Dispositivos e Infraestructuras para Sistemas Multimedia 2018

Introducción a la Programación de GPUs usando CUDA

Alberto García García
Sergio Orts Escolano
José García Rodríguez

8 de noviembre de 2018

Proudly made with LaTeX

Copyright (C) 2017-2018 Alberto García García, Sergio Orts Escolano, José García Rodríguez

Prólogo

Hubo un tiempo en el pasado no tan distante en el que la computación paralela se concebía como una herramienta exótica y poco útil por lo que fue marginada como una especialidad de las ciencias de la computación, reservada para unos pocos atrevidos o directamente para aquella gente con demasiado tiempo libre. Esta percepción ha cambiado profundamente en los últimos años. El mundo de la computación ha mutado tan bruscamente hasta el punto en que, lejos de ser una especialidad olvidada, prácticamente todo programador que se precie ha de poseer conocimientos en programación paralela para ser efectivo en sus proyectos independientemente del ámbito de los mismos. Es un hecho que la revolución de la computación paralela ya ha ocurrido, dejando solamente dos opciones a los ingenieros e investigadores: adaptarse o perecer. Este material pretende servir de introducción a una de las plataformas hardware más utilizadas en la actualidad (los procesadores masivamente paralelos en forma de GPUs) y a uno de los lenguajes y conjunto de herramientas de mayor éxito de entre todos los diseñados para explotar la capacidad de dichos procesadores: NVIDIA Compute Unified Device Architecture (CUDA).

Índice general

Acrónimos

ALU Unidad Aritmético Lógica

ANTIC Alphanumeric Television Interface Controller

API Application Programming Interface

AGP Accelerated Graphics Port

AVX Advanced Vector Extensions

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

CUDART CUDA Runtime

DRAM Dynamic RAM

DSP Digital Signal Processor

GLSL OpenGL Shading Language

GPC Graphics Processing Cluster

GPU Graphics Processing Unit

GPGPU General-Purpose Computing on Graphics Processing Units

GUI Graphical User Interface

HBM High Bandwidth Memory

IEEE Institute of Electrical and Electronics Engineers

JIT Just-in-Time

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MMX Matrix Math eXtension

NVCC NVidia CUDA Compiler

OpenGL Open Graphics Library

PCI Peripheral Component Interconnect

PCIe PCI Express

PTX Parallel Thread eXecution

SFU Special Function Unit

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SISD Single Instruction Single Data

SLI Scalable Link Interface

SM Streaming Multiprocessor

SMI System Management Interface

SMX Kepler Streaming Multiprocessor

SMM Maxwell Streaming Multiprocessor

SSE Streaming SIMD Extensions

TPC Texture Processing Cluster

Capítulo 1

El Nacimiento del Paradigma GPGPU

Contenido

Las tarjetas gráficas o GPUs son uno de los dispositivos más extendidos y valiosos de esta era de la informática. Su creación, como respuesta a una demanda exigente por parte del mercado para conseguir gráficos en tiempo real cada vez más realistas, fue un hito que supuso la aparición de un paradigma de computación que todavía hoy sigue evolucionando a pasos agigantados. De forma concurrente, aunque algo más tardía, toda esta evolución desembocó eventualmente en una nueva revolución en el ámbito científico al reutilizar este tipo de procesadores, junto con su particular forma de cómputo, para la aceleración de otros problemas ajenos al campo de los gráficos tridimensionales. Esta revolución fue acuñada bajo el término General-Purpose Computing on Graphics Processing Units (GPGPU), más conocido actualmente como computación sobre GPUs o *GPU Computing*.

Estudiar la historia y la evolución de las tarjetas gráficas ayuda a comprender los motivos y las decisiones de diseño que desembocaron en las GPUs modernas que conocemos hoy en día así como el nacimiento de este nuevo paradigma de computación. Aunque este conocimiento no tenga una aplicación directa, sí que proporciona una comprensión más profunda de las principales características de este tipo de procesadores como por ejemplo el paralelismo masivo como forma de cómputo, el ancho de banda como prioridad en la interfaz de memoria y la ligereza y simplicidad de sus hilos como filosofía entre otros muchos atributos diferenciadores de los procesadores y del cómputo tradicional. Este entendimiento posee dos efectos colaterales de gran interés para cualquier programador de GPUs: por un lado, conocer los puntos débiles y fuertes del hardware así como los de sus patrones de computación permite ser conscientes de la forma óptima de afrontar el desarrollo de un programa para la plataforma

en cuestión, por otro lado, esta visión del pasado y su evolución otorga un cierto criterio para proyectar y, en ocasiones, definir el futuro de la computación sobre GPUs.

1.1. Hacia la Unidad de Procesamiento Gráfico

La existencia de microprocesadores con funciones gráficas integradas e incluso de chips dedicados a este propósito data desde la década de los 70. El origen de estos dispositivos disponibles para el público, aunque no de forma directa, se sitúa en las máquinas *arcade*, las cuales incorporaban procesadores especializados en ciertas funciones gráficas. Es el caso del Fujitsu MB14241 que gozó de gran popularidad en esta década y fue incorporado el máquinas de diversos desarrolladores. Algunos *arcades* destacables incluyeron su propia solución de aceleración gráfica como es el caso del Namco Galaxian o del Atari 2600. Cabe destacar también el caso de ciertos sistemas de entrenimiento para el hogar como los ordenadores de ocho bits de Atari que incorporaban el conocido Alphanumeric Television Interface Controller (ANTIC) para acelerar diversas funciones y utilidades gráficas.

A pesar de todo ello, hasta finales de la década de los 80 y principios de los 90 los gráficos por computador no tenían una importancia capital en el mercado de consumo. Fue la popularización de los sistemas operativos basados en interfaces gráficas lo que conllevó la aparición de los primeros aceleradores gráficos para el público general. Estos dispositivos primitivos ofrecían aceleración de gráficos 2D para descargar a la Central Processing Unit (CPU) de dicha responsabilidad y así mejorar tanto el aspecto visual como la usabilidad de dichos entornos de usuario. Dentro de estos aceleradores primigenios destacaron el NEC microPD7220 y el IBM 8514.

La aceleración de gráficos 2D seguiría evolucionando con la aparición de otros tantos dispositivos cuyos objetivos principales eran tanto las Graphical User Interfaces (GUIs) de los sistemas operativos que dominaban el mercado como los gráficos de los videojuegos bidimensionales de la época. Esta progresión fue reafirmada por la aparición de interfaces de programación para diversas tareas gráficas en dos dimensiones como WinG para Windows 3.X y posteriormente DirectDraw para Windows 95.

A medida que la aceleración 2D iba madurando, los gráficos tridimensionales comenzaban a cobrar cada vez más importancia. La compañía Silicon Graphics Inc. tuvo un papel pionero durante la década de los 80 ya que popularizaron el uso de gráficos 3D en una ingente cantidad de ámbitos: visualización científica, animación, efectos especiales y otras aplicaciones más específicas y cerradas para instituciones gubernamentales y militares. Su principal producto y contribución fueron estaciones de trabajo completas especializadas en todo tipo de tareas gráficas en tres dimensiones. En cuanto al público general, los primeros procesadores gráficos dedicados a polígonos en tres dimensiones fueron equipados por las máquinas arcade System 21 de Namco y Air System de Taito. Estos chips se trataban de Digital Signal Processors (DSPs) especializados en cómputos matemáticos en 3D y fueron fabricados por Texas Instruments bajo el nombre TMS320C25. Pese a estos casos exitosos, los aceleradores gráficos tridimensionales no tuvieron éxito en el mercado de consumo y dispositivos de diferentes compañías como S3 ViRGE, ATI Rage, Matrox Mystique e incluso la archiconocida 3dfx Voodoo no tuvieron el éxito esperado.

Dentro de toda esta evolución de hardware sucedió otro hito en la parte software de la mano de Silicon Graphics. Esta compañía, que tradicionalmente se dedicó a ofrecer soluciones cerradas y especializadas para renderizado de gráficos tanto 2D como 3D, liberó la interfaz de programación para su hardware: Open Graphics Library (OpenGL). La idea principal detrás de esta Application Programming Interface (API) era la de proporcionar una forma estandarizada de utilizar funcionalidades gráficas tri-

dimensionales tanto software como hardware. Gracias a OpenGL y a la alternativa DirectX de Microsoft, las aplicaciones ganaron acceso a una librería de funciones unificada para acceder a los recursos de los aceleradores gráficos, lo cual simplificó en gran medida el desarrollo y catapultó el progreso de los gráficos por computador.

1.1.1. El Pipeline Gráfico de Función Fija

Llegados a este punto, es obligatorio introducir un concepto de suma importancia: el pipeline gráfico. Este concepto, dentro del campo de los gráficos por computador, hace referencia al modelo que describe los pasos que un sistema gráfico debe realizar para renderizar una escena tridimensional en un soporte bidimensional. Este proceso consiste comúnmente en aplicar una serie de transformaciones a polígonos tridimensionales, frecuentemente triángulos que definen la superficie de los objetos en la escena, con el objetivo de determinar el color de cada uno de los píxeles de una pantalla para visualizar la escena renderizada.

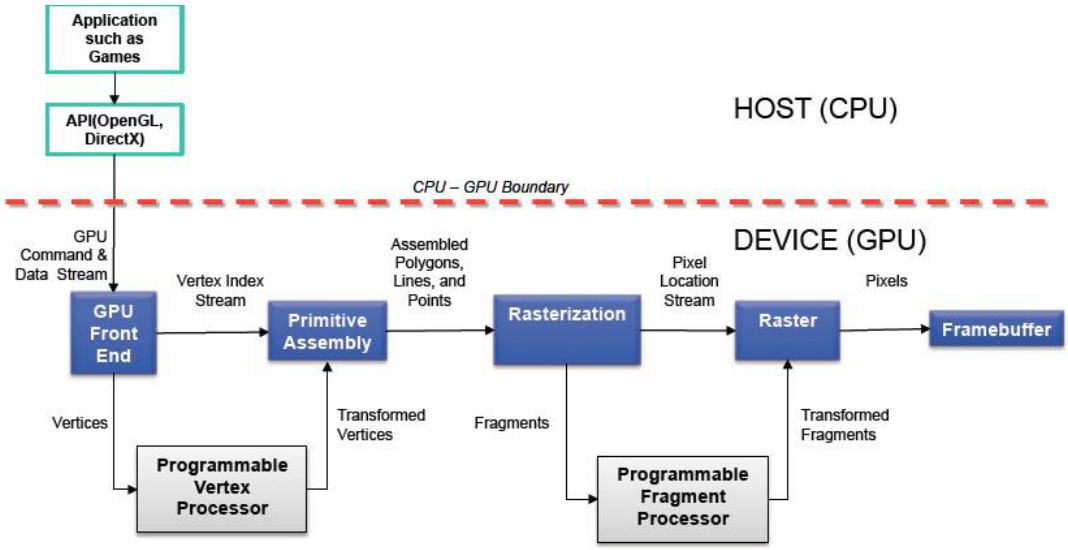

Figura 1.1: Pipeline gráfico de función fija básico en el que la API gráfica marca la frontera de separación entre el *host* y el *device* y a través de la cual se produce la comunicación y el envío de comandos para la configuración del propio pipeline y la ejecución de renderizado de gráficos.

Conviene recalcar que, al tratarse de un modelo, los detalles de su ejecución dependen en gran medida del software y el hardware que implementan sus funcionalidades. Justo en este matiz reside el valor de las APIs mencionadas anteriormente pues permitieron a las aplicaciones comunicarse de forma sencilla y estandarizada con el hardware que va a ejecutar el pipeline gráfico para enviar comandos y configurar, dentro de las limitaciones, ciertas etapas del pipeline. La Figura 1.1 muestra una versión simplificada con los componentes más importantes del pipeline dominante durante las décadas de los 80 y 90.

El pipeline gráfico básico se compone de las siguientes partes con diferentes variaciones y subcomponentes que han ido variando a lo largo del tiempo de acuerdo a las evoluciones que han ido conformando su aspecto actual:

- **Procesado de Vértices**: Esta primera etapa obtiene los datos sobre vértices proporcionados por el host a través de la interfaz y como resultado produce una serie de primitivas tridimensionales ensambladas.
  - ***Vertex Fetching/Pulling/Control***: esta primera fase recibe múltiples nombres según la literatura, su función principal es la de convertir los datos de los polígonos enviados por el host a una forma comprensible por la implementación del pipeline y almacenarlos en una memoria adecuada para las siguientes fases (*vertex cache*).
  - **Teselación**: esta fase, de escasa presencia antiguamente pero común en los gráficos actuales, se encarga de subdividir las primitivas geométricas en otras de menor tamaño y numerosas.
  - ***Vertex/Geometry Shader***: probablemente la fase más importante y conocida de esta etapa. Se encarga de transformar geométricamente los vértices los polígonos y de asignarles valores tales como colores, normales, o coordenadas de texturas.
  - **Ensamblado de Primitivas**: por último, los vértices se agrupan para formar primitivas como líneas y triángulos en forma de ecuaciones. Dicha información será utilizada en etapas posteriores para, por ejemplo, realizar interpolaciones de color o texturas.
- **Clipping y Rasterización**: Esta etapa utiliza las primitivas ensambladas para determinar si deben ser rasterizadas o no y posteriormente determinar qué fragmentos o píxeles son cubiertos por las primitivas. En otras palabras, convierte las primitivas tridimensionales en píxeles.
  - ***Clipping***: las primitivas ensambladas en la etapa anterior se encuentran en el denominado espacio *clip*. En esta fase, las primitivas son transformadas al espacio de la pantalla o a un *viewport* completo. Aquellas primitivas que sean visibles en dicho espacio son las que se seleccionarán para la rasterización.
  - ***Culling***: esta fase sirve como filtro para aquellos triángulos cuya normal indique que no están encarados hacia la cámara. Si esta fase está habilitada, los triángulos que no están encarados serán descartados puesto que en un objeto cerrado serán ocultados por los que sí estén orientados hacia la cámara.
  - **Rasterización**: en esta fase se determinan los píxeles que son cubiertos por cada uno de las primitivas, es decir, se proyecta la primitiva de un espacio 3D a uno bidimensional. Esta lista de píxeles, también denominados fragmentos, por cada primitiva se envía a la siguiente etapa.
- **Procesado de Fragmentos**: En esta etapa se aplican transformaciones a nivel de fragmento para determinar su color, transparencia y profundidad. Todas estas propiedades son resultado de, por ejemplo, aplicar modelos de iluminación y de la texturización.
- **Procesado de Framebuffer**: En esta última etapa de eliminan superficies ocultas y se mezclan los colores de los fragmentos de acuerdo a su transparencia para determinar el color final de cada píxel. Estos valores finales son escritos en el framebuffer del dispositivo de visualización.

Este modelo fue durante muchos años implementado por software y ejecutado en la CPU. Las diferentes etapas fueron progresivamente aceleradas mediante funciones integradas en los chips de los procesadores e incluso por el propio hardware dedicado de los primeros procesadores gráficos. El hito que marcaría el nacimiento de la Graphics Processing Unit (GPU) como el dispositivo que conocemos hoy en día llegaría con la completa implementación y aceleración hardware del pipeline gráfico.

1.1.2. La Unidad de Procesamiento Gráfico

El lanzamiento de la *NVIDIA GeForce 256*[2] (ver Figura 1.2) marcó un antes y un después, al ser la primera tarjeta compatible con *DirectX 7.0* e implementar en el propio hardware una gran parte de la funcionalidad del pipeline gráfico que anteriormente era implementada mediante software y ejecutada en la CPU: transformaciones e iluminación. La introducción del hardware bautizado como *Transform and Lighting Engine* desbloqueó la capacidad de producir aplicaciones con una fidelidad gráfica sin precedentes para aquél momento. Cabe destacar que fue difícil sacar partido de esta capacidad en su momento y no fue hasta pasados varios años que el soporte software para ejecutar esta función en el hardware alcanzó su madurez.

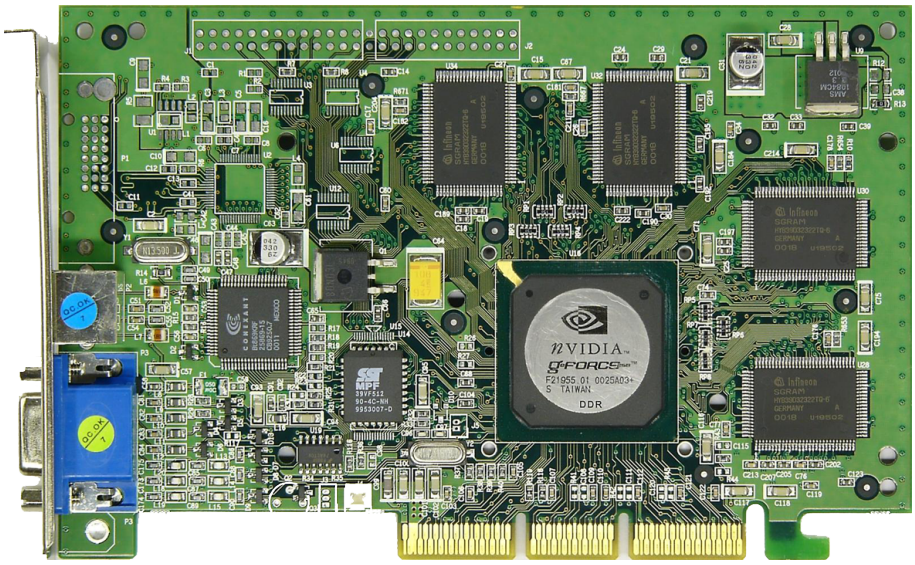

Figura 1.2: La GeForce 256 introducida por NVIDIA en agosto de 1999 empleando su chip NV10 con 32 MB de memoria de vídeo e interfaz AGP×4.

El término Graphics Processing Unit (GPU) fue introducido por primera vez por *NVIDIA* al lanzar al mercado esta primera familia de productos *GeForce* como sucesora de la tarjeta *RIVA TNT2*. Esta tarjeta fue definida por la propia compañía californiana como:

"The first Graphics Processing Unit. A single-chip processor with integrated transform, lighting, triangle setup clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second."

Así pues, desde este momento el término GPU se utilizaría para referirse a aquél dispositivo concebido para la aceleración del proceso de renderizado de datos 3D en imágenes 2D y depositarlas en un framebuffer para su posterior visualización, con todo lo que ello conlleva, es decir, la aceleración de todo tipo de operaciones gráficas: operaciones geométricas sobre vértices, ensamblaje y rasterizado de polígonos, iluminación, mapeado de texturas y reproducción/codificación de vídeo entre otros. Estas operaciones se caracterizan por ser de naturaleza intrínsecamente paralela puesto que sus elementos de trabajo pueden ser procesados de forma independiente. La GPU nace por lo tanto como un procesador capaz de explotar este paralelismo y por lo tanto obtener un rendimiento mucho mayor para este tipo de operaciones que una CPU secuencial convencional.

1.1.3. La Llegada de los Gráficos Programables

Pese a que la *GeForce* 256 (con el aclamado chip NV10) y la serie *GeForce* 2 (con chipsets NV11, NV15 y NV16) introdujeron la GPU como tal al mercado, su falta de programabilidad impidió su éxito completo ya que un procesador lo suficientemente rápido en aquella época podía llegar a ofrecer un rendimiento adecuado y además gozaba de las ventajas de un juego de instrucciones x86 completo que permitía una programabilidad y optimización total.

La llegada de la serie *GeForce 3*[3] de NVIDIA en el año 2001 supuso otra revolución en lo que respecta a tecnologías de procesamiento gráfico. En el chip NV20 no solamente todo el pipeline gráfico era ejecutado en hardware, sino que además las etapas de *vertex* y *pixel/fragment shading* eran completamente programables, siendo la primera tarjeta en cumplir con el estándar *DirectX 8.0*. El empleo de esta arquitectura, bautizada como *nFiniteFX Engine*[11] permitió a los propios desarrolladores tener un control total sobre los cómputos realizados en la GPU.

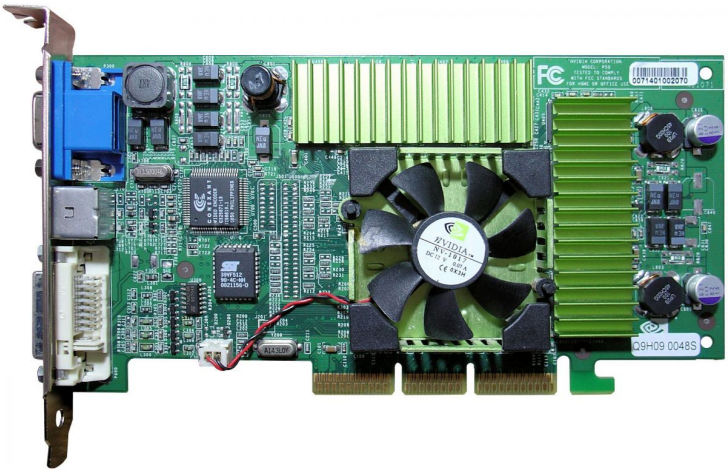

Figura 1.3: La GeForce 3 introducida por NVIDIA en febrero de 2001 con el chip NV20, 64 MB de memoria de vídeo e interfaz AGP$\times$4.

1.2. Primeros Pasos en Computación sobre GPUs

La evolución a pasos agigantados de la progamabilidad e las GPUs atrajo el interés de múltiples investigadores que vieron la oportunidad de reaprovechar la gran capacidad para realizar cómputos paralelos de dichos dispositivos con otro fin distinto al renderizado de gráficos empleando OpenGL o DirectX. El propósito era otro: resolver problemas de cómputo paralelo intensivo en ámbitos científicos.

Los primeros pasos estuvieron constreñidos por las limitaciones intrínsecas de la única manera de acceder a los recursos hardware de las tarjetas: las APIs gráficas. Para poder resolver un problema ejecutando un programa de cómputo general empleando los recursos de estos procesadores era necesario expresar dicho problema con los tipos de datos y operaciones gráficas permitidas por OpenGL o DirectX.

Así pues, estas primeras aproximaciones consistieron en expresar los datos de estos cómputos de propósito general como si de datos gráficos se trataran en forma de vértices, polígonos, colores o texturas. Los resultados producidos por la GPU, aparentemente gráficos en forma de píxeles del framebuffer, eran recuperados de sus respectivas posiciones de memoria e interpretados de forma distinta.

En definitiva, la GPU estaba siendo engañada realizando cómputos de propósito general como si se trataran de operaciones de renderizado. Gracias a la gran capacidad de cómputo que poseían las GPUs del momento para realizar operaciones aritméticas enteras en paralelo, los resultados iniciales fueron prometedores en cuanto al rendimiento y aceleración que se podía conseguir en aplicaciones sencillas. Este paradigma consistente en emplear los recursos de la GPU para realizar cómputos no relacionados con gráficos fue denominado General-Purpose Computing on Graphics Processing Units (GPGPU)

No obstante, existían múltiples limitaciones de diversa naturaleza que dificultaron en gran medida la adopción de este paradigma de computación de propósito general sobre GPUs al público desarrollador:

- La única forma de interactuar con la GPU para realizar los cómputos era mediante DirectX u OpenGL por lo que los desarroladores debían aprender y dominar sus APIs, con el esfuerzo de traducción que conllevaba modificar un programa existente para expresarlo como cómputos gráficos y engañar a la GPU.
- Esta dificultad no se limitaba únicamente a la estructura y flujo general del programa, sino que las propias operaciones que iban a ser paralelizadas debían ser escritas en lenguajes propios de gráficos para los *shaders* como Cg o OpenGL Shading Language (GLSL). Estos dos factores suponían una curva de aprendizaje demasiado pronunciada para la adopción general.
- El soporte para operaciones en en coma flotante (*float* o *double*) no estaba plenamente desarrollado por lo que muchas aplicaciones científicas directamente no podían ser ejecutadas en una GPU.
- Las operaciones de escritura de memoria poseían serias limitaciones. Los *shaders* no tenían la capacidad de escribir posiciones de memoria arbitrarias sino que debían depositar el valor de cada píxel en una posición específica del framebuffer. Aquellos programas que necesitaban patrones del tipo *scatter* (escrituras en posiciones arbitrarias) no podían por lo tanto ser ejecutados.
- No existían métodos de depuración ni de control de errores, por lo que si el programa tenía un fallo podía ocurrir un amplio abanico de situaciones: resultados incorrectos, fallos al terminar e incluso bloqueos o errores inesperados.
- Los recursos eran limitados en todos los sentidos: escasa memoria, tipos de datos limitados, y poca flexibilidad en las operaciones.

