Operación stencil en plataformas multi-core y many-core

Autor:
Bernardo Garcés Chapero

Director:
José Ramón Herrero Zaragoza
Departamento AC

Codirectora:
Beatriz Otero Calviño
Departamento AC

Titulación:
Ingeniería Informática

Centro:
Facultat d'Informàtica de Barcelona (FIB)

Universidad:
Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

22 de Junio, 2011

Resumen

Los problemas derivados de la disipación de energía en la computación secuencial, están haciendo que cada vez se popularice más el uso de máquinas y sistemas con mayor cantidad de núcleos de proceso. Pasando desde pequeños procesadores con un número reducido de núcleos, por *clusters* con varias máquinas secuenciales distribuidas, e incluso por dispositivos de procesamiento gráfico (GPUs) con varios cientos de núcleos que permiten asignar tareas generales a estos, muchos algoritmos están siendo adaptados a estos modelos de paralelización.

En este trabajo se ha llevado a cabo el análisis, implementación, optimización y paralelización de las operaciones *stencil* de 5 puntos y 27 puntos, que son operaciones cuyo origen es la resolución de ecuaciones en derivadas parciales mediante un método finito y que tienen una importancia significativa en el campo de la ciencia. La paralelización se ha llevado a cabo tanto en un sistema *multi-core* con dos procesadores Intel Xeon E5520, como utilizando un dispositivo gráfico Nvidia GeForce GTX 295 con 240 núcleos CUDA.

En cuanto a la optimización del algoritmo, se han aplicado al cálculo una serie de optimizaciones al código secuencial tales como desenrollado de bucles, eliminación de subexpresiones comunes o vectorizaciones mediante instrucciones SSE.

Para la paralelización en el entorno *multi-core*, y con el fin de poder explotar todos los recursos hardware del sistema, se han probado diferentes modelos de programación paralela, tanto basados en sistemas de memoria distribuida como MPI, como basados en sistemas de memoria compartida como OpenMP y POSIX Threads.

En cuanto a la implementación en el entorno *many-core*, se han utilizado dos formas diferentes de enfocar la resolución del problema, una de ellas empleando el método que a simple vista parece más práctico para la resolución, y la otra creando una estructuración de las tareas más apropiado para la arquitectura utilizada. La diferencia de rendimiento de ambas versiones es de un 20 % en el caso del *stencil* de 5 puntos y de un 80 % en el de 27 puntos, en ambos casos a favor de la segunda versión del algoritmo.

Finalmente, los resultados de las optimizaciones han sido diferentes dependiendo de la operación implementada, para el *stencil* de 5 puntos se ha podido conseguir un *Speed Up* de 5.41x en el entorno *multi-core* y de 15.95x en el entorno *many-core*. Para el *stencil* de 27 puntos, el *Speed Up* ha sido de 13.9x en el entorno *multi-core* y de 146.97x en el entorno *many-core*.

Índice general

1. Introducción **5**
1.1. El modelo de computación secuencial 5
1.2. El modelo de computación paralela 6
1.3. Dispositivos con un gran número de núcleos 7
1.4. Operación *Stencil* 8
1.5. Descripción y objetivos del trabajo 10

2. Operación *Stencil* **12**
2.1. Pseudocódigo de la operación 12
2.2. Características de los *stencils* 13
2.2.1. Dimensionalidad 13
2.2.2. Orden 14
2.2.3. Tamaño 15
2.2.4. Huella 15
2.3. Tipos de iteración 15
2.3.1. Jacobi 15
2.3.2. Gauss-Seidel 16
2.3.3. Gauss-Seidel Red-Black (GSRB) 17
2.4. Fronteras 18
2.4.1. Fronteras constantes 18
2.4.2. Fronteras periódicas 18
2.5. Coeficientes 19
2.5.1. Coeficientes constantes 19
2.5.2. Coeficientes variables 19
2.6. *Stencils* de este trabajo 20
2.6.1. *Stencil* de 5 puntos 20
2.6.2. *Stencil* de 27 puntos 21

3. Recopilación de optimizaciones **23**
3.1. Causas del bajo rendimiento 23

3.2. Descomposición del problema 24
3.3. Bloqueo de iteraciones en el tiempo 25
3.3.1. Sesgado del tiempo 25
3.3.2. Cola circular 26
3.4. Posicionamiento de los datos 29
3.4.1. Datos de relleno 29
3.4.2. Acceso a memoria no uniforme 30
3.5. Transferencia de datos 30
3.5.1. Carga temprana de datos explícita 31
3.5.2. Cache bypass 31
3.6. Optimizaciones en el cálculo 32
3.6.1. Desenrollado de bucles 32
3.6.2. Eliminación de subexpresiones comunes 33
3.6.3. Vectorización 33

4. Entorno de pruebas **35**
4.1. Arquitectura 35
4.2. Modelos de programación paralela 36
4.2.1. OpenMP 36
4.2.2. Pthreads 37
4.2.3. MPI 38
4.2.4. Nvidia CUDA 38
4.3. Lenguajes de programación 39
4.4. Compilador 40

5. Metodología **41**
5.1. Implementaciones 41
5.1.1. Paralelización *multi-core* 41
5.1.2. Paralelización *many-core* 42
5.2. Compilación 43
5.3. Ejecuciones 44
5.4. Tiempo de ejecución 45
5.5. Métricas 46

6. Implementaciones *multi-core* **48**
6.1. Implementación secuencial 48
6.2. Paralelización del problema 48
6.2.1. OpenMP 51
6.2.2. MPI 51
6.2.3. Pthreads 55
6.2.4. Mapeo de hilos a procesadores 56

6.3. Optimizaciones en el cálculo 60
6.3.1. Desenrollado de bucles 60
6.3.2. Eliminación de subexpresiones comunes 60
6.3.3. Vectorización 61
6.3.4. Cache Bypass 62

7. Resultados en plataformas *multi-core* 65
7.1. *Stencil* de 5 puntos 65
7.1.1. Escalabilidad 65
7.1.1.1. Iteraciones 65
7.1.1.2. Tamaño de datos 66
7.1.2. Resultados de las optimizaciones en el código secuencial . 67
7.1.3. Resultados de la paralelización 68
7.1.4. Conclusiones 72
7.2. *Stencil* de 27 puntos 73
7.2.1. Escalabilidad 73
7.2.1.1. Iteraciones 73
7.2.1.2. Tamaño de datos 74
7.2.2. Resultados de las optimizaciones en el código secuencial . 74
7.2.3. Resultados de la paralelización 75
7.2.4. Conclusiones 79

8. Implementaciones *many-core* y resultados 81
8.1. Funcionamiento de CUDA 81
8.2. Lineas generales para la implementación 83
8.2.1. Método directo (DIR) 83
8.2.2. Método solapado (SOL) 87
8.3. Lineas a seguir en las optimizaciones 89
8.3.1. Tamaño del bloque de hilos 89
8.3.2. Acceso a memoria consecutiva 90
8.3.3. Acceso a memoria compartida 90
8.3.4. Eliminación de condicionales y saltos 90
8.3.5. Recursos limitados 91
8.4. Optimizaciones y resultados del *stencil* de 5 puntos 91
8.4.1. Método directo 92
8.4.2. Método solapado 93
8.4.3. Conclusiones 93
8.5. Optimizaciones y resultados del *stencil* de 27 puntos 94
8.5.1. Método directo 95
8.5.2. Método solapado 95

8.5.3. Conclusiones 96

9. Conclusiones **98**
9.1. Las optimizaciones en el código secuencial 98
9.2. Paralelización en entornos *multi-core* 99
9.3. Paralelización en entornos *many-core* 101
9.4. Mejora en el rendimiento global 104

10. Trabajo futuro **106**

11. Planificación del trabajo y presupuesto **108**
11.1. Planificación 108
11.2. Análisis de los costes 108
11.2.1. Costes de personal 108
11.2.2. Costes adicionales 110
11.2.3. Costes totales 111

A. Composición de la matriz **112**

Bibliografía **115**

Capítulo 1

Introducción

1.1. El modelo de computación secuencial

Aumentar el rendimiento secuencial de los procesadores ha sido la tendencia habitual durante gran parte de las últimas décadas. En parte, esto ha sido posible gracias a los avances en el tratamiento del silicio, que mejoraban la velocidad de los transistores. Ciertas técnicas en el diseño de los procesadores se han empleado de igual manera para mantener el modelo de programación secuencial, técnicas que aumentaban la complejidad de los procesadores pero que los hacían energéticamente ineficientes. Estas técnicas incluyen la emisión de múltiples instrucciones, el aumento en la segmentación, la ejecución de instrucciones fuera de orden y la ejecución especulativa [1].

Uno de los problemas derivados del aumento del rendimiento en los procesadores era la dificultad de aumentar de forma proporcional la velocidad de acceso a memoria, con lo que la brecha de rendimiento entre estos dos elementos se volvía cada vez más pronunciada. Al ser imposible proporcionar datos desde la memoria principal al procesador de forma más rápida, se ha optado por mitigar el problema usando una jerarquía con varias memorias cache de menor tamaño y mayor proximidad al núcleo, aprovechando la localidad de los datos que se aprecia en la mayoría de las aplicaciones. Sin embargo, en la mayoría de los casos esta jerarquía de memoria no esta controlada directamente por el programador y es diferente en cada máquina, lo que hace difícil poder escribir código genérico que aproveche dicha jerarquía en todas las máquinas a las que es portado.

Para obtener mayor eficiencia en estos procesadores secuenciales, también se incorporaron instrucciones que aprovechaban el paralelismo de datos cuando la misma operación se debía lanzar contra diferentes datos, este modelo es conocido como SIMD (*Single Instruction Multiple Data*). Estas instrucciones eran

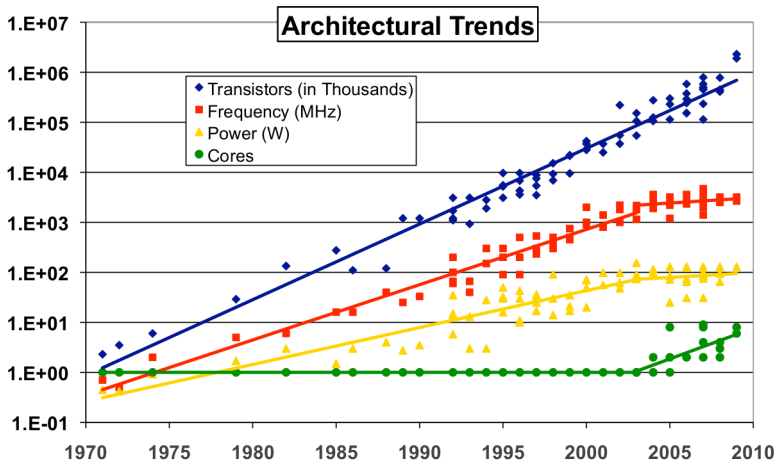

Figura 1.1: Esta figura muestra las tendencias en el número de transistores usados en procesadores, la frecuencia de reloj de los núcleos, la potencia consumida y el número de núcleos por procesador. Puede observarse como en 2004 empezó a aumentar la tendencia a usar procesadores con varios núcleos y que, a su vez, la potencia consumida no aumenta en la misma proporción [3].

específicas en cada arquitectura. En el caso de Intel, se empezó con la extensión de instrucciones MMX para, posteriormente, introducir la extensión SSE. Estas extensiones permitirían aplicar, mediante una sola instrucción, el mismo cálculo a varios elementos en coma flotante consecutivos en memoria. Las contrapartidas del uso de este tipo de instrucciones incluyen la perdida de portabilidad del código y el aumento de la complejidad de este [2].

1.2. El modelo de computación paralela

Tras mucho tiempo y recursos invertidos con el único objetivo de mantener el modelo de computación secuencial, se alcanzó el límite de potencia que un chip era capaz de disipar, por lo que fue necesario dar mayor protagonismo a un modelo de computación diferente, es entonces cuando se empiezan a sustituir los complejos e ineficientes procesadores de un solo núcleo por procesadores con varios núcleos, donde cada núcleo es más simple y con menor consumo energético. Dicha tendencia puede apreciarse en la Figura 1.1.

Sin embargo, el cambio a este nuevo modelo de computación cambió también la forma en que el software debía ser implementado para el total aprovechamiento de los recursos disponibles, cambio que se sigue gestando en la actualidad dada la complejidad de este. Anteriormente, la jerarquía de memoria estaba diseñada para servir peticiones a un solo núcleo, mientras que en este nuevo modelo la memoria puede servir peticiones a más de un núcleo, por lo que es necesaria

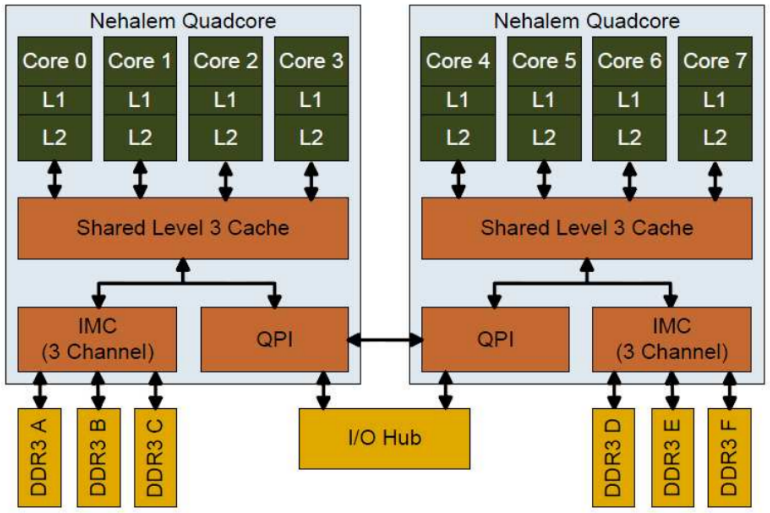

Figura 1.2: Una posible arquitectura NUMA. Multiprocesador Intel Nehalem con dos procesadores y cuatro núcleos por cada uno. Dispone de un mecanismo de interconexión QPI (del inglés QuickPath Interconnect) para servir peticiones de memoria a bancos que no son locales. Dos niveles de memoria cache (L1 y L2) independientes para cada núcleo y uno (L3) compartido por los cuatro [4].

una jerarquía más compleja en la que, por ejemplo, pueda haber una memoria cache para cada uno de los núcleos y otra compartida entre ellos.

También es posible, en el caso de varios procesadores físicos conectados a un mismo nodo, asignar ciertos bancos de memoria principal del nodo a cada uno de estos procesadores, haciendo que los accesos a los bancos locales de cada procesador sean más rápidos. Este tipo de diseño es conocido como acceso a memoria no uniforme (NUMA, Figura 1.2), en contraste con el diseño en el que todos los procesadores de cualquier nodo acceden indistintamente a cualquier posición de memoria (UMA). Los accesos a bancos de memoria no locales serán más lentos, ya que necesitaran de mecanismos de interconexión adicionales que permitan acceder a posiciones de memoria no locales de forma transparente al programador. En el desarrollo de software de alto rendimiento, este factor tendrá que tenerse en cuenta para obtener la máxima velocidad.

1.3. Dispositivos con un gran número de núcleos

Los dispositivos de coprocesamiento como las GPUs (*Graphics Processing Unit*) presentan otra alternativa al modelo de computación secuencial. Este tipo de dispositivos, ideados inicialmente para el coprocesamiento gráfico, disponen de un gran número de núcleos de diseño muy sencillo, que suelen estar orientados

a realizar operaciones de coma flotante. Ha sido en los últimos años cuando los fabricantes han optado por proporcionar herramientas para su uso en aplicaciones de propósito general, técnica que se conoce como GPGPU (*General-Purpose computation on Graphics Processing Units*) [5].

El tipo de paralelismo que utilizan estos dispositivos es conocido como SIMT (*Single Instruction Multiple Thread*), que esta relacionado con el modelo SIMD por el hecho de que ambos intentan explotar el paralelismo de datos. Funciona asignando un hilo de ejecución secuencial a cada uno de los núcleos y ejecutando, en cada uno de los núcleos de un mismo grupo, la misma instrucción a la vez. Cada dispositivo esta pensando para ejecutar paralelamente cientos de hilos a la vez, por lo que la división de datos y la comunicación entre hilos de ejecución juegan un papel aun más importante que en los procesadores de múltiples núcleos mencionados en la sección anterior.

Esta clase de paralelismo es ventajoso en determinados problemas en los que se puede explotar el paralelismo de los datos cuando existen pocas dependencias entre ellos. Normalmente, este paralelismo es explotado en problemas de naturaleza iterativa donde, en cada iteración, cada hilo de ejecución se encarga de una parte de los datos y, al acabar la iteración, se genera una sincronización entre los diferentes hilos para poder empezar la siguiente. Cuando en un problema no existe dependencia alguna entre datos y no es necesario ningún tipo de comunicación entre los diferentes hilos de ejecución, se le conoce por el nombre en inglés de *embarrassingly parallel problem*, y resulta relativamente sencillo desarrollar una implementación para él en este tipo de arquitecturas.

Las GPUs han conseguido durante los últimos años un crecimiento de la productividad y ancho de banda superior al de las CPUs. Si bien esos datos de rendimiento son teóricos y difícilmente alcanzables, su capacidad las convierte en una opción a considerar en el uso de problemas que reúnan las características para ser implementados y debidamente optimizados en este tipo de dispositivos. Por ejemplo, la GPU NVIDIA Tesla C1060 contiene un total de 30 multiprocesadores con 8 SM (*Streamming Processors*) en cada uno de ellos, teniendo pues un total de 240 núcleos preparados para ejecutar código de forma paralela. Dispone de 4 GB de memoria, con un ancho de banda máximo de 102 GB/s. Su pico de productividad es de 1 TeraFLOP [6]. Los gráficos de la Figura 1.3 muestran la tendencia de la productividad y el ancho de banda de las últimas generaciones de GPUs NVIDIA ante las CPUs convencionales.

1.4. Operación *Stencil*

Diferentes problemas en el campo de la ciencia e ingeniería pueden ser implementados eficientemente de forma paralela introduciendo ciertos cambios en

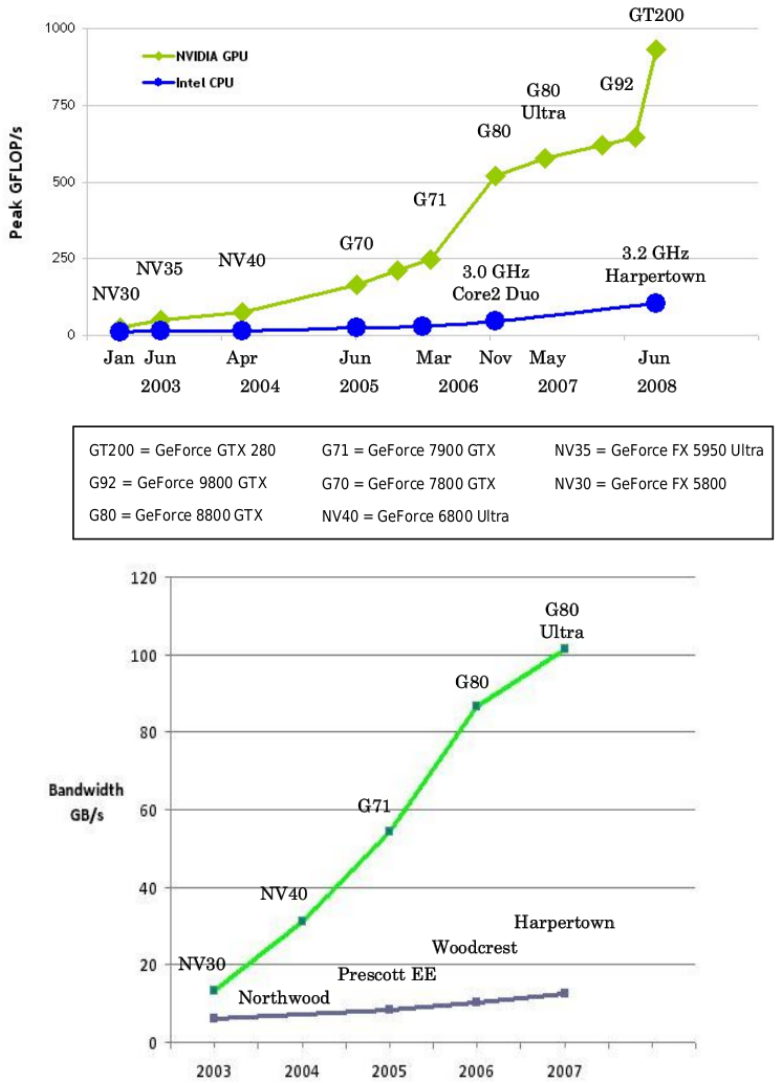

Figura 1.3: El gráfico superior muestra la evolución de la productividad en millones de operaciones de coma flotante por segundo. El gráfico inferior muestra la evolución del ancho de banda [7].

la versión secuencial del algoritmo. La operación *stencil* es uno de ellos.

En primer lugar, conviene aclarar que es una operación *stencil* (o un *stencil* para abreviar) y que utilidad tiene para ser objeto de interés en el campo de la computación paralela.

Diferentes aplicaciones en una amplia variedad de campos de la ciencia e ingeniería requieren para su funcionamiento la resolución de ecuaciones en derivadas parciales (abreviado como EDP o PDE). Este tipo de problemas pueden ser resueltos mediante el método de las diferencias finitas, que calculan una solución de forma aproximada. Frecuentemente, este tipo de operaciones implican el acceso a grandes volúmenes de datos siguiendo patrones regulares. Estos patrones son los que se conocen como operaciones *stencil* o plantilla, y se llevan a cabo accediendo a la matriz donde se guardan los datos y actualizando cada uno de los elementos con valores correctamente ponderados de los elementos vecinos. Los coeficientes usados para ponderar los valores de los diferentes elementos vecinos son diferentes según el problema.

Esta operación sobre la matriz se aplica durante un número finito de veces o unidades de tiempo, ya sea buscando la convergencia de los datos o con la intención de analizar el estado de la matriz tras un cierto número de iteraciones.

1.5. Descripción y objetivos del trabajo

Los objetivos de este trabajo y los pasos necesarios para llevarlos a cabo son los siguientes:

- Desarrollar un estudio del estado del arte de la operación *stencil*.
    - Recolectar y estudiar la bibliografía disponible en este campo.
    - Estudiar las propiedades de los *stencils*.
    - Estudiar las optimizaciones que pueden aplicarse a los *stencils*.
    - Entender como afectan las propiedades del *stencil* en el rendimiento y qué optimizaciones son aplicables.
- Evaluar la implementación de *stencils* en diferentes arquitecturas.
    - Decidir qué tipos de *stencils* resultan interesantes de implementar.
    - Analizar el hardware y software disponible.
    - Evaluar los modelos de programación paralela que van a utilizarse.
    - Especificar qué metodología se llevara a cabo en los experimentos.
    - Especificar qué métricas de rendimiento serán utilizadas.

- Implementar y probar los *stencils*.
  - Analizar cada *stencil* de forma individual.
  - Implementar dicho *stencil* con los modelos de programación paralela establecidos.
  - Evaluar e implementar las optimizaciones software para mejorar el rendimiento.
- Obtener conclusiones sobre su rendimiento
  - Analizar los resultados obtenidos.
  - Identificar qué factores afectan al rendimiento.
  - Proponer mejoras futuras.

Capítulo 2

Operación *Stencil*

Las diferentes operaciones con *stencils* muestran un amplio abanico de propiedades. A continuación se describe como es la operación *stencil* y se muestran algunas de sus propiedades. Al final del capítulo se describen los *stencils* seleccionados para este trabajo: el *stencil* de 5 puntos y el *stencil* de 27 puntos.

2.1. Pseudocódigo de la operación

Para mostrar el cálculo, es preciso definir en primer lugar la nomenclatura que se empleará:

- M es el conjunto de elementos discretos dentro de la matriz.
- T es el número total de iteraciones.
- $v_t(e)$ representa el valor del elemento e en la iteración t.
- $vecinos(e)$ es el conjunto de elementos que están involucrados en el calculo del elemento e, el propio elemento e está usualmente incluido en el conjunto.
- $f(v_t(e')|e' \in vecinos(e))$ es la función que lleva a cabo la operación *stencil* a partir del conjunto de elementos descrito.

El pseudocódigo para el computo del algoritmo es el siguiente [8, 2]:

```
for t=0 to T
  for ∀e < M
    v(t+1)(e)=f({vt(e')|e'∈vecinos(e)})
```

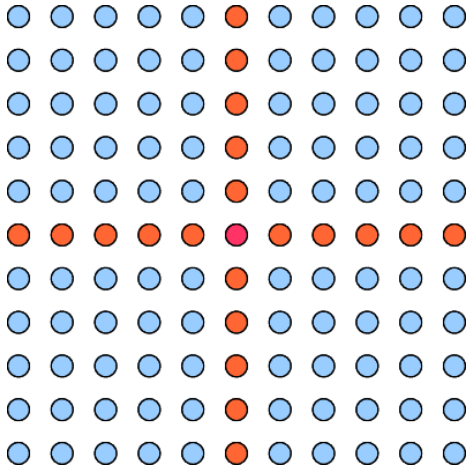

Figura 2.1: En la figura puede observarse un *stencil* bidimensional de 5º orden, tamaño 21 y huella 11^2.

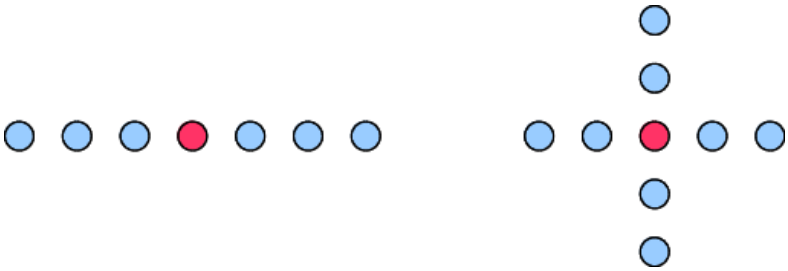

Figura 2.2: A la izquierda, la representación de un *stencil* unidimensional de 7 puntos. A la derecha, la de un *stencil* bidimensional de 9 puntos.

2.2. Características de los *stencils*

Los *stencils* tienen ciertas características básicas que pueden ser empleadas para definirlos [8, 2]. En la Figura 2.1 puede observarse un ejemplo de *stencil* en el que se especifica cuales son sus características básicas y, a continuación, una descripción de cada una de ellas.

2.2.1. Dimensionalidad

La dimensionalidad de un *stencil* depende del número de dimensiones sobre el que se extienda el cálculo de este. En la Figura 2.2 pueden apreciarse dos operaciones *stencil* con diferente dimensionalidad.

La dimensionalidad del *stencil* tiene un fuerte impacto en las optimizaciones que se aplican a este. En un *stencil* unidimensional donde los elementos están almacenados de forma adyacente en memoria, la reutilización de los datos será más sencilla dado que la distancia entre el primer y último elemento usados

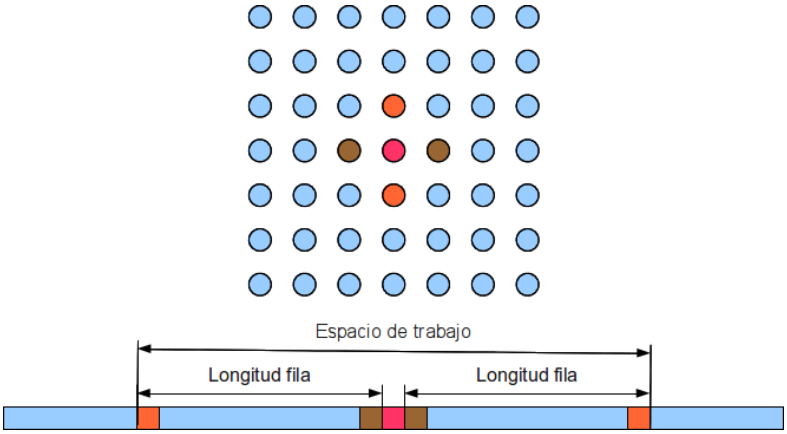

Figura 2.3: La imagen superior muestra una operación *stencil* sobre un elemento situado dentro de una matriz bidimensional, los elementos que participan en la operación han sido pintados con diferentes colores para diferenciarlos. La imagen inferior muestra como dicha matriz queda asignada consecutivamente en memoria usando distribución por filas. Puede observarse como los elementos empleados en el cálculo que están en diferentes filas se encuentran notablemente distanciados del elemento a calcular, haciendo que el espacio de trabajo sobre el que se lleva a cabo la operación sea más grande y existan mayores problemas de localidad.

para cada cálculo será corta. Sin embargo, cuando la dimensionalidad aumenta, también lo hace la distancia entre los elementos usados para cada operación. Por ejemplo, en un *stencil* bidimensional donde los datos se almacenan en una distribución por filas, si el cálculo requiere accesos a elementos en diferentes filas, la distancia entre los elementos accedidos será, en general, notablemente superior a la que se da en el caso de un *stencil* unidimensional. La mayor distancia entre los elementos de un mismo cálculo atenta contra la localidad de los datos y complicará su reutilización, por lo que conviene tener en cuenta este fenómeno a la hora de implementar el algoritmo para obtener un buen rendimiento (Figura 2.3). En el caso de *stencils* de 3 o más dimensiones, la distancia entre elementos teniendo que acceder a diferentes planos se hace aun más grande.

2.2.2. Orden

El orden del *stencil* se define como la distancia máxima entre el punto que se está calculando y el elemento más lejano (en uno de los ejes dimensionales) que se utiliza en el cálculo de este. Esta propiedad es importante cuando se quieren aplicar optimizaciones, ya que a mayor orden, mayor distancia entre los elementos de un mismo cálculo y, por lo tanto, mayor posibilidad de que existan problemas de localidad.

2.2.3. Tamaño

El tamaño del *stencil* está definido como el número de elementos de la matriz que están involucrados en el calculo de una operación *stencil* individual. Esta propiedad tiene también una gran relevancia en la implementación, ya que un mayor tamaño implica un mayor uso de recursos para leer los datos de memoria y, sobre todo, para su posterior almacenamiento en registros, que son un recurso limitado y cuyo uso es importante optimizar.

El tamaño del *stencil* tendrá una relación directa en la intensidad computacional del algoritmo, por lo que las mejoras cuando se lleven a cabo optimizaciones serán más acentuadas. El porqué de este fenómeno es analizado en detalle más adelante.

2.2.4. Huella

La huella del *stencil* se define como el menor volumen que encierra todos los elementos de la matriz involucrados en el cálculo de un elemento. Esta propiedad carece de relevancia cuando se considera cómo implementar el *stencil*, simplemente se utiliza para poder definirlo.

2.3. Tipos de iteración

El tipo de iteración que se lleva a cabo define dónde y en qué orden se actualiza cada uno de los elementos. Hay tres tipos básicos de iteración que son: Jacobi, Gauss-Seidel y Gauss-Seidel Red-Black [3].

