
# UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

## ESCUELA DE INGENIERÍA INFORMÁTICA

### INGENIERÍA INFORMÁTICA

# PROYECTO FIN DE CARRERA

# “Paralelización del algoritmo de desenredo y suavizado de mallas de tetraedros y su implementación en procesadores gráficos”

Autor: Marcos Jesús Santana Quintana
Tutores: Domingo Benítez Díaz
Eduardo Rodríguez Barrera
Noviembre del 2013

# Universidad de Las Palmas de Gran Canaria

Escuela de Ingeniería en Informática

# Ingeniería en Informática

Proyecto fin de carrera

**“Paralelización del algoritmo de desenredo y suavizado de mallas de tetraedros y su implementación en procesadores gráficos”**

Alumno: Marcos Jesús Santana Quintana

Tutores: Domingo Benítez Díaz
Eduardo Barrera Rodríguez

# Agradecimientos

Ante todo agradecer el apoyo recibido por mi familia. Nadie mejor que ellos saben la dedicación y el esfuerzo que me ha supuesto no solo este proyecto final de carrera, sino la superación de cada uno de los cursos de los que constaba los estudios de licenciatura. Su apoyo, no sólo a nivel económico, ha sido vital para que hoy pueda presentar este proyecto y con él terminar mis estudios universitarios. Será algo que nunca podré agradecerles lo suficiente.

Dar también las gracias a mis tutores. Al Dr. Eduardo Rodríguez Barrera por sus explicaciones y su incondicional ayuda para hacerme lo más fácil posible mi integración en un mundo de métodos y algoritmos que desconocía hasta el momento. Y al Dr. Domingo Benítez Díaz por su colaboración incesante a lo largo de todos estos meses de desarrollo e investigación, donde a pesar de las dificultades encontradas en el camino, siempre ha estado dispuesto a guiarme y orientarme para encontrar la mejor solución.

Por último, no puedo olvidarme de mis compañeros de carrera: Diego Nieto, Daniel Tejera, Imanol Pérez y Mahy Quintana. Hemos compartido estos últimos 5 años con todo tipo de vivencias y anécdotas pero sobre tod, hemos crecido en conocimiento de una forma espectacular. Más que compañeros y amigos somos un pequeño grupo de aventureros que no nos cansamos de averiguar qué es lo hay más allá, y que no nos importa cuánto tengamos que esforzarnos para llegar a entender o para manejar algo. Estoy seguro de que esa curiosidad que nos ha traído hasta aquí nos hará llegar tan más lejos como nos propongamos.

# Resumen

Este proyecto tiene como objetivo general el estudio de la implementación del algoritmo de desenredo y suavizado simultáneo de mallas de tetraedros (S.U.S., Simultaneous Untangling and Smoothing) en un entorno de computación dotado de GPU. El algoritmo S.U.S. permite mejorar la calidad de mallas 3D formadas por tetraedros. Disponer de mallas de buena calidad, como se explicará más adelante, permite mejorar los resultados de ciertas técnicas numéricas que usan las mallas como soporte. En particular, del método de los elementos finitos (FEM, Finite Element Method) es muy sensible a la calidad de las mallas y sus resultados se ven perturbados por el uso de mallas de baja calidad, por lo que usar el S.U.S. como preproceso previo al FEM en general mejora los resultados numéricos finales.

Por otra parte, el uso de GPUs (Graphics Processing Units) en entornos de computación científica para realizar cálculos masivamente paralelizables está cada vez más extendido. Las GPUs han dejado el ámbito exclusivo del procesamiento gráfico para convertirse en potentes procesadores numéricos capaces de realizar una enorme cantidad de cálculos por unidad de tiempo. Su bajo coste económico frente a su alta capacidad de cálculo hace de las GPUs una opción muy razonable para un entorno de computación de altas prestaciones (HPC, High Performance Computing). La contribución de las GPUs al mundo de la HPC puede constatarse visitando el ranking top500 (top500.org) que clasifica cada seis meses a los supercomputadores con mayor potencia computacional. Aproximadamente, el 8% de los supercomputadores que figuran en el ranking de Junio de 2013 están formados, por GPUs de NVIDIA.

Dicho esto, hay que destacar que la programación de las GPUs, especialmente cuando se desea utilizar todo su potencial de cálculo, no siempre es un problema fácil y, debido a su carácter vectorial, el rendimiento que se obtiene puede depender del tipo de aplicación que se pretende programar. Concretamente, la programación de aplicaciones que tratan con mallas no estructuradas, como es nuestro caso, es un problema abierto.

Finalmente, se explica brevemente la organización de esta memoria, así como los temas que se abordarán en cada uno de los capítulos.

- Capítulo 1. Introducción. En primer lugar se expondrán los objetivos que se desean alcanzar en este proyecto de fin de carrera, así como los principios básicos del método de elementos finitos. Para entrar en contextualización con el problema que se aborda en este proyecto fin de carrera se explicará el software denominado S.U.S. Code, Por último, se abordará de forma concisa la arquitectura CUDA y más detenidamente la microarquitectura de las máquinas basadas en GPU.

- Capítulo 2. Paralelización del S.U.S. Code. En dicho capítulo se reúnen todos los datos teóricos anteriormente explicados para la obtención de la paralelización del S.U.S. Code. Se aborda la paralelización lógica de los datos y posteriormente la paralelización del código. Para llevar a cabo la paralelización del código, se explicará detalladamente cuales han sido las diferentes fases llevadas a cabo a lo largo del proyecto. Una vez obtenida la paralelización del S.U.S. Code se ha analizado los resultados obtenidos de la ejecución de una batería de pruebas para medir la aceleración, la eficiencia del paralelismo, la sensibilidad al coloreado de la malla, etc. Para ello se han utilizados dos tipos distintos de GPUs de NVIDIA. Finalmente se describen varias optimizaciones del código paralelo y se evalúan sus prestaciones en las mencionadas GPUs.

- Capítulo 3. Conclusiones y líneas futuras. El este apartado se recoge las conclusiones obtenidas a lo largo del proyecto y se presenta una serie de líneas de trabajo abiertas en la continuación de este proyecto de fin de carrera.

- Capítulo 4. Bibliografía. Se añaden las referencias bibliográficas que se han usado para el estudio y realización de este proyecto.

- Capítulo 5. Anexo. En este último apartado se presenta una pequeña guía de usuario, así como un tutorial para la instalación de CUDA y de las librerías PAPI.

A continuación se describirán los diferentes dispositivos usados para la ejecución y desarrollo de este proyecto:

**CPUs**

- Ordenador personal: Intel core i5.
- Computador del Instituto Universitario SIANI denominado como 'Vector': Intel core Xeon E5620.

**GPUs**

**Características de la Nvidia GeForce GT 610 (Tarjeta gráfica personal)**

Esta tarjeta gráfica de gama media/baja está formada por una sola GPU, que a su vez consta de un solo multiprocesador. Esta tarjeta está diseñada con una arquitectura Fermi (se explicará posteriormente los detalles de la misma).

**Tabla 1. Características de la Nvidia GeForce GT 610**

| | |
|---|---|
| **Nombre de la GPU** | GeForce GT 610 |
| **Capacidad memoria global** | 2 GB |
| **Frecuencia de reloj de los núcleos** | 1.62 GHz |
| **Frecuencia de reloj de la memoria** | 667 MHz |
| **Mayor capacidad de CUDA** | 2.1 |
| **Ancho de banda del bus de la memoria** | 64 bits |
| **Capacidad memoria cache L1** | 64 KB |
| **Número de multiprocesadores** | 1 |
| **Número de núcleos por multiprocesador** | 48 |
| **Número de núcleos totales** | 48 |


Figura 1. Nvidia GeForce GT 610.

## Características de la Nvidia Tesla S2050 (Tarjeta gráfica de 'vector.iusiani.ulpgc.es')

Esta tarjeta gráfica de gama alta está formada por 4 GPU's, donde cada una consta de 14 multiprocesadores. Esta tarjeta está constituida de 4 Nvidia Tesla M2050, donde 2 son compatibles con CUDA. A continuación se indicarán las características de una de las 4 Nvidia Tesla M2050, puesto que durante el lanzamiento de un proceso paralelo sólo se usa una tarjeta. Esta tarjeta, al igual que la GeForce GT 610, está diseñada con una arquitectura Fermi.

Tabla 2. Características de la Nvidia Tesla S2050.

| | |
|---|---|
| **Nombre de la GPU** | Tesla M2050 |
| **Capacidad memoria global** | 2.6 GB |
| **Frecuencia de reloj de los núcleos** | 1.15 GHz |
| **Frecuencia de reloj de la memoria** | 1.5 GHz |
| **Mayor capacidad de CUDA** | 2.0 |
| **Ancho de banda del bus de la memoria** | 384 bits |
| **Capacidad memoria cache L1** | 64 KB |
| **Capacidad memoria cache L2** | 768 KB |
| **Número de multiprocesadores** | 14 |
| **Número de núcleos por multiprocesador** | 32 |
| **Número de núcleos totales** | 448 |

Figura 2. Nvidia Tesla S2050.

# Índice

# Índice de Figuras

# Índice de Tablas

# 1. Introducción

## 1.1 Motivación general

Hace años la tecnología evolucionaba rápidamente fabricando procesadores de un solo núcleo (core). Estos procesadores trabajaban a altas frecuencias, pagando de este modo en mayor medida los fallos relacionados con la memoria. Otra desventaja que ofrecía este tipo de procesadores eran las altas temperaturas que podían alcanzar a causa de las frecuencias con las que trabajaban. Esto generaba un serio problema de refrigeración y control de temperatura del mismo.

Pocos años después la tecnología evolucionó, y se comenzó a fabricar procesadores multi-core (muchos núcleos) que trabajaban a menor frecuencia y disponían de una memoria compartida. Este tipo de hardware resultada más eficaz y eficiente. Tal ha llegado esa evolución que hoy en día se puede encontrar en cualquier casa particular computadores con procesadores de 4 núcleos trabajando éstos a frecuencias medias-altas.

Por lo tanto se puede intuir que la evolución de la informática, tanto software como hardware avanza en esa dirección; es decir, a sistemas con múltiples núcleos y algoritmos multi-hilos. Otra tecnología que trabaja desde hace tiempo con esta filosofía son los procesadores gráficos de tipo GPU. Estos procesadores con gran cantidad de núcleos son capaces de realizar procesamiento de datos (sobre todo imágenes) rápidamente gracias a la división del trabajo a realizar entre los distintos cores.

La motivación general de este proyecto es la de conocer con detalle cómo funciona la tecnología de los procesadores gráficos de tipo GPU, su lenguaje de bajo nivel, así como la interfaz de programación *CUDA* que nos ofrece las tarjetas gráficas de *NVIDIA*. Ya que este tipo de tecnología y de programación se utiliza mucho en el campo de la computación de altas prestaciones HPC, resulta interesante aprender los detalles de esta tecnología implementando un algoritmo de investigación desarrollado por el Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (*SIANI*) de la Universidad de Las Palmas de Gran Canaria.

## 1.2 Objetivos del proyecto

El objetivo principal del proyecto es conseguir una implementación paralela que sea lo más escalable y eficiente posible para los recursos hardware disponibles en procesadores gráficos de tipo GPU para el algoritmo de desenredo y suavizado simultáneo de mallas de tetraedros. Para ello se ha de seguir una serie de pasos previos.

Se pretende conocer con detalle el lenguaje de bajo nivel de un procesador gráfico GPU y la interfaz de CUDA para hacer posible una implementación del algoritmo lo más eficiente, eficaz y productiva posible. Para ello se intentará hacer uso de la totalidad de los recursos hardware de los que dispone el procesador GPU en el que se ejecute el algoritmo S.U.S.

En esencia, el objetivo final es implementar el algoritmo S.U.S. code para que se pueda ejecutar en un procesador gráfico de tipo GPU. Una vez se haya conseguido, se pretende realizar un estudio de prestaciones, comparando los tiempos de procesamiento con la ejecución secuencial de la CPU, analizar la cantidad de recursos aprovechados, etc.

Por último, una vez se tenga los resultados obtenidos del estudio de prestaciones y escalabilidad, se pretenderá realizar propuestas de mejora al algoritmo. De este modo se podrá evaluar dichas propuestas e intentar obtener una implementación lo más eficientemente posible en función de los recursos hardware.

Por tanto los objetivos son:

- Conocer con detalle el lenguaje ensamblador de un procesador gráfico GPU así como la interfaz de programación CUDA.

- Implementar el algoritmo de desenredo y suavizado simultáneo de mallas en un procesador gráfico de tipo GPU.

- Realizar un estudio de prestaciones de la implementación del algoritmo en distintas máquinas basadas en GPU.

- Realizar propuestas de mejora de la eficiencia del cómputo y evaluar las prestaciones.

## 1.3 Preámbulo

A continuación se comentará brevemente qué es una malla de tetraedros y para qué son útiles en el método de elementos finitos (MEF). También se explicará qué es el método del Meccano y qué relación tiene con nuestro objetivo.

### 1.3.1 Mallas de tetraedros y MEF

El método de los elementos finitos (MEF en español o FEM en inglés) es un método numérico general para la aproximación de soluciones de ecuaciones diferenciales parciales muy utilizado en diversos problemas de ingeniería y física. Este método está pensado para ser usado en CPU y permite resolver ecuaciones diferenciales asociadas a un problema físico sobre geometrías complicadas. El MEF se usa en el diseño y mejora de productos, en aplicaciones industriales y en la simulación de sistemas físicos y biológicos complejos. La variedad de problemas a los que puede aplicarse ha crecido enormemente, siendo el requisito básico que las ecuaciones constitutivas y ecuaciones de evolución temporal del problema a considerar sean conocidas de antemano.

El MEF permite obtener una solución numérica aproximada sobre un cuerpo, estructura o dominio (medio continuo) sobre el que están definidas ciertas ecuaciones diferenciales en forma débil o integral que caracterizan el comportamiento físico del problema dividiéndolo en un número elevado de subdominios no-intersectantes entre sí denominados “elementos finitos”. El conjunto de elementos finitos forma una partición del dominio también denominada discretización. Dentro de cada elemento se distinguen una serie de puntos representativos llamados “nodos”. Dos nodos son adyacentes si pertenecen al mismo elemento finito; además, un nodo sobre la frontera de un elemento finito puede pertenecer a varios elementos. El conjunto de nodos considerando sus relaciones de adyacencia se llama “malla”.

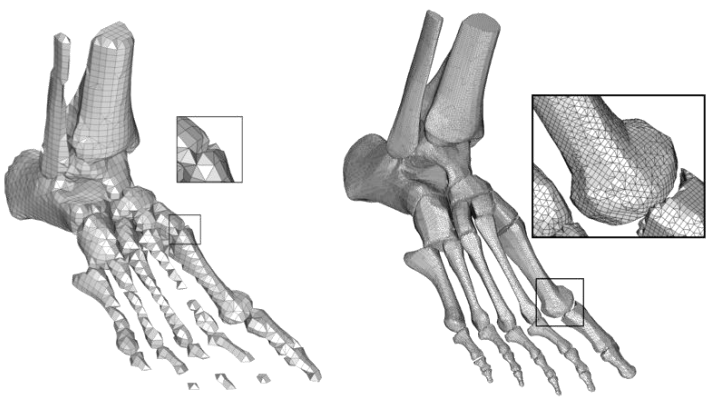

**Figura 3. Malla tridimensional usada en el campo de la medicina.**

Los cálculos se realizan sobre una malla de puntos (llamados nodos), que sirven a su vez de base para discretización del dominio en elementos finitos. Por ejemplo en la Figura 3 se puede ver una malla tridimensional formada por miles de nodos usada en la disciplina de la medicina. La generación de la malla se realiza usualmente con programas especiales llamados generadores de mallas, en una etapa previa a los cálculos que se denomina pre-proceso. De acuerdo con estas relaciones de adyacencia o conectividad se relaciona el valor de un conjunto de variables incógnitas definidas en cada nodo y denominadas grados de libertad. El conjunto de relaciones entre el valor de una determinada variable entre los nodos se puede escribir en forma de sistema de ecuaciones lineales (o linealizadas). La matriz de dicho sistema de ecuaciones se llama matriz de rigidez del sistema. El número de ecuaciones de dicho sistema es proporcional al número de nodos.

Típicamente el método de los elementos finitos se programa computacionalmente para calcular el campo de desplazamientos y, posteriormente, a través de relaciones cinemáticas y constitutivas las deformaciones y tensiones respectivamente, cuando se trata de un problema de mecánica de sólidos deformables o más generalmente un problema de mecánica de medios continuos. El método de los elementos finitos es muy usado debido a su generalidad y a la facilidad de introducir dominios de cálculo complejos (en dos o tres dimensiones). Además el método es fácilmente adaptable a problemas de transmisión de calor, de mecánica de fluidos para calcular campos de velocidades y presiones (mecánica de fluidos computacional, CFD) o de campo electromagnético. Dada la imposibilidad práctica de encontrar la solución analítica de estos problemas, con frecuencia en la práctica ingenieril los métodos numéricos y, en particular, los elementos finitos, se convierten en la única alternativa práctica de cálculo.

Una importante propiedad del método es la convergencia; si se consideran particiones de elementos finitos sucesivamente más finas, la solución numérica calculada converge rápidamente hacia la solución exacta del sistema de ecuaciones. Esta convergencia del método depende en gran medida de la calidad de las mallas que analice dicho método. Si se está analizando una malla de poca calidad el método tendrá pocas posibilidades de converger durante el proceso. Algunos generadores automáticos de mallas tridimensionales construyen, en una primera aproximación, mallas *válidas* pero con elementos de poca calidad. Además, en ciertos casos especiales, pueden aparecer elementos invertidos, dando lugar a lo que llamamos mallas *enredadas*. Esto afecta a las posibilidades de convergencia del método de elementos finitos para estas mallas.

Se considera que una malla no es válida cuando contiene elementos (tetraedros) planos (o casi planos) o con volumen negativo. Este tipo de mallas no pueden usarse como soporte del MEF ni de ninguna otra técnica numérica similar.

### 1.3.2 Método del Meccano

A continuación, se presenta una técnica novedosa de generación de mallas 3D, conocida como el método del Meccano. Este método ha sido presentado hace relativamente poco tiempo en el artículo *The Meccano Method for Isogeometric Solid Modeling*[2][3] por profesores de la Universidad de las Palmas de Gran Canaria, así como del Instituto Universitario SIANI. El algoritmo S.U.S. (el cuál es caso de estudio en este proyecto) es una parte esencial del método del Meccano, ya que además de mejorar la calidad de las mallas, es capaz de convertir mallas no válidas en mallas válidas, al tiempo que mejora la calidad general de la malla.

Los modelos CAD (Computer-Aided Design) por lo general sólo definen el límite de un sólido, pero para la aplicación de análisis isogeométrico o para la aplicación de métodos numéricos como el MEF requieren una representación totalmente volumétrica. Lo cual, supone un problema en el contexto del análisis de isogeométrico el cómo generar una malla volumétrica de un sólido a partir de la descripción CAD de su frontera.

El método de Meccano soluciona entre problema al construir una representación tridimensional de sólidos complejos para la aplicación del análisis isogeométrico. Esta técnica sólo necesita la superficie del sólido como datos de entrada. La clave de este método reside en la obtención de una parametrización volumétrica entre el sólido y un dominio paramétrico sencillo.

El método de Meccano consta de varias fases:

1. Parametrización de la superficie.
2. Refinamiento de Kossaczky.
3. Mapeado de la superficie.
4. Desenredo y suavizado de la malla.

A continuación se explicará brevemente en qué consiste cada una de las fases del método de Meccano, así se tendrá una visión general del entorno y porqué el algoritmo S.U.S. recibe una malla enredada.

1. Parametrización de la superficie.

Supongamos que queremos generar una malla tridimensional de un sólido como el de la Figura 4. En primer lugar, se particiona la superficie del sólido a una determinada distancia de tal modo, que cada parte del mismo esté representada en una de las caras de un cubo. Este proceso se lleva a cabo mediante la técnica de parametrización Floater. En la Figura 5 se puede observar cómo se fracciona cada una de las partes del sólido de modo que se puede observar como cada fracción corresponde a una cara del cubo; representándose en esta cara los puntos de cada fracción.

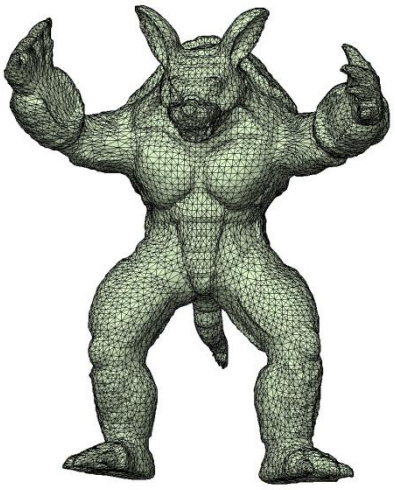

**Figura 4. Sólido tridimensional.**

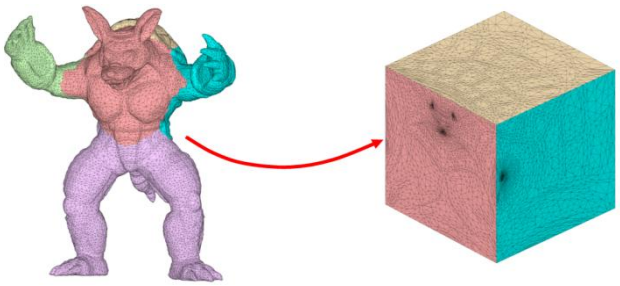

**Figura 5. Parametrización del sólido.**

2. Refinamiento de Kossaczky.

Una vez parametrizada la superficie del sólido, hacemos uso del algoritmo de refinamiento de Kossaczy. Este algoritmo que consiste en generar nuevos puntos dentro del cubo, de modo que éste quede “relleno” de nuevos nodos conectados con los nodos iniciales, colocados equidistantemente. De este modo generamos una malla para el cubo, añadiendo nodos colocados armónicamente. Se puede ver en la Figura 6 un pequeño ejemplo del refinamiento de Kossaczky.

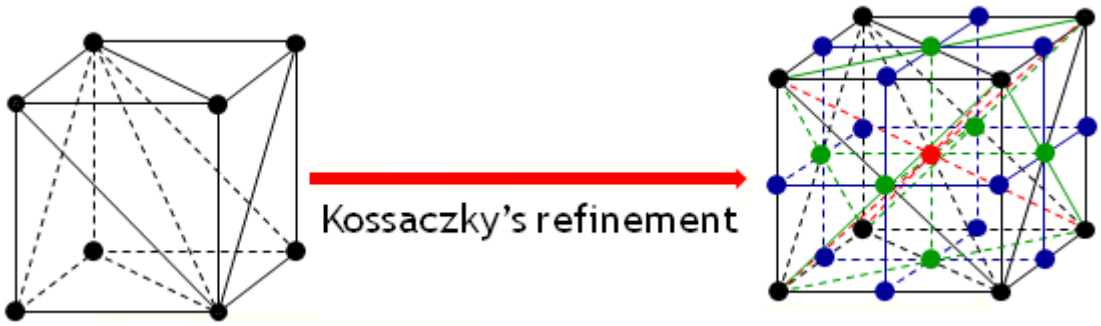


Figura 6. Ejemplo del refinamiento de Kossaczky.

Volviendo al ejemplo inicial, podemos ver en la Figura 7, que tras el refinamiento de Kossaczky muchos de los nodos que pertenecen a la superficie del sólido, se han recolocado intentando imponer una misma distancia entre ellos sin modificar la forma inicial de la figura. Aún así, se han añadido nuevos nodos creando un volumen en el interior del cubo.


Figura 7. Refinamiento de Kossaczky para el ejemplo del sólido.

3. Mapeado de la superficie.

El siguiente paso, supone realizar un mapeado de la superficie de las caras del cubo para volver a la figura inicial. Tras realizar este paso visualmente no obtenemos ninguna mejora destacable, puesto que la superficie del sólido seguirá siendo la misma. Pero si se mira en su interior; realizando una sección del sólido, se pueden ver los nuevos nodos y sus nuevas conexiones que se han creado tras el refinamiento de Kossaczky.

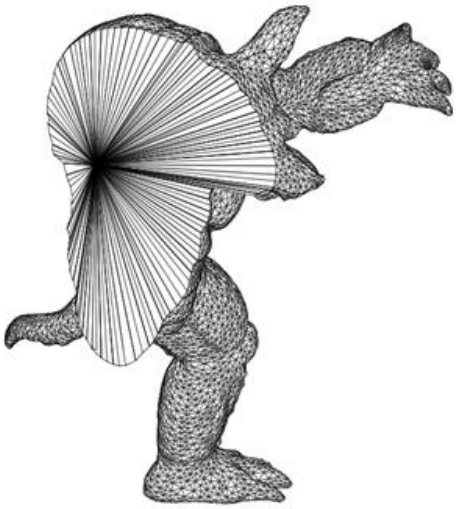

Figura 8. Malla enredada.

Tras realizar el mapeado de la superficie del cubo, los nuevos nodos creados tienden a agruparse en el centro de la figura, formando así tetraedros enredados; es decir, elementos planos o con volumen negativo. Es por esto que necesitamos un algoritmo que nos desenrede la malla y la optimice lo máximo posible.

4. Desenredo y suavizado de la malla.

Esta cuarta fase, trata de desenredar la malla tridimensional generada y simultáneamente suavizarla de modo que quede como resultante una malla tridimensional de calidad. Este proceso se lleva a cabo minimizando una función objetivo en cada nodo, con el objetivo de generar tetraedros, cuyos lados sean lo más parecidos posibles a un triángulo equilátero.

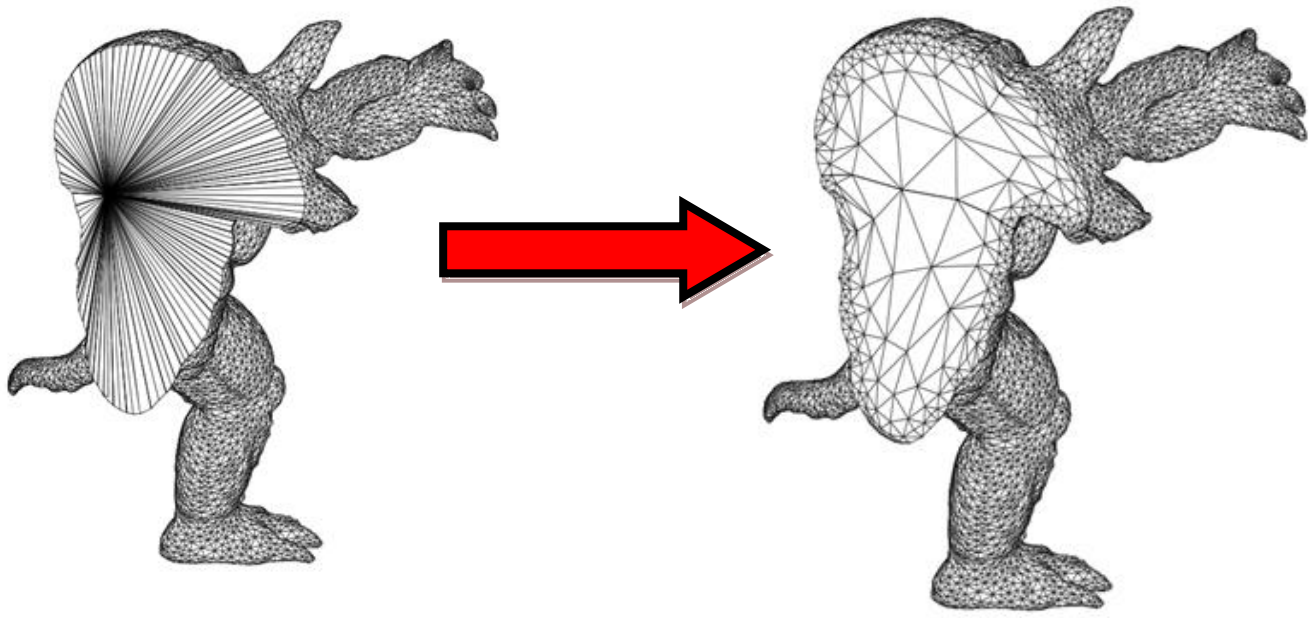

**Figura 9. Proceso del desenredo y suavizado de la malla.**

De esta forma finalizaría el proceso de la generación de una malla tridimensional partiendo de un sólido complejo. De entre las diferentes fases de las que consta el método del Meccano, la fase de desenredo y suavizado simultáneo de la malla es la más costosa de las fases. Por este motivo se han abierto varias líneas de estudio con el objeto de minimizar el tiempo de respuesta de dicho algoritmo y a su vez del método del Meccano.

### 1.4 S.U.S. code

Cómo se comentó anteriormente, uno de los aspectos fundamentales para la simulación de problemas físicos reales es la utilización de mallas de calidad. Por tanto, es necesario utilizar procedimientos capaces de suavizar y desenredar mallas preexistentes.

El suavizado es una de las técnicas más comunes para mejorar la calidad de una malla válida, esto es, una malla que no tiene elementos invertidos. En esencia, esta técnica consiste en desplazar cada nodo de la malla hasta una nueva posición que optimiza una función objetivo. Esta función está construida a partir de cierta medida de calidad de la *submalla local N* (*v*), constituida por los tetraedros conectados al *nodo libre v*. Éste es un proceso que tan sólo mejora la malla localmente y, por ello, se ha de repetir un número suficiente de veces hasta alcanzar la calidad global requerida.

