Sesión 6 – Streams

Kernel 00 (stream00.cu)

Para trabajar con streams vamos a utilizar un código muy simple que trabaja con vectores. El código de partida os lo damos ya hecho, sólo tenéis que compilarlo y ejecutarlo.

Para compilar el programa sólo hay que hacer:

```
make kernel00.exe
```

Para ejecutarlo hay que enviarlo al sistema de colas, tal y cómo hicimos en sesiones anteriores.

```
sbatch job.sh
```

Al ejecutar el kernel podemos pasarle 2 parámetros. El primero es el tamaño del problema (en K elementos). El segundo es un flag ('Y' o 'N') para indicar si queremos comprobar el resultado. Una ejecución podría ser la siguiente:

```
./kernel00.exe 10000 Y
```

Ejecuta el kernel00 con vectores de 10000·1024 elementos y comprueba el resultado. El resultado de esta ejecución podría ser el siguiente:

```
KERNEL 00: NO – Streams
Dimension Problema: 10240000
Invocacion Kernel <<<nBlocks,nKernels>>>(N): <<<10000,1024>>>(N)
nKernels: 1
Usando Pinned Memory
Tiempo Global (00): 8.245632 milseg
Rendimiento Global (00): 500.47 GFLOPS
Tiempo Kernel (00): 2.117824 milseg
Tiempo HtoD (00): 4.566848 milseg
Tiempo DtoH (00): 1.560960 milseg
TEST PASS
```

El programa a evaluar es muy simple:

- Se crean 2 vectores en la CPU y se transfieren a la GPU.
- Se realiza un cálculo con esos vectores para generar un vector resultado.
- Se transfiere de la GPU a la CPU el vector resultado.

El código del kernel se muestra a continuación:

```
__global__ void Kernel0(int N,float a, float b, float *A, float *B, float *C){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j;
  if (i<N) {
      C[i] = a*A[i] + b*B[i];
      for (j=0; j<DUMMY; j++)
        C[i] += a*A[i] + b*B[i];
  }
}
```

Este código no realiza ninguna tarea útil. Simplemente, variando la constante `DUMMY`, podemos equilibrar el coste del cálculo (en tiempo) con el coste de las comunicaciones. La variable `DUMMY` se puede modificar en el `Makefile` antes de compilar.

En concreto para `DUMMY == 100` en los resultados de la ejecución anterior, tenemos:

```
…
Tiempo Kernel (00): 2.117824 milseg
Tiempo HtoD (00): 4.566848 milseg
Tiempo DtoH (00): 1.560960 milseg
…
```

Puede verse que el coste de las transferencias entre CPU y GPU es “similar” al coste del kernel.

Podéis lanzar varias ejecuciones variando el valor de `DUMMY` en el *Makefile* para ver el efecto en el resultado final. Os recomendamos que lanzéis los kernels varias veces. Los tiempos obtenidos no siempre son coherentes.

Hablemos de Streams

En esta sesión vamos a trabajar con streams. En las tarjetas gráficas de NVIDIA existen 3 colas de ejecución: 1 cola para cálculo y 2 colas para mover datos entre host y GPU, una en cada sentido. Las 3 colas pueden funcionar de forma concurrente, siempre y cuando pertenezcan a streams diferentes.

La definición e inicialización de un stream se muestra a continuación:

```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
```

Para soportar la ejecución concurrente es necesario que los datos del host estén almacenados en *Pinned Memory*:

```
cudaMallocHost(&host1, size);
```

y que las transferencias de información sean asíncronas:

```
cudaMemcpyAsync(d1, h1, size, cudaMemcpyHostToDevice, stream1);
```

(se puede ver la forma de indicar el **stream** utilizado).

La forma en que se indica en qué stream se ejecuta un kernel es la siguiente:

```
kernel2<<<grid, block, 0, stream1>>>();
```

Finalmente, al acabar la aplicación es necesario destruir el **stream**:

```
cudaStreamDestroy(stream1);
```

Kernel 01 (stream01.cu)

Modificad la aplicación anterior para que trabaje con streams.

Si miramos el detalle del código se pueden identificar 4 tareas:

- Copiar A: CPU → GPU
- Copiar B: CPU → GPU

- Kernel cálculo C ← Kernel(a, b, A, B)

- Copiar C: GPU → CPU

Todas estas tareas trabajan con vectores de N elementos.

