Operación stencil en CUDA

Autor:
Bernardo Garcés Chapero

Directores:
José Ramón Herrero Zaragoza
Beatriz Otero Calviño

Centro:
Facultat d'Informàtica de Barcelona (FIB)

Universidad:
Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

Julio, 2011

Capítulo 1

Introducción

1.1. Dispositivos *many-core*

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 coprocesamiento gráfico con varios cientos de núcleos que permiten asignar tareas generales a estos. Muchos algoritmos están siendo adaptados a estos modelos de paralelización.

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 en 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*) [1].

El tipo de paralelismo que utilizan estos dispositivos es conocido como SIMT (*Single Instruction Multiple Thread*), que 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 sistemas *multi-core*.

Esta clase de paralelismo es ventajoso en determinados problemas en los que se puede explotar el paralelismo de los datos cuando existen pocas dependen-

cias 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 [2]. Los gráficos de la Figura 1.1 muestran la tendencia de la productividad y el ancho de banda de las últimas generaciones de GPUs Nvidia ante las CPUs convencionales.

Para poder llevar a cabo implementaciones de problemas de propósito general sobre GPUs, es necesario el uso de dispositivos preparados para ello. En el caso de Nvidia, las GPUs diseñadas con arquitectura CUDA son capaces de ejecutar este tipo de problemas mediante el uso de unas librerías.

1.2. Operación stencil

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

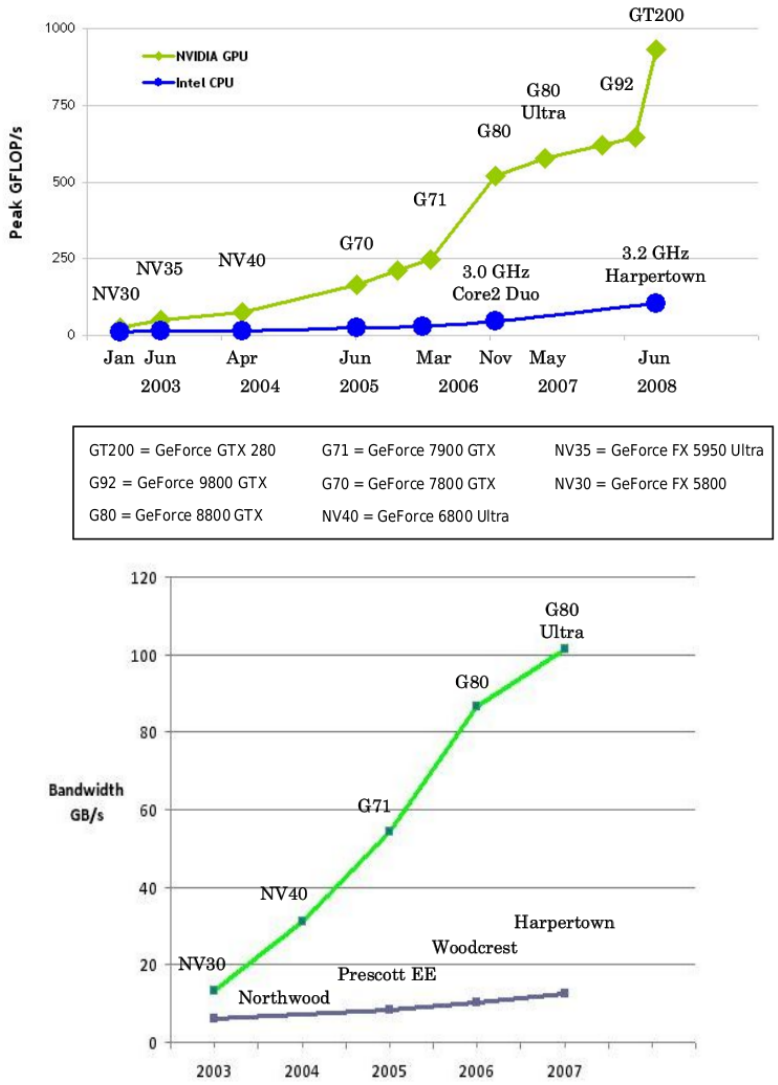

Figura 1.1: El gráfico superior muestra la evolución de la productividad en millones de operaciones de coma flotante por segundo. El gráfico inferior muestra la evolución del ancho de banda [3].