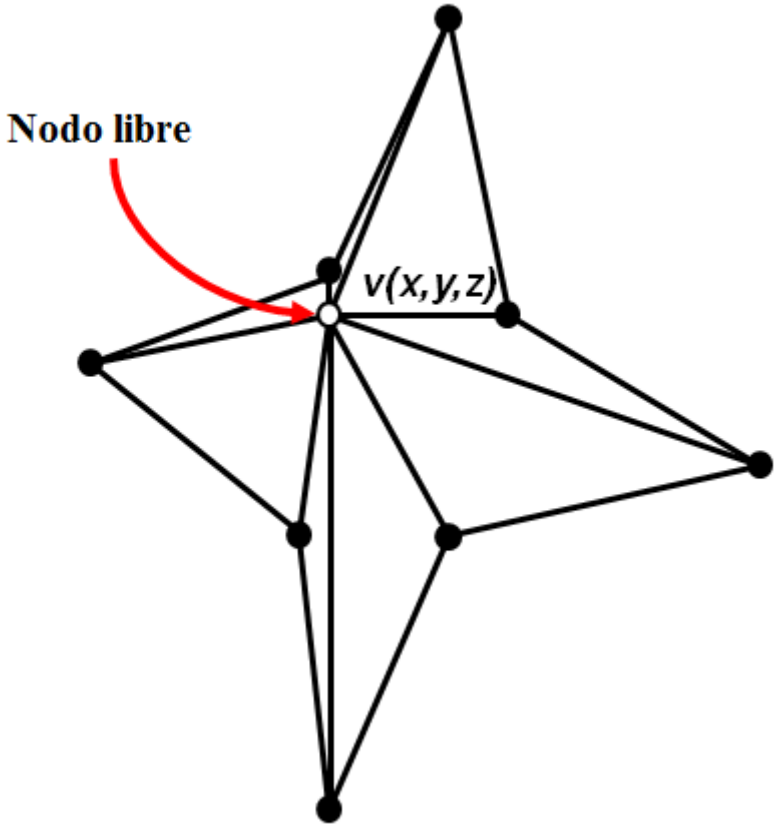


Figura 10. Estrella de tetraedros formados por varios nodos.

En general, las funciones objetivo que mejores resultados dan son las que presentan barreras en el límite de la región factible. En este contexto la región factible es el conjunto de puntos donde puede estar localizado *v* para conseguir que *N* (*v*) sea válida. En ocasiones este conjunto puede ser vacío, por ejemplo, cuando la frontera fija de *N* (*v*) está enredada. Las barreras evitan que el nodo libre abandone la región factible si inicialmente *v* estaba situado en su interior. Sin embargo, por idéntica razón, impedirán que acceda a ella cuando parte de una posición exterior. En esta última situación hay elementos en *N* (*v*) con volumen negativo y, por tanto, la submalla local está enredada.

Ante este problema se puede actuar como propone Freitag[4], en donde el proceso de optimización se divide en dos etapas. En la primera se desenreda la malla mediante un proceso de optimización que maximiza el volumen de los elementos invertidos y, en la segunda, se suaviza haciendo uso las funciones objetivo antes mencionadas. El algoritmo propuesto en el artículo de *Simultaneous untangling and smoothing of tetrahedral meshes*[1] propone una alternativa a este procedimiento, de manera que el desenredo y el suavizado se lleven a cabo en una misma etapa. Para ello se han modificado las funciones objetivo haciendo que se eviten las singularidades introducidas por las barreras pero manteniendo las posiciones en que se encontraban los mínimos en las funciones originales.

Este algoritmo presenta una manera de evitar las singularidades introducidas por funciones de optimización de mallas con barrera. Para ello, se modifican estas funciones de manera que pasen a ser regulares en todo $R^3$. Así, las funciones objetivo modificadas pueden ser utilizadas para desenredar y suavizar la malla simultáneamente. La regularidad que exhiben estas funciones modificadas hace posible utilizar algoritmos usuales de optimización como mínimo descenso, gradiente conjugado, Newton, etc. En principio, la modificación podría ser también aplicable a otras funciones objetivo, es decir, funciones con barrera en el límite de la región factible.

**Funciones objetivo**

Las funciones objetivo se pueden construir partiendo de alguna medida de calidad de los tetraedros. Sin embargo, aquellas que se obtienen mediante operaciones algebraicas son especialmente adecuadas para este propósito ya que el coste computacional requerido para su evaluación puede ser muy bajo.

Sea *T* un elemento tetraédrico en el espacio físico cuyos vértices están dados por x*k* = $(xk, yk, zk)^T$ $\epsilon$ $T_R$, *k* = 0, 1, 2, 3 y sea $T_R$ el elemento de referencia con vértices en $u_0 = (0,0,0)^T$ , $u_1 = (1,0,0)^T$ , $u_2 = (0,1,0)^T$ y $u_3 = (0,0,1)^T$ . Si se elige $x_0$ como vector de traslación, la aplicación afín que transforma $T_R$ en *T* es x =*A*u + $x_0$, donde *A* es la matriz jacobiana de la aplicación referida al nodo $x_0$, y expresada como $A = (x_1 - x_0, x_2 - x_0, x_3 - x_0)$. Sea ahora $T_I$ un tetraedro equilátero de lado unitario y cuyos vértices están situados en $v_0 = (0,0,0)^T$ , $v_1 = (1,0,0)^T$ , $v_2 = (1/2, 3^{1/2}/2, 0)^T$, $v_3 = (1/2, 3^{1/2}/6, 2^{1/2}/3^{1/2})^T$. Sea v =*W*u la aplicación lineal que transforma $T_R$ en $T_I$ , siendo *W* = $(v_1, v_2, v_3)$ su matriz jacobiana.

Por tanto, la aplicación afín que transforma $T_I$ en *T* está dada por x =$AW^{-1}$ v + $x_0$, y su matriz jacobiana es $S = AW^{-1}$. La matriz *S* es independiente del nodo elegido como referencia; diremos que es *invariante nodal*.

Una posible medida de calidad del tetraedro *T* está dada por $q = 3\,\sigma^{2/3}/\,|s|^2$, donde σ = det (*S*). Ésta es una *medida de calidad algebraica* de *T*. El máximo valor que puede tomar esta medida es la unidad y corresponde al tetraedro equilátero. Además, cualquier tetraedro plano o degenerado tiene medida nula. Se puede obtener una función de optimización a partir de esta medida de calidad. Así, sea x = $(x, y, z)^T$ la posición del nodo libre *v*, y *S*m la matriz jacobiana correspondiente al *m*-ésimo tetraedro de *N* (*v*). Se definirá la función objetivo de x asociada al *m*-ésimo tetraedro como

$$\eta_m = \frac{|S_m|^2}{3\sigma^{\frac{2}{3}}} \tag{1}$$

Así, la función objetivo correspondiente a *N* (*v*) se puede construir a partir de la *p*-norma de ($\eta_1$, $\eta_2, \ldots, \eta_M$) como

$$\left|K_\eta\right|_p(x) = \left[\sum_{m=1}^{M} \eta_m^p(x)\right]^{\frac{1}{p}} \tag{2}$$

donde M es el número de tetraedros en N(v).

Si denotamos por Hm los semiespacios definidos por $\sigma_m(x) \geq 0$, entonces, la región factible será el interior del poliedro P definido como:

$$P = \bigcup_{m=1}^{M} H_m$$

Las funciones objetivo $\left|K_\eta\right|_p$ son suaves en la región factible pero tienen a infinito cuando v se

aproxima a la frontera de P. En esta frontera aparece una barrera que impide a los algoritmos basados en gradiente alcanzar el mínimo requerido cuando comienzan desde una posición de v externa a la región factible. Además, en el caso en que no exista dicha región $\left|K_\eta\right|_p$ deja de tener sentido como función de optimización.

**Funciones objetivos modificadas**

En el artículo de *Simultaneous untangling and smoothing of tetrahedral meshes*[1] se propuso una modificación de la función objetivo (2), de manera que se elimine la barrera y se consiga que la nueva función sea suave en $R^3$. Un requisito esencial es que los mínimos de la función objetivo original y la modificada sean casi idénticos cuando *int P* ≠ *Ø*. Nuestra modificación consiste en sustituir *σ* en (2) por la función creciente y positiva:

$$h(\sigma) = \frac{1}{2}\left(\sigma + \sqrt{\sigma^2 + 4\delta^2}\right) \tag{3}$$

donde $\sigma = h(0)$. En la Figura 11 está representada la función h($\sigma$). Así, la nueva función objetivo propuesta en [1] está dada por:

$$\left|K_\eta^*\right|_p(x) = \left[\sum_{m=1}^{M} (\eta_m^*)^p(x)\right]^{\frac{1}{p}} \tag{4}$$

donde

$$\eta_m^* = \frac{|S_m|^2}{3h^{\frac{2}{3}}(\sigma_m)} \tag{5}$$

es la función objetivo modificada por el tetraedro m-ésimo.

El comportamiento de $h(\sigma)$ en función del parámetro $\delta$ es tal que, $lím_{\delta\to 0}\ h(\sigma) = \sigma$, $V\sigma \leq 0$. Así, si *int P* ≠ *Ø*, entonces *V*x $\epsilon$ *int P* se tiene $\sigma_m$ (x) > 0, para *m* = 1, 2,…,*M* y, a medida que se vaya eligiendo valores de $\delta$ más pequeños, $h$ ($\sigma_m$) se va pareciendo más a $\sigma_m$, de manera que, la función original y su correspondiente versión modificada son muy próximas en la región factible. Así, en dicha región, $\left|K_\eta^*\right|_p$ converge puntualmente a $\left|K_\eta\right|_p$ cuando $\delta \to 0$. Además, se considera que $V\sigma > 0$, $lím_{\delta\to 0}\ h'(\sigma)=1$ *y* $lím_{\delta\to 0}\ h^{(n)}(\sigma) = 0$, para $n \geq 2$, es fácil comprobar que las derivadas de la función objetivo verifican la misma propiedad de convergencia. Como resultado de estas consideraciones, se puede concluir que las posiciones de *v* que minimizan la función objetivo original y la modificada son casi idénticas cuando el valor de ± es *pequeño*. En realidad, ± se selecciona en función del punto *v* bajo consideración, haciéndolo tan pequeño como sea posible pero de manera que la evaluación del mínimo de la función objetivo modificada no presente ningún problema computacional.

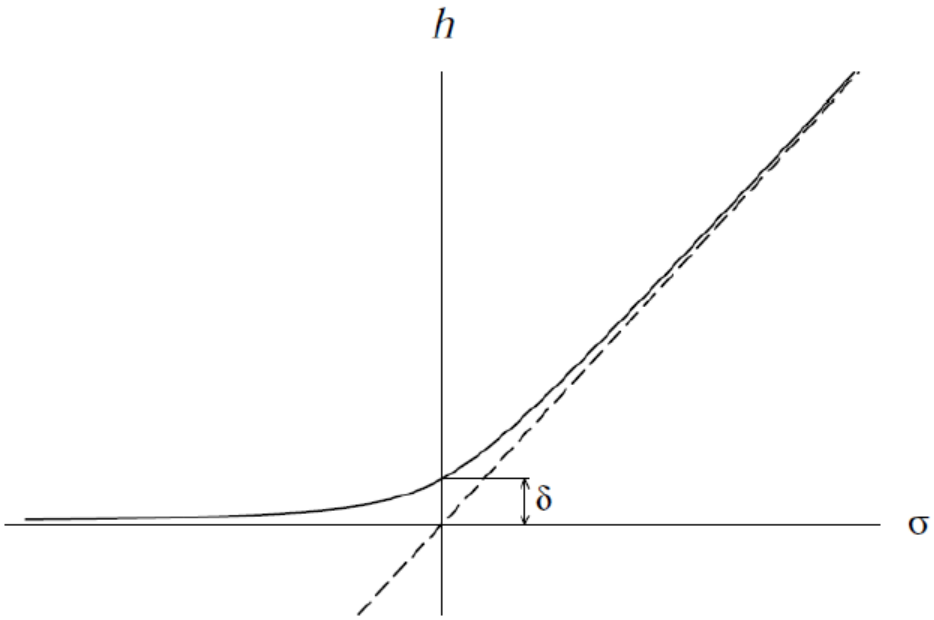


**Figura 11. Representación de la función h(σ)**

Se supone ahora que, *int P* = *Ø* , entonces la función objetivo original, $|K_\eta|_p$ no es adecuada para este propósito ya que no está correctamente definida. Sin embargo, la función objetivo modificada está correctamente definida y tiende a resolver el enredo. Se puede razonar esto desde un punto de vista cualitativo considerando que los términos dominantes en $|K_\eta^*|_p$ son aquellos que están asociados a tetraedros con valores de σ más negativos y, por ello, la minimización de estos términos implica el incremento de estos valores. Se debe remarcar que *h* (σ) es una función creciente y $|K_\eta^*|_p$ tiende a ∞ cuando el volumen de cualquier tetraedro de *N* (*v*) tiende a - ∞, ya que $lím_{\delta\to\infty} h$ (σ) = 0.

En resumen, mediante la función objetivo modificada, se puede desenredar la malla a la vez que se mejora su calidad. Obviamente, esta modificación puede aplicarse fácilmente a otras funciones objetivo del mismo tipo.

Para comprender mejor el comportamiento de la función objetivo y su modificación, se propone el siguiente problema ejemplo. Se considera una malla en 2-D formada por tres triángulos, *vBC*, *vCA* y *vAB*, donde se ha fijado *A*(0,-1), *B*($3^{1/2}$, 0), *C*(0, 1), y *v*(*x*, *y*) es el nodo libre. En este caso, la región factible es el interior del triángulo equilátero *ABC*. En la Figura 12 (a) se muestra $|K_\eta|_2$ (línea continua) y $|K_\eta^*|_2$ (línea discontinua) como una función de *x* para un valor fijo *y* = 0 (la coordenada *y* de la solución óptima). El valor elegido de δ es 0,1. Se puede ver que la función original presenta múltiples mínimos locales y discontinuidades, al contrario de lo que le ocurre a la función modificada. Además, la función objetivo original alcanza su mínimo absoluto fuera de la región factible. Las asíntotas verticales en la función objetivo original corresponden a posiciones del nodo libre para las que $\sigma = 0$ para alguno de los tetraedros de la malla local. Como cabría esperar en este ejemplo, la solución óptima para la función modificada es *v*($3^{1/2}/3$; 0).

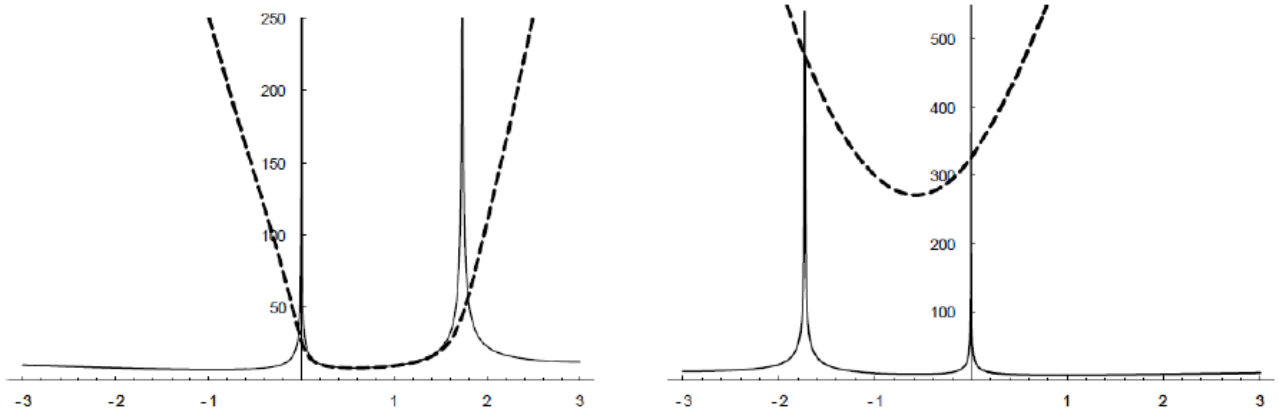


**Figura 12. (a) Corte transversal de $|K_\eta|_2$ (línea continua) y $|K_\eta^*|_2$ (línea discontinua) para el ejemplo 2-D; (b) la misma función objetivo para la malla enredada.**

Se considera ahora la malla enredada obtenida al cambiar la posición del punto $B(3^{1/2}, 0)$ por $B´(-3^{1/2}, 0)$. Aquí, la malla está constituida por los triángulos *vB´C*, *vCA* y *vAB´*, donde *vB´C* y *vAB´* están invertidos. En esta nueva situación no existe región factible. Las gráficas de las funciones $|K_\eta|_2$ y $|K_\eta^*|_2$ están representadas en la Figura 12 (b). Si bien la malla no puede ser desenredada, se obtiene $v(-3^{1/2}/3, 0)$ como posición óptima del nodo libre utilizando la función objetivo. Para esta posición los tres triángulos están "igualmente invertidos" (tienen iguales valores de $\sigma$). En este ejemplo se habrían obtenido los mismos resultados si se hubiera maximizado el mínimo valor de $\sigma$ en la malla.

## Aplicaciones

Para analizar la eficiencia de estas técnicas se va a considerar, en primer lugar, una malla regular de un cubo de lado unidad con 750 tetraedros y 216 nodos uniformemente distribuidos con una valencia máxima de 16. Con objeto de conseguir una malla enredada, transformamos este cubo unitario en otro mayor (10 x 10 *x* 10) cambiando las coordenadas de algunos nodos pero preservando todas sus conectividades. Los nodos interiores permanecen en las posiciones originales, los nodos situados sobre las aristas del cubo unitario son recolocados sobre las aristas del cubo transformado y, finalmente, los nodos interiores de cada cara del cubo original son proyectados sobre las correspondientes caras del cubo transformado. La malla inicial enredada, mostrada en la Figura 13 (a), tiene 10 tetraedros invertidos y una calidad media *qavg* = 0,384 (la calidad media de la malla regular inicial era 0,749). Además, aproximadamente el 50% tiene una calidad muy pobre (menor que 0,04). La medida de calidad elegida aquí, $q = 3 / |S_m||S_m^{-1}|$, para los tetraedros válidos y $q = 0$ para los invertidos. Los resultados después de veinte pasadas del proceso de optimización con $|K_\eta^*|_2$ se muestran en Figura 13 (b). En la Figura 14 presentamos el promedio de calidad, *qavg*, y la calidad mínima, *qmin*, en función del número de iteraciones del proceso de optimización. Se puede apreciar que la calidad promedio decrece inicialmente debido a que el número de tetraedros invertidos aumenta en las primeras iteraciones. La malla tiene 22 tetraedros invertidos después de la primera iteración, 33 después de la segunda, 16 después de la tercera, 11 después de la cuarta y 0 después de la quinta.

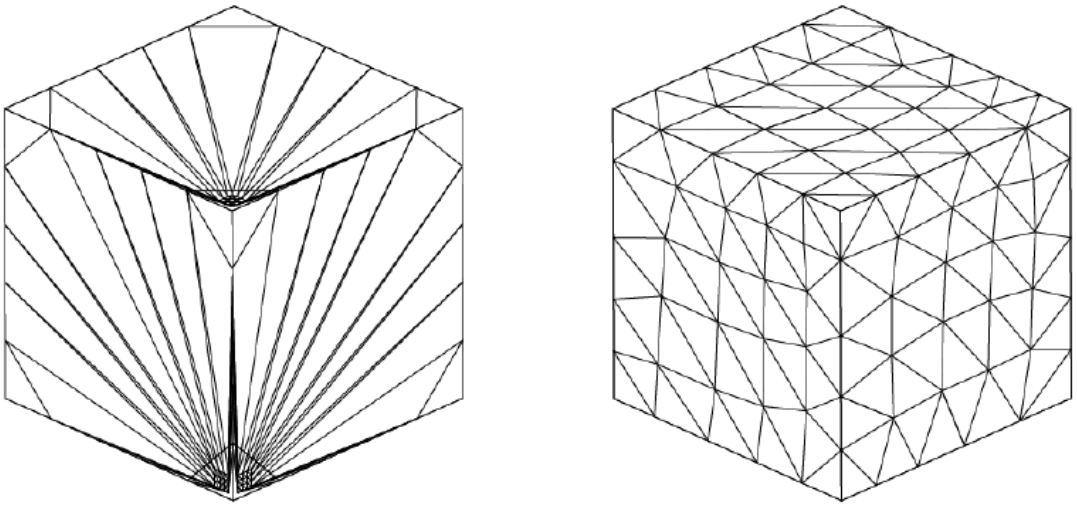

**Figura 13. (a) Malla enredada; (b) Malla desenredada después del proceso del S.U.S. code.**

Estas técnicas de optimización han sido utilizadas también para construir mallas en 3-D adaptadas a superficies complicadas, como las definidas por terrenos irregulares. En la malla resultante puede haber, ocasionalmente, elementos con calidades muy bajas o incluso elementos invertidos, haciendo necesario aplicar algún procedimiento de desenredo y suavizado. Como aplicación del generador de malla y del proceso de optimización se considera el problema test del "volcán" mostrado en la Figura 15. En este caso la malla tiene 20038 tetraedros y 4013 nodos, con una valencia máxima de 31. La malla inicial enredada tiene 576 tetraedros invertidos y una calidad media de *qavg* = 0,529. La distribución de nodos inicial es modificada durante el proceso de optimización de manera que todos los elementos invertidos desaparecen en la cuarta iteración y la calidad media se incrementa hasta *qavg* = 0,615 en la sexta iteración. Este proceso se completa en tan sólo unos pocos segundos de tiempo de CPU en un procesador Intel Xeon. El algoritmo de optimización empleado para minimizar las funciones objetivo fue la variante Levenberg Marquardt del método Newton.

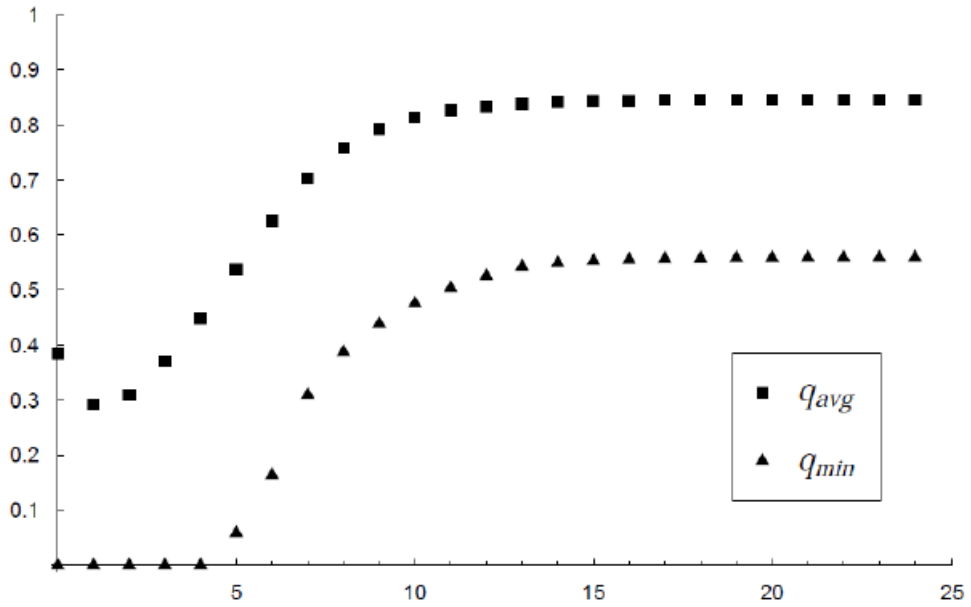


**Figura 14. Valores de calidad promedio qavg y la calidad mínima qmin en función del número del número de iteraciones del proceso de optimización para el problema del cubo.**

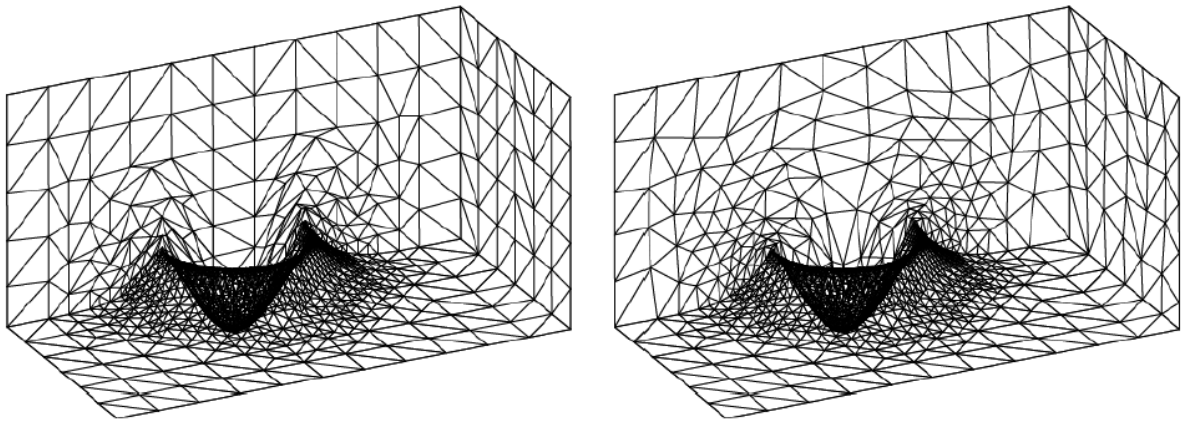

**Figura 15. Problema test del "Volcán", (a) malla inicial con 576 tetraedros invertidos; (b) malla resultante después de seis iteraciones del proceso de optimización.**

### 1.5 CUDA

La programación paralela es una técnica en la que muchas instrucciones se ejecutan simultáneamente. Se basa en el principio de que los problemas grandes se pueden dividir en partes más pequeñas (Divide y Vencerás) que pueden resolverse de forma concurrente ("en paralelo"). Existen varios tipos de programación paralela: paralelismo a nivel de bit, paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tareas.

Durante muchos años, la programación paralela se ha aplicado en la programación de altas prestaciones (HPC), pero el interés en ella ha aumentado en los últimos años debido a las restricciones físicas que impiden el escalado en frecuencia. La programación paralela se ha convertido en el paradigma dominante en la arquitectura de computadores, principalmente en los procesadores multinúcleo. Sin embargo, recientemente, el consumo de energía de los ordenadores paralelos se ha convertido en una preocupación.

Los programas de ordenador paralelos son más difíciles de escribir que los secuenciales porque la concurrencia introduce nuevos tipos de errores de software, siendo las condiciones de carrera las más comunes. La comunicación y la sincronización entre las diferentes subtareas son típicamente las grandes barreras para conseguir un buen rendimiento de los programas paralelos. El incremento de velocidad que consigue un programa como resultado de la paralelización viene dado por la ley de Amdahl.

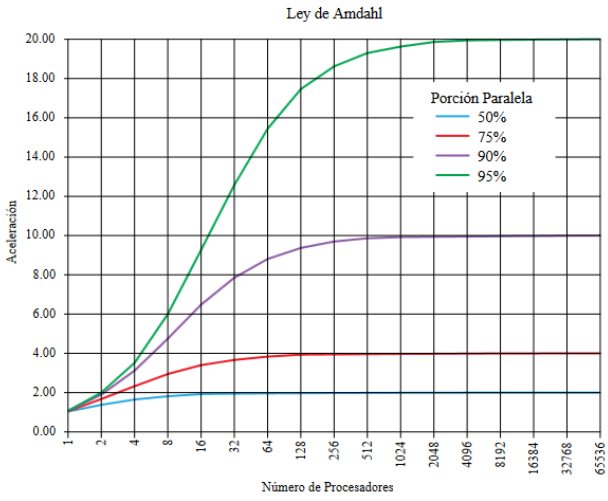


**Figura 16. Ley de Amdahl.**

> "La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente"
>
> Gene Amdahl

El software se ha orientado tradicionalmente hacia la programación en serie. Para resolver un problema, se construye un algoritmo y se implementa en un flujo de instrucciones secuenciales. Estas instrucciones se ejecutan en la unidad central de procesamiento de un ordenador. En el momento en el que una instrucción se termina, se ejecuta la siguiente.

La programación paralela emplea elementos de procesamiento múltiple simultáneamente para resolver un problema. Esto se logra dividiendo el problema en partes independientes de tal manera que cada elemento de procesamiento pueda ejecutar su parte del algoritmo a la vez que los demás. Los elementos de procesamiento pueden ser diversos e incluir recursos tales como un único ordenador con muchos procesadores, varios ordenadores en red, hardware especializado o una combinación de los anteriores.

La tecnología CUDA es una arquitectura de computación paralela desarrollada por uno de los mayores fabricantes de tarjetas gráficas del mercado, *Nvidia*. CUDA son las iniciales de Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo).

Fue introducida en Noviembre de 2006, y se basa en la utilización de un elevado número de nodos de procesamiento para realizar operaciones sobre un gran volumen de datos en paralelo, consiguiendo reducir el tiempo de procesado y obteniendo altas prestaciones.

Uno de los principales objetivos de esta tecnología es resolver problemas complejos que lleven asociados una alta carga computacional sobre la CPU de la forma más eficiente, haciendo uso de las GPUs (Unidades de procesamiento gráficas) incorporadas en la tarjeta gráfica. Para ello se realiza un procesado masivo de los datos, consiguiendo reducir considerablemente los tiempos de ejecución de la aplicación.

Para desarrollar aplicaciones basadas en esta tecnología se pueden utilizar entornos de desarrollo basados en lenguajes de alto nivel como C, uno de los lenguajes de programación más utilizados actualmente, aunque también se pueden utilizar otros lenguajes como C++, Fortran, Java, etc, tal como se puede apreciar en la Figura 17.

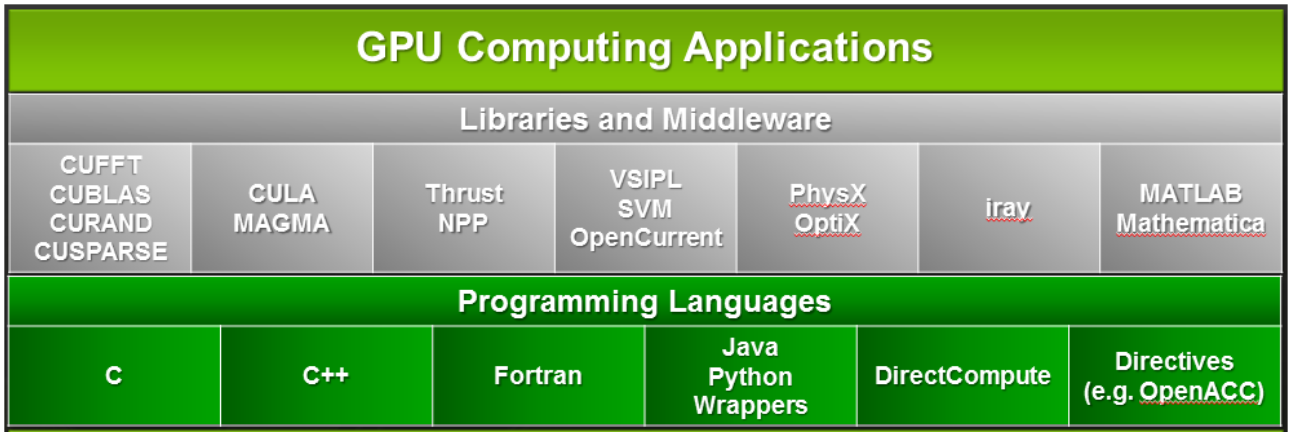


**Figura 17. Aplicaciones computacionales de CUDA.**

El proceso que se sigue consiste en desarrollar aplicaciones basadas en un lenguaje de alto nivel, para posteriormente pasar a un nivel más bajo mediante la inclusión de una serie de extensiones o instrucciones específicas sobre el código original que permitan paralelizar la aplicación. De esta forma se consigue que parte del código sea ejecutado por los multiprocesadores incorporados en la tarjeta gráfica en vez de por la CPU, con el consiguiente ahorro en tiempo de procesado.

