UNIVERSIDAD DE BURGOS

Área de Tecnología Electrónica

INTRODUCCIÓN A LA PROGRAMACIÓN EN CUDA

César Represa Pérez
José María Cámara Nebreda
Pedro Luis Sánchez Ortega

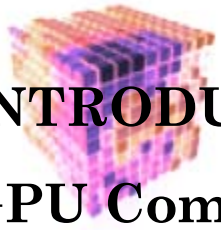
Introducción a la programación en CUDA©2016v3.1

Área de Tecnología Electrónica
Departamento de Ingeniería Electromecánica
Universidad de Burgos

ÍNDICE

En este manual se han utilizado las siguientes funciones de **CUDA v5.0**:

Función **Página**



INTRODUCCIÓN: GPU Computing

1. Computación en sistemas heterogéneos

Podemos definir la computación sobre tarjetas gráficas (en inglés *GPU computing*) como el uso de una tarjeta gráfica (**GPU** - *Graphics Processing Unit*) para realizar cálculos científicos de propósito general. El modelo de computación sobre tarjetas gráficas consiste en usar conjuntamente una **CPU** (*Central Processing Unit*) y una **GPU** de manera que formen un modelo de computación heterogéneo (Figura i.1). Siguiendo este modelo, la parte secuencial de una aplicación se ejecutaría sobre la **CPU** (comúnmente denominada ***host***) y la parte más costosa del cálculo se ejecutaría sobre la **GPU** (que se denomina ***device***). Desde el punto de vista del usuario, la aplicación simplemente se va a ejecutar más rápido porque está utilizando las altas prestaciones de la **GPU** para incrementar el rendimiento.

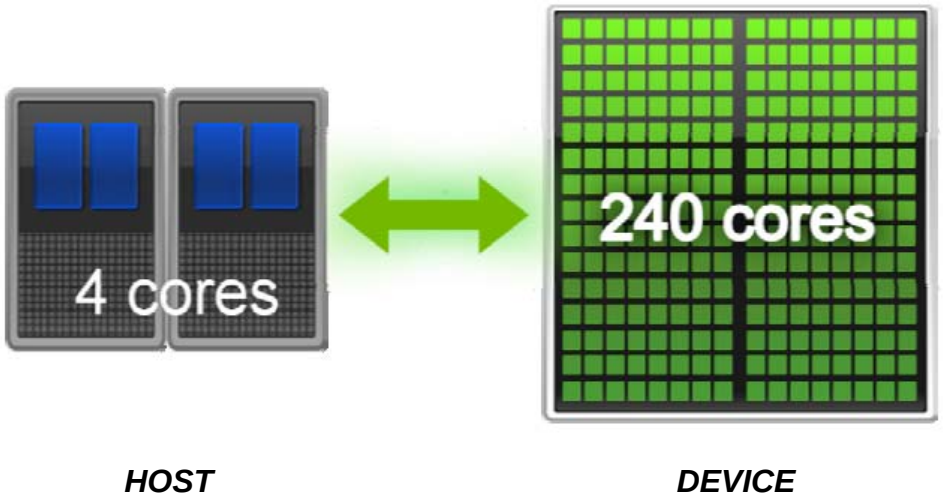


Figura i.1. Modelo de computación en sistemas heterogéneos: **CPU** + **GPU**.

El problema inicial del uso de las tarjetas gráficas para el cálculo científico de propósito general (en inglés **GPGPU** - *General-Purpose Computing on Graphics Processing Units*) era que se necesitaba usar lenguajes de programación específicos para gráficos como el OpenGL o el Cg para programar la **GPU**. Los desarrolladores debían hacer que sus aplicaciones científicas parecieran aplicaciones gráficas convirtiéndolas en problemas que dibujaban

triángulos y polígonos. Esto claramente limitaba el acceso por parte del mundo científico al enorme rendimiento de las **GPU**s.

NVIDIA fue consciente del potencial que suponía acercar este enorme rendimiento a la comunidad científica en general y decidió investigar la forma de modificar la arquitectura de sus **GPU**s para que fueran completamente programables para aplicaciones científicas además de añadir soporte para lenguajes de alto nivel como C y C++. De este modo, en Noviembre de 2009, NVIDIA introdujo para sus tarjetas gráficas la arquitectura **CUDA™** (*Compute Unified Device Architecture*), una nueva arquitectura para cálculo paralelo de propósito general, con un nuevo repertorio de instrucciones y un nuevo modelo de programación paralela, con soporte para lenguajes de alto nivel (Figura i.2), y constituidas por cientos de núcleos que pueden procesar de manera concurrente miles de hilos de ejecución. En esta arquitectura, cada núcleo tiene ciertos recursos compartidos, incluyendo registros y memoria. La memoria compartida integrada en el propio chip permite que las tareas que se están ejecutando en estos núcleos compartan datos sin tener que enviarlos a través del bus de memoria del sistema.

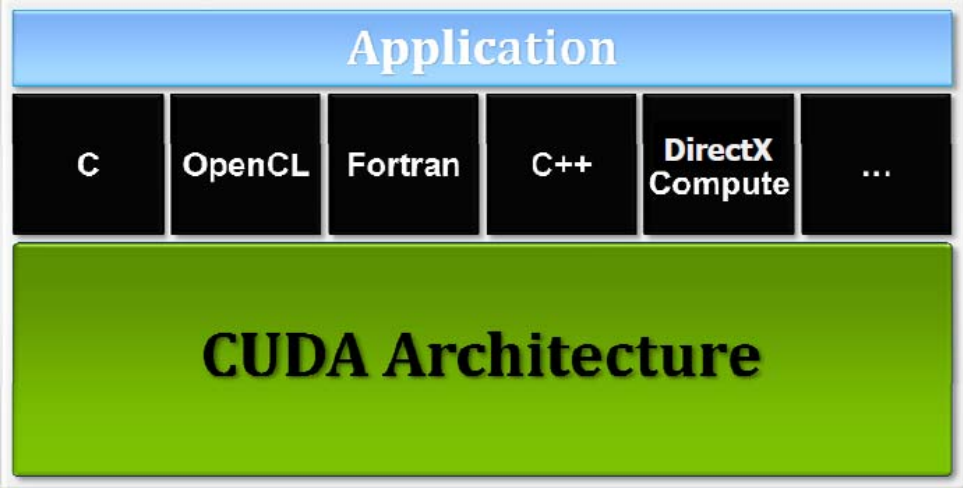


Figura i.2. CUDA está diseñado para soportar diferentes lenguajes de programación.

2. Arquitectura CUDA

En la arquitectura clásica de una tarjeta gráfica podemos encontrar la presencia de dos tipos de procesadores, los procesadores de vértices y los procesadores de fragmentos, dedicados a tareas distintas e independientes dentro del cauce gráfico, y con repertorios de instrucciones diferentes. Esto presenta dos problemas importantes, por un lado el desequilibrio de carga que aparece entre ambos procesadores y por otro la diferencia entre sus respectivos repertorios de instrucciones. De este modo, la evolución natural en la arquitectura

de una **GPU** ha sido la búsqueda de una arquitectura unificada donde no se distinguiera entre ambos tipos de procesadores. Así se llegó a la arquitectura **CUDA™**, donde todos los núcleos de ejecución necesitan el mismo repertorio de instrucciones y prácticamente los mismos recursos.

Figura i.3. Arquitectura de una tarjeta gráfica CUDA-enabled.

En la Figura i.3 se muestra la arquitectura de una tarjeta gráfica compatible con **CUDA**. En ella se puede observar la presencia de unas unidades de ejecución denominadas *Streaming Multiprocessors* (**SM**), 8 unidades en el ejemplo de la figura, que están interconectadas entre sí por una zona de memoria común. Cada **SM** está compuesto a su vez por unos núcleos de cómputo llamados "núcleos CUDA" o *Streaming Processors* (**SP**), que son los encargados de ejecutar las instrucciones y que en nuestro ejemplo vemos que hay 32 núcleos por cada **SM**, lo que hace un total de 256 núcleos de procesamiento. Este diseño hardware permite la programación sencilla de los núcleos de la **GPU** utilizando un lenguaje de alto nivel como puede ser el lenguaje C para **CUDA**. De este modo, el programador simplemente escribe un programa secuencial dentro del cual se llama a lo que se conoce como ***kernel***, que puede ser una simple función o un programa completo. Este *kernel* se ejecuta de forma paralela dentro de la **GPU** como un conjunto de hilos (***threads***) y que el programador organiza dentro de una jerarquía en la que pueden agruparse en bloques (***blocks***), y que a su vez se pueden distribuir formando una malla (***grid***), tal como se muestra en la Figura i.4. Por conveniencia, los

bloques y las mallas pueden tener una, dos o tres dimensiones. Existen multitud de situaciones en las que los datos con los que se trabaja poseen de forma natural una estructura de malla, pero en general, descomponer los datos en una jerarquía de hilos no es una tarea fácil. Así pues, un bloque de hilos es un conjunto de hilos concurrentes que pueden cooperar entre ellos a través de mecanismos de sincronización y compartir accesos a un espacio de memoria exclusivo de cada bloque. Y una malla es un conjunto de bloques que pueden ser ejecutados independientemente y que por lo tanto pueden ser lanzados en paralelo en los *Streaming Multiprocessors* (**SM**).

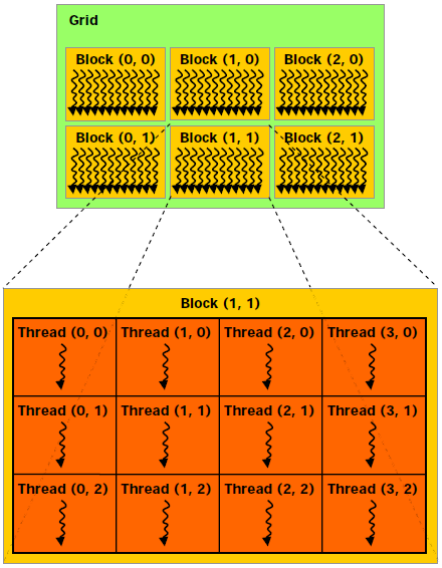


Figura i.4. Jerarquía de hilos en una aplicación CUDA.

Cuando se invoca un *kernel*, el programador especifica el número de hilos por bloque y el número de bloques que conforman la malla. Una vez en la **GPU**, a cada hilo se le asigna un único número de identificación dentro de su bloque, y cada bloque recibe un identificador dentro de la malla. Esto permite que cada hilo decida sobre qué datos tiene que trabajar, lo que simplifica enormemente el direccionamiento de memoria cuando se trabaja con datos multidimensionales, como es el caso del procesado de imágenes o la resolución de ecuaciones diferenciales en dos y tres dimensiones.

Otro aspecto a destacar en la arquitectura **CUDA** es la presencia de una unidad de distribución de trabajo que se encarga de distribuir los bloques entre los **SM** disponibles. Los hilos dentro de cada bloque se ejecutan concurrentemente y cuando un bloque termina, la unidad de distribución lanza nuevos bloques sobre los **SM** libres. Los **SM** mapean cada hilo sobre un núcleo **SP**, y cada hilo se ejecuta de manera independiente con su propio contador de programa y registros de estado. Dado que cada hilo tiene asignados sus propios registros, no

existe penalización por los cambio de contexto, pero en cambio sí existe un límite en el número máximo de hilos activos debido a que cada **SM** tiene un numero determinado de registros.

Una característica particular de la arquitectura **CUDA** es la agrupación de los hilos en grupos de 32. Un grupo de 32 hilos recibe el nombre de ***warp***, y se puede considerar como la unidad de ejecución en paralelo, ya que todos los hilos de un mismo *warp* se ejecutan físicamente en paralelo y por lo tanto comienzan en la misma instrucción (aunque después son libres de bifurcarse y ejecutarse independientemente). Así, cuando se selecciona un bloque para su ejecución dentro de un **SM**, el bloque se divide en *warps*, se selecciona uno que esté listo para ejecutarse y se emite la siguiente instrucción a todos los hilos que forman el *warp*. Dado que todos ellos ejecutan la misma instrucción al unísono, la máxima eficiencia se consigue cuando todos los hilos coinciden en su ruta de ejecución (sin bifurcaciones). Aunque el programador puede ignorar este comportamiento, conviene tenerlo en cuenta si se pretende optimizar alguna aplicación.

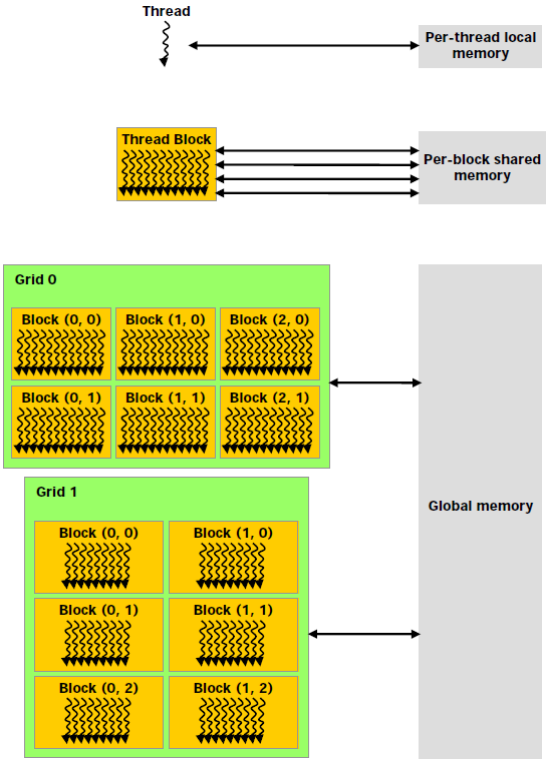


Figura i.5. Jerarquía de memoria dentro de la arquitectura CUDA.

En cuanto a la memoria, durante su ejecución los hilos pueden acceder a los datos desde diferentes espacios dentro de una jerarquía de memoria (Figura i.5). Así, cada hilo tiene una zona privada de **memoria local** y cada bloque tiene una zona de **memoria compartida**

visible por todos los hilos del mismo bloque, con un elevado ancho de banda y baja latencia (similar a una cache de nivel 1). Finalmente, todos los hilos tienen acceso a un mismo espacio de **memoria global** (del orden de MiB o GiB) ubicada en un chip externo de memoria **DRAM**. Dado que esta memoria posee una latencia muy elevada, es una buena práctica copiar los datos que van a ser accedidos frecuentemente a la zona de memoria compartida.

El modelo de programación **CUDA** asume que tanto el *host* como el *device* mantienen sus propios espacios separados de memoria. La única zona de memoria accesible desde el *host* es la memoria global. Además, tanto la reserva o liberación de memoria global como la transferencia de datos entre el *host* y el *device* debe hacerse de forma explícita desde el *host* mediante llamadas a funciones específicas de **CUDA**.

3. Capacidad de cómputo

La capacidad de una **GPU** para abordar un problema depende de los recursos hardware que tenga disponible, esto es, del número de núcleos de cómputo (**SP**), del número de hilos que puede manejar simultáneamente, del número de registros disponibles, de la cantidad de

TABLA 1. Especificaciones de un dispositivo CUDA con capacidad de cómputo 1.0.

- The maximum number of threads per block is 512;
- The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512, and 64, respectively;
- The maximum size of each dimension of a grid of thread blocks is 65535;
- The warp size is 32 threads;
- The number of registers per multiprocessor is 8192;
- The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks (see Section 5.1.2.5);
- The total amount of constant memory is 64 KB;
- The total amount of local memory per thread is 16 KB;
- The cache working set for constant memory is 8 KB per multiprocessor;
- The cache working set for texture memory varies between 6 and 8 KB per multiprocessor;
- The maximum number of active blocks per multiprocessor is 8;
- The maximum number of active warps per multiprocessor is 24;
- The maximum number of active threads per multiprocessor is 768;
- For a one-dimensional texture reference bound to a CUDA array, the maximum width is 2^{13};
- For a one-dimensional texture reference bound to linear memory, the maximum width is 2^{27};
- For a two-dimensional texture reference bound to linear memory or a CUDA array, the maximum width is 2^{16} and the maximum height is 2^{15};
- For a three-dimensional texture reference bound to a CUDA array, the maximum width is 2^{11}, the maximum height is 2^{11}, and the maximum depth is 2^{11};
- The limit on kernel size is 2 million *PTX* instructions;

TABLA 2. Núcleos de procesamiento (**SP**) de un dispositivo CUDA en función de su capacidad de cómputo.

