Introducción a CUDA

¿Cómo funciona la programación en C?

Cuando nosotros trabajamos en C o C++, el código que nosotros escribimos es traducido a lenguaje de máquina por el compilador. Este código de máquina es el que se ejecuta en el procesador. El procesador tiene un conjunto de instrucciones que puede ejecutar. Estas instrucciones son las que nosotros escribimos en C o C++. Esto quiere decir que el procesador ejecuta una instrucción a la vez. Esto es lo que se conoce como programación secuencial. Esto se debe a que el procesador está diseñado para ejecutar una instrucción a la vez.

¿Qué es la programación paralela?

La **programación paralela** es un paradigma de programación que consiste en dividir un problema en partes más pequeñas y resolver cada parte en paralelo. Esto quiere decir que cada parte se ejecuta en un procesador diferente. Esto se puede hacer porque los procesadores actuales y las tarjetas gráficas tienen más de un núcleo. Esto quiere decir que se pueden ejecutar más de una instrucción a la vez. Esto permite que se puedan ejecutar **más instrucciones en menos tiempo**.

¿Qué es CUDA?

CUDA es una plataforma de computación paralela desarrollada por NVIDIA™. Esta plataforma permite que se puedan ejecutar programas en paralelo en las tarjetas gráficas de NVIDIA™. Esto se puede hacer porque las tarjetas gráficas tienen cientos o a veces miles de núcleos. Esto quiere decir que se pueden ejecutar cientos de instrucciones a la vez. Esto permite que se puedan ejecutar más instrucciones en menos tiempo. Este tipo de programación se conoce como programación en paralelo y su uso en tarjetas gráficas se conoce como programación en GPU.

¿Cómo funciona CUDA?

CUDA significa Compute Unified Device Architecture. Esto es una plataforma de programación en paralelo y una interfaz de programación de aplicaciones (API) que permite que se puedan ejecutar programas y sin que estos obligatoriamente sean para la generación de imagenes. Ya que cuando las tarjetas gráficas se empezaron a usar a inicios de los años 90, estas solo se usaban para la generación de imagenes. Fue a inicios de los 2000 que los investigadores descubrieron que con algo de ingenio, era posible usar las tarjetas gráficas para hacer calculos generales. Esto cambió por completo la manera de trabajar principalmente debido a que las tarjetas gráficas al estar diseñadas para desarrollar calculos y procesos de forma paralela son mucho más rápidas que los procesadores **cuando el problema se puede dividir en partes más pequeñas**.

¿Cómo usar CUDA?

Para poder utilizar CUDA, es necesario tener una tarjeta gráfica de NVIDIA™. Luego es necesario tener instalado un compilador de C o C++ siendo el recomendado para esta tarea MSVC el cual es parte de la instalación de **Visual Studio** siendo importante que no se vaya a confundir con Visual Studio Code. Luego es necesario instalar el toolkit de CUDA. Este se puede descargar desde la página de NVIDIA™. Una vez instalado el toolkit, es necesario configurar el compilador para que pueda usar CUDA. Dado que trabajaremos

este curso con Visual Studio Code es necesario configurar la tarea de compilación para que pueda compilar el código en CUDA.

```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build - nvcc",
            "type": "shell",
            "command": "nvcc",
            "args": [
                "-g",
                "${file}",
                "-o",
                "${fileDirname}/${fileBasenameNoExtension}"
            ],
            "group": "build",
        }
    ]
```

Este es el código JSON que incluye las tareas de compilación. Este código debe ser copiado en el archivo `tasks.json` que se encuentra en la carpeta `.vscode` que se encuentra en la carpeta del proyecto. Este archivo se puede crear manualmente o se puede crear desde Visual Studio Code. Para crearlo desde Visual Studio Code, se debe abrir la paleta de comandos con `Ctrl + Shift + P` y escribir `Tasks: Configure Default Build Task`. Luego se debe seleccionar la opción `Create tasks.json file from template` y luego seleccionar la opción `Others`. Esto creará el archivo `tasks.json` en la carpeta `.vscode` del proyecto. Luego se debe copiar el código anterior en el archivo `tasks.json` y guardar el archivo. Una vez hecho esto podremos compilar código en CUDA desde Visual Studio Code utilizando la tecla `Ctrl + Shift + B`.

¿Cómo usar CUDA en un código de C?