La parte del código que será ejecutada por los multiprocesadores se hará mediante paralelización, de forma que una misma operación será ejecutada de forma simultánea sobre varios datos de entrada, (se conoce como arquitectura SIMD), mientras que el resto de la aplicación será ejecutada de forma secuencial por la CPU.

El beneficio que se obtiene al utilizar esta tecnología se puede comprobar en la Figura 18, donde se comparan las máximas prestaciones proporcionadas por diferentes modelos de tarjetas gráficas *Nvidia*, frente a las obtenidas en diferentes modelos de CPUs.

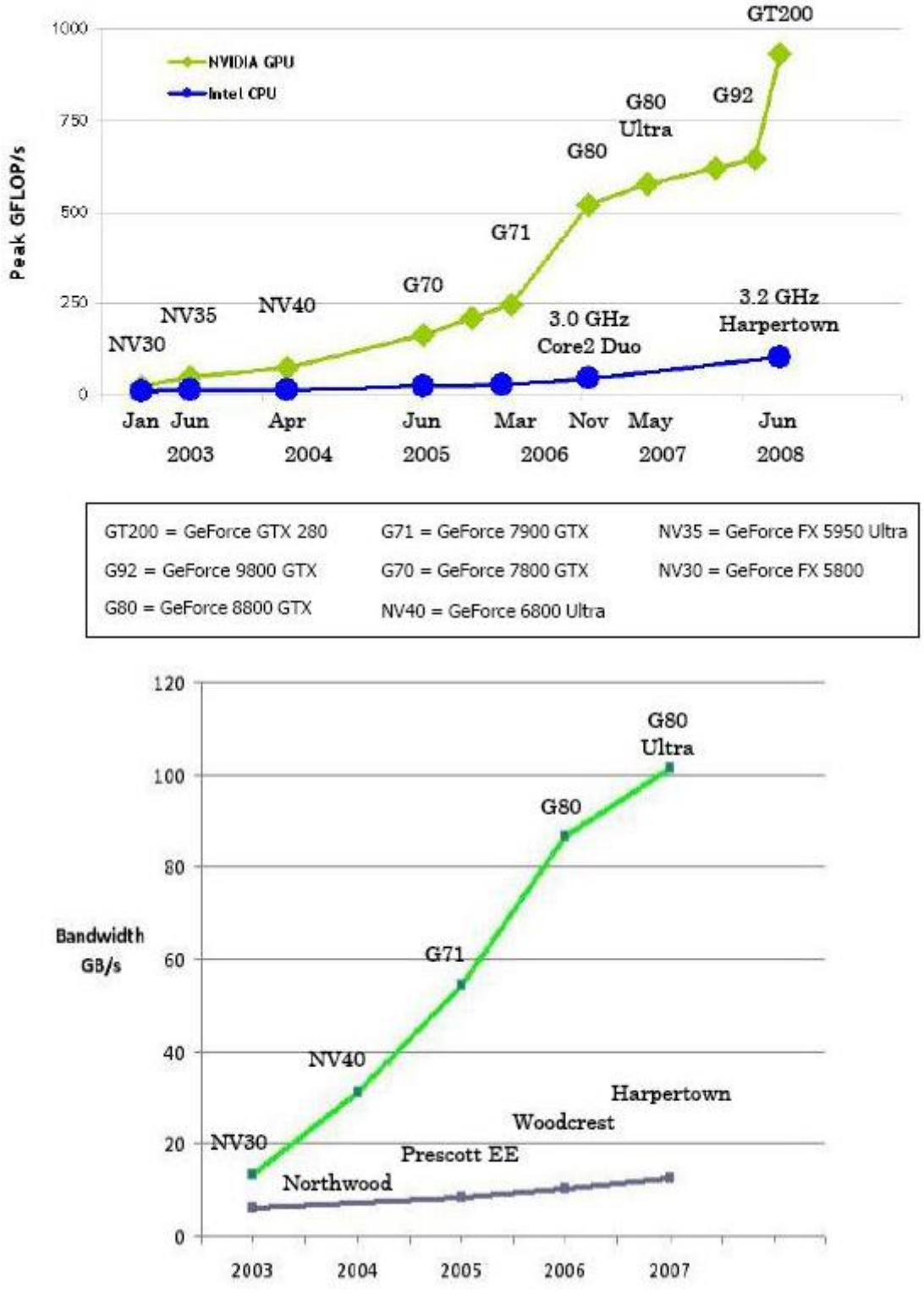


**Figura 18. Comparativa de un benchmark entre CPUs y GPUs.**

Como se puede observar, la tecnología CUDA permite procesar un mayor número de datos por unidad de tiempo en la GPU que en una CPU convencional, así como obtener un ancho de banda considerablemente superior. Las prestaciones proporcionadas por la GPU dependerán del modelo de tarjeta gráfica que se utilice, y como vemos, éstas han aumentado considerablemente en los últimos años, a diferencia de las CPUs, donde la evolución ha sido menor.

Para abordar la tecnología de manera ordenada, concreta y concisa, explicaremos en primer lugar la arquitectura de algunas GPUs compatibles con CUDA, y así entender posteriormente cómo usar dicho potencial explicando la parte de Software.

**Hardware**

La principal diferencia que existe entre una CPU y una GPU, la podemos observar en los siguientes esquemas básicos de ambos dispositivos.


Figura 19. Esquemas básicos de CPU y GPU.

Por un lado, podemos observar que un hardware de tipo GPU consta de un mayor número de núcleos de procesamiento (cores), pero a su vez dispone de menos dispositivos de control y de espacio de memoria de cache. Por otro lado, la CPU está constituida por menos núcleos, que trabajan a mayor frecuencia, y disponen de más espacio de memoria cache y de dispositivos de control.

La línea que sigue el esquema de un hardware de tipo GPU comparado con uno de CPU, es que ofrece mayor potencial para software de mucho cómputo. Este tipo de dispositivo resulta idóneo para aplicaciones de cómputo sencillo pero con gran volumen de datos; por ejemplo, imágenes, vectores, matrices, etc.

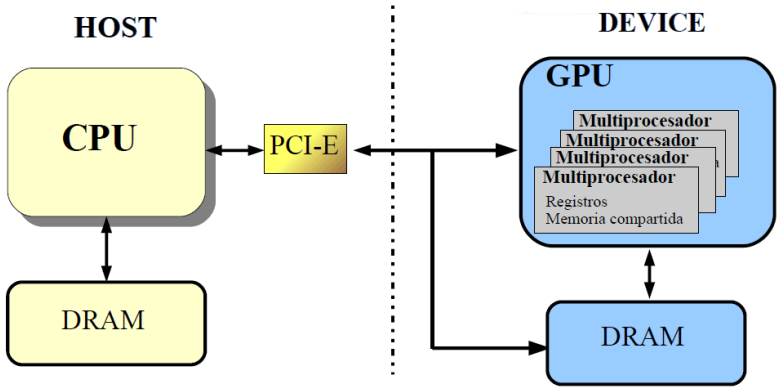


Figura 20. Conectividad entre CPU y GPU.

Tanto la CPU como la GPU están interconectadas mediante un bus PCI (Express normalmente) que proporciona entre 500 MB y 1 GB por segundo (dependiendo si es PCIe 2.0 ó una PCIe 3.0, respectivamente) de velocidad de transferencia de datos.

Adentrándonos cada vez más en los detalles que existen dentro de la GPU, se empezará por los **multiprocesadores**, también nombrados como SM (del inglés, 'Streaming Multiprocessor') en algunas bibliografías.

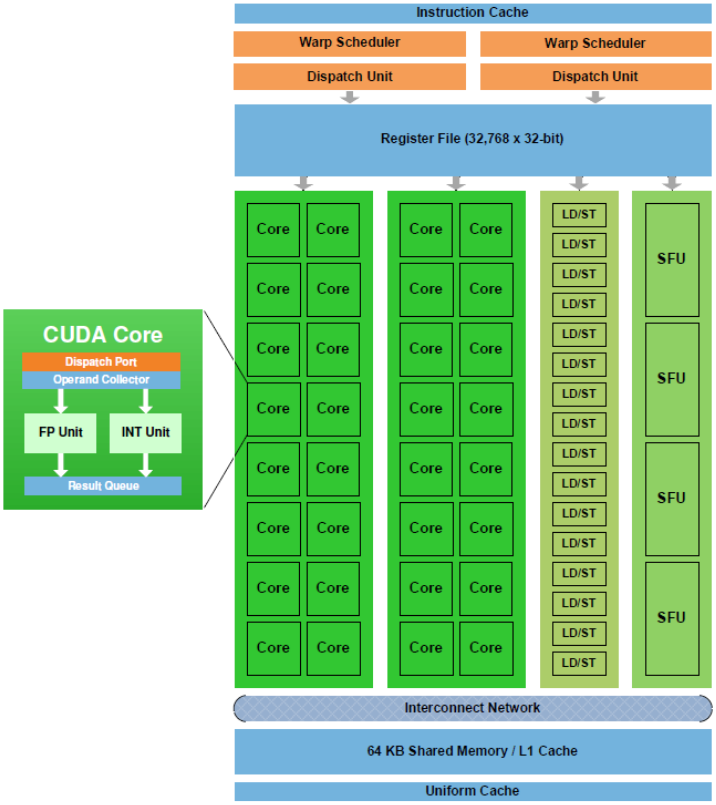


Figura 21. Estructura de un multiprocesador.

Cada multiprocesador consta de una serie de núcleos de procesamiento, unidades de carga/almacenamiento de datos y unidades de funciones especiales (SFU); dependiendo de la arquitectura de la tarjeta gráfica y del modelo de la misma dispondrán de más o menos número de recursos hardware. La Figura 21 corresponde con un multiprocesador propio de una GPU de arquitectura Fermi, la cuál será caso de estudio durante este documento; puesto que se trabajará y se realizará análisis sobre GPUs con dicha arquitectura.

En el caso del SM de la Figura 21 consta de 32 núcleos. Cómo se puede observar en dicha figura, cada core tiene, tanto una unidad para datos enteros, como una unidad de punto flotante. Por otro lado, el SM dispone a su vez de 16 unidades de carga/almacenamiento de datos y 4 unidades de funciones especiales (SFU).

También se puede observar en la parte inferior del esquema, se dispone una memoria cache de corta latencia compartida por todos los cores. Esta cache de datos de nivel 1, de 64 KB configurables, está interconectada con todos los núcleos para reducir los tiempos de acceso a datos que no están en los registros. Los registros, que se encuentran agrupados en la parte superior del esquema, son repartidos a su vez por todos los cores. En el caso de la arquitectura Fermi se dispone de 32.768 registros de 32 bits cada uno.

Por otro lado, en la parte superior de la Figura 21, a parte de la memoria cache de datos disponemos de otra memoria para las instrucciones, la cual se conecta con dos planificadores de **warps**. Un **warp** es la unidad mínima de planificación y ésta está formada por 32 hilos (threads). En un multiprocesador solo puede haber un warp (conjunto de 32 hilos) en ejecución. Los warps son lanzados cuando están listos todos sus operandos; cuando en un determinado momento se necesita un dato y hay que esperar a cargar dicho dato; se marca el warp como inactivo temporalmente y se lanza otro. Cuando esto sucede, la memoria no solo se encarga de traer ese dato que falta, sino varios datos adjuntos en memoria para que, cuando se vuelva a activar el warp, todos los hilos tengan alojados en memoria sus datos correspondientes; puesto que cada hilo dentro del warp ejecuta la misma instrucción en diferentes cores sobre distintos datos.

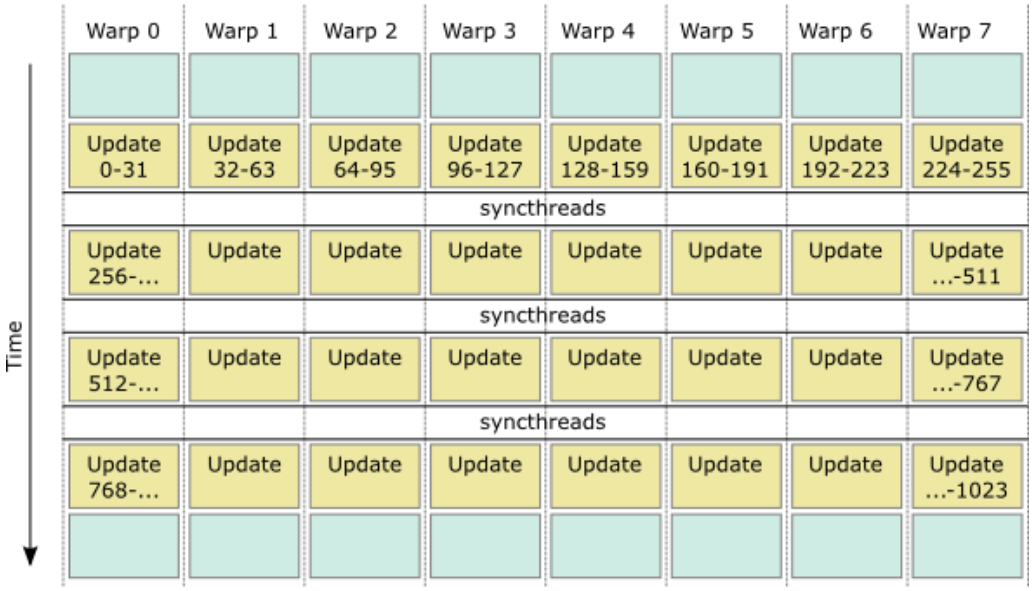


**Figura 22. Agrupamiento de hilos en warps.**

Volviendo a la Figura 21, estos planificadores de warps agrupan las instrucciones provenientes de la cache y son los que se encargan de asignar las instrucciones para los diferentes datos a los distintos cores. Esta planificación es cedida a las unidades de lanzamiento, que 'lanzará' un conjunto de 32 hilos (un warp) a un grupo de 16 cores, una vez se tengan todos sus operandos listos.

El motivo principal de que haya dos unidades de lanzamiento y dos planificadores de warps es porque en la arquitectura Fermi los cores se agrupan en grupos de 16 núcleos. Es por esto que hay tarjetas gráficas con multiprocesadores de 48 núcleos; es decir, 3 grupos de 16 cores.

Al inicio de este capítulo, se comentó que existen varios tipos de programación paralela: paralelismo a nivel de bit, paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tareas. CUDA implementa el paralelismo a nivel de datos mediante su arquitectura.

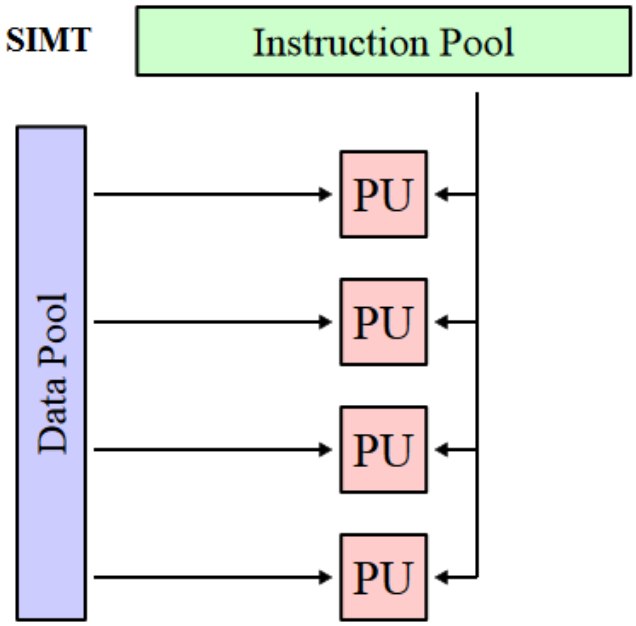


Figura 23. Estructura SIMT.

El paralelismo a nivel de instrucciones se consigue mediante, lo que Nvidia denomina SIMT, Single Instruction, Multiple Thread (Una Instrucción Múltiples Hilos). Esta técnica, muy similar a la técnica SIMD (Single Instruction, Multiple Data) permite en un grupo de 32 hilos (un **warp**), en un determinado instante de tiempo, ejecutar la misma instrucción en un grupo de 16 cores de los que dispone el multiprocesador, pero procesando datos distintos en 2 ciclos de reloj. En un primer ciclo se computan 16 hilos del conjunto total de los 32, y el resto se realiza en el segundo ciclo de reloj.

### Diferentes Arquitecturas

Dentro de las GPUs de Nvidia podemos encontrar diferentes tipos de arquitecturas, como por ejemplo:

- Serie 8 (G80)
- Serie 200 (GT200)
- Kepler
- Fermi

### Serie 8 (G80)

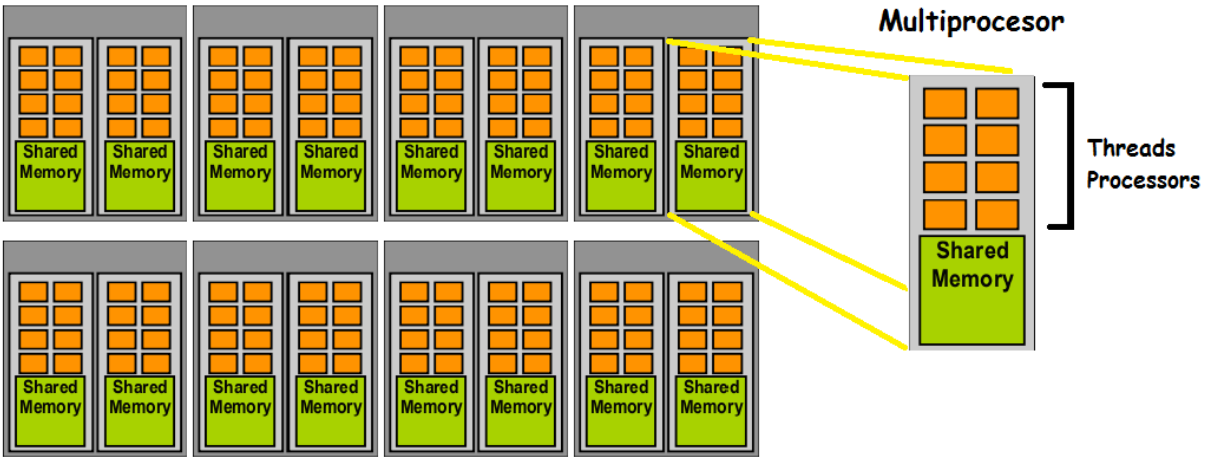


Figura 24. Arquitectura G80.

Se puede ver en la Figura anterior, que aquellas GPUs con arquitectura serie 8 (G80) constan de

varios multiprocesadores (16 en la Figura 24), donde cada uno de ellos contiene 8 cores y una memoria cache de nivel 1 compartida de 16 KB. Esta memoria se comparte entre todos los cores de un mismo multiprocesador. En resumen, una tarjeta gráfica con este tipo de arquitectura puede albergar hasta un máximo de 128 procesadores.

**Serie 200 (GT200)**


Figura 25. Estructura GT200.

Los procesadores gráficos con arquitectura serie 200 (GT200), están formados por varios multiprocesadores (30 en la Figura 25) de 8 cores cada uno. Cada multiprocesador, aparte de los 8 procesadores de cómputo simple, consta de una unidad de procesamiento de doble precisión, además de una memoria compartida de 16 KB para todas las unidades funcionales del mismo SM. En resumen, este tipo de arquitectura puede contener hasta 240 procesadores de cómputo simple además de 30 procesadores de doble precisión.

**Fermi**

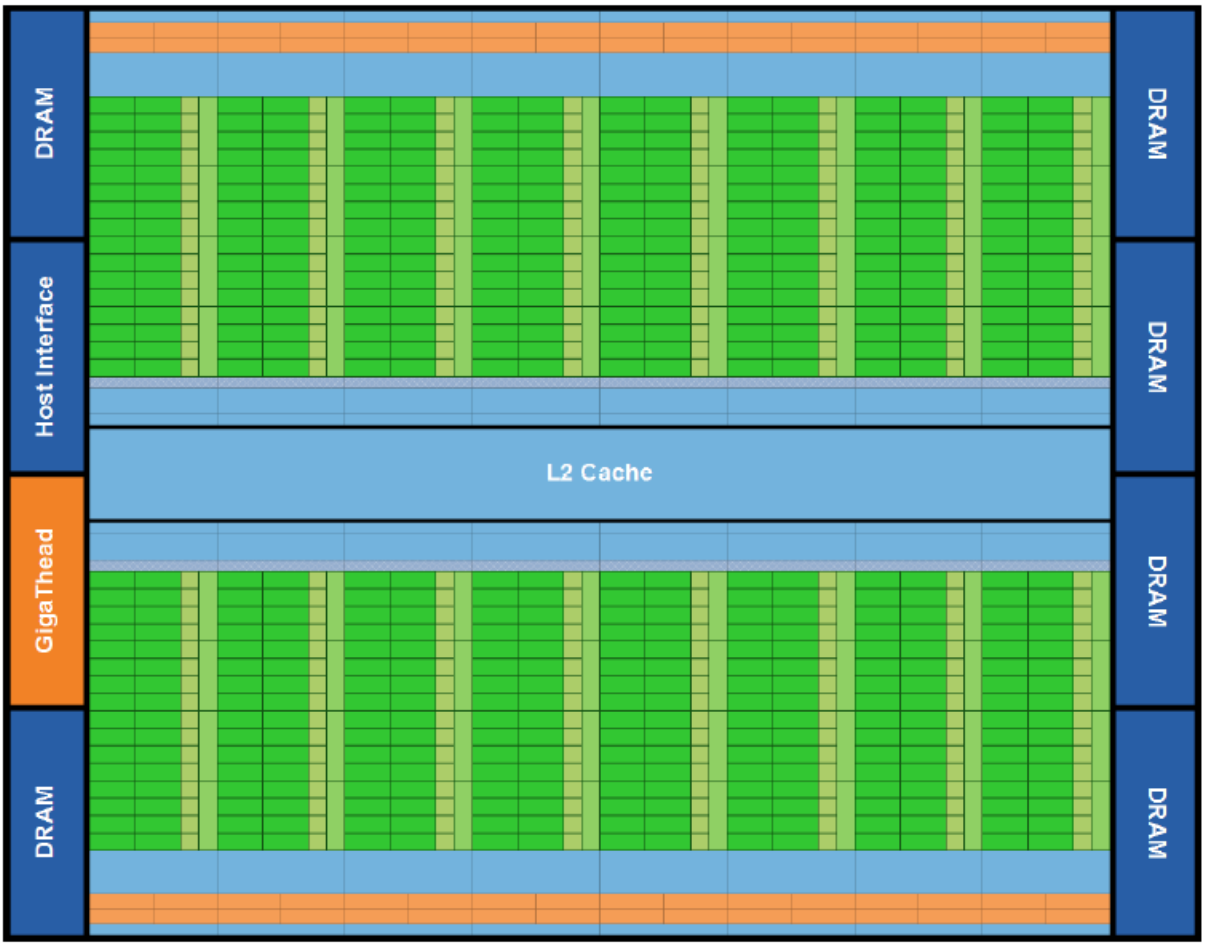


Figura 26. Arquitectura Fermi.

La arquitectura Fermi es una arquitectura más complicada, puesto que los dispositivos con este tipo de esquemas añaden nuevos elementos. Se puede ver como lo que rodea a todo el entramado de cores, son dispositivos de memoria global, así como la interfaz con el bus PCIe, y una unidad 'GigaThead'. Esta nueva unidad es la que se encarga de planificar los bloques de hilos y lanzarlos a ejecutar a los distintos multiprocesadores buscando el mayor aprovechamiento posible de los recursos de los que dispone.

Con respecto a los SM, los dispositivos que siguen esta arquitectura están formados por varios multiprocesadores (en este caso 16), donde cualquiera de ellos consta de 16 unidades de procesamiento en doble precisión y 32 procesadores simples. Estos 32 cores están agrupados en grupos de 16. Cada multiprocesador tiene una memoria cache de nivel 1, compartida por todos los cores del mismo SM, de unos 48 a 64 KB; dependiendo del modelo de la GPU. Otra novedad que ofrece este tipo de arquitectura es la aparición de una memoria cache de nivel 2 compartida por todos los multiprocesadores, de unos 768 KB de capacidad. Con la aparición de esta nueva memoria se reducen los tiempos de acceso a memoria global DRAM. En resumen, una tarjeta gráfica con este tipo de arquitectura puede albergar hasta un máximo de 512 procesadores simples y 256 unidades de procesamiento en doble precisión.

**Kepler**


Figura 27. Arquitectura Kepler.

Esta arquitectura es la sucesora de la arquitectura Fermi. Este tipo de dispositivos, constan de multiprocesadores (8 en este caso) con 32 unidades de procesamiento en doble precisión y 192 procesadores simples agrupados en grupos de 32. Cada multiprocesador dispone de memoria cache de nivel 1 de 48 a 64 KB de capacidad, dependiendo del modelo. Por lo tanto cada GPU consta de una memoria cache de nivel 2 compartida por todos los multiprocesadores de 1536 KB de capacidad. En resumen, este tipo de arquitectura puede contener hasta 1536 procesadores de cómputo simple además de 256 procesadores de doble precisión.

**Software**

A la hora de desarrollar una aplicación con la tecnología CUDA hay que tener muy claro una serie de conceptos característicos de esta tecnología. A continuación se describirán algunos de los aspectos más importantes.

Tres conceptos muy utilizados al diseñar aplicaciones mediante paralelización son: hilo, bloque de hilos y grid.

- **Hilo** (**thread**): Un hilo es una secuencia de instrucciones que se ejecutan en un procesador para unos datos determinados.
- **Bloque de hilos** (**threads block**): Representa un conjunto de hilos que son lanzados sobre el mismo multiprocesador de la tarjeta gráfica. Pueden tener comunicación entre ellos, ya que disponen de un espacio de memoria cache compartida de nivel 1.
- **Grid**: Representa todo el conjunto de bloques de hilos. Los diferentes bloques de hilos se pueden comunicar entre sí por medio del espacio de memoria global o de la memoria cache compartida de nivel 2, si la tarjeta donde se ejecuta dispone de ella.

Estos tres términos se pueden ver gráficamente en la Figura 28, donde se puede comprobar cómo un hilo forma parte de un bloque de hilos, y a su vez un bloque de hilos pertenece a un grid.

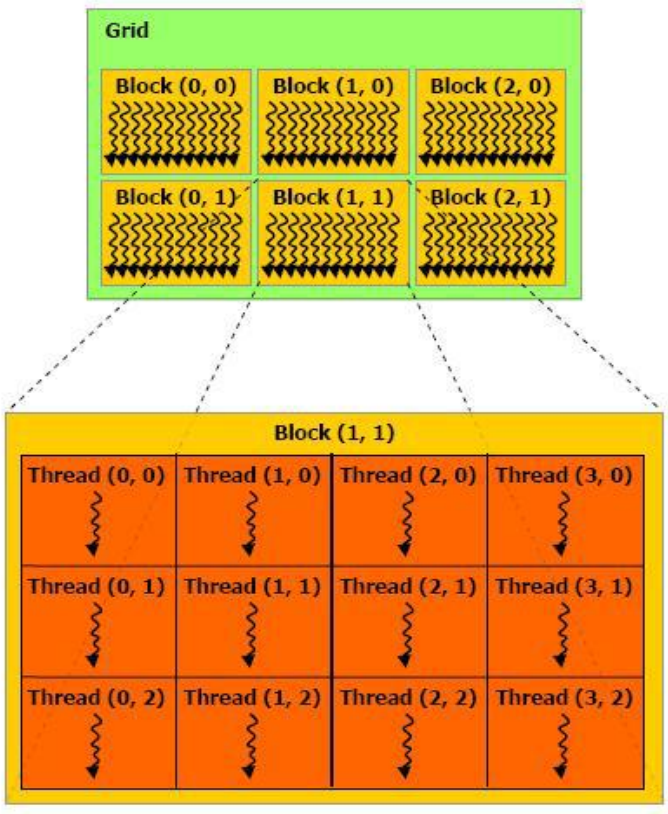


Figura 28. Estructura software de un CUDA kernel.

Otro término muy importante y novedoso de esta tecnología es el concepto de **kernel**. Un **kernel** es una función desarrollada en lenguaje C que será ejecutada de forma simultánea por varios procesadores de los multiprocesadores (si dispone de más de uno) de la tarjeta gráfica. Por lo tanto, va a contener el código que será ejecutado en paralelo por diferentes hilos.

Un kernel se define normalmente con el calificativo '**__global__**', para que éste pueda ser lanzado desde la CPU. Veamos un pequeño ejemplo, de suma de una constante a los elementos de un vector, para que se pueda apreciar la diferencia de filosofía a la hora de programar:

```
# Lenguaje C

for(int i = 0; i < n; i++){
   vector[i] += CONSTANTE;
}
```

```
# CUDA

__global__ void suma (int *vector, int dimension){

   int idx = blockIdx.x * blockDim.x + threadIdx.x;

   if( idx < dimension )
      vector[idx] += CONSTANTE;
}

int main(){
 …
   suma <<< ( dim / 32) , 32 >>> (d_vector, dim);
 …
}
```

Cuando se ejecuta un kernel hay que indicar el número de bloques de hilos y el número de hilos por bloque en los que se va a ejecutar la función, teniendo en cuenta las limitaciones de cada modelo de tarjeta gráfica. Estos parámetros se indican en la propia llamada a la función mediante dos variables, como se puede ver en el siguiente ejemplo:

```
suma <<< dimGrid, dimBlock >>> (d_vector, dimension);
```

En este caso 'dimGrid' indica el número de bloques de hilos, y 'dimBlock' el número de hilos por bloque.

Cada hilo y cada bloque de hilos tienen asociados una información, que permiten identificarlos dentro del kernel, estos identificadores quedan representados por las variables blockDim.x, blockIdx.x y threadIdx.x, las cuales son accesibles desde el kernel. Estos identificadores a parte de la dimensión ‘x’, también disponen de coordenadas ‘y’ y ‘z’.