1.3. Evolución y Estancamiento de las CPUs

Desde la aparición de la CPU, su rendimiento ha seguido una progresión ascendente gracias principalmente a varios factores: el incremento de la frecuencia de reloj, mejoras en la microarquitectura y sobre todo la capacidad de integrar más transistores de dimensiones y costes cada vez más reducidos. Prueba de este hecho es la archiconocida Ley de Moore [15]. Esta ley enunciada por Gordon Moore, el cofundador de *Fairchild Semiconductors* e *Intel*, originalmente en 1965, estableció que el número de transistores integrados en un microprocesador se duplicaba cada año. La Figura 1.4 muestra una representación de esta ley. El propio Moore en 1975 volvió a enunciar la ley para acomodarla al progreso de la siguiente década estableciendo que el período necesario para duplicar la cantidad de transistores integrados sería de dos años. Posteriormente, el ejecutivo de *Intel*, David House, predijo una versión más conocida de la ley según la cual el rendimiento en sí de los microprocesadores se duplicaría cada 18 meses debido a las mejoras continuas en cuanto a capacidad de integración de transistores en el microchip y al aumento de la frecuencia de reloj.

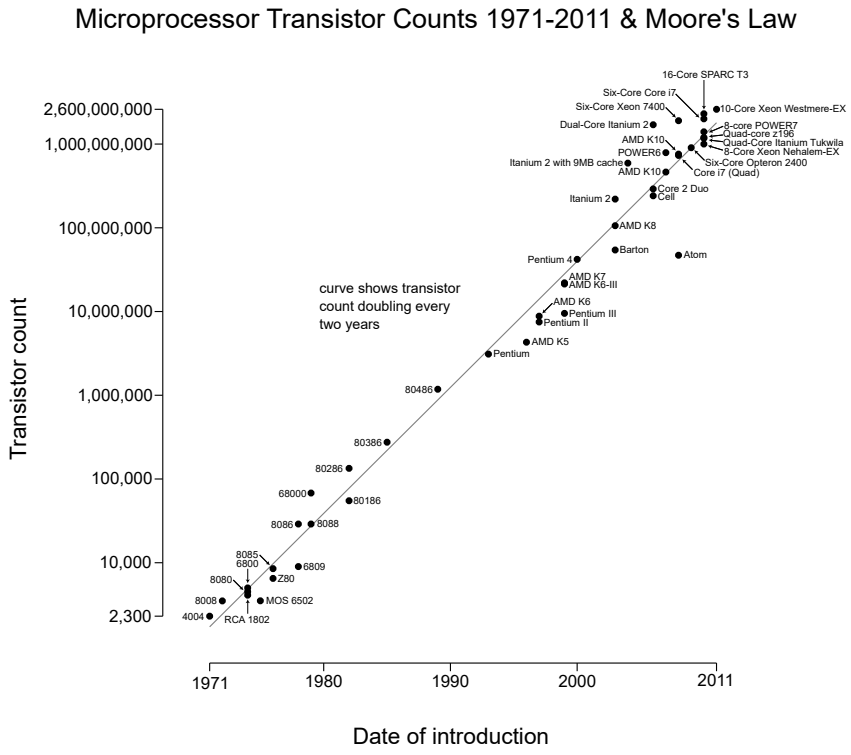

Figura 1.4: Número de transistores integrados en diferentes CPUs frente a la fecha de introducción de las mismas. La escala logarítimca en el eje vertical refleja el crecimiento exponencial de la capacidad de integración.

Sin embargo, aunque estos métodos de incremento de rendimiento en la CPU han funcionado durante una gran cantidad de años, diversos factores han provocado una desaceleración en el aparentemente imaparable crecimiento exponencial del rendimiento de los procesadores [17].

Por un lado, el consumo energético y el calor a disipar producto del aumento de la frecuencia de reloj incrementa exponencialmente por lo que los procesadores actuales se encuentran en un máximo alrededor de los 4 GHz como límite aceptable para el consumo de electricidad y para la disipación de calor de las soluciones factibles para un uso prolongado (aire y líquida).

Por otro lado, nos acercamos, cada vez de forma más costosa, al límite físico de tamaño la tecnología actual en la que se basan los transistores. La fabricación de transistores de tamaño cada vez más reducido conlleva numerosos problemas que impiden el correcto funcionamiento de los mismos. Entre ellos destaca el conocido como efecto túnel (ver Figura 1.5) que se experimenta en las puertas lógicas de aquellos transistores con un proceso de fabricación inferior a siete nanómetros. Superar esta barrera requiere una serie de avances tecnológicos todavía no conseguidos.

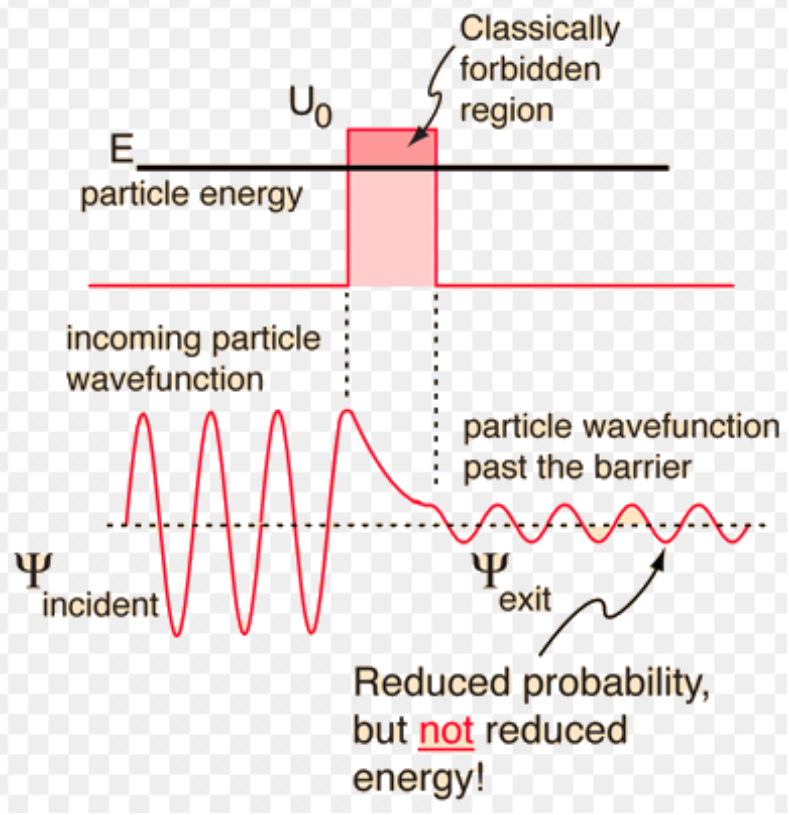

Figura 1.5: Ilustración del fenómeno cuántico conocido como efecto túnel, según el cual una partícula puede atravesar, con una cierta probabilidad reducida pero con la misma energía, una barrera cuya energía es mayor que la de la propia partícula y que por lo tanto no sería capaz de superar de acuerdo a la Física convencional.

Cabe destacar que los supercomputadores también se han nutrido de los incrementos en la frecuencia de reloj y en la integración de transistores para mejorar su rendimiento, pero al verse acorralados por estas limitaciones se recurrió a otra vía para aumentar el rendimiento: el incremento en el número de procesadores trabajando en paralelo y en perfecta sincronización. Esta alternativa dio lugar a una revolución propia en los procesadores de consumo que comenzaron a incorporar más núcleos de procesamiento para poder seguir mejorando el rendimiento sin necesidad de recurrir a la mejora en la capacidad de integración y al incrementod de la frecuencia de reloj y por lo tanto sin sufrir problemas significativos de consumo energético o calor.

Esta evolución de la CPU entronca con los principios de diseño arquitecturales de la GPU y la apuesta por el paralelismo como potenciador del rendimiento. Sin embargo, la CPU posee un propósito diferente a la GPU. Pese a ser capaz de realizar cómputos en paralelo, los procesadores convencionales deben seguir especializados en completar tareas secuenciales de la forma más rápida posible, es decir, poca latencia. Las tarjetas gráficas atacan un nicho propio en el cual los cómputos pueden ser ejecutados de forma masivamente paralela y no reducida a unas pocas unidades de trabajo independientes. Una gran cantidad de problemas de cómputo poseen una naturaleza que se adecúa o puede ser expresada para aprovechar todos los recursos que ofrece el paralelismo masivo de una tarjeta gráfica, consiguiendo así un rendimiento imposible de obtener en una CPU convencional. Es por este motivo que, pese a la evolución de las CPUs, la computación de propósito general sobre GPUs siguió capturando el interés de ingenieros y científicos hasta conseguir despegar definitivamente.

1.4. El Ascenso de la Computación sobre GPUs

Como respuesta a las limitaciones y a la exigente curva de aprendizaje impuesta por las APIs gráficas, *NVIDIA* presentó una nueva tarjeta gráfica revolucionaria y que daría el pistoletazo de salida a la madurez del campo de la computación sobre GPUs: la *GeForce* 8800 GTX [4]. Esta tarjeta sería la primera GPU en ser diseñada con la arquitectura *NVIDIA* Compute Unified Device Architecture (CUDA).

Desde el punto de vista del hardware, la arquitectura CUDA solventó las principales limitaciones hasta la fecha:

- Anteriormente, los recursos físicos de la GPU estaban particionados principalmente en *shaders* de vértices y de píxeles o fragmentos. Con la introducción de CUDA, se incluyeron los llamados *shaders* unificados. De esta forma, todas las unidades computacionales de la tarjeta pueden utilizarse para cualquier tarea programable, consiguiendo un balanceo de carga mucho más efectivo y evitando las dificultades impuestas por la distinción anterior.

- Todas las Unidad Aritmético Lógicas (ALUs) de estas unidades de cómputo fueron construidas para soportar los estándares del Institute of Electrical and Electronics Engineers (IEEE) para operaciones aritméticas en punto flotante de precisión simple (*float*), por lo que se abrió la puerta a un amplio rango de aplicaciones científicas.

- Por otra parte, los patrones de acceso y escritura a memoria se flexibilizaron de forma que las unidades de ejecución fueran capaces de realizar lecturas y escrituras arbitrarias e incluso acceder a una memoria caché de mayor velocidad llamada memoria compartida. Esto provocó que, por ejemplo, aplicaciones que necesitaban el uso de patrones como *scatter* pudieran ser implementadas y ejecutadas en una GPU.

Figura 1.6: La GeForce 8800GTX introducida por NVIDIA en noviembre de 2006 con el chip G80, 768 MB de memoria de vídeo e interfaz PCIe×16.

Sin embargo, el cambio más notable no ocurrió solamente en el hardware. De nada hubieran servido todos los avances antes mencionados si las dificultades para los desarrolladores hubieran continuado en forma de la necesitad de utilizar OpenGL o DirectX para acceder a los recursos de la GPU y los lenguajes de *shading* para expresar los cómputos. En este sentido, la arquitectura CUDA va más allá de un mero avance de hardware, es un ecosistema completo que incluye un lenguaje propio para programar las GPUs compatibles (CUDA C/C++), un compilador de dicho lenguaje a instrucciones comprensibles por la GPU (NVCC) y un driver especializado para explotar toda la capacidad de cómputo de la tarjeta.

Gracias a la eliminación de las barreras hardware y software, se dio rienda suelta a un paradigma que ha cobrado tanta importancia en la actualidad hasta suponer, entre otras cosas, una revolución total en la forma de entender la computación [16].

Capítulo 2

Problemas 1: Cálculo de Rendimientos

Contenido

El rendimiento, o mejor dicho la mejora del mismo, es un factor de gran importancia al desarrollar una aplicación paralela a la hora de compararla bien con su contrapartida secuencial o bien con versiones paralelas previas sin optimizar.

Desde el punto de vista del usuario, el rendimiento puede definirise de diversas formas: los videojuegos funcionan más fluidos, las aplicaciones gráficas muestran mayor interactividad o incluso calidad, etc. Desde el punto de vista científico, como podría ser un supercomputador, podríamos hablar de la cantidad de trabajo realizado en un tiempo en un tiempo determinado.

Como apunte, cabe destacar que la métrica más utilizada para medir la capacidad de cálculo son los llamados FLOPS o Floating-Point Operations per Second. Generalmente se utilizan los prefijos adecuados para simplificar la medición del rendimiento:

- megaFLOPS (MFLOPS) = FLOPS
- gigaFLOPS (GFLOPS) = FLOPS
- teraFLOPS (TFLOPS) = FLOPS
- petaFLOPS (PFLOPS) = FLOPS
- exaFLOPS (EFLOPS) = FLOPS

Actualmente, el supercomputador más potente del mundo (Summit) es capaz de alcanzar un ratio de aproximadamente 122 PFLOPS con más de dos millones de núcleos y un consumo de más de 8000 kW. La lista TOP500 (`https://www.top500.org`) recopila dos veces cada año (junio y noviembre) los 500 supercomputadores más potentes del mundo junto con sus estadísticas.

2.1. Rendimiento Teórico y Ley de Amdahl

Cuantificar el rendimiento teórico que podemos obtener de acelerar una aplicación es útil en muchos sentidos. Por un lado nos puede permitir decidir si merece la pena invertir costes de desarrollo en esa aceleración o no. Por otro lado nos permite establecer métricas para determinar la evolución de nuestras optimizaciones. Para cuantificar ese rendimiento o aceleración teórica se utiliza la Ley de Amdahl.

Según esta Ley, el factor máximo de mejora que podemos lograr para nuestra aplicación depende tanto del porcentaje de tiempo que consuma la porción acelerable como el propio factor de aceleración que podamos conseguir en dicha fracción. Este hecho limita en gran medida el rendimiento que podemos conseguir tal y como se muestra en la Figura 2.1.

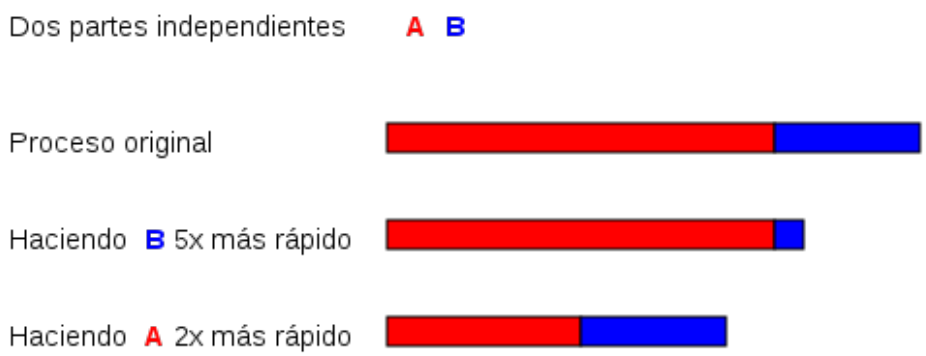

Figura 2.1: Ilustración del efecto de la Ley de Amdahl, según la cual la mejora en el rendimiento depende tanto del porcentaje que ocupa la fracción acelerada como de la aceleración que podemos aplicar sobre dicha fracción.

Desde el punto de vista de la mejora en rendimiento obtenida por una aplicación paralela sobre una GPU respecto a una secuencial sobre la CPU dependemos de diversos factores. Por un lado, el factor de aceleración logrado por el kernel depende de si se expone suficiente paralelismo, de si el flujo de datos y el flujo de control se adaptan al modelo de cómputo de la GPU, de la diferencia de adecuación del código para la CPU y la GPU y de optimizaciones adicionales como los patrones de acceso a memoria. Por otro lado, tal y como hemos destacado anteriormente, la aceleración global va a depender en gran medida del peso que tenga dicha fracción de código sin acelerar en el tiempo de ejecución global de la aplicación.

Si formalizamos todas estas afirmaciones, la aceleración o speedup obtenido según Amdahl se puede cuantificar tal y como se muestra en la Ecuación 2.1.

$$S = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}} \tag{2.1}$$

2.2. Problema 1

Se pretende acelerar un código proporcionado por nuestro jefe con el objetivo de obtener mayor rendimiento y menor tiempo de ejecución. Antes de empezar a desarrollar la nueva aplicación paralelizada se hace un estudio sobre las partes del código que son paralelizables y las diferencias en rendimiento según la tarjeta gráfica empleada tal y como se muestra en la Figura 2.2.

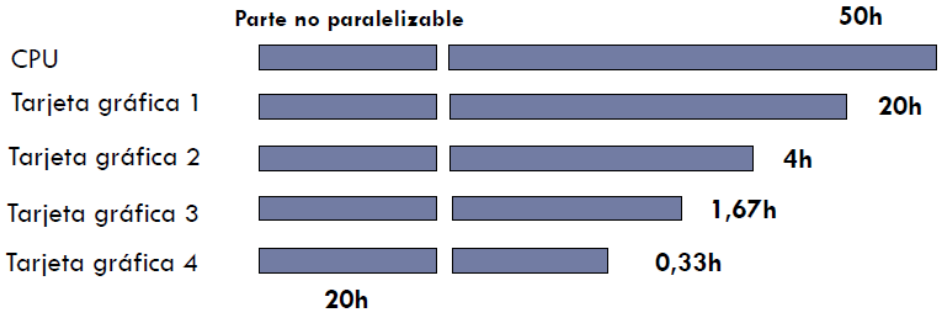

Figura 2.2: Estudio de paralelización con distintas tarjetas gráficas.

Sabiendo los resultados de dicho estudio, completar la siguiente tabla:

Dispositivo	Horas código paralelizable	Aceleración	Horas totales	Aceleración global
CPU	50			
Tarjeta gráfica 1	20			
Tarjeta gráfica 2	4			
Tarjeta gráfica 3	1.67			
Tarjeta gráfica 4	0.33			

2.3. Problema 2

Suponed que estamos considerando una mejora que se ejecute diez veces más rápida que la implementación sobre la CPU, pero solo es utilizable el 40 % del tiempo total. ¿Cúal es la aceleración global lograda al incorporar la mejora?

2.4. Problema 3

Suponer que una cache es 5 veces más rápida que la memoria principal y supongamos que la cache puede ser utilizada el 90 % del tiempo. ¿Qué aumento de velocidad se logrará al utilizar la cache?

2.5. Problema 4

Supongamos que se está considerando mejorar un computador añadiéndole una tarjeta gráfica como dispositivo auxiliar de cómputo. Cuando se ejecuta el algoritmo sobre este dispositivo acelerador, se ejecuta 20 veces más rápido que sobre la CPU. Llamamos porcentaje de tiempo que puede emplearse la GPU porcentaje de cómputo GPGPU.

- Dibujad un gráfico donde se muestre la aceleración como porcentaje del cálculo realizado en la GPU. El eje x debe representar el porcentaje de cómputo GPGPU y el eje y la aceleración global.
- ¿Qué porcentaje de cómputo GPGPU se necesita para conseguir una aceleración global de 2?
- ¿Qué porcentaje de cómputo GPGPU se necesita para conseguir la mitad de la aceleración máxima alcanzable utilizando la GPU?

2.6. Problema 5

La utilización de la tarjeta gráfica como GPGPU mejora en un factor de 5 el procesamiento de números en coma flotante. El tiempo de ejecución de cierto programa es de 1 minuto con la GPU y de 2.5 minutos sin ella.

- Calculad el porcentaje de tiempo de ejecución, sin la GPU instalada, que el programa invierte en realizar operaciones en coma flotante.
- Calcula el tiempo de ejecución del programa, sin la GPU instalada, que el programa invierte en realizar las operaciones en coma flotante.
- Calcula el tiempo de ejecución del programa, con la GPU instalada, para realizar las operaciones en coma flotante.
- Calcula el tiempo de ejecución del programa para realizar las operaciones enteras.

2.7. Falsas Creencias sobre el Rendimiento

- El consumo es gratis, los transistores son caros: La realidad actualmente es que el consumo energético es primordial mientras que el proceso de fabricación de transistores se abarata cada vez más. El problema reside en la cantidad de energía necesaria y el calor generado por todos los transistores que podemos empaquetar en un mismo chip.

- Apostar por el paralelismo a nivel de instrucción: Actualmente el paralelismo a nivel de instrucción ocupa mucha circuitería y su rendimiento está saturado. Para poder desbloquear más rendimiento se necesitan niveles de paralelismo más elevados.
- Las operaciones aritméticas son lentas, mientras que los accesos a memoria son rápidos: Nada más lejos de la realidad, las operaciones aritméticas requieren escasos ciclos mientras que un acceso a memoria puede costar una gran cantidad de ciclos y latencia debido a los diferentes buses de datos.
- Se cumple la Ley de Moore y se seguirá cumpliendo: Hemos alcanzado barreras físicas que nos impiden seguir progresando en esa dirección (consumo, calor, memoria).

Capítulo 3

Introducción a CUDA

Contenido

Compute Unified Device Architecture (CUDA) como comentamos anteriormente va más allá de una arquitectura hardware para GPUs. Se trata de un ecosistema compuesto por dicha arquitectura hardware unificada, un driver optimizado para la misma, un modelo de programación propio, una serie de extensiones para expresar programas en dicho modelo utilizando lenguajes de programación existentes, un compilador capaz de transformar programas escritos en esos lenguajes en una representación comprensible para la GPU y una serie de herramientas de desarrollo y depuración enfocadas a la programación masivamente paralela. Debido a la magnitud del concepto CUDA, es conveniente explorarlo desde dos puntos de vista para comprender toda su extensión: la arquitectura hardware y la arquitectura software.

3.1. Arquitectura Hardware

En lo que respecta a la arquitectura hardware, hay tres aspectos que caracterizan a las GPUs CUDA diferenciándolas claramente de otro tipo de componentes. En primer lugar, su situación física en el propio computador y su método de interconexión. Por otro lado, la distribución de su circuitería y microarquitectura para dar soporte a su modelo de procesamiento particular Single Instruction Multiple Thread (SIMT). Por último, en lo que respecta a CUDA particularmente más que a la GPU en general, la existencia de una serie de procesadores unificados con una serie de características denominados SM.

3.1.1. Situación Física e Interconexión

Desde un punto de vista físico en un computador de consumo, la GPU es una tarjeta de expansión que actualmente se conecta al resto del sistema mediante puertos PCIe presentes en la placa base (en el pasado mediante los obsoletos AGP o los propios Peripheral Component Interconnect (PCI)). Cabe destacar el elemento de probablemente mayor importancia en un computador: el *chipset*, cuya función es la de gestionar las conexiones de la CPU con todo el resto de componentes y orquestrar su correcto funcionamiento. En los sistemas actuales, este *chipset* se divide en dos componentes: el conocido como puente sur (o más comúnmente *southbridge*) y el puente norte (*northbridge*). El primero de ellos conecta la mayoría de los periféricos al sistema mientras que el segundo gestiona los buses de gráficos (actualmente PCIe) y la comunicación con la memoria principal de la CPU (mediante el *front-side bus*). La Figura 3.1 muestra un esquema de un sistema CPU-GPU conectada mediante un puerto PCIe.

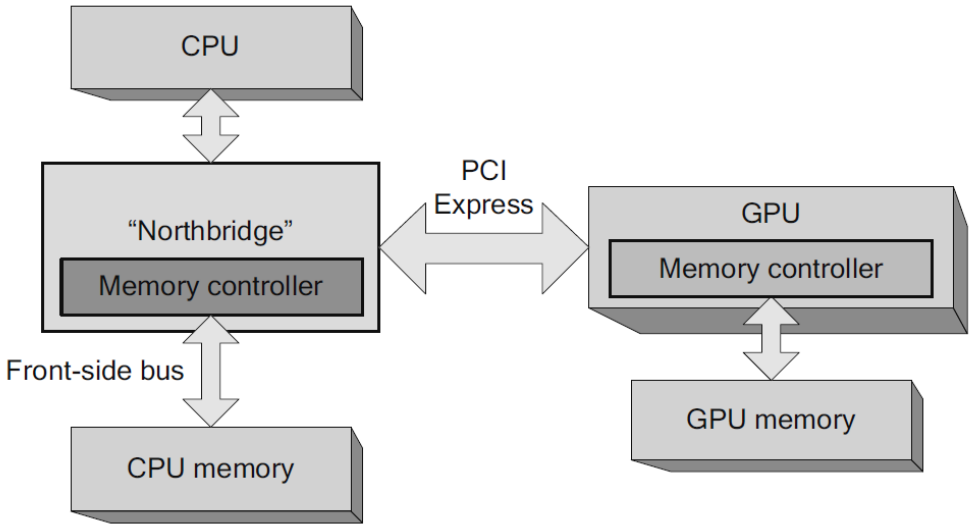

Figura 3.1: Arquitectura CPU-GPU típica en la que el *northbridge* interconecta la CPU con su memoria principal y el bus PCIe en el cual se encuentra la GPU con su propio espacio de memoria. El southbridge del chipset ha sido omitido por simplicidad.

