



CUDA C/C++:
GUÍA DE INSTALACIÓN

César Represa Pérez

CUDA C/C++: *Guía de instalación*©2021

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++**.

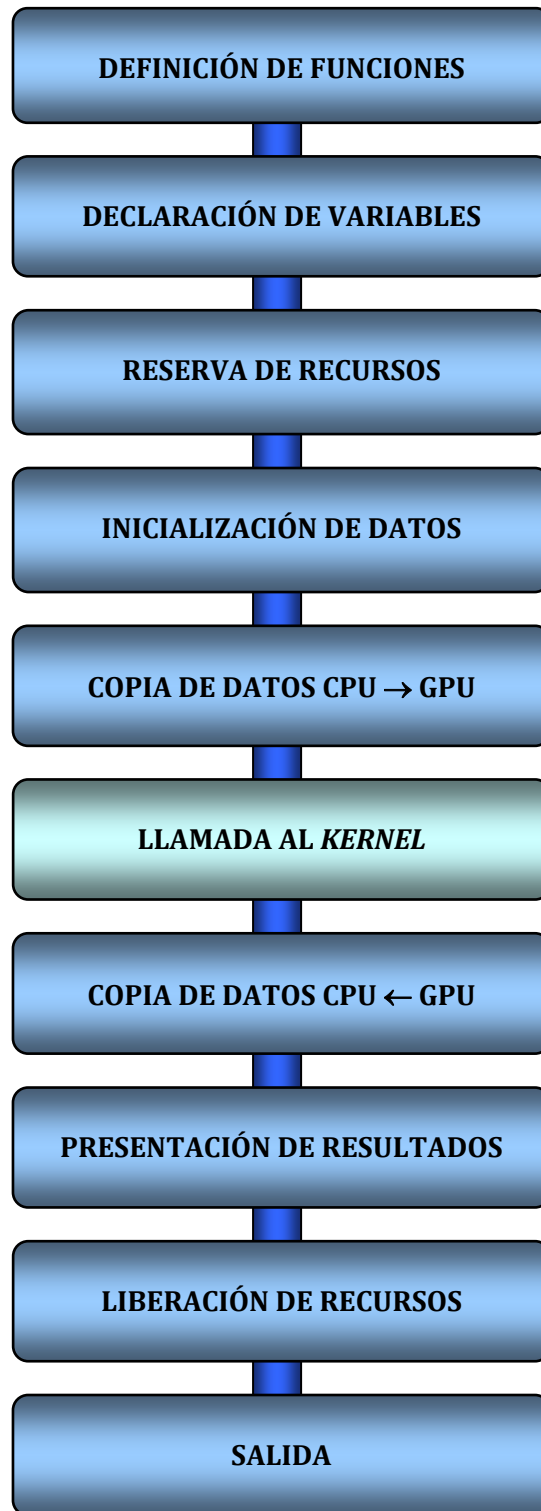
Existen APIs alternativas para desarrollar aplicaciones que utilicen la GPU de cualquier fabricante como acelerador de cálculo. Una de ellas es **OpenCL**, que es un estándar abierto y libre de derechos. Sin embargo, CUDA es la forma más fácil de empezar. De hecho, si fuera necesario utilizar una GPU 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.

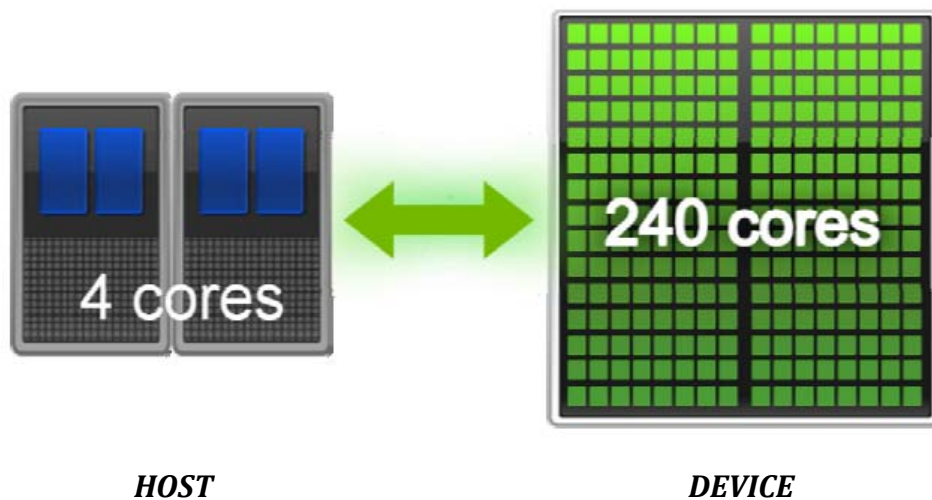


Figura i.1. Modelo de computación en sistemas heterogéneos: CPU + GPU.

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

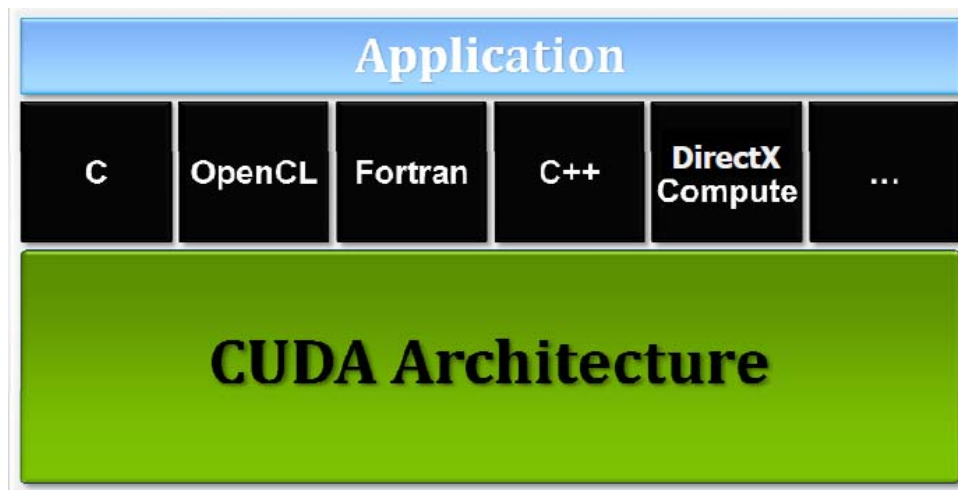


Figura i.2. CUDA está diseñado para soportar diferentes lenguajes de programación.