En la Figura 29 se puede ver la configuración del grid si lanzamos un kernel como el siguiente:

```
suma <<< 4, 8 >>> (d_vector, dimension);
```

Se está lanzando 4 bloques de 8 hilos cada uno (nótese que se está trabajando en una sola dimensión), por lo tanto lo que llega a la GPU es algo tal que así:

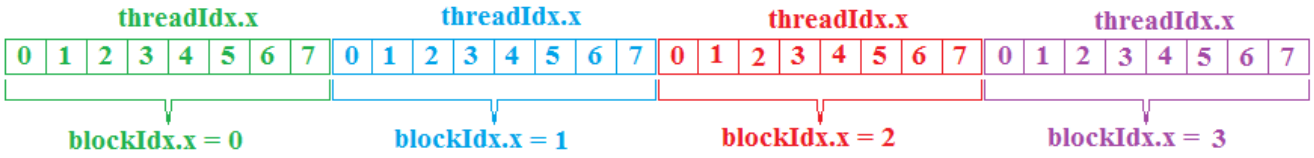


Figura 29. Organización del grid en un kernel.

Cada bloque de 8 hilos será asignado a cada multiprocesador, donde se asignará a cada núcleo un hilo, para que ejecuten en paralelo la misma instrucción en un determinado instante de tiempo. Si el multiprocesador dispone de más cores, éstos estarán ociosos. Para que cada hilo sepa cuál es su identificador a nivel global; es decir, su identificación a nivel del grid se hacen uso de los identificadores de hilos dentro de un bloque y de los identificadores de bloques dentro del grid. De este modo, el hilo, sabiendo cuál es su identificador, sabe qué datos debe procesar. Se suele usar el siguiente cálculo para los casos de bloques de hilos y grids monodimensionales:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Para obtener el máximo rendimiento de la tecnología CUDA hay que intentar ejecutar el mayor número de hilos en cada uno de los multiprocesadores de la tarjeta gráfica, de forma que todos los núcleos de los SMs estén ocupados realizando operaciones, y por lo tanto aprovechando al máximo sus recursos.

Como ya se mencionó anteriormente, para desarrollar una aplicación con CUDA se suele utilizar un entorno de desarrollo basado en lenguaje C, al cual se le añaden una serie de extensiones específicas de esta tecnología. A continuación se comentará algunas de las más importantes.

En CUDA hay una serie de calificativos para las funciones que permiten indicar si sería ejecutada por la CPU o por la GPU, y desde donde puede ser llamada:

- **__device__**: Indica que la función será ejecutada por la tarjeta gráfica, y únicamente puede ser llamada desde la propia tarjeta gráfica.

- **__global__**: Como ya vimos anteriormente, este calificativo se utiliza para definir un kernel, el cual es una función que será ejecutada por la GPU y únicamente puede ser llamada desde la CPU.

- **__host__**: Indica que la función será ejecutada por la CPU, y únicamente puede ser llamada desde la propia CPU. Por lo tanto este calificativo se utiliza para definir una función tradicional de C. Es equivalente a no poner ningún calificativo en la función.

También hay calificativos para las variables que indican donde residirá la variable, así como el tiempo de vida de la misma:

- **__device__**: Se utiliza para declarar una variable que residirá en la memoria global de la tarjeta gráfica. Si no va a acompañado de ningún otro calificativo su tiempo de vida será el mismo que el de la aplicación y será accesible por todos los hilos.

- **__constant__**: Declara una variable que residirá en el espacio de memoria constante de la tarjeta gráfica. Su tiempo de vida será el mismo que el de la aplicación, y será accesible por todos los hilos.

- **__shared__**: Declara una variable que residirá en el espacio de memoria compartida de la tarjeta gráfica (memoria cache de nivel 1). Su tiempo de vida será el mismo que el del bloque de hilos al que pertenece, y solo será accesible por los hilos del bloque de hilos al que pertenece.

Hay que mencionar que estos calificativos no están permitidos al declarar variables pertenecientes a estructuras o uniones.

**Sincronización**

Dentro de un kernel se pueden establecer puntos de sincronización entre todos los hilos de un mismo bloque. Para ello se usa la instrucción:

```
__syncthreads();
```

la cual fija una barrera de forma que cada hilo no continúa con la ejecución del programa hasta que todos los hilos del mismo bloque no hayan llegado al punto de sincronización.

**Manejo de memoria**

Otro aspecto a tener en cuenta cuando se trabaja con la tecnología CUDA es que se dispone de dos espacios totalmente diferenciados. Por un lado todo lo relacionado con la CPU, con su correspondiente espacio de memoria, y por otro lado todo lo relacionado con la tarjeta gráfica, también con su correspondiente espacio de memoria.

Esto da lugar a que en toda aplicación desarrollada con esta tecnología, parte de ella sea ejecutada por la CPU por un único proceso, y otra parte por la GPU de forma simultánea por varios hilos, tal y como se puede ver en la Figura 30:

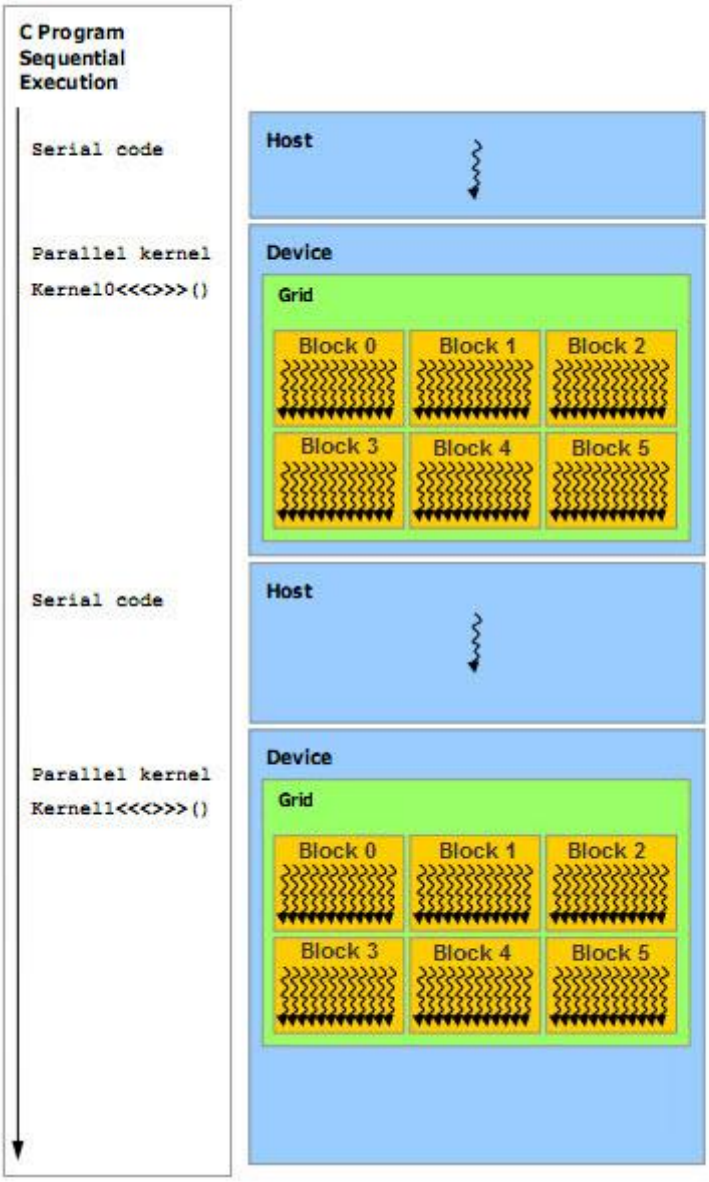


**Figura 30. Traza de una aplicación CUDA.**

Dentro del espacio de la tarjeta gráfica cabe destacar que la memoria define diferentes jerarquías, como se puede ver en la Figura 31:

- Memoria global: Zona de memoria accesible por los hilos de todos los bloques.
- Memoria cache de nivel 2: Zona de memoria accesible por los hilos de todos los bloques. Solamente las tarjetas gráficas de gama alta ofrecen esta jerarquía de memoria.
- Memoria cache de nivel 1: Zona de memoria compartida por todos los hilos que pertenecen a un mismo bloque.
- Registros de un hilo: Zona de memoria exclusiva para un determinado hilo.

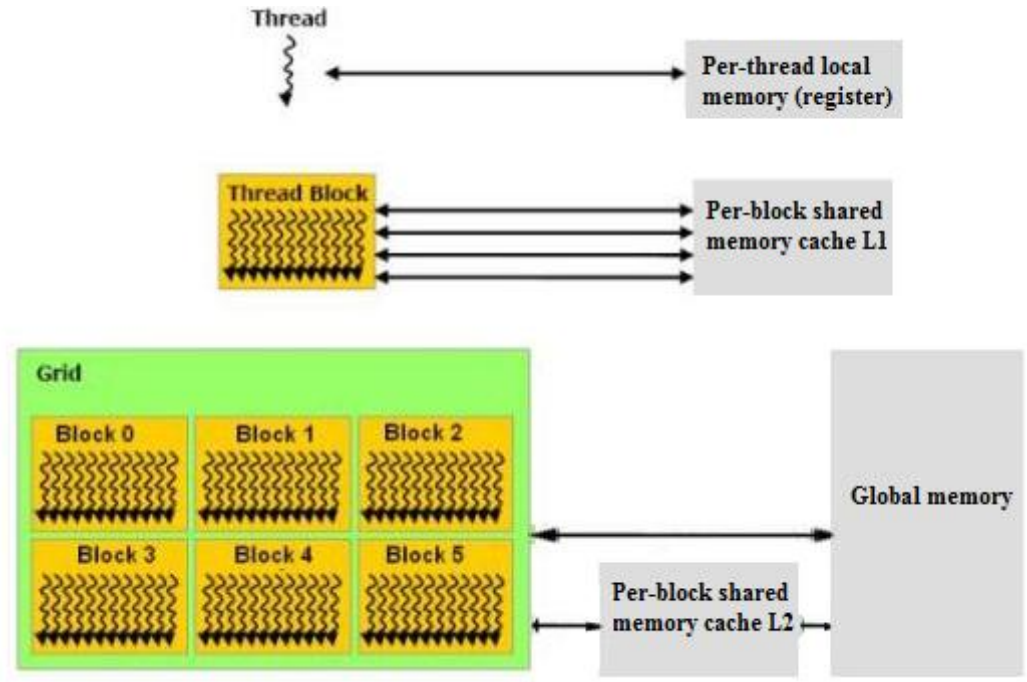


Figura 31. Jerarquía de memoria CUDA.

Hay que resaltar que el espacio de memoria compartida es más rápido que el global para realizar las operaciones, por lo que conviene trabajar sobre este espacio siempre que la aplicación lo permita.

Otro aspecto a tener en cuenta es cómo se lleva a cabo el mantenimiento de memoria en CUDA. Para poder procesar los datos en la tarjeta gráfica es necesario que previamente se hayan transferido a ella desde la CPU. Para ello CUDA provee una serie de funciones que permiten reservar memoria en la tarjeta gráfica, transferir la información desde la CPU a la GPU y viceversa. A continuación se describirá las más importantes.

- **cudaMalloc**: Función que permite reservar un bloque de memoria en la tarjeta gráfica. El formato de la función es:

```
cudaError_t cudaMalloc( void** devPtr, size_t count );
```

  y tiene un funcionamiento igual que su equivalente en C. El bloque de memoria queda reservado en el espacio de memoria global

- **cudaFree**: Función que permite liberar un bloque de memoria en la tarjeta gráfica. El formato de la función es:

```
cudaError_t cudaFree(void* devPtr);
```

  y tiene un funcionamiento igual a su equivalente en C.

- **cudaMemset**: Función que permite inicializar un bloque de memoria de la tarjeta gráfica a un determinado valor. El formato de la función es:

```
cudaError_t cudaMemset( void* devPtr, int value, size_t count );
```

y tiene un funcionamiento igual a su equivalente en C.

- **cudaMemcpy**: Función que permite hacer transferencias de bloques de memoria entre la CPU y la GPU. El formato de la función es:

```
cudaError_t cudaMemcpy( void* dst, const void* src, size_t count,
enum cudaMemcpyKind kind);
```

y tiene un funcionamiento similar a su equivalente en C, salvo que en este caso se dispone del parámetro 'cudaMemcpyKind kind' que indica cómo se va a realizar la transferencia.

Básicamente se suelen utilizar dos modos:

- **cudaMemcpyHostToDevice**: Permite transferir un bloque de memoria de la CPU a la GPU.

- **cudaMemcpyDeviceToHost**: Permite transferir un bloque de memoria de la GPU a la CPU.

Por lo tanto, el esquema seguido en cuanto al manejo de memoria en las tarjetas gráficas en una aplicación paralelizada suele ser el siguiente:

1. Reservar un bloque de memoria en el espacio de memoria global de la tarjeta gráfica para poder copiar los datos de entrada. Para ello se hace uso de la función 'cudaMalloc'.

2. Transferir los datos de entrada de la memoria del host a la memoria global de la GPU. Para ello se utiliza la función 'cudaMemcpy' en modo 'cudaMemcpyHostToDevice'.

3. Una vez que los datos están en la memoria global, se realizan el cómputo sobre éstos.

4. Los datos obtenidos se transfieren de la memoria global de la GPU a la memoria del host. Para ello se hace uso de la función 'cudaMemcpy' en modo 'cudaMemcpyDeviceToHost'.

5. Finalmente se libera el bloque de memoria de la tarjeta gráfica. Para ello se hace uso de la función 'cudaFree'.

**Eventos**

Otro punto importante a destacar dentro de las rutinas que nos ofrece CUDA como interfaz en el lenguaje C es la gestión de eventos. La gestión de eventos nos permite conocer de forma exacta cuanto tiempo se ha consumido en la ejecución de un determinado kernel, por ejemplo. Los eventos de CUDA son un tipo de datos específico con unas rutinas especiales para hacer uso de ellos, a continuación se verán los más importantes:

- **cudaEventCreate**: Función que permite crear un objeto de tipo evento. El formato de la función es:

```
cudaError_t cudaEventCreate( cudaEvent_t* event );
```

como parámetro se le pasa la referencia de una variable cudaEvent_t ya declarada.

- **cudaEventRecord**: Función que permite registrar un determinado evento ya creado. El formato de la función es:

```
cudaError_t   cudaEventRecord(cudaEvent_t   event,   cudaStream_t
stream = 0);
```

como parámetro se le pasa la variable cudaEvent_t ya creada, y en caso opcional podemos asignarle un stream determinado, aunque este por defecto esté a 0. Hay que tener en cuenta que esta función es asíncrona; es decir, que si lanzamos otra función posteriormente a esta puede que se ejecute antes, dependiendo de los recursos que se estén usando en la GPU.

- **cudaEventSynchronize**: Función que permite sincronizar un determinado evento; es decir, si ese evento está pendiente de ejecutarse, se ejecutaría inmediatamente. El formato de la función es:

```
cudaError_t cudaEventSynchronize( cudaEvent_t event );
```

como parámetro se le pasa la variable cudaEvent_t.

- **cudaEventElapsedTime**: Función que permite calcular el tiempo transcurrido entre dos eventos. El formato de la función es:

```
cudaError_t cudaEventElapsedTime( float *ms, cudaEvent_t start,
cudaEvent_t end );
```

como parámetro se le pasa la referencia a una variable float, así como dos eventos: uno de inicio y otro de fin.

- **cudaEventDestroy**: Función que permite destruir un evento. El formato de la función es:

```
cudaError_t cudaEventDestroy ( cudaEvent_t event );
```

como parámetro se le pasa el evento a destruir.

Se mostrará un pequeño ejemplo de su uso:

```
…
cudaEvent_t start, stop;      // se declaran los eventos
cudaEventCreate( &start );    // se crean ambos eventos
cudaEventCreate( &stop );
cudaEventRecord( start, 0 );  // se inicializa el evento de inicio

# lanzamiento de un kernel
suma <<< 1, n >>> (d_vector, dimension);

cudaEventRecord( stop, 0 );   // se inicializa el evento final
cudaEventSynchronize(stop);   // se sincroniza el registro

float elapsedTime;            // se calcula el tiempo (ms)
cudaEventElapsedTime( &elapsedTime, start, stop );

cudaEventDestroy(start);      // se destruyen los eventos
cudaEventDestroy(stop);
…
```

**Compilador**

La compilación de un código CUDA se divide en dos etapas diferenciadas:

- Virtual: Se genera el código PTX (Parallel Thread eXecution). Se trata de un código virtual que se genera por medio de la llamada al compilador '**nvcc**'.

- Física: Se genera el código objeto para la GPU concreta. Esta etapa se genera automáticamente por defecto al realizar la llamada al compilador '**nvcc**' a menos que se especifique la utilización del emulador.

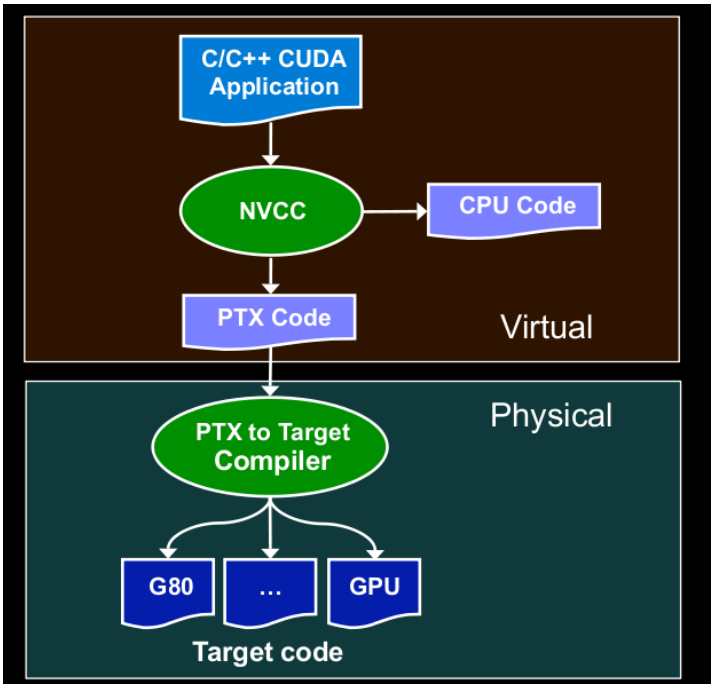


Figura 32. Estructura del compilador CUDA.

El compilador nvcc se utiliza de igual manera que el compilador de C (gcc ó g++); de modo que monitoriza las llamadas a todos los compiladores y herramientas CUDA: cudac, g++, etc. La salida que genera el compilador separa el código de la CPU con el de la GPU; es decir, que es necesario compilar el código C con otra herramienta. Ejemplo de una compilación normal con el nvcc:

```
$ nvcc prueba.cu –o prueba
```

Para el compilador NVCC se le pueden especificar diferentes opciones al compilador a la hora de generar el código objeto:

- **-arch**: Especifica la capacidad de cómputo (compute capability) con la que se compila código C a PTX (conjunto de instrucciones máquinas de CUDA).

```
$ nvcc -o sumavectores sumavectores.cu -arch=sm_13
```

- **-G**: proporciona una herramienta nativa de depuración para CUDA, ejecutando en modo de depuración los códigos en la GPU, con sus limitaciones y comportamiento reales, sin las variaciones introducidas por la emulación.

```
$ nvcc -o sumavectores -g -G sumavectores.cu
```

- **-c**: no se genera binario ejecutable.

```
$ nvcc -c sumavectores.cu
```

Existen otras herramientas importantes, a la hora de analizar un código, sus trazas, los recursos usados, etc. Aquí sólo se han comentado los más usuales.

**Cuda Debugger**

El debugger de CUDA es una herramienta bastante útil para analizar la traza de un programa. Como en el entorno de Linux se compila y se ejecuta todo por medio de la línea de comandos resulta esencial en algunas circunstancias ejecutar el código paso a paso.

Para ello el ToolKit de CUDA consta de un debugger que se puede ejecutar, aunque para ello primero se debe compilar el código con la herramienta de depuración:

```
$ nvcc –g –G prueba.cu –o prueba
```

Una vez compilado, ejecutamos el depurador:

```
$ cuda-gdb prueba
```

Ahora una vez, aquí se pueden:

- Fijar 'breakpoints' a cualquier línea del código que se desee o a cualquier función
- Ejecutar el código paso a paso
- Ver el valor de las variables en cualquier momento mediante 'print'
- Ver los detalles de los threads lanzados a la GPU.
- Etc.

Para la búsqueda de más propiedades de esta herramienta se puede encontrar en la API de Nvidia, siguiendo la guía de usuario:

http://docs.nvidia.com/cuda/cuda-gdb/index.html

**Cuda Profiler**

El profiler es otra herramienta de gran utilidad a la hora de acceder a contadores nativos del hardware si no disponemos de librerías como PAPI.

Para ejecutar el profiler, debemos enlazar la librería para poder tener acceso a las librerías del mismo:

```
$ sudo ln -s `locate libcuinj64.so.5.0.35` /usr/lib/x86_64-linux-
gnu/libcuinj64.so.5.0.35
```

Luego copiamos en el directorio donde tengamos el fichero ejecutable ya compilado que vayamos a analizar, el fichero 'libcuinj64.so.5.0.35' que se encuentra en el directorio:

- PATH-OF-CUDA/lib ó
- PATH-OF-CUDA/lib64

El path de cuda por defecto se encuentra en /usr/local/cuda.

Una vez hecho esto, se puede usar el profiler para conocer datos como por ejemplo:

- El número de registros que se ha usado en el lanzamiento de un kernel
- El número de fallos de cache de nivel 1
- El número de instrucciones ejecutadas
- El número de instrucciones que llegaron a la etapa de Issued
- Etc.

El nombre de los eventos puede variar en función de la máquina y de la versión de CUDA que se tenga instalada. Para conocer todos los eventos que nos puede proporcionar el profiler, se debe de lanzar la siguiente línea de comandos:

```
$ nvprof --query-events
```

Por ejemplo si se desea conocer toda la información referente a las rutinas de CUDA (reserva de memoria, copia de datos, etc.), así como las instrucciones ejecutadas en la GPU (por ejemplo) deberíamos ejecutar:

```
$ nvprof -print-gpu-trace --events inst_executed ./prueba
```

Para más detalles sobre la utilización de esta potente herramienta, se puede encontrar en la API de Nvidia, siguiendo la guía de usuario:

http://docs.nvidia.com/cuda/profiler-users-guide/index.html

**Caso de estudio**

Durante el estudio de la tecnología CUDA, se realizaron varias aplicaciones de prueba como primera toma de contacto con el nuevo entorno, entre ellas, una aplicación de procesamiento a un vector de gran tamaño con elementos de doble precisión.

La aplicación realizaba la siguiente operación para cada uno de los elementos de un vector de más de 67 millones de elementos en doble precisión:

```
for(int i = 0; i < 100; i++)
   arrayDestino[componente] =
           arrayOrigen[componente] +
          (arrayOrigen[componente] * arrayOrigen[componente]);
```

Dentro de la misma aplicación se hicieron diferentes ejecuciones con diferentes herramientas:

1. Se ejecuta de forma secuencial en la CPU.

2. Se ejecuta haciendo uso de la paralelización de OpenMP con 4 hilos.

3. Se ejecuta paralelamente en la GPU.

Los resultados obtenidos en la GPU personal han sido los siguientes:

**Tabla 3. Tabla de tiempos del código del vector en GeForce.**

| Código | Tiempo de cómputo | SpeedUp |
|---|---|---|
| **Secuencial** | 75.8 seg | **1X** |
| **OpenMP** | 87.35 seg | **0.86X** |
| **CUDA** | 2.41 seg | **90.17X** |

NOTA: Estas medidas fueron tomadas con una Nvidia GeForce GT 610.

Analizando el código mediante el profiler en la GPU personal se obtienen los siguientes datos:

**Tabla 4. Tabla del profiling del código del vector en GeForce.**

| | |
|---|---|
| **Número de transacciones de almacenamiento a memoria global** | $419.43 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $4.19 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $0.87 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $0.69 \cdot 10^9$ |
| **Número de instrucciones entre cada fallo de memoria** | 169.68 |

NOTA: Estas medidas fueron tomadas con una Nvidia GeForce GT 610.

A continuación se mostrarán los resultados obtenidos en la GPU de la máquina ‘vector’, del Instituto Universitario SIANI:

**Tabla 5. Tabla de tiempos del vector en Tesla.**

| Código | Tiempo de cómputo | SpeedUp |
|---|---|---|
| **Secuencial** | 53.31 seg | **1X** |
| **OpenMP** | 61.43 seg | **0.87X** |
| **CUDA** | 0.59 seg | **90.36X** |

NOTA: Estas medidas fueron tomadas con una Nvidia Tesla S2050.

**Tabla 6. Tabla del profiling del código del vector en Tesla.**

| | |
|---|---|
| **Número de transacciones de almacenamiento a memoria global** | $423.77 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $4.24 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $1.15 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $0.7 \cdot 10^9$ |
| **Número de instrucciones entre cada fallo de memoria** | 165.09 |

NOTA: Estas medidas fueron tomadas con una Nvidia Tesla S2050.

Se pueden observar que los resultados obtenidos han sido bastante satisfactorios por parte del procesamiento que se ha llevado a cabo en la GPU. Obteniendo hasta un factor de 90 veces más rápida la versión de CUDA con la versión secuencial en CPU. Los datos tomados del profiler se han recogido con el objeto de ser un punto de comparación a la hora de analizar la implementación llevada a cabo en este proyecto fin de carrera.

Nvidia Corporation, fabricante de las tarjetas gráficas con la arquitectura CUDA, recomienda que las aplicaciones que se vayan a procesar en estos dispositivos tengan un ratio de instrucciones por

byte en torno a 3.6 – 4.5. Esto quiere decir, que para aprovechar la gran capacidad de cómputo de CUDA, lo ideal fuese que las aplicaciones ejecutadas en GPU realicen 4 o 5 operaciones aritméticas con cada uno de los bytes que quieran procesarse.

Teniendo esto en cuenta, a continuación se realizará un pequeño análisis de prestaciones de la aplicación para saber cuán buena es esta aplicación para ser ejecutada en CUDA y cuánto aprovecha los recursos hardware. Para ello se hará uso de los datos obtenidos del profiling en la GPU personal Nvidia GeForce GT 610 de la ejecución de la aplicación.

**instructions per bytes**

El factor 'instructions' representa el número de instrucciones que son lanzadas a ejecutar; es decir, el número de instrucciones que llegan a la etapa Issue de un procesador segmentado. Para hallar este valor se hace uso de los resultados obtenidos del profiling de la aplicación, datos que se encuentra en la Tabla 4:

$$\text{instructions} = 32 \text{ x instructions\_issued} = 32 \cdot 0.87 \cdot 10^9 = 27.84 \cdot 10^9$$

Por otro lado, para hallar el factor 'bytes', que representa la cantidad de memoria que hace uso la aplicación, se halla mediante el número de transacciones a memoria global (global_store_transaction) y el número de fallos en memoria local (l1_global_load_miss), cuyos datos se han obtenido con el profiler y se muestran en la Tabla 4. Para el cálculo de este factor se debe tener en cuenta que al realizar cargas/almacenamientos en su totalidad de números en doble precisión, se multiplica por 2 el factor 'bytes', tal y como recomienda Nvidia.

$$\text{bytes} = 128 \text{ B} \cdot (\text{global\_store\_transaction} + \text{l1\_global\_load\_miss})$$
$$\text{bytes} = 2 \cdot 128 \text{ B} \cdot (419.43 \cdot 10^6 + 4.19 \cdot 10^6) = 108.45 \cdot 10^9$$

Por lo tanto, finalmente si llevamos a cabo la operación instruction:bytes obtenemos:

$$\textbf{instructions:bytes} = 27.84 \cdot 10^9 : 108.45 \cdot 10^9 \approx 0.2567$$

NOTA: Estas medidas se llevaron a cabo con los datos obtenidos del profiler en la GPU: Nvidia GeForce GT 610.

Este valor indica que para cada bytes que se procesa en la aplicación, en promedio, no le corresponde ni una sola instrucción aritmética. Esto nos sugiere, a pesar de la notable mejora de tiempo obtenida en comparación con la versión secuencial, que no es una aplicación que aproveche la totalidad del potencial que nos ofrece CUDA.

Esto ha ocurrido porque en la aplicación hace uso de un número muy pequeño de instrucciones por cada dato. Teniendo esto en cuenta y que los datos que se procesan son de doble precisión; es decir de 8 bytes, es coherente haber obtenido un valor tan pequeño de instrucciones aritméticas por byte.

## 2. Paralelización del S.U.S. CODE

### 2.1 Paralelización lógica de los datos

Una forma de abordar la paralelización del algoritmo S.U.S. es desglosando el cómputo sobre los datos que van a ser procesados en distintos hilos CUDA o procesadores de la GPU. En nuestro caso, esta estrategia de paralelización es la más adecuada debido a que muchos de los datos que maneja el algoritmo S.U.S. se pueden tratar de manera independiente. Por lo tanto, basándonos en la Figura 33, que representa una malla de dos dimensiones por simplicidad; un determinado nodo se conecta a un conjunto de vértices, denominados "nodos vecinos" o 'vecindad'. Cada nodo vecino se conecta a su vez a otros nodos, siendo estos sus vecinos y así hasta el conjunto total de nodos.

El algoritmo S.U.S. consiste básicamente en un bucle que recorre todos los nodos de una malla. En cada iteración se actualiza solamente el emplazamiento en el espacio 3D de un nodo individual de la malla. Para ello, el algoritmo S.U.S. usa, sin modificarlos, los emplazamientos espaciales de los vértices que forman la vecindad del nodo que se actualiza en cada iteración. Es decir, en la recolocación de un nodo se toman como puntos de referencia las coordenadas de los nodos con los que éste se conecta. Por tanto, existe una dependencia real de datos entre los vecinos del nodo que se actualiza en cada iteración y el propio nodo que se actualiza.

Para paralelizar el algoritmo S.U.S., se debe asegurar que un hilo CUDA no actualice el emplazamiento espacial de un vértice que a su vez forma parte de la vecindad de otro nodo cuyo emplazamiento espacial se está actualizando en paralelo en otro hilo CUDA. Si se optimizaran dos nodos conectados paralelamente, no habría puntos de referencia fijos, puesto que las coordenadas de estos nodos serían cambiantes a lo largo de este proceso.

Una estrategia de paralelización consiste en encontrar los denominados "conjuntos independientes". Cada conjunto independiente está constituido por los nodos de la malla que no están directamente conectados entre sí a través de un único vértice. El emplazamiento espacial de los nodos de un conjunto independiente se puede actualizar en paralelo ya que se asegura que ninguno de los nodos independientes forma parte de la vecindad de ninguno de los otros nodos del mismo conjunto independiente.

