Trabajo Fin de Máster
en Ingeniería de Computadores.
Curso 2010-2011.

Análisis del uso del Polyhedral Model en herramientas de generación automática de código CUDA

Convocatoria:
Septiembre 2011
Calificación:
Sobresaliente (9)

Autor:
Daniel Tabas Madrid

Director:
Christian Tenllado van der Reijden
Colaborador externo de dirección:
Carlos García Sánchez

Máster en Investigación en Informática.
Facultad de Informática.
Universidad Complutense de Madrid.

Análisis del uso del Polyhedral Model en herramientas de generación automática de código CUDA .

Memoria de Trabajo fin de Master presentada por Daniel Tabas Madrid en la Universidad Complutense de Madrid, realizado bajo la dirección de Christian Tenllado van der Reijden y la colaboración externa de dirección de Carlos Sánchez García.

Madrid, a 1 de octubre de 2011.

Autorización

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales, y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Análisis del uso del Polyhedral Model en herramientas de generación automática de código CUDA", realizado durante el curso académico 2010-2011 bajo la dirección de Christian Tenllado van der Reijden [y con la colaboración externa de dirección de Carlos García Sánchez] en el departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objetivo de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Daniel Tabas Madrid

A mi familia, novia, amigos, en definitiva,
a toda la gente que me ha apoyado para que este
trabajo saliera adelante.

Agradecimientos

Quiero dar las gracias a mis compañeros (y excompañeros) de laboratorio, en especial a Jorge y a Roberto, y también a Carlos por su inestimable ayuda para comprender mejor el funcionamiento de la herramienta que está desarrollando. También quiero dar las gracias a Nacho, ya que sin él no hubiera podido realizar este trabajo.

Resumen

En los últimos tiempos ha habido avances muy importantes en compiladores para arquitecturas paralelas motivados en parte por la consolidación de procesadores multicore en el ámbito de los procesadores de propósito general. También hemos sido testigos de la profunda evolución de los chips gráficos, en un primer momento empleados únicamente para procesar y renderizar imágenes para su visualización, pero que en la actualidad se usan también como coprocesadores para cálculo de propósito general.
Generar "buen" código para este tipo de arquitectura es mucho más complejo que en un procesador de propósito general debido principalmente por la complejidad del hardware y su jerarquía de memoria.
Ante este problema surge el reto de desarrollar una herramienta que de forma automática o semiautomática guíe al programador en la toma de decisiones con el objetivo de crear una aplicación que explote adecuadamente dicha arquitectura. El *Polyhedral Model* es un framework matemático para optimización de bucles que ha demostrado ser muy útil en la generación de código paralelo para procesadores de propósito general. Existen algunas propuestas encaminadas al empleo de este framework para la generación automática de código en tarjetas gráficas donde también sería de gran de gran utilidad lo que ha motivado el comienzo su desarrollo en este ámbito.
El propósito de este trabajo es analizar estas herramientas, estudiar el código generado, examinar defectos del proceso de generación de código y por último proponer mejoras.

Palabras clave: GPGPU, Polyhedral Model, Compiladores, Optimización, Paralelismo, Transformaciones de código

Abstract

There have been important advances in parallel compilers recently motivated partially by multicore processors consolidation. We have also been witnesses of a deep evolution in graphical hardware, first used to render images, but also used as co-processors for general-purpose computation nowadays.
To generate a "good" code is much more complex in this architecture due to a complex hardware and its particular memory hierarchy.
Faced this problem, it arises a challenge of developing a tool which guide automatically or a least in semiautomatic way to programmers in order to create an efficient application in terms of parallel exploitation. *Polyhedral Model* is a mathematical framework for loop nest optimization that has proven to be useful to reach an efficient parallel code in general purpose processors. There are some proposals to use this framework for automatic code generation in graphic scope which has motivated an interesting early development.
The aim of this work is to analyze these tools, study the code generated, detect potential generation faults, and finally suggest improvements.

Key words: GPGPU, Polyhedral Model, Compilers, Optimization, Parallelism, Code transformations

Índice general

Capítulo 1

Introducción

1.1. Origen de GPGPU

Hasta hace unos años se ha estado viviendo una época en cuanto a tecnología de procesadores, en la que imperaban los diseños mononúcleo muy potentes, con una frecuencia de reloj muy elevada, una alta segmentación, y estructuras muy complejas para especulación de instrucciones.
Los Pentium IV [12] son el paradigma de monoprocesador muy potente, con un hardware para especulación muy sofisticado, con una frecuencia de reloj muy alta. En sucesivos modelos de este procesador se fue aumentando la frecuencia de reloj, aumentando la potencia consumida, llegando al punto en que era difícil mantener bien refrigerado el procesador sólo con aire. Esto hizo llevar a Intel a cambiar totalmente la estrategia, y optar por los Core, procesadores mucho menos complejos, pero con la capacidad de montar varios cores en un mismo chip. Así se consiguió mejorar el rendimiento, bajando de una manera espectacular el consumo.
Por lo tanto, la tendencia de los procesadores, desde la salida de los Intel Core, ha sido la de hacer procesadores más simples pero que integren más elementos de procesamiento [19], es decir, se ha hecho un reparto diferente del uso de los transistores de la oblea de silicio destinados a las diferentes partes del procesador. La mejora principal de estos procesadores es el aumento de productividad que consiguen.
En los últimos años, con este desarrollo de los procesadores multicore, se ha hecho mucho hincapié en desarrollar frameworks, librerías y compiladores optimizados para explotar el paralelismo inherente a la arquitectura multicore. Esto ha sido muy beneficioso para el cálculo científico, ya que en el ámbito científico hay muchas tareas que requieren de una capacidad de cómputo enorme, y además muchas de ellas exponen un nivel de paralelismo muy grande. Es por ello que en muchas ocasiones se ha intentado explotar este paralelismo mediante el uso de grids, clusters, con programas que hacen uso de la memoria compartida. Pero un cluster, y mucho menos un grid, no están al alcance de mucha gente.
Utilizar procesadores multicore para cálculo científico por lo tanto es beneficioso, pero a veces no es suficiente, ya que el número de tareas que puede ejecutar un procesador de este tipo es limitado.

Siguiendo un desarrollo totalmente distinto, los avances en el campo de los chips gráficos, movidos muchas veces por el mercado de los videojuegos, han empezado a hacer de las GPUs una opción barata para hacer cálculo científico, a pesar de que eran inicialmente

coprocesadores específicos para la transformación de vértices y pixeles, comenzando a programar directamente la tarjeta con datos transformados en elementos gráficos, perdiendo así parte de su potencia, además de la dificultad que esto conllevaba.
La tarjetas gráficas aparecen en torno a los años 60, cuando se comienzan a utilizar los monitores como elementos de visualización. Eran dispositivos dedicados encargados de crear las imágenes. En 1995 aparecen las primeras tarjetas 2D/3D con chips de gran potencia de cálculo. Desde entonces las mejoras se han orientado a obtener más potencia de cálculo y proceso de algoritmos 3D.
Una GPU típica [24] tiene una arquitectura altamente segmentada, con un gran número de unidades funcionales. Antiguamente estas unidades funcionales estaban divididas entre las que procesaban vértices (vertex shaders) y las que procesaban pixeles (pixel shaders), lo cual era un problema si se quería utilizar una GPU para algo que no fuera procesar gráficos. Al extenderse el uso de los chips gráficos para tareas distintas al procesado de gráficos, debido a la gran cantidad de unidades funcionales disponibles y la alta velocidad y baja latencia de la memoria, los fabricantes comenzaron a cambiar el diseño de éstos para que pudieran hacer cálculos de propósito general sin perder la funcionalidad de trabajar con vértices y pixeles.
Para ello dotaron a estas unidades funcionales de elementos suficientes como para poder funcionar como vertex o pixel shaders (unified shader), así como de un repertorio de instrucciones suficiente para cálculo científico.
La arquitectura actual de una GPU con soporte para GPGPU nos ofrece varios conjuntos que concentran una parte de las unidades funcionales, llamadas multiprocesadores[1] Cada uno de estos multiprocesadores tiene una caché de datos y una unidad de texturas a compartir entre las unidades funcionales que lo forman. Se comunican a través de un bus con varias unidades Load/Store que permiten la transferencia de datos con la memoria global de la tarjeta. Consta además de un gestor de ejecución de hilos, que se encarga del reparto de trabajo. Es por tanto necesario que las aplicaciones exploten esta distribución para aprovechar al máximo los múltiples procesadores de los que se disponen.
Por lo tanto la clave de la arquitectura de las GPU reside en el uso de múltiples procesadores escalares sobre un flujo masivo de datos. Estos procesadores se agrupan para hacer cálculos sobre datos próximos entre sí para aprovechar al máximo el paralelismo y proporcionar una capacidad de cálculo muy potente. Poseen una lógica de decodificación y ejecución integrada de alta velocidad, y la memoria de cada chip puede ser leída muy rápidamente.
En la figura 1.1 podemos ver la diferencia entre CPUs y GPUS en área de los distintos

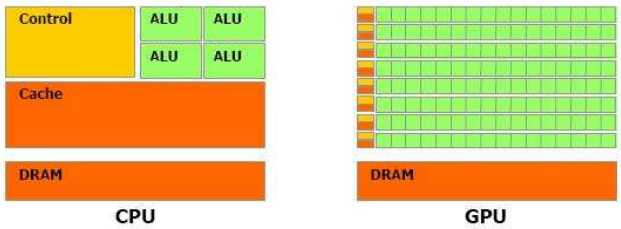


Figura 1.1: Estructura CPU y GPU

elementos que las conforman. Esta diferencia es debida a la tremenda capacidad de cómputo en punto flotante de las tarjetas gráficas, ya que una GPU está especializada en tareas intensivas en cómputo que se realiza de forma paralela. Por lo tanto hay más transistores

[1]También llamados SP, streaming processors.

dedicados al procesamiento de datos en lugar de utilizarse para control de flujo o cacheo de datos.

1.2. Necesidad de herramientas

Como ya se ha comentado en la sección anterior, con el auge de los procesadores multicore, se ha desarrollado una gran cantidad de herramientas para facilitar su uso de forma eficiente. Desde por ejemplo, los Posix Threads, que establecen un estándar, y definen una API para crear y manejar hilos, a APIs de más alto nivel, como OpenMP [3], que abstraen la creación y manipulación de hilos, librerías optimizadas que utilizan múltiples hilos, etc. Además de esto, los compiladores han ido evolucionando teniendo en cuenta estas arquitecturas, realizando multitud de transformaciones de código con el fin de mejorar la calidad de los binarios generados, por ejemplo, minimizando el uso de registros, mejorando el patrón de acceso en bucles, etc.

Pero hasta ahora estos compiladores, en la fase de optimización del código se han dedicado a realizar transformaciones conocidas en el mismo, limitando muchas veces la posibilidad de realizar otras transformaciones. Los bucles son secciones de código muy estudiadas, y con muchos tipos de transformaciones conocidas, como el loop tiling, loop unrolling, loop fusion, etc, en los que cada transformación por separado puede mejorar la eficiencia en la ejecución del código. Pero realizar todas estas transformaciones de forma automática es una tarea muy compleja, ya que el orden de las transformaciones es un factor muy importante, y se puede llegar a una explosión de código debido a una secuencia muy grande de transformaciones, además de que hay muchas secuencias diferentes que llegan al mismo código, lo que produce un conjunto ingente de programas candidatos transformados imposible de construir en un tiempo razonable.

Desde hace un tiempo se viene investigando en un enfoque diferente: transformar determinadas partes del código, llamadas SCoP (Static Control Parts), para que puedan ser representadas algebráicamente mediante un conjunto de inecuaciones. A esto se le llama representación poliédrica, o *Polyhedral Model*. Para modelar el código de esta forma se necesitan inecuaciones que representen el dominio de iteración, las funciones de acceso y las dependencias entre sentencias. Con esto se consigue un conjunto de inecuaciones, que al resolverse mediante programación lineal entera, nos devuelve un espacio que representa el conjunto de transformaciones de ese código que preserva la semántica original del programa ,es decir, que es legal. Por lo tanto tenemos un espacio legal de transformaciones, que todavía es enorme como para explorarlo entero. Pero cualquier subconjunto dentro de este espacio se puede transformar otra vez a código, habiendo respetado todas las dependencias entre statements.

Podemos decir por lo tanto, que todavía queda mucho camino que recorrer en el desarrollo de compiladores para generar códigos lo más eficientes posible para estas arquitecturas multicore.
Pero actualmente para la computación científica, como se ha comentado en la sección anterior, se ha empezado a hacer uso de las tarjetas gráficas como coprocesadores, debido a la gran cantidad de unidades de proceso que albergan, y al potencial del sistema de memoria que contiene. Si hemos dicho que todavía queda camino por recorrer a los compiladores que generan código para CPU, no hablemos ya de la generación de código para estas tarjetas gráficas.

Existen algunos frameworks, como CUDA[22, 23], para tarjetas de Nvidia, y openCL, que es una propuesta de unificación para los frameworks ya existentes, que ya facilitan en cierta forma la generación de código para GPUs, ya que incluyen abstracciones para la división de tareas en hilos que se ejecutarán en las diferentes unidades funcionales, o para utilizar los diferentes espacios de memoria de una forma más cómoda, todo ello para ser utilizado con un compilador propio, que es capaz de realizar algunas pequeñas optimizaciones.
Pero de momento los sistemas de memoria de la tarjeta no se pueden usar de forma transparente, como en la programación en CPU, ya que incluso hay espacios de memoria que se usan como una scratchpad, y la cantidad de unidades funcionales que conforman la GPU hacen más difícil la distribución de las tareas en hilos.
Por lo tanto, el desarrollo de aplicaciones utilizando tarjetas gráficas como coprocesadores requiere un gran esfuerzo por parte del programador, ya que en él recae todo el peso de intentar aprovechar la ventaja de ser una plataforma masivamente paralela en el modelado del problema.
Dado un algoritmo específico, el espacio de búsqueda para el mapeo es enorme. Normalmente la metodología de un programador consiste en evaluar varias alternativas de mapeo, guiado por su experiencia e intuición, lo que resulta ineficiente para el desarrollo software y el mantenimiento.

1.3. Objetivos

Existen ya algunas herramientas basadas en el *Polyhedral Model* que permiten realizar transformaciones en código para CPU, de momento en compiladores *source-to-source*, generando de nuevo código que será procesado por otro compilador, y que están dando buenos resultados. También hay proyectos de compiladores (GCC, LLVM) bastante avanzados en los que se está incluyendo este modelo, para realizar directamente las transformaciones y la generación del código máquina en un mismo compilador.
Ya que producir código eficiente para tarjetas gráficas es una tarea complicada para el programador, algunos grupos de investigación pensaron en utilizar el *Polyhedral Model* para generar código de forma automática para GPUs. Parte de esta generación implicaría transformaciones de bloqueo del código para adaptarlo a la jerarquía de hilos existente en el modelo de programación de tarjetas gráficas. También conllevaría un análisis adicional del reuso de los datos, para poder aprovechar la jerarquía de memoria de las GPUs de forma eficiente, además de otras cosas como la sincronización de hilos, bloques, y definir de forma óptima qué tarea tiene que hacer un hilo.
Estos grupos de investigación están desarrollando una serie de herramientas para hacer esto, algunas basadas en herramientas para generar código para CPU. El camino a seguir por estos grupos es todavía muy largo, ya que estas herramientas sólo generan código para una serie de casos, y ayudándose muchas veces de información que tiene que proporcionar el programador.

Por lo tanto, el objetivo principal de este trabajo es el análisis de estas herramientas, un análisis fundamentalmente práctico, basado en mostrar las diferencias entre un código eficiente escrito a mano, y uno generado con herramientas de este tipo. Este análisis persigue la finalidad de encontrar los puntos débiles de estas herramientas, en cómo por ejemplo se hace la subdivisión del problema en tareas que se mapearán en hilos, o cómo son capaces de aprovechar la jerarquía de memoria, para poder mejorarlas en un futuro.

Capítulo 2

GPU

En esta sección se van a explicar las características, el modelo de programación y el modelo de memoria que se han desarrollado para tarjetas gráficas para permitir el GPGPU. Como se ha comentado anteriormente, la arquitectura de las tarjetas gráficas se ha modificado para poder hacer uso de ellas como un coprocesador de propósito general. CUDA[22, 23] es modelo de programación paralela para tarjetas gráficas más famoso, y todas las herramientas que se van a testear generan código CUDA, pero se va a tratar de explicar de la forma más general posible.

2.1. Arquitectura tarjetas

En las GPUs se introduce el modelo de ejecución SIMT[1]. La arquitectura SIMT es semejante a la organización vectorial SIMD[2], en una única instrucción que controla múltiples elementos a procesar. La diferencia clave es que la organización vectorial SIMD expone el ancho de SIMD al software, mientras que las instrucciones SIMT especifican el comportamiento de ejecución y ramificación de un único hilo. En contraste con máquinas vectoriales SIMD, SIMT permite a los programadores escribir código paralelo a nivel de hilo para hilos independientes, escalares, tanto como código de datos paralelos para hilos coordinados. Las arquitecturas vectoriales, por otra parte, requieren que el software controle la carga entre vectores y su divergencia manualmente.

Cada multiprocesador, como se puede observar en la Figura 2.1, tiene un chip integrado de memoria de uno de los cuatro siguientes tipos:

- Un set de registros de 32-bits por procesador.
- Una unidad de instrucciones común.
- Una cache de datos paralela o memoria compartida que es accedida por todos los núcleos escalares del procesador y que es donde residen los espacios de memoria compartida.
- Una cache constante de sólo lectura compartida por todos los núcleos escalares del procesador y que acelera las lecturas del espacio de memoria constante, la cual es una región de sólo lectura de la memoria del dispositivo.

[1] Single-instruction multiple-thread
[2] Single-instruction, multiple-data

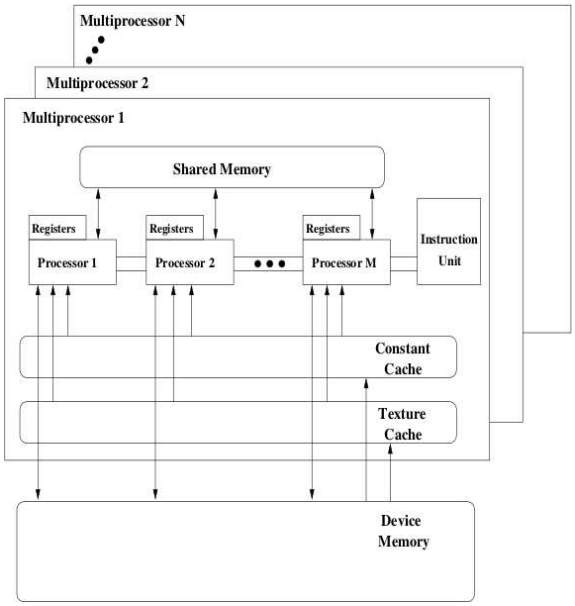


FIGURA 2.1: Un grupo de multiprocesadores Single Instruction Multiple Thread

- Una cache de texturas de sólo lectura que es compartida por todos los núcleos escalares del procesador y que acelera las lecturas del espacio de memoria de texturas.

Varios de estos multiprocesadores conviven en una tarjeta gráfica, y comparten además una DRAM común.

2.2. Modelo de programación

El componente principal del modelo de programación para GPUs son las funciones que se invocan desde el *host*, pero se ejecutan en el dispositivo. Dichas funciones reciben el nombre de *kernels*. Un kernel ejecuta un código secuencial en un número muy grande de hilos de forma paralela.

Todos los hilos creados durante la ejecución de un kernel ejecutan el mismo grupo de instrucciones. El número de hilos debe ser conocido de antemano de manera que puedan ser tratados por grupos, todos ellos del mismo tamaño, y de acuerdo a lo especificado por el programador en la llamada al kernel.

