Sesión 1 – Primeros pasos con CUDA

Antes de empezar

Todo lo necesario para la sesión 01 lo encontraréis en el fichero **Sesion01.tar** que hay que desempaquetar con el siguiente comando:

```
tar xvf Sesion01.tar
```

ScanDevices

Este test sólo se ha de compilar y ejecutar. Para ejecutar un programa en CUDA en boada hay que utilizar el sistema de colas. En nuestro caso para compilar y ejecutar:

```
make
sbatch job.sh
```

En todas las sesiones incluiremos el **Makefile** correspondiente, así como el script para lanzar la ejecución a la cola **cuda** (esta cola está especificada en el fichero **job.sh**).

Al ejecutar este test sabremos qué GPUs tenemos instalada en el servidor y para cada una de ellas (pueden ser diferentes), información detallada de sus características. La información de una GPU es un listado parecido a éste:

```
Device 0: ---
  Major revision number:                         ---
  Minor revision number:                         ---
  Total amount of global memory:                 --- bytes
  Number of multiprocessors:                     ---
  Number of cores:                               ---
  Total amount of constant memory:               --- bytes
  Total amount of shared memory per block:       --- bytes
  Total number of registers available per block: ---
  Warp size:                                     ---
```

```
Maximum number of threads per block:           ---
Maximum sizes of each dimension of a block:    --- x --- x ---
Maximum sizes of each dimension of a grid:     --- x --- x ---
Maximum memory pitch:                          --- bytes
Texture alignment:                             --- bytes
Clock rate:                                    --- GHz
Concurrent copy and execution:                 ---
...
```

Algunos de estos valores (básicamente los marcados en **rojo**) los deberemos tener en mente al programar con CUDA en este servidor. Por ejemplo, si lanzamos un kernel con más threads de los que soporta la GPU, el kernel no funcionará.

Ahora, si miráis el `job.sh`, hay un parámetro interesante:

```
#SBATCH --gres=gpu:1
```

Este comando le dice al gestor cuantas GPUs queréis utilizar. Podéis cambiarlo por este valor:

```
#SBATCH --gres=gpu:4
```

Y volver a ejecutar el programa de otra vez. ¿Qué ha ocurrido?

Es un buen momento para consultar el manual online de CUDA. La información que ofrece este test se obtiene llamando a la función `cudaGetDeviceProperties()`. Si consultamos la siguiente página: `docs.nvidia.com/cuda/cuda-runtime-api/` y buscamos esta rutina, veremos la gran cantidad de información que podemos obtener con ella.

Cuando instalamos una GPU + CUDA este es el primer test que hay que correr. Primero para comprobar que la GPU funciona, y luego para obtener estos parámetros.

SaxpyP

En este test se calcula la operación **Saxpy** que vimos como ejemplo en las clases de teoría. Para compilar y ejecutar usaremos los ficheros que hay en el directorio. En nuestro caso:

```
make
sbatch job.sh
```

Si miráis el fichero (**main.cu**), veréis que tiene una estructura similar a la que hemos visto en clase de teoría con algunos añadidos. Lo más novedoso es la forma en que tomamos los tiempos de ejecución. Teniendo en cuenta que, la ejecución de determinadas operaciones en la GPU es asíncrona con respecto a la CPU, no es posible usar las mismas rutinas que usaríamos en un programa convencional, para medir el tiempo de ejecución.

Hemos de usar los eventos de CUDA (**cudaEvent_t**). Lo que hay que hacer es registrar los eventos necesarios entre el código a medir:

```
cudaEventRecord(E0, 0); cudaEventSynchronize(E0);
// Código a medir
cudaEventRecord(E1, 0); cudaEventSynchronize(E1);
cudaEventElapsedTime(&tiempo, E0, E1);
```

La rutina **cudaEventSynchronize** provocará que el programa espere hasta que el evento quede registrado en la GPU. Esto es imprescindible cuando trabajamos con operaciones asíncronas. La rutina **cudaEventElapsedTime** calcula el tiempo transcurrido (en ms) entre el registro de los 2 eventos.

Con este test se pueden realizar muchas pruebas. Os enumeramos algunas:

1. Compilad, ejecutad y comprobad que funciona correctamente.
2. Calculad los MFLOPS contando sólo el kernel.
3. Calculad los MFLOPS contando también las transferencias de datos.
4. Calculad el ancho de banda de las transferencias CPU → GPU, y GPU → CPU.
5. Calculad los mismos anchos de banda, pero ahora utilizando memoria “pinned”. El código necesario ya está escrito en un comentario.
6. La rutina que comprueba que el resultado es correcto, hace este test de error: **(abs(a-b)/a > E).** Cambiad el test de error por uno de igualdad **(a != b).** ¿Qué ocurre en este caso?

