Universitat Autònoma de Barcelona

PLANIFICACIÓN Y GESTIÓN DE RECURSOS EN LA EJECUCIÓN DE WORKFLOWS CIENTÍFICOS EN ENTORNOS HÍBRIDOS GPGPU

JORDI DELGADO MENGUAL

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIES

DEPARTAMENTO DE ARQUITECTURA DE COMPUTADORES Y SISTEMAS OPERATIVOS

PROGRAMA DE DOCTORADO EN COMPUTACIÓN DE ALTAS PRESTACIONES

PLANIFICACIÓN Y GESTIÓN DE RECURSOS EN LA EJECUCIÓN DE WORKFLOWS CIENTÍFICOS EN ENTORNOS HÍBRIDOS GPGPU

TESIS DOCTORAL DE:

JORDI DELGADO MENGUAL

DIRIGIDA POR:

Dr. JUAN CARLOS MOURE LÓPEZ

Dra. YOLANDA VIVES GILABERT

Cerdanyola del Vallès (Barcelona), 2015

PLANIFICACIÓN Y GESTIÓN DE RECURSOS EN LA EJECUCIÓN DE WORKFLOWS CIENTÍFICOS EN ENTORNOS HÍBRIDOS GPGPU

Esta tesis doctoral ha sido presentada por *Jordi Delgado Mengual* para el grado de Doctor en Computación de Altas Prestaciones por la Universidad Autónoma de Barcelona, bajo la dirección del Dr. Juan Carlos Moure López y la Dra. Yolanda Vives Gilabert, en el Departamento de Arquitectura y Sistemas Operativos

Directores,

Dr. Juan Carlos Moure López

Dra. Yolanda Vives Gilabert

Doctorando,

Jordi Delgado Mengual

Agradecimientos

Quiero empezar estas líneas de agradecimiento, por las dos personas que han tenido más paciencia conmigo en este largo camino que supone una tesis doctoral. Juan Carlos, que ha sabido dirigirme, apoyarme en los momentos duros y motivarme para llegar a buen puerto con su gran dedicación y entrega. Y también a Yolanda que me propuso el reto de empezar la carrera investigadora, me ha dado orientación, apoyo, consejos y me ha brindado con su amistad. A los dos os estaré eternamente agradecido por vuestra enseñanza y por permitirme vivir esta experiencia.

Un agradecimiento destacado por el apoyo y el aporte recibido por parte de Antonio Espinosa y Porfidio Hernández con los que también compartimos parte de este trabajo. También para Javier y Daniel, administradores de los sistemas de cómputo que hemos utilizado, gracias por ponerlo todo tan fácil!

Y como uno no camina solo por la vida, quiero agradecer también a aquellas personas que de alguna manera u otra llenan o han llenado las diferentes facetas de mi vida.

A mis compañeros doctorandos, a todos ellos mi reconocimiento por que también han hecho aportes en este trabajo, pero quiero hacer mención especial a Álex, César, Josefina y Javi. Por otro lado, a todos mis compañeros de carrera, pero en especial a Uri, Mingu, Ricard, Xavier, Álex, Oriol y Pedro, con los que compartimos largas tardes de prácticas, días de estudio y grandes risas.

A mis compañeros del PIC (Port d'Informació Científica), família de la cual formo parte desde hace ya siete años. A todos ellos con los que compartimos trabajo, aficiones y charlas; que tanto me soportan, me ayudan, y que en definitiva enre todos hacemos funcionar ese pequeño gran centro de datos que todos nos apreciamos tanto. En especial y no por peloteo, a Manuel Delfino, director del PIC, quien ha jugado un papel vital en esta tesis, ya no solo por todas las facilidades que me ha dado para poderla llevar a cabo, sino también por la confianza y los consejos que me ha brindado.

A mi querida Xantal, mi otra mitad inseparable e incondicional, mi compañera de viaje. La persona que le pone música a mis silencios y luz a mi oscuridad, por su gran paciéncia sobre todo en mis ausencias. A mi madre, mi hermano y mis sobrinos, esos incondicionales que siempre están ahí, que sufren cuando sufres y disfrutan cuando disfrutas. A mis tíos y primos, con los que juntos somos más que una família.

Un recuerdo muy emocional para mi padre, que se fue cuando este trabajo empezaba a caminar,

pero que sé que en este momento sería el padre más orgulloso del mundo, y también para mis abuelos que tanto amor me dieron.

No me puedo dejar a mi otra família, la que no es de sangre pero que los siento igual de míos, mis suegros, cuñados, tias y abuelos. Ellos que viniendo desde tierras andaluzas y alicantinas buscando prosperidad y respuestas a las preguntas de la vida, han permitido que encuentre mi otra mitad, me han hecho sentir como en casa desde el primer día, y me han apoyado incondicionalmente.

A todos mis amigos, Francesc, Mercè, Rak, Grant, Ari, Miguel, Dote, Laura, Jaime, Patrícia, Álex, Javi, Andrés, Víctor, Esther S., Toni F., Moli, Sandra, Òscar, Toni P., Raúl, Pol, Xavi, Jaime P., Lambert, Esther E, y un largo etc... A mis compañeros de grupo Miguel y Toni, con los que no haremos mucha carrera musical pero pasamos grandes ratos disfruntando en el local de ensayo. Y finalmente a mis compañer@s de gobierno y de partido, con los que estamos trabajando duro para aportar nuestro grano de arena y hacer de este un mundo mejor.

Resumen

Los *workflows* se han convertido en un esquema de organización de tareas ampliamente utilizado para el procesamiento de datos científicos. Esta tesis presenta varias técnicas de planificación y gestión de recursos en la ejecución de *workflows* científicos en entornos de computación híbridos.

En los últimos años, los procesadores de las tarjetas gráficas (GPUs) se han convertido en un recurso programable que se ha introducido en los sistemas de cómputo como aceleradores para ejecutar, a un coste económico bajo, otras tareas que no sea el procesamiento gráfico. Un ámbito donde se han introducido ha sido el campo del procesamiento de imágenes médicas, debido a que la alta resolución y tamaño de las imágenes suponen un alto coste en recursos y tiempo de computación. Este caso se da con FreeSurfer, que es un conjunto de herramientas que definen *workflows* de procesamiento de resonancias magnéticas.

La introducción de algunas implementaciones en GPU en las etapas del *workflow* de procesamiento de resonancias estructurales desde las versiones 5 y 5.3, ha supuesto una importante reducción del tiempo de ejecución por una sola instancia (ejecución serial) del *workflow*, alrededor de un 60-70% en función de la resonancia de entrada. Cuando queremos ejecutar múltiples instancias de *workflow* en un mismo nodo de cómputo nos encontramos que las GPUs resultan ser cuello de botella para la saturación de la memoria de la tarjeta. Para superar estas limitaciones se introducen a lo largo de este trabajo varias técnicas de planificación y gestión de los recursos compartidos, en especial la GPU.

En el primer estudio se presenta la propuesta de planificación que denominamos *inter-workflow*, con una tabla de restricciones que define un porcentaje de utilización de recursos para cada etapa y que sirve para tomar las decisiones de planificación. Esta planificación se aplica en la gestión de las dependencias de datos y de acceso a los recursos para el caso de uso definido por el *workflow* completo de análisis estructural de FreeSurfer. En la ejecución de múltiples instancias, aplicando esta planificación obtenemos una mejora de 6,79x respecto la ejecución serial.

Por otra parte, en el segundo estudio realizado, el caso de uso es el *subflow* de FreeSurfer para la reconstrucción volumétrica. Con la planificación que denominamos *por lotes* se introducen mecanismos de control de combinaciones de las tareas de las diferentes instancias a ejecutar, y obtenemos una ganancia de 10,48x respecto la ejecución serial. Esta mejora supone un 27% respecto a la planificación inter-worklfow aplicada en el mismo *workflow* y los mismos datos de entrada.

Resum

Els *workflows* han esdevingut un esquema d'organització de les tasques àmpliament utilitzat per al processament de dades científiques. Aquesta tesis presenta varies tècniques de planificació i de gestió de recursos per a l'execució de *workflows* científics en entorns de computació híbrids.

En els darrers anys, els processadors de les targetes gràfiques (GPUs) han esdevingut un recurs programable que s'ha introduït en els sistemes de còmput com a acceleradors per a executar, a un cost econòmic baix, altres tasques que no sigui el processament gràfic. Un àmbit on s'han introduït ha estat el camp del processament d'imatges mèdiques, degut a que l'alta ressolució i tamany de les imatges suposen un alt cost en recursos i temps de computació. Aquest cas es dóna amb FreeSurfer, que és un conjunt d'eines que defineixen *workflows* de processament de ressonàncies magnètiques.

La introducció d'algunes implementacions en GPU en les etapes del *workflow* de processament de ressonàncies estructurals des de les versions 5 i 5.3, ha suposat una important reducció del temps d'execució per una sola instància (execució serial) del *workflow*, al voltant d'un 60-70% en funció de la ressonància d'entrada. Quan volem executar múltiples instàncies del *workflow* en un mateix node de còmput ens trobem que les GPUs resulten ser coll d'ampolla per la saturació de la memòria de la targeta. Per superar aquestes limitacions s'introdueixen al llarg d'aquest treball vàries tècniques de planificació i gestió dels recursos compartits, en especial la GPU.

En el primer estudi es presenta la proposta de planificació que denominem *inter-workflow*, amb una taula de restriccions que defineix un percentatge d'utilització de recursos per cada etapa i que serveix per prendre les decisions de planificació. Aquesta planificació s'aplica en la gestió de les dependències de dades i d'accés als recursos per al cas d'us definit pel *workflow* complet d'anàlisi estructural de FreeSurfer. En l'execució de múltiples instancies, aplicant aquesta planificació obtenim una millora de 6,79x respecte l'execució serial.

Per altra banda, en el segon estudi realitzat, el cas d'us és el *subflow* de FreeSurfer per a la reconstrucció volumètrica. Amb la planificació que denominem per lots s'introdueixen mecanismes de control de combinacions de les tasques de les diferents instàncies a executar, i obtenim un guany de 10,48x respecte l'execució serial. Aquesta millora suposa un 27% respecte la planificació *inter-workflow* aplicada en el mateix *workflow* i les mateixes dades d'entrada.

Abstract

Scientific *workflows* have become a framework of working arrangements, widely used for scientific data processing. This thesis presents several scheduling and resource management techniques for the execution of scientific *workflows* on hybrid computing environments.

In recent years, the processors of the graphics cards (GPU) have become a programmable resource that has been introduced into computer systems as accelerators to run, in a low cost way, other tasks than graphics processing. One area where they have been introduced is the medical imaging field, because the high resolution and image size represent a high cost in terms of resources and computation time. This is the case of FreeSurfer, which is a set of tools which define Magnetic Resonance (MRIs) processing *workflows*.

The introduction of some implementations on the GPU in stages the of the processing *workflow* from structural resonances from versions 5 and 5.3, has resulted in a major reduction of the execution time in one *workflow* instance (serial execution), representing about 60-70% based on the input resonance. When we run multiple instances of the FreeSurfer *workflow* in the same computing node we find that GPUs become a *bottleneck* for the saturation of the GPU memory. To overcome these limitations several techniques are introduced throughout this work for planning and management of shared resources, especially the GPU.

We call the first proposal *inter-workflow* scheduling. It uses a list of restrictions that define the utilization rate of resources for each stage and is used to make planning decisions. This planning is applied in the management of data dependencies and resource access for the use case defined by the full FreeSurfer structural analysis *workflow* . In the execution of multiple instances, applying this planning achieves 6,79x *speedup* with respect to the serial execution.

Moreover, in the second study, the use case is the FreeSurfer volumetric reconstruction *workflow.* The batch scheduling that we propose, introduce control mechanisms for combining the execution of tasks from different instances. It obtains a *speedup* of 10,48x versus the serial exeuction This represents a 27% improvement compared to the *inter-worklfow* planning applied in the same *workflow* and to the same input data.

Índice de contenidos

Índice de figuras

Índice de tablas

1 Introducción

Los sistemas de cómputo han sufrido grandes cambios con el paso de los años. Las primeras arquitecturas multiprocesador aparecen en la década de los años 70, pero no ha sido hasta la entrada de los años 2000 donde la tecnología ha permitido extender los multiprocesadores en los ordenadores de uso común. La evolución predicha según la Ley de Moore se ha venido cumpliendo hasta los últimos años gracias a la división del procesador en múltiples núcleos de cómputo. La potencia de cálculo ha ido en aumento, aunque la frecuencia de reloj de los procesadores haya quedado más estancada debido a los límites físicos del silicio.

Por otra parte y en paralelo, las tarjetas gráficas han sufrido un gran avance en los últimos años gracias a la industria del videojuego. La necesidad de mayor realismo y mejor jugabilidad ha propiciado que los procesadores de las tarjetas gráficas, las GPU (Graphic Processing Unit) [1], hayan alcanzado un gran rendimiento en el cálculo vectorial o matricial. Es a partir del año 2006-2007 que los principales fabricantes crean interfaces que convierten a las GPUs en elementos programables para usos de propósitos más generales General Purpose GPU (GPGPU) [2,3]. Es a partir de ahí que nacen los entornos de cómputo híbridos desde el punto de vista de la capacidad de cómputo.

Los entornos híbridos son el presente y el futuro de la computación de altas prestaciones (del inglés HPC – High Performance Computing). Actualmente las GPUs, y otro tipo de aceleradores son componentes que se interconectan con el computador mediante el bus PCI-Express, pero los principales diseñadores de arquitecturas de cómputo ya recuperan el concepto de co-procesador en los nuevos diseños, para integrarlos en una misma placa o chip, algunos ejemplos son el Intel Sandy Bridge o el AMD accelerated processing unit (APU). [4,5].

Otra vez la historia se repite, y el *hardware* avanza más rápido que el *software*. Cada vez son más las aplicaciones que aprovechan la capacidad de las GPUs, pero el proceso no está exento de problemas. Uno de los principales problemas que aparecen al portar aplicaciones a GPU es la gestión de la memoria, ya que las capacidades de las tarjetas aún están por un orden de magnitud por debajo de las máquinas que actúan como huésped. Adicionalmente, debido a la dificultad de

diseño que supone manejar la memoria de la GPU y la API (Application Programming Interface) de CUDA (Computing Unified Device Architecture) [6,7], normalmente no se acaban portando aplicaciones enteras, y se migran solamente pequeñas partes de código, seleccionadas porque se adaptan mejor a las capacidades de la GPU y porque suponen una parte importante del tiempo de ejecución dentro de la aplicación.

Actualmente en muchos campos de la ciencia se realizan cálculos masivos mediante aplicaciones informáticas que consisten en la agrupación de un conjunto de programas ordenados siguiendo unas ciertas dependencias de datos, determinando un flujo de ejecución. A este flujo de ejecución y por extensión a estas aplicaciones se las denomina *workflows.* Como consecuencia de la evolución tecnológica y de las prestaciones de los sistemas de cómputo actuales, algunos de estos *workflows* han sido adaptados y optimizados para utilizar GPUs. En un estudio reciente del *HPC User Site Census* se plantea que de los 50 paquetes HPC más comunes entre sus usuarios, 34 ofrecen soporte GPU y si miramos el top 10, encontramos que 9 ofrecen soporte GPU [88] . Por lo tanto, podemos decir que es frecuente encontrarnos frente a aplicaciones que pueden ejecutar partes de código de manera combinada en CPU y en aceleradores tipo GPU, definiendo un escenario híbrido en el que la gestión de los recursos no es trivial.

Aunque asumamos que los elementos que componen el workflow son una especie de cajas negras en las que no podemos modificar su código fácilmente, existen muchas alternativas para mejorar el rendimiento en su ejecución en un sistema híbrido. Para poder aplicar mejoras es indispensable conocer bien el funcionamiento de la aplicación. La sintonización de los principales parámetros que regulan el acceso a los recursos, como por ejemplo el uso de memoria o disco, son tareas que ayudan a sacar un mayor partido a las arquitecturas y a la par un mayor rendimiento a las aplicaciones. Este trabajo trata de cómo tomando como caso de uso una aplicación que analiza y procesa resonancias magnéticas, asumiendo que los elementes de su flujo de ejecución son cajas negras, podemos mejorar la eficiencia de la gestión de los recursos considerando un entorno híbrido.

El coste en tiempo de ejecución de los *workflows* de análisis mencionados en este trabajo suele ser elevado, del orden de horas. Por lo tanto, uno de nuestros principales objetivos generales es la reducción de ese tiempo mediante la mejora en el uso de los recursos computacionales disponibles. Para mejorar la eficiencia de uso de los recursos existentes en las plataformas hay que tener en cuenta la naturaleza heterogénea de estos recursos en los sistemas computacionales híbridos de hoy en día.

A lo largo de esta tesis se presentan dos aproximaciones que permiten ejecutar aplicaciones

basadas en *workflows* científicos.

Como primera propuesta, inicialmente se explota el paralelismo intra-*workflow*. Es decir, se analizan las etapas que componen el *workflow* y se ejecutan en paralelo aquellas etapas que se muestran libres de dependencias. Como resultado, tenemos una serie de etapas que se ejecutan en serie y una parte en paralelo. Esta última puede aprovechar los recursos híbridos disponibles: las CPUs y las GPUs del sistema.

A pesar de que con la ejecución del *workflow* mediante el esquema paralelo (intra-*workflow*) doblamos los recursos CPU y GPU que usamos, aún quedan recursos disponibles en el sistema que pueden ser aprovechados. Por ello, proponemos una estrategia para ejecutar múltiples instancias del *workflow*, para diferentes datos de entrada, con el objetivo de utilizar el máximo de recursos disponibles, a lo que llamaremos paralelismo *inter-workflow*. Inicialmente, nos centramos en un caso de estudio con la ejecución de 2 instancias idénticas en nuestro sistema de cómputo. Esto nos permite llegar a utilizar todos los procesadores en el conjunto de etapas paralelas.

Como resultado, atendiendo a la utilización de los recursos computacionales, la ejecución de dos *workflows* simultáneos es muy similar al de un único *workflow*. Además, comprobamos que la capacidad de la memoria de la GPU es un recurso limitante que impide garantizar la ejecución de más de dos *workflows* simultáneos compartiendo la GPU en el sistema de cómputo definido.

Para solucionar estos problemas, se presenta un primer planificador centralizado para poder ejecutar múltiples *workflows* simultáneamente. Este planificador gestiona la principal limitación detectada, como es el uso de memoria de la GPU. El planificador centraliza la gestión de todos los datos de entrada y ejecuta las etapas definidas en el esquema paralelo. La planificación se realiza mediante una política que asigna recursos disponibles en función de las tareas pendientes de ejecución, intentando conseguir el máximo rendimiento posible. Esta ejecución planificada permite ejecutar múltiples instancias de forma segura aumentando hasta 6x veces la productividad (*throughput*) del sistema, medida en sujetos procesados por hora.

Después de utilizar este primer planificador se comprueba que los recursos de cómputo (CPU y GPU) no están siendo utilizados de forma eficiente durante una considerable parte del tiempo de ejecución. Por tanto, sigue existiendo un importante margen de mejora para la productividad del sistema.

A partir de aquí se propone un esquema más flexible para la ejecución planificada de las tareas correspondientes a múltiples instancias de un *workflow*. Basándonos en un análisis empírico inicial de las características de la ejecución de las tareas del *workflow*, y del efecto de la ejecución combinada de tareas de diferente tipo, propondremos unas políticas y mecanismos de

planificación que logren un uso eficiente tanto de las CPUs como de las GPUs del sistema. De la misma forma, se busca mantener balanceada la carga asignada a las CPUs y la carga asignada a las GPUs.

Para ello, se presenta un segunda propuesta con un nuevo mecanismo de planificación *por lotes*, que se componen de un conjunto de tareas independientes organizadas en listas. Estas listas separan las tareas según se asocian a diferentes conjuntos de recursos. La política de planificación incorpora un mecanismo de equilibrado de la carga de trabajo entre listas.

En este caso, se consideran un gran número de datos de entrada. Para cada uno de ellos, se ejecuta un *workflow* con varias tareas agrupadas en diversas etapas consecutivas. Al aumentar el paralelismo potencial de tareas a ejecutar, disponemos de una mayor flexibilidad para elegir y combinar las etapas más adecuadas para mejorar la eficiencia de los recursos disponibles en cada momento.

Esta segunda propuesta realiza un análisis empírico inicial de la ejecución de las tareas del *workflow* para diseñar un plan de ejecución estático de las etapas. La clasificación y el orden de las tareas en las listas favorecen el uso equilibrado de los recursos y la ocurrencia de combinaciones de tareas más eficientes. La ejecución de los lotes se realiza en estructura de *pipeline* entrelazado.

El *pipeline* se compone de lotes o grupos de etapas de proceso de múltiples instancias del *workflow* de manera que estos grupos de etapas se organizan en listas de tareas que se ejecutan de manera combinada de la forma más favorable posible. En la estrategia de planificación se introducen mecanismos de re-planificación o compensación, que modifican la distribución de las tareas entre listas de manera dinámica. Con esta propuesta el rendimiento alcanzado es de un 27% superior, respecto a la primera propuesta.

A continuación se presentan los objetivos del presente trabajo así como una breve descripción de la organización del documento.

1.1 Objetivos de la Tesis

Actualmente, los estudios científicos se nutren de grandes cantidades de datos obtenidos gracias a la mejora en los instrumentos y aparatos de medida. Esto nos permite obtener mucha más información y a menudo de más calidad, pero el análisis y el procesamiento de estos datos requiere de mayores necesidades de cómputo. El procesamiento masivo de datos consiste en flujos de trabajo (*workflows*) que se organizan en etapas, y que presentan dependencias entre

ellas. Estos *workflows* definen un *pipeline* de procesamiento.

Esta tesis tiene por objetivo analizar y proponer metodologías para la ejecución de *workflows* científicos maximizando el *throughput*, es decir, el número de *workflows* ejecutados por unidad de tiempo. Se analizará el comportamiento de la ejecución de *workflows* científicos en nodos de cómputo híbridos, formados por una o múltiples CPUs (*multicore*) y por tarjetas GPU que se utilizan como aceleradores (*manycore*).

Concretamente, para validar las propuestas realizadas en el presente trabajo, se han realizado diversas experimentaciones sobre FreeSurfer [9], una aplicación diseñada para el análisis y procesamiento de resonancias magnéticas (del inglés, *Magnetic Resonance Images*, en adelante *MRI*), que es ampliamente utilizada por la comunidad neurocientífica. FreeSurfer contiene diferentes *workflows* de análisis de resonancias magnéticas, tanto para el análisis estructural como funcional de una resonancia. A su vez, cada *workflow* está compuesto por diferentes *subflows* (o subconjuntos de etapas del *workflow*) que se pueden ejecutar separadamente. En concreto, en este trabajo se realiza la experimentación con el *workflow* de análisis estructural de MRIs.

El presente trabajo se centra en el análisis de FreeSurfer tomándolo como una serie de etapas que componen el *workflow* de análisis. La ejecución de este *workflow* es lo suficientemente flexible como para poder reordenar y gestionar la ejecución de estas etapas sin modificar la funcionalidad existente. Esta gestión flexible está cada vez más extendida en la comunidad científica mediante el uso de herramientas y entornos como Taverna [10] o Galaxy [11].

Los análisis de resonancias magnéticas (MRI) que realizan los investigadores clínicos con el *workflow* de análisis estructural, parten de una muestra de sujetos de estudio organizada en varios grupos según el estadio de una enfermedad y un grupo de sujetos sanos. Para poder extraer los datos necesarios (biomarcadores) del análisis a realizar, hay que procesar cada sujeto de la muestra ejecutando una instancia del *workflow* de FreeSurfer. Una vez cada instancia termina, un analista experto (neuroradiólogo, o neuropsicólogo) realiza un control de calidad sobre la imagen procesada e introduce correcciones manuales. Hay tres tipos de correcciones manuales: correcciones sobre la materia blanca, introducción de puntos de control para refinar la segmentación, o bien modificaciones sobre la capa superficial del córtex (pial). En función de las modificaciones introducidas, se tiene que re-procesar el sujeto mediante la re-ejecución de uno o más *subflows* del *workflow* principal.

Una vez el proceso de control de calidad del procesamiento ha finalizado, se extraen datos sobre la imagen procesada. Estos datos pueden ser métricas globales del cerebro, como el grosor

cortical, el volumen intracraneal, etc o parcelados por diferentes regiones del cerebro. Con estos datos se realiza un análisis estadístico con todos los sujetos de la muestra para extraer los biomarcadores relevantes, mediante la comparación entre los grupos de sujetos.

El objetivo general que hemos establecido se puede dividir en sub-objetivos más concretos:

- Analizar la ejecución de una única instancia del *workflow* con un único sujeto:

- Analizar el funcionamiento del *workflow*. Obtener las dependencias entre las etapas del *workflow* y así estimar el paralelismo potencial de la ejecución.

- Analizar el rendimiento de la ejecución del *workflow*. Obtener una caracterización de las etapas del *workflow* basada en el tiempo de ejecución y uso de memoria (tanto en CPU como en GPU).

- Evaluar, para cada una de las etapas adecuadas, cuál es la ventaja del uso del acelerador GPU.

- Proponer un esquema de ejecución aprovechando el paralelismo potencial entre etapas, y utilizando los aceleradores cuando sea conveniente.

- Analizar y mejorar la ejecución de múltiples instancias del *workflow* con múltiples sujetos:

- Analizar la ejecución del esquema paralelo propuesto en el análisis de una única instancia del *workflow*, pero ahora con múltiples instancias y sujetos. Detectar problemas e ineficiencias en el rendimiento de la ejecución.

- Proponer mecanismos y políticas de gestión de los recursos del nodo híbrido de cómputo para la ejecución de múltiples instancias del *workflow*, que resuelvan los problemas detectados en el análisis anterior.

- Controlar la combinación de etapas que coinciden al ejecutar múltiples instancias del *workflow*.

- Analizar el rendimiento del sistema cuando se hacen ejecuciones simultáneas del mismo tipo de etapa.

- Obtener el rendimiento de cada una de las etapas, y los recursos que utilizan de forma conjunta. A la vez, estudiar el efecto de la topología NUMA en sistemas multiprocesador.

- Analizar la afinidad en la ejecución combinada de etapas del *workflow* de diferente tipo. El objetivo es ejecutar el mayor número de etapas por unidad de tiempo, con una gestión eficiente de los recursos.

- Proponer nuevos mecanismos y políticas de planificación, que de forma automática, y a partir de

las combinaciones de etapas propuestas, incrementen el *throughput* al gestionar la ejecución de múltiples instancias de las etapas del *workflow*.

- Evaluar el rendimiento de la ejecución utilizando las propuestas implementadas.

- Generalizar las principales aportaciones del trabajo

1.2 Organización de la memoria

Este trabajo está organizado de la siguiente manera:

- En el capítulo 2 se presentan el contexto en el que se enmarca este trabajo, así como una descripción más detallada sobre la estructura y el funcionamiento de FreeSurfer y finalmente una revisión del estado del arte enmarcado en la planificación y gestión de recusos y también en la gestión de aplicaciones que combinan el uso de recursos CPU y GPU.

- En el capítulo 3 se realiza un primer análisis del *workflow* de estudio para encontrar ineficiencias a la ejecución híbrida mezclando etapas del *workflow* ejecutadas en CPU y etapas ejecutadas en CPU+GPU. Se propone un nuevo esquema para el *workflow* de estudio aplicando una combinación de paralelismo intra-*workflow* (con las etapas del mismo *workflow*) e *inter-workflow*, mediante la ejecución de múltiples instancias del *workflow*. Se propone también un planificador que resuelve los problemas de acceso a los recursos del sistema, en especial en el acceso compartido a la GPU.

- En el capítulo 4 se presenta una segunda propuesta de planificación *por lotes* de tareas que se basa en una metodología de análisis y caracterización del *workflow* científico que describe cómo se usan los recursos del sistema y qué problemáticas aparecen. Se realizan pruebas de ejecuciones combinadas de varias tareas del *workflow* para encontrar aquellas que usen los recursos de manera más favorable, así como un esquema de planificación mediante un plan de ejecución que define la distribución de tareas y de recursos y que se ejecuta en utilizando una estructura *pipeline* de ejecución entrelazada para resolver dependencias de datos.

- En el capítulo 5 se exponen las conclusiones alcanzadas en ambas propuestas de este trabajo. También se describen algunas líneas futuras del presente trabajo y se detallan las publicaciones realizadas como contribuciones originales derivadas del trabajo de esta tesis.

2 Conceptos previos

En este capítulo se presenta una breve introducción al contexto de uso de la aplicación FreeSurfer que usaremos a lo largo de este estudio. A continuación se introducen los principales conceptos que permiten comprender el funcionamiento y la aplicabilidad de FreeSurfer, que es un conjunto de herramientas para el análisis y la visualización de MRIs estructurales y funcionales. Finalmente se hace un repaso sobre el estado del arte en los ámbitos de la planificación y gestión de recursos en la ejecución de *workflows* científicos, asi como también en el uso de los procesadores gráficos GPUs para aplicaciones de propósito general, donde actúan como aceleradores.

2.1 Contexto del problema

En este apartado se presenta una introducción al contexto donde se inscribe el estudio y las propuestas de esta tesis: desde la adquisición de resonancias magnéticas, pasando por el tipo de datos o de procesamiento, hasta presentar la aplicación FreeSurfer, que se analizará con más detalle en secciones posteriores.

El diagnóstico por la imagen es una técnica ampliamente utilizada por la medicina para poder detectar malformaciones o afecciones de enfermedades diversas. Una rama del diagnóstico por la imagen es la aplicación en la neurorradiología la cual centra su estudio en la afectación y evolución de enfermedades neurodegenerativas. Con el constante incremento del campo magnético generado por los escáneres, la resolución de las resonancias magnéticas (MRIs) se ha incrementado en los últimos años de manera significativa [12]. Debido a este fenómeno, las técnicas de imagen médica se convierten en una referencia importante en el diagnóstico de una gran cantidad de enfermedades debido a la principal ventaja que proporcionan al ser técnicas no invasivas.

El protocolo de adquisición de resonancias magnéticas determina el número de secuencias a tomar, la duración y el tipo de adquisición (funcional o estructural). Una secuencia de adquisición de una resonancia magnética puede ir de las 30 MB de datos a las 400 MB. El tiempo de

adquisición puede oscilar entre los 20 y los 50 minutos. Recientemente, se han desarrollado nuevas técnicas de computación para mejorar el procesamiento de imágenes y poder obtener más información sobre las diferentes estructuras anatómicas o funcionamientos, en nuestro caso del cerebro. Este tipo de información puede ser utilizada como biomarcadores (indicadores de un estado biológico) para estudiar la evolución de enfermedades neurodegenerativas como el Alzheimer o el Parkinson. De momento, estas técnicas únicamente se utilizan para investigación, pero ofrecen muy buenos resultados y es previsible que en el futuro también se utilizarán en la práctica clínica.

Habitualmente, el procesamiento de resonancias magnéticas se realiza simultáneamente sobre cientos de sujetos (pacientes) de estudio, en el que el tiempo de respuesta se convierte en un factor clave en este tipo de procesos. Por este motivo, existe la necesidad de disponer de herramientas *software* especializadas en el tratamiento de resonancias magnéticas que puedan ser aplicadas a plataformas *hardware* que permitan procesar paralelamente.

FreeSurfer (FS) es un conjunto de herramientas para el análisis y la visualización de MRIs estructurales y funcionales, desarrollado por el centro de investigación Athinoula A. Martinos Center for Biomedical Imaging, del *Massachusetts Institute of Technology*. FS se encuentra disponible en una versión no comercial en la red. Esta aplicación se publicó a principios de los años noventa y ha sido ampliamente validada con publicaciones científicas dentro de los ámbitos médicos y técnicos. Posteriormente ha sido ampliada con más funcionalidades gracias a su estructura modular.

El *workflow* de FS para el análisis de MRIs estructurales consta de 32 etapas ejecutadas secuencialmente. Estas etapas componen dos grandes *subflows*, el primero para el análisis del volumen cerebral y el segundo para el análisis de superficie del cerebro. Este segundo incluye *subflows* más pequeños que pueden ser ejecutados independientemente.

El análisis de volumen cuantifica el volumen de cada estructura subcortical mediante cinco procedimientos principales descritos en [13,14]:

1. Registro en el espacio de coordenadas Talairach, diseñado con el objetivo de no depender de la patología del paciente y que maximice la precisión de la segmentación posterior.
2. Etiquetado de los volúmenes calculados inicialmente.
3. Normalización de la intensidad de la imagen debido a la falta de homogeneidad del campo magnético
4. Alineación no lineal de las imágenes volumétricas en el atlas Talairach.
5. Etiquetado del volumen a partir de un conjunto de MRIs segmentadas manualmente que se han mapeado en un espacio común para tener una correspondencia punto a punto

entre las imágenes a segmentar y el espacio común.

El análisis de superficie calcula para cada punto del córtex cerebral, el grosor, volumen, área de la superfície y curvatura [15,16]. La ejecución de los *subflows* se gestiona mediante un script en función de los parámetros de entrada.

A continuación se presenta el diagrama de funcionamiento y dependencias de datos entre las diferentes tareas del *workflow* de análisis estructural de FreeSurfer. Dado la envergadura del diagrama de flujo se ha separado en dos figuras, las figuras 2.1 y 2.2. En la figura 2.1 se representa que inicialmente el *workflow* recibe como datos de entrada los diferentes ficheros con los cortes que componen una MRI. La primera tarea une todos los cortes en un volumen único (rawavg.mgz) e inicia la secuencia de etapas del *workflow* con la ejecución de la primera etapa del *workflow* denominada *Motion Correction (MC)*. Observamos también que en la salida de cada tarea, se genera un fichero que quedará almacenado como dato intermedio en el disco, y que sirve como *checkpoint* entre etapas. Finalmente destacar del esquema los diferentes *subflows*: autorecon1 formado por las etapas en violeta, autorecon2(-wm,-pial,-cp) etapas en rojo y autorecon3 con las etapas en azul.

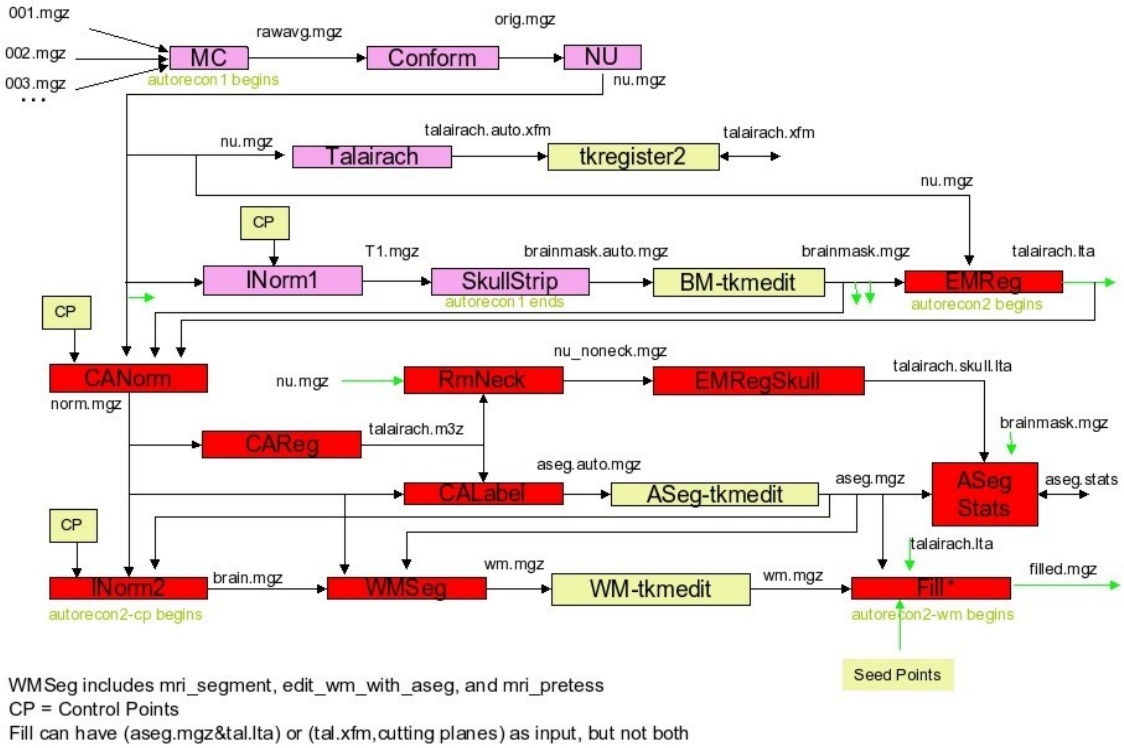


Figura 2.1: representación de las etapas del *workflow* FreeSurfer relativas a la reconstrucción volumétrica de una resonancia magnética. Resaltan en diferentes colores las etapas de procesamiento que componen cada uno de los *subflows* que incluye el flujo principal. En violeta las etapas relativas al *subflow* autorecon1, en rojo las etapas relativas al *subflow* aoutrecon2 y sus variantes -wm (white matter) y -cp (control points). También se indican las dependencias de datos entre etapas así como los datos que se leen o producen en forma de fichero en cada una de las etapas. [figura extraída de 9]

Cabe mencionar también que las etapas contenidas en la figura 2.1, pertenecen principalmente al *subflow* de reconstrucción volumétrica de la MRI y las etapas contenidas en la figura 2.2, pertenecen al análisis de superficie.

