

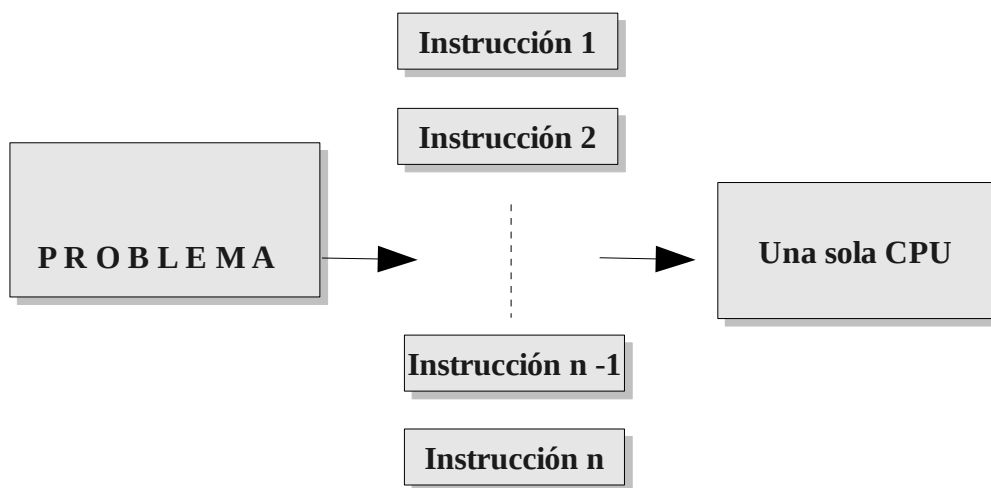
# 1

## Introducción a los Computadores Paralelos

### 1. ¿Qué es la computación paralela?

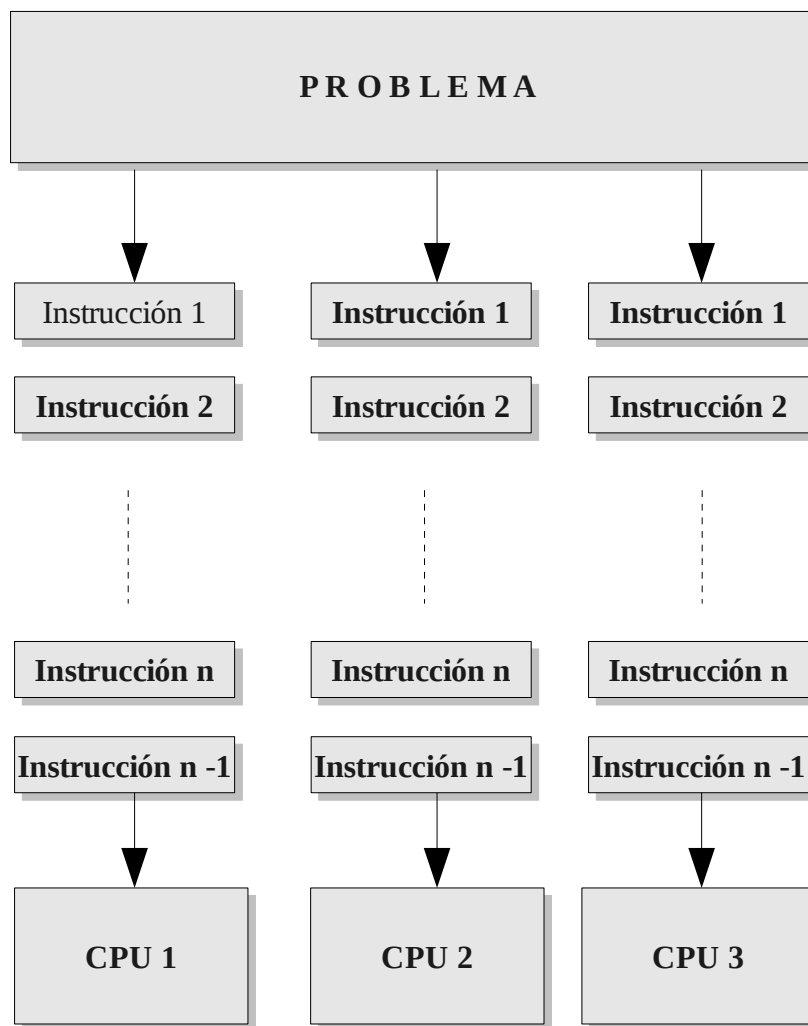
Desde hace varias décadas, el software siempre había sido escrito para el procesamiento en serie, siguiendo los paradigmas de la programación estructurada.

El objetivo del código era ser ejecutado en una computadora con una sola CPU, donde el problema era descompuesto en series discretas de instrucciones que se ejecutaban una detrás de otra, donde sólo una instrucción se podía ejecutar a la vez, como se ilustra en la siguiente figura:



**Figura 1.1** Descomposición de un problema en instrucciones serie

A diferencia de la anterior, en la programación paralela el problema es dividido en partes computacionales con el objetivo de ser ejecutadas en varias CPU. Estas partes discretas en el que el problema es descompuesto se van a resolver concurrentemente y cada parte será dividida en una serie de instrucciones que se ejecutarán simultáneamente en diferentes unidades de procesamiento. De esta forma el tiempo de cómputo es mucho menor al separar la complejidad de un problema en partes individuales que serán ejecutadas por múltiples CPU al mismo tiempo. Estas partes serán programas cuyas instrucciones se ejecutarán en un cauce paralelo diferente, como podemos apreciar en la siguiente figura:



**Figura 1.2** Ejecución paralela

## 2. Aplicaciones de la computación paralela

La computación paralela es una evolución de la computación serie que intenta emular los aspectos del mundo real y natural: eventos complejos e interrelacionados ocurriendo al mismo tiempo dentro de una secuencia. Así, tenemos la analogía de la formación de galaxias, movimiento de planetas, patrones de eventos atmosféricos y de océanos, movimiento de las placas tectónicas, control de tráfico, etc.

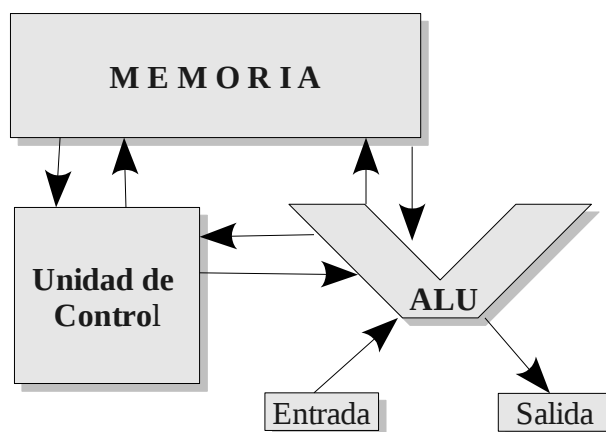
Históricamente la computación paralela se ha considerado el punto final de la más alta computación y ha sido usada para resolver problemas complejos de ingeniería y ciencia que acaecen en el mundo real.

De esta forma encontramos el uso del paralelismo en computación en los siguientes entornos:

- **Cambio climático:** Se están utilizando los patrones de cambios atmosféricos para predecir con exactitud los fenómenos alterados del tiempo.
- **Genoma:** En el secuenciamiento de la estructura del ADN del código genético.
- **Nanotecnología y química:** Para crear nuevos compuestos moleculares con propiedades microscópicas y materiales más resistentes a la fuerza y al calor.
- **Geología y sismología:** Utilizado en el estudio las propiedades geotécnicas y la predicción de movimientos sísmicos.
- **Ingeniería electrónica:** Diseño de circuitos con más nivel de integración, más rápidos y con menos calentamiento del chip.
- **Prospección petrolífera:** Debido al agotamiento de los combustibles fósiles se ha hecho patente la necesidad de búsqueda de nuevos yacimientos para la obtención de petróleo y gas.
- **Buscadores Web:** Con la proliferación incesante de Internet en la última década se necesita una gran cantidad de recursos de cómputo que para rastrear e indexar todos los contenidos publicados en la red.
- **Diagnóstico por imagen:** El avance de la medicina requiere de sistemas de imagen capaces de diagnosticar con precisión y sin errores la existencia de enfermedades.
- **Industria farmacéutica:** En el diseño de nuevos medicamentos eficaces con los mínimos efectos secundarios.
- **Modelado económico y financiero:** Las transacciones financieras realizadas en un mundo globalizado pueden poner en riesgo la economía mundial. El análisis complejo de estas interacciones de agentes puede prevenir una posible crisis económica.
- **Gráficos y realidad virtual:** Con el fin de conseguir imágenes realistas en el mundo de la infografía y los videojuegos.

### 3. Arquitecturas Von Neumann y secuenciales

El matemático húngaro John Von Neumann sentó las bases de la computación moderna a la que pertenecen los ordenadores secuenciales, también denominados computadores serie. Este modelo se basa en la Unidad Central de Procesamiento o CPU (acrónimo anglosajón de *Central Processing Unit*), en la memoria donde se almacenan datos e instrucciones. En esta arquitectura las instrucciones se ejecutan en serie que tratan con un única secuencia de datos. En la sección 1.4 veremos más en profundidad los fundamentos de este modelo.



**Figura 1.3** Arquitectura Von Neumann (CPU y memoria)

Esta arquitectura presenta dos factores limitadores a tener en cuenta: la velocidad a la que se ejecutan las instrucciones y la velocidad de transferencia de datos e instrucciones entre la memoria y la CPU. En el primer factor están implicados los componentes y la tecnología electrónica (frecuencia de reloj, etc.). No obstante, desde los años noventa se han desarrollado nuevas estrategias para aumentar el rendimiento. La más conocida es la **segmentación** que consiste en ejecutar varias partes de un número determinado de instrucciones al mismo tiempo en varias unidades funcionales de cálculo (multiplicadores, sumadores, unidades de carga y almacenamiento,...). Esto se consigue dividiendo las partes de ejecución de una instrucción en etapas, alcanzando un rendimiento bastante mejorado con respecto a una máquina no segmentada o secuencial. En la figura 1.4 podemos ver el funcionamiento de una CPU segmentada.

Instrucción	Ciclo de reloj								
	1	2	3	4	5	6	7	8	9
<i>i</i>	IF	ID	EX	MEM	WB				
<i>i+1</i>		IF	ID	EX	MEM	WB			
<i>i+2</i>			IF	ID	EX	MEM	WB		
<i>i+3</i>				IF	ID	EX	MEM	WB	
<i>i+4</i>					IF	ID	EX	MEM	WB

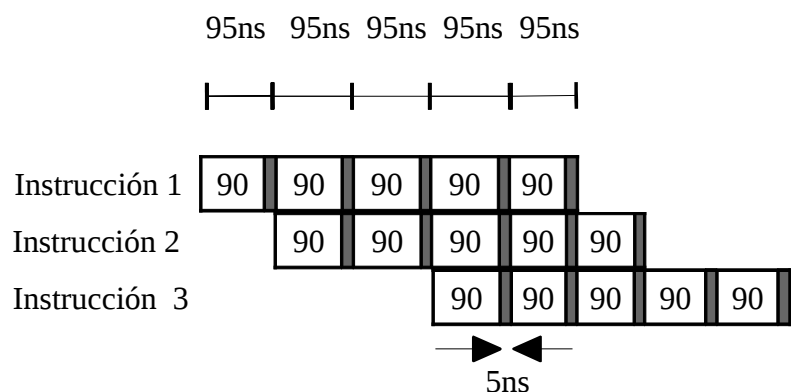
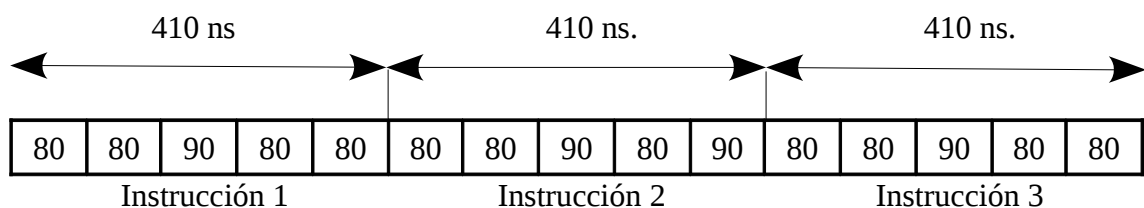
**Figura 1.4** Segmentación RISC

Donde la etapa de ejecución IF es la búsqueda de instrucción en la caché o en la memoria principal, ID es la decodificación de la instrucción y lectura de los registros, EX es la etapa ejecución aritmético lógica, MEM es la lectura y escritura en memoria, y finalmente WB escribe el resultado en los registros.

Como el lector puede apreciar, la ventaja de esta estrategia radica en que es posible adelantar la ejecución de una instrucción en cada ciclo de reloj. En consecuencia en cada ciclo de reloj se finaliza una instrucción y el incremento del rendimiento, sin tener en cuenta los riesgos de la segmentación, es de un factor de cinco con respecto a una máquina que ejecuta las instrucciones en serie.

En contraposición, aunque el incremento de la productividad es muy alto con segmentación, el tiempo de CPU que se le dedica a una instrucción individual es superior al de su homólogo secuencial. Esto es debido a:

- Los cerrojos que hay que colocar en la circuitería para separar la información entre etapas.
- La duración de las etapas es similar y viene determinada por la etapa más larga.
- Los riesgos estructurales, de datos y de control que introducen detenciones en el cauce de ejecución.



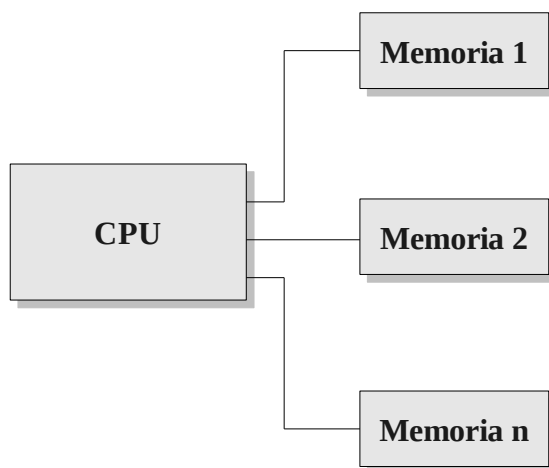
**Figura 1.5** Versión no segmentada (arriba) y segmentada de tres instrucciones (abajo)

Como puede observarse en la figura 1.5, la instrucción no segmentada consume 410 ns. , mientras que su equivalente segmentada consume 475 ns. , a causa de la etapa más lenta que es 90 ns, en este caso suponiendo una segmentación ideal y un retardo de 5 ns. Por los cerrojos de cada etapa segmentada. Esto implica un aumento de productividad que no siempre está asociado a un incremento de velocidad de ejecución.

Con respecto a la segundo problema de la segmentación, éste admitiría dos soluciones.

- **Utilizar memoria entrelazada o intercalada (memory interleaving):**

Esta estrategia arquitectónica permite dividir la memoria en bloques, cada uno con acceso independiente, tal y como puede verse en la siguiente figura:

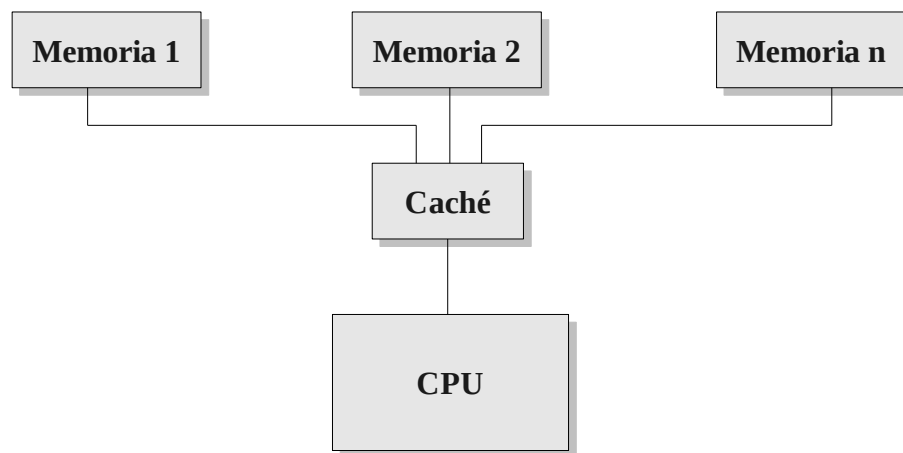


**Figura 1.6** Memoria entrelazada

- **Utilizar una memoria caché:**

Otra estrategia más conocida para aumentar la velocidad de intercambio de información es la memoria caché. Con esta memoria se consigue mayor velocidad de intercambio en un espacio de direcciones más pequeño, en detrimento del coste de la tecnología. Con este técnica se consigue el objetivo del principio de localidad, en la caben dos posibles casos: La *localidad espacial* que consiste en el hecho de que la instrucción siguiente a la que se está ejecutando está en la cercanía del bloque en uso, La *localidad temporal* que consiste en que los datos o instrucciones que serán accedidos a continuación serán los que han sido accedidos recientemente.

En la siguiente figura 1.7 de la siguiente página se muestra una arquitectura que utiliza memoria caché:



**Figura 1.7** Arquitectura con memoria entrelazada y memoria caché

Los procesadores segmentados son útiles para aplicaciones científicas y de ingeniería. Un procesador segmentado, en términos generales, siempre va a utilizar una memoria caché para evitar que las instrucciones de acceso a memoria sean muy lentas.

No obstante, en las aplicaciones científicas, los datos de grandes dimensiones son accedidos con baja localidad, y de esta manera el rendimiento que se consigue en la jerarquía de memoria es bastante pobre y con un efecto resultante de mal rendimiento en el uso de la caché. Una posible solución alternativa sería prescindir de las cachés, siempre y cuando fuera posible determinar qué patrones de acceso son los que definen el acceso a memoria y realizar la segmentación en consecuencia.

## 4. Taxonomía de Flynn

Para clasificar las arquitecturas paralelas existen diferentes clasificaciones; aunque una de las más utilizadas es la que definió el ingeniero Michael J. Flynn en 1972 y es conocida como la taxonomía de Flynn.

Esta clasificación distingue arquitecturas multiprocesadores de acuerdo a dos dimensiones independientes: **datos** e **instrucciones**, donde cada una de estas dimensiones puede tener sólo dos posibles estados: **simple** (single) y **múltiple** (multiple).

A continuación examinaremos cada una de estas clasificaciones.

### 4.1 Single Instruction, Simple Data (SISD)

Dentro de esta categoría se encuadran la mayoría de los computadores secuenciales disponibles actualmente: PC, Estaciones de Trabajo y antiguos Mainframe. Las instrucciones se ejecutan secuencialmente, pero pueden utilizar segmentación y más de una unidad de cálculo. Siempre harán uso de una única unidad de control que gobierna todas las instrucciones. Es traducido como instrucción única, datos únicos.

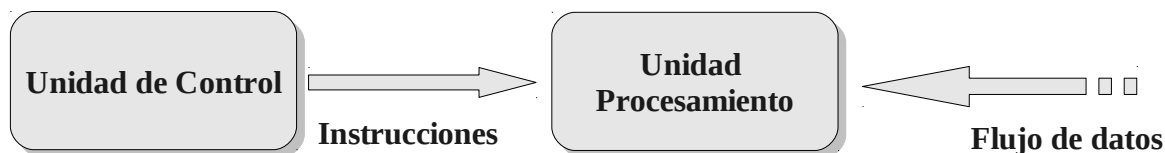
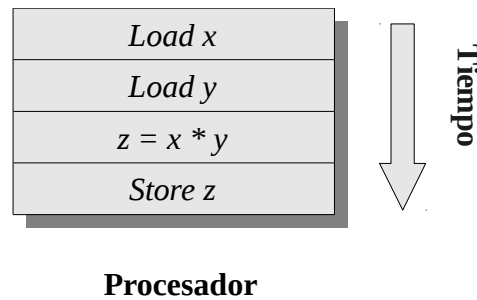


Figura 1.8 Arquitectura SISD

La secuenciación de las instrucciones es producida por cada pulso de reloj, haciendo que la CPU obtenga un dato o una instrucción de memoria para su ejecución secuencial.



Así, de esta manera si las instrucciones se ejecutarían en el orden de la figura 1.9

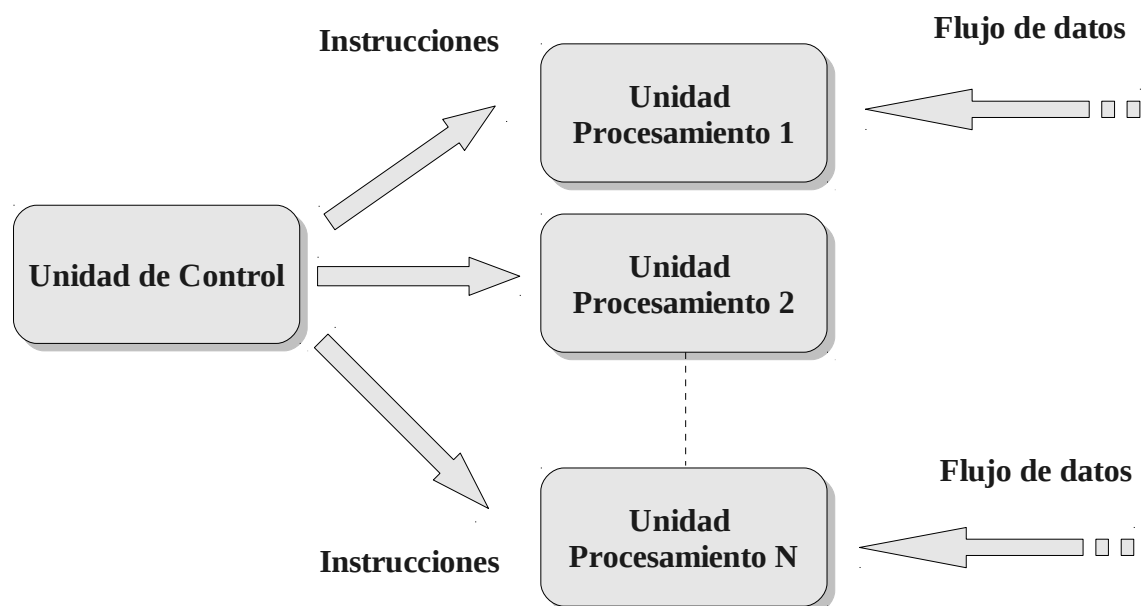


**Figura 1.9** Ejemplo ejecución de instrucciones en SISD

## 4.2 Single Instruction, Multiple Data (SIMD)

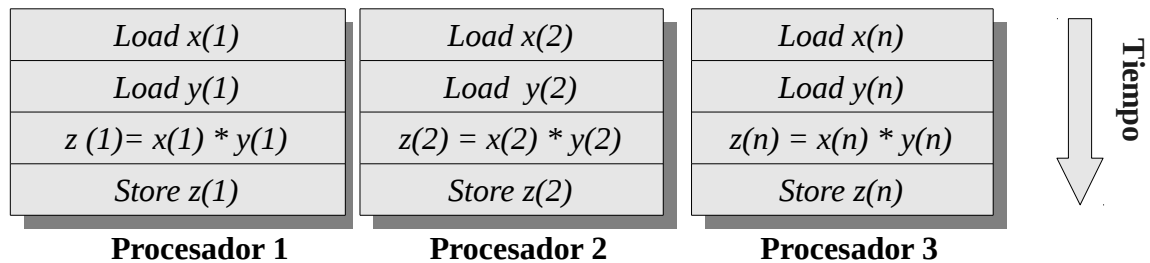
Este modelo se ilustra en la figura 1.10. Es un tipo de arquitectura paralela donde todas las unidades ejecutan la misma instrucción en un ciclo de reloj y cada unidad opera sobre una parte diferente de los datos. Esta arquitectura es especialmente apropiada para problemas de procesamiento de imágenes y gráficos, como es el caso de CUDA, que veremos en el capítulo siguiente.

