Dispositivos e Infraestructuras para Sistemas Multimedia 2018

Introducción a la Programación de GPUs usando CUDA

Práctica 1: Suma de Vectores

Alberto García García
Sergio Orts Escolano
José García Rodríguez

8 de noviembre de 2018

Proudly made with LaTeX

Índice general

Capítulo 1

Práctica 1: Suma de Vectores

En esta primera práctica introductoria realizaremos una suma de vectores. El código fuente para la práctica consiste en un único archivo `suma_vectores.cu`, el cual no compila inicialmente y tendrá que ser completado para implementar la funcionalidad deseada. Los objetivos de esta primera práctica son los siguientes:

- Aprender a crear un proyecto CUDA con Visual Studio.
- Conocer el flujo típico de un programa CUDA.
- Implementar e invocar un kernel sencillo.
- Utilizar los índices de hilos y de bloques en el kernel para distribuir trabajo entre los hilos.
- Conocer limitaciones de la arquitectura mediante el uso de deviceQuery.

1.1. Creación de Proyecto CUDA con Visual Studio

El primer paso para la práctica consiste en la creación de un proyecto CUDA empleando Visual Studio. Para esta demostración emplearemos la última versión de Visual Studio (2015 Update 3) así como la última de CUDA (8.0). Es importante instalar en primer lugar el entorno Visual Studio 2015 Update 3, así como los archivos necesarios para la creación de proyectos C++ (la forma más sencilla de asegurarnos de que estos componentes están instalados es tratar de crear un proyecto C++ ya que si no encuentra los componentes, Visual Studio automáticamente tratará de instalarlos). Una vez instalados tanto Visual Studio como los componentes de desarrollo de proyectos C++, procederemos a instalar CUDA 8.0. Siguiendo este orden, CUDA 8.0 además instalará una serie de componentes para permitir a Visual Studio crear proyectos CUDA 8.0 como veremos a continuación.

Para crear un nuevo proyecto CUDA procederemos a utilizar la opción `File` de la barra de herramientas de Visual Studio, seguidamente `New` y `Project` tal y como se muestra en la Figura 1.1. Esto nos abrirá una nueva ventana/asistente de creación de proyectos.

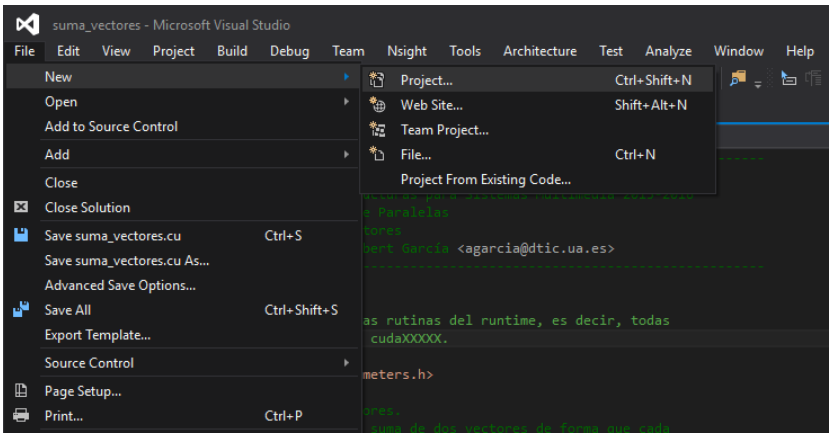

Figura 1.1: Creación de nuevo proyecto con Visual Studio en la barra de herramientas.

El asistente de creación de proyectos mostrado en la Figura 1.2 nos permite escoger diferentes plantillas para diversos lenguajes (Basic, C++, Python...) entre ellos, en la opción NVIDIA, tendremos CUDA 8.0. Simplemente seleccionaremos esta opción y escribiremos un nombre para la solución del proyecto en el cuadro correspondiente.

Tras seleccionar esta opción, Visual Studio creará una carpeta en `USER/Documents/Visual Studio 2015/Projects/` con el nombre que le hayamos dado a la solución. Inicialmente esa carpeta contendrá, además de los archivos propios de Visual Studio para la solución, un archivo llamado kernel.cu (extensión típica de los ficheros CUDA) con un ejemplo plenamente funcional de una suma de vectores sencilla en CUDA. Pulsando `Ctrl + Shift + B` compilaremos el proyecto y con `F5` arrancaremos el mismo con el Debugger (o bien en modo de ejecución Release si así se ha configurado).