2. Arquitectura CUDA

En la arquitectura clásica de una tarjeta gráfica podemos encontrar la presencia de dos tipos de procesadores, los procesadores de vértices y los procesadores de fragmentos, dedicados a tareas distintas e independientes dentro del cauce gráfico y con repertorios de

instrucciones diferentes. Esto presenta dos problemas importantes, por un lado el desequilibrio de carga que aparece entre ambos procesadores y por otro la diferencia entre sus respectivos repertorios de instrucciones. De este modo, la evolución natural en la arquitectura de una **GPU** ha sido la búsqueda de una arquitectura unificada donde no se distinguiera entre ambos tipos de procesadores. Así se llegó a la arquitectura **CUDA**, donde todos los núcleos de ejecución necesitan el mismo repertorio de instrucciones y prácticamente los mismos recursos.

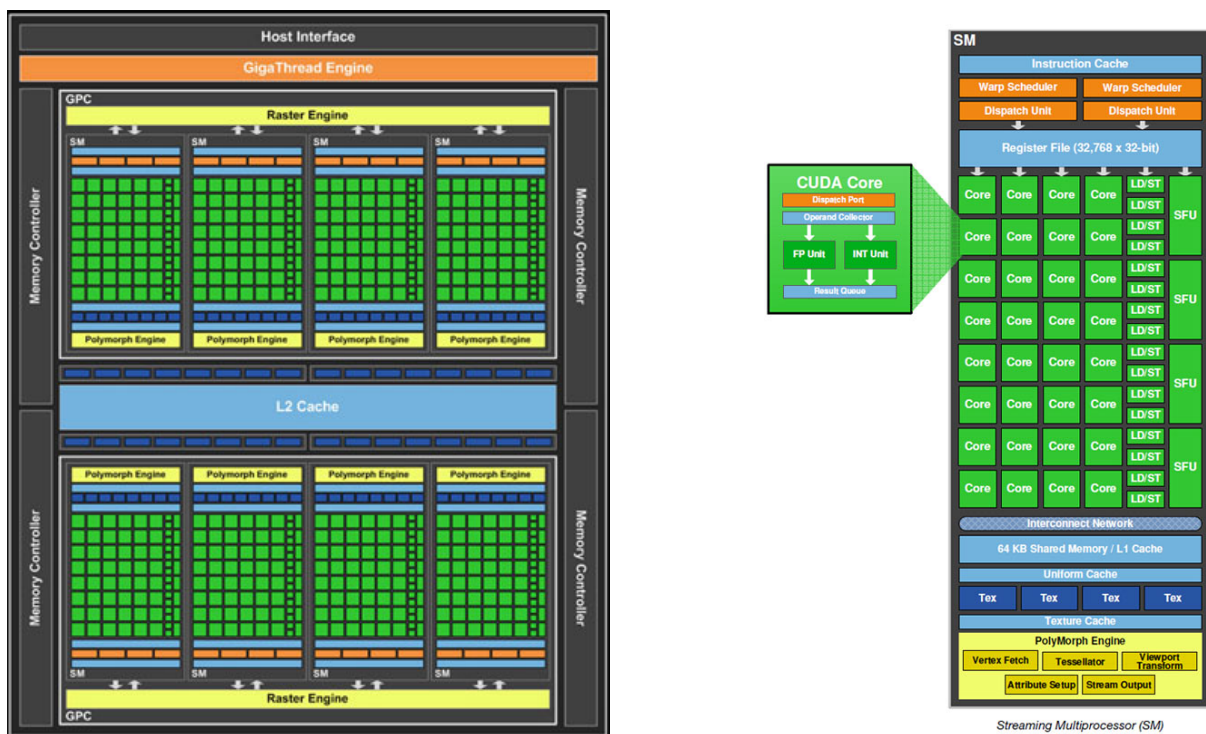


Figura i.3. Arquitectura básica de una tarjeta gráfica CUDA-enabled.

En la Figura i.3 se muestra la arquitectura básica 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)*.

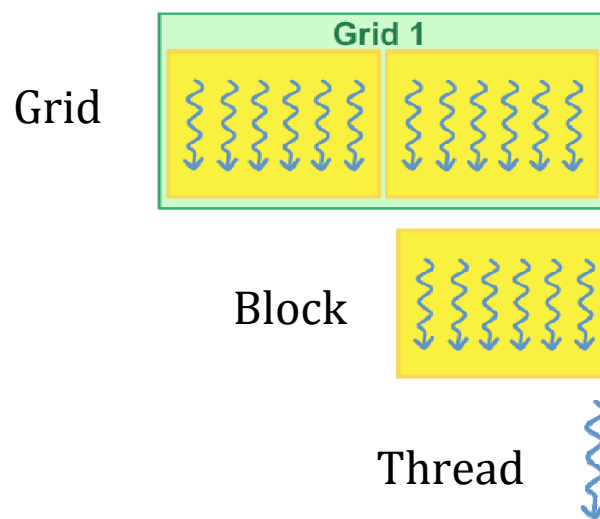


Figura i.4. Jerarquía de hilos en una aplicación CUDA.

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.

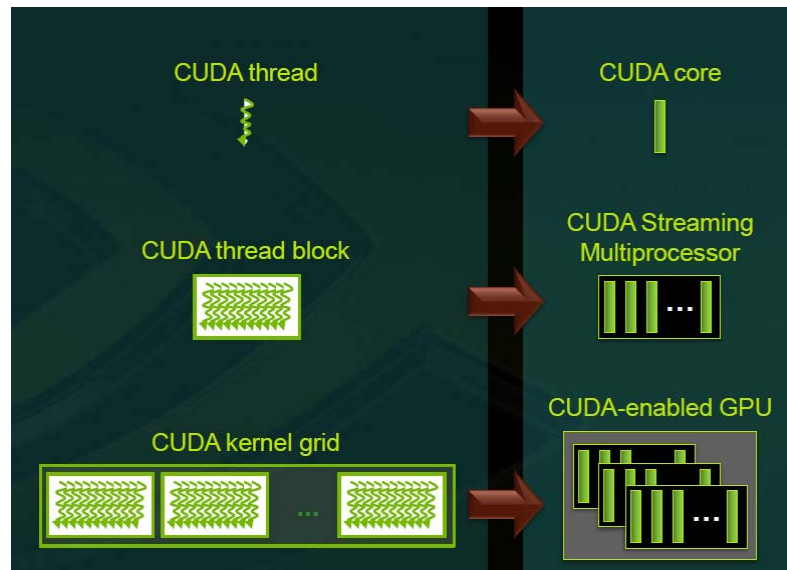


Figura i.5. Asignación de recursos en la ejecución de un *kernel*.

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.