Para saber qué nodos son independientes entre sí, y de este modo saber cuáles pueden ser procesados de forma simultánea, a la malla se le aplica un coloreado. El coloreado básicamente asigna un color a cada nodo de la malla; de modo que no existen dos nodos del mismo color conectados a través de un mismo vértice.

Por lo tanto, todos los nodos de un mismo color indica que éstos son totalmente independientes entre sí; tal y como se puede observar en la Figura 33 donde aparece un ejemplo de una malla 2D. En esta Figura se puede observar que los nodos verdes no están conectados a ningún nodo de su mismo color. Lo mismo ocurre con los nodos de los otros colores.

Después de haber coloreado la malla podemos ejecutar paralelamente la optimización de todos los nodos de un mismo color, asegurándonos así que no habrá problemas de dependencia de datos. Si se lleva a cabo este procedimiento secuencialmente para cada uno de los colores, se procesarán todos los nodos de la malla.

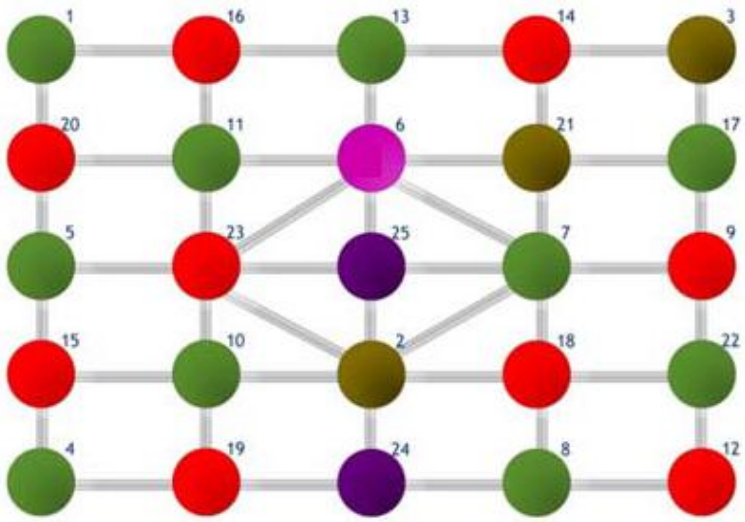


**Figura 33. Ejemplo del coloreado de una malla 2-D.**

Para la paralelización en GPU de El S.U.S. code se han probado los siguientes coloreados:

- Coloreado Montecarlo [5]
- Coloreado Grebemedhim [6]
- Coloreado B [7]

Para seleccionar cualquiera de ellos, se modifica el fichero Makefile que automatiza la compilación de los programas que implementan el algoritmo S.U.S.

## 2.2 Implementación

En primer lugar se debe comentar que se ha seguido un modelo incremental para el diseño e implementación de este código de desenredo y suavizado simultáneo de mallas de tetraedros paralelizado en GPU. El gráfico de tiempos que se muestra en la Figura 34, corresponde al Plan de Trabajo que se ha seguido en este proyecto. En él se muestra que la mayoría del tiempo consumido se ha aplicado al estudio minucioso tanto del funcionamiento de CUDA, como del código de desenredo y suavizado simultáneo de mallas, así como la implementación de la primera versión paralela en GPU del S.U.S. code. También se puede observar que se ha empleado gran cantidad de tiempo en el análisis de esta implementación, instrumentación de cada una de las versiones del algoritmo y optimización de las mismas. Por último, se necesitaron varias sesiones de trabajo para la documentación de todos los resultados obtenidos a lo largo del proyecto y que se muestran en este documento.

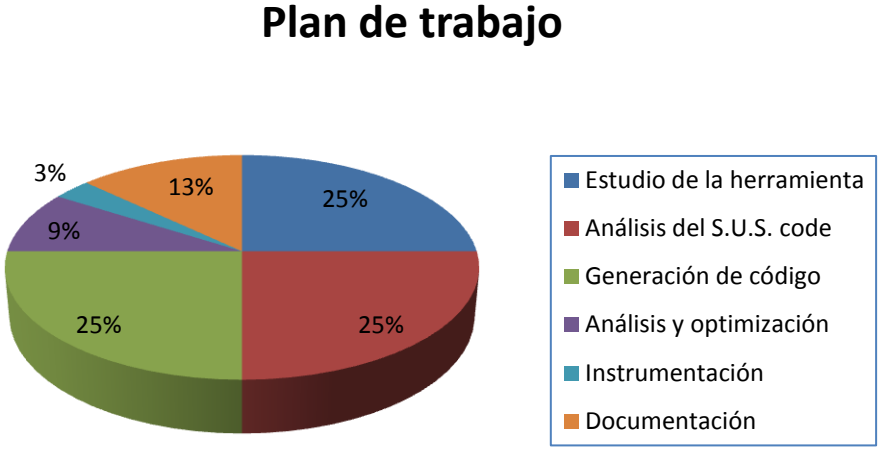


**Figura 34. Plan de trabajo del proyecto.**

A continuación se explicarán los pasos llevados a cabo para la implementación CUDA del algoritmos S.U.S. en GPU.

El código de desenredo y suavizado de mallas original hace uso de una librería que implementa en C++ la minimización sin restricciones (UNConstrained MINimization en C++, UNCMIN++) con el método Line Search, entre otros. Para ello esta librería necesita una clase que implementa un tipo de datos como vectores y matrices, pero con métodos propios ya implementados. Esta clase de la que depende UNCMIN++ es un ToolKit numérico simple en C++ (Simple C++ Numerical ToolKit, SCPPNT). Para la paralelización de S.U.S. en GPU se ha implementado en CUDA la librería UNCMIN++, eliminando su dependencia de la librería SCPPNT.

Para la implementación del código en CUDA se decidió crear una clase interna que implementaba los métodos de minimización sin restricciones. En nuestra implementación CUDA se crearon tanto la clase como el algoritmo que se comunica con la aplicación C++ haciendo uso de un único fichero. De esta forma se facilita la tarea de compilación y se ahorran problemas de enlazamientos entre diferentes ficheros con extensión **.cu**.

Cabe destacar que se encontraron problemas a la hora de compartir la información de las mallas del procedimiento C++ que se ejecuta en el host con el procedimiento CUDA que se ejecuta en la GPU. Para solucionar dicho problema se usaron técnicas clásicas para la copia de la información entre procedimientos.

### 2.2.1 Detalles de la implementación secuencial de S.U.S.

Para entrar en detalle cómo se ha implementado el desenredo y suavizado de mallas de tetraedros de forma paralela, se explicará a continuación cómo se lleva a cabo la optimización de la malla de manera secuencial. De este modo, se tendrá un punto de referencia para luego entender mejor cómo se ha llevado a cabo la paralelización del algoritmo.

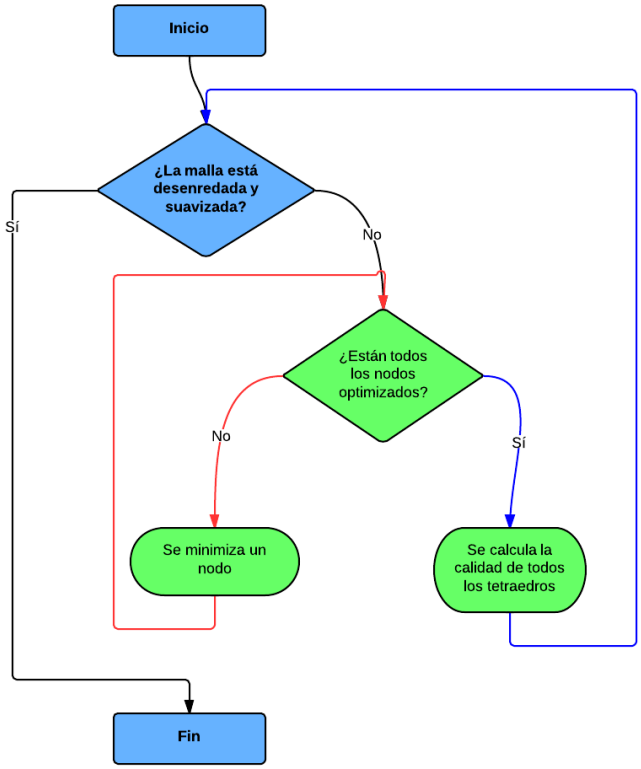


**Figura 35. Diagrama de flujo del algoritmo de desenredo y suavizado de mallas secuencial.**

En primer lugar, se puede observar en la Figura 35, que el algoritmo consta de dos bucles principales. El algoritmo superior, marcado de color azul, es aquel que optimiza toda la malla en cada iteración, cuyos criterios de parada de este bucle tiene en cuenta:

- Que no se supere un número máximo de iteraciones o
- Que la malla ya haya sido optimizada

Para comprobar el segundo criterio de parada, se hace uso de una función que calcula las calidades de todos los tetraedros que forman la malla. Este algoritmo se explicará posteriormente, pero básicamente, éste indica si la forma de todos los tetraedros es regular.

Por otro lado, en la Figura 35, también se puede observar que consta de un bucle interno, marcado en la Figura con el color rojo. Este bucle realiza la optimización de cada uno de los nodos de la malla. Es por esto, que se considera esta versión del algoritmo como versión secuencial.

El siguiente problema que se nos plantea a continuación es, cómo se realiza esta optimización de cada nodo. Como se ha comentado en el capítulo anterior, en la sección de S.U.S. code, cuando se optimiza un nodo (*nodo libre*), se toma como referencia el resto de nodos que están conectados con él a través de una única arista. Teniendo esto en cuenta, la intensión es la de minimizar una función objetivo modificando la posición del nodo libre de modo que, la calidad del tetraedro o tetraedros a los que pertenece ese nodo aumente.

Para ello, el algoritmo que se lleva a cabo es el que se representa en la Figura 36:

El criterio de parada del bucle que encierra al algoritmo es el número de iteraciones o la convergencia del mismo. En primer lugar, se toman unos valores iniciales que se tendrán en cuenta a la hora de realizar comparaciones posteriores.

```
do
   new_FKnupp_with_grad_and_hess(...)
   se define una direccion d | ∇f(x)d < 0
   minimizar( F(α) = f(x + αd) )
   se define Xk+1 = Xk + αd

while ( no converja ó supere numero de iteraciones )
```

Figura 36. Pseudocódigo de la optimización de un nodo.

Estos valores son básicamente el valor de la función Knupp del nodo libre, así como gradiente y el hesiano del mismo. A continuación, teniendo en cuenta el gradiente se elije una dirección de movimiento '$d$' para que la función $\nabla f(x)d < 0$, y de este modo encontrar el mínimo de la función y por lo tanto la posición ideal del nodo. El siguiente paso es minimizar la función $F(\alpha) = f(x + \alpha \cdot d)$, donde se busca un α que indica el desplazamiento en la dirección '$d$', para encontrar el mínimo de la función. El diagrama de flujo del algoritmo es el que aparece en la Figura 37:

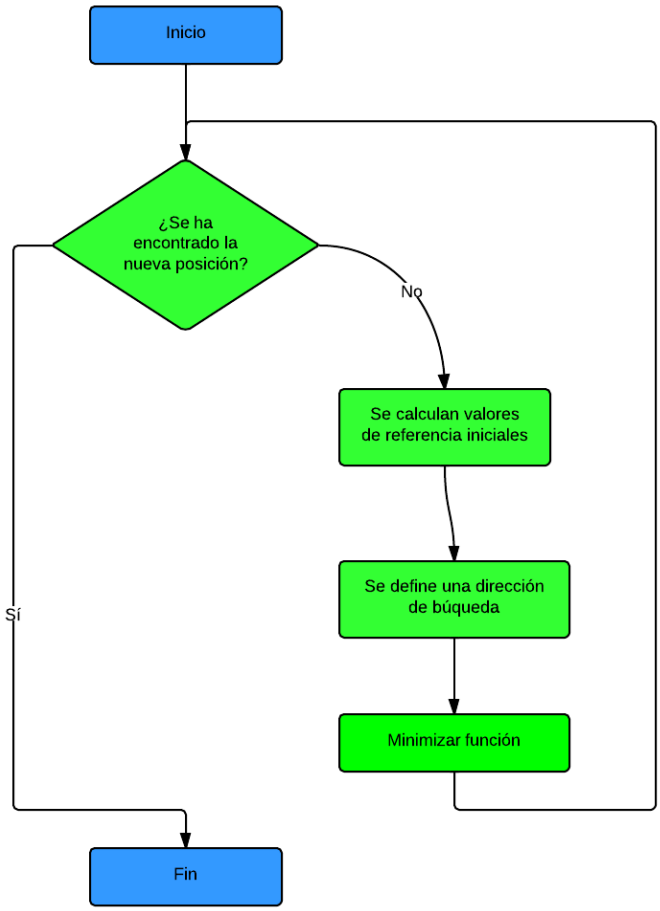


Figura 37. Diagrama de flujo de la optimización de un nodo.

La minimización que se lleva a cabo, para encontrar ese α o desplazamiento, se basa en una minimización por el método del descenso del gradiente.

Básicamente la minimización es un proceso iterativo que evalúa la función en diferentes puntos siguiendo una dirección. En cada uno de los puntos que se analizan se evalúa el gradiente y el hesiano hasta encontrar el valor mínimo de la función. El criterio por el cual se elige la dirección y los nuevos puntos a analizar, se basan principalmente en el descenso del gradiente, tal y como se

puede visualizar en la Figura 38.

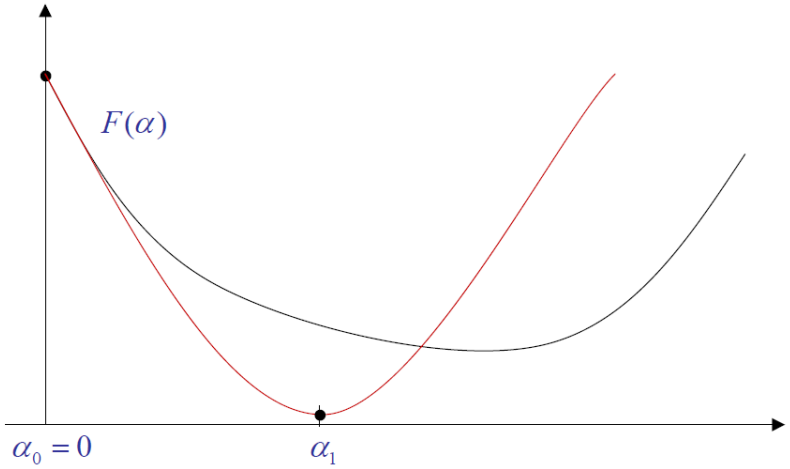


**Figura 38. Función aproximada mediante el descenso del gradiente conjugado.**

Al ser un método numérico de aproximación de soluciones, se puede ver en la Figura 38 como el resultado que se obtiene del método de descenso por gradiente (línea roja, marcada por $\alpha_1$) es un valor muy próximo al mínimo real de la función (línea oscura), pero no es la solución real. De este modo obtendríamos una nueva posición para el nodo que se está analizando en ese momento, pero ésta podría no ser la posición óptima. Es por esto que el algoritmo de minimización de la malla debe ser un proceso iterativo, el cual debe repetirse tantas veces como sea necesario para conseguir la mejor posición de cada uno de los nodos y así obtener una malla de calidad.

Por otro lado, la función que realiza el cálculo de las calidades, como se comentó anteriormente, es la causante de que se detenga el bucle superior en la que en cada iteración optimiza toda la malla. Ésta función básicamente calcula el número de elementos que aún se encuentran invertidos, y calcula la calidad promedio de todos los tetraedros de la malla con las fórmulas que se expusieron en el capítulo anterior. A continuación se entrará en detalle del algoritmo que realiza esta acción.

```
1  for each tetrehedral loop
2
3     Se construye la matriz jacobiana con las coordenadas
4     Se calcula la inversa de dicha matriz
5     Se calcula el determinante de la misma
6     Se calcula la norma Frobenius de la matriz
7     Se calcula finalmente la calidad
8
9  end loop
```

**Figura 39. Pseudocódigo del cálculo de las calidades de los tetraedros.**

El algoritmo de cálculo de las calidades se basa en un bucle que recorre todos los tetraedros, calculando su matriz jacobiana mediante las coordenadas de todos los nodos que forman el tetraedro (ver Figuras 39 y 40). Posteriormente mediante cálculo aritmético, se calcula la inversa de dicha matriz para obtener su norma Frobenius. Por último se calcula la calidad del elemento atendiendo a la siguiente fórmula: $q = {nf^2}/{3 \cdot d^{2/3}}$ donde '*d*' es el determinante que se habrá calculado a priori y siendo '*nf*' la norma frobenius.

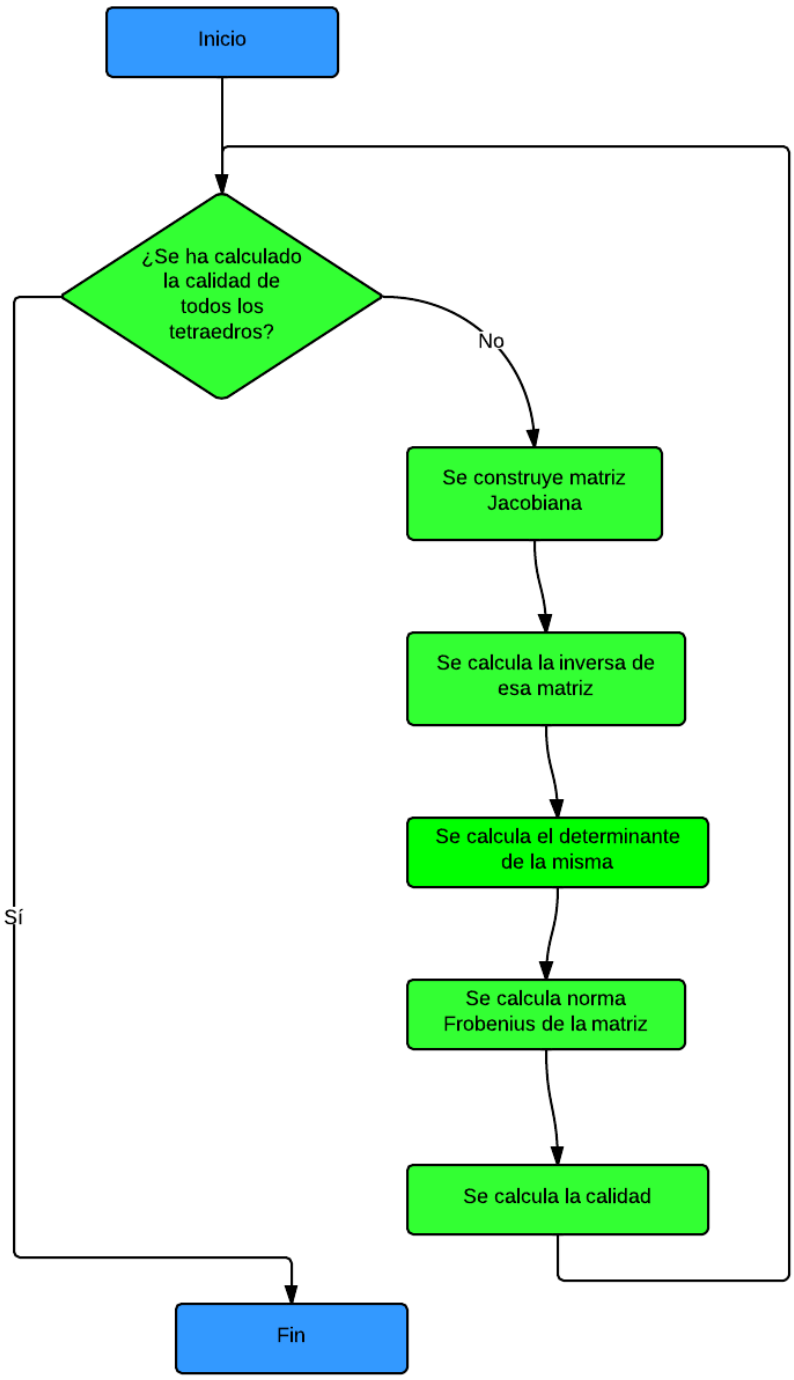


**Figura 40. Diagrama de flujo del cálculo de las calidades de los tetraedros.**

### 2.2.2 Etapas de procesamiento del programa paralelo CUDA

En la Figura 41 se muestra una visión general del algoritmo paralelo implementado. Como se puede ver el algoritmo en general consta de varias fases, las cuáles serán explicadas con detenimiento a continuación:

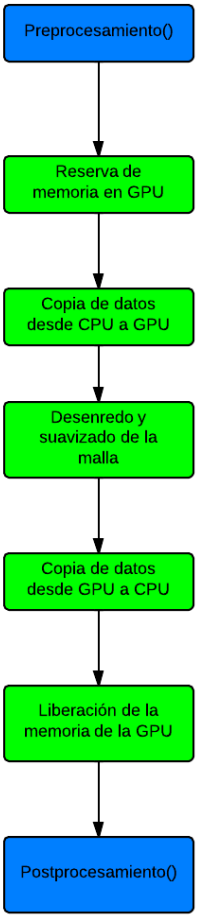


**Figura 41. Traza del algoritmo S.U.S. code**

Etapa 1) Preprocesamiento()

En primer lugar, se realiza la lectura de la malla cuya información se encuentra en dos ficheros. En uno de ellos se almacena las coordenadas de cada uno de los nodos. En otro se almacena la información de cada tetraedro, de modo que indica qué nodos se encuentran conectados formando un elemento de la malla.

En este preprocesamiento, mediante uno de los tres diferentes algoritmos de coloreado que se han implementado, se realiza la identificación de los distintos conjuntos independientes de nodos (colores) de la malla. De este modo se identifican qué nodos pueden procesarse de manera paralela.

Hay que comentar que este preprocesamiento, lectura de malla y coloreado de la misma, se realiza en el host mediante un algoritmo programado en C++ y por lo tanto compilado con g++.

Etapa 2) Reserva de memoria en la GPU

Una vez se obtiene la información de la malla (cantidad de nodos y tetraedros que forman dicha malla, y tamaños de las estructuras de datos asociadas), desde la aplicación CUDA se realiza la reserva de memoria en el dispositivo GPU para alojar toda esta información. Este proceso se realiza mediante las rutinas que nos ofrece CUDA tal y como se ha explicado en el capítulo anterior en la sección de CUDA.

Este proceso, se ejecuta en la CPU del host, a pesar de ser una subrutina de CUDA que lleva a cabo

la reserva de memoria en la GPU.

Etapa 3) Copia de datos desde el host a la GPU

Durante el preprocesamiento se ha leído la información de la malla desde ficheros y se ha almacenado dicha información en la memoria principal de la CPU. Para paralelizar el algoritmo en GPU es necesario trasladar esa información a la memoria principal de la GPU. Como en el proceso anterior ya se ha reservado dicho espacio de memoria, ahora se realiza la copia desde la memoria del host hacia ese espacio de memoria en la GPU mediante unas rutinas de CUDA que se dedican expresamente a eso.

Este proceso, al igual que el anterior, al ser una subrutina de CUDA se lleva a cabo sobre la GPU pero llamando a dicha rutina desde el procedimiento CUDA que se ejecuta en el host.

Etapa 4) Desenredo y suavizado simultáneo de la malla

Una vez se tiene toda la malla almacenada en la GPU podemos de forma simultánea desenredar y suavizar la malla de tetraedros en este dispositivo. De este modo todos los cambios que se realicen sobre la malla se quedan almacenados en la memoria principal de la GPU. De momento basta con saber que en este punto del procedimiento global se realiza la optimización de la malla. En subapartados posteriores se entrará en detalle cómo se realiza esta optimización paralela en la GPU.

Esta etapa implementa básicamente con procedimientos lanzados desde la aplicación de CUDA en CPU pero que se ejecutan en la GPU.

Etapa 5) Copia de datos desde GPU al host

Para poder disponer de la malla desenredada y suavizada en el host con el objeto de almacenarla o para posteriores procesos, es necesario el uso de las subrutinas CUDA de copia de entre la GPU y el host. De este modo se modifica la información de la malla que se encuentra almacenada en la memoria principal de la CPU. Este proceso aún se sigue llevando a cabo en la aplicación de CUDA.

Etapa 6) Liberación de la memoria reservada en la GPU

Por último, la aplicación de CUDA libera el espacio de memoria de la GPU utilizado para la optimización de la malla, dando por finalizado el proceso de desenredo y suavizado simultáneo de mallas.

Etapa 7) Postprocesamiento()

En este último paso del algoritmo, una vez se tiene la malla procesada, se analizan los datos obtenidos de ese cómputo, para comprobar si éste ha tenido éxito y ha optimizado la malla o por el contrario no ha convergido el algoritmo obteniendo una malla aún está sin desenredar y suavizar. A partir de este punto, ya se ha finalizado la aplicación de CUDA y vuelve a la aplicación de C++.

Antes de entrar en los detalles del algoritmo de desenredo y suavizado de la Etapa 4, se considera oportuno comentar una función que ha sido necesaria para mantener la coherencia entre las estructuras de datos que almacenan la malla dentro de la GPU. Esta función se ha denominado como “cudaMemcpyMeshToDevice” y básicamente lo que hace es establecer conexiones entre las direcciones de memoria de las diferentes estructuras de datos almacenadas en memoria.

Para comprender mejor el motivo de esta función, se expondrá un pequeño ejemplo en la Figura 42:

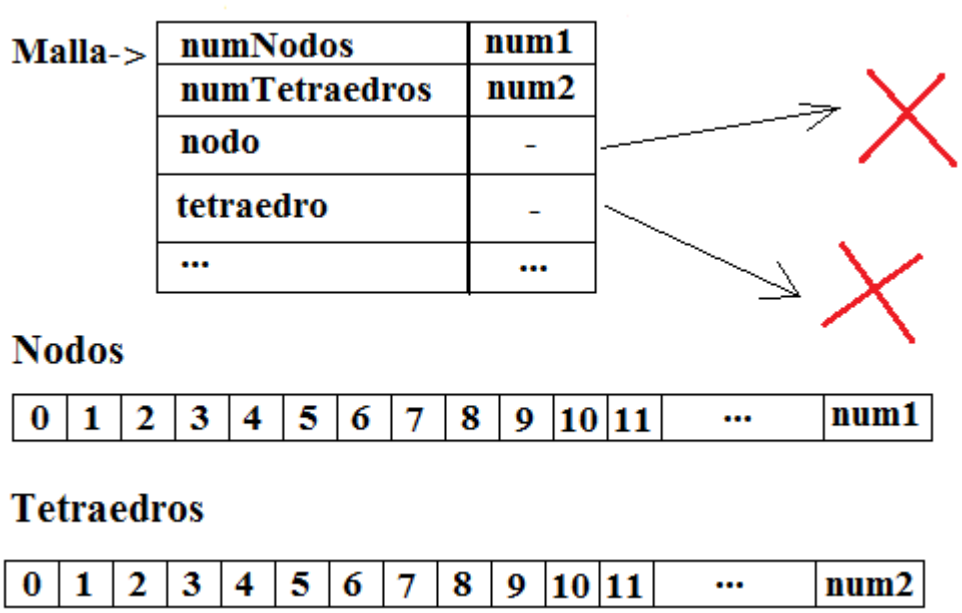


Figura 42. Ejemplo de la información almacenada en la GPU tras una copia de datos.

En la Figura 42 podemos ver qué es lo que hay en la memoria de la GPU una vez se halla reservado y copiado en memoria la información de todos los nodos, la de los tetraedros y la información general de la malla. La malla contiene punteros tanto para acceder a los nodos como a los tetraedros. Cuando se realiza la copia de la malla a la GPU se copian los punteros, los cuales no corresponden a las posiciones donde se encuentran almacenadas ambas estructuras en la memoria de la GPU. Por lo tanto, la función "cudaMemcpyMeshToDevice" actualiza esos punteros para que toda la información pueda ser accedida directamente desde la estructura de la malla. En la Figura 43 se representa el efecto que resulta al aplicar esta función que se ha desarrollado en este proyecto.

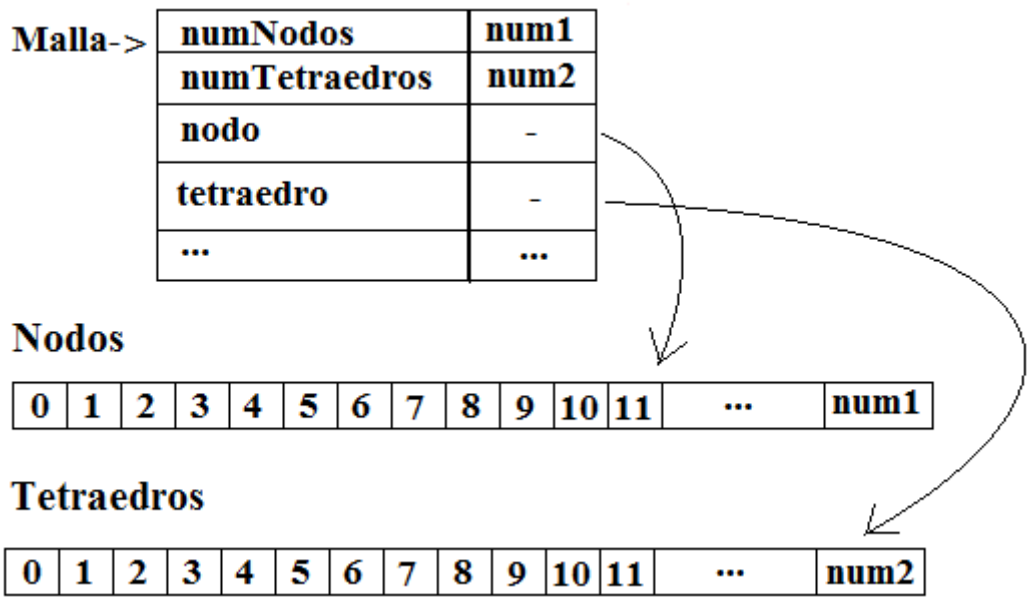


Figura 43. Ejemplo de la información de la GPU tras realizar la redirección.

### 2.2.3 Detalles de la implementación paralela de S.U.S. en CUDA

Una vez explicado cómo se lleva a cabo la optimización de un nodo, y de ver que este proceso llevado a cabo de forma iterativa para cada nodo produce la optimización de la malla en su conjunto, a continuación se comentará cómo se ha llevado a cabo la paralelización de este procedimiento.

