Arquitectura de Computadores

Grado en Ingeniería Informática

Práctica 5

Jose Francisco Romero Rodríguez

Sara Marcos Cornejo

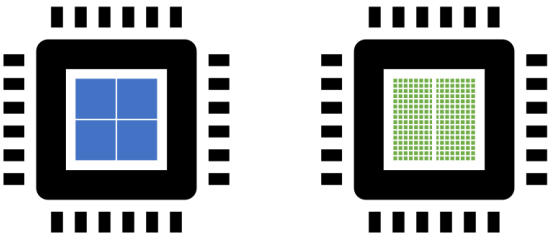
UNIVERSIDAD NEBRIJA

Índice:

Introducción:

GPU Computing. ¿Qué es?

La computación con GPU es el uso de una GPU (unidad de procesamiento de gráficos) como coprocesador para acelerar las CPU para la computación científica y de ingeniería de propósito general. Es la pieza de silicio que tanto AMD como NVIDIA o Intel fabrican y donde se graban los transistores. Algunas diferencias entre la GPU y la CPU serían las siguientes:



CPU	GPU
Central Processing Unit	Graphics Processing Unit
4-8 Cores	100s or 1000s of Cores
Low Latency	High Throughput
Good for Serial Processing	Good for Parallel Processing
Quickly Process Tasks That Require Interactivity	Breaks Jobs Into Separate Tasks To Process Simultaneously
Traditional Programming Are Written For CPU Sequential Execution	Requires Additional Software To Convert CPU Functions to GPU Functions for Parallel Execution

Host **Device**

Problema: Lenguajes diseñados para realizar gráficos en pantalla

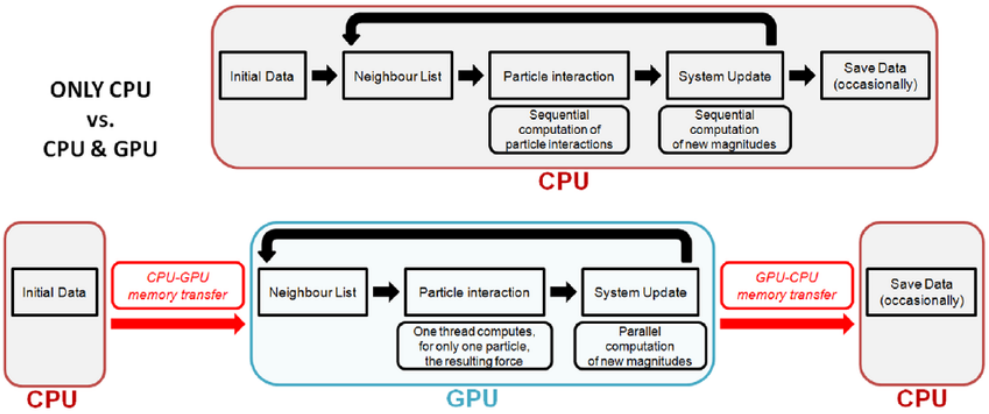
CUDA:

CUDA surge como solución al problema del GPU Computing. Son las siglas de Compute Unified Device Architecture, que hace referencia a una plataforma de computación en paralelo. Incluye un compilador y un conjunto de herramientas de desarrollo creadas por Nvidia que permiten a los programadores usar una variación del lenguaje de programación C o C++ (CUDA C) para codificar algoritmos en GPU de Nvidia.

CUDA se trata de un programa híbrido entre la GPU y la CPU.

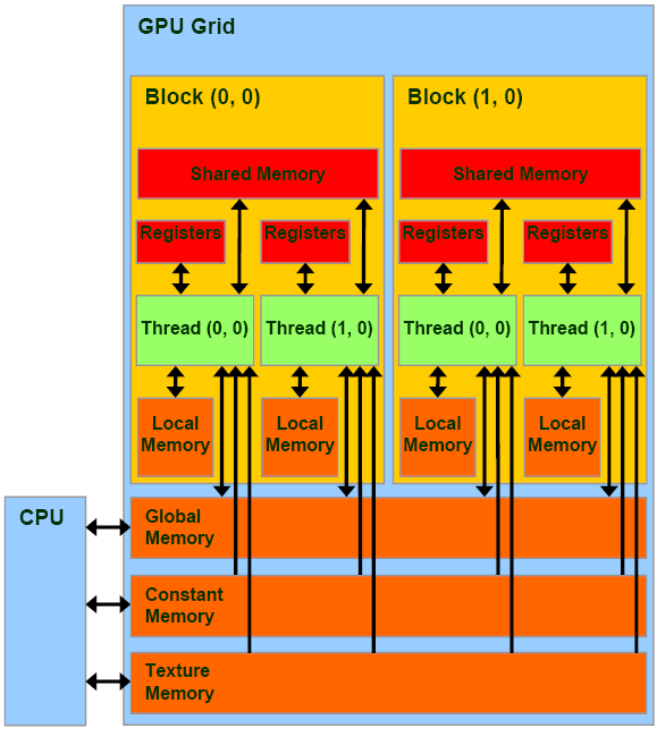
- La CPU ejecuta el código secuencial
- La GPU ejecuta el código paralelo.