Para iniciar con CUDA C no es necesario crear cosas demasiado complejas. Para empezar, se puede crear un programa que imprima un mensaje en la consola. Para esto se puede usar el siguiente código:

```
#include <stdio.h>

__global__ void helloFromGPU(void) {
    printf("Hello World from GPU!\n");
}
int main(void) {
    printf("Hello World from CPU!\n");
    helloFromGPU<<<1, 10>>>();
    cudaDeviceReset();
    return 0;
}
```

No es necesario asustarse, si bien esto se parece a un código de C al que todos estamos acostumbrados, tiene cosas que podrían parecerles atemorizantes al inicio, pero no se preocupen, todo tiene una explicación. Lo primero que se debe notar es que el código tiene una función llamada `helloFromGPU` que tiene un prefijo `__global__`. Este prefijo le indica al compilador que esta función se ejecutará en la GPU. Luego en la función `main` se puede ver que se llama a la función `helloFromGPU` con el operador `<<<1, 10>>>`. Este operador le indica al compilador que la función `helloFromGPU` se debe ejecutar en la GPU. El primer número le indica al compilador cuantos bloques de hilos se deben crear y el segundo número le indica al compilador cuantos hilos se deben crear por bloque. En este caso se está creando un bloque de 10 hilos. Esto quiere decir que se crearán 10 hilos que se ejecutarán en paralelo en la GPU. Esto quiere decir que se ejecutarán 10 veces la función `helloFromGPU` en paralelo. Esto quiere decir que se imprimirá 10 veces el mensaje `Hello World from GPU!`. Luego de esto se llama a la función `cudaDeviceReset` que le indica a la GPU que se debe reiniciar. Esto es necesario para que se puedan ejecutar más programas en la GPU. Finalmente, se retorna 0 para indicar que el programa se ejecutó correctamente.

Explicando *_global_* y <<<1, 10>>>

Los cambios más importantes que se deben hacer en un programa de C para que se pueda ejecutar en la GPU ya los vimos en el código anterior, el primero siendo esa función `__global__`. Esto no es más que una función a la cual se le agrega ese prefijo para indicarle al compilador que esa función se debe ejecutar en la GPU. El segundo cambio importante es el operador `<<<1, 10>>>`. Este operador le indica al compilador que la función `helloFromGPU` se debe ejecutar en la GPU. El primer número le indica al compilador cuantos bloques de hilos se deben crear y el segundo número le indica al compilador cuantos hilos se deben crear por bloque. En este caso se está creando un bloque de 10 hilos. Esto quiere decir que se crearán 10 hilos que se ejecutarán en paralelo en la GPU. Esto quiere decir que se ejecutarán 10 veces la función `helloFromGPU` en paralelo. Esto quiere decir que se imprimirá 10 veces el mensaje `Hello World from GPU!`. Luego de esto se llama a la función `cudaDeviceReset` que le indica a la GPU que se debe reiniciar. Esto es necesario para que se puedan ejecutar más programas en la GPU. Finalmente, se retorna 0 para indicar que el programa se ejecutó correctamente.

Pueden ver que el código es muy similar al código de C que todos conocemos. solo tomando en cuenta que se pueden dividir los problemas y ejecutarlos en paralelo. Esto es lo que hace que CUDA sea tan poderoso. Pero no se preocupen, esto no es todo lo que se puede hacer con CUDA. A continuación veremos como se puede hacer un programa que sume dos vectores.

Sumando dos vectores en CPU

Para sumar dos vectores en CPU se puede usar el siguiente código:

```
#include <stdio.h>
#include <stdlib.h>

void sumVectors(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}

int main(){
```

```
    int n = 100000;
    int *a, *b, *c;
    a = (int*)malloc(n * sizeof(int));
    b = (int*)malloc(n * sizeof(int));
    c = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        a[i] = i;
        b[i] = i;
    }
    sumVectors(a, b, c, n);
    for (int i = 0; i < n; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }
    free(a);
    free(b);
    free(c);
    return 0;
}
```

Este programa si bien es tan simple que no es necesario explicarlo, es importante notar que se está usando un ciclo `for` para sumar los vectores. Esto quiere decir que se está sumando un elemento de cada vector en cada iteración del ciclo `for`. Esta tarea además no es tan compleja como para que se necesite ejecutar en paralelo. Pero si se quisiera sumar dos vectores de 1000000000 elementos, esto podría tomar mucho tiempo. Por lo que sería bueno poder ejecutar esta tarea en paralelo. A continuación veremos como se puede hacer esto.

