiempo. Que hubiera sido de mí sin su "Venga campeona, que tú puedes" a la puerta de casa cuando me dispongo a afrontar cualquier reto de mi vida, como por ejemplo terminar la carrera. A mi madre por que ha sufrido conmigo todos y cada uno de mis mejores y peores momentos, todos en silencio y con buena cara para no desanimarme. Y a mi padre, porque sé lo importante que es para él que yo haya llegado hasta aquí, siempre calmándome con sus palabras aunque su mirada no reflejara serenidad precisamente. ¡Ya tienes un ingeniero en la familia!

A mi hermano Julio, por su amor y su cariño, por sus ganas de aportar su granito de arena en todo momento y por esos abrazos que lo decían todo.

A Rober, por ser un gran compañero de viaje que parecía tan interminable y por la paciencia y el aguante que ha demostrado. Sin ti no hubiera sido posible aunque no te lo creas.

Agradecer también a mis abuelas, que se han ganado el cielo con tantas plegarias, y seguro que se encuentran orgullosas de tener una nieta ingeniero.

Por último, a mis tutores Federico y Santiago, que me han ayudado y guiado en este largo proceso.

A todos, mis más sinceras gracias.

# RESUMEN

Teniendo en cuenta que las actuales unidades de procesamiento gráfico disponen de una interfaz de programación que permite utilizarlas para tareas de propósito general (GPGPU), y que existen algoritmos que se pueden ejecutar por partes, para finalmente unirlas y obtener el resultado correcto. El propósito de este proyecto es muy claro, acoplar esos dos campos de interés general.

Este proyecto busca acelerar la ejecución de un algoritmo propuesto utilizando la interfaz de programación OpenCL. Para ello se ha estudiado la formulación y posterior codificación de estos algoritmos en el modelo de programación paralela de OpenCL.

En particular el estudio trata sobre DTI (*Diffusion Tensor Imaging*), una de las variantes más comunes de la resonancia magnética de difusión. Esta modalidad de resonancia magnética está basada en la medición de la difusión de las moléculas de agua en los tejidos. La difusión que se da en la materia blanca viene determinada por la orientación de los axones que constituyen las fibras nerviosas. A partir de la información proporcionada por una imagen más precisa y completa del cerebro, DTI, se podrá diagnosticar con mayor rapidez y precisión. Además se pueden obtener diferentes medidas desde el análisis de la difusión.

Conociendo los beneficios que tienen las unidades de procesamiento gráfico, la implementación de este algoritmo utilizando la interfaz de programación OpenCL es muy útil y conveniente. En este proyecto se pretenden mostrar las ventajas de utilizar las tarjetas gráficas para acelerar la ejecución de algoritmos altamente paralelizables.

**Palabras clave**: Unidad de procesamiento gráfico, algoritmo paralelizable, OpenCL, Imagen con tensor de difusión, resonancia magnética.

# ABSTRACT

Taking into account that current graphics processing units manufactured provide a programming interface that allows them to perform general purpose computation (GPGPU), and there are algorithms that can be executed by portions to finally unite them and get the correct result. The aim of this project is very clear, coupling these two fields of general interest.

This project seeks to speed up the algorithm proposed using OpenCL programming interface. To this end, it has been studied sequential formulations of these algorithms and their related code using OpenCL parallel programming model.

Specifically, the study is about DTI, one of the most common techniques employed in diffusion magnetic resonance imaging. This modality of MRI is based on the measuring of diffusion of water molecules within the tissues. The diffusion that occurs in the white matter is aligned to the orientation of axons that compose nerve fibers. From the information provided by a more accurate and complete picture of the brain, DTI, it may be diagnosed more quickly and accurately. Furthermore, different measures can be obtained from the analysis of diffusion.

Knowing the benefits that have the graphics processing units, the implementation of this algorithm using OpenCL programming interface is very useful and convenient. This project is aim to show the benefits to use graphics card to accelerate the execution of highly parallel algorithms.

**Keywords**: Graphics Processing Unit, parallel algorithm, OpenCL, Diffusion Tensor Imaging, magnetic resonance.

# 1. INTRODUCCIÓN

La resonancia magnética (RM) es un examen médico, que consiste en la obtención de imágenes de la zona del cuerpo a estudiar. Para ello se utiliza un imán primario de gran capacidad, que genera un campo magnético constante e intenso, y un emisor/receptor de ondas de radio para emitir la radiación electromagnética a una determinada frecuencia de resonancia. Esta técnica se ha convertido en una importante herramienta de diagnóstico cuyo uso ha aumentado significativamente en las tres últimas décadas ya que se diferencia de otras pruebas radiológicas al ser una técnica no invasiva, no usar radiaciones ionizantes y tener una gran sensibilidad diagnóstica. Aunque también es un procedimiento más lento, de mayor coste, y con menor disponibilidad.

En los años 90, surgió una nueva técnica de resonancia magnética, la DTI (*Diffusion Tensor Imaging)* que aporta información complementaria a las secuencias habituales de RM. La DTI permite visualizar y analizar los grandes tractos de sustancia blanca, permitiéndose así examinar la integridad o la direccionalidad de las vías de dicha sustancia. De esta manera, al revelar una imagen más precisa y completa del cerebro, se podrá diagnosticar con rapidez y precisión evidencias de, incluso, pequeñas y ligeras lesiones cerebrales.

Esta técnica utiliza la direccionalidad del desplazamiento de las moléculas del agua, en el interior de los tejidos, a lo largo de los tractos de sustancia blanca, y cuantifica el grado de anisotropía que poseen éstos [1]. Si los tractos son muy densos y muestran un mayor grado de anisotropía, la difusión es direccionalmente dependiente. Si por el contrario, sus propiedades físicas son idénticas en todas las direcciones, y por tanto el agua se desplaza libremente, entonces mostrará un mayor grado de isotropía.

En los medios anisotrópicos se utiliza el tensor de difusión para caracterizar la difusión. El modelo del tensor de difusión consiste en un tensor simétrico de 3x3 de difusión con seis grados de libertad que junto con los datos de la imagen, forman siete imágenes

necesarias para cada *slice*. Utilizando más de seis direcciones mejorará la fiabilidad en la medida del tensor.

Si añadimos dichos gradientes y señales a la ecuación desarrollada por Skejskal – Tanner [2], obtendríamos un sistema de ecuaciones, teniendo tantas ecuaciones como tensores hubiese. La solución del tensor de difusión sería el resultado de este sistema para cada vóxel. La cantidad de vóxels es variable, pero considerable, y la implementación del problema está basada en la repetición de una operación sobre cada uno de ellos independientemente del resto, de tal manera que ejecutándolo por separado y uniéndolo todo al final, obtendríamos el mismo resultado. De esta forma se plantea la resolución del algoritmo de manera paralela en vez de secuencial.

Generalmente, este problema se aborda sobre las CPU (*Central Processing Unit*) de los ordenadores convencionales. Estos han evolucionado centrándose en la ejecución secuencial, teniendo como resultado, para esta aplicación, unos tiempos de ejecución viablemente mejorables.

En los últimos años, gracias a la implantación masiva de la fotografía y el vídeo digital, de programas CAD (*Computer-aided design)* y sobre todo por el auge de la industria de los videojuegos, la evolución de las tarjetas gráficas ha sido vertiginosa, siendo actualmente más eficientes para el cálculo de información gráfica que las CPU. Este crecimiento hizo que se acuñara el término GPU (*Graphics Processing Unit*). Un coprocesador dedicado a los gráficos con una arquitectura altamente paralela, con una gran cantidad de unidades de procesamiento capaces de realizar operaciones gráficas sobre muchos datos simultáneamente.

Además, las últimas generaciones de tarjetas gráficas permiten un estilo de programación orientado a las GPGPU (*General-Purpose Computing on Graphics Processing Units*), lo que abre las puertas a la implementación de un mayor número de algoritmos. Al mismo tiempo aparecen lenguajes de programación sobre GPU adaptados a esta nueva serie de tarjetas gráficas. Tanto AMD [3], como Nvidia [4], tienen API dirigidas a GPGPU, OpenCL [3] [5] y CUDA [4] [6], respectivamente.

Todo esto nos lleva a la necesidad de traducir los algoritmos convencionales a operaciones en paralelo y de realizar sus respectivas implementaciones usando los lenguajes adecuados. *Este Proyecto de Fin de Carrera surge, conforme a lo dicho anteriormente, de la necesidad de implementar el problema de la estimación del tensor de difusión sobre GPU, aprovechando las características propias de esta tecnología, como la*

*potencia de cálculo y el gran paralelismo. Además de considerar los lenguajes de programación, OpenCL y CUDA, relativamente novedosos en el ámbito médico.*

## 1.1. OBJETIVOS

El principal objetivo, en el desarrollo de este Proyecto Fin de Carrera, es reducir el tiempo para la estimación del tensor de difusión en resonancia magnética de difusión, empleando para ello procesado paralelo en GPU. Según un análisis previo podemos establecer que tanto por su orientación a procesamiento en paralelo, como por la potencia que adquiere la nueva generación de tarjetas gráficas, es deseable la necesidad de mejorar las herramientas vigentes.

Debido a esta objetivo general, se propone estudiar e investigar la estimación de parámetros en resonancia magnética de difusión sobre GPU, descomponiéndolo en los siguientes objetivos secundarios:

1. La creación de un marco que simplifique y favorezca el estudio futuro de técnicas de resonancia magnética con la ayuda de GPU. Concretamente de la estimación del tensor de difusión en Resonancia Magnética de Difusión sobre GPU, que proporciona información de los tractos de sustancia blanca en el cerebro.

2. Aportar soluciones para implementar este problema en cada tipo de *hardware*, así como comparar los diversos lenguajes computacionales existentes en este ámbito.

3. Estudiar la ventaja que supone emplear tecnologías que aprovechan la potencia de los procesadores gráficos, frente al método habitual implementado sobre CPU.

4. Proporcionar nuevas herramientas de computación rápida a investigadores de MR (*Magnetic Resonance*), de esta forma, favorecer el desarrollo de nuevos avances científicos y tecnológicos en medicina.

A partir de estos propósitos parciales, los conocimientos adquiridos serán de gran ayuda para poder dotar, al usuario interesado, de herramientas superiores para proseguir con sus investigaciones. Ya sea en el desarrollo de aplicaciones GPU y, por ende, elegir cuál es el lenguaje que más se adecúa a sus medios o necesidades, pero sobre todo para dotar a

los investigadores de RM nuevas herramientas que facilitan el estudio de enfermedades crónicas del sistema nervioso, tumores cerebrales, o accidentes cerebrovasculares.

## 1.2. FASES Y MÉTODOS

La metodología a seguir está condicionada por los propósitos parciales y por el objetivo final.

- La primera fase es la investigación y el aprendizaje del lenguaje que se va a emplear, OpenCL (*Open Computing Language*) [3] [5] [7]. Un lenguaje de programación de propósito general para GPU administrado por Khronos Group.

- El estudio y documentación de herramientas matemáticas para la secuencia de pulsos de Stejskal – Tanner [8] [9]. [10]

- Análisis de la ecuación, y desglose para simplificar y favorecer la resolución de la misma.

- Fase de desarrollo, en la cual se realizará el desarrollo de todo el código necesario, siguiendo la pauta anteriormente explicada Así como programar de manera modular, para poder desarrollar el código en un futuro.

- Pruebas de funcionamiento, desarrolladas durante toda la fase de desarrollo del código, para comprobar que se estaban obteniendo los resultados esperados; primero haciendo pruebas con 6 tensores para pasar luego a problemas más complejos.

- Utilización de otras tarjetas gráficas superiores, y comprobación del funcionamiento del código, así como verificación de resultados de DTI-RM más completas.

- Comparación de tiempos de ejecución del mismo algoritmo sobre GPU o sobre CPU, programándolo en OpenCL en ambas ocasiones.

## 1.3. MEDIOS

Los medios utilizados durante la fase de desarrollo son los indicados a continuación:

Ordenador Portátil HP Pavilion g6:

- Procesador: Intel® Core™ i5-480M (2.66GHz, 3 MB de caché L3).
- Tarjeta gráfica: *AMD Radeon™ HD 6380G*
- Pantalla: LED HP BrightView de 39,6 cm (15,6") Resolución $1366x768$.
- Sistema Operativo: Ubuntu 14.04 X86_64

Ordenador de sobremesa:

- Tarjeta gráfica 1: *Intel HD Graphics 3000*
- Tarjeta gráfica 2: *AMD Radeon R9 290X*

OpenCL 1.2 API.

GNU Octave, (versión gratuita de Matlab)

Entorno de desarrollo integrado Eclipse (C/C++ Developers)

# 2. ARQUITECTURAS ORIENTADAS AL PROCESAMIENTO GRÁFICO

El desarrollo de las arquitecturas basadas en la computación gráfica ha crecido exponencialmente durante las últimas décadas y resulta curioso que el principal impulsor actual de esta tecnología sea el mundo del videojuego. Sin las tarjetas gráficas, no se podrían haber generado imágenes por ordenador, incluso para cine, y es que se ha pasado de visualizar a duras penas texto monocromo a conducir coches a una gran velocidad sin moverse del sofá. Por supuesto esta evolución ha proporcionado algo más que diversión, sólo es necesario interés y saber sacarle partido.

Para conocer el *software* que influye en sectores científicos, primero sería interesante estudiar el marco histórico para entender este progreso, las motivaciones, la demanda de los usuarios y comprender los conocimientos que se poseían con respecto a las herramientas a las que se tenía acceso.

## 2.1. EVOLUCIÓN DE LAS TARJETAS GRÁFICAS

### 2.1.1.Primera etapa: Resoluciones y colores

En los años ochenta se introdujeron unos de los primeros estándares de tarjetas de visualización de vídeo para los ordenadores personales IBM: la tarjeta MDA (*Monochrome Display Adapter*), en 1981. Solo ofrecía modo texto y se usaba con monitores monocromo, de tonalidad normalmente verde. La resolución total de la pantalla del MDA era de 720x350 píxeles, es decir, 80 caracteres a lo ancho con 9 píxeles de ancho cada uno, y 25 caracteres a lo alto con 14 píxeles de alto cada uno. Así pues, con estas dimensiones se formaban

caracteres de gran tamaño, con los que al menos se podían cambiar el brillo, ponerlos en modo inverso o subrayarlos [11] [12].

Otro de los estándares gráficos, y esta vez en color, de IBM, fue la tarjeta gráfica CGA (*Color Graphics Adapter*). Desde el punto de vista gráfico, incluía tres modos diferentes de vídeo configurables por *software*. El modo que optaba a los 16 colores, correspondía con la resolución 160x100 píxeles; los 4 colores se representaban cuando la resolución era de 320x200 píxeles y 640x200 píxeles en monocromo. En cuanto a modos de texto, se encontraba el equivalente a 320x200 píxeles era de 40x25 caracteres, 25 líneas de 40 caracteres de ancho fijo, con 16 colores y el equivalente a 640x200 píxeles era de 80x25 caracteres, con 16 colores también.

Asimismo, eran necesarios monitores con entradas RGB, en el que el rojo, el verde y el azul correspondían a cada uno de los tres rayos catódicos. Mezclando estos tres colores, se creaban otros tres, el cían, el magenta y el marrón, y junto a esos seis, se formaba el blanco, mezcla de los tres rayos, y el negro, la ausencia de ellos. Además de estos ocho colores, se obtenían otros ocho mediante un bit de intensificación, consiguiendo una versión más brillante de cada color.

Poco a poco aparecieron nuevas tarjetas, siempre orientadas a la visualización 2D, con las que se progresó en la profundidad de color y la mayor resolución, tal y como se muestra en la Tabla 2.1.

| Formato | | Año | M. Texto | M. Gráfico | Colores | Memoria |
|---|---|---|---|---|---|---|
| MDA | Monochrome Display Adapter | 1981 | 80*25 | - | 2 | 4 KB |
| CGA | Color Graphics Adapter | 1981 | 80*25 | 640*200 | 4 | 16 KB |
| HGA | Hercules Graphics Card | 1982 | 80*25 | 720*348 | 2 | 64 KB |
| EGA | Enhanced Graphics Adapter | 1984 | 80*25 | 640*350 | 16 | 256 KB |
| IBM 8514 | 8514/A Display Adapter | 1987 | 80*25 | 1.024*768 | 256 | - |

| | | | | | | |
|---|---|---|---|---|---|---|
| MCGA | Multi-Color Graphics Array | 1987 | 80*25 | 320*200 | 256 | - |
| VGA | Video Graphics Array | 1987 | 80*25 | 640*480 | 256 | 256 KB |
| SVGA | Super Video Graphics Array | 1989 | 80*25 | 800*600 | 256 | 512 KB |
| XGA | Extended Graphics Array | 1990 | 80*25 | 1.024*768 | 65.536 | 2 MB |

*Tabla 2.1 - Evolución de los formatos de video en la primera etapa [11] [12]*

En esta misma etapa se desarrollaron tarjetas gráficas como HGC (*Hercules Graphics Card*), compatibles con las MDA, ya que soportaba un modo de texto de alta resolución, pero también tenían un modo de gráficos monocromo con una resolución de 720x348 píxeles, un nivel de detalle considerable y mucho mayor que la CGA. Una singularidad que tenía era que el rango de direcciones de memoria se diferenciaba de las anteriores, de tal manera que se podía instalar una tarjeta Hércules junto con una de las otras y disponer de soporte multimonitor.

Otra de las apuestas de IBM fue EGA (*Enhanced Graphics Adapter*), con modos propios de 16 colores escogidos entre una paleta de 64 y resoluciones de 320x200, 640x200 y como novedad 640x350.

En 1987, la VGA (*Video Graphics Array*), se convirtió en la nueva solución de alto nivel. La gran diferencia con respecto a las anteriores es la capacidad de colores que podía mostrar variando la intensidad del rayo, dándonos así 256 colores simultáneos. Aunque el modo estándar denominado "resolución VGA" era el más extendido con 640x480 píxeles, dependiendo de los colores utilizados era a su vez más eficaz, en este caso sería con solo 16 colores de los 256 que podía proyectar [11]. No obstante, fueron sustituidos por SVGA (Super VGA), que incluían una mejora en la resolución y en la paleta de colores, además de pequeñas variaciones con respecto a sus precesoras y por XGA (eXtended Graphics Array) que conseguía 256 colores con una resolución de 1.024x768 píxeles.

Como se puede comprobar en la Figura 2.1 los modos gráficos han continuado evolucionando, y han aparecido nuevos formatos con más resolución y mayor profundidad.

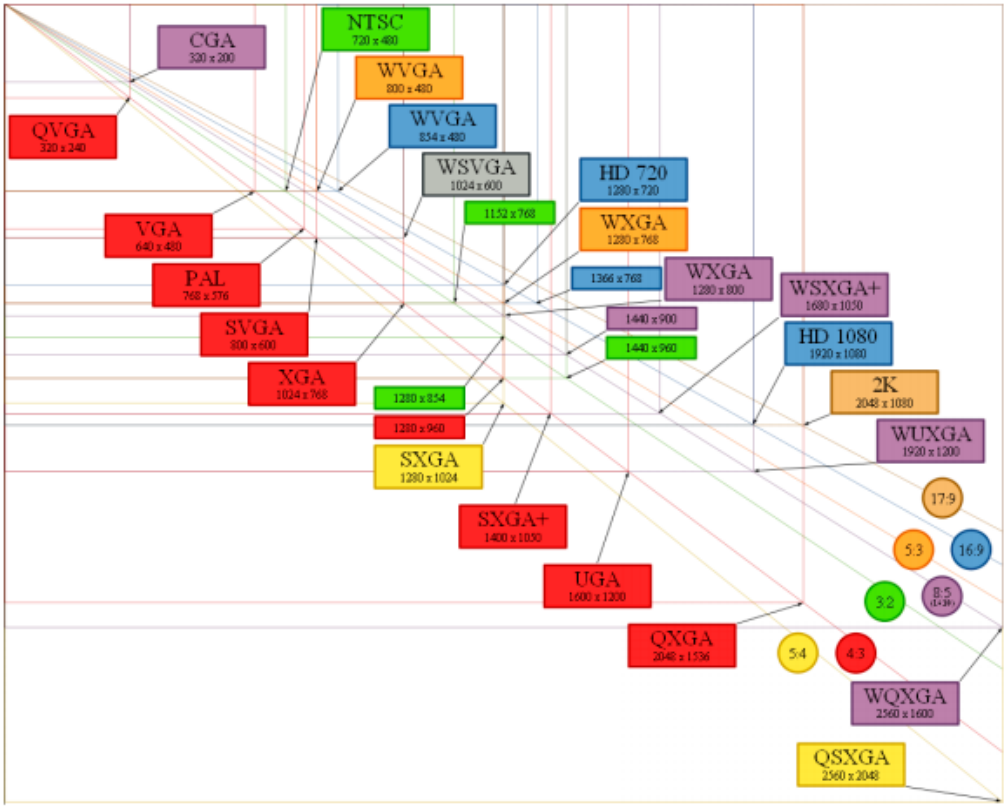


*Figura 2.1 – Formatos de vídeo y sus resoluciones [11] [12]*

### 2.1.2.Segunda etapa: Aceleradoras 3D

Después de esta primera etapa, que se había centrado en las resoluciones y en el número de colores soportados, los juegos de la época seguían exigiendo más funcionalidades. Empezaron a aparecer las primeras tarjetas 2D/3D, aunque, por su precio elevado no suplieron a las SVGA.

En 1997, apareció en el mercado una tarjeta gráfica de la compañía, entonces desconocida, 3DFx Interactive [13], conocida comercialmente por Voodoo. Esta tarjeta era independiente de la tarjeta gráfica que incluía el propio ordenador, además solo se ponía en funcionamiento al ejecutar aplicaciones que hicieran uso específico de la misma. Ventajosa porque era capaz de liberar a la CPU de hacer ciertos cálculos y entre ambas tenían la capacidad de procesar imágenes en tres dimensiones. Sin embargo, un inconveniente notable era que se necesitaba una tarjeta SVGA extra para aplicaciones que no fueran específicamente en 3D. Asimismo, la cantidad de *software* 3D de la época se incrementó, y

ésta compañía dominó durante varios años este mercado con su serie de tarjetas Voodoo, que ofrecían más prestaciones, más resolución, más profundidad de color o más efectos.

La aparición de los productos de Nvidia [4], capaces de fabricar tarjetas con funciones 2D y 3D, hizo que 3DFx empezara a estar lejos de competir con las prestaciones y la potencia de este rival.

Por el momento, las mejoras se habían ceñido en base a dos ideas, un incremento de las resoluciones y de los colores soportados por estas, y una implementación de funciones 3D. Pese a todas las utilidades, aún no eran programables, lo que dio pie a la tercera etapa, en la que se desató un gran enfrentamiento entre Nvidia y ATI.

### 2.1.3.Tercera etapa: Programables

El punto de inflexión de Nvidia fue la aparición, en 1999, de GeForce 256, la primera tarjeta aceleradora capaz de procesar por *hardware* la transformación de los vértices y su iluminación, liberando a la CPU de una gran carga de proceso y mejorando así las posibilidades de desarrollar aplicaciones más interesantes desde el punto de vista visual. La GeForce 256 marcó el comienzo de una progresión natural hacia dispositivos capaces de implementar cada vez más etapas directamente en el procesador gráfico.

A finales del año 2000, 3DFx Interactive se declaró en bancarrota y Nvidia decidió comprar a su competidor, de esta manera se erige como líder en el mercado de las tarjetas gráficas sólo con la competencia de ATI, hoy en día propiedad de AMD [3]. A partir de aquí el conflicto se desató, ambos presentaban sus proyectos al mercado intentando perjudicar a su contrincante demostrando sus progresos durante años.

Posteriormente, en marzo de 2001, Nvidia lanza al mercado la primera tarjeta programable, la GeForce 3. Capaz de programar el procesado de los vértices y el procesado de fragmentos, permitiéndose así un incremento del realismo en las aplicaciones, gracias a la creación de numerosos efectos de procesado. Así pues, por primera vez, los desarrolladores tuvieron un cierto control sobre los cálculos que se podían desarrollar en las GPU, y por consiguiente, se cambió por completo el resultado de la visualización de los modelos en 3D.

Desde el punto de vista arquitectónico, las primeras generaciones de GPU tenían una cantidad de núcleos bastante reducida, pero rápidamente se incrementó. Este aumento hizo que, sobre el 2003, hubiera un salto importante de la capacidad de cálculo en coma flotante de las GPU respecto a las CPU, esto ha hecho que este tipo de dispositivos se estén volviendo muy populares para el cómputo de algoritmos de propósito general y no solamente para la generación de gráficos.

La capacidad de procesar los datos con mayor rapidez y de tener más realismo en las imágenes, hizo que la memoria de las tarjetas gráficas se viera influida, así en 2006, ATI presentó la tarjeta Radeon 1950X que dio ese salto a memorias GDDR4, superando a su rival, Nvidia.

Nvidia, innovó presentando la primera unidad de procesamiento gráfico del mercado en contar con una arquitectura unificada, la GeForce 8800. Esto que permitía utilizar cualquier *shader* para cualquier parte del cálculo de la escena, asignaba los recursos de procesamiento de forma dinámica a las operaciones de geometría, física o sombreado de vértices y píxeles, proporcionando más rendimiento que las GPU de generaciones anteriores.

En 2009, el nuevo estándar de Microsoft [14], DirectX 11 [15], aprovechó la eficacia de los procesadores de varios núcleos y brindó la compatibilidad con técnicas de sombreado y texturas sofisticadas. La primera que se benefició de esta interfaz para programación de aplicaciones fue ATI, que presentó Radeon HD 5870.

Actualmente, para clasificar y/o elegir las tarjetas gráficas hay que tener en cuenta la cantidad de especificaciones que tienen, compatibilidad con API, velocidad de reloj, ancho de banda de memoria, tipo de memoria, unidades de procesamiento, etc. En el presente Nvidia reina con su familia GeForce GTX, y AMD con las Gráficas Radeon serie R9.

Los dispositivos gráficos han evolucionado. Desde las primeras tarjetas que solo eran capaces de ofrecer modo texto, hasta los últimos avances, TressFX Hair [16] en AMD y Hairworks [17] en Nvidia; centenares de estructuras se calculan en tiempo real y cambian de acuerdo al movimiento de los personajes de videojuegos o de las condiciones climatológicas. Desde dispositivos especializados en su tarea para la generación de gráficos, hasta dispositivos computacionales masivamente paralelos aptos para la computación de

propósito general, como por ejemplo la implementación de la secuencia de pulsos de Stejskal – Tanner [8] que fundamenta este proyecto [3].

## 2.2. ARQUITECTURAS HARDWARE: CPU Y GPU

Mientras que las GPU, con una lógica bastante simple, están pensadas para ejecutar una misma operación sobre un volumen considerable de datos, combina el paralelismo a nivel de datos con el paralelismo masivo, el diseño de una CPU está optimizado para la ejecución de código secuencial [11] [18] [19] [20] [21].

Por definición, la CPU controla el funcionamiento del computador, y lleva a cabo sus funciones de procesamiento de datos. Permite que funcionen todos los componentes internos, así como el sistema operativo y las aplicaciones. Ésta unidad varía de tamaño dependiendo de su finalidad, e incluso de número, ya que en las últimas décadas ha habido un uso creciente de varios procesadores en un solo sistema, incrementando el número de instrucciones simultáneas que el sistema puede realizar y la potencia de cálculo.

Una GPU es un procesador integrado en una tarjeta gráfica dedicado al procesamiento gráfico. Se encarga de las operaciones en coma flotante y de liberar la carga del procesador. Nvidia lo acuñó erróneamente; lo usó para diferenciar la primera tarjeta que permitía al programador implementar sus propios algoritmos. Este componente se encarga de interpretar la información procedente del microprocesador para que se muestre en un monitor, o en cualquier otro dispositivo de salida que muestre información gráfica. En ocasiones, se necesitan gran cantidad de cálculos como en los ya mencionados videojuegos, que cada vez son más complejos y más reales tanto en 2D como en 3D.

Con la aparición de las tarjetas aceleradoras, surgieron potentes *chips* que no tenían nada que envidiar ni en potencia de cálculo, ni en complejidad, a los procesadores de Intel o AMD. Se usaron arquitecturas paralelas capaces de ejecutar la misma operación sobre un conjunto de datos de manera simultánea y, con los *pipelines* programables de la siguiente generación de tarjetas gráficas, los *chips* ganaron aún más complejidad y libertad de ejecución en las instrucciones. Se aproximaron mucho a los procesadores de propósito general, no obstante, se mantuvieron en cuanto a la estructura paralela.

