UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Programación de GPUs con CUDA y OpenCL (PGPU)

Tarea 5

Introducción

Esta parte contiene una serie de ejercicios propuestos relativos a la Computación Paralela Heterogénea.

Ejercicio 1

Vamos a transferir datos del host al dispositivo y viceversa. Sigue los siguientes pasos donde están marcados en el código fuente de ejemplo (`TransferenceExample.cu`):

1. Asigna espacio de memoria para las matrices `A` y `B`, de tamaño `m`×`n`, en el host con la función usual `malloc` o `calloc`. Las matrices deben estar en precisión simple (`float`). Por ejemplo, utilizar

   ```
   A = (float*) malloc(m*n*sizeof(float));
   ```

2. Rellena la matriz `A` con datos generados aleatoriamente (utilizar la función `rand`) donde cada elemento esté en el rango $[-1.0, 1.0]$. Por ejemplo, a un elemento de la matriz `A` se le puede asignar un valor aleatorio así `A(i,j) = ( 2.0f * (float) rand() / RAND MAX ) - 1.0f;`.

3. Asigna memoria para las dos matrices de tamaño `m`×`n` llamadas `d_A` y `d_B` en la memoria del dispositivo. Utilizar la función de CUDA `cudaMalloc`, por ejemplo así:

   ```
   CUDA_SAFE_CALL( cudaMalloc( (void **) &d_A, m*n*sizeof(float) ) );
   ```

4. Copia la memoria del host, es decir, la matriz `A` a la memoria del dispositivo en la matriz correspondiente `d_A` usando la función `cudaMemcpy`. Por ejemplo, utilizando

   ```
   CUDA SAFE CALL( cudaMemcpy( d_A, A, m*n*sizeof(float), cudaMemcpyHostToDevice ) );
   ```

   El primer argumento de `cudaMemcpy` siempre es el destino, y el segundo es el origen, cualquiera que sea la dirección de la transferencia utilizada. El último argumento indica esta dirección, que debería ser `cudaMemcpyDeviceToHost` para una transferencia de dispositivo a host.

5. Copia la matriz del dispositivo `d_A` en la matriz del dispositivo `d_B` utilizando la función `cudaMemcpy`. Hay que prestar atención a que ahora se trata de una transferencia dentro del dispositivo, es decir, de dispositivo a dispositivo.

6. Vuelve a copiar los datos (resultantes) de la matriz del dispositivo `d_B` en la matriz de memoria del host `B`.

7. Libera las matrices en la memoria del dispositivo (`d_A` y `d_B`) con la rutina `cudaFree`.

8. Libera las tres matrices del host `A` y `B` con la rutina `free`.

El programa se compila con `nvcc -o TransferenceExample TransferenceExample.cu`. Ejecuta el programa y verifica si hay errores de sintaxis o de ejecución (el error obtenido debería ser 0 si es correcto). Aumenta la cantidad de memoria que utiliza. Agrega al programa un cálculo de la cantidad total de bytes asignados al dispositivo para evitar quedarse sin memoria.

Ejercicio 2

En este ejercicio vamos a realizar una multiplicación matriz-matriz utilizando la librería de CUDA CUBLAS. Sigue los siguientes pasos:

1. Abre el archivo `MatrixMatrixMultiplication.cu`.
2. Lo primero que necesitas para usar CUBLAS es incluir el archivo de encabezado adecuado (línea 7). Búscalo aquí.
3. Para usar las rutinas de CUBLAS necesitas un manejador de CUBLAS (línea 58). Este es un objeto de tipo `cublasHandle_t`. Llámalo `handle`.
4. Inicialízalo usando la rutina `cublasCreate` en el lugar indicado (línea 59).
5. Rellena los espacios dentro de las macros `CUDA_SAFE_CALL()` para transferir las matrices `A` y `B` al dispositivo (líneas 77 y 78).
6. Rellena el espacio vacío dentro de la macro `CUBLAS_SAFE_CALL()` escribiendo una llamada a la rutina de CUBLAS `cublasDgemm` (línea 79). Encontrarás información sobre la API aquí. Puedes utilizar las constantes ya declaradas `ZERO` y `ONE`.
7. Transfiere la matriz `d_C` a `gpu_C` (línea 82).
8. No olvides destruir el manejador de CUBLAS al final del código fuente utilizando la rutina correspondiente (linea 93).

La compilación es sencilla y se realiza igual que en el ejercicio anterior solo que aquí hay que incluir la biblioteca CUBLAS (`-lcublas`).

Ejercicio 3

Con este ejercicio aprenderemos la capacidad de los `stream`'s (flujos) de CUDA para mejorar el rendimiento al superponer la transferencia de datos y el cálculo en la GPU.

El ejercicio consiste en implementar una multiplicación de matrices $A \times B$ particionando la matriz `A` de modo que diferentes flujos puedan realizar diferentes trabajos al mismo tiempo. Particionamos la matriz `A` de CPU (y `d_A` en la GPU) de tamaño `m`×`p` en bloques de tamaño `k`×`p`, como se muestra en la Figura 1. La idea es tener una cantidad de flujos igual a m/k, cada uno de ellos haciendo el mismo trabajo con diferentes datos. El código `MatMulStreams.cu` pide matrices de tamaño `m`, `p`, `n` y la cantidad de flujos (`nstreams`), por lo que el tamaño de bloque se calcula como `k=m/nstreams`. El cálculo realizado por cada flujo consiste en una secuencia de tres operaciones:

1. crear un `stream`,
2. una transferencia de datos de un bloque de `A` al bloque correspondiente de la matriz `d_A` en la GPU,
3. una multiplicación matricial del bloque de `d_A` por la matriz `d_B`,
4. una transferencia de datos del bloque resultante de `d_D` a la CPU, y
5. destruir el `stream` creado al principio de la iteración.

