**Universidad de Valladolid**

E.T.S Ingeniería Informática

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática
Mención en Tecnologías de la Información

# Aplicación de las estrategias de paralelización de algoritmos

Autor:
**D. David Burgos Domínguez**

**Universidad de Valladolid**

E.T.S Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Tecnologías de la Información

# Aplicación de las estrategias de paralelización de algoritmos

Autor:
**D. David Burgos Domínguez**

Tutor:
**D. Iván Santos Tejido**

*A María José, por su eterno compromiso, por ser infinita,*

*A Gabriel, por querer lo mejor de mí (aunque no me lo diga),*

*A Jorge, por aguantarme con paciencia y ser perspicaz, y*

*A Mario, por sus rápidas respuestas, tan sencillas como efectivas.*

*A Roxana, por su gran fuerza en un momento difícil de mi vida.*

*A Jenni, Julio y Omar, por dar cabida a alguien más, sin saber quién podría ser.*

*Escogemos, o nos vemos escogidos, a comenzar ciertos caminos.*

*Los iniciamos solos, y quizás, y solo quizás, en algún momento dejemos de estarlo.*

*Al final, el camino termina, y lo que queda son… los recuerdos.*

*Agradecer a mi tutor, Iván, su apoyo durante el transcurso del trabajo,*
*dándome ideas de cómo proseguir, entendiendo mi situación en cada momento,*
*y ayudando a que todo salga aún mejor.*

// Abstract

The efficiency is an important property in the execution of software. Giving priority to this value can improve the efficacy and the performance of the obtaining of results.

This project has approached the use of the parallel paradigm to adapt a software, which original execution is serial, to its corresponding parallel version. To achieve it, it has been identified the more execution time consuming components, as well as the susceptible of being parallelized, so later it has been executed a series of technical iterations using two parallel models available in the market.

These parallelizations employ general purpose devices, such as the CPUs, and specific purpose devices, as some GPUs disposed to the scientific development, using the OpenMP and CUDA models, respectively.

During the investigation, it has been able to inquire on the potential of the parallel paradigm applied over the software through of the two computing models, different among them, but which final target is the same: explode the capacities of the actual hardware, eminently parallel. By this way, the behavior of the software, as the required time, have been optimized.

The development, implementation, parallelization and testing of the software has resulted in the improvement of its processing capacities, as in a better performance achieved by unit of time. The obtained software not it only as effective as the original, but also the efficiency and utility are better.

# Resumen

La eficiencia es una propiedad importante en la ejecución de software. Dar prioridad a este valor permite aumentar la eficacia y el rendimiento en la obtención de resultados.

En este trabajo se ha abordado el empleo del paradigma paralelo para adaptar un software, cuya ejecución original es serie, a su correspondiente versión paralela. Para ello, se han identificado los componentes que mayor tiempo de ejecución consumen, así como más susceptibles son de ser paralelizados, y posteriormente se ha ejecutado sobre ellos una serie de iteraciones técnicas mediante dos de los modelos de paralelismo disponibles en el mercado.

Dichas paralelizaciones emplean dispositivos de propósito general, como las CPUs, y dispositivos de propósito específico, como algunas GPU dispuestas al desarrollo de software científico, utilizando para ello los modelos OpenMP y CUDA, respectivamente.

Durante la investigación, se ha podido indagar en el potencial del paradigma paralelo aplicado sobre el software a través de dos modelos de cómputo muy diferentes entre sí, pero cuyo objetivo final es el mismo: explotar las capacidades del hardware actual, eminentemente paralelo. De esta forma, el funcionamiento del software, así como el tiempo requerido, han sido optimizados.

El desarrollo, implementación, paralelización y prueba del software ha resultado en el aumento en sus capacidades de procesamiento, así como en un mejor rendimiento por unidad de tiempo. El software obtenido no solo es igualmente eficaz, sino que su eficiencia y utilidad son mejores.

# Tabla de contenidos


**Abstract** .......... 7

**Resumen** .......... 7

**Tabla de contenidos** .......... 9

**Tabla de títulos** .......... 11

**Capítulo 1. El paradigma paralelo. Contexto de desarrollo.** .......... 15
- 1.1. Motivación del trabajo. Introducción al paradigma paralelo. .......... 16

**Capítulo 2. Análisis del software empleado.** .......... 19
- 2.1. Descripción del software proporcionado. .......... 20
- 2.2. Descripción de los sistemas de experimentación y prueba .......... 24
- 2.3. Verificación del software implicado en la investigación. .......... 26
- 2.4. Descripción de los modelos paralelos utilizados. .......... 27

**Capítulo 3. Análisis preliminar del software.** .......... 29
- 3.1. Análisis de la traza del software. .......... 30
- 3.2. Análisis interno del tiempo de ejecución. .......... 31
- 3.3. Optimización previa mediante las opciones del compilador. .......... 35
- 3.4. Análisis del uso de recursos. .......... 37

**Capítulo 4. Paralelización mediante OpenMP.** .......... 39
- 4.1. Introducción. Limitaciones técnicas del modelo basado en CPU. .......... 40
- 4.2. Paralelización de la función Adicional. .......... 41
- 4.3. Paralelización de la función Interna. .......... 42
  - i. Primera salida de resultados: Adicional e Interna. .......... 45
- 4.4. Paralelización de la función Externa. .......... 49
  - ii. Segunda salida de resultados: Externa. .......... 53
- 4.5. Paralelización combinada de las funciones Externa e Interna. .......... 55
  - iii. Tercera salida de resultados: Externa e Interna. .......... 56
- 4.6. Paralelización unificada de las funciones Externa e Interna. .......... 59
  - iv. Cuarta salida de resultados: Unificada. .......... 60
- 4.7. Conclusiones establecidas acerca del modelo. .......... 63

**Capítulo 5. Paralelización mediante CUDA.** .......... 65
- 5.1. Introducción. Limitaciones técnicas del modelo basado en GPU. .......... 66
- 5.2. Paralelización de la función Interna. .......... 70
- 5.3. Paralelización de la función Unificada. Niveles de anidación. .......... 72
- 5.4. Paralelización coalescente de la función Interna. .......... 74
  - v. Quinta salida de resultados. .......... 77
- 5.5. Conclusiones establecidas acerca del modelo. .......... 80

**Capítulo 6. Paralelización combinada de OpenMP y CUDA.** .......... 83
- 6.1. Estrategias de paralelización combinada. .......... 84
  - vi. Sexta salida de resultados. .......... 85

**Capítulo 7. Conclusiones finales.** .......... 87
- 7.1. Elección final para el problema solucionado. .......... 88

**Apéndice. Consideraciones debidas al hardware.** .......... 91
- Apéndice 1. Consumo energético asociado a cada modelo paralelo. .......... 92
- Apéndice 2. Análisis del efecto optimizador del compilador. .......... 94

**Bibliografía** .......... 97

**Anexos** .......... 99
- 1) Distribución, compilación y ejecución de contenidos del soporte digital .......... 99
- 2) Salida completa del análisis de tiempos de ejecución en el Ordenador Estudiante. .......... 100
- 3) Tabla completa del uso de recursos durante la ejecución del software inicial en el Ordenador Estudiante. .......... 103

# Tabla de títulos

## Códigos.

Código 1: Estructura de bucles usada para mover el defecto puntual alrededor del defecto extenso........................................ 22
Código 2: Estructura de bucles usada para mover el defecto puntual alrededor del defecto extenso........................................ 22
Código 3: Descripción de la función que permite contabilizar el tiempo de ejecución de una porción de código. ....................... 31
Código 4: Establecimiento de la región paralela de la función Adicional.................................................................................. 41
Código 5: Anidación de bucles dispuesta en la función Interna. .............................................................................................. 42
Código 6: Procesamientos realizados en relación a la celda Defecto considerada. ................................................................... 42
Código 7: Establecimiento de los índices de procesado de las celdas consideradas.................................................................. 42
Código 8: Descomposición del índice k en sus correspondencias, para cada una de las tres dimensiones. ................................ 43
Código 9: Establecimiento de la región paralela de la función Interna...................................................................................... 44
Código 10: Bucles anidados, dispuestos en la función Externa. ................................................................................................ 49
Código 11: Procesamiento del cuarto bloque para el Defecto A. ............................................................................................... 50
Código 12: Creación de los ficheros temporales. ..................................................................................................................... 51
Código 13: Establecimiento del grupo de hilos de trabajo para la función Externa. ................................................................... 51
Código 14: Concatenación de los ficheros temporales creados por los hilos de proceso. ........................................................... 52
Código 15: Procesamiento unificado de ambas celdas Defecto. ................................................................................................ 59
Código 16: Anidación de bucles dispuesta en la función Interna. ............................................................................................. 70
Código 17: Reducción implementada. ...................................................................................................................................... 71
Código 18: Bucles anidados dispuestos en la función Externa. ................................................................................................. 72
Código 19: Establecimiento de los índices de procesado de las celdas, para cada una de las iteraciones requeridas................. 74
Código 20: Reordenación de las tres celdas implicadas en el procesamiento posterior.. ........................................................... 75
Código 21: Llamada al núcleo de ejecución en el dispositivo gráfico. ........................................................................................ 75
Código 22: Núcleo final desarrollado mediante el modelo CUDA. .............................................................................................. 77
Código 23: Reordenación de los datos de las celdas mediante OpenMP. ................................................................................... 85

## Ecuaciones.

Ecuación 1: Proceso interno llevado a cabo por la función CalcEnerEl........................................................................................ 21

## Figuras.

Figura 1: Esquema general de la estructura de las celdas de simulación utilizadas en el software............................................. 20
Figura 2: Esquema del proceso de identificación de átomos desplazados y posiciones de red vacías. ........................................ 21
Figura 3: Esquema del proceso de cálculo de la energía de interacción elástica entre el defecto puntal (en azul) y el defecto extenso (en rojo). Las zonas sombreadas representan los campos de tensiones que generan. .................................................. 22
Figura 4: Esquema de flujo del software. .................................................................................................................................. 23
Figura 5: Gráfico ilustrativo de los tiempos durante el procesamiento del primer bloque. ......................................................... 32
Figura 6: Gráfico ilustrativo de los tiempos durante el procesamiento del segundo bloque. ...................................................... 32
Figura 7: Gráfico ilustrativo de los tiempos durante el procesamiento del tercer bloque. .......................................................... 33
Figura 8: Gráfico ilustrativo de los tiempos durante el procesamiento del cuarto bloque. ......................................................... 33
Figura 9: Uso de los recursos principales en el tiempo de ejecución del software. .................................................................... 38
Figura 10: Gráfico ilustrativo de los tiempos obtenidos para la función Adicional. .................................................................... 48
Figura 11: Gráfico ilustrativo de los tiempos obtenidos para la función Interna. ....................................................................... 48
Figura 12: Gráfico ilustrativo de los tiempos obtenidos para la función Externa. ...................................................................... 54
Figura 13: Esquema semántico de la distribución de hilos entre las dos funciones involucradas. .............................................. 55
Figura 14: Gráfico ilustrativo de los tiempos obtenidos para las funciones Externa e Interna. ................................................... 58
Figura 15: Gráfico ilustrativo de los tiempos obtenidos para la función Unificada. .................................................................... 62
Figura 16: Características principales del dispositivo gráfico disponible. .................................................................................. 66
Figura 17: Composición completa de Kepler.El modelo disponible, sin embargo, no dispone de los componentes oscurecidos. 67
Figura 18: Gráfico ilustrativo de los tiempos obtenidos para OpenMP y CUDA........................................................................... 78
Figura 19: Gráfico ilustrativo de los tiempos obtenidos para OpenMP, CUDA y su combinación. .............................................. 86

Figura 20: Representación gráfica apilada de los tiempos obtenidos para cada una de las configuraciones disponibles. ............ 89
Figura 21: Representación gráfica de las aceleraciones conseguidas para cada combinación disponible ................................... 95
Figura 22: Resultados de la aplicación del monitor sobre el software en ejecución ................................................................. 104

## Tablas.

Tabla 1: Tabla representativa del uso llevado a cabo de cada función involucrada en la ejecución del software. ....................... 30
Tabla 2: Tabla representativa de los tiempos relacionados con la ejecución optimizada o no del software. ............................... 35
Tabla 3: Tabla representativa de los tiempos de proceso asociados a la ejecución completa del software. ............................... 36
Tabla 4: Tabla representativa del uso de recursos acorde a la ejecución del software. .............................................................. 37
Tabla 5: Representación comparativa de los resultados obtenidos para la función Adicional en el Ordenador Estudiante. ......... 45
Tabla 6: Representación comparativa de los resultados obtenidos para la función Interna en el Ordenador Estudiante. ........... 46
Tabla 7: Representación comparativa de los resultados obtenidos para la función Adicional en la plataforma TFG. .................. 46
Tabla 8: Representación comparativa de los resultados obtenidos para la función Interna en la Plataforma TFG. ..................... 46
Tabla 9: Representación comparativa de los resultados obtenidos para la función Adicional en el Servidor Beta. ...................... 47
Tabla 10: Representación comparativa de los resultados obtenidos para la función Interna en el Servidor Beta. ....................... 47
Tabla 11: Representación comparativa de los resultados para la función Externa en el Ordenador Estudiante. .......................... 53
Tabla 12: Representación comparativa de los resultados para la función Externa en la Plataforma TFG. ................................... 53
Tabla 13: Representación comparativa de los resultados para la función Externa en el Servidor Beta. ...................................... 54
Tabla 14: Representación comparativa de los resultados obtenidos para las combinaciones en la Plataforma TFG. .................... 57
Tabla 15: Representación comparativa de los resultados obtenidos para las combinaciones en el Servidor Beta. ....................... 57
Tabla 16: Representación comparativa de los resultados obtenidos mediante la unificación en el Ordenador Estudiante. ......... 60
Tabla 17: Representación comparativa de los resultados obtenidos mediante la unificación en la Plataforma TFG. .................... 61
Tabla 18: Representación comparativa de los resultados obtenidos mediante la unificación en el Servidor Beta. ....................... 61
Tabla 19: Representación comparativa de los resultados obtenidos mediante el modelo CUDA. ............................................... 78
Tabla 20: Representación de los resultados obtenidos mediante la combinación de ambos modelos. ....................................... 85
Tabla 21: Tiempos totales obtenidos para la Plataforma TFG. ........................................................................................... 89
Tabla 22: Representación de los tiempos de ejecución en serie del software proveído en cada sistema disponible. .................. 94
Tabla 23: Representación de los tiempos, aceleraciones y fallos de páginas obtenidos tras las pruebas. .................................. 94

# Capítulo 1
# El paradigma paralelo
# Contexto de desarrollo

En este primer capítulo se realiza una introducción a las características del paradigma paralelo y sus posibles implementaciones.

A partir de ellas, y teniendo en cuenta el entorno de desarrollo, se exponen la motivación y objetivos del trabajo.

## 1.1. Motivación del trabajo. Introducción al paradigma paralelo.

La capacidad de cómputo de la que dispone casi cualquier dispositivo tecnológico se ha incrementado en gran medida, encontrando, cada día más, procesadores construidos mediante varios núcleos en la mayor parte de computadores o electrodomésticos disponibles en el mercado, así como en los diferentes gadgets existentes en la actualidad, ya se trate de una televisión, un reloj, un vehículo, un teléfono, un enrutador, o incluso dispositivos de bajo coste como el hardware empotrado.

Se debe diferenciar, sin embargo, este incremento en la capacidad de procesamiento por la forma en que se está consiguiendo. Si bien la capacidad bruta de algunos diseños y modelos de procesadores sigue aumentando, esta se debe a la duplicación de sus componentes internos, en contraposición a la evolución tradicional consistente en el desarrollo de componentes de mayor potencia.

El aumento en el coste de la producción de procesadores que permitan una mayor frecuencia de procesamiento, debido a la progresiva dificultad en la miniaturización de componentes y a los problemas en la disipación calorífica que la acompaña, ha conseguido que los investigadores recurran a técnicas alternativas.

Esto se traduce en que los nuevos componentes de procesamiento mantienen o aumentan lentamente su capacidad, mientras incorporan una mejora de cómputo a través de los denominados núcleos de procesamiento. Gracias a ellos es posible llevar a cabo un procesamiento "en equipo", consiguiendo así una mejora en la productividad, si el software se encuentra adaptado a este paradigma de procesamiento. Es en este punto en el que entra con fuerza la computación paralela, proporcionando soluciones en las que se aprovechen adecuadamente los núcleos de procesamiento de los que dispone este tipo de hardware.

Este paradigma permite organizar varios elementos de cómputo de forma que realicen un trabajo cooperativo, maximizando así la eficiencia en tiempo de ejecución. Algunas de las técnicas que permiten obtener cómputo paralelo son:

- la organización de varios hilos de trabajo (el enfoque más extendido, debido al abaratamiento de los costes asociados),
- el establecimiento de redes de procesamiento interconectadas, y
- la utilización de dispositivos de propósito específico (como los dispositivos de procesamiento gráfico).

Dependiendo del modelo sobre el que se desarrolle la solución, así como de las características del problema, el rendimiento obtenido será más o menos adecuado a los recursos empleados para conseguirlo. En líneas generales, se puede decir que este tipo de soluciones permiten una explotación más eficiente que la ejecución en serie del mismo software.

Este trabajo surge del objetivo descrito hasta el momento: mejorar la eficiencia de un software, utilizado activamente por un equipo de investigación, mediante su optimización, de forma que el tiempo empleado en sus cálculos sea mejor aprovechado utilizando las capacidades paralelas del hardware disponible. Gracias a ello, el equipo de investigación podrá progresar y hallar nuevos resultados a sus investigaciones más rápidamente, disminuyendo asimismo el coste económico involucrado en ellas.

Los modelos que se van a analizar e investigar son aquellos cuya implementación puede ser desarrollada en un solo sistema hardware, y que han sido destacados como de utilidad en la labor de optimización. En ese sentido, el proceso de paralelización no se va a llevar a cabo mediante la cooperación de varios sistemas hardware dispuestos en red.

Los dos modelos de explotación escogidos son los siguientes:

- OpenMP, centrado en el aprovechamiento de las capacidades de procesamiento de CPUs multinúcleo, y
- CUDA, centrado en el aprovechamiento de las capacidades de procesamiento de los dispositivos gráficos de la familia de componentes de NVidia.

El desarrollo e implementación de cada una de las soluciones, investigadas y halladas, para cada modelo descrito, se va a realizar de forma incremental, tratando de mejorar continuamente el rendimiento conseguido mediante su aplicación, mejorando el código ya paralelizado, o realizando la extensión del paradigma sobre nuevas zonas susceptibles de ser paralelizadas.

Esta metodología permite mejorar el software de forma escalonada y modular, consiguiendo, durante el proceso de investigación y desarrollo, progresar en el uso que se hace de los modelos estudiados.

**Objetivos principales del Trabajo Fin de Grado:**

- Aplicar y profundizar los conocimientos adquiridos en la Universidad.
- Analizar el software en serie en busca de puntos susceptibles de ser paralelizados.
- Incorporar soluciones aplicando los modelos seleccionados. Verificar la eficiencia de dichos modelos.

# Capítulo 2
## Análisis del software empleado

Este capítulo se dedica a examinar el software proveído, especificando sus características y su propósito final.

Asimismo, se describen los sistemas de experimentación disponibles, el software complementario a utilizar, y los modelos que permitirán aplicar optimizaciones.

## 2.1. Descripción del software proporcionado.

Antes de comenzar a desarrollar la primera solución que optimice el rendimiento del software, es conveniente describir su estructura.

El software proveído es empleado por el grupo de investigación *“Multiscale Materials Modeling”*, Unidad de Investigación Consolidada del Departamento de Electricidad y Electrónica de la Universidad de Valladolid. Su función es calcular la energía de interacción elástica entre defectos puntuales y defectos extensos en materiales a partir de los campos de tensión generados por ellos.

Este software ha sido utilizado para obtener los resultados en diferentes trabajos científicos:

- *“Modelling of defect induced strain fields in crystalline semiconductors”*, I. Santos, et al. Ponencia oral presentada en el congreso internacional *Advances in Materials and Processing Technologies*. Madrid, diciembre 2015.
- *“Atomistic study of the anisotropic elastic interaction between extended and point defects in crystalline silicon and its influence on Si self-interstitial diffusion”*, I. Santos, et al. Ponencia oral que será presentada en el congreso internacional Simulation of Semiconductor Processes and Devices, Nurember, Alemania, septiembre 2016.

El código dispuesto inicialmente se encuentra escrito en lenguaje C y se estructura en un total de cuatro funciones, incluyendo la función principal, que a continuación se describen.

**Bloque 1: Inicialización y lectura de coordenadas atómicas y componentes del campo de strain.**

En este primer bloque se inicializan las diferentes variables utilizadas, y se leen cuatro ficheros de entrada:

- CeldaBackground.
- CeldaDefecto-A.
- CeldaDefecto-B.
- RedPerfecta.

Los tres primeros ficheros contienen las coordenadas atómicas y las componentes de tensión en esas posiciones de las celdas de simulación con el defecto extenso (*CeldaBackground*), y con el defecto puntual en sus configuraciones irreducibles, dos en el caso considerado (*CeldaDefecto-A* y *CeldaDefecto-B*).

El fichero *RedPerfecta* contiene las posiciones de la red de silicio cristalino sin ningún defecto. En la Figura 1 se muestra un esquema general de las dimensiones de estas celdas. Los parámetros *NCELDASX*, *NCELDASY* y *NCELDASZ* controlan el número de celdas fundamentales en cada dirección del espacio. Sus valores los define el usuario teniendo en cuenta las dimensiones de los defectos que se quieren analizar. Para el trabajo realizado, este número se ha fijado en 37 para cada una de las dimensiones.

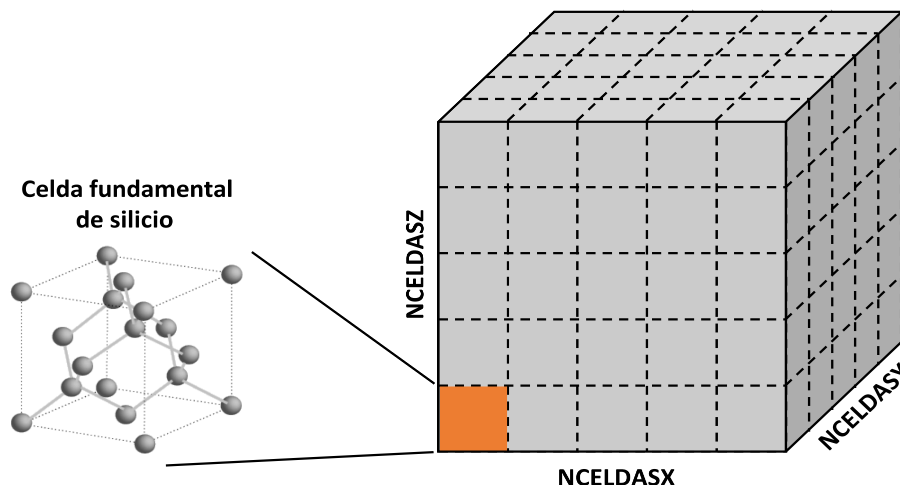


*Figura 1: Esquema general de la estructura de las celdas de simulación utilizadas en el software.*

**Bloque 2: Análisis de los átomos desplazados y de las posiciones de red vacías.**

Una vez leídas las coordenadas atómicas, se determinan los átomos que están fuera de una posición de red (*Displaced Atoms*, DA), y las posiciones de la red que no tienen ningún átomo (*Empty Sites*, ES). Esto se hace mediante una función denominada *AnalisisDAES*, que compara las coordenadas de las celdas *CeldaBackground*, *CeldaDefecto-A* y *CeldaDefecto-B* con las de la celda *RedPerfecta*.

Esta comparación se realiza mediante un bucle que recorre los átomos de la celda con el defecto y las posiciones de la red perfecta, y se calculan distancias entre ellas para hacer la asignación. En la Figura 2 se muestra esquemáticamente este proceso. Por lo tanto, en este bloque se llama tres veces a la función *AnalisisDAES*, debido a la existencia de tres celdas a comparar. A partir de ahora, las optimizaciones sobre esta función serán referenciadas como *"optimizaciones sobre la función **Adicional**"*.

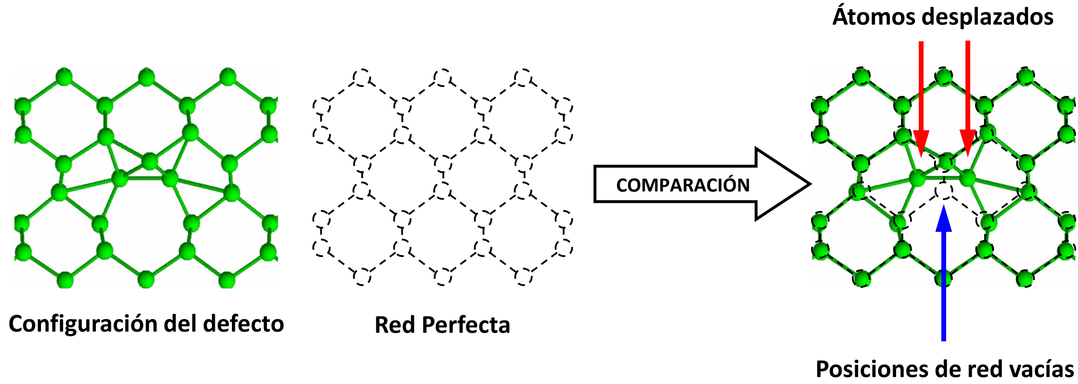


*Figura 2: Esquema del proceso de identificación de átomos desplazados y posiciones de red vacías.*

**Bloque 3: Reparto de las tensiones a las posiciones de red vecinas.**

En este paso, las componentes de tensiones en las posiciones atómicas de *CeldaBackground*, *CeldaDefecto-A* y *CeldaDefecto-B* se reparten a las posiciones de red vecinas más cercanas. Esta parte se realiza mediante la función denominada *RepartirStrain*, que se llama tres veces, una por cada celda que contiene un defecto.

**Bloque 4: Cálculo de la energía de interacción elástica.**

Este es el bloque más relevante del software, el que realiza el cálculo de la energía de interacción elástica entre el defecto puntual y el defecto extenso. En este bloque se llama a la función denominada *CalcEnerEl*, que devuelve la energía de interacción elástica para una posición del defecto puntual en el entorno del defecto extenso utilizando la Ecuación 1.

$$E_{elastic}(\overrightarrow{r_d}) = \frac{1}{2}\sum_{k_x=1}^{NCELDASX}\sum_{k_y=1}^{NCELDASY}\sum_{k_z=1}^{NCELDASZ}\Bigg\{ \frac{1}{Y}\left(\sigma_{11}^{B}[k_x,k_y,k_z]\sigma_{11}^{d}[k_x,k_y,k_z]+\sigma_{22}^{B}[k_x,k_y,k_z]\sigma_{22}^{d}[k_x,k_y,k_z]+\sigma_{33}^{B}[k_x,k_y,k_z]\sigma_{33}^{d}[k_x,k_y,k_z]\right)$$

$$-\frac{\nu}{Y}\Big(\sigma_{11}^{B}[k_x,k_y,k_z]\left(\sigma_{22}^{d}[k_x,k_y,k_z]+\sigma_{33}^{d}[k_x,k_y,k_z]\right)+\sigma_{22}^{B}[k_x,k_y,k_z]\left(\sigma_{21}^{d}[k_x,k_y,k_z]+\sigma_{33}^{d}[k_x,k_y,k_z]\right)+\sigma_{33}^{B}[k_x,k_y,k_z]\left(\sigma_{11}^{d}[k_x,k_y,k_z]+\sigma_{22}^{d}[k_x,k_y,k_z]\right)\Big)$$

$$+\frac{2}{G}\left(\sigma_{12}^{B}[k_x,k_y,k_z]\sigma_{12}^{d}[k_x,k_y,k_z]+\sigma_{13}^{B}[k_x,k_y,k_z]\sigma_{13}^{d}[k_x,k_y,k_z]+\sigma_{23}^{B}[k_x,k_y,k_z]\sigma_{23}^{d}[k_x,k_y,k_z]\right)\Bigg\}$$

*Ecuación 1: Proceso interno llevado a cabo por la función CalcEnerEl.*

En esta ecuación, $\sigma_{ij}^{B}$ son las componentes de tensiones generadas por el defecto extenso (*CeldaBackground*), $\sigma_{ij}^{d}$ son las componentes de tensiones generadas por el defecto puntual en la posición $\overrightarrow{r_d}$ (*CeldaDefecto*), e $Y$, $\nu$ y $G$ son el módulo de Young, el coeficiente de Poisson, y el módulo de deformación transversal, respectivamente. De esta manera se calcula la energía de interacción elástica cuando el defecto puntual está en la posición $\overrightarrow{r_d}$.

Para implementar esta operación matemática se usa la estructura de bucles *for* mostrada en el Código 1.

```
for  (kx=0; kx< 2*NCELDASX; kx++)
{
  for  (ky=0; ky< 2*NCELDASY; ky++)
  {
      for  (kz=0; kz< 2*NCELDASZ; kz++)
      {
          for  (ksl=0; ksl< 2; ksl++)
          {
```

*Código 1: Estructura de bucles usada para mover el defecto puntual alrededor del defecto extenso.*

El cálculo de la energía de interacción elástica en todo el entorno del defecto extenso implica mover el defecto puntual a su alrededor y, para cada nueva posición $\vec{r_d}$, se llama de nuevo a la función *CalcEnerEl*. Este proceso se ha esquematizado en la Figura 3.

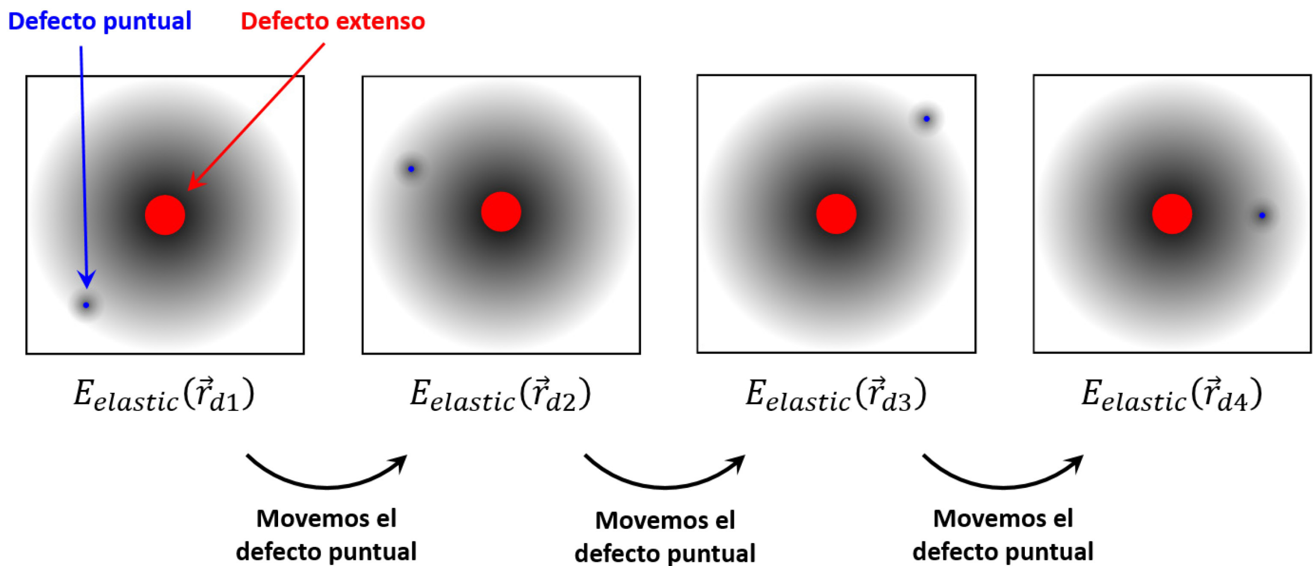


*Figura 3: Esquema del proceso de cálculo de la energía de interacción elástica entre el defecto puntal (en azul) y el defecto extenso (en rojo). Las zonas sombreadas representan los campos de tensiones que generan.*

Para mover el defecto puntual alrededor del defecto extenso se utiliza la estructura de bucles *for* mostrada en el Código 2. Puede verse que, al igual que en la función *CalcEnerEl*, se utilizan las variables $k_x$, $k_y$ y $k_z$, que recorren la celda de simulación.

```
for  (kx=0; kx< 2*NCELDASX; kx=kx+2)//Se salta de celda en celda ==> kx = kx + 2
{
  for  (ky=0; ky< 2*NCELDASY; ky=ky+2)//Se salta de celda en celda ==> ky = ky + 2
  {
    for  (kz=0; kz< 2*NCELDASZ; kz=kz+2)//Se salta de celda en celda ==> kz = kz + 2
    {
      for( kj = 0 ; kj < 4; kj ++)
      {
```

*Código 2: Estructura de bucles usada para mover el defecto puntual alrededor del defecto extenso.*

Para simplificar la comprensión de las optimizaciones realizadas, a partir de este momento las optimizaciones sobre la función *CalcEnerEl* serán referenciadas con el nombre de *"optimizaciones sobre la función **Interna**"*, mostrada parcialmente en el Código 1, y a las optimizaciones relacionadas con los bucles de la Figura 3 y el Código 2, que mueven el defecto puntual alrededor del defecto extenso, serán referenciadas con el nombre de *"optimizaciones sobre la función **Externa**"*.