Sumando dos vectores en GPU

Para sumar dos vectores en GPU se puede usar el siguiente código:

```
#include <stdio.h>
#include <stdlib.h>

__global__ void sumVectors(int *a, int *b, int *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}

int main(void) {
    int n = 10000;
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    size_t size = n * sizeof(int);

    a = (int*)malloc(size);
    b = (int*)malloc(size);
    c = (int*)malloc(size);

    for (int i = 0; i < n; i++) {
```

```
        a[i] = i;
        b[i] = i;
    }

    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    dim3 blockSize(256);  // Number of threads per block (adjust according to the
GPU capability)
    dim3 gridSize((n + blockSize.x - 1) / blockSize.x);  // Number of blocks in
the grid

    sumVectors<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    for (int i = 0; i < n; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    free(a);
    free(b);
    free(c);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Dado que esto es diferente al código visto en CPU vamos a explicarlo parte por parte:

Esto no lo voy a explicar(En caso de ser necesario pueden preguntar en la clase)

```
#include <stdio.h>
#include <stdlib.h>
```

```
__global__ void sumVectors(int *a, int *b, int *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
```

Esta parte del código se llama kernel. Un kernel es una función que se ejecuta en la GPU. En este caso el kernel recibe tres vectores y el tamaño de los vectores. Además, recibe dos variables que se usan para determinar el índice del vector que se va a sumar. El índice se determina con la siguiente fórmula:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

En esta seccion, se está calculando el índice del vector que se va a sumar. Este índice se calcula con la fórmula `blockIdx.x * blockDim.x + threadIdx.x`. Él `blockIdx.x` es el índice del bloque que se está ejecutando. El `blockDim.x` es el tamaño del bloque. Él `threadIdx.x` es el índice del thread que se está ejecutando. Por lo que el índice del vector que se va a sumar es el índice del bloque multiplicado por el tamaño del bloque más el índice del thread. Esto quiere decir que cada thread va a sumar un elemento de cada vector. Esto es lo que hace que se pueda ejecutar en paralelo.

```
int main(void) {
    int n = 10000;
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    size_t size = n * sizeof(int);
```

Ahora, iniciamos la función principal del programa. En esta parte del código se están declarando los vectores que se van a sumar. Además, se están declarando los punteros que se van a utilizar para copiar los vectores de la CPU a la GPU y viceversa. Además, se está declarando la variable `size` que se va a utilizar para determinar el tamaño de los vectores.

```
    a = (int*)malloc(size);
    b = (int*)malloc(size);
    c = (int*)malloc(size);
```

Aquí se están asignando los vectores en la memoria de la CPU.

```
    for (int i = 0; i < n; i++) {
        a[i] = i;
        b[i] = i;
    }
```

Aquí llenamos los vectores con valores para poder sumarlos.

```
    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);
```

Se asigna la memoria en la GPU para los vectores.

```
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

Se copian los vectores de la CPU a la GPU.

```
    dim3 blockSize(256);  // Number of threads per block (adjust according to the
GPU capability)
    dim3 gridSize((n + blockSize.x - 1) / blockSize.x);  // Number of blocks in
the grid
```

En esta sección del código, lo que se está haciendo es determinar el tamaño de los bloques y de la grilla. Esto se hace para poder ejecutar el kernel en la GPU. En este caso se está usando un bloque de 256 threads. Esto quiere decir que cada bloque va a tener 256 threads. La grilla se está calculando para que tenga el número de bloques necesarios para sumar todos los elementos de los vectores. Esto se hace con la siguiente fórmula:

```
(n + blockSize.x - 1) / blockSize.x
```

```
    sumVectors<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
```

En esta parte del código se está ejecutando el kernel en la GPU. Se le está pasando la grilla, el tamaño del bloque y los vectores que se van a sumar.

```
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

Aquí se copia el resultado de la suma de la GPU a la CPU.

```
    for (int i = 0; i < n; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }
```

Por último se imprime el resultado de la suma.

```
    free(a);
    free(b);
    free(c);
    cudaFree(d_a);
```

```
    cudaFree(d_b);
    cudaFree(d_c);
```

Se libera la memoria de la CPU y de la GPU.