2.3.1. Jacobi

Este tipo de iteración es ampliamente usada en diferentes problemas por su sencillez en la implementación y por su potencial capacidad para ser paralelizada. Consiste en llevar a cabo iteraciones en una matriz de lectura y escribiendo el resultado en una matriz de escritura. Al final de cada iteración es necesario intercambiar las matrices, por lo que en la siguiente iteración la matriz de lectura pasará a ser la matriz de escritura y viceversa.

Este modelo hace que sean necesarias por lo menos dos matrices, una de la cual se leen los datos de forma secuencial y otra en la que se escriben los resultados. Siendo necesario el doble de memoria para poder mantener dos copias del volumen de datos, lo que provoca directamente la necesidad de doblar los recursos de almacenamiento y que el ancho de banda disponible quede limitado al tener que realizar el acceso a escritura en una matriz distinta a la original (Figura 2.4).

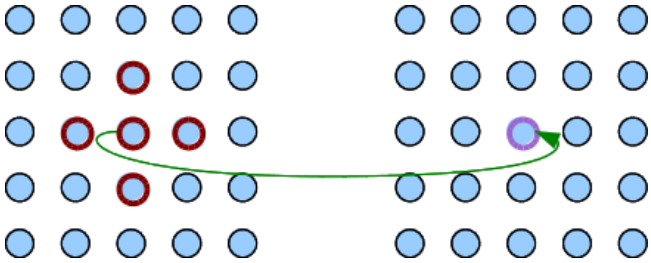

Figura 2.4: A la izquierda, la matriz de lectura desde la cual se lleva a cabo el calculo del *stencil*. A la derecha, la matriz donde se almacena el resultado de la operación. En esta segunda matriz solo interviene el elemento a actualizar.

Sin embargo, y a pesar de las desventajas mencionadas, este tipo de iteración ofrece importantes ventajas en el campo de la optimización, ya que al estar completamente aislados los volúmenes de datos para lecturas y escrituras, el problema puede ser paralelizado de la forma que mejor convenga al programador al no existir dependencias de datos dentro de cada iteración, por lo que cada elemento puede calcularse independientemente del resto.

2.3.2. Gauss-Seidel

Este tipo de iteración lleva a cabo las escrituras en la misma matriz de donde se leen los datos. La ventaja de ello es que los requisitos de memoria se ven drásticamente reducidos, en el caso de solo haber una matriz de lectura, este método requeriría la mitad de memoria que usando Jacobi. De la misma manera también se verá reducido el ancho de banda adicional asociado al método de Jacobi, ya que la escritura se produce en la misma matriz donde se realiza la lectura (Figura 2.5).

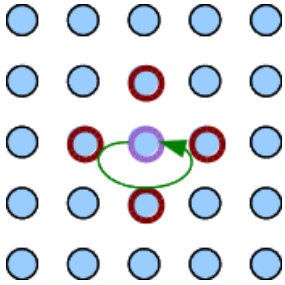

Figura 2.5: El cálculo se lleva a cabo en la misma matriz donde se escribe el resultado, por lo que el tamaño en memoria necesario se reduce directamente a la mitad.

Este tipo de iteración hace que deje de ser posible paralelizar el problema como el programador quiera, ya que los elementos dejan de ser independientes los

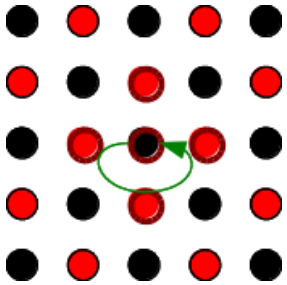

Figura 2.6: En este caso, cada iteración sobre la matriz se divide en dos fases, en la primera se computan los elementos de un color y en la segunda los del otro. Al no haber dependencias entre elementos del mismo color, cada una de estas fases es fácilmente paralelizable.

unos de los otros debido a que algunos elementos serán actualizados antes de que puedan ser consultados por los elementos vecinos para sus cálculos. Es necesario pues establecer un patrón de actualización para que el código sea determinista y se pueda replicar el mismo resultado final en todas las ejecuciones.

2.3.3. Gauss-Seidel Red-Black (GSRB)

Al igual que con Gauss-Seidel, en este tipo de iteración se escribe el resultado del cálculo en la misma matriz de la cual se leen los datos. La diferencia reside en que el patrón de actualización viene dado inicialmente y permite que esta se lleve a cabo en dos partes independientes. Consiste en dividir la matriz en puntos rojos y negros de forma que puntos adyacentes tienen colores diferentes, entonces se actualizan primero los puntos de un color y después los del otro. Dado que los puntos de un mismo color no forman parte del cálculo del resto de puntos de ese color, cada elemento de un color es independiente del resto (Figura 2.6).

Esto es beneficioso ya que ahora es posible, al igual que con Jacobi, paralelizar el volumen de datos como el programador decida, ya que el resultado será siempre el mismo.

A pesar de que los requisitos de memoria son los mismos que en Gauss-Seidel, no es así con el ancho de banda, ya que si se divide la iteración de forma simple calculando primero los puntos de un color y después los del otro, será necesario un tráfico de datos mucho mayor. Sin embargo, es posible aplicar optimizaciones para mitigar esa mayor necesidad de tráfico.

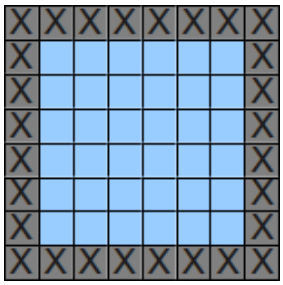

Figura 2.7: Matriz cuyas fronteras tienen un valor fijo que no se actualiza durante el cálculo.

2.4. Fronteras

Otra de las propiedades de los *stencils*, es el tipo de fronteras que tienen, ya que los volúmenes de datos son finitos y por lo tanto hay un limite. Nos encontramos fronteras constantes y periódicas [3].

2.4.1. Fronteras constantes

En este tipo de fronteras, consideramos que nos encontramos en el final del volumen de datos y que los elementos fronterizos no pueden actualizarse.

Esta situación puede simularse de dos formas distintas, la primera de ellas sería creando un halo (o celdas fantasma) alrededor del volumen de datos (el tamaño dependerá del orden del *stencil*) cuyos elementos no se actualicen con el paso del tiempo y mantengan valores fijos, que en cualquier caso pueden ser diferentes los unos de los otros (Figura 2.7).

En caso de que todos los elementos del halo tengan el mismo valor, puede simularse con una simple constante en lugar de emplear una cantidad no despreciable de memoria para simularlo. Sin embargo, está solución requiere modificar el código para que tenga en cuenta cuando un elemento se encuentra en una determinada frontera. Esta solución es necesaria cuando los recursos de memoria son muy limitados y se busca eficiencia en este aspecto.

Cabe destacar que si se subdivide el problema en distintos volúmenes de datos que se calculan por separado en diferentes espacios de memoria, será necesario el uso de halos para cada uno de estos volúmenes, por lo que los requisitos de memoria serán si cabe más elevados.

2.4.2. Fronteras periódicas

Este tipo de fronteras se utilizan cuando la frontera de uno de los lados de la matriz coincide con la del otro. En este caso es posible leer directamente el contenido del otro lado de la matriz, aunque también se puede emplear un halo

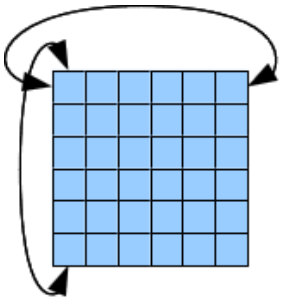

Figura 2.8: Matriz cuyas fronteras periódicas están conectadas entre si, por lo que no existen fronteras con un valor fijo.

como en el caso de las fronteras constantes y actualizar este valor al final de cada iteración, todo depende de la cantidad de memoria que se encuentre disponible para emplear (Figura 2.8).

2.5. Coeficientes

Los coeficientes son los valores mediante los cuales se pondera el peso de cada uno de los elementos sobre otro. Estos pueden ser constantes o variables [3].

2.5.1. Coeficientes constantes

Se da este caso cuando, para todos los elementos, se emplean los mismos valores para ponderar sus vecinos. Sin embargo, estos valores pueden ser diferentes según la distancia del elemento vecino respecto al elemento que esta siendo calculado o de su posición relativa.

Esto hace innecesario tener que leer de memoria los coeficientes en cada cálculo, lo que conlleva reducciones en los requisitos de memoria y en el ancho de banda. También hace posible, al tratarse de valores constantes, que el propio compilador pueda aplicar automáticamente ciertas optimizaciones.

2.5.2. Coeficientes variables

Es posible que no se den las idóneas circunstancias del caso de los coeficientes constantes y que, dependiendo del elemento de la matriz que se esté calculando, los valores de los elementos vecinos tengan un peso diferente en el calculo final. Cuando se da este caso, es necesario asignar memoria para almacenar dichos coeficientes y acceder a ellos para cada elemento, lo que hace que la complejidad del problema crezca y las opciones para optimizar el problema se vean reducidas debido al uso de ancho de banda que deberá utilizarse para leer los coeficientes.

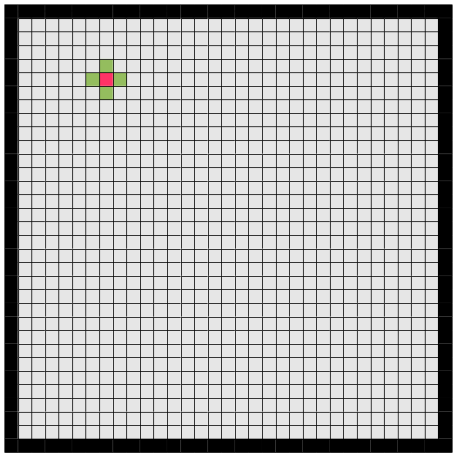

Figura 2.9: Representación dentro de una matriz de los elementos implicados en el cálculo de un *stencil* de 5 puntos. Las fronteras de la matriz, en negro, representan elementos con valor fijo que no son actualizados.

2.6. *Stencils* de este trabajo

A continuación se describen los dos *stencils* que se van a implementar en este trabajo.

2.6.1. *Stencil* de 5 puntos

El *stencil* de 5 puntos que se utiliza en este trabajo es un *stencil* bidimensional, de primer orden, de tamaño 5 y con una huella de 3^2. El tipo de iteración utilizado es Jacobi, por lo que la matriz sobre la que se aplica el cálculo necesita ser almacenada por duplicado. Las fronteras de la matriz representan valores constantes, por lo que durante la ejecución del algoritmo no se actualizarán sus valores, tan solo se consultarán cuando sea necesario. Este *stencil* pondera de igual manera los 5 puntos que se utilizan para el cálculo, por lo que solo se requiere un coeficiente, que será una constante definida en tiempo de compilación. El cálculo de cada elemento de este *stencil* se define con la siguiente expresión (Figura 2.9):

$M_{i,j}^{i+1} = C_0 * (M_{i,j}^{i} + M_{i-1,j}^{i} + M_{i+1,j}^{i} + M_{i,j-1}^{i} + M_{i,j+1}^{i})$

El cálculo de este *stencil* supone un total de 5 operaciones en coma flotante por cada elemento. Considerando un entorno ideal, en el que existe una total reutilización de los datos, solo sería necesario transmitir 12 Bytes por elemento. Este cálculo se obtiene al asumir la transferencia de 3 elementos por cada cálculo, donde en cada cálculo se lee de la matriz de lectura la misma posición en la que se encuentra el elemento a calcular y genera una escritura en la matriz de escritura, que por el tipo de política que la cache utiliza (*write allocate*),

generará una lectura previa. Es posible sin embargo reducir el tráfico ideal a 8 Bytes con la optimización de cache bypass. Hay que hacer notar que el cálculo previo de la transferencia de datos es para un entorno ideal y está lejos de ser el que realmente se obtenga.

Para calcular la intensidad computacional utilizamos la siguiente lógica. Siendo n el total de elementos del volumen de datos, teniendo en cuenta que se requieren 5 operaciones en coma flotante para este cálculo y que se utilizan iteraciones tipo Jacobi (una matriz de lectura y una de escritura), se necesitarán en el mejor de los casos el uso de como mínimo $2n$ elementos de memoria, n de lectura y n de escritura (suponiendo que solo lee cada elemento una vez, aunque en la práctica existirán varías lecturas de datos repetidas). Esto hace que la intensidad computacional óptima del algoritmo sea $I = f/w = 5n/2n = 5/2$.

2.6.2. *Stencil* de 27 puntos

El *stencil* de 27 puntos es un *stencil* tridimensional, de primer orden, de tamaño 27 y con una huella de 3^3. El tipo de iteración utilizado es Jacobi, por lo que la matriz sobre la que se aplica el cálculo necesita ser almacenada por duplicado. Las fronteras de la matriz representan valores constantes, por lo que durante la ejecución del algoritmo no se actualizarán sus valores, tan solo se consultarán cuando sea necesario. Este *stencil* pondera con cuatro coeficientes los diferentes puntos en función de la distancia al elemento central, los coeficientes serán constantes que se definirán en tiempo de compilación. El cálculo de cada elemento de este *stencil* se define con la siguiente expresión (Figura 2.10):

$M^{i+1}_{i,j,k} = \alpha * A + \beta * B + \gamma * C + \delta * D$

$A = M^{i}_{i,j,k}$

$B = M^{i}_{i-1,j,k} + M^{i}_{i+1,j,k} + M^{i}_{i,j-1,k} + M^{i}_{i,j+1,k} + M^{i}_{i,j,k-1} + M^{i}_{i,j,k+1}$

$C = M^{i}_{i-1,j-1,k} + M^{i}_{i-1,j+1,k} + M^{i}_{i-1,j,k-1} + M^{i}_{i-1,j,k+1} + M^{i}_{i+1,j-1,k} + M^{i}_{i+1,j+1,k} + M^{i}_{i+1,j,k-1} + M^{i}_{i+1,j,k+1} + M^{i}_{i,j-1,k-1} + M^{i}_{i,j-1,k+1} + M^{i}_{i,j+1,k-1} + M^{i}_{i,j+1,k+1}$

$D = M^{i}_{i-1,j-1,k-1} + M^{i}_{i-1,j-1,k+1} + M^{i}_{i-1,j+1,k-1} + M^{i}_{i-1,j+1,k+1} + M^{i}_{i+1,j-1,k-1} + M^{i}_{i+1,j-1,k+1} + M^{i}_{i+1,j+1,k-1} + M^{i}_{i+1,j+1,k+1}$

En este caso, el total de operaciones en coma flotante son 30. La cantidad de datos necesarios que se deben transferir en un entorno ideal es la misma que en el *stencil* de 5 puntos.

Para calcular la intensidad computacional utilizamos la misma lógica que con el *stencil* de 5 puntos. Siendo n el total de elementos del volumen de datos, teniendo en cuenta que se requieren 30 operaciones en coma flotante para este cálculo y que se utilizan iteraciones tipo Jacobi (una matriz de lectura y una

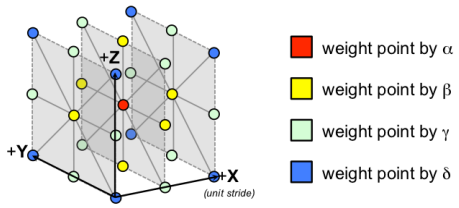

Figura 2.10: Representación de los elementos que engloban un *stencil* de 27 puntos. Los colores de los elementos indican como están ponderados para el cálculo (imagen de[3]).

de escritura), se necesitará traer $2n$ elementos de memoria. Esto hace que la intensidad computacional óptima del algoritmo sea $I = f/w = 30n/2n = 15$.

Capítulo 3

Recopilación de optimizaciones

En general, los códigos que operan con *stencils* pueden presentar un rendimiento bajo cuando se utiliza un código simple sin optimizar. Son muchas las optimizaciones que pueden aplicarse a estos problemas dependiendo de las características de este. En este capítulo se intentará hacer una recopilación de algunas de ellas, obtenidas de la bibliografía existente sobre este problema. Previamente, se intentará dar explicación al porqué del bajo rendimiento y en que pautas convendría centrarse para mejorarlo.

Hay que remarcar que no todas las optimizaciones son compatibles o beneficiosas con todos los tipos de arquitecturas o *stencils* disponibles. Por ejemplo, algunas de ellas se basan en reutilizar al máximo los datos almacenados en memoria cache, por lo que solo serán objetivo de sistemas que lleven implementada una jerarquía de memoria basada en memoria cache.

3.1. Causas del bajo rendimiento

Algunos problemas numéricos tienen el inconveniente de que necesitan una gran cantidad de datos en relación a los cálculos necesarios. Cuando el tiempo requerido para llevar los datos de memoria a las unidades de proceso es mayor al tiempo de procesamiento de estos datos, decimos que el algoritmo está limitado por el ancho de banda. En estos casos, es interesante optimizar el acceso a los datos si se quieren obtener mejoras notables en el rendimiento.

Podemos caracterizar el algoritmo por una métrica llamada intensidad computacional ($I = f/w$), que esta definida como la relación entre el número f de operaciones en coma flotante necesarias para realizar un calculo entre el núme-

ro w de elementos necesarios que hay que traer de memoria para realizar ese calculo. Hay que tener también en cuenta el tamaño de dichos elementos, siendo en la mayoría de arquitecturas modernas x86 4 bytes para el caso de elementos en coma flotante de simple precisión y 8 bytes cuando son de doble precisión. Como ejemplo, siendo n el total de elementos del volumen de datos, un *stencil* de 5 puntos que requiere de 5 operaciones en coma flotante y que utiliza iteraciones tipo Jacobi (una matriz de lectura y una de escritura), requiere en el mejor de los casos el uso de como mínimo 2n elementos de memoria, n de lectura y n de escritura (suponiendo que solo lee cada elemento una vez, aunque en la práctica existirán varías lecturas de datos repetidas). Esto hace que la intensidad computacional óptima sea $I = 5n/2n = 5/2$. En caso de utilizar un *stencil* de 7 puntos que requiera de 8 operaciones en coma flotante, la intensidad computacional óptima pasará a ser $I = 8n/2n = 4$. Resulta obvio a simple vista que, a mayor tamaño de *stencil*, mejor intensidad computacional, ya que se verá incrementado el número de operaciones en coma flotante mientras que el número de elementos necesarios se mantendrá constante. A mayor intensidad computacional, las optimizaciones efectuadas en el cálculo tendrán una mayor mejora de rendimiento, por lo que, generalmente, los *stencils* de menor tamaño no mejorarán su rendimiento tanto como los de mayor tamaño.

Para los algoritmos limitados por el ancho de banda, existe un límite de eficiencia $Peff$ (en FLOPS), que se obtiene con el ancho de banda B del sistema (en bytes/s) y la intensidad computacional (en FLOP/elementos). Si estamos usando datos en coma flotante de simple precisión, el límite vendrá dado por $Peff <= BI/4 = fB/(4w)$. Este límite será frecuentemente inferior al rendimiento teórico del hardware para los cálculos en coma flotante [9].

3.2. Descomposición del problema

En esta sección se discute como llevar a cabo la descomposición del problema. El objetivo principal es poder distribuir los datos y paralelizar los cálculos, cosa que se obtiene estableciendo subproblemas para determinar cual es el trabajo que tiene que hacer cada uno de los diferentes hilos de ejecución.

La descomposición se verá afectada por el tipo de hardware que utilicemos, el esquema será radicalmente diferente dependiendo de si utilizamos sistemas multi-core o many-core. También influirá el modelo de programación paralela utilizado. En modelos basados en memoria compartida, donde el volumen de datos sobre el que se trabaja es el mismo para cada hilo, solo es necesario tener en cuenta la sincronización de los hilos al finalizar las iteraciones, mientras que en modelos basados en memoria distribuida, donde se utilizan diferentes volúmenes de datos para cada hilo, hay que considerar también la comunicación entre hilos

de los elementos en las fronteras de cada subproblema.

Además de lo mencionado, la descomposición de los datos puede tener un efecto secundario positivo en arquitecturas con jerarquías de memoria basadas en cache, y es que puede mejorar la localidad de los datos y ayudar indirectamente a mejorar el rendimiento. También puede indirectamente evitar fallos de cache debidos a conflictos entre diferentes datos que se mapean en la misma posición (*conflict misses*), aunque es deseable que este tipo de problemas se solventen mediante otros métodos como el relleno adicional del volumen de datos.

3.3. Bloqueo de iteraciones en el tiempo

El cálculo de la operación *stencil* se aplica sobre el volumen de datos durante un número finito de iteraciones para llegar al resultado esperado, que puede ser la convergencia o el estado concreto del problema tras un número determinado de iteraciones.

Tras cada iteración, en caso de que el problema haya sido descompuesto en diferentes subproblemas utilizando memoria distribuida, es necesario que se lleve a cabo una sincronización entre los diferentes hilos de ejecución para que los datos en cada uno de ellos sean consistentes, ya que los datos en la frontera del subproblema requerirán los datos de los subproblemas adyacentes, que estarán siendo calculados por otro hilo de ejecución.

En algunas fuentes [10] apuntan como posible optimización llevar a cabo varias iteraciones en cada hilo de ejecución antes de realizar la sincronización con los otros hilos. Esta optimización es llamada bloqueo de iteraciones en el tiempo (del inglés *time blocking*), y tiene como objetivo minimizar el tráfico de memoria maximizando la reutilización de los datos en cache.

En las fuentes se citan principalmente dos formas de utilizar esta optimización, mediante sesgado del tiempo (del inglés *time skewing*) o con el uso de una cola circular (del inglés *circular queue*).

3.3.1. Sesgado del tiempo

Este método consiste en dividir el esquema de progreso del algoritmo con cortes sesgados (en diagonal), generando diferentes bloques con los que trabajar por separado [11]. Los cortes sesgados se hacen para mantener las dependencias de datos entre iteraciones, cosa que garantiza que el bloque i pueda ser calculado independiente del bloque $i+1$ en cualquiera de los casos. El problema es que esa propiedad no se da en el caso contrario (no se puede calcular el bloque $i+1$ antes que el i), por lo que hay que mantener cierto orden en los cálculos de

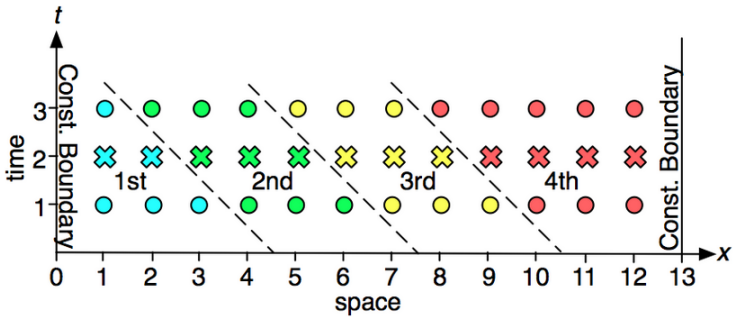

Figura 3.1: En la figura puede verse un esquema de progreso muy simple de un *stencil* unidimensional de 3 puntos. El eje x representa los elementos del volumen de datos unidimensional que se usa para el cálculo. El eje t representa las iteraciones en el tiempo [11]. Los bloques están marcados en diferente color.

los bloques para obtener un resultado correcto. Esto afecta negativamente a las posibilidades de paralelización del algoritmo.

Otro de los inconvenientes de este método es que no todo el trabajo se reparte de forma equitativa entre los diferentes bloques. A pesar de la división inicial del trabajo, que es equitativa, la inclinación del corte hace que uno de los bloques efectúe un número menor de cálculos y otro de ellos efectúe más cálculos que el resto, como puede observarse en la Figura 3.1.

También hay que tener en cuenta que el número de iteraciones que puede llevar a cabo cada uno de los bloques es limitado, ya que en cada iteración las dependencias de datos se vuelven más grandes. Esto puede mitigarse creando bloques más grandes, pero entonces existe el riesgo de perder la localidad de los datos, que es lo que busca la optimización. Hay que buscar pues el tamaño con el que esta optimización resulta óptima, que dependerá de la jerarquía de memoria cache de la máquina donde se ejecuta.

En el trabajo llevado a cabo por Augustin et al. [9], describen la implementación de un tipo de sesgado en el tiempo, junto con otras optimizaciones, sobre ciertos *stencils*. En el trabajo también presentan los resultados con el rendimiento obtenido según el número de iteraciones en el tiempo que llevan a cabo a la vez (Figura 3.2).

Wittmann et al. [12] también presentan en su trabajo otro tipo de sesgado del tiempo mediante oleadas (del inglés *wavefront*).

3.3.2. Cola circular

Este otro método emplea una estructura de datos llamada cola circular. A pesar de que se está haciendo referencia a su uso para llevar a cabo varias iteraciones en el tiempo a la vez, es posible implementarlo para que solo lleve a

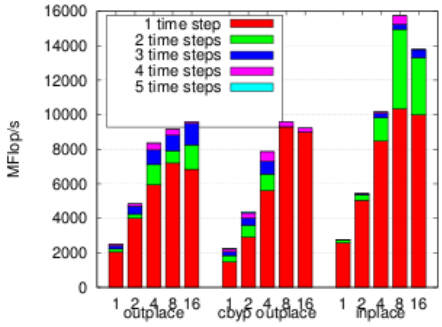

Figura 3.2: En esta figura pueden verse los resultados de las ejecuciones en MFLOPS en una máquina Intel Nehalem de 4 núcleos y con capacidad para 2 hilos por núcleo, con una frecuencia de reloj de 2.27GHz y utilizando un volumen de datos de 512x512x512 elementos. Se trata de un *stencil* de 7 puntos tridimensional y cada cálculo requiere de 8 operaciones en coma flotante. Se puede observar que el sesgado en el tiempo beneficia más a la implementación con Gauss-Seidel (*inplace*) que a las dos implementaciones que usan Jacobi (*outplace* y *cbyp outplace*) [9].

cabo una.

Esta estructura depende de la dimensionalidad del *stencil*. En caso de tratarse de un *stencil* bidimensional, estará formada por vectores, mientras que si el *stencil* es tridimensional, estará formada por planos. Con *stencils* unidimensionales se formaría por elementos individuales, por lo que perdería efectividad. A partir de ahora, para dar la explicación del algoritmo, se considerará que se está usando un *stencil* tridimensional y que, por lo tanto, la cola circular está constituida por planos [11].

El proceso funciona completando el cálculo de planos. Para un *stencil* tridimensional de primer orden, en el caso de que solo se lleve a cabo una iteración a la vez, serán suficientes un total de 3 planos (el actual, el anterior y el posterior, cuando el *stencil* es de primer orden). Una vez acabado con un plano y para pasar al siguiente, solo se tienen que modificar los punteros para que estos apunten a las posiciones de memoria correspondientes a los próximos planos. Para poder llevar a cabo el bloqueo de iteraciones en el tiempo, será también necesario almacenar un total de *t-1* conjuntos de planos adicionales, donde t es el número de iteraciones en el tiempo que se llevan a cabo a la vez (Figura 3.3).

La principal ventaja del algoritmo se encuentra en la facilidad que existe para descomponer el problema en bloques independientes, a diferencia de lo que ocurre con el método de sesgado del tiempo. La forma en la que se llevan a cabo los cálculos consigue que se pueda mantener un buen grado de localidad, lo que ayuda a la memoria cache.

Sin embargo, está opción requiere de un mayor número de recursos almace-

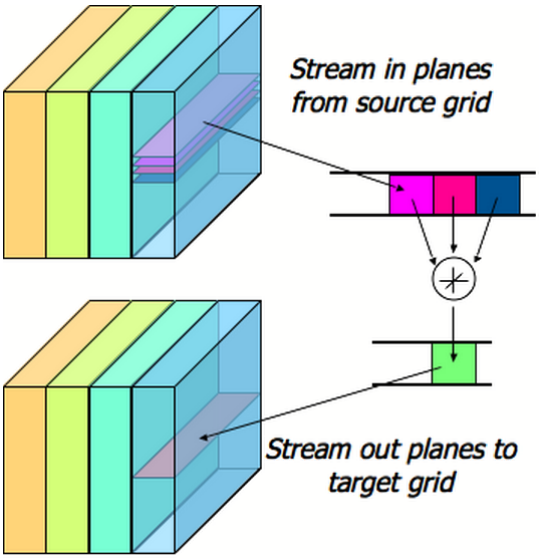

Figura 3.3: En está figura se muestra parte de una iteración del algoritmo como resultado de combinar los 3 planos correspondientes. El volumen superior es el de lectura y el inferior el de escritura. Cada uno de los cortes verticales en el volumen de datos representa un bloque diferente, que puede ser calculado con independencia del resto siempre y cuando se proceda a la sincronización de estos después de cada iteración [11].

nados en memoria, por lo que se corre el riesgo de producir conflictos en memoria cache y eliminar datos que se iban a reutilizar, por lo que es necesario estimar el número de iteraciones a la vez que se quieren llevar a cabo e ir experimentando hasta dar con los valores más apropiados.

Cabe destacar que calcular varias iteraciones en el tiempo requiere realizar operaciones redundantes. Debido a que las operaciones con *stencils* están generalmente limitadas por el ancho de banda y no por el procesamiento de los datos, hay que buscar un compromiso entre ambos valores para que esos cálculos redundantes no empeoren el rendimiento.

En el trabajo de Christen et al. [13, 14] se describe la implementación en una arquitectura CELL BE de un *stencil* de 7 puntos y lo compara con otras implementaciones no optimizadas en otras arquitecturas, en dicha implementación se emplea el método de cola circular (Figura 3.4).

Kamil [11], dispone en su web de un repositorio git con código para esta implementación fácilmente modificable para poder hacer pruebas.

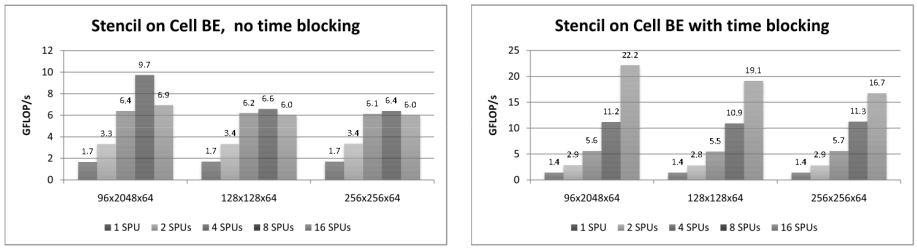

Figura 3.4: En esta figura se ven los resultados de las pruebas en GFLOP/s con el código para CELL BE con y sin bloqueo de iteraciones en el tiempo. Cada grupo representa un volumen de datos diferente y para cada uno se experimenta con distinto número de SPUs. Puede observarse como, sin cola circular, a partir de 16 SPUs el rendimiento baja porque no hay suficiente ancho de banda, sin excepción en todos los volúmenes de datos. Sin embargo la mejora al usar 16 SPUs cuando se dispone de cola circular es muy notoria, lo que deja ver el beneficio de este método en el tipo de hardware empleado [13].