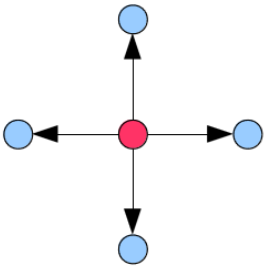

Figura 1.2: Representación de los elementos implicados en el cálculo de un *stencil* de 5 puntos.

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 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.

1.2.1. *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 que supone un total de 5 operaciones en coma flotante por cada *stencil* calculado. El tipo de iteración utilizado es Jacobi, por lo que la matriz sobre la que se requiere de una matriz de lectura y otra de escritura. Las fronteras de la matriz representan valores constantes, por lo que durante la ejecución del algoritmo no se actualizarán sus valores. 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 1.2):

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

1.2.2. *Stencil* de 27 puntos

El *stencil* de 27 puntos es un *stencil* tridimensional, de primer orden, de tamaño 27 y que supone un total de 30 operaciones en coma flotante por cada *stencil* calculado. El tipo de iteración utilizado es Jacobi, por lo que la matriz sobre la que se requiere de una matriz de lectura y otra de escritura. 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

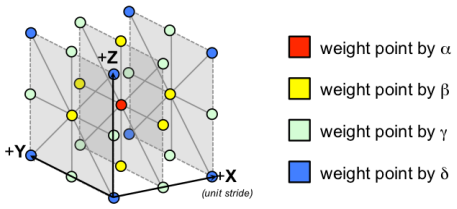

Figura 1.3: Representación de los elementos que engloban un *stencil* de 27 puntos. Los colores de los elementos indican como están ponderados para el cálculo (imagen de[4]).

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 1.3):

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

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

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

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

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

1.3. Descripción del trabajo

El trabajo que se ha llevado a cabo consiste en el estudio, implementación y optimización de dos operaciones stencil para que se ejecuten en una GPU con arquitectura CUDA. La GPU empleada es una Nvidia GeForce GTX 295 con 240 núcleos CUDA, 896 MB de memoria y CC (*Compute Capabilty*) 1.3.

La lista de objetivos es:

- Decidir que operaciones *stencil* van a implementarse.
- Implementar, probar y optimizar las operaciones.
- Obtener conclusiones del rendimiento.

Capítulo 2

Implementación CUDA

2.1. Descripción

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 [5].

Este modelo 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.

Dicho enfoque se utiliza para que cada hilo se encargue de una porción muy pequeña del problema, que puede ser incluso un solo elemento. Esto implica que el algoritmo secuencial tenga que replantearse por completo, buscando 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 uno de sus múltiples núcleos, 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*.

2.2. Arquitectura

CUDA requiere de un enfoque para la resolución de problemas diferente al tradicional. 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 2.1).

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

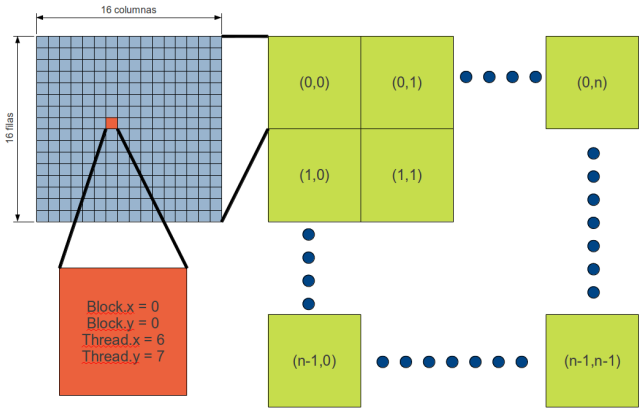

Figura 2.1: La figura ilustra como se trabaja con CUDA para paralelizar un problema. A la derecha se encuentra la matriz de bloques de hilo, donde cada elemento es un bloque que contiene una cantidad determinada de hilos repartidos en filas y columnas. Finalmente, puede verse que cada hilo puede conocer su posición dentro del problema gracias a los identificadores.

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.

