Programación Paralela

Teórica 02 - Introducción a CUDA

Segundo Semestre, 2025

2.1 Introducción

Como dijimos en la práctica anterior, la computación paralela es una técnica que permite realizar cálculos de manera simultánea en múltiples procesadores. Para los ejercicios vamos a utilizar CUDA C que nos permite escribir programas paralelos escalables en sistemas donde conviven tanto CPUs como GPUs. **En este tipo de sistemas donde hay porciones de código que pueden ejecutarse en paralelo, pero que están gobernadas por un código secuencial que corre en la CPU, los llamaremos *sistemas heterogéneos***. CUDA extiende el lenguaje de programación C con una sintaxis mínima que permite tener código secuencial, gobernado por la CPU, y código que puede ser ejecutado en paralelo en GPUs.

Cuando en el software moderno las aplicaciones se ejecutan *lento*, el problema usualmente suele ser que hay demasiados datos para ser procesados. Por ejemplo en el procesamiento de imágenes o videos, la simulación de dinámica de fluidos, el manejo de sistemas complejos como (líneas aéreas), o incluso cosas mucho más sencillas como convertir una imagen de pixels a escala de grises. Estas tareas se pueden fraccionar en tareas más pequeñas que pueden ser ejecutadas de manera independiente y paralela.

Paralelismo de tareas vs. paralelismo de datos

El paralelismo de tareas, se refiere a la ejecución de múltiples tareas (no necesariamente las mismas) al mismo tiempo. Mientras que el paralelismo de datos, se refiere a la ejecución de la misma tarea con datos diferentes al mismo tiempo.

2.1.1 Estructura de un programa en CUDA C

La estructura de un programa en CUDA C es similar a un programa en C, lo que refleja su naturaleza heterogénea donde existe un *host* (CPU) y uno o más *devices* (GPUs) en la computadora. El código fuente CUDA tiene una mezcla de ambos códigos, uno que se ejecuta en el host, y otro en los devices. Por defecto, todo el código se ejecuta en el host, aunque vamos a declarar funciones de una forma especial para que puedan ser corridas en los *devices*.

El código con estas extensiones de CUDA tiene que ser compilado con el compilador de NVIDIA, `nvcc`, que es un wrapper para el compilador de C, `gcc`. El compilador de NVIDIA se encarga de: separar el código que se ejecuta en el host, separar el código que se ejecuta en el device, y de compilarlo con el compilador de C. **El código identificado con las *keywords* (palabras reservadas) de CUDA para las funciones paralelas se denominan *kernels***. Estos kernels son funciones que están asociadas a estructuras de datos y que van a ser ejecutadas en paralelo por GPUs. En las situaciones donde no haya una GPU disponible, el código de todas formas se ejecutará en una CPU (uno podría incluso ejecuar el kernel en una CPU utilizando herramientas como MCUDA) [1].

La ejecuciónm de un programa en CUDA se ilustra en la Figura 1, la ejecución comienza con el código del host (CPU) y cuando se llama a una función kernel (código paralelo del dispositivo) es ejecutada por un gran número de threads en un dispositivo. Todos los threads que son generados por un kernel son colectivamente llamados un *grid*. Estos threads son el

vehículo principal de la ejecución paralela en una plataforma CUDA. Cuando todos los threads de un kernel completan su ejecución, el grid correspondiente termina, y la ejecución continúa en el host hasta que otro kernel es lanzado. Notar que la Figura 1 muestra un modelo simplificado donde la ejecución del código de la CPU y la GPU no se superponen, pero en muchas de las aplicaciones heterogéneas, la CPU y las GPUs para aprovechar al máximo ambos recursos.