Cómo se ha visto en capítulos anteriores e incluso al inicio de este capítulo, una de las principales características del algoritmo S.U.S. es la gran capacidad de paralelización que nos ofrece. Esta capacidad de paralelización se ve claramente reflejada después que el coloreado se le aplica a la malla, ya que un color indica qué nodos pueden ser optimizados de manera independiente sin que haya ningún tipo de dependencias entre ellos. Sabiendo esto, se ha optado por la paralelización del algoritmo de desenredo y suavizado de modo que se optimicen en paralelo todos los nodos que pertenecen a un mismo color.

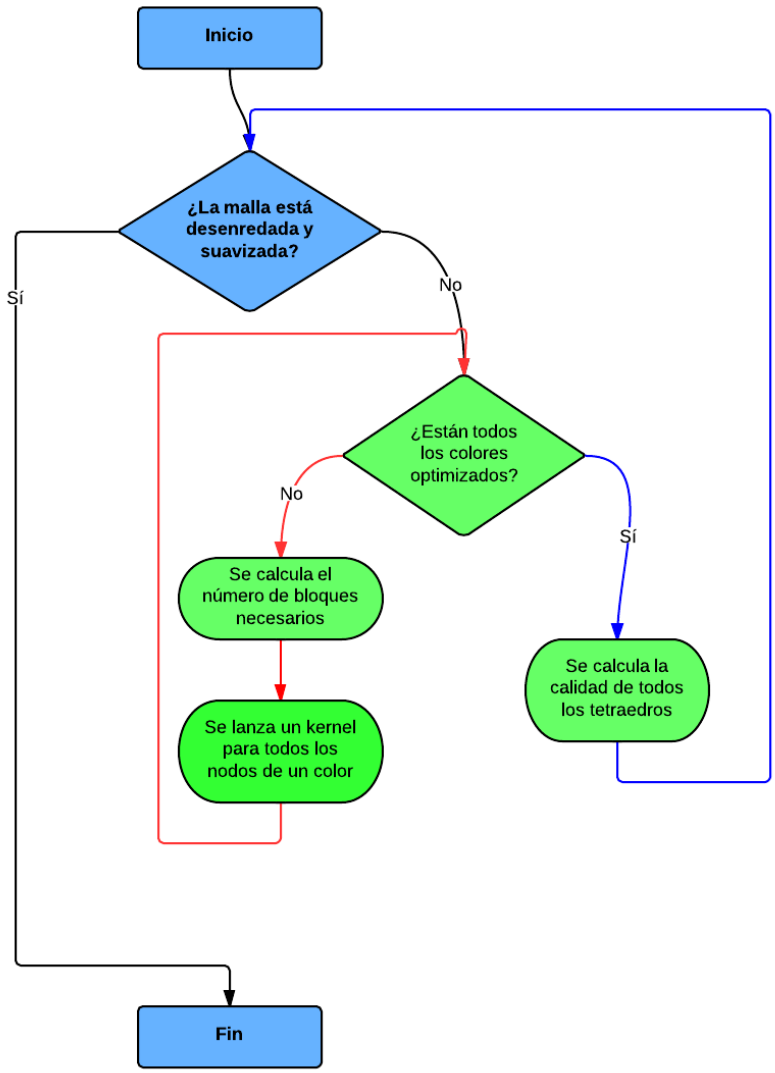


Figura 44. Diagrama de flujo del algoritmo paralelizado del desenredo y suavizado de mallas.

Cómo se puede ver en la Figura 44, la paralelización se ha conseguido al encapsular en un kernel todos los nodos de un color. Este kernel es lanzado a la GPU y se encarga de asignar la optimización de cada nodo a cada core de los que dispone el hardware. Por lo tanto, podemos ver que el bucle interno, indicado en la Figura 44 por el color rojo, es el que lanza en cada iteración un

kernel con todos los nodos de un mismo color; es decir, un conjunto de nodos independientes entre sí. Para ello, se puede observar en la Figura que antes del lanzamiento de un kernel se realiza una configuración del mismo, ajustando el número de bloques necesarios para encapsular todos los nodos de ese color. Durante la configuración se tiene en cuenta el número de hilos por bloque, el cuál es un valor fijo durante todo el algoritmo. Teniendo en cuenta este valor y el número de nodos a procesar paralelamente, se calcula el número de bloques necesarios. Una vez calculado el número de bloques necesarios, antes de realizar la configuración del kernel se comprueba que no se supera el número máximo de bloques en una determinada dimensión. Este valor viene dado por las características de la GPU, cuyo dato es obtenido en tiempo de ejecución. Esto hace que el algoritmo sea fácilmente trasladable a otros dispositivos compatibles con CUDA y aprovechar al máximo los recursos del mismo.

Por otro lado, se puede observar a su vez la existencia de un bucle exterior, coloreado en la Figura 44 con el color azul, que hace la misma función que en la versión original, que es la de optimizar la malla en cada iteración hasta que se cumplan los criterios de parada: convergencia del algoritmo o se alcance el número máximo de iteraciones.

Con respecto al cálculo de las calidades se ha optado por implementarse en CUDA, de modo que se realice en la GPU. Esto se debe a que, si se hiciera este proceso en la CPU se debería copiar a memoria principal del host las nuevas posiciones de los nodos de la malla. Si esto se hiciese en cada una de las iteraciones, hace al algoritmo bastante ineficiente, y dependiente de la velocidad del bus PCIexpress.

A modo ilustrativo, se muestra en la siguiente Figura 45 una malla enredada a la izquierda, y a la derecha la misma malla totalmente desenredada y suavizada, obtenida por el algoritmo S.U.S. paralelo.

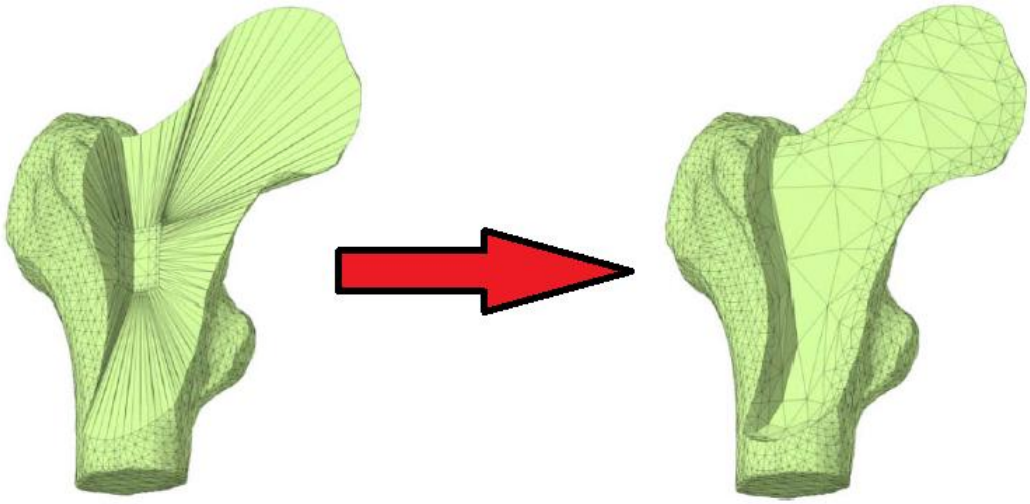

**Figura 45. Comparativa de la malla del hueso enredada y desenredada.**

### 2.3 Análisis de prestaciones

Para el análisis de prestaciones de las diferentes versiones del código S.U.S. se hace uso de tres mallas de tetraedros enredadas. La información de cada una de ellas se puede visualizar en la Tabla 7. Todas las mallas test han sido creadas con el método del Meccano[2][3], sin aplicar la fase de desenredo y suavizado.

Tabla 7. Especificaciones de las diferentes mallas utilizadas durante el proyecto.

| Malla | Número de nodos | Número de tetraedros | Número de tetraedros enredados |
|---|---|---|---|
| **Hueso** | 11525 | 47824 | 1307 |
| **Conejo** | 6358 | 26446 | 1218 |
| **Ej9_9176** | 9176 | 35920 | 13706 |

Para comenzar con el análisis del código implementado, se compara en primer lugar el tiempo de cómputo del S.U.S. code secuencial ejecutado en CPU (host) con el ejecutado en la GPU. Esta comparación se ha llevado a cabo en las dos máquinas que se indicaron al inicio de este documento. En la Tabla 8 se muestran los tiempos de cómputos que fueron obtenidos cuando se utilizó un computador de tipo sobremesa (personal) basado en un procesador Intel i5 y una placa Nvidia GeForce GT 610. En la Tabla 9 se muestran los tiempos de cómputo que fueron obtenidos cuando se utilizó un computador servidor (vector.iusiani.ulpgc.es) basado en un procesador Intel Xeon E5620 y una placa Nvidia Tesla S2050. Los resultados de las versiones paralelas que aparecen en ambas Tablas se obtuvieron utilizando el algoritmo de Montecarlo para el coloreado de la malla.

Tabla 8. Tabla de comparación de tiempos del S.U.S. en el ordenador personal.

| Malla | Tiempo de cómputo Código secuencial (seg) | Nº de iteraciones, versión secuencial | Tiempo de cómputo Código CUDA (seg) | Nº de iteraciones versión paralela | Diferencia (seg) |
|---|---|---|---|---|---|
| **Hueso** | 10.42 | 34 | 100.11 | 34 | **89.69** |
| **Conejo** | 14.62 | Max_Num_Iter 50 | 70.62 | 24 | **70.62** |
| **Ej9_9176** | 11.88 | 26 | 137.85 | 25 | **173.97** |

NOTA: Medidas tomadas con el coloreado de Montecarlo. GPU: Nvidia GeForce GT 610; CPU: Intel i5.

Tabla 9. Tabla de comparación de tiempos del S.U.S. en el computador VECTOR.

| Malla | Tiempo de cómputo Código secuencial (seg) | Nº de iteraciones, versión secuencial | Tiempo de cómputo Código CUDA (seg) | Nº de iteraciones, versión paralela | Diferencia (seg) |
|---|---|---|---|---|---|
| **Hueso** | 19.39 | 36 | 85.57 | 39 | **66.18** |
| **Conejo** | 10.05 | 34 | 93.59 | Max_Num_Iter 50 | **10.05** |
| **Ej9_9176** | 16.77 | 25 | 91.35 | 28 | **74.58** |

NOTA: Medidas tomadas con el coloreado de Montecarlo. GPU: Nvidia Tesla S2050; CPU: Intel Xeon.

Antes de comentar estos resultados, los valores de “Max_Num_Iter” en el número de iteraciones indican que se alcanzó el número máximo de iteraciones sin haber convergido; es decir, sin haber desenredado y optimizado la malla completamente. Esto se debe a que algunas mallas de tetraedros son más sensibles a no converger correctamente en unos u otros dispositivos hardware a causa de sus características. Al tratarse la optimización de un proceso iterativo, los pequeños cambios en la precisión que se puedan encontrar en las diferentes máquinas hacen que varíe los datos con los que

se trabaja, de tal modo que en ocasiones hacen que el algoritmo no converja correctamente. En cualquier caso, el número de iteraciones que el algoritmo necesita para desenredar y suavizar la malla determina el tiempo de ejecución del algoritmo en un determinado dispositivo.

A pesar de lo comentado anteriormente, tal y como se puede observar en los resultados obtenidos en una primera instancia y mostrados en las Tablas 8 y 9, la aplicación que se ejecuta sobre la GPU es mucho más lenta que la versión secuencial. Este resultado no es satisfactorio, por lo que se propuso buscar cuál es la causa de esta ineficiencia del paralelismo en este algoritmo implementado en la GPU.

Para continuar con la fase de análisis de prestaciones y de este modo encontrar cuál es el problema que está surgiendo para que el algoritmo paralelo sea mucho más lento que el secuencial, se hará uso de dos técnicas recomendadas por los diseñadores de Nvidia.

- Análisis de prestaciones con la herramienta profiler
- Análisis de prestaciones modificando el código fuente

### 2.3.1 Análisis de prestaciones con el profiler

Gracias a la ayuda del profiler (nvprof) que acompaña a la distribución CUDA se tomaron durante una ejecución del código S.U.S. paralelizado en la GPU las medidas que se muestran en la Tabla 10 cuando se utilizó la placa Nvidia de un computador de sobremesa y en la Tabla 11 cuando se utilizó la placa Nvidia de un computador servidor. Estas medidas se obtuvieron utilizando la malla denominada “Hueso” y el algoritmo Montecarlo para el coloreado de la malla. Todas estas medidas están relacionadas con el número de instrucciones ejecutadas en la GPU y el comportamiento de parte de su jerarquía de memoria: cache L1 y memoria global.

**Tabla 10. Tabla del profiling del S.U.S. original paralelizado en GeForce.**

| | |
|---|---|
| **Número de transacciones de almacenamiento a memoria global** | $7 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $737 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $23.5 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $18.9 \cdot 10^9$ |
| **Número de instrucciones entre cada fallo de memoria** | 25.64 |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia GeForce GT 610.

**Tabla 11. Tabla del profiling del S.U.S. original paralelizado en Tesla.**

| | |
|---|---|
| **Número de transacciones de almacenamiento a memoria global** | $17.84 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $565.34 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $22.36 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $20.77 \cdot 10^9$ |
| **Número de instrucciones entre cada fallo de memoria** | 36.74 |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia Tesla S2050.

Considerando estos resultados de la Tabla 10 podemos analizar cuán buena es la aplicación para ser ejecutada en CUDA y cuánto aprovecha los recursos suministrados. Para ello, a continuación se lleva a cabo un pequeño análisis del código CUDA tal y como se hizo en el último apartado del capítulo anterior con el caso de estudio del algoritmo de un vector.

NOTA: Estas medidas se llevaron a cabo con los datos obtenidos del profiler en la GPU: Nvidia GeForce GT 610.

**instructions:bytes**

Para hallar el factor 'instructions' se tiene:

$$\text{instructions} = 32 \text{ x instructions\_issued} = 32 \cdot 23.5 \cdot 10^9 = 752 \cdot 10^9$$

Por otro lado, para hallar el factor 'bytes' se debe tener en cuenta que al realizar cargas/almacenamientos en su mayoría de números en doble precisión, a causa de las coordenadas de los nodos, se multiplica por 2 el factor 'bytes':

$$\text{bytes} = 128\ B \cdot (\text{global\_store\_transaction} + \text{l1\_global\_load\_miss})$$
$$\text{bytes} = 2 \cdot 128\ B \cdot (7 \cdot 10^6 + 737 \cdot 10^6) = 190 \cdot 10^9$$

Por lo tanto, finalmente si llevamos a cabo la operación instruction:bytes obtenemos:

$$\textbf{instructions:bytes} = 752 \cdot 10^9 : 190 \cdot 10^9 \approx 3.95$$

Este valor indica que para cada bytes que se procesa en la aplicación, en promedio, se ejecutan casi 4 instrucciones aritméticas. Tal y como recomienda Nvidia Corporation, esta aplicación, al encontrarse el ratio de instrucciones por byte entre 3.6 y 4.5, cumple los requisitos para aprovechar al máximo el potencial de CUDA.

Curiosamente, hemos obtenido resultados de instructions:bytes totalmente distintos al caso del estudio del programa que realiza el producto entre vectores y que fue explicado en el capitulo anterior. En este estudio fue donde se obtuvieron muy buenos tiempos de comparación con la versión secuencial, pero sin embargo el ratio de instrucciones ejecutadas por byte no era un valor tan alto como el que se obtienen para el algoritmo S.U.S., por lo que indicaba que el aprovechamiento de los recursos de la GPU es bajo. En el caso de la aplicación S.U.S. obtenemos un ratio que indica un aprovechamiento mucho mayor del hardware, pero no se ve reflejado en el tiempo de respuesta de la aplicación.

Como la obtención del ratio de instrucciones ejecutadas por byte en el algoritmo S.U.S. nos muestra un posible alto aprovechamiento de la GPU pero sin embargo la versión paralela es mucho más lenta que la secuencial, a continuación se analizará de nuevo el análisis de prestaciones mediante otra técnica: modificación del código fuente. Con esta nueva técnica se pretende tener más información sobre las causas del deterioro de las prestaciones en la GPU respecto al procesador del host.

### 2.3.1 Análisis de prestaciones modificando el código fuente

El análisis del algoritmo S.U.S. mediante la modificación del código fuente trata de alterar el código de modo que se siga manteniendo la misma traza que realiza el código original. Esta alteración consiste en generar dos versiones distintas del código CUDA original. La primera versión se obtiene desactivando todas las operaciones de acceso a memoria, y la segunda versión se obtiene desactivando todas las operaciones aritméticas. De este modo, observando los tiempos de ejecución

de las dos aplicaciones podemos comprobar cuánto depende la aplicación original de uno u otro tipo de operaciones.

En primer lugar, se optó por la modificación del código eliminando todas las instrucciones referentes a memoria. A esta versión se le denominó: **versión aritmética**, puesto que al eliminarse todas las instrucciones de carga y de almacenamiento en memoria; se siguen conservando todas las instrucciones aritméticas. De este modo evitamos el uso de la memoria y los posibles retrasos que ésta pueda producir.

En esta versión aritmética, se cambiaron las operaciones de lectura en memoria por la asignación de coordenadas de un mismo nodo de la malla, el cual fue elegido al azar de entre las variables usadas por el algoritmo. De este modo aseguramos el trazado normal de la ejecución del algoritmo original, con la única diferencia que siempre se está optimizando el mismo nodo de la malla. Por otro lado, se eliminaron los almacenamientos en memoria para evitar el correspondiente tránsito a la memoria global.

Los resultados obtenidos de esta versión son los que se muestran en la Tabla 12 para la placa gráfica del ordenador de sobremesa y la Tabla 13 para la placa gráfica del servidor. En ambos casos, se volvió a utilizar la malla denominada “Hueso” y el algoritmo Montecarlo para el coloreado de la malla.

**Tabla 12. Tabla de resultados de la versión aritmética en GeForce.**

| | | **Diferencia con la versión original del S.U.S. CUDA code** |
|---|---|---|
| **Tiempo de cómputo (segundos)** | 16.03 | -84.08 |
| **Número de transacciones de almacenamiento a memoria global** | 0 | $-7 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | 0 | $-737 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $5.66 \cdot 10^9$ | $-17.84 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $4.78 \cdot 10^9$ | $-14.12 \cdot 10^9$ |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia GeForce GT 610.

**Tabla 13. Tabla de resultados de la versión aritmética en Tesla.**

| | | **Diferencia con la versión original del S.U.S. CUDA code** |
|---|---|---|
| **Tiempo de cómputo (segundos)** | 6.69 | -78.88 |
| **Número de transacciones de almacenamiento a memoria global** | 0 | $-17.84 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | 0 | $-565.34 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $5.66 \cdot 10^9$ | $-16.7 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $4.78 \cdot 10^9$ | $-15.99 \cdot 10^9$ |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia Tesla S2050.

En las Tablas 12 y 13 se puede observar que el número de operaciones en la jerarquía de memoria son 0; lo que quiere decir que se han eliminado correctamente las operaciones relacionadas con la jerarquía de memoria. Por otro lado se puede ver también que tanto en una GPU como en otra, el número de instrucciones ejecutadas es mucho menor que el número de instrucciones que se ejecutan en la versión original. Esto se debe a que el algoritmo de desenredo contiene varios bucles condicionales, en los que se itera en varias ocasiones dependiendo de la información de los nodos vecinos. En esta versión al obligar que optimice un mismo nodo, se reduce el número de iteraciones y por tanto del cómputo.

Una segunda fase de este análisis de prestaciones a través de la modificación del código fuente, constaría de una versión del algoritmo donde sólo se activan los accesos de memoria y se desactivan las operaciones aritméticas. Cómo se ha comentado anteriormente, el algoritmo S.U.S. original se basa en una optimización de modo iterativo, por lo que el código está repleto de instrucciones de control del tipo 'if', 'while', etc. Esto hace que la modificación del código fuente a una versión de sólo memoria no resulte tan trivial. Por este motivo, se va suponer que el resto del tiempo consumido por la aplicación S.U.S. respecto a la versión aritmética se debe a retrasos provenientes de las instrucciones que acceden a memoria.

Los resultados obtenidos de los tiempos de procesamiento a partir de la ejecución de la versión original del algoritmo S.U.S. y de sus dos versiones modificadas del código fuentes son los que se muestran en la Tabla 14 para la placa gráfica del ordenador de sobremesa y la Tabla 15 para la placa gráfica del servidor. En ambos casos, se volvió a utilizar la malla denominada "Hueso" y el algoritmo Montecarlo de coloreado de la malla.

**Tabla 14. Comparativa de tiempos de la versión S.U.S. paralelizada y otras versiones en GeForce.**

| | |
|---|---|
| **Tiempo de la versión S.U.S. paralela original** | 100.11 |
| **Tiempo de la versión aritmética** | 16.03 |
| **Tiempo que se supone de la versión de memoria** | 84.08 |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia GeForce GT 610.

**Tabla 15. Comparativa de tiempos de la versión S.U.S. paralelizada y otras versiones en Tesla.**

| | |
|---|---|
| **Tiempo de la versión S.U.S. paralela original** | 85.57 |
| **Tiempo de la versión aritmética** | 6.69 |
| **Tiempo que se supone de la versión de memoria** | 78.88 |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia Tesla S2050.

Como se ha demostrado el tiempo que se consume no es especialmente grande en la versión aritmética, lo que hace intuir que el aumento de tiempo en la versión original, se debe a problemas de latencia de memoria y de no aprovechamiento del denominado '**memory coalescing**'.

Por un lado, la latencia de memoria es el tiempo que tarda cualquier procesador en disponer de los datos en las unidades funcionales empezando a contar desde que envía la orden de acceso a memoria al controlador de memoria. Este tiempo suele ser en la GPU de varios cientos de ciclos cuando se accede a la memoria global, lo que produce que el tiempo que se consuma en la copia de dichos datos en los registros internos de los procesadores, no se pueda aprovechar para realizar cálculos y por tanto los núcleos de procesamiento de la GPU estén ociosos.

Por otro lado, en la arquitectura de la GPU, los accesos a la memoria global se ven favorecidos si los hilos de un mismo warp acceden a posiciones contiguas de memoria ('**memory coalescing**'), ya

que una única lectura proporciona 128 bytes contiguos. Si los datos que necesitan todos los hilos de un mismo warp están alojados en esas 128 posiciones contiguas de memoria, bastará con un solo acceso para que todos los hilos obtengan sus datos. Por el contrario, si los hilos necesitan acceder a posiciones dispersas (no contiguas) de memoria, los distintos procesadores que ejecutan los hilos de un warp solicitarán realizar varias operaciones de acceso a memoria, y mientras estas no terminen todos los hilos del warp estarán ociosos.

Estos problemas de latencia combinados con los problemas por la falta de localidad de los datos son los principales causantes del consumo del tiempo durante la ejecución en la GPU de la aplicación. Cuando un procesador de la GPU (core) está analizando un nodo, necesita las coordenadas de los nodos con los que éste se conecta, formando uno o varios tetraedros, para encontrar una nueva posición a ese nodo lo más óptima posible por medio de una función objetivo.

Al necesitar la información de todos los nodos vecinos, es necesario volcar en memoria esta información que estará distante entre sí o no. Imaginemos que se ejecutan paralelamente 32 hilos (un warp), donde cada hilo realiza la optimización de un nodo. Cuando un hilo durante la optimización de un nodo necesita la información de sus vecinos, a su vez los 31 hilos restantes necesitarán la información de sus vecinos correspondientes. Teniendo este problema en cuenta y que cada uno de los 32 nodos que se optimizan en paralelo se encuentran almacenados distantemente en memoria (recordemos que son nodos totalmente independientes entre sí) el problema se vuelve insostenible. Es por esto que, tanto este tipo de aplicación, como cualquier otra que requiera el acceso a datos no contiguos en memoria, se encontrarán con problemas de localización y latencia de memoria al ejecutarse en una arquitectura GPU. Este tipo de problemas son un caso de estudio actualmente, donde trabajos recientes tratan de abordarlo, intentado reducir lo máximo posible el impacto en el tiempo de procesamiento en este tipo de aplicaciones.

### 2.4 Optimización

Para intentar solucionar el problema de localización de memoria que sufre actualmente el algoritmo S.U.S., se decidió optimizar el código de modo que se agrupe en memoria los nodos del mismo color, para así tener localizados los nodos independientes y a la hora de acceder a la información de cada uno de ellos se encuentren agrupados en memoria

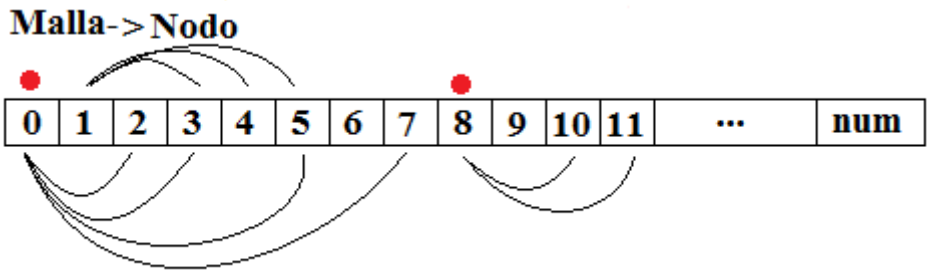


Figura 46. Ejemplo de agrupamiento en memoria del código original.

En la Figura 46 se muestra cómo se encuentran alejados en memoria los nodos 0 y 8, los cuales pertenecen al mismo color; por lo tanto son independientes. En la Figura 46 también se puede ver con quién están conectados estos nodos (los vecinos), mediante las líneas que parten de ellos.

Para llevar a cabo esta optimización, se optó por realizarla en diferentes fases, para asegurar de este modo el correcto funcionamiento del algoritmo y observar gradualmente cuál va siendo la mejora obtenida de cada optimización.

### 2.4.1 1ª aproximación

En un primer lugar, para llevar a cabo la optimización, se decidió cambiar la estructura de datos de los 'Nodos', puesto que esta estructura inicial está compuesta por muchos campos y atributos que no se usan durante el proceso de minimización de la función objetivo. Por lo tanto, lo que se hizo durante esta etapa fue la de cambiar toda la estructura de datos de la Malla, así como la estructura de datos de los nodos, para así almacenar sólo la información relevante y evitar la ocupación innecesaria de memoria.

Los resultados obtenidos para las prestaciones del algoritmo S.U.S. tras esta primera aproximación han sido los que se muestran en la Tabla 16 para la placa Nvidia del ordenador de sobremesa y en la Tabla 17 para el ordenador servidor. En ambos casos, se volvió a utilizar la malla denominada "Hueso" y el algoritmo Montecarlo de coloreado de la malla.

**Tabla 16. Tabla de resultados de la primera optimización en GeForce.**

| | | **Diferencia con la versión original del S.U.S. CUDA code** |
|---|---|---|
| **Tiempo de cómputo (segundos)** | 100.15 seg | -0.06 seg |
| **Número de transacciones de almacenamiento a memoria global** | $7.03 \cdot 10^6$ | $3 \cdot 10^4$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $582.95 \cdot 10^6$ | $-154.05 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $20.28 \cdot 10^9$ | $-3.22 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $18.95 \cdot 10^9$ | 0 |
| **Número de instrucciones entre cada fallo de memoria** | 32.51 | +6.87 |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia GeForce GT 610.

**Tabla 17. Tabla de resultados de la primera optimización en Tesla.**

| | | **Diferencia con la versión original del S.U.S. CUDA code** |
|---|---|---|
| **Tiempo de cómputo (segundos)** | 81.04 | -4.53 |
| **Número de transacciones de almacenamiento a memoria global** | $17.83 \cdot 10^6$ | $-0.01 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $500.01 \cdot 10^6$ | $-65.33 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $22.23 \cdot 10^9$ | $-0.13 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $20.61 \cdot 10^9$ | $-0.16 \cdot 10^9$ |
| **Número de instrucciones entre cada fallo de memoria** | 41.22 | +4.48 |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia Tesla S2050.

En las Tablas 16 y 17 se puede ver cómo se ha reducido mínimamente el tiempo de cómputo, puesto

que en parte, se han reducido tanto los accesos a memoria, como las instrucciones ejecutadas. Lógicamente se han reducido los fallos de carga porque ahora al traerse información más relevante, el espacio que se ocupa en la memoria es información útil, cosa que no sucedía en la versión original. Con respecto a la disminución de las instrucciones ejecutadas en el código no se trata de un valor excesivamente grande, por lo que carece de importancia al ver que en la ejecución en la GPU personal la diferencia es de 0. Por último, comentar que el número de instrucciones entre cada fallo ha aumentado; es decir, que se realiza un mayor número de instrucciones aritméticas entre cada excepción en la ejecución del algoritmo a causa de un dato que no se encuentra en memoria.

En resumen se puede ver como el tiempo de procesamiento ha disminuido, no en gran medida, pero sí para ser una modificación de las estructuras del código sencilla.

### 2.4.2 2ª aproximación

En un segundo lugar, para llevar a cabo la optimización, se recolocaron los nodos en función de su color; es decir, se agruparon los nodos totalmente independientes. Esto hace que todos los nodos que se procesarán en paralelo en los diferentes cores se encuentran alojados de forma contigua en memoria. Esta aproximación plantea un nuevo problema, porque todos los nodos tienen un vector donde se almacena el índice de los nodos a los que está conectado. Esta información es necesaria durante el proceso de la minimización de la función objetivo. Como se han cambiado las posiciones de los nodos, ahora necesitaremos una nueva estructura que nos indiquen cual es la nueva posición dentro de la malla de un determinado nodo a partir de su antiguo índice.

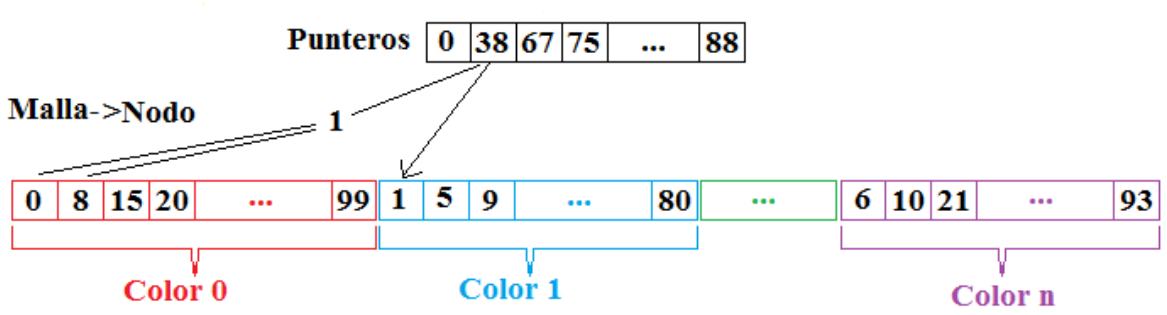


Figura 47. Ejemplo de agrupamiento en memoria después de la segunda optimización.