3.4. Posicionamiento de los datos

Como se ha visto en secciones anteriores, el tratamiento de los datos en este problema es crítico. Además de una buena estrategia de reutilización, es importante asignar los datos en posiciones de memoria que no afecten negativamente al rendimiento de la aplicación. Es importante pues tener en cuenta las características del hardware, por ello hay que valorar si es necesario el uso de técnicas como el relleno adicional del volumen de datos o si el acceso a la memoria es uniforme (UMA) o no uniforme (NUMA) [3].

3.4.1. Datos de relleno

En ocasiones, dos o más elementos necesarios para un cálculo que pertenecen a diferentes posiciones de memoria, tienen asignados el mismo banco de memoria cache, esto puede deberse a que la dimensión del volumen de datos es múltiplo del tamaño de la memoria cache, con lo que datos que se encuentran en la misma columna pero en diferentes filas o planos comparten el mismo banco en cache. Si la asociatividad de dicha cache es suficientemente alta, en principio no existirá problema alguno, ya que existirá espacio para que varios datos asignados al mismo banco puedan coexistir sin expulsarse los unos a los otros. Sin embargo, si la asociatividad no es suficientemente alta, algunos de los datos serán eliminados de la cache y tendrán que ser recuperados de memoria principal de nuevo en un margen pequeño de tiempo, lo que creará un mayor tráfico de memoria innecesario y empeorará el rendimiento de la aplicación.

Para solucionar la situación descrita, es posible introducir relleno en la matriz de datos, en concreto a la dimensión cuyo salto entre elementos adyacentes sea

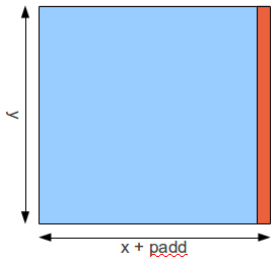

Figura 3.5: La figura representa una matriz a la que se le ha añadido un relleno adicional en la dimensión x para evitar posibles conflictos en cache.

unitario. Sin embargo, este tipo de optimización hace que aumenten los recursos de memoria necesarios y requiere de ciertas modificaciones en el código para que solo se compute la parte útil del volumen de datos (Figura 3.5).

3.4.2. Acceso a memoria no uniforme

En un sistema donde existen varios procesadores físicamente separados, un sistema diseñado para tener acceso a memoria uniforme puede acceder desde cualquier procesador a cualquier módulo físico de memoria con la misma latencia para todos ellos. Sin embargo, en los diseños donde el acceso a memoria es no uniforme, cada procesador físico tiene asignados ciertos módulos de memoria para los cuales la latencia es menor. A pesar de que existan mecanismos hardware para poder acceder a cualquier módulo de memoria, sea local o no, los accesos a módulos no locales tendrán una penalización de tiempo que los hará menos apropiados para trabajar con ello desde ese procesador. Puede verse un ejemplo de esta arquitectura en la Figura 1.2.

En el caso de los problemas con *stencils*, hay que intentar que cada procesador trabaje sobre una porción de datos que pertenece a su memoria local. Esto es un problema cuando se utiliza memoria compartida por todos los hilos de ejecución, pero es fácil de solucionar cuando cada hilo trabaja sobre su propia memoria. Además existen librerías que permiten controlar el comportamiento del sistema en cuanto a la asignación de memoria como libnuma [15].

3.5. Transferencia de datos

En cuanto a la transferencia de datos, solo pueden hacerse dos cosas, reducir la cantidad de datos transferidos o mitigar su efecto en el rendimiento solapando la transferencia de datos con otros cálculos. Dos posibles soluciones para ello son: traer los datos antes de que sean necesarios o evitar que ocurran fallos de escritura en cache que causen transferencias de datos innecesarias [3].

3.5.1. Carga temprana de datos explícita

El hardware de muchos procesadores actuales puede identificar patrones de acceso a los datos en un programa en ejecución y llevar a cabo automáticamente una carga temprana de datos, cosa que se hará de forma implícita sin que el programador o el usuario influyan en la decisión.

Sin embargo, este tipo de carga temprana no es infalible, se basa en ciertos patrones de acceso y en una lógica que esta implementada por hardware y de la cual no existe control por parte del programador. Por ello puede resultar conveniente indicar explícitamente en el código de la aplicación que se traigan datos de memoria principal a memoria cache. Si se hace correctamente, esto puede ayudar a solapar la transferencia de datos con los cálculos, reduciendo el tiempo en que el procesador se encuentra ocioso y aumentando el rendimiento. Suele ser necesario llevar a cabo varios experimentos para obtener resultados positivos, ya que en muchas arquitecturas basadas en cache se da por hecho que los mecanismos hardware son los que se encargarán de este trabajo, por lo que este tipo de optimizaciones no son triviales de aplicar.

En algunos sistemas, como CELL BE, este es el único sistema para transferir datos de memoria principal a cada una de las unidades de proceso. En CUDA, existe un tipo de memoria llamado memoria compartida donde también es necesario traer los datos de forma explicita desde memoria de dispositivo. Otros sistemas basados en jerarquías de memoria cache también ofrecen instrucciones de este tipo.

3.5.2. Cache bypass

En los algoritmos de *stencils* que utilizan iteraciones de tipo Jacobi, no es necesario leer ningún dato de las matrices de escritura, es suficiente con asegurar que el dato se ha escrito en memoria principal.

En algunos diseños de jerarquía de memoria, cuando se intenta escribir un dato en memoria principal, se comprueba si ese dato se encuentra en memoria cache para que pueda escribirse más rápidamente y, en caso de no estarlo, se produce un fallo de escritura, lo que provoca una lectura de memoria principal para traer el dato a la cache y escribirlo allí. Esto puede ser positivo en muchos casos donde el dato escrito va a ser usado en un corto espacio de tiempo, pero resulta en transferencias de datos innecesarias en este problema y en posibles perdidas de rendimiento.

Sin embargo, este comportamiento es inherente al sistema utilizado y sus operaciones de escritura comunes harán uso de él, le interese al programador o no. Es posible evitarlo solo si el sistema ofrece algún tipo de instrucción en ensamblador para escribir el dato directamente en memoria principal, evitando

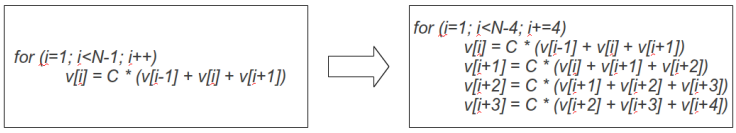

Figura 3.6: Las imágenes representan fragmentos del código de un *stencil* unidimensional de 3 puntos. En la imagen de la izquierda, se encuentra el bucle interno del *stencil* en el que se calcula un elemento por cada iteración. En la imagen de la derecha, se ha procedido a desenrollar el bucle para que este lleve a cabo el cálculo de 4 elementos en vez de uno. Los elementos comunes entre los diferentes elementos pueden almacenarse en registros para tener que consultarlos solo una vez. Una vez aplicada la optimización y si el número de elementos a calcular no es múltiplo del nivel de desenrollado, será necesario añadir código al finalizar el bucle que se encargue de los elementos que no han podido ser calculados.

la lectura de esa posición de memoria en caso de que se produzca un fallo de escritura.

En el caso de las arquitecturas x86, no solo existe una instrucción en ensamblador para ello, sino que es posible hacerlo mediante instrucciones SSE, por lo que no es necesario usar ensamblador para ello [9], sin embargo, este tipo de instrucciones requieren del uso de registros multimedia, que trabajan con datos que están alineados a 128 bits y que escriben a la vez un total de 128 bits, por lo que resulta en grandes modificaciones del código para poder implementarlo. Además del anterior inconveniente, recurrir a este tipo de instrucciones reduce la portabilidad del código entre diferentes sistemas.

3.6. Optimizaciones en el cálculo

Además de optimizaciones para paliar el efecto de la limitación del ancho de banda, también pueden emplearse optimizaciones en el cálculo del *stencil*. Estas optimizaciones afectarán principalmente en función de la intensidad computacional del problema en cuestión, por lo que los *stencils* de mayor tamaño se verán generalmente más beneficiados.

3.6.1. Desenrollado de bucles

Esta optimización consiste en desenrollar el bucle interno de la operación para poder llevar a cabo la actualización de varios elementos en una misma iteración. Este desenrollado se lleva a cabo usualmente en la dimensión cuyo salto entre elementos adyacentes es unitario, pero puede hacerse en cualquier dimensión (Figura 3.6).

Con esto se consigue un mejor aprovechamiento de los registros de cada

núcleo, pero puede provocar consecuencias negativas en caso de desenrollar demasiadas iteraciones, ya que el número de registros disponibles es limitado y, en caso de usarlos todos, algunos datos serán eliminados de registros (*register spill*) y traídos más tarde de memoria, con su consecuente perdida de rendimiento.

Posteriormente a esta actualización, puede ser posible aplicar otras técnicas como la eliminación de subexpresiones comunes, que aparece debido a que elementos adyacentes comparten algunos cálculos. Está optimización está explicada con detalle en la siguiente sección.

Por otra parte, es posible que el compilador efectúe por si solo desenrollado de bucles dependiendo de las opciones de optimización dadas y de si es capaz de comprender el patrón de los cálculos del bucle interior, aunque esto no siempre ocurre y es posible que se tenga que implementar la optimización explícitamente.

3.6.2. Eliminación de subexpresiones comunes

En algoritmos que implementan *stencils*, cuando se ha aplicado la optimización de desenrollado de bucles, es posible que aparezcan cálculos redundantes dentro de una misma iteración debido a que se están actualizando elementos adyacentes que tienen elementos en común en sus cálculos.

Es posible reducir la cantidad de operaciones y mejorar el rendimiento del algoritmo si se eliminan los cálculos redundantes, aunque el impacto de esta optimización dependerá del tipo de *stencil* que se esté implementando.

Es importante destacar que la eliminación de cálculos redundantes puede tener como consecuencia la reducción del número de operaciones en coma flotante. En caso de mostrar el rendimiento en métricas como FLOP/s, este fenómeno tendrá que tenerse en cuenta si se quiere dar un resultado más ajustado a la realidad.

Otro detalle que conviene tener en cuenta, es el hecho de que alterar el orden en que se llevan a cabo las operaciones en coma flotante puede provocar que los resultados finales no sean exactamente iguales a cuando no se aplica la optimización, ya que la no asociatividad de las operaciones en coma flotante afecta al resultado final. Esto ocurre cuando se llevan a cabo previamente cálculos comunes entre elementos adyacentes y se modifica el orden de las instrucciones.

3.6.3. Vectorización

La última optimización pretende explotar los repertorios de instrucciones que hacen uso del paralelismo a nivel de datos (SIMD). Este tipo de instrucciones son exclusivas de cada fabricante y es necesario documentarse para saber cuales son soportadas por el hardware disponible.

En el caso de las máquinas x86 de Intel, existe para dicho fin el repertorio de instrucciones SSE en ensamblador, que utiliza registros de 128 bits que pueden almacenar 4 elementos en coma flotante de simple precisión o 2 de doble precisión [16].

Existe la posibilidad, mediante operaciones intrínsecas, de implementar este tipo de optimizaciones con lenguajes como C [17]. También algunos compiladores son capaces de aplicar este tipo de optimizaciones si se les indica en las opciones de optimización, aunque no siempre encuentran la oportunidad de hacerlo [18].

El mayor inconveniente que traen este tipo de operaciones, es la necesidad de efectuarlas sobre datos alineados a 128 bits, perdiendo eficiencia en caso de que los datos no estén alineados. Esto puede provocar la necesidad de introducir relleno en el volumen de datos para que el elemento inicial de cada fila de la matriz esté alineado correctamente. En otras arquitecturas, como Blue Gene/P, los accesos a datos no alineados son directamente ilegales y causan fallos en la ejecución [3]. Esta optimización, generalmente, requiere modificar gran parte del código y hace que éste deje de ser portable entre diferentes sistemas, incluso cuando comparten arquitectura.

Capítulo 4

Entorno de pruebas

Antes de poder decidir como proceder con las implementaciones, hace falta conocer que recursos hardware y software hay disponibles. En este capítulo se hará un repaso al entorno de pruebas disponible y se valorará de que opciones software sería conveniente emplear en el desarrollo.

4.1. Arquitectura

En primer lugar, es necesario conocer las características del hardware disponible. Las especificaciones del sistema son:

- Dos procesadores Intel Xeon E5520 a 2.26 GHz.
  - La tecnología *HyperThreading* está deshabilitada.
  - Cada procesador consta de 4 núcleos, lo que nos da un total de 8 hilos de ejecución paralelos.
- 8 GB de memoria RAM, repartidos entre los dos procesadores.
  - El acceso a memoria es no uniforme (NUMA), por lo que existen afinidades entre procesadores y bancos.
- Jerarquía de memoria cache con tamaño de linea de 64 Bits y 3 niveles diferentes.
  - Primer nivel de memoria cache de 32 KB, independiente para cada núcleo.
    - 64 bancos y 8-asociativa.
    - Una para datos y otra para instrucciones.

- Segundo nivel de memoria cache de 256 KB, independiente para cada núcleo.
    - 512 bancos y 8-asociativa.
    - Unificada para datos e instrucciones.
- Tercer nivel de memoria cache de 8 MB, compartida entre todos los núcleos de un mismo procesador.
    - 8192 bancos y 16-asociativa.
    - Unificada para datos e instrucciones.

- El repertorio de instrucciones SSE incluye hasta la versión 4.2 de este.
- Dos GPU Nvidia GeForce GTX 295.
    - 240 núcleos CUDA por cada GPU.
    - 896 MB de memoria por cada GPU.
    - CC (*Compute Capabilty*)1.3.

La máquina se utiliza como servidor destinado a este tipo de pruebas, para ello tiene instalado un sistema GNU Linux, más concretamente la distribución Ubuntu Server 10.04.2 LTS compilada para máquinas de 64 bits. El kernel Linux que ejecuta es la versión 2.6.32-28-server.

4.2. Modelos de programación paralela

El objetivo principal de este trabajo es optimizar problemas con *stencils*. Dado que estos problemas permiten paralelizar de forma razonable el trabajo en distintos hilos de ejecución, es importante escoger en primer lugar que modelo o modelos de programación paralela van a emplearse en la implementación. Es interesante pues, seleccionar algunos de los modelos existentes e implementar el problema en todos ellos para observar las ventajas, inconvenientes y el resultado final que nos brinda cada uno de ellos.

Así pues, y teniendo en cuenta los diferentes enfoques que ofrece cada uno, los modelos de programación paralela usados serán OpenMP, Pthreads y MPI cuando se trabaje con el procesador y Nvidia CUDA cuando se trabaje con la GPU.

4.2.1. OpenMP

OpenMP (*Open Multi-Processing*) es una API pensada para proporcionar paralelismo a nivel de hilos de forma sencilla, con simples directivas que se

añaden en el código fuente y que permiten que el compilador pueda paralelizar esas secciones de código como se le indica. Soporta diferentes plataformas y arquitecturas, lo que hace que el código pueda portarse con cierta facilidad [19].

OpenMP trabaja en entornos de memoria compartida, por lo que es una buena solución cuando el trabajo se centra en un único sistema. No resulta útil para *clusters* y sistemas distribuidos, ya que no es posible distribuir el trabajo entre diferentes máquinas solo con lo que la librería ofrece.

Frente a otros modelos de programación paralela, OpenMP tiene la ventaja de que, en general, requiere pequeños cambios en algunas secciones del código original para poder paralelizar de forma efectiva el trabajo. Sin embargo, no ofrece un control tan directo al programador sobre los hilos de ejecución, en contraposición a lo que si hacen otras alternativas como Pthreads. También carece de la posibilidad de establecer la afinidad de los hilos con los núcleos, lo que puede resultar en penalizaciones de rendimiento cuando se utiliza acceso a memoria no uniforme (NUMA).

Otra ventaja que facilita su interoperabilidad con distintas plataformas es que está incluido en las últimas versiones de compiladores como GNU GCC o Intel ICC. Además funciona con los lenguajes de programación C, C++ y Fortran.

El sistema utilizado en este trabajo soporta las especificaciones de OpenMP versión 2.5.

4.2.2. Pthreads

Pthreads (POSIX Threads) es una API pensada para proporcionar paralelismo a nivel de hilos. Esta implementada cumpliendo el estándar IEEE POSIX 1003.1c, usado para la paralelización de hilos en sistemas UNIX [20].

Al igual que OpenMP, Pthreads también trabaja en entornos de memoria compartida. Sin embargo, lejos de imitar la forma en que OpenMP indica como paralelizar los datos, Pthreads requiere de grandes modificaciones en el código para que pueda llevarse a cabo la paralelización de las tareas. Con este sistema, se explicita en el código la creación de hilos y el trabajo que debe hacer cada uno de estos, siguiendo una estructura de hilo maestro e hilos esclavos (en OpenMP también existe esta estructura, pero queda oculta de no ser que el programador utilice ciertas directivas para tomar algo más de control).

A diferencia de OpenMP, aquí tendremos un mayor control sobre la afinidad de los hilos y los datos a los que acceden, por lo que será posible evitar accesos a bancos de memoria no locales que comportarían perdidas de rendimiento.

Pthreads esta definido para usarse con el lenguaje de programación C, por lo que esa será la opción si se quiere utilizar dicha API.

4.2.3. MPI

La API de MPI, a diferencia de la de OpenMP y Pthreads, está pensada para sistemas de memoria distribuida, lo que la hace una buena solución para *clusters* y otros sistemas distribuidos.

MPI dispone de varias implementaciones hechas por distintas organizaciones. Se encuentra disponible para sistemas GNU Linux y para un gran número de plataformas y arquitecturas, lo que hace que el código pueda portarse con relativa facilidad a otros sistemas.

Al igual que con Pthreads, adaptar un código secuencial a MPI requiere la reescritura casi completa de éste, ya que sigue una estructura en la que varios hilos ejecutan su propio código y disponen de su propio espacio de memoria. Esto ayuda a prevenir que ocurran problemas de acceso a memoria no uniforme, ya que si cada hilo dispone de su propio espacio de memoria, este se encontrará en un banco de memoria local (a no ser que se haga explícito lo contrario).

Todo el trabajo de sincronización, intercambio de datos y obtención de resultados corre a cuenta del programador. También a nivel de ejecución se cede un gran control al usuario sobre cuantos hilos van a crearse y donde va a ejecutarse cada uno de ellos. Existen, eso si, algunos problemas para aprovechar la tecnología SMT (*Simultaneous MultiThreading*) con MPI [3].

MPI, dependiendo de la implementación que se utilice, está disponible para lenguajes de programación como C y Fortran. En nuestro sistema tenemos la implementación OpenMPI versión 1.4.1, que es compatible con la especificación MPI-2 [21].

4.2.4. Nvidia CUDA

CUDA (*Compute Unified Device Architecture*) es una arquitectura desarrollada por la compañía Nvidia para posibilitar la ejecución en paralelo de tareas de computación general en dispositivos GPU [22].

Este modelo es completamente diferente a los vistos anteriormente, ya que se enfoca en paralelizar un número alto e indeterminado de hilos a la vez, que puede ser de cientos o miles. La ventaja reside en que suele haber cientos de núcleos disponibles a la vez y que el cambio de contexto entre estos se produce prácticamente sin coste alguno.

El enfoque aquí hace que cada hilo se encargue de una porción muy pequeña del problema, que puede ser incluso un solo elemento. Esto provoca que el código de la aplicación secuencial tenga que ser reescrito de nuevo, posiblemente descartando la metodología utilizada y volviendo a analizar el problema inicial para encontrar una forma de aprovechar esta arquitectura.

CUDA ofrece dos opciones diferentes para su uso en función de las necesidades del desarrollador, una a más alto nivel (C para CUDA) y otra a más bajo nivel (API del controlador). Ambas son mutuamente excluyentes, utilizar una implica no utilizar la otra. En general, el uso de C para CUDA ofrece suficiente nivel de detalle para la mayoría de problemas, y es el que se utilizará en este trabajo.

C para CUDA intenta conseguir que el desarrollo de aplicaciones para las GPUs sea cuanto menos complejo posible para alguien que conociese de antemano el lenguaje C. Consta de dos partes: un repertorio de extensiones para C y un entorno de ejecución.

Las extensiones para C permiten definir funciones especiales llamadas *kernels*, que serán asignadas a una GPU para que las ejecute un determinado número de ocasiones en un hilo diferente cada una de ellas, obteniendo un paralelismo masivo. Dentro de esas funciones puede consultarse el identificador del hilo y del bloque en tiempo de ejecución, cosa que permite calcular sobre que datos trabajará dicho hilo. También existen instrucciones que permiten la sincronización entre hilos de un mismo bloque.

El entorno de ejecución se utiliza principalmente para manipular la memoria de la GPU, también conocida como memoria de dispositivo, además de para definir la composición de los bloques y la de la matriz de bloques que hacen de estructura para ejecutar los *kernels*.

La API del controlador permite trabajar a más bajo nivel con CUDA, pero el programador pasa a ser responsable de la inicialización, creación de contextos, carga de módulos y otros aspectos que hacen la programación más compleja. Esta opción sirve para casos especiales en los que, por ejemplo, es necesario consultar la memoria libre del dispositivo en tiempo de ejecución, cosa que no puede hacerse desde el entorno de ejecución.

Para utilizar CUDA, además de un dispositivo compatible, es necesario instalar un kit de desarrollo que provee la compañía y que funciona sobre el compilador GCC. En la máquina que se utiliza en este trabajo, se dispone de la versión 2.3 de *Cuda compilation tools*.

4.3. Lenguajes de programación

El lenguaje utilizado para las implementaciones en este trabajo será C. La principal razón es que ofrece compatibilidad con todos los modelos de programación paralela previamente mencionados, además de que es una de las opciones más extendidas para implementar problemas de esta índole. También se han considerado razones como la experiencia previa del autor con este lenguaje o la facilidad que ofrece para trabajar con punteros, cosa que resulta cómoda

cuando se trabaja con iteraciones de tipo Jacobi, con dos matrices que deben intercambiarse al final de cada iteración.

Fortran, a pesar de estar también muy extendido para aplicaciones científicas, hubiese requerido de aprendizaje previo. Ya que aprender un nuevo lenguaje no forma parte de los objetivos del trabajo y el uso de C para este problema está extendido, finalmente se ha optado por utilizar éste.

4.4. Compilador

El compilador que se empleará sera GCC, cuyo uso está extendido en sistemas GNU Linux y oferta una gran variedad de optimizaciones y *flags* de compilación, así como la posibilidad de trabajar con los repertorios de instrucciones vectoriales SSE.

Si bien es cierto que ICC también cumple con lo mencionado anteriormente, la experiencia previa con GCC y el hecho de que sea muy fácil la utilización de las librerías para utilizar MPI, OpenMP y Pthreads han sido determinantes en esta decisión. La comparación de los resultados con este compilador puede quedar como trabajo futuro. Por otra parte, el compilador para Nvidia CUDA que proporciona la compañía, NVCC, es solo compatible con GCC, por lo que en este caso su utilización es imprescindible.

Para las implementaciones con MPI, OpenMP y Pthreads se utiliza la versión más actual del compilador que se encuentra instalada en nuestra sistema, que es la versión 4.4.3. Sin embargo, y debido a posibles problemas de compatibilidad (según Nvidia), se utilizará la versión 4.3 del compilador cuando se trabaje con Nvidia CUDA.

Capítulo 5

Metodología

Una vez se conoce todo el entorno y se sabe que implementaciones se van a llevar a cabo, es importante definir una metodología de trabajo que permita mantener orden y claridad cuando empiecen a acumularse las versiones del código. A continuación se describen algunos puntos de la metodología que se pretende seguir.

5.1. Implementaciones

La diversidad de *stencils* diferentes que existen dejan fuera del alcance de este trabajo el implementar y optimizar muchos de ellos. Por ello se han seleccionado dos *stencils* diferentes, cuyas propiedades están analizadas en detalle en el capítulo 2.

- 5p-stencil. Es el primer problema que se implementa. Trabaja sobre matrices de dos dimensiones y lleva a cabo un cálculo muy sencillo, lo cual lo hace ideal para una primera aproximación a este tipo de problemas.
- 27p-stencil. Este *stencil* es tridimensional y conlleva un mayor número de operaciones que el anterior.

En ambos casos se utilizarán números en coma flotante de simple precisión.

5.1.1. Paralelización *multi-core*

En primer lugar, es necesaria una implementación secuencial del algoritmo sin ninguna optimización. Está implementación estará en dos ficheros diferentes, uno de los cuales se hará cargo de la lectura y asignación de datos, así como de presentar los resultados por pantalla, mientras que el otro se encargará única-

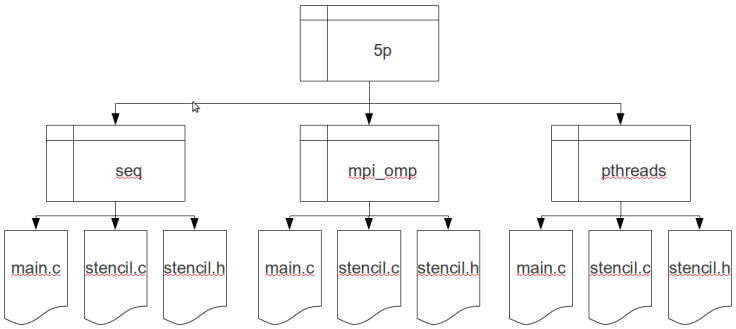

Figura 5.1: En la figura se aprecia la jerarquía de directorios seguida para almacenar cada una de las implementaciones. El archivo main.c contiene la parte que se encarga de hacer todos los preparativos para poder iniciar el algoritmo de *stencil* propiamente dicho, que se encuentra en el fichero stencil.c. En el fichero stencil.h se encuentran las definiciones de funciones y constantes.

mente de la operación *stencil*. Esta decisión se ha tomado con el único objetivo de facilitar la tarea en el proceso de cambios y optimizaciones.

Una vez la implementación inicial está hecha y funciona correctamente, el siguiente paso es modificar el fichero con la operación *stencil* para aplicar las diferentes optimizaciones, asegurando mediante directivas de preprocesador que es posible escoger cualquier versión del código optimizado.

A partir de este momento, con una implementación secuencial funcional y con el cálculo optimizado, se puede portar el código para hacer uso de las diferentes librerías de programación paralela que se han considerado, que en este caso son MPI, OpenMP y Pthreads. Debido a las diferencias en el enfoque que utilizan cada una de ellas, a partir de este momento se deja de utilizar directivas de preprocesador para controlar la versión del código que se compila y pasa a usarse diferentes códigos para cada una de las librerías. En el caso de MPI y OpenMP, se opta sin embargo por una implementación híbridas.

En la Figura 5.1 puede observarse cómo quedarían distribuidos los ficheros con las implementaciones. La estructura es idéntica para los *stencils* de 5 y 27 puntos.

5.1.2. Paralelización *many-core*

A diferencia de las implementaciones para sistemas *multi-core*, aquí no se puede empezar por una implementación secuencial, y es necesario proceder a la implementación paralela desde el primer momento.

Al igual que en el caso anterior, el código esta separado en dos ficheros. En el primero de ellos se encuentra toda la inicialización y asignación de memoria, tanto de el sistema en general como del dispositivo. También se incluye todo lo

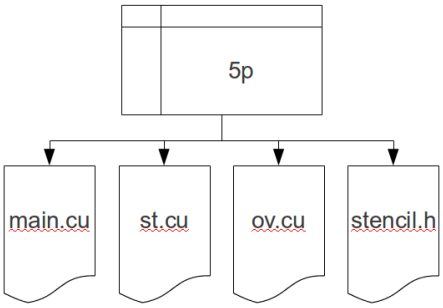

Figura 5.2: En la figura se aprecia la jerarquía de directorios seguida para almacenar cada una de las implementaciones. El archivo main.cu contiene la parte que se encarga de hacer todos los preparativos para poder iniciar el algoritmo de *stencil* propiamente dicho, que se encuentra en los ficheros st.cu (método directo) o ov.cu (método solapado), dependiendo de que algoritmo se quiera utilizar. En el fichero stencil.h se encuentran las definiciones de funciones y constantes.

relacionado a como CUDA repartirá el trabajo. Por otra parte, el otro código implementa un *kernel* que se encarga de hacer una parte muy pequeña del trabajo del algoritmo.

En este caso, y dado que los *kernels* son muy simples y específicos, se hace complejo utilizar directivas de preprocesador entre el código para escoger entre una optimización u otra, por lo que cada vez que se aplica una optimización nueva, se crea un *kernel* específico para esa optimización. Esto aumenta el tamaño del fichero, pero facilita en gran medida la legibilidad del código.

Además de eso, debido a las posibilidades que ofrece CUDA, se han utilizado dos enfoques diferentes para resolver el problema, que hacen que los *kernels* sean muy diferentes entre si, por lo que cada enfoque tiene su código en un fichero distinto. Dichos enfoques han sido nombrados como método directo y método solapado.

En la Figura 5.2 puede observarse como quedarían distribuidos los ficheros con las implementaciones. La estructura es idéntica para los *stencils* de 5 y 27 puntos.

5.2. Compilación

El objetivo principal de este trabajo es la optimización de código, por lo que crear código que decida que optimización aplicar en secciones críticas del código en tiempo de ejecución resultaría perjudicial para los resultados. Es por eso que se hace necesario tener diferentes versiones del ejecutable con cada una de las optimizaciones.

En la sección anterior se ha explicado como se organizan estas versiones a

nivel de ficheros y directorios, así como el hecho de utilizar directivas dentro de los mismos ficheros para separar las diferentes optimizaciones. Sin embargo, tener todos esos detalles en cuenta cada vez que se hacen cambios en el código para volver a compilar todas las versiones resultaría ineficiente, ya que hay un gran número de ejecutables diferentes, por lo que se hace uso de ficheros que se encargan de gestionar la compilación de las distintas versiones, estos ficheros son llamados *Makefiles*.

El uso de estos ficheros facilita esta tarea en gran medida, ya que permite definir las nomenclaturas que se utilizarán en el nombre de los ejecutables para poder identificarlos, así como establecer los *flags* de compilación que se consideren convenientes. Cobra especial importancia el uso de *Makefiles* cuando se compila código para CUDA, ya que ciertos valores que definen el comportamiento del algoritmo y la distribución de este se deciden necesariamente en tiempo de compilación. Teniendo en cuenta la cantidad diferente de posibles combinaciones entre esos parámetros y las optimizaciones que se aplican, la compilación de los ejecutables necesarios sin usar este método se convertiría en una tarea tediosa.

Existen diferentes implementaciones para hacer uso de este tipo de ficheros, sin embargo, y por el uso de ciertas funcionalidades, los *Makefiles* utilizados en este trabajo son solo totalmente compatibles con la implementación GNU Make.

5.3. Ejecuciones

Una vez se dispone de los ejecutables, es el momento de definir qué experimentos se llevarán a cabo y qué proceso se seguirá para ello.

