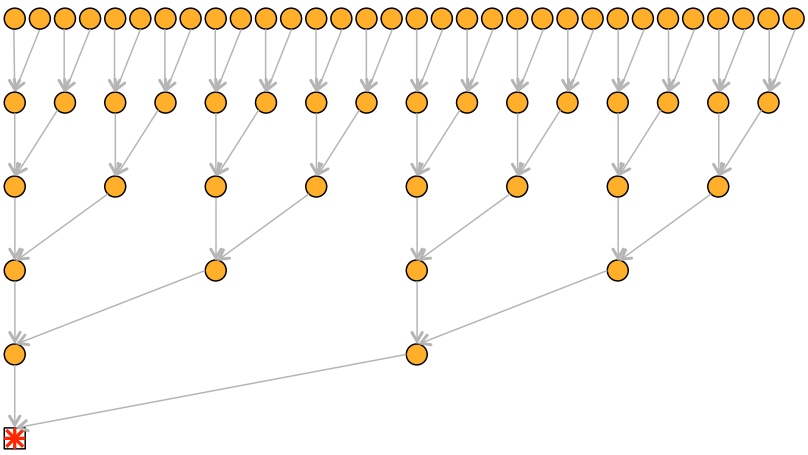
Sesión 3 – Reducciones

Introducción

En esta sesión vamos a trabajar con un algoritmo muy común, fácil de implementar en CUDA, pero difícil de optimizar. Es lo que conocemos como una reducción. Un ejemplo secuencial de reducción es la suma de los elementos de un vector:

```
sum = 0.0;
for (i=0; i<N; i++)
  sum = sum + v[i];
```

El esquema típico para realizar una reducción en paralelo es construir un árbol similar a éste:



Empezamos con 2^n *Threads*, que realizan 2^n sumas parciales en paralelo. En el siguiente paso 2^{n-1} *Threads* dejan de trabajar y el resto realiza 2^{n-1} sumas parciales. Seguimos así, hasta que al final sólo queda un thread que hace la suma final. Este tipo de algoritmo necesita que el número de *Threads* a utilizar sea potencia de 2.

Todas las ideas de esta sesión las hemos obtenido de un tutorial de NVIDIA que podéis encontrar en el siguiente enlace:

http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Por simplicidad, supondremos que el tamaño de vector que utilizaremos en todas las versiones es múltiplo del número de *Threads* (nTHreads = 512 y N = 16.777.216 =2^{24}).

Kernel 01 (main01.cu)

Este test os lo daremos ya hecho y sólo tenéis compilarlo y ejecutarlo.

Utilizamos un kernel que calcula la suma de los elementos de un vector de 512 elementos. El código es el siguiente:

```
__global__ void Kernel01(double *g_idata, double *g_odata) {
  __shared__ double sdata[512];
  unsigned int s;

  // Cada thread carga 1 elemento desde la memoria global
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();

  // Hacemos la reduccion en la memoria compartida
  for(s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0)
      sdata[tid] += sdata[tid + s];
    __syncthreads();
  }

  // El thread 0 escribe el resultado de este bloque en la memoria global
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];

}
```

Cada uno de los *Thread Blocks* da como resultado una suma parcial que se guarda en `g_odata`.

Para compilar el programa sólo hay que hacer:

```
make kernel01.exe
```

Para ejecutarlo hay que enviarlo al sistema de colas, tal y cómo hicimos en la sesión anterior.

Observad, que el compilador nos da información útil de cómo estamos utilizando los recursos hardware: cuántos registros utilizamos por *thread*, cuánta memoria compartida utilizamos por *Thread Block*, …

En este código sólo invocamos el kernel 1 vez, de tal forma que al acabar tenemos un vector de sumas parciales con tantos elementos como `nBlocks`. Para calcular el resultado final hay que hacer la suma final de estos elementos.

Este código se muestra a continuación[1]:

```
// Ejecutar el kernel
 Kernel01<<<nBlocks, nThreads>>>(d_v, d_w);
 // Obtener el resultado parcial desde el host
 cudaMemcpy(h_w, d_w, numBytesW, cudaMemcpyDeviceToHost);
 SUM = 0.0;
 for (i=0; i<nBlocks; i++)
   SUM = SUM + h_w[i];
```

Para obtener los tiempos de ejecución de código que se ejecutan en la CPU y la GPU es necesario utilizar *Events* de CUDA.

Para comparar rendimientos de forma más visual, calcularemos el "ancho de banda" de sumar los elementos del vector (tamaño del vector a sumar en bytes/tiempo de ejecución).

La ejecución de este código dará algo parecido a esto:

```
KERNEL 01
Vector Size: 16777216
nThreads: 512
nBlocks: 32768
Tiempo Total 7.464704 milseg
Ancho de Banda 8.990 GB/s
TEST PASS
```

Estos tiempos corresponden a la ejecución en una GPU diferente a la utilizada en el laboratorio de TGA.