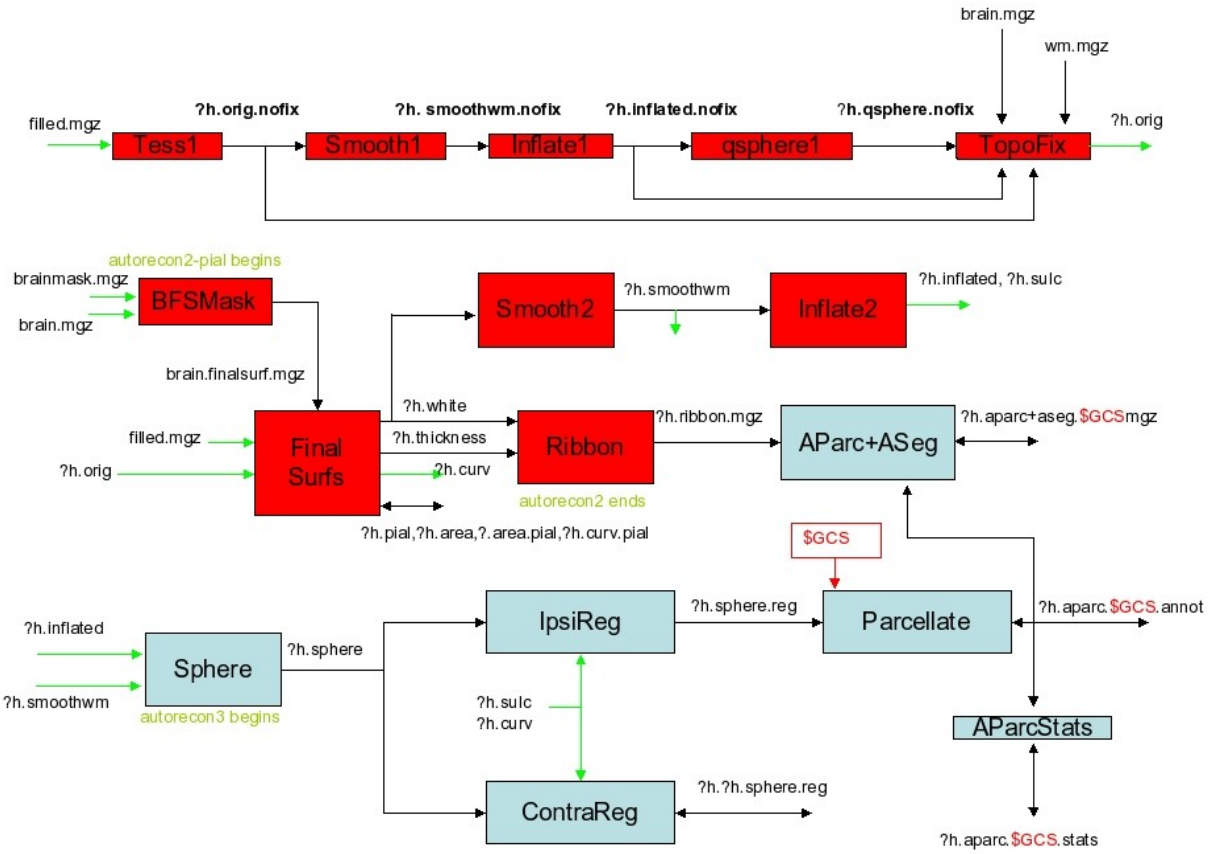


Figura 2.2: continuación de la Figura 2.1, con la representación de las etapas que corresponden al flujo de análisis de superficie de la MRI. En rojo resaltan las etapas del *subflow* autorecon2 y por otra parte en azul las etapas relativas al *subflow* autorencon3. Figura extraída de [9]

Desde la versión 5 de FreeSurfer (actualmente la última estable es la versión 5.3), se incorporan mecanismos de paralelización en las etapas más costosas en tiempo de ejecución. Concretamente, se incorporan varias etapas implementadas con soporte GPU en CUDA [17]. En el presente estudio tendremos en cuenta las implementaciones en CUDA para las etapa *CAReg, SkullStrip, QSphere y Sphere.*

2.2 Descripción del problema

Los *workflows* científicos se componen de una serie de etapas interrelacionadas entre sí. Una etapa puede estar formada por una aplicación individual o un conjunto de varias. A partir de aquí, cada etapa del *workflow* lee unos datos de entrada, los procesa y los envía a la siguiente etapa hasta obtener el resultado final.

Habitualmente los *workflows* son computacionalmente costosos, debido a que toman del orden de varias horas de cálculo. Algunas de las causas que provocan este costo de tiempo son o la gran cantidad de datos que deben procesar, o bien por la secuencia de etapas que describen los diferentes estadios de procesamiento del *workflow* y los algoritmos asociados.

En nuestro caso, cada una de las ejecuciones del *workflow* pasa por una serie de tareas que están agrupadas en un conjunto de etapas. Cada etapa puede corresponderse con uno o más procesos de cómputo y estos procesos siempre se ejecutan en CPU, pudiendo requerir el uso adicional de una GPU. Este uso adicional de recursos externos a la arquitectura clásica del computador, como son las GPUs, ha sido introducida como una estrategia de paralelismo útil para tratar de reducir el tiempo de ejecución permitiendo mejorar el uso de los recursos de una forma más eficiente.

En el contexto de este trabajo nos planteamos un escenario donde tenemos un *workflow* con capacidad de usar las GPUs en algunas de sus etapas de ejecución. Esto supone poner atención en la gestión de los recursos de cómputo en un entorno híbrido, y debe hacerse tanto para garantizar un uso eficiente de las CPUs del sistema como también de la GPU. A esta complejidad se le debe añadir la dificultad de ejecutar múltiples instancias del mismo *workflow* dentro de un mismo nodo de cómputo de manera que pretendemos explotar al máximo las capacidades del mismo.

El propósito de usar un nodo de cómputo para realizar ejecuciones de *workflows*, nos encaja con el contexto antes descrito en el que tomamos el *workflow* de análisis estructural de resonancias magnéticas que describe FreeSurfer como caso de uso. Supongamos que a lo largo de un día de reclutamiento se han realizado N resonancias magnéticas, utilizando un nodo de cómputo conectado a la consola de adquisición, nos permitiria incluir nuevos sujetos a una muestra de un estudio clínico en un corto plazo tiempo tras el reclutamiento del sujeto en la máquina de resonancia, facilitando así poder realizar los análisis necesarios y ayudando en el diagnóstico y atención a los pacientes. Este planteamiento supone una solución económicamente asequible en una infraestructura hospitalaria y además favorece el cumpliemiento de requerimientos de privacidad y ética, por los que se rige la manipulación con datos clínicos.

Posteriormente se define un segundo entorno de análisis. En este caso se necesitan procesar los datos de cientos o miles de sujetos disponibles en un repositorio de datos biomédico. En este caso, nuestro objetivo es mejorar la productividad medida en número de sujetos analizados por unidad de tiempo aprovechando el máximo de los recursos computacionales existentes. Esta eficiencia en el uso de los recursos va a ser analizada midiendo el *speedup* o mejora entre la

ejecución de las tareas usando todos los recursos disponibles del sistema respecto a la ejecución serie.

Nuestras propuestas consideran una serie problemas que se producen típicamente en entornos de ejecución de *workflows* con muchas etapas en vuelo. Específicamente, se estudian y se presentan soluciones para los siguientes problemas que enumeramos a continuación:

- **Afinidad entre etapas**: es necesario analizar la afinidad entre las tareas correspondientes a diferentes etapas. Es decir, existen ciertas combinaciones de etapas que al ser ejecutadas simultáneamente ofrecen un rendimiento favorable. Mientras, existen otras combinaciones que tienen un rendimiento peor que al ejecutarse de forma individual. En la práctica, observamos que cuando los cuellos de botella de una tarea son distintos de los de otra, su ejecución combinada se complementa. De aquí, nuestro objetivo es hacer uso de esta afinidad entre tareas para diseñar un mejor plan de ejecución global. Se diseña un plan de ejecución estático de cada lote de etapas del *workflow*. El plan de ejecución se define mediante un conjunto de listas de tareas, de forma que la clasificación de las tareas del lote y a su vez el orden que toman en cada lista favorece el uso eficiente de los recursos y la ocurrencia de combinaciones más eficientes de tareas.

- **Gestión del solapamiento entre etapas**: un objetivo adicional del plan de ejecución es que el solapamiento entre lotes de ejecución sea mínimo. Uno de los problemas más importantes a gestionar es que se cumplan las dependencias de datos entre etapas de diferentes lotes. Como solución a este problema, se presenta un esquema entrelazado entre lotes de ejecución pares y lotes impares que permite el solapamiento de hasta un lote completo. El entrelazado propuesto aumenta la latencia de procesamiento de los sujetos de entrada pero facilita la gestión de las dependencias entre tareas.

- **Gestión de los dominios NUMA para favorecer la reutilización de datos y minimizar la contención**: los sistemas de memoria principal en la actualidad se dividen en dominios de forma que es necesario pasar a través de otro procesador para acceder a un dominio no local. Los gestores de tareas no suelen tener en cuenta estos dominios a la hora de asignar tareas a CPUs, con lo que se pueden producir situaciones de contención por acceso a un dominio remoto. En nuestro caso, al ejecutar el máximo número de etapas simultáneas en los procesadores, se diseña una configuración del sistema de forma que las tareas se ejecutan siempre en el dominio NUMA más cercano.

- **Uso de *hyperthreading* para mejorar la eficiencia del uso de los recursos**: la capacidad de ejecutar múltiples *threads* en un mismo núcleo de cómputo es frecuente en

los sistemas de cómputo actuales. En el presente trabajo se muestra de forma empírica que el uso de varios *threads* de ejecución de forma simultánea en el mismo núcleo de CPU mejora la eficiencia de uso de las CPUs y además, permite abrir la posibilidad de considerar qué posibles combinaciones de tareas tienen mayor afinidad a la hora de compartir la CPU.

- **Adaptación del plan de ejecución al sistema en uso**: en una primera fase, se caracterizan las etapas del *workflow* en función de su rendimiento. Después, se analiza la afinidad de estas etapas ejecutando tareas que comparten el mismo núcleo de cómputo, un núcleo diferente dentro del mismo procesador o la misma GPU. Cuando se crea el plan de ejecución, se define un tamaño de ventana (w) el cual servirá para definir el número de tareas de cada tipo que se pueden ejecutar como máximo simultáneamente. Cómo se desarrollará a lo largo del capítulo 4 de este trabajo, se debe priorizar el uso de la GPU (*best-GPU-effort*) o el uso de la CPU (*higher-CPU-efficiency*) dependiendo de la capacidad de ejecución en CPU y en GPU que tenga el sistema de cómputo en uso.

2.3 Entornos híbridos

La mayoría de las aplicaciones tradicionalmente han sido desarrolladas para ejecutarse en un solo flujo (*Thread*) de ejecución. Han sido necesarios más de 30 años de evolución tanto del *hardware* como del *software* para poder desarrollar códigos que pudieran ejecutar varios flujos (*Threads*).

Esta evolución tecnológica de los procesadores ha seguido la Ley de Moore, que expresa que cada dos años se duplica el número de transistores que tiene un ordenador. Esta ley se ha ido cumpliendo en los procesadores *mono-thread* hasta que aproximadamente en el año 2003 se encontró con un límite fijado por la relación potencia de cómputo y la disipación de calor de la circuitería que ofrecían los computadores.

Debido a estos problemas, los diseñadores de arquitecturas de computadores se han visto obligados a adoptar una estrategia basada en la división de los procesadores en núcleos más pequeños de cómputo y que trabajan de forma relativamente independiente. Cada chip físico contiene actualmente más de un núcleo de cómputo, llegando a los *multicores*, término que se utiliza a nivel más genérico. Además, cada máquina puede contener más de un chip procesador.

Paralelamente a la evolución de las CPUs, las tarjetas gráficas, y concretamente sus chips llamados GPUs (Graphic Processing Unit) [1], han evolucionado de manera drástica en los últimos años. Esta evolución ha sido causada por la industria de los videojuegos, ya que el realismo de los juegos ha mejorado mucho y ha requerido que los chips de las tarjetas tuvieran que procesar y

renderizar cada vez más polígonos, texturas, etc..

Los principales fabricantes de tarjetas gráficas, NVIDIA y ATI, publicaron el año 2007 unos entornos de programación para las tarjetas gráficas. NVIDIA publicó *CUDA (Compute Unified Device Architecture)* [6] y *ATI publicó ATI FireStream* (Basado en OpenCL) [18]. Esto abriría las puertas a poder utilizar las tarjetas gráficas para otros propósitos más generales que no sólo la visualización y renderización de imágenes. Los llamados entornos GPGPU (General Purpose GPU) [2,3], utilizan la GPU para aplicaciones de propósito general, actuando como un co-procesador. Un amplio número de los principales supercomputadores mundiales recogidos en los listados del TOP500 [19] y del Green500 [20] incorporan actualmente las GPUs.

En el mercado podemos encontrar principalmente dos gamas de tarjetas GPU. Una gama más básica pero con mucho potencial de cómputo, es la gama doméstica orientada a los videojuegos. Permite realizar eficientemente operaciones de precisión simple, y ofrecen un menor consumo de potencia que hay que tener en cuenta en su instalación en entornos de sobremesa. Por otra parte, encontramos una gama más profesional orientada al cómputo científico, que ofrece mejores prestaciones en el cómputo de operaciones con doble precisión y una mejor in tegración en *clusters*. Ambas gamas ofrecen buenas prestaciones y buenos rendimientos, teniendo en cuenta la relación coste económico/rendimiento. Se pueden obtener GPUs de muy buen rendimiento a precios alrededor de los 300 €.

Mientras que las CPUs o procesadores *multicores* actuales contienen unas pocas docenas de núcleos de cómputo, las GPUs actuales contienen cientos de núcleos, por lo que las GPUs se llaman *manycores*. En la figura 2.3 se puede observar la tendencia en el rendimiento de las GPUs comparado con el de las CPUs tradicionales.

En la gráfica de la figura 2.3 se observa la evolución de las CPUs, con la aparición de los primeros procesadores dual-core o quad-core, así como también la evolución de los dos principales fabricantes de tarjetas gráficas (ATI-AMD) y NVIDIA. A partir de 2006 se empieza a acentuar el hueco entre los GFLOPS que ofrecen las CPUs frente a las GPUs.

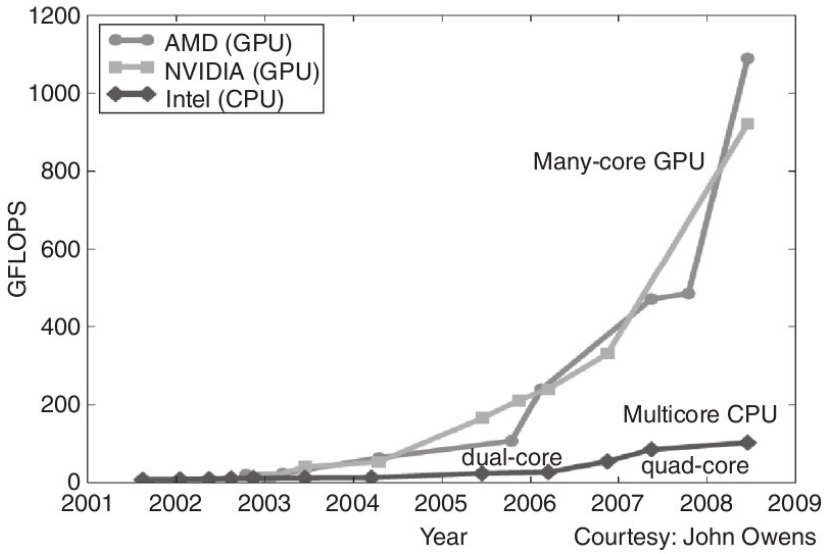


Figura 2.3: evolución de las GPUs respecto las CPUs en GFLOPS. Extraída de [21].

Los entornos de programación también han ido evolucionando. NVIDIA ha evolucionado su entorno CUDA, adaptándolo a cada generación de tarjetas (Fermi, Kepler, Maxwell ...) para poder extraer todas las potencialidades, lo que le ha llevado a controlar el mercado y la aplicación de los GPUs como co-procesadores. Por otra parte, ATI aparentemente parece no haberse posicionado con tanta fuerza en el mercado, aunque su arquitectura funcione en base a OpenCL (Open Computing Language) que es un framework *Open Source* multiplataforma. Estos paradigmas de programación están orientados al modelo *SIMD (Single Instruction Multiple Data)*. Pretenden explotar las capacidades de la tarjeta GPU ejecutando múltiples operaciones simples en cada procesador *(Stream Processor)* de la arquitectura de la GPU.

La arquitectura de las GPUs difiere mucho de la arquitectura típica de las CPUs. Podemos ver en la figura 2.4 una representación de las dos arquitecturas.

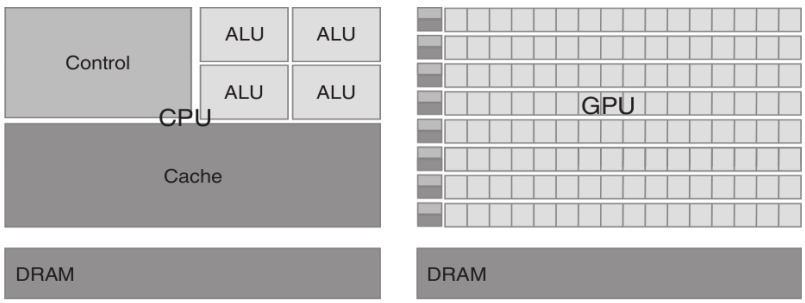


Figura 2.4: arquitectura CPU (izquierda) y arquitectura GPU (derecha). Extraída de [21].

A simple vista hay que destacar que las GPUs tienen muchos más núcleos de procesamiento *(Streaming Processors, SPs)*, agrupados en (*Streaming Multiprocessors, SMs)* donde comparten la lógica de control y diferentes memorias locales. Cada GPU tiene una memoria interna propia (memoria global, memoria de texturas...) además de tener conexión, mediante el Bus, a la memoria principal del computador.

Uno de los factores que ha facilitado la incorporación de las GPUs en los que podemos llamar entornos híbridos, CPU+GPU, ha sido la relación coste económico, potencia de cálculo. Otro factor es la facilidad de instalación, mediante el bus PCI-Express, hace que sea una tecnología muy accesible.

2.4 Trabajos relacionados

En esta sección se presenta un breve resumen del estado del arte enmarcado en dos centros de interés principales; por una parte la planificación y la gestión de recursos en la ejecución de worklfows científicos, y por otra, la gestión y el impacto del uso de tarjetas GPU en los entornos y aplicaciones híbridas, que combinan parte de su ejecución en CPU y parte en GPU.

La planificación y gestión de recursos en entornos homogéneos de cómputo paralelo, dada su influencia en las prestaciones del sistema, es uno de los tópicos más estudiado y analizado; existiendo gran cantidad de aportaciones en la literatura. No obstante, y aunque la tendencia generalizada de las nuevas arquitecturas multi/*manycore* es hacia la heterogeneidad de los componentes, la mayoría de los estudios en el campo antes mencionado, lo han capitalizado los sistemas distribuidos (clusters, multiclusters, grid y clouds). Siendo esto así, no deja de sorprender, que dentro del área definida por el nodo de cómputo, en el ámbito de las arquitecturas heterogéneas; las aportaciones que aparecen en la literatura son más bien escasas. Así pues, podemos deducir que nos encontramos frente a una nueva problemática que no se ha estudiado en profundidad y que merece la pena abordar. Este trabajo trata de poner un poco de luz en esta línea de actuación, proporcionando técnicas, metodologías de trabajo, y propuestas de planificación.

Si bien es cierto, que al igual que en los sistemas distribuidos heterogéneos la problemática a resolver es similar a la tratada en este trabajo, las características específicas propias de cada infraestructura de cómputo paralelo del tipo cluster/grid/cloud, genera una mayor complejidad a la hora de elaborar una propuesta; dado que el nodo es sólo una parte más del sistema a considerar. No obstante, aspectos como la impredecibilidad y dinamismo de la carga, así como el costo de la comunicación en los sistemas de múltiples núcleos heterogéneos (de varios órdenes de magnitud

menor que en los sistemas distribuidos), obliga a la adaptación de las políticas utilizadas para estas infraestructuras, si se desean utilizar para el caso que nos ocupa.

La mayoría de los estudios relacionados con los sistemas distribuidos heterogéneos, proponen políticas de gestión de recursos para *workflows*, basados en la asignación de trabajos y planificación de tareas tanto a nivel estático como dinámico; dependiendo del conocimiento que se tenga de los parámetros que caracterizan las aplicaciones del *workflow*. Siendo los modelos utilizados para representar las aplicaciones grafos dirigidos (DAG), donde los volúmenes de cómputo y de comunicaciones, son las variables determinantes para implementar las políticas de gestión.

Para el caso de unidades de cómputo homogéneas, una revisión de la literatura a efectos de análisis, muestra que las técnicas existentes para la solución del problema planteado con pequeñas modificaciones, bien pueden ser utilizadas en el ámbito de un nodo, que es nuestro interés.

Dado que la ejecución de una instancia de un *workflow* puede no requerir todas las CPUs disponibles, surge la necesidad de considerar múltiples instancias, hecho que obliga a repensar las soluciones clásicas de planificación para su adaptación a la nueva situación de ejecución simultánea con múltiples instancias del mismo o de diferentes *workflows* [22]. Si bien hay numerosos estudios referentes a métodos de planificación para *workflows* [23], todavía hay mucho por hacer en el ámbito de la planificación para ejecuciones multi *workflow* [24,25,26].

Con el objetivo de asignar las aplicaciones que componen los *workflows* científicos a sistemas de núcleos heterogéneos [27, 28, 29, 30], es de vital importancia determinar el nivel de granularidad de las etapas del grafo de tareas que configuran el esquema del *workflow*, para su posterior ejecución concurrente. La aplicación de técnicas de clusterización a nivel de etapas, con efecto de mejorar el balanceo de carga cuando consideramos la ejecución paralela de múltiples instancias de un *workflow*, será un elemento clave y pieza importante de las soluciones aportadas.

Existen actualmente múltiples referencias en la literatura que recogen las principales ventajas y aportes de incorporar las unidades de cómputo GPU en las aplicaciones científicas [31]. Campos como el que nos ocupa en este trabajo en el ámbito del procesamiento de resonancias magnéticas, o bien la genética o la biologia han incorporado ya esta dualidad CPU/GPU en las recientes versiones. Esto es debido a que ya sea mediante tarjetas de gama doméstica, mayoritariamente empleadas en la industria del videojuego, como las de gama professional orientadas al cómputo, el rendimiento que se obtiene en términos de productividad es muy relevante. En algunos artículos se habla de mejoras del 50-60% o superiores respecto a las

ejecuciones con una sola CPU [32].

No obstante, el uso de las tarjetas GPU para el cómputo científico no está exento de problemas ya conocidos y detallados en la literatura. Por una parte nos encontramos que la memória que disponen las tarjetas GPU es de un orden de magnitud inferior a la que disponen los sistemas *multicore*. En el caso que el tamaño de los datos es inferior a la memoria disponible en la GPU, esta suele encargargarse de realizar todo el procesamiento, però por contra si el tamaño de los datos no cupieran en memoria, esto obliga a una gestión más fina mediante la generación de un algoritmo de planificación y un particionado de los datos [33].

Por otra parte otro problema frecuente que nos encontramos es que las GPUs suelen ser recursos compartidos entre varios núcleos e incluso nodos de cómputo. Esto hace que la planificación del uso de la tarjeta GPU quede en manos de una política FIFO (First In, First Out) para el procesamiento y priorización de las tareas. En un escenario donde dos aplicaciones o *workflows* accedan a una tarjeta en modo compartido, se deben incluir mecanismos de ordenación y prioridad de las tareas a ejecutar en función del uso del recurso que hagan. A un nivel más bajo, en la literatura encontramos algunas propuestas para la gestión y planificación de las tareas GPU, también denominados *kernels GPU,* en este contexto de uso compartido [34].

La aplicación de heurísticas basadas en la ordenación de las listas de prioridad de las aplicaciones para generar una mezcla adecuada, así como la inclusión de un método estático de predicción de rendimiento, constituyen el núcleo central del trabajo que se presenta y la singularidad del mismo teniend o en cuenta la gesitón compartida de los recursos GPUs, en relación a las políticas de planificación propuestas en la literatura.

Según nuestro conocimiento, este es el primer trabajo que considera la planificación de múltiples aplicaciones derivadas de la ejecución de múltiples instancias de un *workflow* real de procesamiento de resonancias magnéticas, FreeSurfer, en una arquitectura heterogénea multi/ many-core en un único nodo.

3 Ejecución planificada de un *workflow* en un entorno híbrido

En este capítulo se realiza un análisis del *workflow* de FreeSurfer utilizado para el procesamiento de resonancias magnéticas estructurales, que utilizaremos como objeto de estudio. El objetivo del análisis es describir el funcionamiento del *workflow* y las relaciones de dependencia entre las etapas y extraer las potencialidades que ofrece el *workflow* para poder paralelizarlo.

Por otro lado, se analiza la ejecución de múltiples instancias de *workflow* para detectar conflictos, problemáticas e ineficiencias. Introducimos también en este punto un primer análisis de rendimiento en cuanto a tiempo de ejecución, al añadir la ejecución de las etapas que tienen compatibilidad con GPU. Finalmente, se presentan unas primeras aportaciones con el objetivo de aumentar el *throughput* de la ejecución del *workflow*.

3.1 Configuración del nodo de computación

Los resultados presentados en el presente capítulo de esta tesis, hacen referencia a un entorno experimental con la siguiente configuración *hardware*:

- Se ha utilizado una estación de trabajo DELL XPS 730 con cuatro núcleos de cómputo, (2 chips dual-core Intel Core 2 Extreme con procesadores a 3,67 GHz), 12 MB de Caché L3 y 8 GB de memoria RAM.

- La tarjeta GPU utilizada es una NVIDIA GeForce 480 GTX con 15 Stream Multi-processors a 1.401 MHz, 1,5 MB de memoria GDDR5 y compatibilidad CUDA 2.0.

Se ha utilizado el siguiente *software*:

- Sistema Operativo CentOS 5.6 a 64 bits.
- *Driver*s NVIDIA 4.0 para Linux con soporte CUDA.
- CUDA Toolkit 3.2 y el CUDA *Software* Development Kit para Linux.
- FreeSurfer 5.0, compilado con soporte de CUDA 2.0.

Todas las pruebas se han hecho con un conjunto de imágenes compuesto por 100 resonancias magnéticas T1-MPRAGE3D, adquiridas con un escáner Philips Achieva 3T que se encuentra en el Hospital de la Santa Creu i Sant Pau en Barcelona. Las MRIs, pertenecen a un estudio de Parkinson dividido en 19 sujetos de control sanos, 37 sin afectación cognitiva, 24 con afectación cognitiva media y 20 con Demencia.

3.2 Análisis funcional del *workflow* de FreeSurfer

FreeSurfer se compone de varias herramientas (editores, visores y *workflows* para el análisis estructural o funcional de las resonancias magnéticas), y a su vez cada uno de estos *workflows* contienen *subflows* que permiten agrupar conjuntos de etapas.

Además, FreeSurfer incorpora versiones optimizadas para GPU en varias etapas, esto permite, en función de nuestro interés, poder ejecutar la misma etapa utilizando sólo la CPU o bien utilizar también la GPU, sólo especificando el parámetro de entrada "use-gpu”

Una de las primeras decisiones que hay que analizar es el nivel de detalle con el que podremos trabajar o incidir en el *workflow*. Esto es importante para dimensionar las propuestas de mejora que se puedan formular.

FreeSurfer se distribuye compilado en su versión destinada a la investigación, pero también está disponible el código fuente en los repositorios para desarrolladores. FreeSurfer tiene un código muy extenso, que entrelaza muchas llamadas a funciones y procedimientos, o bien enlaces con librerías, lo que complica su comprensión y su seguimiento. Es por ello que nos proponemos tratar FreeSurfer como una caja negra y trabajar a partir de cada una de las etapas que componen el *workflow* para mejorar el *throughput* de su ejecución. El script que mediante el cual se realizan las llamadas a las diferentes etapas nos aporta flexibilidad a la hora de reordenar y gestionar la ejecución del *workflow* sin la necesidad de realizar ninguna modificación funcional al código de FreeSurfer.

Esta situación es frecuente en los *workflows* científicos debido a que tienen una organización modular con múltiples y complejos cálculos a realizar. La gestión de las etapas habitualmente se realiza mediante un script principal, pero también existen herramientas [35] como Taverna [10] o Galaxy [11] que permiten la implementación de *workflows* con una gran flexibilidad a la vez que permiten realizar la ejecución de *workflows* masivamente con muchos datos de entrada.

El *workflow* completo de análisis de resonancias estructurales se compone de 37 etapas de procesamiento que se ejecutan secuencialmente. El *workflow* tiene como entrada una resonancia

magnética, que supone un conjunto de imágenes/cortes, con las que genera una única imagen/volumen 3D. Este representa el paso previo al inicio del *workflow* y la imagen resultante es la imagen de entrada a la primera etapa del *workflow*. A partir de aquí, la salida de una etapa constituye los datos de entrada a la siguiente etapa, junto con otros archivos propios de FreeSurfer (plantillas, archivos de coordenadas, archivos de etiquetas ...), que contienen datos de sólo lectura.

En la figura 3.1 podemos ver una abstracción del esquema original del *workflow*. Destacan dos conjuntos de etapas que se repiten con diferentes datos de entrada y mostradas con un color diferente. En concreto, estas etapas hacen referencia a un análisis que se realiza para cada uno de los hemisferios del cerebro, de ahí las etiquetas LH (izquierda/Left) y RH (derecho/Right).

Figura 3.1: abstracción del esquema del *workflow* de FreeSurfer

3.3 Análisis de rendimiento del *workflow* de FreeSurfer

Analizaremos ahora el rendimiento del *workflow*, con la ejecución de una única instancia del *workflow* correspondiente al procesamiento de un único sujeto.

En la figura 3.2 se presentan los tiempos de ejecución de cuatro instancias del *workflow* completamente independientes, para cuatro sujetos diferentes. Como se puede observar, la duración de cada una de las ejecuciones varía en función de los datos de entrada. Incluso con los

mismos datos de entrada, misma resonancia, los tiempos de ejecución son diferentes. Esta variabilidad depende en algunos casos, como la segmentación, del número de iteraciones que necesitan algunos procedimientos para converger en un resultado con el margen de error mínimo necesario. En los análisis realizados en este trabajo siempre se consideran un mínimo de tres ejecuciones de cada experimentación para tener en cuenta posibles variaciones en el tiempo de ejecución. Con los datos obtenidos de cada repetición de la experimentación utilizamos el tiempo medio obtenido entre todas ellas.

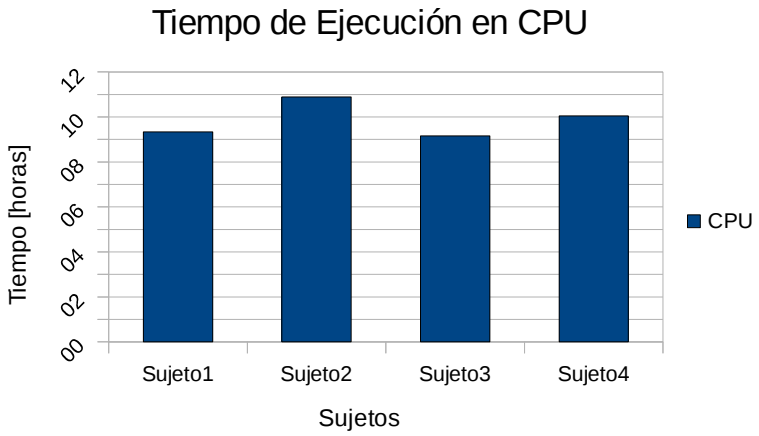


Figura 3.2: ejecución completa de una instancia del *workflow* por cuatro sujetos diferentes.

Es interesante conocer también cuáles son las partes del *workflow* que consumen más tiempo de ejecución. Estas son las partes en las que nos interesa más poder incidir para que la reducción de los tiempos de ejecución de cada una de estas partes por separado, supongan una mayor reducción del tiempo total de ejecución del *workflow*.

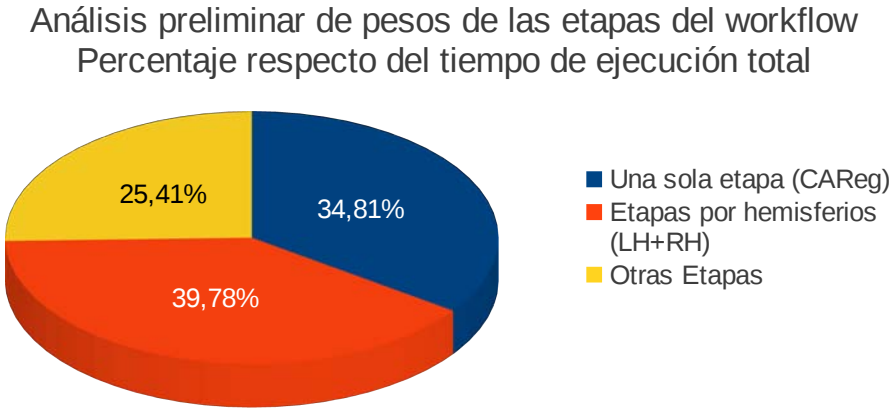


Figura 3.3: clasificación de las etapas del *workflow* en función de su tiempo de ejecución

En la Figura 3.3 se presenta la clasificación de las etapas del *workflow* que consumen más tiempo

de ejecución. Las medidas parten de la ejecución mostrada en la figura 3.2 y calculando los porcentajes medios de los cuatro sujetos. El *workflow* queda dividido en tres grandes partes. La primera, corresponde a un 34,81% del tiempo total de ejecución, y hace referencia a una única etapa (llamada CAReg) que tarda aproximadamente entre 3 y 3,5h de ejecución. La segunda parte hace referencia a un conjunto de etapas que se ejecutan para el procesamiento del hemisferio derecho e izquierdo. Esto supone aproximadamente un 39,78% del tiempo total de ejecución. El porcentaje restante, el 25,41% se refiere al resto de etapas que no pertenecen a las anteriores porciones graficadas.

Como ya se ha mencionado anteriormente, el *workflow* de FreeSurfer ofrece la opción de poder ejecutar algunas de estas etapas utilizando las GPUs. Estas implementaciones, que se introdujeron en la versión 5 de FreeSurfer, aceleran la ejecución de cuatro etapas del *workflow*. Concretamente, hay dos en la parte inicial del *workflow*, CAReg que es la que hemos destacado anteriormente en la figura 3.3, como la etapa que corresponde al 34,81% del tiempo de ejecución y SkullLTA. Las otras dos se encuentran dentro del procesamiento por hemisferios (Qsphere y Sphere) y que por tanto, se ejecutan dos veces cada una según el esquema original del *workflow* de FreeSurfer.

Para poder evaluar la incidencia de las GPUs en los tiempos de ejecución, se presentan a continuación los resultados de re-ejecutar los *workflows* correspondientes a los mismos sujetos mostrados en la figura 3.2, ahora utilizando las GPUs. Podemos observar como las GPUs tienen una importante incidencia en el tiempo de ejecución. Si bien la ejecución por los cuatro sujetos utilizando sólo la CPU está en torno a las 10 horas de media, una vez activadas las etapas que se ejecutan mediante CPU+GPU, el tiempo de ejecución medio pasa a estar en torno a las 5 horas, es decir, se reduce el tiempo aproximadamente a la mitad.

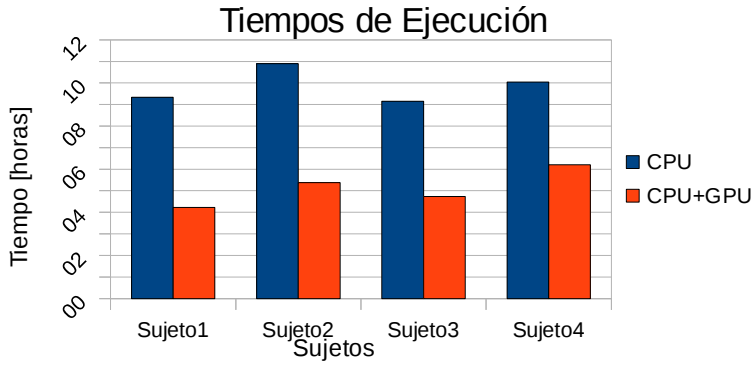


Figura 3.4: comparativa de los tiempos de ejecución CPU / CPU + GPU

Para concretar la incidencia de las etapas ejecutadas con las GPU, en la siguiente tabla se

muestran los tiempos de ejecución de cada una de las etapas que se pueden ejecutar tanto en CPU como en GPU. También se muestra la diferencia y la mejora (*speedup*) entre la ejecución utilizando la CPU, y la ejecución utilizando la GPU.

Etapa	**Duración media CPU [seg]**	**Duración media GPU [seg]**	***Speedup***
CA Reg	10589	1497	7,07
SkullLTA	1165	190	6,13
Qsphere	196	157	1,25
Sphere	3107	161	19,3

Tabla 1: tiempo de ejecución y mejora de las etapas tanto en CPU como en CPU+GPU

En la tabla 3.1 encontramos la duración para cada una de las etapas que se pueden ejecutar tanto en CPU como CPU+GPU. Podemos observar diferencias sustanciales entre las ejecuciones en CPU respecto las ejecuciones en CPU+GPU. La etapa CAReg va 7,07 veces más rápida, la segunda etapa SkullLTA 6,13 veces más rápida. La tercera etapa es la que menos ganancia obtiene (1,25), pero la última etapa se ejecuta 19.3 veces más rápida que su versión en CPU. A la vez que las GPUs inciden en el tiempo de ejecución de algunas de las etapas, también hacen variar los pesos de estas etapas respecto al tiempo total de ejecución del *workflow*.

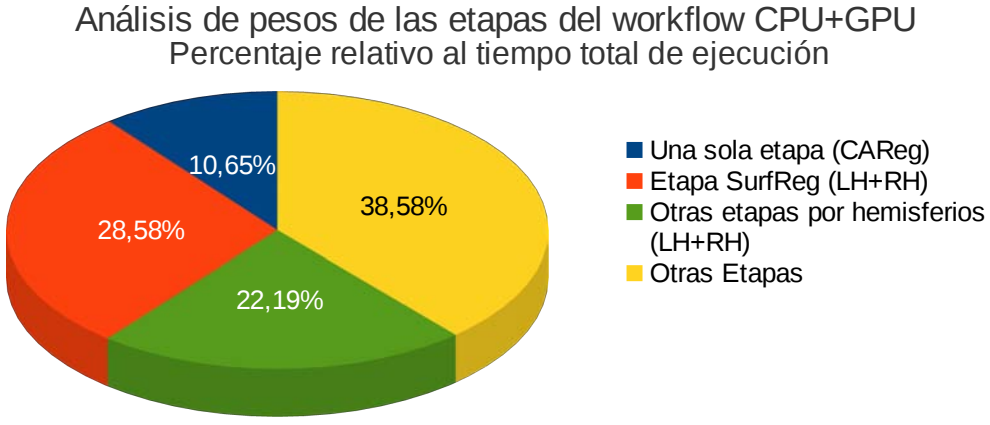


Figura 3.5: análisis de los pesos de las etapas con la ejecución del *workflow* CPU+GPU

En la figura 3.5 se presentan los pesos de las partes más relevantes del *workflow* una vez activado el uso de las GPUs. Los resultados se expresan en porcentaje respecto el tiempo de ejecución total. Si comparamos este gráfico con el de la figura 3.3 , vemos que la etapa (CAReg) que en CPU acumulaba un 34,81% del tiempo total, ahora, con el uso de las GPUs, llaga tan sólo a un 10,65%.