Se traduce como instrucción única, datos múltiples. En esta clasificación se encuadran los procesadores matriciales en los que existen más de una unidad de procesamiento trabajando sobre flujos de datos distintos, pero ejecutando la misma instrucción proporcionada por una única unidad de control como vemos en la figura 1.10:



**Figura 1.10** Arquitectura SIMD

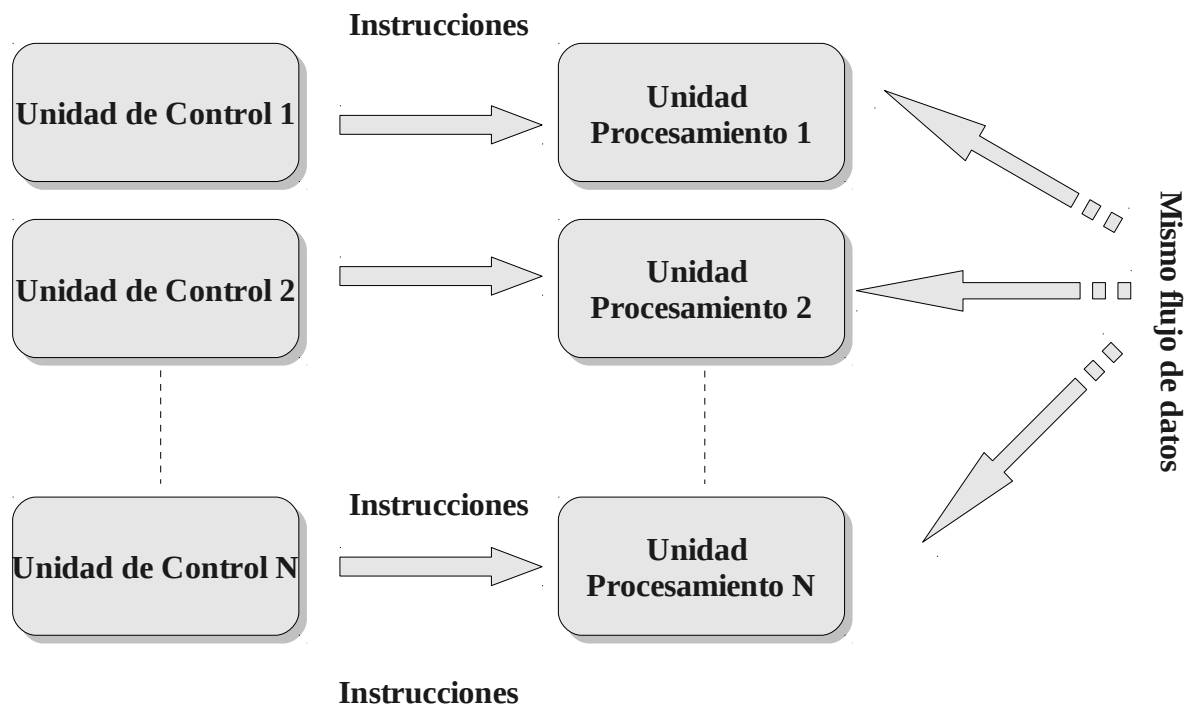
Por lo tanto, un ejemplo de orden de ejecución de las instrucciones queda tal como se muestra en la siguiente figura:



**Figura 1.10** Ejemplo de ejecución en arquitectura SIMD

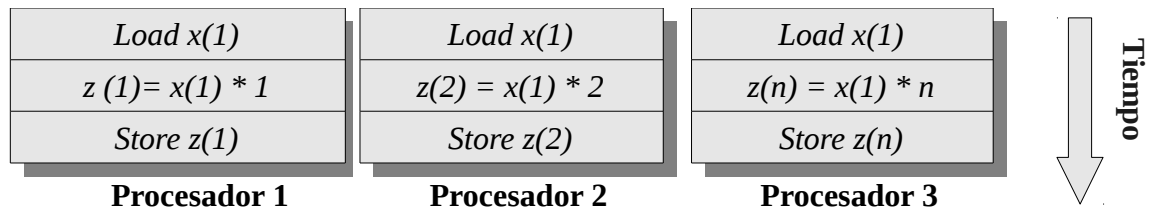
### 4.3 Multiple Instruction, Simple Data (MISD)

Es la traducción de instrucciones múltiples, datos únicos. Esta organización se caracteriza por el hecho de la existencia de varias unidades de procesamiento cada una ejecutando una instrucción diferente pero sobre el mismo dato. Es una arquitectura teórica y no se conoce ninguna materialización de esta categoría.



**Figura 1.11** Arquitectura MISD

De esta forma, un ejemplo que clarifica el flujo de instrucciones sería el siguiente:



**Figura 1.10** Ejemplo de ejecución en arquitectura MISD

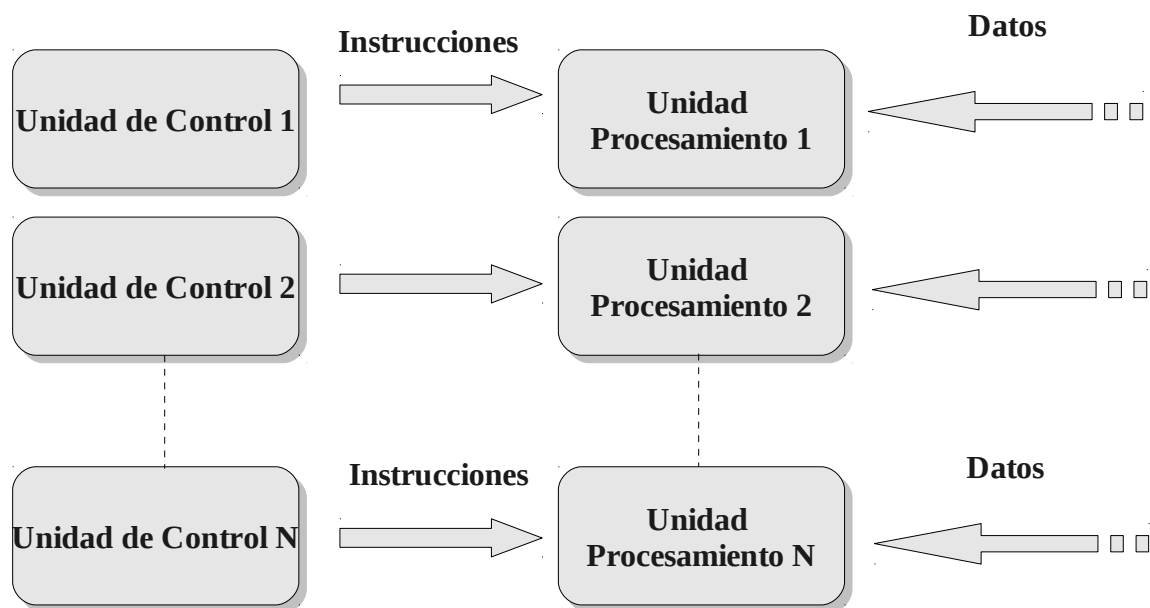
#### 4.4 Multiple Instruction, Multiple Data (MIMD)

Instrucciones múltiples, datos múltiples. Dentro de esta categoría se encuadran la mayoría de sistemas multiprocesadores y multicomputadores, donde cada programa se ejecuta en cada unidad de procesamiento. Esta arquitectura implica una coordinación entre procesadores, puesto que todos los flujos de memoria se obtienen de un espacio compartido para todos ellos. Los procesadores en esta arquitectura trabajan de manera independiente.

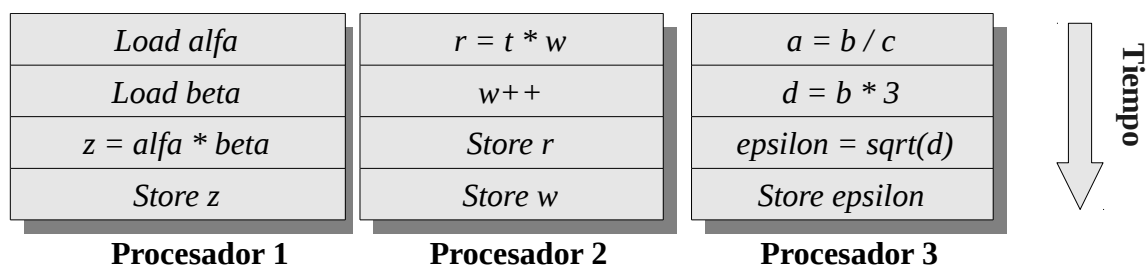
Por ejemplo, dos procesadores podrían estar realizando la Transformada Rápida de Fourier, mientras otro realiza la salida gráfica. Esto es lo que habitualmente se denomina *descomposición de control*. Aunque no debemos pasar por alto la *descomposición de datos*, en la que los datos del problema se reparten entre los distintos procesadores y todos ejecutan el mismo programa sobre la zona de datos que se le ha asociado. Éste modelo es mucho más potente que las arquitecturas SIMD.

La gran ventaja de los sistemas SIMD con respecto a los MIMD es que necesitan menos hardware, ya que sólo requieren una unidad de control. Los sistemas MIMD son más independientes y ejecutan cada uno de ellos una copia del programa y del Sistema Operativo. Además de esto, los sistemas SIMD necesitan menos tiempo de inicio para las comunicaciones entre procesadores, debido a que son transferencias entre registros de los procesadores.

Como conclusión, aunque los sistemas SIMD aparentemente sean más baratos que los MIMD, esto no es cierto, pues los sistemas MIMD pueden construirse con ordenadores convencionales de baja potencia.



**Figura 1.11** Arquitectura MIMD



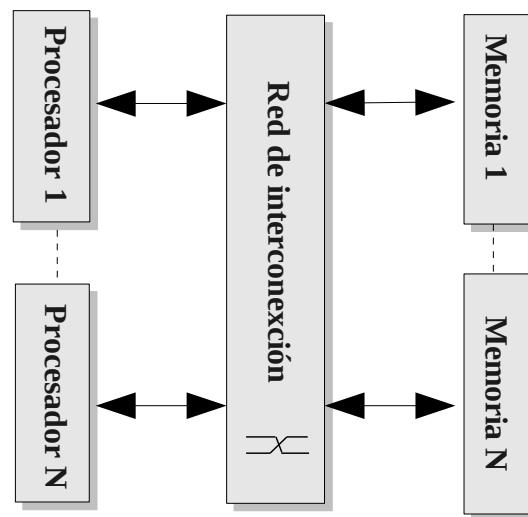
**Figura 1.12** Ejemplo de ejecución en arquitectura MIMD

## 4. Organización del espacio de direcciones de memoria

Dependiendo de cómo se organice el sistema de almacenamiento primario, los procesadores se comunicarán de diferentes formas. Atendiendo a esta organización, se podrán encontrar las siguientes arquitecturas:

### 4.1 Sistemas de memoria compartida o multiprocesadores

Estos sistemas se caracterizan porque los procesadores comparten físicamente la memoria y acceden al mismo espacio de direcciones de memoria. Por ejemplo, si un procesador escribe en una de las direcciones de memoria, otro procesador del mismo sistema puede acceder a esa misma posición directamente. Consecuentemente, los cambios realizados por un procesador en la memoria común son visibles por los otros procesadores del sistema a través de una *red de interconexión*, como puede verse en la siguiente figura:



**Figura 1.13** Esquema básico de los sistemas con memoria compartida

Ahora bien, dentro de estos sistemas de memoria compartida podemos encontrar otra clasificación dentro de ella. Atendiendo a la duración del tiempo de acceso a la memoria. Así nos encontramos con una nueva clasificación:

- Sistemas de acceso uniforme a memoria (UMA)
- Sistemas de acceso no uniforme a memoria (NUMA)

Estos sistemas consiguen una mejora del rendimiento añadiendo a cada procesador una memoria local, donde se almacena el código que ejecuta cada uno, así como los datos que no son compartidos entre procesadores del sistema y por tanto son locales a ese procesador. De esta forma se consigue evitar un exceso de intercambio de información en la red de interconexión para buscar código y datos locales, y consecuentemente una mejora de la eficacia.

### 4.1.1 Sistemas de acceso uniforme a memoria (UMA)

Son las siglas de Uniform Memory Access. Estos sistemas se caracterizan por tener igual tiempo de acceso a memoria, además los procesadores se caracterizan por ser idénticos.

La mayoría de los sistemas de memoria compartida se distinguen por incorporar una memoria caché local en cada procesador, consiguiendo por tanto un incremento en el ancho de banda entre el procesador y la memoria local. Los así llamados CC-UMA (Cache Coherent UMA) son sistemas UMA donde la coherencia de caché <sup>1</sup>debe intentar que si un procesador modifica una dirección de memoria, todos los procesadores del sistema deben estar informados al respecto. La figura 1.13 es un ejemplo de arquitectura UMA.

Una de las limitaciones de la arquitectura UMA es la escalabilidad y normalmente encontramos estos sistemas con un número de procesadores comprendido entre 16 y 32; aunque algunos sistemas han logrado alcanzar hasta los 64 procesadores como el Sun E10000. Se ha estudiado que si saturamos al sistema de procesadores, no se conseguirá ninguna mejora. Actualmente, los sistemas UMA se utilizan para la construcción de arquitecturas multiprocesador de bus compartido con un número reducido de procesadores.

Estos sistemas son comúnmente representados por los multiprocesadores simétricos.

### 4.1.2 Sistemas de acceso no-uniforme a memoria (NUMA)

Los sistemas de acceso no-uniforme a memoria se les denomina también como sistemas de memoria compartida distribuida, o sus siglas en inglés (*Distributed Shared Memory*). En estos sistemas se consigue una mejora del rendimiento eliminando la memoria global a la que acceden los procesadores, eso sí, siempre que el tiempo de acceso a la memoria local sea muy inferiores al tiempo de acceso por la red de interconexión. En estas arquitecturas los procesadores no tienen un tiempo igual de acceso a las memorias.

El sistema KSR-1 es el primero que incorporó coherencia por hardware. A este tipo de sistemas se le denominó ccNUMA, aunque también se les conoce como multiprocesadores simétricos escalables.

Actualmente la mayoría de los sistemas de arquitectura de memoria compartida son NUMA. En este caso no existe el problema de la escalabilidad, por lo que no es difícil encontrar sistemas con más de 512 procesadores interconectados, incluso hasta 2.048.

En general NUMA no necesita de mecanismos especiales que aseguren que los procesadores interconectados tengan un conocimiento coherente del espacio de direcciones global, siendo el programador el que las controle.

---

1 La coherencia es diseñada a nivel hardware

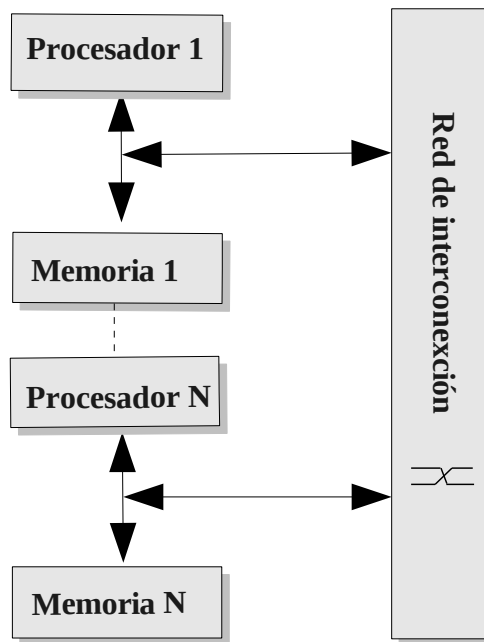


Figura 1.14 Arquitectura NUMA

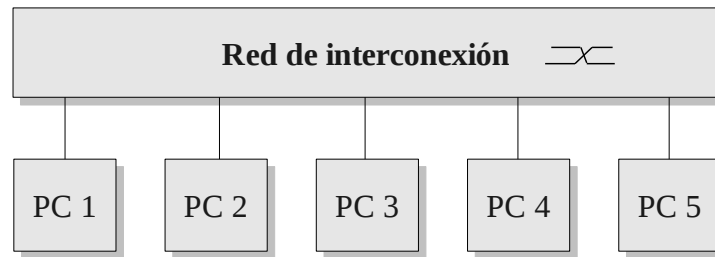
## 4.2 Sistemas de memoria distribuida o multicomputadores

Los sistemas de memoria distribuida se caracterizan porque cada sistema contiene su propio procesador y su propia memoria, comunicándose cada sistema entre sí por medio de *paso de mensajes* a través de una red de computadores. Aunque este apartado es extensible a una inmensa bibliografía, aquí vamos a explicarla de manera superficial, ya que no es materia relacionada con el objetivo del presente libro.

Los sistemas de memoria distribuida pueden ser interconexiones entre estaciones de trabajo, Pcs con diferentes sistemas operativos, etc. Normalmente suelen interconectarse en una red LAN Ethernet a través de un *switch* o conmutador.

Puesto que la latencia de la red puede ser alta, hay que tener en cuenta el factor de eficiencia llamado **granularidad del computador paralelo**. Este factor es la relación o cociente entre el tiempo requerido para realizar una *operación básica de comunicación* y el *tiempo requerido para realizar una operación básica de cálculo de los procesos*. De esta forma, la estrategia de programación de los sistemas de memoria distribuida debe tener siempre como objetivo *disminuir la granularidad de las comunicaciones*, es decir, minimizar el número de mensajes y maximizar el tamaño.

Cuando se combinan los sistemas MIMD con programación paralela de paso de mensajes se le denomina sistemas **multicomputadores**.



**Figura 1.15** Arquitectura distribuida

Igualmente, los sistemas de memoria distribuida o multicomputadores pueden ser computadores con múltiples CPUs comunicadas por un bus de datos o bien múltiples ordenadores interconectados a través de un red LAN o la misma Internet, siempre que la velocidad sea más o menos rápida.

En el primer caso hablaremos de *sistemas masivamente paralelos* (MPP), y en el segundo de *clusters*, como podría ser una colección de estaciones de trabajo, junto con PCs interconectados por una red de comunicaciones rápida.

Los *clusters* pueden ser de dos clases:

- *Beowulf*: Cada computador del *cluster* está dedicado exclusivamente al mismo.
- *NOW*: Red de estaciones de trabajo. Sus computadores no están exclusivamente dedicados al *cluster*..

Los sistemas ***Beowulf*** están compuestos por nodos minimalistas<sup>2</sup> interconectados en una red de comunicaciones de bajo coste, cuya topología está orientada a la resolución de un determinado problema de computación, y donde cada nodo se dedica de manera exclusiva a procesos del supercomputador.

La red de interconexión de los sistemas ***NOWs*** de redes de interconexión de estaciones de trabajo suelen utilizarse *switches*, mientras que en los *Beowulf* la interconexión se realiza de manera barata, por medio de tarjetas Ethernet con un conector RJ-45 y cable Ethernet 100. Por último, la programación en *Beowulf* se caracteriza porque es muy dependiente de la arquitectura y siempre se utiliza el modelo de programación paralela basada en paso de mensajes.