Capacidad de Cómputo	Nombre de la Arquitectura	Núcleos por multiprocesador (**SP**/**SM**)
1.x	Tesla	8
2.0	Fermi	32
2.1	Fermi	48
3.x	Kepler	192
5.x	Maxwell	128

memoria local, constante o compartida del dispositivo, etc. Dentro de la terminología de **CUDA** esto recibe el nombre de **capacidad de cómputo** (*Compute Capability*) y se indica mediante dos números de la forma ***M.m***, que representan la revisión mayor y la revisión menor de la arquitectura del dispositivo, respectivamente.

Los dispositivos con un mismo número de revisión mayor se han diseñado con la misma arquitectura. Por otro lado, el número de revisión menor indica una mejora adicional en dicha arquitectura, posiblemente incluyendo nuevas características. En la TABLA 1 se muestran las especificaciones básicas que posee un dispositivo con una capacidad de cómputo 1.0. Por otro lado, en la TABLA 2 aparece el nombre de las diferentes arquitecturas así como el número de núcleos de cómputo o *Streaming Processors* (**SP**) característicos de cada una de ellas en función de la capacidad de cómputo.

Estas y otras características importantes como son el número de multiprocesadores (**SM**), la frecuencia de reloj o la cantidad de memoria global disponible que son particulares de cada implementación pueden ser consultadas en tiempo de ejecución mediante llamadas a funciones diseñadas para ese fin.

PRÁCTICA nº 1: Dispositivos CUDA

1. Entorno de desarrollo CUDA

Para programar en la arquitectura CUDA los desarrolladores pueden hoy en día utilizar el lenguaje C estándar, uno de los lenguajes de programación de alto nivel más usado en el mundo. La arquitectura CUDA y su software asociado fueron desarrollados teniendo en cuenta varios propósitos:

- Proporcionar un pequeño conjunto de extensiones a los lenguajes de programación estándar, como C, que permitieran la implementación de algoritmos paralelos de manera sencilla. Con CUDA y **CUDA-C**, los programadores se pueden centrar en la tarea de paralelizar algoritmos en vez de invertir el tiempo en su implementación.
- Soportar la computación en sistemas heterogéneos, donde las aplicaciones utilizan tanto la CPU con la GPU. La porción secuencial de la aplicación se sigue ejecutando en la CPU, y la porción paralela se descarga sobre la GPU. La CPU y la GPU son tratadas como dispositivos independientes que tienen sus propios espacios de memoria. Esta configuración también permite la computación simultánea tanto en la CPU como en la GPU sin competir por los recursos de memoria.

1.1. Requisitos del sistema

Para poder ejecutar aplicaciones CUDA se necesita tener instalado en nuestro sistema (Microsoft Windows XP, Vista, W7, o W8) los siguientes elementos:

- **Tarjeta gráfica NVIDIA** con la característica "CUDA-enabled". Esto no es difícil encontrar hoy en día ya que desde la salida al mercado de la tarjeta GeForce 8800 GTX todas tienen dicha característica.
- **Driver de dispositivo**. Nunca está de más tener instalado el driver más reciente.
- **Compilador de C** estándar, que en sistemas basados en Windows puede ser Microsoft Visual Studio 2008 o superior.
- **Software de CUDA** (disponible en http://www.nvidia.com/cuda).

1.2. Instalación del software

El software proporcionado por NVIDIA para crear y lanzar aplicaciones CUDA está dividido en dos partes:

- **CUDA Toolkit**, que contiene el compilador **nvcc**, necesario para construir una aplicación CUDA y que se encarga de separar el código destinado al *host* del código destinado al *device*. Incluye además librerías, ficheros de cabecera y otros recursos.
- **CUDA SDK** (*Software Development Kit*), que incluye multitud de ejemplos y proyectos prototipo configurados para poder construir de manera sencilla una aplicación CUDA utilizando Microsoft Visual Studio.

A partir de la versión 5.0 de CUDA el entorno de programación viene integrado en un único archivo instalador de tipo `.msi`, siendo recomendable seguir los pasos indicados en la guía "*Getting Started Guide*" y leer las correspondientes "*Release Notes*.

Sin embargo, si queremos instalar versiones anteriores necesitamos instalar ambos paquetes por separado, siendo los pasos a seguir para instalar el software de NVIDIA los siguientes:

- Desinstalar cualquier versión que hayamos instalado previamente de NVIDIA CUDA Toolkit y NVIDIA CUDA SDK. Esto es estrictamente necesario sólo para las versiones 3.1 y anteriores, ya que desde la versión 3.2 se utilizan directorios distintos para cada una de ellas y se pueden instalar múltiples versiones simultáneamente.
- Instalar el NVIDIA CUDA Toolkit.
- Instalar el NVIDIA CUDA SDK.

1.3. Comprobación de la instalación

Una vez instalado el software es importante comprobar que los programas CUDA que vamos a desarrollar pueden encontrar y comunicarse con el hardware. Para ello es recomendable compilar y ejecutar alguno de los programas de ejemplo que vienen incluidos en el SDK. Estos programas de ejemplo así como los archivos de proyecto necesarios para su compilación están organizados en diferentes carpetas y se encuentran ubicadas dentro de la ruta de instalación del SDK. La forma más rápida de llegar a esta ubicación es mediante el "*NVIDIA GPU Computing SDK Browser*", que es un explorador que nos proporciona el propio SDK (Figura 1.1) y que es accesible desde el *escritorio* a través de su correspondiente acceso directo. Desde la ventana de este explorador podemos acceder a los diferentes

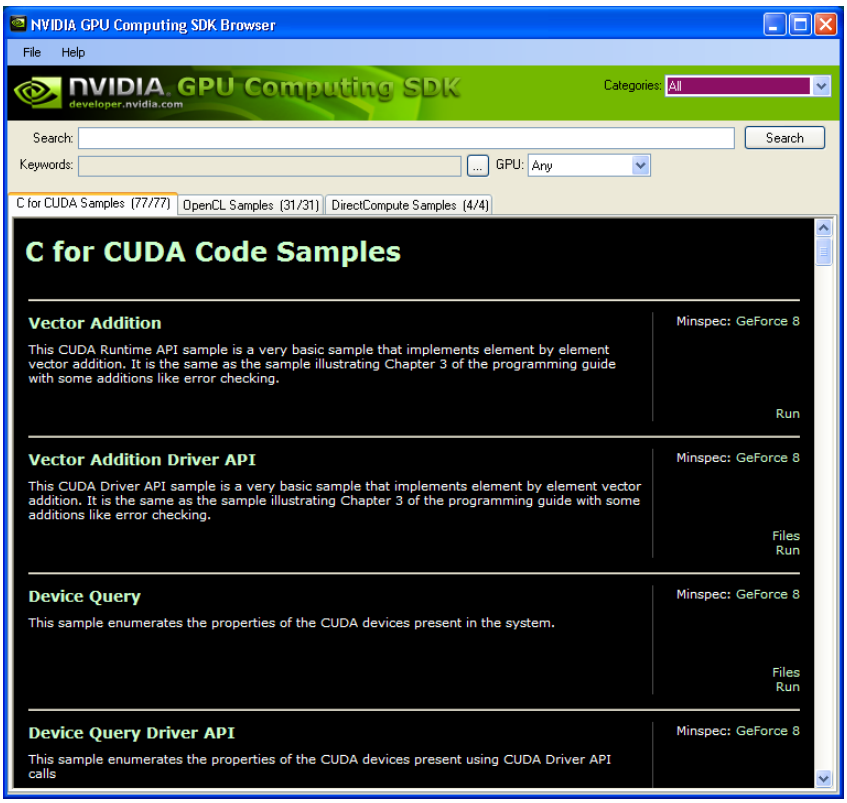


Figura 1.1. NVIDIA GPU Computing SDK Browser.

ejemplos suministrados y ejecutarlos directamente. De este modo, pinchando sobre la opción "*Run*" de alguno de los ejemplos, se abrirá la ventana de comandos de Windows con el resultado de la ejecución del programa seleccionado. Pinchando sobre la opción "*Files*", se abrirá el explorador de Windows directamente sobre la carpeta de CUDA-C en la que se encuentran todos los ejemplos con los ficheros de configuración de proyecto y los ficheros fuente. A partir de estos archivos podemos construir de forma rápida un proyecto en Visual Studio, y se pueden construir proyectos de 32-bit o 64-bit (en modo *release* o modo *debug*) usando los archivos de solución **`*.sln`** para diferentes versiones de Microsoft Visual Studio (versiones 2005, 2008, y 2010). Si ejecutamos alguno de los ejemplos sin problema querrá decir el entorno de NVIDIA para CUDA se ha instalado correctamente.

2. Configuración de un proyecto en Visual Studio C++

Crear una nueva aplicación CUDA a partir del entorno proporcionado por el NVIDIA GPU Computing SDK es sencillo. Basta con utilizar el proyecto que se suministra dentro de los ejemplos como plantilla ("*template*") y modificarlo a nuestra conveniencia. Para ello debemos realizar los siguientes pasos:

1. Copiar el contenido de la carpeta "`\CUDA Samples\v5.0\0_Simple\template`" en una nueva carpeta creada en la misma ruta que el resto de ejemplos y con el nombre que deseemos (por ejemplo `\CUDA Samples\v5.0\practicas`).
2. Eliminar los archivos de proyecto y solución que no correspondan a nuestra versión de Visual Studio (por ejemplo, para la versión 2010 nos quedaremos con los ficheros "`template_vs2010.vcxproj`" y "`template_vs2010.sln`").
3. Abrir la solución "`template_vs2010.sln`" con Visual Studio y construirla para *win32* en modo "*debug*".
4. Ejecutar el resultado "`template.exe`" desde Visual Studio o bien desde su ubicación en `\CUDA Samples\v5.0\bin\win32\Debug`.

Si hemos logrado construir y ejecutar el proyecto sin errores querrá decir que todo el entorno de NVIDIA para CUDA está instalado correctamente y que podemos comenzar a editar los diferentes archivos fuente y modificarlos para realizar nuestros propios cálculos. Para ello comenzamos eliminando de nuestro proyecto los archivos "`template_cpu.cpp`" y "`template_kernel.cu`". De este modo, nuestro proyecto contendrá un único archivo fuente llamado "`template.cu`", que es el único que debemos utilizar para escribir nuestro propio código.

Por otro lado, aunque no es necesario, podemos estar interesados en renombrar nuestros archivos y utilizar otro nombre, como por ejemplo "*practica*" . Para ello, una vez cerrado Visual Studio y con los cambios anteriores guardados, podemos proceder de la siguiente forma:

5. Modificar los nombres de los ficheros fuente, proyecto y solución sustituyendo la palabra "*template*" por otro nombre (por ejemplo, por la palabra "*practica*"). De este modo la carpeta quedará con 3 archivos:
   "`practica.cu`"
   "`practica_vs2010.sln`"
   "`practica_vs2010.vcxproj`"
6. Editar el contenido de los archivos `*.sln y *.vcxproj` con un editor de texto y reemplazar todas la ocurrencias de la palabra "*template*" por la palabra escogida anteriormente (en nuestro ejemplo, por la palabra "*practica*").

3. Propiedades de un dispositivo CUDA

Además de las características generales de una tarjeta gráfica con arquitectura CUDA existen otras características que dependen de cada dispositivo en particular. Por ese motivo, es muy importante conocer las características actuales del dispositivo o dispositivos con los que vamos a trabajar, como por ejemplo cuánta memoria y qué capacidad de cómputo posee. Por otro lado, si disponemos de más de un dispositivo, también es conveniente poder seleccionar el que mejor se adapte a nuestras necesidades.

Así pues, lo primero que tenemos que hacer es saber cuántos dispositivos CUDA tenemos instalados en nuestro sistema. Para ello llamamos a la función **`cudaGetDeviceCount()`**:

```
cudaGetDeviceCount(int *count);
```

Esta función devuelve en la variable *`count`* el número de dispositivos con capacidad de cómputo igual o superior a 1.0 (que es la especificación mínima de un dispositivo con arquitectura CUDA). Los distintos dispositivos encontrados se identifican mediante un número entero en el rango que va desde 0 hasta *`count`*-1. Una vez que conocemos el número total de dispositivos, podemos iterar a través de ellos y obtener su información utilizando la función **`cudaGetDeviceProperties()`**:

```
cudaGetDeviceProperties(cudaDeviceProp *propiedades, int deviceID);
```

Con cada llamada a esta función, todas las propiedades del dispositivo identificado con el número *`deviceID`* quedan almacenadas en *`propiedades`*, que es una estructura del tipo **`cudaDeviceProp`** y que contiene la información organizada en diferentes campos tal como se indica en la TABLA 3.

Si estamos trabajando en un entorno con varios dispositivos instalados, otras funciones que resultan de utilidad son la función **`cudaGetDevice()`** y la función **`cudaSetDevice()`**:

```
cudaGetDevice(int *getID);
cudaSetDevice(int setID);
```

La primera devuelve en la variable *`getID`* el número de identificación del dispositivo seleccionado mientras que la segunda nos permite seleccionar un dispositivo cuyo número de identificación sea *`setID`*.

TABLA 3. Campos contenidos en la estructura **`cudaDeviceProp`**.

DEVICE PROPERTY	DESCRIPTION
`char name[256];`	An ASCII string identifying the device (e.g., `"GeForce GTX 280"`)
`size_t totalGlobalMem`	The amount of global memory on the device in bytes
`size_t sharedMemPerBlock`	The maximum amount of shared memory a single block may use in bytes
`int regsPerBlock`	The number of 32-bit registers available per block
`int warpSize`	The number of threads in a warp
`size_t memPitch`	The maximum pitch allowed for memory copies in bytes

DEVICE PROPERTY	DESCRIPTION
`int maxThreadsPerBlock`	The maximum number of threads that a block may contain
`int maxThreadsDim[3]`	The maximum number of threads allowed along each dimension of a block
`int maxGridSize[3]`	The number of blocks allowed along each dimension of a grid
`size_t totalConstMem`	The amount of available constant memory
`int major`	The major revision of the device's compute capability
`int minor`	The minor revision of the device's compute capability
`size_t textureAlignment`	The device's requirement for texture alignment
`int deviceOverlap`	A boolean value representing whether the device can simultaneously perform a `cudaMemcpy()` and kernel execution
`int multiProcessorCount`	The number of multiprocessors on the device
`int kernelExecTimeoutEnabled`	A boolean value representing whether there is a runtime limit for kernels executed on this device
`int integrated`	A boolean value representing whether the device is an integrated GPU (i.e., part of the chipset and not a discrete GPU)
`int canMapHostMemory`	A boolean value representing whether the device can map host memory into the CUDA device address space

REALIZACIÓN PRÁCTICA

1. Comprobar que el entorno de programación en **CUDA** está en orden y funciona correctamente escribiendo el ejemplo inicial por excelencia, aquel en el que se muestra por pantalla el mensaje de bienvenida: *"¡Hola, mundo!"*:

```
////////////////////////////////////////////////////////////////////////////
// includes
////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

////////////////////////////////////////////////////////////////////////////
// defines
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// declaracion de funciones
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// rutina principal ejecutada en el host
int main(int argc, char** argv)
{
	// cuerpo del programa
	printf("Hola, mundo!!\n");

	// salida del programa
	printf("\npulsa INTRO para finalizar...");
	fflush(stdin);
	char tecla = getchar();
	return 0;
}
////////////////////////////////////////////////////////////////////////////
```

2. Buscar todos los dispositivos instalados en el sistema que sean compatibles con CUDA y mostrar las siguientes propiedades de cada uno de ellos:

- » Nombre del dispositivo.
- » Capacidad de cómputo.
- » Número de multiprocesadores, *Streaming Multiprocessors* (**SM**).
- » Número de núcleos de cómputo, *Streaming Processors* (**SP**)[1].
- » Memoria global (en MiB).