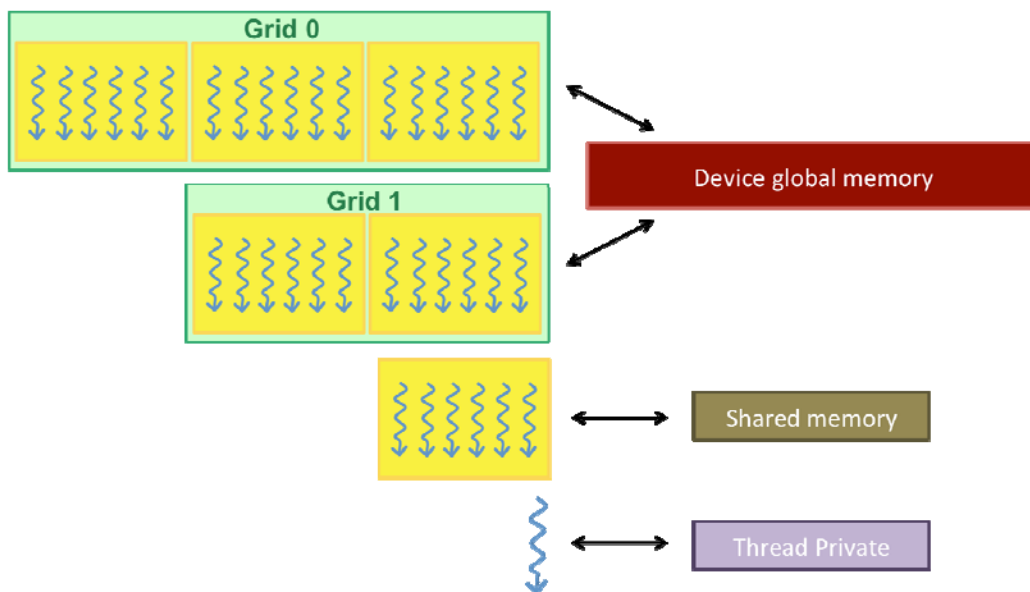


Figura i.6. Jerarquía de memoria dentro de la arquitectura CUDA.

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
7.0	Volta	64
7.5	Turing	64
8.0	Ampere	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**. El driver también debe ser compatible con la versión de CUDA. (ver **TABLA 4**).
- **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 – 5.3
7.5	2.0 – 5.3
8.0	2.0 – 6.2
9.2	3.0 – 7.2
10.2	3.0 – 7.5
11.0	5.0 – 8.0
¿?	¿?

TABLA 4. CUDA Toolkit y versiones del driver compatibles.

CUDA Toolkit	Linux x86_64 Driver Version	Windows x86_64 Driver Version
CUDA 11.0.194	>= 450.51.05	>= 451.48
CUDA 11.0.171 RC	>= 450.36.06	>= 451.22
CUDA 10.2.89	>= 440.33	>= 441.22
CUDA 10.1 (10.1.105 general release, and updates)	>= 418.39	>= 418.96
CUDA 10.0.130	>= 410.48	>= 411.31
CUDA 9.2 (9.2.148 Update 1)	>= 396.37	>= 398.26
CUDA 9.2 (9.2.88)	>= 396.26	>= 397.44
CUDA 9.1 (9.1.85)	>= 390.46	>= 391.29
CUDA 9.0 (9.0.76)	>= 384.81	>= 385.54
CUDA 8.0 (8.0.61 GA2)	>= 375.26	>= 376.51
CUDA 8.0 (8.0.44)	>= 367.48	>= 369.30
CUDA 7.5 (7.5.16)	>= 352.31	>= 353.66
CUDA 7.0 (7.0.28)	>= 346.46	>= 347.62

Como regla general podemos decir que una tarjeta con una capacidad de cómputo baja es más conveniente utilizarla con una versión de CUDA baja. Sin embargo, a veces esto supone utilizar una versión del driver que no sea la más reciente. En general, el driver incluido en el propio instalador suele ser el más indicado.

Por todo ello, es absolutamente recomendable seguir los pasos de instalación indicados en esta guía y leer las correspondientes “[Release Notes](#)” de la versión de CUDA que se desea instalar.

3. Instalación de las herramientas de desarrollo

Lo más importante: antes de la instalación del software de CUDA proporcionado por NVIDIA hay que 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 esta guía se van a describir los diferentes procesos de instalación para los sistemas basados en Windows utilizando versiones estables tanto de Microsoft Visual Studio (2013) como de CUDA (7.5).

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 ha dejado de estar soportada en actuales versiones de CUDA (ver **TABLA 5**).

TABLA 5. Versiones de Visual Studio soportadas por la versión 11.0 de CUDA.

Compiler*	IDE	Native x86_64	Cross (x86_32 on x86_64)
MSVC Version 192x	Visual Studio 2019 16.x (RTW and all updates)	YES	YES
MSVC Version 191x	Visual Studio 2017 15.x (RTW and all updates)	YES	YES
MSVC Version 1900	Visual Studio 2015 14.0 (RTW and updates 1, 2, and 3)	YES	YES
	Visual Studio Community 2015	YES	YES
MSVC Version 1800	Visual Studio 2013 12.0	YES	YES
MSVC Version 1700	Visual Studio 2012 11.0	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 e independientemente de la versión actual de CUDA 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).

The screenshot shows the NVIDIA website's download page for CUDA 7.5. The page has a dark header with the NVIDIA logo and navigation links: Downloads, Training, Ecosystem, Forums, Register Now, and Log in. Below the header is a banner for "GET TO KNOW NVIDIA PASCAL" with a "LEARN MORE" button. The main content area is titled "Select Target Platform" and contains a form with the following selections:

- Operating System: Windows
- Architecture: x86_64
- Version: 10
- Installer Type: exe (network)

To the right of the form is a "Documentation" sidebar with links to: CUDA Quick Start Guide, Release Notes, EULA, Online Documentation, CUDA Toolkit Overview, Download Checksums, and Legacy Toolkits. Below the form, there is a "Learn more about CUDA Toolkit 7.5, check out:" section with two links to blog posts and a note about reporting bugs.

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).

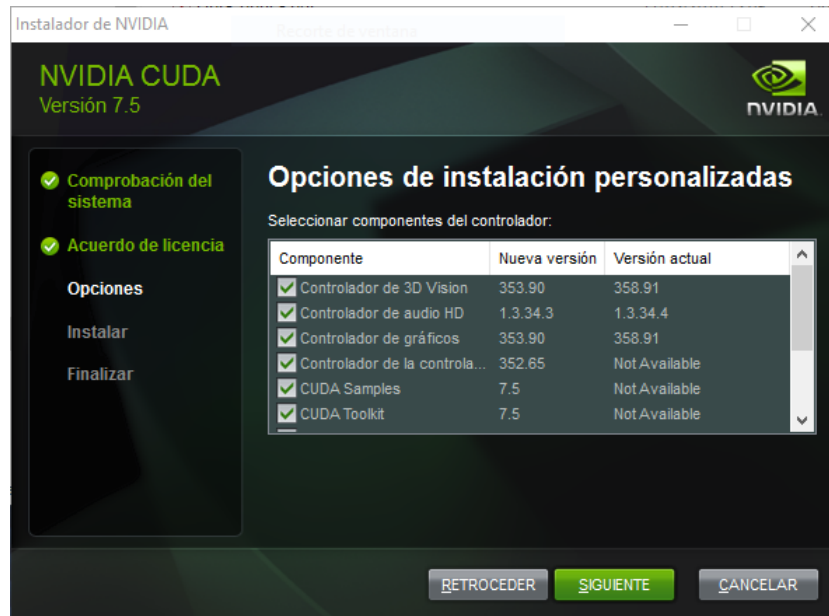


Figura a.2. Ejecución del instalador de CUDA v7.5.