---

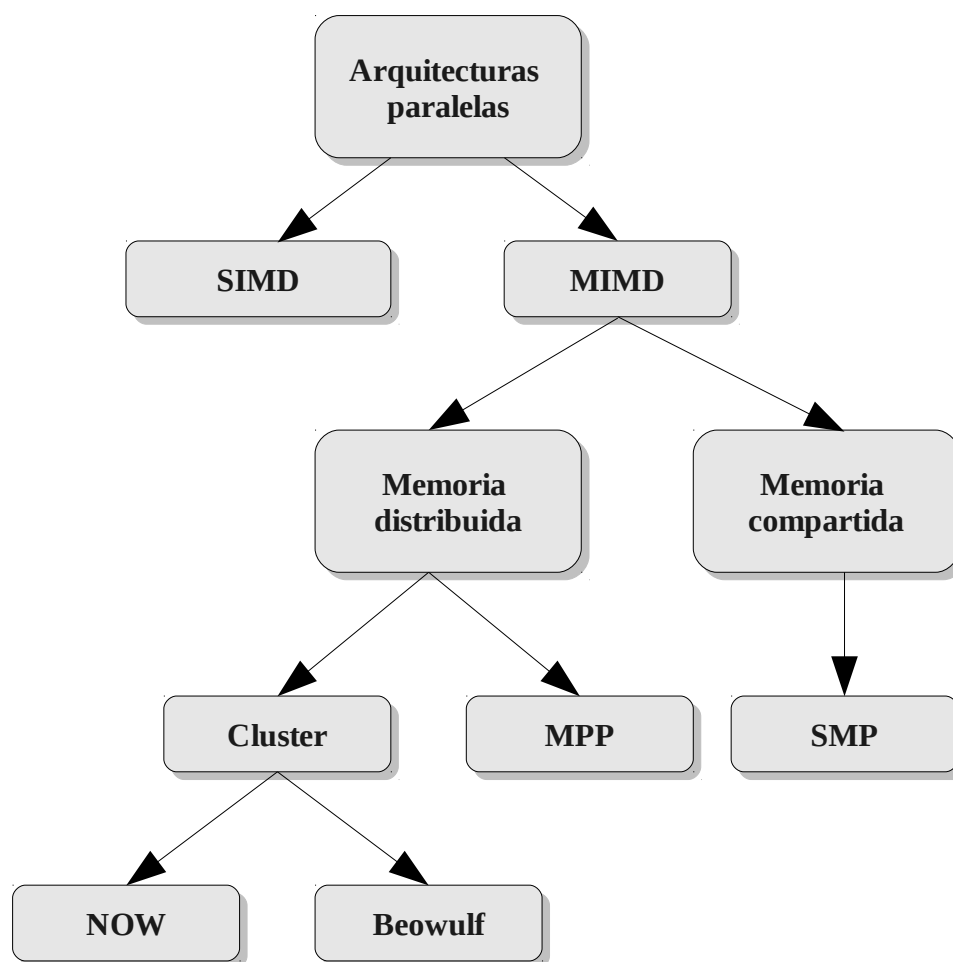
<sup>2</sup> Los nodos minimalistas constan de una placa base, CPU, memoria y dispositivos de E/S.



### 4.3 Conclusión

La ventaja de los *clusters* con respecto a las arquitecturas como NUMA es que es sencillo y poco costoso construir un sistema paralelo con varios ordenadores interconectados en red, de forma rápida y con alto rendimiento en computación. El único problema estriba en el modelo de programación: mientras que en los sistemas de memoria compartida es solo necesario establecer unas pocas directivas al compilador, en los sistemas de memoria distribuida hay que programar explícitamente todas las comunicaciones.

Atendiendo a las diferentes clasificaciones expuestas anteriormente y siguiendo la *taxonomía de Flynn*, puede diseñarse un breve diagrama a modo de resumen:



**Figura 1.16** Diagrama de clasificación de las arquitecturas paralelas<sup>3</sup>

3 MPP: Procesadores Masivamente Paralelos  
SMP: Multiprocesadores Simétricos

## 5. Modelos de programación paralela

Una vez vistas las diferentes arquitecturas en las que se basan los computadores paralelos y su organización de memoria, es ahora factible conocer a grandes rasgos los diferentes modelos más comúnmente usados de programación en estos sistemas.

Para comenzar, los modelos de programación existen como una abstracción sobre las arquitecturas hardware y software. Sin que decir que estos modelos se pueden combinar para crear otros modelos basados en sus antecesores, y, aunque no lo parezca, los modelos que veremos a continuación no son específicos de una determinada arquitectura.

Qué modelo es el más adecuado para usar es a menudo determinado por cual es más adecuado para el dominio del problema y cual es preferible por el programador. No hay ningún modelo mejor, aunque ciertamente existen unos modelos mejores que otros.

### 5.1 Modelo de memoria compartida

En este modelo las tareas comparten un espacio común de direcciones en la que leen y escriben de manera asíncrona, pudiendo sincronizarlas mediante el uso de semáforos<sup>4</sup> para evitar común a memoria compartida.

Una ventaja de este modelo es que el programador no tiene noción de la propiedad de los datos, por lo que no es necesario especificar la comunicación entre tareas y el trabajo para el programador se simplifica. Sin embargo, es importante recalcar que es más difícil entender y controlar la localidad de los datos, por lo que esto puede afectar al rendimiento.

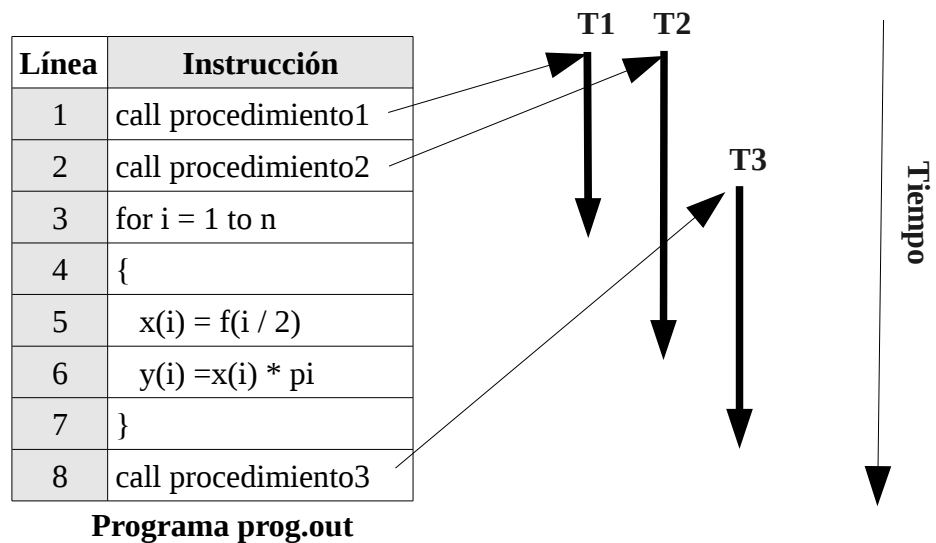
Actualmente no existen actualmente implementaciones de este modelo

### 5.2 Modelo de threads (hilos)

En el modelo de threads, un único proceso puede tener múltiples vías concurrentes de ejecución.

---

<sup>4</sup> Generalmente se implementa con la primitiva lock



**Figura 1.7** Ejemplo de programación con modelo de threads

Como podemos ver el programa **prog.out** es ejecutado en el sistema operativo y obtiene los recursos del sistema que le son necesarios. **Prog.out** lleva cierta ejecución en serie y después crea un número determinado de threads (hilos) que son gestionados por el sistema operativo y pueden verse como llamadas a procedimientos. Cada thread contiene sus datos locales, aunque también comparten sus recursos, mientras se benefician de la memoria global ya que comparten el espacio de memoria de **Prog.out**.

Los threads de **prog.out** utilizan la memoria global para comunicarse, y consecuentemente necesitan algún método de sincronización para asegurarse que más de un thread no esté escribiendo en la misma posición de memoria.

El modelo de threads se le asocia comúnmente al modelo de arquitectura de memoria compartida y programación concurrente en sistemas operativos.

Las implementaciones más conocidas de threads son las siguientes:

#### Threads POSIX:

- Está basado en librerías y en el lenguaje C.
- Se les referencia como Pthreads.
- El paralelismo se gestiona de forma explícita, por lo que implica que el programador ponga atención en los detalles del código.

### OpenMP

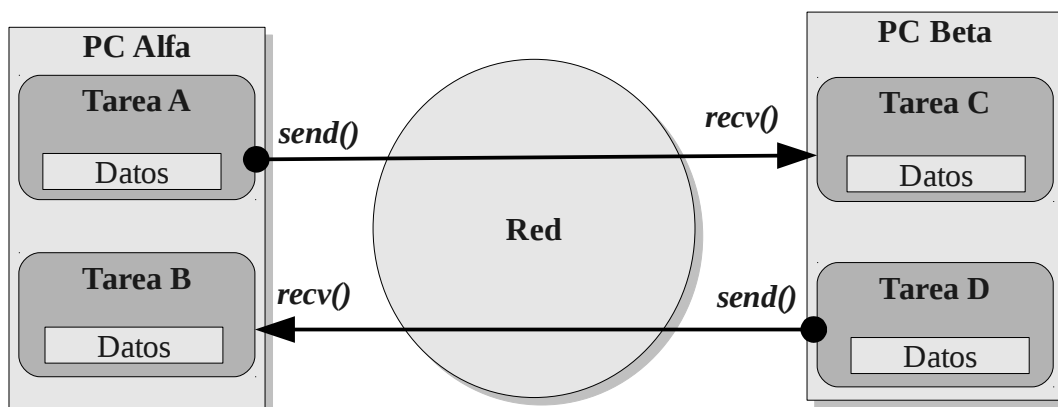
- Como CUDA, está basado en directivas del compilador, por lo que puede ser usado para generar código serie.
- La API está implementada para los lenguajes Fortran y C++
- Multiplataforma y escalable
- Generalmente se utiliza para programar arquitecturas de memoria compartida

```
int main(int argc, char *argv[]) {  
    const int N = 100000;  
    int i, a[N];  
  
    #pragma omp parallel for  
    for (i = 0; i < N; i++)  
        a[i] = 2 * i;  
  
    return 0;  
}
```

**Código 1.1** Ejemplo en OpenMP que inicia un array en paralelo

### 5.3 Modelo de paso de mensajes

En este modelo las tareas utilizan su propia memoria local durante la computación. Múltiples tareas pueden residir en la misma máquina física y/o a través de un número de un número de máquina arbitrario. De esta manera, las tareas intercambian sus datos entre ellas por mensajes de comunicación de envío y recibimiento. Las transferencias de datos requieren cooperación entre las primitivas<sup>5</sup> implementadas en cada proceso. Así si se lleva a cabo un operación de envío en una máquina, requiere que en otra máquina exista la primitiva correspondiente de recibimiento.



**Figura 1.8** Ejemplo del modelo de paso de mensajes

---

<sup>5</sup> Estas primitivas suele denominarse send y recv

Respecto a la implementación, el modelo de paso de mensajes consta de un conjunto de librerías que se enlazan en el compilador de la plataforma específica donde se implemente el código fuente y que contienen la implementación de la API y primitivas de paralelismo.

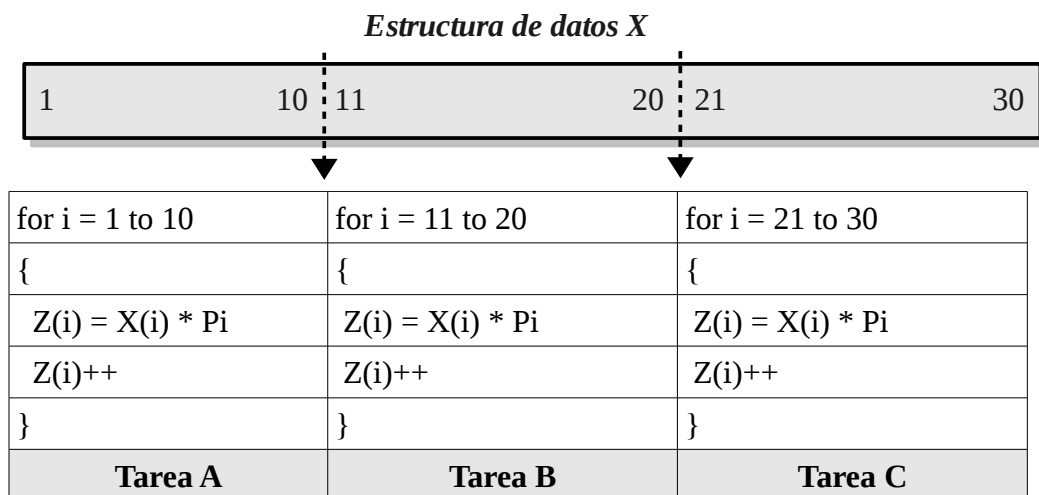
Cuando se diseña una aplicación con paso de mensajes es el programador el que se encarga de determinar todo el paralelismo.

Desde los años ochenta han habido muchas implementaciones de este modelo; aunque las más conocidas han sido **PVM**<sup>6</sup> (Parallel Virtual Machine) y **MPI**<sup>7</sup> (Message Passing Interface).

## 5.4 Modelo de paralelismo a nivel de datos

En el modelo de paralelismo de datos se caracteriza por su enfoque en las operaciones sobre conjuntos de datos. Estos datos se suelen organizar en estructuras de datos tales como arrays o cubos (arrays tridimensionales). De esta forma, un determinado número de tareas trabajan colectivamente sobre una parte diferente de esa estructura de datos. Este modelo será aplicado sistemáticamente a lo largo de los capítulos dedicados a la programación en CUDA.

Como vemos en la figura 1.9, la característica principal es que las tareas siempre aplican la misma operación en la parte que les corresponde de la estructura de datos.



**Figura 1.9** Ejemplo del modelo de paralelismo de datos

<sup>6</sup> [http://www.csm.ornl.gov/pvm/pvm\\_home.html](http://www.csm.ornl.gov/pvm/pvm_home.html)

<sup>7</sup> <http://www.mcs.anl.gov/research/projects/mpi/>

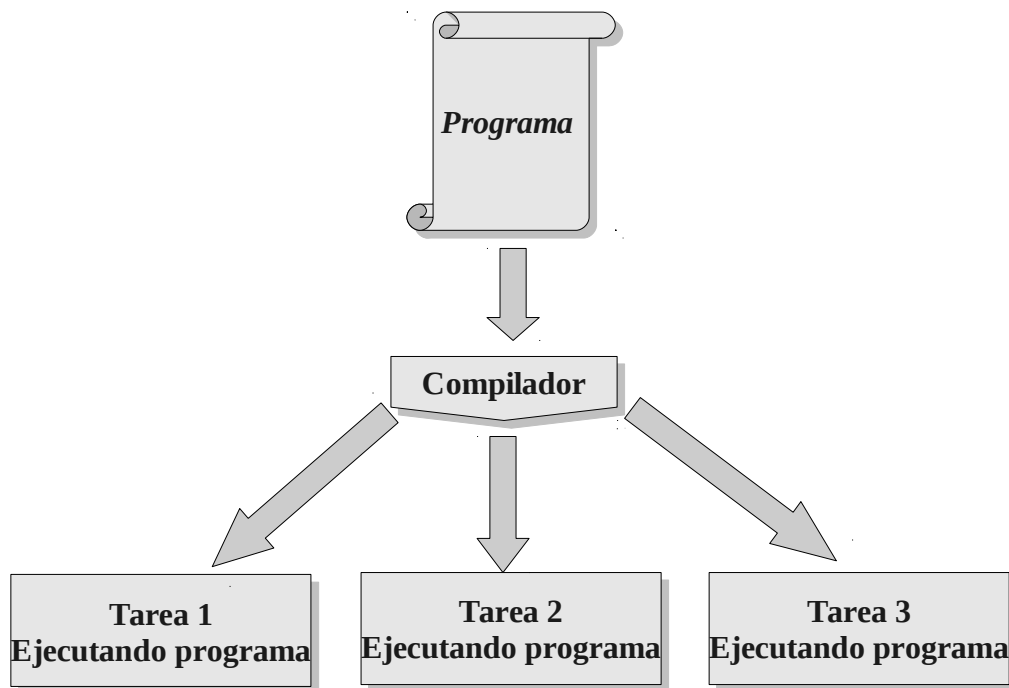
En las arquitecturas de memoria compartida, todas las tareas pueden tener acceso a la estructura de datos a través de la memoria global. En las arquitecturas de memoria distribuida la estructura de datos es dividida y reside en pedazos en la memoria local de cada tarea.

## 5.5 Otros modelos

Existen otros modelos de programación paralela, aunque los más comúnmente usados son los que se han explicado en los apartados anteriores. A continuación veremos dos modelos de entre los más importantes que son también comúnmente utilizados en arquitecturas de memoria distribuida.

### Single Program Multiple Data (SPMD)

En este modelo de programación paralela, al arrancar una aplicación se lanzan  $N$  copias del mismo programa (procesos). Estos procesos no avanzan sincronizados instrucción a instrucción sino que la sincronización debe ser realizada explícitamente.

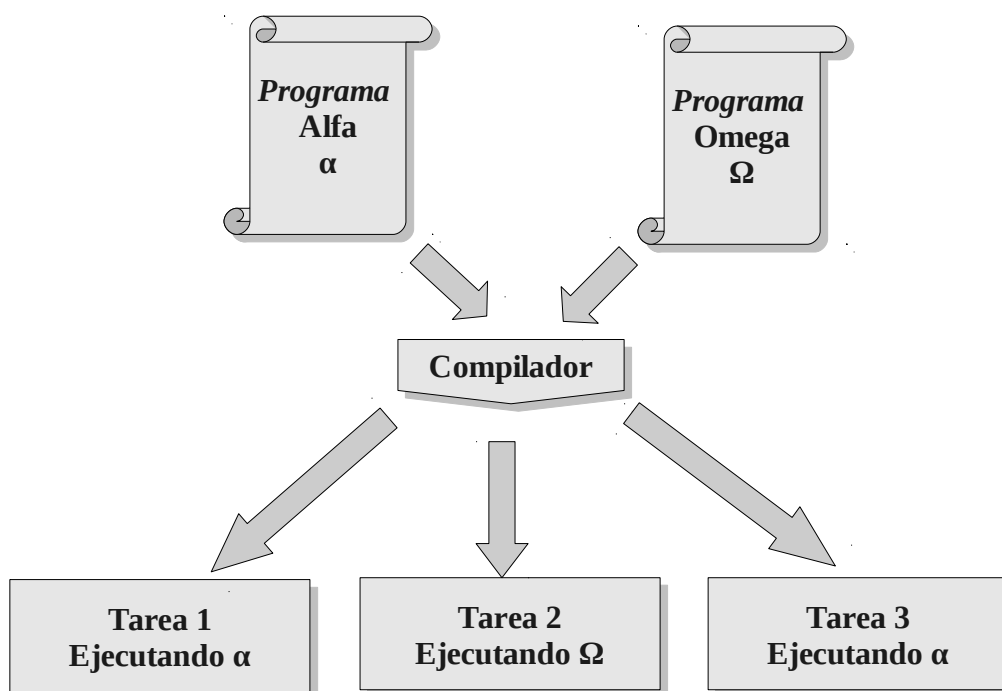


**Figura 1.10** Modelo SPMD ejecutado en la misma plataforma

En cualquier momento, las tareas pueden ejecutar la misma o diferentes instrucciones con el mismo programa y pueden usar diferentes datos.

### Multiple Program Multiple Data (MPMD)

En esta metodología se escribe un programa propio y diferente para cada procesador. En el enfoque *maestro/esclavo* un procesador ejecuta un programa (proceso maestro) que posteriormente arrancará los demás procesos (procesos esclavos). El arranque de esos procesos es relativamente costoso desde una perspectiva computacional. Como es lógico, las tareas pueden usar diferentes datos.



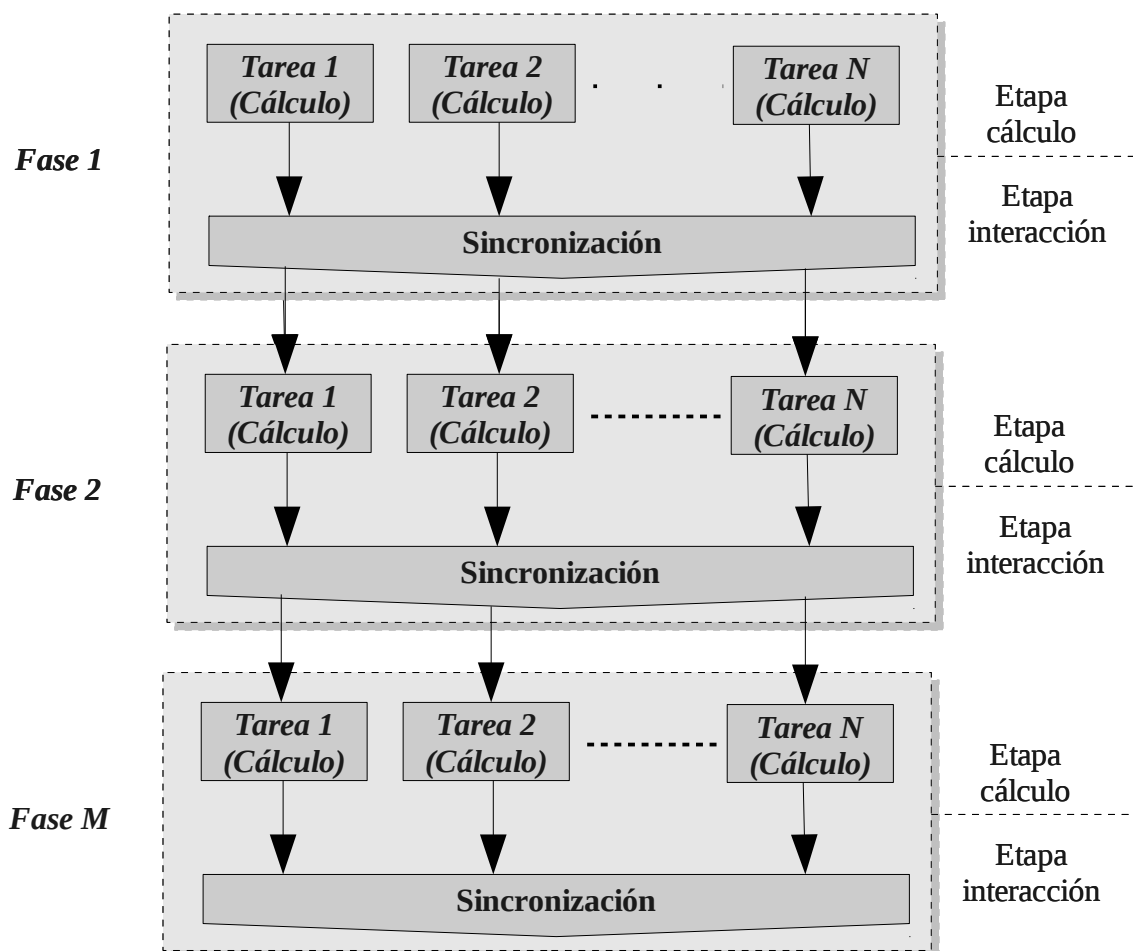
**Figura 1.11** Modelo MPMD ejecutado en la misma plataforma

## 6. Paradigmas algorítmicos de paralelización

A continuación se expondrán las diferentes estrategias algorítmicas para abordar un problema de programación paralela:

### 6.1 Fases paralelas

Se basa en la sucesión de un número arbitrario de fases para llegar a la solución del problema, donde cada fase consta de dos etapas: cálculo e interacción.