**Salida y finalización del software.**

Durante la realización de los cálculos, los resultados se van imprimiendo, según se obtienen, en el fichero *StrainEnergy.data*, que contiene la energía de interacción elástica entre el defecto puntual y el defecto extenso para todas las posiciones posibles del defecto puntual en torno al defecto extenso.

Una vez descritos los diferentes bloques en los que se divide el software objeto de estudio, en la Figura 4 se muestra el flujo completo.

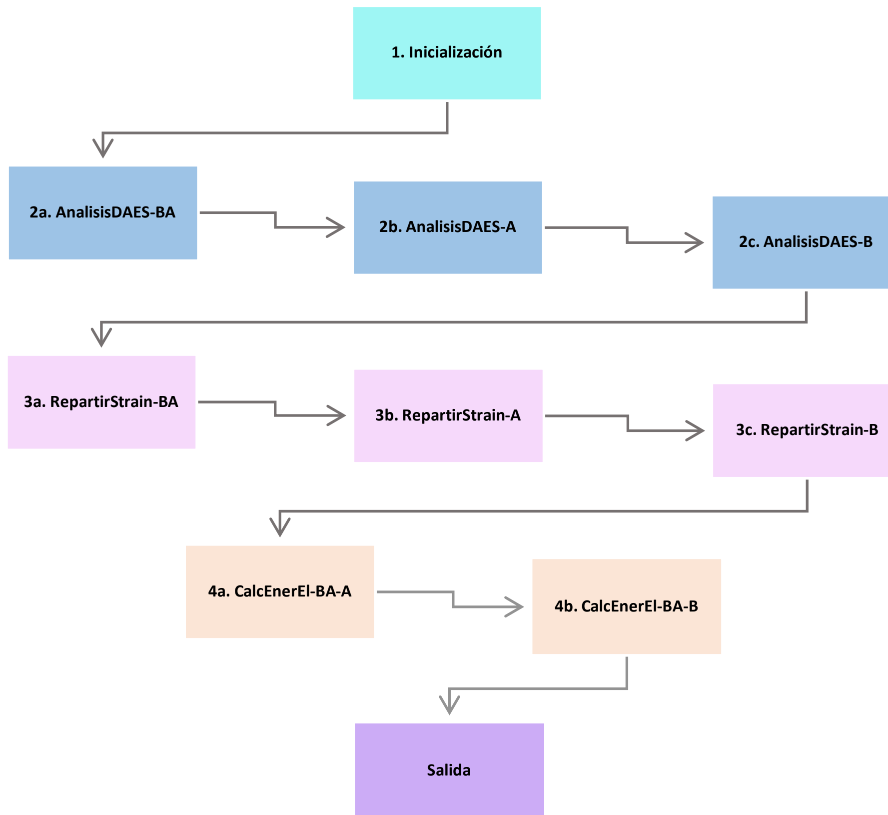


*Figura 4: Esquema de flujo del software.*

## 2.2. Descripción de los sistemas de experimentación y prueba.

Durante la investigación y desarrollo de este trabajo se han dispuesto de hasta un total de tres sistemas hardware sobre los que realizar pruebas para verificar las mejoras de rendimiento desarrolladas.

Debido a las distintas características de cada uno de estos sistemas, los resultados obtenidos han sido diferentes, notando así como el código implementado se adapta a los componentes hardware de manera más o menos eficiente. La obtención de los cálculos en referencia a las aceleraciones se ha basado asimismo en sus correspondientes tiempos originales, consiguiendo así conclusiones coherentes y bien definidas, particularizadas al sistema de experimentación.

A continuación se van a pasar a describir estos sistemas. Con la finalidad de disponer de referencias de seguimiento durante la investigación, cada uno de ellos va a ser denominado con un nombre, teniendo en cuenta su contexto específico.

**a)** **Ordenador Estudiante**: denominado así por ser el sistema disponible por el estudiante, de forma personal, y que ha colaborado en la obtención de una muestra de resultados más amplia.

Este sistema se encuentra soportado mediante un sistema operativo Windows 8.1 en el que se ha virtualizado una distribución Ubuntu 14.04.1 de 64 bits en la que se han ejecutado las pruebas.

Dicha máquina virtual ha trabajado con un total de tres núcleos de procesamiento en conjunción con 4GB de memoria RAM. Las características técnicas de la plataforma huésped son las siguientes:

- Intel Core i5 2500k a 4GHz (4 núcleos efectivos, 3 utilizables por la máquina virtual).
- 8GB de memoria RAM (4 utilizables por la máquina virtual).

**b)** **Plataforma TFG**: denominada de esta forma por ser el sistema asignado de forma exclusiva para desarrollar el trabajo. Disponible para su uso directo y en remoto, es el único sistema con capacidad para ejecutar soluciones CUDA, requeridas en el trabajo. Su funcionamiento se encuentra dispuesto mediante una distribución de Ubuntu 14.04.1 de 64 bits.

Las características hardware técnicas son las siguientes:

- Intel Core i7 860 a 2,8Ghz (4 núcleos físicos de procesamiento, extendidos en 8 hilos lógicos mediante la tecnología HyperThreading de Intel).
- 4GB de memoria RAM.
- NVidia GeForce GTX 650Ti, equipada con 2GB de memoria VRAM.

**c)** **Servidor Beta**: autodenominado con esta nomenclatura por el equipo de investigación de la Universidad, cuyo acceso únicamente en remoto comprende una dirección formada por este nombre. Se trata de un servidor formado por un total de quince nodos soportados por un gestor de colas SGE (*Sun Grid Engine*).

Su funcionamiento consiste en lo siguiente: los usuarios identificados envían trabajos al sistema, especificando el número de núcleos a utilizar. El servidor seguidamente los introduce en una cola, para procesarlos finalmente de manera concurrente y aislada utilizando las capacidades que en ese momento se encuentren disponibles. Mientras tanto, el resto de trabajos se almacenan en una pila, esperando su turno de ejecución.

Si bien teóricamente cada computación es ejecutada de forma aislada, la concurrencia de trabajos consecuentemente disminuye en cierta medida, según la carga total del servidor, el rendimiento de la implementación a evaluar. Sin embargo, está pérdida de rendimiento habitualmente no es de importancia y los resultados han continuado siendo válidos a efectos de obtener conclusiones.

El funcionamiento del sistema se encuentra disponible mediante una distribución CentOS 6.5 de 64 bits. Cada uno de los quince nodos cuenta con las siguientes características hardware:

- Intel Xeon E5-2407 v2 a 2.40GHz (8 núcleos físicos de procesamiento por procesador instalado).
- 8GB de memoria RAM.

Dado que las tres plataformas disponen de suficientes recursos para realizar eficientemente el procesamiento previsto, el tiempo de ejecución depende únicamente de las capacidades de cada procesador, por lo que la única forma de mejorar el rendimiento bruto es mediante un incremento en dicho hardware.

Al experimentar en tres sistemas basados en x86, cuyos núcleos de procesamiento difieren en su tecnología interna, dicho componente es el que marca la diferencia, ya que el resto de parámetros o bien son similares, o no influyen profundamente en la ejecución del software.

## 2.3. Verificación del software implicado en la investigación.

Como paso previo al comienzo del desarrollo del trabajo, se debe verificar que el software requerido se encuentre instalado y preparado para su funcionamiento.

En primer lugar, es necesario que los compiladores asociados a los modelos paralelos a implementar se encuentren disponibles en los sistemas dispuestos. Debido a la naturaleza de las plataformas software, el compilador *gcc*, requerido para compilar tanto el código desarrollado utilizando OpenMP como CUDA, ya se encontraba instalado en todos los sistemas.

Mientras en el Servidor Beta la versión instalada no es reemplazable, dado que dicho servidor es de uso compartido, los dos sistemas restantes disponen de la versión actualizada de dicho compilador, ya que se realizaron instalaciones con el propósito de este trabajo. Así, las versiones disponibles son *gcc 4.4.7*, en el Servidor Beta, y *gcc 4.8.4*, en los dos sistemas restantes.

Por su parte, se requirió la instalación del *CUDA Toolkit* en la Plataforma TFG, la única capaz de ejecutar núcleos CUDA. La versión instalada fue la *7.5.17*. La instalación es sencilla: únicamente requiere la instalación del *Toolkit* a través del repositorio oficial de NVidia, utilizando para ello la herramienta *apt-get*, disponible en las distribuciones de Ubuntu, y el establecimiento posterior de las variables de entorno, requeridas para su correcto funcionamiento, según se describe en la documentación oficial.

Finalmente, otro software requerido en el desarrollo de este trabajo, y que ya se encontró instalado en los sistemas disponibles, es el siguiente:

- *gnuprof*: una herramienta de profiling, la cual permite observar el trazado de un software a través de su código.
- *md5sum*: un generador de sumas de comprobación de la integridad de los datos, el cual permite comprobar la corrección de la salida para los códigos desarrollados.
- *sadc*/*sar*: un medidor software que permite monitorizar los parámetros del sistema, en relación al uso de recursos.
- *top*: un administrador de procesos, el cual permite observar la actividad del sistema en tiempo real.

## 2.4. Descripción de los modelos paralelos utilizados.

El trabajo planteado consiste en el desarrollo de soluciones software que permitan aprovechar y optimizar el tiempo de procesamiento para un software científico mediante la utilización de dos tecnologías paralelas seleccionadas de entre las disponibles en el mercado. A continuación se realiza una breve introducción a ellas y se justifica su motivación.

Como tecnologías paralelas se ha escogido, por un lado, OpenMP, por sus capacidades de paralelización en CPU y, por otro lado, CUDA, por su alto grado de paralelización utilizando las capacidades de GPUs específicas.

Ambas tecnologías afectan a diferentes componentes hardware, por lo que ambas pueden asimismo combinarse intentando obtener un mejor rendimiento paralelo.

La compilación de los códigos se ha llevado a cabo principalmente utilizando el compilador *gcc* en todos los incrementos de desarrollo realizados.

Durante el desarrollo del modelo CUDA, el compilador utilizado ha sido el proporcionado por NVidia, *nvcc*, el cual precompila el código para terminar siendo compilado por *gcc* (lo que permite aplicar el modelo de OpenMP en complemento con CUDA).

**Conclusiones principales determinadas en este capítulo:**

- Aplicar el uso del software de medida seleccionado con el fin de obtener resultados significativos acerca del uso de recursos.
- Utilizar convenientemente los sistemas dispuestos para la realización del trabajo.
- Examinar el software en busca de los mayores puntos potenciales de explotación de los modelos seleccionados.

# Capítulo 3
## Análisis preliminar del software

La finalidad de este capítulo es establecer las porciones de código que consumen una mayor proporción del tiempo de ejecución en serie.

Para ello se analizarán las funciones y sus estructuras principales, la traza del software y el uso de recursos hardware asociado.

Asimismo, se establece una primera optimización debida al compilador utilizado.

## 3.1. Análisis de la traza del software.

Mediante una herramienta de *profiling*, como es *gnuprof*, es posible descubrir el camino que sigue el software mientras se ejecuta.

Así, es posible conocer específicamente, de forma concisa y exacta, el número de llamadas que este hace a las diversas funciones que lo forman, los cambios de contexto implicados en la realización de estas llamadas y el tiempo empleado en cada una de las funciones referidas.

A pesar de que el uso de esta herramienta de *profiling* añade una sobrecarga, esta no es influyente en los parámetros a evaluar, ya que estos últimos no dependen de la velocidad de ejecución, ni del tiempo de procesamiento de las librerías añadidas por *gnuprof*.

La utilización del *profiler* es sencilla. Añadiendo la baliza de compilación *–pg* a la orden de compilación de *gcc*, se consigue añadir y compilar el código asociado a *gnuprof*, el cual evalúa en tiempo de ejecución las funciones que el código contiene, almacenando los resultados en un fichero binario que después la herramienta de *profiling* interpreta.

Realizado este procedimiento, ejecutado sobre el Ordenador Estudiante, y tras una selección de los datos, se obtienen los resultados descritos en la Tabla 1.

| | **gnuprof - serie** | | | |
|---|---|---|---|---|
| **Función** | **% utilizado** | **Tiempo (s)** | **Llamadas procesadas** | **s/llamada (media)** |
| ***Main*** | **0,0** | **0,39** | - | - |
| ***AnalisisDAES*** | **15,3** | **1780,50** | 3 | 593,50 |
| ***RepartirStrainRP*** | **0,0** | **0,20** | 3 | 0,07 |
| ***CalcEnerEl*** | **84,6** | **9795,84** | 405224 | 0,02 |
| **Total** | **100,0** | **11576,93** | | |

*Tabla 1: Tabla representativa del uso llevado a cabo de cada función involucrada en la ejecución del software.*

En primer lugar, se observa que el código se encuentra estructurado en cuatro funciones, incluyendo la función principal, y que de estas funciones, *CalcEnerEl* representa el 84,6% del tiempo general de ejecución, recibiendo un total de 405224 llamadas. El tiempo por llamada sin embargo es, de media, de 0,02 segundos. Gracias a la herramienta de *profiling* se ha podido prestar atención a esta función, la cual, sin su utilización, y debido al grano fino de este último resultado, hubiera pasado desapercibida al representar un tiempo por llamada pequeño.

Con estos datos se puede determinar que esta función es representativa del software, a la cual se realizan un gran número de llamadas, lo que probablemente sea también un indicativo de cambios de contexto entre la función principal y esta función, representando una posible fuga de rendimiento.

Volviendo a la tabla de resultados, se puede ver que existe una segunda función, *AnalisisDAES*, que emplea un tiempo que, si bien es bastante menor si se toma como referencia la función previa, es también notable, al consumir un 15,3% del tiempo total de ejecución, una cantidad que no se puede considerar despreciable.

Al contrario, sin embargo, esta función solo es requerida únicamente en tres ocasiones, utilizando para el procesamiento de cada una de ellas un tiempo considerable, por lo que los cambios de contexto asociados a estas llamadas son más razonables, en términos de rendimiento por llamada.

Terminando la tabla, se encuentran finalmente las dos funciones restantes, cuyo tiempo de ejecución no representa apenas el 1% del total procesado, por lo que estas funciones no parecen ser prioritarias en el plan de optimización a desarrollar.

## 3.2. Análisis interno del tiempo de ejecución.

Este análisis se realiza añadiendo una función al software que permite contabilizar de manera precisa el tiempo de procesamiento para una porción de código controlada por el desarrollador. Probablemente este es el análisis más característico, debido a su carácter interno.

Para realizar la medición del tiempo, se establece como referencia la fecha *epoch* (la cual difiere de la plataforma hardware considerada: en GNU/Linux se trata del 1 de Enero de 1970 (cuyo tiempo se especifica en milisegundos, siendo esta fecha el tiempo 0)). Haciendo dos llamadas a esta función, y calculando la diferencia entre ambos valores temporales, se puede contabilizar el tiempo empleado entre los dos puntos seleccionados del código. En el Código 3 puede examinarse el contenido de dicha función.

```
double elapsedTime(){
  struct timeval tv;
  gettimeofday(&tv, 0);
  return tv.tv_sec + 1.0e-6 * tv.tv_usec;
}
```

*Código 3: Descripción de la función que permite contabilizar el tiempo de ejecución de una porción de código.*

El inconveniente de este análisis es la ligera sobrecarga que incorpora, al requerir la llamada a la función en dos ocasiones y calcular la diferencia entre los dos tiempos, para cada una de las porciones a examinar, imprimiendo asimismo el resultado. A pesar de ello, los resultados son esclarecedores.

Este análisis debe realizarse manualmente, estableciendo los puntos que se desea controlar. Al disponer de resultados acerca de la estructura y el seguimiento de la traza del software, se puede realizar una mejor planificación de los puntos del código a analizar.

De manera general, se han establecido balizas en los puntos que se han considerado de mayor consumo para determinar si realmente lo son y elaborar, de esta forma, un esquema muy concreto del tiempo en ejecución de cada bloque y función que forma el código.

En el análisis se ha tratado de cubrir cada punto procesado por parte del software, midiendo cada una de las funciones completas, los procesos de lectura y escritura de ficheros, así como el inicializado de vectores y cualquier otro proceso que, desde el punto de vista del desarrollador, pueda consumir un tiempo de ejecución notable. La motivación principal de la situación de los puntos de medida se encuentra en el carácter paralelo posterior al cual se va a someter el software.

Este análisis ha permitido observar, de forma desglosada y aislada, el tiempo empleado en cada una de los bucles. Igualmente, el análisis ha permitido comprobar el número de repeticiones de cada uno de ellos. Un ejemplo de notable importancia se encuentra especialmente en el número de llamadas que realiza la función Externa a la Interna. Esta última información ha permitido, por otro lado, comprobar la coherencia con la salida obtenida a través del *profiler* anterior.

Es conveniente puntualizar que este criterio de análisis depende en gran medida de la experiencia que el desarrollador dispone del lenguaje de programación y de sus estructuras, así como del paradigma de programación en que se basa. No se puede garantizar que este análisis se haya ejecutado de forma óptima, si bien los resultados se consideran concluyentes.

Una vez establecida la función de referencia del tiempo y, disponiendo de los puntos de medida dispuestos en el código, el siguiente paso consiste en compilar el código y obtener los resultados mediante su ejecución. Un proceso que se ha realizado, nuevamente, en el Ordenador Estudiante.

A continuación se muestra el desglose obtenido a través de gráficos ilustrativos, justificando los resultados. La estructuración se realiza para cada bloque que conforma el código de la función principal del software.

Asimismo, con el fin de no generar un desglose demasiado extenso, las secciones repetitivas, así como la descripción de los números concretos hallados, han sido omitidas, pudiéndose consultar en la sección correspondiente de los Anexos.

**Bloque 1: Inicialización y lectura de coordenadas atómicas y componentes del campo de strain.**

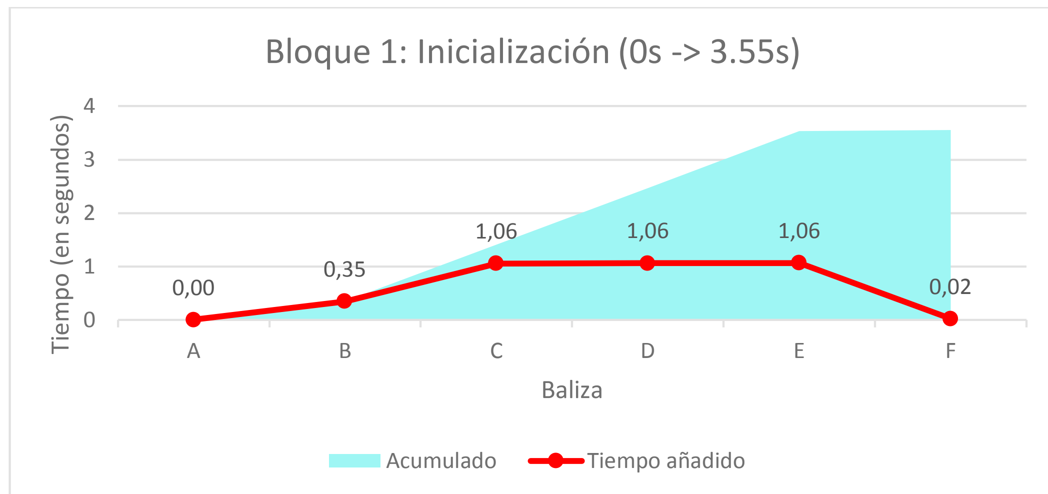


*Figura 5: Gráfico ilustrativo de los tiempos durante el procesamiento del primer bloque.*

Durante la inicialización del software se realizan varias operaciones preliminares entre las que destaca el establecimiento de variables y constantes, llevado a cabo en las balizas B y F, así como la lectura de los ficheros de análisis para su posterior procesamiento, según puede observase en las balizas intermedias, C, D, y E, de la Figura 5.

El proceso de lectura incluye la apertura de los ficheros y su posterior lectura, línea por línea, de forma iterativa mediante la utilización de bucles.

Si bien los tiempos son dependientes del tamaño de la muestra en sus tres dimensiones, en este caso son aceptables y los bucles no dan resultados lo suficientemente grandes como para pensar que una optimización mediante paralelización de las lecturas de ficheros pueda suponer una mejora notable en el rendimiento.

**Bloque 2: Análisis de los átomos desplazados y de las posiciones de red vacías.**

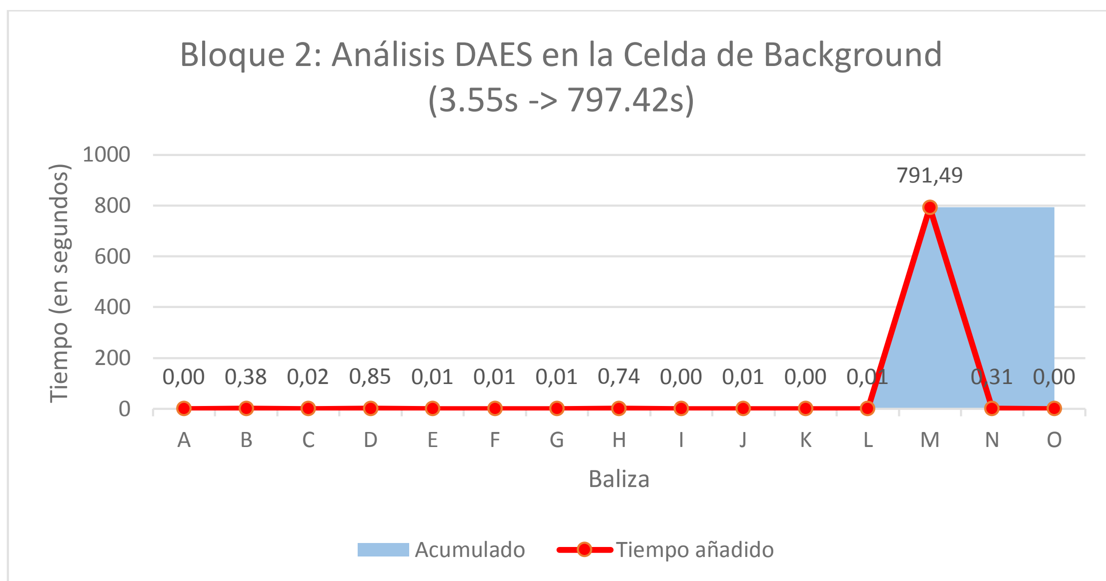


*Figura 6: Gráfico ilustrativo de los tiempos durante el procesamiento del segundo bloque.*

El segundo bloque de procesamiento incluye las tres llamadas a la función Adicional. Durante la ejecución de esta función se puede observar el procesamiento de varios bucles, los cuales asimismo disponen de varias bifurcaciones. Entre todos los bucles, representados por las balizas B a N, destaca uno de ellos, para cada una de las llamadas, el cual consume la gran totalidad del tiempo anteriormente obtenido por el *profiler*, según se puede observar en la baliza M de la Figura 6. En esta porción de código se lleva a cabo el proceso de comparación de átomos de la celda con defecto con la red perfecta, según se describió en el correspondiente Bloque 2, en el Capítulo 2.

Gracias al empleo de este análisis, se ha conseguido detectar el punto exacto de la función en que se emplea un mayor consumo de tiempo.

Por otro lado, para cada llamada, los tiempos observados en los bucles son bastante similares, lo que da lugar a un procesamiento homogéneo, haciendo pensar que, teóricamente, la paralelización puede seguir el mismo comportamiento.

**Bloque 3: Reparto de las tensiones a las posiciones de red vecinas.**

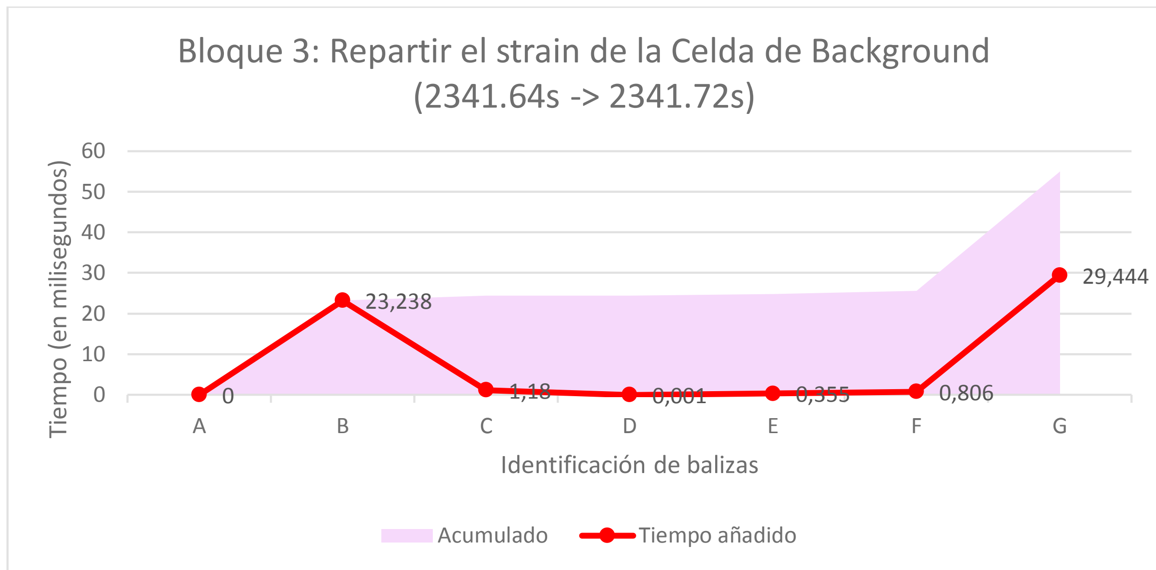


*Figura 7: Gráfico ilustrativo de los tiempos durante el procesamiento del tercer bloque.*

El tercer bloque de procesamiento, al igual que el segundo, dispone de tres llamadas a una función, la denominada *RepartirStrain*.

De la misma manera que ocurre anteriormente, los tiempos de ejecución obtenidos son similares para cada una de las tres llamadas, lo que da lugar a un comportamiento homogéneo, según se puede ver en la Figura 7.

A diferencia, sin embargo, de la función Adicional, esta función se encuentra a varios ordenes de magnitud por debajo, en términos de tiempo de ejecución, lo que impide considerar a dicha función como potencial para efectuar su paralelización.

**Bloque 4: cálculo de la energía de interacción elástica.**

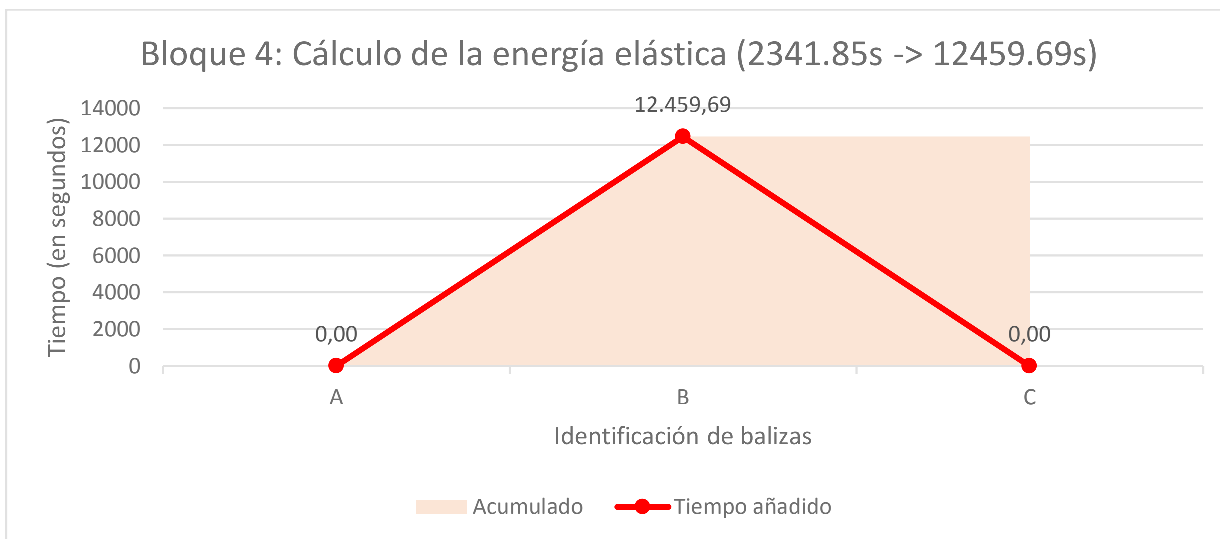


*Figura 8: Gráfico ilustrativo de los tiempos durante el procesamiento del cuarto bloque.*

Finalmente, el software alcanza el cuarto bloque de procesamiento, el cual incluye el cálculo de la energía elástica. Como se describió, este proceso implica el procesamiento de todas las posibles combinaciones entre la celda de Background y cada una de las celdas Defecto consideradas.

Este proceso es laborioso y conlleva la consecución de varios bucles anidados y sus consiguientes iteraciones, dependientes de las dimensiones de las celdas a procesar.

Revisando los resultados ofrecidos en la Figura 8, se observa que en este caso el tiempo empleado se corresponde con una alta proporción del tiempo total de ejecución, en consonancia con la salida ofrecida por la herramienta de *profiling*. Es por lo tanto que este último bloque de procesamiento será el punto de inicio del desarrollo de una solución paralela.

Asimismo, debido a la estructura de esta porción del código, se sabe que la función Externa es la que realiza múltiples llamadas a la función Interna, por lo que el proceso de paralelización puede subdividirse, a su vez, en dos niveles: la función Interna, que realiza el cálculo de la energía elástica para una combinación concreta de celda de Background y Defecto, y la función Externa, que se encarga de establecer las combinaciones y hacer las llamadas a la función Interna.

Por lo tanto, mediante esta porción del código se accede a varias opciones de paralelización: la paralelización externa, que emplea una proporción considerable de tiempo, en relación al tiempo total, y que depende de la función Interna, sobre la que, asimismo, puede establecerse una paralelización interna, debido al número considerable de veces que es llamada, obteniendo así una suma de pequeños tiempos, acompañados por el tiempo de sobrecarga en relación al cambio de contexto implicado en cada llamada.

Siguiendo consistentemente la salida ofrecida por el *profiler*, se discierne que el cuarto bloque, que incluye 405224 llamadas a la función Interna, utiliza la mayor porción de tiempo, a pesar de que por sí misma, cada llamada, emplee únicamente 0,02 segundos, de media. El tiempo crece debido al número acumulativo de llamadas a las que es sometida dicha función.

## 3.3. Optimización previa mediante las opciones del compilador.

Tras haber realizado dos de los tres análisis previstos, y en vista de los resultados ofrecidos por cada uno de ellos, se va a tratar de mejorar el tiempo de ejecución del software mediante los mecanismos que ofrece el compilador a la disposición del desarrollador.

El compilador utilizado para realizar todas las pruebas y análisis, como se sabe, es *gcc*. Este compilador, en la versiones disponibles durante el desarrollo del trabajo, provee varios niveles de optimización, denominados de la siguiente manera: *–O2*, *–O3* y *–Ofast*. Cada nivel de optimización representa la activación de un mayor conjunto de balizas que dan lugar a un proceso de compilación más optimizado.

La mayor parte de este tipo de compilaciones optimizadas no se realiza de forma predeterminada debido a que el funcionamiento del software podría no ser el esperado, por lo que se deben aplicar con precaución. Es por ello que, si bien se van a aplicar varias de estas optimizaciones (especialmente *–Ofast*, la cual, por otro lado, no se encuentra disponible en el Servidor Beta, debido a la versión de su compilador *gcc*), seguidamente se comprobará la corrección de la salida del software, ya que esta es conocida por el desarrollador.

Por lo tanto, utilizando el Ordenador Estudiante, y tras compilar el código mediante el nivel más agresivo de optimización, *–Ofast*, se pasa a realizar la correspondiente ejecución y a comprobar, consecuentemente, que los resultados fuesen correctos.

Mediante el comando *time* se pudo verificar el tiempo de ejecución, obteniendo los siguientes tiempos, descritos en la Tabla 2.

| | time (en segundos) | | | |
|---|---|---|---|---|
| **Tipo** | **Serie** | **Sin llamadas** | **-Ofast** | **Speedup** |
| **user** | **11576,93** | **8,13** | **4085,59** | **2,83** |
| sys | 1,06 | 1,51 | 0,08 | 12,82 |
| real/total | 11577,99 | 9,90 | 4092,13 | 2,83 |

*Tabla 2: Tabla representativa de los tiempos relacionados con la ejecución optimizada o no del software.*

En primer lugar, se pasa a especificar el contenido de esta tabla. En cada columna se mencionan los tiempos que el sistema operativo provee mediante la herramienta *time* para el procesado del software:

- en serie, mediante el código original,
- eliminando las llamadas a las funciones implicadas, de manera que puede verse el tiempo invertido en los procesos previos y posteriores a los cálculos intensivos llevados a cabo por dichas funciones, y
- mediante la baliza más agresiva de optimización (*–Ofast*).

Para este último caso se muestra adicionalmente la aceleración conseguida tomando como referencia el tiempo en serie. A través de la tabla se pueden establecer varias conclusiones.

Aplicando la baliza *–Ofast* como parte del proceso de compilación se puede observar como el tiempo de ejecución se reduce en gran medida, obteniendo una aceleración de 2,83 con respecto a la misma ejecución sin optimizar. Como consecuencia de la mejora obtenida, desde este momento todas las pruebas se realizarán utilizándolo (en el caso del Servidor Beta se utilizará la baliza más agresiva disponible, la cual es *–O3*).

Esta decisión se ve motivada debido a que el objetivo último de la optimización es la maximización de la eficiencia.