Por otro lado, la parte de las etapas que se ejecutan para cada uno de los hemisferios del cerebro (LH+RH) ahora pesa un 50,77%, los cuales se reparten en dos partes diferenciadas. La más importante se refiere a una única fase, SurfReg, que se ejecuta dos veces (una por hemisferio) y que pesa un 28,58% del tiempo total de ejecución. El 22,19% de la parte de hemisferios hace referencia a otras etapas.

Finalmente, destacamos el 38,58% Que hace referencia a las etapas previas al tratamiento de los hemisferios, así como la parte posterior, que ahora con la nueva distribución de pesos representa un ligero incremento de 13 puntos porcentuales aproximadamente con respecto a la distribución de pesos en el caso de la ejecución sólo con CPU.

Concluimos en este apartado que el uso de las GPUs hace una importante incidencia en el tiempo de ejecución de una única instancia del *workflow*, interviniendo en etapas costosas y reduciendo su tiempo de ejecución. Por otro lado vemos que la parte que procesa los hemisferios es ahora la parte mayoritaria en peso respecto al tiempo total de ejecución. En el siguiente apartado se analiza esta parte del *workflow* para analizar el paralelismo potencial.

3.4 Propuesta de esquema de ejecución

Como se ha dicho anteriormente, el *workflow* de FreeSurfer se compone de 37 etapas. Para simplificar el número de etapas, las agrupamos en bloques. El criterio de agrupación de etapas se basa en separar aquellas que se puedan ejecutar con la compatibilidad CPU+GPU. De esta manera nos resulta un esquema más sencillo, con 11 bloques o conjuntos de etapas en total.

3.4.1 Paralelismo Intra-*workflow*

Con la ejecución del *workflow* vemos que las etapas que hacen referencia al análisis por separado de cada uno de los hemisferios del cerebro son completamente independientes. Los archivos de entrada así como los generados por cada una de las etapas tienen la extensión de su nombre como .lh/.rh en función del hemisferio al que hacen referencia.

La gestión por separado de los archivos de entrada y salida, así como la estructura de programación nos demuestra que hay un paralelismo potencial intra-*workflow*. Si desplegamos el bucle de procesamiento por hemisferios, lo podemos convertir en dos hilos de proceso separados, uno para el hemisferio derecho y otro para el hemisferio izquierdo.

Del estudio previo de tiempo de ejecución podemos extraer que la ejecución de las dos etapas completas (ambos hemisferios), supone aproximadamente un 40% del tiempo total de ejecución,

cuando se hace completamente en CPU, y de un 51%, cuando hacemos una ejecución utilizando CPU + GPU.

En la figura 3.6a se presenta la abstracción del esquema original de FreeSurfer, donde destacan los conjuntos de etapas que se procesan para cada uno de los hemisferios. Al lado, la figura 3.6b hace referencia a la propuesta de esquema aprovechando el paralelismo *inter-workflow* donde ambos conjuntos de etapas para los hemisferios, se ejecutan en paralelo. El esquema resultante tiene una parte de etapas en serie, y una parte paralela que puede ser utilizada tanto para la ejecución utilizando sólo las CPUs, como para la ejecución híbrida donde se utilicen las CPUs+GPUs.

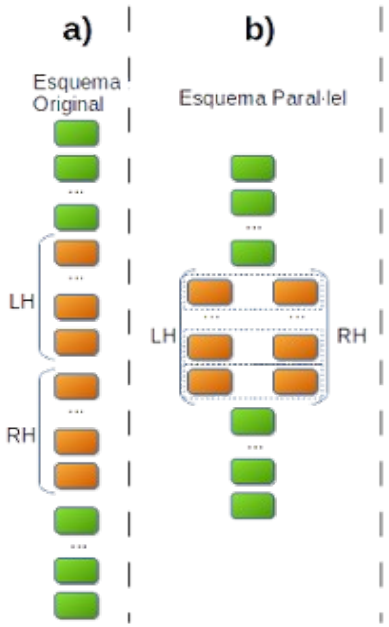


Figura 3.6: propuesta del esquema paralelo para el *workflow* de FreeSurfer

Si tenemos en cuenta un sistema de cómputo como el descrito en la sección 3.1 Configuración del sistema, disponemos de cuatro núcleos de cómputo CPU y una tarjeta GPU con 15 SMPS. En la figura 3.7 se presenta una aproximación de cómo se distribuyen los recursos de cómputo entre un conjunto de etapas del *workflow*, teniendo en cuenta las etapas que se ejecutan secuencialmente y las etapas que se ejecutan en paralelo. El diagrama de Gantt, muestra cómo para las etapas b1, b3, b5, b6 y b8 se utiliza un núcleo de cómputo CPU, mientras que los bloques b2, b4, b7 y b9 asignan un cierto número de núcleos de cómputo (SMPS) de la GPU del sistema. Vemos también cómo cada tarea GPU ocupa un núcleo CPU: esto se debe a que cada tarea GPU tiene una tarea de control CPU que es la encargada de gestionar la llamada a los *kernels* (funciones) de GPU.

Las etapas que aplican el paralelismo intra-*workflow*, resultan ser dos copias de la misma etapa

cambiando los datos de entrada, teniendo en cuenta los datos del hemisferio izquierdo y derecho respectivamente. Esto hace que estas etapas demanden un núcleo de cómputo CPU para cada una, a la vez que en los casos que se utilice la GPU se doblen los requerimientos de SMPS.

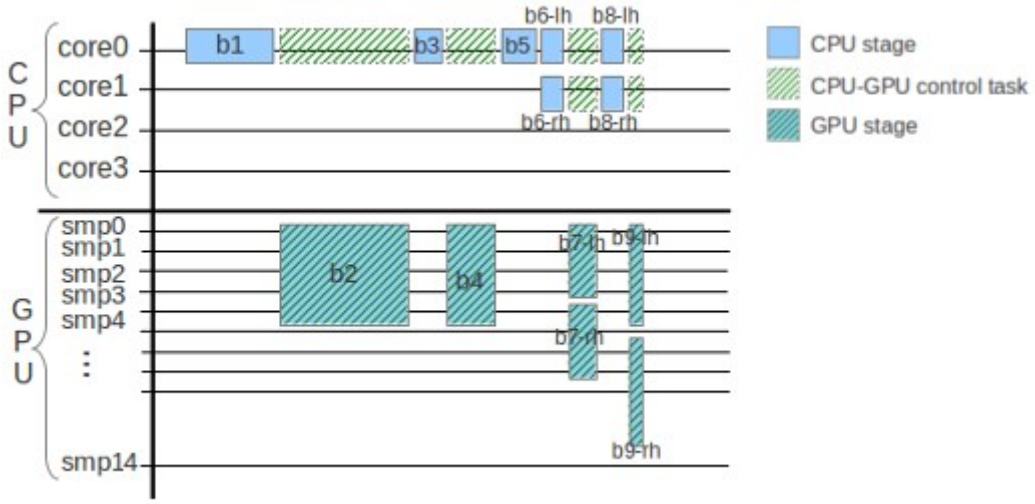


Figura 3.7: representación aproximada del uso de los recursos utilizando el paralelismo intra-*workflow*.

El tiempo de ejecución de un *workflow* de FreeSurfer se reduce respecto al esquema completamente serial gracias a la ejecución simultánea de las etapas de los hemisferios.

En la figura 3.8 se presenta la comparativa de los tiempos de ejecución para la ejecución de cuatro sujetos en los escenarios donde se utiliza las implementaciones CPU o las CPU+GPU (el mismo que en la figura 3.4) y en este caso se añade el escenario de ejecutar los *workflows* completos con CPU+GPU+paralelismo intra-*workflow*. Vemos que en todos los casos el uso del paralelismo intra-*workflow* aporta una reducción adicional al tiempo de ejecución del *workflow*, aunque la reducción de tiempo más importante se consigue con el uso de las implementaciones en GPU.

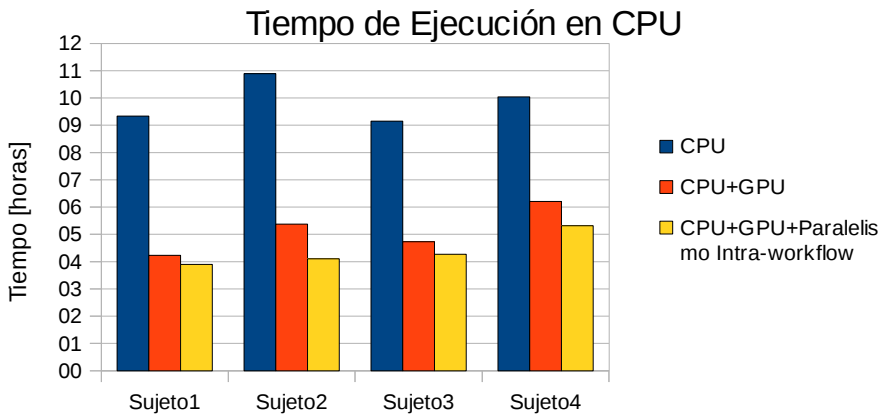


Figura 3.8: comparativa de los tiempos de ejecución CPU/CPU+GPU/CPU+GPU+intra-*workflow*

A pesar de que con la ejecución del *workflow* mediante el esquema paralelo (intra-*workflow*) doblamos los recursos CPU y GPU que usamos, aún quedan recursos disponibles en el sistema que pueden ser aprovechados. Por ello, proponemos una estrategia para ejecutar múltiples instancias de *workflow*, para diferentes sujetos de entrada, con el objetivo de utilizar el máximo de recursos disponibles, a lo que llamaremos paralelismo *inter-workflow*.

Para completar el análisis medimos el aporte de los escenarios donde la ejecución combina el uso CPU+GPU, así como la introducción del esquema paralelo. Para ello, tomamos como referencia las ejecuciones para los cuatro sujetos usados anteriormente y calculamos el *speedup* respecto a la ejecución serie que se realiza en el escenario que usa únicamente la CPU.

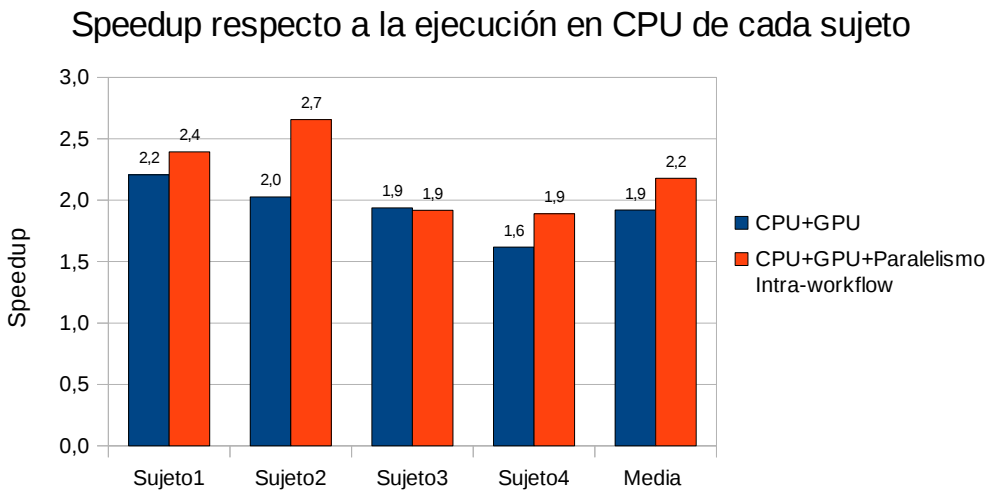


Figura 3.9: *speedup* obtenido en los escenarios CPU+GPU respecto a la ejecución CPU

En la Figura 3.9 se muestra el *speedup* obtenido en los escenarios CPU+GPU y CPU+GPU+Esquema paralelo respecto a la ejecución en CPU. El *speedup* se calcula para varias ejecuciones del *workflow* de FreeSurfer con diferentes datos de entrada (sujetos).

Comparativamente en cada una de las ejecuciones para cada sujeto, el mayor aporte lo da la introducción de la GPU, con una mejora de cerca del 50% de media. Es a partir de la introducción del esquema paralelo que la mejora se incrementa de media al 55% respecto a la ejecución CPU. En el peor de los casos, el *Speedup* se mantiene en ambos escenarios.

3.4.2 Paralelismo *Inter-workflow*

La ejecución de múltiples instancias es de interés en la mayoría de ámbitos científicos, debido a que requiere realizar computación masiva de datos. Esto se da en el caso real de aplicación de FreeSurfer, debido a que la ejecución se realiza para múltiples sujetos de entrada, constituyendo el escenario habitual en los proyectos de investigación en el ámbito de las neurociencias.

Si no tuviéramos en cuenta las limitaciones de nuestro sistema de cómputo en ejecutar K instancias idénticas del *workflow* de FreeSurfer con diferentes sujetos de entrada, tendríamos una perfecta escalabilidad y el tiempo total de ejecución de las K instancias rondaría en un factor de 1/K, comparado con la ejecución de una única instancia del *workflow*.

Ahora bien, los sistemas no tienen recursos infinitos y por lo tanto nos centramos en un caso de estudio con la ejecución de K=2 instancias idénticas en nuestro sistema de cómputo. Este valor de K nos permite llegar a utilizar los 4 procesadores en el conjunto de etapas paralelas, aunque todavía haya recursos infrautilizados, constituyendo aún así un buen caso de análisis del comportamiento de la ejecución paralela *inter-workflow*.

En la figura 3.10 observamos un diagrama similar al presentado anteriormente en la figura 3.7. En este caso, se representa la aproximación del uso de recursos para dos instancias de *workflow* de FreeSurfer. La notación es similar a la anterior figura, donde las etapas están etiquetadas por b1, b2 ... b8-lh o b8-rh, pero incluyendo ahora el prefijo s1 o s2 en función del sujeto relacionado con cada instancia.

Las dos instancias en paralelo utilizan el doble de recursos, asignándose un núcleo de cómputo CPU para cada etapa, al igual que para las etapas donde se utilizan las GPU, también se utilizan aproximadamente el doble de los SMPS. Observamos también como en el conjunto de etapas para hemisferios se utilizan los cuatro núcleos de cómputo. La distribución de los núcleos de cómputo queda, en este caso, a decisión del Sistema Operativo.

Por otra parte, es interesante explicar cómo se gestiona la GPU. Las tarjetas NVIDIA tienen una interfaz de gestión llamada *NVIDIA System Management Interface* [36] que permiten gestionar la configuración y obtener información de todas las GPUs mediante la interacción con el *driver* de las tarjetas. Concretamente, en nuestro escenario hemos tenido que tener en cuenta el parámetro llamado *Compute Node* el cual nos permite definir si la GPU está en modo compartido o en modo exclusivo.

El modo establecido es el compartido, la misma GPU podrá ser utilizada con varios procesos simultáneamente y será el *Driver* quien vaya gestionando los *kernels* que se ejecutan por los diferentes procesos. Si por el contrario, el modo elegido es exclusivo, la tarjeta sólo podrá ser utilizada por un único proceso a la vez.

Vemos que la ejecución de dos *workflows* simultáneos es muy parecido al de un único *workflow*, teniendo en cuenta el diferente uso de los recursos. En la parte baja de la figura 3.10, se ha graficado el uso aproximado de memoria GPU durante la ejecución de múltiples etapas que utilizan la GPU. Vemos que las dos etapas b2 ocupan aproximadamente de 1100 a 1200 MB de

memoria de la GPU como pico de la gráfica. Esto supone, y se ha comprobado empíricamente, que una tercera etapa b2 no se puede ejecutar por falta de memoria.

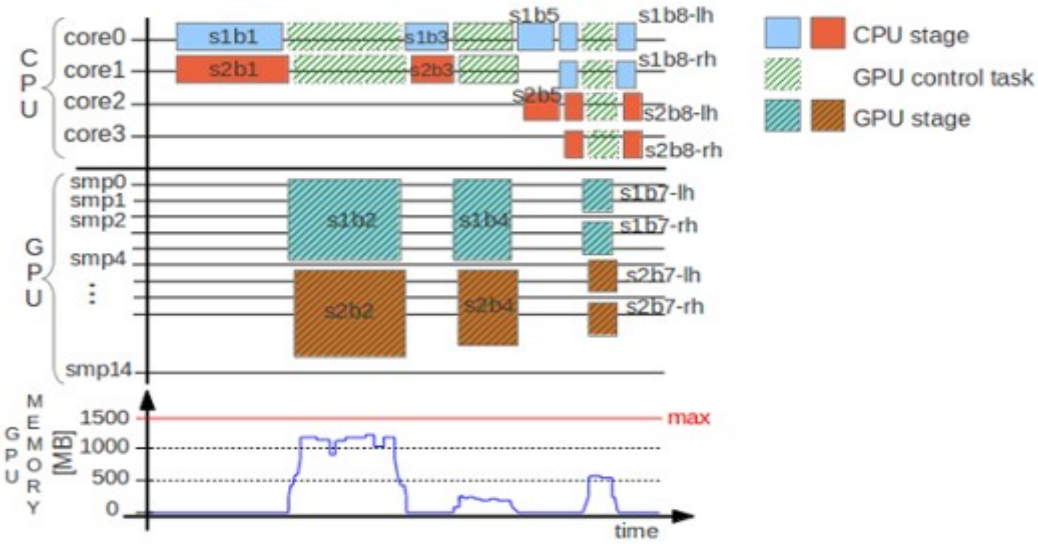


Figura 3.10: representación aproximada del uso de los recursos utilizando el paralelismo *inter-workflow*. Se muestra, tanto los recursos de cómputo como la capacidad de memoria que se utiliza en la GPU.

Se demuestra pues, que en nuestro escenario, la memoria de la GPU es un recurso limitante, y por tanto, no podemos garantizar la ejecución de más de dos *workflows* simultáneos compartiendo la misma GPU. Por otro lado también vemos que ni el *workflow* de FreeSurfer tiene mecanismos para gestionar la falta de memoria, ni tampoco el *driver* o el sistema operativo pueden manejar esta situación. La ejecución queda parada y por lo tanto se rompe la dependencia de datos para el bloque b3, provocando el error en cadena de todas las etapas posteriores.

Las problemáticas abiertas en este punto son principalmente las ineficiencias que se presentan en nuestro sistema para no ocupar todos los recursos disponibles; así como la falta de control en la gestión de los recursos y el problema de limitación del número de ejecuciones simultáneas, debido a la limitación de la memoria de la GPU. El objetivo es poder tener un sistema con un mayor *throughput*, medido en sujetos por unidad de tiempo.

3.4.3 Paralelismo mediante un planificador

Para solucionar los problemas descritos en la anterior sección, hemos desarrollado un planificador centralizado para poder ejecutar múltiples sujetos simultáneamente. Este planificador gestiona la principal limitación detectada, como es el uso de memoria de la GPU.

El planificador centraliza la gestión de todos los sujetos de entrada y ejecuta las etapas definidas

en el esquema paralelo antes descrito para el *workflow* de FreeSurfer. La planificación se realiza mediante una política que asigna recursos disponibles en función de las tareas pendientes de ejecución, intentando conseguir el máximo rendimiento posible.

La estructura del planificador se presenta en la figura 3.11. Con el fin de reducir el overhead de gestión de etapas, el planificador trabaja a partir del esquema paralelo que agrupa las 37 etapas iniciales del *workflow* de FreeSurfer en 11 bloques, en función de dos criterios: la dependencia de datos y la disponibilidad de las implementaciones CPU+GPU. Estos bloques representan la unidad mínima (una etapa formada por subetapas) y se etiquetan con b1 hasta b11.

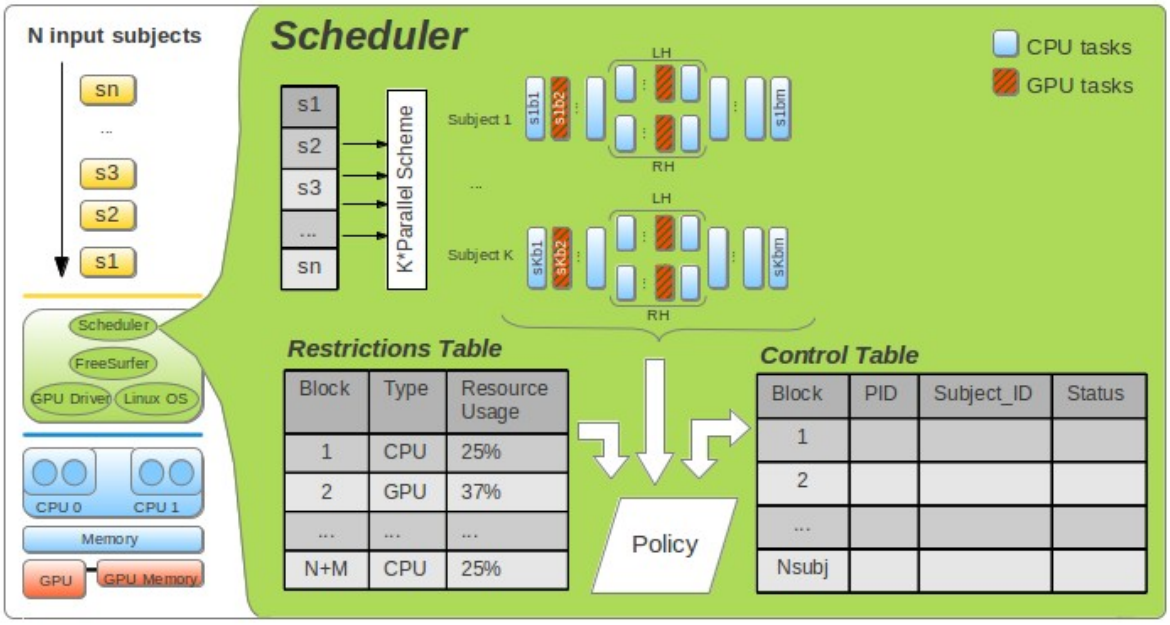


Figura 3.11: esquema y componentes del planificador multi-*workflow*

El planificador recibe tres parámetros de entrada: una lista con los sujetos a procesar (N sujetos), el número de tareas simultáneas que define un tamaño de ventana con el número de *workflows* que se pueden ejecutar (K) y una tabla de restricciones que contiene informaciones sobre el recurso limitante (CPU o GPU) y el porcentaje estimativo del uso del recurso para cada instancia.

El porcentaje de uso del recurso CPU para cada bloque se asigna a partir de la división del 100% del recurso entre K. De esta manera, el planificador permite la ejecución de K CPU bloques simultáneamente. En el caso de la GPU, el porcentaje de uso lo calcula el usuario a partir del tamaño máximo de memoria utilizada por cada bloque, tomando medidas empíricas en un conjunto de sujetos de prueba. De esta forma se puede asegurar que no habrá conflictos en el uso y asignación de recursos en cada etapa.

El planificador hace un seguimiento del uso de los recursos, a partir de la suma acumulada de los porcentajes de uso de cada tarea en ejecución, tanto por CPU como por GPU. Esta acumulación nunca debe superar el 100% del uso del recurso.

La tabla de control es utilizada por el planificador para almacenar información sobre el estado de la ejecución de los diferentes bloques. Un bloque puede ser uno de estos cuatro estados:

1. «No comenzado» - estado inicial de un bloque.
2. «En ejecución» - se asigna cuando un bloque puede ejecutarse.
3. «En espera» - se asigna cuando un bloque requiere más recursos de los disponibles.
4. «Finalizado» - se asigna cuando un bloque finaliza la ejecución.

3.5 Resultados

Un bloque pasa al estado de espera cuando necesita más recursos de los disponibles. La lista de todos los bloques en espera es comprobada periódicamente por el planificador, que a su vez consulta la lista de sujetos pendientes de iniciar, la tabla de restricciones y los recursos disponibles en el momento. El planificador selecciona el bloque que puede pasar al estado de ejecución, priorizando siempre los sujetos que están en espera, en contra de los sujetos de la lista aún no iniciados. De esta manera, el planificador de manera dinámica maximiza el uso total de los recursos.

En esta sección se presentan los resultados obtenidos por un conjunto de pruebas realizadas en los diferentes escenarios planteados a lo largo de la sección 3 con diferentes números de ejecuciones simultáneas.

El uso del planificador soluciona el problema de la capacidad de la memoria de la GPU, permitiendo que el número de ejecuciones simultáneas supere (K>2) y podamos obtener una plena explotación del paralelismo inter e intra *workflow*.

3.5.1 Comportamiento del Planificador

La figura 3.12 presenta unas trazas de Gantt con las etapas iniciales de la ejecución del *workflow* de FreeSurfer mediante el planificador que hemos descrito en la sección 3.5 para cuatro sujetos simultáneos (K = 4). La notación es una abreviación respecto las anteriores figuras 3.7 y 3.10. Las tareas de control CPU así como el número de núcleos de cómputo o SMPS se han omitido, en pro de mostrar el comportamiento de las etapas.

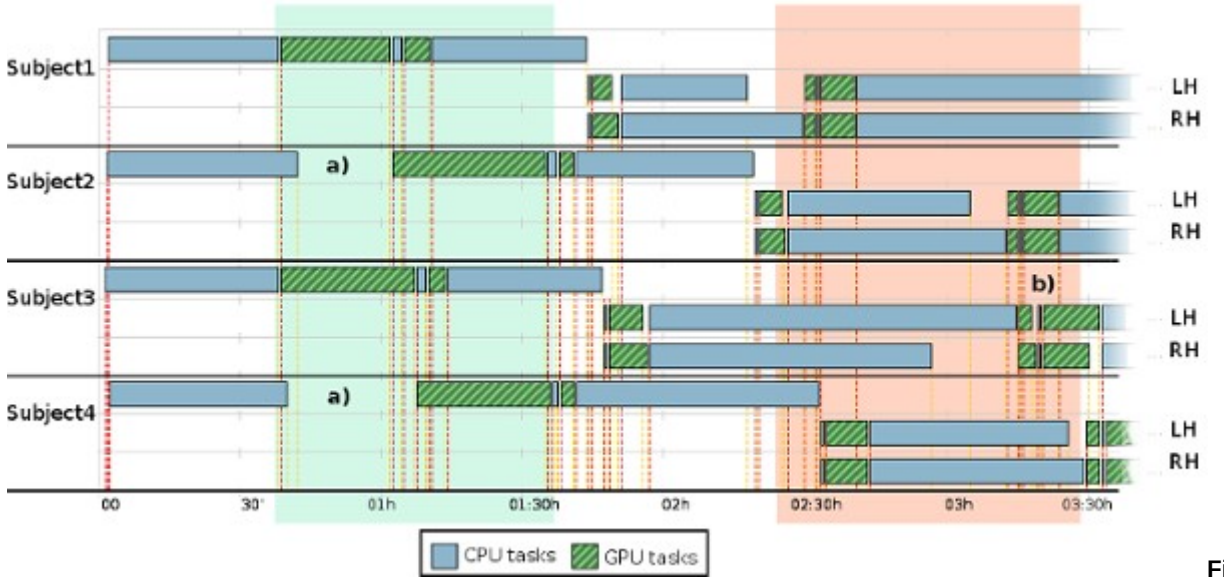


Figura 3.12: traza de Gantt para la ejecución simultánea de cuatro instancias del *workflow* de FreeSurfer

Fig

Inicialmente los cuatro sujetos comienzan a la vez ejecutándose en paralelo, cada uno tiene asignado un núcleo de cómputo CPU. La segunda etapa de procesamiento requiere del uso de la GPU y como habíamos visto en la figura 3.10 un gran cantidad de memoria GPU. El planificador sólo permite ejecutar dos etapas GPU simultáneamente, para gestionar las restricciones de memoria especificadas en la tabla de restricciones. Las otras dos etapas GPU, pasan al estado de espera mientras no haya recursos disponibles (figura 3.12, etiqueta a). Por ejemplo, cuando la primera etapa GPU correspondiente al sujeto 1 termina, la tarea GPU del sujeto 2 comienza su ejecución a petición del planificador.

El planificador también puede manejar otro situaciones más complicadas con múltiples bloques ejecutándose simultáneamente (figura 3.12, etiqueta b). En estos casos, las etapas consumen poca cantidad de recursos por lo que el planificador permite la ejecución de más tareas que núcleos de cómputo disponibles.

3.5.2 *Speedup* de la ejecución de múltiples *workflows*

Hemos ejecutado varios experimentos con diferente número de sujetos (N), y diferente número de ejecuciones simultáneas (K). Con el ánimo de recopilar la variabilidad de los resultados que dependen del orden en el que se ejecutan los sujetos, hemos realizado experimentos con múltiples combinaciones de los N sujetos y hemos extraído medidas medias del tiempo totales de ejecución. Hemos calculado la productividad media, medida en sujetos por hora y la productividad relativa, o *SpeedUp*, utilizando la ejecución de una única instancia del *workflow* en CPU y completamente serial, como caso base para la normalización.

La tabla 3 resume los datos obtenidos en las diferentes pruebas. Los escenarios probados son:

1. Ejecuciones seriales de las etapas (con y sin GPUs)
2. Introducción del esquema paralelo intra-*workflow* (sección 3.4)
3. Introducción de la ejecución paralela *inter-workflow* (sección 3.4), con limitación de K = 2 instancias de *workflow*, debida a la capacidad de memoria GPU.
4. Introducción de la paralelización inter e intra *workflow*, mediante el planificador (sección 3.5), para resolver las limitaciones de memoria GPU.

Escenario	**N**	**K**	**Tiempo fin experimento [seg]**	**Tiempo por sujeto [horas]**	**Productividad**	**Product. Normalizada**
Serial CPU	6	1	220866	10,23	0,098	1
Inter paralelo CPU-solo	6	2	140685	6,51	0,154	1,57
Inter paralelo CPU-solo	6	3	105314	4,88	0,205	2,10
Inter paralelo CPU-solo	6	4	84255	3,90	0,256	2,62
Inter paralelo CPU-solo	6	6	85431	3,96	0,253	2,59
Inter paralelo CPU-solo	8	8	115428	4,01	0,250	2,55
Inter paralelo CPU-solo	10	10	134342	3,73	0,268	2,74
Intra paralelo CPU-solo	6	1	196272	9,09	0,110	1,13
Inter+Intra-workfl. CPU-solo	2	2	37113	5,15	0,194	1,98
Serial CPU+GPU	4	1	110596	7,68	0,130	1,33
Inter paralelo CPU+GPU	6	2	51462	2,38	0,420	4,29
Intra paralelo CPU+GPU	4	1	65128	4,52	0,221	2,26
Intra+Inter-worfl CPU+GPU	2	2	34308	2,38	0,420	4,29
Planificación de recursos	6	3	38984	1,80	0,554	5,67
Planificación de recursos	6	4	36035	1,67	0,599	6,13
Planificación de recursos	6	6	33050	1,53	0,654	6,68
Planificación de recursos	8	8	43361	1,51	0,664	6,79
Planificación de recursos	10	10	57968	1,61	0,621	6,35

Tabla 2: resultados de los tests en diferentes escenarios de ejecución

La figura 3.13 muestra la evolución del *Speedup* en función de las K ejecuciones simultáneas. En el escenario de uso CPU, y utilizando el sistema operativo Linux como planificador por defecto, la productividad se estanca en K ≥ 4. Este resultado indica que los cuatro núcleos de cómputo son un cuello de botella. Hemos medido el consumo de memoria y el uso del disco no representa un problema. La variación en el rendimiento por K ≥ 4 se debe a los diferentes sujetos de entrada, la contención irregular entre las tareas concurrentes compartiendo memoria y los recursos computacionales y la desigual cantidad de trabajo pendiente en la fase final de cada ejecución.

El escenario CPU+GPU junto con la política de planificación del planificador (sección 3.5), permite explotar mejor el paralelismo *inter-workflow* hasta K = 6 ejecuciones simultáneas, permitiendo incrementar el rendimiento. Dado que la carga de trabajo se reparte entre la CPU y la GPU, hay que hacer un análisis más en profundidad para identificar cuellos de botella del sistema. Utilizar diferente *hardware*, determinará un valor de K diferente donde el rendimiento se estanque. En este ejemplo, la caída de la productividad por K > 10 es un efecto de la variabilidad descrita anteriormente.

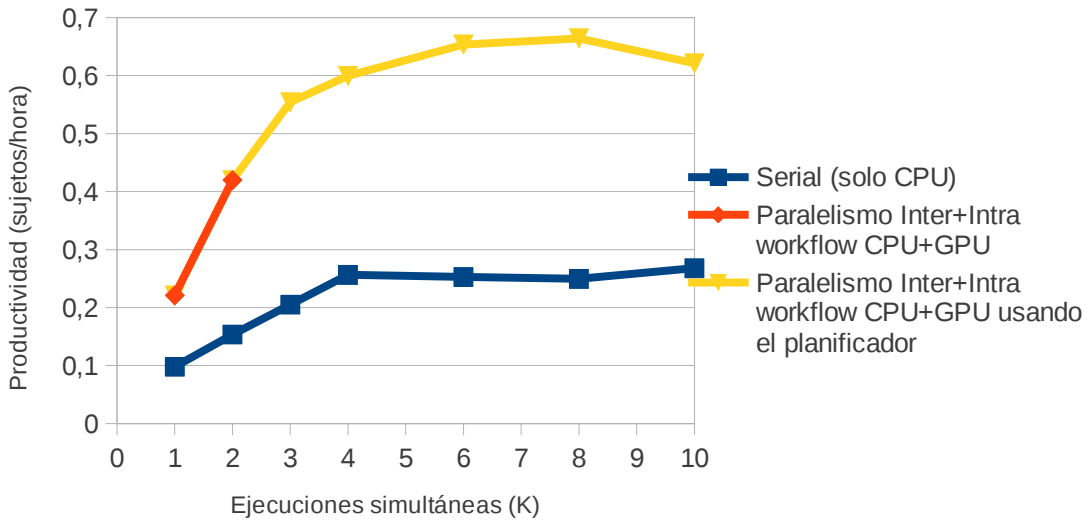


Figura 3.13: *speedup* para los escenarios, serial, paralelo y planificado

La tabla 3.3 recoge los mejores resultados obtenidos por cada esquema. Utilizando el modesto entorno de pruebas antes descrito, con suficientes sujetos a procesar, el mejor escenario proporciona una productividad de 0.664 sujetos por hora. La GPU proporciona el mejor rendimiento en 1.33 veces en el escenario serial, pero proporciona una mejora 2 veces cuando utilizamos paralelismo intra- e inter- *workflow*. El planificador va más allá y mejora el rendimiento hasta el 6,79.

Esquema	Productividad CPU-solo	Productividad CPU+GPU	CPU-solo *speedup*	CPU+GPU *Speedup*
Serial	0,098	0,130	1	1,33
Intra-paralelo	0,110	0,221	1,13	2,26
Inter+Intra paral. (K=2)	0,194	0,420	1,98	4,29
Planificación de recursos (K=8)	-	0,664	-	6,79

Tabla 3: productividad máxima (sujetos/hora) y el *speedup* en los diferentes escenarios

3.5.3 Conclusiones

La solución de planificación propuesta para la ejecución de múltiples *workflows* está basada en la explotación del paralelismo intra+inter *workflow*, con la gestión de restricciones para la ejecución de tareas en las GPUs. Esta estrategia aporta una mejora del *throughput* del procesamiento de resonancias magneticas (MRIs) de más de seis veces comparada con el esquema de ejecución serie y de 2,5 veces comparada a la versión serial acelerada con el uso de las GPUs.

Este planificador es lo suficientemente genérico para poderse utilizar en otros *workflows* científicos parecidos. Una de las principales ventajas es que gestiona las agrupaciones de procesos o tareas del *workflow*, que hemos denominado bloques de tareas y que representan etapas del *workflow*, sin necesidad de modificar el codigo fuente de la aplicación. Esto facilita la adaptación del planificador a cualquier otro *workflow* de estudio, solamente se requiere definir las dependencias y los recursos necesarios para cada etapa.

Hemos identificado un problema con la gestión de la memoria GPU para ciertas etapas del *workflow*, con la ejecución de más de dos instancias simultáneas del *workflow*. El planificador diseñado resuelve este problema mediante una tabla de restricciones, que gestiona con una política que explota el máximo paralelismo para mantener los recursos ocupados el máximo tiempo posible.

La implementación del planificador presentada en este capítulo tiene algunas limitaciones. Por una parte, la política no tiene en cuenta ni el rendimiento ni el comportamiento de las etapas en la planificación, por lo que la mezcla de etapas de las diferentes instancias que se producen pueden provocar ineficiencias que perjudiquen el rendimiento global de la ejecución. Por otra parte, solo se planifican tareas en una única GPU del sistema. La resolución de las limitaciones expuestas, puede aportar aún un margen de mejora del *throughput* de las ejecuciones.

4 Ejecución planificada *por lotes*

En la primera parte de este trabajo se ha propuesto un esquema de ejecución planificada de múltiples instancias de un *workflow* híbrido (que usa tanto CPUs como GPUs), que resuelve los problemas y restricciones que se presentan en el uso de los recursos, en especial el uso de la GPU. Esta ejecución planificada permite ejecutar múltiples instancias de forma segura y de esta manera se aumenta la productividad (*throughput*) del sistema, medida en sujetos procesados por hora.

Analizando con más detalle el trabajo descrito en el capítulo anterior aplicado al *workflow* de procesamiento de imágenes de resonancias magnéticas llamado FreeSurfer, nos damos cuenta de que los recursos de cómputo (CPU y GPU) no están siendo utilizados de forma eficiente durante una considerable parte del tiempo de ejecución, y que, por tanto, existe un importante margen de mejora para el rendimiento o productividad del sistema. En este capítulo propondremos y analizaremos un esquema más flexible para la ejecución planificada de las tareas correspondientes a múltiples instancias de un *workflow*. Basándonos en un análisis empírico inicial de las características de la ejecución de estas tareas, y del efecto de la ejecución combinada de tareas de diferente tipo, propondremos unas políticas que logren un uso eficiente tanto de las CPUs como de las GPUs del sistema, y un buen balance entre la carga asignada a las CPUs y la carga asignada a las GPUs. A continuación, se propondrá y describirá con detalle un mecanismo de planificación con listas separadas asociadas a diferentes conjuntos de recursos, que incorpora un mecanismo de equilibrado de la carga de trabajo. Finalmente, se evaluarán las mejoras en el rendimiento que se logran con nuestras propuestas.