**Figura 1.12** Fases paralelas

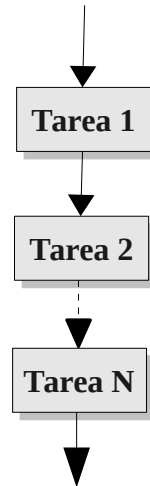
Durante la etapa de cálculo, cada tarea realiza sus operaciones independientemente. Cuando se termina esta etapa se pasa a la etapa de interacción, en la que se sincronizan las tareas con primitivas con el objetivo del intercambio de información y resultados entre tareas. Finalmente, después de la etapa de sincronización se prosigue en la siguiente fase.



## 6.2 Segmentación (pipeline)

Las tareas implicadas en el paralelismo se comportan como etapas de una segmentación virtual (de manera parecida a las arquitecturas segmentadas) en la que los datos avanzan continuamente, por lo que la ejecución de las etapas se solapan sobre los datos de la carga de trabajo.

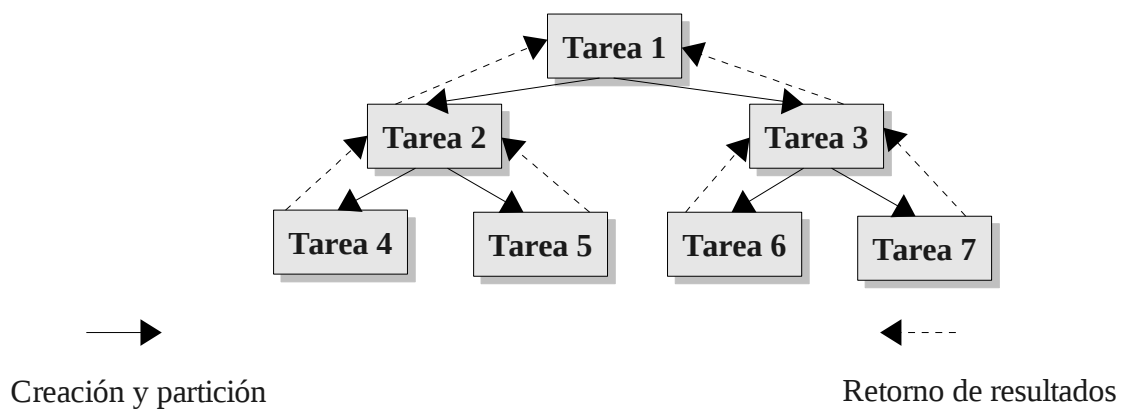
*Flujo de datos*



**Figura 1.13** Segmentación

## 6.3 Divide y vencerás

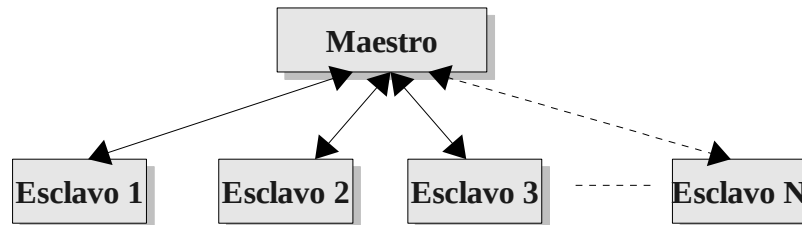
La carga de trabajo es dividida por los padres en los sucesivos hijos, que a su vez vuelven a dividir la carga de trabajo. Por último, el proceso padre es el encargado de recoger todos los resultados parciales de los hijos y combinarlos para dar un resultado final. Este método también se le denomina arborescente, por su descomposición arbórea.



**Figura 1.14** Divide y vencerás

#### 6.4 Maestro-esclavo (host-node)

En este paradigma, la tarea maestra realiza la parte secuencial del algoritmo paralelo y lanza sus tareas esclavas, que ejecutan de manera paralela la carga de trabajo que representa el problema.



**Figura 1.15** Maestro-esclavo

Este paradigma está basado en tres fases. La primera se encarga de la inicialización de los grupos de procesos y la distribución de la carga de trabajo. La segunda fase se encarga del cálculo del problema en sí, y la última se encarga de la recopilación y visualización de la solución.

## 7. Breve historia de la computación paralela

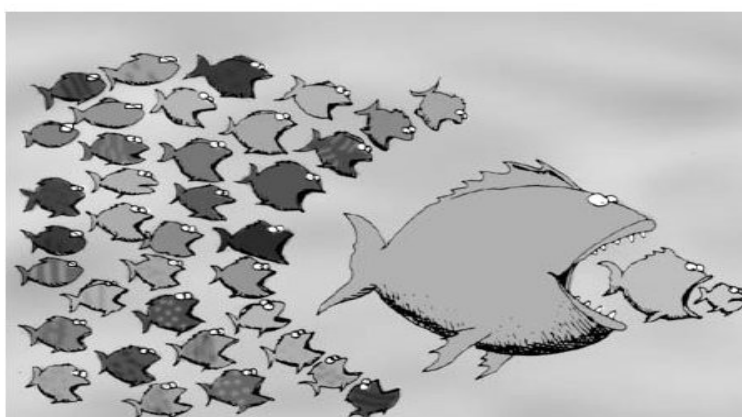
La historia de la computación paralela se encuentra muy entrelazada con la propia historia de la computación en general. Si cierto es que los orígenes de la computación ya se remontan a tiempos pretéritos, por ejemplo, con el invento del ábaco, en el caso de la computación paralela sus inicios se podría establecer en torno a mediados del siglo XX con el desarrollo del primer computador de segunda generación transistorizado.

En 1958, Gill ya escribió sobre las bases de la programación paralela y una año después Holland planteó la posibilidad pudiera ejecutar un número determinado de programas simultáneamente. En 1963 Conway describe el diseño de un computador paralelo su programación.

Pero no fue hasta 1981 cuando se presentó el primer sistema paralelo comercial, el **Butterfly**, distribuido por **BBN Computers Advanced**. Capaz de dividir su trabajo entre 256 procesadores, en concreto microprocesadores **Motorola 68000**, que se conectaban a través de una red multietapa con memorias de 500Kbytes por procesador. Se consiguieron vender 35 máquinas, la mayoría a universidades y centros de investigación.

En los años ochenta y principios de los noventa comenzó la denominada era de la época dorada de la programación paralela. Particularmente en el paralelismo a nivel de datos. Entre las arquitecturas más destacadas caben citar la **Connection Machine**, **MasPar** y **Cray**, así como verdaderos supercomputadores, increíblemente exóticos, poderosos y muy costosos. Esto derivó en lo que se denominó época oscura de la computación paralela, puesto que fue difícil vender estos equipos tan potentes. La industria y los centros de investigación se vieron forzados a detener sus actividades. La solución fue reemplazar las costosas máquinas masivamente paralelas por clusters, grids, etc.

En los últimos años el rendimiento de los computadores paralelos ha aumentado significativamente. El último fenómeno es la democratización de la programación paralela, con la llegada de las GPUs multinúcleo a todos los hogares.



**Figura 1.16** Evolución de la computación paralela

# 2

## Arquitectura de CUDA

### 1. Introducción

Una vez entendidos los conceptos básicos de paralelismo explicados en el capítulo anterior, podemos comenzar a desgranar con detalle los entresijos de la tecnología CUDA de Nvidia.

Comenzaremos en este capítulo con la estructura que define un sistema CUDA, sus núcleos, memoria y su modelo de abstracción del paralelismo a nivel de hardware.

En noviembre de 2006 Nvidia introduce la *Arquitectura Unificada* (CUDA) con la presentación de su modelo GeForce 8800. Desde entonces han aparecido en el mercado diferentes modelos también compatibles tales como los modelos Tesla y Quadro. Estos modelos eran muy similares, diferenciándose principalmente en el ancho de banda del bus, la cantidad de memoria, los núcleos y la cantidad de registros.

Las nuevas GPUs de Nvidia se construyeron a partir de la replicación de un bloque constructivo básico, como acelerador con memoria integrada. La principal ventaja de las nuevas GPUs multinúcleo se basan en una jerarquía de *threads* mapeados sobre el hardware, que consiguen un paralelismo transparente y eficiente. Además, permite creación, planificación y ejecución transparente de miles de *threads* de manera concurrente.

Una de las ventajas de las GPUs multinúcleo es la liberación de la CPU de carga de trabajo que pueden ser delegadas específicamente a la tarjeta gráfica. De esta forma, se consigue mayor rendimiento para ciertas tareas, así, por ejemplo, podemos comprobar el uso de esta tecnología en los siguientes tipos de problemas:

- Multiplicación de matrices densas
- Tratamiento digital de señales, especialmente vídeo
- Cálculo de potencial eléctrico
- Resolución de ecuaciones polinomiales

Aunque parezca que todos los tipos de problema o algoritmos se podrían adecuar a una implementación en CUDA, esto no es cierto, pues se considera una falacia pensar que todos los algoritmos son susceptibles de adaptarse al paralelismo de las GPUs. Así, mientras que las CPUs se orientan a optimizar la ejecución de aplicaciones de propósito general, las GPUs son dedicadas a obtener un mayor *rendimiento pico* y de procesamiento gráfico.

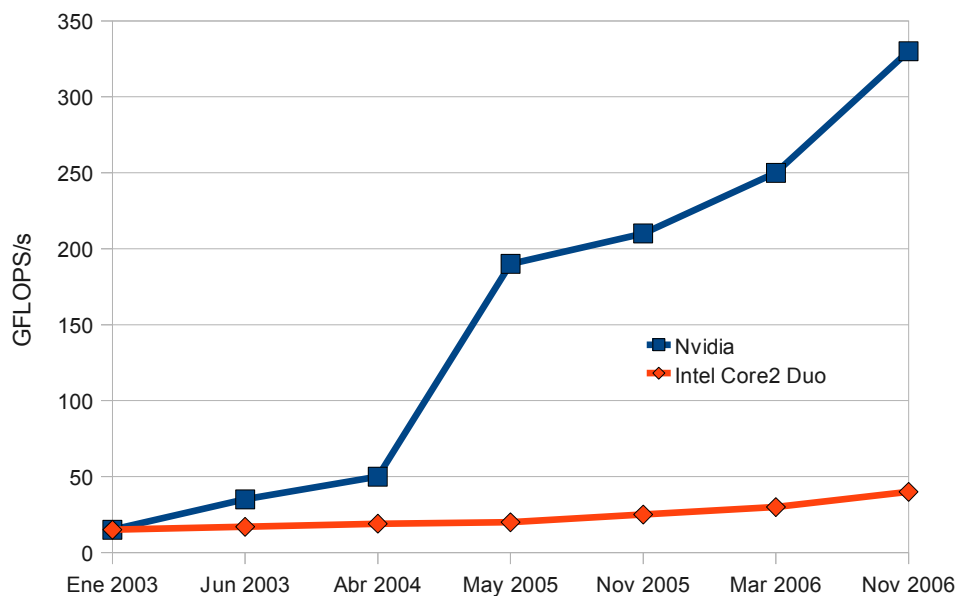
Actualmente, podemos encontrar GPUs con un rendimiento aproximado de unos

500Gflops<sup>1</sup>.

Puesto que las GPUs liberan a la CPU de carga de trabajo gráfica, esto ha propiciado un auge en el mundo de la tecnología de videojuegos, y ahora mismo las GPUs multinúcleo se pueden encontrar en ordenadores de sobremesa o servidores integradas en la placa o como tarjetas externas.

Como veremos en próximos capítulos capítulo las tarjetas Nvidia están bien integradas con las tecnologías de software gráfico como son DirectX y OpenGL.

Por último, y puesto que las GPUs se están volviendo cada vez más rápidas, presentaremos una gráfica de la evolución de las tarjetas de vídeo en esta última década, antes de entrar en los recovecos del laberinto de CUDA.



**Figura 2.1** Evolución de las tarjetas gráficas Nvidia

<sup>1</sup>  $10^9$  operaciones en punto flotante

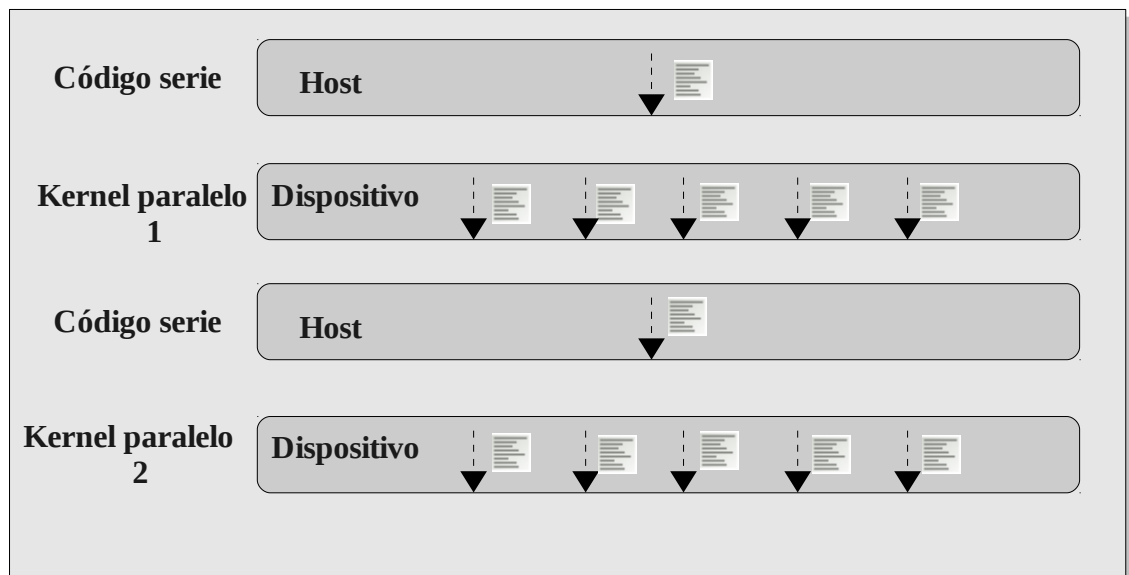
## 2. Entrando en la GPU

GPU son las siglas en ingles que se establecieron como acrónimo de *Graphics Processing Unit* o traducido al castellano, *Unidad Procesamiento Gráfico*. Mucho ha llovido desde los inicios de la informática gráfica, desde el primer hardware monocromático integrado en las placas base hasta las modernas tarjetas como las que trata este libro. Su aparición no es exclusiva del PC, sino que también podemos encontrar tarjetas en una gran variedad de sistemas como videoconsolas, PDAs y teléfonos móviles.

Las actuales GPUs de Nvidia se caracterizan por su chips con tecnología *multithread masivo en multinúcleo*. En el caso del modelo Tesla el número de procesadores escalares llega a 128, con 12.000 threads concurrentes y un rendimiento sostenido de 470 GFLOPS.

Esta tecnología no ha pasado desapercibida a los científicos e ingenieros donde han conseguido incrementar en 100 veces su potencia de cálculo.

La característica principal de la programación en CUDA es que permite combinar la implementación de código serie en el *host*<sup>2</sup> con la implementación de código paralelo en el *dispositivo*. De esta manera conseguiremos entrelazar ejecuciones serie con ejecuciones paralelas en los *kernel*.



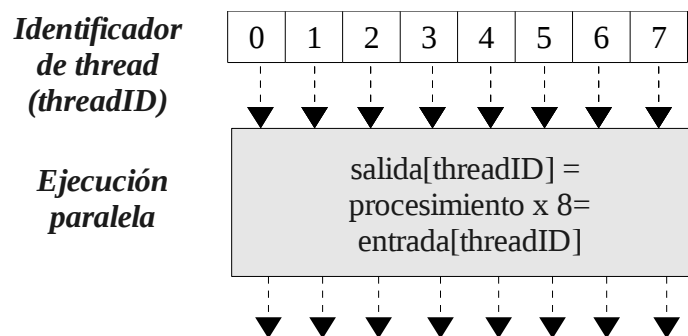
**Figura 2.2** Modelo de ejecución en CUDA

---

<sup>2</sup> Nos referiremos al *host* como el sistema del usuario donde reside la CPU y al *dispositivo* como la GPU Nvidia

La ejecución en la parte del dispositivo se realiza por un conjunto finito de threads paralelos que actúan sobre una parte diferente de los datos. De esta forma podemos definir como **kernel** a un conjunto determinado de muchos threads concurrentes, donde sólo un **kernel** es ejecutado al mismo tiempo en el dispositivo, y muchos threads participan en la ejecución del **kernel**. Cada thread tiene su propio identificador para poder realizar un paralelismo transparente sobre los datos. Los threads en CUDA pueden ser de dos clases:

- *Thread físicos*: Son los threads de la GPU Nvidia, donde la creación y cambio de contexto de los threads de la GPU son esencialmente libres.
- *Threads virtuales*: Donde, por ejemplo, un núcleo de la CPU podría ejecutar múltiples threads CUDA.



**Figura 2.3** Ejecución paralela de un grupo 8 de threads

Como puede observarse en la figura 2.3, los 8 threads lanzados realizan un procesamiento sobre la zona respectiva de los datos a que cada thread identifica, almacenando la salida en cada zona de datos respectiva a cada identificador.

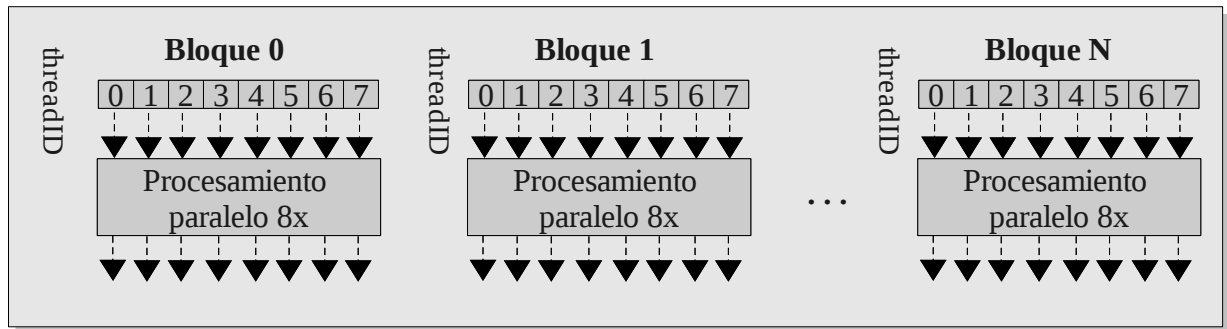
Los threads se pueden agrupar en bloques, de esta forma podemos entender a un **kernel** como un *grid* de bloques de threads. Debemos tener presente que la sincronización de threads dentro de un bloque debe realizarse con barreras (*barriers*):

```

2  salida[threadID] = entrada[threadID];
   _syncthreads();
   float xscale = salida[threadID - 1];
1
3

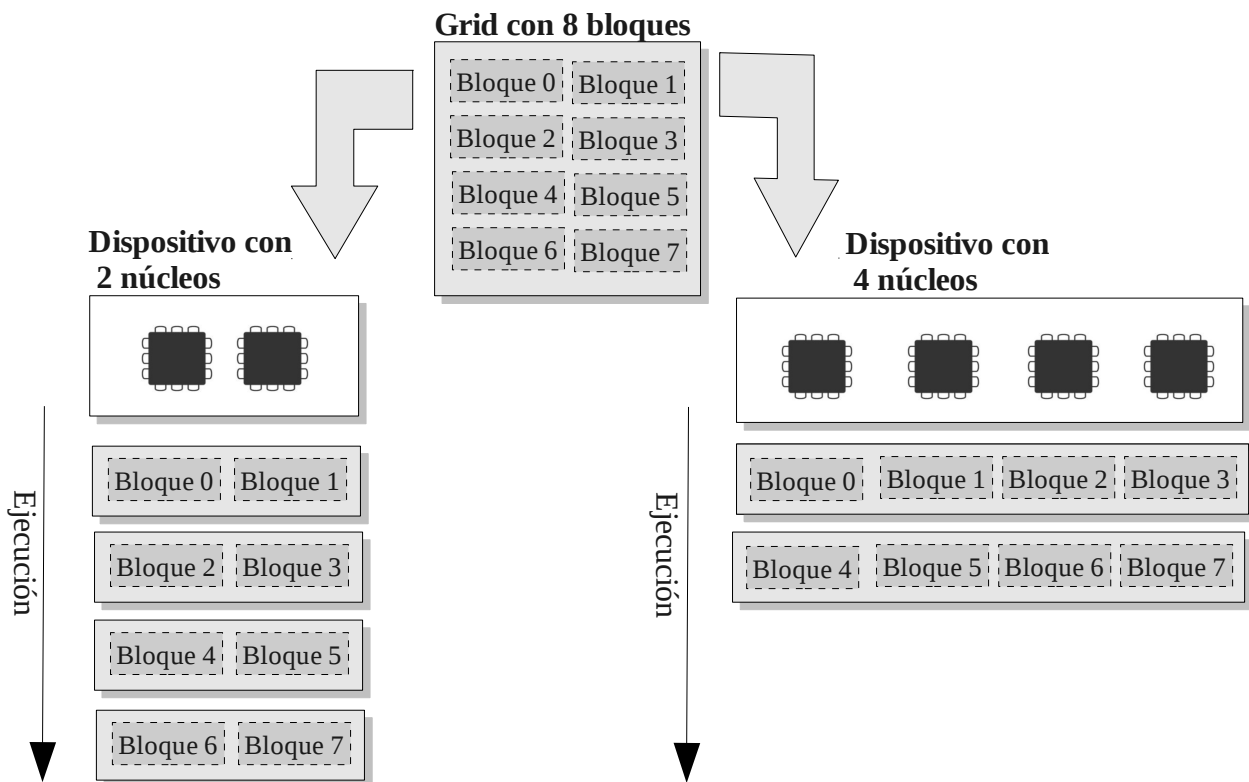
```

En este ejemplo, los threads implicados en la parte (1), después de realizar su procesamiento deben esperar hasta que todos los threads de su grupo alcancen la primitiva `_syncthreads`, parte (2), para poder entrar en la nueva parte de procesamiento (3).



