Ministerio de Educación Superior

Universidad Central "Marta Abreu" de Las Villas

Facultad de Matemática, Física y Computación

Ingeniería Informática

Trabajo de Diploma

Acercamiento a la programación paralela usando CUDA

***Autora**: Diana Isabel Triana Brito*

***Tutor**: Dr. Daniel Gálvez Lio*

Santa Clara, 2012

Declaración Jurada

La que suscribe, *Diana Isabel Triana Brito* hago constar que el trabajo titulado *Acercamiento a la programación paralela usando CUDA* fue realizado en la Universidad Central “Marta Abreu” de Las Villas como parte de la culminación de los estudios de la especialidad de *Ingeniería Informática*, autorizando a que el mismo sea utilizado por la institución, para los fines que estime conveniente, Tanto de forma parcial como total y que además no podrá ser presentado en eventos ni publicado sin la autorización de la Universidad.

_____

Firma del autor

Los abajo firmantes, certificamos que el presente trabajo ha sido realizado según acuerdos de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

_____ _____

Firma del tutor Firma del jefe del Laboratorio

_____

Fecha

Dedicatoria

A mi madre, Delma y abuelos, Briseida y Manolo por su amor, apoyo y consejos.

A mi hermana, Elizabeth por su dedicación y apoyo incondicional.

A mi esposo por su amor, comprensión y confianza.

A todos los que me han apoyado.

Agradecimientos

A mi tutor Gálvez por su guía y profesionalidad.

A Mayito por su colaboración y decisión a ayudar.

A mis contactos en la Red UCLV en especial a Miriel.

A todos mis compañeros y conocidos que me desearon éxitos.

Resumen

Debido al uso cada vez más extendido de aceleradores de hardware en entornos de computación de altas prestaciones ha surgido una nueva tecnología de la programación en paralelo: CUDA (*Compute Unified Device Architecture*) un nuevo hardware y arquitectura de software para la emisión y gestión de cálculos en la GPU (*Graphics Processor Unit*) como un dispositivo de computación de datos en paralelo. Las ventajas de CUDA frente al empleo de otras arquitecturas clásicas son numerosas, el bajo coste de los dispositivos para ejecutar estos programas, el relativo poco esfuerzo para adaptar los algoritmos a esta filosofía de trabajo y la alta rentabilidad en tiempo de ejecución que se consigue en numerosos casos. En este trabajo se presentan las características de la tecnología, aplicaciones que usan esta arquitectura, se establece un procedimiento de trabajo y se prueban ejemplos en la solución de problemas haciendo uso de la tecnología.

Abstract

Due to the extended use of hardware accelerators in computer environments of high benefits, a new technology of the programming in parallel has arisen: CUDA (*Computes Unified Device Architecture*), a new hardware and software architecture for the emission and administration of calculations in the GPU (*Graphics Processor Unit*) as a computer device of data in parallel. The advantages of CUDA are numerous in front of the employment of other classic architectures. Between them we can find, the first floor cost of the devices to execute these programs, the relative little effort to adapt the algorithms to this working philosophy and the high profitability in run-time that is gotten in numerous cases. In this work the characteristics of the technology applications that use this architecture are presented. Therefore a work procedure and the troubleshooting making use of the technology are establish.

Contenido

Índice de figuras

Introducción

En cuestión de pocos años, la unidad de procesamiento gráfico (GPU) se ha convertido en una gran potencia de la computación, las GPU actuales ofrecen increíbles recursos, tanto para procesamientos gráficos y no gráficos (NVIDIA_Corporation, 2011).

La razón principal detrás de esta evolución es que la GPU está especializada en cálculo intensivo, computación altamente paralela, exactamente lo que se trata es de la representación de gráficos y por lo tanto, está diseñada de tal manera que más transistores se dedican al procesamiento de datos en lugar de almacenamiento en caché y control de flujo (NVIDIA_Corporation, 2011). Específicamente, la GPU es especialmente adecuada para abordar los problemas que se pueden expresar como cálculos de datos paralelos, el mismo programa se ejecuta en varios hilos en paralelo que acceden a diferentes datos, con una intensidad aritmética alta.

Debido a que el mismo programa se ejecuta para cada elemento de datos, hay un menor requerimiento de control de flujo y con dicha capacidad aritmética la latencia de acceso a la memoria se puede ocultar con cálculos en lugar de caché de datos grandes (NVIDIA_Corporation, 2011).

Muchas aplicaciones que procesan grandes conjuntos de datos, tales como *arrays* pueden utilizar un modelo de programación de datos en paralelo para acelerar los cálculos. En 3D de grandes conjuntos de píxeles y vértices se asignan a hilos en paralelo. Del mismo modo, la imagen y los medios de procesamiento de aplicaciones tales como post-procesamiento de las imágenes renderizadas, codificación y decodificación de vídeo, escalado de imagen, la visión estéreo, y el reconocimiento de patrones se pueden asignar bloques de píxeles de la imagen y en paralelo a los hilos del procesamiento ((Valle, 2011, Zeller, 2008)). De hecho, muchos algoritmos fuera del campo de representación de imágenes y procesamiento son acelerados por datos de procesamiento paralelo, desde el procesamiento de señales en general o simulación de la física, de las finanzas computacionales o biología computacional ((Caballero de Gea, 2011, Cheng et al., 2006)).

Con la masificación del uso de las GPU por NVIDIA en sus tarjetas de video se pone a disposición de los usuarios un nuevo recurso computacional que puede ser

aprovechado por los programadores para mejorar el rendimiento de las aplicaciones de alto costo computacional, a través de CUDA: una arquitectura de programación paralela y un modelo de programación de la GPU en un lenguaje de alto nivel. El auge que ha ido alcanzando esta tecnología y su accesibilidad hace necesaria la exploración y su asimilación para evaluar su utilización en los diferentes campos de investigación.

El **objetivo general** de este trabajo:

- Explorar las facilidades de CUDA en el acceso a los recursos de las tarjetas gráficas NVIDIA para escribir programas paralelos.

Los **objetivos específicos**:

- Presentar la tecnología CUDA y sus aplicaciones actuales.
- Caracterizar la programación paralela sobre las GPU presentes en las tarjetas gráficas NVIDIA usando CUDA.
- Establecer un procedimiento de trabajo para el uso de CUDA en la paralelización de programas.
- Aplicar el procedimiento y la tecnología CUDA en los ejemplos que dan solución a algunos problemas.

El trabajo se justifica a partir de que puede constituir un referente metodológico y práctico para los especialistas de esta temática de la programación en paralelo, pues se reúne toda la información y se establece un procedimiento para el uso de una nueva arquitectura de programación (CUDA).

Este informe escrito consta de tres capítulos, conclusiones, recomendaciones, referencias bibliográficas y anexos. En el primer capítulo se realiza un análisis de las características principales de la computación paralela, las GPU, la nueva arquitectura CUDA y algunas aplicaciones que usan esta tecnología. En el capítulo dos se aborda la implementación del hardware, la API (*Application Programming Interface*) y se establece un procedimiento de trabajo que sirve de apoyo para el uso de CUDA. En el capítulo tres se brindan los mecanismos para la instalación del software de CUDA y la forma de creación de un nuevo proyecto CUDA usando VisualStudio, además algunas especificaciones generales y finalmente la práctica de varios ejemplos.

Capítulo 1. CUDA una nueva arquitectura de computación paralela en la GPU y sus aplicaciones

Partiendo del análisis de la computación paralela en la CPU (*Central Processing Unit*) y la arquitectura de la GPU, a continuación se realiza una descripción detallada de la tecnología CUDA, la estructura que tiene el software, los componentes que la integran, ventajas, desventajas y el modelo de programación para llegar a conocer diferentes aplicaciones que utilizan la tecnología CUDA en su funcionamiento.

1.1 La computación paralela

La computación paralela es el empleo de la concurrencia para disminuir el tiempo de ejecución en la solución de problemas costosos computacionalmente y aumentar la dimensión del problema que puede resolverse. Presenta un problema que está dado por la dificultad en la programación de algoritmos (Almeida et al., 2008, Asanovic et al., 2006)

1.1.1 Costo de la computación paralela

El empleo de múltiples procesadores encarece el sistema computacional y la aceleración en el cálculo raramente es proporcional a la cantidad de procesadores que se añaden.

La programación es más compleja pues requiere de mecanismos de coordinación de las acciones desarrolladas por los distintos procesadores. La detección de errores de programación también es más complicada (Regis et al.).

Los programas escritos para una arquitectura paralela muy raramente pueden ser ejecutados de manera eficiente en otra arquitectura. Por tanto es necesario reescribir los programas para diferentes arquitecturas ((Asanovic et al., 2009, Hoeger, 1997).

1.1.2 Taxonomía de computadoras según la clasificación de Flynn

- *SISD (Single Instruction Single Data)*: un único flujo de instrucciones operando sobre un único flujo de datos.

- *SIMD (Single Instruction Multiple Data)*: Un único flujo de instrucciones operando sobre múltiples flujos de datos.
- *MISD (Multiple Instruction Single Data)*: Múltiples flujos de instrucciones operando sobre un único flujo de datos.
- *MIMD (Multiple Instruction Multiple Data)*: Múltiples flujos de instrucciones operando sobre múltiples flujos de datos.
  - *SPMD (Single Process Multiple Data)*.
  - *MPMD (Multiple Process Multiple Data)*.

1.1.3 Fuentes de paralelismo

- *Paralelismo de Control*: Proviene de la constatación de que una aplicación informática se compone de acciones que podemos "hacer al mismo tiempo". Se distinguen dos clases de dependencias, la de control de secuencia y control de comunicación.
- *Paralelismo de Datos*: Proviene de la constatación de que cierta aplicación trabaja con estructuras de datos muy regulares repitiendo una misma acción sobre cada elemento de la estructura.
- *Paralelismo de Flujo*: Proviene de la constatación de que ciertas aplicaciones funcionan en modo cadena, es decir, se dispone de un flujo de datos, generalmente semejantes, sobre los que debemos efectuar una sucesión de operaciones en cascada.

1.2 La Unidad de Procesamiento Gráfico (GPU)

La GPU no es más que una analogía de las CPU o microprocesadores clásicos (Rodriguez et al.) o sea es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos y o aplicaciones 3D interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la unidad central de procesamiento (CPU) puede dedicarse a otro tipo de cálculos como la inteligencia artificial o los cálculos mecánicos en el caso de los videojuegos (Astle and Hawking, 2004).

Una GPU implementa ciertas operaciones gráficas llamadas primitivas optimizadas para el procesamiento gráfico. Una de las primitivas más comunes para el procesamiento gráfico en 3D es el *antialiasing,* que suaviza los bordes de las figuras para darles un aspecto más realista (Blythe, 2006). Adicionalmente existen primitivas para dibujar rectángulos, triángulos, círculos y arcos. Las GPU actualmente disponen de gran cantidad de primitivas, buscando un mayor realismo en los efectos (Sengupta et al., 2007).

Si bien la denominación GPU hace referencia a una arquitectura especializada, dirigida específicamente al tratamiento gráfico, cada vez es mayor el número de aplicaciones que aprovechan la potencia de estos circuitos integrados para otro tipo de propósitos (Cheng et al., 2006, Santamaría et al., 2010).

Es aquí, precisamente, donde cobran protagonismo soluciones como Cg (*C for graphics*), ATI *Stream*, OpenGL(*Open Graphics Language*) y CUDA, infraestructuras compuestas de bibliotecas, compiladores y lenguajes que dan a luz a una nueva filosofía de desarrollo: GPGPU (*General Purpose computing on Graphics Processing Units*), la computación de propósito general usando GPU en lugar de CPU(Boggan and Pressel, 2007).

1.2.1 La Arquitectura de las GPU

Una GPU está altamente segmentada, lo que indica que posee gran cantidad de unidades funcionales. Estas unidades funcionales se pueden dividir principalmente en dos: aquéllas que procesan vértices, y aquéllas que procesan píxeles. Por tanto, se establecen el vértice y el píxel como las principales unidades que maneja la GPU (Engel, 2003).

Adicionalmente, y no con menos importancia, se encuentra la memoria. Esta se destaca por su rapidez, y juega un papel relevante a la hora de almacenar los resultados intermedios de las operaciones y las texturas que se utilicen.

Inicialmente, a la GPU le llega la información de la CPU en forma de vértices. El primer tratamiento que reciben estos vértices se realiza en el *vertex shader*. Aquí se realizan transformaciones como la rotación o el movimiento de las figuras. Luego, se

define la parte de estos vértices que se va a ver (*clipping*), y los vértices se transforman en píxeles mediante el proceso de rasterización. Estas etapas no poseen una carga relevante para la GPU. Donde sí se encuentra el principal cuello de botella del chip gráfico es en el paso siguiente: el *pixel shader*. Aquí se realizan las transformaciones referentes a los píxeles, tales como la aplicación de texturas. Cuando se ha realizado todo esto, y antes de almacenar los píxeles en la caché, se aplican algunos efectos como el *antialiasing*, *blending* y el efecto niebla. Otras unidades funcionales toman la información guardada en la caché y preparan los píxeles para su visualización. También pueden encargarse de aplicar algunos efectos. Luego se almacena la salida en el *frame buffer* con dos opciones: o tomar directamente estos píxeles para su representación en un monitor digital, o generar una señal analógica a partir de ellos, para monitores analógicos. En caso de ser este último, han de pasar por un DAC (*Digital-Analog Converter)*, para ser finalmente mostrados en pantalla.

La arquitectura de las GPU está formada por cuatro puntos fundamentales (Ortega et al., 2005): Procesador de hilo, multiprocesadores de apoyo (SMs, *Streaming Multiprocessor*), procesadores escalares (SPs, *Scalar Processor*) y Memoria (On-chip y Off-chip). La figura 2, tomada de (Rodriguez et al.) muestra esta arquitectura.

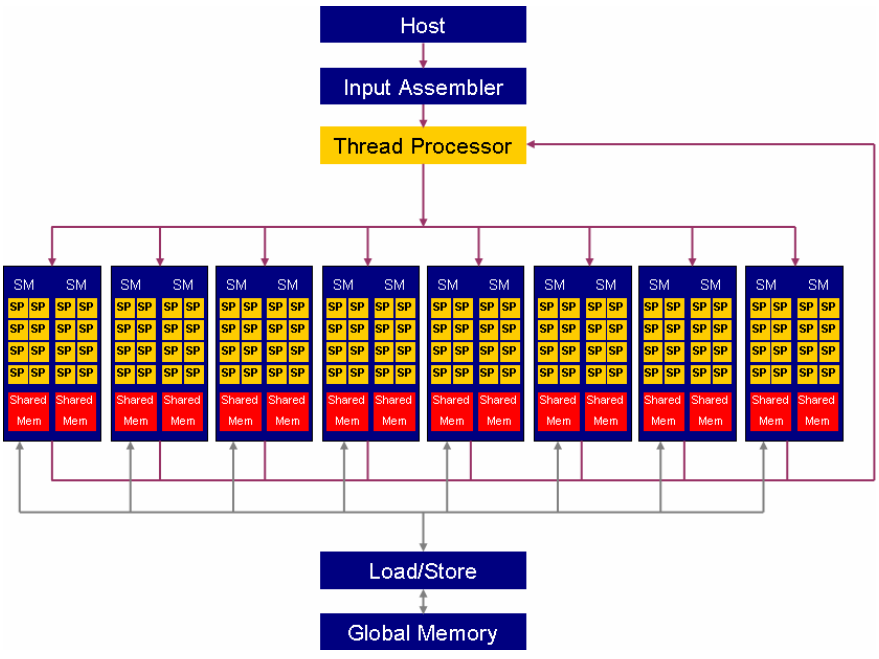


Figura 1. Arquitectura de la GPU

El procesador de hilo es el encargado de coordinar la ejecución de los hilos.

