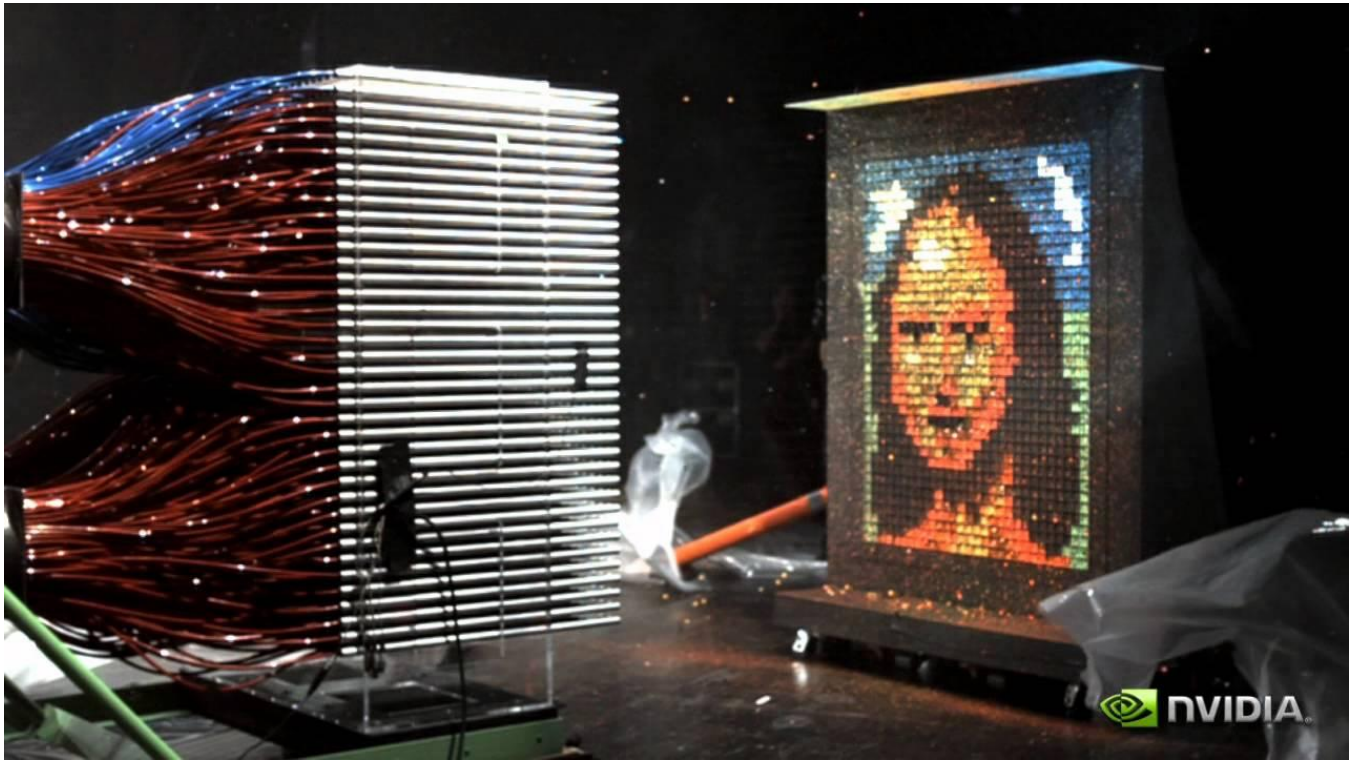


INTRODUCCIÓN A CUDA

Netz Romero

¿Qué es una GPU?

