POLITECNICO
MILANO 1863

POLITECNICO MILANO 1863
NECST laboratory

Introduction to CUDA: CUDA Programming and Memory Models

Beatrice Branchini
beatrice.branchini@polimi.it

Advanced Computer Architecture Course
T.1.1 - 5/5/2024

Who I Am

Beatrice Branchini

POLITECNICO MILANO 1863

Ph.D. Information Technology
2022 – Now

AMD

Intern DCGPU HPC Solutions
Feb 2025 – Now

Pacific Northwest NATIONAL LABORATORY

Ph.D. Intern (HPC Group)
Sept 2023 – Mar 2024

Research interests:

- Genomics
- High Performance Computing
- Heterogeneous architectures (GPUs and FPGAs)
- Quantum Computing

More Computing Power

Financial Analysis

Scientific Simulation

Engineering Simulation

Data Intensive Analytics

Medical Imaging

Digital Audio Processing

Digital Video Processing

Computer Vision

Biomedical Informatics

Electronic Design Automation

Statistical Modeling

Numerical Methods

Ray Tracing Rendering

Interactive Physics

How Can We Program GPUs?

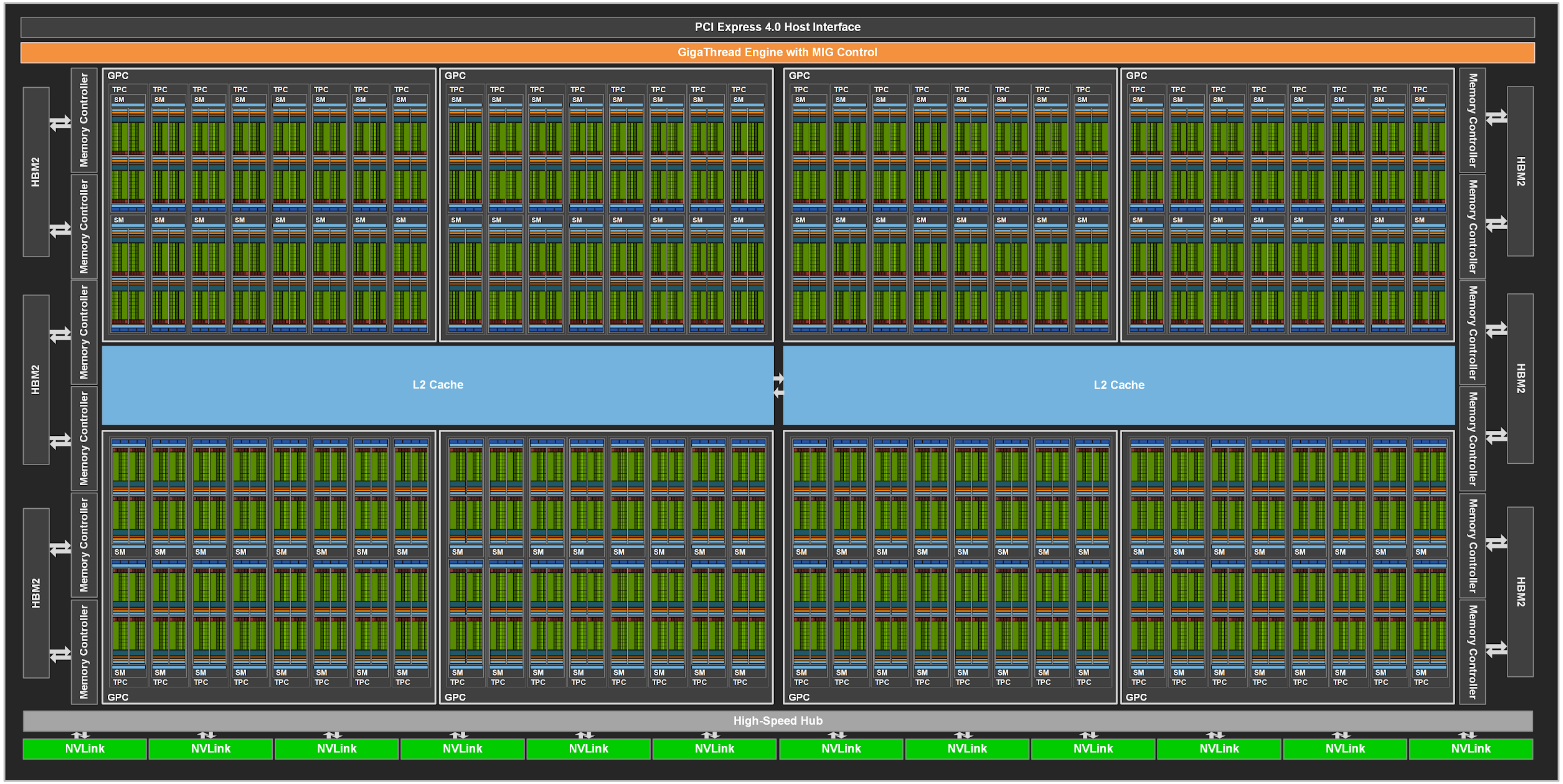
- **CUDA** (Compute Unified Device Architecture) has been introduced in 2006 with NVIDIA Tesla architecture. It offers a **C/C++-like language** which can be used to write programs leveraging GPUs as a processing resource to accelerate functions.
- CUDA's abstraction level closely **match** the **characteristics** and **organization** of NVIDIA GPUs

How Can We Program GPUs?

- **CUDA** (Compute Unified Device Architecture) has been introduced in 2006 with NVIDIA Tesla architecture. It offers a **C/C++-like language** which can be used to write programs leveraging GPUs as a processing resource to accelerate functions.
- CUDA's abstraction level closely **match** the **characteristics** and **organization** of NVIDIA GPUs

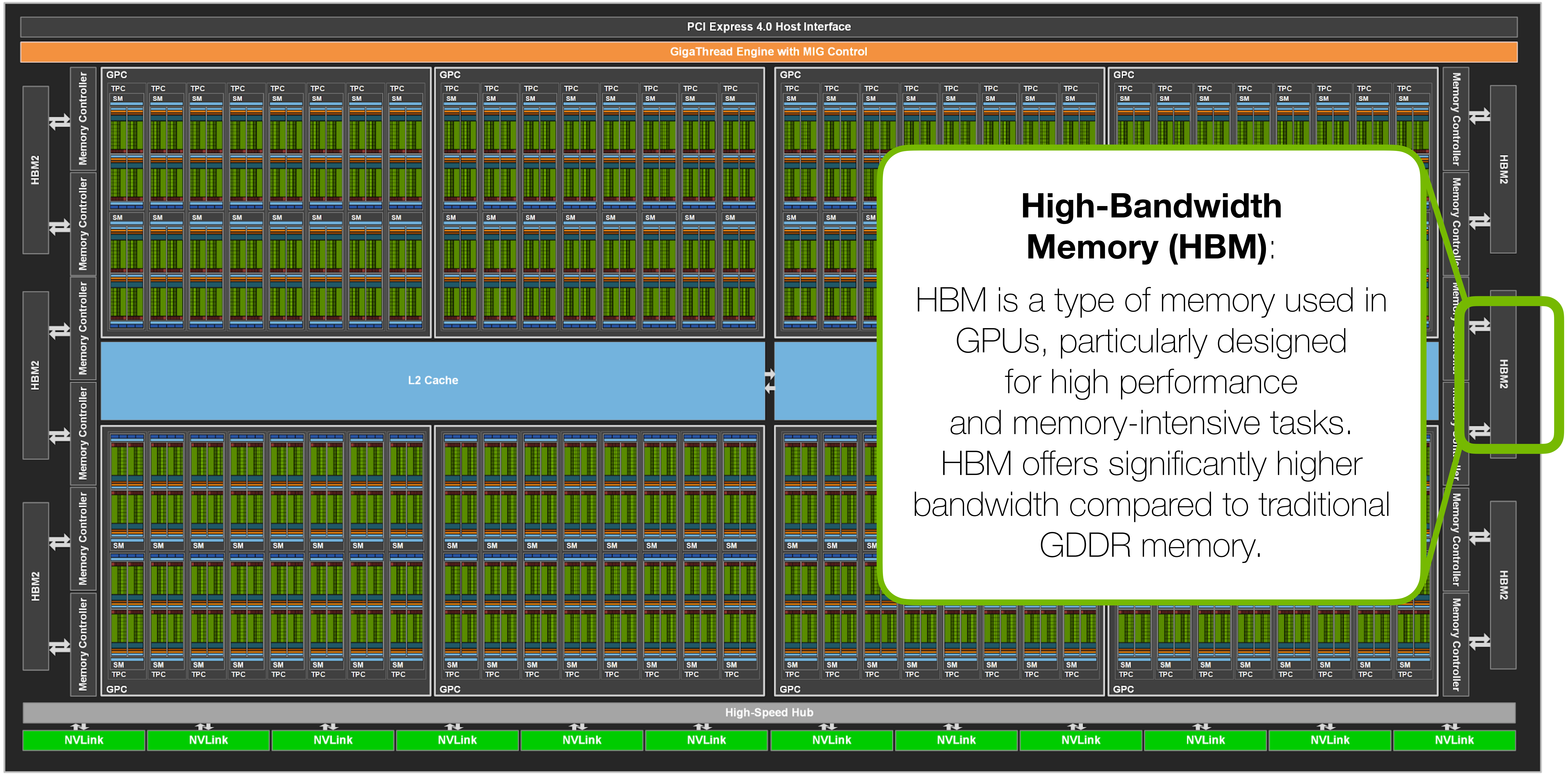
It is paramount to have a deep understanding of the device architecture to exploit GPUs efficiently

Recap: Device Structure



https://developer.download.nvidia.com/devblogs/ga100-full-gpu-128-sms.png

Recap: Device Structure

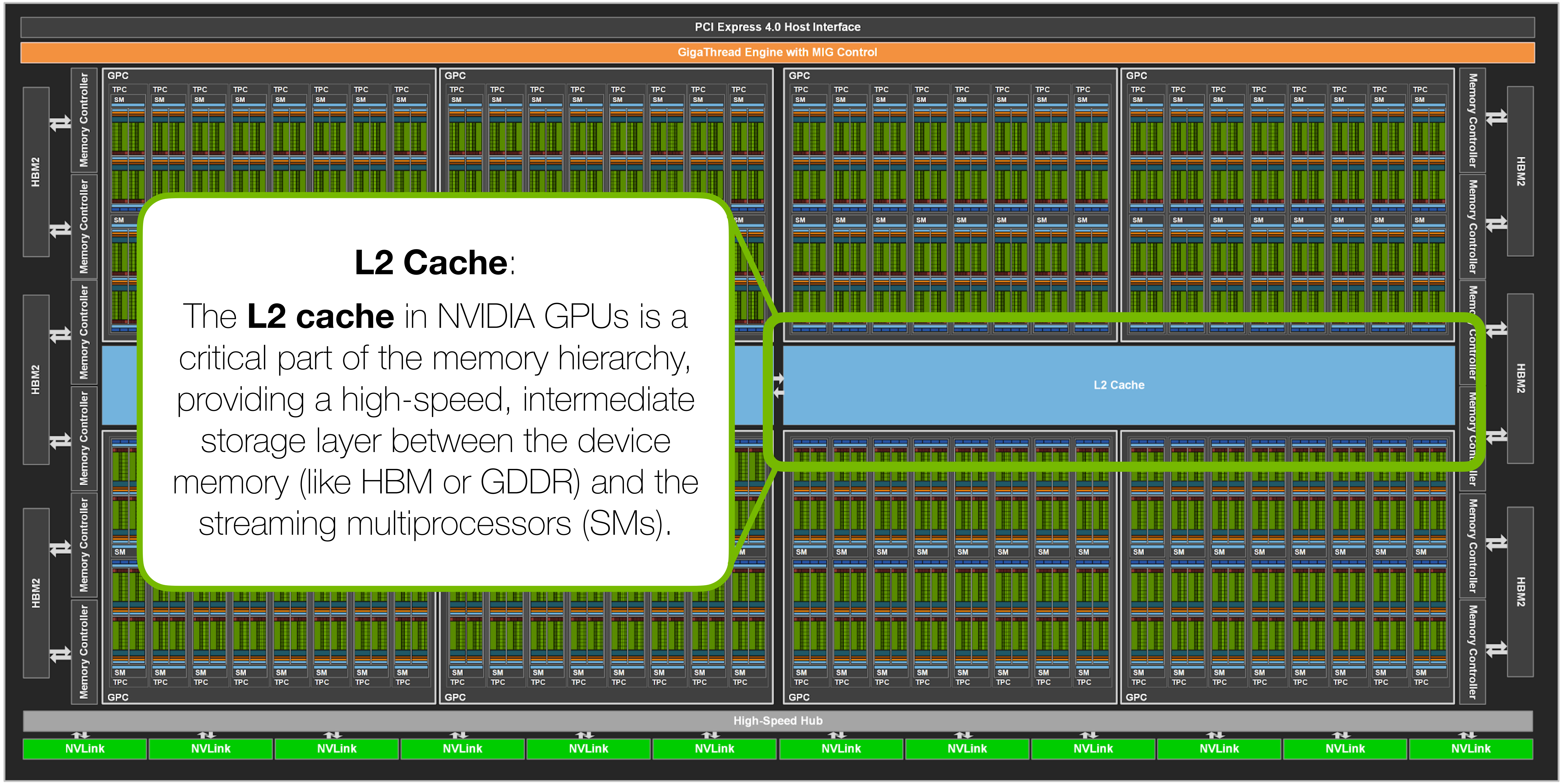

PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

High-Bandwidth Memory (HBM):

