Computación de alto desempeño en GPU

María Fabiana Piccoli

Computación de alto desempeño en GPU

cacic2011 XVII CONGRESO ARGENTINO DE CIENCIA DE LA COMPUTACIÓN

XV ESCUELA INTERNACIONAL DE INFORMATICA

FACULTAD DE INFORMÁTICA
Universidad Nacional de La Plata

Gil Costa, Graciela Verónica
Consultas sobre espacios métricos en paralelo. - 1a ed. -
La Plata: Universidad Nacional de La Plata, 2011.
136 p.; 24x16 cm.

ISBN 978-950-34-0721-9

1. Estrategias. 2. Procesamiento. 3. Web. I. Título
CDD 005.3

Computación de alto desempeño en GPU

María Fabiana Piccoli

Coordinación Editorial: Anabel Manasanch
Corrección: María Eugenia López, María Virginia Fuente, Magdalena Sanguinetti y Marisa Schieda.
Diseño y diagramación: Ignacio Bedatou | Andrea López Osornio

Editorial de la Universidad Nacional de La Plata (Edulp)
47 N° 380 / La Plata B1900AJP / Buenos Aires, Argentina
+54 221 427 3992 / 427 4898
editorial@editorial.unlp.edu.ar
www.editorial.unlp.edu.ar

Edulp integra la Red de Editoriales Universitarias (REUN)

Primera edición, 2011
ISBN Nº 978-950-34-0721-9

Impreso en Argentina

Índice

Prólogo

La constante demanda de mayores prestaciones hizo que la industria de los *monoprocesadores* se encontrara en una situación límite respecto al cumplimiento de la ya conocida *Ley de Moore* sobre rendimiento del hardware. Como los *monoprocesadores* alcanzaron su rendimiento máximo, se debió buscar otras alternativas. Una de ellas fueron los *multiprocesadores*, computadoras de propósito general con dos o más núcleos.

Al mismo tiempo que los *multiprocesadores* eran aceptados por la sociedad en general y ante el auge de la industria de los videojuegos, grandes y relevantes avances tecnológicos fueron hechos en las GPU (*Graphic Processing Units*) con el objetivo de liberar a la CPU del proceso de *renderizado* propio de las aplicaciones gráficas. La gran demanda de gráficos de alta calidad motivó el incremento de la potencia de cálculo transformando a las GPU en potentes co-procesadores paralelos. Si bien su origen fue brindar asistencia en aplicaciones gráficas, su uso como co-procesador paralelo de la CPU para resolver aplicaciones de propósito general constituye uno de los tópicos más actuales en la computación de alto desempeño. Si se observa la lista de computadoras más poderosas en el TOP500, la mayoría son una combinación de elementos *multicore* (CPU) y *many-cores* (GPU).

A partir del 2005, se comenzó a utilizar la gran potencia de cálculo y el alto número de procesadores de las GPU como arquitectura masivamente paralela para resolver tareas no vinculadas con actividades gráficas, es decir utilizarlas en *programación de aplicaciones de propósito general (GPGPU).* En este ámbito surgieron varias técnicas, lenguajes y herramientas para la programación de GPU como co-procesador genérico a la CPU. La evolución de éstas fue tan rápida como su popularización. Una de las herramientas más difundidas es CUDA (*Compute Unified Device Architecture*), desarrollada por Nvidia para sus GPU desde de la generación G80.

No todas las aplicaciones son adecuadas para el desarrollo en la GPU, para que las implementaciones sean exitosas deben cumplir con

ciertas condiciones, las más relevantes son: tener un alto grado de paralelismo de datos y gran cantidad de cálculos aritméticos independientes, poder tomar ventaja de los anchos de banda y la jerarquía de memoria de la GPU y requerir una mínima interacción entre la CPU y la GPU.

Este libro es el resultado del trabajo de investigación sobre las características de la GPU y su adopción como arquitectura masivamente paralela para aplicaciones de propósito general. Su propósito es transformarse en una herramienta útil para guiar los primeros pasos de aquellos que se inician en la computación de alto desempeños en GPU. Pretende resumir el estado del arte considerando la bibliografía propuesta.

El objetivo no es solamente describir la arquitectura *many-core* de la GPU y la herramienta de programación CUDA, sino también conducir al lector hacia el desarrollo de programas con buen desempeño.

El libro se estructura de la siguiente manera:

Capítulo 1: se detallan los conceptos básicos y generales de la computación de alto rendimiento, presentes en el resto del texto.

Capítulo 2: describe las características de la arquitectura de la GPU y su evolución histórica. En ambos casos realizando una comparación con la CPU. Finalmente detalla la evolución de la GPU como co-procesador para el desarrollo de aplicaciones de propósito general.

Capítulo 3: este capítulo contiene los lineamientos básicos del modelo de programación asociado a CUDA. CUDA provee una interfaz para la comunicación CPU-GPU y la administración de los threads. También se describe las características del modelo de ejecución SIMT asociado.

Capítulo 4: analiza las propiedades generales y básicas de la jerarquía de memoria de la GPU, describiendo las propiedades de cada una, la forma de uso y sus ventajas y desventajas.

Capítulo 5: comprende un análisis de los diferentes aspectos a tener en cuenta para resolver aplicaciones con buena performance. La programación de GPU con CUDA no es una mera transcripción de un código secuencial a un código paralelo, es necesario tener en cuenta diferentes aspectos para usar de manera eficiente la arquitectura y llevar a cabo una buena programación.

Finalmente se incluyen tres apéndices. En el primero se describen los calificadores, tipos y funciones básicos de CUDA, el segundo detalla algunas herramientas simples de la biblioteca `cutil.h` para el control de la programación en CUDA. El último apéndice describe las capacidades de cómputo de CUDA para las distintas GPU existentes, listando los modelos reales que las poseen.

CAPÍTULO 1

Introducción a la Computación de Alto Desempeño

La constante demanda de mayor poder computacional, no sólo de la comunidad científica y académica, ha determinado grandes avances en el hardware de computadoras, hoy en día es muy difícil encontrar computadoras con un único procesador. El software no fue ajeno a estos avances, cada día se proponen más herramientas, metodologías y técnicas que facilitan la tarea de los desarrolladores de software paralelo y les permiten obtener buen desempeño en sus aplicaciones.
Los Sistemas Paralelos implican simultaneidad espacial y temporal en la ocurrencia de eventos, todos relacionados e involucrados en la resolución de un único problema. Diseñar programas paralelos no es una tarea fácil, si bien actualmente existen herramientas que ayudan en los desarrollos, se debe tener en cuenta varias características propias de paralelismo, las cuales determinarán el desempeño o performance del sistema. Al igual que en la programación secuencial, para lograr buenos programas paralelos es necesario seguir una metodología que permite su desarrollo asegurando un software portable, escalable y con buen rendimiento.
En este capítulo se exponen los conceptos básicos de un sistema paralelo, mostrando las características básicas del hardware y el software paralelo. En este último caso, se describen los distintos paradigmas y qué se debe tener en cuenta para el desarrollo de software paralelo.

1.1. Sistemas Paralelos

Las ventajas que puede ofrecer el procesamiento paralelo de un problema son muchas cuando se cuenta con el problema apto para ser resuelto de esta manera. No siempre esto es suficiente, la tarea se torna ardua si no se tiene el software adecuado para la paralelización y una arquitectura de hardware eficiente en comunicaciones.

La construcción de una aplicación en paralelo no siempre deriva naturalmente de la aplicación secuencial. En toda aplicación paralela se deben considerar especialmente las comunicaciones existentes entre los distintos procesos, las cuales pueden conducir a una mala performance del sistema completo.
En Sistemas Paralelos (Akl, 1989) (Takis, 1995) podemos distinguir dos conceptos: el Software Paralelo y la Computadora Paralela. El primero se caracteriza por el procesamiento de la información, enfatizando la manipulación concurrente de los elementos de datos pertenecientes a uno o más procesos, quienes colaboran en la resolución de un único problema. El segundo se refiere a una arquitectura con múltiples unidades de procesamiento, capaz de soportar procesamiento paralelo (Quinn, 1994).
Es claro entonces, la existencia en todo sistema paralelo de dos aspectos bien diferenciados, el Hardware y el Software.

1.1.1. Hardware de un Sistema Paralelo

Por definición una arquitectura paralela es aquella que cuenta con varias unidades de procesamiento y permite el procesamiento paralelo. A través de varias configuraciones se pueden obtener distintas arquitecturas paralelas. La disposición de las conexiones entre los procesadores y la memoria determina el origen de distintas arquitecturas, las cuales serán más adecuadas para resolver ciertas aplicaciones a otras.
Existen distintas clasificaciones para el hardware de una computadora. Una de las más conocidas es la taxonomía de Flynn (Flynn M., 1995) (Flynn M. R., 1996). Para su clasificación, Flynn consideró dos aspectos: *el flujo de instrucciones* y *el flujo de datos*, considerando la multiplicidad de cada uno, arribó a la siguiente clasificación:

11. *SISD*: Son las computadoras con *un único* flujo de instrucciones y *un único* flujo de datos. En este grupo se encuentran todas las computadoras con una única CPU.
    La organización SISD abarca la mayoría de las computadoras secuenciales. Las instrucciones se ejecutan secuencialmente, aunque pueden estar superpuestas en la etapa de ejecución. La mayoría de los monoprocesadores SISD usan esta técnica para las diferentes etapas de ejecución de sus instrucciones. Todas las unidades funcionales están bajo el control de una única unidad de control.
12. *SIMD*: En este grupo se incluyen aquellas computadoras con *un único* flujo de instrucciones y *múltiple* flujo de datos. Estas arquitecturas cuentan con varias CPU, donde todas ejecutan el mismo flujo de instrucciones sobre distintos datos.

La clase SIMD engloba a los procesadores vectoriales. En estas arquitecturas existen diferentes unidades de procesamiento supervisadas por una única unidad de control. Todos los elementos de procesamiento reciben, de la unidad de control, la orden de ejecutar la misma instrucción sobre diferentes conjuntos de datos procedentes de distintos flujos de datos.

13. *MISD*: Aquí están las computadoras con *múltiple* flujo de instrucciones y *un único* flujo de datos.
    En este caso, las*n* unidades de procesamiento reciben diferentes instrucciones, las cuales operan sobre el mismo flujo de datos. Ninguna de las arquitecturas existentes implementa este tipo de computadoras.
14. *MIMD*: En este caso, las computadoras tienen *múltiple* flujo de instrucciones y *múltiple* flujo de datos. Son todas unidades de procesamiento independientes con sus propios flujos de instrucciones y de datos. Podemos encontrar dos tipos de MIMD, los Multiprocesadores, arquitecturas con memoria compartida, y las Multicomputadoras, arquitectura con memoria distribuida.
    La mayoría de los sistemas de multiprocesadores y multicomputadoras están dentro de la categoría MIMD. Una arquitectura MIMD se caracteriza, además, por ser *fuertemente acoplada* si el grado de interacción entre los procesadores es alto. Por el contrario, se considera *débilmente acoplada* si la interacción es poco frecuente. La mayoría de las computadoras MIMD comerciales están dentro de ésta última categoría. SIMD se lo puede considerar como un caso especial de MIMD.

Otra taxonomía existente, y una de las más populares, es aquella derivada de considerar a la memoria y la forma de comunicación entre los distintos procesadores como las características discriminantes. En esta clasificación surgen dos posibles y grandes grupos: *los multiprocesadores con memoria compartida* y *los multiprocesadores con memoria distribuida*. En el primer grupo están todas las arquitecturas integradas por un conjunto de unidades de procesamiento (procesadores o *cores*), los cuales comparten toda la memoria de la arquitectura paralela. Al segundo grupo, en cambio, pertenecen todas aquellas arquitecturas paralelas formadas por varios procesadores y sus memorias conectados mediante una red de interconexión. Cada procesador posee una memoria local y se comunica con el resto por medio de mensajes enviados a través de la red. Estas últimas arquitecturas son también conocidas como multiprocesadores con pasaje de mensaje o multicomputadoras. Una arquitectura que combine ambas filosofías de diseño también es

válida. En los siguientes párrafos se realiza una explicación más detallada de estas arquitecturas.
Una manera natural para extender el modelo de único procesador es conectar múltiples procesadores (P_i) a múltiples módulos de memoria (M_j), de manera tal que cada procesador puede acceder a cualquier módulo de memoria. La figura 1.1 muestra dicha disposición del hardware. La conexión entre los procesadores y la memoria es a través de alguna red de interconexión. Un multiprocesador con memoria compartida aplica el concepto de *espacio único de direcciones*. Esto implica que toda posición de memoria es única y forma parte de un sistema de memoria principal. Todos los procesadores pueden acceder a cualquier dirección de memoria, de la misma forma que lo haría un procesador en su memoria local.

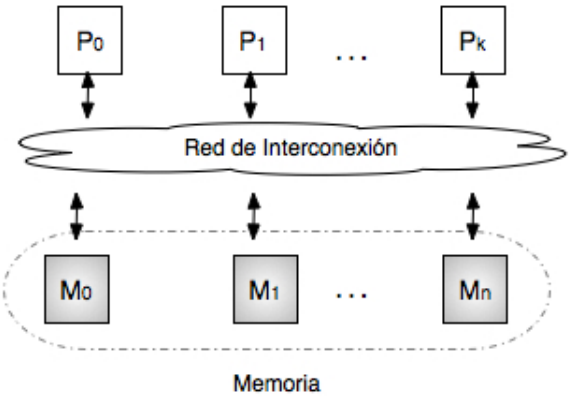


Figura 1.1. Arquitectura de una máquina paralela con memoria compartida

Ejemplos reales de este tipo de arquitectura son los procesadores con varios cores (dual, quad, i3, i7, xeon E7, etc.), las unidades de procesamiento gráfico (GPU), entre otros.
Una alternativa a las computadoras de memoria compartida, especialmente construidas, son como se mencionó anteriormente, las computadoras con memoria distribuidas. Este tipo de computadoras paralelas se forma conectando varias computadoras completas a través de una red de interconexión, como lo muestra la figura 1.2. Cada procesador paralelo es un procesador con memoria local. A esta memoria sólo puede acceder el procesador local. En esta arquitectura, la memoria de la máquina paralela está distribuida entre los distintos procesadores. La red de interconexión permite a los procesadores comunicarse enviando o recibiendo mensajes a otros procesadores en la máquina. Los mensajes pueden incluir datos requeridos por otros procesadores. La comunicación a través de mensajes debe ser

explícitamente especificada. Las computadoras paralelas con pasaje de mensajes son físicamente escalables.

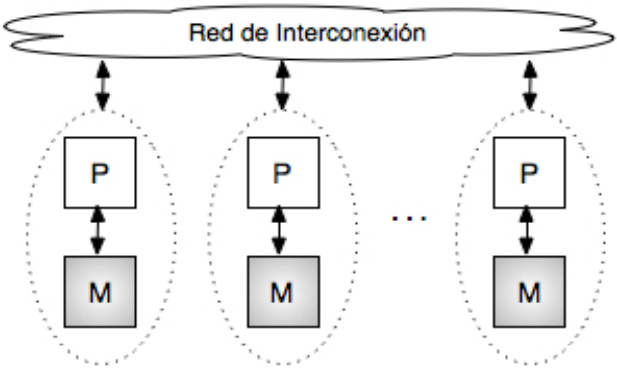


Figura 1.2. Arquitectura de una máquina paralela con pasaje de mensajes

El ejemplo más popular de este tipo de arquitecturas son los cluster de computadoras. Entre las multicomputadoras conocidas se encuentra la *Blue Gene* (Supercomputer, 2011).

Las arquitecturas de *Memoria distribuida Compartida* son aquellas que combinan ambas disposiciones, ver figura 1.3. La arquitectura está formada por procesadores con su espacio de memoria. Cada procesador tiene acceso a un único espacio de memoria. Si la dirección a acceder no es local, el procesador utiliza el pasaje de mensajes para comunicarse con el procesador, el cual tiene los datos como locales. Estas máquinas son comúnmente denominadas máquinas con *Memoria Compartida Virtual*.

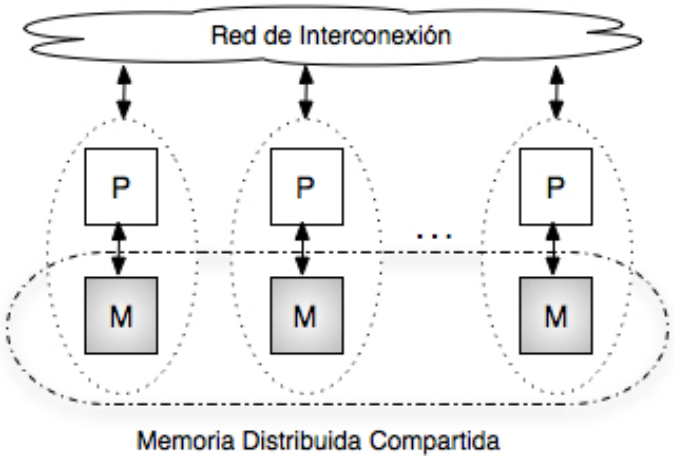


Figura 1.3. Arquitectura de una máquina paralela con memoria compartida distribuida

Una computadora con este tipo de arquitectura es la *Pleidades* (Supercomputer, 2011), la cual está constituida por una SGI Altix ICE 8200EX.

1.1.2. Software de un Sistema Paralelo

La construcción del software paralelo se basa en la siguiente idea: Un problema grande es dividido en subproblemas cada uno de los cuales se resuelve en forma concurrente al resto. Intuitivamente es de esperar un mejor desempeño de la versión paralela del problema respecto a la versión secuencial, resolver los subproblemas independientes en simultáneo hace pensar en obtener mayor rapidez respecto a resolver todo el problema secuencialmente (Blelloch, 1996) (Wilkinson, 1999). Si bien los programas paralelos surgen con el objeto de optimizar la performance en la resolución de un problema, el diseño en general es mucho más complejo.

Desear e implementar algoritmos paralelos que ejecuten sobre máquinas paralelas no es una tarea fácil. No existe una única solución paralela. La naturaleza del problema, la relación entre los datos y la independencia o dependencia de los resultados, son determinantes para obtener multiplicidad de algoritmos paralelos para resolver un mismo problema. Además surgen otros aspectos que se deben considerar:

- ¿Cómo dividir el trabajo en tareas?
- ¿Cómo asignar tareas a los distintos procesadores?
- ¿Cómo se relacionan los procesadores? ¿Están agrupados?
- ¿Cómo se organizan las comunicaciones?

Cada uno de estos aspectos, si bien parecen independientes unos de otros, las decisiones adoptadas en uno influyen directamente en los otros. Además, la manera de encarar a cada uno hace que surjan distintos enfoques de computación paralela.

Desarrollar software paralelo con el único objetivo de mejorar el tiempo de resolución de un problema, implica no sólo un gran esfuerzo en el desarrollo de la aplicación, sino también pagar un alto costo al momento de migrar la aplicación paralela a otras arquitecturas, requiriendo, en muchos casos, hasta un nuevo desarrollo de la aplicación. Todos estos problemas se resuelven, si los programadores de aplicaciones paralelas consideran una metodología para los desarrollos.

En las siguientes secciones se detallan distintos paradigmas de programación, modelos estándares de la computación paralela; sus ventajas y desventajas.

1.2. Modelos Paralelos Estándares: Paradigmas

Aquellos modelos que describen estructuras o patrones típicos para resolver un problema, se los denomina paradigmas. Un paradigma, muy conocido, es el paradigma para la resolución de problemas *divide y vencerás*. Este se caracteriza por resolver un problema a través de la solución de subproblemas más pequeños y de la misma naturaleza que el problema original.
En paralelismo, existen varios paradigmas. El origen de ellos se debe a la naturaleza de los sistemas paralelos y a los distintos aspectos que se pueden paralelizar en un problema. La idea de dar una solución paralela se basa en la filosofía: *Un problema grande es dividido en subproblemas cada uno de los cuales se resuelve en forma concurrente al resto* (Blelloch, 1996) (Carlini, 1991) (Wilkinson, 1999).
Cuando se construye algoritmos paralelos se debe considerar distintos aspectos:

1. La naturaleza del problema,
2. La relación entre los datos,
3. La posibilidad de dividir los datos o el problema,
4. La independencia o dependencia de los resultados.

Todo esto hace que surjan distintos paradigmas de computación paralela. Una clasificación se deriva de considerar el objeto a paralelizar: los datos o la computación. Dos categorías surgen de esta clasificación el *Paralelismo de Datos* y el *Paralelismo de Control* (Quinn, 1994). En Foster (Foster, 1994), si la técnica de paralelismo considera primero los datos asociados al problema, luego determina una apropiada partición y finalmente trabaja sobre ellos, se la denomina *Paralelismo por Descomposición de Dominios*. En cambio cuando se prioriza la descomposición de la computación sobre los datos lo denomina *Paralelismo por Descomposición Funcional*.
Otros paradigmas paralelos son aquellos orientados a la arquitectura. Los más conocidos son aquellos que consideran la memoria de la máquina paralela como un recurso compartido, *Paradigma de Memoria Compartida* o *Paradigma de Memoria Distribuida* (Wilkinson, 1996).

En las próximas secciones se explican cada uno de los paradigmas mencionados anteriormente y los que surgen de la combinación de alguno de ellos.

1.2.1. Paralelismo de Datos

El paralelismo de datos se caracteriza por la ejecución paralela de la misma operación sobre distintos datos. En otras palabras, múltiples unidades funcionales aplican simultáneamente la misma operación a un subconjunto de elementos. Dicho subconjunto integra la partición del conjunto total de datos.

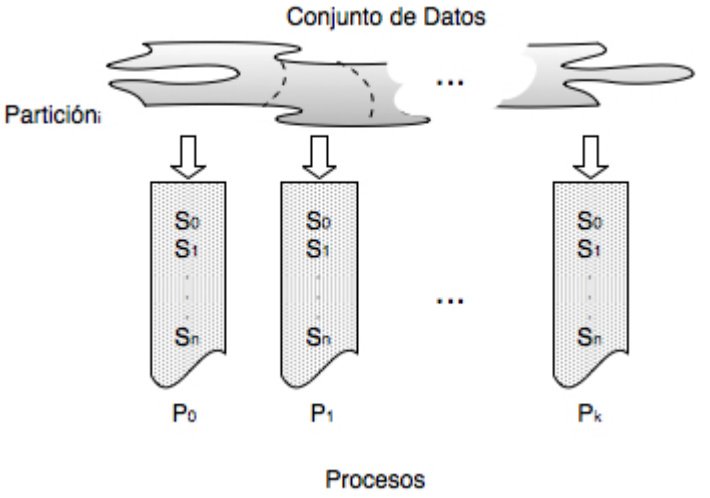


Figura 1.4. Paralelismo de datos

El modelo de programación paralelo de datos está orientado a arquitecturas con un simple espacio de direcciones o a aquellas con memoria distribuida físicamente. En este último caso, la distribución de los datos juega un papel importante. Las arquitecturas SIMD y NUMA son las que mejores se adaptan a este modelo, aunque existen implementaciones de alta performance sobre cluster (Schreiber, 1999). Aplicar paralelismo de datos tiene numerosas ventajas, una de ellas es la simplicidad de programación. La implementación final es un programa con un único thread de control. Generalmente un programa paralelo es una secuencia de pasos, algunos de ellos paralelos y otros secuenciales, la figura 1.4. ilustra la estructura de una solución paralela resuelta aplicando el paradigma de paralelismo de datos.

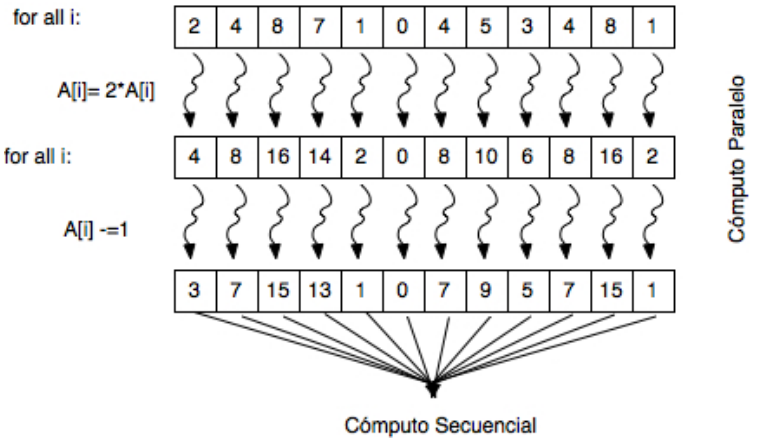


Figura 1.5. Estructura típica de un programa paralelo aplicando paralelismo de datos

En la figura 1.5, el conjunto de datos es dividido en subconjuntos o dominios. Cada subconjunto es asignado a una unidad de procesamiento diferente. La división de los datos, en la mayoría de las herramientas lenguajes, debe ser especificada por el programador, quien generalmente decide cómo realizarla. Como puede observarse en el ejemplo, cada subconjunto tiene cardinalidad 1 y en todas las secciones paralelas, todas las unidades de procesamiento aplican la misma instrucción. La figura 1.6 muestra el mismo problema pero para una división de datos distinta.

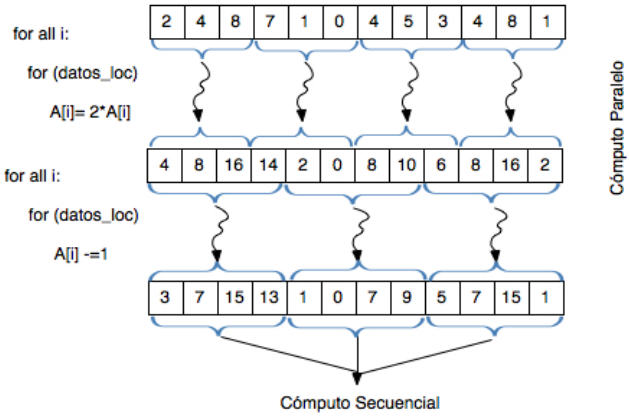


Figura 1.6. Otra solución paralela de la figura 1.5 aplicando paralelismo de datos

Las características de los problemas resueltos con paralelismo de datos son aptas para ejecutarse en computadoras SIMD.

1.2.2. Paralelismo de Tareas

El Paralelismo de Control consiste, en cambio, en aplicar simultáneamente diferentes operaciones a distintos elementos de datos. El flujo de datos entre los distintos procesos que aplican paralelismo de control puede ser muy complejo.
El paralelismo de control, si bien no es simple de programar, su principal característica es la *eficiencia* y *adaptabilidad* para resolver problemas con estructuras de datos irregulares (Hardwick, 1996) (Gonzalez, Supporting Nested Parallelism, 2000) (Gonzalez, Toward Standard Nested Parallelism, 2000).
La figura 1.7 muestra gráficamente esta clasificación, S_i, Q_j y L_t (donde $0<=i<=n$, $0<=j<=m$ y $0<=y<=x$) son las sentencias que ejecutan, respectivamente, cada uno de las unidades de procesamiento P_z con $0<=z<=k$.

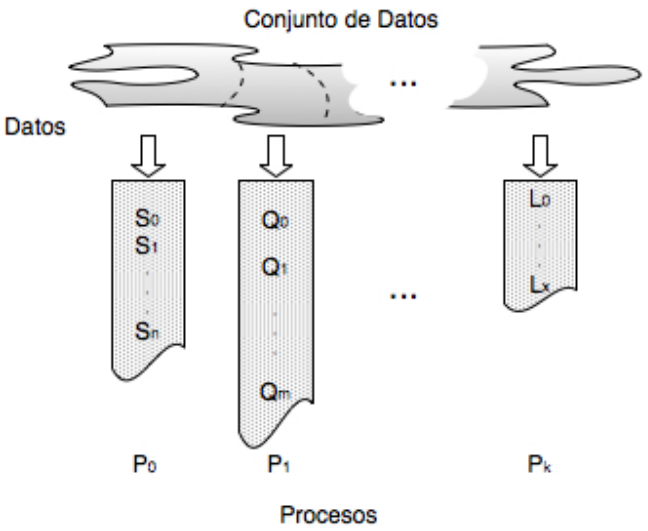


Figura 1.7. Paralelismo de control

Una buena herramienta para entender este paradigma es desarrollar el grafo de dependencias de las tareas involucradas en el sistema paralelo. En él se reflejan las interrelaciones de las distintas tareas y sus dependencias. Por ejemplo si el grafo describe un camino simple y lineal entre las distintas tareas, entonces se dice que la solución es pipelineada. La figura 1.8 (a) muestra el grafo de dependencia de este tipo de algoritmos. En la parte (b) se muestra un grafo de dependencia

arbitrario, se puede observar la existencia de varias dependencias, por ejemplo la tarea T_9 depende de T_6. Las tareas ubicadas en distintos caminos del grafo, pueden ejecutarse en paralelo.

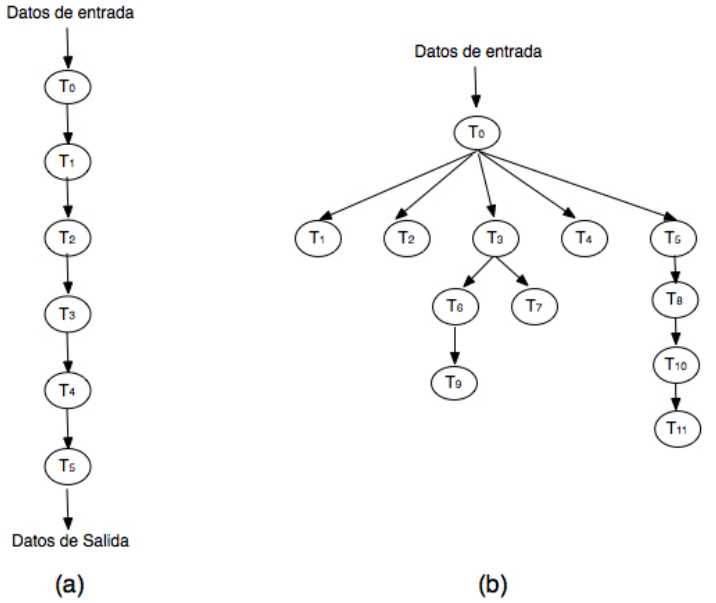


Figura 1.8. Grafo de dependencia de tareas de una aplicación paralela

Las computaciones *pipelineadas* son un caso especial del paralelismo de control donde la computación se divide en etapas; cada etapa trabaja en paralelo sobre una parte del problema tomando como entrada los datos producidos por la etapa anterior y dando como salida la entrada de la etapa siguiente. En este flujo de datos el primer y último estado del pipe son la excepción, la entrada del primero y la salida del último son, respectivamente, los datos de entrada y los resultados de la aplicación.
Al igual que en el paralelismo de datos, los algoritmos que aplican paralelismo de control se adaptan mejor a las arquitecturas MIMD. Sin embargo, esto no siempre ocurre.

1.2.3. Paralelismo Anidado de Datos

Cuando el estilo SIMD, propio del paralelismo de datos, deriva en un estilo SPMD, *Simple Programa-Múltiples Datos*, operaciones complejas sobre los dominios de los datos, tienen lugar. Si los dominios de datos son en si mismos estructurados, y si las operaciones paralelas a aplicarles también lo son, se está en presencia de un nuevo modelo de paralelismo: *Paralelismo Anidado de Datos*.

Como se mencionó anteriormente, el paralelismo de datos es uno de los esfuerzos más exitosos para introducir paralelismo explícito en lenguajes de alto nivel. Este tipo de paralelismo es conveniente por dos razones: es fácil de programar y tiene buena escalabilidad en problemas de gran tamaño. La mayoría de las implementaciones de lenguajes de datos paralelos existentes se centra en la explotación del paralelismo de datos no estructurado, es decir del paralelismo plano o chato, también conocido como *flat*.

Varias generalizaciones para el modelo de datos paralelos fueron propuestas. Ellas permiten el anidamiento de constructores paralelos de datos para especificar computaciones paralelas a lo largo del anidamiento y sobre estructuras de datos irregulares. Estas extensiones incluyen la capacidad de invocaciones paralelas anidadas, es decir combinan la facilidad de programación del modelo de paralelismo de datos y la eficiencia del modelo de paralelismo de control.

Los problemas resueltos a través de la técnica *Divide y Vencerás* son los que mejor se adaptan para ser resueltos aplicando paralelismo anidado de datos. La metodología propuesta por esta técnica consiste en dividir el problema en múltiples subproblemas, los cuales son resueltos independientemente y las soluciones de cada uno son combinadas en una única solución. Cada uno de los subproblemas es resuelto de la misma forma que el problema original, pero en menor escala. De esto se deduce que la técnica procede en forma recursiva hasta que el problema no pueda ser dividido más.

Los algoritmos *divide y vencerás* brindan la oportunidad de explotar el paralelismo no sólo a nivel de tareas sino también a nivel de datos en las fases de división y combinación. El paralelismo de datos se introduce, entonces, haciendo que cada procesador trabaje en una subsección de los datos de entrada en la fase de división y en una subsección de los datos de salida en la fase de combinación. El paralelismo de control está presente en las invocaciones paralelas recursivas resultantes de la división del problema.

Varios enfoques han sido propuestos para integrar paralelismo de datos y paralelismo de tareas en un único lenguaje de programación paralela, en (Bal H. a., 1998) se comparan diferentes lenguajes que integran ambos paradigmas. La combinación de ambos paradigmas puede ser hecha de dos maneras diferentes. Una propone introducir paralelismo de tareas en lenguajes paralelos de datos, como es el caso de *High Performance Fortran* (Koelbel, 1994). Existen varios enfoques que siguen esta idea como son Opus (Chapman B. M., 1994), Fx (Subhlok, 1997), HPF2 (Schreiber, 1999) y $COLT_{HPF}$ (Orlando, 1999). La metodología típica es dividir el flujo de control de un grupo de procesadores en varios subgrupos, todos independientes.

Cada uno de estos subgrupos trabaja como una máquina individual, aplicando paralelismo de datos o continuando la división en grupos. El otro enfoque consiste en extender los lenguajes que proveen paralelismo de tareas como es el caso de OpenMP (Board O. A., OpenMP: A Proposed Industry Standard API for Shared Programming., 1997), (Board O. A., OpenMP Specifications: FORTRAN 2.0., 2000) Java (Campione, 1998) y Orca (Bal H. K., 1992). La extensión se logra mediante directivas, las cuales introducen el paralelismo de datos, por ejemplo especifican la distribución de arreglo. En (Chapman B. M., 1998), (Ben Hanssen, 1998) y (Leair, 2000) se detallan extensiones para OpenMP y Orca.

1.3. Paradigmas orientados a la Arquitectura

Una computadora convencional consiste de un procesador, el cual ejecuta un programa almacenado en su memoria. El procesador puede acceder a cada dirección de memoria.
Extender el modelo de único procesador, significa tener múltiples procesadores y múltiples módulos de memorias conectados de alguna manera. La topología de la conexión entre los procesadores y los módulos de memoria está determinada por cómo se considera a la memoria: *única* o *propia*. En el primer caso todos los módulos integran un único espacio de direcciones; para el segundo caso, la memoria se considera propia de cada procesador. A continuación se explican cada uno de los paradigmas que surgen.

1.3.1. Paradigma de Memoria Compartida

La programación de un multiprocesador implica tener el código ejecutable y los datos necesarios para su ejecución almacenados en la memoria compartida. De esta manera cada unidad de procesamiento accede a la memoria, ya sea para ejecutar el código, obtener los datos para la ejecución o guardar los resultados de ella. El modelo de programación se basa en el modelo de *threads* (Leopold, 2001).
Desde el punto de vista del programador, la memoria compartida ofrece una atracción especial producto de la conveniencia de compartir los datos. Su principal dificultad está relacionada al hardware, los accesos a todas las direcciones de la memoria compartida deben ser rápidos y se deben instrumentar los mecanismos para proveer la consistencia de la misma.
Generalmente, los programas desarrollados siguiendo este modelo se basan en constructores paralelos provistos por algún lenguaje de

programación. Varios lenguajes fueron desarrollados siguiendo este paradigma, entre ellos están fork (Keller, 2001), openMP, nanos (Ayguadé E. M., 1999) (Ayguadé E. G., 1999). Actualmente y ante la adopción de las tarjetas gráficas como procesadores paralelos, surgieron diferentes herramientas, CUDA es una de las más difundidas. Alcanzó su popularidad por su flexibilidad, fácil aprendizaje y posibilidad de programar aplicaciones de propósito general.

1.3.2. Paradigma de Pasaje de Mensaje

La programación de una multicomputadora implica la división del problema en partes o tareas. Cada una de las tareas puede ejecutarse en una computadora o procesador diferente de la multicomputadora. La comunicación entre las distintas tareas se realiza a través del pasaje de mensajes. Generalmente se utiliza una librería que provee las herramientas para ello.
El paradigma de pasaje de mensajes no es tan atractivo como el de memoria compartida. Varias razones determinan esta preferencia:

- El pasaje de mensaje debe ser especificado en el código del programa que resuelve el problema.
- Los datos son propios de cada tarea, no son datos compartidos.

A pesar de ello tiene sus ventajas que lo hacen muy interesante. No necesita tener mecanismos especiales para controlar accesos simultáneos a datos, lo cual puede implicar un incremento en la performance del sistema; y, quizás la determinante, el paradigma de pasaje de mensajes es adecuado para aplicar en la mayoría de las computadoras existentes.
Un programa paralelo con pasaje de mensajes se puede obtener utilizando cualquier tipo de biblioteca que provea las herramientas para la comunicación entre procesos mediante pasaje de mensajes. Existen herramientas estándares, entre las más conocidas se encuentra MPI (Pacheco P., 1997)(Gropp, 1998) y PVM (Geist, 1994).
Es posible una combinación de ambos paradigmas, es decir se puede desarrollar una aplicación donde se aplique el pasaje de mensajes para comunicar los procesos en computadoras distintas y dentro de ellas aplicar el paradigma de memoria compartida (Jacobsen, 2010) (Pennycook, 2011) (Quinn, Parallel Programming in C with MPI and OpenMP, 2004). Una arquitectura paralela adecuada para este tipo de paradigmas son las computadoras con memoria distribuida, la cual puede ser por ejemplo un cluster de procesadores con 4 cores y GPU.

1.4. Resumen

Al referirse a un sistema paralelo, se hace referencia tanto al hardware como al software, un sistema paralelo es la combinación de ambos.
El hardware tiene distintas clasificaciones, Flynn dividió a las computadoras según el flujo de datos y el flujo de sentencias. Si bien esta clasificación es una de las más populares, otra, y no menos popular, es aquella que considera la disposición de la memoria respecto a los procesadores. De ella surgen dos tipos: las computadoras con memoria compartida y las computadoras con memoria distribuida.
El software paralelo tiene, también, su clasificación, ella depende de varios factores: la división del trabajo en las distintas tareas, la asignación de las tareas a los procesadores, la relación de los procesadores y la organización de las comunicaciones. Surgen modelos estándares, también llamados paradigmas: *Paralelismo de Datos* y *Paralelismo de Tareas*.
Tanto el hardware como el software han evolucionado en forma independiente, existen aplicaciones paralelas desarrolladas para un tipo de arquitectura, y también existen computadoras paralelas fabricadas para resolver una aplicación específica. Esta dependencia debilita los beneficios potenciales de aplicar paralelismo.
La adopción de la computación paralela como la solución a los requerimientos de mayor poder de cómputo debe afrontar, por un lado la resistencia de la comunidad de computación serial a cambiar de paradigma de computación, la cual dispone de un modelo de computación universalmente aceptado, el modelo *von Neumann* (Burks A. G., 1946), y de herramientas, técnicas y ambientes de desarrollo robustos que le facilitan la tarea de la programación.
Por el otro lado, la multiplicidad de arquitecturas paralelas, los múltiples factores que influyen en las soluciones paralelas y la falta de una metodología universalmente aceptada, hacen que la programación paralela requiera de mucho esfuerzo, principalmente en las etapas de desarrollo y depuración de las aplicaciones.
En este capítulo se detallaron las características generales de la computación paralela, sus ventajas y posibles falencias, permitiendo establecer el marco teórico para los capítulos siguientes.

1.5 Ejercicios

10. De las computadoras más poderosas del mundo mostradas en el Top 500 (http://www.top500.org/), analice las características de cada uno: Tipo de arquitectura, Cantidad de procesadores, Características de la memoria, entre otras.

11. Busque en el ranking, aquellas super-computadoras en América Latina. Analice sus características.

12. ¿Cuáles son las mayores diferencias entre las arquitecturas de memoria compartida con las de memoria distribuida? Enumere las ventajas y desventajas de cada una.

13. Si Ud. quiere escribir un programa paralelo ¿Qué debe tener en cuenta a la hora de elegir el paradigma de programación?

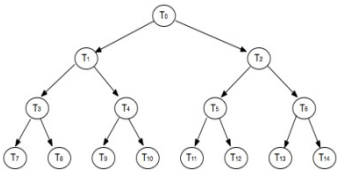
14. Realice el mismo análisis del ejercicio 1.4 pero para los paradigmas de programación de memoria compartida y de memoria distribuida. ¿Cuál elegiría para implementar la solución de un problema y por qué?

15. Dadas las siguiente secuencia de programas analizar las dependencias de los datos y su posible paralelización:

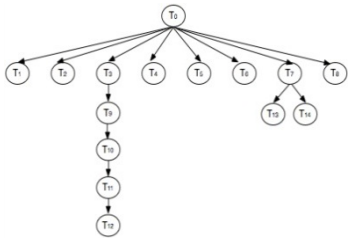
a)
```
DO i=1,N
  a[i]= a[i+1] + x
END DO
```
b)
```
DO i=1,N
  a[i]= a[i] + b[i]
END DO
```
c)
```
x = base
DO i=1,N
a[x ] = a[ x ] - b [ i ]
x = x + stride
END DO
```
d)
```
DO i=1, n
b[i ] = ( a[ i ] - a [i-1] ) * 0.5
END DO
```

16. Dados los siguientes grafos de dependencia de tareas, analice en cada caso: el grado de concurrencia (Máxima cantidad de tareas ejecutando en paralelo), cuáles las tareas pondrían ejecutarse en paralelo y cantidad de procesadores necesarios para la ejecución.

a)



b)



17. Para cada uno de los grafos de dependencia del ejercicio anterior, analice qué ocurre respecto a la utilización de los recursos si se tiene una computadora con 14 procesadores.

18. Analice diferentes soluciones paralelas para resolver los siguientes problemas:
    a. Suma de dos vectores.
    b. Multiplicación de una matriz por un vector.

    Para cada caso, desarrolle el pseudo-código y considere los distintos tipos de paralelismo: de Datos y de Tareas, y los paradigmas de programación: Memoria compartida o distribuida.