Cada multiprocesador de apoyo (SMs) contiene 8 procesadores escalares (SPs): 4, 14, 16, 30 SMs según la arquitectura (G80, G92,…., GT200).

- ✓ GeForce GTX 285: 30 SM x 8 SP = 240 Cores
- ✓ GeForce GTX 295: Dual 30 SM x 8 SP x 2 GPU = 480 Cores
- ✓ Tesla C1070: 30 SM x 8 SP x 4 GPU = 960 Cores
- ✓ Fermi (GT300): 16 SM x 32 SP = 512 Cores.

Los cores que conforman los procesadores escalares (SPs) contienen unidades funcionales en coma flotante de simple precision, 2 unidades en coma flotante de doble precisión IEEE 754-1985 y Fermi: 4 Unidades conforme estándar IEEE 754-2008.

En la Memoria On-chip se encuentran:

- Registros (R/W): Almacenan las variables declaradas en los *kernel* (núcleos), accesible por los hilos, con un tamaño de 16 K en registros de 32 bit
- Memoria compartida (R/W): Almacenan arreglos, compartida entre todos los hilos del bloque con un tamaño de 16 Kb (Fermi: 48Kb + 16Kb configurable)

En la Memoria Off-chip se encuentran:

- Memoria del dispositivo (R/W): Con tamaños de 512Mb, 1GB, 2GB, 4GB etc. Alta latencia entre 600-700 ciclos por acceso aleatorio, Accesibles por las mallas.
- Caché de texturas (R) 6Kb-8Kb por SM
- Caché constantes (R) 64Kb

1.2.2 La programación de las GPU

Al inicio, la programación de la GPU se realizaba con llamadas a servicios de interrupción de la BIOS. Luego la programación de la GPU se empezó a hacer en el lenguaje ensamblador específico a cada modelo. Posteriormente, se introdujo un nivel más entre el hardware y el software, con la creación de interfaces de programación de aplicaciones (API) específicas para gráficos, que proporcionaron

un lenguaje más homogéneo para los modelos existentes en el mercado. La primera API usada ampliamente fue OpenGL(McReynolds and Blythe, 2005), tras el cuál Microsoft desarrolló DirectX (Astle and Hawking, 2004, Blythe, 2006).

Con el desarrollo de API, se decidió crear un lenguaje más natural y cercano al programador, es decir, desarrollar lenguajes de alto nivel para gráficos. El lenguaje estándar de alto nivel, asociado a la biblioteca OpenGL es el GLSL (*OpenGL Shading Language)*, implementado en principio por todos los fabricantes.

NVIDIA creó un lenguaje propietario llamado Cg (*C for graphics*), con mejores resultados que GLSL en las pruebas de eficiencia. En colaboración con NVIDIA, Microsoft desarrolló (HLSL, *High Level Shading Language*), prácticamente idéntico a Cg, pero con ciertas incompatibilidades.

1.2.3 Desventajas de la GPU

Hasta ahora, sin embargo, el acceso a toda la potencia computacional en la GPU aprovechando las aplicaciones no gráficas presentan algunos inconvenientes (NVIDIA_Corporation, 2007)

La GPU sólo puede ser programada a través de una API de gráficos, la imposición de una alta curva de aprendizaje para el principiante y la sobrecarga de una API inadecuada para las aplicaciones no gráficas.

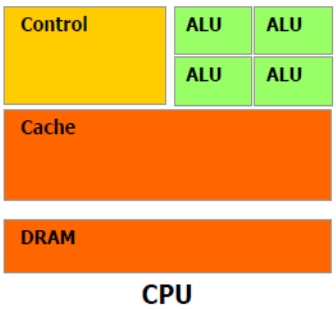
La DRAM de la GPU puede ser leída de manera general, pero no se puede escribir, la eliminación de una gran cantidad de programación flexibiliza la disponibilidad en la CPU.

Algunas aplicaciones son un cuello de botella en el ancho de banda de memoria DRAM, con una baja utilización de la potencia de las GPU.

1.2.4 Diferencias entre CPU y GPU

Las actuales CPU son multiprocesadores y son capaces de ejecutar dos hilos de manera simultánea (dos por núcleo en algunos casos), las GPU más avanzadas disponen de hasta 1024 núcleos de procesamiento y tienen capacidad para ejecutar hasta 128 hilos por procesador, lo que ofrece un total de hilos muy superior. Sistemas compuestos únicamente de ocho tarjetas de vídeo de este tipo están

superando, en cuanto a rendimiento se refiere, a clústeres de ordenadores como Blue Gene, formados por 512 nodos con CPU clásicas (Rodriguez et al.). La razón de estas diferencias entre la capacidad de la CPU y la GPU, es que la GPU está diseñada para realizar computación-intensiva, altamente paralela y por ello está dotada de mayor cantidad de transistores que se dedican al procesamiento de datos en las unidades lógicas aritméticas (ALU, *Arithmetic Logic Unit*) en lugar de almacenar datos en caché o controlar el flujo de información. Esto quiere decir que las GPU están especialmente diseñadas para llevar a cabo gran cantidad de operaciones en paralelo. Puesto que cada elemento de procesamiento posee sus propias localidades de memoria, no se requiere habilitar un control de flujo sofisticado. Además, la latencia por accesos a memoria se disminuye. En la figura 1 se muestra estas diferencias.



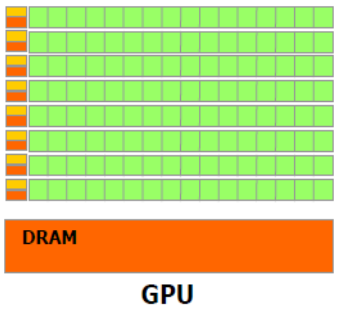


Figura 2 Comparación entre CPU y GPU

1.3 La tecnología CUDA

CUDA fue creado en el 2006 por NVIDIA, hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en las GPU de NVIDIA. Por medio de otras interfaces se puede usar Python, Fortran y Java en vez de C/C++ y en el futuro también se añadirá OpenGL y Direct3D. CUDA intenta explotar las ventajas de las GPU frente a las CPU de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos.

El primer SDK se publicó en febrero de 2007 en un principio para Windows, Linux, y más adelante en su versión 2.0 para Mac OS. Actualmente se ofrece la versión 4.2

para Windows XP, Windows Vista, Windows 7, Linux y Mac OS ya sean de 32/64 bits.

La tecnología CUDA de NVIDIA, convierte los procesadores gráficos (GPU) en una nueva arquitectura de computadoras de propósito general (GPGPU) incrementando hasta 400 veces la velocidad de acceso a la información (Mora, 2008).

NVIDIA presentó en un tour europeo su software CUDA, tecnología capaz de convertir cualquier tarjeta gráfica en un procesador. Hasta la fecha, las tarjetas gráficas se han utilizado para el tratamiento de imágenes en juegos, en programas relacionados con el mundo gráfico tanto en 2D como en 3D, procesos que precisan de este tipo de hardware de alto nivel para tratar las imágenes con una resolución muy alta y a la vez visualizar vídeos de alta definición con soportes en DVD o *Blu-ray.*

El uso de CUDA en cualquier ordenador puede convertirlo en un "superordenador" ya que potenciaría de forma importante los numerosos procesadores que están alojados en el núcleo de su tarjeta gráfica. El hecho de que los datos de proceso se realicen en paralelo hace que la velocidad de ejecución sea 400 veces superior a la de un ordenador sin este software, es habitual emplear CUDA para utilizar las GPU de NVIDIA como coprocesador para acelerar ciertas partes de un programa, generalmente aquellas con una elevada carga computacional por hilos, ya que es en este tipo de cálculos donde más rendimiento se suele obtener (Peña et al., Schwarz and Stamminger, 2009).

El avance que NVIDIA presentó en el Centro de Supercomputación de Barcelona (BSC), de mano de su creador David Kirk, jefe científico de NVIDIA, se centró en una conferencia sobre los resultados de la investigación científica sobre cómo CUDA puede contribuir a mejorar los procesos de los superordenadores Fuente: Diario Ti. Según el Dr. David Anderson, Científico Investigador del Laboratorio de Ciencias Espaciales Berkeley de la Universidad de California y fundador de la BOINC, "*la tecnología CUDA de NVIDIA abre las puertas a una nueva potencia de procesamiento en la investigación científica que antes era inexistente y que los investigadores no podían permitirse*." También añadió que la tecnología CUDA facilita a los científicos e investigadores la tarea de optimizar los proyectos BOINC para GPU de NVIDIA.

Se están utilizando en aplicaciones de dinámica molecular, predicción de estructuras proteicas, imágenes médicas, modelización meteorológica y climática, así como en muchos otros ámbitos (Harish and Narayanan, 2007).

La solución que ofrece CUDA es mucho más flexible y potente, además, se basa en estándares existentes. Los programas se escriben en lenguaje C, no en el ensamblador de un cierto procesador o en un lenguaje especializado como es el caso de Cg. Esto facilita el acceso a un grupo mucho mayor de programadores.

Al desarrollar una aplicación CUDA el programador escribe su código como si fuese a ejecutarse en un único hilo, sin preocuparse de crear y lanzar hilos, controlar la sincronización, etc. Ese código será ejecutado posteriormente en un número arbitrario de hilos, asignado cada uno de ellos a un núcleo de proceso, de manera totalmente transparente para el programador. Éste no tendrá que modificar el código fuente, ni siquiera recompilarlo, dependiendo de la arquitectura del hardware donde vaya a ejecutarse.

Incluso existe la posibilidad de recompilar el código fuente dirigido originalmente a ejecutarse sobre una GPU para que funcione sobre una CPU clásica, asociando los hilos CUDA a hilos de CPU en lugar de núcleos de ejecución de GPU. Obviamente el rendimiento será muy inferior ya que el paralelismo al nivel de CPU no es actualmente tan masivo como en una GPU.

1.3.1 Características de la tecnología CUDA

CUDA es un nuevo hardware y arquitectura de software para la emisión y gestión de cálculos en la GPU como un dispositivo de computación de datos en paralelo. El mecanismo de multitarea del sistema operativo se encarga de gestionar el acceso a la GPU CUDA y a varias aplicaciones de gráficos corriendo al mismo tiempo (NVIDIA_Corporation, 2007).

La pila de software CUDA está compuesto de varias capas como se ilustra en la figura 3: un controlador de hardware, una interfaz de programación de aplicaciones (API), el tiempo de ejecución y bibliotecas matemáticas de uso común.

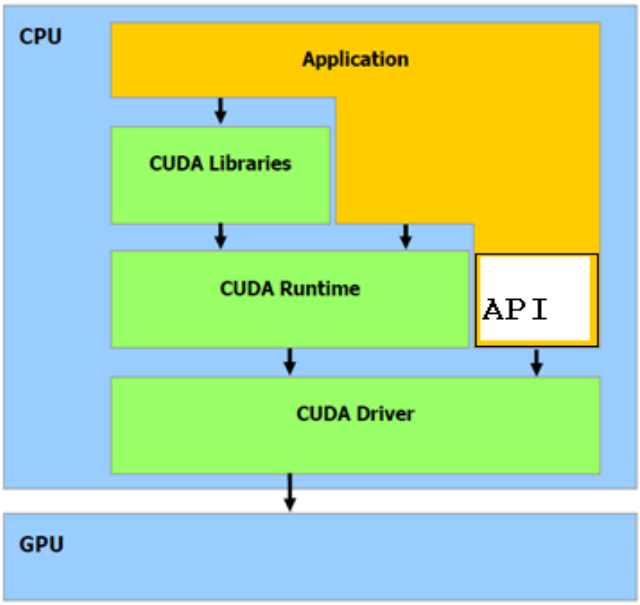


Figura 3. Pila del Software CUDA

El hardware ha sido diseñado para soporte de controladores ligeros y capas (layers) en tiempo de ejecución, dando como resultado un alto rendimiento.

La API de CUDA comprende una extensión del lenguaje de programación C para una curva de aprendizaje mínima.

CUDA proporciona una dirección de memoria general DRAM como se ilustra en la figura 4, para obtener más flexibilidad en la programación: tanto la dispersión y recopilación de operaciones de memoria. Desde una perspectiva de programación, esto se traduce en la capacidad de leer y escribir datos en cualquier lugar de DRAM, al igual que en una CPU.

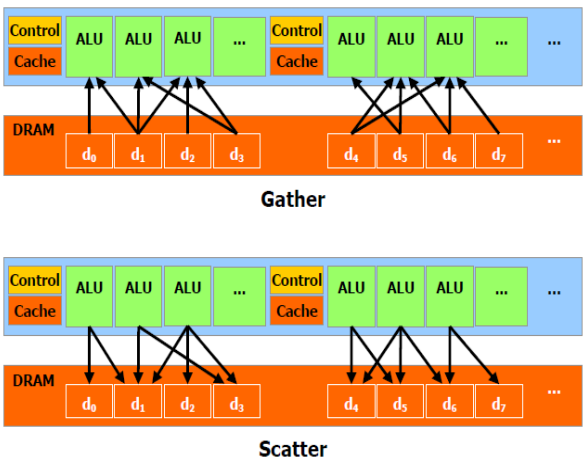


Figura 4. Recopilación y Dispersión de las operaciones de memoria

CUDA cuenta con una caché de datos en paralelo o en el chip de memoria compartida con el acceso de la DRAM, muy rápidas velocidades de lectura y escritura, que los hilos deben usar para compartir datos entre sí (NVIDIA_Corporation, 2008) Como se ilustra en la figura 5, las aplicaciones pueden tomar ventaja de ella, minimizando viajes de ida y vuelta de DRAM y por lo tanto, ser menos dependientes del ancho de banda de memoria DRAM.

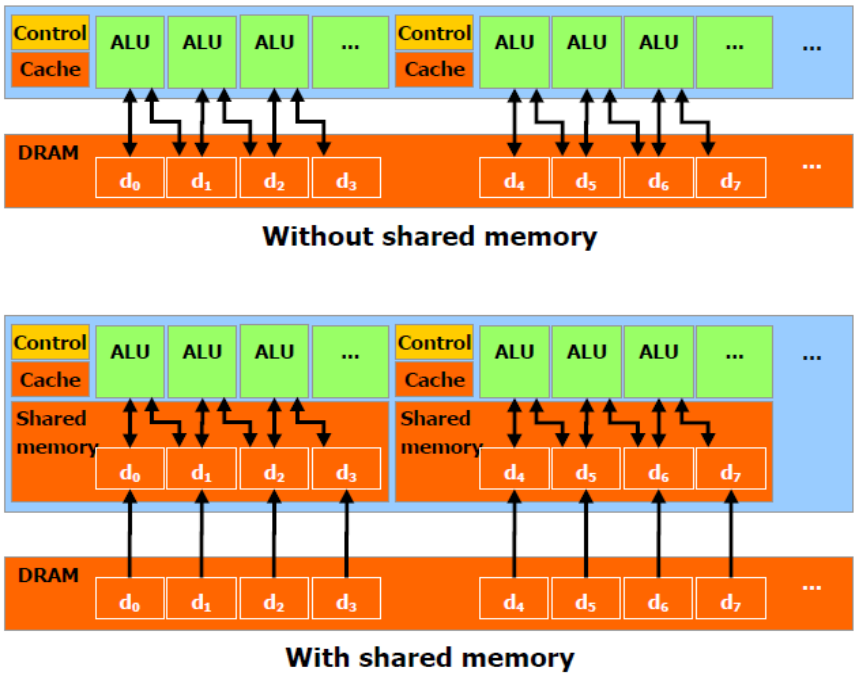


Figura 5. Diferencias en el uso de la memoria compartida

1.3.2 Componentes de CUDA

El controlador CUDA es el componente básico, ya que es el encargado de facilitar la ejecución de los programas y la comunicación entre CPU y GPU. En cualquier caso se requiere una cantidad mínima de 256 MB de memoria gráfica para poder funcionar, por lo que en adaptadores con menos memoria no es posible aprovechar CUDA (NVIDIA_Corporation, 2008).