Volviendo a la motivación de este capítulo, cabe aclarar que, si bien esta optimización no invalida las conclusiones previas en referencia especialmente a la utilización del *profiler*, sí se han podido producir variaciones en los porcentajes de uso de cada función, lo que sí se reflejaría, por otro lado, en la prioridad del plan de optimización a desarrollar.

Por ello, es conveniente revisar, mediante una segunda prueba, los resultados que en este momento pueda proveer el *profiler*. Por lo tanto, se realiza una segunda ejecución en el Ordenador Estudiante, obteniendo la Tabla 3.

| | gnuprof - serie -Ofast | |
|---|---|---|
| **Función** | **% utilizado** | **Tiempo (s)** |
| **Main + CalcEnerEl** | 76,5 | **3121,35** |
| **AnalisisDAES** | 23,5 | **963,05** |
| **RepartirStrainRP** | 0.00 | **0,08** |
| **Total** | 100,00 | **4084,48** |

*Tabla 3: Tabla representativa de los tiempos de proceso asociados a la ejecución completa del software.*

Como se preveía, realizada la optimización, el porcentaje de utilización ha cambiado. La función Interna, junto a la función principal (esta última anteriormente no consumía un tiempo tangible numéricamente), se ha optimizado lo suficiente como para que ahora represente un 76,5% del tiempo total de ejecución, aumentando así la proporción de tiempo requerido por la función Adicional, que en este momento pasa a consumir un 23,5% del tiempo.

El plan de optimización a desarrollar no cambia sin embargo, debido a que ambas funciones se encuentran en el mismo orden de prioridad que en la prueba anterior.

## 3.4. Análisis del uso de recursos.

El tercer tipo de análisis se relaciona con el uso de recursos que hace el software según se procesa su ejecución.

Mediante el empleo de software de medición como es *sadc/sar*, es posible establecer medidores software (sondas) con el fin de monitorizar los cambios en el uso de la memoria RAM, de la CPU, el número de entradas y salidas, el número de cambios de contexto, o la utilización del disco, entre otros parámetros, en intervalos regulares previamente establecidos.

La inicialización y utilización de este tipo de herramientas son también sencillas de establecer. Para el dúo de herramientas mencionadas se pasa a especificar su funcionamiento.

*sadc* es la herramienta que permite obtener los datos de los parámetros monitorizados. A través de *sar*, la herramienta que complementa al software de medida, se indican los parámetros a monitorizar, así como el intervalo de recogida de muestras. Una opción muy útil de la que dispone es la posibilidad de monitorizar todos aquellos parámetros capturables por el software, obteniendo la información más completa posible.

Así, dispuesto el software de medida, y con el Ordenador Estudiante en reposo, el paso siguiente es ejecutar el software a analizar. Siguiendo este modelo de análisis, la herramienta de medida toma muestras antes, durante y después de la ejecución del software, lo que permite ver la carga que este último produce en el sistema.

Los resultados generados por *sadc* se encuentran en formato binario, por lo que, mediante *sar*, es posible realizar su correcta lectura. Algo similar sucedía en el primero de los análisis, donde se hacía uso del *profiler*.

Para el análisis a realizar se ha establecido la recogida de muestras de todos los parámetros disponibles, estableciendo un intervalo entre muestras de un minuto. Se escogió un minuto debido a los altos tiempos de procesamiento de cada una de las porciones de código con mayor tiempo de ejecución, según se vio en el *profiler*. Asimismo se ha tenido en cuenta el carácter iterativo del software, proporcionado por el análisis manual de tiempos.

Tras la ejecución del software y la posterior recogida de resultados desde la salida ofrecida por *sar*, se puede obtener una tabla de resultados, conteniendo una instantánea por minutos del uso de recursos del sistema en cada uno de los minutos monitorizados.

La tabla completa puede examinarse en el correspondiente apartado de Anexos, mientras que a continuación, en la Tabla 4, se muestra un resumen de dicha tabla, indicando únicamente los minutos más representativos de la ejecución realizada. Una representación gráfica de toda la ejecución se encuentra en la Figura 9.

| | **Uso de recursos (serie -Ofast)** | | |
|---|---|---|---|
| **Minuto** | **%CPU** | **E/S** | **RAM usada (MB)** |
| **0 - Reposo** | 0,01 | 0,13 | 0,00 |
| **1 - Inicio - Min RAM** | 87,59 | 18,58 | 749,73 |
| **22 - Max E/S** | 100,00 | 204,37 | 1049,61 |
| **66 - Max RAM** | 100,00 | 0,57 | 1078,11 |
| **67 - Fin** | 25,07 | 0,22 | 0,88 |
| **68 - Reposo** | 0,01 | 0,13 | 0,01 |

*Tabla 4: Tabla representativa del uso de recursos acorde a la ejecución del software.*

En esta tabla resumen se puede observar que se han seleccionado seis minutos concretos en relación a la ejecución completa del software. A continuación se pasan a describir.

En primer lugar, en el minuto 0 el sistema se encontraba en reposo, lo que permite obtener los datos iniciales de uso de recursos del sistema operativo. Todos los recursos se encuentran sin uso.

Pasando al primer minuto de ejecución del software, se puede observar el ascenso en el porcentaje de uso de la CPU, encontrándose cercano al 100%. Igualmente, durante este intervalo se realiza una pequeña cantidad de operaciones de entrada y salida. El uso de RAM asciende hasta aproximadamente los 750MB. De esta forma

se puede verificar que la ejecución inicial del software requiere de dicha cantidad de RAM para ser inicializado. Dicha cantidad de RAM se debe al establecimiento de todas las librerías, variables y constantes en memoria, así como a la lectura de cada uno de los ficheros involucrados en la ejecución.

Seguidamente, y hasta terminar la ejecución del software, el sistema se mantiene al 100% de utilización de la CPU, con un ligero aumento creciente, minuto a minuto, de la memoria RAM utilizada.

Estos incrementos en la memoria tienen lugar debido a nuevos establecimientos de variables en el tiempo. Asimismo, las optimizaciones impuestas por el compilador mediante la baliza anteriormente mencionada también tienen su efecto en el uso de recursos.

En el minuto 22 se destaca el número de entradas y salidas efectuadas, disparando también el uso de RAM. En este punto el software termina la ejecución de la última de las llamadas a la función Adicional y realiza los procedimientos previos a la ejecución del cuarto bloque y, en especial, a las llamadas a la última de las funciones, la función Interna.

Desde este punto, y hasta llegar al minuto 66, el uso de RAM continúa creciendo durante la ejecución, llegando al consumo aproximado de 1GB, con respecto a la instantánea en reposo. Mediante este ascenso continuado en el uso de RAM se puede verificar que durante toda la ejecución del software se están realizando nuevas escrituras, bien sea accediendo desde disco (como se puede observar asimismo mediante el número de E/S), o como resultado de las llamadas efectuadas a la función Interna.

En el minuto 67 se finaliza la ejecución del software, liberando los recursos. Finalmente, en el minuto 68 puede observar, nuevamente, que el uso de recursos vuelve a ser nulo, indicando la inactividad del sistema.

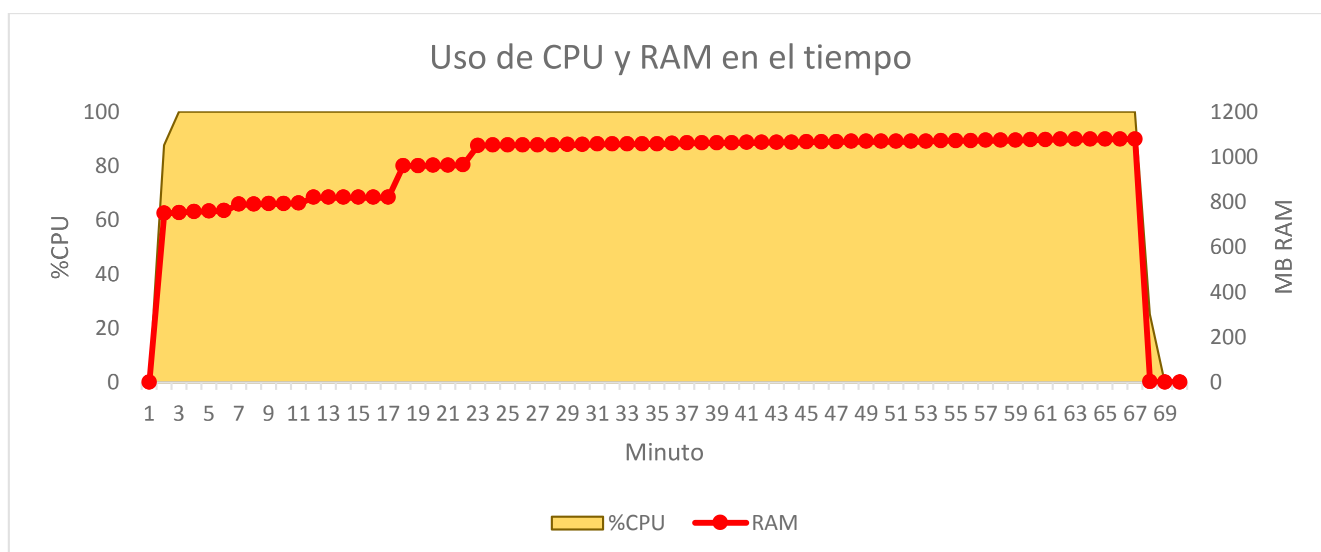


*Figura 9: Uso de los recursos principales en el tiempo de ejecución del software.*

**Objetivos principales determinados en este capítulo:**

- Aplicar los modelos seleccionados sobre los puntos localizados.
- Verificar las optimizaciones realizadas mediante las pruebas, utilizando los sistemas disponibles.
- Caracterizar los resultados obtenidos.

# Capítulo 4
# Paralelización mediante OpenMP

Durante este capítulo se van a explorar las capacidades del modelo paralelo que implementa OpenMP, haciendo uso de los recursos relacionados con la CPU.

El proceso a seguir comprende el desarrollo incremental de funcionalidad paralela, con el fin último de obtener un software cuyo tiempo de ejecución se encuentre optimizado.

## 4.1. Introducción. Limitaciones técnicas del modelo basado en CPU.

OpenMP es un modelo de procesamiento paralelo en memoria compartida. Este tipo de paralelismo se consigue mediante el modelo *fork-join*, es decir, mediante el desdoblamiento del hilo principal de procesamiento en varios hilos paralelos, para realizar la misma, una parte o distintas funciones durante el mismo tiempo de ejecución, maximizando el uso de los recursos. Finalmente, estos hilos de ejecución, que colaboran entre sí, reúnen su trabajo para terminar la ejecución en equipo a la que hayan sido sometidos.

OpenMP está basado en directivas de compilador, denominadas *pragmas*, lo que significa que el comportamiento final del software depende de la plataforma sobre la que se ejecute, al igual que ocurre con las librerías que mantienen C, por ejemplo. Consecuentemente, la compilación, el comportamiento y el tiempo de ejecución pueden diferir dependiendo del entorno.

Especificada la base de funcionamiento de OpenMP, se puede observar que sus limitaciones se encuentran en el propio modelo de cómputo basado en *fork-join* y en su caracterización intrínseca a la plataforma software en la que el modelo se ejecute.

Haciendo referencia al tipo de recursos utilizables por este modelo, no parece existir ninguna limitación en cuanto al uso de las CPUs y memorias actualmente disponibles, a excepción de los límites que el propio hardware impone.

De forma automática, el modelo es capaz de asignar un número de hilos para cada porción paralelizada, así como una cantidad conveniente de memoria a cada hilo de procesamiento. Este control, sin embargo, puede ser establecido también por el desarrollador, realizando un ajuste más fino.

Se debe tener en cuenta que, dependiendo de cómo se realice el establecimiento de la estructura paralela del software, se puede incurrir en un uso inadecuado de la memoria RAM disponible en el sistema, penalizando el rendimiento general de este último. Mientras se realiza el procesamiento paralelo, las variables que toman parte de ello deben, o no, según el caso, ser accesibles por el resto de hilos.

Dependiendo de la privacidad de las variables para cada uno de los hilos que forman el grupo de trabajo, la ejecución paralela podría traducirse en la multiplicación, en el segmento paralelo, de la memoria RAM requerida. Esta multiplicación puede tener un factor máximo, en el peor caso, igual al número de hilos establecidos, si se quiere un procesamiento 100% privado para cada hilo, lo cual no significa necesariamente que este vaya a realizarse mejor. De hecho, generalmente este tipo de procesamiento extremadamente privado suele concluir en resultados incorrectos, ya que el modelo de procesamiento se encuentra diseñado para que cada hilo pueda colaborar en la tarea principal para la que ha sido generado.

Por lo tanto, se puede concluir que, si bien el modelo de OpenMP puede mejorar el rendimiento de un software, debe emplearse de manera responsable, tratando de hacer un uso eficiente de los recursos.

**Objetivos principales a desarrollar mediante este modelo:**

- Aplicar los mecanismos que el modelo proporciona de manera responsable sobre las diversas porciones de código sujetas a ser paralelizadas.
- Incorporar los desarrollos realizados, tratando de obtener la versión más eficiente posible del software.
- Caracterizar las mejores soluciones desarrolladas y establecer una versión final para el modelo implementado.

## 4.2. Paralelización de la función Adicional.

Como consecuencia de su situación en el flujo de ejecución, el primer proceso de optimización consiste en la paralelización de la función Adicional.

La función se encuentra formada por una serie de asignaciones y ajustes de variables, acompañados de una serie de pequeños bucles, junto al bucle objetivo, considerado por emplear un tiempo de ejecución que, además de ser notable, merece ser reducido en las primeras iteraciones de forma que se aceleren las pruebas durante el desarrollo restante del trabajo. Este bucle es el encargado del procesamiento intensivo en relación a la comparación de los átomos de la celda con defecto respecto de la red perfecta.

**Descripción de la función a paralelizar.**

Durante el procesamiento del segundo bloque se hacen tres llamadas a esta función (una para cada uno de las celdas a procesar: la celda de Background y las celdas de los Defectos A y B), por lo que paralelizar este bucle permite no solo disminuir el tiempo de procesamiento de la función, sino reducirlo en un factor de tres sobre el tiempo total de cómputo del bloque 2.

El bucle implicado en el consumo es un bucle *for* en el cual se realizan varias comprobaciones mediante bifurcaciones a través de *ifs*, para finalmente realizar asignaciones y correcciones a diversas variables.

La característica que ralentiza el procesamiento de este bucle se encuentra localizada en el número de veces que debe ser iterado.

**Estrategia de optimización y creación de la zona paralela.**

Para el bucle objetivo, el proceso pasa por indicar las variables que deben ser consideradas como privadas y compartidas, el establecimiento del número de hilos que forman el grupo paralelo y el esquema de desglose del bucle entre los hilos.

Como no existen dependencias dentro del bucle, no es necesario modificar el código, aunque sí que se deben tener en cuenta las variables que son asignadas y/o escritas durante el procesamiento interno, estableciéndolas en consecuencia como privadas. En el Código 4 puede observarse la región paralela creada.

El número de hilos a utilizar es modificado según la variable de control establecida al efecto y el esquema de desglose elegido es el estático, garantizando así un enfoque de procesamiento homogéneo.

Un examen más exhaustivo de las decisiones a tomar durante el establecimiento de la región paralela se realiza en el siguiente incremento del desarrollo, la paralelización de la función Interna, debido a la no profundización en la paralelización de la función Adicional.

```
omp_set_dynamic(0);
#pragma omp parallel for default(shared) private(j, xij, yij, zij, r2)
num_threads(hilosdaes)
for(i=0;i<natoms;i++)
```

*Código 4: Establecimiento de la región paralela de la función Adicional.*

### 4.3. Paralelización de la función Interna.

El segundo proceso de optimización consiste en la paralelización de la función Interna, la cual es llamada dos veces por cada iteración realizada por la función Externa.

**Descripción de la función a paralelizar.**

La función se encuentra sometida a la anidación de un total de 3+1 bucles. Se han separado de esta manera ya que los primeros tres bucles anidados comprenden iteraciones desde 0 a ((2·NCELDASX)·(2·NCELDASY)·(2·NCELDASZ)), mientras que el último de ellos solamente comprende 2 iteraciones. De forma gráfica se puede observar esta anidación en el Código 5.

```
for (kx=0; kx< 2*NCELDASX; kx++)
{
    for (ky=0; ky< 2*NCELDASY; ky++)
    {
        for (kz=0; kz< 2*NCELDASZ; kz++)
        {
            for (ksl=0; ksl< 2; ksl++)
            {
```

*Código 5: Anidación de bucles dispuesta en la función Interna.*

En el interior de estos cuatro bucles, el software realiza el ajuste y asignación de varias variables de control, que son las que permiten seguidamente el procesamiento efectivo.

Una vez realizadas dichas asignaciones, se ejecuta un proceso de suma y producto de varios valores obtenidos previamente mediante los vectores correspondientes a la celda Background y al Defecto que se encuentre procesando en ese momento (bien sea el defecto A o B). El Código 6 detalla estos procesamientos, siguiendo asimismo la Ecuación 1 descrita en el Capítulo 2.

```
El0=El0 + ( cstcbrp[indicecb+0]*cstdrp[indiced+0] +
cstcbrp[indicecb+1]*cstdrp[indiced+1] + cstcbrp[indicecb+2]*cstdrp[indiced+2])/Y;

El2=El2 + ( cstcbrp[indicecb+3]*cstdrp[indiced+3] +
cstcbrp[indicecb+4]*cstdrp[indiced+4] + cstcbrp[indicecb+5]*cstdrp[indiced+5])*4.0/G;

El1=El1 + -1.0*( cstcbrp[indicecb+0]*(cstdrp[indiced+1] + cstdrp[indiced+2]) +
cstcbrp[indicecb+1]*(cstdrp[indiced+0] + cstdrp[indiced+2]) +
cstcbrp[indicecb+2]*(cstdrp[indiced+0] + cstdrp[indiced+1]) )*nu/Y;
```

*Código 6: Procesamientos realizados en relación a la celda Defecto considerada.*

Los índices accedidos en cada una de las operaciones se distancian, por iteración, en 6 unidades, desde que son asignados. Sin embargo, la asignación inicial del índice se realiza de forma no lineal, según puede observarse en el Código 7. Esta dependencia es inherente al comportamiento buscado en el procesamiento del software y no puede modificarse.

```
indicecb= kx*2*NCELDASY*2*NCELDASZ*2*6 + ky*2*NCELDASZ*2*6 + kz*2*6 + ksl*6;

indiced= auxx*2*NCELDASY*2*NCELDASZ*2*6 + auxy*2*NCELDASZ*2*6 + auxz*2*6 +
ksl*6;
```

*Código 7: Establecimiento de los índices de procesado de las celdas consideradas.*

Una vez procesadas todas las sumas parciales durante el procesado de los cuatro bucles, se realiza su multiplicación por un factor denominado *ocsi*. Tras este producto, se devuelve el control a la función principal.

**Estrategia de optimización.**

Descrito el funcionamiento de la función a paralelizar, se observa que un punto a tener en cuenta es el consistente en los tres primeros bucles anidados, dentro de los cuales se produce todo el procesamiento. El problema de estos tres bucles es que se encuentran desglosados según la posición de la celda analizar en el

espacio, existiendo por lo tanto un bucle para cada dimensión (X, Y, y Z), algo que puede traducirse, de manera menos gráfica, en un bucle efectivo de 8·NCELDASX·NCELDASY·NCELDASZ iteraciones.

La solución a llevar a cabo consiste en unir estos tres bucles en uno solo, de forma que se pueda utilizar el *pragma* que proporciona OpenMP para bucles *for*. La estrategia consiste en unificarlos.

Esto se puede realizar multiplicando los tres índices y creando un único bucle desde el valor inicial hasta el valor final, que ahora es el producto del valor máximo que alcanza cada uno de ellos.

Seguidamente, aplicando las propiedades de la división entera y el resto, es posible descomponer el valor de la variable de control del bucle en los valores correspondientes de X, Y, y Z, según se puede ver en el Código 8.

Esta es la primera porción de código que va a modificarse, para lo cual se obtendrá, asimismo, una mejora inherente en el rendimiento debido a la reducción de la sobrecarga implícita de incorporar tres bucles anidados.

```
for (k=0; k<8*NCELDASX*NCELDASY*NCELDASZ; k++)
{
  //Obtención de los valores de la red a través del índice k.
  kx=k/(4*NCELDASY*NCELDASZ);
  ky=(k%(4*NCELDASY*NCELDASZ))/(2*NCELDASZ);
  kz=(k%(4*NCELDASY*NCELDASZ))%(2*NCELDASZ);

  for (ksl=0; ksl< 2; ksl++)
  {
```

*Código 8: Descomposición del índice k en sus correspondencias, para cada una de las tres dimensiones.*

Con estos ajustes, si bien la lectura del código por un humano es ligeramente más complicada, se permite simplificar el funcionamiento de la función, haciéndola más sencilla y eficiente de paralelizar con OpenMP.

En esta iteración del desarrollo, el cuarto bucle *for*, el más interno, no se integra en el acabado de obtener, ya que dicho bucle únicamente se ejecuta dos veces por iteración.

**Creación de la zona paralela.**

El siguiente paso consiste en establecer la zona paralela haciendo uso de OpenMP. Para ello, se debe asignar una serie de hilos de procesamiento y comprobar si el rendimiento mejora. Mediante la directiva *omp_set_dynamic(0)* se desactiva la asignación automática de hilos por parte del modelo, permitiendo así al desarrollador asignar, manualmente, el número a utilizar para llevar a cabo el procesamiento de la región paralela.

Para que OpenMP sepa exactamente cómo debe procesar el bucle de forma paralela, se debe indicar, mediante el *pragma* dispuesto explícitamente a este fin, el número de hilos a utilizar. Este número de hilos se indica mediante una variable de control escrita por el desarrollador para este propósito.

Además, el procedimiento más importante para paralelizar este bloque de código consiste en indicar la visibilidad de todas las variables implicadas, qué hacer con los resultados parciales que cada hilo obtendrá durante su procesamiento particular, y cómo desglosar el procesamiento del bucle entre los hilos asignados. A continuación se pasan a describir todas estas decisiones.

**a) Visibilidad de las variables.**

Como se introdujo anteriormente, en OpenMP es posible indicar si las variables implicadas deben ser consideradas de forma privada o compartida, según el criterio del desarrollador. En este caso, las variables de control, así como la reasignación concreta de las variables correspondientes a las dimensiones a analizar y las utilizadas para almacenar las sumas parciales, deben ser consideradas de cómputo privado, de forma que cada hilo pueda ejecutar sus operaciones sin hacer colisión con el resto de hilos.

Sin embargo, otras variables de solo lectura, como los vectores correspondientes a las celdas a procesar, pueden computarse de manera compartida, evitando así su multiplicación en memoria, sobre los cuales no hay problemas de colisión, exceptuando las lecturas concurrentes inherentes al procesado. Este último inconveniente subyace a la gestión de memoria por parte del sistema operativo, aunque a este nivel apenas

añade sobrecarga, debido principalmente a la optimización interna del compilador y a la utilización de la memoria caché/paginación por parte de la CPU y el disco duro.

Las sumas parciales se van a procesar, en primer lugar, de manera independiente por cada hilo, para finalmente realizar una fusión final, de forma que se obtenga el mismo resultado que en el modelo en serie. Este proceso, denominado como reducción, puede llevarse a cabo debido a que la operación suma es conmutativa, por lo que el orden de los términos de la suma no es de consideración.

La manera de procesar esta última operación de suma de forma transparente a cada procesamiento particular consiste en hacer uso de la operación de reducción, en forma de suma, que se encuentra implementada en el modelo de OpenMP.

**b) Desglose de las iteraciones entre los hilos de ejecución.**

Para terminar, se debe establecer la organización del bucle entre los hilos de trabajo. Aunque, en primera instancia, este desglose pueda parecer independiente del tiempo final de ejecución ya que, en definitiva, todos los hilos deberían realizar un procesamiento similar, no es así.

Cada tipo de desglose afecta a los accesos a los vectores involucrados y, dependiendo de la naturaleza interna de cada bucle en concreto, puede conllevar una sobrecarga subyacente, por lo que es conveniente escoger el desglose más adecuado a cada tipo de paralelización.

En OpenMP se dispone de dos opciones principales de desglose, las cuales se denominan como estática y dinámica. Una variación de esta última se denomina como guiada.

La opción estática divide el bucle en porciones proporcionales a los hilos disponibles, realizando cada uno de ellos un procesamiento homogéneo. Esto permite eliminar la sobrecarga que implica escoger, en este caso, un desglose dinámico.

Por otro lado, el desglose dinámico, en contraposición, permite a OpenMP ir asignando iteraciones del bucle (cantidades controladas por una variable interna, denominada *chunk*) a los hilos, según terminan de procesar el *chunk* anterior asignado. La sobrecarga implicada en este desglose se puede ver de forma sencilla, ya que continuamente se realizan nuevas asignaciones, perdiendo tiempo útil de procesamiento (ya que la asignación dinámica no permite mejorar de ninguna manera el rendimiento en el proceso intrínseco que considera).

La variación de este desglose, denominada como guiada, se diferencia en que el *chunk* asignado a cada hilo varía según la cantidad de carga restante, siendo decreciente en iteraciones más avanzadas. Por lo tanto, al principio el *chunk* es grande, para terminar siendo muy pequeño. De la misma manera que ocurre con el esquema dinámico, la sobrecarga en este caso también se encuentra en la continua asignación de iteraciones a los hilos, la cual no conlleva ninguna mejora que permita considerar su utilización.

Por todo ello, y debido a la naturaleza homogénea del bucle, el esquema escogido es el estático: cada hilo obtiene una porción de iteraciones a procesar, en una única asignación.

```
omp_set_dynamic(0);
#pragma omp parallel for reduction(+:El0, El2, El1) default(shared)
private(kx, ky, kz, ksl, auxx, auxy, auxz, indicecb, induced)
num_threads(hilosdentro)
for (k=0; k<8*NCELDASX*NCELDASY*NCELDASZ; k++)
```

*Código 9: Establecimiento de la región paralela de la función Interna.*

Una vez establecidas todas las decisiones requeridas por el *pragma* de OpenMP, visibles en el Código 9, el software ya se encuentra en un estado que permite la prueba de las dos implementaciones llevadas a cabo hasta el momento (Adicional e Interna), y que utilizan el tiempo total de ejecución, examinando así la mejora en el rendimiento mediante la obtención de resultados que sean esclarecedores de la nueva configuración desarrollada.

La batería de pruebas a realizar comprende la ejecución del software utilizando para su procesamiento en un total de 2, 4 y 8 hilos en ejecución concurrente. Así se puede observar si el ajuste es adecuado y el rendimiento mejora convenientemente. Esta conveniencia se verificará si el rendimiento aumenta de forma lineal, es decir, si el tiempo de ejecución para la porción paralelizada se reduce en el mismo o similar factor al número de hilos asignados.

Se debe puntualizar que debido a que se dispone de tres sistemas hardware sobre los que realizar los pruebas, y que estos disponen de componentes distintos entre sí, así como de diferentes tecnologías, los resultados que se van a mostrar pueden diferir dependiendo del procesamiento realizado sobre cada porción de código concreta, especialmente teniendo en cuenta el juego de instrucciones, la funcionalidad y la frecuencia por núcleo de los que disponga cada uno de los sistemas.

Sin embargo, la referencia serie que se ha tomado para comparar y obtener los resultados de cada aceleración ha tenido la misma procedencia que los correspondientes resultados paralelos.

### i. Primera salida de resultados: Adicional e Interna.

Como se sabe, cada uno de estos sistemas de experimentación dispone de características bien diferenciadas, por lo que los resultados que se van a mostrar se han dividido precisamente según el sistema en el que se hayan obtenido. Asimismo, los datos correspondientes a la ejecución serie también se han obtenido de cada una de los correspondientes sistemas.

En primer lugar, se pasa a analizar el comportamiento de la ejecución de la función Adicional. Esta función es requerida en tres ocasiones, por lo que se dispone de tres muestras por cada experimento realizado. Así, se obtiene finalmente una media de tiempos a través de los tres obtenidos.

Se dispone de tiempos para la función original, su versión optimizada mediante el compilador, y la correspondiente versión paralelizada mediante el número de núcleos que en cada sistema se encuentre disponible.

Seguidamente se analiza el comportamiento de la función Interna, estableciendo la medida de sus tiempos. Junto a ellos se acompañan los valores de aceleración obtenidos para dicha función. De manera asociada se ha añadido el número de fallos de paginación que se han dado durante la ejecución del software, si dicho dato se encuentra disponible.

En las tablas relacionadas con la representación simultánea de las funciones Adicional e Interna, el tiempo total de ejecución del software se corresponde con la suma del tiempo obtenido para cada una de ellas.

Asimismo, todos los resultados temporales son totales, no proporcionales al número de hilos utilizados en cada prueba. En las pruebas realizadas, y como ya se especificó, todas las compilaciones son efectuadas mediante la baliza más agresiva de optimización proporcionada por cada versión del compilador disponible, bien sea *–Ofast*, u *–O3*, en el caso del Servidor Beta.

Esta notación se mantiene durante todas las exposiciones de resultados, a menos que específicamente sean modificadas.

Dispuesta la organización de los resultados, a continuación se pasan a describir.

#### a. Ordenador Estudiante.

| | Serie | | OpenMP | |
|---|---|---|---|---|
| | *1* | **1 opt (ref)** | **2** | **3** |
| **Adicional Back** | *922,77* | 433,64 | 223,55 | 147,10 |
| **Adicional A** | *918,10* | 425,39 | 221,70 | 148,56 |
| **Adicional B** | *917,17* | 414,58 | 221,62 | 146,00 |
| **Media** | ***919,35*** | **424,54** | **222,29** | **147,22** |

*Tabla 5: Representación comparativa de los resultados obtenidos para la función Adicional en el Ordenador Estudiante.*

En este sistema, de recursos más limitados, se han realizado las pruebas de experimentación, dentro de sus límites hardware. Para ellas, y según se puede observar en la Tabla 5, se puede observar una optimización creciente de la función Adicional.

| | Serie | | OpenMP | |
|---|---|---|---|---|
| | *1* | 1 opt (ref) | 2 | 3 |
| **Adicional** | *919,35* | 424,54 | 222,29 | 147,22 |
| **Speedup** | -> | **2,17** | **1,91** | **2,88** |
| **Interna** | *10167,27* | 4502,82 | 3494,17 | 3259,86 |
| **Speedup** | -> | **2,26** | **1,29** | **1,38** |
| **Page Faults** | *55208,00* | 1275,00 | 1302,00 | 1822,00 |

*Tabla 6: Representación comparativa de los resultados obtenidos para la función Interna en el Ordenador Estudiante.*

Los datos de aceleración obtenidos para esta función son adecuados. Se ha obtenido una aceleración correspondiente al número de hilos asignados.

En relación a la función Interna, los tiempos disminuyen, a pesar de obtener aceleraciones menores, no consiguiendo superar un factor de 1,5, llegando únicamente a 1,38 en la utilización de tres hilos, como puede observarse en la Tabla 6.

### b. Plataforma TFG.

| | Serie | | OpenMP | | |
|---|---|---|---|---|---|
| | *1* | 1 opt (ref) | 2 | 4 | 8 |
| **Adicional Back** | *1134,99* | 324,14 | 160,52 | 91,20 | 88,67 |
| **Adicional A** | *1135,50* | 325,35 | 160,50 | 91,33 | 89,07 |
| **Adicional B** | *1132,23* | 322,74 | 160,54 | 91,75 | 88,14 |
| **Media** | ***1134,24*** | **324,08** | **160,52** | **91,43** | **88,63** |

*Tabla 7: Representación comparativa de los resultados obtenidos para la función Adicional en la plataforma TFG.*

Pasando al siguiente sistema, se puede observar, de manera similar, una disminución considerable en los tiempos de ejecución para la función Adicional, según la Tabla 7.

Sin embargo, algo parece ocurrir en el paso de cuatro a ocho hilos de ejecución, obteniendo una aceleración muy similar a la obtenida en el caso inmediatamente anterior, lo que puede indicar que existe algún problema de rendimiento interno, al haber obtenido un paso de aceleración muy leve en relación a los anteriores.

| | Serie | | OpenMP | | |
|---|---|---|---|---|---|
| | *1* | 1 opt (ref) | 2 | 4 | 8 |
| **Adicional** | *1134,24* | 324,08 | 160,52 | 91,43 | 88,63 |
| **Speedup %** | -> | **3,50** | **2,02** | **3,54** | **3,66** |
| **Interna** | *11387,37* | 3922,70 | 2762,79 | 2033,65 | 2216,22 |
| **Speedup %** | -> | **2,90** | **1,42** | **1,93** | **1,77** |
| **Page Faults** | *779,00* | 782,00 | 1311,00 | 1828,00 | 1933254,00 |

*Tabla 8: Representación comparativa de los resultados obtenidos para la función Interna en la Plataforma TFG.*

Revisando las aceleraciones obtenidas en la Tabla 8 para la función Adicional, se puede confirmar mediante esta medida que el rendimiento decrece en el último paso de hilos de ejecución.

En relación a los resultados de la función Interna, se observa una aceleración, debida al compilador, cercana a tres.

En relación a la aceleración obtenida mediante la paralelización de dicha función, ocurre algo similar a lo descrito en la función Adicional. El rendimiento mejora, aunque en menor medida que en el caso anterior. De nuevo, para el caso en el que se asignan ocho hilos de ejecución, la aceleración incluso decrece, si se toma como referencia la misma prueba para cuatro hilos.