[1] Consultar TABLA 2.

PRÁCTICA nº 2:
Transferencia de datos

1. Manejo de memoria en CUDA

Como ya hemos mencionado anteriormente, el modelo de programación CUDA asume un sistema compuesto por un *host* y un *device*, y cada uno de ellos con su espacio de memoria. Un *kernel* sólo pueden operar sobre la memoria del dispositivo, por lo que necesitaremos funciones específicas para reservar y liberar la memoria del dispositivo, así como funciones para la transferencia datos entre la memoria del *host* y del *device*. En la Figura 2.1 se han representado los diferentes niveles dentro de la jerarquía de memoria. Vemos que las únicas zonas de memoria accesibles desde el *host* son la **memoria global** (*Global Memory*) y la **memoria constante** (*Constant Memory*) y desde estas zonas de memoria el *kernel* puede transferir datos al resto de niveles. Se puede observar cómo todos los hilos pueden acceder a la zona de memoria global/constante (lo que resulta en un canal de comunicación entre todos ellos) mientras que únicamente sólo los hilos pertenecientes a un mismo bloque pueden acceder a una zona de memoria denominada **memoria compartida**

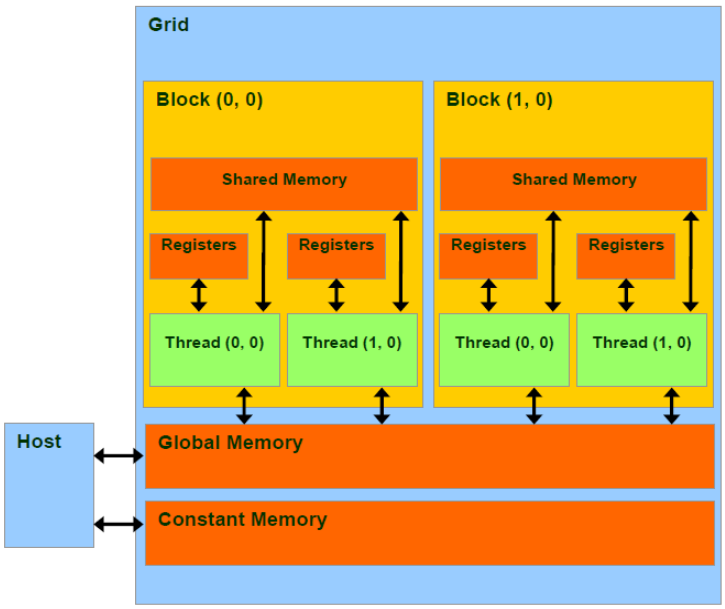


Figura 2.1. Jerarquía de memoria en CUDA.

(*Shared Memory*).

Para poder reservar espacio en la zona de memoria global del *device* y poder acceder a ella desde el *host* se utiliza la función **`cudaMalloc()`**, que tiene un comportamiento similar a la correspondiente función estándar de C:

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

El primer argumento de esta función es un doble puntero, que corresponde a la dirección del puntero (`devPtr`) en el que vamos a almacenar la dirección de la memoria reservada en el dispositivo. El segundo argumento es la cantidad de memoria expresada en bytes que queremos reservar. De esta forma podemos reservar `size` bytes de memoria lineal dentro de la memoria global de la tarjeta gráfica.

Una vez que tenemos el espacio de memoria reservado en la memoria global de nuestro dispositivo, lo siguiente que podemos que hacer es transferir datos entre esta memoria y la memoria de nuestra CPU. Para ello utilizamos otra función parecida a la que disponemos en C estándar para tal efecto, sólo que en este caso tendremos algún parámetro adicional que nos permita especificar el origen y el destino de los datos. Esta función es **`cudaMemcpy()`**:

```
cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind);
```

El primer parámetro (`dst`) corresponde al puntero con la dirección de destino de los datos, el segundo (`src`) es el puntero con la dirección de origen, es decir, donde se encuentran los datos que queremos copiar, `count` es el número de bytes que vamos a transferir y `kind` es el tipo de transferencia que vamos a realizar. En la TABLA 4 tenemos las cuatro posibilidades que existen para `kind` y corresponden a los distintos sentidos de transferencia de datos entre el *host* y el *device*.

Por último, la memoria reservada por **`cudaMalloc()`** también se puede liberar, y para ello se utiliza la función **`cudaFree()`**:

TABLA 4. Tipos de transferencias de datos en CUDA.

Tipo de transferencia	Sentido de la transferencia
`cudaMemcpyHostToHost`	host → host
`cudaMemcpyHostToDevice`	host → device
`cudaMemcpyDeviceToHost`	device → host
`cudaMemcpyDeviceToDevice`	device → device

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

Con esta llamada liberamos el espacio de memoria apuntado por *devPtr*, el cual debe ser un puntero devuelto por la función **cudaMalloc()** en una llamada previa.

Otra función que puede ser de utilidad a lo largo de la ejecución de un programa es la función **cudaMemGetInfo()**:

```
cudaMemGetInfo(size_t *memLibre, size_t *memTotal);
```

que devuelve en las variables `memLibre` y `memTotal` el valor en bytes de la cantidad de memoria libre en el dispositivo así como el valor de memoria total, respectivamente.

2. Ejemplo

En el siguiente ejemplo se muestran las diferencias y las similitudes que existen a la hora de reservar memoria tanto en el *host* como en el *device*. En este ejemplo se reserva espacio para una matriz cuadrada de `N×N` elementos, se inicializa en el *host* con valores aleatorios[2] (entre 0 y 9) de tipo `float` y después se transfieren los datos desde el *host* hasta el *device*:

```
// includes
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>
#define N 8
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
// declaracion
      float *hst_matriz;
      float *dev_matriz;
// reserva en el host
      hst_matriz = (float*)malloc( N*N*sizeof(float) );
// reserva en el device
      cudaMalloc( (void**)&dev_matriz, N*N*sizeof(float) );
// inicializacion de datos
      srand ( (int)time(NULL) );
      for (int i=0; i<N*N; i++)
      {
            hst_matriz[i] = (float)( rand() % 10 );
      }
// copia de datos
      cudaMemcpy(dev_matriz, hst_matriz, N*N*sizeof(float), cudaMemcpyHostToDevice);
// salida
      cudaFree( dev_matriz )
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      return 0;
}
```

[2] La función estándar `int` **`rand()`** genera números aleatorios comprendidos entre 0 y un valor máximo definido en una constante llamada **`RAND_MAX`**. Para generar números distintos cada vez que se ejecuta el programa es necesario utilizar la función **`srand(`**`int` ***`semilla`***`)`** donde ***`semilla`*** debe ser un valor distinto en cada ejecución.

REALIZACIÓN PRÁCTICA

1. Escribir un programa que tenga como objetivo la transferencia de datos entre el *host* y el *device*, tal como se muestra en la Figura 2.2.

2. El array `hst_A` debe ser declarado de tipo `float` e inicializado con `N` elementos aleatorios comprendidos entre 0 y 1.

3. Imprimir por pantalla el contenido de los arrays `hst_A` y `hst_B` utilizando sólo dos cifras decimales y comprobar que son iguales.

4. Utilizar la función **`cudaMemGetInfo()`** para calcular la cantidad de memoria reservada en el *device* expresada en MiB.

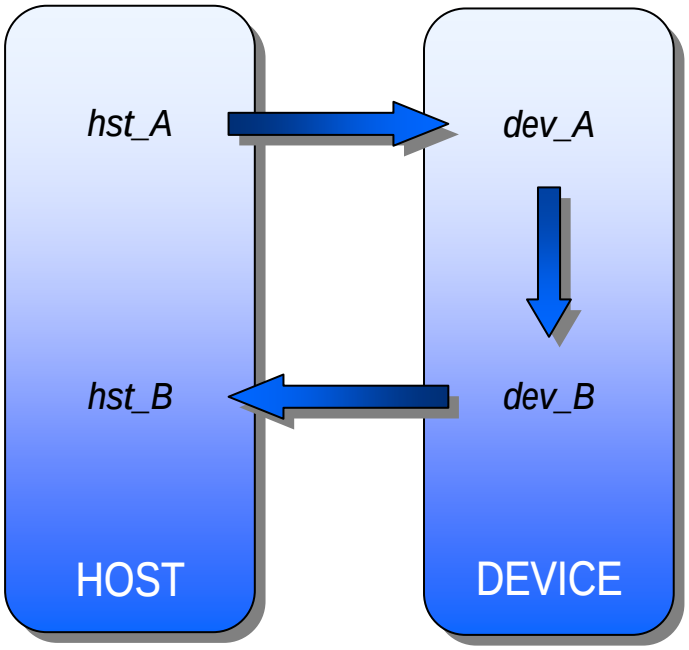


Figura 2.2. Transferencia de datos entre *host* y *device*.

PRÁCTICA nº 3:
Lanzamiento de un Kernel

1. Modelo de programación heterogéneo

Para un programador de CUDA, el sistema de cómputo consiste en una CPU tradicional (*host*), y una o más tarjetas gráficas (*device*) que son procesadores masivamente paralelos, equipados con un gran número de unidades aritmético-lógicas. En la mayoría de las aplicaciones actuales, a menudo hay secciones del programa donde los datos presentan un claro paralelismo, es decir, que sobre esos datos se pueden realizar muchas operaciones aritméticas de manera simultánea. En ese escenario, los dispositivos CUDA pueden acelerar

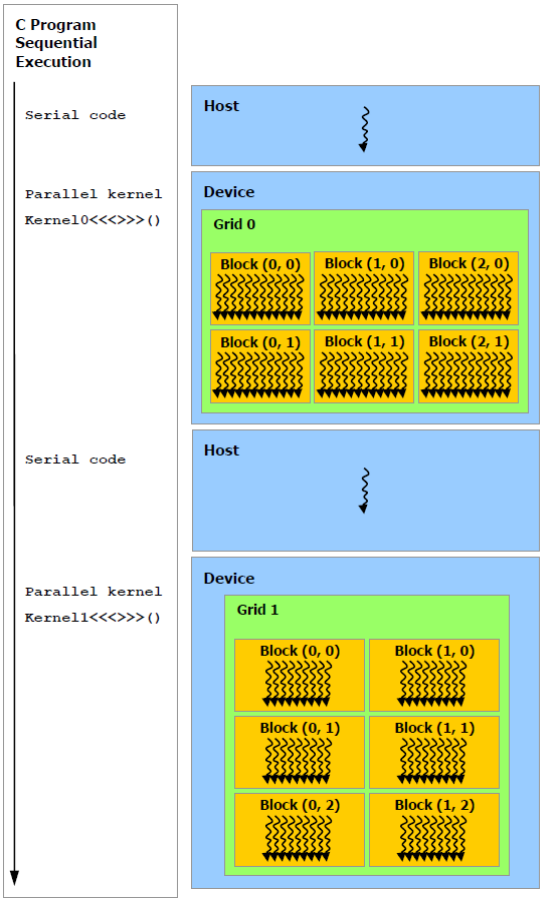


Figura 3.1. Modelo de programación heterogéneo.

dichas aplicaciones aprovechando esa gran cantidad de paralelismo en los datos. Por ello, el modelo de programación asume que el *device* funciona como un coprocesador del *host*, descargando parte del cálculo sobre la GPU y ejecutando el resto del programa sobre la CPU. La ejecución de un programa típico de CUDA se ilustra en la Figura 3.1. En esta figura se aprecia que la ejecución comienza en el *host*, y cuando se invoca un *kernel*, la ejecución se mueve hasta el *device*, donde se genera un gran número de hilos (distribuidos en bloques y formando una malla) para aprovechar el paralelismo de los datos. Cuando todos los hilos que forman el *kernel* terminan, la ejecución continúa en el *host* hasta que se lanza otro *kernel*.

De acuerdo con este modelo, el desarrollo de un programa CUDA consiste en una o más fases que se ejecutan bien en el *host* (CPU) o bien en el *device* (GPU). Las fases que no presentan paralelismo de datos se implementan en código del *host* y las fases que exhiben una gran cantidad de paralelismo de datos se implementan en código del *device*. Lo más interesante de cara al programador es que el programa se presenta como un único código fuente que incluye ambos códigos, y es el compilador de NVIDIA (**nvcc**) el encargado de separar ambos.

Llegados a este punto hay que decir que un *kernel* no es otra cosa que una función que se va a ejecutar en nuestra GPU. Por lo tanto, sigue las pautas de creación y utilización de cualquier otra función de C estándar salvo dos diferencias:

- Cambios en la declaración.
- Cambios en la sintaxis requerida para la llamada.

Los cambios en la declaración de una función que se va a ejecutar en la GPU son necesarios para que el compilador sepa distinguir qué funciones se deben ejecutar en la CPU y cuáles en la GPU. Por ello necesitamos colocar delante de cada función un determinado especificador. Mediante estos especificadores el compilador puede saber dónde se va a ejecutar una determinada función y desde dónde se la puede llamar. Los distintos especificadores o calificadores que se pueden utilizar aparecen reflejados en la TABLA 5. De acuerdo con esta clasificación, dado que un *kernel* es una función que se invoca desde la CPU

TABLA 5. Especificadores de tipo para funciones.

Especificador	Llamada desde	Ejecutada en	Ejemplo de sintaxis
`__host__`	host	host	`__host__ float HostFunc()`
`__global__`	host	device	`__global__ void KernelFunc()`
`__device__`	device	device	`__device__ float DeviceFunc()`

(*host*) y se ejecuta en la GPU (*device*), éste se debe declarar como un tipo especial denominado **__global__**. Además, vemos que el tipo de dato devuelto por un *kernel* es siempre de tipo vacío (`void`), por lo que el retorno de datos desde el *device* hasta el *host* se debe realizar siempre por referencia a través de punteros pasados como parámetros del *kernel*:

```
__global__ void myKernel(arg_1,arg_2,...,arg_n)
{
// . . . Codigo para ejecutar en la GPU . . .
}
```

Vemos que el resto de funciones declaradas como **__host__** o como **__device__** sí pueden devolver datos al regresar desde la llamada (a través de la sentencia **return**). Por otro lado, el especificador **__host__** se utiliza para las funciones llamadas y ejecutadas en la CPU, es decir, para las funciones de C estándar. Por ese motivo este especificador no es necesario y se suele utilizar únicamente para dar uniformidad al código.

La otra peculiaridad de las funciones ejecutadas en la GPU se refiere a la sintaxis requerida para hacer la llamada. Así, la sintaxis CUDA para lanzar un *kernel* es la siguiente:

```
myKernel<<<blocks,threads>>>(arg_1,arg_2,...,arg_n);
```

donde podemos observar que entre el nombre de la función y la lista de parámetros se ha añadido la expresión "**<<< , >>>**". Como en cualquier otra función escrita en lenguaje C, *myKernel* es el nombre que le hemos asignado a nuestra función, y desde *arg_1* hasta *arg_n* son los parámetros o argumentos que le pasamos a la función que va a ejecutarse en la GPU, y que pueden ser por "valor" o por "referencia" mediante punteros. El significado de las variables *blocks* y *threads* lo analizaremos en detalle más adelante y está relacionado con el número de copias de nuestra función que deseamos lanzar sobre la tarjeta gráfica.