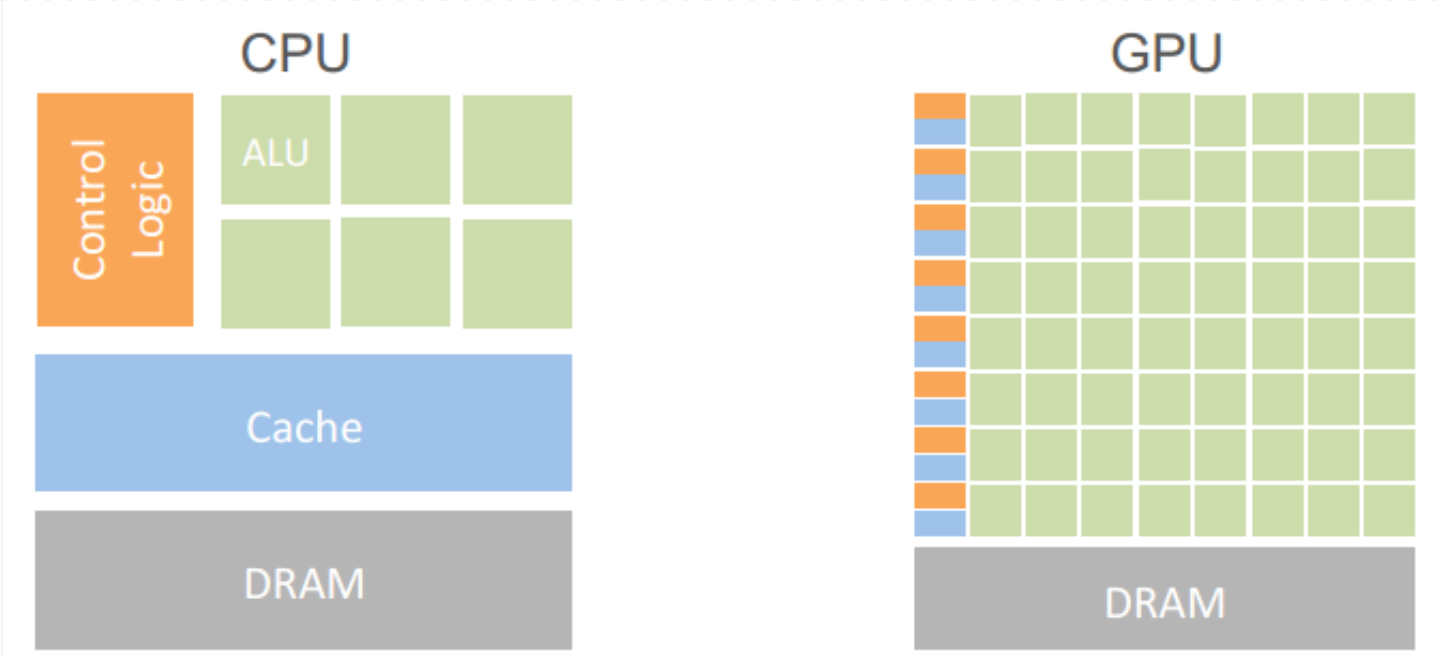
- La unidad de procesamiento gráfico o GPU es un procesador dedicado para el procesamiento de gráficos, para aligerar la carga de trabajo del procesador central.

Tecnología de GPU's

- Unidades de procesamiento gráfico
- Fabricantes: NVIDIA, ATI, Intel…
- Procesadores vectoriales
- Se van a aprovechar los GPU's para realizar operaciones computacionales de alto rendimiento.

CPU vs GPU

- Entre los CPU's y GPU's existen fundamentos de diseño diferente.



Diferencia entre CPU y GPU

- Una diferencia claramente explicada en:

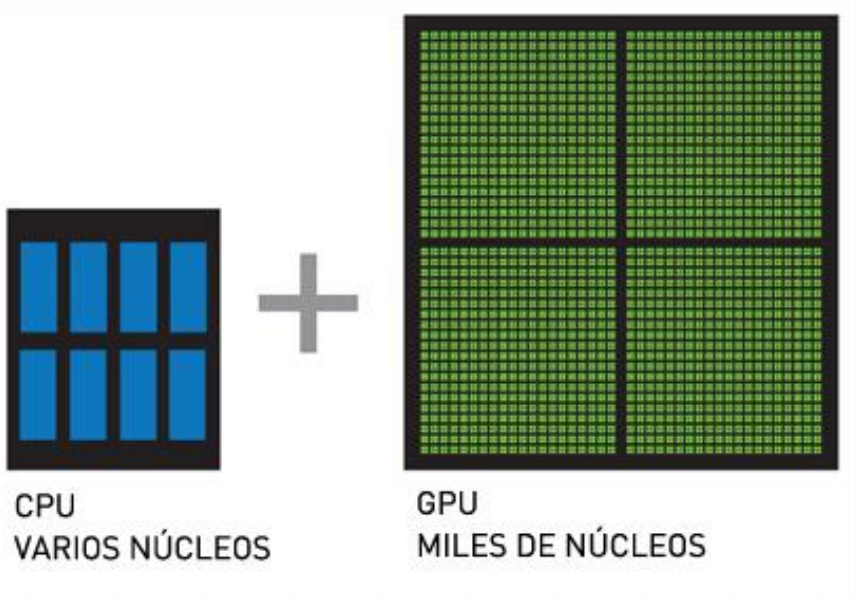
http://www.youtube.com/watch?v=-P28LKWTzrI