En la Figura 47, se puede observar como el nodo 0 y el nodo 8 que pertenecen al mismo color tienen en común el nodo número 1, pero para acceder a la información de este nodo primero se ha de acceder al vector de índices para conocer cuál es la nueva ubicación de ese nodo.

Los resultados obtenidos tras esta segunda aproximación han sido los que se incluyen en las Tablas 18 y 19. En ambos casos, se volvió a utilizar la malla denominada “Hueso” y el algoritmo Montecarlo de coloreado de la malla.

Tabla 18. Tabla de resultados de la segunda optimización en GeForce.

| | | Diferencia con la versión original del S.U.S. CUDA code |
|---|---|---|
| **Tiempo de cómputo (segundos)** | 103.09 | 2.98 |
| **Número de transacciones de almacenamiento a memoria global** | $5.99 \cdot 10^6$ | $-1.01 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $789.16 \cdot 10^6$ | $52.16 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $20.44 \cdot 10^9$ | $-3.06 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $19.13 \cdot 10^9$ | $2.3 \cdot 10^8$ |
| **Número de instrucciones entre cada fallo de memoria** | 24.24 | -1.40 |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia GeForce GT 610.

Tabla 19. Tabla de resultados de la segunda optimización en Tesla.

| | | Diferencia con la versión original del S.U.S. CUDA code |
|---|---|---|
| **Tiempo de cómputo (segundos)** | 90.13 | 4.56 |
| **Número de transacciones de almacenamiento a memoria global** | $16.29 \cdot 10^6$ | $-1.55 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $714.35 \cdot 10^6$ | $149.01 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $23.65 \cdot 10^9$ | $1.29 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $22.01 \cdot 10^9$ | $1.24 \cdot 10^6$ |
| **Número de instrucciones entre cada fallo de memoria** | 30.81 | -5.93 |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia Tesla S2050.

Como se puede observar en los resultados obtenidos, vemos como tanto el número de instrucciones (ejecutadas y enviadas) y el número de fallos de memoria cache han aumentado. Básicamente se debe a que se han tenido que añadir nuevas instrucciones para la búsqueda de las nuevas posiciones y hemos trasladado el problema de localización de los nodos a la localización de una posición dentro del vector. Aunque se trate de un fallo que supone el volcar menos información, supone un fallo igualmente, por lo que el tiempo de cómputo se ve afectado débilmente. Este empeoramiento también se puede ver reflejado en los valores de número de instrucciones entre cada fallo, donde en ambos dispositivos ha disminuido.

En resumen, se puede decir, que esta optimización ha planteado un nuevo problema de localización, por ese motivo no se han obtenido resultados del todo favorables.

### 2.4.3 3ª aproximación

Por último, en esta tercera aproximación se han modificado la información de todos los nodos actualizando el vector que indica los nodos vecinos. Se actualiza este vector, cambiando el índice antiguo de cada nodo por la nueva posición que ocupa ese vértice en la malla. De este modo ya no es necesaria esa nueva estructura añadida en la optimización número 2. Una vez, hecho esto, hemos conseguido recolocar los nodos en la malla

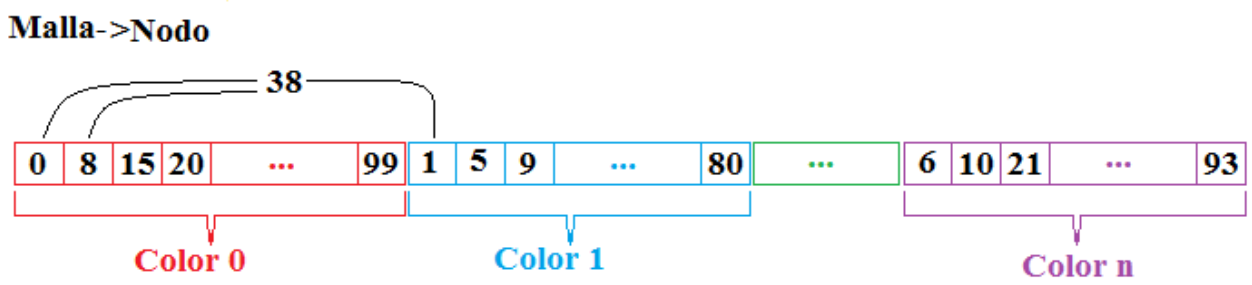


Figura 48. Ejemplo de agrupamiento en memoria después de la tercera optimización.

Como en el ejemplo anterior, el nodo 0 y el nodo 8 tienen en común el nodo 1, pero en este caso ya no es necesario saber donde está ubicado este nodo 1 puesto que ahora en lugar de ser el nodo 1 es el nodo número 'x', siendo 'x' el índice de su nueva ubicación.

Los resultados obtenidos tras esta segunda aproximación han sido los que se muestran en las Tablas 20 y 21. En ambos casos, se volvió a utilizar la malla denominada "Hueso" y el algoritmo Montecarlo de coloreado de la malla.

Tabla 20. Tabla de resultados de la tercera optimización en GeForce.

| | | **Diferencia con la versión original del S.U.S. CUDA code** |
|---|---|---|
| **Tiempo de cómputo (segundos)** | 101.06 | 0.95 |
| **Número de transacciones de almacenamiento a memoria global** | $5.99 \cdot 10^6$ | $-1.01 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $601.04 \cdot 10^6$ | $-135.96 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $20.03 \cdot 10^9$ | $-3.47 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $18.96 \cdot 10^9$ | 0 |
| **Número de instrucciones entre cada fallo de memoria** | 31.55 | +5.91 |

NOTA: Estas medidas fueron tomadas con una Nvidia GeForce GT 610.

Tabla 21. Tabla de resultados de la tercera optimización en Tesla.

| | | Diferencia con la versión original del S.U.S. CUDA code |
|---|---|---|
| **Tiempo de cómputo (segundos)** | 85.30 | -0.27 |
| **Número de transacciones de almacenamiento a memoria global** | $17.88 \cdot 10^6$ | $0.04 \cdot 10^6$ |
| **Número de fallos de carga de memoria cache L1 a memoria global** | $507.35 \cdot 10^6$ | $-57.99 \cdot 10^6$ |
| **Número de instrucciones que llegaron a la etapa de decodificación** | $22.23 \cdot 10^9$ | $-0.13 \cdot 10^9$ |
| **Número de instrucciones ejecutadas** | $20.64 \cdot 10^9$ | $-0.13 \cdot 10^9$ |
| **Número de instrucciones entre cada fallo de memoria** | 40.68 | +9.87 |

NOTA: Medidas tomadas con la malla del hueso y el coloreado de Montecarlo. GPU: Nvidia Tesla S2050.

Finalmente podemos observar, que no se produce una mejora notable tras la reubicación de los nodos. No existe gran diferencia ni en el tiempo de cómputo, ni en el número de operaciones en memoria, ni en el número de instrucciones entre cada fallo de memoria. Es decir, que no se ha logrado solucionar el problema en su totalidad.

### 2.5 Resultados

Los resultados que se han obtenido van a girar en torno a los tiempos de cómputo del desenredo y suavizado de las mallas tanto en la CPU (secuencialmente) como en la GPU (paralelamente). Por lo general se obtienen mejores resultados en lo que se refiere a tiempo en las ejecuciones realizadas en la máquina denominada “vector”. Esta mejora en comparación con la otra máquina que se ha usado se debe a las prestaciones que nos ofrece la tarjeta gráfica Nvidia Tesla S2050. Como se indicó al inicio de este documento al mostrar las características de cada GPU, se podía observar como la Nvidia Tesla S2050 nos ofrecía catorce multiprocesadores mientras que la Nvidia GeForce GT 610 solamente nos ofrecía una. A pesar de ello, la GPU de la máquina “vector” disponía de una mayor capacidad de almacenamiento, lo cual se traduce en un menor índice de fallos de acceso a datos durante la ejecución del algoritmo.

Esto quiere decir, que a pesar de haber conseguido un algoritmo lo suficientemente eficiente en cuanto al máximo aprovechamiento de los recursos hardware se refiere, también nos ofrece la portabilidad, y la posibilidad de ejecutarse sin problemas independientemente de la máquina (Nvidia) en la que se ejecute.

Sin embargo, a pesar de haber conseguido esto, los resultados en cuanto a tiempo de procesamiento de la aplicación paralela no son del todo satisfactorios con los resultados obtenidos en la versión original secuencial. Véase las siguientes Figuras 49 y 50, donde se comparaban gráficamente estos tiempos.

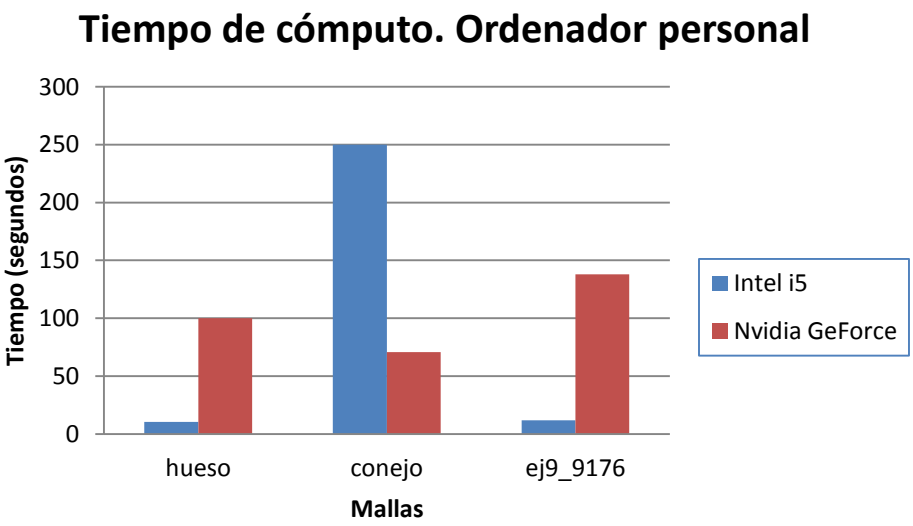


Figura 49. Comparativa de tiempo de cómputo entre CPU y GPU personales.

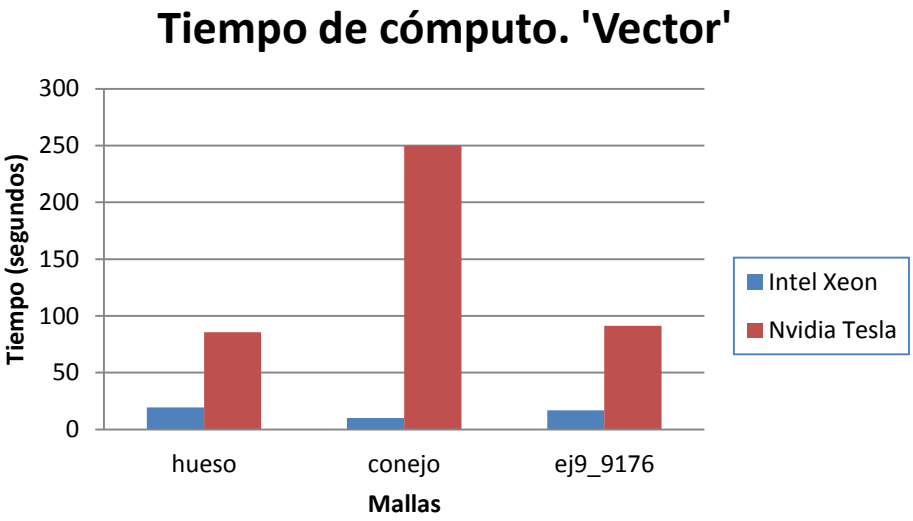


Figura 50. Comparativa de tiempo de cómputo entre CPU y GPU de 'vector'

Se puede ver en las Figuras 49 y 50 cómo los resultados no son lo suficientemente satisfactorios como se esperaban, es por ello que se hizo un análisis de prestaciones del código CUDA para averiguar dónde podría encontrarse el 'cuello de botella' y de este modo intentar subsanar el problema. Desde un punto de vista lógico se consideran problemáticos estos resultados porque conociendo las posibilidades que nos ofrecen los procesadores gráficos de tipo GPU y sabiendo la gran capacidad de paralelismo del algoritmo S.U.S., teóricamente debiera ser menor el tiempo de cómputo de un procesamiento que se realiza paralelamente a uno que se lleva a cabo secuencialmente.

Es por esto, que se hizo profiling de la ejecución paralela con el objetivo de saber cuáles habían sido los recursos utilizados a lo largo del proceso. Con estos datos se podían visualizar (tal y como se ha comentado en apartados anteriores) el número de instrucciones ejecutadas, número de transacciones en memoria, etc. Estos datos nos servirían para realizar un pequeño análisis recomendado por los desarrolladores de Nvidia denominado como 'factor instructions:bytes'. Curiosamente este valor indicaba el gran aprovechamiento de los recursos pero aún seguía siendo una incógnita el retraso en el tiempo de procesamiento de la aplicación.

Posteriormente se realizó un análisis en base a la modificación del código fuente original para ver el comportamiento de estas modificaciones en los tiempos de respuesta. Cómo era de esperar, el código de desenredo se caracterizaba por la gran cantidad de datos que usaba y la poca localización de dichos datos en memoria. Este tipo de aplicaciones trasladadas a CUDA suponen un gran problema puesto que esta arquitectura está diseñada para el tratamiento de datos secuencialmente localizados. Su filosofía se basa en la de volcar el mayor número posible de datos localizados secuencialmente, y de este modo "abastecer" los accesos de los diferentes hilos que se ejecutan en paralelo. Esta característica de CUDA unida a la falta de localización de los datos en el proceso de desenredo y suavizado de una malla, hace que la memoria tenga que volcar datos en memoria a cada instante, consumiendo así muchos ciclos de reloj y dejando ociosos a los núcleos de procesamiento.

Para intentar solucionar estos problemas de localización se implementaron tres nuevas versiones optimizadas en memoria del algoritmo paralelizado de desenredo y suavizado de mallas de tetraedros. En ellas se intentó minimizar lo máximo posible la dispersión de los datos, para aprovechar al máximo ese vuelco de información que llevaba a cabo CUDA durante el proceso de desenredo y suavizado de la malla. En estas optimizaciones se obtuvieron resultados para cada dispositivo GPU no muy distantes a los originales. En cada uno de los casos (Figuras 51, 52, 53 y 54), se muestran los resultados obtenidos en cada máquina del profiling durante el desenredo y suavizado de la malla denominada "Hueso" con el algoritmo de coloreado de la malla Montecarlo.

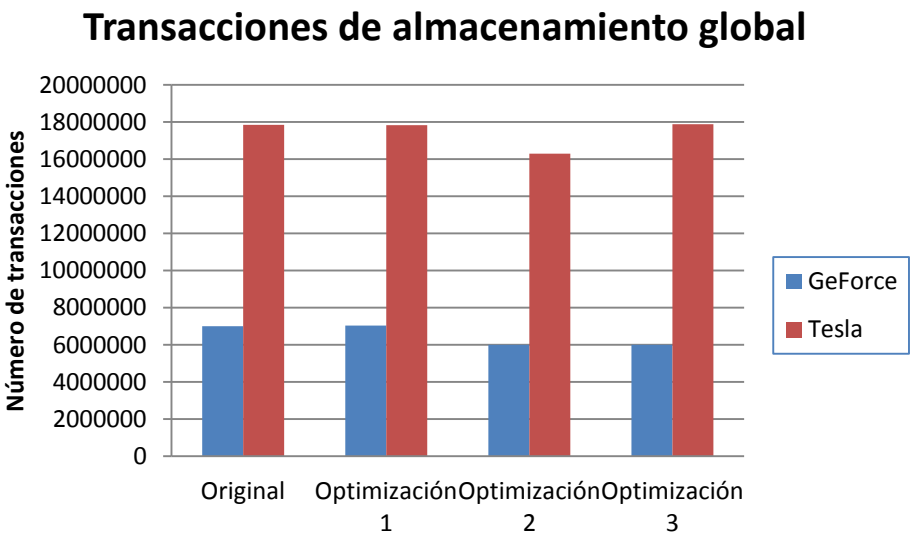


**Figura 51. Número de transacciones de almacenamiento global en ambas GPUs.**

En la Figura 51, se muestra el número de almacenamientos que se llevan a cabo en el algoritmo S.U.S. tanto en la GPU del ordenador de sobremesa como en la GPU de vector.iusiani.ulpgc.es. Se puede observar que en la GPU de **vector** el número de transacciones es notablemente mayor. Esto sucede porque el número de iteraciones que se llevan a cabo en la ejecución de la tarjeta gráfica Tesla es superior al número que se llevan a cabo en GeForce. En este caso particular, tal y como se muestran en las Tablas 8 y 9, en el dispositivo GeForce se realizan 34 iteraciones, frente a las 39 iteraciones ejecutadas en Tesla. Esto se traduce en un mayor número de instrucciones, un mayor número de reposicionamiento de los nodos y por lo tanto en un mayor número de almacenamientos.

Con respecto a la comparativa que podemos hacer en torno a los resultados obtenidos de las cuatro versiones implementadas del código S.U.S. paralelo, cabe comentar que en lo que se refiere al número de almacenamientos no se observa una variabilidad destacable.

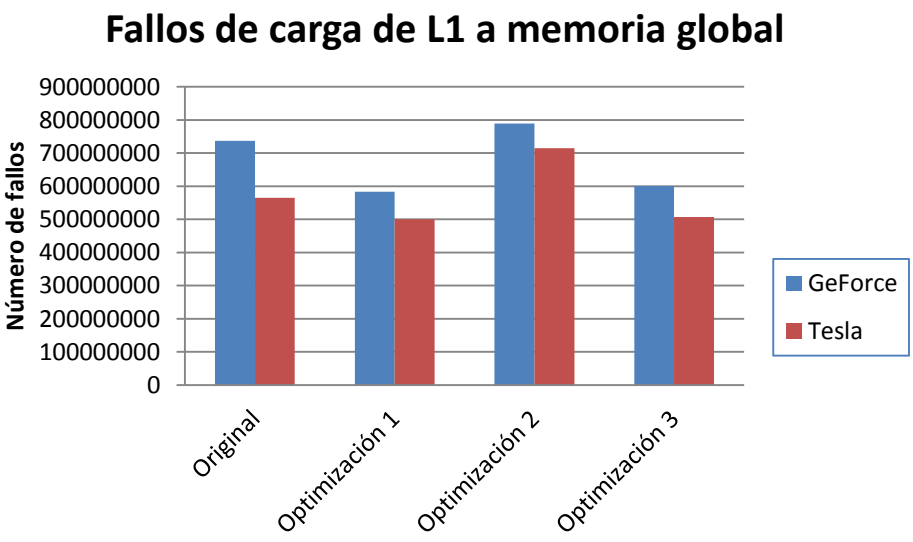


Figura 52. Número de fallos en memoria cache L1 a memoria global en ambas GPUs.

Por otro lado, con respecto al número de fallos en memoria local, se observa en la Figura 52 que en la GPU Tesla hay un menor número de fallos, aún teniendo en cuenta que realiza un mayor número de iteraciones. Esto es debido a las características de dicha tarjeta gráfica, puesto que dispone de una mayor capacidad de almacenamiento en la misma. Particularmente, tal y como se muestran en las Tablas 1 y 2, el dispositivo Tesla dispone de una memoria cache de nivel 1 así como de de una memoria cache de nivel 2 de 768 KB, sin embargo la tarjeta gráfica GeForce solamente dispone de una memoria cache de nivel 1. Esto le permite al procesador gráfico del computador **vector** almacenar mayor cantidad de información y fallar menos a la hora de acceder a un dato solicitado por un hilo.

En la Figura 52, a su vez se puede observar que al comparar la versión original del código S.U.S. paralelo con sus optimizaciones se obtienen diferencias destacables. Para el caso de las optimizaciones número uno y tres, se observa un menor número de fallos de acceso a memoria global, lo cual deriva en un mejor tiempo de procesamiento de la aplicación tal y como lo muestran las Tablas 16 y 17 para la primera optimización y las Tablas 20 y 21 para la tercera. En contrapartida observamos la optimización número dos se obtiene un mayor número de fallos durante la ejecución del algoritmo.

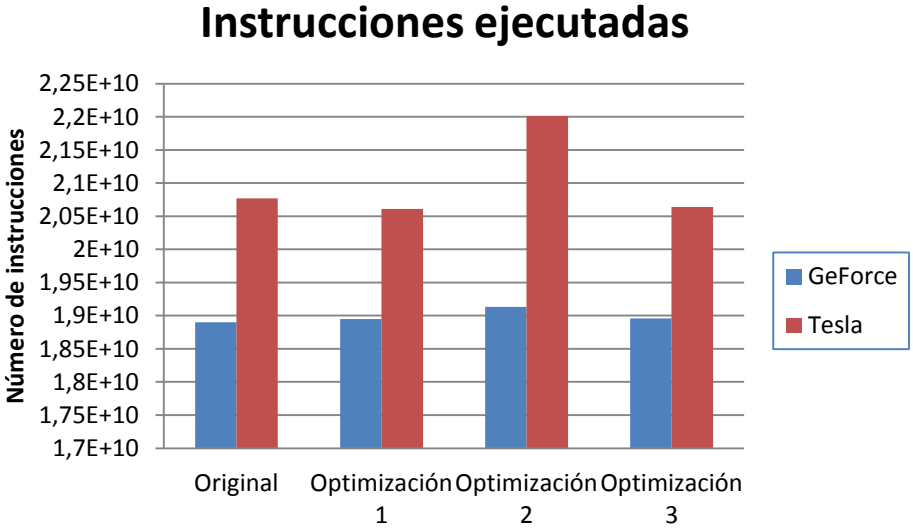


Figura 53. Número de instrucciones ejecutadas en ambas GPUs.

En la Figura 53, se puede observar que el número de instrucciones ejecutadas es mayor en la GPU Tesla que en la tarjeta gráfica GeForce. Esto es debido al mayor número de iteraciones que se ejecutan en Tesla frente a las que se ejecutan en GeForce, tal y como sucedía con el número de almacenamientos que se comentó con anterioridad.

Por otro lado, con respecto a la comparativa entre las diferentes versiones, se observa que en la optimización número dos el número de instrucciones se dispara en comparación con las demás versiones. Como se explicó en el subapartado de la segunda optimización, para esta aproximación fue necesario incorporar una estructura que enlazara los nodos con las nuevas posiciones de sus vecinos. Esta nueva estructura hizo que cada vez que se quisiese acceder a un nodo, antes de acceder al mismo debía conocer su nuevo índice en la memoria consultando esa información en un vector. Estas lecturas a memoria finalmente derivan en un mayor número de instrucciones de acceso y por lo tanto en un mayor número de fallos de memoria.

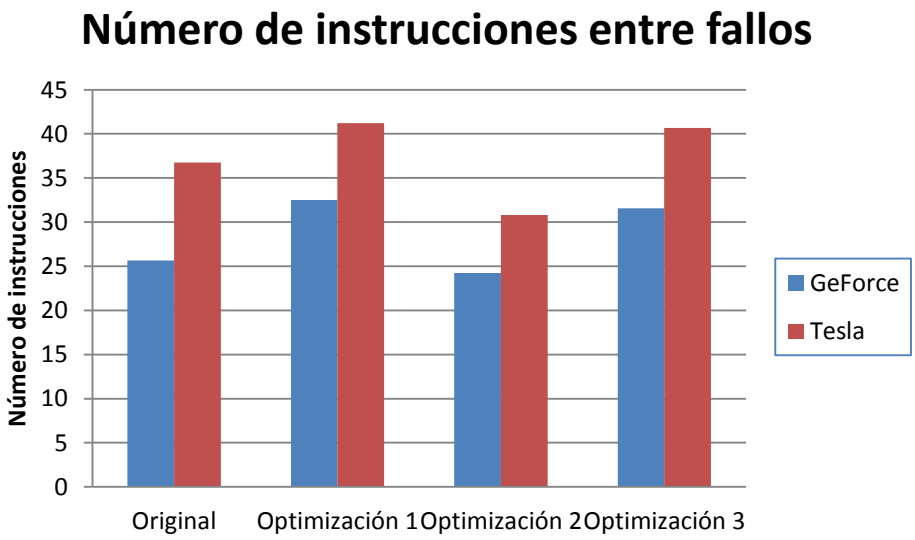


**Figura 54. Número de instrucciones entre fallos en ambas GPUs.**

Por último, comentar la gráfica del número de instrucciones entre fallos en las distintas ejecuciones de las diferentes versiones implementadas (ver Figura 54). Se puede observar como los valores de la tarjeta gráfica de Tesla son mayores en todas las versiones debido a la mayor capacidad de almacenamiento en comparación con la GPU GeForce. De este modo el dispositivo Tesla evita fallos y vuelco de datos desde memoria global, que la tarjeta grafica GeForce no puede pasar por alto.

En lo que a la comparativa de los resultados de la Figura 54 se refiere, se puede observar que los mejores resultados obtenidos de número de instrucciones entre fallos los vemos en las optimizaciones número uno y tres. Por lo tanto, esta Figura muestra gráficamente las mejoras introducidas en el código S.U.S. paralelizado tras realizar estas optimizaciones. A pesar de no haber obtenido grandes mejoras en cuanto al tiempo de cómputo, la Figura 54 muestra que se han mejorado algunos aspectos importantes en dichas versiones.

Abriendo un poco más el abanico de posibilidades, e intentando buscar algún tipo de mejora realizando diferentes modificaciones en diferentes partes del código de desenredo y suavizado, en primer lugar se decidió utilizar el antiguo algoritmo de coloreado de la malla. El algoritmo con el que se llevó a cabo este pequeño estudio fue el algoritmo Montecarlo. La versión antigua de este algoritmo de coloreado implementado hacía uso de un mayor número de colores (concretamente 72 colores en la malla denominada ‘Hueso’, siendo 19 colores los necesarios en el algoritmo

actualizado). En las Figuras 55 y 56 se muestran los resultados que se obtuvieron con la malla denominada “hueso” y usando el algoritmo Montecarlo antiguo y el mismo algoritmo actualizado en ambas GPUs:

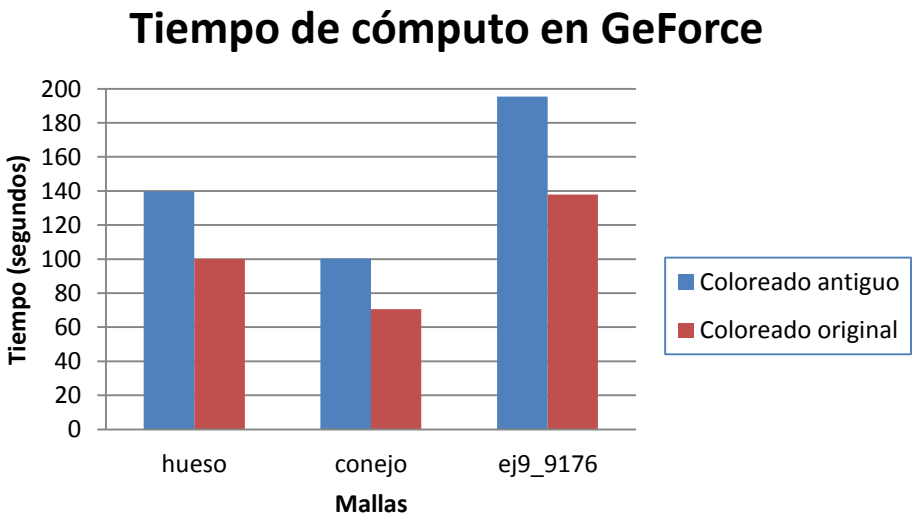


Figura 55. Comparativa de tiempos con diferentes coloreados en GeForce.

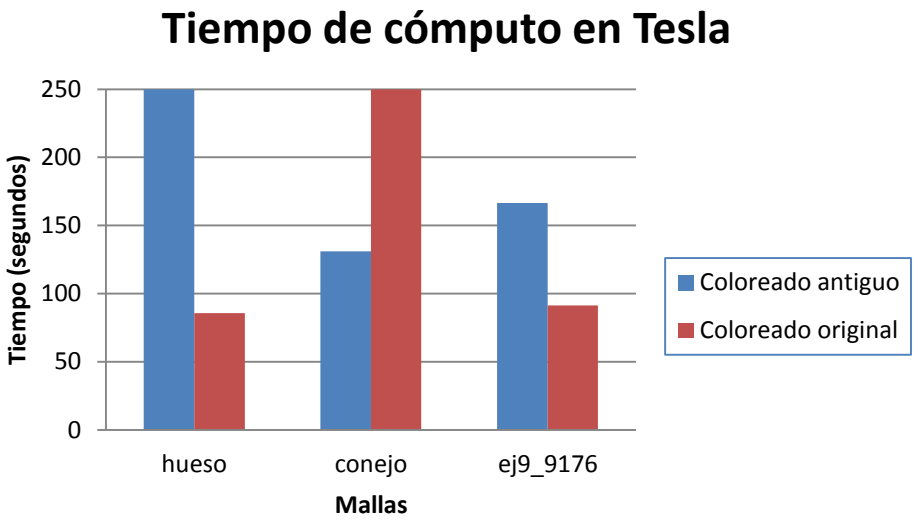


Figura 56. Comparativa de tiempos con diferentes coloreados en Tesla.

Se puede observar que al usar un coloreado con mayor número de colores y donde cada color consta de un menor número de nodos, esto en CUDA se traduce en un mayor número de kernel lanzados (un kernel por color) donde cada kernel consta de un menor número de hilos. Esto hace que se lancen muchos kernels y que en cada kernel sólo trabajen unos pocos núcleos de procesamiento, estando el resto ocioso y desaprovechando el gran número de recursos que nos ofrece la GPU. Es por esto que se obtienen peores resultados.

Otra línea de investigación que se llevó a cabo, fue la de lanzar varias veces el algoritmo de desenredo y suavizado de mallas para todos los nodos de un color, en donde se cambiaban los parámetros del kernel. Es decir, se modificó el número de hilos por bloque, cambiándose automáticamente así el número de bloques necesario, queriéndose observar cuál era la mejora o el empeoramiento del tiempo de cómputo; entre otras cosas.