Los hilos como ya se ha mencionado, se organizan agrupándose en bloques de hasta tres dimensiones, que se utilizarán según las necesidades del programador. La organización en bloques permite que los hilos pertenecientes a un mismo bloque puedan cooperar entre sí compartiendo datos en un espacio de memoria llamado memoria compartida. También se puede coordinar su ejecución a través de funciones de sincronización, que actúan como barrera ante la cual los hilos de un bloque tienen que esperar a que el resto llegue para poder continuar con su ejecución. La única forma de sincronización que existe por lo tanto es entre hilos de un mismo bloque, no existen funciones de sincronización entre bloques, lo que condiciona la forma en la que se programa utilizando este modelo.

Hay un límite en el número de hilos por bloque, ya que se espera que residan en el mismo multiprocesador y deben compartir los recursos de memoria que les ofrece, que son limitado. En GPUs actuales, un bloque puede contener hasta 1024 hilos, y en arquitecturas algo más antiguas, hasta 512. Para contrarrestar esta limitación, un kernel puede ser ejecutado por múltiples bloques de hilos de igual forma. De esta manera, el número total de hilos en ejecución será igual al número de bloques multiplicado por el número de hilos en cada bloque.

Cada hilo puede ser identificado usando un índice unidimensional, bidimensional o tridimensional, formando un bloque de hilos unidimensionales, bidimensionales o tridimensionales según queramos representar el espacio del problema. Esto nos proporciona una forma natural de llamar los elementos en un dominio como puede ser un vector, matriz o volumen, además de que de esta manera se puede definir el comportamiento específico de cada uno de los hilos.

El índice de un hilo y su ID de hilo están relacionados entre sí de la siguiente forma: para un bloque unidimensional, son iguales; para un bloque bidimensional de tamaño (D_x, D_y), el ID de hilo de índice (x, y) es $(x + yD_x)$; para un bloque tridimensional de tamaño (D_x, D_y, D_z), el ID de hilo de índice (x, y, z) es $(x + yD_x + zD_xD_y)$.

La forma de organizar los bloques es mediante una rejilla, que puede ser unidimensional o bidimensional. La organización y tamaño de la rejilla de bloques es el primer parámetro especial que se le pasa a un kernel cuando se invoca. El segundo parámetro es el tamaño y la organización que va a tener cada bloque. En el caso de CUDA, estos parámetros van encerrados dentro del operador especial <<< ... >>>.

Es importante destacar que la ejecución de cada bloque debe ser independiente. Tiene que ser posible ejecutar cada bloque en cualquier orden. Este requisito de independencia permite que cada bloque de hilos sea asignado a cualquier procesador y en cualquier orden, lo que aporta escalabilidad al código. El número de bloques de una rejilla viene determinado por el tamaño de los datos que se van a procesar y no por el número de procesadores de los que dispone el sistema.

Por lo tanto, como resumen, un kernel divide su ejecución de la siguiente forma:

- Grid: la división inicial que se aplica sobre la jerarquía de hilos. Un Grid está formado por un conjunto uni, bi, o tridimensional de bloques. El número de bloques en un grid normalmente está condicionado por el tamaño del conjunto de datos que va a ser procesado.
- Bloques: conjuntos uni o bidimensionales de hilos que componen el grid, se tratan de lanzar paralelamente.
- Hilos: es la unidad de trabajo básica de los kernels. Cada hilo realiza trabajo sobre uno de los elementos del objeto inicial. Si el objeto inicial es una matriz podemos tener que cada hilo trabaja sobre un elemento $a_{n,m}$ de la matriz.

La creación de hilos y la planificación se realiza enteramente en hardware. Cada multiprocesador crea, maneja, planifica y ejecuta hilos en grupos de 32 hilos paralelos llamados warps. Los hilos que componen un warp comienzan juntos en la misma dirección de programa, pero tienen su propio contador de dirección de instrucción y registro de estado para poder ejecutarse independientemente. Cuando un multiprocesador recibe bloques para la ejecución, los particiona en warps que se planifican para la ejecución. Un warp ejecuta una

instrucción común cada vez, por lo que para maximizar la eficiencia los 32 hilos deben coincidir en su camino de ejecución. Si divergen, el warp ejecuta de forma serializada cada rama del salto, deshabilitando los hilos que no están en ese camino, y una vez todos los caminos se hayan completado, los hilos vuelven a converger.

2.3. Jerarquía de memoria

Los hilos pueden acceder a los datos de múltiples espacios de memoria durante su ejecución. Cada hilo tiene una memoria local privada y cada bloque de hilos tiene una memoria compartida visible por todos los hilos del mismo bloque y en el mismo tiempo de ejecución del bloque. Finalmente, todos los hilos pueden acceder a la misma memoria global.
Hay dos espacios de memoria de sólo lectura adicionales accesibles por todos los hilos: los espacios de memoria constante y de texturas. Los espacios de memoria global, constante y de texturas están optimizadas para diferentes usos de memoria. La memoria de texturas además ofrece distintos tipos de direccionamiento, como filtrado de datos, para algunos tipos de datos específicos. Los espacios de memoria global, constante y de texturas son persistentes a través de los lanzamientos de kernels de la misma aplicación.
No todos estos espacios de memoria son independientes, algunos de ellos son espacios lógicos dentro de un mismo tipo de memoria, tales como memoria constante y de texturas, que se mapean dentro de la memoria global, utilizando un cacheo para la de texturas.

Memoria global Estos tres espacios de memoria de acceso global están optimizados para diferentes usos. Asimismo, son persistentes a través de los diferentes kernels que ejecute una misma aplicación. El espacio de memoria global no es cacheado y es capaz de leer palabras de 2, 4 u 8 bytesen una única instrucción de lectura. El espacio de constantes está cacheado, el valor solicitado se lee desde la cache de constantes y sólo se lee la memoria global en un fallo de cache. Leer de la cache de constantes puede ser tan rápido como leer de un registro, siempre que todos los hilos de un warp lean la misma dirección. El espacio de texturas, al igual que el de constantes, está cacheado, pero con la diferencia de que está optimizado para localidad espacial 2D. De modo que los hilos pertenecientes a un mismo warp que lean direcciones que estén cercanas conseguirán un mejor rendimiento. Además tiene un tiempo de acceso constante, independientemente de si el dato se lee de cache o de memoria; la única diferencia es que reduce el ancho de banda utilizado con la memoria. De esta manera, resulta ventajoso leer datos del espacio de texturas en lugar de hacerlo del espacio de memoria global o de constantes.

Memoria compartida La memoria compartida (o shared) se trata de una memoria común a cada bloque de hilos. Está construida on-chip, con lo cual un acceso a ella es mucho más rápido que uno a memoria global. De hecho, para todos los hilos de un warp, acceder a la memoria compartida es tan rápido como acceder a un registro, siempre y cuando no haya conflictos de bancos entre los hilos. Para obtener un mayor ancho de banda, la memoria compartida de cada multiprocesador se encuentra dividida en módulos de memoria de igual tamaño llamados bancos, que pueden ser accedidos simultáneamente. De esta forma, la memoria puede atender con éxito un pedido de lectura o escritura constituido por n direcciones que pertenezcan a n bancos distintos. Así se consigue un ancho de banda n veces mayor al de un banco simple. Ahora bien, si dos direcciones de una petición caen en el mismo banco de memoria, hay un conflicto y el acceso tiene que ser serializado. El hardware entonces divide el acceso a memoria en tantos accesos libres de conflicto como sean necesarios, perdiendo obviamente ancho de banda. Para conseguir el máximo rendimiento, es

importante conocer la organización de los bancos de memoria, y cómo se organiza el espacio de direcciones. Así se podrán programar los accesos a memoria compartida de manera que se minimicen los conflictos. Los bancos están organizados de forma que palabras sucesivas de 32-bits pertenecen a bancos sucesivos (distintos). El ancho de banda de cada banco es de 32-bits cada 3 ciclos de reloj. El número de bancos existentes en una tarjeta gráfica es de 16.

La memoria compartida también ofrece un mecanismo de transmisión (broadcast) donde una palabra de 32 bits se puede leer y transmitir a varios hilos simultáneamente cuando se sirve una sola petición de lectura de memoria. Esto reduce el número de conflictos de banco cuando muchos hilos de un warp leen de una dirección con la misma palabra de 32 bits. Más precisamente, una petición de lectura de memoria hecha por muchas direcciones se sirve en muchos pasos en el tiempo, un paso cada dos ciclos de reloj, dando un subconjunto de servicios libres de conflicto de estas direcciones por paso hasta que todas las direcciones son servidas. A cada paso, el subconjunto se construye con las sobrantes direcciones de memoria que todavía no se han dado usando el siguiente proceso:

- Seleccionar una de las palabras apuntadas por las direcciones sobrantes de memoria como la palabra a transmitir.
- Incluir en el subconjunto:
  - Todas las direcciones que están dentro de la palabra a emitir.
  - Una dirección para cada banco apuntando a el resto de las direcciones.

Qué palabra se elige como la palabra a emitir y cual es la dirección seleccionada para cada banco en cada ciclo no esta especificado.

Registros Generalmente, acceder a los registros no significa ningún ciclo de reloj por instrucción, pero pueden ocurrir retrasos debido a dependencias de lectura después de escritura de registro y conflictos de banco de memoria de registros. Este retraso introducido por dependencias se puede ignorar tan pronto como haya al menos 192 hilos activos por multiprocesador para esconderlos. El compilador y el planificador de hilos planifican las instrucciones de forma óptima para evitar tantos conflictos de banco de memoria de registros como sea posible. Se obtienen mejores resultados cuando el número de hilos por bloque es múltiplo de 64.

Host y dispositivo Este modelo de programación asume que los hilos se ejecutan en un *device* separado físicamente que opera como un coprocesador del *host* que ejecuta el programa en C.
Podemos considerar la memoria de la CPU como un nivel más externo de memoria. Se mantienen sus propios espacios separados de memoria, referidos como memoria del *host* y memoria del *device*. Por lo tanto desde el programa en C se tienen que manejar estos espacios de memoria de la tarjeta, lo que incluye la asignación, desasignación y transferencias de memoria entre las memorias del *host* y *device*.
Las transferencias CPU-GPU son las más costosas, por lo que es importante minimizar estas transacciones. Además hemos de tener en cuenta que una vez que se lanza un kernel no es posible realizar estas comunicaciones.

Capítulo 3

Polyhedral Model y Herramientas

En este capítulo presentamos la base teórica del *Polyhedral Model* y las herramientas se han ido desarrollando y hacen uso del mismo.

3.1. Polyhedral model

Observemos este código de ejemplo:

```
for(i=1; i<N; i++)
   for(j=1; j<N; j++)
      A[i][j] = A[i-1][j] + A[i][j-1]
```

En este código hay una sentencia o *statement*, que se ejecuta N^2 veces. A cada una de estas instancias de la sentencia podemos llamarla operación. Lo que define a una operación es el diferente valor para i y j asociados a la misma.
También existe un orden (orden de ejecución), que puede modificarse, pero debe tenerse en cuenta debido a las dependencias de datos que contiene.
Por lo tanto necesitamos encontrar una representación que considere cada operación, que incorpore dependencias entre operaciones, que haga posible tratar con bucles paramétricos e infinitos, que simplifique la transformación y optimización del código, que permita una paralelización, y pueda regenerar el código desde esa representación. Esta representación es el modelo del poliedro.

En las aplicaciones intensivas en datos por lo tanto, las operaciones, las dependencias, y el orden en que se realizan, son muy importantes. Puede haber millones de operaciones y elementos de datos, pero normalmente es posible detectar mucha regularidad en como son manejados. Esto abre la oportunidad de agrupar elementos para que se traten de un mismo modo.
Como resumen, ya que ahora se extenderá la explicación teórica, en el *Polyhedral Model* cada una de estas sentencias o *statements* que van agrupadas de un modo que ahora veremos, se representan como un nodo, el cual a su vez representa a un poliedro entero en el que cada uno de los puntos de su interior es una operación, o instancia del *statement*, y cada dependencia entre *statements* se representa como una arista entre ellos, la cual a su vez representa un poliedro en el que cada punto es un par de operaciones, cada una de un *statement*, que indica el orden en el que tienen que ir realizandose éstas. En la Figura 3.1 podemos ver un ejemplo de como sería esta representación. Representando el código de esta forma se pueden hacer transformaciones, como veremos a continuación, dirigidas a un

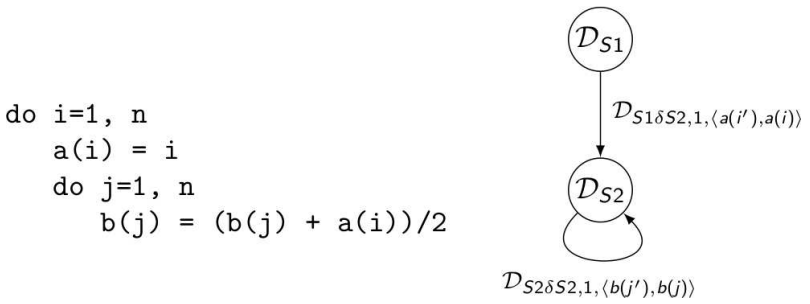


FIGURA 3.1: Representación de *statements* y dependencias mediante un grafo

propósito en concreto, como por ejemplo para acercar la producción de un dato y su uso, y así mejorar el uso de memoria.

3.1.1. Antecedentes y notación

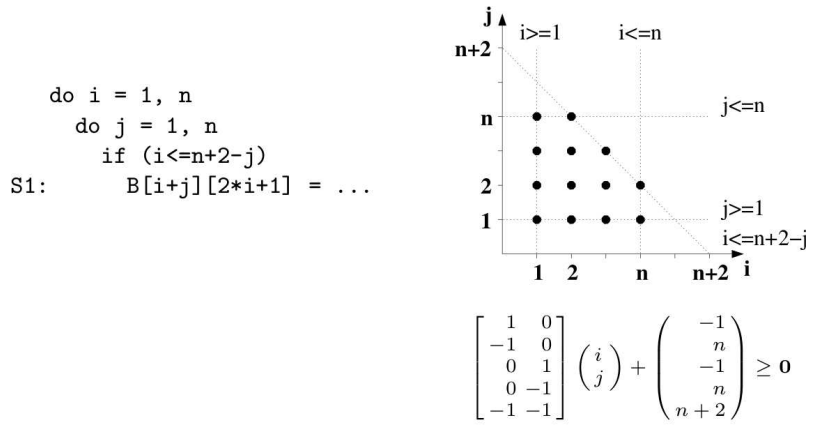
Un bucle en un lenguaje imperativo como C puede ser representado usando un vector columna de n entradas llamado vector de iteración:

$$x = \begin{pmatrix} i_1 \\ i_2 \\ . \\ . \\ . \\ i_n \end{pmatrix},$$

donde i_k es el k-ésimo índice del bucle y n es el bucle más interno. Los condicionales y los bucles que rodean a un *statement* definen su dominio de iteración. El *statement* se ejecuta una vez por cada elemento del dominio de iteración. Cuando los límites del bucle y los condicionales sólo dependen de los contadoradores de los bucles que lo rodean, de parámetros y constantes, el dominio de iteración siempre puede ser especificado por un conjunto de inecuaciones lineales que definen un poliedro [16].
El término poliedro se usa en este sentido para denotar un conjunto de puntos convexo en una rejilla (también llamado $\mathbb{Z}$-poliedro), por ejemplo un conjunto de puntos en un vector en el espacio $\mathbb{Z}$ acotado por inecuaciones afines [26]. Un conjunto maximal de *statements* consecutivos en un programa con un dominio de iteración de este tipo que forma un poliedro se denomina un *static control part*(SCoP) [11].

En la Figura 3.2 se puede observar la correspondencia entre el código original y el dominio del poliedro. Cada punto entero del poliedro corresponde a una operación, es decir, una instancia de un *statement*. La notación $S(\boldsymbol{x})$ se refiere a operación correspondiente al *statement* S con el vector de iteración $\mathbf{x}$. La ejecución de las operaciones sigue un orden lexicográfico. Esto significa que en un poliedro de n dimensiones, la operación correspondiente al punto entero definido por las coordenadas $(a_1...a_n)$ se ejecuta antes que el punto que corresponde a las coordenadas $(b_1...b_n)$ si y sólo si:

$$\exists i, 1 \leq i < n, (a_1...a_i) = (b_1...b_i) \wedge a_{i+1} < b_{i+1}.$$



(a) surrounding control of S_1 (b) iteration domain of S_1

FIGURA 3.2: Código y dominio de iteración

Cada *statement* puede incluir una o varias referencias a arrays, o escalares. Cuando la función relacionada $f(\boldsymbol{x})$ de una referencia es afín, podemos escribirla $f(\boldsymbol{x}) = F\boldsymbol{x} + \boldsymbol{a}$ donde F es la matriz relacionada y **a** es un vector constante. Por ejemplo, la referencia al array B en la Figura 3.2(a) es $B[f(\boldsymbol{x})]$ con $f\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.
En esta sección, las matrices siempre se denotan por letras mayúsculas, los vectores y funciones no. Cuando un elemento es específico de un *statement*, se le añade un subíndice como A_S, exceptuando en general fórmulas donde todos los elementos son específicos de un *statement* para evitar una notación muy pesada.

3.1.2. Transformaciones afines

El objetivo de una transformación es el de modificar el orden original de ejecución de las operaciones. Una forma conveniente de expresar el nuevo orden es dar a cada operación un tiempo virtual de ejecución. Esto nos da una referencia de qué operaciones pueden realizarse a la vez, y cuales deben ir antes o después. Pero hacer esto de una forma muy desestructurada, por ejemplo con una tabla, y asignando tiempos virtuales operación por operación es inabarcable, pudiendo impedir la posible mejora creada con este nuevo orden, por ejemplo, por la cantidad de casos distintos que se generarían, lo que provocaría una explosión de código. Por lo tanto la optimización de compiladores requiere de la construcción de planificaciones a nivel de *statement* encontrando una función que especifique un momento de ejecución para cada instancia del correspondiente *statement*. Las funciones que se eligen para esto son afines por múltiples razones: es el único caso donde podemos decidir exactamente la legalidad de la transformación y donde sabemos cómo generar el código objetivo. Por lo tanto, las funciones de planificación tienen la siguiente forma:

$$\theta_S(\boldsymbol{x}_S) = T_S\boldsymbol{x}_S + \boldsymbol{t}_S, \tag{3.1}$$

donde $\boldsymbol{x}_S$ es el vector de iteración, T_S es una matriz de transformación constante, y $\boldsymbol{t}_S$ es un vector constante. Se ha demostrado ampliamente que las transformaciones lineales

pueden expresar la mayoría de las transformaciones útiles. En particular, transformaciones de bucles (tales como inversión de bucles, permutación...) pueden ser modeladas como un simple caso particular llamado transformaciones unimodulares (donde la matriz T_S tiene que ser cuadrada y su determinante ± 1) [8, 31]. Transformaciones complejas tales como el bloqueado (tiling) [32] pueden conseguirse utilizando también transformaciones lineales [33]. Estas transformaciones modifican el poliedro de partida creando el poliedro objetivo que contiene los mismos puntos, pero en un orden lexicográfico nuevo. Considerando un poliedro definido por el sistema de de restricciones afines $\boldsymbol{Ax + c \geq 0}$ y la función de transformación θ que lleva al índice objetivo $\boldsymbol{y = Tx}$, deducimos que el poliedro transformado puede ser definido por $\left(AT^{-1}\right)\boldsymbol{y} + \boldsymbol{c} \geq \boldsymbol{0}$.
Por ejemplo, consideremos el poliedro de la Figura 3.3(a) y la función de transformación $\theta \begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$. La transformación correspondiente es un espacio de iteración torcido, y el poliedro resultante se puede apreciar en la Figura 3.3(c).

