ECAR2012, Foundations track
Curso de CUDA, parte 1

Carlos Bederián, Nicolás Wolovick
{bc,nicolasw}@famaf.unc.edu.ar
FaMAF - Universidad Nacional de Córdoba

Objetivos

- Pedir y liberar memoria global de la GPU.
- Mover datos entre el host y el device.
- Configurar los parámetros de ejecución de un kernel.
- Llamar un kernel.
- Escribir un kernel sencillo.

Introducción

Los siguientes ejercicios consisten en código parcialmente implementado, donde el faltante está señalado con comentarios `FALTA` que describen el código a completar.

Para resolver estos ejercicios (y programar en CUDA en general) es necesario utilizar algunas funciones esenciales de la API de CUDA, que manejan la memoria de la placa y la sincronización entre device y host:

- `cudaError_t cudaDeviceSynchronize()` bloquea al host hasta que todo el trabajo asíncrono encolado (kernels en este caso) en la GPU termine.
- `cudaError_t cudaMalloc(void ** ptr, size_t size)` pide `size` bytes[1] en la memoria global de la placa y devuelve el puntero al bloque en el valor apuntado por `ptr`. Notar la diferencia con una llamada habitual a `malloc`:

```
// libc
float * p = (float *) malloc(elems * sizeof(float));
// CUDA
float * q;
cudaMalloc((void **) &q, elems * sizeof(float));
```

- `cudaError_t cudaFree(void * ptr)` libera la memoria previamente pedida con `cudaMalloc`.
- `cudaError_t cudaMemcpy(void * dest, const void * src, size_t size, cudaMemcpyKind kind)` copia `size` bytes desde `src` hacia `dest`. En `kind` se debe declarar la dirección de la copia (`cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`) pero para GPUs de generación Fermi en adelante en sistemas de 64 bits podemos usar `cudaMemcpyDefault` que resuelve la ubicación de los operandos automáticamente. `cudaMemcpy` bloquea de manera similar a `cudaDeviceSynchronize`, esperando a que terminen de correr los kernels antes de realizar copias desde la GPU.
- `cudaError_t cudaMemset(void * dest, int a, size_t size)` rellena el arreglo apuntado por `dest` con `size` copias del byte `a`.

[1] Recordar multiplicar la cantidad de elementos por el tamaño del tipo!

Todas las funciones de la API de CUDA devuelven un `cudaError_t` con el estado de la operación completada. Estos valores de retorno deben ser verificados, ya que los errores en la GPU no generan excepciones, señales ni interrupciones en el código del host. La biblioteca de CUTIL que viene en el SDK de CUDA contiene una función `cutilSafeCall` que realiza esto automáticamente y sale con un error y algo de información cuando falla alguna llamada:

```
#include <cutil_inline.h>
(...)
cutilSafeCall(cudaMalloc((void **) &p, size));
```

Además es necesario realizar las llamadas a kernels propiamente dichas, que son de la forma

```
kernel<<<grid, block>>>(parametros...)
```

donde:

- `kernel` es el nombre del kernel a ejecutar en la GPU. El kernel debe ser una función de tipo `__global__ void`.
- `grid` es una variable de tipo `dim3` que contiene la cantidad de bloques de threads a ejecutarse en cada dimensión[2].
- `block` es una variable de tipo `dim3` que contiene las dimensiones de los bloques de threads a ejecutarse.

Ejercicio 1

El código entregado en el directorio `unscramble` deshace una ofuscación sencilla de una imagen utilizando un kernel CUDA. Al código le resta realizar el manejo de memoria de la GPU y realizar la llamada al kernel.

Note el tamaño del grid. ¿Por qué se usa ese cálculo?

Ejercicio 2

El código del directorio `grid_1d` tiene un kernel que pinta un arreglo de elementos con un color distinto según el bloque, desde el azul al rojo. Además del faltante del ejercicio anterior, deberá configurar el tamaño del grid. Si el grid es demasiado chico, los colores no cubrirán la imagen completa. Si es muy grande[3], la escala no irá del azul al rojo.

Ejercicio 3

El código del directorio `grid_2d` tiene un kernel que pinta una matriz 2D de elementos con un color distinto según el bloque. Deberá configurar el tamaño del grid de manera similar al ejercicio anterior.

Note que aunque el kernel está definido sobre una grilla de dos dimensiones, las coordenadas de un thread aún deben convertirse a un índice en una sola dimensión para acceder a la posición en el arreglo que almacena la matriz.

Ejercicio 4

El directorio `gamma` tiene el código de una aplicación que muestra una imagen especificada y a la par la misma imagen a la que se le ha hecho corrección de gamma por un valor especificado por el usuario. En este caso se provee la función `applygamma` que toma el valor de color de un pixel y el gamma a utilizar, y devuelve el pixel con la corrección de gamma aplicada.

[2] Tenga en cuenta que las dimensiones máximas del grid y los bloques varían según la arquitectura de la GPU.

[3] Esto generalmente no afecta la corrección debido a la guarda que comprueba que el thread esté dentro del rango, pero afecta la performance.

La función `applygamma` es de tipo `__device__`, que se compila para correr en la GPU y puede devolver un valor de cualquier tipo a diferencia de las funciones `__global__`, pero no puede ser llamada desde el host sino que debe ser llamada desde dentro de un kernel, que deberá escribir.