Segunda Parte: TECNOLOGÍA CUDA

(compute unified device architecture)

I. CUDA

CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento del rendimiento del sistema.

Los sistemas informáticos están pasando de realizar el "procesamiento central" en la CPU a realizar "coprocesamiento" repartido entre la CPU y la GPU. Para posibilitar este nuevo paradigma computacional, NVIDIA ha inventado la arquitectura de cálculo paralelo CUDA, que se incluye en las GPUs GeForce, ION, Quadro y Tesla.

CUDA son las siglas de (***C****ompute* ***U****nified* ***D****evice* ***A****rchitecture*) que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA que permiten a los programadores usar una variante del lenguaje de programación C para codificar algoritmos en GPUs de NVIDIA.

Por medio de wrappers se puede usar Python, Fortran y Java en vez de C/C++ y en el futuro también se añadirá OpenCL y Direct3D.

Funciona en todas las GPUs NVIDIA de la serie G8X y posteriores, 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.

CUDA intenta explotar las ventajas de las GPUs frente a las CPUs de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un elevado número de hilos simultáneos. Por ello, si una aplicación está diseñada utilizando numerosos hilos que realizan tareas independientes (que es lo que hacen las GPUs al procesar gráficos, su tarea natural), será idónea para ser ejecutada en una GPU que podrá ofrecer un gran rendimiento.

La primera 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 para Windows XP/Vista/7, para Linux 32/64 bits y para Mac OS.

El procesamiento CUDA se realiza ejecutando los siguientes pasos:

1. Se copian los datos de la memoria principal a la memoria de la GPU.
2. La CPU encarga el proceso a la GPU.
3. La GPU lo ejecuta en paralelo en cada núcleo.
4. Se copia el resultado de la memoria de la GPU a la memoria principal.

CUDA presenta ciertas ventajas frente a otros tipos de computación en GPU utilizando APIs gráficas; ya que:

1. Pemite realizar lecturas dispersas: se puede consultar cualquier posición de memoria.

2. Posee memoria compartida: CUDA pone a disposición del programador un área de memoria que se compartirá entre hilos (threads). Dado su tamaño y rapidez puede ser utilizada como caché.

3. Permite lecturas más rápidas desde y hacia la GPU.

4. Ofrece soporte para enteros y operadores a nivel de bit.

Sin embargo también presenta ciertas limitaciones.porque:

1. No se puede utilizar recursividad, punteros a funciones, variables estáticas dentro de funciones o funciones con número de parámetros variable.

2. No está soportado el renderizado de texturas.

3. En precisión simple no soporta números desnormalizados o NaNs

4. Puede existir un cuello de botella entre la CPU y la GPU por los anchos de banda de los buses y sus latencias.

5. Los hilos, por razones de eficiencia, deben lanzarse en grupos de al menos 32, con miles de hilos en total.

CUDA intenta aprovechar el gran paralelismo, y el alto ancho de banda de la memoria en las GPUs en aplicaciones con un gran coste aritmético que compensan el coste de numerosos accesos a memoria principal, lo que podría actuar como cuello de botella.

El modelo de programación de CUDA está diseñado para que se creen aplicaciones que de forma transparente escalen su paralelismo para poder ser ejecutados en un número de núcleos computacionales. Este diseño contiene tres puntos claves, que son la jerarquía de grupos de hilos, las memorias compartidas y las barreras de sincronización.

Un multiprocesador CUDA contiene ocho procesadores escalares, también llamadoa “Cores” o núcleos, dos unidades especiales para funciones trascendentales, una unidad multihilo de instrucciones (Warp Scheduler) y una memoria compartida. El multiprocesador crea y maneja los hilos sin ningún tipo de coste adicional por la planificación, lo cual unido a una rápida sincronización por barreras y una creación de hilos muy ligera, consigue que se pueda utilizar CUDA en problemas de muy baja granularidad, incluso asignando un hilo a un elemento por ejemplo de una imagen (un pixel).

1. MODELO DE MEMORIA CUDA

En CUDA tanto el Host (CPU) como el Device (GPU) tienen memorias independientes.

Para la ejecución de un kernel (sección del programa que se ejecuta en la GPU) en un dispositivo, se necesita asignar memoria en la GPU y transferir los datos apropiados de la memoria del Host a la posición designada de la memoria del dispositivo. Además, del mismo modo, tras la ejecución en el dispositivo, se deben transferir los resultados al host y liberar la memoria de la GPU. CUDA proporciona APIs que realizan estas operaciones.