Existen configuraciones alternativas en los computadores de consumo como pueden ser las CPUs integradas en la placa base de forma que el *northbridge* las alberga y comparten tanto controlador de memoria como la propia memoria física con la CPU. Este tipo de configuración se muestra en la Figura 3.2.

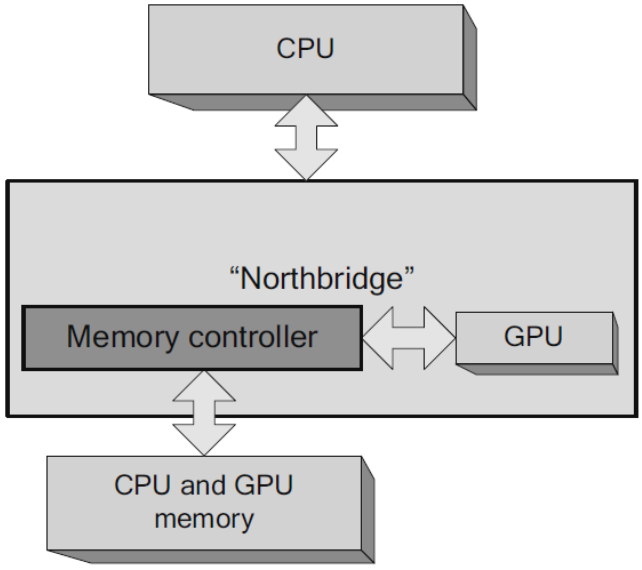

Figura 3.2: Arquitectura CPU-GPU integrada en la que la GPU se encuentra físicamente en el chipset de la placa base compartiendo el espacio de memoria principal físico con la CPU.

Otra de las configuraciones alternativas, aunque quizás menos común, es la de múltiples GPUs en diferentes puertos PCIes e incluso múltiples GPUs intercomunicadas, bien en la misma tarjeta o bien empleando un puente para evitar la comunicación por el propio puerto PCIe. Una alternativa a este último caso consiste en emplear las tecnologías Scalable Link Interface (SLI) en tarjetas NVIDIA o *CrossFire* en placas AMD para interconectar múltiples GPUs en diferentes puertos para trabajar al unísono sobre problemas particulares y de esta forma conseguir escalar el rendimiento del sistema. Es importante destacar que este tipo de configuraciones implican sobrecostes adicionales en términos de comunicación y sincronización que pueden deteriorar el rendimiento o en la mayoría de los casos impedir alcanzar el que sería el máximo teórico según una escalabilidad lineal. En cualquier caso, por lo general cada GPU posee su propio espacio y controlador de memoria físico, pero pueden emplear el puente de menor latencia y mayor ancho de banda (ya sea integrado en el chip o en forma de conector SLI o *CrossFire*) para evitar el puerto PCIe a la hora de realizar comunicaciones entre ellas. Este hecho parece trivial, pero atendiendo a los diagramas podemos observar que cualquier comunicación a través del puerto PCIe debe pasar por el *northbridge*, lo cual supone un incremento de latencia y además la limitación de ancho de banda que posea el bus. Un ejemplo de configuración multi-GPU se muestra en la Figura 3.3.

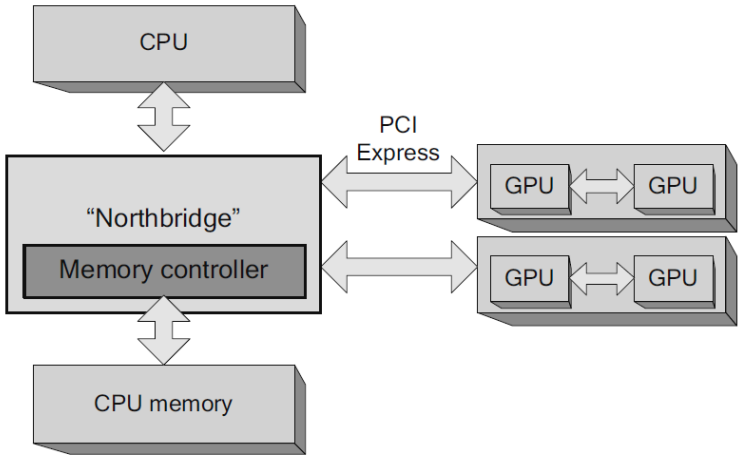

Figura 3.3: Arquitectura CPU-multi-GPU en la que se disponen de varias tarjetas gráficas conectadas por diferentes puertos PCIe. Estas tarjetas a su vez poseen más de una CPU en su propio chip.

3.1.2. Single Instruction Multiple Threads (SIMT)

Las GPUs son procesadores especializados en aprovechar el paralelismo existente en ciertos problemas, es decir, computar de forma que varios cálculos se realizan de forma simultánea (generalmente priorizando la cantidad de trabajo producido o *throughput* sobre la latencia). Desde un punto de vista arquitectural, conviene introducir la Taxonomía de Flynn para clasificar a este tipo de procesadores. En 1972, Michael Flynn estableció una clasificación para arquitecturas de computadores en cuatro categorías[13][12]: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD) y Multiple Instruction Multiple Data (MIMD). Atendiendo a esta taxonomía (ver Figura 3.4), las GPUs se encuadran dentro de la categoría Single Instruction Multiple Thread (SIMT), resultante de combinar SIMD con procesadores multihilo.

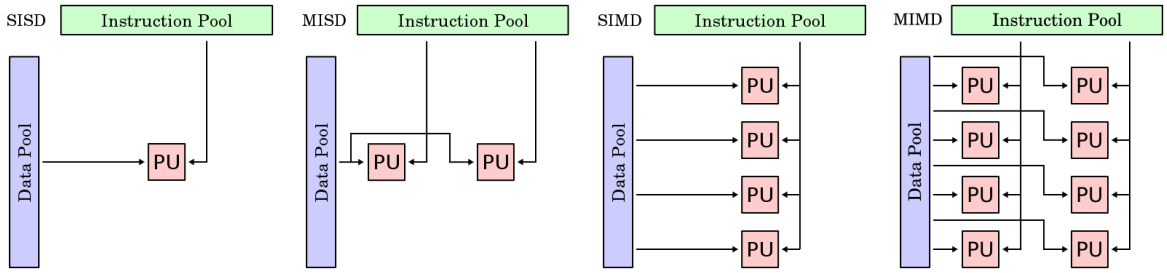

Figura 3.4: Arquitecturas de la taxonomía de Flynn: SISD, SIMD, MISD y MIMD.

Este modelo de ejecución fue introducido en el chip G80 antes mencionado y fue planteado como una adición a la taxonomía pero no está oficialmente reconocido. En esta arquitectura, un procesador ejecuta múltiples hilos de forma concurrente (los cuales se agrupan para ser ejecutados físicamente en paralelo) sobre diferentes datos.

La arquitectura de una CPU tradicional se enmarca dentro de SISD para el caso de los procesadores monolíticos completamente secuenciales y en SIMD para aquellos que incluyen múltiples núcleos de procesamiento o extensiones vectoriales en su juego de instrucciones para soportar este tipo de modelo como por ejemplo Matrix Math eXtension (MMX), Streaming SIMD Extensions (SSE) o Advanced Vector Extensions (AVX). Esta diferencia a nivel de arquitectura y de modelo de ejecución implica la existencia de diferencias notables en el hardware de ambos tipos de dispositivos, principalmente en lo referente a la distribución de los recursos para los distintos componentes (ver Figura 3.5):

- Cachés grandes (CPU) / cachés reducidos (GPU).
- Unidad de control compleja con funcionalidades como predicción de saltos o adelantamiento de datos (CPU) / unidad de control simple (GPU).
- ALUs complejas y poco numerosas con escasa latencia (CPU) / un gran número de ALUs simples y centradas en la cantidad de trabajo (GPU).
- En general: latencia (CPU) / cantidad de trabajo (GPU).

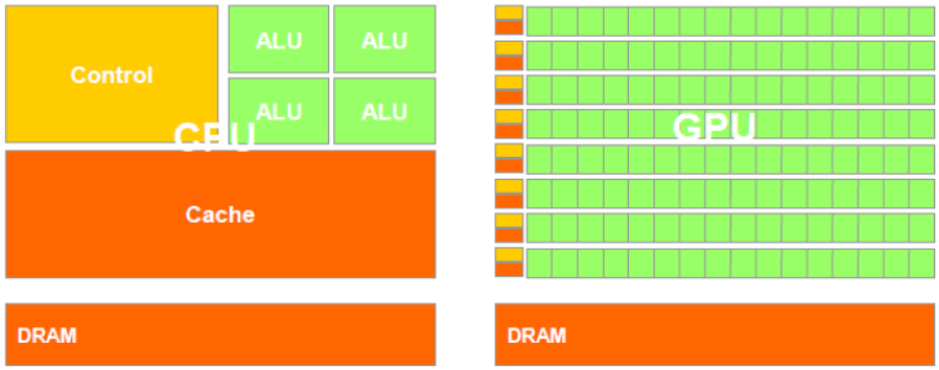

Figura 3.5: Ilustración gráfica de las diferencias entre los recursos hardware de una CPU y una GPU.

En definitiva, estas diferencias hardware consecuencia de la adopción de la arquitectura SIMT hacen que el modelo de ejecución en las GPUs se caracterice por poseer una latencia relativamente elevada (principalmente a causa de las costosas operaciones de memoria como veremos en capítulos posteriores y de la menor potencia bruta de los núcleos de procesamiento) que debe ser camuflada o compensada por la capacidad mantener una alta ocupación de los recursos con cómputos (gran cantidad de trabajo paralelo) y por la facilidad de cambiar de contexto los hilos dada la sencillez de la lógica de la unidades de control (asegurando así que los hilos a la espera de operaciones de memoria puedan ser fácilmente sustituidos por aquellos que están listos para realizar cómputos con escasa penalización temporal).

3.1.3. Streaming Multiprocessors (SMs)

Si bajamos un nivel más en el hardware, podemos descomponer una GPU de arquitectura CUDA en las siguientes partes de forma simplificada:

- **Una interfaz que conecta la GPU al bus PCIe.** Esta interfaz se encarga de leer los comandos emitidos por el procesador y enviarlos al hardware apropiado así como de proveer la lógica necesaria para realizar operaciones de sincronización CPU-GPU y GPU-GPU.
- ***Copy engines***. Estos componentes se encargan de realizan transferencias de memoria entre el *host* y el *device* de forma asíncrona al cómputo. Los primeros modelos no disponían de ninguna, los actuales disponen de hasta dos dado que cada uno puede saturar una dirección del bus PCIe.
- **Una interfaz de memoria Dynamic RAM (DRAM).** Encargada de conectar la GPU a su memoria interna a través de una jerarquía de cachés.
- **Cierto número de Texture Processing Clusters (TPCs) o Graphics Processing Clusters (GPCs)** dependiendo de la generación. Estos *clusters* contienen cachés y una cierta cantidad de la unidad computacional más importante de una GPU con arquitectura CUDA: los Streaming Multiprocessors (SMs).

Los SMs son, en lo tocante al hardware, las unidades de cómputo de la arquitectura CUDA. También son conocidos en las generaciones más recientes como Kepler Streaming Multiprocessor (SMX) en la generación Kepler o comoMaxwell Streaming Multiprocessor (SMM) en Maxwell. Actualmente se agrupan dentro de GPCs en un número que suele superar los dos SMs por GPC. Por ejemplo, el chip GM204 de la generación Maxwell contiene cuatro GPCs con cuatro SMMs cada uno, 16 SMMs en total. La Figura 3.6 muestra un SMX de la generación Kepler, con 192 núcleos CUDA. En general, cada SM/SMX/SMM está compuesto por los siguientes componentes:

- **Núcleos CUDA**, unidades de ejecución capaces de llevar a cabo operaciones aritméticas enteras o de punto flotante.
- **Unidades de funciones especiales** (conocidas como Special Function Units (SFUs)) para computar aproximaciones de raíces cuadradas, funciones trigonométricas o exponenciales/logarítmicas.
- ***Warp schedulers*** para distribuir los distintos *warps* a las unidades de ejecución.
- **Memoria caché de constantes**.
- **Memoria compartida para todos los hilos**.
- **Una archivo de registros**.

En capítulos posteriores abordaremos los diferentes conceptos presentados a lo largo de la descripción de la arquitectura (como los *warps* o las memorias), así como su influencia en el software y en el modelo de programación.

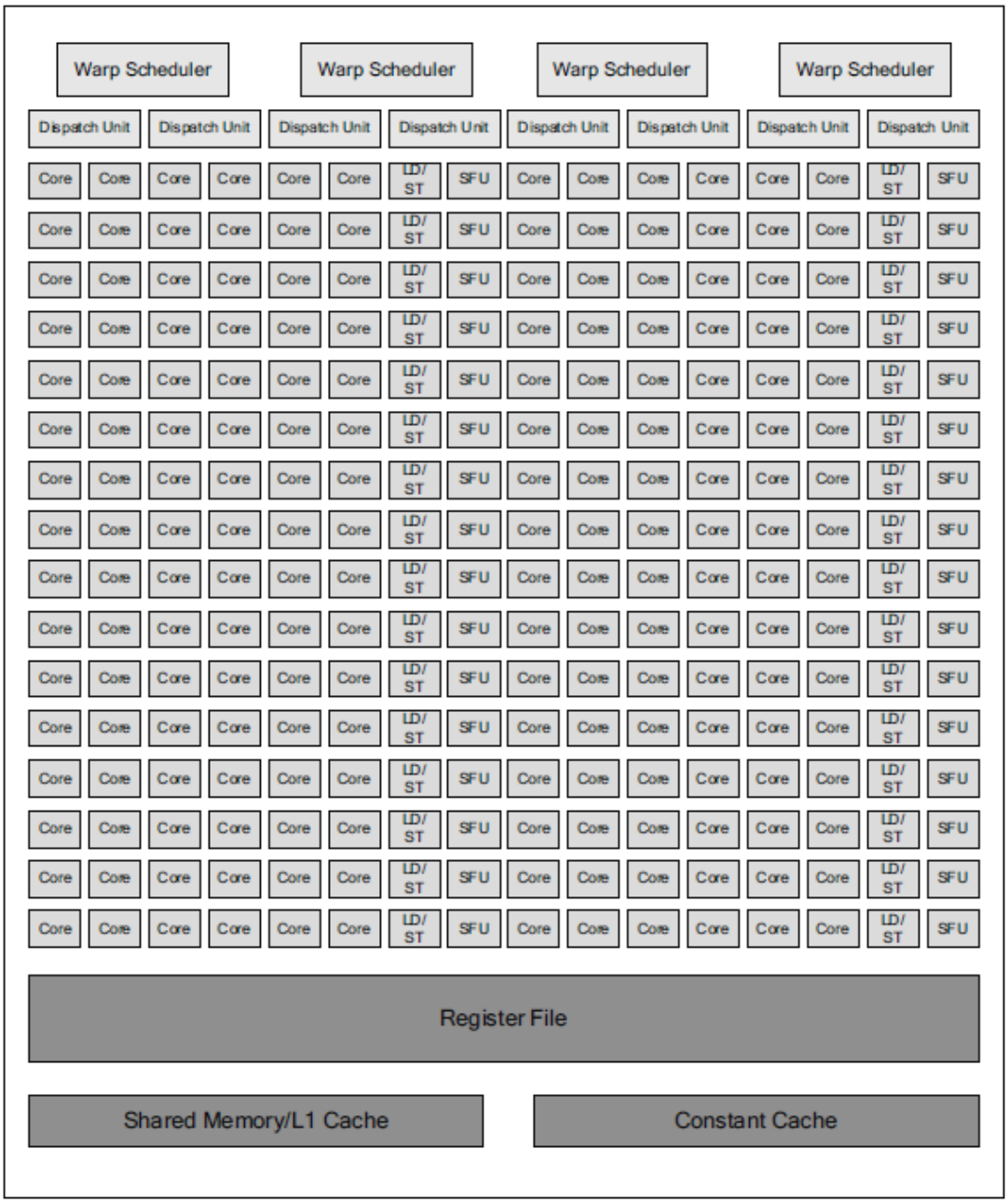

Figura 3.6: SMX de la arquitectura Kepler con 192 núcleos.

3.2. Arquitectura Software

Desde el punto de vista del software, como ya comentamos con anterioridad, CUDA es un ecosistema compuesto de una serie de capas que se sitúan en todos los niveles existentes entre el hardware y las propias aplicaciones o software construido por los desarrolladores. Por otro lado, CUDA también se caracteriza por especificar un lenguaje intermedio propio y un compilador de lenguajes de alto nivel con extensiones CUDA a dicho lenguaje. A alto nivel, dichas extensiones hacen uso de un conjunto de abstracciones que permiten expresar un programa siguiendo el modelo y el flujo de ejecución de CUDA.

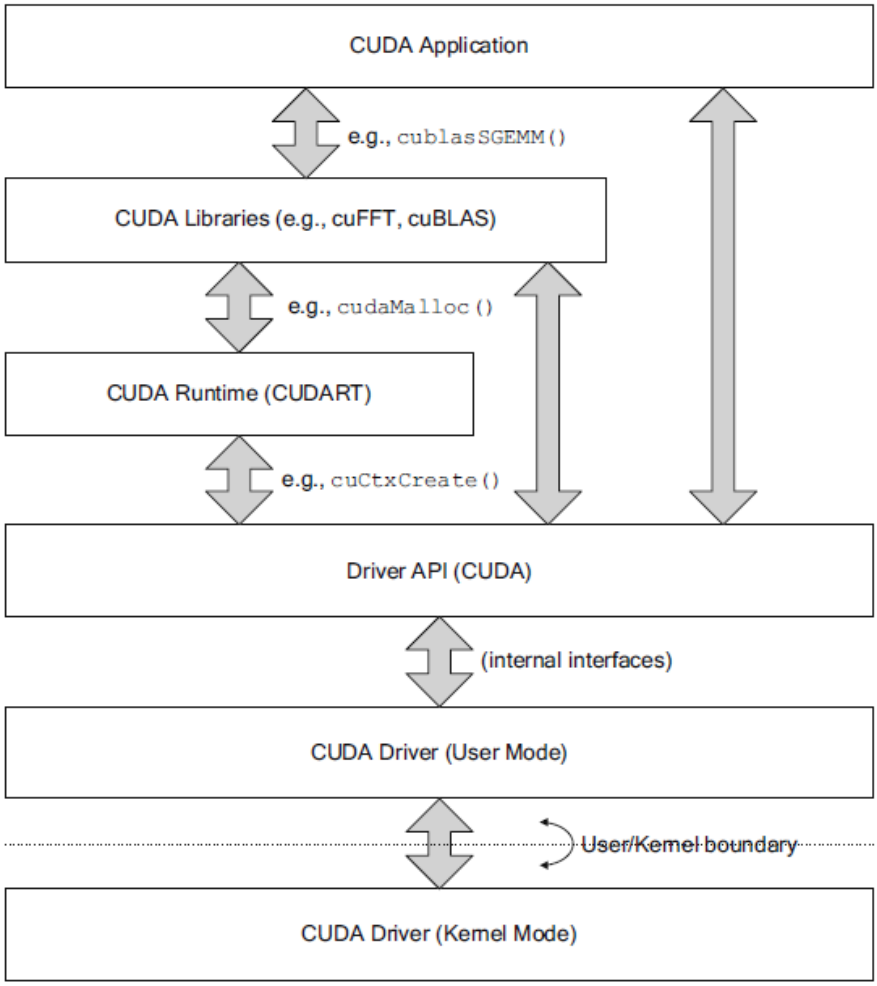

Figura 3.7: *Stack* de CUDA.

3.2.1. Capas

Desde el punto de vista del software, CUDA se compone de diversas partes que forman una pila o *stack*. Cada una de ellas aporta una funcionalidad concreta a un nivel determinado tal y como se muestra en la Figura 3.7. De forma resumida, en la parte más baja de la pila podemos encontrar el *driver* en sus dos modos (*kernel* y usuario). Este driver además expone una API, la cual es utilizada por los componentes superiores de la pila. Una aplicación CUDA, a alto nivel, puede recurrir directamente a la API de este driver para una flexibilidad y control total, puede utilizar la API CUDA Runtime (CUDART) que simplifica el acceso a la proia API del driver, o bien puede recurrir a librerías en CUDA ya desarrollada que encapsulan clases y funciones para distintos problemas y que simplifican en gran medida el desarrollo de programas.

3.2.2. Compilación (NVCC y PTX)

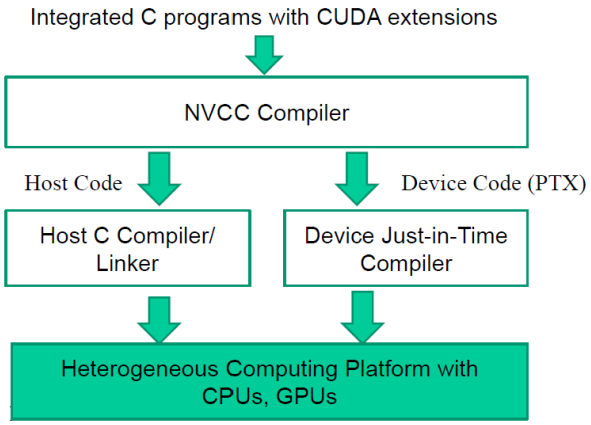

Figura 3.8: Flujo de compilación simplificado de NVCC en el que el código *host*y *device* son separados y compilados independientemente. La compilación del código GPU usualmente genera una representación intermedia PTX que es embebida en el código *host* y posteriormente interpretada por el driver de CUDA para generar microcódigo ejecutable por la GPU.

A nivel de código, un programa CUDA tiene dos partes claramente diferenciadas y a la vez entremezcladas. Independientemente del lenguaje elegido con soporte para extensiones CUDA, por una parte tenemos código *host*, el cual se ejecuta en la CPU y contiene instrucciones típicas del lenguaje además de comandos de la API de CUDA. Por otra parte, tenemos el código *device*, aquél que se ejecuta en la GPU. A la hora de compilar estos programas, CUDA emplea su propio compilador denominado NVidia CUDA Compiler (NVCC). El funcionamiento de este compilador, mostrado en la 3.8 de forma simplificada, consiste en separar aquellas porciones de código que son exclusivas del *host* y las proias del *device*. El código host es compilado y enlazado por un compilador de C/C++ tradicional presente en el sistema.

El código GPU es compilado por el propio NVCC en una representación intermedia denominada Parallel Thread eXecution (PTX). Esta representación intermedia es vital para que las aplicaciones CUDA sean independientes de la versión utilizada ya que dicho código es luego traducido a microcódigo ejecutable por la gráfica mediante el driver instalado en el sistema para la GPU específica sobre la cual se vaya a ejecutar. Esta traducción tanto a PTX (`.ptx` p `.fatbin`) como a microcódigo (`.cubin`) puede hacerse online Just-in-Time (JIT) u *offline*. El código GPU es entonces embebido en el programa final junto con el código CPU para ser ejecutado en una plataforma heterogénea que contenga tanto CPUs como GPUs.

3.2.3. Conceptos básicos

Situándonos la nivel de la aplicación, nos encontramos con una serie de conceptos abstractos propios del ecosistema CUDA que merece la pena destacar en esta introducción.

Dispositivo

En el argot CUDA, el concepto dispositivo (comúnmente llamado *device* en la literatura) hace referencia a cada GPU física existente en el sistema. En las Secciones 3.3.3 y 3.3.4 expondremos una serie de utilidades utilizadas para extraer información de los dispositivos conectados al sistema y obtener un listado de sus características.

Contexto

Los procesos en la CPU tienen su análogo en la GPU en los llamados contextos. Esta abstracción es un contenedor que alberga los diferentes objetos de un programa CUDA. A su vez, el contexto controla la vida de dichos objetos a lo largo de la ejecución de la aplicación, de forma que cuando el contexto es destruido los objetos que alberga también se destruyen.

Funciones y Kernels

Los *kernels* son probablemente la abstracción software más destacada de CUDA. Se tratan de funciones que expresan el paralelismo de datos de la aplicación, ya que al ejecutarse lo harán sobre una gran cantidad de hilos y exclusivamente en la GPU. En definitiva, son funciones cuya ejecución es ordenada de forma asíncrona por la CPU pero cuya ejecución se realiza en la GPU de forma masivamente paralela.

Dentro de un programa pueden residir tres tipos de funciones. El tipo de las mismas se indica decorando la declaración de la función mediante las palabras reservadas `__host__`, `__device__` o `__global__`. Las funciones *host* son aquellas que son lanzadas y ejecutadas en la CPU. Las funciones *device* son aquellas lanzadas y ejecutadas por la GPU. Por último, las funciones *global* hacen referencia a los *kernels*, funciones lanzadas por el *host* pero ejecutadas en el *device*.

