UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Programación de GPUs con CUDA y OpenCL (PGPU)

Tarea 1

Introducción

Para compilar los ejercicios se puede utilizar el siguiente comando:

```
nvcc -lineinfo -Xptxas=-v -arch=sm_60 -o ejercicio ejercicio.cu
```

Antes de comenzar conviene situarse delante de la máquina que tenemos entre manos. Para ello vamos a seguir los siguientes pasos:

1. Comenzamos utilizando el comando `nvidia-smi`. Este comando da una idea del tipo de GPU de nuestro equipo, incluso muestra si hay más de una GPU.

2. El segundo paso consiste en averiguar qué GPU(s) hay en mi ordenador y las características que tienen. Para ello, utilizaremos un ejemplo de NVIDIA que viene con el SDK[1] de CUDA. Este ejemplo consiste en un sencillo programa llamado `deviceQuery` que hace uso de una rutina de librería llamada `cudaGetDeviceProperties` que accede a todas las GPUs NVIDIA disponibles en el host y rellena una estructura de tipo `cudaDeviceProp`. Su esquema es básicamente el siguiente:

```
1    int deviceCount = 0;
2    for (dev = 0; dev < cudaGetDeviceCount(&deviceCount); ++dev)
     {
3        cudaDeviceProp deviceProp;
4        cudaGetDeviceProperties(&deviceProp, dev);
5        /* Show properties accesing members of deviceProp */
6    }
```

Código 1: CUDA structure `cudaDeviceProp`.

Cada miembro de esta estructura de C contiene una información dada sobre la GPU. Sigue estos pasos:

a) Compila el programa `deviceQuery` escribiendo `make`.

b) Ejecuta el programa: `./deviceQuery`.

[1] Software Development Kit.

c) Después de unos segundos verás la información adquirida por la herramienta. Analiza cada elemento y presta atención a elementos especiales como: Capacidad, números de la versión de la GPU, cantidad de memoria global, Multiprocesadores (SM) y núcleos, tamaño de *warp* y número máximo de núcleos por bloque, dimensión de bloques y cuadrícula (`grid`).

Ejercicio 1

En este ejercicio vamos a construir una infraestructura que vamos a utilizar en lo sucesivo para el resto de ejercicios. El código facilitado en el fichero `SumaVectores.cu` contiene dicho esqueleto. Los siguientes pasos van a consistir en el rellenado de dicho esqueleto.

1. La función `main` está completamente implementada. Se llama a la función `vector_sum`, que suma los dos vectores de tamaño `n` pasados como segundo y tercer argumentos, respectivamente, y devuelve el vector sumado como último argumento. Esta función servirá para comprobar que el resultado es correcto. La implementación es trivial y ya se encuentra implementada.

2. La función `cu_vector_sum` debe hacer lo propio en la GPU. En esta función se distinguen los punteros a la memoria de la CPU (precedidos por `h_`) de aquellos que apuntan a la memoria de la GPU (precedidos por `d_`). Véase que hay declarados tres punteros a GPU: `d_a`, `d_b` y `d_c`. Lo primero que hay que hacer es reservar memoria, apuntada por dichos punteros, en GPU mediante la función `cudaMalloc` en el lugar indicado.

3. Copiar los vectores de CPU `h_a` y `h_b` en GPU, es decir, en los vectores `d_a` y `d_b`, respectivamente, mediante la función `cudaMemcpy`.

4. Calcular los bloques de threads totales que se necesitan para "cubrir" los `n` elementos a sumar de los vectores sobre la variable `nblocks`. Por ejemplo, si `n=1300` y `blocksize=32`, el número total de bloques será de `nblocks=41`.

5. A continuación se crean dos variables, `dimGrid` y `dimBlock`, de tipo `dim3` para representar la malla de bloques y el tamaño de bloque de threads. En caso de que la malla de bloques (o del tamaño de bloque) sean "lineales" o 1D no es necesario crear esta variable pero, si lo hacemos así, nos servirá para el futuro.

6. En este paso se debe llamar al kernel mediante la notación

   ```
   nombre_de_kernel<<<...>>>( argumentos ).
   ```

   Obsérvese que al principio del fichero puede encontrarse la implementación inacabada de un kernel (`compute_kernel`). Se debe implementar dicho kernel utilizando los argumentos especificados. Para ello, hay que tener en cuenta que cada thread debe encargarse de sumar dos elementos de los vectores `d_a` y `d_b`, respectivamente, en un elemento del vector `d_c`. Para ello, hay que tener en cuenta que cada thread puede encargarse del elemento de índice

   ```
   indice = threadIdx.x + blockDim.x * blockIdx.x.
   ```

   Algo importante a tener en cuenta es que los threads "cubren" un espacio mayor que la memoria reservada para los vectores (`n`), por lo que se deberá evitar que ningún thread acceda a posiciones de memoria que no se han reservado previamente.