Al final del programa se comprueba que el resultado es correcto. En caso contrario aparecería un mensaje como éste:

```
...
TEST FAIL
```

[1] Este es el código que vamos a evaluar. Para comparar las diferentes versiones calcularemos el tiempo de ejecución de esta porción de código.

Kernel 02 (main02.cu)[2]

En el Kernel01 lanzamos 32.768 *Thread Blocks*, eso nos obliga a sumar un vector de 32.768 elementos en la CPU. Una optimización que podría hacerse es lanzar una nueva invocación del mismo kernel con (32.768/512)=64 *Thread Blocks* y 512 *Threads*. Así sólo necesitaremos sumar un vector de 64 elementos en la CPU. Además, nos aprovecharemos de que el vector de 32.768 elementos ya está almacenado en la GPU.

Esta optimización tenéis que hacerla vosotros.

Kernel 03 (main03.cu)

En el kernel que hemos implementado hay dos problemas. El más importante es que la ejecución de los *warps* es muy ineficiente, en muy pocas iteraciones estamos utilizando muy pocos *threads* por *warp*, y además de forma dispersa. El segundo problema es que utilizamos una operación que es muy costosa: %, el resto de la división entera. Teniendo en cuenta que es una división por un valor potencia de 2, podríamos utilizar máscaras de bits y operaciones lógicas. Para resolver los 2 problemas y utilizando como punto de partida el código del `main02.cu`, simplemente debéis sustituir esta porción de código:

```
// Hacemos la reduccion en la memoria compartida
for(s=1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
```

Por esta otra:

```
// Hacemos la reduccion en la memoria compartida
for(s=1; s < blockDim.x; s *= 2) {
  int index = 2 * s * tid;
  if (index < blockDim.x)
    sdata[index] += sdata[index + s];
  __syncthreads();
}
```

[2] Para los diferentes kernels se recomienda generar un nuevo fichero. En este caso con el nombre "`main02.cu`". De esa forma podréis usar el makefile para compilarlos con "`make kernel02.exe`".

Este código mejora el uso de los *threads*, siguen siendo pocos, pero son consecutivos, y además evita el uso de operaciones costosas (%).

Kernel 04 (main04.cu)

Si analizamos el patrón de acceso a los datos, veremos que el patrón es muy desafortunado, utilizando *strides* potencia de dos (2, 4, 8, …) lo que provoca numerosos conflictos al acceder a los bancos de memoria. Necesitamos modificar el patrón de acceso para que los datos sean accedidos en posiciones consecutivas de memoria.

Además, el kernel que hemos implementado utiliza los *Threads* de forma muy ineficiente. Primero trabajan todos los *Threads*, luego la mitad, luego la cuarta parte, … Teniendo en cuenta que CUDA ejecuta los *Threads* en grupos de 32 (del 0 al 31, del 32 al 63, …) provoca que el rendimiento sea muy deficiente. A partir de la tercera iteración en cada *warp* sólo se ejecutan unos pocos *Threads*.

Para solventar los dos problemas, simplemente debéis sustituir esta porción de código:

```
// Hacemos la reduccion en la memoria compartida
for(s=1; s < blockDim.x; s *= 2) {
  int index = 2 * s * tid;
  if (index < blockDim.x)
    sdata[index] += sdata[index + s];
  __syncthreads();
}
```

Por esta otra:

```
// Hacemos la reduccion en la memoria compartida
for (s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
```

Podéis comprobar fácilmente que estamos accediendo a posiciones consecutivas de memoria y que el uso de los *Threads* es más eficiente.

Kernel 05 (main05.cu)

Si estudiamos detenidamente el último kernel podemos observar que todos los *Threads* traen 1 dato desde la memoria global a la compartida, pero que, en la primera iteración del bucle, la mitad de los *Threads* ya no tienen nada que hacer. Esto supone una pérdida de recursos muy importante que podemos resolver fácilmente.

La solución es sencilla:

- Hay que dividir a la mitad el número de *Thread Blocks* que ejecutamos,
- sustituir la parte inicial del kernel, de tal forma que cada *Thread* lea dos datos de la memoria global, los sume, y guarde el resultado en la memoria compartida.