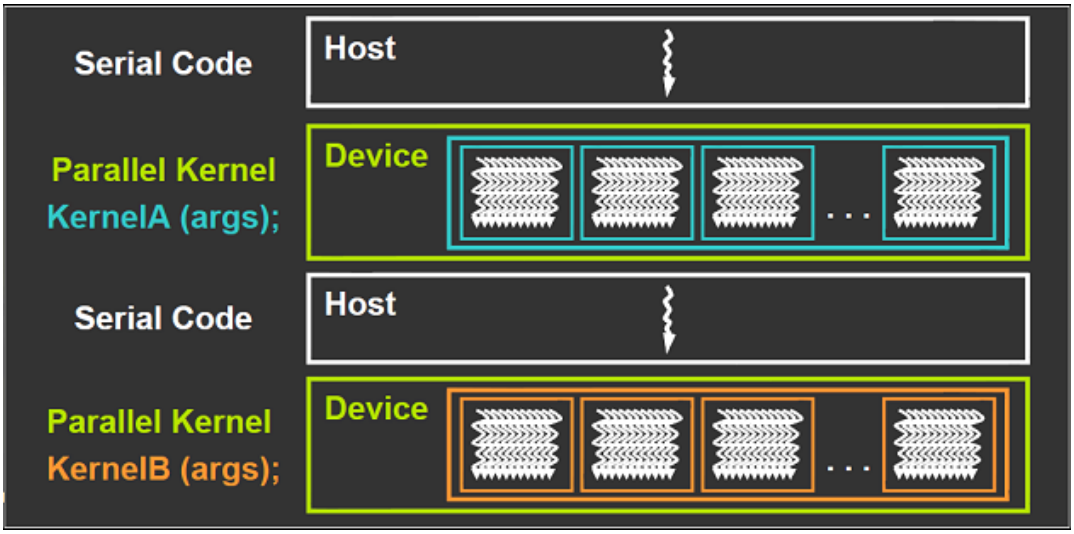

Figure 1: Ejecución de un programa en CUDA

2.2 Sumar dos vectores

2.2.1 Suma secuencial

Una vez que las funciones para los kernels están definidas en un código fuente, ya no se puede utilizar el compilador de C tradicional y hay que utilizar un compilador que entienda estas declaraciones adicionales como es el caso de `nvcc`.

Como primera aproximación utilizaremos un ejemplo muy sencillo que consiste en sumar las componentes de dos vectores que es una de las operaciones más sencillas que podemos realizar en computación paralela. Antes de adentrarnos en el código del *kernel* para la suma de dos vectores, es útil revisar cómo funciona una función de suma de vectores con un algoritmo secuencial.

Algorithm 1 Suma de dos vectores

Define: vecAdd(a, b)
Input: $a = \{a_0, a_1, a_2, \ldots, a_n\}$, $b = \{b_0, b_1, b_2, \ldots, b_n\}$
Initialization: $c_0 = 0, c_1 = 0, c_2 = 0, \ldots, c_n = 0$
Output: $c_0 = a_0 + b_0, c_1 = a_1 + b_1, c_2 = a_2 + b_2, \ldots, c_n = a_n + b_n$
1: **for** $i = 0$ to n **do**
2: $\quad c_i = a_i + b_i$

El mismo algoritmo en C sería algo así:

```
1  void vecAdd(float* h_A, float* h_B, float* h_C, int n)
2  {
3    for (int i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
4  }
5  int main() {
6    // Reservamos memoria para h_A, h_B, and h_C // I/O para leer h_A and h_B
   (los N elementos cada uno)
7    vecAdd(h_A, h_B, h_C, N);
8  }
```

El código muestra una función que suma dos vectores. Se van a utilizar los prefijos `h_` para las variables que son procesadas por el host y `d_` para las variables que son procesadas por el device. En este caso, todas las variables son procesadas por el host, por lo que sólo se ven las variables `h_`.

