Introducción a la Programación Paralela (CUDA)

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].

Kernel

Los kernels son funciones que están asociadas a estructuras de datos que se van a ejecutar en paralelo por GPUs.

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.

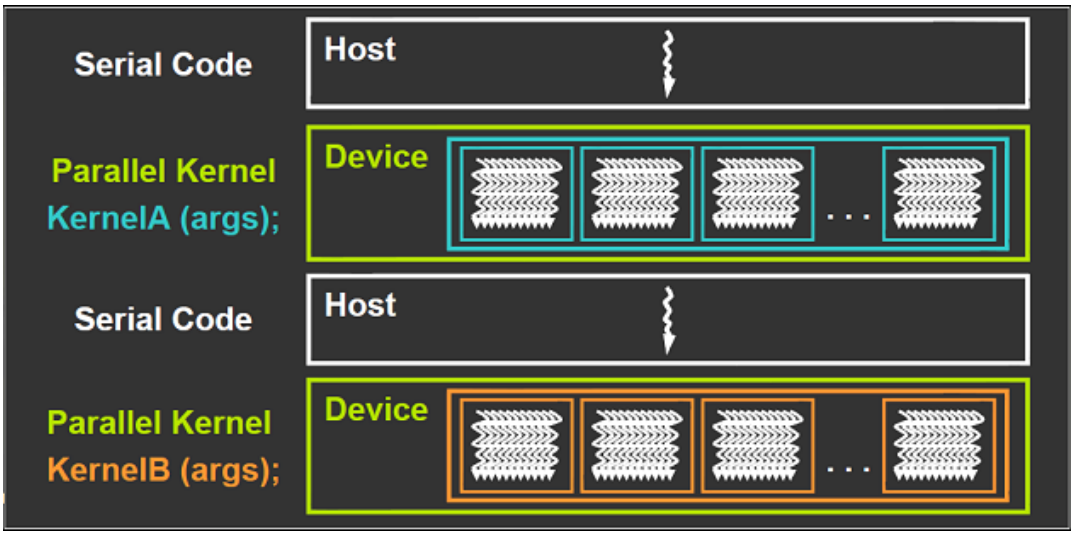

Figure 1: Ejecución de un programa en CUDA

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: $\quad c_i = a_i + b_i$

La implementación del algoritmo en C, sería la siguiente:

```
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    int n = 100; // Numero de elementos en el vector
7
8    // Reservamos memoria para h_A, h_B, and h_C
9    int* h_A = (int *)malloc(n * sizeof(int));
10   int* h_B = (int *)malloc(n * sizeof(int));
11   int* h_C = (int *)malloc(n * sizeof(int));
12
13   // inicializamos h_C en 0
14   for (int i = 0; i < n; i++) h_C[i] = 0;
15
16   // I/O para leer h_A and h_B   (los N elementos cada uno)
17   // ...
18
19   // Llamamos a la funcion de suma
20   vecAdd(h_A, h_B, h_C, N);
21
22   return 0;
23 }
```

El algoritmo presentado y su correspondiente código en C, son ejemplos de un algoritmo secuencial para la suma de dos vectores de números almacenados en memoria. En este caso este código se ejecuta en la CPU de manera secuencial; para diferenciar las variables que son procesadas por el host y las que son procesadas por el device, vamos 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, como todo el código se ejecuta en la CPU del host, 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:

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

Algorithm 2 Suma de dos vectores en paralelo

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: $c_0 = a_0 + b_0, c_1 = a_1 + b_1, c_2 = a_2 + b_2, \ldots, c_n = a_n + b_n$
// 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
// Copiar los vectores desde la memoria del host a la memoria del dispositivo.
$d_A = a$
$d_B = b$
// Lanzar la ejecución paralela del kernel de suma de vectores en el dispositivo.
launchKernel vecAdd(d_A, d_B, d_C)
// Copiar el vector suma C desde la memoria del dispositivo a la memoria del host.
$c = d_C$
// Liberar los vectores en la memoria del dispositivo.
freeMemory d_A
freeMemory d_B
freeMemory d_C

los resultados desde la memoria del dispositivo a la memoria del host y liberar la memoria global asignada en el dispositivo.

La figura 2, 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.