La diferencia más general anida en la cantidad de núcleos que tiene cada una. Una CPU contiene varios núcleos preparados para el procesado en serie, mientras que una GPU consta de cientos, incluso miles de núcleos más pequeños diseñados para manejar múltiples tareas simultáneamente [4]. Sin embargo, hay que recalcar que, aun teniendo más núcleos, estos están preparados para ir a menor velocidad, y en el caso de realizar indebidamente la programación del algoritmo, se desperdiciarían las capacidades además de ejecutarse en más tiempo.

Una de las características más relevante de las arquitecturas GPU es la técnica que emplea para conseguir y explotar el paralelismo, mientras que los ordenadores convencionales, ejecutan las instrucciones de manera secuencial, el orden viene dado por la colocación al almacenarlas en memoria.

Como se puede ver en la Figura 2.2, en la distribución interna de la GPU se simplifica la lógica de control del procesador con respecto a la de la CPU, así como se minimiza la caché interna del mismo. Esto hace que mientras unas ALU están a la espera para acceder a la DRAM (*Dynamic Random Access Memory*), el resto pueda ir ejecutando su cometido resultando la espera menor. La memoria compartida que contiene cada multiprocesador en la GPU es muy veloz, y la memoria DRAM de la GPU es más rápida que la de la CPU. En consecuencia, mucha más área del chip se dedica al procesamiento de datos en coma flotante.

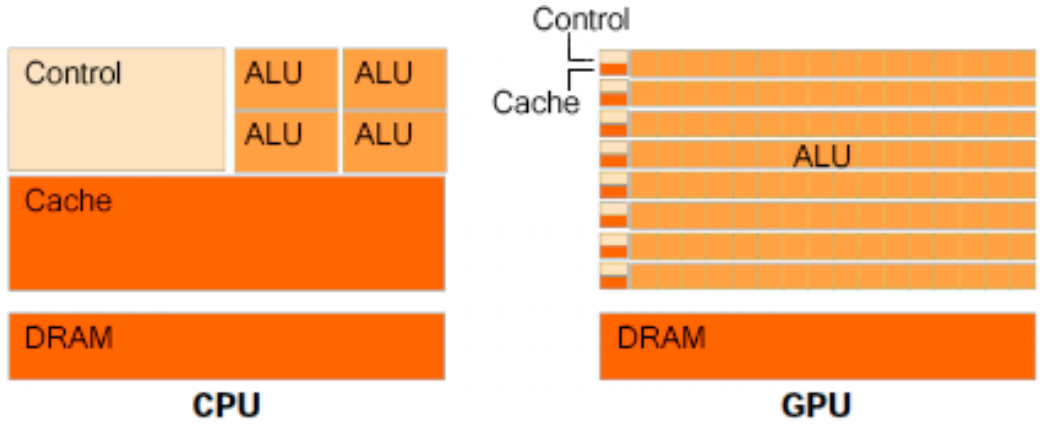


*Figura 2.2 – Distribución de transistores CPU y GPU [22]*

Otro de los aspectos importantes, y diferenciadores, es la capacidad del bus de datos para transmitir información, ya sea entre la CPU y la memoria de la placa base, o entre la

CPU y la GPU. Genéricamente se les denomina buses internos, los propios de la CPU que permiten comunicarse con los componentes internos, y buses externos, que permiten la comunicación de la CPU con el resto de elementos. El ancho del bus, determinado por el microprocesador, define el número de bits que podemos transmitir al mismo tiempo, y la frecuencia indica el número de paquetes de datos que pueden ser enviados o recibidos en un segundo. Estas propiedades han ido un paso por detrás de la capacidad de computación de los microprocesadores [11] [12]. La norma AGP (*Accelerated Graphics Port*) apareció por primera vez en 1996; un bus dedicado exclusivamente a la GPU, con diferentes modos de funcionamiento aprovechando los ciclos de reloj mediante un multiplicador, que perduró hasta la aparición de PCI-Express. Estructurado con varias conexiones bidireccionales en serie y tal y como se muestra en la Tabla 2.2 hecha a partir de la Ecuación 2.1, PCI-Express 1.0 transporta 250 Mb/s en cada dirección. PCI-Express 2.0 dobla esa tasa, y PCI-Express 3.0 la dobla de nuevo. En la Tabla 2.2 se adjuntan los datos de USB para cotejarlos con el resto, ya que estos buses resultan más habituales.

$$Capacidad\ de\ transferencia\ (MB/s) = \frac{Ancho\ el\ bus\ (bits) * Frecuencia\ (MHz)}{8\ (bits/byte)}$$

*Ecuación 2.1 [19]*

Hoy en día las tarjetas gráficas llevan incorporado multitud de microchips que permiten liberar al propio procesador de carga de trabajo. RAMDAC (Random Access Memory Digital to Analog Converter) es el elemento encargado del refresco de la pantalla, número de veces que puede dibujarse la pantalla completa en cada segundo, también realiza la conversión entre las señales digitales que representan las imágenes dentro de PC y la información gráfica analógica que usa el monitor1 [19].

[1] Con conectores VGA.

| BUS | ANCHO DEL BUS (bits) | FRECUENCIA O VELOCIDAD DEL BUS (MHz) | CAPACIDAD DE TRANSFERENCIA DEL BUS (MB/s) |
|---|---|---|---|
| ISA 8 | 8 | 4,77 | 4,77 |
| ISA 16 | 16 | 8,33 | 16,66 |
| EISA | 8-16-32 | 8,33 | 8,33-16,66-33,32 |
| VESA Local Bus | 32-32 | 33-40 | 132-160 |
| PCI 32 | 32-32 | 33-66 | 132-264 |
| PCI 64 | 64-64-64 | 33-66-100 | 264-528-800 |
| AGP | 32 | 66 | 264 |
| AGPx2 | 32 | 66x2 | 528 |
| AGPx4 | 32 | 66x4 | 1056 |
| AGPx8 | 32 | 66x8 | 2112 |
| USB 1.0 | 1 | 12 | 1,5 |
| USB 2.0 | 1 | 480 | 60 |
| USB 3.0 | 1 | 5000 | 625 |
| PCI Express 1.0 (x2) | 2 | 2000 | 500 |
| PCI Express 1.0 (x4) | 4 | 2000 | 1000 (1GB/s) |
| PCI Express 1.0 (x8) | 8 | 2000 | 2000 (2GB/s) |
| PCI Express 1.0 (x16) | 16 | 2000 | 4000 (4GB/s) |
| PCI Express 1.0 (x32) | 32 | 2000 | 8000 (8GB/s) |
| PCI Express 2.0 (x4) | 4 | 4000 | 2000 (2GB/s) |
| PCI Express 2.0 (x8) | 8 | 4000 | 4000 (4GB/s) |
| PCI Express 2.0 (x16) | 16 | 4000 | 8000 (8GB/s) |
| PCI Express 2.0 (x32) | 32 | 4000 | 16000 (16GB/s) |
| PCI Express 3.0 (x16) | 16 | 8000 | 16000 (16GB/s) |
| PCI Express 3.0 (x32) | 32 | 8000 | 32000 (32GB/s) |

*Tabla 2.2 – Formatos de bus entre la GPU y la placa base [19]*

Otro de los elementos básicos es la memoria. En función de la memoria que se tenga, se representaran imágenes con mayor o menor resolución y con mayor o menor cantidad de colores. Como se observa en la Tabla 2.3, según fueron evolucionando las tarjetas gráficas, era deseable procesar con más rapidez los datos, que cada vez eran más, así los módulos de memoria orientados a las tarjetas gráficas se optimizaron para lograr altas

frecuencias de reloj. Con el tiempo, sus características se han centrado en el ancho de banda, aunque la evolución no ha sido tan sincronizada con las aplicaciones de GPGPU y los procesados gráficos como se esperaba, y actualmente representa un cuello de botella.

| TIPO | AÑO | FRECUENCIA DE LA MEMORIA (MHz) | ANCHO DE BANDA (GB/s) |
|---|---|---|---|
| DDR | 1999 | 166-950 | 1,2-30,4 |
| DDR2 | 2003 | 533-1000 | 8,5-16 |
| GDDR3 | 2003 | 700-1800 | 5,6-54,4 |
| GDDR4 | 2005 | 1600-2400 | 64-156,6 |
| GDDR5 | 2008 | 3000-3800 | 130-230 |

*Tabla 2.3 – Evolución de la memoria en tarjetas gráficas [19]*

La GPU tiene propiedades que con el tiempo han ido progresando notablemente, entre ellas el avance dentro de cada arquitectura especificando los bits de signo, los de mantisa y los bits de exponente empleados para cada representación, creando así más realismo debido al mayor nivel de detalle.

## 2.3. ARQUITECTURA DE UNA GPU

Para entender el funcionamiento de una GPU, partimos de que no hay una arquitectura global para las GPU, si no que centrándonos en las más activas y fuertes, tanto Nvidia como AMD tienen una propia. Además hay que analizar los fundamentos de la computación gráfica [21] [23] [24].

### 2.3.1.Evolución de aplicaciones gráficas

En la década de los ochenta, se trabajaba a muy bajo nivel, ya que no había librerías. La forma más extendida de elaborar una aplicación gráfica, era mediante un programa en C con llamadas a servicios de interrupción ligados a las operaciones gráficas. Algunos de estos servicios se encontraban en la BIOS de la placa base, otros eran proporcionados por el *driver* o controlador del dispositivo de la tarjeta gráfica.

En los noventa nació DirectX [15] y OpenGL [25], uno por parte de Microsoft destinado solo a Windows y a Xbox, y el otro como iniciativa abierta y multiplataforma. Se trata de dos API (*Application Program Interface*) o interfaces para la programación de aplicaciones que facilitan el trabajo a los programadores de gráficos 3D. Conformaba un conjunto de librerías, proporcionando una serie de operadores típicos para las gráficas de la época. Gráficas en el caso de OpenGL; DirectX se ocupa también de otras funciones, la parte estrictamente de gráficos 3D se denomina Direct3D.

Aunando las API y la evolución a la que estaba sometido el *hardware*, surgen aplicaciones importantes para el avance y desarrollo de la industria. La empresa id Software lanzó los videojuegos Doom y Quake, las siguientes versiones de este último incorporaban sombreadores de píxeles y su diseño demostró que se podría ejecutar sobre un *hardware* gráfico con pocos recursos aritméticos, siempre que se pudieran aplicar varias pasadas de renderizado, usando mucho ancho de banda y poca ALU. La comparativa de este avance se puede ver en la Figura 2.3, además del aumento de resolución, se percibe un suavizado en la segunda imagen que corresponde con la tarjeta de 3DFx, que elimina bastante ruido. En ese momento se incorporaron dos procesadores programables dentro de la GPU, en las primeras etapas uno adherido al procesamiento de vértices, y en las últimas otro encargado del procesamiento de píxeles.

En 2002, la versión 9.0 de DirectX se extendió, incorporando a estos procesadores instrucciones de salto condicional en el procesador de vértices y coordenadas de texturas y computación de punto flotante en el procesador de píxeles.

Nvidia y Microsoft estaban interesadas en extender la funcionalidad de los sombreadores. Por otro lado, 3DLabs y ATI comenzaron a desarrollar GLSL (*OpenGL Shading Language*) que finalizó en 2004, mismo año en el que se incorporó la versión 2.0 de OpenGL, que permitía programar los vértices y los píxeles.


*Figura 2.3 – Comparativa del videojuego Quake*

A partir de este momento, estas dos API se han establecido como librerías gráficas viables. OpenGL es una API estándar de uso libre, ya que una vez obtenido el producto se puede modificar y adecuar el código al propósito propio, que es público; mientras que DirectX no, lo que hace que solo esté implementada en la familia de sistemas operativos de Microsoft. OpenGL tiene implementaciones en muchas plataformas.

Dado el crecimiento del *hardware*, y al abanico de posibilidades, en la actualidad existe cierta competencia. El motivo es la computación de propósito general, con arquitecturas para el cálculo paralelo de Nvidia con CUDA, ATI con Stream, o el antagonista libre OpenCL.

### 2.3.2.*Pipeline* – Funcionamiento básico de GPU

*Pipeline* se refiere al conjunto de alteraciones y transformaciones de imagen que se realizan en el procesamiento de gráficos. De una manera muy simplificada, se puede decir que se compone de 4 etapas de procesado. Como se puede ver en la Figura 2.4 y Figura 2.5 cada una de las cuales toma como entrada la salida de la anterior.

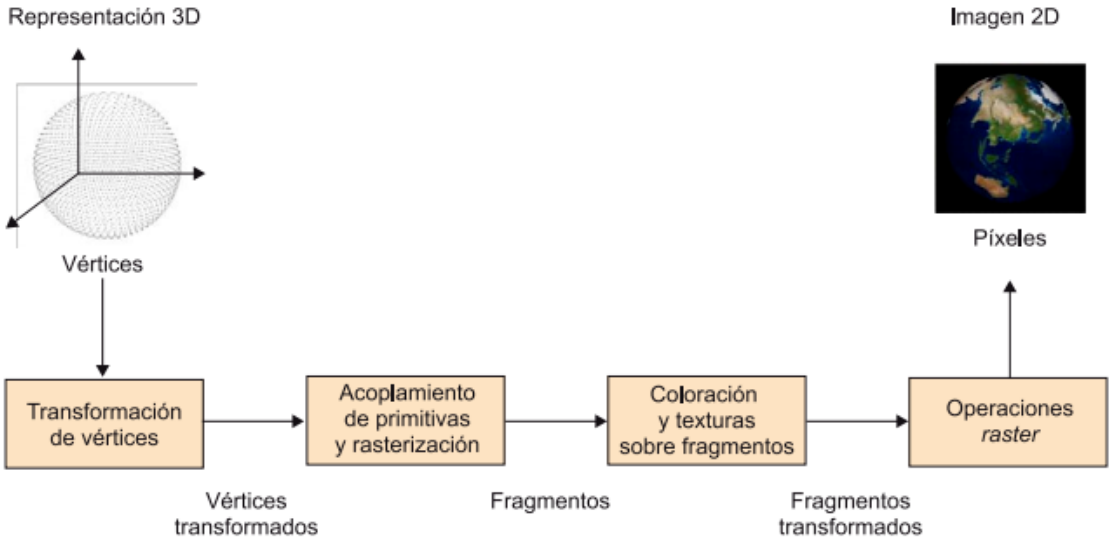


*Figura 2.4 – Pipeline simplificado de un procesador gráfico*

La primera de las etapas es la de transformación de vértices, que consiste en la ejecución de una secuencia de operaciones matemáticas sobre los vértices de entrada. Estas operaciones generan coordenadas para poder aplicar texturas o colorear los vértices, por tanto la salida de esta fase así como la entrada de la siguiente, es un conjunto de vértices coloreados. En la siguiente etapa, acoplamiento de primitivas y rasterización, estos vértices se agrupan en primitivas geométricas, y se obtiene una secuencia de triángulos, líneas y puntos, reduciendo el número de triángulos necesarios teniendo en cuenta cuáles están tapados o fuera de escena dado el punto de vista del observador (proceso de recorte o *clipping*) [26] [27]. Se rasterizan los triángulos en 3D para lograr una imagen de píxeles en 2D, además se obtiene información relativa al color, al brillo o textura de un conjunto de píxeles o fragmentos, como norma a partir de 3 píxeles.

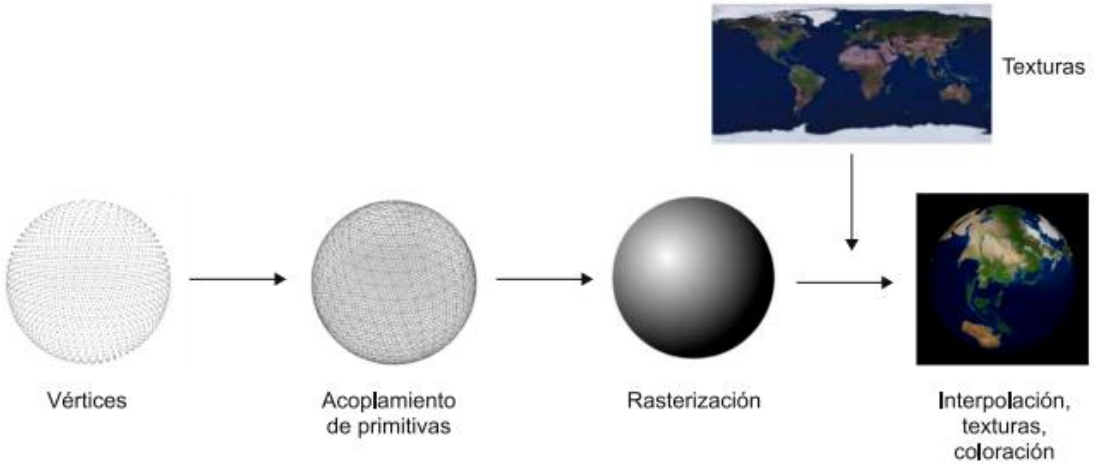


*Figura 2.5 – Resumen de las funcionalidades del pipeline gráfico*

En la tercera etapa, aplicación de texturas y coloreado, se obtienen fragmentos coloreados y con texturas a partir de operaciones de interpolación. Las texturas son mapas de bits de imágenes reales, que sirven para dar un gran realismo a las imágenes sin necesidad de complicar el modelo de vértices. También confluyen técnicas de suavizado de texturas y de *anti-aliasing* (suavizado de bordes de los objetos), importante para tener imágenes realistas. En la última etapa se realizan operaciones llamadas *raster*, testea los fragmentos y determina los valores que tomarán finalmente los píxeles.

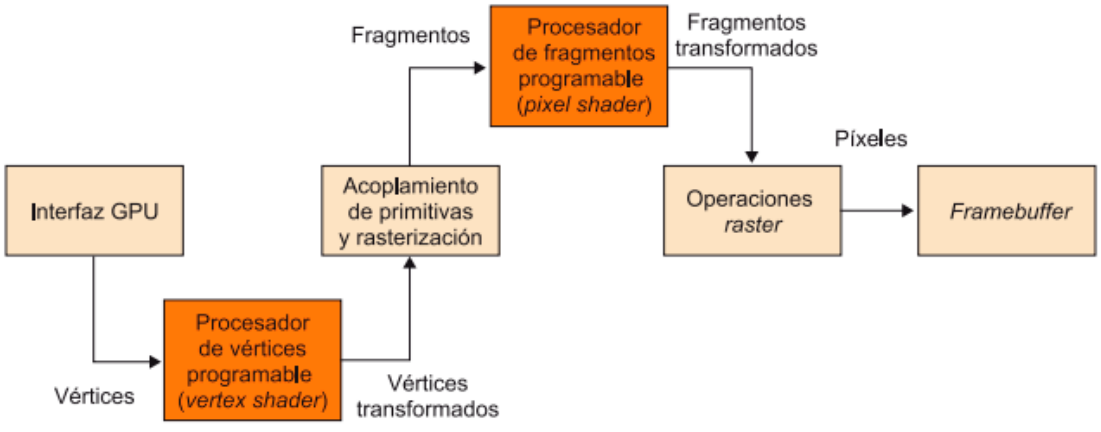


*Figura 2.6 – Pipeline gráfico programable*

Habitualmente el *pipeline* contiene dos etapas adicionales intermedias (Figura 2.6), *Vertex Shader* o procesador de vértices y *Pixel Shader* o procesador de fragmentos, ambas programables [23]. Simplificando, por un lado el funcionamiento de un procesador de vértices programable consiste en la manipulación los vértices de figuras en 3D. Modifica propiedades del mismo para que repercutan en la geometría del objeto al que pertenece. Son capaces de recrear movimientos de olas en tiempo real o expresiones corporales. Por el otro, los procesadores de fragmentos programables, rasterizan la escena, deciden de qué color se pinta cada píxel considerando la textura que tuviera el objeto real. Permite realizar cálculos relacionados con la iluminación, teniendo la posibilidad de iluminar cada píxel por separado. Si se extrapola de nuevo, definiría la iluminación de una escena.

Hay una diferencia entre ambos, y es que los *vertex shaders* no necesitan un soporte de *hardware* compatible, mientras que los *pixel shaders* requieren una tarjeta con capacidad para manipularlos. En la Tabla 2.4 se muestra una pequeña evolución de las diferentes versiones de estos últimos, los *pixel shaders*.

| Versión pixel shader | 2.0 | 2.0a | 2.0b | 3.0 | 4.0 |
|---|---|---|---|---|---|
| **Número de texturas dependientes** | 8 | Ilimitado | 8 | Ilimitado | Ilimitado |
| **Instrucciones ejecutadas** | 32+64 | 512 | 512 | 65.536 | Ilimitado |
| **Indirecciones de textura** | 4 | Ilimitado | 4 | Ilimitado | Ilimitado |
| **Registros temporales** | 12 | 22 | 32 | 32 | 4.096 |
| **Registros constantes** | 32 | 32 | 32 | 224 | 16x4.096 |
| **Control dinámico de ejecución** | No | No | No | 24 | Si |
| **Operaciones lógicas** | No | No | No | No | Si |
| **Enteros nativos** | No | No | No | No | Si |

*Tabla 2.4 – Características de las diferentes versiones de los pixel shader [23]*

### 2.3.3.GPGPU

El uso de tarjetas gráficas para la resolución de problemas de propósito general se le denomina GPGPU. Está motivada por la optimización de las características de intrínsecas

de la GPU para cualquier ámbito. Por lo general, el procesamiento básico que sigue una GPU es inicialmente el tratamiento de vértices que le envía la CPU [23] [24]. A partir de este procedimiento, se realiza un proceso indicado en la Figura 2.7, por el cual se define las caras de las figuras que se van a visualizar, se les aplican distintas texturas, y los colores que sean oportunos, y por último se testea en Raster Operator Unit (ROP) por si ha habido algún error.

Esta iniciativa, que empezó con los primeros *pipelines* programables, utiliza los *shaders* para efectuar cálculos en operaciones entre números en coma flotante. Estos elementos estaban diseñados para programadores de videojuegos. Tenían flujos de entrada y de salida fijos, con un límite de longitud y un número de variables finito. En las últimas especificaciones de los lenguajes adecuados que se utilizan para este fin y que se verán a continuación, se han solventado algunos de estos problemas facilitando su uso general.

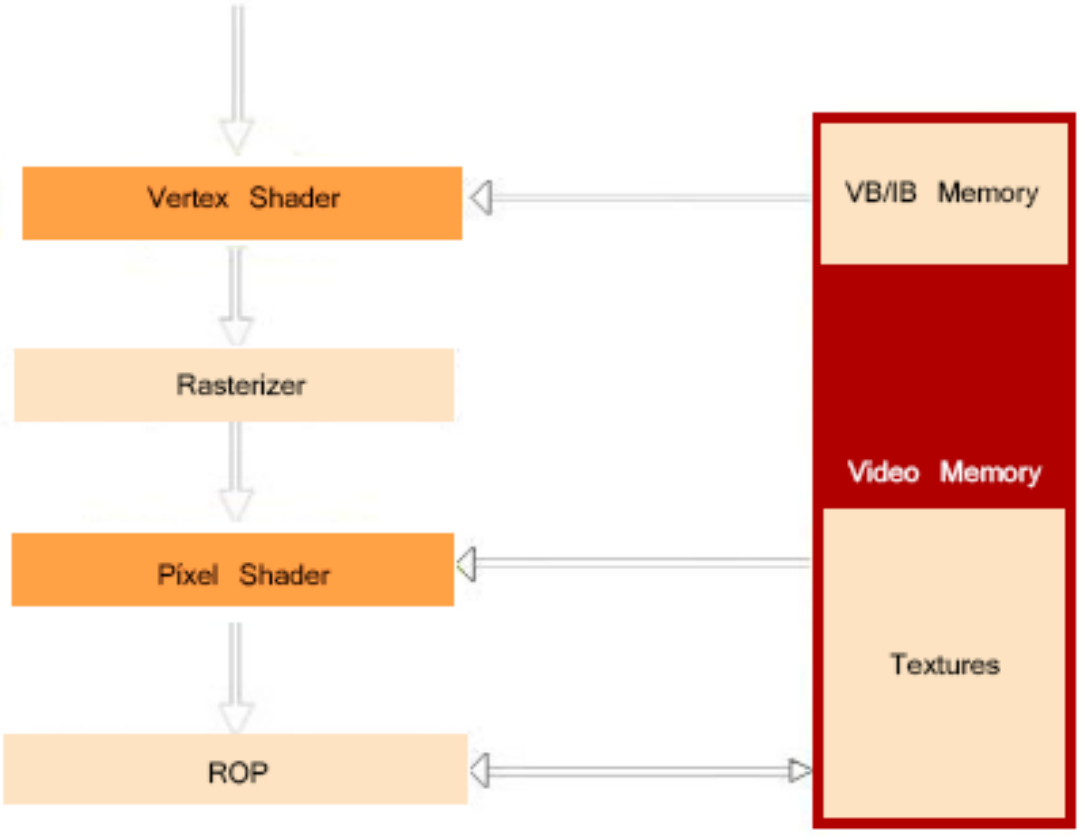


*Figura 2.7 – Proceso de renderizado en una GPU*

Las estructuras de datos que entran en el proceso de renderizado son declaradas como conjuntos de datos organizados con coordenadas, de una, dos o tres dimensiones, y se accede a cada uno de sus elementos mediante direcciones de las dimensiones correspondientes. Este proceso acaba siendo tan simple como operaciones entre estructuras en coma flotante que, cambiando la finalidad, de visualización 3D a implementación de problemas del ámbito científico, se pueden implementar algoritmos reduciendo tiempos, una tarea muy útil y productiva.

Se deben estudiar las características arquitectónicas y de funcionamiento de cada algoritmo, ya que no son válidos todos para su implementación en GPU. De manera genérica, los que mejor pueden aprovechar este tipo de computación son los que trabajan con vectores de datos grandes y tienen un gran paralelismo.

Si se quiere optar por una programación eficiente, se debe optimizar el paralelismo de datos utilizando el modelo SIMD (*Single Instruction Multiple Data*) [28] de un único flujo de instrucciones. Además se puede solapar el procesamiento con comunicaciones a través del bus gráfico, que contempla el transporte bidireccional. Igualmente, al trabajar con volúmenes grandes de datos de forma continua, se muestra una ineficiencia. Por tanto es más efectivo realizar un solo pase de renderizado implicando pocos vectores de un gran volumen de datos, que descomponer el problema en varios pases sobre reducidos volúmenes de datos.

Para simplificar el trabajo de transformar los algoritmos diseñados para CPU, se han desarrollado API, ya sean de propiedad como CUDA o abiertos como OpenCL que proporcionan herramientas facilitando este cometido.

### 2.3.4.Características de la tarjeta gráfica

La serie de tarjetas AMD Radeon R9 es una familia de tarjetas gráficas muy potentes, en especial la AMD R9 290X, que es la empleada para este estudio [3].

#### *Arquitectura Graphics Core Next 2.0*

Fue el primer producto de AMD basado en la arquitectura gráfica *Graphics Core Next* 2.0.

La arquitectura enunciada partía de su antecesora VLIW4 (*Very long Instruction Word 4*). El procesador que incluían estas tarjetas estaba compuesto por miles de bloques pequeños, cada uno con cuatro unidades de procesado aritmético. Eran capaces de realizar cuatro operaciones en paralelo, ya que la creación de imágenes tridimensionales necesita realizar muchas operaciones divisibles en grupos de cuatro. Cuando se quiere emplear la gran potencia de cálculo de las tarjetas gráficas para otro tipo de utilidad, en esa arquitectura se quedaba sin utilizar entre un 25% y un 50%.

De este modo AMD rediseñó la arquitectura para que no dependiera tanto de la especialización en las instrucciones dedicadas a la creación de gráficos. Se realizó el paso de 40 namómetros a los 28 actuales, que permite incluir el doble de transistores en la misma área que con la anterior generación. Se modificó la frecuencia de funcionamiento y el control de consumo que se redujo. Igualmente soporta el estándar PCI –Express 3.0.

La unidad de cómputo de la arquitectura que utiliza la tarjeta gráfica, está formada por múltiples Compute Units y son capaces de operar de forma completamente independiente unos de otros, pudiendo ejecutar cargas de trabajo de enteros y punto flotante, con salida condicional hacia el controlador de memoria.

### *Serie R9*

Las tarjetas gráficas de la Serie R9 soportan DirectX 11.2. Además son las primeras GPU con la memoria de alto ancho de banda (HBM) en el chip, cuenta con bajo consumo energético y mayor rapidez de transmisión.

