Sesión 5 – multi GPU

Introducción

En esta sesión vamos a trabajar con varias GPUs. Vamos a utilizar en una misma aplicación las 4 GPUs que tenemos en nuestro servidor.

Para trabajar con varias GPUs hay que descomponer el problema en partes (similar a lo que hay que hacer para trabajar con streams) y distribuirlo entre las diversas GPUs. En general, será necesario disponer de algún tipo de mecanismo para que las diversas GPUs se comuniquen. Tanto para sincronizarse, como para compartir datos.

Las razones para trabajar con varias GPUs son siempre las mismas:

- Aumentar la velocidad de cálculo
- El tamaño del problema supera la capacidad de memoria de 1 GPU

Existen diversos escenarios posibles:

- Un único nodo con varias GPUs
- Varios nodos conectados en red con 1 o varias GPUs por nodo

Por motivos obvios, nos centraremos en el primer caso.

Por definición, todas las llamadas CUDA se ejecutan en la *current GPU*. Para trabajar con varias GPUs sólo necesitamos una rutina que nos permita cambiar la *current GPU*. Esa rutina es:

```
cudaError_t cudaSetDevice(int device)
```

El parámetro `device` es simplemente un número entero entre 0 y N-1, siendo N el número de GPUs disponibles. Para saber el número de GPUs disponibles hay que invocar esta función:

```
cudaError_t cudaGetDeviceCount(int *count)
```

Al acabar la rutina, `*count` tendrá el número de GPUs disponibles en el sistema. Esta rutina ya la utilizamos en la Sesion01 (podéis mirar el código), en la aplicación ScanDevices.

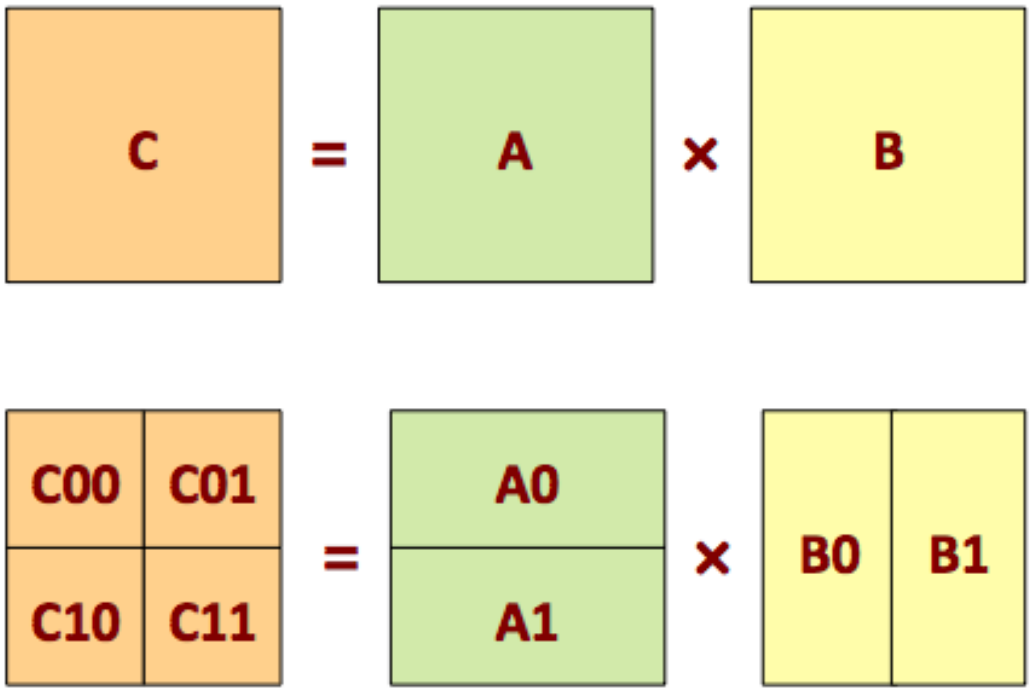
El siguiente código permite ejecutar varios kernels en las GPUs disponibles:

```
cudaGetDeviceCount(&count);
for (dev = 0; dev < count; dev++) {
  . . .
  cudaSetDevice(dev);
  kernel<<< . . . >>>(. . .);
  . .
}
```

KERNEL MultiGPU

Como ejemplo vamos a utilizar el producto de matrices que ya vimos en la sesión anterior y para simplificar el código vamos a realizar un algoritmo en el que no tengamos que mover datos entre GPUs.

La idea básica del algoritmo que vamos a implementar se muestra en la siguiente figura:



Para simplificar las cosas supondremos que las matrices son cuadradas, y mejor áun, que las dimensiones son potencia de 2 (p.e.: 2048 × 2048). La distribución de cálculos es la siguiente:

- GPU 0: calculamos C00 = A0 · B0, necesita A0 y B0
- GPU1: calculamos C01 = A0 · B1, necesita A0 y B1

- GPU2: calculamos C10 = A1 · B0, necesita A1 y B0
- GPU3: calculamos C11 = A1 · B1, necesita A1 y B1

Para hacer el cálculo de producto de matrices, podéis utilizar el kernel a bloques de la sesión anterior. En concreto el que funcionaba para tamaños de problema múltiplo del número de threads.

En los ficheros de la sesión encontraréis un fichero (MM4GPUs.cu). En este fichero, está parte del algoritmo implementado. En concreto encontraréis el cálculo de C00, así como la infraestructura necesaria para crear las matrices, comprobar el resultado y tomar tiempos.