Hilos, Warps, Bloques y Mallas

Los kernels se lanzan mediante mallas de bloques de hilos que se agrupan en unidades de 32 denominadas *warps*.

Los hilos o *threads* son la abstracción más reducida, cada uno de ellos posee un identificador único dentro del bloque al que pertenece y ejecuta el código del *kernel* empleando dicho identificador para determinar sobre qué datos debe actuar. Adicionalmente, cada uno de ellos tiene disponible una serie de registros del SM, cantidad que depende del número de hilos presente en el multiprocesador. Los hilos se agrupan en la unidad mínima de computación denominada *warp*. Un *warp* está conformado por 32 hilos (cada uno de ellos denominado línea o *lane*) y estos se ejecutan de forma físicamente paralela como si del modelo SIMD se tratara.

Los bloques o *blocks* son agrupaciones de hilos en una, dos o tres dimensiones. Al igual que los hilos, cada bloque tiene un identificador único dentro de la malla a la que pertenece. Cada bloque se ejecuta sobre un único SM de forma íntegra. Cabe destacar que, dependiendo de la capacidad del mismo, un SM puede tener asignados varios bloques para ejecución.

La malla o *grid* es una forma de estructurar los bloques, también en una, dos o tres dimensiones.

3.2.4. Flujo de Ejecución

Respecto al flujo de ejecución de una aplicación CUDA (mostrado en la Figura 3.9), cabe destacar que encontrarmos siempre un proceso maestro sobre la CPU que orquestra todo el programa y realiza las llamadas a la API de CUDA. De forma simplificada, se pueden destacar los siguientes pasos:

1. Inicialización de la GPU.
2. Reserva de memoria en *host* y *device*.
3. Copia de datos desde el *host* al *device*.
4. Lanzamiento de *kernel*.
5. Copia de datos desde el *device* al *host*.
6. (Repetición de 3-5 las veces que sea necesario).
7. Liberación de memoria y finalización del proceso.

De este proceso, lo más destacable es el hecho de que se necesita reservar memoria tanto en la CPU como en la GPU puesto que físicamente poseen espacios de memoria distintos. Esto implica a su vez que los datos sobre los que se va a operar tienen que ser copiados de la memoria principal de la CPU a la GPU. De la misma forma, los resultados que sean necesarios para el programa tienen que ser copiados de vuelta de la GPU a la CPU.

3.3. Evolución Generacional

Desde la primera microarquitectura GPU que dio comienzo a CUDA han sucedido diversas evoluciones con múltiples objetivos, principalmente con la mejora del rendimiento siempre en mente pero también la inclusión de nuevas características a nivel de hardware para los desarrollos e incluso la reducción del consumo energético.

Las especificaciones generales y las características soportadas por una tarjeta en concreto vienen indicadas por la denominada *compute capability*. Para poder consultar las características del hardware

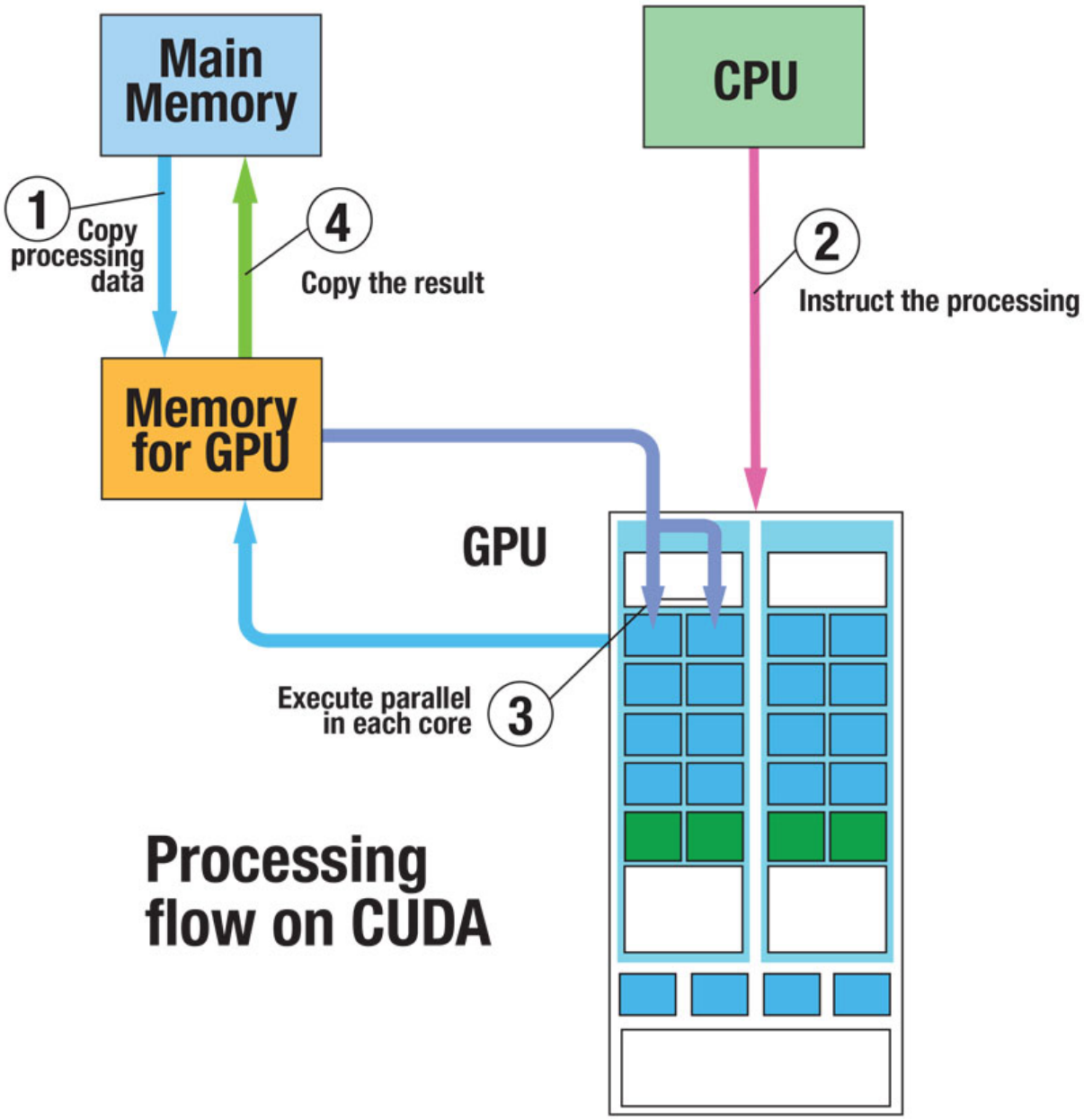

Figura 3.9: Flujo de procesamiento en CUDA

disponible en un sistema, CUDA ofrece herramientas tales como NVIDIA-System Management Interface (SMI) o el juego de funciones *device query*.

Cabe destacar que, junto con la evolución del *hardware*, la plataforma *software* también ha tenido versiones a lo largo del tiempo para acomodar todas estas mejoras y hacerlas accesibles para dos desarrolladores.

Figura 3.10: Roadmap de CUDA en el que se muestran las diferentes generaciones así como su año de introducción frente a su capacidad de cómputo estimada normalizada (factor multiplicador respecto a la primera generación).

3.3.1. Microarquitecturas

La introducción de la serie *GeForce 8* en el año 2006 dio el pistoletazo de salida a las microarquitecturas CUDA con Tesla. Desde entonces se ha producido una evolución constante a lo largo de cinco generaciones más hasta la fecha. Esta evolución de diferentes microarquitecturas se puede apreciar de un vistazo en el *roadmap* mostrado en la Figura 3.10. A continuación describiremos en mayor detalle cada una de estas generaciones hardware, destacando sus principales características y aportes.

Tesla

El nombre en clave para la primera microarquitectura CUDA fue Tesla [14]. Esta generación se caracterizó por emplear por primera vez *shaders* unificados y dar soporte al ecosistema CUDA con caraterísticas limitadas pero punteras para aquél entonces. Las tarjetas de la serie *GeForce 8*, *GeForce 9*, *GeForce 100*, *GeForce 200* y *GeForce 300* poseen esta microarquitectura, fabricadas con un proceso de 90, 80, 65 y 55 nanómetros respectivamente.

Fermi

La microarquitectura sucesora de Tesla recibió el nombre de Fermi [19] y fue introducida en las series *GeForce 400*, *GeForce 500* (y algunas *GeForce 600* y *GeForce 700*) con un proceso de fabricación de 40 nanómetros para las GPUs de escritorio. El principal salto aportado por esta microarquitectura fue el

soporte de operaciones en punto flotante de doble precisión (64 bits) lo cual supuso la ruptura de una de las barreras más importantes para el cómputo científico.

Kepler

Kepler [6] fue el nombre dado a la microarquitectura sucesora de Fermi. Esta generación fue la primera en poner el foco en la eficiencia energética y en la programabilidad más allá del rendimiento puro y duro. El principal cambio para mejorar la eficiencia energética fue la introducción de la velocidad de reloj unificada para los SM de nueva generación SMX así como la simplificación de los *schedulers*. La programabilidad se vio mejorada gracias a la introducción de una gran cantidad de nuevas características en CUDA, siendo la más destacada de ellas el paralelismo dinámico, dotanto a los kernels de la capacidad de lanzar otros kernels dentro de ellos. La mayoría de las tarjetas de las series *GeForce 600*, *GeForce 700* y algunas de la *GeForce 800* utilizan Kepler con un proceso de fabricación de 28 nanómetros.

Maxwell

La microarquitectura Maxwell [5] como sucesora de Kepler, también en 28 nanómetros, fue introducida en los últimos modelos de la serie *GeForce 700* y en las series *GeForce 800* y *GeForce 900*. Esta generación introdujo un tipo de SM mejorado denominado Maxwell Streaming Multiprocessor (SMM) con una gran eficiencia energética. Básicamente, la arquitectura permitía un control más preciso de la distribución del trabajo a los recursos disponibles, reduciendo el consumo energético drásticamente de aquellos recursos no utilizados.

Pascal

Pascal [8] es la microarquitectura introducida tras Maxwell en la serie *GeForce* 10 con un proceso de fabricación de 16 nanómetros. Desde el punto de vista arquitectural, Pascal es uno de los hitos más importantes en la evolución de las GPUs dada la gran cantidad de avances tecnológicos introducidos: memoria unificada, procesamiento específico para operaciones de 16 bits (ejecutadas sobre procesadores de 32 al doble de velocidad), soporte para un nuevo tipo de bus entre la CPU y la GPU (NVLink) con mayor ancho de banda y velocidad que el ya desactualizado PCIe así como memoria en disposición 3D con un ancho de banda ampliamente mejorado denominada High Bandwidth Memory (HBM).

Volta

Volta [9] es la microarquitectura introducida tras Pascal con un gran enfoque en el aprendizaje profundo. Prueba de ello es la introducción de otro tipo de unidades de procesamiento denominadas *Tensor Cores* con el propósito de acelerar el entrenamiento de redes neuronales. Además, esta generación incorpora una versión mejorada de la memoria 3D introducida por Pascal denominada HBM-2 y a su vez la revisión del bus NVLink-2.0.

A pesar de las mejoras, Volta es una microarquitectura que no llegó al mercado de consumo, quedando relegada a los centros de procesamiento de datos con el supercomputador DGX-1 y a las tarjetas de visualización y alta gama Quadro GV100 y Titan V respectivamente.

Turing

Turing [10] es la microarquitectura sucesora de Volta, con foco en el raytracing en tiempo real. Fue presentada en SIGGRAPH 2018 y dio el pistoletazo de salida a una nueva generación de GPUs con la denominación RTX. Sus principales bazas son la introducción de un nuevo tipo de unidades de procesamiento denominadas *RayTracing Cores* así como la optimización de los anteriores *Tensor Cores*, también llamados *AI Cores*. Los *RayTracing Cores* están diseñados para acelerar el cálculo de quadtrees, jerarquías esféricas y las comprobaciones de colisión de rayos con triángulos. Por otro lado, los *AI Cores* siguen especializados en la ejecución de rutinas de aprendizaje profundo, en concreto ayudan al raytracing mediante técnicas de denoising. Esta combinación de unidades de procesamiento especializados ha permitido a las tarjetas Turing el ejecutar rutinas de raytracing (con ciertas limitaciones) en tiempo real.

Además de esas características principales, la actual arquitectura Turing incorpora otra serie de cambios de gran importancia: memoria GDDR6, VirtualLink VR, DisplayPort 1.4 y un el puente NVLINK para stacking de VRAM entre múltiples tarjetas.

3.3.2. CUDA Toolkit

De forma paralela a las generaciones hardware, el conjunto de herramientas software (también conocido como CUDA Toolkit) ha evolucionado para dar soporte a las mejoras introducidas por las nuevas microarquitecturas y facilitar las tareas de los programadores. Una lista exhaustiva de todas las versiones liberadas hasta la fecha así como sus notas de versión asociadas pueden encontrarse en el archivo de CUDA Toolkit [1]. En la Figura 3.11 mostramos una vista resumida de esta evolución.

Figura 3.11: Roadmap del CUDA Toolkit.

[1] https://developer.nvidia.com/cuda-toolkit-archive

3.3.3. NVIDIA-SMI

Dado el entramado de versiones software y generaciones hardware es de gran utilidad el poder consultar las características técnicas de los dispositivos presentes en el sistema así como de los paquetes software instalados en el mismo. Para ello, NVIDIA proporciona una herramienta en línea de comandos conocida como SMI [7]. La Figura 3.12 muestra un ejemplo de ejecución de esta utilidad.

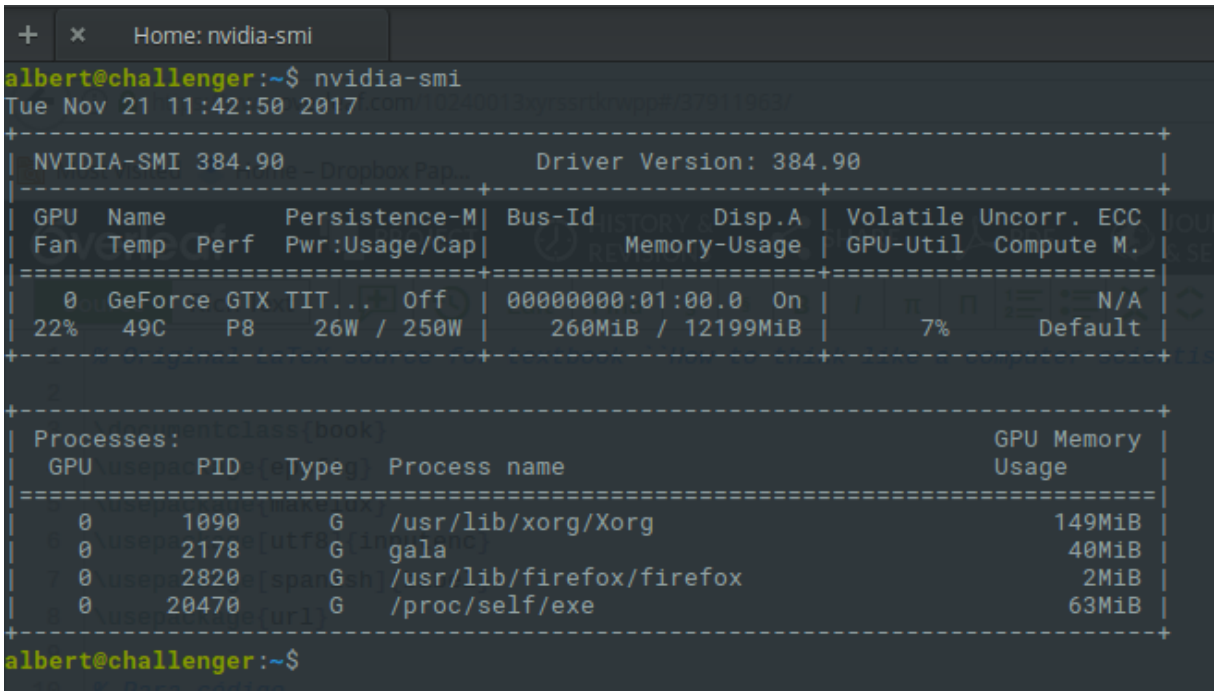

Figura 3.12: Resultados de NVIDIA SMI para una GPU Titan X (Maxwell).

3.3.4. Device Query

Existe una forma de obtener las especificaciones de las GPUs presentes en el sistema de forma programática. El *runtime* de CUDA proporciona una serie de funciones para recuperar información sobre los dispositivos y los drivers instalados como puede ser la versión de CUDA, la cantidad de memoria de la tarjeta, qué CUDA *Compute Capability* posee, diferentes limitaciones de la propia tarjeta y la arquitectura, etc. Uno de los ejemplos más destacados del CUDA Toolkit se denomina *deviceQuery* y recopila todas las funciones de la API que proporcionan información sobre los dispositivos instaladas. La Figura 3.13 muestra la salida resultante de ejecutar dicho programa sobre un sistema con CUDA 8.0 y una GPU Titan X Maxwell.

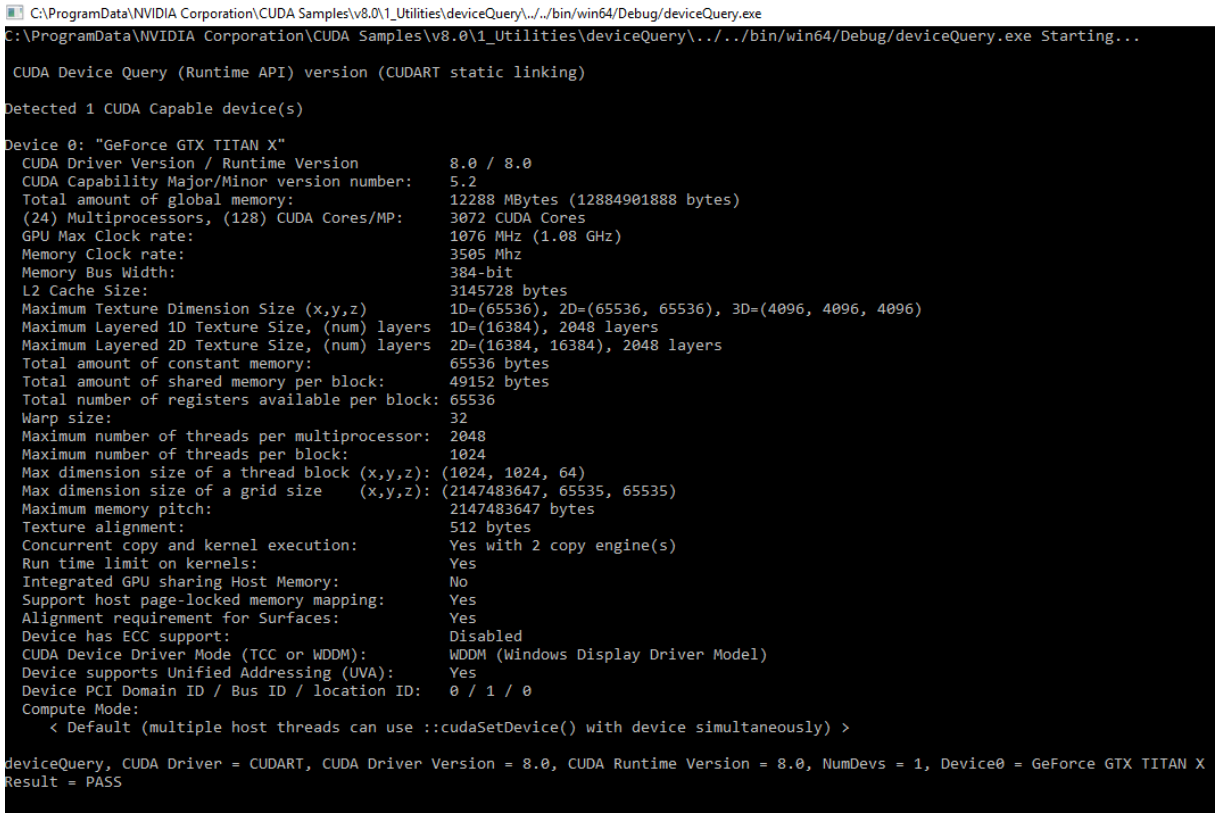

Figura 3.13: Resultados de deviceQuery para una GPU Titan X (Maxwell).

3.3.5. CUDA Compute Capability

A medida que las arquitecturas han ido evolucionando se han incorporando nuevas funcionalidades (operaciones en punto flotante de 16 bits), características (paralelismo dinámico) e incluso se han ampliado limitaciones existentes (cantidad de memoria compartida por multiprocesador). CUDA *Compute Capability* es un término utilizado para referirse a las características soportadas por una GPU concreta. Es por lo tanto un concepto ligado tanto a la arquitectura hardware como software, ya que por lo general las características deben soportarse tanto física como virtualmente. La Tabla 3.14 muestra un resumen de las características y funcionalidades soportadas en función del *Compute Capability* (en la página `https://en.wikipedia.org/wiki/CUDA` se puede encontrar un listado más detallado).

Feature support (unlisted features are supported for all compute abilities)	Compute ability (version)									
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5, 3.7, 5.0, 5.2	5.3	6.x
Integer atomic functions operating on 32-bit words in global memory	No	Yes								
atomicExch() operating on 32-bit floating point values in global memory										
Integer atomic functions operating on 32-bit words in shared memory	No	Yes								
atomicExch() operating on 32-bit floating point values in shared memory										
Integer atomic functions operating on 64-bit words in global memory										
Warp vote functions										
Double-precision floating-point operations	No			Yes						
Atomic functions operating on 64-bit integer values in shared memory	No				Yes					
Floating-point atomic addition operating on 32-bit words in global and shared memory										
_ballot()										
_threadfence_system()										
_syncthreads_count(), _syncthreads_and(), _syncthreads_or()										
Surface functions										
3D grid of thread block										
Warp shuffle functions	No					Yes				
Funnel shift	No						Yes			
Dynamic parallelism	No							Yes		
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No								Yes	
Atomic addition operating on 64-bit floating point values in global memory and shared memory	No									Yes

Data Type	Operation	Supported since	Supported since for Global Memory	Supported since for Shared Memory
16-bit integer	general operations			
32-bit integer	atomic functions		1.1	1.2
64-bit integer	atomic functions		1.2	2.0
16-bit floating point	addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	5.3		
32-bit floating point	atomicExch()		1.1	1.2
32-bit floating point	atomic addition		2.0	2.0
64-bit floating point	general operations	1.3		
64-bit floating point	atomic addition		6.0	6.0

Figura 3.14: Tabla de características y especificaciones Compute Capability.

Capítulo 4

Problemas 2: Hilos en CUDA

Contenido

4.1. Problema 1

Un estudiante ha mencionado que es capaz de multiplicar dos matrices de 1024x1024 utilizando un código basado en tiling y utilizando 1024 hilos por bloque en la arquitectura G80 con Compute Capability 1.3. Además ha mencionado que cada hilo es capaz de calcular un elemento de la matriz resultante. ¿Cuál sería tu reacción y por qué?

4.2. Problema 2

Relacionado con el ejercicio 1, para la multiplicación de matrices utilizando tiling, ¿deberíamos utilizar bloques de hilos de tamaño 8x8, 16x16, o 32x32 suponiendo que tenemos disponible una tarjeta con arquitectura GT200 con Compute Capability 1.3?

4.3. Problema 3

Imagina que subdivimos el espacio 3D en vóxels. Queremos calcular de forma paralela una primitiva sobre cada uno de los vóxels que componen el espacio 3D. ¿Cómo organizarías los hilos para llevar a cabo la ejecución? ¿Por qué? Si guardamos el resultado de cada uno de los hilos en un vector unidimensional, ¿cómo calcularías el índice global?

Capítulo 5

Modelo de procesamiento

Contenido

Los kernels CUDA son ejecutados en la GPU siguiendo un modelo de procesamiento especial que permite ocultar la latencia de las diversas operaciones que se llevan a cabo para conseguir aceleración mediante paralelismo. Este modelo de procesamiento conlleva una serie de pormenores en cuanto al lanzamiento de los kernels, la división de sus hilos, su planificación para conseguir una escalabilidad transparente y una serie de limitaciones temporales y espaciales.