2.2.2 Suma paralela

Una forma sencilla de ejecutar la suma de dos vectores en paralelo es modificar la función `vecAdd` y mover sus cálculos a un dispositivo. La forma de hacer esto es:

Algorithm 2 Suma de dos vectores

Define: vecAdd(a, b)
Input: $a = \{a_0, a_1, a_2, \ldots, a_n\}$, $b = \{b_0, b_1, b_2, \ldots, b_n\}$
Initialization: $c_0 = 0, c_1 = 0, c_2 = 0, \ldots, c_n = 0$
Output: $c_0 = a_0 + b_0, c_1 = a_1 + b_1, c_2 = a_2 + b_2, \ldots, c_n = a_n + b_n$

1: // Reservar espacio en la memoria del dispositivo para almacenar copias de los vectores A, B y C.
reserveMemory d_A
reserveMemory d_B
reserveMemory d_C
2: // Copiar los vectores desde la memoria del host a la memoria del dispositivo.
$d_A = a$
$d_B = b$
3: // Lanzar la ejecución paralela del kernel de suma de vectores en el dispositivo.
launchKernel vecAdd(d_A, d_B, d_C)
4: // Copiar el vector suma C desde la memoria del dispositivo a la memoria del host.
$c = d_C$
5: // Liberar los vectores en la memoria del dispositivo.
freeMemory d_A
freeMemory d_B
freeMemory d_C

En los sistemas CUDA actuales, los dispositivos son a menudo tarjetas de hardware que vienen con su propia memoria dinámica de acceso aleatorio (DRAM). Por ejemplo, la NVIDIA GTX1080 viene con hasta 8 GB de DRAM, llamada memoria global. Usaremos los términos memoria global y memoria del dispositivo de manera intercambiable. Para ejecutar un kernel en un dispositivo, el programador necesita asignar memoria global en el dispositivo y transferir datos pertinentes desde la memoria del host a la memoria del dispositivo asignada. Del mismo

modo, después de que el kernel haya terminado de ejecutarse, el programador necesita transferir los resultados desde la memoria del dispositivo a la memoria del host y liberar la memoria global asignada en el dispositivo.

La figura **??**, muestra una imagen de alto nivel del modelo de memoria del host y dispositivo CUDA para que los programadores razonen sobre la asignación de memoria del dispositivo y el movimiento de datos entre el host y el dispositivo. La memoria global del dispositivo puede ser accedida por el host para transferir datos hacia y desde el dispositivo, como se ilustra en las flechas bidireccionales entre estas memorias y el host. La memoria constante puede ser accedida de manera de solo lectura por funciones de dispositivo.

Por ahora sólo importa saber que NVIDIA creó dos funciones nuevas, `cudaMalloc` y `cudaFree`. La función `cudaMalloc` puede ser llamada desde el código del host para asignar un pedazo de memoria global del dispositivo para un objeto. El primer parámetro de la función `cudaMalloc` es la dirección de un puntero que será configurado para apuntar al objeto asignado. La dirección del puntero debe ser convertida a `(void **)` porque la función espera un puntero genérico; la función de asignación de memoria es una función genérica que no está restringida a ningún tipo particular de objetos. Este parámetro permite a la función `cudaMalloc` escribir la dirección de la memoria asignada en el puntero. El segundo parámetro de la función `cudaMalloc` da el tamaño de los datos a asignar, en número de bytes. El uso de este segundo parámetro es consistente con el parámetro de tamaño de la función `malloc` de C.

Por el otro lado la función `cudaFree` puede ser llamada desde el código del host para liberar la memoria global del dispositivo asignada a un objeto. La función `cudaFree` toma un puntero a la memoria global del dispositivo que se va a liberar. La función `cudaFree` no necesita cambiar el contenido del puntero, solo necesita usar el valor del puntero para devolver la memoria asignada al pool disponible.

2.3 Convertir una Imagen a Escala de Grises

El procesamiento de imágenes es un clásico ejemplo de computación paralela y comenzaremos la introducción a CUDA viendo un ejemplo de procesamiento de imágenes para ejemplificar los primeros conceptos. En este caso vamos a ver cómo convertir una imagen a escala de grises.