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.

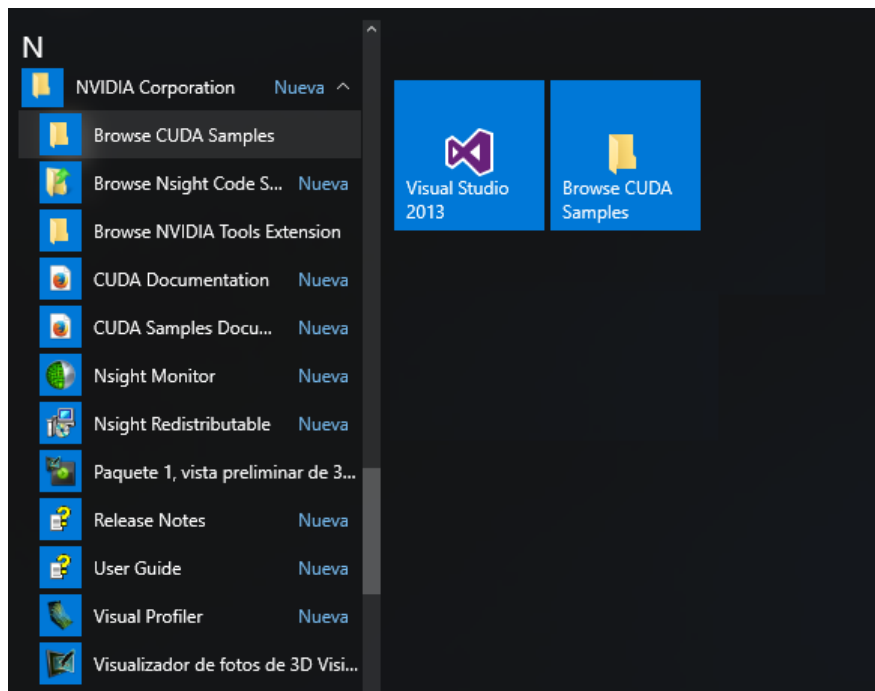


Figura a.3. Explorador NVIDIA Browse CUDA Samples.

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_Uilities	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_CUDALibraries	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

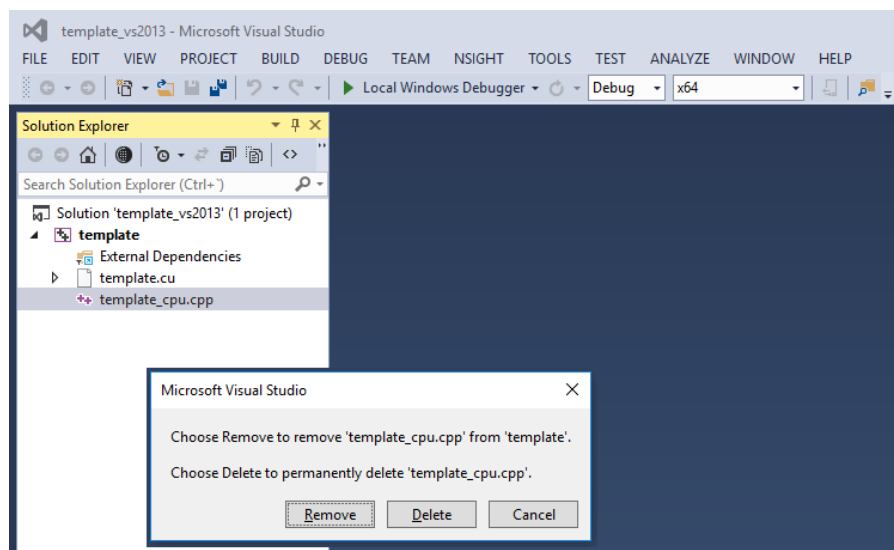
2. **Copiar** la carpeta `..\CUDA Samples\v7.5\0_Simple\template` dentro de nuestra nueva carpeta y renombrarla una vez copiada, como por ejemplo: `..\CUDA Samples\v7.5\practicas\p0`:

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

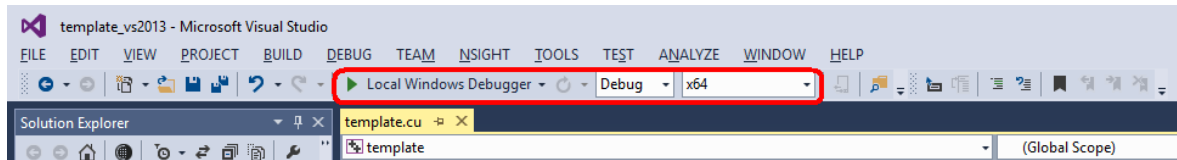
3. **Eliminar** los archivos de proyecto y solución que no correspondan a nuestra versión de Visual Studio (por ejemplo, para la versión 2013 nos quedaremos con los ficheros “template_vs2013.vcxproj” y “template_vs2013.sln”) y mantener el fichero fuente llamado “template.cu”, que es el único que debemos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

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.

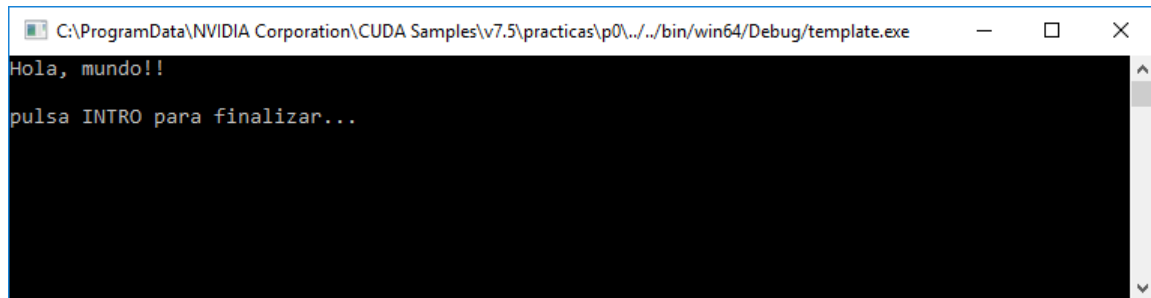


Figura a.4. Salida con el mensaje “*Hola, mundo!!*”.

