

INTRODUCCIÓN A LA PROGRAMACIÓN EN CUDA

ARQUITECTURA DE COMPUTADORES

Grado en Ingeniería Informática

César Represa Pérez

José María Cámara Nebreda

Pedro Luis Sánchez Ortega

Introducción a la programación en CUDA©2019 v7.0

Área de Tecnología Electrónica

Departamento de Ingeniería Electromecánica

Universidad de Burgos

ÍNDICE

INTRODUCCIÓN: GPU Computing	9
ANEXO: Configuración de CUDA	17
EJEMPLO: Dispositivos CUDA.....	31
BÁSICO 1: Memoria Global	39
BÁSICO 2: Función Kernel.....	45
BÁSICO 3: Hilos y Bloques.....	55
BÁSICO 4: Arrays Multidimensionales	63
BÁSICO 5: Temporización GPU	71
ENTREGABLE 1: Reducción paralela	77
ENTREGABLE 2: Ordenación por rango	81
ENTREGABLE 3: Gráficos en CUDA.....	83
ENTREGABLE 4: Procesamiento de imágenes.....	89
ENTREGABLE 5: Rendimiento GPU vs CPU	95

PREFACIO

Para una determinada arquitectura, incluso asumiendo el 100% de utilización de la **CPU**, existen límites físicos que acotan el crecimiento de las prestaciones de un computador. El modo de incrementar las prestaciones evitando así los límites impuestos por las leyes físicas a la tecnología es introduciendo el paralelismo en el diseño. Existen dos formas de introducir el paralelismo en el diseño:

- De manera transparente al programador, actuando sobre el hardware mediante la réplica de recursos y el solapamiento de instrucciones.
- De manera visible al programador, haciendo que las aplicaciones hagan uso de un hardware conformado por multiprocesadores y multicomputadores.

En este manual de prácticas vamos a abordar el paralelismo de forma explícita, haciendo que el programador sea el responsable de gestionar correctamente los recursos hardware con el fin de aumentar el rendimiento de un programa. En particular vamos a utilizar una tarjeta gráfica o **GPU** (*Graphics Processing Unit*) debido a su inherente arquitectura paralela. Además, vamos a realizar aplicaciones de cálculo de propósito general utilizando un entorno de programación heterogéneo constituido por una CPU y una GPU con la idea de que la parte secuencial del programa se ejecute en la CPU mientras que la parte paralela se ejecute en la GPU. Esta forma de trabajar utilizando las tarjetas gráficas para cálculo de propósito general ha dado lugar a lo que se conoce como *GPU computing*. La idea es utilizar las docenas o cientos de ALUs de una GPU y aprovechar esta cantidad de potencia computacional en los programas que escribimos.

Desde el punto de vista del programador, cualquier GPU actual permite ser utilizada para realizar cómputo de propósito general. Sin embargo, en este manual vamos a utilizar tarjetas desarrolladas por el fabricante **NVIDIA** debido a que fue pionero en modificar la arquitectura de sus GPU para hacerlas asequibles al cálculo científico de propósito general y en proporcionar un entorno de desarrollo para sacar partido a sus GPU desde lenguajes de alto nivel de una manera sencilla. Esta nueva arquitectura introducida por NVIDIA ha sido denominada **CUDA** (*Compute Unified Device Architecture*) y su entorno de desarrollo (también denominado CUDA) consiste en una

serie de APIs desarrolladas en lenguajes de alto nivel como Fortran, C/C++, Python... En este manual utilizaremos el lenguaje **C/C++**.

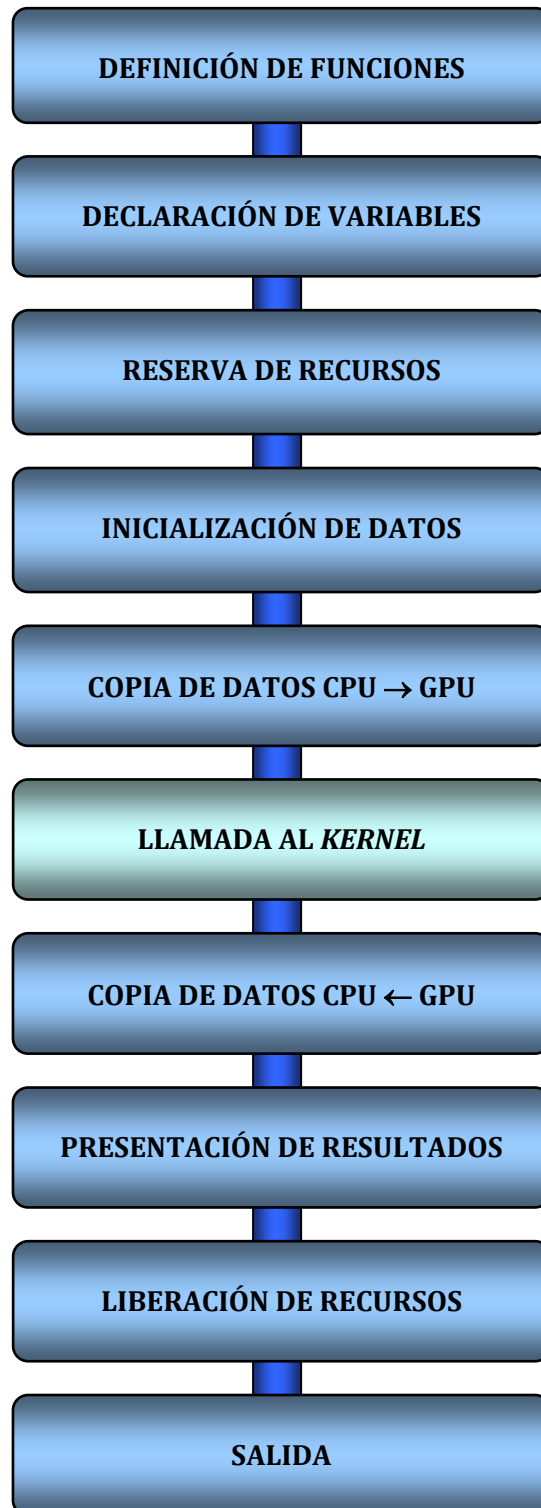
Existen APIs alternativas para desarrollar aplicaciones que utilicen la GPU de cualquier fabricante como acelerador de cálculo. Una de ellas es **OpenCL**, que es un estándar abierto y libre de derechos. Sin embargo, CUDA es la forma más fácil de empezar. De hecho, si fuera necesario utilizar una GPU de otro fabricante que no fuera NVIDIA, el mejor camino sería desarrollar la aplicación en CUDA y posteriormente portar la aplicación a una de las otras APIs. Esto es así porque cualquiera que esté familiarizado con CUDA puede pasar a OpenCL con relativa facilidad, ya que los conceptos fundamentales son bastante similares. Sin embargo, OpenCL es algo más complejo de usar que CUDA ya que gran parte del trabajo que hace la API de CUDA en tiempo de ejecución debe realizarse explícitamente por el programador en OpenCL.

Es importante resaltar que CUDA es adecuado para algoritmos altamente paralelos, en particular que tengan un gran paralelismo a nivel de datos. Con el fin de utilizar de forma eficiente una GPU necesitamos cientos de hilos de ejecución que hagan uso de los cientos de ALUs de la GPU (que en la terminología de NVIDIA reciben el nombre de *Streaming Processors*). Si nuestro algoritmo es fundamentalmente serie, entonces no tiene mucho sentido utilizar CUDA. Es más, muchos algoritmos serie tienen un equivalente paralelo pero otros muchos no.

La principal idea de CUDA es disponer de cientos de hilos ejecutándose simultáneamente en la GPU. Cada uno de estos **hilos** (también denominados *threads*) ejecutan la misma función denominada *kernel*. La idea es que aunque todos los hilos ejecuten la misma función, cada hilo trabaje con datos distintos. Cada hilo tendrá su propio identificador y en base a ese identificador podrá determinar sobre qué conjunto de datos debe trabajar. Cabe destacar que la ubicación de los datos se encuentra en una zona de **memoria compartida** por todos los hilos denominada *memoria global*.

Finalmente es importante recordar que nuestro programa no necesita estar escrito enteramente en CUDA. La mayor parte del código estará escrito en C/C++ y sólo la parte computacionalmente intensa (el *kernel*) estará escrita en CUDA para que se ejecute en la GPU. Esto quiere decir que nuestro programa principal escrito en C/C++ se ejecuta en la CPU y utilizando las funciones de la API de CUDA preparamos la GPU para después poder ejecutar el *kernel* bajo la forma de cientos de hilos.

ESTRUCTURA DE UNA APLICACIÓN BÁSICA EN CUDA



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** (denominada *host*) y la parte más costosa del cálculo se ejecutaría sobre la **GPU** (denominada *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.

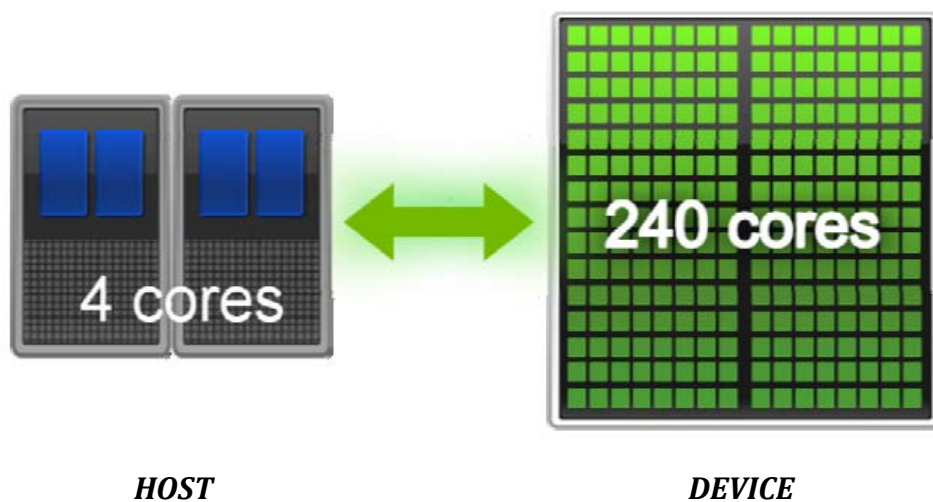


Figura i.1. Modelo de computación en sistemas heterogéneos: **CPU + GPU**.

El principal inconveniente 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**.

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 **GPUs** 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.

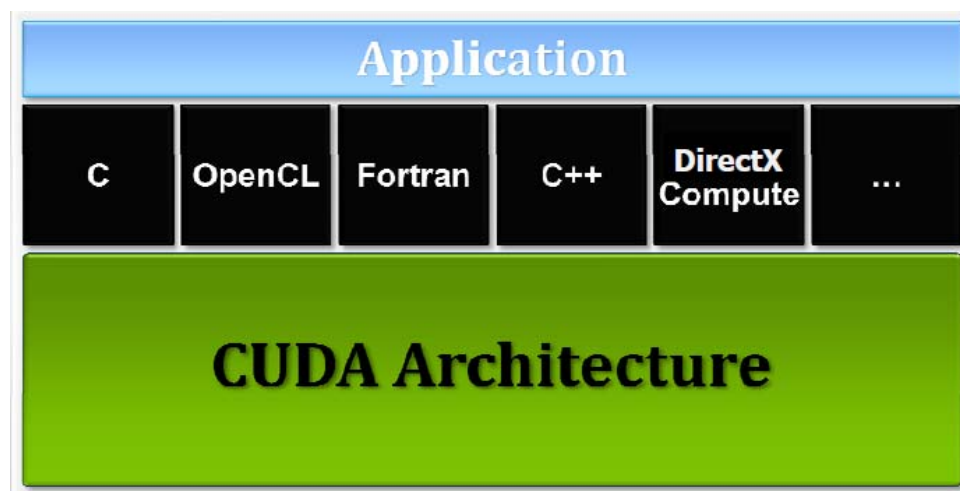


Figura i.2. CUDA está diseñado para soportar diferentes lenguajes de programación.

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.

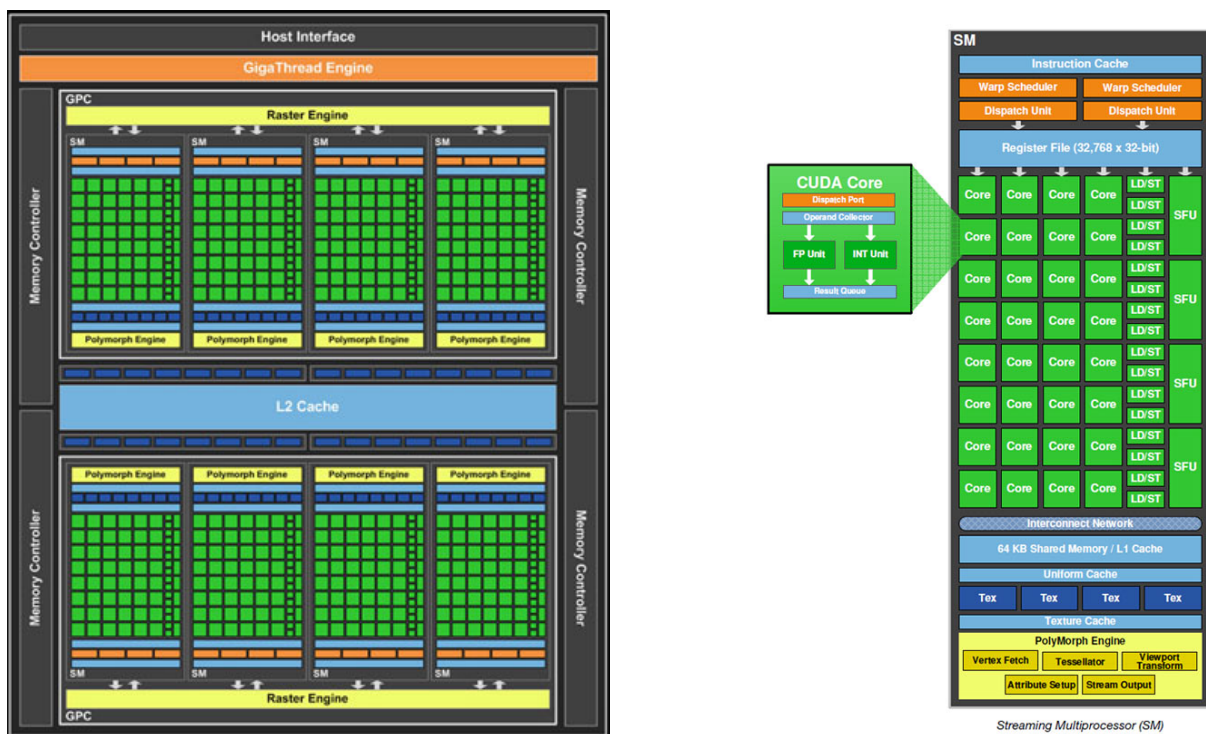


Figura i.3. Arquitectura de una tarjeta gráfica CUDA-enabled.

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)*.

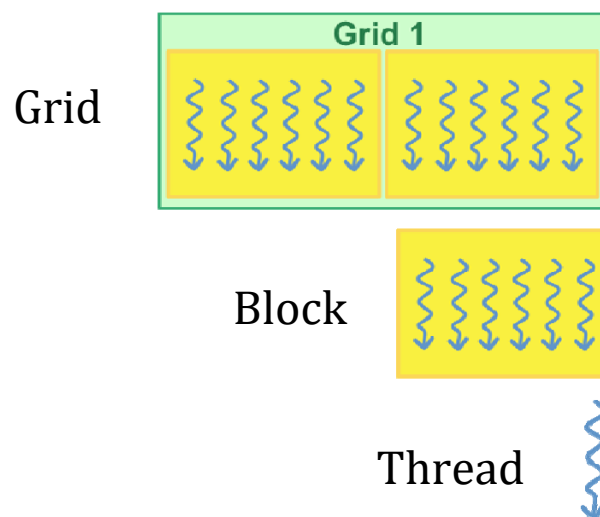


Figura i.4. Jerarquía de hilos en una aplicación CUDA.

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 procesamiento 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 (Figura i.5). 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.

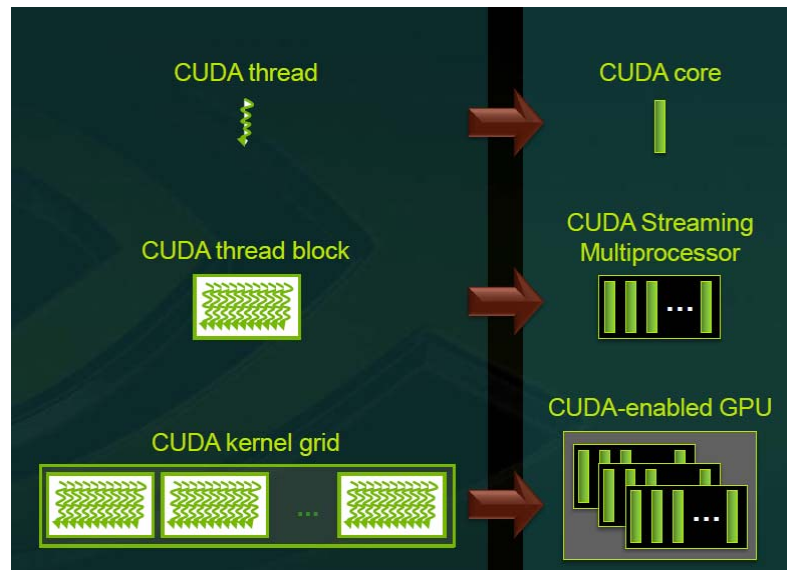


Figura i.5. Asignación de recursos en la ejecución de un *kernel*.

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.

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.6). 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.

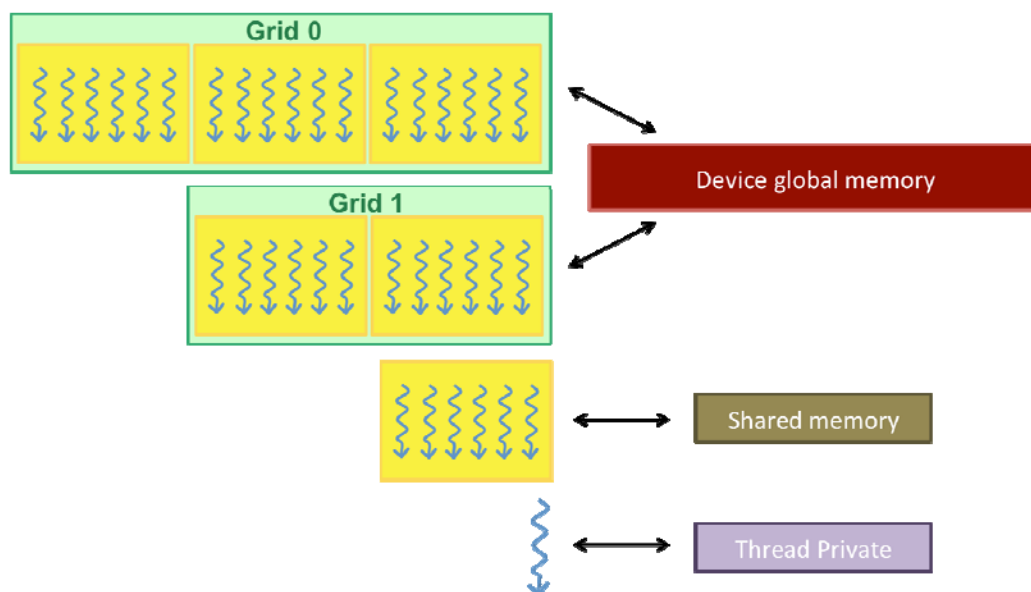


Figura i.6. Jerarquía de memoria dentro de la arquitectura CUDA.

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 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.

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. Por ejemplo, en la **TABLA 1** 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.

En la **TABLA 2** se muestran las especificaciones básicas que posee un dispositivo con una capacidad de cómputo 1.0, que es la primera arquitectura CUDA desarrollada por NVIDIA y que fue denominada con el nombre de “*Tesla*”.

TABLA 1. Núcleos de procesamiento (**SP**) de un dispositivo CUDA por cada multiprocesador (**SM**) en función de su capacidad de cómputo.

Capacidad de Cómputo	Nombre de la Arquitectura	Núcleos por multiprocesador (SP)
1.x	Tesla	8
2.0	Fermi	32
2.1	Fermi	48
3.x	Kepler	192
5.x	Maxwell	128
6.x	Pascal	64
7.x	Volta	64

TABLA 2. 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;



ANEXO: Configuración de CUDA

1. Desarrollo de aplicaciones CUDA

Para programar en la arquitectura CUDA se pueden utilizar diferentes APIs. En este manual vamos a utilizar el lenguaje C estándar, uno de los lenguajes de programación de alto nivel más usado en el mundo. El origen de la arquitectura CUDA y su software correspondiente 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/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.

