2. Programación en CUDA

Arquitectura CUDA

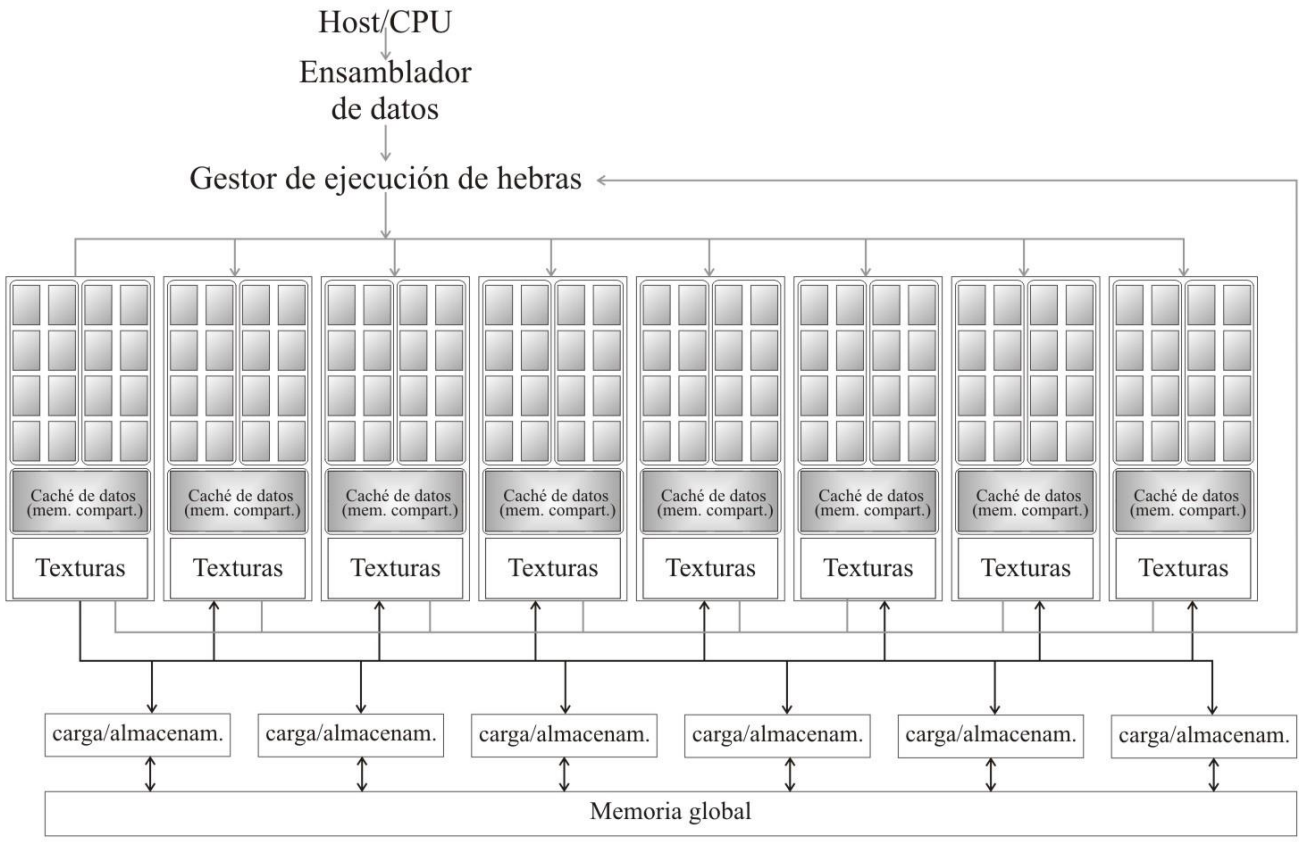
Arquitectura de procesadores

- Arquitectura de tipo “streaming”
  - En una arquitectura tradicional (Von Newmann) la instrucción busca datos en la memoria y estos se pasan a la CPU
  - En las tarjetas gráficas los datos “fluye” a través de la GPU
    - Evita cuellos de botella cuando se tienen cientos de procesadores