### *Radeon R9 290X*

Esta tarjeta incluye 4 Gb de memoria GDDR5 que funcionan a 5 GHz y con un bus de 512 bits tienen un ancho de banda de memoria de 320 GB/s. Su núcleo funciona a 1 GHZ teniendo una capacidad de cómputo global de 5.6 TFlops. Además cuenta con tecnología "TrueAudio" y con la posibilidad de CrossFireX automático.

Una de las tecnologías que incluye es AMD ZeroCore Power, apaga prácticamente la tarjeta gráfica cuando no se está utilizando y de esta manera ahorrar energía y reducir el calor, ya que cuando está en funcionamiento tiene temperaturas muy elevadas.

# 3. DIFFUSION MAGNETIC RESONANCE IMAGING (DMRI)

Las imágenes por resonancia magnética de difusión estudian el proceso físico mediante el cual las partículas de materia se mueven de manera aleatoria de un lugar a otro dentro de un medio debido a movimientos térmicos moleculares, denominados movimientos brownianos. De esta manera, se puede estudiar la conectividad de las fibras nerviosas del cerebro en la sustancia blanca.

Stejskal – Tanner [8] [10] fueron los primeros en aplicar la propiedad de difusión a las secuencias de RM en 1965. Más de 20 años después Le Bihan desarrolló la primera DMRI cerebral y Warach fue el primero en aplicar esta técnica para el estudio del infarto cerebral en 1992.

Comparándola a otras técnicas MRI (Magnetic Resonance Imaging) ha tenido una rápida evolución, ya que la materia blanca, que está compuesta por fibras mielinizadas, aparece como algo homogéneo en las imágenes MR, y gracias a DMRI, se puede representar con gran detalle la compleja organización estructural de la sustancia blanca del cerebro.

## 3.1. PRINCIPIO FÍSICO

Albert Einstein explicaba en el artículo titulado *"Sobre el movimiento requerido por la teoría cinética molecular del calor de pequeñas partículas suspendidas en un líquido*

*estacionario"*, que los granos de polen que observó en el microscopio Brown[2] se movían de manera aleatoria por que las partículas imperceptibles que los forman colisionan, como si fueran bolas de billar, con las partículas que forman el agua. De esta manera, la existencia de esas partículas microscópicas, los átomos, quedaría probada [2] [8] [29] [30] [31] [32] [33].

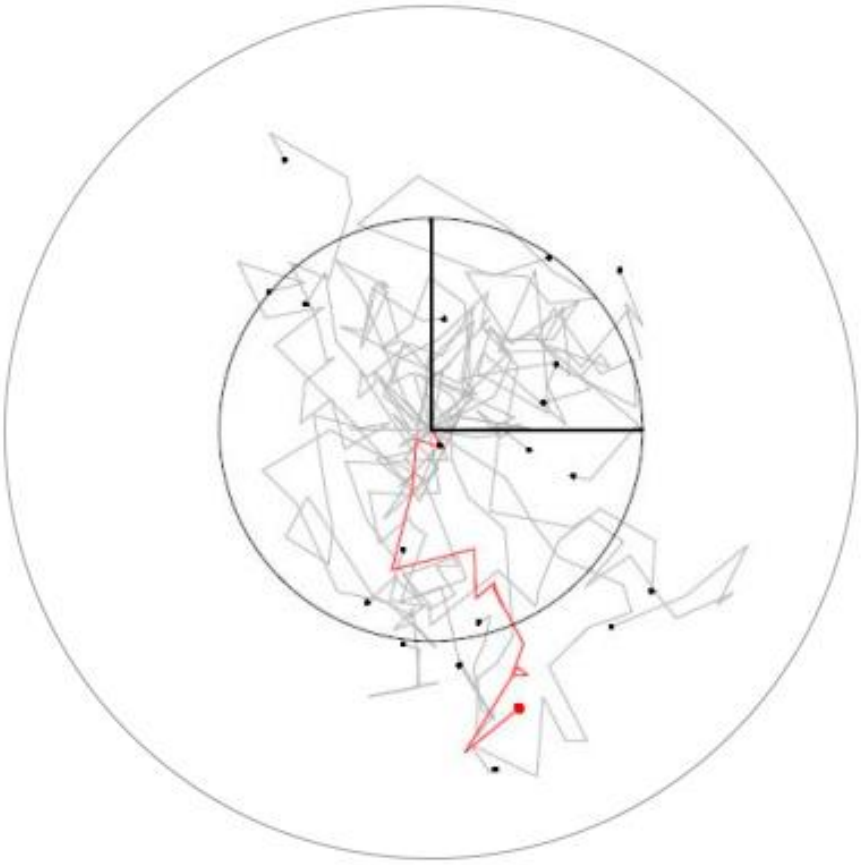

*Figura 3.1 – Movimiento Browniano Isotrópico para varias moléculas de agua*

2 Robert Brown: Biólogo y botánico escocés, había descubierto que los granos de polen se movían de manera impredecible al flotar en un líquido, a pesar de que éste estuviera en total reposo. Aunque fue Einstein quien explicó este movimiento, recibió el nombre de "movimiento browniano" en honor a su descubridor.

En la Figura 3.1 se puede observar que cada línea gris es el camino que siguen las moléculas, y cada punto marca la situación final después de una cantidad de tiempo transcurrido.

El efecto de difusión libre implica la dispersión de las moléculas con el paso del tiempo, como una gota de tinta en un vaso de agua, alejándose del punto inicial en forma simétrica, disminuyendo su concentración del punto de partida y alcanzando mayores distancias. Este espacio, que se puede expresar numéricamente, se denomina radio promedio de la distribución.

La primera ley de Fick (Ecuación 3.1) describe este proceso, en el que la densidad de la corriente de las partículas es proporcional al gradiente de concentración. Einstein demostró, a partir de esa ley que el coeficiente de difusión está relacionado con ese radio promedio de la distribución (Ecuación 3.2).

$$J = \mathrm{D}\nabla \mathrm{c} \qquad \textit{Ecuación 3.1}$$

$$\Delta r^2 = 2nD\Delta t \qquad \textit{Ecuación 3.2}$$

Donde $J$ es el flujo producido por la difusión, $D$ es el coeficiente de difusión valor que representa la facilidad con que cada soluto en particular se mueve en un disolvente determinado, y $\nabla c$ es el gradiente de concentración.

Siendo precisos, el coeficiente de difusión, es proporcional al cuadrado medio de la distancia recorrida por las partículas ($\Delta r^2$) entre el tiempo transcurrido ($\Delta t$) y el número de dimensiones del medio en el que se produce la difusión ($n$) (Ecuación 3.3)

$$D = \frac{\Delta r^2}{2n\Delta t} \qquad \textit{Ecuación 3.3}$$

El uso específico de estas ecuaciones es conocer la forma en la que se desplazan las moléculas en una dirección y distancia determinadas, así cuando las moléculas no tienen restricciones, la dirección del movimiento es aleatoria y los desplazamientos de las moléculas están descritos por una distribución gaussiana, se le denomina difusión isótropa. Por el contrario, si la dirección de las moléculas depende del medio se designa difusión anisótropa.

En consecuencia, en este caso, la difusión anisótropa es dependiente del tejido neuronal, formado por los axones, encargados de transmitir los impulsos nerviosos entre las células. Ordenados y rodeados por las células gliales dificultan así el movimiento de las moléculas del agua, sobretodo en la dirección perpendicular a la dirección de los axones que en la paralela a éstos.

## 3.2. DWI

Variante de la resonancia magnética, aporta información adicional a las imágenes de resonancia magnética habituales, detectando el movimiento de las moléculas del agua en los tejidos. La materia gris, formada por dendritas y cuerpos neuronales que no tienen mielina, no posee una dirección determinada por tanto la movilidad de las moléculas es independiente al tejido. Sin embargo la materia blanca, compuesta por fibras nerviosas cubiertas de mielina, condiciona esa movilidad limitando el movimiento de las moléculas condicionado por los axones orientados a lo largo de las fibras. Ambas sustancias se diferencian por la presencia o no de mielina, materia que rodea la fibra nerviosa y posibilita la transmisión veloz de los impulsos nerviosos entre las diferentes partes del cuerpo [31] [34] [35].

Al efectuar una DMRI, se obtiene como resultado un lote de imágenes sensibilizadas a la difusión en una determinada dirección, que se llaman DWI (*Diffusion Weighted Imaging*). Para analizar esta cantidad de imágenes se emplea la secuencia de Stejskal – Tanner (Figura 3.2). Ésta secuencia aplica dos pulsos de gradiente adicionales, iguales en magnitud y opuestos en dirección.

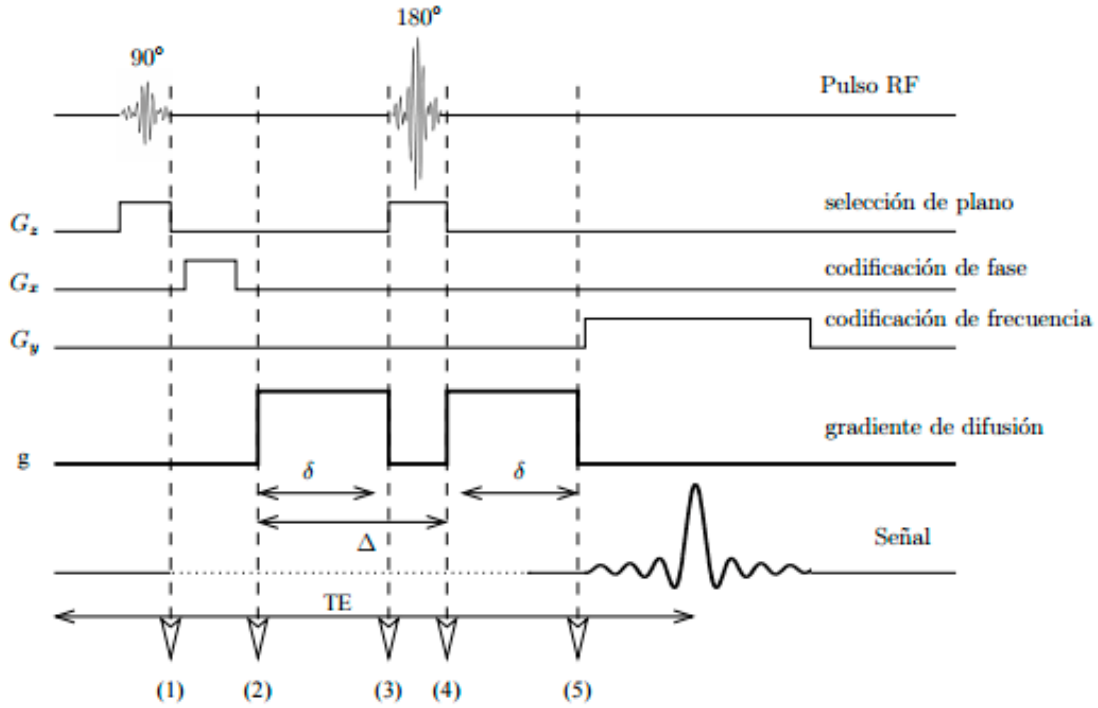


*Figura 3.2 – Secuencia de pulsos aproximada Stejskal – Tanner [35]*

El primer gradiente, llamado desfase, induce un cambio de fase para todos los espines, y el segundo gradiente, llamado gradiente de reposición de fase, invierte el cambio de fase. De esta manera, para las moléculas estacionarias, ambos desfases se cancelan mutuamente. No obstante, si las moléculas no son estacionarias, estarán sometidas al primer pulso de gradiente en una posición determinada, y al segundo pulso en otra posición distinta, así los gradientes serán distintos en magnitud pero no se anularan. Y la diferencia entre estos será proporcional al desplazamiento que sufren las moléculas en una dirección entre estos mismos gradientes. Así la intensidad de señal resultante de un vóxel se obtiene de la Ecuación 3.4:

$$S = S_0 e^{-bD} \qquad \textit{Ecuación 3.4 [2]}$$

Donde $S$ es la intensidad medida tras aplicar el gradiente de difusión, $S_0$ es la intensidad de la señal en ausencia de ese gradiente, $D$ es el coeficiente de difusión y $b$ es el factor de sensibilidad de la difusión definido por LeBihan en 1986, que describe la secuencia de pulsos, la intensidad del gradiente y las constantes físicas.

$$b = \gamma^2 \delta^2 \left[\Delta - \left(\frac{\delta}{3}\right)\right] |g|^2 \qquad \textit{Ecuación 3.5 [2]}$$

En esta ecuación $\gamma$ es la relación giromagnética del protón, $|g|$ es la fuerza de los pulsos de gradiente, $\delta$ es ancho del gradiente aplicado y $\Delta$ es el tiempo entre los pulsos.

Al no poderse diferenciar el movimiento generado por los gradientes de concentración con el movimiento molecular, la ecuación para el caso anisótropo, es más precisa si se transforma el coeficiente de difusión $D$, al coeficiente aparente de difusión ($ADC$), generado por todos los movimientos producidos en cada vóxel, tanto en el interior de las células como el de las moléculas exteriores (Ver Figura 3.3).

*Figura 3.3 – Movimientos relativos al ADC [35]*

$$S = S_0 e^{-bADC}$$ *Ecuación 3.6*

Los tensores se pueden calcular con esta ecuación obtenida al generalizar y añadir información de la difusión anisótropa en diferentes direcciones.

$$S = S_0 e^{-\gamma^2 \delta^2 \left[\Delta - \left(\frac{\delta}{3}\right)\right] g^T D g}$$ *Ecuación 3.7 [2]*

Esta fórmula se restablece en el caso isótropo con $D = DI$, siendo $I$ el tensor identidad. Además insertando vectores gradientes normalizados $\hat{g} = \frac{g}{|g|}$, se puede reescribir haciendo uso del factor *b* de Le Bihan (Ecuación 3.5).

$$S_k = S_0 e^{-b \hat{g}_k^T D \hat{g}_k}$$ *Ecuación 3.8 [2]*

Como ya se ha explicado anteriormente $S_0$ es la intensidad de la señal en ausencia de un gradiente de difusión, $S_k$ es la intensidad medida tras aplicar el $k - ésimo$ gradiente de difusión en la dirección $\hat{g}_k$, $\hat{g}_k^T D \hat{g}_k$ representa el coeficiente de difusión en la dirección $\hat{g}_k$ y $b$ es el factor de ponderación de difusión Le Bihan (Ecuación 3.5) correspondiente gradiente aplicado.

Para cuantificar la difusión, es necesario analizarla en cada uno de los ejes coordenados de un espacio tridimensional. Por lo que por medio de gradientes de campo magnético se reorientan las moléculas para determinar la libertad con la que se pueden desplazar en una dirección específica. Este estudio se hace con un número mínimo de

gradientes con diferentes direcciones. Tal es así, que para cada dirección de gradiente que se tome se genera un volumen de datos DWI.

Llegado a este punto hay que tener en cuenta que la información de difusión no es una cantidad escalar, particularmente en la sustancia blanca, por tanto es necesario el uso de la notación tensorial (Ecuación 3.8), y que el lote de volúmenes generado es considerable, tanto como para buscar otras maneras de facilitar y agilizar los cálculos en otros soportes, como GPU.

## 3.3. PROCESADO DE DTI (*DIFFUSION TENSOR IMAGING*)

DTI es una de las técnicas de DMRI en la que describe la difusión del agua mediante un modelo gaussiano. A partir del volumen de datos, es capaz de calcular para cada vóxel del volumen un tensor de segundo orden, a partir de la medida de la difusión en distintas direcciones [2] [31] [36].

La variable $D$ (Ecuación 3.1) es la difusividad o coeficiente de difusión, y se obtiene mediante el procesado de DTI. También se ha explicado en el último punto, que debido a que la información de difusión varía dependiendo de la dirección, se necesita un tensor que describa la movilidad de las moléculas a lo largo de cada dirección y la correlación entre estas direcciones.

Para determinar los valores de dicho tensor, se necesitan al menos seis imágenes sensibilizadas con distintos gradientes, así como una imagen no sensibilizada a la difusión.

### 3.3.1.Interpretación de los datos del tensor

Al caracterizar la difusión gaussiana en medios anisótropos se emplean tensores de difusión, matrices de $3x3$ simétricas positivas (Ecuación 3.9). Al ser de estas características se obtienen tres autovectores ortogonales y tres autovalores positivos, y así con esta particularidad se garantiza la condición PSD (*Positive Semi-Definite*). El mayor de los autovectores indica la dirección principal de difusión y los tres autovalores siendo $(\lambda_1 \geq \lambda_2 \geq \lambda_3)$ indican la difusividad en la dirección de cada autovector.

$$D = \begin{pmatrix} D_{xx} & D_{xy} & D_{xz} \\ D_{yx} & D_{yy} & D_{yz} \\ D_{zx} & D_{zy} & D_{zz} \end{pmatrix} \qquad \textit{Ecuación 3.9}$$

Donde $D_{ij} = D_{ji}$ para $i \neq j$.

Si se quiere calcular esa matriz, hay que remontarse a la ecuación de Stejskal – Tanner para el caso anisótropo (Ecuación 3.8). En la que se refleja que la intensidad de la señal en cada vóxel disminuye en presencia de difusión gaussiana.

Dichos tensores son generalmente representados por elipsoides.

En la Figura 3.4 parte A muestra el tensor de difusión como un elipsoide, teniendo los ejes principales a lo largo de los vectores propios $(\lambda_1, \lambda_2, \lambda_3)$. Sin embargo en la parte B el tensor de difusión se representa mediante una función de distribución de la orientación.

Al ser una matriz $3x3$ tiene seis grados de libertad, así que para determinar el tensor se deben realizar al menos siete medidas con diferentes direcciones de gradiente, de las cuales seis son ortogonales entre sí y la restante es en ausencia del gradiente. Si el sistema de coordenadas sobre el que se ha realizado la medición coincide con el del autosistema, todos los componentes del tensor, salvo las de la diagonal serán nulos. Si esto es así los desplazamientos en cada una de las direcciones perpendiculares resultaran independientes entre sí.

Para resolver el sistema de ecuaciones resultante, se utilizará la manera más habitual, mínimos cuadrados. Se obtendrán los vectores de difusión para cada vóxel, obtenidos mediante la resolución del sistema de ecuaciones.

La magnitud que expresa la anisotropía del tejido es particular, ya que representa el grado de variación de la difusión dependiendo de la dirección. A partir de los autovalores se puede obtener tres variaciones de las múltiples existentes.

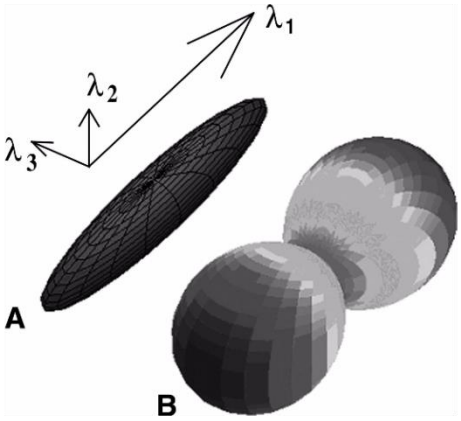


*Figura 3.4 – Diagrama de tensor de difusión*

*· Coeficiente de volumen (VR, Volume Ratio)*

Cociente entre el volumen del elipsoide y el de una esfera con la misma difusividad media[3]. Sus valores oscilan entre 1 totalmente isótropo y 0 totalmente anisótropo. En ocasiones se emplea $1 - VR$, ya que así la relación del valor con el comportamiento sea igual que en los casos anteriores.

$$VR = \frac{(\lambda_1 + \lambda_2 + \lambda_3)}{D^3}$$

*Ecuación 3.10 [2]*

3 Difusividad media: Valor medio de los autovalores del tensor y representa la cantidad de difusión en cada vóxel.

$$MD = \frac{(\lambda_1 + \lambda_2 + \lambda_3)}{3} = \frac{(D_{xx} + D_{yy} + D_{zz})}{3}$$

Traza: medida equivalente a la MD, es la suma de los elementos diagonales del tensor.

$$Trace = (\lambda_1 + \lambda_2 + \lambda_3) = (D_{xx} + D_{yy} + D_{zz})$$

*· Anisotropia relativa (RA, Relative Anisotropy)*

Es el coeficiente de la variación de los autovalores. Los valores que toma son desde 0, isótropo y $\sqrt{2}$, idealmente anisótropo.

$$RA = \frac{1}{\sqrt{2}} \frac{\sqrt{(\lambda_1 - \lambda_2)^2 + (\lambda_2 - \lambda_3)^2 + (\lambda_1 - \lambda_3)^2}}{(\lambda_1 + \lambda_2 + \lambda_3)} = \frac{\sqrt{3}}{\sqrt{2}} \frac{\left| D - \frac{1}{3} trace(D) I \right|}{trace(D)}$$

*Ecuación 3.11 [2]*

*· Anisotropia fraccional (FA, Fractional Anisotropy)*

Mide la diferencia entre el elipsoide, que es el tensor, y una esfera. Varía entre 0, totalmente isótropo y por tanto una esfera, y 1, totalmente anisótropo y difusión lineal. Se puede llegar a obtener sin calcular los autovalores explícitamente.

$$FA = \frac{1}{\sqrt{2}} \frac{\sqrt{(\lambda_1 - \lambda_2)^2 + (\lambda_2 - \lambda_3)^2 + (\lambda_1 - \lambda_3)^2}}{\sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}} = \frac{\sqrt{3}}{\sqrt{2}} \frac{\left| D - \frac{1}{3} trace(D) I \right|}{|D|}$$

*Ecuación 3.12 [2]*

### 3.3.2.Estimación de tensores

Para realizar la estimación de tensores, hay que partir de la ecuación descrita anteriormente (Ecuación 3.8) y de un tensor simétrico $3x3$ de difusión $D$. En este caso típico representado en la Figura 3.5, el tensor tiene seis grados de libertad, así que para hallar el tensor, al menos son necesarias 6 mediciones, recogidas de diferentes gradientes de dirección no colineales, a mayores es necesario el baseline, la imagen base $S_0$.

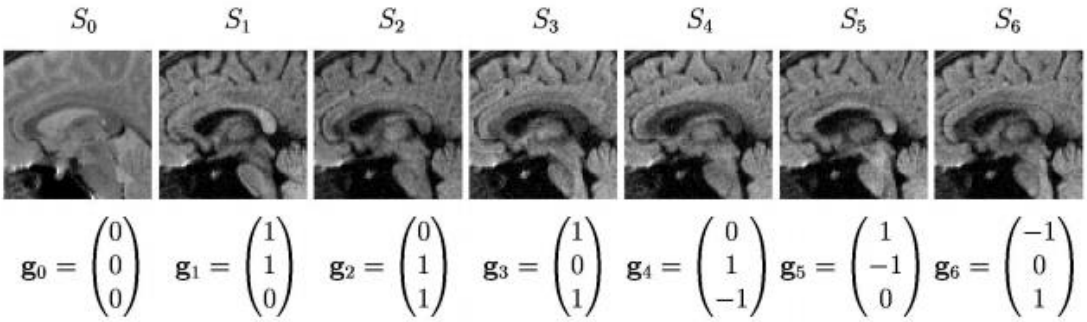


*Figura 3.5 – Ejemplo de baseline más seis mediciones con diferentes gradientes de dirección [2]*

En la Figura 3.5 se muestra un sencillo ejemplo, un conjunto de imágenes correspondientes a las diferentes direcciones de los gradientes, donde $\{S_0, S_1, \dots, S_6\}$ contienen la intensidad de la señal en presencia de gradientes $g_k$ Hay que especificar que el gradiente $|g_0| = 0$, contiene la intensidad de señal en ausencia de un gradiente de campo.

Como ya se ha mencionado al principio de este punto, hay que partir de la Ecuación 3.8. Si se incorporan los gradientes a la ecuación, se obtiene como resultado un sistema de seis ecuaciones. Resolviendo este sistema de ecuaciones para cada vóxel del volumen de datos, se hallarían los valores del tensor de difusión.

$$\begin{cases} ln(S_1) = ln(S_0) - bg_1^T D g_1 \\ ln(S_2) = ln(S_0) - bg_2^T D g_2 \\ ln(S_3) = ln(S_0) - bg_3^T D g_3 \\ ln(S_4) = ln(S_0) - bg_4^T D g_4 \\ ln(S_5) = ln(S_0) - bg_5^T D g_5 \\ ln(S_6) = ln(S_0) - bg_6^T D g_6 \end{cases}$$

*Ecuación 3.13 [2]*

### 3.3.3.Visualización bidimensional y reconstrucción de las fibras

La tractografía es el procedimiento que se utiliza para obtener una representación de las trayectorias de las fibras cerebrales *in vivo*, mejorando la descripción de los datos.

Considera numerosas intersecciones entre las fibras, aunque sus resultados dependen del algoritmo de seguimiento utilizado [37].

La difusión anisótropa en la materia blanca está organizada en haces de fibras. Las fibras representadas mediante tractografía se suelen considerar como representación de axones individuales o fibras nerviosas, pero es más acertado verlas como líneas de rápida difusión que solo reflejan la arquitectura axonal. La Figura 3.6 se ha realizado una composición de tres formas diferentes de representar la difusión.

La tractografía es una técnica prometedora que intenta reconstruir los caminos de la sustancia blanca en el cerebro de manera no invasiva. Por ende, actualmente se están desarrollando continuamente mejoras en los procedimientos de tractografía. En concreto, la tractografía global lidera esta técnica, por encima de los métodos locales. Estos últimos reconstruyen cada fibra de manera independiente, a pesar de que son los más comunes, se ven afectados por su falta de exactitud, así los pequeños errores se pueden llegar a acumular y afectan representativamente al resultado final. Por el contrario los métodos globales intentan reconstruir todas las estructuras neuronales a la vez, buscando la organización que mejor describen los datos proporcionados.

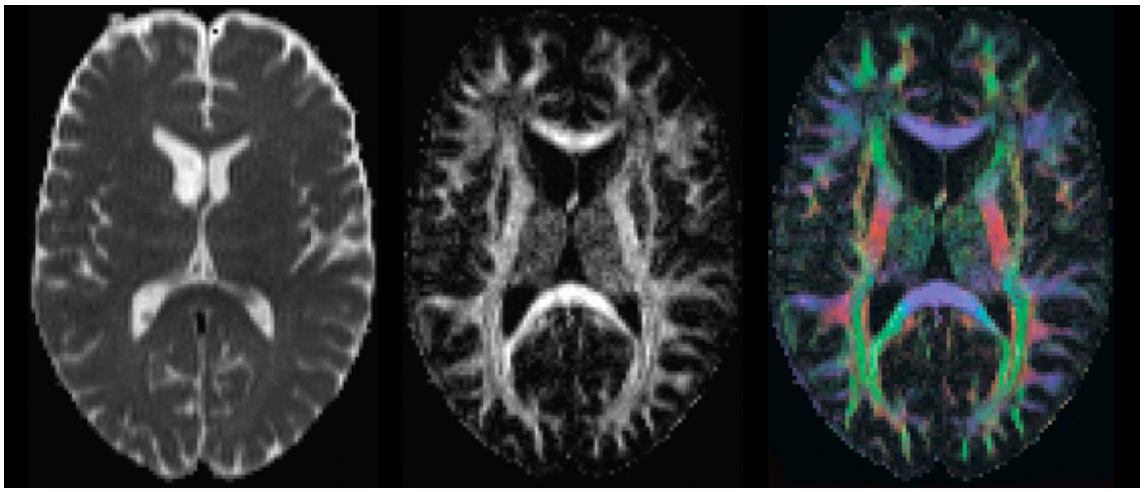

*Figura 3.6 – Representación de la difusión promediada en 3 direcciones ortogonales (izq.), mediante FA (centro) y mediante código de colores (drch.) [38]*

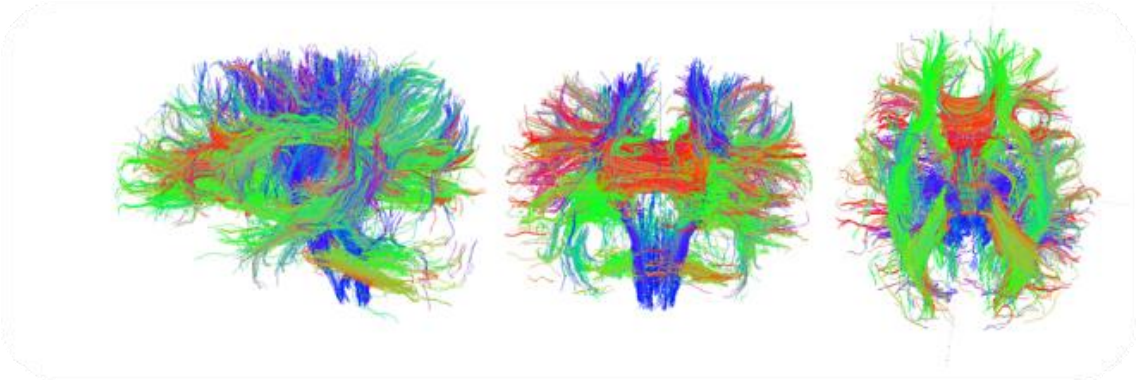

*Figura 3.7 – Ejemplo de tractografía cerebral*

Los colores tan llamativos mostrados tanto en la Figura 3.7 como en la Figura 3.8, no son cuestión de estética, si no que existe un acuerdo a la hora de colorearlos. Se los agrupa por colores en función de la dirección preferencial que sigan las fibras.

