PRACTICA 5

JOSE IGNACIO MORENO Y
DIEGO GONZALEZ

INDICE

Introducción

La computación acelerada por GPU es el uso de una unidad de procesamiento de gráficos junto a una CPU para acelerar el funcionamiento de las aplicaciones de aprendizaje profundo, análisis e ingeniería. También, podemos decir que utiliza el procesamiento paralelo para acelerar el trabajo en aplicaciones exigentes, desde IA y análisis hasta simulaciones y visualizaciones.

Nacida en el PC, ahora también existe en nuestros móviles y en cloud. E incluso todas las empresas la están adoptando para transformar sus negocios de datos.

Las computadoras aceleradas combinan las CPU y otros tipos de procesadores como iguales en una arquitectura a veces llamada computación heterogénea.

Para 2006, NVIDIA lideraba el campo de los proveedores de gráficos.

Algunos investigadores ya estaban desarrollando su propio código para aplicar el poder de las GPU a tareas fuera del alcance de las CPU.

En 2006, lideró el lanzamiento de CUDA, un modelo de programación para aprovechar los motores de procesamiento paralelo dentro de la GPU para cualquier tarea. Aparte, CUDA impulsó una nueva línea de las GPU de NVIDIA que llevó la computación acelerada a una gama cada vez mayor de aplicaciones industriales y científicas.

Objetivos:

Tener unos conocimientos claros de modelo de computación por GPU denominado CUDA para las arquitecturas de NVIDIA. Además, saber el uso y el desarrollo que se ha llevado a cabo sobre GPU computing, Workflow de aplicaciones GPU computing, Compilación, ejecución y profiling.

Ejercicio 1:

Primero nos creamos 3 variables float con nombres “a”, “b” y “out” que a continuación deberemos inicializarlo, y otras variables float con los nombres “d_a”, “d_b”, “d_out” que serán variables para el device.

Después declaramos y asignamos la memoria para host que lo haremos con la función malloc que sirve para solicitar un bloque de memoria del tamaño suministrado como parámetro.

Después de declarar y asignar la memoria del host inicializamos con un dato (el que queramos) las variables “a” y “b”

Después declaramos y asignamos la memoria para device que lo haremos con la función malloc que sirve para solicitar un bloque de memoria del tamaño suministrado como parámetro. Aloca N*sizeof(float) bytes en memoria global del device los cuales serán apuntados por d_a (d_b y d_out respectivamente).

La función vector_add, lo que hace es obtener cuatro valores. En la primera operación lo que hacemos es la siguiente operación: Numero de bloque * dimensión de bloque + hilo dentro del bloque y nos dará como resultado el índice al elemento del vector o dicho de otra manera la posición donde vas a añadir la información al vector out. Después hacemos la suma del dato a y el dato b y lo almacenamos en out, ya que es el resultado de ambas. El if, lo que hace si no se necesitan

todos los thread del último bloque, si el índice el thread que se va ejecutando es mayor o igual a n, no se necesitan loa thread ni los posteriores.

A continuación, transferimos los datos del host al device, con la siguiente función "cudaMemcpy", es decir pasamos la información de "a" o "b" o "out" a "d_a", "d_b", "d_out"

Ahora ejecutamos uno o más kernels. Primero calculamos el número de bloques que vamos a usar y el cálculo es la variable "N" entre 256, en la siguiente línea hacemos una comprobación de si el resto de N /256 no es 0 que le sume un bloque más. Y después debemos poner esta línea "vector_add<<<bloques,256>>>(d_out, d_a, d_b, N). Lo que significa es que llamamos al vector_add con numero de bloques que es la variable "bloques" que hemos hecho el cálculo anteriormente y el número de threads que se lanzan, que en nuestro caso es 256. Los demás parámetros es el "d_out" donde almacenara los datos de la función, el "d_a" y "d_b" donde esta los datos que anteriormente los hemos transferidos y "N" es la variable definida arriba del todo.

A continuación, transferimos datos desde el device al host, es decir, de la variable "d_out" a "out".

Para saber que no hay ningún fallo tenemos que hacer una verificación.

Por último, liberamos los recursos. Para hacerlo designamos memoria del device y del host.

Conclusión:

En esta práctica hemos analizado el modelo de computación por GPU para la arquitectura de NVIDIA. Además de saber conceptos de comunicación colectivo entre procesos de MPI. También hemos tenido que informarnos sobre que es la GPU computing, Workflow de aplicaciones GPU computing, la compilación, ejecución y profiling. Por último, hemos profundizado en analizar las diferencias entre la CPU y la GPU.

Bibliografía

https://la.blogs.nvidia.com/2021/11/19/que-es-la-computacion-acelerada/

https://developer.nvidia.com/cuda-downloads.

https://developer.nvidia.com/cuda-gpus

https://campusvirtual.nebrija.es/webapps/blackboard/execute/content/file?cmd=view&content_id=_2068477_1&course_id=_43028_1

http://sopa.dis.ulpgc.es/fso/cpp/intro_c/introc75.htm

https://fisica.cab.cnea.gov.ar/gpgpu/images/2017/clase_1_cuda.pdf

https://davidlopez.es/suma-concurrente-de-vectores-con-nvidia-cuda/