En la figura 1 se representa la estructura de memoria y las distintas operaciones permitidas.

Hay que observar que disponemos de una memoria Global y otra memoria llamada Constante. En ambas, el Host puede escribir y leer. En cambio la GPU sólo accede a la memoria Constante en modo sólo lectura.

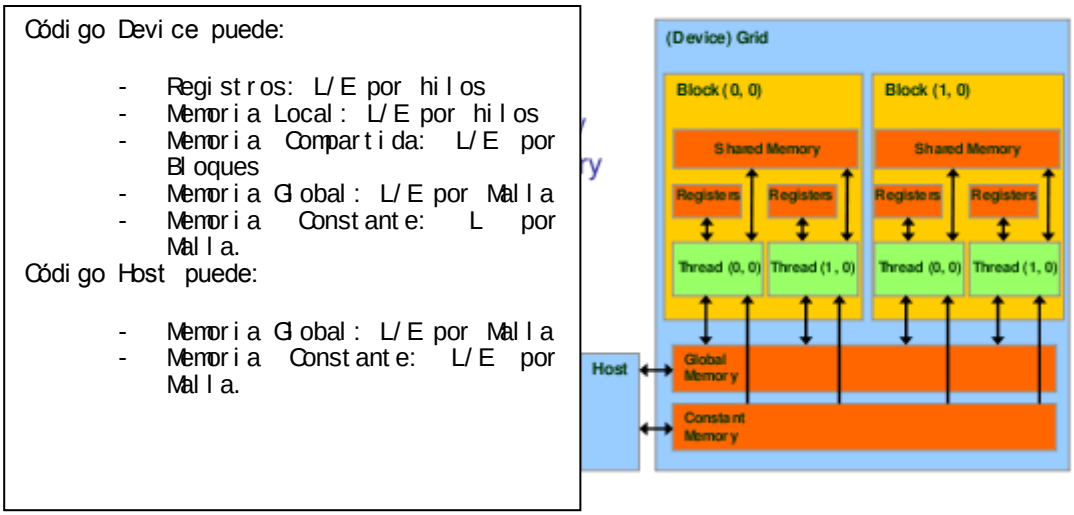

Fig. 1: Modelo de memoria de CUDA.

En el diagrama las flechas indican el tipo de acceso, lectura o escritura o ambos. También indican los dispositivos o las unidades que pueden acceder a cada tipo.

Mediante la función "cudaMalloc()" llamada desde el host se asigna memoria de tipo Global.

Esta memoria podrá ser liberada mediante la función "cudaFree()".

Para intercambiar datos entre las diferentes memorias de la GPU, se usa la función "cudaMemcopy()" que nos permite hacer copias Host-Host, Host-GPU, GPU-Host y GPU-GPU.

La memoria global se implementa mediante DRAM generalmente y es una memoria con tiempos de acceso largos (del orden de centenares de ciclos de reloj) y con un ancho de banda limitado. Por tanto, aunque el disponer de varios hilos para la ejecución de un código, debería aumentar la velocidad en teoría de manera significativa, en realidad lo hace de manera limitada, debido a los tiempos de acceso.

De hecho se puede presentar la situación en que se produzca una congestión en la memoria Global y sólo permita a unos pocos hilos ejecutarse. Para evitar este problema, el modelo CUDA añade una serie de memorias con el objeto de filtrar la mayor parte de peticiones dirigidas a la memoria Global y redirigirlas hacia otro tipo de memorias disminuyendo los tiempos de acceso.

a. TIPOS DE MEMORIA CUDA

Cada dispositivo CUDA tiene varias memorias que pueden ser usadas para aumentar la eficiencia en la ejecución. Se define el CGMA como la razón entre operaciones realizadas y accesos a la memoria Global que se han realizado. El objetivo es hacerlo lo mayor posible; es decir: realizar el mayor número de operaciones aritméticas con el menor número de trasiegos con la memoria Global.

En la figura 2 se representa el modelo de distribución de memorias particular para el modelo GeForce 8800 GTX.

En la parte inferior tenemos la memoria Global y la Constante, ambas son accesibles por parte del Host en modo lectura-escritura. La memoria Constante sólo es accesible en modo lectura por el dispositivo, sin embargo ofrece accesos más rápidos y más paralelos que la Global.