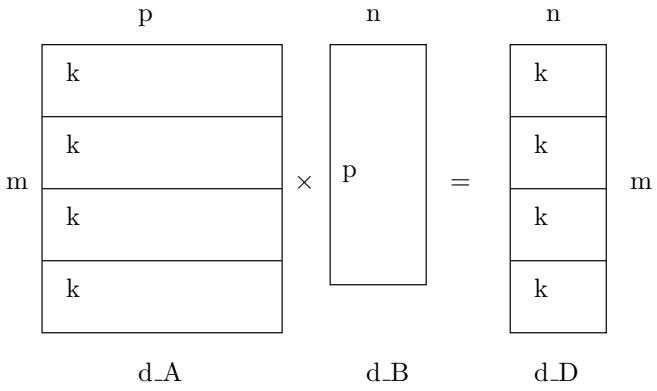

Figura 1: Multiplicación de matrices con `stream`'s.

El ejercicio consiste en reemplazar los comentarios en los espacios indicados por las acciones requeridas en el fichero fuente. Recuerda utilizar memoria *pinned* (`HostAlloc`) cuando sea necesario asignar memoria en el host. Para la transferencia de datos es conveniente utilizar rutinas de CUBLAS ya que es más fácil seleccionar el bloque de matriz apropiado para enviar/recibir (`cublasGetMatrixAsync` y `cublasSetMatrixAsync`). Hay que utilizar versiones asíncronas ya que la operación debe ser asíncrona con respecto al host para operar como se espera. Para incluir una operación de CUBLAS en un flujo CUDA determinado es necesario establecer (*set*) este flujo justo antes de la función de llamada con cublasSetStream. Es importante tener en cuenta que las matrices están almacenadas en memoria por COLUMNAS, ya que así trabajan las funciones de BLAS/LAPACK. Para compilar el ejercicio tenéis el fichero `Makefile.MatMulStreams`.

El programa debe mostrar un valor pequeño como error para comprobar si es correcto. Si variamos el número de `stream`'s (`k`), podemos encontrar un valor de equilibrio que permita reducir el tiempo de ejecución de la versión con `stream`'s con respecto a la secuencial.

Ejercicio 4

Con este ejercicio aprenderemos a utilizar más de una GPU si la tenemos disponible en nuestra máquina (podéis utilizar el comando `nvidia-smi` para saber cuántas GPUs hay en el nodo, o también la herramienta utilizada en el primer ejercicio, que muestra más información). La posibilidad de utilizar varias GPUs puede realizarse con `streams` de CUDA y/o operaciones asíncronas, sin embargo, utilizaremos otro mecanismo más básico y natural que consiste en crear dos hilos de OpenMP, cada uno de ellos dedicado a enviar operaciones a una GPU.

En este ejercicio vamos a utilizar dos GPUs. El objetivo es dividir un solo cálculo en dos partes para que cada una de ellas se dirija a cada una de las dos GPUs disponibles en nuestro nodo. Para simplificar, utilizaremos el ejemplo de multiplicación de matrices.

Dado el producto $C = A \times B$, $C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times p}$ y $B \in \mathbb{R}^{p \times n}$, con matrices `A`, `B` y `C` creadas e inicializadas en CPU, vamos a seguir los siguientes pasos:

1. Suponemos un particionado de la matriz `A` como el de la Figura 1 con dos bloques de tamaño $k \times p$, donde $k = m/2$, es decir,

$$C = A \times B \rightarrow \begin{pmatrix} C_0 \\ C_1 \end{pmatrix} = \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} \times B.$$

2. Establecer la GPU 0 como objetivo con

```
CUDA_SAFE_CALL( cudaSetDevice( 0 ) );
```

3. Reservar memoria para las matrices necesarias en GPU.

4. Enviar el bloque A_0 y la matriz B a la GPU 0.

5. Realizar el producto $C_0 = A_0 \times B$ en la GPU 0.

6. Enviar el bloque C_0 de la GPU 0 al host.

Como puede verse en el código ejemplo facilitado (`MatMulTwoGPUs.cu`), se crean dos secciones de OpenMP, cada una de ellas se ejecutará en paralelo y se encargará de enviar operaciones a una GPU. Los pasos descritos anteriormente se corresponden con la primera sección, la cuál, se encarga de enviar operaciones a la GPU 0. La otra sección, por tanto, hará lo propio con la GPU 1. En este seguno caso hay que tener en cuenta que

- Se debe enviar el bloque A_1 y la matriz B a la GPU 1.
- El producto a realizar es $C_1 = A_1 \times B$ en la GPU 1.
- Se debe enviar el resultado, que es el bloque C_1, de la GPU 1 al host.

En el host se comprueba que el resultado es correcto. Además, se incluyen las rutinas de toma de tiempos para comparar el tiempo de ejecución con una y con dos GPUs.