En la siguiente imagen se puede observar la diferencia del flujo de instrucciones entre un sistema que usa solamente la CPU frente a este sistema híbrido entre CPU y GPU.



Su estructura sería la siguiente:

Los hilos están contenidos en bloques, y los bloques dentro de grids.



Cada hilo y cada bloque tiene un identificador, que correspondería con: threadIdx y blockIdx.

- Cada hilo posee su memoria local privada
- Cada bloque de hilos posee su memoria compartida
- Todos los hilos pueden acceder a la memoria global

Los hilos CUDA se ejecutan en el device, el resto del programa en el host.

Algunas de las ventajas que supone la implementación mediante CUDA serían:

- **Lecturas dispersas**: se puede consultar cualquier posición de memoria.
- **Memoria compartida**: CUDA pone a disposición del programador un área de memoria de 16KB que se compartirá entre hilos del mismo bloque. Dado su tamaño y rapidez puede ser utilizada como caché.
- **Lecturas más rápidas** de y hacia la GPU.
- **Soporte** para enteros y operadores a nivel de bit.

La secuencia de operaciones en CUDA es la siguiente:

1. **Declarar y asignar memoria para host y device.**
2. **Inicializar datos del host.**
3. **Transferir datos del host al device.**
4. **Ejecutar uno o más kernels.**
5. **Transferir datos desde el device al host.**
6. **Liberar recursos.**

Compilación y ejecución de programas

NVCC es el compilador de NVIDIA. Este separa código que se ejecuta en el host y en el device:

- Las funciones del device (kernel()) se procesan por el NVIDIA compilador
- Las funciones del host (main())se compilan con los compiladores clásicos como gcc.

Compilación de un programa CUDA: nvcc codigo_fuente.cu -o ejecutable
Ejecución de programas MPI: ./ejecutable

Profile: nvprof ./ejecutable

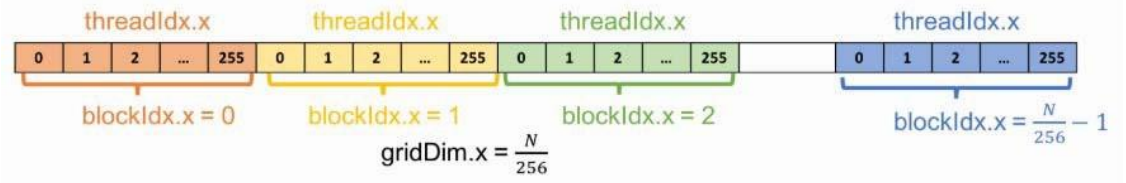
Ejemplo suma de vectores en el device en paralelo

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <cuda.h>
#include <cuda_runtime.h>
#define N 10000000
#define MAX_ERR 1e-6
__global void vector_add(float *out, float *a, float *b, int n) {
	int index = threadIdx.x;
	int stride = blockDim.x;
	for(int i = index; i < n; i += stride){
		out[i] = a[i] + b[i];
	}
}

int main(){
	float *a, *b, *out;
	float *d_a, *d_b, *d_out;

	// Allocate host memory
	a = (float*)malloc(sizeof(float) * N);
	b = (float*)malloc(sizeof(float) * N);
```

```
	out = (float*)malloc(sizeof(float) * N);

	// Initialize host arrays
	for(int i = 0; i < N; i++){
		a[i] = 1.0f;
		b[i] = 2.0f;
	}

	// Allocate device memory
	cudaMalloc((void**)&d_a, sizeof(float) * N);
	cudaMalloc((void**)&d_b, sizeof(float) * N);
	cudaMalloc((void**)&d_out, sizeof(float) * N);

	// Transfer data from host to device memory
	cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
	cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);

	// Executing kernel
	vector_add<<<1,256>>>(d_out, d_a, d_b, N);

	// Transfer data back to host memory
	cudaMemcpy(out, d_out, sizeof(float) * N,
cudaMemcpyDeviceToHost);

	// Verification
	for(int i = 0; i < N; i++){
		assert(fabs(out[i] - a[i] - b[i]) < MAX_ERR);
	}
	printf("PASSED\n");

	// Deallocate device memory
	cudaFree(d_a);
	cudaFree(d_b);
	cudaFree(d_out);