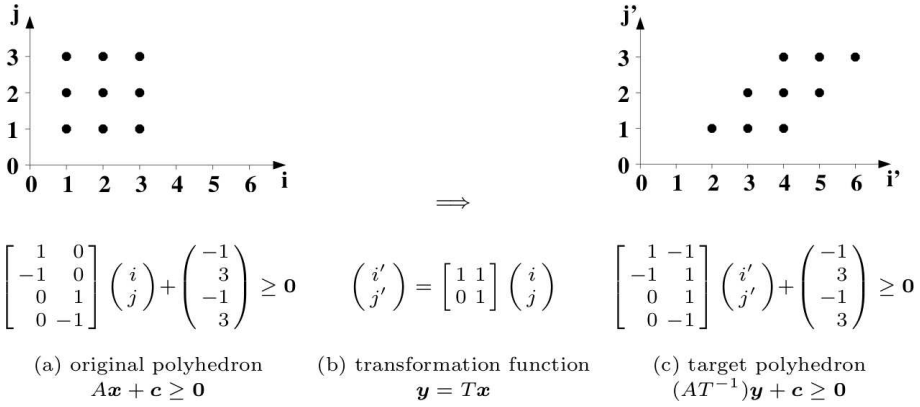


FIGURA 3.3: Transformación

Legalidad

No es posible aplicar cualquier transformación a un programa sin cambiar su semántica. Instancias del *statement* que leen y escriben en la misma posición de memoria deben hacerse en el mismo orden. Esas operaciones se dice que son dependientes entre ellas. Las relaciones de dependencia dentro de un programa deben dirigir la construcción de la transformación. Aquí veremos cómo las dependencias pueden expresarse de forma exacta usando (in)ecuaciones. Entonces mostraremos cómo construir el espacio legal de transformaciones donde cada transformación del programa debe encontrarse.

Grafo de dependencias Un modo muy conveniente de representar las restricciones de la planificación entre las operaciones es el grafo de dependencias. En este grafo dirigido, cada *statement* del programa es representado usando un vértice único, y las relaciones de dependencia existentes entre diferentes instancias del *statement* se representan utilizando aristas. Cada vértice se etiqueta con el dominio de iteración del *statement* correspondiente y las aristas con el poliedro de dependencias que describe la relación de dependencia entre el *statement* fuente y el destino. La relación de dependencia puede definirse de la siguiente manera:

Un *statement* R depende de un *statement* S (escrito SδR) si existe una operación $S(\boldsymbol{x_1})$, una operación $R(\boldsymbol{x_2})$ y una posición de memoria m tal que:

1. $S(\boldsymbol{x_1})$ y $R(\boldsymbol{x_2})$ se refieren a la misma posición de memoria m, y al menos uno de ellos escribe en esa posición.
2. $\boldsymbol{x_1}$ y $\boldsymbol{x_2}$ pertenecen respectivamente al dominio de iteración de S y de R.
3. En el orden secuencial original, $S(\boldsymbol{x_1})$ se ejecuta antes que $R(\boldsymbol{x_2})$.

De esta definición, podemos fácilmente describir el poliedro de dependencias de cada relación de dependencia entre dos *statements* con (in)ecuaciones afines. En este poliedro, cada punto entero representa una dependencia entre dos instancias de los correspondientes *statements*. Los sistemas tienen los siguientes componentes:

1. Misma posición de memoria: asumiendo que m es una posición de un array, esta restricción esla igualdad de las funciones de un par de referencias al mismo array: $F_S\boldsymbol{x}_S + \boldsymbol{a}_S = F_R\boldsymbol{x}_R + \boldsymbol{a}_R$.
2. Dominios de iteración: los dominios de iteración de S y R pueden ser descritos usando inecuaciones afines, respectivamente $A_S\boldsymbol{x}_S + \boldsymbol{c}_S \geq \boldsymbol{0}$ y $A_R\boldsymbol{x}_R + \boldsymbol{c}_R \geq \boldsymbol{0}$.
3. Orden de precedencia: esta restricción puede ser separada en una disyunción de tantas partes como bucles comunes a S y R. Cada caso corresponde a una profundidad de un bucle común y se llama nivel de dependencia. Para cada nivel de dependencia l, las restricciones de precedencia son la igualdad de los índices de bucle de las variables en una profundidad menor a l: $x_{R,i} = x_{S,i}$ para $i < l$ y $x_{R,l} > x_{X,l}$ si l es menor que el nivel de anidamiento común. De otra manera, no hay restricciones adicionales y la dependencia solo existe si S es textualmente anterior a R. Estas restricciones pueden ser escritas usando inecuaciones lineales: $P_S\boldsymbol{x}_S - P_R\boldsymbol{x}_R + \boldsymbol{b} \geq \boldsymbol{0}$.

Por lo tanto, el poliedro de dependencias para SδR en un nivel dado l y para un par de referencias p se puede describir usando el siguiente sistema de (in)ecuaciones:

$$\boldsymbol{D}_{S\delta R,l,p} : D\begin{pmatrix} \boldsymbol{x}_S \\ \boldsymbol{x}_R \end{pmatrix} + \boldsymbol{d} = \left[\begin{array}{cc} F_S & -F_R \\ \hline A_S & 0 \\ 0 & A_R \\ P_S & -P_R \end{array}\right] \begin{pmatrix} \boldsymbol{x}_S \\ \boldsymbol{x}_R \end{pmatrix} + \left(\begin{array}{c} \boldsymbol{a}_S - \boldsymbol{a}_R \\ \hline \boldsymbol{c}_S \\ \boldsymbol{c}_R \\ \boldsymbol{b} \end{array}\right) \begin{matrix} = \\ \hline \geq \end{matrix} \boldsymbol{0} \tag{3.2}$$

Hay una dependencia SδR si existe un punto entero dentro de $\boldsymbol{D}_{S\delta R,l,p}$. Esto puede ser fácilmente comprobado con algún tipo de herramienta de programación lineal entera como PipLib [14]. Si el poliedro no es vacío, existe una arista en el grafo de dependencias desde el vértice que corresponde a S hacia el que corresponde a R etiquetado con $\boldsymbol{D}_{S\delta R,l,p}$. Por simplicidad se pueden obviar los subíndices l y p y referirnos en el futuro como $\boldsymbol{D}_{S\delta R}$ como el único poliedro de dependencias que describe a SδR.

Espacio legal de transformaciones Considerando las transformaciones como funciones de planificación, el intervalo de tiempo en el programa objetivo entre las ejecuciones de dos operaciones $R(\boldsymbol{x}_R)$ y $S(\boldsymbol{x}_S)$ es

$$\Delta_{R,S}\begin{pmatrix} \boldsymbol{x}_S \\ \boldsymbol{x}_R \end{pmatrix} = \theta_R(\boldsymbol{x}_R) - \theta_S(\boldsymbol{x}_S).$$

Si existe una dependencia SδR, por ejemplo si $\boldsymbol{D}_{S\delta R}$ no es vacío, entonces $\Delta_{R,S}\begin{pmatrix}\boldsymbol{x}_S\\ \boldsymbol{x}_R\end{pmatrix}-\mathbf{1}$ debe tener una forma no negativa en $\boldsymbol{D}_{S\delta R}$ (intuitivamente, el intervalo de tiempo entre dos operaciones $R(\boldsymbol{x}_R)$ y $S(\boldsymbol{x}_S)$ tal que $R(\boldsymbol{x}_R)$ depende de $S(\boldsymbol{x}_S)$ debe ser al menos de 1, el intervalo de tiempo más pequeño. Esto garantiza que la operación $R(\boldsymbol{x}_R)$ es ejecutada antes que $S(\boldsymbol{x}_S)$ en el programa objetivo). Esta función afín puede ser expresada en términos de D y $\mathbf{d}$ aplicando el lema de Farkas [15].

Lema de Farkas Sea D un poliedro no vacío definido por las inecuaciones $A\boldsymbol{x}+\boldsymbol{b}\geq\mathbf{0}$. Entonces cualquier función afín $f(\boldsymbol{x})$ es no negativa en D sí y solo sí es una combinación afín positiva:

$$f(\boldsymbol{x})=\boldsymbol{\lambda}_0+\Lambda^T(A\boldsymbol{x}+\boldsymbol{b}),\text{ con }\boldsymbol{\lambda}_0\geq\mathbf{0}\text{ y }\Lambda^T\geq\mathbf{0},$$

donde $\boldsymbol{\lambda}_0$ y Λ^T son los multiplicadores de Farkas.

De acuerdo con este lema, para cada arista en el grafo de dependencias, podemos encontrar un vector positivo $\boldsymbol{\lambda}_0$ y una matriz Λ^T (los multiplicadores de Farkas) tales que:

$$T_R\boldsymbol{x}_R+\boldsymbol{t}_R-(T_S\boldsymbol{x}_S+\boldsymbol{t}_S)-\mathbf{1}=\boldsymbol{\lambda}_0+\Lambda^T\left(D\begin{pmatrix}\boldsymbol{x}_S\\ \boldsymbol{x}_R\end{pmatrix}+\boldsymbol{d}\right),\boldsymbol{\lambda}_0\geq\mathbf{0},\Lambda^T\geq 0. \tag{3.3}$$

Esta fórmula puede dividirse en tantas igualdades como variables independientes (componentes de $\boldsymbol{x}_S$ y $\boldsymbol{x}_R$, parametros y valor escalar) igualando sus coeficientes en ambos lados de la fórmula. Los multiplicadores de Farkas pueden eliminarse utilizando el algoritmo de proyección de Fourier-Motzkin /citeSchrijver:1986:TLI:17634 para encontrar las restricciones sobre las funciones de transformación. Los sistemas de restricciones describen el espacio legal de transformaciones, donde cada punto entero corresponde a una solución legal.

3.2. Herramientas

De momento sólo hemos presentado los fundamentos teóricos del modelo poliédrico. Realizar todo este proceso a mano constituiría una tarea muy ardua, por lo que se han ido desarrollando herramientas que hacen todo el proceso de manera automática. Hay principalmente dos grupos que han trabajado en estas herramientas, los cuales también han desarrollado también la base teórica de este modelo, uno en el INRIA [1] liderado por Paul Feautrier [15], y otro en la Universidad de Stanford, liderado por Monica Lam [31]. A continuación se presenta este conjunto de librerías y herramientas, los cuales se han unido para formar PoCC [4]. PoCC es un compilador flexible *source-to-source* iterativo, y tiene la ventaja de poder utilizar todas estas herramientas de una forma compacta, evitando el paso mediante ficheros de representaciones intermedias, asegurando así la corrección del código generado. Además, contiene todas las opciones de las herramientas que generan la nueva planificación de forma condensada, por lo que es mucho más fácil de utilizar.

El proceso sería el siguiente:

- Obtención de una representación poliédrica de un SCoP en C.
- Análisis de dependencias, generando los poliedros de dependencias.

[1] Institut National de Recherche en Informatique et en Automatique

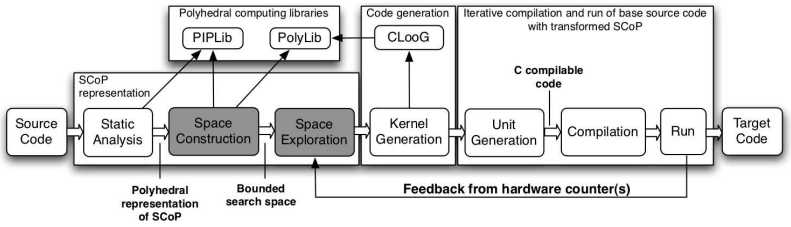


FIGURA 3.4: Proceso de transformación del código

- Generación del espacio legal de transformaciones, y búsqueda dentro del mismo para encontrar una planificación óptima en algún sentido.
- Generación de código C optimizado a partir de la planificación obtenida.

En la Figura 3.4 podemos observar el flujo de transformación.

3.2.1. Librerías para trabajar con poliedros

Todas estas herramientas necesitan representar y operar con poliedros. Para ello utilizan una serie de librerías que comento a continuación:

Polylib Polylib [25] es una librería escrita en C para la manipulación de poliedros. La librería opera con objetos del tipo vector, matriz, rejilla, poliedro, Z-poliedro, uniones de poliedros, y muchas otras estructuras intermedias.
La librería está basada en el algoritmo Chernikova, en el que se redescubrió la representación de la doble descripción que muestra la dualidad entre la representación geométrica y una lista de restricciones proporcionada como sistemas de ecuaciones lineales e inecuaciones por un lado, y la representación por un conjunto finito de vértices y lineas por otro. Después se hicieron mejoras en el proceso de conversión entre varios tipos de representaciones, y fue extendida para poder manejar poliedros paramétricos y Z-poliedros. También incluye el polinomio de Ehrhart, para contar los puntos enteros en un poliedro paramétrico

ISL ISL [27] es una librería escrita en C con hilos, que sirve para manipular conjuntos y relaciones de puntos enteros limitados por unas restricciones afines. Es una librería más moderna que Polylib, es bastante más potente, y por ejemplo se usa como librería para gestión y operación de poliedros en la herramienta CLooG.

Piplib Piplib [14, 6] es una librería que se utiliza para resolver problemas de programación lineal entera paramétricos. Encuentra el mínimo (o máximo) lexicográfico en el conjunto de puntos enteros que pertenecen a un poliedro convexo.

3.2.2. Herramientas de análisis

Clan

Clan (*Chunky Loop ANalyzer*) /citeBastoul08clana es un software y una librería que traduce ciertas partes de programas de alto nivel escritos en C, C++,C# o Java a una representación poliédrica. Esta representación puede ser manipulada por otras herramientas para, por ejemplo, conseguir complejas reestructuraciones del programa (para optimización,

paralelización, o cualquier otro tipo de manipulación).
Clan es una herramienta muy básica dado que sólo es un traductor de un programa dado en una representación, a otra distinta. Como hemos dicho antes, considera cierta parte del código, que está encapsulado por los pragmas #pragma scop y #pragma endscop.

Candl

No hay demasiada documentación sobre esta herramienta, sólo la página de desarrollo [9]. Es un software y una librería que, a partir de una representación poliédrica de un código, por ejemplo, la generada por Clan, analiza las dependencias existentes entre los *statements* y genera los poliedros de dependencias.

3.2.3. Herramientas de generación de código

CLooG

CLooG [10] es un software y una librería que encuentra un código que alcanza cada punto entero de uno o más poliedros parametrizados. Usa un algoritmo de generación de código conocido como el algoritmo de Quilleré et al., pero con mejoras y extensiones propias. El usuario tiene control total en la calidad del código generado. Por un lado, el tamaño del código generado tiene que ser optimizado en favor de la legibilidad o el uso de cache de instrucciones. Por otro lado, hay que asegurarse de que un mal manejo del control no obstaculice el rendimiento del código generado, por ejemplo produciendo instrucciones de control redundantes o límites de bucles muy complejos. Es una herramienta diseñada para evitar la sobrecarga de control y producir un código muy eficiente.

Supongamos que tenemos un conjunto de restricciones afines que describen un dominio del espacio, y queremos escanearlo. Consideremos por ejemplo el siguiente conjunto de restricciones donde i y j son desconocidas (son las dos dimensiones del espacio) y m y n son los parámetros:

```
2<=i<=n
2<=j<=m
j<=n+2-1
```

Consideremos también que tenemos un conocimiento parcial de los valores de los parámetros, llamado contexto, expresado como restricciones afines, por ejemplo:

```
m>=2
n>=2
```

En la figura 3.5 podemos ver una representación gráfica de esto, donde los puntos enteros se representan como puntos. Las restricciones afines del dominio y del contexto son la entrada para CLooG. La salida es un pseudocódigo que dice la forma de escanear esos puntos enteros del dominio de entrada de acuerdo con el contexto:

```
for (i=2;i<=n;i++)
{
    for (j=2;j<=min(m,-i+n+2);j++)
    {
        S1(i,j) ;
    }
}
```

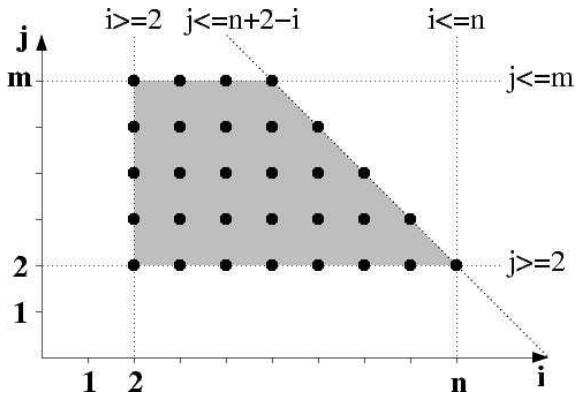


FIGURA 3.5: Representación del dominio y restricciones

3.2.4. Herramientas para la transformación de código

Pluto

Pluto [13, 7] es una herramienta de paralelización automática basada en el *polyhedral model*. Transforma programas en C para optimizarlos para paralelismo de grano grueso y localidad de datos. Esto lo hace encontrando transformaciones afines para un bloqueado y una fusión eficientes, pero no limitado por ellas. Puede generar código paralelo en OpenMP para multicores de forma automática desde código C secuencial. A pesar de ser una herramienta totalmente automática, hay que proporcionar una serie de opciones para afinar aspectos tales como tamaño de bloque, factor de desenrollado, etc.

El bloqueo es una transformación clave en la optimización para paralelismo y localidad de datos. El bloqueo para localidad requiere agrupar puntos de un espacio de iteración en bloques más pequeños (*tiles*) permitiendo reuso en múltiples direcciones cuando el bloque cabe en una memoria más rápida (registros, cache L1 o L2). Para el bloqueo para paralelismo de grano grueso hay que particionar el espacio de iteración en bloques que puedan ser ejecutados concurrentemente en diferentes procesadores con una frecuencia y un volumen de comunicaciones interprocesador reducidos. Un bloque es ejecutado de forma atómica en un procesador con comunicaciones sólo antes y después de su ejecución.

LeTSeE

LeTSeE [21] es un programa y también librería, dedicada a explorar el espacio legal de transformaciones de un programa en forma SCoP.
LeTSeE a partir de la representación poliédrica del código, construye el espacio legal de transformaciones y es capaz de hacer búsquedas de varios tipos dentro de este espacio.

En [21] explican que para planificaciones unidimensionales se puede generar todo el espacio legal de transformaciones y hacer un recorrido exhaustivo a través de él. Pero para planificaciones multidimensionales es generalmente imposible construir un conjunto de todas las planificaciones, debido a que es extremadamente difícil garantizar la completitud. Por ello LeTSeE puede hacer uso de heurísticas de diferentes tipos para realizar la búsqueda a través del espacio legal de transformaciones. La opcion walk es la que nos permite elegir si hacer una búsqueda exhaustiva (exhaust), o utilizar una heurística para la búsqueda (random, skip, m1, dh o ga).

Otra de las opciones importantes es letsee-bounds, que son los límites de los coeficientes para los iteradores, parámetros y constantes. Los valores mínimo y máximo por defecto son -1 y 1, y estos parámetros limitan el subconjunto del espacio legal de transformaciones que se generará y sobre el que se hará el recorrido.

Por lo tanto, LeTSeE no es un programa que intente encontrar el óptimo dentro del espacio legal de transformaciones, sino que hace una búsqueda por todo ese espacio (siempre que sea posible), o por una parte bien delimitada, y genera código C de todas esas posibles planificaciones. Una de las opciones que tiene es la de insertar un contador de tiempo en el código generado, permitiendo por profiling encontrar la planificación óptima en tiempo de ejecución. Pero también se pueden realizar estudios de otros tipos con todos estos códigos generados, intentando buscar el mejor por ejemplo en uso de cache.