5.1. Lanzamiento de Kernels

Los kernels pueden ser invocados o lanzados utilizando tanto las funciones propias del driver de CUDA como el API *runtime* simplificado con una sintaxis especial tal y como se muestra a continuación:

```
kernel <<< grid, block, shmem, stream >>> (param1, ... , paramN);
```

En dicha llamada, `kernel` es una función `__global__` llamada desde la CPU pero ejecutada en la GPU. En dicha llamada debemos especificar ciertos parámetros específicos de CUDA y del kernel en cuestión entre los caracteres `<<< ... >>>`. Estos parámetros de lanzamiento indican las dimensiones

de la malla `grid`, de los bloques `block`, la cantidad de memoria compartida a reservar `shmem` (ver Capítulo 7) y el *stream* sobre el que se ejecutará el kernel `stream` (ver Capítulo **??**). Estos dos últimos parámetros de lanzamiento son opcionales y se pueden omitir (en capítulos posteriores describiremos su significado y funcionamiento). Por último, de forma análoga a cualquier llamada de función de C/C++, los parámetros de la misma se especifican entre paréntesis.

De forma alternativa, esta invocación puede ser realizada utilizando la API del driver en lugar del *runtime* de CUDA. Para ello haremos uso de la función `cuLaunchKernel` que posee la siguiente interfaz:

```
CUResult cuLaunchKernel (
        CUfunction kernel,
        unsigned int gridDimX,
        unsigned int gridDimY,
        unsigned int gridDimZ,
        unsigned int blockDimX,
        unsigned int blockDimY,
        unsigned int blockDimZ,
        unsigned int shmem,
        CUstream stream,
        void ** params,
        void ** extra);
```

5.2. Mallas, Bloques, Hilos, Warps y Lanes

Como ya comentamos anteriormente, el kernel se ejecuta sobre la malla de hilos de forma que cada uno de ellos ejecuta una copia de dicho kernel. La malla se divide en bloques, los cuales están compuestos por los propios hilos que a su vez se agrupan en las unidades mínimas de ejecución conocidas como *warps*. Los warps son agrupaciones de 32 hilos en las cuales cada uno de los hilos recibe el nombre de *lane*. Esta división no es trivial y responde a motivos de escalabilidad que abordaremos posteriormente.

Cabe destacar que, a pesar de que cada hilo ejecuta el mismo código del kernel, cada uno de ellos debe operar sobre datos distintos. Para este propósito, cada hilo utiliza una serie de variables de CUDA (automáticamente declaradas y disponibles dentro de cualquier kernel) con valores específicos para cada hilo y bloque con el objetivo de identificarse y decidir sobre qué dato o posiciones de memoria tiene que operar:

- `gridDim`: dimensión de la malla.
- `blockDim`: dimensión de los bloques.
- `blockIdx`: identificador del bloque dentro de la malla.
- `threadIdx`: identificador del hilo dentro del bloque.

Dichas variables son del tipo especial `dim3`, el cual posee tres dimensiones o campos (p. ej., `gridDim.x`, `gridDim.y` y `gridDim.z`) para poder organizar tanto la malla como los bloques en una,

dos o tres dimensiones tal y como se muestra en las Figuras 5.1 (2D) y 5.2 (3D). Cabe destacar que no existe ninguna ganancia de rendimiento directa al utilizar bloques o mallas de una dimensionalidad particular, es una decisión puramente de aplicación.

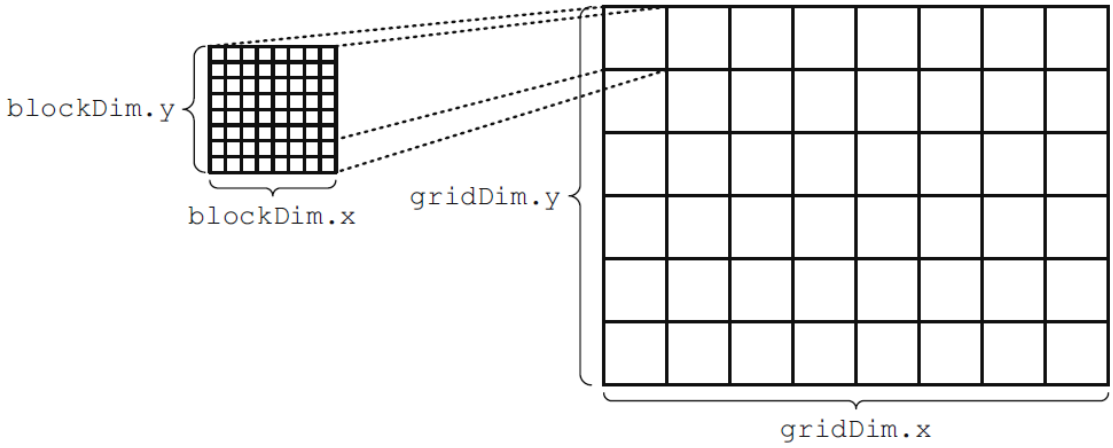

Figura 5.1: Malla 2D y bloques 2D.

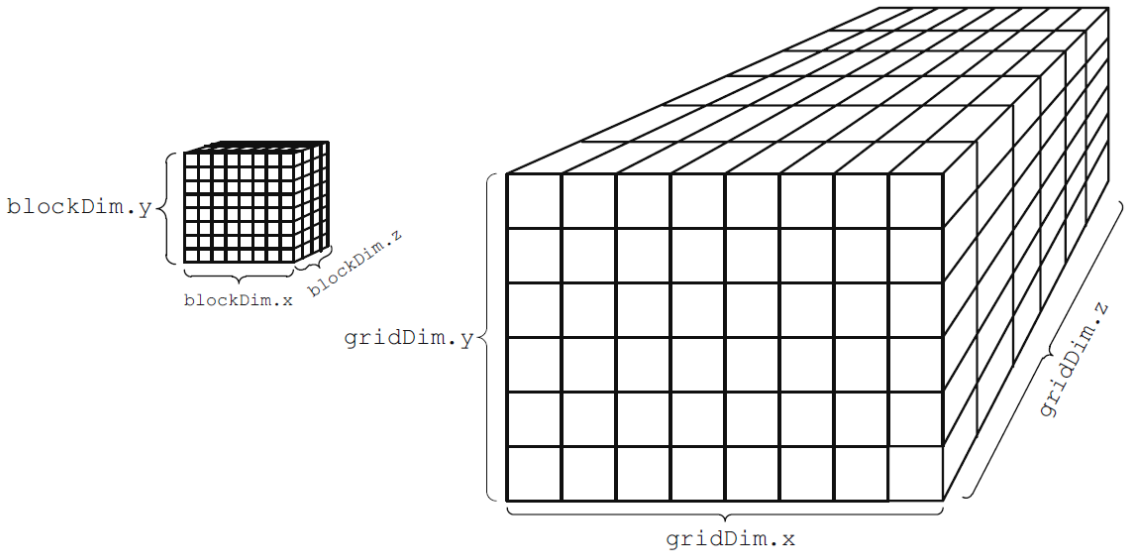

Figura 5.2: Malla 3D y bloques 3D.

Debido a las agrupaciones en *warps*, es común elegir configuraciones de bloque cuyo número de hilos sea un múltiplo del tamaño del *warp*. En caso contrario, ciertos warps han de ser completados con hilos inactivos tal y como se muestra en la Figura 5.3.. Este proceso denominado *padding* conlleva una pérdida de rendimiento dado que no se utilizan al completo todos los recursos.

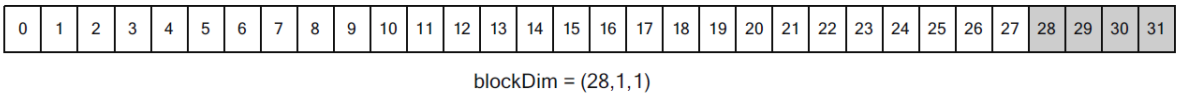

Figura 5.3: Bloque de 28 hilos con 4 hilos inactivos para completar el warp.

Cuando se realiza la llamada al *kernel* empleando el API *runtime* de CUDA tal y como mostramos anteriormente, las dimensiones de malla y de bloque son especificadas por los parámetros de lanzamiento `grid` y `block` respectivamente. Para ello se emplea el tipo de dato `dim3` que permite definir los tamaños con tres componentes para las dimensiones X, Y y Z que por defecto se inicializan a 1. Tras la llamada, cada hilo dispone de toda la información que necesita para operar: los argumentos pasados al kernel sobre los que operará (generalmente *arrays* en memoria GPU) y las variables especiales para identificarse y direccionar los datos.

```
        // 1024 hilos en bloque de 32 x 32 x 1 (2D)
        dim3 block_dim (32, 32, 1);
```

```
        // 16 bloques en malla de 4 x 4 x 1 (2D)
        dim3 grid_dim (4, 4, 1);
```

```
kernel <<< grid_dim, block_dim >>> (param1, ... , paramN);
```

El ejemplo más sencillo para ilustrar este concepto es la suma de vectores: disponemos de dos vectores de entrada, A y B de 1024 elementos cada uno, los cuales queremos sumar y depositar el resultado de la suma de sus elementos en un vector C. Para realizar esta operación en paralelo lanzamos un kernel con un tamaño de bloque de, por ejemplo 4x1x1 (por simplicidad de la explicación), y por lo tanto un tamaño de malla de 256x1x1 para lanzar un total de 1024 hilos (uno por cada elemento de los vectores). Cada hilo debe computar la suma de una posición y depositarla en el vector de salida por lo que la operación a realizar es clara:

```
        C[i] = A[i] + B[i];
```

Utilizando las variables especiales anteriores podemos mapear cada hilo a una posición del vector para determinar el índice i que le corresponde:

```
        i = blockIdx.x * blockDim.x + threadIdx.x;
```

De esta forma, y dado que threadIdx es local a cada bloque, los cuatro hilos del bloque 0 procesarán los cuatro primeros elementos del vector (blockIdx.x = 0, ya que es el primer bloque, y blockDim.x = 4 pues es la dimensión que hemos decidido):

```
Primer elemento, primer bloque) 0 * 4 + 0 = 0
(Segundo elemento, primer bloque) 0 * 4 + 1 = 1
(Tercer elemento, primer bloque) 0 * 4 + 2 = 2
(Cuarto elemento, primer bloque) 0 * 4 + 3 = 3
```

Seguidamente, los hilos del segundo bloque computarán los cuatro elementos siguientes (blockIdx.x = 1, ya que es el segundo bloque, y blockDim.x = 4 puesto que el tamaño no cambia):

```
(Primer elemento, segundo bloque) 1 * 4 + 0 = 4
(Segundo elemento, segundo bloque) 1 * 4 + 1 = 5
(Tercer elemento, segundo bloque) 1 * 4 + 2 = 6
(Cuarto elemento, segundo bloque) 1 * 4 + 3 = 7
```

5.3. Limitaciones de Memoria

Como veremos en sesiones posteriores sobre el modelo de memoria de CUDA. Cada hilo dentro de una malla necesita una cierta cantidad de memoria para ejecutarse, además de la memoria necesaria para los datos que pueden ser accedidos por todos los hilos. Ciertas variables propias de cada hilo pueden alojarse en la asignación correspondiente a cada hilo del banco de registros del SM. No obstante, la cantidad de registros es relativamente escasa para este propósito por lo que parte de la memoria necesaria acaba siendo utilizada de una abstracción de la memoria de la GPU denominada memoria local.

Dado que generalmente se utiliza una cantidad elevada de hilos en ejecución, es presumible que también se necesite una porción considerable de la memoria de la GPU para asignar la cantidad necesaria a cada uno de los hilos. Desafortunadamente, la memoria en la GPU no es ilimitada, y por motivos de eficencia no se lleva a cabo paginación de memoria virtual en la GPU (existen formas de utilizar la memoria del host de forma transparente empleando Unified Virtual Addressing, pero se trata de u tema fuera del ámbito de esta explicación). Para evitar la posibilidad de que un kernel se quede sin memoria a mitad de ejecución, CUDA hace una reserva anticipada de la memoria necesaria por hilo. De esta forma, si la reserva de memoria falla por falta de espacio, el lanzamiento del kernel falla con el error correspondiente.

5.4. Limitaciones de Tiempo

En sistemas en los cuales la GPU utilizada para cómputo está siendo a su vez utilizada para visualización, la ejecución de un kernel costoso puede provocar un impacto negativo en la interactividad del sistema puesto que la GPU no es capaz de cambiar de contexto de ejecución de cómputos CUDA

a cálculo de gráficos para el sistema. Por este motivo, el driver y los sistemas operativos fuerzan un tiempo máximo de ejecución para las GPUs empleadas a su vez para visualización. Una vez expirado ese tiempo, la GPU se reinicia para servir de nuevo su propósito principal.

En el caso de Linux, el tiempo está fijado en 2 segundos. En el caso de Windows, el driver puede configurarse en dos modos: WDDM (Windows Display Driver Model), en cuyo caso el sistema operativo decide dicho tiempo, y TCC (Tesla Compute Cluster), modo en el que no se fuerza ningún tipo de tiempo máximo de cómputo.

5.5. Escalabilidad Transparente y Planificación

Durante la ejecución, el hardware se encarga de asignar los bloques de la malla a los SM. De esta forma, no todos los bloques son ejecutados de forma concurrente y no se da ninguna garantía respecto al orden de ejecución de los mismos, más allá de que los hilos de un mismo bloque sean ejecutados por el mismo SM. Cada SM tiene suficientes recursos para mantener el contexto de varios bloques en ejecución (8 a la vez antes de los SM de tercera generación y 16 a partir de ella). Este modelo de ejecución es necesario para lograr que un mismo programa pueda ejecutarse y escalar su eficiencia de forma transparente dependiendo de los recursos disponibles de la GPU en la que se ejecute.

Para cubrir los costes del paralelismo (latencia tanto de la memoria como de las instrucciones) es necesario que cada SM ejecute más hilos de los que un bloque por sí solo puede contener. Así pues, como comentábamos anteriormente, varios bloques son asignados a cada SM. Una vez un bloque ha sido asignado a un SM, sus hilos son ejecutados juntos, siguiendo un modelo SIMD, en agrupaciones de 32 hilos consecutivos denominadas warps. Los hilos o lanes de un warp se ejecutan físicamente en paralelo pues el warp completo se envía los núcleos del SM.

Los warps son por lo tanto la unidad de ejecución de CUDA y el motivo de su existencia y su tamaño no son decisiones triviales. Este modelo de procesamiento basado en los warps como unidades de planificación permite a la GPU ocultar la latencia del resto de operaciones (por ejemplo, realizando accesos de memoria coalescentes para servir información a varios hilos de un warp como veremos el próximas sesiones). Los SM implementan una planificación con zero-overhead, en la cual se mantiene un conjunto de warps elegibles para ejecución (aquellos cuya siguiente instrucción tiene sus operandos preparados para ejecutarse) los cuales son elegidos para ejecución mediante colas de prioridad. **Es importante destacar de nuevo que todos los hilos de un warp ejecutan la misma instrucción cuando son seleccionados.**

5.6. Métodos de Sincronización

Anteriormente hemos descrito cómo el modelo de procesamiento de CUDA fuerza a que a que cada bloque sea ejecutado por un SM y cómo, para conseguir transparencia en la escalabilidad, no se garantiza ningún tipo de orden en la ejecución de los mismos. Esto implica que los hilos de diferentes bloques no pueden cooperar de una forma sencilla. A pesar de todo, utilizando diferentes funciones de CUDA es posible programar mecanismos de sincronización entre bloques. No obstante, estos mecanismos artificiales suelen presentar muchas complicaciones y suponen una pérdida de eficiencia total.

Las opciones de sincronización eficientes se producen entre hilos de un mismo bloque ya que sí tenemos la garantía de que se ejecutarán en un mismo SM. Estos hilos de un mismo bloque pueden

sincronizarse de tres maneras distintas:

- Barreras de sincronización (ver Figura 5.4): la función intrínseca `__syncthreads()` puede ser utilizada para imponer una barrera de sincronización en la que todos los hilos del bloque deben esperar hasta que todos hayan alcanzado dicho punto en la ejecución. Una vez todos los hilos han alcanzado la barrera, la ejecución puede continuar con normalidad. **Es muy importante tener en cuenta que es posible generar una situación de bloqueo si se utiliza código condicional con barreras de sincronización**, pues si no todos los hilos del bloque alcanzan la instrucción de barrera la ejecución de los hilos a la espera no podrá continuar.

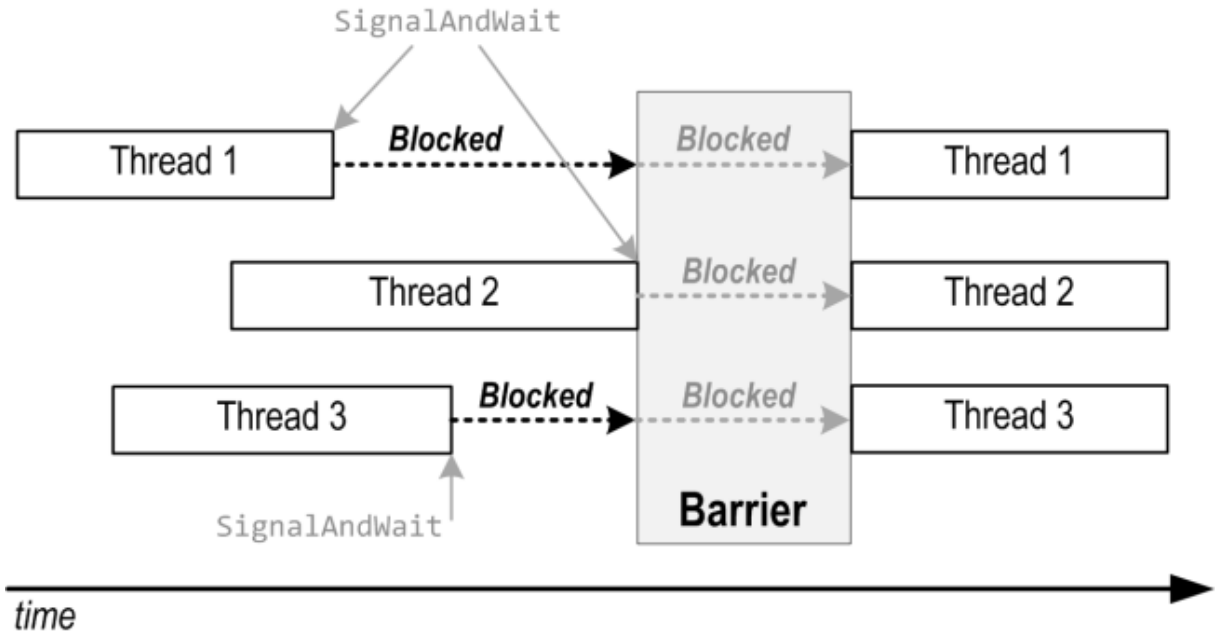

Figura 5.4: Ilustración de barrera de sincronización de hilos.

- Operaciones atómicas: las funciones atómicas son aquellas que llevan acabo operaciones de lectura-modificación-escritura en memoria global o compartida de forma que garantizan que dicha operación será realizada sin interferencia de otros hilos. En otras palabras, garantizan que esa lectura y modificación de la memoria será realizada por un único hilo a la vez, el cual impone un bloqueo sobre la posición en cuestión que solamente es liberado cuando finaliza la función. Esto permite realizar operaciones sobre una misma posición de memoria sin caer en condiciones de carrera, obviamente con cierta penalización al rendimiento dada la naturaleza secuencial del bloqueo.
- Memoria compartida: la porción de memoria compartida por los hilos de un mismo bloque puede ser utilizada a su vez para realizar ciertos tipos de sincronización. Este tipo de memoria y su funcionamiento será detallado la siguiente sesión sobre el modelo de memoria de CUDA.

5.7. Control de flujo

Las operaciones condicionales pueden ser utilizadas dentro de kernels CUDA, sin embargo, dado que todos los hilos de un warp deben ejecutar las mismas instrucciones al ser planificados, el tratamiento de las condiciones se realiza de una forma especial. Para ello, el compilador predica las instrucciones de condición de forma que todos los hilos ejecuten las mismas instrucciones. Esto significa que, por ejemplo, ante la siguiente situación:

```
if (x < 0.0)
  z = x - 2.0;
else
  z = sqrt(x);
```

El compilador predicará las instrucciones para que todos los hilos ejecuten las mismas instrucciones. Así pues, todos evaluarán la condición, posteriormente se llevará a cabo una pasada con todos los hilos ejecutando una rama y seguidamente una segunda ejecutando la otra rama.

```
cond: p = (x < 0.0);
p: z = x - 2.0;
!p: z = sqrt(x);
```

La diferencia reside en que en la primera pasada, todos los hilos que no cumplan la condición son enmascarados y marcados con un flag para ejecutar el equivalente a un NOP en el procesador. En la segunda pasada ocurre lo mismo pero con los hilos del warp que cumplan la condición.

Así pues, el caso común de un condicional implica que hilos de un mismo warp tomen caminos diferentes y por lo tanto se provoca una penalización al rendimiento al tener que ejcutar ambas ramas. Obviamente, este problema se agrava con el anidamiento de condiciones.

Sin embargo, dependiendo de la granularidad del salto, puede no haber divergencia. En el caso anterior, el más frecuente, la divergencia se produce en base al identificador del hilo, por ejemplo:

```
if (threadIdx.x > 2)
  dosomething;
else
  dootherthing;
```

En esta situación, como adelantamos anteriormente, los hilos de un mismo warp divergen y debemos ejecutar ambas ramas predicándolas. De forma técnica, se dice que la granularidad del salto es inferior al tamaño del warp. No obstante, si la granularidad del salto es múltiplo del tamaño del warp, como por ejemplo:

```
if (threadIdx.x/WARP_SIZE > 2)
  dosomething;
else
  dootherthing;
```

todos los hilos del warp siguen el mismo camino y el compilador puede optimizar el código generado evitando la necesidad de predicar las instrucciones. En otras, palabras, no existe divergencia.

5.8. Streams

Como ya enunciamos anteriormente, un kernel es una función invocada por la CPU pero cuya ejecución se produce en la GPU en forma de una gran cantidad de hilos siguiendo un patrón SIMT de forma paralela. A esta definición es necesario sumarle el hecho de que, por naturaleza, el lanzamiento de un kernel es asíncrono. La CPU simplemente emite las órdenes a la GPU depositándolas en un buffer en forma de cola circular. Esto quiere decir que una vez la CPU ordene el lanzamiento, el hilo en el host puede seguir su flujo de ejecución normal (a no ser que alcance una función de sincronización con la GPU o una operación bloqueante en el buffer de comandos a la GPU). Esta particularidad da pie al uso de streams para solapar cómputo y otras operaciones bloqueantes y de larga duración como las copias en memoria.

Esta funcionalidad avanzada de la tecnología CUDA permite el manejo de múltiples flujos de ejecución sobre una misma GPU, así como el uso y combinación de múltiples flujos con múltiples GPUs.

Además del paralelismo de grano fino expuesto por el modelo de procesamiento a la hora de procesar los hilos de un mismo warp de forma físicamente paralela y las jerarquías superiores de forma concurrente, el modelo de procesamiento de CUDA también admite concurrencia de un grano más grueso para mejorar la eficiencia de los programas:

- Concurrencia entre CPU y GPU: como hemos destacado, los kernels se ejecutan de forma asíncrona, por lo que una vez emitida la orden de lanzamiento por parte del host, éste puede reanudar su ejecución y realizar otro tipo de cómputos de forma independiente a la GPU.

- Concurrencia entre cómputo de kernel y transferencia de memoria: las GPUs poseen copy engines que actúan de forma independiente a los SM, por lo que se pueden realizar transferencias de memoria de forma independiente al trabajo realizado en los SMs.

- Concurrencia entre kernels: aquellas GPUs con SMs de generación 2.X o posterior pueden ejecutar hasta cuatro kernels en paralelo.

- Concurrencia multi-GPU: múltiples GPUs pueden operar en paralelo (este tema está fuera del ámbito de la explicación).

La forma de expresar esta concurrencia en CUDA es utilizando streams. Los streams son abstracciones para referirse a una secuencia de operaciones que se ejecutan de forma secuencial según el orden de envío en la GPU. La posibilidad de tener múltiples streams ejecutándose en la GPU implica que las operaciones de diferentes streams pueden ser ejecutadas de forma concurrente según los recursos disponibles y que además operaciones de diferente naturaleza pueden ser solapadas para utilizar los recursos de forma independiente (ver Figura 5.5).

El ejemplo más sencillo y efectivo de streams se puede dar considerando una aplicación típica que realiza una copia de memoria host-to-device, la ejecución de un kernel y posteriormente otra copia de memoria device-to-host. Las GPUs modernas disponen de dos copy engines (uno para transferencias host-to-device y otro para las device-to-host). Podemos utilizar varios streams, tal y como se muestra en la Figura 5, para solapar las copias en ambas direcciones junto con la ejecución de un kernel para conseguir un máximo aprovechamiento de todos los recursos del chip gráfico de forma simultánea.