4.1 Definición de la nueva propuesta

En este apartado consideramos un escenario de ejecución del *workflow* de FreeSurfer (generalizable a cualquier otro *workflow* similar) en el que se necesitan procesar los datos correspondientes a centenares o miles de sujetos. Por tanto, se dispone de un amplio grado de paralelismo que nos otorgará una gran flexibilidad para planificar de forma adecuada la ejecución de las tareas correspondientes a las etapas del *workflow*. Nuestro objetivo de rendimiento es maximizar la productividad (*throughput*) del sistema, medida en sujetos procesados por unidad de tiempo.

La figura 4.1a muestra el esquema de ejecución serie: se procesa completamente cada sujeto de forma secuencial, sin ningún tipo de paralelismo. Este esquema de ejecución representa el caso base y es el que nos servirá para comparar las mejoras de rendimiento que se obtienen en las propuestas presentadas en esta tesis. Los diferentes sujetos de entrada se representan como s1, s2, … Para cada uno de los sujetos se ejecuta un *workflow* de n tareas que corresponden con las etapas T1, T2, ... Tn. En lo que sigue, cuando hagamos referencia a etapas del *workflow* queremos indicar las tareas correspondientes a cada etapa. Se debe recordar que cada etapa puede corresponderse con uno o más procesos de cómputo, y que estos procesos siempre se ejecutan en CPU, pero pueden requerir el uso adicional de una GPU. Cuando hablamos de tareas de GPU se sobreentiende, por tanto, que requieren la ejecución de procesos que también usan la CPU. Asimismo, nos referiremos a un *workflow* o a una instancia de *workflow* para indicar las tareas necesarias para procesar los datos correspondientes a un determinado sujeto de entrada.

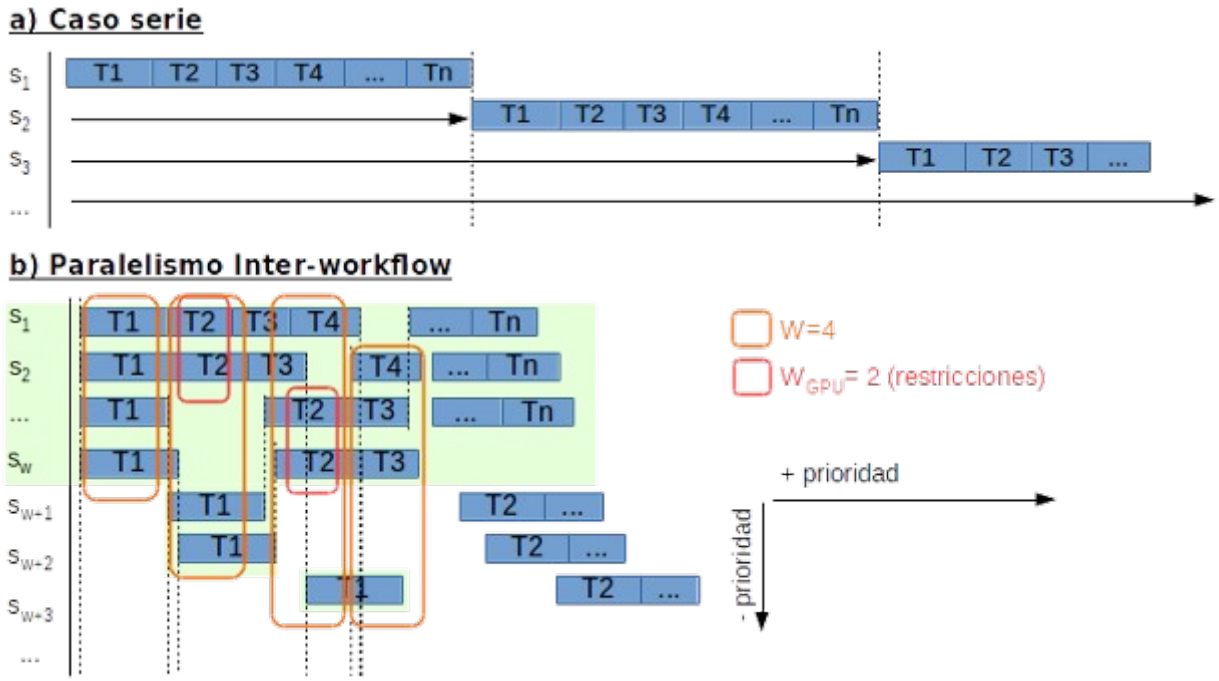


Figura 4.1: esquemas de planificación de ejecución de etapas del *workflow*: (a) ejecución serie y (b) ejecución con paralelismo *inter-workflow*.

La figura 4.1b muestra el escenario de ejecución con paralelismo *inter-workflow* que se definió en el capítulo 3. En este escenario se mantiene una ventana de al menos w *workflows* que son procesados en paralelo. Es necesario aplicar restricciones de acceso a los recursos (CPUs y GPUs), tanto para asegurar una ejecución correcta (evitar que se agote la memoria disponible en la GPU) como para evitar situaciones de sobrecarga de la memoria RAM que puedan dar lugar a ejecuciones muy ineficientes. Se aplica una política que prioriza la ejecución de las etapas del *workflow* más antiguo; esto es, si i<j se prioriza el avance de la ejecución de la etapa T_k del

workflow s_i frente a la etapa T_r del *workflow* s_j, independientemente de si k es mayor o menor que r. Solamente se incorpora un nuevo *workflow* a la ventana cuando finaliza completamente uno anterior, o si uno de los dos tipos de recursos de cómputo, CPU o GPU, está libre y ninguna de las etapas de los *workflows* actuales está disponible para ser ejecutada en ese tipo de recurso.

El esquema de planificación anterior logra un rendimiento importante comparado con la ejecución serie, pero no consigue utilizar los recursos del sistema de la forma más eficiente. Hay varias causas que lo impiden. En muchas fases de la ejecución no se consigue solapar el uso de las GPUs y de las CPUs. En otras fases, o bien no se ocupan completamente todas las CPUs, o bien la carga de tareas en la GPU es insuficiente para lograr todo el potencial de rendimiento. No se analiza la afinidad entre las tareas correspondientes a diferentes etapas: ciertas combinaciones de etapas ejecutadas simultáneamente pueden ofrecer un rendimiento favorable, mientras que otras combinaciones pueden tener un comportamiento conjunto perjudicial. Tampoco hay ningún control sobre la asignación de tareas a dominios de memoria NUMA (que se explicarán en la sección siguiente) ni a núcleos de ejecución concretos, que favorezca la reutilización de datos dentro de la jerarquía de memoria. Finalmente, el esquema de planificación no permite valorar si el rendimiento que obtenemos es aceptable o está lejos del rendimiento adecuado: solamente podemos comparar el tiempo serie con el tiempo en paralelo y desear que la mejora sea cercana al número de recursos que estamos utilizando de forma concurrente.

La política representada por la figura 4.1b no ofrece suficiente flexibilidad para escoger las tareas a ejecutar en cada momento, y tiene la tendencia a ejecutar siempre etapas del mismo tipo o etapas consecutivas dentro del *workflow*: por ejemplo, la etapa T_k para un cierto sujeto suele ejecutarse con las etapas T_{k-1}, T_k o T_{k+1} del resto de sujetos dentro de la misma ventana. Por otra parte, el parámetro w representa tanto el grado de paralelismo para ejecuciones en CPU dentro del sistema físico (cuántas tareas pueden estar ejecutándose en CPU de forma simultánea), como el grado de paralelismo *inter-workflow* (la ventana de sujetos procesados de forma independiente). Como dentro del mismo *workflow* las etapas suelen ser dependientes (en el caso del *workflow* de FreeSurfer considerado en el capítulo anterior el paralelismo promedio era inferior a 1,5), el resultado es que el paralelismo utilizado por el esquema de planificación anterior no es suficiente para ocupar completamente en todo momento los recursos de cómputo del sistema. Por último, no se hace ningún tipo de actuación cuando la carga de cómputo entre las CPUs y las GPUs está desbalanceada, de modo que no se asegura que el tipo recurso que resulta crítico para la aplicación (el *bottleneck* o cuello de botella) sea el que salga favorecido en la selección de etapas a ejecutar.

La nueva propuesta de planificación de la ejecución de *workflows* que presentamos en este

capítulo supone aumentar mucho más el paralelismo potencial de tareas, y disponer de mayor flexibilidad para escoger las etapas adecuadas para ejecutar en cada momento. Tal como se ilustra en la figura 4.2, se divide el conjunto de sujetos de entrada en grupos de w elementos, y la ejecución de estos grupos se realiza usando un esquema de tipo *pipeline* (o segmentado). Suponiendo que el *workflow* se compone de n etapas, la fase transitoria inicial utiliza n-1 grupos de w sujetos para ejecutar un cierto número de etapas de cada grupo (siempre teniendo en cuenta las limitaciones del sistema). Como esta fase de la ejecución es corta comparada con el total de la ejecución, nos centraremos en la fase que denominamos estacionaria.

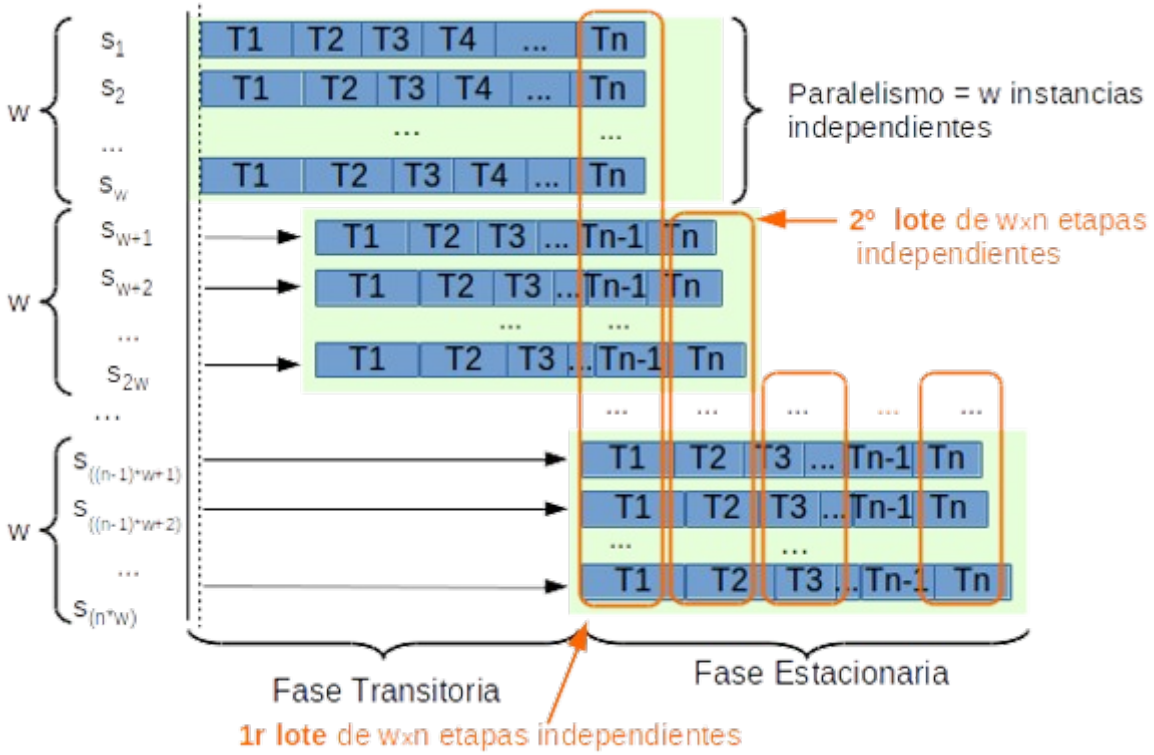


Figura 4.2: nuevo esquema de ejecución, indicando tanto la fase transitoria como la fase estacionaria. El primer lote de ejecución de la fase estacionaria está remarcado y supone un conjunto de ***w×n*** etapas independientes.

En la fase estacionaria se ejecutan lotes de ***w×n*** etapas completamente independientes entre sí, y con una mezcla homogénea de w etapas de cada tipo (en la figura 4.2 aparece remarcado el primer lote de ejecución de la fase estacionaria). Los lotes se tratan como una unidad dentro de la política de planificación de la ejecución de tareas.

En la figura 4.3 se muestra cómo la ejecución de todas las tareas del lote no tiene que ser ni simultánea, ni en el orden de las etapas (porque todas las tareas pertenecen a sujetos diferentes y son independientes). El procesamiento de cada lote de tareas está desplazado una etapa, tal y como ocurre típicamente en los esquemas de ejecución en forma de *pipeline*. La figura 4.3 ilustra cómo para comenzar la ejecución del lote se necesita introducir los datos iniciales de un nuevo

grupo de w sujetos de entrada, y se utilizan los datos intermedios (temporales) de los sujetos que todavía no se han procesado completamente. La ejecución de cada lote supone finalizar el procesamiento correspondiente a w sujetos (finalizar la última etapa correspondiente al grupo de sujetos que hace más tiempo que se comenzó a procesar) y generar los datos intermedios que se necesitan para continuar el procesamiento de los sujetos de los n-1 grupos restantes.

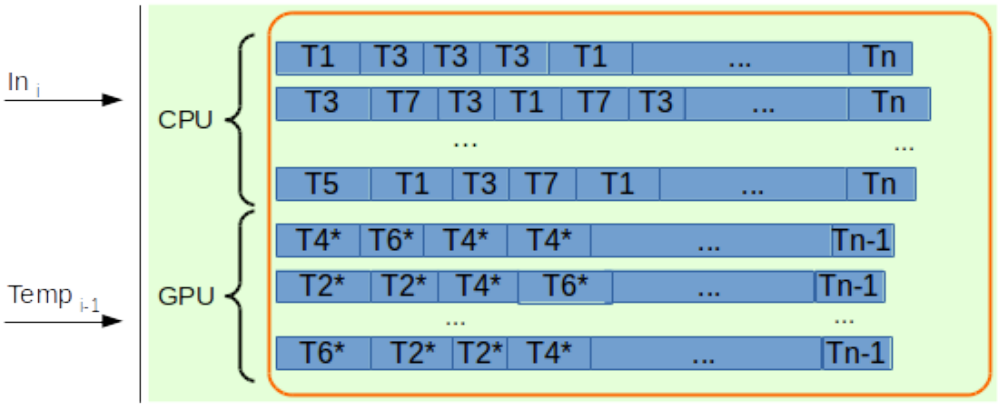


Figura 4.3: posible orden de ejecución en un sistema híbrido (CPU+GPU) de las tareas correspondientes a un lote. Las ***w×n*** etapas independientes que constituyen el lote se pueden ejecutar en cualquier orden: el objetivo de este capítulo es encontrar un orden adecuado que consiga un buen rendimiento.

Observar que el valor w no está directamente determinado por el número de recursos del sistema (número de CPUs o GPUs) o por el máximo número de tareas que se puedan ejecutar simultáneamente, y se puede escoger para asegurar un exceso de paralelismo que sea suficiente para ocupar razonablemente bien los recursos. Por otro lado, se dispone de total flexibilidad para escoger las combinaciones de tareas a ejecutar, permitiendo seleccionar aquellas combinaciones que hagan un uso más eficiente de las CPUs y de las GPUs. Finalmente, y no menos importante, es posible analizar la ejecución de este conjunto fijo de tareas para medir la carga de trabajo tanto en CPU como en GPU y encontrar un posible desbalanceo que nos señale el recurso crítico o, lo que es lo mismo, el principal factor limitante del rendimiento.

El recurso crítico es aquel que necesita más tiempo para realizar las tareas correspondientes a un lote. Si, por ejemplo, las tareas de GPU tardan más tiempo en ejecutarse que las tareas de CPU, entonces el sistema funcionará al ritmo de la GPU, y no serviría de nada aumentar los recursos de cómputo en CPU. En el caso de que el recurso crítico sea la GPU, proponemos mover parte de la carga de trabajo a la CPU y/o priorizar el uso de las CPUs para tareas que usan la GPU, evitando que otras tareas no críticas puedan ralentizar las tareas GPU. Asumimos que, como ocurre en el caso de la aplicación FreeSurfer, para todas las tareas que tienen una implementación en GPU también se dispone de una implementación usando únicamente la CPU, que obviamente tienen una mayor latencia de ejecución.

Proponemos realizar un análisis empírico inicial de la ejecución de las tareas del *workflow* (de FreeSurfer, en nuestro caso concreto) que proporcione datos que permitan realizar un buen plan de ejecución estático de cada lote de etapas o tareas. El plan de ejecución se define mediante un conjunto de listas de tareas, de forma que la clasificación de las tareas del lote en listas diferentes y el orden de las tareas en cada lista favorecen el uso equilibrado de los recursos y la ocurrencia de las combinaciones de tareas más eficientes. El tiempo y esfuerzo invertido en este análisis previo es aceptable en sistemas donde una gran parte del tiempo se dedica a ejecutar un único tipo de *workflow*, como es el caso del escenario que hemos planteado. Alternativamente, sería posible obtener estos datos de caracterización de forma dinámica, mientras se van ejecutando las etapas de los *workflows*. Esta idea se propone posteriormente como trabajo futuro de esta tesis.

El plan de ejecución estático diseñado para cada lote de etapas:

1. Priorizará la ejecución eficiente del recurso, CPU o GPU, que se considere crítico, y decidirá cuánta carga hay que trasladar de GPU a CPU, si es necesario
2. Promoverá la ejecución combinada de etapas que proporcionen una mejor eficiencia en el uso de los recursos, beneficiando sobre todo a la eficiencia obtenida sobre el recurso crítico (CPU o GPU)
3. Favorecerá que no se den combinaciones que sobrepasen las restricciones debidas a límites físicos del sistema de cómputo (capacidad de memoria de GPU …). Un mecanismo dinámico adicional garantizará que las combinaciones no seguras no puedan llegar a ejecutarse, frenando la ejecución si es necesario.
4. Decidirá qué tareas son adecuadas para que un mecanismo dinámico equilibre la carga entre CPUs y/o entre GPUs durante las transiciones entre lotes consecutivos de ejecución (explicado a continuación)

En la figura 4.3 se puede observar que las últimas tareas ejecutadas del lote no acaban a la vez, y se puede inferir que en un caso real las diferencias pueden ser importantes. Para limitar estas diferencias (este desequilibrio en el tiempo de uso de los recursos de cómputo), se define un mecanismo dinámico que adapte la carga de trabajo durante las transiciones entre lotes de ejecución. Las últimas tareas de cada lista son las candidatas para corregir el plan de ejecución estático, y ejecutarse en CPUs diferentes a las previstas: hay que tratar que sean tareas con latencias de ejecución pequeña y que se combinen de forma favorable (o poco desfavorable) con la ejecución de otras tareas.

El objetivo conjunto del plan de ejecución estático y de la estrategia de balanceo dinámico de carga durante las transiciones entre lotes es que el solapamiento entre lotes de ejecución sea pequeño, y que la ejecución combinada de tareas que se realiza dinámicamente corresponda en todo lo posible a la prevista por el plan de ejecución, supuestamente favorable. Como no se puede asegurar un límite máximo al solapamiento, debido a las condiciones variables de la ejecución, es

necesario asegurar que se cumplen las dependencias de datos entre etapas de diferentes lotes.

En la figura 4.4.a se muestra el esquema de ejecución de lotes en forma de *pipeline* secuencial, donde las tareas del lote i+1 necesitan los datos producidos por el lote i. Para poder solapar la ejecución de tareas de diferentes lotes es necesario controlar qué tareas del primer lote han finalizado para poder decidir qué tareas del siguiente lote pueden comenzar. Esto complica el diseño y la gestión dinámica del plan de ejecución. Para no imponer una restricción demasiado estricta, que penalice el rendimiento, se propone un esquema entrelazado entre lotes de ejecución pares y lotes impares (ver figura 4.4.b) que permite un solapamiento de hasta un lote completo. El entrelazado aumenta la latencia de procesado de los sujetos (desde que entran a ejecutarse hasta que se obtienen sus resultados) pero facilita la gestión de las dependencias entre tareas. El control que debe realizar el planificador durante la ejecución se limita a impedir que tareas del mismo “color” se puedan solapar; es decir, la ejecución del lote i debe acabar completamente antes de poder comenzar a ejecutar ninguna tarea del lote i+2.

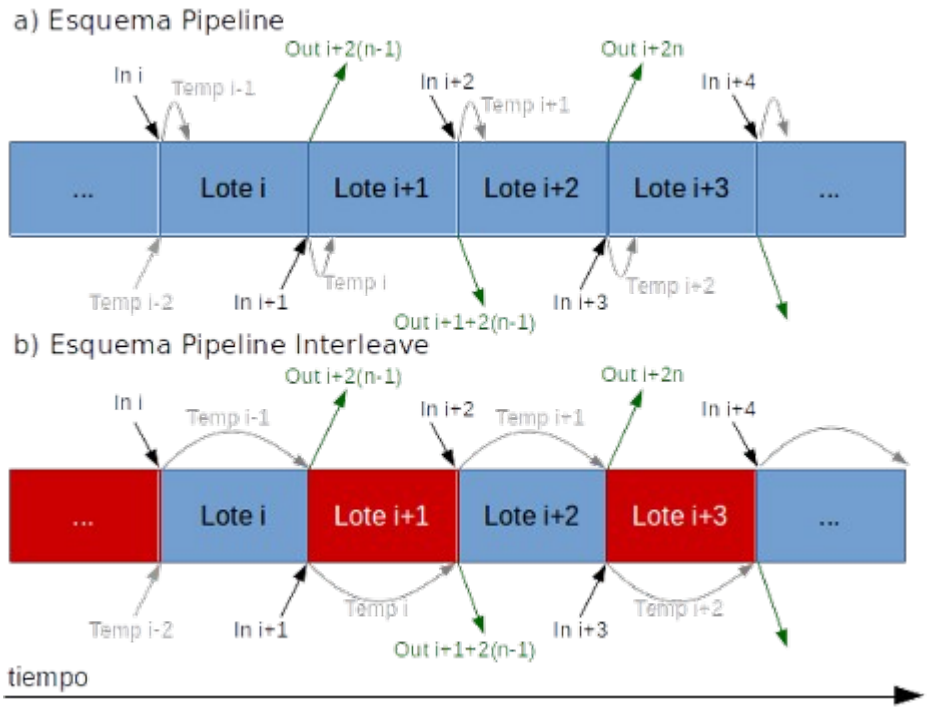


Figura 4.4: esquema *pipeline* con ejecución de los lotes (a) secuencialmente y (b) entrelazados (interleave). Los datos de entrada (sujetos) se etiquetan como In y los de salida como Out. Los datos intermedios generados por cada lote, se etiquetan como Temp. Los lotes pares e impares se distinguen con diferente color.

Para poder controlar las combinaciones de tareas que se ejecutan dentro de un mismo núcleo, el planificador invoca la ejecución de una tarea forzando a que el Sistema Operativo enlace la tarea con una CPU específica. Así se evita también que el Sistema Operativo migre la tarea, lo que podría provocar ineficiencias en el acceso a los datos dentro de la jerarquía de memoria

En las siguientes secciones explicaremos cómo obtener los datos empíricos sobre las etapas del *workflow* de FreeSurfer, cómo calcular la carga de trabajo en CPU y GPU y decidir si hay que mover carga entre ellos, cómo diseñar un plan de ejecución apropiado y cómo funciona el algoritmo de planificación dinámica con listas ordenadas de tareas.

4.2 Metodología experimental

Los experimentos realizados en el presente capítulo se han llevado a cabo en un nodo de cómputo con las siguientes características:

- Dos procesadores Intel Xeon E5645 funcionando a una frecuencia de reloj de 2,40 GHz y con 6 núcleos de cómputo y 12 MB de memoria caché de último nivel cada uno de los procesadores, y un total de 96 GB de memoria RAM. Cada núcleo de cómputo permite ejecutar dos *threads* H/W de forma simultánea (*hyperthreading*), lo que supone disponer de un total de 6×2×2 = 24 CPUs virtuales.

- Dos tarjetas NVIDIA GeForce GTX 750 Ti con 640 CUDA cores y memoria GDDR de 2048 MB cada una.

- Dos discos duros de 1,5 TB configurados de manera independiente y un sistema de almacenamiento configurado con NFS, distribuido mediante LVM en diferentes particiones, y con un espacio total de 7,3 TB. La configuración en el servidor es de 6 discos de 2 TB cada uno en RAID-6. En concreto, usamos una partición de 500 GB para almacenamiento de la aplicación Freesurfer, los códigos de los experimentos, y los datos (resonancias) de entrada y salida.

Los 96 GB de memoria principal se dividen en dos dominios NUMA (Non-uniform memory access), de forma que cada procesador está conectado directamente a un dominio de 48 GB de memoria principal, y tiene que acceder a través del otro procesador para acceder a los 48 GB del otro dominio NUMA. En la figura 4.5 se muestra la distribución de CPUs virtuales en los dos dominios NUMA correspondientes a los dos procesadores (o *sockets*).

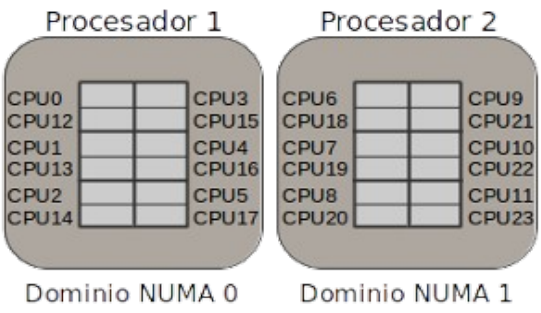


Figura 4.5: distribución de las CPUs virtuales en cada dominio NUMA

Se ha utilizado el siguiente *software*:

- Sistema Operativo Scientific Linux release 6.6 (*kernel* 2.6.32-431.20.3.el6.x86_64)
- *Driver*s NVIDIA 4.0 para Linux con soporte CUDA.
- CUDA Toolkit 5.0 y el CUDA *Software* Development Kit para Linux.
- FreeSurfer 5.3, compilado para CUDA 5.0.

En el estudio de caracterización de las etapas del *workflow* que tomamos como caso de uso, y que se presentará en secciones posteriores, se ha contado con varias herramientas de medición a modo de instrumentación de la ejecución. En primer lugar, tenemos las medidas que pueda emitir la propia aplicación (qué recurso utiliza, tiempo de ejecución). En segundo lugar, para las etapas que se ejecutan en GPU, contamos con la aplicación NVIDIA Profiler (NVPROF) que nos ofrece información de bajo nivel sobre la ejecución de las funciones en la GPU (*kernels*), así como del uso de la API (Application Programming Interface) de CUDA. También disponemos del comando NVIDIA-SMI (System Management Interface) [35] que nos provee de información el uso de ciertos recursos de cada una de las tarjetas NVIDIA disponibles en el sistema, como el tamaño de memoria usado, el porcentaje de uso de la capacidad de cómputo, la temperatura, etc.

Los experimentos realizados a lo largo de este capítulo involucran datos correspondientes a 200 sujetos diferentes. Los datos de entrada de cada sujeto se encuentran en un directorio montado en un sistema de ficheros en red (NFS), y el resultado final del procesado se almacena en este mismo sistema compartido. Asumimos por tanto un sistema fiel a la realidad, en el que los datos de entrada van llegando desde el exterior, y los resultados se van leyendo hacia el exterior. Aunque este sistema de ficheros compartido tienen un rendimiento relativamente malo comparado a un sistema de ficheros paralelo o al uso de discos locales, debido a que el tamaño de los datos no es muy grande, el tiempo total de lectura y escritura es muy pequeño comparado con el tiempo de ejecución. Lo que sí es muy importante es gestionar de manera eficaz los datos intermedios producidos por las etapas del *workflow*, guardados en el sistema de ficheros, y los ficheros que contienen las trazas de cada una de las ejecuciones. Para ello sí que se utiliza el directorio de trabajo asignado al espacio del disco local, con un mecanismo de caché de disco que evita mucho porcentaje de tráfico con el disco.

Todas las ejecuciones de los experimentos que se realizan a lo largo de este capítulo para caracterizar la aplicación cuentan con entre 5 y 7 repeticiones (menos repeticiones cuando el tiempo total de la ejecución es muy alto). Estas repeticiones nos permiten calcular la desviación estándar de los tiempos de ejecución, y también promediar el resultado de combinar la ejecución de tareas en diferentes fases de su ejecución. Por ejemplo, el comportamiento de la fase inicial y

de la fase final de la ejecución de una tarea puede ser distinto, y al combinar la ejecución con otras tareas el rendimiento puede depender del desfase temporal entre el inicio de la ejecución de estas tareas.

En muchos de los experimentos de caracterización del rendimiento de la aplicación, para simplificar el experimento, los resultados intermedios de cada etapa han sido computados con antelación y se han almacenado y duplicado en el directorio de trabajo. Las escrituras de los resultados intermedios producidos por las etapas se mantienen, pero no modifican las copias pre-calculadas. De este modo no es necesario controlar las dependencias entre las tareas, y es muy útil cuando se ejecutan simultáneamente muchas instancias del *workflow* para el mismo sujeto de entrada. Para los experimentos finales, que suponen la ejecución planificada de 200 sujetos, sí que se usa una planificación realista, en la que se aseguran las dependencias entre las tareas, y que las tareas generan y usan los datos temporales adecuados.

A continuación se presenta el estudio empírico de las características de la ejecución de las etapas del *workflow* y del efecto de la ejecución combinada entre etapas.

4.3 Caracterización Inicial del *workflow*

En este apartado presentamos un estudio empírico realizado en el sistema de cómputo antes descrito y tomando como caso de uso una parte del *workflow* completo de FreeSurfer que realiza la reconstrucción volumétrica del cerebro a partir de una secuencia de cortes de resonancias magnéticas. El objetivo de este estudio es obtener las principales características de ejecución de las etapas de este *workflow*.

El *workflow* de FreeSurfer utilizado en el capítulo 3 consta de 31 procesos en total, que decidimos agrupar en 13 bloques o etapas. Para reducir el tiempo total de la experimentación hemos analizado únicamente las primeras 5 etapas del estudio anterior, que representan el sub-*workflow* real de reconstrucción volumétrica, y que suponen un total de 15 procesos de los 31 originales y aproximadamente el 50% del tiempo de ejecución del *workflow* analizado en el capítulo 3. De las cinco etapas del *workflow*, tres se ejecutan sólo en CPU (T1, T3, T5) y las otras dos se pueden ejecutar en una versión que solamente usa CPU (T2, T4) o en una versión que utiliza adicionalmente la GPU como acelerador (T2*, T4*). Este conjunto de 5 etapas es representativo de la mezcla de etapas que tienen los múltiples *workflows* de FreeSurfer.

A continuación se presenta el estudio de caracterización de las etapas ejecutadas completamente en CPU y posteriormente de las etapas que también utilizan la GPU.

4.3.1 Caracterización de las etapas ejecutadas completamente en CPU

El objetivo de este análisis es el de comprender cómo y en qué medida las etapas del *workflow* utilizan las CPUs y la memoria del sistema. Primero analizaremos la ejecución serie, usando una única CPU, luego la ejecución con *hyperthreading*, usando las dos CPUs lógicas asociados a un núcleo de ejecución físico, y finalmente el uso de todos los núcleos de cómputo de los dos procesadores.

En el primer experimento se realizan ejecuciones puramente secuenciales, usando una única CPU virtual y un único núcleo de ejecución del sistema (en concreto, la CPU0). Mostramos los resultados para tres sujetos de entrada diferentes, escogidos como representativos entre todos los que hemos analizado. Con el objetivo de medir las posibles fuentes de variabilidad en el rendimiento, para cada sujeto se realizan múltiples ejecuciones. Se mide la latencia o tiempo de ejecución de cada etapa y el consumo de memoria principal del sistema. La variabilidad se expresa usando la desviación estándar de los tiempos de todas las ejecuciones realizadas. La Tabla 4.1 muestra los resultados de las ejecuciones, con el tiempo de ejecución, medido en segundos, el porcentaje de la desviación estándar, y el consumo máximo de memoria medido en GigaBytes.

Etapa del *Workflow*	**Tiempo Sujeto1 [Seg]**	**Tiempo Sujeto2 [Seg]**	**Tiempo Sujeto3 [Seg]**	**Desviación estándar entre sujetos**	**Máx. Memoria [GB]**
T1	3302 ± 1,24%	3522 ± 0,02%	3250 ± 0,38%	4,30%	2,37
T2	10528 ± 0,39%	13422 ± 0,10%	12022 ± 0,12%	12,07%	2,69
T3	143 ± 0%	110 ± 0,35%	134 ± 0,62%	13,22%	2,48
T4	1732 ± 1,25%	1682 ± 0,58%	1709 ± 1,09%	1,47%	2,07
T5	1401 ± 0,42%	1406 ± 0,15%	1368 ± 0,21%	1,48%	3,34

Tabla 4: tiempo de ejecución medio (segundos), desviación estándar (%) y consumo de memoria máximo (GigaBytes) de cada etapa del *workflow* para tres sujetos de entrada diferentes.

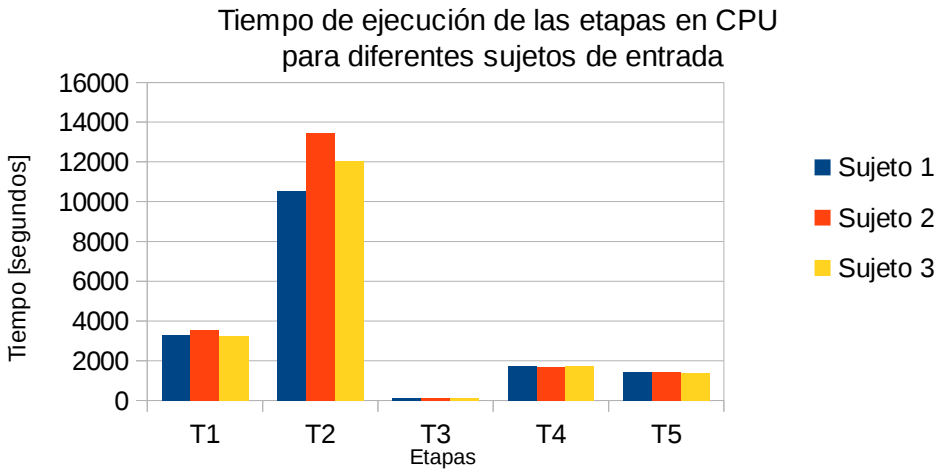


Figura 4.6: tiempo de ejecución medio (segundos) de cada etapa para diferentes sujetos, correspondientes alos datos presentados en la tabla 4.1

La Tabla 4.1 (y la Figura 4.6) muestra que hay una variabilidad importante en los tiempos de ejecución entre diferentes sujetos de entrada. Las etapas T2 y T3 muestran desviaciones alrededor del 12%, y la etapa T1 de 4,3%. Esta variabilidad es debida al contenido particular de las imágenes de entrada, ya que sujetos diferentes tienen una morfología diferente y, por lo tanto, el procesamiento de la imagen está sujeto a variaciones. La implicación de estos resultados es que la planificación que hagamos basándonos en datos promedios va a tener una variabilidad que puede ser importante, y que debe ser considerada en la práctica.

Por otro lado, también podemos observar una cierta variabilidad no despreciable en las diferentes ejecuciones de cada etapa realizadas con los mismos datos de entrada, aunque en ninguno de los casos supera un porcentaje mayor al 1,3%. Esta variabilidad se puede justificar teniendo en cuenta dos aspectos:

1) Las etapas T1, T3 y T5 se componen de diferentes subprocesos de tiempos de ejecución más pequeños que se agrupan en único bloque de ejecución. En el caso de T2 y T4, son etapas que ejecutan un único proceso. Las lecturas y escrituras de datos intermedios realizadas por los subprocesos suponen una fuente de variabilidad en la latencia de ejecución.

2) Algunos algoritmos de FreeSurfer implementan métodos de cálculo numéricos que son iterativos y convergentes, en función de valores «semilla» aleatorios y de un margen de error para la convergencia. Hemos observado que diferentes ejecuciones con los mismos datos de entrada pueden suponer la ejecución de un número de iteraciones en algunos de los algoritmos que es diferente cada vez.

Finalmente, la Tabla 4.1 incluye el consumo máximo de memoria del nodo (medido en GigaBytes) para cada una de las etapas, entre todas las ejecuciones de cada etapa para todos los sujetos. El consumo es muy similar en todas las ejecuciones y para todos los sujetos, así que no se muestra la desviación estándar. En nuestro sistema de cómputo, con 96 GB de memoria principal, se podrían ejecutar hasta 28 instancias de la etapa de mayor consumo de memoria, que es T5 (96 / 3,34 = 28,74). Como en la práctica no tiene sentido ejecutar más de 24 tareas a la vez, no consideraremos ninguna restricción en el número total de tareas a ejecutar en la CPU.

A continuación medimos el rendimiento de las etapas que se ejecutan en CPU cuando escalamos el número de instancias que se ejecutan simultáneamente en el nodo de cómputo. Este análisis nos proporciona información de cómo se aprovecha la capacidad del sistema para ejecutar dos *threads* simultáneamente en el mismo núcleo de cómputo, y cómo se aprovecha la capacidad multi-núcleo y multiprocesador. También nos indicará el número adecuado de instancias a ejecutar simultáneamente dentro de nuestra propuesta de planificación de la fase estacionaria que hemos introducido anteriormente.

Los datos de este análisis se presentan en la tabla 4.2, y representan la mejora (*speedup*) del rendimiento cuando se aumenta el número de tareas ejecutadas en paralelo. En todos los casos, para medir la escalabilidad del rendimiento se usan instancias de la misma tarea y correspondientes al mismo sujeto, y así reducir el efecto de la variabilidad entre sujetos y entre tareas (esto último se analizará más adelante). Las medidas se realizan por separado con los tres sujetos escogidos en el estudio y se muestra el promedio de las mejoras obtenidas. No se muestra la desviación estándar de este promedio porque es muy pequeña (inferior al 0,5%).