Por encima de los cajetines correspondientes a los hilos encontramos los registros y las memorias compartidas. Los datos almacenados en ambos tipos son accesibles de manera muy rápida. Los registros son asignados a hilos de manera individual; cada hilo sólo puede acceder a sus propios registros. Una función kernel usará los registros para almacenar datos que son usados frecuentemente, pero que son particulares para cada hilo. Las memorias Compartidas (Shared) son asignadas a bloques de hilos; todos los hilos de un bloque pueden acceder a datos en la memoria compartida. El uso de la memoria compartida es esencial para que los hilos cooperen compartiendo datos.

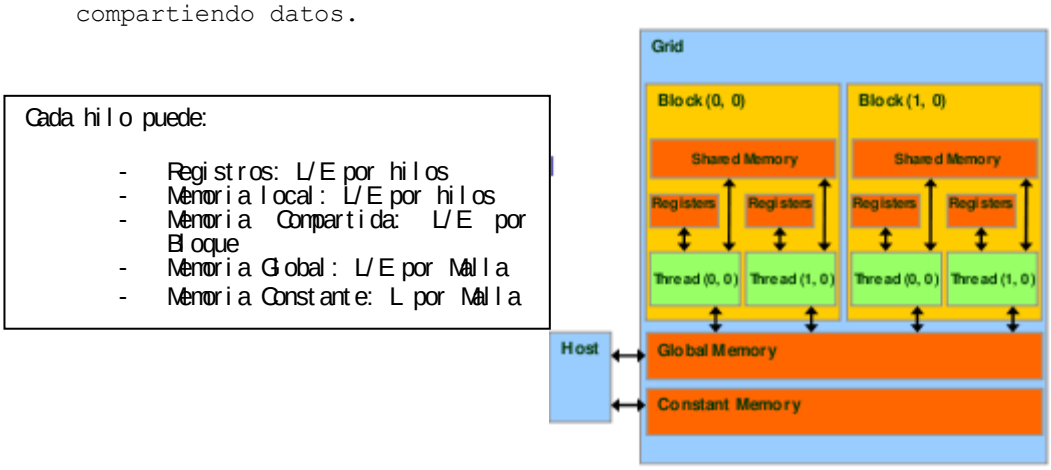

Fig.2: Modelo de memorias del modelo GeForce 8800 GTX.

La tabla 1 muestra la sintaxis CUDA para declarar variables de cada tipo de memoria. Cada declaración asigna así mismo un ámbito y un plazo de vida. El ámbito identifica el rango de hilos que pueden acceder a una variable: un sólo hilo individual, los hilos de un bloque o bien todos los hilos de la malla. Si el alcance es de hilo, se creará una copia particular para cada hilo. Cada hilo accederá a la copia que se le ha asignado.

Declaración de la Variable	**Memoria**	**Ámbito**	**Duración**
Variables Automáticas que no son arrays	Registros	Hilo	Kernel
Variables automáticas de tipo array	Global	Hilo	Kernel
__device__ __shared__ int SharedVar;	Compartida	Bloque	Kernel
__device__ int GlobalVar;	Global	Malla	Aplicación
__device__ __constant__ int ConstantVar;	Constante	Malla	Aplicación

Tabla 1: Características de las variables según su definición.

La duración especifica el trozo de programa durante el cual la variable estará disponible. El plazo de vida de una variable puede caer dentro del plazo de invocación de un kernel o durante toda la aplicación. En el primer caso debe ser declarada dentro del código de la función kernel y sólo estará disponible durante su ejecución. Si el kernel es invocado varias veces, en cada llamada se generará esa variable y no mantendrá su valor entre las distintas llamadas. Por otro lado si su duración se extiende a toda la aplicación, debe ser declarada fuera del código de cualquier función. Así su contenido se mantendrá durante toda la aplicación, y estará disponible para todos los kernels.

Las variables automáticas, excepto los array, son escalares y son declarados en el kernel y en las funciones del dispositivo y son colocadas dentro de los registros. El ámbito de las variables escalares se limita a los hilos individuales. Cuando una función de kernel declara una variable automática, se genera una copia particular para cada hilo que ejecuta la función. Cuando un hilo concluye, todas sus variables automáticas dejan de existir. El acceso a dichas variables es muy rápido y paralelo, pero se debe tener cuidado en no superar el límite de capacidad del registro que depende de la implementación Hardware. Las variables "array" automáticas no se almacenan en registros sino en la memoria global, por ello los tiempos de acceso son largos y se pueden producir congestiones. Su ámbito es el mismo que para los escalares; es decir son exclusivos para cada hilo. Cuando termina el hilo también desaparecen éstas variables. Debido a su largo tiempo de acceso se deben evitar.