HBM is a type of memory used in GPUs, particularly designed for high performance and memory-intensive tasks. HBM offers significantly higher bandwidth compared to traditional GDDR memory.

https://developer.download.nvidia.com/devblogs/ga100-full-gpu-128-sms.png

Recap: Device Structure

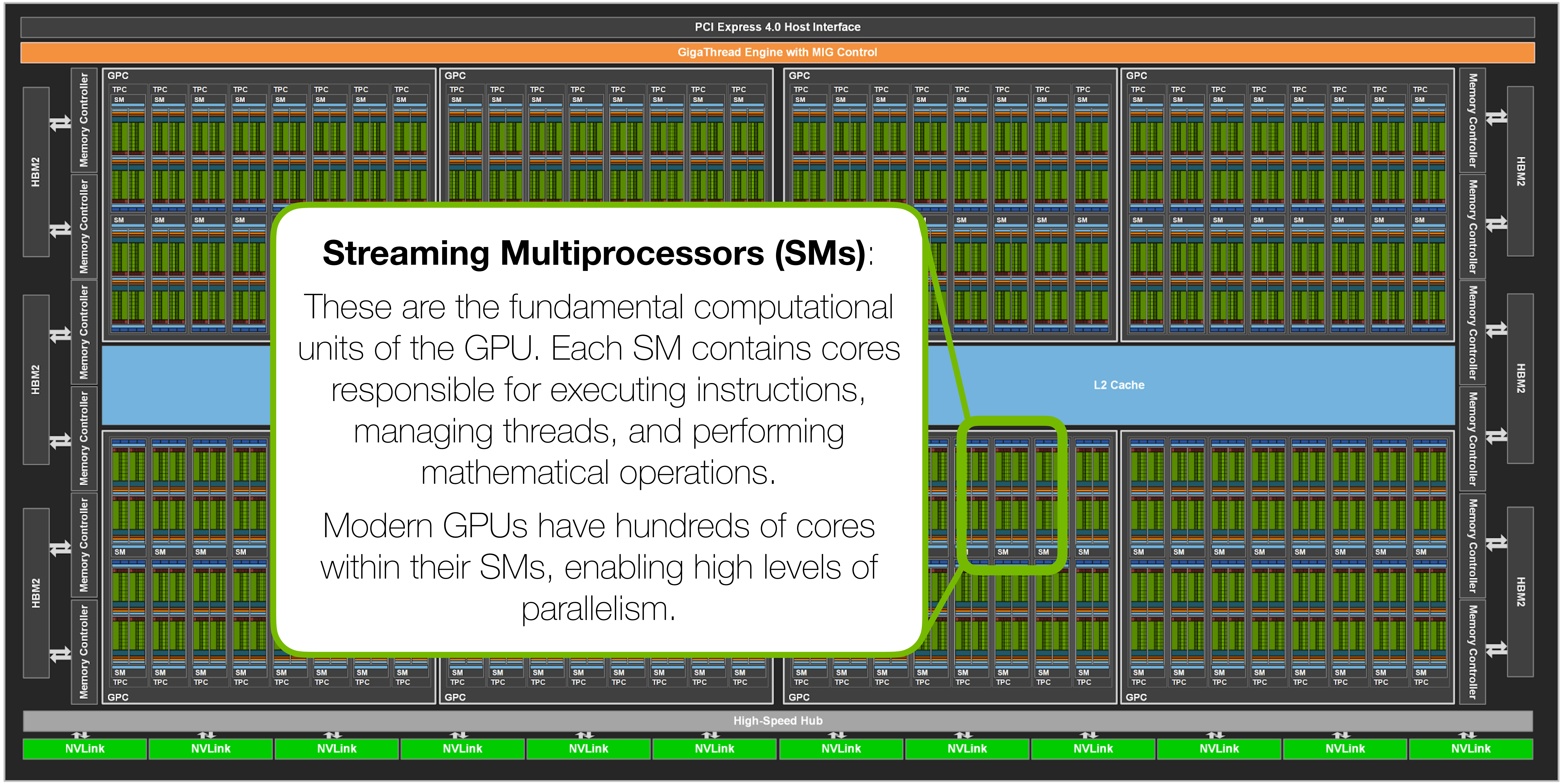

PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

L2 Cache:

The **L2 cache** in NVIDIA GPUs is a critical part of the memory hierarchy, providing a high-speed, intermediate storage layer between the device memory (like HBM or GDDR) and the streaming multiprocessors (SMs).

https://developer.download.nvidia.com/devblogs/ga100-full-gpu-128-sms.png

Recap: Device Structure



PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
High-Speed Hub
NVLink

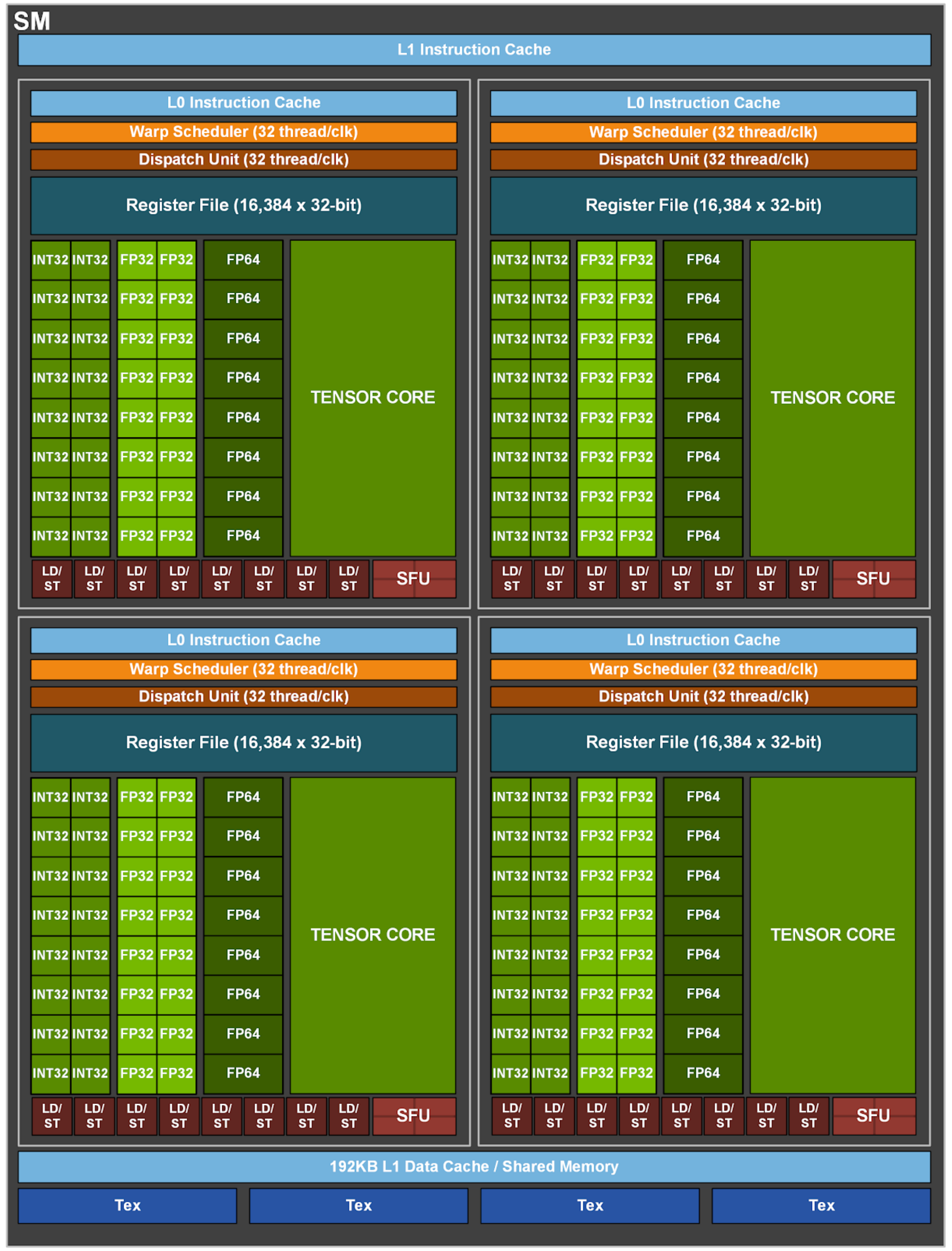
Streaming Multiprocessors (SMs):

These are the fundamental computational units of the GPU. Each SM contains cores responsible for executing instructions, managing threads, and performing mathematical operations.

Modern GPUs have hundreds of cores within their SMs, enabling high levels of parallelism.

https://developer.download.nvidia.com/devblogs/ga100-full-gpu-128-sms.png

Recap: Device Structure

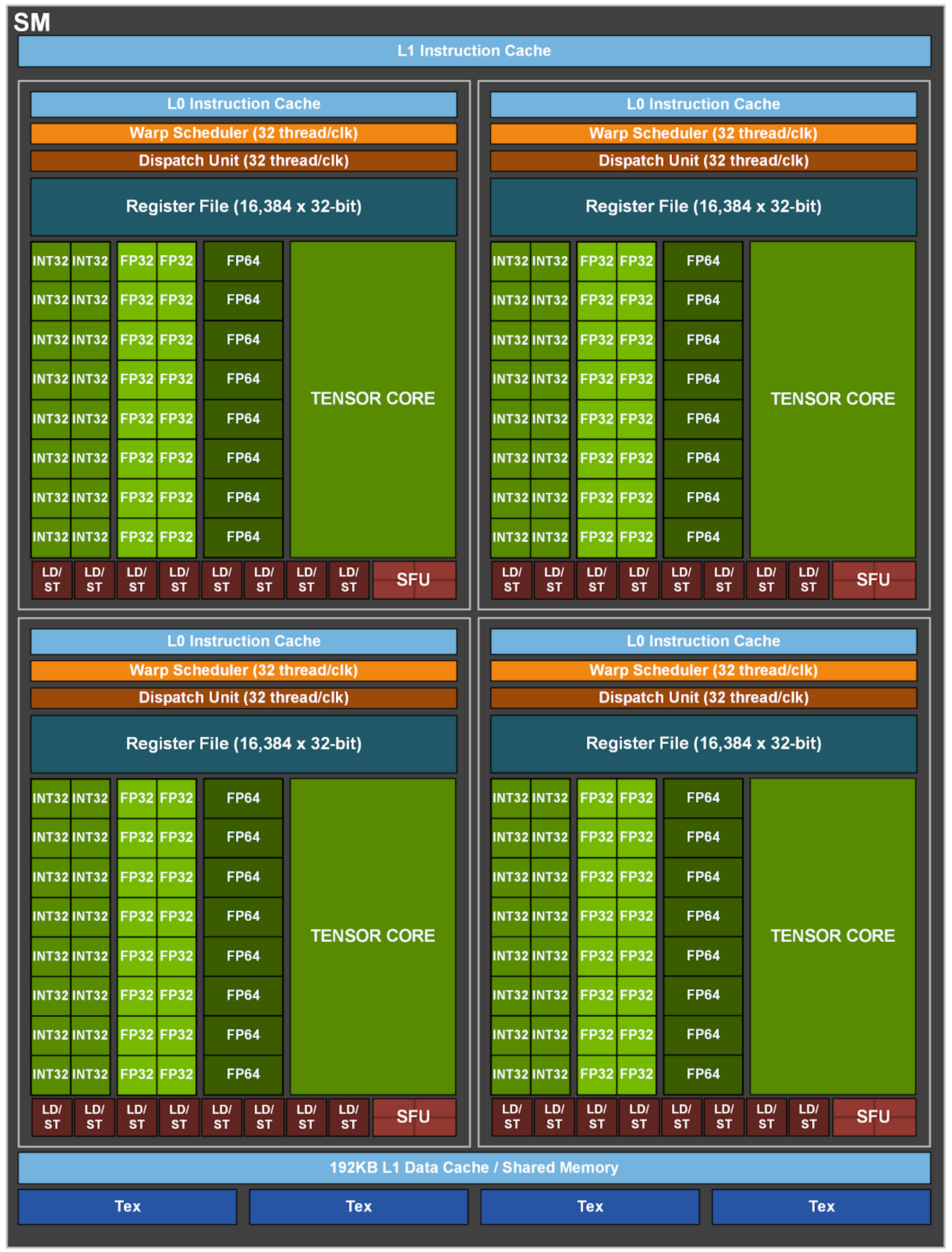

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

SMs feature:

- **CUDA Cores**: Responsible for general-purpose computations, including floating-point and integer arithmetic.
- **Specialized units**: These handle specific tasks, like:
    - **Tensor Cores**: Present in recent architectures like Volta, Turing, and Hopper, Tensor Cores accelerate deep learning operations by performing matrix multiplications at high speeds.

https://developer.download.nvidia.com/devblogs/ga100-full-gpu-128-sms.png

Recap: Device Structure


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

SMs feature:

- **L1 Cache**: a smaller, faster cache compared to the L2 cache, located within each SM. It primarily stores data that the SMs access frequently, reducing the need for repeated access to slower global memory.
- **Registers**: fastest form of memory available to each SM. They play a crucial role in storing data that threads frequently access, such as variables and intermediate results during computations.

https://developer.download.nvidia.com/devblogs/ga100-full-gpu-128-sms.png

CUDA

A **platform** designed jointly at **software** and **hardware levels**
to make use of GPUs in **general-purpose computing**:

- **Software** –CUDA allows to program the GPU with minimal extensions to enable heterogeneous programming and attain an efficient and scalable execution
- **Firmware** – CUDA offers a driver oriented to GPGPU programming and APIs to manage devices, memory, etc.
- **Hardware** – CUDA exposes GPU parallelism for general-purpose computing via a number of multiprocessors endowed with cores and a memory hierarchy

CUDA

From an application perspective, CUDA defines:

- **Architecture model**
  - with many processing cores grouped in multiprocessors who share a SIMT control unit
- **Programming model**
  - based on **massive data parallelism** and fine-grained parallelism
  - **scalable**: the code is executed on a different number of cores without recompiling it
- **Memory model**
  - more **explicit** to the programmer, where **caches are not transparent** anymore

CUDA

From an application perspective, CUDA defines:

- **Architecture model**
    - with many processing cores grouped in multiprocessors who share a SIMT control unit
- **Programming model**
    - based on **massive data parallelism** and fine-grained parallelism
    - **scalable**: the code is executed on a different number of cores without recompiling it
- **Memory model**
    - more **explicit** to the programmer, where **caches are not transparent** anymore

CUDA Programming Model

It **abstracts** the **complexities of GPU hardware**, allowing developers to write code that leverages the massive parallelism provided by GPUs.