19. Proponga el pseudo-código de una solución paralela para resolver:
    a. La multiplicación de matrices.
    b. El ordenamiento quicksort.

Nota: Para los ejercicios 1.9 y 1.10 desarrolle el pseudo-código de la solución secuencial.

CAPÍTULO 2

Introducción a GPGPU

Las unidades de procesamiento gráfico, GPU, se han convertido en una parte integral de los sistemas actuales de computación. El bajo costo y marcado incremento del rendimiento, permitieron su fácil incorporación al mercado. En los últimos años, su evolución implicó un cambio, dejó de ser un procesador gráfico potente para convertirse en un co-procesador apto para el desarrollo de aplicaciones paralelas de propósito general con demanda de anchos de banda de procesamiento y de memoria sustancialmente superiores a los ofrecidos por la CPU.
La rápida adopción de las GPU como computadora paralela de propósito general se vio favorecida por el incremento tanto de sus capacidades como de las facilidades y herramientas de programación. Actualmente la GPU se ha posicionado como una alternativa atractiva a los sistemas tradicionales de computación paralela.
En este capítulo se describe el inicio de las GPU, su filosofía de diseño y las características básicas relacionadas al hardware, haciéndose especial énfasis en aquellas que, de ser tenidas en cuenta, permiten el desarrollo de aplicaciones paralelas con buena performance. Finalmente se realiza un análisis de la GPGPU, computación paralela de propósito general sobre GPU, y su evolución desde las primeras GPU hasta las actuales.

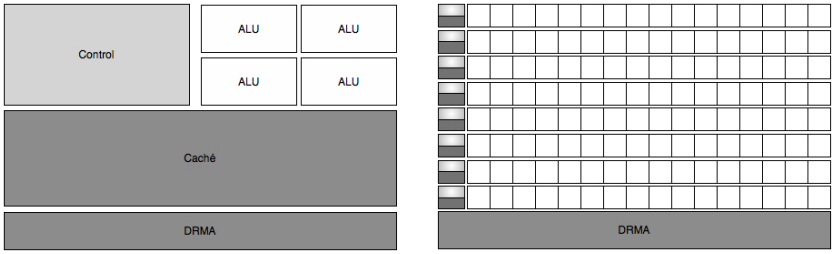
2.1. CPU y GPU

La evolución de los sistemas de computación con multiprocesadores ha seguido dos líneas de desarrollo: las arquitecturas *multicore* (*multi-núcleos*) y las arquitecturas *many-cores* (*muchos-núcleos* o *muchos-cores*). En el primer caso, los avances de la arquitectura se centraron en el desarrollo de mejoras con el objetivo de acelerar las aplicaciones, por ejemplo la incorporación de varios núcleos de

procesamiento. Ante los límites físicos alcanzados por las computadoras personales, la industria tomó la idea a partir de las supercomputadoras existentes e incorporó procesadores a sus desarrollos. Es así que surgieron las computadoras 2, 3, 4, 8 y más procesadores por unidad central (*multicore*). Los *multicores* se iniciaron con sistemas de 2 núcleos y con cada generación se duplica este número, actualmente el procesador Intel *Core* i7 cuenta con versiones de 2 a 6 núcleos y el Intel Xeon E7 con 10 núcleos (hasta 20 threads). En (Hwu, 2008) (Sutter, 2005) se enuncian las características fundamentales de las nuevas arquitecturas, entre las cuales se encuentra el poder de procesamiento. En la actualidad ya no existen computadoras con un único procesador, es más ya está por desaparecer las de sólo 2 *cores*.

En el caso de las arquitecturas *many-cores*, los desarrollos se centran en optimizar el desempeño de las aplicaciones. Dentro de este tipo de arquitectura se encuentran las están las Unidades de Procesamiento Gráfico (*Graphics Processing Unit*, GPU). En su inicio, la primer GPU, GeForce 256, estaba formada por un gran número de pequeños núcleos. Con cada nueva generación, los núcleos se duplicaron, la actual GTX 590 cuenta con 1024 *cores*. Desde sus comienzos, este tipo de arquitecturas *many-core* le llevaron ventaja a los procesadores de propósito general *multicores*, principalmente respecto a las mejoras en la performance, a partir del 2009 la diferencia de las velocidades provistas por ambas arquitecturas es de 10 a 1 (GPU-CPU), 1 TB versus 100GB.

Uno podría preguntarse por qué existe una brecha tan grande entre el rendimiento de una CPU *multicore* y una GPU *many-core*. La respuesta la tienen las diferencias en la filosofía de diseño de ambos. Las optimizaciones de las arquitecturas *multicore* son hechas para proveer mejor desempeño a las soluciones secuenciales, es por ello que las optimizaciones se basan en, por ejemplo, proveer lógica de control compleja para la ejecución paralela de código secuencial o incluir memorias caché más rápidas y más grandes para disminuir la latencia de las instrucciones. Si bien la intención es muy buena, logrando tener computadoras rápidas, con 4 o más núcleos, estas no necesariamente mejoran la performance de las aplicaciones. En la figura 2.1 se muestran las diferencias de filosofías en ambos desarrollos.



CPU GPU

Figura 2.1. Diseño de las Arquitecturas

La filosofía de diseño de las arquitecturas *many-cores* como la GPU y sus avances están regidos por la industria del videojuego y su constate demanda de mejores prestaciones. La idea subyacente es optimizar el *throughput* de muchos *threads* ejecutando en paralelo, de manera tal que si alguno de ello está esperando por la finalización de una operación, se le asigne trabajo y no permanezca ocioso. Las memorias caché son pequeñas, su función es ayudar a mantener el ancho de banda definido para todos los *threads* paralelos. Estas características determinan por ello que la mayor parte de la arquitectura está dedicada a cómputo y no a técnicas para disminuir la latencia.

Otro punto de discrepancia entre ambos tipos de arquitecturas es el ancho de banda de la memoria, las GPU mantuvieron siempre una brecha en el ancho de banda diez veces superior al de las CPU contemporáneas. Esto obedece a que las arquitecturas de propósito general deben optimizar el ancho de banda para atender a todas las aplicaciones, operaciones de entrada/salida y funciones del sistema operativo coexistentes en el sistema. Por el contrario en la GPU con su modelo de memoria más simple y menor número de limitaciones, es más fácil lograr mayor ancho de banda de memoria. Uno de los más recientes chips de Nvidia GTX 590 soporta alrededor de 327 GB/s y para los Intel Xeon E7 en *multicore*, el ancho de banda de la memoria es de 102GB/s, siendo inferior al provisto por la GPU.

Como se desprende de lo expuesto, la filosofía de diseño para las arquitecturas de las CPU y las GPU son distintas, estas últimas son diseñadas como computadoras especializadas en cómputo numérico. Es de esperar que las aplicaciones secuenciales trabajen bien en las CPU, mientras que aquellas con intensivo cómputo numérico lo hagan mejor en la GPU. Poder contar con una arquitectura de trabajo híbrida permite aprovechar las ventajas de la CPU y las GPU. Esto es posible gracias a CUDA (*Compute Unified Device Architecture*) (Kirk, 2010) (NVIDIA, 2011) (Sanders, 2010), el cual fue presentado por Nvidia en 2007 y su principal característica es permitir el desarrollo de aplicaciones en un modelo de arquitectura CPU-GPU.

Además de las prestaciones ofrecidas por las arquitecturas, otros factores influyen en la selección de una arquitectura para una determinada aplicación, dos de ellas son: la aceptación popular y las herramientas adecuadas para desarrollo de software. En el primer caso se refiere a la presencia en el mercado, así como las CPU existentes hoy en el mercado son *multicores* (Es muy raro encontrar computadoras con un único *core*), la mayoría de las sistemas de computación tienen una GPU incorporada. Esto hizo que arquitecturas masivamente paralelas de bajo costo lleguen a la mayoría de los usuarios y en consecuencia se piense en el desarrollo de aplicaciones paralelas para resolver problemas comunes. Es bueno recordar que los altos costos de las computadoras paralelas hicieron que los desarrollos fueran exclusivos de los gobiernos, las grandes empresas o los ámbitos académicos. Con la llegada de las *many-cores* todo cambió, la computación paralela llegó como un producto de mercado masivo, por ejemplo hasta el 2010 se vendieron más 200 millones de unidades de procesadores G80 y sus sucesores. Este número crece día a día, esperándose un incremento mayor con la nueva generación de GPU para dispositivos móviles como celulares, tablets, GPS, entre otros.

Respecto a los desarrollos de software, todas las características de las nuevas arquitecturas *multicore* no tienen sentido si se continúa desarrollando aplicaciones siguiendo la metodología de programación secuencial *von Neumann* (Burks, 1946). Si bien las aplicaciones se ejecutan más rápido en las nuevas tecnologías, éstas no aprovechan todas sus potencialidades. No ocurre lo mismo con aquellas aplicaciones desarrolladas para mejorar su rendimiento tratando de aprovechar las características de la arquitectura subyacente. Estas aplicaciones generalmente son desarrolladas siguiendo otros modelos de computación como es el modelo paralelo.

En el capítulo anterior se especificó que un programa paralelo está formado por múltiples unidades de procesamiento (procesos o *threads*), las cuales cooperan para resolver un único problema. La computación paralela no es un tópico nuevo, durante mucho tiempo los principales desarrollos se dieron en ámbitos académicos y de investigación utilizando computadoras paralelas de alto costo. Actualmente, la tendencia es aplicarla en todos los ámbitos, la popularidad alcanzada por las arquitecturas *multicore* y su bajo costo hacen que la computación paralela ocupe lugares hasta ahora exclusivos de la computación secuencial. Las aplicaciones candidatas a una implementación paralela son aquellas cuya solución secuencial es muy costosa, tanto en recursos computacionales como en tiempo. Cuanto más trabajo hay para hacer, más oportunidades existen de dividirlo entre varias unidades menores, las cuales trabajando

cooperativamente y en paralelo obtienen la solución al problema en menor tiempo.
En el caso de la GPU, hasta el 2006 su utilización como co-procesadores era muy difícil, se las utilizaba a través de las técnicas de la programación gráfica, tal como OpenGL (Shreiner, 2004) o Direct3D (Glidden, 1997). Con el lanzamiento de CUDA por Nvidia a partir de la serie G80 de GPU, todo cambió.
Los programas de CUDA no son necesariamente aplicaciones gráficas, es una herramienta para la programación paralela de aplicaciones de propósito general sobre la GPU. Además posee una curva de aprendizaje muy pequeña, puede verse como la extensión de herramientas de programación bien conocidas como son C (Kernighan, 1978) o C++ (Satir, 1995).
Como se mencionó antes, la industria de las computadoras basó su evolución en dos aspectos, por un lado en las prestaciones que ofrecían al usuario común, lo cual determinó la evolución de la computación personal. Por el otro, se basó en ofrecer mejor performance en los ambientes de Supercomputadoras. En el primer caso, la evolución desde la década del 80, y por 30 años, estuvo marcada por el incremento de la velocidad del reloj del procesador, la cual fue desde 1MHz hasta llegar a velocidades que oscilan entre 1 y 4 GHz. Respecto a los avances en las supercomputadoras si bien obtuvieron ganancias de los avances de la computación personal, la demanda de mayor velocidad fue resuelta mediante la incorporación de más procesadores a su supercomputadora, llegando a tener miles de ellos; y de la inclusión de otros dispositivos más simples y con mayores prestaciones, como es el caso de la GPU. Es interesante destacar que hasta el momento la supercomputadora más rápida del mundo (Supercomputer, 2011), Tianhe-1A, es un cluster construido a través de PCs con GPU conectadas por una red de alta performance. Esto, sin duda, es un indicador de la potencialidad y competitividad de la GPU en la computación de alta performance.

2.2. Evolución Histórica de la GPU

La historia de las GPU se inicia en la década del 60, cuando se pasa de la impresora como medio de visualización de resultados a los monitores. Si bien las primeras tarjetas gráficas tuvieron capacidades muy reducidas, su crecimiento no se detuvo con el correr de los años.
Así como la velocidad del procesador avanzó a través de dos líneas: incrementando la velocidad del reloj y/o incrementando el número de *cores*, la evolución de las unidades de procesamiento gráfico tuvo su

origen a finales de los 80 cuando se inicia la gran demanda de mejores interfaces gráficas por parte de los sistemas operativos, por ejemplos de Microsoft Windows. Esto implicó que a principios de los 90 se comenzara a vender aceleradores gráficos 2D para computadoras personales.

La evolución de las tarjetas gráficas dio un giro importante en 1995 con la aparición de las primeras tarjetas 2D/3D, fabricadas por Matrox, Creative, S3 y ATI, entre otros. Dichas tarjetas cumplían el estándar SVGA, pero incorporaban funciones 3D. En 1997, 3dfx lanzó el chip gráfico Voodoo, con una gran potencia de cálculo, así como nuevos efectos 3D (*Mip Mapping, Z-Buffering, Antialiasing*, entre otros). A partir de allí los desarrollos no se detuvieron, día a día la potencia de cada nuevo producto fue mayor llegando al punto que el puerto *PCI-Express* donde se conectaba la GPU a la CPU(*host*) constituyera un cuello de botella por su bajo ancho de banda.

Al mismo tiempo, en la comunidad profesional y mediante la empresa Silicon Graphics, se introduce al mercado el uso de gráficos 3D en una gran variedad de ámbitos. Además se desarrolla la librería OpenGL para ser usada como método independiente de la plataforma y poder escribir aplicaciones gráficas 3D.

La demanda de aplicaciones gráficas 3D tuvo un gran crecimiento, el cual fue alentado primero, por el desarrollo de juegos en primera persona, FPS (Harrigan, 2004) como Doom, Duke Nukem 3D y Quake, los cuales no sólo le dieron el puntapié para los desarrollos gráficos sino que constantemente demandaron mayores requisitos para lograr mayor realismo en la interfase.

Compañías como Nvidia, ATI Technologies y 3fdx Interactive inician una competencia feroz para lograr buenos aceleradores gráficos. En 1999 Nvidia lanza al mercado la tarjeta gráfica Geforce 256, la cual permitía realizar transformaciones e iluminación a través del hardware, además de brindar mejores condiciones de visualización. Este desarrollo es considerado la piedra basal para los posteriores desarrollos en tarjetas gráficas. La siguiente generación de Nvidia constituyó un gran paso en la tecnología de las GPU, fue considerada la primer GPU con implementación nativa de la primera versión de DirectX8. Por primera vez los desarrolladores tienen el control de la computación a realizarse en la GPU.

Desde 1999 hasta 2002, Nvidia dominó el mercado de las tarjetas gráficas con las GeForce. En ese período, las mejoras se orientaron hacia el campo de los algoritmos 3D y la velocidad de los procesadores gráficos. Sin embargo, las memorias también necesitaban mejorar su velocidad, por lo que se incorporaron las memorias DDR (Jacob, 2007) a las tarjetas gráficas. Las capacidades

de memoria de vídeo en esa época pasan de los 32 MB de las Geforce2 a los 64 y 128 MB de la GeForce 4.
A partir del 2006, Nvidia y ATI (comprada por AMD) constituyen los competidores directos, repartiéndose el liderazgo del mercado con sus series de chips gráficos GeForce y Radeon, respectivamente.
En las siguientes secciones se describen las características básicas de las arquitecturas de las primeras GPU y de las actuales, mostrando su evolución.

2.3. Inicio de la GPU

La GPU desde sus inicios fue un procesador con muchos recursos computacionales (ver figura 2.1). Actualmente ha adquirido notoriedad por su uso en aplicaciones de propósito general, pasó de ser un procesador con funciones especiales a ser considerado un co-procesador masivamente paralelo, capaz de ser la arquitectura base de aplicaciones paralelas. En las próximas secciones se explica su evolución, desde el *pipelinegráfico* hasta las arquitecturas actuales.

2.3.1. Estructura del Pipeline Gráfico

Para poder entender la GPU como unidad de procesamiento de propósito general, GPGPU, es necesario comprender su funcionamiento desde el punto de vista de la especialidad del hardware: los gráficos. Esto permitirá realizar una analogía de cada uno de los mecanismos propios con los que se aplican en GPGPU.
Tradicionalmente, el funcionamiento de las GPU es sintetizado como un *pipeline* de procesamiento formado por etapas con una función especializada cada una. Estas son ejecutadas en paralelo y según un orden preestablecido, cada etapa recibe como entrada la salida de la etapa anterior y su salida es la entrada a la siguiente etapa. En la figura 2.2 se muestra el *pipeline gráfico* básico utilizado por las primeras GPU, a partir de la arquitectura de la serie G80 de Nvidia, ésta fue modificada.

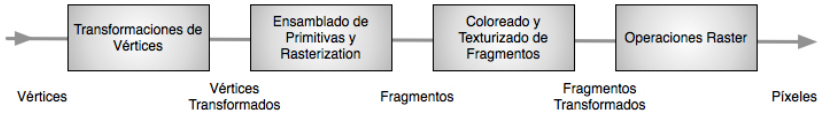


Figura 2.2: *Pipeline gráfico* de una GPU

¿Cómo trabaja el sistema de computación con GPU? La aplicación envía a la GPU una secuencia de vértices, agrupados en lo que se denominan primitivas geométricas: polígonos, líneas y puntos, las cuales son tratadas secuencialmente a través de cuatro etapas bien diferenciadas. En la figura 2.3 se muestra la secuencia de funciones y los datos con los que trabaja cada etapa del *pipeline gráfico*. La tarea/s a realizar por cada etapa es/son:

- Primera etapa: *Transformación de Vértices*
  Es la primera etapa del *pipeline* de procesamiento gráfico. En ella se lleva a cabo una secuencia de operaciones matemáticas sobre cada uno de los vértices suministrados por la aplicación. Entre las operaciones se encuentran: transformación de la posición del vértice en una posición en pantalla, generación de las coordenadas para la aplicación de texturas y asignación de color a cada vértice.
- Segunda etapa: *Ensamblado de Primitivas yRasterization*
  Los vértices generados y transformados en la etapa anterior pasan a esta segunda etapa, donde son agrupados en primitivas geométricas basándose en la información recibida junto con la secuencia inicial de vértices. Como resultado, se obtiene una secuencia de triángulos, líneas o puntos.

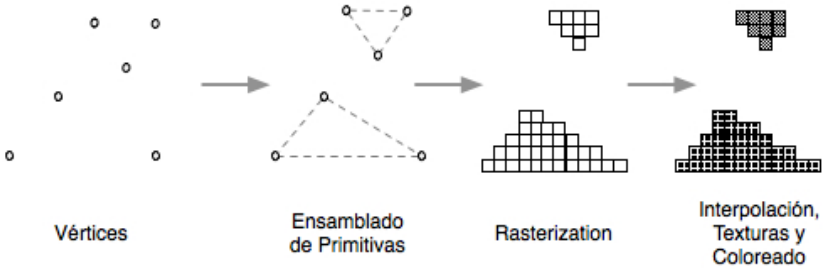


Figura 2.3: Representación de las funciones realizadas por el *pipeline gráfico*

Dichos puntos son sometidos a la *rasterization.* La *rasterization* es el proceso por el cual se determina el conjunto de píxeles "cubiertos" por una primitiva determinada. Los resultados de la *rasterization* son conjuntos de localizaciones de píxeles y conjuntos de fragmentos.
Es importante definir correctamente el concepto de fragmento, por la importancia que cobra cuando se trabaja en computación general. Un fragmento tiene asociada una localización de píxel e información relativa a su color y uno o

más conjuntos de coordenadas de textura. Se puede pensar en un fragmento como en un "píxel en potencia": si el fragmento supera con éxito el resto de etapas del *pipeline*, se actualizará la información de píxel como resultado.

- Tercera etapa: *Interpolación, Texturas y Colores*
  Una vez hecha la *rasterization*, el/los fragmentos obtenidos son sometidos a operaciones de interpolación, operaciones matemáticas y de textura y determinación del color final de cada fragmento. Además de establecer el color final del fragmento, en esta etapa es posible descartar un fragmento determinado para impedir que su valor sea actualizado en memoria; por tanto, esta etapa emite uno o ningún fragmento actualizado para cada fragmento de entrada.
- Cuarta etapa: *OperacionesRaster*
  En esta última etapa del *pipeline* se realizan las operaciones llamadas *raster*, analizan cada fragmento, sometiéndolo a un conjunto de pruebas relacionadas con aspectos gráficos del mismo. Estas pruebas determinan los valores que tomará el píxel generado en memoria a partir del fragmento original. Si cualquiera de estas pruebas fallan, es en esta etapa cuando se descarta el píxel correspondiente, y por tanto no se realiza la escritura en memoria del mismo. En caso contrario, y como último paso, se realiza la escritura en memoria, denominada *framebuffer* (Godse, 2009), como resultado final del proceso.

Un *pipeline gráfico* recibe como entrada la representación de una escena en 3D dando como resultado una imagen 2D a ser proyectada en una pantalla. Como se ve, esto se logra luego de muchas transformaciones a lo largo del *pipeline*. Éste puede implementarse vía software o hardware. En dispositivos dedicados todas las etapas son resueltas por el hardware lo que lo hace muy rápido. En sistemas más convencionales como PC, todo se realiza vía software y por tanto es más lento.

2.3.2. Pipeline en CPU y en GPU

La filosofía de diseño de las GPU y las CPU siguió distintos objetivos y en consecuencia distintos caminos. Desde los inicios de la GPU estuvo presente el paradigma paralelo. En el *pipeline gráfico* la entrada de un estado es la salida del anterior, en este caso el paralelismo es a nivel de tareas, varios datos pueden estar al mismo tiempo en distintos estados del *pipeline*.
Un *pipeline* en la CPU implica la utilización de todos los recursos para un estado, un cambio de estado implica retirar los recursos de la etapa corriente y asignarlos a la siguiente. En la GPU esto es diferentes, los

recursos son divididos entre las distintas etapas del *pipeline*, por lo que varias unidades de cómputo trabajan en paralelo en los diferentes estados del *pipeline*. En otras palabras los recursos en la GPU se dividen en el espacio, los de la CPU en el tiempo. Esto implica un nivel de paralelismo interno en la etapa y otro entre las etapas del *pipeline*. En una GPU si bien una operación sobre gráficos puede implicar miles de ciclos de reloj, pero con el paralelismo de tareas y de datos de las etapas del *pipeline* y entre ellas, es posible obtener un alto *throughput*.

2.3.3. GPU: Arquitectura Fijas vs. Unificadas

Como el *pipeline gráfico* de funciones fijas carecía de generalidad para expresar operaciones más complejas de sombreado o iluminación, se propuso reemplazar las funciones fijas sobre vértices y fragmentos por programas especificados por el usuario. Si bien este reemplazo resolvió el problema, con el correr de los años, los programas de vértices y de fragmentos fueron más capaces, y aunque tenían un completo conjunto de instrucciones, poseían límites respecto a los tamaños de los problemas.
Después de muchos años de trabajar con un conjunto separado de operaciones sobre vértices y fragmentos, las GPU incorporan el Modelo *Shader* unificado para tanto el sombreado de vértices y como el de fragmentos (Blythe, 2006). Éste propone:

- El hardware debe soportar sombreado de los programas de al menos 65KB de instrucciones estáticas e instrucciones dinámicas ilimitadas.
- El conjunto de instrucciones, por primera vez, soporta tantas operaciones sobre números enteros de 32 bits y de punto flotante de 32 bits.
- El hardware debe permitir un número arbitrario de lecturas directas o indirectas de la memoria global (textura).
- Debe soportar el control de flujo dinámico para iteraciones y *branches*.

Este modelo *Shader* se convirtió en un referente, el cual permitió el desarrollo de aplicaciones sobre GPU, incrementando la complejidad de las operaciones de vértice y fragmentos. La evolución de la arquitectura de las GPU se centró principalmente en las etapas programable del *pipeline gráfico*. Es más, las anteriores generaciones de GPU se pueden describir sus capacidades de programabilidad para el *pipeline gráfico*, las de hoy se caracterizan por contar con un motor programable y unidades de cómputo responsables de realizar funciones específicas.

La arquitectura de la serie GeForce 6 de Nvidia puede definirse como una arquitectura dividida a nivel de *shaders* o procesadores programables: existe en ella hardware especializado para ejecutar programas que operan sobre vértices, y otro dedicado exclusivamente a su ejecución sobre fragmentos. La serie GeForce 7 de Nvidia se basó en una arquitectura similar a su antecesora, la diferencia radicó en el incremento de la potencia en base al aumento de la superficie del chip. En la figura 2.4 se muestra la estructura del *pipeline gráfico* con procesadores *shaders* para los vértices y los fragmentos (diferenciados con otro color).

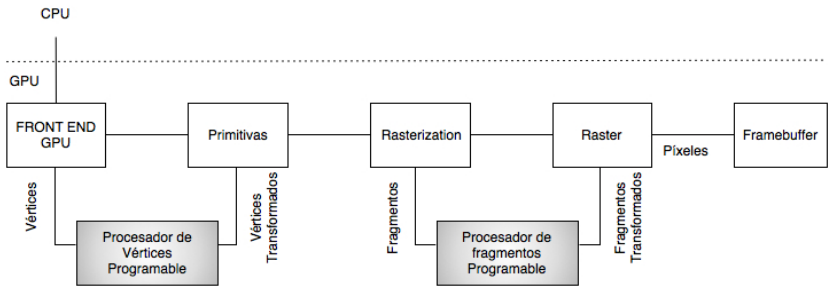


Figura 2.4. *Pipeline gráfico* programable

Pese a que el hardware dedicado puede adaptarse en mayor medida a su función, tiene ciertos inconvenientes, razón por la cual se optó por arquitecturas totalmente diferentes a la hora de desarrollar una nueva generación de procesadores gráficos, basados en una arquitectura unificada. El problema principal de las arquitecturas anteriores surgía en aquellas aplicaciones gráficas cuya carga de trabajo a nivel de geometría (vértices) y de procesamiento de píxeles (fragmentos) no estaba equilibrada. Así, aplicaciones con gran carga de trabajo geométrico implicaban una total ocupación de los procesadores de vértices (Generalmente presentes con menor número en la GPU), y un desaprovechamiento de capacidad de cálculo a nivel de fragmento, las unidades responsables de procesar fragmentos permanecían ociosas. El mismo problema ocurría en las aplicaciones con mayor carga a nivel de fragmentos y menor nivel de procesamiento geométrico.
A partir de la G80 de Nvidia o de la serie R600 de ATI se adoptó la solución de crear arquitecturas unificadas a nivel de *shaders*. En este tipo de arquitecturas, no existe ya división a nivel de hardware entre procesadores de vértices y procesadores de fragmentos. Cada unidad de procesamiento que la integra es capaz de trabajar tanto a nivel de vértice como a nivel de fragmento, sin estar especializado en el

procesamiento de ninguno de los dos en concreto. Dichas unidades reciben el nombre de procesadores de *Stream* (*StreamProcessors*).
En la arquitectura unificada, el número de etapas del *pipeline* se reduce de forma significativa, pasa de un modelo secuencial a un modelo cíclico. El *pipeline* clásico utiliza tipos de *shaders* distintos a través de los cuales los datos fluyen de forma secuencial. En la arquitectura unificada, con una única unidad de *shaders* no especializada, los datos que llegan a la misma (en forma de vértices) en primera instancia son procesados por la unidad, enviando los resultados de salida a la entrada para ser procesados nuevamente, en cada ciclo se realiza una operación distinta. De esta manera se imita el comportamiento del *pipeline* clásico visto en secciones anteriores. La acción se repite hasta que los datos han pasado por todas las etapas del *pipeline*. Finalmente son enviados hacia la salida de la unidad de procesamiento.
El cambio de arquitectura implicó también un cambio en el *pipeline gráfico*: con la arquitectura unificada, ya no existen partes específicas del chip asociadas a una etapa concreta del *pipeline*, sino que una única unidad central de alto rendimiento será la encargada de realizar todas las operaciones, sea cual sea su naturaleza.

2.4. Arquitecturas Unificadas de GPU

La principal ventaja de las arquitecturas unificadas es la posibilidad que brinda del auto equilibrio de la carga computacional. El conjunto de procesadores pueden ahora asignarse a una tarea u otra dependiendo de la carga exigida por la aplicación. De esta manera, a cambio de una mayor complejidad en cada uno de los procesadores de la GPU y de una mayor generalidad en su capacidad de procesamiento, se consigue reducir el problema del equilibrado de carga y de la asignación de unidades de procesamiento a cada etapa del *pipeline gráfico*.
En las siguientes secciones se presentan distintas arquitecturas de GPU, desde la G80 hasta las actuales GF100, todas ellas aptas para la programación de aplicaciones de propósito general.

2.4.1. Arquitectura G80

Como se dijo antes, las arquitecturas anteriores a la G80, la serie GeForce6 y GeForce7, se pueden definir como arquitecturas divididas a nivel de *shaders* o procesadores programables; existe en ellas hardware especializado para ejecutar programas que operan sobre

vértices o están dedicados exclusivamente a su ejecución sobre píxeles. La diferencia entre ambas series de arquitecturas es la potencia, mayor en la serie GeForce7. El principal inconveniente de ambas fue su arquitectura no unificada, por un lado estaban los procesadores de vértices y por el otro los dedicados al cálculo a nivel de píxeles, si la aplicación no tenía un trabajo equilibrado para todos estos tipos de procesadores, algunos eran subutilizados.

La serie G80, en cambio, es una arquitectura unificada a nivel de *shaders*. Ya no existe una división a nivel de hardware entre procesadores de vértices y procesadores de fragmentos. Las unidades de procesamiento, llamadas *StreamProcessors*, son capaces de trabajar tanto a nivel de vértice como a nivel de fragmento, sin especializarse en ninguno de los dos. Esto implicó un cambio en el *pipeline gráfico*: la arquitectura unificada. En ella no existen componentes específicas asociadas a una etapa concreta del *pipeline*, sólo una única unidad central de alto rendimiento será la responsable de realizar todas las operaciones, cualquiera sea su naturaleza. Entre las ventajas de esta arquitectura es la posibilidad de equilibrar la carga entre los distintos procesadores, los cuales se pueden asignar a una tarea u otra dependiendo del trabajo a realizar. Como desventaja se puede considerar la mayor complejidad de los procesadores y su no especificidad en un tipo de problemas.

La Nvidia GeForce 8800 fue presentada en el 2006, dando origen al nuevo modelo de GPU. La G80, base de GeForce 8800, introdujo muchas innovaciones, las cuales fueron claves para la programación de GPU. Entre las nuevas características de la G80 se encuentran:

- Reemplazó los *pipelines* separados de píxeles y vértices por un único y unificado procesador, en el cual se puede ejecutar programas de gráficos (geometría, vértice y *pixel*) y programas de computación en general.
- Reemplazó la administración manual de registros de vector por parte de los programadores por un procesador escalar de *threads*.
- Presentó el modelo de ejecución Simple Instrucción-Múltiples *Threads* (SIMT) para que múltiples *threads* independientes ejecuten concurrentemente una simple instrucción.
- Introdujo la memoria compartida y la sincronización por barreras para la comunicación entre *threads*.
- Brindó la posibilidad de contar con soporte para lenguaje C, permitiendo a los programadores utilizar el poder de la GPU, sin tener que aprender un nuevo lenguaje de programación.

La arquitectura G80 es una arquitectura totalmente unificada, no existe diferencia a nivel de hardware entre las distintas etapas del

pipeline gráfico, está totalmente orientada a la ejecución de *threads* considerando el balance de carga. Opera de forma integral con una precisión de 32 bits para datos de punto flotante, ajustándose al estándar IEEE 754. La figura 2.5 muestra el esquema completo de la arquitectura G80.

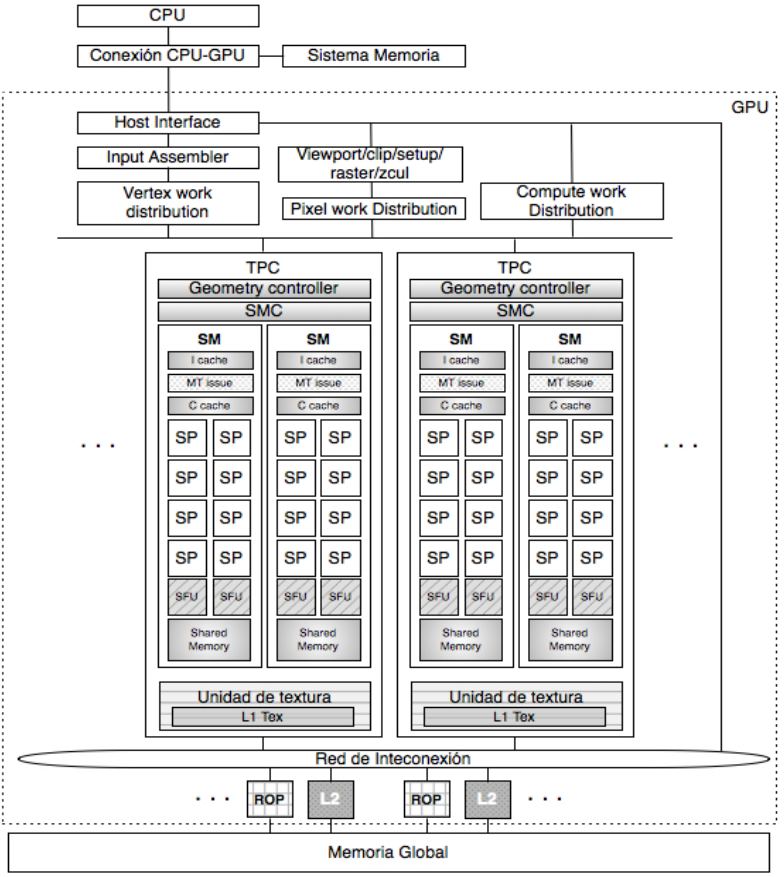


Figura 2.5. Arquitectura de G80

En el diagrama se muestra la arquitectura de una GPU Geforce 8800, la cual cuenta con 128 procesadores de *streaming* o escalares (SP), organizados en 16 multiprocesadores de *streaming* (SM) en ocho unidades de procesamiento independiente denominadas *cluster procesador/texture* (*Texture/Porcessor Cluster*, TPC). El trabajo comienza desde la CPU, quien le indica el trabajo a la GPU a través del bus *PCI-Express*.

Como se puede observar, la GPU tiene varios componentes dispuestos en una jerarquía. En el más alto nivel se observa el arreglo de procesadores de *streaming* (SPA), formados por varios TPC, quienes a su vez están formados como se indica en el párrafo anterior. El sistema de memoria está compuesto por un control de DRAM externa o memoria global y procesadores de operaciones *raster* de función fija (ROP), los cuales se encargan de realizar operaciones específicas como color y profundidad directamente sobre la memoria. A través de una red interconexión se accede desde el SPA a los ROP, se resuelven las lecturas de la memorias de textura desde el SPA a la memoria global, y las lecturas sobre la memoria global a la caché de nivel 2 y al SPA.

El trabajo dentro de la GPU se desarrolla de la siguiente manera, en primer lugar los datos recibidos desde el *host* son preprocesados en hardware específico a fin de organizarlos para aprovechar la máxima capacidad de cálculo del sistema y evitar la existencia de unidades ociosas. Una vez hecha esta organización, se transfiere el control de la ejecución a un controlador global de *threads*, quien decide qué *thread* se ejecutará en cada instante y dónde. Además existe un planificador de *threads* para determinar su ejecución dentro de las unidades de procesamiento.

El *Input Assembler* recolecta el trabajo de *vertex* que ingresa y lo pasa al *distribuidor de trabajo de vertex* (*Vertex work distribution*) quien distribuye los paquetes de trabajo a los TCP en el SPA. Los TCP ejecutan programas *shader* de *vertex* y programas *shaderde geometría*. Los datos de salida son escritos en buffer *on-chip*, estos buffer luego pasan sus resultados a la unidad responsable de la *rasterization (viewport/ clip/ setup/ raster/ zcull block*). La *unidad de distribución de trabajo* de *pixel* distribuye fragmentos de pixeles a los TPC apropiados para el procesamiento. Los fragmentos de pixeles sombreados son enviados a través de la red de interconexión para el procesamiento del color y la profundidad a las ROP. La *unidad de distribución de trabajo de cómputo* (*Compute work distribution*) envía los *threads* de cómputo general a los distintos TPC del SPA para su ejecución. Todas las unidades que integran la arquitectura de la G80, incluido los múltiples relojes que las rigen, proveen independencia y optimización de la performance.

Los varios TPC están organizados como un arreglo de Multiprocesadores *Stream*ing (SM). Por cada TPC hay dos SM. Todos los TPC acceden a la memoria global. Existe un planificador de *threads* asociado a cada *cluster*, así como memoria cache L1 y unidades de acceso y filtrado de texturas propias (*Unidades de Textura*). Cada grupo de 16 SP dentro de un mismo cluster comparten tanto unidades de acceso a texturas como la

memoria caché L1. La figura 2.6 muestra un esquema de la estructura de cada uno de los clusters que forman la arquitectura G80 de Nvidia.

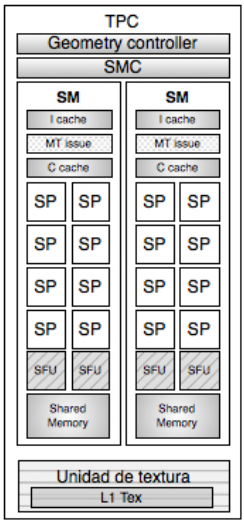


Figura 2.6. Arquitectura de un TPC

Cada SM está formado por ocho *Stream*ing or Scalar Processors (SP), los cuales comparten la lógica de control y la caché de instrucciones. Además de los SP, un SM posee 2 SFUs (*Special Function Units*), una pequeña caché de instrucciones, una unidad MT (*MT issues*), responsables de enviar instrucciones a todos los SP y SFUs en el grupo, una caché de sólo lectura de datos y una memoria *shared* o compartida, generalmente de 16KB. Las SFU llevan a cabo las funciones de punto flotante tal como la raíz cuadrada o funciones transcendentales como seno, coseno, entre otras. Cada SFU ejecuta una instrucción por *thread* por ciclo de reloj. Si las SFU están ocupadas, la unidad de planificación ejecuta otras sentencias para evitar el tiempo ocioso. La figura 2.7 muestra la estructura de un SM.
Respecto a los SP, ellos son responsables de las operaciones matemáticas o de direccionamiento de datos en memoria y posterior transferencia de los mismos, trabajan sobre datos escalares. Cada uno cuenta con una unidad MAD (*Multiply-Add*) y una unidad de multiplicación adicional. En la figura 2.8 se muestra la estructura típica de un SP.

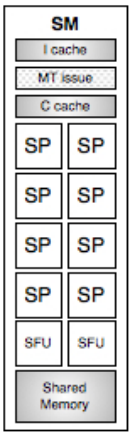


Figura 2.7. Estructura de un SM

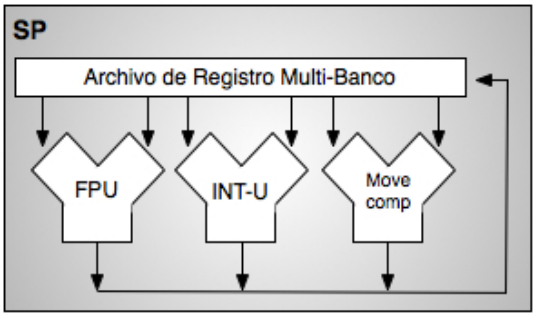


Figura 2.8. Arquitectura de SP

Por las características de la arquitectura G80: 8 grupos de 16 SP, es posible realizar una abstracción la arquitectura y representarla como una configuración MIMD de 8 nodos de computación, cada uno de los cuales formado por 16 unidades o clusters de procesamiento SIMD (SP). Cada uno de los TPC puede procesar *threads* en forma independiente y en un mismo ciclo de reloj.
La comunicación entre la CPU y la GPU es un aspecto importante en el desarrollo de toda aplicación debido al costo que implica. Actualmente la comunicación entre ambos sistemas se realiza a través de un bus *PCI-Express*, constituyendo en muchos casos un cuello de botella de las prestaciones de la aplicación.

2.4.2. Arquitectura GT200

En junio de 2008, Nvidia presenta una revisión de la arquitectura G80, la GT200 tanto en las GeForce, la Quadro y la Tesla (Lindholm, 2008). Se aumentó el número de SP. La capacidad de registros del procesador se duplicó, permitiendo un mayor número de *threads* en ejecución en el chip en un momento determinado. Se adicionó hardware para mejorar el acceso eficiente a la memoria (*memory access coalescing*, concepto explicado en el capítulo 5). Se incluyó soporte para operaciones de punto flotante de doble precisión a fin de satisfacer las demandas de los científicos y de las aplicaciones de computación de alto desempeño (*high performance computing*).
El chip G80 soporta hasta 768 *threads* por SM, la serie GT200 soportan 1024 *threads* por SM, esto significa por ejemplo en la GeForce GTX 260 pueden 24.576 *threads* ser atendidos en paralelo (24 SM) y en la GeForce GTX 295 tiene 60 SM, lo cual implica 61.440 *threads* simultáneos (30.720 por placa). Esto conlleva a que el nivel de paralelismo desde el hardware de la GPU se incrementó rápidamente, lo importante entonces es esforzarse por lograr dichos niveles de paralelismo en el desarrollo de aplicaciones.
En la figura 2.9 puede verse la estructura del TPC para esta serie de arquitecturas.

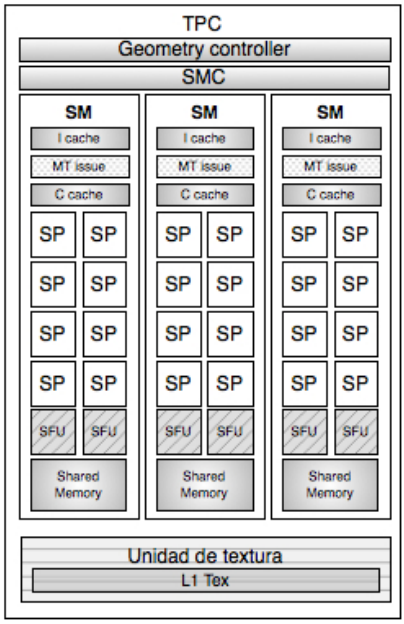


Figura 2.9. Estructura de TPC para la GT200

Como puede apreciarse, el número de SM por TCP en esta generación se incrementó en un 50%. El incremento del número de SP no es la única mejora respecto a las G80, en la tabla de la figura 2.10 se resumen las características fundamentales de las arquitecturas G80 y GT200.