Cuando se realiza una configuración de hilos por bloque, si el número de datos a procesar no es múltiplo del número de hilos por bloque, siempre habrá que lanzar un bloque que no aprovechará todos los recursos del hardware. Es decir, supongamos que se quiere procesar 2370 nodos, si se agrupan 32 hilos en cada bloque, se necesitará lanzar 75 bloques (2370 / 32 = 74.06). Al lanzar estos 75 bloques, se están lanzando 30 hilos de más (32 · 75 – 2370 = 30).

Con este ejemplo se comprende mejor el dilema de la configuración de los hilos por bloque. Si se agrupan pocos hilos en un bloque se desaprovecharán recursos en cada lanzamiento de un bloque pero se lanzarán menos hilos de más. Sin embargo, si se lanzan muchos hilos por bloque, se aprovechan en mayor medida los recursos del hardware en cada multiprocesador, excepto en el último, donde muchos de los cores estarán ociosos, por ese exceso de hilos lanzados.

A continuación se muestran la optimización de 2370 nodos independientes entre sí, correspondientes al color número 0 de la malla denominada 'Hueso' tras ser aplicado el coloreado Montecarlo (actualizado). Estos resultados se muestran en las Figuras 57, 58 y 59, los cuales fueron tomados en la tarjeta gráfica Nvidia GeForce GT 610.

Antes de nada hay que comentar que los puntos señalados en las gráficas, (cuadrados azules y triángulos rojos) indican cuál es la mejor configuración en función del tiempo de cómputo.

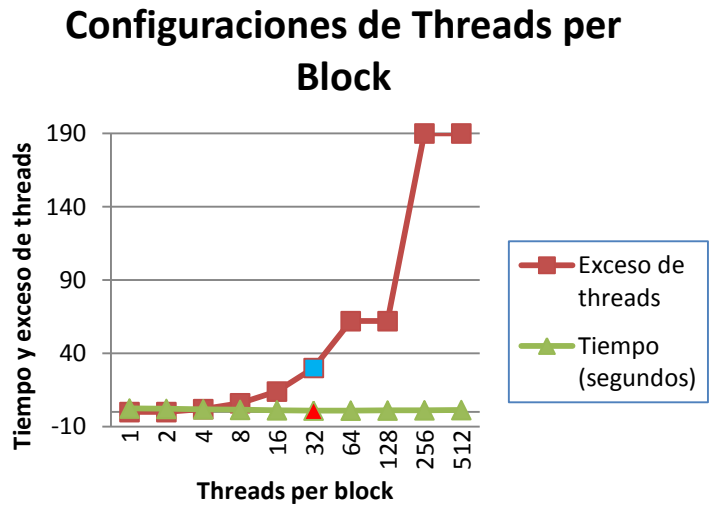


Figura 57. Configuración de hilos por bloque en una sola dimensión.

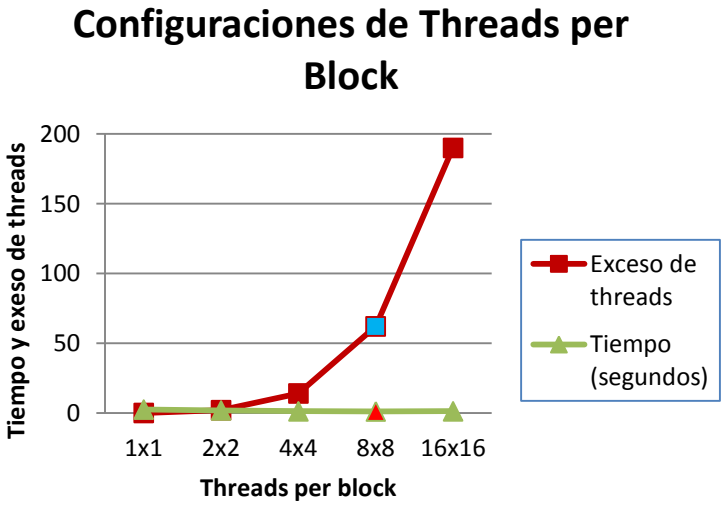


Figura 58. Configuración de hilos por bloque en dos dimensiones.

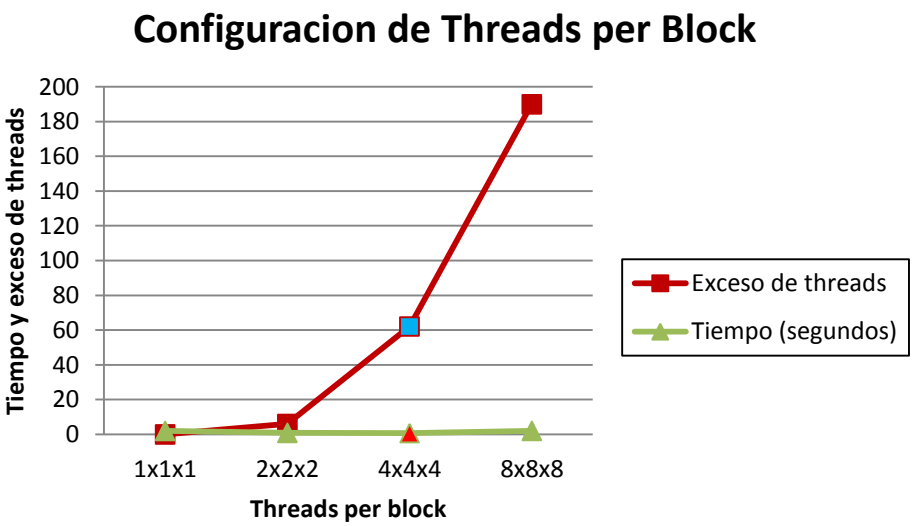


**Figura 59. Configuración de hilos por bloque en tres dimensiones.**

En la Figura 57 se puede observar que la mejor configuración obtenida con respecto al tiempo de procesamiento para bloques de hilos monodimensionales corresponde con el valor 32. Resulta lógico que este valor sea el mejor puesto que CUDA está diseñada para que en un momento determinado se ejecute un warp (conjunto de 32 hilos) en un multiprocesador. Se puede observar a su vez en las Figuras 58 y 59 que las configuraciones bidimensionales y tridimensionales el valor total de hilos por bloque resultan ser múltiplos de 32:

$$(8 \cdot 8 = 64;\ 64 / 32 = 2 \parallel 4 \cdot 4 \cdot 4 = 64;\ 64 / 32 = 2)$$

Curiosamente las mejores configuraciones son aquellas que el número total de hilos por bloques es múltiplo de 32 y cuyo valor sea menor. Es decir, para la configuración de hilos por bloque tridimensional ($8 \cdot 8 \cdot 8 = 512$) el valor total también es múltiplo de 32, pero se obtienen peores resultados.

La idea principal que se intenta plasmar en las gráficas anteriores, es la de justificar cuál ha sido el motivo de llevar a cabo una configuración de hilos por bloque y no otra. También se muestran con la intención de que al configurar un kernel se debe intentar es llegar a un equilibrio en la configuración del número de hilos por bloque para obtener los mejores resultados (mejor tiempo de cómputo señalado en las gráficas) y desaprovechar o sobresaturar lo menos posible los recursos hardware.

Por último, se decidió observar el comportamiento del tiempo en el código CUDA para el desenredo y suavizado de la malla. Durante el proceso del desenredo y suavizado se usan dos funciones principales que se ejecutaban en la GPU, una función que se encargaba de reubicar los nodos y otra de calcular las calidades de los tetraedros resultantes de esa reubicación. La primera función es la que se encarga de desenredar y suavizar la malla. Y la otra, se encarga de analizar todos los tetraedros que forman la malla y comprobar las calidades de cada uno de ellos, para decidir si la malla se considera por optimizada o no, y de este modo detener el proceso de la aplicación o no, respectivamente.

En la Figura 60 se muestran los porcentajes de tiempo consumido de cada función con respecto al tiempo total de la aplicación. Los datos mostrados en la Figura 60 son los obtenidos durante el desenredo y suavizado de la malla denominada “hueso” con el algoritmo de coloreado de la malla Montecarlo en la GPU Nvidia GeForce GT 610.

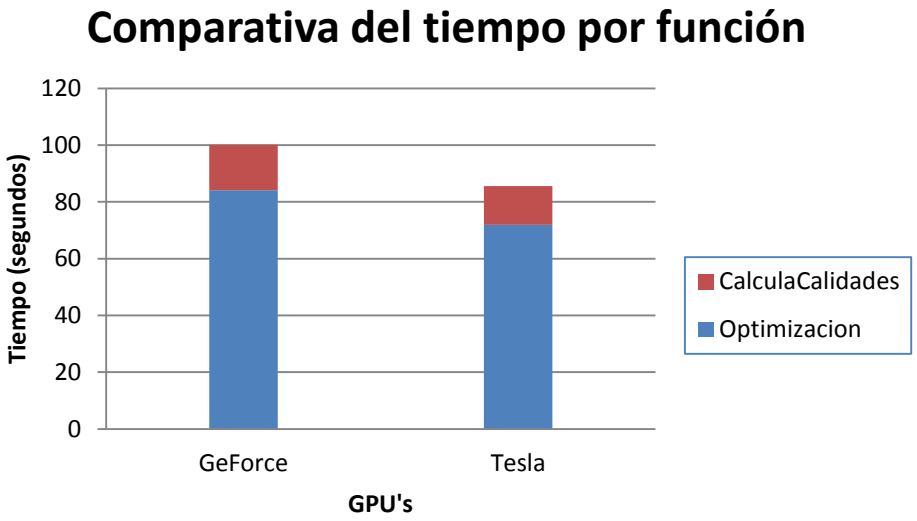


**Figura 60. Comparativa del tiempo consumido por las dos funciones principales en el algoritmo de CUDA.**

En la gráfica anterior de la Figura 60 se puede observar cómo el cálculo de las calidades consume entorno al 16 % - 17 % del tiempo total que necesita la aplicación para llevar a cabo el desenredo y suavizado de una malla. Con estos resultados se observa que la gran mayoría del tiempo consumido durante la aplicación se necesita para el desenredo y suavizado paralelo en la GPU de una malla de tetraedros.

# 3. Conclusiones y líneas futuras

## 3.1 Conclusiones

En primer lugar hay que indicar que se ha conseguido el reto que se propuso en un principio que consistía en diseñar e implementar el algoritmo de desenredo y suavizado de mallas de tetraedros para su ejecución paralela en la GPU. Dicha implementación optimiza las mallas de tetraedros con las que se han realizado las pruebas (malla del 'hueso', 'conejo' y 'ej9_9176') del mismo modo que lo hace el algoritmo S.U.S. original secuencial. Por lo tanto se ha cumplido el segundo de los objetivos propuestos en la lista de objetivos que se expuso en el primer capítulo de este documento.

Una vez llevada a cabo la implementación del S.U.S. se ha realizado un análisis de prestaciones tanto en la versión del algoritmo que se ejecuta en la GPU como en la CPU. Con los resultados obtenidos se ha realizado un análisis del rendimiento, comparando los resultados que se han mostrado a lo largo de este documento. Con esto se ha cumplido el tercer objetivo propuesto.

En base a los resultados obtenidos en el análisis se ha diseñado e implementado tres aproximaciones a soluciones del algoritmo paralelizado en GPU. Tras la implementación de cada una de estas optimizaciones se ha llevado a cabo un análisis de prestaciones para poder deducir de dónde podían derivar algunos problemas y de este modo orientar el resto de optimizaciones. Con esto se ha cumplido el cuarto de los objetivos propuestos.

A lo largo de las implementaciones se han solucionado múltiples problemas que han surgido a la hora de trasladar un algoritmo en C++ para g++ a un algoritmo paralelo en CUDA compilado con nvcc. Así como también se ha realizado un estudio minucioso de las posibilidades de la herramienta CUDA haciendo uso del profiler y otras herramientas. Con esto se cumple el primer objetivo y con él, todos los objetivos propuestos en el proyecto fin de carrera.

Como conclusiones referentes a los resultados obtenidos del algoritmo S.U.S. paralelizado en GPU cabe destacar que se trata de un programa candidato a tener mejores resultados en GPU que en CPU. Esto se debe a la gran capacidad de paralelismo que nos ofrece el algoritmo y al elevado número de elementos que pueden ser procesados de forma independiente.

Esto sin embargo, no se ha conseguido a pesar que se han realizado grandes esfuerzos para aprovechar al máximo los recursos del hardware. Principalmente el problema que impide no obtener mejores resultados en el algoritmo paralelizado en la GPU se debe a la dispersión de los datos en memoria. En las optimizaciones propuestas se han recolocado los datos en la memoria mejorando los accesos a la misma pero el núcleo del algoritmo donde se realizan la mayoría de los accesos a memoria no se ha modificado. Este núcleo es el responsable de que no se obtengan tan buenos tiempos de la aplicación como se debería.

## 3.2 Líneas futuras

Este es un problema englobado en los algoritmos de tratamiento de grafos en los que se está trabajando actualmente por parte de la comunidad científica que trabaja en GPU. El principal problema reside en que resulta muy difícil de almacenar secuencialmente en memoria los nodos conectados entre sí. Puesto que cada nodo está conectado a unos vecinos y éstos a su vez con tantos otros (tal y como sucede en el tratamiento de grafos), el agrupar la información de esta estructura a una mono-dimensional como lo es la memoria no resulta sencillo.

Buscar estructuras de datos para mallas no estructuradas, que permitan optimizar la ejecución de programas de este tipo en GPU es un problema abierto que se está estudiando actualmente. Las

mallas no estructuradas son aquellas que no se sabe a priori cuántos vecinos tiene cada nodo (por lo tanto, no se sabe a cuántos tetraedros pertenece ese nodo). Al no tener esta información y teniendo en cuenta las posibilidades de conectividad entre nodos de toda la malla, la agrupación en memoria de esta información para ahorrar accesos no es un problema trivial.

Por otro lado, como trabajo futuro para mejorar el tiempo de respuesta de la aplicación podría ser la de paralelizar el procedimiento del cálculo de las calidades. Este proceso que forma parte del algoritmo de desenredo y suavizado de una malla de tetraedros, conforma entre el 16 y 17 % del tiempo total de la aplicación, tal y como se comentó en el capítulo anterior. De modo que mejorar el tiempo de respuesta de este procedimiento implicaría una mejora general del algoritmo S.U.S.

## 4. Bibliografía

Se ha hecho uso de gran cantidad de páginas webs, tanto especializadas como para principiantes en cuanto a CUDA. Puesto que se ha indagado bastante también en la búsqueda y comprensión del algoritmo de desenredo y suavizado simultáneo de mallas y sus aplicaciones se han usado algunos artículos importantes. Es por esto que las diferentes fuentes se han categorizado:

S.U.S. code


- **Autores**: J. M. Escobar, J. M. González-Yuste, R. Montenegro, G. Montero, E. Rodríguez
  **Título**: Simultaneous untangling and smoothing of tetrahedral meshes
  **Revista**: Comput. Meth.in
  **Volúmen**: 192
  **Páginas:** 2775-2787
  **Año:** 2003

- **Autores**: J. M. Escobar, J.M. Cascón, R. Montenegro, E. Rodríguez
  **Título**: The Meccano Method for Isogeometric Solid Modeling
  **Libro**: Proceeding of the 20th International Meshing Roundtable
  **Volumen**: -
  **Páginas:** 551-568
  **Año:** 2012

- **Autores**: J. M. Escobar, J.M. Cascón, R. Montenegro, E. Rodríguez, G. Montero
  **Título**: An automatic Strategy for Adaptive Tetrahedral Mesh Generation
  **Libro**: Appl Num Math
  **Volumen**: 59
  **Páginas:** 2203-2217
  **Año:** 2009

- **Autores**: L. Freitag, M. Jones, P. Plassmann
  **Título**: A parallel algorithm for mesh smoothing
  **Libro**: SIAM J. Sci. Comput
  **Volumen**: 20
  **Páginas:** 2023-2040
  **Año:** -

- **Autores**: U.V. Catalyurek, J. Feo, A.H. Gebremedhin, M. Halappanavar, and A. Pothen
  **Título**: Graph Coloring Algorithms for Multi-core and Massively Multithreaded Architectures.
  **Libro**: Parallel Computing
  **Volumen**: 38
  **Número**: 10-11
  **Páginas:** 576-594
  **Año:** Octubre-Diciembre 2012

- **Autores**: Jones, M.T., Plassmann
  **Título**: A parallel graph coloring heuristic
  **Revista**: SIAM J. Sci. Comput
  **Volumen**: 14
  **Páginas:** 654-669
  **Año:** 1993

- **Autores**: Freitag, L., Jones, M., Plassmann, P
  **Título**: A Parallel Algorithm for Mesh Smoothing
  **Revista**: SIAM J. Sci. Comput
  **Volumen**: 20
  **Páginas:** 2023-2040
  **Año:** 1999

CUDA

- **Empresa**: Nvidia Corporation
  **Título**: CUDA ToolKit Documentation
  **Enlace**: https://www.clear.rice.edu/comp422/resources/cuda/html/index.html
  **Accedido**: Noviembre 2013

- **Corporación**: Georgia Tech Hotel and Conference Center
  **Título**: NSF Keeneland GPU Tutorial
  **Enlace**: http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/
  **Accedido**: Noviembre 2013

- **Autor/es**: Michael Wolfe
  **Título**: Understanding the CUDA Data Parallel Threading Model
  **Enlace**: http://www.pgroup.com/lit/articles/insider/v2n1a5.htm
  **Accedido**: Noviembre 2013

- **Corporación**: Carnegie Mellon University
  **Título**: GPU Architecture & CUDA Programming
  **Enlace**: http://15418.courses.cs.cmu.edu/spring2013/lecture/gpuarch
  **Accedido**: Noviembre 2013

- **Autor/es**: Cliff Woolley
  **Título**: Advanced Topics in CUDA
  **Enlace**: http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/12-advanced_topics_in_cuda.pdf
  **Accedido**: Noviembre 2013

- **Autor/es**: Sergio Orts Escolano, Vicente Morell Giménez
  **Título**: Introducción a la computación paralela con GPUs
  **Enlace**: http://www.dtic.ua.es/jgpu11/material/sesion1_jgpu11.pdf
  **Accedido**: Noviembre 2013

- **Autor/es**: José Miguel Mantas Ruiz
  **Título**: Programación de GPUs
  **Enlace**: http://lsi.ugr.es/jmantas/PGAP/pgap4.1.pdf
  **Accedido**: Noviembre 2013

- **Autor/es**: Paulius Micikevicius
  **Título**: Analysis-Driven Optimization
  **Enlace**: http://www.nvidia.com/content/GTC-2010/pdfs/2012_GTC2010.pdf
  **Accedido**: Noviembre 2013

- **Autores**: David B. Kirk, Wen-mei W. Hwu
  **Título**: Programming Massively. Parallel Processors.

**Libro**: SIAM J. Sci. Comput
**Volumen**: Segunda Edición
**Enlace:** http://www.amazon.com/Programming-Massively-Parallel-Processors-Edition/dp/0124159923
**Accedido**: Noviembre 2013

Especificaciones de las arquitecturas

- **Corporación**: VideoCardZ.com
  **Título**: Nvidia GeForce GT 610 Specifications
  **Enlace**: http://videocardz.com/nvidia/geforce-600/geforce-gt-610
  **Accedido**: Noviembre 2013

- **Corporación**: Nvidia Corporation
  **Título**: Fermi Architecture
  **Enlace**: http://www.nvidia.com/object/fermi-architecture.html
  **Accedido**: Noviembre 2013

- **Corporación**: AnandTech
  **Título**: Nvidia Goes Superscalar
  **Enlace**: http://www.anandtech.com/show/3809/nvidias-geforce-gtx-460-the-200-king/2
  **Accedido**: Noviembre 2013

- **Autor/es**: Manuel Ujaldón
  **Título**: Inside Kepler
  **Enlace**: http://www.jornadassarteco.org/js2012/papers/inside-Kepler.pdf
  **Accedido**: Noviembre 2013

# 5. Anexo

En este último capítulo se comentará como se pueden instalar las librerías PAPI así como el ToolKit de CUDA completo, de este modo se aborda en un marco teórico un pequeño capítulo práctico, ilustrativo y auxiliar.

## 5.1 Instalación del ToolKit de CUDA

A continuación se explicará cómo podemos instalar el ToolKit de CUDA, así como los drivers de NVIDIA y otros paquetes necesarios en Linux.

**Requisitos**

Lo primero que necesitamos para usar CUDA es una tarjeta gráfica de NVIDIA compatible. Para saber si una tarjeta gráfica es o no compatible, podemos consultar la lista de tarjetas compatibles con CUDA en: https://developer.nvidia.com/cuda-gpus

El resto del tutorial supone que estamos usando Ubuntu (12.10) aunque debería servir para las distribuciones Debian similares.

**Instalación**

La instalación completa consta de los siguientes pasos:

- Descarga de los controladores de NVIDIA y ToolKit de CUDA.
- Instalación del controlador de la tarjeta gráfica NVIDIA.
- Instalación del ToolKit de CUDA.

*Descarga de los controladores de NVIDIA y ToolKit de CUDA.*

Los controladores de la tarjeta los podemos descargar de la página de NVIDIA, indicando el tipo de tarjeta, el modelo de la misma, el sistema operativo donde se instalará (en este caso Ubuntu 12.10) y el idioma.

http://www.nvidia.es/Download/index.aspx?lang=es

El ToolKit de CUDA lo podemos encontrar en la página de NVIDIA:

https://developer.nvidia.com/cuda-downloads

*Instalación del controlador de la tarjeta gráfica NVIDIA*

Una vez hayamos descargado lo indicado en el apartado anterior y sabiendo donde lo tenemos almacenado, en primer lugar instalaremos algunas librerías de las que se dependerá más adelante.

```
# sudo apt-get install build-essential freeglut3-dev libxi-dev
  libxmu-dev mpich2
```

Posteriormente enlazaremos la librería '*libglut.so*' donde el instalador del controlador lo espera encontrar.

```
# sudo ln –s /usr/lib/x86_64-linux-gnu/libglut.so
  /usr/lib/libglut.so
```

En segundo lugar, debemos entrar en el modo consola para detener los procesos gráficos KDE, GNOME o XDM.
Para entrar en modo consola debemos pulsar:

[Ctrl] + [Alt] + [F1]

Una vez aquí, nos identificamos como '*root*' e introducimos la contraseña para detener dichos procesos:

```
# /etc/init.d/lightdm stop
```

En cuanto hayamos detenido el proceso de la interfaz de Linux; a continuación, buscaremos los paquetes de NVIDIA instalados por defecto en la instalación del sistema operativo y los desinstalaremos. Para la búsqueda y eliminación de paquetes utilizaremos el 'aptitude', que hemos de instalar.

```
# apt-get install aptitude
# aptitude search nvidia | grep –E ^i
```

Después de lanzar la última línea de comandos obtendremos una serie de paquetes (3 normalmente), los cuáles deberemos desinstalar a continuación.

```
# aptitude remove PACKAGE-NAME
```

Posteriormente a la desinstalación de todos los controladores gráficos, nos situamos en el directorio donde tengamos almacenado los controladores descargados e instalaremos los controladores propios de nuestra tarjeta gráfica.

```
# cd PATH-WHERE-WE-HAVE-THE-INSTALLER
# chmod +x NVIDIA-Linux-x86-319.17.run
# sh NVIDIA-Linux-x86-319.17.run
```

Actualizamos el 'path' y el entorno de las librerías instaladas para que sirvan a todos los usuarios.

```
# vi /etc/environment
# Y añadimos /usr/local/cuda-5.0/bin
# Ahora creamos el fichero /etc/ld.so.conf.dcuda.conf
# vi /etc/ld.so.conf.d/cuda.conf
# Y añadimos /usr/local/cuda-5.0/lib
             /usr/local/cuda-5.0/lib64
```

Finalmente lanzamos el configurador del sistema y reiniciamos el sistema.

```
# ldconfig
# reboot
```

Una vez se haya reiniciado el sistema ya tendríamos instalado el controlador correctamente.

*Instalación del ToolKit de CUDA.*

La instalación del ToolKit será más sencilla, simplemente debemos alojarnos en el directorio donde tengamos el instalador, lanzar dicho lanzador y aceptar todos los términos, condiciones y directorios donde queremos alojar las librerías.

```
# cd PATH-WHERE-WE-HAVE-THE-INSTALLER
# sudo chmod +x cuda-5.0.35_linux_64_ubuntu12.10.run
# sudo sh cuda-5.0.35_linux_64_ubuntu12.10.run
```

Una vez hayamos terminado la instalación, para comprobar que se ha instalado todo correctamente, podemos ejecutar los ejemplos que vienen de prueba con el ToolKit.

```
# cd /usr/local/cuda-5.0/samples
# Compilamos todos los ejemplos
# sudo make
# cd 1_Utilities/deviceQuery
# Ejecutamos un ejemplo que muestra las características de la GPU
# ./deviceQuery
```

Si se nos muestra en la línea de comandos la lista de características de nuestra tarjeta gráfica, ya podemos empezar a desarrollar en CUDA.

### 5.2 Instalación de las librerías PAPI

Para la instalación de las librerías de PAPI, no supondrá un gran problema, puesto que en antiguas versiones de plataformas como Debian debíamos parchear el kernel, pero los kernel 3.0 y superiores ya vienen con dicho parche. Como estamos trabajando en un Ubuntu 12.10 que ya viene con el kernel 3.2.0 no debemos preocuparnos por tener que recompilar el núcleo.

En primer lugar debemos descargarnos la librería desde su página oficial:

http://icl.cs.utk.edu/papi/software/

Una vez descargado, descomprimimos el paquete.

```
# cd PATH-WHERE-WE-HAVE-THE-PACKAGE
# gzip -d papi-5.1.0.tar.gz
# tar -xvf papi-5.1.0.tar
```

A continuación procedemos a configurar los componentes de CUDA de la librería PAPI.

```
# cd papi-5.1.0/src/components/cuda
# ./configure -with-cuda-incdir=/usr/local/cuda-5.0/include
              -with-cupti-incdir=/usr/local/cuda-5.0/extras/CUPTI
/include
              -with-cupti-libdir=/usr/local/cuda-5.0/extras/CUPTI
/lib ó lib64 según si estamos en una arquitectura de 32 o 64 bits
```

Después de configurar los componentes de CUDA, configuraremos la instalación de PAPI y compilamos.

```
# cd papi-5.1.0/src
# ./configure --prefix=/opt/PAPI
              --with-perf-events
              --with-components=cuda
# make
```

Posterior a la compilación de la librería PAPI, antes de realizar la instalación exportamos la librería CUPTI y finalmente instalamos la librería completa con los test y pruebas.

```
# export LD_LIBRERY_PATH=/usr/local/cuda-5.0/extras/CUPTI/lib o
lib64:$LD_LIBRARY_PATH
# make fulltest
# make test
# sudo make install-all
```

Una vez instalado la librería ya está lista para ser usada en nuestros códigos.

### 5.3 Guía de usuario

Este código es una implementación paralelizada del método de suavizado y desenredo simultáneo de mallas de tetraedros. El S.U.S. code obtiene una malla optimizada de tetraedros partiendo de una malla enmarañada, mediante el uso de la reubicación del nodo local, mantenimiento la topología de la malla inicial.

**Compilación**

Como se ha comentado, el código se ha desarrollado en un entorno de Debian Linux (Ubuntu 12.04), utilizando g++ versión 4.7. Se debe compilar y ejecutar en cualquier entorno similar a Linux. Es aconsejable que se consulte las siguientes variables de Makefile y modificarlas de acuerdo a las rutas de su sistema:

CUDA_PATH: ruta de acceso a los archivos del ToolKit de CUDA.

**Ejecución**

Para ejecutar el programa, el usuario debe proporcionar la información de la malla inicial que debe definir con dos argumentos obligatorios en la línea de comandos:

-n fnodes: archivo de nodos
-e felems: archivo de elementos (tetraedros)

El programa tiene los argumentos opcionales siguientes:

-**fsmooth**: archivo de salida con los nodos de malla optimizada (devuelve sólo el archivo nodo modificado, mientras que el archivo de elementos se mantiene sin modificación).

-**fstats**: archivo de salida con los datos estadísticos, con el contenido de las mejores y peores calidades, así como la media de la malla y el número de elementos enredados en cada iteración.

-**d**: acoplamiento de entrada, el cual normaliza de forma predeterminada. Use **-d** para evitar la normalización.

-**qprefix**: salida de un archivo de calidad en cada iteración. El nombre del archivo es qprefix + iteration_number.

-**max_iter**: El número máximo de iteraciones de optimización de la malla, incluyendo tanto desenredado como suavizado (50 iteraciones por defecto).

-**smooth_iter**: Número de la iteración de suavizado después de que la malla este desenredada (20 iteraciones por defecto).

**Formato de los archivos de definición de la malla**

Con el fin de definir una malla, es necesario definir dos archivos ASCII:

1) archivo de nodos: ubicación de los nodos de la malla

La primera línea no comentada del fichero es el número total de nodos de la malla.
Después de esta línea, cada nodo de la malla es listado, uno por línea con el siguiente formato:

x y z ref

donde x, y, z son las coordenadas del nodo y ref es una referencia designada con un número entero que indica la zona a la que pertenece ese nodo dentro de la malla. El procedimiento sólo modificará los nodos con número de referencia igual a 0 que determina a los nodos internos de la malla.

2) archivo de elementos: contiene información acerca de tetraedros

La primera línea de comentario no comentada contiene el número de elementos de la malla. A continuación, todos los elementos de la malla se debe especificar de la siguiente manera, cada elemento por línea obedeciendo al siguiente formato:

0 n0 n1 n2 n3

donde 0 es un número de referencia (irrelevante para nuestros propósitos) y n0, n1, n2 y n3 son los números de los nodos del elemento. La numeración de los nodos está directamente relacionada con los nodos determinados en el archivo de nodos, empezando la numeración con el 1.

Por ejemplo, teniendo en cuenta los siguientes archivos:
nodes:
# This is an optional comment line (begins with #)
5
0.0 0.0 0.0 1
1.0 0.0 0.0 1
0.0 1.0 0.0 1
0.0 0.0 1.0 0
0.0 0.0 -1.0 0

elements:
# This is an (optional) comment line (begins with #)
2
0 0 1 2 3

0 0 2 1 4

Tenga en cuenta que los elementos deben tener una orientación positiva (siguiendo lo determinado por el artículo [1]). Ambos archivos del nodo y elemento puede tener comentarios. Esas líneas que comienzan con el carácter # se ignoran.