ARQUITECTURA
DE
COMPUTADORES
PRÁCTICA 5

Ignacio Jesús García Estévez

Introducción

En esta quinta y última práctica veremos un caso mas real de paralelización de procesos utilizando la GPU que es un hardware dedicado a el procesamiento de tareas en paralelo. Para esto utilizaremos una tecnología creada por NVIDIA llamada CUDA, que mediante la programación en C o C++ nos permite especificar que partes del código queremos que ejecute el host (CPU) y cuales queremos que ejecute el Device (GPU). Realizaré esta practica a través de Google Colab que me permite utilizar las funcionalidades de una tarjeta grafica NVIDIA sin necesidad de tener una en mi PC.

Ejercicios

Ejercicio 1

En este ejercicio hay que modificar el ejemplo dado, para que en vez de crear un bloque con 100 hilos se cree un bloque de 25 hilos y se ejecute 4 veces. Para hacer esto hay que cambiar la ejecución del kernel, y poner <<1,25>> para que sea 1 bloque de 25 hilos, y en la función del Kernel CUDA habrá que hacer un for de 4 iteraciones en la que la posición de suma_device sea el id del thread + 25 ya que el bloque tiene 25 threads * por la iteración en la que está para que lo haga 100 veces.

```
%%cu
#include <iostream>
#include <cuda_runtime.h>
#define T_VEC 100
//DEFINICIÓN KERNEL CUDA
__global__ void sumaVectores(int* v1_device, int* v2_device, int*
suma_device) {
printf("Índice de hilo dentro del bloque: %d\n", threadIdx.x);
printf("Índice del bloque: %d\n", blockIdx.x);
printf("Tamaño del bloque (número de hilos): %d\n", blockDim.x);
//al tener 25 hilos se ejecuta 25 veces por bloque entonces lo unico quue hay que hacer es
//hacer un bucle de 4 iteraciones sumandole al id del thread 25* la iteracion asía haciendose solo 100 veces
for (int i = 0; i < 4; i++) {
suma_device[threadIdx.x+(25*i)] = v1_device[threadIdx.x] + v2_device[threadIdx.x];
  printf("I del bucle: %d\n", (threadIdx.x+(25*i)));

}

}
//FUNCIÓN DEL HOST
int main() {
//1 - INICIALIZAR DATOS DEL HOST
int v1_host[T_VEC], v2_host[T_VEC], suma_host[T_VEC];
for (int i = 0; i < T_VEC; i++) {
v1_host[i] = 1;
v2_host[i] = 2;
}
//2 - DECLARAR Y ASIGNAR MEMORIA PARA EL DEVICE
int *v1_device, *v2_device, *suma_device; //punteros al device
cudaMalloc((void**)&v1_device, sizeof(int) * T_VEC); //asignar memoria para el device
cudaMalloc((void**)&v2_device, sizeof(int) * T_VEC);
cudaMalloc((void**)&suma_device, sizeof(int) * T_VEC);
//3 - TRANSFERIR DATOS DEL HOST AL DEVICE
cudaMemcpy(v1_device, v1_host, sizeof(int) * T_VEC,
cudaMemcpyHostToDevice); //transferir el valor del host al device
cudaMemcpy(v2_device, v2_host, sizeof(int) * T_VEC,
cudaMemcpyHostToDevice);
cudaMemcpy(suma_device, suma_host, sizeof(int) * T_VEC,
cudaMemcpyHostToDevice);
//4 - EJECUTAR EN UNO O MÁS KERNELS
sumaVectores<<<1, 25>>>(v1_device, v2_device, suma_device);
//lanzar el kernel
cudaDeviceSynchronize(); //espera hasta que el kernel termine
//5 - TRANSFERIR DATOS DEL DEVICE AL HOST
cudaMemcpy(suma_host, suma_device, sizeof(int) * T_VEC,
cudaMemcpyDeviceToHost); //transferir el valor del device al host
//6 - LIBERAR RECURSOS
cudaFree(v1_device); //liberar memoria del device
cudaFree(v2_device);
cudaFree(suma_device);
//MOSTRAR RESULTADO
for (int i = 0; i < T_VEC; i++) {
std::cout << suma_host[i] << " ";
}
std::cout << "\n";
return 0;
}
```

Solución

```
I del bucle: 73
I del bucle: 74
I del bucle: 75
I del bucle: 76
I del bucle: 77
I del bucle: 78
I del bucle: 79
I del bucle: 80
I del bucle: 81
I del bucle: 82
I del bucle: 83
I del bucle: 84
I del bucle: 85
I del bucle: 86
I del bucle: 87
I del bucle: 88
I del bucle: 89
I del bucle: 90
I del bucle: 91
I del bucle: 92
I del bucle: 93
I del bucle: 94
I del bucle: 95
I del bucle: 96
I del bucle: 97
I del bucle: 98
I del bucle: 99
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

Ejercicio 2

En este segundo ejercicio se pide realizar una suma de 4 vectores con 200 valores y que se ejecute en 10 bloques de 20 hilos. Para esto utilizaré como base el ejercicio anterior, y hay que añadir 2 nuevos vectores, en la definición del kernel en la suma, al inicializar el host al declarar y asignar la memoria del Device y al transferir los datos del host al Device, también al realizarse una suma por cada thread no habrá que hacer un bucle, solo ponerle el índice de el id del thread + el id del bloque *20 ya que cada bloque tiene 20 threads.