- Los tractos del eje X o comisurales, de derecha a izquierda cruzando así de un hemisferio al otro, como las fibras del cuerpo calloso, se muestran de color rojo.
- Los tractos del eje Y, representan principalmente las vías ópticas que se proyectan en la dirección anteroposterior, y reciben unos matices verdes.
- Los tractos del eje Z, dirección rostro – caudal o superior – inferior, como por ejemplo los haces córtico – espinales descendentes e incluso si se da el caso, la médula espinal, aparecen de color azul.

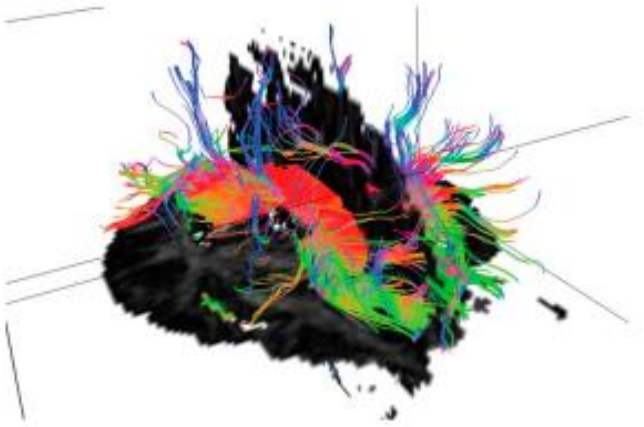

*Figura 3.8 – Ejemplo en 3D de tractografía cerebral [38]*

## 3.4. ANÁLISIS DE LA MATERIA BLANCA CON DTI

Gran parte del avance del análisis de la sustancia blanca, del conocimiento de la conectividad cerebral y del funcionamiento de las redes neuronales ha venido de la mano del desarrollo de técnicas de tractografía como las de Imágenes de Tensor de Difusión en Resonancia Magnética [38]. La anisotropía es una característica que se da sobre todo en las zonas de la materia blanca (Figura 3.9), estudiadas con esta nueva técnica.

El desarrollar técnicas que sean capaces de conocer qué tractos están lesionados, permite comprender mejor la asociación entre la localización y la extensión de la lesión y la sintomatología clínica. Por el momento, la aplicación experimental permitiría crear un mapa de conectividad cerebral para poder entender el funcionamiento normal y patológico del cerebro, o incluso de la etapa de maduración cerebral en la infancia y los cambios no patológicos de la materia blanca en el envejecimiento normal. En la Figura 3.10 se observan las diferencias existentes en los tractos del cerebro dependiendo de la edad.

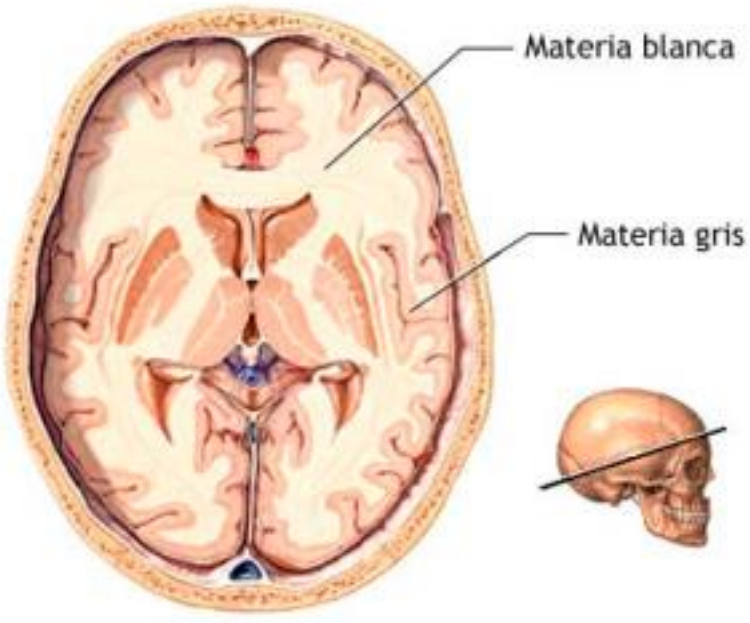


*Figura 3.9 – La materia blanca en el cerebro [35]*

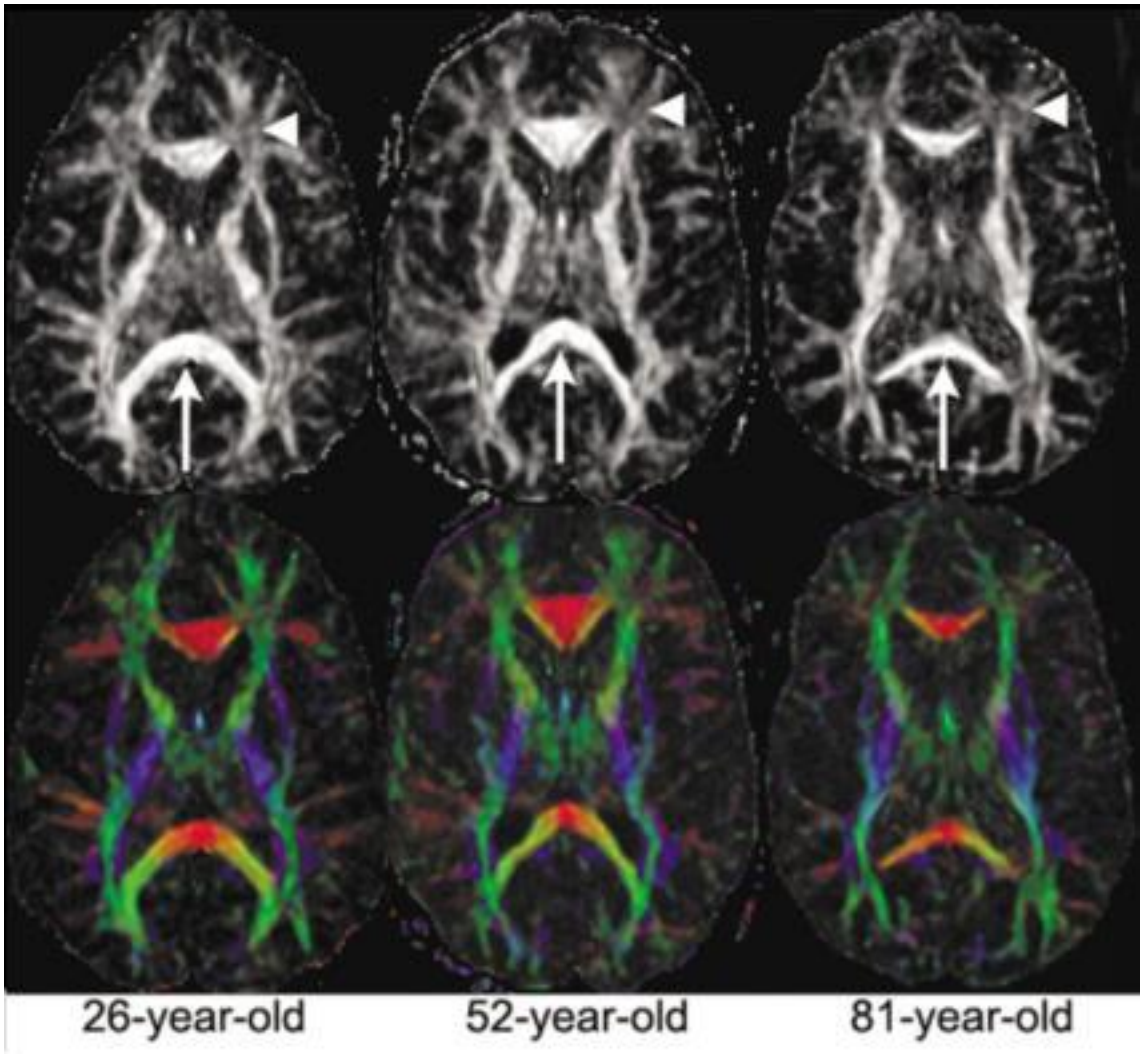


*Figura 3.10 – Comparación mediante DTI en base a la edad en el cuerpo calloso y en la SB frontal [38]*

Existen varias formas de analizar la materia blanca, para comprender mejor estas técnicas se van a explicar brevemente dos de ellas.

### 3.4.1.VBM (Voxel – Based Morphometry)

Se emplea para caracterizar diferencias regionales de volumen y concentración del tejido cerebral a partir de imágenes de resonancia magnética. El planteamiento base consiste en la normalización espacial de todos los sujetos en un mismo espacio, dando como resultado un mapa estadístico que muestra las regiones que presentan diferencias significativas entre los grupos estudiados.

Consta de unas fases bien diferenciadas para llevar a cabo dicho análisis:

· *Creación del atlas*

Hay que realizar una normalización espacial, un alineamiento de las imágenes. Se puede utilizar un atlas pre-existente en un espacio estándar, o crearse a partir de los volúmenes que se van a estudiar.

· *Normalización espacial*

Se realizan transformaciones tanto lineales como no lineales de los volúmenes, registrando las imágenes en un mismo atlas común.

· *Segmentación*

Para simplificar la observación, se desglosa la materia gris de la materia blanca y de otras regiones no cerebrales. Está cimentada en la complementación del conocimiento de las estructuras cerebrales, junto con el estudio de intensidad de los vóxeles.

· *Suavizado*

Se suavizan las imágenes segmentadas utilizando un núcleo gaussiano isótropo, ayudando así a que los datos presenten una distribución más gaussiana, y haciendo que cada vóxel contenga el valor medio de la región colindante al mismo. Además mejora las posibles erratas naturales de la normalización espacial y de la segmentación.

· *Análisis estadístico*

Se realizan estadísticas, involucrando un grupo con sujetos sanos, y otro de pacientes, para compararlos estadísticamente.

### 3.4.2.TBSS (Tract-Based Spatial Statics)

TBSS está ideado para DTI, no como VBM que estaba ideado para comparar la materia gris en imágenes estructurales de resonancia magnética convencional. Así que dos de los problemas fundamentales de VBM, la duda de si se está comparando la misma parte

de materia blanca al comparar los dos mismos vóxeles de dos volúmenes distintos y la controversia de la cantidad de suavizado utilizado, para no alterar los resultados, son disminuidos con este otro método; este método será el que se utilice para el desarrollo de este proyecto.

Para alcanzar estos objetivos, se calcula un esqueleto común a partir de FA, que representa los haces de fibras, en general, pertenecientes a todos los sujetos de estudio. Más tarde se proyecta la FA de cada sujeto sobre el esqueleto obtenido, de tal forma que cada vóxel del esqueleto toma el valor de la FA dentro de un área cercana y pequeña, resolviendo los problemas de alineado y de suavizado.

Las fases en las que se divide este método son las siguientes:

*· Preprocesamiento*

Alineación de las imágenes DWI de cada sujeto entre sí, para corregir los movimientos de la cabeza producidos en la prueba.

*· Registrado*

Alineación de las imágenes de FA de los sujetos sobre un mismo espacio. Primero una alineación lineal y después una no lineal, para ajustarse a la solución y ajustar los detalles respectivamente. De la misma manera que pasaba en VBM, se puede crear un atlas o utilizar una plantilla.

*· Creación de la imagen de FA media y esqueleto*

Se calcula la media de las imágenes FA obtenidas, y el resultado es una imagen FA media. Para conseguir el esqueleto, se compara cada vóxel con sus vecinos, y se registra el de mayor valor.

*· Umbral al esqueleto*

El umbral aplicado es del valor de la FA, se consigue restringir el esqueleto, eliminando los valores erróneos producidos de las etapas anteriores.

*· Proyección de las imágenes FA sobre el esqueleto*

Se proyectan las imágenes y se busca en cada punto del esqueleto, en la dirección perpendicular a los tractos, el valor máximo de FA y se asigna al vóxel correspondiente.

*· Análisis estadístico*

Comparación vóxel a vóxel de las imágenes de cada sujeto.

# 4. ANÁLISIS DE LOS MODELOS DE PROGRAMACIÓN PARA GPGPU

El impulso de las capacidades de las tarjetas gráficas se ha desarrollado considerablemente, como ya se ha visto en el capítulo 0. Gracias a esto, se han creado varios modelos de programación para la computación de propósito general. En este capítulo se va a hacer un pequeño estudio de las dos arquitecturas más desarrolladas. No son las únicas, están, entre otras, ATI STREAM [39] y Larrabee [40], pero distan mucho de parecerse tanto a CUDA como a OpenCL.

## 4.1. ARQUITECTURA CUDA

La demanda ocasionada por la popularidad de la GPGPU, provocó que Nvidia desarrollara una arquitectura de cálculo paralelo, CUDA (*Compute Unified Device Architecture*), aunque solo se encuentra disponible para la gama de tarjetas gráficas Nvidia serie GeForce 8 y posteriores [4] [6] [27] [41].

La pila de *software* de CUDA, como se muestra en la Figura 4.1 está compuesta por un controlador de *hardware* denominado CUDA Driver, una API llamada CUDA *Runtime* y dos librerías matemáticas de alto nivel, CUDA *Libraries* [4]. Nvidia consigue un alto rendimiento al haber rediseñado el *hardware* que se dedicaba a hacer procesado de vértices, fragmentos o geometría. Dejó de ser específico de las etapas del *pipeline* y se generalizó para ser capaz de llevar acabo cualquiera de las etapas programables. Este diseño se llamó *Unified shader model*.

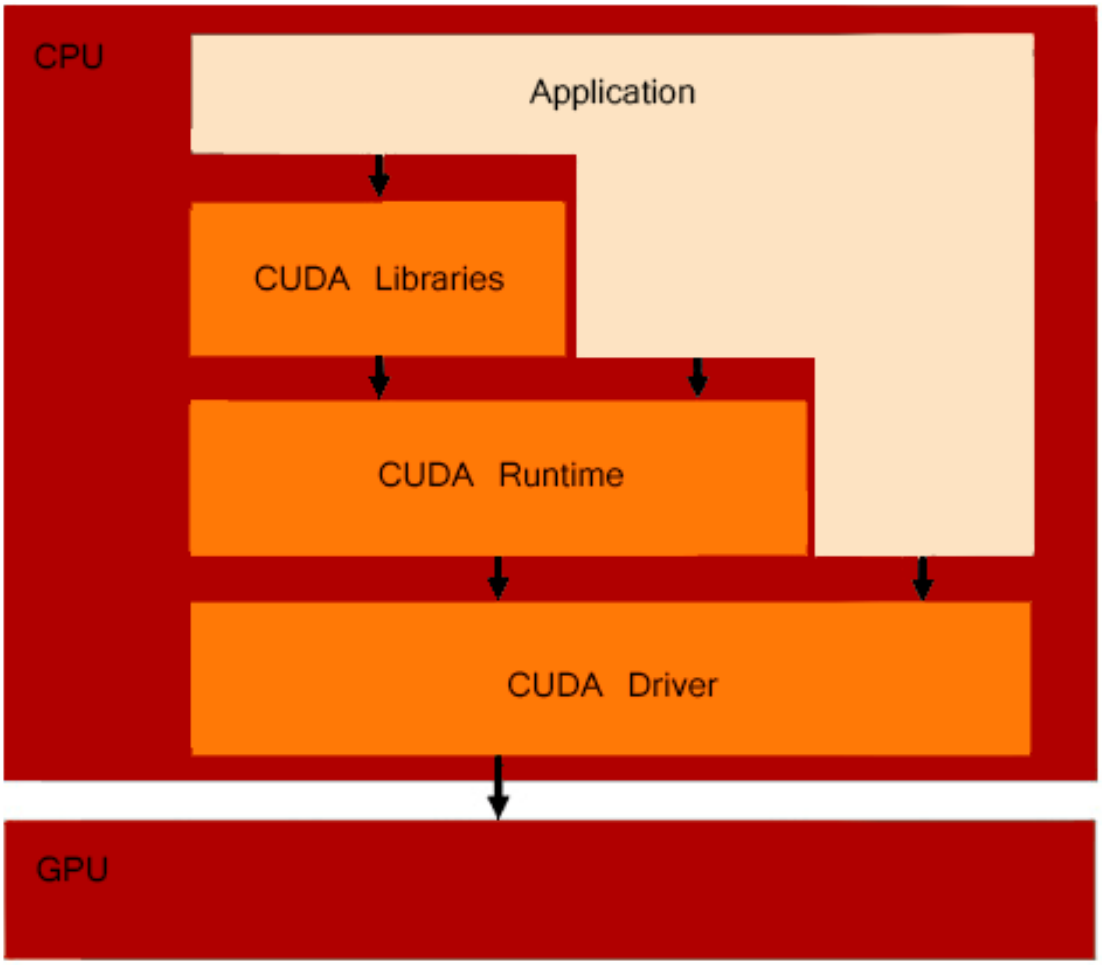


*Figura 4.1 – Pila de Cuda [22]*

CUDA proporciona direccionamiento general de memoria RAM, que funciona como memoria externa con mucho ancho de banda y con una latencia elevada, para mayor flexibilidad de programación al suministrar operaciones de memoria de dispersión y reunión (*scatter* y *gather*). Desde la perspectiva de la programación, esto se traduce en la habilidad para leer y escribir desde cualquier posición de la RAM, como en CPU. En la Figura 4.2 se puede observar que este modelo de direccionamiento permite obtener mayor flexibilidad en la programación, en el sentido de que ofrece tanto la operación de reparto de datos como la de obtención de estos.

El modelo de programación de CUDA está diseñado para crear aplicaciones que optimicen el paralelismo de las GPU. Este diseño contiene tres puntos clave que se van a ver a continuación.

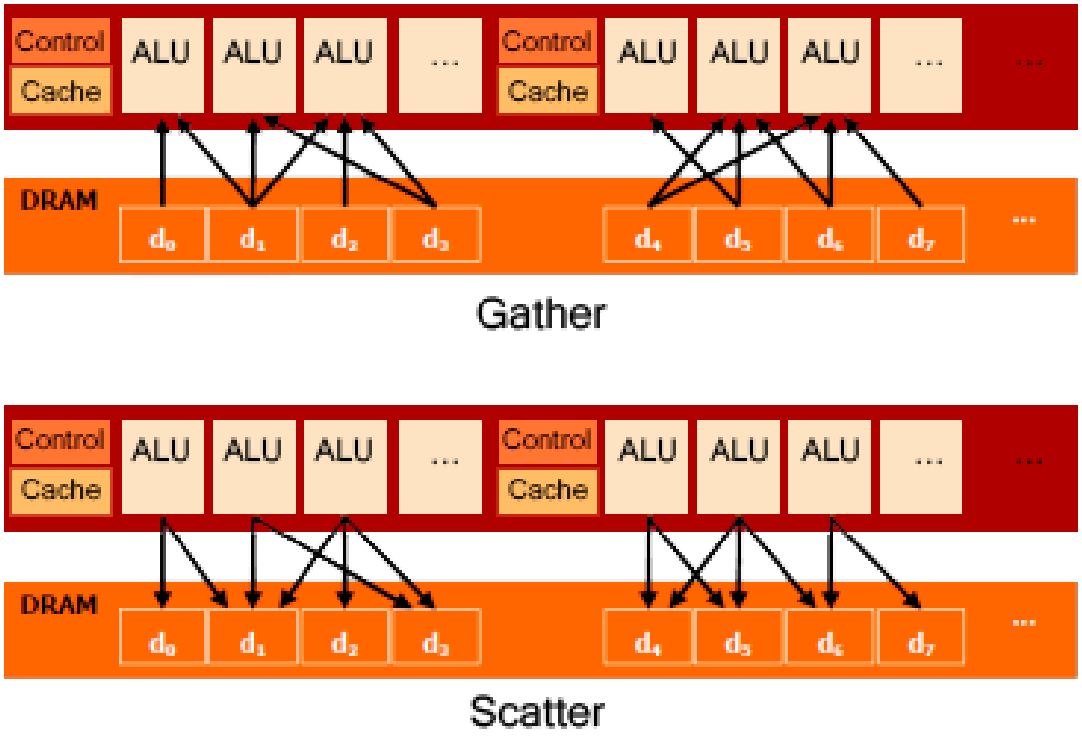


*Figura 4.2 – Operaciones de memoria Gather y Scatter (dispersión y reunión) [22]*

### 4.1.1.Un coprocesador multi-hilo

Al programar mediante CUDA, la GPU se ve como un dispositivo de computación capaz de ejecutar un elevado número de hilos en paralelo. Realiza las funciones actuando como coprocesador de la CPU principal.

De manera más precisa, una parte de una aplicación que es ejecutada muchas veces sobre datos distintos, puede ser aislada en una función que se ejecuta en el mecanismo tantas veces como hilos distintos hay. Como dicha función es compilada, la instrucción obtenida a partir del *device* (GPU), y el programa resultante, llamado *kernel*, se descargaran en el *device*.

Ambas mantienen su memoria RAM, memoria de *host* y memoria de dispositivo, y pueden copiarse datos desde una a la otra a través de llamadas API optimizadas que usan DMA (*Direct Memory Access*).

### 4.1.2.Agrupación de hilos

Un bloque de hilos es una agrupación de hilos que pueden cooperar juntos compartiendo datos eficientemente mediante memoria compartida rápida y sincronizando su ejecución para coordinar accesos a memoria y asegurar consistencia en los datos. Cabe destacar que los hilos CUDA son considerablemente ligeros. Esto hace que se creen en un tiempo muy breve y que el proceso de almacenar o restaurar el contexto de un dispositivo para que múltiples procesos puedan compartirlo, llamado conmutación de contexto, sea instantáneo. El programador tiene la responsabilidad de declarar los miles de hilos que la GPU necesita.

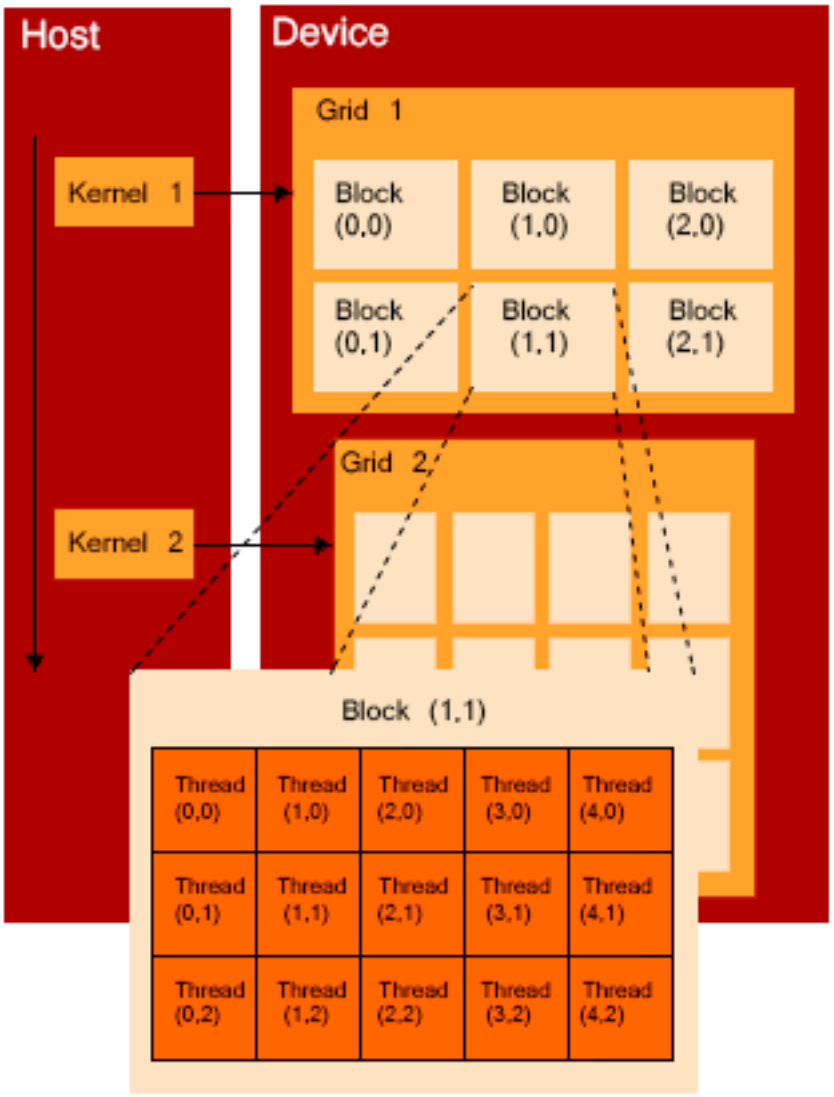


*Figura 4.3 – Organización de los threads en CUDA [22]*

Desde el punto de vista del programador, el sistema está compuesto por una CPU, llamado *host*, y por uno o más dispositivos, llamados *devices*, que son GPU. En la Figura 4.3 se ilustra como el *host* envía, para ejecutar en el *device,* los diferentes *kernels*, los programas que se quieren ejecutar en paralelo, de manera secuencial al dispositivo.

Un programa CUDA es un programa híbrido, las inicializaciones o la lectura de datos de entrada, se realizan en la CPU, luego muestra y almacena los resultados. Por otro lado, como indica la Figura 4.4, el código paralelo se ejecuta en la GPU.

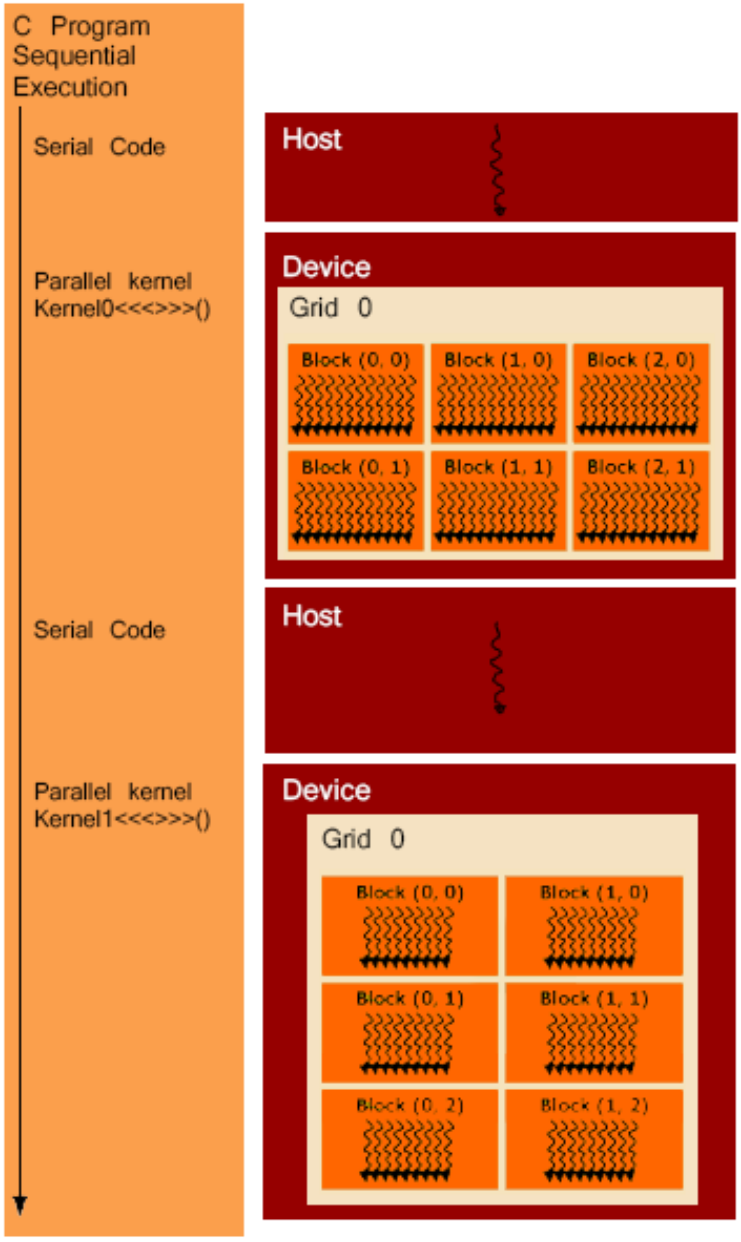


*Figura 4.4 – Etapas en la ejecución de un programa típico en CUDA [22]*

Un bloque de hilos es un lote de hilos que pueden cooperar juntos compartiendo datos eficientemente a través de algunas memorias rápidas de compartición de datos y sincronizando sus ejecuciones para coordinar los accesos a memoria. Existe un número limitado de hilos que un bloque puede contener. Sin embargo pueden ser agrupados en un *grid*, conjunto de bloques, si estos tienen las mismas dimensiones y ejecutan el mismo *kernel*. Existen dos identificadores, uno para los hilos y otro para los bloques, *thread ID* y *block ID* respectivamente. Los *thread ID* pueden ser definidos con dos o tres dimensiones para ayudar con los direccionamientos complejos.