**Figura 2.4** Grid de N bloques de threads

Hay que considerar, a partir de este modelo que mientras los threads de dentro de un bloque sí se pueden sincronizar, los bloques de threads no permiten sincronización entre ellos; aunque la arquitectura de CUDA permite su ejecución en cualquier orden, concurrentemente o en serie. Por lo tanto, se considera que existe una barrera de sincronización implícita entre dos *kernels* lanzados consecutivamente. Como consecuencia, esta independencia proporciona una gran escalabilidad, donde los *grids* se escalan dependiendo del número de núcleos paralelos.



**Figura 2.4** Gestión de la escalabilidad



Respecto a la jerarquía de memoria es imprescindible destacar que existen tres tipos de memoria que caracterizan la arquitectura CUDA. La primera de ellas es la *memoria local* dedicada a cada thread. Ésta es sin duda la parte de la memoria que es privativa de un único thread y se encuentra dentro del ambiente de la función que implementa el *kernel*. Es la memoria para datos locales y utilizada exclusivamente para operaciones implementadas en la misma función. En segundo lugar tenemos la *memoria compartida* para los threads. Este tipo de memoria permite que varios threads compartan información entre ellos cuando se ejecutan dentro de un mismo *kernel*, permitiendo principalmente optimizar el rendimiento. Por último tenemos la *memoria global* reservada para propósitos de almacenamientos de datos de entrada y resultados del *kernel*. En la siguiente imagen podemos distinguir los diferentes tipos:

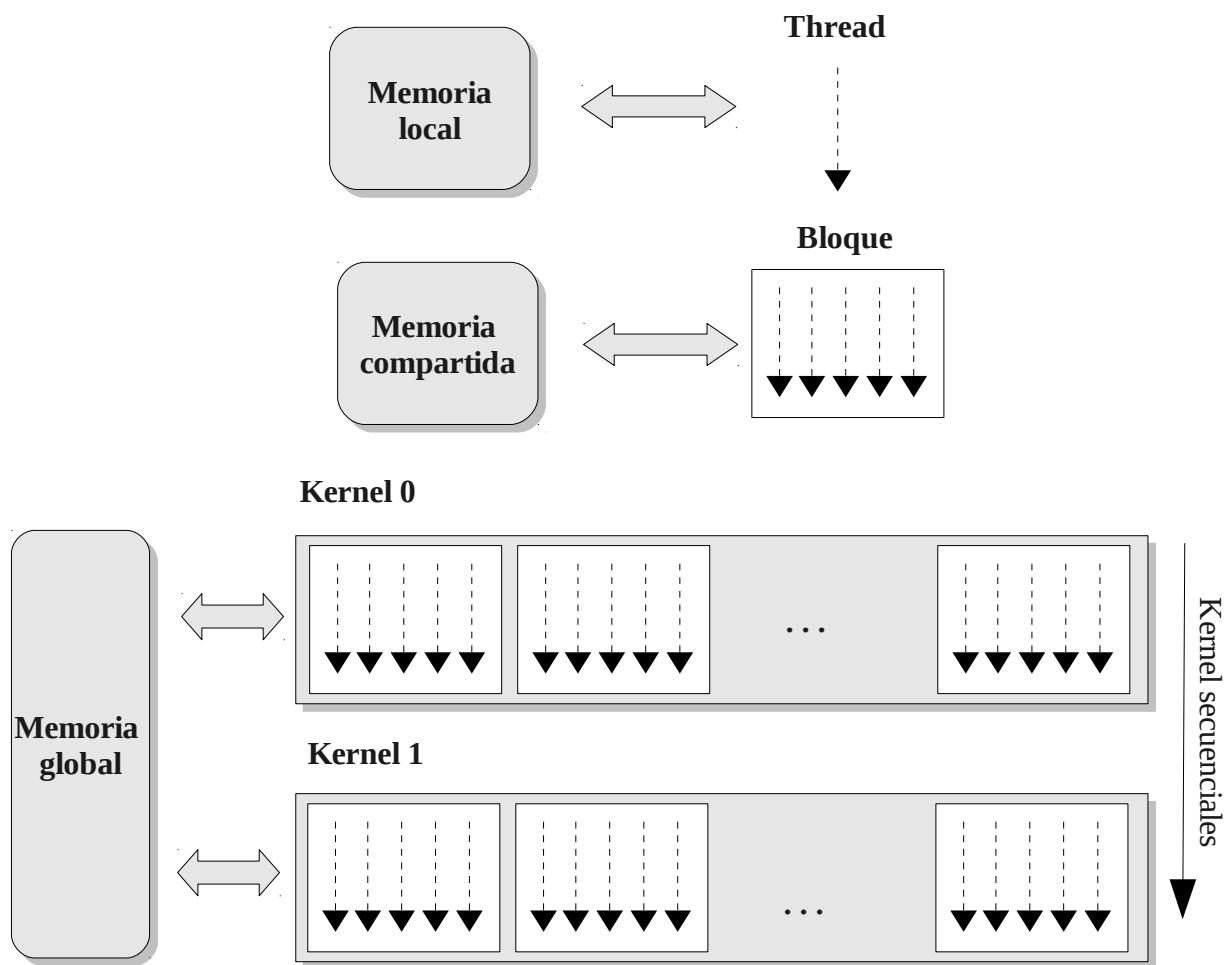


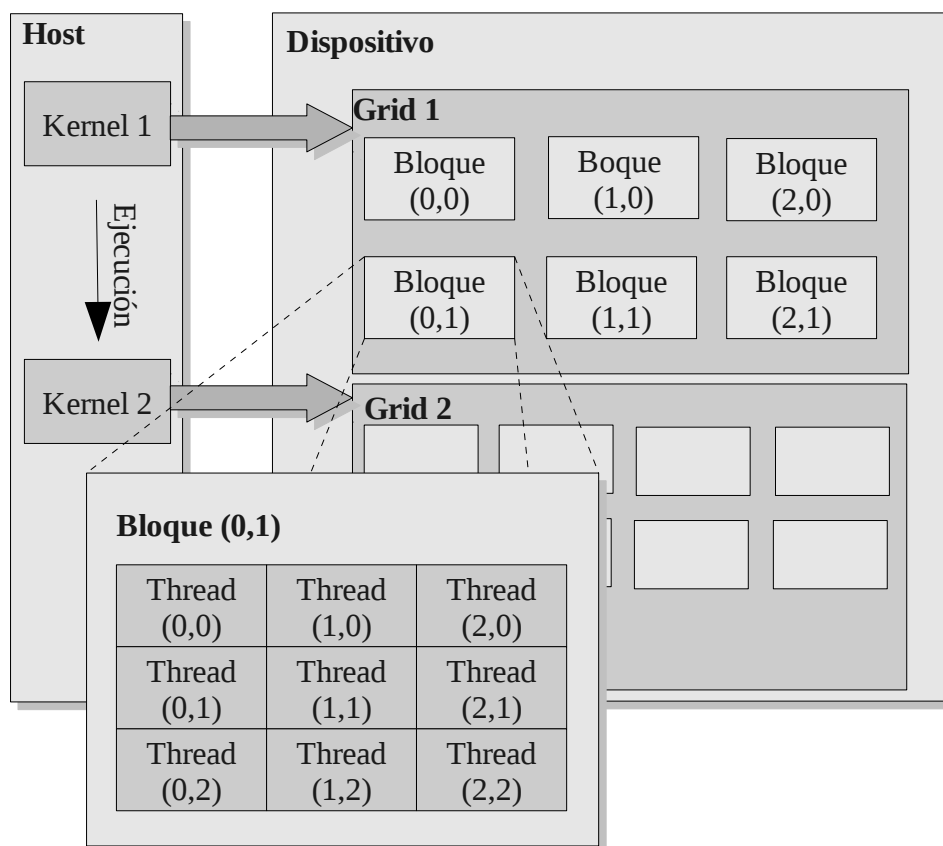
Figura 2.5 Jerarquía de memoria

### 3. CUDA en profundidad

Ahora que hemos visto las características principales de la arquitectura CUDA, pasaremos a describir con más detalle cada una de las partes vistas anteriormente.

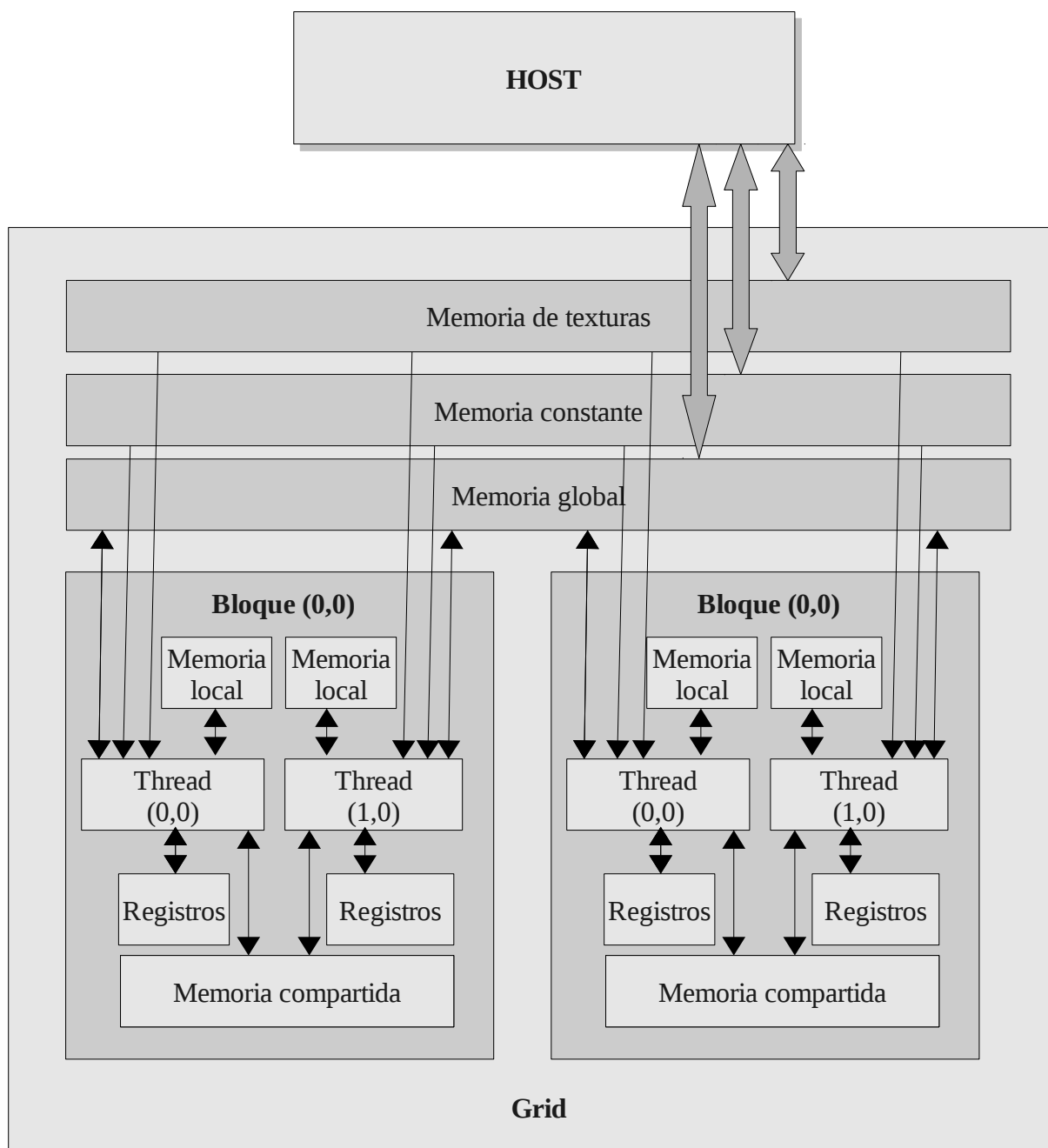
Proseguiremos con la estructura de threads y bloques de un *kernel*. Como hemos comentado antes, tanto los threads como los bloques contienen un identificador unívoco que permite distinguirlos durante la ejecución, de esta forma, el programador puede por medio de los identificadores decidir sobre qué datos trabajar independientemente de los que hagan otros threads. Así un programador puede decir sobre qué parte de los datos proporcionados al *kernel* debe acceder un thread cuando se ejecute su porción de código que también ejecutarán otros threads con sus identificadores.

Como veremos en próximos capítulos, los identificadores de los bloques pueden ser unidimensionales (1D) o bidimensionales (2D), mientras que los identificadores de los threads pueden ser unidimensionales, bidimensionales o tridimensionales (3D). Esto simplifica enormemente el direccionamiento de memoria para datos multidimensionales, por lo que puede ser muy útil en procesamiento de imágenes y resolución de problemas matemáticos complejos que impliquen matrices.



**Figura 2.6** Ejemplo de distribución de threads y bloques

Con respecto a la distribución de la memoria, fíjese el lector en la figura 2.7:



**Figura 2.7** Detalle de la distribución de memoria

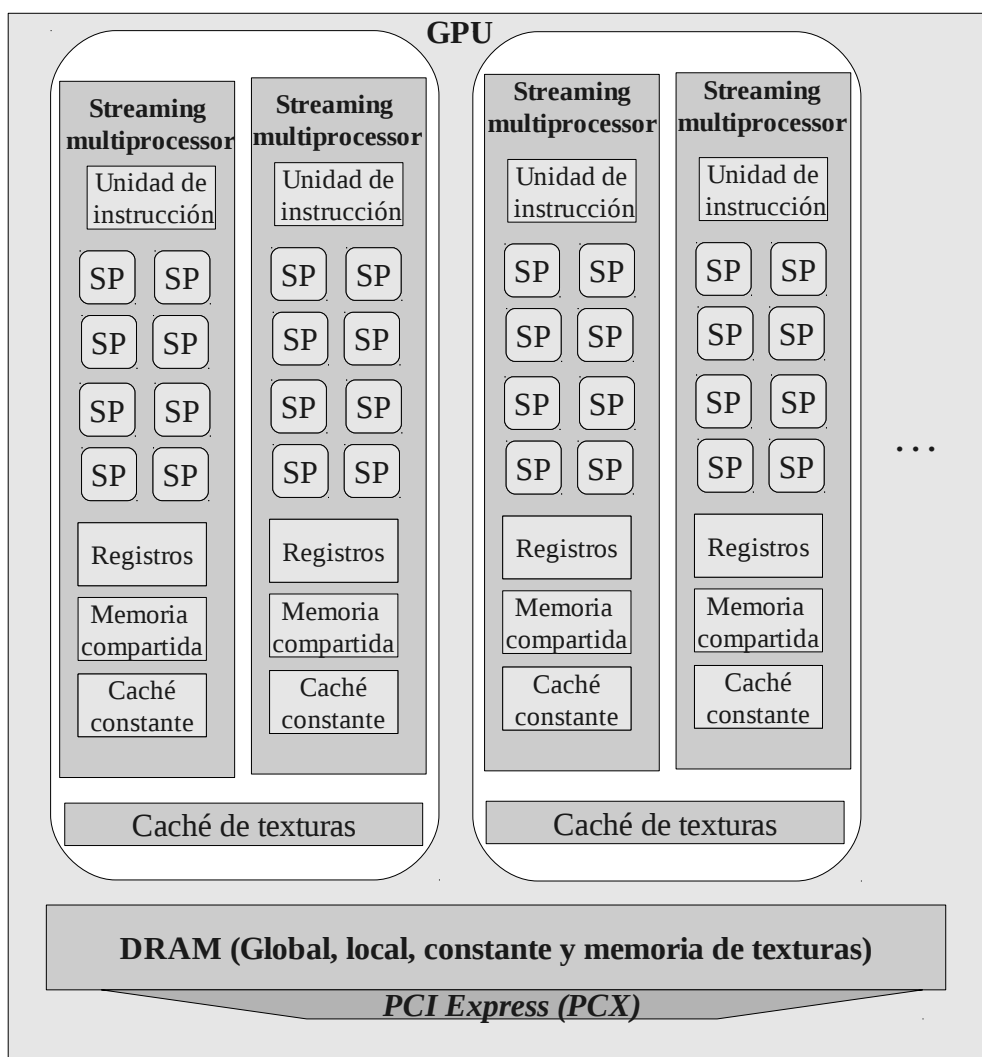
Como el lector podrá apreciar, existen varias clases de memorias en la arquitectura de CUDA. En primer lugar, como puede verse en la figura, existe una estructura de *registros* para los propósitos del thread, utilizada para intercambio de datos entre el procesador y la memoria, así como operaciones escalares. Existen un total de 8192 registros repartidos entre los diferentes *bloques de threads* en ejecución. Los threads pueden leer y escribir en los

registros de propósito general con un tiempo de ejecución muy pequeño. La *memoria local*, como puede verse también en la figura, permite el almacenamiento de datos locales del propio ambiente del *kernel*: los threads pueden leer y escribir en esta memoria los datos que le sean necesarios. En esta memoria privada para cada thread se almacenan datos para la pila y las variables locales. La *memoria compartida*, que tiene una capacidad de 16 KB, permite intercambiar información entre los threads de un mismo bloque, por ejemplo en operaciones de *reducción*. Esta memoria también permite la lectura y escritura desde los threads que pertenecen al mismo bloque.

Por último tenemos tres tipos de memoria más extensas y ubicadas en la memoria DRAM de la tarjeta. Éstas son la *memoria global*, con una capacidad de hasta 1.5 GB, permite tanto a todos los threads como al host leer y escribir en ella en común; la *memoria constante* que permite un alto rendimiento y tiene una capacidad de 64 KB con 8 KB de memoria caché permite a todos los threads leer el mismo valor de memoria constante simultáneamente en un ciclo de reloj. El tiempo de acceso a la memoria constante es similar al de los registros y sólo admite lectura desde los threads y lectura/escritura desde el host. La *memoria de texturas* explota la localidad espacial con vectores de unidimensionales o bidimensionales y el tiempo de acceso elevado pero menos que el de la memoria global. Al igual que la memoria constante, sólo permite lectura desde los threads y lectura/escritura desde el host.

## 4. Hardware de la GPU

La GPU CUDA contiene un conjunto de  $N^3$  *Streaming Multiprocessors*, donde cada *Streaming Multiprocessor* es un conjunto de 8 *Streaming Processors (SP)*. Cada *Streaming Multiprocessor* crea, planifica y ejecuta hasta 24 *warps*<sup>4</sup> pertenecientes a uno o más bloques (768 threads). De esta forma, cada *warp* ejecuta una instrucción en 4 ciclos de reloj.



**Figura 2.8** Hardware de la GPU con CUDA

Como vimos en la figura 2.4, cada bloque se asigna a un núcleo o *Streaming Multiprocessor*, y un *Streaming Multiprocessor* asigna a cada bloque los recursos necesarios (contextos de threads, registros, etc.).

Los *Streaming Processors* realizan operaciones escalares sobre tipos de datos simples de enteros y reales de 32 bits, así mismo, cada uno ejecuta threads independientes,

<sup>3</sup> En el dispositivo G80 hay un total de 16 multiprocesadores

<sup>4</sup> Un bloque se divide en grupos de 32 threads denominados *warps*

aunque todos los *Streaming Processors* deberían ejecutar la instrucción leída por la *Unidad de Instrucción* en cada instante, de esta manera se sigue un modelo de arquitectura SIMT (*Single Instruction Multiple Thread*), donde un conjunto de threads ejecuta las misma instrucción apuntada en el *kernel*, explotando el paralelismo de datos, y en menor medida el de tareas. Los threads son gestionados por el hardware sin necesidad del que el programador realice estas tareas complejas.

# 3

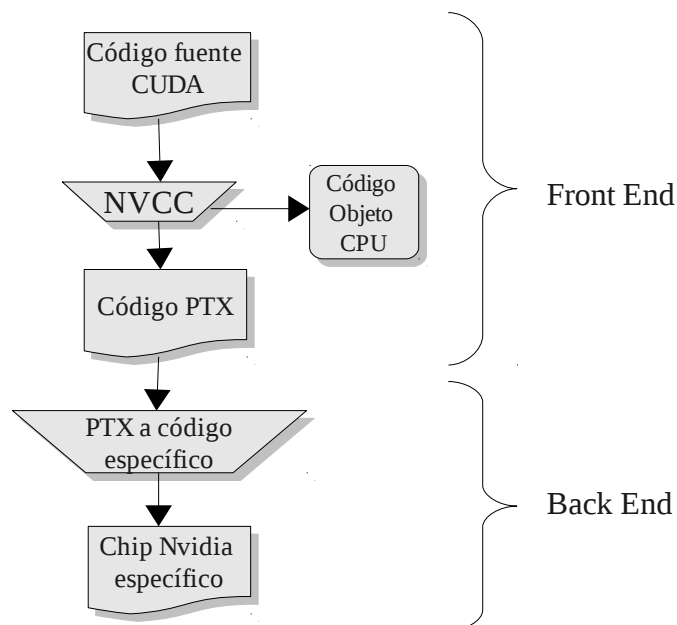
## Introducción a la programación en CUDA

### 1. Conceptos preliminares

Antes de comenzar nuestro viaje por los mundos de la programación paralela en CUDA es necesario tener presentes algunos conceptos básicos sobre el modo de diseño de programas con esta tecnología.

Los programas en CUDA deben estar escritos en lenguaje C, sabiendo de antemano que hay un conjunto de palabras reservadas definidas para la programación en CUDA y que sólo el compilador podrá traducir. Obviamente, aunque la sintaxis de CUDA es básicamente C, es posible, y de hecho así se hará en este libro utilizar la ampliación del lenguaje C++. Por eso, si el lector no posee conocimientos previos de estos lenguajes, puede consultar la bibliografía del final del libro donde se le proporcionará una variedad de fuentes donde podrá iniciarse en un tiempo prudencial en este lenguaje.

Puesto que los programas en CUDA están escritos en lenguaje C es posible intercalar sentencias con instrucciones propiamente del compilador de CUDA con instrucciones estándar C/C++.