Mediante los datos acerca de los fallos de paginación se puede observar la potencial causa de la pérdida de rendimiento y es que dichos fallos se disparan para este último caso. Algo que parece indicar la saturación del sistema, debido probablemente al acceso a la pila o a la extenuación de la memoria RAM del sistema.

La utilización de la baliza en la compilación del código influye directamente en el uso realizado de la pila del sistema, aumentando considerablemente su utilización en ejecuciones más agresivas, lo que puede haber provocado la saturación del recurso, impidiendo así lograr rendimientos más acordes al número de hilos considerado.

#### c. Servidor Beta.

| | Serie | | OpenMP | | |
|---|---|---|---|---|---|
| | ***1*** | **1 opt (ref)** | **2** | **4** | **8** |
| **Adicional Back** | *1596,50* | 365,25 | 188,89 | 94,47 | 48,01 |
| **Adicional A** | *1587,21* | 364,82 | 188,45 | 94,32 | 47,97 |
| **Adicional B** | *1589,05* | 364,59 | 188,62 | 94,38 | 47,95 |
| **Media** | ***1590,92*** | **364,88** | **188,65** | **94,39** | **47,98** |

*Tabla 9: Representación comparativa de los resultados obtenidos para la función Adicional en el Servidor Beta.*

Finalmente, para el último sistema puede observarse el comportamiento de la función Adicional según se asignan más hilos de ejecución. Como se puede observar, los tiempos son decrecientes según aumenta el número de hilos, según informa la Tabla 9.

| | Serie | | OpenMP | | |
|---|---|---|---|---|---|
| | ***1*** | **1 opt (ref)** | **2** | **4** | **8** |
| **Adicional** | *1590,92* | 364,88 | 188,65 | 94,39 | 47,98 |
| **Speedup** | -> | **4,36** | **1,93** | **3,82** | **7,21** |
| **Interna** | *15145,25* | 9980,81 | 7232,86 | 6184,05 | 5577,31 |
| **Speedup** | -> | **1,52** | **1,33** | **1,61** | **1,79** |

*Tabla 10: Representación comparativa de los resultados obtenidos para la función Interna en el Servidor Beta.*

En la Tabla 10 puede observarse que la primera optimización, debida al compilador, consigue una gran mejora en el rendimiento. Tomando este último dato como referencia, la paralelización llevada a cabo en esta función resulta muy efectiva, obteniendo aceleraciones cercanas al número de hilos establecidos, lo que indica una aceleración lineal.

Pasando a la función Interna, se puede observar una aceleración, más leve en este caso, debida a la optimización realizada por el compilador. Al contrario de los resultados obtenidos para la función Adicional, la paralelización de la función Interna no ha conseguido reducir los tiempos en una escala lineal, logrando una aceleración cercana a dos, menos eficiente teniendo en cuenta que para ello se han utilizado ocho hilos de ejecución.

De manera teórica se puede decir que el rendimiento depende directamente del sistema de prueba, como se puede observar en los tiempos de ejecución en serie y en serie optimizado por el compilador.

Esta es una de las razones de realizar incrementos sobre el proceso de optimización, con la motivación de desarrollar un código que aproveche mejor las capacidades de cada sistema.

Para terminar, se debe mencionar que, para este sistema, por su manera de procesar trabajos, a través de colas, no se han podido obtener los datos en relación a los fallos de paginación.

Mediante las Figuras 10 y 11 se puede observar el rendimiento conseguido, para cada uno de los sistemas, de manera gráfica.

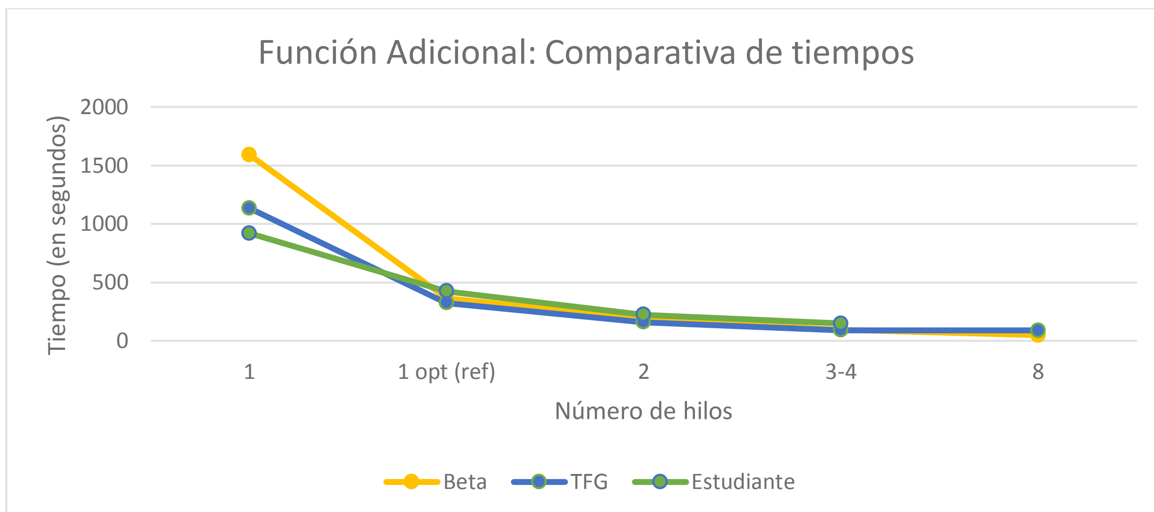


*Figura 10: Gráfico ilustrativo de los tiempos obtenidos para la función Adicional.*

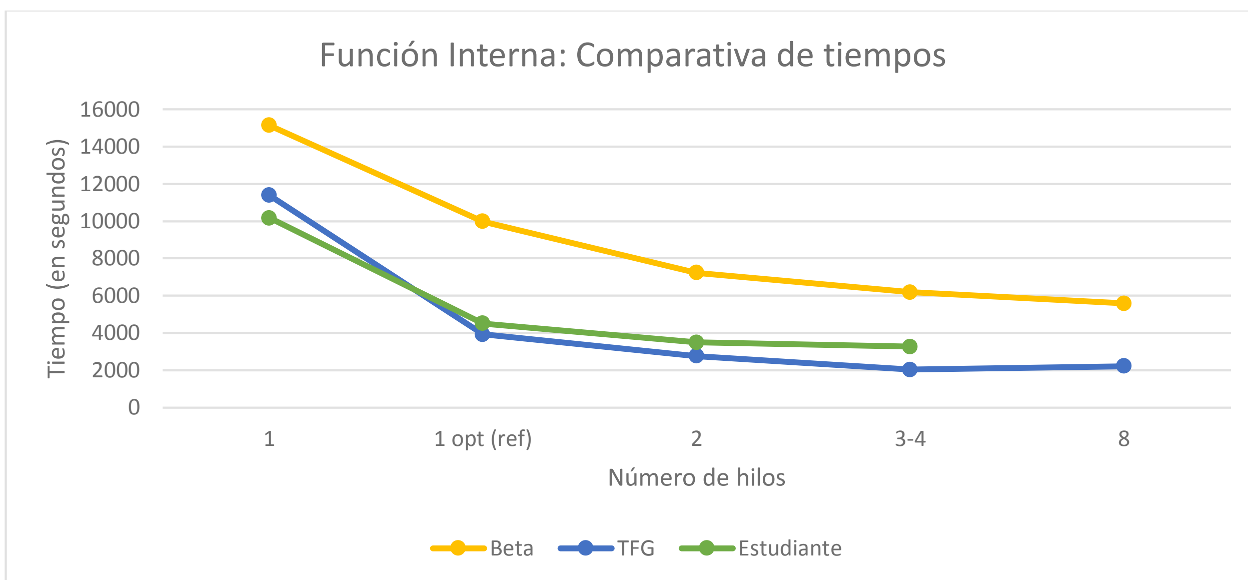


*Figura 11: Gráfico ilustrativo de los tiempos obtenidos para la función Interna.*

## 4.4. Paralelización de la función Externa.

Llega el momento de continuar con el siguiente incremento y comenzar a desarrollar una paralelización para la función Externa.

La idea consiste en mejorar el rendimiento del código que realiza las llamadas a la función Interna, intentando lograr, de manera complementaria, un aumento en el rendimiento.

El código asociado a este bloque, a pesar de su relativa similitud a la función Interna, es un código más complicado, cuya paralelización, si bien se va a basar en la ya comentada, requiere la solución de problemas intermedios que surgen de su mayor complejidad.

Con la finalidad de obtener una mejor comprensión del proceso a realizar, se va a analizar la estructura del código implicado.

**Descripción de la función a paralelizar.**

En primer lugar, durante el inicio del bloque, el código establece ciertas variables de control para, seguidamente, ejecutar una serie de bucles que recuerdan perfectamente, por su estructura, al modelo ya paralelizado en la función Interna. Esta serie de bucles, visibles mediante el Código 10, consiste en la anidación de tres iteradores. Los índices, en este caso, se inician desde 0 hasta ((2·NCELDASX)·(2·NCELDASY)·(2·NCELDASZ)), con un paso, para cada bucle, de dos.

En el interior de los tres bucles se encuentra un bucle adicional que realiza cuatro iteraciones por cada una de las iteraciones de los bucles previos. Este bucle es el que permite establecer, mediante asignaciones y ajustes en variables, las combinaciones entre cada una de las celdas Defecto y la celda Background, y es lo que proporciona, en definitiva, la capacidad de realizar un procesamiento exhaustivo y detallado haciendo uso de la función Interna.

```
// Nos movemos de celda en celda ==> hay que incrementar k en 2 unidades
for (kx=0; kx< 2*NCELDASX; kx=kx+2)//Se salta de celda en celda ==> kx = kx +
2
{
  for (ky=0; ky< 2*NCELDASY; ky=ky+2)//Se salta de celda en celda ==> ky = ky
+ 2
  {
    for (kz=0; kz< 2*NCELDASZ; kz=kz+2)//Se salta de celda en celda ==> kz =
kz + 2
    {
      for( kj = 0 ; kj < 4; kj ++)
      {
```

*Código 10: Bucles anidados, dispuestos en la función Externa.*

Tras este proceso previo, se efectúan dos llamadas a la función Interna, una para cada Defecto, respectivamente. Primeramente se llama a la función para el procesado del Defecto A. Después se establece una serie de últimas asignaciones y se realiza la escritura de los resultados en el fichero de salida. Este proceso se puede ver en el Código 11.

Seguidamente se repiten las operaciones sobre el Defecto B. Una vez finalizadas, la iteración del bucle termina y se continúa con la siguiente, teniendo en cuenta, de nuevo, que se trata de procesar cuatro bucles anidados. Con esto termina la porción de código considerada.

```
//Defecto A
EnElastica[0]=0.0; EnElastica[1]=0.0; EnElastica[2]=0.0;
calcEnerEl(cstcbrp,cstdarp,-kxaux,-kyaux,-kzaux,EnElastica);
////Defecto A - Coordenada X
xaux=xdefa*semired+kxaux*semired;
if(xaux>=lsidex)
    xaux=xaux-lsidex;
else if(xaux<0)
    xaux=xaux+lsidex;
////Defecto A - Coordenada Y
yaux=ydefa*semired+kyaux*semired;
```

```
if(yaux>=lsidey)
    yaux=yaux-lsidey;
else if(yaux<0)
    yaux=yaux+lsidey;
////Defecto A - Coordenada Z
zaux=zdefa*semired+kzaux*semired;
if(zaux>=lsidez)
    zaux=zaux-lsidez;
else if(zaux<0)
    zaux=zaux+lsidez;
////Defecto A - Imprimir en fichero
fprintf(salida,"%lf        %lf        %lf        %lf        %lf        %lf
%lf\n",xaux,yaux,zaux,factor*(EnElastica[0]+EnElastica[1]+EnElastica[2]),fa
ctor*(EnElastica[0]),factor*(EnElastica[1]),factor*(EnElastica[2]));
```

*Código 11: Procesamiento del cuarto bloque para el Defecto A.*

**Estrategia de optimización.**

La primera idea de paralelización que surge es la de repetir el proceso ya realizado para la función Interna. Se dispone de tres bucles anidados de idénticas características, acoplados a un cuarto bucle, más pequeño. En parte es posible llevar a cabo el mismo proceso, ya que se puede realizar la agrupación de los tres primeros bucles en uno único que después asigne los valores correspondientes a las dimensiones a considerar de X, Y, y Z, ya que el procesamiento es exhaustivo y recorre todos las posibles combinaciones.

Sin embargo, aparecen problemas asociados a la naturaleza de este código, en concreto en relación a la escritura de los resultados en el fichero de salida.

El problema principal surge del inherente esquema paralelo: no puede conocerse el orden en el que va a llevarse a cabo la escritura de los resultados, debido al funcionamiento independiente de cada hilo asignado, a menos que se obligue a mantenerlo, estableciendo en ese caso una penalización demasiado grande como para ser admitida.

Esto es debido a que dicho mecanismo obliga a que todos los hilos deban esperar la escritura del hilo que corresponda en cada momento, lo que puede ralentizar el proceso de aceleración en un factor, en caso extremo, igual al número de hilos asignados (ya que la escritura debe realizarse en serie). Tal es así que es necesario encontrar una solución alternativa.

Esta solución alternativa existe, y no solo permite llevar a cabo la paralelización sin pasar por un mecanismo que reduce parte del código a una ejecución serie, sino que además consigue una mejora inherente del código, afín al modelo que propone OpenMP.

Esta solución consiste en la creación, en la región paralela, de ficheros temporales que den soporte al procesamiento particular que cada hilo de trabajo realice. La paralelización pasa entonces de un primer estado en el que se debía paralelizar un bucle o conjunto de bucles, a paralelizar la función completa.

**Creación de la zona paralela.**

La implementación de la solución requiere la creación de una región paralela en la que se asigne a cada hilo una parte proporcional del procesamiento total. El punto de partida de esta división se encuentra en el bucle. Sin embargo, esta vez no se realiza el desglose haciendo uso de alguno de los esquemas que OpenMP incluye, debido a que la paralelización no incluye únicamente el bucle, sino además las asignaciones previas, incluyendo la creación de cada fichero temporal y las posteriores escrituras en dichos ficheros. Asimismo, el grupo paralelo debe fusionarse al final para colaborar en la obtención del resultado final esperado, según su homólogo serie.

**a) División equitativa de las iteraciones entre el número de hilos disponible.**

Un problema subyacente a la solución que se está mostrando tiene que ver con el número de iteraciones a realizar y el número de hilos asignados al procesamiento. Aplicando el procedimiento de la división entera y el resto, se puede asignar la misma cantidad de procesamiento a todos los hilos (siguiendo el desglose estático de OpenMP, que en este caso garantiza el orden buscado), además de una pequeña cantidad de hilos, equivalente al resto de dicha división (cuyo valor máximo equivale al número de hilos asignados menos uno).

El Código 12 modela el proceso de división del trabajo, mientras que el Código 13 modela los puntos de inicio y fin para cada uno de los hilos, así como el inicio de la región paralela.

```
//Creamos los ficheros parciales y los asignamos al array.
  for(k=0; k<hilos;k++)
  {
    //Convertimos el integer (k) en caracter.
    sprintf(nombre, "%d", k);
    //Creamos el fichero según el índice del hilo.
    remove(nombre);
    nombre_hilos[k]= fopen(nombre,"w");
  }

//Dividimos el número total de iteraciones en bloques de acuerdo al número de
hilos.
int bloque = NCELDASX*NCELDASY*NCELDASZ / hilos;
int ultimo = NCELDASX*NCELDASY*NCELDASZ % hilos;
```

*Código 12: Creación de los ficheros temporales.*

Esta última asignación puede producir una pequeña sobrecarga, sin apenas notoriedad, ya que se debe tener en consideración que este hilo sobrecargado estaría procesando una cantidad de iteraciones adicionales varios ordenes inferior a la que el desglose estático asigna de manera general.

```
//Establecemos variables antes de entrar en la zona paralela.
int numerohilo;
int comienzo;
int final;

//Comienzo de la zona paralela.
omp_set_dynamic(0);
#pragma omp parallel default(shared) private(numerohilo, comienzo, final, k,
kx, ky, kz, kj, kincx, kincy, kincz, xaux, yaux, zaux, xauxb, yauxb, zauxb,
kxaux, kyaux, kzaux, EnElastica, EnElasticaB) num_threads(hilosfuera)
{
    numerohilo = omp_get_thread_num();
    comienzo = (numerohilo)*bloque;
    final = comienzo + bloque;
    //Asignamos el resto de la división de los bloques al último hilo.
    if (numerohilo ==(hilos-1))
      final = final + ultimo;

    //Comienzo del procesamiento paralelo del cálculo de energía
    for (k=comienzo; k<final; k+=1)
    {
      //Obtención de los valores de la red a través del índice k.
      kx=2*(k/(NCELDASY*NCELDASZ));
      ky=2*((k%(NCELDASY*NCELDASZ))/(NCELDASZ));
      kz=2*((k%(NCELDASY*NCELDASZ))%(NCELDASZ));

      for( kj = 0 ; kj < 4; kj ++)
      {
```

*Código 13: Establecimiento del grupo de hilos de trabajo para la función Externa.*

Por otro lado, es requisito imprescindible llevar a cabo la creación de tantos ficheros temporales como hilos disponibles para efectuar el procesamiento, almacenando así los resultados parciales obtenidos, para finalmente ser concatenados en el fichero de salida. Este nuevo proceso, junto con las escrituras parciales y la escritura final, también involucran una pequeña sobrecarga adicional.

Sin embargo, esta sobrecarga es pequeña y se puede aceptar debido a la mejora en el rendimiento que internamente conlleva, destacando el procesamiento independiente, mientras se obtiene una ejecución limpia.

Una vez todos los hilos terminan su procesamiento paralelo, existe un punto de unión (*join*) en el que los hilos deben cohesionar, algo que efectivamente se realiza mediante el mecanismo de barrera que incorpora OpenMP, y que permite sincronizar el grupo de hilos de trabajo, esperando por aquellos que aún estén realizando algún procesamiento (lo cual se traduce en este caso en el hilo ligeramente sobrecargado).

El esquema escogido se ha dispuesto con la motivación de que la espera por este mecanismo de sincronización fuera mínima, ya que todos los hilos computan una cantidad igual (o muy similar en el caso del hilo sobrecargado) de iteraciones.

**b) Concatenación del trabajo realizado en paralelo.**

Sincronizados todos los hilos, la etapa paralela finaliza, pero aún es necesario el procesado final de los resultados, que involucra la concatenación en orden de los ficheros temporales generados por el grupo de trabajo paralelo.

Este proceso añade una ligera sobrecarga, compensada sin embargo con la solución implementada al resolver, definitivamente, la obligación de mantener el orden en el procesamiento paralelo.

Finalmente, tras la lectura en orden de los ficheros temporales, la salida final es obtenida y el software obtiene el mismo resultado que su homólogo en serie, como puede observarse en el Código 14.

```
for(k=0; k<hilos;k++)
  {
    fclose(nombre_hilos[k]);
    sprintf(nombre, "%d", k);
    nombre_hilos[k]= fopen(nombre,"r");
    //Leemos el fichero k-esimo hasta que no haya más.
while(fscanf(nombre_hilos[k],"%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",&xaux,&
yaux,&zaux,&resul0,&resul1,&resul2,&resul3) == 7)
    {
fprintf(salida,"%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\n",xaux,yaux,zaux,resul0
,resul1,resul2,resul3);
        lineas++;
    }
    fclose(nombre_hilos[k]);
    remove(nombre);
  }
```

*Código 14: Concatenación de los ficheros temporales creados por los hilos de proceso.*

**c) Establecimiento de la región paralela.**

Esta implementación en el código no podría efectuarse sin hacer uso, de nuevo, de un *pragma* por parte de OpenMP, en este caso, para ejecutar una sección del código de forma paralela, como se ha podido ver en el Código 13.

Las opciones en dicho *pragma* establecen, por un lado, el número de hilos que forman parte del grupo de trabajo y que, de nuevo, se encuentra controlado por una variable de control al efecto.

Por otro lado, se debe indicar, con mayor importancia sí cabe en este caso, el ámbito de cada una de las variables, bien se deban considerar privadas o compartidas. Las variables que solo se utilizan con propósito de lectura (ya que no existen de escritura disjunta en este caso) pueden establecerse como compartidas, mientras que aquellas que deban ser leídas y escritas, deben declararse como privadas. De esta forma se evitan las condiciones de carrera, los autobloqueos, la necesidad de emplear mecanismos de escritura atómica y la aparición de otros comportamientos no previstos.

Con esto concluye la implementación de del tercer incremento en el desarrollo paralelo del software. En este momento, disponiendo de una paralelización sobre la función Adicional, en conjunción con las realizadas sobre las funciones Externa e Interna, se dispone de una versión del software sobre la que ejecutar una segunda batería de pruebas para obtener nuevos resultados.

La batería de pruebas, de forma análoga a la anterior, se va a realizar utilizando 2, 4 y 8 hilos de ejecución, de forma que se pueda observar si el rendimiento mejora conforme el número de hilos asignados aumenta.

### ii. Segunda salida de resultados: Externa.

Con la finalidad de ver de forma más adecuada la comparación de resultados, se va a mostrar la tabla de resultados para la función Externa junto a los resultados más característicos previamente obtenidos.

De nuevo, dichos resultados van a ser comentados haciendo referencia al sistema sobre el que se haya llevado su ejecución.

#### a. Ordenador Estudiante.

| | Serie | OpenMP | |
|---|---|---|---|
| | 1 opt (ref) | 2 | 3 |
| **Adicional** | 424,54 | 222,29 | 147,22 |
| **Speedup** | -> | **1,91** | **2,88** |
| **Externa** | 4502,82 | 3015,10 | 2709,58 |
| **Speedup** | -> | **1,49** | **1,66** |
| **Page Faults** | 1275,00 | 1300,00 | 1816,00 |
| **Interna** | | 3494,17 | 3259,86 |
| **Speedup** | -> | **1,29** | **1,38** |
| **Page Faults** | | 1302,00 | 1822,00 |

*Tabla 11: Representación comparativa de los resultados para la función Externa en el Ordenador Estudiante.*

El rendimiento obtenido para este primer sistema, siguiendo la Tabla 11, mejora con respecto a la paralelización Interna, pero no de forma lineal, por lo que la aceleración conseguida no es adecuada al número de hilos asignado al software.

#### b. Plataforma TFG.

| | Serie | OpenMP | | |
|---|---|---|---|---|
| | 1 opt (ref) | 2 | 4 | 8 |
| **Adicional** | 324,08 | 160,52 | 91,43 | 88,63 |
| **Speedup %** | -> | **2,02** | **3,54** | **3,66** |
| **Externa** | 3922,70 | 2545,57 | 1533,90 | 1399,99 |
| **Speedup %** | -> | **1,54** | **2,56** | **2,80** |
| **Page Faults** | 782,00 | 1806,00 | 1308,00 | 1840,00 |
| **Interna** | | 2762,79 | 2033,65 | 2216,22 |
| **Speedup %** | -> | **1,42** | **1,93** | **1,77** |
| **Page Faults** | | 1311,00 | 1828,00 | 1933254,00 |

*Tabla 12: Representación comparativa de los resultados para la función Externa en la Plataforma TFG.*

En este segundo sistema se puede observar algo similar, según se puede ver en la Tabla 12. El rendimiento mejora, pero no se obtiene una cifra de aceleración adecuada, aunque sí hay una diferencia relativamente sustancial con su homóloga para la paralelización Interna. La aceleración se encuentra en este momento en una cifra cercana a tres, insuficiente para el número de hilos asignados, ocho.

En este punto, asimismo se puede observar que el número de fallos de paginación se encuentra en un valor más bajo, de nuevo en referencia a la versión de la función Interna paralelizada, en la cual dicho valor creció varios órdenes de magnitud.

Se puede acotar este problema de rendimiento al procesamiento inherente de la función Interna, el cual, debido a su carácter de función, puede estar aumentando considerablemente el tiempo de ejecución final al requerir de una mayor cantidad de cambios de contexto y de un mayor uso de RAM para obtener el mismo resultado que obtiene el resto de versiones del software. Esto es debido, asimismo, a la paralelización de una función que es llamada y, por lo tanto, es creada y destruida un número notable de veces.

### c. Servidor Beta.

| | Serie | OpenMP | | |
|---|---|---|---|---|
| | 1 opt (ref) | 2 | 4 | 8 |
| **Adicional** | 364,88 | 188,65 | 94,39 | 47,98 |
| **Speedup** | -> | **1,93** | **3,82** | **7,21** |
| **Externa** | 9980,81 | 7513,84 | 5189,11 | 4155,77 |
| **Speedup** | -> | **1,33** | **1,92** | **2,40** |
| **Interna** | | 7232,86 | 6184,05 | 5577,31 |
| **Speedup** | -> | **1,33** | **1,61** | **1,79** |

*Tabla 13: Representación comparativa de los resultados para la función Externa en el Servidor Beta.*

Realizando una comparación entre los dos tipos de paralelización llevados a cabo para este sistema, para el cual se pueden ver los resultados a través de la Tabla 13, se puede observar una mejora en las aceleraciones conseguidas, si bien el rendimiento continúa sin ser acorde al número de hilos asignados. Así, para ocho hilos, la aceleración se encuentra en 2,40, un número alejado del esperado, si bien mejora el obtenido en la paralelización de la función Interna.

Mediante la Figura 12 pueden observarse, de manera gráfica, los resultados obtenidos para cada uno de los sistemas.

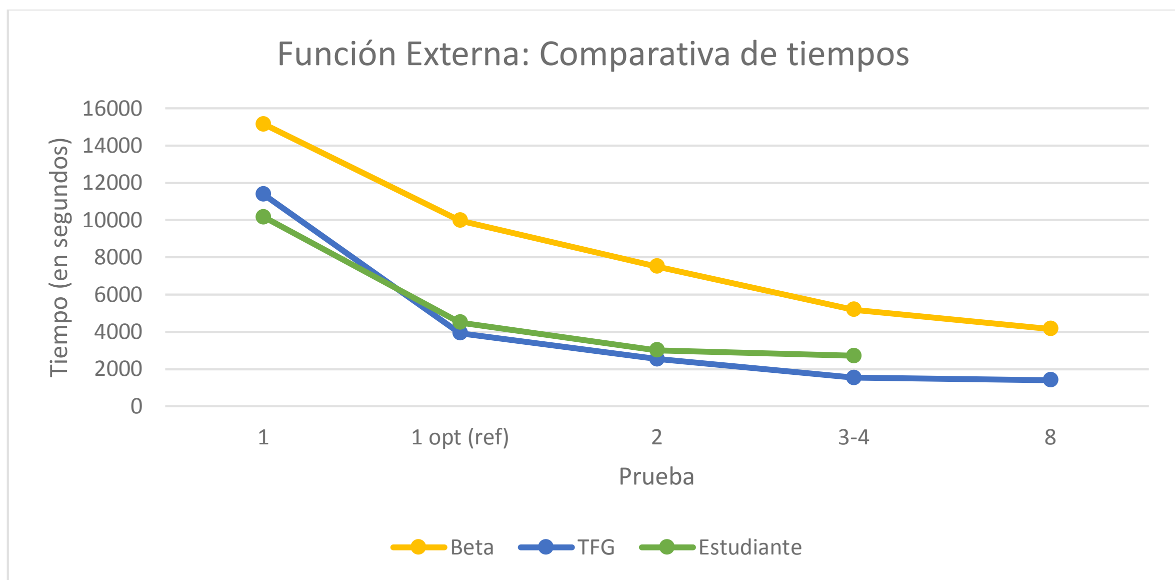


*Figura 12: Gráfico ilustrativo de los tiempos obtenidos para la función Externa.*

## 4.5. Paralelización combinada de las funciones Externa e Interna.

Hasta el momento se ha avanzado en el proceso de optimización del código fuente, consiguiendo mejorar los tiempos de las principales funciones de consumo dentro del software.

Con dichas funciones paralelizadas, el siguiente incremento en el desarrollo consiste en intentar aprovechar este trabajo para generar un código aún más eficiente mediante la combinación de las dos porciones paralelas implicadas en el procesamiento del cuarto bloque, de forma que, complementando ambos trabajos, se obtenga una mejora más notable.

Como se sabe, la función Externa realiza un gran número de llamadas a la función Interna, la cual, por otro lado, consume un tiempo que si bien se podría considerar despreciable, se ha comprobado mediante los resultados tras su paralelización que supone un coste importante para la ejecución del software. En cada una de las invocaciones a la función Interna se está fomentando la aparición de una sobrecarga debido al mecanismo de llamada.

Esta situación empeora al tener en cuenta que el número de llamadas a la función Interna depende del número de iteraciones de la función Externa, y que la cantidad se sabe que es importante, involucrando, según el análisis preliminar, un 76,5% aproximadamente del tiempo de ejecución total, según el código optimizado por el compilador.

En consecuencia, parece ser necesario intentar minimizar este problema que subyace el lenguaje de programación hasta el nivel de sistema operativo. La idea que podría permitir mejorar el rendimiento se encuentra en la combinación de ambas zonas paralelas, de forma que colaboren entre ellas para lograr un rendimiento superior, ya que por sí mismas se encuentran en un alto grado de paralelización.

En términos prácticos, los sistemas de prueba disponen de hasta un máximo de ocho núcleos de procesamiento hardware, lo que permite realizar dos tipos de combinaciones que resulten en el uso efectivo de esta cantidad de núcleos por parte de las funciones involucradas, según se puede observar en la Figura 13.

Dichas combinaciones incluyen el establecimiento de dos o cuatro hilos para cada una de las funciones, de forma que su producto dé lugar a los ocho núcleos disponibles.

Así pues, dado que se dispone de recursos hardware suficientes, se va a comprobar si la mejora mediante la combinación de ambos paralelismos mejora el rendimiento.

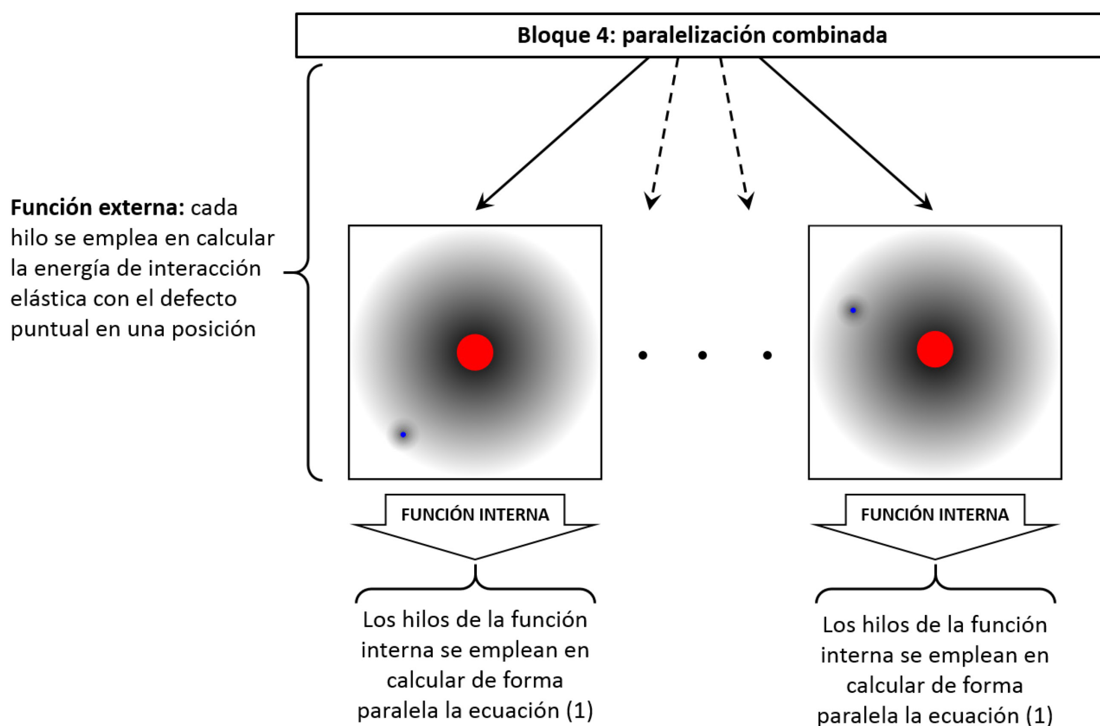


*Figura 13: Esquema semántico de la distribución de hilos entre las dos funciones involucradas.*

A continuación se van a realizar varias deducciones teóricas que van a permitir acercar la solución al problema inherente de combinar las zonas ya paralelas, obteniendo una estimación de la posible mejora que puede conseguirse.

**Detalle de la combinación de paralelismos.**

Desde un punto de vista conceptual, se dispone de la función Interna, la cual emplea, en su ejecución serie, un tiempo por llamada en un orden de magnitud de $10^{-2}$ segundos. Dicho tiempo se magnifica debido al número de llamadas realizadas desde la función Externa.

Se debe tener en cuenta que la máxima aceleración a conseguir se corresponde, cómo máximo, de manera proporcional al número de hilos a utilizar, con la salvedad del orden de magnitud en la que se encuentra el problema a mejorar.

No es sencillo reducir el tiempo de una función cuyo orden de ejecución se encuentra en $10^{-2}$ segundos, en comparación con otra cuyo orden se encuentre en $10^{2}$ segundos, debido a las limitaciones inherentes a la capacidad de procesamiento de la CPU.