En primer lugar, es importante destacar que el ejecutable que se utiliza es tan solo un parámetro más, ya que en la operación también influyen otros factores como el tipo de división que se hace del problema, el tamaño del volumen de datos o el número de iteraciones, y todos ellos se deciden en tiempo de ejecución mediante parámetros. Se hace pues necesaria la creación de una serie de *scripts* que permitan combinar de forma sencilla y eficiente dichos parámetros, ya que de nuevo se hace imposible realizar dicha tarea manualmente.

Además de lo anterior, hay que añadir el hecho de que se está utilizando una máquina cuyo acceso es público y que se puede ver afectada por tareas de mantenimiento del propio sistema, por lo que es conveniente llevar a cabo varias ejecuciones de cada experimento para asegurar que el valor obtenido se corresponde con la realidad. En este caso se almacena como valido el resultado de menor tiempo que se da en todas las ejecuciones, ya que realizar la media de estos podría dar lugar a ciertas alteraciones cuando uno de los resultados ha dado un valor muy superior al esperado por razones desconocidas.

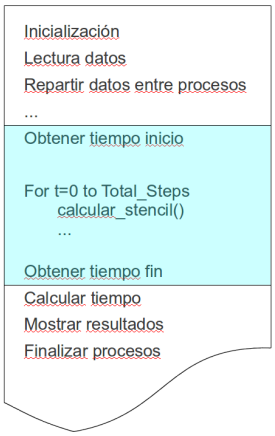

Figura 5.3: En la figura se observa a alto nivel los pasos principales del algoritmo. La parte sombreada corresponde a la fracción de código que se utiliza para dar el resultado del tiempo de ejecución.

De esta forma también se facilita el análisis de resultados, que el *script* se encarga de escribir en un fichero utilizando un formato que resulta sencillo de recuperar para, por ejemplo, incluirlo en una hoja de cálculo.

5.4. Tiempo de ejecución

Existen diferentes alternativas para medir el tiempo de ejecución del algoritmo. Si bien es posible utilizar programas como GNU Time para medir cuanto tarda en ejecutarse una aplicación, esta alternativa no resulta aplicable aquí, ya que no ofrece suficiente granularidad.

Lo que se busca en este trabajo, es medir única y exclusivamente el núcleo del algoritmo de la operación *stencil*, por tanto quedan fuera de este cálculo la preparación previa del algoritmo, que incluye la lectura de datos de disco, la asignación de memoria y el computo que se encarga de decidir como repartir el trabajo; así como el tratamiento posterior al algoritmo, que incluye la visualización de resultados y el propio cálculo del tiempo de ejecución (Figura 5.3).

La alternativa de medir también el tiempo de lectura de datos y la asignación inicial de estos en memoria podría ser también considerada. Sin embargo, dicha tarea se lleva a cabo una sola vez y se da por hecho que el tiempo que necesite dependerá única y exclusivamente del tamaño del problema y no del tipo de problema a resolver, por lo que su valor se mantendrá constante. A medida que se aumente el número de iteraciones que lleva a cabo el algoritmo, la lectura de los datos tenderá a ser una fracción más y más pequeña del tiempo de ejecución,

por lo que acabaría resultando en un tiempo despreciable.

Así pues, y con el fin de obtener una precisión superior a la de los segundos, se ha optado por utilizar la librería que se encuentra definida en *sys/time.h*, que con su función *gettimeofday()* permite obtener el instante actual de tiempo con una precisión de microsegundos.

Para realizar el cálculo del tiempo entre los dos puntos del algoritmo que llevan a cabo la operación de *stencil*, solo es necesario obtener el tiempo entre ambos puntos y, mediante una función que provee la fundación GNU en su apartado sobre como calcular el tiempo entre dos puntos de un código (*http://www.gnu.org/s/libc/manual/html_ node/Elapsed-Time.html*), obtener el tiempo exacto que se emplea para el cálculo.

Los tiempos de ejecución se guardan en coma flotante de simple precisión, pero tras el proceso que se les da para analizar los resultados se convierten a números reales con dos dígitos decimales de precisión, lo cual es suficiente considerando el tiempo de ejecución total del algoritmo.

5.5. Métricas

El tiempo de ejecución del algoritmo, tal y como se ha explicado en la sección anterior, es una métrica fiable para comparar el tiempo entre las diferentes versiones del mismo algoritmo, ya que permite observar directamente si una optimización aplicada afecta positiva o negativamente al código.

Sin embargo, el tiempo de ejecución puede resultar insuficiente para comparar el rendimiento entre distintas operaciones de *stencils* o para comparar diferentes tamaños de problema con la misma versión del algoritmo, por lo que resulta interesante emplear también otras métricas.

Una de estas métricas que se pueden emplear, es la de operaciones en coma flotante por segundo (GFlop/s o GFLOPS). Esta métrica permite obtener una lectura del rendimiento efectivo del algoritmo con independencia de la operación *stencil* que estemos calculando o del tamaño del problema. Además es una métrica ampliamente utilizada por muchos de los resultados dados en el campo de la computación de alto rendimiento, por lo que conviene utilizarla. Hay que hacer sin embargo un pequeño inciso, y es que en caso de utilizar ciertas optimizaciones como la eliminación de subexpresiones comunes, también se están eliminando operaciones en coma flotante del cálculo, por lo que aunque considerar en el cálculo de esta métrica que la cantidad de operaciones en coma flotante son las originales, en realidad se están eliminando dichas operaciones.

Para evitar el problema anterior, también puede utilizarse como métrica los *stencils* por segundo (GStencil/s). En este caso, y aunque se consigue mantener un rendimiento constante independientemente del tamaño del problema, no es

útil para comparar el rendimiento entre diferentes operaciones *stencil*.

Por los motivos dados anteriormente, en este trabajo se utilizarán como métricas tanto el tiempo de ejecución como las operaciones en coma flotante por segundo, ya que ambos nos darán una lectura fiable del rendimiento del algoritmo. Cualquier otro tipo de métricas, incluyendo las que tienen en cuenta la eficiencia energética u otros aspectos, se dejarán como posible trabajo futuro, ya que quedan fuera del alcance de este trabajo.

Capítulo 6

Implementaciones *multi-core*

El objetivo de este capítulo es describir como se han implementado los *stencils* de 5 y 27 puntos para entornos *multi-core*. Por motivos de claridad, se ha dejado el análisis de las pruebas de rendimiento en otro capítulo.

6.1. Implementación secuencial

La implementación secuencial del problema es una tarea relativamente sencilla y el código es similar para los dos *stencils* con los que se trabaja. En primer lugar es necesario asignar dos espacios de memoria idénticos, con suficiente tamaño para importar los datos del fichero donde se encuentra almacenada la matriz que se va a utilizar como estado inicial del problema. Dependiendo de la dimensionalidad de la matriz se tendrán que hacer los cambios pertinentes en el proceso de lectura.

Una vez hecho eso ya se puede proceder al cuerpo del algoritmo, que en su versión más simple sin optimizar se reduce al pequeño fragmento de código mostrado en el Algoritmo 6.1 para el caso del *stencil* de 5 puntos y en el Algoritmo 6.2 para el de 27 puntos. Con todo el trabajo hecho, ya solo queda hacer el cómputo del tiempo de ejecución y mostrar el resultado.

6.2. Paralelización del problema

Paralelizar un problema utilizando librerías como las que ofrece MPI o Pthreads no es una tarea simple, ya que implica importantes modificaciones en el código para poder repartir el trabajo.

Observando el tipo de operación que el *stencil* lleva a cabo, en la que cada elemento accede a otros elementos que le son cercanos, resulta evidente pensar que la mejor división del problema consiste en dividir la matriz de datos en

Algoritmo 6.1 Código que representa al cuerpo del cálculo de un *stencil* de 5 puntos sin aplicar ningún tipo de optimización. El primer bucle es el que se encarga de llevar a cabo el número de iteraciones que se pasa como parámetro, seguido de los bucles que se encargan de recorrer el interior de la matriz bidimensional y del sencillo cálculo que debe aplicarse a cada elemento. Tras cada recorrido por la matriz de datos, se procede a intercambiar los roles entre los espacios de memoria asignados a las matrices de lectura y escritura.

```
#define GET(I,J) readMat[(I)*cols + (J)]
#define PUT(I,J) writeMat[(I)*cols + (J)]

for (cur_time=0; cur_time<timeSteps; cur_time++) {
  for (i=1; i<rows-1; i++) {
    for (j=1; j<cols-1; j++) {
      PUT(i,j) = ALPHA*(GET(i-1,j) +
                        GET(i,j-1) +
                        GET(i,j) +
                        GET(i,j+1) +
                        GET(i+1,j));
    }
  }
  auxPointer = writeMat;
  writeMat = readMat;
  readMat = auxPointer;
}
```

Algoritmo 6.2 Código que representa al cuerpo del cálculo de un *stencil* de 27 puntos sin aplicar ningún tipo de optimización. El primer bucle es el que se encarga de llevar a cabo el número de iteraciones que se pasa como parámetro, seguido de los bucles que se encargan de recorrer el interior de la matriz tridimensional y del cálculo que debe aplicarse a cada elemento. Tras cada recorrido por la matriz de datos, se procede a intercambiar los roles entre los espacios de memoria asignados a las matrices de lectura y escritura.

```
#define GET(I,J,K) readMat[(I)*sliceSize + (J)*cols + (K)]
#define PUT(I,J,K) writeMat[(I)*sliceSize + (J)*cols + (K)]

for (cur_time=0; cur_time<timeSteps; cur_time++) {
  for (i=1; i<slices-1; i++) {
    for (j=1; j<rows-1; j++) {
      for (k=1; k<cols-1; k++) {
        PUT(i,j,k) = ALPHA * (/*ALPHA ELEMENT*/) +
                     BETA  * (/*BETA ELEMENTS*/) +
                     GAMMA * (/*GAMMA ELEMENTS*/) +
                     DELTA * (/*DELTA ELEMENTS*/);
      }
    }
  }
  auxPointer = writeMat;
  writeMat = readMat;
  readMat = auxPointer;
}
```

varias secciones de menor tamaño y asignar cada una de ellas a un hilo, que se encargará de computar los elementos que queden dentro del subespacio asignado. Esto beneficiará a la localidad de los datos.

Cada librería tiene sus propias peculiaridades. En las siguientes secciones se describe como se ha procedido en cada una de ellas.

6.2.1. OpenMP

En cuanto a la aparición de OpenMP en este código, su inclusión resulta en una tarea bastante sencilla. El usuario, al ejecutar el código, debe indicar en un parámetro el número de subprocesos en OpenMP que se quieren crear en cada subproblema. Ese número de procesos se asigna al sistema mediante el entorno de ejecución de OpenMP con una llamada a una librería. Tras ello, tan solo es necesario crear un bucle adicional que recoja cierto número de filas (en caso del *stencil* de 5 puntos) o planos (en caso del *stencil* de 27 puntos) y forzar que OpenMP lo paralelice con una de sus directivas. El Algoritmo 6.3 es un ejemplo de esto para el caso del *stencil* de 5 puntos, el de 27 puntos se hace de forma similar, por lo que se omite su representación.

Es importante tener en cuenta que, en sistemas con acceso a memoria no uniforme, si alguno de los hilos queda mapeado en un procesador diferente al que ha llevado a cabo la asignación de memoria para la matriz, el acceso a memoria será más lento desde ese hilo que desde el resto, lo que se traducirá en un aumento del tiempo de ejecución.

6.2.2. MPI

Esta implementación es un híbrido que incluye la modificación de OpenMP de la anterior sección, ya que OpenMP es muy poco intrusivo y permite que se pueda utilizar MPI sobre la nueva versión.

La razón por la que conviene incluir estas dos librerías en el mismo ejecutable, es que gracias a la combinación de ambos se hace posible repartir el trabajo en entornos distribuidos como un *cluster* y, a la vez, paralelizar el subproblema creado con MPI con OpenMP.

En primer lugar, hay que reescribir el código para que éste permita su paralelización con MPI. La mayoría de modificaciones se llevan a cabo fuera del núcleo del algoritmo donde se lleva a cabo el cálculo, y consisten en inicializar ciertas variables para poder identificar el hilo con el que se está trabajando y preparar los datos para cada subproblema.

Al ejecutar el algoritmo se pasan como parámetros tanto los datos con los que se va a trabajar, que pueden tener cualquier tamaño en las dos o tres dimensiones (dependiendo del *stencil*), como el número de hilos que se van a

Algoritmo 6.3 Código que representa al cuerpo del cálculo de un *stencil* de 5 puntos con la modificación necesaria para que pueda paralelizarse automáticamente con OpenMP. La variable *nrows* tiene como valor el número de filas que cada hilo de OpenMP debe calcular. Solo han cambiado tres lineas de código respecto al núcleo del algoritmo original.

```
#define ALPHA 0.2f
#define GET(I,J) readMat[(I)*cols + (J)]
#define PUT(I,J) writeMat[(I)*cols + (J)]

for (cur_time=0; cur_time<timeSteps; cur_time++) {
  #pragma omp parallel for private(ii, i, j)
  for (ii=1; ii<rows-1; ii+=nrows) {
    for (i=ii; i<MIN(ii+nrows,rows-1); i++) {
      for (j=1; j<cols-1; j++) {
        PUT(i,j) = ALPHA*(GET(i-1,j) +
                          GET(i,j-1) +
                          GET(i,j) +
                          GET(i,j+1) +
                          GET(i+1,j));
      }
    }
  }
  auxPointer = writeMat;
  writeMat = readMat;
  readMat = auxPointer;
}
```

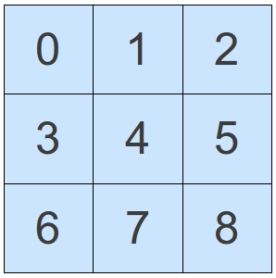

Figura 6.1: El cuadrado de la imagen representa la matriz a la cual se quiere aplicar la operación *stencil* de 5 puntos. Cada uno de los cuadrados de menor tamaño representan los subproblemas que se derivan de querer dividir la matriz en 3 partes en cada una de las dimensiones, dando un total de 9 subproblemas. Los números representan el identificador (rango) que MPI da a cada uno de estos subproblemas para poder identificarlos y conocer qué porción de los datos tienen que trabajar. En el caso del *stencil* de 27 puntos, también se tendría que indicar el número de divisiones que quieren hacerse en la dimensión de los planos.

emplear. Las dimensiones de la matriz se leen del mismo fichero donde ésta se encuentra, mientras que el número de hilos que se van a emplear viene dado por el usuario en valores diferentes, que son el número de divisiones que se van a hacer en la matriz para crear los subproblemas en todas las dimensiones (Figura 6.1).

Aunque también sería posible un modelo en el que se dividiera el trabajo en subproblemas basándose solo en la dimensión con mayor salto entre elementos (filas para el *stencil* de 5 puntos y planos para el de 27), esta opción no sería realista cuando se utilizan grandes volúmenes de datos que requieren que el problema sea enviado a diferentes máquinas.

Otra de las razones por la que se permite esta división es por posibles mejoras en el rendimiento. Si bien es cierto que los mecanismos de predicción hardware para traer datos a memoria de forma temprana se ven beneficiados cuando no hay saltos en la dimensión de salto unitaria (columnas en todos los casos), el tener filas o planos compuestos por muchos elementos y que el total de estos sean potencias de dos, puede conllevar conflictos en memorias cache de baja asociatividad, ya que entonces dos filas o planos diferentes que necesitan ser accedidos en el mismo cálculo podrían estar situados en el mismo bloque de cache (esto puede evitarse con relleno en alguna de las dimensiones).

Una vez está claro qué parte del trabajo tiene que hacer cada uno de los hilos de ejecución, el mecanismo para hacer que cada uno tenga su propio subproblema preparado se pone en marcha. En primer lugar, sabiendo ya cuales son las dimensiones de las matrices que contienen los subproblemas, se asigna memoria

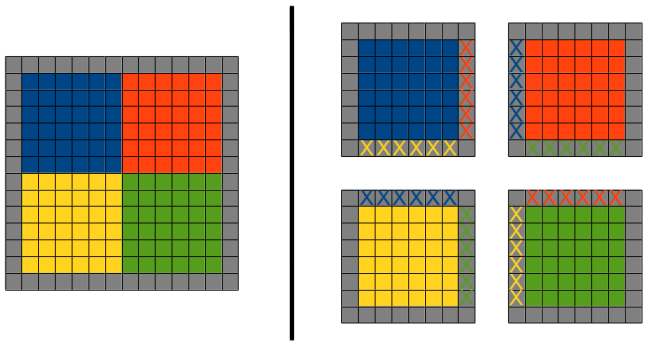

Figura 6.2: En la imagen se ilustra como un problema que es dividido en 4 partes requiere.

a cada hilo para crear dos de ellas, la de lectura y la de escritura. Esta simple operación, dependiendo de las políticas de asignación de memoria que tenga el sistema, eliminaría cualquier problema que pudiese existir por acceso a memoria no uniforme, ya que la memoria de cada hilo se habrá creado en módulos de memoria afines al procesador donde se ejecuta. Siempre y cuando el hilo no cambie de procesador físico durante toda la ejecución. Nótese que será necesario asignar a cada subproblema dos planos, filas y columnas adicionales que contengan los datos que hacen frontera con subproblemas adyacentes (Figura 6.2).

En segundo lugar, y utilizando las funciones de comunicación que otorga la API de MPI, se procedería a enviar desde el subproceso 0 (que es el que inicialmente lee el contenido de la matriz desde el fichero) los datos que corresponden a cada uno de los subprocesos. Si bien está tarea podría optimizarse, se ha decidido no hacerlo y mantener un modelo de comunicación síncrono y secuencial entre los procesos, ya que la preparación de los datos no se incluye en el cálculo del tiempo de ejecución y es una tarea que se realiza una sola vez.

Cuando cada hilo tenga sus correspondientes matrices inicializadas correctamente, es posible iniciar el algoritmo con OpenMP independientemente en cada uno. El único cambio que introduce MPI en el núcleo del Algoritmo 6.1 es la sincronización al final de cada iteración del bucle externo. Es necesario que los procesos que comparten planos, filas o columnas con los procesos adyacentes sincronicen sus datos antes de empezar una nueva iteración en el tiempo (Figura 6.3).

Este proceso de sincronización, representa una fracción despreciable del tiempo de ejecución del algoritmo. Esto se ha comprobado no solo en el sistema usado para este trabajo, sino también en un *cluster* con nodos independientes, donde se ha probado comunicación asíncrona y envío temprano de datos para solapar cálculo y sincronización. Puesto que se vio que esta linea de optimización no iba

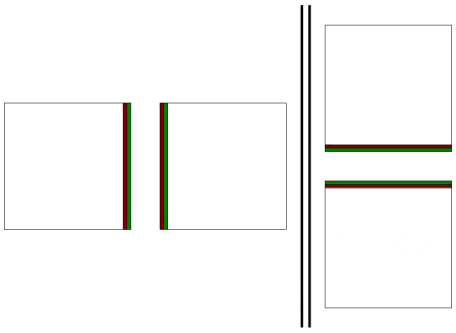

Figura 6.3: En las imágenes pueden verse ejemplos de comunicación para el *stencil* de 5puntos. La figura de la izquierda es la representación de dos subproblemas adyacentes en diferentes columnas y los datos que deben intercambiar entre ellos al finalizar cada iteración. En la figura de la derecha dos subproblemas adyacentes en diferentes filas y los datos que deben intercambiar. En rojo, los datos que el primer hilo debe enviar al segundo y donde los almacenará éste. En verde, los datos que el segundo hilo debe enviar al primero y donde los almacenará éste. En el caso del *stencil* de 27 puntos, en lugar de vectores unidimensionales, se compartirán planos.

Algoritmo 6.4 Representación a alto nivel del algoritmo de *stencil* con MPI. Este mismo esquema es ejecutado por cada hilo.

```
stencil_MPI:
  inicializar_valores();
  if (id_hilo == 0)
    leer_matriz_de_fichero();
  fi
  crear_matrices_locales();
  calculo_subproblema_stencil();
  finalizar_procesos();
```

a aportar ningún beneficio, se decidió descartar las optimizaciones y la inclusión de experimentos en un *cluster* distribuido en el trabajo.

Finalmente, y una vez el cálculo ha acabado, son necesarios también cambios para mostrar el resultado correctamente, donde cada subproblema tiene su parte del resultado, así como en el computo del tiempo de ejecución, ya que cada subproblema ha utilizado su propio contador, por lo que se utilizará como valido el que tenga mayor tiempo de ejecución (Algoritmo 6.4).

6.2.3. Pthreads

La implementación híbrida de MPI y OpenMP cubre todo el rango de actuación que se puede exigir para paralelizar un problema. Sin embargo, pueden

existir ocasiones en las que por una razón u otra librerías como MPI no estén disponibles en el sistema y OpenMP no nos ofrezca suficiente versatilidad a la hora de paralelizar el código. Esto hace de Pthreads una muy buena opción, ya que es la implementación de un estándar incorporado en la mayoría de sistemas UNIX. Por estas razones se ha creído conveniente desarrollar una implementación con esta librería, para profundizar en el aprendizaje de su metodología y observar qué resultados se obtienen.

A diferencia de MPI, los diferentes hilos no se crean desde la linea de comandos, sino que se tienen que crear y gestionar mediante código una vez el programa se está ejecutando. Esto crea un entorno de hilos maestros y esclavos y cambia algunos detalles de la implementación, pero el enfoque que se utiliza aquí será parecido al que se ha empleado con MPI. A pesar de que estén enfocados a entornos diferentes.

Inicialmente, gestionaremos desde el hilo maestro la cantidad de hilos y las dimensiones de los subproblemas, entonces se crearan los diferentes hilos de ejecución y cada uno de ellos se encargará de asignar su propio espacio de memoria, cosa que se hará en una estructura común a todos los hilos para que estos tengan constancia de en que posición de memoria se encuentran los datos del resto de hilos (aquí no se dispone de librerías que permiten la comunicación entre procesos; es necesario tener la información del resto de procesos en variables globales a todos los hilos). Como forzamos que cada hilo asigne su propia memoria, en este caso también evitamos que existan problemas por el acceso a memoria no uniforme. Destacar que a diferencia de la implementación anterior, solo se ha implementado la división en subproblemas a nivel de filas (para el *stencil* de 5 puntos) o planos (para el de 27 puntos), ya que aquí tratamos con memoria compartida y las razones para ofrecer división del problema en todas las dimensiones no se aplican (Figura 6.4).

La sincronización entre iteraciones sigue el mismo concepto que con MPI, ya que cada hilo ha asignado sus propios datos y por lo tanto no se encuentran en posiciones de memoria consecutivas. Es por eso que conocer la localización de los datos del resto de hilos a través de la estructura compartida creada para tal efecto se hace necesario (Algoritmo 6.5).

6.2.4. Mapeo de hilos a procesadores

En secciones anteriores se ha explicado como se van a utilizar las diferentes librerías para paralelizar el problema, pero una vez los hilos de ejecución se han encontrado, no se ha detallado ningún sistema para controlar donde exactamente se están ejecutando dichos hilos.

El sistema que se utiliza en este trabajo consta de un nodo con 2 procesadores

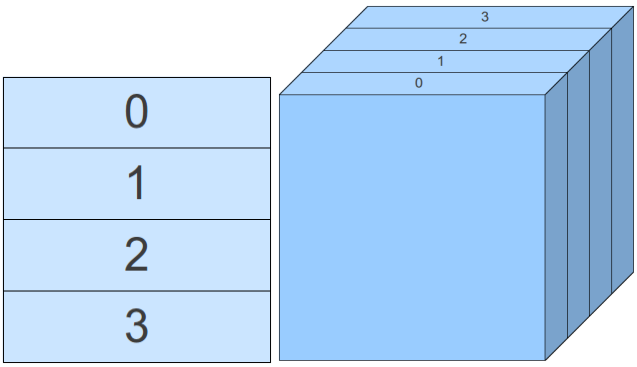

Figura 6.4: La figura de la izquierda representa la matriz a la cual se quiere aplicar la operación *stencil* de 5 puntos con el algoritmo que utiliza Pthreads utilizando 4 hilos, mientras que la figura de la derecha representa la misma situación para la operación *stencil* de 27 puntos. Los números representan el identificador que se da a cada uno de estos subproblemas para poder identificarlos y conocer con que porción de los datos tienen que trabajar.

Algoritmo 6.5 Representación a alto nivel del algoritmo de *stencil* con Pthreads. Este esquema es ejecutado por el hilo principal, que creará los hilos que a su vez ejecutarán la *funcion_hilo()* definida posteriormente.

```
stencil_Pthreads:
  leer_parametros();
  if (id_hilo == 0)
    leer_matriz_de_fichero();
  fi
  crear_hilos();

funcion_hilo:
  crear_matrices_locales();
  calculo_subproblema_stencil();
  finalizar_proceso();
```

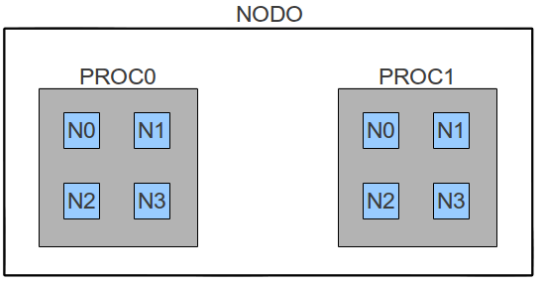

Figura 6.5: Diagrama del sistema hardware que se utiliza en este trabajo.

físicos, cada uno de los cuales tiene 4 núcleos que son capaces de ejecutar 1 hilo a la vez (Figura 6.5). Esto da un total de 8 hilos en paralelo si se quieren aprovechar los recursos al máximo.

En cuanto a la jerarquía de memoria, cada procesador tiene su propio bus de datos a sus módulos de memoria locales, aunque es posible acceder a cualquier módulo de memoria desde cualquier procesador con cierta perdida de rendimiento. Esto quiere decir que 2 hilos pueden estar situados cada uno en uno de los procesadores y estar trabajando de forma totalmente independiente al otro sin que existan interferencias en el uso del bus. Pero si esos 2 hilos están trabajando en el mismo procesador tendrán que compartir el bus de datos y, por lo tanto, cabe la posibilidad de que dicho bus no pueda proporcionar suficientes datos para los dos hilos.

Esta situación motiva la necesidad de controlar donde se mapean cada uno de los hilos durante la ejecución de nuestros algoritmos, ya que no se conoce como el hardware mapea los hilos por defecto y puede resultar en perdidas de rendimiento. En el caso de este trabajo, solo es necesario asignar hilos a nivel de procesador, y no es importante mapear cada hilo a un núcleo determinado, por lo que se dejará que el hardware tome la decisión a nivel de núcleo. Para observar pues como el mapeo de los hilos afecta al rendimiento, se han buscado mecanismos para mapear de forma manual los hilos a los procesadores en tiempo de ejecución.

En el caso de MPI, el propio entorno de ejecución de OpenMPI permite mapear los hilos creados a nivel de nodo, procesador o núcleo, ya sea mediante instrucciones en la linea de comandos o mediante ficheros donde se establece una política de mapeo, llamados *rankfiles*.

Puesto que con Pthreads no se puede utilizar este mismo sistema, será necesario recurrir a librerías adicionales para mapear los procesos. GNU Linux dispone de la librería que se encuentra definida en *sched.h*, pero las funcionalidades que da son pobres y solo es capaz de mapear procesos a nivel de núcleo. Existe otra librería llamada *libnuma*, que se construye sobre la librería anterior,

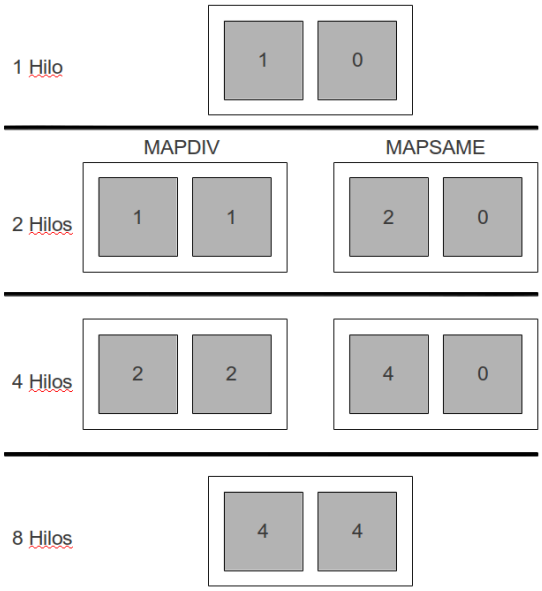

Figura 6.6: Ilustración de los dos modelos de mapeo y como se reparten los hilos en cada caso.

que permite trabajar a nivel de nodo o procesador, es por eso que se utilizará *libnuma* en este trabajo, porque trabajando a nivel de procesador se facilita la tarea.

Los hilos creados con OpenMP heredarán el mapeo del hilo de MPI del que proceden. Sin embargo, no se podrán repartir los hilos de OpenMP entre distintos procesadores, por lo que OpenMP solo creará hilos dentro del mismo procesador. Esto es así porque con OpenMP no tenemos control suficiente como para realizar este mapeo desde el código.

Así pues, una vez se conoce como se va a implementar el mapeo de hilos, en la Figura 6.6 se ilustran los dos tipos de mapeo se utilizarán: uno que reparte los hilos entre los dos procesadores, llamado MAPDIV; y otro que intenta mapearlos todos en el mismo, llamado MAPSAME. Cabe destacar que diferenciar entre ambos solo tiene sentido cuando se utilizan 2 o 4 hilos, ya que con 8 hilos se hace necesario utilizar los dos procesadores si se establece la política de un solo hilo por núcleo.

El hecho de mantener MAPSAME a pesar de conocer que el rendimiento con MAPDIV siempre será igual o mejor, ya que se repartirá la carga de datos entre dos buses independientes, es solamente para ver como afecta este fenómeno a los diferentes *stencils*.

6.3. Optimizaciones en el cálculo

En la sección anterior, se han descrito la descomposición del problema en subproblemas o la asignación de memoria en cada hilo para evitar problemas de acceso a memoria no uniforme. Dichas acciones son parte de las optimizaciones que se aplican al algoritmo, pero están fuera del núcleo de este. En esta sección se tratan las optimizaciones aplicadas en el núcleo del algoritmo, dentro de los bucles donde se recorre la matriz de datos.

6.3.1. Desenrollado de bucles

Esta optimización resulta muy común en problemas donde se procesan los elementos uno a uno. Tiene como objetivo intentar aprovechar recursos como los registros, evitar accesos a memoria repetidos y reducir el número de instrucciones.

Al ser una optimización bastante común y relativamente simple de lograr, algunos compiladores la aplican automáticamente con ciertos *flags* de compilación, pero hemos podido comprobar que en las implementaciones de los dos problemas el compilador no es capaz de aplicarla automáticamente, por lo que se hace necesario aplicarla a mano. Lo que se ha hecho pues, es aumentar el número de elementos que calcula el bucle más interno (el que recorre las columnas) a 2 y a 4 por iteración.