Con este modelo los *kernels* pueden ejecutarse sin necesidad de recompilar en varios dispositivos de la misma arquitectura con diferentes capacidades de paralelismo.

### 4.1.3.Modelo de memoria

Un dispositivo CUDA está organizado como un conjunto de multiprocesadores o *streaming multiprocessors* (SM) como se puede advertir en la Figura 4.5, con una cantidad elevada de flujos de ejecución. Cada multiprocesador tiene una Instrucción Única y arquitectura de Datos Múltiples lo que se corresponde a una arquitectura SIMD (*Single Instruction Multiple Data*): en cualquier ciclo de reloj, cada procesador ejecuta la misma instrucción pero opera sobre datos diferentes.

Los espacios de memoria local y global son implementados como regiones de la memoria del *device*. Cada multiprocesador accede a la caché de textura mediante una unidad de textura que implementa diferentes modos de direccionamiento, así como un filtro de datos para algunos formatos específicos.

Desde el punto de vista del dispositivo se puede acceder a diferentes tipos de memoria con los modos siguientes:

· Acceso de lectura/escritura a la memoria global, por *grid*

·Acceso solo de lectura a la memoria constante, por *grid*;

· Acceso de lectura/escritura a los registros, por flujo

· Acceso de lectura/escritura a la memoria local, por flujo

· Acceso de lectura/escritura a la memoria compartida, por bloque

· Acceso de lectura/escritura.

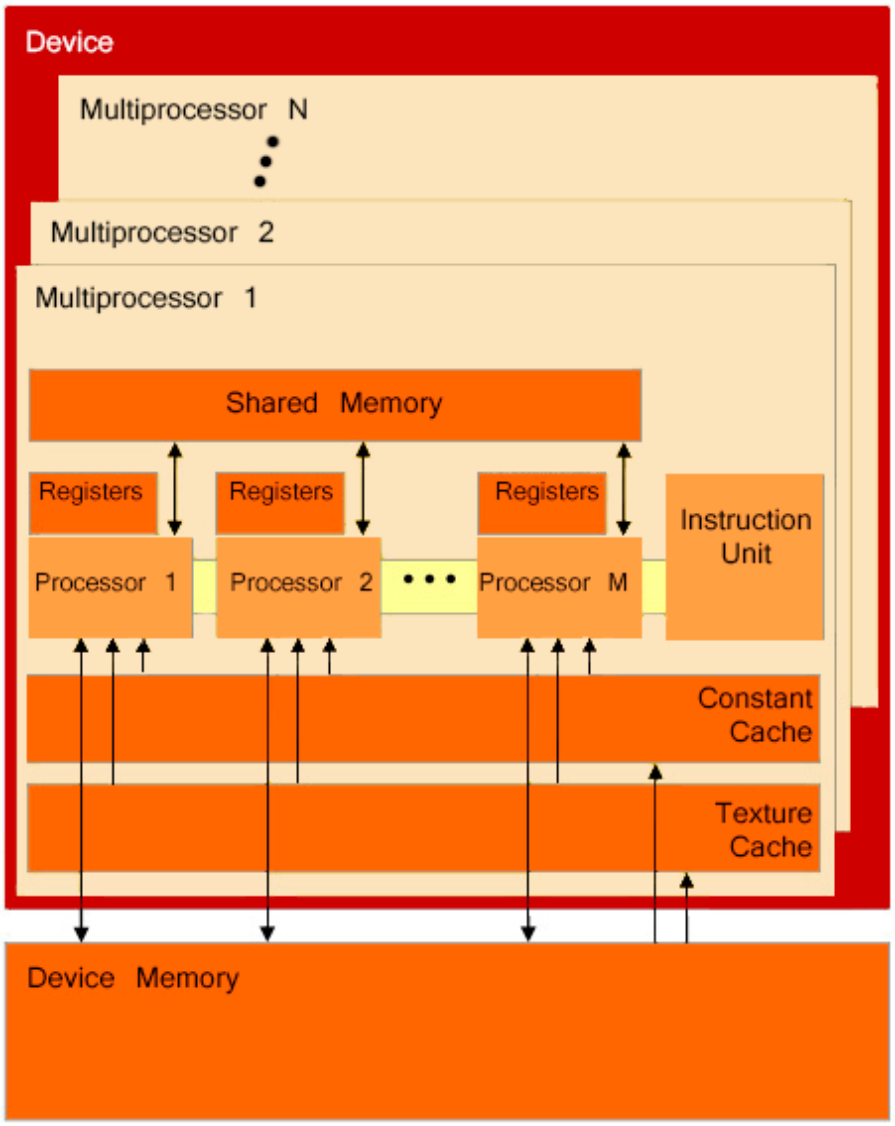


*Figura 4.5 – Modelo Hardware en CUDA [22]*

### 4.1.4.Gestión de kernels y organización de flujos

La función que ejecutan los diferentes flujos durante la fase paralela en el dispositivo se denomina *kernel*, que está identificado por la palabra clave `__global__`. Es necesario que sea llamado desde el procesador principal cuando se invoca un *kernel,* para generar el conjunto de flujos o *grid* de flujos que se ejecutará en el dispositivo. El *kernel* finaliza cuando

todos sus flujos finalizan la ejecución en el *grid* correspondiente. Una vez finalizado el *kernel*, la ejecución del programa continúa en el procesador principal hasta que se invoca otro *kernel*. En la Figura 4.4 vista anteriormente, se pueden observar dos *grids* de flujos.

Existen dos palabras clave más que se pueden utilizar ante la declaración de una función: `__device__` que indica que la función declarada es una función CUDA de dispositivo, que únicamente se ejecuta en un dispositivo CUDA, y `__host__` que indica que la función es una función simple de C que se ejecuta en el procesador principal y que puede ser llamada desde cualquier función de procesador principal. Se pueden utilizar simultáneamente al declarar las funciones, generando por parte del compilador dos versiones de la misma función, una que se ejecuta en el procesador principal y que solo puede ser llamada desde una función de procesador principal y otra que se ejecuta en el dispositivo que solo puede ser llamada desde el dispositivo o función de *kernel*.

Un *kernel* se ejecuta mediante un conjunto de flujos diferenciados. Para este cometido, CUDA incorpora palabras clave para hacer referencia al índice de un flujo, `threadIdx.x`, `threadIdx.y`, `threadIdx.z`, de esta manera cada uno de los flujos tendrá diferentes valores en cada ejecución.

Los *grids* que utiliza CUDA están formados por muchos flujos organizados en una jerarquía de dos niveles, tal y como se puede ver en la Figura 4.3. Todos los bloques de un *grid* tienen el mismo número de flujos y se organizan de la misma manera. Además cada bloque en un *grid* tiene una coordenada única en un espacio de dos dimensiones mediante las palabras clave `blockIDx.x` y `blockIDx.y`. En la Figura 4.3, el *Grid 1* está compuesto por 6 bloques y está organizado como una matriz de $3x2$ bloques y cada bloque está organizado en un espacio de $5x3x1$ flujos de ejecución.

Cuando el procesador principal llama al kernel, se tienen que especificar las dimensiones del *grid* y de los bloques de flujos mediante los parámetros de configuración. Y para que todos los flujos de un bloque no pierdan la sincronización, CUDA ofrece un mecanismo para regularizarlos, bloqueando el flujo que ejecuta la función `__syncthreads()`, hasta que todos los flujos de su bloque lleguen hasta ese mismo punto.

## 4.2. ARQUITECTURA DE OPENCL

OpenCL es una API estándar, abierta, libre y multiplataforma orientado a la GPGPU. Este estándar permite ejecutar aplicaciones de manera paralela en procesadores gráficos (GPU) o centrales (CPU), diferentes en su arquitectura y ubicados en un sistema heterogéneo. De esta manera implica a todos los núcleos del procesador central o toda la capacidad de computación de la unidad de procesamiento gráfico a realizar la misma tarea. Incluye un lenguaje de programación, incorpora también una API para la librería de control de plataformas, además de librerías y un sistema de ejecución para soportar el desarrollo de *software* [7] [42] [43].

OpenCL consta de tres partes, la especificación de un lenguaje multiplataforma, una interfaz a nivel de entorno de computación y una interfaz para coordinar la computación paralela entre procesadores heterogéneos.

### 4.2.1.Arquitectura conceptual

Como se ve en la Figura 4.6, la arquitectura de OpenCL está constituida por un procesador principal, normalmente CPU, que es quien ejecuta el programa principal, y que está conectado a uno o más dispositivos OpenCL. Un dispositivo OpenCL tiene una o varias unidades de cómputo CU (*Compute Units*), y cada una de ellas está formada por uno o varios elementos de procesamiento PE (*Processing Elements*), donde se realizará la ejecución del programa.

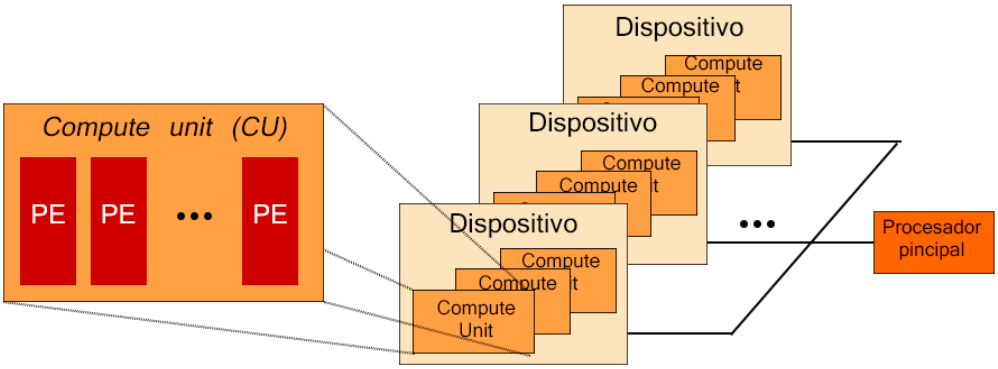


*Figura 4.6 – Arquitectura conceptual de OpenCL [5]*

### 4.2.2.Modelo de paralelismo a nivel de datos

La manera de funcionar de OpenCL es muy similar a la de CUDA. Un programa en el procesador principal que controla e invoca la ejecución de los *kernels*, que son programas simples que se ejecutan en uno o más dispositivos. Cuando se hace la invocación de un *kernel*, el código se ejecuta en tareas elementales, *work items* que corresponderían a los flujos de CUDA, y los grupos de tareas elementales se identifican, como se puede observar en la Figura 4.7, dentro del rango de un espacio de índices de dimensión N, NDRanges, llamado *work groups*. El análogo en CUDA son los bloques. Tanto las tareas elementales como los grupos de tareas elementales, tienen su identificador local o identificador global correspondientemente.

En este modelo se dispone de interfaces para referir las tareas elementales. Por un lado la función `get_global_id()`, en la que dada una dimensión devuelve el identificador único de tarea elemental a la dimensión especificada. Por el otro `get_local_id()`, dada una dimensión devuelve el identificador de la tarea elemental dentro de su grupo a la dimensión especificada. Otras funciones establecidas devuelven la cantidad tareas elementales dentro de un grupo y la cantidad total de tareas elementales, `get_local_size()` y `get_global_size()`.

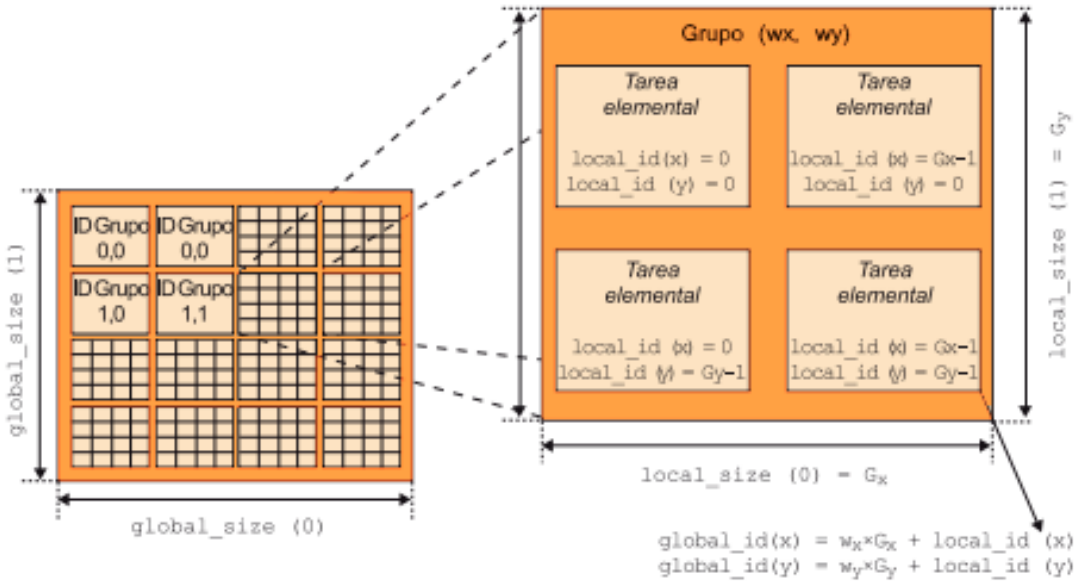


*Figura 4.7 – Ejemplo de rango de dimensión N de OpenCL en el que se pueden observar las tareas elementales, los grupos que forman e identificadores asociados [5]*

Las tareas elementales se pueden sincronizar entre ellas si están dentro de un mismo grupo. Si están en diferentes grupos, estas tareas no pueden sincronizarse a no ser que se termine el *kernel* o la invocación de uno nuevo.

### 4.2.3.Modelo de memoria

Las tareas elementales tienen acceso a 4 regiones de memoria distintas (ver Figura 4.8). Estas regiones tienen diferentes permisos dependiendo desde donde se acceda, si desde el *host* o desde el *kernel* (Tabla 4.1) [7]:

- La memoria global es la que pueden utilizar todas las unidades de cálculo de un dispositivo. Los *work items* pueden leer o escribir en cualquier elemento de un objeto de memoria, pero depende de las particularidades del dispositivo para leer o escribir en la memoria caché de datos de memoria global.

- La memoria constante se mantiene durante la ejecución del *kernel*, así se puede utilizar para almacenar datos constantes para acceso de solo lectura de todos los *work items* de un dispositivo durante la ejecución de un *kernel*. Es el procesador principal quien se encarga de asignar e inicializar los objetos de memoria que residen en el espacio de memoria.

- La memoria local es la memoria que se asigna a un grupo de trabajo. Se utiliza para asignar variables que son compartidas por todos los *work items* dentro de un *work group*.

- La memoria privada es la que utiliza únicamente cada elemento de trabajo. Las variables definidas dentro de esta memoria no son visibles para el resto de los *work item*.

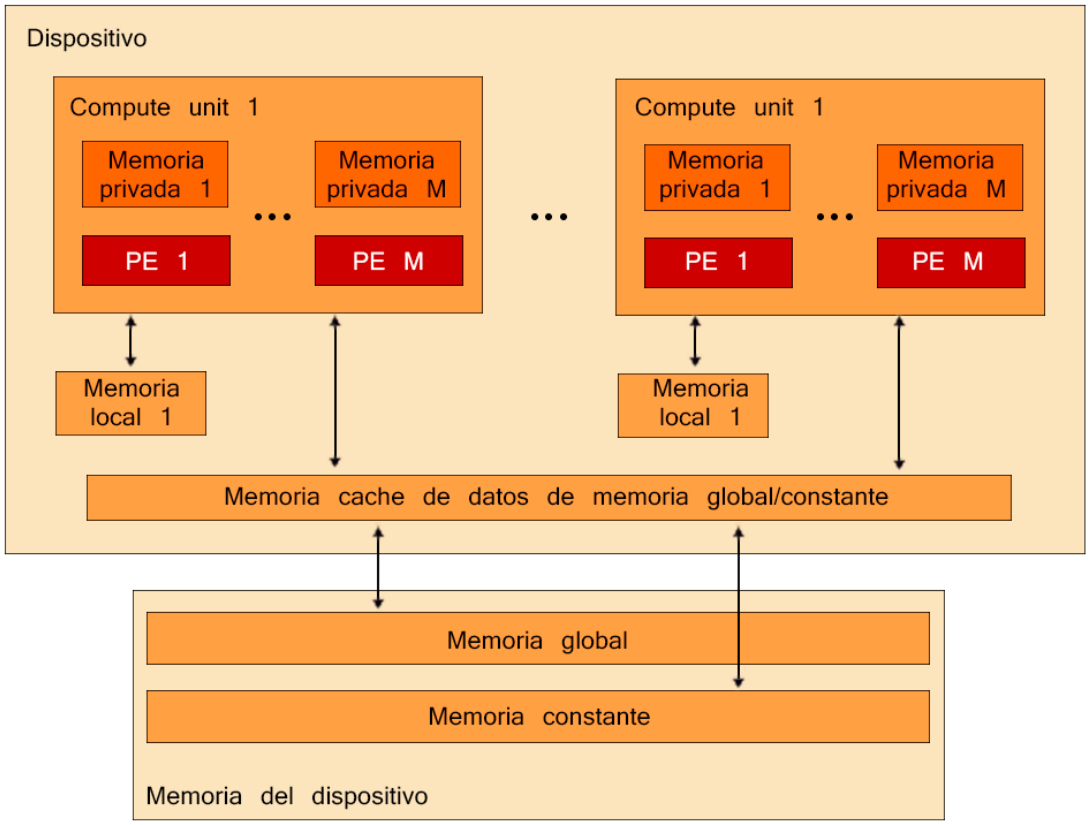


*Figura 4.8 – Arquitectura y jerarquía de memoria de un dispositivo OpenCL [5]*

| | Global | Constante | Local | Privada |
|---|---|---|---|---|
| Host | Reserva dinámica<br>Lectura/ Escritura | Reserva dinámica<br>Lectura/ Escritura | Reserva dinámica<br>Inaccesible | No reservable<br>Inaccesible |
| Kernel | No reservable<br>Lectura/ Escritura | Reserva estática<br>Sólo lectura | Reserva estática<br>Lectura/ Escritura | Reserva estática<br>Lectura/ Escritura |

*Tabla 4.1 – Tipo de acceso en el modelo de memoria de OpenCL [7]*

Para crear y componer objetos en la memoria de un dispositivo, la aplicación se ejecuta en el *host* y utiliza la interfaz de OpenCL. Hay dos formas de que la memoria del procesador principal y los dispositivos interactúen, copiando los datos explícitamente. Para

ello el procesador principal envía las instrucciones de transferencia entre la memoria del objeto y la memoria del procesador principal. Estas llamadas pueden ser bloqueantes, una vez haya finalizado la llamada se puede acceder a los datos desde el procesador principal con seguridad, o no bloqueantes, en las que la llamada a la función finaliza inmediatamente y se saca de la cola, a pesar de que no es seguro utilizar la memoria desde el procesador principal. La segunda forma de interacción es mapeando y desmapeando regiones de un objeto OpenCL en la memoria. Esta forma permite que el procesador principal tenga una región correspondiente a objetos OpenCL en su espacio de memoria. Estas instrucciones también pueden ser bloqueantes o no bloqueantes.

### 4.2.4.Gestión de *kernels* y dispositivos

Posteriormente, los kernels se ejecutarán en algunos de los dispositivos soportados por OpenCL: GPU s, CPU s y otros dispositivos.

Un *kernel* es una función declarada en un programa. Está identificado por la palabra clave `__kernel`. La jerarquía de memorias definidas anteriormente, sirve para declarar argumentos dependiendo de en qué zona de memoria se encuentren. En la implementación del cuerpo del *kernel* es definido el índice asociado, que se utiliza para seleccionar los datos que corresponden a cada una de las tareas elementales que instancie el *kernel*.

Antes de que el programa ejecute el *kernel* hay que gestionar los dispositivos, ya que OpenCL permite abstraer diferentes plataformas de *hardware*, algo que CUDA no puede hacer. Al crear el contexto, con la función `clCreateContext()`, se deben indicar ciertos parámetros como la cantidad y el tipo de dispositivos y plataformas del sistema, para ello existe `clGetDeviceIDs()` y `clGetPlatformIDs()` que lo examinan. También se debe crear una cola de instrucciones, asociada a un dispositivo, mediante `clCreateCommandQueue()`. Una vez creado esto, se crea el programa y se compila. Tiene que haber reservado espacio de memoria para los argumentos de entrada y de salida del *kernel*, que también se tiene que declarar en el programa principal. Cuando el dispositivo esté disponible para ejecutar el *kernel*, este se elimina de la cola y pasa a ser ejecutado (ver Figura 4.9).

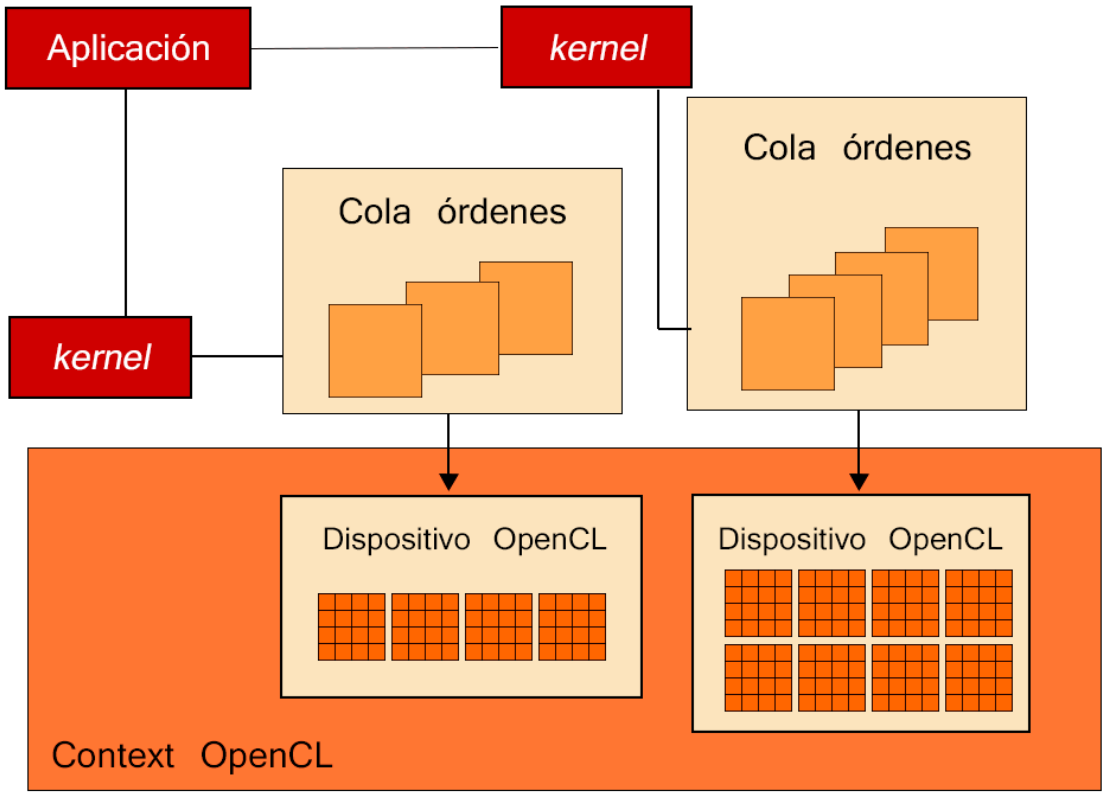


*Figura 4.9 – Gestión de dispositivos de OpenCL mediante contextos [5]*

## 4.3. CUDA VS OPENCL

Después de haber analizado estos lenguajes, y de haber investigado varios artículos ( [44] y [45] los más relevantes), se puede comenzar exponiendo una comparativa referente al modelo de programación.

Respecto al modelo de programación, ambos lenguajes son muy similares, los hilos de CUDA se corresponden con los *work-item* de OpenCL, los bloques de hilos con los *work-groups*, (ver Tabla 4.2). A pesar de la gran similitud en el modo de entender ambos lenguajes, se observan pequeñas diferencias; en el modelo de memoria, no es lo mismo la memoria local de CUDA y la memoria local de OpenCL, aunque si son semejantes las memorias global y constante de ambos.

| OpenCL | CUDA |
|---|---|
| Kernel | Kernel |
| Programa procesador principal | Programa procesador principal |
| NDRange (Rango de dimensión N) | Grid |
| Work item (Tarea elemental) | Flujo |
| Work group (Grupo de tareas) | Bloque |
| get_global_id(0) | blockIdx.x*blockDim.x+threadIdx.x |
| get_local_id(0) | threadIdx.x |
| get_global_size(0) | gridDim.x*blockDim.x |
| get_local_size(0) | blockDim.x |
| Compute Unit (CU) | Streamming Multiprocessor (SM) |
| Processing Element (PE) | Streamming Processor (SP) |

*Tabla 4.2 – Algunas equivalencias entre OpenCL y CUDA*

Debido a estas similitudes, es sencillo realizar una portabilidad de los *kernels*, tanto de CUDA a OpenCL, como viceversa. Aunque en la parte del *host* no hay apenas correspondencia, encontrándose así una de las principales diferencias. En CUDA la programación del *host* para ejecutar un determinado *kernel* es sencilla, mientras que en OpenCL para ejecutar ese mismo *kernel* supone más complicaciones.

Si se analiza la portabilidad de código de unos sistemas a otros, se advierte que con el lenguaje de programación CUDA no se tiene el abanico de posibilidades que se obtendría en OpenCL. En tarjetas gráficas que tengan la arquitectura de Nvidia, puede ejecutarse OpenCL y CUDA, mientras que en las que tengan arquitectura de ATI solo puede ejecutarse OpenCL.

OpenCL tiene un modelo de gestión de recursos más complejo, soportando múltiples plataformas y diferentes fabricantes, por eso le permite abstraer diferentes plataformas *hardware* disponibles y seleccionar las deseadas.

En definitiva, en lo referente a facilidad de programación destacaría CUDA sobre OpenCL, pero si se da más importancia a la portabilidad, como se ha hecho en este proyecto, la elección del lenguaje es OpenCL.

# 5. ANÁLISIS DE LA APLICACIÓN

En los capítulos anteriores se ha hecho una comparación de los diferentes modelos de programación aptos para el desarrollo de este algoritmo en particular, determinando cual es el más idóneo para este marco. Además se ha hecho un estudio del análisis la sustancia blanca, los beneficios y los métodos.

Antes de acoplar estos dos pilares fundamentales del proyecto, se plantea un punto a definir, la estimación de tensores de difusión. Se ha visto el caso general y en este apartado se muestra una técnica estándar para calcular estos tensores de difusión, los mínimos cuadrados (LS*, Least Squares*).

Exponiendo está técnica, se comprenderá mejor el algoritmo a implementar, con el que se calculará el logaritmo de los DWI.

## 5.1. MÍNIMOS CUADRADOS

Cuando se manejan grandes cantidades de datos, es importante relacionar las variables del problema mediante una expresión matemática. En ocasiones esa expresión, así planteada no tiene solución, sin embargo si es posible determinar una pseudosolución que mejor minimice una norma $\|Ax - b\|$. Si la norma que se emplea es la norma euclidea, al problema se le conoce como el de estimación por mínimos cuadrados [46] [47].