```
    return 0;
}
```

Se retorna 0 para indicar que el programa se ejecutó correctamente.

Ejemplo

Crear un programa que multiplique dos matrices de 1000x1000. El programa debe imprimir el tiempo que tarda en ejecutarse.

El proceso de multiplicacion de matrices es el siguiente:

Multiplicación de Matrices:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{np} \end{bmatrix}$$

Multiplicación de Matrices:

$$A \times B = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + \cdots + a_{1n}b_{n1} & \dots & a_{11}b_{1p} + a_{12}b_{2p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + a_{22}b_{21} + \cdots + a_{2n}b_{n1} & & \end{bmatrix}$$

Cuando hacemos el programa en CPU, el código se ve así:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Declaración de las matrices A y B
int A[1000][1000];
int B[1000][1000];

// Declaración de la matriz C, que será el resultado de la multiplicación de A y B
```

```
int C[1000][1000];

// Función que calcula la multiplicación de dos matrices
void multiplicar_matrices(int A[1000][1000], int B[1000][1000], int C[1000][1000],
int n) {
  int i, j, k;

  // Recorrer cada celda de la matriz C
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      // Calcular el valor de la celda de C
      C[i][j] = 0;
      for (k = 0; k < n; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}

// Función principal
int main() {
  // Inicializar las matrices A y B
  int i, j;
  for (i = 0; i < 1000; i++) {
    for (j = 0; j < 1000; j++) {
      A[i][j] = rand() % 100;
      B[i][j] = rand() % 100;
    }
  }

  // Inicializar la matriz C
  for (i = 0; i < 1000; i++) {
    for (j = 0; j < 1000; j++) {
      C[i][j] = 0;
    }
  }

  // Obtener el tiempo actual
  clock_t inicio = clock();

  // Multiplicar las matrices A y B
  multiplicar_matrices(A, B, C, 1000);

  // Obtener el tiempo final
  clock_t fin = clock();

  // Imprimir el tiempo que tardó la multiplicación
  double tiempo = (double)(fin - inicio) / CLOCKS_PER_SEC;
  printf("El tiempo que tardó la multiplicación fue de %f segundos\n", tiempo);

  // Regresar 0
  return 0;
}
```

Cuando hacemos el programa en GPU, el código se ve así:

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <time.h>

// Declaración de las matrices A y B
int *A, *B, *C;

// Función que calcula la multiplicación de dos matrices
__global__ void multiplicar_matrices_kernel(int *A, int *B, int *C, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;

  // Calcular el valor de la celda de C
  C[i * n + j] = 0;
  for (int k = 0; k < n; k++) {
    C[i * n + j] += A[i * n + k] * B[k * n + j];
  }
}

// Función principal
int main() {
  // Inicializar las matrices A y B
  int n = 1000;
  A = (int *)malloc(n * n * sizeof(int));
  B = (int *)malloc(n * n * sizeof(int));
  C = (int *)malloc(n * n * sizeof(int));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      A[i * n + j] = rand() % 100;
      B[i * n + j] = rand() % 100;
    }
  }

  // Inicializar la matriz C
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      C[i * n + j] = 0;
    }
  }

  // Alojar memoria en la GPU
  int *A_d, *B_d, *C_d;
  cudaMalloc(&A_d, n * n * sizeof(int));
  cudaMalloc(&B_d, n * n * sizeof(int));
  cudaMalloc(&C_d, n * n * sizeof(int));

  // Copiar las matrices A y B a la GPU
  cudaMemcpy(A_d, A, n * n * sizeof(int), cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(B_d, B, n * n * sizeof(int), cudaMemcpyHostToDevice);

    // Lanzar el kernel
    dim3 gridDim(n / 32, n / 32);
    dim3 blockDim(32, 32);

    // Iniciar el evento para medir el tiempo
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);

    multiplicar_matrices_kernel<<<gridDim, blockDim>>>(A_d, B_d, C_d, n);

    // Finalizar el evento para medir el tiempo
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    // Copiar el resultado de la GPU a la CPU
    cudaMemcpy(C, C_d, n * n * sizeof(int), cudaMemcpyDeviceToHost);

    // Liberar la memoria en la GPU
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);

    // Liberar la memoria en la CPU
    free(A);
    free(B);
    // Note: We are not freeing C here because it holds the result.

    // Imprimir los resultados
    printf("El tiempo que tardó la multiplicación fue de %f segundos\n",
milliseconds / 1000.0);

    // Regresar 0
    return 0;
}
```

Cuando ejecutamos el programa en CPU, obtenemos el siguiente resultado:

```
El tiempo que tardó la multiplicación fue de 7.170000 segundos
```

Cuando ejecutamos el programa en GPU, obtenemos el siguiente resultado:

```
El tiempo que tardó la multiplicación fue de 0.147166 segundos
```

Aquí podemos ver claramente que la multiplicación de matrices en GPU es mucho más rápida que en CPU. Esto se debe a que la GPU tiene muchos más núcleos que la CPU, y, por lo tanto, puede realizar muchas más operaciones en paralelo. Si bien no todos los problemas se pueden resolver en paralelo, los que sí se pueden resolver en paralelo se pueden resolver mucho más rápido en GPU que en CPU.