Características \ GPU	G80	GT200
SP por TPC	16	24
Unidades de Direcciones de Textura por TPC	4	8
Unidades de Filtrado de Textura por TPC	8	8
Total de SP	128	240
Total de Unidades de Direcciones de Textura	32	80
Total de Unidades de Filtrado de Textura	64	80

Figura 2.10. G80 vs. GT200

Respecto a la cantidad de memoria *shared* por SM se mantuvo en 16KB, no así la cantidad de memoria de registros, la cual se duplicó, y la cantidad de *warps* residentes en cada SM, fue incrementado en un 33%.
Resumiendo, la diferencia entre una GPU serie G80 y una GT200 se basa en la ampliación de los recursos, área del chip, no en cambios de estructura como ocurre con la siguiente arquitectura.

2.4.3. Arquitectura GF100

En el diseño de cada nueva generación de GPU, la filosofía aplicada por Nvidia es mejorar tanto el rendimiento de las aplicaciones existentes y como la programación de las GPU. Aunque obtener mejor performance para aplicaciones existentes tiene sus ventajas, lo que ha transformado a las GPU en un procesador paralelo de gran aceptación son las facilidades de su programación. La GF100 es la sucesora de la arquitectura GT200 (NVIDIA, 2010).
La primera GPU basada en la arquitectura Fermi cuenta con hasta 512 *cores* (SP). Cada *core* resuelve una instrucción de entero o punto flotante por ciclo de reloj. Los 512 SP están organizados en 16 SM de 32 SP cada uno. La GPU cuenta con seis particiones de memoria de

64 bits, para una interfaz de memoria de 384 bits, la cual soporta hasta un total de memoria DRAM de 6 GB de memoria GDDR5. Los SM comparten una caché de nivel 2 (L2). Además cada SM tiene un planificador, un *despachador* (*dispatch*), registros y memoria caché L1.

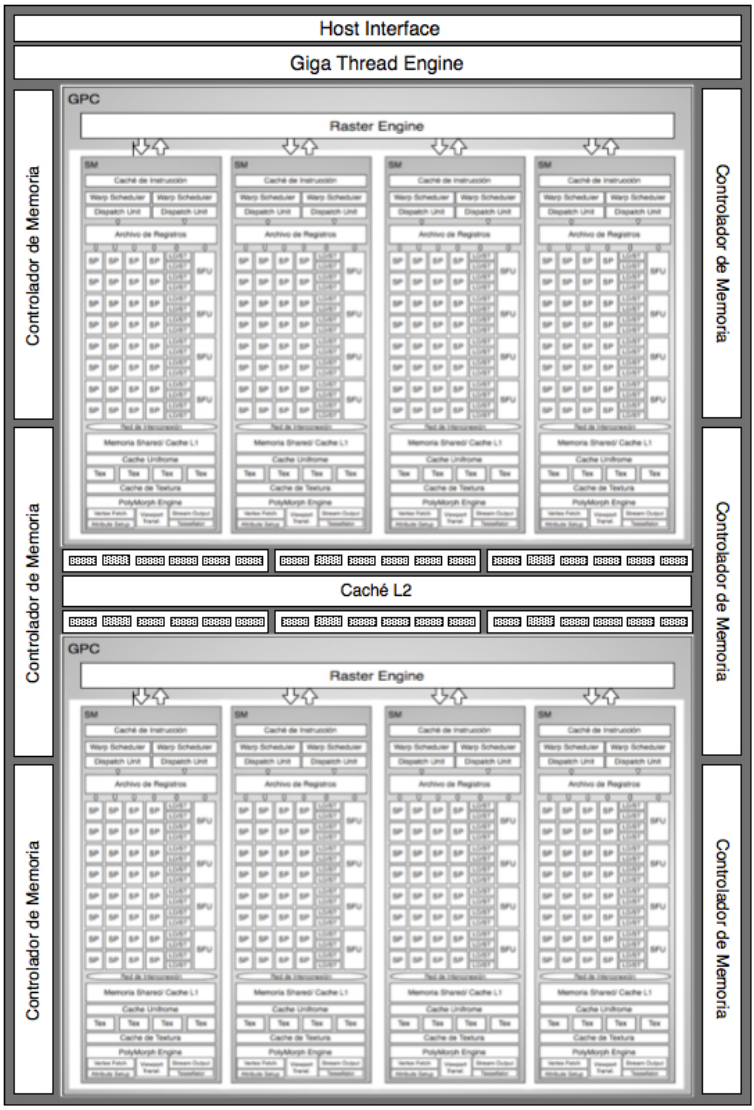


Figura 2.11. Esquema de la arquitectura GF100

La GF100 se basa en un arreglo de *Cluster de Procesamiento Gráfico* (*Graphics Processing Clusters*, GPC), *multiprocesadores de*

streaming (SM) y los controladores de memoria. Una GF100 tiene 4 GPC, 16 SM y 6 controladores de memoria. Estas cantidades dependen del modelo de la arquitectura. Los controladores de memoria atienden a las unidades ROP (*RasterOperation Processor*) para operaciones específicas como *blending* de píxeles y operaciones atómicas sobre la memoria global. Los 48 controladores son divididos en 6 grupos de ocho. La memoria caché L2, los controladores de memoria y los grupos de ROP están íntimamente relacionados, la ampliación o modificación de uno influye a los demás. En la figura 2.11 se muestra un diagrama de la arquitectura GF100.
Un GPC contiene una *RasterEngine* (RE) y hasta cuatro SM (Ver Figura 2.12). Es el más alto nivel de bloques de hardware. Sus dos principales innovaciones son la incorporación de la unidad de :

- *RasterEngine*, escalable para la inicialización de triángulos (*trianglesetup*), *rasterization* y *Z-cull* (recorte de la tercer coordenada, z, de los pixeles o triángulos, pasa de 3D a 2D);
- *PolyMorph Engine* (PME) para el búsqueda de los atributos de *vertex* (*vertex attribute fetch*) y la *tessellation* (división poligonal).

El RE se encuentra en el GPC y el PME en el SM.

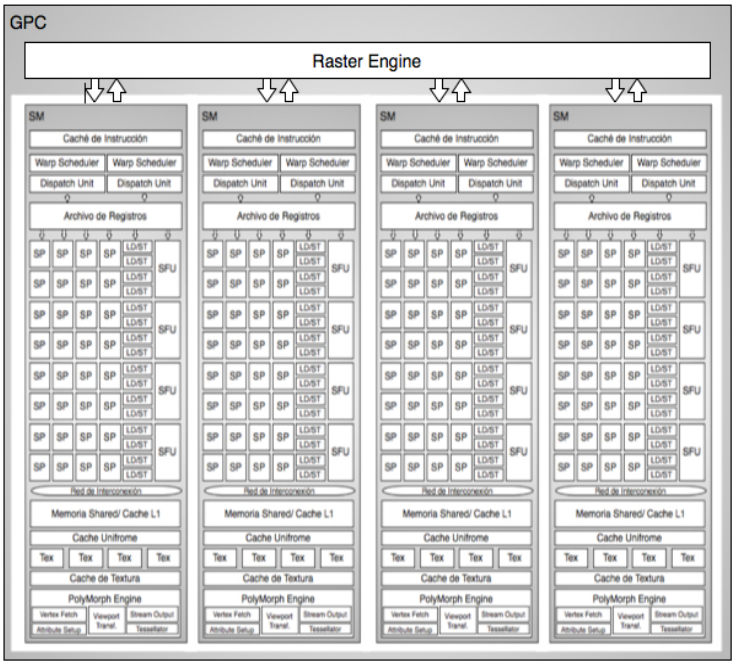


Figura 2.12. Esquema de la arquitectura de un GPC

El GPC encapsula todas las unidades claves de procesamiento de gráficos, a excepción de las funciones ROP, un GPC puede ser considerado como una GPU autónomo, una GF100 cuenta con cuatro GPC.
En las anteriores GPU, los SM y las unidades de textura se agrupan en bloques de hardware, los TPC. En la GF100, cada SM tiene cuatro unidades de textura, eliminando la necesidad de tener TCP.
Los SM presentan varias innovaciones, las cuales lo hacen no sólo más potente sino también más eficiente y programable. En la figura 2.13 se muestra un diagrama de la arquitectura de un SM.

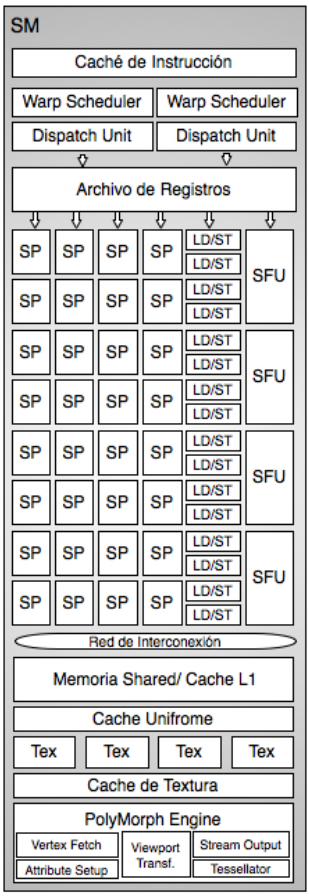


Figura 2.13. Arquitectura de un SM de GF100

Cada SM tiene 32 SP, es decir cuatro veces más que sus antecesores. Además cuenta con 16 unidades *load/store*, las cuales permiten calcular direcciones de origen o destino a 16 *threads* por reloj. Estas

unidades leen o escriben los datos de cada dirección en la caché o la DRAM. Las SFU tienen las mismas características que en las de las anteriores generaciones de GPU.
Por su parte un SP tiene una unidad aritmética/lógica de enteros (ALU) y una unidad de punto flotante (FPU) con una aritmética de punto flotante según el estándar IEEE 754-2008, proporcionando instrucciones fusionadas de *multiplicar-sumar* (FMA), tanto para aritmética de precisión simple como doble. Este tipo de instrucciones mejora las instrucciones de multiplicar-sumar (MAD) haciendo la multiplicación y la suma con un simple paso de redondeo final, sin ninguna pérdida de precisión en la suma, la FMA es más precisa que la realización de las operaciones por separado.
En esta arquitectura se planteó un nuevo diseño de ALU para números enteros. Ésta soporta una precisión de 32 bits para todas las instrucciones. La ALU es también optimizada para soportar operaciones de 64-bit. Varias instrucciones específicas son soportadas, incluyendo operaciones: *booleanas, shift, move, compare, convert, extracción del bit-field, inserción de bit-reverse* y *el recuento de la población*.
Otra de las características importantes de las GF100, es la aritmética de doble precisión. Esta arquitectura fue específicamente diseñada para ofrecer una performance sin precedentes en doble precisión, hasta 16 operaciones multiplicar-sumar pueden ser realizadas por SM por ciclo de reloj. Esta mejora es muy importante respecto a la arquitectura GT200. En la figura 2.14 se muestra la estructura de cada SP.

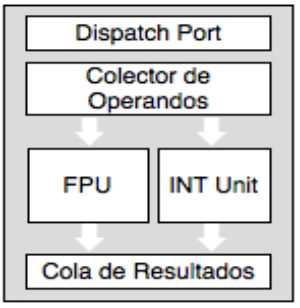


Figura 2.14. Arquitectura SP de GF100

Respecto a la administración de los *threads*, estos son administrados a los SM en grupos de 32, a lo que se llama *warp*. Cada SM tiene dos administradores de *warp* y dos unidades de *dispatch* de instrucciones, permitiendo a dos *warps* ejecutar concurrentemente. El *Planificador*

dual de *warp* selecciona a dos *warp* y da una instrucción de cada uno a un grupo de 16 SP, 16 unidades load/store o a 4 SFU. En el capítulo siguiente se explica con más detalle el concepto de *warp*.
La memoria compartida es una facilidad que permite optimizar la performance de las aplicaciones. En las arquitecturas anteriores, por cada SM se tiene 16KB de memoria compartida, en la GF100 se la extendió a 64KB, la cual se puede configurar hasta 48KB de memoria compartida y 16 KB de caché L1 o a la inversa, 16 KB de memoria compartida y 48 KB de caché L1. Esto triplica la capacidad de la memoria compartida para las aplicaciones que la utilizan, pero para aquellas que no la utilizan se benefician automáticamente de la caché L1.
En la figura 2.11 se muestran dos componentes no descriptas aún, la interfaz con el *host* y la unidad *GigaThread*. Los comandos de la CPU son leídos por la GPU a través de la interfaz de *host* y pasados a la unidad *GigaThread* quien obtiene los datos especificados desde la memoria global y los copia al *framebuffer*. La *GigaThread* crea y envía bloques de *threads* a varios SM. Cada SM, en turno, administra los *warps* (grupos de 32 *threads*) a los SP y otras unidades de ejecución. También es su responsabilidad la redistribución del trabajo de los SM cuando se produce una sobrecarga en el *pipelinegráfico*, por ejemplo después de las etapas de *tessellation* y *rasterization*.
Al igual que las anteriores versiones esta GPU se conecta al *host* a través de *PCI-Express*.

2.5. GPGPU: Computación de Propósito General en GPU

Las GPU comenzaron siendo *pipelines* de procesamiento de gráfico con funciones fijas. Con el paso de los años, estos chips se fueron haciendo más programables, lo que permitió a Nvidia introducir la primera GPU o unidad de procesamiento gráfico del mercado. Entre los años 1999 y 2000, científicos e investigadores de disciplinas como el diagnóstico por imagen o electromagnetismo, empezaron a usar las GPU para aplicaciones de cálculo de propósito general y descubriendo que su gran rendimiento en operaciones de punto flotante implicaba un extraordinario aumento de la velocidad de ejecución en la gran variedad de aplicaciones científicas. Este fue el nacimiento de un nuevo concepto denominado GPGPU o GPU de propósito general.
Dadas las características antes enunciadas de la GPU, ésta constituye una alternativa válida a las arquitecturas masivamente paralelas. Cada día más desarrolladores de software paralelo las están teniendo en

cuenta como arquitectura paralela para computación de propósito general, es decir utilizar el hardware formulado para aplicaciones gráficas en cualquier otro tipo de aplicaciones. Aunque las operaciones son las mismas, la terminología de la computación gráfica es diferente a la de la computación de propósito general.
En (Harris, 2005) se brinda una excelente descripción del proceso de instanciación entre ambos tipo de computaciones. Se realiza un análisis de ambas programaciones desde la descripción de la programación sobre GPU utilizando terminología de gráficos, luego se muestran cómo los mismos pasos se pueden utilizar en la programación de propósito general, y, finalmente, cómo se lleva a cabo en forma sencilla y directa sobre las GPU actuales. Los tres casos a describir son:

1. Programación de una GPU para Aplicaciones Gráficas:
   En este caso se hace referencia a los aspectos programables del *pipeline gráfico* de la GPU. Para el desarrollo de una aplicación gráfica se deben realizar los siguientes pasos:
   1) El programador especifica la geometría que cubre una región en la pantalla. La *rasterization* genera un fragmento en cada píxel cubierto por la geometría.
   2) Cada fragmento es sombreado por el programa de fragmentos.
   3) El programa fragmento calcula el valor del fragmento mediante una combinación de operaciones matemáticas y lecturas en la memoria global desde una memoria de textura.
   4) La imagen resultante de las acciones anteriores se puede utilizar como textura para futuros pasos a través del *pipeline gráfico*.
2. Programación de propósito general sobre las primeras GPU :
   La computación de propósito general se llevaba a cabo siguiendo los mismos pasos en el *pipeline gráfico*, pero usando diferente terminología. Si se considera como aplicación la simulación de fluidos sobre una malla: en cada paso del tiempo, se calcula el estado del líquido de cada punto de la malla en el siguiente instante de tiempo teniendo en cuenta el estado actual del punto y de sus vecinos. Los pasos a seguir son:
   1) El programador especifica una primitiva geométrica para cubrir todo el dominio de la computación. El proceso de *rastering* genera un fragmento en cada píxel cubierto por la geometría definida. Para el ejemplo, la geometría debe cubrir una malla de fragmentos del mismo tamaño que el dominio de la simulación de fluidos considerada.

2) Cada fragmento es sombreado mediante un programa SPMD de fragmentos de propósito general. Para el ejemplo, en forma paralela se ejecuta el mismo programa a cada punto de la malla para actualizar su estado.
3) El programa de fragmentos calcula el valor del fragmento mediante la combinación de operaciones matemáticas y accesos a la memoria global para la obtención (*gather*) de los resultados. En el ejemplo, cada punto de la malla accede al estado de sus vecinos para computar su próximo estado.
4) Los resultados de una etapa son almacenados en la memoria global y pueden ser usados para el cómputo de los siguientes estados. En el ejemplo, el estado actual de los fluidos será utilizado para el cómputo de los próximos estados.

3. Programación de propósito general sobre GPU modernas:

Entre las principales dificultades a las que se enfrentaron los programadores de aplicaciones de propósito general en GPU (GPGPU) fue pensar en cómo resolver una aplicación que no tiene nada en común con los gráficos y programarla con APIs gráficas. Los programas debían seguir el *pipeline gráfico*, no se podía acceder a estados internos del *pipeline* sin antes pasar por los anteriores. Se propusieron muchos entornos de programación, los cuales facilitaban al programación de aplicaciones no gráficas para GPU, entre ellos podemos encontrar BrookGPU (Buck, 2004), Sh (McCool M. D., 2004), Microsoft's Accelerator (Tarditi, 2006,), RapidMind (McCool M., 2006), Peak*Stream* (PeakStream.) CTM (Hensley, 2007) de AMD. Hoy uno de los más populares y al cual se hará referencia en este libro es CUDA de Nvidia (NVIDIA., NVIDIA CUDA C Programming Guide. Versión 4.0., 2011) (Sanders, 2010) (Kirk, 2010).

Las aplicaciones de propósito general para GPU se estructuran de la siguiente manera:

1) El programador define el dominio de la computación como una estructura *grid* de *threads*.
2) Un programa de propósito general es ejecutado por cada *thread* en paralelo sobre distintos datos según el modelo de programación es SPMD.
3) Cada *thread* realiza cálculos mediante una combinación de operaciones matemáticas y de accesos de escritura y lectura en la memoria global. La diferencia con las dos formas de programación anteriores está en que el mismo buffer puede ser usado tanto para leer como para escribir, por ejemplo se pueden programar algoritmos *in-place*.

4) Los resultados de la operación que se guardan en la memoria global pueden ser usados por otras computaciones de la misma aplicación en la GPU.

El modelo de programación propuesto es muy poderoso por las siguientes razones:

- Permite al hardware tomar ventaja del máximo paralelismo de datos de la aplicación. Esto se logra mediante la especificación explícita del paralelismo en el programa.
- Ofrece un balance entre la generalidad en la programación a costa de algunas restricciones como son el modelo de programación SPMD, la comunicación de los datos entre las unidades de computación, entre los *kernels* (funciones ejecutadas en la GPU e invocadas desde el *host*), entre otras.
- Elimina la complejidad que enfrentaban los programadores de las primeras GPU, los cuales debían programar su aplicación de propósito general utilizando la interfaz gráfica. Los programas actuales son expresados en un lenguaje de programación familiar, generalmente resultado de la extensión de un lenguaje secuencial como C. Además son más simples y fáciles de construir y depurar, llegando a tener algunos herramientas que favorecen estas tareas.

La adopción de los sistemas CPU-GPU obedece principalmente a que comparados con las supercomputadoras clásicas (*mainframes*) proveen:

- Un poder de cálculo comparable con supercomputadoras.
- Hardware de bajo costo.
- Bajo costo de mantenimiento.
- Alta escalabilidad. Además de las características ya enunciadas de las arquitecturas, varios sistemas CPU-GPU pueden ser conectados con redes de alta performance.
- Reducido costo de programación, el código no paralelo se sigue usando en la CPU mientras que el paralelo puede seguir usándose a pesar de los constantes avances de las arquitecturas.

Desde la modificación de la arquitectura propuesta por Nvidia, donde la GPU se convirtió en un dispositivo totalmente programable, y con el desarrollo de herramientas como CUDA para su programación mediante la extensión de lenguajes de alto nivel como C, C++ y Fortran (Gehrke, 1996), la GPU es considerada una arquitectura apta para resolver problemas masivamente paralelos.

Dadas todas estas características, el resultado es un modelo de programación, el cual permite extraer el máximo provecho de la potencialidad del hardware de la GPU y expresar aplicaciones cada vez más complejas mediante programación de alto nivel.

2.6. Resumen

Desde sus inicios, la GPU fue diseñada como un procesador paralelo. El *pipeline gráfico* implementado por las primeras GPU provee paralelismo a nivel de etapas. Desde sus inicios se diferenció de las CPU, sus filosofías de desarrollo son distintas, los avances de las CPU obedecen a ofrecer mejores prestaciones y reducir el tiempo de las aplicaciones secuenciales existentes. No ocurre lo mismo con las GPU donde los avances se deben a la necesidad de optimizar el throughput de aplicaciones paralelas.
En ambos casos, el número de *cores* seguirá aumentando en la medida que sea posible. Por su parte, las GPUs continuarán disfrutando de la evolución arquitectónica vigorosa. Es de esperar que la brecha entre ambas arquitecturas se siga manteniendo en un tiempo en el futuro.
Respecto al rendimiento de las aplicaciones desarrolladas en ambas sistemas, en el caso de las CPU es necesario instaurar a la computación paralela como la alternativa. Respecto a la GPU, si bien su utilización en aplicaciones de propósito general es muy reciente, los rendimientos alcanzados son muy buenos. Esto unido a los constantes avances en las nuevas arquitecturas y el constante desarrollo de herramientas que facilitan su programación, permiten enunciar que se constituirán en un recurso económico y válido para el desarrollo de aplicaciones en general.
En este capítulo se analizó la historia de la GPU, realizando un resumen de la evolución del hardware de gráficos hacia una mayor capacidad de programación para finalizar con un análisis de las nuevas tendencias de computación: GPGPU. Comprender los hechos históricos ayuda al lector a entender mejor el estado actual y las tendencias futuras en su evolución.

2.7. Ejercicios

2.1. De las supercomputadoras detalladas en el top500, analizar cuáles poseen GPU como co-procesadores. ¿Qué características tienen en cada caso?

2.2. Analice las características de las CPU y GPU contemporáneas. Entre las características principales considere: capacidad de procesamiento, capacidad de memoria, ancho de banda, entre otros.

2.3. ¿Qué debe tener en cuenta para utilizar a la GPU como una computadora paralela para resolver una aplicación? ¿Es importante conocer las capacidades de cómputo para desarrollar una aplicación en la GPU?

2.4. Si desarrolla una aplicación utilizando las capacidades 1.0 de la GPU ¿Puede ser ejecutado en una GPU con capacidades superiores? ¿Cuáles serían las desventajas?

2.5. Si desarrolla una aplicación utilizando las capacidades 2.0 de la GPU
   a. ¿Puede ser ejecutado en una GPU con capacidades inferiores? Justifique su respuesta.
   b. ¿Qué características debe tener el código para:
      1. Ejecutar en cualquier arquitectura de GPU
      2. Aprovechar las ventajas de la arquitectura subyacente.

2.6. Analice si las siguientes aplicaciones se pueden resolver en la GPU
   a. Suma de dos matrices
   b. Multiplicación de matrices

CAPÍTULO 3

Programación de GPU: Modelo de Programación CUDA

En su inicio, la GPU fue usada para aplicaciones específicas: aplicaciones gráficas, tridimensionales o videojuegos. En los últimos años su evolución permitió el uso en un sinnúmero de aplicaciones computacionales altamente paralelizables, constituyéndose en una alternativa válida a las supercomputadoras. El éxito de su notable expansión se basa fundamentalmente en su gran potencia de cálculo, bajo costo, reducido consumo relativo a su poder computacional y la posibilidad de complementarse con la CPU. Todas estas características le permiten comportarse como un co-procesador para resolver tareas altamente paralelas, mientras que el código menos paralelizable puede continuar ejecutándose en la CPU.

Existen diferentes alternativas para procesamiento en GPU, la más ampliamente utilizada es la tarjeta Nvidia, para la cual se ha desarrollado un *kit* de programación en C: CUDA (*Compute Unified Device Architecture*). CUDA ha sido diseñado para simplificar el trabajo de sincronización y comunicación de *threads,* y la comunicación con la GPU, provee una interfaz CPU-GPU. Define una arquitectura de GPU, el modelo de programación asociado es SIMD (Simple Instrucción-Múltiples Datos) y el modelo de ejecución de programas es SIMT (Simple Instrucción-Múltiples *Threads*).

En este capítulo se introduce la programación de la GPU mediante CUDA, detallando sus características básicas: definición de funciones *kernel*, características, *threads*, *bloques* y *grid*, organización y asignación de recursos, ejecución de un *kernel*, entre otras. Además, se muestran ejemplos de aplicaciones simples.

3.1. Introducción a CUDA

Como se mencionó en el capítulo anterior, las GPU implementan en hardware el *pipeline gráfico*, el cual al ser un algoritmo inherentemente paralelo con cómputo intensivo, determinó la evolución de las GPU, basaron sus avances en satisfacer la demanda de mejor rendimiento en la ejecución del *pipeline*. Esto unido a su bajo costo hace que cualquier computadora personal cuente con una GPU. Sin embargo, en un principio su programación no era tan simple, ella implicaba conocer detalles de la arquitectura del pipeline gráfico y utilizar APIs como de OpenGL (Wright, 2007) o DirectX (Root, 1999) (Walsh, 2008).
En el año 2006 NVIDIA presenta la tecnología *Compute Unified Device Architecture* (CUDA) para su última generación de tarjetas gráficas, la serie 8 (G80). CUDA propone una filosofía integradora, con un lenguaje de programación genérico como es C y la arquitectura paralela de una GPU, desvinculándose además del *pipeline gráfico* (En el capítulo anterior se mostró como cambió la programación de la GPU a través de los años).
La tecnología CUDA permite considerar a la GPU como una arquitectura paralela para la resolución de problemas de propósito general. El desarrollo de dichas aplicaciones paralelas es posible principalmente, por dos razones:

1. Las tarjetas gráficas NVIDIA, por las características antes enunciadas, son una componente común en la mayoría de las computadoras personales actuales.
2. Es de fácil aprendizaje, más para aquellos programadores cercanos a lenguajes tipo C o C++.

En este capítulo se detallan las propiedades principales de CUDA para la programación de aplicaciones de propósito general en cualquier GPU de Nvidia y cuya arquitectura sea igual o posterior a las arquitecturas de la serie G80.

3.2. Arquitectura y Modelo de Programación CUDA

En las próximas secciones se muestra la arquitectura de la GPU según CUDA y las características del modelo de programación propuesto.

3.2.1. Arquitectura de GPU según CUDA

CUDA presenta a la arquitectura de la GPU como un conjunto de multiprocesadores MIMD (*Múltiple Instrucciones-Múltiple Datos*). Cada multiprocesador posee un conjunto de procesadores SIMD (*Simple Instrucción-Múltiples Datos*).
Respecto a la memoria, existen numerosos modelos conviviendo en esta arquitectura: cada procesador SIMD posee una serie de registros a modo de memoria local (sólo accesible por el procesador), a su vez cada multiprocesador posee una memoria compartida o *shared* (accesible por todos los procesadores SIMD del multiprocesador) y finalmente la memoria global, la cual es accesible por todos los multiprocesadores y, por ende, por todos y cada uno de los procesadores SIMD. En el siguiente capítulo se describe cada una de las memorias de la GPU.

3.2.2. Modelo de Programación CUDA

CUDA propone un modelo de programación SIMD (*Simple Instrucción-Múltiples Datos*) con funcionalidades de procesamiento de vector (Flynn, 1995). La programación de GPU se realiza a través de una extensión del lenguaje estándar C/C++ con constructores y palabras claves. La extensión incluye dos características principales: la organización del trabajo paralelo a través de *threads* concurrentes y la jerarquía de memoria de la GPU con sus diferentes costos de acceso.
Los *threads* en el modelo CUDA son agrupados en *Bloques,* los cuales se caracterizan por:

- El tamaño del *bloque*: cantidad de *threads* que lo componen. Es determinado por el programador.
- Todos los *threads* de un *bloque* se ejecutan sobre el mismo SM.
- Los *threads* de un *bloque* comparten la memoria *shared*, la cual pueden usar como medio de comunicación entre ellos.

Varios *bloques* forman un *Grid* y los *threads* de diferentes *bloques* de un *grid* no se pueden comunicar entre si, esto permite que el administrador de *bloques* sea rápido y flexible, no tiene en cuenta el número de SM utilizados para la ejecución del programa.
Además de las variables en la memoria compartida, los *threads* tienen acceso a otros dos tipos de variables: *Locales* y *Globales*. Las variables locales residen en la memoria DRAM de la tarjeta, son privadas a cada *thread*. Las variables globales también residen en la memoria DRAM de la tarjeta, se diferencian de las locales en que pueden ser accedidas por todos los *threads* aunque pertenezcan a

distintos *bloques*. Esto lleva a una manera de sincronización global de los *threads*. Como la memoria DRAM es más lenta que la memoria compartida, los *threads* de un *bloque* se pueden sincronizar mediante una instrucción especial, la cual es implementada en memoria compartida. La figura 3.1 muestra la relación de los *threads* y la jerarquía de memoria.
Además de la memoria local, *shared* y global, existen dos espacios adicionales de memoria de sólo lectura, ambos pueden ser accedidos por todos los *threads* de un *grid*. Éstas son la *memoria de constantes* y la *memoria de texturas*, ambas optimizadas para determinados usos. En el capítulo 4 se explica detalladamente la jerarquía de memoria de la GPU.

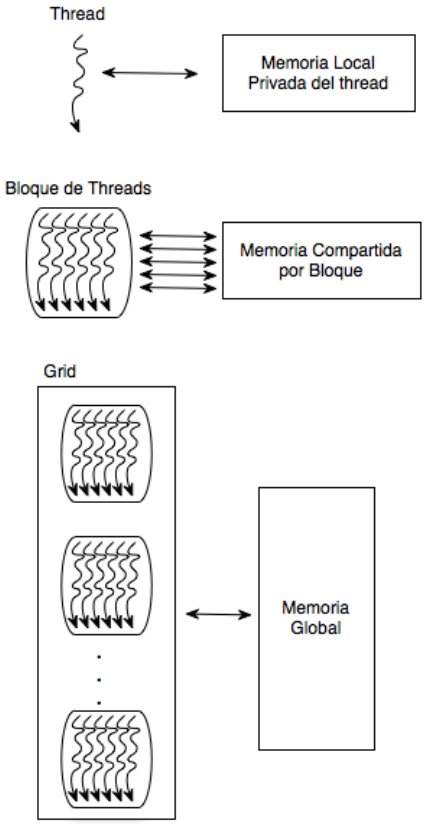


Figura 3.1. *Threads* y jerarquía de memoria

En un programa CUDA se diferencian dos ámbitos: el *host* y el dispositivo (*device)*. El *host* es el ordenador al cual está conectada la tarjeta gráfica y será quien rija su comportamiento. El *device* es la tarjeta gráfica. La comunicación de datos entre el *host* y el dispositivo se lleva a

cabo a través de la memoria global, de constante y de textura. Esta se realiza, como se dijo en el capítulo anterior, mediante el *PCI-Express*.

3.3. Generalidades de la Programación con CUDA

No todos los problemas pueden ser resueltos en la GPU, los más adecuados son aquellos que pueden resolverse mediante la aplicación del paradigma paralelo de datos, es decir aplican la misma sentencia o secuencia de código a todos los datos de entrada. Se puede decir que una solución de un problema en GPU será más ventajosa respecto a la solución en la CPU si la aplicación tiene las siguientes propiedades:

- El algoritmo tiene un orden de ejecución cuadrático o superior: el tiempo necesario para realizar la transferencia de datos entre la CPU y la GPU tiene un gran costo, el cual no suele verse compensado por el bajo costo computacional de un método lineal.
- Es mayor la carga de cálculo computacional en cada *thread*: de nuevo para compensar el tiempo de transferencia de información es conveniente que cada *thread* posea una carga computacional considerable.
- Es menor la dependencia entre los datos para realizar los cálculos, esto es posible si cada SM sólo necesita de los datos de su memoria local o compartida y no necesita acceder a memoria global, la cual tiene un acceso más lento.
- Es menor la transferencia de información entre CPU y GPU. La situación óptima es cuando la transferencia sólo se realiza una vez, al comienzo y al final del proceso. Esto significa una transferencia de los datos de entrada, desde la CPU a la GPU, y una al final, desde la GPU a la CPU, para obtener los resultados. Es bueno no tener transferencias intermedias, ya sea de resultados parciales o datos de entradas intermedios.
- No existen secciones críticas, es decir, varios procesos no necesitan escribir en las mismas posiciones de memoria, las lecturas de memoria global y compartida puede ser simultánea, pero las escrituras en la misma posición de memoria plantea un acceso a un recurso compartido, lo cual implica contar con mecanismo de acceso seguro. Este proceso hace más lenta la solución del proceso global.

Además, es necesario que las estructuras de datos en la aplicación que ejecuta en la CPU se adapten o puedan transformarse a estructuras más simples del tipo matriz o vector a fin de poder ser compatibles con las estructuras que maneja la GPU.

Como se mencionó en secciones anteriores, el modelo de programación CUDA asume que los *threads* CUDA se ejecutan en una unidad física distinta, la cual actúa como co-procesador (*device*) al procesador (*host*). Como CUDA C es una extensión del lenguaje de programación C, permite al programador definir funciones C, llamadas *kernels*, las cuales al ser invocadas son ejecutadas en paralelo por *N threads* diferentes en la GPU. Para las placas anteriores a la generación de GPU Fermi, los *kernels* se ejecutan de forma secuencial en el *device*. En la nueva generación de GPU, GF100 y siguientes, es posible ejecutar *kernels* en paralelo. En la figura 3.2 se muestra un diagrama de la arquitectura del sistema CPU-GPU donde se ejecutaran los programas CUDA C.

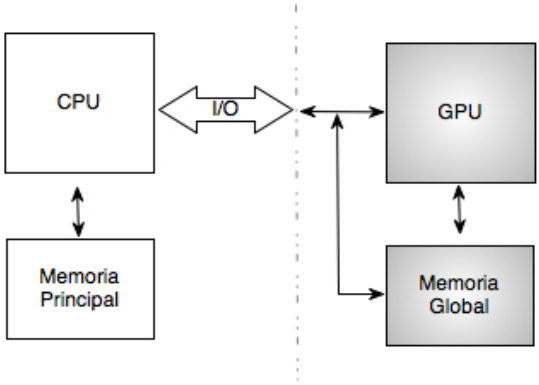


Figura 3.2. Arquitectura CPU-GPU

Los *kernels* son el componente principal del modelo de programación de CUDA, son funciones invocadas desde el *host* y ejecutadas en el *device*. Cuando se invoca un *kernel*, éste se ejecuta *N* veces en *N*threads diferentes. Cada *threads* se diferencia de los demás por su identificador, el cual es único y accesible en el *kernel* a través de una variable interna y predefinida de CUDA (*built-in*) llamada *thread*Idx. A través de *thread*Idxse puede definir el comportamiento específico de cada uno de los *threads*.

Para la definición de un *kernel* se deben respetar varias condiciones, las cuales se enuncian a continuación:

- El tipo de la función *kernel* es void.
- Debe llevar la etiqueta __global__, la cual identifica a un *kernel* y determina que la función es invocada desde el *host* y ejecutada en el *device*.
- Todos los *threads* que se activen durante la ejecución del *kernel*, ejecutan el mismo programa, el cual coincide con el *kernel* que lo activó.

- El número de *threads* es conocido antes de la ejecución del *kernel*, ellos serán agrupados, según se indica en la invocación, en grupos denominados *bloques*. Todos los *bloques* tienen igual número de *threads*.

Existe una jerarquía perfectamente definida sobre los *threads* de CUDA. Los *threads* se agrupan en *bloques*, los cuales se pueden ver como vectores (una dimensión) o matrices (dos o tres dimensiones). Como se mencionó antes, los *threads* de un mismo *bloque* pueden cooperar entre si, compartiendo datos y sincronizando sus ejecuciones. Sin embargo, *threads* de distintos *bloques* no pueden cooperar entre si. Los *bloques* a su vez, se organizan en *grid*, éste l cual puede ser de una o dos dimensiones (En las nuevas arquitecturas, GF100, se admiten tres dimensiones). Cuántos *bloques* y *threads* por *bloque* tendrá un *grid* son valores establecidos antes de la invocación, los cuales permanecen invariables durante toda la ejecución del *kernel*.
Dada la organización que provee CUDA para los *threads* y como cada uno de ellos tiene un identificador único: *thread*Idx, esta es una variable de tres dimensiones, dim3 en CUDA (Ver apéndice A). Más específicamente *thread*Idx tiene tres componentes (*x, y, z*), permitiendo según la dimensión del *bloque*, identificar unívocamente a cada *thread*. Cuando el *bloque* es de una dimensión, las componentes *y* y *z* tienen el valor 1, en el caso de un *bloque* de dos dimensiones sólo la componente *z* tiene el valor 1. Lo mismo ocurre con los *bloques* y los *grids*, pero para ellos CUDA tiene definida tres variables: blockIdx y blockDim para *bloques*, y gridDim para *grid*, todas de tipo dim3. blockIdx permite identificar a los *bloques* y las variables blockDim y gridDim contienen el tamaño de cada *bloque* y de cada *grid*, respectivamente. Todas estas variables son variables *built-in,* sólo accesibles dentro del *kernel*. Al igual que *thread*Idx, tienen tres componentes. Los campos de las dimensiones (identificador o tamaño) no definidas tienen el valor por omisión 1.

3.4. Estructura de un Programa en CUDA

Un programa CUDA está compuesto de una o más fases, las cuales son ejecutadas o en el *host* o en el dispositivo. Aquellas partes que exhiben poco o nada de paralelismo se implementan en el código a ejecutar sobre el *host*, no así las que pueden ser resueltas aplicando paralelismo de datos, éstas son implementadas a través de código que se ejecutará en el dispositivo, en este caso la GPU. Si bien en el programa CUDA existen dos partes bien diferenciadas, será el compilador el responsable de su diferenciación. Para ello, el código

desarrollado para ejecutarse en el *host* será compilado con el compilador estándar de C (o el del lenguaje secuencial utilizado) y ejecutado en la CPU como un proceso común. El código a ejecutarse en el dispositivo, escrito en C extendido con palabras claves que expresan el paralelismo de datos y las estructuras de datos asociadas, será compilado con el compilador propio de CUDA (*nvcc* por ejemplo (NVIDIA, 2007)).

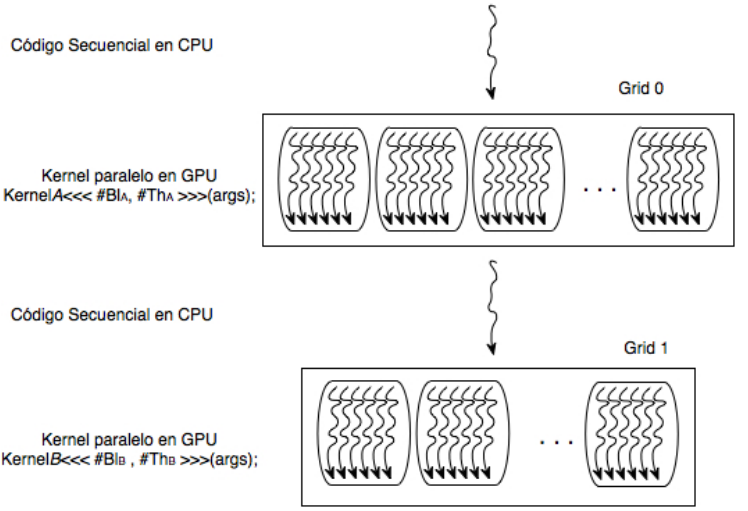


Figura 3.3. Esquema de la estructura de un programa en CUDA

Un *kernel* se diferencia de una llamada a una función común en el código secuencial porque además de las especificaciones enunciadas anteriormente, en su invocación se establecen los parámetros necesarios para configurar la ejecución. En la figura 3.3 se muestra la estructura de un programa CUDA y la forma cómo se realiza la invocación a un *kernel*, en este caso particular se invocan dos *kernel*: *kernelA* y *kernelB*, indicando para cada caso cuántos *bloques*($\#Bl_A$, $\#Bl_B$) y *threads* por *bloques* ($\#Th_A$, $\#Th_B$) resolverán el problema respectivamente. Dichas especificaciones son realizadas siguiendo la sintaxis

"función_*kernel*" <<<#Bl, #Th>>>(argumentos);

donde los argumentos de la función *kernel* son expresados en *(argumentos)*.

Para poder realizar una explicación más simple, la misma se va a realizar a través de un ejemplo sencillo y apto para ser resuelto en GPU: la suma de dos vectores. Dados dos vectores *A* y *B,* su suma

algebraica resulta en otro vector *C*, donde cada componente de *C* es al suma de las correspondientes componentes de *A* y *B*. En la figura 3.4 se muestra gráficamente el proceso.

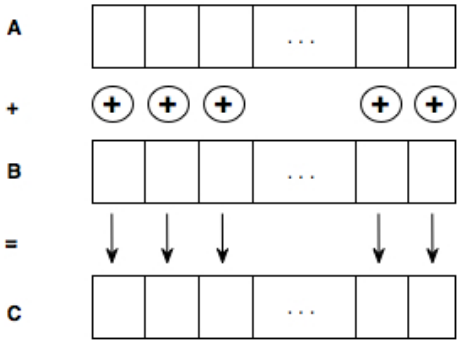


Figura 3.4. Suma de dos vectores

El siguiente pseudo-código (Figura 3.5) muestra la solución computacional de la suma de vectores en la CPU.

```
#define N 10
void Suma_vec( int *a, int *b, int *c ) {
  int i = 0; // Los vectores van desde 0..N-1
  for (i=0; i < N; i++)
c[i] = a[i] + b[i];
}
int main( void ) {
  int a[N], b[N], c[N];
inicializa(a);     //Inicializa los vectores de entrada a y b
  inicializa(b);
  Suma_vec( a, b, c );
  mostrar(a,b,c);  // muestra el resultado
  return 0;
}
```

Figura 3.5. Suma de vectores en CPU

Según lo mostrado en la función *Suma_vec()*, el cálculo de cada una de las componentes del vector *C* es independiente del resto y, en consecuencia, pueden ser realizados en paralelo. Por ejemplo una solución paralela para una arquitectura con dos núcleos (*cores*) sería de la forma mostrada la figura 3.6. Un proceso se encarga de las componentes pares (figura 3.6(a)) y el otro de las componentes impares (figura 3.6(b)).

```
// CORE #0
voidSuma_vec( int *a, int *b, int *c ) {
int i = 0; // Sumará las posiciones pares
for (i=0; i < N; i+=2)
c[i] = a[i] + b[i];
}
```

(a) Componentes Pares

```
//CORE #1
voidSuma_vec( int *a, int *b, int *c ) {
int i = 1; // Sumará las posiciones impares
for (i=1; i < N; i++)
c[i] = a[i] + b[i];
}
```

(b) Componentes Impares

Figura 3.6. *Suma_vec* paralela con 2 procesos

En caso de querer resolverlo con más tareas paralelas, se deberá modificar el código de manera tal que permita la creación de procesos para realizar la suma de las componentes, además de contar con el soporte de hardware necesario para la ejecución paralela de los procesos indicados. A continuación se verá cómo resolver el problema en la GPU mediante CUDA C.