Etapa del *Workflow*	***Speedup* 2 instancias**	***Speedup* 12 instancias**	***Speedup* 24 instancias**
T1	1,21	6,23 (5,16)	12,31 (1,98)
T2	1,18	4,34 (3,68)	8,09 (1,86)
T3	1,23	6,50 (5,26)	12,26 (1,89)
T4	1,21	6,28 (5,20)	12,36 (1,97)
T5	1,22	6,02 (4,94)	12,03 (2,00)

Tabla 5: *speedup* promedio (correspondiente a tres sujetos) para la ejecución de cada una de las etapas del workload, cuando se ejecutan 2, 12 y 24 instancias simultáneas, respecto a la ejecución de una única instancia. Entre paréntesis se presenta el *speedup* relativo a la columna anterior (de la izquierda), que permite ver la ganancia relativa. Para 2 instancias se usan las dos CPUs virtuales de un mismo núcleo de ejecución. Para 12 instancias se usan los 6 núcleos de cómputo de un procesador (cada núcleo con 2 instancias usando la capacidad de *hyperthreading*). Con 24 instancias se usan los dos procesadores.

El uso de dos instancias en el mismo núcleo de ejecución permite evaluar la mejora de rendimiento que aporta la capacidad del denominado *hyperthreading*. Se han medido mejoras de entre un 18% y un 23%. Estos valores son positivos e indican que las ejecuciones secuenciales de un único *thread* están limitadas por dependencias de datos y latencias; estas latencias se ocultan parcialmente gracias a la ejecución multi-*thread*. Aunque el máximo rendimiento esperado es de una mejora del 100%, en la práctica es difícil encontrar mejoras superiores al 30%.

En el caso de ejecutar 12 etapas simultáneas se usan los seis núcleos de cómputo de un único procesador y se configura el sistema para que las tareas fijen el uso del Dominio NUMA número 0 (es decir, utilizan la mitad de la memoria cercana al procesador). La escalabilidad ideal sería una mejora de 6 sobre la ejecución de 2 instancias (se usan ahora 6 núcleos en lugar de uno sólo). En la práctica, el *speedup* es superior a 5,1 para las etapas T1, T3 y T4, es alrededor de 5 para la etapa T5, y es inferior a 3,7 para la etapa T2. La falta de escalabilidad indica que algún recurso compartido, como el acceso a la memoria principal o al disco, está limitando parcialmente el rendimiento. Como veremos posteriormente, una adecuada combinación de tareas diferentes puede repartir mejor el uso de los recursos compartidos y lograr un rendimiento mayor.

En el caso de 24 etapas simultáneas utilizamos ambos procesadores presentes en el sistema: seis núcleos por procesador y la capacidad de *hyperthreading* en cada núcleo. La configuración del sistema liga a las tareas ejecutadas con el dominio NUMA más cercano: Dominio 0 para el multiprocesador 1 y Dominio 1 para el multiprocesador 2. La ganancia de usar un segundo procesador roza el 100% (el máximo esperable) en tres de las etapas (T1, T4 y T5), y es del 89% y 86% para las tareas T3 y T2. Hay alguna interferencia, sobre todo para estas dos últimas etapas, entre los procesadores, debida a que siguen existiendo algunos accesos cruzados entre los dominios NUMA, y a la gestión compartida de las llamadas al sistema operativo, entre ellas las que suponen el acceso al disco local compartido. La figura 4.7 presenta los mismos resultados de *speedup* que en la tabla 4.2.

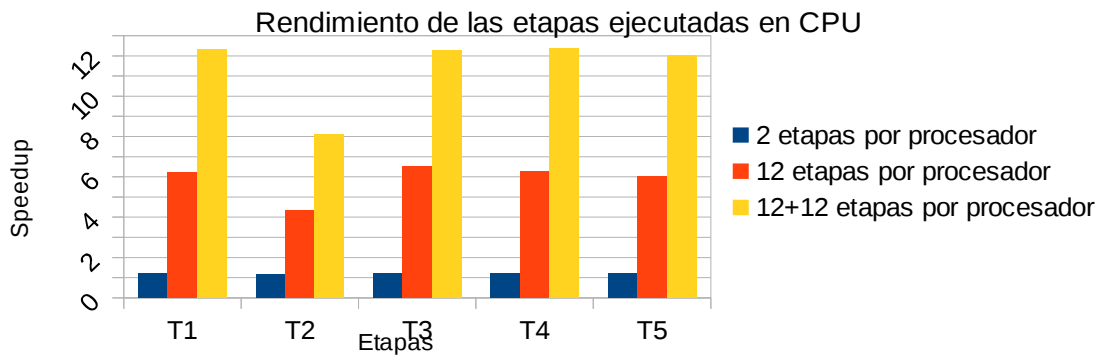


Figura 4.7: *speedup* por etapas para ejecuciones de 2, 12 y 24 instancias simultáneas, correspondiente a los datos presentados en la Tabla 4.2

Finalmente, para acabar el estudio de caracterización de las etapas ejecutadas en CPU, calculamos la productividad obtenida para cada una de las etapas en cada uno de los experimentos con 1, 2 y 12 instancias simultáneas en un solo procesador. La tabla 4.3 muestra los datos del tiempo total del experimento en segundos, y la productividad obtenida medida en sujetos cada 1000 segundos. El valor de la productividad nos será útil para los análisis que se realizarán con posterioridad.

Etapa del *workflow*	**Número de Instancias simultáneas**	**Tiempo Experimento [segundos]**	**Productividad [sujetos/1000 segundos]**
T1	1 (1 núcleo)	3302	0,30
	2 (1 núcleo)	5509	0,36 (1,21x)
	12 (6 núcleos)	6373	1,88 (6,23x)
T2	1 (1 núcleo)	10528	0.09
	2 (1 núcleo)	17844	0.11 (1,18x)
	12 (6 núcleos)	29120	0.41 (4,34x)
T3	1 (1 núcleo)	143	6,99
	2 (1 núcleo)	231	8,66 (1,23x)
	12 (6 núcleos)	264	45,45 (6,50x)
T4	1 (1 núcleo)	1732	0,58
	2 (1 núcleo)	2863	0,70 (1,21x)
	12 (6 núcleos)	3311	3,62 (6,28x)
T5	1 (1 núcleo)	1401	0,71
	2 (1 núcleo)	2297	0,87 (1,22x)
	12 (6 núcleos)	2795	4,29 (6,02x)

Tabla 6: tiempo total del experimento en segundos y productividad obtenida en las ejecuciones con 1, 2 y 12 instancias simultáneas (usando un núcleo, un núcleo con dos *threads*, y un procesador de 6 núcleos, respectivamente). En la última columna, el valor entre paréntesis indica la mejora relativa a la ejecución con una instancia.

4.3.2 Caracterización de las etapas ejecutadas en GPU

En este subapartado analizaremos las etapas del *workflow* de estudio que se ejecutan en CPU y que utilizan la GPU como acelerador, que concretamente son dos, T2* y T4* (usamos el asterisco para indicar que nos referimos a las implementaciones que usan GPU, y no a las que usan únicamente la CPU, más lentas). En este caso particular, ambas etapas constan de un único proceso monolítico que durante su ejecución invoca la ejecución de código en GPU en múltiples ocasiones.

Antes de detallar la caracterización de las etapas, conviene recordar cómo es el flujo de ejecución usando una GPU. Cada tarea que usa la GPU comienza como un proceso que se ejecuta en CPU, y que realiza los siguientes pasos:

1. Inicializa la GPU
2. Reserva memoria para los datos tanto en el Host (CPU) como en el Device (tarjeta GPU)
3. Copia los datos a procesar desde la memoria del Host a la memoria del Device.
4. Lanza a ejecutar de forma secuencial o simultánea uno o múltiples *kernels* (funciones de código GPU, que ejecutan un número masivo de *threads* de forma independiente).
5. Copia los datos desde el Device a la memoria del Host.
6. Se repiten los pasos 3-5 tantas veces como sea necesario.
7. Se libera la memoria reservada en la GPU y se finaliza la ejecución del proceso CPU.

La caracterización de las etapas en GPU también realiza varias ejecuciones con 3 sujetos de entrada diferentes, y se repiten las ejecuciones múltiples veces para estudiar la variabilidad entre ejecuciones. Se ha medido el número de *kernels* ejecutados por cada tarea, así como el tiempo acumulado que se emplea en el cómputo de dichos *kernels*. Por otra parte, se ha medido el tiempo de transferencia de datos entre el Host y el Device, expresado en segundos más la desviación estándar (en porcentaje) para indicar la variabilidad. Finalmente, se ha medido el máximo tamaño de memoria GPU empleada en las ejecuciones.

La tabla 4.4 presenta los datos para la etapa T2*. Un resultado significativo es la mayor variabilidad en el rendimiento de las ejecuciones, tanto para diferentes sujetos (7,3%) como para el mismo sujeto (mismos datos de entrada), donde oscila entre el 5 y el 16%. Además de las razones ya argumentadas en el caso de la ejecución únicamente en CPU, en este escenario de uso de la GPU aparecen nuevos factores que introducen mucho más “ruido” durante la ejecución, y que analizaremos en breve.

	Tiempo Total Ejecución [seg]	**Número de *Kernels* Ejecutados**	**Tiempo Acumulado *Kernels* [seg]**	**Tiempo Transferencia Host-Dev [seg]**	**Máximo Memoria GPU [MB]**
Sujeto1	1502,00 ± 11,3%	109208	332,99	160,6 ± 19,8%	442
Sujeto2	1622,83 ± 15,6%	112953	308,34	154,4 ± 25,4%	
Sujeto3	1403,37 ± 4,8%	118029	382,82	107,5 ± 8,44%	
Desv. Estándar	7,3%	3,9%	11,1%	20,6%	

Tabla 7: caracterización de la etapa T2*: tiempo total de ejecución, número y tiempo de cómputo de los *kernels*, tiempo de transferencia Host-Device y máximo consumo de memoria. Se usan tres sujetos diferentes de entrada y se realizan múltiples ejecuciones para cada sujeto

En los casos analizados se ejecutan 110.000 *kernels* en promedio y el tiempo acumulado promedio en la ejecución de estos *kernels* es de alrededor de 342 segundos, lo que supone dedicar al cómputo en GPU un tiempo cercano al 23% del tiempo total de la ejecución de la etapa. Por otro lado, el tiempo total acumulado de transferencia de datos entre el Host y el Device es de unos 142 segundos de media, aproximadamente un 10% del tiempo total. El tiempo de cómputo de los *kernels* más el tiempo de transferencia son 484 segundos, aproximadamente el 33% del tiempo total de ejecución de la etapa. Debemos estudiar qué está sucediendo en la ejecución de la etapa para el 67% del tiempo restante.

Para profundizar en el comportamiento de la ejecución de la etapa T2* hacemos un perfilado de la ejecución. La figura 4.8 muestra la traza de Gantt de los eventos que suceden en la GPU durante la ejecución.

La tarea T2* presenta tres fases diferenciadas en su ejecución: la primera (del segundo 0 al 320 de la ejecución) hace un uso importante de la GPU; la segunda (entre el segundo 320 y el 600) no utiliza la GPU (realiza todo su procesamiento en la CPU); y la tercera fase es la más extensa y vuelve a hacer un uso importante de la GPU. Los 280 segundos de la fase dos, sumados al tiempo acumulado de los múltiples huecos en el uso de la GPU en las fases 1 y 3, de una duración promedio del orden de los milisegundos, representan un total de 410 segundos de media en que la GPU está sin utilizar. Este tiempo supone un 27% del tiempo total, y no explica el 40% de tiempo restante en que las unidades de cómputo de la GPU están ociosas.

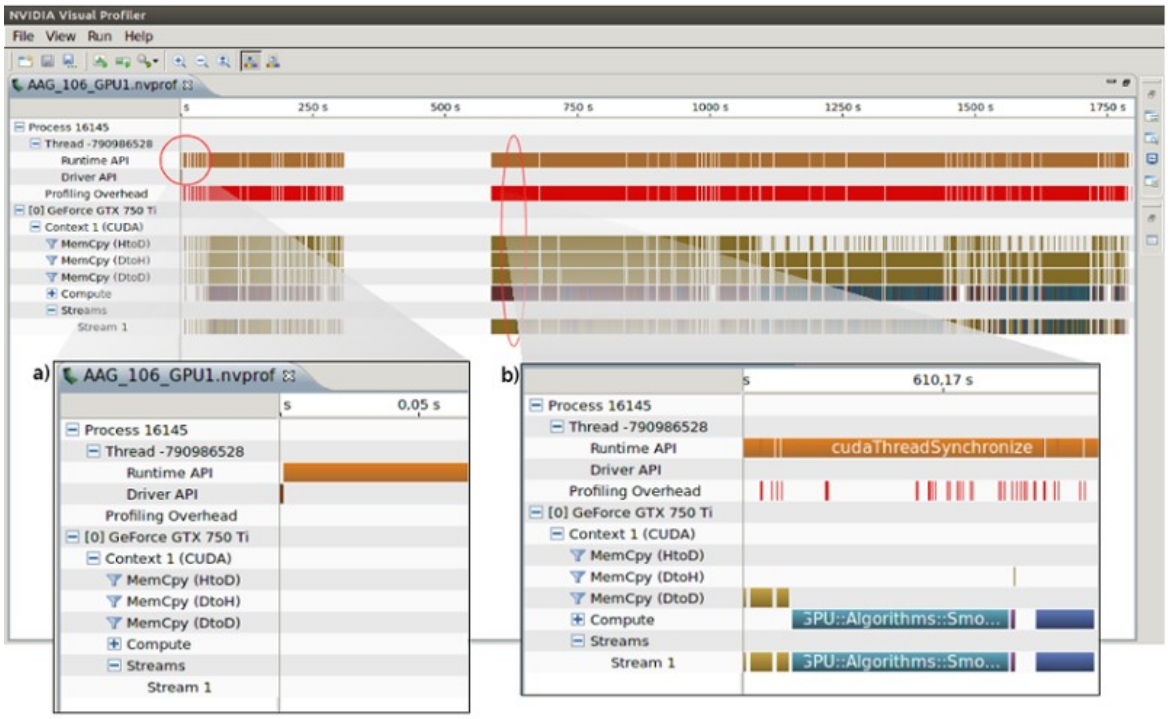


Figura 4.8: profiling de la etapa T2* mostrado mediante NVPROF: la ampliación a) (izquierda) presenta el tiempo inicial de inicialización del *Driver*. La ampliación b) (derecha) muestra el funcionamiento solapado entre transferencias (MemCpy) y la ejecución de un *kernel* con las llamadas a la API de CUDA.

La ampliación a) de la figura 4.8 muestra el tiempo de inicialización del *Driver* de la GPU, que es relativamente pequeño. En cambio, la ampliación b) muestra cómo aparecen huecos entre las operaciones de copia de datos entre Host y Device o entre las ejecuciones de los *kernels* (las filas etiquetadas como “MemCpy” y “Compute”), que son debidos a las latencias de gestión de las llamadas a la API de CUDA (la latencia total de estas llamadas a la API se ve en la fila etiquetada como “Runtime API”). Todas estas latencias adicionales para gestionar y sincronizar las operaciones son responsables del 40% del tiempo de ejecución. Si calculamos la relación entre el número de *kernels* ejecutados por la tarea T2* con el tiempo total acumulado obtenemos que la latencia de cada *kernel* es en promedio de unos 3,1 milisegundos. El análisis de los resultados que nos ofrece el perfilado nos indica que por cada llamada a un *kernel* se añaden 5,5 milisegundos adicionales debidos a las latencias en su gestión.

La implicación del análisis anterior es que la GPU no está siendo utilizada en su máximo potencial. Hay un margen de mejora de 4,35 veces (100% / 23%) si se consiguen utilizar los núcleos de cómputo de la GPU en su máxima capacidad. Para ello, hay que solapar el uso de estos núcleos con las transferencias entre memorias (que representan ahora alrededor del 10%) y sobre todo hay que solapar las llamadas a la API de CUDA, para permitir aprovechar ese 40% de tiempo que hemos calculado. A continuación mostraremos cómo se consigue escalar el rendimiento de manera efectiva cuando varias etapas T2* se ejecutan simultáneamente usando la misma GPU.

Antes de establecer los experimentos de escalado hay que notar que el pico máximo de consumo de memoria GPU por parte de las etapas T2* es de 442 MB. Este valor representa la reserva mínima de memoria que debemos garantizar en nuestro sistema para poder ejecutar una instancia de la etapa T2*, y por tanto nos limita a un máximo de 4 instancias simultáneas para no sobrepasar los 2 GB de memoria disponibles y evitar que las ejecuciones aborten de forma no controlada.

La tabla 4.5 muestra los resultados de rendimiento (tiempo total de ejecución y productividad medida en sujetos procesados cada 1000 segundos) obtenidos para las ejecuciones de 1, 2, 3 y 4 etapas T2* de forma simultánea en la misma GPU, y de 2 y 8 etapas T2* repartidas en 1 y 4 etapas por GPU, respectivamente (recordar que el sistema dispone de 2 tarjetas GPU idénticas). La figura 4.9 siguiente muestra en forma de gráfica las mejoras obtenidas al escalar la carga de trabajo en la GPU.

Etapa del *Workflow*	**Número de instancias simultáneas**	**Tiempo Experimento [segundos]**	**Productividad [sujetos / 1000 segundos]**
T2*	1 (GPU0)	1502	0,67
	2 (GPU0)	1663	1,20 (1,79x)
	3 (GPU0)	1778	1,69 (2,52x)
	4 (GPU0)	1959	2,04 (3,04x)
	1 (GPU0) + 1 (GPU1)	1505	1,33 (1,99x)
	4 (GPU0) + 4 (GPU1)	2402	3,33 (4,97x)

Tabla 8: tiempo total del experimento en segundos y productividad obtenida en las ejecuciones con 1, 2, 3 y 4 instancias simultáneas de la etapa T2* (usando una GPU) y de 2 y 8 instancias simultáneas (usando dos GPUs). En la última columna, el valor entre paréntesis indica la mejora relativa a la ejecución con una instancia

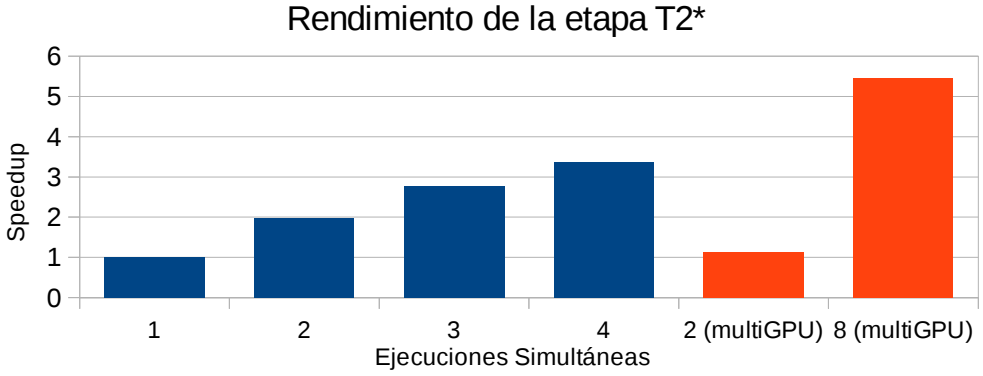


Figura 4.9: mejora obtenida en la ejecución de la etapa T2* al realizar múltiples ejecuciones simultáneas en una GPU (en los dos últimos caso usando dos GPUs).

El resultado más significativo es que ejecutando 4 tareas T2* de forma simultánea se logra mejorar el rendimiento de una sola GPU 3,04 veces. Usando los resultados del análisis del perfilado de la ejecución de una instancia, esto implica un uso de los recursos de cómputo de la GPU de un 3,04 × 23% = 69,92%. También implica que se logran solapar las llamadas a la API de CUDA de forma satisfactoria. Al tener una limitación de capacidad de memoria en la GPU no hemos podido comprobar si el uso de 5 o más instancias lograría mejorar la utilización de la GPU.

Hay que tener en cuenta que la fase 2 de la tarea T2*, tal como se veía en la Figura 4.8, no utiliza la GPU en absoluto; así, durante el intervalo de tiempo en que una instancia está en esta fase, no colabora en el aprovechamiento de los recursos de la GPU, y el paralelismo efectivo del que se nutre la GPU es inferior. Cabe recordar aquí que los experimentos realizan múltiples ejecuciones de las instancias de forma continua (en este caso hemos llegado a 15), para así ejecutar de forma simultánea tareas que están desfasadas temporalmente entre sí. En el caso de T2* hay una importante variabilidad en el rendimiento (de hasta un 10%) dependiendo del desfase entre las tareas. Este problema aparecerá más tarde, cuando busquemos un plan de ejecución para el workload. A todo esto, se suma la variabilidad intrínseca a la tarea T2* que vimos en la Tabla 4.4 que podía oscilar entre el 5 y el 15%, dependiendo del sujeto de entrada.

El otro resultado significativo es que el uso de una segunda GPU prácticamente dobla el rendimiento cuando se ejecuta una tarea por GPU, consiguiendo el máximo rendimiento esperado, pero cuando se usan 4 tareas por GPU la mejora es solamente del 63,2% (la productividad sube de 2,04 a 3,33), lejos de este 100% esperable. Esto indica la existencia de contención, bien en algún recurso físico compartido, o bien en la secuencialidad implícita establecida en las comunicaciones con el *software* que maneja las GPUs (el *driver*). Respecto a los recursos físicos, cada GPU está conectada a un bus PCI express que le permite comunicarse con la memoria principal del nodo de cómputo. Los controladores de memoria y la topología de las conexiones con los dominios NUMA podrían ser un factor importante para explicar esta contención, pero todas las pruebas que hemos hecho cambiando la asignación de procesos a GPUs, o cambiando físicamente las GPUs de los puertos PCI han mostrado diferencias de rendimiento menores al 5%. En cualquier caso, el uso del bus PCI para una tarea representaba el 10% del tiempo total de ejecución de la tarea, así que la mejora de 4,97 veces que se obtiene usando 8 tareas supone un uso del bus PCI de un 50% del tiempo total, que no debería suponer un cuello de botella. Así, nuestra conclusión es que las 8 tareas simultáneas saturan el *driver* común a las dos GPUs, mientras que cuando solamente se ejecutan 2 tareas no se tiene este problema.

Cabe remarcar que cuando mejoramos el rendimiento de la GPU aumentando el grado de paralelismo a nivel de procesos, lo hacemos con el coste de un mayor uso de núcleos de cómputo

de CPU. Cabe recordar también que la fase 2 de la tarea T2*, que representa alrededor del 18% del tiempo de ejecución, es puramente de cómputo en CPU, y por tanto el “consumo” de capacidad de cómputo en CPU que hace la tarea T2* es importante. En el siguiente apartado analizaremos la ejecución combinada de la tarea T2* con otras tareas puramente de CPU para evaluar empíricamente el uso de la CPU que hace la tarea T2*. Esta dualidad en el uso de GPU y CPU hará que el diseño del plan de ejecución para el workload, combinando tareas que usan y que no usan la GPU, sea más complejo.

A continuación se presenta el mismo tipo de análisis anterior, pero aplicado a la etapa T4* del *workflow*. En la Tabla 4.6 presentamos los datos obtenido, y observamos como la variabilidad entre sujetos diferentes (1,55%) y entre repeticiones con los mismos datos de entrada (entre 0,15% y 0,24%) es ahora muy pequeña. El número de *kernels* ejecutados es siempre el mismo, para todas las repeticiones y para todos los sujetos. En definitiva, el comportamiento de la etapa T4* es muy homogéneo.

	Tiempo Total Ejecución [seg]	**Número de *Kernels* Ejecutados**	**Tiempo Acumulado *Kernels* [seg]**	**Tiempo Transferencia Host-Dev [seg]**	**Máximo Memoria GPU [MB]**
Sujeto1	534,6 ± 0,15%	5.100.000	20,57	4,23 ± 0	90
Sujeto2	520,2 ± 0,22%	5.100.000	20,97	4,23 ± 0	
Sujeto3	534,2 ± 0,24%	5.100.000	20,84	4,23 ± 0	
Desv. estándar	1,55%	0%	0,98%	0%	

Tabla 9: caracterización de la etapa T4*: tiempo total de ejecución, número y tiempo de cómputo de los *kernels*, tiempo de transferencia Host-Device y máximo consumo de memoria. Se usan tres sujetos diferentes de entrada y se realizan múltiples ejecuciones para cada sujeto.

Analizando los tiempos de las diferentes partes que intervienen en la ejecución de la etapa T4*, vemos que la suma del tiempo acumulado de ejecución de los *kernels* más el tiempo acumulado de la transferencias Host-Device es aproximadamente un 5% del tiempo total de ejecución. Analizando el perfilado de la ejecución, como hicimos con el de la tarea T2*, encontramos que las latencias de llamadas de la API de CUDA representan un enorme porcentaje del tiempo total de ejecución. Si dividimos el tiempo acumulado de los *kernels* (alrededor de 20,8 segundos) entre el número total de *kernels* (5,1 millones) obtenemos un tiempo promedio para ejecutar cada *kernel* de 4,08 microsegundos, que es un tiempo tres órdenes de magnitud inferior al de los *kernels* de la tarea T2*, y por tanto justifica el mayor impacto negativo de las latencias para gestionar la ejecución de los *kernels*.

En el caso de la tarea T4* el consumo de memoria llega a un máximo de 90MB en todas las ejecuciones realizadas, así que es posible ejecutar más de 20 tareas simultáneamente en la misma GPU. La tabla 4.7 muestra el rendimiento obtenido al escalar el número de instancias de la etapa T4* ejecutadas en paralelo, usando tanto una como dos GPUs. En la figura 4.10 se representa en forma de gráfica el *speedup* obtenido al aumentar el paralelismo. La productividad al usar 2 instancias en la misma GPU aumenta un 56% comparado al caso de usar una única instancia y apenas crece al aumentar a 3 y 4 el paralelismo de procesos compartiendo la GPU. Aumentar aún más el grado de paralelismo no solamente no mejora el rendimiento, sino que incluso lo empeora de forma muy significativa. Estos resultados confirman que la ejecución de la etapa T4* presenta una fuerte limitación para usar la GPU, que no es ni la capacidad de cómputo ni la capacidad de comunicación de datos entre Host y Device (que están muy lejos de estar saturados).

Etapa del *workflow*	**Número de instancias simultáneas**	**Tiempo Experimento [segundos]**	**Productividad [sujetos / 1000 segundos]**
T4*	1 (GPU0)	534	1,87
	2 (GPU0)	686	2,92 (1,56x)
	3 (GPU0)	963	3,12 (1,67x)
	4 (GPU0)	1272	3,14 (1,68x)
	6 (GPU0)	3106	1,93 (1,03x)
	1 (GPU0) + 1 (GPU1)	666	3,00 (1,60x)
	2 (GPU0) + 2 (GPU1)	1266	3,16 (1,69x)

Tabla 10: tiempo total del experimento en segundos y productividad obtenida en las ejecuciones con 1, 2, 3, 4 y 6 instancias simultáneas de la etapa T4* usando una GPU, y de 2 y 4 instancias simultáneas usando dos GPUs. En la última columna, el valor entre paréntesis indica la mejora relativa a la ejecución con una instancia

Por otro lado, la ejecución usando dos GPUs tiene un rendimiento muy parecido al que se obtiene si solamente se usa una GPU: con una instancia por GPU se mejora el rendimiento un 60%, mientras que dos instancias en la misma GPU mejora un 56%; y con 2 instancias por GPU se mejora un 69%, mientras con 4 instancias en una sola GPU la mejora es del 68%. Por tanto, el problema que limita el rendimiento no parece ser un recurso propio de cada GPU sino un recurso compartido por las dos GPUs. La explicación cabe encontrarla de nuevo en el *software*: el *driver* de gestión de las GPUs que gestiona las llamadas a la API de CUDA realizadas desde los dos multiprocesadores no es capaz de aceptar peticiones a un ritmo mayor. El resultado es un uso de

los recursos de cómputo disponibles en la GPU inferior al 7% de su capacidad.

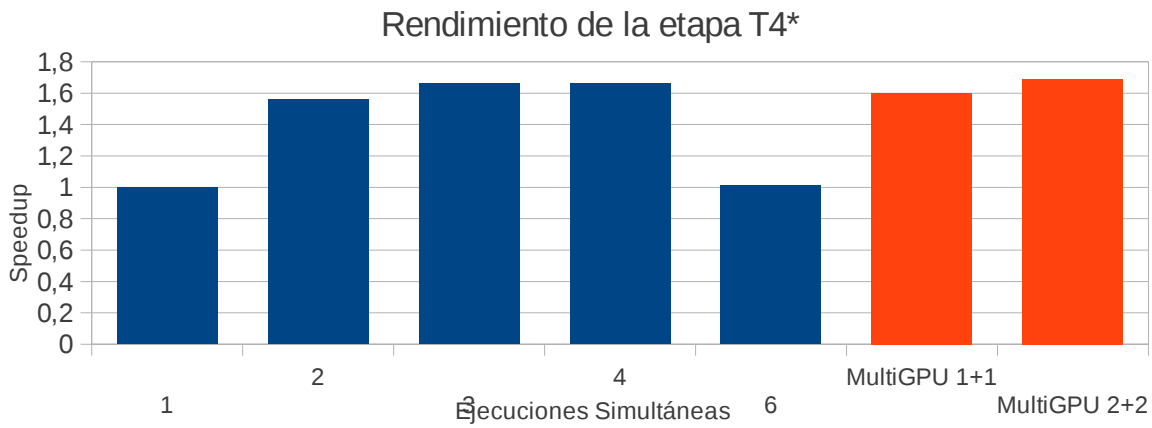


Figura 4.10: mejora obtenida en la ejecución de la etapa T4* al realizar múltiples ejecuciones simultáneas en una GPU (en el último caso en dos GPUs).

Como resumen de este apartado se debe destacar que, si bien el comportamiento de las tareas que usan solamente la CPU es bastante homogéneo, con un rendimiento que escala bien con el uso del *hyperthreading* y de múltiples núcleos de cómputo y procesadores; en cambio, el comportamiento del rendimiento de la GPU es mucho más complejo y menos intuitivo. Veremos en el próximo apartado cómo la selección de las combinaciones en la ejecución de las tareas puede suponer una importante diferencia en el rendimiento.

4.4 Afinidad entre etapas

En este apartado presentamos un análisis de afinidad entre las diferentes tareas que corresponden a las etapas del *workflow*. Ya vimos que la ejecución simultánea de un cierto número de instancias de la misma tarea producía mejoras muy importantes en el rendimiento. Mediante este análisis mostraremos que la ejecución combinada de ciertas tareas correspondientes a etapas diferentes es más productiva que si solamente se ejecutan instancias del mismo tipo de etapa. La explicación radica en que los elementos que resultan ser cuello de botella para una tarea pueden ser diferentes a los de otra tarea, de forma que se complementan durante su ejecución combinada. El objetivo final de este apartado es usar la información de afinidad en la definición del plan de ejecución, y lograr un mejor rendimiento global.

Para poder abordar el análisis de forma práctica, analizaremos la ejecución combinada solamente de dos tipos de etapas diferentes. Consideraremos que, o bien dos tareas diferentes comparten el mismo núcleo de cómputo físico (una tarea en cada una de las 2 CPUs virtuales), o bien las tareas comparten el mismo procesador, pero en diferentes núcleos físicos. La ejecución de tareas de

CPU en cada uno de los dos procesadores del sistema, tal como se ha visto anteriormente, genera poca contención (gracias en gran medida a fijar la afinidad a cada domino NUMA concreto), con una reducción máxima en el rendimiento del 6% debido al uso concurrente de los dos procesadores. Por tanto, las combinaciones de tareas que se realicen entre cada uno de los procesadores no tienen mucho margen para suponer un impacto importante en el rendimiento.

El uso de la GPU es más complicado de analizar, puesto que las tareas usan simultáneamente tanto la GPU como la CPU, y además el uso de las dos GPUs del sistema se ve muy afectado por la compartición del *software* de gestión (*driver*). De hecho, aunque la versión de una tarea en GPU sea más rápida que la versión en CPU, puede que una combinación de ambas sea la que mejor rendimiento global proporciona. Hemos decidido simplificar el análisis y centrarnos en este apartado en el uso de una única GPU. Veremos más adelante que, para el caso concreto del *workflow* que estamos analizando, el uso de una segunda GPU no aporta ninguna ventaja en la productividad del sistema.

La buena o mala afinidad entre dos etapas diferentes se mide usando la productividad (o *throughput*) que se obtiene al ejecutarlas de forma combinada respecto a la ejecución de cada etapa por separado. La existencia de una mala afinidad indica que las etapas están compitiendo por los mismos recursos HW y SW del computador.

4.4.1 Afinidad en el uso del *hyperthreading* en un núcleo de cómputo

En este apartado analizaremos la ejecución de todas las combinaciones de las tareas T1, T2*, T3, T4, T4* y T5; es decir, de todas las tareas excepto la tarea T2 ejecutada únicamente sobre CPU. Como la latencia de T2 es en promedio 8 veces superior a la de T2*, solamente tendría sentido ejecutar la tarea T2 si la capacidad global de ejecución en CPU fuera muchísimo mayor que la capacidad de ejecución en GPU, que no es el caso.

El rendimiento de la ejecución combinada de dos tareas no es simple de cuantificar, ya que el ritmo al que se ejecuta cada una de las dos tareas suele ser diferente, y en la ejecución combinada una de las tareas acabará siempre antes que la otra tarea, y en algunos casos bastante antes. En la Tabla 4.8 se muestran los resultados, medidos en sujetos procesados cada 1000 segundos. Cada fila de la tabla muestra el rendimiento para la tarea indicada a la izquierda. El primer valor de la fila, etiquetado con el símbolo ‘Ø’, representa el rendimiento de la tarea cuando se ejecuta sola, sin combinar, en un único núcleo de ejecución (sin utilizar la capacidad de *hyperthreading*). El resto de valores de la fila representan el rendimiento de la tarea indicada a la izquierda al combinarla con la tarea indicada en lo alto de la columna, siempre usando un único

núcleo de cómputo. La diagonal sombreada representa el rendimiento de combinar una tarea consigo misma, y hay que multiplicarlo por dos (sumar el rendimiento de cada una de las dos tareas del mismo tipo) para compararlo con el rendimiento de la ejecución mono-tarea (primera columna).

Sujetos procesados / 1000 seg	**Ø**	**T1**	**T2***	**T3**	**T4**	**T4***	**T5**
T1	0,30	0,18	0,19	0,19	0,18	0,21	0,19
T2*	0,67	0,54	0,46	0,56	0,51	0,52	0,51
T3	6,99	4,18	4,30	4,32	4,17	4,72	4,23
T4	0,58	0,35	0,38	0,37	0,35	0,40	0,36
T4*	1,87	1,48	1,31	1,49	1,47	1,16	1,47
T5	0,71	0,46	0,42	0,42	0,45	0,46	0,43

Tabla 11: productividad obtenida por cada tarea (fila) al combinar su ejecución con otra tarea (columna) en el mismo núcleo de cómputo (una tarea en cada CPU virtual) medida en tareas procesadas cada 1000 segundos de ejecución. La columna etiquetada "Ø" indica la ejecución sin combinar (una única tarea ejecutada en el núcleo de cómputo).

Ilustraremos el uso de la información de la tabla con un ejemplo. La tarea T1, en la primera fila, tiene un rendimiento de 0,30 tareas cada 1000 segundos al ejecutarse sin combinar, y un rendimiento de 0,18 + 0,18 = 0,36 tareas cada 1000 seg. al combinar su ejecución con otra tarea T1. Este valor ya aparecía en la Tabla 4.3, y supone una mejora de 1,21x en la productividad debida al uso de la capacidad de *hyperthreading*. La combinación de T1 con tareas diferentes proporciona para T1 un rendimiento igual o superior (entre 0,18 y 0,21) que si se combina consigo misma (0,18). Así, es conveniente combinar la ejecución de T1 con otras tareas, sobre todo T4*, en lugar de combinarla consigo misma. En cambio, la tarea T3 combina mejor consigo misma (4,32) que con el resto de tareas que solamente usan CPU (entre 4,17 y 4,23), y solamente da mejor rendimiento cuando se combina con T4*. Esto nos indica que todas las etapas excepto T3 (la de menor latencia total, y por tanto la menos importante) tienen una buena afinidad al ejecutarse conjuntamente, por lo que mezclar la ejecución de etapas diferentes es una buena opción.

La Figura 4.11 refleja los casos de ejecución separada y ejecución combinada para dos etapas diferentes (T1 y T5), utilizando los datos de la Tabla 4.8. En la ejecución separada se ejecutan dos

tareas T1 y luego dos tareas T5: para calcular los tiempos de ejecución se utilizan las productividades por separado (0,18 para ejecutar 2 tareas T1 y 0,43 para ejecutar 2 tareas T5). A partir de las productividades combinadas (0,19 para T1 y 0,46 para T5) se puede calcular que cuando finaliza una tarea T5 la tarea T1 se ha completado al 42,2%. La mejor manera de ejecutar dos etapas de cada tipo y maximizar el tiempo en que dos etapas diferentes se ejecutan a la vez es alternar la ejecución de T1 y T5 en cada CPU virtual del núcleo de cómputo, tal como se muestra en la figura. Así, la predicción de rendimiento para la ejecución combinada es un 3% mayor que para la ejecución separada. La mejora es algo menor a lo que se podía esperar de los datos de la tabla (alrededor del 5%) porque la ejecución combinada de T1 y T5 (2.222,2 + 2.222,2 seg.) solamente supone un 58% del tiempo total (7.655,4 seg.), mientras que el resto del tiempo se ejecutan dos tareas T1.

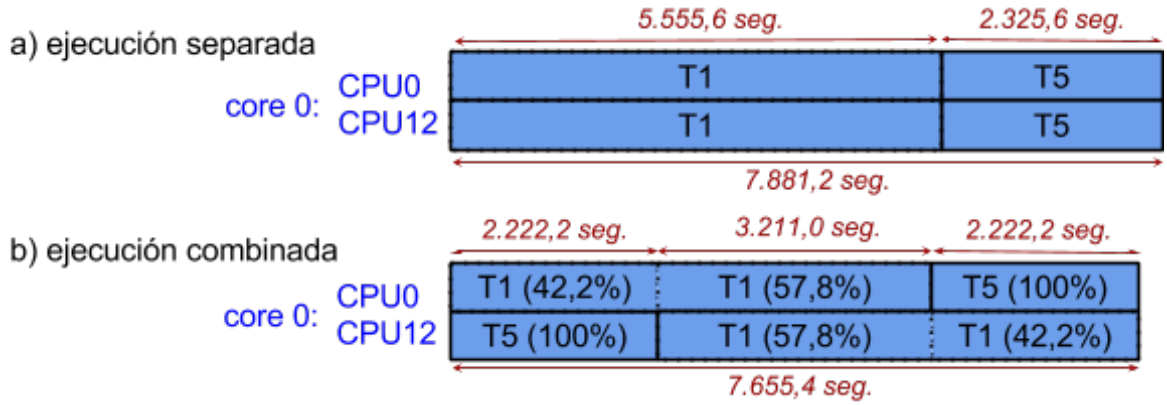


Figura 4.11: ejecución de dos tareas (T1 y T5) en cada CPU virtual de un mismo núcleo de cómputo. Los datos temporales provienen de las productividades promedio mostradas en la Tabla 4.8. La opción a) muestra la ejecución de tareas diferentes por separado. La opción b) representa la ejecución combinada de tareas: como la tarea T1 tiene una mayor latencia, es necesario alternar la ejecución de 2 tareas de cada tipo y hay un porcentaje del trabajo asociado a la tarea T1 que no se puede combinar con T5.