Instalado el controlador, el siguiente componente fundamental para el desarrollo de aplicaciones es el *toolkit* CUDA, compuesto a su vez de un compilador de C llamado ***nvcc***, un depurador específico para GPU, un *profile* y una serie de bibliotecas con funciones de utilidad ya predefinidas, entre ellas la implementación de la Transformada rápida de Fourier (FFT) y unas subrutinas básicas de álgebra lineal (BLAS).

El tercer componente de interés es el CUDA *Developer* SDK, un paquete formado básicamente por código de ejemplo y documentación. Se ofrece más de medio centenar de proyectos en los que se muestra cómo integrar CUDA con DirectX y OpenGL, cómo paralelizar la ejecución de un algoritmo y cómo utilizar las bibliotecas FFT y BLAS para realizar diversos trabajos: generación de un histograma, aplicación de convolución a una señal, operaciones con matrices, etc.

Conjuntamente estos tres componentes ponen al alcance del programador todo lo que necesita para aprender a programar una GPU con CUDA y comenzar a desarrollar sus propias soluciones, apoyándose en código probado como el de los ejemplos facilitados o el de las bibliotecas FFT y BLAS.

1.3.3 Estructura de una aplicación CUDA

El código de un programa escrito para CUDA siempre estará compuesto de dos partes (Luebke, 2008): una cuya ejecución quedará en manos de la CPU y otra que se ejecutará en la GPU. Al código de la primera parte se le denomina código para la CPU (*host*) y al de la segunda código para el GPU (*device*).

Al ser procesado por el compilador ***nvcc***, el programa generará por una parte código objeto para la GPU y por otro código fuente u objeto para la CPU. Ver epígrafe 2.3.4.

La finalidad del código para la CPU es inicializar la aplicación, transfiriendo el código a la GPU, reservando la memoria necesaria en el dispositivo y llevando a la GPU los datos de partida con los que se va a trabajar. Esta parte del código puede escribirse en C/C++, lo cual permite aprovechar el paradigma de orientación a objetos si se quiere.

El código a ejecutar en el dispositivo debe seguir estrictamente la sintaxis de C, contemplándose algunas extensiones de C++. Normalmente se estructurará en funciones llamadas *kernel*, cuyas sentencias se ejecutarán en paralelo según la configuración hardware del dispositivo final en el que se ponga en funcionamiento la aplicación.

Lo que hace el entorno de ejecución de CUDA, a grandes rasgos, es aprovechar el conocido como paralelismo de múltiples hilos o SIMT (*Single Instruction Multiple Threads)*, consistente en dividir la información de entrada, por ejemplo una gran

matriz de valores, en tantos bloques como núcleos de procesamiento existan en la GPU. Cada núcleo ejecuta el mismo código, pero recibe unos parámetros distintivos que le permiten saber la parte de los datos sobre los que ha de trabajar.

En una CPU moderna, como los *Athlon Phenom* o *Core i7*, es posible dividir los datos de entrada en cuatro o seis partes pero sin ninguna garantía de que se procesarán en paralelo, salvo que se programe explícitamente el reparto trabajando a bajo nivel. En una GPU y usando CUDA, por el contrario, esos datos se dividirán en bloques mucho más pequeños, al existir 240, 512, 1024 o más núcleos de procesamiento, garantizándose la ejecución en paralelo sin necesidad de recurrir a la programación en ensamblador.

1.3.4 Un coprocesador altamente multihilo

Cuando se programa a través de CUDA, la GPU es vista como un dispositivo de cálculo capaz de ejecutar un gran número de hilos en paralelo. Funciona como un coprocesador de la CPU principal, o de acogida: En otras palabras, los datos en paralelo, con porciones de aplicaciones de cálculo intensivo que se ejecutan en la CPU se pueden realizar en la GPU.

Precisamente, una parte de una aplicación que se ejecuta muchas veces, pero de forma independiente en diferentes datos, se puede aislar a una función que se ejecuta en el dispositivo en procesos diferentes (*Kernel)*.

Tanto la CPU y la GPU mantienen su propia DRAM, conocida como la memoria principal y la memoria del dispositivo, respectivamente. Se puede copiar datos de una DRAM a la otra a través de llamadas a la API optimizada, que utiliza el dispositivo de alto rendimiento desde el Acceso Directo a Memoria (DMA) a la máquina.

1.3.5 Ventajas de la tecnología CUDA.

CUDA presenta ciertas ventajas sobre otros tipos de computación sobre GPU:

- Proporciona una mínima curva de aprendizaje para la programación, ya que los programas se escriben en C con algunas extensiones de C++.
- Es capaz de adaptar otras arquitecturas a la suya propia, sin la necesidad de reescribir todo el código.

- Permite el procesamiento tanto gráfico como no gráfico.
- Escala de forma totalmente transparente para el programador la alta paralelizacion que ofrecen sus múltiples núcleos.

En http://www.ecured.cu/index.php/CUDA#Ventajas. :

- Lecturas dispersas: se puede consultar cualquier posición de memoria.
- Memoria compartida: CUDA pone a disposición del programador un área de memoria de 16KB que se compartirá entre hilos. Dado su tamaño y rapidez puede ser utilizada como caché.
- Lecturas más rápidas desde y hacia la GPU.
- Soporte para enteros y operadores a nivel de bit.

Las ventajas en tiempo con respecto a la CPU son mayores cuando:

- El algoritmo tiene un orden de ejecución cuadrático o superior: el tiempo necesario para realizar el traspaso de CPU a GPU tiene un gran coste que no suele verse compensado por el bajo coste computacional de un método lineal.
- Es mayor la carga de cálculo computacional en cada hilo
- Es menor la dependencia entre los datos para realizar los cálculos: esta situación se da cuando cada procesador SIMT sólo necesita de datos guardados en memoria local o compartida y no necesita acceder a memoria global, de acceso más lento.
- Es menor el trasiego de información entre CPU y GPU, siendo óptimo cuando este proceso sólo se realiza una vez, al comienzo y al final del proceso: lo que significa que los datos de entrada al sistema siempre son los mismos y que el proceso no se retroalimenta con la CPU en estados intermedios.
- No existen secciones críticas, es decir, varios procesos no necesitan escribir en las mismas posiciones de memoria: mientras la lectura de memoria global y compartida puede ser simultánea, la escritura en la misma posición de memoria plantea un mecanismo de acceso bloqueo. Este mecanismo está disponible en CUDA mediante las operaciones atómicas (a partir de CUDA 1.1)

1.3.6 Limitaciones de la tecnología CUDA.

- No se puede utilizar recursividad, punteros a funciones, variables estáticas dentro de funciones o funciones con número de parámetros variable.
- En precisión simple no soporta números desnormalizados o NaNs.
- Los hilos, por razones de eficiencia, deben lanzarse en grupos de al menos 32, con miles de hilos en total.
- No está soportado el renderizado de texturas.

1.4 El Modelo de Programación de CUDA

CUDA intenta aprovechar el gran paralelismo, y el alto ancho de banda de la memoria en las GPU en aplicaciones con un gran coste aritmético frente a realizar numerosos accesos a memoria principal, lo que podría actuar de cuello de botella (NVIDIA_Corporation, 2008).

El modelo de programación de CUDA está diseñado para que se creen aplicaciones que de forma transparente escalen su paralelismo para poder incrementar el número de núcleos computacionales (Rondán, 2009). Este diseño contiene 5 puntos clave ((NVIDIA_Corporation, 2008, NVIDIA_Corporation, 2011)) las funciones llamadas Kernel, la jerarquía de hilos, la jerarquía de memorias, la programación heterogénea y la capacidad de cálculo .

La estructura que se utiliza en este modelo está definida por una malla (*grid)*, dentro de la cual hay bloques (*block*) que están formados por distintos hilos (*threads*). El lote de hilos que ejecuta un *kernel* está organizado como una malla de bloques de hilo como se muestra en la figura 6. En general, se puede ver un malla como una representación lógica de la propia GPU, un bloque como un procesador multinúcleo y un hilo como uno de estos núcleos de proceso.

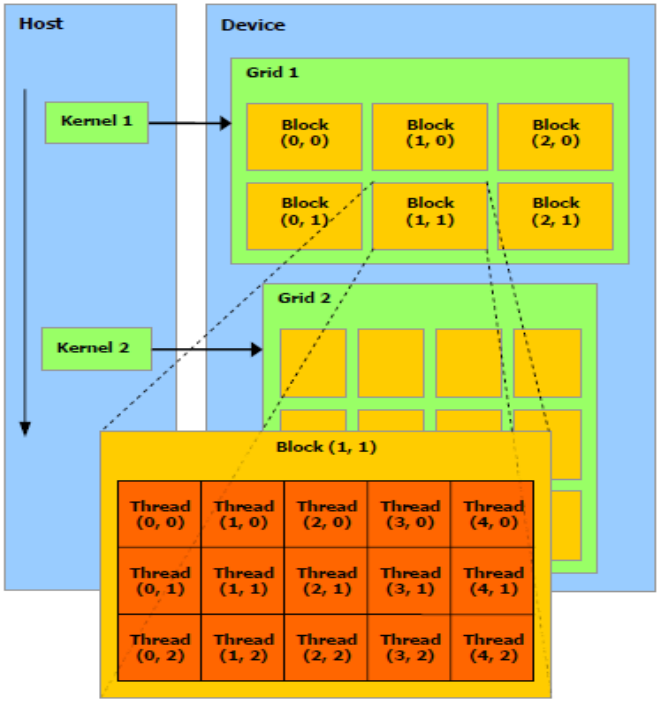


Figura 6. Lotes de Hilos

1.4.1 Los Kernels

Un *kernel* es una función a ejecutarse en la GPU. Una de las extensiones que hizo CUDA al lenguaje de programación C es la definición de la función llamada *Kernel*, esta función se define usando la declaración especifica `_global_`, el número de hilos que ejecutará ese *Kernel* está dado por una nueva sintaxis de configuración de ejecución **<<<…>>>** Ver epígrafe (**¡Error! No se encuentra el origen de la referencia.**). El número de hilos por bloque y el número de bloques por malla especificados en la sentencia **<<<…>>>** pueden ser de tipo `int` o `dim3`. En la figura 7 se muestra un ejemplo de la forma de definición e invocación de un *kernel.*

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Figura 7. Ejemplo de definición e invocación de un kernel

1.4.2 Jerarquía de hilos

Cada hilo está identificado con un identificador único, que se accede con la variable `threadIdx`. Esta variable es muy útil para repartir el trabajo entre distintos hilos. Por conveniencia `ThreadIdx` es un vector de 3 componentes (x, y, z) así estos hilos pueden identificarse usando una, dos o tres dimensiones coincidiendo con las dimensiones de los bloques de hilos.

Al igual que los hilos, los bloques se identifican mediante `blockIdx`. Otro parámetro útil es `blockDim`, para acceder al tamaño de bloque.