Se aplican logartimos a partir de la ecuación de Stejskal – Tanner, Ecuación 3.8 y a continuación se desarrollará el método de los mínimos cuadrados

$$\log\left(\frac{S_k}{S_0}\right) = -bg_k^T D g_k \qquad \textit{Ecuación 5.1}$$

Para facilitar la comprensión se asumirá que $-bg_k^T g_k$ es $A$, una matriz de $(kx6)$.

$$S = A \quad D \qquad \textit{Ecuación 5.2}$$

$$(kx1) = (kx6)\,(6x1)$$

$$A^T S = (A^T A) \quad D \qquad \textit{Ecuación 5.3}$$

$$(6xk)(kx1) = (6xk)(kx6)\,(6x1)$$

$$(A^T A)^{-1} \quad A^T S = D \qquad \textit{Ecuación 5.4}$$

$$(6xk)(kx6)\,(6xk)(kx1) = (6x1)$$

Donde $S$ es un vector de tantas componentes como gradientes haya. $D$ hace referencia al tensor de difusión, que como ya se ha indicado antes el tamaño en cualquiera de los casos es de $(6x1)$. Y las dimensiones de $A$, simplificada anteriormente, dependen de los gradientes.

## 5.2. DESARROLLO DE LA SECUENCIA DE STEJSKAL - TANNER

Al caracterizar el caso típico anterior (Figura 3.5), el cual tenía seis gradientes que corresponderían al subíndice $k$, cada gradiente contenía tres parámetros, y para simplificar se sustituye el logaritmo de $\frac{S_k}{S_0}$ por $S$, quedaría tal como la Ecuación 5.5

$$S = -b \begin{pmatrix} g_{11} & g_{12} & g_{13} \\ g_{21} & g_{22} & g_{23} \\ g_{31} & g_{32} & g_{33} \\ g_{41} & g_{42} & g_{43} \\ g_{51} & g_{52} & g_{53} \\ g_{61} & g_{62} & g_{63} \end{pmatrix} D \begin{pmatrix} g_{11} & g_{21} & g_{31} & g_{41} & g_{51} & g_{61} \\ g_{12} & g_{22} & g_{32} & g_{42} & g_{52} & g_{62} \\ g_{13} & g_{23} & g_{33} & g_{43} & g_{53} & g_{63} \end{pmatrix} \qquad \textit{Ecuación 5.5}$$

El factor de sensibilidad de la difusión definido por LeBihan $b$ es un escalar. El logaritmo del cociente de la intensidad medida tras aplicar un gradiente entre la intensidad en ausencia de él, corresponde a una matriz de $(kx1)$ elementos, siendo $k$, el número de gradientes tal y como se había indicado anteriormente. Y como se puede discernir de la Ecuación 5.5, las dimensiones de $g_k^T$ son $(kx3)$ y las de $g_k$ son $(3xk)$, siendo ambas dependientes de los gradientes. De esta manera, y desarrollando los mínimos cuadrados en esta ecuación, se obtendría que las dimensiones del coeficiente de difusión $D$ serían $(6x1)$. Es decir un vector de tantas componentes como gradientes haya.

A continuación se calculará este vector desarrollando el método de los mínimos cuadrados.

Si se desarrolla $-bg_1^T D g_1$, a partir de la Ecuación 5.1 se obtiene:

$$\log\left(\frac{S_1}{S_0}\right) = -b\,(g_{11}g_{12}g_{13}) \begin{pmatrix} a & b & c \\ b & d & e \\ c & e & f \end{pmatrix} \begin{pmatrix} g_{11} \\ g_{12} \\ g_{13} \end{pmatrix} \qquad \textit{Ecuación 5.6}$$

Esta ecuación está descrita para el primer gradiente con sus tres componentes. Si se multiplican las matrices y se describe para $k$ gradientes se obtiene:

$$\begin{pmatrix} \log\left(\frac{S_1}{S_0}\right) \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \log\left(\frac{S_k}{S_0}\right) \end{pmatrix} = -b \begin{pmatrix} {g_{11}}^2 & 2g_{11}g_{12} & 2g_{11}g_{13} & {g_{12}}^2 & 2g_{12}g_{13} & {g_{13}}^2 \\ & & \cdot & & & \\ & & \cdot & & & \\ & & \cdot & & & \\ & & \cdot & & & \\ & & \cdot & & & \\ & & \cdot & & & \\ {g_{k1}}^2 & 2g_{k1}g_{k2} & 2g_{k1}g_{k3} & {g_{k2}}^2 & 2g_{k2}g_{k3} & {g_{k3}}^2 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix}$$

*Ecuación 5.7*

Resumiendo, para conseguir la matriz A de $(kx6)$, que será la que se emplee para el desarrollo de la técnica de los mínimos cuadrados, hay que multiplicar los componentes de cada gradiente de la manera propuesta en la matriz central de la Ecuación 5.7. Dependiendo de los gradientes que se tengan se obtendrán ese mismo número de filas.

Posteriormente, para resolver la Ecuación 3.8, hay que reorganizar aplicando la técnica de mínimos cuadrados, desarrollada en las ecuaciones anteriores (Ecuación 5.2, Ecuación 5.3 y Ecuación 5.4). De tal manera que en un lado de la igualdad se halle la relación entre la intensidad de la señal medida tras aplicar el $k-$ é$simo$ gradiente y la intensidad de la señal en ausencia de él en cada vóxel, se aplique el logaritmo a esa relación y se multiplique como se ha visto en la Ecuación 5.4, siendo $A$ la matriz readaptada de los gradientes. En el otro lado de la igualdad queda el tensor de difusión, incógnita a la que se quiere dar solución.

Como hay que realizar esa operación tantas veces como vóxeles haya, eso implica cuantiosas veces, tantas que se puede reducir el tiempo de cómputo de una CPU si se implementa el algoritmo en una GPU. A raíz de esa idea es por la cual se ha desarrollado ese proyecto.

# 6. DISEÑO E IMPLEMENTACIÓN

En este capítulo se va a tratar el diseño y a analizar la implementación del desarrollo de la secuencia de Stejskal – Tanner. Igualmente se profundizará en las técnicas de paralelización vistas anteriormente, así como se especificarán el resto de detalles que conforman el programa.

## 6.1. CABECERA

El primer paso consiste conocer el formato que tiene el archivo a analizar y preparar el programa. Para eso hay que estudiar el archivo de cabecera donde se encuentran especificadas las diversas variables. Es importante determinar el valor exacto de las variables, ya que almacenar datos equívocamente hace que la ejecución del programa no sea útil, e incluso existe la posibilidad de que no se llegue a terminar dicha ejecución.

Por tanto se van a estudiar los campos más destacados que aparecen en la cabecera.

### 6.1.1. *Type*

Este campo especifica el formato de dato que se ha utilizado en el archivo. Dependiendo del tipo de dato varía la dimensión del mismo [48]. (Ejemplo: *short* - 2 *bytes*, *unsigned short* - 2 *bytes* o *float* - 4 *bytes*)

### 6.1.2. *Dimension*

En esta línea, detalla el número de dimensiones en el que está dividido el archivo. Los casos estudiados contienen cuatro dimensiones, que corresponderían al alto y al ancho de cada *slice*, al número de *slice* al que se le aplica cada gradiente, y al número de gradientes que se van a emplear.

### 6.1.3. *Space*

Como ya se ha explicado en el punto 3.3.3, las fibras están organizadas en tres grupos dependiendo la dirección que sigan. Los grupos son de derecha a izquierda, cruzando de un hemisferio al otro; anteroposterior, concierne a las fibras que están encaminadas de delante atrás; rostro - caudal, que correspondería a las fibras que están orientadas de arriba abajo

En la línea que se está analizando, aparece el nombre de esos tres campos, en un orden determinado. Ese orden es el que se ha de seguir al estudiar los datos del archivo.

### 6.1.4. *Sizes*

El valor de cada dimensión de las que se han definido anteriormente. Por norma existen cuatro dimensiones, y en este punto se establece el valor que toma cada una de esas dimensiones.

Para estructurar el programa, se declaran *array*. En este punto hay que tener en cuenta que los vectores tienen que tener un tamaño cómodo y equilibrado respecto a los datos que se van a ejecutar. El problema se establece en la dificultad para pasarlos de la función al *kernel* y el manejo dentro de cada una.

### 6.1.5. *Thicknesses*

Parámetro que índica el grosor del corte. Determina el tamaño del vóxel.

### 6.1.6.*Kinds*

Como ya se ha explicado, existen cuatro dimensiones, el alto y el ancho de cada *slice*, al número de *slices* a los que se le aplica un gradiente determinado y el número de gradientes

existentes. Este campo te determina el orden en el que están recogidas esas cuatro dimensiones en el archivo de datos.

Al leer la cabecera, hay que interpretar este campo, que junto con el campo *Sizes* (6.1.4) obtendremos las cuatro dimensiones y su orden correspondiente. Habitualmente el número de gradientes está determinado por el término "*list*" o "*vector*", y acostumbra a estar en la primera o en la cuarta posición. De esta manera las otras tres posiciones serían las dimensiones "x" "y" y "z" definidas así en el programa.

### 6.1.7.*Endian*

Determina el formato en el que se almacenan los datos de más de un *byte* [49]. Este término no es exclusivo de las resonancias magnéticas. Existen dos sistemas el *big-endian* que consiste en representar los bytes en el orden natural, y el sistema *little-endian* en el que se almacenan los datos de menos significativo a más significativo. En este último sistema se hace más intuitivo el acceso a los datos por que se efectúa de manera incremental de menos relevante a más relevante.

Hay que reparar en cuál de los dos sistemas se tienen almacenados los datos, para realizar bien los cálculos.

### 6.1.8.*Encoding*

Tipo de codificación que tiene el archivo. Por ejemplo raw o gzip.

### 6.1.9.*Data file*

Nombre del archivo en el que están los datos relacionados con la cabecera que se está comprobando.

### 6.1.10. *Modality*

Variedad de resonancia magnética a la que corresponde el archivo analizado. Existen diferentes variedades de resonancia magnética y en este campo indica cuál de ellos es el que se está examinando.

En este caso la modalidad es siempre DWMRI que corresponde al lote de imágenes (DWI) que se obtiene de realizar una resonancia magnética de difusión (DMRI)

#### 6.1.11. *DWMRI_b-value*

Factor de sensibilidad de la difusión definido por LeBihan, el factor b. Este factor se emplea para resolver la ecuación de Stejskal – Tanner.

En la cabecera se lee y se guarda para ser utilizado más tarde.

#### 6.1.12. *DWMRI_gradient*

El valor de los gradientes que van a ser utilizados en este análisis. Dependiendo de los gradientes que existan en el estudio, así serán las líneas registradas en la parte final de la cabecera. Cada línea de estas cuenta con tres campos diferentes, ya que son vectores de tres componentes. Además el nombre del gradiente está declarado con el número de gradiente que concierne.

Como el número de gradientes no tiene por qué ser siempre el mismo, primero hay que saber cuántos gradientes existen en ese documento y reservar memoria dinámica. A partir de ahí se guardan en una matriz de tantas filas como gradientes haya.

## 6.2. PROGRAMA PRINCIPAL

Una vez visto el formato de los archivos de la cabecera, es momento de analizar el programa principal. Este módulo realiza diversas tareas que disponen y organizan la ejecución. Se van a iniciar las variables que se obtienen del archivo de cabecera y se va a llamar al resto de funciones de manera organizada.

Para empezar y guardar todos los datos de la cabecera que conciernen al estudio se ha declarado una estructura para que sea más cómodo el paso entre funciones. En esa estructura se han declarado variables tales como:

`Size.td` – Tamaño dato: El tamaño varía dependiendo de qué tipo sean los datos que están almacenados en el archivo (Véase 6.1.1). Para extraer los datos del

archivo y después separarlos es necesario saber qué tamaño tienen, si dos o cuatro o más *bytes*.

`Size.x`, `Size.y`, `Size.z` y `Size.g` - Dimensiones: El tamaño de cada dimensión viene dado en la cabecera en el apartado *Sizes* (Vease 6.1.4). Al pensar en el desarrollo del programa, cada gradiente afecta siempre a un número determinado de *slices* (corresponde a la variable `Size.z`) y cada *slice* tiene un ancho (`Size.x`) y un alto (`Size.y`). De esta manera existe un número que va a resultar muy útil en la ejecución del programa, `Size.xyz`, resulta de multiplicar el alto por el ancho por el número de *slices*. Así se tendrá un bloque de datos al que se le habrá aplicado un gradiente de entre todos los coexistentes (`Size.g`, número de gradientes).

`Size.b` – Factor de sensibilidad: Este dato, explicado anteriormente (Véase: 6.1.11) viene definido en la cabecera. Factor que más tarde se va a emplear en el desarrollo de la secuencia.

Igualmente se estudian los campos *endian* y *data file* (Vease 6.1.7 y 6.1.9). Uno para saber el orden en el que están almacenados los datos, y el otro campo para saber en qué archivo están guardado el resto de los datos.

En el archivo cabecera, también está definido el valor de cada gradiente (Véase 6.1.12). Este valor es distinto en cada resonancia magnética de difusión, por tanto para simplificar los cálculos, los gradientes se van a guardar en una matriz de tres columnas por tantas filas como gradientes halla.

En el apartado 5 Análisis de la aplicación, se vio que para desarrollar la secuencia de Stejskal – Tanner, había que despejar las incógnitas y en consecuencia se tenía que operar con los gradientes siguiendo un patrón. Este patrón fue el que se desarrolló en la Ecuación 5.7 en el que se obtienen 6 términos para cada gradiente. Para tenerlo presente en este punto, se va a escribir la relación que existe para un solo gradiente.

$$S_k = (g_{k1}{}^2 \quad 2g_{k1}g_{k2} \quad 2g_{k1}g_{k3} \quad g_{k2}{}^2 \quad 2g_{k2}g_{k3} \quad g_{k3}{}^2) \qquad \textit{Ecuación 6.1}$$

Por tanto, y después de este recordatorio, en el programa principal se realiza esta ecuación para cada uno de los gradientes que se tienen. Se guardan en una matriz esta vez de las mismas filas, una por cada gradiente y de seis columnas, una por cada término hallado.

En este punto de la ejecución se resuelven los mínimos cuadrados. Desarrollo que se verá en el punto 6.3. Al igual que la función que multiplica el resultado de los mínimos cuadrados con cada vóxel, también se explicará más adelante en el punto 6.4.

Para finalizar, en el programa principal, la última tarea que realiza es guardar los datos que estaban en el archivo detallado en 6.1.9 en un espacio reservado. Este espacio depende de los parámetros analizados antes, y por tanto el tamaño es variable. Con estos datos se opera en la función `multDatos()` mencionada anteriormente, que devuelve una matriz de dimensiones semejantes que se guarda en un fichero con la misma codificación que la que tenía *data file*.

## 6.3. MÍNIMOS CUADRADOS

Una vez detallado el algoritmo y expuesto el programa principal, el siguiente paso es implementar la técnica de mínimos cuadrados.

Para este punto, se ha decidido aprovechar las propiedades de las GPU. Para esto primero hay que confeccionar un código en el *host* necesario para inicializar el *kernel*, que más tarde se tratará.

### 6.3.1. Generación de código del host

Uno de los primeros pasos para la preparación es conseguir plataformas y elegir la adecuada. Para el objetivo de este programa, lo indicado es elegir un dispositivo GPU [7].

`clGetPlatformIDs()`: Esta función proporciona una lista de plataformas disponibles de OpenCL.

`clGetDeviceIDs()`: La función busca plataforma más adecuada y devuelve la lista de dispositivos de OpenCL encontrados. Para encontrar esos

dispositivos existe una opción que especifica el tipo de dispositivo de OpenCL que se está buscando.

Después de encontrar los dispositivos, el siguiente paso es crear un contexto para poder gestionar colas de comandos, objetos de memoria y objetos tipo *kernel*. Las colas de comandos es lo primero que se va a crear, asociado al contexto antes designado.

`clCreateContext()`: Crea un contexto en OpenCL. Un contexto se crea con uno o más dispositivos, y son utilizados durante el tiempo de ejecución para la gestión de diferentes objetos.

`clCreateCommandQueue()`: Objeto que lanza *kernel* en un dispositivo especifico, asociado a un contexto denominado anteriormente. Se pueden especificar una lista de propiedades en las que se determina cuando se ejecutará dicha objeto.

Después de estos pasos se tiene que construir un programa. Para eso primero

`clCreateProgramWithSource()`: Crea un programa para un contexto, y carga el código fuente especificado.

`clBuildProgram()`: Esta función construye un ejecutable para todos los dispositivos en el contexto asociado de OpenCL. El ejecutable se construye a partir del código fuente cargado en la función anterior.

Para que el programa funcione correctamente, es necesario definir uno o varios *buffer*, un *kernel*, y los argumentos que el *kernel* necesita para desarrollar su cometido. Todos estos objetos están asociados al dispositivo elegido anteriormente, a la cola de comandos específica para un contexto creado, y a un programa cargado de un fichero exterior.

`clCreateBuffer()`: Se crea un objeto que almacena un lote unidimensional de elementos. Pueden ser datos escalares o vectores o alguna estructura definida previamente. Estos *buffer* proporcionan a los argumentos la estructura necesaria para declararse, de esta manera los *buffer* pueden ser de lectura y de escritura, o de solo lectura o de solo escritura.

`clCreateKernel()`: Declara una función que se ejecuta en la GPU, denominado *kernel*. Este programa se identifica por la expresión `__kernel`, así mismo necesita los valores de los argumentos para ejecutarse.

`clSetKernelArg()`: Especifica el argumento que requiere el *kernel* para ejecutarse. Este argumento puede tener valores para más tarde emplearlos en la ejecución o puede estar declarado simplemente para almacenar los resultados de la ejecución. Un mismo *kernel* puede tener varios argumentos definidos en el segundo parámetro de la función desde el 0 hasta el número de argumentos – 1.

Por último, solo resta ejecutar el *kernel* y guardar en la memoria del *host* el resultado de la ejecución.

`clEnqueueNDRangeKernel()`: Ejecuta el *kernel* asociado a una cola de comandos y por tanto asociado a un contexto y a un dispositivo, y que tiene relacionado uno o varios argumentos. En esta función se definen dos parámetros necesarios para la ejecución del kernel, ya que designan las repeticiones del kernel:

`global_work_size`: Describe el número de elementos de trabajo globales. Dimensiones y tamaño de cada dimensión.

`local_work_size`: Describe el número de elementos de trabajo que conforman un grupo de trabajo. Dimensiones y tamaño de cada dimensión. Menor por definición que la variable anterior.

`clEnqueueReadBuffer()`: Lee desde el objeto *buffer* y guarda el resultado en la memoria del *host*.

Esta secuencia de pasos es la que se tiene que realizar para preparar el *host*. De hecho, se va a repetir en varias ocasiones.

En el diseño de esta función se programaron tres *kernel*. Para la programación de estos no es necesario buscar las plataformas tres veces, ya que son las mismas, ni tampoco cambiar de contexto, ya que su ejecución es secuencial. Así mismo, si los *kernel* se encuentran en el mismo archivo, a la hora de realizar el ejecutable solo sería necesario hacerlo una vez. De esta manera solo habría que volver a lanzar las funciones de crear *buffer* y *kernel*, asociar los argumentos, ejecutar el *kernel* y almacenar los resultados.

Tal y como se explicó en la sección 5.1, la técnica de mínimos cuadrados consiste en despejar una parte de la ecuación tratando de hallar el vector que mejor se aproxime a la igualdad.

Los pasos a realizar para llegar a esa aproximación son:

Transpuesta de la matriz $A => A^T$

Multiplicación $A^t * A$

Inversa de $(A^T * A)^{-1}$ *(Véase: 6.5 Inversa de la matriz)*

Multiplicación $(A^T * A)^{-1} * A^T$

*Transpuesta de la matriz* $A => A^T$

- *Buffer*:
  - Matriz original, actúa de entrada.
  - Matriz transpuesta, dispone de una etiqueta que especifica "solo escritura". En esta matriz se almacena la salida del *kernel*.
- *Kernel*:
  - Se crea un *kernel* asociado al programa creado a partir de un archivo fuente. Este archivo dispone de tres *kernel* distintos, a partir del nombre el programa distingue cuál de los tres está relacionado a esta ejecución.
- Argumentos:
  - *Buffer* de la matriz original.
  - Número de columnas que tiene la matriz original, seis.
  - Número de gradientes, depende de archivos.
  - *Buffer* de la matriz transpuesta. Para almacenar los datos.
- Ejecutar *kernel*:
  - `Global_work_size`: Dos dimensiones, (Número de columnas, gradientes) para realizar la transpuesta.
- Guardar resultados:
  - Se almacena la matriz transpuesta para ser utilizada más adelante.

### *Multiplicación $A^t * A$*

- *Buffer*:
  - Matriz multiplicada, actúa de salida, etiqueta: "solo escritura".
- *Kernel*:
  - Se crea un *kernel* asociado al programa creado a partir de un archivo fuente. Este archivo dispone de tres *kernel* distintos, a partir del nombre el programa distingue cuál de los tres está relacionado a esta ejecución.
- Argumentos:
  - *Buffer* de la matriz transpuesta. No ha sido necesario crearlo, el *buffer* de la salida del *kernel* anterior es apropiado.
  - *Buffer* de la matriz original. El *buffer* creado para el *kernel* anterior también es apto.
  - Número de columnas que tiene la matriz original, seis.
  - Número de gradientes, depende de archivos.
  - *Buffer* de la matriz multiplicada. Para almacenar los datos.
- Ejecutar *kernel*:
  - `Global_work_size`: Dos dimensiones, (Número de columnas, Número de columnas). La matriz multiplicada resultante es cuadrada de $(6\ x\ 6)$.
- Guardar resultados:
  - Se almacena la matriz multiplicada para ser utilizada más adelante.

### *Multiplicación $(A^T * A)^{-1} * A^T$*

- *Buffer*:
  - Matriz inversa, actúa de entrada.
  - Matriz multiplicada 2, no confundir con la multiplicación y el *buffer* anterior. Actúa de salida, etiqueta: "solo escritura".
- *Kernel*:
  - Se crea un *kernel* asociado al programa creado a partir de un archivo fuente. Este archivo dispone de tres *kernel* distintos, a partir del nombre el programa distingue cuál de los tres está relacionado a esta ejecución.
- Argumentos:
  - *Buffer* de la matriz inversa.

- *Buffer* de la matriz transpuesta. No ha sido necesario crearlo, el *buffer* de la salida del primer *kernel* es apropiado.
- Número de columnas que tiene la matriz original, seis.
- Número de gradientes, depende de archivos.
- *Buffer* de la matriz multiplicada 2. Para almacenar los datos.

- Ejecutar *kernel*:
  - `Global_work_size`: Dos dimensiones, (Número de gradientes, Número de columnas). La matriz multiplicada resultante es cuadrada de ($6\ x\ nº\ Gradientes$).
- Guardar resultados:
  - Se almacena la matriz multiplicada para ser utilizada más adelante.

### 6.3.2.Generación de código OpenCL

En este apartado se describe el lenguaje de programación OpenCL utilizado para crear *kernel* que se ejecutan en el dispositivo [7].

Para empezar, el calificador que diferencia el código OpenCL y el código del host es el termino `__kernel`. Esta expresión declara una función que se ejecutará en un dispositivo OpenCL.

Además existen otro tipo de términos que especifican la región de memoria que se utiliza para designar el objeto que recibe el *kernel*. El término que más se va a emplear es `__global`, ya que se utiliza para hacer referencia a objetos de memoria establecidos en la memoria global.

Para realizar los bucles necesarios, se necesita el término inicializado antes de lanzar el *kernel*, en la parte del *host*: `Global_work_size`. Este término declarado y explicado en las secciones anteriores, es el que determina cuantas veces tiene que ejecutar el *kernel*. Puede tener entre una y tres dimensiones. La manera que tiene el *kernel* de realizar las ejecuciones es simple. Si se tuviera una dimensión, el *kernel* se ejecuta tantas veces como anuncie esa dimensión. Si fueran dos dimensiones, el *kernel* se ejecuta $D_1 * D_2$ veces. Y si fueran tres dimensiones el *kernel* tendría que ejecutarse $D_1 * D_2 * D_3$ veces. De esta manera está organizado en hilos por eso se hace por vectores en vez de por escalares.

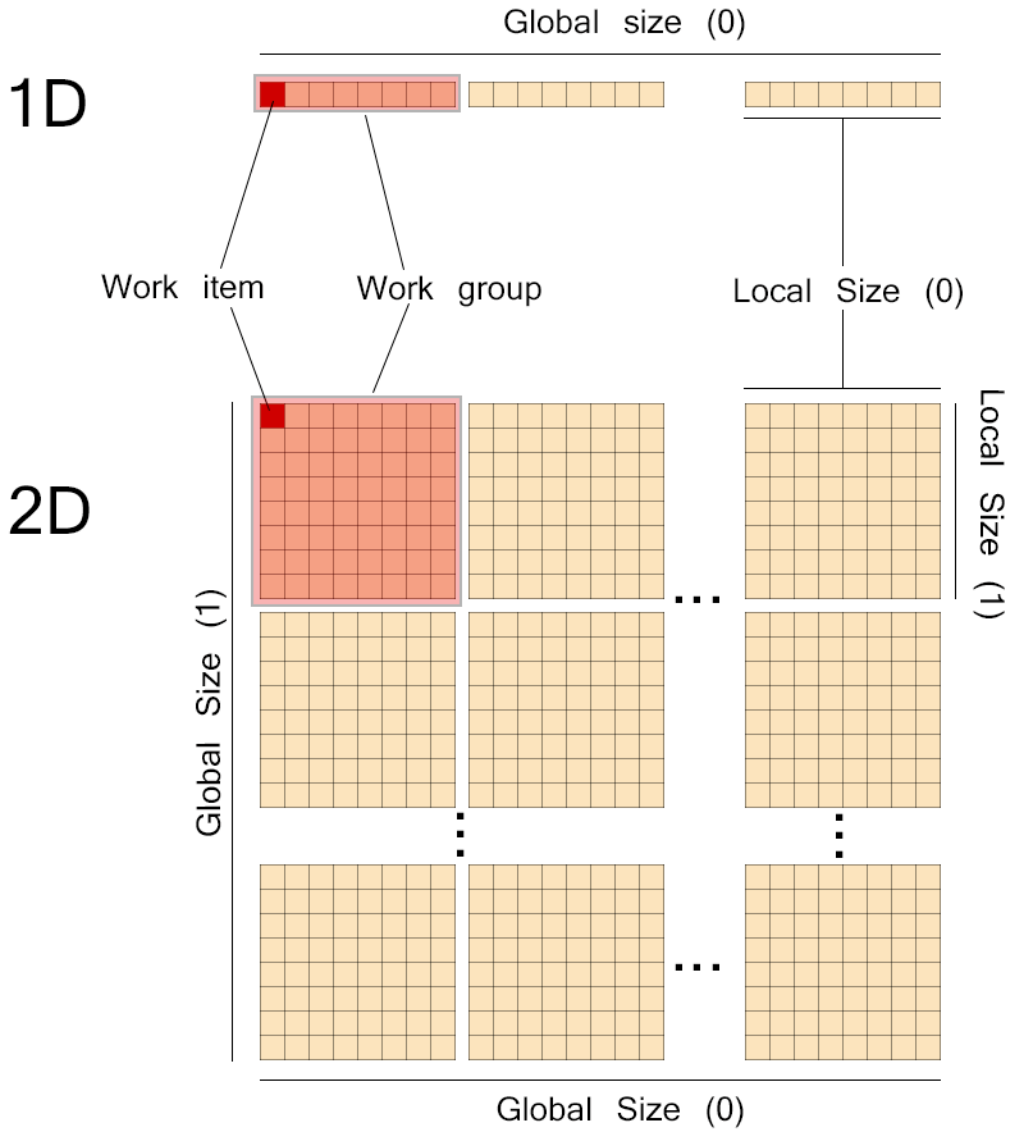


*Figura 6.1 – Esquema OpenCL para una y dos dimensiones.*

### *Transpuesta de la matriz $A => A^T$*

Realizar el algoritmo que cumpla:

$$(A^t)_{ij} = A_{ji} \qquad \textit{Ecuación 6.2}$$

Donde $i$ y $j$, son números enteros positivos mayores que 1.

```
1  __kernel void matrixTrans (__global float *matrixA,
2                             int col,
3                             int fil,
4                             __global float* matrixB){
5        int gx = get_global_id(0);
6        int gy = get_global_id(1);
7        matrixB[gx*fil+gy] = matrixA[gy*col+gx];
8  }
```

*Código 6.1 – Transpuesta de una matriz*

La matriz de entrada tenía tantas filas como gradientes hubiera y seis columnas. La matriz de salida será ($6\ x\ nº\ Gradientes$).