Capítulo 4

Herramientas de Polyhedral Model para GPUs

En este capítulo se presentan una serie de herramientas basadas en el *Polyhedral Model* que tratan de generar automáticamente código paralelo eficiente para tarjetas gráficas.

4.1. Par4All

Par4All [5] es un paralelizador automático y un compilador para códigos C y Fortran. A pesar de saber que utiliza técnicas del modelo del poliedro, es muy difícil conocer su modo de funcionamiento interno, ya que es una solución comercial. Aparentemente utilizan Polylib como núcleo matemático. Las técnicas del *polyhedral model* utilizadas por Par4All parecen ser bastante poco agresivas, generalmente aplicables de un modo mejor, pero poco avanzadas. Los tamaños de bloque y el número de hilos son siempre fijos, por lo que puede afectar negativamente al rendimiento.

El punto débil de Par4All está en el manejo de la memoria, ya que no mapea ningún dato a memoria compartida. Par4All fue desarrollado como un generador de código para CPU y luego extendido para generar código GPU, por lo que esta extensión para CUDA no es muy eficiente.

4.2. Gpuloc

Esta herramienta es un *fork* de Pluto. La idea básica utilizada en esta herramienta es usar Pluto para generar una planificación y buscar una banda bloqueable de dos bucles. Entonces esta banda se bloquea y aplica una transformación para dejar las dependencias entre iteraciones sólo de los bucles más externos, para asegurarse de que la parte interna (el bucle interno que divide entre bloques, y el bucle que va recorriendo cada bloque interno) sea paralela. Estos dos bucles paralelos entonces se asignan a bloques e hilos CUDA. El usuario es el que tiene que proporcionar los tamaños de bloque. Todos los bucles más externos, y hasta el primer bucle de bloqueo forman la planificación del *host*, el segundo bucle de bloqueo se asigna a bloques CUDA, el primer bucle que recorre internamente los bloques y los bucles que le siguen forman la planificación de GPU, y el bucle más interno que recorre los bloques se asigna a hilos. El bucle más externo que va recorriendo los bloques internamente

se ejecuta de forma síncrona por todos los hilos, esto es, que los hilos necesitan una barrera de sincronización después de cada iteración. El único uso que hacen de diferentes niveles de la jerarquía de memoria de la tarjeta es para aprovechar la memoria compartida. Ellos dividen la cantidad de memoria compartida por bloque por dos, y usan estas dos regiones como *buffers* para cargar datos de iteraciones consecutivas.

4.3. Pluto para CUDA

Existe una versión de Pluto, (basada en la revisión 0.6.2) que genera código CUDA. Esta versión primeramente utiliza Pluto para generar una planificación. Entonces bloquea la banda más externa basándose en un tamaño de bloque especificado por el usuario. La implementación no genera ningún código para llamar al kernel que se genera, sino que se lo deja al usuario, por lo que partes de una implementación para GPU aparte de la llamada de los kernels se quedan fuera, como las copias de memoria del *host* al dispositivo, y el retorno de los resultados de la tarjeta otra vez a memoria principal, o el tamaño y número de dimensiones de rejillas y bloques.
La planificación para GPU es el resultado de mapear los dos bucles bloqueados más externos a los bloques, y los dos bucles más externos ya dentro del bloqueado sobre los hilos. El número de bloques e hilos por bloque necesarios tienen que especificarse en el archivo *decls.h*. Este mapeo se realiza sobre el código generado por CLooG. Esta tarea es delicada, ya que podría asumirse que sólo un bucle es generado por cada dimensión. Los bucles son primeramente mapeados sobre la dirección y, y entonces sobre la dirección x. Esto es importante para el código añadido que transfiere datos de memoria global a compartida, que está construida aparentemente para acceder a elementos sucesivos de la memoria global en sucesivas iteraciones del bucles más interno (haciendo un uso eficiente en el patrón de acceso a la memoria global). Para esta copia se asume que la parte de un array accedido por un bloque es rectangular.

También implementan una forma de sincronizar bloques. Cuando se necesita sincronización entre bloques, se asume que todos los bloques pueden ejecutarse en paralelo al mismo tiempo, y la memoria global es utilizada para la comunicación entre bloques. Esto es necesario para tareas que generalmente se dividirían entre varios kernels, ya que Pluto para CUDA sólo es capaz de generar un único kernel con todo el código transformado.

4.4. Polyhedral Parallel Code Generator(PPCG)

Esta herramienta en sus inicios estaba basada en Pluto, pero ahora es una herramienta independiente. PPCG [28] utiliza la librería ISL como núcleo matemático, con la que también implementa las estructuras y métodos necesarios para modelar el planificador. El código generado es similar al de Pluto en su versión compatible con CUDA, pero PPCG genera código para *host* y *device*. Por esto, tiene la ventaja de poder hacer más de una llamada a un kernel, y también utilizar varios kernels. Como en la versión de Pluto compatible con CUDA, PPCG bloquea la banda bloqueable más externa, pero realiza tres bloqueos en vez de dos, lo que resulta en un código más robusto. Por ahora PPCG necesita que el usuario le proporcione los valores para diferentes bloqueos. Se necesitan estos tres ficheros:

- tile.sizes: define los tamaños de bloque que se van a aplicar a la banda más externa bloqueable. Es el primer nivel de bloqueo y también es utilizado para definir el bloque de memoria compartida.

- grid.sizes: define el tamaño de Grid CUDA. PPCG utiliza este fichero para mapear hasta dos bucles bloqueados paralelos sobre los bloques CUDA.
- block.sizes: define el tamaño de bloque CUDA. PPCG utiliza este fichero para mapear hasta tres bucles internos paralelos sobre los hilos.

Además, PPCG como mejora a Pluto en su versión compatible con CUDA, es capaz de elegir el mejor nivel de memoria para asignar cada dato. Por el momento sólo tiene en cuenta tres niveles de memoria: registros, memoria compartida y memoria global. PPCG hace un análisis de reuso de datos para hacer esta decisión, y entonces los datos son asignados siguiendo estos criterios:

- Si un elemento sólo lo usa un hilo, entonces el dato se asigna a registros.
- Si un elemento lo utiliza un solo hilo, pero el patrón de acceso no es fusionado, entonces los datos se asignan en memoria compartida.
- Si un elemento es utilizado por varios hilos en un bloque el dato se asigna en memoria compartida.
- En otro caso se utiliza la memoria global.

Capítulo 5

Benchmarks utilizados

En este capítulo se va a hacer una breve descripción de los benchmarks utilizados para estudiar la eficiencia del uso de las herramientas de paralelización automática de código.
Estos benchmarks han sido seleccionados de una colección de benchmarks, llamada PolyBench [20]. PolyBench es una colección de benchmarks que contienen secciones *SCoP*. Su propósito es el de uniformizar la ejecución y el monitorizado de kernels. Por lo tanto es una colección de benchmarks perfecta para nuestro propósito, ya que los *#pragma scop* y *#pragma endscop* ya están colocados en código, delimitando claramente los kernels que vamos a querer optimizar. Además se ha añadido un benchmark un poco más complejo, la factorización no negativa de matrices (NMF), ya que la mayoría de los benchmarks de PolyBench se pueden mapear en un sólo kernel, y éste es un poco más complejo en el sentido de que hay una mayor cantidad de bucles, se pueden hacer un mayor número de transformaciones distintas, seguramente se necesite más de un kernel para mapear el código a GPU, y es interesante ver qué pueden hacer las herramientas de generación automática de código con él.

5.1. Multiplicación de matrices

La multiplicación de matrices es un código que ha sido muy estudiado, debido a su uso directo muy frecuente para cálculos científicos. Se ha dado mucha importancia a esta operación en GPUs, debido al alto potencial de mejora de rendimiento que se puede obtener. Mismamente, en el manual del programador CUDA que proporciona Nvidia [23], se muestra como ejemplo para uso de diferentes niveles de la jerarquía de memoria.
En CUDA la multiplicación de matrices que se toma como referencia, por ser la más eficiente de todas, es la versión de *gemm* de la librería CUBLAS [1]. En las primeras versiones de CUBLAS se utilizaba una forma optimizada de una multiplicación de matrices bloqueada, utilizando memoria compartida. Vasily Volkov creó un nuevo código mucho más eficiente [29], que fue incluido a partir de la versión 2 de la librería.
La versión original a partir de la cual vamos a realizar las transformaciones es la siguiente:

```
#pragma scop
for( i=0; i < N; i++ )
{
   for( j=0; j < M; j++ )
   {
      C[i][j] = 0;
      for( k=0; k < K; k++ )
      {
         C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

```
      }
   }
}
#pragma endscop
```

Se puede observar el código rodeado de los *#pragma scop*, por lo que el código está preparado para poder utilizar las herramientas sobre él.
Comentar que también se ha transformado el código de entrada para las herramientas, por ver si cambia de alguna forma el código generado. Intercambiando los bucles j y k, o los bucles i y j, el resultado final es el mismo, por lo que no hace falta comentar estas versiones por separado, ya que los mapeos que se harán por las herramientas automáticas serán iguales.

5.2. Trasposición de matrices

La trasposición de matrices puede parecer un kernel trivial, muy simple de implementar, pero también hay bastantes estudios sobre este algoritmo mapeado en tarjetas gráficas. El interés se centra en que una trasposición es básicamente una copia sencilla de un grupo de posiciones de memoria a otra, pero al cambiar el *layout* de filas a columnas, el rendimiento baja de forma considerable.
El código que se va a utilizar como referencia, y para pasar a los programas es el siguiente:

```
#pragma scop
   for (i = 0; i < N; i++)
   {
      for (j = 0; j < M; j++)
      {
         B[j][i] = A[i][j];
      }
   }
#pragma endscop
```

5.3. Jacobi

Estos algoritmos tienen la misma base, son similares a una convolución, en el primer caso con un vector, y en el segundo con una matriz.

Jacobi 1D

Este código es bastante simple, y similar a una convolución. Va iterando sobre el vector, cogiendo el elemento actual, el anterior y el posterior, y haciendo la media, dividiéndolo entre tres. Veamos el código:

```
#pragma scop
for (t = 0; t < T; t++)
{
   for (i = 2; i < N - 1; i++)
   {
      b[i] = 0.33333 * (a[i-1] + a[i] + a[i + 1]);
   }
   for (j = 2; j < N - 1; j++)
   {
      a[j] = b[j];
   }
}
#pragma endscop
```

El resultado lo va almacenando en un vector auxíliar, que vuelve a volcar los nuevos resultados en el vector principal después de iterar por todo el vector. Hay un bucle externo que

nos dice que esto se realiza T veces.
Podemos observar a simple vista que iteraciones consecutivas de la generación de b acceden a varias posiciones del vector b idénticas, por lo que el potencial de reuso es evidente.

Jacobi 2D

Este código es exactamente igual que el anterior, pero aplicado a una matriz. Veámoslo:

```
#pragma scop
for (t = 0; t < T; t++)
{
   for (i = 2; i < N - 1; i++)
   {
      for (j = 2; j < N - 1; j++)
      {
         B[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][1+j] + A[1+i][j] + A[i-1][j]);
      }
   }
   for (i = 2; i < N-1; i++)
   {
      for (j = 2; j < N-1; j++)
      {
         A[i][j] = B[i][j];
      }
   }
}
#pragma endscop
```

Tenemos una matriz que se va recorriendo, y se va generando un nuevo valor con la media del valor de la posición para esa iteración, y el elemento superior, inferior, derecho e izquierdo. Posteriormente, cuando se ha generado la nueva matriz, se traspasan los nuevos valores a la matriz original. Esta operación se realiza T veces debido al bucle externo que contiene al resto.
Aquí también podemos observar que en la generación de elementos cercanos de B se acceden a varias posiciones de A idénticas, por lo que aquí también es evidente que puede haber un reuso de datos.

5.4. LU

Este algoritmo es un poco peculiar, ya que no es tan sencillo ver el reuso de memoria que se puede hacer, pero es bastante optimizable. Veamos el código:

```
#pragma scop
for (k=0; k<N; k++)
{
   for (j=k+1; j<N; j++)
   {
      a[k][j] = a[k][j]/a[k][k];
   }
   for(i=k+1; i<N; i++)
   {
      for (j=k+1; j<N; j++)
      {
         a[i][j] = a[i][j] - a[i][k]*a[k][j];
      }
   }
}
#pragma endscop
```

En este código primeramente se va iterando por filas, rellenando la parte de la derecha de la diagonal principal con ese mismo dato dividido entre el dato de esa fila que corresponde a la diagonal. Una vez rellenada esa fila, se rellena el rectángulo de la matriz que tiene como

elemento más a la izquierda y más arriba el elemento de la diagonal principal siguiente al que se ha utilizado para el cálculo de la fila, y todos los elementos que estén por debajo y por la derecha de este hasta el final de la matriz, utilizando la fila que se ha rellenado antes, y la columna que corresponde a la anterior al comienzo del rectángulo que se va a rellenar. Por lo tanto podríamos aprovechar reuso al generar primero los datos de una fila, que luego se van a utilizar además para otros cálculos.

5.5. FDTD-2D

En este código se hacen cálculos sobre tres matrices, dos son simplemente auxiliares, y una tercera donde se va guardando el resultado final.

```
#pragma scop
for(t=0; t<tmax; t++)
{
   for (j=0; j<ny; j++)
      ey[0][j] = t;
   for (i=1; i<nx; i++)
      for (j=0; j<ny; j++)
         ey[i][j] = ey[i][j] - 0.5*(hz[i][j]-hz[i-1][j]);
   for (i=0; i<nx; i++)
      for (j=1; j<ny; j++)
         ex[i][j] = ex[i][j] - 0.5*(hz[i][j]-hz[i][j-1]);
   for (i=0; i<nx; i++)
      for (j=0; j<ny; j++)
         hz[i][j]=hz[i][j]-0.7*(ex[i][j+1]-ex[i][j]+ey[i+1][j]-ey[i][j]);
}
#pragma endscop
```

Primeramente se rellena la primera fila de una de las matrices auxiliares con el valor de la iteración en la que nos encontramos. Las dos matrices auxiliares se van actualizando con datos de la principal, restando el resultado del valor del dato de la matriz principal menos el valor anterior, en un caso en el eje horizontal, y en el otro en el vertical, al valor de la matriz auxiliar.
Una vez hecho esto se actualiza el valor de la posición actual de la matriz principal, restando a su valor anterior una operación realizada con valores de las matrices auxiliares.
En los tres últimos bucles podemos observar que para la generación de elementos consecutivos, se utiliza parte de los mismos datos, por lo que se puede aprovechar esto para el reuso de los mismos.

5.6. Seidel

Este código ha sido elegido porque las herramientas automáticas de generación de código para GPU no funcionan con él. Echemos primeramente un vistazo al código utilizado para las transformaciones:

```
#pragma scop
for (t=0; t<=T-1; t++)
{
   for (i=1; i<=N-2; i++)
   {
      for (j=1; j<=N-2; j++)
      {
         a[i][j] = (a[i-1][j-1] + a[i-1][j] + a[i-1][j+1]
         + a[i][j-1] + a[i][j] + a[i][j+1]
         + a[i+1][j-1] + a[i+1][j] + a[i+1][j+1])/9.0;
      }
   }
}
#pragma endscop
```

Básicamente se realizan T iteraciones sobre un doble bucle que va recorriendo la matriz a, donde cada elemento de la matriz pasa a ser el valor medio del valor que tenía antes el elemento en esa posición, y los elementos de su alrededor, ya que divide por nueve la suma de nueve elementos. Aquí nos encontramos con un problema, y es que si intentáramos paralelizar este algoritmo, nos encontraríamos con que vamos a necesitar los valores de la matriz actualizados en un orden secuencial, es decir, no podemos actualizar el valor de la matriz $(i, j+1)$ sin haber actualizado antes el valor (i, j). Se podría intentar utilizar una estructura auxiliar con los valores actualizados, y un booleano por posición que indicara si esa posición ya ha sido actualizada o no, para utilizar el valor actualizado de la estructura auxiliar, o el valor antiguo de la matriz original. Esto sigue siendo bastante ineficiente.

Si utilizamos Pluto para C, obtenemos un código menos paralelo aún, ya que el bucle externo donde antes se utilizaban T iteraciones, ahora itera sobre un mayor rango. Encuentra un único bucle paralelo, pero con un rango demasiado pequeño como para obtener alguna ventaja mapeándolo en GPU.
Con PPCG no se llega a obtener ninguna solución. Obviamente esto es normal, ya que es un algoritmo puramente secuencial.

5.7. Non-Negative Matrix Factorization (NMF)

Como hemos dicho al introducir el capítulo, este código es algo más complejo, no en las operaciones realizadas, ya que son básicamente multiplicaciones de matrices, trasposiciones, y operaciones punto a punto sobre las mismas, sino en el mayor número de las mismas dentro de una sección SCoP, por lo que se pueden realizar un mayor número de transformaciones distintas sobre ellas. Hay varias versiones del algoritmo, pero nos vamos a centrar en la versión Gaussiana del mismo, o GNMF [18, 17]. El código utilizado ha sido el siguiente:

```
#pragma scop
for( i=0; i < K; i++ )
   for( j=0; j < M; j++ )
   {
      WtV[i][j] = 0;
      for( p=0; p < N; p++ )
         WtV[i][j] = WtV[i][j] + W[p][i]*V[p][j];
   }
for( i=0; i < K; i++ )
   for( j=0; j < K; j++ )
   {
      WtW[i][j] = 0;
      for( p=0; p < N; p++ )
         WtW[i][j] = WtW[i][j] + W[p][i]*W[p][j];
   }
for( i=0; i < K; i++ )
   for( j=0; j < M; j++ )
   {
      WtWH[i][j] = 0;
      for( p=0; p < K; p++ )
         WtWH[i][j] = WtWH[i][j] + WtW[i][p]*H[p][j];
   }
for( i=0; i < K; i++ )
   for( j=0; j < M; j++ )
      H[i][j] = H[i][j] * WtV[i][j] / WtWH[i][j];
for( i=0; i < N; i++ )
   for( j=0; j < K; j++ )
   {
      VHt[i][j] = 0;
      for( p=0; p < M; p++ )
         VHt[i][j] = VHt[i][j] + V[i][p]*H[j][p];
   }
for( i=0; i < K; i++ )
   for( j=0; j < K; j++ )
```