Si la declaración de una variable viene precedida por las palabras "__shared__'', entonces se está declarando como variable compartida y se le asignará memoria de ese tipo. Opcionalmente se puede añadir la palabra "__device__" antes de "__shared__" en la declaración, obteniéndose el mismo resultado.

Esa variable será accesible por un bloque de hilos. Todos los hilos de un mismo bloque ven el mismo valor. Una versión particular se genera y se hace accesible para cada bloque de hilos y se define dentro de una función kernel o de una función de dispositivo. Se conservará mientras dure la ejecución del kernel. Cuando el kernel concluye, el contenido de la memoria compartida desaparece. Este tipo de variables, permite la colaboración eficiente entre hilos de un mismo bloque. El acceso a la memoria compartida es muy rápido. Generalmente en programación CUDA se usa esta memoria para almacenar la parte de memoria Global que se usa con frecuencia en la fase de ejecución de un kernel.

Por eso se debe adaptar el algoritmo para crear etapas que usan masivamente pequeñas porciones de la memoria Global.

Si una variable viene precedida por la palabra "__constant__" entonces se declara como variable de tipo contante. Se declaran fuera del cuerpo de las funciones. Es accesible a todas las mallas. Todos los hilos de todas las mallas de un dispositivo acceden al mismo valor. Dura lo mismo que la aplicación. Generalmente son usados para variables que proporcionan valores de entrada a las funciones kernel. Las variables constantes son alojadas en la memoria Global, pero son almacenadas en la caché para aumentar la eficiencia de acceso.

Una variable precedida por la palabra "__device__" es una variable global. El acceso a este tipo de variables es lento. Sin embargo son visibles para todos los hilos de todos los kernels. Su contenido persiste durante toda la aplicación. Se pueden usar para que hilos de distintos bloques colaboren entre sí. Sin embargo se debe tener presente que no se pueden sincronizar hilos de distintos bloques. Generalmente este tipo de variables son usados para transmitir información de un kernel a otro.

Se debe observar que hay una limitación en cuanto al uso de punteros para variables CUDA. Los punteros sólo pueden usarse para variables alojadas en la memoria Global.

Una estrategia para reducir el acceso a la memoria Global consiste en dividir los datos en grupos llamados "tejas" (Tiles) tal que cada teja se aloja en la memoria compartida. Naturalmente se debe cumplir que la ejecución del kernel sobre cada teja (grupo de datos) es independiente del resto para garantizar el paralelismo. Por otro lado el tamaño de estas tejas no debe exceder el tamaño de la memoria compartida.

b. LA MEMORIA COMO ELEMENTO LIMITADOR DE LA EFICIENCIA.

Tanto los registros, como la memoria compartida y la Constante son de acceso rápido y permiten reducir los accesos a la memoria Global, sin embargo su tamaño constituye una limitación importante. Cada dispositivo dispone de una determinada cantidad de memoria, que limita el número de hilos que pueden ser alojados en los Multiprocesadors o SM (Streaming Multiprocessors).

En general cuanto más espacio de memoria necesita un hilo, menos hilos pueden ser alojados en un SM, y de ahí menos hilos pueden ser alojados en el procesador en su conjunto.

Lo mismo sucede con la memoria Compartida que limita el número de bloques que pueden ser alojados por un SM.

2. MODELO DE HILOS EN CUDA

La granularidad fina en el uso de los hilos para datos en paralelo es esencial para la eficiencia en CUDA. Cuando se ejecuta un kernel se crean mallas de hilos que son las que ejecutan la función kernel. De hecho la función kernel especifica las partes que serán ejecutadas por cada hilo individual.

a. ORGANIZACIÓN DE LOS HILOS

Los hilos se organizan en dos niveles usando coordenadas para su indexación que se llaman "blockId" y "threadId". Estos valores son definidos por el sistema y accesibles a las funciones kernel.

Todos los hilos de un mismo bloque tienen el mismo valor de "blockId". El rango de "blockId" viene definido por la dimensión de la malla.

La organización de una malla se determina durante el lanzamiento del kernel. El primer parámetro especifica las dimensiones de una malla en función de los bloques que contiene. El segundo parámetro especifica las dimensiones de un bloque en función de los hilos que lo componen.

Por ejemplo:

```
dim3 dimBlock(4, 2, 2);

dim3 dimGrid(2, 2, 1);

KernelFunction<<<dimGrid, dimBlock>>>(...);
```

Este código define una malla bidimensional de 2x2 bloques y bloques tridimensionales de 4x2x2 hilos.