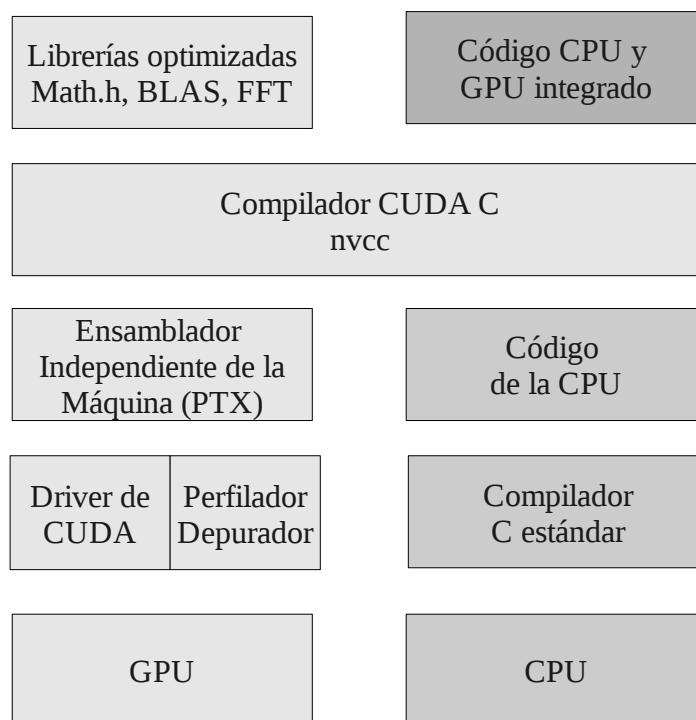


**Figura 3.1** Compilación en dos etapas

En la figura 3.1 se muestra las dos fases de compilación de un programa en CUDA.

El compilador NVCC discriminará entre el código escrito en lenguaje C y C++ y el lenguaje propiamente de CUDA, generando de esta manera dos salidas: código máquina para la CPU específica de la arquitectura en la que trabaja el sistema operativo y código PTX, que es un especie de código ensamblador independiente de la máquina, generado por la parte frontal del compilador de Nvidia. Posteriormente será traducido para generar código máquina específico de cada arquitectura de GPU CUDA.

De esta manera podemos definir la estructura de programación en CUDA como se muestra en la figura 3.2.



**Figura 3.2** Estructura de programación en CUDA

## 2. Requerimientos

Los requisitos para desarrollar en CUDA C son los siguientes:

- Un procesador habilitado para CUDA en una tarjeta gráfica Nvidia
- Un driver para la tarjeta Nvidia
- El kit de desarrollo de CUDA
- Un compilador estándar de C/C++



Puede adquirir el driver y el kit de desarrollo en la página web oficial de Nvidia cuya dirección es <http://www.nvidia.com/cuda>. Allí podrá descargar el driver genérico, el kit de desarrollo donde se encuentra el compilador NVCC, las librerías y conjunto de recursos para profundizar en la programación con GPUs CUDA.

Para el compilador genérico puede utilizar Microsoft Visual C++ si se encuentra en Windows, GNU C++ (gcc) si es usted usuario de Linux, y en caso de que utilice un sistema operativo en Mac OS X deberá tener instalada la versión 10.5.7 de “Snow Leopard”. Por último tendrá que instalar el kit de desarrollo de Apple Xcode para poder compilar en C++.

Para facilitar las cosas<sup>1</sup>, en este libro se anima al lector a utilizar el entorno de desarrollo multiplataforma **CodeLite** para C/C++ que permite la especificación de diversos compiladores según el tipo de código fuente. No obstante, si usted tiene suficientes conocimientos de programación es libre de usar cualquier entorno de desarrollo, e incluso, si es una avezado programador, no utilizar ninguno.

No debe olvidar que si su idea es adquirir una tarjeta Nvidia para programar con los conocimientos adquiridos en este libro, no deje de echar un vistazo a la tabla 3.1 donde se listan los modelos de GPU Nvidia con soporte para CUDA.

Nvidia GeForce	Nvidia GeForce Mobile	Nvidia Quadro	Nvidia Quadro mobile
GeForce GTX 590	GeForce GT 555M	Quadro 6000	Quadro 5000M Quadro
GeForce GTX 580	GeForce GT 550M	Quadro 5000	FX 3800M Quadro FX
GeForce GTX 570	GeForce GT 540M	Quadro 4000	3700M Quadro FX
GeForce GTX 560 Ti	GeForce GT 525M	Quadro 2000	3600M Quadro FX
GeForce GTX 550 Ti	GeForce GT 520M	Quadro 600	2800M Quadro FX
GeForce GTX 480	GeForce GTX 480M	Quadro FX 5800	2700M Quadro FX
GeForce GTX 470	GeForce GTX 470M	Quadro FX 5600	1800M Quadro FX
GeForce GTX 465	GeForce GTX 460M	Quadro FX 4800	1700M Quadro FX
GeForce GTX 460	GeForce GT 445M	Quadro FX 4700 X2	1600M Quadro FX
GeForce GTX 460	GeForce GT 435M	Quadro FX 4600	880M Quadro FX
SE GeForce GTS	GeForce GT 425M	Quadro FX 3800	770M Quadro FX
450 GeForce GT 440	GeForce GT 420M	Quadro FX 3700	570M Quadro FX
GeForce GT 430	GeForce GT 415M	Quadro FX 1800	380M Quadro FX
GeForce GT 420	GeForce GTX 285M	Quadro FX 1700	370M Quadro FX
GeForce GTX 295	GeForce GTX 280M	Quadro FX 580	360M Quadro NVS
GeForce GTX 285	GeForce GTX 260M	Quadro FX 570	320M Quadro NVS
GeForce GTX 280	GeForce GTS 360M	Quadro FX 380	160M Quadro NVS
GeForce GTX 275	GeForce GTS 350M	Quadro FX 370	150M Quadro NVS
GeForce GTX 260	GeForce GTS 260M	Quadro NVS 450	140M Quadro NVS
GeForce GTS 250	GeForce GTS 250M	Quadro NVS 420	135M Quadro NVS
GeForce GTS 240	GeForce GT 335M	Quadro NVS 295	130M

1 En el apéndice de este libro se incluye una sección de configuración del entorno de desarrollo

GeForce GT 240	GeForce GT 330M	Quadro NVS 290
GeForce GT 220	GeForce GT 325M	Quadro Plex 1000
GeForce 210/G210	GeForce GT 320M	Model IV
GeForce 9800 GX2	GeForce 310M	Quadro Plex 1000
GeForce 9800 GTX+	GeForce GT 240M	Model S4
GeForce 9800 GTX	GeForce GT 230M	
GeForce 9800 GT	GeForce GT 220M	
GeForce 9600 GSO	GeForce G210M	
GeForce 9600 GT	GeForce GTS 160M	
GeForce 9500 GT	GeForce GTS 150M	
GeForce 9400 GT	GeForce GT 130M	
GeForce 9400 mGPU	GeForce GT 120M	
GeForce 9300 mGPU	GeForce G110M	
GeForce 9100 mGPU	GeForce G105M	
GeForce 8800 Ultra	GeForce G103M	
GeForce 8800 GTX	GeForce G102M	
GeForce 8800 GTS	GeForce G100	GeForce
GeForce 8800 GT	9800M GTX	GeForce
GeForce 8800 GS	9800M GTS	GeForce
GeForce 8600 GTS	9800M GT	GeForce
GeForce 8600 GT	9800M GS	GeForce
GeForce 8600 mGT	9700M GTS	GeForce
GeForce 8500 GT	9700M GT	GeForce
GeForce 8400 GS	9650M GT	GeForce
GeForce 8300 mGPU	9650M GS	GeForce
GeForce 8200 mGPU	9600M GT	GeForce
GeForce 8100 mGPU	9600M GS	GeForce
<b>Nvidia Tesla</b>	9500M GS	GeForce
Tesla C2050/2070	9500M G	GeForce
Tesla M2050/M2070	9400M G	GeForce
Tesla S2050	9300M GS	GeForce
Tesla S1070	9300M G	GeForce
Tesla M1060	9200M GS	GeForce
Tesla C1060	9100M G	GeForce
Tesla C870	8800M GTX	GeForce
Tesla D870	8800M GTS	GeForce
Tesla S870	8700M GT	GeForce
	8600M GT	GeForce
	8600M GS	GeForce
	8400M GT	GeForce
	8400M GS	GeForce
	8400M G	GeForce
	8200M G	

**Tabla 3.1** Lista de GPUs con capacidad para CUDA

Version	GPUs	Modelo de tarjeta
1.0	G80	GeForce 8800GTX/Ultra/GTS, Tesla C/D/S870, FX4/5600, 360M
1.1	G86, G84, G98, G96, G96b, G94, G94b, G92, G92b	GeForce 8400GS/GT, 8600GT/GTS, 8800GT/GTS, 9600GT/GSO, 9800GT/GTX/GX2, GTS 250, GT 120/30, FX 4/570, 3/580, 17/18/3700, 4700x2, 1xxM, 32/370M, 3/5/770M, 16/17/27/28/36/37/3800M, NVS420/50
1.2	GT218, GT216, GT215	GeForce 210, GT 220/40, FX380 LP, 1800M, 370/380M, NVS 2/3100M
1.3	GT200, GT200b	GeForce GTX 260, GTX 275, GTX 280, GTX 285, GTX 295, Tesla C/M1060, S1070, Quadro CX, FX 3/4/5800
2.0	GF100, GF110	GeForce (GF100) GTX 465, GTX 470, GTX 480, Tesla C2050, C2070, S/M2050/70, Quadro Plex 7000, GeForce (GF110) GTX570, GTX580, GTX590
2.1	GF108, GF106, GF104, GF114, GF116	GeForce GT 420, GT 430, GT 440, GTS 450, GTX 460, GTX 550 Ti, GTX 560 Ti, 500M, Quadro 600, 2000, 4000, 5000, 6000

**Tabla 3.2** Tabla de capacidad de cómputo (versión de CUDA soportada) por chip y modelo de tarjeta

### 3. Un primer programa en CUDA

Antes de comenzar a codificar el primer programa en CUDA, y si usted ha seguido las especificaciones del apéndice, comprobará que efectivamente el compilador ejecuta correctamente el código C++. Para ello intente escribir en CodeLite o su IDE preferido el siguiente fragmento en C:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hola mundo\n");
    return 0;
}
```

La simplicidad de este ejemplo estriba en el hecho de que sólo genera código en el lado del *host*<sup>2</sup>.

Pasemos ahora a realizar la primera llamada al *kernel*. Definiremos un *kernel* como el procedimiento en C, no recursivo y no anidado que implementa el código paralelizado en CUDA. Para ello será necesario identificar el procedimiento con la palabra clave `__global__`

Dicho de otra forma, el procedimiento que implementa el *kernel* no puede devolver

<sup>2</sup> Llamaremos a la CPU y su memoria como *host* y a la GPU y su memoria como dispositivo

ningún valor y debe estar antecedido por el calificador `__global__`

De esta forma quedaría una estructura de llamada al *kernel* como en el siguiente ejemplo:

```
#include <stdio.h>

__global__ void kernel(void){
}

__global__ void foo(void){
}

int main(int argc, char **argv){
    kernel <<<1,1>>>();
    foo <<<1,1>>>();
    printf("Hola mundo\n");
    return 0;
}
```

Observe que en primer lugar se hace una llamada al *kernel* `kernel` y posteriormente se llama al *kernel* `foo`. Obviamente estas llamadas al estar vacías no realizarán ninguna operación en la GPU, simplemente volverán sin realizar nada.

La palabra reservada `__global__` indica al compilador `nvcc` que genere código para la GPU en vez de la CPU.

Otra consideración a tener en cuenta son los ángulos para indicar parámetros que serán pasados al sistema en tiempo de ejecución. Estos parámetros no indican argumentos que serán pasados al cuerpo del procedimiento implementado en el *kernel*, sino que indicarán al sistema cómo lanzar el código que se ejecutará en el dispositivo. Es posible pasar parámetros al *kernel* especificándolos con paréntesis, aunque esto se verá más adelante.

Quizá sea útil para el lector realizar una llamada a una función desde el *kernel*. Para este propósito existe la palabra reservada `__device__` que indicará al compilador de CUDA que genere código exclusivamente como una función para ser llamada desde el *kernel* o desde otra función. De esta manera tendríamos el siguiente ejemplo:

```
#include <stdio.h>

__device__ float fx(float x, float y){
    return x + y;
}

__global__ void kernel(void){
    funcion(1.0, 2.0);
}

int main(int argc, char **argv){
    kernel <<<1,1>>>();
    printf("Llamada a función desde kernel\n");
    return 0;
}
```

Donde la función `fx` realiza la suma de dos parámetros de tipo `float`.

## 4. Reservando memoria

Otro aspecto importante de CUDA es la reserva y la liberación de memoria. Estos son aspectos cruciales en el momento de diseñar una aplicación.

Normalmente se reservará memoria para realizar alguna operación en el dispositivo, tales como devolver valores al *host* o pasar grandes estructuras de datos, tales como tablas o matrices al dispositivo. Para llevar a cabo esto recurriremos a la función reservada `cudaMalloc()`, similar al `malloc()` de C y que reserva memoria en una zona de especial llamada *Heap*.

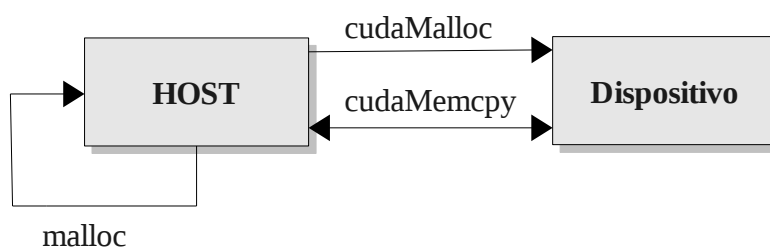
El prototipo de las funciones que veremos a continuación son las siguientes:

```
cudaError_t cudaMalloc (void** devPtr, size_t size)
```

Donde el primer parámetro es un puntero a puntero donde queremos almacenar los datos reservados en la memoria de la GPU, y el segundo es el tamaño de la reserva en bytes.

```
cudaError_t cudaMemcpy (void* dst, const void *src, size_t count,  
enum cudaMemcpyKind kind)
```

El primer parámetro es un puntero a la zona de memoria de destino de la copia; el segundo la zona fuente de copia; el tercero, la cuenta en bytes de la copia, y por último, el cuarto, que especifica si el movimiento de bytes es desde el *host* al dispositivo o al contrario.



**Figura 3.3** Orden de acción de las primitivas de reserva de memoria y copia

Para comprender los conceptos sobre reserva de memoria, considere el siguiente fragmento de código:

```
#include <stdio.h>
#include <iostream>

// Incluye utilidades de CUDA
#include <cutil.h>
```

```
__global__ void buscaCadena(char *cadena, int longitud, char caracter,
bool *encontrado){
    int i = 0;
    while ((cadena[i] != caracter) && i < longitud)
        i++;

    *encontrado = (i != longitud) ? true : false;
}

int main(int argc, char **argv){

    char HOSTcad[34] = {"Volverán las oscuras golondrinas"};
    bool HOSTencontrado;
    char *GPUcad;
    bool *GPUencontrado;

    CUDA_SAFE_CALL(cudaMalloc((void**)&GPUencontrado, sizeof(bool)));
    CUDA_SAFE_CALL(cudaMalloc((void**)&GPUcad, sizeof(char) * 32));
    CUDA_SAFE_CALL(cudaMemcpy(GPUcad,
                              HOSTcad, sizeof(char) * 32,
                              cudaMemcpyHostToDevice));

    buscaCadena<<<1,1>>>(GPUcad, 32, 'i', GPUencontrado);

    CUDA_SAFE_CALL(cudaMemcpy(&HOSTencontrado,
                              GPUencontrado, sizeof(bool),
                              cudaMemcpyDeviceToHost));

    std::cout << "Carácter encontrado: " << ((HOSTencontrado) ? "si" :
    "no") << std::endl;

    cudaFree(GPUencontrado);
    cudaFree(GPUcad);

    return 0;
}
```

Éste es un pequeño programa de ejemplo que realiza la búsqueda de un carácter en una cadena. Aunque la dimensión del problema es nimia para las capacidades de la GPU, se supone que ilustra adecuadamente los propósitos del ejemplo.

Como puede observar el lector, las reservas de memoria en el lado del host se realizan de la misma forma que la declaración de variables en programación imperativa convencional de C. Sin embargo, para las variables como GPUencontrado, que se representan como un puntero, es necesario reservar memoria en la zona del dispositivo mediante la primitiva `cudaMalloc` que adquirirá tantos bytes en la memoria de la GPU como le indiquemos en el segundo parámetro. Note que todas las variables que son de intercomunicación entre el host y el dispositivo deben ser reservadas de esta manera.

Una vez adquirida la memoria en la zona de la GPU, se precisa gestionar las transacciones, ya sea entre el host y el dispositivo o a la inversa. Para llevar a cabo este cometido tendremos que utilizar la primitiva `cudaMemcpy` proporcionada por el compilador y cuyo cometido es realizar la copia de bytes entre el host y el dispositivo. Tal es el caso de la primera llamada del programa de ejemplo, donde se realiza una copia entre los

datos de la cadena de texto creada en el host (`HOSTcad`) y la zona de memoria con formato de cadena en la GPU (`GPUcad`).

El detalle más característico de esta llamada es el cuarto parámetro. Aquí se indica el sentido de la copia, denotado como `cudaMemcpyToDevice` para indicar al host que debe transferir la información a la memoria de la tarjeta de vídeo.

Una vez preparados los datos, ya es posible llamar al *kernel* con los argumentos definidos anteriormente. En el caso de que la función del *kernel* encuentre el carácter buscado, devolverá el valor `true` en la variable reservada previamente en la GPU para almacenar el resultado. En resumen, una vez finalizado el procesamiento en el dispositivo, se procede a copiar la variable `GPUencontrado` en `HOSTencontrado` mediante la llamada `cudaMemcpy` con el cuarto parámetro igual a `cudaMemcpyDeviceToHost` que indicará que la transferencia del resultado se realizará desde la GPU al host.

Si la compilación y ejecución han ido correctamente el programa deberá visualizar:

```
Carácter encontrado: si
```

Por último, falta aún la parte más importante de la gestión de la memoria, que es su liberación y devolución al sistema. De acuerdo a un orden de correcta gestión de memoria, se precisará de la primitiva del compilador `cudaFree`. Esta función libera la memoria previamente reservada con la llamada a `cudaMalloc`. Fíjese el lector que la política seguida por el compilador de CUDA es idéntica que la utilizada en el estándar C.

A lo largo del programa se ha utilizado la llamada a la macro `CUDA_SAFE_CALL`. Esta macro proporciona información y parada de la ejecución cuando se detecta un valor erróneo en la llamada o el retorno de cualquier función CUDA.

Hemos visto un ejemplo clave que ilustra el manejo de punteros y memoria, pero desafortunadamente, el compilador no puede proteger al programador de un error de gestión de los punteros en tiempo real. Por ello, se resumen aquí los posibles casos y restricciones en el uso de punteros en CUDA:

- Se pueden pasar punteros reservados con `cudaMalloc` a funciones que se ejecuten en el dispositivo
- Se pueden usar punteros reservados con `cudaMalloc` para leer o escribir en memoria desde el código que se ejecuta en la GPU
- Se pueden pasar punteros reservados con `cudaMalloc` a funciones que se ejecutan en el lado del host
- No se pueden usar punteros reservados con `cudaMalloc` para leer o escribir en memoria desde el código que se ejecuta en el host

**Importante:** Mover datos desde memoria del *host* a memoria de la GPU consume tiempo. La causa es cuello de botella del ancho de banda del bus PCI express.



# 4

## Programación paralela

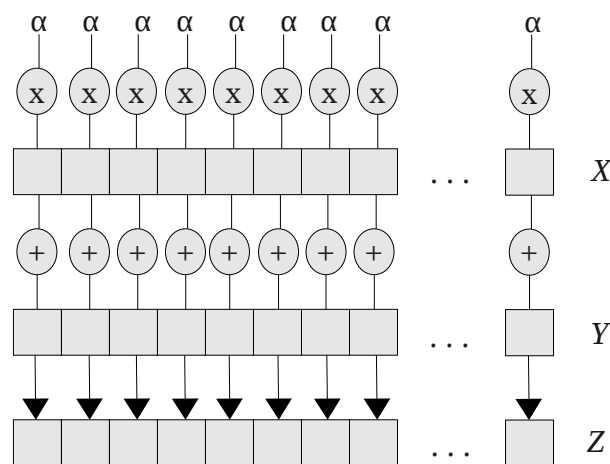
### 1. Un sencillo ejemplo

Comenzaremos esta introducción a la programación paralela exponiendo un breve ejemplo de programa para calcular el algoritmo SAXPY (Single-precision real Alpha X Plus Y), muy utilizado en álgebra lineal, cuya fórmula viene dada como:

$$y = \alpha x + y$$

Imagine dos secuencias de números enteros,  $x$  e  $y$ , almacenadas en dos arrays. La idea que pretendemos es multiplicar el primer vector de enteros por un escalar  $\alpha$  y sumarle los componentes del segundo vector  $y$ , finalmente el resultado se almacenará en el array  $z$ .

Gráficamente nuestro propósito es el siguiente:



**Figura 4.1** Ejemplo del algoritmo SAXPY

Antes de desarrollar el ejemplo en el respectivo código paralelo, considere el lector el siguiente ejemplo de algoritmo en forma secuencial para SAXPY.