7. Ahora, de vuelta a la función `cu_vector_sum` y después de la llamada al kernel realizamos una copia del vector `d_c` en GPU a la CPU, o sea, en el vector `h_c` mediante la función `cudaMalloc`.

8. Terminamos la implementación de la función `cu_vector_sum` liberando la memoria creada en GPU mediante la función `cudaFree`. Este paso de liberar memoria es importante realizarlo dado que puede servir para detectar errores que podrían pasar desapercibidos.

Ejercicio 2

En este ejercicio se va a realizar la suma de dos matrices. En realidad, se trata de una generalización de la suma de vectores anterior solo que a dos dimensiones, aunque se trata más de una cuestión conceptual. La parte importante y diferenciada es que ahora vamos a trabajar con bloques de threads bidimensionales y mallas de bloques también bidimensionales.

Lo primero que vamos a tratar es de la representación de los datos. Las matrices bidimensionales en C pueden declararse con dos indirecciones de manera que el acceso al elemento i, j de la matriz A se realizaría así: `A[i][j]`. Sin embargo, nosotros vamos a utilizar otra manera. Las matrices se almacenarán en un array unidimensional. Sea pues una matriz matemática $A \in \mathbb{R}^{m \times n}$, la declaración de la misma en C la realizamos de la siguiente manera:

```
float *A = (float*) malloc ( m*n*sizeof(float) );
```

Para nosotros, las filas de la matriz A se almacenarán consecutivamente en memoria, es decir, el elemento $A_{i,j+1}$ se encuentra a continuación del elemento $A_{i,j}$. Para más sencillez, utilizaremos la siguiente notación para acceder al elemento $A_{i,j}$, por ejemplo, para asignarle un valor:

```
A( i, j ) = 4.1;
```

Evidentemente, lo anterior no es notación C. Para que funcione, es necesario que estén definidas las macros correspondientes, en este caso:

```
#define A(i,j)          A[ (j) + ((i)*(n)) ]
```

El fichero `SimpleMatrixSum.cu` contiene las macros necesarias ya definidas. Hay que prestar atención a esta definición, es decir, si quisiéramos que las matrices estuviesen almacenadas "por columnas", por ejemplo, porque queremos utilizar las bibliotecas BLAS/LAPACK, la macro tendría el siguiente aspecto:

```
#define A(i,j)          A[ (i) + ((j)*(m)) ]
```

Esta manera de trabajar es cómoda pero tiene inconvenientes ya que debe haber coherencia entre la longitud de las filas (columnas) de la matriz declarada y de la definición de su macro correspondiente, lo que suele generar errores. También es necesario declarar una macro por matriz.

A continuación realizaremos los siguientes pasos sobre el fichero anterior:

1. La función principal se encuentra implementada. Pasamos a la función `cu_matrix_sum` que contiene la construcción de la infraestructura necesaria para llamar al kernel que resolverá el problema: `compute_kernel`. Implementamos las partes indicadas: reserva de memoria, transferencia de datos, cálculo de la malla de bloques y llamada al kernel. Es importante observar que la malla de bloques de threads debe "cubrir" completamente el espacio de `m`×`n` elementos que ocupan las matrices a sumar.

2. Implementación del kernel. En este punto hay que tener en cuenta que un thread tiene dos coordenadas dentro del bloque bidimensional de threads (`threadIdx.x` y `threadIdx.y`) y, además, cada bloque de threads tiene sus coordenadas dentro de la malla bidimensional de bloques de threads (`blockIdx.x` y `blockIdx.y`). Teniendo en cuenta que las dimensiones de los bloques son `blockDim.x` y `blockDim.y` para las dos dimensiones, respectivamente, podemos deducir fácilmente que cada thread tiene acceso a un elemento de la matriz `d_A` (o a un elemento de las otras dos, ya que todas se han almacenado de la misma manera en la GPU) utilizando, por ejemplo, las siguientes coordenadas:

```
int i = threadIdx.x + blockDim.x * blockIdx.x;
int j = threadIdx.y + blockDim.y * blockIdx.y;
```

Una vez implementado y probado el programa, hay que pasar al análisis correspondiente para ver si el alineamiento en memoria que hemos utilizado es adecuado. Este análisis se puede realizar con `computeprof` o con `nvprof`. Tomamos nota del tiempo que tarda en ejecutarse el kernel.

A continuación realizamos otra implementación del mismo kernel, esta vez ocupándonos de que los threads que tienen posiciones consecutivas en la dirección `x` accedan a posiciones consecutivas en memoria. Por ejemplo, el thread cuyo valor `threadIdx.x=17` debe acceder a la posición siguiente en memoria a la que accede el thread `threadIdx.x=16`. Una vez realizado esto se debe volver a analizar la aplicación con el profiler para ver si el tiempo de ejecución del kernel ha cambiado.

