Sesión 2 – Puzzles

Introducción

En esta sesión vamos a trabajar con ejemplos de código muy simples. El objetivo es que os familiaricéis con el particionado de cualquier problema en Bloques y Threads, y aprendáis a repartir los threads de ejecución de un kernel considerando la arquitectura y los datos del problema a resolver.

Existen situaciones en donde puede resultar interesante distribuir los datos del problema de forma análoga a la estructura de los threads de ejecución, determinada al invocar un kernel. Como ejemplo de tales situaciones, se pueden mencionar aquellos kernels que realizan operaciones con matrices de 2 o 3 dimensiones.

También haremos énfasis en problemas cuyo tamaño NO es múltiplo del número de threads y evaluaremos si eso supone una pérdida de rendimiento.

Grids, Bloques y Threads

Durante la ejecución de un kernel podemos utilizar una serie de variables predefinidas para identificar el thread en ejecución:

```
dim3 threadIdx; // número de thread dentro del block
dim3 blockIdx;  // número de block dentro del grid
dim3 blockDim;  // Dimensiones del block
dim3 gridDim;   // Dimensiones del grid
```

El tipo **dim3**, es un tipo predefinido en CUDA y representa un vector de 3 componentes de tipo **unsigned int**. Este tipo puede ser usado en el código de la CPU de la siguiente forma:

```
dim3 BT(8, 8, 4); // define e inicializa BT
tx = BT.x;        // acceso al primer elemento de BT (.x)
ty = BT.y;        // acceso al segundo componente de BT (.y)
tz = BT.z;        // acceso al tercer componente de BT (.z)
```

Las variables **blockDim** y **gridDim** se definen en la invocación del kernel (por defecto, estas variables están inicializadas a 1). Las variables **threadIdx** y **blockIdx** se definen en la GPU al lanzar la ejecución de un bloque con sus correspondientes threads. Por ejemplo, si tenemos la siguiente invocación:

```
dim3 dimGrid(200, 300, 50);       //#bloques = 200*30*50 = 3000000
dim3 dimBlock(8, 16, 4);          //#threads = 8·16·4 = 512
KERNEL<<<dimGrid, dimBlock>>>(…);
```

Las variables predefinidas tienen los siguientes rangos y valores:

- **blockDim.x**: 8, **blockDim.y**: 16, **blockDim.z**: 4
- **gridDim.x**: 200, **gridDim.y**: 300, **gridDim.z**: 50
- **threadIdx.x**: [0..7], **threadIdx.y**: [0..15], **threadIdx.z**: [0..3]
- **blockIdx.x**: [0..199], **blockIdx.y**: [0..299], **blockIdx.z**: [0..49]

El número de bloques que se ejecutarán serán 3.000.000 y en cada uno de estos bloques se lanzarán 512 threads. Eso hace que para resolver este KERNEL, la GPU ejecutara 1.536.000.000 threads en total.

Un buen conocimiento y correcto uso de estas variables es fundamental para programar eficientemente en CUDA.

Puzzle 1D

El primer código con el que vamos a trabajar es el siguiente:

```
void puzzle1DSeq(int N, float *z, float *x, float *y) {
  int i;
  for (i=0; i<N; i++)
    z[i] = 0.5*x[i] + 0.75*y[i] + x[i]*y[i];
}
```

En primer lugar, hay que implementar la versión paralela en CUDA, completando este código:

```
__global__ void puzzle1DPAR(int N, float *z, float *x, float *y)
{
```

```
  . . .
}
```

En la primera versión a implementar podéis suponer que el tamaño del problema es múltiplo del número de threads por bloque.

Editando el fichero puzzle1D.cu que encontraréis en el directorio Puzzle1D tenéis que hacer lo siguiente:

1. Completad el código para su ejecución en la GPU. Ejecutadlo con 1024 threads y tamaño de problema múltiplo de 1024.
2. Cambiad el tamaño para que NO sea múltiplo del número de threads. Modificad el código, dónde sea necesario, para hacer que funcione correctamente.
3. ¿Cómo modificarías la implementación si los datos no caben en la memoria de la GPU?
4. ¿Cómo modificarías la implementación si quisiéramos que el número de Blocks y Threads fuera fijo, independientemente del tamaño del problema?