2. Requisitos del sistema

Para poder ejecutar aplicaciones CUDA se necesitan los siguientes elementos:

- **Tarjeta gráfica NVIDIA** con la característica “[CUDA-enabled](#)”. Dependiendo de la versión de CUDA que vayamos a utilizar necesitaremos una tarjeta con una mínima capacidad de cómputo (ver **TABLA 3**).
- **Driver de dispositivo**. Nunca está de más tener instalado el driver más reciente.
- **Compilador de C/C++**.
- **Software de CUDA** disponible en la página web de [NVIDIA](#).

TABLA 3. Relación entre la versión de CUDA y la capacidad de cómputo requerida.

Versión de CUDA	Capacidad de cómputo soportada
6.5	1.0 – 5.x
7.5	2.0 – 5.x
8.0	2.0 – 6.x
9.0	3.0 – 7.x

3. Instalación de las herramientas de desarrollo

Antes de la instalación del software de CUDA proporcionado por NVIDIA es muy importante tener instalado en nuestro sistema el compilador de C para que el instalador de CUDA encuentre su ubicación. Por otro lado, en los sistemas basados en Windows es recomendable utilizar el IDE de **Microsoft Visual Studio**, ya que el SDK de CUDA proporciona cientos de ejemplos y proyectos ya configurados para ser compilados y ejecutados.

En nuestro caso vamos a describir los diferentes procesos de instalación para los sistemas basados en Windows.

3.1. Instalación de Microsoft Visual Studio

La versión de Visual Studio que necesitamos está relacionada con la versión de CUDA que queramos a instalar, lo que a su vez influye en la tarjeta gráfica necesaria. Generalmente la versión Visual Studio 2010 Express Edition es suficiente, si bien esta versión dejará de ser soportada en futuras versiones de CUDA (ver **TABLA 4**).

TABLA 4. Versiones de Visual Studio soportadas por la versión 9.0 de CUDA.

Compiler	IDE	Native x86_64	Cross (x86_32 on x86_64)
Visual C++ 15.0	Visual Studio 2017	YES	NO
Visual C++ 14.0	Visual Studio 2015	YES	NO
	Visual Studio Community 2015	YES	NO
Visual C++ 12.0	Visual Studio 2013	YES	YES
Visual C++ 11.0	Visual Studio 2012	YES	YES
Visual C++ 10.0 DEPRECATED	Visual Studio 2010	YES	YES

3.2. Instalación de CUDA

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 el IDE de Microsoft Visual Studio.

Desde la versión 5.0 de CUDA el entorno de programación viene integrado en un único archivo instalador. El proceso de instalación es similar en todas las versiones y aunque actualmente nos encontramos ante la versión CUDA 10.1, las figuras mostradas corresponden a los pasos para instalar la versión CUDA 7.5 para sistemas Windows con arquitectura x86_64 versión 10 (esto es, Windows 10 de 64 bits), siendo recomendable seguir los pasos indicados en la guía “*CUDA Quick Start Guide*” y leer las correspondientes “[Release Notes](#)” (Figura a.1).

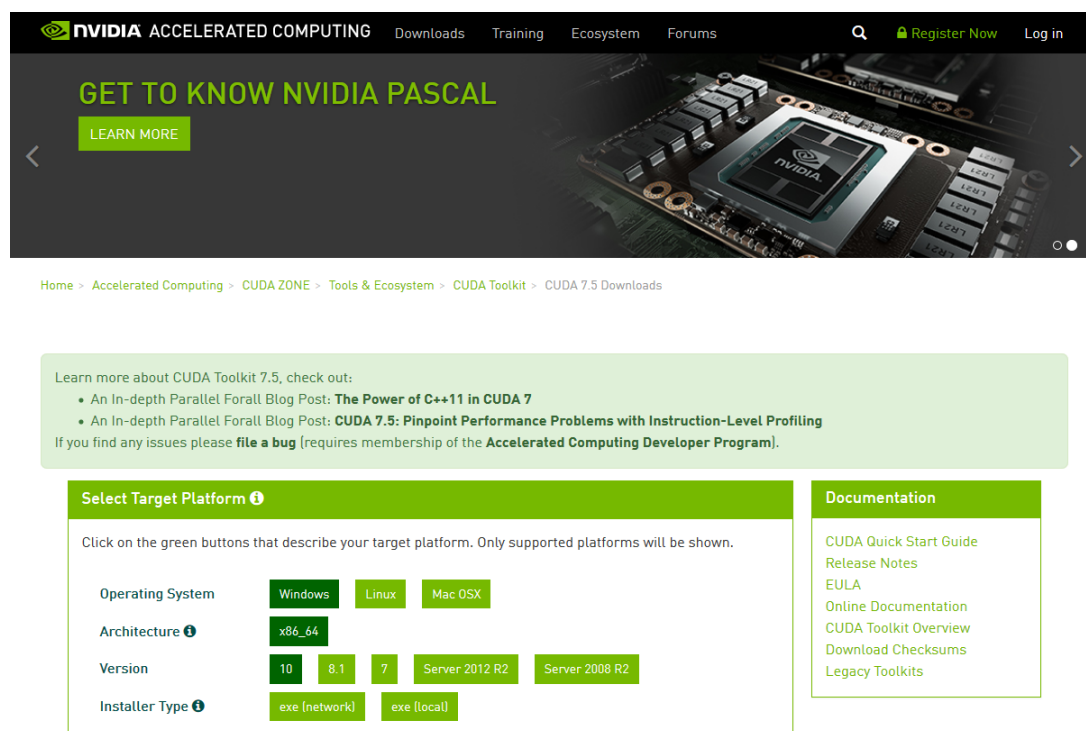


Figura a.1. Descarga del instalador de CUDA v7.5.

Una vez descargado el instalador, lo ejecutamos y seguimos sus instrucciones. En caso de disponer de un driver más actualizado que el incluido en el propio instalador, es recomendable realizar una instalación personalizada y desmarcar la opción correspondiente (Figura a.2).

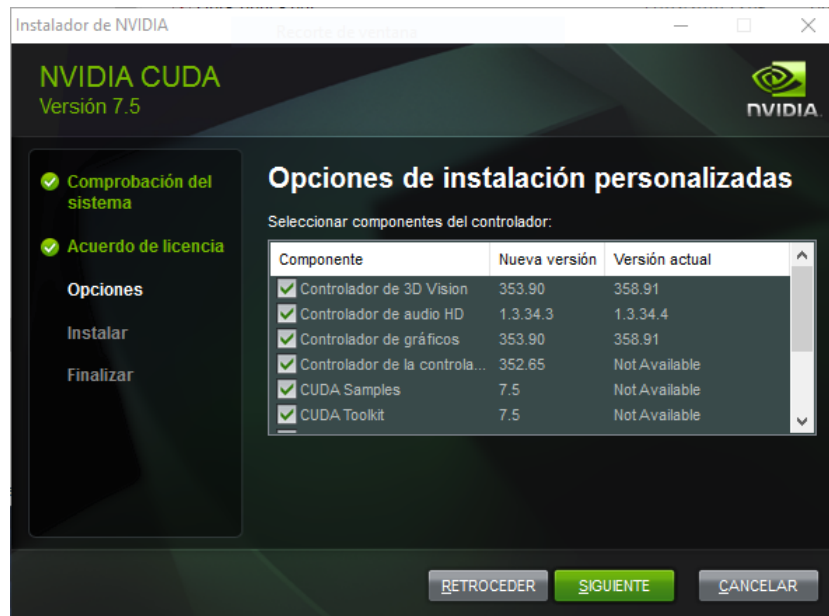


Figura a.2. Ejecución del instalador de CUDA v7.5.

3.3. Ubicación de la instalación de CUDA

Una vez instalado el software podemos comprobar la ubicación de los archivos y 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 Browse CUDA Samples*” (Figura a.3), que nos abre el explorador de archivos directamente en la ruta que estamos buscando (en nuestro caso: `C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.5`).

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

Crear una nueva aplicación CUDA utilizando el IDE de Microsoft Visual Studio es bastante sencillo. Podemos crear un proyecto nuevo o bien utilizar el proyecto que se suministra dentro de los ejemplos como plantilla (“*template*”) y modificarlo a nuestra

conveniencia. Esta segunda opción es más segura y nos permite comprobar que todo está en orden.

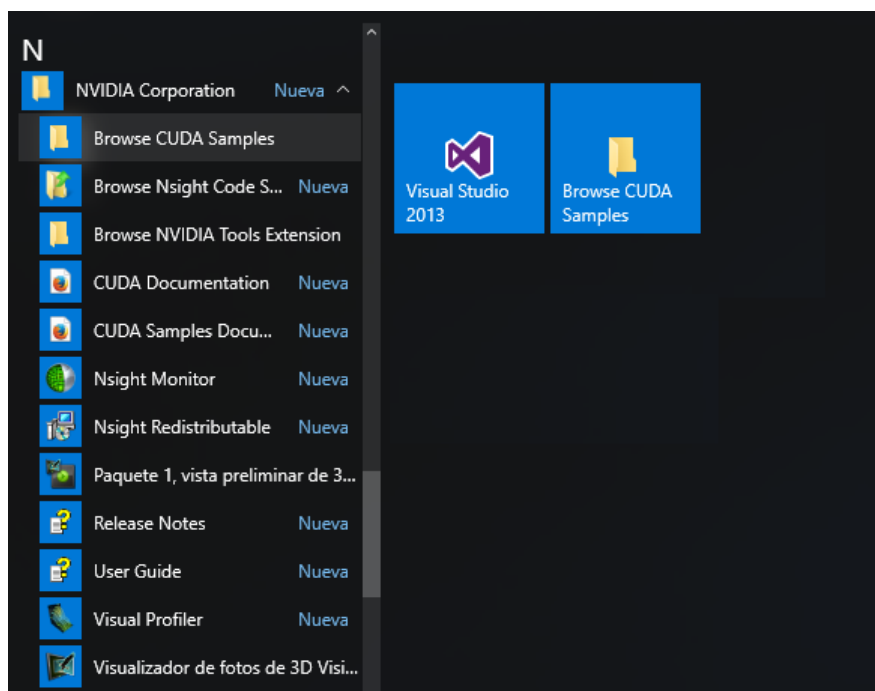


Figura a.3. Explorador NVIDIA Browse CUDA Samples.

4.1. Modificación de un proyecto prototipo

Para utilizar el proyecto suministrado como plantilla (“*template*”) debemos situarnos en la ruta de la que cuelgan todos los ejemplos suministrados con el SDK (`..\CUDA Samples\v7.5\`) y seguir los siguientes pasos:

1. **Crear** una nueva carpeta para colocar en ella nuestros propios proyectos, por ejemplo `..\CUDA Samples\v7.5\practicas`.

Nombre	Fecha de modificación	Tipo	Tamaño
0_Simple	25/04/2016 20:54	Carpeta de archivos	
1_Uutilities	25/04/2016 20:54	Carpeta de archivos	
2_Graphics	25/04/2016 20:54	Carpeta de archivos	
3_Imaging	25/04/2016 20:54	Carpeta de archivos	
4_Finance	25/04/2016 20:54	Carpeta de archivos	
5_Simulations	25/04/2016 20:54	Carpeta de archivos	
6_Advanced	25/04/2016 20:54	Carpeta de archivos	
7_CUDA Libraries	25/04/2016 20:54	Carpeta de archivos	
bin	25/04/2016 20:54	Carpeta de archivos	
common	25/04/2016 20:54	Carpeta de archivos	
practicas	23/04/2016 20:18	Carpeta de archivos	
Samples_vs2010.sln	07/07/2015 15:00	Microsoft Visual Studio Solution	108 KB
Samples_vs2012.sln	07/07/2015 15:00	Microsoft Visual Studio Solution	108 KB
Samples_vs2013.sln	07/07/2015 15:00	Microsoft Visual Studio Solution	108 KB

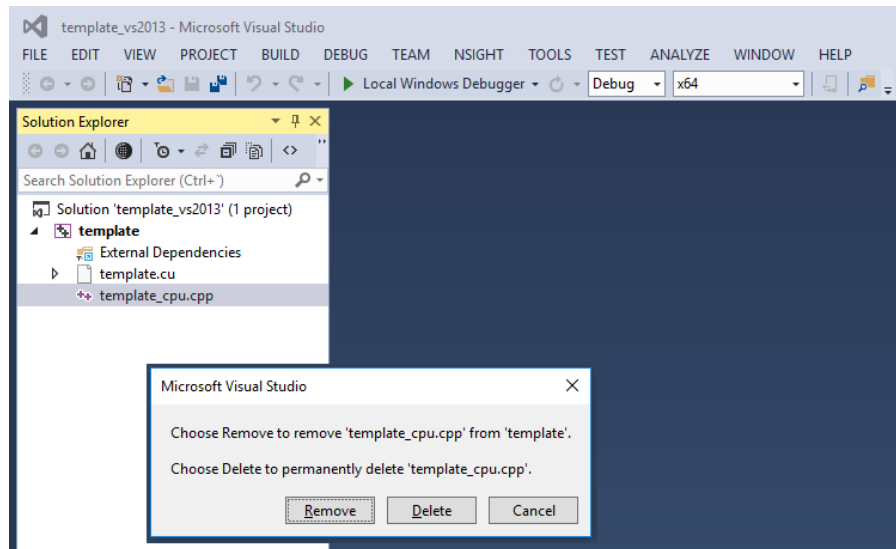
2. **Copiar** la carpeta `..\CUDA Samples\v7.5\0_Simple\template` dentro de nuestra nueva carpeta y renombrarla una vez copiada, como por ejemplo: `..\CUDA Samples\v7.5\practicas\p0`:

ProgramData > NVIDIA Corporation > CUDA Samples > v7.5 > practicas > p0			
Nombre	Fecha de modificación	Tipo	Tamaño
doc	27/04/2016 18:10	Carpeta de archivos	
readme.txt	16/08/2015 14:32	Documento de tex...	1 KB
template.cu	27/05/2015 16:39	Archivo CU	6 KB
template_cpu.cpp	21/03/2015 1:15	C++ Source	2 KB
template_vs2010.sln	16/08/2015 14:32	Microsoft Visual S...	1 KB
template_vs2010.vcxproj	16/08/2015 14:32	VC++ Project	5 KB
template_vs2012.sln	16/08/2015 14:32	Microsoft Visual S...	1 KB
template_vs2012.vcxproj	16/08/2015 14:32	VC++ Project	5 KB
template_vs2013.sln	16/08/2015 14:32	Microsoft Visual S...	1 KB
template_vs2013.vcxproj	16/08/2015 14:32	VC++ Project	5 KB

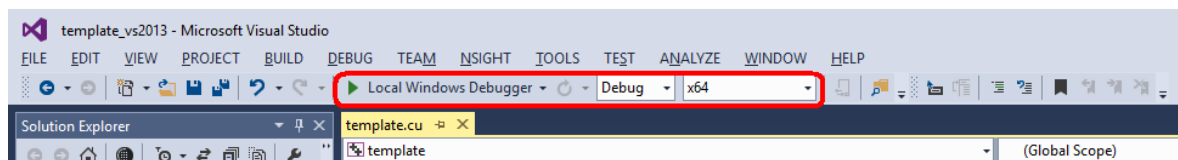
3. **Eliminar** los archivos de proyecto y solución que no correspondan a nuestra versión de Visual Studio (por ejemplo, para la versión 2013 nos quedaremos con los ficheros “template_vs2013.vcxproj” y “template_vs2013.sln”) y mantener el fichero fuente llamado “template.cu”, que es el único que debemos utilizar para escribir nuestro propio código:

ProgramData > NVIDIA Corporation > CUDA Samples > v7.5 > practicas > p0			
Nombre	Fecha de modificación	Tipo	Tamaño
template.cu	27/05/2015 16:39	Archivo CU	6 KB
template_vs2013.sln	16/08/2015 14:32	Microsoft Visual S...	1 KB
template_vs2013.vcxproj	16/08/2015 14:32	VC++ Project	5 KB

4. **Abrir** la solución “template_vs2013.sln” desde el IDE de Microsoft Visual Studio y utilizando el explorador de soluciones **eliminar** de la solución todos los archivos excepto “template.cu”:



5. **Editar** el código fuente de dicho archivo sobrescribiéndolo con el código prototipo del archivo “`practica.cu`” (ver el código al final del anexo).
6. **Construir** y ejecutar la solución para “x64” en modo “*Debug*”:



Si hemos logrado construir y ejecutar el proyecto sin errores aparecerá una ventana con el mensaje “*Hola, mundo!!*” (Figura a.4). Esto quiere 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.

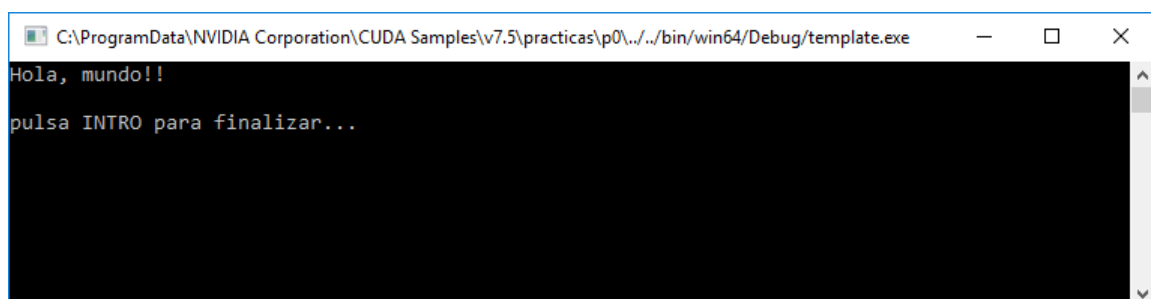


Figura a.4. Salida con el mensaje “*Hola, mundo!!*”.

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:

7. **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”

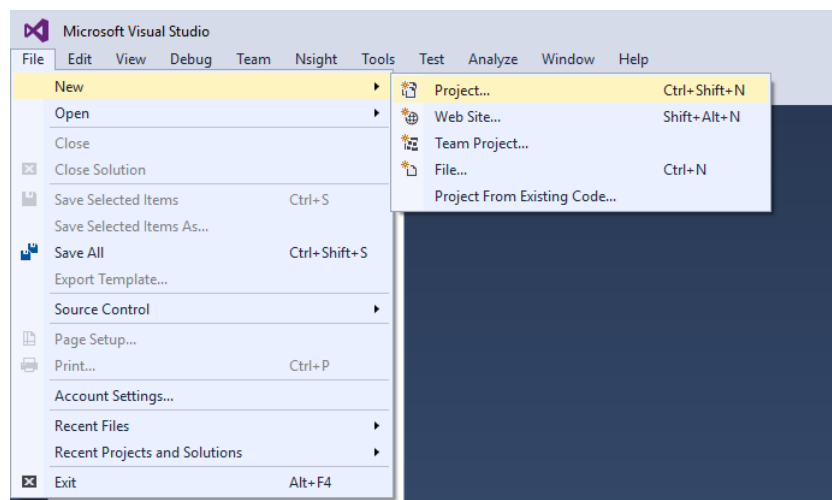
“practica_vs2013.sln”

“practica_vs2013.vcxproj”

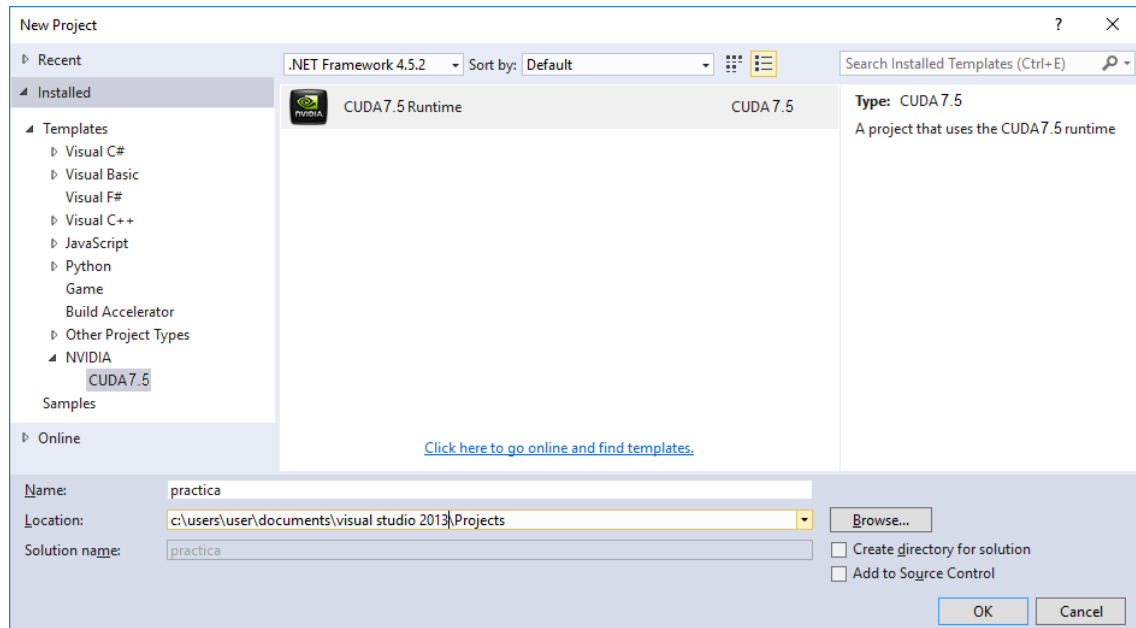
8. **Editar** el contenido de los archivos *.sln y *.vcxproj con un editor de texto y reemplazar todas la ocurrencias de la palabra “*template*” por la palabra escogida anteriormente (en nuestro ejemplo, por la palabra “*practica*”).