Como se dijo antes, un programa CUDA consta de una o más fases, las cuales se ejecutan en la CPU o en la GPU. Aquellas fases que exhiben paralelismo de datos pueden resolverse en paralelo y ejecutarse en la GPU. Al realizarse las llamadas a funciones en la GPU a través de una función *kernel*, es en ese momento cuando se determina el número de *threads* con los cuales se resolverá en paralelo el problema. La multiplicación de vectores es una aplicación paralela de datos, por lo cual puede ser resuelta en la GPU. Para ello se debe definir una función *kernel* donde cada *thread* es responsable de calcular la suma de un elemento. De esa manera el número de *threads* dependerá de la dimensión de los vectores a sumar, por ejemplo para sumar vectores de 100 elementos serán necesarios 100 *threads*, la figura 3.7 muestra gráficamente cómo se resolverá el problema en la GPU.

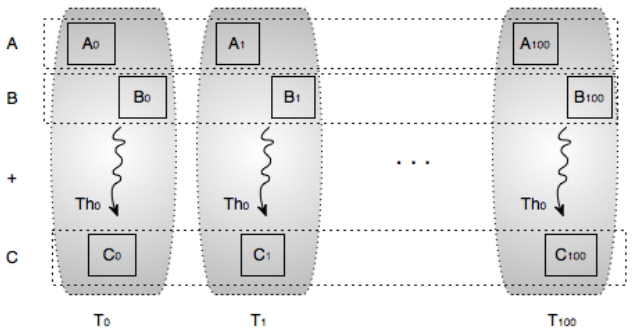


Figura 3.7. Suma de vectores en GPU

Como cada *thread* calcula una componente del vector suma, cada *thread*T_i realizará la operación $A_i + B_i = C_i$.
En la figura 3.8 se muestra el código CUDA C correspondiente a la función main(). Como puede observarse, la función responsable de realizar la suma de vectores tiene las mismas características que la definida en la figura 3.5, pero incluye la invocación a la función *kernel* en la GPU. En la línea 13 se realiza la invocación al *kernelSuma_vec* con los parámetros actuales *(dev_a, dev_b, dev_c)*. Además se especifica entre <<<…>>> el número de *bloques* y la cantidad de *threads* por *bloque*, en este caso se define un *bloque* con 100 *threads*.

```
1.  #defi
    ne N 100
2.  int
    main( void ) {
3.  int
    a[N], b[N], c[N];
4.  int
    *dev_a, *dev_b, *dev_c;
5.  inicial
    iza(a);    //Inicializa los vectores de entrada a y b
6.  inicial
    iza(b);
7.  cuda
    Malloc( (void**)&dev_a, N * sizeof(int)); // reserva memoria en la GPU
8.  cuda
    Malloc( (void**)&dev_b, N * sizeof(int));
9.  cuda
    Malloc( (void**)&dev_c, N * sizeof(int));
10. //
    copia los vectores A y B a la GPU
11. cuda
    Memcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
12. cuda
    Memcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );
13. Sum
    a_vec<<<1,N>>>( dev_a, dev_b, dev_c );
14. //
    copia el resultado desde la GPU en el vector C de la CPU
15. cuda
    Memcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);
16. cuda
    Free( dev_a ); // libera la memoria reservada de la GPU
17. cuda
    Free( dev_b );
```

```
18.                                                    cuda
    Free( dev_c );
19.                                                    mostr
    ar_resultados ( C); // Muestra los resultados
20.                                                    retur
    n 0;
21.                                                    }
```

Figura 3.8. Suma de vectores en el *host*

El código fue organizado de manera tal que todo lo relacionado con la ejecución en el dispositivo está agrupado, no siempre es posible hacerlo de esta forma. La versión para la GPU tiene varias diferencias con la implementación secuencial sobre la CPU, todas ellas relacionadas a aspectos de la ejecución de la suma de vectores en el dispositivo, por ejemplo la inicialización y las transferencias a la memoria de éste. En las siguientes secciones se detallan las principales.

3.4.1. Transferencia de datos CPU-GPU

En el sistema CPU-GPU, cada uno de las componentes, *host* y dispositivo, tienen su propio espacio de memoria, las cuales son independientes. Para resolver un problema en la GPU, el programador necesita transferir los datos de entrada del programa la GPU y, una vez obtenidos los resultados, transferirlos a la CPU. CUDA provee funciones para realizar estas tareas, en las líneas 11 y 12 se lleva a cabo la transferencia de datos desde la CPU a la GPU y en la línea 15 la transferencia es en el sentido inverso. Observe que la función responsable de la transferencia es la misma, la diferencia radica en el último parámetro. Éste indica la constante el sentido de la transferencia a través de una *built-in*, más adelante se explica con más detalle las transferencias entre las memorias.
Las transferencias de datos entre la CPU y la GPU se dan a nivel de la memoria principal de cada uno.
En la figura 3.9 se muestra la visión general del modelo de memoria CUDA de la GPU, detallando las posibles transferencias y accesos a los distintos tipos de memoria (En el capítulo 4 se abordará detalladamente).

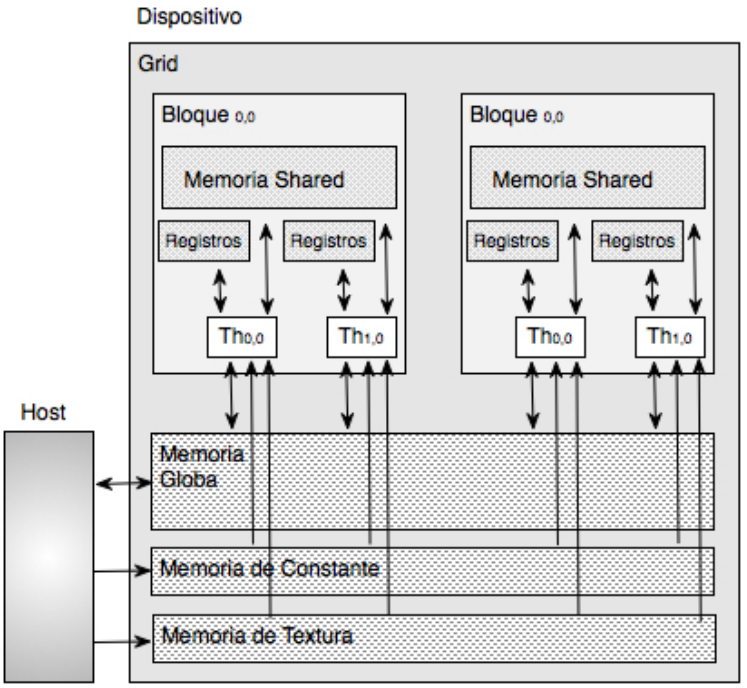


Figura 3.9. Jerarquía de memoria y accesos

Se puede observar el acceso desde:

- El *host*:
  - A la memoria global del dispositivo, tanto para lectura como para escritura.
  - A la memoria de constante del dispositivo, pero sólo para escritura.
  - A la memoria de textura del dispositivo para sólo escritura.
- El dispositivo:
  - A la memoria de registros y a la local, propias de cada *thread*.
  - A la memoria *shared*, memoria compartida por todos los *threads* de un *bloque*. Cada *thread* puede leer o escribir en esta memoria.
  - A la memoria de constante y de textura. Estos accesos sólo son de lectura.
  - A la memoria global, a la cual pueden acceder todos los *threads* tanto para lectura como para escritura.

En esta sección, y para todo este capítulo, sólo se tendrá en cuenta la memoria global.

Para realizar las transferencias a memoria, es necesario gestionar la memoria global en el dispositivo, para ello CUDA provee al programador con funciones para asignar y liberar espacio de memoria en la GPU. En la Figura 3.8 se muestran las funciones de la API para la asignación (líneas 7, 8 y 9) y liberación (líneas 16,17 y 18) de la memoria global. La función cudaMalloc() sólo puede ser llamada en el código del *host* y permite asignar a una variable una cierta cantidad de espacio, todo es indicado como parámetro en la invocación de la función. Puede observarse una similitud entre cudaMalloc() y la función malloc() de ANSI C. Estas son las características que hacen fácil el aprendizaje de CUDA.
El primer parámetro de cudaMalloc() es la dirección de una variable puntero, la cual contendrá la dirección del objeto asignado en la memoria global después de la llamada. Puede solicitarse espacio para cualquier tipo de objetos. El segundo parámetro especifica el tamaño, en bytes, del objeto a asignar. Para el ejemplo aquí señalado, se puede observar que se reserva lugar para tres vectores, *dev_a*, *dev_b* y *dev_c*. Los dos primeros se corresponden con los vectores de entrada *a* y *b*, y en *dev_c* se calculará el resultado de la suma, el cual será copiado en *c*. Para mayor claridad, en los ejemplos se antepuso el prefijo "*dev_*" cuando la variable es del dispositivo.
Para liberar memoria global en la GPU, CUDA provee la funcion cudaFree(), la cual libera el espacio de memoria apuntado por la variable que recibe como parámetro.
Una vez asignada la memoria en el dispositivo a cada uno de los objetos de datos con los que se va a trabajar, si es necesario para el problema, se deben transferir los datos desde el *host* al dispositivo. Como se mencionó antes, ésto se hace a través de la función cudaMemcpy(), la cual tiene cuatro parámetros, ellos son:

- Primer parámetro: Un puntero a la ubicación destino de la copia.
- Segundo parámetro: Puntero a la ubicación desde donde se van a copiar los datos.
- Tercer parámetro: Cantidad de bytes a copiar.
- Cuarto parámetro: Indica el sentido de la transferencia. Esto se hace a través de 4 constantes pre-definidas, ellas son:
  - cudaMemcpyHostToHost: Copia de la memoria principal a la memoria principal.
  - cudaMemcpyHostToDevice: De la memoria principal a la memoria del dispositivo.
  - cudaMemcpyDeviceToHost: Copia desde la memoria del dispositivo a la memoria principal del host.

- cudaMemcpyDeviceToDevice: De la memoria del dispositivo a la memoria del dispositivo. Este tipo de transferencia es muy rápida.

La función cudaMemcpy() permite realizar copias entre ubicaciones de memoria del *host* y del dispositivo, no entre varios dispositivos (Ambientes con múltiples GPU).

Resumiendo, en la figura 3.8 se muestra la función main() para resolver la suma algebráica de vectores en un sistema CPU-GPU. Es en el *host* donde se realizan las asignaciones y liberaciones de memoria en el dispositivo, la transferencia de datos entre el *host* y el dispositivo o a la inversa, y la activación de la ejecución en el dispositivo de la función *kernelSuma_vec* (para este ejemplo) sobre un determinado número de *threads*. En la próxima sección se define y analiza la función *kernel*.

3.4.2. Función kernel y Threads

Hasta el momento se analizó el código del *host*, cómo administrar la memoria global del dispositivo, transferir datos e invocar la función *kernel* en el dispositivo. En esta sección se analizará la función *kernel* en si y qué ocurre en el dispositivo cuando ésta es invocada.

En CUDA, la función *kernel* especifica el código a ser ejecutado por todos los *threads* en forma paralela en el dispositivo. Como los *threads* ejecutan el mismo código sobre distintos conjunto de datos, el código es un ejemplo del modelo de programación *Simple Instrucción-Múltiples Datos* (SIMD) o una generalización de éste, *Simple Programa-Múltiple Datos* (SPMD) (Leopold, 2001).

La figura 3.10 muestra la función *kernel* de la suma de vectores. Lo primero que se observa es la especificación de la palabra clave __global__ antes de la declaración de la función *Suma_vec()*. Esta palabra clave clasifica la función declarada a continuación, determinando que es una función *kernel*, la cual es invocada desde el *host* generando en el dispositivo los *threads* paralelos. Cada *thread* realizara las instrucciones indicadas en el cuerpo del *kernel*.

```
1.  __global__ void Suma_vec( int *a, int *b, int *c ) {
2.     int tid = threadIdx.x; // Identificador del thread
3.     c[tid] = a[tid] + b[tid];
4.  }
```

Figura 3.10. Función *kernel* de la suma de vectores

En CUDA existen tres calificadores de función, ellos son:

- __global__ indica una función *kernel,* la cual se ejecuta en el dispositivo y sólo puede ser invocada desde el *host*. Su invocación genera un *grid* de *bloques* con un número fijo e igual de *threads* cada uno.
- __device__ establece que la función es una función del dispositivo, se ejecuta en él y sólo puede ser invocada desde un *kernel* u otra función del dispositivo. CUDA tiene restricciones para las funciones del dispositivo, ellas son:
    - No pueden ser recursivas.
    - No pueden tener invocaciones indirectas a funciones a través de punteros.
- __*host*__ determina una función del *host* o simplemente una función de C tradicional a ejecutarse en el *host* y sólo invocada desde el main() u otra función de *host*. Si ningún calificador es antepuesto a la declaración de una función, por omisión, la misma es considerada función del *host*.

De esta manera cuando un programador desea desarrollar un programa CUDA a partir de uno existente, sólo deberá incluir las funciones a ejecutarse en el dispositivo, las existentes serán portables al nuevo sistema CPU-GPU.
En la siguiente tabla se resumen las características de cada uno de los calificadores claves, en ella se especifica donde se ejecutan y desde donde se invocan.

Calificador	*Se invoca desde*	*Se ejecuta*
__global__	*Host*	Dispositivo
__device__	Dispositivo	Dispositivo
__host__	*Host*	*Host*

Puede usar para la misma función los calificadores __host__ y __device__, lo cual indicará al compilador la generación de dos versiones de la misma función, una para el *host* y la otra para el dispositivo. Estas características refuerzan las propiedades de CUDA, cuando el mismo código puede ser ejecutado en ambos componentes del sistema, no es necesario desarrollar dos versiones distintas.
En el código de la figura 3.10 se observa la palabra clave *threadIdx.x*, la cual como se explicó antes hace referencia al identificador del *thread* en la primer dimensión. Para este problema y de la forma que fue planteado, sólo esta dimensión es suficiente para identificar a cada *thread*.

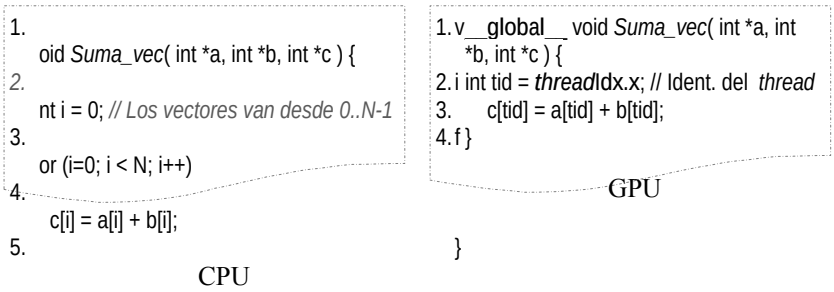


Figura 3.11. Función *Suma_vec* para la CPU y la GPU

En la figura 3.11 muestra ambas funciones, la definida para ejecutar en secuencial en la CPU (De la figura 3.5) y la correspondiente a la GPU (Figura 3.10). Se puede observar que la iteración sobre cada uno de los elementos del vector (línea 3-CPU) no está presente en el código de la GPU, en ella cada elemento es calculado por un *thread*, la determinación de qué elemento le corresponde a cada *thread* se calcula según el identificador *thread*Idx.x.

Al invocarse un *kernel*, se genera un *grid* con tantos *threads* como son indicados en la invocación. La cantidad de *threads* son, generalmente, los suficientes como para aprovechar las características del hardware y expresar el máximo paralelismo del problema a resolver.

Como se mencionó en la sección 3.2.2, los *threads* son organizados en una jerarquía de dos niveles, en el nivel superior se encuentra el *grid*, un *grid* tiene uno o más *bloques* de *threads*. Todos los *bloques* tienen la misma cantidad de *threads* y la misma organización. Cada *threads* dentro de un *bloque* es identificado con la variable reservada *thread*Idx y los *bloques* por blockIdx. Los *bloques* pueden tener hasta 512 *threads* en las GPU Nvidia de la generación de G80 y 1024 en las nuevas generaciones (GT200 y GF100), el número de *threads* por *bloque* es especificado en el campo max*Thread*sPerBlock de la estructura cudaDeviceProp, la cual especifica las propiedades del dispositivo.

Los *threads* de un *bloque* pueden ser organizados en arreglos de una, dos o tres dimensiones. Dependiendo de cómo se organiza el *bloque*, la identificación de cada *thread* será a través de la misma cantidad de dimensiones de la variable reservada *thread*Idx: x, y, z. Por ejemplo si el *bloque* tiene dos dimensiones, el par (*thread*Idx.x, *thread*Idx.y) identificará a un único *thread* de un *bloque*. Lo mismo ocurre con el *grid*, los *bloques* se pueden organizar en una y dos dimensiones para la G80 y GT200 y tres dimensiones para la GF100, siendo la variable reservada blockIdx y sus campos x, y y z los que permitirán la identificación de cada *bloque*. En la figura 3.12 se muestran *bloques* de una, dos y tres dimensiones. La figura 3.13 muestra un *grid* bidimensional con *bloques* tridimensionales.

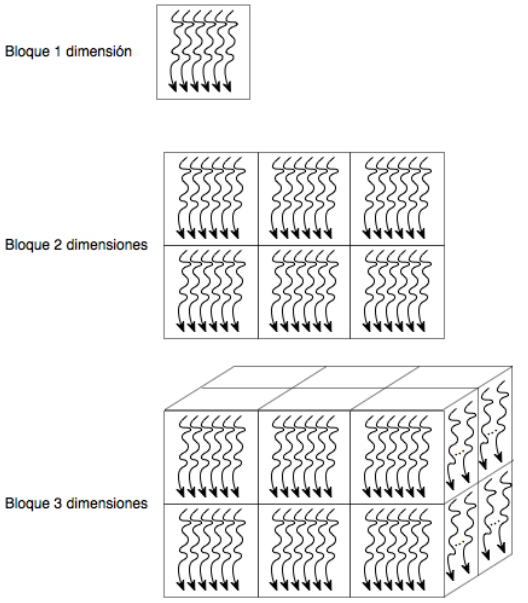


Figura 3.12. Organización de los *threads* en un *bloque*

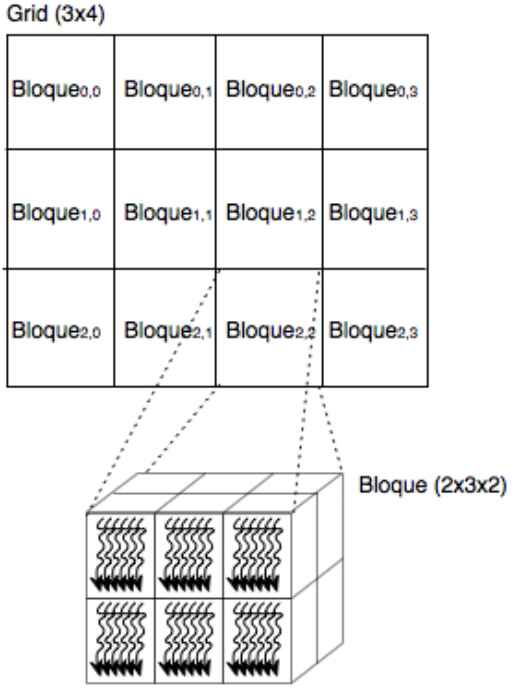


Figura 3.13. Organización de los *Bloques* en un *grid*

El número de *threads* por *bloque* y la cantidad de *bloques* en un *grid* pueden afectar la performance de la aplicación. En el capítulo 5 se explican los conceptos que ayudaran al programador a decidir el mejor número de *threads* y de *bloques* a fin de lograr un buen desempeño de la aplicación.

3.4.3. Multi-Bloques y Multi-Threads

En el ejemplo de la suma de vectores de la figura 3.10 se puede observar que no existe referencia al identificador de *bloque*, esto obedece a que la solución fue desarrollada para realizar la suma de un vector con tantas componentes como lo indique la máxima cantidad de *threads* por *bloque*. La pregunta es ¿Qué pasa si los vectores tienen más de max*Threads*PerBlock componentes? La respuesta está en el código de la figura 3.14, en la cual sólo se reflejan los cambios de la función main definida en la figura 3.8 y la modificación de la función *kernelSuma_vec* de la figura 3.10. En este caso se consideran vectores de 4096 elementos (4KB).

```
1.  #define N 4096
2.  int main( void ) {
    …
    //Inicializa los vectores de entrada a y b
    …
13. Suma_vec<<<N/512, 512>>>( dev_a, dev_b, dev_c );
    …
21. }
22. __global__ void Suma_vec( int *a, int *b, int *c ) {
23. int tid = blockIdx.x * blockDim.x + threadIdx.x; // Ident. del  thread
24.     c[tid] = a[tid] + b[tid];
25. }
```

Figura 3.14. Suma de vectores modificada

La solución presentada en el código anterior es válida cuando la cardinalidad del vector es múltiplo de la cantidad de *threads* por *bloque* a activar, 512 para este ejemplo. En caso que no lo fuera, el cálculo de la cantidad de *bloques* y de *threads* debería modificarse a fin de reflejar la condición que todos los *bloques* tienen el mismo número de *threads*. En esta nueva solución puede observarse la utilización de otra variable predefinida blockDim, la cual contiene la cantidad de *threads* del *bloque* según la dimensión. Como todos los *bloques* tienen la misma cantidad y organización de los *threads*, ésta

será constante para todos los *bloques* de un *grid*. Lo mismo ocurre en el caso de las dimensiones del *grid,* la variable gridDim. El número máximo de *bloques* por dimensión está indicado por el campo maxGridSizede la estructura cudaDeviceProp, siendo en los dispositivos actuales de 65.535 por dimensión. Esto significa que para el ejemplo de la figura 3.14 se puede tener un vector con 65.535*512 = 33.553.920 (31MB) elementos como máximo.
El problema presentado trabaja sobre una estructura de datos vectorial, lo cual sólo hace necesario tener una dimensión tanto de *bloques* como de *threads*. La pregunta ahora es ¿Cómo se hace para activar *grid* y *bloques* de dimensiones mayores a 1? La respuesta se detalla en las siguientes líneas de código, las cuales se incluyen en la función main y antes de la invocación al *kernel*.

```
dim3  my_blocks(128,100);
dim3  my_threads(16,16);
kernel<<<my_blocks, my_threads>>>( ...args... );
```

En estas líneas de código se define un ambiente de ejecución mediante dos variables de tipo dim3, *my_blocks* y *my_threads*. La primera describe laconfiguración del *grid*, en este caso 128x100 *bloques*. La segunda variable especifica las características de cada *bloque*, define *bloques* de 16x16hilos. Finalmente se invoca al *kernel* con la configuración establecida, en este caso 12.800 *bloques* con 256 *threads* cada uno, la estructura es de forma similar a la mostrada en la figura 3.15.

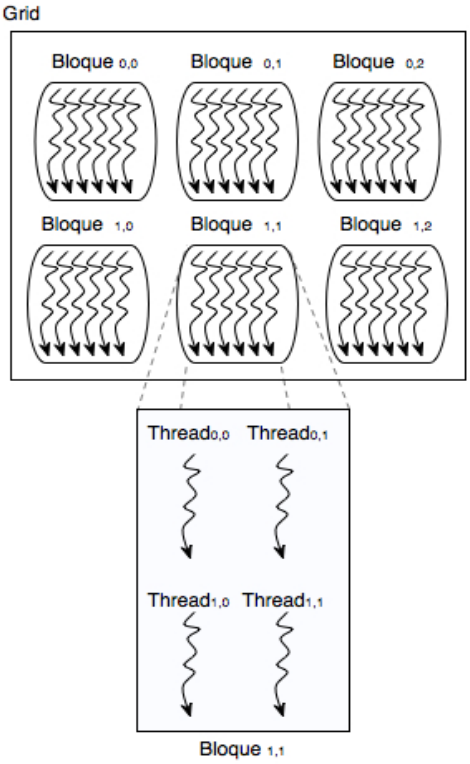


Figura 3.15. *Grid* de 2*3 y *bloques* de 2*2

Es importante destacar que la ejecución de cada *bloque* es independiente, los *bloques* se ejecutan en cualquier orden, puede ser en paralelo, en serie o ambos. Esto permite que cada *bloque* de *threads* sea asignado a cualquier SM y en cualquier orden, aportando así escalabilidad al código. En el caso de las GPU, el número de *bloques* del *grid* es determinado generalmente por el tamaño de los datos a procesar y no por el número de procesadores de la arquitectura.

3.4.4. Sincronización de Threads

La sincronización por barrera es un método simple y popular para coordinar actividades paralelas. Un ejemplo de la vida real donde se aplica una sincronización de este tipo es cuando 2 o más personas van a un lugar en un mismo auto. Al llegar a destino, cada una puede realizar una actividad distinta (actividades en paralelo). La sincronización es necesaria a la salida del lugar, las personas deben esperar por todas, es decir sólo pueden marcharse cuando todas hayan finalizado su tarea. Si así no lo hicieran se está en presencia de un grave problema, por ejemplo dejar olvidado a alguien.
Si bien para la suma de vectores no es necesaria la sincronización de los *threads*, existen muchas aplicaciones donde una sincronización de actividades entre las distintas tareas es vital para la resolución del problema. CUDA provee la sincronización por barrera de *threads* de un mismo *bloque* a través de la invocación a la función __sync*threads*(). Cuando un *kernel* hace un __sync*threads*(), todos los *threads* del *kernel* permanecerán en el lugar de la invocación hasta que todos los *threads* del *bloque* lleguen a ejecutar dicha sentencia. Esto asegura que todos los *threads* de un *bloque* completen la fase anterior a la invocación de __sync*threads*().
Cuando una sentencia __sync*threads*() es ejecutada, la deben hacer todos los *threads* del *bloque*. Qué ocurre cuando la función _sync*threads*() es parte de las sentencias de una sentencia condicional if-then-else, el programador debe asegurar que todos los *threads* o ninguno ejecuten dicha sincronización. Esto significa la misma sincronización, es decir si existe una sincronización por el if-then y otra por el else estas son consideradas distintas, ambos __sync*threads*() son diferentes puntos de sincronización por barrera. Esta clase de errores son comunes y llevan a una situación de bloqueo del sistema.
Respetando las características de una buena programación paralela, las sincronizaciones no deben llevar mucho tiempo, CUDA facilita estas características a través de la asignación de recursos en tiempo de ejecución a los *threads* de un *bloque*. Al momento de ejecutarse un *bloque*, se le asignan los recursos necesarios para todos los *threads* como una unidad, todos los *threads* de un *bloque* tendrán los mismos recursos de ejecución, asegurando su proximidad en el tiempo y evitando largas espera por la sincronización. Todas estas características de la implementación de la sincronización por barrera en CUDA hacen que exista una independencia entre el desarrollo de programas CUDA y su ejecución, independiente del número de *bloques* y dadas las características de estos, no existe un requerimiento para la ejecución de los *bloques*, es más se pueden ejecutar en cualquier orden.

3.5. Modelo de Ejecución

El modelo de programación CUDA asume que los *threads* se ejecutan en una unidad física distinta, la cual actúa como co-procesador (*device* o *dispositivo*) al procesador (*host*) donde se ejecuta el programa. Gráficamente se puede ver la ejecución de un programa CUDA C como lo muestra la figura 3.16.

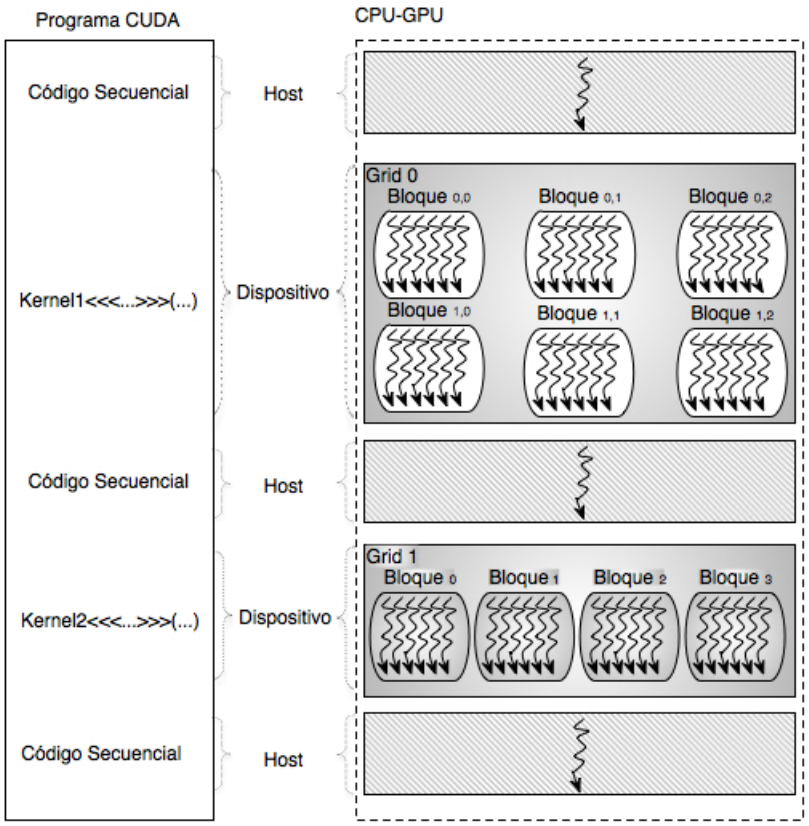


Figura 3.16. Ejecución de un programa CUDA C en un Sistema CPU-GPU

La arquitectura del sistema CPU-GPU se muestra en la figura 3.17. En ella se observa el flujo de procesamiento de un *kernel* en la GPU, pudiendo verse las distintas acciones a realizarse antes, durante y después de su ejecución. En orden cronológico, ellas son:

1. Copia de datos de la memoria de la CPU a la memoria global de la GPU.
2. La CPU ordena la ejecución del *kernel* en la GPU.

3. En cada uno de los *cores* (SP) se ejecutan los *threads* indicados en la invocación del *kernel*. Los *threads* acceden a la memoria global del dispositivo para leer o escribir datos.
4. El resultado final se copia desde la memoria del dispositivo a la memoria de la CPU.

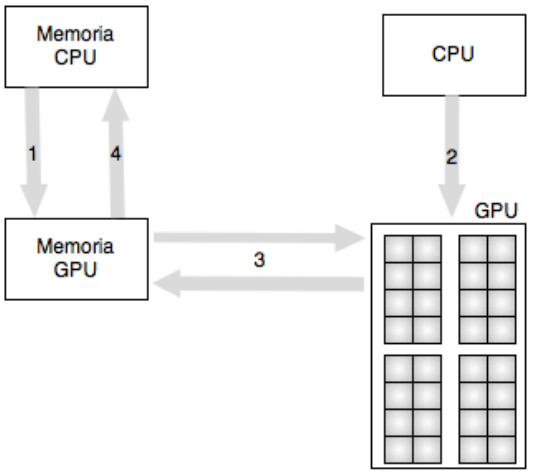


Figura 3.17. Flujo de procesamiento de un *kernel* en la GPU

El modelo de ejecución de los programas CUDA está íntimamente ligado a la arquitectura de la GPU. Como se vio en el capítulo 2, la arquitectura se basa en un arreglo escalable de SM.
Los *threads* se asignan a los recursos de ejecución en base a los *bloques*. Al estar los recursos de ejecución organizados por SM, por ejemplo en la GTX 275 (de la serie GT200) tiene 30 SM, cada SM cuenta con 8 procesadores escalares. Esto significa que en dicha arquitectura pueden ejecutar en paralelo hasta 240 *threads*. Como un SM necesita recursos para administrar los *threads*, los identificadores de los *bloques* y el estado de las ejecuciones, existe una cantidad de *threads* máxima por cada *bloque,* dicha limitación es impuesta por el hardware. En la GT200 el límite de los *threads* a administrar por SM es 1024, lo cual significa 30.720 *threads* ejecutando simultáneamente. La G80 administra hasta 768 *threads*, si el modelo tiene 16 SM (GTX 8800), entonces pueden co-existir 12.288 *threads* en ejecución.
Cuando un programa CUDA invoca un *kernel* con una configuración determinada, los *bloques* del *grid* son enumerados y distribuidos a los SM disponibles. Todos los *threads* de un *bloque* se ejecutan concurrentemente en un único SM. Cuando un *bloque* finaliza, un nuevo *bloque* es ejecutado en su lugar. La ejecución de los *bloques* es paralela/concurrente, en paralelo se ejecutan tantos *bloques* como SM

tenga el dispositivo y concurrente porque si existen más *bloques* que SM, estos se ejecutan concurrentemente entre sí. En otras palabras en paralelos se ejecutan grupos de tantos *bloques* como SM tenga la GPU.
Como se mencionó en el capítulo anterior, cada SM tiene un número fijo de SP, cada uno con sus registros (en la G80: 8 SP y 8K de registros, la GT200: 8 SP por SM y 16K para registros, y en la GF100: 32 SP por SM y 32K de registros).
Cada SM crea, gestiona y ejecuta los *threads* concurrentemente en hardware, sin sobrecarga de planificación. Para poder administrar cientos de *threads* en ejecución, el SM emplea una arquitectura denominada SIMT (*Simple Instrucción-Múltiples Threads*) (Patterson, 2008). La arquitectura SIMT es similar a la arquitectura vectorial SIMD (*Simple Instrucción-Múltiples Datos*) en el sentido en que una instrucción controla varios elementos de procesado. Sin embargo, la diferencia clave es la organización de los vectores SIMD, en ella se exponen el ancho del vector al software, mientras que desde el punto de vista SIMT, las instrucciones especifican la ejecución y comportamiento de un único *thread*. A diferencia de las máquinas SIMD, SIMT permite al programador describir paralelismo a nivel de *thread* para *threads* independientes, así como paralelismo a nivel de datos para *threads* coordinados. El programador puede ignorar el comportamiento SIMT para el correcto funcionamiento de su código, sin embargo es un detalle muy importante para lograr un buen rendimiento.
El SM asigna un *thread* a cada SP, ejecutando cada uno el código de forma independiente basándose en sus registros de estado. El multiprocesador SIMT crea, administra y organiza la ejecución. Ejecuta los *threads* en grupos de 32 *threads* paralelos llamados *warps*. Los *threads* que componen un *warp* empiezan a ejecutar el código de forma conjunta en la misma dirección de programa, siendo libres para ejecutar sentencias condicionales.
La pregunta ahora es: ¿Cómo es dividido un *bloque* en warps? Los *threads* son divididos en grupos según sus identificadores y en orden creciente, el *thread* 0 pertenece al primer *warps*. En la figura 3.18 se muestra la división de *N bloques* en *warps*.

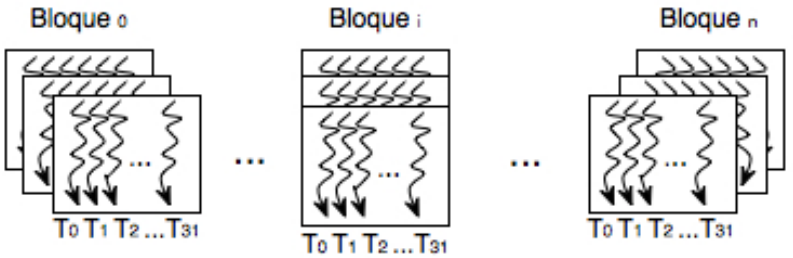


Figura 3.18. División de *Nbloques* en *warps*

La unidad SIMT selecciona un warp listo para ejecutar y prepara la siguiente instrucción para los *threads* activos del warp. Un warp ejecuta una instrucción por vez, de modo que la mayor eficiencia se obtiene cuando los 32 *threads* del warp coinciden en el mismo camino de ejecución, es decir no hay sentencias condicionales por la cual se produce una bifurcación en la ejecución. Si la ejecución de los *threads* toma diferentes caminos, el warp ejecuta en forma serializada cada una de las ramas, deshabilitando los *threads* no pertenecientes a dicha rama; cuando todas las ramas finalizan, los *threads* se reúnen en el camino común. La divergencia de saltos ocurre únicamente entre los *threads* de un warp; los warps diferentes ejecutan código de forma independiente, sin importar qué camino están ejecutando los otros. En el capítulo 5 se aborda nuevamente este tema desde el punto de vista de la performance.
Los SM pueden ejecutar varios *bloques* a la vez, el número depende de cuánta memoria compartida necesita cada *bloque*. Si los registros o la memoria compartida del SM no son suficientes para procesar como mínimo un *bloque*, el *kernel* fallará en su ejecución. Un SM puede ejecutar hasta 8 *threads* de un *bloque* en forma paralela.

3.5.1. Administración de Threads

Para poder explicar la *administración* (*scheduling*) de los *threads* es necesario hacerlo respecto de un hardware específico. En este caso se considera la implementación de hardware GT200. En dicha arquitectura, cuando un *bloque* se asigna a un SM se lo divide en grupos de 32 *threads*, warps (La cantidad de *threads* por warps depende del hardware, no es una especificación de CUDA). Un warp es la unidad de administración de *threads*. En la figura 3.18 se muestra la división de los *threads* en warps de 32 *threads* cada uno. Como se mencionó antes, los identificadores de los *threads* son consecutivos, es así que los *threads* 0 al 31 forman el primer warp, los *threads* 32 al

63 al segundo warps y así sucesivamente. El número de warps a tratar por cada SM es establecido según la cantidad de *bloques* a resolver y la cantidad de *threads* por *bloque*, aunque existen máximos definidos para cada arquitectura, en la G80 el máximo es 24 warps por cada SM, en la GT200 son 32 y en la GF100 son 48.
La administración de los *threads* en warps permite realizar diferentes optimizaciones, entre las más importantes se encuentran:

- Ejecutar eficientemente operaciones de gran latencia como son los accesos a la memoria global del dispositivo. Si una instrucción debe esperar por otra iniciada antes y con mayor latencia, el warp no es seleccionado para su ejecución, otro sin estas características será el elegido. Si existen varios warps listos para ejecutar, se selecciona uno según una prioridad establecida.
- Llevar a cabo ejecuciones más eficientes de operaciones de alta latencia como son las operaciones aritméticas de punto flotante y las instrucciones de *branch* (salto). Si hay muchos warps, es posible la existencia de alguno listo para ejecutar mientras se resuelven las operaciones más lentas.

Todas estas características son posibles porque la selección de un warps listo no implica costo, esto es conocido como *scheduling* de *threads* con *overhead* cero (costo cero). Esta optimización de la ejecución de *threads*, evitando las demoras de unos ejecutando otros es lo que en la bibliografía se conoce como ocultamiento de la latencia (*latency hiding*) (Hwang, 1993). Esta propiedad es una de las principales razones por las que la GPU no dedican áreas del chip a memoria caché y mecanismos de predicción de salto como lo hace la CPU, dichas áreas pueden ser utilizados como recursos para la ejecución de operaciones de punto flotante por ejemplo.
Conocer todas estas características de ejecución permite, a la hora de resolver un problema en la GPU, determinar el valor óptimo de *bloques* y de *threads* por *bloques* para resolver el problema en consideración, pudiendo establecer la subutilización del dispositivo por ejemplo. No siempre la máxima cantidad de *threads* por *bloques* es la cantidad óptima. En el capítulo 5 se hace referencia a los aspectos relacionados a la ejecución en warp y a tener en cuenta para obtener buena performance.

3.6. Otro ejemplo: Multiplicación de Matrices

En esta sección se analizará un ejemplo donde se tienen en cuenta todas las características explicadas en este capítulo.
Como se mencionó anteriormente, el paralelismo de datos se refiere a la propiedad de un programa de realizar muchas operaciones aritméticas en forma segura y simultánea sobre estructuras de datos. La multiplicación de matrices es un ejemplo de ello, además es una operación muy utilizada en la resolución de problemas y muy costosa de realizar en la CPU. Supone muchos accesos a memoria (la mayoría duplicados). En la figura 3.19 se puede observar gráficamente cómo se resuelve la multiplicación de dos matrices.

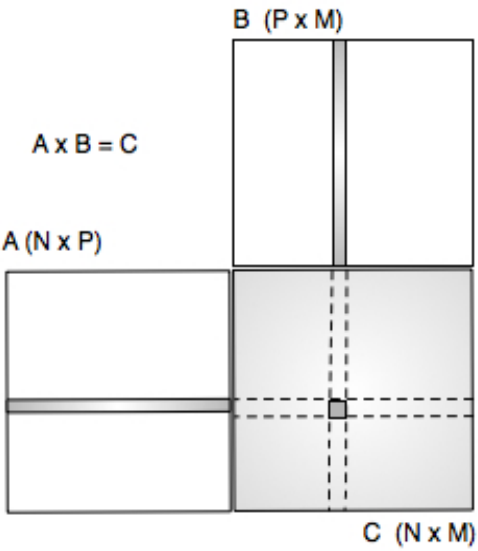


Figura 3.19. Multiplicación de matrices

La solución computacional de la multiplicación de matrices resulta un algoritmo uniforme: siempre realiza el mismo cómputo sobre distintos datos. Esta propiedad la hace un problema apto a ser resuelto según el modelo de programación SPMD y el de ejecución SIMT propios de la GPU. La solución clásica en la CPU es mostrada en la figura 3.20.

```
for ( int i =0; i<N; ++i ){
for ( int j =0; j<M; ++j ){
for ( int k=0; k<P; k++){
C[ i ] [ j ] += A[ i ] [ k ] * B[ k ] [ j ] ;
    }
  }
}
```

Figura 3.20: Implementación secuencial clásica de la multiplicación de matrices

La implementación en la GPU tomará como base el código secuencial. Para ello se debe establecer qué calculará cada *threads*. Si cada *threads* calcula un elemento de la matriz resultado, cada uno deberá leer una fila entera de la matriz *A* y la correspondiente columna de la matriz *B*, operarlas y escribir el valor resultante en la matriz *C*. Esto significa que se tendrán tantos *threads* como elementos en la matriz resultado existan. Como *A* es de dimensiones *N*P* y *B* de *P*M*, la matriz *C* tendrá dimensiones *N*M*. La estructura general de la función main() en la CPU será de la forma mostrada en la figura 3.21.

```
int main( int argc, char** argv)
{
  // Inicializar las matrices de entrada.

  // Ubicar lugar en la memoria del dispositivo y copiarlas en él

  // Configurar el ambiente de ejecución:
  //     grid(dimGrid) y bloques(dimBlock)

  // invocación del kernel

Mul_Matrix<<< dimGrid, dimBlock >>>(...arg...);

  // Liberación de la memoria del dispositivo.
}
```

Figura 3.21. Estructura general de la función main() de la CPU

Se puede observar varias etapas antes de la invocación del *kernel*: Inicializar las estructuras en la CPU, solicitar lugar para las matrices en la GPU, configurar la invocación del *kernel*, invocar el *kernel* y, finalmente, liberar la memoria en el dispositivo.
Uno de los primeros aspectos a tratar es la forma en que se almacenarán las matrices en la GPU. Para trabajar con matrices, es necesario ubicarlas en la memoria global del dispositivo en forma lineal. La forma de hacerlo es igual que la realizada en la memoria de la CPU, ubicando todos los elementos de una fila en forma consecutiva y las filas en orden consecutivo también. En la figura 3.22 se muestra la forma de ubicar una matriz de 3*5 en un arreglo lineal de 15 elementos.