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`”

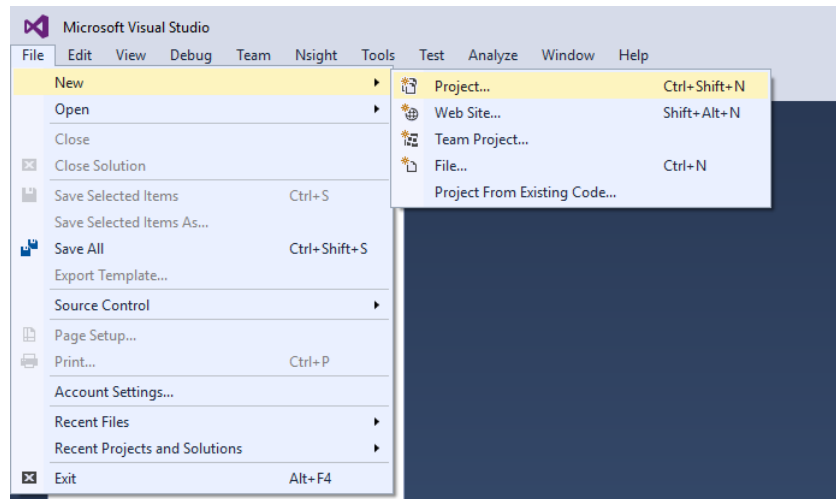
“`practica_vs2013.sln`”

“`practica_vs2013.vcxproj`”

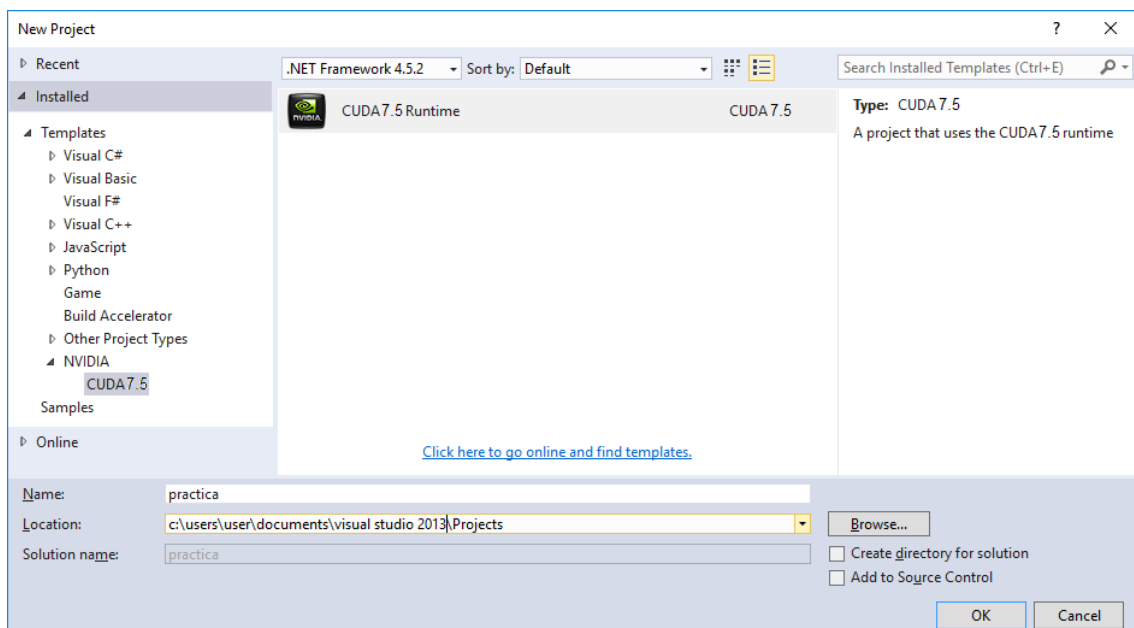
8. **Editar** el contenido de los archivos *.sln y *.vcxproj con un editor de texto y reemplazar todas la ocurrencias de la palabra “*template*” por la palabra escogida anteriormente (en nuestro ejemplo, por la palabra “*practica*”).

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 `CUDAxx.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):



ANEXO: Configuración de CUDA

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

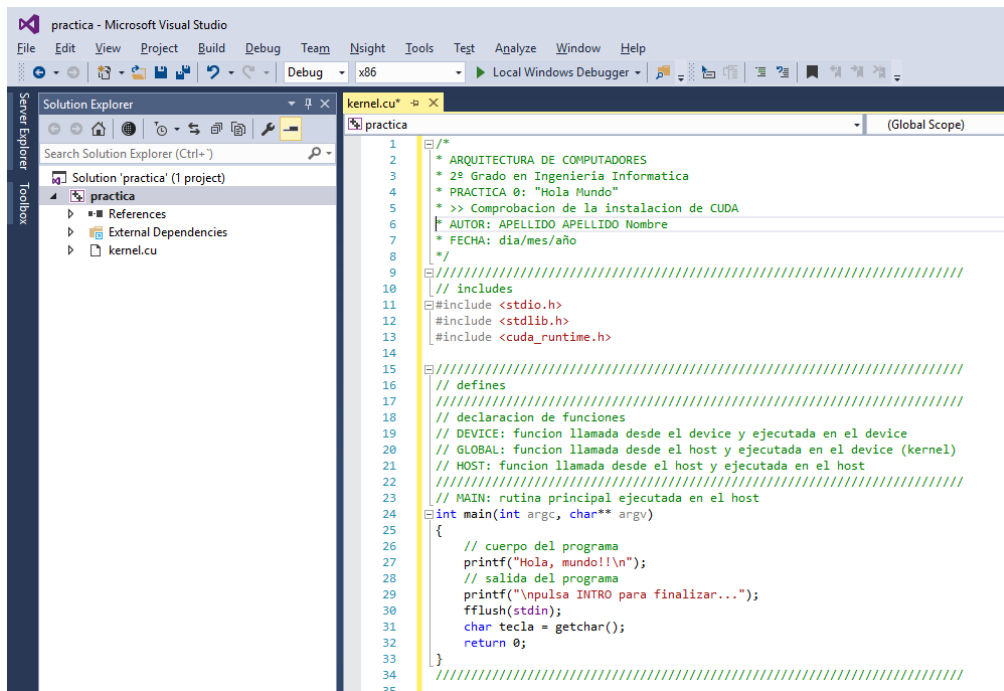
“kernel.cu”

“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*”:



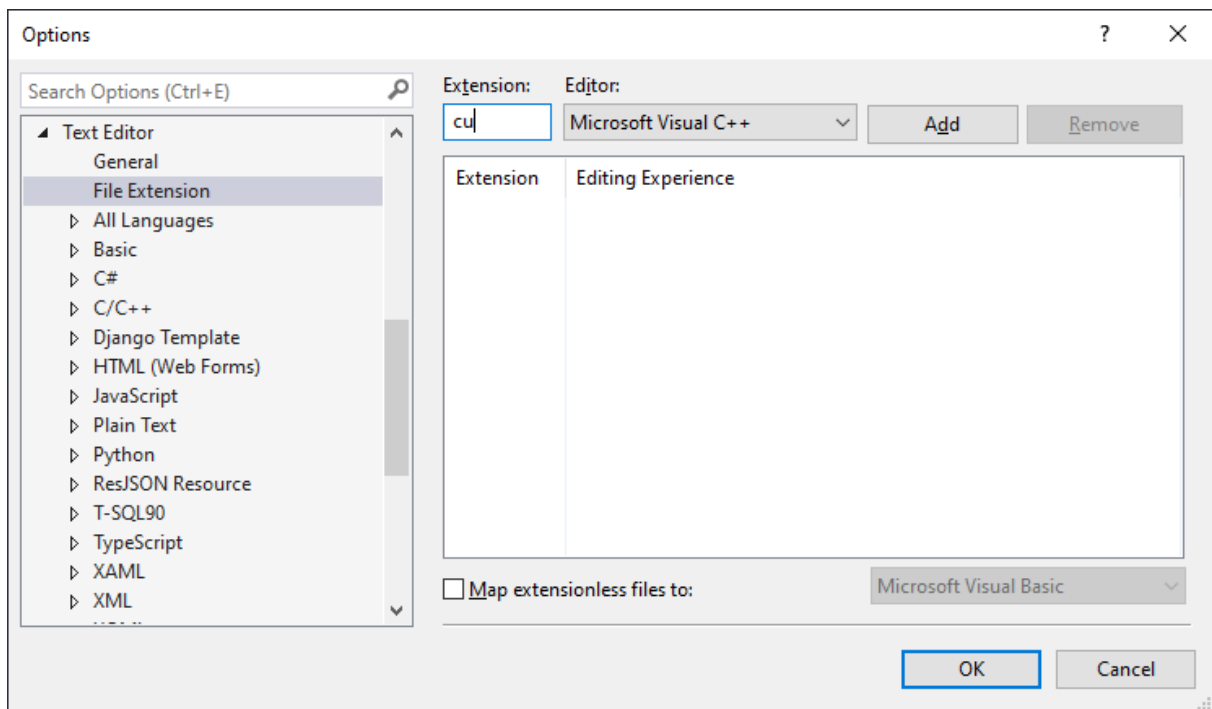
4.3. Sintaxis en color

Si queremos que en el editor de texto de Visual Studio se vea nuestro código fuente con códigos de color para facilitar la lectura y una mejor comprensión del mismo, lo que hay que hacer es que Visual Studio reconozca la extensión .cu.

Para ello hay que ir a la barra de herramientas y en “**Herramientas**” (*Tools*) buscar “**Opciones**” (*Options*), que es donde se pueden configurar muchas características del IDE y que muchas tienen que ver sólo con su apariencia.

En particular nos interesa que el “**Editor de Texto**” (*Text Editor*) en su sección “**Extensión de archivo**” (*File Extension*) reconozca la extensión `.cu` dentro de *Microsoft Visual C++*.

Sólo hay que añadir la extensión `cu` y la próxima vez que se reinicie Visual Studio aparecerán las palabras clave en color:



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 nos 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.

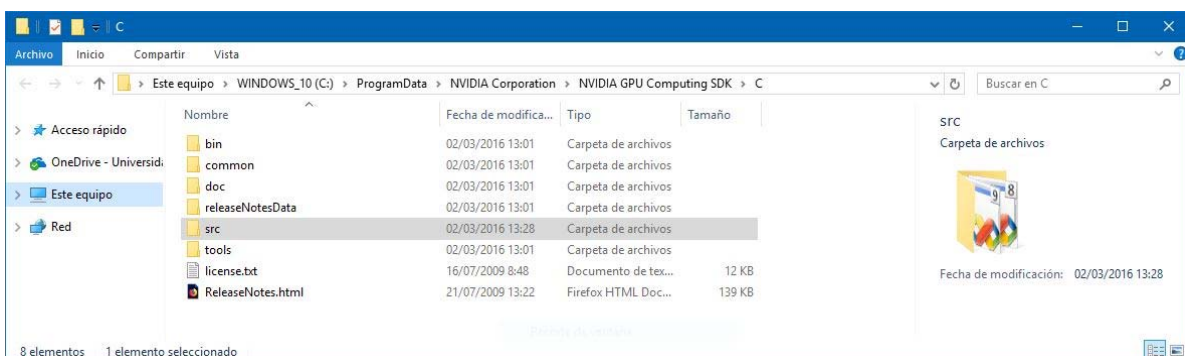
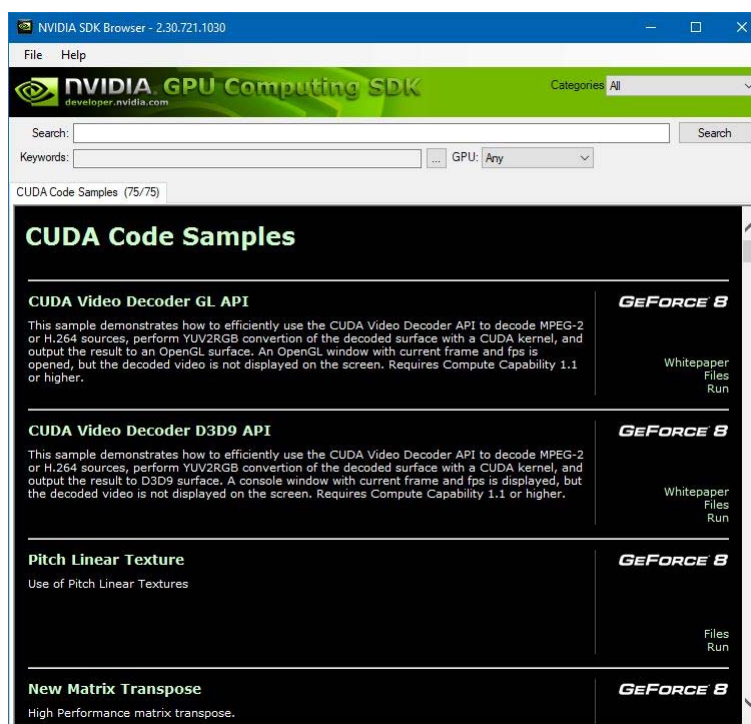
Por otro lado, para instalar versiones anteriores de CUDA ([Legacy Releases](#)), como por ejemplo la versión 2.3, hay que tener en cuenta que no hay un único archivo instalador sino que necesitamos instalar el **CUDA Toolkit** y el **CUDA SDK** por separado. En este caso, los pasos a seguir para instalar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