Para limpiar el proyecto, ya que comenzaremos desde cero, deberemos utilizar el explorador de soluciones en la barra derecha para seleccionar el archivo kernel.cu y lo eliminaremos permanentemente (botón derecho, eliminar o suprimir). Seguidamente, podemos o bien añadir nuevos archivos fuente al proyecto vacíos o podemos importar archivos existentes al proyecto si hacemos click derecho en el mismo y utilizamos la opción `Add > Existing item...` o pulsamos `Shift + Alt + A`. Tal y como se muestra en la Figura 1.3.

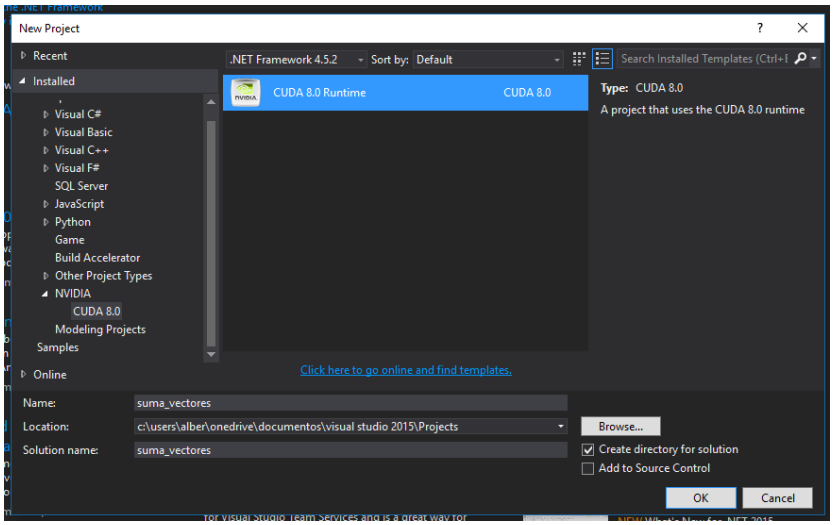

Figura 1.2: Selección de plantilla NVIDIA CUDA 8.0 en el asistente de creación de proyectos.

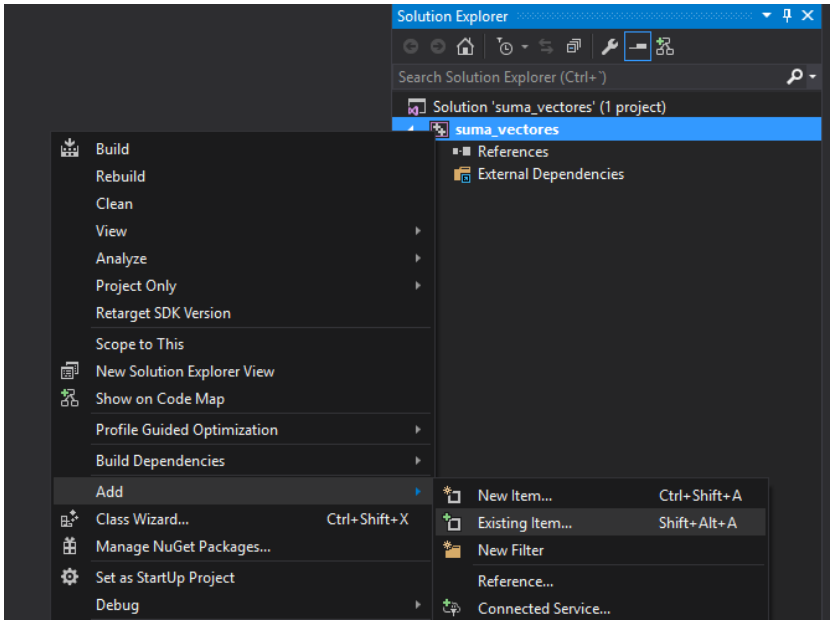

Figura 1.3: Añadiendo elemento existente al proyecto CUDA.

1.2. Flujo de Ejecución del Programa CUDA de Suma de Vectores

El flujo de ejecución del programa suma de vectores es un ejemplo típico del flujo de procesamiento tradicional realizado en CUDA, el cual consiste de cuatro pasos básicos (ilustrados en la Figura 1.4):

1. Copia de los datos a procesar de la memoria de la CPU a la de la GPU.
2. Lanzamiento del kernel u operaciones a realizar en la GPU (orden emitida por la CPU).
3. Ejecución de hilos del kernel en paralelo en los diferentes cores.
4. Copia de los datos resultantes de la memoria de la GPU a la de la CPU.