Ejercicio 3

El siguiente ejercicio sirve de repaso. Se parece mucho al primer ejercicio. La idea aquí consiste en implementar la operación `saxpy`. Esta operación viene definida en la biblioteca BLAS y se define de la siguiente manera. Dados dos vectores $x, y \in \mathbb{R}^n$ y un escalar $a \in \mathbb{R}$ la operación `saxpy` tiene la forma

$$y = a \cdot x + y.$$

El fichero `cu_saxpy.cu` contiene el esqueleto necesario para realizar la implementación de dicha operación en GPU. Dado que se trata de dos vectores lineales (1D) no va a ser necesario crear una malla bidimensional de bloques rectangulares de threads, será suficiente con crear bloques de threads 1D y una malla 1D de bloques de threads.

Ejercicio 4

Este ejercicio resuelve el mismo problema que el ejercicio anterior. En este caso, vamos a tratar la situación en la que la malla de threads no "cubre" todo el tamaño de los vectores. El código facilitado (`cu_saxpy1.cu`) pide tanto el tamaño de bloque de threads como el tamaño de malla o número de bloques. El número total de threads (`blockDim.x*gridDim.x`) puede no ser suficiente para que cada thread pueda tener asignado una posición de los vectores, tal como sucedía antes. La solución consiste en que cada thread se encargará de acceder a las posiciones:

```
threadIdx.x + blockId.x + blockIdx.x * blockDim.x + blockDim.x * gridDim.x
threadIdx.x + blockId.x + blockIdx.x * blockDim.x + 2*blockDim.x * gridDim.x
threadIdx.x + blockId.x + blockIdx.x * blockDim.x + 3*blockDim.x * gridDim.x
...
```

de los vectores para realizar la operación *saxpy* con esas componentes. Esto significa también que el kernel tendrá un bucle. La parte interesante de este ejercicio es averiguar la forma que tendrá dicho bucle.

Ejercicio 5

El siguiente ejercicio también es sencillo aunque no tanto como parece. La idea aquí consiste en implementar la operación *dot product* o producto escalar. Esta operación viene definida en la biblioteca BLAS y se define de la siguiente manera. Dados dos vectores $x, y \in \mathbb{R}^n$ la operación `dot` tiene la forma

$$a = \sum_{i=0}^{n-1} x_i \cdot y_i.$$

donde a es un escalar, $a \in \mathbb{R}$.

Hasta ahora el tamaño de la salida (cantidad de datos del resultado) ha sido siempre el mismo que el de la entrada. Eso ha facilitado bastante las cosas a la hora de paralelizar un algoritmo. Sin embargo, ahora nos enfrentamos al hecho de que el resultado tiene tamaño 1, es decir, es un escalar, mientras que la entrada es de tamaño n. Esto, que en paralelismo se le conoce con el nombre de *reducción*, supone un problema tanto de implementación como de eficiencia.

En un principio vamos a adoptar una solución "para salir del paso" utilizando las herramientas de las que disponemos. La solución pasará por implementar los dos kernels siguientes que podemos encontrar en el código facilitado (`cu_dot.cu`):

1. `compute_kernel1`: Este kernel, cuya signatura se puede ver en el fichero facilitado, recibe el tamaño de los vectores, los vectores a multiplicar y un vector de 32 elementos. El kernel será llamado de manera fija por una malla formada por un solo bloque de threads. Este bloque será 1D de 32 threads. La forma más fácil de implementar la solución de este kernel es aquella en la que cada thread se va a encargar de realizar la siguiente operación:

   $$a_t = x_t \cdot y_t + x_{32\cdot 1+t} \cdot y_{32\cdot 1+t} + x_{32\cdot 2+t} \cdot y_{32\cdot 2+t} + \ldots + x_{32\cdot i+t} \cdot y_{32\cdot i+t} + \cdots ,$$

   siendo t el identificador del thread, o sea, `threadIdx.x`. Se calcularán términos mientras se cumpla $32i + t \leq n$. Cada thread guardará el resultado calculado, a_t, en una posición del vector `v` pasado como argumento, es decir, `v[threadIdx.x] = a;`.

2. `compute_kernel2`: El kernel anterior ha realizado un cálculo parcial que se encuentra almacenado en las entradas del vector `d_v`, un vector para el que se ha reservado memoria en la GPU. Esto se ha realizado así porque, al menos hasta ahora, no sabemos cómo comunicar datos entre los threads de un bloque. Lo que sí sabemos es que los datos en memoria de la GPU son persistentes entre llamadas a kernels diferentes, es decir, mientras dura la ejecución del programa. El kernel que proponemos aquí se va a encargar de sumar los valores del vector `d_v` y devolver el resultado, que será el resultado de la operación producto escalar.

   Este kernel será llamado por una malla de bloques de threads consistente en un solo bloque, igual que antes, pero ahora el bloque de threads estará formado por un solo thread. El segundo argumento del kernel será un vector de un solo elemento, que habrá sido creado antes de la llamada a dicho kernel como cualquier otro vector. El kernel se limitará a sumar los elementos del vector `v` y asignar el resultado a ese último vector de un solo elemento (`d_result`). De momento esto lo hacemos así dado que los kernels no pueden devolver un valor, siempre devuelven `void`.