Continuando con el análisis teórico, se tiene, en la función Externa, un número total de llamadas igual a ((NCELDASX)·(NCELDASY)·(NCELDASZ))·(4)·(2), el cual, para el tamaño disponible, se corresponde con un total de 405224 llamadas a la función Interna.

Para dicha función Externa es sabido que requiere de un tiempo de procesamiento notable, debido precisamente al número de llamadas a la función Interna. Sin embargo, teniendo en cuenta este concepto, se puede realizar una planificación que marque la diferencia en el uso de recursos por parte de ambas funciones.

Esto es debido a que la ejecución actual de ambas funciones incluye la continua construcción y destrucción de la zona paralela que permite el procesamiento. Esta planificación afecta al rendimiento final del software, obligando a este, y subyacentemente al sistema operativo, que es la entidad que controla la asignación de hilos al software, a estar manipulando continuamente las asignaciones de procesador.

Sin embargo, si se atiende únicamente a la paralelización de la función Externa, se puede ver que se realiza una construcción y una destrucción de la zona paralela asociada, debido a que no se encuentra supeditada a la llamada por parte de ninguna otra función.

Teóricamente parece aceptable la idea de asignar una mayor cantidad de hilos a la función Externa, en la que se sabe que los hilos no van a ser destruidos, mientras se aplica un número menor de ellos a la función Interna, minimizando en cierta medida su tiempo de procesamiento, y teniendo en cuenta asimismo que la reducción del tiempo de ejecución es más complicada debido a su orden de magnitud.

Tras investigar el posible impacto en el rendimiento a obtener, se puede pasar a realizar los ajustes requeridos en el código para conseguir que ambas zonas paralelas colaboren en el procesamiento.

Como ya se ha visto, la zona paralela Externa hace llamadas a la función paralela Interna, ejecutando de esta manera lo que se conoce como una anidación de zonas paralelas. Esta anidación no se encuentra habilitada por OpenMP de forma predeterminada, por lo que se debe añadir la directiva que lo permita. Esta directiva es *omp_set_nested(1)*.

Es importante aplicar esta directiva para obtener resultados correctos ya que, de no hacerlo, la zona paralela interna sería ejecutada en serie, a pesar de que disponga, de forma correcta, de un *pragma* aplicado y de una serie de hilos asignados. Este comportamiento es debido a que la directiva es responsable del control del paralelismo a uno o más niveles.

Tras esta modificación, únicamente es necesario establecer, mediante las variables de control asociadas, el número de hilos a asignar a cada una de las zonas paralelas. Dispuestas dichas variables, se puede continuar con la realización de una tercera batería de pruebas que permita observar el comportamiento de la combinación de zonas paralelas según el número de hilos asignado a cada una de ellas.

### iii. Tercera salida de resultados: Externa e Interna.

En esta salida de resultados se van a analizar los resultados de combinar ambas paralelizaciones llevadas a cabo mediante el número de hilos disponible en cada sistema.

Debido a que el Ordenador Estudiante únicamente dispone de tres hilos útiles de ejecución, no es posible llevar a cabo ninguna combinación entre ambas paralelizaciones, por lo que en este caso dicho sistema no será utilizado con el propósito de experimentación.

De igual forma que en las salidas de resultados anteriores, se van a aportar, de nuevo, los resultados más característicos obtenidos previamente, pudiendo así visualizarlos de manera sencilla.

### a. Plataforma TFG.

| | Serie | OpenMP | | | OpenMP ($h_{ext}$ x $h_{int}$) | |
|---|---|---|---|---|---|---|
| | 1 opt (ref) | 2 | 4 | 8 | 2 x 4 | 4 x 2 |
| **Adicional** | 324,08 | 160,52 | 91,43 | 88,63 | | |
| **Speedup %** | -> | **2,02** | **3,54** | **3,66** | | |
| **Externa** | 3922,70 | 2545,57 | 1533,90 | 1399,99 | 1820,83 | 1755,18 |
| **Speedup %** | -> | **1,54** | **2,56** | **2,80** | **2,15** | **2,23** |
| **Page Faults** | 782,00 | 1806,00 | 1308,00 | 1840,00 | 43551,00 | 1687,00 |
| **Interna** | | 2762,79 | 2033,65 | 2216,22 | | |
| **Speedup %** | -> | **1,42** | **1,93** | **1,77** | | |
| **Page Faults** | | 1311,00 | 1828,00 | 1933254,00 | | |

*Tabla 14: Representación comparativa de los resultados obtenidos para las combinaciones en la Plataforma TFG.*

Para el primero de los sistemas, se puede observar que la aceleración máxima conseguida mediante la combinación no consigue superar la máxima paralelización conseguida hasta el momento, y que coincide con la paralelización de la función Externa, según se puede ver en la Tabla 14.

Así, priorizando la asignación de hilos a la función Interna, el rendimiento es mejor que el obtenido en la paralelización única de dicha función, aunque el número de fallos de paginación hace pensar que dicha combinación está provocando una fuga de rendimiento.

En efecto, invirtiendo la prioridad, se consigue mejorar el rendimiento conseguido, aunque no se superan las aceleraciones conseguidas mediante la paralelización única de la función Externa utilizando cuatro y ocho hilos.

Sin embargo, en este último caso de combinación sí se puede observar que el número de fallos de paginación ha vuelto a los niveles habituales del resto de experimentos, por lo que, de nuevo, este aumento parece indicar que se encuentra relacionado con la función Interna y el número de llamadas que esta debe atender para llevar a cabo el procesamiento completo del software.

### b. Servidor Beta.

| | Serie | OpenMP | | | OpenMP ($h_{ext}$ x $h_{int}$) | |
|---|---|---|---|---|---|---|
| | 1 opt (ref) | 2 | 4 | 8 | 2 x 4 | 4 x 2 |
| **Adicional** | 364,88 | 188,65 | 94,39 | 47,98 | | |
| **Speedup** | -> | **1,93** | **3,82** | **7,21** | | |
| **Externa** | 9980,81 | 7513,84 | 5189,11 | 4155,77 | 6183,09 | 5109,35 |
| **Speedup** | -> | **1,33** | **1,92** | **2,40** | **1,61** | **1,95** |
| **Interna** | | 7232,86 | 6184,05 | 5577,31 | | |
| **Speedup** | -> | **1,33** | **1,61** | **1,79** | | |

*Tabla 15: Representación comparativa de los resultados obtenidos para las combinaciones en el Servidor Beta.*

Para el Servidor Beta, el rendimiento conseguido también se encuentra en un punto intermedio entre ambas paralelizaciones independientes, en el mejor de los casos, según apunta la Tabla 15.

Utilizando una mayor cantidad de hilos en la función Interna, el rendimiento es equivalente a la asignación de cuatro hilos únicamente en dicha función, por lo que la utilización de dos hilos como grupo de trabajo externo para los que, a su vez, anidan cuatro hilos más, parece perjudicar el rendimiento final.

Algo similar puede verse en el caso opuesto: asignando cuatro hilos a la función Externa el rendimiento mejora, pero parece que la utilización de dos hilos de forma anidada para el procesamiento de la función Interna

penaliza el rendimiento final, obteniendo un rendimiento similar a la utilización de cuatro hilos únicamente en dicha función Externa.

Analizados los resultados, estas combinaciones no han permitido mejorar el mejor de los rendimientos obtenido hasta el momento mediante la paralelización de la función Externa.

En la Figura 14 puede observarse, de manera gráfica, los resultados obtenidos.

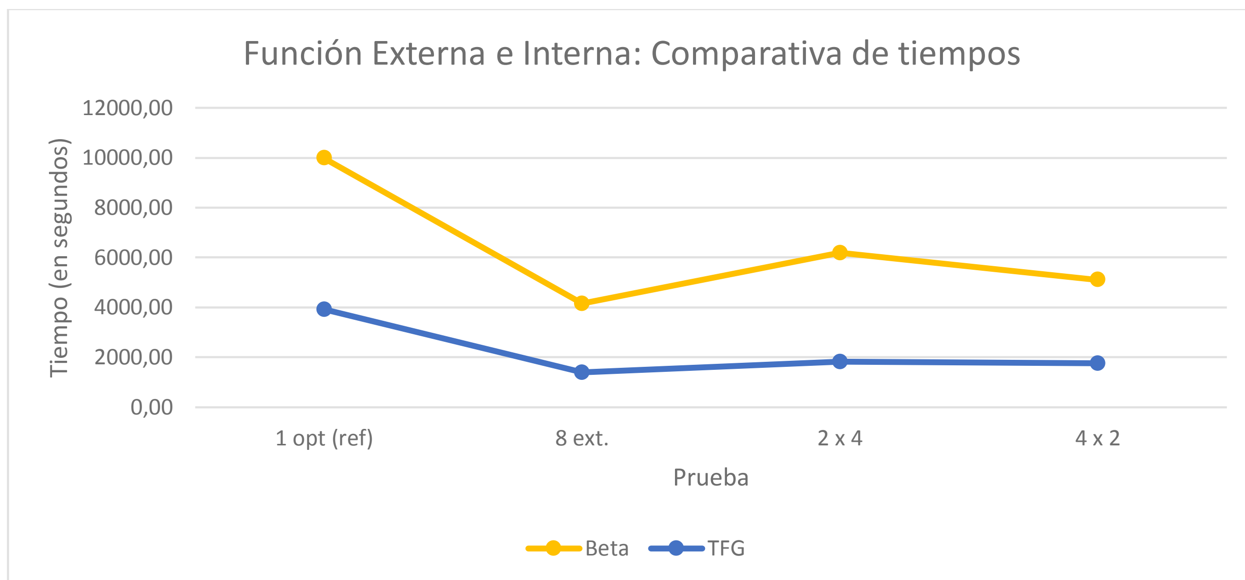


*Figura 14: Gráfico ilustrativo de los tiempos obtenidos para las funciones Externa e Interna.*

## 4.6. Paralelización unificada de las funciones Externa e Interna.

Tras los últimos resultados obtenidos, se llega al último incremento a desarrollar, consistente en la unificación de ambas regiones paralelas en una sola que permita eliminar al máximo las pérdidas superfluas de tiempo y mejorar correspondiente el rendimiento.

Mediante este nuevo mecanismo de combinación a través de la unificación se puede efectuar una simplificación adicional, especialmente en relación a la creación y destrucción de hilos anteriormente comentada. A continuación se van a detallar todos los cambios a realizar para lograr la versión final del software. Esta unificación, por lo tanto, será la forma definitiva del código a través del modelo OpenMP.

**Estrategia de optimización.**

Se ha documentado que la función Externa hace dos llamadas a la función Interna por cada iteración a procesar, requiriendo para ello de dos cambios de contexto y la construcción y posterior destrucción de la región paralela que procese dicha función Interna. Ambas llamadas a la función Interna representan la misma funcionalidad, con la única diferencia de que en una de ellas se realiza el procesamiento del Defecto A y, en la otra, del Defecto B.

En términos prácticos, la diferencia se encuentra en la realización de las operaciones parciales, bien sean sumas o productos, que se deben efectuar sobre cada defecto considerado en cada momento, pero incluso las posiciones de acceso a los vectores asociados se comparten. Este detalle cambia el cuarto bloque de procesamiento por completo.

Por todo ello, existen dos puntos de mejora teórica que se van a implementar sobre el código ya paralelo, utilizando como base la zona paralela correspondiente a la función Externa.

En primer lugar, se va a reducir de dos a una las llamadas producidas a la función Interna, para lo cual lo que se va a conseguir es procesar, en cada iteración de la función Externa, ambos defectos en una sola llamada a la función Interna. Esta modificación en ambas funciones permitirá obtener una aceleración teórica máxima de dos sobre los tiempos actuales. En el Código 15 puede observarse esta modificación de la función Interna.

```
EnElA0=EnElA0 + ( cstcbrp[indicecb+0]*cstdarp[indiced+0] +
cstcbrp[indicecb+1]*cstdarp[indiced+1] +
cstcbrp[indicecb+2]*cstdarp[indiced+2])/Y;

EnElA2=EnElA2 + ( cstcbrp[indicecb+3]*cstdarp[indiced+3] +
cstcbrp[indicecb+4]*cstdarp[indiced+4] +
cstcbrp[indicecb+5]*cstdarp[indiced+5])*4.0/G;

EnElA1=EnElA1 + -1.0*( cstcbrp[indicecb+0]*(cstdarp[indiced+1] +
cstdarp[indiced+2]) + cstcbrp[indicecb+1]*(cstdarp[indiced+0] +
cstdarp[indiced+2]) + cstcbrp[indicecb+2]*(cstdarp[indiced+0] +
cstdarp[indiced+1]) )*nu/Y;

EnElB0=EnElB0 + ( cstcbrp[indicecb+0]*cstdbrp[indiced+0] +
cstcbrp[indicecb+1]*cstdbrp[indiced+1] +
cstcbrp[indicecb+2]*cstdbrp[indiced+2])/Y;

EnElB2=EnElB2 + ( cstcbrp[indicecb+3]*cstdbrp[indiced+3] +
cstcbrp[indicecb+4]*cstdbrp[indiced+4] +
cstcbrp[indicecb+5]*cstdbrp[indiced+5])*4.0/G;

EnElB1=EnElB1 + -1.0*( cstcbrp[indicecb+0]*(cstdbrp[indiced+1] +
cstdbrp[indiced+2]) + cstcbrp[indicecb+1]*(cstdbrp[indiced+0] +
cstdbrp[indiced+2]) + cstcbrp[indicecb+2]*(cstdbrp[indiced+0] +
cstdbrp[indiced+1]) )*nu/Y;
```

*Código 15: Procesamiento unificado de ambas celdas Defecto.*

Por otro lado, de manera conjunta a este cambio, se va a realizar la integración del código de la función Interna en la región paralela Externa, eliminando así los cambios de contexto implicados en este proceso.

Ambos códigos, sin embargo, continúan siendo fácilmente diferenciables debido a que se pueden separar conceptualmente mediante los mecanismos que el lenguaje C provee (el uso de las llaves). Como únicamente

es el cuarto bloque el que realiza llamadas a la función Interna, este acoplamiento no va a resultar en una posterior duplicación de código, la cual podría, por otro lado, dificultar su mantenibilidad y su portabilidad.

Los beneficios, sin embargo, se notan rápidamente: el procesamiento del bloque ahora es llevado a cabo de forma más limpia y eficiente, al haber eliminado las sobrecargas inherentes a la abstracción de código mediante la función Interna.

Esta ganancia de rendimiento debe notarse más aún por la minimización implícita en el esfuerzo que el sistema operativo debe realizar para obtener los mismos resultados, consiguiendo así un uso más responsable y eficiente de los recursos.

Las implementaciones descritas son sencillas de incorporar, debido a uno de los desarrollos incrementales previamente implementados, el correspondiente a la paralelización de la función Externa, que permite ahora utilizar su región paralela (en la que se hizo una reestructuración del código) como base para completar el bloque de manera unificada a través de la función Interna, ya paralelizada también.

En términos prácticos, los cambios a realizar implican sustituir las llamadas a la función Interna por su código correspondiente y adaptar dicho código para procesar ambos defectos consecutivamente.

La totalidad del resto del código se encuentra ya en un estado final, debido a que ya se unificó la salida de los resultados de los dos defectos, así como el ajuste de todas las variables que forman parte de dicho proceso.

Por lo tanto, tras esta restructuración, es el momento de establecer la prueba del código en los sistemas disponibles, observando si se logra una mejora de rendimiento.

Para la obtención de estos resultados no va a ser necesario realizar pruebas con un número de núcleos variable, ya que en este punto la referencia adecuada es el mejor rendimiento conseguido hasta el momento, correspondiente con la paralelización de la función Externa mediante la utilización del máximo número de hilos disponible.

### iv. Cuarta salida de resultados: Unificada.

La última de las salidas de resultados relacionada con OpenMP comprende el análisis y comparación de rendimientos entre la función Externa unificada y el resto de paralelizaciones llevadas a cabo previamente, tomando como referencia principal el mejor de los tiempos conseguidos hasta el momento para cada sistema en experimentación.

A continuación se pasa a mostrar estos resultados, de nuevo, divididos según el sistema de experimentación.

#### a. Ordenador Estudiante.

| | Serie | OpenMP | |
|---|---|---|---|
| | **1 opt (ref)** | **3** | **3 unif.** |
| **Adicional** | 424,54 | 147,22 | |
| **Speedup** | -> | **2,88** | |
| **Externa** | 4502,82 | 2709,58 | 2138,27 |
| **Speedup** | -> | **1,66** | **2,11** |
| **Page Faults** | 1275,00 | 2709,58 | 101805,00 |
| **Interna** | | 3259,86 | |
| **Speedup** | -> | **1,38** | |
| **Page Faults** | | 1822,00 | |

*Tabla 16: Representación comparativa de los resultados obtenidos mediante la unificación en el Ordenador Estudiante.*

Nuevamente este sistema puede ofrecer resultados, aportando un punto de vista adicional. Mediante la paralelización unificada del cuarto bloque se ha obtenido el mejor de los tiempos que hasta el momento se disponía, logrando una aceleración de 2,11, según se puede ver en la Tabla 16.

Se debe tener en cuenta que el número de fallos de página ha aumentado con respecto a los valores obtenidos en las iteraciones anteriores, lo que puede indicar, de nuevo, la saturación de los recursos disponibles.

Así, mediante esta última iteración, se ha conseguido mejorar finalmente el cuarto bloque del sistema desde los 10167,27 segundos iniciales hasta los 2138,27 segundos, obteniendo una aceleración final correspondiente a 4,75. Este factor, que sobrepasa el número de hilos asignados, tiene sentido al haber restructurado el código original y al haberlo sometido a la optimización del compilador.

### b. Plataforma TFG.

| | Serie | OpenMP | | |
|---|---|---|---|---|
| | 1 opt (ref) | 8 | 4 x 2 | 8 unif. |
| **Adicional** | 324,08 | 88,63 | | |
| **Speedup %** | -> | **3,66** | | |
| **Externa** | 3922,70 | 1399,99 | 1755,18 | 1372,82 |
| **Speedup %** | -> | **2,80** | **2,23** | **2,86** |
| **Page Faults** | 782,00 | 1840,00 | 1687,00 | 1262507,00 |
| **Interna** | | 2216,22 | | |
| **Speedup %** | -> | **1,77** | | |
| **Page Faults** | | 1933254,00 | | |

*Tabla 17: Representación comparativa de los resultados obtenidos mediante la unificación en la Plataforma TFG.*

Revisando la Tabla 7 se puede ver que el procesamiento del software en la Plataforma TFG ha conseguido el mejor de los rendimientos disponibles hasta el momento, llegando a obtener una aceleración que, si bien se mantiene cercana a la correspondiente a la paralelización de la función Externa, ha sido superada, llegando a una cota aún más cercana a tres, la cual por otro lado sigue manteniéndose lejos del número de hilos utilizados para obtenerla.

Se puede observar, de nuevo mediante los datos acerca de los fallos de paginación, que mediante este incremento se ha obtenido una gran cantidad de ellos, lo que puede indicar que el procesamiento de este código se encuentra limitado por las capacidades del sistema operativo, al involucrar una gran cantidad de datos, lo que puede haber saturado los recursos de memoria.

A través de este sistema se ha conseguido reducir el tiempo de ejecución de esta función desde los 11387,37 segundos inicialmente obtenidos hasta los 1372,82 segundos, obteniendo una aceleración efectiva de 8,29.

### c. Servidor Beta.

| | Serie | OpenMP | | |
|---|---|---|---|---|
| | 1 opt (ref) | 8 | 4 x 2 | 8 unif. |
| **Adicional** | 364,88 | 47,98 | | |
| **Speedup** | -> | **7,21** | | |
| **Externa** | 9980,81 | 4155,77 | 5109,35 | 3746,12 |
| **Speedup** | -> | **2,40** | **1,95** | **2,66** |
| **Interna** | | 5577,31 | | |
| **Speedup** | -> | **1,79** | | |

*Tabla 18: Representación comparativa de los resultados obtenidos mediante la unificación en el Servidor Beta.*

Como puede observarse en la Tabla 18, el rendimiento conseguido mediante la ejecución del software de manera unificada es superior a cualquiera de los resultados previamente obtenido, excediendo incluso el mejor resultado previo, el referido a la paralelización de la función Externa.

A pesar de todo ello, la aceleración conseguida continúa siendo inferior a la esperada, en relación al número de hilos asignado, si bien el tiempo final de ejecución del cuarto bloque ha disminuido desde los 15145,25 segundos conseguidos por el código serie original, hasta los 3746,12 segundos, obteniendo así una aceleración final de 4,04. Un valor que, si bien es mejor, sigue lejano al número de hilos asignados al trabajo.

De nuevo, mediante una representación gráfica, realizada en la Figura 15, puede observarse de manera más comparativa los resultados conseguidos a través de los tres sistemas.

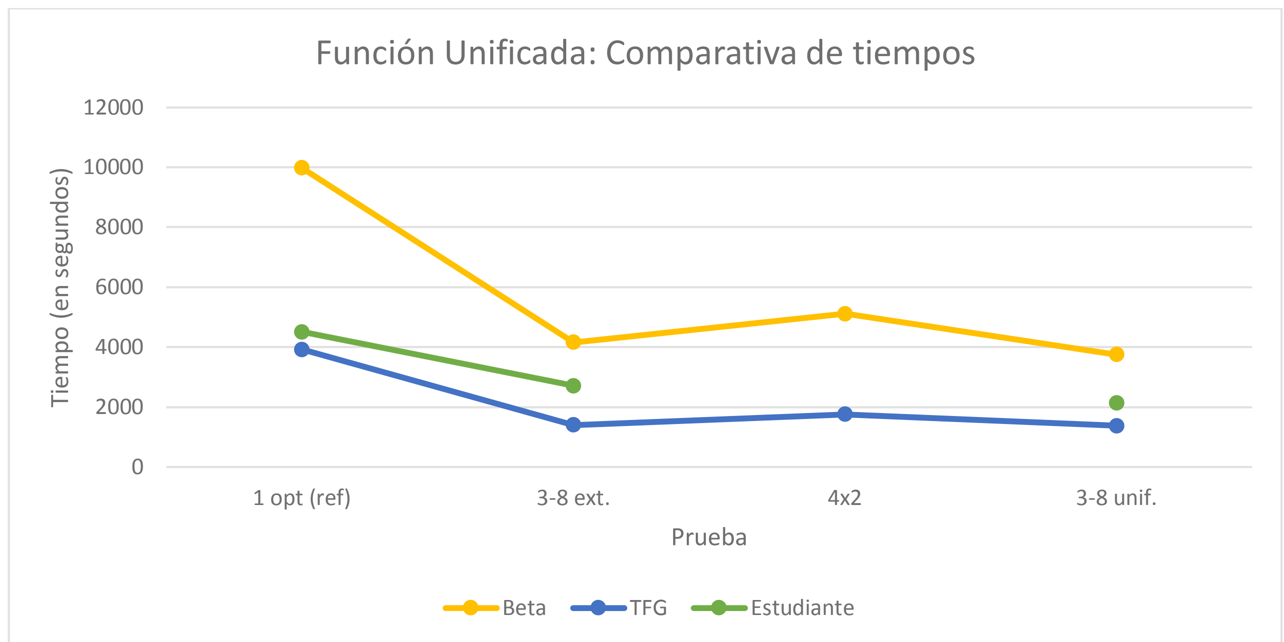


*Figura 15: Gráfico ilustrativo de los tiempos obtenidos para la función Unificada.*

## 4.7. Conclusiones establecidas acerca del modelo.

Tras el desarrollo realizado, se ha logrado implementar un código estable, limpio y compacto que, si bien no es tan modular como el código inicial, debido a que una de las funciones se ha unificado en la función principal, ha logrado un tiempo de ejecución muy eficiente, especialmente en comparación con el tiempo original en serie.

Con este último incremento se llega al final del proceso de desarrollo de código paralelo mediante el modelo que implementa OpenMP. El rendimiento se ha visto mejorado conforme ha avanzado el desarrollo, observando nuevas vías de organización y combinación entre las diferentes regiones paralelas desarrolladas.

Se concluye que la mejor opción se encuentra representada por el último código obtenido, en el que las funciones Interna y Externa han sido unificadas. Se puede decir, en este momento, que dicho código es la versión optimizada del software mediante el modelo OpenMP.

Haciendo balance, el código final obtenido es aquel en el que ambas regiones paralelas han sido unificadas, eliminando la función Interna, debido a que solo el cuarto bloque dependía de ella y, sin embargo, causaba una gran pérdida de rendimiento en el tiempo de ejecución por los cambios de contexto asociados al proceso. En primera instancia, sin embargo, su independencia ha sido aprovechada para poder establecer una primera paralelización del cuarto bloque.

Una característica muy importante de la paralelización basada en el modelo *fork-join* a través del uso de la CPU, y que no se ha destacado durante el proceso, es que el modelo que implementa OpenMP ha permitido desacoplar el tamaño de la muestra del tipo de implementación efectuada.

Aunque el tamaño de la muestra no es cerrado y de hecho será variado en el futuro, sí que se fijó para el desarrollo del trabajo, por lo que se ha intentado siempre mantener el diseño del software de manera parametrizada. Prueba del funcionamiento independiente del tamaño de la muestra es que se han realizado pruebas externas con celdas cuyas dimensiones se acercaban a las cien unidades por dimensión y su procesamiento se ha continuado realizando de forma eficaz.

La solución final implementada se ha podido lograr gracias a varios factores:

- Por un lado, debido al procesamiento paralelo de los bucles mediante *pragmas* que pone a disposición el modelo OpenMP, en los que es posible escoger el esquema de desglose a utilizar, según el tipo de problema del que se trate. Como se sabe, el esquema elegido en todos los casos se corresponde con aquel que dota a cada hilo de una parte proporcional de la carga total a procesar.
- Por otro lado, el carácter modular e independiente que se ha conseguido durante la paralelización de la función Externa ha sido determinante para la obtención de la versión final del software, al disponer de una zona paralela desacoplada del número de hilos asignados.

Para terminar, el propio modelo de OpenMP, cuyo procesamiento se basa en el uso de la potencia, tecnología y número de núcleos de CPUs y memorias RAM convencionales, permiten liberar al desarrollador de muchos de los detalles de la implementación del modelo, minimizando y ajustando en gran medida las limitaciones debidas a la plataforma hardware que se utilice. Esto ha sido de especial utilidad para poder generar un código tan robusto como el conseguido en el desarrollo realizado.

**Conclusiones principales tras la aplicación de este modelo paralelo:**

- OpenMP ha permitido mejorar notablemente el tiempo de ejecución para cada uno de los sistemas.
- El desarrollo de soluciones ha contribuido a la creación de un software mejor estructurado, lo que se ha reflejado en un aumentado adicional en su eficiencia.
- La solución final utilizando este modelo permite al software continuar siendo escalable y apenas ha generado dependencia en relación al hardware huésped.

# Capítulo 5
## Paralelización mediante CUDA

A continuación se va a explorar el modelo basado en GPU, implementado por la arquitectura CUDA.

Durante el capítulo se realizará un acercamiento a sus características, investigando sobre las diversas soluciones posibles.

El avance incremental dará lugar a implementaciones más complejas y refinadas, consiguiendo un mejor aprovechamiento del dispositivo hardware implicado.

## 5.1. Introducción. Limitaciones técnicas del modelo basado en GPU.

CUDA es una arquitectura de cálculo paralelo que permite aprovechar las características de los dispositivos GPU fabricados por la compañía desarrolladora de la arquitectura, NVidia, al contar con gran cantidad de hilos.

Esencialmente esta arquitectura se diferencia del modelo propuesto por OpenMP en que esta primera requiere que el desarrollador elabore cambios y adaptaciones sustanciales en el código, consiguiendo así que este pueda ser ejecutado en la GPU, aprovechando así sus capacidades.

A pesar de que, adaptando los términos, ejecutar un núcleo (*kernel*) CUDA equivaldría a la llamada a una función con una región paralela en OpenMP, lo cierto es que el procesamiento interno es radicalmente distinto, debido a que el espacio de trabajo se lleva a cabo en la GPU y no en el sistema de memoria habitual del sistema huésped.

Debido a los costes en la construcción, y a la práctica imposibilidad de ampliar los recursos disponibles en la GPU, esta arquitectura puede sufrir de insuficientes recursos, especialmente en términos de memoria, impidiendo así el procesamiento de ciertos tipos de problemas paralelizables. A diferencia del procesamiento que se puede conseguir mediante OpenMP, la utilización de esta arquitectura se dedica a problemas de propósito muy específico, en los que el procesamiento conseguido es, generalmente, muy eficiente.

Para el sistema disponible sobre el que se van a realizar las pruebas y análisis de rendimiento, la denominada Plataforma TFG, se encuentra montado el modelo de GPU NVidia GeForce GTX 650 Ti, cuyas características se listan en la Figura 16. En la Figura 17 puede observarse la distribución y disponibilidad de los componentes que componen la arquitectura.

Gracias al acceso remoto que se hace de este sistema, el aprovechamiento del dispositivo es cercano al 100%, al no requerir la creación de un entorno gráfico que de soporte al desarrollo y ejecución de los núcleos de desarrollo.

- Modelo: NVidia GeForce GTX 650 Ti
- Arquitectura: Kepler
- Versión del Driver CUDA: 7.5
- Versión de CUDA ejecutable: 3.0
- Memoria global: 2047MBytes (2146762752 bytes)
- 4 multiprocesadores, 192 Núcleos CUDA/MP: 768 Núcleos CUDA
- Velocidad de reloj de la GPU: 1032MHz (1.03GHz)
- Velocidad de reloj de la memoria: 2700Mhz
- Total de memoria constante: 65536 bytes
- Total de memoria compartida por bloque: 49152 bytes
- Número total de registros por bloque: 65536
- Tamaño del warp: 32
- Número máximo de hilos por multiprocesador: 2048
- Número máximo de hilos por bloque: 1024
- Dimensión máxima del bloque de hilos (X,Y,Z): (1024, 1024, 64)
- Dimensión máxima de la red (X,Y,Z): (2147483647, 65535, 65535)

*Figura 16: Características principales del dispositivo gráfico disponible.*

Atendiendo a los parámetros técnicos del dispositivo, especialmente en referencia a su arquitectura concreta, Kepler, se dispone de un conjunto de propiedades que permiten desarrollar soluciones sobre la GPU, aunque también implica una serie de limitaciones, las cuales, conforme la investigación y el desarrollo de nuevos modelos hardware son llevados a cabo, van siendo superadas por el fabricante. Estas limitaciones inherentes al hardware, superadas en versiones más actuales de la arquitectura, no pueden ser aprovechadas por el modelo disponible para experimentación.

A continuación se van a pasar a describir las propiedades que la versión disponible de la arquitectura pone al servicio del desarrollador.

**Propiedades de la arquitectura Kepler.**

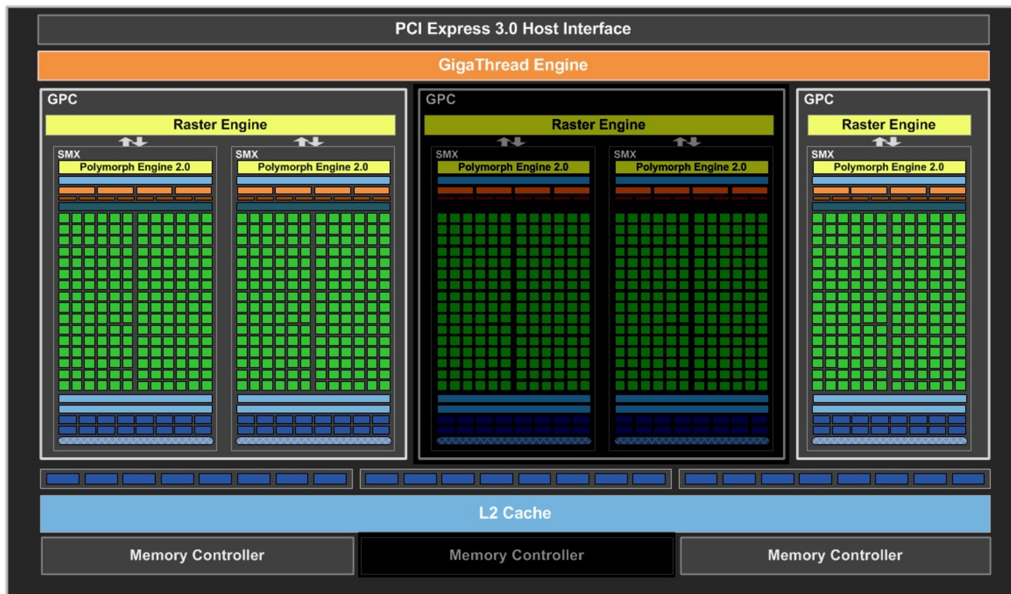


*Figura 17: Composición completa de Kepler. El modelo disponible, sin embargo, no dispone de los componentes oscurecidos.*

**a) Generación de hilos de procesamiento.**

El número de hilos a utilizar en cada ejecución concreta es determinado de manera manual por el desarrollador (o mediante la asistencia de documentación de referencia), generalmente en tiempo de compilación, permitiendo de esta forma que el compilador CUDA realice internamente la planificación más conveniente.

Este número de hilos, a diferencia del modelo de OpenMP, se define mediante el establecimiento de una red de hasta tres dimensiones (según el desarrollador requiera) de bloques e hilos, formando una cuadricula (*grid*). La red contiene tantos hilos como se defina en la estructura de red de las dimensiones utilizadas, teniendo como límite, en cualquier caso, el número máximo impuesto por el modelo concreto de GPU.