Comparativa CPU - GPU

- CPU como procesador secuencial donde la latencia importa, puede ser mucho mas rápido que un GPU para código secuencial.
- Los algoritmos paralelos aprovechan los atributos del GPU.
- Los GPU's proporcionan también una solución al procesamiento no gráfico (General-Purpose Computing on Graphics Processing Units, GPGPU).

¿Qué es el GPU Computing?

- Es el uso de la GPU junto con una CPU para acelerar operaciones de cálculo científico o técnico de propósito general.



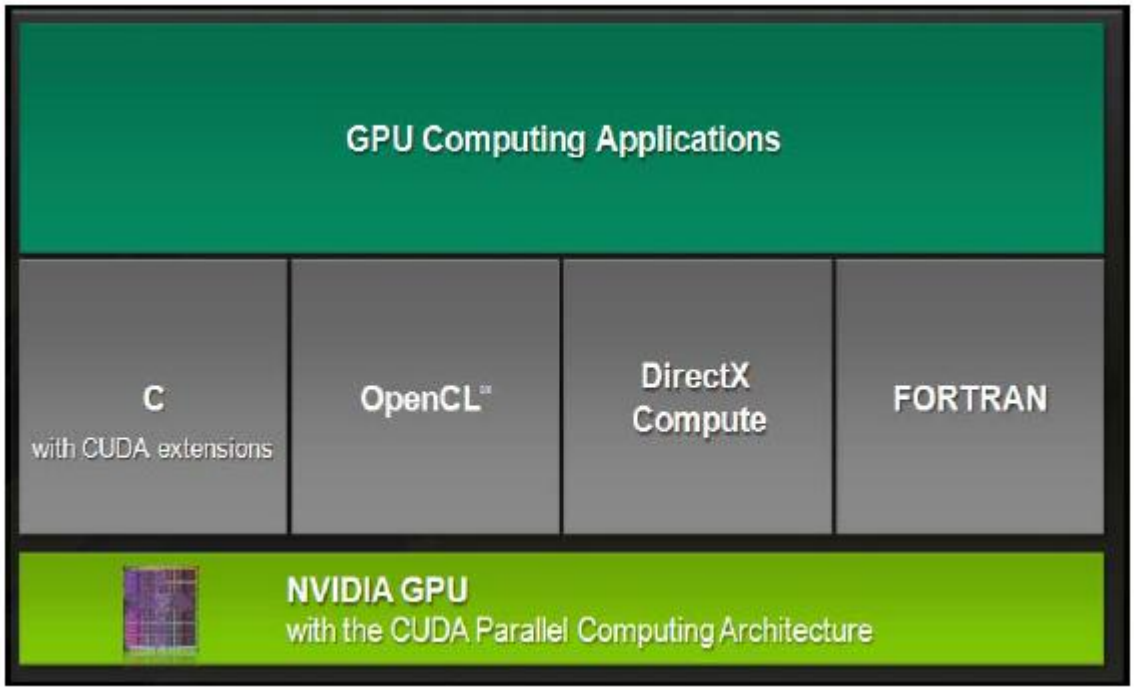
Hardware GPU

Plataforma de desarrollo

- La herramienta que utilizaremos para elaborar programas que se ejecuten en las arquitecturas CPU-GPU será CUDA.
- CUDA son las siglas de Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo)