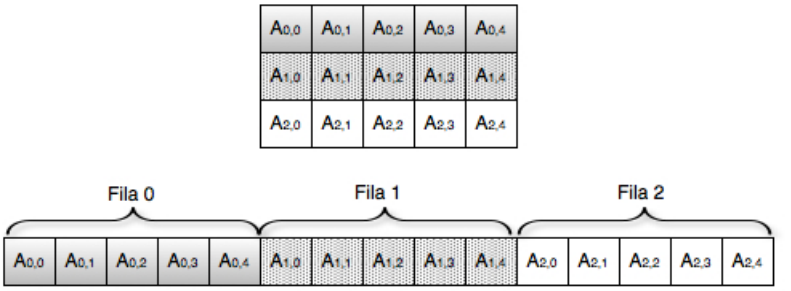


Figura 3.22. Ubicación de una matriz 2-dimensiones en un arreglo lineal

Con los elementos ubicados de esta forma, el acceso a ellos cambia, el elemento de la posición *A[i][j]* se encuentra en la posición *i*N + j*, donde *N* es la cantidad de columnas de la matriz, por ejemplo el elemento *A[2][1]* de la matriz de la figura 3.22 se encuentra en la posición 11 (2*5+1) del arreglo lineal. De esta forma se puede reformular el código de la figura 3.20 para resolver la multiplicación de matrices en *C* suponiendo que las matrices fueron almacenadas en forma lineal (Ver figura 3.23).

```
for ( int fila =0; fila<N; fila++ ){
for ( int col =0; col<M; col++ ){
for ( int k=0; k<P; k++)
C[ fila*P+ col ] += A[ fila *P+ k ] * B[ k *P+ col ] ;
  }
}
```

Figura 3.23. Multiplicación de matrices en almacenamiento lineal

Las dos primeras iteraciones dependen de la cantidad de filas de la matriz *A* y la cantidad de columnas de la matriz *B*. La iteración más interna lo hace sobre las columnas de *A* y las filas de *B*.
El código de la figura 3.24 muestra la función main() en CUDA correspondiente a la figura 3.21.

```
1.  int main( int argc, char** argv)
    {
2.  int *a,*b,*c;
3.  int *d_a,*d_b,*d_c;

4.  // Reserva memoria en el host
    y en el dispositivo
5.  a = (int )
    malloc(sizeof(int)*N*P);
6.  cudaMalloc((void**) &d_a,
    sizeof(int)*N*P);
7.  b = (int )
    malloc(sizeof(int)*P*M);
8.  cudaMalloc((void**)
    &d_b,sizeof(int)*P*M);
9.  c = (int *)
    malloc(sizeof(int)*N*M);
10. cudaMalloc((void**)
    &d_c,sizeof(int)*N*M);

11. Inicializa(a); Inicializa(b); //
    Inicializa las matrices

    // Copia al dispositivo las matrices
12. cudaMemcpy(
    d_a,a,memSize,cudaMemcpyHostToDevice);
13. cudaMemcpy(
    d_b,b,memSize,cudaMemcpyHostToDevice);

14. // Configurar el ambiente de
    ejecución:
15. // grid(dimGrid) y
    bloques(dimBlock)

    // invocación del kernel
16. Mul_Matrix<<< dimGrid,
    dimBlock >>>(...arg...);

    // Copia del dispositivo la matriz resultado
17. cudaMemcpy(
    c,d_c,memSize,cudaMemcpyDeviceToHost);
```

```
    // Libera la memoria del host y del dispositivo
18.                                 cudaFree(d_a); free(a);
19.                                 cudaFree(d_b); free(b);
20.                                 cudaFree(d_c); free(c);
21.                                 return 0;
22.                                 }
```

Figura 3.24. Función main() de CUDA para la multiplicación de matrices.

Puede observarse las distintas tareas a realizar antes y después de la invocación del *kernel*, las líneas correspondientes a cada una de las tareas son:

- Declaración de variables en la CPU y en la GPU: líneas 3 (variables: *a, b* y *c*), 4 (variables: *d_a, d_b* y *d_c*) respectivamente. En ese ejemplo se antepone "d_" para aquellas variables definidas en el dispositivo.
- Reserva de memoria para cada una de las variables en:
  - CPU: líneas 5, 7, 9.
  - GPU: líneas 6, 8, 10.

  La cantidad de memoria se corresponde con las dimensiones de las matrices detalladas en la figura 3.19.
- Inicialización de las matrices a multiplicar: línea 11. No se especifica cómo se realiza la inicialización porque no es de interés aquí. Ésta puede ser interactiva, desde archivos, etc.
- Copia de las matrices a la memoria del dispositivo: línea 12, 13.
- Copia de la matriz resultado desde la GPU a la CPU: línea 17.
- Liberación de la memoria del dispositivo y de la CPU: líneas 18, 19 y 20.

Las líneas del recuadro corresponden a la invocación y configuración del *kernel*, lo cual se va a explicar a continuación.
Considerando la resolución de la multiplicación de matrices en un *bloque* de (*N*M*) *threads*, la configuración del *kernel* y su invocación en la figura 3.23 se definen como sigue:

```
14.   dim3 dimGrid(1,1);
15.   dim3 dimBlock(N,M);

16.  Mul_Matriz <<< dimGrid, dimBlock
      >>> (d_a,d_b,d_c,P);
```

En la figura 3.25 se muestra la función *kernel* para realizar la multiplicación según la configuración planteada.

```
__global__ void Mul_Matrix(int *d_a,int *d_b,int *d_c,int p )
{
int fila = threadIdx.x;
int col = threadIdx.y;

int pvalor=0;
for(int i = 0; i < p; i++)
        pvalor+=(d_a[ fila * p + i] * d_b[ i * p + col]);

d_c[ fila* p +col]=pvalor;
}
```

Figura 3.25. Función *kernelMul_Matrix*().

Comparando la solución secuencial (Figura 3.20) y la solución en la GPU de la figura 3.25, se puede observar que ésta última tiene sólo una iteración, la más interna de la solución secuencial. Las otras dos iteraciones son sustituidas por los *threads* del *grid*: cada uno de los *threads* hace referencia a uno de los elementos de las iteraciones anidadas (Primer y segunda iteración). Las variables *fila* y *col* son instanciadas con los identificadores del *threadthread*Idx.x y *thread*Idx.y respectivamente. Los identificadores del *thread* permiten establecer qué elemento de la matriz *C* va a resolver, por ejemplo el $thread_{5,3}$ será responsable de calcular *C*[5][3], realizando el producto punto de la fila 5 con la columna 3.

Como la solución presentada utiliza un único *bloque* para resolver el problema, no es necesario referenciar a la variable blockIdx, sólo existe un único *thread* con el mismo identificador. Además, al existir un único *bloque* existe un límite para la cantidad de *threads*, éste es establecido por max*Threads*PerBlock (512 *threads* para la generación G80, 1024 para las arquitectura más modernas). Suponiendo que se trabaja con una arquitectura moderna (GF100), el límite de *thread* por *bloques* es 1024, esto significa para esta solución sólo existirán 1024 *threads* trabajando en paralelo, aplicado a matrices *N*M* significa que *N*M*=1024 elementos, o sea *C* puede ser una matriz de dimensiones (64*16), (32*32) o cualquier otras dimensiones que no superen los 1024 elementos.

Por todo lo expuesto en los párrafos anteriores, las limitaciones del código diseñado son inadmisibles para las características y bondades del hardware subyacente, eso sin considerar que la mayoría de las aplicaciones requieren multiplicaciones de matrices de millones de elementos. Es necesario redefinir la solución propuesta de manera que pueda resolver la multiplicación de matrices muy grandes y, en consecuencia, tome ventaja de las facilidades paralelas de la GPU. Una solución posible es realizar la multiplicación de matrices entre varios *bloques,* la cual se explica en los siguientes *bloques.*
Si se toma el número máximo de *threads* como referencia para establecer el tamaño de un sector de la matriz resultado a computar, es posible calcular el producto para mucho más elementos de la matriz. En la figura 3.26 se muestra gráficamente la idea de resolver el producto de matrices mediante muchos *bloques*, cada uno responsable de un sector cuadrado de la matriz resultado.

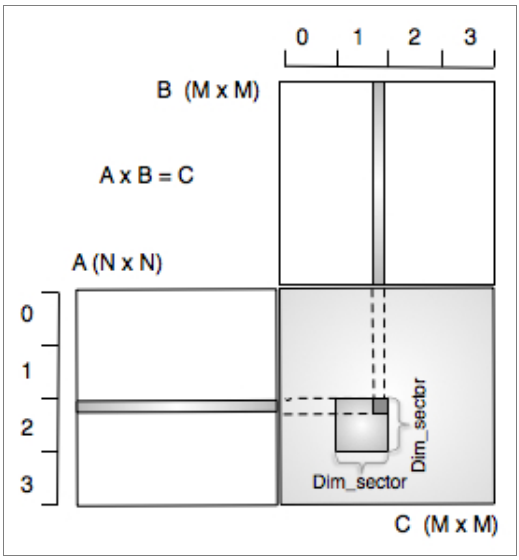


Figura 3.26. Multiplicación de matrices mediante múltiples *bloques*

Si se considera un dispositivo GF100, el número máximo de *threads* por *bloques* es 1024, lo cual implica poder calcular sectores cuadrados de la matriz de dimensión 32x32 (Por cuestiones de simplicidad se analiza la solución para matrices cuadradas).
Es necesario determinar el ancho/alto del sector (*Dim_sector*) a resolver por un *bloque*. *Dim_sector* debe ser menor o igual a 32 (recuerde el número máximo de *threads* por *bloques* para la arquitectura elegida) y

divisor de la dimensión de la matriz. La cuestión ahora es determinar cómo hacer para resolver la multiplicación de cada sector en cada *bloque*. Lo primero a considerar es la existencia de varios *threads* con los mismos identificadores. Al haber varios *bloques*, varios *threads* tendrán los mismos valores de *thread*Idx.x y *thread*Idx.y, es por ello que será necesario su diferenciación, la misma se logra utilizando la variable blockIdx, significando que la determinación del elemento dependerá de los valores de *thread*Idx y blockIdx. En la figura 3.27 se muestra un ejemplo de la división en sectores para una matriz de 6*6 considerando *Dim_sector* = 3.

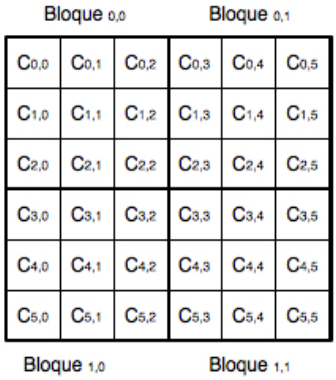


Figura 3.27. Ejemplo de la división en sectores de una matriz de 6*6

Ahora ¿Cómo determinar cada elemento en función de *thread*Idx y blockIdx? Los índice *x* e *y* del elemento $C_{x,y}$se calcula de la siguiente forma:

- *x*= (blockIdx.x * *Dim_sector* + *thread*Idx.x) e
- *y* = (blockIdx.y * *Dim_sector* + *thread*Idx.y).

Esto significa que el $thread_{threadIdx.x,\ threadIdx.y}$ del $bloque_{blockIdx.x,blockIdx.y}$usará para determinar el elemento que le corresponde la fila (blockIdx.y * *Dim_sector* + *thread*Idx.y)de *A* y la columna(blockIdx.x * *Dim_sector* + *thread*Idx.x) de la matriz *B*. Analizando el ejemplo de la figura 3.27, como el valor de *Dim_sector* es 3, existirán 4 *bloques*, cada uno responsable de calcular 9 elementos de la matriz resultado. En consecuencia cada *bloque* tendrá 9 *threads* organizados como una matriz de 3*3. Para esta configuración la tabla de la figura 3.28 muestra qué elemento calcula cada *thread* de cada *bloque* de la matriz resultado *C*.

$Bloque_{0,0}$		$Bloque_{0,1}$		$Bloque_{1,0}$		$Bloque_{1,1}$	
$thread_{0,0}$	$C_{0,0}$	$thread_{0,0}$	$C_{0,3}$	$thread_{0,0}$	$C_{3,0}$	$thread_{0,0}$	$C_{3,3}$
$thread_{0,1}$	$C_{0,1}$	$thread_{0,1}$	$C_{0,4}$	$thread_{0,1}$	$C_{3,1}$	$thread_{0,1}$	$C_{3,4}$
$thread_{0,2}$	$C_{0,2}$	$thread_{0,2}$	$C_{0,5}$	$thread_{0,2}$	$C_{3,2}$	$thread_{0,2}$	$C_{3,5}$
$thread_{1,0}$	$C_{1,0}$	$thread_{1,0}$	$C_{1,3}$	$thread_{1,0}$	$C_{4,0}$	$thread_{1,0}$	$C_{4,3}$
$thread_{1,1}$	$C_{1,1}$	$thread_{1,1}$	$C_{1,4}$	$thread_{1,1}$	$C_{4,1}$	$thread_{1,1}$	$C_{4,4}$
$thread_{1,2}$	$C_{1,2}$	$thread_{1,2}$	$C_{1,5}$	$thread_{1,2}$	$C_{4,2}$	$thread_{1,2}$	$C_{4,5}$
$thread_{2,0}$	$C_{2,0}$	$thread_{2,0}$	$C_{2,3}$	$thread_{2,0}$	$C_{5,0}$	$thread_{2,0}$	$C_{5,3}$
$thread_{2,1}$	$C_{2,1}$	$thread_{2,1}$	$C_{2,4}$	$thread_{2,1}$	$C_{5,1}$	$thread_{2,1}$	$C_{5,4}$
$thread_{2,2}$	$C_{2,2}$	$thread_{2,2}$	$C_{2,5}$	$thread_{2,2}$	$C_{5,2}$	$thread_{2,2}$	$C_{5,5}$

Figura 3.28. Elemento de la matriz *C* y *thread* del *bloque* que lo calcula

Aplicando la fórmula, usted puede analizar distintas dimensiones de matrices y hasta la validez y/o modificaciones para matrices no cuadradas. Con todo lo expuesto, se está en condiciones de mostrar la modificación de la función *kernel* de la figura 3.25. Ésta es explicitada en la figura 3.29.

```
__global__ void Mul_Matrix(int *d_a,int *d_b,int *d_c,int p )
{
int fila = (blockIdx.y * Dim_sector + threadIdx.y) ;
int col = (blockIdx.x * Dim_sector + threadIdx.x);


int pvalor=0;
for(int i = 0; i < p; i++)
      pvalor+=(d_a[fila*p + i] * d_b[i* p + col]);

d_c[fila* p +col]=pvalor;
}
```

Figura 3.29. Función *kernelMul_Matrix()* para múltiples *bloques*

El análisis realizado para determinar los elementos a calcular por cada *thread* de cada *bloque* se reflejan en el código de la figura 3.29. Ahora bien ¿cuál es el tamaño máximo de las matrices? La pregunta se contesta analizando los máximos permitidos por cada arquitectura para la cantidad de *threads* por *bloques* y la cantidad de *bloques* por *grid*. Recuerde que el máximo número de *bloques* es 65.535 por dimensión y la máxima cantidad de *threads* por *bloque* es 512 para las arquitecturas G80 y GT200, y 1024 para las GF100, esto significa

matrices de 1TB para las primeras arquitecturas y 4TB para las segundas. En caso de tener matrices más grandes, se puede dividir el cálculo en submatrices del tamaño máximo establecido para la arquitectura subyacente y resolverla en forma iterativa con la CPU.
Sólo queda establecer la configuración del *grid* y de los *bloques* a ejecutar la función *kernelMul_Matrix()*. Esta sería:

```
14.  dim3 dimGrid(N/Dim_sector, N/Dim_sector);
15.  dim3 dimBlock(Dim_sector, Dim_sector);
16. Mul_Matriz <<< dimGrid, dimBlock
    >>>(d_a,d_b,d_c,N);
```

Tenga en cuenta que *Dim_sector* debe ser divisor de *N* y que *Dim_sector * Dim_sector* no debe superar el máximo número de *threads* por *bloque* aceptados por la arquitectura del dispositivo.
La solución presentada para matrices grandes fue hecha considerando matrices cuadradas, puede usted analizar como resolver el producto de matrices de cualquier dimensión.

3.7. Resumen

La programación de la GPU ha cambiado mucho desde sus inicios, principalmente para su utilización como co-procesador en la resolución de aplicaciones de propósito general. Actualmente y ante el desarrollo de una arquitectura más general de la GPU, existen diferentes herramientas, las cuales permiten una programación genérica igual que ocurre con la CPU. Una de las herramientas más populares para la programación de propósito general es CUDA, definido por Nvidia para sus GPU a partir de la serie G80.
En este capítulo se realizó una introducción básica a la programación con CUDA, explicándose su modelo de programación, la manera en que define la arquitectura de la GPU y las características relevantes del modelo de ejecución.
CUDA es una extensión del lenguaje C, fácil de aprender, posee varios aspectos importantes de programación: variables y tipos predefinidos, variables para la administración de *threads* y de *bloques*, definición de variables y funciones en el dispositivo, función *kernel*, sincronización de los *threads* y transferencias de memoria entre la CPU y la GPU. En el apéndice A se detallan las funciones, tipos y variables básicas que ofrece.
Respecto a la ejecución se mostró el modelo de ejecución y las ventajas que ofrece respecto a la performance, los puntos a tener en

cuenta por el programador a fin de organizar la ejecución y las limitaciones propias de la arquitectura, por ejemplo número máximo de *threads* y de *bloques* por SM.
Finalmente dos ejemplos fueron desarrollados, la suma de dos vectores y la multiplicación de matrices. En ambos, el desarrollo fue incremental. En el siguiente capítulo se vuelve a considerar el ejemplo de la multiplicación de matrices para mostrar otro aspecto importante de CUDA como es la jerarquía de memoria.

3.8. Ejercicios

3.1- Dado el siguiente código:

```
__global__ void my_first_kernel(float *x)
{
int tid = threadIdx.x;

x[tid] = (float) threadIdx.x;
}

int main(int argc, char **argv)
{
float *h_x, *d_x;
int  nblocks, nthreads, nsize, n;

nblocks = 1;
nthreads = 8;
nsize  = nblocks*nthreads ;

 h_x = (float *)malloc(nsize*sizeof(float));
cudaMalloc((void **)&d_x, nsize*sizeof(float));

 my_first_kernel<<<nblocks,nthreads>>>(d_x);

cudaMemcpy(h_x,d_x,nsize*sizeof(float),cudaMemcpyDeviceT
oHost);

for (n=0; n<nsize; n++) printf(" n, x = %d %f \n",n,h_x[n]);

cudaFree(d_x);
free(h_x);
cudaThreadExit();
return 0;
}
```

Analizar:

a. ¿Dónde se ejecuta cada una de las instrucciones?
b. ¿Qué hace el programa?
c. ¿Cuántos threads y bloques se ejecutan?
d. Describa qué hace cada una de las instrucciones de CUDA incluidas en el código.

3.2- Modificar el código anterior de manera que existan:

a. Varios *bloques* con un *thread* cada uno.
b. Varios *bloques* con varios *threads* cada uno.

3.3- Implemente en la GPU la suma de vectores desarrollada en este capítulo. Considerando que la cantidad de bloques y de threads por bloques depende del tamaño de los datos de entrada.

3.4- Desarrolle un programa CUDA para sumar a una matriz un valor escalar. El tamaño de la matriz determina la cantidad de bloques y de threads por bloques. ¿Cuál sería el tamaño máximo de la matriz?

3.5- Aplique las técnicas de dividir la matriz en sectores para resolver el problema anterior. ¿Qué beneficios tendría utilizar dicha técnica para este problema?

3.6- Implemente la multiplicación de matrices explicada en la sección 3.6.

3.7- Modifique el código del ejercicio anterior de manera tal que se tengan en cuenta matrices no cuadradas.

3.8- Un histograma de un objeto (vector de enteros, imágenes, población, etc.) brinda información estadística del objeto en un espacio dado. Por ejemplo en un espacio de números enteros en el intervalo [a..b], el histograma de un conjunto C de elementos en el intervalo es un vector definido como:

$$H(C) = (h_a(C); h_{a+1}(C); \ldots ; h_{b-1}(C); h_b(C))$$

donde $h_x(C)$ es la cantidad de elementos de *C* con valor x. La $\Sigma_{i=a..b}\ h_i$ es la cardinalidad del conjunto *C*.

Realice un programa CUDA para calcular el histograma de un vector cuyos valores enteros están entre 0 y 63.
El vector es de tamaño 1024.

3.9- Modifique el programa anterior de manera que el histograma sea realizado sobre 256 valores distintos y el tamaño del vector es superior 4GB de enteros en el rango de 0 a 255.

3.10- Modifique el programa anterior para que realice el histograma de 256 sobre un vector de números reales. El tamaño del vector es indicado como parámetro de entrada del programa.

CAPÍTULO 4

Jerarquía de Memoria

La CPU se comunica con la GPU a través de la memoria, tanto al inicio de la computación enviando los datos de entrada como al final de la misma para obtener los resultados de tarea paralela. Los datos de entrada para cada uno de los *threads* están disponibles en la memoria del dispositivo, generalmente es la memoria global.
En la GPU se distinguen dos grandes tipos de memoria, la memoria *on-chip*, la cual reside en el dispositivo y, en consecuencia, tiene un rápido acceso; y la memoria *off-chip*, alojada fuera del dispositivo y con un costo de acceso mayor.
En función de los distintos tipos de memoria del dispositivo, CUDA posee una jerarquía de memoria, en la cual existen distintos tipos de memoria, cada una con sus características: tipo y costo de acceso, alcance y ciclo de vida de los objetos definidos en ella, costo de accesos y mecanismos de optimización.
En este capítulo se describen cada uno de las memorias de la jerarquía: Memoria Global, Memoria Compartida o *Shared*, Memoria Local, Memoria de Registros, Memoria de Constante y Memoria de Texturas, detallando en cada caso las propiedades antes señaladas. Conociendo estas características, el programador puede tomar decisiones respecto a qué memoria utilizar a fin de mejorar la eficiencia de la aplicación en la GPU.

4.1. Modelo de Memoria de GPU

Como se mencionó en el capítulo anterior, la tarjeta gráfica posee distintos niveles de memoria, algunos de ellos *on-chip* (dentro de la GPU) y otros *off-chip* (fuera de la GPU). Desde el punto de vista de la ejecución, las instrucciones a memoria se tratan de forma diferente según a qué nivel se esté accediendo. La latencia de una instrucción de memoria varía dependiendo del espacio de memoria al que se accede e incluso del patrón de acceso, tal y como se explica en este capítulo.

Recuerde que los cambios de contexto en la ejecución de un *threads* se producen cuando se ejecuta una instrucción de memoria, evitando así que el SM permanezca ocioso.

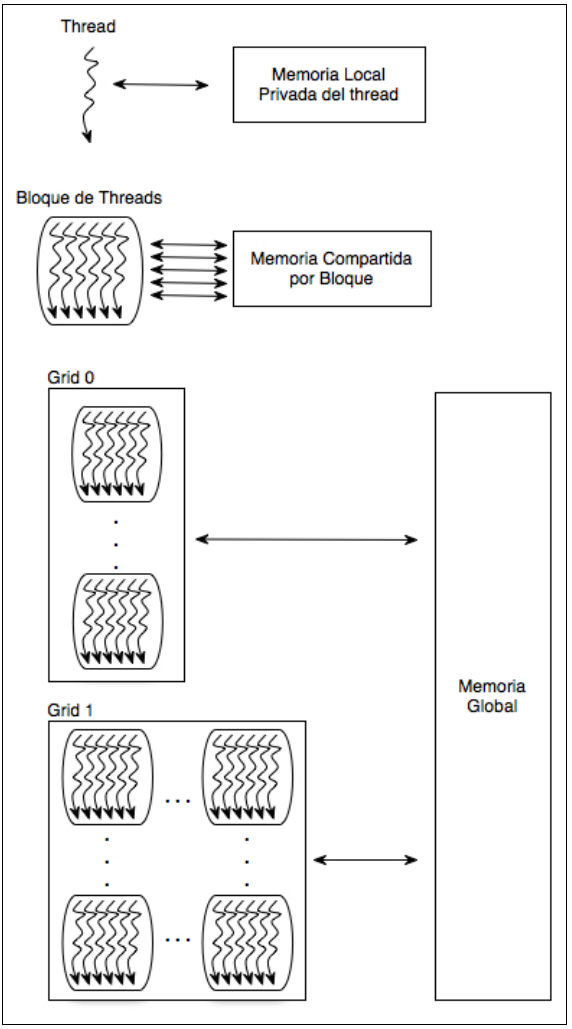


Figura 4.1. Jerarquía de memoria en CUDA

Los *threads* pueden acceder a distintos tipos de memoria de la GPU durante un *kernel*. En la figura 4.1 se muestran los distintos tipos de memoria a los que pueden acceder los *threads*, los cuales están dispuestos en *bloques* y *grid*.
En la figura 4.2. se muestra la jerarquía completa y quienes pueden acceder a cada una de ellas, las flechas indican el tipo de acceso, sólo

lectura (Unidad de procesamiento←Memoria), sólo escritura (Unidad de procesamiento→Memoria)y lectura/escritura (Unidad de procesamiento ↔Memoria).

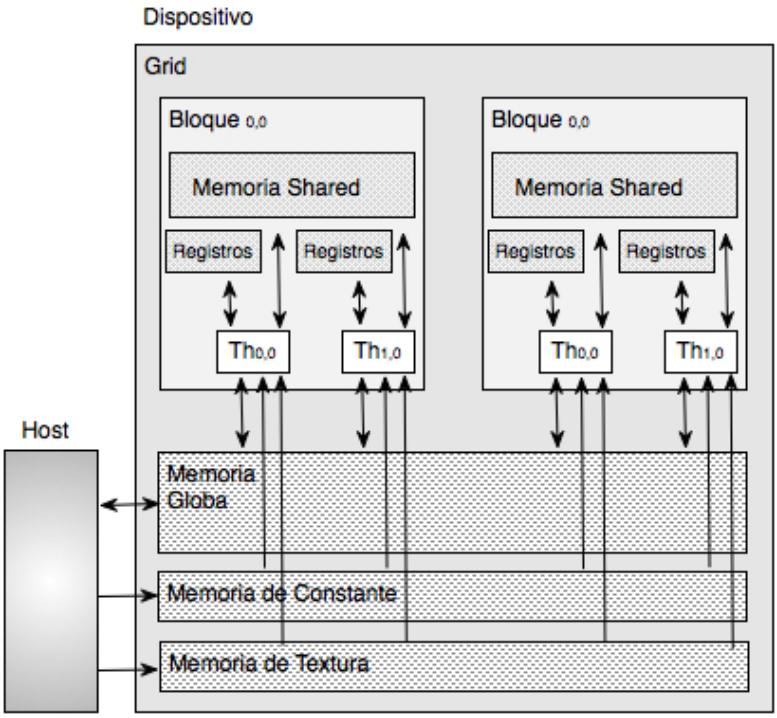


Figura 4.2. Accesos a la jerarquía de memoria

Los distintos tipos de memorias se diferencian entre ellos, no sólo por las restricciones de acceso y los tamaños, sino también por las velocidades de acceso, cada una de las memorias tiene su propio ancho de banda. Esto es un punto a tener en cuenta para lograr buena performance de la aplicación desarrollada.
Es el programador quien decide las características de las variables a utilizar en sus soluciones, estableciendo en qué memoria las aloja. Con ello, no sólo determina la velocidad y visibilidad de la variable a los participantes de la solución, sino también su alcance y tiempo de vida. Cada uno de estos conceptos se refieren a:

- El *alcance:* Establece qué *threads* acceden a la variable. Un único *thread*, todos los *threads* de un *bloque*, todos los *threads* de un *grid* o todos los *threads* de una aplicación. Dadas las características del modelo de programación SPMD, si una variable es declarada en un *kernel* como variable

privada, una instancia privada se creará para cada *thread* ejecutando el *kernel*, pudiendo acceder únicamente a su versión. Por ejemplo, si un *kernel* ejecutado por un *grid* con 1000 *bloques* de 256 *threads* cada uno, declara 4 variables privadas, 256000 instancias de cada una de las variables privadas conviven al mismo tiempo en el dispositivo.

- El *ciclo de vida*: Indica la duración de la variable en la ejecución del programa, este puede ser durante la ejecución de un *kernel* o durante toda la aplicación, es decir permanece y es accesible por todos los *kernel* involucrados en la aplicación. En el primer caso, están incluidas todas aquellas variables declaradas dentro de un *kernel*, su valor se mantiene durante la ejecución del mismo, diferentes ejecuciones de un *kernel* no implica mantener el mismo valor en dichas variables. En el segundo caso se encontrarán todas aquellas variables declaradas en el ámbito de la aplicación, no de un *kernel*, para ello los valores de estas variables se mantienen a través de los distintos *kernel* de la aplicación.

Los distintos tipos de memoria de la jerarquía son:

- *Memoria global:* Es una memoria de lectura/escritura, se localiza en la tarjeta de la GPU y la acceden todos los *threads* de una aplicación.
- *Memoria compartida o shared:* Es una memoria de lectura/escritura para los *threads* de un *bloques*, el acceso es rápido porque se encuentra dentro del circuito integrado de la GPU.
- *Memoria de registros*: Es la memoria más rápida, de lectura/escritura para los *threads*. Está dentro del circuito integrado de la GPU y sólo accesible por cada *thread*.
- *Memoria local:* Es una memoria de lectura/escritura privada de cada *thread*. Es *off-chip* y se ubica en la memoria global del dispositivo.
- *Memoria de constante*: Es una memoria *off-chip*, pero como tiene una caché *on-chip* el acceso es rápido. La acceden todos los *threads* de una aplicación sólo para realizar lecturas.
- *Memoria de texturas*: Al igual que la memoria de constante, es una memoria *off-chip* con caché. Los *threads* de toda la aplicación la acceden para leer datos escritos por el *host*.

En las próximas secciones se detallan las características de cada una de las memorias especificadas aquí.

4.2. Memoria Global

La memoria global es una memoria *off-chip*, por lo tanto su acceso es lento. Esta memoria se divide en tres espacios de memoria separada: la Memoria Global, la Memoria de Textura y la Memoria de Constantes. El primer espacio es de acceso tanto de lectura como de escritura, no así la memoria de constante y de textura, a la cual todos los *threads* de un *grid* pueden acceder sólo para lecturas. Tanto la memoria de textura y constante poseen caché *on-chip*. En esta sección se hace referencia a la memoria global, las otras se explican en las secciones siguientes.

Para declarar una variable en memoria global se antepone, opcionalmente, a la declaración de la misma la palabra clave `__device__`. Una variable en memoria global es declarada en el *host*. Todos los *threads* de todos los *bloques* del *grid* pueden acceder a la variable global, ésta puede verse como una variable compartida por todos los *threads* del *grid*. El ciclo de vida de una variable global es todo la aplicación, esto significa que una vez finalizado un *kernel* los valores de las variables globales modificadas persisten y pueden ser recuperados y accedidos por los siguientes *kernels*. Por lo tanto, las variables globales pueden utilizarse como medio de comunicación de *threads* de distintos *bloques*. Generalmente se utilizan para transmitir información entre distintos *kernels*.

El acceso a una variable global es más lento y como todos los *threads* de un *grid* pueden acceder a ella, son necesarios mecanismos de sincronización para asegurar la consistencia de los datos, una de ellas son las funciones atómicas (Apéndice A y C).

CUDA tiene limitaciones respecto al uso de variables punteros a la memoria del dispositivo. En general, se usan punteros a datos de la memoria global. Los punteros se pueden usar de dos maneras, una cuando un objeto es alojado en la memoria del dispositivo desde el *host* a través de la función `cudaMalloc()` y pasado como parámetro en la invocación del *kernel*. El segundo uso es para asignarle la dirección de una variable global para trabajar sobre ella. En la figura 4.3 se muestra el código del *kernel* de la suma de vectores detallada en el capítulo anterior. Las variables *dev_a*, *dev_b* y *dev_c* pasadas como parámetro del *kernel* son ejemplo de variables punteros a objetos residente en la memoria global del dispositivo.

```
1.    #define N 100
2.    int main( void ) {
3.    int a[N], b[N], c[N];
4.    __device__ int *dev_a, *dev_b, *dev_c;
5.    inicializa(a);    //Inicializa los vectores de entrada a y b
6.    inicializa(b);
7.    cudaMalloc( (void**)&dev_a, N * sizeof(int)); // reserva
      memoria en la GPU
8.    cudaMalloc( (void**)&dev_b, N * sizeof(int));
9.    cudaMalloc( (void**)&dev_c, N * sizeof(int));
10.   // copia los vectores A y B a la GPU
11.   cudaMemcpy( dev_a, a, N * sizeof(int),
      cudaMemcpyHostToDevice);
12.   cudaMemcpy( dev_b, b, N * sizeof(int),
      cudaMemcpyHostToDevice );
13.   Suma_vec<<<1,N>>>( dev_a, dev_b, dev_c );
14.   // copia el resultado desde la GPU en el vector C de la CPU
15.   cudaMemcpy( c, dev_c, N * sizeof(int),
      cudaMemcpyDeviceToHost);
16.   cudaFree( dev_a ); // libera la memoria reservada de la GPU
17.   cudaFree( dev_b );
18.   cudaFree( dev_c );
19.   mostrar_resultados ( C); // Muestra los resultados
20.   return 0;
21.   }
```

Figura 4.3. Variables globales en la *suma de vectores* de la GPU

Como se mencionó antes, el acceso a la memoria global es más lento, dependiendo del patrón de acceso puede variar la velocidad. Los accesos a la memoria global se resuelven en medio *warp*. Cuando un acceso a memoria global es ejecutada por un *warp*, se realizan dos peticiones: una para la primera mitad del *warp* y otra para la segunda mitad. Para aumentar la eficiencia de la memoria global, el hardware puede unificar las transacciones dependiendo del patrón de acceso. Las restricciones para la unificación depende de la arquitectura, en las más modernas GPU basta con que todos los *threads* de medio *warp* accedan al mismo segmento de memoria (las restricciones de las arquitecturas más antiguas pueden encontrarse en (NVIDIA., 2011). El patrón de acceso dentro del segmento no importa, varios *threads* pueden acceder a un dato, puede haber permutaciones, etc. Sin embargo, si los *threads* acceden a *n* segmentos distintos de memoria, entonces se producen *n* transacciones. El tamaño del segmento puede ser:

- 32 bytes si todos los *threads* acceden a palabras de 1 byte,
- 64 bytes si todos los *threads* acceden a palabras de 2 bytes,
- 128 bytes si todos los *threads* acceden a palabras de 4 bytes.

Si los *threads* no acceden a todos los datos del segmento, entonces se leen datos que no serán usados y se desperdicia ancho de banda. Por ello, el hardware facilita un sistema para acotar la cantidad de datos a traer dentro del segmento, pudiendo traer subsegmentos de 32 bytes o 64 bytes. La figura 4.4 muestra algunos ejemplos de acceso a la memoria global.

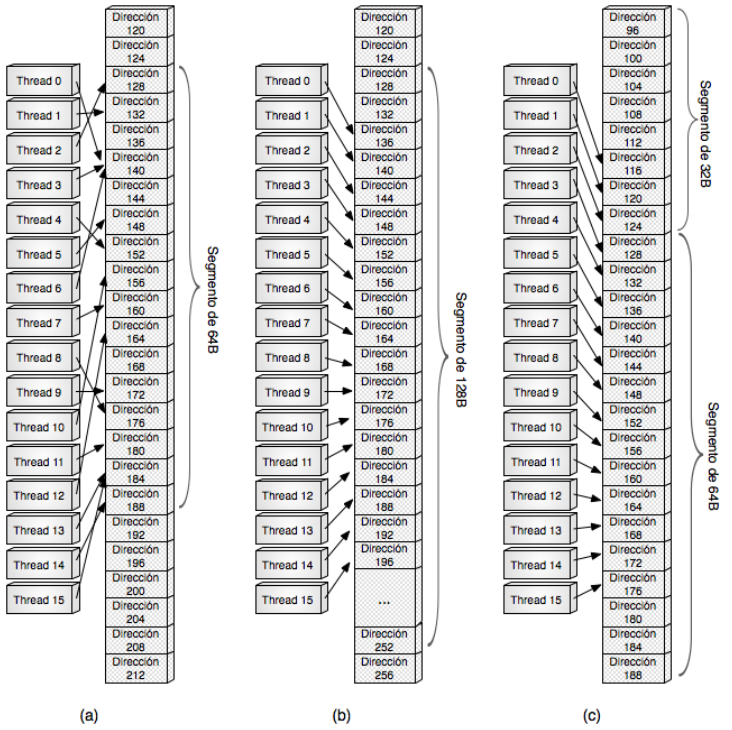


Figura 4.4. Patrones de acceso a la memoria global

En los tres casos los threads acceden a palabras de 4B, esto implica un tamaño de segmento de 128B. En el caso de la figura 4.4. (a), los threads acceden a 16 posiciones consecutivas alineadas con el segmento de 128B, por lo que el hardware reduce el tamaño a 64B para evitar leer datos inútiles. De esta forma, realiza una única transacción de 64B. En la figura 4.4. (b) ocurre todo lo contrario, también se accede a 16 posiciones, pero al no estar alineadas, el hardware no puede reducir la cantidad de datos a leer y genera una transacción de 128B. Por último, en la figura 4.4. (c), los threads

acceden a palabras alojadas en distintos segmentos de 128B implicando la generación de dos transacciones, las cuales son una de 32B y la otra de 64B.
La memoria global es una de las memorias más utilizadas, es el medio de comunicación entre el host y la GPU. Además es la memoria de mayor tamaño, su buen uso implica tener en cuenta todas las consideraciones aquí mencionadas para obtener buena performance.

4.3. Memoria Compartida

La memoria compartida o memoria *shared* recibe su nombre porque es una memoria común a todos los *threads* de un *bloque*, siendo el medio de comunicación entre ellos. Es una memoria *on-chip*, su acceso es rápido, llegando a ser en algunos casos igual de rápido como acceder a la memoria de registro o a la memoria constante.
La declaración de una variable compartida está precedida por la palabra clave __share__. Una variable compartida se declara dentro del ámbito de una función *kernel* o de una función del dispositivo. El alcance de una variable compartida está circunscripto a un *bloque*, esto significa que todos los *threads* del *bloque* ven la misma instancia de la variable compartida. Cada *bloque* tiene una instancia propia de dicha variable, lo cual hace que pueda verse como una variable privada a nivel de *bloque*.
El tiempo de vida de una variable compartida es la función donde es definida, es decir la función *kernel* o la función de dispositivo. Al finalizar la ejecución de la función que declaró la variable compartida, ésta dejará de existir.
Las variables compartidas son un medio eficiente para la comunicación de los *threads* dentro de un *bloque*, mediante ellas un *thread* puede colaborar con los demás. Como es una memoria más rápida que la memoria global, los programadores suelen utilizarla para almacenar datos de la memoria global muy utilizados por el *kernel*. Esta propiedad se va a ver en los ejemplos detallados en las siguientes secciones.
Para tener una velocidad de acceso a la memoria compartida similar a la de la memoria de registro se debe asegurar la no existencia de conflictos de accesos. El conflicto de acceso se da a nivel de *bancos*, un *banco* de memoria compartida es la unidad en la que está organizada, todos tienen el mismo tamaño (1KB) y pueden ser accedidos simultáneamente. Los *bancos* están organizados de forma que palabras sucesivas de 32-bits pertenecen a *bancos* sucesivos (pudiendo ser distintos). El ancho de banda de cada *banco* es de 32-

bits cada 3 ciclos de reloj. El número de *bancos* es de 16 para GPU de capacidad *1.x* y 32 para las de capacidad *2.x* (Ver Apéndice C).
La organización de la memoria compartida en *bancos* se realiza para obtener un mayor ancho de banda, al estar dividida en *bancos*, la memoria puede atender con éxito una lectura o una escritura en *n* direcciones pertenecientes a *nbancos* distintos. De esta manera se puede conseguir un ancho de banda *n* veces mayor a si existe un único *banco*.
Un conflicto de memoria implica dos accesos simultáneos de *threads* distintos al mismo *banco* de memoria. La resolución se hace a través de la serialización de los accesos a la memoria, siendo responsabilidad del hardware la serialización. Es este quien divide el acceso a memoria en tantos accesos libres de conflicto como sean necesarios, llevando a una pérdida del ancho de banda en un factor igual al número de peticiones libres de conflicto.
Para conseguir el máximo rendimiento es importante evitar los conflictos de *bancos*, esto implica conocer la organización de los *bancos* de memoria y comprender la forma en la cual se acomodan los datos en ellas. De esta manera se podrán programar los accesos a memoria compartida a fin de minimizar los conflictos.
Teniendo en cuenta que las arquitecturas vistas en este libro (G80, GT200 y GF100) tienen 32 *threads* por *warp*, siempre habría, al menos, conflicto de grado 2 si las peticiones se hiciesen por *warp* entero. Para evitar esto, cuando un *warp* ejecuta una instrucción sobre la memoria compartida, la petición se separa en dos peticiones: una para la primera mitad del *warp* y otra para la segunda mitad. De este modo cada uno de los 16 *threads* del medio *warp* puede acceder a un *banco* y obtener el máximo rendimiento. En la figura 4.5 se muestran algunos ejemplos de patrones de acceso sin conflictos en la memoria compartida.