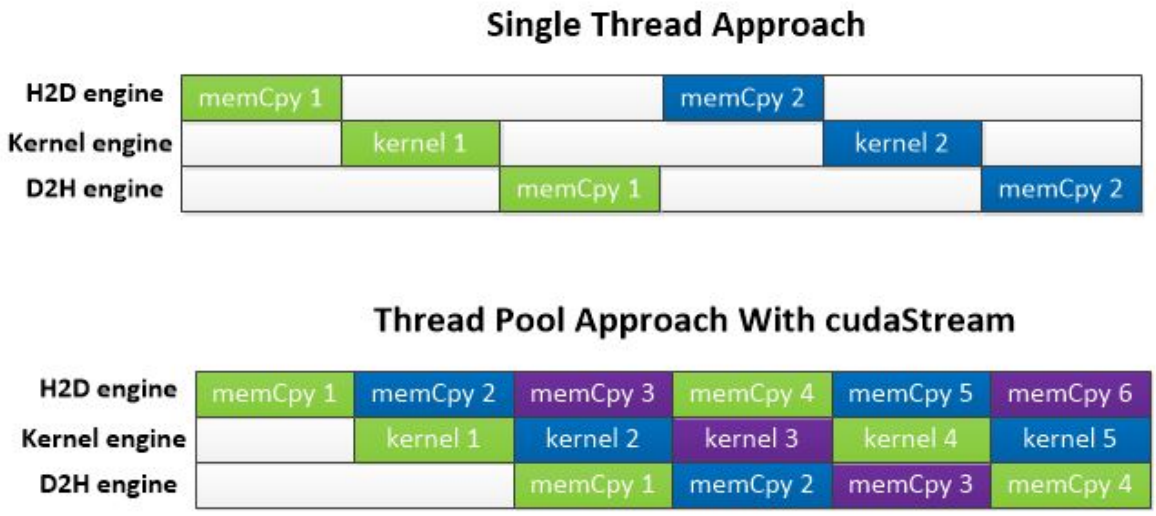

Figura 5.5: Concurrencia de copias de memoria y kernels mediante streams.

5.9. Medición de Tiempos, Sincronización Host-Device y Eventos

Analizar el rendimiento de nuestro código CUDA es vital para determinar la ganancia que estamos obteniendo con nuestra implementación y así establecer un punto de partida para optimizaciones posteriores. La medición de tiempos para calcular el rendimiento de la implementación se suele realizar en la parte host, bien utilizando temporizadores de la CPU o bien temporizadores específicos de CUDA. De cualquier manera, el modelo de procesamiento de CUDA impone ciertas restricciones que deben ser tenidas en cuenta a la hora de realizar las mediciones para asegurar su validez.

Como ya adelantamos, la ejecución de los kernels es asíncrona, lo cual quiere decir que una vez la CPU emite la orden y ésta se almacena en el buffer de comandos de la GPU, la parte host continúa su ejecución. Este hecho condiciona en gran medida la medición de tiempos. Por ejemplo, observemos el siguiente código.

```
cudaMemcpy(d_x_, h_x_, vector_sz_bytes_, cudaMemcpyHostToDevice);
cudaMemcpy(d_y_, h_y_, vector_sz_bytes_, cudaMemcpyHostToDevice);

kernel<<<grid, block>>>(d_x_, d_y_, N);

cudaMemcpy(h_y_, d_y_, vector_sz_bytes_, cudaMemcpyDeviceToHost);
```

En este código de ejemplo, las instrucciones cudaMemcpy son síncronas o bloqueantes, lo cual significa que no se ejecutarán hasta que todos los comandos CUDA anteriores hayan sido completados y, de igual manera, los comandos posteriores no comenzarán hasta que las copias hayan terminado. Esto contrasta con el hecho de que la llamada al kernel sea asíncrona, la CPU emitirá el comando apropiado

para que la GPU ejecute el kernel (dicho comando se almacenará en el buffer de la GPU para ser ejecutado en el momento que corresponda) y continuará su ejecución hasta alcanzar la instrucción de copia de memoria. No obstante, la naturaleza síncrona de las copias de memoria asegura que no se produzca ninguna condición de carrera.

Si tuviéramos que calcular el tiempo de ejecución empleado por el kernel, midiendo desde el host, podríamos hacerlo de la siguiente manera:

```
cudaMemcpy(d_x_, h_x_, vector_sz_bytes_, cudaMemcpyHostToDevice);
cudaMemcpy(d_y_, h_y_, vector_sz_bytes_, cudaMemcpyHostToDevice);

unsigned long t_start_ = cpu_timer();
kernel<<<grid, block>>>(d_x_, d_y_, N);
unsigned long t_end_ = cpu_timer();

cudaMemcpy(h_y_, d_y_, vector_sz_bytes_, cudaMemcpyDeviceToHost);
```

Sin embargo, dada la naturaleza asíncrona del kernel, únicamente estaríamos midiendo el tiempo que se tarda en emitir la orden de lanzamiento y no el tiempo de ejecución del kernel en sí. Para poder medir ese tiempo necesitamos imponer una barrera de sincronización explícita entre la CPU y la GPU para bloquear la ejecución en el host hasta que los comandos en el buffer del dispositivo hayan sido completados. Esto lo podemos conseguir empleando la instrucción `cudaDeviceSynchronize`.

```
cudaMemcpy(d_x_, h_x_, vector_sz_bytes_, cudaMemcpyHostToDevice);
cudaMemcpy(d_y_, h_y_, vector_sz_bytes_, cudaMemcpyHostToDevice);

unsigned long t_start_ = cpu_timer();
kernel<<<grid, block>>>(d_x_, d_y_, N);
cudaDeviceSynchronize();
unsigned long t_end_ = cpu_timer();

cudaMemcpy(h_y_, d_y_, vector_sz_bytes_, cudaMemcpyDeviceToHost);
```

El problema con esta aproximación es que bloquean tanto el flujo de ejecución en la CPU como en la GPU. Existe una forma alternativa para, entre otras utilidades, medir el tiempo de ejecución de partes del programa en la GPU. Para ello podemos utilizar los denominados eventos con la API de CUDA. Los eventos son esencialmente marcas temporales (del inglés timestamps) en la GPU que son establecidas en puntos especificados por el usuario. Estas operaciones generan una sobrecarga prácticamente nula en la ejecución pues es la propia GPU la que genera el timestamp.

Su utilización con la API de CUDA RT permite medir tiempos de forma sencilla. Con la función `cudaEventCreate` podemos crear eventos del tipo `cudaEvent_t`, la función `cudaEventRecord` nos permite especificar esos puntos en los que se va a almacenar el timestamp determinado en el evento. Por último, es importante sincronizar y bloquear la ejecución en el host para evitar leer el temporizador antes de que la GPU haya llegado a almacenar el timestamp. Para ello utilizaremos la función `cudaEventSynchronize`. Hecho esto, la función `cudaEventElapsedTime` nos permite calcular el

número de milisegundos entre dos timestamps con una resolución de medio microsegundo aproximadamente.

A continuación se muestra un ejemplo de código completo para medir el tiempo de ejecución correctamente de un kernel en la GPU.

```
cudaEvent_t t_start_, t_stop_;
cudaEventCreate(&t_start_);
cudaEventCreate(&t_stop_);

cudaMemcpy(d_x_, h_x_, vector_sz_bytes_, cudaMemcpyHostToDevice);
cudaMemcpy(d_y_, h_y_, vector_sz_bytes_, cudaMemcpyHostToDevice);

cudaEventRecord(t_start_);
kernel<<<grid, block>>>(d_x_, d_y_, N);
cudaEventRecord(t_stop_);

cudaMemcpy(h_y_, d_y_, vector_sz_bytes_, cudaMemcpyDeviceToHost);

cudaEventSyncrhonize(t_stop_);
float milliseconds_ = 0.0f;
cudaEventElapsedTime(&milliseconds_, t_start_, t_stop_);
```

Capítulo 6

Problemas 3: Hilos en CUDA (II)

Contenido

La ocupación es el ratio entre el número de hilos o warps que se ejecutan sobre un SM para un kernel concreto y el número máximo de hilos o warps que se podrían ejecutar potencialmente sobre dicho SM para ese kernel. Es decir: $occ = threadsPerSM/maxThreadsPerSM$.

La ocupación es el ratio entre el número de hilos o warps que se ejecutan sobre un SM para un kernel concreto y el número máximo de hilos o warps que se podrían ejecutar potencialmente sobre dicho SM para ese kernel. Es decir:

Los hilos en ejecución (reales) en un SM dependen de muchos factores, entre ellos: el tamaño de bloque elegido, el número de registros en uso por cada hilo, la memoria compartida necesaria y su configuración e incluso la compute capability de la GPU. El denominador, el máximo número de hilos teórico por SM es únicamente dependiente de la generación de la tarjeta y su compute capability.

Así pues, por ejemplo para una tarjeta con compute capability 3.7 tenemos un máximo de 2048 hilos por multiprocesador y una limitación de 16 bloques por multiprocesador. De esa forma, si ejecutamos un kernel con una configuración de 16 hilos por bloque, podremos tener un total de 256 hilos sin exceder la limitación de 16 bloques por multiprocesador. Esto significa que como mucho ejecutaremos 256 hilos a la vez, mientras que el máximo teórico es de 2048 por lo que la ocupación será de 256/2048 = 0.125 o lo que es lo mismo del 12.5 %, una utilización bastante pobre de los recursos.

Por norma general, una mayor ocupación implica una mejor utilización de los recursos de cómputo de la tarjeta y por lo tanto un mayor rendimiento. No obstante, el rendimiento final depende de varios factores y no solo de la ocupación por lo que en ocasiones merece más la pena ejecutar un kernel con una ocupación baja que aproveche mejor otros factores (Un gran ejemplo de este fenómeno es la presentación de Volkov [1].

6.1. Problema 1

Si necesitamos tantos hilos como elementos en dos vectores para llevar a cabo la suma de los mismos, ¿qué expresión de las siguientes sería la correcta para llevar a cabo el mapeo entre hilos y elementos?

- idx = threadIdx.x + threadIdx.y;
- idx = blockIdx.x + threadIdx.x;
- idx = blockIdx.x * blockDim.x + threadIdx.x;
- idx = blockIdx.x * threadIdx.x;

6.2. Problema 2

Continuando con el ejercicio anterior, si además queremos que cada hilo calcule dos posiciones adyacentes del vector de elementos, ¿qué expresión de las siguientes sería la correcta para llevar a cabo el mapeo entre hilos y elementos?

- idx = blockIdx.x * blockDim.x + threadIdx.x + 2;
- idx = blockIdx.x * threadIdx.x * 2;
- idx = (blockIdx.x * blockDim.x + threadIdx.x) * 2;
- idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;

6.3. Problema 3

Para una operación suma de vectores, asume que el tamaño de los vectores a sumar es 2000, cada hilo calcula 1 elemento de salida, y el tamaño del bloque es de 512 hilos, ¿cuántos hilos habrá en total en el grid ejecutado?

- 2000
- 2024
- 2048
- 2096

6.4. Problema 4

Si un SM de un dispositivo CUDA puede ejecutar hasta 1536 hilos y hasta 4 bloques a la vez, ¿cuál de las siguientes configuraciones obtendría mayor rendimiento (en este caso mayor número de hilos en ejecución)? Asumimos que en nuestro problema, el rendimiento viene determinado principalmente por la ocupación.

- 128 hilos por bloque
- 256 hilos por bloque
- 512 hilos por bloque
- 1024 hilos por bloque

6.5. Problema 5

Necesitamos escribir un kernel que opere sobre una imagen de tamaño 400x900 píxeles. Queremos asignar un hilo para cada pixel. El número de hilos por bloque tiene que ser cuadrado y utilizar el máximo número de hilos por bloque posible en tu dispositivo (arquitectura Fermi con Compute Capability 2.0).

- ¿Qué tamaño de grid y bloque elegirías?
- ¿Cuántos hilos no realizarán ningún cómputo?

Capítulo 7

Memorias CUDA

Contenido

El uso de las memorias es un aspecto fundamental en la programación de GPUs. El uso incorrecto de la jerarquía de memorias disponible en las GPUs puede ocasionar que no consigamos acelerar el algoritmo que estamos implementando de forma paralela, sino que además su tiempo de ejecución sea superior a una implementación secuencial en la CPU. Por ello, la elección del tipo de memoria más adecuada, así como el acceso a los datos en memoria, es realmente importante. El desarrollo de aplicaciones complejas puede requerir el uso de grandes cantidades de memoria, por ello, como ya veremos más adelante, tendremos que utilizar estrategias eficientes para el acceso a los datos. Por mucha capacidad de cómputo que tenga nuestra GPU o CPU, si la velocidad de acceso/escritura de las memorias no es suficiente, toda esta capacidad de cómputo estará desaprovechada.

7.1. Jerarquía de memorias

En general, debido a que la memoria con menor latencia esta muy limitada debido a su coste de producción en la mayoría de arquitecturas de computadores, se utilizan organizaciones piramidales donde encontramos memorias de distintos tipos. El objetivo es conseguir el rendimiento de una memoria con una latencia muy baja, al coste de una memoria con una latencia muy alta. Por ello se proponen jerarquías como las que vamos a estudiar a continuación.

En la CPU y en la GPU encontramos una jerarquía de memorias bien definida y donde cada memoria tiene un propósito distinto.

En la Figura 7.1 podemos ver un ejemplo típico de la jerarquía de memorias que encontramos en una CPU.

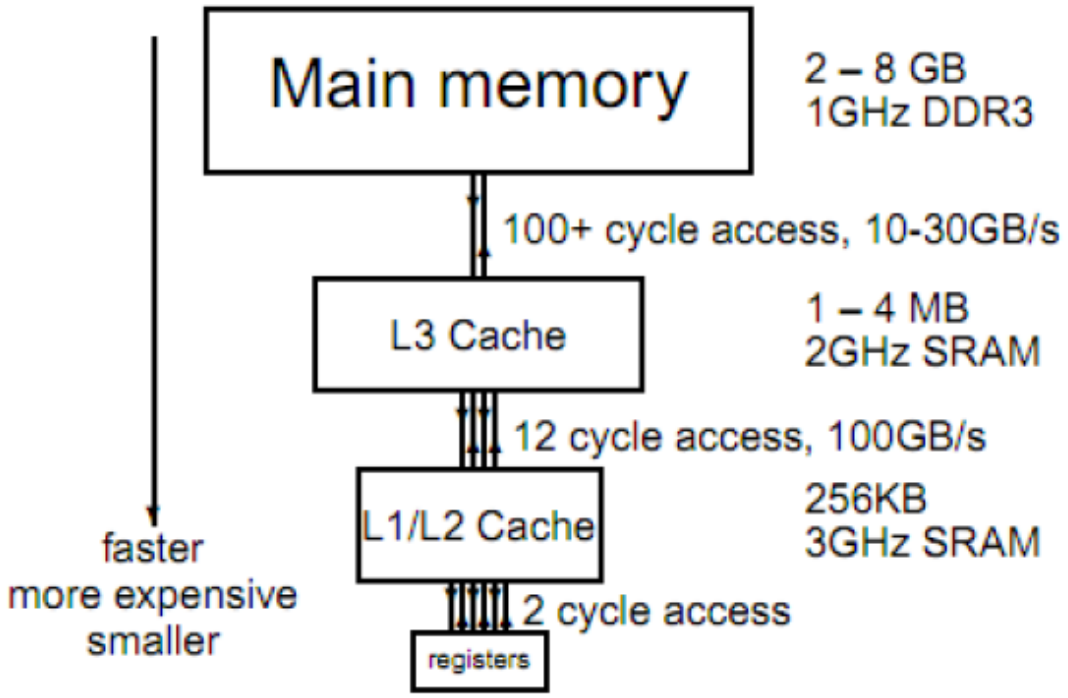

Figura 7.1: Esquema de jerarquía de memorias en un ordenador convencional.

Las memorias se distribuyen en distintos niveles, por un lado encontramos memorias con más capacidad de almacenamiento (Memoria RAM) pero tienen la desventaja de tener un tiempo de acceso superior. Por otro lado, si nos movemos a un escalón inferior dentro de esta jerarquía, encontramos memorias de menor capacidad de almacenamiento, pero mayor velocidad de acceso. En el nivel más

bajo, o más cercano al procesador, encontramos los registros. Los registros son un recurso muy limitado pero con una latencia muy baja. En los niveles intermedios, encontramos otro tipo de memorias con una capacidad de almacenamiento media, y con un tiempo de acceso también medio. Este tipo de memorias es muy útil para implementar lo que se conoce en arquitectura de computadores, como memoria caché. En este tipo de memorias se almacena un subconjunto de datos almacenados en la memoria principal. En concreto, el sistema operativo intentará almacenar los datos que más se estime que se vayan a reutilizar (mayor frecuencia) a lo largo de la ejecución de nuestro programa. La jerarquía de memoria va a permitir aumentar el rendimiento gracias a la explotación del principio de localidad. Encontramos dos tipos de principios de localidad:

- Localidad temporal: un dato que ha sido utilizado recientemente es probable que se utilice de nuevo a corto plazo.
- Localidad espacial: es probable usar los datos adyacentes a los usados recientemente, por ello se usan caches para guardar varios datos en una línea del tamaño del bus de acceso a memoria.

Esta jerarquía de memoria junto con el principio de localidad utilizado de forma correcta, permite sacar el máximo rendimiento a nuestros programas en la CPU, reduciendo el tiempo consumido en acceder a la memoria del ordenador.

7.2. Memoria en el Host (CPU)

En CUDA, la memoria Host se refiere a la memoria accesible por la CPU. Por defecto, esta memoria es paginable, lo que implica que el sistema operativo puede mover ciertas partes o desalojarlas si fuese necesario. Esto ocurre debido a que cuando utilizamos paginación de memoria la localización física puede variar. Para permitir que una porción de memoria reservada no se pagine, se utiliza un concepto de bloqueo, o de página bloqueada, donde no se permite su cambio o desalojo por parte del sistema operativo. Esto posibilita que dispositivos externos como una GPU puedan utilizar de forma eficiente esta porción de memoria utilizando DMA (Direct Memory Access).

7.2.1. Pinned memory

En CUDA, cuando hablamos de `Pinned memory`, nos referimos a este tipo de memoria que está bloqueada y mapeada para ser accedida directamente por la GPU. Este tipo de memoria ofrece un mayor rendimiento a la hora de llevar a cabo transferencias de información entre la memoria CPU y GPU. Además, permite que desde un Kernel en CUDA se pueda acceder directamente a esta memoria en la CPU, aunque obviamente su rendimiento va a ser muy inferior comparado con el acceso a memoria en la GPU.

La Pinned memory se reserva y libera en CUDA utilizando una serie de funciones disponible especialmente para ello: `cudaHostAlloc()/cudaFreeHost()`. Estas funciones permiten reservar a través del sistema operativo memoria siguiendo el esquema de paginas bloquedas, del Inglés, page-locked.

El uso de la pinned memory ha ido evolucionando conjuntamente con la salida de nuevas versiones de CUDA. CUDA 2.2 introdujo ciertas mejoras sobre este tipo de reserva de memoria permitiendo que esta memoria fuera portable, lo que significa que distintas GPUs pueden acceder a este tipo de memoria

mapeada para su uso por GPUs. CUDA 4.0 introdujo mejoras adicionales, permitiendo bloquear una zona de memoria ya existente para su uso por la GPU, así como la introducción de un nuevo concepto, Unified Virtual Addressing (UVA). UVA es un espacio de direcciones unficado que permite identificar de forma única todos los punteros a memoria, incluyendo punteros a memoria host y device (múltiples GPUs). De esta forma, además mediante UVA, es posible conocer si una dirección de memoria se encuentra reservada en el host o en el device. En CUDA 6.0 se hizo otra mejora sobre el sistema de memorias, introduciendo la memoria unificada, donde el runtime de CUDA se encarga de gestionar la copia automática de datos entre CPU y GPU. Explicaremos con detalle esta característica más reciente en el capítulo 6.

Por defecto, la reserva de memoria en Host es paginable, por lo tanto, cuando hacemos una transferencia de memoria reservada en la CPU con`malloc` en C o `new` en C++, el driver de CUDA internamente tiene que crear un buffer de memoria no paginable para llevar a cabo la copia a memoria device. Este doble proceso de copia se puede evitar utilizando la reserva explícita en memoria host de memoria no paginable. En la Figura 7.2 podemos ver un ejemplo de ambas situaciones, copia de datos desde memoria paginable y copia desde memoria no paginable (pinned memory).

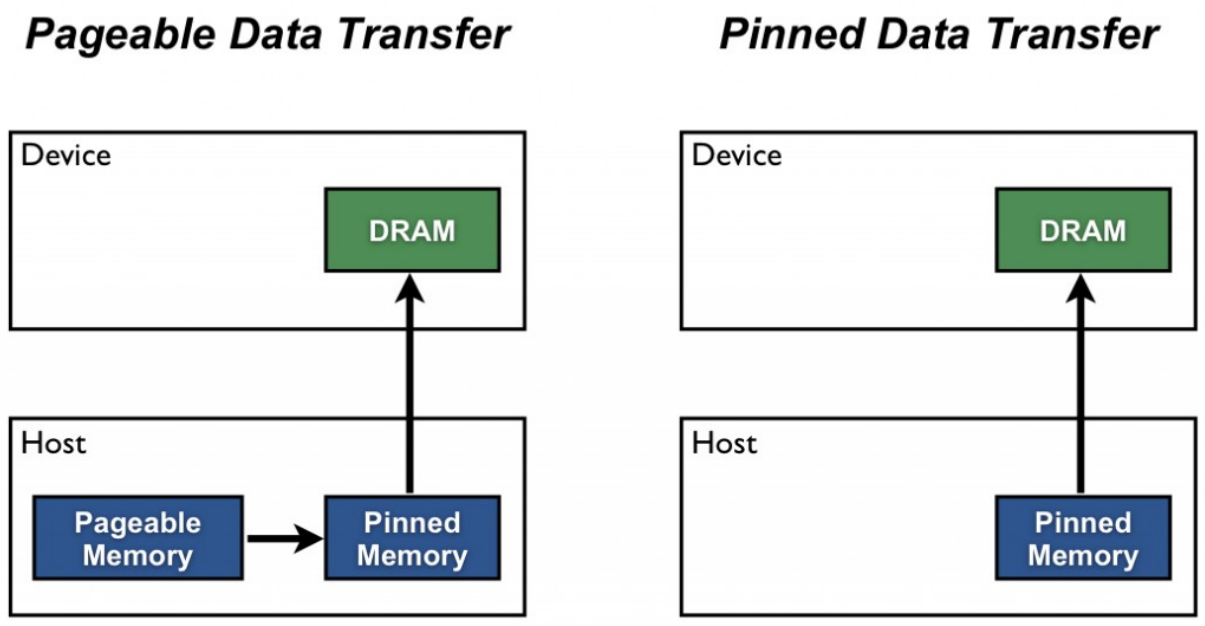

Figura 7.2: Transferencia de datos CPU/GPU desde memoria paginable (izquierda) y no paginable o pinned memory (derecha)

De esta forma, gracias al uso de la pinned memory, se puede llegar a duplicar el ancho de banda de transferencia entre CPU y GPU. Sin embargo, la memoria es un recurso escaso, por lo que es recomendable solo reservar la cantidad de memoria no paginable necesaria. Reservar demasiada memoria no paginable en la CPU puede reducir el rendimiento general del programa, al reducir la cantidad de

memoria disponible para el sistema operativo y el resto de programas.

A continuación vamos a ver un ejemplo completo de reserva y transferencia de datos usando pinned memory.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

const int NUM_ELEMENTOS = 1000;

int main(void) {

  int size = NUM_ELEMENTOS * sizeof(int);
  int* h_array;
  int* d_array;

  // Reserva memoria no paginable (pinned memory)
  cudaHostAlloc((void**)&h_array, size, cudaHostAllocDefault);

  // Reserva memoria GPU (device memory)
  cudaMalloc((void**)&d_array, size);

  // Transferencia entre memoria CPU-GPU
  cudaMemcpy(d_array, h_array, size, cudaMemcpyHostToDevice);

  // Llamada a un kernel que procese la información transferida
  ...