```
#include <stdio.h>
#include <iostream>
#include <cutil.h>

#define N 10

void saxpySerie(int *x, int *y, int alfa, int* z)
{
    for (int i=0; i<N; i++)
        z[i] = alfa * x[i] + y[i];
}

int main(int argc, char **argv)
{
    const int alfa = 2;

    int CPUx[N], CPUy[N], z[N];

    srand (time(NULL));

    for (int i=0; i<N; i++)
    {
        CPUx[i] = rand()%10;
        CPUy[i] = rand()%10;
    }

    saxpySerie(CPUx, CPUy, alfa, z);

    std::cout << "Resultado procesamiento serie:" << std::endl;

    for (int i=0; i<N; i++)
        std::cout << alfa << " * " <<
            CPUx[i] << " + " << CPUy[i] <<
            " = " << z[i] << std::endl;

    return 0;
}
```

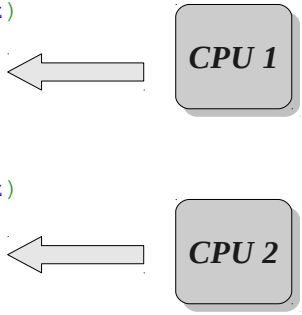
Obviamente este código es únicamente llevado a cabo por una sola CPU en forma secuencial. Si se fija en el bucle de `saxpySerie` observará que la misma CPU itera N veces sobre la misma operación en los arrays. Esto quedaría ilustrado de forma desenrollada como:

$$\begin{aligned} z[0] &= \text{alfa} * x[0] + y[0] \\ z[1] &= \text{alfa} * x[1] + y[1] \\ z[2] &= \text{alfa} * x[2] + y[2] \\ z[3] &= \text{alfa} * x[3] + y[3] \\ z[4] &= \text{alfa} * x[4] + y[4] \\ z[5] &= \text{alfa} * x[5] + y[5] \\ z[6] &= \text{alfa} * x[6] + y[6] \\ z[7] &= \text{alfa} * x[7] + y[7] \\ z[8] &= \text{alfa} * x[8] + y[8] \\ z[9] &= \text{alfa} * x[9] + y[9] \end{aligned}$$

Fíjese cómo el bucle se repite idénticamente para la misma operación, tan sólo cambia el índice del array. Aprovechando este modelo se podría hacer una adaptación a la versión con varios microprocesadores. Así, de ésta forma, si tuviéramos dos CPU, el código anterior quedaría como:

```
void saxpyCPU1(int *x, int *y, int alfa, int* z)
{
    for (int i=0; i<N; i+=2)
        z[i] = alfa * x[i] + y[i];
}

void saxpyCPU2(int *x, int *y, int alfa, int* z)
{
    for (int i=1; i<N; i+=2)
        z[i] = alfa * x[i] + y[i];
}
```



Obviamente, esta aproximación no es óptima, pues replicar el código para cada CPU requeriría una gran infraestructura y una razonable cantidad de espacio en memoria.

Consecuentemente, esta aproximación no es del todo acertada.

## 2. Algoritmo SAXPY en GPU

Realicemos ahora la misma operación pero de una manera más eficiente y que aproveche los recursos multinúcleo de la GPU CUDA. El código que veremos a continuación es muy similar al anterior en ciertos aspectos, aunque tanto el *kernel* como la función `main()` le pueden ser familiares si entendió correctamente el capítulo 3.

Empecemos exponiendo el programa principal:

```
#include <stdio.h>
#include <iostream>
#include <cutil.h>

#define N 10

int main(int argc, char **argv)
{
    const int alfa = 2;

    int CPUx[N], CPUy[N], z[N];

    int *GPUx, *GPUy;

    srand (time(NULL));

    for (int i=0; i<N; i++)
    {
        CPUx[i] = rand()%10;
        CPUy[i] = rand()%10;
    }
}
```

```
CUDA_SAFE_CALL(cudaMalloc(&GPUx, sizeof(int) * N));
CUDA_SAFE_CALL(cudaMalloc(&GPUy, sizeof(int) * N));

CUDA_SAFE_CALL(cudaMemcpy(GPUx, CPUx,
    sizeof(int) * N, cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy(GPUy, CPUy,
    sizeof(int) * N, cudaMemcpyHostToDevice));

saxpyParalelo<<<1,N>>>(GPUx, GPUy, alfa);

CUDA_SAFE_CALL(cudaMemcpy(z, GPUy,
    sizeof(int) * N, cudaMemcpyDeviceToHost));

std::cout << "Resultado procesamiento paralelo:" << std::endl;

for (int i=0; i<N; i++)
    std::cout << alfa <<
        " * " << CPUx[i] << " + " <<
        CPUy[i] << " = " << z[i] << std::endl;

cudaFree(GPUx);
cudaFree(GPUy);
return 0;
}
```

Al inicio del programa se reservan tres arrays en la memoria del host, que son CPUx, CPUy y z. El primer array reserva memoria para albergar los componentes del vector x, el segundo almacena el vector y y por el último reserva memoria para devolver los resultados. A continuación creamos dos punteros: \*GPUx y \*GPUy donde se tendrá una referencia a la memoria creada en el dispositivo. Habrá notado el lector la llamada a la función:

```
srand(time(NULL))
```

cuyo cometido es generar un nuevo valor de semilla para la generación de números aleatorios. En este caso generaremos valores al azar entre 0 y N – 1; aunque para simplificar el ejemplo se ha preferido optar por un número pequeño, en este caso el valor 10.

Después de proceder a la inicialización de los valores x e y llegamos a la parte de reserva de memoria en el lado de la GPU. En este caso, se llama a la función `cudaMalloc` con los punteros GPUx y GPUy que serán los responsables de mantener referencias a la memoria adquirida en el dispositivo. Como queremos reservar N veces tantos bytes como ocupe un entero, debemos multiplicar N por el tamaño en bytes devuelto por la función `C sizeof()`. Posteriormente procederemos a volcar la memoria con los componentes aleatorios de los vectores x e y del lado del host a la zona del dispositivo. Para ello haremos una llamada a `cudaMemcpy` pasando un puntero al array en el dispositivo como destino de copia y otro puntero al array en el host como fuente de inicio de la copia. Por último, se indica que el sentido de la copia: `cudaMemcpyHostToDevice`.

Ahora llegamos a la parte clave del programa: la llamada al *kernel*.

Todas las llamadas al *kernel* tiene el siguiente patrón:

**función-kernel<<<bloques, threads>>>(parámetros)**

Ahora introduciremos dos nuevas palabras a la terminología CUDA: el *thread* y los *bloques*. La división de la lógica del paralelismo en CUDA implica el concepto de *thread* que es parecida a la idea de proceso ligero creado por el sistema operativo y puesto a disposición del programador de sistemas. El concepto de *thread* aquí descrito se refiere a la unidad básica de procesamiento paralelo. Estos threads son agrupados en bloques que son procesados por los núcleos. Por lo tanto podemos lanzar un *kernel* con hasta 65.535 bloques como máximo por dimensión, donde el máximo de threads ejecutándose en cada bloque es de 512.

Por consiguiente, en la llamada:

```
saxpyParalelo<<<1,N>>>(GPUx,GPUy, alfa);
```

Se lanzarán un total de 10 threads en paralelo. La parte de implementación del *kernel* es la siguiente:

```
__global__ void saxpyParalelo(int *x, int *y, int alfa)
{
    int i = threadIdx.x;

    if (i < N)
        y[i] = alfa * x[i] + y[i];
}
```

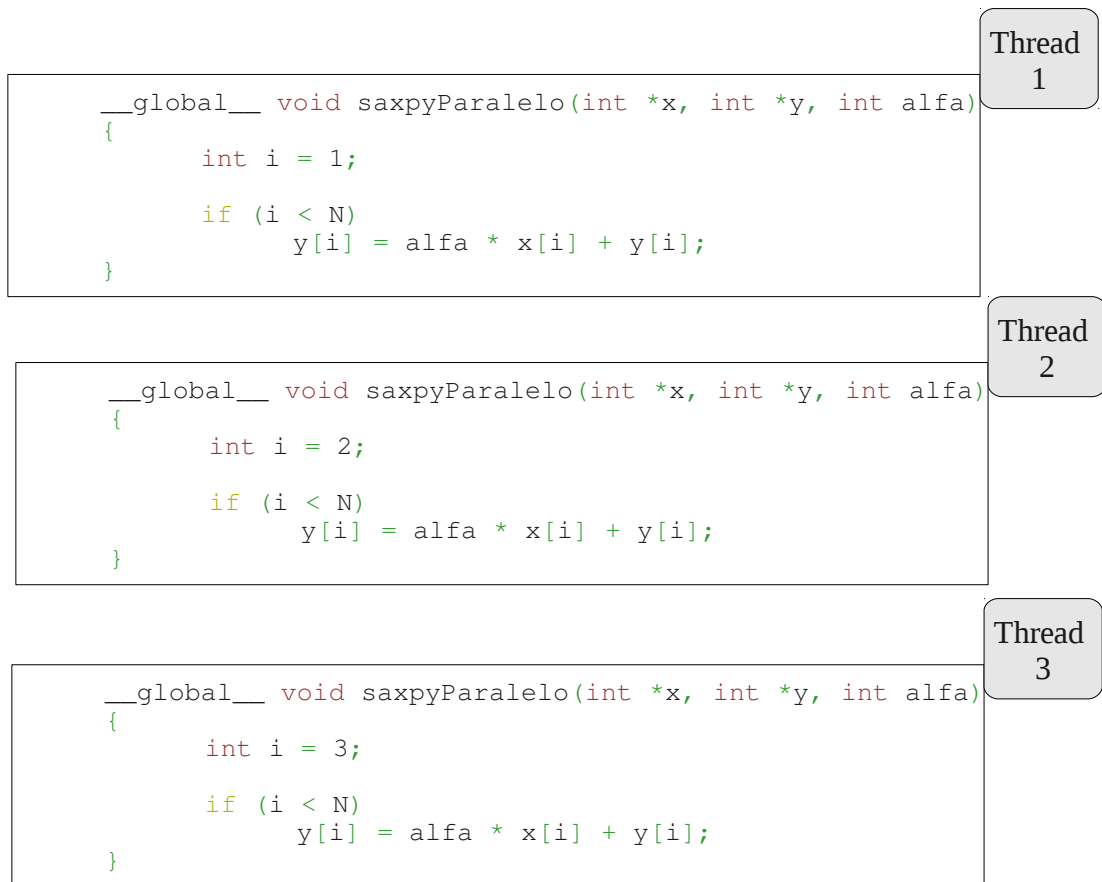
Habrás observado una nueva palabra reservada dentro del lenguaje CUDA. Ésta es ***threadIdx.x***, cuya función es devolver el identificador del thread paralelo dentro del bloque en cuestión, donde el identificador puede variar entre 0 y 9 en este ejemplo.

Pero considere el lector que N es igual a 4 esta vez. Tendríamos la siguiente ejecución en paralelo:

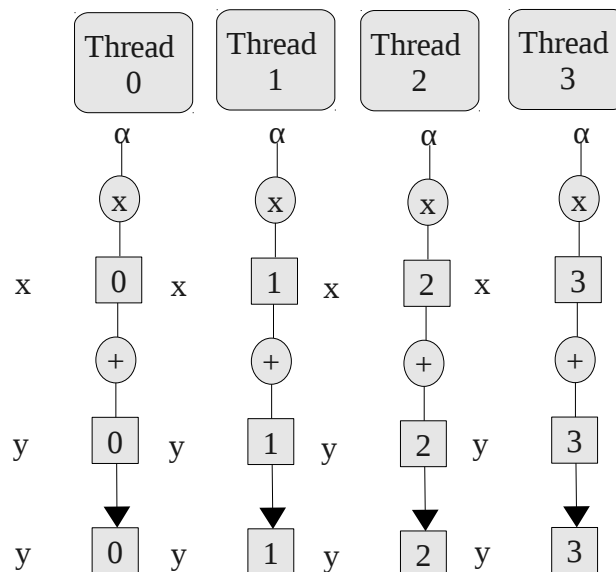
```
__global__ void saxpyParalelo(int *x, int *y, int alfa)
{
    int i = 0;

    if (i < N)
        y[i] = alfa * x[i] + y[i];
}
```

Thread  
0



Estos cuatro threads se ejecutarán en paralelo y accederán a una parte distinta de los datos, tal como se muestra en la figura 4.2.



**Figura 4.2** Ejecución paralela de threads

Aunque también sería posible realizar la misma operación de la siguiente manera:

```
__global__ void saxpyParalelo(int *x, int *y, int alfa)
{
    int i = blockIdx.x;

    if (i < N)
        y[i] = alfa * x[i] + y[i];
}
```

Donde `blockIdx.x` devolverá el identificador del bloque lanzado en paralelo, de la misma forma con que lanzamos anteriormente los threads. Por tanto la llamada al *kernel* se realizará como sigue:

```
saxpyParalelo<<<N,1>>>(GPUx, GPUy, alfa);
```

Con el primer parámetro entre ángulos indicando el número de bloques a lanzar en paralelo.

Cabe preguntarse por qué utilizamos la salvaguarda `if (i < N)` dentro dentro del cuerpo del procedimiento del *kernel*. La respuesta a esta pregunta es simple. Imagine que se hubiera realizado la siguiente llamada:

```
saxpyParalelo<<<100,1>>>(GPUx, GPUy, alfa);
```

Si no se hubiera utilizado la salvaguarda se hubiera accedido a posiciones ilegales de la memoria al lanzar el *kernel* y dejando posiblemente la computadora bloqueada.

Por último, introduciremos un nuevo vocablo a la terminología CUDA y también muy utilizado en computación paralela: el *grid*. Podemos denominar un *grid* como un grupo de bloques paralelos que realizan una tarea determinada. En este ejemplo que se ha mostrado el *grid* ha sido de una sola dimensión.

### 3. Suma de matrices

Ahora veremos otro ejemplo de utilización del *kernel* en dos dimensiones, en la que se utilizarán nuevas variables del lenguaje CUDA C para identificar las filas y las columnas tanto de bloques como de threads. Para ilustrar estas nuevas variables reservadas utilizaremos como ejemplo el algoritmo de la suma de matrices, en el aquí exponemos su expresión:

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} & x_{1m} \\ x_{21} & x_{22} & x_{23} & x_{2m} \\ x_{31} & x_{32} & x_{33} & x_{3m} \\ x_{n1} & x_{n2} & x_{n3} & x_{nm} \end{pmatrix} + \begin{pmatrix} y_{11} & y_{12} & y_{13} & y_{1m} \\ y_{21} & y_{22} & y_{23} & y_{2m} \\ y_{31} & y_{32} & y_{33} & y_{3m} \\ y_{n1} & y_{n2} & y_{n3} & y_{nm} \end{pmatrix} = \begin{pmatrix} x_{11}+y_{11} & x_{12}+y_{12} & x_{13}+y_{13} & x_{1m}+y_{1m} \\ x_{21}+y_{21} & x_{22}+y_{22} & x_{23}+y_{23} & x_{2m}+y_{2m} \\ x_{31}+y_{31} & x_{32}+y_{32} & x_{33}+y_{33} & x_{3m}+y_{3m} \\ x_{n1}+y_{n1} & x_{n2}+y_{n2} & x_{n3}+y_{n3} & x_{nm}+y_{nm} \end{pmatrix}$$

Esta estructura matemática es idónea para la programación paralela según la filosofía de CUDA según veremos a continuación.

Antes de explicar el código fuente veamos un nuevo tipo de datos:

```
dim3 Grid(DIM, DIM)
```

El tipo implícito de CUDA C `dim3` no es un tipo C estándar y permite crear una tupla tridimensional. Se estará preguntando porqué utilizaremos una estructura de tres dimensiones cuando una suma de matrices es siempre en dos dimensiones. Pues bien, la realidad es que aunque CUDA soporte el tipo interno `dim3` actualmente no está implementada la posibilidad de crear estructuras de tres dimensiones, por lo que el compilador de CUDA siempre esperará que una variable de este tipo sea siempre igual a 1 en su último parámetro, es decir, se considera que la última dimensión equivaldrá implícitamente a 1.

Considere el siguiente fragmento de código de la función principal:

```
#include <stdio.h>
#include <iostream>
#include <cutil.h>

#define N 10
#define M 10

#define Threads 4

int main(int argc, char **argv)
{
    int CPUx[N][M], CPUy[N][M], CPUz[N][M];
    int *GPUx, *GPUy, *GPUz;

    CUDA_SAFE_CALL(cudaMalloc((void**)&GPUx, N*M*sizeof(int)));
    CUDA_SAFE_CALL(cudaMalloc((void**)&GPUy, N*M*sizeof(int)));
    CUDA_SAFE_CALL(cudaMalloc((void**)&GPUz, N*M*sizeof(int)));
```



```
dim3 dimBloque(Threads, Threads);
dim3 dimGrid((int)ceil(float(N) / float(Threads)),
              (int) ceil(float(M) / float(Threads)));
```

Quizá le resultará algo familiar las primeras líneas del programa. Su intención es declarar tres arrays bidimensionales en el lado del host para almacenar las matrices de origen y destino. Así mismo, se declaran tres variables puntero donde se reservará la memoria para las matrices en el dispositivo.

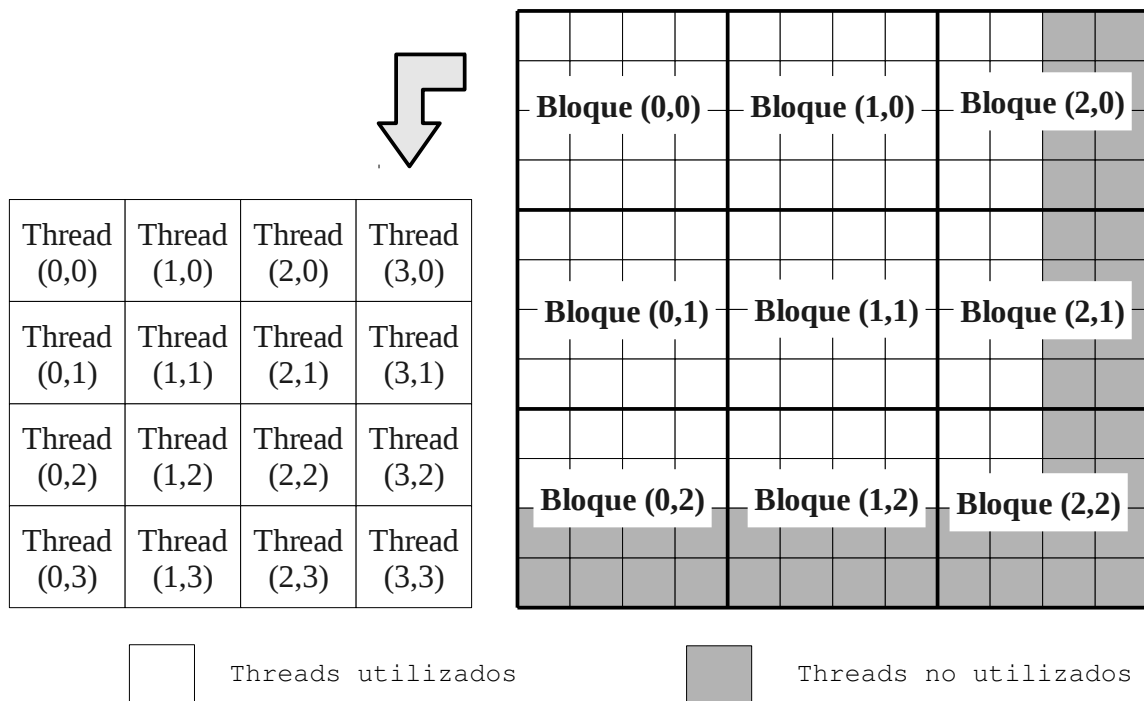
Ahora fíjese en la siguiente declararción:

```
dim3 dimBloque(Threads, Threads);
```

Aquí estamos indicando al compilador de CUDA C que cree un bloque de 4 x 4 threads inscritos dentro de los bloques del *grid* definido por las siguientes expresión:

$$bloquesX = \left\lceil \frac{\text{columnas matriz}}{\text{número de threads por bloque}} \right\rceil$$

$$bloquesY = \left\lceil \frac{\text{filas matriz}}{\text{número de threads por bloque}} \right\rceil$$



**Figura 4.3** Estructura del grid

en la llamada a:

```
dim3 dimGrid((int)ceil(float(N) / float(Threads)),
             (int)ceil(float(M) / float(Threads)));
```

En el siguiente fragmento de código realizamos la inicialización de los arrays

```
srand (time(NULL));

for (int i=0; i<N; i++)
    for(int j=0; j<M; j++)
    {
        CPUx[i][j] = rand()%10;
        CPUy[i][j] = rand()%10;
    }

CUDA_SAFE_CALL(cudaMemcpy(GPUx, CPUx, N*M*sizeof(int),
                          cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy(GPUy, CPUy, N*M*sizeof(int),
                          cudaMemcpyHostToDevice));
```

con valores en el intervalo 0-9 y copiamos las matrices origen a las direcciones de memoria apuntadas por los punteros reservados en el lado del dispositivo.

Previo a realizar la llamada al *kernel* considere la declaración del mismo:

```
__global__ void sumaMatrices(int *x, int *y, int *z)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    int indice = i + j * N;
    if (i < N && j < M)
        z[indice] = x[indice] + y[indice];
}
```

Nos encontramos aquí con una nueva variable de CUDA implícita que nos indica la dimensión del bloque. Esta variable son `blockDim.x` y `blockDim.y` para las columnas y las filas de la matriz respectivamente. Una vez localizadas las posiciones absolutas a partir de las variables de posicionamiento relativo, procedemos a obtener el valor del desplazamiento en el array de origen y destino por medio de la fórmula:

$$\text{desplazamiento} = x + y * \text{número de columnas matriz}$$

Y utilizamos una salvaguarda para evitar que los threads no utilizados, es decir, los threads sombreados en la figura 4.3 violen posiciones de memoria del dispositivo no permitidas.

Ya sólo queda llamar al *kernel* con las dos variables declaradas con el nuevo tipo de datos antes explicado:

```
sumaMatrices<<<dimGrid, dimBloque>>>(GPUx, GPUy, GPUz);