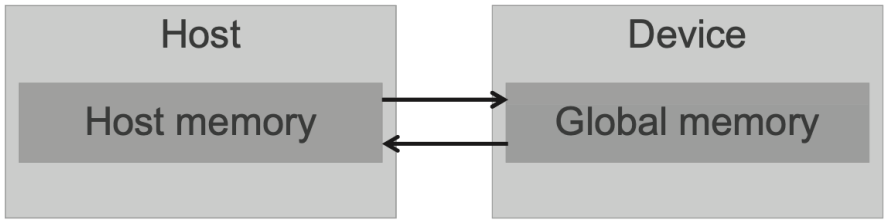

Figure 2: Modelo de memoria del host y dispositivo CUDA

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.

El código para crear la memoria en el device sería algo así:

```
1 float *d_A;
2 int n = 100; // Numero de elementos en el vector
3 cudaMalloc((void **)&d_A, n * sizeof(float));
4 ...
5 cudaFree(d_A);
```

Una vez que el host ha asignado memoria en el dispositivo, puede transferir datos desde la memoria del host a la memoria del dispositivo utilizando las funciones API de CUDA.

El código para asignar y copiar memoria del host al device y viceversa se da de la siguiente manera:

```
1  void vecAdd(float* h_A, float* h_B, float *h_C, int n) {
2    int size = n * sizeof(float);
3    float *d_A, *d_B, *d_C;
4    cudaMalloc((void **) &d_A, size);
5    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
6    cudaMalloc((void **) &d_B, size);
7    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
8
9    cudaMalloc((void **) &d_C, size);
10
11   // Invocacion al kernel... (veremos esto mas adelante)
12
13   cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
14   cudaFree(d_A);
15   cudaFree(d_B);
16   cudaFree(d_C);
17 }
```

Notar que `cudaMemcpyHostToDevice` y `cudaMemcpyDeviceToHost` son dos constantes que indican la dirección de la transferencia de memoria. La función `cudaMemcpy` es una función de la API de CUDA que copia memoria entre el host y el device. El primer parámetro de la función `cudaMemcpy` es la dirección de la memoria de destino, el segundo parámetro es la dirección de la memoria de origen, el tercer parámetro es el tamaño de la memoria a copiar en bytes, y el cuarto parámetro es una constante que indica la dirección de la transferencia de memoria.

El manejo de errores es una parte fundamental de cualquier software que desarrollemos es muy importante manejar los errores y CUDA no es la excepción. Por ejemplo cuando en el código anterior se llama a `cudaMalloc` se deebría verificar si la asignación de memoria fue exitosa, para ello se puede hacer algo así:

```
1 cudaError_t err = cudaMalloc((void **) &d_A, size);
2 if (err != cudaSuccess) {
3   printf("%s in %s at line %d\n", cudaGetErrorString(err), __FILE__,
        __LINE__);
```

```
4     exit(EXIT_FAILURE);
5 }
```

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.

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 3).

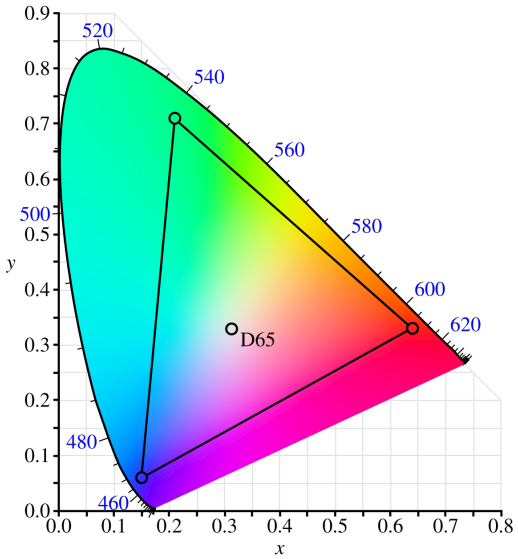

Figure 3: Triángulo AdobeRGB

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**: $I = \frac{max(R,G,B)+min(R,G,B)}{2}$

Es común que se utilice la *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".

Este ejemplo es un buen caso de una aplicación no trivial de algo que se puede hacer en paralelo, ya que cada pixel de la imagen es independiente de los demás, por lo que cada thread puede calcular el valor de luminosidad de un pixel sin necesidad de esperar a que terminen los demás.

2.4 Funciones Kernel y Threads

Estamos en ccondiciones de hablar más sobre las funciones kernel de CUDA y el efecto de lanzar estas funciones. En CUDA, una función kernel especifica el código que será ejecutado por todos los threads durante una fase paralela.