  // Liberamos memoria CPU (pinned) y GPU (device)
  cudaFreeHost(h_array);
  cudaFree(d_array);

  return 0;
}
```

Podemos observar que la reserva en memoria CPU se lleva a cabo con la instrucción `cudaHostAlloc`. Esta instrucción recibe tres parámetros, un puntero a memoria CPU donde se reservará la memoria, el tamaño de la memoria a reservar y finalmente el tipo de reserva. En este caso se ha usado el modificador `cudaHostAllocDefault` para especificar definir que la memoria no es paginable. Podemos usar otros modificadores para además mapear la memoria y poder usarla directamente en un kernel CUDA `cudaHostAllocMapped`, hacerla portable (visible múltiples GPUs) `cudaHostAllocPortable` y/o hacerla más eficiente para un esquema combinado de escritura por parte de la CPU y lectura desde la GPU (buffer write combined) `cudaHostAllocWriteCombined`. Todos estos modificadores son ortogonales entre ellos, por lo que podemos usarlos de forma combinada sin ninguna restricción.

Declaración variables	Memoria	Ámbito	Tiempo de vida
Variables en un kernel	Registros	Hilo ejecución	Kernel
Variables de tipo array en un kernel	Local	Hilo ejecución	Kernel
__device__, __shared__, int SharedVar;	Compartida	Block	Kernel
__device__, int GlobalVar;	Global	Grid	Programa
__device__, __constant__, int ConstVar;	Constante	Grid	Programa

Cuadro 7.1: Tipos de variables para el uso de distintos tipos de memorias en CUDA

7.3. Memoria en el Device (GPU)

La GPU, y en concreto una GPU compatible con CUDA, dispone de una jerarquía de memorias (Figura 7.3) ligeramente distinta a la que vimos anteriormente en la CPU, la mayoría de estas diferencias se deben a la capacidad de computación paralela de las mismas.

Además, la jerarquía de memorias de la GPU dispone de una interfaz para comunicarse con la memoria principal de la CPU, ya que la GPU necesita de una CPU para su funcionamiento. La CPU es la encargada de coordinar la copia de datos a las memorias de la GPU, así como la ejecución de código sobre sus procesadores. A partir de ahora, cuando hablemos de memoria en la GPU, nos referiremos a esta usando el término *Device memory*. La *device memory* dispone de un controlador de memoria dedicado encargado de gestionar todas las transferencias entre el host y el device (CPU/GPU). La memoria device es la memoria de mayor capacidad en la GPU, y normalmente nos referimos a ella como la memoria GDDR. Por ejemplo, la tarjeta gráfica de NVIDIA GTX480 dispone de 1.5 GB de memoria GDDR3. Por lo tanto la capacidad de la device memory de esta tarjeta gráfica es de 1.5 GB. Si consultamos la capacidad de otras tarjetas más modernas, por ejemplo, la NVIDIA GTX1080, esta tarjeta gráfica dispone de 8 GB de memoria GDDR5X. Esta memoria dispone de más capacidad y ancho de banda.

La device memory puede ser utilizada en CUDA de distintas maneras, de forma que según el tipo de reserva que hagamos, la memoria tendrá un uso diferente y por lo tanto a su vez también un rendimiento distinto. También es importante destacar que CUDA presenta una sintáxis ligeramente distinta para declarar variables en las distintas memorias disponibles. Cada declaración de esas variables además dotará a la misma de un ámbito de actuación así como un tiempo de vida durante la ejecución de nuestros programas. En la Tabla 7.1 mostramos ejemplos de sintáxis para declarar variables en las distintas memorias, además también se muestra el ámbito de dichas variables y su tiempo de vida.

A continuación vamos a ver y analizar con detalle las características de cada una de las distintas formas en que podemos reservar memoria en la device memory de una GPU compatible con CUDA.

7.3.1. Memoria global

La memoria global es la memoria de mayor tamaño que encontramos en la GPU. A su vez, es la memoria con mayor latencia. Debido a su capacidad, esta memoria se utiliza normalmente como contenedor cuando copiamos datos entre el Host (CPU) y el Device (GPU). Posteriormente los datos pueden ser copiados a otras memorias, dentro de la GPU, más eficientes para optimizar el acceso a los datos, por ejemplo, a través del uso de la memoria compartida o registros.

La memoria global es compartida por todos los Streaming Multiprocessors de la GPU, por lo que todos los hilos podrán acceder al mismo espacio de direcciones de forma simultánea. El acceso a esta

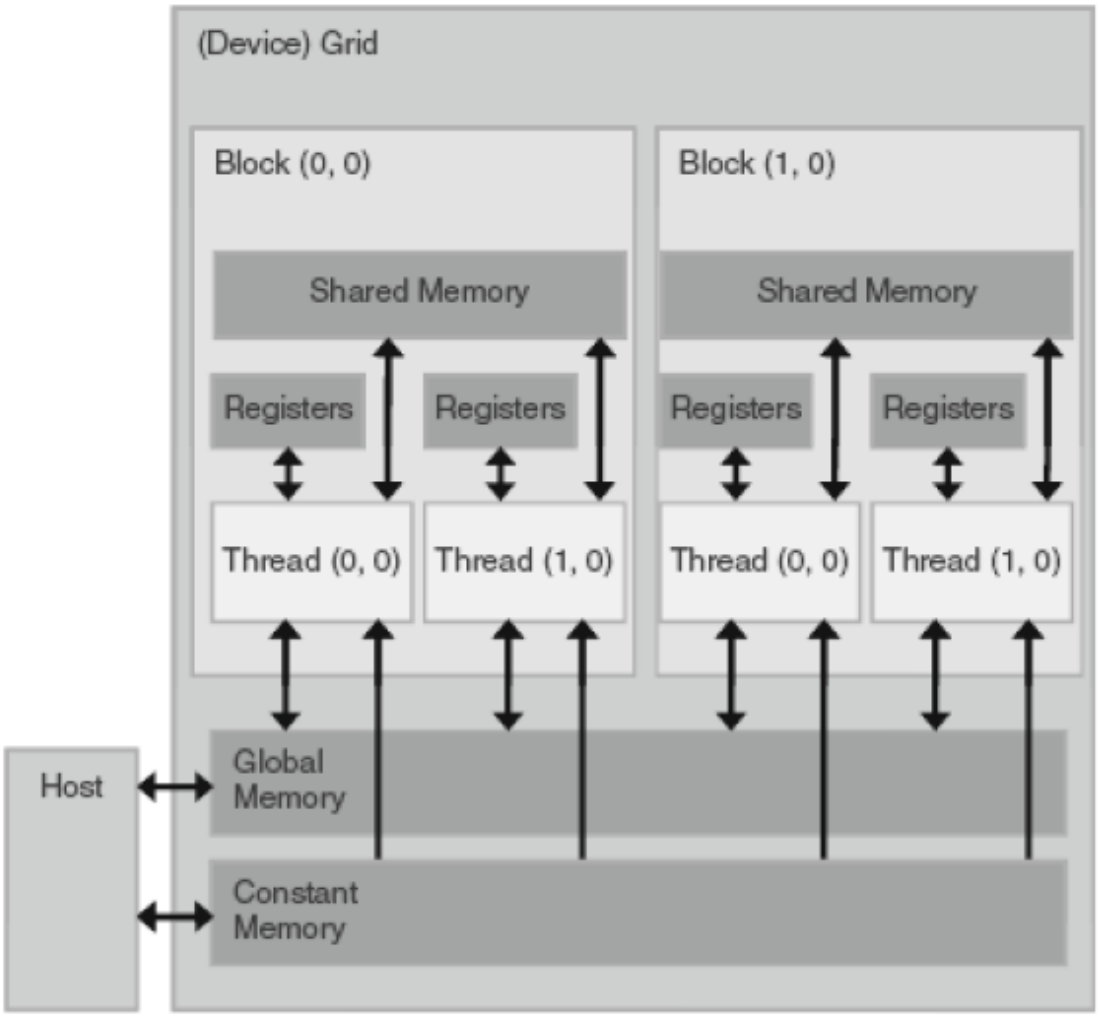

Figura 7.3: Esquema de jerarquía de memorias en una GPU compatible con CUDA.

memoria tiene una elevada latencia, por lo que hay que minimizar su uso desde los kernels. Para intentar acelerar en la medida de lo posible el acceso a esta memoria, es muy recomendable seguir un patrón de acceso coalescente, de forma que hilos consecutivos accederán a posiciones contiguas en memoria. De esta forma, con la lectura de una línea de memoria, podremos proporcionar datos para múltiples hilos

de procesamiento en ejecución. Más adelante explicaremos esta y otras recomendaciones para garantizar el máximo rendimiento.

A continuación vamos a ver un ejemplo de como reservar dinámicamente memoria global de la GPU en nuestros programas:

```
int *dev_a = 0;
// Reserva memoria en la GPU para un vector de enteros.
cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
```

En el ejemplo de arriba, hemos utilizado la instrucción CUDA `cudaMalloc` para reservar en la memoria GPU espacio para un vector de enteros de tamaño *size*. Como primer parámetro recibe la dirección de memoria del puntero que hemos declarado para manejar un vector de enteros en la memoria global de la GPU, *dev_a*. Como segundo parámetro especificamos el tamaño en bytes que queremos reservar. En este caso utilizaremos la función de C/C++ `sizeof(tipo de datos)` para calcular el número de bytes que ocupa cada elemento del tipo de datos que vamos a almacenar, en nuestro caso un valor entero ocupa internamente 4 bytes. El puntero `dev_a` nos servirá para acceder a la memoria declarada desde nuestros kernels CUDA. Una buena práctica es poner delante del nombre de estas variables algo que nos permita identificar rápidamente si la memoria esta reservada en la GPU o la CPU, por ejemplo: `dev_nombrevariable` o `d_nombrevariable`. Este puntero hace referencia a una dirección de memoria en la GPU, este espacio de direccionamiento es distinto al espacio de direccionamiento de la CPU. De esta forma, si intentamos referenciar una dirección de memoria de la GPU desde la CPU, obtendremos un error, ya que esa memoria esta protegida y solo puede ser accedida desde la GPU (por ejemplo, desde un kernel).

La reserva de memoria en la GPU se lleva a cabo desde la CPU. El proceso de reservar memoria es un proceso costoso, el cual además fuerza por defecto a sincronizar las operaciones pendientes en la GPU, de forma que eliminaría cualquier nivel de concurrencia o solapamiento que estuviesemos intentando conseguir entre la CPU y la GPU. Por ello, la reserva o liberación de memoria global se lleva a cabo al principio y al final de nuestros programas respectivamente, evitando reservar memoria en mitad de nuestro código, ya que esto puede causar un gran impacto en el rendimiento de nuestra aplicación. Ya veremos que algo similar ocurre con las transferencias entre memorias, especialmente entre host y device.

Al igual que en C/C++, la memoria dinámica tiene que ser desalojada de forma manual. De esta forma la API de CUDA también dispone del siguiente comando para liberar la memoria reservada una vez terminamos de utilizarla. Al final de nuestro programa liberaremos toda la memoria de la GPU que hayamos reservado de forma dinámica utilizando la siguiente isntrucción:

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

7.3.2. Consultar cantidad memoria global disponible

Utilizando la API de programación CUDA es posible en tiempo de ejecución consultar la cantidad de memoria global disponible, de esta forma podremos consultar en nuestros programas si tenemos suficiente memoria antes de proceder a reservarla. Si intentamos reservar más memoria global de la

disponible en nuestra tarjeta gráfica, la llamada a la función `cudaMalloc` nos devolverá un código de error.

Para obtener la cantidad de memoria llamamos a la función `cudaGetDeviceProperties()`. Esta función nos rellenará una estructura de datos propia de CUDA con información sobre la GPU instalada en nuestro sistema. Para consultar la memoria disponible accedemos al campo `cudaDeviceProp.totalGlobalMem`. Este campo contiene la cantidad de memoria global disponible en bytes.

```
// tamaño total de la memoria global en bytes
size_t totalGlobalMemory;
```

7.3.3. Transferencia de datos entre memoria CPU y memoria global GPU

La memoria global en la GPU suele ser el punto de entrada para alojar los datos que va a utilizar nustra aplicación en tiempo de ejecución. Por ello, una vez hemos reservado memoria global en la GPU, podemos transferir datos en ambas direcciones CPU/GPU utilizando el comando `cudaMemcpy()` de la API de CUDA. La función `cudaMemcpy()` recibe cuatro parámetros de entrada. El primer parámetro es un puntero a la dirección en memoria GPU donde vamos a copiar los datos. El segundo parámetro es un puntero a la dirección de memoria CPU dond se encuentran los datos que vamos a copiar a la memoria GPU. El tercer parámetro especifica el número de bytes a copiar desde la dirección origen a la dirección destino. En este caso, si conocemos el tipo de datos que vamos a utilizar podemos ayudarnos de la función `sizeof(tipo de dato)` para obtener el tamaño del tipo de dato que vamos a copiar. Finalmente, el cuarto parámetro especifica la dirección en la que vamos a llevar a cabo la transferencia de datos. En la API de CUDA encontramos las siguientes opciones:

- `cudaMemcpyHostToDevice` : transferencia de datos de memoria CPU a memoria GPU.
- `cudaMemcpyDeviceToHost` : transferencia de datos de memoria GPU a memoria CPU.
- `cudaMemcpyDeviceToDevice` : transferencia de datos entre dos direcciones de memoria GPU.

El flujo habitual de transferencia de datos entre CPU y GPU se realiza al principio y al final de nuestra aplicación, evitando así transferencias de datos durante la ejecución de nuestra aplicación. Por lo tanto, copiaremos datos de memoria CPU a GPU al princpio de nuestra aplicación, lanzaremos una serie de trabajos (kernels) en la GPU, y finalmente copiaremos los resultados de vuelta a la CPU para su visualización o escritura en disco. Este es el flujo ideal en cualquier aplicación que desarrollemos en la GPU. De hecho, si los resultados computados en los kernels CUDA tienen como único objetivo su visualización, no sería necesario copiar esta información de vuelta a la CPU, se podrían visualizar directamente utilizando por ejemplo el lenguaje OpenGL o DirectX, lenguajes de programción gráfica. Ambos lenguages pueden interoperar con direcciones de memoria reservadas utilizando la API de CUDA, de forma que no requiere copiar de vuelta los datos a la CPU y copiarlos de nuevo a la memoria de la GPU utilizando la API de OpenGL y/o DirectX. CUDA ofrece una serie de mecanismos de comunicación con OpenGL y DirectX, permitiendo al desarrollador bloquear esa porción de memoria hasta que se ha visualizado de forma gráfica. En el capítulo **X** explicaremos este proceso con detalle.

A continuación podemos ver el ejemplo básico de la suma de vectores. Inicialmente copiamos los vectores almacenados en memoria CPU a memoria GPU. Posteriormente ejecutamos el kernel CUDA para computar la suma de ambos, y finalmente copiamos los resultados de vuelta a la memoria de la CPU para su visualización.

```
cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);
addKernel<<<1, size>>>(dev_c, dev_a, dev_b);
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

Es importante saber que la ejecución de la instrucción `cudaMemcpy()` es síncrona. Lo que significa que fuerza a la CPU a esperar a que la GPU finalice todas las tareas que tenga pendientes. Ya veremos que en la API de CUDA existen dos tipos de instrucciones, las síncronas y las asíncronas. Por defecto las instrucciones de copia entre memorias son síncronas y por ejemplo la invocación de kernels es asíncrona. Normalmente existen variaciones de la mayoría de funciones que permiten su ejecución en ambas modalidades, síncronas y no síncronas. En el caso de la función `cudaMemcpy()` encontramos la variante `cudaMemcpyAsync`. Esta instrucción no fuerza a la sincronización de la CPU y la GPU, por lo que es útil para permitir la ejecución solapada de kernels, transferencias entre memorias y trabajo en la CPU. El desarrollador será el encargado de asegurar que no existen dependencias entre las instrucciones asíncronas que vamos a utilizar. La utilización de instrucciones asíncronas, junto con la utilización de `cudaStreams` nos permitirá obtener un mayor rendimiento en nuestra aplicación en aquellos casos en que ciertas tareas se puedan solapar en el tiempo.

7.3.4. Reserva estática de memoria global

También es posible reservar de forma estática memoria global. Para ello podemos utilizar el modificador `__device__` delante la variable que declaremos. De esta forma le estamos indicando al compilador de CUDA que esa variable se va a alojar en memoria global de la GPU. Cuando nuestro programa se ejecute alojará esa memoria de forma automática y no tendremos que encargarnos de liberar la memoria de forma manual, ya que el driver de CUDA lo hará de forma automática.

Podemos copiar información a este espacio de memoria declarado de forma estática utilizando las siguientes instrucciones:

```
cudaError_t cudaMemcpyToSymbol(
  char *symbol,
  const void *src,
  size_t count,
  size_t offset = 0,
  enum cudaMemcpyKind kind = cudaMemcpyHostToDevice
);

cudaError_t cudaMemcpyFromSymbol(
  void *dst,
  char *symbol,
  size_t count,
```

```
  size_t offset = 0,
  enum cudaMemcpyKind kind = cudaMemcpyDeviceToHost
);
```

Mediante estas dos instrucciones podemos copiar y leer, respectivamente, información de estas variables alojadas de forma estática en la memoria de la GPU.

7.3.5. Memoria local

La memoria local es una memoria de tipo stack a nivel de hilo de ejecucion. Cuando un hilo que se ejecuta dentro de un SM ha ocupado el número máximo de registros disponibles en ese SM, se produce lo que se conoce como register spilling, y las variables declaradas que ya no caben en los registros pasan a almacenarse en memoria local. En las primeras version de CUDA la memoria local se implementaba únicamente en memoria global, por lo que cuando esto ocurría, el rendimiento de la aplicación se degradaba de forma considerable. A partir de la arquitectura Fermi, este problema se mejora con la introducción de memorias caches L1 y L2. De forma que estas variables pasan a almacenarse en estas memorias. Estas memorias cache L1 y L2 son bastante más rápidas que la memoria global, reduciendo así el impacto en el rendimiento de nuestra aplicación. Es posible analizar detalladamente el uso de registros y memoria local de nuestra aplicación. Si especificamos el modificador `-Xptxas -v,abi=no` en la compilación de nuestro programa, el compilador imprimirá información sobre la utilización de los registros, así como la memoria local para cada uno de los kernels de nuestro programa.

A continuación vamos a analizar un ejemplo de la salida producida por el compilador cuando especificamos el modificador `-Xptxas -v` (Análisis estático de los kernels CUDA)

```
ptxas info: Compiling entry function _Z17medianFilter2D_smPhS_
ptxas info: Function properties for _Z17medianFilter2D_smPhS_
16 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 27 registers, 16 bytes cumulative stack size,
             324 bytes smem, 328 bytes cmem[0]
```

Podemos ver en la información generada que el kernel CUDA `medianFilter2D_sm` utiliza 27 registros por hilo, 328 bytes en la memoria de constantes y 16 bytes de tamaño de pila. Es importante destacar que en este kernel no se produce la ocupación del número máximo de registros y por lo tanto, no se utiliza la memoria local. Esto empeoraría el rendimiento del kernel. Esta información la extraemos de este mensaje: `0 bytes spill stores, 0 bytes spill loads`. El hecho de ocupar demasiados registros y tener que mover cierta información a memoria local es conocido como 'register spilling'. Aunque en este kernel no utilizamos de forma explicita la memoria de constantes, CUDA la utiliza para copiar la información de los punteros que pasamos como argumentos de la función. Esta información se genera para cada kernel que tengamos en nuestros ficheros CUDA.

También podemos solicitar información sobre la ocupación de la memoria local en tiempo de ejecución mediante el siguiente comando:

```
cuFuncGetAttribute(CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES).
```

Con lo descrito anteriormente podemos ver que la gestión de la memoria local es automática, aunque tendremos que tomar precauciones respecto a su uso, ya que esto puede ocasionar pérdidas de rendi-

miento. Por otro lado, el uso de la memoria local nos permite llevar a cabo transacciones en memoria de forma coalescente, evitando este problema cuando el acceso a memoria global no se produce de forma coalescente.

7.3.6. Registros

La memoria más cercana a los procesadores, de más bajo nivel y con menor latencia que encontramos en la GPU son los registros. En una GPU compatible con CUDA cada SM tiene miles de registros que se reparten entre todos los hilos de ejecución de nuestros kernels. Por ejemplo, en una GPU de arquitectura Kepler (SM 3.0), cada SM contiene 65536 registros o 256 KB de memoria para este propósito.

Podemos saber el número de registros que cada kernel utiliza en tiempo de compilación añadiendo el siguiente modificar durante la compilación de nuestro código: `--ptxas-options --verbose`.

El número de registros utilizado por un kernel afecta al número de hilos que pueden ejecutarse en un multiprocesador de forma paralela. Ajustando el número de registros que dispone cada hilo de ejecución podremos ajustar de forma más específica el rendmiento de ciertos kernels de nuestro código. El número máximo de registros utilizados por cada hilo se puede especificar en tiempo de compilación con el modificador `--ptxas-options --maxregcount N` donde N es el número máximo de registros por hilo.

7.3.7. Memoria de constantes

Esta memoria esta optimizada para accesos de solo lectura de forma simultánea por múltiples hilos en paralelo. La memoria de constantes reside en la memoria device aunque el acceso a la misma se lleva a cabo con instrucciones distintas permitiendo su uso como una memoria cache especializada. Los desarrolladores tienen a su disposición 64 KB que pueden ser utilizados para almacenar constantes. El rendimiento de la memoria de constantes se encuentra muy relacionado con la utilización de esos datos por todos los hilos de un warp, de forma que si todos los hilos de un warp acceden a la misma porción de memoria constante se conseguirá un alto rendimiento. Sin embargo, si varios hilos dentro de un warp acceden a distintas direcciones de memoria constante en un instante de tiempo el rendimiento se degradará siendo mejor opción utilizar otras memorias como la de texturas.

Para declarar en nuestro código una variable como constante utilizaremos el siguiente modificador delante de las variables declaradas: `__constant__`.

La memoria constante puede ser modificada mediante la utilización de instrucciones CUDA de copia de datos entre memorias o accediendo mediante el puntero a memoria GPU desde un kernel. No se debe acceder a memoria de constantes en el mismo kernel que utilizamos para escribir en esta parte de la memoria, ya que CUDA no mantiene coherencia de la misma en tiempo de ejecución.

Para copiar datos a memoria de constante utilizaremos las siguientes instrucciones:

```
// Nos permite copiar datos a memoria de constantes
cudaMemcpyToSymbol()
// Nos permite leer el contenido de una variable alojada
// en memoria de constantes
cudaMemcpyFromSymbol()
```

También podemos leer/modificar la memoria de constantes accediendo al puntero que contiene la dirección de memoria GPU donde se almacena dicha información. Para obtener la dirección de memoria se tiene que utilizar el comando:

```
cudaError_t cudaGetSymbolAddress( void **d_ptr,
                                  char *variable_constante );
```

También podemos obtener en tiempo de ejecución la cantidad de memoria de constantes utilizada mediante el comando `cuFuncGetAttribute(CU_FUNC_ATTRIBUTE_CONSTANT_SIZE_BYTES)`.