```
   {
      HHt[i][j] = 0;
      for( p=0; p < M; p++ )
         HHt[i][j] = HHt[i][j] + H[i][p]*H[j][p];
   }
for( i=0; i < N; i++ )
   for( j=0; j < K; j++ )
   {
      WHHt[i][j] = 0;
      for( p=0; p < K; p++ )
         WHHt[i][j] = WHHt[i][j] + W[i][p]*HHt[p][j];
   }
for( i=0; i < N; i++ )
   for( j=0; j < K; j++ )
      W[i][j] = W[i][j] * VHt[i][j] / WHHt[i][j];
#pragma endscop
```

Se puede observar lo dicho anteriormente, que las operaciones básicas son la multiplicación de matrices, y las operaciones punto a punto entre ellas. Pero por ejemplo, para la generación de WtWH se necesita la matriz WtW, al igual que para la generación de WHHt se necesita HHt, por lo que son multiplicaciones de matrices encadenadas, con la posibilidad de acercar la generación del dato de la primera matriz a su uso en la generación de la segunda.

Capítulo 6

Mapeo de Benchmarks y Resultados

En este capítulo se describe el estudio hecho para cada benchmark. Se hará un mapeo manual de cada uno para GPU, otro mapeo basado en la observación de la planificación generada por Pluto, y el código que genera para C, guiando una generación manual de código, y finalmente se comparará con el código generado automáticamente por PPCG. Se han presentado más herramientas que generan código de forma automática para GPU en el capítulo 4, pero para el análisis del mapeo automático sólo vamos a comentar el producido por PPCG, ya que se han probado el resto de herramientas, y producen unos resultados mucho más pobres e ineficientes, con muchas más limitaciones en la generación.
El propósito de la generación de todas estas versiones de código es el de descubrir los puntos débiles que pudieran tener las herramientas de generación automática de código optimizado para GPU, ya que el uso del *polyhedral model* tiene la finalidad de pasar a una representación más fácil de manejar y transformar, no genera automáticamente un código óptimo para una plataforma, simplemente puede permitir mejorar el uso de la jerarquía de memoria, o explotar de un mejor modo el paralelismo existente en un código. Para generar un código eficiente para una tarjeta gráfica se necesita precisamente esto, que se aproveche la jerarquía de memoria al máximo y que se explote al máximo el paralelismo existente en el código, haciendo una distribución de hilos y bloques lo mejor posible.
Pero no siempre se puede generar un código que aproveche ambas cosas, por lo que hay que decidir qué transformación realizar. Veamos en las siguientes secciones cómo estas herramientas se comportan para los diferentes benchmarks, y qué se podría mejorar.

6.1. Multiplicación de matrices

Mapeo manual

Naive

El primer mapeo, y el más sencillo que se puede realizar, es simplemente utilizar un hilo por elemento de la matriz destino que se quiere calcular. Tendríamos por ejemplo, A y B que serían las matrices que se multiplican de un tamaño de $N \times K$ y $K \times M$, por lo que la matriz resultado C sería de tamaño $N \times M$. En el código original en C tenemos que los bucles más externos son los que corresponden con estas dimensiones N y M, por lo

que son los que se paralelizan, creando una rejilla de $N \times M$, por comodidad con bloques bidimensionales de, normalmente 16×16 hilos. El trabajo que realizaría aquí un hilo, como se puede observar en la Figura 6.1, sería el de iterar a lo largo de toda la fila de A y la columna de B correspondiente al hilo, multiplicando los elementos y sumándolos, generando así un sólo elemento de la matriz C.
Con este mapeo sólo podemos hacer uso de la memoria global, ya que no se explota el reuso

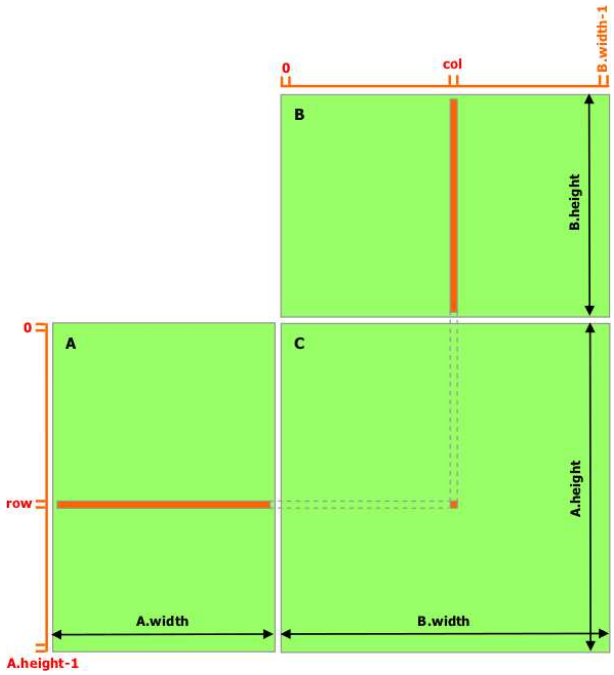


FIGURA 6.1: Multiplicación de matrices sin memoria compartida

de los datos. De hecho, cada columna de la matriz B será leída por hilos distintos hasta un total de N veces, lo que hace finalmente que el rendimiento sea muy pobre.

Memoria compartida

Intentando mejorar la eficiencia de la multiplicación, lo más sencillo sería intentar aprovechar algo mejor la jerarquía de memoria que nos ofrece la tarjeta gráfica.
En esta segunda versión, como se puede observar en la Figura 6.2, cada bloque de hilos trae progresivamente los datos de las filas y columnas. La configuración de bloques y rejilla es igual a la versión anterior. Pero en este kernel, primeramente los hilos colaboran en traer dos bloques de 16×16 datos de las matrices A y B a memoria compartida, y luego se va calculando un resultado parcial, iterando para generar el cálculo final. Por lo tanto se está haciendo un mejor aprovechamiento de la jerarquía de memoria, al poder introducir los datos que se van a multiplicar en una memoria de más rápido acceso.

Versión óptima (Volkov)

Las primeras versiones de *gemm* implementadas por CUDA CUBLAS fueron sustituidas por un algoritmo creado por Vasily Volkov [30] que obtuvo una mejora de rendimiento del 60 %.

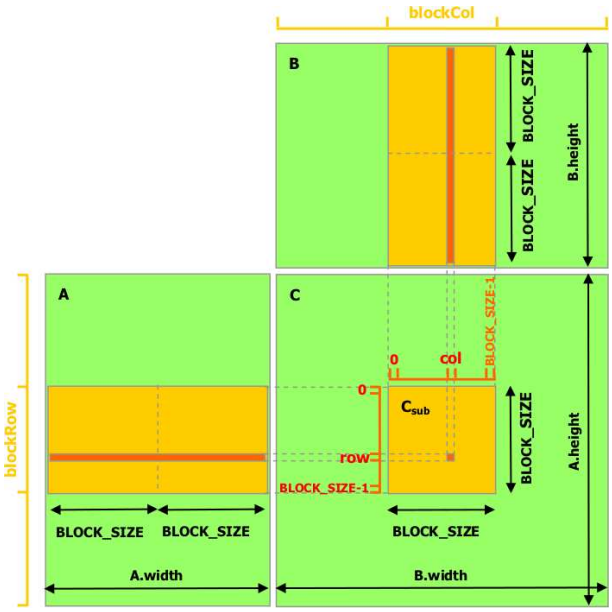


FIGURA 6.2: Multiplicación de matrices con memoria compartida

Primero reservan un vector de una longitud mínima de 64 posiciones, que es el tamaño mínimo que permite que se alcance la máxima eficiencia en las operaciones de punto flotante. En el algoritmo original orientan los vectores a lo largo de las columnas de la matriz resultado C para conseguir un acceso alineado para la búsqueda y guardado posterior de los bloques de C. De forma similar, los bloques de la matriz B son de un tamaño 16×16, ya que esto permite cargas alineadas para bloques en B y B traspuesta (ya que este algoritmo es de multipicación de matrices general (*gemm*)). Esto deja tres alternativas para los bloques de C: 16×16, 32×16 o 64×16, no se necesitan bloques más grandes. Estas decisiones requieren que los bloques de la matriz B estén en memoria compartida. Se prefiere un *layout* por filas ya que permite acceso alineado a la memoria compartida en el bucle interno de la multiplicación de matrices que realiza la actualización del bloque de la matriz resultado C. Este *layout* necesita que se traspongan los bloques de B en el caso de la multiplicación $A \times B$ y un *padding* en la memoria compartida para evitar los conflictos de banco. Esta trasposición es transparente ya que se realiza al pasar el bloque de B en memoria global a memoria compartida. Los hilos se crean en una disposición bidimensional (rejilla y bloques de dos dimensiones).
La estructura del algoritmo tal y como la presenta Volkov se puede observar en la Figura 6.3.

Las principales ventajas de esta implementación con respecto al resto son que aprovecha de forma mucho más eficiente la jerarquía de memoria de la tarjeta, aprovechando al máximo el uso de registros, cuyo tiempo de acceso es menor que el de la memoria compartida, y utilizando éstos de tal manera que se pueda solapar la latencia de acceso a memoria con las operaciones.

Mapeo semi-automático

Para el mapeo semi-automático vamos a utilizar la herramienta Pluto para C. En este caso la opción de *fuse* o fusionado de bucles no genera nada distinto, ya que no tenemos va-

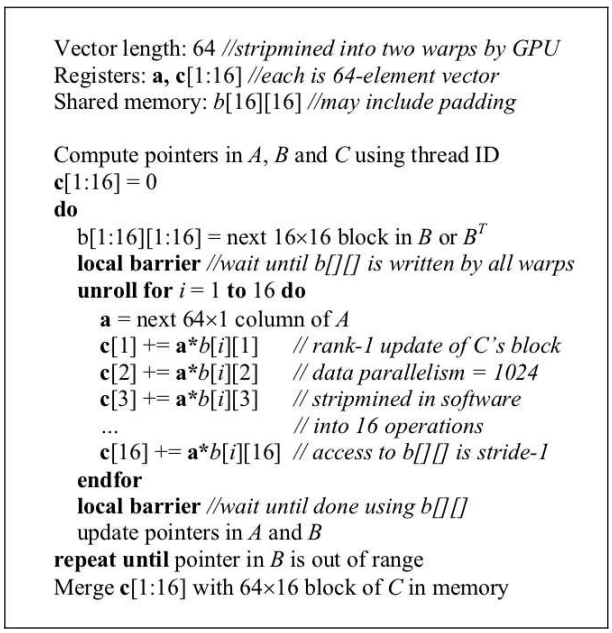


FIGURA 6.3: Estructura del algoritmo de multiplicación de Volkov

rios bucles a fusionar. Como comentario, viendo el código original, el único cambio realizado pasando esta opción a Pluto es que la inicialización de los valores de la matriz resultado C pasan de estar dentro de los bucles originales, creando unos bucles donde se inicializa de forma completa a 0 antes de empezar a multiplicar.

Aplicando la opción de *tiling* ya obtenemos algo un poco distinto. Básicamente aplica una subdivisión en bloques a la multiplicación, por lo que pasamos de tener un bucle for por cada dimensión de las matrices (N, M y K), a tener dos, uno externo en el que se divide la matriz en bloques, y otro interno en el que se recorre internamente cada uno de los bloques. Esto en código C podría proporcionarnos una ventaja al tener mayor localidad espacial en los datos, debido a usar bloques más pequeños para multiplicarlos parcialmente. Un código CUDA podría decirse que ya tiene una parte bloqueada por defecto, debido a la división de la rejilla o grid en bloques de hilos. De este modo, el equivalente al mapeo manual directo del código original a CUDA consistiría en un *tiling* de los dos bucles más externos (en este caso i y j), pero que se realiza de forma transparente. Por lo que en el caso del tiling que realiza Pluto, básicamente recorreríamos el bucle más interno (en este caso k) en trozos. Hacer esto sin aprovechar la jerarquía de memoria de la tarjeta es inútil, ya que al tener el *tiling* de los dos bucles más externos de forma transparente, estarían los bucles internos juntos, resultando en un código equivalente a tener un recorrido lineal completo de k. Ahora bien, aprovechando la jerarquía de memoria, al poder dividir la generación de un elemento de la matriz resultado C en varias partes, podemos insertar bloques de las matrices que se multiplican en la memoria compartida para mejorar el rendimiento. Precisamente esto es lo que se hace en el mapeo manual que utiliza la memoria compartida. Intentar dividir los dos bucles más internos (en este caso de k) para que se calculen por hilo sólo parcialmente elementos de la matriz resultado (es decir, mapear por hilo sólo el bucle k interno) no sirve, ya que habría hilos que intentarían escribir a la vez en la misma posición de memoria, y dado el sistema de coherencia relajado que mantiene la tarjeta gráfica, sobreescribiría datos

dando un resultado incorrecto.

Pluto también nos da la opción de *loop unrolling*, que en este caso podría ser equivalente al *#pragma unroll* de CUDA.

Mapeo automático

Utilizando la herramienta PPCG introduciendo unos tamaños de tiling, bloques y grid iguales a los usados en la versión manual para tarjeta gráfica, se puede observar que el código generado es muy parecido a la versión manual con memoria compartida. Crea dos bloques de memoria compartida de tamaño 16×16 para albergar trozos de las matrices A y B, los cuales se rellenan de forma cooperativa en cada bloque por los hilos que lo forman, y se va iterando generando resultados parciales que al final de la ejecución del hilo forman un elemento de la matriz resultado. En este caso, para almacenar el resultado parcial utiliza también un registro, que aparece de forma de matriz de tamaño 1×1. Por lo tanto la versión generada por PPCG es tan buena como la mejor versión generada a mano, aparentemente, ya que han aparecido nuevos algoritmos aún mejores, uno de los cuales veremos a continuación.

Resultados

En este benchmark era claro que era muy difícil mejorar la versión de Volkov. La diferencia de tiempo entre esta versión, y una versión bloqueada que utiliza memoria compartida para iterar por las matrices que se van multiplicando, es abismal.
Hay que comentar que la implementación de Volkov no es nada trivial, y es mucho más eficiente porque aprovecha todos los niveles de la jerarquía de memoria de la tarjeta de una forma muy superior.

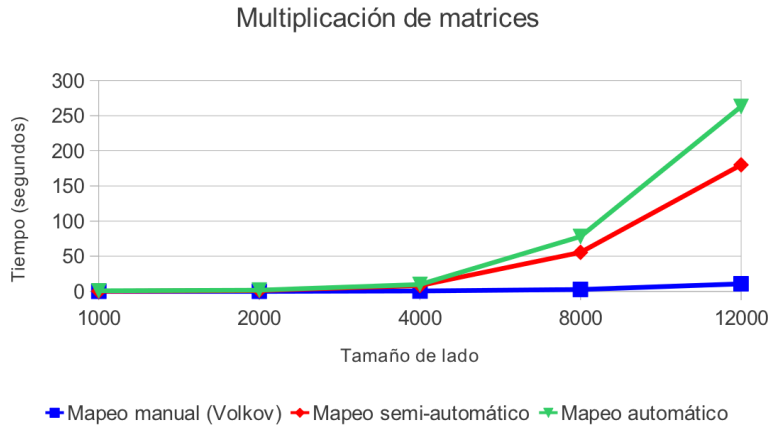


FIGURA 6.4: Comparación de mapeos en multiplicación de matrices

En este caso PPCG es posible que en un futuro pudiera generar código más similar al de Volkov, ya que PPCG es capaz de manejar todos los espacios de memoria que se utilizan en esta versión. De todas formas, mientras que sea el programador el que tenga que proporcionar los datos de tile, bloque y tamaño de rejilla, es imposible generar una versión mejor, ya que el *tiling* que se utiliza en Volkov hace que se mapeen de diferente manera las matrices que se van a multiplicar, y de momento con PPCG, los tamaños de tile son iguales para las dos.

PPCG genera un código bastante similar en rendimiento a un código manual generado con la ayuda de Pluto para C, es decir, una ayuda sobre la que comenzar la implementación, sobre como particionar la rejilla en bloques, etc. La versión automática sigue siendo algo peor ya que introduce más condiciones innecesarias dentro del kernel, e introduce escrituras intermedias a registros.

6.2. Trasposición de matrices

Mapeo manual

Naive

Una primera versión muy directa del código de trasposición de matrices para GPUs sería haciendo que cada hilo copiara un elemento de la matriz origen a la matriz destino. Esto se puede hacer utilizando memoria global, ya que no hay reuso de los datos. La disposición de hilos sería bidimensional, creando una rejilla de $N \times M$ hilos, usando bloques de, por ejemplo, 16×16 hilos, donde N y M son las dimensiones de la matriz. Es un kernel por lo tanto sumamente sencillo de implementar, pero no es muy eficiente.

Memoria compartida

Dado que la memoria global de la tarjeta gráfica tiene una latencia mucho mayor y un ancho de banda más bajo que la memoria compartida, hay que prestar especial atención sobre cómo se realizan los accesos a memoria global, en nuestro caso cargando el dato desde la posición **idata** y guardándolo en la posición **odata**. Todos los accesos a memoria global hechos por un semiwarp de hilos pueden fusionarse en una o dos transacciones si se cumplen ciertas condiciones. Estos criterios dependen de la capacidad de computación de la GPU. En nuestro caso, hemos puesto el límite inferior en 1,3, por lo que los requerimientos para la fusión son relajados. Esta fusión por lo tanto puede ocurrir cuando los datos están en segmentos alineados en grupos de 32, 64 o 128 bytes, sin importar del patrón de acceso hecho por los hilos en el segmento.
Para el mapeo naive, las cargas de **idata** se fusionan. Por cada grupo de hilos cada semiwarp lee 16 palabras de 32 bits. Utilizando *cudaMalloc()* y utilizando un tamaño de bloque múltiplo de 16 nos aseguramos el alineamiento en el segmento de memoria, por lo tanto todas las cargas se fusionan.

Este comportamiento de fusión difiere escribiendo el dato de salida **odata**. En el mapeo naive , para cada grupo de hilos, un semiwarp escribe datos de una columna en diferentes segmentos de memoria, resultando en transacciones de memoria separadas.

La forma de evitar accesos no fusionados a memoria global es introducir los datos a leer en memoria compartida, introduciendo cada semiwarp los datos de forma no contigua en esta memoria, para que al escribir de memoria compartida a memoria global se haga de forma contigua. No hay una penalización por acceder a datos no contiguos en memoria compartida, y lo único que se necesita es una barrera de sincronización *__syncthreads()* para asegurarnos de que todos los datos se encuentran en memoria compartida antes de empezar a escribir a memoria global.
Por lo tanto tenemos que cada bloque de hilos de, por ejemplo, 16×16 elementos utilizará un tamaño de memoria compartida de 16×16 datos en punto flotante, que se escribirán

leyendo desde memoria global en una escritura fusionada, y posteriormente, tras el *__syncthreads()*, se escribirá desde posiciones no contiguas de memoria compartida a memoria global, de forma también fusionada.

La memoria compartida está dividida en 16 módulos de igual tamaño, llamados bancos, que están organizados de tal manera que palabras sucesivas de 32 bits son asignadas a bancos sucesivos. Estos bancos pueden ser accedidos de forma simultánea, y para alcanzar el máximo ancho de banda de y hacia la memoria compartida, los hilos en un semiwarp deben acceder a memoria compartida asociada a bancos distintos. La excepción a esta regla es cuando todos los hilos en un semiwarp leen la misma dirección en memoria compartida, lo que resulta en un *broadcast* donde el dato que corresponde a esa dirección es enviado a todos los hilos en una transacción. Utilizando un array de floats de 16×16 de memoria compartida, tenemos una columna por banco, pero tenemos que acceder por columnas, lo que hace que varios hilos accedan al mismo banco. Utilizando un tamaño 16×17 no solo cada elemento de una fila estará en un banco diferente, sino que cada elemento de una columna también, lo que hará que se evite el conflicto.

En definitiva, el hecho de tener que pasar de orden de filas a orden de columnas nos ha obligado a utilizar memoria compartida para realizar los accesos fusionados y no perder en eficiencia, y añadir un *padding* para evitar tener conflictos de banco. Con ello tendríamos algo muy similar a una copia simple de matriz a matriz, salvo por un pequeño detalle, que veremos en el siguiente apartado.

Trasposición diagonal

Para usar la memoria global de forma efectiva, los accesos concurrentes a memoria global por todos los warps activos deberían estar divididos entre particiones. Hay casos en los que los accesos a la memoria global están dirigidos a un subconjunto de particiones, dejando otras sin utilizar, y causando colas de peticiones al subconjunto donde van todos los accesos. Esto es similar a la fusión de accesos de memoria global a nivel de semiwarp, pero aplicado a los accesos a la memoria global por todos los warps activos.

Cuando se lanza un kernel, el orden en el que los bloques son asignados a los multiprocesadores está determinado por el ID de bloque, definido normalmente como:

