PRACTICA

GPU Computing

María González Herrero

Santiago Molpeceres Díaz

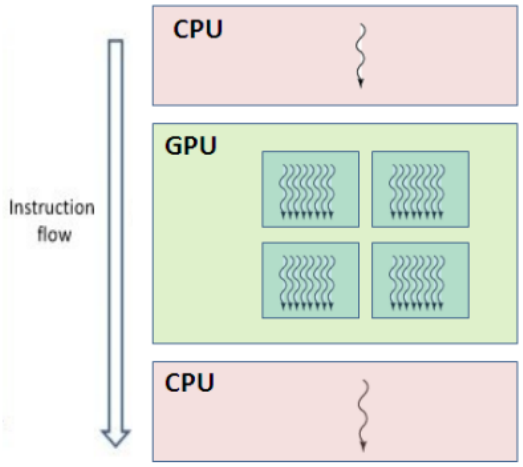
UNIVERSIDAD NEBRIJA

Índice

Introducción

Un programa CUDA es un programa híbrido en el que:

- El código secuencial se ejecuta en la CPU
- El código paralelo se ejecuta en la GPU



(a)

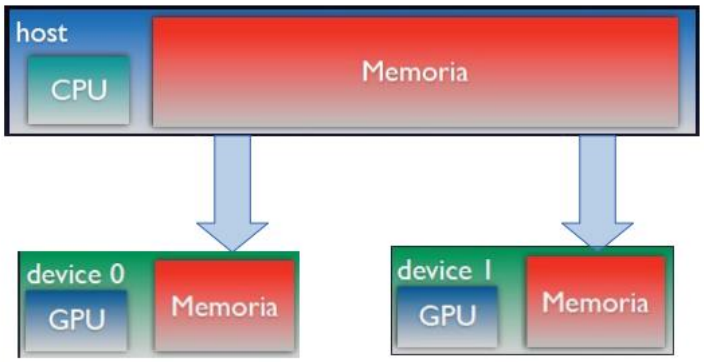
Un Programa CUDA invoca funciones paralelas denominadas Kernels.
Un Kernel se ejecuta en paralelo a través de threads paralelos.

Por tanto, para la realización del ejercicio se dividirán las tareas del siguiente modo.

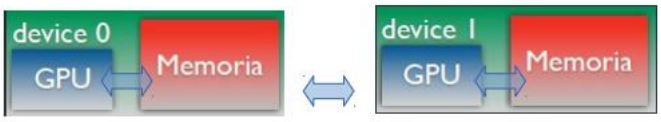
- CPU
  - Inicializaciones
  - Lectura de datos
  - Distribución de datos
  - Recepción de resultados
  - Muestra los resultados
- GPU
  - Cálculos

La diferencia de estos dos componentes es que la CPU corre los programa a modo de HOST y la GPU corre programas a modo de DEVICE.

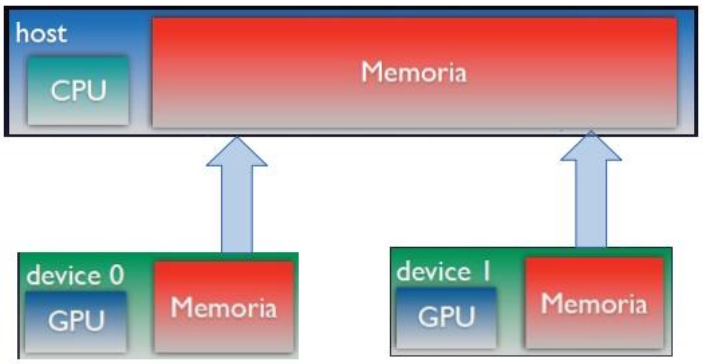
Los módulos device, tienen una pequeña memoria integrada en la que reciben los datos de la memoria principal, generan una copia, y luego modifican la memoria principal tal que así.



//



//



Código

Primero declaramos los espacios en la memoria principal que vamos a utilizar.

```
    float *A = (float *)malloc(N * sizeof(float));
    float *B = (float *)malloc(N * sizeof(float));
    float *resolver = (float *)malloc(N * sizeof(float));
```

- A -> Primera array involucrada.
- B -> Segunda array involucrada.
- Resolver -> Array con la solución suma.

Luego declaramos las variables que va a utilizar la GPU

```
float *d_A, *d_B, *d_resolver;
```

Estas actúan como copias de los arrays de la memoria principal de la CPU;esta función les asigna un espacio de memoria en la memoria del device.

```
    cudaMalloc((void **)&d_A, sizeof(float) * N);
    cudaMalloc((void **)&d_B, sizeof(float) * N);
    cudaMalloc((void **)&d_resolver, sizeof(float) * N);
```

Más tarde llenamos las arrays de la memoria principal tal que el array A va a estar llena de “1” y el array B va a estar llena de “2”.

```
    for (int i = 0; i < N; i++)
    {
        *A[i] = 1.0f;
        *B[i] = 2.0f;
    }
```

Transferimos los datos del módulo HOST al módulo DEVICE con la siguiente función.

```
    cudaMemcpy(d_A, A, sizeof(float) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, sizeof(float) * N, cudaMemcpyHostToDevice);
```

Para el control de los kernels hacemos las siguientes declaraciones

Las declaraciones de tipo Dim3 son vectores de 3 enteros que se utilizan para especificar dimensiones. Como componentes x.y.z . si alguno de estos no se inicializa, este valdrá 1.

```
    dim3 nthreads(256);
    dim3 nblocks((N / nthreads.x) + (N % nthreads.x ? 1 : 0));
```

- Nthreads => Recoge el número total de hilos.
- Nblocks => Recoge el número de bloques que vamos a utilizar.

Llamamos a la función kernel.

```
 kernel_suma<<<nblocks.x, nthreads.x>>>(A,B, N);
```

Cuya función se corresponde con:

```
__global__ void kernel_suma(float *v1, float *v2, int dim)
{
    int id = threadIdx.x + (blockIdx.x * blockDim.x);
    if (id < dim)
    {
        v1[id] = v1[id] + v2[id];
    }
}
```

La cual ejecuta la suma por bloques.

Con la siguiente función hacemos que se sincronicen y esperemos hasta que todos los kernels queden resueltos.

```
cudaDeviceSynchronize();
```

Devolvemos los valores a él array resolver.

```
 cudaMemcpy(resolver, A, sizeof(float) * N,
      cudaMemcpyDeviceToHost);
```

y liberamos la memoria, tanto principal como perteneciente a los device.

```
    free(A);
    free(B);
    free(resolver);
    cudaFree(d_A);
    cudaFree(d_B);
/* El código no se ha podido probar ya que no tenemos NVIDIA*/
```

WEBGRAFÍA

https://www.it-swarm-es.com/es/cuda/mejores-practicas-de-memoria-constante-de-cuda/1040623943/

http://dis.um.es/~domingo/apuntes/PPCAP/1718/PAM_2017_01_CUDA_basico.pdf

https://www.etsisi.upm.es/sites/default/files/asigs/arquitecturas_avanzadas/practicas/CUDA/guiacuda.pdf