Para trabajar con streams hay que tener en cuenta varias cosas:

- Vamos a utilizar 4 streams.
- Hay que aplicar el algoritmo en 4 partes (con N/4 elementos por tarea) y asociar cada parte a un stream diferente.
- En el código resultante habrá:
  - 4 transferencias de partes del vector A, en streams diferentes
  - 4 transferencias de partes del vector B, en streams diferentes
  - 4 ejecuciones del kernel, en streams diferentes
  - 4 transferencias de partes del vector C, en streams diferentes
  - Para trabajar con diferentes partes de los vectores, sólo hay que jugar con los punteros.
- El kernel no se ha de modificar.
- La forma de emitir las diferentes llamadas influye en el rendimiento, os aconsejamos que probéis diversas soluciones.

Kernel 02 (stream02.cu)

La forma que os hemos propuesto para utilizar streams es demasiado simplista. Hemos partido una XXX (podéis llamar esta operación como más os guste) de vectores con N elementos en 4 XXX de vectores de N/4 elementos.

Una solución más útil, pensando en problemas futuros, sería partir el problema en tareas con un número fijo de elementos (por ejemplo, si N=50000·1024, podríamos usar Nk=2500·1024) y enviar a ejecutar las operaciones que sean necesarias con tamaño Nk.

Modificad el código anterior, teniendo en cuenta que enviaremos kernels con Nk=2500·1024 (por ejemplo). Ahora podéis utilizar el número de streams que queráis, por ejemplo 8 streams. Os ayudará utilizar un vector de streams como éste:

```
cudaStream_t stream[8];
```

En definitiva, la idea es emitir las diferentes llamadas al kernel, con las transferencias necesarias, utilizando un bucle. Obviamente, cada grupo de transferencias-kernel hay que emitirlo a un stream diferente:

```
for (---, str=0; ---; ---, str = (str+1)%nStreams) {
  . . .
  Kernel00<<< ---, stream[str]>>>( --- );
  . . .
}
```

Algunos comentarios respecto a la toma de tiempos

Para la toma de tiempos utilizamos el mismo método que en sesiones anteriores:

```
cudaEventRecord(E0, 0);
//
// CÓDIGO A EVALUAR
//
cudaEventRecord(E3, 0); cudaEventSynchronize(E3);
cudaEventElapsedTime(&TiempoTotal,  E0, E3);
```

Al acabar, en la variable `TiempoTotal`, tenemos el tiempo de ejecución de nuestro código.

Hay que puntualizar que el segundo parámetro de la función `cudaEventRecord` indica el stream en el que se registra el evento. Pero, según el manual de CUDA: " … *events in stream zero are completed after all preceding tasks and commands in all streams are completed* …" (¡Nos va perfecto!).

Por último, los tiempos de ejecución no siempre son coherentes. Para las pruebas os aconsejamos que sigáis el siguiente protocolo. Para probar que los algoritmos funcionan, utilizad tamaños pequeños:

```
./kernel01.exe 10000 Y
./kernel02.exe 10000 Y
```

Para tomar tiempos, utilizad tamaños más grandes y desactivad la comprobación del resultado:

```
./kernel01.exe 300000 N
./kernel02.exe 300000 N
```

Consideraciones Finales

Igual que hicimos en sesiones anteriores, para facilitaros las cosas, y sólo para evitar que os quedéis encallados, hemos incluido un directorio en el que están todos los códigos solucionados. En algunas de las soluciones se incluyen diferentes formas de lanzar los streams.

Con esto se acaban las sesiones dirigidas. A partir de ahora dedicaremos las sesiones de laboratorio a vuestro proyecto. Algunas consideraciones respecto al proyecto:

- Se realiza en grupos de 2 estudiantes.
- El proyecto os ha de costar unas 30-40 horas de trabajo, teniendo en cuenta que nos quedan 5 sesiones de laboratorio.
- En la entrega del proyecto, tenéis que hacer una pequeña memoria (10-12 páginas) y entregar el código del mismo.
- Contactaremos con cada grupo para hacer un seguimiento del proyecto.