La figura 8 muestra un uso de estas tres variables en una porción de código usado para sumar dos matrices.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
```

Figura 8. Ejemplo de un uso de las variables incorporadas.

Un bloque de hilos es un lote de hilos que pueden cooperar juntos de manera eficiente el intercambio de datos a través de una parte de la memoria compartida y rápida sincronización de su ejecución para coordinar los accesos a memoria. Como los distintos hilos dentro de un bloque colaboran entre ellos y pueden compartir datos, se requieren unas directivas de sincronización. En un *kernel*, se puede explicitar una barrera incluyendo una llamada a `__syncthread()`, en la que todos los hilos se esperarán a que los demás lleguen a ese mismo punto.

Cada hilo se identifica por su identificador de hilo, que es el número de hilo dentro del bloque. Para un bloque de una dimensión el ID del hilo es el mismo, para uno bi-dimensional de tamaño (Dx, Dy), el ID del hilo se obtiene a partir de su índice (x,

y) por la expresión (x + y Dx) y en el caso de un bloque tri-dimensional de tamaño (Dx, Dy, Dz), el ID del hilo de un hilo de índice (x, y, z) es (x + y Dx + z Dx Dy).

Hay un número máximo limitado de hilos que un bloque puede contener, consultar epígrafe 3.4. Sin embargo, los bloques de la misma dimensión y tamaño que ejecutan el mismo *kernel* puede agruparse juntos en una malla de bloques, de modo que el número total de hilos que pueden ser lanzados en la invocación de un sólo *kernel* es mucho mayor. Esto se produce a expensas de reducir la cooperación entre hilos, ya que los hilos en los bloques de hilos diferentes de la misma malla no pueden comunicarse y sincronizarse entre sí.

Este modelo permite a los *kernel* ejecutar de manera eficiente en varios dispositivos sin recompilación y con diferentes capacidades paralelas todos los bloques de una malla, de forma secuencial si se tiene muy poca capacidad de paralelismo, en paralelo si se tiene mucha capacidad de paralelismo, o por lo general una combinación de ambos.

Cada bloque se identifica por su identificador de bloque, que es el número de bloque dentro de una malla.

Una aplicación también puede especificar una malla como una arreglo bidimensional de tamaño arbitrario e identificar cada bloque con un índice en su lugar de dos componentes. Para un bloque de dos dimensiones de tamaño (Dx, Dy), el ID de un bloque de índice (x, y) es (x+ yDx).

1.4.3 Jerarquía de memoria

Los hilos en CUDA pueden acceder a los datos desde múltiples espacios de memoria como se muestra en la figura 9. Cada hilo tiene una memoria local privada, mientras que un bloque de hilos tiene una memoria compartida visible para todos los hilos del bloque y con el mismo tiempo de vida que el bloque.

Todos los hilos que conforman una malla tienen acceso a la misma memoria global. También hay dos espacios de memorias adicionales de solo lectura, la memoria constante y la memoria compartida.

Los espacios de memoria global, constante y de textura están optimizados por el uso de memorias diferentes.

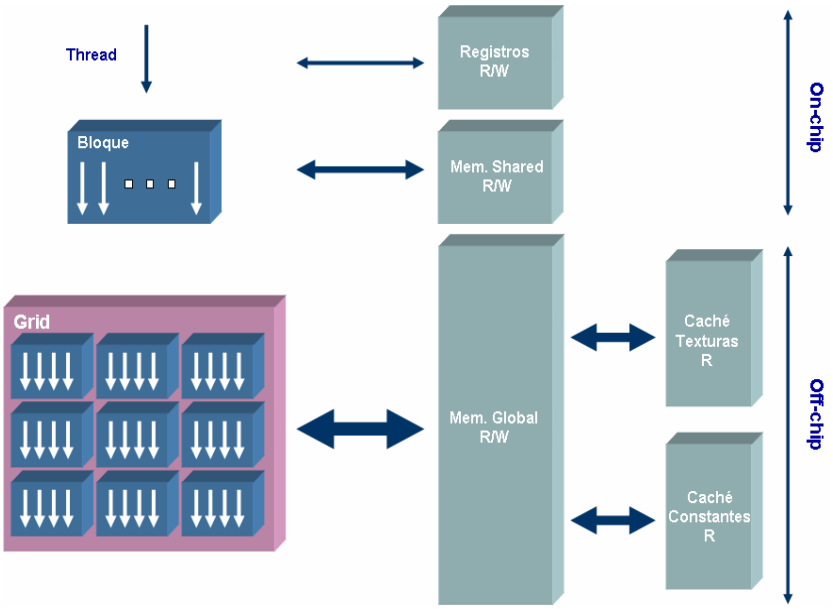


Figura 9. Jerarquía de memoria

Un hilo que se ejecuta en el dispositivo sólo tiene un acceso a memoria RAM del dispositivo y en el chip de memoria a través de los siguientes espacios de memoria, como se ilustra en la figura 10:

- Lectura y escritura por hilo de registros,
- Lectura y escritura por hilos de memoria local,
- Lectura y escritura por bloques de memoria compartida,
- Lectura y escritura por mallas de memoria global,
- Sólo lectura por mallas de memoria constante;
- Sólo lectura por mallas de memoria de textura.

Los espacios de memoria global, constante, y de textura puede ser leído o escrito por el host y son persistentes a través de *kernel* lanzados por la misma aplicación.

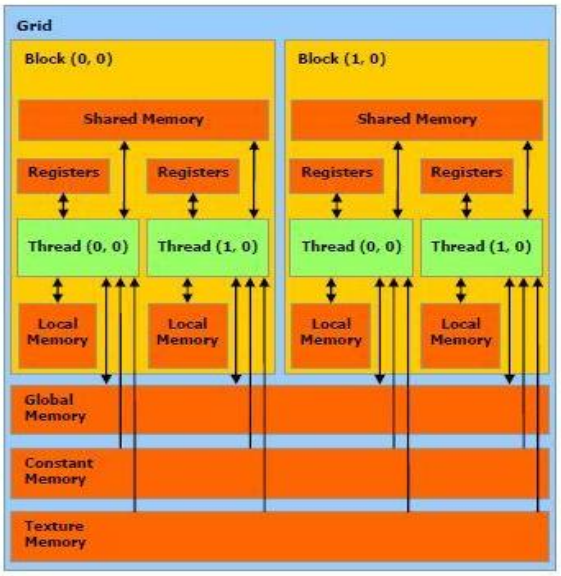


Figura 10. Modelo de memoria

1.4.4 Programación heterogénea

Como se muestra en la figura 11, el modelo de programación de CUDA asume que los hilos se ejecutan separados físicamente, como un dispositivo que opera como un coprocesador a los programas C corriendo en el host. Este es el caso, por ejemplo, cuando los kernels se ejecutan en una GPU y el programa C restante en la CPU. Además este modelo asume que tanto el host como el dispositivo mantienen sus propios espacios de memorias separados en DRAM, como la memoria para el host y memoria para el dispositivo respectivamente. Por tanto un programa administra los espacios de memoria global, constante y de textura visibles a los kernels a través de llamadas al tiempo de ejecución de CUDA.

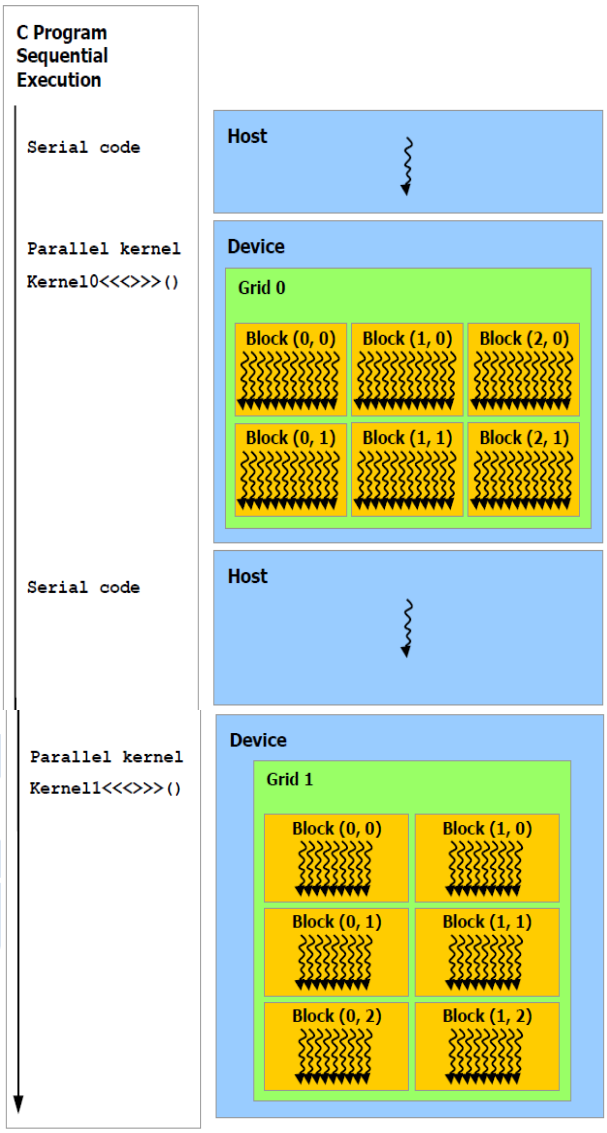


Figura 11. Ejecución alternada entre la CPU y GPU

1.4.5 Capacidad de cálculo

La capacidad de cálculo describe las características del *hardware* y refleja el conjunto de instrucciones soportadas por el dispositivo, también otras especificaciones como el número máximo de hilos por bloques y el número de registros por multiprocesador (NVIDIA_Corporation, 2012a). Ver epígrafe 2.2.3

1.4.6 Claves computacionales para programar la GPU usando CUDA

- Acceso coalescente o fusionado a memoria global: Es la capacidad de la arquitectura para obtener 16 palabras de memoria simultáneamente (en un único acceso) por medio de los 16 hilos del half-warp (mitad de los grupos

warp que son utilizados por el multiprocesador), reduce la latencia de memoria a 16 veces menos, primera cuestión de eficiencia en el desarrollo del *kernel* y se consigue bajo ciertos patrones de acceso, estos patrones dependen de la capacidad de cálculo de la GPU.

- Alineación de segmento.
- Registros o memoria compartida: Proporciona un uso beneficioso al ser memorias de baja latencia, compartidos entre todos los hilos del bloque, existen limitados por sus tamaños, al excederse se hace uso de una memoria especial gestionada sólo por el compilador (*memoria local) l*a memoria local está ubicada en la memoria del dispositivo, por lo que tiene alta latencia. Para controlar el espacio gastado en registros y memorias compartidas, usamos un *flag* al compilador (-Xptxas=-v).
- Uso de cachés (Constante y Textura): Su uso es beneficioso cuando el problema presenta cierta reutilización de datos, presenta algunas limitaciones como el tamaño y permite sólo lectura. En la práctica, el uso de la caché de texturas produce un aumento considerable en el rendimiento final del algoritmo.
- Ancho de banda: Medida que define el buen uso que realizamos de la memoria global, cada GPU tiene un *peak bandwith* que representa el máximo teórico que se puede alcanzar. Para su cálculo no se tiene en cuenta el acceso a cachés, un valor bajo indica que el punto de partida en la optimización del *kernel* es revisar el cómo se realiza el acceso a memoria.
- Divergencia: Ocurre cuando se producen saltos condicionales en el código, (evitar situaciones de este tipo) uso de sentencias ***if condición then… else,*** serialización hasta que se acaba el ***if*** por lo que se pierde paralelismo al estar muchos hilos parados
- Transferencias CPU a GPU y viceversa: La CPU se comunica con la GPU por medio del bus PCIX, existe una alta latencia en comparación con el acceso a memoria GDDR4 y GDDR5, el punto máximo de ancho de banda es igual a 8Gbps con PCIX 16x Gen2. Para optimizar las transferencias debe hacerse la comunicación de CPU a GPU, procesamiento y resultados de

GPU a CPU. En ciertos problemas, los datos a procesar pueden generarse directamente en la GPU sin necesidad de la comunicación inicial

- Ocupación: Mide el grado en el que el *kernel* que se ejecuta mantiene ocupados a los SMs, la ejecución de un *warp* (grupos de 32 hilos que ejecuta un multiproccesaror) mientras otro está pausado o estancado es la única forma de ocultar latencias y mantener el hardware ocupado, una alta ocupación no implica siempre un alto rendimiento, una baja ocupación imposibilita la ocultación de latencias por lo que resulta en una reducción del rendimiento, se puede calcular con: ***Cuda occupancy calculator (xls)*** y ***Cuda visual profiler.***
- Reglas para determinar el tamaño del bloque: Número de hilos debe ser múltiplo del tamaño del *warp* (32) facilita coalescencia, evitar *warps* incompletos, si existen suficientes bloques por SMs comenzar con un tamaño mayor o igual a 64 hilos por bloque, comprobar experimentalmente con 512.
- Cuda Visual Profiler: Es una herramienta para medir parámetros de rendimiento de un *kernel* como el número de accesos coalescentes, de saltos condicionales, divergencia, ancho de banda entre otros. Obtiene valores de la estadística sobre varios SMs, no sobre el total, extrapola los resultados al total de SMs (es una aproximación) y el ancho de banda incluye todas las estructuras de memoria, no sólo las relevantes para el algoritmo.
- Optimización en operaciones:
    - +, *, add-multiply: 8 operaciones/ciclo
    - / : 0.88 operaciones/ciclo
    - Uso de operaciones fast_math: para comprobar precisión requerida

Operator/Function	Device Function
x/y	__fdividef(x,y)
sinf(x)	__sinf(x)
cosf(x)	__cosf(x)
tanf(x)	__tanf(x)
sincosf(x,sptr,cptr)	__sincosf(x,sptr,cptr)
logf(x)	__logf(x)
log2f(x)	__log2f(x)
log10f(x)	__log10f(x)
expf(x)	__expf(x)
exp10f(x)	__exp10f(x)
powf(x,y)	__powf(x,y)

Tabla 1. Operadores y funciones.

1.5 Aplicaciones que usan la tecnología CUDA

CUDA puede ser usado para diferentes propósitos ya sea de tipo científico, matemático, físico, químico, etc. Entre ellos hay aplicaciones que requieren de una intensidad aritmética muy alta, como algebra lineal densa, diferencias finitas, etc. Otras que usan un gran ancho de banda como la secuencia de escaneo de virus, bases de datos, entre otras y de computación visual como la representación de gráficos, procesamiento de imágenes, tomografías, visión de máquinas etc. (Luebke, 2008). Se pueden consultar otras en:

http://www.nvidia.es/object/cuda_home_es.html

Uso de CUDA para diferentes fines:

- La empresa Elcomsoft a creado un software que aprovecha la tecnología CUDA para reventar las claves WPA y WPA2 casi sin problemas, además se ha probado que ese mismo software sirve del mismo modo para descifrar contraseñas WPA y WPA2 en tiempos récord, lo que ha hecho que varios analistas avisen del peligro que tiene ahora cualquier tipo de red inalámbrica. La potencia bruta de las nuevas tarjetas gráficas se está convirtiendo en un gran aliado a la hora de romper todo tipo de sistemas de cifrado. Estos procesadores gráficos disponen de una potencia mucho mayor que la de cualquier procesador actual, y Elcomsoft ha logrado trasladar este tipo de tareas a un lenguaje que la GPU puede comprender y ejecutar gracias a la tecnología CUDA de NVIDIA. De hecho, la velocidad de descifrado es diez mil veces superior a la que lograría una CPU en la gran parte de los casos.

- Matlab y Photoshop han creado *plugins* que usan CUDA para acelerar los cálculos que realizan, ya que con esta tecnología se pueden ejecutar miles de hilos en cada uno de los núcleos que contiene la GPU a una velocidad superior a la de una CPU actual. Son aplicaciones que realizan una gran cantidad de cálculos matemáticos donde es fundamental que las operaciones duren lo menos posible.
- El software BadaBOOM, desarrollado por Elemental Technologies, nos permite codificar vídeo desde y hacia distintos formatos, un proceso que puede ocupar intensivamente nuestra CPU. Pero, la diferencia de BadaBOOM con cualquier otro software de conversión de vídeos es que no ocupa masivamente la CPU, sino que realiza los cálculos utilizando la GPU. NVIDIA está apoyando a muchos grupos de desarrolladores que están programando software para GPU mediante CUDA, donde podemos encontrar desde clientes de Folding@Home a programas profesionales de cálculo científico.
- NVIDIA ha anunciado un software CUDA beta para Mac que se define como un kit desarrollador que permite a los usuarios crear trabajos para objetivos académicos, comerciales o personales.
- La tarjeta PhysX, creada en su día por AGEIA, integrará su motor software o API en la arquitectura CUDA desarrollada por NVIDIA. De este modo, los desarrolladores pueden usar los procesadores gráficos para diferentes cálculos sin que tengan relación con la información que se vaya a visualizar en 3D.
- El software S3FotoPro fue presentado por el fabricante S3, bastante desconocido para el público en general. Con él, pretenden utilizar los procesadores de las gráficas de S3 como sistemas de propósito general, para el procesado de otro tipo de datos. El S3FotoPro permite crear varios miles de hilos de ejecución en paralelo dentro del procesador de la tarjeta S3. Con esto generan varios *gigaFLOPS* (operaciones de punto flotante por segundo) de procesamiento que pueden utilizarse para diferentes tareas tales como el procesado de imágenes, la edición de vídeos y su conversión a formatos de uso general, para el procesado de sistemas de datos con fines medicinales o

científicos, e incluso también para crear motores físicos basados en la realidad. En todos estos casos se trata de liberar a la CPU del mayor trabajo posible y dividir las tareas cuando la GPU esté libre.

- SciFinance es el producto estrella de la empresa americana SciComp, con sede en Austin, Texas. Dispone de un software de derivados avanzado para acortar el tiempo de desarrollo y acelerar el resultado de las simulaciones realizadas mediante el método de Monte Carlo. La compañía ha perfeccionado SciFinance para acelerar el cálculo de los modelos de valoración de derivados mediante la tecnología CUDA de NVIDIA, lo que permite ejecutar estos modelos hasta 100 veces más rápido de lo que se ejecutan con código no paralelo. Pero lo más importante es que esta aceleración puede lograrse sin ningún trabajo o programación manual extra, algo que en un mercado donde las esperas o la falta de precisión pueden generar millones de pérdidas, es un avance esencial. La clave para este aumento de la velocidad es utilizar la GPU, para efectuar los cálculos. Esta extraordinaria capacidad de procesamiento se logra con la ayuda de la arquitectura CUDA de NVIDIA.
- Apple está adaptando y publicando sus nuevas herramientas de desarrollo basadas en la tecnología CUDA para, de esta forma, poder facilitar el lanzamiento de aplicaciones de computación paralela. La compañía parece estar muy interesada en esta nueva tecnología y de hecho, está planeando realizar su propia versión de herramientas de desarrollo aprovechando los APIs de CUDA para desarrollar aplicaciones de todo tipo que utilizan el procesamiento en paralelo de los últimos chips de las tarjetas gráficas de NVIDIA.
- El Adobe Creative Suite 4 es un conjunto de programas de edición de imágenes, fotografías, vídeos y páginas web de la famosa compañía Adobe. Entre otros programas incluye el Adobe PhotoShop y el Adobe Premiere por citar dos ejemplos.

  Estas aplicaciones utilizan la tarjeta NVIDIA Quadro CX, una gran novedad perteneciente a la gama más potente y profesional de todo el catálogo que posee NVIDIA. Esta tarjeta hace uso del motor CUDA para mejorar el

rendimiento de todas estas nuevas aplicaciones de Adobe en aspectos como los movimientos, el uso del zoom, la carga de ficheros, la exportación de vídeo o la velocidad de aplicación de filtros.

- El raytracing, una técnica de renderizado que representa la luz de una manera muy eficaz y que la aprovecha para crear gráficos en tres dimensiones por ordenador. Consiste en el trazado de rayos en tiempo real.

  Gracias a la arquitectura creada por NVIDIA, el raytracing ha evolucionado a un nivel de trabajo de 30 imágenes por segundo. Existen GPUs que realizan el trazado de rayos a 30 fps (frames por segundo) y con una resolución de 1920 x 1080 píxeles.

  Otras tareas que se pueden realizar con el raytracing pueden ser la realización de polígonos, imágenes de pintura con base de Shader, trazado de rayos y sombras, refracciones y reflexiones.

1.6 Consideraciones finales del capítulo

A medida que han evolucionado las capacidades de las tarjetas gráficas y la incorporación de nuevas funcionalidades y también las nuevas API de programación (OpenGL y DirectX), que hacen referencia a una arquitectura especializada dirigida específicamente al tratamiento gráfico. Estas arquitecturas presentan un leguaje de alto nivel y no aprovechan al máximo las posibilidades para otro tipo de propósito.

Es aquí donde cobra protagonismo CUDA una nueva arquitectura de software para la emisión y gestión de cálculo en la GPU permitiendo explotar al máximo la gran potencia de las GPU tanto para aplicaciones gráficas y no gráficas. Sus características ofrecen una solución mucho más flexible y potente además los programas se escriben en lenguaje C para una curva de aprendizaje mínima, lo cual facilita el acceso a un grupo mucho mayor de programadores.