4.2. Creación de un proyecto nuevo

En vez de modificar el proyecto prototipo incluido en el SDK de CUDA podemos generar un proyecto nuevo utilizando las opciones del IDE de Microsoft Visual Studio:



Lo más importante es generar un proyecto compatible con CUDA ya que las opciones de compilación son complejas y utiliza sus propias reglas (*CUDA build customization rules*) que suelen estar establecidas en un archivo del tipo `CUDAx.x.props`. En nuestro caso seleccionamos la opción de crear un proyecto para CUDA 7.5 y dejamos que se ubique en la ruta establecida por defecto (...\\documents\\visual studio 2013\\Projects):



Una vez hecho esto, en nuestra carpeta “*practica*” tendremos los tres archivos que constituyen nuestro proyecto:

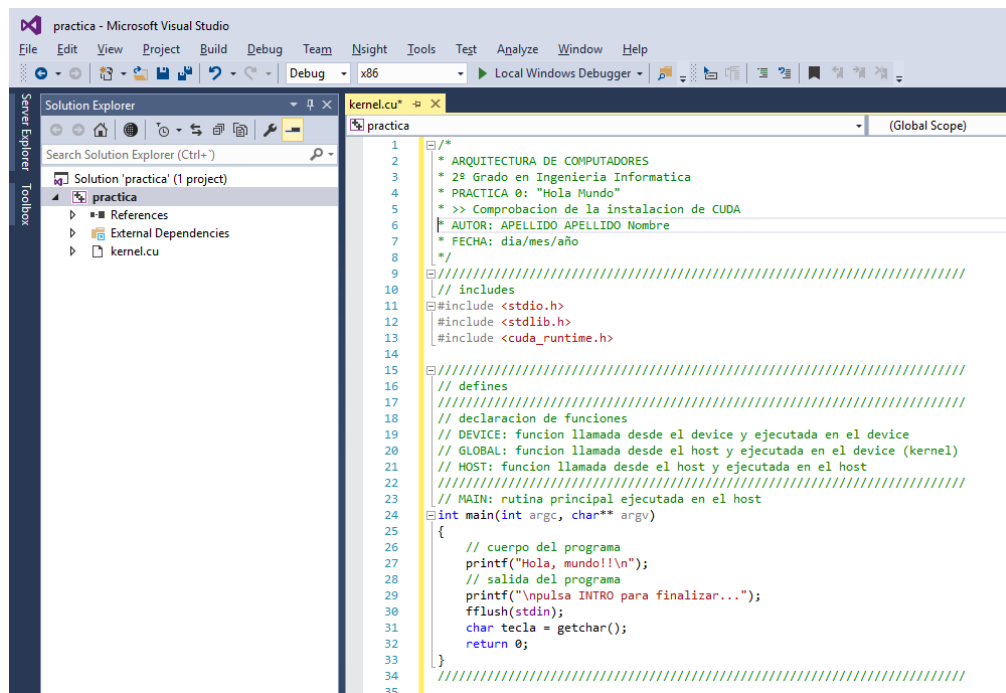
“kernel.cu”

“practica.sln”

“practica.vcxproj”

Nombre	Fecha de modifica...	Tipo	Tamaño
Debug	21/05/2018 22:14	Carpeta de archivos	
kernel.cu	21/05/2018 22:14	Archivo CU	4 KB
practica.sln	21/05/2018 22:14	Microsoft Visual S...	2 KB
practica.VC.db	21/05/2018 22:14	Data Base File	220 KB
practica.vcxproj	21/05/2018 22:14	VC++ Project	9 KB

Ahora podemos editar el código fuente generado por defecto y sustituirlo por el código prototipo del archivo “practica.cu” (ver el código al final del anexo) y posteriormente construir y ejecutar la solución para “x86” en modo “*Debug*”:



5. Programación en CUDA sin GPU

Uno de los requisitos para programar en CUDA es disponer de una tarjeta gráfica NVIDIA. Actualmente esta es una condición imprescindible. Sin embargo, con el fin de acercar la arquitectura CUDA, hacerla asequible a la comunidad científica y, por supuesto, situarse en una posición privilegiada en el mercado de las tarjetas gráficas, en las primeras versiones del entorno de desarrollo esto no era así ya que se incluía un compilador que permitía la opción de emular una GPU. Lógicamente la solución construida de esta forma se ejecutaba en la CPU pero al menos servía para adquirir destrezas en la programación CUDA. Esta opción se mantuvo hasta la **versión 2.3** de CUDA, siendo ésta la última versión que podemos utilizar si queremos programar en CUDA sin disponer de una GPU adecuada.

5.1. Instalación del software

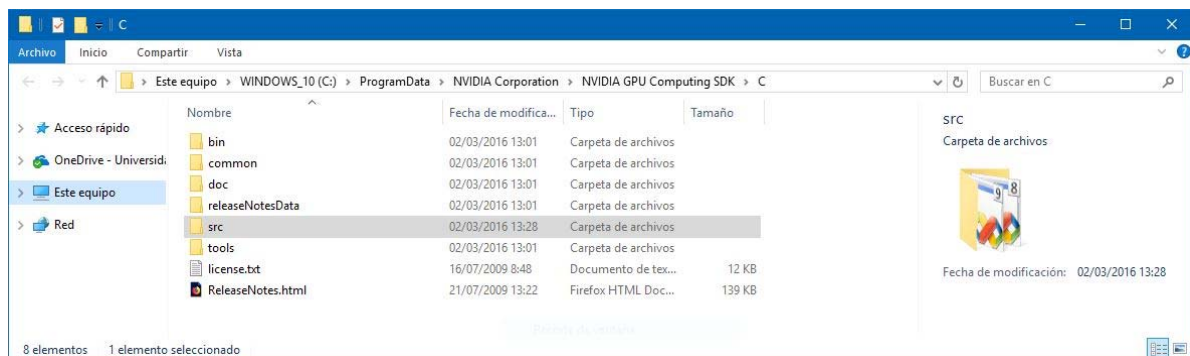
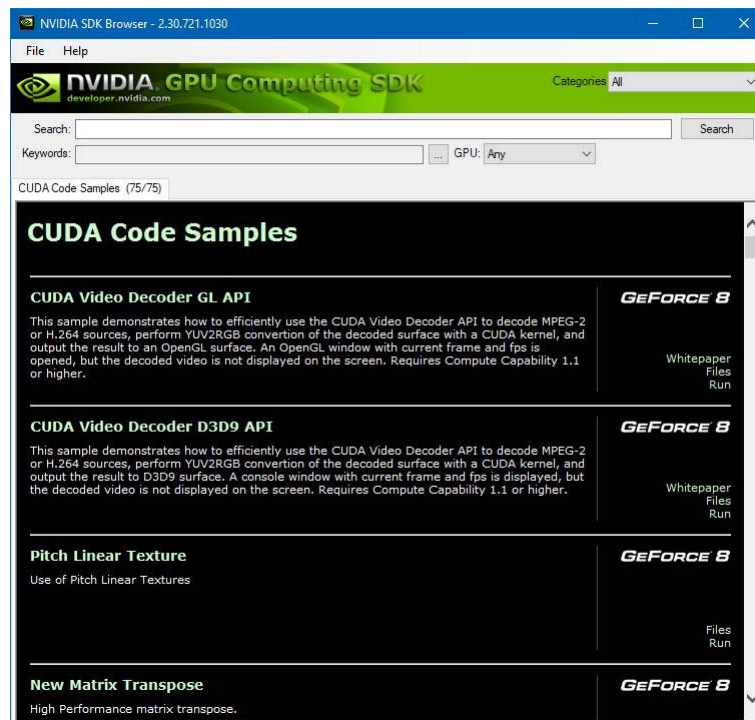
Si estamos interesados en utilizar una versión antigua de CUDA hay que tener en cuenta también que necesitamos disponer de un entorno de desarrollo compatible. Por ejemplo, hasta la versión 2.3 de CUDA se incluye dentro del SDK multitud de ejemplos con proyectos/soluciones configurados para ser utilizados con el IDE de **Microsoft Visual Studio 2008 Express Edition**. Por este motivo, este entorno de desarrollo es lo primero que debemos tener instalado en nuestro sistema.

Por otro lado, para instalar versiones anteriores de CUDA ([Legacy Releases](#)), como por ejemplo la versión 2.3, hay que tener en cuenta que no hay un único archivo instalador sino que necesitamos instalar el **CUDA Toolkit** y el **CUDA SDK** por separado. En este caso, los pasos a seguir para instalar el software de NVIDIA son los siguientes:

1. Desinstalar cualquier versión que hayamos instalado previamente de NVIDIA CUDA Toolkit y NVIDIA CUDA SDK.
2. Instalar el **NVIDIA CUDA Toolkit**.
3. Instalar el **NVIDIA CUDA SDK**.

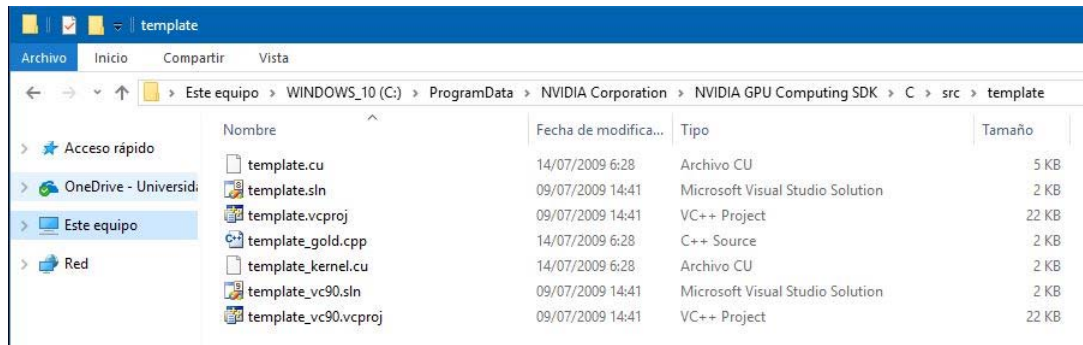
Si hemos decidido instalar una versión antigua es porque el compilador **nvcc** de NVIDIA incluido en el Toolkit admite como opción de compilación un modo de emulación que permite generar código y ejecutarlo sin necesidad de disponer de una tarjeta NVIDIA. En este caso, como es lógico, **no es necesario descargar el driver**. Además, es importante tener en cuenta que es necesario descargar las versiones de 32 bits, ya que los proyectos incluidos en el SDK están configurados únicamente para generar aplicaciones de 32 bits.

Una vez instalado el software de CUDA podemos explorar los programas de ejemplo que vienen incluidos en el SDK. Los archivos de proyecto están localizados en los directorios de cada ejemplo en `C:\...\NVIDIA Corporation\NVIDIA GPU Computing SDK\C\src`, siendo la forma más rápida de llegar a esta ubicación a través del “*NVIDIA GPU Computing SDK Browser*”, que es un explorador que nos proporciona el propio SDK y nos permite llegar a la ruta haciendo click sobre la palabra “*Files*”:



5.2. Construcción de un proyecto en modo simulado

Para construir un proyecto en modo simulado basta con modificar el proyecto "template" que se incluye como plantilla. En este caso debemos seguir los pasos indicados en el apartado 4.1 "Modificación de un proyecto prototipo" pero utilizando los archivos de proyecto y solución específicos de Visual Studio 2008 Express Edition, que son los ficheros "template_vc90.vcproj" y "template_vc90.sln". Una vez abierta la solución desde el IDE de Visual Studio eliminamos todos los archivos excepto el fichero fuente llamado "template.cu", que es el único que debemos utilizar para escribir nuestro propio código:



Tal y como se explica en el apartado 4.1, los ficheros fuente, proyecto y solución se pueden cambiar de nombre 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_vc90.sln”

“practica_vc90.vcproj”

Finalmente, para construir el proyecto y ejecutarlo sin disponer de una tarjeta compatible con CUDA basta con escoger como configuración la opción “*EmuDebug*” o “*EmuRelease*” (Figura a.5).

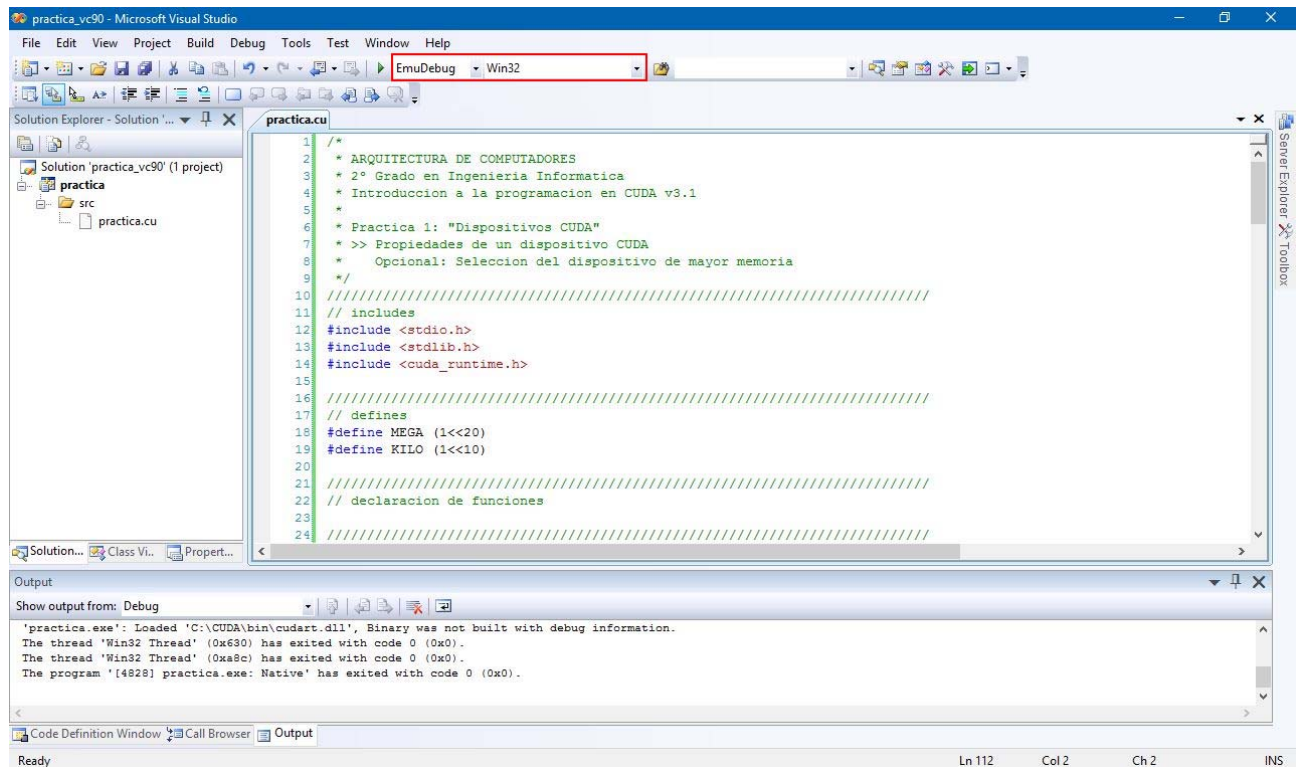


Figura a.5. Construcción de la solución en modo “*EmuDebug*” para Win32.

Archivo “practica.cu”

El siguiente código permite comprobar que el entorno de programación de **CUDA** está en orden y funciona correctamente si al ejecutarlo se muestra por pantalla el mensaje de bienvenida: “¡Hola, mundo!”:

```
/*
 * ARQUITECTURA DE COMPUTADORES
 * 2º Grado en Ingenieria Informatica
 *
 * PRACTICA 0: "Hola Mundo"
 * >> Comprobacion de la instalacion de CUDA
 *
 * AUTOR: APELLIDO APELLIDO Nombre
 */
/////////////////////////////////////////////////////////////////
// includes
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

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

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

// DEVICE: funcion llamada desde el device y ejecutada en el device

// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)

// HOST: funcion llamada desde el host y ejecutada en el host

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

    // salida del programa
    time_t fecha;
    time(&fecha);
    printf("\n*****\n");
    printf("Programa ejecutado el dia: %s\n", ctime(&fecha));
    printf("<pulsa [INTRO] para finalizar>");
    // Esto es necesario para que el IDE no cierre la consola de salida
    getchar();
    return 0;
}
/////////////////////////////////////////////////////////////////
```

EJEMPLO: Dispositivos CUDA

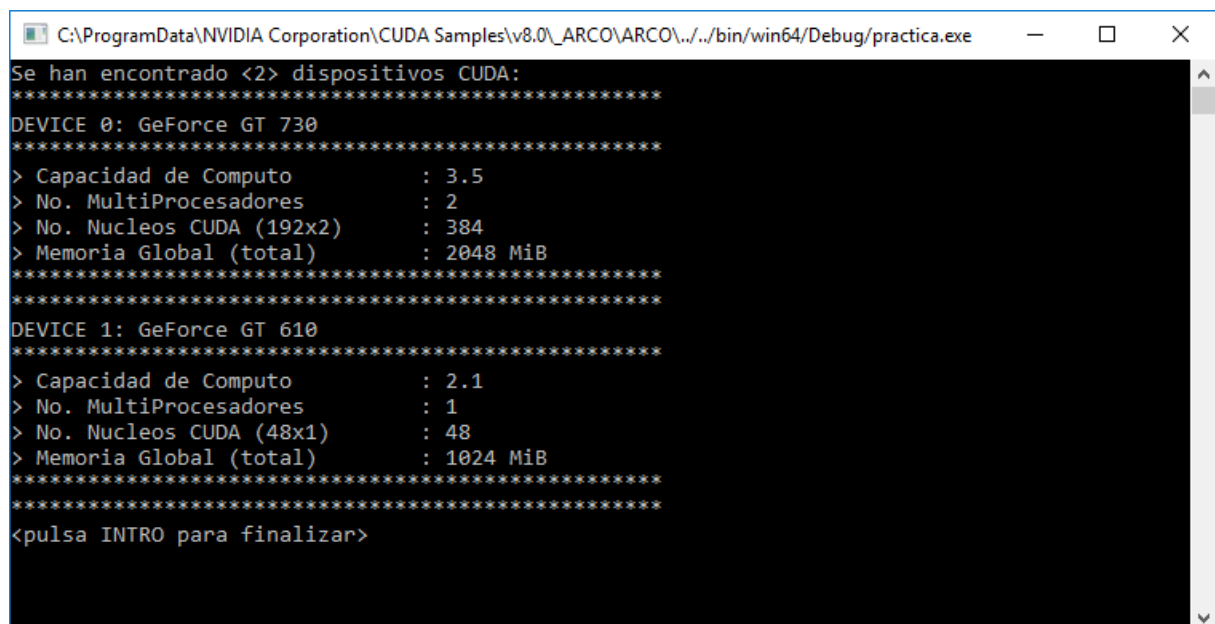
1. Objetivo

Construir una aplicación CUDA que compruebe la presencia de dispositivos CUDA instalados en nuestro sistema.

2. Ejercicio

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 o *Streaming Multiprocessors (SM)*.
- » Número de núcleos de cómputo o *Streaming Processors (SP)*.
- » Memoria global (en MiB).



```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0_ARCO\ARCO\..\bin\win64\Debug\practica.exe
Se han encontrado <2> dispositivos CUDA:
*****
DEVICE 0: GeForce GT 730
*****
> Capacidad de Computo      : 3.5
> No. MultiProcesadores    : 2
> No. Nucleos CUDA (192x2) : 384
> Memoria Global (total)   : 2048 MiB
*****
DEVICE 1: GeForce GT 610
*****
> Capacidad de Computo      : 2.1
> No. MultiProcesadores    : 1
> No. Nucleos CUDA (48x1)  : 48
> Memoria Global (total)   : 1024 MiB
*****
<pulsa INTRO para finalizar>
  
```

Figura e.1. Salida por pantalla de la solución propuesta.

3. Elementos CUDA

Para realizar esta práctica se pueden utilizar las siguientes funciones:

- `cudaGetDeviceCount(int *count)`
Obtiene en la variable *count* el número de dispositivos (tarjetas gráficas) compatibles con CUDA.
- `cudaGetDeviceProperties(cudaDeviceProp *propiedades, int deviceID)`
Obtiene en la variable *propiedades* la estructura de propiedades del dispositivo cuyo identificador es *deviceID*.
- `cudaGetDevice(int *getID)`
Obtiene en la variable *getID* el número identificador del dispositivo seleccionado para lanzar las aplicaciones CUDA.
- `cudaSetDevice(int setID)`
Especifica como dispositivo para lanzar las aplicaciones CUDA el dispositivo cuyo identificador es *setID*.

4. Conceptos teóricos

Además de las características básicas de una tarjeta gráfica con arquitectura CUDA mostradas en la **TABLA 2** existen otras características que dependen de cada dispositivo en particular. Es importante conocer las características reales del dispositivo o dispositivos con los que vamos a trabajar, como por ejemplo cuánta memoria o qué capacidad de cómputo posee, ya que si disponemos de más de un dispositivo resulta muy conveniente poder seleccionar el que mejor se adapte a nuestras necesidades.