A continuación mostramos un ejemplo de programa completo que hace uso de la memoria de constantes.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cmath>
#include <ctime>
#include <cstdlib>
#include <stdio.h>

//Declaración memoria constantes
__constant__ int constant_values[100];

__global__ void test_kernel(int* d_array)
{
  int idx;

  // Calculamos el índice del hilo
  idx = blockIdx.x * blockDim.x + threadIdx.x;

  for (int i = 0; i<100; i++)
    d_array[idx] = d_array[idx] + constant_values[i];
  return;
}

int main(int argc, char** argv)
{
  int size = 100 * sizeof(int);
  int* d_array;
  int h_angle[360];
  int BLOCK_SIZE = 64;
  cudaError_t cudaStatus;

  std::srand(std::time(0));

  // Reserva espacio device memory
```

```
  cudaMalloc((void**)&d_array, sizeof(int)*size);

  // Inicialización memoria device a 0
  cudaMemset(d_array, 0, sizeof(int)*size);

  // Inicializamos en el host la información constante
  for (int i = 0; i<100; i++)
    h_angle[i] = std::rand();

  // Copia datos a memoria constante en CUDA
  cudaMemcpyToSymbol(constant_values,h_angle, sizeof(int)*100);

  test_kernel <<<100/BLOCK_SIZE,BLOCK_SIZE>>>(d_array);

  // Comprueba errores llamada al kernel
  // (se han obviado el resto de comprobaciones)
  cudaStatus = cudaGetLastError();
  if (cudaStatus != cudaSuccess) {
    fprintf(stderr,"Error:%s\n",cudaGetErrorString(cudaStatus));
    return 1;
  }

  // liberamos memoria device
  cudaFree(d_array);
  return 0;
}
```

Es importante destacar algunos aspectos de este programa:

1. Podemos hacer uso de la variable `constant_values` dentro del kernel sin pasarla como argumento, ya que está definida una vez copiada a memoria GPU con el comando `cudaMemcpyToSymbol` esta se encuentra disponible de forma global para todos los kernels.

2. El acceso de los hilos a memoria constante cumple lo explicado anteriormente, todos los hilos acceden a la mismas direcciones de memoria constante, además de hacerlo de forma contigua. De esta forma, hasta 16 hilos (medio warp) consiguen ahorrar 15 lecturas a memoria global: los datos obtenidos mediante una lectura a memoria son reutilizados por varios hilos en ejecución.

7.3.8. Memoria de texturas

En CUDA, cuando trabajamos con la memoria de textura, lo haremos a través de dos objetos o entidades distintas. Por un lado usaremos el tipo de datos `cudaArray` para llevar a cabo la reserva de memoria en la GPU. Por otro lado, utilizaremos un objeto denominado 'referencia a textura', del inglés 'texture reference', el cual contiene información sobre el tipo de información almacenada en la memoria de texturas, como se debe llevar a cabo el acceso a memoria de texturas, así como el acceso a la misma.

Cuando utilizamos un `cudaArray` para acceder a la memoria de texturas, la GPU internamente utiliza una memoria cache de solo lectura, la cual proporciona un mayor rendimiento, especialmente cuando no se producen accesos coalescentes. La memoria de texturas es recomendable para estos casos, donde el acceso a memoria global no se produce de forma coalescente. La memoria de textura es especialmente eficiente cuando tenemos datos de solo lectura sobre los que vamos a realizar operaciones que requieren accesos no coalescentes a memoria, por ejemplo, aplicar distintos filtros o convoluciones 2D sobre una imagen.

También cabe destacar que la memoria de texturas se puede configurar para que el acceso a los datos se haga utilizando interpolación lineal y/o normalización de los datos de forma automática. Estas operaciones de normalización o de acceso a los datos utilizando interpolación bilineal tienen un coste computacional muy bajo gracias a la utilización de la memoria de texturas. Este tipo de memorias son una herencia de las memorias utilizadas comúnmente en lenguajes de programación gráfica como OpenGL.

7.3.9. Memoria compartida

La memoria compartida, del inglés 'Shared memory' es una memoria un tanto especial y diferente respecto a las que hemos visto hasta ahora. La memoria compartida se encuentra dentro de cada SM, y tiene una latencia muy baja. Su principal propósito es el intercambio de información entre hilos dentro de un mismo bloque que se ejecutan en un SM. Aunque esta memoria es realmente rápida, su rendimiento es todavía alrededor de 10 veces más lento que los registros, pero mucho más rápida que la memoria global. Por ello, la utilización de forma combinada de todas las memorias disponibles es la clave para obtener el máximo rendimiento en nuestra aplicación.

La memoria compartida es un recurso esencial que se utiliza para copiar datos que se van a reutilizar dentro de nuestros kernels, por lo que copiaremos la información desde memoria global a memoria compartida y los posteriores accesos se llevarán a cabo a través de la memoria compartida, reduciendo así la latencia causada por un acceso repetido a la memoria global.

La memoria compartida es manejada por los desarrolladores, por lo que la reserva de esta memoria, así como la copia de datos a la misma es tarea del desarrollador. La memoria compartida se puede ver como una memoria caché manejada de forma manual por el desarrollador. Para la utilización de la memoria compartida en nuestros kernel deberemos llevar a cabo los siguientes pasos:

1. Declarar la cantidad de memoria compartida a utilizar por nuestro kernel.

2. Cargar los datos en memoria compartida, normalmente los datos se leen desde memoria global y se almacenen en memoria compartida. Para asegurarnos que todos los hilos han terminado de copiar su porción de los datos se ejecuta una instrucción de sincronización a nivel de bloque de hilos: `__syncthreads()`.

3. A continuación llevamos a cabo el procesamiento requerido, cálculos, leyendo los datos almacenados en la memoria compartida.

4. Finalmente deberemos sincronizar `__synchtreads()` de nuevo los hilos antes de empezar a escribir en memoria para asegurarnos que todos los hilos del bloque han terminado de llevar a cabo los cálculos requeridos por nuestro kernel. Tras esto, escribiremos los resultados en memoria

global para poder acceder posteriormente a ellos, ya que la memoria compartida no será accesible tras finalizar la ejecución de nuestro kernel.

Finalmente deberemos sincronizar, `__synchtreads()`, de nuevo los hilos antes de empezar a escribir en memoria para asegurarnos que todos los hilos del bloque han terminado de llevar a cabo los cálculos requeridos por nuestro kernel. Tras esto, escribiremos los resultados en memoria global para poder acceder posteriormente a ellos, ya que la memoria compartida no será accesible tras finalizar la ejecución de nuestro kernel.

Podemos declarar memoria compartida en nuestro kernel de dos formas distintas. Utilizando el modificador `__shared__` delante de las variables que definamos en nuestro kernel. Podemos ver un ejemplo a continuación:

```
/* CUDA Kernel 2D shared memory */
__global__ void medianFilter2D_sm(unsigned char *d_output,
                                  unsigned char *d_input)
{
  // declaración de memoria compartida para
  // almacenar una porción de una imagen
  __shared__ unsigned char d_input_sm[(BLOCK_H)*(BLOCK_W)];
  ...
}
```

La otra forma de declarar memoria compartida en nuestro kernel consiste en declarar variables sin tamaño y especificar el tamaño de memoria compartida que necesitamos en nuestro kernel a la hora de invocarlo. Ejemplo:

```
/* CUDA Kernel 2D shared memory */
__global__ void medianFilter2D_sm(unsigned char *d_output,
                                  unsigned char *d_input)
{
  extern __shared__ unsigned char d_input_sm[];
  ...
}

...

// llamando al kernel
medianFilter2D_sm<<< gridSize, blockSize,
            (BLOCK_H)*(BLOCK_W)*sizeof(char) >>>(...);
```

Como podemos ver, el tamaño de la memoria compartida se especifica cuando invocamos el kernel, de forma que cada bloque tendrá asignada esa capacidad de memoria compartida: `(BLOCK_H)*(BLOCK_W)*sizeof(char)`. Además dentro del kernel declaramos variables para manejar esa porción de memoria compartida utilizando los modificadores extern y `__shared__`.

Al igual que vimos anteriormente con la memoria local y con los registros, podemos consultar la cantidad de memoria compartida empleada en nuestro código con el

modificador `-Xptxas -v,abi=no`. y en tiempo de ejecución utilizando la instrucción `cuFuncGetAttribute(CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES)`.

7.3.10. Ejemplo convolución 1D usando memoria compartida

A continuación vamos a analizar un ejemplo completo donde el uso de la memoria compartida permite optimizar el tiempo invertido en acceder a memoria global en la GPU. La memoria compartida se puede aprovechar de igual manera en otros ejemplos muy similares que usan la misma estrategia (subivisión del problema en partes, en inglés, tiling), como la convolución 2D, o la multiplicación de matrices. Esta estrategia, se basa en copia de los datos del subproblema (tile) a memoria compartida. Normalmente estos datos son accedidos de forma repetido y solapada por los hilos dentro de un mismo bloque, por lo que copiando estos datos a memoria compartida reducimos de forma drástica el número de accesos necesarios a memoria global.

Para entender mejor esta estrategia vamos a resolver el problema de aplicar una convolución 1D utilizando memoria compartida. La descripción formal del operator convolución es la siguiente: La convolución es un operador matemático que transforma dos funciones f y g en una tercera función que en cierto sentido representa la magnitud en la que se superponen f y una versión trasladada e. En la Figura 7.4 podemos ver un ejemplo de aplicar la función g, también conocida como máscara o filtro, sobre los datos originales o función f generando una nueva versión trasladada e.

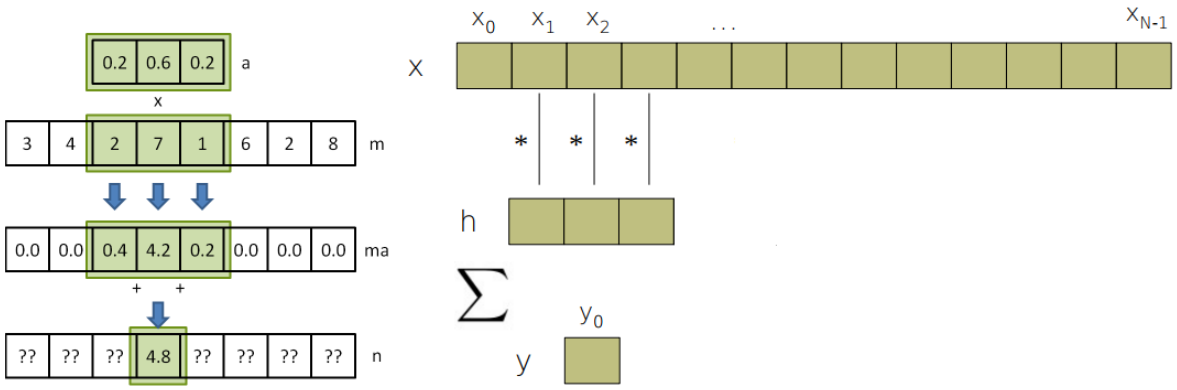

Figura 7.4: Ejemplo operador convolución en 1D (máscara de tamaño 3).

Primero vamos a analizar la versión secuencial del operador convolución, tal y como la mostramos a continuación. Esta tiene una complejidad $O(n * m)$, donde n es el número de elementos del vector de entrada y m es el tamaño del filtro. Aunque se pueden llevar a cabo algunas optimizaciones utilizando instrucciones de vectorización del procesador, la capacidad de cómputo de las GPUs permiten obtener una aceleración todavía mayor.

```
void conv_1d_cpu(float *input, float *d_mask,
             float *d_output, int vector_size)
{
  int index;
  for (int i = 0; i < vector_size; i++)
  {
    float new_value = 0.0f;
    // conv 1D
        for (int j = 0; j<mask_width; j++)
        {
          index = i + j - mask_radius;
          if(index >= 0 && index < vector_size)
            new_value += (input[index] * d_mask[j]);
          }
          // Escritura memoria global
          d_output[i] = new_value;
    }
}
```

A continuación vamos a ver una primera versión paralela en CUDA haciendo uso de la memoria global:

```
__global__ void conv_1d(float *d_input, float *d_mask,
                        float *d_output, int vector_size)
{
  // indice acceso entrada/salida
  int idx_input = blockIdx.x * blockDim.x + threadIdx.x;
  float new_value = 0.0;
  int index;

  // Convolución 1D y escribimos valor a memoria global
  if (idx_input<vector_size)
  {
          // conv 1D
          for (int i = 0; i<mask_width; i++)
          {
      index = idx_input + i - mask_radius;
          if (index>=0 && index<vector_size)
          new_value += (d_input[index] * d_mask[i]);
        }
        // Escritura memoria global
        d_output[idx_input] = new_value;
  }
}
```

Esta versión es poco eficiente, ya que varios hilos de un mismo bloque llevan a cabo varias operaciones de lectura en memoria global sobre las mismas direcciones: `new_value += (d_input[index] * d_mask[i]);`. En el caso de la convolución, un mayor tamaño de filtro causa un mayor solape en el acceso a memoria, por lo que el rendimiento que obtendremos al usar la memoria compartida será también mayor cuando el tamaño del filtro se incremente.

Finalmente, a continuación vamos a ver la implementación del convolución 1D haciendo uso de la memoria compartida.

```
1  const int mask_width = 5; // tamaño mascara convolucion
2  const int num_elements = 128; // num elementos
3  const int mask_radius = mask_width/2;
4  const int tile_size = 32; // tamaño subdivisión problema
5  // tamaño de bloque teniendo en cuenta
6  // tamaño del tile y el halo del mismo
7  const int block_size_tiling = tile_size + (mask_radius*2);
8
9  __global__ void conv_1d_sm(float *d_input, float *d_mask,
10                             float *d_output, int vector_size)
11 {
12   // Declaramos memoria compartida
13   __shared__ float input_sm[block_size_tiling];
14
15   // indice acceso vector salida
16   // usamos tile_size, sin tener encuenta el halo
17   int idx_output = blockIdx.x * tile_size + threadIdx.x;
18
19   // indice acceso vector entrada
20   int idx_input = idx_output - mask_radius;
21
22   float new_value = 0.0;
23
24   // Cargamos vector de entrada en memoria compartida (tile)
25   if (idx_input >= 0 && idx_input<vector_size)
26     input_sm[threadIdx.x] = d_input[idx_input];
27   else
28     input_sm[threadIdx.x] = 0.0f;
29
30   // sincronizamos hilos del bloque
31   // aseguramos que todos los hilos han copiado su valor
32   __syncthreads();
33
34   // Convolución 1D y escribimos valor a memoria global
35   if (threadIdx.x<tile_size)
```

```
36    {
37      // conv 1D
38      for (int i = 0; i<mask_width; i++)
39      {
40        new_value += (input_sm[threadIdx.x + i]*d_mask[i]);
41      }
42      // Escritura memoria global
43      d_output[idx_output] = new_value;
44    }
45  }
```

Como podemos observar, se diferencian dos etapas en el kernel CUDA. Una primera etapa donde los hilos copian los datos a memoria compartida, líneas 24-28. (Normalmente cada hilo se encarga de copiar un trozo de memoria global a memoria compartida). Una vez copiados los datos, se ejecuta la instrucción `__syncthreads();` de forma que nos aseguramos que todos los hilos han terminado de copiar la información. A partir de este momento, los hilos pueden llevar a cabo de forma segura las operaciones sobre la memoria compartida (Líneas 38-41). Finalmente, se escribe el resultado en memoria global (Línea 43). El aspecto más importante a destacar en esta implementación para uso de memoria compartida, es la necesidad de invocar más hilos que el número de elementos a procesar. Esto se debe a que en los límites de cada tile o subcobjuntos de datos, tendremos que copiar información de los tiles adyacentes. Por ello, el tamaño de bloque se calcula de la siguiente forma: `int block_size_tiling = tile_size + (mask_radius*2);`. Donde al tamaño del tile o subbloque, le añadimos el tamaño adicional de dos veces el radio del filtro o máscara a aplicar. Para el caso de una máscara de tamaño 5, el radio es igual a 2, teniendo que añadir 4 elementos adicionales de procesamiento para la copia de datos a memoria compartida.

7.4. Consideraciones de rendimiento en el uso de memorias GPUs

Como ya nos podemos imaginar, la memoria va a jugar un papel clave en los algoritmos que implementemos en la GPU. Aunque la combinación del uso de las distintas memorias (registros, memoria compartida, textura, ...) nos va a permitir reducir el número de accesos a la GPU, tenemos que tener cuidadode no excedernos tampoco en el uso de estas memorias, ya que también van a producir una reducción de la ocupación de los procesadores de nuestra GPU. Es decir, cada dispositivo CUDA ofrece una cantidad limitada de memoria CUDA, la cual nos va limitar el número máximo de bloques e hilos que pueden estar ejecutándose de forma simultánea en los SMs. Por ejemplo, en la reciente NVIDIA GTX 1080, cada multiprocesador (SM) dispone de las siguientes características:

- 128 cores y 64K de registros
- 96 KB de memoria compartida
- 48 KB cache L1
- 16 KB cache para constantes

- hasta 2048 hilos por SM

Si tomamos como ejemplo el número de registros, nos damos cuenta de que cada hilo solo puede utilizar un número reducido de ese total de aproximadamente 64000 registros. Dado que para ocupar la capacidad de cómputo total de cada SM deberíamos ejecutar 2048 hilos a la vez, y que cada SM dispone de un máximo de 64000 registros, nos da alrededor un máximo de 30 registros por hilo. Si nuestros hilos necesitan utilizar más registros provocará que la capacidad de cómputo del SM se vea mermada al tener que mover datos a memorias más lentas, y por lo tanto no estemos aprovechándo todo el potencial de cómputo. Algo similar ocurre con la memoria compartida, de la que solo disponemos una cantidad limitada por SM. Por ello, será necesario hacer un uso equilibrado de las diferentes memorias en la GPU para maximizar el rendimiento de nuestra aplicación.

7.5. Exprimiendo el ancho de banda de la memoria global

El acceso a la memoria global es uno de los principales causantes de degradamiento del rendimiento de una aplicación en la GPU. Debido a que normalmente las aplicaciones CUDA requieren procesar datos de forma masiva, es normal que se requiera acceder a grandes cantidades de información de la memoria global. Por ello, como ya hemos discutido haremos uso de la memoria de compartida y de los registros para aliviar estas latencias. Sin embargo, todavía es necesario mover esos datos desde memoria global a estas memorias más rápida y por lo tanto este movimiento de datos debe hacerse de la forma más eficiente posible.

La memoria global se encuentra implementada normalmente en memorias DRAMs (Dynamic Random Access Memory). La lectura de este tipo de memorias se lleva a cabo por líneas, de forma que cuando se lee un dato, también se leen las posiciones contiguas al dato que hemos leido. Teniendo en cuenta esta consideración, si queremos obtener el máximo rendimiento a la hora mover datos de la memoria global a otras memorias, será necesario implementar un acceso coalescente a memoria. Un acceso coalescente significa que los hilos en ejecución accederán a posiciones contiguas de memoria en el mismo instante de tiempo, de forma que mediante una única lectura de memoria global podemos alimentar varios hilos en ejecución. La Figura 7.5 muestra un gráfico de accesos coalescentes. Posteriormente veremos varios kernels que implementan las situaciones representadas en la Figura 7.5.

Ejemplo de kernel acceso coalescente a memoria global

```
__global__ coalesced_access(float*x)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  x[tid] = threadIdx.x;
}
```

Ejemplo de kernel acceso no coalescente a memoria global

```
__global__ not_coalesced_access(float*x)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  x[tid*100] = threadIdx.x;
}
```

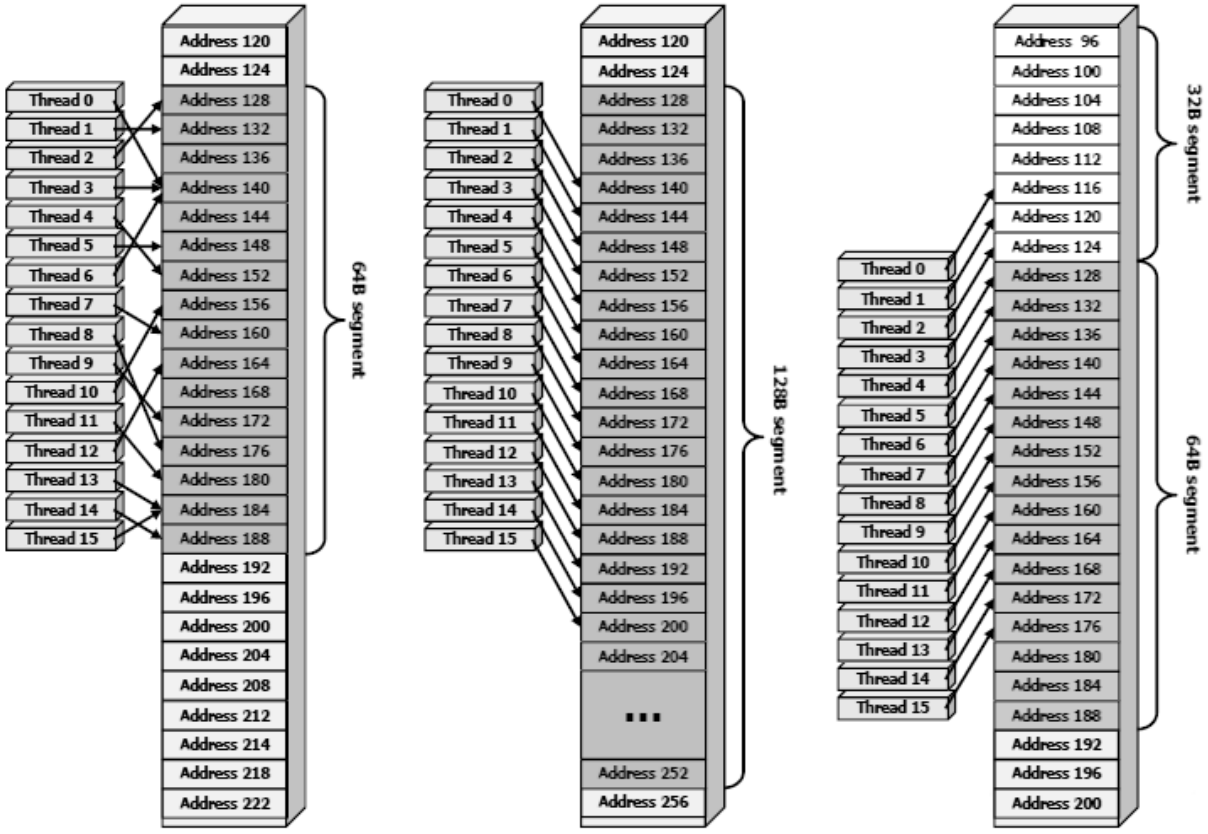

Figura 7.5: Diagramas de ejemplo de acceso a memoria: no coalescente y coalescente.

La única forma de conseguir el máximo ancho de banda a la hora de transferir datos entre memoria global y otras memorias será mediante la correcta utilizando de patrones de acceso coalescente a memoria global, en caso contrario, estaremos limitando nuestra aplicación y no conseguiremos obtener un rendimiento máximo en la transferencia de datos.

Capítulo 8

Problemas 4: Memoria en CUDA

Contenido

8.1. Problema 1

Considera el cálculo suma de matrices en el que cada elemento de la matriz resultado es la suma de los elementos correspondientes de las matrices de entrada. ¿Es posible utilizar memoria compartida para reducir el acceso a memoria global?. Pista: Analiza los elementos accedidos por cada hilo. ¿Existe algún patrón de acceso entre los hilos?

8.2. Problema 2

Considera el cálculo multiplicación de matrices. ¿Es posible utilizar memoria compartida para este caso, reduciendo el acceso a memoria global?

8.3. Problema 3

¿Qué tipo de comportamiento incorrecto puede ocurrir si olvidamos utilizar la instrucción `__syncthreads()` en el kernel que se muestra a continuación?. Existen dos llamadas a la función `__syncthreads()` justifica cada una de ellas.

```
__global__
void matrix_mul_kernel(
        float* Md, float* Nd, float* Pd, const int cWidth)
{
  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx_ = blockIdx.x;
  int by_ = blockIdx.y;
  int tx_ = threadIdx.x;
  int ty_ = threadIdx.y;

  int row_ = by_ * TILE_WIDTH + ty;
  int col_ = bx_ * TILE_WIDTH + tx;

  float p_value_ = 0.0f;

  for (int m = 0; m < cWidth / TILE_WIDTH; ++m)
  {
    Mds[ty_][tx_] = Md[row_ * cWidth + (m * TILE_WIDTH + tx_)];
    Nds[ty_][tx_] = Nd[(m * TILE_WIDTH + ty_) * cWidth + col_];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k)
      p_value_ += Mds[ty_][k] * Nds[k][tx_];

    __syncthreads();
  }

  Pd[row_ * cWidth + col_] = p_value_;
}
```

Bibliografía

[1] Better performance at lower occupancy. http://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf. Accedida: 07-11-2017.

[2] Geforce 256. http://www.nvidia.com/page/geforce256.html. Accedida: 22-10-2017.

[3] Geforce 256. http://www.nvidia.es/page/geforce3.html. Accedida: 22-10-2017.

[4] Geforce 8800. http://www.nvidia.com/page/geforce_8800.html. Accedida: 29-10-2017.

[5] Nvidia geforce gtx 980, featuring maxwell, the most advanced gpu ever made. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF. Accedida: 07-11-2017.

[6] Nvidia kepler gk110 architecture whitepaper. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. Accedida: 07-11-2017.

[7] Nvidia smi documentation. http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf. Accedida: 07-11-2017.

[8] Nvidia tesla p100, the most advanced datacenter accelerator ever built featuring pascal gp100, the world's fastest gpu. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. Accedida: 07-11-2017.

[9] Nvidia tesla v100 gpu architecture the world's most advanced data center gpu. http://www.nvidia.com/object/volta-architecture-whitepaper.html. Accedida: 07-11-2017.

[10] Nvidia turing gpu architecture: Graphics reinvented. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. Accedida: 08-11-2018.

[11] Techical brief: Nvidia nfinitefx engine. https://www.evga.com/articles/images/11vertexshaders.pdf. Accedida: 26-10-2017.

[12] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[13] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[15] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[16] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[17] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[18] Nicholas Wilt. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[19] Craig M Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, 31(2):50–59, 2011.