Arquitectura de procesadores

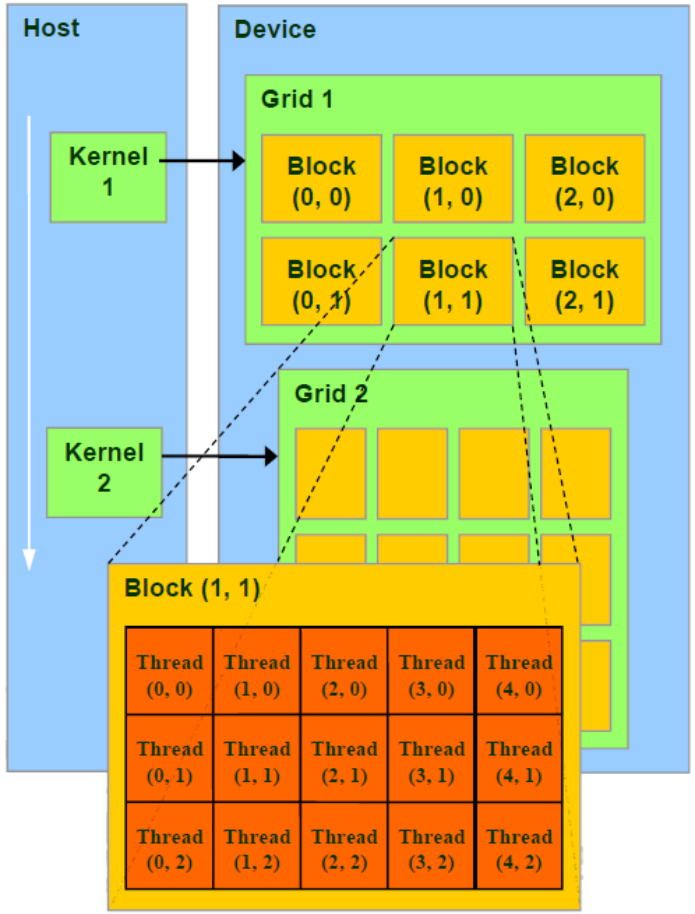
- Multiprocesador streaming
  - Bloques de multiprocesadores
  - Cada multiprocesador contiene
    - Un nivel L1 de cache compartido con los procesadores del mismo bloque
    - Una zona de memoria rápida compartida por el bloque
    - Bus de comunicaciones con la memoria principal