### *Multiplicación $A^t * A$*

En esta ocasión el Código 6.2 del *kernel* toma como parámetros los tres vectores con los datos de las matrices A y B y un tercero que se rellenará con la información de C. Las variables `col` y `fil` se corresponden con el número de columnas y filas que tienen las matrices A y B. Recordar que la matriz A es la transpuesta de la matriz B y el valor de las

columnas y de las filas es el mismo, sólo que transpuesto. Al principio se obtienen los índices globales, que indican al sistema en qué posición de las matrices A y B se encuentra los datos que tiene que multiplicar. Después se realiza la multiplicación de la suma parcial en el único bucle "*for*" del *kernel*, para posteriormente copiar en el vector del dispositivo el resultado de la multiplicación.

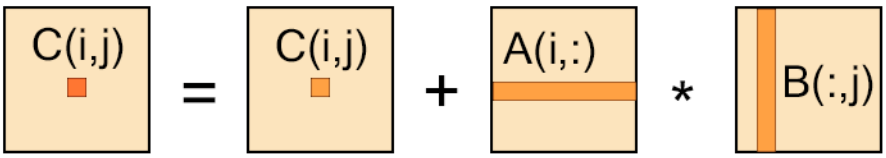


*Figura 6.2 – Multiplicación de matrices*

Esta función implementa la Ecuación 6.3:

$$C_{i,j} = C_{i,j} + \sum_{k=0}^{r} A_{i,k} * B_{k,j} \qquad \textit{Ecuación 6.3}$$

$$0 \leq i \leq N$$

$$0 \leq j \leq M$$

```
1  __kernel void matrixMul (__global float *matrixA,
2                           __global float* matrixB,
3                           int col,
4                           int fil,
5                           __global float* matrixC){
6      int gx = get_global_id(0);
```

```
7       int gy = get_global_id(1);
8       float sum=0;
9       for(int k=0; k<fil; k++){
10          sum+=(matrixA[gy*fil+k]*matrixB[k*col+gx]);
11      }
12      matrixC[gy*col+gx] = sum;
13  }
```

*Código 6.2 – Multiplicación dos matrices*

*Multiplicación* $(A^T * A)^{-1} * A^T$

El código generado para este último *kernel*, es similar al segundo, ya que corresponde a una multiplicación de matrices. Al estar el Código 6.2 explicado y ser tan semejantes, no se va a reparar más en él.

```
1   __kernel void matrixMul (__global float *matrixA,
2                            __global float* matrixB,
3                            int col,
4                            int fil,
5                            __global float* matrixC){
6       int gx = get_global_id(0);
7       int gy = get_global_id(1);
8       float sum=0;
9       for(int k=0; k<col; k++){
```

```
10            sum+=(matrixA[gy*col+k]*matrixB[k*fil+gx]);
11        }
12        matrixC[gy*fil+gx] = sum;
13    }
```

*Código 6.3 – Multiplicación dos matrices*

## 6.4. MULTIPLICAR DATOS

Después de haber detallado la técnica de mínimos cuadrados y posteriormente haberla configurado para beneficiarse de las capacidades de cómputo de una GPU, es momento de analizar los datos de la resonancia magnética.

Estos datos se obtienen en el programa principal, y se almacenan en un vector que se pasa por parámetro a la función actual.

Para la preparación se utilizan las mismas funciones que se han descrito en el apartado 6.3.1. Con la diferencia de que en esta función solo se va a llamar a un *kernel*. Por tanto solo se va a crear un *kernel* con sus correspondientes *buffer* y argumentos, se lanzará y se almacenará el resultado.

### 6.4.1.Generación de código del host

*Multiplicación* $[(A^T * A)^{-1} * A^T] * D$

- *Buffer*:
  - Matriz multiplicada 2, resuelta previamente $(A^T * A)^{-1} * A^T$, actúa de entrada. Al ser una función diferente hay que volver a crear el buffer no se puede reutilizar el anterior.
  - Matriz D, extraída del archivo que contiene los datos, actúa de entrada.

  - Matriz S solución, es la multiplicación resuelta de las matrices anteriores. Actúa de salida, etiqueta: "solo escritura".
- *Kernel*:
  - Se crea un *kernel* asociado al programa creado a partir de un archivo fuente. Este archivo dispone de un solo *kernel*.
- Argumentos:
  - *Buffer* de la matriz multiplicada 2.
  - *Buffer* de la matriz D.
  - Número que da de multiplicar el alto por el ancho de cada *slice,* por el número de *slices*. Este número es el correspondiente a un bloque de datos al que se le habrá aplicado un gradiente.
  - Número de gradientes, depende de archivos.
  - *Buffer* de la matriz S solución. Para almacenar los datos.
- Ejecutar *kernel*:
  - `Global_work_size`: Una dimensión. Número que da de multiplicar el alto y el ancho de cada *slice,* por el número de *slices*.
- Guardar resultados:
  - Se almacena la matriz S solución. Se devuelve a la función principal y se guarda en un archivo con la misma extensión que el que se ha recibido.

### 6.4.2.Generación de código OpenCL

#### *Multiplicación* $[(A^T * A)^{-1} * A^T] * D$

El algoritmo que se realiza no es más que una multiplicación de matrices. La diferencia que si se quieren obtener buenos datos hay que paralelizarlo, para ello se emplean las herramientas de OpenCL. En el cuerpo de la función se declaran una serie de variables de tipo entero inicializadas con el valor de `get_group_id(0)`, `get_global_id(0)` y `get_local_id(0)` (Líneas de la 6 a la13 del Código 6.4). Esto hará que en cada instancia de ejecución del *kernel*, la variable tome el valor del identificador global o local de la instancia. A lo largo de la ejecución esta expresión de OpenCL devolverá todos los valores del espacio de iteraciones. El parámetro de bloque de las líneas anteriores, sirve para indicar de qué tamaño es cada bloque, ya que el *kernel* utiliza *work-groups*.

```
const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
CLK_FILTER_NEAREST;

__kernel void multDatos(__global float *matrixA,

 __read_only image3d_t matrixB,

                        int g,

                        __global float* matrixC){

    int bx = get_group_id(0);

    int by = get_group_id(1)

    int gx = get_global_id(0);

    int gy = get_global_id(1);

    int gz = get_global_id(2);

    int lx = get_local_id(0);

    int ly = get_local_id(1);

    int lz = get_local_id(2);

    float4 coord;

    int aBegin = lx * 6 * by;

    int aEnd = aBegin + lx - 1;

    int aStep = 6;

    int bBegin = 6 * bx;

    float Csub = 0;
```

```
20      for (int a = aBegin, b = bBegin;
21           a <= aEnd;
22           a += aStep, b += aStep){
23          __local float As[16][16];
24          __local float Bs[16][16];
25          coord.x=gy;
26          coord.y=gz;
27          float4 voxel=read_imagef(matrixB,sampler,coord);
28          As[gx][lx] = matrixA[a + gx * ly + lx];
29          coord.z==lx;
30          Bs[lx][1] = voxel.x;
31          barrier(CLK_LOCAL_MEM_FENCE);
32          Csub += As[gx][lx] * Bs[lx][1];
33          barrier(CLK_LOCAL_MEM_FENCE);
34      }
35      int c = 1 * 6 * by + 6 * bx;
36      matrixC[gx-c] = Csub
37  }
```

*Código 6.4 – Multiplicación matriz Datos*

Este *kernel* crea dos *arrays* en la memoria local del dispositivo que se rellenan con la información de las matrices de la memoria global. Hay que destacar que la memoria local

del dispositivo es más rápida que la memoria del host, ya que el *kernel* no tiene que pedir la información a la memoria del host, después transferirla a la memoria global/constante del dispositivo y de ahí transferir la información a la memoria local del dispositivo. Si se realiza de una sola vez es más rápido que realizando este proceso para cada dato que se necesite.

Otro punto a destacar es el uso de las barreras (Línea 31 y 33 del Código 6.4) para realizar la sincronización de los *work-groups*, de lo contrario no se tendría la certeza de que todos los hilos han terminado su ejecución antes de pedir cargar en memoria local un nuevo bloque de submatrices.

## 6.5. INVERSA DE LA MATRIZ

Función que realiza la inversa de una matriz. En un momento del programa principal y del desarrollo del algoritmo se necesita hacer una inversa.

El método que se ha empleado es el de Gauss – Jordan, se basa en una triangularización superior y luego otra inferior de la matriz a la cual se quiere calcular la inversa.

La matriz de entrada tiene que ser cuadrada, y la matriz resultante también resultará una matriz cuadrada.

Esta matriz siempre va a tener las mismas dimensiones, ($6x6$), de tal forma que no son tamaños desmesurados, ni operaciones en las que se note la potencia de la tarjeta gráfica. De esta manera el código de esta función no se ha implementado en OpenCL.

# 7. RESULTADOS

A lo largo del estudio se han empleado dos métodos distintos para la estimación de parámetros en resonancia magnética de difusión. El primer procedimiento, y objeto de este estudio ha sido la ejecución sobre GPU. El segundo, que ha aportado la comparativa con los resultados actuales y los que se pueden obtener después de este proyecto ha sido la ejecución sobre CPU.

En este capítulo se presentan los resultados obtenidos en las pruebas. En primer lugar se muestra el entorno de pruebas usado. A continuación se muestran los resultados obtenidos para las distintas aplicaciones analizadas en los distintos entornos de pruebas utilizados.

## 7.1. ENTORNO DE PRUEBAS

A continuación se describe el entorno que se ha utilizado para la ejecución de las pruebas de rendimiento de los diferentes procedimientos. En la Tabla 7.1 se muestran las principales características de los equipos de pruebas y en la Tabla 7.2 se muestra información detallada sobre la GPU empleada.

Para más información sobre el sistema, consultar las secciones de A.A.1 donde se encuentra la información de entorno de OpenCL y la sección A.A.2 en el cual se ubica la información del procesador.

| | |
|---|---|
| **Sistema operativo** | Scientific Linux reléase 7.1 (Nitrogen) |
| **Compilador** | gcc versión 4.8.3 |
| **Kernel Drive GPU** | fglrx_pci |
| **CPU** | Xeon E3 - 1200 |
| **GPU** | Radeon R9 290X |

*Tabla 7.1 – Características del sistema de pruebas*

| | |
|---|---|
| **Marca** | Asus |
| **Series** | 90YV0551-U0NA00 |
| **Peso del producto** | 998g |
| **Dimensiones del producto** | 3,6 x 27,7 x 10,9 cm |
| **Máxima resolución de pantalla** | 2560x1600 |
| **Velocidad del procesador** | 1 GHz |
| **Capacidad del disco duro** | 4 GB |
| **Coprocesador gráfico** | Radeon R9 290X |
| **Ancho de banda de memoria** | 320 GB/s |
| **Tamaño de la memoria** | GDDR5 de hasta 8 GB |
| **Tipo de memoria gráfica** | PC2-4200 |
| **Interfaz de la tarjeta gráfica** | PCI-Express 3.0 x16 |
| **Número de puertos HDMI** | 1 |

| | |
|---|---|
| **Arquitectura GPU** | 28 nm |
| **Stream Processors** | 2816 unit |

*Tabla 7.2 – Características de la GPU [3]*

## 7.2. ANÁLISIS DE LOS RESULTADOS

Una de las facilidades que tiene OpenCL es la posibilidad de buscar dispositivos tipo CPU o GPU por separado, además de hacer correr el programa en cualquiera de las dos opciones.

Por ende, el programa que estaba preparado para trabajar en la GPU, con escasas alteraciones se ha dispuesto para trabajar en la CPU. Esta característica intrínseca de OpenCL hace que la comparativa de tiempos sea más sencilla y más equivalente. Sencilla porque se han realizado pocos cambios para poder obtenerla. Equivalente porque es el mismo código, con las mismas instrucciones y por tanto la ganancia obtenida será atribuida a la capacidad de la GPU y a la paralelización de la aplicación.

Particularmente, en la aplicación que atañe en estos momentos, se han realizado cuatro programas independientes. Estos cuatro programas, a los que se les ha denominado *kernel*, son los afectados si el dispositivo cambia de CPU a GPU o viceversa.

Se encuentran tres de ellos, los secundarios (Ver sección 6.3). El volumen de datos con el que operan es nimio en comparación con los datos de la resonancia magnética, por tanto la ganancia no sería apreciable con tan pocos datos. El más significativo, ya sea por el volumen de datos ya considerable, o por su alta paralelización, se realiza la misma operación a todos y cada uno de los *pixel*, está explicado en el apartado 6.4.

De esta manera, el capítulo actual, se va a centrar en los resultados que se han alcanzado con la GPU Radeon R9 290x del cuarto *kernel*, y así observar las diferencias de una manera más clara.

### 7.2.1.Primera prueba

El primer estudio de la estimación de los paramétros de difusión en resonancia magnética, se va a efectuar en una resonancia pequeña. Las características de la cabecera más destacadas se muestran en la Tabla 7.3.

| | |
|---|---|
| ***Type*** | short |
| ***Dimensión*** | 4 |
| ***Size*** | 192 256 33 7 |
| ***Kinds*** | space space space list |
| ***Endian*** | big |
| ***Encoding*** | raw |

*Tabla 7.3 – Atributos de la prueba 1*

Se han realizado 1.000 ejecuciones (Ver Figura 7.1), para que los datos sean lo más ajustados posibles. Se ha estudiado el promedio de todos, la mediana y los percentiles 5 y 95, tanto de las ejecuciones en la CPU como en la GPU. Los resultados se muestran a continuación en la Tabla 7.4.

| | CPU | GPU |
|---|---|---|
| **Promedio** | 211,75293 | 54,827486 |
| **Mediana** | 211,3805 | 54,933 |
| **Percentil 5** | 206,1768 | 53,96895 |
| **Percentil 95** | 215,9438 | 55,22105 |

*Tabla 7.4 - Resultado prueba 1*

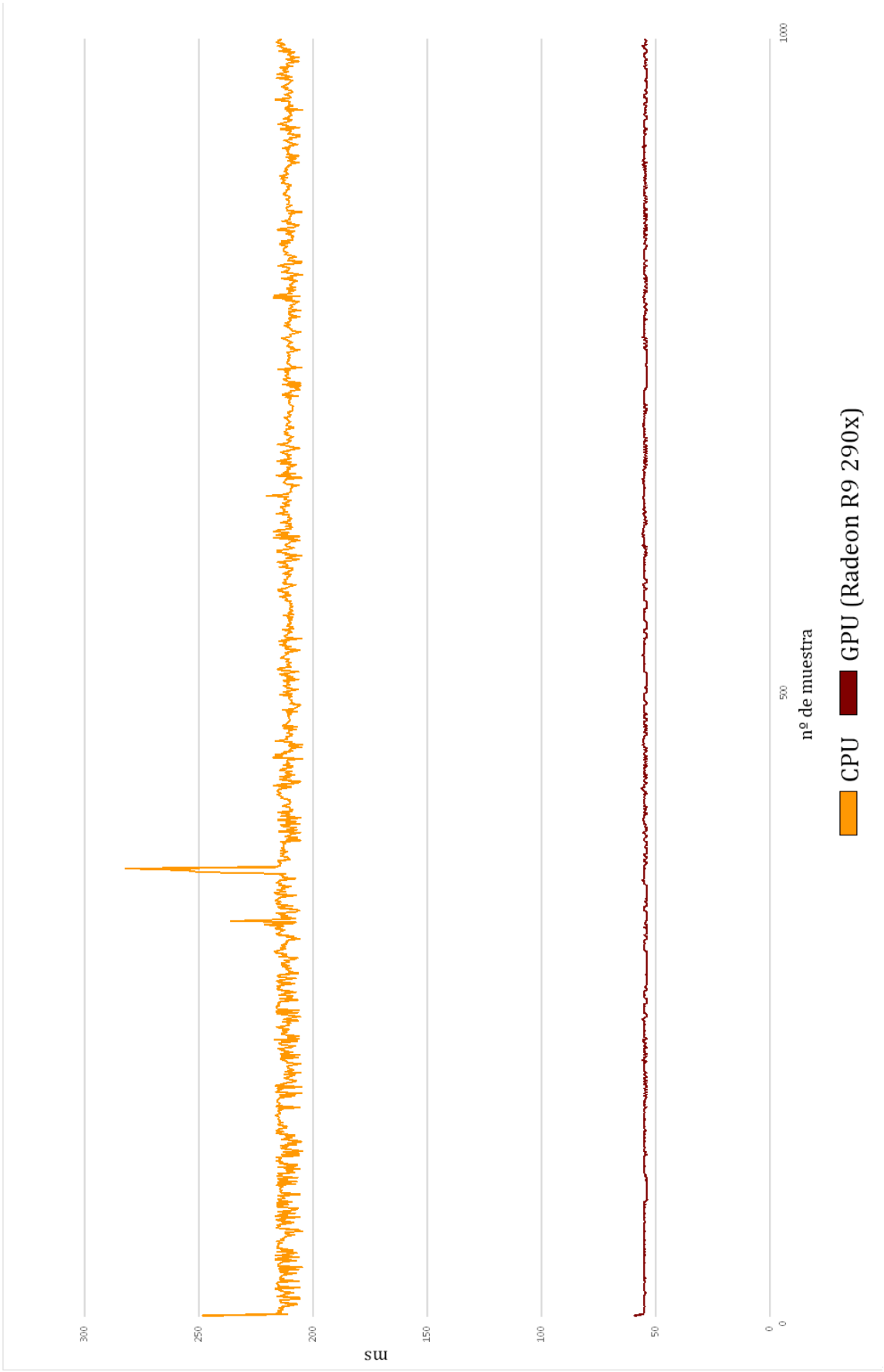


*Figura 7.1 – Resultados prueba 1. 1.000 ejecuciones.*

### 7.2.2.Segunda prueba

El segundo estudio, se trata de una resonancia magnética con las características que aparecen en la Tabla 7.5.

| | |
|---|---|
| ***Type*** | float |
| ***Dimensión*** | 4 |
| ***Size*** | 7 256 256 24 |
| ***Kinds*** | vector space space space |
| ***Encian*** | little |
| ***Encoding*** | raw |

*Tabla 7.5 – Atributos de la prueba 2*

En la Tabla 7.6, se pueden observar los parámetros que se han estudiado.

| | **CPU** | **GPU** |
|---|---|---|
| **Promedio** | 207,624397 | 55,081314 |
| **Mediana** | 207,5495 | 54,9925 |
| **Percentil 5** | 205,5094 | 54,0209 |
| **Percentil 95** | 209,58005 | 55,28305 |

*Tabla 7.6 - Resultado prueba 2*

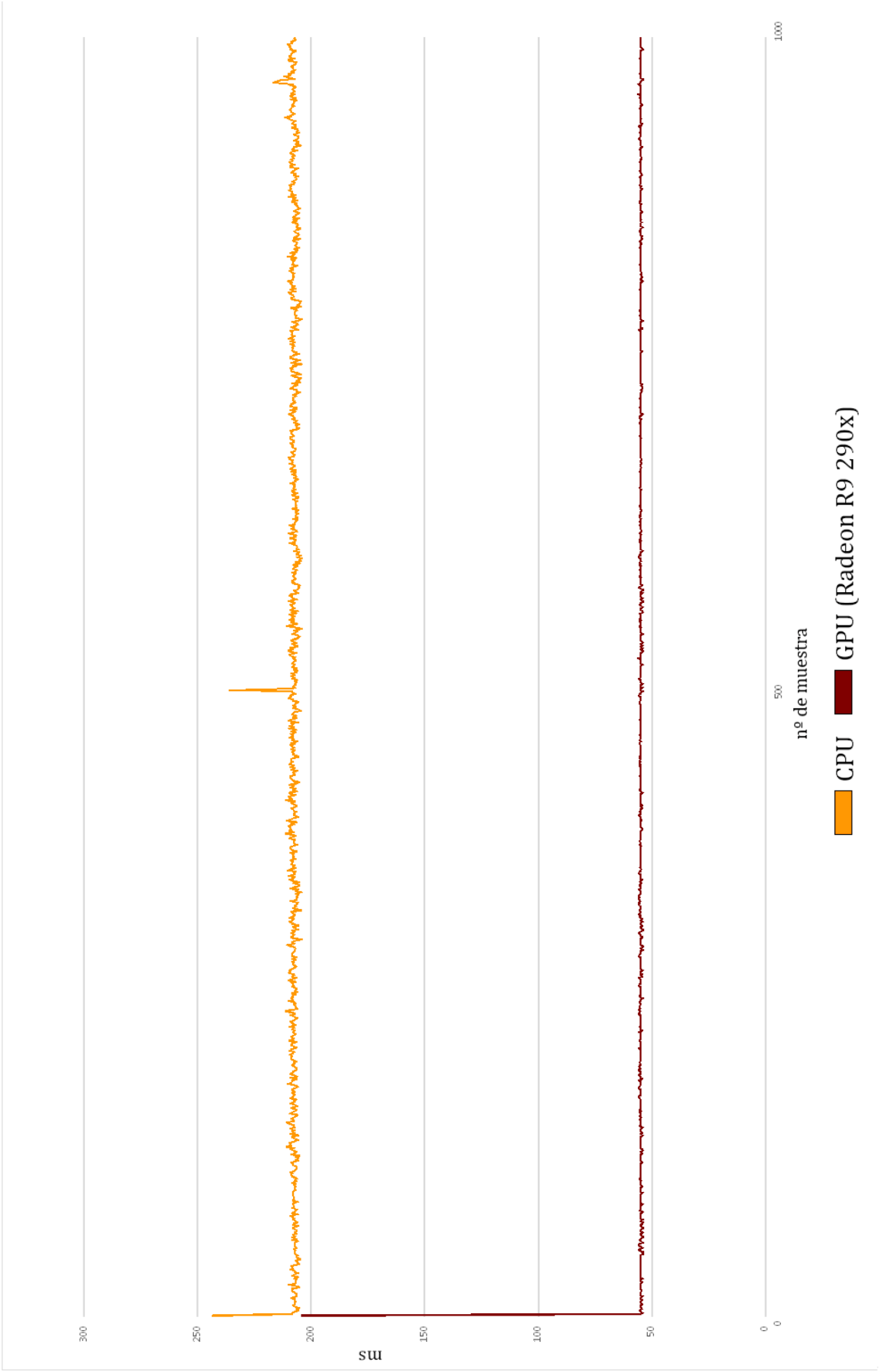


*Figura 7.2 – Resultados prueba 2. 1.000 ejecuciones.*

### 7.2.3.Tercera prueba

Por último la tercera prueba, contiene un volumen de datos más habitual. En la Tabla 7.7 se encuentran los datos más relevantes de una manera más precisa.

| | |
|---|---|
| ***Type*** | short |
| ***Dimensión*** | 4 |
| ***Size*** | 256 256 81 59 |
| ***Kinds*** | space space space list |
| ***Encian*** | Little |
| ***Encoding*** | gzip |

*Tabla 7.7 - Atributos de la prueba 3*

La siguiente tabla (Tabla 7.8) muestra un compendio de los resultados obtenidos.

| | **CPU** | **GPU** |
|---|---|---|
| **Promedio** | 6008,63239 | 578,182791 |
| **Mediana** | 6003,685 | 579,932 |
| **Percentil 5** | 5964,879 | 569,98905 |
| **Percentil 95** | 6046,8855 | 580,87605 |

*Tabla 7.8 - Resultado prueba 3*

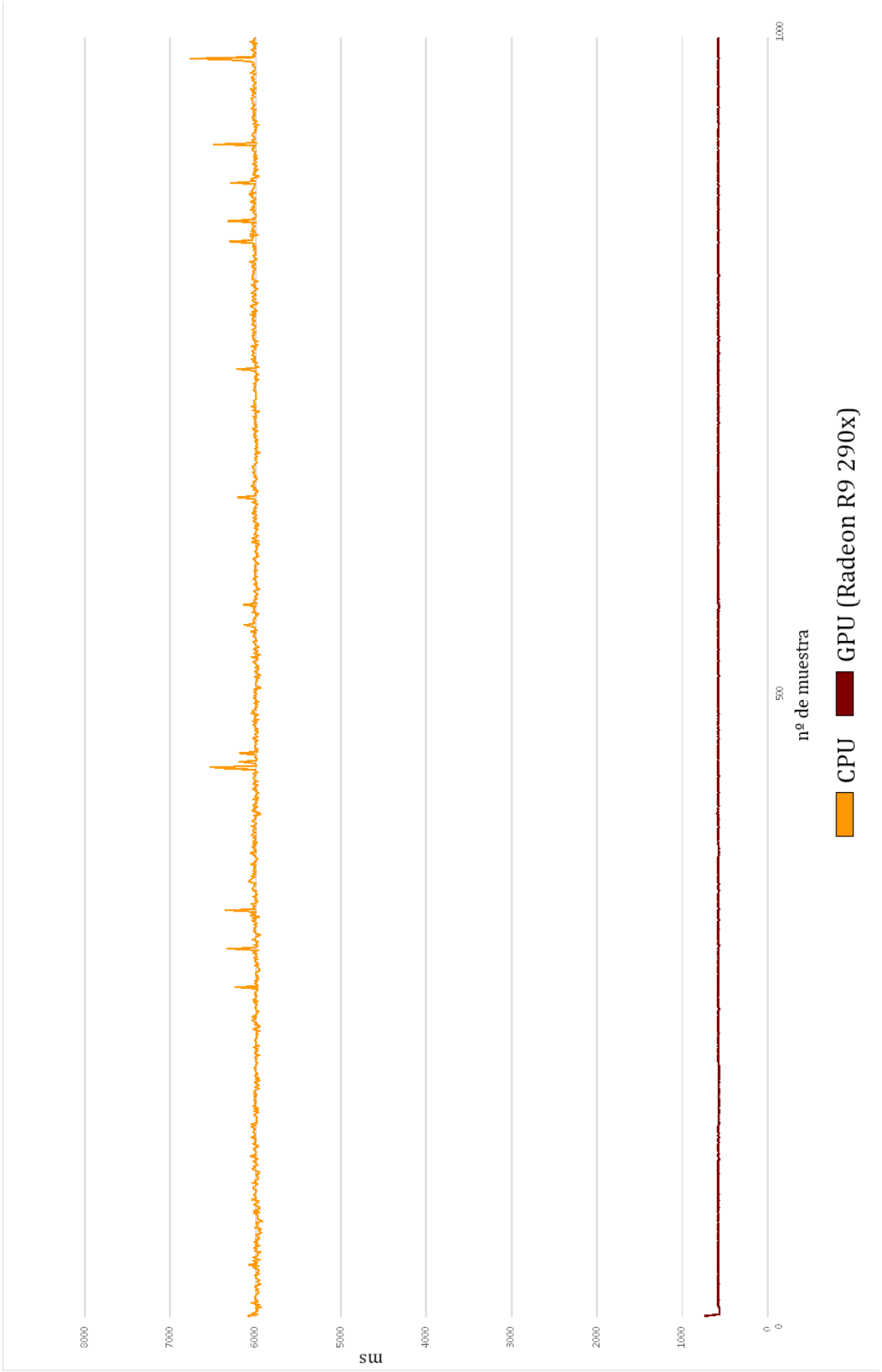


*Figura 7.3 – Resultados prueba 3. 1.000 ejecuciones.*

## 7.3. COMPARATIVA DE RENDIMIENTO

Como se puede ver en el apartado 7.2, las ejecuciones realizadas en la GPU son considerablemente más rápidas. Esto es debido a que pese a ahorrarse las transferencias de los datos con el dispositivo, la CPU no es capaz de explotar el paralelismo que presentan las tareas de manera eficiente en todos los núcleos disponibles del sistema, tal y como lo hacen las GPU.

Para comprobar cuanto más rápida es la GPU, se han relacionado de cada prueba los tiempos, medidos en milisegundos, promedio y la mediana de las mil ejecuciones. Los resultados de cada prueba se muestran en la siguiente tabla (Tabla 7.9)

| | **Prueba 1** | **Prueba 2** | **Prueba 3** |
|---|---|---|---|
| **Ganancia Promedio (CPU/GPU)** | 3,8717231 | 3,76941619 | 10,3922712 |
| **Ganancia Mediana (CPU/GPU)** | 3,847969 | 3,77414193 | 10,3523948 |

*Tabla 7.9 – Ganancias de las pruebas 1 2 y 3.*

La estimación de parámetros de resonancia magnética es un ejercicio largo a causa de la gran cantidad de datos que hay que procesar. A medida que el volumen de datos que se procesa en la aplicación el tiempo que se emplea es mayor. De esta manera se puede comprobar que aunque el proceso se alargue, no se prolonga lo mismo en la CPU como en la GPU. El volumen de datos es elevado y la relación que existe entre las ejecuciones en la CPU y en la GPU se hace más notable.