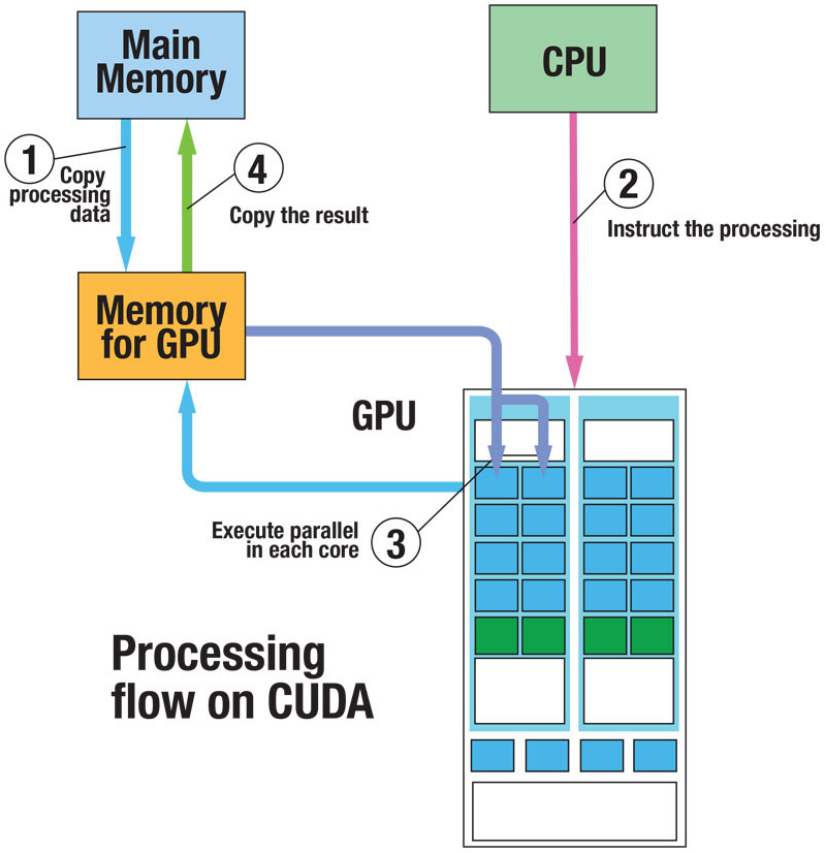

Figura 1.4: Flujo de procesamiento en CUDA.

En este caso particular, tal y como se mostrará en el Ejercicio 1 de forma guiada, nuestro programa se compone de los siguientes pasos en mayor detalle:

1. Inicialización y selección del dispositivo CUDA con `cudaSetDevice`.
2. Declaración y reserva de vectores A, B y C en CPU con `malloc`.
3. Inicialización de vectores A y B con valores arbitrarios en CPU.
4. Declaración y reserva de vectores A, B y C en GPU con `cudaMalloc`.
5. Transferencia de datos de los vectors A y B en CPU a los vectores A y B en GPU con `cudaMemcpy`.
6. Establecimiento de parámetros de lanzamiento e invocación del kernel de suma de vectores.
7. Ejecución de la suma en paralelo, en su versión básica cada hilo computa una suma de un par de elementos.
8. Transferencia de datos del vector resultado C en GPU al vector C en CPU con `cudaMemcpy`.
9. Verificación de resultados.
10. Liberación de memoria reservada en CPU y GPU con `free` y `cudaFree` respectivamente.
11. Reseteo del estado del dispositivo CUDA con `cudaDeviceReset`.

1.3. Ejercicios

A continuación se presentan los ejercicios evaluables de esta práctica. Como norma de entrega, la práctica debe explicarse y mostrarse funcionando durante las horas de prácticas en el laboratorio siendo la fecha límite la última sesión de prácticas. Los ejercicios a realizar son descritos a continuación junto con su puntuación:

1.3.1. Ejercicio 0 (1 punto): Familiarización con el Entorno

Familiarizarse con la creación de proyectos CUDA empleando Visual Studio, composición de un proyecto, opciones de compilación para NVCC, ejemplos existentes en el Toolkit de CUDA . Este ejercicio será guiado durante la sesión de prácticas.