Así pues, antes de lanzar cualquier aplicación 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 5**.

A la hora de imprimir por pantalla cualquiera de los campos de dicha estructura hay que fijarse el tipo de dato que contiene. Por ejemplo, los datos que se refieren a tamaños de memoria (o direcciones) suelen ser del tipo `size_t`, el cual corresponde a un tipo `unsigned int` y para imprimirlo puede utilizarse el especificador “%u”, si bien cada compilador suele sugerir el especificador más adecuado.

Por último, si estamos trabajando en un entorno con varios dispositivos instalados, existe un par de funciones que resultan de utilidad a la hora de escoger la GPU más adecuada. Éstas son las funciones `cudaGetDevice()` y `cudaSetDevice()`:

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

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

TABLA 5. Campos contenidos en la estructura `cudaDeviceProp`.

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

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

5. Solución

El siguiente código corresponde a una posible solución del ejercicio propuesto:

```

/*
 * ARQUITECTURA DE COMPUTADORES
 * 2º Grado en Ingeniería Informática
 *
 * EJEMPLO: "Dispositivos CUDA"
 * >> Propiedades de un dispositivo CUDA
 *
 * AUTOR: APELLIDO APELLIDO Nombre
 */
/////////////////////////////////////////////////////////////////
// includes
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

/////////////////////////////////////////////////////////////////
// defines
#define KILO (1<<10)
#define MEGA (1<<20)

/////////////////////////////////////////////////////////////////
// declaracion de funciones
// HOST: funcion llamada desde el host y ejecutada en el host
__host__ void propiedades_Device(int deviceID)
{
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, deviceID);
    // calculo del numero de cores (SP)
    int cudaCores = 0;
    int SM = deviceProp.multiProcessorCount;
    int major = deviceProp.major;
    int minor = deviceProp.minor;
    switch (major)
    {
        case 1:
            //TESLA
            cudaCores = 8;
            break;
        case 2:
            //FERMI
            if (minor == 0)
                cudaCores = 32;
            else
                cudaCores = 48;
            break;
        case 3:
            //KEPLER
            cudaCores = 192;
            break;
        case 5:
            //MAXWELL
            cudaCores = 128;
            break;
        case 6:
            //PASCAL
            cudaCores = 64;
            break;
        case 7:
            //VOLTA
            cudaCores = 64;
            break;
        default:
            //ARQUITECTURA DESCONOCIDA
            cudaCores = 0;
            printf("!!!!dispositivo desconocido!!!!\n");
    }

    // presentacion de propiedades
    printf("*****\n");
    printf("DEVICE %d: %s\n", deviceID, deviceProp.name);
    printf("*****\n");
    printf("> Capacidad de Computo \t: %d.%d\n", major, minor);
    printf("> No. MultiProcesadores \t: %d \n", SM);
    printf("> No. Nucleos CUDA (%dx%d) \t: %d \n", cudaCores, SM, cudaCores*SM);
}

```

```

        printf("> Memoria Global (total) \t: %u MiB\n",
deviceProp.totalGlobalMem/MEGA);
        printf("*****\n");
    }
}
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
    // buscando dispositivos
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    if (deviceCount == 0)
    {
        printf("!!!!No se han encontrado dispositivos CUDA!!!!\n");
        printf("<pulsa [INTRO] para finalizar>");
        getchar();
        return 1;
    }
    else
    {
        printf("Se han encontrado <id> dispositivos CUDA:\n", deviceCount);
        for (int id = 0; id < deviceCount; id++)
        {
            propiedades_Device(id);
        }
    }

    // salida del programa
    time_t fecha;
    time(&fecha);
    printf("*****\n");
    printf("Programa ejecutado el: %s\n", ctime(&fecha));
    printf("<pulsa [INTRO] para finalizar>");
    getchar();
    return 0;
}

```

6. Comentarios

6.1. Prototipos de funciones

Todas las funciones CUDA empleadas a lo largo de este manual se van a describir a partir de su prototipo. En él se especifica el número de argumentos así como su tipo. Por ejemplo, consideremos el prototipo de la función `cudaGetDeviceProperties()`:

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

Vemos que tiene dos argumentos, *propiedades* y *deviceID*. El primero de ellos, *propiedades*, es una referencia (dirección de memoria) a una variable de tipo `cudaDeviceProp`. En general, los argumentos pasados por referencia se suelen utilizar como parámetros de salida. El otro argumento, *deviceID*, es un valor de tipo entero (`int`) y cualquier argumento pasado por valor se utiliza como parámetro de entrada. Así, para un uso correcto de esta función es necesario pasar como argumentos una dirección de memoria (salida) y un número entero (entrada):

```

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

```

```

        cudaDeviceProp propiedades;
        int deviceID = 1;
// ...
        cudaGetDeviceProperties(&propiedades, deviceID);
// ...
        printf("\nDEVICE: %s\n", propiedades.name);
// ...

        return 0;
}

```

Recordemos que las funciones escritas en el lenguaje C permiten devolver un valor al programa principal de dos formas distintas: mediante una sentencia `return` o a través de una referencia (dirección de memoria). En el caso de utilizar una sentencia `return`, la función debe declararse del mismo tipo que al valor devuelto mientras que en el segundo caso la función será de tipo vacío (`void`) y el valor devuelto se almacenará en la dirección de memoria pasada como referencia.

6.2. Ejecución con emulador

Cuando se ejecuta un proyecto construido en modo “*EmuDebug*”, es decir, sin tarjeta gráfica (*device*), el programa se ejecuta en la CPU (*host*). Por tanto, si ejecutamos nuestro ejemplo se mostrarán las propiedades de un dispositivo ficticio (“*Device Emulation*”) cuya capacidad de cómputo es 9999.9999.

```

c:\ProgramData\NVIDIA Corporation\NVIDIA GPU Computing SDK\C\bin\win32\EmuDebug\practica.exe
*****
Se han encontrado <1> dispositivos CUDA:
!!!!dispositivo desconocido!!!!
*****
DEVICE 0: Device Emulation (CPU)
*****
> Compute Capability      : 9999.9999
> No. of Multi Processors : 16
> No. of CUDA Cores (0x16) : 0
> Global Memory (total)   : 4095 MiB
> Shared Memory (per block) : 16 KiB
> Constant Memory (total) : 64 KiB
*****
Dispositivo seleccionado: DEVICE 0
> Nombre : Device Emulation (CPU)
> Memoria: 4095 MiB
*****
pulsa INTRO para finalizar...

```

Figura e.2. Ejecución con emulador: salida por pantalla.

BÁSICO 1: Memoria Global

1. Objetivo

Utilizar la memoria global del *device* para leer y escribir datos.

2. Ejercicio

Transferir un array de datos entre la memoria del *host* y la memoria del *device* de acuerdo con el esquema mostrado en la Figura 1.1:

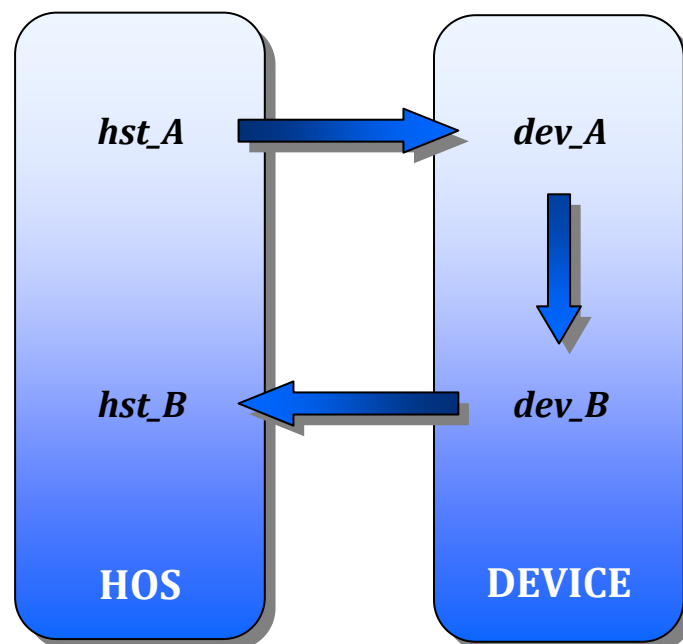


Figura 1.1. Transferencia de datos entre *host* y *device*.

Los arrays `hst_A` y `hst_B` se reservan en la memoria del *host* y los arrays `dev_A` y `dev_B` se reservan en la memoria global del *device*, todos ellos declarados de tipo `float`. Desde el punto de vista del programador es conveniente distinguir entre las variables reservadas en la memoria del *host* y las reservadas en la memoria del *device*. Por ese motivo resulta de gran utilidad añadir un prefijo que permita diferenciales. En nuestro caso hemos utilizado el prefijo `hst_` para las variables del *host* y el prefijo `dev_` para las variables del *device*.

El programa debe mostrar las principales propiedades del dispositivo seleccionado (ver figura 1.2) y además cumplir los siguientes requisitos:

- » Inicializar el array `hst_A` en el *host* con N elementos aleatorios de tipo `float` comprendidos entre 0 y 1.
- » Imprimir por pantalla los arrays `hst_A` y `hst_B` utilizando dos cifras decimales con el fin de comprobar que al terminar las transferencias ambos son iguales.

Los números aleatorios se pueden generar con la función estándar `int rand()`. Esta función genera números aleatorios enteros (tipo `int`) comprendidos entre 0 y un valor máximo determinado por el sistema (dicho valor está definido en una constante llamada `RAND_MAX`). Por otro lado, si queremos generar números aleatorios distintos cada vez que se ejecute el programa es necesario utilizar la función `srand(int semilla)` donde *semilla* debe ser un valor entero distinto en cada ejecución (ver el apartado 5. Ejemplo).

```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\ARCO\ARCO\..\bin\win64\Debug\practica.exe
*****
DEVICE 0: GeForce GT 730
*****
> Capacidad de Computo           : 3.5
> No. de MultiProcesadores       : 2
> No. de CUDA Cores (192x2)     : 384
> Memoria Global (total)        : 2048 MiB
*****
ENTRADA (hst_A):
0.15 0.09 0.76 0.18 0.29 0.24 0.87 0.17 0.45 0.25 0.75 0.69 0.71 0.10 0.26 0.25
SALIDA (hst_B):
0.15 0.09 0.76 0.18 0.29 0.24 0.87 0.17 0.45 0.25 0.75 0.69 0.71 0.10 0.26 0.25
*****
<pulsa INTRO para finalizar>

```

Figura 1.2.: Salida por pantalla de una posible solución.

3. Elementos CUDA

Para realizar esta práctica se pueden utilizar las siguientes funciones:

- `cudaMalloc(void **devPtr, size_t size)`
Reserva dinámicamente espacio en la memoria global del *device*. El valor *size* especifica el tamaño reservado en bytes y la dirección de memoria reservada se almacena en *devPtr*.
- `cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind)`
Lee *count* bytes almacenados en la dirección de memoria dada por *src* y los escribe en la posición de memoria dada por *dst*. El valor *kind* especifica el sentido de la transferencia.
- `cudaFree(void *devPtr)`
Libera el espacio de memoria apuntado por *devPtr*.

4. Conceptos teóricos

4.1. Jerarquía de memoria en CUDA

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 de datos entre la memoria del *host* y del *device*. En la Figura 1.3 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** (*Shared Memory*).

4.2. Memoria Global

La memoria global es el lugar donde la GPU puede leer o escribir datos. Para poder reservar espacio en esta zona de memoria global del *device* y poder acceder a ella desde el

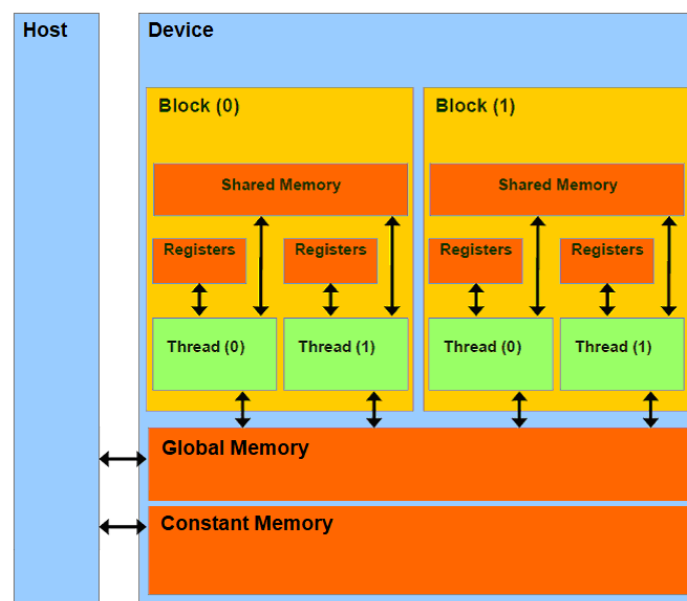


Figura 1.3. Jerarquía de memoria en CUDA.

host se debe utilizar la función `cudaMalloc()`, que tiene un comportamiento similar a la función `malloc()` 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, ya podemos 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 tipo);
```

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 *tipo* es el tipo de transferencia que vamos a realizar. En la **TABLA 6** tenemos las cuatro posibilidades que existen para la variable *tipo* 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 6. Tipos de transferencias de datos en CUDA.

Tipo de transferencia	Sentido de la transferencia
<code>cudaMemcpyHostToHost</code>	host → host
<code>cudaMemcpyHostToDevice</code>	host → device
<code>cudaMemcpyDeviceToHost</code>	device → host
<code>cudaMemcpyDeviceToDevice</code>	device → device

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

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

5. Ejemplo

El siguiente ejemplo sirve para mostrar las diferencias y las similitudes que existen a la hora de reservar memoria tanto en el *host* como en el *device* utilizando las funciones `malloc()` y `cudaMalloc()` respectivamente. En este ejemplo se hace una reserva en ambos espacios de memoria para un vector de *N* elementos de tipo `float`, después se inicializa el vector del *host* con valores aleatorios entre 0 y 1, y finalmente 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*sizeof(float) );

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

    // inicializacion de datos en el host
    srand ( (int)time(NULL) );
    for (int i=0; i<N; i++)
    {
        hst_matriz[i] = (float) rand() / RAND_MAX;
    }

    // visualizacion de datos en el host
    printf("DATOS:\n");
    for (int i=0; i<N; i++)
    {
        printf("A[%i] = %.2f\n", i, hst_matriz[i]);
    }

    // copia de datos CPU -> GPU
    cudaMemcpy(dev_matriz, hst_matriz, N*sizeof(float), cudaMemcpyHostToDevice);

    // salida
    time_t fecha;
    time(&fecha);
    printf("*****\n");
    printf("Programa ejecutado el: %s\n", ctime(&fecha));
    printf("<p>pulsa [INTRO] para finalizar>");
    getchar();
    return 0;
}
```




BÁSICO 2: Función *kernel*

1. Objetivo

Lanzar un *kernel* (función ejecutada en la tarjeta gráfica) de un solo bloque con múltiples hilos o *threads*.

2. Ejercicio

Calcular la suma de dos vectores de tamaño N . El programa debe cumplir los siguientes requisitos:

- » El tamaño N debe estar definido en tiempo de ejecución por el usuario.
- » Uno de los vectores se debe generar en el *host* con N valores aleatorios de tipo `int` comprendidos entre 0 y 9.
- » El segundo vector se debe generar en el *device* y será el inverso del primer vector, es decir, los mismos valores pero en orden inverso.
- » La suma será realizada en el *device* aprovechando todos los hilos o *threads* lanzados sin sobrepasar el máximo permitido.
- » Por pantalla se debe mostrar:
 - Las principales propiedades del dispositivo seleccionado (nombre, capacidad de cómputo, número de multiprocesadores, número de núcleos y tamaño de la memoria global en MiB).
 - Los dos vectores iniciales y el vector con el resultado final.

3. Elementos CUDA

Para realizar esta práctica se puede utilizar cualquiera de las funciones estudiadas anteriormente junto con los siguientes elementos:

- `__global__`
Etiqueta o especificador que se coloca junto a la declaración de la función *kernel* para indicar al compilador que dicha función se va a ejecutar en el *device*.
- `<<<num_bloques,num_hilos>>>`
Sintaxis de llamada a la función *kernel* para indicar el número de copias que se desea lanzar.
- `threadIdx.x`
Variable de tipo `int` que se genera automáticamente dentro de la función *kernel* y que permite identificar cada una de las copias que se han lanzado.

```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\ARCO\ARCO\..\bin\win64\Debug\practica.exe
*****
DEVICE 0: GeForce GT 730
*****
> Capacidad de Computo          : 3.5
> No. de MultiProcesadores      : 2
> No. de CUDA Cores (192x2)    : 384
> Memoria Global (total)       : 2048 MiB
*****
  Introduce el numero de elementos: 2000
> ERROR: numero maximo de hilos superado! [1024 hilos]
  Introduce el numero de elementos: 24
> Vector de 24 elementos
> Lanzamiento con 1 bloque de 24 hilos
VECTOR 1:
 1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 6 1 4 2 3
VECTOR 2:
 3 2 4 1 6 7 2 5 1 1 7 1 5 5 4 2 8 8 4 9 0 4 7 1
SUMA:
 4 9 8 1 15 11 10 13 3 5 12 6 6 12 5 3 13 10 11 15 1 8 9 4
*****
<pulsa [INTRO] para finalizar>

```

Figura 2.1.: Salida por pantalla de una posible solución.

4. Conceptos teóricos

4.1. 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*). Estos últimos 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 dichas aplicaciones aprovechando esa gran cantidad de paralelismo en los datos. Por ello, el

modelo de programación en CUDA 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 2.2. 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.

4.2. Modificaciones respecto a un programa C/C++ estándar

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

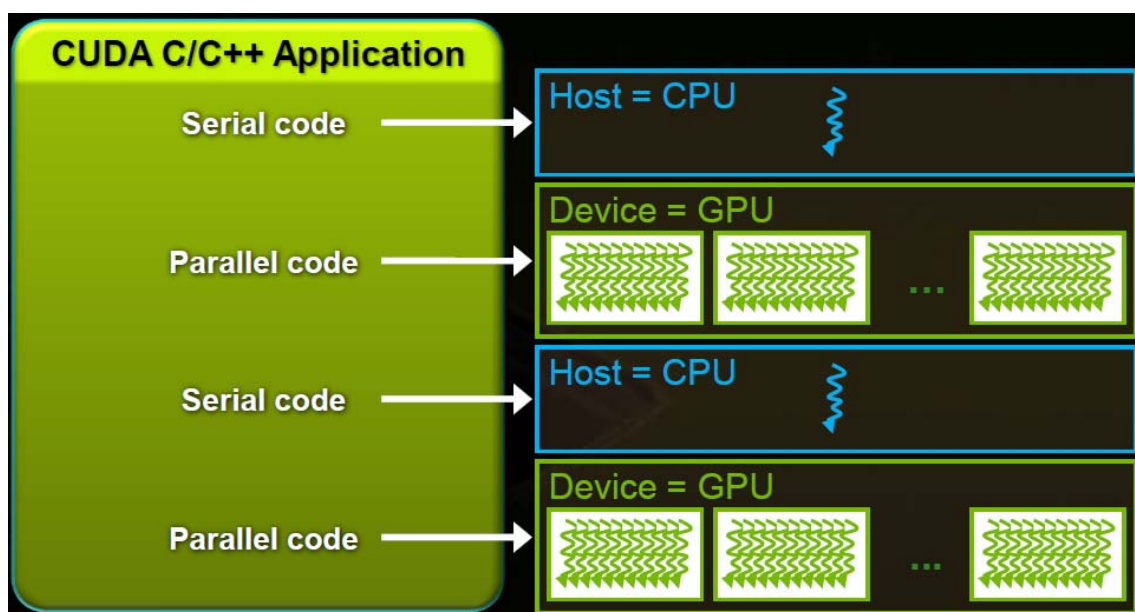


Figura 2.2. Modelo de programación heterogéneo.

TABLA 7. Especificadores de tipo para funciones.

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

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 consisten simplemente en añadir una etiqueta o especificador junto al prototipo de la función. Esto es necesario para que el compilador sepa distinguir qué funciones se deben ejecutar en la CPU y cuáles en la GPU. 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 7**. De acuerdo con esta clasificación, dado que un *kernel* es una función que se invoca desde la CPU (*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)
{
    // . . . Código 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 modificación que se requiere para las funciones ejecutadas en la GPU se refiere a la sintaxis necesaria para ser invocadas. 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.

4.3. Rutina de ejecución de una aplicación CUDA

Según nuestro modelo de programación heterogéneo, el mecanismo de ejecución de una aplicación CUDA involucra tres fases:

- Una primera fase en la que los datos iniciales (si los hay) se copian desde la memoria del *host* hacia la memoria del *device* (Figura 2.3).