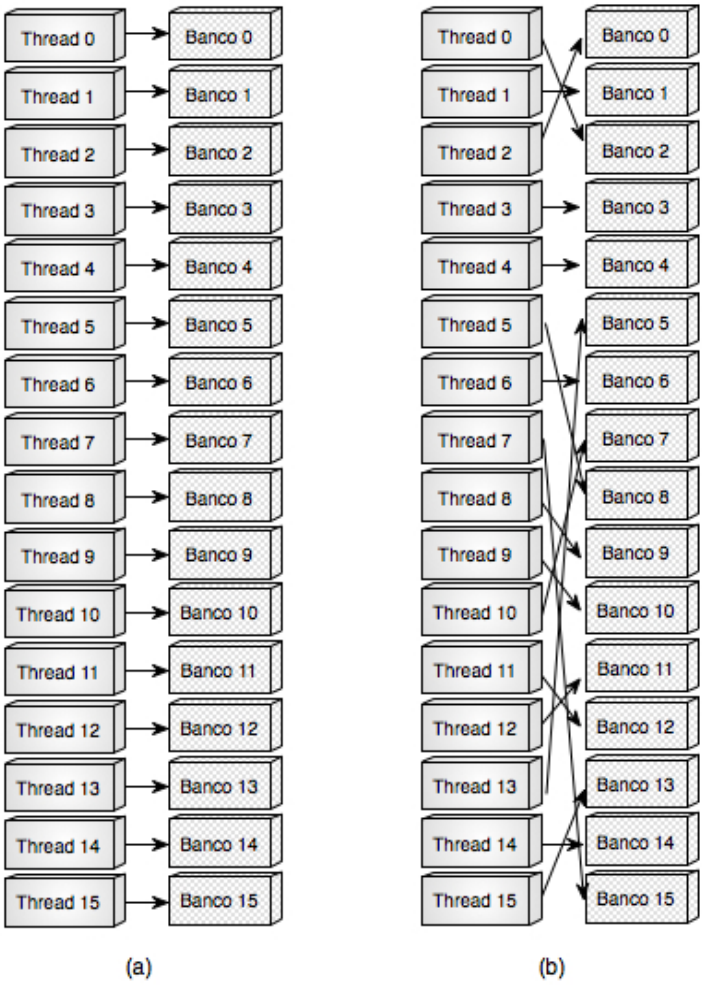


Figura 4.5. Accesos sin conflictos a la memoria compartida

En la figura 4.5, cualquiera de los dos patrones de acceso a los *bancos* de memoria no producen conflictos, cada uno accede a un *banco* diferente. La diferencia entre ambos esquemas de acceso es la forma en la que lo hacen. En (a) cada *thread* accede al *banco* de memoria en forma lineal, en cambio en (b) el acceso se hace en forma aleatoria, no existe ningún patrón de acceso.

Los accesos al mismo *banco* sin conflicto son permitidos. Esto es posible porque la memoria compartida implementa un mecanismo de distribución a través del cual una palabra puede ser leída por varios *threads* simultáneamente en la misma acción de lectura sin producir conflicto. Esto reduce el número de conflictos sobre un *banco*,

siempre y cuando todos los *threads* accedan a leer la misma posición de memoria. La figura 4.6 muestra dos casos en los cuales no existe conflicto porque los *threads* leen del mismo *banco* la misma posición.

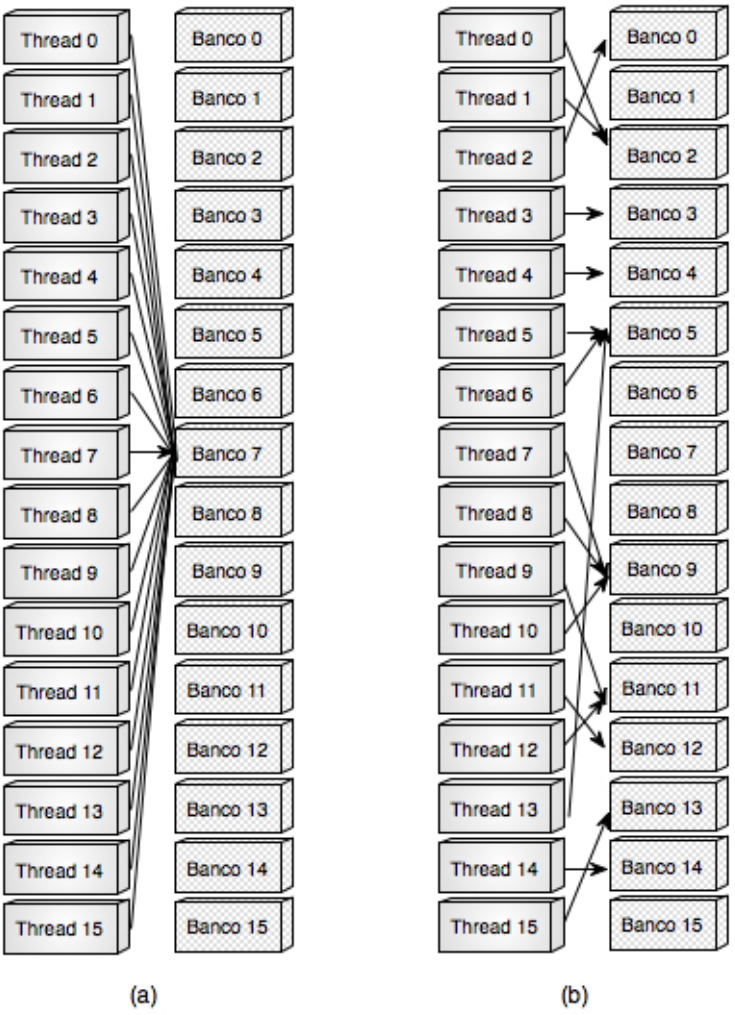


Figura 4.6. Accesos sin conflictos a la memoria compartida

En el caso (a) todos acceden a la misma posición del mismo *banco*, no así en el caso (b) donde se accede a distintos *bancos* y en el caso de existir coincidencia es a la misma posición. Si los *threads* acceden al mismo *banco* de memoria pero a diferentes posiciones, entonces existe conflicto de *banco*. Como se mencionó antes, la solución será serializar los accesos. En la figura 4.7 se muestran dos situaciones de conflicto de acceso.

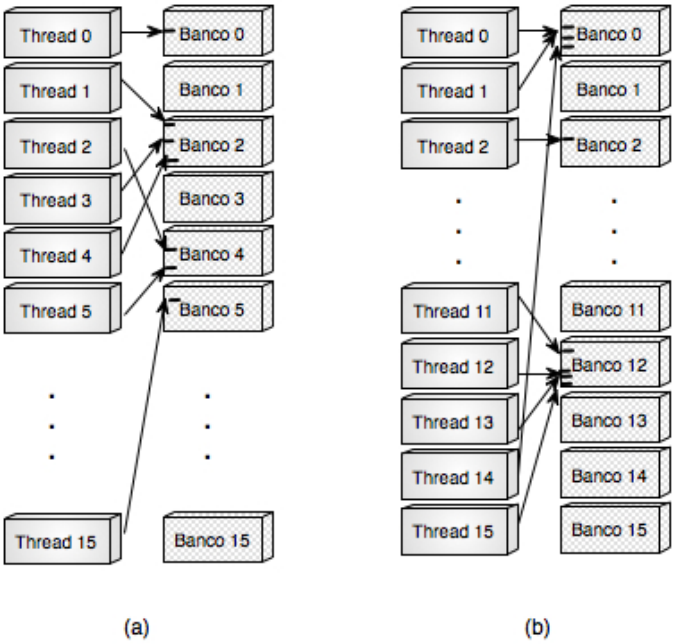


Figura 4.7. Accesos con conflicto en memoria compartida

El grado de conflicto en la figura 4.7(a) es 3, los *threads* 1, 3, 4 acceden al *banco* 2. En la parte (b) el grado de conflicto es 4 por los accesos de los *threads* 11, 12, 13 y 15.
Al ser una memoria rápida, su uso es muy bueno para reducir los costos de acceso a la memoria global. Su efectividad se logra si se tiene en cuenta el tamaño máximo por SM (16KB para GPU *1.x* y 48KB para las de capacidad *2.x*) y un patrón de acceso óptimo.

4.4. Memoria de Registros

La memoria de registro reside *on-chip*, permite accesos a muy alta velocidad y es altamente paralela. El acceso es considerado de costo cero. Los registros se asignan a los *threads* individuales, teniendo cada uno la privacidad sobre ellos. Su principal función es almacenar en ella aquellos datos, los cuales se necesitan frecuentemente y son privados a cada *thread*.
Todas aquellas variables escalares, no matrices o arreglos, declaradas dentro de un *kernel* son ubicadas automáticamente en la memoria de registros. Su alcance es dentro de cada *thread* individual y el ciclo de vida está limitado al *kernel* donde se declaró. Al invocarse el *kernel*,

una copia privada de cada una de las variables automáticas es creada para cada *thread*. En el código de la figura 4.8 la variables *tid* es una variable escalar automática, la cual se alojará en la memoria de registro, siendo privada para cada *thread*, observe que cada uno le asigna su identificador.

```
__global__ void suma_vec( int *d_a, int *d_b, int *d_c ) {
int tid = threadIdx.x; // Ident. del thread
c[tid] = a[tid] + b[tid];
}
```

Figura 4.8. Función *kernelsuma_vec*

En el caso de las variables automáticas no escalares, su almacenamiento se realiza en la memoria global, lo cual implica mayor demora en su acceso. Si bien residen en la memoria global, su tiempo de vida es el mismo que el de las de memoria de registro, sólo a un *kernel* y privado para cada *thread*. Tenga en cuenta además que la declaración de una variable no escalar automática implica una instancia privada de dicha variable para cada uno de los *threads* ejecutando en el *kernel*, esto puede significar un alto consumo de la memoria global del dispositivo.
El tamaño máximo de esta memoria depende del tipo de arquitectura del dispositivo. En la siguiente tabla se especifica el tamaño máximo de la memoria de registro para las diferentes capacidades de las arquitecturas de GPU.

Capacidad de Cómputo	Tamaño
1.0 y 1.1	8KB
1.2 y 1.3	16KB
2.x	32KB

Es importante tener en cuenta a la hora de la declaración de este tipo de variables no exceder el límite de memoria asignada a este tipo, en el capítulo 5 se describen las posibles consecuencias en la performance de la aplicación.

4.5. Memoria Local

El espacio de memoria local se encuentra dentro del espacio de memoria global y por lo tanto, es tan costoso acceder a ella como lo es

acceder a la memoria global. La memoria local se utiliza de forma automática por el compilador para alojar variables que no tienen lugar en los registros o demandan un gran número de registros a ser usados por cada *thread*. El exceso de registros se ubica en esta memoria y las variables no escalares mencionadas en la sección anterior.
Según la arquitectura, la memoria local es limitada para cada *thread*. La máxima cantidad de memoria local es de 16KB por *thread* en las arquitecturas con capacidades *1.x* y 512KB en la de capacidades *2.x*.

4.6. Memoria de Constante

La memoria constante es una memoria *off-chip*, aunque soporta baja latencia y gran ancho de banda porque tiene una memoria caché asociada *on-chip*. Los *threads* de un *grid* acceden a la memoria constante sólo para lectura, es el *host* el responsable de escribir en ella.
La declaración de una variable en la memoria constante de la GPU es hecha en el *host* y debe estar precedida por la palabra clave __constant__, opcionalmente se le puede agregar la palabra clave __device__ antes de __constant__ y se logra el mismo efecto.
El ámbito de una variable de memoria de constante es la aplicación, todos *threads* de cada uno de los *grids* de la aplicación ven la misma variable. El ciclo de vida de estas variables es toda la aplicación. Desde el *host* se escriben datos a la memoria de constante mediante la función

```
cudaMemcpyToSymbol(d_vble, vble, size);
```

donde *vble* es una variable del *host*, desde origen de los datos, *d_vble* es la variable en la memoria del dispositivo, destino de la copia, y *size* el tamaño de los datos a copiar.
El uso más frecuente de las variables de memoria de constantes es el de proporcionar valores de entrada a las funciones de un *kernel*. Las variables constantes se almacenan en la memoria global, pero se cargan en la caché para un acceso eficiente. Con los patrones de acceso adecuado, el acceso a la memoria de constante es rápido y paralelo.
El tamaño total de la memoria de constante para una aplicación se limita a 64KB para las arquitecturas con capacidades *1.x* y *2.x*, el límite de la caché para esta memoria y para todas las arquitecturas es de 8KB.

El costo de acceso a la memoria constante puede ser diferente, esto obedece al hecho de que como es un espacio con caché, el costo de acceso sería:

- Semejante al acceso a la memoria de registro, si la referencia está en la caché y dos o más *threads* del medio *warp* acceden a la misma posición de caché.
- Linealmente superior al anterior si las direcciones están en la caché y son distintas.
- Igual de costoso a una lectura en memoria global cuando se produce fallo de caché.

La memoria de constante es muy útil cuando la aplicación utiliza datos provistos por el *host* y no se modifican durante toda la aplicación, por ejemplo límites preestablecidos o datos calculados en CPU y necesarios en la GPU.

4.7. Memoria de Texturas

El espacio de memoria de textura es una memoria *off-chip* de sólo lectura para los *threads*. Una variable en la memoria de texturas tiene un ciclo de vida igual al de la aplicación y es accedida por todos los *threads* de los diferentes *grid*.

CUDA soporta un subconjunto del hardware de textura que usan las GPU. El acceso a la memoria de textura desde el *kernel* se hace mediante la invocación de funciones denominadas *texture fetches* (Sanders, 2010).

Una referencia a textura especifica qué parte de la memoria de textura es traída. Pueden existir varias referencias, las cuales pueden ser a la misma o a texturas solapadas. La referencia a textura tiene varios atributos, ellos son: la dimensionalidad (dependiendo de cómo va a ser referenciada: 1, 2 o 3 dimensiones), el tipo de los datos de entrada y salida de la textura, y la forma en que las coordenadas son interpretadas y hecho el procesamiento.

La definición de un objeto en la memoria de textura se realiza en el *host* a través de la siguiente definición:

```
texture<Type, Dim, ReadMode> texRef;
```

donde

- Type especifica el tipo de datos a ser retornado cuando se trae una textura. Puede ser de tres tipos básicos: entero, punto flotante de simple precisión o alguna de las componentes de un vector de 1, 2 y 4 componentes.

- Dim establece la dimensionalidad de la referencia a la textura, puede ser igual a 1, 2 o 3. Es un argumento opcional y por omisión es 1.
- ReadMode es igual a cudaReadModeNormalizedFloat o cudaReadModeElementType; Si es cudaReadModeNormalizedFloat y Type es un entero de 16-bit o 8-bit, el valor es retornado como punto flotante y el rango total de los enteros es instanciado en un rango de [0.0, 1.0] para entero sin signo y en [-1.0, 1.0] para enteros con signos.
  ReadMode es también un argumento opcional, por omisión es cudaReadModeElementType.

Una referencia a textura puede solamente ser declarada como una variable global estática y no puede ser pasada como parámetro en ninguna función.
A igual que la memoria de constante, tiene una memoria caché *on-chip*. Esto significa accesos lentos cuando se producen fallos de caché. Además de la caché, el costo de acceso es distinto a las otras memorias, la memoria de texturas está optimizada para aprovechar la localidad espacial de dos dimensiones, desde el punto de vista de las imágenes es muy probable que si se accede a una parte de ésta, se lea la imagen completa. Esta propiedad permite obtener mejor rendimiento cuando los *threads* de un mismo *warp* acceden a direcciones cercanas de la memoria.
Realizar las lecturas a través de la memoria de texturas puede tener algunos beneficios que la convierte en una mejor alternativa a la memoria global o a la memoria constante, ellos son:

- Si las lecturas no se ajustan a los patrones de acceso específicos para la memoria global o la memoria de constantes, es posible obtener mayor ancho de banda explotando las ventajas de localidad en la memoria de texturas.
- La latencia producida por el cálculo de direcciones se oculta mejor, pudiendo mejorar el rendimiento de las aplicaciones con acceso aleatorio a los datos.
- Los datos pueden ser distribuidos a variables separadas en una única instrucción.
- Los enteros de 8 bits y 16 bits pueden ser convertidos a punto flotante de 32 bits (en rangos [0.0, 1.0] o [-1.0, 1.0]).

Pero no todo son ventajas, para trabajar con la memoria de texturas se deben dar varias condiciones, uno de ellas es trabajar con una estructura de datos específica con direccionamiento no lineal y de cálculo no trivial. Además para realizar cualquier tipo de operaciones en ella es necesario que todos los datos están presentes en la caché

correspondiente. Estos dos factores hacen que se deba analizar muy bien el uso de la memoria de textura en la solución de cualquier aplicación.

4.8. La Memoria como límite del Paralelismo

Aunque en la jerarquía de memoria existen muchos tipo de memoria donde las velocidades de acceso son rápidas: memoria de registro, memoria compartida y memoria constante, no siempre es posible su utilización ya que son muy pequeñas comparadas con la capacidad de la memoria global. Las capacidades de memoria dependen de las características del dispositivo y el número de *threads* paralelos del *kernel*. Por ejemplo en las arquitecturas G80, cada SM tiene 8KB de registros, lo cual equivale a 128KB para el procesador entero si se cuenta con 16 SM, como es el caso de la GTX 8800. Si bien es una cantidad grande, cada *thread* puede usar una cantidad limitada de la misma. Como en la G80, en cada SM pueden residir hasta 768 *threads*, cada uno puede usar sólo 10 registros. Si los *threads* necesitan más de 10 registros, el número de *threads* ejecutando concurrentemente disminuye, reduciendo la granularidad del *bloque*, el número de *warps* a administrar y la habilidad del procesador para trabajar eficientemente con operaciones de gran latencia.
El uso de memoria compartida también puede limitar el número de *threads* asignados a cada SM. En el G80, hay 16 KB de memoria compartida para cada SM. Como la memoria compartida es usada por los *bloques* y cada SM tiene capacidad para 8 *bloques*, estos no pueden utilizar más de 2 KB por cada uno. Si utilizan más de 2KB, la cantidad de *bloques* residentes en el SM será igual al número de *bloques* resultante de sumar los requerimientos de memoria compartida y que no superen los 16KB. Por ejemplo, si cada *bloque* utiliza 5 KB de memoria compartida, a lo más 3 *bloques* pueden ser asignados a cada SM.
Recuerde que la arquitectura GT200 tiene un número máximo de *threads* por SM de 1024 y las capacidades de memoria compartida y de registros igual a 16KB cada una. En el caso de las GF100, el número máximo de *threads* por SM es 1536, y la memoria compartida asciende a 48 KB y 32KB para la de registros. En todos los casos, la cantidad máxima de *bloques* por SM es siempre 8.
En los ejemplos que se presentan a continuación, se puede determinar el factor limitante de la memoria en cada una de las aplicaciones.

4.9. Ejemplos del uso de la Jerarquía de Memorias

La diferencia en las prestaciones de las distintas memorias de la jerarquía como es la velocidad de acceso hacen posible optimizar las aplicaciones a través de su uso en las soluciones.
El programador, antes de desarrollar la solución a un problema en la GPU utilizando una memoria determinada, debe hacer un análisis exhaustivo de las características de la aplicación en función de las propiedades de las memorias existentes. Muchas veces es necesario desarrollar un análisis de costo-beneficio para tomar la mejor decisión.
La tabla de la figura 4.9 resume la jerarquía de memoria, tipo de acceso, alcance y ciclo de vida. Además muestra los modificadores necesarios para la declaración de variables en cada una de ellas.

Calificador	Memoria	Ciclo de Vida	Alcance
Variables Automáticas escalares	De Registro	*kernel*	*Thread*
Variables Automáticas arreglos	Local	*kernel*	*Thread*
__shared__	Compartida	*kernel*	*Bloque*
__constant__	De Constante	Aplicación	*Grid*
__device__ ypunteros a memoria en GPU.	Global	Aplicación	*Grid*

Figura 4.9. Jerarquía de Memoria, calificadores, ciclo de vida y alcance

En las siguientes secciones se muestran y explican dos aplicaciones donde se utiliza alguna de las memorias explicadas aquí.

4.9.1. Producto Punto

Si bien ya se habló en este libro del producto punto entre dos vectores en la sección 3.5, al explicarse la multiplicación de matrices, en esta sección se muestra una solución en GPU.
Dados dos vectores v y z, donde $v = (v_0, v_1, v_2, \ldots, v_N)$ y $z = (z_0, z_1, z_2, \ldots, z_N)$, el producto punto entre v y z se define como

$$v \bullet z = (v_0, v_1, v_2, \ldots, v_N) \bullet (z_0, z_1, z_2, \ldots, z_N) = (v_0 * z_0) + (v_1 * z_1) + (v_2 * z_2) + \ldots + (v_N * z_N)$$

La solución computacional secuencial es mostrada en la figura 4.10.

```
int r=0;
for ( int k=0; k<N; k++)
    r += v [ k ] * z[ k ] ;
return (r);
```

Figura 4.10. Implementación clásica del producto punto

Se puede observar una dependencia entre todos los elementos para obtener el resultado final, si bien el producto de cada uno de los componentes es independiente, la suma requiere un tratamiento cuidadoso. ¿Cómo se puede resolver el producto punto en la GPU? Lo primero a determinar es cómo se puede resolver el problema aplicando el paradigma de paralelismo de datos. Una posible solución es realizar el producto de las correspondientes componentes en paralelo, para luego resolver la suma en una manera ordenada y controlada. En la figura 4.11 se muestra el código de la función main().

Al igual que en los ejemplos del capítulo anterior, la función main() implica diferentes tareas, las cuales son: Declaración de las variables y asignación de la memoria en la CPU (líneas 2 y 6-8) y en la GPU (líneas 3 y 11-13); inicialización de los vectores (línea 16); copia de los vectores a multiplicar en la GPU (líneas 19-20); invocación del *kernel*dot(...); obtención del resultado final (línea 25) y liberación de memoria en la GPU (líneas 28-30) y en la GPU (línea 33). Las variables *d_v*, *d_z* y *d_partial_r* son variables punteros a variables en la memoria global del dispositivo. Una vez leído o generado cada uno de los vectores de entrada, estos se copian en las posiciones en la memoria global indicadas por dichos punteros (Líneas 19 y 20).

La solución paralela formulada para el cálculo del *producto punto* entre los vectores *v* y *z* propone dividir el producto en distintos *bloques* cuyos *threads* calcularán el producto de dos componentes y entre todos realizarán la suma parcial de sus cómputos. Como todos los productos deben ser sumados en una única variable, una reducción por suma es necesaria. En la figura 4.12 se muestra el código de la función *kernel*dot().

```
1.  int main( void ) {
2.  float  *v, *z, r, *partial_r;
3.  float  *d_v, *d_z, *d_partial_r;
4.
5.  // Asigna memoria en CPU
6.  v = (float*)malloc( N*sizeof(float) );
7.  z = (float*)malloc( N*sizeof(float) );
8.  partial_r = (float*)malloc( bloquePorGrid*sizeof(float) );
9.
10. // Asigna memoria en GPU
11. cudaMalloc( (void**)&d_v, N*sizeof(float));
12. cudaMalloc( (void**)&d_z, N*sizeof(float));
13. cudaMalloc( (void**)&d_partial_r, bloquePorGrid*sizeof(float)) ;
14.
15. // Inicializa los vectores
16. inicializa(v); inicializa(z);
17.
18. // copia los vectores a la GPU
19. cudaMemcpy( d_v, a, N*sizeof(float),
    cudaMemcpyHostToDevice ) ;
20. cudaMemcpy( d_z, b, N*sizeof(float),
    cudaMemcpyHostToDevice );
21.
22. dot<<<bloquePorGrid,threadsPorBloque>>>( d_v, d_z,
    d_partial_r );
23.
24. // Obtener resultado final.
25. resultado_final(...);
26.
27. // Libera la memoria global de la GPU
28. cudaFree( d_v ) ;
29. cudaFree( d_z ) ;
30. cudaFree( d_partial_r ) ;
31.
32. // Libera la memoria de la CPU
33. free( v ); free( z ); free( partial_r );
34. }
```

Figura 4.11. Función main() para el *producto punto* de vectores en la GPU

```
1.  __global__ void dot( float *v, float *z, float *r ) {
2.  __shared__ float cache[threadsPorBloques];
3.   int tid = threadIdx.x + blockIdx.x * blockDim.x;
4.  int cacheIndex = threadIdx.x;
5.   int i;

6.   cache[threadIdx.x] = v[tid] * z[tid];

7.  __syncthreads();  // Sincroniza los threads del bloque

8.   // En este caso blockDim.x debe ser potencia de 2.

9.   i = blockDim.x>>1;

10.  while (i != 0) {
11.    if (cacheIndex < i)
12.      cache[cachcIndex] += cache[cacheIndex + i];
13.  __syncthreads();
14.    i=i >>1 ;
15.  }
16.  if (cacheIndex == 0)
17.    r[blockIdx.x] = cache[0];
18.  }
```

Figura 4.12. Función *kernel*dot()

Puede observarse la declaración de una variable compartida (__*shared*__ float cache...), la cual será utilizada para ir computando la suma parcial de cada *bloque*. La variable compartida *cache* es inicializada por los *threads*, cada uno asigna en su posición el producto de sus componentes (línea 7). Como es necesario asegurar que todos los *threads* hayan asignado el producto en *cache*, se realiza la sincronización (línea 9). Una vez inicializada la memoria compartida, se procede a realizar una reducción sobre *caché*. En la línea 13 se inicializa la variable automática *i*, el valor asignado es la mitad de la cantidad de *threads* en el *bloque*. Observe que en lugar de dividir por 2 se hace una operación de shift (>>), ésta es más económica de realizar en la GPU que la operación de división por 2 (NVIDIA, NVIDIA CUDA C Programming Guide. Versión 4.0., 2011). Sólo trabajan la mitad de los *threads* (Línea 16) y cada uno suma dos elementos, uno de la primer mitad y el otro de la segunda (Línea 17). Una vez realizada la suma, se sincronizan todos los *threads*, *i* es actualizada a su próximo valor en la iteración, repitiendo el proceso hasta quellegue a 0. Finalmente, línea 21-22, el valor de *cache[0]* es copiado en la variable

global *r* en la posición correspondiente al identificador del *bloque*. En la figura 4.13 se muestra gráficamente el procesado de reducción implementado en el *kernel*dot(), se considera para el ejemplo de la figura un número de *threads* igual a 8.

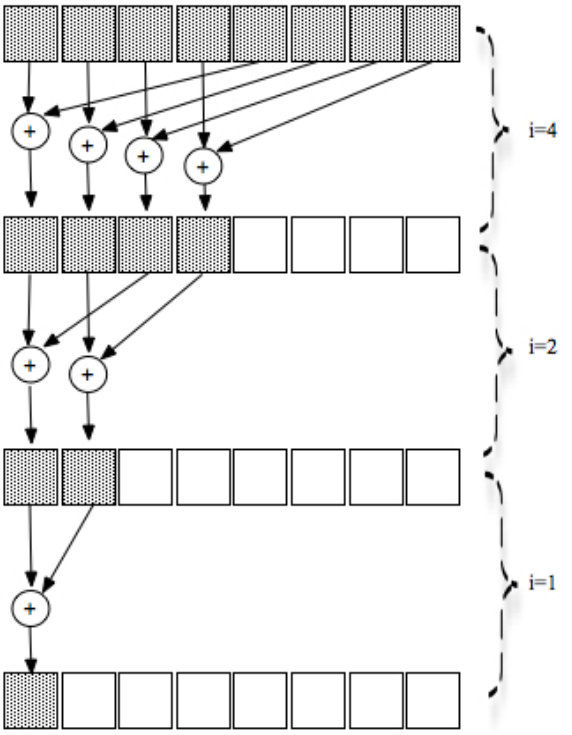


Figura 4.13. Tres pasos de la reducción por suma

Hasta el momento no se habló de cuántos *bloques* y cuántos *threads* por *bloques* se ejecutarán al ser invocado el *kernel*. La determinación merece un análisis más detallado. Para establecer cuántos *threads* por *bloque* es necesario analizar respecto a la posibilidad de declarar la variable en memoria compartida. Suponga que está ejecutando en un GPU G80 o GT200 donde la memoria compartida tiene 16KB. Si se declaran 512 *threads* por *bloque*, la variable *cache* ocupa 512 * sizeof(float) = 2 KB, implicando 8 bloques por SM, lo cual es posible. Muchas veces es necesario realizar un análisis más profundo ya que no siempre la máxima cantidad de *threads* es el valor más adecuado, puede que con 256 *threads* por ejemplo trabaje mejor. De esta manera, la configuración del ambiente de ejecución del *kernel*dot() será

```
bloquePorGrid= N/512;
threadsPorBloque= 512;
```

donde*N* es el tamaño de los vectores.
Hasta aquí se ha calculado un vector, *parcial_r*, el cual contiene las sumas parciales de cada uno de los *bloques*, falta la combinación final de los resultados parciales. La misma puede ser hecha de dos maneras, una en la CPU, en forma secuencial, siendo necesaria la copia de las sumas parciales desde la memoria global del dispositivo a la CPU. En el código siguiente se completa el código de la figura 4.11, desarrollándose la combinación de los resultados parciales en la CPU. Para ello es necesario copiar desde la GPU las sumas parciales.

```
cudaMemcpy(parcial_r,d_parcial_r,
bloquesPorGrid*sizeof(float),cudaMemcpyDeviceToHost);
c = 0;
for (int i=0; i<bloquesPorGrid; i++) {
c += parcial_c[i];
 }
```

La otra opción es través de un *kernel*, el cual sería similar al código de la figura 4.12 para la parte de la reducción por suma, pero sobre la variable *r* en memoria global. En este caso se deberá realizar un análisis de la cantidad de *bloques* que van a intervenir en la reducción, recuerde que la sincronización de *threads* es dentro de un *bloque*. Puede analizar resolver de la misma forma y en el mismo *kernel* de la figura 4.10 (*Nota*: Considere que los *threads 0* de cada bloque intervienen en la reducción por suma en la memoria global).
En el caso del producto punto, la memoria compartida no es un factor limitante. Como cada *bloque* requiere de 2KB de almacenamiento, entonces 8 *bloques* pueden residir en un SM, el máximo permitido. En este caso, el límite lo impone el número de *threads* máximo por SM. Como cada *bloque* se definió de 512 *threads*, implica que sólo un *bloque* puede residir en un SM. Si se reduce a 256 el número de *threads* por *bloque*, los *bloques* por SM aumentan a 3. Estos números son válidos para la G80.

4.9.2. Multiplicación de Matrices usando Memoria Compartida

Dadas las características enunciadas para las memorias, la memoria global es grande pero muy lenta, no así la memoria compartida, la cual es muy rápida pero pequeña. Una estrategia común en los

desarrolladores de programas CUDA es utilizar ambas dividiendo los datos en subconjuntos denominados *tile* (sector, fragmento, azulejo o baldosa), de manera tal que cada una de estas porciones entre en la memoria compartida. Un criterio para determinar cómo dividir en *sectores* es asegurar que los cálculos en cada uno son independientes siempre y cuando la estructura de datos lo permita.
El concepto de *sector* se puede mostrar con el ejemplo de la multiplicación de matrices. La Figura 4.14 muestra el código del *kernel* que resuelve la multiplicación de matrices desarrollada en el capítulo anterior.

```
1.  __global__ void Mul_Matrix(int *d_a,int *d_b,int *d_c,int
    p )
2.  {
3.   int fila = (blockIdx.y * Dim_sector + threadIdx.y) ;
4.   int col = (blockIdx.x * Dim_sector + threadIdx.x);
5.
6.  int pvalor=0;
7.   for(int i = 0; i < p; i++)
8.         pvalor+=(d_a[fila*p + i] * d_b[i* p + col]);
9.
10. d_c[fila* p +col]=pvalor;
11. }
```

Figura 4.14. Función *kernel*Mul_Matrix() para múltiples *bloques*

En la figura 4.15 se muestra gráficamente la multiplicación de matrices para 4 *bloques* con 9 *threads* cada uno (3*3). Se puede ver que el $Bloque_{0,0}$ calcula los elementos $C_{0,0}$, $C_{0,1}$, $C_{0,2}$, $C_{0,0}$, $C_{1,0}$, $C_{1,1}$, $C_{1,2}$, $C_{2,0}$, $C_{2,1}$y $C_{2,2}$, observándose gráficamente qué filas y columnas intervienen en cada cálculo.

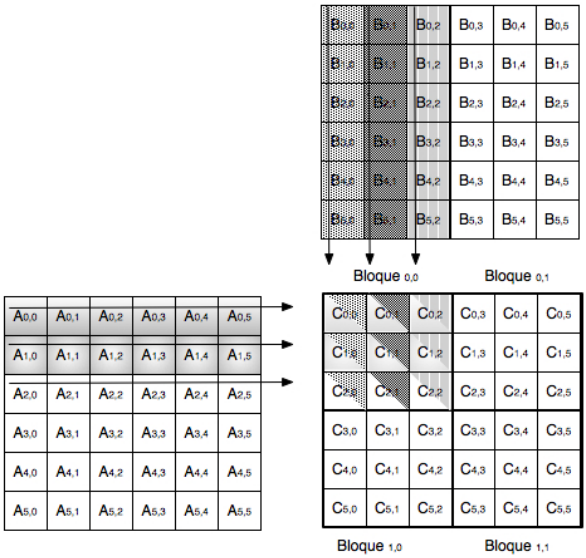


Figura 4.15. Multiplicación de matrices con varios *bloques*

La figura 4.16 detalla los accesos a la memoria global hechos por los *threads* del $Bloque_{0,0}$. Cada *thread* tiene acceso a 6 elementos de *A* y a 6 de *B*, superponiéndose en el acceso a algunas componentes, por ejemplo los $threadsT_{0,0}$, $T_{0,1}$ y $T_{0,2}$acceden a la posición $A_{0,2}$ y a todos los elementos de la fila cero de *A*. Los $threadsT_{0,2}$, $T_{1,2}$ y $T_{2,2}$ acceden a $B_{0,2}$ y a todos los elementos de la columna 2 de *B*.

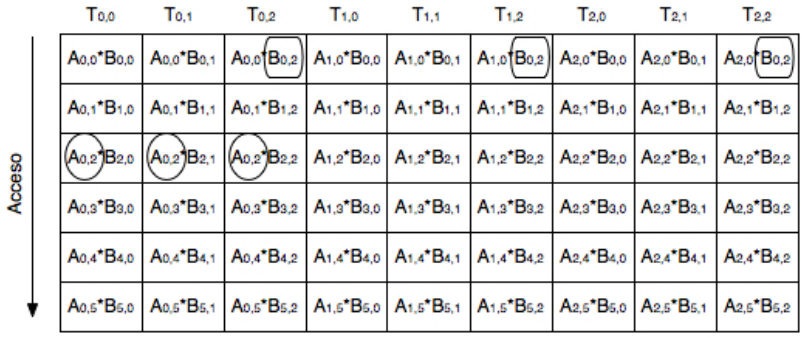


Figura 4.16. Acceso a la memoria global por los *threads* del $bloque_{0,0}$

Una mejora podría ser hecha organizando los accesos de los *threads* a la memoria global de manera que exista la menor cantidad de superposiciones y en consecuencia reducirlos.
Cada elemento de *A* y *B* se accede tres veces en el ámbito del $Bloque_{0,0}$, si los nueve *thread* colaboran entre sí, pueden reducir el

acceso a la memoria global y reducir significativamente los accesos a ella, siendo esta reducción proporcional a la dimensión de la matriz.
La idea es que los *threads* en forma colaborativa lean los elementos de *A* y *B* y los copien a la memoria compartida antes de usarlos en el producto punto. A fin de no sobrepasar el tamaño de la memoria compartida, se divide a las matrices *A* y *B* en *tiles* o *sectores* cuyo tamaño se ajuste a ella. En la figura 4.17 se muestra como se divide en sectores las matrices *A* y *B* para la colaboración de los *threads*. Los *tiles* son matrices de (3*3) indicadas con líneas más gruesas.

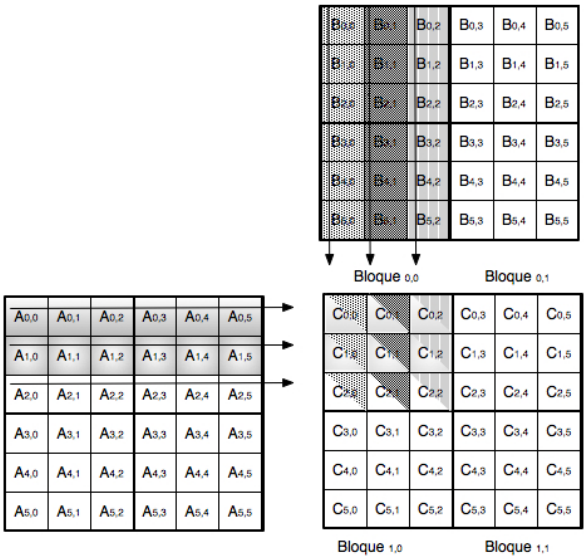


Figura 4.17. División en *sectores* de *A* y *B*

El código de la función *kernel* para la multiplicación de matrices es detallado en la figura 4.18. Las variables en la memoria compartida tienen el tamaño del *sector* y en cada fase se utilizan para cargar los valores desde las matrices en la memoria global. En la figura 4.19 se muestran las lecturas realizadas en la memoria global en la primer y segunda iteración para el ejemplo de la figura 4.17. En ella se especifica también la posición de la memoria compartida donde se guardan los valores leídos.

```
1.   __global__ void MatMul_Shared(Matrix A, Matrix B, Matrix C)
2.   {
3.     __shared__ float As[Dim_sector][Dim_sector];
4.     __shared__ float Bs[Dim_sector][Dim_sector];
5.
6.   int bx = blockIdx.y;
7.     int by = blockIdx.x;
8.     int fila = by * Dim_sector+threadIdx.y;
9.     int col = bx * Dim_sector + threadIdx.x;
10.  float Cvalor = 0;
11.
12.    for (int m = 0; m < (N/ Dim_sector); m++) {
13.
14.      As[threadIdx.y][threadIdx.x] = A[fila*N+m*Dim_sector+threadIdx.x];
15.      Bs[threadIdx.y][threadIdx.x] = B[(m*Dim_sector+threadIdx.y)*N + col];
16.
17.  __syncthreads();
18.
19.      for (int k = 0; k < Dim_sector; k++)
20.        Cvalor += As[fila][k] * Bs[k][col];
21.
22.  __syncthreads();
23.    }
24.
25.    C[fila*N+col]= Cvalor;
26.  }
```

Figura 4.18. Función *kernel* de la multiplicación de matrices usando memoria compartida

Thread	Iteración 0		Iteración 1	
$T_{0,0}$	$A_{0,0} \rightarrow As_{0,0}$	$B_{0,0} \rightarrow Bs_{0,0}$	$A_{0,3} \rightarrow As_{0,0}$	$B_{0,0} \rightarrow Bs_{0,0}$
$T_{0,1}$	$A_{0,1} \rightarrow As_{0,1}$	$B_{0,1} \rightarrow Bs_{0,1}$	$A_{0,4} \rightarrow As_{0,1}$	$B_{0,1} \rightarrow Bs_{0,1}$
$T_{0,2}$	$A_{0,2} \rightarrow As_{0,2}$	$B_{0,2} \rightarrow Bs_{0,2}$	$A_{0,5} \rightarrow As_{0,2}$	$B_{0,2} \rightarrow Bs_{0,2}$
$T_{1,0}$	$A_{1,0} \rightarrow As_{1,0}$	$B_{1,0} \rightarrow Bs_{1,0}$	$A_{1,3} \rightarrow As_{1,0}$	$B_{1,0} \rightarrow Bs_{1,0}$
$T_{1,1}$	$A_{1,1} \rightarrow As_{1,1}$	$B_{1,1} \rightarrow Bs_{1,1}$	$A_{1,4} \rightarrow As_{1,1}$	$B_{1,1} \rightarrow Bs_{1,1}$
$T_{1,2}$	$A_{1,2} \rightarrow As_{1,2}$	$B_{1,2} \rightarrow Bs_{1,2}$	$A_{1,5} \rightarrow As_{1,2}$	$B_{1,2} \rightarrow Bs_{1,2}$
$T_{2,0}$	$A_{2,0} \rightarrow As_{2,0}$	$B_{2,0} \rightarrow Bs_{2,0}$	$A_{2,3} \rightarrow As_{2,0}$	$B_{2,0} \rightarrow Bs_{2,0}$
$T_{2,1}$	$A_{2,1} \rightarrow As_{2,1}$	$B_{2,1} \rightarrow Bs_{2,1}$	$A_{2,4} \rightarrow As_{2,1}$	$B_{2,1} \rightarrow Bs_{2,1}$
$T_{2,2}$	$A_{2,2} \rightarrow As_{2,2}$	$B_{2,2} \rightarrow Bs_{2,2}$	$A_{2,5} \rightarrow As_{2,2}$	$B_{2,2} \rightarrow Bs_{2,2}$

Figura 4.19. Accesos a la jerarquía de memoria para la multiplicación de matrices

Observe que dos sincronizaciones son necesarias, la primera asegura que todo el *sector* de ambas matrices se ha cargado en la memoria compartida y el segundo es para asegurar que la suma parcial se complete antes de pasar a la siguiente fase o finalizar el producto punto.
La solución de la multiplicación de matrices utilizando la memoria compartida presenta muchas ventajas, una de ellas es la reducción del acceso a la memoria global en función de la dimensión del *sector*. Además acelera la solución, los accesos a los mismos elementos son hechos en la memoria compartida, más rápida que la global, y de lectura. Se deberá realizar un análisis cuidadoso para determinar la no existencia de conflictos de accesos.
El análisis de la memoria como factor determinante del paralelismo tiene las mismas características que el hecho para el ejemplo anterior. Para *tiles* de 16*16, cada *bloque* requiere 1KB para almacenar cada una de las matrices en la memoria compartida, esto significa que cada *bloque* necesita 2KB de memoria compartida. De la misma forma se puede determinar la cantidad de registros.

4.10. Resumen

Un *thread* en CUDA puede acceder a los distintos tipos de memoria de la jerarquía. Dependiendo de a cuál acceda será la velocidad de acceso. Se distinguen dos grandes tipos de memoria, las memorias *on-chip*: Memoria Compartida y de Registros, y las *off-chip*: Memoria Global, Local, de Constantes y de Texturas. Respecto a las velocidades de acceso, las memorias del primer grupo tienen accesos más rápidos que las del segundo a excepción de la de Constante y de Textura, las cuales al tener caché en el *chip*, tienen velocidades de acceso similares a la memoria de registros y compartida. Estás dos son memorias de sólo lectura para los *threads* de la aplicación, la escritura la realiza el *host*.
Otra de las diferencias entre las distintas memorias es su tamaño, la memoria global es la más grande y el resto, además de ser mucho más pequeñas, está limitado según la arquitectura. Los programadores deben tener en cuenta los límites, su exceso implica incorporar un factor limitante para el número de *threads* o de bloques en un SM, en el apéndice C se describen los máximos para cada una de las arquitecturas y sus capacidades.
Se presentaron distintas técnicas para reducir la latencia de acceso a la memoria global y evitar los conflictos de bancos de la memoria

compartida. En ambos casos se debe tener en cuenta la organización y planificación de los accesos.
Tener en cuenta los distintos tipos de memoria, le permiten al programador rediseñar sus aplicaciones de manera de minimizar los accesos a la memoria global y utilizar alguna de las memorias más rápidas. Un ejemplo de ello es la multiplicación de matrices desarrollada.
En este capítulo se presentaron los conceptos básicos y las características de cada memoria, el aprovechamiento de las propiedades más avanzadas de cada una implicará un estudio más profundo del presentado en este texto.