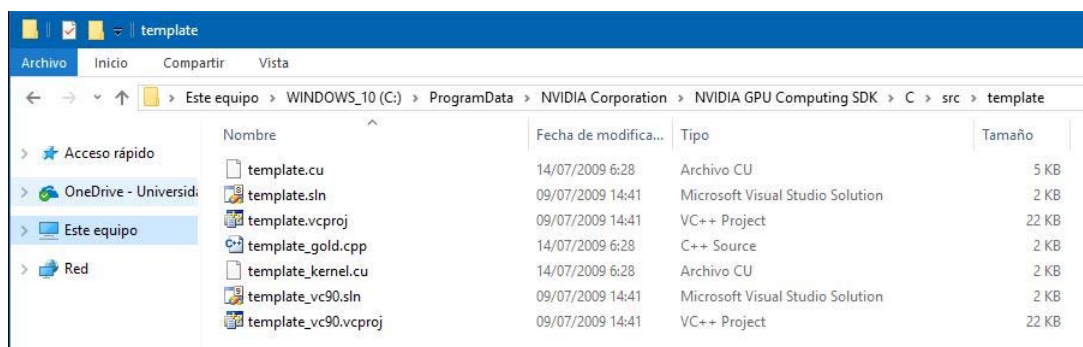


Los proyectos incluidos en el SDK están configurados con rutas relativas a la instalación. Por este motivo, si deseamos crear nuestra propia carpeta para generar código es necesario que ésta cuelgue de la carpeta `src`. Por ejemplo, nuestra carpeta `practica` estaría en la ruta:

```
C:\...\NVIDIA Corporation\NVIDIA GPU Computing SDK\C\src\practica
```

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 y copiarlos en nuestra carpeta de trabajo `practica`.



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 con el fin de sobrescribir en dicho archivo nuestro propio código.

Los ficheros fuente, proyecto y solución también se pueden cambiar de nombre sustituyendo la palabra “`template`” por otro nombre (por ejemplo, por la palabra “`practica`”). Esto implica que debemos editar el contenido de los archivos “.sln” y “.vcproj” siguiendo los pasos de los puntos 7 y 8 del apartado 4.1.

Finalmente nuestra carpeta de trabajo quedará con 3 archivos:

“practica.cu”

“practica_vc90.sln”

“practica_vc90.vcproj”

Por último, 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).

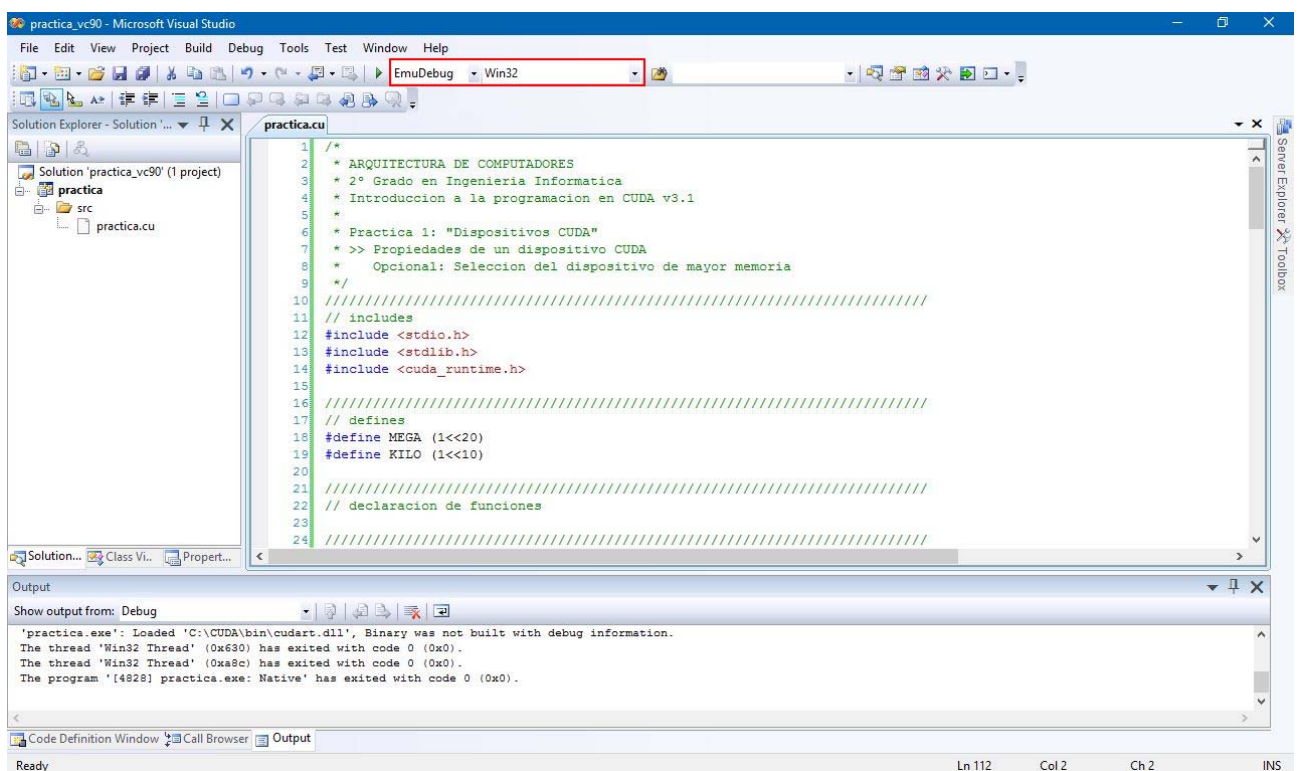


Figura a.5. Construcción de la solución en modo “*EmuDebug*” para Win32.

Si queremos que en el editor de texto de Visual Studio se vea nuestro código fuente con códigos de color tenemos que hacer que Visual Studio reconozca la extensión `.cu`, y para ello tenemos que seguir los pasos indicados en el apartado 4.3 “*Sintaxis en color*” (Figura a.6).