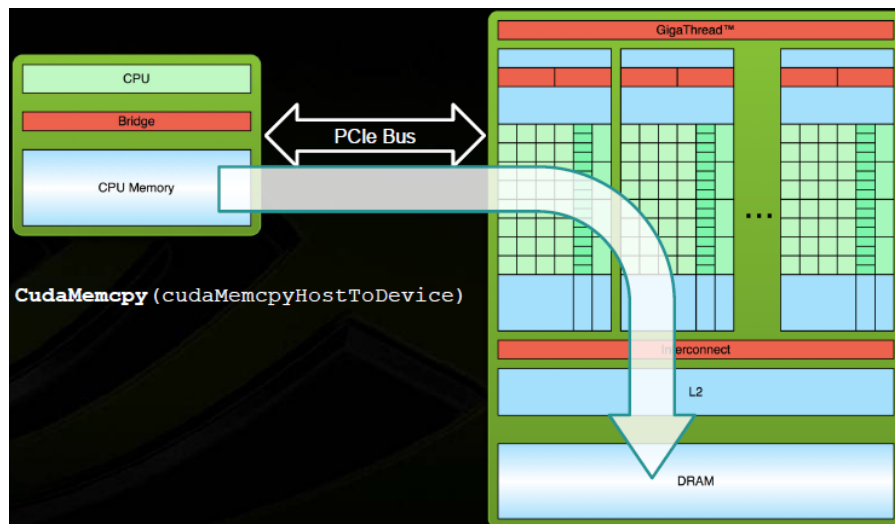
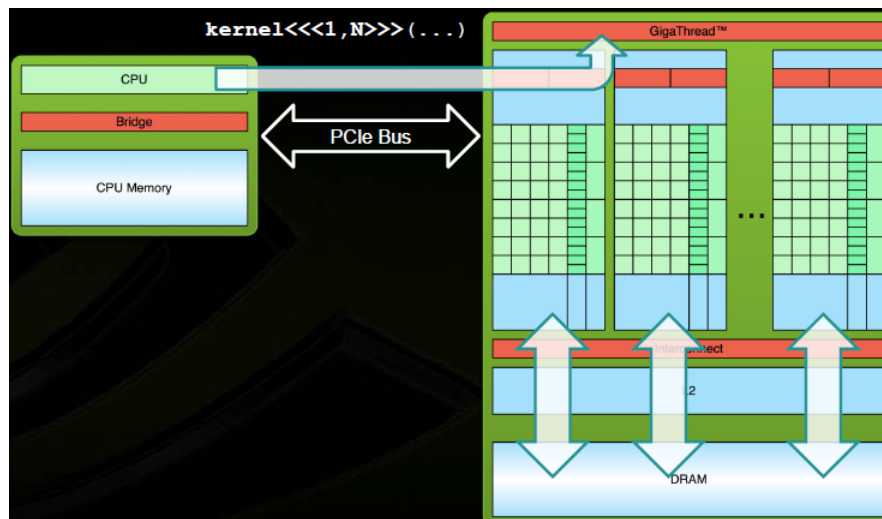
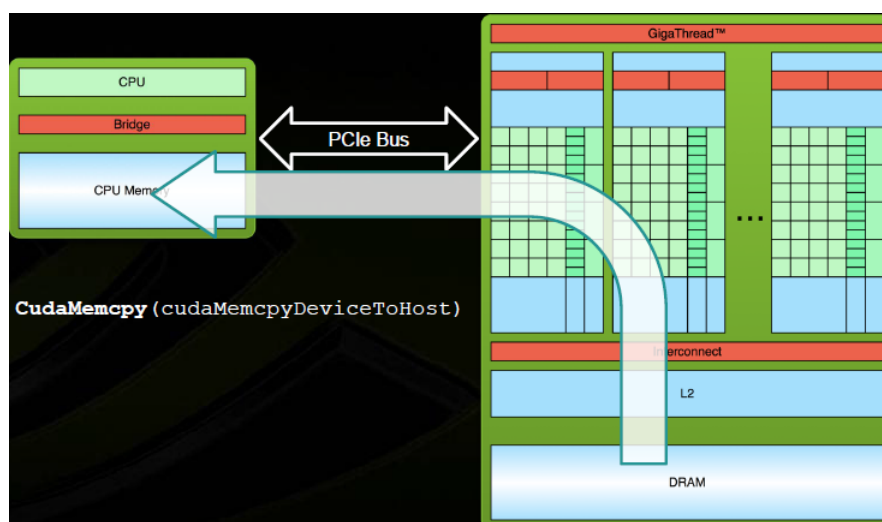


Figura 2.3. Copia de datos desde la memoria del *host* al *device*.

- Una segunda fase en la que se invoca a la función *kernel* que se ejecuta en el *device* empleando múltiples hilos (Figura 2.4).

Figura 2.4. Lanzamiento del *kernel*.

- Y una tercera fase en la que los resultados generados por el *kernel* se copian desde la memoria del *device* a la memoria del *host* (Figura 2.5).

Figura 2.5. Copia de resultados desde la memoria del *device* al *host*.

4.4. Hilos o *threads*

Los hilos o *threads* constituyen la base de la ejecución de aplicaciones en paralelo. Cuando se lanza un *kernel* se crea una malla de hilos donde todos y cada uno de ellos ejecuta el mismo programa. Estos hilos se pueden agrupar en uno o más bloques. Precisamente las variables *blocks* y *threads* antes mencionadas en la sintaxis de llamada al *kernel* especifican el número de bloques lanzados y el número de 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 ejecutaría simultáneamente N copias de nuestro *kernel*, todas ellas agrupadas en mismo bloque. 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, que cada hilo trabaje con 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 denominada `threadIdx` que se genera automáticamente para cada hilo. Esta variable es de tipo `int` y únicamente puede utilizarse dentro del código del *kernel*. Así, una vez lanzado el *kernel* esta variable adquiere un valor distinto para cada hilo:

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

Cada uno de los N hilos lanzados tendrá un valor distinto de `threadIdx.x` que irá desde el 0 hasta el $N-1$. Es decir, cada hilo tendrá un índice distinto que permitirá al programador decidir sobre qué datos debe trabajar cada uno de ellos. Los hilos lanzados se ordenan imaginariamente a lo largo de las tres coordenadas del espacio. Por defecto se considera que los hilos se reparten a lo largo del eje x (Figura 2.6), de ahí que se añada el sufijo “`.x`” para indicar dicha dirección.

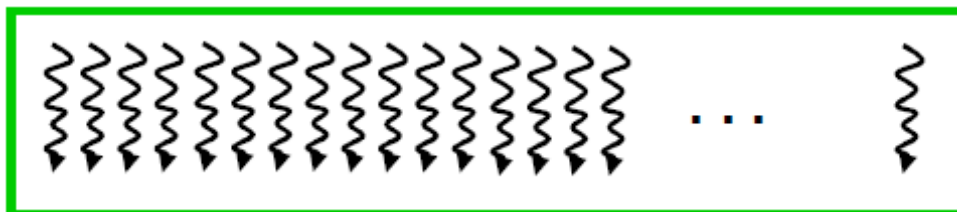


Figura 2.6. Malla (*grid*) formada por un único bloque (*block*) de N hilos paralelos (*threads*) repartidos a lo largo del eje x . Cada hilo se puede identificar mediante la variable `threadIdx.x`.

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.

5. Ejemplo

En el siguiente ejemplo se van a calcular los N primeros números impares de dos formas distintas. Una de ellas mediante una función ejecutada en la GPU (un *kernel*) y que por tanto se declarará como `__global__` y la otra ejecutada en la CPU, por lo que se va a declarar como una función de tipo `__host__`. La llamada al *kernel* se realizará utilizando la sintaxis “<<<1,N>>>” lo que indica que el *kernel* constará de un solo bloque con N hilos.

Si analizamos el ejemplo podremos darnos cuenta de que la principal diferencia entre ambas funciones es que la función *kernel* nos permite prescindir del bucle `for`, ya que el índice que se necesita para iterar en una función ejecutada en la CPU se puede sustituir por el índice identificativo de cada hilo lanzado (`threadIdx.x`). Esto es lo que se conoce como “bucle desenrollado” y representa la característica más importante dentro de la programación paralela con tarjetas gráficas ya que nos permite prescindir de un nivel de un bucle `for` en la mayoría de los problemas con paralelismo de datos, aquellos en los que no hay dependencias.

```

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

// defines
#define N 15

// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void impares_GPU(int *dev_impares)
{
    int id = threadIdx.x;
    dev_impares[id] = 2*id + 1;
}

// HOST: funcion llamada y ejecutada desde el host
__host__ void impares_CPU(int *hst_impares)
{
    for (int i = 0; i < N; i++)
    {
        hst_impares[i] = 2*i + 1;
    }
}

// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
    // declaraciones
    int *hst_salida;
    int *dev_salida;

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

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

    // EJECUCIÓN EN EL DEVICE
    // llamada a la funcion "impares_GPU()"
    impares_GPU <<<1,N>>>(dev_salida);

    // recogida de datos desde el device
    cudaMemcpy(hst_salida, dev_salida, N*sizeof(int), cudaMemcpyDeviceToHost);

    // impresion de resultados GPU
    printf("> SALIDA DE LA GPU:\n");
    printf("Los primeros %d numeros impares son:\n", N);
    for (int i = 0; i < N; i++)
        printf("[%2d]: %2d\n", i, hst_salida[i]);

    // EJECUCIÓN EN EL HOST
    // llamada a la funcion "impares_CPU()"
    impares_CPU(hst_salida);

    // impresion de resultados CPU
    printf("> SALIDA DE LA CPU:\n");
    printf("Los primeros %d numeros impares son:\n", N);
    for (int i = 0; i < N; i++)
        printf("[%2d]: %2d\n", i, hst_salida[i]);

    // salida
    time_t fecha;
    time(&fecha);
    printf("*****\n");
    printf("Programa ejecutado el: %s\n", ctime(&fecha));
    printf("<pulsa [INTRO] para finalizar>");
    getchar();
    return 0;
}

```


BÁSICO 3: Hilos y Bloques

1. Objetivo

Lanzar un *kernel* con múltiples hilos o *threads* agrupados en bloques.

2. Ejercicio

Reprogramar el ejercicio “Básico 2” de manera que sea válido para vectores de cualquier tamaño. El programa debe cumplir los siguientes requisitos:

- » El lanzamiento del *kernel* se debe realizar utilizando bloques de tamaño fijo (por ejemplo, bloques de 10 hilos), ajustando el número de bloques lanzados al tamaño de los vectores.
- » Por pantalla se debe mostrar:
 - Las principales propiedades del dispositivo seleccionado: nombre, capacidad de cómputo, número de multiprocesadores, número de núcleos, máximo número de hilos permitidos en cada eje y máximo número de bloques permitidos en cada eje.
 - Los dos vectores iniciales y el vector con el resultado final.

3. Elementos CUDA

Para realizar esta práctica se puede utilizar cualquiera de las funciones estudiadas anteriormente junto con los siguientes elementos:

- **gridDim.x**
Variable de tipo `int` que se genera automáticamente dentro de la función *kernel* donde se almacena el número de bloques que se han lanzado.
- **blockIdx.x**
Variable de tipo `int` que se genera automáticamente dentro de la función *kernel* y que permite identificar cada bloque que se ha lanzado.
- **blockDim.x**

Variable de tipo `int` que se genera automáticamente dentro de la función *kernel* y donde se almacena el número de hilos que se ha lanzado dentro de cada bloque.

- **`threadIdx.x`**

Variable de tipo `int` que se genera automáticamente dentro de la función *kernel* y que permite identificar cada hilo dentro de su bloque.

```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\ARCO\ARCO\...\bin/win64/Debug/practica.exe
*****
DEVICE 0: GeForce GT 730
*****
> Capacidad de Computo           : 3.5
> No. de MultiProcesadores       : 2
> No. de CUDA Cores (192x2)      : 384
> No. maximo de Hilos (por bloque) : 1024
[eje x -> 1024]
[eje y -> 1024]
[eje z -> 64]
> No. maximo de Bloques (por eje):
[eje x -> 2147483647]
[eje y -> 65535]
[eje z -> 65535]
*****
Introduce el numero de elementos: 33
> Vector de 33 elementos
> Lanzamiento con 4 bloques de 10 hilos (40 hilos)
VECTOR 1:
1 7 4 0 9 4 8 8 2 4 5 5 1 7 1 1 5 2 7 6 1 4 2 3 2 2 1 6 8 5 7 6 1
VECTOR 2:
1 6 7 5 8 6 1 2 2 3 2 4 1 6 7 2 5 1 1 7 1 5 5 4 2 8 8 4 9 0 4 7 1
SUMA:
2 13 11 5 17 10 9 10 4 7 7 9 2 13 8 3 10 3 8 13 2 9 7 7 4 10 9 10 17 5 11 13 2
*****
<pulsa [INTRO] para finalizar>

```

Figura 3.1.: Salida por pantalla de una posible solución.

4. Conceptos teóricos

4.1. Organización de hilos en CUDA

Como ya hemos comentado anteriormente, cuando se lanza un *kernel* se crea una malla o *grid* de hilos donde todos ellos ejecutan el mismo programa. Las opciones para agrupar estos hilos 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 es lanzar varios bloques con varios hilos en cada bloque.

Ya hemos visto que la sintaxis para lanzar un *kernel* con N hilos de ejecución, todos ellos agrupados en un único bloque, es la siguiente:

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

Dentro del código de la propia función *kernel* podemos utilizar variables que nos permiten conocer las dimensiones del *kernel* que hemos lanzado. Por ejemplo, el tamaño de la malla, que se refiere al número de bloques empleados, se almacena automáticamente en la variable `gridDim.x`, que en el ejemplo anterior tomaría el valor 1 al haber un único bloque. Por otro lado, el tamaño del bloque, es decir, el número de hilos por bloque, también está disponible dentro del código del *kernel* en la variable `blockDim.x` y que en este caso tomaría el valor N . Por último, también sabemos que para identificar cada uno de los hilos de un bloque se utiliza la variable `threadIdx.x` y que para cada hilo esta variable toma un valor distinto entre 0 y $N-1$.

4.2. Organización en bloques

El hardware de la GPU limita el número de hilos por bloque que podemos utilizar para lanzar un *kernel*. Este número puede ser mayor o menor dependiendo de la capacidad de cómputo de nuestra GPU y además puede ser distinto en cada eje del espacio. Hasta ahora nosotros estamos utilizando exclusivamente el eje x (de ahí la terminación `.x`) pero también se pueden utilizar también los ejes y y z . El número de hilos admitidos por una determinada GPU se puede averiguar a partir del campo `maxThreadsDim[i]` de la estructura de propiedades (ver **TABLA 5**), el cual nos da el máximo número de hilos que podemos lanzar en cada una de las direcciones del espacio ($i = 0, 1, 2 \Rightarrow$ ejes x, y , y z respectivamente) pero sin superar nunca el valor dado por `maxThreadsPerBlock`.

Para evitar la limitación del máximo número de hilos por bloque tenemos la alternativa de lanzar un *kernel* de N bloques con 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 también dispondríamos de N hilos pero ahora tendríamos una malla formada por N bloques repartidos a lo largo del eje x y con un hilo cada uno (Figura 3.2).

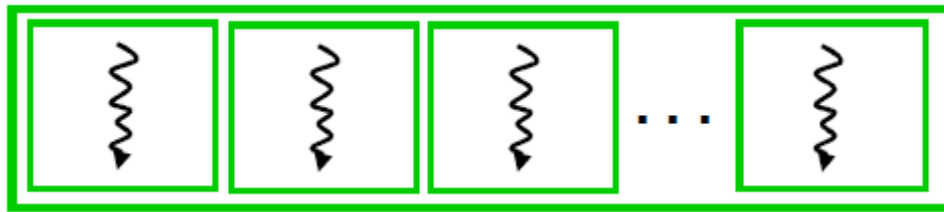


Figura 3.2. Malla formada por N bloques paralelos de 1 hilo repartidos a lo largo del eje x . Cada bloque se puede identificar mediante la variable `blockIdx.x`.

De igual modo, las dimensiones de este *kernel* las podemos obtener a partir de la variable `gridDim.x`, que en este caso tomaría el valor N y de la variable `blockDim.x`, que en este caso tomaría el valor 1. Sin embargo, si ahora queremos identificar los hilos para poder repartir tareas no podemos utilizar la variable `threadIdx.x`, ya 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` (el valor 0). Ahora la identificación de los hilos pasa por identificar el bloque al que pertenecen. Esto lo podemos hacer mediante la variable denominada `blockIdx.x`:

```
int myBlockID = blockIdx.x;
```

En nuestro ejemplo, al tener N bloques, cada uno de los hilos tendrá un valor distinto de `blockIdx.x` que irá desde 0 hasta $N-1$. No obstante, también existe un límite impuesto por el hardware en el número máximo de bloques que podemos lanzar, siendo como mínimo de 65535 bloques. El valor particular de una GPU lo podemos averiguar a partir del campo `maxGridSize[i]` de nuestra estructura de propiedades (ver **TABLA 5**), el cual nos da el máximo número de bloques permitidos en cada una de las direcciones del espacio ($i = 0, 1, 2 \Rightarrow$ ejes $x, y, y z$ respectivamente).

El siguiente código muestra cómo obtener todas estas limitaciones indagando en la estructura de propiedades `cudaDeviceProp`:

```
// 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 Hilos por bloque: %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...");
    getchar();
    return 0;
}

```

Por último, el caso más general sería el lanzamiento de un *kernel* con M bloques y N hilos por bloque (Figura 3.3), 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);
```

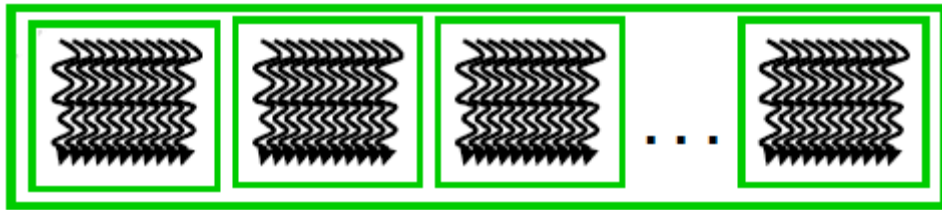


Figura 3.2. Malla formada por M bloques paralelos (gridDim.x) de N hilos cada uno (blockDim.x) repartidos a lo largo del eje x .

Ahora habrá un total de $M \times N$ hilos ejecutándose en paralelo (gridDim.x será igual a M y blockDim.x igual a N) y para identificarlos de forma unívoca será necesario hacer un uso conjunto de todas las variables estudiadas. De este modo, dentro del *kernel* cada hilo se puede identificar de forma unívoca mediante la siguiente expresión (Figura 3.4):

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

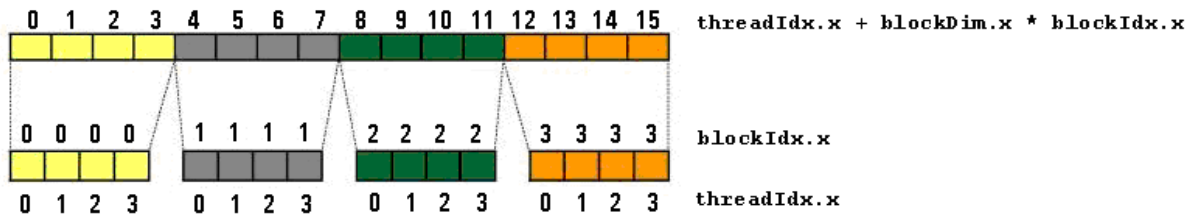


Figura 3.4. Identificación de los hilos dentro de una malla formada por cuatro bloques de cuatro hilos cada uno y repartidos a lo largo del eje x .

5. Ejemplo

La identificación de los hilos lanzados es una parte muy importante en cualquier programa paralelo ya que el índice que identifica a cada hilo es el único elemento del que disponemos para repartir el trabajo entre los diferentes hilos, es decir, para determinar qué operación debe realizar cada hilo o en qué zona de memoria debe leer o escribir.

Ya hemos visto que un mismo *kernel* puede ser lanzado de diferentes formas y esto afecta a forma que tenemos para identificar cada hilo. En este ejemplo se va a lanzar un *kernel* compuesto de 35 hilos pero con diferentes opciones de organización:

1. Todos los hilos en un mismo bloque.
2. Cada hilo en un bloque distinto.
3. Todos los hilos repartidos en 7 bloques de 5 hilos cada uno.

En este ejemplo, el trabajo del kernel es sencillo: cada hilo va a escribir en tres vectores o arrays distintos cada uno de sus tres índices identificativos (local, de bloque y global). El *kernel* se va a lanzar tres veces para observar cómo varían los índices en cada una de las opciones.

Es importante recordar que desde la propia función *kernel* no se debe llamar a la rutina del sistema operativo de impresión por pantalla `printf()`. Por ese motivo necesitamos que la función *kernel* rellene los tres arrays de 35 elementos de tipo `int` (un array para cada tipo de índice) para que después sean copiados en la memoria del *host* con el fin de ser impresos por pantalla.

En las versiones actuales de CUDA se ha implementado una versión de la rutina `printf()` que permite utilizarla dentro del código del *kernel*, pero no es compatible con todas las versiones y sólo se debe utilizar para depurar código. Además, al tratarse de una aplicación paralela y no secuencial, el orden de ejecución tanto de los hilos como de los bloques no es predecible y la interpretación de la salida puede resultar confusa.