De forma práctica, el caso habitual de uso incluye la utilización de una única dimensión, la relacionada con el eje X, la cual, asimismo, lleva asociada un número notablemente más grande de hilos posibles a definir por parte de la arquitectura, según se puede verificar en la Figura 16.

Es posible establecer una geometría aprovechando las características que la arquitectura permite, dotando a la red de una estructura lineal, rectangular u ortoédrica, dependiendo del número de dimensiones utilizadas. Generalmente, y a excepción de problemas que realizan un uso eficiente de la geometría, suele utilizarse únicamente una sola dimensión, ya que el empleo de dimensiones adicionales habitualmente dificulta las operaciones sobre los datos y la comunicación entre los hilos.

La red de hilos definida se subdivide, asimismo, en porciones, denominadas como bloques. El concepto de bloque es una abstracción que agrupa una cantidad de hilos explícitamente definida por el desarrollador, agrupándolos para trabajar de manera cooperativa.

Por lo tanto, para llevar a cabo la creación efectiva de los hilos requeridos para un procesamiento concreto, es necesario definir manualmente:

- el número de hilos por bloque (también denominado como tamaño del bloque), y
- el número de bloques en la red (también denominado como tamaño de la red).

De esta manera, el número efectivo de hilos de los que dispondrá el núcleo creado se corresponde con el producto de los números anteriormente definidos. La agrupación de los hilos en bloques permite el acceso compartido a cada uno de estos hilos que forma parte del bloque, a un espacio de memoria intermedia adicional, varios órdenes de magnitud más rápida que la memoria global disponible para toda la red de hilos, denominada como memoria compartida (*shared*).

### b) Operativa en la elección de hilos y el acceso a memoria. Concepto de warp.

Escoger un número de bloque adecuado a cada problema a resolver tiene su importancia en la posibilidad de realizar operaciones de manera cooperativa haciendo uso de la memoria compartida, debido a que su latencia es muy pequeña, lo que en términos de rendimiento permite incrementar la eficiencia en el procesamiento de código.

Es posible escoger, como ya se ha mencionado, cualquier número como cantidad de hilos para formar un bloque, dentro de los límites de la arquitectura (que en este caso es de 1024 hilos por bloque), siendo lo habitual escoger un número múltiplo de 32.

Este número tiene su importancia en la manera en que CUDA realiza las lecturas y escrituras que tienen lugar durante la ejecución del núcleo, las cuales se realizan en porciones de, precisamente, 32 elementos. Esta unidad de acceso y escritura se denomina *warp*.

Existe una notable diferencia entre el modelo de acceso y escritura implementado por CUDA y el habitual de la CPU, ya que mientras que el primero accede en porciones de 32 elementos (aunque se requieran menos de este número), el último lo hace en porciones de una unidad. El empleo correcto del acceso y escritura a través de *warps* da lugar a un rendimiento mejorado en referencia a cualquier otro tipo de acceso posible en la GPU.

La estrategia consiste en escoger un número de hilos efectivo del que pueda beneficiarse el software y calcular, en base a estas condiciones, el tamaño del bloque y el tamaño de la red más adecuados para procesar de manera paralela el código requerido.

Una limitación adicional a tener en cuenta en el desarrollo de software para su procesamiento en CUDA es la consideración del número de multiprocesadores que van a ser empleados. Este número, si bien en términos prácticos no implica una pérdida real de rendimiento, debido a que es escogido automáticamente por la arquitectura CUDA en tiempo de compilación, se asigna teniendo en cuenta la geometría de la red escogida por el desarrollador.

Es útil manipular dicha geometría de forma que CUDA establezca una mayor cantidad de multiprocesadores. La razón de ello se encuentra en que la memoria compartida está asociada a los multiprocesadores disponibles en la arquitectura.

La asignación de memoria compartida al cómputo general del núcleo CUDA creado depende en última instancia de la geometría de red escogida y del número de hilos que sean requeridos. Así, el tamaño máximo de la memoria compartida, para esta arquitectura, se encuentra en los 48KB, un tamaño que no es posible ampliar en ningún momento, y que sin embargo sí disminuye si deben hacer uso de la memoria un número más amplio de hilos.

Es importante tener en cuenta este detalle ya que es posible que no exista cantidad suficiente de memoria para suplir el procesamiento paralelo completo del software desarrollado.

### c) Coalescencia.

El esquema de acceso eficiente a memoria que proporciona la arquitectura CUDA es el realizado mediante *warps*. Para que estos accesos se efectúen eficientemente, se debe poder aprovechar la lectura/escritura de cada uno de los 32 elementos implicados en ellos.

Esto se traduce en que cada lectura/escritura de 32 elementos ha de ser establecida en posiciones consecutivas desde/hasta memoria, bien se trate de memoria de acceso global o compartido, relacionando simultáneamente cada uno de esos 32 elementos a 32 hilos en el mismo momento temporal.

Aunque teóricamente no es imprescindible aprovechar el acceso que utiliza la GPU en forma de *warps*, es altamente recomendado puesto que la arquitectura los lleva a cabo siempre de esta manera, accediendo consecuentemente en un solo acceso a 32 elementos consecutivos, aunque solo se requiera uno de esos 32 elementos.

Por ello, habitualmente es necesario aprovechar esta propiedad y establecer los denominados accesos coalescentes. El significado de este concepto consiste en que cada hilo acceda/escriba a/en una posición consecutiva de memoria, de manera que el núcleo CUDA pueda realizar una sola lectura/escritura de 32 elementos válidos en vez de 32 lecturas de $32 \cdot 31$ elementos inválidos. Establecer un acceso no coalescente penaliza el rendimiento interno de CUDA.

Este tipo de acceso coalescente requiere que los datos a leer/escribir se encuentren alineados, es decir, dispuestos de la manera consecutiva en que después son accedidos. Para lograrlo, o bien los datos ya siguen la estructura requerida, o bien se debe realizar una adaptación explicita que permita el posterior acceso coalescente por parte del dispositivo GPU, con la consiguiente sobrecarga implicada en el tiempo requerido para ello.

**d) Propiedades adicionales inherentes a la versión de la arquitectura.**

Para terminar, existen dos propiedades adicionales. Hasta la arquitectura Kepler, es decir, para la arquitectura que va a utilizarse, es requisito imprescindible que el tiempo total del núcleo desarrollado no supere los cinco segundos de procesamiento.

Esta limitación se impone con la finalidad de proteger la GPU, consiguiendo el reinicio de su estado en caso de que el comportamiento no sea el esperado, e impidiendo de esta manera que el desarrollador deba apagar el sistema en caliente o que el dispositivo se dañe.

Sin embargo, este mecanismo de seguridad puede convertirse en una limitación durante el desarrollo de soluciones, ya que el procesamiento de grandes cantidades de datos, en especial si estos no realizan accesos coalescentes a memoria, puede requerir más tiempo del disponible. Es por ello que, en versiones posteriores de la arquitectura, existe una baliza que elimina esta restricción.

La última de las propiedades es la posibilidad de lanzar varios núcleos de manera simultánea a la misma GPU, uno a continuación de otro, según indica la documentación. Desde la versión de la arquitectura utilizada, Kepler, esto es posible. Sin embargo, se debe tener en cuenta que, a pesar de poder llevar a cabo el lanzamiento simultáneo de varios núcleos, los recursos disponibles en la GPU no se multiplican, por lo que esto puede resultar en una posibilidad muy limitada si el núcleo, o la combinación de núcleos que se pretenden lanzar a la GPU, utilizan una gran cantidad de recursos.

Cabe mencionar que, si bien durante la redacción de esta memoria se conoce en gran medida la arquitectura CUDA, durante el desarrollo de las soluciones se ha ido profundizando en el conocimiento acerca de ella, obteniendo mejores resultados, más avanzados y complejos.

Sin embargo, en el transcurso de esta investigación se han realizado, asimismo, una gran cantidad de pruebas fallidas, cuya justificación se encuentra basada en la propia arquitectura. Gracias a estos errores, a las pruebas realizadas y a la superación continua de las limitaciones debidas al hardware, se ha podido aprender acerca de la arquitectura, pudiendo en este momento establecer un marco introductorio.

A continuación, mediante los siguientes apartados se van a mencionar y describir los desarrollos implementados en el contexto del cuarto bloque del software proveído, ya que es la porción de código que mayor tiempo de ejecución requiere.

Tratando de abarcar en cada incremento una mayor cantidad de código, los niveles de desarrollo de las soluciones comprenden la paralelización de la función Interna, de la función Externa unificada, y de paralelizaciones intermedias entre la función Interna y la función Externa completa, estableciendo el nivel de paralelización en el bucle anidado sobre el que implementar el desarrollo.

**Objetivos principales a desarrollar mediante este modelo:**

- Estudiar la arquitectura CUDA con el fin de aplicar y aprovechar sus características de propósito específico basadas en GPU.
- Desarrollar soluciones software a varios niveles de anidamiento, tratando de cubrir las capacidades del dispositivo gráfico disponible.
- Caracterizar las mejores soluciones desarrolladas y establecer una versión final para el modelo implementado.

## 5.2. Paralelización de la función Interna.

De manera similar al proceso seguido en la implementación del modelo OpenMP, el primer paso consiste en desarrollar un código que permita paralelizar la función Interna.

El enfoque inicial a llevar a cabo consiste en definir una red de hilos equivalente al número total de iteraciones requeridas para procesar cada llamada a la función.

Teóricamente, conseguir paralelizar un bucle anidado de ((2·NCELDASX)·(2·NCELDASY)·(2·NCELDASZ))·2 iteraciones en una única llamada a un núcleo, según se describe en el Código 16, debería suponer una reducción de tiempo proporcional a dicho número, salvando las sobrecargas que influyen en el proceso de adaptación del código a la arquitectura CUDA.

```
for (kx=0; kx< 2*NCELDASX; kx++)
{
  for (ky=0; ky< 2*NCELDASY; ky++)
  {
      for (kz=0; kz< 2*NCELDASZ; kz++)
      {
          for (ksl=0; ksl< 2; ksl++)
          {
```

*Código 16: Anidación de bucles dispuesta en la función Interna.*

Sin embargo, aunque hasta el momento la solución parece factible de desarrollar, en el proceso se encontraron varios problemas.

**Estrategia de optimización y problemas detectados.**

Debido a la estructura modular inherente al concepto de función, este primer desarrollo fue iniciado mediante la adaptación del código en un núcleo CUDA que posteriormente fuera llamado desde la función Externa, utilizándolo así como reemplazo de la función Interna.

Sin efectuar grandes cambios en la estructura interna del código de la función, el núcleo realizaría las mismas operaciones que la función original, con la salvedad de que el procesamiento se haría de forma paralela haciendo uso de las capacidades de CUDA en el espacio de trabajo de la GPU.

**Reducción de los resultados parciales generados en el código paralelo.**

El primer problema encontrado en la adaptación del código tiene relación con la generación de un proceso de reducción análogo al seguido en el código desarrollado en OpenMP, para el cual, en este último, la reducción se efectúa mediante un *pragma* incluido en dicho modelo, que realiza las operaciones requeridas de forma eficiente y transparente al desarrollador.

Para obtener un procesamiento similar en CUDA, la solución pasa por establecer en memoria compartida cada uno de los resultados parciales obtenido por los hilos. Dado que todos los hilos ejecutan las mismas operaciones y que estas son conmutativas, ya que pertenecen a la misma llamada de la función Interna, se puede realizar la reducción utilizando el nivel de memoria compartida.

Así, recursivamente, utilizando la mayor cantidad de hilos posibles en cada iteración, se obtiene la reducción buscada. Este proceso puede verse en el Código 17.

La estrategia a seguir consiste en realizar la suma entre los hilos pares y sus contiguos impares y almacenar el resultado en estos primeros, utilizándolos como pivotes. Este procedimiento de selección y suma de resultados se establece a través de una bifurcación, en la cual se comprueba si el hilo que ejecuta la porción del código es divisible exactamente por dos, es decir, si es par.

Repitiendo el proceso, siguiendo una estructura de árbol (controlada por la variable *s* en el Código 17), el resultado se encuentra finalmente en el hilo 0 de cada bloque considerado.

```
//Hacemos la reducción al hilo 0 de cada bloque
for(unsigned int s=1; s <blockDim.x; s *= 2)
{
    if(tid % (2*s) == 0)
    {
        Elastica[(tid*3)+0] += Elastica[(tid+s)*3+0];
        Elastica[(tid*3)+1] += Elastica[(tid+s)*3+1];
        Elastica[(tid*3)+2] += Elastica[(tid+s)*3+2];
        }
    __syncthreads();
    }
__syncthreads();
```

*Código 17: Reducción implementada.*

Para terminar el proceso de reducción, se deben sumar los resultados parciales de los hilos 0 de todos los bloques implicados en el procesamiento. Esta última suma se ejecuta en la CPU, por lo que es necesario realizar la escritura de los resultados reducidos en un vector de salida, que seguidamente es sumado para obtener así la reducción final que se obtendría, de manera equivalente, mediante la función Interna.

**Tiempo de ejecución.**

Solucionado el problema de la reducción de los resultados parciales, pudo ejecutarse el núcleo.

A través de dicha ejecución se pudieron observar problemas de entrada y salida en la lectura y escritura de los elementos implicados, es decir, en las celdas Defecto y Background a procesar. Este inconveniente se da en consecuencia a la realización del mismo tipo de acceso que el efectuado para la CPU, es decir, un acceso no alineado y por lo tanto, no coalescente.

Mediante el acceso convencional no coalescente de lectura y escritura se penalizó el tiempo de ejecución, ya que se debe tener en cuenta que cada una de las lecturas a las celdas requeridas, de tamaño del orden de $10^6$ datos, ejecuta activamente una lectura de 32 datos de los cuales se descartan 31.

A pesar de ello, la ejecución del núcleo adaptado sí conseguía finalizar, lo que produjo la no detección en ese momento de este fallo en la construcción del código, aunque sí se vio que los tiempos producidos, en torno a las ocho horas, no se consideraron eficientes, lo que permitió continuar con el desarrollo, con el fin de mejorarlos.

La causa de establecer el siguiente incremento, sin embargo, se debe a la conclusión de que el núcleo desarrollado no hacía un uso eficiente de las capacidades paralelas del dispositivo gráfico y por lo tanto, el procesamiento era lento, lo que permitía justificar el dato de tiempo obtenido.

Por todo ello, el siguiente incremento consiste en ampliar la estrategia de forma que, si bien la granularidad sigue considerándose de grano fino, se pueda explotar un mayor número de hilos del dispositivo gráfico. Esto se consigue ascendiendo en el número de bucles anidados a paralelizar.

### Conclusiones principales obtenidas durante la implementación de este desarrollo:

- La primera implementación en CUDA ha resultado ser de grado muy fino y el rendimiento no parece ser adecuado.
- El acceso desde/hasta los datos se realiza de forma convencional y no coalescentemente.
- Se ha tenido que desarrollar una reducción de los datos parciales hallados en GPU, de forma que se disponga de una funcionalidad similar a la disponible en OpenMP.

## 5.3. Paralelización de la función Unificada. Niveles de anidación.

El siguiente incremento consiste en ampliar la estrategia de procesamiento de datos, con el fin de generar un núcleo más eficiente y que disminuya el tiempo de ejecución requerido. Analizando los requisitos del número de iteraciones del código y comparando el total de ellas con el número de hilos que el dispositivo gráfico permite proveer, se ha visto que, para procesar el cuarto bloque en una gran iteración a través de un núcleo CUDA, se dispone de una cantidad de hilos suficiente. Sin embargo, nuevos problemas aparecen debido a las limitaciones internas de la arquitectura.

**Estrategia de optimización y problemas detectados.**

En primer lugar, la acción realizada consistió en adaptar el código implicado para que pudiera procesarse de manera paralela mediante CUDA. Para ello, de forma similar al caso OpenMP, se transformaron los bucles existentes en la función Externa (en este caso los cuatro existentes originalmente, según se puede ver en el Código 18), con el objetivo de que cada hilo tuviese en su posesión una de las combinaciones de la quíntupla que forman las cuatro variables de control implicadas, y el tipo de defecto considerado a procesar, A o B.

```
// Nos movemos de celda en celda ==> hay que incrementar k en 2 unidades
for (kx=0; kx< 2*NCELDASX; kx=kx+2)//Se salta de celda en celda ==> kx = kx +
2
{
  for (ky=0; ky< 2*NCELDASY; ky=ky+2)//Se salta de celda en celda ==> ky = ky
+ 2
  {
    for (kz=0; kz< 2*NCELDASZ; kz=kz+2)//Se salta de celda en celda ==> kz =
kz + 2
    {
      for( kj = 0 ; kj < 4; kj ++)
      {
```

*Código 18: Bucles anidados dispuestos en la función Externa.*

La adaptación de la función Interna se realizó haciendo pequeñas correcciones sobre el código desarrollado en el desarrollo previo. Sin embargo, se eliminó el proceso final de reducción debido a la heterogeneidad que ahora se encuentra en la red de hilos existente, pues en este desarrollo no solo se almacena una llamada a la función Interna, sino todas las requeridas por la función Externa. Así, en este desarrollo se ve implicada la existencia de bloques de hilos cuyas variables corresponden a distintas llamadas de la función originalmente Interna.

**Superación del tiempo de ejecución límite.**

Una vez desarrollado el código se realizó su ejecución, resultando en la finalización incompleta e incorrecta debido a la superación del tiempo límite que impone la arquitectura para la versión concreta disponible, Kepler.

A pesar de eliminar la operación de reducción debido a la heterogeneidad de los datos disponibles en la red establecida, el problema de la superación del tiempo máximo de ejecución en la GPU fue el desencadenante principal de la no ejecución total del núcleo desarrollado. Asimismo, la imposibilidad de obtener un acceso coalescente penalizó aún más el tiempo requerido de ejecución.

**Heterogeneidad de los datos en los hilos lanzados.**

A pesar de la distribución heterogénea generada, sí es posible realizar el procesamiento, aunque no de forma eficiente, debido a un inherente acceso no coalescente. La solución para este problema implica establecer un tamaño de bloque de hilo que impida que ninguno de estos últimos se encuentre junto a otros hilos correspondientes a una llamada diferente de la función Interna.

Esto se puede conseguir realizando una división entre el número total de hilos requerido por el cuarto bloque y algún número que dé lugar a una división exacta. Este último número, el divisor, que establecería el número de hilos por bloque, generalmente no es múltiplo de 32, lo que se traduce en que el acceso coalescente no puede producirse en estas condiciones.

Si el tiempo de ejecución fue ineficiente en la paralelización anterior, en este caso el problema escala en un factor equivalente al número de iteraciones requeridas por la función Externa.

Sin embargo, se realizó un desarrollo aplicando esta variación, estableciendo una red de bloques coherente con el número de hilos requerido por cada llamada a la función Interna, y recuperando el proceso de reducción. El resultado obtenido fue el mismo, es decir, la finalización incorrecta del código debido a la superación del tiempo de ejecución permitido.

En conclusión, mediante esta variación del código no solo no se consiguió lograr la finalización de la ejecución del núcleo, sino que se había conseguido que su tiempo de ejecución se incrementase.

**Limitación en la memoria compartida requerida.**

A pesar de los inconvenientes, incrementar la carga de trabajo en esta variación del núcleo desarrollado permitió ver otra de las limitaciones inherentes al funcionamiento de CUDA, que tiene relación con el proceso de reducción. Ejecutar este procedimiento para la gran cantidad de hilos implicados da lugar a un consumo de memoria compartida proporcional a ellos. El uso de memoria compartida, en contraposición con la memoria global, se debe a su visibilidad a nivel de bloque y a sus menores latencias, en referencia a la memoria global.

Esta memoria, además, no solo es limitada, ya que es dependiente del hardware gráfico, sino que es responsabilidad del desarrollador tener en cuenta la cantidad en uso de ella, debido a que solamente en tiempo de ejecución se obtienen los primeros errores en relación a la imposibilidad de algunos hilos de poder reservarla. Es decir, no se realiza una comprobación en tiempo de compilación que permita analizar si la cantidad necesaria de memoria compartida se encuentra disponible en el dispositivo de ejecución.

Este inconveniente representa un problema importante ya que, si bien se puede calcular manualmente el uso de memoria compartida necesario, no se puede saber si dicha cantidad se va a encontrar accesible al desarrollador, debido a la dependencia entre el número de hilos, el número de multiprocesadores asociados a ellos y la cantidad de memoria compartida asignada por el hardware a estos últimos.

Observando los problemas encontrados, se pensó que, si el problema se encuentra ahora en el gran volumen de datos a procesar, el cual conseguía superar el máximo tiempo de ejecución permitido, así como el máximo de memoria compartida disponible, entonces la solución podría encontrarse en el establecimiento de un procesamiento intermedio al obtenido hasta el momento, balanceando la carga del núcleo, por un lado, y el uso de recursos utilizado, por otro.

Mediante este planteamiento, los dos siguientes incrementos, variaciones en realidad de este último desarrollo, consisten en la paralelización de la función Interna junto a distintos niveles de anidamiento de los bucles de la función Externa. En concreto, la paralelización se estableció para una y dos de las tres dimensiones de las celdas, para lo cual la carga de trabajo se reduce en el mismo factor que el número asociado a cada dimensión: NCELDASX y (NCELDASX)·(NCELDASY), respectivamente, en relación al número de llamadas a la función Interna.

Para los dos planteamientos propuestos, el resultado obtenido es el mismo que el conseguido hasta el momento: la superación del tiempo máximo de ejecución. Sin embargo, la ejecución fallida de estos dos últimos desarrollos, superando los inconvenientes encontrados hasta el momento, dio lugar a la detección final del problema a solucionar: la restructuración del código en busca de la alineación de los datos que permita un acceso coalescente por parte del dispositivo gráfico, de forma que no se supere el tiempo de ejecución límite.

**Conclusiones principales obtenidas durante la implementación de este desarrollo:**

- Las implementaciones de grano más grueso han conseguido saturar los recursos del dispositivo gráfico, así como llegar al tiempo de ejecución máximo.
- Reducir el tamaño de la muestra enviada a la GPU no ha permitido solucionar estos inconvenientes.
- El problema principal parece encontrarse en el acceso de los datos de forma convencional y no coalescentemente.

## 5.4. Paralelización coalescente de la función Interna.

El desarrollo final en la obtención de un núcleo paralelo ejecutable en CUDA consiste en evolucionar la primera de las implementaciones aplicando una restructuración a la forma en que los datos son leídos y escritos, con el objetivo de que estos se encuentren en condiciones de ser procesados óptimamente.

Concretamente, la adaptación en el código se hace atendiendo al acceso que originalmente se realiza sobre los datos iniciales.

**Adaptación del acceso a los datos por parte de CUDA.**

Efectuar esta adaptación, mejor denominada como la reordenación de los datos utilizados para obtener un procesamiento alineado de la función Interna en el núcleo, implica una penalización en el tiempo de ejecución final al requerir de un consumo adicional de tiempo. La reordenación consiste en intercambiar los datos relativos a cada una de las celdas enviadas para su procesamiento en el núcleo, de modo que la lectura por parte de este último se haga coalescentemente.

```
indicecb= kx*2*NCELDASY*2*NCELDASZ*2*6 + ky*2*NCELDASZ*2*6 + kz*2*6 + ksl*6;

indiced= auxx*2*NCELDASY*2*NCELDASZ*2*6 + auxy*2*NCELDASZ*2*6 + auxz*2*6 +
ksl*6;
```

*Código 19: Establecimiento de los índices de procesado de las celdas, para cada una de las iteraciones requeridas.*

La manera de llevarla a cabo consiste en simular dicha lectura en CPU, reorganizando los índices de las celdas según sean requeridos en GPU. Esto se realiza siguiendo los índices correspondientes al procesamiento de las celdas, según se puede ver en el Código 19, y teniendo en cuenta el desfase existente, el cual se determina por las dimensiones de las celdas, ((2xNCELDASX)·(2·NCELDASY)·(2·NCELDASZ))·2. En el Código 20 se puede observar la reordenación total de las tres celdas implicadas.

```
desfase=((2*NCELDASX)*(2*NCELDASY)*(2*NCELDASZ))*2;

for (kde=0; kde<((2*NCELDASX)*(2*NCELDASY)*(2*NCELDASZ))*2; kde++)
{
    //Obtenemos los valores de la red a través del índice kde.
    kx=kde/(4*NCELDASY*NCELDASZ*2);
    ky=(kde%(4*NCELDASY*NCELDASZ*2))/(2*NCELDASZ*2);
    kz=((kde%(4*NCELDASY*NCELDASZ*2))%(2*NCELDASZ*2))/(2);
    ksl=((kde%(4*NCELDASY*NCELDASZ*2))%(2*NCELDASZ*2))%(2);

    //Aplicamos el desplazamiento solo a la celda del defecto
    auxx=kx+incx;
    if(auxx>=nx)
        auxx=auxx-nx;
    else if(auxx<0)
        auxx=auxx+nx;

    auxy=ky+incy;
    if(auxy>=ny)
        auxy=auxy-ny;
    else if(auxy<0)
        auxy=auxy+ny;

    auxz=kz+incz;
    if(auxz>=nz)
        auxz=auxz-nz;
    else if(auxz<0)
        auxz=auxz+nz;

    indicecb= kx*2*NCELDASY*2*NCELDASZ*2*6 + ky*2*NCELDASZ*2*6 + kz*2*6 + ksl*6;
    indiced= auxx*2*NCELDASY*2*NCELDASZ*2*6 + auxy*2*NCELDASZ*2*6 + auxz*2*6 + ksl*6;

    //Realizamos la reordenación de las matrices de manera desglosada, prescindiendo de
bucles
    //Defecto A
    cstdarpc[kde+(desfase*0)] = cstdarp[indiced+0];
    cstdarpc[kde+(desfase*1)] = cstdarp[indiced+1];
    cstdarpc[kde+(desfase*2)] = cstdarp[indiced+2];
```

```
cstdarpc[kde+(desfase*3)] = cstdarp[indiced+3];
cstdarpc[kde+(desfase*4)] = cstdarp[indiced+4];
cstdarpc[kde+(desfase*5)] = cstdarp[indiced+5];

//Defecto B
cstdbrpc[kde+(desfase*0)] = cstdbrp[indiced+0];
cstdbrpc[kde+(desfase*1)] = cstdbrp[indiced+1];
cstdbrpc[kde+(desfase*2)] = cstdbrp[indiced+2];
cstdbrpc[kde+(desfase*3)] = cstdbrp[indiced+3];
cstdbrpc[kde+(desfase*4)] = cstdbrp[indiced+4];
cstdbrpc[kde+(desfase*5)] = cstdbrp[indiced+5];

//Background
cstcbrpc[kde+(desfase*0)] = cstcbrp[indicecb+0];
cstcbrpc[kde+(desfase*1)] = cstcbrp[indicecb+1];
cstcbrpc[kde+(desfase*2)] = cstcbrp[indicecb+2];
cstcbrpc[kde+(desfase*3)] = cstcbrp[indicecb+3];
cstcbrpc[kde+(desfase*4)] = cstcbrp[indicecb+4];
cstcbrpc[kde+(desfase*5)] = cstcbrp[indicecb+5];
}
```

*Código 20: Reordenación de las tres celdas implicadas en el procesamiento posterior.*

Gracias a la optimización de la compilación en CPU, se consigue que la sobrecarga implicada en este proceso sea mínima, obteniendo así reordenamientos muy rápidos, del orden de $10^{-2}$ segundos. A pesar de ello, para cada llamada al núcleo se debe hacer una nueva reordenación, por lo que el tiempo acumulativo sí es notable. El número de llamadas al núcleo, y por tanto el número de reordenaciones requeridas, se corresponde con el número de iteraciones efectuadas por la función Externa.

En este último código, el Código 20, se puede ver que lo que se persigue es conseguir la lectura total de cada una de las celdas en seis únicas iteraciones.

En la primera de ellas, cada uno de los hilos implicados en CUDA, de un total de ((2xNCELDASX)·(2·NCELDASY)·(2·NCELDASZ))·2, leerá la posición número 0 que le corresponde. Así, consecutivamente, todos los hilos leerán su posición 0 requerida.

El proceso continúa para las siguientes posiciones, hasta llegar a la posición número 5. Con las seis posiciones correspondientes a cada hilo leídas, y repitiendo el proceso para el resto de celdas requeridas, cada uno de estos hilos ya puede realizar su propio procesamiento, en forma de sumas y productos. Todo este proceso puede verse en el Código 22, referente al núcleo CUDA desarrollado.

Una vez alineados los datos, el siguiente paso es enviarlos a la GPU. Se sabe que la paralelización que se pretende implementar es equivalente al procesamiento de la función Interna y que, si bien al principio el número de llamadas a esta función por parte de la Externa era de dos, este requisito se consiguió unificar en una sola llamada en la iteración final. De manera análoga se va a realizar en este caso, simplificando las dos llamadas mediante la duplicación de las operaciones a procesar por cada uno de los hilos asignados.

Determinada la forma en que se va a producir el procesamiento de la función Interna, y disponiendo en este momento de todos los datos de manera alineada para que puedan ser leídos coalescentemente por la arquitectura, el código del núcleo obtenido es en este momento sencillo. El Código 21 proporciona su llamada.

```
////Llamada al núcleo
calcEnerEl<<<gridShape, blockShape>>>(cstdarp_cuda, cstdbrp_cuda, cstcbrp_cuda, salidac);
```

*Código 21: Llamada al núcleo de ejecución en el dispositivo gráfico.*

Siguiendo el Código 22, correspondiente al núcleo desarrollado, únicamente se deben leer de manera consecutiva los datos para cada una de las celdas a procesar (algo implícito por la lectura mediante *warps* buscada), realizar seguidamente las operaciones correspondientes con la función originalmente Interna y depositar estos resultados, de la misma forma en que se han leído, es decir, coalescentemente, sobre memoria global, para ser finalmente reducidos en CPU y obtener el resultado buscado.

```
__global__  void  calcEnerEl(double  *cstdarpc,  double  *cstdbrpc,  double
*cstcbrpc, double *salida)
{
    //Mapeamos  en  primer  lugar  cada  uno  de  los  hilos  de  procesamiento,
asignándole un índice
    //Índice del bloque en X * dimensión de cada bloque en X + índice del hilo
(dentro del bloque seleccionado)
    int desfase=((2*NCELDASX)*(2*NCELDASY)*(2*NCELDASZ))*2;
    int índice;
    índice = (blockIdx.x) * blockDim.x  + threadIdx.x;

    //Establecemos las matrices de procesamiento de cada hilo
    double cstcbrp[6];
    double cstdarp[6];
    double cstdbrp[6];

    //Ordenamos terminar a los hilos que sobran en el procesamiento
    if (índice > desfase)
        return;

    //Realizamos las asignaciones de datos
    //Se  realizan  desglosadas  para  favorecer  la  coalescencia  y  mejorar  el
rendimiento

    //Background
    cstcbrp[0] = cstcbrpc[(desfase*0)+ índice];
    cstcbrp[1] = cstcbrpc[(desfase*1)+ índice];
    cstcbrp[2] = cstcbrpc[(desfase*2)+ índice];
    cstcbrp[3] = cstcbrpc[(desfase*3)+ índice];
    cstcbrp[4] = cstcbrpc[(desfase*4)+ índice];
    cstcbrp[5] = cstcbrpc[(desfase*5)+ índice];

    //Defecto A
    cstdarp[0] = cstdarpc[(desfase*0)+ índice];
    cstdarp[1] = cstdarpc[(desfase*1)+ índice];
    cstdarp[2] = cstdarpc[(desfase*2)+ índice];
    cstdarp[3] = cstdarpc[(desfase*3)+ índice];
    cstdarp[4] = cstdarpc[(desfase*4)+ índice];
    cstdarp[5] = cstdarpc[(desfase*5)+ índice];

    //Defecto B
    cstdbrp[0] = cstdbrpc[(desfase*0)+ índice];
    cstdbrp[1] = cstdbrpc[(desfase*1)+ índice];
    cstdbrp[2] = cstdbrpc[(desfase*2)+ índice];
    cstdbrp[3] = cstdbrpc[(desfase*3)+ índice];
    cstdbrp[4] = cstdbrpc[(desfase*4)+ índice];
    cstdbrp[5] = cstdbrpc[(desfase*5)+ índice];

    // Componentes del strain
    // i=0 -> xx, i=1 -> yy, i=2 -> zz
    // i=3 -> xy, i=4 -> xz, i=5 -> yz

    //Establecemos directamente la salida, que asimismo se realiza de manera
coalescente
    salida[indice+0*desfase]= ( cstcbrp[0]*cstdarp[0] +
cstcbrp[1]*cstdarp[1] + cstcbrp[2]*cstdarp[2] );

    salida[indice+1*desfase]= ( cstcbrp[0]*(cstdarp[1] + cstdarp[2]) +
cstcbrp[1]*(cstdarp[0] + cstdarp[2]) + cstcbrp[2]*(cstdarp[0] + cstdarp[1])
);

    salida[indice+2*desfase]= ( cstcbrp[3]*cstdarp[3] +
cstcbrp[4]*cstdarp[4] + cstcbrp[5]*cstdarp[5] );

    salida[indice+3*desfase]= 10 + ( cstcbrp[0]*cstdbrp[0] +
cstcbrp[1]*cstdbrp[1] + cstcbrp[2]*cstdbrp[2] );

    salida[indice+4*desfase]= ( cstcbrp[0]*(cstdbrp[1] + cstdbrp[2]) +
cstcbrp[1]*(cstdbrp[0] + cstdbrp[2]) + cstcbrp[2]*(cstdbrp[0] + cstdbrp[1])
);
```

```
    salida[indice+5*desfase]= ( cstcbrp[3]*cstdbrp[3] +
cstcbrp[4]*cstdbrp[4] + cstcbrp[5]*cstdbrp[5] );

    __syncthreads();
}
```

*Código 22: Núcleo final desarrollado mediante el modelo CUDA.*

Descrito el código CUDA, se puede observar que, para este desarrollo, no se ha decidido aplicar una reducción previa en la GPU de los resultados parciales obtenidos por los hilos.