CUDA

- Es una extensión del lenguaje de programación C



Primer Programa

```
1  #include<stdio.h>
2  __global__ void holaKernel(void) {
3  }
4  int main(void) {
5     holaKernel<<<1,1>>>();
6     printf("Hello, World!\n");
7     return 0;
8  }
```

- Para compilar un programa en CUDA es:

```
nvcc holaCuda.cu -o holaCuda
./holaCuda
```

Kernel y threads

- **Kernel.-** representa la sección de código que puede correr en los procesadores del GPU.
- El kernel es definido usando la declaración: `__global__`
- Cada thread que ejecuta el kernel esta dado por un único valor (thread ID) y puede ser accesible dentro del kernel.
- El número de threads se especifica en: `<<< , >>>`

Actividad

- Realizar un módulo que sume dos valores y el resultado lo regrese por los argumentos.
- Realizar el main para comprobar el módulo.
- Comprender paso por valor y paso por referencia.
- Comprender la asignación de memoria.

```
1   #include <stdio.h>
2   #include <book.h>
3   __global__ void add(int a, int b, int *c) {
4     *c = a + b;
5   }
6   int main( void ) {
7     int c;
8     int *dev_c;
9     HANDLE_ERROR(cudaMalloc((void**)&dev_c, sizeof(int)));
10    //Invocando al kernel
11    add<<<1,1>>>(2, 7, dev_c);
12    HANDLE_ERROR(cudaMemcpy(&c, dev_c, sizeof(int),
13                            cudaMemcpyDeviceToHost));
14    printf("2 + 7 = %d\n", c);
15    cudaFree(dev_c);
16    return 0;
17  }
```

	Código del kernel
4	`__global__ void add(int a, int b, int *c) {`
5	`  *c = a + b;`
6	`}`

	Llamada al kernel
	`...`
10	`//Invocando al kernel`
11	`add<<<1,1>>>(2, 7, dev_c);`
	`...`

Asignación de memoria

`cudaError_t cudaMalloc( void** devPtr, size_t size)`

Asigna `size` bytes de memoria en el dispositivo y devuelve un puntero `devPtr` a la memoria asignada.

Parametros:

`devPtr` - apuntador a la memoria asignada en el dispositivo

`size` - Tamaño en bytes de la memoria a alojar

Regresos:

cudaSuccess, cudaErrorMemoryAllocation

`cudaError_t cudaFree( void* devPtr)`

Libera memoria del espacio donde hace referencia el apuntador `devPtr`, el cuál debe ser asignado previa llamada a cudaMalloc

Parametros:

`devPtr` - apuntador a la memoria a liberar

Regresos:

cudaSuccess, cudaErrorInvalidDevicePointer, cudaErrorInitializationError

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

Copia `count` bytes del área de memoria apuntada por `src` a el área de memoria apuntada por `dst` y los valores de `kind` son: cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost y cudaMemcpyDeviceToDevice

Parametros:

`dst` – Destino del área de memoria

`size` – Origen del área de memoria

`count` – Tamaño en bytes a copiar

`kind` – tipo de transferencia

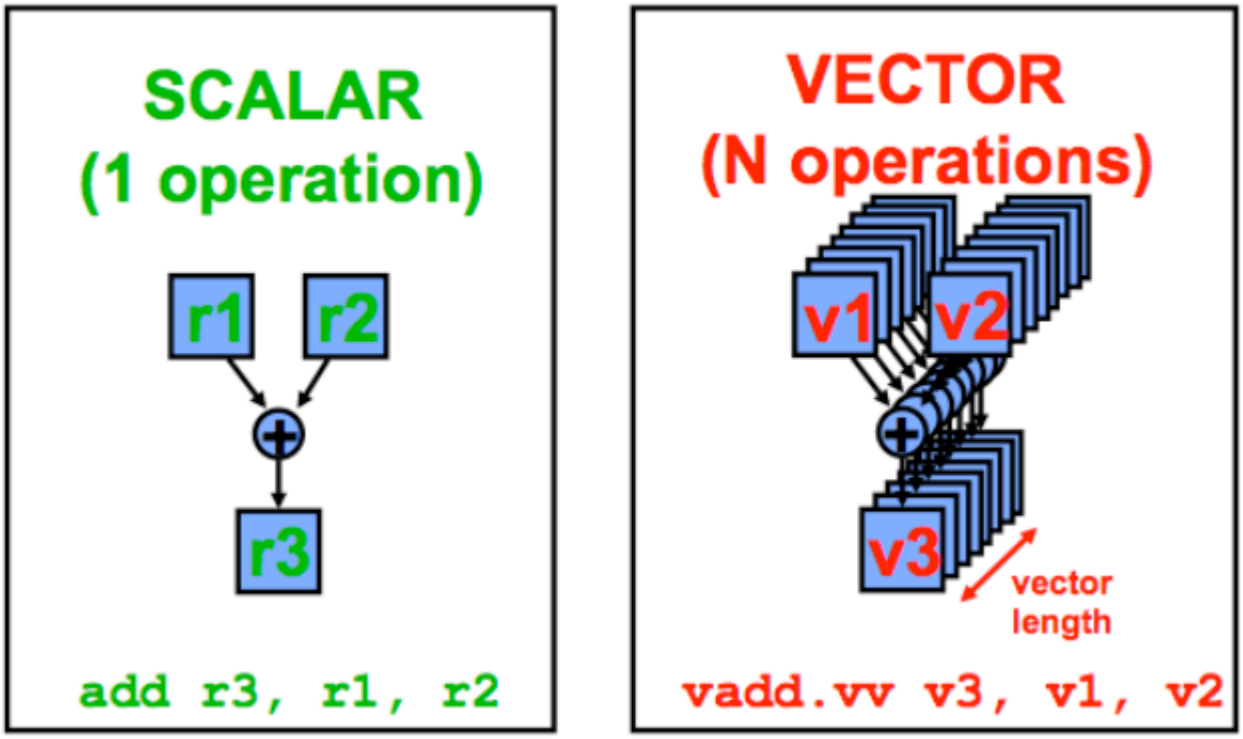
Regresos:

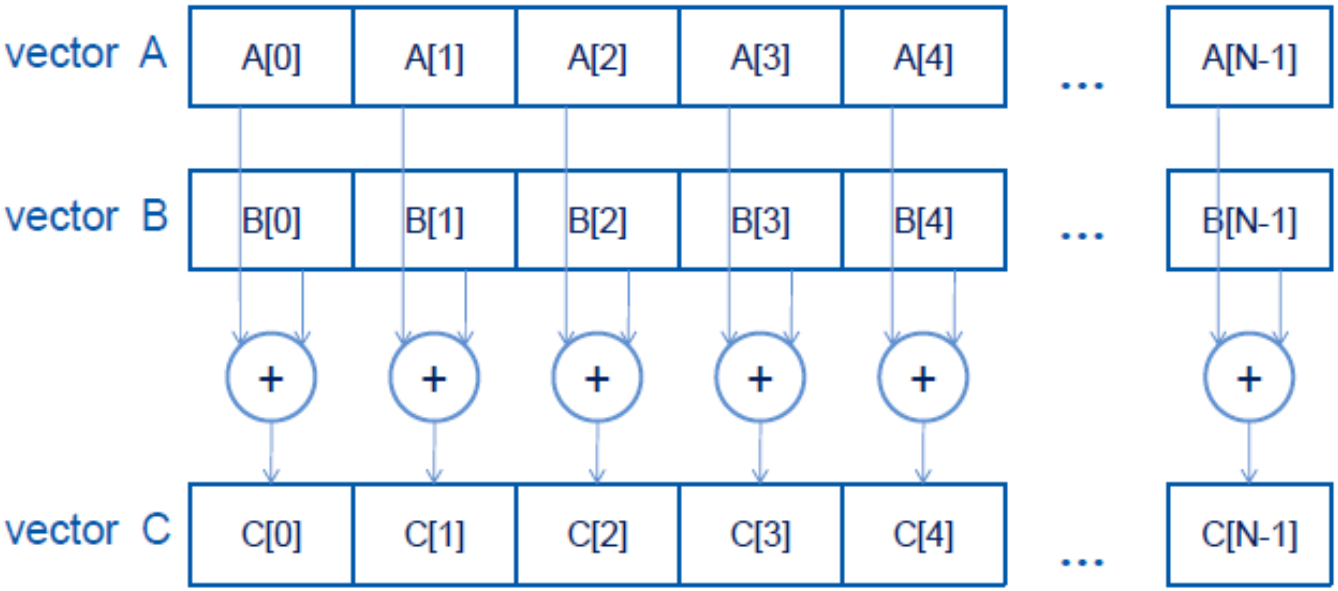
cudaSuccess, cudaErrorInvalidMemcpyDirection, cudaErrorInvalidValue, cudaErrorInvalidDevicePointer,