```

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

// defines
#define N 35 // Lanzamiento con 35 hilos

// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void kernel(int *hilos, int *bloques, int *global)
// Funcion que escribe los indices de cada hilo
{
    int idHilo = threadIdx.x;
    int idBloque = blockIdx.x;
    int idGlobal = threadIdx.x + blockDim.x * blockIdx.x;
    // escritura en los tres buffer
    hilos[idGlobal] = idHilo;
    bloques[idGlobal] = idBloque;
    global[idGlobal] = idGlobal;
}
////////////////////////////////////
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
    // buscando dispositivos
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    if (deviceCount == 0)
    {
        printf("!!!!No se han encontrado dispositivos CUDA!!!!\n");
        printf("pulsa [INTRO] para finalizar"); getchar();
        return 1;
    }

    // declaraciones
    int *hst_hilos, *hst_bloques, *hst_global;
    int *dev_hilos, *dev_bloques, *dev_global;
    int Nbloques;
    int hilosB;

    // reserva en el host
    hst_hilos = (int*)malloc(N * sizeof(int));
    hst_bloques = (int*)malloc(N * sizeof(int));
    hst_global = (int*)malloc(N * sizeof(int));

    // reserva en el device
    cudaMalloc((void**)&dev_hilos, N * sizeof(int));
    cudaMalloc((void**)&dev_bloques, N * sizeof(int));
    cudaMalloc((void**)&dev_global, N * sizeof(int));

    // Numero de hilos
    printf("KERNEL de %d hilos:\n", N);

    // OPCION 1
    // dimensiones del kernel: 1 bloque de 35 hilos
    Nbloques = 1;
    hilosB = 35;

    // llamada al kernel
    kernel <<<Nbloques, hilosB >>> (dev_hilos, dev_bloques, dev_global);

    // recogida de datos desde el device
    cudaMemcpy(hst_hilos, dev_hilos, N * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(hst_bloques, dev_bloques, N * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(hst_global, dev_global, N * sizeof(int), cudaMemcpyDeviceToHost);

    // impresion de resultados
    printf("\n> Opcion 1:\n");
    printf("Lanzamiento con %d bloques de %d hilos (%d hilos)\n", Nbloques, hilosB,
    Nbloques*hilosB);

    printf("indice de hilo:\n");
    for (int i = 0; i < N; i++)
        printf("%2d ", hst_hilos[i]);
    printf("\n");

    printf("indice de bloque:\n");
    for (int i = 0; i < N; i++)

```



```

        printf("%2d ", hst_bloques[i]);
    printf("\n");

    printf("indice global:\n");
    for (int i = 0; i < N; i++)
        printf("%2d ", hst_global[i]);
    printf("\n");

    // OPCION 2
    // dimensiones del kernel: 35 bloques de 1 hilo
    Nbloques = 35;
    hilosB = 1;

    // llamada al kernel
    kernel <<<Nbloques, hilosB >>> (dev_hilos, dev_bloques, dev_global);

    // recogida de datos desde el device
    cudaMemcpy(hst_hilos, dev_hilos, N * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(hst_bloques, dev_bloques, N * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(hst_global, dev_global, N * sizeof(int), cudaMemcpyDeviceToHost);

    // impresion de resultados
    printf("\n> Opcion 2:\n");
    printf("Lanzamiento con %d bloques de %d hilos (%d hilos)\n", Nbloques, hilosB,
Nbloques*hilosB);

    printf("indice de hilo:\n");
    for (int i = 0; i < N; i++)
        printf("%2d ", hst_hilos[i]);
    printf("\n");

    printf("indice de bloque:\n");
    for (int i = 0; i < N; i++)
        printf("%2d ", hst_bloques[i]);
    printf("\n");

    printf("indice global:\n");
    for (int i = 0; i < N; i++)
        printf("%2d ", hst_global[i]);
    printf("\n");

    // OPCION 3
    // dimensiones del kernel: 7 bloques de 5 hilos
    Nbloques = 7;
    hilosB = 5;

    // llamada al kernel
    kernel <<<Nbloques, hilosB >>> (dev_hilos, dev_bloques, dev_global);

    // recogida de datos desde el device
    cudaMemcpy(hst_hilos, dev_hilos, N * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(hst_bloques, dev_bloques, N * sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(hst_global, dev_global, N * sizeof(int), cudaMemcpyDeviceToHost);

    // impresion de resultados
    printf("\n> Opcion 3:\n");
    printf("Lanzamiento con %d bloques de %d hilos (%d hilos)\n", Nbloques, hilosB,
Nbloques*hilosB);

    printf("indice de hilo:\n");
    for (int i = 0; i < N; i++)
        printf("%2d ", hst_hilos[i]);
    printf("\n");

    printf("indice de bloque:\n");
    for (int i = 0; i < N; i++)
        printf("%2d ", hst_bloques[i]);
    printf("\n");

    printf("indice global:\n");
    for (int i = 0; i < N; i++)
        printf("%2d ", hst_global[i]);
    printf("\n");

    // salida del programa
    printf("*****\n");
    printf("<pulsa [INTRO] para finalizar>"); getchar();
    return 0;
}

```


BÁSICO 4: Arrays Multidimensionales

1. Objetivo

Utilizar los hilos de ejecución de un *kernel* lanzado con dos dimensiones.

2. Ejercicio

Dada una matriz de tamaño $M \times N$ formada por valores aleatorios comprendidos entre 0 y 9, obtener una nueva matriz con el valor '0' en las posiciones donde el valor de la matriz inicial sea menor que 5 y con el valor '1' donde sea mayor o igual que 5.

El *kernel* debe lanzarse como un solo bloque bidimensional donde cada hilo de ejecución se encargue de obtener un único elemento de la matriz final. El programa debe mostrar las propiedades del dispositivo, el número de hilos lanzados en cada eje, la matriz inicial y la matriz final.

```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\ARCO\ARCO\..\bin\win64\Debug\practica.exe
*****
DEVICE 0: GeForce GT 730
*****
> Capacidad de Computo          : 3.5
> No. de MultiProcesadores      : 2
> No. de CUDA Cores (192x2)    : 384
> No. maximo de Hilos (por bloque) : 1024
*****
> KERNEL de 1 BLOQUE con 144 HILOS (24x6):
  eje x -> 24 hilos
  eje y -> 6 hilos
> MATRIZ ORIGINAL:
9 9 4 3 3 1 9 0 1 3 6 2 4 9 7 7 4 2 3 3 8 5 7 6
6 1 3 7 8 4 5 7 5 4 3 0 6 5 7 7 9 0 2 7 9 1 0 8
3 7 0 8 3 9 9 5 8 4 5 9 7 0 9 1 3 8 6 8 1 3 0 6
6 8 4 6 1 3 4 6 7 9 8 2 9 6 8 3 9 6 8 2 5 3 6 8
3 3 9 5 1 3 7 0 7 4 5 7 4 2 2 7 2 4 5 3 8 6 4 9
1 7 2 1 1 9 4 7 7 9 2 7 9 0 0 9 3 9 0 1 5 7 3 4

> MATRIZ FINAL:
1 1 0 0 0 0 1 0 0 0 1 0 0 1 1 1 0 0 0 0 1 1 1 1
1 0 0 1 1 0 1 1 1 0 0 0 1 1 1 1 1 0 0 1 1 0 0 1
0 1 0 1 0 1 1 1 1 0 1 1 1 0 1 0 0 1 1 1 0 0 0 1
1 1 0 1 0 0 0 1 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 1
0 0 1 1 0 0 1 0 1 0 1 1 0 0 0 1 0 0 1 0 1 1 0 1
0 1 0 0 0 1 0 1 1 1 0 1 1 0 0 1 0 1 0 0 1 1 0 0
*****
<pulsa INTRO para finalizar>

```

Figura 4.1.: Salida por pantalla de una posible solución.

3. Elementos CUDA

Para realizar esta práctica se pueden utilizar los siguientes elementos:

- **dim3**
Etiqueta que se coloca junto a la declaración de una variable para indicar que se trata de una variable tridimensional.
- **gridDim.{x,y,z}**
Variables de tipo `int` que se generan automáticamente dentro de la función *kernel*. Cada una almacena el número de bloques lanzados en la correspondiente dirección del espacio (x,y o z).
- **blockIdx.{x,y,z}**
Variables de tipo `int` que se generan automáticamente dentro de la función *kernel*. Cada una permite identificar a cada bloque en la correspondiente dirección del espacio (x,y o z).
- **blockDim.{x,y,z}**
Variables de tipo `int` que se generan automáticamente dentro de la función *kernel*. Cada una almacena el número de hilos lanzados en la correspondiente dirección del espacio (x,y o z).
- **threadIdx.{x,y,z}**
Variables de tipo `int` que se generan automáticamente dentro de la función *kernel*. Cada una permite identificar cada hilo en la correspondiente dirección del espacio (x,y o z).

4. Conceptos teóricos

4.1. 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 4.2). 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.

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);
```

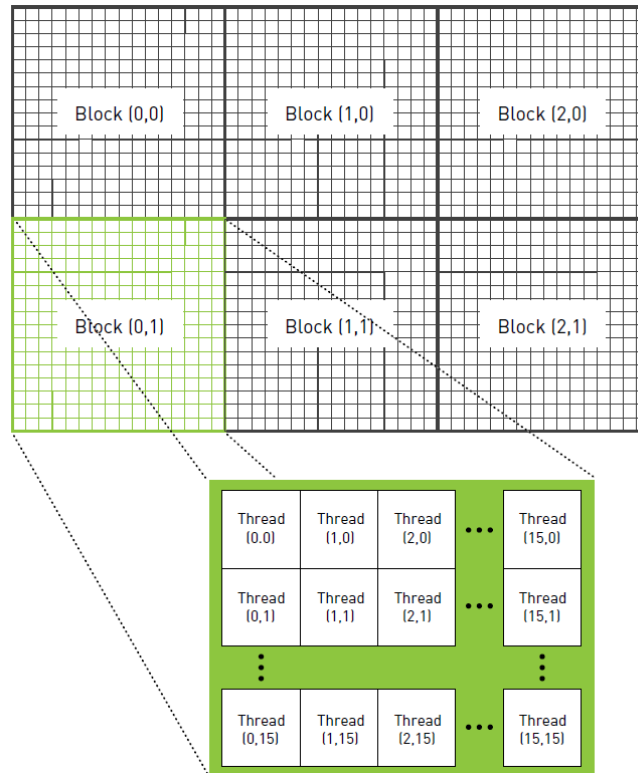


Figura 4.2. Distribución bidimensional de hilos en CUDA.

Sin embargo, ahora las variables `blocks` y `threads` hay que declararlas como variables “tridimensionales” utilizando un nuevo tipo de dato: `dim3`.

Al declarar este tipo de variables debemos especificar el tamaño de cada una de sus tres componentes (x, y, z) con un valor siempre mayor o igual que 1. Si alguna de ellas 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);
```

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, sólo que ahora debemos especificar en qué dirección del espacio. En tiempo de ejecución cada una de esas variables adquirirá un valor dentro del rango que hayamos indicado 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

```

Las dimensiones del *kernel* lanzado también se almacenan en las constantes `gridDim` y `blockDim`, sólo que ahora 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 4.2, 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.

Existen muchas posibilidades a la hora de distribuir los hilos. La decisión final de cómo hacerlo depende del problema en particular que vayamos a resolver pero teniendo presente que no podemos sobrepasar los límites impuestos por el hardware, límites que en su mayor parte dependen de la capacidad de cómputo de la GPU.

4.2. Reserva de memoria

Con el fin de poder acceder a la memoria del *device* desde el *host* es necesario realizar una reserva dinámica utilizando la función `cudaMalloc`. Esta función nos proporciona un espacio de memoria consecutivo (lineal) al que únicamente se puede acceder utilizando un solo índice. Por ejemplo:

```
cudaMalloc(void &dev_matriz, N*sizeof(int));
```

reserva espacio para un array de N elementos de tipo `int` donde el elemento i de dicho array será `dev_matriz[i]`. Sin embargo, si queremos trabajar con matrices el hecho de trabajar con un solo índice puede ser un inconveniente ya que es más fácil e intuitivo disponer de 2 índices para localizar un determinado dato (fila y columna). Esto se puede conseguir reservando memoria de manera estática. Por ejemplo, si deseamos trabajar con una matriz A de tamaño $M \times N$ y queremos utilizar dos índices, podemos reservar espacio en memoria de manera estática mediante:

```
int A[M][N];
```

Así, el elemento (i, j) de dicho array será `A[i][j]`. Sin embargo, si hemos reservado espacio en la memoria del *device* utilizando la función `cudaMalloc` para copiar los datos de la matriz A desde el *host* al *device* o viceversa, se nos plantea un problema:

```
cudaMalloc(void &dev_A, M*N*sizeof(int));
```

¿cuál será la posición del elemento (i, j) en el array del *device* `dev_A[.]`? Para resolver este problema hay que saber que existe una relación entre la posición de un elemento en un array bidimensional y su correspondiente posición en un array lineal tal como se indica en la Figura 4.3. De acuerdo con esta relación, tenemos que el elemento `A[i][j]` en el *host* se corresponde con elemento `dev_A[j + N*i]` en el *device*.

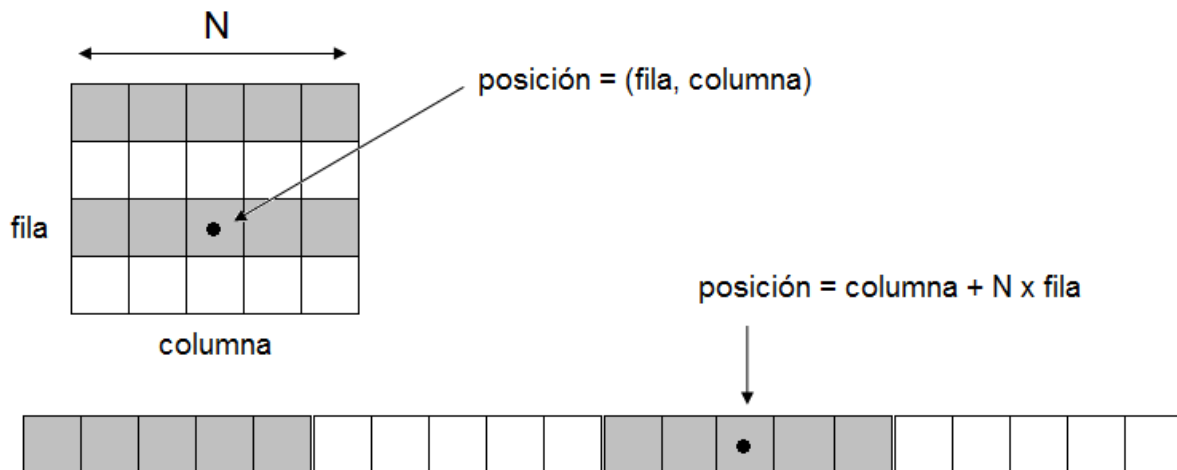


Figura 4.3. Relación entre la posición de un elemento en un array bidimensional y en un array lineal.

5. Ejemplo

El siguiente ejemplo muestra 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 $N \times N$. El lanzamiento se ha realizado utilizando un único bloque con $N \times N$ 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 <time.h>
#include <cuda_runtime.h>

// defines
#define N 10 // tamaño de la matriz NxN

// 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 fila
    int fila = threadIdx.y;
    // indice de columna
    int columna = threadIdx.x;

    // Calculamos la suma:
    // C[fila][columna] = A[fila][columna] + B[fila][columna] =
    // Para ello convertimos los indices de 'fila' y columna' en un 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));
    hst_B = (float*)malloc(N*N*sizeof(float));
    hst_C = (float*)malloc(N*N*sizeof(float));

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

    // inicializacion
    srand((int)time(NULL));
    for(int i=0;i<N*N;i++)
    {
        hst_A[i] = (float) rand() / RAND_MAX;
        hst_B[i] = (float) rand() / RAND_MAX;
    }

    // copia de datos
    cudaMemcpy( dev_A, hst_A, N*N*sizeof(float), cudaMemcpyHostToDevice );
    cudaMemcpy( dev_B, hst_B, 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("MATRIZ A:\n");
    for(int i=0;i<N;i++)
    {
        for(int j=0;j<N;j++)
        {
            printf("%2.1f ",hst_A[j+i*N]);
        }
        printf("\n");
    }

    printf("MATRIZ B:\n");
    for(int i=0;i<N;i++)
    {
        for(int j=0;j<N;j++)
        {
            printf("%2.1f ",hst_B[j+i*N]);
        }
        printf("\n");
    }

    printf("MATRIZ C:\n");
    for(int i=0;i<N;i++)
    {
        for(int j=0;j<N;j++)
        {
            printf("%2.1f ",hst_C[j+i*N]);
        }
        printf("\n");
    }

    // salida
    time_t fecha;
    time(&fecha);
    printf("*****\n");
    printf("Programa ejecutado el: %s\n", ctime(&fecha));
    printf("<pulsa [INTRO] para finalizar>");
    getchar();
    return 0;
}

```


BÁSICO 5:

Temporización GPU

1. Objetivo

Medir el tiempo de ejecución de un *kernel*.

2. Ejercicio

Dada una matriz de tamaño $M \times N$ formada por valores aleatorios comprendidos entre 0 y 5, obtener la matriz resultante de desplazar una posición hacia abajo todas las filas.

El programa debe mostrar las propiedades del dispositivo (nombre, capacidad de cómputo, número de multiprocesadores, número de núcleos y máximo número de hilos por bloque), el número de hilos lanzados, el tiempo de ejecución y las matrices inicial y final.

```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\ARCO\ARCO\...\bin/win64/Debug/practica.exe
*****
DEVICE 0: GeForce GT 730
*****
> Capacidad de Computo          : 3.5
> No. de MultiProcesadores      : 2
> No. de CUDA Cores (192x2)    : 384
> No. maximo de Hilos (por bloque) : 1024
*****
> KERNEL de 1 BLOQUE con 168 HILOS (28x6):
> Tiempo de ejecucion: 0.046112 ms
> MATRIZ ORIGINAL:
5 0 2 1 2 3 0 0 0 4 3 4 0 5 0 0 4 0 2 3 2 2 3 1 5 5 4 3
0 0 1 3 0 5 4 5 1 0 5 2 1 3 1 1 1 0 0 4 5 2 2 0 4 5 4 3
3 5 5 3 5 0 3 3 4 5 3 3 4 1 5 5 1 0 3 3 4 5 5 1 1 4 2 4
1 3 2 3 1 4 2 3 1 4 1 5 4 4 1 5 2 3 0 1 2 5 3 2 1 4 2 3
5 2 1 1 1 0 3 1 2 2 3 0 3 4 1 2 0 4 3 0 3 0 2 3 2 0 0 1
2 1 3 1 3 2 1 2 3 0 3 5 1 3 0 4 2 1 3 1 0 5 3 3 1 4 2 2
> MATRIZ FINAL:
2 1 3 1 3 2 1 2 3 0 3 5 1 3 0 4 2 1 3 1 0 5 3 3 1 4 2 2
5 0 2 1 2 3 0 0 0 4 3 4 0 5 0 0 4 0 2 3 2 2 3 1 5 5 4 3
0 0 1 3 0 5 4 5 1 0 5 2 1 3 1 1 1 0 0 4 5 2 2 0 4 5 4 3
3 5 5 3 5 0 3 3 4 5 3 3 4 1 5 5 1 0 3 3 4 5 5 1 1 4 2 4
1 3 2 3 1 4 2 3 1 4 1 5 4 4 1 5 2 3 0 1 2 5 3 2 1 4 2 3
5 2 1 1 1 0 3 1 2 2 3 0 3 4 1 2 0 4 3 0 3 0 2 3 2 0 0 1
*****
<pulsa INTRO para finalizar>

```

Figura 5.1.: Salida por pantalla de una posible solución.

3. Elementos CUDA

Para realizar esta práctica se pueden utilizar las siguientes funciones:

- `cudaEventCreate (cudaEvent_t *marca)`
Devuelve en la variable *marca* un evento para almacenar marcas de tiempo.
- `cudaEventRecord(cudaEvent_t marca, cudaStream_t stream)`
Graba una marca de tiempo en el evento *marca*. El valor *stream* especifica la lista de tareas a la que pertenece el evento, que por defecto es 0.
- `cudaEventSynchronize(cudaEvent_t marca)`
Detiene la ejecución del *host* hasta que el *device* haya alcanzado el evento señalado como *marca*.
- `cudaEventDestroy(cudaEvent_t marca)`
Elimina el evento *marca*.
- `cudaEventElapsedTime(float *tiempo, cudaEvent_t marca1, cudaEvent_t marca2)`
Almacena en la variable *tiempo* la diferencia de tiempo en milisegundos transcurridos entre los eventos indicados como *marca1* y *marca2*.

4. Conceptos teóricos

4.1. Sincronismo

En nuestro modelo de programación heterogéneo disponemos de dos elementos (el *host* y el *device*) que trabajan de forma independiente. Esto quiere decir que los dos elementos pueden estar trabajando simultáneamente. Esta forma de operación se conoce como *asíncrona*. Así, cuando lanzamos un *kernel* la GPU comienza a ejecutar su correspondiente código y la CPU continúa ejecutando el suyo (el programa principal *main*) sin esperar a que la GPU haya terminado. Esto es muy interesante desde el punto de vista del rendimiento porque permite aprovechar al máximo todos los recursos de computación pero, por otro lado, complica la medida del tiempo de ejecución ya que cualquier temporizador que utilicemos en el programa principal (*main*) sólo nos va a medir el tiempo de CPU.

Esta característica asíncrona de nuestro sistema hace que para medir el tiempo de ejecución de una aplicación CUDA tengamos que utilizar unas marcas de tiempo en la GPU denominadas *eventos*, para después leerlas desde el programa principal y así poder medir la diferencia de tiempo entre las marcas. Como veremos a continuación, esto implica además tener que sincronizar ambos dispositivos, es decir, detener la ejecución del programa

principal hasta estar seguros de que el trabajo realizado por la GPU se ha completado antes de leer las marcas.

4.2. 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. La forma de medir el 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 o lista de tareas en la que grabar el evento, donde 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()`. Como ya hemos dicho, esto es necesario debido al carácter asíncrono de las funciones de CUDA. Es decir, 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. El tiempo transcurrido entre ambos eventos lo podemos calcular utilizando la función `cudaEventElapsedTime()`:

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

Esta función 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`.

5. Ejemplo

En el siguiente fragmento de código se muestra cómo utilizar las funciones relacionadas con la gestión de eventos para medir el tiempo de ejecución transcurrido en la GPU. Es importante resaltar que estas funciones sólo sirven para medir tiempo de GPU, nunca para medir el tiempo de CPU:

```
// 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

    //...<<bloques,hilos>> ...