Arquitectura de procesadores



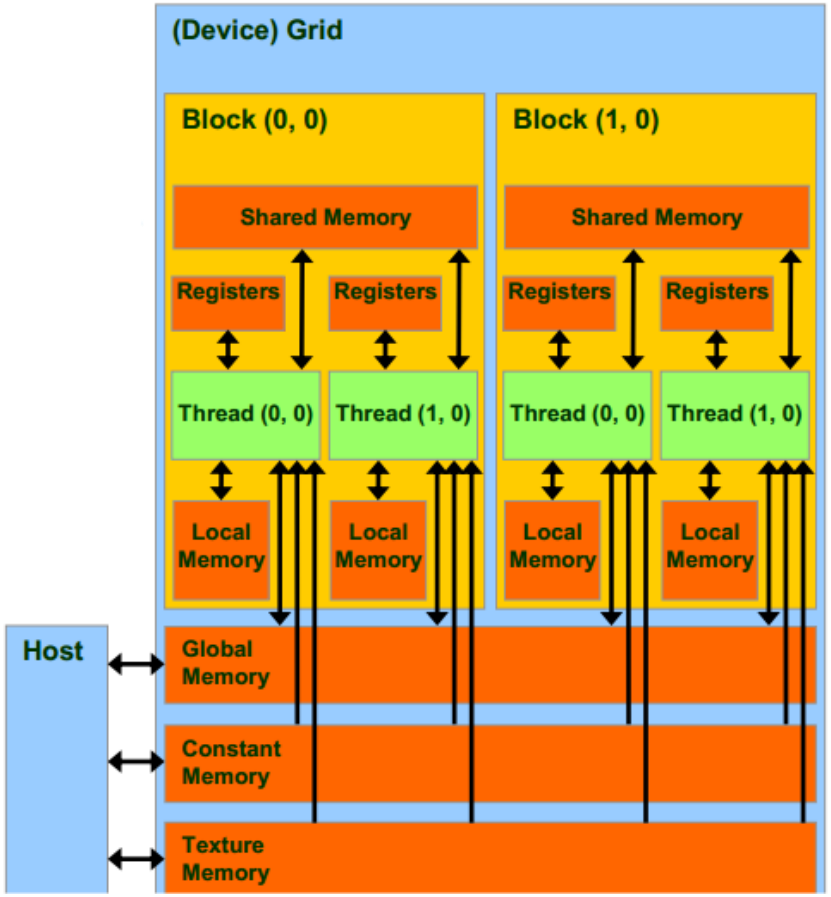
Arquitectura de procesadores

- La ejecución de un kernel en multiples hilos se organiza como un grid de bloques de hilos
- Los hilos dentro de un bloque se ejecutaran en el mismo multiprocesador, y gracias a esta localidad pueden cooperar entre ellas para
  - Sincronizar su ejecución
  - Compartir datos de forma eficiente
- Los hilos de diferentes multiprocesadores no pueden cooperar
- Los hilos y los bloques tienen identificadores, por lo que cada uno puede elegir sobre que datos trabaja
  - Simplifica el acceso a memoria al procesar datos multidimensionales



Arquitectura de memoria

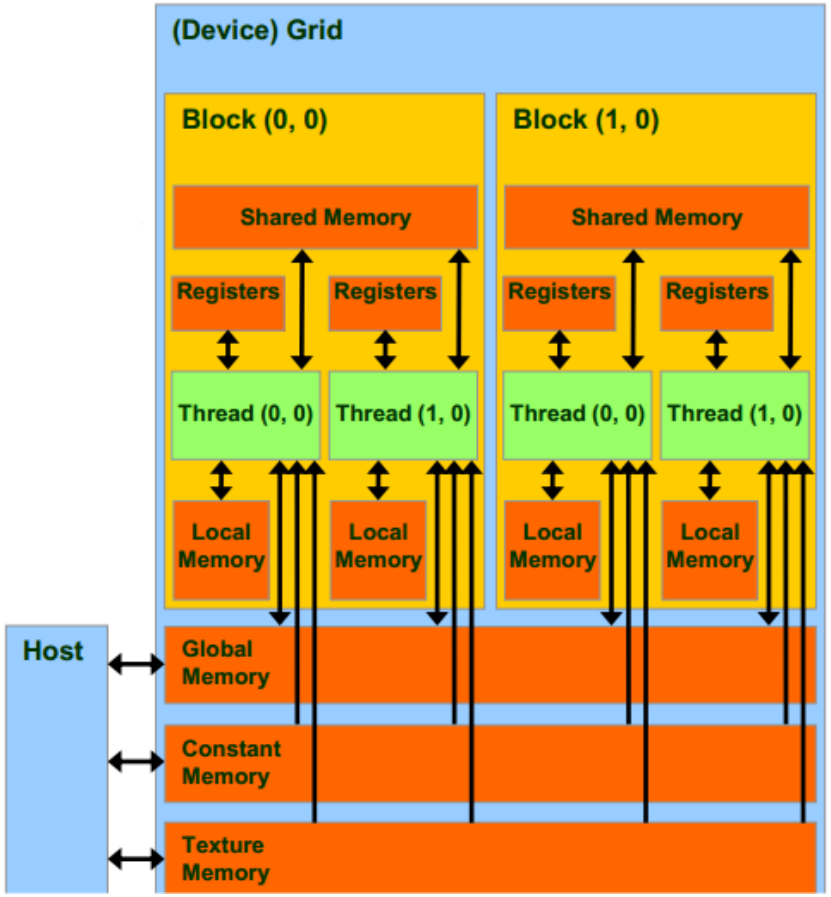
- Modelo de memoria:
  - Baja latencia
    - Memoria local de cada thread
    - Memoria compartida
      - Accesible para el mismo BLOQUE
  - Alta latencia
    - Memoria de Textura (Rápida en lectura, para operaciones especiales)
    - Memoria de constantes
    - Memoria principal
      - Accesibles por todos los bloques y por la CPU