CUDA Programming Model

It **abstracts** the **complexities of GPU hardware**, allowing developers to write code that leverages the massive parallelism provided by GPUs.

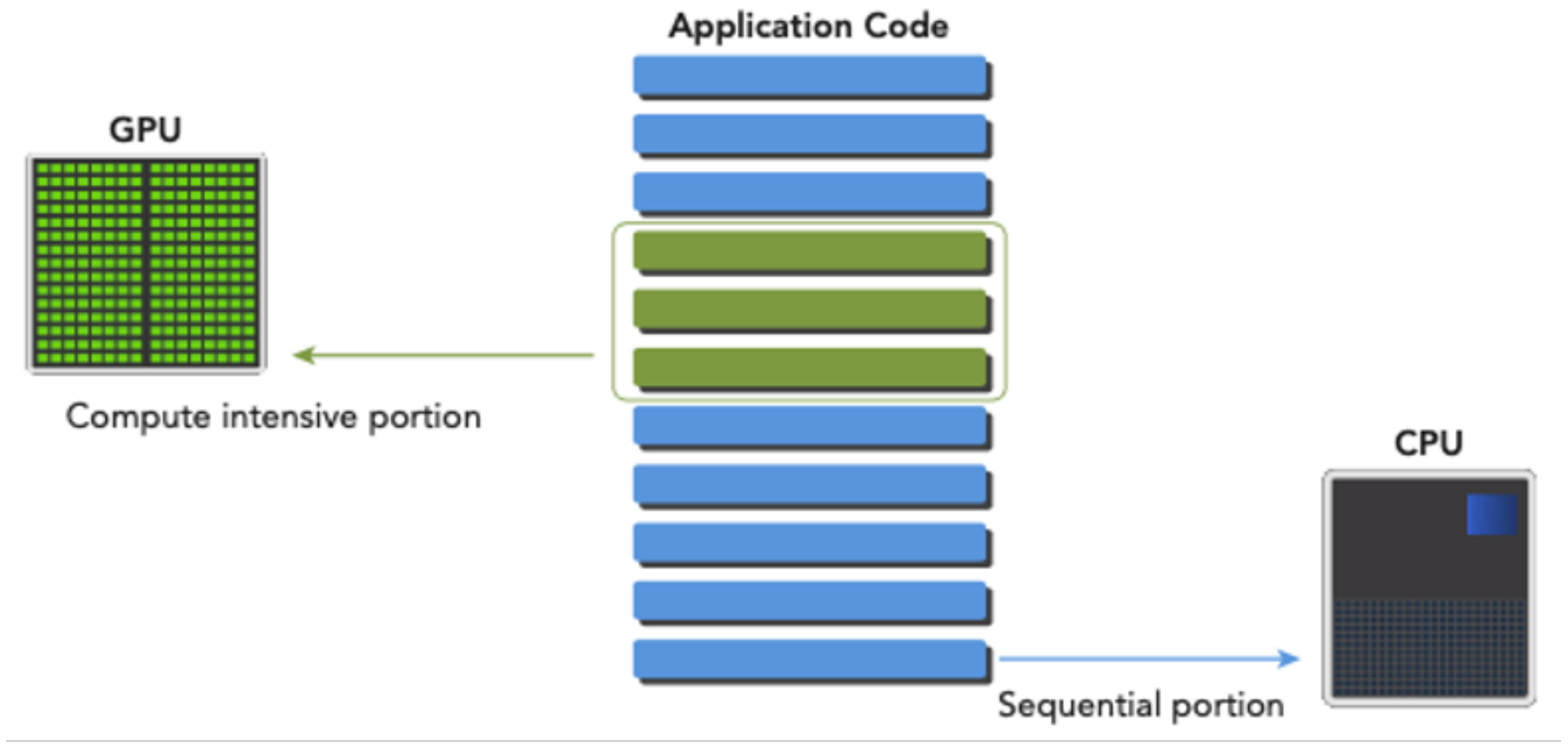
CUDA operates in a **heterogeneous computing environment**, where an application runs on two types of devices:

- **Host**: Typically the **CPU**, which runs the main application logic and controls the execution.
- **Device**: The **GPU**, which performs highly parallel tasks delegated by the host.

CUDA Program

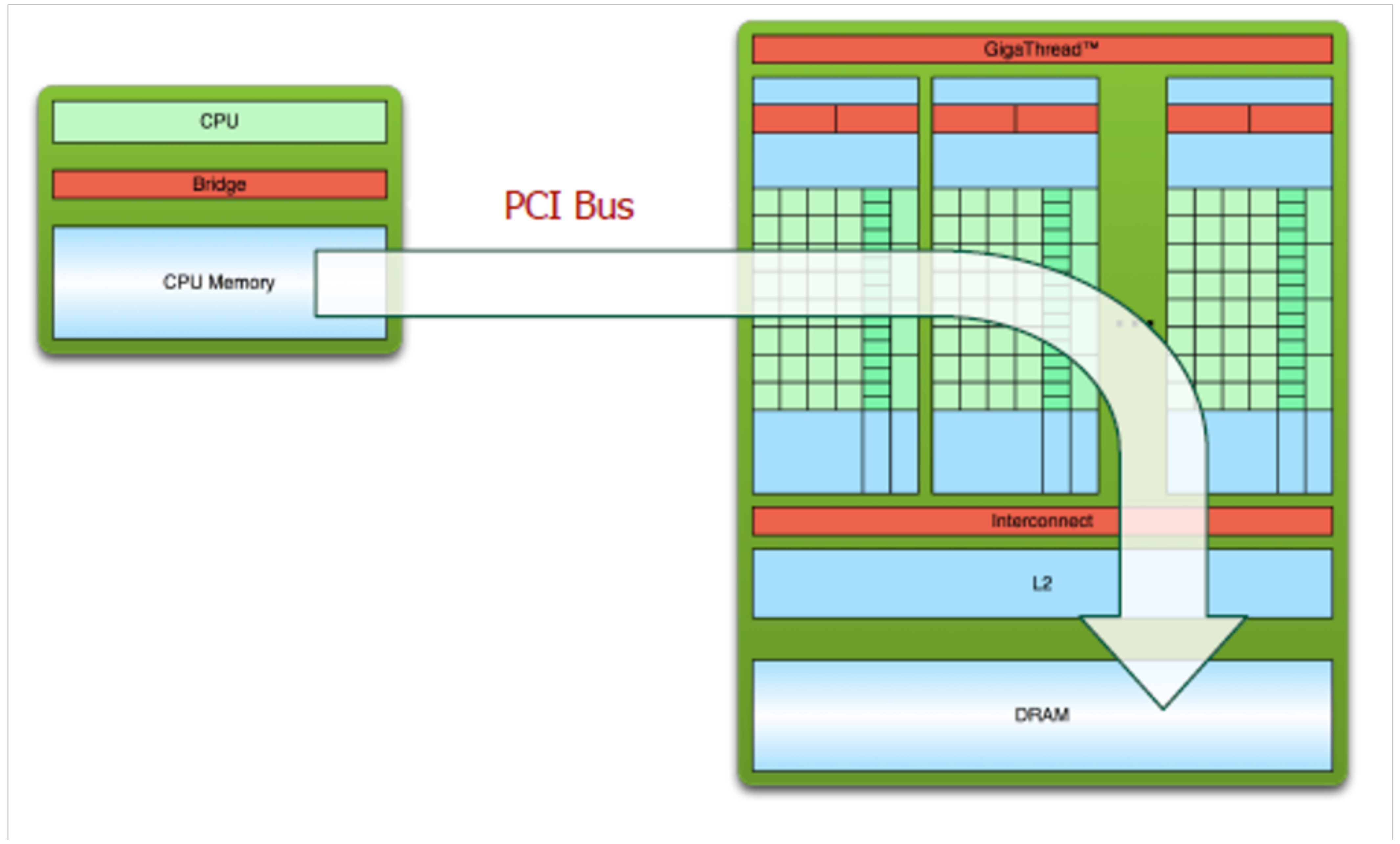
A CUDA program is organized in

- The **serial code** executed on the host (i.e., the CPU)
- One or many independent **parallelized** functions (called **kernels**) running on the **device** (i.e., the GPU)