Threads

La ejecución de un kernel genera un gran número de threads para explotar el paralelismo de datos. En el caso de la conversión de una imagen color a escala de grises cada thread podría ser utilizado para computar un pixel de la imagen de salida. En este caso, el número de threads que serán generados por el kernel es igual al número de pixels en la imagen. Para imágenes grandes, un gran número de threads serán generados. En la práctica, cada thread puede procesar múltiples pixels para eficiencia. Los programadores de CUDA pueden asumir que estos threads toman muy pocos ciclos de reloj para ser generados y programados debido al soporte eficiente del hardware. Esto es en contraste con los threads tradicionales de la CPU que típicamente toman miles de ciclos de reloj para ser generados y programados.

2.3.1 Representación de una imagen en la computadora

Antes de comenzar a escribir cualquier tipo de código, tenemos que saber cómo se representa una imagen en la computadora. Esencialmente una imagen se representa como una matriz de

tuplas (R, G, B) donde R, G y B son los valores de los colores (canales) rojo, verde y azul respectivamente. Cada uno de estos canales tiene un valor que va desde 0 ($0x00$) a 255 ($0xFF$) y representan la intensidad de cada color de un pixel en una imagen. Existen 256^3 colores posibles y estos colores se representan dentro del triángulo AdobeRGB (ver Figura 2).

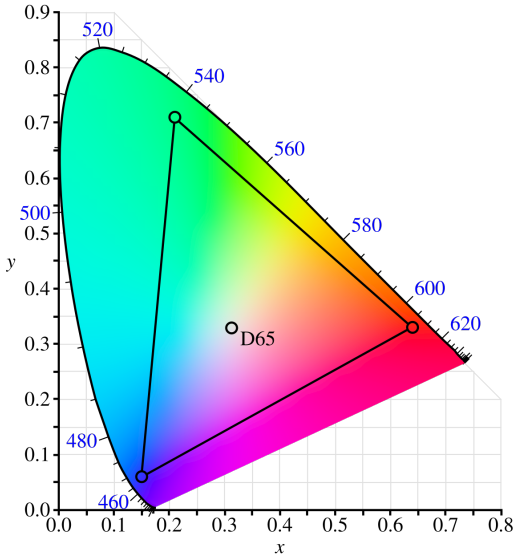

Figure 2: Triángulo AdobeRGB

Como dijimos, estos valores de R, G y B representan los *canales* y para convertir estos canales a escala de grises se debe utilizar una fórmula, ya que hay que decidir cuál va a ser la intensidad final del pixel en escala de grises en la imagen blanco y negro. Esto se hace realizando una combinación lineal de los valores de los canales de color de alguna manera y, lógicamente hay muchas formas de convertir estos canales a escala de grises. Por ejemplo podríamos tomar sólo el valor del canal G (verde), haciendo la combinación lineal $pixel_{gris} = R \cdot 0 + G \cdot 1 + B \cdot 0$. De esta manera la imagen sólo se vería representada por ese canal, lo cual no sería muy verídico, ya que el ojo humano percibe la intensidad de los 3 canales. Como podrán imaginar, hay muchas formas de convertir una imagen a escala de grises [1], pero hay algunas que son más comunes que otras.

Algunas fórmulas comunes para convertir a escala de grises, son las siguientes:

- **Promedio**: $I = \frac{R+G+B}{3}$
- **Luminosidad**: $I = 0.21R + 0.72G + 0.07B$
- **Desaturación**: $I = \frac{max(R,G,B)+min(R,G,B)}{2}$

Para nuestro ejemplo, vamos a usar la fórmula de *luminosidad* para convertir la imagen a escala de grises que es un promedio pesado de los canales de color, y que representa la percepción humana de la luminosidad, pero como dijimos no es la única forma y cambiar esta fórmula puede dar diferentes resultados que podrían considerarse como "filtros de imagen".

[1] ¡Podrían probar con diferentes fórmulas y ver cómo se ve la imagen!, hay 256^3 combinaciones posibles.

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry.," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005, ISSN: 1542-7730. DOI: 10.1145/1095408.1095421. [Online]. Available: https://doi.org/10.1145/1095408.1095421.