Arquitectura de memoria

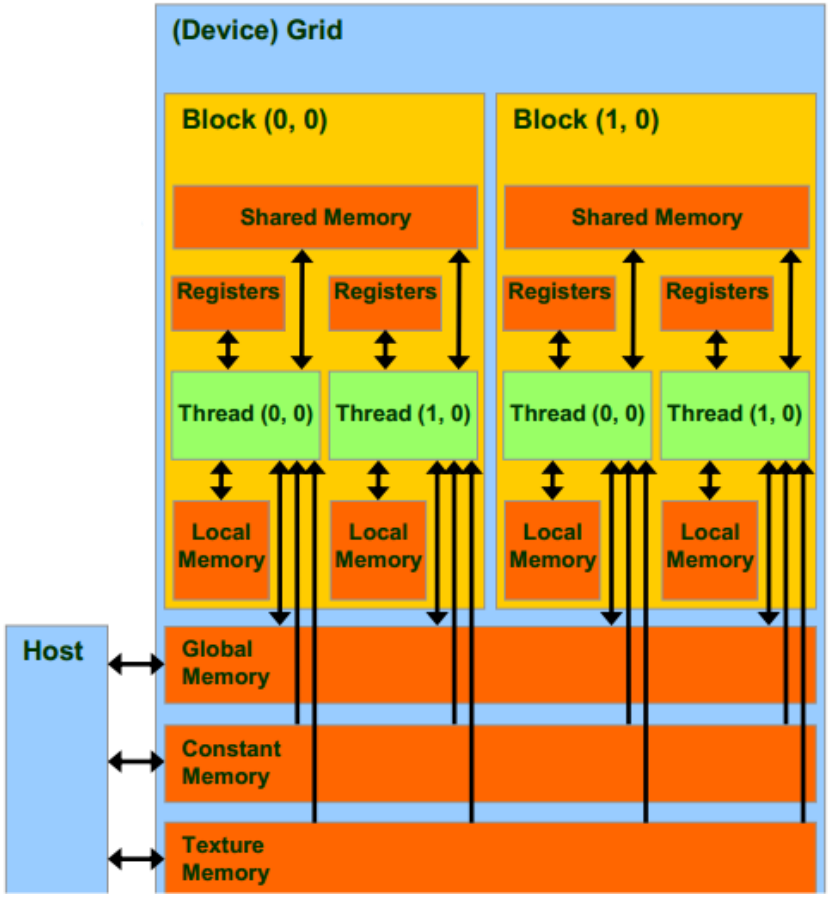
- Cada thread puede:
  - R/W registros para cada hebra
  - R/W memoria local a cada hebra
  - R/W memoria compartida para cada bloque
  - R/W memoria global en el grid
  - Consultar memoria de constantes en el grid
  - Consultar memoria de texturas en el grid

- La CPU (host), puede leer y escribir en la memoria global, de constantes, y de textura



Arquitectura de memoria

- Los espacios de memoria de texturas, constantes y global son regiones de la memoria de vídeo
- Cada multiprocesador tiene:
  - Un conjunto de registros de 32 bits por procesador
  - Memoria compartida On-chip
  - Cache para el acceso a la memoria de constantes
  - Caché para el acceso a memoria de texturas
    - Espera cierta localidad en los accesos
    - Es de sólo lectura