2.3. Lineas generales para las implementaciones

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 bidimen-

Algoritmo 2.1 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();
```

sional 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 2.1.

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

2.3.1. 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 y cada hilo de CUDA, quedando los bordes no computables de la matriz fuera de la estructura de hilos de CUDA (Figura 2.2).

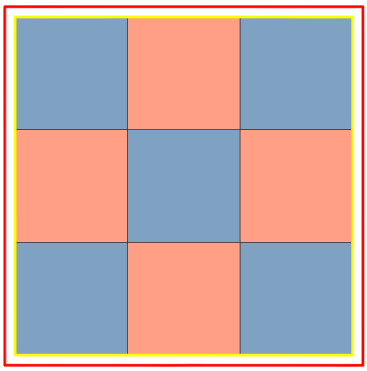

Figura 2.2: Mapeo directo de la estructura de hilos de CUDA y los elementos computables de la matriz de datos. La linea roja muestra el límite de la matriz de datos y la linea amarilla deja fuera a los bordes no computables de la matriz. Dentro de los elementos computables, se puede apreciar el esquema 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 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 2.3).

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

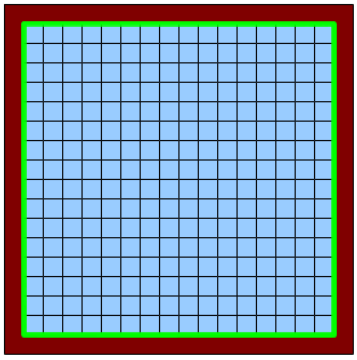

Figura 2.3: La figura ilustra la situación dentro de un bloque de hilos en el cálculo de un *stencil*. En azul se muestran los elementos dentro del bloque de hilos. Fuera del bloque de hilos, se muestra los elementos que también son necesarios para llevar a cabo el cálculo de todos los *stencils*, pues los elementos de los bordes dentro del bloque de hilos los requieren en el cálculo.

de 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. 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 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 [6].

2.3.2. 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

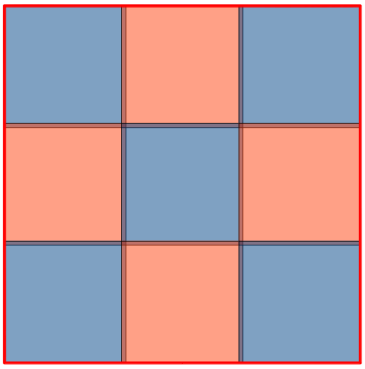

Figura 2.4: Mapeo solapado de la estructura de hilos en CUDA y la matriz de datos. La linea roja muestra el limite de la matriz de datos y puede verse como todos los elementos de la matriz están asignados a algún hilo. Utilizando esta opción los bloques de hilos deben solapar dos de sus elementos con el bloque adyacente, ya que dichos elementos han de ser accedidos desde ambos bloques.

memoria. Por este motivo, los bloques de hilos adyacentes están solapados unos con otros, porque algunos elementos tienen que leerse desde diferentes bloques (Figura 2.4 y Figura 2.5).