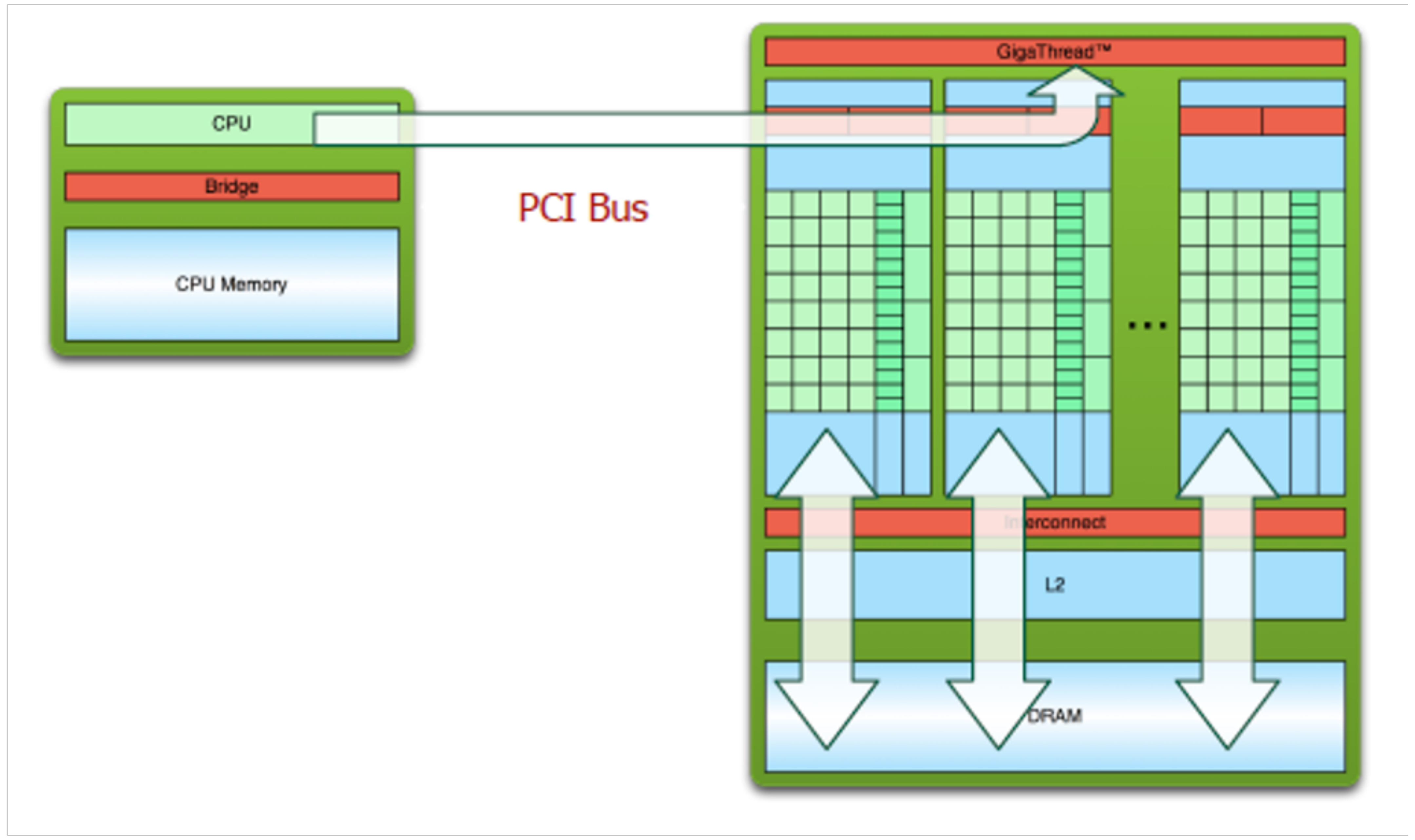
Program Execution Flow

- Transfer data from the host to the device



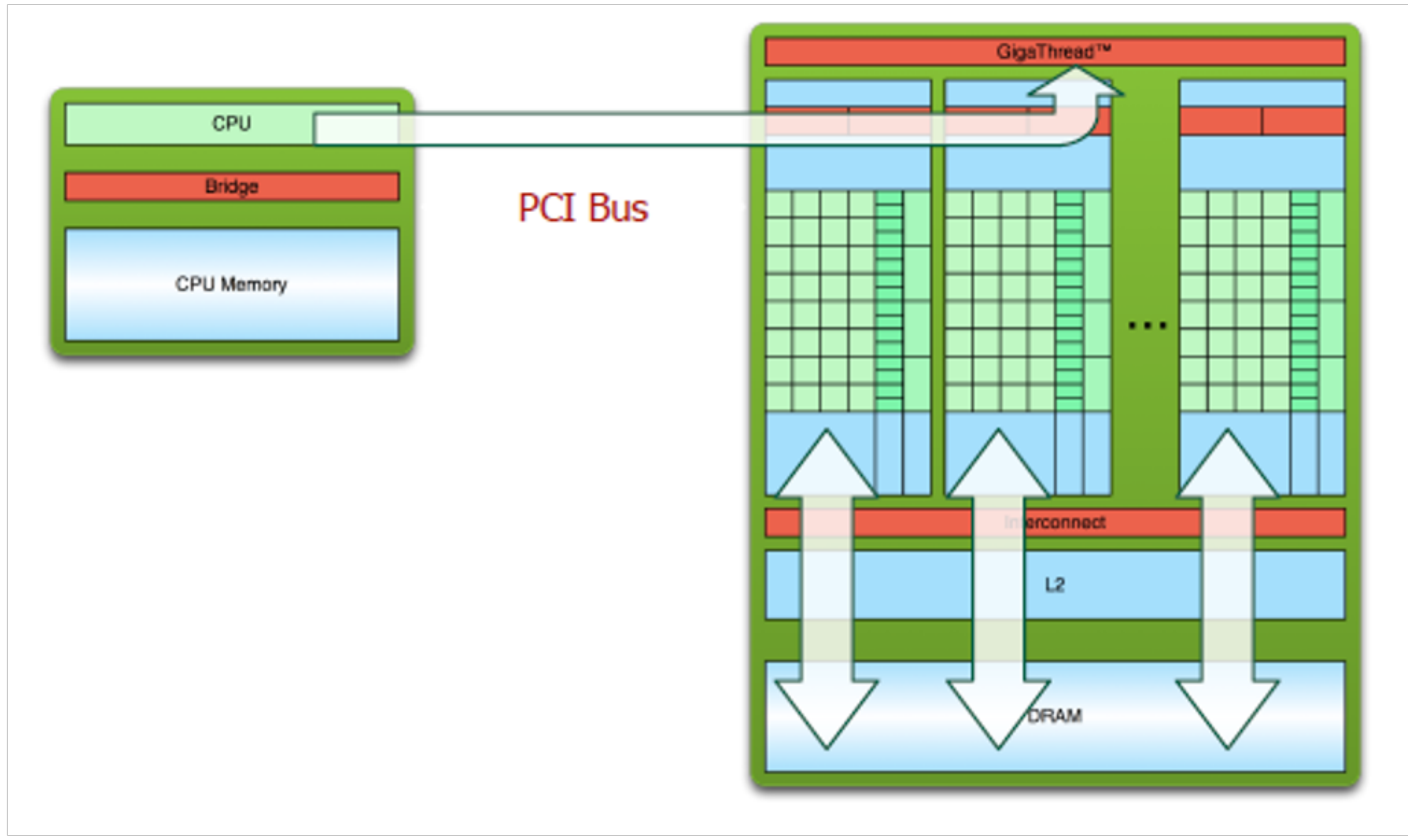
Program Execution Flow

- Transfer data from the host to the device
- Launch the kernel



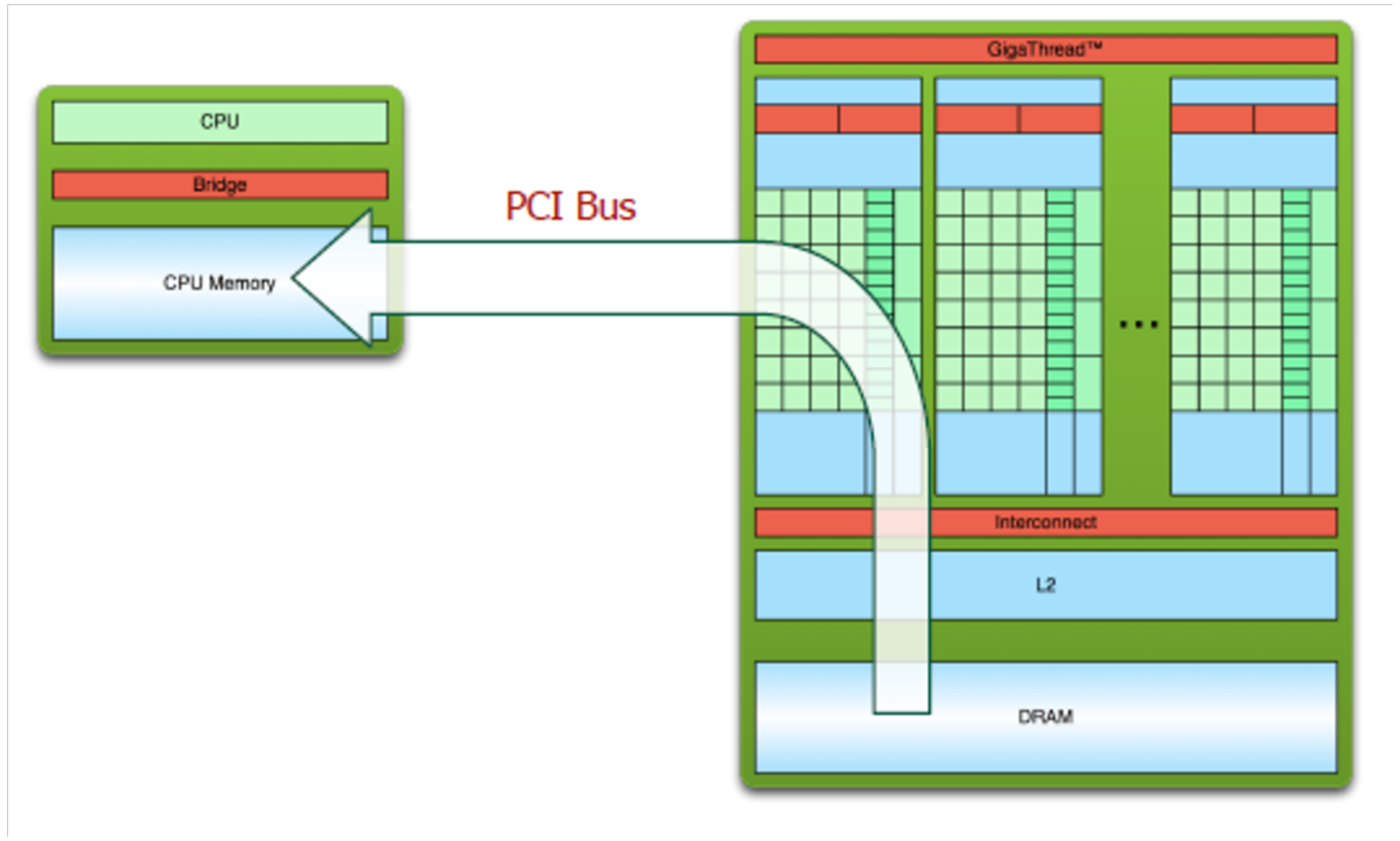
Program Execution Flow

- Transfer data from the host to the device
- Launch the kernel
- Synchronize to ensure all threads complete



Program Execution Flow

- Transfer data from the host to the device
- Launch the kernel
- Synchronize to ensure all threads complete
- Transfer results back to the host



CUDA Programming Model

It **abstracts** the **complexities of GPU hardware**, allowing developers to write code that leverages the massive parallelism provided by GPUs.

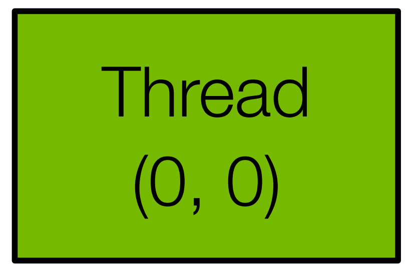
CUDA operates in a **heterogeneous computing environment**, where an application runs on two types of devices:

- **Host**: Typically the **CPU**, which runs the main application logic and controls the execution.
- **Device**: The **GPU**, $\boxed{\text{which performs highly parallel tasks delegated by the host.}}$

In the CUDA programming model,
the function running on GPU is called kernel

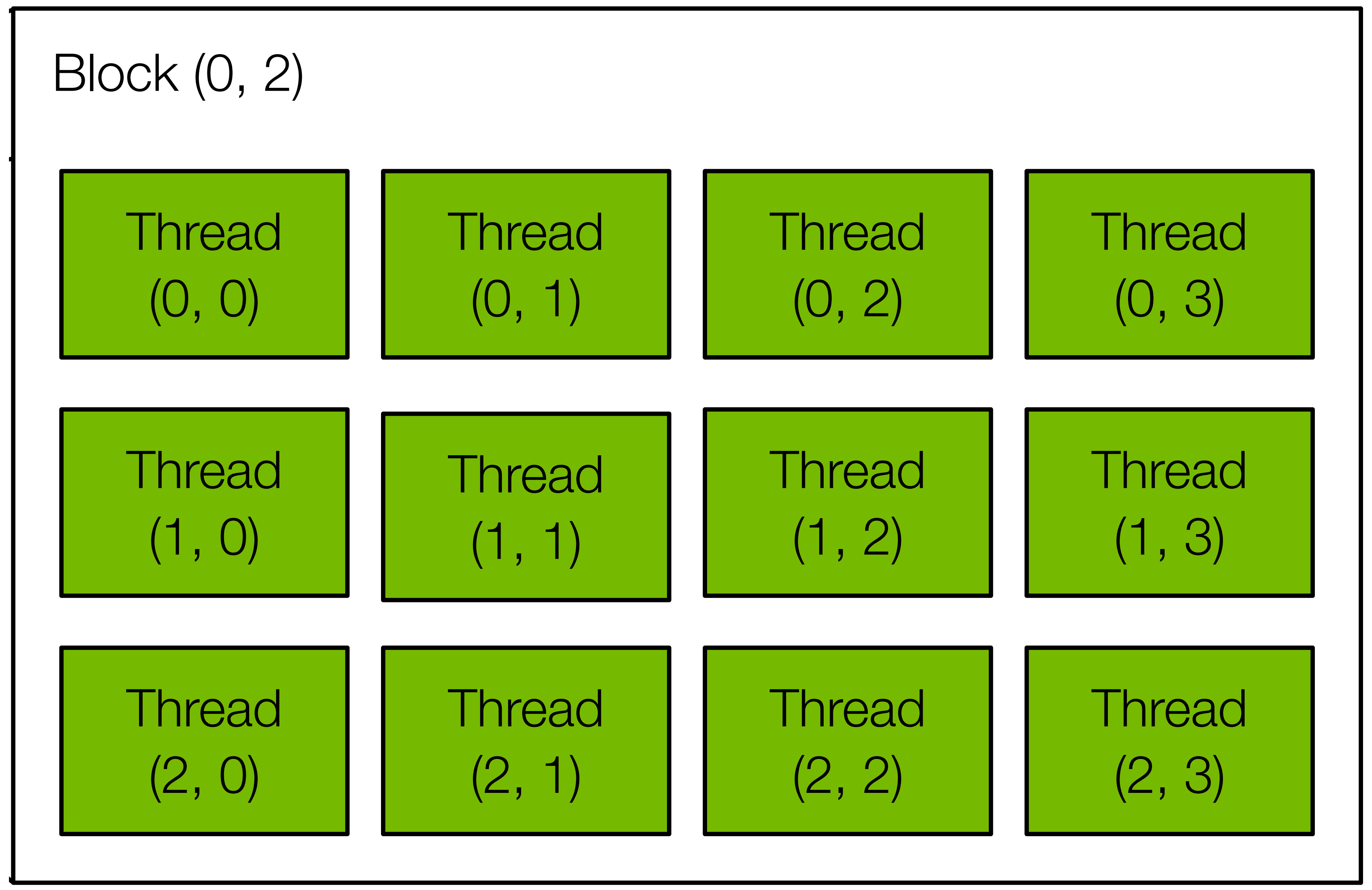
CUDA Programming Model

The **smallest unit of parallel execution** in CUDA. Each thread executes the kernel code independently and has its own set of registers and local memory.