Las dos primeras líneas inicializan las dimensiones y la tercera es la que arranca el kernel propiamente dicho.

Lo esencial en cuanto a "BlockId" y "ThreadId" es indexar los hilos de manera unívoca y permitir así manejarlos de manera individualizada, concretamente, asignándoles los grupos de datos sobre los cuales operarán.

b. ESCALABILIDAD

La capacidad de ejecutar la misma implementación a distintas velocidades es lo que se denomina Escalabilidad Transparente, lo que reduce la carga para los programadores y mejora la usabilidad de las aplicaciones.

CUDA admite escalabilidad ya que permite a los hilos pertenecientes a un mismo bloque coordinar sus actividades usando una función de barrera de sincronización "syncthreads()". Cuando un kernel realiza una llamada a "syncthreads()", todos los hilos de un mismo bloque son retenidos en el punto de llamada hasta que cada uno alcance ese punto. Esto garantiza que todos los hilos de un bloque completen una etapa de ejecución del kernel antes de pasar a la siguiente etapa.

La capacidad de sincronización impone limitaciones a los hilos de un mismo bloque. Los hilos se deberían ejecutar en tiempos parecidos para evitar tiempos de espera largos. Esto se logra asignando recursos de ejecución a los hilos de un mismo bloque tomados como unidad. Todos los hilos son asignados al mismo recurso.

Se debe tener presente que al no existir sincronización entre hilos de distintos bloques, tampoco se imponen limitaciones en la ejecución del programa por diferentes bloques. Ningún bloque está obligado a esperar a otro para llevar a cabo sus tareas. La flexibilidad así definida permite implementaciones escalables.

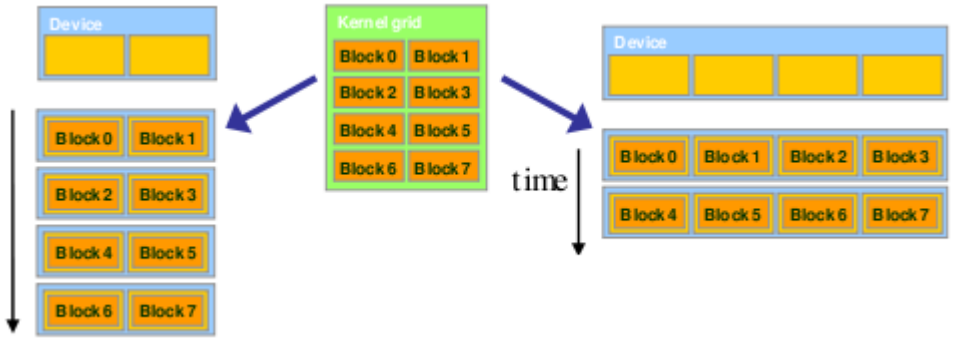

Fig. 3: Ejemplo de escalabilidad en la ejecución del mismo kernel en dos sistemas distintos.

En una implementación con pocos recursos, como es el caso del ejemplo de la izquierda de la figura 3, se puede elegir ejecutar pocos bloques a la vez. En cambio en situaciones más exigentes se puede recurrir a la implementación de la derecha de la misma figura que nos permite ejecutar varios bloques simultáneamente.

La capacidad de ejecutar una aplicación en un amplio rango de velocidades permite producir distintas implementaciones alternativas dependiendo del coste, energía, y necesidades de eficiencia concretos.

c. ASIGNACIÓN DE HILOS

En cuanto se arranca un kernel, el sistema CUDA genera la malla de hilos correspondiente. Estos hilos son asignados a recursos de ejecución en grupos de bloques. En la serie GeForce 8, los recursos de ejecución se organizan en "Streaming Multiprocessors". Por ejemplo la GeForce 8800GTX implementa hasta 16 "Streaming Multiprocessors", dos de los cuales se muestran en la figura 4. Se pueden asignar hasta 8 bloques para cada SM en este diseño, siempre y cuando haya suficientes recursos (por ejemplo, memoria para almacenar los registros) para las necesidades de cada bloque. En caso de que esto no se cumpla, CUDA automáticamente reduce el número de bloques asignados a cada SM. Con 16 SM, en el modelo GeForce 8800 GTX podemos tener hasta un máximo de 128 bloques asignados simultáneamente a los SMs. Sin embargo, debemos tener presente que muchas mallas contienen más de 128 bloques.