Profiling

Como en muchos otros entornos, tenemos herramientas para analizar el rendimiento de una aplicación CUDA. La herramienta más simple es **nvprof**. La forma de invocarla es muy simple:

```
nvprof ./MiPrograma parametros
```

El resultado del profiling se escribirá en el fichero de error que nos devuelve la cola cuda. La información que ofrece **nvprof** tiene el siguiente aspecto:

```
==70298== NVPROF is profiling process 70298, command: ./SaxpyP.exe
==70298== Profiling application: ./SaxpyP.exe
==70298== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 71.76%  94.713ms         2  47.357ms  47.325ms  47.389ms  [CUDA memcpy HtoD]
 27.36%  36.105ms         1  36.105ms  36.105ms  36.105ms  [CUDA memcpy DtoH]
  0.88%  1.1624ms         1  1.1624ms  1.1624ms  1.1624ms  saxpyP()
```

Para más detalles se puede consultar **docs.nvidia.com/cuda/profiler-users-guide**.

Control de Errores en CUDA

En este último test hemos incluido el código necesario para comprobar los errores en CUDA. En CUDA, todas las llamadas devuelven un código de error (a excepción de la ejecución de un kernel). El código de error es un tipo predefinido de CUDA: **cudaError_t**. También existe una función de CUDA que devuelve el código del último error:

```
cudaError_t cudaGetLastError(void)
```

Para saber de que error se trata, disponemos de una función CUDA que devuelve un string con la descripción del error:

```
char* cudaGetErrorString(cudaError_t)
```

La rutina de error que podéis encontrar en el código es la siguiente:

```
void CheckCudaError(char sms[]) {
  cudaError_t error;

  error = cudaGetLastError();
```

```
  if (error) printf("%s:%s\n", sms, cudaGetErrorString(error));
}
```

Esta rutina se puede usar después de cada invocación a una rutina CUDA, como por ejemplo:

```
cudaMemcpy(H_y, d_y, numBytes, cudaMemcpyDeviceToHost);
CheckCudaError((char *) "Copiar Datos Device --> Host");
```

En caso de que la rutina **cudaMemcpy** produzca un error, nos aparecerá el mensaje:

```
Copiar Datos Device --> Host: descripción del error
```

Para ver cómo funciona el control de errores podéis probar las siguientes cosas:

1. Modificad el valor de N hasta encontrar el tamaño de vector que provoca que el código deje de funcionar. ¿Porqué no funciona?
2. Modificad el número de threads, siempre con valores múltiplo de 32, hasta que el código deje de funcionar. No os olvidéis de probar con valores menores que 256. ¿Porqué no funciona?
3. El código que tenemos funciona correctamente con N múltiplo del número de threads. Modificad el código para que funcione con cualquier valor de N, pero sin modificar el kernel (p. e. Aumentando el tamaño de los vectores hasta que sea múltiplo).

Recursos Adicionales

Junto con la distribución gratuita de CUDA que se puede obtener en la página web de NVIDIA podemos encontrar algunos recursos muy útiles. Estos recursos adicionales los podéis encontrar en: **/Soft/cuda/VER**[1]. En particular nos interesa lo siguiente:

- **Ejemplos**. En el directorio **/Soft/cuda/VER/samples** encontraréis numerosos ejemplos de cuda. Desde aplicaciones muy simples, hasta ejemplos avanzados. Estos ejemplos se pueden compilar fácilmente modificando los **Makefiles** de la sesión actual.
- **Documentación**. En el directorio **/Soft/cuda/VER/doc/pdfs** se podían encontrar todos los manuales de cuda de la versión VER: manuales de consulta, de buenas prácticas, de optimización, … Este directorio existía hasta la versión 9. Para encontrar información más actualizada hay que visitar: https://docs.nvidia.com/cuda (ahí está la información de todas las versiones del compilador).

[1] **VER** depende de la versión de cuda instalada, nosotros estamos usando la 11.2.1.

- **Herramientas**. En el directorio **`/Soft/cuda/VER/tools`** encontraréis el fichero Excel **`CUDA_Occupancy_Calculator.xls`**. La utilidad de esta herramienta la tendréis que investigar vosotros.