CUDA Programming Model

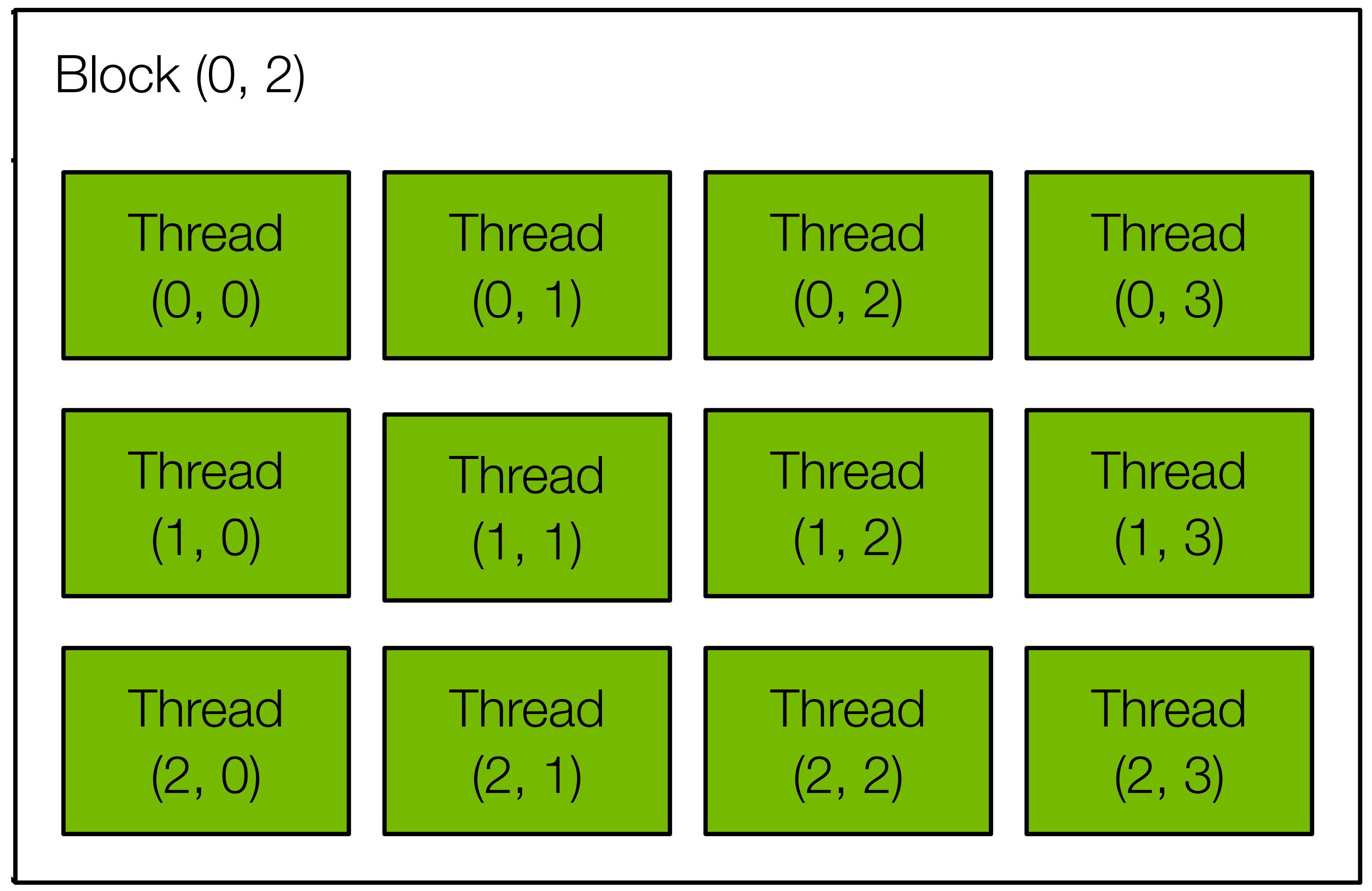
Threads are grouped into **thread blocks**. All threads in a block can communicate and synchronize on tasks. Threads in a block execute concurrently on a SM. Block dimensions can be 1D, 2D, or 3D depending on the problem at hand.



CUDA Programming Model

Threads are grouped into **thread blocks**. All threads in a block can communicate and synchronize on tasks. Threads in a block execute concurrently on a SM. Block dimensions can be 1D, 2D, or 3D depending on the problem at hand.

When a CUDA kernel is launched, threads within each thread block are organized into groups of 32 threads called **warps**. A warp **always** contains **32 threads**, regardless of the GPU architecture. All threads in a warp execute the **same instruction at the same time (SIMT paradigm)**, but each thread operates on its own data element.

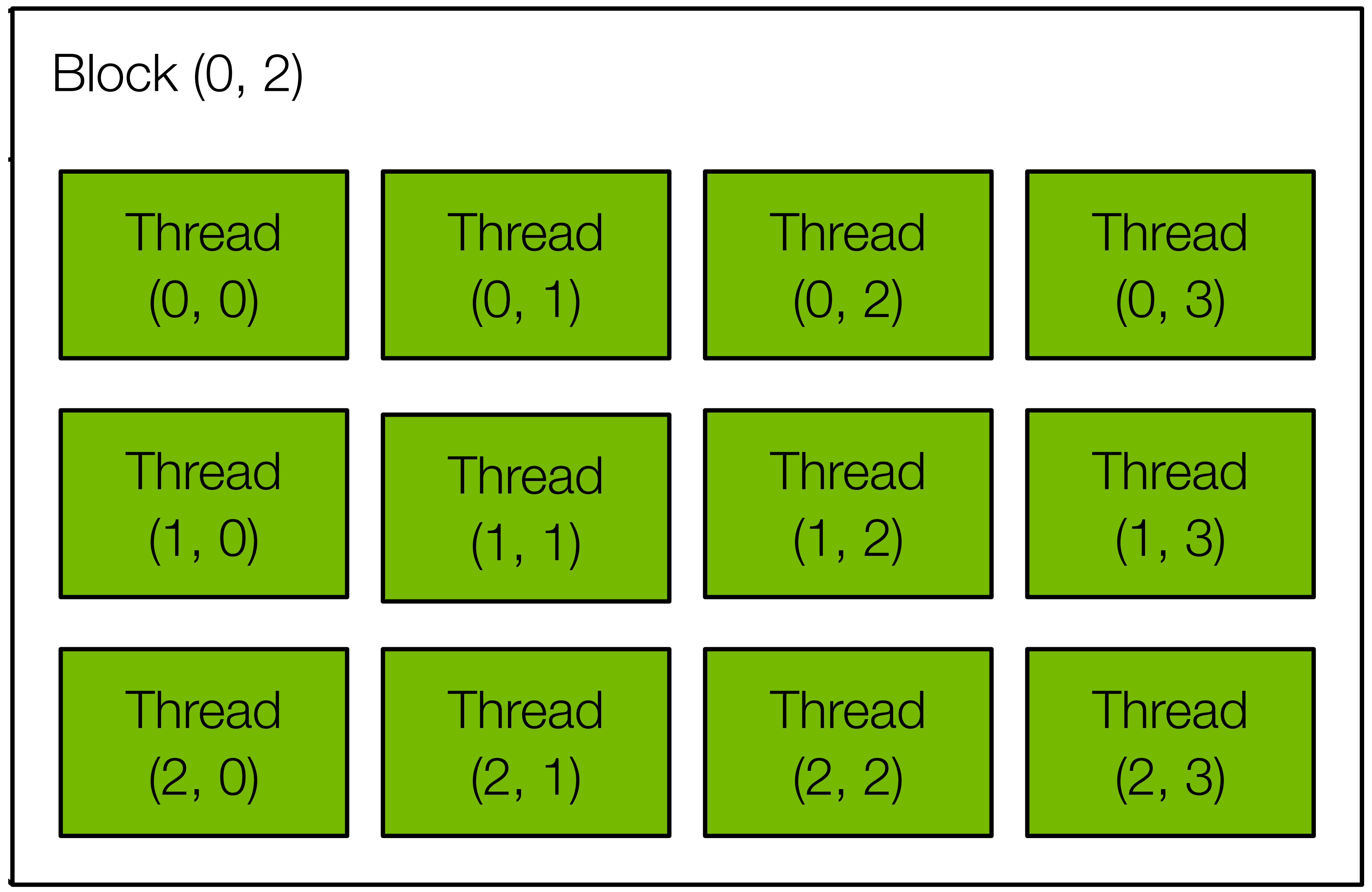


CUDA Programming Model

Threads are grouped into **thread blocks**. All threads in a block can communicate and synchronize on tasks. Threads in a block execute concurrently on a SM. Block dimensions can be 1D, 2D, or 3D depending on the problem at hand.

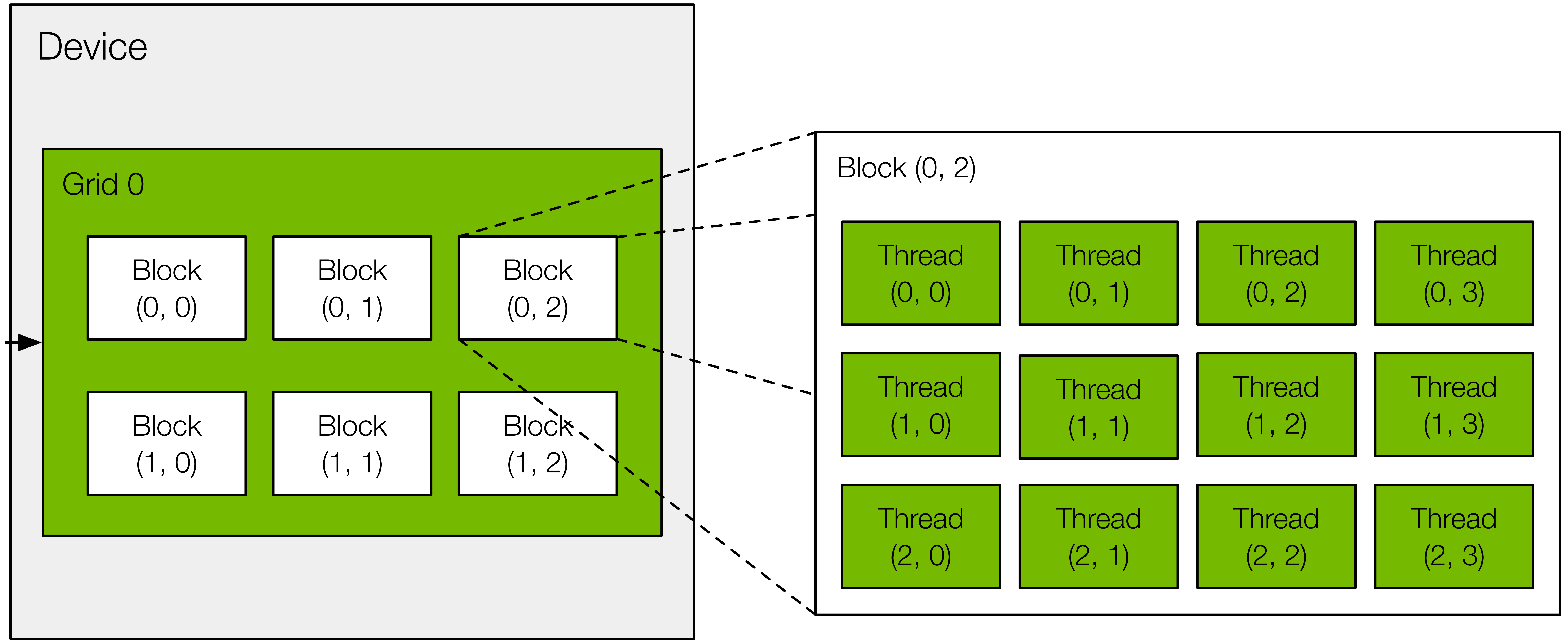
When a CUDA kernel is launched, threads within each thread block are organized into groups of 32 threads called **warps**. A warp **always** contains **32 threads**, regardless of the GPU architecture. All threads in a warp execute the **same instruction at the same time (SIMT paradigm)**, but each thread operates on its own data element.

Beware of divergence!

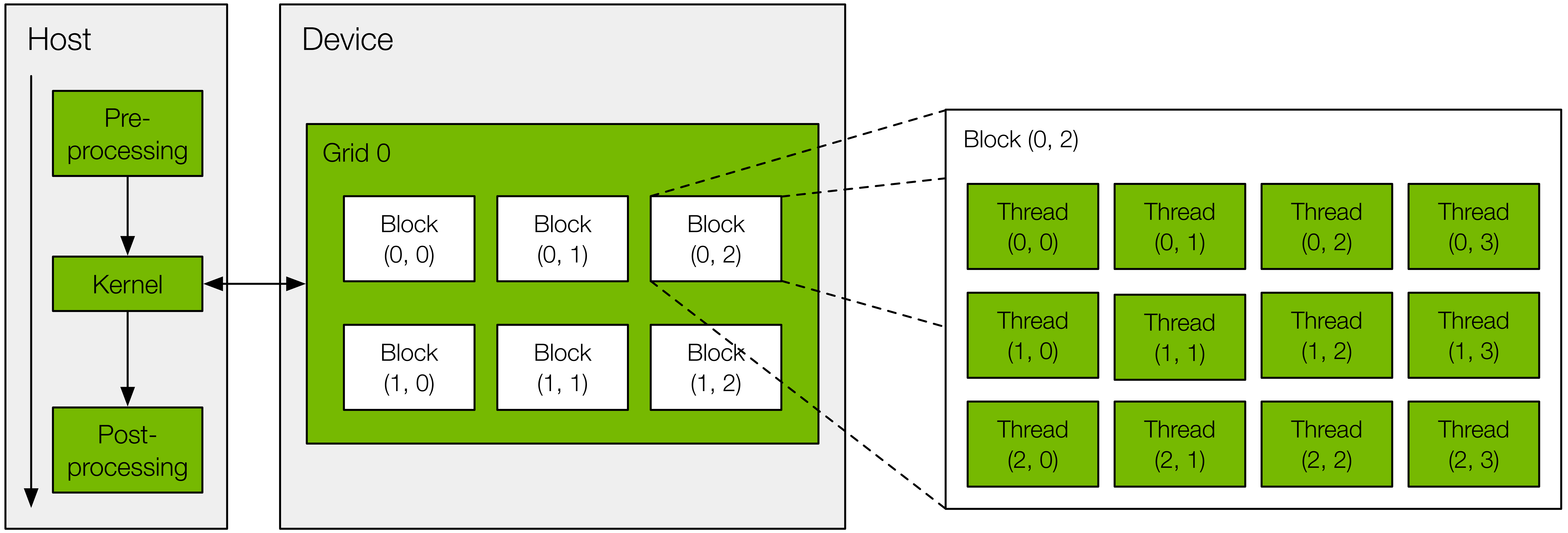


CUDA Programming Model

A collection of blocks forms a **grid**.
Blocks in the grid are independent of each other, meaning there's no direct synchronization or communication between them*.