Dado que todos los threads ejecutan el mismo código, la programación en CUDA es una instancia del conocido estilo de programación paralelo *Single-Program Multiple-Data* (SPMD) [Ata 1998], un estilo de programación para sistemas masívamente paralelos. Cuando el código del host lanza un kernel, el sistema de ejecución de CUDA genera una matriz de *threads* que están organizados en una jerarquía de dos niveles. En un nivel tenemos los *bloques de threads* (referidos normalmente como **bloques**), cada uno de estos bloques tendrá el mismo tamaño y contendrá hasta 1024 threads. Como se puede ver en la figura 4, cada bloque de threads tiene (en este caso de ejemplo), N bloques con 6 threads cada uno, y todos los threads van a ejecutar el mismo código.

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

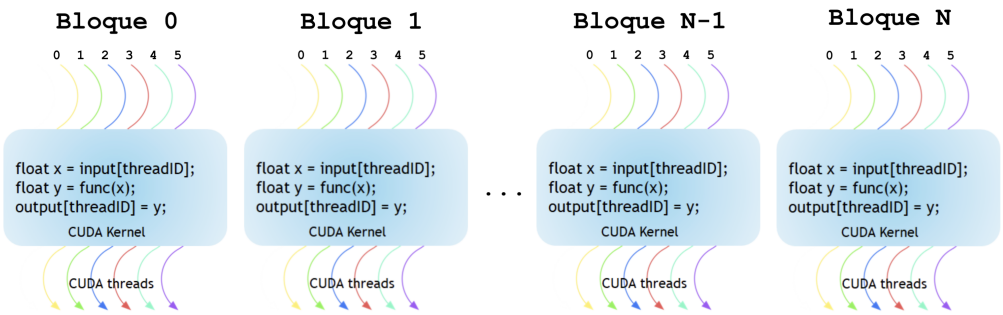

Figure 4: Bloques de threads

El número total de threads en cada bloque de threads se especifica en el código del host al lanzar el kernel; no es necesario que siempre se lance con el mismo número de threads cada vez. Para cada una de estas matrices el número de threads en un bloque se define en la variable `blockDim`.

Estos kernels de CUDA tienen acceso a dos variables llamadas `blockIdx` y `threadIdx` que son variables especiales que contienen el índice del bloque y el índice del thread respectivamente. Por ejemplo en la figura 4, `blockIdx` irá de 0 a N y `threadIdx` irá de 0 a 5. En base a esto se puede calcular un índice *idx* general para cada thread en el bloque, que se puede calcular de la siguiente manera:

$$idx = blockIdx.x * blockDim.x + threadIdx.x$$

En CUDA C hay tres calificadores que se pueden usar para definir el tipo de función que se va a ejecutar en el dispositivo. Estos son:

Table 1: *Palabras clave para definición de funciones*

keyword	**Ejecutado por**	**Llamado desde**
`__device__`	device	device
`__global__`	device	host
`__host__`	host	host

Aunque esta es la regla general, hay algunas excepciones a esto, por ejemplo los sistemas CUDA que soportan *paralelismo dinámico*.

La palabra clave `__global__` indica que la función está declarada como una función de kernel, estas funciones se ejecutan en el device y sólo pueden ser llamadas desde el host. La palabra calve `__device__` indica que la función está declarada como una función de *device* y puede ser llamada desde el device, mientras que la palabra clave `__host__` indica que la función está declarada como una típica función tradicional de C que se ejecuta en el host y puede ser llamada desde cualquier otra función del host, por default todas las funciones en un programa en CUDA son funciones de host.

Vale señalar que estas palabras clave pueden combinarse en una misma declaración, por ejemplo si se añade `__host__` y `__device__` a una función, el compilador de CUDA generará dos funciones separadas, una para el host y otra para el device.

Veamos un ejemplo de cómo se vería el kernel de la suma de dos float en CUDA C:

```
__global__ void vecAdd(float* d_A, float* d_B, float* d_C, int n) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < n) {
    d_C[idx] = d_A[idx] + d_B[idx];
  }
}
```

En este código podemos observar algunas particularidades inmediatas. La primera es que existen variables *built-in*[2] que son `blockIdx`, `blockDim` y `threadIdx`. Estas variables son para poder hacer algunas diferenciaciones entre el código que se ejecuta en el kernel para poder direccionar los datos adecuadamente de acuerdo a quién esté ejecutando el código. Poruqe lógicamente cada thread va a estar ejecutando el mismo código sobre diferentes datos, entonces cada código debe saber sobre qué datos está trabajando y dónde debe colocar el resultado.