	// Deallocate host memory
	free(a);
	free(b);
	free(out);
}
```

Ejercicio 1:

Generar un código con el ejemplo anterior de la suma de vectores, añadiendo múltiples bloques que aumenten el grado de paralelización. En el caso de no poder probar el código en CUDA explicar en la memoria claramente los pasos seguidos para la implementación.



En primer lugar, se implementa la **función** que realiza la suma de vectores en el device con el kernel vector_add, siendo index el índice del hilo y stride el número de hilos por bloque.

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
int index = threadIdx.x; int stride = blockDim.x;
for(int i = index; i < n; i += stride){
    out[i] = a[i] + b[i];
    }
}
```

Se declaran **las variables** a utilizar, siendo:

- a, b y out los punteros para los vectores en el host
- d_a, d_b, d_out los punteros para los vectores en el host

```
float *a, *b, *out;
float *d_a, *d_b, *d_out;
```

Seguidamente se realiza **la reserva de memoria en el host** mediante la instrucción malloc para cada vector de esta forma:

```
a = (float*)malloc(sizeof(float) * N);
b = (float*)malloc(sizeof(float) * N);
out = (float*)malloc(sizeof(float) * N);
```

Una vez que cada vector tiene un espacio de memoria reservado, se **inicializan** los dos vectores de entrada del host. En este caso se le asigna al vector a el valor de 1f y al vector b el valor de 2f.

```
for(int i = 0; i < N; i++){
    a[i] = 1.0f;
    b[i] = 2.0f;
}
```

A continuación, se realiza el mismo proceso de asignación de memoria para cada vector, pero esta vez para el device. Esto se realiza mediante cudaMalloc

```
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);
```

El siguiente paso es la **transferencia de los datos del host al device**, copiando los vectores a y b al device. A través de cudaMemcpyHostToDevice es posible hacer una copia de host a device.

```
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
```

blockDim.x indica el número de hilos por bloque que habrá. Con blockDim.x = 256, se ejecutará el kernel vector_add con 256 hilos por bloque.

```
blockDim.x = 256;
```

Para aumentar el grado de paralelización existen las siguientes alternativas:

1. aumentar el número de bloques
2. aumentar el número de hilos por bloque
3. aumentar el tamaño del vector

En este caso, como se pide aumentar el número de bloques se usará la instrucción gridDim.x:

gridDim.x indica el número de bloques que habrá. Con blockGrid = (2*N + blockDim.x - 1) / blockDim.x. Se ejecutará el kernel vector_add, y el grado de paralelización será mayor.

```
gridDim.x = (2*N + blockDim.x - 1) / blockDim.x;
```

A continuación, se **ejecuta el kernel vector_add** en el dispositivo con el número de bloques y el número de hilos por bloque establecidos anteriormente.

```
vector_add<<< gridDim.x, blockDim.x >>>(d_out, d_a, d_b, N);
```

Una vez ejecutado el kernel se procede a copiar el vector out del device al host, ahora con la instrucción alternativa: cudaMemcpyDeviceToHost (**copia de device a host**)

```
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

Para **comprobar** que el resultado es correcto se implementa una función de tal forma que se verifique que: out - a – b = 0.

Si no es el caso, se produce un error y se muestra un mensaje de error.

Para esta comprobación se utiliza:

- fabs: devuelve el valor absoluto de un número.
- assert: comprueba que la condición es verdadera.

```
for(int i = 0; i < N; i++){
assert(fabs(out[i] - a[i] - b[i]) < MAX_ERR);
}
printf("PASSED\n");
```

Se **libera la memoria de cada vector en el device** mediante cudaFree

```
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_out);
```

Por último, se realiza la **liberación de memoria de cada vector en el host** mediante free, cerrando y finalizando todo el proceso

```
free(a);
free(b);
free(out);
```

Conclusiones:

En esta primera práctica se ha podido tener un primer contacto con el funcionamiento de las GPUs y la programación paralela con CUDA. Algunas de las conclusiones que se han podido sacar de su implementación son:

- GPU útil como dispositivo acelerador de códigos con paralelismo de datos
- Permite aumentos impresionantes en el rendimiento del computador al aprovechar las unidades de procesamiento gráfico (GPU) alojadas en la tarjeta gráfica.
- Eficiencia basada en ejecución concurrente de miles de threads
- Gran número de unidades funcionales
- Control relativamente sencillo

Bibliografía:

- https://www.researchgate.net/figure/Flow-diagram-showing-the-differences-of-the-CPU-and-GPU-implementations-Implementation_fig1_51240348
- https://es.wikipedia.org/wiki/CUDA
- https://www.fdi.ucm.es/profesor/rhermida/AC-Grado/GPU-EjemploCUDA.pdf