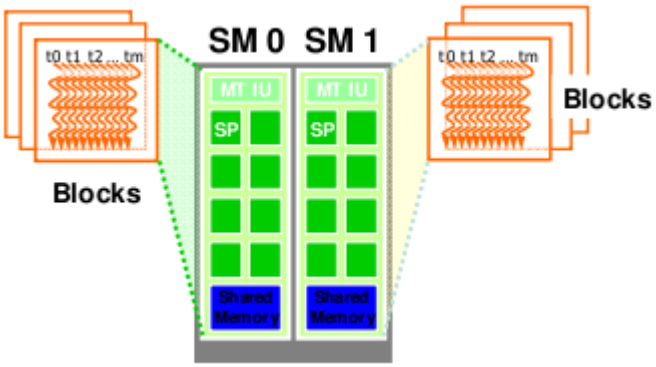

Fig. 4: Esquema de Multiprocesador ("Streaming Multiprocessor") y asignación de tareas.

Una limitación de los SM es el número total de hilos que pueden monitorizar simultáneamente. En el modelo GeForce 8800GTX se pueden asignar hasta un máximo de 768 hilos para cada SM, mientras que la GeForce 260GTX tiene un máximo de 1024 hilos.

Esto se traduciría en 3 bloques de 256 hilos, o 6 bloques de 128 hilos, por ejemplo. Sin embargo el modelo con 12 bloques de 64 hilos no es posible ya que cada SM sólo puede alojar a 8 bloques. Por tanto se pueden tener hasta 12,288 hilos dispuestos para ser ejecutados.

d. PROGRAMACIÓN DE HILOS

La programación de hilos es un concepto que depende estrechamente de cada implementación y por ello se analiza dependiendo de cada Hardware en particular.

En el caso de la GeForce 8800GTX, cuando un bloque es asignado a un "Streaming Multiprocessor", se divide en unidades de 32 hilos que forman "warps". El tamaño de cada "warp" se ha mantenido hasta el momento en las implementaciones.

De hecho los "warps" no forman parte de la definición del lenguaje CUDA. Si embargo su conocimiento permite optimizar el rendimiento. Ya que los "warps" son las unidades de programación dentro de los SMs. La figura 5 muestra la división de bloques en "warps" para el caso particular de la GeForce 8800GTX. Cada "warp" está formado por 32 hilos consecutivos. En este ejemplo hay 3 "warps" en cada uno de los 3 bloques. Los 3 bloques han sido asignados al mismo SM.

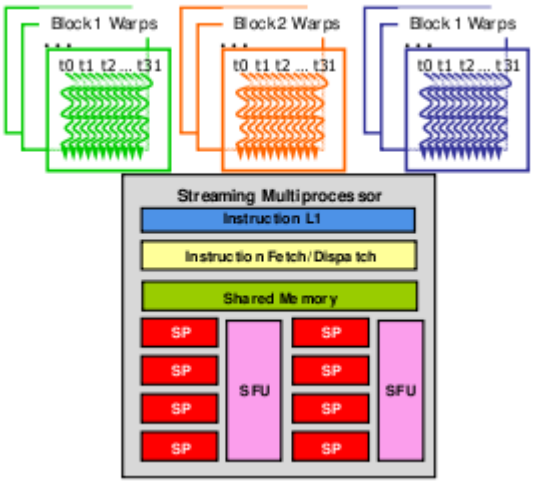

Fig. 5: Esquema de los bloques constitutivos del Multiprocesador y asignación de Warps de distintos Bloques.

Ya que, como máximo puede haber 768 hilos en cada SM entonces puede haber un máximo de 768/32 = 24 "warps". Para el caso de la 260 GTX tendríamos hasta 32 Warps.

En un determinado momento sólo un "warp" es ejecutado en cada SM. Esto es debido a que el diseño hace que de esta manera los procesadores ejecutan de manera eficiente operaciones lentas como accesos a la memoria Global. Si una instrucción que se está ejecutando en un "warp" necesita esperar por ejemplo un resultado, el "warp" se coloca en modo espera y otro "warp" es activado.

3. MODELO DE PROGRAMACIÓN CUDA

El modelo de programación consiste en un Host tradicional, es decir, una CPU y uno o más dispositivos dedicados al procesamiento paralelo con muchas unidades de ejecución aritmética, en nuestro caso una GPU.