4.11. Ejercicios

4.1. Desarrolle la multiplicación de matrices explicada en la sección 4.8, donde se utiliza la memoria compartida.

4.2. Analice el programa del histograma desarrollado en el ejercicio 3.8 del capítulo anterior y proponga una reducción de los accesos a memoria global utilizando otras memorias de la jerarquía.

4.3. Proponga un programa en CUDA para resolver el histograma sobre 256 valores utilizando memoria compartida.

4.4. Resuelva la multiplicación por reducción de los elementos de un vector reduciendo los accesos a memoria global.

4.5. Para la solución propuesta en el ejercicio 3.10 analice:
   a. Los accesos a memoria global y a la memoria compartida.
   b. Cantidad de:
      1. Memoria de registro.
      2. Memoria Global.
      3. Memoria Compartida.

4.6. Dado el siguiente código:
En el main

```
const BANDAS=256;
const N= 16384; //16KB

int bands[BANDAS+1];
```

```
int dato [N];
__constant__ int band_GPU[BANDAS+1];
…
bands[0]=0;
for (int i = 1; i < BANDAS; i++)
bands[i]=bands[i-1] +(rand()%(N-bands[i-1]));
bands[i]=N;
cudaMemcpyToSymbol(band_GPU,bands,(sizeof
(int) * (BANDAS+1)));
…
kernel<<<1,BANDAS>>>(dato,d_r);
…
```

En el *kernel*

```
 __shared__ int sum[blockDim.x];
int i;

sum[threadIdx.x]=0;
size=(band_GPU[threadIdx.x]-
band_GPU[threadIdx.x-1]);

for(i=band_GPU[threadIdx.x-
1];i<band_GPU[threadIdx.x]+1;i++)
sum[threadIdx.x]+=dato[i];

 __syncthreads();
if (threadIdx.x==0){
for (i=0, d_r=0;i<blockDim.x;i++)
   d_r+=sum[threadIdx.x];
}
```

Analice:

a. ¿Qué hace la parte del programa enunciado?
b. Tipos de memoria que utiliza.
c. Tamaño solicitado para cada una de las memorias.
d. Complete el código de programa con las sentencias faltantes.

CAPÍTULO 5

Análisis de Rendimiento y Optimizaciones

Por las propiedades de la GPU, es de esperar que las aplicaciones desarrolladas en ella obtengan buena performance. No siempre es así, las GPU poseen sus limitaciones, las cuales dependen de las características de la arquitectura y, si son desconocidas por el programador, pueden conducir a aplicaciones con bajo rendimiento.
En los capítulos anteriores cada vez que se introdujo un concepto, se hizo referencia no sólo a sus características, sino también a las posibles optimizaciones para lograr un buen desempeño.
En este capítulo se analizan distintos factores limitantes del desempeño de una aplicación y para los cuales existe una solución, pudiendo ser una solución diferente y más eficiente de la aplicación. Los factores considerados son: respecto a la ejecución de los *threads*, a la memoria global: organización de los accesos y técnicas de *prefetching* de datos, al rendimiento de las instrucciones: mezcla y granularidad, y, finalmente, la asignación de recursos en un SM. El objetivo de este capítulo es mostrar al programador las posibles consecuencias de sus elecciones y tratar que, al menos intuitivamente, desarrolle las aplicaciones tratando de evitar las limitaciones.

5.1. Ejecución de *Threads*

Al ejecutar un kernel en la GPU, como se mencionó en capítulos anteriores, se genera un *grid* de una o dos dimensiones de *bloques*, cada uno con una, dos o tres dimensiones de *threads*. La ejecución de los *threads* se realiza en algún orden, asumiéndose para ello que las ejecuciones son independientes entre sí. La utilización de mecanismos de sincronización entre *threads* puede determinar una pérdida de performance de la aplicación.
La ejecución de los *threads* de un *bloque* se realiza a través de los *warps*, permitiendo reducir costos y realizar algunas optimizaciones como la administración de los accesos a memoria. En todas las

implementaciones de GPU existentes, el tamaño de los *warps* es de 32 *threads*, es de esperar que este valor cambie con las nuevas generaciones de GPU.
Los *threads* son asignados a los distintos *warps* de un *bloque* según su identificador, si el *bloque* es de una dimensión la asignación es directa, según threadIdx.x. Si el *bloque* es bi-dimensional, la asignación se debe linealizar, esto significa que primero se asignaran al *warp* los *threads* con igual identificador de threadIdx.y y en orden ascendente por threadIdx.x. Lo mismo se aplica a los *bloques* de tres dimensiones pero considerando en primer lugar al identificador threadIdx.z, luego se procede de la misma forma que ocurre para los bloques de dos dimensiones. En la figura 5.1 se muestra la división de los *threads* en *warps* dependiendo de la dimensionalidad del *bloque*: (a) *bloque* unidimensional, (b) *bloque* bidimensional y (c) *bloque* tridimensional.

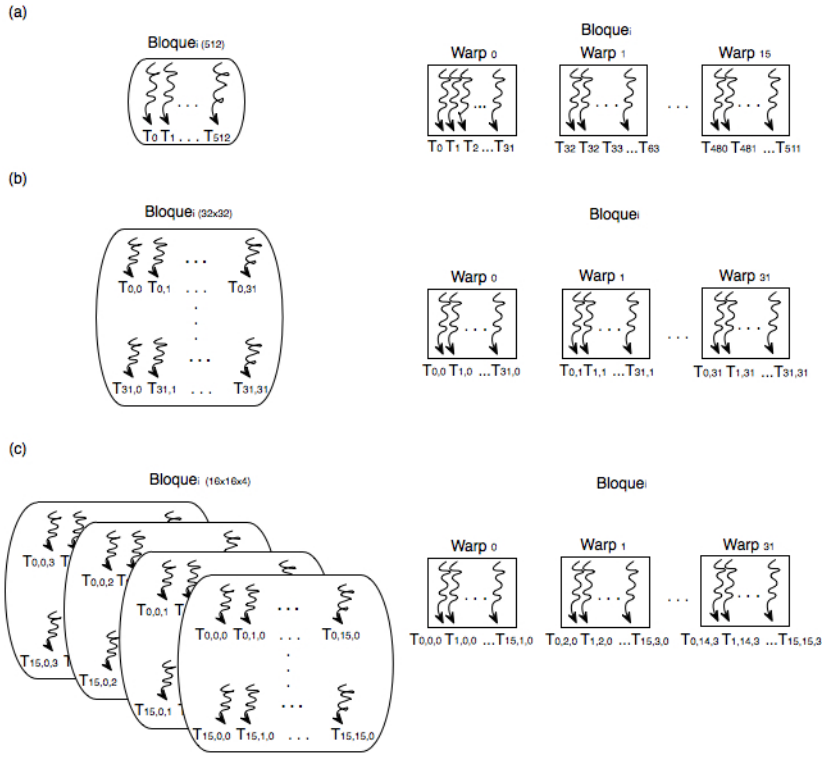


Figura 5.1. Asignación de *threads* de un *bloque* a *warps*

En el caso que la cantidad de *thread* no es múltiplo de 32, el último *warp* se completa con *threads* adicionales.

Como se mencionó en el capítulo 3, los *warps* aplican un modelo de ejecución SIMT (*Simple Instrucción-Múltiples Threads*), la misma instrucción es ejecutada por todos los *threads*, permitiendo que el gasto de traer una instrucción y procesarla se amortice con la ejecución de muchos *threads*, siempre y cuando los *threads* no diversifiquen sus caminos. Esto ocurre, por ejemplo, cuando los *threads* ejecutan sentencias condicionales y no todos los *threads* siguen el mismo camino.
Cuando existen varios caminos, la ejecución del *warp* implica varios pasos, uno para cada una de las partes en las que se diversifica el camino, luego los pasos se ejecutan secuencialmente, lo cual implica más tiempo. Éste será igual a la suma de los tiempos que demore en ejecutar los *threads* de cada opción. Las sentencias involucradas en esta forma de ejecución de los *warps* son todas aquellas que implican dividir el grupo de *threads* en 2 o más caminos de ejecución de un grupo de *threads*. Un ejemplo del uso de la sentencia condicional se ve en el ejemplo de la reducción por suma en paralelo desarrollado en el capítulo 4. En la figura 5.2 se desarrollan dos soluciones a la reducción por suma. A través de ellas se analiza la influencia en la performance de las sentencias condicionales.

```
1.  __shared__ float cache[threadsPorBloques];
2.   int tid = threadIdx.x;
3.   int salto;

4.   salto= 1;

5.   while (salto< blockDim.x ) {
6.     if (tid % (2*salto) == 0)
7.       cache[tid] += cache[tid+ salto];
8.     __syncthreads();
9.     salto=salto <<1 ;
10.  }
11. }
```

(a) Versión A

```
1.  __shared__ float cache[threadsPorBloques];
2.    int tid = threadIdx.x;
3.    int salto;

4.    salto= blockDim.x>>1;

5.    while (salto> 0) {
6.      if (tid < salto)
7.        cache[tid] += cache[tid+ salto];
8.      __syncthreads();
9.      salto=salto >>1 ;
10.   }
11.  }
```

(b) Versión B

Figura 5.2. Dos versiones de la reduc_sum()

En ambos código se detalla solamente la parte correspondientes a la resolución de la reducción por suma en paralelo. Se propone realizar la reducción en paralelo a través de varias iteraciones, las cuales realizan la operación de reducción sobre la misma variable, *in place*, definida en memoria compartida. Esto significa que la iteración *i*utiliza los resultados

de la iteración *i-1*. En la última iteración el resultado de la reducción queda en la posición cero de la variable *cache*. Existe la necesidad de la sincronización, línea 8 de ambos códigos, no se puede pasar a la iteración siguiente si todos los *threads* no calcularon las sumas parciales. Observe que la sincronización la hacen todos los *threads*, no sólo aquellos que ejecuten por el if (Recuerde que una sincronización por la rama del if y otra por la del else son considerados puntos de sincronización distintos).
En el código de la figura 5.2 (a) la variable *salto* se utiliza para determinar q qué distancia está el otro elemento a sumar, ésta es inicializada en 1. La sentencia de la línea 6 selecciona a los *threads* pares para sumar al valor que se encuentra en la posición correspondiente a su identificador con el valor a *salto* de distancia de él, en la primer iteración la distancia es 1. Para cada iteración un paso sería utilizado para ejecutar la sentencia de la línea 7 y otro para los que no cumplen con la condición. Observe que la primera iteración sólo los *threads* con identificador par estarán en el primer paso, los otros en el segundo. A medida que se avanza en las iteraciones, cada vez menos *threads* pertenecen al paso que ejecuta la sentencia de la línea 7. Para un blockDim.x igual a 256 *threads*, son necesarias 8 iteraciones. La figura 5.3 muestra un esquema de cómo se realiza la reducción por suma cuando blockDim.x es 8. En ella se puede observar la secuencia de sumas que se realizan y los *threads* involucrados en ellas.

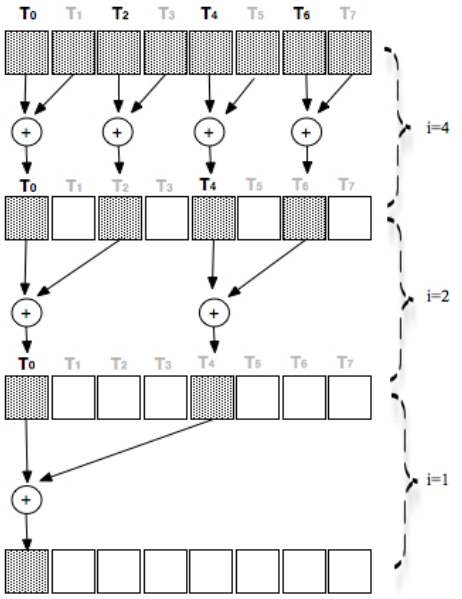


Figura 5.3. Ejecución de la reducción por suma versión (a)

En el código de la figura 5.2 (b) se presenta una mejora del código anterior. En esta propuesta, la suma no se realiza con los vecinos más cercanos sino que se divide el vector en dos mitades y los elementos que se suman están en la misma posición pero de mitades distintas. En la primera iteración los *threads* trabajan sobre el vector completo, en la segunda sobre la primera mitad, en la tercera sobre el primer cuarto, y así siguiendo. Observe la variable *salto,* se inicializa en la mitad del arreglo y en cada división se lo divide por 2. En la figura 5.4 puede observarse el esquema de ejecución para un vector de 8 elementos. En la parte superior de la figura se especifican los *threads* que trabajan en cada iteración.

Aunque la cantidad de pasos involucrados en la ejecución de todos los *threads* en ambas soluciones es la misma, la diferencia está en el orden en que se realizan. Observe que en esta última propuesta los *threads* que ejecutan la sentencia de la línea 7 son *threads* consecutivos. Si el *bloque* tiene 512 *threads*, los primeros 256 ejecutan la línea 7 y el resto no hace nada, esto significa que de los 16 *warps* en los que se dividen los *threads* para ejecutarse, los 8 primeros *warps* ejecutan la sentencia y los otros restantes no hacen nada. En la segunda iteración, los 4 primeros *warps* sólo ejecutaran la sentencia.

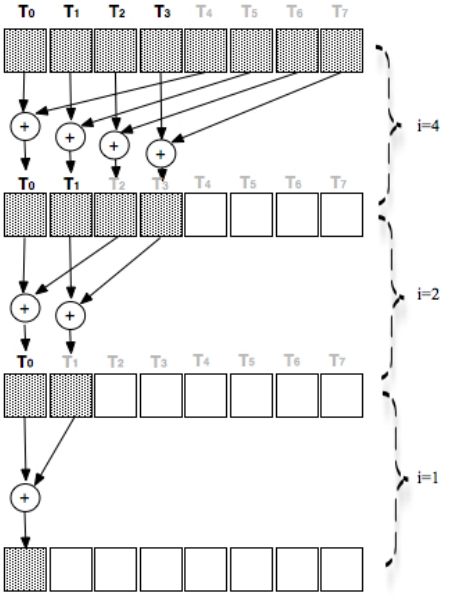


Figura 5.4. Ejecución de la reducción por suma versión (b)

Con esta solución se puede evitar parte de la divergencia presente en todos los *warps* de la anterior propuesta. Cuando el número de *threads*

es menor a la cantidad de *threads* en un *warps*, se presenta nuevamente la divergencia y en consecuencia la secuencialización de los caminos distintos que toman los *threads* en un *warp*. Observe cómo con una simple organización de los *threads* que ejecutan una sentencia de selección se puede disminuir el tiempo de ejecución reduciendo la divergencia de los caminos en un *warp*.
La estructura de la solución presentada para la reducción por suma puede ser utilizada para resolver la mayoría de las operaciones de reducción en la GPU. Recuerde que cualquier operación conmutativa y asociativa puede ser utilizada para realizar una reducción en paralelo. Puede Ud. analizar por qué son necesarias estas características para una función por reducción.

5.2. Memoria Global

Como se mencionó en los capítulos anteriores, la GPU posee una jerarquía de memoria, cada una con distintas características. La memoria global es la memoria más utilizada, es el medio de comunicación entre las CPU y la GPU, los *threads* acceden a ella para leer o escribir datos, los cuales pueden ser accedidos por *threads* de distintos kernels de la aplicación. La mayoría de las aplicaciones en GPU realizan accesos masivos a la memoria global, siendo estos muy costosos.
En las próximas secciones se analizan dos aspectos a considerar para mejorar el desempeño de una aplicación, ellas son la organización de los accesos y la carga anticipada (*prefetching*) de datos.

5.2.1. Organización de Accesos

Los accesos a memoria global son uno de los aspectos a considerar a la hora de ajustar la performance de una aplicación. En esta sección se analiza qué y cómo se puede optimizar.
La memoria global es una memoria *off-chip*, está implementada como memorias de acceso aleatorio DRAM. Conocer su organización y forma de lectura permite al programador organizar los accesos de manera de hacerlos en forma eficiente. La mayoría de las DRAM realizan lecturas simultáneas de una posición y sus vecinos más cercanos, lo cual implica que si uno accede a una posición de la memoria, las posiciones anexas también son leídas al mismo tiempo. Las GPU actuales poseen estos mecanismos de optimización en sus memorias. Esto unido al hecho que la misma instrucción es ejecutada

por todos los *threads* del kernel hace que la optimización de los accesos a memoria global sea posible.
El patrón de acceso a memoria óptimo es cuando se realiza a posiciones consecutivas. En el caso de la GPU, cuando los *threads* de un *grid* ejecutan una instrucción de acceso a la memoria, ésta detecta si los accesos son a posiciones consecutivas, de ser así el hardware las organiza mediante su combinación o fusión en un único acceso de posiciones consecutivas de la DRAM, esto es conocido como accesos *coalesced* (Kirk, 2010) (NVIDIA., 2011). Por ejemplo, para un *warp* dado, si sus *threads* acceden a posiciones contiguas de memoria, es decir si el $thread_0$ accede a la posición x, el $thread_1$ a la posición $x+1$ y así siguiendo hasta el *thread* 31 a la posición $x+31$, todos se unirán en un único acceso a la DRAM. Al mismo tiempo se accede a varias posiciones de memoria pudiendo obtener una velocidad de acceso cercana al ancho de banda teórico de la memoria.
Para el ejemplo de la multiplicación de matrices, si la matriz A es almacenada en la memoria como lo muestra la figura 5.5, en la figura 5.6 se muestran dos patrones de accesos, (a) *no coalesced* y (b) *coalesced*.

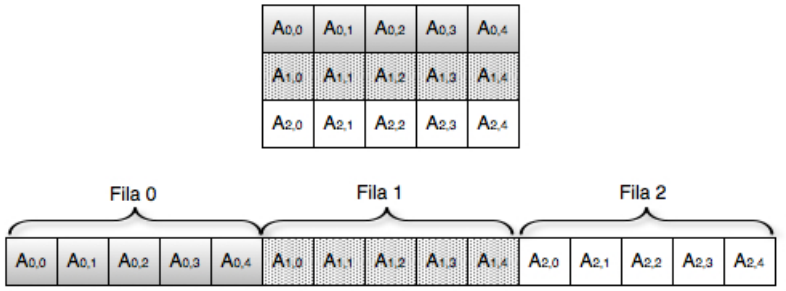
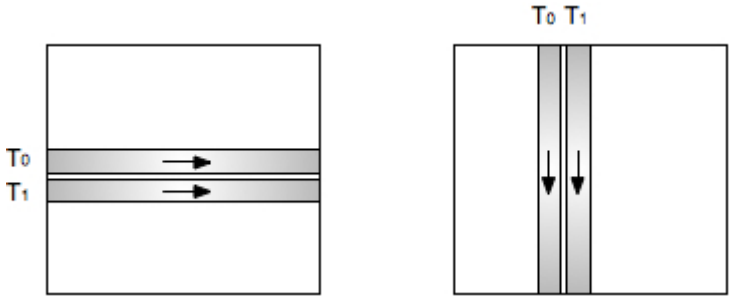


Figura 5.5. Almacenamiento lineal de una matriz de dos dimensiones

En la figura 5.6 (a), el acceso de los *threads* es *no coalesced*, ellos acceden a una fila completa, lo cual significa que en un *warp* los *threads* 0 a 31 leen el elemento 0 de las filas 0 a la fila 31, en la siguiente iteración estos mismos *threads* leen la posición siguiente de la lista, o sea la posición 1 de todas las filas. Observe que no se pueden realizar accesos *coalesced* porque los elementos accedidos por los *threads* en cada iteración no se encuentran almacenados consecutivamente en la memoria, para tres *threads*, la secuencia de accesos simultáneos de la figura 5.5 sería:

- Iteración 0: $A_{0,0}$, $A_{1,0}$, $A_{2,0}$
- Iteración 1: $A_{0,1}$, $A_{1,1}$, $A_{2,1}$
- Iteración 2: $A_{0,2}$, $A_{1,2}$, $A_{2,2}$
- Iteración 3: $A_{0,3}$, $A_{1,3}$, $A_{2,3}$
- Iteración 4: $A_{0,4}$, $A_{1,4}$, $A_{2,4}$.



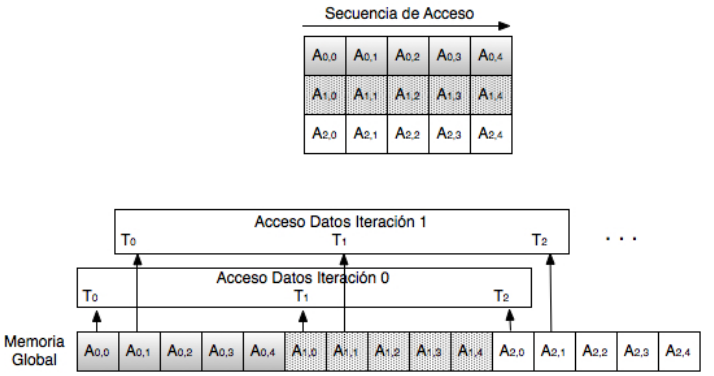
(a) No coalesced (b) Coalesced

Figura 5.6. Accesos a memoria global

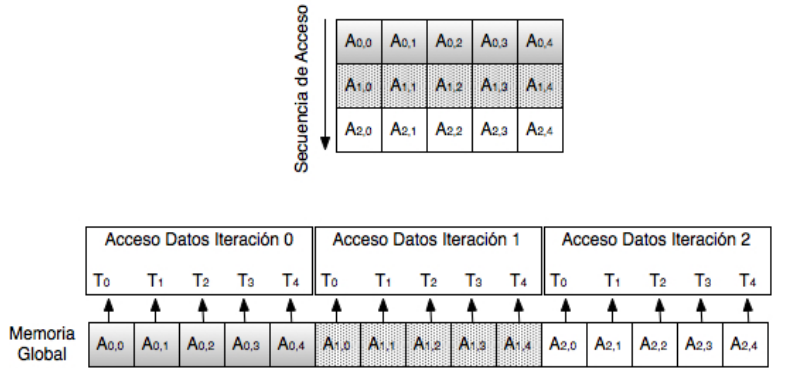
En el caso del patrón de acceso de la figura 5.6 (b) no ocurre lo mismo, los *threads* acceden a las columnas de la matriz, es así que para un *warp*, el *threads* 0 a 31 acceden al elemento 0 de las columnas 0 a la 31, en la siguiente iteración acceden al elemento 1 de las mismas columnas, y así siguiendo para todos los elementos de las columnas. Para el ejemplo de la figura 5.5, si se tienen 5 *threads* para acceder a todos los elementos de las columnas, la secuencia de acceso simultáneos serán:

- Iteración 0: $A_{0,0}$, $A_{0,1}$, $A_{0,2}$, $A_{0,3}$, $A_{0,4}$
- Iteración 1: $A_{1,0}$, $A_{1,1}$, $A_{1,2}$, $A_{1,3}$, $A_{1,4}$
- Iteración 2: $A_{2,0}$, $A_{2,1}$, $A_{2,}2$, $A2_{,3}$, $A_{2,4}$.

Comparando la secuencia de accesos anterior con la figura 5.5, estos se realizan en el mismo orden como están almacenados los elementos de la matriz, lo cual significa que se pueden fusionar y en un único acceso recuperar todos los elementos referenciados. En la figura 5.7 se muestra gráficamente ambas secuencia de acceso, (a) *no coalesced* y (b) *coalesced*.



(a) No coalesced



(b) Coalesced

Figura 5.7. Resolución de los patrones de accesos a memoria global

Por todo lo expuesto, ahora se procederá a analizar el código de la multiplicación de matrices expuesto en el capítulo 4, figura 4.17. La solución propuesta dividía a las matrices en *bloques*, sobre los cuales se computaban los productos parciales y finalmente eran combinados para obtener el resultado final.

Si resolvemos la multiplicación de la matriz *A* por *B* de la manera trivial, cada *thread* del *grid* lee una fila de la matriz *A* y una columna de la matriz *B*. Como el primer caso es un acceso *no coalesced*, la propuesta es reducir las consecuencias de este tipo de accesos mediante la partición de la matriz en *bloques* de datos. De esta manera, no sólo se logra utilizar la memoria compartida, sino también realizar accesos *coalesced* a la memoria global. Para realizarlo se debe hacer una cuidadosa planificación de los accesos a la memoria de cada *threads*, es decir se debe prestar especial atención a las sentencias de las líneas 11 y 12 del código de la figura 5.8. Estas dos líneas se corresponden con la carga a memoria compartida de cada uno de los segmentos de las matrices con los que va a trabajar cada *bloque* del *grid*. Una vez que los datos están en la memoria compartida, no se tendrá más que tratar de organizar los accesos para que puedan unificarse en uno sólo.

```
1.  __global__ void MatMul_Shared(Matrix A, Matrix B, Matrix C)
2.  {
3.   __shared__ float As[Dim_sector][Dim_sector];
4.   __shared__ float Bs[Dim_sector][Dim_sector];

5.  int bx = blockIdx.y;
6.   int by = blockIdx.x;
7.   int fila = by * Dim_sector+threadIdx.y;
8.   int col = bx * Dim_sector + threadIdx.x;
9.  float Cvalor = 0;

10.  for (int m = 0; m < (N/ Dim_sector); m++) {

11.    As[threadIdx.y][threadIdx.x] = A[fila*N+m*Dim_sector+threadIdx.x];
12.    Bs[threadIdx.y][threadIdx.x] = B[(m*Dim_sector+threadIdx.y)*N + col];

13.  __syncthreads();

14.   for (int k = 0; k <Dim_sector; k++)
15.     Cvalor += As[fila][k] * B[k][col];

16.  __syncthreads();
17.  }

18.  C[fila*N+col]= Cvalor;
19. }
```

Figura 5.8. Función kernel de la multiplicación de matrices usando memoria compartida

Ahora se analizará cómo acceder de forma ordenada a los datos en la memoria global. En la solución de la figura 5.8 cada *thread* es responsable de acceder a un elemento por iteración de la memoria global y copiar los datos a la memoria compartida, línea 10. Al ser *bloques* bi-dimensionales con *Dim_sector * Dim_sectorthreads*, estos se identifican únicamente con las variables threadIdx.x y threadIdx.y, ambas utilizadas para el acceso a los elementos en la memoria global. Las filas del *sector* de la matriz *A* son copiadas en la línea 11, donde *Dim_sectorthreads* se diferencian por el identificador threadIdx.x, el cual toma valores consecutivos significando que pertenecerán al mismo *warp*. La organización de los accesos está dada por la expresión

Fila*N+m*Dim_sector*+threadIdx.x

la cual determina que los accesos sean en la misma fila. Como se debe asegurar que los accesos consecutivos sean hechos por *threads* consecutivos, analizando la expresión anterior, se puede observar que (m*Dim_sector*+threadIdx.x) es idéntico para todos los *threads*, a excepción de threadIdx.x, los cuales son valores consecutivos para los *threads* del *bloque*. Por todo lo expuesto se llega a la conclusión que los accesos según este patrón de acceso son *coalesced.*
Para el caso de los elementos de *B*, los accesos se dan según la expresión

(m*Dim_sector*+threadIdx.y)*N + Col

donde tienen el mismo valor para todos los *threads*, salvo *Col*, la cual se define en función de threadIdx.x, valor diferencial y consecutivo de cada *thread*. Al igual que para la matriz *A*, en este caso los accesos también son *coalesced* por ser valores adyacentes en la memoria global.
La figura 5.9 muestra gráficamente, y para un ejemplo de matices de 6*6 y segmentos de 3*3, los accesos *coalesced* a las matrices *A* y *B*, ambas almacenadas linealmente por filas.

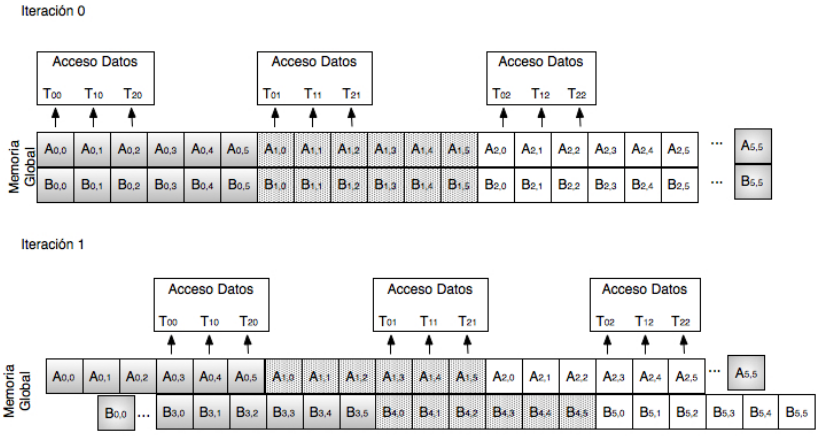


Figura 5.9. Resolución de los patrones de accesos a las matrices *A* y *B*

La organización de los accesos a la memoria global descripta aquí puede ser utilizada por los algoritmos que trabajan con matrices divididas por *sectores*, la estructura diagramada es la estructura común a utilizar.

5.2.2. Prefetching de Datos

La técnica propuesta en la sección anterior permite, a través de la organización de los accesos a la memoria global, lograr mejoras en la performance de la aplicación, pero muchas veces esto no es suficiente. Por ello, CUDA provee de mecanismos para reducir la latencia de la memoria global. El modelo de *threading* permite a algunos *warps* ejecutar o avanzar en la ejecución mientras otros esperan por accesos de memoria, es decir mientras unos *warps* esperan por la resolución de una operación de acceso a memoria, otros ejecutan sentencias menos costosas, por ejemplo operaciones aritméticas.
Ejecutar otras sentencias entre accesos a memoria es posible si los *threads* son programados de manera que entre las operaciones de accesos a memoria existen instrucciones independientes, las cuales se pueden ejecutar mientras se resuelve el acceso. Se consideran instrucciones independientes a todas aquellas que al momento de ejecutarse, los datos necesarios para su resolución están presentes. Una buena técnica para mejorar la performance entonces es asegurar que todos los datos necesarios para las operaciones estén presentes al momento de ejecutarlas. Esta técnica se denomina *prefetching* de datos (carga anticipada), otra forma de reducir la latencia de la memoria.
Analizando el ejemplo de la multiplicación de matrices de la figura 5.8, en las líneas 11 y 12 se realiza la carga de los datos desde la memoria global a la compartida. En la figura 5.10 se muestra a la izquierda las líneas de interés de la figura 5.8. y a la derecha la descripción de los efectos de la/las sentencias.

```
10. for (int m = 0; m < (N/ Dim_sector); m++) {        Iteración {
11.  As[threadIdx.y][threadIdx.x] =
     A[fila*N+m*Dim_sector+threadIdx.x];               Carga el sector actual a
12.  Bs[threadIdx.y][threadIdx.x] =                      mem.shared.
     B[(m*Dim_sector+threadIdx.y)*N + col];

13. __syncthreads();                                   Sincronización.

14.  for (int k = 0; k <Dim_sector; k++)
15.     Cvalor += As[fila][k] * B[k][col];             Computa el sector actual.
16. __syncthreads();                                   Sincronización.
17. }                                                  }
```

Figura 5.10. Multiplicación de matrices sin *prefetching*

La sentencias de la línea 11 y 12 tienen dos partes dependientes entre si, en la primera se accede a la memoria global para leer el contenido de *A* y *B* en la posición indicada. La segunda parte se corresponde con

almacenar el valor leído en la posición establecida de la memoria compartida. Los *warps* deben esperar a que el acceso se resuelva antes de continuar con otra operación. La existencia de muchos *warps* permitirá reducir las esperas por estas operaciones. Si en cambio se aplican técnicas de *prefetching*, la idea es traer con antelación desde la memoria global los datos para trabajar en las operaciones e la siguiente iteración. En la figura 5.11 se puede observar la modificación del código de la figura 5.10 para realizar el *prefetching* de los datos.

```
10.  float Ra= A[fila*N+threadIdx.x];
11.  float Rb= B[(threadIdx.y)*N + col];          // Carga sector en registros.

12.  for (int m = 0; m < (N/ Dim_sector); m++) {  // Iteración {
13.   As[threadIdx.y][threadIdx.x] = Ra ;
14.   Bs[threadIdx.y][threadIdx.x] = Rb;          // Guarda registros en mem. Shared.

15.  Ra= A[fila*N+(m+1)*Dim_sector+threadIdx.x];
16.  Rb= B[((m+1)*Dim_sector+threadIdx.y)*N + col]; // Carga el próximo sector en los registros

17.  __syncthreads();                            // Sincronización.

18.   for (int k = 0; k <Dim_sector; k++)
19.      Cvalor += As[fila][k] * B[k][col];       // Computa el sector actual.

20.  __syncthreads();                            // Sincronización.
21.  }                                           // }
```

Figura 5.11. Multiplicación de matrices sin *prefetching*

Varias modificaciones o agregados fueron hechas, primero se definen dos registros *Ra* y *Rb*, a los cuales se les asigna los datos leídos desde la memoria global, líneas 10 y 11. El contenido de *Ra* y *Rb* es almacenado en la memoria compartida definida para realizar la multiplicación, líneas 13 y 14. En las líneas 15 y 16 se copian en los registros los datos de la próxima iteración. Finalmente se operan los datos actuales desde la memoria *shared*. Estas modificaciones permiten reducir la inactividad del sistema, mientras unos *threads* estén inactivos esperando se completen sus accesos a memoria, otros lo están solicitando. Cuando todos completan la lectura, pasan la sincronización operan los datos presentes en la memoria compartida e inician de nuevo la carga de los datos. En la siguiente iteración los últimos elementos cargados se convierten en los actuales y se procede a una nueva carga.

De la manera planteada, la nueva solución de la multiplicación de matrices pudo lograr independencia en las operaciones y así reducir los tiempos de ejecución de los *threads* tomando ventaja del modelo de *threading* de CUDA.

5.3. Rendimiento de las Instrucciones

En las secciones previas se analizaron distintos aspectos a tener en cuenta para la mejora de la performance considerando la ejecución de los *threads* y a diferentes aspectos de la memoria global. En esta sección se consideran aquellos relacionados a las instrucciones del kernel.

5.3.1. Mezcla de Instrucciones

En la GPU, no todas las instrucciones implican el mismo tiempo, por ejemplo una operación de punto flotante, una instrucción de acceso a memoria o una sentencia de *branch* son las más costosas. De todas estas instrucciones, algunas pueden evitarse bajo ciertas condiciones, por ejemplo una sentencia de iteración implica varias instrucciones extras, ellas son: una para la evaluación de la condición (sentencia *branch*) y otra para la actualización del contador. Si la iteración puede reemplazarse por una única sentencia donde se explicita todo la iteración, esto implica evitar las sentencias de: *branch* y aritmética sobre el contador.
Si se analiza la multiplicación de matrices, se puede observar que la iteración donde se realiza el producto punto entre los dos vectores de las matrices, líneas 15 y 16 de la figura 5.8, tienen lugar varios tipos de instrucciones, ellas son:

- Dos operaciones de punto flotante, uno para el producto y otra la suma
- Una instrucción para la modificación del contador de la iteración *k*
- Una operación *branch* de la iteración.
- Dos direccionamiento aritméticos, el uso de *k* en el índice de los elementos a operar implica también operaciones aritméticas.

Todas ellas compiten por el ancho de banda del procesamiento de instrucción. En esta mezcla de instrucciones, sólo un tercio de las operaciones son de punto flotante, debiendo competir por el ancho de banda del procesador de instrucciones y limitando de la performance, la cual no podrá ser más de un tercio del ancho de banda límite.
La solución al problema antes enunciado es evitar la mezcla de instrucciones, tratando de lograr el máximo aprovechamiento del

ancho de banda para procesamiento de instrucciones. Una alternativa posible es eliminar la sentencia de iteración. Para el ejemplo de la multiplicación de matrices, el reemplazo es mostrado en la figura 5.12.

```
18. for (int k = 0; k <Dim_sector; k++)
19.     Cvalor += As[fila][k] * B[k][col];
```

(a) Propuesta Original

```
18.  Cvalor = As[fila][0] * B[0][col]+
     As[fila][1] * B[1][col]+…+
     As[fila][Dim_sector] *
                B[Dim_sector][col];
```

(b) Optimización

Figura 5.12. Optimización para la mezcla de instrucciones

En la figura 5.12(a) se muestra la sentencia de iteración original, mientras que en la parte (b) se detalla cómo se puede reemplazar a fin de evitar la mezcla de instrucciones. ¿Cuáles fueron las modificaciones? Se eliminó la instrucción *branch*, la actualización del contador y los direccionamientos aritméticos. Esto será posible cuando el programador conozca los índices de la iteración y pueda desarrollarla en una única sentencia.
Para maximizar el rendimiento de las aplicaciones respecto a la mezcla de instrucciones se debe:

- Minimizar el uso de instrucciones aritméticas con bajo rendimiento. CUDA además provee de instrucciones propias las cuales se pueden utilizar en lugar de las instrucciones regulares como así también funciones de simple precisión en lugar de doble precisión
- Reducir el número de instrucciones evitando la mezcla, a fin de no competir por el ancho de banda del procesamiento de instrucciones cuando existen operaciones muy costosas.

En este caso, lo ideal es contar con herramientas de software que realicen estas mejoras, por ejemplo compiladores que traduzcan una iteración como la mostrada en el ejemplo a su equivalente instrucción.

5.3.2. Granularidad

Como ocurre en el desarrollo de todo sistema paralelo, un punto importante a tener en cuenta es la granularidad de las computaciones. Las aplicaciones desarrolladas para la GPU son por naturaleza y por las características de la arquitectura, de gránulo fino. De todas maneras a la hora de desarrollar la aplicación surge la inquietud de cuánto trabajo se asigna a cada *threads* y cuántos *threads* ejecutan en paralelo.

No existe una metodología a aplicar para determinar la mejor granularidad de un *thread*, para ello se debe evaluar las características de la aplicación y las del kernel que la implementa. Si del análisis del kernel, por ejemplo, se deduce que varios *threads* realizan accesos a los mismos datos, esto puede reducirse haciendo que cada *threads* calcule más de un valor del resultado, los cuales serán todos aquellos que utilizan los datos previamente accedidos. Pero toda decisión implica realizar un cuidadoso análisis respecto a las otras características antes descriptas y que afectan a la performance de una aplicación. Para este caso, la única posibilidad de analizar la mejor granularidad de los *threads* es a través de la experimentación.

5.4. Asignación de los recursos de un SM

Un SM tiene diferentes recursos, los cuales son asignados dinámicamente a los *threads* para su ejecución. Los recursos a asignar en un SM son: los registros, los *slots* para los *bloques* de *threads* y los *slots* de *threads*. La cantidad de estos recursos dependen de la arquitectura. En la siguiente tabla (figura 5.10) se especifican el tamaño de cada uno de los recursos para cada arquitectura descripta en el capítulo 2.

GPU Recursos de SM	G80	GT200	GF100
Registros	8KB	16KB	32KB
Slots para *Bloques*	8		
Slots para *Threads*	768	1024	1536
Memoria *Shared*	16KB		48KB

Figura 5.13. Recursos de los SM para cada generación de GPU

Cada uno de estos valores constituyen los límites de los recursos, los cuales pueden influir directamente en la performance de las aplicaciones, se los debe considerar a la hora de diseñar una aplicación. A continuación se analizan cada uno de los distintos casos:

- Respecto a los *Threads*:
  - Número de *Threads*:
    Al momento de ejecutar un *grid*, los recursos se particionan y asignan a los distintos *bloques* de *threads*. La partición depende de la cantidad de *threads* por *bloques*, por ejemplo, para el caso de un SM de la G80, si los *bloques* tienen 256 *threads* cada uno, sólo 3 *bloques* de *threads* residen en el SM,

si en cambio cada *bloque* tiene 512 *threads* uno sólo podrá residir en el SM y si el *bloque* de *threads* tiene 128 *threads*, los *bloques* residentes en el SM serán 6. En el caso de un SM de la GT200, los valores correspondientes según las cantidades de *threads* antes mencionadas serán: 4, 2, 8 *bloques* por SM. Lo mismo se puede calcular para las GF100. Esta asignación dinámica permite un ajuste de los recursos en función de las necesidades de la aplicación, una partición fija y/o estática implica la subutilización de los recursos si estos no son necesarios o no son suficientes.

- Número de *Bloques*:
  Puede ocurrir que los *bloques* tienen pocos *threads* y, en consecuencia, requieren pocos recursos. Por ejemplo si se está trabajando con una arquitectura G80 y los *threads* por *bloques* son 32, entonces cada SM puede tener 24 *bloques*. Esto no será posible alcanzar porque cada SM puede albergar hasta 8 *bloques* como máximo, significando que, para el ejemplo dado sólo 256 *slots* del SM para *threads* serían utilizados, el resto no.

- Respecto a la demanda de Memoria:
  - Memoria de Registro:
    Otra de las limitaciones es la memoria de registros. Recuerde que en esta memoria se almacenan todas las variables que requieren de un uso frecuente y de accesos rápidos, en CUDA las variables automática no estructuradas declaradas en el ámbito de un kernel son consideradas variables automáticas. Como se mostró en la figura 5.10, las distintas arquitecturas tienen distintas capacidades de memoria de registro para sus SM. La memoria de recursos se divide entre todos los *bloques* residentes en el SM, derivando en una variable de ajuste a la hora de definir la cantidad de *bloques* y de *threads* por *bloques*.
    Suponga que la multiplicación de matrices requiere 9 registros por *threads* para su ejecución, si se tienen *bloques* de 16 x16 *threads*, cada *bloque* necesita 9*16*16 = 2304 registros, si la arquitectura es GT200, un SM puede satisfacer las necesidades de registros de 7 *bloques* (7*2304 = 16128) quedando 256 registros libres. Si una nueva implementación requiere 11 registros por *threads*, un *bloque* necesita 2816 registros por lo cual sólo 5 *bloques* serán ubicados por SM, restando 2304 registros sin utilizar. Esto muestra un débil equilibrio, a pequeñas modificaciones, 2 registros más por *threads*, se reduce el paralelismo entre *bloques* de 7 a 5, dejando muchos recursos sin utilizar.