Capítulo 2. Implementación del hardware y la API

En este capítulo se parte de la arquitectura SIMT (*Single-Instruction, Multiple-Thread*) en la GPU, la implementación del *hardware*, la API y el componente de tiempo de ejecución en el dispositivo y el *host* para establecer un procedimiento de trabajo usando la tecnología CUDA.

2.1 La arquitectura SIMT en la GPU

Un multiprocesador está diseñado para ejecutar cientos de hilos concurrentemente, manejar tanta cantidad de hilos es empleada por una única arquitectura llamada SIMT (NVIDIA_Corporation, 2011). Los multiprocesadores crean, manejan, programan y ejecutan hilos en grupos de 32 hilos paralelos llamados (*warps*) como se muestra en la figura 12. Un (*half-warp*) es la primera o segunda mitad de un *warp* y un (*quarter-warp)* la primera, segunda o cuarta parte de un *warp*. La forma en que un *warp* esta particionado dentro de un bloque es siempre la misma, cada *warp* contiene hilos con ID de hilos consecutivos y crecientes con el primer *warp* que contiene hilo 0. Esta organización tiene el objetivo de aprovechar al máximo las posibilidades de paralelismo de la arquitectura, ocultando latencias mediante la ejecución concurrente de hilos. La arquitectura SIMT es semejante a SIMD ya que su organización es vectorial, en esa instrucción simple se controlan múltiples elementos procesándose. Una diferencia es que la organización de los vectores en SIMD expone el ancho de SIMD al software, mientras que SIMT especifica las instrucciones de ejecución y división procedentes de un simple hilo.

Figura 12. Los warps dentro de un bloque

De una manera muy bien definida dentro de cada bloque activado hay una distribución de *warps* con diferentes porciones de tiempos, en otras palabras los hilos dentro del *warp* son ejecutados físicamente en paralelo, mientras que los *warps* y los bloques son ejecutados lógicamente en paralelo (Zeller, 2008).

2.1.1 Multiprocesadores SIMT con un chip de memoria compartida

El dispositivo se implementa como un conjunto de multiprocesadores, como se ilustra en la figura 13. Cada multiprocesador tiene una sola instrucción, la arquitectura de hilos múltiples SIMT. En cualquier ciclo de reloj, cada procesador del multiprocesador ejecuta la misma instrucción, pero opera sobre hilos diferentes (NVIDIA_Corporation, 2008).

Cada multiprocesador tiene en el chip de memoria lo siguientes:

- Un conjunto de registros locales de 32-bit por procesador.
- Una cache de datos en paralelos o memoria compartida (*shared memory)* que es compartida por todos los procesadores y pone en práctica el espacio de memoria compartida.
- Sólo lectura de la caché constante (*constant cache)* y la caché de textura (*texture cache)* que es compartida por todos los procesadores y acelera lecturas desde los espacios de memoria constante y de textura respectivamente, que se implementan como una región de solo lectura de la memoria del dispositivo.

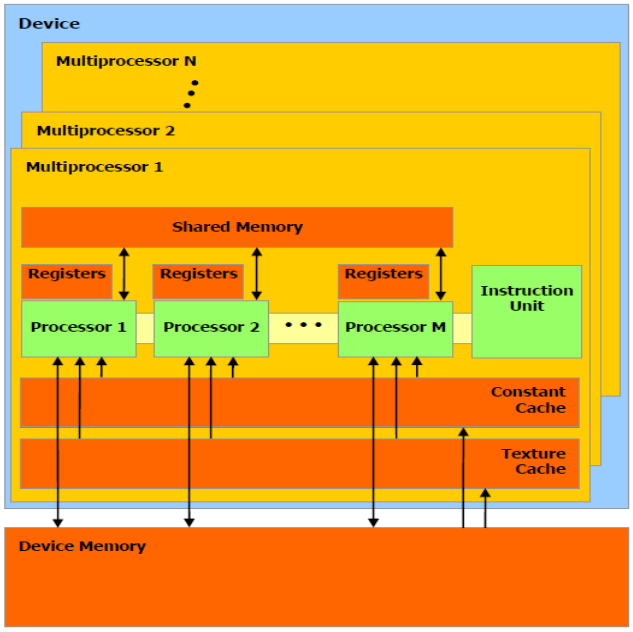


Figura 13. Modelo Hardware

2.2 Implementación del Hardware

La arquitectura CUDA está construida por arreglos escalables de multihilos (SMs). Cuando un programa CUDA en el host invoca la malla de un *kernel*, los bloques de la malla están enumerados y distribuidos a multiprocesadores con capacidades de ejecución habilitadas. La implementación del hardware en CUDA está formada por cuatro partes; modelo de ejecución, capacidad de cálculo, los dispositivos múltiples y los cambios de modo, (NVIDIA_Corporation, 2012c), a continuación se aborda cada una de ellas.

2.2.1 Modelo de ejecución

Una malla de bloques de hilos se ejecuta en el dispositivo mediante la ejecución de uno o más bloques en cada multiprocesador usando un tiempo de corte, o sea, los distintos tipos de warps.

Un bloque es procesado por sólo un multiprocesador, de modo que el espacio de memoria compartida reside en el *on-chip* de la memoria compartida principal que lleva muy rápido a los accesos a memoria. Los registros del multiprocesador son asignados entre los hilos del bloque. Si el número de registros utilizados por hilo,

multiplicado por el número de hilos en el bloque es mayor que el número total de registros por multiprocesadores, el bloque no puede ser ejecutado y el *kernel* correspondiente no se inicia.

Varios bloques pueden ser procesados por el mismo multiprocesador al mismo tiempo mediante la asignación de registros de los múltiples procesadores y la memoria compartida entre los bloques.

El orden de emisión de los *warps* dentro de un bloque no está definido, pero su ejecución se puede sincronizar, para coordinar los accesos a memoria global o compartida.

El orden de emisión de los bloques dentro de una malla de bloques de hilo no está definido y no existe un mecanismo de sincronización entre los bloques, por lo que los hilos a partir de dos bloques diferentes de la misma malla no se pueden comunicar con seguridad entre sí a través de la memoria global durante la ejecución de la malla.

Si una instrucción no atómica ejecutada por un *warp* escribe en la misma ubicación dentro de la memoria global o compartida por más de uno de los hilos del *warp*, el número de escrituras serializadas que se producen en ese lugar y el orden en el que se producen no está definido, pero una de las escrituras es garantía de éxito. Si una instrucción atómica ejecutada por un *warp* lee, modifica, y escribe en la misma ubicación de la memoria global por más de uno de los hilos del *warp*, cada lectura, modificación y escritura en ese lugar se produce y todos ellos estarán en serie, pero el orden en que se producen no está definido.

2.2.2 Dispositivos múltiples

El uso de múltiples GPU como dispositivo CUDA por una aplicación que se ejecuta en un sistema multi-GPU sólo se garantiza que funcione si estas GPU son del mismo tipo. Si el sistema está en modo SLI sin embargo, sólo una GPU puede ser utilizado como un dispositivo CUDA ya que todas las GPU se fusionan en el nivel más bajo, en la pila de controladores. El modo SLI tiene que ser desactivado en el

panel de control para que CUDA pueda ver cada GPU como dispositivos independientes.

2.2.3 Características de la Capacidad de cálculo

La capacidad de cálculo (*compute capability*) de un dispositivo se define por un número de revisión principal y un número de revisión secundaria.

Los dispositivos con el mismo número de revisión principal son de la misma arquitectura de *cores*. El número de revisión principal basados en la arquitectura Fermi es de 2, la serie GeForce 8, Quadro FX 5600/4600, y las soluciones Tesla tienen capacidad de cálculo 1.x, su número de revisión principal es 1.

El número de revisión secundaria corresponde a una mejora incremental en la arquitectura de *cores*, incluyendo nuevas características. En el Anexo 2 puede consultar un listado de todos los dispositivos habilitados en CUDA con sus capacidades de cálculo (NVIDIA_Corporation, 2011).

2.2.4 Cambios de modo

Las GPU dedican parte de la memoria DRAM a la llamada superficie principal (*primary surface*), que se utiliza para refrescar la pantalla del dispositivo, cuya salida es vista por el usuario. Cuando los usuarios inician un cambio de modo (*mode switch)* de la pantalla para cambiar la resolución o profundidad de bits (con NVIDIA panel de control o el panel de control de pantalla en Windows), la cantidad de memoria necesaria para la superficie principal cambia. Por ejemplo, si el usuario cambia la resolución de pantalla de 1280x1024x32 bits a 1600x1200x32 bits, el sistema debe dedicar 7,68 MB a la superficie principal en lugar de 5,24 MB.

En Windows otros eventos que pueden iniciar cambia modo de pantalla, incluir el lanzamiento de una aplicación de pantalla completa de DirectX, pulsando Alt + Tab, para tareas lejos de un cambio de una aplicación a pantalla completa de DirectX o para bloquear el equipo, presionando Ctrl + Alt + Supr.

Si el cambio de modo aumenta la cantidad de memoria necesaria para la superficie principal, el sistema dará como resultado una colisión dedicada a las aplicaciones CUDA.

2.3 La API

El objetivo de la interfaz de programación CUDA es proporcionar un camino relativamente sencillo para los usuarios familiarizados con el lenguaje de programación C, para que se pueda escribir fácilmente programas que son ejecutados por el dispositivo. A continuación se muestran las características de la API de CUDA.

2.3.1 La API una extensión al lenguaje de programación C

Se compone de:

- Un conjunto mínimo de extensiones para el lenguaje C: Permiten al programador tomar porciones del código fuente para su ejecución en el dispositivo.
- Una biblioteca de ejecución dividido en:
  - ✓ Un componente para la CPU, se ejecuta en la CPU y proporciona funciones de control y acceso a uno o más dispositivos desde ella.
  - ✓ Un componente dispositivo, se ejecuta en el dispositivo y ofrece funciones específicas del dispositivo.
  - ✓ Un componente común, proporciona un sistema integrado tipos vectores y un subconjunto de la biblioteca estándar de C que admiten tanto la CPU como el código del dispositivo.

Cabe destacar que las únicas funciones de la biblioteca estándar de C que son soportadas para ejecutarse en el dispositivo son las funciones proporcionadas por el componente de tiempo de ejecución común.

2.3.2 Extensiones del lenguaje

Existen extensiones al lenguaje de programación C que es usado por CUDA (NVIDIA_Corporation, 2011):

- Funciones de tipo calificativas*:* para especificar si una función se ejecuta en la CPU o en el dispositivo y si se puede llamar desde la CPU o desde el dispositivo.

  - ✓ **`_device_`** (Función que se ejecuta en el dispositivo, invocable solamente desde el dispositivo).

  - ✓ **`_global_`** (Función que se ejecuta en el dispositivo, invocable solamente desde la CPU).

  - ✓ **`_host_`** (Función que se ejecuta e invocable en la CPU solamente).

    Es equivalente a declarar una función con solo el calificador **`_host_`** o declarar sin ninguno de los calificadores (**`_host_`**, **`_device_`**, **`_global_`**) en cualquier caso la función es compilada solamente por la CPU. Sin embargo el calificador **`_host_`** puede ser usado con el calificador **`_device_`** en cualquier caso la función es compilada tanto por la CPU como por el dispositivo.

  - ✓ **`_noinline_`** y **`_forceinline_`** (Funciones calificativa que pueden ser usadas como una sugerencia para que el compilador en el caso de `_noinline_` no en-linear y para `_forceinline_` forzar a en-linear la función si es posible).

    Cuando el código está siendo compilado por un dispositivo de capacidad de cálculo 1.x una función `_device_` por defecto está siempre en-lineada, mientras que para dispositivos de capacidad de cálculo 2.x una función `_device_` está solamente en-lineada cuando se considera apropiado para el compilador.

- Variables de tipo calificativas*:* para especificar la localización de memoria de una variable en el dispositivo

  - ✓ **`_device_`** (Este calificador declara una variable que reside en el dispositivo, si esta junto con los calificadores descritos posteriormente sirve para especificar aún más qué espacio de memoria le pertenece a la variable, si ninguno de ellos está presente la variable se encuentra en el espacio de memoria global, además tiene la duración de una aplicación y es accesible desde todos los hilos dentro de la malla y desde el host a través de la biblioteca de tiempo de ejecución).

  - ✓ **`_constant_`** (Este calificador se usa opcionalmente junto con **`_device_`**, declara una variable que reside en el espacio de memoria constante, tiene la duración de una aplicación y es accesible desde todos los hilos dentro de la malla y desde el host a través de la biblioteca de tiempo de ejecución).

  - ✓ **`_shared_`** (Este calificador se usa opcionalmente junto con **`_device_`**, declara una variable que reside en el espacio de memoria compartida de un bloque de hilos, tiene la duración del bloque y es solo accedido desde todos los hilos dentro del bloque. Existe una coherencia secuencial de variables compartidas dentro de un hilo, sin embargo el orden de pedido a través de los hilos es sincronizado. Sólo después de la ejecución de `_syncthreads ( )` se escribe desde otros hilos que están garantizados para ser visibles. El compilador es liberado para optimizar la lectura y escritura en la memoria compartida siempre y cuando la declaración anterior se cumpla. Cuando se declara un variable en la memoria compartida como una matriz externa como *`extern_shared_float shared [ ]`*; el tamaño de la matriz se determina en el momento del lanzamiento, todas las variables declaradas de esta manera van a comenzar en la misma dirección en memoria, de modo que la distribución de las variables de la matriz deben ser administradas a través de compensaciones (*offsets*).

- Una nueva directiva para especificar como un *kernel* es ejecutado en el dispositivo desde la CPU:

  - ✓ Cualquier llamada a una función **`_global_`** debe especificar la configuración de la ejecución de esa llamada. La configuración de ejecución define la dimensión de la malla y de los bloques que se utilizarán para ejecutar la función en el dispositivo. Se especifica mediante la inserción de una expresión de la forma <<< **Dg** , **Db** , **Ns** >>> entre el nombre de la función y la lista de argumentos entre paréntesis, donde:

    - **Dg** es de tipo **`dim3`** (especifica la dimensión y el tamaño de la malla, de manera que **Dg. x * Dg. y** es igual al número de bloques que se puso en marcha.
    - **Db** es de tipo **`dim3`** (especifica la dimensión y el tamaño de cada bloque, de manera que **Db. x * Db. y * Db. z** es igual al número de hilos por bloque.
    - **Ns** es de tipo **`size_t`** (especifica el número de bytes en la memoria compartida que se asigna dinámicamente por bloque, para estas llamadas además se asigna memoria estáticamente. Esta asignación en memoria dinámicamente es utilizada por cualquiera de las variables como una matriz externa. **Ns** es un argumento opcional que por defecto es 0. Los argumentos de la ejecución de configuración son evaluados antes que los argumentos de la función actual.

      A modo de ejemplo, una función declarada como:

      ```
      _global_ void func (float parámetro *);
      ```

      Se debe llamar de esta manera:

      ```
      func <<<Dg, Db, Ns>>> (parámetro);
      ```

- Cinco variables incorporadas para especificar las dimensiones e índices.

  - ✓ **`gridDim`** (esta variable es de tipo **`dim3`** y contiene las dimensiones de la malla).

  - ✓ **`blockIdx`** (esta variable es de tipo **`unit3`** y contiene el índice del bloque dentro de la malla).

  - ✓ **`blockDim`** (esta variable es de tipo **`dim3`** y contiene las dimensiones del bloque).

  - ✓ **`threadIdx`** (esta variable es de tipo **`unit3`** y contiene el índice del hilo dentro del bloque).

  - ✓ **`warpSize`** (esta variable es de tipo **`int`** y contiene el tamaño del *warp* en hilos)

  Estas variables tienen como restricciones que no se permite tomar la dirección de cualquiera de las variables incorporadas ni asignar valores a cualquiera de estas variables.

- Tipo de vectores incorporados.

  Existen tipos de vectores derivados desde enteros básicos a punto flotantes, consultar Anexo 3 con el listado de todos los tipos de vectores. Están estructurados en el primer, segundo, tercero y cuarto componente que son accesibles a través de los campos x, y, z y w respectivamente. Todos ellos vienen con una función constructor de la forma **`make_<type name>`**, por ejemplo creando un vector de tipo int2 con valor (x,y) sería:

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

Cada archivo que contiene estas extensiones debe ser compilado con el compilador CUDA ***nvcc***. Cada una de estas extensiones viene con algunas restricciones. ***Nvcc*** dará un error o un aviso en alguna violación de estas restricciones

2.3.3 Gestión de datos y comunicación entre CPU y GPU

Para gestionar memoria en la GPU se utiliza la función:

- **cudaMalloc**((**void** **)*nombre_variable, cantidad_de_bytes)

Esta función se invoca desde la CPU y gestiona la cantidad de *bytes* indicados en la GPU, los cuales serán referenciados por medio de una variable dinámica específica, el siguiente ejemplo muestra como se gestiona memoria para un arreglo de tres números de tipo **float**, a los que se hará referencia mediante la variable v:

```
float *v;
cudaMalloc((void **)&v, 3 * sizeof(float));
```

Para liberar la memoria gestionada, se utiliza la función:

- **cudaFree**(nombre_variable);

El contenido de las variables gestionadas con **cudaMalloc**, no puede modificarse ni visualizarse directamente desde la CPU, ni los datos que residen en el CPU son accesibles desde el GPU; por ello se hace necesario copiar los datos desde y hacia la GPU, utilizando la siguiente función:

- **cudaMemcpy**(destino, fuente, cant, tipo);

Donde:

Destino: Es la dirección de memoria donde se almacenarán los datos copiados.

Fuente: Es la dirección de memoria donde residen los datos que van a copiarse.

Cant: Es la cantidad de bytes que van a copiarse.

Tipo: Indica el tipo de transferencia que se realizará y puede tomar los siguientes valores:

- ✓ **cudaMemcpyHostToDevice**: Los datos se copian desde el host hacia el dispositivo.
- ✓ **cudaMemcpyDeviceToHost**: Los datos se copian desde el dispositivo hacia el host.
- ✓ **cudaMemcpyDeviceToDevice**: Los datos se copian desde el dispositivo hacia el dispositivo.
- ✓ **cudaMemcpyHostToHost**: Los datos se copian desde el host hacia el host. En este caso, la función es equivalente a utilizar la función **memcpy** del C estándar.

Al usar cualquiera de estos tipos de transferencia, hay que tener especial cuidado en que la ubicación de las variables fuente y destino sea la correcta, de acuerdo al sentido de la copia.

2.3.4 Compilación con NVCC

El compilador ***nvcc*** es un *driver* de compilación que simplifica el proceso de compilar un código CUDA. Ofrece opciones simples y de líneas de comandos familiares, que los ejecuta mediante la invocación de la colección de herramientas que están implementadas en diferentes etapas de la compilación.

El flujo de trabajo básico de ***nvcc*** consiste en separar el código del dispositivo del código de la CPU y compilar el código del dispositivo en una forma binaria u objeto (*cubin*). El código generado por la CPU puede salir ya sea como código C que se deja para ser compilado usando otra herramienta o como código objeto directamente mediante la invocación del compilador de la CPU durante la última etapa de compilación.

Las aplicaciones pueden pasar por alto el código generado por la CPU y cargar el objeto (*cubin*) en el dispositivo, ejecutando el código del dispositivo mediante el uso del *driver* de CUDA API o un enlace al código generado por la CPU que incluye el objeto (*cubin*) como una inicialización global de arreglos de datos, además contiene una traducción de la sintaxis de configuración de ejecución en el código de inicio necesario en tiempo de ejecución CUDA para cargar y lanzar cada *kernel* compilado. La tabla 2 muestra un conjunto de objetos habilitados en CUDA.

Object	Handle	Description
Device	CUdevice	CUDA-capable device
Context	CUcontext	Roughly equivalent to a CPU process
Module	CUmodule	Roughly equivalent to a dynamic library
Function	CUfunction	Kernel
Heap memory	CUdeviceptr	Pointer to device memory
CUDA array	CUarray	Opaque container for one-dimensional or two-dimensional data on the device, readable via texture references
Texture reference	CUtexref	Object that describes how to interpret texture memory data

Tabla 2. Objetos habilitados en el driver API de CUDA.

2.4 Componente de tiempo de ejecución en el dispositivo y la CPU

Estos componentes son muy importantes ya que proporcionan funciones ya sea para el dispositivo o la CPU

2.4.1 Tiempo de ejecución en el dispositivo

El componente de tiempo de ejecución del dispositivo sólo se puede utilizar en las funciones del dispositivo:

- Funciones matemáticas: El compilador tiene una opción (`_fast_math`) para obligar a todas las funciones que compile a su equivalente menos preciso, si existe.

- Función de sincronización: `void __syncthreads ()`; Sincroniza todos los hilos en un bloque, una vez que todos los hilos han llegado a este punto, la ejecución continúa normalmente; `__syncthreads ()` se utiliza para coordinar la comunicación entre los hilos de un mismo bloque. Cuando algunos hilos dentro de un bloque acceden a las mismas direcciones en la memoria compartida o global, hay riesgos potenciales de lecturas después de escrituras, escrituras después de lecturas o escrituras después de escrituras de algunos de estos accesos a memoria. El peligro de estos datos se pueden evitar mediante la sincronización de los hilos entre estos accesos. `__syncthreads ()` está permitido en el código condicional, pero sólo si la condición se evalúa completa de forma idéntica en todos los hilos del bloque, de lo contrario la ejecución del código es probable que se cuelgue o producir efectos secundarios no deseados.

- Funciones de conversión: Los sufijos a continuación en las funciones (IEEE-754) indican modos de redondeo: Con **`rn`** se redondea al más cercano (incluyéndolo), **`rz`** a cero, **`ru`** hacia infinitos positivo y **`rd`** hacia infinitos negativos.

int __float2int_ [rn, rz, ru, rd] (float); (convierte el argumento de punto flotante a un entero, utilizando la modalidad de redondeo especificado).

unsigned int __float2uint_ [rn, rz, ru, rd] (float); (convierte el argumento de punto flotante a un entero sin signo, utilizando la modalidad de redondeo especificado).

floatn __int2float_ [rn, rz, ru, rd] (int); (convierte el argumento entero en un número de punto flotante, utilizando la modalidad de redondeo especificado).

floatn __uint2float_ [rn, rz, ru, rd] (unsigned int); (convierte el argumento entero sin signo a un número de punto flotante, utilizando la modalidad de redondeo especificado).

- Funciones de textura:
  - ✓ Textura de la memoria del dispositivo: Se accede con la familia de funciones `tex1Dfetch ()`. Estas funciones buscan la región de la memoria lineal obligando a la textura de referencia texturas de coordenadas x.
  - ✓ Textura de arreglos CUDA*:* Se accede con `tex1D ()` o `tex2D ()`. Estas funciones que obtienen los arreglos CUDA obligan a la textura de referencia texturas de coordenadas x e y.
- Funciones Atómicas: Sólo están disponibles para los dispositivos de la capacidad de cálculo 1.1. Una función atómica realiza una operación de lectura-modificación-escritura atómica en una palabra de 32 bits en la memoria global. Por ejemplo, `atomicAdd ()` lee una palabra de 32 bits en alguna dirección de la memoria global, agrega un entero a ella, y escribe el resultado de nuevo en el misma dirección. La operación es atómica en el sentido de que está garantizado para llevar a cabo sin interferencias de otros hilos. En otras palabras, ningún otro hilo puede acceder a esta dirección

hasta que se complete la operación. Las operaciones atómicas sólo funcionan con enteros de 32 bits con signo y sin signo.

2.4.2 Tiempo de ejecución en la CPU.

El componente de tiempo de ejecución de la CPU sólo puede ser utilizado por sus funciones propias. Este componente proporciona funciones para controlar:

- La administración del dispositivo,
- La administración de contexto,
- La administración de memoria,
- La administración de módulos de código,
- Control de ejecución,
- La administración de referencias de texturas,
- Interoperabilidad con OpenGL y Direct3D

Se compone de dos API que son mutuamente excluyentes, una aplicación debe utilizar uno u otro:

- Uno de bajo nivel llamado *driver* (conductor).
- Uno de alto nivel llamado *runtime* (tiempo de ejecución), que se implementa en la parte superior del conductor API de CUDA.

El tiempo de ejecución CUDA facilita la gestión del dispositivo al proporcionar el código de inicialización implícito al administrador de contexto y administración de módulos. El código C de la CPU generado por **nvcc** se basa en el tiempo de ejecución de CUDA, por lo que las aplicaciones que enlazan con este código deben utilizar el tiempo de ejecución API de CUDA.

En contraste, el conductor API de CUDA requiere más código, es más difícil de programar y depurar, pero ofrece un mayor nivel de control y es independiente del lenguaje, ya que sólo se ocupa de los objetos *cubin*. En particular, es más difícil de configurar y poner en marcha los *kernel* utilizando el driver API de CUDA, ya que la

configuración de ejecución y los parámetros del *kernel* se debe especificar con llamadas a la función explícita en lugar de la sintaxis de configuración de ejecución.

El conductor API de CUDA se proporciona a través de la biblioteca dinámica **cuda** y todos sus puntos de entrada tienen el prefijo **.cu**.

El tiempo de ejecución API de CUDA se proporciona a través de la biblioteca dinámica **cudart** y todos sus puntos de entrada tienen el prefijo **cuda**.

2.5 Procedimiento CUDA en la paralelización de programas

Un procedimiento es la acción de proceder o el método de ejecutar algunas cosas. Se trata de una serie común de pasos definidos, que permiten realizar un trabajo de forma correcta.

El modelo de computación vincula el término de procesador SIMT con un hilo y cada multiprocesador con un bloque. Normalmente para llevar a cabo una implementación en CUDA se parte de una estrategia de solución en CPU.

Habrá ocasiones en que la solución CPU sea fácilmente extensible a CUDA, sin embargo en otras se deberá estudiar su viabilidad. Las estructuras de datos en CPU deben adaptarse a otras más simples tipo matriz o vector, para hacerlas compatibles con las de GPU.

Posteriormente se sigue una serie de pasos, donde se declaran las estructuras de datos en GPU y se habilita memoria para ellas. El siguiente paso consiste en traspasar los datos desde CPU a GPU, para posteriormente repartir estos datos de entrada entre los diferentes hilos (procesadores) y bloques (multiprocesadores). Todos los procesadores SIMT ejecutan el mismo trozo de código pero con diferentes hilos. Una vez terminado el proceso, los resultados se devuelven a la CPU. A continuación se presenta el procedimiento para el uso de la arquitectura CUDA.

1. Definir y reservar memoria para estructuras de datos en GPU.