Puzzle 2D

En este apartado vamos a trabajar con un problema similar, pero ahora con matrices:

```
void puzzle2DSeq(int Nfil, int Mcol,
                 float z[][], float x[][], float y[][]) {
  int i, j;
  for (i=0; i<Nfil; i++)
    for (j=0; j<Mcol; j++)
      z[i][j] = 0.5*x[i][j] + 0.75*y[i][j] + x[i][j]*y[i][j];
}
```

El mismo código, pero trabajando con punteros sería el siguiente:

```
void puzzle2DSeq(int Nfil, int Mcol,
                 float *z, float *x, float *y) {
  int i, j, ind;
  for (i=0; i<Nfil; i++)
    for (j=0; j<Mcol; j++) {
      ind = i * Mcol + j;
```

```
        z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
    }
}
```

De este código os vamos a pedir 3 versiones:

1. En la primera versión cada thread se va a ocupar de 1 columna de la matriz resultado.
2. En la segunda versión cada thread se va a ocupar de 1 fila de la matriz resultado.
3. En la última versión cada thread se va a ocupar de 1 elemento de la matriz resultado.

En este caso vamos a implementar directamente el código para que acepte cualquier tamaño de matriz. En particular tendremos matrices NO cuadradas y NO múltiplo del número de threads. El programa de test con todo el código necesario para probar las 3 versiones está en el fichero puzzle2D.cu que encontraréis en el directorio Puzzle2D.

Puzzle 3D

Finalmente, vamos a trabajar con matrices tridimensionales:

```
void puzzle3DSeq(int Ncar, int Nfil, int Mcol,
                 float z[][][], float x[][][], float y[][][]) {
  int i, j, t;
  for (t=0; t<Ncar; t++)
    for (i=0; i<Nfil; i++)
      for (j=0; j<Mcol; j++)
        z[t][i][j] = 0.5*x[t][i][j] +
                     0.75*y[t][i][j] + x[t][i][j]*y[t][i][j];
}
```

El mismo código, pero trabajando con punteros sería el siguiente:

```
void puzzle3DSeq(int Ncar, int Nfil, int Mcol,
                 float *z, float *x, float *y) {
  int i, j, t, ind;
  for (t=0; t<Ncar; t++)
    for (i=0; i<Nfil; i++)
      for (j=0; j<Ncol; j++) {
        ind = t*Nfil*Ncol + i*Ncol + j;
        z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
      }
}
```

De este código sólo os vamos a pedir la versión en que 1 thread se ocupa de 1 elemento de la matriz resultado. Tenéis que implementar el código para que acepte matrices NO cuadradas. El programa de test con todo el código necesario para comprobar el resultado está en el fichero puzzle3D.cu que encontraréis en el directorio Puzzle3D.

nvprof

nvprof es la herramienta que nos proporciona NVIDIA para hacer profiling de nuestras aplicaciones CUDA. Junto con los ficheros de la sesión os hemos dejado una copia del **man** del comando **nvprof** (fichero **help.nvprof**).

Sería muy interesante que lanzarais alguna de las ejecuciones añadiendo alguno de estos flags al **nvprof**:

- **--print-gpu-summary**, genera un sumario de tiempos de los kernels y transferencias
- **--print-gpu-trace**, genera una traza de los kernels y transferencias
- **--metrics all**, para cada kernel muestra una serie de métricas de rendimiento (las genera TODAS, y hay muchas). Las métricas disponibles dependen de la GPU.
- Podemos especificar las métricas que queremos por separado. Algunas métricas interesantes son las siguientes:
  - **sm_efficiency**
  - **achieved_occupancy**
  - **gld_requested_throughput**
  - **gst_requested_throughput**
  - **dram_utilization**

Se pueden invocar así: **`--metrics sm_efficiency,dram_utilization,…`**

`nvprof` es una herramienta muy útil para los programadores de CUDA. En las siguientes sesiones iremos ampliando las cosas que podemos hacer con **`nvprof`**.

Consideraciones Finales

Los ficheros **`job.sh`** ya incluyen los diferentes tamaños de matriz a probar. Cuando queráis que los tamaños de matriz no sean múltiplo del número de threads, hacedlo en el código. Por ejemplo, con una sentencia como ésta:

```
if (N % nThreads == 0) N--;
```

Para facilitaros las cosas, y sólo para evitar que os quedéis encallados, hemos incluido un directorio en el que están todos los kernels solucionados.