CUDA Programming Model

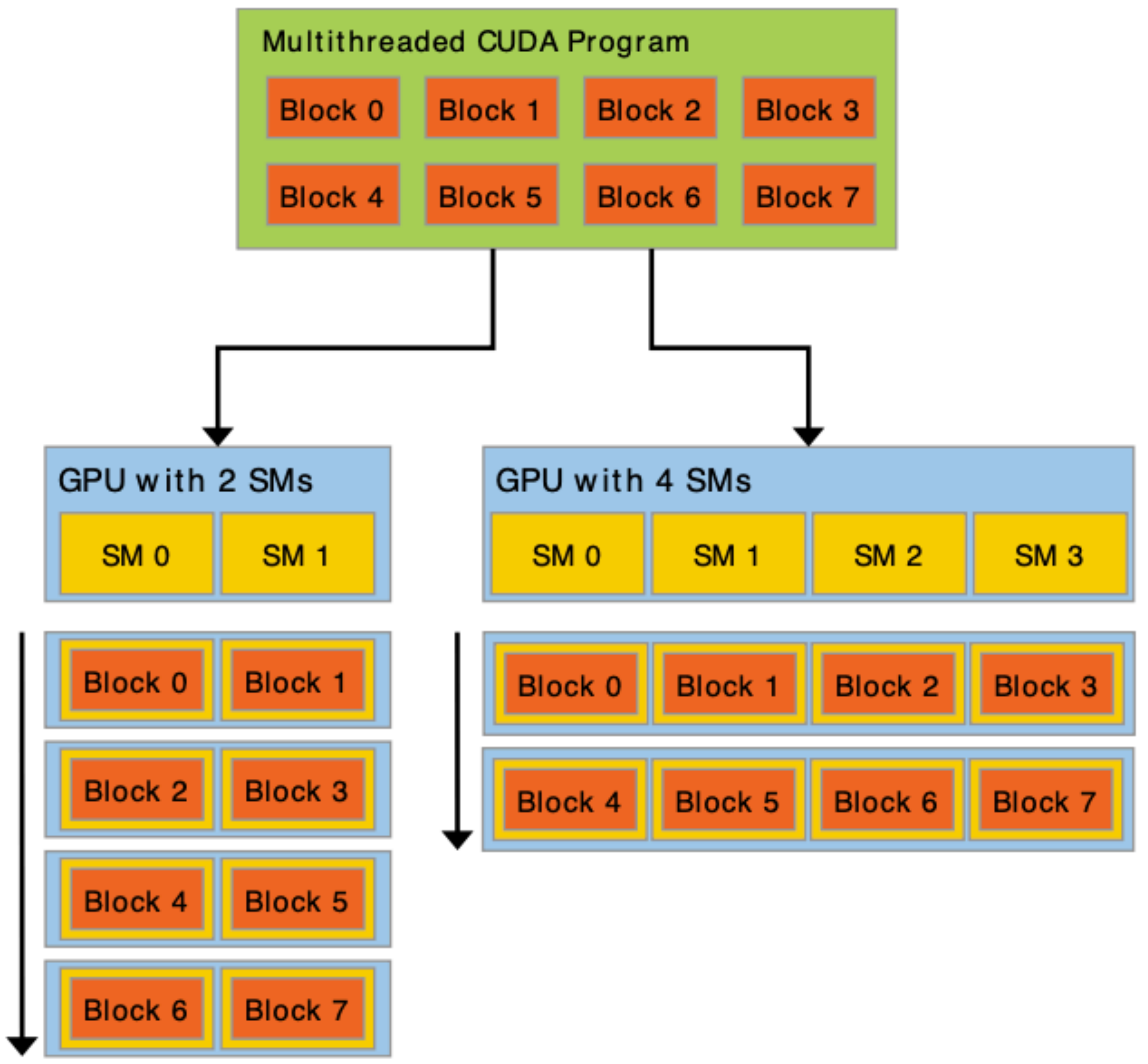


Block Dispatching

Each block is assigned to a single SM (no relocation!)

Multiple blocks may be assigned to a single SM.

Blocks are executed in **any order independently** from each other

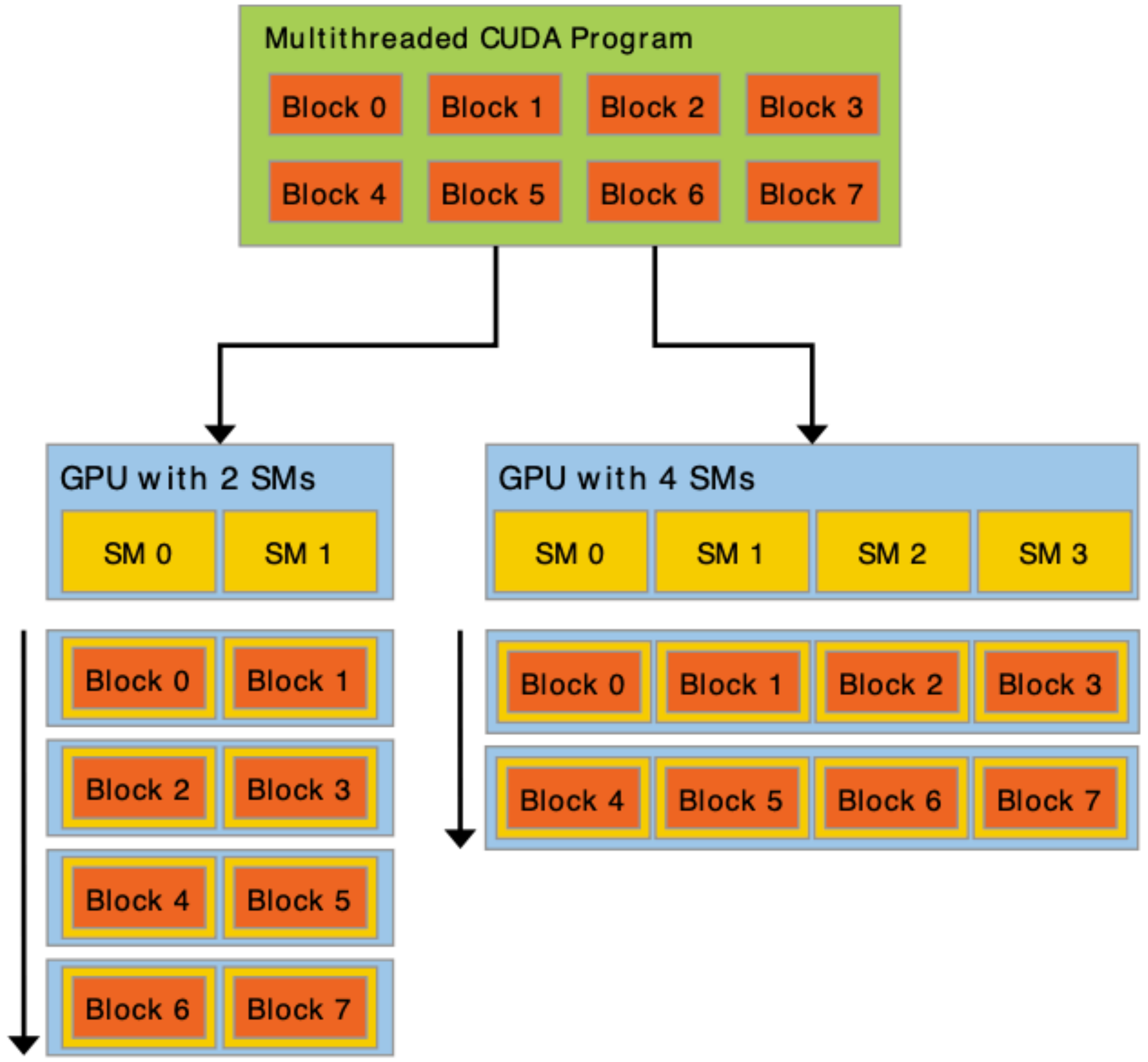


Block Dispatching

A block is assigned to an SM only **if there are enough resources** and those resources are **reserved for the block for its entire execution**

Each SM has predefined resources:

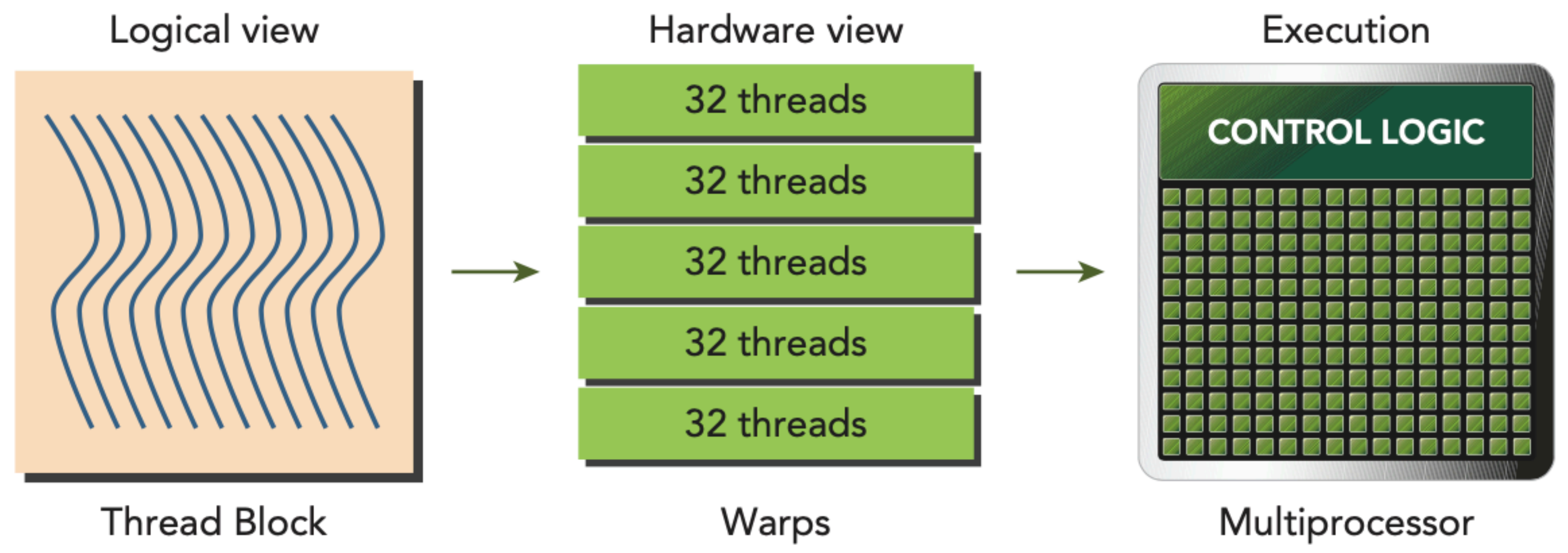
- max number of threads
- max number of blocks
- number of registers
- shared memory size



Warp Scheduling

The SM divides the block in **warps** (32 threads each). A warp is executed with a **SIMT** approach: all threads in a warp execute the same instruction.

To improve throughput, the scheduler interleaves the execution of multiple warps.



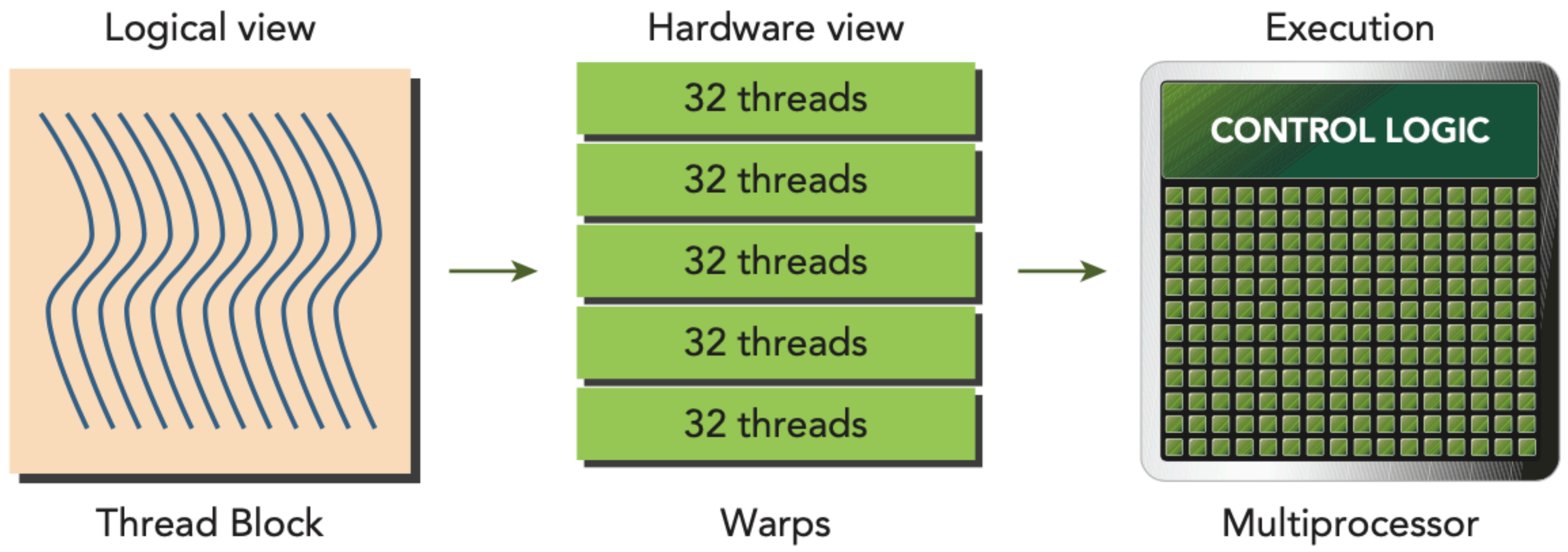
If block size is not multiple of 32, **inactive threads are added to the last warp**