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.

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

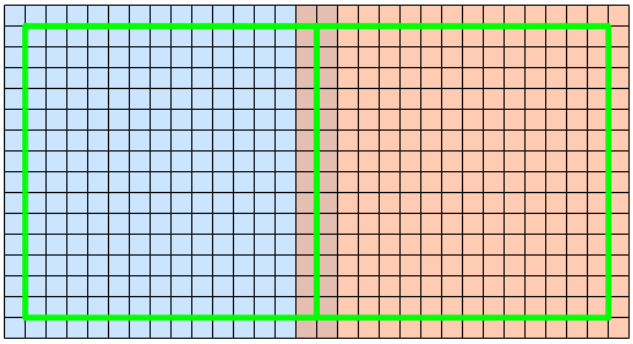

Figura 2.5: En la figura pueden verse dos bloques de hilos de 16 filas por 16 columnas. En el interior del área verde se muestran los elementos que se van a computar, de ahí que sea necesario solapar dos de las columnas de los bloques para que el algoritmo llegue a todos los elementos.

Capítulo 3

Resultados y conclusiones

3.1. *Stencil* de 5 puntos

En esta sección se mostrarán 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.

3.1.1. 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 efec-

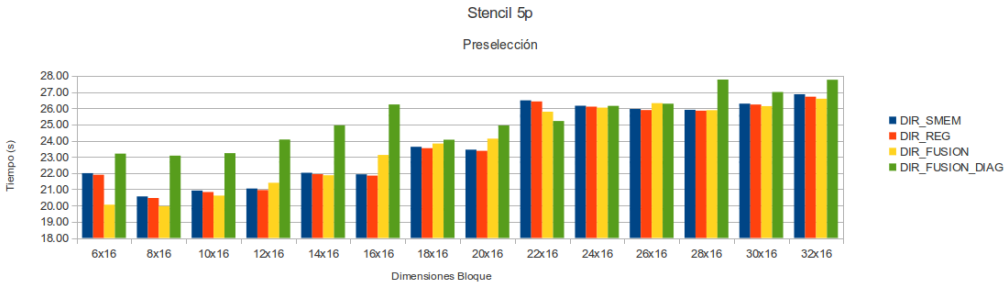

Figura 3.1: Gráfico de resultados del *stencil* de 5 puntos con método directo. Con los valores del eje vertical aumentados acotando el rango para poder apreciar cual es la mejor ejecución y seleccionarla, con el fin de poderla comparar con el resto de implementaciones.

tos 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 3.1).

3.1.2. 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

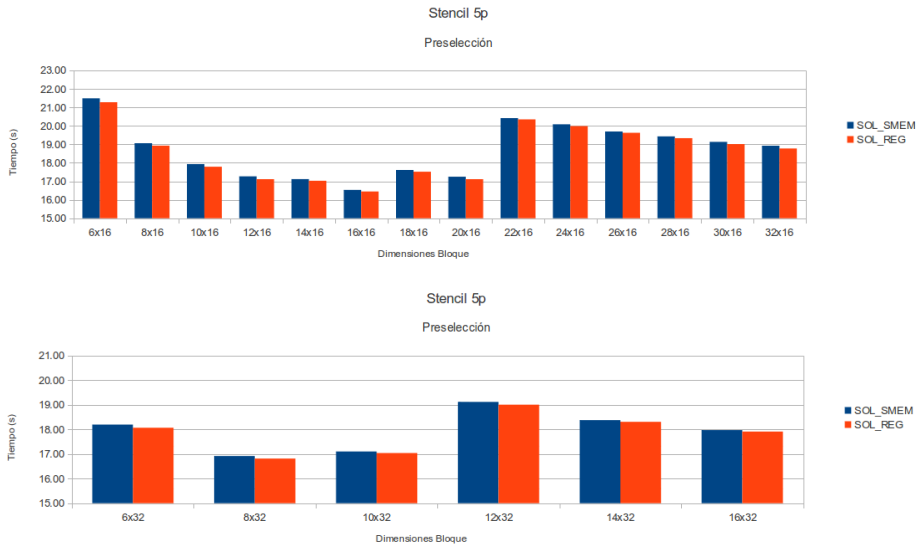

Figura 3.2: Gráficos de resultados del *stencil* de 5 puntos con método solapado y tamaño de columna 16 y 32. Con los valores del eje vertical aumentados acotando el rango para poder apreciar cual es la mejor ejecución y seleccionarla, con el fin de poderla comparar con el resto de implementaciones.

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

Cuadro 3.1: Resultados en tiempo de ejecución y GFlop/s de las mejores ejecuciones de las dos versiones del algoritmo del *stencil* de 5 puntos.

(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 3.2).

3.1.3. 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 no sean optimizaciones en el acceso a los datos.

En la Figura 3.3 y el Cuadro 3.1 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 %.

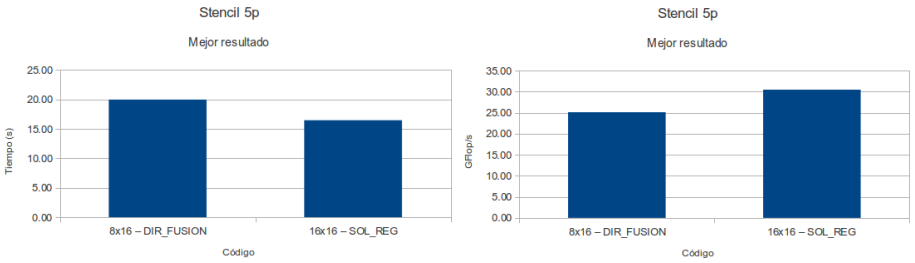

Figura 3.3: Gráficos que muestran los resultados finales con las mejores ejecuciones de las dos versiones del algoritmo del *stencil* de 5 puntos.

3.2. *Stencil* de 27 puntos