    // 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...");
    getchar();
    return 0;
}
```


ENTREGABLE 1:

Reducción paralela

1. Objetivo

Hacer que los hilos de ejecución de un *kernel* trabajen de forma cooperativa.

2. Ejercicio

Calcular de forma aproximada el valor del $\ln 2$ a partir de la siguiente expresión:

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} \dots$$

Además de mostrar las propiedades del dispositivo (nombre, capacidad de cómputo, número de multiprocesadores, número de núcleos y máximo número de hilos por bloque) y el número de hilos lanzados, el programa debe cumplir los siguientes requisitos:

- » Pedir por teclado un número de términos que sea potencia de 2.
- » Generar cada término de la sucesión dentro de la función *kernel*.
- » Sumar todos los términos utilizando un algoritmo de reducción paralela.
- » Mostrar el valor obtenido y el error relativo tomando como referencia el valor $\ln 2 = 0,6931472$.
- » Mostrar el tiempo de ejecución en ms.

```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\ARCO\ARCO\..\bin\win64\Debug\practica.exe
*****
DEVICE 0: GeForce GT 730
*****
> Capacidad de Computo      : 3.5
> No. de MultiProcesadores  : 2
> No. de CUDA Cores (192x2) : 384
> No. maximo de Hilos (por bloque) : 1024
*****
Introduce el numero de terminos (potencia de 2): 512
> Lanzamiento con 1 bloque de 512 hilos
> Valor calculado      : 0.692172
> Valor de ln(2)      : 0.693147
> Error relativo      : -0.140716%
> Tiempo de ejecucion: 0.028640 ms
*****
<pulsa INTRO para finalizar>

```

Figura 1.1.: Salida por pantalla de una posible solución.

3. Elementos CUDA

Para realizar esta práctica se puede utilizar cualquiera de las funciones estudiadas anteriormente además de la siguiente función del *device*:

- `__syncthreads()`

Función llamada desde el *device* y ejecutada en el *device*. Detiene el avance de los hilos de un mismo bloque hasta que todos hayan alcanzado este punto.

4. Conceptos teóricos

4.1. 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 una 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 hayan completado las instrucciones precedentes a `__syncthreads()` antes de que el hardware lance la siguiente instrucción para cualquier otro hilo. De este modo garantizamos 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.

5. 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*”. 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, de forma paralela vamos a aprovechar la presencia de múltiples hilos de ejecución para repartir el trabajo. Por sencillez, vamos a implementar un algoritmo de reducción paralela donde la longitud N del vector sea una potencia de 2. Así, el número de pasos necesarios para completar la suma será de $\log_2(N)$ en vez de los N pasos necesarios en una implementación secuencial. En cada paso tenemos que hacer que cada hilo sume el par de valores correspondientes a su posición y su posición más la mitad (de ahí la necesidad de que N sea una potencia de 2), guardando el resultado parcial en su respectiva posición de memoria tal como se muestra en la Figura 1.2. El vector de datos estará almacenado en una zona de memoria accesible por todos los hilos y en cada paso los hilos que trabajan son la mitad que en el paso anterior.

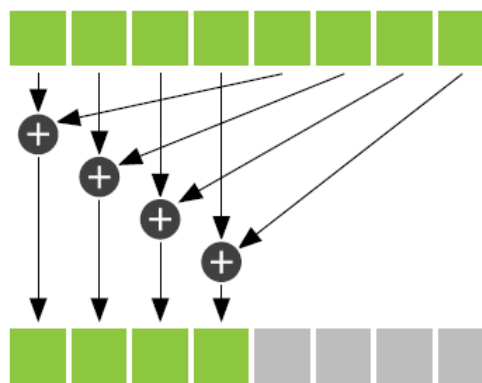


Figura 1.2. Primer paso en el algoritmo de reducción paralela de un vector de 8 elementos.

El aspecto crítico de este algoritmo es el sincronismo 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 dejando el resultado final en la primera posición de memoria del vector original.

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

```
// defines
#define N 16 // numero de terminos

// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void reduccion(int *datos, int *suma)
{
    // indice local de cada hilo -> kernel con un solo bloque de N hilos
    int myID = threadIdx.x;
    // rellenamos el vector de datos
    datos[myID] = myID + 1;
    // sincronizamos antes de seguir
    __syncthreads();

    // REDUCCION PARALELA
    int salto = N / 2;
    // realizamos log2(N) iteraciones
    while (salto > 0)
    {
        // en cada paso solo trabajan la mitad de los hilos
        if (myID < salto)
        {
            datos[myID] = datos[myID] + datos[myID + salto];
        }
        // sincronizamos los hilos
        __syncthreads();
        salto = salto / 2;
    }

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

ENTREGABLE 2: Ordenación por rango

1. Objetivo

Aprovechar el paralelismo de una GPU para optimizar algoritmos.

2. Ejercicio

Ordenar de menor a mayor un vector de tamaño N (definido en tiempo de ejecución por el usuario) con elementos aleatorios comprendidos entre 0 y 50. La ordenación debe realizarse utilizando el método de ordenación por rango, donde cada hilo de ejecución del *kernel* debe encargarse de calcular el rango de un único elemento y colocarlo en el vector final.

El programa debe mostrar las propiedades del dispositivo (nombre, capacidad de cómputo, número de multiprocesadores, número de núcleos y máximo número de hilos por bloque), el número de elementos a ordenar y además:

- » El vector de entrada desordenado.
- » El vector de salida ordenado.
- » El tiempo de ejecución en ms.

```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0_ARCO\ARCO\..\bin\win64\Debug\practica.exe
*****
DEVICE 0: GeForce GT 730
*****
> Capacidad de Computo           : 3.5
> No. de MultiProcesadores       : 2
> No. de CUDA Cores (192x2)     : 384
> No. maximo de Hilos (por bloque) : 1024
*****
  Introduce el numero de elementos a ordenar: 26
> Tiempo de ejecucion: 0.095264 ms
> VECTOR INICIAL:
  44 46 17 14 27 28 16 37 31  8  9 48 36 24 18 36 45 42 12 18  0  1 12 16 13 27
> VECTOR ORDENADO:
  0  1  8  9 12 12 13 14 16 16 17 18 18 24 27 27 28 31 36 36 37 42 44 45 46 48
*****
<pulsa INTRO para finalizar>

```

Figura 2.1.: Salida por pantalla de una posible solución.

3. Algoritmo de ordenación por rango

El algoritmo de ordenación por rango es muy sencillo aunque poco óptimo para ser implementado de forma secuencial. El algoritmo consiste en comparar cada elemento a ordenar con todos los demás con el fin de averiguar su rango. El rango de un elemento se refiere a su posición dentro el vector final. Si disponemos de n elementos, el algoritmo implementado de forma secuencial tiene una complejidad de orden $O(n^2)$, pero una implementación paralela da lugar una complejidad de orden $O(n)$.

En un ordenamiento de menor a mayor, para calcular el rango de un elemento basta con contar el número de elementos que son más pequeños que él. En el caso de que dos elementos sean iguales, se considera que es más pequeño el que originalmente se encuentra en una posición más baja. Por ejemplo, dada la siguiente lista de elementos:

```
vector_inicio = [ 12 20 11 30 20 14 18 25 22 23]
```

donde hay dos elementos repetidos, el rango de cada uno de ellos sería:

```
( 1 4 0 9 5 2 3 8 6 7)
```

Utilizando estos rangos podemos generar la lista de elementos ordenados de menor a mayor, la cual quedaría del siguiente modo:

```
vector_final = [ 11 12 14 18 20 20 22 23 25 30]
```

ENTREGABLE 3: Gráficos en CUDA

1. Objetivo

Dibujar una imagen de tipo *bitmap*.

2. Ejercicio

Dibujar un tablero de ajedrez como el mostrado en la figura 3.1. Se debe utilizar un *kernel* bidimensional con bloques de 16×16 hilos.

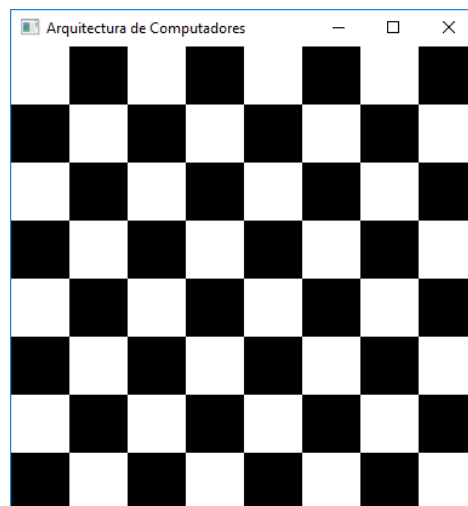


Figura 3.1. Tablero de ajedrez.

3. Elementos CUDA

Para realizar esta práctica se puede utilizar cualquiera de los recursos ya estudiados. Por otro lado, para poder visualizar el resultado y hacer que el Sistema Operativo nos muestre una ventana con la imagen del *bitmap* es necesario utilizar algunas funciones especiales. Las funciones que vamos a utilizar pertenecen a la API de **OpenGL** y están definidas en el archivo de cabecera “`cpu_bitmap.h`”.

4. Conceptos teóricos

4.1. Imágenes digitales

Hay situaciones donde resulta interesante distribuir los hilos de ejecución de un *kernel* de CUDA de una manera análoga al problema que se pretende resolver, como es el caso de las operaciones con matrices. Esta situación se da también cuando estamos trabajando con imágenes digitales ya que, como veremos a continuación, una imagen digital es también una matriz de números.

Una imagen “real” es 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 . Si convertimos estas cantidades continuas en cantidades discretas a través de los procesos denominados *muestreo* y *cuantificación* (Figura 3.2) obtenemos una imagen “digital”:

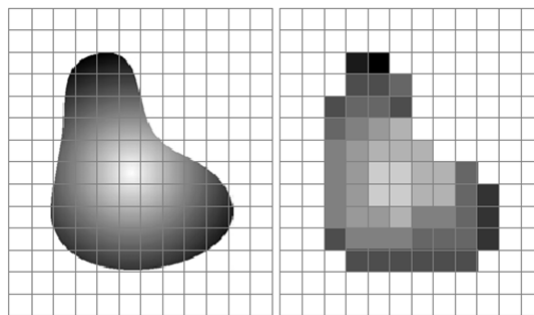


Figura 3.2. Imagen digital obtenida tras un proceso de muestreo y cuantificación.

- Cada muestra recibe el nombre de *elemento de imagen* o **píxel**.
- Cada valor de intensidad luminosa recibe el nombre de *nivel de gris*. El número de niveles de gris depende del número de bits empleados en el proceso de cuantificación. Por ejemplo, utilizando 8 bits el número de niveles disponibles es 256, desde el 0 (negro) hasta el 255 (blanco).

Tras el proceso de muestreo y cuantificación lo que obtenemos es una matriz de números donde cada número representa la intensidad luminosa en ese punto. Así, podemos decir que una imagen digital no es más que la representación de una imagen real a partir de una matriz numérica (Figura 3.3).

En conclusión, esto nos hace ver que la forma más adecuada para trabajar con imágenes digitales es utilizando un hardware que nos permita operar con grandes matrices de números. En este contexto, vemos que nuestro entorno de programación CUDA encaja

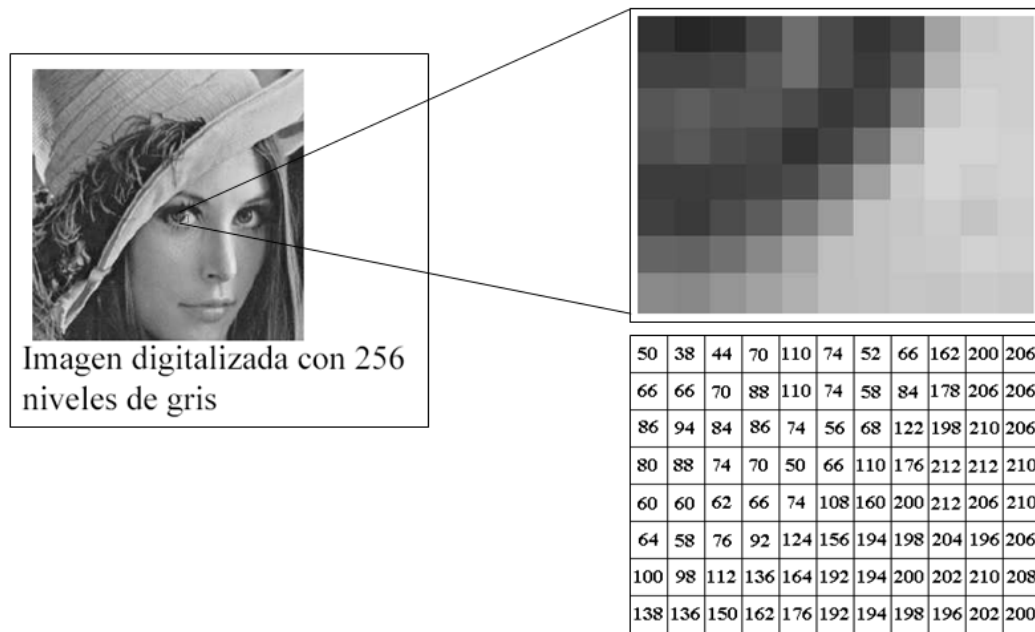


Figura 3.3. Representación de una imagen digital como una matriz de números.

perfectamente con dicho propósito ya que nos permite lanzar un *kernel* bidimensional con hilos repartidos en ambas direcciones del espacio y haciendo corresponder a cada hilo con un píxel de nuestra imagen digital.

4.2. 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 (o en 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 3.3).
- Imágenes en color, donde se necesita almacenar 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 3.4).

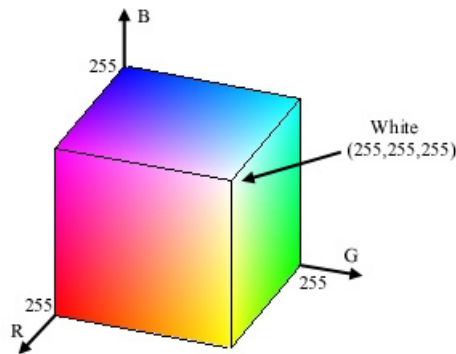


Figura 3.4. Representación del espacio de color **RGB** utilizando valores de 8 bits.

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 necesita disponer de al menos tres valores por cada píxel, los correspondientes a las componentes roja, verde y azul.

4.3. 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 3.5).

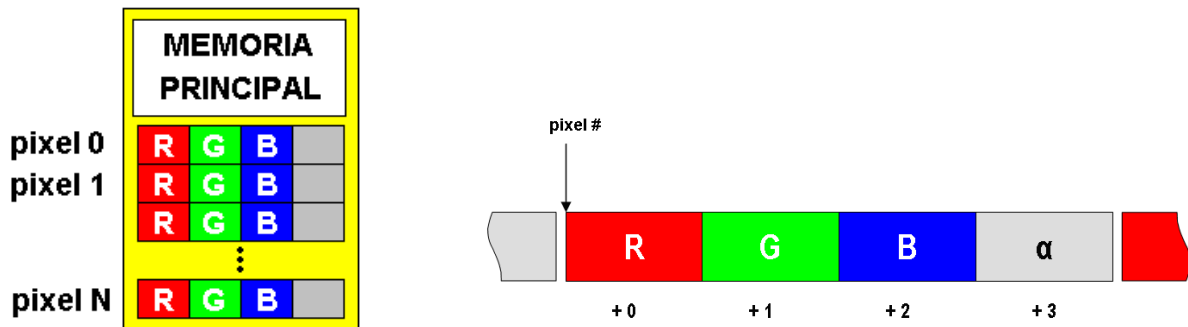


Figura 3.5. Ubicación en memoria principal de un mapa de bits de 4 canales y 8 bits por canal.

4.4. 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. Ejemplo

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 `gpu_bitmap.h` suministrado

junto a este manual. En este ejemplo se dibuja un *bitmap* donde cada hilo se encarga de dar color a un píxel asignando un número entero de 8 bits (un valor comprendido entre 0 y 255) para cada canal. En este ejemplo, para obtener un resultado “vistoso”, el color de cada píxel se ha escogido asignando 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.

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

// defines
#define ANCHO 1024 // Dimension horizontal
#define ALTO 512 // Dimension vertical

// 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 hilo
    int posicion = x + y * blockDim.x * gridDim.x;
    // cada hilo obtiene la posicion de un pixel
    int pixel = posicion * 4;
    // cada hilo rellena los 4 canales con un valor arbitrario
    imagen[pixel + 0] = y % 256; // canal R
    imagen[pixel + 1] = (x + y) % 256; // canal G
    imagen[pixel + 2] = x % 256; // canal B
    imagen[pixel + 3] = 0; // canal alfa
}

// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
    // Declaracion del bitmap
    RenderGPU foto(ANCHO, ALTO);

    // Tamaño en bytes
    size_t size = foto.image_size();

    // Asignacion a la memoria del host
    unsigned char *host_bitmap = foto.get_ptr();

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

    // Lanzamos un kernel con bloques de 256 hilos (16x16)
    dim3 hilosB(16,16);
    // Calculamos el numero de bloques necesario
    dim3 Nbloques(ANCHO/16, ALTO/16);
    // Generamos el bitmap
    kernel<<<Nbloques,hilosB>>>( dev_bitmap );