En una aplicación real, recibiríamos las matrices completas (A, B y C), y por programa, deberíamos construir las submatrices que necesitamos. Teniendo en cuenta que el objetivo de la práctica es usar varias GPUs, y para simplificar el trabajo, hemos creado las matrices A0, A1, B0, B1, C00, C01, C10 y C11 por separado.

Para compilar y ejecutar la aplicación utilizaremos las mismas herramientas de otras sesiones:

```
make kernel4GPUs.exe
sbatch job.sh
```

La primera tarea de la Sesión consiste en completar el programa para que calcule el producto de la matriz C completa. Al ejecutar la aplicación, en el caso de que queráis comprobar el resultado, es conveniente que no utilicéis tamaños de matriz muy grandes (p.e. 1024).

Tomando Tiempos

Cuando queremos averiguar el tiempo de ejecución de una aplicación CUDA que se ejecuta en una sola GPU, utilizábamos eventos de la siguiente forma:

```
float TiempoTotal;
cudaEvent_t E0, E3;

cudaEventCreate(&E0); cudaEventCreate(&E3);

cudaEventRecord(E0, 0);
//AQUÍ ESTARÍA LA PORCIÓN DE CÓDIGO A EVALUAR
cudaEventRecord(E3, 0); cudaEventSynchronize(E3);

cudaEventElapsedTime(&TiempoTotal, E0, E3);
// Al acabar el TiempoTotal tenemos el tiempo de ejecución del código en ms

cudaEventDestroy(E0); cudaEventDestroy(E3);
```

Lo que hacemos es registrar eventos (E0 y E3) en la cola de ejecución de la GPU y sincronizarnos con su finalización. Junto con el evento el sistema guarda el instante de tiempo en el que acaba. Los eventos se ejecutan cuando todas las llamadas previas a CUDA, en esa GPU y en ese *stream*, han terminado. Para acabar sólo necesitamos la diferencia de tiempos entre los eventos E0 y E3.

Cuando estamos trabajando con varias GPUs hay que hacer alguna cosa más. Para empezar, sólo podemos registrar eventos en nuestra propia GPU y no podemos acceder a la información de un evento en otra GPU. Pero, lo qué si podemos hacer, es sincronizarnos con un evento de otra GPU.

La idea es la siguiente. En la GPU 0, calculamos el tiempo de ejecución global, guardando un evento al principio del cálculo (E0) y otro al final (E3). Pero para que la toma de tiempos sea correcta nos hemos de asegurar que la ejecución en las otras GPUs haya terminado. Lo que hacemos es registrar un evento al final del cálculo en cada GPU: X1, X2 y X3. Y en la GPU 0 nos sincronizaremos con los 3 eventos.

El código necesario para tomar tiempos ya está en la aplicación que os hemos pasado.

En otras sesiones, también calculábamos el tiempo de ejecución del kernel. En este caso no es viable, porque la sincronización necesaria influye en el tiempo global de la aplicación. Si queremos saber el tiempo de los kernels hay que consultar los datos que nos da `nvprof`.

También es posible calcular el tiempo que tarda la aplicación utilizando `nvprof`. En concreto, con la siguiente llamada:

```
nsys nvprof --print-gpu-trace ./kernel4GPUs.exe 2048 N
```

En la información que proporciona `nvprof`, podemos encontrar en qué instante de tiempo se inicia cada llamada CUDA a evaluar (transferencias y kernels) y cuánto dura su ejecución. Con un poco de esfuerzo es posible calcular el tiempo de ejecución de la aplicación paralela en las 4 GPUs.

cudaMemcpyAsync

Si miráis de nuevo el código, veréis que todas las comunicaciones entre el Host y el Device son del tipo `cudaMemcpyAsync`. En este punto, deberíamos preguntarnos, ¿porqué usamos `cudaMemcpyAsync` y no `cudaMemcpy`? Para responder a esta pregunta, os recomendamos hacer lo siguiente:

1. Ejecutad el código original con la siguiente llamada:

   ```
   nsys nvprof --print-gpu-trace ./kernel4GPUs.exe 2048 N
   ```

   Observad en qué orden se ejecutan las transferencias y los kernels.
2. Modificad el código para que todas las transferencias sean del tipo `cudaMemcpy` y repetid la llamada anterior. Observad de nuevo en qué orden se ejecutan las transferencias y los kernels, y veréis claramente porqué usamos `cudaMemcpyAsync` y no `cudaMemcpy`.

CUDA Samples y Documentación

Si tenéis tiempo es un buen momento para mirar los ejemplos de CUDA que publica NVIDIA. Antes, estos ejemplos venían con el software de CUDA. Ahora, estos ejemplos actualizados, se pueden obtener en: https://github.com/NVIDIA/cuda-samples

El ejercicio que os proponemos es que escojáis uno de los ejemplos (los que hay en el directorio `0_Simple`, son los más asequibles), lo compiléis y lo ejecutéis. Para compilarlo tendréis que modificar el `Makefile` y definir un nuevo `job.sh`.

Y aprovechando, podéis darle un vistazo a la documentación que ofrece CUDA. La podréis encontrar en https://docs.nvidia.com/cuda (ahí está la información para todas las versiones del compilador).

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 está el kernel solucionado.