Si se analiza la ejecución de las tareas T2* y T4*, que además de usar la CPU también usan la GPU, también se puede calcular que la mejora de la ejecución combinada (T2* y T4* ejecutadas en el mismo núcleo el 53,8% del tiempo, y T2* con T2* el 46,2% restante) es del 7% respecto a la ejecución separada (dos tareas T2* a la vez y luego otras dos tareas T4* a la vez). En este caso la afinidad permite un mejor uso tanto de la CPU como de la GPU.

4.4.2 Escalabilidad de la ejecución multi-núcleo

Una vez analizada la mejora obtenida al usar la capacidad de *hyperthreading* con etapas combinadas, siempre positiva, vamos a analizar la escalabilidad del rendimiento al usar los múltiples núcleos de cómputo del procesador. La Tabla 4.9 muestra el *speedup* obtenido tanto en la ejecución separada como en la ejecución combinada, para cada tipo de tarea que se ejecuta solamente en CPU (sin considerar la tarea T3, cuya latencia es muy pequeña comparada con las

otras tareas). La escalabilidad es prácticamente la misma si se ejecuta por separado o combinado, y un poco diferente dependiendo de la tarea concreta. La conclusión del análisis es que primero hay que buscar la solución más adecuada para usar un núcleo de cómputo y después simplemente escalar la solución para usar los núcleos del mismo procesador.

Etapas ejecutadas	***Speedup* Ejec. Separada con 6 núcleos**	***Speedup* Ejec. Combinada con 6 núcleos**
T1	5,16x	5,22x
T4	5,20x	5,20x
T1	5,16x	5,20x
T5	4,94x	4,91x
T4	5,20x	5,22x
T5	4,94x	4,91x

Tabla 12: mejora del rendimiento al usar 6 núcleos de cómputo relativa a la ejecución con un núcleo de cómputo, tanto con la ejecución separada de cada tipo de etapa como con la ejecución combinada (solamente tareas CPU).

Escalar el uso de la GPU no es tan sencillo. Lo más importante es que, dependiendo de la carga de trabajo, existen dos escenarios opuestos (y un amplio rango de escenarios intermedios) que requieren dos criterios contrapuestos. Si la GPU es el recurso cuello de botella, nos interesa la combinación que consiga más rendimiento del uso de la GPU, usando los núcleos de CPU que sean necesarios; es decir, nos interesa acabar las tareas GPU en el menor tiempo posible (escenario *best-GPU-effort*). En cambio, en el otro extremo, si el volumen de trabajo asignado a la GPU es muy inferior a su capacidad, y son los núcleos de CPU los que son el cuello de botella, entonces nos interesa la combinación que haga un uso más eficiente de los núcleos de cómputo; es decir, interesa ejecutar las tareas CPU lo antes posible, mientras concurrentemente se van ejecutando las tareas GPU (escenario *higher-CPU-efficiency*). En un escenario intermedio es más complicado encontrar el punto óptimo entre eficiencia en CPU y baja latencia en GPU.

Por otro lado, y según los datos obtenidos en la caracterización de etapas del subapartado 4.3.2, el límite de instancias simultáneas por GPU para la etapa T2* es de 4, debido a restricciones de capacidad memoria, y también de 4 para la etapa T4*, pero ahora debido a que el rendimiento se satura a partir de este valor debido a limitaciones de la gestión de la API de CUDA. Las combinaciones de etapas T2* y T4* también están limitadas por la capacidad de la memoria, y solamente se pueden ejecutar simultáneamente tres tareas de cada tipo.

Hemos realizado combinaciones de las etapas T2* y T4* usando de uno a seis núcleos de cómputo, y usando o no la capacidad de *hyperthreading*. La Tabla 4.10 muestra la productividad de aquellas combinaciones que resultan más relevantes (algunos resultados ya habían sido presentados con anterioridad, y se presentan para facilitar el análisis).

Sujetos procesados / 1000 seg	**1 núcleo x 1 *thread***	**1 núcleo x 2 *threads***	**2 núcleos x 1 *thread***	**2 núcleos x 2 *threads***	**4 núcleos x 1 *thread***	**6 núcleos x 1 *thread***
T2*	0,67	0,92 (1,38x)	1,20 (1,79x)	1,64 (2,45x)	2,04 (3,04x)	---
T4*	1,87	2,31 (1,23x)	2,92 (1,56x)	2,88 (1,54x)	3,14 (1,68x)	1,93 (1,03x)
T2*	---	0,52	0,67 (1,29x)	1,03 (1,98x)	1,22 (2,35x)	1,67 (3,21x)
T4*		1,31	1,86 (1,42x)	1,67 (1,27x)	1,85 (1,41x)	1,66 (1,27x)

Tabla 13: productividad obtenida por cada tarea (fila) al escalar la ejecución en varios núcleos de cómputo usando o no las dos CPUs virtuales de cada núcleo, medida en tareas procesadas cada 1000 segundos de ejecución. Entre paréntesis se muestra el *speedup* respecto a la ejecución de una única tarea.

La lectura de los datos de la Tabla 4.10 nos indica que el uso de *hyperthreading* mejora la eficiencia en el uso de los núcleos de cómputo, pero no representan la opción más rápida en GPU. Por ejemplo, la ejecución de 4 tareas T2* usando cuatro núcleos es más rápida (2,04) que usando dos núcleos con *hyperthreading* (1,64). La escalabilidad en GPU no es buena, y consume los recursos del procesador de forma cada vez menos eficiente.

Hemos analizado las posibilidades a partir de la Tabla 4.10 y mostramos las más interesantes en la Tabla 4.11. La mejor configuración en un escenario *best-GPU-effort* se muestra en la primera fila, y consiste en usar los seis núcleos de cómputo con 3 tareas T2* y 3 tareas T4*, consiguiendo ejecutar, en promedio, una tarea T2* y una tarea T4* cada 601,67 segundos (es decir, 1,66 etapas T2* y T4* cada 1000 segundos). Si dividimos la productividad por los seis núcleos CPU utilizados tenemos 0,2777 etapas T2* y T4* ejecutadas cada mil segundos por núcleo utilizado. La ejecución no combinada está en la tercera fila y usa 4 núcleos de cómputo, pero es peor que la ejecución combinada de la fila 2, que también usa 4 núcleos de cómputo. Esta última es más eficiente en el uso de los núcleos CPU que la primera (0,3534 frente a 0,2777), pero tiene mayor latencia (707,47 frente a 601,67). Por supuesto, ejecutar la etapa T4 en CPU en lugar de en GPU (T4 en vez de

T4*) sería una forma de reducir aún más el uso de la GPU para un escenario aún más estricto. Finalmente, las configuraciones que usan uno o dos núcleos de CPU (siempre es positivo combinar la ejecución de T2* y T4* el mayor tiempo posible) son más eficientes en el uso de la CPU, pero alargan la ejecución en GPU. La penúltima configuración resulta ser muy adecuada como compromiso entre latencia y eficiencia, ya que aumenta un 38% la latencia respecto al mejor caso (fila 1), pero con una eficiencia que es un 116% mejor, y que es solamente un 17% peor que el caso de mejor eficiencia (fila 5), que tiene una latencia un 70% mayor.

Configuraciones de Ejecución	**Nº núcleos de CPU**	**tiempo por tarea T2* y T4***	**Productividad / nº núcleos CPU**
3T2* + 3T4* (99,3%) ; 4T4* (0,7%)	6 y 4	601,67 segundos	0,2777 T2*+T4* / seg
2T2* + 2T4* (76,4%) ; 4T4* (23,6%)	4	707,47 segundos	0,3534 T2*+T4* / seg.
4T2* (60,6%) ; 4T4* (39,4%)	4	808,67 segundos	0,3091 T2*+T4* / seg.
2x(T2*+T4*) (72%) ; 2x(T2*+T2*) (28%)	2	832,89 segundos	0,6003 T2*+T4* / seg.
T2* + T4* (53,8%) ; T2*+T2* (46,2%)	1	1.418,8 segundos	0,7048 T2*+T4* / seg.

Tabla 14: para cada configuración de ejecución se muestra entre paréntesis el porcentaje de tiempo que se usa, el número de núcleos de cómputo utilizados, el tiempo promedio para ejecutar dos tareas, una T2* y una T4*, y la productividad normalizada, medida en tareas T2* y T4* procesadas cada 1000 segundos por núcleo utilizado.

Como resumen y conclusión final de este apartado debemos remarcar que las ejecuciones combinadas son en general la mejor opción, tanto dentro del núcleo de CPU como al escalar a muchos núcleos, pero que el uso de la GPU es más complejo, y supone encontrar el punto adecuado de equilibrio entre la latencia y la eficiencia de uso de la CPU. En el siguiente apartado abordaremos en su totalidad el problema que hemos planteado al inicio del capítulo, y retomaremos la propuesta esbozada en su introducción.

4.5 Diseño del plan de ejecución

En el apartado anterior hemos realizado una caracterización del rendimiento de las etapas del *workflow* de FreeSurfer, considerando la afinidad al ejecutar tareas que comparten el mismo

núcleo de cómputo, o un núcleo diferente dentro del mismo procesador, o la misma GPU. En este apartado describiremos y utilizaremos la metodología para crear un plan de ejecución estático basado en el análisis previo. Dependiendo de la capacidad de ejecución en CPU y en GPU que tenga el sistema de cómputo, deberemos priorizar el uso de la GPU (*best-GPU-effort*) o el uso de la CPU (*higher-CPU-efficiency*). También propondremos un esquema de planificación con listas separadas asociadas a diferentes conjuntos de recursos de cómputo, que incorporan un mecanismo de equilibrado de la carga de trabajo. Comenzaremos analizando el caso para el sistema más simple, un único núcleo de CPU con una única GPU, y luego generalizaremos la propuesta a sistemas más complejos.

4.5.1 Planificación de la ejecución en un único núcleo de cómputo

En el caso de tener un núcleo de CPU (con *hyperthreading*) y una GPU, la mejor planificación vendrá determinada por la selección de aquellas combinaciones de tareas con mejor afinidad, tal y como se desprende de los resultados mostrados en la Tabla 4.8. Diseñaremos la planificación de un lote de n=5 etapas comenzando por la tarea T1, porque es la que tiene una mayor latencia de ejecución. La tarea T5, la segunda de mayor latencia, se planifica para ser ejecutada conjuntamente con T1, ya que el rendimiento combinado es mejor que el rendimiento por separado. Utilizando el mismo criterio se planifican el resto de tareas, y se obtiene el esquema mostrado en la Figura 4.12

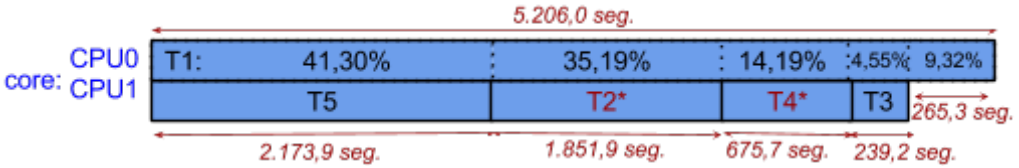


Figura 4.12: planificación estática promedio de la ejecución de las 5 etapas de FreeSurfer usando las dos CPUs virtuales de un núcleo de cómputo (los datos temporales provienen de la Tabla 4.8) y una GPU (las tareas que usan GPU se muestran en rojo y con *). Los porcentajes representan la proporción de la tarea T1.

En este ejemplo concreto, la carga promedio entre las dos CPUs virtuales está repartida de forma bastante equilibrada, y se decide finalizar el lote de tareas, haciendo que el valor w (el grado de paralelismo de ejecución) sea 1. En el caso de que el resultado quede desbalanceado se aumentaría el valor de w, para tener más tareas con las que equilibrar la carga entre las dos CPUs virtuales.

Es necesario un mecanismo que compense dinámicamente la diferencia entre las cargas promedio de cada CPU virtual. Como primera idea, proponemos alternar la carga entre los lotes pares e impares, tal como se muestra en la Figura 4.13. De este modo, la predicción de tiempo de

ejecución en cada CPU virtual se equilibraría perfectamente. Considerando las Figuras 4.12 y 4.13, el tiempo estimado de ejecución por sujeto sería de (5.206,0 + 5.206,0 - 265,3) / 2 = 10.146,7 / 2 = 5.073,35 segundos por sujeto.

Figura 4.13: ejecución alternada entre las CPUs virtuales para los lotes pares e impares.

Sin embargo, esta estrategia no es suficiente en un caso general: es necesario un mecanismo que compense las variaciones en el tiempo de ejecución que aparezcan sobre el promedio de tiempo analizado, y que en algunos casos hemos visto que podían llegar a ser superiores al 10% (Tablas 4.1 y 4.4). Para ello proponemos usar listas estáticas de tareas y planificación dinámica de tareas al llegar al final de la lista. Como esquema preliminar que funcionaría para este caso simple proponemos el siguiente mecanismo y política de planificación:

- cada lote de ejecución se compone de dos listas de tareas, Lista 1 = { T11, T12, ... T1r } y Lista 2 = { T21, T22, ... T2s }, definidas de manera estática, donde la primera lista, que denominaremos lista prioritaria, tiene un tiempo promedio de ejecución estimado mayor al de la segunda lista, y le daremos prioridad al comenzar la ejecución de cada nuevo lote

- se asigna cada lista del lote a cada una de las CPUs virtuales del núcleo de ejecución

- cuando una CPU queda desocupada y tiene asignada la lista X

    a. si la lista X tiene tareas pendientes de ejecutar: ejecutar la siguiente tarea en la CPU que ha quedado ociosa (usando la GPU, si es necesario)

    b. si la otra lista (lista Y) ya ha sido consumida y su última tarea está en ejecución: se comienza a procesar un nuevo lote, asignando la lista 1 (prioritaria) a la CPU que ha acabado antes y la lista 2 (no prioritaria) a la CPU que aún está activa. Por tanto, se ejecuta la primera tarea del lote de la lista prioritaria en la CPU que ha quedado ociosa

    c. si la otra lista (lista Y) tiene tareas pendientes de enviar a ejecutar: se “roba” la última tarea no ejecutada de la lista Y y se ejecuta en la CPU que está desocupada

La Figura 4.14 muestra tres posibles situaciones para el caso que estamos analizando: las dos primeras corresponden al caso en que las dos listas finalizan casi a la vez, y la última representa el caso en que una de las listas tiene trabajo pendiente. Las dos primeras son situaciones esperadas: es muy improbable acabar simultáneamente, aunque la carga esté equilibrada. Simplemente comenzamos la ejecución de un nuevo lote, empezando por la lista prioritaria, y asignándola a la CPU que queda antes desocupada. La tercera situación corresponde al caso en que la lista 2 se demora más de lo previsto: para compensar, se adelanta la ejecución de una tarea pendiente de la lista 2 (la última no ejecutada

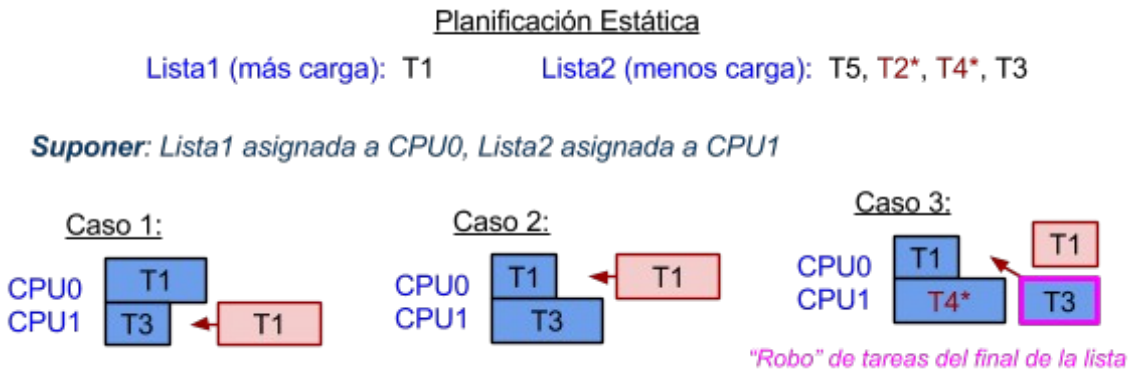


Figura 4.14: planificación usando dos listas estáticas de tareas y usando un mecanismo de compensación para amortiguar el efecto de las diferencias en el tiempo de ejecución respecto a la predicción estática.

La política dinámica de “compensación” de tareas entre listas reduce la cantidad de tiempo durante la que se solapa la ejecución de tareas pertenecientes a lotes consecutivos, y equilibra la carga de cómputo de forma natural, asegurando una alta ocupación de los recursos. También reduce la posibilidad de tener que frenar la ejecución para asegurar una gestión correcta de las dependencias entre tareas. A cambio, esta política dinámica introduce un problema potencial: pueden llegar a ejecutarse de forma simultánea tareas de la lista no prioritaria, que en principio no deberían hacerlo. Así, el uso de esta política dinámica impide utilizar la planificación estática como único mecanismo que asegure un uso máximo de la memoria de la CPU y/o (sobre todo) de la GPU. La planificación estática puede hacer todo lo posible para limitar el máximo uso compartido de recursos, pero se deberá añadir un control adicional para asegurar este límite. Este mecanismo adicional de control del uso de los recursos, idéntico al que proponíamos en el capítulo 3, será necesario cuando, de forma no prevista, se ejecutan simultáneamente tareas de la lista no prioritaria, y puede provocar que en ciertas situaciones se frene la ejecución y que aparezcan huecos en el uso de los núcleos de cómputo. La definición del plan estático de ejecución, basada en la predicción del tiempo promedio de ejecución, debe tratar de reducir el tiempo acumulado de estos casos.

Volviendo al ejemplo de las Figuras 4.12-4.14, la planificación asume que las tareas T1 de diferentes lotes se puedan solapar (en promedio sería un porcentaje cercano al 9%), pero no prevee que, por ejemplo, las tareas T2* y T4*, que usan la GPU, se ejecuten a la vez. Aunque la posibilidad de que esto ocurra es bastante pequeña, el problema no es únicamente que la ejecución pueda ser ineficiente sino que la GPU no disponga de suficiente memoria para la ejecución conjunta, y en consecuencia alguna de las tareas aborte de forma descontrolada. Una forma de actuar para reducir la probabilidad de ocurrencia de este caso sería reordenar las tareas de la lista 2 = { T4*, T2*, T5, T3 }, de forma que las tareas que usan la GPU se ejecutan lo antes posible dentro del lote de ejecución.

4.5.2 Planificación de la ejecución en varios núcleos de cómputo

En el caso de usar varios núcleos de CPU y una GPU, una posible opción para diseñar la planificación es simplemente replicar la planificación usada para un núcleo. Sería equivalente a replicar el esquema mostrado en la Figura 4.12. Sin embargo, esta planificación tiene un problema en el uso de la GPU: podrían coincidir tantas tareas que usan la GPU ejecutándose a la vez como núcleos de cómputo usados en el sistema (además de la posibilidad menos frecuente de conflictos debidos al mecanismo dinámico de “compensación” de tareas).

En la Figura 4.12 se muestra que la predicción de uso de la GPU durante un lote es del 49,82% (el tiempo de uso de la GPU para un sujeto es de 1.851,9 + 675,7 = 2.527,6 segundos sobre 5.073,35 segundos del total). Como este porcentaje no llega al 50%, proponemos crear una lista específica con tareas que usan la GPU. El plan de ejecución estará ahora compuesto por cuatro listas, tal como se muestra en la Figura siguiente.

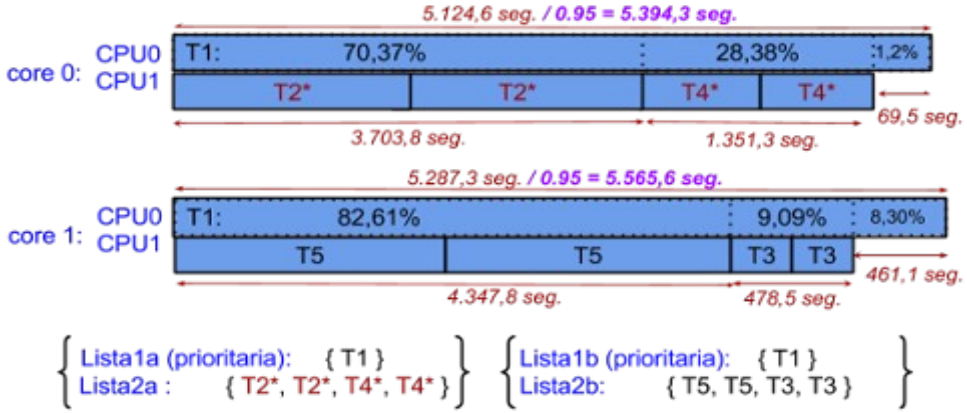


Figura 4.15: planificación estática de la ejecución de las 5 etapas de FreeSurfer usando dos núcleos de cómputo con 4 CPUs virtuales y una GPU (los datos temporales provienen de la Tabla 4.8). Se define una jerarquía de 2 listas para cada núcleo. Los cálculos de tiempo promedio usan los datos suponiendo que no hay interacción entre las ejecuciones en los dos núcleos de cómputo y se corrigen aplicando un factor de degradación estimado (Tabla 4.9).

Las listas 1a y 2a se asociarán a cada CPU de un núcleo de cómputo (con un tiempo estimado total promedio de 5.089,85 segundos por lote de tareas) y las listas 1b y 2b se asociarán al otro núcleo de cómputo (estimación de 5.056,80 segundos por lote de tareas). Estos tiempos utilizan los datos de la Tabla 4.8 sobre rendimiento combinado dentro de un núcleo de cómputo, y no tienen en cuenta la interacción entre los dos núcleos de cómputo (que comparten la memoria cache de último nivel y la memoria principal). La Tabla 4.9 nos proporciona datos sobre la escalabilidad en el rendimiento en la CPU al usar más núcleos, mientras que la Tabla 4.10 nos indica la escalabilidad usando la GPU. A partir del factor de degradación en el rendimiento de 0.87 usando 6 núcleos de CPU, extrapolamos a un factor de 0.95 la degradación usando 2 núcleos. Así esperaríamos unos tiempos aproximados de 5.357,7 y 5.322,9.

Para este ejemplo tendríamos ahora un valor de w (grado de paralelismo) de 2 sujetos completos por lote. De nuevo, si la carga de cómputo hubiera quedado muy desbalanceada se podría aumentar el valor de w.

Aparece ahora la necesidad de equilibrar la carga entre núcleos. La propuesta de balanceo se generaliza de forma jerárquica: en primer lugar se balancea la carga de forma dinámica dentro del núcleo de cómputo, y si las listas asociadas al núcleo de cómputo actual ya se han consumido, entonces se balancea la carga sobrante en las listas de los otros núcleos de cómputo. Si la CPU que queda ociosa estaba ejecutando tareas de una lista prioritaria (no prioritaria), preferiblemente se roban tareas prioritarias (no prioritarias).

En el caso representado en la Figura 4.16, la CPU1 del núcleo 0 queda ociosa, y ya se han consumido todas las tareas del lote asociado al núcleo 0, pero en el núcleo 1 la lista 2b tiene una tarea T3 pendiente. En esta situación se permite el balanceo de carga, “robando” la tarea y ejecutándola en la CPU1 del núcleo 0.

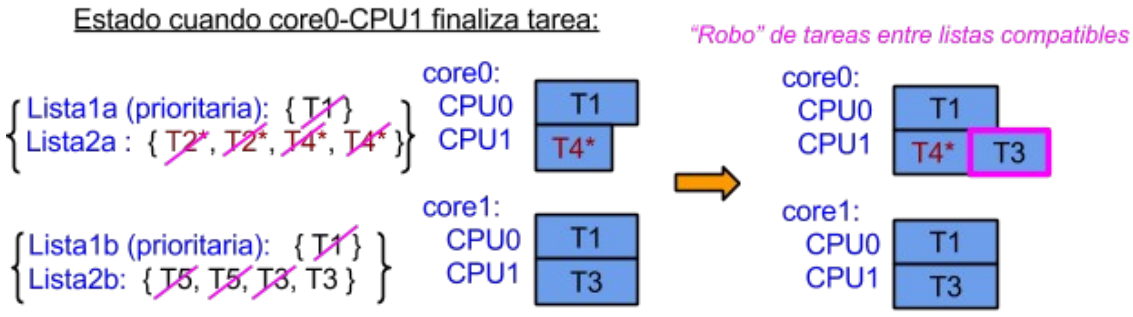


Figura 4.16: mecanismo de compensación entre listas de tareas asociadas a núcleos de cómputo diferentes.

Una vez establecido el mecanismo de coordinación entre núcleos de cómputo se puede crear un plan estático de ejecución para utilizar los 6 núcleos de un procesador y una GPU, simplemente

replicando 3 veces la propuesta de la Figura 4.14. Esta propuesta tentativa se muestra en la Figura siguiente y corresponde con un valor de w (grado de paralelismo) de 6 sujetos completos por lote.

Figura 4.17: propuesta tentativa de planificación estática de la ejecución de las 5 etapas de FreeSurfer usando 6 núcleos de cómputo. Se definen 6 grupos de listas entre las que se utiliza el mecanismo dinámico de compensación.

La estimación del tiempo promedio esperado para cada lote de ejecución tiene que ser de nuevo escalada al uso de seis núcleos de cómputo de forma simultánea. El escalado en CPU es simple, ya que debemos usar los factores encontrados en el Tabla 4.9, que son muy similares para todos los tipos de tareas (mejoras de entre 5,20 y 5,26, que representan una eficiencia de entre 0,86 y 0,87). El escalado en GPU, en cambio, es muy dependiente del tipo de tareas que se mezclan. Por ejemplo, de la Tabla 4.10 se puede extraer que la coincidencia de 3 tareas T2* tiene una eficiencia de 0.84, mientras que para 3 tareas T4* la eficiencia es de 0.56, bastante más baja. La eficiencia de usar a la vez una tarea T2* y una T4* es de 0.99.

La propuesta de planificación de la Figura 4.17, dado el orden de las listas, favorece las combinaciones de 3 tareas T2* (relativamente eficiente) pero también las combinaciones de 3 tareas T4* (poco eficiente). Se pueden reordenar las tareas de las listas 2a, 2c y 2e para favorecer las ocurrencias más beneficiosas, pero las variaciones respecto a los tiempos promedio de las tareas pueden generar combinaciones no deseadas durante periodos de tiempo largos.

Para poder controlar mejor estas combinaciones se propone una última variación en el mecanismo de planificación: asociar una lista de tareas a más de una CPU del sistema, de forma que la lista “alimente” de forma indistinta a ese grupo de CPUs. La Figura 4.18 muestra la propuesta final de planificación para un lote de tareas de w = 6 sujetos. En breve se presentará el mecanismo con detalle, pero la idea general es que las tareas de cada lista se ejecuten en un conjunto predefinido de CPUs, en este caso de 6 o 3 CPUs, y que dentro de estas CPUs se haga el balanceo dinámico de la carga de forma natural. Por ejemplo, la lista 2a, que contiene las tareas que usan la GPU, se ordena para que tres tareas consecutivas de la lista nunca puedan ser de tipo T4*, evitando la configuración más perjudicial, y promoviendo los casos de ejecución más eficientes de dos tareas

T2* con una tarea T4*, o de tres tareas T2*. Las variaciones de tiempo entre las tareas de la lista se amortiguan mediante el mecanismo dinámico de asignación a las 3 CPUs. El mecanismo de compensación de carga entre listas solamente se activará si la suma de todos los tiempos de las tareas de la lista crece de forma no prevista.

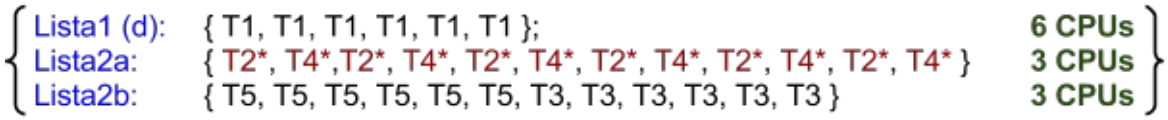


Figura 4.18: propuesta final de planificación estática con tres listas para ejecutar múltiples instancias del *workflow* de 5 etapas de FreeSurfer usando 6 núcleos de cómputo y una GPU. La lista1 tiene prioridad al comenzar un nuevo lote, utiliza 6 CPUs, y cada una de un núcleo de ejecución distinto. Las listas 2a y 2b utilizan 3 CPUs cada una de ellas, pero siempre de núcleos de cómputo diferentes.

La tabla 4.12 muestra la estimación del tiempo promedio esperado en la ejecución con 6 núcleos. Para las tareas de CPU se usa la eficiencia medida empíricamente en la tabla 4.2, que corresponde a la ejecución en solitario de cada tipo de tarea, pero que es muy similar a la que se obtenía en la tabla 4.9 en las ejecuciones combinadas. Para las tareas de GPU, como el tiempo de la tarea T2* ejecutada en solitario es 2,74 veces el tiempo de la tarea T4*, se puede analizar el despliegue de las combinaciones que aparecen, y suponen aproximadamente un 80% del tiempo la combinación 2×T2*+T4* y el 20% la combinación 3×T2*. Así, el factor de escalado para las tareas T2* se reparte proporcionalmente para los dos casos, mientras que el de la tarea T4* corresponde al caso de la combinación 2×T2*+T4*.

Tareas	tiempo previsto para 1 núcleo	factor de escalado	tiempo final
T1	5124,6 seg.	5,16 / 6 = 0,86	5958,8 seg
T2*	1851,9 seg.	0,88×80% +0,84×20%=0,87	2128,6 seg.
T3	239,25 seg.	5,26 / 6 = 0,88	271,9 seg.
T4*	675,65 seg.	0,75	900,9 seg.
T5	2173,9 seg.	4,94 / 6 = 0,82	2651,1 seg.
TOTAL	(5958,8+2128,6+271,9+900,9+2651,1) / 12 CPUs = 992,6 seg. / sujeto		

Tabla 15: escalado del tiempo promedio de ejecución de cada tarea al pasar de usar un núcleo de cómputo (Figura 4.15) a usar 6 núcleos de cómputo, usando los factores de las Tablas 4.2, 4.9 y 4.10.

Para finalizar este subapartado, se precisa a continuación el algoritmo que realiza la política de

planificación con listas de tareas generalizado a un sistema con múltiples núcleos de cómputo:

- cada lote de ejecución se compone de varias listas de tareas, definidas de manera estática, donde cada lista tiene asociado el número de CPUs que necesita. Las listas se clasifican como prioritarias o no prioritarias.
- inicialmente se asigna cada lista del lote a tantas CPUs virtuales como tiene asociadas, de forma que a cada núcleo de ejecución se asocia una lista prioritaria y otra no prioritaria
- cuando una CPU queda desocupada y está asignada a la lista X:
    - si la lista X contiene más tareas: ejecutar la primera tarea pendiente en la CPU
    - si la lista Y asociada al mismo núcleo tiene tareas pendientes de ejecutar: se “roba” la última tarea no ejecutada de la lista Y y se ejecuta en la CPU
    - si alguna lista Z en otro núcleo de cómputo tiene tareas pendientes de ejecutar: se “roba” la última tarea no ejecutada de la lista Z
    - si no: marcar todas las CPUs como no asignadas y comenzar el procesamiento de un nuevo lote (pasar al siguiente punto)
- cuando una CPU queda desocupada y no está asignada:
    - si no hay ninguna CPU asignada en el núcleo de cómputo: asignar la primera lista prioritaria que aún no tenga asignadas el número de CPUs que necesita
    - si ya hay una CPU asignada en el núcleo de cómputo: asignar la CPU a la primera lista no prioritaria que aún no tenga asignadas el número de CPUs que necesita

Además de los pasos indicados en el algoritmo de planificación anterior, cada vez que se envía a ejecutar una tarea, se controla que no se sobrepasen los límites en el uso de la memoria de la CPU y de la GPU, y que no se pueda comenzar un nuevo lote i+2 si aún hay tareas en ejecución del lote i.

Como resumen de este apartado, se ha propuesto un algoritmo de planificación dinámico, basado en un plan de ejecución estático, que considera la afinidad de la ejecución entre diferentes etapas y la carga de trabajo en CPU y GPU del lote de etapas a ejecutar. A continuación presentaremos los resultados obtenidos al procesar un gran número de sujetos utilizando la propuesta de este capítulo, y los compararemos con los obtenidos usando únicamente la propuesta desarrollada en el capítulo 3 de esta tesis.

El ejemplo real de ejecución dará lugar a una nueva ampliación de la propuesta de algoritmo de planificación, el uso de listas de tareas que denominaremos exclusivas. Estas listas servirán para controlar el uso de la GPU y acelerar el procesamiento en los casos en que la GPU se convierte en el recurso que limita el rendimiento. Consideramos que el momento adecuado para explicar esta nueva aportación es una vez analizado el caso real.

4.6 Resultados

En esta sección se presentan resultados experimentales de las propuestas de ejecución planificada de *workflows* científicos introducidas a lo largo de este capítulo. Se comparan los resultados obtenidos en tres escenarios: (1) usando un procesador -6 núcleos de cómputo con 2 *threads* por núcleo-, y una GPU, de forma que el rendimiento de la ejecución está limitado por la CPU, (2) usando dos procesadores y una GPU, para que el cuello de botella del rendimiento esté en la GPU, y (3) usando dos procesadores y dos GPUs. Con este análisis se corrobora que controlar el orden de ejecución de las etapas y el uso de la GPU, guiado por los resultados de un análisis más fino del uso que hace el *workflow* de los recursos, permite aprovechar las afinidades entre etapas y mejorar el uso de recursos, y aporta una reducción significativa del tiempo de ejecución.

Los experimentos realizados consisten en la ejecución de múltiples instancias del sub-*workflow* de reconstrucción volumétrica incluido dentro del *workflow* de FreeSurfer, ampliamente descrito en los capítulos iniciales de este trabajo. Para todas las ejecuciones realizadas se ha tomado una lista de 200 sujetos de entrada. El tiempo promedio por sujeto considera la parte inicial y final de la ejecución, es decir, los periodos transitorios de la ejecución, con un método que se explicará para cada caso.

La planificación *por lotes* definida durante este capítulo se compara con la ejecución secuencial de un único *workflow*, que se tomará como caso base, con la ejecución sin planificación controlada (que denominaremos multi-*workflow*), y con la planificación *inter-workflow* definida en el capítulo 3.

La ejecución multi-*workflow* deja que sea el sistema operativo linux el que distribuya los procesos entre los recursos y, debido a las restricciones de memoria de la etapa T2* que se ejecuta en GPU, debe limitarse a cuatro instancias del *workflow* de forma simultánea. Cada vez que una de las instancias finaliza su ejecución, inmediatamente se inicia la ejecución de una nueva instancia. En este caso el rendimiento usando uno o dos procesadores es el mismo, ya que el límite de 4 *workflows* a la vez impide aprovechar el segundo procesador de forma efectiva.

La planificación *inter-workflow* define un máximo de K=12 y 24 tareas o etapas simultáneas para los dos escenarios, es decir, K es igual al número de CPUs virtuales del sistema, de forma que se pueda ejecutar hasta un máximo de una tarea por CPU. Tal como se describió en el capítulo 3, para asegurar una ejecución segura de las tareas que usan GPU se definen restricciones de ejecución en forma de porcentajes de uso: 25% para las tareas T2* y 10% para las tareas T4*. Así, el límite permite hasta 4 instancias simultáneas de la tarea T2* o hasta 10 instancias simultáneas de la tarea T4* (o combinaciones que mezclan ambas). Recordamos aquí que la versión que tenemos del planificador no permite distinguir entre diferentes GPUs, así que aunque el uso de varias GPUs fuera apropiado, con este esquema no se podrían utilizar. En cualquier caso, el esquema propuesto en este capítulo mejora en varios aspectos al planificador *inter-workflow*, que incluimos aquí solamente a efectos de comparación.

Finalmente, la planificación *por lotes* utiliza las listas de tareas que se han descrito en el apartado anterior para el escenario 1, y utiliza otras configuraciones para los escenarios 2 y 3, que se describirán y justificarán rmás tarde.

4.6.1 Resultados Generales

La figura 4.19 presenta el *speedup* de cada caso respecto a la ejecución secuencial, usando una o dos de las GPUs del sistema. La ejecución multi-*workflow*, sin mecanismo adicional de planificación, obtiene una mejora de rendimiento algo inferior a cuatro (3,64), que sería lo máximo que podemos esperar al ejecutar 4 *workflows* concurrentemente. Probablemente, la migración de procesos entre CPUs que hace el sistema operativo no es la más adecuada, e impide alcanzar el rendimiento ideal. Tanto en la ejecución secuencial como multi-*workflow* se considera la ejecución completa desde el sujeto nº 1 al 200. Hemos comprobado experimentalmente que descartar entre 1 y 50 de los primeros y de los últimos sujetos apenas varía el tiempo promedio de ejecución por sujeto.

Las dos propuestas de planificación mejoran el rendimiento cuando se usa un procesador y una GPU: la planificación *inter-workflow* mejora 5,48 veces la ejecución secuencial, y la ejecución *por lotes* mejora 6,09 veces. Es decir, la planificación *por lotes* en este escenario mejora un 11,1% a la planificación *inter-workflow*. El uso de 2 procesadores acentúa aún más la mejora de la planificación *por lotes* respecto a la planificación *inter-workflow*, llegando a un *speedup* de 10,48x frente al de 8,27x; es decir, un 27% mejor. La razón de este mejor funcionamiento es que el rendimiento está limitado por la GPU, y que la planificación *por lotes* ofrece mecanismos, como las listas exclusivas y la migración de tareas de GPU a CPU, que permiten adaptar mejor la ejecución.