2. Ejemplo

En el siguiente ejemplo se van a definir dos funciones para efectuar la suma de dos números. Una de ellas se va a ejecutar en la CPU y devolverá el resultado a través de una sentencia **return**, por lo que se va a declarar como **__host__** y de tipo `int`, mientras que la otra se va a ejecutar en la en la GPU (el *kernel*) y por tanto se declarará como **__global__** y de tipo `void`. Además, la llamada al *kernel* se realizará utilizando la sintaxis "**<<<1,1>>>**", lo que indica que se va a utilizar un solo bloque y un solo hilo:

```
// includes
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void suma_GPU(int a, int b, int *c)
{
      *c = a + b;
}

// HOST: funcion llamada y ejecutada desde el host
__host__ int suma_CPU(int a, int b)
{
      return (a + b);
}

// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
// declaraciones
      int n1 = 1, n2 = 2, c = 0;
      int *hst_c;
      int m1 = 10, m2 = 20;
      int *dev_c;

// reserva en el host
      hst_c = (int*)malloc( sizeof(int) );

// reserva en el device
      cudaMalloc( (void**)&dev_c, sizeof(int) );

// llamada a la funcion suma_CPU
      c = suma_CPU(n1, n2);

// resultados CPU
      printf("CPU:\n");
      printf("%2d + %2d = %2d \n",n1, n2, c);

// llamada a la funcion suma_GPU
      suma_GPU<<<1,1>>>(m1, m2, dev_c);

// recogida de datos desde el device
      cudaMemcpy( hst_c, dev_c, sizeof(int), cudaMemcpyDeviceToHost );

// resultados GPU
      printf("GPU:\n");
      printf("%2d + %2d = %2d \n",m1, m2, *hst_c);

// salida
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      return 0;
}
```

REALIZACIÓN PRÁCTICA

1. Lanzar un *kernel* compuesto por un solo bloque y un solo hilo utilizando la sintaxis "`<<<1,1>>>`" que resuelva la ecuación de segundo grado:

$$a \cdot x^2 + b \cdot x + c = 0$$

cuyas soluciones son:

$$x_1 = -\frac{b}{2a} + \frac{\sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = -\frac{b}{2a} - \frac{\sqrt{b^2 - 4ac}}{2a}$$

2. Los coeficientes de la ecuación se deben pedir por teclado utilizando la función **`scanf()`**.

3. El programa debe proporcionar las soluciones para cualquier valor de *a*, *b* y *c* introducido por el teclado, incluyendo posibles soluciones imaginarias[3].

[3] Recuerda: $\sqrt{-4} = \pm 2i$

PRÁCTICA nº 4: Hilos y Bloques

1. Organización de hilos en CUDA

Los hilos (*threads*) en CUDA constituyen la base de la ejecución de aplicaciones en paralelo. Como ya hemos comentado anteriormente, cuando se lanza un *kernel* se crea una malla de hilos donde todos ellos ejecutan el mismo programa. Es decir, el *kernel* especifica las instrucciones que van a ser ejecutadas por cada hilo individual. En la práctica anterior no hicimos uso de esta propiedad ya que lanzamos lo que podríamos llamar un "*kernel escalar*", mediante la sintaxis `<<<1,1>>>`, esto es, un bloque con un solo hilo de ejecución. Esto no es otra cosa que una función estándar de C que se ejecuta de manera secuencial. En esta práctica vamos a hacer uso del paralelismo a nivel de hilo que nos proporciona CUDA lanzando más de un hilo. Las opciones para esta labor son múltiples, ya que podemos lanzar todos los hilos en un único bloque, podemos lanzar varios bloques con un solo hilo cada uno, o la opción más flexible que sería lanzar varios bloques con varios hilos en cada bloque.

Por ejemplo, para lanzar un *kernel* con *N* hilos de ejecución, todos ellos agrupados en un único bloque, la sintaxis sería:

```
myKernel<<<1,N>>>(arg_1,arg_2,...,arg_n);
```

De este modo, la GPU ejecuta simultáneamente *N* copias de nuestro *kernel*. La forma de aprovechar este paralelismo que nos brinda la GPU es hacer que cada una de esas copias o hilos realice la misma operación pero con datos distintos, es decir, asignar a cada hilo sus propios datos. La pregunta que nos podemos hacer en este punto es ¿y cómo podemos identificar cada uno de los hilos para poder repartir el trabajo? La respuesta está en una variable incorporada (*built-in*) denominada **`threadIdx`**, que es de tipo `int` y que únicamente puede utilizarse dentro del código del *kernel*. Esta variable adquiere un valor distinto para cada hilo en tiempo de ejecución. Cada hilo puede almacenar su número de identificación en una variable entera:

```
int myID = threadIdx.x;
```

Cuando antes lanzamos el *kernel* mediante la sintaxis anterior, especificamos que queríamos una malla formada por un único bloque y *N* hilos paralelos. Esto le dice al sistema que queremos una malla unidimensional (los valores escalares se interpretan como unidimensionales) con los hilos repartidos a lo largo del eje *x* (Figura 4.1), ya que por defecto se considera este eje como la dimensión de trabajo (de ahí que se añada el sufijo "**`.x`**" para indicar dicha dirección). De este modo, cada uno de los hilos tendrá un valor distinto de **`threadIdx.x`** que irá desde 0 hasta *N*–1. Es decir, cada hilo tendrá su propio valor de *`myID`* que permitirá al programador decidir sobre qué datos debe trabajar cada uno de ellos.

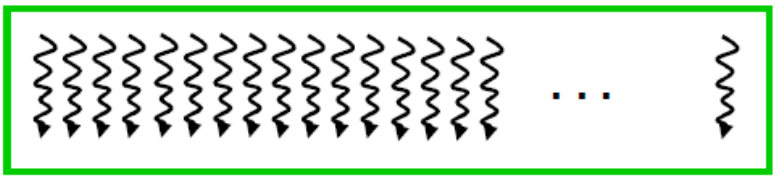


Figura 4.1. Malla (*grid*) formada por un único bloque (*block*) de *N* hilos paralelos (*threads*) repartidos a lo largo del eje *x*. Cada hilo se puede identificar mediante la variable **`threadIdx.x`**.

El hardware de la GPU limita el número de hilos por bloque con que podemos lanzar un *kernel*. Este número puede ser mayor o menor dependiendo de la capacidad de cómputo de nuestra GPU y en particular no puede superar el valor dado por **`maxThreadsPerBlock`**, que es uno de los campos que forman parte de la estructura de propiedades **`cudaDeviceProp`**. Para las arquitecturas con capacidad de cómputo 1.0 este límite es de 512 hilos por bloque.

Otra alternativa para lanzar un *kernel* de *N* hilos consistiría en lanzar *N* bloques pero de un hilo cada uno. En este caso la sintaxis para el lanzamiento sería:

```
myKernel<<<N,1>>>(arg_1,arg_2,...,arg_n);
```

Mediante esta llamada tendríamos una malla formada por *N* bloques repartidos a lo largo del eje x y con un hilo cada uno (Figura 4.2).

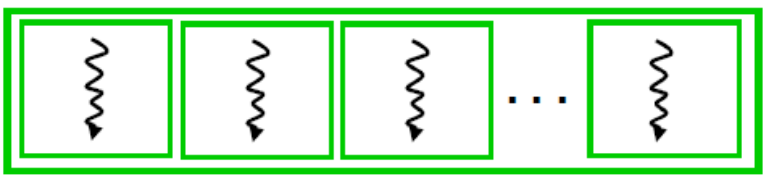


Figura 4.2. Malla formada por *N* bloques paralelos de 1 hilo repartidos a lo largo del eje *x*. Cada bloque se puede identificar mediante la variable **`blockIdx.x`**.

Si ahora queremos identificar los hilos para poder repartir tareas, no podemos utilizar la variable anterior (**`threadIdx.x`**), ya que esta variable sólo permite identificar a los hilos que pertenecen al mismo bloque. Esto quiere decir que dentro de cada bloque todos los hilos comienzan a numerarse a partir del número 0, y por lo tanto, en nuestro ejemplo, al haber un único hilo en cada bloque, todos los hilos tendrían el mismo valor de **`threadIdx.x`** y por tanto en todos ellos tendríamos *`myID`* = 0. Ahora la identificación de los hilos pasa por identificar el bloque en el que se encuentra cada hilo. Esto lo podemos hacer mediante otra variable similar a la anterior denominada **`blockIdx`**:

```
int myID = blockIdx.x;
```

Al igual que antes, cada uno de los hilos tendrá un valor distinto de **`blockIdx.x`** que irá desde 0 hasta N–1. También existe un límite impuesto por el hardware en el número máximo de bloques que podemos lanzar, que para la mayoría de las arquitecturas es de 65.535 bloques. El valor particular para nuestra GPU lo podemos averiguar a partir del campo **`maxGridSize[i]`** de nuestra estructura de propiedades, y que nos da el máximo número de bloques permitidos en cada una de las direcciones del espacio (`i` = 0, 1, 2 ⇒ ejes x, y, y z respectivamente).

Por último, el caso más general sería el lanzamiento de un *kernel* con M bloques y N hilos por bloque (Figura 4.3), en cuyo caso tendríamos un total de $M \times N$ hilos . La sintaxis en este caso sería:

```
myKernel<<<M,N>>>(arg_1,arg_2,...,arg_n);
```

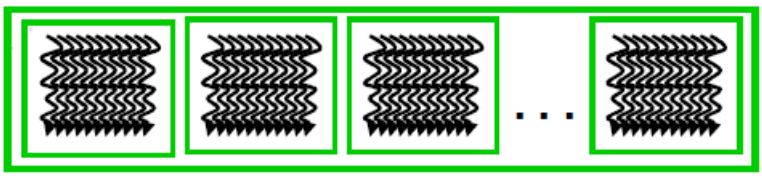


Figura 4.3. Malla formada por M bloques paralelos (**`gridDim.x`**) de N hilos cada uno (**`blockDim.x`**) repartidos a lo largo del eje x.

Ahora habrá un total de $M \times N$ hilos ejecutándose en paralelo que para identificarlos será necesario hacer uso conjunto de las dos variables anteriores y de dos nuevas constantes que permitan a la GPU conocer en tiempo de ejecución las dimensiones del *kernel* que hemos lanzado. Estas dos constantes son **`gridDim.x`** y **`blockDim.x`**. La primera nos da el número de bloques (en nuestro caso sería M) y la segunda el número de hilos que tiene cada bloque (N en

nuestro ejemplo). De este modo, dentro del *kernel* cada hilo se puede identificar de forma unívoca mediante la siguiente expresión (figura 4.4):

```
int myID = threadIdx.x + blockDim.x * blockIdx.x;
```

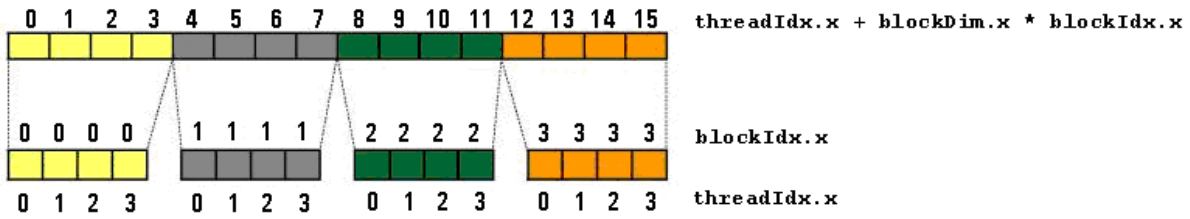


Figura 4.4. Identificación de los hilos dentro de una malla formada por cuatro bloques de cuatro hilos cada uno y repartidos a lo largo del eje *x*.

2. Ejemplo

En el siguiente ejemplo se muestra forma de aprovechar el paralelismo de datos programando un *kernel* que realice la suma de dos vectores de longitud `N` inicializados con valores aleatorios comprendidos entre 0 y 1:

```
// includes
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
// defines
#define N 16    // tamano de los vectores
#define BLOCK 5 // tamano del bloque
// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void suma( float *a, float *b, float *c )
{
      int myID = threadIdx.x + blockDim.x * blockIdx.x;
      // Solo trabajan N hilos
      if (myID < N)
      {
            c[myID] = a[myID] + b[myID];
      }
}
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
// declaraciones
      float *vector1,     *vector2,     *resultado;
      float *dev_vector1, *dev_vector2, *dev_resultado;
      // reserva en el host
      vector1   = (float *)malloc(N*sizeof(float));
      vector2   = (float *)malloc(N*sizeof(float));
      resultado = (float *)malloc(N*sizeof(float));
// reserva en el device
      cudaMalloc( (void**)&dev_vector1,   N*sizeof(float));
      cudaMalloc( (void**)&dev_vector2,   N*sizeof(float));
      cudaMalloc( (void**)&dev_resultado, N*sizeof(float));
// inicializacion de vectores
      for (int i = 0; i < N; i++)
      {
            vector1[i] = (float) rand() / RAND_MAX;
            vector2[i] = (float) rand() / RAND_MAX;
      }
```

```
// copia de datos hacia el device
      cudaMemcpy(dev_vector1, vector1, N*sizeof(float), cudaMemcpyHostToDevice);
      cudaMemcpy(dev_vector2, vector2, N*sizeof(float), cudaMemcpyHostToDevice);
// lanzamiento del kernel
// calculamos el numero de bloques necesario para un tamaño de bloque fijo
      int nBloques = N/BLOCK;
      if (N%BLOCK != 0)
      {
            nBloques = nBloques + 1;
      }
      int hilosB = BLOCK;
      printf("Vector de %d elementos\n", N);
      printf("Lanzamiento con %d bloques (%d hilos)\n", nBloques, nBloques*hilosB);
      suma<<< nBloques, hilosB >>>( dev_vector1, dev_vector2, dev_resultado );
// recogida de datos desde el device
      cudaMemcpy(resultado, dev_resultado, N*sizeof(float), cudaMemcpyDeviceToHost);
// impresion de resultados
      printf( "> vector1:\n");
      for (int i = 0; i < N; i++)
      {
            printf("%.2f ", vector1[i]);
      }
      printf("\n");
      printf( "> vector2:\n");
      for (int i = 0; i < N; i++)
      {
            printf("%.2f ", vector2[i]);
      }
      printf("\n");
      printf( "> SUMA:\n");
      for (int i = 0; i < N; i++)
      {
            printf("%.2f ", resultado[i]);
      }
      printf("\n");
// liberamos memoria en el device
      cudaFree( dev_vector1 );
      cudaFree( dev_vector2 );
      cudaFree( dev_resultado );
// salida
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      return 0;
}
```

REALIZACIÓN PRÁCTICA

1. Ejecutar un *kernel* compuesto por 24 hilos que se encargue de rellenar tres arrays con los índices de identificación de cada hilo (un array por cada tipo de índice).

2. Cada hilo escribirá en el correspondiente array su índice de hilo (**`threadIdx.x`**), su índice de bloque (**`blockIdx.x`**) y su índice global (**`threadIdx.x + blockDim.x * blockIdx.x`**).

3. El *kernel* se debe ejecutar tres veces, cada una de ellas con diferentes opciones de organización:

- 1 bloque de 24 hilos.
- 24 bloques de 1 hilo.
- 4 bloques de 6 hilos.

4. En cada ejecución se debe mostrar por pantalla el contenido de los tres arrays:

```
>> Opcion 1: 1 bloque 24 hilos
indice de hilo:
0     1     2     3     4     5     6     7 ...
indice de bloque:
0     0     0     0     0     0     0     0 ...
indice global:
0     1     2     3     4     5     6     7 ...

>> Opcion 2: 24 bloques 1 hilo
...
...
...

>> Opcion 3: 4 bloques 6 hilos
...
...
...
```

PRÁCTICA nº 5:
Temporización y Errores

1. Corrientes (*Streams*)

Hasta ahora hemos visto cómo el procesamiento de datos en paralelo en una GPU puede proporcionar grandes mejoras en el rendimiento en comparación con el mismo código ejecutado en una CPU. Sin embargo, hay otra clase de paralelismo para ser explotado en los procesadores gráficos NVIDIA. Este paralelismo es similar al paralelismo de tareas (*task parallelism*) que se encuentra en las aplicaciones de una CPU multiproceso. En lugar de calcular simultáneamente la misma función sobre una gran cantidad de datos como se hace en el paralelismo de datos, el paralelismo de tareas implica hacer dos o más tareas completamente diferentes en paralelo.