```
cudaMalloc((void **)*nombre_variable, cantidad_de_bytes)
```

2. Copiar Datos de CPU a GPU.

```
cudaMemcpy (destino, fuente, cant, CudaMemcpyHostToDevice);
```

3. Definir las variables número de hilos y número de bloques.

4. Ejecutar la función de código en cada hilo.

5. Copiar los datos de GPU a CPU.

```
cudaMemcpy (destino, fuente, cant, CudaMemcpyDeviceToHost);
```

6. Liberar las estructuras de datos en la GPU.

```
cudaFree(nombre_variable)
```

2.6 Consideraciones finales del Capítulo.

Luego de abordarse las características de la nueva tecnología CUDA en el capítulo anterior, en este se caracterizó la implementación del hardware además de la API y algunos componentes esenciales, para finalmente establecer un procedimiento que permite seguir un proceso de trabajo para el uso de una aplicación CUDA.

Capítulo 3. Programar con CUDA

En este capítulo se abordarán los mecanismos para la instalación y configuración para el uso de CUDA en su PC. Algunas especificaciones técnicas y finalmente se procederá a la realización de varios ejemplos CUDA.

3.1 Requisitos para la Instalación

Para instalar CUDA en su computadora son necesarios una serie de requisitos:

- Tarjeta gráfica de NVIDIA compatible con CUDA, funciona en todas las GPU NVIDIA de la serie G8X en adelante, incluyendo GeForce, Quadro y la línea Tesla. NVIDIA afirma que los programas desarrollados para la serie GeForce 8 también funcionarán sin modificaciones en todas las futuras tarjetas NVIDIA, gracias a la compatibilidad binaria (Zeller, 2008). En el Anexo 1 puede consultar el listado actual de las tarjetas soportadas, además http://www.nvidia.com/object/cuda_gpus.html (NVIDIA_Corporation, 2012c)
- Software de NVIDIA para CUDA que incluye *Driver* CUDA, *Toolkit* y SDK, todos deben coincidir en la versión y pueden ser descargados de http://developer.nvidia.com/cuda_downloads/. Los softwares a descargar deben estar de acuerdo al sistema operativo que el usuario tenga disponible en su computadora. NVIDIA ofrece variantes para Windows, Linux, o Mac Os y para arquitecturas de 32 o 64 bits.

3.2 Pasos para la instalación de CUDA

Luego de tener disponibles los elementos anteriores, se procederá a la instalación en el orden siguiente (NVIDIA_Corporation, 2012b):

1- Instalación del Driver.

2- Instalación del Toolkit para CUDA que contiene las herramientas necesarias para compilar un programa CUDA, medir el rendimiento y la ocupación de la GPU.

3- Instalación del SDK que contiene ejemplos y documentación sobre CUDA.

4- Instalar VisualStudio 2005,2008 o 2010: Del VisualStudio sólo es necesario el compilador de C/C++ (cl.exe y sus componentes).

5- Configuración para el compilador ***nvcc***: En Windows debemos incluir el compilador del VisualStudio en la variable PATH del sistema con el objetivo de que el compilador de CUDA (***nvcc***) lo utilice para compilar los programas. Para modificar la variable PATH: hacer clic con el botón derecho del ratón sobre mi PC y luego en propiedades, ahora se selecciona la pestaña "Opciones avanzadas" y en ella se hace clic en "Variables de entorno", en la sección "Variables del sistema" se selecciona la variable PATH y luego se da clic en "modificar", modificando el campo "Valor de variable" con la ruta completa del compilador de Visual Studio. Una vez hecho todo esto el sistema estará preparado para compilar.

Estas instalaciones se realizan según el sistema operativo que se disponga, consultar (NVIDIA_Corporation, 2012c) para el proceso de instalación, en este trabajo se usó Windows 7 de 64 bits.

3.3 Crear un nuevo proyecto CUDA en VisualStudio

Para crear un nuevo proyecto CUDA existe un *plugin* (CUDA VS Winzard) es una pequeña de aplicación que añadirá un nuevo tipo de proyecto CUDA en VisualStudio, soporta aplicaciones Win32 y Win64. La figura 14 muestra el plugin incorporado en VisulStudio. Se pueden descargar en:

http://sourceforge.net/projects/cudavswizard/files/cudavswizard/CUDA_VS_Wizard_2.0%20Release/CUDA_VS_Wizard_W64.2.0.1.zip/download.

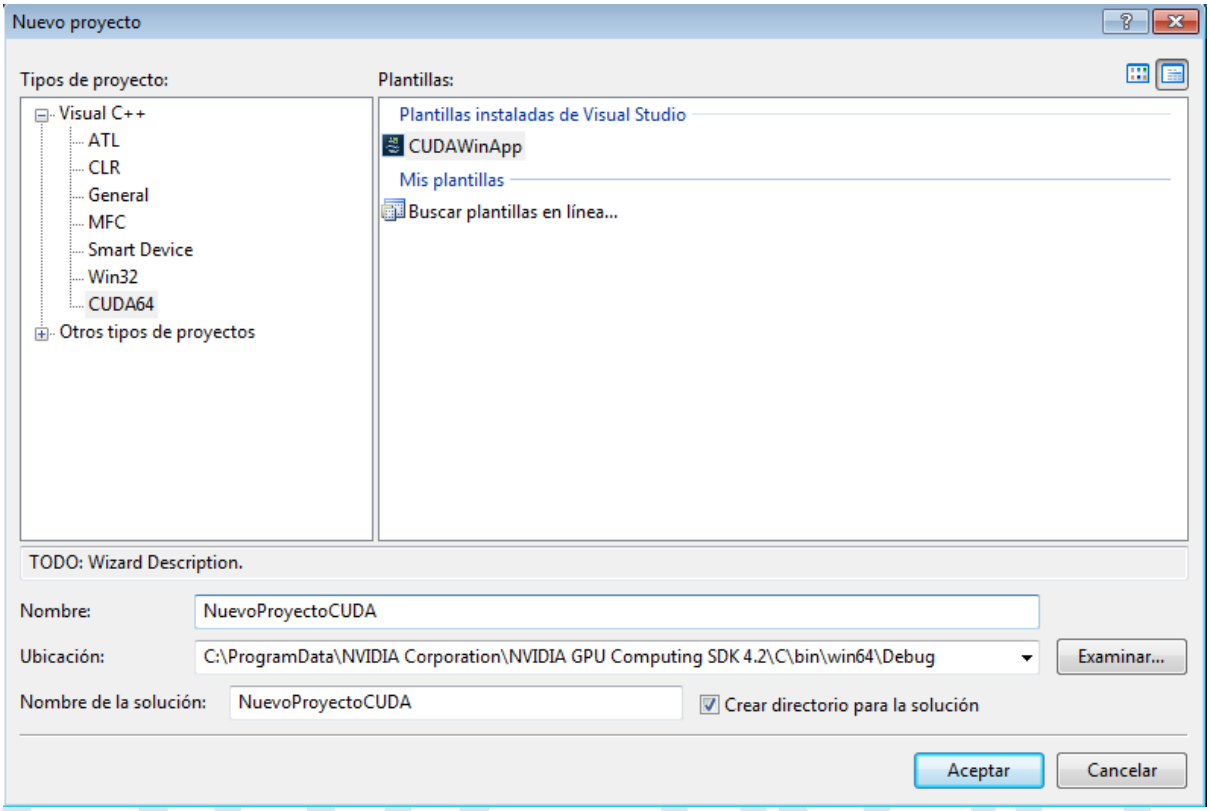


Figura 14. Ejemplo del plugin instalado

3.4 Especificaciones generales

- El número máximo de hilos por bloque es 512.
- El tamaño máximo de las dimensiones x-, y- y z- de un bloque de hilo es de 512, 512 y 64 respectivamente.
- El tamaño máximo de cada dimensión de una malla de bloques de hilos es 65535.
- El tamaño de un *warp* es 32 hilos.
- El número de registros por multiprocesador es 8192.
- La cantidad de memoria compartida disponible por multiprocesador es 16 KB organizados en 16 bancos.
- La cantidad de memoria constante disponible es 64 KB con una caché trabajando en grupos de 8 KB por multiprocesador.

- La caché de trabajo creados para una dimensión de texturas es de 8 KB por multiprocesador.
- El número máximo de bloques que pueden ejecutarse simultáneamente en un multiprocesador es de 8.
- El número máximo de *warps* que pueden ejecutarse simultáneamente en un multiprocesador es de 24.
- El número máximo de hilos que pueden ejecutarse simultáneamente en un multiprocesador es de 768.
- Para una referencia de textura unida a una matriz unidimensional CUDA, la anchura máxima es de 2^{13-}.
- Para una referencia de textura unida a una matriz de dos dimensiones CUDA, la anchura máxima es de 2^{16} y la altura máxima es de 2^{15-}.
- Para una referencia de textura unida a la memoria lineal, la anchura máxima es de 2^{27-}.
- El límite en el tamaño de un *kernel* es de 2 millones de instrucciones nativas
- Cada multiprocesador se compone de ocho procesadores, de modo que un multiprocesador es capaz de procesar los 32 hilos de un *warp* en cuatro ciclos de reloj.

3.5 Práctica de ejemplos usando CUDA

En este epígrafe se mostrarán ejemplos de código fuente usando CUDA para la solución de algunos problemas, permitiendo la familiarización de esta nueva tecnología.

3.5.1 Ejemplos de Hola mundo

La figura 15 muestra un ejemplo de Hola Mundo muy sencillo donde no se utiliza la GPU, mientras que en la figura 16 hay dos adicciones notables a este programa original, una es la función vacía llamada `kernel()` calificada con `_global_` y la otra una llamada a la función con la sintaxis `<<<1,1>>>`. Este segundo código

imprimirá el resultado usando la GPU al ser invocado desde la CPU el Kernel correspondiente, en este caso para una malla de 1sólo hilos.

```
#include <stdio.h>

int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

Figura 15 Hola Mundo con CPU

```
#include <iostream>

__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Figura 16. Hola mundo con GPU

3.5.2 Ejemplo sumar dos números

Este ejemplo retoma el código anterior pero en este caso se declara una nueva función `add()` pasándole dos parámetros. Además se usa la función `cudaMalloc()` para gestionar la memoria de `dev_c` en la GPU, la función `cudaMemcpy()` para transferir los datos del dispositivo a la GPU y finalmente `cudaFree ()` para liberar la memoria que se reservó anteriormente.

```
__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}

int main( void ) {
    int c;
    int *dev_c;
 ( cudaMalloc( (void**)&dev_c, sizeof(int) ) );
add<<<1,1>>>( 2, 7, dev_c );
 ( cudaMemcpy( &c, dev_c, sizeof(int), cudaMemcpyDeviceToHost ) );
   printf( "2 + 7 = %d\n", c );
   cudaFree( dev_c );
return 0;
}
```

Figura 17. Suma de dos números

3.5.3 Ejemplos suma de vectores

A continuación se muestra un simple ejemplo que tiene dos listas de números, se desea sumar los elementos correspondientes de cada lista y mostrar los resultados en una lista resultante, en la figura 18 se muestra el flujo de este proceso (Jason and Edward, 2010).

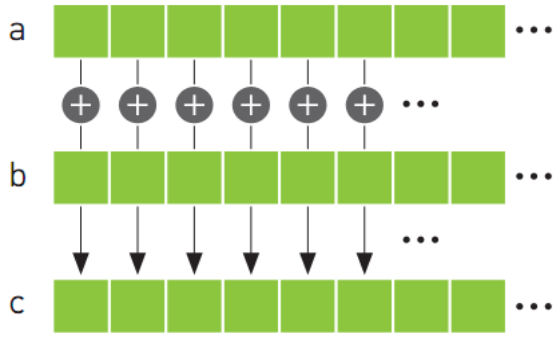


Figura 18. Ejemplo suma de vectores

El código que aparece a continuación, figura 19, es una manera tradicional usando código C en la CPU.

```
#define N   10

void add( int *a, int *b, int *c ) {
    int tid = 0;    // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;   // we have one CPU, so we increment by one
    }
}

int main( void ) {
    int a[N], b[N], c[N];

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add( a, b, c );

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}
```

Figura 19. Suma de vectores en CPU

Nosotros podemos adaptar esta misma adición muy similar en una GPU como se muestra en la figura 20, pero agregando el método `add()` como una función en el dispositivo, esto se hace anteponiendo `_global_` delante de la función ya que esto indica que será ejecutado en el dispositivo e invocado desde el host. Para obtener el ID de cada hilo se usa `int i = threadIdx.x + (blockIdx.x * blockDim.x)`. A diferencia del código del host anterior debe reservarse un espacio de memoria para el host y el dispositivo usando la función `cudaMalloc ()` y realizar la transferencia de los datos que serán procesados por la GPU con `cudaMemcpy ()`. A la hora de invocar el kernel hay que tener en cuenta el tamaño

de malla y del bloque, como se explicó en epígrafes anteriores es conveniente que el tamaño de los bloques sea múltiplo de 32, por lo que se decide un `MAX_BLOCKSIZE = 256`, luego se devuelven los resultados al host solo cambiando el tipo de transferencia `DevicetoHost y` finalmente se liberan los espacios de memoria reservados anteriormente con `cudaFree ()` y `Free ()`.

```
#include <cuda.h>
#include <malloc.h>
//Este ejercicio realiza la suma de dos vectores.
//Código device
// Kernel
__global__ void VecAdd_kernel(float *d_A, float * d_B, float *d_C, int N)
{
int i = threadIdx.x + (blockIdx.x * blockDim.x);
if (i < N){
  d_C[i] = d_A[i] + d_B[i];
 }
}
//Código host
int main(){
 int N = 1000;
 size_t size = N * sizeof(float);
 int MAX_BLOCKSIZE = 256;
 //Reservar memoria en host
 float* h_A = (float*)malloc(size);
 float* h_B = (float*)malloc(size);
 float* h_C = (float*)malloc(size);
// Asignarle valores a los vectores
...
// Reserva memoria en dispositivo
float *d_A; cudaMalloc((void **) &d_A, size);
float *d_B; cudaMalloc((void **) &d_B, size);
float *d_C; cudaMalloc((void **) &d_C, size);
//Copiar datos a la GPU
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
//Topología bloques, grid
int BLOCKSX=ceil((float) N/MAX_BLOCKSIZE); //3.90 -> 4
dim3 blocksize(MAX_BLOCKSIZE,1);
dim3 gridsize(BLOCKSX,1);
 //Llamada al kernel
VecAdd_kernel<<<gridsize, blocksize>>>(d_A, d_B, d_C, N);
//Obtener resultados de la GPU
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
//imprimir resultados
...
//Liberar memoria GPU
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
//Liberar memoria CPU
free(h_A);
free(h_B);
free(h_C); }
```

Figura 20. Suma de vectores en GPU

3.5.4 Ejemplo hallar el cuadrado de los elementos de un arreglo

```
#include "stdafx.h"
#include <stdio.h>
#include <cuda.h>
// kernel que se ejecuta en el dispositivo.
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}

// Código a ejecutar en el host.
int main(void)
{
  float *a_h, *a_d;  // Puntero al arreglo en el host y en el dispositivo
  const int N = 10;  // Numero de elementos en el arreglo
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        // Arreglo asignado en host
  cudaMalloc((void **) &a_d, size);   // Arreglo asignado en el dispositivo
  // Inicializa los arreglos en el host y los copia al dispositivo
  for (int i=0; i<N; i++)
    a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  // Haciendo el cálculo en el dispositivo
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // Recupera los resultados desde el dispositivo
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // Imprime los resultados
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // libera las memorias
  free(a_h); cudaFree(a_d);
}
```

Figura 21. Cuadrado de los elementos de un arreglo

3.5.5 Ejemplo multiplicación de matrices

La Tarea de calcular el producto C de dos matrices A y B de dimensiones (*wA, hA*) y (*wB, wA*), respectivamente, se divide entre varios hilos de la siguiente manera:

- Cada bloque de hilo es el responsable de calcular un cuadrado sub-matriz (*Csub*) de C.
- Cada hilo dentro del bloque es responsable de calcular uno de los elementos de (*Csub)*.

La dimensión *block_size* de *Csub* se elige igual a 16, por lo que el número de hilos por bloque es un múltiplo del tamaño del *warp* y se mantiene por debajo del máximo número de hilos por bloque. Como se ilustra en la figura 18.

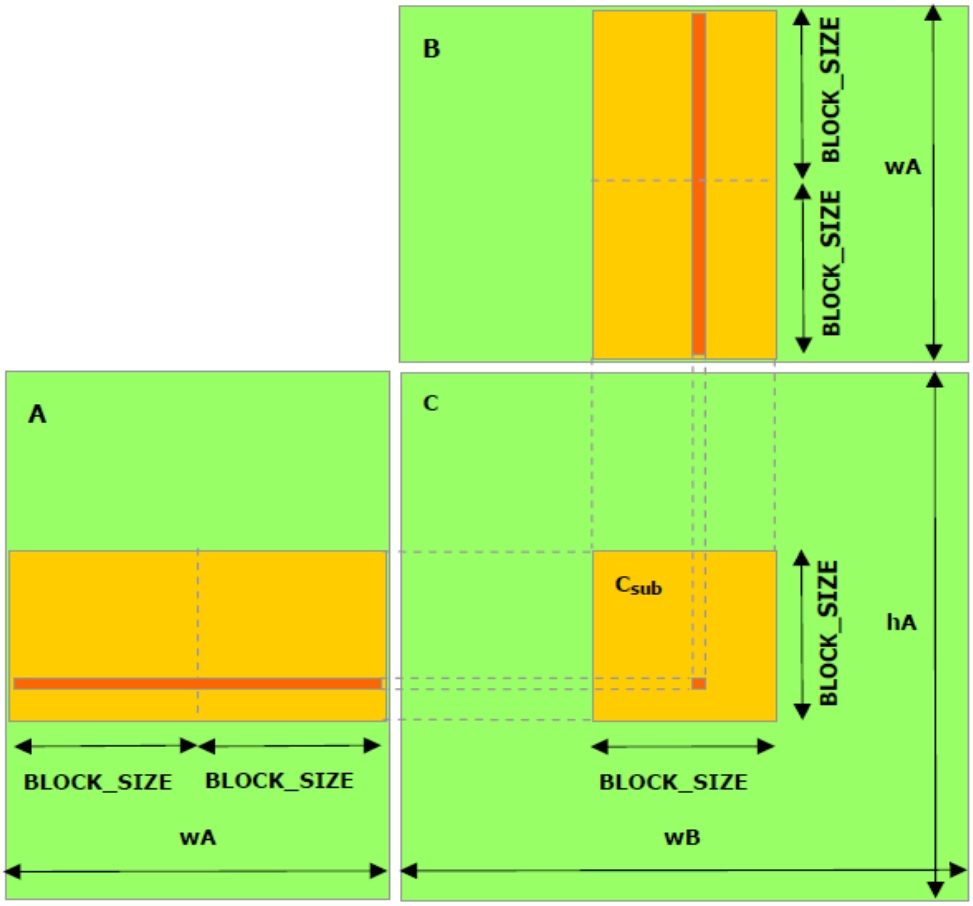


Figura 22. Multiplicación de Matices

Csub es igual al producto de dos matrices rectangulares: la submatriz de A de dimensión (*wA, block_size*) que tiene los índices de la misma línea que *Csub*, y la sub-matriz de B de la dimensión (*block_size, wA*), que tiene los índices de la misma columna que *Csub*. Con el fin de encajar en los recursos del dispositivo, estas dos matrices rectangulares están divididas en tantas matrices cuadradas de dimensión *block_size* según sea necesario y *Csub* se calcula como la suma de los productos de estas matrices cuadradas. Cada uno de estos productos se realiza cargando las primera dos matrices cuadradas correspondiente desde la memoria global hasta memoria compartida con un hilo de carga de un elemento de cada matriz, haciendo

que cada hilo calcule un elemento del producto. Cada hilo se acumula en el resultado de cada uno de estos productos en un registro y una vez hecho esto escribe el resultado en la memoria global.

Al restringir el cálculo de esta manera, se aprovecha la memoria compartida y se ahorra ancho de banda de memoria global, ya que A y B se leen de la memoria global únicamente (*wA* / *block_size*) veces. Sin embargo, este ejemplo ha sido escrito para ilustrar con claridad diversos principios de la programación CUDA, no con el objetivo de proporcionar un núcleo de alto rendimiento para la multiplicación de la matriz genérica y no debe interpretarse como tal. A continuación se muestrea el código fuente de este ejemplo:

La figura 23 muestra el código para el host, en ella se muestra la declaración de la función `Muld ()` que se ejecutará en el dispositivo, además la función `Mul ()` donde se reservar memoria para (A, B y C), el tipo de transferencia de datos que se va a hacer, y liberar la memoria que se reservó anteriormente en el dispositivo.

La figura 24 muestra el código para el dispositivo donde se encuentra la implementación de la función `Muld ()` llamada desde el host.