CUDA_SAFE_CALL(cudaMemcpy(CPUz, GPUz, N*M*sizeof(int),
                          cudaMemcpyDeviceToHost));

for (int i=0; i<N; i++)
{
    for(int j=0; j<M; j++)
        std::cout <<" GPU: " << CPUx[i][j] << " + " << CPUy[i]
        [j] << " = " << CPUz[i][j] <<
        " CPU: " << CPUx[i][j] << " + " << CPUy[i][j] << " = "
        << CPUx[i][j] + CPUy[i][j] << std::endl;

    std::cout << "----- Fila: " << i << std::endl;
}

cudaFree(GPUx);
cudaFree(GPUy);
cudaFree(GPUz);

return 0;
}
```

Como podrá apreciar, se llamará al *kernel* indicando al sistema las dimensiones del *grid* y de los bloques. Al finalizar el procedimiento paralelo en la GPU, copiamos el resultado devuelto en el puntero GPUz a la variable bidimensional CPUz y procedemos a imprimir las matrices con los resultados, calculados tanto por medio de la GPU como de la CPU. Por último, no debemos olvidar liberar la memoria reservada en el dispositivo.

## 4. Sincronización de threads y memoria compartida

Entraremos ahora a explicar una nueva forma de programación paralela en CUDA que sigue el paradigma algorítmico “*Divide y Vencerás*” por medio del uso de un nuevo tipo de memoria que incrementa la eficiencia y el rendimiento. Este nuevo tipo de memoria lo llamaremos *memoria compartida*<sup>1</sup> que junto a las palabras reservadas de CUDA C `__device__` y `__global__` denotará un nuevo tipo de palabra clave que definiremos como `__shared__` en inglés. Este tipo de variable crea una copia de la variable por cada bloque que se lanza en la GPU. Cada thread en ese bloque comparte dicha memoria, pero los threads de cada bloque no pueden acceder o modificar la copia de la variable compartida declarada en otros bloques. La memoria compartida reside físicamente en la GPU en contraposición a la memoria DRAM de la tarjeta. En consecuencia, la latencia de acceso a memoria compartida tiende a ser más baja que a los típicos búferes. Debido a este nuevo modelo de memoria introducido en la programación CUDA, será necesario nuevas técnicas y mecanismos para la sincronización entre threads.

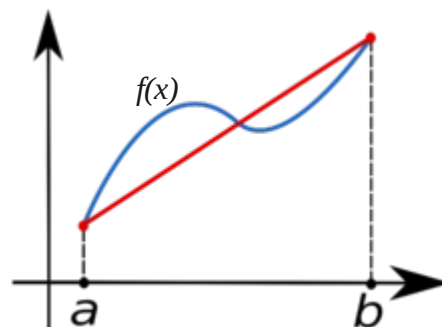
Veamos un ejemplo sencillo que ilustra este caso:

### 4.1 Método de integración del trapecio

La regla del trapecio es un método de integración numérica muy utilizada en matemáticas para calcular aproximadamente el valor de la integral definida

$$\int_a^b f(x) dx$$

La idea consiste en aproximar el valor de la integral de  $f(x)$  por el de la función lineal que pasa a través de los puntos  $(a, f(a))$  y  $(b, f(b))$ . Por lo tanto la integral de la función  $f(x)$  es igual al área del trapecio bajo la gráfica de la función lineal:



**Figura 4.4** La función curvilínea  $f(x)$  es aproximada por la función lineal

<sup>1</sup> Shared memory en terminología CUDA

y para calcularla se sigue la siguiente fórmula:

$$\int_a^b f(x) dx \approx (b-a) \frac{f(a) + f(b)}{2}$$

donde el error se define como:

$$-\frac{(b-a)^3}{12} f^{(2)}(\xi)$$

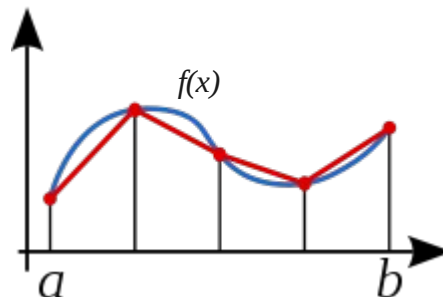
Siendo  $\xi$  un número comprendido en el intervalo  $[a,b]$  y  $f^{(2)}$  la derivada segunda de  $f(x)$ .

## 4.2 Regla del trapecio compuesta

Basándonos en la formula anterior aplicaremos una nueva técnica que consiste en aproximar una integral definida utilizando  $n$  veces la regla del trapecios. En esta formulación se supone que  $f(x)$  es continua y positiva en el intervalo  $[a,b]$ . De tal modo que la integral definida

$$\int_a^b f(x) dx$$

representa el área de la región delimitada por la gráfica de  $f(x)$  y el eje  $x$ , desde  $x=a$  hasta  $x=b$ . Primero se divide el intervalo  $[a,b]$  en  $n$  subintervalos, cada uno de ancho  $\Delta x = (b-a)/n$ , como puede verse en la figura 4.5.



**Figura 4.5** Integración aproximada por subintervalos equidistantes

Después de realizar todo el proceso matemático se llega a la siguiente fórmula:

$$\int_a^b f(x) dx \approx \frac{h}{2} [f(a) + 2f(a+h) + 2f(a+2h) + \dots + f(b)]$$

Donde  $h = \frac{(b-a)}{n}$  y  $n$  es el número de divisiones.

Finalmente la expresión se puede simplificar como:

$$\int_a^b f(x) dx \approx \frac{(b-a)}{n} \left( \frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k \frac{b-a}{n}\right) \right)$$

### 4.3 Implementación en CUDA con memoria compartida

Veamos la forma de implementar la integral por la regla del trapecio utilizando la memoria compartida (*shared memory*) y la sincronización de threads. Considere el siguiente fragmento del código del *kernel*:

```
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <cutil.h>

// Número de intervalos
const int N = 80000;
const int numThreads = 256;
const int bloquesPorGrid = min(32, (N + numThreads - 1) / numThreads);

__device__ float funcion_gpu(float x)
{
    return sin(1/x)*pow(x,3)/(x + 1)*(x+2);
}

float funcion_cpu(float x)
{
    return sin(1/x)*pow(x,3)/(x+1)*(x+2);
}

__global__ void trapecios(float a, float b, float h, float* resultado)
{
    __shared__ float parcial[numThreads];

    int iteracion = threadIdx.x + blockIdx.x* blockDim.x;

    float temp = 0;

    while ( iteracion < N )
    {
        if (iteracion != 0)
            temp += funcion_gpu(a + h * iteracion);
    }
}
```

```

        iteracion += blockDim.x * gridDim.x;
    }

    parcial[threadIdx.x] = temp; // Almacena los resultados parciales

```

Trataremos de aproximar:

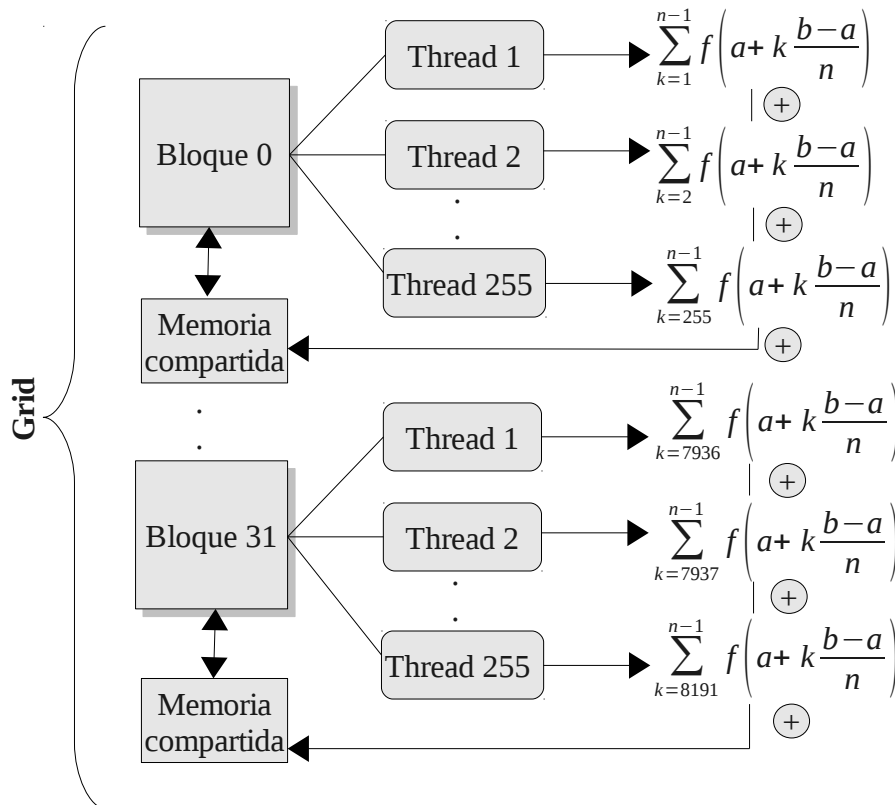
$$\int_{1.5}^{2.78} f(x) dx \quad f(x) = \frac{\sin\left(\frac{1}{x}\right) * x^3 * (x+2)}{(x+1)}$$

Como podrá observar el lector utilizaremos una variable de memoria compartida para cada bloque con un tamaño de 256 threads<sup>2</sup> donde almacenaremos las sumas parciales de

```
__shared__ float parcial[numThreads];
```

$$\sum_{k=1}^{n-1} f\left(a + k \frac{b-a}{n}\right)$$

Donde cada thread procesa un intervalo del algoritmo de trapezios iterando hasta N.



**Figura 4.6** Suma de los subintervalos

<sup>2</sup> Recuerde que el máximo de bloques que se pueden lanzar en cada dimensión del grid es de 65535 y el máximo de threads por bloque es de 512

donde `k` representa la variable `iteración`.

así tendremos un total de 8191 threads ejecutándose simultáneamente para calcular las sumas parciales e iterando con el salto definido por:

```
iteracion += blockDim.x * gridDim.x;
```

Aquí se nos presenta una nueva variable implícita definida por el compilador CUDA C: `gridDim` que con sus propiedades `gridDim.x` y `gridDim.y` nos proporciona la dimensión del grid en bloques, que en este caso es 32, puesto que únicamente utilizamos la dimensión `x`. De esta forma, el salto en el incremento anterior será de un total de 8192 iteraciones que sumarán todos los subintervalos.

Quizá se habrá fijado por qué no utilizamos la iteración 0. Esto lo haremos así por la sencilla razón de no tener que sumar un intervalo incorrecto al sumatorio.

Finalmente, al salir del bucle se utilizará la memoria compartida para almacenar cada resultado accediendo con el identificador del thread en cada bloque.

```
parcial[threadIdx.x] = temp;
```

Una vez que la primera fase ha calculado los subintervalos y almacenado las sumas parciales, procedemos a agrupar los resultados de cada array que representa la variable compartida. Pero, previamente a realizar esta operación tenemos que esperar a que todos los threads terminen la fase de cálculo de los subintervalos, puesto que esto es una operación potencialmente peligrosa. Necesitamos un método que garantice que todas las escrituras en el array `parcial[]` se completen antes de que cualquier otra operación de lectura/escritura se realice en este búfer. Por suerte, tenemos la siguiente primitiva de sincronización en CUDA C:

```
__syncthreads()
```

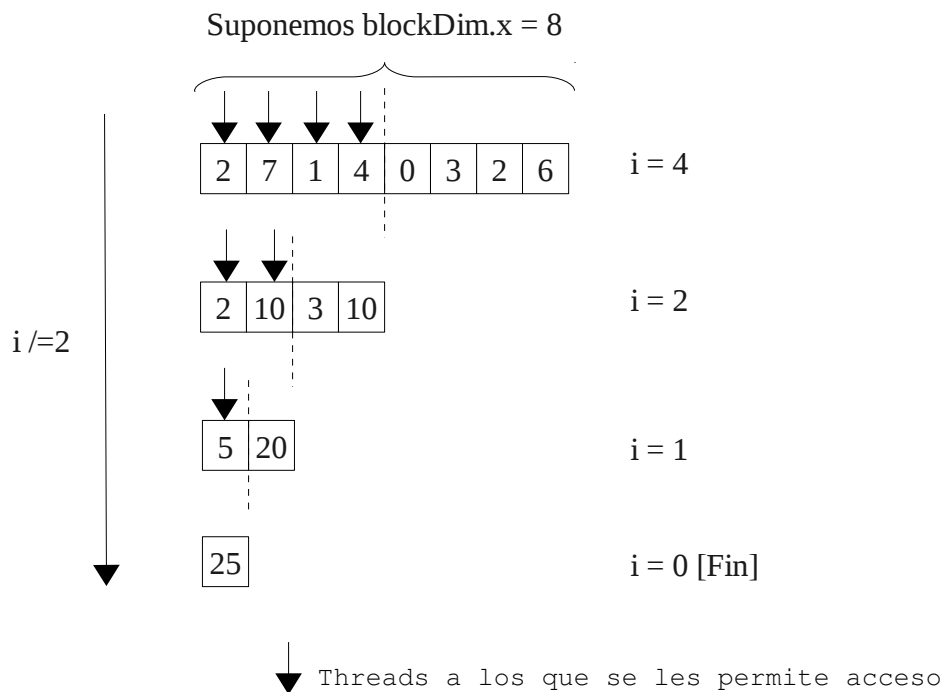
Con esta primitiva nos aseguramos que cuando el primer thread ejecute la primera intrucción después de `__syncthreads()`, todos los otros threads del bloque han llegado también a `__syncthreads()`.

Ya hemos conseguido que nuestro array se complete con los valores adecuados, ahora podemos proceder a sumarlos. Denominaremos esta simple operación de suma de totales de un array de entrada en un array más pequeño con los resultados finales como *reducción*. La reducción es una técnica muy utilizada a menudo en programación paralela.

La idea que aplicaremos consiste en particionar el array de memoria compartida en dos partes iguales para cada thread del bloque. Una vez que el thread entra en el bucle *while* procede de manera similar, dividiendo el array en dos partes iguales (siempre y cuando el valor de `i` permita acceder al thread actual) y sumando la posición que representa el thread en el array con su semejante en la otra parte. Como vemos existe un segundo `__syncthreads()` que permite que todos los threads anteriores accedan otra vez al array cuando hayan terminado su parte, sin infringir las lecturas/escrituras con otros threads posteriores. Este procedimiento se repite  $\log_2$  número de threads por bloque veces. La figura



4.7 ilustra este algoritmo de reducción general:



**Figura 4.7** Proceso de reducción

A continuación se expone el código que implementa el algoritmo de reducción:

```
int i = blockDim.x / 2;

while ( i!=0 ) // Fase de reducción
{
    if (threadIdx.x < i)
        parcial[threadIdx.x] += parcial[threadIdx.x + i];
    __syncthreads();
    i /= 2;
}
```

Por último se procede a devolver el valor del resultado de las sumas parciales del array `parcial[]` en memoria global:

```
if (threadIdx.x == 0)
    resultado[blockIdx.x] = parcial[0];
}
```

Se preguntará por qué el valor global se almacena para `threadIdx.x == 0`. Esto es así puesto que sólo hay un número que necesita escribir en memoria global, solamente un único thread necesita realizar dicha operación. En verdad, podría haber sido cualquier thread el que hubiera realizado este trabajo y el programa funcionaría de manera

exactamente igual, pero esto crearía una gran cantidad de tráfico en memoria innecesario para almacenar el valor.

Ahora que hemos visto cómo se lanza el kernel para el cálculo de la integral, podemos finalmente exponer el programa principal para el ejemplo de la regla del trapecio:

```
int main(int argc, char **argv)
{
    const float a = 1.5f;
    const float b = 2.78f;
    float h = (b - a) / N;

    float *resultado_cpu;
    float *resultado_gpu;

    resultado_cpu = (float*)malloc(bloquesPorGrid * sizeof(float));

    CUDA_SAFE_CALL(cudaMalloc(&resultado_gpu, bloquesPorGrid *
                             sizeof(float)));

    trapecios<<<bloquesPorGrid,numThreads>>>(a, b, h, resultado_gpu);
}
```

El código que se expone es fácilmente entendible. Para que la explicación no le parezca un aburrimiento, pasaremos a resumir brevemente su cometido:

- Establece los límites de integración inferior (1.5) y superior (2.78) como constantes.
- Reserva memoria en el lado del host para el resultado devuelto por el *kernel*.
- Reserva memoria en el lado del dispositivo para almacenar las sumas parciales devueltas por el *kernel*.
- Llama al kernel con un determinado número de threads y bloques para el *grid*.

En la sección 4.3 se vió la inicialización de las constantes en el encabezamiento del programa. Para su claridad volvemos mostrarlas:

```
// Número de intervalos
const int N = 80000;
const int numThreads = 256;
const int bloquesPorGrid = min(32, (N + numThreads - 1) / numThreads);
```

Primero establecemos el número de subintervalos a 80000, con un total de 256 threads por bloque. Quizá ahora se esté preguntando cómo calculamos el número de bloques por *grid* a partir de la formula expuesta, puesto que merece la pena examinar este valor. Vimos cómo para calcular la regla del trapecio se utiliza un proceso de reducción para

calcular las sumas parciales. El tamaño de esta lista de sumas parciales debería ser adecuada para la CPU y lo suficientemente grande de manera que tengamos suficientes bloques ejecutándose para mantener la GPU lo máximo ocupada. Para conseguir este equilibrio hemos seleccionado un tamaño de 32 bloques, aunque el rendimiento de esta cifra puede variar dependiendo de la velocidad de la CPU y la GPU.

En caso de que el valor de  $N$  sea muy pequeño y los 32 bloques con 256 threads sea demasiado grande para el cómputo, utilizaremos la fórmula matemática general que devuelve siempre el múltiplo más pequeño de `numThreads` que es más grande o igual a  $N$ :

$$\frac{(N + (\text{numThreads} - 1))}{\text{numThreads}}$$

Y por tanto el número de bloques que ejecutamos será el menor entre 32 o el valor de la fórmula anterior:

```
const int bloquesPorGrid = min(32, (N + numThreads - 1) / numThreads);
```

Después de llamar al kernel debemos de copiar los datos devueltos por el procesamiento de la GPU, mediante la llamada a `cudaMemcpy`:

```
CUDA_SAFE_CALL(cudaMemcpy(resultado_cpu, resultado_gpu, bloquesPorGrid *
                          sizeof(float), cudaMemcpyDeviceToHost));
```

Ya sólo falta sumar las sumas parciales devueltas en el array `resultado_cpu` teniendo claro de antemano que la longitud de los datos es suficientemente pequeña para la CPU y que no va a afectar al rendimiento:

```
float suma_parciales = (funcion_cpu(a) + funcion_cpu(b)) / 2.0f;

for (int i = 0; i < bloquesPorGrid; i++)
    suma_parciales += resultado_cpu[i];

suma_parciales *= h;

std::cout << "Resultado de integral con GPU: " << suma_parciales <<
std::endl;
```

Puesto que `resultado_cpu[]` contiene  $\sum_{k=1}^{n-1} f\left(a + k \frac{b-a}{n}\right)$  nos falta sumarle  $\frac{f(a)+f(b)}{2}$  y por último multiplicar el valor por  $h$ .

A continuación añadimos el fragmento de código que realiza la misma operación de trapecios en serie sobre la CPU:

```
suma_parciales = (funcion_cpu(a) + funcion_cpu(b)) / 2.0f;

for(int i=1; i<N; i++)
    suma_parciales += funcion_cpu(a + i*h);

suma_parciales *= h;

std::cout << "Resultado de integral con CPU: " << suma_parciales <<
std::endl;

cudaFree(resultado_gpu);

free(resultado_cpu);

return 0;
}
```

Como podrá claramente comprobar, el código para el lado del host es mucho más simple que su homólogo para la GPU. La diferencia estriba en el rendimiento, pues calcular los 80000 subintervalos con la CPU es mucho más lento que paralelizado en la GPU. En el capítulo 8 veremos más detalladamente los aspectos de medida del rendimiento en programación paralela versus serie.

Finalmente, y para mayor claridad, añadimos el código fuente completo para el algoritmo de cálculo de una integral por la regla del trapecio:

```
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <cutil.h>

// Número de intervalos
const int N = 80000;
const int numThreads = 256;
const int bloquesPorGrid = min(32, (N + numThreads - 1) / numThreads);

__device__ float funcion_gpu(float x)
{
    return sin(1/x)*pow(x,3)/(x + 1)*(x + 2);
}

float funcion_cpu(float x)
{
    return sin(1/x)*pow(x,3)/(x + 1)*(x + 2);
}

__global__ void trapecios(float a, float b, float h, float* resultado)
{
    __shared__ float parcial[numThreads];

    int iteracion = threadIdx.x + blockIdx.x* blockDim.x;

    float temp = 0;

    while ( iteracion < N )
    {
```

```
        if (iteracion != 0)
            temp += funcion_gpu(a + h * iteracion);

        iteracion += blockDim.x * gridDim.x;
    }

    parcial[threadIdx.x] = temp; // Almacena los resultados parciales
    __syncthreads(); // Sincroniza threads

    int i = blockDim.x / 2;

    while ( i!=0 ) // Fase de reducción
    {
        if (threadIdx.x < i)
            parcial[threadIdx.x] += parcial[threadIdx.x + i];
        __syncthreads();
        i /= 2;
    }

    if (threadIdx.x == 0)
        resultado[blockIdx.x] = parcial[0];
}

int main(int argc, char **argv)
{

    const float a = 1.5f;
    const float b = 2.78f;
    float h = (b - a) / N;

    float *resultado_cpu;
    float *resultado_gpu;

    resultado_cpu = (float*)malloc(bloquesPorGrid * sizeof(float));

    CUDA_SAFE_CALL(cudaMalloc(&resultado_gpu, bloquesPorGrid *
                               sizeof(float)));

    trapecios<<<bloquesPorGrid,numThreads>>>(a, b, h, resultado_gpu);

    CUDA_SAFE_CALL(cudaMemcpy(resultado_cpu, resultado_gpu,
                               bloquesPorGrid *sizeof(float), cudaMemcpyDeviceToHost));

    float suma_parciales = (funcion_cpu(a) + funcion_cpu(b)) / 2.0f;

    for (int i = 0; i < bloquesPorGrid; i++)
        suma_parciales += resultado_cpu[i];

    suma_parciales *= h;

    std::cout << "Resultado de integral con GPU: " << suma_parciales <<
    std::endl;

    suma_parciales = (funcion_cpu(a) + funcion_cpu(b)) / 2.0f;
```

```
for(int i=1; i<N; i++)
    suma_parciales += funcion_cpu(a + i*h);

suma_parciales *= h;

std::cout << "Resultado de integral con CPU: " << suma_parciales <<
std::endl;

cudaFree(resultado_gpu);

free(resultado_cpu);

return 0;
}
```