En este contexto de paralelismo, una *tarea* puede ser cualquier actividad. Por ejemplo, una aplicación podría estar realizando dos tareas: volver a dibujar la pantalla de un monitor mientras se descarga una actualización en la red. Estas tareas se ejecutan en paralelo, a pesar

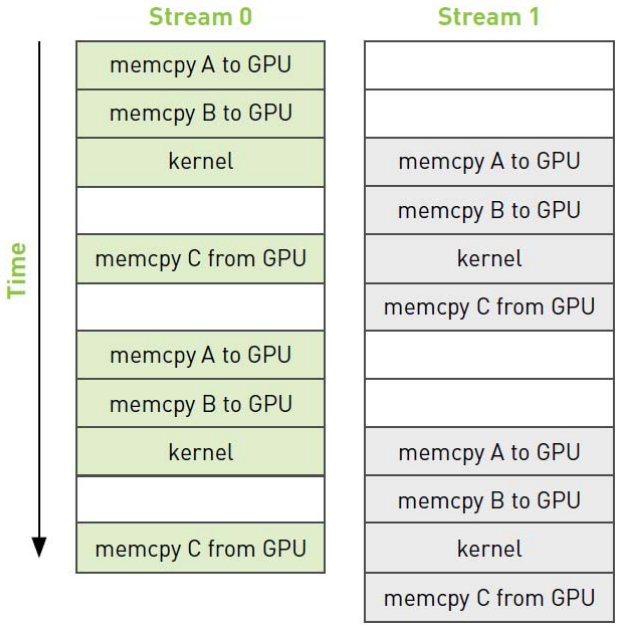


Figura 5.1. Cronograma de ejecución de una aplicación con dos corrientes independientes.

de no tener nada en común. Aunque el paralelismo de tareas en una GPU no es actualmente tan flexible como en un procesador de propósito general, puede ser una oportunidad para extraer mayor rendimiento a ciertas aplicaciones. La idea que subyace es tener siempre ocupada la GPU realizando tareas.

En CUDA, una corriente o *stream* representa una lista de operaciones que se ejecutan en la GPU en un orden específico. Estas operaciones pueden ser lanzamientos de un *kernel* o copias de memoria desde la CPU a la GPU o viceversa (Figura 5.1). El orden en que las operaciones se añaden a la corriente especifica el orden en el que serán ejecutadas. Podemos pensar en una corriente o *stream* como una lista de tareas en la GPU, y hay que aprovechar cualquier oportunidad para que estas tareas se puedan ejecutar en paralelo. La capacidad de una GPU para poder ejecutar estas tareas de forma concurrente depende de su capacidad de cómputo, y por tanto es algo que debemos averiguar a partir de la estructura de propiedades **`cudaDeviceProp`** de nuestro dispositivo. En particular, hay que consultar el valor del campo **`deviceOverlap`**, que es una variable binaria que nos informa si nuestro dispositivo soporta la ejecución de un *kernel* simultáneamente con la transferencia de datos (**`true`**) o no lo soporta (**`false`**). Para nuestros propósitos de momento nos bastará con utilizar la GPU empleando una única corriente de tareas.

2. Eventos (*Events*)

Un evento o *event* en CUDA es esencialmente una marca de tiempo en la GPU que nosotros podemos grabar en un determinado instante de tiempo. El hecho de que esta marca de tiempo se grabe en la propia GPU elimina una gran cantidad de problemas que podríamos encontrar cuando intentáramos medir el tiempo de ejecución de un *kernel* utilizando los temporizadores de la CPU. La forma de medir un tiempo de ejecución es relativamente fácil, ya que básicamente consiste en crear *eventos* para después grabar en ellos marcas temporales con el fin de calcular la diferencia de tiempos entre las marcas. Los eventos se almacenan en un nuevo tipo de dato denominado **`cudaEvent_t`**, y la función para crear un evento es **`cudaEventCreate()`**:

```
cudaEventCreate (cudaEvent_t *marca);
```

Debemos crear tantos eventos como marcas temporales vayamos a necesitar, que en general serán dos, una de inicio y otra de fin. Una vez creados los eventos donde vamos a

guardar las marcas inicial y final, podremos grabar dichas marcas temporales en el momento que deseemos. Esta grabación se realiza utilizando la función **`cudaEventRecord()`**:

```
cudaEventRecord(cudaEvent_t marca, cudaStream_t stream);
```

donde `marca` es alguna de las variables declaradas anteriormente para almacenar las marcas temporales de inicio o fin y *`stream`* es la corriente en la que grabar el evento. Habitualmente se pone un `0` para tener en cuenta todas las corrientes (si es que hay más de una). A continuación lanzamos el trabajo que queremos temporizar sobre la GPU y después grabamos sobre la marca final el tiempo transcurrido.

Lo siguiente sería calcular el tiempo transcurrido entre dos eventos. Sin embargo, es necesario añadir llamada adicional después del último **`cudaEventRecord()`** y que tiene como objetivo sincronizar el *host* con el *device*. La función requerida para ello es:

```
cudaEventSynchronize(cudaEvent_t marca);
```

Esta función detiene la CPU hasta la finalización de todo el trabajo pendiente en la GPU y que precede a la llamada más reciente a **`cudaEventRecord()`**. El porqué de la necesidad de incluir esta función tiene que ver con el hecho de que algunas de las llamadas que hacemos en CUDA realmente son *asíncronas*. Por ejemplo, cuando lanzamos un *kernel*, la GPU comienza a ejecutar su correspondiente código, pero la CPU continua ejecutando la siguiente línea de nuestro programa principal (*main*) antes de que la GPU haya terminado. Esto es muy interesante desde el punto de vista del rendimiento porque significa que podemos tener trabajando simultáneamente a la CPU y a la GPU, pero por otro lado complica la medida del tiempo ya que una llamada a la función **`cudaEventRecord()`** no implica que se grabe el tiempo sino que queda pendiente en las tareas de la GPU grabar ese tiempo. Por eso, hasta que no utilizamos la función **`cudaEventSynchronize()`** no podemos estar seguros de obtener la marca de tiempo correcta.

Tras la llamada a esta función sabemos que todo el trabajo anterior se habrá completado y por lo tanto también será seguro leer su correspondiente marca de tiempo. Por tanto, una vez que tenemos las dos marcas temporales de inicio y fin, podemos calcular el tiempo transcurrido entre ambos eventos utilizando la función **`cudaEventElapsedTime()`**:

```
cudaEventElapsedTime(float *tiempo, cudaEvent_t marca1, cudaEvent_t marca2);
```

que nos devuelve en la variable *tiempo*, de tipo `float`, el tiempo en milisegundos transcurrido entre los eventos `marca1` y `marca2`. Cabe mencionar que dado que los eventos se implementan directamente sobre la GPU, sólo sirven para temporizar código para la GPU y nunca para la CPU.

Para terminar podemos decir que los eventos creados también pueden destruirse utilizando una función denominada **`cudaEventDestroy()`**:

```
cudaEventDestroy(cudaEvent_t marca);
```

y de este modo liberamos los recursos asociados con el evento *marca*. En resumen, si queremos medir el tiempo de ejecución en la GPU es necesario incluir en nuestro código las funciones relacionadas con la gestión de eventos de manera similar a como se muestra en el siguiente ejemplo:

```
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
// ...

// declaracion de eventos
      cudaEvent_t start;
      cudaEvent_t stop;
// creacion de eventos
      cudaEventCreate(&start);
      cudaEventCreate(&stop);
// marca de inicio
      cudaEventRecord(start,0);
// codigo a temporizar en el device

//...<<< , >> ...

// marca de final
      cudaEventRecord(stop,0);
// sincronizacion GPU-CPU
      cudaEventSynchronize(stop);
// calculo del tiempo en milisegundos
      float elapsedTime;
      cudaEventElapsedTime(&elapsedTime,start,stop);
// impresion de resultados
      printf("> Tiempo de ejecucion: %f ms\n",elapsedTime);
// liberacion de recursos
      cudaEventDestroy(start);
      cudaEventDestroy(stop);
// salida
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      return 0;
}
```

3. Manejo de errores

A la hora de depurar un programa resulta muy útil disponer de algún mecanismo para localizar los diferentes errores que se hayan ido produciendo durante su ejecución. Con el fin de facilitar dicho manejo de errores, en CUDA todas las funciones (excepto los lanzamientos de un *kernel*) devuelven un código de error del tipo **`cudaError_t`**. Este código es

simplemente un valor entero y toma distintos valores dependiendo del tipo de error encontrado. Para una llamada finalizada con éxito, el código de error toma un valor definido como **cudaSuccess**. De esta forma, comprobando el código de error devuelto tras la llamada a una función podemos saber si ésta ha terminado con éxito o no. El principal inconveniente es que este procedimiento puede resultar tedioso al tener que comprobar continuamente el código de error devuelto por cada función e imprimir el mensaje de error correspondiente, como por ejemplo:

```
// ...
cudaError_t error;
// ...
error = cudaMalloc( . . . );
if (error != cudaSuccess)
{
      printf("\nERROR en cudaMalloc: %s \n", cudaGetErrorString(error) );
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      exit(-1);
}
// ...
```

donde la función **cudaGetErrorString()** se encarga de devolver un mensaje de texto explicando el tipo de error que ha sucedido:

```
char *cudaGetErrorString(cudeError_t codigo);
```

Sin embargo, resulta mucho más útil definir una función dedicada al chequeo de errores, con un único parámetro de entrada que sea el mensaje de error que deseamos que aparezca por pantalla, y colocarla justo después de la llamada a la función cuyo código de error queramos saber. Un ejemplo de este tipo de función puede ser el siguiente:

```
__host__ void check_CUDA_Error(const char *mensaje)
{
  cudaError_t error;
  cudaDeviceSynchronize();
  error = cudaGetLastError();
  if(error != cudaSuccess)
  {
      printf("ERROR %d: %s (%s)\n", error, cudaGetErrorString(error), mensaje);
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      exit(-1);
  }
}
```

En la definición de esta función se han utilizado dos nuevas funciones de CUDA que son **cudaDeviceSynchronize()** y **cudaGetLastError()**. Como ya hemos dicho anteriormente, algunas de las llamadas que hacemos en CUDA realmente son *asíncronas*, por eso la primera función es necesaria para que la CPU no continúe con la ejecución del programa hasta que la GPU no haya terminado (en realidad sólo sería necesaria para comprobar los errores después del lanzamiento de un *kernel*). En cuanto a la segunda función, ésta se encarga de capturar el

último error que se haya producido en alguna de las llamadas realizadas en tiempo de ejecución. De este modo, si queremos detectar posibles errores en alguna transferencia de datos o en el lanzamiento del *kernel*, podemos incluir en el código llamadas a la función **`check_CUDA_Error()`**, con un mensaje de error particular para cada caso, del siguiente modo:

```
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
// ...

// copia de datos hacia el device
      cudaMemcpy( dev_A, hst_A, size, cudaMemcpyHostToDevice );
      check_CUDA_Error("ERROR EN cudaMemcpy");

// ...

// llamada al kernel
      multiplica <<< nBloques,hilosB>>> (dev_A);
      check_CUDA_Error("ERROR EN multiplica");

// ...

// salida
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      return 0;
}
```

REALIZACIÓN PRÁCTICA

1. Ejecutar un *kernel* que invierta el orden de los elementos de un vector de tamaño `N` inicializado con números aleatorios comprendidos entre 0 y 9.

2. Cada hilo se debe encargar de intercambiar un único elemento.

3. El número de bloques lanzados se debe calcular imponiendo como condición que el número de hilos por bloque sea 4.

4. El *host* se encargará de generar el vector original y el *device* devolverá un nuevo vector con los elementos reordenados.

5. Temporizar la ejecución del *kernel* utilizando las funciones estudiadas anteriormente.

6. Incluir en el código del programa llamadas a la función **`check_CUDA_Error()`** para detectar posibles errores.

PRÁCTICA nº 6:
Arrays Multidimensionales

1. Distribución multidimensional de hilos

En prácticas anteriores hemos visto cómo repartir los hilos de ejecución de un *kernel* a lo largo de una de las direcciones del espacio, que por defecto era el eje *x*. Sin embargo, hay situaciones donde puede ser interesante distribuir los hilos de una manera análoga al problema que se pretende resolver, como por ejemplo las operaciones con matrices. En estos casos, con el fin de imitar la ordenación de los elementos de una matriz en filas y columnas, se pueden lanzar los hilos de un *kernel* repartidos a lo largo del eje *x* y del eje *y* (Figura 6.1). De hecho, CUDA también permite distribuir hilos en el eje *z* con el fin de adecuar la disposición espacial de los hilos a la resolución de un posible problema tridimensional.

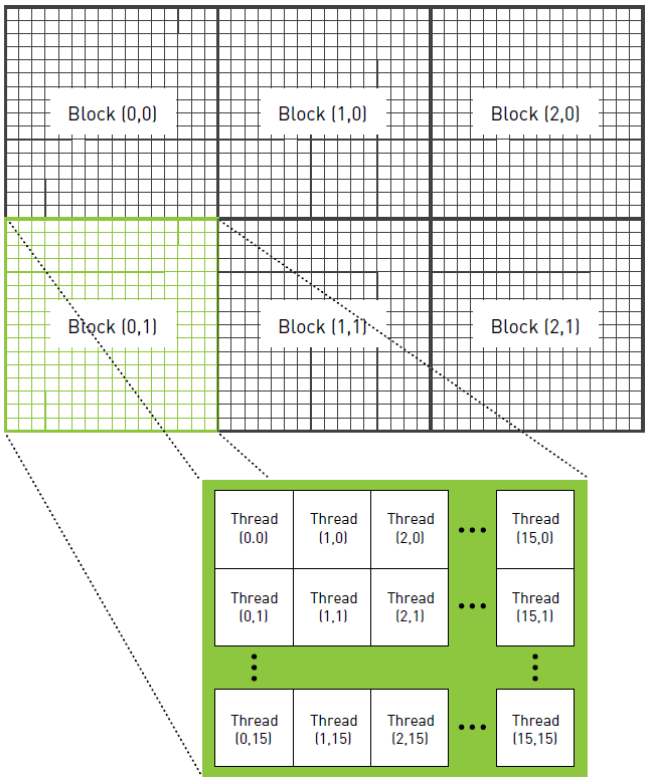


Figura 6.1. Distribución bidimensional de hilos en CUDA.