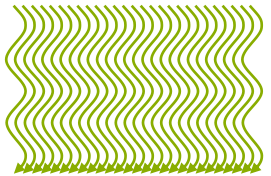
API CUDA

API CUDA

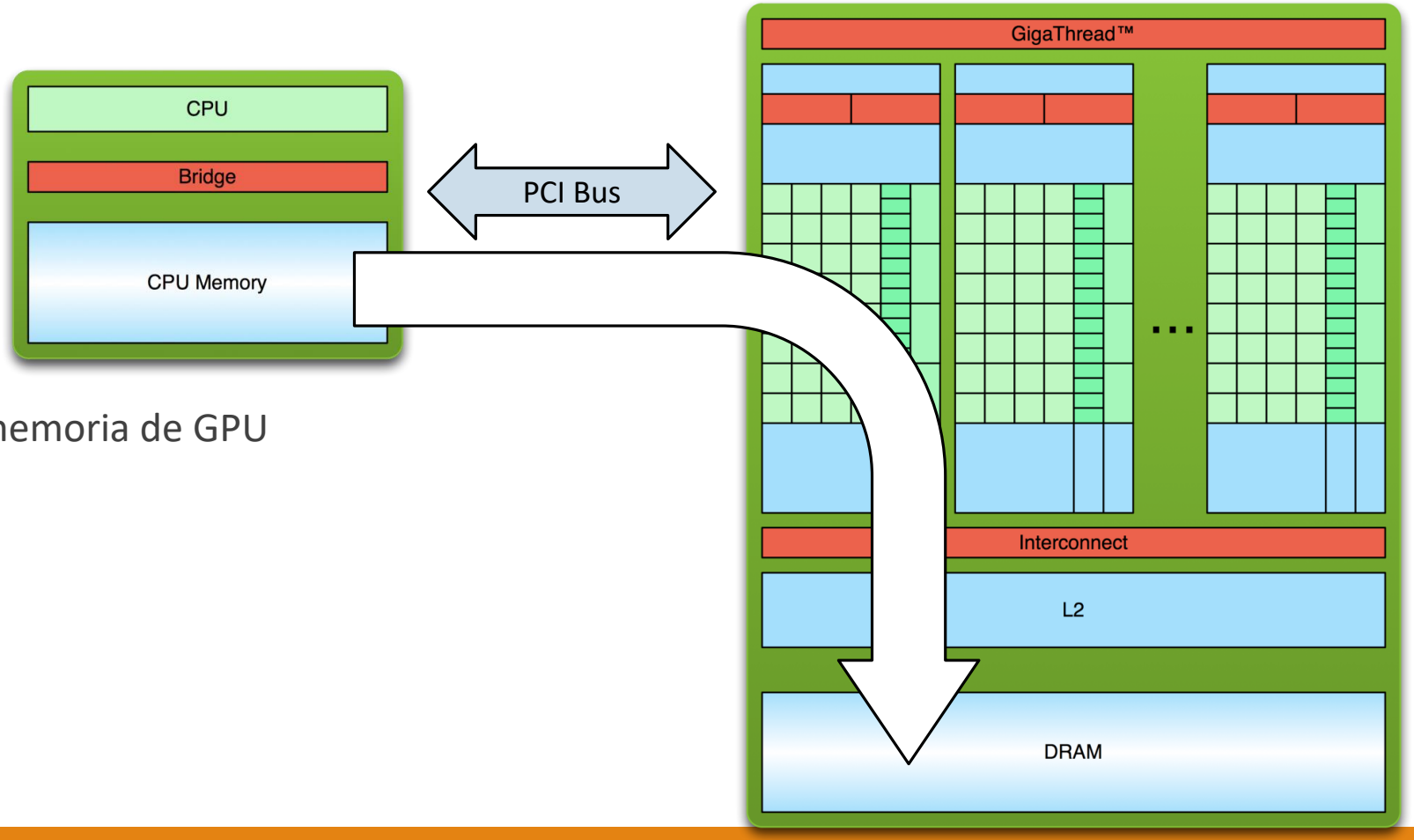
- Cuda es una extensión de C para programar con arquitecturas masivamente paralelas
  - Extiende el lenguaje C y C++, aunque existe para otros lenguajes
- Define secciones del código para colocar variables y métodos, según sean para
  - El Host (Memoria de CPU)
  - EL Device (Memoria de GPU)
- Normalmente se llama desde Host a las funciones de GPU usando métodos predefinidos

API CUDA

- Nuestros programas contendrán:
  - Codigo de Host:
    - Se ejecuta de forma secuencial
    - Metodos de programación tradicionales
    - Prepara los datos para GPU
  - Código de Device
    - Metodos “masivamente paralelos”
    - Se crearan cientos de theads
    - Al terminar devuelve el control a la CPU, y recuperará los resultados