```
id = blockIdx.x + gridDim.x*blockIdx.y;
```

que es un orden por filas de los bloques en la rejilla. Una vez que se alcanza la máxima ocupación en la tarjeta, los bloques adicionales son asignados a multiprocesadores tal y como se vayan necesitando. La rapidez y el orden en que los bloques vayan completando su ejecución no es algo determinado, por lo que bloques activos inicialmente contiguos pueden estar más alejados mientras la ejecución del kernel va progresando.
En la Figura 6.5 podemos ver la forma en la que se van leyendo los datos de la matriz de entrada, donde los bloques que se ejecutan de forma concurrente acceden de una forma más o menos distribuida entre particiones de memoria, lo contrario que ocurre con la escritura de los datos de salida, donde se accede a muy pocas particiones de la memoria.

Para evitar esto, mientras que el programador no tiene control directo sobre el orden en el que los bloques se planifican, lo que es determinado por el valor de la variable **blockIdx**, existe una cierta flexibilidad sobre cómo interpretar los componentes de **blockIdx**. Normalmente se asume que los componentes de esta variable como un sistema de coordenadas

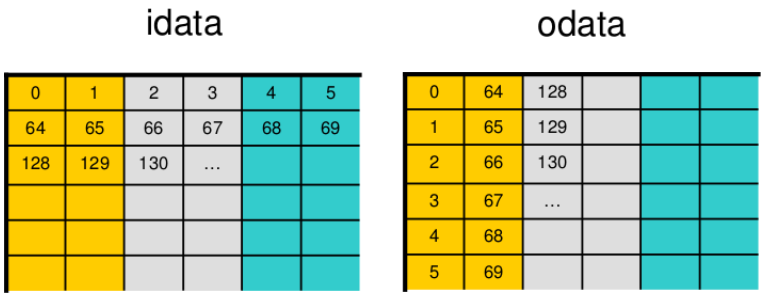


FIGURA 6.5: Reparto de los datos en la memoria en hilos en ejecución

cartesianas. Pero no es la única forma de hacer esto.
Una forma de evitar el problema señalado más arriba es usar una interpretación diagonal de los componentes de **blockIdx**; la componente **y** representa las diferentes diagonales, mientras que la componente **x** representa la posición dentro de la diagonal. Una interpretación de coordenadas cartesianas, y otra de coordenadas diagonales puede apreciarse en la Figura 6.6 La forma de hacer esto en código sería la siguiente:

```
if(N==M) // si matriz cuadrada
{
   blockIdx_y=blockIdx.x;
   blockIdx_x=(blockIdx.x+blockIdx.y)%gridDim.x;
}
else
{
   int bid = blockIdx.x+gridDim.x*blockIdx.y;
   blockIdx_y=blockIdx.x;
   blockIdx_x=((bid/gridDim.y)+blockIdx_y)%gridDim.x;
}
```

Permitiendo matrices cuadradas y no cuadradas. Revisitando la Figura 6.5, podemos ver en la Figura 6.7 como el reordenamiento diagonal soluciona el problema.

Mapeo semi-automático

Aquí también vamos a utilizar Pluto para C para generar una planificación sobre la que partir. La opción *fuse* aquí obviamente tampoco hace nada, ya que no hay varios bucles que fusionar. La opción *tile* divide los dos bucles que van recorriendo los elementos de la matriz en dos partes, donde los bucles externos recorren los bloques externos de la matriz, y los bucles internos recorren estos bloques, realizando de este modo la trasposición de una forma bloqueada. Como ya dijimos en el apartado de la multiplicación de matrices, estos dos bucles divididos en dos son los que implícitamente en la codificación para CUDA se convierten en los bloques de la rejilla, y el número de hilos de cada bloque, por lo que un mapeo directo nos llevaría a hacer lo mismo que en la implementación manual sencilla con memoria global, o como mucho aprovechar la jerarquía de memoria y hacer una versión igual al mapeo manual con memoria compartida. Por lo tanto en este paso no se nos aporta nada nuevo ni interesante para un mapeo en GPU.

Mapeo automático

Utilizando la herramienta PPCG introduciendo unos tamaños de tiling, bloques y grid iguales a los usados en la versión manual para tarjeta gráfica, se puede observar que el código generado es muy parecido a la versión manual con memoria compartida, al igual que

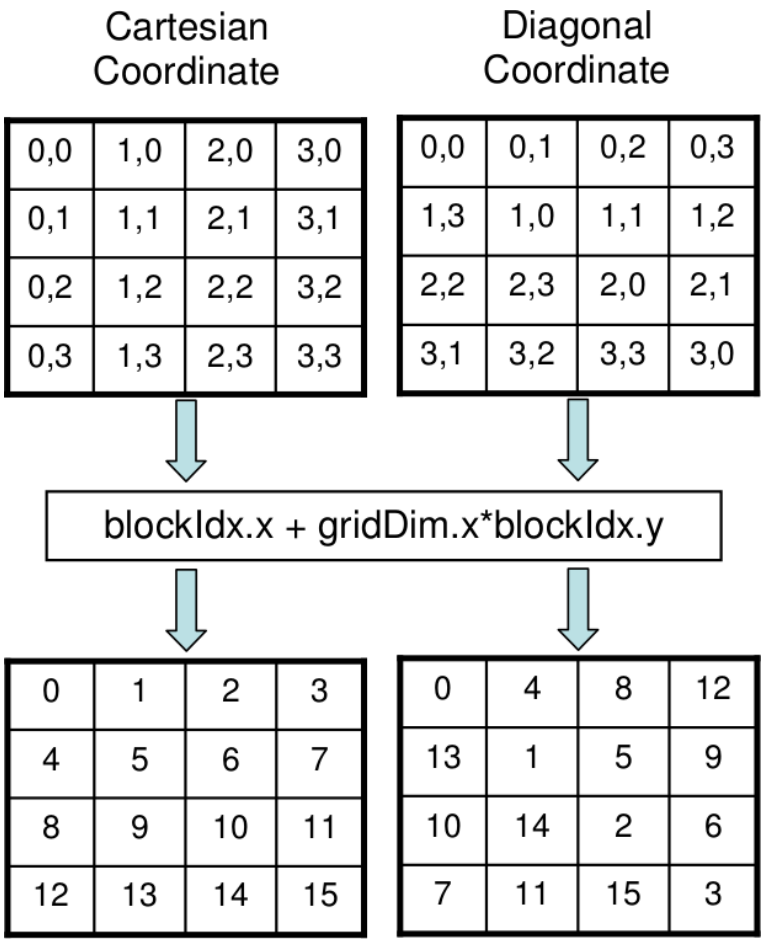


FIGURA 6.6: Distintos mapeos de blockIdx

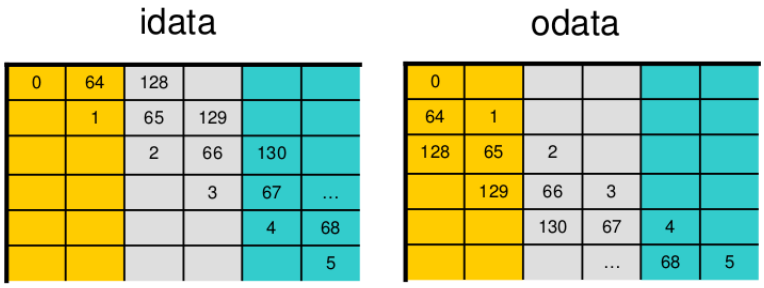


FIGURA 6.7: Reparto diagonal de los datos en la memoria en hilos en ejecución

ocurría con la multiplicación de matrices. En este caso como diferencias al mapeo manual usando memoria compartida, se observa que al reservar los 16×16 elementos de memoria compartida no se inserta un *padding* para evitar conflictos de banco. Además genera un código algo más ofuscado, ya que introduce dos bucles for dentro del kernel que no sirven para nada, ya que sólo se ejecutan una vez, y añade un *__syncthreads()* después de la escritura de memoria compartida a memoria global, que no necesita estar ahí.

Resultados

En esta ocasión las tres versiones estudiadas de la trasposición de matrices han obtenido un resultado similar.

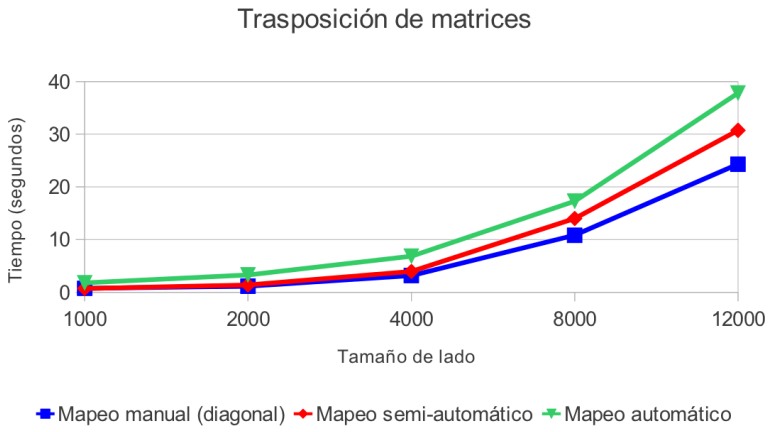


FIGURA 6.8: Comparación de mapeos en trasposición de matrices

La versión manual utilizando una distribución de lanzamiento de bloques en diagonal es ligeramente mejor que las demás, precisamente por lo comentado anteriormente, ya que de este modo se reparte la escritura de datos entre las diferentes particiones de memoria global. El mapeo semi-automático, que sería igual que un mapeo manual sin utilizar esta distribución diagonal, es ligeramente peor, ya que sufre de una distribución irregular de escrituras en las diferentes regiones de memoria, lo que hace que se limite el ancho de banda a memoria global por hilo. El mapeo automático vimos que era muy similar al semi-automático, pero introduciendo más código innecesario. Eliminando este código innecesario directamente en la generación de código por parte de PPCG podría mejorar el resultado, e igualarlo a una implementación manual en la que no se ha tenido en cuenta la distribución en regiones de la memoria global.

No parece demasiado complicado incluir un chequeo en PPCG de la forma en la que se acceden los datos de memoria global, simplemente habría que ver si los accesos o escrituras se apiñan en un momento determinado en una región de la memoria global, y si es así, utilizar una forma alternativa de ir lanzando los bloques, como la distribución diagonal.

6.3. Jacobi

Jacobi 1D

Mapeo manual

Hay un mapeo muy directo que sería realizar un kernel con una rejilla unidimensional con N hilos divididos en bloques de 64 o 128 hilos, donde cada hilo generaría un elemento del nuevo vector, utilizando para ello memoria global. Después de esto, se utilizaría otro kernel muy sencillo para volcar los elementos del vector b generado en a. El kernel tendría que llamarse T veces.

Pero aprovechando la jerarquía de memoria, y haciendo en este caso reuso de datos, podemos mejorar la eficiencia de la implementación. En vez de utilizar directamente los datos de memoria global, habría que llevarlos a memoria compartida. Para un bloque de 128 hilos, como los hilos primero y último tienen que acceder al dato anterior al primero, y al posterior al último, necesitamos un array de memoria compartida de $128 + 2$ elementos. Primero los hilos tienen que colaborar para traer cada uno un elemento de memoria global a memoria compartida, excepto el primer y último hilos del bloque, que además tienen que traer el anterior al primero y el posterior al último. Una vez hecho esto necesitamos una barrera de sincronización para que al leer los datos estén ya todos escritos en memoria compartida, y una vez hecho esto, haríamos el cómputo utilizando estos datos de memoria compartida.
En este caso cada dato se está utilizando tres veces, por lo que es conveniente tener ese dato en una memoria que nos permita un acceso más rápido, como es en este caso la memoria compartida.

Mapeo semi-automático

Hemos utilizado la herramienta Pluto para C para generar una planificación para a partir de ella generar código CUDA. Hemos utilizado diferentes opciones de la herramienta, y lo que ocurre es que, para el bucle externo que no es paralelizable, añade más iteraciones, por lo que hemos de suponer que elimina paralelismo de las partes más internas del código. Luego añade un bucle paralelo pero con muy pocas iteraciones, que no son suficientes para poder hacer uso de una GPU, haciendo distinción de casos especiales, como son los extremos del vector, donde no se realizan las operaciones. Además los límites de los bucles que podrían ser más paralelos pasan de ser constantes a ser variables, necesitando un tamaño de rejilla distinto para cada iteración.

Mapeo automático

Primero vamos a comentar el código generado por la versión de Pluto para CUDA. Este programa tiene la limitación de no poder generar código para el *host*, por lo que sólo genera un kernel. En esta ocasión el código generado es idéntico al de Pluto para C, pero con restricciones para los hilos. Inserta dentro del kernel el bucle externo que en las versiones manuales iría fuera del kernel, haciendo un *__syncblocks()* al final de cada iteración. Comentar que el código no genera un resultado válido, y dada la complejidad del código introducido dentro del kernel, es difícil de ver donde podría encontrarse el fallo.

Después se ha probado a utilizar PPCG, con unos resultados muy diferentes. Se han introducido unos tamaños de tiling, bloques y grid iguales a los usados en el mapeo manual, y el código resultante es idéntico al mapeo manual utilizando memoria compartida. Con bloques

unidimensionales de 128 hilos, es capaz de darse cuenta de que necesita 2 datos más de la frontera, y reserva 130 elementos en el array de memoria compartida. Posteriormente, lo único distinto que hace es guardar la operación de la suma de las tres posiciones consecutivas del vector en un registro, y luego volcar ese registro en memoria global.
En el código del host mantiene el bucle externo, llamando al kernel T veces. El segundo kernel, el de copia del resultado al vector original, es un poco extraño. Funcionalmente es correcto, pero en vez de hacer una copia directa de memoria global a memoria global, lo que hace es copiar de memoria global a un registro, posteriormente ese registro lo introduce en otro registro, y finalmente este segundo registro vuelca su contenido en memoria global.

Resultados

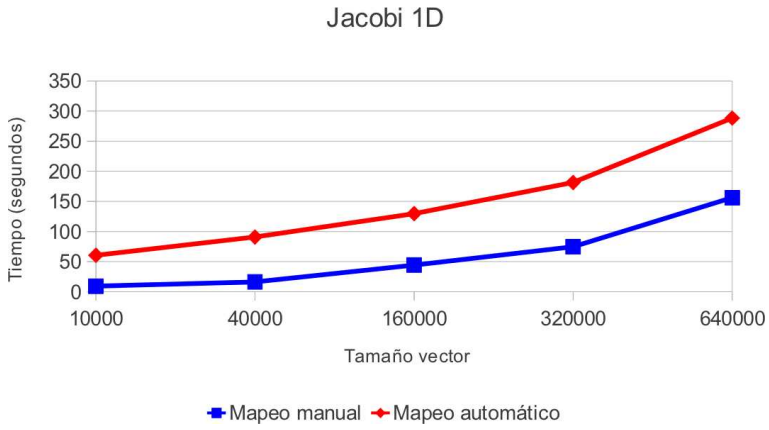


FIGURA 6.9: Comparación de mapeos en Jacobi 1D

En la versión de Jacobi se aprecia una diferencia de tiempo bastante notable entre el mapeo manual y el automático. En esta ocasión no hay resultados de mapeo semi-automático debido a la mala planificación generada por Pluto para C en el sentido de que no es regular ni lo suficientemente paralela como para mapearla para GPU.
En este caso, el segundo kernel del mapeo automático, teniendo que pasar cada dato por dos registros antes de almacenarse en memoria global de nuevo, puede ser la causa de esta diferencia de rendimiento, además de alguna condición innecesaria y bucles de una sóla iteración en el primer kernel, que podrían evitarse.

Jacobi 2D

Mapeo manual

Es exactamente igual que la versión unidimensional del algoritmo, por lo que podemos generar una versión sólo con memoria global. Tendríamos una rejilla de $N \times N$ hilos repartidos en, por ejemplo, bloques de 16×16 hilos. Cada hilo generaría un nuevo valor de la matriz B, leyendo los valores de A de memoria global. Posteriormente otro kernel de copia simple traspasaría los valores de la matriz B a la matriz A. Estos kernels se llamarían T veces.

Al igual que la versión unidimensional con memoria compartida, aquí podemos hacer algo equivalente. Si tenemos bloques de 16×16 hilos, necesitaremos un array bidimensional de memoria compartida de $16+2 \times 16+2$ elementos, para poder almacenar también los bordes

para los hilos con índice inicial o final para cualquiera de las dos dimensiones. Por lo tanto cada hilo traería un elemento de la matriz A a memoria compartida, excepto los hilos de los bordes, que traerían un elemento adicional. Una vez hecho esto, se pone una barrera de sincronización, y se opera sobre los datos en memoria compartida. En este caso, al tener bloques de memoria compartida de 18×18 elementos, no se producen conflictos de banco.

Mapeo semi-automático

Utilizando Pluto para C para intentar conseguir una planificación para CUDA, podemos observar que el código generado es también similar al unidimensional, sacando iteraciones al bucle más externo, perdiendo paralelismo, y generando unos bucles internos mucho menos paralelos, y con una gran cantidad de condiciones. Los límites de los bucles internos son también variables, por lo que paralelizar esto para GPU no sería para nada eficiente.

Mapeo automático

Se ha utilizado la herramienta Pluto para generar código CUDA, y básicamente ha cogido el código C, lo ha introducido en un kernel, y ha puesto condiciones para diferenciar el trabajo de cada hilo. También ha introducido el bucle externo dentro del kernel, utilizando *__syncblocks()* al final de cada iteración. Este código tampoco genera un resultado válido.

Utilizando PPCG con los valores de tiling, bloques y grid iguales a los del mapeo manual hemos obtenido un resultado muy similar al del mapeo manual utilizando memoria compartida. Para la memoria compartida ha tenido en cuenta los bordes, y con un bloque de 16×16 hilos, ha creado un array bidimensional de memoria compartida de 18×18 elementos. Primero rellena con valores de la memoria global el bloque de memoria compartida, e introduce una barrera de sincronización. Una cosa extraña que hace es generar un array de 16×16 registros para almacenar los cálculos que luego se escribirán en memoria global de nuevo. Cada hilo accede al array de registros con el mismo índice en la escritura y en la lectura, por lo que se está haciendo un uso innecesario de registros. Además después de rellenar el array de registros, crea otra barrera de sincronización, cosa que podría llegar a ralentizar el código.
En el segundo kernel, el de la copia simple para trasvasar los valores de la matriz B a la matriz A, hace lo mismo que en la versión unidimensional, pero además utilizando dos arrays bidimensionales de 16×16 registros. Primero lee de memoria global e introduce en el primer array de registros. Posteriormente lee el registro escrito, y lo introduce en el segundo array de registros. Por último, lee el registro de la posición correspondiente y lo escribe en memoria global. Aquí también se está haciendo un uso innecesario de registros, ya que además se podría hacer una copia simple de memoria global a memoria global.

Resultados

En la versión bidimensional de Jacobi, sigue existiendo cierta diferencia en el tiempo de ejecución entre la versión manual y la automática. Aquí también se hace un paso innecesario por dos registros de cada dato en el segundo kernel, que realiza una copia simple. El primer kernel, exceptuando un bucle for con una sola iteración, es prácticamente igual a la versión manual, con el mismo uso de la memoria compartida.
La eliminación de código innecesario de paso por registros debería de ser fácilmente implementable en PPCG, simplemente observando el origen y destino de los datos de cada hilo, y eliminando variables intermedias.

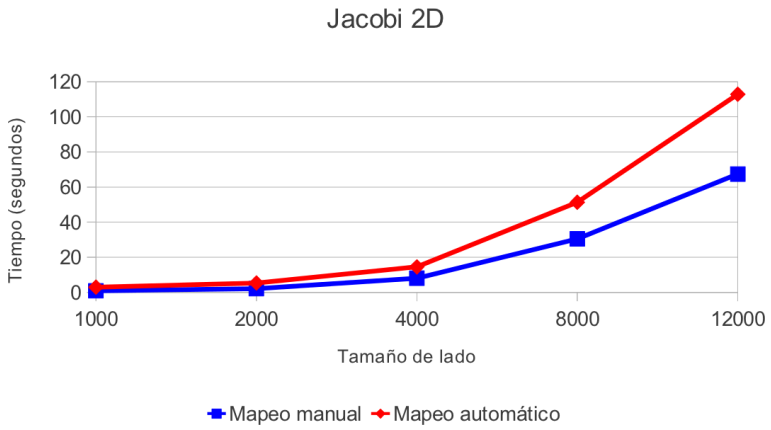


FIGURA 6.10: Comparación de mapeos en Jacobi 2D

6.4. LU

Mapeo manual

Un mapeo directo utilizando tan solo memoria global constaría de dos kernels, un primer kernel con una rejilla unidimensional, con un tamaño de N hilos divididos en bloques unidimensionales de 64 o 128 hilos. En este kernel se generaría una fila de los elementos que más tarde se volverán a utilizar. Después habría otro kernel, esta vez con una rejilla bidimensional, con $N \times N$ hilos divididos en bloques de 16×16. En esta ocasión cada hilo generaría un elemento del rectángulo que se actualiza dentro de la matriz, todo ello leyendo los datos de memoria global. Estos kernels se llamarían dentro de un bucle, que iría iterando a través de las filas de la matriz.

En este caso como posible mejora, ya que la forma de reutilizar los datos es distinta, sería la siguiente. En vez de generar toda la fila en un kernel para luego utilizarlo en el siguiente, se unifica en un sólo kernel, que por hilo generaría un sólo elemento de esa fila, lo introduce en un registro, y posteriormente se reutiliza ese elemento calculado para generar toda una columna del rectángulo dentro de la matriz que se está actualizando. Para esta actualización necesitamos el dato de la fila que está en un registro, y la fila exterior izquierda del rectángulo, que puede leerse directamente de memoria global, o para tener escrituras fusionadas, de memoria compartida.

Mapeo semi-automático

Se ha utilizado Pluto para C para ver el código generado y su posible aplicación a una implementación con CUDA. Con la opción *fuse* el código se transforma y se crea una versión a partir de la cual se podría inferir un mapeo para CUDA como el manual optimizado que se ha explicado antes. En vez de calcular primero toda la fila, y luego utilizarla para actualizar el rectángulo dentro de la matriz, calcula un sólo elemento de la fila, y actualiza parcialmente el rectángulo, acercando así la producción del dato y su consumo. Aplicando además la opción *tile* a Pluto, obtenemos la banda externa paralela, es decir, la distribución de hilos entre distintos bloques, y la forma de recorrer los hilos dentro de uno.

Mapeo automático

Para el mapeo automático del código se ha utilizado PPCG con los valores de tile, bloque y rejilla iguales a los del mapeo manual optimizado. En este caso, la solución dada por PPCG no es exactamente igual a la manual, o una semi-automática pensada a partir de Pluto par C. En vez de un sólo kernel en el que se vaya generando un dato de una fila y posteriormente actualizando la matriz, PPCG ha diferenciado dos. El primero de ellos hace el cálculo de la nueva fila. Crea un array para memoria compartida de un único elemento donde introduce el dato de la diagonal que va a ser leído para realizar la división por el resto de elementos. De esta forma, sólo un hilo de cada bloque realiza la lectura de este elemento, pero posteriormente necesita de una sincronización de hilos. Una vez pasada la barrera de sincronización, se hace la operación de actualización de los elementos a la derecha de la diagonal en esa fila, actualizando un elemento por hilo.
El segundo kernel realiza la actualización del rectángulo dentro de la matriz una vez calculada la nueva fila. Lee el elemento que se va a actualizar, y lo introduce en un registro. Introduce los datos de la columna a la izquierda del rectángulo que se va a actualizar, y que se van a utilizar para las operaciones, en memoria compartida, y los datos de la fila que se actualizó en el anterior kernel en un array de registros. Los datos introducidos en la memoria compartida, es decir, el trozo de columna con el que se va a operar, son los que se van reutilizando, mientras que los datos de la fila sólo se necesitan una vez, de ahí que vayan a registros. Una vez hecho todo esto, el resultado de la operación va otra vez a registro, y de ahí se pasa a memoria global.
En este mapeo automático, para cada kernel se hace más uso de varios niveles de la jerarquía de memoria, intentando optimizar cada uno de ellos por separado, por lo tanto es una aproximación distinta al mapeo manual, y una alternativa a tener en cuenta.

Resultados

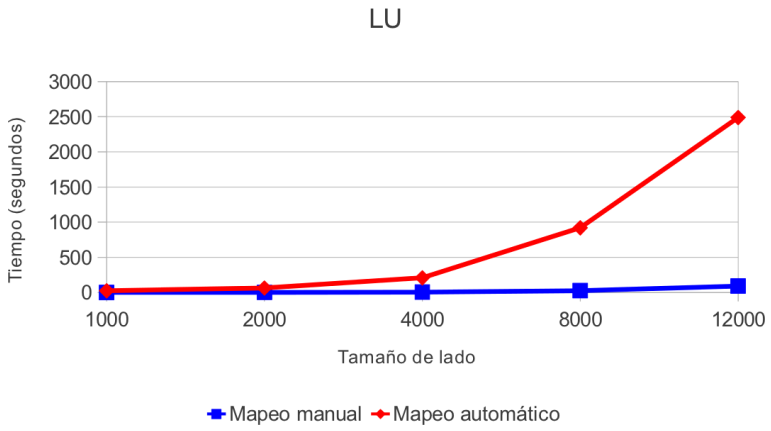


FIGURA 6.11: Comparación de mapeos en LU

Para este benchmark la diferencia de tiempos de ejecución es abismal. Sólo se han tenido en cuenta el mapeo manual y el automático, ya que el semi-automático conducía a una planificación exactamente igual a la manual.
En este caso es claro el motivo de la diferencia de rendimiento. En el mapeo manual sólo se utilizan registros, una pequeña cantidad de memoria compartida, y obviamente la memoria global, pero en el automático, por intentar explotar algo más el uso de varios niveles de

la jerarquía de memoria de la tarjeta, utilizando algo más la memoria compartida, se ha caido en una mala implementación. La implementación manual contiene un solo kernel, y aprovecha al máximo el reuso de los datos, además de acercar productor y consumidor. En la implementación automática aleja la generación y el consumo de datos con tal de introducir más cantidad de información en memoria compartida, dividiendo la ejecución en dos kernels, evitando así el poder introducir datos en registros.
Esto nos lleva a pensar que la utilización de los niveles de memoria de la tarjeta gráfica por parte de PPCG todavía tiene que mejorar, ya que para cada dato usado en un kernel la decisión de en qué nivel ponerlo no sólo debería depender del nivel de reuso que se pueda hacer de él, y con esto hacer una división de kernels, sino que este nivel de reuso pueda hacer que se fusionen o dividan kernels.

6.5. FDTD-2D

Mapeo manual

En una primera aproximación se podría dividir el código en tres kernels distintos, uno que actualice la fila de la matriz auxiliar *ey* con el valor de la iteración, simplemente asignando el valor a memoria global. Un segundo kernel haría la actualización de las dos matrices auxiliares, ya que no interfieren una con la otra en su actualización, y se puede hacer en paralelo. Se leerían los valores de la matriz principal de la memoria global. Y un último kernel haría la actualización de la matriz principal, utilizando los datos en memoria global de las matrices auxiliares.

En una versión más sofisticada, se mantiene esta división de kernels, pero aprovechando la jerarquía de memoria de la tarjeta gráfica. El primer kernel, donde se actualiza la primera fila de una de las matrices auxiliares, no puede mejorarse de ninguna manera, ya que no hay reuso de memoria de ningún tipo. Existe una primitiva de inicialización de memoria a un valor en CUDA, llamada *cudaMemSet()*, pero funciona a nivel de byte, por lo que no sirve para nuestro propósito.
En el segundo kernel, donde se van actualizando las matrices auxiliares con valores de la matriz principal, se puede observar que las posiciones de la matriz principal que se leen para las dos actualizaciones son casi las mismas para un bloque de hilos, todas dentro de un rectángulo. Por lo tanto se introduce un bloque de la matriz principal a memoria compartida, cooperando los hilos de un bloque llevando cada uno un dato, excepto un grupo de hilos, que además traen una fila y una columna adicionales. En total el array de memoria compartida para un bloque tendría un tamaño de 17×17 teniendo un tamaño de bloque de 16×16. Después se añade una barrera de sincronización de hilos, y se escribe en memoria principal el nuevo valor de las matrices auxiliares, resultado del cálculo de los valores de la matriz principal que están en memoria compartida. Por último, el kernel de actualización de la matriz principal, también hace reuso de datos de las matrices, de forma muy similar a como se hacía en el segundo kernel. En un bloque de 16×16 hilos, se necesitan para cada una de las matrices auxiliares una fila, y una columna más de datos. Por lo tanto se tienen dos arrays de memoria compartida, uno con 16×17 elementos, y otro con 17×17 elementos, en este caso una fila y además una columna más, pero en este caso para evitar conflictos de banco. Una vez introducidos los datos de las matrices auxiliares en memoria compartida, se hace una barrera de sincronización, y se genera el nuevo valor para la matriz principal, operando sobre los datos de las matrices auxiliares en memoria compartida.

Mapeo semi-automático

Se ha utilizado Pluto para C para intentar obtener un código CUDA a partir del código transformado por este programa. Para este caso particular, el código generado pierde muchísimo paralelismo, ya que trata de ir haciendo las operaciones de actualizar las matrices auxiliares y la principal de forma seguida, en vez de en bucles distintos, para poder utilizar directamente los datos generados en las matrices auxiliares para generar el siguiente dato de la matriz principal. Por lo tanto, se mejora en localidad de datos, pero la pérdida de paralelismo hace imposible obtener un código para GPU a partir de esta versión.

Mapeo automático

Para el mapeo automático del código se ha utilizado PPCG con los valores de tile, bloque y rejilla iguales a los del mapeo manual optimizado. El código generado por PPCG es muy similar al del mapeo manual utilizando memoria compartida. También tiene una división en tres kernels. El primero de ellos va realizando la actualización de la primera fila de una de las matrices auxiliares. En vez de escribir el valor directamente sobre memoria global, se escribe el dato en un registro y posteriormente se escribe en memoria global.
En el segundo kernel, donde se actualizan las matrices auxiliares, también se utiliza memoria compartida, un array con el mismo tamaño que en el mapeo manual optimizado. Primero lee los datos de la matriz principal y los escribe en memoria compartida, y luego el nuevo dato de las matrices auxiliares se calcula utilizando los datos almacenados en memoria compartida.
Por último, en el kernel de actualización de la matriz principal, también se hace uso de la memoria compartida, introduciendo en arrays de memoria de este tipo los valores de las matrices auxiliares que necesitará cada bloque de hilos. Los tamaños de los arrays de memoria compartida son iguales que en mapeo manual optimizado, y la única diferencia es que primero se opera sobre los datos en memoria compartida, y el resultado se almacena en un registro, que posteriormente se escribe en la posición correspondiente de memoria global.

Resultados

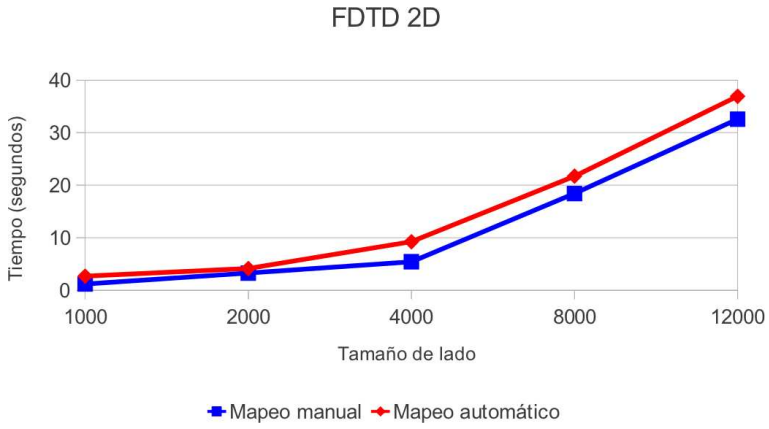


Figura 6.12: Comparación de mapeos en FDTD 2D

En esta ocasión la diferencia existente entre el tiempo de ejecución de la versión manual del código y la versión automática no es demasiado grande, sobre todo para tamaños

pequeños. Aquí también se hacen transferencias innecesarias a registros, lo que deteriora algo el rendimiento, pero no tanto como en otros benchmarks, ya que el resto del mapeo es prácticamente igual.