1.3.2. Ejercicio 1 (2 puntos): Flujo e Invocación del Kernel

Editar el punto de entrada del programa `main(...)` en el fichero `suma_vectores.cu` para completar la funcionalidad de la suma de vectores una vez implementado el kernel. Los pasos a completar son aquellos mencionados anteriormente en el fujo de ejecución del programa GPU de suma de vectores, este ejercicio será guiado durante la sesión de prácticas:

1. Declarar los arrays para entrada y salida tanto en CPU como en GPU utilizando `cudaMalloc(...)`.

2. Transferir datos de los arrays en CPU a los arrays en GPU utilizando `cudaMemcpy(...)`.

3. Definir los parámetros de invocación del kernel (dimensión de bloque y dimensión de grid) e invocar al kernel previamente implementado utilizando esos parámetros.

4. Transferir los resultados del array en GPU al array en CPU empleando `cudaMemcpy(...)`.

5. Verificar resultados y liberar memoria CPU y GPU con `cudaFree(...)`.

1.3.3. Ejercicio 2 (2 puntos): Implementación de un Kernel Básico

Editar la función `suma_vectores(...)` en el fichero `suma_vectores.cu` para completar el kernel e implementar la funcionalidad de la suma de vectores en la GPU. En esta primera versión se espera que cada hilo de ejecución en la GPU sume un único elemento de los vectores y deposite el resultado en el vector de salida en la posición correspondiente. Así pues, el ejercicio se puede reducir a dos pasos:

1. Definir los índices del elemento a ser sumado por cada hilo, empleando las variables especiales de CUDA que permiten obtener los índices de hilo (`threadIdx`) y de bloque (`blockIdx`).

2. Sumar las posiciones de los vectores de entrada correspondientes al hilo en cuestión, teniendo en cuenta que cada hilo va a computar la suma de un elemento, y depositar el resultado en la posición adecuada del vector de salida.

Tras realizar los pasos, ejecutad el programa y comprobar que la salida es correcta.

1.3.4. Ejercicio 3 (2.5 puntos): Número de Elementos Arbitrario

Modificar el número de elementos en los vectores de entrada y comprobar de nuevo la salida del programa. Como se podrá comprobar, el resultado es incorrecto. Este hecho se debe a que anteriormente el número de elementos a sumar era divisible por el número de hilos por bloque. Si modificamos el número de elementos para utilizar uno no divisible por el número de hilos por bloque la consecuencia será que se lanzarán menos bloques de los necesarios (si truncamos la división o redondeamos a la baja) por lo que ciertos elementos no serán sumados o bien lanzaremos bloques de más (si redondeamos al alza) y computaremos elementos fuera del rango de memoria de los vectores. En cualquier caso, se produce una situación incorrecta cuya salida puede ser errónea o tener consecuencias inesperadas por lo que se deberán realizar cambios tanto en la invocación del kernel como en la propia función para evitar esta situación:

1. Modificar los parámetros de invocación del kernel para lanzar suficientes hilos o bloques.

2. Modificar el kernel para evitar accesos de memoria fuera de rango.

Tras realizar los pasos indicados, ejecutar el programa y verificar que la salida es correcta para tamaños de vector arbitrarios.

1.3.5. Ejercicio 4 (2.5 puntos): Tamaño Considerable

Modificar el número de elementos en los vectores de entrada para que se puedan sumar vectores de un tamaño considerable, por ejemplo, más de diez millones de elementos, y comprobar de nuevo la salida del programa. Como se puede comprobar, la salida del programa es incorrecta, produciéndose un error de invocación del kernel. Este error se debe a la limitación existente en cuanto al tamaño máximo tanto de bloque como de grid dependiendo de la generación de GPU que estemos empleando. Así pues, dado que tenemos una limitación en el tamaño de los bloques y del grid, tenemos que prestar atención a los límites de la tarjeta que estemos empleando y reformular nuestro código en consecuencia:

1. Abre, compila y ejecuta el ejemplo de CUDA `deviceQuery` para conocer los límites de la tarjeta que estés utilizando, en concreto el tamaño máximo de bloque y de grid.

2. Encuentra una forma de solventar esta limitación haciendo que cada hilo compute más de un elemento del vector en lugar de uno solo. Para ello tendrás que modificar tanto el kernel como los parámetros de invocación. Se valorará la flexibilidad y elegancia de la solución de cara a conseguir la máxima puntuación en este ejercicio.