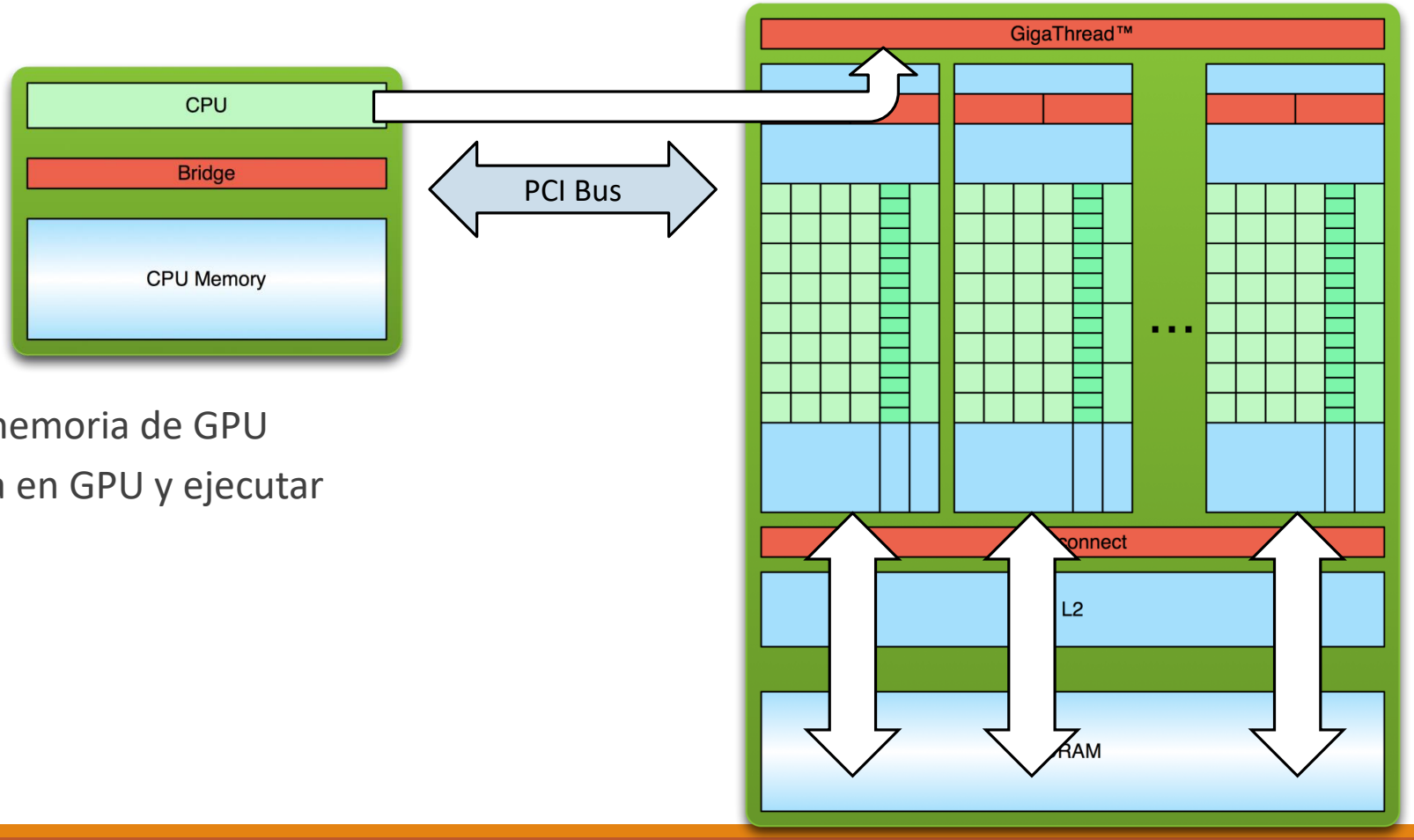


API CUDA



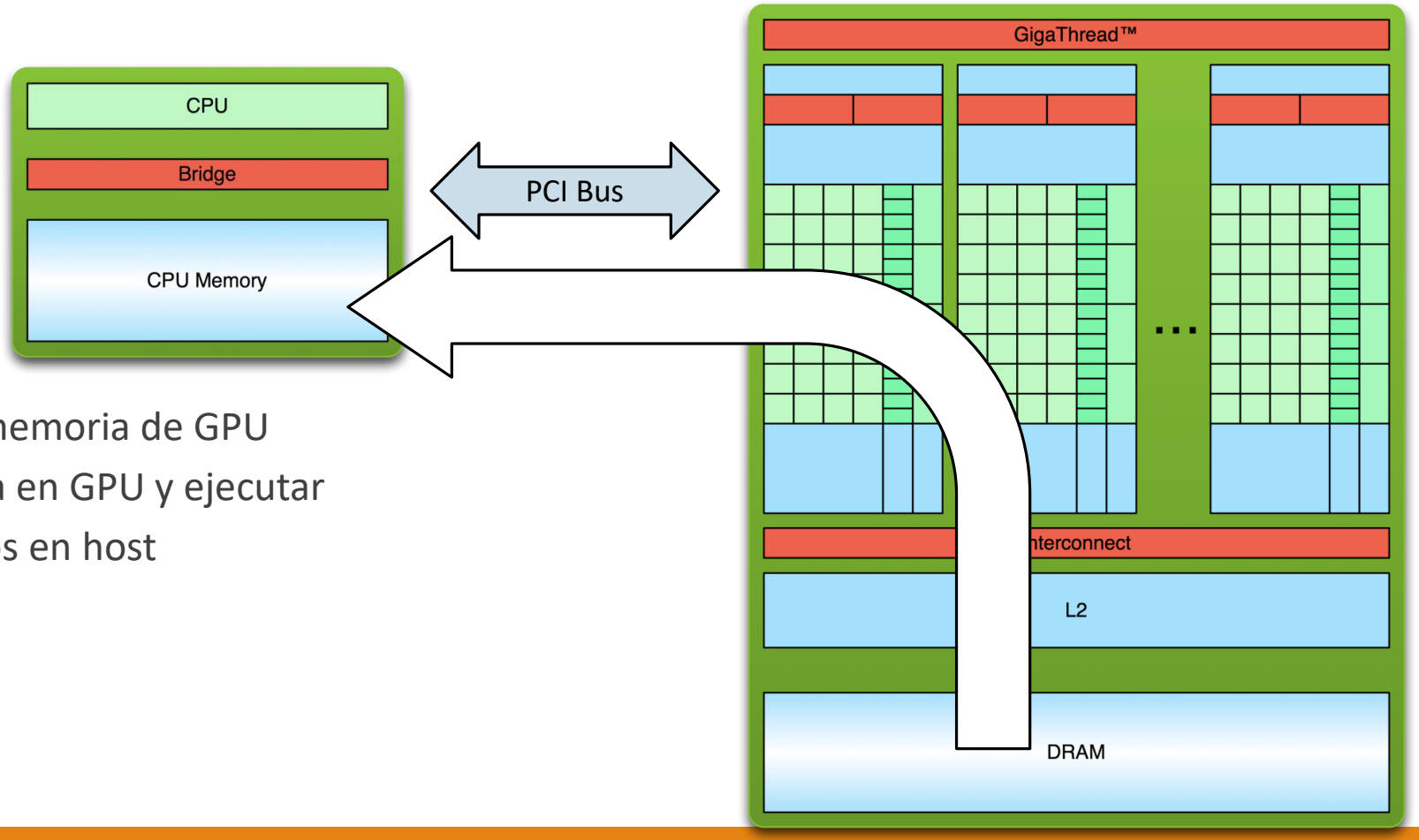
- 1º Copiar datos a memoria de GPU

API CUDA



- 1º Copiar datos a memoria de GPU
- 2º Cargar programa en GPU y ejecutar

API CUDA



- 1º Copiar datos a memoria de GPU
- 2º Cargar programa en GPU y ejecutar
- 3º Copiar resultados en host

API CUDA

“¡Hola mundo!”

```
__global__ void holamundo(void) {
            printf(“¡Hola mundo!\n");
      }

      int main(void) {
            holamundo<<<1,1>>>();
            return 0;
      }
```

API CUDA

- cudaMalloc()
  - Reserva espacion en la memoria global del dispositivo (Memoria de video)
- cudaFree()
  - Libera el espacio de memoria que comienza en el puntero dado
- cudaMemcpy()
  - Transfiere datos entre:
    - Host-Host
    - Host-Device
    - Device-Host
    - Device-Device
  - Asíncrono a partir de CUDA 5.5

```
int *d_a, *a;
cudaMalloc((void **)&d_a, sizeof(int));
a=(int*)malloc(sizeof(int));
*a=10;
cudaMemcpy(d_a,a,sizeof(int,cudaMemcpyHostToDevice);
cudaMemcpy(a,d_a,sizeof(int,cudaMemcpyDeviceToHost);
cudaFree(d_a);
free(a);
```

API CUDA

- La API está hecha de forma que nos resulte como una extensión del lenguaje C
- Tenemos:
  - Extensiones del lenguaje
    - Algunas partes del código se compilan para la tarjeta gráfica
- Una librería runtime con las siguientes partes:
  - Un componente común con tipos predefinidos y un subconjunto de C que nos permite escribir código en la CPU (host) y la GPU (device)
  - Un componente en el host para controlar el acceso a uno ó más devices
  - Un componente de device que nos da la funcionalidad propia de la GPU (memoria de vídeo, arrays de procesadores…)

API CUDA

	Memoria	Ámbito	Tiempo de vida
`__device__ __local__ int LocalVar;`	local	hebra	hebra
`__device__ __shared__ int SharedVar;`	compartida	bloque	bloque
`__device__ int GlobalVar;`	global	grid	aplicación
`__device__ __constant__ int ConstantVar;`	constante	grid	aplicación

- __device__ es opcional cuando se usa con__local__, __shared__, or __constant__
- Las variables “automáticas” sin un “cualificador” se guardan como registros en la GPU
- Excepto los arrays que se encuentran en memoria local

API CUDA

- Los punteros sólo pueden apuntar a zonas de memoria global:
  - Reservadas en el host y copiadas al kernel:

```
__global__ void KernelFunc(float* ptr)
```