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

    // Visualizacion y salida
    printf("\n...pulsa [ESC] para finalizar...");
    foto.display_and_exit();

    return 0;
}
```

ENTREGABLE 4: Procesamiento de imágenes

1. Objetivo

Manipular una imagen digital de tipo *bitmap* leída desde un archivo.

2. Ejercicio

Convertir una imagen en color a escala de grises. Una imagen en escala de grises tiene el mismo valor o nivel de gris (**Y**) en sus tres canales. Este valor **Y** de cada píxel se puede obtener a partir de su color **RGB** utilizando la siguiente expresión:

$$Y = 0,299 \times R + 0,587 \times G + 0,114 \times B$$

La práctica debe cumplir los siguientes requisitos:

- » El programa principal (*main*) se debe encargar de leer el archivo “*imagen.bmp*” y almacenar los datos en la memoria del *host* en el orden **RGBA**.
- » El *kernel* se debe encargar de convertir la imagen a escala de grises.
- » Al terminar se debe mostrar el tiempo de ejecución y visualizar la imagen resultante en escala de grises utilizando las funciones de la API de **OpenGL** definidas en el archivo de cabecera “*gpu_bitmap.h*”.



Figura 4.1. Imagen en color y su correspondiente versión en escala de grises.

3. Archivos de imagen

En general, cuando hablamos de formatos de imagen nos referimos al formato del archivo que contiene los datos. La disposición típica de los datos en un archivo de imagen (figura 4.2) consiste en una serie de bytes con información de control (cabecera) seguido de los bytes relativos a la imagen propiamente dicha (datos).

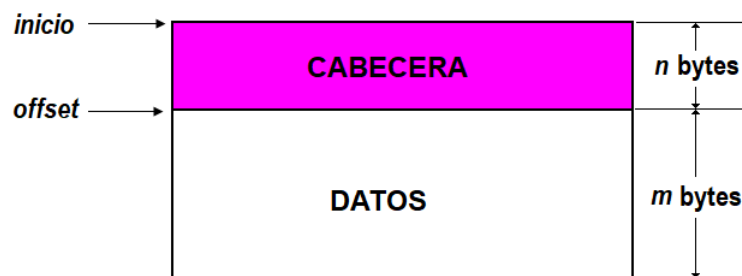


Figura 4.2. Disposición de los datos en un archivo de imagen.

La información de control codificada en la cabecera o *header* del archivo es necesaria para interpretar el bloque con los datos de imagen. Entre otras cosas esta información incluye un identificador del tipo de archivo, las dimensiones de la imagen, la profundidad de color, la ubicación del bloque de datos (*offset*), comentarios, etc.

El bloque correspondiente a los datos de imagen es una codificación del mapa de bits que describe la imagen. Dependiendo del tipo de archivo estos datos pueden estar a su vez comprimidos mediante algún tipo de algoritmo especificado en la cabecera.

3.1. Archivos BMP

El formato de archivo **BMP** es un formato de imagen de mapa de bits propio del sistema operativo Microsoft Windows, en cuyo sistema también se conoce como mapa de bits independiente de dispositivo o *device independent bitmap file format (DIB)*. Puede albergar imágenes con diferentes profundidades de color (32 bits, 24 bits, 8 bits y 1 bit). En general los datos están sin comprimir (de ahí su gran tamaño) aunque a veces se les puede aplicar una compresión sin pérdida de calidad conocida como compresión **RLE** (Run-length encoding).

Los archivos tipo **BMP** constan de al menos dos cabeceras. La primera de ellas tiene 14 bytes de longitud y en ella se almacena información general del archivo. Esta cabecera comienza con las letras 'BM' codificadas en ASCII (valores 0x42 y 0x4D) lo cual permite

identificar el tipo de archivo. La segunda cabecera, conocida como cabecera DIB y generalmente de 40 bytes de longitud, almacena información detallada de la imagen y de la profundidad de color. Al ser un formato típico de los sistemas Windows, el ordenamiento de los datos es de tipo little-endian. Una información completa y detallada del contenido de la cabecera de un fichero **BMP** se muestra en la **TABLA 8**.

TABLA 8. Estructura de la cabecera de un fichero **BMP**.

Dirección	Bytes	Descripción	
0x00	2	Tipo de fichero ("BM").	Bitmap file header
0x02	4	Tamaño del archivo en bytes (incluye cabeceras y datos).	
0x06	2	Reservado.	
0x08	2	Reservado.	
0x0A	4	Dirección de inicio de los datos de la imagen.	
0x0E	4	Tamaño en bytes de la segunda cabecera (40 bytes).	DIB header
0x12	4	Anchura en píxels.	
0x16	4	Altura en píxeles.	
0x1A	2	Número de planos.	
0x1C	2	Número de bits por píxel.	
0x1E	4	Método de compresión utilizado.	
0x22	4	Tamaño de la imagen en bytes.	
0x26	4	Resolución horizontal.	
0x2A	4	Resolución vertical.	
0x2E	4	Tamaño de la tabla de color.	
0x32	4	Contador de colores importantes.	

4. Manejo de archivos

El manejo de archivos en C se realiza a través de una serie de funciones definidas en `<stdio.h>`. Todas ellas consideran que un archivo es una estructura que contiene la información necesaria para su manejo, incluyendo un puntero al buffer de datos, un indicador de posición dentro del buffer y distintos indicadores de estado. Esta estructura se denomina **FILE** y la función que crea e inicializa esta estructura para poder manejar un

archivo en particular es `fopen()`. Hay varias decenas de funciones relacionadas con el manejo de ficheros (<http://www.cplusplus.com/reference/cstdio/>) pero las más básicas son las siguientes:

- **FILE*** `fopen(const char *filename, const char *mode)`
Abre el archivo cuyo nombre está especificado en `filename`. En la apertura en modo binario el parámetro `mode` puede ser “rb” para lectura, “wb” para escritura o “ab” como apéndice. En modo texto¹ no se añade la letra “b”. Si ha habido éxito el archivo se asocia a una estructura `FILE` devuelta como puntero para posteriores referencias.
- **int** `fclose(FILE *stream)`
Cierra el archivo asociado a la estructura `FILE` devuelta por la función `fopen()`. Si ha habido éxito la función devuelve el valor 0.
- **size_t** `fread(void *buf, size_t size, size_t count, FILE *stream)`
Lee `count` elementos, cada uno con un tamaño de `size` bytes, desde el fichero asociado a la estructura `stream` y los almacena en el buffer apuntado por `buf`. La función devuelve el número de elementos leídos con éxito.
- **size_t** `fwrite(void *buf, size_t size, size_t count, FILE *stream)`
Escribe `count` elementos, cada uno con un tamaño de `size` bytes, almacenados en el buffer apuntado por `buf` y los escribe en el fichero asociado a la estructura `stream`. La función devuelve el número de elementos escritos con éxito.
- **int** `fseek(FILE *stream, size_t offset, int origin)`
Fija el indicador de posición dentro del archivo asociado a la estructura `stream` en una nueva posición definida sumando el valor `offset` a la posición de referencia especificada por `origin`. Esta referencia puede ser el comienzo del archivo (`SEEK_SET`), la posición actual (`SEEK_CUR`) o el final del archivo (`SEEK_END`). Si ha habido éxito la función devuelve el valor 0.
- **void** `rewind(FILE *stream)`
Fija el indicador de posición del archivo asociado a la estructura `stream` al comienzo del archivo.

4.1. Lectura de un archivo BMP

El siguiente código muestra cómo extraer los datos de un archivo de tipo **BMP**. Dado que la información de color está almacenada en el orden **BGR**, para visualizar la imagen es necesario reordenar los canales y añadir el canal α y así disponer los datos en el orden **RGBA** y poder utilizar las funciones de **OpenGL** encapsuladas dentro del fichero de cabecera `gpu_bitmap.h`:

¹ Los archivos de texto son archivos que contienen caracteres especiales. Dependiendo del sistema donde se esté ejecutando la aplicación se pueden producir conversiones de alguno de estos caracteres especiales durante las operaciones de entrada/salida en modo texto con el fin de adaptarlos al formato de archivo de texto específico del sistema. Aunque hay sistemas que no hacen distinción, utilizar el modo adecuado facilita la portabilidad.

```

////////////////////////////////////
// includes
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "gpu_bitmap.h"

// HOST: funcion llamada desde el host y ejecutada en el host
__host__ void leerBMP_RGBA(const char* nombre, int* w, int* h, unsigned char **imagen);
// Funcion que lee un archivo de tipo BMP:
// -> entrada: nombre del archivo
// <- salida : ancho de la imagen en pixeles
// <- salida : alto de la imagen en pixeles
// <- salida : puntero al array de datos de la imagen en formato RGBA

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

    //////////////////////////////////////
    // Leemos el archivo BMP
    unsigned char *host_color;
    int ancho, alto;

    leerBMP_RGBA("imagen.bmp", &ancho, &alto, &host_color);

    //////////////////////////////////////
    // Declaracion del bitmap RGBA
    RenderGPU foto(ancho, alto);

    // Tamaño en bytes de una imagen de tipo RGBA
    size_t img_size = foto.image_size();

    // puntero al framebuffer vinculado con la estructura RenderGPU
    unsigned char *host_bitmap = foto.get_ptr();

    // Copiamos los datos de la imagen al framebuffer:
    cudaMemcpy(host_bitmap, host_color, img_size, cudaMemcpyHostToHost);

    // Salida
    printf("\n...pulsa [ESC] para finalizar...");

    // Visualizacion
    foto.display_and_exit();

    // Fin
    return 0;
}

// Funcion que lee un archivo de tipo BMP:
__host__ void leerBMP_RGBA(const char* nombre, int* w, int* h, unsigned char **imagen)
{
    // Lectura del archivo .BMP
    FILE *archivo;

    // Abrimos el archivo en modo solo lectura binario
    if ((archivo = fopen(nombre, "rb")) == NULL)
    {
        printf("\nERROR ABRIENDO EL ARCHIVO %s...", nombre);
        // salida
        printf("\npulsa [INTRO] para finalizar");
        getchar();
        exit(1);
    }
    printf("> Archivo [%s] abierto:\n", nombre);

    // En Windows, la cabecera tiene un tamaño de 54 bytes:
    // 14 bytes (BMP header) + 40 bytes (DIB header)

    // BMP HEADER
    // Extraemos cada campo y lo almacenamos en una variable del tipo adecuado
    // posicion 0x00 -> Tipo de archivo: "BM" (leemos 2 bytes)
    unsigned char tipo[2];
    fread(tipo, 1, 2, archivo);

    // Comprobamos que es un archivo BMP
    if(tipo[0] != 'B' || tipo[1] != 'M' )
    {
        printf("\nERROR: EL ARCHIVO %s NO ES DE TIPO BMP...", nombre);
        // salida

```

```

        printf("\npulsa [INTRO] para finalizar");
        getchar();
        exit(1);
    }

    // posicion 0x02 -> Tamaño del archivo .bmp (leemos 4 bytes)
    unsigned int file_size;
    fread(&file_size, 4, 1, archivo);

    // posicion 0x06 -> Campo reservado (leemos 2 bytes)
    // posicion 0x08 -> Campo reservado (leemos 2 bytes)
    unsigned char buffer [4];
    fread(buffer, 1, 4, archivo);

    // posicion 0x0A -> Offset a los datos de imagen (leemos 4 bytes)
    unsigned int offset;
    fread(&offset, 4, 1, archivo);

    // imprimimos los datos
    printf(" \nDatos de la cabecera BMP\n");
    printf("> Tipo de archivo      : %c%c\n", tipo[0], tipo[1]);
    printf("> Tamano del archivo      : %u KiB\n", file_size / 1024);
    printf("> Offset de datos          : %u bytes\n", offset);

    // DIB HEADER
    // Extraemos cada campo y lo almacenamos en una variable del tipo adecuado
    // posicion 0x0E -> Tamaño de la cabecera DIB (BITMAPINFOHEADER) (leemos 4 bytes)
    unsigned int header_size;
    fread(&header_size, 4, 1, archivo);

    // posicion 0x12 -> Ancho de la imagen (leemos 4 bytes)
    unsigned int ancho;
    fread(&ancho, 4, 1, archivo);

    // posicion 0x16 -> Alto de la imagen (leemos 4 bytes)
    unsigned int alto;
    fread(&alto, 4, 1, archivo);

    // posicion 0x1A -> Numero de planos de color (leemos 2 bytes)
    unsigned short int planos;
    fread(&planos, 2, 1, archivo);

    // posicion 0x1C -> Profundidad de color (leemos 2 bytes)
    unsigned short int color_depth;
    fread(&color_depth, 2, 1, archivo);

    // posicion 0x1E -> Tipo de compresion (leemos 4 bytes)
    unsigned int compresion;
    fread(&compresion, 4, 1, archivo);

    // imprimimos los datos
    printf(" \nDatos de la cabecera DIB\n");
    printf("> Tamano de la cabecera: %u bytes\n", header_size);
    printf("> Ancho de la imagen   : %u pixeles\n", ancho);
    printf("> Alto de la imagen    : %u pixeles\n", alto);
    printf("> Planos de color      : %u\n", planos);
    printf("> Profundidad de color : %u bits/pixel\n", color_depth);
    printf("> Tipo de compresion   : %s\n", (compresion == 0) ? "none" : "unknown");

    // LEEMOS LOS DATOS DEL ARCHIVO
    // Calculamos espacio para una imagen de tipo RGBA:
    size_t img_size = ancho * alto * 4;

    // Reserva para almacenar los datos del bitmap
    unsigned char *datos = (unsigned char*)malloc(img_size);

    // Desplazamos el puntero FILE hasta el comienzo de los datos de imagen: 0 + offset
    fseek(archivo, offset, SEEK_SET);

    // Leemos pixel a pixel, reordenamos (BGR -> RGB) e insertamos canal alfa
    unsigned int pixel_size = color_depth / 8;
    for (unsigned int i = 0; i < ancho * alto; i++)
    {
        fread(buffer, 1, pixel_size, archivo); // leemos un pixel
        datos[i * 4 + 0] = buffer[2]; // escribimos canal R
        datos[i * 4 + 1] = buffer[1]; // escribimos canal G
        datos[i * 4 + 2] = buffer[0]; // escribimos canal B
        datos[i * 4 + 3] = buffer[3]; // escribimos canal alfa (si lo hay)
    }

```

```
// Cerramos el archivo
fclose(archivo);

// PARAMETROS DE SALIDA
// Ancho de la imagen en pixeles
*w = ancho;
// Alto de la imagen en pixeles
*h = alto;
// Puntero al array de datos RGBA
*imagen = datos;

// Salida
return;
}
```


ENTREGABLE 5: Rendimiento GPU vs CPU

1. Objetivo

Calcular la ganancia en rendimiento de la GPU respecto de la CPU.

2. Ejercicio

Ordenar de menor a mayor un vector de N elementos aleatorios comprendidos entre 0 y 50 utilizando el método de ordenación por rango.

Se debe medir el tiempo de ejecución en la CPU y en la GPU para diferentes valores de N . Para cada valor de N el tiempo debe obtenerse como el resultado de una media aritmética de varias ejecuciones (por ejemplo, 5 veces).

La práctica debe cumplir los siguientes requisitos:

- » Mostrar por pantalla las principales características del *device*, el número de hilos por bloque y el número de bloques lanzados, el tamaño del vector, los tiempos de ejecución y el número de iteraciones utilizadas para realizar los cálculos.
- » Rellenar una tabla donde aparecen los tiempos medios de ejecución del *host*, del *device* y la ganancia correspondiente para diferentes valores de N .
- » Representar los valores de la tabla en una gráfica.

N	32	64	128	256	512	1024	2048	4096
Tiempo GPU (ms)								
Tiempo CPU (ms)								
$G_{GPU/CPU}$								

```

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\ARCO\ARCO\../bin/win64/Debug/practica.exe
*****
EJECUCION GPU...
Iteracion no. [0]: tiempo = 2.081792 ms
Iteracion no. [1]: tiempo = 2.143072 ms
Iteracion no. [2]: tiempo = 2.077696 ms
Iteracion no. [3]: tiempo = 2.081024 ms
Iteracion no. [4]: tiempo = 2.078976 ms
> Tiempo medio de ejecucion GPU: 2.092512 ms

EJECUCION CPU...
Iteracion no. [0]: tiempo = 21.389265 ms
Iteracion no. [1]: tiempo = 21.411075 ms
Iteracion no. [2]: tiempo = 21.353662 ms
Iteracion no. [3]: tiempo = 21.167632 ms
Iteracion no. [4]: tiempo = 22.897389 ms
> Tiempo medio de ejecucion CPU: 21.643805 ms

> Ganancia GPU/CPU = 10.343456
*****
> Vector 11:
> Numero de elementos a ordenar: [4096]
> Lanzamiento con bloques de 512 hilos y 8 bloques.
> TOTAL: 4096 hilos
*****
EJECUCION GPU...
Iteracion no. [0]: tiempo = 5.704864 ms
Iteracion no. [1]: tiempo = 5.768160 ms
Iteracion no. [2]: tiempo = 5.706976 ms
Iteracion no. [3]: tiempo = 6.219776 ms
Iteracion no. [4]: tiempo = 5.707776 ms
> Tiempo medio de ejecucion GPU: 5.821510 ms

EJECUCION CPU...
Iteracion no. [0]: tiempo = 85.519852 ms
Iteracion no. [1]: tiempo = 87.156274 ms
Iteracion no. [2]: tiempo = 85.071136 ms
Iteracion no. [3]: tiempo = 85.150359 ms
Iteracion no. [4]: tiempo = 88.206380 ms
> Tiempo medio de ejecucion CPU: 86.220802 ms

> Ganancia GPU/CPU = 14.810727
*****
RESUMEN DE RESULTADOS
*****
> N = 2 [GPU: 0.008294 ms] [CPU: 0.000064 ms] [Ganancia GPU/CPU = 0.007734]
> N = 4 [GPU: 0.009722 ms] [CPU: 0.000257 ms] [Ganancia GPU/CPU = 0.026394]
> N = 8 [GPU: 0.014234 ms] [CPU: 0.000513 ms] [Ganancia GPU/CPU = 0.036055]
> N = 16 [GPU: 0.022214 ms] [CPU: 0.001219 ms] [Ganancia GPU/CPU = 0.054866]
> N = 32 [GPU: 0.039117 ms] [CPU: 0.007762 ms] [Ganancia GPU/CPU = 0.198430]
> N = 64 [GPU: 0.072109 ms] [CPU: 0.023478 ms] [Ganancia GPU/CPU = 0.325595]
> N = 128 [GPU: 0.136339 ms] [CPU: 0.104433 ms] [Ganancia GPU/CPU = 0.765982]
> N = 256 [GPU: 0.264051 ms] [CPU: 0.329465 ms] [Ganancia GPU/CPU = 1.247733]
> N = 512 [GPU: 0.513504 ms] [CPU: 1.337811 ms] [Ganancia GPU/CPU = 2.605260]
> N = 1024 [GPU: 1.037037 ms] [CPU: 5.265158 ms] [Ganancia GPU/CPU = 5.077118]
> N = 2048 [GPU: 2.092512 ms] [CPU: 21.643805 ms] [Ganancia GPU/CPU = 10.343456]
> N = 4096 [GPU: 5.821510 ms] [CPU: 86.220802 ms] [Ganancia GPU/CPU = 14.810727]
*****

```

Figura 5.1.: Fragmento de la salida por pantalla de una posible solución.

3. Temporización CPU

Como hemos mencionado anteriormente, si queremos medir el tiempo de ejecución de un programa ejecutado en la CPU debemos recurrir a funciones de la biblioteca de C. Sin embargo, para conseguir una mayor resolución temporal vamos a ver en el siguiente ejemplo cómo utilizar eventos del sistema operativo para medir el tiempo de cálculo de una multiplicación entre dos matrices cuadradas:

```

////////////////////////////////////
// includes
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#ifdef __linux__
#include <sys/time.h>
typedef struct timeval event;
#else
#include <windows.h>
typedef LARGE_INTEGER event;
#endif

////////////////////////////////////
// declaracion de funciones
// HOST: funcion llamada desde el host y ejecutada en el host
__host__ void setEvent(event *ev)
/* Descripcion: Genera un evento de tiempo */
{
#ifdef __linux__
    gettimeofday(ev, NULL);
#else
    QueryPerformanceCounter(ev);
#endif
}

__host__ double eventDiff(event *first, event *last)
/* Descripcion: Devuelve la diferencia de tiempo (en ms) entre dos eventos */
{
#ifdef __linux__
    return
        ((double)(last->tv_sec + (double)last->tv_usec/1000000))-
        (double)(first->tv_sec + (double)first->tv_usec/1000000))*1000.0;
#else
    event freq;
    QueryPerformanceFrequency(&freq);
    return ((double)(last->QuadPart - first->QuadPart) / (double)freq.QuadPart) * 1000.0;
#endif
}

////////////////////////////////////
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
    // declaraciones
    float *hst_A, *hst_B, *hst_C;
    const int TAM = 1000; // Tamano de las matrices: TAMxTAM
    // reserva en el host
    hst_A = (float*)malloc(TAM*TAM*sizeof(float));
    hst_B = (float*)malloc(TAM*TAM*sizeof(float));
    hst_C = (float*)malloc(TAM*TAM*sizeof(float));

    // inicializacion
    srand((int)time(NULL));
    for (int i = 0; i<TAM*TAM; i++)
    {
        hst_A[i] = (float) rand() / RAND_MAX;
        hst_B[i] = (float) rand() / RAND_MAX;
        hst_C[i] = 0;
    }

    // TEMPORIZACION
    // Mediante eventos del sistema. Precision: fracciones de 1 ms.
    // La funcion setEvent() devuelve un evento de tiempo.

```



```
// La funcion eventDiff() calcula la diferencia de tiempo (en milisegundos) entre dos
eventos.

event start; // variable para almacenar el evento de tiempo inicial.
event stop; // variable para almacenar el evento de tiempo final.
.
double tiempo_ms;

// Iniciamos el contador
setEvent(&start); // marca de evento inicial

// Multiplicacion de matrices
printf("*****\n");
printf("TEMPORIZACION CPU\n");
printf("*****\n");
printf("Multiplicando matrices %dx%d...", TAM,TAM);
for (int i = 0; i<TAM; i++)
{
    for (int j = 0; j<TAM; j++)
    {
        hst_C[j + i*TAM] = 0;
        for (int k = 0; k<TAM; k++)
        {
            hst_C[j + i*TAM] += hst_A[k + i*TAM] * hst_B[j + k*TAM];
        }
    }
}
printf("listo!\n");

// Paramos el contador
setEvent(&stop); // marca de evento final

// Intervalos de tiempo
tiempo_ms = eventDiff(&start, &stop); // diferencia de tiempo en ms
printf("Tiempo de ejecucion: %.8f s\n", tiempo_ms/1000);

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



```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v8.0\ARCO\ARCO\..\bin\win64\Debug\practica.exe
*****
TEMPORIZACION CPU
*****
> Multiplicando matrices 1000x1000...listo!
> Tiempo de ejecucion: 4.48258488 s
pulsa INTRO para finalizar...
```

Figura 5.2.: Salida por pantalla de la ejecución del código de ejemplo.