Por si sola, no se espera que esta optimización suponga una gran mejora en el rendimiento, aunque es posible que esto sí que ocurra por el mejor aprovechamiento de los registros de cada procesador. Lo importante sin embargo es que abre la puerta a otra optimización que es la eliminación de subexpresiones comunes, detallada a continuación.

6.3.2. Eliminación de subexpresiones comunes

Cuando se desenrollan bucles y varios elementos consecutivos se calculan en la misma iteración, algunas subexpresiones de los diferentes elementos pueden coincidir. Se abre entonces la posibilidad de calcular estas expresiones una sola vez para reducir la cantidad de trabajo del procesador.

En el caso del *stencil* de 5 puntos, existen pocos elementos en común en dos elementos consecutivos, por lo que no se espera de esta optimización una reducción importante del tiempo de ejecución, motivado también por el hecho de que la intensidad computacional de este problema es muy baja y que, por lo tanto, las optimizaciones cuyo objetivo es reducir el tiempo de proceso tienen menor impacto en el resultado.

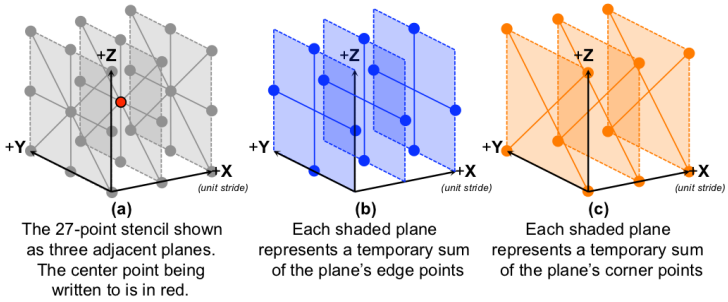

Figura 6.7: En la figura, extraída de la tesis de Datta [3], se ilustra qué datos son compartidos por elementos adyacentes para poder sumarlos una sola vez, consiguiendo una reducción importante del número de operaciones en coma flotante una vez se ha desenrollado el bucle.

Sin embargo, es posible que en el *stencil* de 27 puntos sí que se traduzca en una reducción importante del tiempo de ejecución, ya que elementos en columnas adyacentes comparten hasta nueve elementos en el cálculo, por lo que se pueden eliminar una cantidad considerable de operaciones una vez se ha desenrollado el bucle interno. En la Figura 6.7 se muestra qué elementos pueden agruparse y sumarse solo una vez para ser utilizados por elementos adyacentes.

6.3.3. Vectorización

Es posible plantear el problema de forma que se puedan utilizar las operaciones que ofrece SSE, sin embargo hay que llevar a cabo modificaciones en el código algo más profundas que con el resto de optimizaciones. Es por eso que está optimización solo se aplica al *stencil* de 5 puntos, ya que el *stencil* de 27 puntos es más complejo. Está decisión también está motivada por los pobres resultados que se han visto tras aplicarla al *stencil* de 5 puntos y que se detallan en el capítulo de resultados.

En primer lugar, es necesario alinear los datos del inicio de cada fila de las matrices que contienen los subproblemas a 128 bits, por lo que se requiere de cambios en la asignación de memoria para asegurar que se añade relleno al final de cada fila. También serán necesarias modificaciones en los bucles y los accesos a la matriz para que tengan en cuenta el relleno añadido, que no debe computarse como el resto de los elementos.

En segundo lugar, hay que tener en cuenta el número de elementos que hay que procesar a la vez para que esta optimización sea posible. En este caso, tenemos que cada elemento ocupa 4 bytes, por lo que hay que calcular a la vez 4 elementos para utilizar los registros vectoriales de 16 bytes que se utilizan con estas instrucciones.

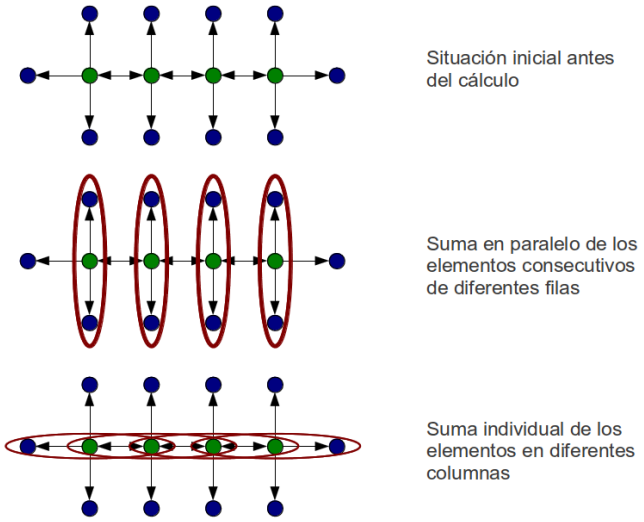

Figura 6.8: Las figuras representan el nuevo enfoque que se utiliza para hacer el cálculo de 4 elementos a la vez con operaciones vectoriales. La imagen superior muestra la situación inicial de los 4 elementos que se van a calcular (en verde) y los elementos adyacentes a estos que se consultan durante el cálculo (en azul). En la imagen del medio se muestra cual es el uso que se da de las instrucciones vectoriales. Lo que se hace en concreto es, aprovechando que los elementos marcados en cada una de las lineas son consecutivos en memoria, se suman de 4 en 4 utilizando las instrucciones vectoriales. En la última imagen, se ve el último paso del cálculo que consistiría en sumar individualmente y con instrucciones no vectoriales los elementos que se encuentran a izquierda y derecha de cada elemento. Por último, y dado que todas las operaciones se llevan a cabo sobre un registro vectorial y no sobre la matriz de escritura, sería necesario copiar el resultado a dicha matriz.

Por último, hay que reconsiderar la forma en la que se lleva a cabo el cálculo de los elementos. En este caso, puede hacerse de forma sencilla como se muestra en la Figura 6.8 y el Algoritmo 6.6.

6.3.4. Cache Bypass

Al trabajar con una cache tipo *write allocate*, toda escritura que se hace a memoria requiere de una lectura previa de dicha posición (en caso de que haya un fallo de cache). Dado que el dato que se está escribiendo no será necesario consultarlo de forma temprana, esta lectura resulta en una transferencia de memoria innecesaria.

Desgraciadamente, no existen mecanismos en el conjunto de instrucciones x86 para hacer una escritura en memoria que evite esta lectura, pero si que existen instrucciones para evitarlo cuando se utilizan instrucciones SSE.

Algoritmo 6.6 En esta sección del código podemos ver el bucle interno del algoritmo una vez se le aplica vectorización. Puede verse como inicialmente se asignan los punteros para que apunten a las direcciones de memoria correctas, se hacen las sumas con operaciones vectoriales, se suma individualmente los elementos restantes al resultado y, finalmente, se multiplica por el coeficiente (también a todos a la vez) y se guarda el resultado en la matriz de escritura.

```
#define GET(I,J) readMat[(I)*cols + (J)]
#define PUT(I,J) writeMat[(I)*cols + (J)]
// m0_2 es un registro vectorial inicializado a 0.2f

union floatSSE {
  __m128 m;
  float f[4];
} aux;
__m128 *srcA, *srcB, *srcC, coeff = m0_2;
float aux1, aux2;

// i es la fila actual
for (j=4; j<cols-4; j=j+4) {
  srcA = (__m128*) &GET(i-1,j);
  srcB = (__m128*) &GET(i,j);
  srcC = (__m128*) &GET(i+1,j);

  aux.m = _mm_add_ps(*srcA, *srcC);
  aux.m = _mm_add_ps(*srcB, aux.m);

  aux1 = GET(i,j+1);
  aux2 = GET(i,j+2);
  aux.f[0] += GET(i,j-1) + aux1;
  aux.f[1] += GET(i,j) + aux2;
  aux.f[2] += aux1 + GET(i,j+3);
  aux.f[3] += aux2 + GET(i,j+4);

  aux.m = _mm_mul_ps(aux.m, coeff);

  _mm_stream_ps(&PUT(i,j), aux.m);
}
```

El hecho de que sea necesario utilizar instrucciones SSE es un inconveniente, pues para poder llevar a cabo esta optimización se necesita que los datos estén alineados y hay que almacenarlos de cuatro en cuatro, ya que la operación utiliza registros vectoriales de 16 Bytes. Es por eso que esta optimización solo se aplica a la versión del código optimizada con instrucciones vectoriales y, por lo tanto, tampoco se aplica al *stencil* de 27 puntos.

En el Algoritmo 6.6 se utiliza la operación *_mm_stream_ps()* para escribir los datos directamente en memoria saltándose la política *write allocate* de la memoria cache.

Capítulo 7

Resultados en plataformas *multi-core*

En este capítulo se analizan los resultados obtenidos tras llevar a cabo pruebas de rendimiento con las implementaciones para entornos *multi-core* descritas en el capítulo anterior.

7.1. *Stencil* de 5 puntos

Siguiendo la metodología mencionada en el capítulo 5, se han llevado a cabo las ejecuciones de las diferentes versiones de este algoritmo. Con el fin de obtener un tiempo lo suficientemente significativo, se ha decidido emplear como base una matriz de datos de 256 MB (8192 filas por 8192 columnas) y realizar un total de 500 iteraciones en el tiempo. En la siguiente sección mostraremos cómo estos parámetros pueden fijarse ya que las variaciones en éstos tienen un efecto proporcional y totalmente predecible sobre el tiempo de ejecución del algoritmo.

7.1.1. Escalabilidad

Se han llevado a cabo unos experimentos para demostrar cómo afectan los diferentes parámetros al tiempo de ejecución del algoritmo, con el objetivo de poder fijar algunos parámetros para poder experimentar con las optimizaciones.

7.1.1.1. Iteraciones

En primer lugar, comprobaremos cómo afecta el número de iteraciones en el tiempo que se llevan a cabo al tiempo de ejecución del algoritmo. Ya que en cada iteración se hace exactamente el mismo trabajo (recorrer todos los elementos de

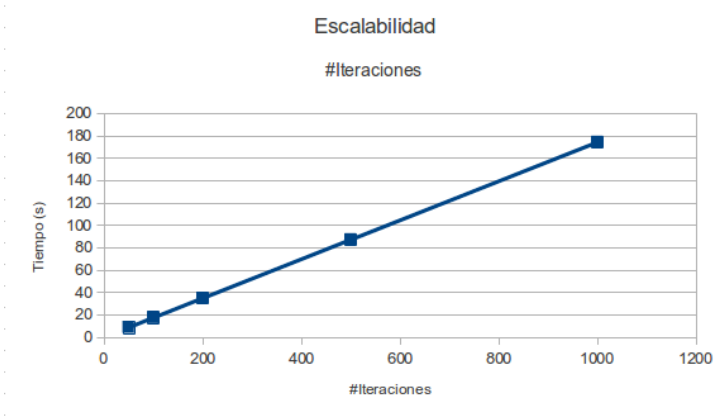

Figura 7.1: En la figura se observa el tiempo de la ejecución secuencial de un *stencil* de 5p con una matriz de datos de 256 MB. Los diferentes valores en el eje de las abscisas son el número de iteraciones en el tiempo que se llevan a cabo. Puede observarse que el aumento del tiempo de ejecución es lineal cuando se aumenta el número de iteraciones y que, por lo tanto, puede concluirse que el rendimiento es el mismo en todos los casos.

la matriz y llevar a cabo el mismo cálculo en cada uno de ellos), se espera que el tiempo de ejecución aumente de forma proporcional al número de iteraciones.

Aunque la predicción es correcta, como podrá comprobarse en el resultado mostrado en la Figura 7.1, hay que destacar que durante los primeros experimentos esto no fue así debido a problemas con la composición de la matriz que desencadenaban cálculos con números *denormales* y que disminuían el rendimiento a partir de cierto número de operaciones. En el Apéndice A se explica con detalle este problema y como se ha solucionado.

7.1.1.2. Tamaño de datos

En segundo lugar, comprobaremos que el tamaño de la matriz de datos también afecta de forma proporcional al tiempo de ejecución y que, por lo tanto, el rendimiento es el mismo independientemente de los datos.

En este caso, hay que matizar que aunque los resultados confirmen la suposición inicial como muestra la Figura 7.2, existen limites inferior y superior en este tamaño de datos. La matriz de datos más pequeña en los experimentos es de 16 MB (2048 filas por 2048 columnas), sin embargo, es muy posible que el rendimiento mejore considerablemente utilizando matrices de tamaño inferior al de la memoria cache.

Por otra parte, la matriz más grande utilizada es de 1 GB (16184 filas por 16184 columnas), que no es lo suficientemente para ocupar los 8 GB de memoria de los que dispone el sistema utilizado para los experimentos (recordando que

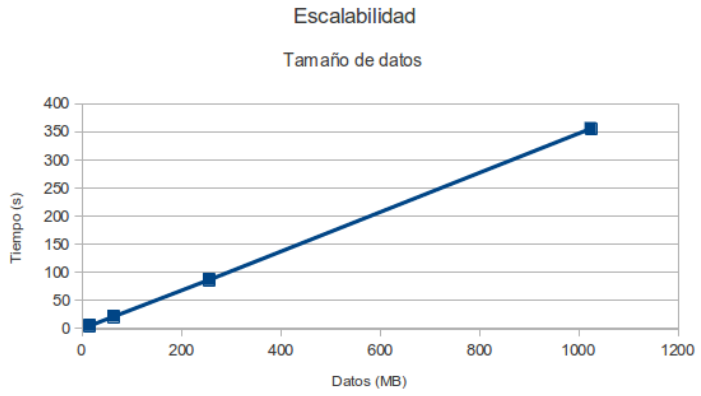

Figura 7.2: En la figura se observa el tiempo de la ejecución secuencial de un *stencil* de 5p durante 500 iteraciones en el tiempo. Los diferentes valores en el eje de las abscisas son los tamaños de matriz de datos utilizados para los experimentos. Puede observarse que el aumento del tiempo de ejecución es lineal cuando se aumenta el tamaño de la matriz y que, por lo tanto, puede concluirse que el rendimiento es el mismo en todos los casos.

hay que asignar espacio para dos matrices a la vez). En caso de superar las capacidades de memoria del sistema, sería necesario recurrir a memoria *swap* en disco, lo que provocaría tiempos de ejecución mucho mayores y una gran perdida de rendimiento.

7.1.2. Resultados de las optimizaciones en el código secuencial

Antes de empezar a experimentar con la paralelización del problema se harán pruebas para decidir cual de las optimizaciones aplicadas sobre el cálculo ofrece mejores tiempos. Puesto que las optimizaciones se aplican al cálculo, una vez se conozca qué código es el que da menor tiempo se pueden hacer los experimentos que conlleven paralelizar el problema solamente con ese código.

Tal y como se ve en la Figura 7.3 y el Cuadro 7.1, solo hay mejora en el rendimiento sobre el *stencil* de 5 puntos cuando se aplica desenrollado de bucle con dos elementos por iteración junto con eliminación de subexpresiones comunes. Cualquier otro cambio en el código deja inalterado o empeora el rendimiento, como el caso en el que se aplica desenrollado de bucle con cuatro elementos por iteración y que reduce el rendimiento a la mitad, aunque recupera parte del rendimiento cuando también se le aplica eliminación de subexpresiones comunes. Este comportamiento es posiblemente debido al uso excesivo de registros por parte del procesador, así como una mala gestión de estos.

Otro resultado negativo que se ha visto es el de aplicar vectorización al

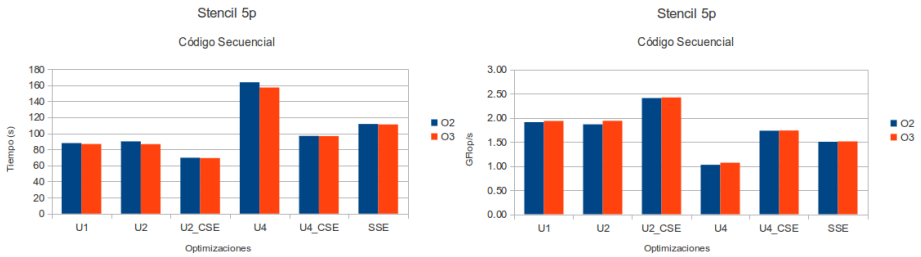

Figura 7.3: Gráficas que muestran los tiempos de ejecución y el rendimiento en GFlop/s de los diferentes códigos correspondientes a cada una de las optimizaciones aplicadas al cálculo del *stencil* y dependiendo también del *flag* de optimización. El primer resultado (U1) es el código original sin optimizar. La letra *U* (*unrolling*) seguida de un número indica el grado de desenrollado del bucle, mientras que CSE (*Common Subexpression Elimination*) se refiere a los código que aplican eliminación de subexpresiones comunes. Por último está el código SSE, que es el que aplica vectorización (es implícito que el desenrollado en este caso es de cuatro elementos).

código, que ha empeorado el rendimiento respecto al original significativamente. La posible explicación a este fenómeno es que, a pesar de que parte de los cálculos se hacen empleando las funciones vectoriales correspondientes, otros no se han podido beneficiar de ello, y la complejidad añadida a un código bastante simple y con una intensidad computacional tan baja han provocado este empeoramiento del rendimiento.

Resulta sorprendente que, en ningún caso, el compilador haya sido capaz de aplicar automáticamente las optimizaciones que se han aplicado a mano, ni siquiera con nivel de compilación O3, ya que el cálculo del *stencil* de 5 puntos no es tan complejo. Se puede observar que el código compilado con O3 no ofrece mejoras significativas respecto a O2, por lo que en posteriores experimentos se utilizará la versión O2.

Se puede concluir pues, que el código que presenta mejor resultado es el que aplica desenrollado de dos elementos por iteración y eliminación de subexpresiones comunes, por lo tanto ésta es la implementación que se utilizará para los experimentos en los que se paralelice el trabajo.

7.1.3. Resultados de la paralelización

El código resultante con todos los parámetros fijados ha sido ejecutado con diferente número de hilos de ejecución hasta llegar a los 8 que nos proporciona la máquina. Para cada cantidad de hilos, se evalúan los resultados con las distintas distribuciones de subproblemas posibles para cada modelo de programación paralela utilizado (Figura 7.4 y Cuadro 7.2).

1 Hilo	Tiempo (s)	GFlop/s	GStencil/s
U1_O2	87.76	1.91	0.38
U2_O2	89.91	1.87	0.37
U2_CSE_O2	69.58	2.41	0.48
U4_O2	163.49	1.03	0.21
U4_CSE_O2	96.72	1.73	0.35
SSE_O2	111.53	1.50	0.30

1 Hilo	Tiempo (s)	GFlop/s	GStencil/s
U1_O3	86.66	1.94	0.39
U2_O3	86.48	1.94	0.39
U2_CSE_O3	69.19	2.42	0.48
U4_O3	156.95	1.07	0.21
U4_CSE_O3	96.53	1.74	0.35
SSE_O3	110.98	1.51	0.30

Cuadro 7.1: Cuadro que muestra los tiempos de ejecución y el rendimiento, en GStencil/s y GFlop/s, de los experimentos llevados a cabo con el código secuencial y las distintas optimizaciones, la nomenclatura es la misma que en la Figura 7.3.

En los casos donde se utiliza el ejecutable híbrido de MPI y OpenMP, el valor junto OMP representa los subproblemas que se generan con OpenMP a partir de cada uno de los subproblemas de MPI, por ejemplo, en el caso del resultado para *MPI.4_ OMP.2*, se debe leer como que el problema inicial se ha dividido en 4 subproblemas independientes con MPI y que cada uno de ellos se ha dividido en 2 subproblemas más utilizando OpenMP.

También en consonancia con lo descrito en el capítulo anterior sobre la implementación, se hace distinción entre los dos tipos de reparto físico de los hilos, aunque cuando solo se utiliza OpenMP para paralelizar el problema esta distinción no existe, ya que los hilos con OpenMP se crean y se destruyen al final de cada iteración y no hay forma eficiente de mapearlos a distintos procesadores. El identificador DIV se utiliza cuando los hilos se reparten en diferentes procesadores físicos y SAME cuando se mapean todos los hilos al mismo procesador. Cuando se utilizan los 8 núcleos, siempre se reparten los hilos entre ambos procesadores pues cada uno tiene capacidad para 4.

Se observa que cuando se utilizan 2 núcleos la diferencia entre mapearlos físicamente en el mismo o distintos procesadores no tiene un efecto muy importante en el rendimiento, esto es debido a que un mismo procesador físico es capaz de mantener el tráfico de datos necesario para alimentar a 2 hilos de ejecución independientes. La distribución de hilos entre MPI, OpenMP o Pthreads no tiene un impacto importante en el rendimiento, aunque puede apreciarse que usar MPI para la distribución de los 2 hilos da tiempo de ejecución ligeramente inferiores.

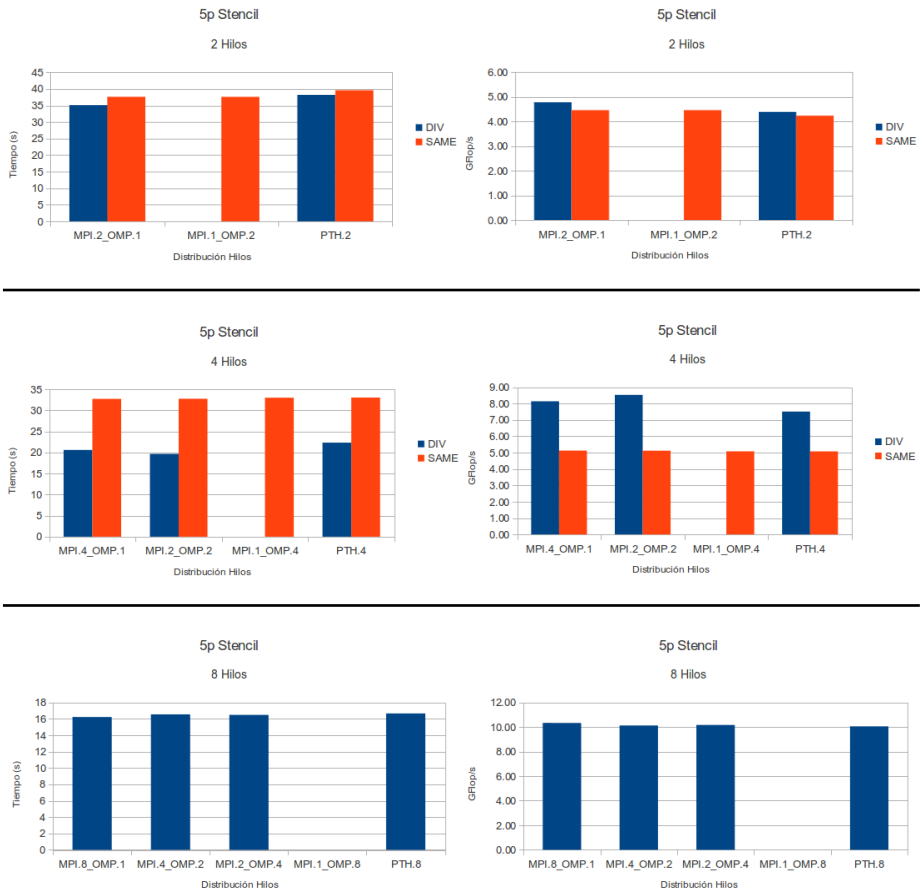

Figura 7.4: Gráficas que muestran los tiempos de ejecución y el rendimiento en GFlop/s de los experimentos llevados a cabo con varios hilos de ejecución. No se muestran datos para el caso de 8 hilos con OpenMP porque estos desvirtuarían el gráfico, el valor de estos resultados puede comprobarse en la tabla.

2 Hilos – DIV	Tiempo (s)	GFlop/s	GStencil/s
MPI.2_OMP.1	35.08	4.78	0.96
MPI.1_OMP.2	NA	NA	NA
PTH.2	38.2	4.39	0.88

2 Hilos – SAME	Tiempo (s)	GFlop/s	GStencil/s
MPI.2_OMP.1	37.61	4.46	0.89
MPI.1_OMP.2	37.6	4.46	0.89
PTH.2	39.59	4.24	0.85

4 Hilos – DIV	Tiempo (s)	GFlop/s	GStencil/s
MPI.4_OMP.1	20.61	8.14	1.63
MPI.2_OMP.2	19.67	8.53	1.71
MPI.1_OMP.4	NA	NA	NA
PTH.4	22.34	7.51	1.50

4 Hilos – SAME	Tiempo (s)	GFlop/s	GStencil/s
MPI.4_OMP.1	32.74	5.12	1.02
MPI.2_OMP.2	32.78	5.12	1.02
MPI.1_OMP.4	33.02	5.08	1.02
PTH.4	33.05	5.07	1.01

8 Hilos	Tiempo (s)	GFlop/s	GStencil/s
MPI.8_OMP.1	16.23	10.33	2.07
MPI.4_OMP.2	16.56	10.13	2.03
MPI.2_OMP.4	16.49	10.17	2.03
MPI.1_OMP.8	33.79	0.99	4.96
PTH.8	16.67	10.06	2.01

Cuadro 7.2: Cuadros que muestran los tiempos de ejecución y el rendimiento, en GStencil/s y GFlop/s de los experimentos llevados a cabo con varios hilos de ejecución.

Cuando se pasa a utilizar 4 núcleos, la diferencia entre el mapeo físico de estos cobra mucha importancia. Como puede observarse en los gráficos y tablas de resultados, mapear los 4 hilos en el mismo procesador en vez de poner dos en cada uno, provoca una perdida de rendimiento que sobrepasa el 50 %. Esto es así porque el bus de datos es incapaz de alimentar 4 hilos a la vez, por lo que dividir los hilos entre procesadores se hace una necesidad. Al igual que en el caso anterior, la distribución de los hilos entre procesos de MPI, OpenMP o Pthreads no resulta en cambios significativos en el rendimiento.

Utilizando la totalidad de los 8 núcleos disponibles en la máquina se encuentra la evolución esperada desde el paso anterior. Nótese el efecto que tienen los accesos a memoria no uniforme cuando se utilizan 8 hilos con OpenMP, donde la mitad de los hilos deben acceder a datos que se encuentran en un módulo de memoria no local. Este es el único caso en el que se dan accesos a memoria no local de todos los experimentos.

7.1.4. Conclusiones

El *stencil* de 5 puntos no da lugar a grandes optimizaciones. Al tratarse de una operación con una intensidad computacional baja es difícil conseguir una gran mejora en el rendimiento a través de optimizaciones en el cálculo. Además de ello, la mejora de asignar varios hilos al mismo procesador físico se ve limitada por la cantidad de datos que se pueden suplir a la vez a dicho procesador, como puede observarse en la Figura 7.5 y el Cuadro 7.3.

Entre la versión secuencial del algoritmo sin optimizar y la versión final del algoritmo repartiendo el trabajo entre los 8 núcleos físicos que ofrece el sistema utilizado, se ha conseguido un *Speed Up* de 5.41x (Cuadro 7.4), valor nada despreciable pero que muestra que si no se solventa el problema del ancho de banda no será posible mejorar más el rendimiento.

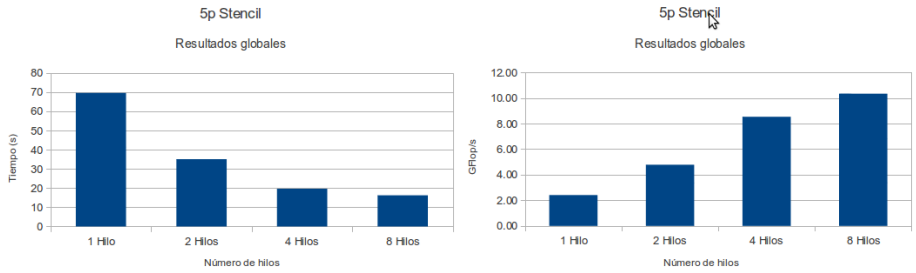

Figura 7.5: Gráfica que muestra los tiempos de ejecución y el rendimiento en GFlop/s de los mejores resultados obtenidos para cada cantidad de hilos.

5p Stencil	Tiempo (s)	GFlop/s	GStencil/s	Speed Up
1 Hilo	69.58	2.41	0.48	1x
2 Hilos	35.08	4.78	0.96	1.98x
4 Hilos	19.67	8.53	1.71	3.54x
8 Hilos	16.23	10.33	2.07	4.29x

Cuadro 7.3: Cuadro que muestra los tiempos de ejecución y el rendimiento, en GStencil/s, GFlop/s y Speed Up de los mejores resultados obtenidos sobre la versión optimizada del *stencil* de 5 puntos para cada cantidad de hilos.

5p Stencil	Tiempo (s)	GFlop/s	GStencil/s	Speed Up
Peor versión (1 Hilo y sin optimizar)	87.76	1.91	0.38	1x
Mejor versión (8 Hilos y optimizado)	16.23	10.33	2.07	5.41x

Cuadro 7.4: Cuadro que muestra los tiempos de ejecución y el rendimiento, en GStencil/s, GFlop/s y Speed Up del mejor resultado posible de la versión optimizada comparado con la implementación inicial del *stencil* de 5 puntos.

7.2. *Stencil* de 27 puntos

La metodología es la misma que la que se ha seguido con el *stencil* de 27 puntos. En este caso se ha fijado una matriz de datos también de 256 MB (258 caras por 512 filas por 512 columnas) y 100 iteraciones en el tiempo. En la siguiente sección mostraremos cómo, también en ese *stencil*, estos parámetros pueden fijarse ya que las variaciones tienen un efecto proporcional y predecible sobre el tiempo de ejecución del algoritmo.

7.2.1. Escalabilidad

La demostración en este caso es la misma que la que se ha llevado a cabo en el *stencil* de 5 puntos, por lo que las explicaciones pertinentes pueden consultarse en esa sección. En esta sección pues solo se mostrarán los datos que demuestran que es posible fijar el número de iteraciones en el tiempo y el tamaño de la matriz de datos a la hora de calcular el rendimiento del algoritmo.

7.2.1.1. Iteraciones

El número de iteraciones, al igual que en el caso anterior, hace aumentar el tiempo de forma proporcional, por lo que su valor no es relevante al calcular el rendimiento (Figura 7.6).

Tiene también el mismo problema que el *stencil* de 5 puntos cuando se utiliza una determinada composición de la matriz de datos, este fenómeno se detalla en el Apéndice A que se incluye al trabajo.