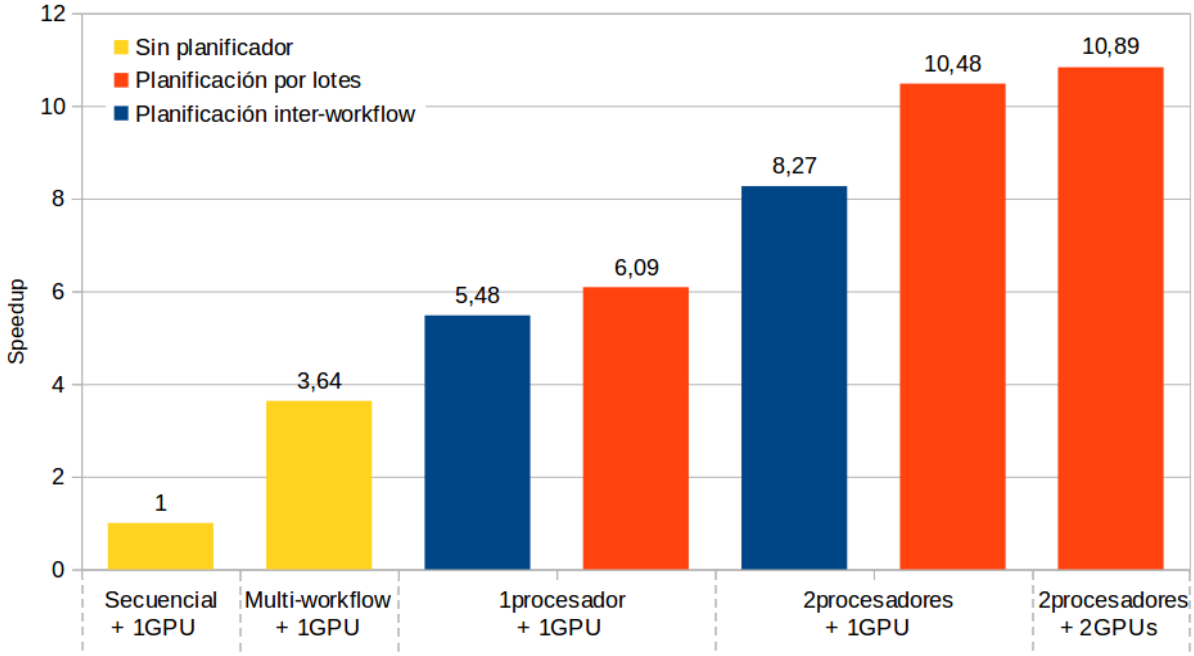


Figura 4.19: mejora de rendimiento respecto a la ejecución secuencial con GPU obtenida usando la ejecución concurrente de 4 instancias del *workflow* (multi-*workflow*), y las propuestas de planificación detalladas en el capítulo 3 (planificación *inter-workflow*) y 4 (planificación *por lotes*) de este trabajo, al usar uno o dos procesadores del nodo de cómputo con una y dos GPUs.

Finalmente, el *workflow* que analizamos prácticamente no se beneficia de usar una segunda GPU: logra un *speedup* total de 10,89x frente a 10,48x, que es un incremento de apenas el 4%. Como las tareas que usan la GPU también necesitan la CPU para ejecutar parte del código de la aplicación y para ejecutar el *driver* de la GPU, y el uso de GPU y CPU está bastante equilibrado en el escenario de dos procesadores con una GPU, la segunda GPU no ofrece una ganancia importante si a la vez no se aumenta el número de CPUs del sistema.

En la ejecución *inter-workflow* se ha comprobado experimentalmente que descartar parte de la ejecución inicial y final (entre 1.000 y 100.000 segundos) apenas varía el promedio de rendimiento, así que se calculan los resultados para la ejecución completa de los 200 sujetos. En el caso de la ejecución *por lotes* el periodo transitorio inicial es poco eficiente, ya que se limita el paralelismo de forma bastante artificial. Esta fase se podría acelerar fácilmente relajando las restricciones de la ejecución por lote y, por ejemplo, utilizando el esquema de ejecución *inter-workflow*. Como entendemos que esta fase transitoria no debería ser significativa en un entorno real de ejecución, no hemos profundizado en este análisis. Para los experimentos simplemente hemos descartado el periodo transitorio.

4.6.2 Resultados Detallados usando un procesador y una GPU

Para ilustrar el análisis detallado se presentan las trazas de Gantt con la secuencia de tareas ejecutadas a partir de un punto intermedio de la ejecución y durante un intervalo de tiempo de unos 30.000 segundos. Servirán para visualizar el funcionamiento de los mecanismos de planificación, así como sus posibles deficiencias.

La planificación *inter-workflow* restringe el número de tareas T2* y T4* ejecutadas en GPU a las que se puede garantizar que no superarán el límite de memoria de la GPU. En la configuración de un procesador (12 CPUs virtuales) y una GPU se obtiene una productividad de 1222 segundos por sujeto, que es un 23% inferior al cálculo teórico de 992,6 segundos por sujeto que se ha realizado en la tabla 4.12. La figura 4.20 muestra un fragmento de la traza de Gantt que comienza a partir del inicio de la ejecución del sujeto número 105.

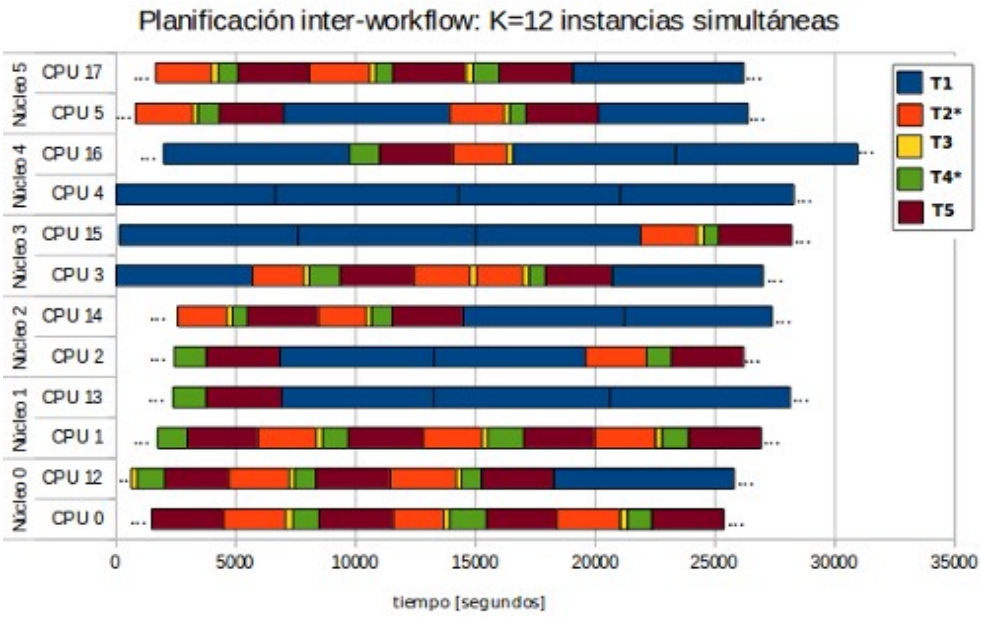


Figura 4.20: diagrama de Gantt para la ejecución de tareas en un procesador con 6 núcleos de cómputo (12 CPUs) con la planificación *inter-workflow*. Se muestra a partir del inicio de la ejecución del sujeto nº 105

El número de tareas de cada tipo que se ejecutan en el intervalo está bastante equilibrado, entre 20 y 25, lo que indica que la ejecución ha alcanzado un estado estable en el que se comienza la ejecución del mismo número de sujetos que se finalizan. Un dato que no se aprecia en la gráfica es que el desfase entre las tareas T1 y las tareas T2* en ejecución es de alrededor de 23 sujetos. Como el resto de tareas consecutivas tienen un desfase de unos 4-5 sujetos, la ventana de sujetos diferentes en ejecución es de aproximadamente 50-55 sujetos. La restricción de poder ejecutar simultáneamente únicamente 4 tareas T2* provoca que al inicio se acumulen las tareas

T2* pendientes de ejecutar y se avance con la ejecución de las tareas T1 de nuevos sujetos, hasta que el sistema se equilibra. A partir de aquí la política de planificación prioriza la ejecución de las etapas que pertenecen a las instancias del *workflow* más antiguas. El hecho de que la ejecución de tareas se equilibre implica que la GPU no está siendo el cuello de botella de la ejecución. Cuando analicemos el escenario con dos procesadores veremos el caso contrario.

Se puede observar un orden secuencial parcial en la ejecución de etapas, de forma que en una misma CPU se suelen ejecutar de forma consecutiva una tarea T2*, una tarea T3, una tarea T4* y una tarea T5 correspondientes al mismo sujeto. Cuando una etapa T2* correspondiente a un cierto sujeto acaba su ejecución en una CPU, el candidato natural más antiguo para ejecutarse es la tarea T3 del mismo sujeto. Y así sucesivamente. También se observa que no hay un patrón definido en la asignación de tareas a las CPUs del mismo núcleo, ya que la mezcla de tareas en cada núcleo de cómputo no se controla.

El plan de ejección *por lotes* para este mismo escenario se muestra en la siguiente tabla (es el mismo que se describió al final del apartado 4.4, en la Figura 4.18). Para diseñar este plan se utilizan los datos de la Tabla 4.12, en la que se estima que las proporciones de tiempo requeridos por cada tarea son, aproximadamente: T1= 50%, T2*+T4*= 25% y T3+T5= 25%. Como se dispone de 12 CPUs, la partición más simple es definir w=6, el número de sujetos del lote, de forma que se asignen la mitad de las CPUs a las 6 tareas T1, y las otras 6 CPUs se repartan entre las tareas que usan GPU y el resto de las tareas que no usan GPU. Así, se define una lista de 6 tareas T1 que se consideran prioritarias, porque son las que tienen un tiempo promedio de ejecución mayor y las que conviene ejecutar lo antes posible; además se promueve que cada tarea T1 sea asignada a una CPU en un núcleo de cómputo diferente, y que la ejecución de tareas T1 se mezcle lo menos posible dentro del mismo núcleo de cómputo. Las otras 6 CPUs del sistema se dividen en dos grupos de 3: el primer grupo se destina a las tareas que usan la GPU y el segundo grupo al resto de tareas.

Plan de Ejecución, w=6, 1 procesador x 6 núcleos x 2 *threads* + 1 GPU	**Tiempo / sujeto**
6 CPUs: Lista Prioritaria {T1, T1, T1, T1, T1, T1} 3 CPUs: Lista No Prioritaria {T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*} 3 CPUs: Lista No Prioritaria {T5, T5, T5, T5, T5, T5, T3, T3, T3, T3, T3, T3}	***1101,5*** seg.

Tabla 16: descripción del plan de ejecución para la planificación *por lotes* utilizando un único procesador. Se muestra el tiempo de ejecución promedio por sujeto obtenido.

Agrupar las tareas que usan GPU en la misma lista consigue reducir mucho la probabilidad de colisiones por culpa de las restricciones de memoria en GPU: solamente en el caso de que se migre un tarea GPU al finalizar el lote se podría provocar que más de 4 tareas intentaran usar la GPU. Además, las tareas T2* y T4* están alternadas en la lista con el objetivo de evitar la ejecución simultánea en la GPU de solamente tareas T2* o de solamente tareas T4*, que son las opciones menos eficientes (ver Tabla 4.8). Finalmente, en la última lista, las tareas T3 se planifican después de todas las tareas T5 porque las tareas T3 tienen un tiempo de ejecución bastante inferior y así son más adecuadas para repartir la carga de trabajo entre CPUs al final de la ejecución del lote.

El tiempo promedio por sujeto de 1101,5 segundos mejora al obtenido por la planificación *inter-workflow* alrededor de un 11%. El tiempo, sin embargo, no alcanza el valor teórico estimado en la tabla 4.12, y es alrededor de un 10% mayor. Esta variación respecto al tiempo teórico puede ser debida a la variabilidad de los tiempos de las tareas en función de los datos de entrada, que modifica la proporción en que unas tareas se mezclan con otras en el mismo núcleo de cómputo y en diferentes núcleos de cómputo. En cualquier caso, la mejora obtenida por la planificación controlada sí que muestra que la mezcla de tareas tiene un efecto positivo significativo en el rendimiento.

El diagrama de Gantt de la figura 4.21 muestra la ejecución de la planificación *por lotes* para un total de 4 lotes completos de w=6 sujetos. Como cada lote supone ejecutar wx5 = 30 tareas, se muestra la ejecución de un total de 30x4=120 tareas, aproximadamente las mismas que en la figura 4.20, y durante la ejecución conviven aproximadamente 6*(5+4) = 54 sujetos de forma simultánea (aunque la ventana de sujetos diferentes en ejecución puede crecer o decrecer un poco según el solapamiento que se produzca entre lotes). Por tanto, el número de sujetos diferentes que en promedio están ejecutando el *workflow* es bastante parecido con los dos esquemas de planificación. Un alto número de sujetos “en vuelo” supone dos implicaciones importantes: (1) el crecimiento de la latencia total para procesar cada sujeto, y (2) el incremento en las necesidades de almacenamiento de todos los datos intermedios en disco. En el segundo escenario que analizaremos, el primer método de planificación no puede garantizar que no se sobrepasa una ventana máxima de sujetos en “vuelo“ y puede potencialmente generar un problema de falta de espacio de almacenamiento temporal.

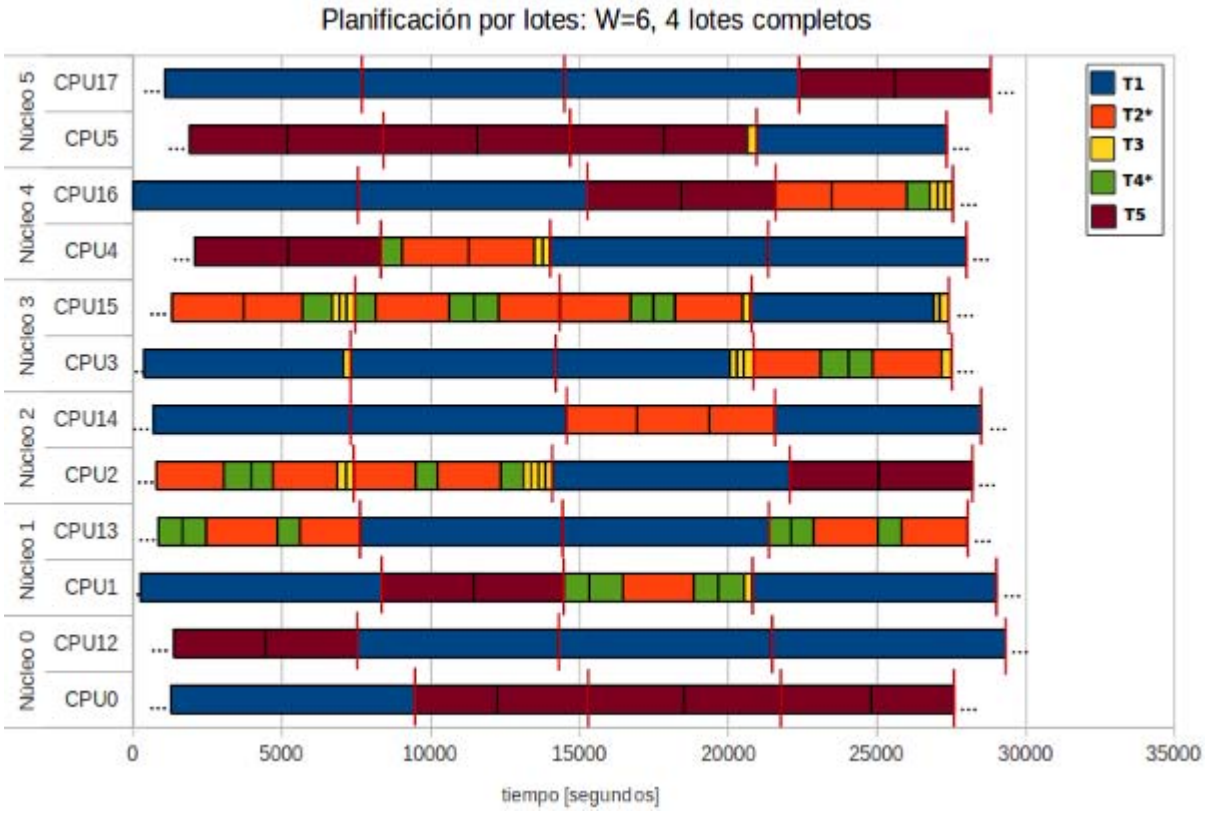


Figura 4.21: diagrama de Gantt para la ejecución de tareas en un procesador con 6 núcleos de cómputo (12 CPUs) con la planificación *por lotes* de w=6 sujetos por lote. El final de cada uno de los 4 lotes ejecutados se indica mediante una línea vertical que actúa como separador.

Las rayas rojas verticales dentro del diagrama de Gantt muestran las fronteras entre lotes y nos permiten ver cómo se solapa la ejecución de un lote con el siguiente. En cada núcleo de cómputo una de las CPUs ejecuta tareas T1 (prioritarias) mientras que la otra CPU normalmente ejecuta tareas no prioritarias, que vimos anteriormente que era la opción preferible. La coincidencia de la ejecución de tareas T1 en el mismo núcleo es poco frecuente, y sólo se produce para compensar la carga de trabajo al iniciar un nuevo lote (como por ejemplo en el núcleo 0, al comenzar el segundo lote, en el que las dos CPUs se intercambian su papel). Por tanto, comprobamos experimentalmente que se promueve la ejecución de mezclas afines dentro de un núcleo de cómputo.

El otro aspecto muy importante que se aprecia en la ejecución es que nunca llegan a solaparse 4 o más tareas que usan la GPU (T2* y T4*), y que con cierta frecuencia solamente se ejecutan dos tareas simultáneas en GPU porque alguna de las CPUs responsables están ejecutando tareas T3. Esto implica que la GPU no se está convirtiendo en un cuello de botella del rendimiento y que es acertada la previsión de asignar 3 CPUs a las tareas que usan GPU, tal y como se deducía del análisis hecho anteriormente.

El mecanismo de migración de tareas entre núcleos se puede apreciar notando cómo las tareas T3 se ejecutan al final del primer lote en CPUs distintas (2, 3 y 15) a las que ejecutan las tareas T5 (CPUs 4, 5 y 12), que están en la misma lista de tareas. Esta migración o compensación consigue equilibrar la carga del lote entre las CPUs y que éstas no finalicen las tareas del lote en momentos muy distanciados. De todos modos, no se puede evitar que los lotes solapen un poco su ejecución, es decir, que mientras tareas del lote i aún no han finalizado, se inicie la ejecución de tareas del lote i+1.

Para finalizar la comparación, en la Tabla 4.14 se muestran los tiempos promedio de cada tipo de tarea para los dos esquemas de planificación. El esquema *por lotes* mejora el tiempo promedio de todas las tareas, excepto de la T5. Es notoria la degradación de un 22% del rendimiento de las tareas T4*. Este hecho se puede asociar a la ocurrencia más frecuente de múltiples tareas T4* ejecutándose a la vez (ver figura 4.20), que vimos empíricamente que no daba buen rendimiento.

Tarea	**T1**	**T2***	**T3**	**T4***	**T5**
inter-workflow	7075 ± 9%	2313 ± 11%	251 ± 15%	1168 ± 32%	2992 ± 12%
por lotes	6785 ± 20%	2211 ± 7%	234 ± 28%	956 ± 35%	2987 ± 20%
Mejora	1,04	1,05	1,07	1,22	1,00

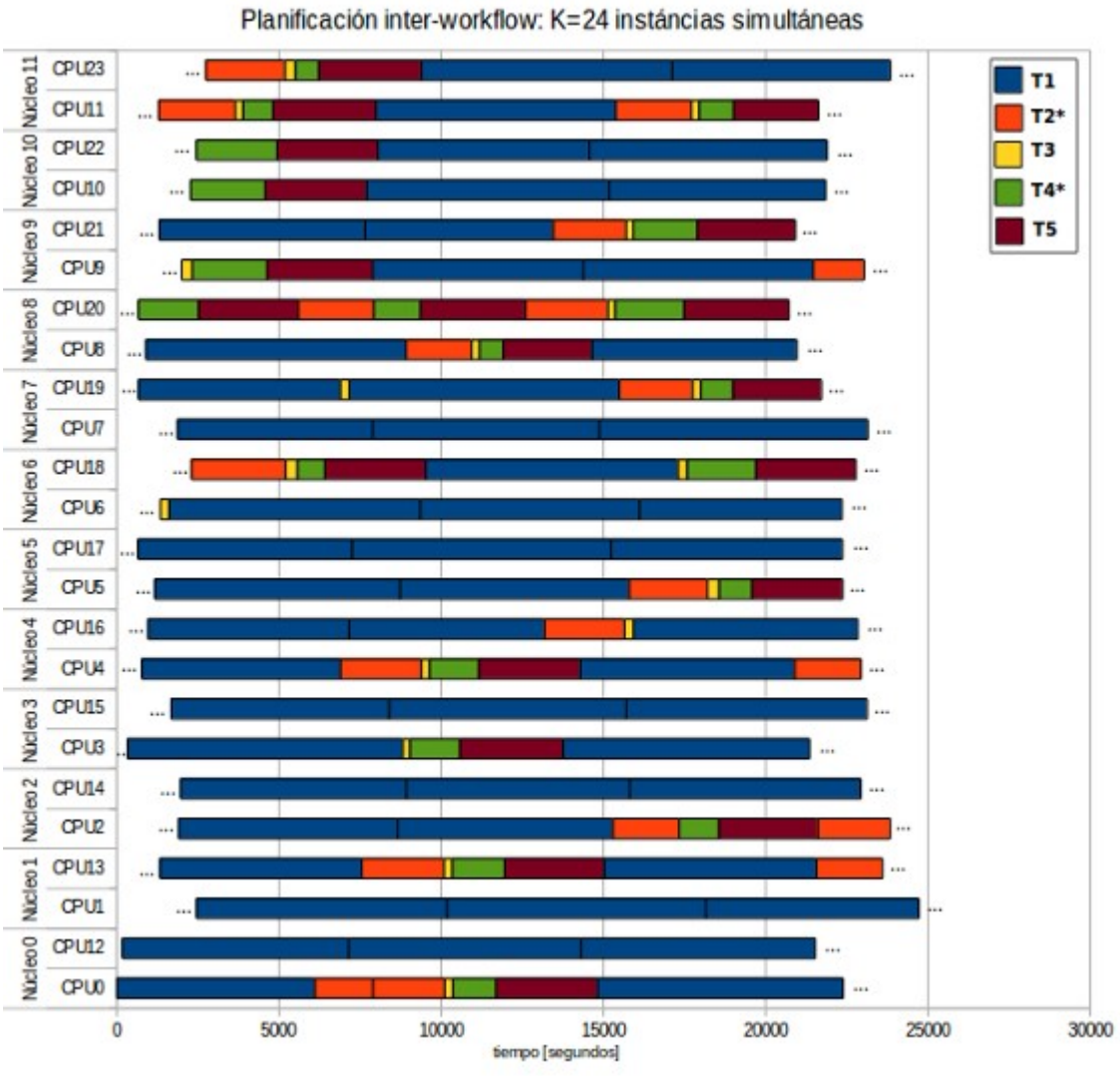
Tabla 17: tiempos promedios de ejecución de cada tarea y desviación estándar para las ejecuciones con planificación *inter-workflow* y *por lotes* en un procesador de 6 núcleos de cómputo y una GPU

4.6.3 Resultados Detallados usando dos procesadores y una GPU

A continuación se repite el análisis para el escenario 2, utilizando 2 procesadores y una GPU. Recordemos que en este caso la planificación *inter-workflow* y la planificación *por lotes*, respectivamente, presentan un *speedup* de 8,27x y de 10,48x respecto a la ejecución serie. Por tanto, la planificación *por lotes* mejora en un 27% el rendimiento de la planificación *inter-workflow*.

La figura 4.22 muestra la traza de Gantt para la ejecución utilizando la planificación *inter-workflow* con K=24 instancias simultáneas y con la misma restricción de tareas simultáneas usando la GPU que para el anterior escenario. La traza comienza en la ejecución de la tarea T1 correspondiente al sujeto nº 149. Lo más llamativo es que se está ejecutando un mayor número de tareas T1 (52) que del resto de tareas (alrededor de 20). Esto es un síntoma de que la GPU es un cuello de botella, y que frente a la imposibilidad de ejecutar las tareas T2* a suficiente ritmo, el sistema

ejecuta tareas T1 de nuevos sujetos. Para este *workflow* concreto, duplicar la capacidad de cómputo en CPU y mantener la misma capacidad de cómputo en GPU produce un sistema desequilibrado.



Fi

gura 4.22: diagrama de Gantt para la ejecución de tareas en dos procesadores con la planificación *inter-workflow*. Se muestra a partir del inicio de la ejecución del sujeto nº 149

El tiempo promedio de la ejecución es de 810 segundos por sujeto, pero se identifican dos fases de ejecución con un comportamiento muy diferente. En la primera fase, de aproximadamente la mitad del tiempo total de ejecución, se ejecutan etapas de todos los tipos, pero con una mayor proporción de etapas del tipo T1 (como en la figura anterior): las restricciones de uso de la GPU impiden avanzar la ejecución de tareas antiguas, y esto provoca la acumulación de tareas T1 correspondientes a nuevos sujetos. La segunda fase comienza cuando se han ejecutado las tareas T1 de los 200 sujetos, y solamente se han completado 66 sujetos completos (por tanto, se

ejecutan solamente etapas T2*, T3, T4* y T5). Es decir, que el desfase llega a un valor máximo de 134 sujetos “en vuelo”. En esta segunda fase las tareas no ocupan todos los recursos, de forma que la productividad es menor, aunque las latencias individuales de cada tarea mejoran. Aún teniendo en cuenta las ineficiencias de la segunda fase, el rendimiento total obtenido al usar dos procesadores es un 51% mejor que cuando se utiliza un solo procesador.

Hay que destacar que la primera fase de la ejecución, bastante más eficiente que la segunda fase, no es sostenible en un sistema con un workload de miles de sujetos de entrada. En algún punto de la ejecución el tamaño de almacenamiento en disco no sería suficiente para mantener los datos intermedios de todas las instancias de *workflow* que están a medias en su procesamiento. Por tanto, se debería controlar el máximo número de sujetos que están “en vuelo”. Este control es el que proporciona la planificación *por lotes*, y que analizamos a continuación.

Dado que el análisis previo del *workflow* que estamos considerando nos indica que al usar dos procesadores en lugar de uno la GPU se convertirá en cuello de botella, se introducen dos variaciones muy significativas en la planificación *por lotes* respecto a la ejecución con un procesador: (1) las listas exclusivas de tareas y (2) la sustitución de algunas tareas que usan GPU por la versión de las tareas que no usa GPU.

El uso de una lista exclusiva supone reservar antes de comenzar la ejecución un cierto número de núcleos de cómputo completos, que se mantendrán asignados durante toda la ejecución. A cada núcleo de cómputo se le va asignando de forma dinámica una tarea de esta lista, de forma que solamente habrá de forma simultánea una única tarea en el núcleo de cómputo, y no se utilizará la capacidad de ejecución *multi-thread*. El objetivo genérico de esta lista es minimizar la latencia de ciertas tareas impidiendo que otra tarea compita por los mismos recursos de cómputo dentro del mismo núcleo. El sentido que tiene usarla en esta aplicación es promover el uso eficiente de la GPU: puesto que la memoria de la GPU limita la concurrencia en el uso de la GPU sin llegar, ni mucho menos, a saturar su capacidad de cómputo, lo mejor es que las tareas que usan GPU tengan la menor latencia posible. De este modo, el uso más eficiente de la GPU se consigue a cambio de un uso menos eficiente de la CPU.

Se considera que no tiene sentido migrar tareas de una lista exclusiva a las CPUs que no tiene asignadas, ni migrar tareas de listas no exclusivas a los núcleos reservados para las listas exclusivas. Por tanto, la ejecución se divide en dos partes: los lotes CPU para tareas que solamente usan CPU y los lotes GPU para tareas que usan GPU, y que se asocian a la lista exclusiva. La ejecución de estos lotes es desacoplada, pero se controla que no se solape durante dos lotes completos; es decir, no se puede comenzar la ejecución del lote i+2 de sujetos para un

grupo de tareas hasta que el otro grupo de tareas no haya ejecutado completamente las que corresponden a los sujetos del lote i.

La Tabla 4.15 muestra el plan de ejecución utilizado para obtener los resultados de la Figura 4.19. La lista exclusiva contiene las tareas que usan GPU, alternadas para favorecer la mezcla de tareas diferentes. Las otras dos listas contienen tareas que usan únicamente la CPU, divididas entre las tareas T1, prioritarias por ser las que tienen mayor latencia, y el resto de tareas que tienen buena afinidad con las tareas T1. Justificaremos algo más adelante la decisión de usar 4 núcleos de forma exclusiva para GPU. Como quedan 16 CPUs para dedicar a las listas no exclusivas, se decide usar un valor w=8 y colocar 8 tareas T1 en la lista prioritaria y el resto de tareas en la lista no prioritaria. La proporción de carga en cada lista no está equilibrada (alrededor del 65% en la lista prioritaria y del 35% en la no prioritaria) pero se deja que sea el mecanismo dinámico de robo entre listas el que equilibre la ejecución.

Plan de Ejecución, w=8, 2 procesadores x 6 núcleos x 2 *threads* + 1 GPU	**Tiempo/Suj.**
4 núcleos: Lista exclusiva {T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*} 8 CPUs: Lista prioritaria {T1, T1, T1, T1, T1, T1, T1, T1} 8 CPUs: Lista no prioritaria {T5,T5,T5,T5,T5,T5,T5,T5,T4,T3,T3,T3,T3,T3,T3,T3,T3}	**639 (CPU)** (GPU: 620)

Tabla 18: descripción del plan de ejecución para la planificación *por lotes* utilizando dos procesadores y una GPU. Se "mueve" una tarea T4* a tarea T4. Se muestra el tiempo de ejecución promedio por sujeto (limitado por tareas CPU) y también el tiempo de las tareas que no son cuello de botella (GPU).

La primera ejecución se realizó con el plan de ejecución de la Tabla 4.16 y permitió constatar que la ejecución de las "tareas GPU" se quedaba atrás y se convertía en el cuello de botella del rendimiento: 650 seg./sujeto para las tareas GPU y 620 seg./sujeto para tareas CPU. Así que en la definición del plan de ejecución final que se muestra en la Tabla 4.15 sustituimos una tarea T4* que se debería ejecutar usando la GPU, como la versión equivalente T4 que se ejecuta únicamente en CPU. Esta sustitución se realiza de forma estática, y sirve para equilibrar la carga que tienen los lotes CPU y los lotes GPU. Solamente consideramos la "migración" de las tareas T4* porque que la versión CPU de las tareas T2* tiene un tiempo de ejecución demasiado elevado, y supondría en cualquier caso un mayor desequilibrio en la carga. La proporción de migrar una tarea T4* de cada 8 a la forma de tarea T4 se ha obtenido haciendo un cálculo a partir de los resultados del apartado 4.4 y luego verificando de forma experimental que es mejor opción que la proporción 2 a 8 o 1 a 16.

Plan de Ejecución, w=8, 2 procesadores x 6 núcleos x 2 *threads* + 1 GPU	**Tiempo/Suj.**
4 núcleos: Lista excl. {T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*} 8 CPUs: Lista prioritaria {T1, T1, T1, T1, T1, T1, T1, T1} 8 CPUs: Lista no prioritaria {T5,T5,T5,T5,T5,T5,T5,T5,T3,T3,T3,T3,T3,T3,T3,T3}	***650 (GPU)*** (CPU: 615)

Tabla 19: descripción de un plan de ejecución para la planificación *por lotes* utilizando dos procesadores y una GPU. No se incluye el "movimiento" de tareas T4* a tareas T4. Se muestra el tiempo de ejecución promedio por sujeto, que corresponde a las tareas GPU, y también el tiempo de las tareas CPU.

El resultado del plan de ejecución final es que ahora las tareas de CPU tienen un tiempo promedio de 639 seg./sujeto, mientras que las tareas de GPU tienen un tiempo de 620 seg./ sujeto; es decir, el rendimiento final es de un promedio de 639 seg./sujeto y está limitado por la capacidad de cómputo de la CPU.

Para decidir la proporción de recursos asignados a las listas se han utilizado los datos de la Tabla 4.11 y 4.12. La Tabla 4.11 nos indica que el uso de 4 núcleos de cómputo para tareas GPU nos puede ofrecer un rendimiento inferior a 700 seg./sujeto, y cercano al mejor posible (*best-GPU-effort*) de unos 600 seg./sujeto. En realidad, los resultados que se muestran en la Tabla 4.16 muestran que incluso sin ahorrarnos la ejecución de algunas tareas GPU, moviéndolas a tareas CPU, se puede obtener un tiempo medio de 650 seg./sujeto para las tareas GPU. La explicación de esta mejora para las tareas GPU respecto a la predicción de la Tabla 4.12 es que ahora se están usando dos núcleos de cada procesador, mientras que en los resultados anteriores se usaban los 4 núcleos del mismo procesador. Usar 5 núcleos para tareas GPU no es una buena opción, porque a partir de los datos de la Tabla 4.12 (y del resultado empírico de la Tabla 4.16, de 615 seg./sujeto para tareas CPU usando 16 CPUs) podemos calcular que el tiempo para las tareas CPU usando las 14 CPUs restantes sería algo superior a los 700 seg./sujeto, por encima del tiempo esperado para las tareas GPU.

La figura 4.23 muestra la traza de Gantt de la ejecución planificada *por lotes* para un total de 4 lotes completos. De los cuatro núcleos dedicados a las tareas GPU, se asignan dos núcleos del procesador 0 y dos del procesador 1 para que el uso de ambos procesadores sea simétrico y disponer de los mismos recursos libres en cada procesador. Se observa cómo las ejecuciones del lote CPU y del lote GPU avanzan de forma independiente, hasta que en el núcleo 6, en la transición entre el lote 3 y lote 4, se produce un hueco en la ejecución debido a la espera de las

tareas GPU que se están adelantando a las tareas CPU. Es necesario asegurar que el lote 4 no empiece antes de que finalice el lote 2 (la tarea de la CPU 20 no ha finalizado) y asegurar las dependencias entre tareas. De nuevo, el hecho de que la ejecución de tareas GPU tenga que frenarse indica que la ejecución está limitada por CPU.

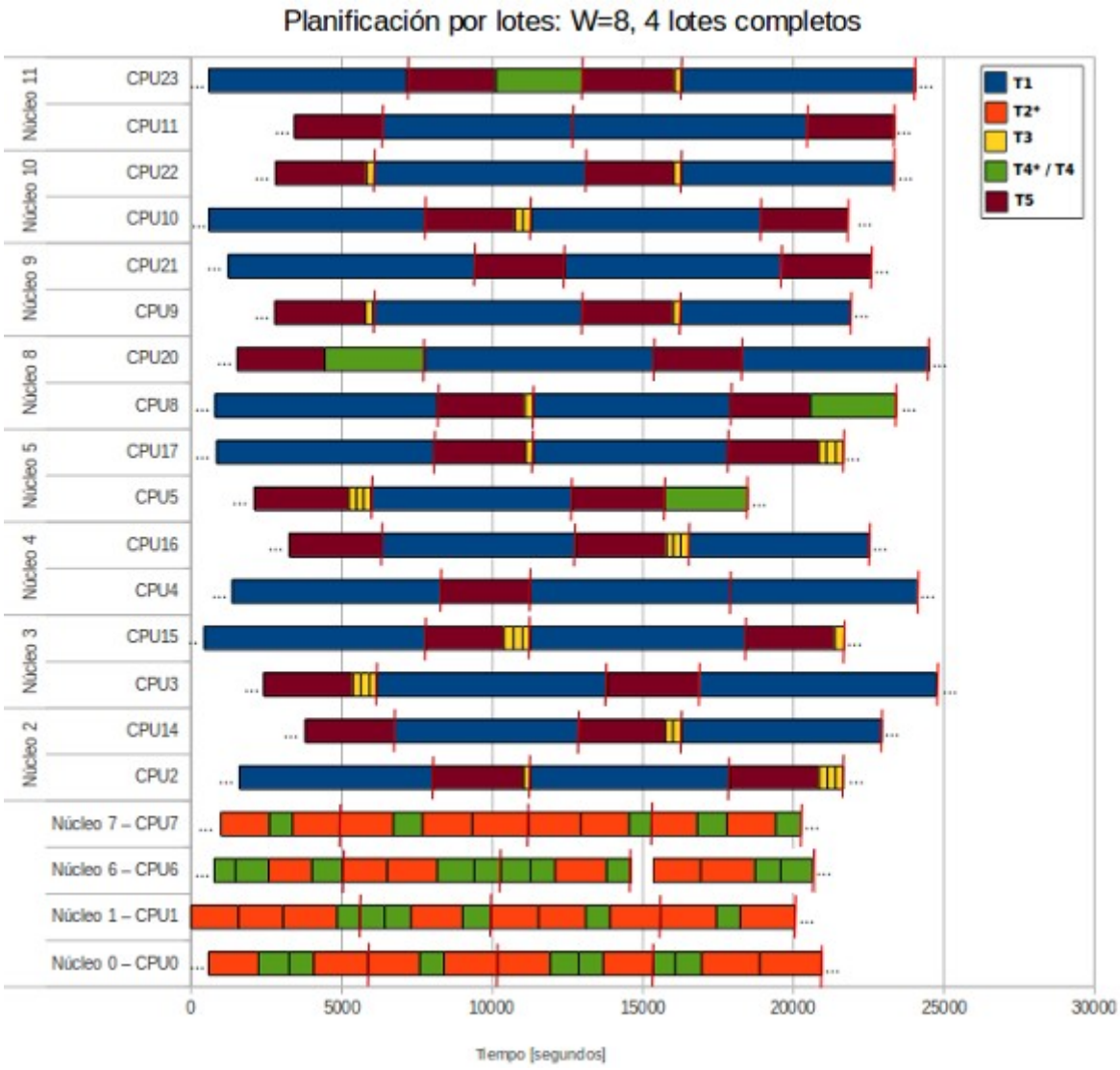


Figura 4.23: diagrama de Gantt para la ejecución de tareas en dos procesadores con la planificación *por lotes* de cuatro lotes de w=8 x 5 tareas.Se indica el final de cada lote mediante una línea vertical que actúa como separador.