Otra particularidad es que estas variables poseen un sufijo `.x` que implica que habrá otros sufijos como `.y` y `.z` que se pueden usar para acceder a otras dimensiones de la variable.

En la kernel function está declarada la variable `idx` que es privada para cada thread, con lo cual si lanzamos 10.000 threads, habrá 10.000 variables `idx` que serán generadas y cada una de ellas tendrá un valor diferente para cada thread siendo sólo accesible para el thread que la generó.

Siguiendo con el código vemos que hay una guarda que dice `if (idx < n)`, esto es para evitar que el kernel acceda a memoria fuera de los límites del vector. Porque es muy probable que la cantidad de elementos del vector no sea un múltiplo de la cantidad de threads que se lanzan, y si no se pone esta condición el kernel podría intentar acceder a memoria que no existe y eso generaría un error de ejecución. Es **muy** importante tener en cuenta esto.

Ahora si comparamos este código con el algoritmo presentado en 1, podemos ver que no hay un *loop* en el código del kernel. Este *loop* ha sido reemplazado por una matriz de threads que se ejecutan en paralelo. **A este tipo de paralelismo se le denomina *loop parallelism*** donde las iteraciones secuenciales del código original se ejecutan por threads en paralelo.

Con lo cual nos falta ver cómo se ejecutaría este kernel desde el host, para ello podemos usar el siguiente código:

```
int vectAdd(float* d_A, float* d_B, float* d_C, int n) {
  // Aqui iria toda la parte de la asignacion de memoria y la copia de datos
  // ...

  vecAddKernel<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```

En este código se ve que se utiliza una sintaxis nueva `<<` y `>>` que es la forma de pasar cuáles son los parámetros de configuración del kernel. En este caso se está pasando el número de bloques y el número de threads por bloque. Por ejemplo imaginemos que tenemos un vector de 1000 elementos y queremos lanzar 256 threads por bloque, en este caso el número de bloques que se van a lanzar es 4 (ya que $1000/256 = 3.90625$), pero al sacar *ceil* el resultado será 4 (por esto es que se utiliza *float* para el resultado de la división). Matemáticamente esto se puede expresar como:

$$\text{bloques} = \left\lceil \frac{1000}{256} \right\rceil = 4$$

[2]Estas variables son variables que son generadas por el compilador y no están declaradas en el código explícitamente.

Habrá entonces 4 bloques de 256 threads cada uno para un total de 1024 threads, pero sólo se van a usar 1000, y habrá 24 threads que no van a hacer nada gracias a la guarda que pusimos en el kernel.

2.4.1 Ejecución del kernel

Ya estamos en condiciones de poner todos los elementos juntos y ver cómo se ejecutaría este código. La cantidad de bloques dependerá del tamaño del vector. Si el vector tiene menos de $n < 256$ elementos, entonces sólo se lanzará un bloque, mientras que si $n = 2.000.000$ entonces se lanzarán 7813 bloques. Donde cada uno de los threads va a ejecutar en diferentes posiciones del vector de entrada. Es importante señalar que **el paralelismo implica que NO podemos depender del orden de ejecución**, ya que no sabemos en qué orden se van a ejecutar los threads. **NO se pueden hacer suposiciones sobre el orden de ejecución de los threads**.

Este código es agnóstico de la GPU, es decir, no importa si la GPU tiene 1 o 1000 cores, el código va a funcionar de la misma manera, mientras que una GPU tenga soporte para CUDA.

El código entonces sería algo así:

```
1  void vecAdd(float* A, float* B, float* C, int n) {
2    int size = n * sizeof(float);
3    float *d_A, *d_B, *d_C;
4
5    cudaMalloc((void **) &d_A, size);
6    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
7    cudaMalloc((void **), &d_B, size);
8    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
9
10   cudaMalloc((void **), &d_C, size);
11
12   vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
13
14   cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
15
16   // Liberar memoria
17   cudaFree(d_A);
18   cudaFree(d_B);
19   cudaFree(d_C);
20 }
```

Este ejemplo es extremadamente sencillo, pero es un buen punto de partida para entender cómo hacer un código en CUDA, sin embargo en la práctica no es un ejemplo viable ya que existe una gran cantidad de overhead en reservar la memoria en el device, transferir los datos del host al device, copiar los resultados del device al host y liberar la memoria. Esto se da porque la cantidad de procesamiento en el kernel es mínima comparada con la cantidad de datos que va a procesar. Normalmente un kernel debería hacer muchísimo más trabajo en relación a los datos provistos.

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.