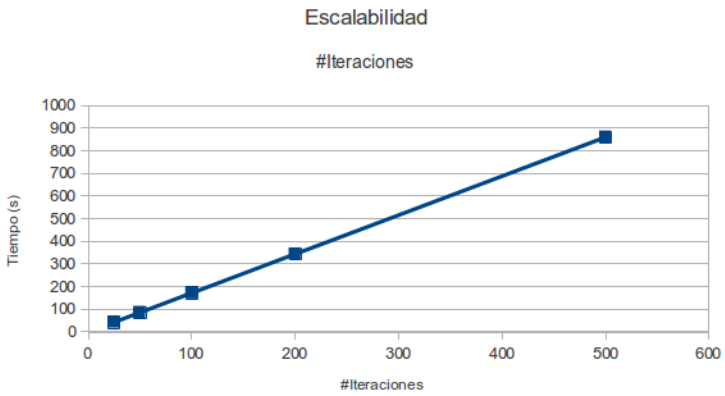

Figura 7.6: En la figura se observa el tiempo de la ejecución secuencial de un *stencil* de 27p con una matriz de datos de 256 MB. Los diferentes valores en el eje de las abscisas son el número de iteraciones en el tiempo que se llevan a cabo. Puede observarse que el aumento del tiempo de ejecución es lineal cuando se aumenta el número de iteraciones y que, por lo tanto, puede concluirse que el rendimiento es el mismo en todos los casos.

7.2.1.2. Tamaño de datos

Al igual que en el caso anterior, y como muestra la Figura 7.7, los tamaños de la matriz de datos que estén por encima del tamaño de memoria cache y por debajo de la cantidad de memoria total del sistema tiene un efecto predecible sobre el tiempo de ejecución del algoritmo.

7.2.2. Resultados de las optimizaciones en el código secuencial

Antes de empezar a experimentar con la paralelización del problema se harán pruebas para decidir cual de las optimizaciones aplicadas sobre el cálculo ofrece mejores tiempos. Puesto que las optimizaciones se aplican al cálculo, una vez se conozca que código es el que da menor tiempo se pueden hacer los experimentos que conlleven paralelizar el problema solamente con ese código.

Estos resultados son algo más positivos en cuanto a mejora conseguida que los del *stencil* de 5 puntos. En primer lugar, y dado que la operación utiliza un gran número de registros (requiere leer 27 floats por cada elemento), no hay un empeoramiento tan significativo cuando se procede al desenrollado de cuatro elementos como ocurre en el otro *stencil*. Aplicar aquí desenrollado de bucle no mejora en ningún caso el rendimiento, sin embargo hay una gran reducción del tiempo de ejecución cuando se aplica eliminación de subexpresiones comunes, ya que en este caso se reducen significativamente las operaciones realizadas (Figura

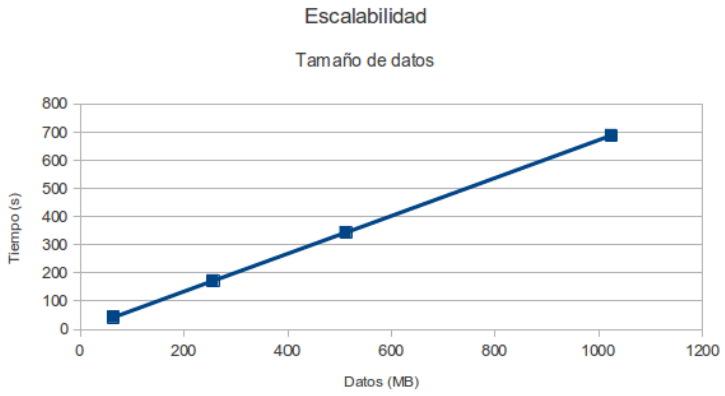

Figura 7.7: En la figura se observa el tiempo de la ejecución secuencial de un *stencil* de 27p durante 100 iteraciones en el tiempo. Los diferentes valores en el eje de las abscisas son los tamaños de matriz de datos utilizados para los experimentos. Puede observarse que el aumento del tiempo de ejecución es lineal cuando se aumenta el tamaño de la matriz y que, por lo tanto, puede concluirse que el rendimiento es el mismo en todos los casos.

7.8 y Cuadro 7.5).

La alta intensidad computacional de este *stencil*, hace que las mejoras aplicadas en el cálculo tengan un gran impacto. Puede verse como la mejora asciende a más del 80 % cuando se aplica eliminación de subexpresiones comunes al desenrollado de cuatro elementos.

Al igual que en el *stencil* de 5 puntos, el compilador demuestra no haber sido capaz de aplicar optimizaciones que mejoren el rendimiento del algoritmo al pasar de nivel de optimización O2 a O3, aunque aquí resulta algo más comprensible por el hecho de que este *stencil* es bastante más complejo que el de 5 puntos. Se utilizará código compilado con O2 para posteriores experimentos.

Se puede concluir pues, que el código que presenta mejor resultado es el que aplica desenrollado de cuatro elementos por iteración y eliminación de subexpresiones comunes, por lo tanto ésta es la implementación que se utilizará para los experimentos en los que se paralelice el trabajo.

7.2.3. Resultados de la paralelización

Al igual que con el *stencil* de 5 puntos, una vez fijados todos los parámetros (número de iteraciones, tamaño de la matriz de datos y código optimizado óptimo), se procede a calcular el tiempo de ejecución y el rendimiento con diferente número de hilos de ejecución hasta llegar a los 8 que nos proporciona la máquina. Para cada cantidad de hilos, se evalúan los resultados con las distintas distribuciones de subproblemas posibles para cada modelo de programación

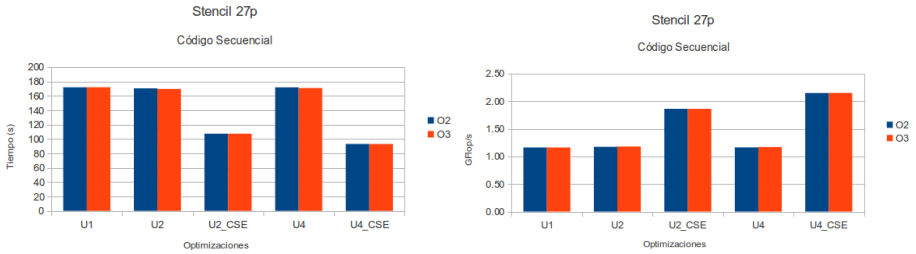

Figura 7.8: Gráficas que muestran los tiempos de ejecución y el rendimiento en GFlop/s de los diferentes códigos correspondientes a cada una de las optimizaciones aplicadas al cálculo del *stencil* y dependiendo también del *flag* de optimización. El primer resultado (U1) es el código original sin optimizar. La letra *U* (*unrolling*) seguida de un número indica el grado de desenrollado del bucle, mientras que CSE (*Common Subexpression Elimination*) se refiere a los código que aplican eliminación de subexpresiones comunes.

1 Hilo	Tiempo (s)	GFlop/s	GStencil/s
U1_O2	171.91	1.16	0.04
U2_O2	170.36	1.17	0.04
U2_CSE_O2	107.33	1.86	0.06
U4_O2	171.78	1.16	0.04
U4_CSE_O2	93.03	2.15	0.07

1 Hilo	Tiempo (s)	GFlop/s	GStencil/s
U1_O3	171.93	1.16	0.04
U2_O3	169.54	1.18	0.04
U2_CSE_O3	107.29	1.86	0.06
U4_O3	170.77	1.17	0.04
U4_CSE_O3	92.96	2.15	0.07

Cuadro 7.5: Cuadro que muestra los tiempos de ejecución y el rendimiento, en GStencil/s y GFlop/s, de los experimentos llevados a cabo con el código secuencial y las distintas optimizaciones, la nomenclatura es la misma que en la Figura 7.8.

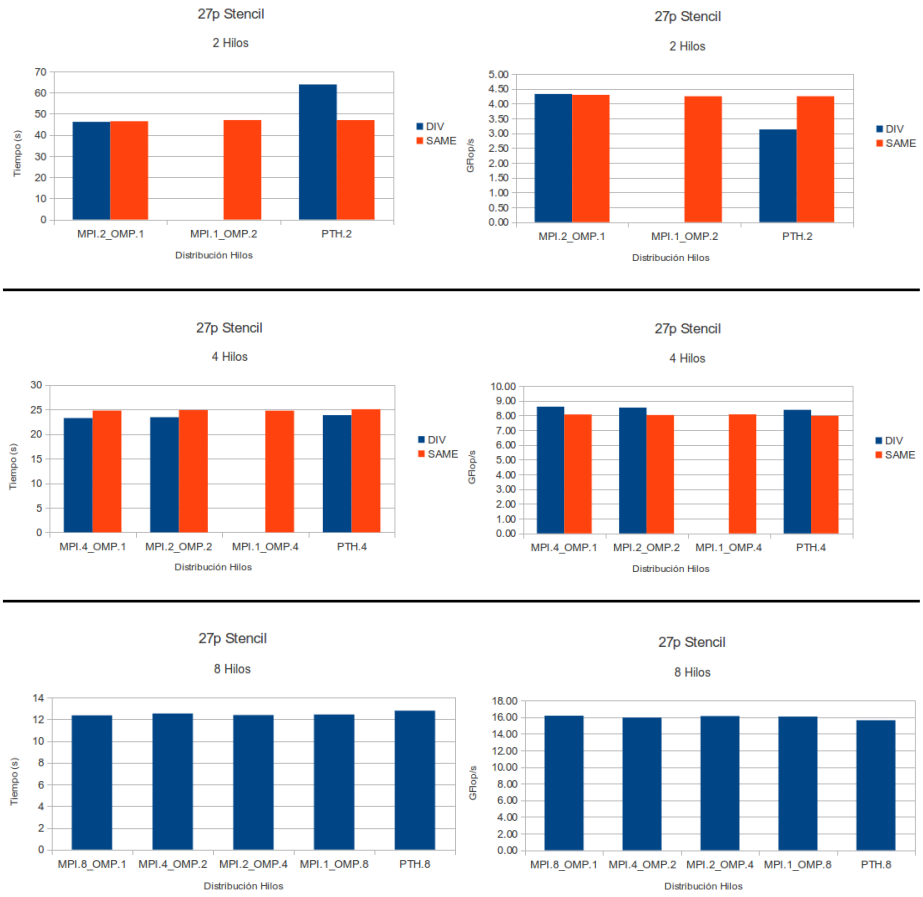

Figura 7.9: Gráficas que muestran los tiempos de ejecución y el rendimiento en GFlop/s de los experimentos llevados a cabo con varios hilos de ejecución. No se muestran datos para el caso de 8 hilos con OpenMP porque estos desvirtuarían el gráfico, el valor de estos resultados puede comprobarse en la tabla.

paralela utilizado (Figura 7.9 y Cuadro 7.6).

Los experimentos y la nomenclatura son los mismos que los establecidos para el *stencil* de 5 puntos en la sección anterior.

Otra gran diferencia de los resultados obtenidos por este *stencil* frente a los del *stencil* de 5 puntos es como afecta el mapeo físico de los hilos a los procesadores. Mientras en el *stencil* de 5 puntos se podían observar perdidas de rendimiento superiores al 50 % con 4 hilos cuando se mapeaban todos ellos al mismo procesador, en el caso del *stencil* de 27 puntos las perdidas están en torno al 5 %. Esto es así porque el *stencil* de 27 puntos no está hambriento de datos como ocurre con el de 5 puntos, por lo que hay suficiente ancho de banda para alimentar a 4 hilos que están mapeados en el mismo procesador.

En las ejecuciones para 2 hilos se observa una anomalía de la que por el

2 Hilos – DIV	Tiempo (s)	GFlop/s	GStencil/s
MPI.2_OMP.1	46.17	4.33	0.14
MPI.1_OMP.2	NA	NA	NA
PTH.2	63.85	3.13	0.10

2 Hilos – SAME	Tiempo (s)	GFlop/s	GStencil/s
MPI.2_OMP.1	46.48	4.30	0.14
MPI.1_OMP.2	47.03	4.25	0.14
PTH.2	47.01	4.25	0.14

4 Hilos – DIV	Tiempo (s)	GFlop/s	GStencil/s
MPI.4_OMP.1	23.24	8.60	0.29
MPI.2_OMP.2	23.41	8.53	0.28
MPI.1_OMP.4	NA	NA	NA
PTH.4	23.84	8.38	0.28

4 Hilos – SAME	Tiempo (s)	GFlop/s	GStencil/s
MPI.4_OMP.1	24.76	8.07	0.27
MPI.2_OMP.2	24.87	8.03	0.27
MPI.1_OMP.4	24.74	8.07	0.27
PTH.4	25.04	7.98	0.27

8 Hilos	Tiempo (s)	GFlop/s	GStencil/s
MPI.8_OMP.1	12.36	16.16	0.54
MPI.4_OMP.2	12.53	15.94	0.53
MPI.2_OMP.4	12.39	16.12	0.54
MPI.1_OMP.8	12.44	16.06	0.54
PTH.8	12.79	15.62	0.52

Cuadro 7.6: Cuadros que muestran los tiempos de ejecución y el rendimiento, en GStencil/s y GFlop/s de los experimentos llevados a cabo con varios hilos de ejecución.

momento no se ha encontrado explicación. Ocurre únicamente cuando se utiliza el código implementado con Pthreads y se mapea cada uno de los dos hilos en procesadores diferentes, y causa un tiempo de ejecución anormalmente alto y para nada esperado. Puesto que la implementación con Pthreads utiliza la libreria *libnuma* para mapear los hilos a procesadores (MPI lo hace desde su entorno de ejecución) y no se repite la anomalía para 4 y 8 hilos, resulta muy probable que se trate de algún error en dicha librería. Es conveniente pues descartar el resultado obtenido y quedarse únicamente con el que se obtiene con la implementación híbrida de MPI y OpenMP.

En las ejecuciones para 4 hilos lo destacable es que los diferentes tipos de mapeo de los hilos (DIV y SAME) no afecte significativamente al resultado. La explicación que se puede dar a ello es el hecho de que la intensidad computacional del *stencil* de 27 puntos es más alta y, por lo tanto, existe más un problema de optimización del tiempo de proceso que de el aprovechamiento de memoria.

Cuando se utilizan todos los núcleos disponibles para ejecutar 8 hilos, se ve un aumento que por primera vez en este *stencil* no llega a ser lineal respecto a la ejecución anterior, lo cual puede ser consecuencia de estar llegando al límite de datos que el bus de datos puede proporcionar a cada procesador. Sin embargo, no ocurre como en el caso del *stencil* de 5 puntos cuando todos los hilos son utilizados por OpenMP, ya que aquí la penalización en el rendimiento no se nota, pues el *stencil* está limitado por la CPU y no por el ancho de banda..

7.2.4. Conclusiones

El *stencil* de 27 puntos ha permitido que el aporte de las optimizaciones en el cálculo sea mucho más significativo que en el *stencil* de 5 puntos. Además de eso, se ha conseguido una escalabilidad casi lineal al aumentar el número de hilos, como se puede apreciar en la Figura 7.10 y el Cuadro 7.7, aunque esta ya no llegaba a ser lineal en el último tramo cuando se pasa a utilizar 8 hilos.

Por otra parte, y gracias a ese aporte de las optimizaciones en el cálculo, el *Speed Up* total tras comparar el tiempo de ejecución del algoritmo secuencial sin optimizar y el del optimizado utilizando 8 hilos, es mucho mayor que el obtenido en el *stencil* de 5 puntos y asciende a 13.91x (Cuadro 7.8). Está mejora conlleva la reducción de un orden de magnitud frente al código secuencial, lo cual es un dato muy positivo.

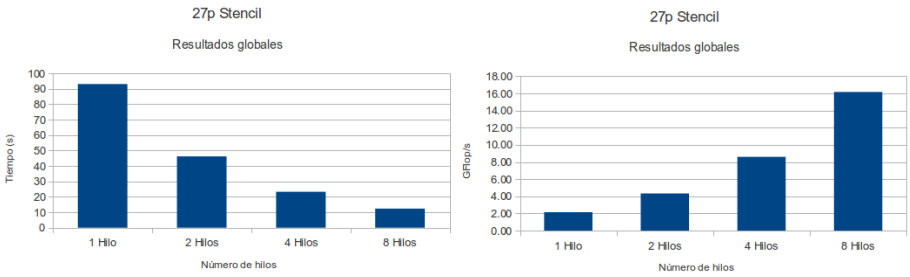

Figura 7.10: Gráfica que muestra los tiempos de ejecución y el rendimiento en GFlop/s de los mejores resultados obtenidos para cada cantidad de hilos.

27p Stencil	Tiempo (s)	GFlop/s	GStencil/s	Speed Up
1 Hilo	93.03	2.15	0.07	1x
2 Hilos	46.17	4.33	0.14	2.02x
4 Hilos	23.24	8.60	0.29	4.00x
8 Hilos	12.36	16.16	0.54	7.53x

Cuadro 7.7: Cuadro que muestra los tiempos de ejecución y el rendimiento, en GStencil/s, GFlop/s y Speed Up de los mejores resultados obtenidos sobre la versión optimizada del *stencil* de 27 puntos para cada cantidad de hilos.

27p Stencil	Tiempo (s)	GFlop/s	GStencil/s	Speed Up
Peor versión (1 Hilo y sin optimizar)	171.91	1.16	0.04	1x
Mejor versión (8 Hilos y optimizado)	12.36	16.16	0.54	13.91x

Cuadro 7.8: Cuadro que muestra los tiempos de ejecución y el rendimiento, en GStencil/s, GFlop/s y Speed Up del mejor resultado posible de la versión optimizada comparado con la implementación inicial del *stencil* de 27 puntos.

Capítulo 8

Implementaciones *many-core* y resultados

Este es el capítulo dedicado a evaluar el uso del entorno *many-core* con CUDA para los *stencils* de 5 y 27 puntos. También se analizarán los resultados de los experimentos a la vez que se describen las optimizaciones que se han aplicado.

8.1. Funcionamiento de CUDA

CUDA requiere de un enfoque para la resolución de problemas que dista mucho del utilizado para las implementaciones *multi-core*. El objetivo en este caso es crear pequeñas funciones llamadas *kernels*, que se ejecutan en hilos destinados para ello y que lleven a cabo una fracción muy reducida del problema. Está arquitectura permite que la creación de hilos y los cambios de contexto se hagan sin prácticamente penalización alguna, por lo que pueden crear y destruir miles de hilos sin temer que esto afecte al rendimiento. Lo importante es mantener la mayor simplicidad posible en el *kernel*.

Los hilos que ejecutan los *kernels* son agrupados en CUDA por bloques de hilos, que comparten ciertos recursos hardware. Estos bloques se crean en forma de matriz, y cada uno de ellos tiene ciertas variables especiales que indican al hilo en tiempo de ejecución el identificador de fila y columna correspondiente respecto al bloque al que pertenece. De la misma forma en que se agrupan los hilos en bloques de hilos, los bloques en si también se organizan en forma de matriz a nivel global, por lo que los hilos también disponen en tiempo de ejecución del identificador de fila y columna del bloque en el que se encuentran (Figura 8.1).

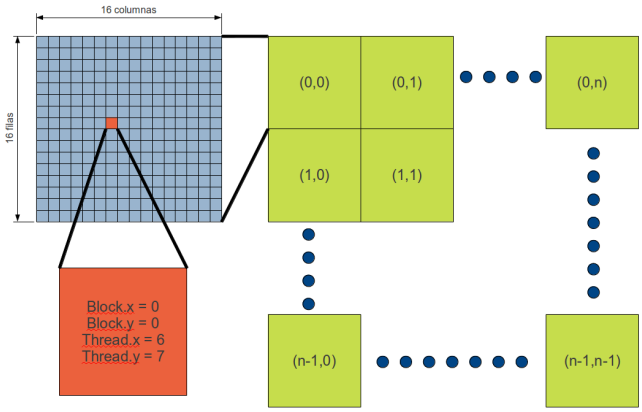

Figura 8.1: La figura ilustra como se trabaja con CUDA para paralelizar un problema. A la derecha se encuentra la matriz de bloques de hilo, donde cada elemento es un bloque que contiene una cantidad determinada de hilos repartidos en filas y columnas. Finalmente, puede verse que cada hilo puede conocer su posición dentro del problema gracias a los identificadores.

El modelo descrito es ideal para poder mapear de forma individual cada *kernel* a un elemento de una matriz bidimensional, ya que el programador puede configurar el tamaño de los bloques de hilos y de la matriz de bloques para que exista una correspondencia entre elementos de la matriz de datos y *kernels* que van a ejecutarse.

Los bloques de hilos no deben superar los 512 hilos en total, aunque los manuales de Nvidia recomiendan utilizar 128 o 256 bloques y no exceder ese tamaño. Tampoco es recomendable utilizar menos, ya que se corre el riesgo de no estar aprovechando los recursos disponibles. En cuanto a la matriz de bloques, aquí las restricciones son más laxas y es posible crear matrices de grandes dimensiones.

Entre los recursos que comparten los bloques de hilos, se encuentra la memoria compartida, que es una especie de memoria cache gestionada manualmente por el usuario y con tiempos de acceso casi tan pequeños como el de acceso a los registros locales de cada hilo. Este recursos es limitado y no pueden usarse más de 16 KB de memoria compartida en total por cada bloque de hilos. Existen otras alternativas de memoria en CUDA además de la memoria global de dispositivo y la compartida, pero no se han considerado necesarias para el problema con *stencils* que se implementa y por tanto no se profundiza en ellas.

8.2. Lineas generales para la implementación

Conociendo el funcionamiento básico de CUDA, es el momento de plantear las posibles implementaciones que se pueden hacer para el problema de *stencil*.

En primer lugar, se intentará definir la correspondencia entre la estructura de hilos que se crea con CUDA y la matriz de datos. Puesto que los mecanismos que ofrece CUDA para crear su estructura de bloques hilos son solamente bidimensionales (los bloques de hilos pueden ser tridimensionales, pero no la estructura que los contiene), será fácil hacer un mapeo para la matriz bidimensional del *stencil* de 5 puntos, pero para el *stencil* de 27 puntos eso no es posible, pues la matriz es tridimensional. Lo que se hará en el caso de la matriz tridimensional, es tan simple como tratar cada plano de la matriz como en el caso de 5 puntos, pero recorriendo los diferentes planos dentro del *kernel* mediante un bucle, con lo que se completarán todos los elementos de la matriz.

Por otra parte, y antes de entrar a detallar aspectos de las diferentes implementaciones, hay que dejar clara la diferencia entre algunos conceptos que aparecerán en la descripción de estas implementaciones y que, pese a poder ser parecidos, no hacen referencia a lo mismo. Dichos conceptos se listan a continuación en forma de glosario:

- Matriz de datos: Hace referencia a toda la matriz.
- Elementos computables de la matriz de datos: Son todos los elementos dentro de la matriz a los que se les aplica la operación *stencil*. Estos son todos los elementos menos los valores fijos que se encuentran en los bordes y que son solo de lectura.
- Elementos computables del bloque de hilos: Son todos los hilos del bloque de hilos de CUDA que se encargan de hacer el cálculo de un elemento, no forman parte de este grupo los hilos que solo tienen como objetivo traer un elemento de memoria de dispositivo.

La preparación del entorno antes de ejecutar los *kernels* en CUDA es muy similar en todos los casos y sigue una estructura como la mostrada en el Algoritmo 8.1.

En los siguientes capítulos, se muestran los métodos utilizados para mapear la matriz de datos con la estructura de hilos de CUDA.

8.2.1. Método directo (DIR)

Este método consiste en mapear la estructura de hilos de CUDA directamente sobre los elementos computables de la matriz de datos. Esto quiere decir que habrá una correspondencia única y directa entre cada elemento computable

Algoritmo 8.1 Algoritmo en alto nivel que muestra la preparación del entorno para empezar a ejecutar el algoritmo de *stencil*.

```
stencil_CUDA:
  leer_parametros();
  leer_matriz_de_fichero();
  crear_matrices_en_dispositivo();
  preparar_entorno_cuda();

  for (cur_time = 0; cur_time < total_time; cur_time++) {
    lanzar_kernels();
    intercambiar_matrices();
  }

  fin_algoritmo();
```

y cada hilo de CUDA, quedando los bordes no computables de la matriz fuera de la estructura de hilos de CUDA (Figura 8.2).

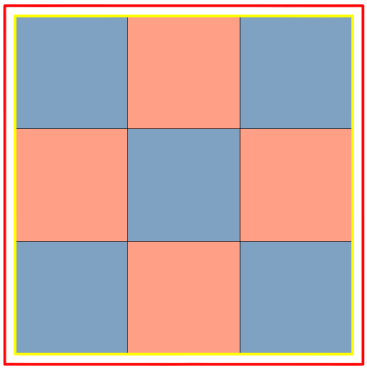

Figura 8.2: Mapeo directo de la estructura de hilos de CUDA y los elementos computables de la matriz de datos. La linea roja muestra el límite de la matriz de datos y la linea amarilla deja fuera a los bordes no computables de la matriz. Dentro de los elementos computables, se puede apreciar el esquema que siguen los bloques de hilos para mapearse sobre cada elemento, donde cada hilo llevará a cabo el cálculo de un elemento.

Una vez se sabe como están mapeados los hilos a los elementos y, por lo tanto, se conoce como calcular la posición relativa del elemento de un hilo dentro de la matriz, hay que pensar en como llevar a cabo el cálculo desde esa posición.

En primer lugar, es necesario traer de memoria de dispositivo a memoria compartida todos los elementos necesarios para el cálculo dentro de un bloque de hilos. De esta forma se consigue que el acceso a dichos elementos solo sea necesario una vez y se ahorran varios accesos a memoria de dispositivo. Sin embargo, hay que notar que los elementos que se sitúan en los bordes del bloque

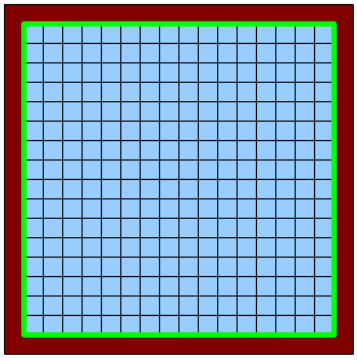

Figura 8.3: La figura ilustra la situación dentro de un bloque de hilos en el cálculo de un *stencil*. En azul se muestran los elementos dentro del bloque de hilos. Fuera del bloque de hilos, se muestra los elementos que también son necesarios para llevar a cabo el cálculo de todos los *stencils*, pues los elementos de los bordes dentro del bloque de hilos los requieren en el cálculo.

de hilos requieren de los elementos adyacentes a estos, que se encuentran fuera del propio bloque de hilos, por lo que también se hace necesario traer esos elementos antes de empezar con el cálculo. En este punto, está claro pues que para poder computar todos los elementos de un bloque de hilos, hace falta traer los elementos relativos a ese bloque de hilos más los elementos de las filas y columnas inmediatamente anteriores y posteriores a estos (Figura 8.3).

Para traer todos los elementos, es necesario incluir algún tipo de ramificación con condicionales dentro del kernel, lo cual es perjudicial para el rendimiento, ya que este tipo de núcleos consiguen su mejor rendimiento cuando no hay control de flujo en el código.

Antes de iniciar el cálculo, cuando cada hilo se ha encargado de traer los elementos necesarios de memoria de dispositivo a memoria compartida, hay que llevar a cabo una sincronización entre los hilos del bloque (que afecta poco al rendimiento, ya que CUDA está optimizado para este tipo de operaciones a nivel de bloque de hilos). Finalmente, se lleva a cabo el cálculo y se concluye la ejecución de este *kernel* hasta la siguiente iteración (Algoritmo 8.2). En el *kernel* del *stencil* de 27 puntos, hay que incluir un bucle que recorra todos los planos de la matriz accediendo a los elementos necesarios y haciendo los cálculos.

Está implementación es fácil de obtener y es quizás el método que más fácilmente se concibe para tratar el problema utilizando CUDA. Sin embargo tiene algún inconveniente como las ramificaciones en el código en la parte de la obtención de datos.

En los ejemplos de CUDA, hay un *stencil* de 25 puntos que está implementado siguiendo una filosofía muy similar a la que aquí se muestra [23].

Algoritmo 8.2 Algoritmo que muestra el *kernel* que calcula un elemento de un *stencil* de 5 puntos con el método directo, el código es el original antes de aplicar ninguna optimización.

```
#define BLOCKROWS 16
#define BLOCKCOLS 16

__global__ void compute_stencil(float *inMat, float *outMat,
                                const int matCols) {
  // Allocate shared memory
  __shared__ float sData[BLOCKROWS+2][BLOCKCOLS+2];

  // Matching matrix element using threablock relative values
  int matRow = blockIdx.y * BLOCKROWS + threadIdx.y + 1;
  int matCol = blockIdx.x * BLOCKCOLS + threadIdx.x + 1;
  int matIndex = matRow*matCols + matCol;

  // Compute relative index for sData matrix
  int sRow = threadIdx.y + 1;
  int sCol = threadIdx.x + 1;

  if (threadIdx.y == 0)
    // Populate first row and thread col of sData
    sData[0][sCol] = inMat[matIndex-matCols];
  else if (threadIdx.y == BLOCKROWS-1)
    // Populate last row and thread col of sData
    sData[BLOCKROWS+1][sCol] = inMat[matIndex+matCols];

  if (threadIdx.x == 0)
    // Populate thread row and first col of sData
    sData[sRow][0] = inMat[matIndex-1];
  else if (threadIdx.x == BLOCKCOLS-1)
    // Populate thread row and last col of sData
    sData[sRow][BLOCKCOLS+1] = inMat[matIndex+1];

  // Populate sData element of this thread
  sData[sRow][sCol] = inMat[matIndex];

  // Synchronise threadblock
  __syncthreads();

  // Compute stencil
  outMat[matIndex] = ALPHA * (sData[sRow-1][sCol] +
                              sData[sRow][sCol-1] +
                              sData[sRow][sCol] +
                              sData[sRow][sCol+1] +
                              sData[sRow+1][sCol]);
}
```

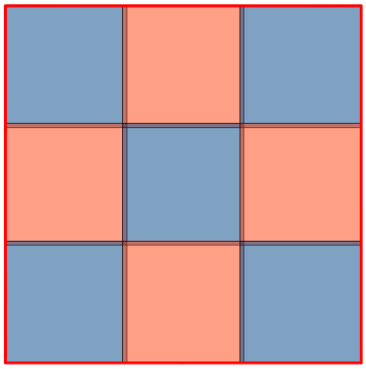

Figura 8.4: Mapeo solapado de la estructura de hilos en CUDA y la matriz de datos. La linea roja muestra el limite de la matriz de datos y puede verse como todos los elementos de la matriz están asignados a algún hilo. Utilizando esta opción los bloques de hilos deben solapar dos de sus elementos con el bloque adyacente, ya que dichos elementos han de ser accedidos desde ambos bloques.

8.2.2. Método solapado (SOL)

En este método, en vez de mapear la estructura de hilos de CUDA con los elementos computables, se mapea sobre la totalidad de la matriz de datos. La filosofía en este caso es que cada hilo trae el dato que le corresponde por su posición y, si forma parte de los elementos computables del bloque, hace el cálculo.