La sintaxis para lanzar un *kernel* multidimensional es la misma que en los casos anteriores, esto es:

```
myKernel<<<blocks,threads>>>(arg_1,arg_2,...,arg_n);
```

pero en este caso, las variables `blocks` y `threads` hay que declararlas como variables "tridimensionales" utilizando un nuevo tipo de dato: **`dim3`**. Al definir/declarar estas variables debemos especificar el tamaño de cada una de sus tres componentes (siempre mayor o igual que 1). Si alguna de ellas se queda sin especificar, automáticamente se inicializa a 1 (una variable escalar es lo mismo que una variable de tipo **`dim3`** con sus tres componentes inicializadas a 1):

```
dim3 blocks(Bx, By, Bz);
dim3 threads(hx, hy, hz);
```

Al igual que en la práctica anterior, una vez que hemos lanzado el *kernel* podemos identificar los bloques dentro de la malla y los hilos dentro de un bloque utilizando las variables **`blockIdx`** y **`threadIdx`** respectivamente. De este modo, en tiempo de ejecución cada una de esas variables adquirirá un valor dentro del rango que hayamos especificado en `blocks` y `threads`:

```
0≤ blockIdx.x ≤ Bx-1
0≤ blockIdx.y ≤ By-1
0≤ blockIdx.z ≤ Bz-1
0≤ threadIdx.x ≤ hx-1
0≤ threadIdx.y ≤ hy-1
0≤ threadIdx.z ≤ hz-1
```

Como ya sabemos, las dimensiones del *kernel* lanzado se almacenan en las constantes **`gridDim`** y **`blockDim`**. Estas constantes contienen los valores asignados a `blocks` y `threads` en cada una de sus dimensiones, esto es:

```
gridDim.x ≡ Bx
gridDim.y ≡ By
gridDim.z ≡ Bz
blockDim.x ≡ hx
blockDim.y ≡ hy
blockDim.z ≡ hz
```

Vemos que las posibilidades de organización de los hilos son muchas y muy variadas. Por ejemplo, si queremos distribuir los hilos formando un array bidimensional como el de la Figura 6.1, tenemos que lanzar un *kernel* con bloques e hilos repartidos a lo largo de los ejes *x* e *y* de la siguiente forma:

```
// . . .
dim3 Nbloques(3,2);
dim3 hilosB(16,16);
// . . .
myKernel<<<Nbloques,hilosB>>>( . . . );
// . . .
```

Dentro del código de la función *myKernel* podemos utilizar las variables que hemos mencionado anteriormente. De hecho, la variable **`gridDim.x`** valdrá 3 y **`gridDim.y`** valdrá 2, las variables **`blockDim.x`** y **`blockDim.y`** tomarán ambas el valor 16, mientras que las variables **`blockIdx`** y **`threadIdx`** tendrán como siempre sus valores dentro del rango definido en cada una de sus dimensiones, y que nos permitirán identificar cada bloque y cada hilo, tal como se muestra en la Figura 6.1.

Existen muchas posibilidades a la hora de distribuir los hilos. La decisión final de cómo hacerlo se puede tomar teniendo en cuenta las características de nuestra arquitectura particular, ya que no podemos sobrepasar los límites impuestos por el hardware y que dependen de su capacidad de cómputo. El siguiente código muestra cómo obtener dichas limitaciones indagando en la estructura de propiedades **`cudaDeviceProp`**. Ahora hay que tener en cuenta el número máximo de hilos permitidos en cada una de las direcciones del espacio y esto se puede obtener a partir de **`maxThreadsDim[i]`**, pero sin olvidar que en total no podemos superar el valor dado por **`maxThreadsPerBlock`**:

```
// includes
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
      cudaDeviceProp deviceProp;
      int deviceID;
      cudaGetDevice( &deviceID );
      cudaGetDeviceProperties( &deviceProp,deviceID );
      printf("MAX Threads per block: %d\n",deviceProp.maxThreadsPerBlock);
      printf("MAX BLOCK SIZE\n");
      printf(" [x -> %d]\n  [y -> %d]\n  [z -> %d]\n",deviceProp.maxThreadsDim[0],
deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2]);
      printf("MAX GRID SIZE\n");
      printf(" [x -> %d]\n  [y -> %d]\n  [z -> %d]\n",deviceProp.maxGridSize[0],
deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);
// salida
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      return 0;
}
```

2. Ejemplo

El siguiente ejemplo demuestra el gran potencial de una GPU para aprovechar el paralelismo de datos. El ejemplo consiste en un *kernel* bidimensional que realiza la suma de dos matrices cuadradas de tamaño N×N. El lanzamiento se ha realizado utilizando un único bloque con N×N hilos repartidos a lo largo de los ejes *x* e *y* de manera análoga al problema que se pretende resolver, es decir, cada hilo se va a encargar de calcular un elemento de la matriz final. Hay que tener presente que el espacio de memoria reservada en el *device* con **`cudaMalloc()`** es un espacio lineal, por lo que el acceso a los datos hay que realizarlo mediante un índice lineal obtenido a partir de los índices correspondientes a los ejes *x* e *y*:

```
// includes
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

// defines
#define N 20

// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void suma_gpu( float *A, float *B, float *C )
{
      // indice de columna
      int columna = threadIdx.x;
      // indice de fila
      int fila = threadIdx.y;
      // indice lineal
      int myID = columna + fila * blockDim.x;

      // sumamos cada elemento
      C[myID] = A[myID] + B[myID];
}

// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
// declaraciones
      float *hst_A, *hst_B, *hst_C;
      float *dev_A, *dev_B, *dev_C;

// reserva en el host
      hst_A = (float*)malloc(N*N*sizeof(float));
      . . .

// reserva en el device
      cudaMalloc( (void**)&dev_A, N*N*sizeof(float));
      . . .

// incializacion
      for(int i=0;i<N*N;i++)
      {
            hst_A[i] = (float)( rand() % 10 );
            . . .
      }

// copia de datos
      cudaMemcpy( dev_A, hst_A, N*N*sizeof(float), cudaMemcpyHostToDevice );
      . . .

// dimensiones del kernel
      dim3 Nbloques(1);
      dim3 hilosB(N,N);

// llamada al kernel bidimensional de NxN hilos
      suma_gpu<<<Nbloques,hilosB>>>(dev_A, dev_B, dev_C);
```

```
// recogida de datos
      cudaMemcpy( hst_C, dev_C, N*N*sizeof(float), cudaMemcpyDeviceToHost );

// impresion de resultados
      printf("A:\n");
      . . .
      printf("B:\n");
      . . .
      printf("C:\n");
      for(int i=0;i<N;i++)
      {
            for(int j=0;j<N;j++)
            {
                  printf("%2.0f ",hst_C[j+i*N]);
            }
            printf("\n");
      }

// salida
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      return 0;
}
```

REALIZACIÓN PRÁCTICA

1. Ejecutar un *kernel* que opere sobre una matriz de 16×16 elementos.

2. La operación consiste en sustituir cada elemento de la matriz original por la suma de los elementos adyacentes (figura 6.2).

3. Los elementos situados en los bordes no deben modificarse.

4. La matriz original se debe inicializar en el *host* con valores aleatorios que sean 0 ó 1.

5. Utilizar una función auxiliar del tipo **`__device__`** para calcular el índice lineal a partir de los índices de fila y columna:

```
__device__ int index (int fila, int columna);
```

6. El programa debe mostrar por pantalla:

- Matriz inicial y matriz final.
- Tiempo de ejecución.

7. Incluir en el código del programa llamadas a la función **`check_CUDA_Error()`** para detectar posibles errores.

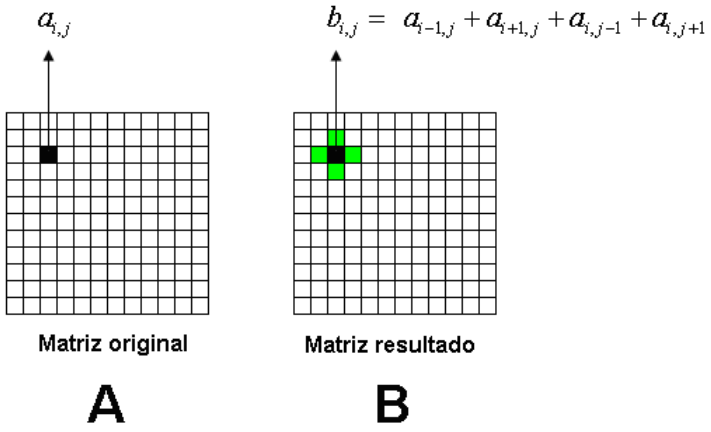


Figura 6.2. Cada elemento de la matriz **B** se obtiene como la suma de los elementos de la matriz **A** ubicados en posiciones adyacentes.

PRÁCTICA nº 7: Memoria Compartida

1. Jerarquía de memoria

Como ya hemos visto en prácticas anteriores, los hilos de CUDA pueden acceder a los datos desde diferentes espacios de memoria durante su ejecución. (Figura 7.1). Sabemos que cada hilo tiene sus propios registros y su propia memoria local. También que cada hilo tiene una memoria compartida que es visible para todos los hilos de su mismo bloque y que todos los hilos tienen acceso a la misma memoria global.

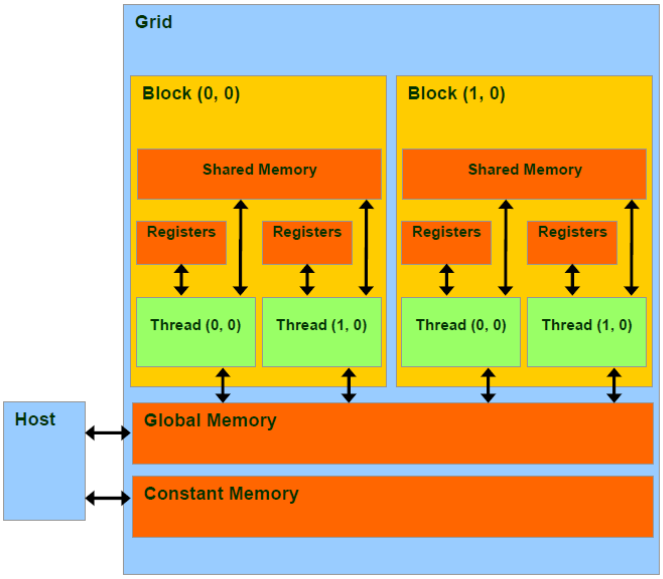


Figura 7.1. Diferentes espacios de memoria en CUDA.

La característica más importante de la **memoria compartida** (*shared memory*) es que ésta se encuentra cerca de cada uno de los núcleos de procesamiento (semejante a una memoria cache de nivel 1), lo que hace que sea una memoria con muy baja latencia. Debido a que está en el mismo chip que el procesador, la memoria compartida es mucho más rápida que los espacios de memoria local y global. Por ello, cualquier oportunidad de reemplazar los accesos a memoria global por accesos a memoria compartida debe ser aprovechada al máximo. En este sentido, lo más habitual será utilizar esta zona de memoria para realizar

cálculos intermedios, para colocar datos que son frecuentemente accedidos, o bien para leer o escribir resultados que otros hilos del mismo bloque necesitan para su ejecución.

Desde el punto de vista del hardware, para alcanzar este elevado ancho de banda, la memoria compartida se divide en módulos de memoria de igual tamaño denominados "bancos", los cuales pueden ser accedidos de forma simultánea. Esto quiere decir que cualquier acceso a *n* direcciones de memoria que correspondan a *n* bancos diferentes pueden ser atendidos de manera simultánea, resultando en un ancho de banda que es *n* veces el correspondiente a un solo módulo. El problema está cuando dos accesos coinciden con el mismo banco, en cuyo caso los accesos de realizan de forma secuencial y el ancho de banda se ve seriamente reducido.

El hecho de que sólo los hilos pertenecientes al mismo bloque puedan compartir este espacio de memoria es otro parámetro de decisión a la hora de lanzar un *kernel* y organizarlo en diferentes bloques e hilos, además de las conocidas limitaciones del hardware.

La sintaxis para reservar memoria en este espacio se realiza a través del identificador **__shared__** dentro del código de la función **__global__** (el *kernel*). Por ejemplo, podemos reservar espacio para un array de 32 elementos de tipo `float` de la siguiente forma:

```
__shared__ float vector[32];
```

Esto crea un espacio de almacenamiento para cada uno de los bloques que hemos lanzado sobre la GPU, pero con la característica de que los hilos de un bloque no pueden ver ni modificar los datos de otros bloques, lo que constituye un excelente medio por el cual los hilos de un mismo bloque pueden comunicarse y colaborar en la realización de un cálculo.

2. Sincronización

El gran potencial de cálculo que nos proporciona CUDA al dividir una tarea en cientos o miles de hilos se basa en la posibilidad de que todos los hilos se puedan ejecutar de manera independiente y simultánea (o casi) actuando sobre sus propios datos. Sin embargo, dado que los hilos son independientes, si esperamos que los hilos de un mismo bloque puedan cooperar compartiendo datos a través de alguna zona de memoria, también necesitamos algún mecanismo de sincronización entre ellos, es decir, necesitamos sincronizar su ejecución para coordinar los accesos a memoria. Por ejemplo, si un hilo *A* escribe un valor en una zona de memoria compartida y queremos que otro hilo *B* haga algo con ese valor, no podemos hacer que el hilo *B* comience su trabajo hasta que no sepamos que la escritura del hilo *A* ha

terminado. Sin una sincronización, tenemos un riesgo de datos del tipo **RAW** (*Read After Write*) donde la corrección del resultado depende de aspectos no deterministas del hardware.

Para evitar este problema, en CUDA podemos especificar puntos de sincronización dentro del código del *kernel* mediante llamadas a la función:

```
__syncthreads();
```

Esta función actúa como barrera en la que todos los hilos de un mismo bloque deben esperar antes de poder continuar con su ejecución. Con esta llamada garantizamos que todos los hilos del bloque han completado las instrucciones precedentes a **`__syncthreads()`** antes de que el hardware lance la siguiente instrucción para cualquier otro hilo. De este modo sabemos que cuando un hilo ejecuta la primera instrucción posterior a **`__syncthreads()`**, todos los demás hilos también han terminado de ejecutar sus instrucciones hasta ese punto.

3. Ejemplo: reducción paralela

Una aplicación donde se pone de manifiesto la necesidad de sincronizar la ejecución de los hilos es la conocida como "*reducción paralela*". En general, un algoritmo de reducción es aquel donde el tamaño del vector de salida es más pequeño que el vector de entrada (se ha reducido). En nuestro caso, la aplicación que vamos a estudiar consiste en la suma de los componentes de un vector donde todos los hilos colaboran para realizar la suma. El punto clave estará en la sincronización de los hilos para que los resultados intermedios sean correctos.

La forma secuencial de realizar esta suma sería iterar a lo largo del vector e ir sumando de forma acumulativa cada elemento. Sin embargo, en nuestro ejemplo vamos a aprovechar la presencia de varios hilos de ejecución para repartir el trabajo. Para implementar nuestro algoritmo de reducción paralela necesitamos que la longitud n del vector sea una potencia de 2. Si no fuera así, bastaría con completar el vector con ceros hasta alcanzar la longitud deseada. El número de pasos que necesitamos para implementar este algoritmo es de $log_2(n)$, y en cada uno de ellos tenemos que hacer que cada hilo sume los valores correspondientes a su posición y su posición más la mitad (de ahí la necesidad de que n sea una potencia de 2), guardando los resultados parciales en sus respectivas posiciones de memoria tal como se muestra en la Figura 7.2. El vector de datos estará almacenado en una zona de memoria compartida por todos los hilos y en cada paso los hilos que trabajan son la mitad que en el paso anterior. Vemos que en cada paso es muy importante sincronizar la ejecución de todos

los hilos ya que es necesario que cada hilo haya realizado su suma antes de continuar con la suma siguiente. Al final de este proceso, sólo uno de los hilos realizará la última suma, el primero, dejando el resultado final en la primera posición de memoria del vector original.

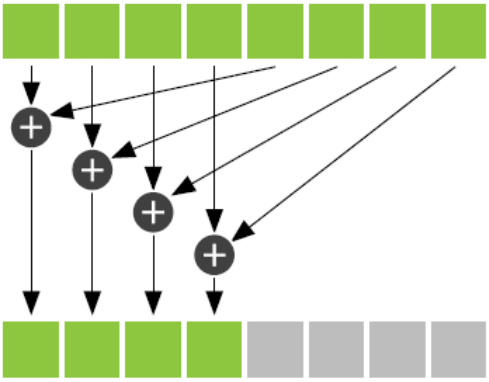


Figura 7.2. Primer paso en el algoritmo de reducción paralela.