Actividad

- Realizar un módulo que sume dos vectores y el resultado lo regrese por los argumentos.
- Realizar el main para comprobar el módulo.
- Comprender el paso por referencia para arreglos de una dimensión.
- Comprender la asignación de memoria.

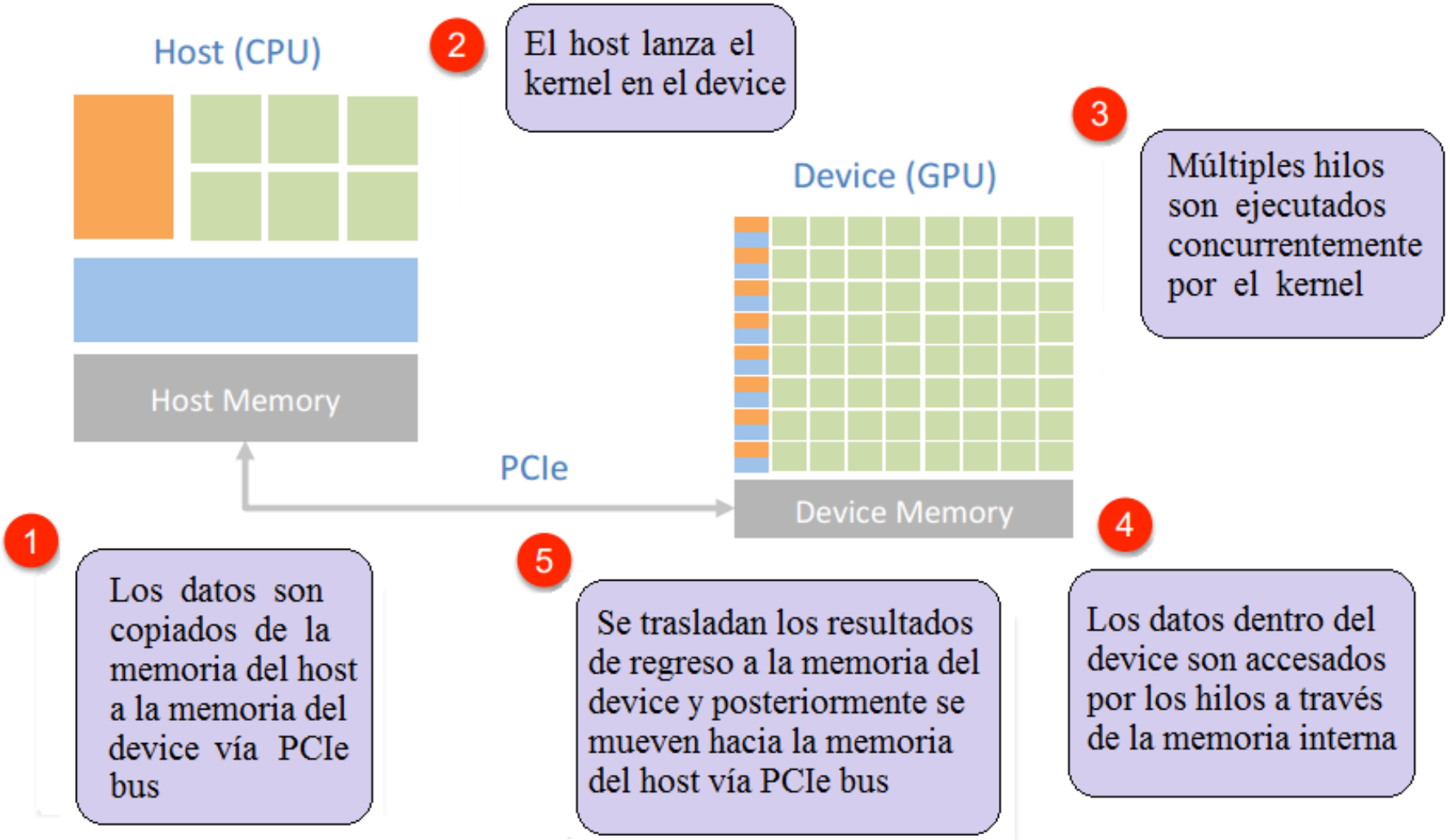
Suma de Vectores




vector A
A[0]
A[1]
A[2]
A[3]
A[4]
...
A[N-1]
vector B
B[0]
B[1]
B[2]
B[3]
B[4]
...
B[N-1]
+
+
+
+
+
+
vector C
C[0]
C[1]
C[2]
C[3]
C[4]
...
C[N-1]

```
1   #include <stdio.h>
2   #include <book.h>
3   #define N 4
4
5    // 3 El kernel es ejecutado por múltiples hilos
6    // 4 y los datos se procesan en una memoria interna
7   __global__ void add( int *a, int *b, int *c ) {
8    int tid = blockIdx.x;
9    if (tid < N)
10   c[tid] = a[tid] + b[tid];
11  }
12
13  int main( void ) {
14   int a[]={2,4,6,8}, b[]={1,2,3,4}, c[N];
15   int *dev_a, *dev_b, *dev_c;
16   // allocate the memory on the GPU
17   HANDLE_ERROR(cudaMalloc((void**)&dev_a, N*sizeof(int)));
18   HANDLE_ERROR(cudaMalloc((void**)&dev_b, N*sizeof(int)));
19   HANDLE_ERROR(cudaMalloc((void**)&dev_c, N*sizeof(int)));
```

```
20  // 1 Los datos son copiados del host al device(GPU)
21  HANDLE_ERROR(cudaMemcpy(dev_a, a, N*sizeof(int),
22  cudaMemcpyHostToDevice));
23  HANDLE_ERROR(cudaMemcpy(dev_b, b, N*sizeof(int),
24  cudaMemcpyHostToDevice));
25  // 2 El host lanza el kernel
26  add<<<N,1>>>(dev_a, dev_b, dev_c );