La causa se encuentra en que realizar este proceso de reducción da como resultado conseguir que el hilo 0 de cada bloque disponga del resultado de aplicar la reducción. Sin embargo, este conjunto de hilos 0 no se encuentra distribuido de forma consecutiva, sino que cada uno pertenece a una posición asociada al bloque en el que se encuentre.

Esto se traduce en la pérdida de la propiedad de coalescencia que se ha buscado durante el desarrollo implementado, de forma que el volcado de los datos finales a memoria global se debiese realizar de manera convencional, volviendo de nuevo a obtener una pérdida de rendimiento.

**Procesamiento de operaciones llevado a cabo en CPU y GPU.**

Otra de las diferencias entre el procesamiento efectuado en CPU y GPU se encuentra inherente a las arquitecturas implicadas y, concretamente, a la forma en que se calculan los resultados, obteniendo discrepancias que, si bien no son notorias de manera aislada, sí generan una variación en la mayor parte de las cifras decimales de los resultados finales.

La causa de la no coincidencia de los resultados, para los que se ha aplicado el mismo algoritmo de procesamiento, se encuentra en el error de redondeo implícito a cada arquitectura concreta, que realiza las operaciones a bajo nivel de forma diferente. Este error de redondeo se maximiza debido a la variabilidad de los números implicados en las operaciones, encontrándose en un rango de entre $10^{-6}$ y $10^{6}$.

Así, se llega a la conclusión de que algunos de estos resultados se han considerado incorrectos, al existir una discrepancia lo suficientemente grande entre ambas versiones, cuya diferencia se encuentra a varios ordenes de magnitud y con hasta cuatro, de seis en total, cifras decimales significativas incorrectas.

Debido a que el objetivo final es obtener la misma salida que inicialmente se consigue con el código original, se considera errónea la obtención de resultados excesivamente distintos.

Por todo ello, la decisión finalmente adoptada en la elaboración del núcleo ha consistido, como puede verse en el Código 22, en restringir el número de operaciones en CUDA, teniendo siempre en cuenta la maximización en el aprovechamiento de cada llamada a la GPU.

En términos prácticos, se han eliminado, siguiendo este planteamiento, todos los procesamientos de productos y cambios de signo, los cuales se pueden efectuar en una sola operación en la CPU.

Obtenido finalmente un núcleo cuyo procesamiento aprovecha las propiedades de CUDA, se ha conseguido paralelizar la función Interna en una única iteración que realiza, de manera equivalente, un total de (((2xNCELDASX)·(2·NCELDASY)·(2·NCELDASZ))·2)·2 iteraciones. Disponiendo del código implementado, el siguiente paso es verificar su eficiencia.

### v. Quinta salida de resultados.

Teóricamente parece que el volumen de procesamiento escogido para la GPU es aceptable y que el rendimiento se va a ver incrementado en consecuencia, pero lo cierto es que no es así. Mediante la realización de varias simulaciones del software, se ha podido ver que los resultados no han conseguido una mejora suficiente como para encontrarse a una escala similar al modelo paralelo proporcionado por OpenMP.

A continuación se muestran, en la Tabla 19, los resultados de las pruebas, acompañados, de nuevo, por los resultados de mejor rendimiento obtenidos previamente, tanto mediante el código en serie optimizado por el compilador, como aplicando el modelo OpenMP.

| | Serie | OpenMP | CUDA |
|---|---|---|---|
| | 1 opt (ref) | 8 unif. | Base |
| **Externa** | 3922,70 | 1372,82 | 22974,29 |
| **Speedup %** | -> | **2,86** | **0,17** |
| **Page Faults** | 782,00 | 1262507,00 | 164931383,00 |

*Tabla 19: Representación comparativa de los resultados obtenidos mediante el modelo CUDA.*

Como se puede observar, el rendimiento no es adecuado, al obtener una aceleración de 0,17.

Las causas se encuentran en la penalización en el tiempo añadido mediante la reordenación de los datos. Asimismo, se debe tener en cuenta que durante la ejecución de este desarrollo se está llevando a cabo, de forma simultánea, el procesamiento de los dos defectos implicados en cada una de las llamadas originales a la función Interna.

Además de las causas relacionadas con el procesamiento de los datos y con los límites de la arquitectura, mediante estos resultados se puede observar un aumento altamente significativo en el número de fallos de página sucedidos durante la ejecución.

En el experimento CUDA realizado durante este desarrollo, el problema parece encontrarse, de manera análoga a lo que ocurría cuando se trataba de paralelizar la función Interna mediante OpenMP, en el número de llamadas realizadas al núcleo de procesamiento, implicando una gran cantidad de cambios de contexto en llamadas a una función, un núcleo en este caso, cuyo tiempo de procesamiento es del orden de $10^{-2}$ segundos por llamada.

La Figura 18 permite ver, de manera gráfica, la comparación de rendimiento entre las distintas pruebas efectuadas para este trabajo.

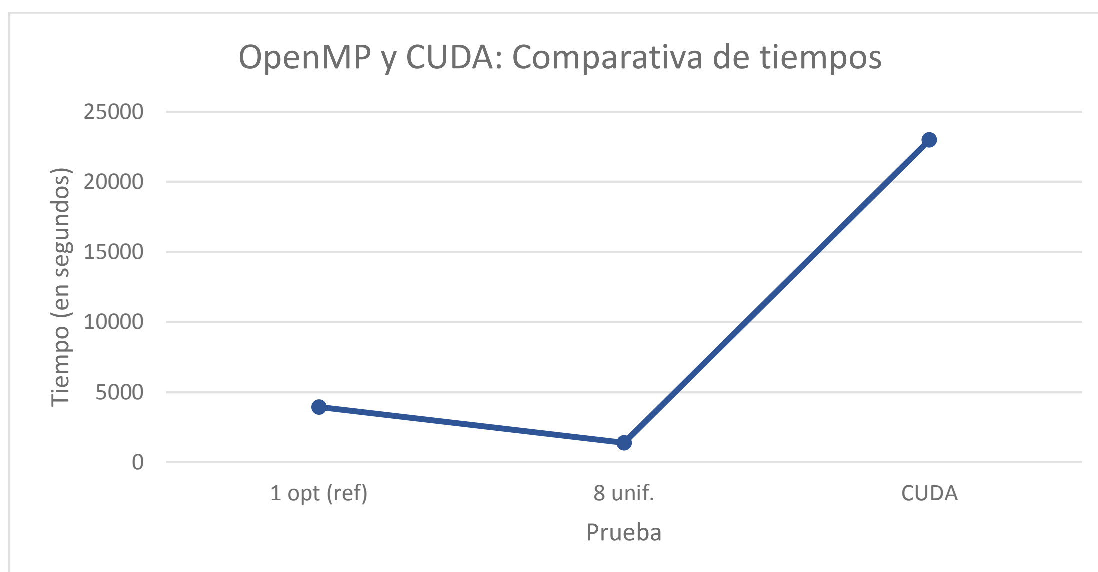


*Figura 18: Gráfico ilustrativo de los tiempos obtenidos para OpenMP y CUDA.*

**Consideraciones adicionales.**

Haciendo referencia a las limitaciones inherentes al modelo de procesamiento, en relación a la extensión de datos a procesar, su ampliación manteniendo la coalescencia se dificulta.

Por un lado, el código original impone dimensiones en sus celdas de 37. Este número es primo, definiendo unas limitaciones en sus capacidades de divisibilidad, complicando la obtención de bloques de hilos que respeten la organización que los niveles de anidación requieren para llevar a cabo un procesamiento adecuado, así como su divisibilidad por un factor de 32, requerido para llevar a cabo el acceso coalescente.

Escoger un número más adecuado podría minimizar o solventar esta limitación, aunque igualmente el código deberá ser siempre adaptado a cada condición específica, debido a las características de la arquitectura.

Por otro lado, a niveles más externos de procesamiento, se tiene un mayor riesgo de incurrir nuevamente en problemas de exceso en el tiempo máximo de ejecución.

Existen asimismo dos inconvenientes adicionales que impiden ampliar el desarrollo en esta versión de la arquitectura CUDA, Kepler:

- la dificultad en la creación de una red de hilos, mediante la concatenación de varios de los bucles implicados en la función Externa, que no exceda los recursos disponibles, así como la creación de un desarrollo que soporte el acceso coalescente requerido por CUDA, y
- las limitaciones inherentes al hardware, en referencia al procesamiento interno de las operaciones, así como al tiempo máximo posible de ejecución.

Se concluye así que el núcleo obtenido es el desarrollo que mejor aprovecha las características del hardware existente, teniendo en cuenta las limitaciones impuestas por el modelo de funcionamiento de la arquitectura y su versión específica, Kepler.

El núcleo desarrollado es eficiente y coalescente, en el contexto CUDA. Haciendo referencia al desarrollo realizado en OpenMP no puede decirse lo mismo, según se ha observado en la Figura 18.

## 5.5. Conclusiones establecidas acerca del modelo.

Tras el desarrollo efectuado, y después de haber estudiado las posibilidades disponibles con la motivación de aprovechar la arquitectura elegida, ha llegado el momento en que pueden establecerse las conclusiones de acuerdo al rendimiento obtenido por la solución funcional válida, que ha consistido en la paralelización coalescente de la función Interna del código inicialmente proveído.

A pesar del progreso llevado a cabo mediante estos desarrollos, se ha podido observar, a través de los resultados, que ninguna de las soluciones se puede tomar como válida si se toma en consideración el fin último de la investigación, la mejora del tiempo original que se obtenía en primera instancia.

Las causas que han impedido explotar aún más la arquitectura se encuentran en su naturaleza y, relacionado muy directamente, en el problema a resolver.

La arquitectura CUDA se encuentra especialmente diseñada para efectuar el procesamiento de grandes volúmenes de números enteros, y el problema que este software pretende solucionar emplea números en coma flotante de muy distinto rango, con órdenes de entre $10^{-6}$ hasta $10^{6}$.

Esta variabilidad es lo que inicialmente comienza a provocar un procesamiento más ineficiente, debido a que si bien la plataforma puede operar con números en coma flotante, su procesamiento es más limitado, según la documentación técnica.

Tanto es así que dichas operaciones se encuentran mejor resueltas, en relación a la operativa interna, así como al control de desbordamientos y de errores de redondeo, en la arquitectura proveída por las CPUs de propósito general. Esta última diferencia en el cálculo de los resultados es lo que ha impedido que desde la GPU se hayan obtenido valores iguales a los que se obtienen en la CPU.

Dos problemas surgen inherentemente de la aplicación de la arquitectura en la optimización de este software mediante la adaptación y paralelización del código.

- Por un lado, las limitaciones en el tiempo de ejecución máximo permitido, así como la saturación de la memoria compartida al utilizar el número de hilos requerido, impiden realizar un procesamiento exhaustivo.
- Por otro lado, existe un problema en la escalabilidad permitida originalmente en el software en su ejecución a través de CUDA. El número de unidades utilizado para representar cada una de las dimensiones de las celdas consideradas es de 37, un número que, por ser primo, da pocas posibilidades de ser adaptado a un núcleo CUDA, ya que el código realiza el procesamiento de cierto número de bucles anidados que disponen de variables de control basadas en este número.

  Si bien este problema podría ser adecuado escogiendo otro número, el desarrollo continúa siendo complicado debido a la dificultad en la búsqueda de una implementación que pueda ser parametrizada, y para la cual se pueda garantizar su ejecución a través de un dispositivo de recursos limitados para el problema que se pretende resolver.

La implementación conseguida, sin embargo, sí es eficiente, al aprovechar los recursos del dispositivo gráfico. Sin embargo, se tiene en consideración que el aprovechamiento general de este recurso no ha sido adecuado al no poder ejecutar una estrategia más acorde al modelo.

Para terminar, haciendo una analogía en referencia al código obtenido a través de OpenMP, en CUDA, la dificultad de obtener un software cuya ejecución sea independiente del tamaño de la muestra de entrada requiere en primera instancia del cálculo de la red de hilos para realizar el procesamiento.

Asimismo, este cálculo es dependiente, no solo del tamaño de la muestra de entrada, sino también del tipo de procesamiento a llevar a cabo, dependiendo del nivel paralelizado, bien sea únicamente la función Interna o cualquiera de los niveles de la función Externa, o la función completa, representados por los bucles anidados implicados.

Todo ello da lugar a un fuerte acoplamiento y a una dependencia del software frente a la versión de la arquitectura considerada. Así, características conseguidas en OpenMP, como la robustez o la mantenibilidad, se ven limitadas en este modelo.

El modelo que ofrece OpenMP, en el que el procesamiento se efectúa mediante grupos de trabajo que reparten el total de trabajo entre cada uno de los hilos de procesamiento disponibles, es más adecuado al ofrecido por CUDA, en el cual, si bien el trabajo también es desglosado entre su gran cantidad de hilos, este trabajo es de una granularidad mucho más fina y dependiente de la red establecida.

Aunque sería posible establecer la misma cantidad de hilos para el procesamiento CUDA, no puede hacerse lo mismo con las características de memoria de las que sí dispone OpenMP, impidiendo que el planteamiento funcione. Por cada nueva iteración que cada uno de estos hilos CUDA debiera llevar a cabo, se debe duplicar la cantidad de información a almacenar en la GPU, al requerir la entrada reordenada de los datos para cada procesamiento a efectuar, si se quiere mantener el acceso coalescente.

**Conclusiones principales tras la aplicación de este modelo paralelo:**

- CUDA ha permitido la generación de un núcleo de procesamiento eficiente de datos. Sin embargo, las propiedades de la arquitectura, así como las características del software, han impedido una explotación más adecuada de la arquitectura.
- Actuando en el tamaño y procesamiento de las muestras de entrada, en consistencia con el acceso a datos requerido por CUDA, se podría obtener una mejora en el rendimiento conseguido.
- La escalabilidad de las soluciones depende de la mejora de rendimiento inherente al desarrollo realizado, así como de la versión de la arquitectura utilizada.

# Capítulo 6
# Paralelización combinada de OpenMP y CUDA

Tras la investigación y desarrollo de las capacidades de ambos modelos de procesamiento aplicados al software proveído, en este capítulo se analizan las estrategias posibles de combinación entre ellos.

Así, se exploran nuevas mejoras en los rendimientos conseguidos hasta el momento.

### 6.1. Estrategias de paralelización combinada.

Explorados los dos modelos de paralelismo, surge un nuevo desarrollo potencial: la combinación de ambos, tratando de explotar sus beneficios conjuntamente para un mejor aprovechamiento de los recursos del sistema en ejecución. Para ello, se han elaborado dos estrategias a efectuar.

**Mejora en el rendimiento del núcleo CUDA mediante el lanzamiento concurrente a través de OpenMP.**

La primera de las estrategias consiste en paralelizar el núcleo CUDA de manera que sean lanzados varios núcleos concurrentemente a la GPU, acelerando así el procesamiento en la medida que las capacidades del dispositivo gráfico lo permitan.

El desarrollo pasa por establecer una región paralela justo antes de la preparación del lanzamiento del núcleo CUDA, en la que se realiza la reserva de recursos. La región paralela finaliza tras la liberación de estos recursos reservados, de forma que queden libres para nuevas iteraciones.

De manera similar a los desarrollos paralelos en OpenMP, se indicaron en la región paralela las variables que deben ser consideradas como compartidas y privadas. En este sentido, los datos correspondientes con las celdas a procesar se establecen como compartidos, al ser únicamente de solo lectura. No ocurre lo mismo con los punteros a la GPU de dichos datos, los cuales se consideraron privados, al pertenecer, cada uno de ellos, a diferentes lanzamientos del núcleo.

Una vez realizados los ajustes requeridos, se estableció una batería de pruebas para comprobar las limitaciones del dispositivo CUDA, en relación a la disponibilidad de memoria e hilos, al lanzar varios núcleos consecutivos.

No fueron requeridas demasiadas pruebas ya que, a través de las diversas comprobaciones de errores dispuestas en el código desarrollado, CUDA devolvió mensajes de error haciendo referencia a problemas de adquisición y liberación de memoria, así como, seguidamente, tras varias iteraciones, códigos de errores no determinados en los lanzamientos de los núcleos. Así pues, debido a las limitadas capacidades de gestión y almacenamiento de recursos en la GPU, no se pudo continuar con esta estrategia.

**Mejora en la estrategia paralela CUDA mediante la reordenación paralela de datos a través de OpenMP.**

La segunda estrategia a desarrollar, no intrusiva a nivel del dispositivo gráfico, consiste en paralelizar la reordenación de los datos correspondientes a las celdas a procesar, que seguidamente son enviados a la GPU para su acceso de forma coalescente.

La manera de implementar esta optimización sobre el código se realiza mediante el establecimiento de la habitual región paralela a través del *pragma* del que dispone OpenMP.

Así pues, se establecieron las variables de control como privadas, así como de manera compartida los datos referentes a las celdas de proceso, y se estableció un desglose estático, permitiendo así una división equitativa del trabajo, además de una reordenación independiente por cada uno de los hilos, lo que da lugar a un procesamiento limpio y disjunto. Con ello, el bucle se encontraría ya paralelizado. El Código 23 describe el *pragma* incorporado, así como el bucle de reordenación.

```
//Reorganizamos los índices y recolocamos de acuerdo a la arquitectura de CUDA

#pragma omp parallel for default(shared) private(kx, ky, kz, ksl, auxx, auxy,
auxz, indicecb, indiced, kde) num_threads(hilos)

for (kde=0; kde<8*NCELDASX*NCELDASY*NCELDASZ*2; kde++)
{
    //Obtenemos los valores de la red a través del índice kde.
    kx=kde/(4*NCELDASY*NCELDASZ*2);
    ky=(kde%(4*NCELDASY*NCELDASZ*2))/(2*NCELDASZ*2);
    kz=((kde%(4*NCELDASY*NCELDASZ*2))%(2*NCELDASZ*2))/(2);
    ksl=((kde%(4*NCELDASY*NCELDASZ*2))%(2*NCELDASZ*2))%(2);

    indicecb= kx*2*NCELDASY*2*NCELDASZ*2*6 + ky*2*NCELDASZ*2*6 + kz*2*6 +
ksl*6;
    indiced= auxx*2*NCELDASY*2*NCELDASZ*2*6 + auxy*2*NCELDASZ*2*6 + auxz*2*6
+ ksl*6;
```

```
    //Realizamos la reordenación de las matrices de manera desglosada,
prescindiendo de bucles
    //Defecto A
    cstdarpc[kde+(desfase*0)] = cstdarp[indiced+0];
    cstdarpc[kde+(desfase*1)] = cstdarp[indiced+1];
    cstdarpc[kde+(desfase*2)] = cstdarp[indiced+2];
    cstdarpc[kde+(desfase*3)] = cstdarp[indiced+3];
    cstdarpc[kde+(desfase*4)] = cstdarp[indiced+4];
    cstdarpc[kde+(desfase*5)] = cstdarp[indiced+5];

    //Defecto B
    cstdbrpc[kde+(desfase*0)] = cstdbrp[indiced+0];
    cstdbrpc[kde+(desfase*1)] = cstdbrp[indiced+1];
    cstdbrpc[kde+(desfase*2)] = cstdbrp[indiced+2];
    cstdbrpc[kde+(desfase*3)] = cstdbrp[indiced+3];
    cstdbrpc[kde+(desfase*4)] = cstdbrp[indiced+4];
    cstdbrpc[kde+(desfase*5)] = cstdbrp[indiced+5];

    //Background
    cstcbrpc[kde+(desfase*0)] = cstcbrp[indicecb+0];
    cstcbrpc[kde+(desfase*1)] = cstcbrp[indicecb+1];
    cstcbrpc[kde+(desfase*2)] = cstcbrp[indicecb+2];
    cstcbrpc[kde+(desfase*3)] = cstcbrp[indicecb+3];
    cstcbrpc[kde+(desfase*4)] = cstcbrp[indicecb+4];
    cstcbrpc[kde+(desfase*5)] = cstcbrp[indicecb+5];
    }
```

*Código 23: Reordenación de los datos de las celdas mediante OpenMP.*

La aceleración a conseguir mediante esta región paralela creada a través del *pragma* es, a lo sumo, proporcional al número de hilos de trabajo asignados a ella. Mediante una batería de pruebas adicional se va a comprobar el verdadero efecto de haber incorporado esta región paralela.

### vi. Sexta salida de resultados.

Tras realizar las pruebas correspondientes en la Plataforma TFG, mediante la Tabla 20 puede observarse el resultado obtenido de la aplicación cooperativa de OpenMP y CUDA, junto a los mejores resultados hallados previamente para ambos modelos.

| | Serie | OpenMP | CUDA | |
|---|---|---|---|---|
| | 1 opt (ref) | 8 unif. | Base | + 8 x OpenMP |
| **Externa** | 3922,70 | 1372,82 | 22974,29 | 22553,02 |
| **Speedup %** | -> | **2,86** | **0,17** | **0,17** |
| **Page Faults** | 782,00 | 1262507,00 | 164931383,00 | 164929195,00 |

*Tabla 20: Representación de los resultados obtenidos mediante la combinación de ambos modelos.*

En ella se puede ver que el resultado no es el mismo que el planteado de forma teórica. Esto es debido a que para llevar a cabo un procesamiento paralelo útil, en este caso por parte de OpenMP, se requiere de un tiempo mínimamente reducible de procesamiento en las operaciones internas subyacentes, como los movimientos en la caché/memoria RAM, realizados por el sistema operativo, al operar.

Teniendo en cuenta que el tiempo de procesamiento de la reordenación se encuentra en ordenes de los $10^{-2}$ segundos, la división de este tipo de trabajo en varios hilos de ejecución dificulta la obtención de una aceleración efectiva, al apenas disponer de tiempo sobre el que operar paralelamente. Algo similar ocurría, aunque menos notablemente, en la paralelización inicial de la función Interna mediante OpenMP.

Aplicando una paralelización utilizando disposiciones de dos y tres hilos de ejecución, sí se observa que el tiempo se reduce en un factor aproximadamente proporcional al número de hilos asignados en cada caso concreto.

Asignar más hilos de cómputo consigue mejoras adicionales, aunque más sutiles, reduciendo el factor de aceleración por hilo utilizado. Sin embargo, dado que dichos hilos restantes permanecerían ociosos en este punto del código, la prueba realizada hace uso de la mayor cantidad de hilos disponible.

La aplicación del modelo OpenMP, como complemento al procesamiento paralelo que realiza el modelo CUDA, no ha permitido obtener una mejora suficiente en el tiempo de ejecución efectivo, ya que la cifra en relación al tiempo ha variado muy levemente con respecto al tiempo conseguido únicamente mediante la aplicación de CUDA.

De esta forma, el tiempo se mejora en apenas 400 segundos, un tiempo insuficiente como para significar una mejora notable. En consecuencia, la aceleración final obtenida se mantiene en 0,17. Mediante la Figura 19 puede observarse, de manera gráfica, cada una de las cifras de tiempo obtenidas durante todo el trabajo.

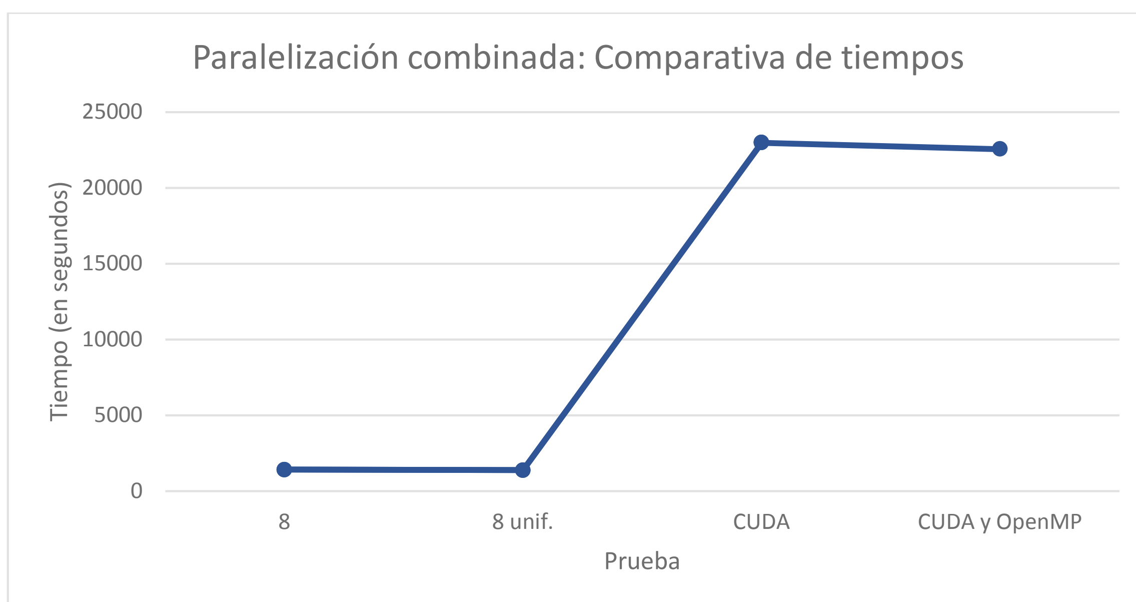


*Figura 19: Gráfico ilustrativo de los tiempos obtenidos para OpenMP, CUDA y su combinación.*

# Capítulo 7
## Conclusiones finales

Este capítulo cierra el desarrollo del trabajo exponiendo los resultados finales obtenidos tras la investigación realizada.

Finalmente se elige la solución más adecuada para el problema planteado según las soluciones obtenidas a través de los dos modelos paralelos implementados en el código proveído.

## 7.1. Elección final para el problema solucionado.

Tras la investigación e implementación de soluciones de forma incremental en el software, con la motivación de obtener aceleraciones y procesamientos más eficientes y depurados, se puede establecer una conclusión final.

**Mejor optimización conseguida: paralelización de la función Adicional y de la función Unificada mediante OpenMP.**

El desarrollo que ha permitido obtener la mejor de las aceleraciones se corresponde con el desarrollo a través del modelo OpenMP, en el que se ha integrado la función Interna en la función Externa para, de esta forma, producir una unificación entre ambas funciones paralelas, consiguiendo asimismo el procesamiento simultaneo de los datos correspondientes a los dos defectos, simplificando el software y mejorando la productividad.

Atendiendo a la facilidad de adaptación del código y a la implementación correspondiente a ambos modelos, dependiente de la metodología subyacente, bien sea a través del mecanismo *fork-join*, o por la utilización de una arquitectura que difiere del procesamiento convencional en la CPU, se puede concluir que OpenMP proporciona una solución más adecuada al contexto del problema, permitiendo obtener un software cuyo procesamiento es independiente de las dimensiones de las celdas consideradas.

Además, OpenMP permite mantener el grado de mantenibilidad al conseguir un código paralelo cuyas diferencias con respecto al código original son mínimas, consiguiendo así que un equipo no especializado en tareas de desarrollo de software paralelo pueda realizar posteriores ampliaciones sobre el código obtenido, de forma sencilla.

Por otro lado, la optimización conseguida se sostiene en un procesamiento independiente para cada uno de los hilos, al procesar porciones equitativas de datos, haciendo uso para ello de ficheros temporales de almacenamiento intermedio de resultados, permitiendo así la colaboración entre los hilos y logrando finalmente seguir el orden previsto en serie que realiza el software original.

Por último, destacar el carácter autocontenido del software, al hacer un uso eficiente de los recursos hardware disponible, efectuando operaciones de limpieza sobre los recursos utilizados de manera temporal, de forma que la salida final únicamente comprende el fichero de salida que se obtendría mediante el software original.

Esta funcionalidad adicional, si bien fue requerida en el diseño y desarrollo de las implementaciones mediante OpenMP, ha permitido obtener un software mejor formado, más eficiente e igualmente eficaz.

**Comparación de la optimización con las posibilidades ofrecidas por CUDA.**

La funcionalidad suplementaria desarrollada para OpenMP no ha podido aprovecharse durante la implementación de los desarrollos CUDA, debido a la alta dependencia, con respecto a la arquitectura, del dispositivo gráfico.

La manera inherente en que el paralelismo se efectúa es el concepto que marca la diferencia para cada uno de los dos modelos. Mientras que OpenMP emplea un desglose basado en la distribución equitativa de trabajo entre el, generalmente, reducido número de hilos disponibles, CUDA utiliza un procesamiento altamente paralelo de un código de grano muy fino, al disponer de una gran cantidad de hilos.

Esto permite a OpenMP escalar el problema tanto como el desarrollador requiera o hilos se encuentren disponibles, sin por ello penalizar la programación o las estructuras de datos involucradas, mientras CUDA sí requiere una adaptación continua del código a las condiciones cambiantes del entorno de ejecución, para poder obtener una versión que cumpla las propiedades específicas de la arquitectura destino.

Se debe también tener en cuenta que el procesamiento aritmético del software comprende la operación de una gran cantidad de sumas y productos sobre números en coma flotante, en muy diferentes órdenes de magnitud, el cual debe ser llevar a cabo de manera exacta al efectuado por el software original.

Mientras el objetivo de CUDA generalmente es el procesamiento masivo de enteros, el uso de la CPU por parte de OpenMP, así como el no requerir de cambios de contexto adicionales, permite concluir que es el modelo que mejor se adecúa a la solución del problema, aprovechando eficientemente los recursos y

obteniendo una aceleración, por unidad de tiempo, superior a CUDA en todos los desarrollos dispuestos durante la elaboración de este trabajo.

En la Tabla 21 y en la Figura 20 puede observarse el rendimiento conseguido en la Plataforma TFG para cada configuración desarrollada, optimizada por el compilador, utilizando cada uno de los dos modelos. Asimismo, se muestra el tiempo originalmente proveído por el software en serie.

| | Plataforma TFG - Tiempos totales | | | |
|---|---|---|---|---|
| | **Serie** | **OpenMP** | **CUDA** | **CUDA + OpenMP** |
| **Adicional** | 1134,24 | 88,63 | | |
| **Unificada** | 11387,37 | 1372,82 | 22974,29 | 22553,02 |
| **No Paralelizado** | 0,59 | | | |
| **Total** | **12522,20** | **1462,04** | **23063,51** | **22642,24** |
| **Speedup** | -> | **8,56** | **0,54** | **0,55** |

*Tabla 21: Tiempos totales obtenidos para la Plataforma TFG.*

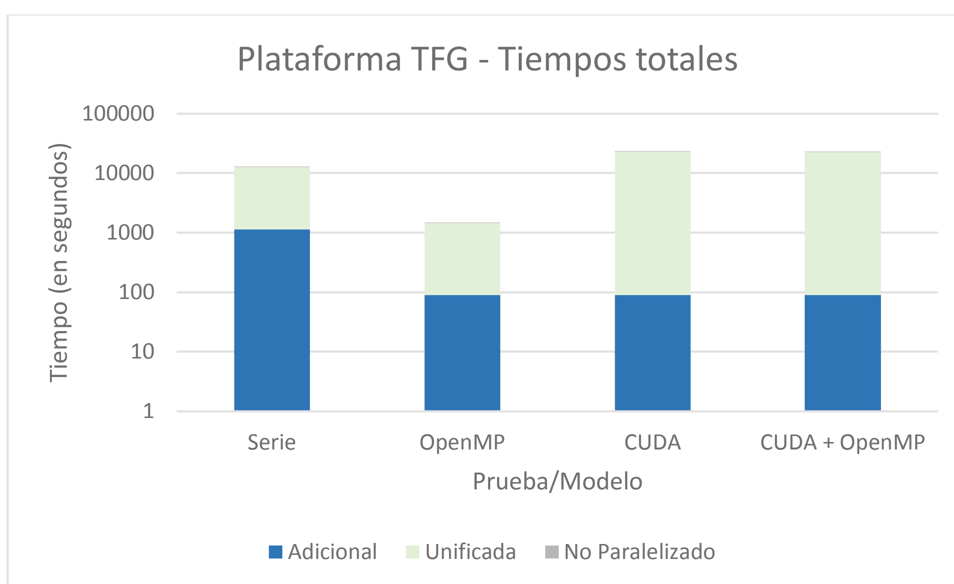


*Figura 20: Representación gráfica apilada de los tiempos obtenidos para cada una de las configuraciones disponibles.*

# Apéndice
## Consideraciones debidas al hardware

Tras concluir el desarrollo, en este apéndice, que sirve de cierre del trabajo, se pasan a exponer resultados complementarios a la investigación.

Para ello se discute acerca del consumo de energía efectuado en las implementaciones de ambos modelos y del efecto optimizador del uso del compilador.

# Apéndice 1. Consumo energético asociado a cada modelo paralelo.

Un aspecto a tener en cuenta en el desarrollo de investigaciones de cálculo intensivo es el coste que tiene efectuarlas. Durante este trabajo se han realizado pruebas y experimentos aplicando los dos modelos de trabajo considerados. Asimismo, dichos experimentos se han llevado a cabo sobre un total de tres sistemas informáticos de características diferentes.