Actualmente se pueden localizar e identificar secciones de código con un número elevado de datos en paralelo. Estas secciones pueden ser perfectamente ejecutadas de una manera independiente y simultánea. Los dispositivos CUDA pueden acelerar este tipo de programas ejecutando las secciones de código en paralelo.

a. PROGRAMACIÓN EN PARALELO

El término "datos en paralelo" se refiere a que en un programa se pueden realizar operaciones aritméticas sobre la estructura de datos de manera segura, simultánea e independiente.

Un ejemplo sencillo es el producto de 2 matrices, donde la multiplicación de filas por columnas, elemento a elemento y su suma nos da el valor del elemento de la matriz resultado. Es obvio que cada elemento de la matriz resultado puede obtenerse independientemente de los demás. De ahí la programación en paralelo.

b. ESTRUCTURA DE UN PROGRAMA CUDA

Un programa CUDA consiste en una o varias fases que son ejecutadas en el host (CPU) o en un dispositivo como la GPU. Las etapas que tienen pocos paralelismo de datos o ninguno son implementadas en código host (Programación Secuencial). Aquellas que presentan mucho paralelismo se implementan en código dispositivo. El código del programa es único y comprende ambas secciones. EL Compilador C de NVIDIA (nvcc) separa ambos tipos. El código host es C convencional, es compilado con el compilador normal del host y es ejecutado como cualquier programa convencional. En cambio el código de dispositivo está escrito en C convencional con extensiones que permiten etiquetar las funciones de datos paralelos. Estos códigos son llamados kernels, tienen una estructura de datos asociada particular y además son compilados con "nvcc" y ejecutados en el dispositivo GPU.

Los kernels generan un elevado número de hilos para aprovechar el paralelismo de datos. Además en CUDA los hilos son más ligeros que en una CPU, por eso es correcto asumir que los hilos CUDA tardan poco tiempo (Unos cuantos ciclos de reloj) en ser generados y programados debido a la eficiencia del diseño del hardware. En cambio los hilos de la CPU tardan miles de ciclos de reloj.

```
Código secuencial CPU
        Kernel paralelo GPU
                kernelA<<<nBlk, nTid>>>(args);
Código secuencial CPU
        Kernel paralelo GPU
                KernelA<<<nBlk, nTid>>>(args);
```

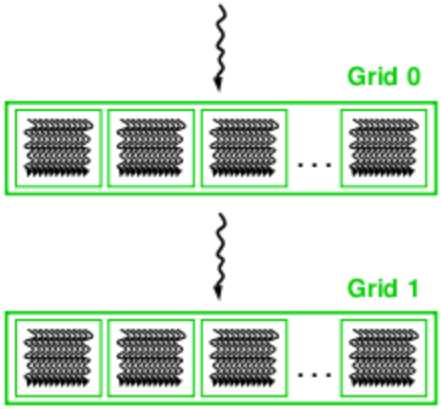

Fig. 6: Ejecución de un programa CUDA.

La ejecución de un programa CUDA se muestra en la figura 6. Empieza ejecutando el código en el host (CPU). Cuando el kernel es invocado, se traslada la ejecución al dispositivo (GPU), donde un elevado número de hilos son creados para aprovechar el paralelismo de datos. Los hilos creados por un kernel durante una ejecución son llamados malla (Grid). La misma figura representa la ejecución de dos mallas de hilos. Cuando los hilos de kernel terminan, las mallas respectivas también concluyen, y la ejecución vuelve al host hasta que un nuevo kernel es invocado.

c. FUNCIONES DE KERNEL E HILOS.

En CUDA las funciones de kernel especifican la parte de código que será ejecutado en la GPU por todos los hilos en una etapa en paralelo. Debido a que los hilos de una etapa paralela ejecutan el mismo código, CUDA es un ejemplo del modelo "Programa Único-Datos Múltiples" (SPMD).

La sintaxis de las funciones es similar a la de ANSI C con unas extensiones concretas.

La palabra "__global__"antes de la declaración de una función indica que esa función es una función kernel que puede ser invocada por el host para crear hilos.

Las palabras "threadIdx.x" and "threadIdx.y" se refieren a los índices de un hilo. Permiten indexar los hilos y por tanto identificarlos de manera unívoca. Debido a que todos los hilos ejecutan el mismo kernel, esto permite distinguirlos y asignarlos de manera unívoca para tratar la parte asignada de datos que les corresponda.

Estas palabras permiten al hilo acceder a las memorias de registro respectivas. Los índices reflejan la disposición multidimensional de los hilos.