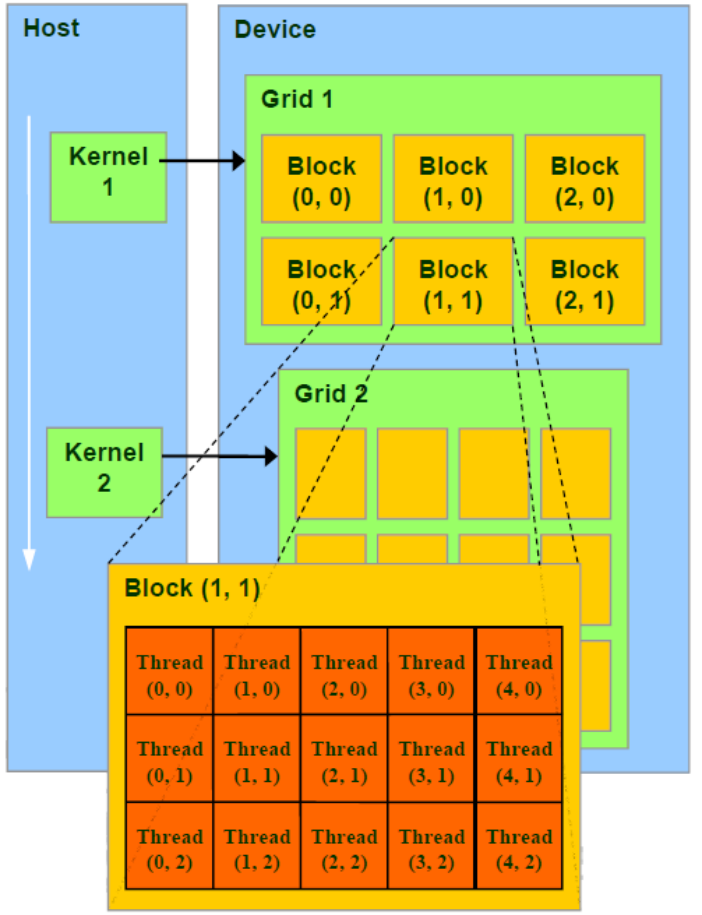
- O como dirección obtenida de una variable global:

```
float* ptr = &GlobalVar;
```

API CUDA

Variables y registros útiles

- `dim3 gridDim;`
  - Las dimensiones del grid en número de bloques (`gridDim.z` no se usa)
- `dim3 blockDim;`
  - Las dimensiones del bloque en número de hebras
- `dim3 blockIdx;`
  - El índice/identificador del bloque dentro del grid
- `dim3 threadIdx;`
  - El índice/identificador de la hebra dentro del bloque



API CUDA

Además nos ofrece:

- Funciones matemáticas (*sin*, *cos*, *exp*, *pow*, *sqrt*…)
  - En CPU utilizan la implementación de C (si existe)
- Tipos vectoriales predefinidos
  - *float1*, *float2*, *float3*, *float4*
  - *int1*, *int2*, …
  - *uint1*, *uint2*, …
  - *char1*, …
  - *dim3*: basado en *uint3*, se usa para especificar dimensiones
    - Todos tienen los campos *x*, *y*, *z*, *w*.

API CUDA

Runtime para Host

- Nos ofrece funciones para:
  - Gestión del Device (también para sistemas con multiples GPUs)
  - Gestión de Memoria
  - Gestión de Errores
- Se inicializa la primera vez que se llama a una función del runtime

API CUDA

Sincronización

```
void __syncthreads();
```

- Sincroniza todos los threads en un bloque
- Una vez todas las hebras llegan a este punto, la ejecución continúa normalmente
- Se utilizan para evitar riesgos RAW/WAR/WAW al acceder a memoria compartida ó memoria global
- Se pueden usar dentro de estructuras condicionales sólo si la condición es uniforme en todo el bloque de hebras

API CUDA

Sincronización

```
for(int jump=1;jump<blockDim.x;jump*=2){
        if(i%(2*jump) != 0) return;
        f_val[i] += f_val[i+jump];
        __syncthreads();
}
```

API CUDA

Compilación

- Cualquier fichero de código fuente que tenga extensiones del lenguaje con CUDA debe ser compilado con nvcc
- nvcc es un compiler driver
  - Invoca a todas las herramientas y compiladores necesarios: cudacc, g++, cl, ...
- nvcc da como salida:
  - Código C
    - Que tendrá que ser compilado con el resto de aplicaciones
  - O directamente código objeto

Recomendaciones