6.6. Non-Negative Matrix Factorization (NMF)

Mapeo manual

Las operaciones punto a punto se mapean como un kernel bidimensional en el que cada hilo realiza la operación punto a punto sobre un único elemento. En este caso, a pesar de existir un alto grado de paralelismo, el reuso inexistente de los datos impide aprovechar la jerarquía de memoria de la GPU de un modo eficiente. Por lo tanto cada hilo leerá un elemento de cada matriz, realizará la operación correspondiente, y se escribirá en el sitio correspondiente de la matriz resultado.
Para las multiplicaciones de matrices se hace uso de la librería de CUDA CUBLAS [1],

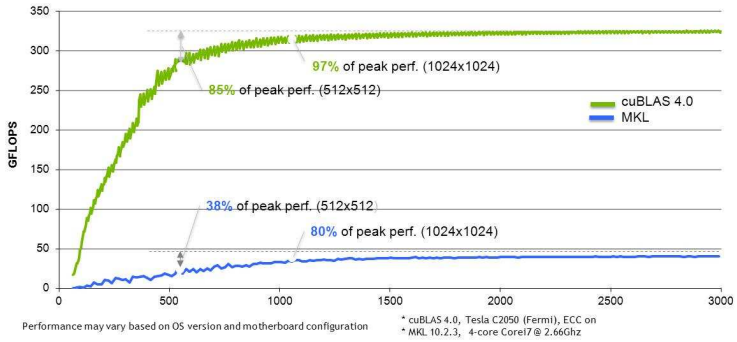


Figura 6.13: CUBLAS vs Intel MKL

una librería de algebra lineal optimizada para GPUs. En la Figura 6.13 podemos observar la diferencia de rendimiento que se puede llegar a alcanzar entre el uso de CUBLAS, y el uso de MKL [2] sobre CPU.
En CUBLAS existe una versión de *gemm* para la multiplicación de matrices. *Gemm* (General Matrix Multiply), es una función existente en la mayoría de las librerías de algebra lineal la cual realiza la operación $C = \alpha AB + \beta C$. En nuestro caso α es 1, mientras que β es 0, lo que nos deja la operación en $C = AB$. También permite pasar como parámetro si la matriz a multiplicar está traspuesta o no, lo que nos permite ahorrarnos el paso de invertir una matriz antes de multiplicarla, con la consecuente mejora de rendimiento. La versión inicial de esta operación sobre GPUs para esta librería, disponible en las versiones de CUBLAS 1.x, fue sustituida por una con un rendimiento bastante superior, desarrollada por Vasily Volkov [30].

En este caso se ha optado por ir un paso más allá generando una versión manual del código multiGPU, debido a que las herramientas de paralelización automática de código todavía no son capaces de usar más de una. Al utilizarse las GPUs como coprocesador, es posible dividir el problema entre varias, pudiendo añadir un nivel más de paralelismo, que podría ser tenido en cuenta, ya que se puede multiplicar la potencia de cómputo para problemas lo suficientemente grandes. Una forma muy sencilla de hacer esto sería utilizando hilos con OpenMP.

Para este algoritmo en particular, se puede hacer una división por filas o por columnas de la matriz V haciendo cálculos de trozos de matrices que luego tendrían que volver a unirse. Hay que tener en cuenta que este algoritmo es iterativo, y que una vez terminada la generación de las nuevas matrices W y H, se necesitan completas otra vez para la siguiente iteración. Por lo tanto si cada tarjeta hace un cálculo parcial de estas matrices, la unión de estos parciales se debería hacer en GPU. Esto hace que se necesiten transferencias entre las tarjetas gráficas, en forma de broadcast, por lo que se necesita que los parciales pasen de una GPU a la memoria principal del sistema, y de ahí al resto. Minimizar las transferencias CPU-GPU es una de las claves para mejorar la eficiencia de un programa creado para GPU, por lo que también debería ser tenido en cuenta.

Mapeo semi-automático

Primeramente se ha hecho uso de la herramienta Pluto para generar un código C optimizado, más sencillo de generar correctamente por este tipo de herramientas, para observar ya a nivel de código qué transformaciones ha hecho sobre la planificación original.
Primero se ha utilizado una versión sencilla en C sin ningún tipo de optimización, como la mostrada al principio de esta sección, para que la herramienta pueda trabajar de la mejor forma posible, y que sigue estos pasos de forma secuencial:

- Actualización de H:
  - $W'V = W' \times V$
  - $W'W = W' \times W$
  - $W'WH = W'W \times H$
  - $H = H. \times W'V. \div W'WH$
- Actualización de W:
  - $VH' = V \times H'$
  - $HH' = H \times H'$
  - $WHH' = W \times HH'$
  - $W = W. \times VH'. \div WHH'$

Pluto en este caso, intenta acercar lo más posible acercar el lugar donde se producen los datos al lugar donde se consumen, produciendo varias fusiones de bucles.

- Actualización de H:
  - Un primer bucle fusionado entre las operaciones de generación de $W'W$ y $W'WH$, reutilizando el elemento generado por $W'W$ para generar el siguiente dato de $W'WH$.
  - Posteriormente se genera la matriz $W'V$ de forma normal.
  - Multiplicación punto a punto de H y $W'V$ y división punto a punto por $W'WH$ realizada de forma igual a la original.
- Actualización de W:
  - Un primer bucle fusionado entre las operaciones de generación de HH' y WHH' reutilizando el elemento generado de HH' para generar el siguiente dato de WHH'. Además, en el bucle donde se va generando HH', se hace un cálculo parcial de $K \times K$ elementos de la matriz VH'.

- Multiplicación de los $N - K \times K$ elementos restantes de la matriz VH'.
- Multiplicación punto a punto de W por VH' y división punto a punto por WHH' realizada de forma igual a la original.

En este caso perdemos algo de paralelismo, ya que los bucles externos son a los que se pueden aplicar los *#pragma parallel for* de OpenMP. A cambio, estamos ganando en localidad espacial, lo que permite mejorar el uso de la jerarquía de memoria.

Kernels CUDA generados

Para los bucles que fusionan la generación de $W'W$ y $W'WH$, a partir de la planificación generada en C hemos creado tres versiones distintas:

- Una primera versión en la que se paralelizan los dos bucles más externos , obteniendo una rejilla con un total de $K \times K$ hilos. En este caso, cada hilo genera un elemento de $W'W$, que puede ser almacenado en un registro para reutilizarse para el cálculo de una parte de una fila de $W'WH$. Al necesitar varios hilos para generar un elemento de $W'WH$, esta operación requiere de un acceso exclusivo de la memoria, para no pisar la escritura de algún hilo. Esto se hace mediante un mutex que utiliza las funciones atómicas incluidas en la librería de CUDA, y que resulta en una serialización del código, por lo tanto la eficiencia de esta versión en una GPU es muy baja.

- Una segunda versión en la que sólo se paraleliza el bucle más externo, por lo que se tiene una rejilla de K hilos. En este caso cada hilo genera una fila de $W'W$, y una fila de $W'WH$ completa, secuencializando la parte de la suma de los resultados parciales de $W'WH$. En este caso también se aprovecha cada elemento de $W'W$ introduciéndolo en un registro, para utilizarlo en el cálculo de $W'WH$.

- Una tercera versión en la que también se paraleliza únicamente el bucle más externo, pero esta vez tenemos una rejilla de $K \times$tamaño de bloque. Un único hilo de cada bloque generaría una parte de una fila de $W'W$, que se introduciría en memoria compartida para utilizarla para generar una parte de una fila de $W'WH$, evitando también de esta forma la escritura por parte de varios hilos en la misma posición de memoria.

La multiplicación de $W'V$ la hemos realizado con la operación *gemm* incluida en la librería CUBLAS, ya que es óptima.
Las multiplicaciones y divisiones punto a punto se han realizado paralelizando los dos bucles más externos, utilizando únicamente memoria global, ya que no hay reuso de ningún tipo de memoria. En el caso de la actualización de W se crearía una rejilla de $N \times K$ hilos, mientras que en el caso de la actualización de H se crearía una rejilla de $K \times M$ hilos, repartidos en bloques bidimensionales de 16×16 hilos.

Para los bucles que generan HH', WHH' y parte de VH' también se han hecho varias versiones:

- Una primera versión en la que se paralelizan los dos bucles más externos, obteniendo una rejilla con un total de $K \times K$ hilos. En esta ocasión, cada hilo genera un elemento de HH', que puede ser almacenado en un registro para reutilizarse para el cálculo de una parte de una columna de WHH'. Al necesitar varios hilos para generar un elemento de WHH', al igual que pasaba con la generación de $W'WH$, esta operación requiere de un acceso exclusivo a memoria, para no realizar escrituras desde varios hilos

a la misma posición de memoria. También se realiza mediante mutex implementados con operaciones atómicas, por lo que el rendimiento también es muy bajo. En este caso también se aprovecha para generar una parte de VH', generando cada hilo un elemento de la nueva matriz.

- Una segunda versión en la que sólo se paraleliza el bucle más externo, por lo que se tiene una rejilla de K hilos. En este caso cada hilo genera una fila de HH', y una fila de WHH' completa, secuencializando la parte de la suma de los resultados parciales de WHH'. En este caso también se aprovecha cada elemento de HH' introduciéndolo en un registro, para utilizarlo en el cálculo de WHH'. Aquí también se generaría por hilo una fila de VH'.

- Una tercera versión en la que también se paraleliza únicamente el bucle más externo, pero esta vez tenemos una rejilla de $K\times$tamaño de bloque. Un único hilo de cada bloque generaría una parte de una fila de HH', que se introduciría en memoria compartida para utilizarla para generar una parte de una fila de WHH', evitando también de esta forma la escritura por parte de varios hilos en la misma posición de memoria. En este caso, el primer hilo de cada bloque también realizaría el cálculo de una fila de VH'.

- Una última versión igual a la segunda versión, pero donde se fusionaría este kernel con el kernel de realizar la multiplicación parcial de los $N - K \times K$ elementos restantes. Aquí hay que diferenciar para saber que bloque o que hilo tiene que hacer el cálculo del primer kernel, y cual el del segundo.

Para la versión del kernel anterior no fusionada con el cálculo parcial de los $N - K \times K$ elementos restantes de VH', se ha optado por una multiplicación de matrices bloqueada, en la que se aprovecha la memoria compartida para acelerar los cálculos.

Mapeo automático

Para este código, PPCG genera código, pero el resultado final del algoritmo no es correcto. Comentar brevemente que la herramienta genera un primer kernel en el que inicializa $W'WH$ a 0, un kernel bastante complejo donde realiza casi todos los cálculos, y un último kernel que realiza parte de las operaciones punto a punto. Observando la planificación bloqueada sin añadir aún el bloqueo externo para la paralelización con CUDA se ha podido observar que no genera un código muy diferente al generado por la versión C de Pluto, sin embargo la parte del algoritmo de PPCG que transforma esto a código CUDA, intenta maximizar el reuso de los datos a través de todo el flujo de una iteración del algoritmo, perdiendo también algo de paralelismo.
En un análisis algo más profundo, se ha observado que de la actualización de las dos matrices resultado, W y H, la generación de H sí es correcta, pero la herramienta se para ahí y no genera código para la actualización de W. Este fallo se ha reportado a los desarrolladores de la herramienta, los cuales no saben aún a ciencia cierta la causa del comportamiento extraño en este caso en particular.

Resultados

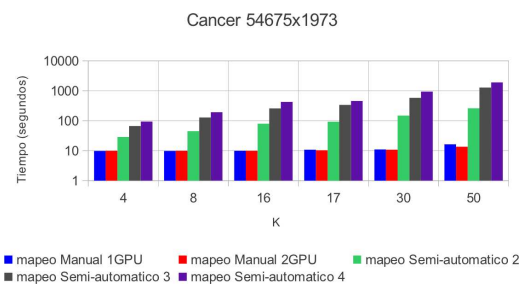
Al ser este un algoritmo usado en casos reales, se ha decidido hacer las pruebas sobre tamaños de problema conocidos, tales como ORL, un conjunto de datos compuesto por 400 imágenes de caras de la base de datos ORL de Cambridge, correspondiendo a 40 personas,

10 imágenes por persona, y cada imagen con un tamaño de 92×112 pixeles.

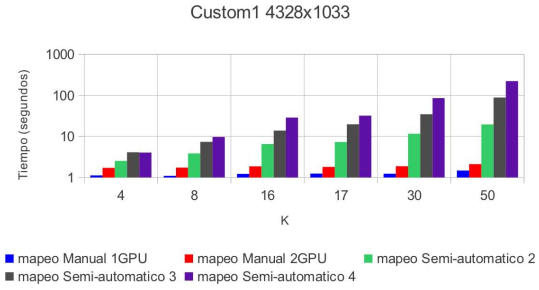
Se han tenido en cuenta los mapeos manuales en 1 y 2 GPUs, y 3 de las 4 versiones de código semi-automático. La versión del código semi-automático con las operaciones atómicas no se ha incluido aquí porque para tamaños de problema grandes el algoritmo no termina. La segunda versión es, como ya se comentó antes, aquella en la que sólo se paraleliza el bucle más externo. La tercera versión también paralelizaba únicamente el bucle más externo, pero intentaba hacer más reuso de datos. Y la cuarta versión es como la segunda, pero con kernels fusionados.

Los resultados son todos muy similares, siempre son mejores para los mapeos manuales. Para tamaños de problema pequeños, la diferencia de tiempo entre los mapeos manuales, y los mapeos semi-automáticos aumenta con Ks mayores. En tamaños de problema más grandes también se observa, pero de un modo más sutil.
En casi todos los experimentos, la diferencia de tiempo de ejecución entre la versión manual con 1 y 2 GPUs es mínima. Para tamaños de problema pequeños, o con una de las dos dimensiones pequeña, parece que no compensa dividir el problema. Únicamente en el experimento Custom2 se aprecia claramente que compensa dividir el problema entre varias tarjetas gráficas, debido a un tamaño de problema muy grande, con las dos dimensiones de la misma magnitud.
Comparando las versiones semi-automáticas, se puede apreciar que siempre la mejor versión es la segunda, seguida de la tercera, y por último, la más ineficiente es la versión en la que se fusionan los kernels. Al fusionar kernels, hay un mayor número de hilos que no tienen que hacer nada, pero ocupan los multiprocesadores de la tarjeta gráfica un tiempo, y hacen que se pierda rendimiento. La segunda versión podría parecer más ineficiente que la tercera, pero hay que tener en cuenta que en la tercera versión de los kernels, un único hilo de cada bloque genera un resultado que luego usarán el resto, pero todos estos otros hilos deben esperar al hilo que genera el primer resultado, por lo que también se produce una espera innecesaria, que puede empeorar el rendimiento.

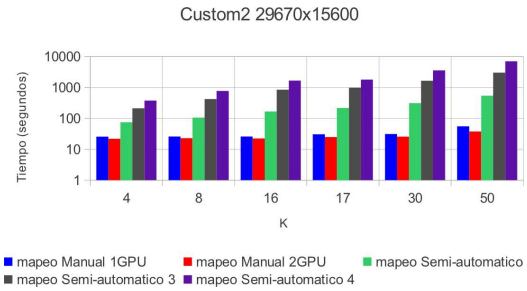
En este caso es claro que la forma de transformar los datos por los programas de paralelización automática de código es poco eficiente, ya que acerca datos que se producen y consumen, pero modifica el código de tal manera que se los bucles que se pueden paralelizar tienen una dimensión mucho menor.



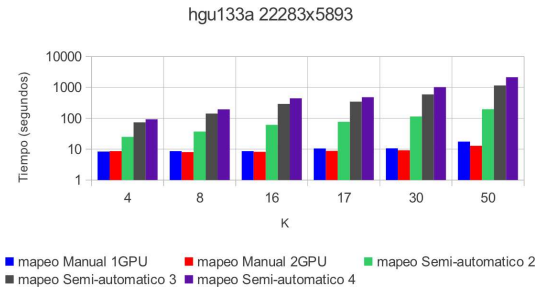
Comparación de mapeos en NMF 54675x1973



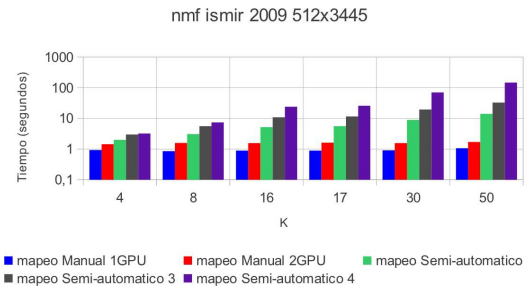
Comparación de mapeos en NMF 4328x1033



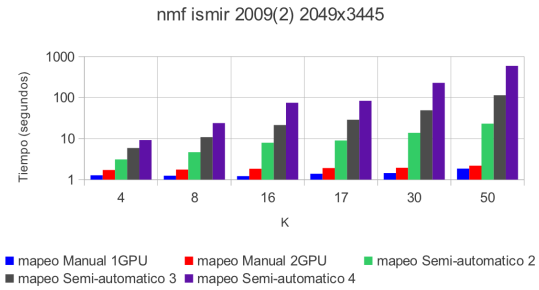
Comparación de mapeos en NMF 29670x15600



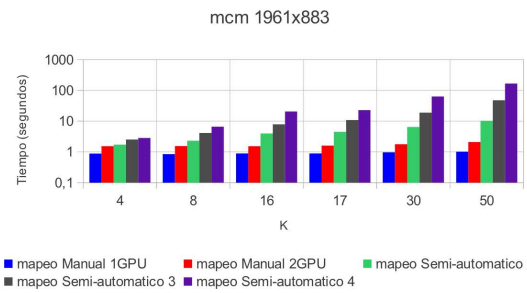
Comparación de mapeos en NMF 22283x5893



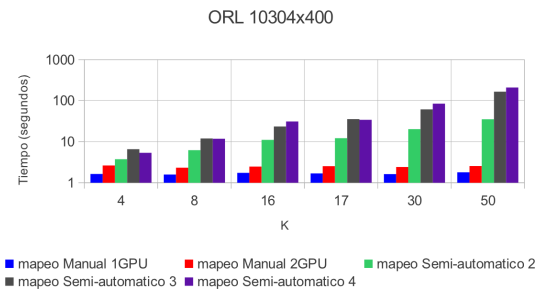
Comparación de mapeos en NMF 512x3445



Comparación de mapeos en NMF 2049x3445



Comparación de mapeos en NMF 1961x883



Comparación de mapeos en NMF 10304x400

Tabla 6.1: Resultados experimentos NMF

Capítulo 7

Conclusiones

En este trabajo se ha realizado una descripción de las herramientas existentes para generar código para tarjetas gráficas automáticamente, y se han analizado las que mejores resultados nos han proporcionado, comparándolas con código generado de forma manual. En el capítulo 6 se han presentado resultados sobre esta comparación, de donde se pueden obtener unas conclusiones muy similares para todos ellos.

Primeramente podemos decir que las herramientas de generación automática de código que utilizan el *Polyhedral Model* están mucho más avanzadas en la generación de código paralelo para CPU. Existe un framework (PoCC) bastante maduro con un gran abanico de opciones para realizar transformaciones, y hay proyectos en desarrollo para introducir el *Polyhedral Model* en herramientas de compilación muy populares, como GCC (Graphite) o LLVM.
En cambio las herramientas para generar código paralelo para GPU son mucho más limitadas. Muchas de ellas se basan en herramientas ya existentes para generar código paralelo para CPU, lo que, sin una modificación muy profunda de la herramienta, el código generado no puede alcanzar niveles de eficiencia aceptables. En el caso de las herramientas estudiadas, en este caso entrarían Par4all y Pluto para CUDA.
PPCG es la herramienta de las estudiadas más avanzada de todas, y a pesar de nacer a partir de Pluto, que genera código para CPU, tiene una modificación bastante más profunda. A pesar de ello, es una herramienta muy nueva, que ni siquiera puede descargarse de ninguna página web, sólo hay un repositorio para los desarrolladores al que he podido tener acceso.

Puntos débiles de la herramienta El análisis de esta herramienta tiene la finalidad de encontrar fallos en el mapeo, puntos débiles a la hora de generar código paralelo. Mediante el análisis del código generado, comparado con un código escrito manualmente, se han podido observar varias diferencias, que sacan a la luz cosas que la herramienta no está haciendo bien:

- Como fallo más destacable podemos decir que la herramienta no puede funcionar sin la interacción del usuario. Como se ha dicho anteriormente, PPCG necesita de tres ficheros distintos que proporcionen el tamaño de rejilla, bloque, y tile. Para poder proporcionar estos datos adecuadamente ya se necesita un cierto nivel de conocimientos

de CUDA. Por lo tanto, mientras esto sea necesario, la herramienta estará alejada de un uso general. Pluto en su versión para generar código para CPU, con la opción de tile activada, tiene un tamaño de tile fijo, a no ser que también en este caso se pase un fichero con el tamaño de tile deseado.

- El uso de las diferentes regiones de memoria de la tarjeta intenta ser óptimo a partir de la planificación generada, pero no siempre la planificación generada es la mejor. Esto en parte se debe a que la planificación se tiene que guiar por los datos de los ficheros que proporciona el usuario, y que impiden una planificación totalmente automática.
- De momento sólo se aprovechan tres regiones distintas de memoria de la tarjeta gráfica, memoria global, memoria compartida y registros. Esto no es óptimo para todos los casos, ya que por ejemplo, hay algoritmos que funcionan muy mal en GPUs a no ser que se utilice la memoria de texturas, que tiene unas propiedades especiales que la hacen más rapida para accesos aleatorios a la memoria.
- En el código generado hay una gran cantidad de código innecesario, como bucles con una única iteración, o transferencias inútiles a registros, o de registros a otros registros. Además a veces se generan unas condiciones difíciles de evaluar, que además no afectan en nada al flujo de ejecución.

Posibles mejoras No hay una solución fácil a todos los fallos que presenta PPCG, pero en este apartado se presentan una serie de posibles mejoras:

- El problema de tener que proporcionar datos de entrada no es fácil de solucionar. Hay una nueva versión de la herramienta que permite inferir de forma automática el tamaño de la rejilla, viendo el tamaño y dimensión de los datos de entrada, pero es casi más importante el poder tener una solución que genere automáticamente los tamaños de bloque y tile, que probablemente mejorarían la calidad del código generado.
- Para el problema del uso de sólo una parte de la jerarquía de memoria de la GPU, habría que tener en cuenta en qué casos se pueden aprovechar por ejemplo la cache de texturas o la de constantes. Como he comentado antes, la memoria de texturas es útil para accesos que no sigan un patrón conocido, y la de constantes es útil para utilizarlo como buffer de un tamaño mayor al que nos proporciona la memoria compartida, con la limitación de que no se puede reescribir en ella, por lo que aquí se podría mapear por ejemplo una máscara que se aplique sobre una imagen que se esté procesando. Una vez hecho esto, permitir a la herramienta que genere código con las funciones adecuadas para llamar a este tipo de memoria no parece complicado.
- Para el problema de la generación de gran cantidad de código innecesario, primero aclarar que en muchas ocasiones el compilador es capaz de eliminar todo este código a la hora de generar el código máquina. Pero una herramienta de generación automática de código debe siempre intentar generar un código lo más legible y compacto posible. Un análisis de definiciones y usos del código podría ser capaz, con una pasada adicional sobre el mismo, de eliminar por ejemplo el paso de datos por registros de forma innecesaria.

Por lo tanto, queda aún un gran camino que recorrer en el desarrollo de estas herramientas, que de momento son útiles en algunas ocasiones, pero que todavía no son soluciones viables para la mayoría de los problemas reales.

Bibliografía

[1] CUDA CUBLAS Library. `http://developer.nvidia.com/cublas`.

[2] Intel Math Kernel Library. `http://software.intel.com/en-us/articles/intel-mkl/`.

[3] OpenMP. `http://openmp.org/`.

[4] PoCC (the Polyhedral Compiler Collection). `http://pocc.sf.net/`.

[5] sitio web de par4all. `http://www.par4all.org/`.

[6] sitio web de piplib. `http://www.piplib.org`.

[7] Sitio web de Pluto. `http://pluto-compiler.sourceforge.net/`.

[8] U. Banerjee. Unimodular transformation of double loops. In *Advances in Languages and Compilers for Parallel Processing*, pages 192–219, 1990.

[9] Cedric Bastoul. Página de desarrollo de candl. `http://www.lri.fr/~bastoul/development/candl`.

[10] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[11] Cedric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, Olivier Temam, A Group, and Inria Rocquencourt. Putting polyhedral loop transformations to work. In *In Workshop on Languages and Compilers for Parallel Computing (LCPC'03), LNCS*, pages 209–225, 2003.

[12] Bob Bentley. Validating the intel pentium 4 microprocessor. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 244–248, New York, NY, USA, 2001. ACM.

[13] Uday Bondhugula, J. Ramanujam, and et al. Pluto: A practical and fully automatic polyhedral program optimization system. In *IN: PROCEEDINGS OF THE ACM SIGPLAN 2008 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI 08*, 2008.

[14] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Op'erationnelle*, 22, 1988.

[15] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21:313–348, October 1992.

[16] David L. Kuck. *Structure of Computers and Computations*. John Wiley & Sons, Inc., New York, NY, USA, 1978.

[17] Daniel D. Lee and H. Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401:788–791, 1999.

[18] Chao Liu, Hung-chih Yang, Jinliang Fan, Li-Wei He, and Yi-Min Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 681–690, New York, NY, USA, 2010. ACM.

[19] Geoff Lowney. Why intel is designing multi-core processors. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '06, pages 113–113, New York, NY, USA, 2006. ACM.

[20] Louis noël Pouchet. Sitio web de PolyBench. `http://www-roc.inria.fr/~pouchet/software/polybench/`.

[21] Louis noël Pouchet, Cédric Bastoul, and Albert Cohen. Letsee: the legal transformation space explorator.

[22] NVIDIA. Información sobre cuda. `http://developer.nvidia.com/what-cuda`.

[23] NVIDIA. Manual de programación en cuda version 4.0. `http://developer.download.nvidia.com/compute/doc/CUDA_C_Programming_Guide.pdf`.

[24] John Owens. Gpu architecture overview. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[25] Fabio Remondino and Niclas Bőrlin. Polylib - a library of polyhedral functions. http://icps.u-strasbg.fr/polylib/ or http://www.irisa.fr/polylib. In *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, Vol. XXXIV, Part 5/W16, H.-G. Maas and D. Schneider (Eds*, 2004.

[26] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[27] Sven Verdoolaege. isl: an integer set library for the polyhedral model. In *Proceedings of the Third international congress conference on Mathematical software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] Sven Verdoolaege and Carlos Juega. Repositorio de desarrollo de ppcg. `git://repo.or.cz/w/ppcg.git`.

[29] Vasily Volkov. Hilo de Nvidia Forums donde se expuso la versión de Volkov de multiplicación de matrices. `http://forums.nvidia.com/index.php?showtopic=47689&st=40&p=314014&#entry314014`.

[30] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[31] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26:30–44, May 1991.

[32] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[33] Jingling Xue. On tiling as a loop transformation, 1997.

Índice de figuras

Índice de tablas