Es por ello que en este apartado se va a abordar, mediante una rápida perspectiva, el rendimiento obtenido en cada prueba, en relación al consumo de energía correspondiente para cada uno de los sistemas considerados.

Los datos acerca de los consumos de los componentes han sido obtenidos de la respectiva documentación de cada uno de los fabricantes.

Para realizar los análisis comparativos se van a tener en cuenta los dispositivos de procesamiento intensivo, es decir, la CPU y la GPU, si es el caso. Se desprecian los consumos de memoria RAM, así como los relacionados con los discos duros y otros componentes, debido a un uso energético inferior y mucho más complejo de medir en estas condiciones.

Asimismo, se va a utilizar el valor de la tarifa por defecto del PVPC del día 10 de Julio de 2016 para obtener el coste que tiene ejecutar el procesamiento en cada uno de los sistemas disponibles. El valor de la tarifa es de 0,11250 €/kWh.

El tiempo utilizado para calcular los consumos de energía de cada sistema considerado se ha obtenido escogiendo el mejor para cada modelo implementado, bien sea OpenMP o CUDA.

A continuación se pasa a analizar cada uno de los sistemas utilizados, comenzando por aquel que obtuvo el mejor tiempo durante las pruebas, la Plataforma TFG.

**a)** **Plataforma TFG.** Este sistema cuenta con las características requeridas para afrontar el procesamiento del software mediante OpenMP y CUDA. Los componentes asociados a estos procesamientos consumen, a máximo rendimiento, un total de 95 Watios para la CPU, y de 110 Watios para la GPU.

Haciendo referencia a los resultados obtenidos durante el análisis de rendimiento inicial mediante el *profiler*, así como en relación a las conclusiones finales establecidas en el capítulo anterior, se sabe que el tiempo de uso de la CPU a máximo rendimiento es cercano al 100%, por lo que, en términos prácticos, el sistema se encuentra ejecutando código paralelo la práctica totalidad del tiempo de ejecución.

Así pues, se pueden hacer los siguientes cálculos. Para la paralelización mediante OpenMP, el consumo de energía es de 95 Watios x (88,63 + 1372,82) segundos = **138837,75 Julios (0,004338 €)**.

A su vez, haciendo referencia al rendimiento obtenido mediante OpenMP y CUDA, utilizando para el cálculo los tiempos en referencia al procesamiento de la función Adicional mediante OpenMP, y al procesamiento de la función Externa mediante CUDA, se obtiene un consumo de energía de 95 Watios x 88,63 segundos + 110 Watios x 22974,29 segundos = 8419,85 Julios + 2527171,9 Julios = **2535591,75 Julios (0,079237 €)**.

El consumo de energía es muy superior en la prueba en la que se utiliza CUDA, debido al notable tiempo de ejecución asociado a su procesamiento, el cual penaliza el uso de los recursos. Así, el rendimiento mediante el uso únicamente de OpenMP es 18,26 veces mejor que su homólogo en CUDA.

**b)** **Ordenador Estudiante.** Este sistema se encuentra limitado al uso de tres núcleos, de los cuatro reales disponibles, accesibles para procesar software paralelo mediante OpenMP. En este caso, el consumo máximo de la CPU, a máximo rendimiento, es de un total de 95 Watios. Dado que se están utilizando únicamente tres de los cuatro núcleos, el consumo efectivo a considerar será la proporción, es decir, 71,25 Watios.

De nuevo, haciendo los cálculos, se dispone de un consumo de energía de 71,25 Watios x (147,22 + 2138,27) segundos = **162841,16 Julios (0,005088 €)**.

**c)** **Servidor Beta.** Este sistema permite la ejecución de software paralelizado mediante OpenMP en hasta ocho núcleos de procesamiento. Estos núcleos consumen, a máximo rendimiento, un total de 80 Watios.

Por lo tanto, repitiendo los cálculos, se tiene un consumo de energía de 80 Watios x (47,98 + 3746,12) segundos = **303528 Julios (0,009485 €)**.

Observando los resultados obtenidos, se puede determinar que el mejor rendimiento, en términos energéticos, es el logrado mediante el procesamiento del software paralelizado a través de OpenMP sobre el sistema denominado como Plataforma TFG, con un consumo de **138837,75 Julios**.

Asimismo, este sistema es el que mejor rendimiento ofrece por unidad de tiempo, por lo que, en conclusión, es el mejor de los sistemas contemplados, logrando un procesamiento del software paralelo en un tiempo de **1462,04 segundos**.

Para terminar, tomando finalmente este sistema como referencia, el consumo energético de la versión inicial del software es de 95 Watios x (1134,24 + 11387,37) segundos = 1189552,95 Julios (0,037173 €). El consumo de energía ha sido optimizado en un factor de **8,5679** veces sobre el rendimiento inicial.

A su vez, el tiempo de ejecución de la primera versión del software se encuentra en 12521,61 segundos. Un tiempo que ha sido mejorado en un factor de **8,5644** veces sobre este tiempo original.

## Apéndice 2. Análisis del efecto optimizador del compilador.

Desde el comienzo del desarrollo de este trabajo se ha implementado una mejora en el código software mediante alguna de las balizas que el compilador pone a disposición del desarrollador para este fin.

Sin embargo, según se ha avanzado en el proceso iterativo de desarrollo, se ha notado que el efecto de las balizas ha podido enmascarar el rendimiento conseguido en los desarrollos finales de OpenMP, sobre los que efectúa optimizaciones.

Con el fin de conocer exactamente el rendimiento de cada una de las soluciones implementadas, y la diferencia entre utilizar o no dichas balizas, se ha elaborado una batería de pruebas persiguiendo especialmente este objetivo. A través de los resultados se han podido elaborar las siguientes tablas.

La Tabla 22 contiene una referencia a los tiempos obtenidos, para el software en su versión original, a través de cada uno de los sistemas disponibles.

| | | Serie (sin optimizar) | | |
|---|---|---|---|---|
| | | TFG | Beta | Estudiante |
| **Adicional** | **Real time** | 1134,24 | 1590,92 | 919,35 |
| **Externa** | **Real time** | 11387,37 | 15145,25 | 10167,27 |
| | **Page Faults** | 779,00 | - | 55208,00 |

*Tabla 22: Representación de los tiempos de ejecución en serie del software proveído en cada sistema disponible.*

La tabla 23 contiene la obtención de los tiempos de ejecución correspondientes al máximo número de hilos disponible, dependiendo de cada sistema, para cada una de los desarrollos realizados a través del modelo que proporciona OpenMP. Asimismo, tales tiempos de ejecución se han separado teniendo en cuenta si el software había sido o no compilado mediante la baliza de optimización.

| | | OpenMP | | | OpenMP (Optimizado) | | |
|---|---|---|---|---|---|---|---|
| | | TFG (8) | Beta (8) | Estudiante (3) | TFG (8) | Beta (8) | Estudiante (3) |
| **Adicional** | **Real time** | 378,96 | 185,39 | 316,07 | 88,63 | 47,98 | 147,22 |
| | **Speedup** | **2,99** | **8,58** | **2,91** | **12,80** | **33,16** | **6,24** |
| **Externa** | **Real time** | 4719,86 | 4254,63 | 4944,58 | 1399,99 | 4155,77 | 2709,58 |
| | **Speedup** | **2,41** | **2,68** | **2,06** | **8,13** | **3,64** | **3,75** |
| | **Page Faults** | 1494,00 | - | 1472,00 | 1840,00 | - | 1816,00 |
| **Interna** | **Real time** | 4971,05 | 6071,71 | 5305,15 | 2216,22 | 5577,31 | 3259,86 |
| | **Speedup** | **2,29** | **2,49** | **1,92** | **5,14** | **2,72** | **3,12** |
| | **Page Faults** | 1487,00 | - | 1473,00 | 1933254,00 | - | 1822,00 |
| **Unificada** | **Real time** | 2758,54 | 3749,23 | 3024,27 | 1372,82 | 3746,12 | 2138,27 |
| | **Speedup** | **4,13** | **4,04** | **3,36** | **8,29** | **4,04** | **4,75** |
| | **Page Faults** | 1241310,00 | - | 37939,00 | 1262507,00 | - | 101805,00 |

*Tabla 23: Representación de los tiempos, aceleraciones y fallos de páginas obtenidos tras las pruebas.*

Analizando esta última tabla se pueden establecer varias conclusiones. En primer lugar, se puede observar que, en todos los casos, el compilador realiza un trabajo correcto de optimización, generando tiempos de ejecución menores, lo que indica que este último no interfiere ni en el código desarrollado, ni en el modelo utilizado.

En algunos de los casos, la mejora establecida por el compilador es drástica, consiguiendo cotas que, de manera lineal, serían imposibles de obtener mediante el uso de los recursos físicamente disponibles. Así, se consigue una aceleración de 12,80, y de 33,16, para las ejecuciones de la función Adicional en la Plataforma TFG y en el Servidor Beta, respectivamente. Sin el soporte del compilador, sin embargo, el rendimiento obtenido es de 2,99 y 8,58, correspondientemente.

Sin embargo, según avanza el proceso iterativo llevado a cabo en el desarrollo de soluciones, se puede observar que la ganancia adicional obtenida por la optimización del compilador decrece, en relación a la no optimizada por este último. Algunos ejemplos pueden verse en el rendimiento de la función Unificada para el Servidor Beta o, con algo de distancia, en el rendimiento de esta misma función en el Ordenador Estudiante.

La finalidad de este análisis, sin embargo, es comprobar el acercamiento de los desarrollos implementados a los sistemas de experimentación.

Así, se puede ver que el rendimiento de la función Unificada es de 4,13, 4,04 y 3,36 para la Plataforma TFG, el Servidor Beta y el Ordenador Estudiante, respectivamente.

Tales aceleraciones, si bien se encuentran alejadas de las buscadas, a excepción de la obtenida para el Ordenador Estudiante, representan una ganancia más significativa que la establecida en el correspondiente apartado de desarrollo, debido a que se tomó como referencia el tiempo original optimizado por el compilador. En dicho apartado, la máxima ganancia obtenida fue de 2,86 en la Plataforma TFG.

En este punto se sabe que actualmente la máxima ganancia obtenida para dicha función es de 4,13, y de 8,29 si el código se compila con la baliza de optimización, en referencia, en ambos casos, al tiempo obtenido mediante el código original.

Este último análisis no invalida los resultados descritos durante el trabajo, debido a que el objetivo final de la optimización del software es la mejora del tiempo de ejecución, motivo por el cual se realizaron los cálculos de esa forma.

Sin embargo, mediante esta última tabla de resultados se puede verificar la ganancia de rendimiento debida únicamente al modelo de paralelismo utilizado, para cada uno de los sistemas, aclarando así el efecto optimizador en la compilación del código. De manera gráfica, en la Figura 21 puede observarse la distribución de los resultados obtenidos.

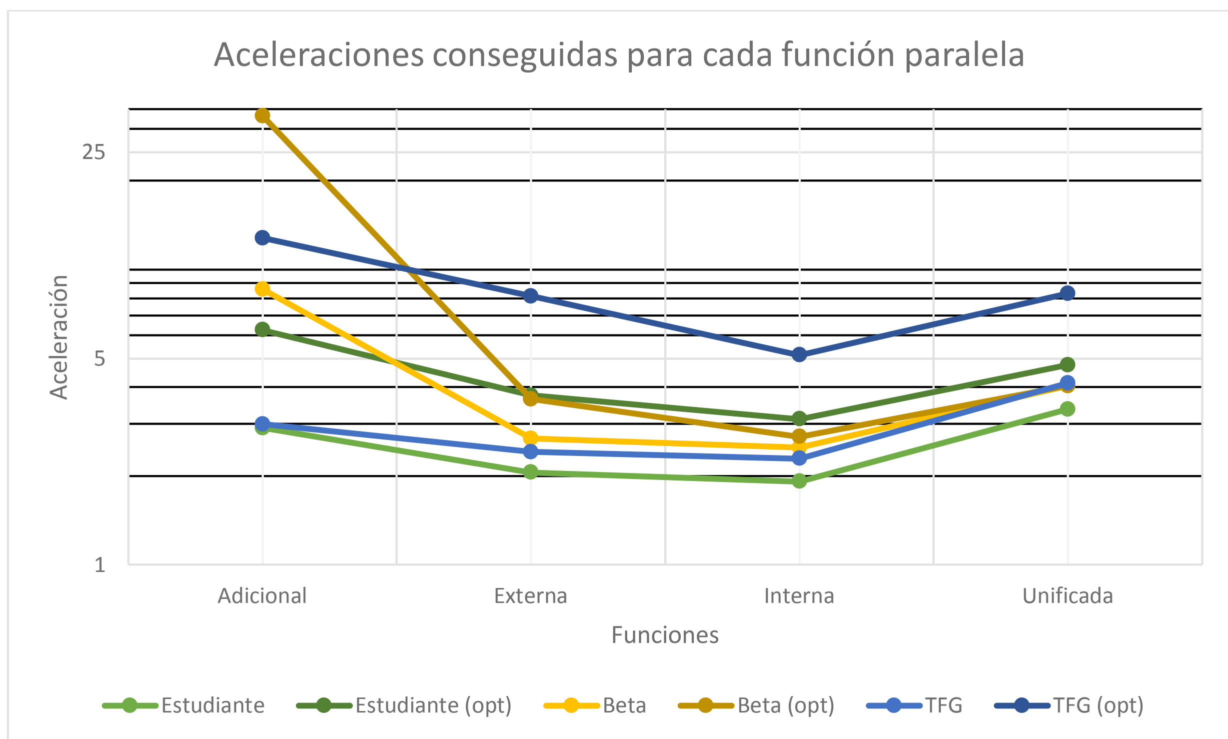


*Figura 21: Representación gráfica de las aceleraciones conseguidas para cada combinación disponible*

# Bibliografía


- Aldea, Sergio, Fresno, Javier. *Programación de sistemas de memoria compartida: OpenMP*. No disponible de forma libre en Internet.
- Arora, Himanshu. *GPROF Tutorial – How to use Linux GNU GCC Profiling Tool*. http://www.thegeekstuff.com/2012/08/gprof-tutorial/
- Barney, Blaise. *OpenMP*. https://computing.llnl.gov/tutorials/openMP/
- Dahnken, Christopher, Klemm, Michael, Semin, Andrey y Supalov, Alexander. *Optimizing HPC Applications with Intel(R) Cluster Tools*. http://pdf.th7.cn/down/files/1508/Optimizing%20HPC%20Applications%20with%20Intel%20Cluster%20Tools.pdf
- gcc.gnu. *Options That Control Optimization*. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- Godard, Sebastien. *sadc(8) - Linux man page*. http://linux.die.net/man/8/sadc
- Harris, Mark. *Optimizing Parallel Reduction in CUDA*. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf
- Intel. *Intel Guide For Developing Multithreaded Applications*. http://www.intel.com/software/threading-guide
- Lahey/GNU Fortran. *Optimization Level*. http://www.lahey.com/docs/lgf10help/LFCOo0o1.htm
- Matloff, Norm. *Programming on Parallel Machines*. http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf
- Mattson, Tim, Meadows, Larry. *A "Hands-on" Introduction to OpenMP*. http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
- Microsoft. *Reduction*. https://msdn.microsoft.com/es-es/library/88b1k8y5.aspx
- NVidia. *CUDA Quick Start Guide*. http://developer.download.nvidia.com/compute/cuda/7.5/Prod/docs/sidebar/CUDA_Quick_Start_Guide.pdf
- NVidia CUDA Toolkit Documentation. *NVIDIA CUDA Getting Started Guide for Linux*. http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/#axzz4BNYGrybp
- NVidia CUDA ToolKit Documentation. *CUDA-MEMCHECK*. http://docs.nvidia.com/cuda/cuda-memcheck/#axzz4BNYGrybp
- NVidia. *Procesamiento paralelo CUDA*. http://www.nvidia.es/object/cuda-parallel-computing-es.html
- OpenMP API User's Guide. *Nested Parallelism*. https://docs.oracle.com/cd/E19059-01/stud.10/819-0501/2_nested.html
- Ortega, Héctor, Torres, Yuri. *Programación de GP-GPU: CUDA*. No disponible de forma libre en Internet.
- Red Eléctrica de España. Término de facturación de energía activa del PVPC. https://www.esios.ree.es/es/pvpc
- Supercomputing Blog, The. Tutorial – *Parallel For Loops with OpenMP*. http://supercomputingblog.com/openmp/tutorial-parallel-for-loops-with-openmp/
- Vandenbout, Dave. *Threads and blocks and grids, oh my!* https://llpanorama.wordpress.com/2008/06/11/threads-and-blocks-and-grids-oh-my/
- Wikipedia. *OpenMP*. https://en.wikipedia.org/wiki/OpenMP

- Yliluoma, Joel. *Guide into OpenMP: Easy multithreading programming for C++*. http://bisqwit.iki.fi/story/howto/openmp/
- Zahran, Mohamed. *Lecture 5: CUDA Threads*. http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture5.pdf
- Zahran, Mohamed. *Lecture 6: CUDA Memories*. http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture6.pdf

# Anexos

## 1) Distribución, compilación y ejecución de contenidos del soporte digital.

El soporte digital entregado dispone de los siguientes contenidos, desarrollados durante el transcurso de este Trabajo Fin de Grado.

- Una copia de esta memoria, denominada con el nombre ***memoria.pdf***.
- Una carpeta, denominada con el nombre ***codigos***, con el software desarrollado en versión fuente. Dentro de esta carpeta se encuentran los siguientes ficheros de código:
  - ***elasticEnergy_cuda.cu***: contiene la versión paralela final del software, desarrollada utilizando la arquitectura CUDA para la función Interna, y utilizando el modelo OpenMP para la función Adicional.

    Asimismo, se mantiene la función de contabilización del tiempo y las balizas en referencia a cada una de las funciones a optimizar. Esta función y estas balizas se mantienen adicionalmente en los demás códigos paralelizados.

    Se compila mediante: *nvcc elasticEnergy_cuda.cu*.
  - ***elasticEnergy_cuda_omp.cu***: contiene la versión paralela final del software, desarrollada utilizando, de manera combinada, CUDA y OpenMP.

    Se compila mediante: *nvcc —compiler-options "–fopenmp –Ofast" elasticEnergy_cuda_omp.cu*.
  - ***elasticEnergy_omp_unificado_final.c***: contiene la versión paralela final del software, desarrollada utilizando el modelo OpenMP.

    Se compila mediante: *gcc –fopenmp –Ofast elasticEnergy_omp_unificado_final.c*.
  - ***elasticEnergy_serie.c***: contiene la versión en serie original del software proveído inicialmente.

    Se compila mediante: *gcc elasticEnergy_serie.c*.
- Una carpeta, denominada con el nombre ***ficheros_prueba***, con los ficheros de ejemplo utilizados y requeridos para poder ejecutar las versiones desarrolladas.

  Todas se ejecutan, tras el proceso de compilación descrito, mediante:

  *./a.out CeldaBackground CeldaDefecto-A CeldaDefecto-B RedPerfecta*

## 2) Salida completa del análisis de tiempos de ejecución en el Ordenador Estudiante.

### Bloque 1: Inicialización (0s -> 3.550505s).

Lectura de las posiciones de la Red Perfecta ...
**Tiempo transcurrido en 20a serie de bucles: 0.348533**
Lectura de las posiciones de la Celda de Background ...
**Tiempo transcurrido en 21a serie de bucles: 1.403675**
Lectura de las posiciones de la Celda del Defecto en la subred A ...
**Tiempo transcurrido en 22a serie de bucles: 2.465645**
Lectura de las posiciones de la Celda del Defecto en la subred B ...
**Tiempo transcurrido en 23a serie de bucles: 3.529457**
**Tiempo transcurrido en 24a serie de bucles: 3.550505**

### Bloque 2a: Análisis DAES en la Celda de Background... (3.550505s -> 797.426179s)

... dentro de la rutina de analisisDAES ...
... asignación de cubos a las posiciones de la red perfecta ... 405264
**Tiempo transcurrido en 1a serie de bucles: 0.378277**
**Tiempo transcurrido en 2a serie de bucles: 0.402282**
... posición de red más cercana ... 405264
**Tiempo transcurrido en 3a serie de bucles: 0.402297 - 0.402294 = 0.000003 x 405263 veces.**
**Tiempo transcurrido en 3a serie de bucles: 1.253648 (último). Tiempo de los bucles 3 = 0.851357s.**
**Tiempo transcurrido en 4a serie de bucles: 1.253649**
... vecinos de átomos ...
**Tiempo transcurrido en 5a serie de bucles: 1.264705**
**Tiempo transcurrido en 6a serie de bucles: 1.279609**
**Tiempo transcurrido en 7a serie de bucles: 1.290825**
**Tiempo transcurrido en 8a serie de bucles: 1.290852 - 1.290848 = 0.000034 x 405263 veces.**
**Tiempo transcurrido en 8a serie de bucles: 2.026458 (último). Tiempo de los bucles 8 = 0.735644s.**
**Tiempo transcurrido en 9a serie de bucles: 2.026459**
**Tiempo transcurrido en 10a serie de bucles: 2.033402**
**Tiempo transcurrido en 11a serie de bucles: 2.034368**
**Tiempo transcurrido en 12a serie de bucles: 2.045671**
... posición de red más cercana a los DA ...
**Tiempo transcurrido en 13a serie de bucles: 793.539101**
... posición de red más cercana a los ES ...
**Tiempo transcurrido en 14a serie de bucles: 793.846772**
**Tiempo transcurrido para AnalisisDAES: 793.846791**
**Tiempo transcurrido en 26a serie de bucles: 797.411175**
**Tiempo transcurrido en 27a serie de bucles: 797.426179**

**Bloque 2b: Análisis DAES en la Celda del Defecto en la subred A... (797.426179s -> 1571.602866s)**

... dentro de la rutina de analisisDAES ...
... asignación de cubos a las posiciones de la red perfecta ... 405225
**Tiempo transcurrido en 1a serie de bucles: 0.333827**
**Tiempo transcurrido en 2a serie de bucles: 0.355977**
... posición de red más cercana ... 405225
**Tiempo transcurrido en 3a serie de bucles: 0.355991 - 0.355989 = 0.000002 x 405224 veces.**
**Tiempo transcurrido en 3a serie de bucles: 1.187962 (último). Tiempo de los bucles 3 = 0.831975s.**
**Tiempo transcurrido en 4a serie de bucles: 1.187963**
... vecinos de átomos ...
**Tiempo transcurrido en 5a serie de bucles: 1.197177**
**Tiempo transcurrido en 6a serie de bucles: 1.211397**
**Tiempo transcurrido en 7a serie de bucles: 1.222105**
**Tiempo transcurrido en 8a serie de bucles: 1.222183 - 1.222149 = 0.000034 x 405224 veces.**
**Tiempo transcurrido en 8a serie de bucles: 1.940201 (último). Tiempo de los bucles 8 = 0.718086s.**
**Tiempo transcurrido en 9a serie de bucles: 1.940202**
**Tiempo transcurrido en 10a serie de bucles: 1.946918**
**Tiempo transcurrido en 11a serie de bucles: 1.947502**
**Tiempo transcurrido en 12a serie de bucles: 1.959252**
... posición de red más cercana a los DA ...
**Tiempo transcurrido en 13a serie de bucles: 774.148673**
... posición de red más cercana a los ES ...
**Tiempo transcurrido en 14a serie de bucles: 774.149616**
**Tiempo transcurrido para AnalisisDAES: 774.149653**
**Tiempo transcurrido en 28a serie de bucles: 1571.589825**
**Tiempo transcurrido en 29a serie de bucles: 1571.602866**

**Bloque 2c: Análisis DAES en la Celda del Defecto en la subred B... (1571.602866s -> 2341.641819s)**

... dentro de la rutina de analisisDAES ...
... asignación de cubos a las posiciones de la red perfecta ... 405225
**Tiempo transcurrido en 1a serie de bucles: 0.330441**
**Tiempo transcurrido en 2a serie de bucles: 0.351718**
... posición de red más cercana ... 405225
**Tiempo transcurrido en 3a serie de bucles: 0.351780 - 0.351755 = 0.000025 x 405224 veces.**
**Tiempo transcurrido en 3a serie de bucles: 1.172994 (ultimo) Duración de los bucles 3 = 0.821264s.**
**Tiempo transcurrido en 4a serie de bucles: 1.172995**
... vecinos de átomos ...
**Tiempo transcurrido en 5a serie de bucles: 1.182233**
**Tiempo transcurrido en 6a serie de bucles: 1.196959**
**Tiempo transcurrido en 7a serie de bucles: 1.207947**
**Tiempo transcurrido en 8a serie de bucles: (1.208035 - 1.207997) = 0.000038 x 405224 veces.**
**Tiempo transcurrido en 8a serie de bucles: 1.924068 (último). Tiempo de los bucles 8 = 0.716109s.**
**Tiempo transcurrido en 9a serie de bucles: 1.924069**
**Tiempo transcurrido en 10a serie de bucles: 1.930290**
**Tiempo transcurrido en 11a serie de bucles: 1.931871**
**Tiempo transcurrido en 12a serie de bucles: 1.942872**
... posición de red más cercana a los DA ...
**Tiempo transcurrido en 13a serie de bucles: 769.934163**
... posición de red más cercana a los ES ...
**Tiempo transcurrido en 14a serie de bucles: 769.934749**
**Tiempo transcurrido para AnalisisDAES: 769.934798**
**Tiempo transcurrido en 30a serie de bucles: 2341.538698**
**Tiempo transcurrido en 31a serie de bucles: 2341.539702**
CDM defecto A: 3.394990 3.395010 3.395000
CDM defecto B: 4.753000 4.753000 4.753000
**Tiempo transcurrido en 32a serie de bucles: 2341.641819**

**Bloque 3a: Repartir el strain de la Celda de Background... (2341.641819s -> 2341.722326s)**

> **Tiempo transcurrido en 15a serie de bucles: 0.023238**
> **Tiempo transcurrido en 16+17a serie de bucles: (0.023622+0.023630) - (0.023600+0.023612) x 311 veces.**
> **Tiempo transcurrido en 16a serie de bucles: 0.024418**
> **Tiempo transcurrido en 17a serie de bucles: 0.024419**
> **Tiempo transcurrido en 18a serie de bucles: 0.024774**
> **Tiempo transcurrido en 19a serie de bucles: 0.025580**
> **Tiempo transcurrido para repartirStrainRP: 0.055024**
> **Tiempo transcurrido en 33a serie de bucles: 2341.722326**

**Bloque 3b: Repartir el strain de la Celda del Defecto en la subred A... (2341.722326s -> 2341.803470s)**

> **Tiempo transcurrido en 15a serie de bucles: 0.023333**
> **Tiempo transcurrido en 18a serie de bucles: 0.024187**
> **Tiempo transcurrido en 19a serie de bucles: 0.024993**
> **Tiempo transcurrido para repartirStrainRP: 0.054659**
> **Tiempo transcurrido en 34a serie de bucles: 2341.803470**

**Bloque 3c: Repartir el strain de la Celda del Defecto en la subred B... (2341.803470s -> 2341.858132s)**

> **Tiempo transcurrido en 15a serie de bucles: 0.022683**
> **Tiempo transcurrido en 18a serie de bucles: 0.023572**
> **Tiempo transcurrido en 19a serie de bucles: 0.024390**
> **Tiempo transcurrido para repartirStrainRP: 0.054662**

**Bloque 4: Calculo de la energía elástica... (2341.858132s -> 12459.690251s)**

> **Tiempo transcurrido para calcEnerEl: 0.024024 - 0.023510 x 405223 veces.**
> **Tiempo transcurrido para calcEnerEl: 0.024316 (último).**
> **Tiempo transcurrido en 35a serie de bucles: 12459.690251 (movimiento de celda en celda)**
> **Tiempo transcurrido del programa completo: 12459.690319**

**3) Tabla completa del uso de recursos durante la ejecución del software inicial en el Ordenador Estudiante.**

| Uso de recursos (serie -Ofast) | | | | |
|---|---|---|---|---|
| **Minuto** | **%CPU** | **E/S** | **RAM (MB)** | **Paso** |
| **0 - Reposo** | **0,01** | **0,13** | **0,01** | **0,00** |
| **1 - Inicio - Min RAM** | **87,59** | **18,58** | **749,73** | **749,72** |
| **2** | 100,00 | 0,12 | 751,77 | 2,04 |
| **3** | 100,00 | 1,30 | 755,99 | 4,22 |
| **4** | 100,00 | 0,12 | 757,87 | 1,88 |
| **5** | 100,00 | 0,07 | 759,93 | 2,06 |
| **6** | 99,98 | 0,07 | 788,00 | 28,07 |
| **7** | 100,00 | 0,12 | 788,05 | 0,06 |
| **8** | 100,00 | 0,07 | 790,10 | 2,04 |
| **9** | 100,00 | 0,07 | 792,05 | 1,95 |
| **10** | 100,00 | 0,12 | 792,48 | 0,43 |
| **11** | 100,00 | 0,07 | 818,53 | 26,05 |
| **12** | 100,00 | 0,07 | 818,53 | 0,00 |
| **13** | 100,00 | 1,85 | 819,17 | 0,64 |
| **14** | 100,00 | 0,07 | 819,17 | 0,00 |
| **15** | 100,00 | 0,10 | 819,17 | 0,00 |
| **16** | 100,00 | 0,07 | 819,49 | 0,32 |
| **17** | 99,98 | 0,48 | 959,41 | 139,92 |
| **18** | 100,00 | 0,48 | 959,88 | 0,47 |
| **19** | 100,00 | 0,55 | 960,61 | 0,73 |
| **20** | 100,00 | 0,47 | 961,21 | 0,61 |
| **21** | 100,00 | 0,47 | 963,71 | 2,49 |
| **22 - Max E/S** | **100,00** | **204,37** | **1049,61** | **85,90** |
| **23** | 100,00 | 1,58 | 1050,32 | 0,72 |
| **24** | 100,00 | 0,50 | 1050,81 | 0,48 |
| **25** | 100,00 | 0,50 | 1051,29 | 0,48 |
| **26** | 100,00 | 0,47 | 1051,90 | 0,61 |
| **27** | 100,00 | 0,43 | 1051,90 | 0,00 |
| **28** | 100,00 | 0,43 | 1053,52 | 1,62 |
| **29** | 100,00 | 0,48 | 1053,52 | 0,00 |
| **30** | 100,00 | 0,48 | 1054,73 | 1,21 |
| **31** | 100,00 | 0,57 | 1055,21 | 0,48 |
| **32** | 100,00 | 0,57 | 1055,70 | 0,48 |
| **33** | 100,00 | 0,57 | 1056,18 | 0,48 |
| **34** | 100,00 | 0,57 | 1056,67 | 0,48 |
| **35** | 100,00 | 0,57 | 1057,15 | 0,48 |
| **36** | 100,00 | 0,57 | 1059,59 | 2,44 |
| **37** | 100,00 | 0,68 | 1060,20 | 0,61 |
| **38** | 100,00 | 0,68 | 1060,81 | 0,61 |
| **39** | 100,00 | 0,68 | 1061,41 | 0,61 |
| **40** | 100,00 | 0,68 | 1062,02 | 0,61 |
| **41** | 100,00 | 0,68 | 1062,63 | 0,61 |
| **42** | 100,00 | 0,68 | 1063,23 | 0,61 |
| **43** | 100,00 | 0,68 | 1063,96 | 0,73 |
| **44** | 100,00 | 0,48 | 1064,69 | 0,73 |
| **45** | 100,00 | 0,48 | 1065,43 | 0,73 |
| **46** | 100,00 | 0,48 | 1066,16 | 0,73 |

| **47** | 100,00 | 0,48 | 1066,89 | 0,73 |
|---|---|---|---|---|
| **48** | 100,00 | 0,48 | 1067,58 | 0,69 |
| **49** | 100,00 | 0,47 | 1067,60 | 0,02 |
| **50** | 100,00 | 0,47 | 1067,62 | 0,02 |
| **51** | 100,00 | 0,47 | 1067,63 | 0,02 |
| **52** | 100,00 | 0,47 | 1067,65 | 0,02 |
| **53** | 100,00 | 0,47 | 1070,31 | 2,66 |
| **54** | 100,00 | 0,52 | 1070,64 | 0,33 |
| **55** | 100,00 | 0,52 | 1070,97 | 0,33 |
| **56** | 100,00 | 0,52 | 1071,29 | 0,33 |
| **57** | 100,00 | 0,52 | 1071,62 | 0,33 |
| **58** | 100,00 | 0,52 | 1071,95 | 0,33 |
| **59** | 100,00 | 0,53 | 1074,71 | 2,77 |
| **60** | 100,00 | 0,53 | 1075,53 | 0,81 |
| **61** | 100,00 | 0,53 | 1076,34 | 0,81 |
| **62** | 100,00 | 0,53 | 1076,37 | 0,03 |
| **63** | 100,00 | 0,53 | 1076,60 | 0,23 |
| **64** | 100,00 | 0,43 | 1076,77 | 0,18 |
| **65** | 100,00 | 0,47 | 1077,38 | 0,61 |
| **66 - Max RAM** | **100,00** | **0,57** | **1078,11** | **0,73** |
| **67 - Fin** | **25,07** | **0,22** | **0,88** | **-1077,22** |
| **68 - Reposo** | **0,01** | **0,13** | **0,01** | **-0,87** |
| **69** | 0,01 | 0,28 | 0,26 | 0,25 |

*Figura 22: Resultados de la aplicación del monitor sobre el software en ejecución.*