```
%%cu
#include <iostream>
#include <cuda_runtime.h>
#define T_VEC 200
//DEFINICIÓN KERNEL CUDA
__global__ void sumaVectores(int* v1_device, int* v2_device,int* v3_device,int* v4_device, int*
suma_device) {

printf("Índice del bloque: %d\n", blockIdx.x);
printf("Índice de hilo dentro del bloque: %d\n", (threadIdx.x+(blockIdx.x*20)));
//como en este caso hay 10 bloques de 20 hilos no es necesario hacer ningún bloque para que se haga las 200 veces
//simplemente hay que hacer la suma y el indice en el que se guarda es el id del thread + el id del bloque* 20 ya que cada bloque tiene 20 threads

suma_device[threadIdx.x+(blockIdx.x*20)] = v1_device[threadIdx.x+(blockIdx.x*20)] + v2_device[threadIdx.x+(blockIdx.x*20)]
 +v3_device[threadIdx.x+(blockIdx.x*20)]+v4_device[threadIdx.x+(blockIdx.x*20)];

}
//FUNCIÓN DEL HOST
int main() {
//1 - INICIALIZAR DATOS DEL HOST
int v1_host[T_VEC], v2_host[T_VEC], v3_host[T_VEC], v4_host[T_VEC], suma_host[T_VEC];
for (int i = 0; i < T_VEC; i++) {
v1_host[i] = 1;
v2_host[i] = 2;
v3_host[i] = 3;
v4_host[i] = 4;
}
//2 - DECLARAR Y ASIGNAR MEMORIA PARA EL DEVICE
int *v1_device, *v2_device,*v3_device,*v4_device, *suma_device; //punteros al device
cudaMalloc((void**)&v1_device, sizeof(int) * T_VEC); //asignar memoria para el device
cudaMalloc((void**)&v2_device, sizeof(int) * T_VEC);
cudaMalloc((void**)&v3_device, sizeof(int) * T_VEC);
cudaMalloc((void**)&v4_device, sizeof(int) * T_VEC);
cudaMalloc((void**)&suma_device, sizeof(int) * T_VEC);
//3 - TRANSFERIR DATOS DEL HOST AL DEVICE
cudaMemcpy(v1_device, v1_host, sizeof(int) * T_VEC,
cudaMemcpyHostToDevice); //transferir el valor del host al device
cudaMemcpy(v2_device, v2_host, sizeof(int) * T_VEC,
cudaMemcpyHostToDevice);
cudaMemcpy(v3_device, v3_host, sizeof(int) * T_VEC,
cudaMemcpyHostToDevice);
cudaMemcpy(v4_device, v4_host, sizeof(int) * T_VEC,
cudaMemcpyHostToDevice);
cudaMemcpy(suma_device, suma_host, sizeof(int) * T_VEC,
cudaMemcpyHostToDevice);
//4 - EJECUTAR EN UNO O MÁS KERNELS
sumaVectores<<<10, 20>>>(v1_device, v2_device,v3_device,v4_device, suma_device);
//lanzar el kernel
cudaDeviceSynchronize(); //espera hasta que el kernel termine
//5 - TRANSFERIR DATOS DEL DEVICE AL HOST
cudaMemcpy(suma_host, suma_device, sizeof(int) * T_VEC,
cudaMemcpyDeviceToHost); //transferir el valor del device al host
//6 - LIBERAR RECURSOS
cudaFree(v1_device); //liberar memoria del device
cudaFree(v2_device);
cudaFree(v3_device);
cudaFree(v4_device);
cudaFree(suma_device);
//MOSTRAR RESULTADO
for (int i = 0; i < T_VEC; i++) {
std::cout << suma_host[i] << " ";
}
std::cout << "\n";
return 0;
}
```

Solución

```
Índice de hilo dentro del bloque: 61
Índice de hilo dentro del bloque: 62
Índice de hilo dentro del bloque: 63
Índice de hilo dentro del bloque: 64
Índice de hilo dentro del bloque: 65
Índice de hilo dentro del bloque: 66
Índice de hilo dentro del bloque: 67
Índice de hilo dentro del bloque: 68
Índice de hilo dentro del bloque: 69
Índice de hilo dentro del bloque: 70
Índice de hilo dentro del bloque: 71
Índice de hilo dentro del bloque: 72
Índice de hilo dentro del bloque: 73
Índice de hilo dentro del bloque: 74
Índice de hilo dentro del bloque: 75
Índice de hilo dentro del bloque: 76
Índice de hilo dentro del bloque: 77
Índice de hilo dentro del bloque: 78
Índice de hilo dentro del bloque: 79
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
```

Ejercicio 3

El numero de bloques generalmente dependerá de lo grande o pequeño que sea el programa que estemos realizando y también dependiendo del ancho de banda con el que se esté trabajando, sin embargo el numero de threads dependerá mas de metricas como la sincronización y el overhead, si hay mucha sincronización se utilizarán menos hilos por bloque y si el overhead es muy pequeño se podrán utilizar muchos mas threads sin que los cambios de contexto supongan demasiado problema.

Conclusiones

Con esta práctica hemos visto una aplicación más realista de los problemas que hemos estado diseñando durante todas las anteriores prácticas, he podido entender de manera superficial como compartir información entre el Host y el Device utilizando CUDA y con el ejercicio 3 he utilizado los conocimientos de la teoría de la asignatura.