```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
//   hA is the height of A
//   wA is the width of A
//   wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB,
         float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

Figura 23. Código para el Host en la multiplicación de matrices

```
// Device multiplication function called by Mul()
// Compute C = A * B
//   wA is the width of A
//   wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;
```

```
    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Figura 24. Código para la GPU en la multiplicación de matices

El código fuente contiene dos funciones:

- **Mul ()** es una función del host que sirve como un contenedor para **Muld ()**, toma como entrada: dos punteros a la memoria del host que apunta a los elementos de A y B, altura y ancho de A y el ancho de B y un puntero a la memoria del host donde esos puntos de C deben ser escritos. Esta función

además realiza las siguientes operaciones: Se asigna suficiente memoria global para almacenar A, B y C usando **cudaMalloc ()**, se copia A y B desde la memoria del host a la memoria del dispositivo usando **cudaMencpy ()**, se llama a **Muld ()** para calcular C en el dispositivo, se copia C desde la memoria global a la memoria del dispositivo **cudaMencpy ()** y liberar la memoria global asignada para A, B y C usando **cudaFree ().**

- **Muld ()** es un kernel que ejecuta la matriz de multiplicación en el dispositivo, tiene la misma entrada que **Mul ()** salvo que los punteros apuntan a la memoria del dispositivo en lugar de la memoria del host. Para cada bloque, **Muld ()** recorre todas las sub-matrices de A y B para calcular *Csub.* En cada iteración: Se carga una sub-matriz de A y una de B desde la memoria global a la memoria compartida, se sincroniza para asegurarse de que ambas sub-matrices están cargadas completamente de todos los hilos dentro del bloque, se calcula el producto de las dos sub-matrices y lo agrega al producto que se obtiene durante la iteración anterior, se sincroniza de nuevo para asegurarse de que el producto de las dos sub-matrices se realizan antes de comenzar la siguiente iteración, una vez que todas las sub-matrices han sido manipuladas, *Csub* es totalmente calculada y **Muld ()** escribe en la memoria global. **Muld ()** esta escrito para maximizar el rendimiento de la memoria

En efecto, suponiendo que *wA* y *wB son* múltiplos de 16 la memoria global está asegurada porque A, B y C son todos múltiplos de BLOCK_SIZE que es igual a 16. Tampoco hay conflicto de memoria compartida del banco ya que para cada half-warp, ***ty*** y ***k*** son los mismos para todos los hilos y ***tx*** varía de 0 a 15, por lo que cada hilo tiene acceso a un banco diferente para el acceso a la memoria **As [ty] [tx], Bs [ty] [tx]**, y **Bs [k] [tx]** y el mismo banco para el acceso a la memoria **As [ty] [k].**

3.6 Consideraciones finales del capítulo

En este capítulo se ha visto que con un sencillo proceso de instalación y configuración puede comenzarse a usar CUDA, a partir de diferentes ejemplos sencillos se les demuestra a los programadores el uso del procedimiento definido

para programar usando CUDA y el uso de las extenciones del lenguaje C para codificar algoritmos en GPU de NVIDIA.

Conclusiones

- Se presentó las características de la nueva tecnología CUDA y algunas de sus aplicaciones actuales.
- Se caracterizó la implementación del hardware e interfaz de programación que ofrece CUDA sobre las GPU, para adaptar sus funciones y modo de programación al lenguaje estándar de C/C++.
- Se estableció un procedimiento de trabajo para el uso de CUDA en la paralelización de programas.
- Se probaron algunos ejemplos haciendo uso del procedimiento y la tecnología en la solución de algunos problemas.

Recomendaciones

- Explorar las facilidades adicionales que ofrece CUDA y el uso de sus herramientas para comprobar rendimiento y ocupación.

Referencias bibliográficas

ALMEIDA, F., GIMÉNEZ, D., MANTAS, J. M. & VIDAL, A. 2008. *Introducción a la programación paralela*.

ASANOVIC, K., BODIK, R., CATANZARO, B., GEBIS, J., HUSBANDS, P., KEUTZER, K., PATTERSON, D., PLISHKER, W., SHALF, J., WILLIAMS, S. & YELICK, K. 2006. The Landscape of Parallel Computing Research: A View from Berkely. *In:* CALIFORNIA, U. O. (ed.). Berkely: UCB/EECS-2006-183.

ASANOVIC, K., BODIK, R., DEMMEL, J., KEAVENY, T., KEUTZER, K., KUBIATOWICZ, J., MORGAN, N., PATTERSON, D., SEN, K., WAWRZYNEK, J., WESSEL, D. & YELICK, K. 2009. *A View of the Parallel Computing Landscape.*

ASTLE, D. & HAWKING, K. 2004. *Beginning OpenGL Game Programing*.

BLYTHE, D. 2006. *The Direct3D 10 system.* .

BOGGAN, S. & PRESSEL, D. 2007. *GPUs: An Emerging Platform for General-Purpose Computation.*

CABALLERO DE GEA, D. L. 2011. Aceleración de Aplicaciones Científicas mediante GPUs.

CHENG, M. M. C., CUDA, G., BUNIMOVICH, Y. L., GASPARI, M., HEATH, J. R., HILL, H. D., MIRKIN, C. A., NIJDAM, A. J., TERRACCIANO, R. & THUNDAT, T. 2006. Nanotechnologies for biomolecular detection and medical diagnostics. *Current Opinion in Chemical Biology*, 10.

ENGEL, W. 2003. *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*, Wordware Publishing

HARISH, P. & NARAYANAN, P. 2007. Accelerating large graph algorithms on the GPU using CUDA. *High Performance Computing–HiPC 2007*.

HOEGER, H. 1997. Introducción a la Computación Paralela. *Universidad de Los Andes*.

JASON, S. & EDWARD, K. 2010. *CUDA BY EXAMPLE An Introduction to General-Purpose GPU Programing*

LUEBKE, D. 2008. CUDA Fundamentals. Beyond Programmable Shading: In Action.

MCREYNOLDS, T. & BLYTHE, D. 2005. *Advanced Graphics Programming Using OpenGL*.

MORA, G. 2008. Introducción a CUDA. Basado en la transparencia de: David Kirk (NVIDIA), Wen-mei Hwu (Univ. of IIinois)

NVIDIA_CORPORATION 2008. Scalable Parallel Programming whit CUDA. ACM Queue ed.

NVIDIA_CORPORATION. 2011. *NVIDIA CUDA C Programming Guide v4.0* [Online]. Available: http://www.nvidia.com [Accessed Nov 2011].

NVIDIA_CORPORATION 2012a. CUDA C Best Practices Guide.

NVIDIA_CORPORATION 2012b. GETTING STARTED WITH CUDA SDK SAMPLES.

NVIDIA_CORPORATION 2012c. NVIDIA CUDA GETTING STARTED GUIDE FOR MICROSOFT WINDOWS, Installation and Verification on Windows

NVIDIA_CORPORATION, C. 2007. NVIDIA CUDA Compute Unified Device Architecture Programming Guide v1.0.

ORTEGA, J., ANGUITA, M. & PRIETO, A. 2005. Arquitectura de computadores. *Thomson editores Spain Parainfo SA*.

PEÑA, A., CLAVER, J., SANJUAN, A. & ARNAU, V. Análisis paralelo de secuencias de ADN mediante el uso de GPU y CUDA.

REGIS, O., MARTÍNEZ, C. & CALVET, H. C. COMPUTACIÓN PARALELA.

RODRIGUEZ, F. J. J., MARTÍN, G. E., GARCÍA, M. J. A., CASTRO, B. R. & LÓPEZ, V. F. M. Computación en procesadores gráficos. Programación con CUDA.

RONDÁN, A. 2009. CUDA:MODELO DE PROGRAMACIÓN,.

SANTAMARÍA, J., ESPINILLA, M., RIVERA, A. & ROMERO, S. 2010. Potenciando el aprendizaje proactivo con ILIAS&WebQuest: aprendiendo a paralelizar algoritmos con GPUs. *actas de XVI Jornadas de Enseñanza Universitaria de la Informática (JENUI 2010).*

SCHWARZ, M. & STAMMINGER, M. 2009. *Fast GPU-based Adaptive Tessellation with CUDA.*

SENGUPTA, S., HARRIS, M., ZHANG, Y. & OWENS, J. D. 2007. Scan primitives for GPU computing. *In:* ASSOCIATION, E. (ed.).

VALLE, Y. 2011. *Modelo de Representación de Superficies de Terreno para su Visualización en Tres Dimensiones.* UCI.

ZELLER, C. 2008. Tutorial CUDA. NVIDIA Developer Technology.

Anexos

Anexos 1. Tarjetas soportadas

NVIDIA GeForce	NVIDIA GeForce Mobile	NVIDIA Quadro	NVIDIA Quadro Mobile	Nvidia Tesla
GeForce GTX 295	GeForce GTS 360M	Quadro FX 5800	Quadro NVS 360M	Tesla S1070
GeForce GTX 285	GeForce GTX 280M	Quadro FX 5600	Quadro NVS 140M	Tesla C1060
GeForce GTX 280	GeForce GTX 260M	Quadro FX 4800	Quadro NVS 135M	Tesla C870
GeForce GTX 275	GeForce GTS 260M	Quadro FX 4700 X2	Quadro NVS 130M	Tesla D870
GeForce GTX 260	GeForce GTS 250M	Quadro FX 4600		Tesla S870
GeForce GTS 250	GeForce GT 330M	Quadro FX 3700		
GeForce GT 220	GeForce GT 240M	Quadro FX 1800		
GeForce G210	GeForce GT 230M	Quadro FX 1700		
GeForce 9800 GX2	GeForce GT 220M	Quadro FX 570		
GeForce 9800 GTX+	GeForce G210M	Quadro FX 370		
GeForce 9800 GTX	GeForce 9800M GTX	Quadro NVS 290		
GeForce 9800 GT	GeForce 9800M GTS	Quadro FX 3600M		
GeForce 9600 GSO	GeForce 9800M GT	Quadro FX 1600M		
GeForce 9600 GT		Quadro FX 770M		
GeForce 9500 GT	GeForce 9700M GTS	Quadro FX 570M		
GeForce 9400 GT	GeForce 9700M GT	Quadro FX 370M		
GeForce 9400 mGPU	GeForce	Quadro Plex 1000 Model IV		
GeForce 9300 mGPU		Quadro Plex 1000 Model S4		

GeForce 8800 Ultra GeForce 8800 GTX GeForce 8800 GTS GeForce 8800 GT GeForce 8800 GS GeForce 8600 GTS GeForce 8600 GT GeForce 8600 mGT GeForce 8500 GT GeForce 8400 GS GeForce 8300 mGPU GeForce 8200 mGPU GeForce 8100 mGPU	GT 130M GeForce 9650M GT GeForce 9650M GS GeForce 9600M GT GeForce 9600M GS GeForce 9500M GS GeForce 9500M G GeForce 9400M G GeForce 9300M GS GeForce 9300M G GeForce 9200M GS GeForce 9100M G GeForce 8800M GTS GeForce 8700M GT GeForce 8600M GT GeForce 8600M GS GeForce 8400M GT GeForce 8400M GS GeForce 8400M G GeForce 8200M G			

Tabla 3. Tarjetas soportadas

Anexo 2. Capacidad de cálculo, número de multiprocesadores y core

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
GeForce GTX 560 Ti	2.1	8	384
GeForce GTX 460	2.1	7	336
GeForce GTX 470M	2.1	6	288
GeForce GTS 450, GTX 460M	2.1	4	192
GeForce GT 445M	2.1	3	144
GeForce GT 435M, GT 425M, GT 420M	2.1	2	96
GeForce GT 415M	2.1	1	48
GeForce GTX 580	2.0	16	512
GeForce GTX 570, GTX 480	2.0	15	480
GeForce GTX 470	2.0	14	448
GeForce GTX 465, GTX 480M	2.0	11	352
GeForce GTX 295	1.3	2x30	2x240
GeForce GTX 285, GTX 280, GTX 275	1.3	30	240
GeForce GTX 260	1.3	24	192
GeForce 9800 GX2	1.1	2x16	2x128
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512, GTX 285M, GTX 280M	1.1	16	128
GeForce 8800 Ultra, 8800 GTX	1.0	16	128
GeForce 9800 GT, 8800 GT,	1.1	14	112

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
GTX 260M, 9800M GTX			
GeForce GT 240, GTS 360M, GTS 350M	1.2	12	96
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTS 260M, GTS 250M, 9800M GT	1.1	12	96
GeForce 8800 GTS	1.0	12	96
GeForce GT 335M	1.2	9	72
GeForce 9600 GT, 8800M GTS, 9800M GTS	1.1	8	64
GeForce GT 220, GT 330M, GT 325M, GT 240M	1.2	6	48
GeForce 9700M GT, GT 230M	1.1	6	48
GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS	1.1	4	32
GeForce 210, 310M, 305M	1.2	2	16
GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU, 9300 mGPU, 8300 mGPU, 8200 mGPU, 8100 mGPU, G210M, G110M	1.1	2	16

GeForce 9300M GS, 9200M GS, 9100M G, 8400M G, G105M	1.1	1	8
Tesla C2070, M2070, C2050, M2050	2.0	14	448
Tesla S1070	1.3	4x30	4x240
Tesla C1060, M1060	1.3	30	240
Tesla S870	1.0	4x16	4x128
Tesla D870	1.0	2x16	2x128
Tesla C870	1.0	16	128
Quadro 2000	2.1	4	192
Quadro 600	2.1	2	96
Quadro Plex 7000	2.0	2x16	512
Quadro 6000	2.0	14	448
Quadro 5000	2.0	11	352
Quadro 5000M	2.0	10	320
Quadro 4000	2.0	8	256
Quadro Plex 2200 D2	1.3	2x30	2x240
Quadro Plex 2100 D4	1.1	4x14	4x112
Quadro Plex 2100 Model S4	1.0	4x16	4x128
Quadro Plex 1000 Model IV	1.0	2x16	2x128
Quadro FX 5800	1.3	30	240
Quadro FX 4800	1.3	24	192
Quadro FX 4700 X2	1.1	2x14	2x112

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
Quadro FX 3700M, FX 3800M	1.1	16	128
Quadro FX 5600	1.0	16	128
Quadro FX 3700	1.1	14	112
Quadro FX 2800M	1.1	12	96
Quadro FX 4600	1.0	12	96
Quadro FX 1800M	1.2	9	72
Quadro FX 3600M	1.1	8	64
Quadro FX 880M, NVS 5100M	1.2	6	48
Quadro FX 2700M	1.1	6	48
Quadro FX 1700, FX 570, NVS 320M, FX 1700M, FX 1600M, FX 770M, FX 570M	1.1	4	32
Quadro FX 380 LP, FX 380M, NVS 3100M, NVS 2100M	1.2	2	16
Quadro FX 370, NVS 290, NVS 160M, NVS 150M, NVS 140M, NVS 135M, FX 360M	1.1	2	16
Quadro FX 370M, NVS 130M	1.1	1	8

Tabla 4. Capacidades de cálculo, números de multiprocesadores y cores

Anexo 3 Tipo de vectores y alineación

Type	Alignment
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 if sizeof(long) is equal to sizeof(int), 8, otherwise
long2, ulong2	8 if sizeof(long) is equal to sizeof(int), 16, otherwise
long3, ulong3	4 if sizeof(long) is equal to sizeof(int), 8, otherwise
long4, ulong4	16
longlong1, ulonglong1	8
longlong2, ulonglong2	16
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16

Tabla 5. Tipos de vectores