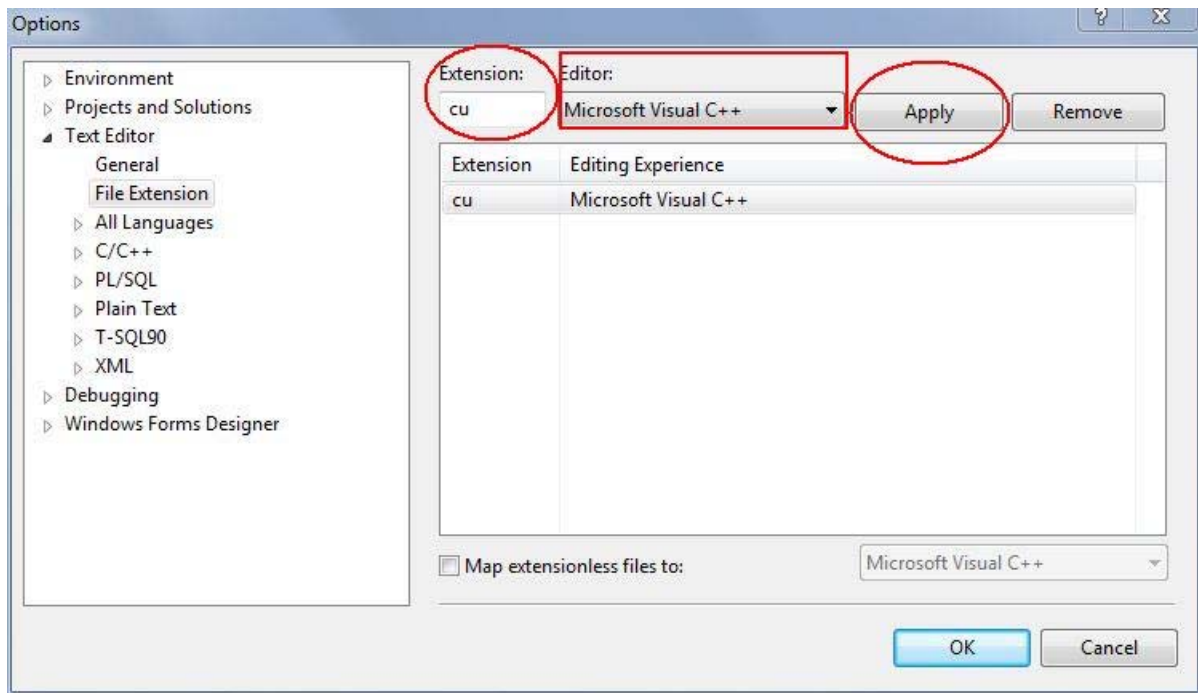


Figura a.6. Configuración para la sintaxis en color en Visual Studio.

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 Ingeniería Informática
 *
 * 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");

    // buscando dispositivos
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    if (deviceCount == 0)
        printf("ERROR: No se han encontrado dispositivos CUDA\n");
    else
        printf("Se han encontrado <%d> dispositivos CUDA\n", deviceCount);

    // 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 puede ser necesario para que VisualStudio 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 6**.

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 6. 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
 *
 */
///////////////////////////////////////////////////////////////////
// includes
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda_runtime.h>

///////////////////////////////////////////////////////////////////
// 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;
    const char *archName;
    switch (major)
    {
    case 1:
        //TESLA
        archName = "TESLA";
        cudaCores = 8;
        break;
    case 2:
        //FERMI
        archName = "FERMI";
        if (minor == 0)
            cudaCores = 32;
        else
            cudaCores = 48;
        break;
    case 3:
        //KEPLER
        archName = "KEPLER";
        cudaCores = 192;
        break;
    case 5:
        //MAXWELL
        archName = "MAXWELL";
        cudaCores = 128;
        break;
    case 6:
        //PASCAL
        archName = "PASCAL";
        cudaCores = 64;
        break;
    case 7:
        //VOLTA(7.0) //TURING(7.5)
        cudaCores = 64;
        if (minor == 0)
            archName = "VOLTA";
        else
            archName = "TURING";
        break;
    case 8:
        // AMPERE
        archName = "AMPERE";
        cudaCores = 64;
        break;
    default:
        //ARQUITECTURA DESCONOCIDA
        archName = "DESCONOCIDA";
    }
}

```

```

    int rtV;
    cudaRuntimeGetVersion(&rtV);
    // presentacion de propiedades
    printf("*****\n");
    printf("DEVICE %d: %s\n", deviceID, deviceProp.name);
    printf("*****\n");
    printf("> CUDA Toolkit          \t: %d.%d\n", rtV/1000,(rtV%1000)/10);
    printf("> Arquitectura CUDA      \t: %s\n", archName);
    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/(1024*1024));
    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 <%d> 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

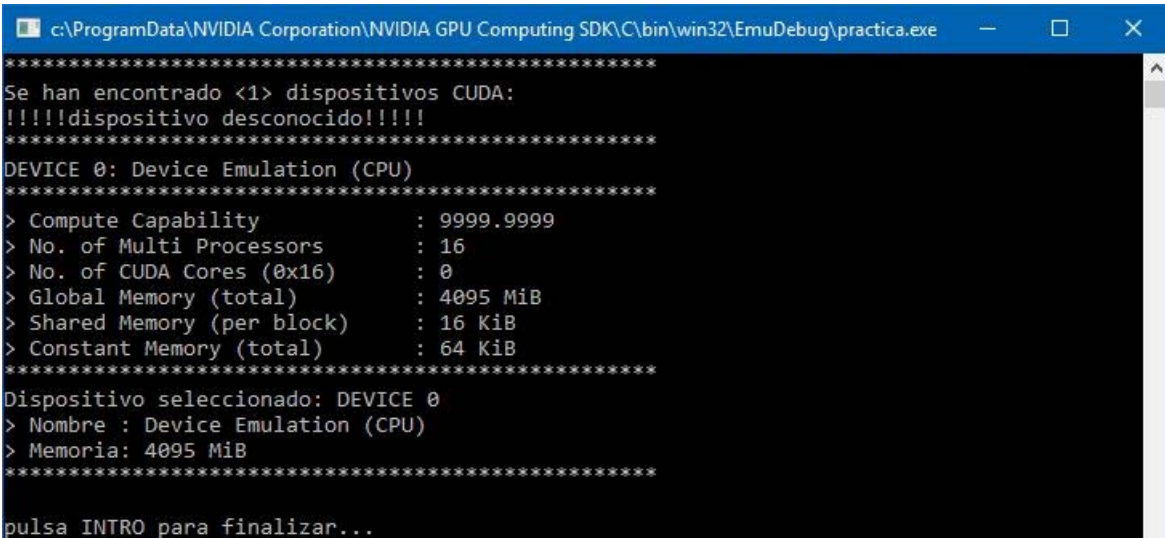
uso correcto de esta función es necesario pasar como argumentos una dirección de memoria (salida) y un número entero (entrada):

```
int main(int argc, char** argv)
{
    // ...
    cudaDeviceProp propiedades;
    int deviceID;
    int cudaGetDevice(&deviceID)
    // ...
    cudaGetDeviceProperties(&propiedades, deviceID);
    // ...
    printf("\nDEVICE %d: %s\n", deviceID, 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.