Warp Scheduling

Each active warp may be classified as:

- **Selected warp** – if currently executed
- **Stalled warp** – if not ready to be executed
- **Eligible warp** – if ready to be executed

Warp schedulers will select from **eligible** warps



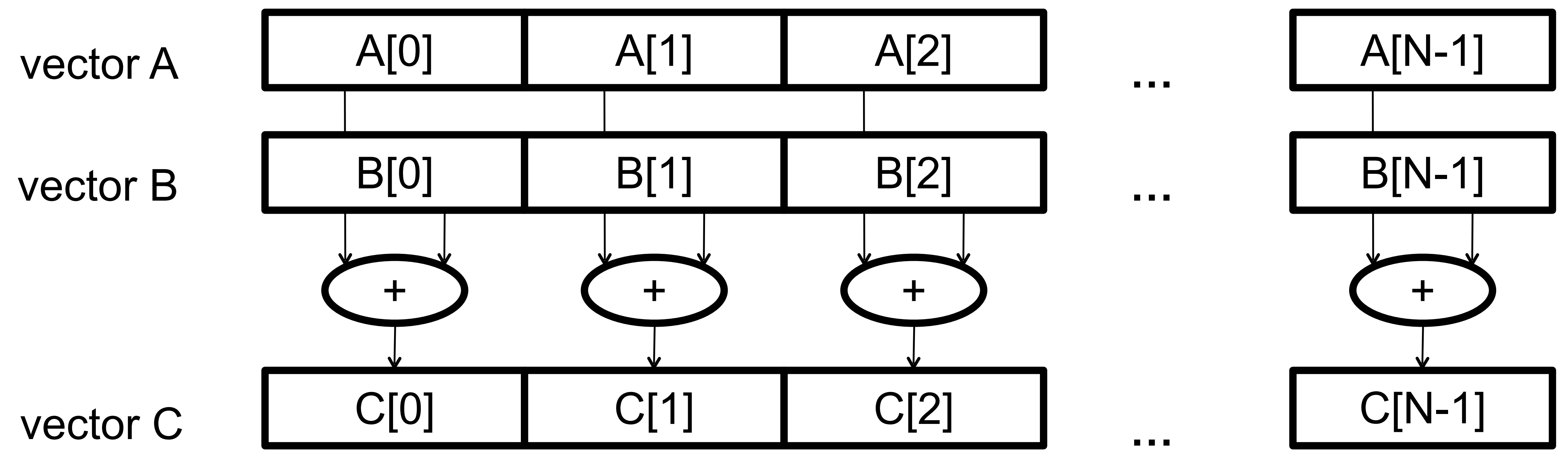
Vector Addition Example

C program computing the sum of two integer vectors:

```
#define N 1024

void vsum(int* a, int* b, int* c){
  int i;
  for (i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}

int main(){
  int va[N], vb[N], vc[N];
  /*... input data acquisition ...*/
  vsum(va, vb, vc);
  /*... output data visualization ...*/
}
```



Structure of a CUDA Program

```
#include <cuda_runtime.h>
#define N 1024

/* kernel function*/

int main(){
  int h_va[N], h_vb[N], h_vc[N];
  /*... input data acquisition ...*/
  /* device memory allocation */
  /* CPU->GPU data transmission */
  /* kernel launch */
  /* GPU->CPU data transmission */
  /* device memory freeing */
  /*... output data visualization ...*/
}
```

Structure of a CUDA Program

```
#include <cuda_runtime.h>
#define N 1024

/* kernel function*/

int main(){
  int h_va[N], h_vb[N], h_vc[N];
  /*... input data acquisition ...*/
  /* device memory allocation */
  /* CPU->GPU data transmission */
  /* kernel launch */
  /* GPU->CPU data transmission */
  /* device memory freeing */
  /*... output data visualization ...*/
}
```

Device code

Host code

- It is a C/C++ program with CUDA extensions
- The kernel function has some restrictions
  - Access to device memory only
  - `void` return type
  - Asynchronous behavior
  - No variable number of parameters
  - No static variables
  - No function pointers

Kernel Function

```
#include <cuda_runtime.h>
#define N 1024


__global__ void vsumKernel(
                int* a,
                int* b,
                int *c)
{
  int i = blockIdx.x * blockDim.x
         + threadIdx.x;

  c[i] = a[i] + b[i];
}
```

Kernel Function

```
#include <cuda_runtime.h>
#define N 1024


__global__ void vsumKernel(
                int* a,
                int* b,
                int *c)
{
  int i = blockIdx.x * blockDim.x
           + threadIdx.x;

  c[i] = a[i] + b[i];
}
```

Each GPU function has a
function qualifier

	Executed on the:	Only callable from the:
`__device__`	device	device
`__global__`	device	host
`__host__`	host	host

Kernel Function

```
#include <cuda_runtime.h>
#define N 1024


__global__ void vsumKernel(
                 int* a,
                 int* b,
                 int *c)
{
  int i = blockIdx.x * blockDim.x
         + threadIdx.x;

  c[i] = a[i] + b[i];
}
```

Input to the __global__ kernel are **pointers to the device memory**

- Host and device codes have **separate scopes**
- The memory locations pointed by the parameters are accessed by **all the threads**

Kernel Function

```
#include <cuda_runtime.h>
#define N 1024


__global__ void vsumKernel(
                int* a,
                int* b,
                int *c)
{
  int i = blockIdx.x * blockDim.x
          + threadIdx.x;

  c[i] = a[i] + b[i];
}
```

Local variables are
per-thread private

Kernel Function

```
#include <cuda_runtime.h>
#define N 1024


__global__ void vsumKernel(
                int* a,
                int* b,
                int *c)
{
  int i = blockIdx.x * blockDim.x
          + threadIdx.x;

  c[i] = a[i] + b[i];
}
```

- **blockIdx**: id of the block in the grid
- **blockDim**: size of the block
- **threadIdx**: id of the thread in the block
- **gridDim**: size of the grid

CUDA built-in struct variables with three fields: x, y, z (3D grid!)

Kernel Function

```
#include <cuda_runtime.h>
#define N 1024


__global__ void vsumKernel(
                int* a,
                int* b,
                int *c)
{
  int i = blockIdx.x * blockDim.x
         + threadIdx.x;

  c[i] = a[i] + b[i];
}
```

- **Mapping of each thread to the single data element**
- **Elaboration on the data element**

Device Memory Allocation

```
#include <cuda_runtime.h>
#define N 1024

int main(){
  int h_va[N], h_vb[N], h_vc[N];
  int *d_va, *d_vb, *d_vc;
  /* device memory allocation */
  cudaMalloc(&d_va, N*sizeof(int));
  cudaMalloc(&d_vb, N*sizeof(int));
  cudaMalloc(&d_vc, N*sizeof(int));
  /* ... */
}
```

Pointer storing the address of the allocated memory

Size in bytes to be allocated

Device memory has to be allocated similarly to host memory (heap)

Device memory is separate from the host one!
Pointers to device memory CANNOT be used in the host program

Data Transfer to the Device

```
int main(){
  /*...*/
  /* CPU->GPU data transmission */
  cudaMemcpy(d_va, h_va, N*sizeof(int),
                 cudaMemcpyHostToDevice);
  cudaMemcpy(d_vb, h_vb, N*sizeof(int), cudaMemcpyHostToDevice);
  /* ... */
}
```

Data must be transferred explicitly to the device memory

Destination memory

Source memory

Size in bytes to be transmitted

Transmission direction

- `cudaMemcpyDeviceToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToDevice`
- `cudaMemcpyHostToHost`

Kernel Launch

```
/*...*/
int main(){
  /*...*/
  /* kernel launch */
  dim3 blocksPerGrid(N/256, 1, 1);
  dim3 threadsPerBlock(256, 1, 1);
  vsumKernel<<<blocksPerGrid, threadsPerBlock>>>(d_va, d_vb, d_vc);
  /* ... */
}
```

Execution configuration parameters: block and grid dimension (dim3 is a struct with three fields: x, y, z)

Kernel Launch

```
/*...*/
int main(){
  /*...*/
  /* kernel launch */
  dim3 blocksPerGrid(N/256, 1, 1);
  dim3 threadsPerBlock(256, 1, 1);
  vsumKernel<<<blocksPerGrid, threadsPerBlock>>>(d_va, d_vb, d_vc);
  /* ... */
}
```

The parameters are the pointers to device memory

Data Transfer to the Host

```
/*...*/


int main(){
  /*...*/
  /* GPU->CPU data transmission */
  cudaMemcpy(h_vc, d_vc, N*sizeof(int), cudaMemcpyDeviceToHost);
  /* ... */
}
```

Data must be transferred explicitly to the host memory

Destination memory

Source memory

Size in bytes to be transmitted

Transmission direction

Device Memory Freeing

```
/*...*/


int main(){
  /*...*/
  /* device memory freeing */
  cudaFree(d_va);
  cudaFree(d_vb);
  cudaFree(d_vc);
  /* ... */
}
```

Device memory has to be **explicitly released** by the application via cudaFree()

`cudaDeviceReset()` may be called to reset all resources of the device used by the process

Complete Code

```
#include <cuda_runtime.h>
#define N 1024

__global__ void vsumKernel(int* a, int* b, int *c){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}

int main(){
  int h_va[N], h_vb[N], h_vc[N];
  int *d_va, *d_vb, *d_vc;

  /*... input data acquisition ...*/

  cudaMalloc(&d_va, N*sizeof(int));
  cudaMalloc(&d_vb, N*sizeof(int));
  cudaMalloc(&d_vc, N*sizeof(int));

  cudaMemcpy(d_va, h_va, N*sizeof(int),
                  cudaMemcpyHostToDevice);
  cudaMemcpy(d_vb, h_vb, N*sizeof(int),
             cudaMemcpyHostToDevice);

  dim3 blocksPerGrid(N/256, 1, 1);
  dim3 threadsPerBlock(256, 1, 1);
  vsumKernel<<<blocksPerGrid, threadsPerBlock>>>
              (d_va, d_vb, d_vc);

  cudaMemcpy(h_vc, d_vc, N*sizeof(int),
             cudaMemcpyDeviceToHost);

  cudaFree(d_va);
  cudaFree(d_vb);
  cudaFree(d_vc);

  /*... output data visualization ...*/
  return 0;
}
```

CUDA

From an application perspective, CUDA defines:

- **Architecture model**
    - with many processing cores grouped in multiprocessors who share a SIMT control unit
- **Programming model**
    - based on **massive data parallelism** and fine-grained parallelism
    - **scalable**: the code is executed on a different number of cores without recompiling it
- **Memory model**
    - more **explicit** to the programmer, where **caches are not transparent** anymore

CUDA Memory Model

From an application perspective, CUDA defines:

- **Memory model**
    - more **explicit** to the programmer, where **caches are not transparent** anymore

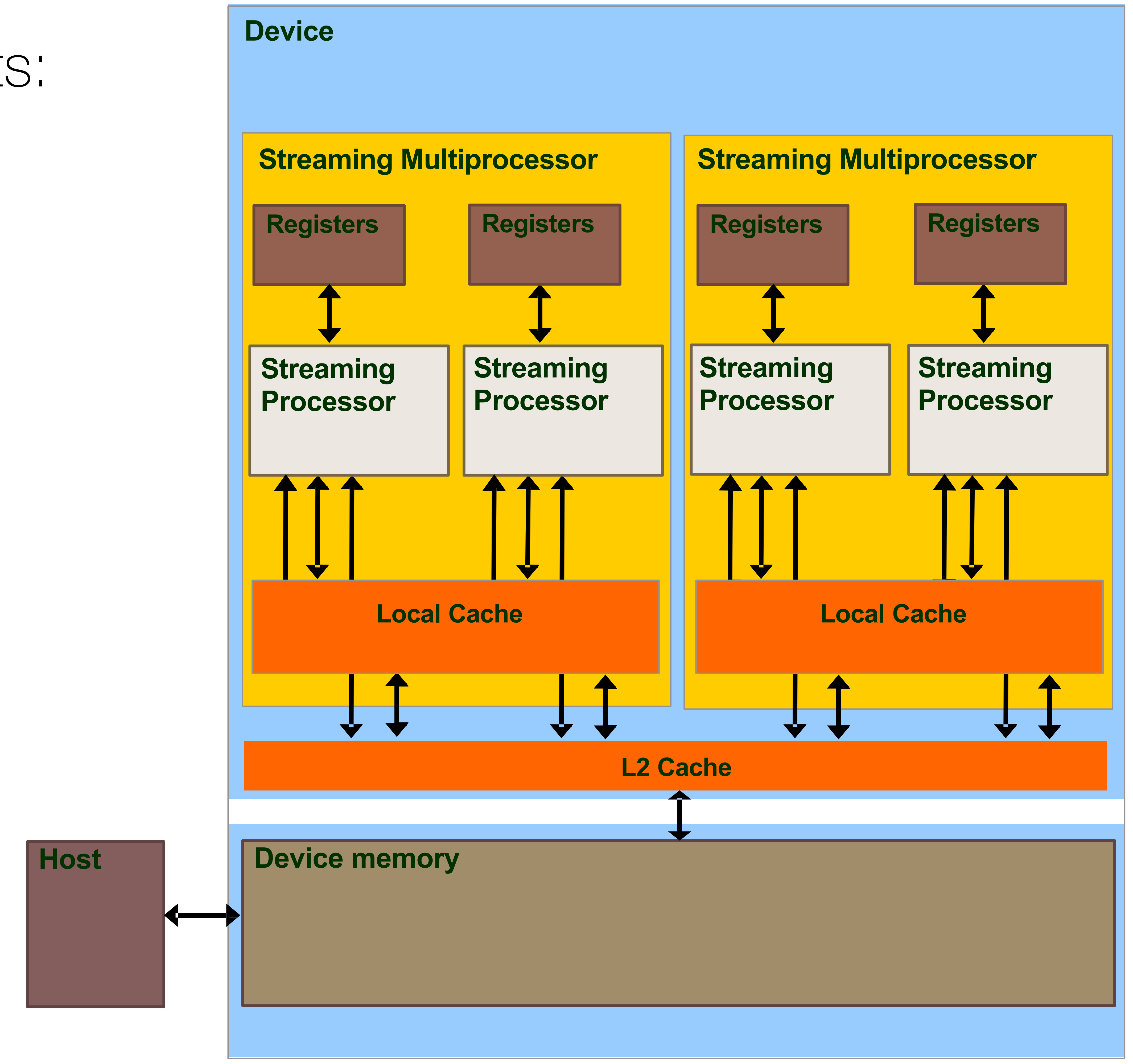
The **CUDA memory model** defines various types of **memory** available in a CUDA-enabled GPU, and it specifies how data is shared and accessed between threads, thread blocks, and the host (CPU).

