Arquitectura de Computadores

Grado en Ingeniería Informática

Práctica 5

UNIVERSIDAD NEBRIJA

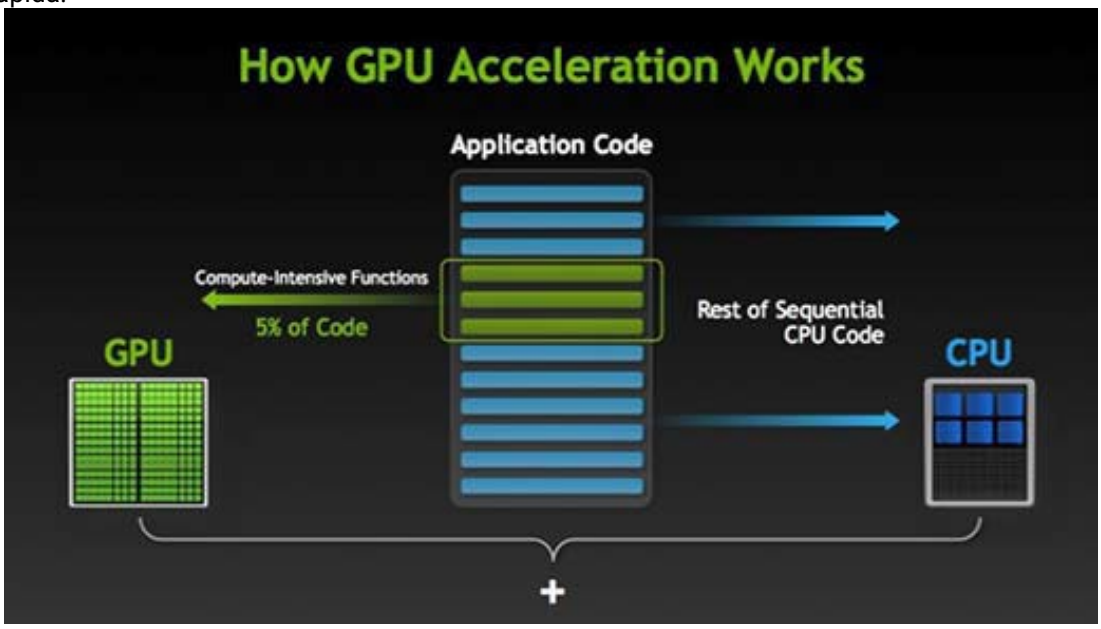
GPU Computing

Introducción

La computación acelerada por GPU es el uso de una unidad de procesamiento de gráficos (GPU, por sus siglas en inglés) junto a una CPU para acelerar el funcionamiento de las aplicaciones de aprendizaje profundo, análisis e ingeniería. Hoy en día, los aceleradores de GPU, una innovación de NVIDIA allá por el 2007, permiten el funcionamiento de centro de datos con eficiencia energética en laboratorios gubernamentales, universidades, empresas, y pequeñas y medianas empresas en todo el mundo. Tienen un papel muy importante en la aceleración de aplicaciones en plataformas, que abarcan desde la inteligencia artificial hasta automóviles, drones y robots.

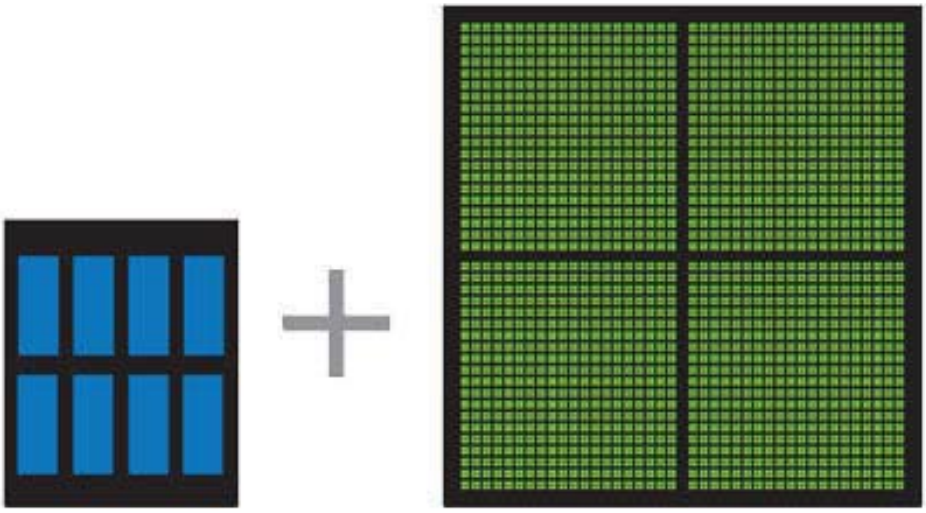
Aceleración hardware utilizando las GPUs

La computación acelerada por GPU permite asignar a la GPU el trabajo de los aspectos de la aplicación donde la computación es más intensiva, mientras que el resto del código se ejecuta en la CPU. Desde la perspectiva del usuario, las aplicaciones se ejecutan de forma mucho más rápida.