Si cada hilo solo se encarga de traer de memoria de dispositivo un solo dato, esto implica que los hilos que se encuentren a los bordes del bloque de hilos no podrán efectuar el cálculo del *stencil* y solo se utilizarán para traer el dato de memoria. Por este motivo, los bloques de hilos adyacentes están solapados unos con otros, porque algunos elementos tienen que leerse desde diferentes bloques (Figura 8.4 y Figura 8.5).

El *kernel*, en primer lugar, identifica qué elemento de la matriz se corresponde con el hilo y lo trae de memoria de dispositivo a memoria compartida. Entonces se espera a que el resto de hilos del bloque hayan traído el elemento y, si el elemento que corresponde al hilo es computable (es decir, que no está en el borde del bloque de hilos), se lleva a cabo el cálculo de ese *stencil*.

El código resultante es bastante más sencillo y limpio que en el caso del método directo, pues se evitan varias ramificaciones en el código, solo es necesaria una ramificación para decidir si hacer o no el cálculo (Algoritmo 8.3).

No se ha encontrado ninguna fuente que intente resolver el problema de *stencil* con este enfoque. Aunque pueda parecer una mala idea dejar hilos de CUDA sin llevar a cabo ninguna operación salvo traer un elemento de memoria de dispositivo, las características de CUDA pueden hacer que sea ventajoso

Algoritmo 8.3 Algoritmo que muestra el *kernel* que calcula un elemento de un *stencil* de 5 puntos con el método solapado, el código es el original antes de aplicar ninguna optimización.

```
#define BLOCKROWS 16
#define BLOCKCOLS 16

__global__ void compute_stencil(float *inMat, float *outMat,
                                const int matCols) {
  // Allocate shared memory
  __shared__ float sData[BLOCKROWS][BLOCKCOLS];

  // Matching matrix element using threablock relative values
  int matRow = blockIdx.y * (BLOCKROWS-2) + threadIdx.y;
  int matCol = blockIdx.x * (BLOCKCOLS-2) + threadIdx.x;
  int matIndex = matRow*matCols + matCol;

  // Populate sData element of this thread
  sData[sRow][sCol] = inMat[matIndex];

  // Synchronise threadblock
  __syncthreads();

  // Avoid computation for boundary threads
  if (threadIdx.x != 0 &&
      threadIdx.x != BLOCKCOLS-1 &&
      threadIdx.y != 0 &&
      threadIdx.y != BLOCKROWS-1) {
    outMat[matIndex] = ALPHA * (sData[sRow-1][sCol] +
                                sData[sRow][sCol-1] +
                                sData[sRow][sCol] +
                                sData[sRow][sCol+1] +
                                sData[sRow+1][sCol]);
  }
}
```

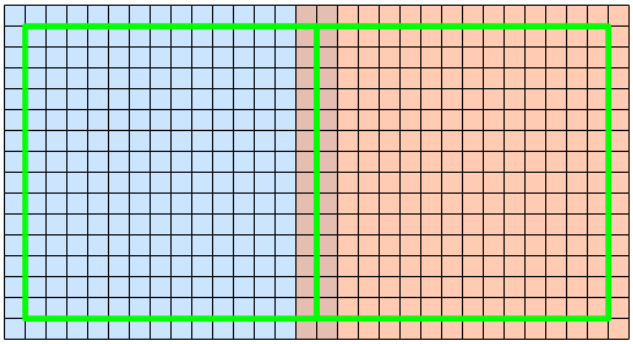

Figura 8.5: En la figura pueden verse dos bloques de hilos de 16 filas por 16 columnas. En el interior del área verde se muestran los elementos que se van a computar, de ahí que sea necesario solapar dos de las columnas de los bloques para que el algoritmo llegue a todos los elementos.

eliminar las ramificaciones y los saltos en el código que aparecen en el método directo.

8.3. Lineas a seguir en las optimizaciones

Una vez tenemos la implementación inicial de las dos versiones del algoritmo, es posible empezar a valorar las optimizaciones que pueden aplicarse al cálculo del *stencil* y a otros detalles del *kernel* para reducir el tiempo de ejecución.

CUDA no es un sistema tradicional, por lo que hay que pensar en sus peculiaridades para especular sobre que optimizaciones podrían mejorar el rendimiento. La propia guía para programadores da algunas claves a lo largo de su lectura que pueden ayudar a planear las optimizaciones que conviene aplicar al *kernel*.

8.3.1. Tamaño del bloque de hilos

Para poder explicar por qué escoger cierto tamaño para el bloque de hilos es importante, primero hay que explicar como CUDA ejecuta los hilos dentro de este.

Cuando es el turno de un bloque de hilos para utilizar los recursos de un multiprocesador, CUDA agrupa y ejecuta los hilos dentro de dicho bloque en conjuntos de 32 hilos consecutivos, estos conjuntos son llamados *warps*. Cabe esperar pues, que para obtener el máximo aprovechamiento de los recursos, el tamaño del bloque de hilos sea un múltiplo del tamaño del *warp*, ya que de lo contrario se crearían grupos de menor tamaño y parte de los recursos quedarían ociosos.

En la guía para programadores, también se hace referencia a utilizar bloques

de hilos de tamaño múltiplo de 64 para evitar conflictos en la asignación de registros a cada hilo, aunque esta recomendación no viene acompañada de ningún tipo de explicación o evidencia del porqué de esto.

8.3.2. Acceso a memoria consecutiva

Los accesos a memoria de dispositivo se hacen a nivel de *half-warp* (16 hilos) en vez de hacerse a nivel de *warp* (32 hilos). Ocurre en muchas ocasiones que elementos consecutivos en un *half-warp* acceden a elementos consecutivos en memoria de dispositivo. Cuando eso ocurre y todos los datos solicitados se encuentran en el mismo segmento de 128 bytes (alineado también a 128 bytes), solo se lleva a cabo un acceso a memoria de dispositivo y se traen a la vez todos los datos solicitados. En general, un *half-warp* que acceda a datos en memoria de dispositivo hará tantos accesos como diferentes segmentos de 128 bytes estén siendo accedidos.

Puede parecer pues una buena idea que el número de columnas en el bloque de hilos sea un múltiplo de 16 en el problema de *stencils*, ya que en ese caso se garantiza que, a lo sumo, solo se hagan dos accesos a memoria por cada solicitud.

8.3.3. Acceso a memoria compartida

De una forma parecida a lo que ocurre con los accesos a memoria de dispositivo, los accesos a memoria compartida también pueden optimizarse.

La memoria compartida esta formada por 16 bancos diferentes, cuando los hilos de un mismo *half-warp* acceden cada uno a un banco diferente, el acceso a memoria compartida se paraleliza totalmente y resulta óptimo. En caso de que varios hilos accedan al mismo banco, los accesos se serializan y se pierde rendimiento.

8.3.4. Eliminación de condicionales y saltos

Cualquier instrucción que en un *kernel* cree dos o más caminos de ejecución diferentes dentro de un mismo *warp*, hará en consecuencia que dichos caminos se serialicen, por lo que parte de los hilos tendrán que esperar a que el resto de hilos completen su camino para poder continuar.

A mayor número de caminos de ejecución, más posibilidades de que estos tengan que serializarse, por lo que conviene intentar evitar este tipo de instrucciones de control del flujo del programa (de ahí el interés en probar el método solapado).

8.3.5. Recursos limitados

Para gestionar las ejecuciones, CUDA asigna bloques de hilos a los diferentes multiprocesadores del dispositivo. Es conveniente que todos los multiprocesadores tengan por lo menos 2 bloques de hilos ejecutándose concurrentemente en ellos para poder optimizar el uso de recursos, ya que mientras un bloque de hilos está sincronizando o haciendo E/S, el otro puede estar llevando a cabo cálculos.

Sin embargo, los recursos de los multiprocesadores como la memoria compartida o los registros son limitados, y se comparten entre todos los bloques de hilos asignados al mismo multiprocesador. En caso de que un bloque de hilos supere el total de recursos del multiprocesador, se intentará detectar en tiempo de compilación y no permitir compilar dicho código, aunque se dan casos en los que el código compila correctamente y después CUDA lanza un error durante la ejecución por falta de recursos.

Es posible que un bloque de hilos acapare tantos recursos por si solo que no sea posible asignar 2 bloques al mismo multiprocesador. Está situación derivará seguramente en un empeoramiento del rendimiento.

El tamaño del bloque de hilos afecta directamente al uso de recursos de este, por lo que la mejor manera de encontrar el tamaño óptimo es probando con diferentes tamaños y escoger la opción que dé mejores resultados, que es lo que se hará durante los experimentos. Intentar analizar los recursos que se emplean a priori para seleccionar el mejor candidato es una tarea más lenta que probarlos todos, además de que no existe ninguna garantía de que pueda calcularse el tamaño de bloque óptimo sin experimentar.

8.4. Optimizaciones y resultados del *stencil* de 5 puntos

En esta sección se mostrarán las diferentes optimizaciones aplicadas a las diferentes implementaciones del *stencil* de 5 puntos. Para cada optimización, se ha probado un rango diverso de tamaño de bloque, siempre manteniendo el número de columnas en un múltiplo de 16 por los motivos explicados en la sección anterior. Esto se hace así porque es difícil prever qué tamaño de bloque será el óptimo para la ejecución, ya que depende de muchos factores internos del hardware y de la cantidad de recursos utilizados y disponibles.

Para las ejecuciones, se ha utilizado una matriz de 256MB y se han llevado a cabo un total de 1500 iteraciones en el tiempo.

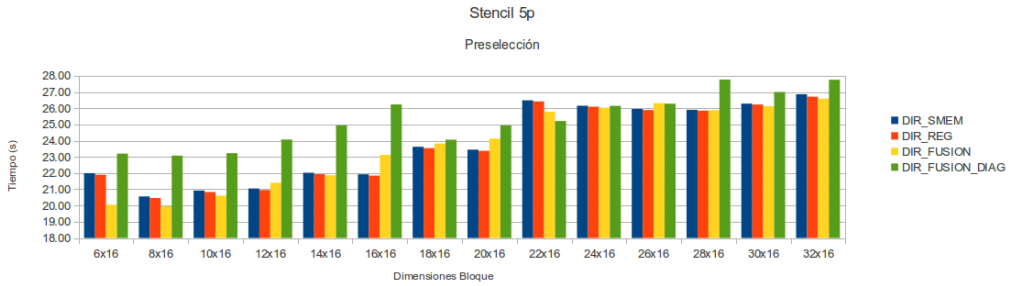

Figura 8.6: Gráfico de resultados del *stencil* de 5 puntos con método directo. Con los valores del eje vertical aumentados acotando el rango para poder apreciar cual es la mejor ejecución y seleccionarla, con el fin de poderla comparar con el resto de implementaciones.

8.4.1. Método directo

A partir de la versión inicial del algoritmo con memoria compartida (DIR_SMEM) se han podido aplicar algunas optimizaciones. La primera de ellas consiste únicamente en emplear un registro para almacenar el valor del elemento que se trae de memoria de dispositivo y utilizarlo después en el cálculo del *stencil* en vez de llevar a cabo un acceso a memoria compartida (DIR_REG). Esta optimización es muy simple, aunque la mejora en el tiempo de ejecución es siempre menor al 5 %, por lo que no es significativo.

El resto de optimizaciones que se pueden aplicar aquí, están relacionadas con las instrucciones de control de flujo para traer los elementos de los bordes. Lo que se ha hecho es reducir los 4 posibles caminos con una condición muy simple, en 2 con una condición algo más compleja (DIR_FUSION) y en 1 con otro tipo de condición (DIR_FUSION_DIAG).

En los resultados puede verse como las diferentes optimizaciones tienen efectos diferentes en el tiempo de ejecución según el tamaño de bloque de hilos que se emplee. El análisis de todos y cada uno de los resultados depende de factores que son difíciles de analizar, sin embargo puede observarse como en la mayoría de ocasiones el mejor resultado viene dado, aunque por una diferencia de tiempo pequeña, por la optimización DIR_FUSION. En concreto, la mejor ejecución aparece cuando se emplea un bloque de hilos de 8 filas por 16 columnas (128 hilos).

Nótese también que el aumentar el tamaño de bloque causa un aumento paulatino del tiempo de ejecución. Esto es así probablemente porque a mayor tamaño de bloque se agotan algunos de los recursos del dispositivo y éste no es capaz de mantener de forma concurrente tantos bloques de hilos a la vez (Figura 8.6).

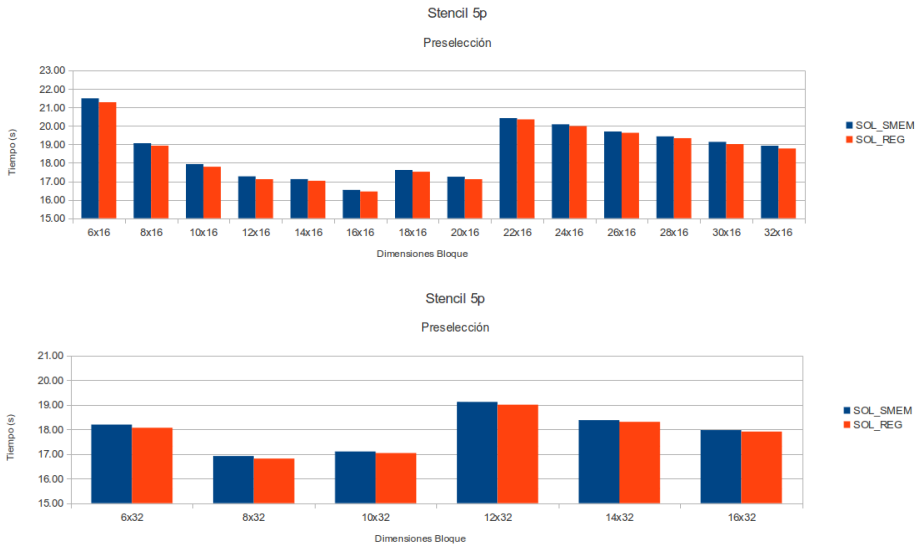

Figura 8.7: Gráficos de resultados del *stencil* de 5 puntos con método solapado y tamaño de columna 16 y 32. Con los valores del eje vertical aumentados acotando el rango para poder apreciar cual es la mejor ejecución y seleccionarla, con el fin de poderla comparar con el resto de implementaciones.

8.4.2. Método solapado

En este caso, la versión inicial del algoritmo (SOL_SMEM) no da lugar más que a una optimización, que es la misma que en la otra implementación de almacenar el propio elemento en un registro además de en memoria compartida (SOL_REG).

Al igual que en el caso anterior, la única optimización aplicada ofrece una mejoría que ni siquiera es significativa por la cantidad de tiempo que reduce (no llega al 5 %). Sin embargo, si puede verse como las dimensiones del bloque de hilos afectan de forma diferente en esta versión, y es que debido a que el código es más sencillo y utiliza menos recursos tanto de registros como de memoria compartida, es posible utilizar tamaños de bloque mayores y obtener mejor rendimiento.

En este caso, se logra el valor óptimo con bloques de 16 filas y 16 columnas (256 hilos). La misma cantidad de hilos pero con diferente distribución cuando se utilizan 8 filas y 32 columnas da resultados similares, pero en el primer caso son algo mejores (Figura 8.7).

8.4.3. Conclusiones

El *stencil* de 5 puntos, también en CUDA, deja muy poco espacio a optimizaciones, ya que este es muy simple y se hace difícil mejorarlo por medios que

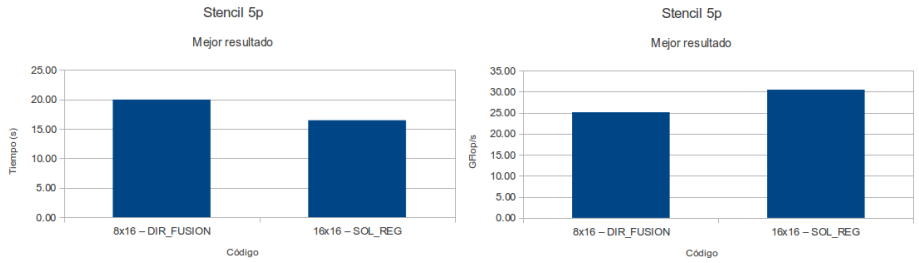

Figura 8.8: Gráficos que muestran los resultados finales con las mejores ejecuciones de las dos versiones del algoritmo del *stencil* de 5 puntos.

256MB - 1500 its	Tiempo (s)	GFlop/s	GStencil/s
8x16 – DIR_FUSION	19.97	25.11	5.02
16x16 – SOL_REG	16.45	30.47	6.09

Cuadro 8.1: Resultados en tiempo de ejecución, GFlop/s y GStencil/s de las mejores ejecuciones de las dos versiones del algoritmo del *stencil* de 5 puntos.

no sean optimizaciones en el acceso a los datos.

En la Figura 8.8 y el Cuadro 8.1 pueden verse comparados los resultados en tiempo de ejecución y rendimiento de las dos versiones del algoritmo. Se puede apreciar como el método solapado ofrece una mejora significativa del rendimiento respecto al método directo, alrededor del 21 %.

8.5. Optimizaciones y resultados del *stencil* de 27 puntos

En esta sección, al igual que se ha hecho en la anterior, se mostrarán los resultados de las diferentes optimizaciones aplicadas a las diferentes implementaciones del *stencil* de 27 puntos. El procedimiento es también el mismo, probar las diferentes optimizaciones junto con los diferentes tamaños de bloque con el objetivo de poder seleccionar la mejor ejecución y compararla con la otra versión del algoritmo. El análisis de los tiempos de ejecución en función de estas variables se hará en lineas generales, ya que son muchos y muy complejos los factores que alteran los tiempos de ejecución dependiendo de la cantidad de recursos que utiliza el *kernel*.

Para las ejecuciones, se ha utilizado una matriz de 256MB y se han llevado a cabo un total de 750 iteraciones en el tiempo.

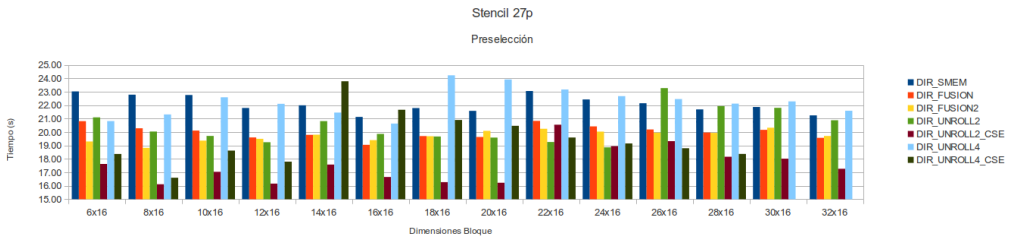

Figura 8.9: Gráfico de resultados del *stencil* de 27 puntos con método directo. Con los valores del eje vertical aumentados acotando el rango para poder apreciar cual es la mejor ejecución y seleccionarla, con el fin de poderla comparar con el resto de implementaciones.

8.5.1. Método directo

En este caso, y gracias al bucle que itera a través de los diferentes planos, es posible aplicar más optimizaciones a la versión inicial del algoritmo (DIR_SMEM) y hacer un desenrollado de hasta 4 elementos por iteración del bucle que recorre cada uno de los planos de la matriz tridimensional, con su correspondiente eliminación de subexpresiones comunes. Hay que tener en cuenta que el desenrollado de bucles aumenta considerablemente el uso de los recursos necesarios, tanto de memoria compartida como de registros utilizados, por lo que habrá que encontrar la combinación de nivel de desenrollado y tamaño del bloque de hilos que de menor tiempo de ejecución. Dicha tarea solo será posible mediante la ejecución de todas las combinaciones posibles y la posterior comprobación visual.

Al igual que en el *stencil* de 5 puntos, también se pueden volver a aplicar pequeños cambios sobre las instrucciones de control de flujo para ver qué versión da mejor resultados (DIR_FUSION y DIR_FUSION2), aquí sin embargo si que se produce una reducción significativa del tiempo de ejecución (Figura 8.9).

En los resultados se aprecia como la mejor optimización con casi todos los tamaños de bloque de hilos es el desenrollado de bucle de 2 elementos por cada iteración con eliminación de subexpresiones comunes. Aunque en varios tamaños de bloque distintos el tiempo es muy similar, el que da mejor resultado por un pequeño margen es el que está compuesto por 8 filas y 16 columnas de nuevo (128 hilos).

8.5.2. Método solapado

Debido de nuevo a la ausencia de elementos susceptibles de ser optimizados en el código original (SOL_SMEM), las únicas optimizaciones posibles son las que permiten un desenrollado de hasta 8 elementos por iteración del bucle

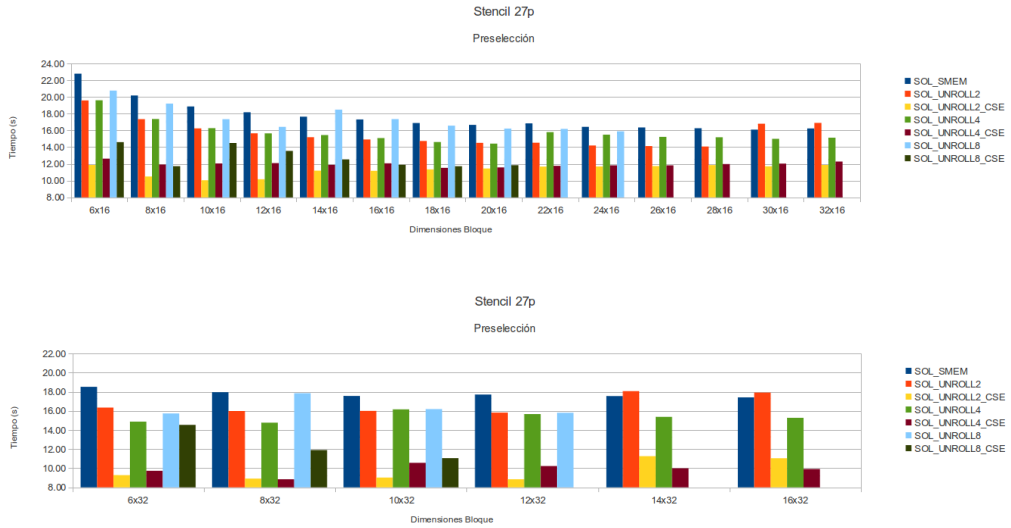

Figura 8.10: Gráficos de resultados del *stencil* de 27 puntos con método solapado y tamaño de columna 16 y 32. Con los valores del eje vertical aumentados acotando el rango para poder apreciar cual es la mejor ejecución y seleccionarla, con el fin de poderla comparar con el resto de implementaciones.

que recorre los planos de la matriz tridimensional, así como la eliminación de subexpresiones comunes.

Los resultados muestran que el menor tiempo de ejecución cuando los bloques son de 16 columnas es el desenrollado de nivel 2, pero con bloques de 32 columnas los resultados no son tan concluyentes. Hay dos dimensiones de bloques de hilos diferentes (8 filas por 32 columnas y 12 filas por 32 columnas) que dan resultados idénticos, la primera con el desenrollado de nivel 4 y la segunda con el desenrollado de nivel 2 (Figura 8.10).

8.5.3. Conclusiones

Aunque tampoco son muchas las optimizaciones que pueden aplicarse en el *stencil* de 27 puntos, el poder aplicar desenrollado de bucles y eliminación de subexpresiones comunes ha dado lugar a mejoras importantes en el rendimiento respecto al código original, reduciendo hasta la mitad el tiempo de ejecución en algunos casos tras ser aplicada la optimización.

En la Figura 8.11 y el Cuadro 8.2 pueden verse comparados los resultados en tiempo de ejecución y rendimiento de las dos versiones del algoritmo, aunque en este caso se muestran dos códigos diferentes del método solapado, ya que han dado resultados idénticos en las anteriores pruebas. En este caso, el rendimiento que ofrece la versión con método solapado, que es la versión ideada durante

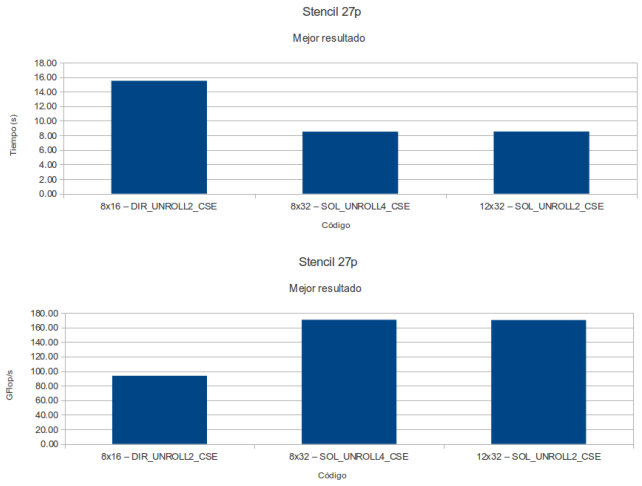

Figura 8.11: Gráficos que muestran los resultados finales con las mejores ejecuciones de las dos versiones del algoritmo del *stencil* de 27 puntos. De la versión solapada se dan dos resultados diferentes porque los dos han dado tiempos de ejecución prácticamente idénticos.

256MB - 750 its	Tiempo (s)	GFlop/s	GStencil/s
8x16 – DIR_UNROLL2_CSE	15.50	93.65	3.12
8x32 – SOL_UNROLL4_CSE	8.51	170.48	5.68
12x32 – SOL_UNROLL2_CSE	8.54	170.04	5.67

Cuadro 8.2: Resultados en tiempo de ejecución, GFlop/s y GStencil/s de las mejores ejecuciones de las dos versiones del algoritmo del *stencil* de 27 puntos.

la realización de este trabajo, es muy superior al de la versión con método directo, alrededor de un 80 %. Dicha mejora está posiblemente influenciada por la eliminación del control de flujo en la mayor parte del algoritmo.

Capítulo 9

Conclusiones

En este capítulo se mostraran las valoraciones y conclusiones sobre las diferentes partes del trabajo. Todo ello se hará mostrando la evolución en el rendimiento de las dos implementaciones tras los diferentes cambios.

9.1. Las optimizaciones en el código secuencial

Tras implementar las versiones secuenciales sin optimizar del algoritmo, la primera conclusión a la que se puede llegar, mucho antes de empezar siquiera a pensar en paralelización del código, es la baja mejora en el rendimiento que las optimizaciones causan en el *stencil* de 5 puntos frente a lo que ocurre con el de 27 puntos, donde las optimizaciones en el cálculo tendrán un efecto notable sobre el rendimiento de este *stencil* (Figura 9.1).

Ya desde este momento se puede observar un comportamiento que seguirá apareciendo durante todo el proceso de optimización de los algoritmos, y es que la baja intensidad computacional del *stencil* de 5 puntos dificulta que las optimizaciones en el cálculo tengan efectos significativos, ni siquiera empleando vectorización. Mientras tanto, todo lo contrario ocurre con el *stencil* de 27 puntos, cuyo rendimiento se ve mejorado con mayor intensidad cuando se aplican optimizaciones.

Otra conclusión que se puede sacar de esta parte del trabajo todavía secuencial, es que es imprescindible asumir cuantas menos cosas mejor y experimentar con todo lo que este al alcance. La prueba de ello es como una arbitraria decisión de utilizar una determinada composición para los datos puede provocar que el rendimiento se vea afectado. Es difícil pensar que el valor de unos operandos, que llegan a un procesador para llevar a cabo una instrucción aritmética básica, pueda influir en el número de ciclos que emplea la instrucción. Sin embargo esto es así cuando se trata de números *denormales*, valores muy cercanos al cero que

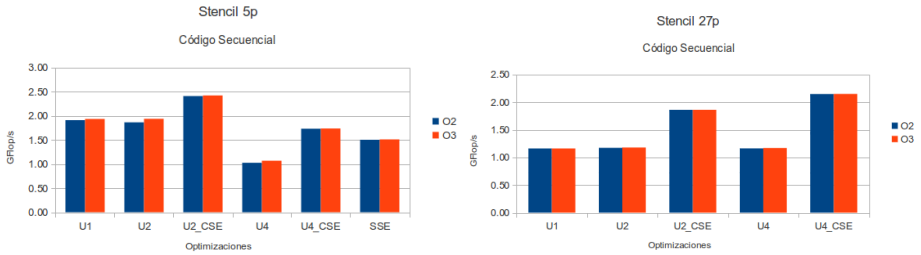

Figura 9.1: Gráficas que muestran el rendimiento en GFlop/s de los diferentes códigos correspondientes a cada una de las optimizaciones aplicadas al cálculo de los *stencils* de 5 y 27 puntos. El primer resultado (U1) es el código original sin optimizar. La letra *U* (*unrolling*) seguida de un número indica el grado de desenrollado del bucle, mientras que CSE (*Common Subexpression Elimination*) se refiere a los código que aplican eliminación de subexpresiones comunes. Por último está el código SSE, que es el que aplica vectorización (es implícito que el desenrollado en este caso es de cuatro elementos).

no son bien tolerados cuando llega la hora de hacer un cálculo.

Esta situación se da cuando se emplea una matriz llena de ceros y con bordes con valores fijos, que tras cada iteración se propagan hacia el interior con un valor cada vez menor, que eventualmente llega al limite y se vuelve un valor *denormal*. Es entonces cuando el rendimiento del *stencil* empieza a decrecer de forma notable. Este fenómeno está explicado con detalle en el Apéndice A.

9.2. Paralelización en entornos *multi-core*

Llegado el punto en el que se inician los preparativos para paralelizar el algoritmo, surge la duda de que modelo de programación paralela ofrecerá mejor rendimiento en el sistema con el que se trabaja. Uno puede inicialmente pensar que los modelos basados en memoria compartida tendrán ventaja frente a los de memoria distribuida, por el hecho de que ahorrarán la sobrecarga innecesaria que provoca un entorno de ejecución distribuido trabajando de forma local. Tan pronto como empiezan a observarse los resultados, uno llega a la conclusión de que la incidencia en el rendimiento de este algoritmo cuando use un modelo basado en memoria compartida frente a otro basado en memoria distribuida no es significativa. Esto es debido principalmente al alto protagonismo que cobran los *stencils*, ya sea por la necesidad de ancho de banda del *stencil* de 5 puntos, o por la necesidad de más tiempo de proceso del *stencil* de 27 puntos.

Este descubrimiento lleva a una conclusión posterior, y es que el tiempo que se requiere para sincronizar los datos al final de cada iteración con es insignificante y no merece ser optimizado, ya que aun empleando complicados mecanismos de comunicación asíncrona o enviando datos de forma temprana

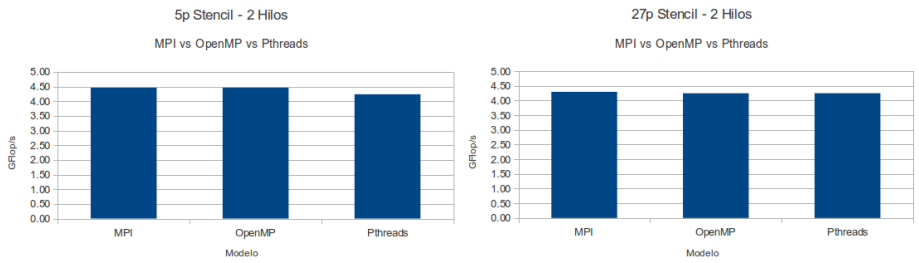

Figura 9.2: Gráficas que muestran el rendimiento en GFlop/s de los diferentes modelos de programación paralela utilizando 2 hilos a la vez.

para solapar cálculo y comunicación, la ley de *Amdahl* es clara en este caso y no se obtendrá ninguna mejora en el rendimiento.