El código del *kernel* que implementa esta reducción paralela es el siguiente:

```
// . . .
// defines
#define N 16
// kernel
__global__ void reduccion(float *vector, float *suma )
{
// Reserva de espacio en la zona de memoria compartida
      __shared__ float temporal[N];

// Indice local de cada hilo -> kernel con un solo bloque
      int myID = threadIdx.x;

// Copiamos en 'temporal' el vector y sincronizamos
      temporal[myID] = vector[myID];
      __syncthreads();

//Reduccion paralela
      int salto = N/2;

// Realizamos log2(N) iteraciones
      while(salto)
      {
      // Solo trabajan la mitad de los hilos
            if(myID < salto)
            {
                  temporal[myID] = temporal[myID] + temporal[myID+salto];
            }
            __syncthreads();
            salto = salto/2;
      }

// El hilo no.'0' escribe el resultado final en la memoria global
      if(myID==0)
      {
            *suma = temporal[myID];
      }
}
// . . .
```

REALIZACIÓN PRÁCTICA

1. Calcular de forma aproximada el valor del número π utilizando la siguiente relación entre el número π y la serie formada por los inversos de los cuadrados de los números naturales:

$$\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots + \frac{1}{n^2} + \cdots$$

2. Cada hilo de ejecución del *kernel* debe generar un término de la expresión anterior.

3. El programa debe mostrar por pantalla:

- Cantidad de memoria compartida disponible en KiB.
- Máximo número de hilos disponibles.
- Número de hilos lanzados.
- Valor aproximado del número π.
- Tiempo de ejecución.

4. Incluir en el código del programa llamadas a la función **`check_CUDA_Error()`** para detectar posibles errores.

PRÁCTICA nº 8: Memoria Constante

1. Jerarquía de memoria

La característica primordial de una GPU actual es su enorme potencia de cálculo (determinada en última instancia por su capacidad de cómputo). De hecho, esta superioridad desde el punto de vista computacional de las GPUs sobre las CPUs ha ayudado a precipitar el interés en usar los procesadores gráficos para la computación de propósito general. Sin embargo, la presencia de cientos de núcleos de procesamiento en una GPU hace que el cuello de botella no sea el cálculo en sí, sino el ancho de banda de la memoria. Es decir, hay tal cantidad de núcleos de procesamiento que es difícil conseguir que la entrada de datos sea lo suficientemente rápida como para mantener el elevado ritmo de cálculo. Por este motivo, se hace necesario disponer de mecanismos que permitan reducir el tráfico de memoria requerido por un determinado problema.

Con el fin de aliviar el problema del ancho de banda de la memoria, ya hemos visto que los hilos de ejecución de una aplicación CUDA pueden acceder a los datos desde diferentes espacios de memoria. (Figura 8.1).

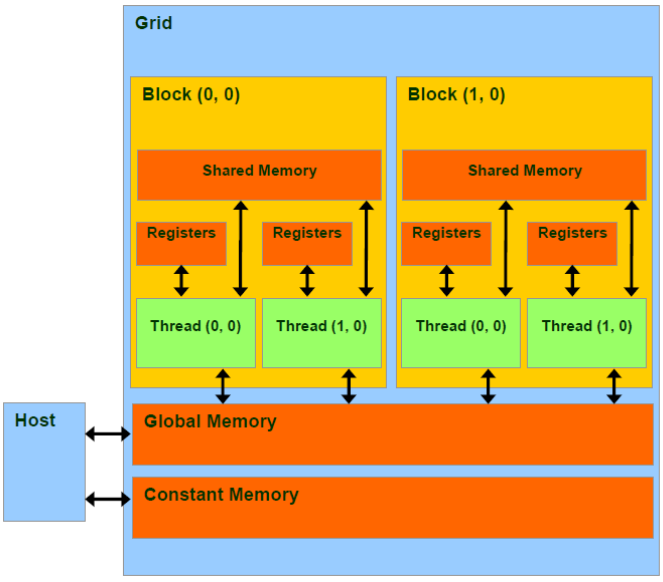


Figura 8.1. Jerarquía de memoria en CUDA.

En esta jerarquía de memoria cada hilo tiene sus propios registros, su propia memoria local y una memoria compartida visible para todos los hilos de su mismo bloque. Por otro lado todos los hilos tienen acceso a la misma memoria global. Sin embargo, dentro de la arquitectura CUDA aún podemos disponer de otro tipo de memoria adicional conocida como **memoria constante** (*Constant Memory*), que también es accesible para todos los hilos pero que es tratada de forma distinta a la memoria global.

2. Utilización de la memoria constante

La memoria constante, como su propio nombre indica, se usa para albergar datos que no cambian durante el transcurso de ejecución de un *kernel*. La capacidad de memoria constante disponible depende de la capacidad de cómputo de la GPU, y su principal ventaja es que en algunas situaciones el uso de memoria constante en lugar de la memoria global pueda reducir considerablemente el ancho de banda de memoria requerido por la aplicación. Dado que no podemos modificar la memoria constante, no podemos utilizarla para dejar los resultados de un cálculo, sino sólo valores de entrada.

El mecanismo para declarar memoria constante es similar al utilizado para declarar memoria compartida. Basta con utilizar el identificador **`__constant__`** delante de la variable. Sin embargo, la declaración de variables constantes debe estar fuera del cuerpo de cualquier función. Por ejemplo, podemos reservar espacio para un array de 32 elementos de tipo `float` escribiendo al comienzo de nuestro código:

```
__constant__ float vector[32];
```

Esta declaración reserva estáticamente espacio en la memoria constante. De este modo no es necesario utilizar la función **`cudaMalloc()`**, pero es necesario decidir en tiempo de compilación el tamaño de las variables constantes.

Para copiar los datos desde el *host* hasta la zona de memoria constante en el *device* es necesario utilizar una función específica. Esta función es **`cudaMemcpyToSymbol()`**:

```
cudaMemcpyToSymbol(void *dst, void *src, size_t count);
```

El primer parámetro (`dst`) corresponde a la dirección de la variable definida en la zona de memoria constante del *device*, el segundo (`src`) es el puntero con la dirección de origen de los datos que queremos copiar desde el *host*, y `count` es el número de bytes que vamos a transferir.

El hecho de declarar memoria como constante restringe su uso a "sólo-lectura". En principio, esta desventaja se ve compensada con el hecho de que se trata de una memoria con un elevado ancho de banda. Una de las razones de este elevado ancho de banda es que la memoria constante tiene asociado un nivel de memoria cache que se encuentra dentro de cada *Streaming Multiprocessors* (**SM**), por lo que lecturas consecutivas sobre la misma dirección no implican ningún tráfico adicional con la memoria. Otra razón del elevado ancho de banda se debe a una particularidad de la arquitectura CUDA y es que las lecturas realizadas por un determinado hilo se copian a otros 15 hilos del mismo *warp*. Lógicamente esto es un beneficio sólo si todos los hilos tienen que acceder a los mismos datos. De no ser así, en vez de ser un beneficio podría llegar a ser incluso contraproducente.

3. Ejemplo

El siguiente ejemplo calcula la traspuesta de una matriz utilizando el espacio de memoria constante para almacenar los datos de la matriz inicial:

```
// includes
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

// defines
#define N 8

// CUDA constants
__constant__ float dev_A[N][N];

// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void traspuesta( float *dev_B)
{
// kernel lanzado con un solo bloque y NxN hilos
      int columna = threadIdx.x;
      int fila    = threadIdx.y;
      int pos     = columna + N*fila;

// cada hilo coloca un elemento de la matriz final
      dev_B[pos] = dev_A[columna][fila];
}

// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
// declaraciones
      float *hst_A, *hst_B;;
      float *dev_B;

// reserva en el host
      hst_A = (float*)malloc(N*N*sizeof(float));
      hst_B = (float*)malloc(N*N*sizeof(float));

// reserva en el device
      cudaMalloc( (void**)&dev_B, N*N*sizeof(float));



// inicializacion
      for(int i=0; i<N*N; i++)
      {
            hst_A[i]= (float)i;
      }
```

```
// copia de datos
      cudaMemcpyToSymbol( dev_A, hst_A, N*N*sizeof(float));

// dimensiones del kernel
      dim3 Nbloques(1);
      dim3 hilosB(N,N);

// llamada al kernel bidimensional de NxN hilos
      traspuesta<<<Nbloques,hilosB>>>(dev_B);

// recogida de datos
      cudaMemcpy( hst_B, dev_B, N*N*sizeof(float), cudaMemcpyDeviceToHost );

// impresion de resultados
      printf("Resultado:\n");
      printf("ORIGINAL:\n");
      for(int i=0; i<N; i++)
      {
            for(int j=0; j<N;j++)
            {
                  printf("%2.0f ",hst_A[j + i*N]);
            }
            printf("\n");
      }
      printf("TRASPUESTA:\n");
      for(int i=0;i<N;i++)
      {
            for(int j=0; j<N; j++)
            {
                  printf("%2.0f ",hst_B[j + i*N]);
            }
            printf("\n");
      }

// salida
      printf("\npulsa INTRO para finalizar...");
      fflush(stdin);
      char tecla = getchar();
      return 0;
}
```

REALIZACIÓN PRÁCTICA

1. Lanzar un *kernel* bidimensional para multiplicar dos matrices **A** y **B** de N×N elementos:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}\ ,\ c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b_{k,j}$$

2. Cada hilo se debe encargar de calcular un elemento $c_{i,j}$ de la matriz final.

3. Las matrices se deben inicializar con valores aleatorios comprendidos entre 0 y 1.

4. Utilizar la memoria constante para almacenar las matrices **A** y **B**.

5. El programa debe mostrar por pantalla:

- Cantidad de memoria constante disponible en KiB.
- Cantidad de memoria constante utilizada.
- Matrices iniciales y matriz final.
- Tiempo de ejecución.

6. Incluir en el código del programa llamadas a la función **`check_CUDA_Error()`** para detectar posibles errores.

PRÁCTICA nº 9: Gráficos en CUDA

1. Imágenes digitales

Acabamos de estudiar que hay situaciones donde puede ser interesante distribuir los hilos de ejecución de un *kernel* de CUDA de una manera análoga al problema que se pretende resolver, como son las operaciones con matrices. A continuación vamos a ver cómo esta situación también se da cuando estamos trabajando con imágenes digitales.

Podemos definir una imagen "real" como una función continua bidimensional $f(x, y)$ que representa el valor de una intensidad, generalmente luminosa, en función de las coordenadas espaciales x e y. Cuando los valores de x, y y f son cantidades discretas y finitas, denominamos a la imagen como imagen "digital". Por lo tanto, si queremos convertir una imagen real en una imagen digital lo que tenemos que hacer es convertir estas cantidades continuas en cantidades discretas. Para ello es necesario realizar dos procesos denominados *muestreo* y *cuantificación* (Figura 9.1):

- El proceso de muestreo implica la discretización de las coordenadas espaciales x e y. El número de muestras tomadas de la imagen determina la resolución espacial, y a cada muestra se le denomina *elemento de imagen* o **píxel**.
- El proceso de cuantificación implica la discretización de la amplitud de la señal f. El número de niveles de intensidad que podemos distinguir por de cada una de las muestras tomadas (también denominados *niveles de gris*) determina la resolución

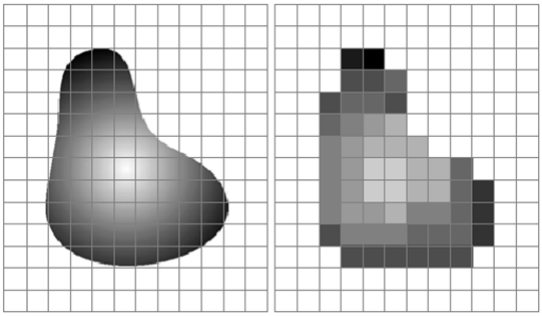


Figura 9.1. Imagen digital obtenida tras un proceso de muestreo y cuantificación.

luminosa y depende del número de bits empleados. Por ejemplo, si utilizamos k bits por cada muestra podremos representar cada una de ellas con 2^k niveles de gris distintos. Por tanto, utilizando 8 bits el número de niveles disponibles es 256.

Vemos que tras el proceso de muestreo y cuantificación de una imagen lo que obtenemos es una matriz de números, es decir, que una imagen digital no es más que la representación de una imagen real a partir de una matriz numérica (Figura 9.2).

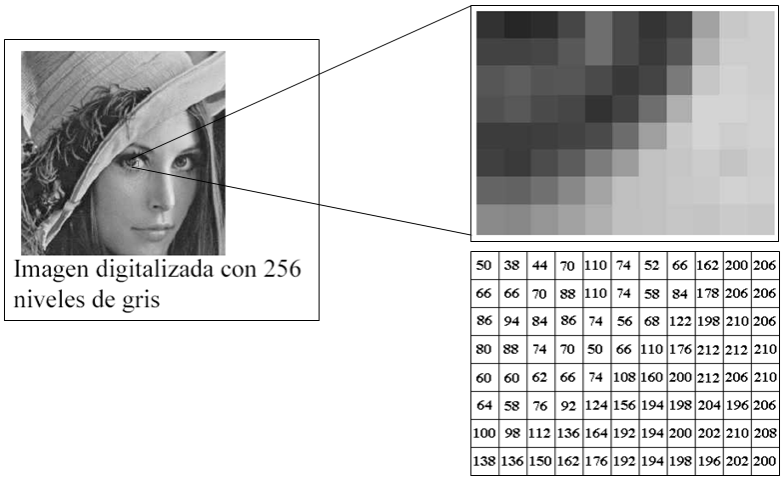


Figura 9.2. Representación de una imagen digital como una matriz de números.

Esto nos hace ver que la forma más adecuada para trabajar con imágenes digitales es utilizando un hardware que nos permita operar simultáneamente con grandes matrices de números. En este contexto, vemos que nuestro entorno de programación CUDA encaja perfectamente con dicho propósito ya que nos permite lanzar un *kernel* bidimimensional con hilos repartidos en ambas direcciones del espacio y haciendo corresponder a cada hilo con un píxel de nuestra imagen digital.

2. Imágenes en color

Desde el punto de vista de la información almacenada en una imagen digital, podemos clasificar las imágenes digitales como:

- Imágenes de intensidad (escala de grises), donde cada elemento de la matriz representa un nivel de gris dentro del rango determinado por el número de bits empleado en la cuantificación (Figura 9.2).
- Imágenes en color, donde para este tipo de imágenes se necesita tener información de tres colores primarios, definidos por el espacio de color que se esté utilizando.

El espacio de color que se utiliza para aplicaciones tales como la adquisición o generación de imágenes en color es el **RGB**, donde la combinación aditiva de los colores primarios rojo (*Red*), verde (*Green*) y azul (*Blue*) produce todo el rango de colores representables en dicho espacio (Figura 9.3).

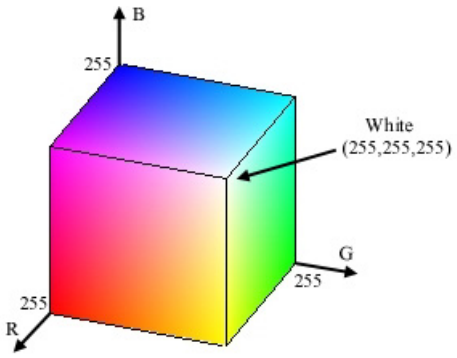


Figura 9.3. Representación del espacio de color **RGB** utilizando valores de 8 bits.

Una imagen en escala de grises también puede considerarse como una imagen en color en la que sus tres componentes son iguales. Así pues, para representar una imagen en color se necesitan como mínimo tres matrices correspondientes a las componentes roja, verde y azul de cada píxel individual (Figura 9.4).

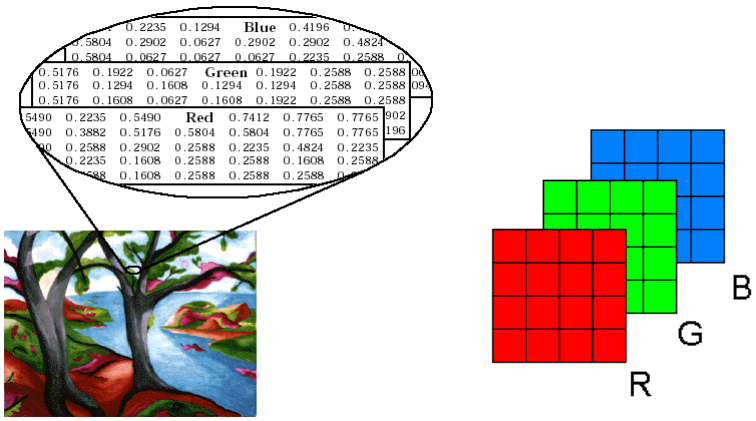


Figura 9.4. Detalle de una imagen en color **RGB**.