Resumiendo, debéis sustituir esta porción de código:

```
// Cada thread carga 1 elemento desde la memoria global
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
```

Por esta:

```
// Cada thread carga 2 elementos desde la memoria global,
// los suma y los deja en la memoria compartida
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

No os olvidéis de dividir a la mitad el número de Thread Blocks que ejecutamos.

La mejora de rendimiento que se obtiene es importante.

Kernel 06 (main06.cu)

Si analizamos los resultados, veremos que el ancho de banda que obtenemos áun está lejos del ancho de banda de la memoria global de la GPU. Hemos de hacer optimizaciones más agresivas.

El número de *Threads* activos se va reduciendo en cada iteración del bucle. Llegará un punto en que sólo tendremos 32 *Threads* activos (1 *Warp*). Dentro de 1 *Warp* todas las instrucciones se ejecutan de forma síncrona [se ejecutan los 32 *Threads* simultáneamente]. Eso implica que la instrucción

`__syncthreads()` no es necesaria cuando se ejecuta el último *Warp*, y que podemos desenrollar la parte final del bucle.

Resumiendo, debéis sustituir esta porción de código:

```
// Hacemos la reduccion en la memoria compartida
for (s=blockDim.x/2; s>0; s>>=1) {
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
```

Por esta otra:

```
// Hacemos la reduccion en la memoria compartida
for (s=blockDim.x/2; s>32; s>>=1) {
  if (tid < s)
    sdata[tid] += sdata[tid + s];
  __syncthreads();
}
// desenrollamos el ultimo warp activo
if (tid < 32) {
  volatile double *smem = sdata;
  smem[tid] += smem[tid + 32];
  smem[tid] += smem[tid + 16];
  smem[tid] += smem[tid + 8];
  smem[tid] += smem[tid + 4];
  smem[tid] += smem[tid + 2];
  smem[tid] += smem[tid + 1];
}
```

De este código que ejecutamos en el último *Warp* activo hay que decir algunas cosas:

- La parte final no necesita sincronización explícita. Sólo se está ejecutando 1 *Warp* y todos los *Threads* del *Warp* están ejecutando la misma instrucción simultáneamente.
- Los 32 *Threads* ejecutan todo el código. Algunos de ellos hacen operaciones inútiles, e incluso, sin sentido. Pero al acabar, en el *Thread* 0 tenemos el resultado que estamos buscando.
- La sentencia `volatile double *sem = sdata` es un recurso del lenguaje que informa al compilador que esa variable no se puede guardar en cache. Podéis comentar la sentencia y veréis que el programa deja de funcionar.

- Si utilizamos C++, podemos escribir un código con templates, completamente desenrollado, que funcione para cualquier tamaño de *Thread Block*. Eso nos obligaría a escribir el código de todos los posibles casos. En la página web de NVIDIA se puede encontrar este código.

Kernel 07 (main07.cu) [ÚLTIMO TEST]

La última optimización mezcla parte secuencial [en los *Thread Blocks*] con el código que ya tenemos. La idea es sencilla:

- Reducir el número de *Thread Blocks*, por ejemplo a 2048, con 512 *Threads* en cada uno.
- El primer paso del algoritmo es sumar, secuencialmente, en cada *thread* los N/(2048·512) datos que le corresponden y dejar la suma parcial en la memoria compartida.
- A continuación, cada *Thread Block* calcula la suma parcial, utilizando el mismo kernel de la versión anterior.
- En el host, es suficiente con hacer una invocación del kernel, y dejar a la CPU la suma final de las 2048 sumas parciales.

Resumiendo, debéis sustituir esta porción de código:

```
// Cada thread carga 2 elementos desde la memoria global,
// los suma y los deja en la memoria compartida
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

Por esta otra:

```
 // Cada thread realiza la suma parcial de los datos que le
// corresponden y la deja en la memoria compartida
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
unsigned int gridSize = blockDim.x*2*gridDim.x;
sdata[tid] = 0;
while (i < N) {
  sdata[tid] += g_idata[i] + g_idata[i+blockDim.x];
  i += gridSize;
}
__syncthreads();
```

Para que todo funcione, hay que incluir la N como parámetro del kernel y modificar el código del host adecuadamente.

Profiling

En las antiguas versiones de CUDA utilizábamos para hacer profiling la herramienta `nvprof`. En la actualidad, `nvprof` sigue existiendo, pero la información que ofrece es muy limitada.

La nueva herramienta de profiling que os vamos a mostrar es `ncu`. Hemos montado un fichero llamado `jobInfoNCU.sh`, que podéis lanzar a ejecución con un `sbatch` para que os muestre las diversas opciones y el help de `ncu`.

Por ahora, usaremos estas tres posibilidades:

- `ncu ./kernel01.exe`
- `ncu --set full ./kernel01.exe`
- `ncu --set detailed ./kernel01.exe`

Lo que muestra cada una de estas llamadas es la información de profiling de un conjunto de secciones predefinidas. Si queréis saber que secciones pertenecen a cada conjunto podéis hacer:

```
ncu --list-sets
```

Esta herramienta muestra mucha información, os recomendamos que la leáis detenidamente.

Consideraciones Finales

En esta sesión os damos como punto de entrada el fichero main01.cu. A partir de aquí hay que ir construyendo los diferentes kernels con las ideas que os hemos dado.

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