# 8. CONCLUSIÓN

Una vez realizado el proyecto y teniendo una visión global de OpenCL y del proceso de desarrollo del algoritmo para este lenguaje, se puede observar que, si bien la ejecución del código, y los resultados son ciertamente rápidos, los problemas que se encuentran al programar una GPU pueden suponer un obstáculo y en ocasiones una gran limitación.

A la hora de programar la aplicación, se advierte que este lenguaje no dispone de herramientas sencillas que permitan acceder o comunicar lo que ocurre en el momento de ejecutar el *kernel*. Esto hace que para realizar un proceso de *debugging* se tengan que desarrollar técnicas poco sofisticadas. Una de las primeras conclusiones de este proyecto es la **necesidad de mejorar el entorno de desarrollo de OpenCL**. Si bien es cierto, que fuera del *kernel*, se encuentran funciones que alertan del error en ese instante, y finalizan la ejecución para su restablecimiento.

Se han encontrado diferencias de ejecución entre la aplicación en el dispositivo portátil y en el ordenador de mesa, de tal manera que otra de las conclusiones existentes es la **no homogeneidad de las implementaciones de OpenCL**. Una de las grandes finalidades de este proyecto es exportarlo para su utilización a diferentes dispositivos, y el hecho de tener esa diversidad dificulta el posterior trabajo.

Como contrapartida otra de las grandes conclusiones deducidas a partir de este proyecto es que **OpenCL proporciona un incremento en el rendimiento de aplicaciones paralelizables**. Como ya se ha explicado en ocasiones anteriores, dependiendo del algoritmo la utilización de GPGPU será más o menos favorable, de tal manera que si el algoritmo que se quiere mejorar en tiempo, presenta un método de paralelización sencillo, la implementación a OpenCL puede aportar un aumento de rendimiento considerable.

Las tarjetas gráficas tendrán más relevancia gracias a los resultados que pueden ofrecer, todo ello gracias a los nuevos procesadores que van aumentando el número de cores y el cambio en el modelo de programación. Este cambio de momento está regido por la complejidad de los lenguajes actuales.

Así que después de conocer más a fondo este lenguaje y la programación con GPU en general, como opinión personal veo cada vez **más futuro en estas tecnologías y en los dispositivos que las acompañan**.

# 9. LÍNEAS FUTURAS

Dado que este proyecto se ha centrado en demostrar la mejora que ofrece la paralelización de determinados algoritmos secuenciales, las líneas de desarrollo se centran en una amplia experimentación con la GPU de la que se dispone, así como con las que están por venir.

Una de las vías que se puede seguir dado este trabajo es profundizar en la optimización del *kernel* de OpenCL. Conociendo ya el lenguaje, se puede enfocar el algoritmo de otra manera y así profundizar más en mecanismos de optimización que permitieran que el código se ejecutara de forma más eficiente en las plataformas disponibles. Este proyecto se ha pensado para que se pueda ejecutar en varias plataformas, pero teniendo en cuenta el *hardware* donde se ejecutaría se podría realizar alguna modificación estructural sobre el *kernel* y de este modo conseguir explotar de forma más completa todos los recursos disponibles. Además con la evolución que se prevé en este ámbito, el rendimiento de las futuras generaciones de tarjetas gráficas será destacable, propendiendo a favorecer la resolución de problemas de propósito general.

Otro de los cambios que se podría realizar es la implementación de un sistema de transferencia de memoria asíncrono entre la CPU y la GPU. En este caso se ve como el cuello de botella se encuentra en las transferencias de memoria entre los dispositivos, ya que el uso de memoria no se puede reducir más. Esta mejora precisa adaptar todo el código a un sistema de semáforos entre transferencias, para que pueda tener la información de la siguiente iteración en memoria. Es una de las vías que se pueden desarrollar en futuras versiones del código.

Después del tiempo de aprendizaje de OpenCL, y del desarrollo de la aplicación, se perciben fallos de implementación. Estos fallos están ocasionados por el desconocimiento parcial, en un primer momento, de este lenguaje y una idea principal desacertada. Por tanto,

aunque la idea sea apropiada, útil, beneficiosa y rentable, una comprensión superior del funcionamiento de OpenCL a la hora de empezar a programar, haría que cualquier mejora dentro del código fuera posible.

# 10. BIBLIOGRAFÍA

[1] D. M. y R. G., «Imágenes de tractografía por tensor de difusión: ¿avance científico, herramienta clínica o sólo una figura bonita?,» *Neurology* , nº 68, pp. 9-10, 2007.

[2] C. Westin, S. Maier, H. Mamata, A. Nabavi, F. Jolesz y R. Kikinis, «Processing and visualization for diffusion tensor MRI,» *Medical Image Analysis,* nº 6, pp. 93-108, 2002.

[3] A. M. D. Inc., «AMD. Enabling Today. Inspiring tomorrow.,» [En línea]. Available: http://www.amd.com/en-us. [Último acceso: Agosto 2015].

[4] Nvidia Corporation, «Nvidia,» [En línea]. Available: www.nvidia.es/. [Último acceso: Julio 2015].

[5] Khronos, Grupo, «AMD. Developer Central.,» [En línea]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/. [Último acceso: Agosto 2015].

[6] Nvidia Corporation, «Nvidia. Procesamiento paralelo CUDA,» [En línea]. Available: http://www.nvidia.es/object/cuda-parallel-computing-es.html. [Último acceso: Julio 2015].

[7] A. Munshi, «The OpenCL Specification. Khronos OpenCL Working Group Version 1.2,» 2011.

[8] J. E. T. a. E. O. Stejskal, «Spin Diffusion Measurements: Spin Echoes in the Presence of a Time Dependent Field Gradient,» *The Journal of Chemical Physics,* vol. 42, nº 1, pp. 288-292, Enero 1965.

[9] R. C. Almeida, E. M. Moreno, A. T. Vega y M. M. Fernández, «Una herramienta para el Procesado y Visualización de Imágenes de Resonancia Magnética de Tensor de Difusión,» de *XXVI Congreso anual de la Sociendad Española de Ingeniería Biomédica*, Valladolid, 15, 16 y 17 de octubre de 2008.

[10] Z. P. Liang y P. C. Lauterbur, Principles of Magnetic Resonance Imaging a Signal Processing Perspective, New York, 2000. (ISBN: 978-0-7803-4723-6).

[11] J. E. Herrerías Rey, Hardware y componentes, Madrid, 2006. (ISBN: 84-415-1979-X).

[12] J. M. M. Pozuelo, Hardware microinformático: viaje a las profundidades del PC, Madrid: RA-MA , 2005. (ISBN: 84-7897-661-2).

[13] W. d. A. 3Dfx., «3Dfx,» 18 Octubre 2000. [En línea]. Available: http://web.archive.org/web/20001018102838/http:/www.3dfx.com/. [Último acceso: Julio 2015].

[14] Microsoft, «Microsoft,» [En línea]. Available: https://www.microsoft.com/es-es/. [Último acceso: Julio 2015].

[15] S. Telecom, «DirectX,» [En línea]. Available: http://www.directx.com.es/. [Último acceso: Julio 2015].

[16] S. Davis, «AMD. TressFX Hair: Cross-platform and v2.0,» 12 mayo 2015. [En línea]. Available: https://community.amd.com/community/gaming/blog/2015/05/12/tressfx-hair-cross-platform-and-v20. [Último acceso: Julio 2015].

[17] Nvidia Corporation, «Nvidia HairWorks,» 18 Mayo 2015. [En línea]. Available: https://developer.nvidia.com/hairworks. [Último acceso: Julio 2015].

[18] W. Stallings, Organización y arquitectura de computadores, Madrid, 2005. (ISBN: 978-84-8966-082-3).

[19] I. M. J. Cumbreras, Fundamentos del Hardware, Madrid, 2013. (ISBN: 978-84-1545-260-7).

[20] A. G. Higuera, El control automático en la industria, Cuenca: Universidad de Castilla-La Mancha , 2005. (ISBN: 84-8427-405-5).

[21] C. González Morcillo, J. A. Albusac Jiménez, S. Pérez Camacho, J. López González y C. Mora Castro, Desarrollo de Videojuegos: Programación Gráfica, Ciudad Real: Universidad de Castilla-La Mancha, 2012. (ISBN: 978-84-686-1958-0).

[22] Nvidia Developer, «Nvidia Documentation,» [En línea]. Available: http://docs.nvidia.com/. [Último acceso: Julio 2015].

[23] M. D. Network, «Asm Shader Reference,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/bb219840(v=vs.85).aspx. [Último acceso: Julio 2015].

[24] M. Ujaldón Martínez, Procesadores gráficos para PC, Málaga: Editorial Ciencia-3, S.L., 2005. (ISBN: 84-95391-09-0).

[25] Khronos Groups, «OpenGL. The Industry's Foundation for High Performance Graphics,» Agosto 2015. [En línea]. Available: https://www.opengl.org/. [Último acceso: Agosto 2015].

[26] J. D. Foley, A. v. Dam, S. K. Feiner y J. F. Hughes, Computer Graphics, principles and practice, Cornell University, 1996. (ISBN: 0-321-21056-5).

[27] Nvidia Corporation, GPU Gems 3, United Stated, 2007. (ISBN: 978-0-321-51526-1).

[28] J. Stokes, «SIMD architectures,» *arsTechnica,* 22 Marzo 2000.

[29] Ó. A. C. Lizardo, «Secuencias funcionales en resonancia magnética, difusión, DTI, espectroscopia,» *Arch Neurocien (Mex),* vol. 14, nº 1, pp. 58-68, 2009.

[30] A. Einstein, Investigation on the Theory of the Brownian Movement, Dover, 1905.

[31] P. Hagmann, L. Jonasson, P. Maeder, J.-P. Thiran, V. J. Wedeen y R. Meuli, «Understanding Diffusion MR Imaging Techniques: From Scalar Diffusion-weighted Imaging to Diffusion Tensor Imaging and Beyond,» *RadioGraphics,* vol. 26, pp. S205-S209, 2006.

[32] A. Duque, E. Roa y J. Castedo, «Anatomía de la sustancia blanca mediante tractografía por tensor de difusión,» *Radiología,* vol. 50, nº 2, pp. 99-111, Marzo 2008.

[33] G. B. Arranz, S. A. Fernández y M. M. Fernández, «Estudio de los efectos de los parámetros de adquisición en la tractografía global,» de *XXX Congreso Anual de la Sociedad Española de Ingieneria Biomédica - CASEIB*, San Sebastián, 2012.

[34] G. E. Christensen y M. Sonka, «Information Processing in Medical Imaging,» de *19th International Conference, IPMI 2005*, Glenwood Springs, CO, USA, julio 2005.

[35] S. A. Fernández y R. d. L. García, «Capítulo 3: Imágenes Médicas,» de *Procesado de Imagen Médica (Master Universitario de Investigación en Tecnologías de la Información y las Comunicaciones)*, Valladolid, 2015, pp. 51-96.

[36] P. E. Grant, R. G. Gonzalez y P. W. Schaefer, «Diffusion-weighted MR Imaging of the brain,» *Radiology,* vol. 217, pp. 331-345, 2000.

[37] J. P. Thiran, J. Jonasson y P. Hagmann, «DTI mapping of human brain connectivity: statiscal fibre tracking and virtual dissection,» *NeuroImage,* vol. 19, pp. 545-554, 2003.

[38] J. Cisternas, T. Asahi, M. Gálvez, G. Rojas y E. Bravo, «Desarrollo y puesta en marcha de software de tractografía,» *Revista Chilena de Radiología,* vol. 14, nº 1, pp. 31-35, 2008.

[39] AMD, «AMD. Developer Central,» [En línea]. Available: http://developer.amd.com/tools-and-sdks/archive/legacy-cpu-gpu-tools/. [Último acceso: Agosto 2015].

[40] Intel, «Intel. Developer Zone. Larrabee,» 20 Abril 2011. [En línea]. Available: https://software.intel.com/en-us/articles/larrabee/. [Último acceso: Agosto 2015].

[41] M. P. García y J. M. Zamarreño, «Optimización multiarranque en paralelo sobre GPU,» Valladolid, Septiembre 2012.

[42] AMD, «OpenCL Optimization Case Study Fast Fourier Transform - Part I,» [En línea]. Available: http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-fast-fourier-transform-part-1/. [Último acceso: Agosto 2015].

[43] AMD, «OpenCL Optimization Case Study Fast Fourier Transform - Part II,» [En línea]. Available: http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-fast-fourier-transform-part-ii/. [Último acceso: Agosto 2015].

[44] K. Karimi, N. G. Dickson y F. Hamze, «A Performance Comparison of CUDA and OpenCL,» Canada.

[45] J. Fang, «A Comprehensive Performance Comparison of CUDA and OpenCL,» de *International Conference on Parallel Processing* , Taipei City (ISSN: 0190-3918), 13 -16 Septiembre .

[46] J. L. d. l. F. O'connor, Técnicas de cálculo para Sistemas de Ecuaciones, Programación Lineal y Programación Entera, Barcelona: Editorial Reverté, S.A., 1998. (ISBN: 84-291-2606-6).

[47] J. G. Cabello, Álgebra Lineal. Sus Aplicaciones en Economía, Ingenierias y otras Ciencias., Madrid, 2006. (ISBN: 84-96477-12-6).

[48] Microsoft, «Tipos fundamentales,» [En línea]. Available: https://msdn.microsoft.com/es-es/library/cc953fe1.aspx. [Último acceso: Agosto 2015].

[49] Microsoft, «Explicación de las arquitecturas Big Endian y Little Endian,» [En línea]. Available: https://support.microsoft.com/es-es/kb/102025. [Último acceso: Agosto 2015].

# APÉNDICES

# APÉNDICE A: INFORMACIÓN DEL SISTEMA

## A.1. RESULTADOS DE LA EJECUCIÓN DE CLINFO

CLInfo es una utilidad distribuida con el SDK de AMD que permite obtener información detallada sobre el entorno OpenCL de un sistema.

```
1  bash-4.2$ clinfo
2  Number of platforms:                 1
3    Platform Profile:                  FULL_PROFILE
4    Platform Version:                  OpenCL2.0AMD-APP(1702.3)
5    Platform Name:
       AMDAcceleratedParallelProcessing
6    Platform Vendor:                   AdvancedMicroDevices,Inc.
7    Platform Extensions:
       cl_khr_icd
       cl_amd_event_callback
       cl_amd_offline_devices
8    Platform Name:
```

```
      AMDAcceleratedParallelProcessing
Number of devices:                2
  Device Type:                    CL_DEVICE_TYPE_GPU
  Vendor ID:                      1002h
  Board name:
  Device Topology:                PCI[B#1,D#0,F#0]
  Max compute units:              44
  Max work items dimensions:      3
   Max work items[0]:             256
   Max work items[1]:             256
   Max work items[2]:             256
  Max work group size:            256
  Preferred vector width char:    4
  Preferred vector width short:   2
  Preferred vector width int:     1
  Preferred vector width long:    1
  Preferred vector width float:   1
  Preferred vector width double:  1
  Native vector width char:       4
  Native vector width short:      2
  Native vector width int:        1
  Native vector width long:       1
  Native vector width float:      1
  Native vector width double:     1
  Max clock frequency:            1030Mhz
  Address bits:                   64
  Max memory allocation:          3008888832
  Image support:                  Yes
  Max number of images
read arguments:                   128
  Max number of images
write arguments:                  64
  Max image 2D width:             16384
  Max image 2D height:            16384
  Max image 3D width:             2048
  Max image 3D height:            2048
  Max image 3D depth:             2048
  Max samplers within kernel:     16
  Max size of kernel argument:    1024
  Alignment (bits) of base
address:                          2048
  Minimum alignment (bytes)
for any datatype:                 128
  Single precision floating
point capability
   Denorms:                       No
```

```
    Quiet NaNs:                     Yes
    Round to nearest even:          Yes
    Round to zero:                  Yes
    Round to +ve and infinity:      Yes
    IEEE754-2008 fused
multiply-add:                       Yes
  Cache type:                       Read/Write
  Cache line size:                  64
  Cache size:                       16384
  Global memory size:               4250927104
  Constant buffer size:             65536
  Max number of constant args:      8
  Local memory type:                Scratchpad
  Local memory size:                32768
  Kernel Preferred work group
size multiple:                      64
  Error correction support:         0
  Unified memory for Host and
Device:                             0
  Profiling timer resolution:       1
  Device endianess:                 Little
  Available:                        Yes
  Compiler available:               Yes
  Execution capabilities:
    Execute OpenCL kernels:         Yes
    Execute native function:        No
  Queue properties:
    Out-of-Order:                   No
    Profiling :                     Yes
  Platform ID:                      0x00007f5121c4f630
  Name:                             Hawaii
  Vendor:                           AdvancedMicroDevices,Inc.
  Device OpenCL C version:          OpenCLC2.0
  Driver version:                   1702.3(VM)
  Profile:                          FULL_PROFILE
  Version:                          OpenCL2.0AMD-APP(1702.3)
  Extensions:
      cl_khr_fp64
      cl_amd_fp64
      cl_khr_global_int32_base_atomics
      cl_khr_global_int32_extended_atomics
      cl_khr_local_int32_base_atomics
      cl_khr_local_int32_extended_atomics
      cl_khr_int64_base_atomics
      cl_khr_int64_extended_atomics
      cl_khr_3d_image_writes
      cl_khr_byte_addressable_store
      cl_khr_gl_sharing
```

```
      cl_ext_atomic_counters_32
      cl_amd_device_attribute_query
      cl_amd_vec3
      cl_amd_printf
      cl_amd_media_ops
      cl_amd_media_ops2
      cl_amd_popcnt
      cl_khr_image2d_from_buffer
      cl_khr_spir
      cl_khr_subgroups
      cl_khr_gl_event
      cl_khr_depth_images
  Device Type:                    CL_DEVICE_TYPE_CPU
  Vendor ID:                      1002h
  Board name:
  Max compute units:              8
  Max work items dimensions:      3
   Max work items[0]:             1024
   Max work items[1]:             1024
   Max work items[2]:             1024
  Max work group size:            1024
  Preferred vector width char:    16
  Preferred vector width short:   8
  Preferred vector width int:     4
  Preferred vector width long:    2
  Preferred vector width float:   8
  Preferred vector width double:  4
  Native vector width char:       16
  Native vector width short:      8
  Native vector width int:        4
  Native vector width long:       2
  Native vector width float:      8
  Native vector width double:     4
  Max clock frequency:            3999Mhz
  Address bits:                   64
  Max memory allocation:          4174709760
  Image support:                  Yes
  Max number of images
read arguments:                   128
  Max number of images
write arguments:                  64
  Max image 2D width:             8192
  Max image 2D height:            8192
  Max image 3D width:             2048
  Max image 3D height:            2048
  Max image 3D depth:             2048
  Max samplers within kernel:     16
```

```
  Max size of kernel argument:    4096
  Alignment (bits) of base address:                                 1024
  Minimum alignment (bytes) for any datatype:                       128
  Single precision floating point capability
   Denorms:                       Yes
   Quiet NaNs:                    Yes
   Round to nearest even:         Yes
   Round to zero:                 Yes
   Round to +ve and infinity:     Yes
   IEEE754-2008 fused multiply-add:                                 Yes
  Cache type:                     Read/Write
  Cache line size:                64
  Cache size:                     32768
  Global memory size:             16698839040
  Constant buffer size:           65536
  Max number of constant args:    8
  Local memory type:              Global
  Local memory size:              32768
  Kernel Preferred work group size multiple:                        1
  Error correction support:       0
  Unified memory for Host and Device:                               1
  Profiling timer resolution:     1
  Device endianess:               Little
  Available:                      Yes
  Compiler available:             Yes
  Execution capabilities:
   Execute OpenCL kernels:        Yes
   Execute native function:       Yes
  Queue properties:
   Out-of-Order:                  No
   Profiling :                    Yes
  Platform ID:                    0x00007f5121c4f630
  Name:
      Intel(R)Core(TM)i7-4790CPU@3.60GHz
  Vendor:                         GenuineIntel
  Device OpenCL C version:        OpenCLC1.2
  Driver version:                 1702.3(sse2,avx)
  Profile:                        FULL_PROFILE
  Version:                        OpenCL1.2AMD-APP(1702.3)
  Extensions:
```

```
      cl_khr_fp64
      cl_amd_fp64
      cl_khr_global_int32_base_atomics
      cl_khr_global_int32_extended_atomics
      cl_khr_local_int32_base_atomics
      cl_khr_local_int32_extended_atomics
      cl_khr_int64_base_atomics
      cl_khr_int64_extended_atomics
      cl_khr_3d_image_writes
      cl_khr_byte_addressable_store
      cl_khr_gl_sharing
      cl_ext_device_fission
      cl_amd_device_attribute_query
      cl_amd_vec3
      cl_amd_printf
      cl_amd_media_ops
      cl_amd_media_ops2
      cl_amd_popcnt
      cl_khr_spir
      cl_khr_gl_event
```

*Código A.1 - Información del entorno OpenCL*

# A.2. CONTENIDO /PROC/CPUINFO

El fichero /proc/cpuinfo de Linux proporciona información detallada sobre los procesadores presentes en el sistema.

```
1  processor       : 0
2  vendor_id       : GenuineIntel
3  cpu family      : 6
4  model           : 60
5  model name      : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
6  stepping        : 3
7  microcode       : 0x1c
8  cpu MHz         : 1332.421
9  cache size      : 8192 KB
10 physical id     : 0
11 siblings        : 8
```

```
core id           : 0
cpu cores         : 4
apicid            : 0
initial apicid    : 0
fpu               : yes
fpu_exception     : yes
cpuid level       : 13
wp                : yes
flags             : fpu vme de pse tsc msr pae mce cx8 apic
     sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
     fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
     lm constant_tsc arch_perfmon pebs bts rep_good nopl
     xtopology nonstop_tsc aperfmperf eagerfpu pni
     pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
     fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
     popcnt tsc_deadline_timer aes xsave avx f16c rdrand
     lahf_lm abm ida arat xsaveopt pln pts dtherm
     tpr_shadow vnmi flexpriority ept vpid fsgsbase
     tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
bogomips          : 7183.41
clflush size      : 64
cache_alignment   : 64
address sizes     : 39 bits physical, 48 bits virtual
power management :

processor         : 1
vendor_id         : GenuineIntel
cpu family        : 6
model             : 60
model name        : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
stepping          : 3
microcode         : 0x1c
cpu MHz           : 1299.937
cache size        : 8192 KB
physical id       : 0
siblings          : 8
core id           : 1
cpu cores         : 4
apicid            : 2
initial apicid    : 2
fpu               : yes
fpu_exception     : yes
cpuid level       : 13
wp                : yes
flags             : fpu vme de pse tsc msr pae mce cx8 apic
     sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
     fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
```

```
     lm constant_tsc arch_perfmon pebs bts rep_good nopl
     xtopology nonstop_tsc aperfmperf eagerfpu pni
     pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
     fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
     popcnt tsc_deadline_timer aes xsave avx f16c rdrand
     lahf_lm abm ida arat xsaveopt pln pts dtherm
     tpr_shadow vnmi flexpriority ept vpid fsgsbase
     tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
bogomips        : 7183.41
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management :

processor       : 2
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
stepping        : 3
microcode       : 0x1c
cpu MHz         : 1303.031
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 2
cpu cores       : 4
apicid          : 4
initial apicid  : 4
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic
     sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
     fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
     lm constant_tsc arch_perfmon pebs bts rep_good nopl
     xtopology nonstop_tsc aperfmperf eagerfpu pni
     pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
     fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
     popcnt tsc_deadline_timer aes xsave avx f16c rdrand
     lahf_lm abm ida arat xsaveopt pln pts dtherm
     tpr_shadow vnmi flexpriority ept vpid fsgsbase
     tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
bogomips        : 7183.41
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
```

```
77  power management :
78
79  processor        : 3
80  vendor_id        : GenuineIntel
81  cpu family       : 6
82  model            : 60
83  model name       : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
84  stepping         : 3
85  microcode        : 0x1c
86  cpu MHz          : 910.828
87  cache size       : 8192 KB
88  physical id      : 0
89  siblings         : 8
90  core id          : 3
91
92  cpu cores        : 4
93  apicid           : 6
94  initial apicid   : 6
95  fpu              : yes
96  fpu_exception    : yes
97  cpuid level      : 13
98  wp               : yes
99  flags            : fpu vme de pse tsc msr pae mce cx8 apic
        sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
        fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
        lm constant_tsc arch_perfmon pebs bts rep_good nopl
        xtopology nonstop_tsc aperfmperf eagerfpu pni
        pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
        fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
        popcnt tsc_deadline_timer aes xsave avx f16c rdrand
        lahf_lm abm ida arat xsaveopt pln pts dtherm
        tpr_shadow vnmi flexpriority ept vpid fsgsbase
        tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
100 bogomips         : 7183.41
101 clflush size     : 64
102 cache_alignment  : 64
103 address sizes    : 39 bits physical, 48 bits virtual
104 power management :
105
106 processor        : 4
107 vendor_id        : GenuineIntel
108 cpu family       : 6
109 model            : 60
110 model name       : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
111 stepping         : 3
112 microcode        : 0x1c
```

```
cpu MHz           : 814.921
cache size        : 8192 KB
physical id       : 0
siblings          : 8
core id           : 0
cpu cores         : 4
apicid            : 1
initial apicid    : 1
fpu               : yes
fpu_exception     : yes
cpuid level       : 13
wp                : yes
flags             : fpu vme de pse tsc msr pae mce cx8 apic
      sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
      fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
      lm constant_tsc arch_perfmon pebs bts rep_good nopl
      xtopology nonstop_tsc aperfmperf eagerfpu pni
      pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
      fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
      popcnt tsc_deadline_timer aes xsave avx f16c rdrand
      lahf_lm abm ida arat xsaveopt pln pts dtherm
      tpr_shadow vnmi flexpriority ept vpid fsgsbase
      tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
bogomips          : 7183.41
clflush size      : 64
cache_alignment   : 64
address sizes     : 39 bits physical, 48 bits virtual
power management  :
processor         : 5
vendor_id         : GenuineIntel
cpu family        : 6
model             : 60
model name        : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
stepping          : 3
microcode         : 0x1c
cpu MHz           : 953.578
cache size        : 8192 KB
physical id       : 0
siblings          : 8
core id           : 1
cpu cores         : 4
apicid            : 3
initial apicid    : 3
fpu               : yes
fpu_exception     : yes
cpuid level       : 13
```

```
wp               : yes
flags            : fpu vme de pse tsc msr pae mce cx8 apic
     sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
     fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
     lm constant_tsc arch_perfmon pebs bts rep_good nopl
     xtopology nonstop_tsc aperfmperf eagerfpu pni
     pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
     fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
     popcnt tsc_deadline_timer aes xsave avx f16c rdrand
     lahf_lm abm ida arat xsaveopt pln pts dtherm
     tpr_shadow vnmi flexpriority ept vpid fsgsbase
     tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
bogomips         : 7183.41
clflush size     : 64
cache_alignment  : 64
address sizes    : 39 bits physical, 48 bits virtual
power management :

processor        : 6
vendor_id        : GenuineIntel
cpu family       : 6
model            : 60
model name       : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
stepping         : 3
microcode        : 0x1c
cpu MHz          : 970.453
cache size       : 8192 KB
physical id      : 0
siblings         : 8
core id          : 2
cpu cores        : 4
apicid           : 5
initial apicid   : 5
fpu              : yes
fpu_exception    : yes
cpuid level      : 13
wp               : yes
flags            : fpu vme de pse tsc msr pae mce cx8 apic
     sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
     fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
     lm constant_tsc arch_perfmon pebs bts rep_good nopl
     xtopology nonstop_tsc aperfmperf eagerfpu pni
     pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
     fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
     popcnt tsc_deadline_timer aes xsave avx f16c rdrand
     lahf_lm abm ida arat xsaveopt pln pts dtherm
     tpr_shadow vnmi flexpriority ept vpid fsgsbase
     tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
```

```
bogomips        : 7183.41
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management :

processor       : 7
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
stepping        : 3
microcode       : 0x1c
cpu MHz         : 1237.640
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 3
cpu cores       : 4
apicid          : 7
initial apicid  : 7
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic
     sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
     fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
     lm constant_tsc arch_perfmon pebs bts rep_good nopl
     xtopology nonstop_tsc aperfmperf eagerfpu pni
     pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
     fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
     popcnt tsc_deadline_timer aes xsave avx f16c rdrand
     lahf_lm abm ida arat xsaveopt pln pts dtherm
     tpr_shadow vnmi flexpriority ept vpid fsgsbase
     tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
bogomips        : 7183.41
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management :
```

*Código A.2 - Información del procesador*