- Memoria *Shared*:
  Al igual que los registros, cada SM tiene una capacidad máxima de memoria compartida, la cual es repartida entre los *bloques* residentes en el SM. Para el caso de la multiplicación de matrices, cada *bloque* requiere 2KB, esto significa que para las arquitecturas G80 y GT200 los *bloques* por SM pueden ser 8 (8*2KB = 16KB), en el caso de la GF100 también serán 8 quedando 30KB sin utilizar.

Cada uno de los recursos puede determinar una cantidad máxima de *bloques* por SM, la cual puede no coincidir con la determinada por otro recurso. El programador debe analizarlos en forma conjunta y tomar las decisiones en función de maximizar la performance de la aplicación. Esta tarea puede ser muy ardua, en (Ryoo, 2008) se propuso una metodología para reducir los esfuerzos de programación y obtener la mejor configuración para aprovechar todas las ventajas de la GPU.

5.5. Resumen

Para el desarrollo de aplicaciones con buen desempeño es necesario tomar ventajas de la arquitectura subyacente. Por ello es imperativo conocer las características y factores limitantes de la performance en las GPU.
En este capítulo se presentaron distintas técnicas para la optimización de una aplicación, abordando distintos aspectos, los cuales van desde la etapa de diseño hasta la de ejecución.
Uno de los aspectos considerados es la ejecución de los *threads*. El modelo de ejecución SIMT implica que todos los *threads* en el *warps* ejecutan la misma instrucción, esto no es posible cuando se está en presencia de instrucciones de divergencia de caminos. Una técnica es organizar el programa de manera tal que los *threads* que ejecuten un camino pertenezcan al mismo warp, evitando así la secuencialización de la ejecución de las sentencias.
Otro es la optimización de los accesos a la memoria global. Dos técnicas son presentadas, la primera es la organización de los accesos a fin de aprovechar las características de las lecturas, para ello se debe intentar realizar accesos *coalesced*. La otra es desarrollar una aplicación donde la latencia de los accesos a memoria se reduzca por la ejecución de sentencias independientes, para ello se implementan técnicas de *prefetching* de datos.
Además, es necesario prestar atención a los costos de las instrucciones, minimizando el uso de sentencias que utilicen el ancho de banda de procesamiento de instrucciones. Una de las técnicas

propuestas es el reemplazo de las sentencias de iteración por sentencias donde la operación esté desagregada. Respecto a la granularidad se debe evaluar, según la aplicación, si es beneficioso que los threads resuelvan varias operaciones en lugar de una.
Finalmente, la asignación de recursos en un SM es uno de los factores limitantes que implican un análisis detallado de la aplicación y su demanda de recursos, muchas veces solicitar una instancia más de un recurso, por ejemplo un registro, implica perder paralelismo y subutilizar los recursos disponibles. Es claro que exceder los límites lleva a pérdida de performance y de paralelismo.
El objetivo de este capítulo es mostrar al programador las posibles consecuencias de sus elecciones y tratar que, al menos intuitivamente, desarrolle las aplicaciones evitando u optimizando las limitaciones. Muchos de los aspectos considerados tienen influencia directa entre sí, optimizar uno puede implicar empeorar otro. Es por ello que un ajuste debe ser hecho considerando todos los factores en forma conjunta, evaluando los beneficios alcanzados versus los costos.

5.6. Ejercicios

5.1. Dado la siguiente secuencia de código de un kernel

```
__shared__ int sum[blockDim.x];
int i;

sum[threadIdx.x]=0;
size=(band_GPU[threadIdx.x]-
band_GPU[threadIdx.x-1]);

for(i=band_GPU[threadIdx.x-
1];i<band_GPU[threadIdx.x]+1;i++)
sum[threadIdx.x]+=dato[i];

  __syncthreads();
if (threadIdx.x==0){
for (i=0, d_r=0;i<blockDim.x;i++)
   d_r+=sum[threadIdx.x];
```

Analizar:
a. La ejecución de los threads
b. Características de los accesos a la memoria
Si es necesario, proponer las modificaciones necesarias para mejorar la performance.

5.2. Analice la asignación de recurso del problema de reducción por suma desarrollado en la sección 5.1.
¿Cuántos threads y bloques se ejecutan en paralelo por SM? Considere que cuenta con una arquitectura:
a. G80
b. GF100
Justifique cada caso.

5.3. Desea sumar dos matrices (cada elemento de la matriz resultado es la suma de los elementos correspondientes de las matrices de entrada) ¿Se puede utilizar la memoria compartida para reducir el ancho de banda de la memoria global?
Nota: Analizar si existe similitud en los elementos accedidos por cada thread.

5.4. Analizar si el tamaño de los sectores en la multiplicación de matrices influye en el ancho de banda de la mamaria global.
Nota: Realice un análisis similar al mostrado en la figura 4.15 para los siguientes tamaños de sectores: 2x2, 4x4 y 8x8.

5.5. Suponiendo que cuenta con una GPU sin límites de capacidad de las memorias de registros y de la compartida.
a. Analice si en el caso de la multiplicación de matrices es posible utilizar la memoria de registros en lugar de la memoria compartida. ¿Cuáles serían las consecuencias?
b. Proponga un ejemplo donde es conveniente usar la memoria compartida en lugar de la de registros para guardar valores leídos desde la memoria global.
En ambos casos, explique su respuesta.

5.6. Dada las siguientes secuencias de código de kernel:
A:

```
__shared__ int sum[blockDim.x];
int i;

sum[threadIdx.x]=0;
size=(band_GPU[threadIdx.x]-
band_GPU[threadIdx.x-1]);

for(i=band_GPU[threadIdx.x-
1];i<band_GPU[threadIdx.x]+1;i++)
sum[threadIdx.x]+=dato[i];
```

```
if (threadIdx.x==0){
for (i=0, d_r=0;i<blockDim.x;i++)
   d_r+=sum[threadIdx.x];
}
```

B:

```
__shared__ int sum[blockDim.x];
int i;

sum[threadIdx.x]=0;
size=(band_GPU[threadIdx.x]-
band_GPU[threadIdx.x-1]);

for(i=band_GPU[threadIdx.x-
1];i<band_GPU[threadIdx.x]+1;i++)
sum[threadIdx.x]+=dato[i];

if (threadIdx.x%2){
   d_r+=sum[threadIdx.x];
   __syncthreads();
}
else){
   d_r+=sum[threadIdx.x];
__syncthreads();
}
```

Analice:

a. Posibles inconvenientes en cada código.
b. Modifique los códigos de manera que funcionen bien.
c. La ejecución de los threads de ambas versiones y la granularidad.

5.7. Son necesarias las dos sincronizaciones en la multiplicación de matrices de los códigos de las figuras 5.10 y 5.11.
Analice para ambos casos, ¿Qué ocurre si?:

a. No están ambas sincronizaciones.
b. Sólo existe la primera.
c. Sólo existe la segunda.

En caso de ser posible, muestre ejemplos de las consecuencias.

5.8. Para la solución propuesta al problema del histograma (Ejercicio 4.3), analice todos los aspectos a considerar para la mejora de la performance.

APÉNDICE A

CUDA: Extensiones básicas del lenguaje C

En este apéndice se detallan los calificadores, tipos y funciones básicas para el desarrollo de programas en CUDA. Para una especificación más detallada ir a (NVIDIA, NVIDIA CUDA C Programming Guide. Versión 4.0., 2011).

A.1 Calificadores de tipo de Función

Los calificadores de tipo de función especifican donde se ejecuta la función, si es sobre el host o en el dispositivo. Además determina desde donde se puede invocar la función.

__device__
Especifica una función con las siguientes características:
Se ejecuta en el dispositivo
La función sólo puede ser invocada desde el dispositivo.

__global__
Determina que la función es una función kernel. La cual es:
Ejecutada en el dispositivo
Se invoca solamente desde el host.

__host__
Especifica que la función es:
Ejecutada en el host.
Invocada desde el host
En caso de no especificarse este calificador, tiene el mismo efecto.

__global__ y __host__ no pueden ser usados juntos en una misma función.

__device__ y __host__ pueden ser usados juntos, el efecto es la compilación de la misma función tanto para el dispositivo como para el host.

A.2. Calificadores de tipo de Variables

Los calificadores de tipos de variables establecen en qué memoria de la jerarquía de memorias del dispositivo donde va a residir la variable. Una variable automática declarada en el dispositivo sin ninguno de los calificadores aquí descriptos reside en la memoria de registros. Si es una variable estructurada, reside en la memoria local.

__device__
Declara la variable como variable del dispositivo. Puede ser usado con los otros calificadores de memoria. Si estos no están presentes entonces:

- La variable reside en la memoria global.
- Su ciclo de vida es toda la aplicación.
- La acceden todos los threads del grid.
- Puede ser accedida desde el host mediante la invocación de funciones específicas de la biblioteca, tal como: cudaGetSymbolAddress(), cudaGetSymbolSize(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol(), entre otras.

__constant__
Puede ser usada con el calificador __device__.Declara una variable con las siguientes propiedades:

- Reside en la memoria constante
- Su ciclo de vida es toda la aplicación
- La acceden todos los threads del grid
- Puede ser accedida desde el host a través de las funciones de la biblioteca: cudaGetSymbolAddress(), cudaGetSymbolSize(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol(), entre otras.

__shared__
Opcionalmente usado con __device__. La variable calificada tiene las siguientes características:

- Reside en el espacio de memoria compartida de un bloque de threads.
- Su tiempo de vida se limita al grid.
- Su alcance es a todos los threads de un bloque.

A.3. Tipos vector *built-in*

char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, longlong1, ulonglong1, longlong2, ulonglong2, float1, float2, float3, float4, double1, double2.

Estos tipos vector derivan desde los tipos básicos entero (int) y de punto flotante (float). Son estructurados, tienen 1, 2, 3 y 4 componentes. Según el caso, los campos se acceden a través de los campos x, y, z y w.
Todos poseen una función constructor de la forma make_<type name>; por ejemplo:

```
int2 make_int2(int x, int y);
```

crea un vector del tipo int2 con valores (*x, y*).

En el código del host, el requisito de alineación de un tipo vector es igual a la exigencia de la alineación de su tipo base. En la GPU no siempre es el caso. En la siguiente tabla se detallan para cada tipo.

Tipo	Alineación
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4
short3, ushort3	2
short4, ushort4	8
int1, uint1	4
int2, uint2	8
int3, uint3	4
int4, uint4	16

long1, ulong1	4 - Si el sizeof(long) es igual al sizeof(int), 8 - En el otro caso
long2, ulong2	8 - Si el sizeof(long) es igual al sizeof(int), 16 - En el otro caso
long3, ulong3	4 - Si el sizeof(long) es igual al sizeof(int), 8 - En el otro caso
long4, ulong4	16
longlong1, ulonglong1	8
longlong2, ulonglong2	16
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16

dim3

Este tipo es un vector de enteros basado en el tipo uint3. Su función es usarlo para especificar dimensiones, por ejemplo las dimensiones de un grid y de un bloque. Cuando se define una variable del tipo dim3, las componentes no especificadas son inicializadas en 1.

A.4. Variables Built-in

Las variables built-in especifican las dimensiones de un grid y de un bloque, además de los identificadores de los bloques y los threads. Sólo pueden ser usadas dentro del ámbito de funciones ejecutadas en el dispositivo.

gridDim

Variable de tipo dim3. Contiene las dimensiones del grid.

blockIdx

Variable de tipo uint3, contiene el identificador del bloque en un grid.

blockDim
: Variable de tipo dim3. Contiene las dimensiones del bloque.

threadIdx
: Variable de tipo uint3, contiene el identificador del threads en un bloque.

warpSize
: Variable de tipo int. Indica la cantidad de threads en el warp.

A.5. Funciones para Administración de Memoria

Las siguientes funciones permiten reservar y liberar espacio lineal en memoria. Todas son invocadas desde el código del host.
Las funciones son:

cudaMalloc((void **) &ptr_vble, size)
: Reserva en la memoria global *size* bytes los cuales serán apuntados por *ptr_vble*.

cudaFree(void ptr_vble).
: Libera la memoria apuntada por *ptr_vble.*

cudaMemcpy(destino, origen, size, DIRECCION_COPIA)
: Copia *size* bytes desde la dirección *origen* a la dirección *destino* en la memoria global. El parámetro *DIRECCION_COPIA* indica el sentido de la copia, se hace a través de las siguientes constantes definidas:
  - cudaMemcpyHostToHost: Copia desde la memoria principal a la memoria principal.
  - cudaMemcpyHostToDevice: Desde la memoria principal a la memoria del dispositivo.
  - cudaMemcpyDeviceToHost: Copia desde la memoria del dispositivo a la memoria principal.
  - cudaMemcpyDeviceToDevice: Desde la memoria del dispositivo a la memoria del dispositivo.
  - La función cudaMemcpy() permite realizar copias entre ubicaciones de memoria del host y del dispositivo, no entre varios dispositivos (Ambientes con múltiples GPU).

cudaMemcpyToSymbol(destino, origen, size)

Copia *size* bytes desde la dirección *origen* a la dirección *destino* en la memoria de constantes.

A.6. Funciones de Sincronización

Para hacer una sincronización por barrera se pueden usar una de las siguientes funciones de sincronización.

void __syncthreads();

Esta función realiza una sincronización por barrera de todos los threads de un bloque. Ningún threads avanza hasta que todos los threads del bloque alcancen la función.

Esta función es usada para coordinar la comunicación entre los threads de un bloque. Muchas veces pueden ocurrir errores producto de operaciones del tipo leer-escribir, escribir-leer o escribir-escribir por distintos threads a la misma posición de la memoria (global o compartida). Existen casos en que dichos problemas pueden ser evitados utilizando la sincronización de los threads.

__syncthreads() es permitida en una sentencia condicional si todos los threads la ejecutan.

Los dispositivos de capacidad 2.*x* soportan tres variaciones de esta función de sincronización. Ellas son:

int __syncthreads_count(int *predicado*);

Idéntica a __syncthreads() con la característica adicional que evalúa *predicado* para todos los threads del bloque y retorna el número de threads donde *predicado* fue verdadero (no-cero).

int __syncthreads_and(int *predicado*);

Idéntica a __syncthreads() con la evaluación adicional de *predicado* para todos los threads del bloque. Retorna no-cero si todos los threads evaluaron no-cero a *predicado*.

int __syncthreads_or(int *predicado*);

Idéntica a __syncthreads() con la evaluación de *predicado* por todos los threads del bloque . Retorna no-cero si alguno de los threads evaluó a *predicado* como no-cero.

A.7. Funciones Atómicas

Una función atómica realiza operaciones de lectura-modificación-escritura sobre palabras de 32-bit o 64-bit de la memoria global o compartida.
Una operación es atómica cuando se realiza sin interferencia de otros threads, en otras palabras si un thread accede a una posición de memoria, ningún otro thread puede acceder al mismo tiempo.
El uso de las funciones atómicas depende de las capacidades de procesamiento de la GPU. Pueden usarse desde la capacidad 1.1. No todas las capacidades soportan todas las funciones atómicas. Las funciones atómicas sobre la memoria compartida y para palabras de 64-bit en la memoria global son posibles en las GPU de capacidades 1.2. Las funciones atómicas de palabras de 64-bit en la memoria compartida son soportadas en dispositivos de capacidad 2.*x*.
Las funciones atómicas son:

Funciones Aritméticas

atomicAdd()

int atomicAdd(int* address, int val);

unsigned int atomicAdd(unsigned int* address, unsigned int val);

unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);

float atomicAdd(float* address, float val);

Lee el contenido, palabra de 32-bit o 64-bit,apuntado por *address* ubicado en la memoria global o compartida, al contenido le suma *val*, y, finalmente, almacena el resultado en *address*. Las tres operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

atomicSub()

int atomicSub(int* address, int val);

unsigned int atomicSub(unsigned int* address, unsigned int val);

Lee el contenido, palabra de 32-bit,apuntado por *address* ubicado en la memoria global o compartida, al contenido le resta *val*, y, finalmente, almacena el resultado en *address*. Las tres operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

atomicExch()

int atomicExch(int* address, int val);

unsigned int atomicExch(unsigned int* address, unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address, unsigned long long int val);
float atomicExch(float* address, float val);

Lee el contenido, palabra de 32-bit o de 64-bit,apuntado por *address* ubicado en la memoria global o compartida y almacena el valor *val* en *address*. Las dos operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

atomicMin()

int atomicMin(int* address, int val);
unsigned int atomicMin(unsigned int* address, unsigned int val);

Lee el contenido, palabra de 32-bit,apuntado por *address* ubicado en la memoria global o compartida, obtiene el mínimo ente el contenido y *val*, y lo almacena en *address*. Las tres operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

atomicMax()

int atomicMax(int* address, int val);
unsigned int atomicMax(unsigned int* address, unsigned int val);

Lee el contenido, palabra de 32-bit,apuntado por *address* ubicado en la memoria global o compartida, obtiene el máximo ente el contenido y *val*, y lo almacena en *address*. Las tres operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

atomicInc()

unsigned int atomicInc(unsigned int* address, unsigned int val);

Lee el contenido, palabra de 32-bit,apuntado por *address* ubicado en la memoria global o compartida, computa ((contenido>=*val)? 0: (*contenido+*1))*, y, finalmente, almacena el resultado en *address*. Las tres operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

atomicDec()

unsigned int atomicDec(unsigned int* address, unsigned int val);

Lee el contenido, palabra de 32-bit,apuntado por *address* ubicado en la memoria global o compartida, computa (((contenido==*0)* | (contenido>*val))? val: (*contenido-*1))*, y, finalmente, almacena el resultado en *address*. Las tres

operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

atomicCAS()

int atomicCAS(int* address, int compare, int val);

unsigned int atomicCAS(unsigned int* address, unsigned int compare, unsigned int val);

unsigned long long int atomicCAS(unsigned long long int* address, unsigned long long int compare, unsigned long long int val);

Lee el contenido, palabra de 32-bit,apuntado por *address* ubicado en la memoria global o compartida, computa ((contenido==*compare)? val:* contenido*)*, y, finalmente, almacena el resultado en *address*. Las tres operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

Funciones Bit-wise

atomicAnd()

int atomicAnd(int* address, int val);

unsigned int atomicAnd(unsigned int* address, unsigned int val);

Lee el contenido, palabra de 32-bit,apuntado por *address* ubicado en la memoria global o compartida, computa (contenido *&val)*, y, finalmente, almacena el resultado en *address*. Las tres operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

atomicOr()

int atomicOr(int* address, int val);

unsigned int atomicOr(unsigned int* address, unsigned int val);

Lee el contenido, palabra de 32-bit,apuntado por *address* ubicado en la memoria global o compartida, computa (contenido | *val)*, y, finalmente, almacena el resultado en *address*. Las tres operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

atomicXor()

int atomicXor(int* address, int val);

unsigned int atomicXor(unsigned int* address, unsigned int val);

Lee el contenido, palabra de 32-bit,apuntado por *address* ubicado en la memoria global o compartida, computa (contenido *val)*, y, finalmente, almacena el resultado en *address*. Las tres operaciones son hechas en una transacción atómica. La función retorna el contenido apuntado por *address*.

APÉNDICE B

Funciones adicionales para programas CUDA

En este apéndice se detallan algunas de las funciones de la biblioteca cutil de gran ayuda para realizar ciertas operaciones como medir el tiempo, chequeo de errores, finalización de la ejecución, etc.
Es una biblioteca incluida en el kit de desarrollo del software (SDK), no es propia de CUDA, fue desarrollada para reducir el código de los ejemplos, no es robusta.

B.1. Funciones para el reporte de Errores

cutilExit

cutilExit(int argc, char **argv)
Comprueba si la finalización del programa fue correcta. En caso de ser correcta muestra en pantalla la leyenda "Press ENTER to exit...".

cutilSafeCall

cutilSafeCall(función);
Se utiliza para chequear si la llamada a *función* es segura.

cutilCheckMsg

cutilCheckMsg(function);
Se utiliza para la invocación de la función *kernelfuntion*. Comprueba si hay mensajes de error del *kernel*.

cutilCheckError

cutilCheckError(función);
Reporta errores en la invocación de *función*, la cual es una función de SDK

B.2. Funciones para medir el Tiempo

cutCreateTimer

cutCreateTimer(unsigned int* name)

Crea un nuevo reloj cuyo nombre es *name*. Retorna true si tuvo éxito, de otra manera false.

cutDeleteTimer

cutDeleteTimer(unsigned int name)

Elimina el reloj *name*. Retorna true si la eliminación fue exitosa, sino false.

cutStartTimer

cutStartTimer(const unsigned int name)

Inicia el conteo del tiempo en el reloj *name*.

cutStopTimer

cutStopTimer(const unsigned int name)

Detiene el conteo del tiempo del reloj *name*. No lo vuelve a iniciar.

cutResetTimer

cutResetTimer(const unsigned int name)

Reinicia el contador del reloj *name*.

cutGetAverageTimerValue

cutGetAverageTimerValue(const unsigned int name)

Retorna el tiempo promedio del reloj *name* como el tiempo total del reloj dividido por el número de ejecuciones completas. Excluye el tiempo de la ejecución actual si el reloj está andando.

cutGetTimerValue

cutGetTimerValue(const unsigned int name)

Retorna el tiempo del reloj *name* transcurrido desde la creación del reloj o de su reiniciación.

APÉNDICE C

Modelos de GPU y Capacidades de Cómputo

En la descripción de cada una de las generaciones de GPU, estas tienen distintas características, las cuales determinan sus ventajas y limitaciones. Esto hizo que se implementara un sistema estandarizado para la descripción de las capacidades CUDA de cada GPU. Este sistema se describe a través de las capacidades de hardware. Las especificaciones generales y las características de un dispositivo de cálculo dependerán de su capacidad de computación.
En este apéndice se detallan cada una de las capacidades de cómputo CUDA y los modelos de GPU de Nvidia que las poseen. Para más detalles ver (NVIDIA, CUDA GPU, 2011).

C.1. Capacidades de las Arquitecturas

La descripción de capacidades están íntimamente ligadas a CUDA, estas se inician con la capacidad 1.0, teniendo hasta el momento las capacidades 1.1, 1.2, 1.3, 2.0 y 2.1. Una nueva capacidad indica una nueva generación de dispositivos, el cual tiene las capacidades de la anterior generación y otras nuevas.
El nombre de la capacidad está dado por dos dígitos, el primero se corresponde con la revisión principal y el segundo con una versión secundaria. Capacidades con el mismo número de revisión principal indica que tienen la misma arquitectura. El cambio de este indicador lo dio la serie de GPU con arquitectura Fermi, las cuales tienen capacidad *2.x*. Toda arquitectura anterior tiene capacidad *1.x*. El número de versión secundaria se corresponde con una mejora de la arquitectura básica, posiblemente a través de la inclusión de nuevas características.
Las características básicas de cada capacidad son:

Capacidad 1.0: Permite todas las operaciones básicas definidas para las arquitecturas a partir de la G80. Constituye el punto de partida de las capacidades de cómputo provistas por la GPU y a partir de las cuales se realizarán las futuras extensiones.

Capacidad 1.1: Incluye el soporte para las funciones atómicas sobre memoria global y para palabras de 32 bits.

Capacidad 1.2: En estas arquitecturas se permiten funciones atómicas sobre memoria compartida, funciones atómicas sobre memoria global para palabras de 64 bits y funciones de voto de *warps*.

Capacidad 1.3: Permite las operaciones sobre números de punto flotante de doble precisión.

Capacidad 2.0: Incluye el soporte para *grid* de tres dimensiones y operaciones de suma atómica de números de punto flotante sobre la memoria compartida y global para palabras de 32 bits. Además incluye distintas alternativas de sincronización de barreras (todas ellas implican la evaluación de condiciones), otras funciones de voto en los *warps* y funciones de superficie.

En la siguiente tabla se resumen las características soportadas por cada una de las capacidades, recuerde que toda nueva capacidad incluye las anteriores.

<table>
<tr><th></th><th colspan="5">Capacidad de Cómputo</th></tr>
<tr><th>Características soportadas</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th></tr>
<tr><td>Grid 3D</td><td colspan="4">No</td><td>Si</td></tr>
<tr><td>Funciones atómicas operando sobre la memoria global con palabras de 32 bit.</td><td>No</td><td colspan="4">Si</td></tr>
<tr><td>Funciones atómicas de enteros sobre la memoria global con palabras de 64 bits.</td><td colspan="2" rowspan="3">No</td><td colspan="3" rowspan="3">Si</td></tr>
<tr><td>Funciones atómicas de enteros sobre la memoria compartida con palabras de 32 bits.</td></tr>
<tr><td>Funciones de voto de warp.</td></tr>
<tr><td>Números de punto flotante de doble precisión.</td><td colspan="3">No</td><td colspan="2">Si</td></tr>
<tr><td>Operaciones de suma atómica de números de punto flotante sobre la memoria compartida y global para palabras de 32 bits.</td><td colspan="4" rowspan="3">No</td><td rowspan="3">Si</td></tr>
<tr><td>__ballot()</td></tr>
<tr><td>__threadfence_system()</td></tr>
</table>

__syncthreads_count(), __syncthreads_and(), __syncthreads_or()		
Funciones de Superficie		

En la siguiente tabla se muestran las especificaciones técnicas de cada una de las capacidades de cómputo.

	Capacidad de Cómputo				
Especificaciones Técnicas	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Máxima dimensión de un *grid*.	2				3
Máximo de la dimensión *x, y* y *z* de un *grid*.	65535				
Máxima cantidad de dimensiones de los bloques de *threads*.	3				
Máximo de la dimensión x e y de un bloque de *threads*.	512				1024
Máximo de la dimensión z en un bloque.	64				
Máximo número de *threads* por bloques.	512				1024
Tamaño de un *warp*	32				
Máximo número de bloques residentes en un SM.	8				
Máximo número de *warps* residentes por SM.	24		32		48
Máximo número de *threads* residentes por SM.	768		1024		1536
Número de registros de 32-bit por SM.	8 K		16 K		32 K
Máxima cantidad de memoria compartida por SM.	16 KB				48 KB
Número de bancos de memoria compartida.	16				32
Cantidad de memoria local por *thread*.	16 KB				512 KB
Tamaño de la memoria constante.	64 KB				
Cache por SM para memoria constante.	8 KB				
Cache por SM para memoria de textura.	Depende del Dispositivo, entre 6KB y 8KB				
Máximo ancho para una referencia de textura de 1D limitado a un arreglo CUDA.	8192				32768
Máximo ancho para una referencia de textura 1D limitado a memoria lineal.	227				
Máximo ancho y número de capaz para una para una referencia de textura 1D por capas.	8192 x 512				16384 x 2048
Máximo ancho y alto para una referencia de textura 2D limitado a memoria lineal o a un arreglo CUDA.	65536 x 32768				65536 x 65535

Máximo ancho, alto y número de capas para una referencia de textura 2D por capas.	8192 x 8192 x 512	16384 x 16384 x 2048
Máximo ancho, alto y profundidad para referencias de textura 3D limitado a un arreglo CUDA.	2048 x 2048 x 2048	
Máximo número de texturas, las cuales pueden ser limitado a un kernel.	128	
Máximo ancho para una referencia de superficie 1D limitado a un arreglo CUDA.	N/A	8192
Máximo ancho y alto para una referencia de superficie 2D limitado a un arreglo CUDA.		8192 x 8192
Máximo número de superficies, las cuales pueden ser limitadas a un kernel		8
Máximo número de instrucciones por kernel	2 millones	

Conocer las capacidades de cómputo de la arquitectura de la GPU permite el desarrollo de software orientado a tomar la máxima ventaja de la arquitectura subyacente.

C.2. GPU y sus capacidades

Aquí se especifican las capacidades de cómputo para cada modelo de GPU.

Capacidad 1.0

Tipo	GPU
Tesla	Tesla C870, Tesla D870, Tesla S870
Quadro	Quadro FX 5600, Quadro FX 4600, Quadro Plex 2100 S4
GeForce Desktop	GeForce 8800 Ultra, GeForce 8800 GTX, GeForce GT 340*, GeForce GT 330*, GeForce GT 320*, GeForce 315*, GeForce 310*, GeForce 9800 GT, GeForce 9600 GT, GeForce 9400GT, GeForce GT 440, GeForce GT 440*, GeForce GT 430, GeForce GT 430*, GeForce GT 420*

Capacidad de Cómputo 1.1

Tipo	GPU
Quadro	Quadro FX 4700 X2, Quadro FX 3700, Quadro FX 1800, Quadro FX 1700, Quadro FX 580, Quadro FX 570, Quadro FX 470, Quadro FX 380, Quadro FX 370, Quadro FX 370 Low Profile, Quadro NVS 450, Quadro NVS 420, Quadro NVS 295, Quadro Plex 2100 D4

Quadro Mobile	Quadro FX 3800M, Quadro FX 3700M, Quadro FX 3600M, Quadro FX 2800M, Quadro FX 2700M, Quadro FX 1700M, Quadro FX 1600M, Quadro FX 770M, Quadro FX 570M, Quadro FX 370M, Quadro FX 360M, Quadro NVS 320M, Quadro NVS 160M, Quadro NVS 150M, Quadro NVS 140M, Quadro NVS 135M, Quadro NVS 130M
NVS Desktop	Quadro NVS 450, Quadro NVS 420, Quadro NVS 295
GeForce Desktop	GeForce GTS 250,GeForce GTS 150, GeForce GT 130*, GeForce GT 120*, GeForce G100*, GeForce 9800 GX2, GeForce 9800 GTX+, GeForce 9800 GTX, GeForce 9600 GSO, GeForce 9500 GT, GeForce 8800 GTS, GeForce 8800 GT, GeForce 8800 GS, GeForce 8600 GTS, GeForce 8600 GT, GeForce 8500 GT, GeForce 8400 GS, GeForce 9400 mGPU, GeForce 9300 mGPU, GeForce 8300 mGPU, GeForce 8200 mGPU, GeForce 8100 mGPU
GeForce Notebook	GeForce GTX 285M, GeForce GTX 280M, GeForce GTX 260M, GeForce 9800M GTX, GeForce 8800M GTX, GeForce GTS 260M, GeForce GTS 250M, GeForce 9800M GT, GeForce 9600M GT, GeForce 8800M GTS, GeForce 9800M GTS, GeForce GT 230M, GeForce 9700M GT, GeForce 9650M GS, GeForce 9700M GT, GeForce 9650M GS, GeForce 9600M GT, GeForce 9600M GS, GeForce 9500M GS, GeForce 8700M GT, GeForce 8600M GT, GeForce 8600M GS, GeForce 9500M G, GeForce 9300M G, GeForce 8400M GS, GeForce G210M, GeForce G110M, GeForce 9300M GS, GeForce 9200M GS, GeForce 9100M G, GeForce 8400M GT, GeForce G105M

Capacidad de Cómputo 1.2

Tipo	**GPU**
Quadro	Quadro FX 380 Low Profile, NVIDIA NVS 300
Quadro Mobile	Quadro FX 1800M, Quadro FX 880M, Quadro FX 380M
NVS Desktop	NVIDIA NVS 300
NVS Mobile	NVS 5100M, NVS 3100M, NVS 2100M
GeForce Desktop	GeForce GT 520, GeForce GT 240, GeForce GT 220*, GeForce 210*
GeForce Notebook	GeForce GTS 360M, GeForce GTS 350M, GeForce GT 335M, GeForce GT 330M, GeForce GT 325M, GeForce GT 240M, GeForce G210M, GeForce 310M, GeForce 305M

Capacidad de Cómputo 1.3

Tipo	**GPU**
Tesla	Tesla C1060, Tesla S1070, Tesla M1060
Quadro	Quadro FX 5800, Quadro FX 4800, Quadro FX 4800 for Mac, Quadro FX 3800, Quadro CX, Quadro Plex 2200 D2
GeForce Desktop	GeForce GTX 295, GeForce GTX 285, GeForce GTX 285 for Mac, GeForce GTX 280, GeForce GTX 275, GeForce GTX 260

Capacidad de Cómputo 2.0

Tipo	GPU
Tesla	Tesla C2050/C2070, Tesla S2050, Tesla M2050/M2070/M2090
Quadro	Quadro 6000, Quadro 5000, Quadro 4000, Quadro 4000 for Mac, Quadro 2000, Quadro 2000D, Quadro 600, Quadro Plex 7000
Quadro Mobile	Quadro 5010M, Quadro 5000M, Quadro 4000M, Quadro 3000M, Quadro 2000M, Quadro 1000M
GeForce Desktop	GeForce GTX 590, GeForce GTX 580, GeForce GTX 570, GeForce GTX 480, GeForce GTX 470, GeForce GTX 465
GeForce Notebook	GeForce GTX 480M

Capacidad de Cómputo 2.1

Tipo	GPU
NVS Mobile	NVS 4200M
GeForce Desktop	GeForce GTX 560 Ti, GeForce GTX 550 Ti, GeForce GTX 460, GeForce GTS 450, GeForce GTS 450*
GeForce Notebook	GeForce GT 580M, GeForce GT 570M, GeForce GT 560M, GeForce GT 555M, GeForce GT 550M, GeForce GT 540M, GeForce GT 525M, GeForce GT 520MX, GeForce GT 520M, GeForce GTX 485M, GeForce GTX 470M, GeForce GTX 460M, GeForce GT 445M, GeForce GT 435M, GeForce GT 420M, GeForce GT 415M

Bibliografía

[1] S. Akl. (1989). *The Design and Analysis of Parallel Algorithms.* Prentice- Hall.

[2] E. G. Ayguadé. (1999). "Nanos Compiler. A Reasearch Platform for OpenMP. Extensions". *In First European Workshop on OpenMP.*

[3] E. M. Ayguadé. (1999). "Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study". *InThe 1999 International Conference on Parallel Processing.* Aizu (Japan).

[4] G. Blelloch. (1996). "Programming Parallel Algorithms". *InCommunications of ACM, 39*(3).

[5] D. Blythe. (2006). "The Direct3D 10 system". *InACM Trans. Graph., 25*(3), 724–734.

[6] H. H. Bal. (1998). "Approaches for integrating task and data parallelism". *InIEEE Concurrency, 6*(3), 74-84.

[7] H. K. Bal. (1992). "Orca: a language for parallel programming of distributed systems". *InIEEE Transactions on Software Engineering, 18*(3), 190–205.

[8] S. B. Ben Hanssen. (1998). "A task and data parallel programming language based on shared objects". *InACM Transactions Languages Systems*.

[9] O. A. Board. (2000). *OpenMP Specifications: FORTRAN 2.0.*Obtenido de www.openmp.org/specs/mp-documents/fspec20.ps.

[10] O. A. Board. (1997). "OpenMP: A Proposed Industry Standard API for Shared Programming". *InOpenMP Architecture. Review Board.* <www.openmp.org>.

[11] I. F. Buck. (2004). "Brook for GPUs: Stream computing on graphics hardware". *InACM Trans. Graph, 23*(3), 777-786.

[12] A. G. Burks. (1946). *Prelimirary discussion of the logical desin of an electric computing instrument. Part 1. Technical report, T.* (T. U. Department, Ed.) Princeton: The Institute of Advanced Study.

[13] M. W. Campione. (1998). *The Java Tutorial 2nd edition.* Addison Wesley.

[14] B. M. Chapman. (1994). "A software architecture for multidisciplinary applications: Integrating task and data parallelism". *InCONPAR.*

[15] B. M. Chapman. (1998). "OpenMP and HPF: Integrating two paradigms". *InEuro-Par'98 Conference.Springer Verlag LNCS*.

[16] V. Carlini. (1991). *Transputers and Parallel Architectures.* Ellis Horowood Limited.

[17] M. Flynn. (1995). *Computer Arquitecture Pipelined and Parallel Processor Design.* Jones and Bartlett.

[18] M. R. Flynn. (1996). “Parallel Arquitecture”. *InACM Computing Surveys, 28*(1), 67-70.
[19] I. Foster. (1994). *Designing and Building Parallel Programs.* Addison-Wesley.
[20] R. Glidden. (1997). *Graphics programming with Direct3D: techniques and Concepts.* Addison-Wesley Developers Press.
[21] W. Gehrke. (1996). *Fortran 95 language guide.* Springer.
[22] A. Geist. (1994). *PVM: parallel virtual machine:a users' guide and tutorial for networked parallel computing Scientific and engineering computation.* MIT Press.
[23] A. Godse. (2009). *Computer Graphics.* Technical Publications.
[24] J. L. Gonzalez. (2000). “Supporting Nested Parallelism”. *InVI Congreso Argentino de Ciencias de la Computación.* Ushuaia. Argentina. Universidad de Tierra de Fuego.
[25] J. L. Gonzalez. (2000). “Toward Standard Nested Parallelism”. *In7mo Euro PVM/MPI Users Groups Meeting: Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Balatonfured. Hungry: Springer. 96-103.
[26] W. Gropp. (1998). *The MPI-2 extensions. The complete reference* (Vol. 2). MIT Press.
[27] J. Hardwick. (1996). “An Efficient Implementation of nested data Parallelism for Irregular Divide and Conquer Algorithms”. *InFirst International Workshop on High programming Models and Supportive Environments.*
[28] P. Harrigan. (2004). *First person: new media as story, performance, and game* . MIT Press.
[29] M. Harris. (2005). “Mapping computational Concepts to GPUs”. *InGPU Gems 2*. Addison-Wesley. 493-508.
[30] J. Hensley. (2007). “AMD CTM overview”. *InACM SIGGRAPH*
[31] K. Hwang. (1993). *Advanced computer architecture: parallelism, scalability, programmability.* McGraw-Hill.
[32] W. W. Hwu. (July/August de 2008). “The concurrency challenge”. *InIEEE Design and Test of Computers*, 312–320.
[33] B. S. Jacob. (2007). *Memory systems: cache, DRAM, disk.* Morgan Kaufmann.
[34] D. T. Jacobsen. (2010). “An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters”. *InInanc Senocak AIAA Aerospace Sciences Meeting proceedings.*
[35] J. K. Keller. (2001). *Practical PRAM programming.* John Wiley & Sons inc.
[36] B. R. Kernighan. (1978). *The C programming language.* Prentice-Hall.

[37] D. H. Kirk. (2010). *Programming Massively Parallel Processors A Hands-on Approach.* Elseiver- Morgan Kaufmann.
[38] C. Koelbel. (1994). *The High performance Fortran handbook . Scientific and engineering computation .* MIT Press.
[39] M. M. Leair. (2000). "Distributed OMP - a programming model for SMP clusters". *In8th Workshop on Compilers for Parallel Computers*, 229–238.
[40] C. Leopold. (2001). *Parallel and Distributed Computing: A survey of models, paradigms, and approaches.* John Wiley & Sons inc.
[41] E. N. Lindholm. (2008). "NVIDIA TESLA: A UNIFIED GRAPHICS AND COMPUTING ARCHITECTURE". *InIEEE Computer Society*, 39-45.
[42] M. D. McCool. (2004). "Shader algebra". *InACM Trans. Graph, 23*(3), 787–795.
[43] M. McCool. (2006). "Data-parallel programming on the cell BE and the GPU using the RapidMind development platform". *InGSPx Multicore Applicat. Conf.*
[44] NVIDIA. (2010). "NVIDIA GF100. Whitepaper". *InWorld's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism*.
[45] NVIDIA. (2011). *CUDA GPU*. Recuperado el 2011, de CUDA GPU. <developer.nvidia.com/cuda-gpus>.
[46] NVIDIA. (2011). *NVIDIA CUDA C Programming Guide. Versión 4.0.* NVIDIA.
[47] NVIDIA. (2007). "The CUDA Compiler Driver NVCC".*InThe CUDA Compiler Driver NVCC* .
[48] S. A. Orlando. (1999). "COLTHPF a run-time support for the high-level co-ordination of HPF tasks". *InConcurrency: Practice and Experience, 11*(8), 407–434.
[49] P. Pacheco. (1997). *Parallel programming with MPI.* Morgan Kaufmann.
[50] D. H. Patterson. (2008). *Computer organization and design: the hardware/software interface* (Edition 4 ed.). Morgan Kaufmann.
[51] PeakStream. (s.f.). *The PeakStream platform: High productivity software development for multi-core processors.* Obtenido de <www.peakstreaminc.com/reference/
peakstream_platform_technote.pdf>.
[52] S. J. Pennycook. (2011). "Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark". *InSIGMETRICS Perform. Eval. Rev., 38*(4), 23-29.
[53] M. Quinn. (1994). *Parallel Computing. Theory and Practice. Second edition.* McGraw-Hill. Inc.

[54] M. Quinn. (2004). *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill.
[55] M. B. Root. (1999). *DirectX complete. Complete Series.*Ed. McGraw-Hill.
[56] S. R. Ryoo. (2008). "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA". *In13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Salt Lake: ACM. 73-82.
[57] R. Schreiber. (1999). "High Performance Fortram, version 2". *InParallel Processing Letters, 7*(4), 437–449.
[58] E. K. Sanders. (2010). *CUDA by Example, An Introduction to General Purpose GPU Programming.* Addison-Weslley.
[59] G. B. Satir. (1995). *C++: the core language.* O'Reilly Media, Inc.
[60] D. Shreiner. (2004). *OpenGL programming guide: the official guide to learning OpenGL* (4 ed., Vol. 2003). Addison-Wesley.
[61] J. A. Subhlok. (1997). "A New model for integrated nested task and data parallel programming". *InACM SigplanSymposium Principles and Practice of Parallel Programming.* ACM press.
[62] Supercomputer, T. 5. (2011). *Top 500 Supercomputer*. Recuperado el 2011, de <www.top500.org/list/2010/11/100>.
[63] H. L. Sutter. (2005). "Software and the concurrency revolution". *InACM Queue, 3*(7), 54-62.
[64] P. Takis. (1995). "Fundamental Ideas for a Parallel Computing Course". *InACM Computing Surveys, 27*(2), 284–286.
[65] D. P. Tarditi. (2006). "Accelerator: Using data-parallelism to program GPUs for general-purpose uses". *InProc. 12th Int. Conf. Architect. Support Program. Lang. Oper. Syst.* 325–335.
[66] P. Walsh. (2008). *Advanced 3D game programming with DirectX 10.0.* Ed. Jones & Bartlett Learning.
[67] B. Wilkinson. (1996). *Computer Arquitecture Design and Performance, 2nd ed.* London: Prentice Hall.
[68] B. Wilkinson. (1999). *Parallel Programming: Tecniques and Aplications using Networked Workstations and Parallel Computers.* New Jersey: Prentice Hall.
[69] R. L. Wright. (2007). *OpenGL superbible: comprehensive tutorial and referente OpenGL Series.* (Edition 4. ed.). Addison-Wesley.

Esta publicación se terminó de imprimir
en el mes de octubre de 2011,
en la ciudad de La Plata,
Buenos Aires,
Argentina.