27  // 4 Los resultados se mueven del device al host
28  HANDLE_ERROR(cudaMemcpy(c, dev_c, N*sizeof(int),
29  cudaMemcpyDeviceToHost));
30  for (int i=0; i<N; i++) {
31     printf("%d + %d = %d\n", a[i], b[i], c[i]);
32  }
33  // free the memory allocated on the GPU
34  cudaFree(dev_a);
35  cudaFree(dev_b);
36  cudaFree(dev_c);
37  return 0;
38 }
39
```


Host (CPU)
Host Memory
PCIe
Device (GPU)
Device Memory
1
Los datos son copiados de la memoria del host a la memoria del device vía PCIe bus
2
El host lanza el kernel en el device
3
Múltiples hilos son ejecutados concurrentemente por el kernel
4
Los datos dentro del device son accesados por los hilos a través de la memoria interna
5
Se trasladan los resultados de regreso a la memoria del device y posteriormente se mueven hacia la memoria del host vía PCIe bus

- Cuando se lanza el kernel, se especifican N número de bloques paralelos (N = 4) en un grid.
- Se ejecutan varios threads que son identificados en este caso por `blockIdx.x`, desde el valor 0 hasta el valor N-1.
- Podemos imaginar cuatro bloques corriendo al mismo tiempo con los N-1 valores proporcionados por `blockIdx.x`, ver siguiente figura:

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Referencias

- Sito de NVIDIA, https://developer.nvidia.com/
- Aplicaciones, https://computing.llnl.gov/tutorials/parallel_comp/
- Tipos de memoria, http://www.cs.rit.edu/~ark/lectures/pj04/notes.shtml