En esta sección, al igual que se ha hecho en la anterior, se mostrarán los resultados de las diferentes implementaciones del *stencil* de 27 puntos. El procedimiento es el mismo que en el *stencil* anterior, 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.

3.2.1. 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

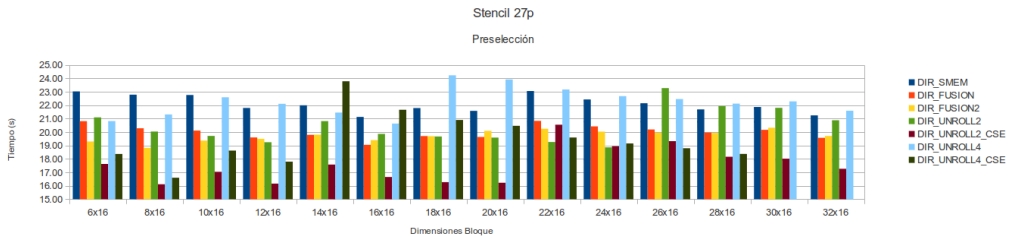

Figura 3.4: Gráfico de resultados del *stencil* de 27 puntos con método directo. Con los valores del eje vertical aumentados acotando el rango para poder apreciar cual es la mejor ejecución y seleccionarla, con el fin de poderla comparar con el resto de implementaciones.

se produce una reducción significativa del tiempo de ejecución (Figura 3.4).

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).

3.2.2. 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 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 3.5).

3.2.3. 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.

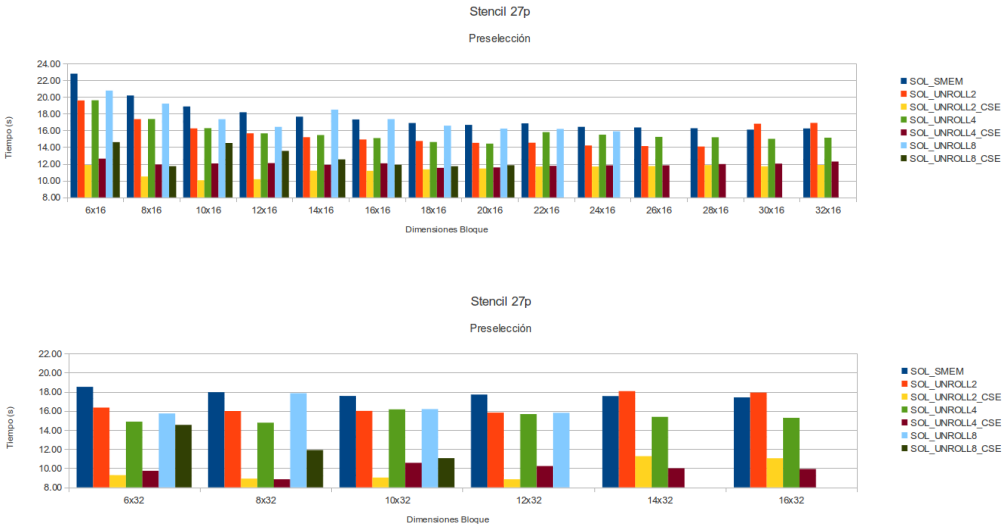

Figura 3.5: Gráficos de resultados del *stencil* de 27 puntos con método solapado y tamaño de columna 16 y 32. Con los valores del eje vertical aumentados acotando el rango para poder apreciar cual es la mejor ejecución y seleccionarla, con el fin de poderla comparar con el resto de implementaciones.

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

Cuadro 3.2: Resultados en tiempo de ejecución y GFlop/s de las mejores ejecuciones de las dos versiones del algoritmo del *stencil* de 27 puntos.

En la Figura 3.6 y el Cuadro 3.2 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 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 una buena parte del algoritmo.

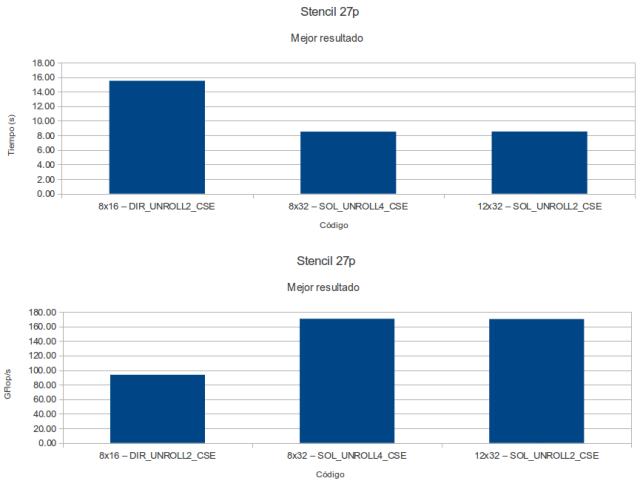

Figura 3.6: Gráficos que muestran los resultados finales con las mejores ejecuciones de las dos versiones del algoritmo del *stencil* de 27 puntos. De la versión solapada se dan dos resultados diferentes porque los dos han dado tiempos de ejecución prácticamente idénticos.

Bibliografía

[1] 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.

[2] 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.

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

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

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

[6] 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.