En la figura anterior también se puede apreciar que la proporción del tiempo en que se solapa la ejecución de dos tareas T1 en el mismo núcleo de cómputo es ahora mayor, y corrobora el hecho de que la carga de trabajo de tareas T1 es mayor que la de tareas T3, T4 y T5.

La Tabla 4.17 compara las latencias de cada tipo de tarea en los escenarios de ejecución con un y dos procesadores (y una GPU). El uso de una lista exclusiva para las tareas GPU logra reducir la

latencia promedio de las tareas GPU de forma considerable. En cambio, al aumentar la proporción de tiempo en que las tareas T1 se ejecutan de forma solapada con otras tareas T1 estamos perjudicando ligeramente la latencia de estas tareas. El pequeño peso de las tareas T3 hace que la degradación no sea importante.

Tarea	**T1**	**T2***	**T3**	**T4***	**T5**
1 procesador	6785	2211	234	956	2987
2 procesad.	6908	1671	257	601	2942
Mejora	0,98	1,32	0,91	1,59	1.02

Tabla 20: tiempos promedios de ejecución de cada tarea para las ejecuciones con planificación *por lotes* en uno y dos procesadores (y una GPU).

4.6.4 Uso de dos GPUs y Conclusiones

En el apartado anterior hemos constatado que la GPU es cuello de botella si no se sustituyen tareas de GPU a CPU, pero por un pequeño margen de 650 seg./sujeto en GPU y 615 seg./sujeto en CPU. Al sustituir tareas GPU como tareas CPU modificamos la proporción y se logra un equilibrio de 620 seg./sujeto en GPU y 639 seg./sujeto en CPU. Esto implica que no podemos esperar demasiada mejora del uso de una segunda GPU. Para corroborar este análisis se ha realizado un experimento utilizando las dos GPUs idénticas instaladas en el sistema de cómputo. Se usa el mismo plan de ejecución que antes, con cuatro núcleos asignados de forma exclusiva a tareas GPU y sin mover tareas T4* a T4, y con w=8. El resultado se muestra en la Tabla 4.18.

Plan de Ejecución, w=8, 2 procesadores x 6 núcleos x 2 *threads* + 2 GPUs	**Tiempo/Suj.**
4 núcleos: Lista excl. {T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*,T2*,T4*} 8 CPUs: Lista prioritaria {T1, T1, T1, T1, T1, T1, T1, T1} 8 CPUs: Lista no prioritaria {T5,T5,T5,T5,T5,T5,T5,T5,T3,T3,T3,T3,T3,T3,T3,T3}	***615 (CPU)*** (GPU: 575)

Tabla 21: descripción de un plan de ejecución para la planificación *por lotes* utilizando dos procesadores y dos GPUs. Se muestra el tiempo de ejecución promedio por sujeto, que corresponde a las tareas CPU, y también el tiempo de las tareas GPU.

El tiempo por sujeto del lote CPU es 615 segundos, el mismo tiempo que cuando se usa una GPU (ver Tabla 4.16), mientras que el tiempo del lote GPU baja de 650 seg./sujeto a 575 seg./sujeto. El lote GPU se acelera un 13% con el uso de dos GPUs, pero ahora la ejecución pasa a estar limitada por las tareas CPU. En el escenario anterior, con el rendimiento limitado por la única GPU del sistema, la “migración” de algunas tareas GPU a tareas CPU permitía equilibrar un poco la carga y mejorar el rendimiento global de 650 a 639 (casi un 2%). En el nuevo escenario es la CPU el recurso limitante, y para acelerar la parte de CPU deberíamos “quitar” una de las CPUs que están usando las tareas GPU, y esto provocaría que la ejecución de tareas GPU se degradara considerablemente y que el rendimiento global sea peor.

El problema general es que el uso de la GPU no es ortogonal al uso de la CPU, y se pueden producir interacciones complejas en el rendimiento. Como las tareas que usan la GPU también necesitan la CPU para ejecutar parte del código de la aplicación y para ejecutar el *driver* de la GPU, un mayor número de GPUs puede no ofrecer una ganancia importante si a la vez no se aumenta el número de CPUs del sistema.

Como conclusión de este capítulo podemos enumerar las siguientes reflexiones. En primer lugar, la ejecución multi-*workflow*, donde la planificación la realiza el sistema operativo, se ve limitada por no poder lanzar más de 4 instancias en paralelo para evitar la posibilidad de que alguna tarea que usa la GPU aborte por falta de memoria en GPU.

La ejecución *inter-workflow*, en comparación con la planificación *por lotes*, sufre por la migración de procesos entre CPUs que realiza el sistema operativo, y por las mezclas de tareas que se ejecutan de manera incontrolada y que no siempre aprovechan la capacidad de *hyperthreading*. Pero la gran ventaja de la planificación *por lotes* es que ofrece una forma de controlar la ejecución que permite equilibrar la carga entre tareas CPU y GPU.

Actualmente el diseño del plan de ejecución lo hemos hecho de forma “manual”, a partir del análisis de ejecuciones individuales y conjuntas de las tareas que constituyen el *workflow*. El plan de ejecución se puede acabar de sintonizar con unas pocas ejecuciones diferentes. Los puntos clave son decidir el número de CPUs asignadas a la lista exclusiva destinada a las tareas que usan la GPU (o si se decide no usar ninguna lista exclusiva). En el caso de que sea conveniente mover tareas GPU a tareas exclusivamente CPU también puede ser necesario experimentar con la proporción adecuada.

5 Conclusiones

Las arquitecturas de los computadores del presente y del futuro están evolucionando a un diseño híbrido: los sistemas de cómputo ya no estarán formados únicamente por uno o varios procesadores a los que se les multiplica el número de núcleos CPU, sino que también incluirán aceleradores que se usarán como co-procesadores por parte de las aplicaciones para aumentar la velocidad de cómputo. Ese futuro, que lo podemos imaginar como sistemas integrados en una misma placa base o chip, es ya una realidad en algunos procesadores de los principales fabricantes aunque aún no están del todo extendidos y habitualmente los dispositivos que actuan como co-procesador, en nuestro caso las GPUs, se conectan vía el BUS PCI-express.

5.1 Resumen y Conclusiones

En este trabajo hemos tratado como caso de ejemplo una aplicación que destina parte de su cómputo a la GPU. Concretamente, hemos analizado el *workflow* científico de procesamiento de resonancias magnéticas estructurales contenido en FreeSurfer. En un primer análisis y experimentación, que se ha recogido en forma de estudio en el capítulo 3 de esta memoria, hemos observado cómo la gestión de los recursos en estos entornos híbridos no es trivial. Esta complicación aumenta con el uso compartido de la GPU entre diferentes aplicaciones, o diferentes instancias de una misma aplicación.

Para las aplicaciones que no sean tolerantes a fallos, no se garantiza una correcta ejecución, al permitir el acceso concurrente a la GPU. En nuestro caso de uso, cuando ejecutamos múltiples instancias de una etapa de FreeSurfer que usa la GPU, se saturaba la memoria de la tarjeta y la aplicación abortaba de forma no controlada.

Después de un análisis de dependencias entre las diferentes etapas del *workflow*, propusimos un esquema de ejecución basado en el uso del paralelismo intra- e *inter-workflow,* con la gestión planificada de acceso a los recursos del sistema. Esto nos permitió en un primer momento mejorar la explotación de los recursos disponibles alcanzando una mejora de la productividad (*throughput*) de más de 6x comparado con la implementación base serial y de más de 2.5x comparada con la implementación base con el soporte de la aceleración de etapas en GPU.

La mejora que podamos obtener depende en una parte muy importante de la plataforma *hardware* en la que se ejecuta la aplicación. La configuración usada en el primer estudio es una plataforma

económica basada en una estación de trabajo de sobremesa muy asequible para cualquier grupo de investigación. En cambio, en el segundo estudio contamos con un nodo de cómputo similar en cuanto a prestaciones a un nodo de cómputo de un sistema cluster.

En la ejecución *multi-workflow*, donde la planificación de la ejecución la realiza el sistema operativo, hemos detectado que para la etapa 2 del *workflow* de FreeSurfer hay una limitación de sólo poder ejecutar dos instancias simultáneas en la configuración *hardware* del primer estudio y de cuatro instancias simultáneas en la configuración *hardware* empleada en el segundo estudio de este trabajo. Esta limitación afecta a todas las ejecuciones, sean o no planificadas, ya que hace referencia a un límite físico del dispositivo.

La limitación en la gestión de la memoria aparecía en la literatura como una posibilidad con aplicaciones que manejan una cantidad superior de datos a los que las memorias de las distintas GPUs empleadas pueden almacenar. En nuestro caso el problema sucede con la ejecución de múltiples instancias de la misma aplicación. En cualquier caso, este problema se convierte en un problema clásico de planificación y gestión de recursos que en este trabajo se ha solucionado mediante la implementación de políticas de planificación.

En la primera aproximación de esta tesis, se ha implementado un planificador que gestiona las GPUs mediante una tabla de restricciones. El planificador se diseña con un propósito general, para poder ser usado en otros *workflows* similares. Una de las principales ventajas de que la planificación se realice a nivel de etapas es que se pueden tratar las implementaciones como cajas negras sin entrar a modificar el código de la aplicación. Para la adaptación a otros *workflows* debemos considerar la organización de sus etapas, y las dependencias entre los datos de entrada y salida de cada etapa. En el caso de detectar alguna limitación en el uso del sistema de cómputo, se deberá explicitar en la tabla de restricciones.

La propuesta de planificación *inter-workflow* nos aporta una mejora de 2.7 veces entre la ejecución CPU y la ejecución CPU+GPU cuando utilizamos el paralelismo inter- e *intra-workflow*. El mayor incremento de la productividad se obtiene ejecutando, bajo el sistema de cómputo definido, entre seis y ocho instancias simultáneas. Los experimentos han mostrado que usar más de ocho en el mismo sistema de cómputo no aporta ningún beneficio.

Al finalizar el análisis y la verificación de las propuestas de la planificación *inter-workflow*, nos dimos cuenta que aún existía un margen de mejora en lo que al rendimiento de la ejecución de *workflows* científicos se refiere. El planificador propuesto inicialmente permitía gestionar el acceso a los recursos de forma segura, pero no tenía en cuenta el rendimiento que se estaba obteniendo, por lo que la planificación estaba lejos de ser eficiente durante gran parte de su ejecución. Es por

eso que decidimos mejorar el sistema de planificación teniendo en cuenta una mejor utilización de los recursos tanto CPU como GPU.

En el apartado 4.1 de este trabajo propusimos un nuevo esquema de planificación, más flexible, para la ejecución de las tareas correspondientes a múltiples instancias de un *workflow* oganizadas en un conjunto de listas que definen un lote de procesamiento. Esta propuesta se ha basado en un análisis empírico inicial que ha servido para extraer las principales características de cada etapa y el efecto de la ejecución combinada de las tareas de diferente tipo.

En el caso de estudio que se presenta en la segunda aproximación de esta tesis, se ha analizado el *subflow* de la aplicación FreeSurfer que realiza el procesamiento y análisis volumétrico de una resonancia magnética de entrada. Este sub-*workflow* consta de 5 etapas.

Hemos definido un escenario en dos fases, una fase inicial transitoria y una fase estacionaria. En lfase transitoria se ejecutan etapas de diferentes instancias del *workflow* con diferentes datos de entrada. En la fase estacionaria hemos centrado el estudio y la propuesta de planificación, a partir de un conjunto de etapas, cada una con datos independientes y que hemos llamado *lote*. Cada lote define un buen plan de ejecución que organiza la ejecución en listas de tareas mediante las cuales se generan combinaciones favorables de tareas.

La generación de un plan de ejecución de las tareas que promueva las combinaciones favorables entre tareas se ha realizado con la definición de un conjunto de reglas. El objetivo es hacer un uso eficiente tanto de las CPUs como de las GPUs del sistema, asegurando una mejor afinidad entre tareas que comparten el mismo núcleo de cómputo y un buen balance entre la carga asignada a las CPUs y a las GPUs. El mecanismo que gestiona estas listas se basa en organizar las tareas en listas que pueden ser de tres tipos: prioritarias, no-prioritarias y exclusivas. Esta tipología de listas viene determinada por el tipo de tareas que se les asignan. En el caso de las prioritarias se asignan las tareas de mayor tiempo de ejecución. En el caso de las no-prioritarias se pueden asignar tareas que sean compatibles con las tareas prioritarias. Y finalmente en las listas exclusivas se asignan las tareas cuya latencia de ejecución queremos reducir, ya que se ejecutan sin utilizar el *hyperthreading*. En nuestro escenario se han usado con tareas que usan la GPU.

La combinación de las diferentes listas describe cómo las tareas se asignan posteriormente a los núcleos de cómputo y a las CPUs para usar los recursos de forma eficiente. Aunque pueda parecer que el no usar el *hyperthreading* en los núcleos de cómputo asignados a las listas exclusivas suponga una pérdida de potencial de cómputo, los datos de los experimentos y el análisis de afinidades han demostrado que para obtener un mayor *throughput* de la aplicación, es mejor no mezclar la ejecución de las tareas GPU con otras tareas. Éste es un punto clave del sistema de planificación *por lotes* propuesto.

Para llevar a cabo la planificación de múltiples tareas y poder afrontar así ejecuciones con grandes cantidades de datos, hemos definido un sistema de ejecución de los lotes de manera entrelazada entre los lotes pares e impares (*pipeline interleaving*). Esta ejecución evita la necesidad de verificar las dependencias de datos entre lotes consecutivos, a la vez que aumenta la latencia de ejecución para una instancia. Este aumento está controlado y depende de la organización de las tareas dentro de los lotes y del número total de tareas que tenga el *workflow*. En el caso de estudio, la latencia de un sujeto queda fijada en 9 lotes, debido a que el *wokflow* está compuesto por 5 etapas.

Sobre el análisis realizado en el capítulo 4 cabe destacar que el comportamiento de las tareas que usan solamente la CPU es bastante homogéneo, con un rendimiento que escala bien con el uso del *hyperthreading* y de múltiples núcleos de cómputo y procesadores; en cambio, el comportamiento del rendimiento de la GPU es mucho más complejo y menos intuitivo. Las etapas implementadas en GPU presentan un mal rendimiento y escalabilidad debido a, por ejemplo, a restricciones de memoria en el caso de la etapa T2* y a restricciones derivadas de la gestión de las llamadas a la API de CUDA en el caso de T4*. Cabe destacar que la ejecución de etapas en GPU implica tener asignada también una CPU. Por el análisis realizado hemos observado cómo las CPUs que se asgnan a tareas GPU están infrautilizadas y, por tanto, representan recursos que no se están aprovechando para ejecutar otras tareas CPU, por ejemplo.

La etapa T4* del *workflow* ofrece un mal rendimiento debido, probablemente, a una mala implementación. Observamos que ejecutar demasiadas tareas T4* de forma concurrente resulta ser una mala opción, aunque quepan en la memoria de la GPU, puesto que su uso intensivo de las llamadas a la API de CUDA provoca este mal rendimiento. Una recomendación que se puede derivar de este trabajo es que la etapa T4 deberia ser revisada y re-implementada.
Por otra parte, resulta ser una buena idea migrar tareas T4* a CPU (T4) como mecanismo de compensación entre las cargas soportadas por los recursos CPU y GPU. Afinar en la correcta proporción, según la evolución de la planificación en lotes sucesivos, es un punto clave de la propuesta. En este trabajo la proporción en la migración (1 de cada 8 tareas T4*) está establecida «manualmente» en el plan de ejecución, pero en un futuro, introducir un mecanismo dinámico que gestione las migraciones aportaría mejoras en el rendimiento global de la ejecución planificada.

A modo de conclusiones finales podemos decir, en primer lugar, que las propuestas presentadas en este trabajo aportan una mejora del rendimiento del *workflow* de estudio en todos los casos. Pasamos de una ejecución *multi-workflow,* en la que la planificación es gestionada por el sistema operativo y en la que aparecen problemas de gestión de la GPU, a la ejecución bajo la planificación *inter-workflow* que los soluciona y permite mejorar el rendimiento. No obstante, cuando escalamos el número de instancias a ejecutar simultáneamente, la ejecución *inter-*

workflow, en comparación con la planificación *por lotes*, sufre por la migración de procesos entre CPUs que realiza el sistema operativo, y por las mezclas de tareas que se ejecutan de manera incontrolada y que no siempre aprovechan la capacidad de *hyperthreading*. La gran ventaja de la planificación *por lotes* es que ofrece una forma de controlar la ejecución que permite balancear la carga entre tareas CPU y GPU.

Hemos observado también cómo la introducción de más GPUs en el mecanismo de planificación no siempre comporta una gran mejora en el rendimiento. Esto se debe en parte a la reducción del número de CPUs disponibles para ejecutar todo el volumen de trabajo CPU. Por otra parte también tiene consecuencias en el rendimiento de ejecutar múltiples etapas GPU simultáneamente, sobre todo si tenemos en cuenta que la etapa T2* computa también en la CPU. Finalmente, debemos mencionar que el rendimiento al usar dos o más GPUs también viene condicionado por la gestión que hace el *driver* para atender a las transferencias de datos entre la GPU y la máquina huésped, así como también de las llamadas a la API de CUDA. Este también es un punto clave en el que se debe mantener el equilibrio entre el rendimiento obtenido por el trabajo CPU y el trabajo GPU.

5.2 Publicaciones realizadas

A lo largo de la realización de este trabajo de tesis se han realizado diferentes publicaciones listadas a continuación:

El primer artículo de este trabajo, relativo a las propuestas presentadas en el capítulo 3 de esta tesis, ha sido publicado por la revista NeuroInformatics, que en el momento de la publicación tenía un factor de impacto del 2,825 y teniendo una posición 13 de 102 (Q1) en la categoría de Computer Science según Web of Knowledge y en posición 124 de 252 (Q2) en la categoría de Biociencias.

- J. Delgado, J.C. Moure, Y. Vives-Gilabert, M.Delfino, A.Espinosa, B.Gómez-Ansón, Improving the Execution Performance of FreeSurfer: A new scheduled *pipeline* scheme for optimizing the use of CPU and GPU resources. Neuroinformatics Journal, Volume 12, Issue 3, July 2014, Pages 413-421. Ed. Springer Editors-in-Chief: G.A. Ascoli; E. de Schutter; D.N. Kennedy ISSN: 1539-2791 (print version) ISSN: 1559-0089 (electronic version)

La publicación anteriormente listada ha sido citada por Craddock, R Cameron et al.: Connectomics and new approaches for analyzing human brain functional connectivity. GigaScience. Volume: 4. Issue: 1. 2015.

En la figura 5.1 podemos observar el impacto de este primer artículo en la comunidad, podemos decir que ha tenido un impacto inmediato. El repositorio en el que se aloja el código de manera abierta, ha recibido desde su publicación en octubre de 2013 hasta la fecha actual, 158 descargas y numerosas consultas de soporte via la *mailing list* general de FreeSurfer. El código se puede descargar en [48].

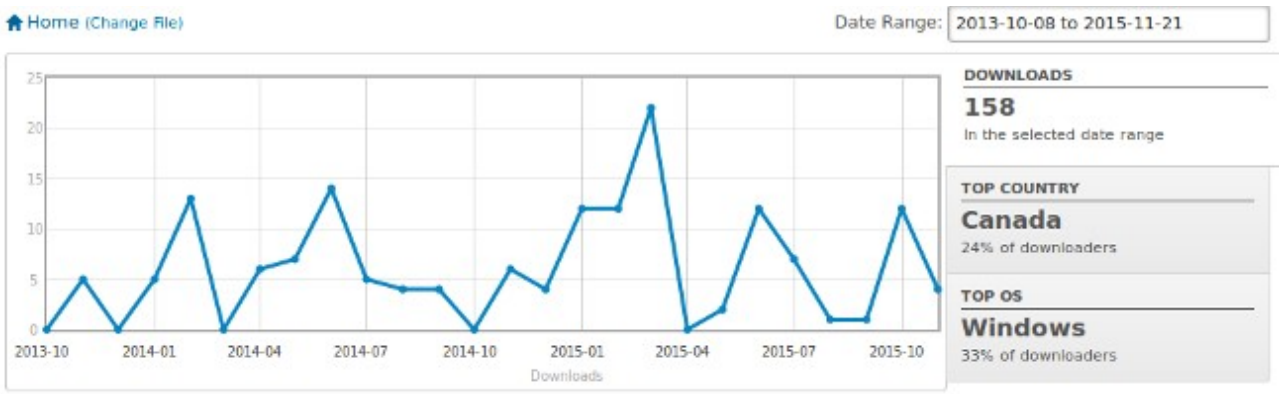


Figura 5.1: Evolución de las descargas del repositorio público que contiene el código del planificador *inter-workflow*. Fuente, repositorio *sourceforge.*

Realizamos también un trabajo conjunto con otro compañero doctorando del departamento en el que portamos el esquema de ejecución de FreeSurfer propuesto en el capítulo 3 de este trabajo al entorno Galaxy [11], una herramienta de diseño y gestión de *workflows* científicos. Este trabajo describe una plataforma científica basada en Galaxy para lograr ejecutar simultáneamente múltiples análisis de altas prestaciones. Se propone una extensión de Galaxy capaz de realizar ejecuciones paralelas y distribuidas usando DRMAA (Distributed Resource Manager Application API) [45] como interfaz a SLURM (Simple Linux Utility for Resource Management) [46] como sistema de scheduling y manejo de recursos computacionales. Lo anterior permite por tanto, la implementación de *workflows* científicos usando patrones paralelos.

- César Acevedo, Porfidio Hernández, Antonio Espinosa, Gonzalo Vera, Javier Navarro, Jordi Delgado, Yolanda Vives y Manuel Delfino. Plataforma de ejecución paralela de *Workflows* científicos en Galaxy. XXV Jornadas Sarteco de Paralelismo. 17-19 de septiembre de 2014, Valladolid.

También se ha empleado el conocimiento obtenido en la primera parte del trabajo, con el nuevo esquema de ejecución del *workflow* sobre la herramienta FreeSurfer, para colaborar, en forma de transferencia de conocimiento, en un análisis clínico realizado conjuntamente con el Hospital de la Vall d'Hebrón (Barcelona) y el “Institut de Diagnòsi per la Imatge” IDI. El trabajo ha sido presentado por los autores principales en dos ámbitos:

- Pareto Onghena, D.; Sastre Garriga, J.; Vives Gilabert, Y.; Delgado, J; Huerga, E.; Tintoré Subirana, M.; Montalban Gairin, X.; Auger, C.; Rovira Cañellas, Á. Valoración del papel de

las lesiones yuxtacorticales en las medidas de grosor cortical. LXV Reunión Anual de la Sociedad Española de Neurología (SEN), Barcelona, 19-23 Noviembre 2013

Y una posterior ampliación del trabajo en:

- Pareto Onghena, D.; Sastre Garriga, J.; Vives Gilabert, Y.; Delgado, J; Huerga, E.; Tintoré Subirana, M.; Montalban Gairin, X.; Auger, C.; Rovira Cañellas, Á. Juxtacortical lesions and cortical thinning in multiple sclerosis. American Journal of Neuroradiology, impact factor 3.589 (Aceptado, en segunda ronda de correcciones)

Finalmente mencionar que las últimas contribuciones relacionadas con la estrategia de planificación *por lotes* de etapas que maximice el *throughput* del sistema, detallada en el capítulo 4 de este trabajo, se encuentran en fase de redacción y posterior envío en el momento de finalización de esta tesis.

5.3 Vías de Continuación

Esta tesis ha tomado el *workflow* de procesamiento de resonancias magnéticas estructurales de la aplicación FreeSurfer como caso de ejemplo para proponer estrategias y mecanismos de planificación. El escenario definido ha sido la ejecución de múltiples instancias en un nodo de cómputo híbrido que se compone de varios procesadores y varias tarjetas con GPUs, de manera que se compartieran todos los recursos entre diferentes instancias del *workflow*. Las propuestas recogidas en este trabajo se han generalizado para ser aplicables a otros *workflows* científicos.

Como una primera línea de continuación se propone ampliar el análisis realizado en la planificación *por lotes* para otras partes o *subflows* de FreeSurfer que incluyan más etapas que usen GPU y en las que puedan aparecer nuevas proporciones entre etapas CPU y GPU. También hay que tener en cuenta que en el momento de presentar esta tesis, una nueva versión de FreeSurfer (versión 6) se encuentra en su fase beta, por lo que próximamente con su versión estable será interesante analizar las nuevas implementaciones de los procedimientos del *workflow* que se publiquen.

Con el objetivo de extender la aplicabilidad de las estrategias de planificación presentadas en este trabajo, planteamos otras vías de continuación analizando otras aplicaciones del mismo ámbito de la investigación neurocientífica o también denominada neuroimagen. Un buen ejemplo es la aplicación FSL (Funtional MRI of Brain *Software* Library) [37,38,39,40] es una libreria completa para el análisis de resonancias magnéticas. Contiene herramientas de análisis para resonancias funcionales, estructurales, o bien para otras técnicas de análisis de Difusión MRI [41]. Esta herramienta incorpora implementaciones en GPU en algunas de sus funcionalidades [42] que pueden acelerarse hasta dos órdenes de magnitud respecto a la ejecución con una sola CPU.

Dentro del mismo campo de la neuroimagen existe otro conjunto de aplicaciones más genérica, que se denomina *Nypipe* [43,44], en el que aplicar este trabajo. Nypipe es una herramienta que proporciona una interfaz uniforme para los principales *software*s de análisis del ámbito de la neuroimágen, entre ellos los ya mencionados FreeSurfer y FSL. Esta herramienta ofrece un entorno con el que se permite implementar *workflows* que combinen diferentes *software*s con una gran flexibilidad.

Otras via de continuación que nos planteamos es la de abrir colaboraciones con otros grupos de investigación en otros ámbitos de la ciencia. En este caso puede ser en el campo de la física de partículas, concretamente en el estudio de los Neutrinos dentro del *T2K Experiment* [47], en el cual el *Instituto de Física de Altes Energies,* presente en el campus de la UAB y al cual pertenece el autor de este trabajo, ha mostrado su interés en utilizar GPUs en los *pipeline*s de análisis y aplicar las estrategias de planificación y gestión de recursos que se han expuesto en esta tesis.

Si bien la mayor parte de los criterios de planificación empleados en este trabajo han surgido de un análisis y una posterior implementación «manual», tanto en la política de planificación *inter-workflow* como de la planificación *por lotes*, se desprenden algunas vías más de continuación en la línea de automatizar la adopción de estos criterios. Por una parte, se pueden incluir algunos mecanismos que gestionen de forma dinámica la política de planificación. En el caso del planificador *inter-workflow* se podría rellenar y/o modificar la tabla de restricciones con la monitorización de una ejecución. Esta monitorización tomaria principalmente medidas de uso de las GPUs, aspecto que ha resultado tener mayores restricciones en los escenarios planteados en este trabajo. Por ejemplo utilizando la herramienta nvidia-smi con la que podemos controlar el uso de memoria de la GPU. Añadiendo a esta monitorización un control de errores de las etapas GPU, podríamos ajustar el valor de la tabla de restricciones.

Por otra parte, en el caso de la planificación *por lotes*, hay también varios aspectos que se podrían llegar a automatizar con el objetivo de definir un plan de ejecución de forma más dinámica e ir modificandolo a lo largo de la ejecución. En el análisis de las afinidades de las tareas GPU con otras tareas ya sean GPU o no, se podría introducir mecanismos de aprendizaje a partir de ejecuciones incrementales en las que de manera paulatina se generaran combinaciones en las que se analizara el rendimiento obtenido y se almacenaran aquellas que muestren mejor afinidad para ser reproducidas en próximas ocasiones a lo largo de la ejecución.

Otro aspecto en el que se podría incluir un mecanismo automático es en la migración de tareas de ejecutarse en GPU a CPU. Este mecanismo haría una previsión de los tiempos de ejecución de la parte CPU del lote y de la parte GPU. Si esta previsión se desbalancea de manera que las GPUs se retrasan le eliminaría una tarea GPU de la lista de pendientes pasándola a CPU.

Finalmente también destacar que se podría llegar a automatizar la detección del cuello de botella

entre las partes del lote CPU o GPU. Comparando los tiempos entre los dos conjuntos de etapas (CPU y GPU) que componen el lote, podemos detectar si la parte GPU es cuello de botella o no. En caso de serlo y detectar un tiempo de espera, se puede mitigar el retraso asignando en tiempo de ejecución y para la ejecución del siguiente lote, una distribución diferente de las CPUs, de manera que se puedan usar más o menos CPUs en exclusiva por parte de las tareas GPU.

Como línea futura final se podría tomar en consideración la integración de esta política de planificación en entornos de diseño y gestión de *workflows* tal como hicimos en una colaboración intra-departamental a lo largo de esta tesis. Esto significaría ampliar el trabajo de implementar el *workflow* de FreeSurfer en la plataforma Galaxy [11], pero añadiendo en el *back-end* de planificación la estrategia de planificación *por lotes*.

6 Bibliografía

[1] http://www.nvidia.com/object/gpu.html

[2] http://www.gpgpu.org

[3] Using Multiple Graphics Cards as a General Purpose Parallel Computer :

Applications to Computer Vision

[4] The AMD Fusion Family of APUs. http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx

[5] Yi Yang, Ping Xiang, Mike Mantor, Huiyang Zhou CPU-assisted GPGPU on fused CPU-GPU architectures Proceeding HPCA '12 Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture Pages 1-12 IEEE Computer Society Washington, DC, USA ©2012

[6] NVIDIA CUDA Programming Guide - pp. 86, 136-139 – http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming Guide.pdf

[7] NVIDIAs Next Generation CUDA Compute Architecture: Fermi. White Paper (Oct 2009) http://www.nvidia.com/content/PDF/fermi white papers/NVIDIA Fermi Compute Architecture Whitepaper.pdf.

[8] Christopher G. Willard, Ph.D., Addison Snell, Michael Feldman, HPC Application Support for GPU Computing, Intersect360 Research (2015)

[9] FreeSurfer Wiki - http://surfer.nmr.mgh.harvard.edu/fswiki/

[10] T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics work ows. Bioinformatics, 20(17):3045{3054}, 2004.

[11] J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biology , 11(8):1{13}, 2010

[12] "The Evolution of Magnetic Resonance Imaging: 3T MRI in Clinical Applications" (Http://www.eradimaging.com/site/article.cfm?ID=426), Terry Duggan-Jahn, www.eradimaging.com

[13] Dale, A. M., Fischl, B., & Sereno, M. I. (1999). Cortical Surface-Based Analysis I: Segmentation and Surface Reconstruction. NeuroImage, 9(2), 179–194.

[14] Fischl, B., Sereno, M. I., & Dale, A. M. (1999). Cortical Surface-Based Analysis II: Ination, Flattering and a Surface-Based Coordinate System. NeuroImage, 9(2), 175–207.

[15] Fischl, B., Salat, D. H., Busa, E., Albert, M., Dieterich, M., Haselgrove, C., et al. (2002). Whole Brain Segmentation: Automated Labeling of Neuroanatomical Structures in the Human Brain. Neuron, 33, 341–355.

[16] Fischl, B., van der Kouwe, A., Destrieux, C., Halgren, E., Segonne, F.,Salat, D., et al. (2004). Automatically Parcellating the Human Cerebral Cortex. Cerebral Cortex, 14, 11–22.

[17] Edgar R. (2011). Acceleration of the Freesurfer Suite for Neuroimaging Analysis, GPU Technology Conference.

[18] https://www.khronos.org/opencl/

[19] Top500 - http://www.top500.org

[20] Green500 - http://www.green500.org

[21] Programming massively Parallel Processors, A Hands-on approach. David B.Kirk, Wen-mei W. Hwu (NVIDIA)

[22] Uma Boregowda, Venugopal Chakravarthy. Sharing of Cluster Resources among Multiple *Workflow* Applications. International Journal of Computer Science & Information Technology (IJCSIT) Vol 6, No 2, April 2014

[23] J. Yu and R. Buyya, “A taxonomy of scientific *workflow* systems for grid computing”, ACM Sigmod Rec., 2005.

[24] H. Zhao and R. Sakellariou, “Scheduling multiple DAGs onto heterogeneous systems,” Parallel Distrib. Process., 2006.

[25] Z. Yu and W. Shi, “A Planner-Guided Scheduling Strategy for Multiple *Workflow* Applications,” in 2008 International Conference on Parallel Processing - Workshops, 2008, pp. 1–8

[26] A. Kumar Singh, M. Shafique, Akask Kunmar, J. Henkel, “Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends”, ACM DAC´13 2013.

[27] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task graphs to Multiprocessors. ACM Computing Surveys, 31(4):406–471, 1999

[28] O. Beaumont, V.Boudet, and Y.Robert. A realistic model and an efficient heuristic for scheduling with heterogeneous processors. In 11th Heterogeneous Computing Workshop, 2002

[29] H. Topcuouglu, S. Hariri, and M. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” IEEE Transactions on Parallel and Distribution Systems, vol. 13, no. 3, pp. 260–274, 2002.

[30] Beisel, T., Wiersema, T., Plessl, C., Brinkmann, A.: Cooperative Multitasking for Heterogeneous Accelerators in the Linux Completely Fair Scheduler. In: Proceedings of the 22nd IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP). pp. 223–226. IEEE Computer Society, Santa Monica, CA, USA (Sep 2011)

[31] Pydiura Nikolay, Karpov Pavel, Blume Yaroslav, Hybrid CPU-GPU calculations – a promising future for computational biology, Third International Conference "High Performance Computing" HPC-UA 2013 (Ukraine, Kyiv, October 7-11, 2013)

[32] Nikolay Pydiura, Pavel Karpov and Yaroslav Blume, On the Efficiency of CPU and Hybrid CPU-GPU Systems in Computational Biology Tasks, Comput. Sci. Appl. Volume 1, Number 1, 2014, pp. 48-59

[33] Gustavo Wolfmann, Optimización de cómputo paralelo científico en un entorno híbrido *Multicore* - MultiGPU, XIV Workshop de Investigadores en Ciencias de la Computación (2012)

[34] Richard Membarth, Jan-Hugo Lupp, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert, Dynamic Task-Scheduling and Resource Management for GPU Accelerators in Medical Imaging, Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS), Munich, Germany, February 28–March 2, 2012.

[35] Ji Liu, Esther Pacitti, Patrick Valduriez, Marta Mattoso, A Survey of Data-Intensive Scientific *Workflow* Management, Journal of Grid Computing 01/2015

[36] NVIDIA System Management Interface - https://developer.nvidia.com/nvidia-system-management-interface

[37] FSL wiki - http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/

[38] M.W. Woolrich, S. Jbabdi, B. Patenaude, M. Chappell, S. Makni, T. Behrens, C. Beckmann, M. Jenkinson, S.M. Smith. Bayesian analysis of neuroimaging data in FSL. NeuroImage, 45:S173-86, 2009

[39] S.M. Smith, M. Jenkinson, M.W. Woolrich, C.F. Beckmann, T.E.J. Behrens, H. Johansen-Berg, P.R. Bannister, M. De Luca, I. Drobnjak, D.E. Flitney, R. Niazy, J. Saunders, J. Vickers, Y. Zhang, N. De Stefano, J.M. Brady, and P.M. Matthews. Advances in functional and structural MR image analysis and implementation as FSL. NeuroImage, 23(S1):208-19, 2004

[40] M. Jenkinson, C.F. Beckmann, T.E. Behrens, M.W. Woolrich, S.M. Smith. FSL. NeuroImage, 62:782-90, 2012

[41] Patric Hagmann, , Lisa Jonasson, , Philippe Maeder, , Jean-Philippe Thiran, , Van J. Wedeen, , and Reto Meuli, Understanding Diffusion MR Imaging Techniques: From Scalar Diffusion-weighted Imaging to Diffusion Tensor Imaging and Beyond (2006) RadioGraphics Journal, Volume 26, Issue suppl 1

[42] Hernandez M, Guerrero GD, Cecilia JM, Garcia JM, Inuggi A, Jbabdi S, Behrens TEJ, Sotiropoulos SN, (2013) Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using GPUs. PLoS ONE 8(4):e61892

[43] http://nipy.org/nipype/

[44] Gorgolewski K, Burns CD, Madison C, Clark D, Halchenko YO, Waskom ML, Ghosh SS. (2011). Nipype: a flexible, lightweight and extensible neuroimaging data processing framework in Python. Front. Neuroinform. 5:13

[45] DRMAA Hrabri Rajic, Roger Brobst, Waiman Chan, Fritz Ferstl, Jeff Gardiner, Andreas Haas, Bill Nitzberg, and John Tollefsrud, “Distributed resource management application api specification 1.0,” 2004.

[46] SLURM: Simple Linux Utility for Resource Management, Yoo, A., Jette, M. & Grondona, M. (2003). Job Scheduling Strategies for Parallel Processing, volume 2862 of Lecture Notes in Computer Science, (pp. 44–60). Springer-Verlag.

[47] http://t2k-experiment.org/

[48] http://sourceforge.net/projects/freesurferscheduler/

Glosario

API: Application Programming Interface

CPU: Central Processing Unit

CUDA: Compute Unified Device Architecture

DRMAA: Distributed Resource Manager Application API

FIFO: First In First Out

FS: FreeSurfer

GFLOPS: Giga Floating Operations Per Second

GPGPU: General Purpose GPU

GPU: Graphic Processing Unit

HPC: High Performance Computing

MRI: Magnetic Ressonance Image

NFS: Network File System

NUMA: Non Uniform Memory Access

NVPROF: NVIDIA Profiler

SIMD: Single Instruction Multiple Data

SLURM: Simple Linux Utility for Resource Management

SMPS: Stream Multi-processors

Créditos

Este trabajo ha sido posible gracias a la colaboración entre:

Departamento de Arquitectura de Computadores y Sistemas Operativos

y parcialmente financiado por:

Talent Empresa

Projecte: “Estudi sobre l’aplicació de noves tecnologies de la informació per accelerar el processament de ressonàncies magnètiques del cervell”

Anualitats 2011-2014

High Performance Computing Applications for Science and Engineering

Proyecto: “Desafíos de los entornos computacionales multi-many core”

MICINN-España convocatoria TIN2011-28689-C02-01.