La consecuencia de la situación anterior, siendo la comunicación el único punto del algoritmo donde el modelo de programación paralela puede influir notablemente, es que independientemente de con que librería se paralelice el trabajo, el rendimiento será similar, como puede apreciarse en el ejemplo de la Figura 9.2.

Otro factor a considerar es el tipo de mapeo de los hilos a los procesadores, pues el sistema dispone de dos procesadores independientes y cada uno puede hacer uso de su propio bus de datos para alimentar a los hilos que estén mapeados en el. Una vez más, se hace notable la diferencia entre el *stencil* de 5 puntos y el de 27 puntos. Mientras en el *stencil* de 5 puntos se hace imposible alimentar a 4 hilos a la vez en el mismo procesador sin una perdida del rendimiento considerable, el efecto que tiene en el *stencil* de 27 puntos es mucho más reducido y el rendimiento es similar tanto si se reparten los hilos como si se mapean al mismo, ya que este *stencil* está limitado por el cálculo y no por el ancho de banda (Figura 9.3).

Al estar utilizando una máquina con acceso a memoria no uniforme (NUMA), se puede perder rendimiento cuando un hilo que accede a memoria lo hace a un

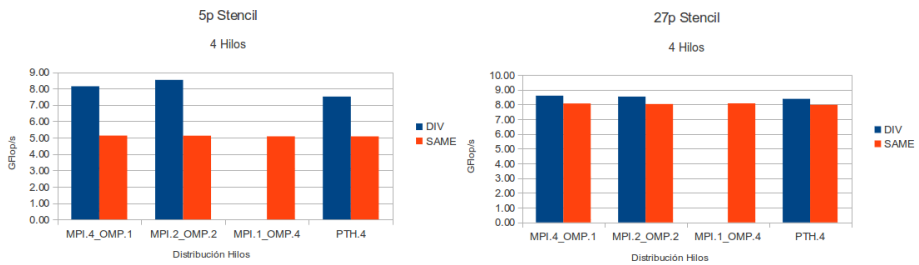

Figura 9.3: Gráficas que muestran el rendimiento en GFlop/s de las diferentes distribuciones de hilos con los dos diferentes tipos de mapeo para 4 hilos.

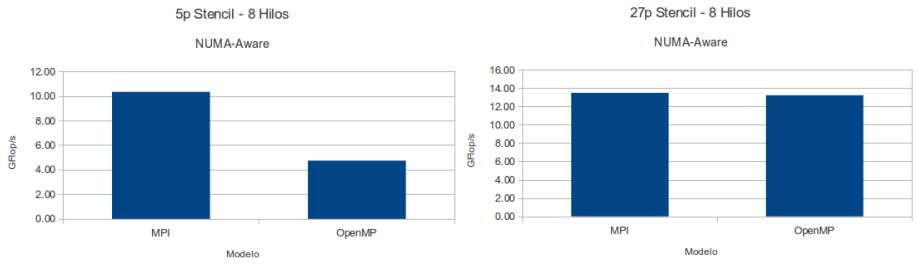

Figura 9.4: Gráficas que muestran el rendimiento en GFlop/s al utilizar una ejecución que tiene en cuenta el acceso a memoria no uniforme (MPI) y una que no lo tiene (OpenMP), con un total de 8 hilos y sin ningún tipo de mapeo.

módulo que no sea local. Se puede forzar que esto ocurra ejecutando el código para los 8 hilos y OpenMP sin ningún tipo de mapeo, por lo que aunque los 8 hilos estarán repartidos entre los dos procesadores, los datos estarán en el banco local de solo uno de ellos, por lo que 4 hilos estarán haciendo accesos a un banco no local. Puede apreciarse en la Figura 9.4 como esto solo afecta al *stencil* de 5 puntos, donde la perdida de rendimiento es superior al 50 %, mientras que el de 8 puntos no se ve prácticamente alterado. De nuevo, esto es atribuible al hambre de datos que pasa el *stencil* de 5 puntos, que se ve aun más agraviada por el acceso a datos en bancos no locales.

Finalmente, puede concluirse que el *stencil* de 5 puntos no es capaz de escalar tan bien como el de 27 puntos, ya que el segundo no está hambriento por falta de datos y puede continuar escalando de forma prácticamente lineal. Se aprecia en los resultados de la Figura 9.5 y el Cuadro 9.1 como el *stencil* de 5 puntos no pasa de un *Speed Up* de 4.29x mientras que el de 27 puntos llega a un *Speed Up* algo más cerca de ser lineal, de 7.53x. También resulta interesante observar los *Speed Up* lineales con el uso de 2 hilos en ambos *stencils* y el de 4 hilos en el *stencil* de 27 puntos, lo cual indica que existe una mejora producida por la mayor localidad de los datos, ya que aun siendo una fracción muy poco significativa del tiempo de ejecución, la comunicación entre hilos al final de cada iteración debería haber evitado un *Speed Up* lineal.

9.3. Paralelización en entornos *many-core*

En el entorno *many-core* de este trabajo, podría concluirse que existen tres componentes que determinan el rendimiento del resultado: el algoritmo, el tamaño de bloque y las optimizaciones.

Escoger el algoritmo que utiliza el método directo o solapado es posiblemente la decisión más influyente en el rendimiento que se puede tomar en el entorno

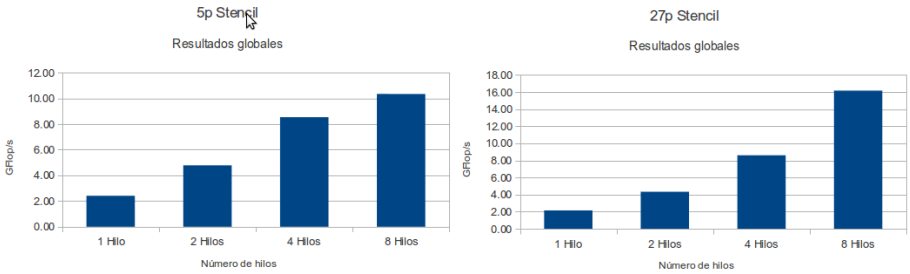

Figura 9.5: Gráficas que muestran el rendimiento en GFlop/s para diferentes números de hilos en el entorno *multi-core*.

5p Stencil	GFlop/s	GStencil/s	Speed Up
1 Hilo	2.41	0.48	1x
2 Hilos	4.78	0.96	1.98x
4 Hilos	8.53	1.71	3.54x
8 Hilos	10.33	2.07	4.29x

27p Stencil	GFlop/s	GStencil/s	Speed Up
1 Hilo	2.15	0.07	1x
2 Hilos	4.33	0.14	2.02x
4 Hilos	8.60	0.29	4.00x
8 Hilos	16.16	0.54	7.53x

Cuadro 9.1: Cuadro que muestra el rendimiento en GStencil/s, GFlop/s de los resultados de la ejecución para diferentes números de hilos en el entorno *multi-core*, junto con el *Speed Up* logrado. Destacar que la métrica de GStencil/s solo es valida para comparar el mismo *stencil*, por lo que no es correcto comparar dicho valor entre los *stencils* de 5 y 27 puntos.

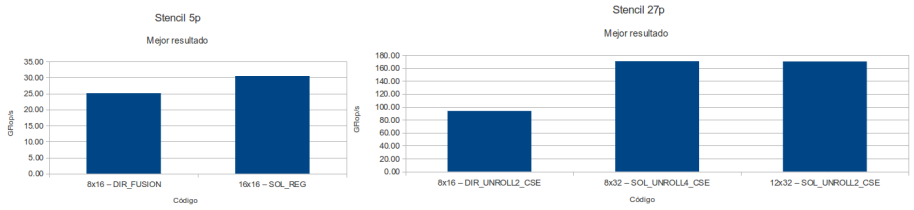

Figura 9.6: Gráficas que muestran el rendimiento en GFlop/s para los dos algoritmos en el entorno *many-core*.

many-core. Entre los dos algoritmos implementados, esta el directo, que es el método que se podría plantear de forma natural y que de hecho así aparece en los ejemplos de la web de CUDA, y el solapado, que es una implementación que se ha desarrollado durante la realización de este trabajo al observar que este otro sistema podría mejorar el rendimiento.

Se ha visto como para ambos *stencils* el uso del método solapado se corresponde con una mejora clara en el rendimiento. Si bien la diferencia de rendimientos no es tan notable en el *stencil* de 5 puntos, ésta si que es significativa, ya que está en torno al 20 %.

Por otra parte, la mejora en el rendimiento del *stencil* de 27 puntos es muy superior, estando alrededor del 80 %. Resulta pues evidente que el uso del método solapado es la mejor opción para este tipo de *stencils* de primer orden (Figura 9.6).

Sobre el efecto del tamaño del bloque de hilos, si bien no es tan influyente como la elección del algoritmo, afecta también al rendimiento, ya que para bloques de hilos muy pequeños, es posible que no se estén utilizando los recursos de forma óptima, mientras que si se usan bloques de hilos demasiado grandes, cabe la posibilidad de que no puedan asignarse suficientes bloques a cada multiprocesador del dispositivo CUDA para que haya una ejecución concurrente de ellos.

En general, para encontrar el tamaño de bloque óptimo, suele ser necesario realizar pruebas sobre el dispositivo hasta dar con el tamaño adecuado. Aunque en caso de existir diferentes versiones optimizadas de un mismo código, estas deberían combinarse con los diferentes tamaños de bloque con el fin de encontrar el código y el tamaño de bloque óptimo.

Esto es así porque algunas de las optimizaciones, como el desenrollado de bucles, requiere de un gran consumo adicional de recursos respecto al código no optimizado, por lo que hay que probar de ejecutar todas las combinaciones posibles para encontrar la mejor ejecución. En la Figura 9.6 está el ejemplo que ilustra este comportamiento en el *stencil* de 27 puntos, donde un desenrollado de nivel 4 con un tamaño de bloque de 256 tiene un rendimiento prácticamente

idéntico a un desenrollado de nivel 2 con un tamaño de bloque de 384. Esto ocurre porque los recursos necesarios por bloque (memoria compartida y registros) del desenrollado de nivel 2 son menores a los del desenrollado de nivel 4, por lo que diferentes tamaños de bloque benefician a uno o al otro.

De nuevo, destacar que es difícil llevar a cabo este tipo de deducciones a priori, por lo que es necesario experimentar con las posibles combinaciones de tamaños de bloque y optimizaciones en el código para poder encontrar el óptimo.

9.4. Mejora en el rendimiento global

En esta sección se pretende simplemente ilustrar la comparación de los resultados obtenidos para entornos *multi-core* y *many-core*, con el fin de poder dar una imagen de la diferencia de rendimiento entre sistemas.

En el Cuadro 9.2 se aprecia el rendimiento obtenido para los dos *stencils* en ambos entornos. Si bien el entorno *many-core* ha dado, como se esperaba, mejor rendimiento para el mismo trabajo, la ganancia total en el caso del *stencil* de 27 puntos es de dos orden de magnitud en cuanto a GFlop/s, lo que es un resultado realmente interesante.

Con estos resultados se puede concluir definitivamente que la alta intensidad computacional del *stencil* de 27 puntos hace que el utilizar un entorno destinado a llevar a cabo problemas que requieran de una gran potencia de cálculo, como es el caso de CUDA, sea muy beneficioso.

En el caso del stencil de 5 puntos, aunque no ha salido tan beneficiado como el de 27 puntos, si que se ha conseguido un *Speed Up* importante con el uso de CUDA, ya que posiblemente se ha beneficiado no solo de la mayor potencia de cálculo, sino también de los métodos de acceso a memoria de dispositivo y el uso de memoria compartida.

5p-Stencil	GFlop/s	GStencil/s	Speed Up
1 Hilo - CPU	1.91	0.38	1x
8 Hilos – CPU	10.33	2.07	5.41x
GPU	30.47	6.09	15.95x

27p-Stencil	GFlop/s	GStencil/s	Speed Up
1 Hilo – CPU	1.16	0.04	1x
8 Hilos – CPU	16.16	0.54	13.9x
GPU	170.48	5.68	146.97x

Cuadro 9.2: Resultados de rendimiento en GFlop/s y GStencil/s de las implementaciones para *stencils* para entornos *multi-core* y *many-core*. Comparando el resultado secuencial sin optimizar, con los mejores resultados optimizados para ambos entornos, con *Speed Up*. Destacar que la métrica de GStencil/s solo es valida para comparar el mismo *stencil*, no es correcto comparar dicho rendimiento entre los *stencils* de 5 y 27 puntos.

Capítulo 10

Trabajo futuro

En el desarrollo de este trabajo se han explorado muchos temas diferentes pero relacionados entre si por el tipo de problema que se estaba tratando. Por una parte, se han probado dos *stencils* diferentes que comparten ciertas características y difieren en otras. También se han probado diferentes librerías para paralelizar el código y se ha utilizado tanto un sistema multiprocesador con acceso a memoria no uniforme como un dispositivo GPU. A esto hay que añadir las diferentes optimizaciones y enfoques que se han utilizado durante la implementación y las métricas utilizadas para medir el rendimiento.

A pesar de ello, se ha dejado mucho campo sin explorar, ya que se salía fuera del alcance de los objetivos y los recursos disponibles para este proyecto. Es por ello, que se ha redactado una lista con algunos de los nuevos campos que se podrían explorar en futuros trabajos relacionados con este.

- Por una parte, existe una gran variedad de plataformas y arquitecturas que pueden probarse, como por ejemplo CELL BE, procesador IBM PPC o de marca AMD en lugar de Intel.

- También están los tipos de *stencils* utilizados. Si bien es cierto que aquí se han implementado dos *stencils* diferentes, no se han explorado demasiadas opciones diferentes. Hay muchas características de los *stencils* que no se han probado y que abrirían caminos a nuevas optimizaciones y implementaciones, como por ejemplo *stencils* de alto orden, iteraciones tipo Gauss-Seidel o GSRB o coeficientes variables.

- En cuanto a optimizaciones aplicadas, aunque en el trabajo se han aplicado unas cuantas, hay optimizaciones que se ha decidido aparcar por falta de tiempo o por ser de una complejidad demasiado elevada. Entre ellas puede incluirse el bloqueo de iteraciones en el tiempo mediante el uso de estructuras especiales como colas circulares o la relajación de la

sincronización entre iteraciones, aunque esta última debe hacerse de forma que se garantice que el error que puede surgir en el resultado por el indeterminismo que introduce sea asumible.

- Otro campo que podría explorarse es el de repartir trabajo en una implementación híbrida que utilice tanto los recursos de la CPU como los de la GPU. Para ello bien podrían utilizarse las implementaciones hechas en este trabajo o utilizar un *framework* diferente como OpenCL que permita hacer uso de ambas.

- Utilizar números en coma flotante de simple precisión puede hacer que con el tiempo el error en los cálculos se haga insostenible. Es por eso que una de las opciones que podrían probarse a partir de este mismo trabajo es el uso de coma flotante de doble precisión, tanto para comparar como afecta a la precisión del resultado como para ver que diferencia en términos de rendimiento hay.

- Por último, sería interesante poder emplear otro tipo de métricas para calcular el rendimiento, como por ejemplo métricas basadas en la eficiencia energética del algoritmo sobre un hardware determinado.

Capítulo 11

Planificación del trabajo y presupuesto

En este capítulo se presenta la planificación del trabajo que se ha seguido y el análisis de los costes del trabajo.

11.1. Planificación

Para la planificación se muestra un diagrama de Gantt con la distribución de tareas que se han ido haciendo. La granularidad de las tareas no es demasiado fina, pero eso es debido a que se ha hecho especialmente difícil rastrear las tareas y el tiempo dedicado a cada una de ellas, por lo que se darán las tareas generales de las que se tiene constancia que se estaba trabajando durante la fecha (Figura 11.1).

11.2. Análisis de los costes

11.2.1. Costes de personal

Para considerar los costes, tendremos en cuenta los siguientes roles:

- Jefe de proyecto: Encargado de la planificación y el cumplimiento de los plazos. Se encargará de mantener reuniones bimensuales de 4 horas con los miembros del equipo que se encuentren trabajando de forma activa. También se encargará de preparar la presentación oral.

- Analista: Es la persona encargada del estudio del problema. Será el encargado de llevar a cabo el estudio del estado del arte y de proponer una

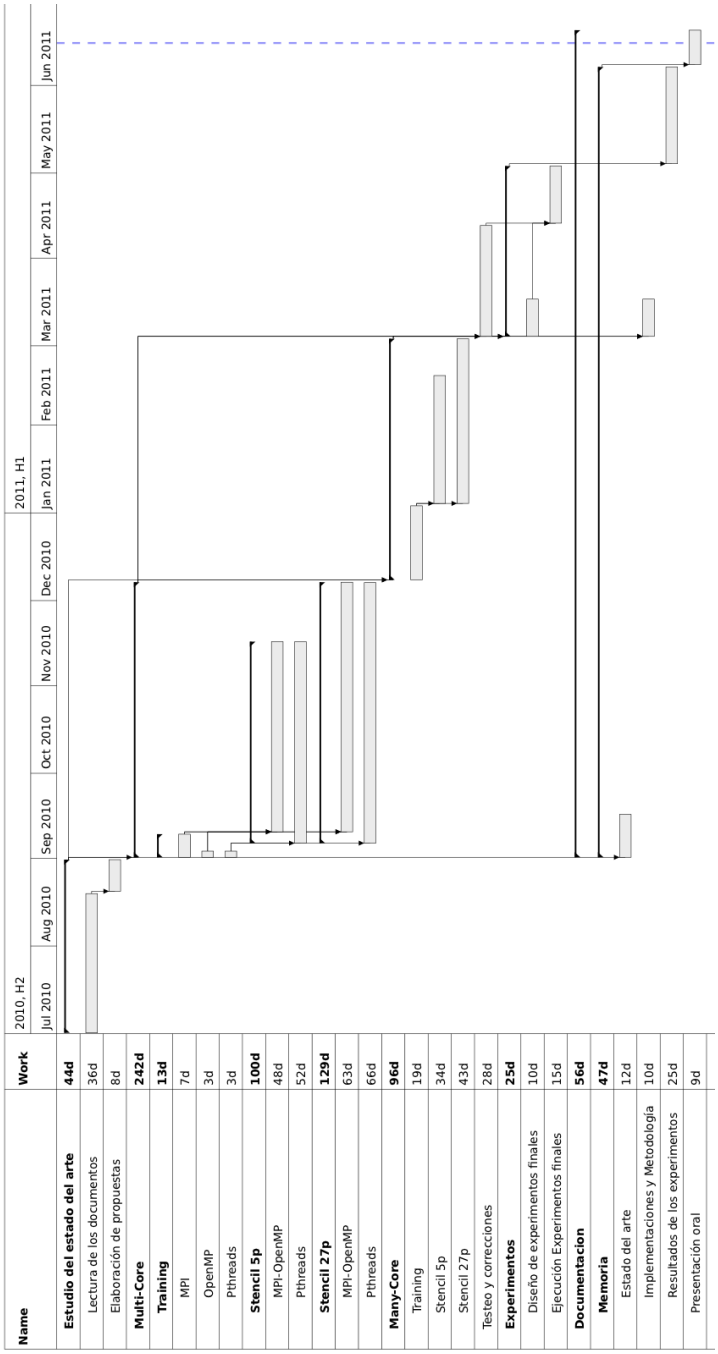

Figura 11.1: Diagrama de Gantt del proyecto.

solución con los modelos de programación paralela que se utilizarán. También se encargará de la parte proporcional de la documentación.

- Desarrollador: Será el encargado de toda la parte de implementación, optimización y corrección de errores, al igual que de parte de la documentación.
- Tester: Se encargará de diseñar y llevar a cabo los experimentos, así como de completar el informe de la documentación que muestre los resultados de estos.

Se asume que cada uno de los miembros del equipo hace 4 horas por día trabajado, excepto el. En el caso del jefe de proyecto, hay que sumar al total de horas las reuniones bimensuales del jefe de proyecto durante los 12 meses que ha durado.

En el Cuadro 11.1 puede observarse el cálculo total de los precios de personal a partir de los datos anteriores y de su correspondencia con las días trabajados según el Gantt de la Figura 11.1.

Personal	Días	Horas	€/Hora	Total
Jefe de proyecto	33	132	60	7920€
Analista	56	224	40	8960€
Desarrollador	142	568	35	19880€
Tester	50	200	25	5000€
Total	281	1124		41760€

Cuadro 11.1: Cuadro con el total del análisis de costes de personal. Al jefe de proyecto se le han sumado también los días de las reuniones.

11.2.2. Costes adicionales

En esta sección describimos los costes adicionales que surjan. Entre ellos tenemos las máquinas que hemos utilizado como hardware en el Cuadro 11.2.

Hardware	Precio
Máquina Servidor	2500€
Máquina cliente + Monitor	700€
Total	3200€

Cuadro 11.2: Cuadro con los costes de hardware.

También tendremos en cuenta los costes de alquiler por el uso del *cluster* de la universidad para las pruebas de MPI en un entorno real (Cuadro 11.3).

Por último, el coste del software, que en este caso se queda en 0€, ya que tanto el SO empleado, como el compilador o las librerías para paralelizar el problema son herramientas gratuitas. Con esto ya podemos calcular los costes adicionales totales en el Cuadro 11.4.

Servicios	Precio/Hora	Horas	Precio
Cluster	15€	20	300€

Cuadro 11.3: Cuadro con el coste por el uso del servidor.

Adicionales	Precio
Hardware	3200€
Servicios	300€
Software	0€
Total	3500€

Cuadro 11.4: Cuadro con el total de los costes adicionales.

11.2.3. Costes totales

Una vez disponemos de los costes salariales y los costes adicionales, se puede proceder a calcular el coste total del proyecto como se muestra en el Cuadro 11.5.

Proyecto	Precio
Personal	41760€
Adicionales	3500€
Total	45260€

Cuadro 11.5: Cuadro con el total de los costes.

Apéndice A

Composición de la matriz

Se ha dicho cuando se analizaban los resultados que la composición de la matriz no debería afectar al tiempo de ejecución del algoritmo. Resulta evidente pensar eso cuando uno asume que el algoritmo no tiene información inicial de los valores que pueden tener los elementos de la matriz y que, sea el valor que sea, lleva a cabo las mismas operaciones.

Con ese dato en mente, se diseñó de una forma bastante simple el tipo de matriz que se usaría para los experimentos, que contenía un valor fijo en los bordes y 0 en todos los valores del interior (Figura A.1). Con esto, podría comprobarse visualmente si las primeras iteraciones del algoritmo dan resultados correctos mientras se van implementando las optimizaciones, con el fin de comprobar que los cambios realizados no alteran el resultado del algoritmo.

Este diseño inicial, que puede parecer correcto, mostraba comportamientos no esperados en la escalabilidad al aumentar el número de iteraciones que se llevaban a cabo. Hacía que el aumentar el número de iteraciones no aumentase el tiempo de ejecución de forma proporcional, sino mayor. Se decidió investigar dicho comportamiento, pues tiraba por tierra muchas hipótesis iniciales y

1000	1000	1000	1000	1000	1000
1000	0	0	0	0	1000
1000	0	0	0	0	1000
1000	0	0	0	0	1000
1000	0	0	0	0	1000
1000	1000	1000	1000	1000	1000

Figura A.1: Composición inicial defectuosa de la matriz utilizada para los experimentos.

1000	1000	1000	1000	1000	1000
1000	1	1	1	1	1000
1000	1	1	1	1	1000
1000	1	1	1	1	1000
1000	1	1	1	1	1000
1000	1000	1000	1000	1000	1000

Figura A.2: Composición final de la matriz utilizada para los experimentos.

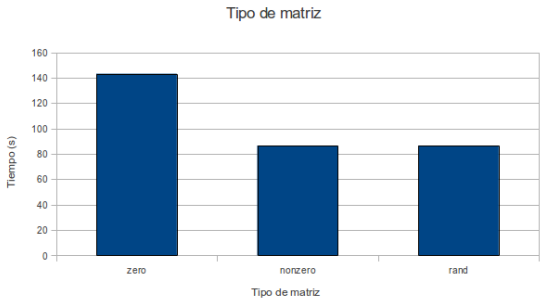

Figura A.3: Resultados del experimento para comprobar si la composición de la matriz afectaba o no al tiempo de ejecución. Puede verse un empeoramiento muy significativo de este cuando se utiliza la matriz original.

obstaculizaba el análisis de los resultados.

De forma casi casual, se descubrió que cuando se dividía la matriz en varios subproblemas, algunos de ellos diferían en cuanto al tiempo de ejecución, y curiosamente solo coincidían los que tenían una composición similar (al dividir la matriz en más de tres trozos, por alguna dimensión, alguno de los subproblemas tendrá una composición algo diferente al resto). Investigando este asunto, se decidió llevar a cabo pruebas con diferentes composiciones de matriz. Además de utilizar la original, se empleo una cuyos valores interiores estaban inicializados a 1 (Figura A.2) y otra cuyos valores eran completamente aleatorios.

El resultado de ese experimento es claro (Figura A.3), tan solo la matriz cuyos elementos internos están inicializados a 0 tenía un tiempo de ejecución superior realizando el mismo trabajo. Además de eso, las pruebas de escalabilidad daban resultados proporcionales cuando se empleaban los nuevos tipos de matrices, por lo que el problema residía exclusivamente en la composición original de la matriz.

La explicación de esto reside en que, con el paso de las iteraciones y teniendo en cuenta como se propaga el *stencil*, los nuevos valores que se obtenían al efectuar un cálculo y con los que se operaba posteriormente eran muy cercanos a 0

pero sin llegar a ser 0. Este tipo de valores están definidos como números denormales, y tienen su límite en la constante FLT_MIN, donde cualquier número más bajo formará parte de este grupo.

Resulta que en determinados procesadores Intel, el cálculo de operaciones en coma flotante con números denormales es más lento que cuando se opera con otros números [24], por lo que de ahí el aumento del tiempo de ejecución.

Existen varias formas de solucionar este problema, la que parece más procedente es modificar el comportamiento del hardware para que, en caso de producir un valor denormal tras una operación, lo convierta directamente a 0 (*flush-to-zero*). Esto puede hacerse llamando al principio del código a la función *_MM_SET_FLUSH_ZERO_MODE (_MM_FLUSH_ZERO_ON)*, definida en *xmmintrin.h*,que se encarga de modificar el comportamiento del hardware durante el resto de la ejecución del programa. Sin embargo, hacer esto implica una perdida de precisión en el cálculo, pues se está tratando como 0 a valores que no lo son.

Finalmente, y aunque en el código se ha añadido la llamada a función para solventar este problema, se usan matrices con valores interiores inicializados a algún valor diferente de 0, por si en alguna máquina no estuviese disponible la función para modificar el comportamiento.

Bibliografía

[1] Krste Asanovic, Rastislav Bodík, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David A. Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine A. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[2] Hikmet Dursun, Ken ichi Nomura, Weiqiang Wang, Manaschai Kunaseth, Liu Peng, Richard Seymour, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. In-core optimization of high-order stencil computations. In Hamid R. Arabnia, editor, *PDPTA*, pages 533–538. CSREA Press, 2009.

[3] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[4] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:261–270, 2009.

[5] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. EUROGRAPHICS 2005 STAR – state of the art report A survey of general-purpose computation on graphics hardware, February 07 2008.

[6] Michael Bader, Hans-Joachim Bungartz, Dheevatsa Mudigere, Srihari Narasimhan, and Babu Narayanan. Fast gpgpu data rearrangement kernels using cuda. Technical Report arXiv:1011.3583, Nov 2010.

[7] Cuda programming guide 2.3. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`, 2009.

[8] Liu Peng, Richard Seymour, Ken ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, Alexander Loddoch, Michael Netzband, William R.

Volz, and Chap C. Wong. High-order stencil computations on multicore clusters. In *Proc. 23rd IEEE International Symposium on Parallel and Distributed Processing (23rd IPDPS'09)*, pages 1–11, Rome, Italy, May 2009. IEEE Computer Society.

[9] Werner Augustin, Vincent Heuveline, and Jan-Philipp Weiss. Optimized stencil computation using in-place calculation on modern multicore systems. In Henk J. Sips, Dick H. J. Epema, and Hai-Xiang Lin, editors, *Euro-Par*, volume 5704 of *Lecture Notes in Computer Science*, pages 772–784. Springer, 2009.

[10] Sundaresan Venkatasubramanian and Richard W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd International Conference on Supercomputing (23rd ICS'09)*, pages 244–255, Yorktown Heights, NY, USA, June 2009. ACM Press. GA Tech.

[11] Stencilprobe: A microbenchmark for stencil applications. `http://www.cs.berkeley.edu/~skamil/projects/stencilprobe/`.

[12] Markus Wittmann, Georg Hager, and Gerhard Wellein. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. *CoRR*, abs/0912.4506, 2009. informal publication.

[13] Matthias Christen, Olaf Schenk, Peter Messmer, Esra Neufeld, and Helmar Burkhart. Accelerating stencil-based computations by increased temporal locality on modern multi- and many-core architectures. In *First International Workshop on New Frontiers in High-performance and Hardware-aware Computing (HipHaC'08) held in conjunction with the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41), November 8, 2008 Lake Como, Italy*, 2009.

[14] Matthias Christen, Olaf Schenk, Esra Neufeld, Peter Messmer, and Helmar Burkhart. Parallel data-locality aware stencil computations on modern micro-architectures. In *IPDPS*, pages 1–10. IEEE, 2009.

[15] A NUMA API for LINUX. `http://www.novell.com/docrep/2009/11/4621437_en.pdf`.

[16] Streaming SIMD extensions. `http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions`.

[17] Intel C++ compiler for Linux intrinsics reference. `http://www.cs.fsu.edu/~engelen/courses/HPC-adv/intref_cls.pdf`.

[18] Auto-vectorization in GCC. http://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[19] OpenMP. http://openmp.org/wp/about-openmp/.

[20] Pthreads. https://computing.llnl.gov/tutorials/pthreads/.

[21] MPI. http://www.open-mpi.org/.

[22] Nvidia CUDA. http://www.nvidia.com/object/what_is_cuda_new.html.

[23] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, (2nd GPGPU'09) in conjunctions with (14th ASPLOS'09)*, volume 383 of *ACM International Conference Proceeding Series*, pages 79–84, Washington, DC, USA, March 2009. ACM.

[24] Intel C++ Floating-point Operations. http://cache-www.intel.com/cd/00/00/34/76/347605_347605.pdf.