Cuando se invoca un kernel, éste se ejecuta como malla de hilos paralelos. En CUDA cada malla comprende generalmente desde miles a millones de hilos. Generar suficientes hilos para usar de manera eficiente el Hardware requiere una elevada cantidad de paralelismo.

Un bloque es un lote de hilos que pueden colaborar entre sí mediante:

- Sincronización de la ejecución.
- Accesos seguros a memoria compartida de baja latencia.

Dos hilos de Bloques diferentes no pueden cooperar entre sí.

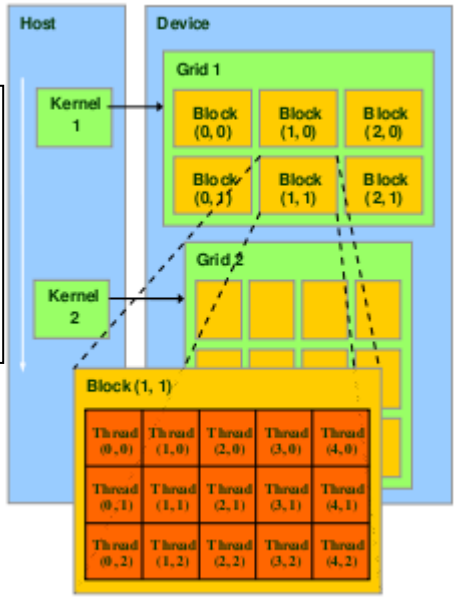

Fig. 7: Representación de la disposición dimensional de los hilos en bloques y de los bloques en Mallas.

Los hilos en una malla se organizan, como se muestra en la figura 7, en dos niveles. Por simplicidad se ha representado un número muy reducido de hilos. En el nivel superior los hilos se organizan en bloques. Cada malla está formada por uno o varios bloques de hilos. Todos los bloques de una malla tienen el mismo tamaño. En esta figura, la malla 1 (Grid 1) está formada por 6 bloques de hilos formando una matriz de 2X3 bloques. En CUDA, cada bloque está unívocamente identificado mediante las palabras "blockIdx.x" y "blockIdx.y". Por otro lado, todos los bloques deben contener el mismo número de hilos y además deben estar organizados de la misma forma.

Cada bloque se organiza en una matriz tridimensional con un total máximo de 512 hilos por bloque. Las coordenadas de los hilos en un bloque están definidas mediante los índices "threadIdx.x", "threadIdx.y", y "threadIdx.z". Sin embargo si uno de los índices fuera constante e igual a uno daría una matriz bidimensional como la del ejemplo de la figura 7.

En el caso representado, cada bloque está dispuesto en forma de matriz bidimensional de 3x5 hilos. Por tanto la malla 1 contiene 90 hilos en total.

Cuando un código host invoca a un kernel, lo hace definiendo las dimensiones tanto de la malla como de los bloques. Estos valores los pasa como parámetros. Para ello se usan dos tipos de estructuras "dim3" una para la malla y otra para el bloque.

Ejemplo de un Kernel:

Un kernel en "C para CUDA", es una función que al ejecutarse lo hará en N distintos hilos en lugar de manera secuencial. Se define incluyendo __global__ en la declaración. Por ejemplo:

```
//Definición del kernel
__global__ void f(int a, int b, int c)
{
}
```

Si deseamos que la función f calcule la diferencia entre dos vectores A y B y almacene el resultado en un tercero C:

```
__global__ void f(int* A, int* B, int* C)
{
        int i = threadIdx.x;
        C[i] = A[i] - B[i];
}
```

Esta función se ejecutaría una vez en cada hilo, reduciendo el tiempo total de ejecución en gran medida, y dividiendo su complejidad, O(n), por una constante directamente relacionada con el número de procesadores disponibles.

El mismo ejemplo con matrices sería:

```
__global__ void f(int** A, int** B, int** C)
{
        int i = threadIdx.x; //Columna del bloque que
ocupa este determinado hilo
        int j= threadIdx.y; //Fila
        C[i][j] = A[i][j] - B[i][j];
}
```

Ejemplo de invocación de un kernel:

En una llamada a un kernel, se le ha de pasar el tamaño de grid y de bloque, por ejemplo, en el main o bien en la sección HOST del ejemplo anterior podríamos añadir:

```
dim3 bloque(N, N); //Definimos un bloque de hilos de NxN
dim3 grid(M, M) //Grid de tamaño MxM
f<<<grid, bloque>>>(A, B, C);
```

En el momento que se invoque esta función, los bloques de una malla se indexarán y distribuirán por los distintos multiprocesadores disponibles.