Rendimiento de la GPU vs. El de la CPU

Una forma sencilla de comprender la diferencia entre una GPU y una CPU es comparar la forma en que procesan las tareas. Una CPU tiene unos cuantos núcleos optimizados para el procesamiento en serie secuencial, mientras que una GPU cuenta con una arquitectura en paralelo enorme que consiste en cientos, miles de núcleos más pequeños y eficaces, y que se diseñaron para resolver varias tareas al mismo tiempo.
Las GPU tienen miles de núcleos para procesar cargas de trabajo en paralelo de forma eficiente:



Workflow: secuencia de operaciones CUDA

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

Objetivos

En esta práctica se analizará el modelo de computación por GPU denominado CUDA (extensión de C++ y APIs para programación de GPUs) para las arquitecturas de NVIDIA.
En esta práctica se aprenderá el concepto de comunicación colectivo entre procesos de MPI.
Los objetivos fijados son los siguientes:

- Qué es GPU computing.
- Workflow de aplicaciones GPU computing.
- Compilación, ejecución y profiling.
- Ejemplos de uso

Toda la información para poder arrancar con CUDA se puede encontrar en la siguiente dirección: https://developer.nvidia.com/cuda-downloads. Por supuesto es requisito indispensable contar con una GPU de NVIDIA que soporte CUDA (https://developer.nvidia.com/cuda-gpus).

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`

Ejemplos

Hola mundo

C	CUDA

`void c_hola(){`
`  printf("Hola Mundo!\n");`
`}`
`int main() {`
`  c_hola();`
`  return 0;`
`}`
	`__global__ void cuda_hola(){`
	`  printf("Hola Mundo desde GPU!\n");`
	`}`
	`int main() {`
	`  cuda_hola<<<1,1>>>();`
	`  return 0;`
	`}`

global: función que corre en el device (GPU). Se instancia desde el host (main) y recibe el nombre de kernel.

<<<...>>>: la ejecución del kernel viene provista de este operador conocido como configuración de la ejecución del kernel (kernel launch).

Suma de vectores

SumaVectores.c

```
#define N 10000000
void vector_add(float *out, float *a, float *b, int n) {
	for(int i = 0; i < n; i++){
		out[i] = a[i] + b[i];
	}
}
int main(){
	float *a, *b, *out;
	// Allocate memory
	a = (float*)malloc(sizeof(float) * N);
	b = (float*)malloc(sizeof(float) * N);
	out = (float*)malloc(sizeof(float) * N);
	// Initialize array
	for(int i = 0; i < N; i++){
		a[i] = 1.0f; b[i] = 2.0f;
	}
	// Main function
	vector_add(out, a, b, N);
}
```

SumaVectores.cu

```
__global__ void vector_add(float *out,
	float *a, float *b, int n) {
	for(int i = 0; i < n; i++){
		out[i] = a[i] + b[i];
	}
}
vector_add<<<1,1>>>(out, a, b, N);
```

Gestión de memoria del device:

```
cudaMalloc(void **devPtr, size_t count);
cudaFree(void *devPtr);
```

Transferencia de memoria:

```
cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind)
```

Compilación Ejecución y profile:

```
$> nvcc vector_add.cu -o vector_add
$> time ./vector_add
$> nvprof ./vector_add
```

```
==6326== Profiling application: ./vector_add
==6326== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 97.55%  1.42529s         1  1.42529s  1.42529s  1.42529s  vector_add(float*, float*,
float*, int)
  1.39%  20.318ms         2  10.159ms  10.126ms  10.192ms  [CUDA memcpy HtoD]
  1.06%  15.549ms         1  15.549ms  15.549ms  15.549ms  [CUDA memcpy DtoH]
```

Suma de Vectores en el Device:

```
#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) {
	for(int i = 0; i < n; i ++){
		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<<<1,1>>>(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("out[0] = %f\n", out[0]);
        printf("PASSED\n");

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

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

Paralelización Suma de Vectores: <<< B , T>>>

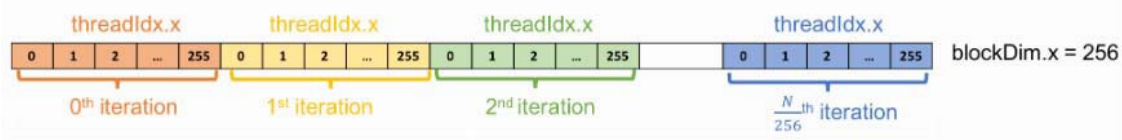
Se indican cuantos Threads se lanzan a través de su agrupación en bloques (B) y el número de threads por bloque: N_threads_total = B * T
Por ejemplo: `vector_add <<< 1 , 256 >>> (d_out, d_a, d_b, N);`

Variables de cuda built-in:

- threadIdx.x índice del thread dentro del bloque. (de 0 a 255 para el ejemplo).
- blockDim.x número de threads en el bloque. (256 para el ejemplo).

Si queremos ejecutar la suma de vectores con 256 threads como sugiere el ejemplo entonces en cada iteración cada thread ejecuta:

```
__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];
        }
}
```



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 (10 puntos)

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.