3. Imágenes en mapa de bits (bitmap)

Un ***bitmap*** (o *mapa de bits*) es la forma natural de organizar en memoria la información de una imagen digital. Así, dado que una imagen digital se representa como una matriz rectangular de píxeles o puntos de color, en un mapa de bits la información de cada píxel se almacena en posiciones consecutivas de la memoria formando un array cuyo tamaño depende

de la altura y la anchura de la imagen (número de píxeles) y de la información de color contenida en cada píxel (bits por píxel).

La información de color de cada píxel se codifica utilizando canales separados, cada uno de los cuales corresponde a uno de los colores primarios del espacio de color utilizado (generalmente **RGB**). A veces, se puede añadir otro canal que representa la transparencia del color respecto del fondo de la imagen y se denomina canal α (modelo **RGBA**). Por lo tanto, el tamaño necesario para almacenar en memoria un mapa de bits de una imagen en color de $M \times N$ píxeles, con 8 bits por canal y cuatro canales (R, G, B y α) es de $4 \times M \times N$ bytes. Y esta información se almacena en memoria principal ocupando posiciones consecutivas de memoria (Figura 9.5).

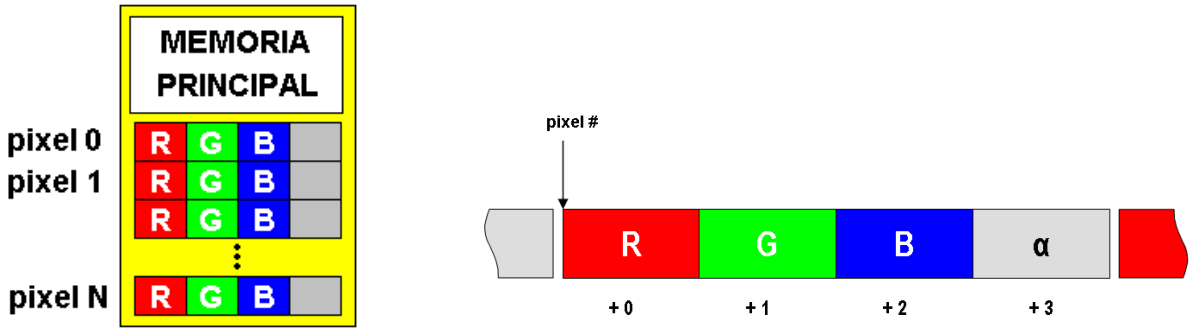


Figura 9.5. Ubicación en memoria principal de un mapa de bits de 4 canales y 8 bits por canal.

4. OpenGL

Con el fin de poder visualizar una imagen digital en nuestro sistema necesitamos funciones que hagan la labor de "abrir una ventana" y de "dibujar" en ella. La forma más sencilla es utilizar una **API** (*Application Programming Interface* - Interfaz de programación de aplicaciones) destinada a tal efecto como es **OpenGL**.

OpenGL (***Open Graphics Library***) es una especificación estándar, es decir, un documento que describe un conjunto de funciones y el comportamiento exacto que deben tener (partiendo de ella, los fabricantes de hardware pueden crear implementaciones). Consta de más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos.

El funcionamiento básico de **OpenGL** consiste en aceptar primitivas tales como puntos, líneas y polígonos, y convertirlas en píxeles. Dado que está basado en procedimientos de bajo nivel, requiere que el programador dicte los pasos exactos necesarios para "renderizar" una

escena. Esto contrasta con otras interfaces descriptivas, donde un programador sólo debe describir la escena y puede dejar que la biblioteca controle los detalles para representarla. El diseño de bajo nivel de **OpenGL** requiere que los programadores conozcan en profundidad la pipeline gráfica, a cambio de darles libertad para implementar algoritmos gráficos novedosos.

5. Ejemplos

El siguiente ejemplo muestra una forma muy sencilla de generar un *bitmap* y de mostrarlo por pantalla. Las llamadas a las funciones de **OpenGL** necesarias para su visualización están encapsuladas dentro del fichero de cabecera **`cpu_bitmap.h`**:

```
// Includes
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "cpu_bitmap.h"
// Defines
#define DIM 1024 // Dimensiones del Bitmap
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void kernel( unsigned char *imagen )
{
// coordenada horizontal
      int x = threadIdx.x + blockIdx.x * blockDim.x;
// coordenada vertical
      int y = threadIdx.y + blockIdx.y * blockDim.y;
// coordenada global de cada pixel
      int pixel = x + y * blockDim.x * gridDim.x;
// cada hilo pinta un pixel con un color arbitrario
      imagen[pixel *4 + 0] = 255*x/( blockDim.x * gridDim.x); // canal R
      imagen[pixel *4 + 1] = 255*y/( blockDim.y * gridDim.y); // canal G
      imagen[pixel *4 + 2] = 2*blockIdx.x + 2*blockIdx.y;     // canal B
      imagen[pixel *4 + 3] = 255;                             // canal alfa
}
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
// declaracion del bitmap
      CPUBitmap bitmap( DIM, DIM );

// tamaño en bytes
      size_t size = bitmap.image_size();

// reserva en el host
      unsigned char *host_bitmap = bitmap.get_ptr();

// reserva en el device
      unsigned char *dev_bitmap;
      cudaMalloc( (void**)&dev_bitmap, size );

// generamos el bitmap
      dim3 Nbloques(DIM/16,DIM/16);
      dim3 hilosB(16,16);
      kernel<<<Nbloques,hilosB>>>( dev_bitmap );

// recogemos el bitmap desde la GPU para visualizarlo
      cudaMemcpy( host_bitmap, dev_bitmap, size, cudaMemcpyDeviceToHost );

// liberacion de recursos
      cudaFree( dev_bitmap );

// visualizacion y salida
      printf("\n...pulsa ESC para finalizar...");
      bitmap.display_and_exit();

      return 0;
}
```

El código anterior dibuja un *bitmap* donde cada hilo se encarga de dar color a un píxel, asignado a cada canal un valor relacionado con su posición espacial. El *bitmap* se dibuja en la pantalla de izquierda a derecha y de abajo a arriba, es decir, el píxel de coordenadas (0, 0) se sitúa en la esquina inferior izquierda.

REALIZACIÓN PRÁCTICA

1. Lanzar un *kernel* que genere el *bitmap* correspondiente al tablero de ajedrez mostrado en la figura 9.6.

2. Utilizar bloques de 16×16 hilos.

3. Incluir en el código del programa llamadas a la función **`check_CUDA_Error()`** para detectar posibles errores.

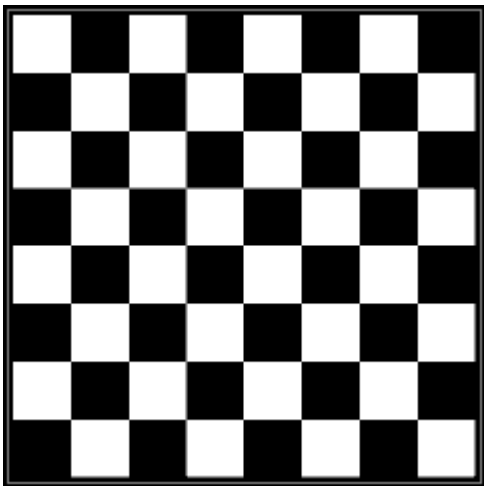


Figura 9.6. Tablero de ajedrez.

PRÁCTICA nº 10:
Ejercicio de aplicación práctica

1. Memoria Compartida

Dentro de la jerarquía de memoria de una GPU con arquitectura CUDA hemos visto que la característica más importante de la memoria compartida (*shared memory*) es que se encuentra dentro de cada uno de los *Streaming Multiprocessors* (**SM**) semejante a una memoria cache de nivel 1, y al estar en el mismo chip que el procesador, es mucho más rápida que los espacios de memoria local y global (figura 10.1). Por ello, cualquier oportunidad de reemplazar los accesos a memoria global por accesos a memoria compartida debe ser aprovechada al máximo. En este sentido, lo más habitual es utilizar esta zona de memoria para realizar cálculos intermedios.

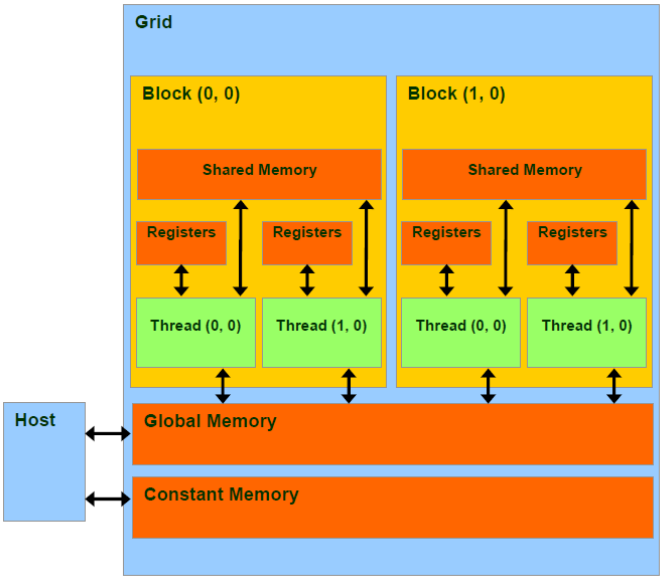


Figura 10.1. Diferentes espacios de memoria en CUDA.

La sintaxis para reservar memoria en este espacio se realiza dentro del cuerpo de una función de tipo **__global__** a través del identificador **__shared__**. Por ejemplo, la siguiente declaración reserva espacio para un array de 32 elementos de tipo `float`:

```
__shared__ float vector[32];
```

Esta declaración crea de forma estática un espacio de almacenamiento en la memoria compartida de cada uno de los bloques lanzados sobre la GPU, pero obliga a conocer en tiempo de compilación el tamaño de las variables. Sin embargo, existe la posibilidad de reservar dinámicamente espacio en la memoria compartida. Esto se consigue añadiendo el identificador **`extern`** en la declaración. En el ejemplo anterior, si hacemos lo siguiente:

```
extern __shared__ float vector[];
```

ahora la variable `vector` sigue residiendo en la zona de memoria compartida pero con el tamaño aún sin especificar. El tamaño definitivo de la variable se resuelve en tiempo de ejecución especificándolo dentro de la llamada al *kernel*, añadiendo un tercer parámetro junto a las dimensiones del *kernel* del siguiente modo:

```
myKernel<<<blocks,threads,SharedMem>>>(arg_1,arg_2,...,arg_n);
```

siendo el valor de *`SharedMem`* el tamaño en bytes asignado a la variable ubicada en memoria compartida.

2. El número π

Una aplicación muy interesante para poner en práctica la potencialidad de la programación paralela mediante una GPU es la obtención del número π a partir de la siguiente expresión:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

La expresión anterior representa la integral definida de una función en el intervalo [0, 1], y su valor es igual al área limitada por la gráfica de la función, el eje de abscisas, y las rectas verticales $x = 0$ y $x = 1$ (Figura 10.2).

Los métodos para evaluar de forma aproximada dicho área reciben el nombre de métodos de integración numérica, y se basan en aproximar la función a integrar por otra función de la cual se conoce la integral exacta (generalmente funciones polinómicas). No obstante, los métodos más sencillos dividen la región cuyo área deseamos calcular en cuadriláteros (rectángulos o trapecios) y después suman el área de todos ellos para estimar el área encerrada bajo la curva (Figura 10.3). Entonces:

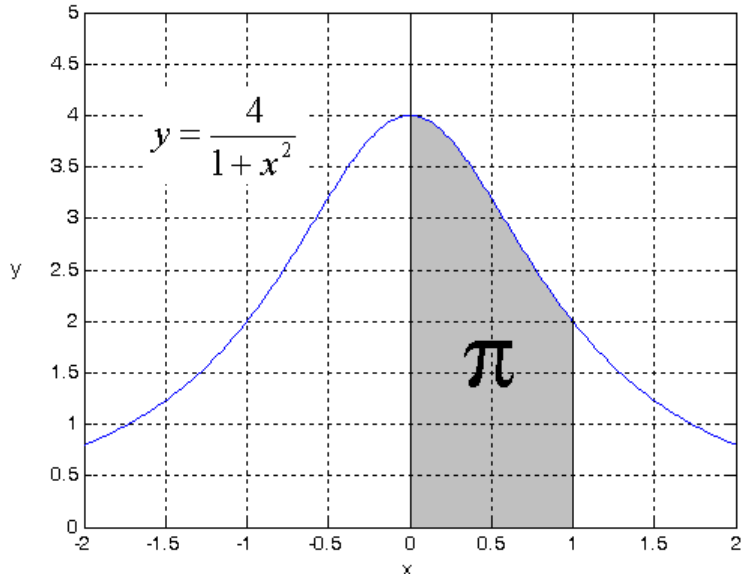


Figura 10.2. Gráfica de la función $y = 4/(1 + x^2)$. El área de la región sombreada es igual a π.

$$\pi \approx A_1 + A_2 + \cdots + A_n$$

donde A_i es el área del cuadrilátero i. De este modo, cuanto mayor sea el número de subintervalos en los que dividimos el intervalo de integración, mayor precisión tendremos en el resultado final.

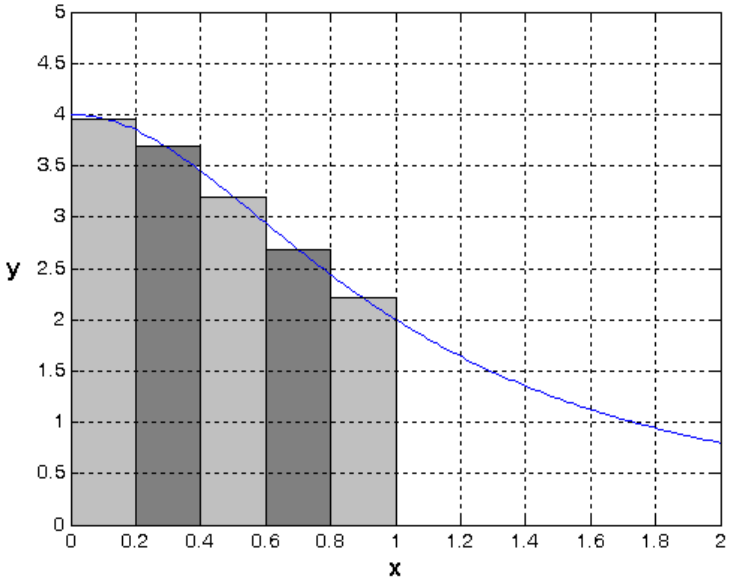


Figura 10.3. Aproximación del área encerrada por la curva mediante rectángulos.

REALIZACIÓN PRÁCTICA

1. Calcular de forma aproximada el valor del número π resolviendo la integral:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

2. El número de hilos lanzados se debe especificar en tiempo de ejecución.

3. Reservar dinámicamente espacio en memoria compartida para almacenar las áreas parciales.

4. Utilizar una función auxiliar de tipo **__device__** para calcular la altura de cada rectángulo.

5. El programa debe mostrar por pantalla:

- Cantidad de memoria compartida disponible.
- Cantidad de memoria compartida utilizada.
- Máximo número de hilos disponibles.
- Número de hilos lanzados.
- Valor aproximado del número π.
- Tiempo de ejecución.

6. Incluir en el código del programa llamadas a la función **check_CUDA_Error()** para detectar posibles errores.