This memory hierarchy is designed to **optimize** the **performance** and **manage** the **bandwidth** constraints of GPU-based parallel processing.

GPU Memory Hierarchy

GPU memory hierarchy is composed of three elements:

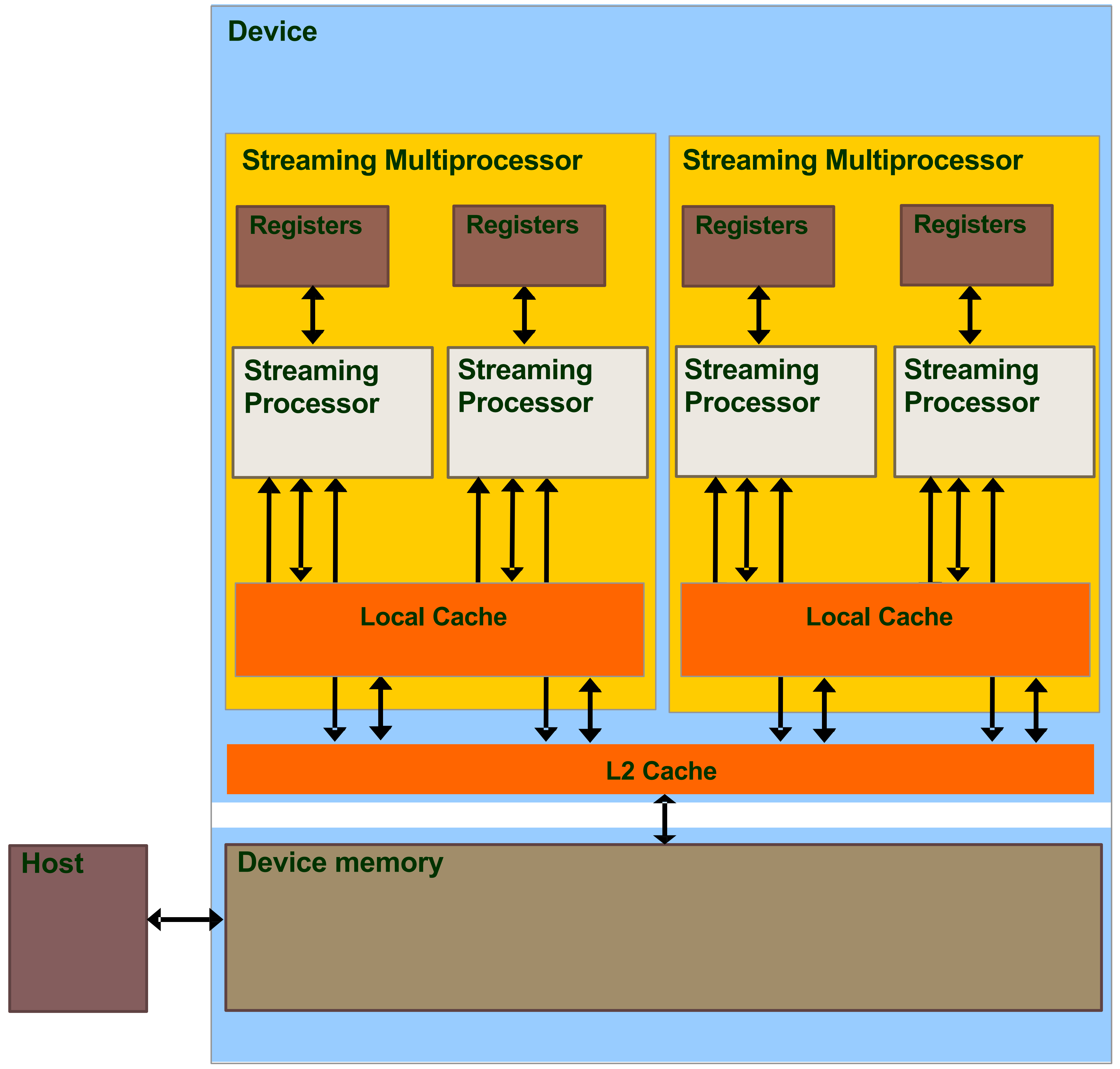
1. **Registers**
2. **Caches**
3. **Device memory**



GPU Memory Hierarchy

Registers:

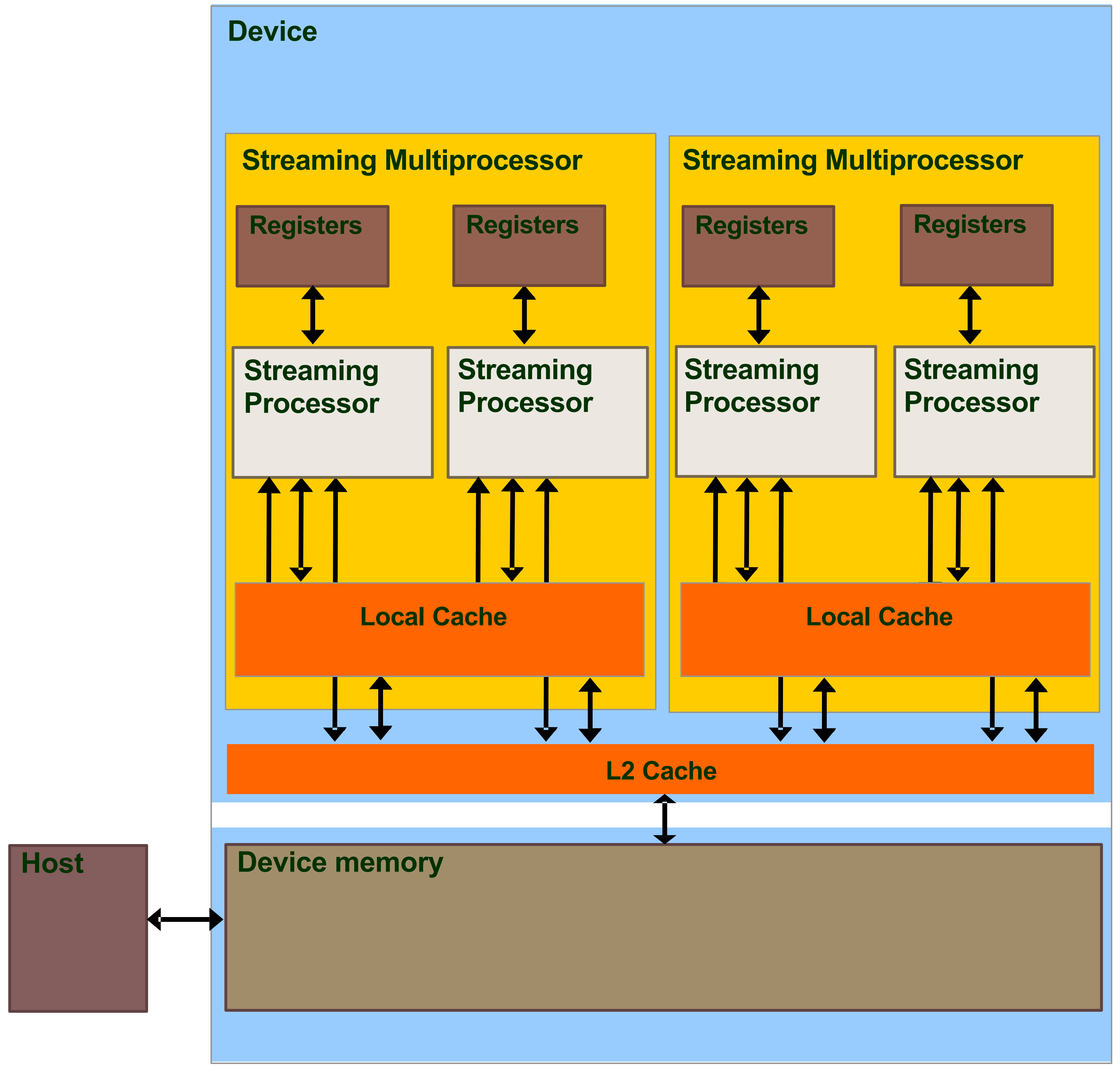
- **Private** register file within each SM
- Partitioned among threads running in the SM
- Not enough to store all private data of each thread



GPU Memory Hierarchy

Caches:

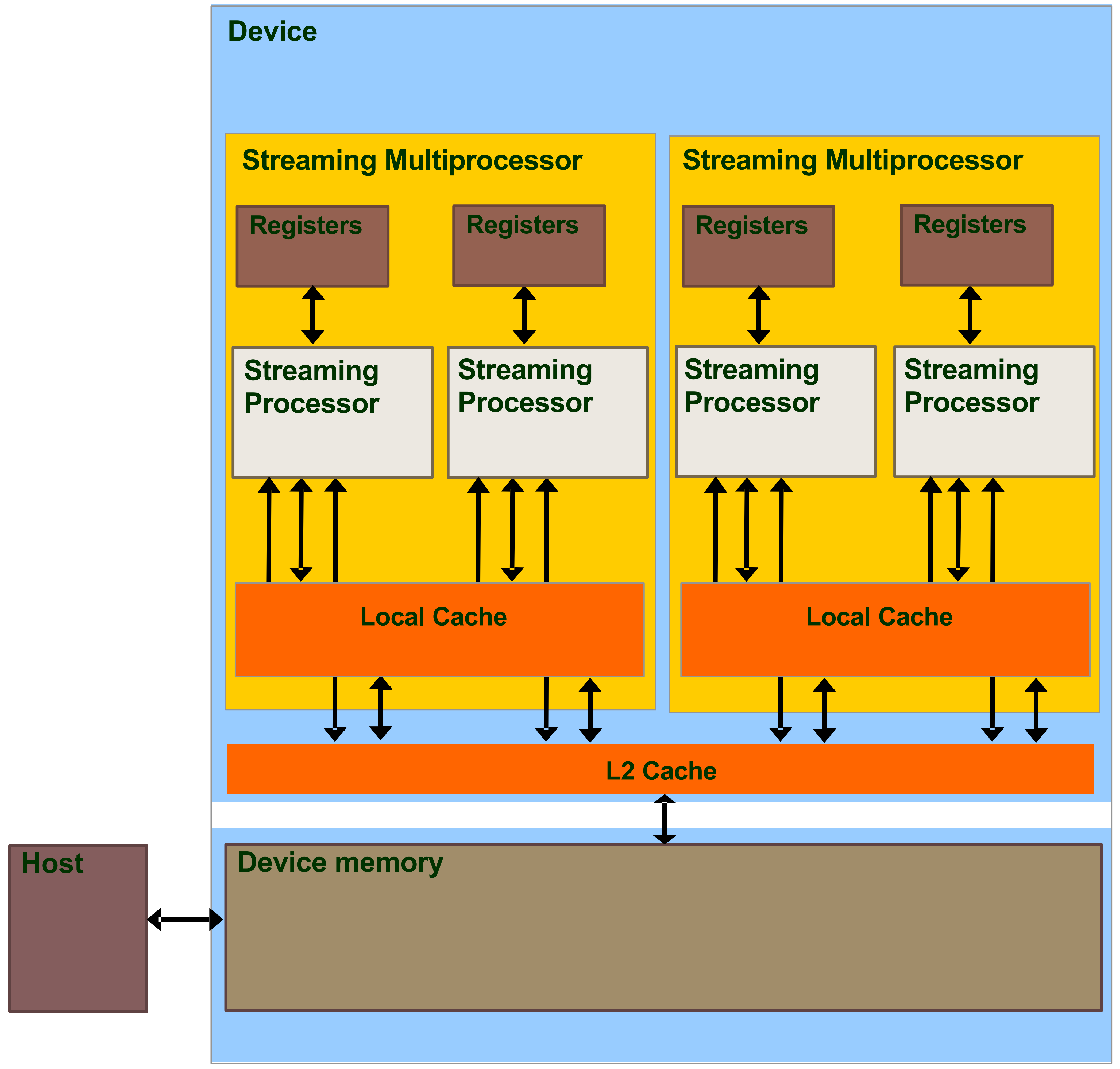
- Global **L2** cache
- Per-SM **L1** cache
- Per-SM **constant** cache
- Per-SM **read-only/texture** cache
- Per-SM **shared** memory
- Access depends on the type of cache



GPU Memory Hierarchy

Device memory:

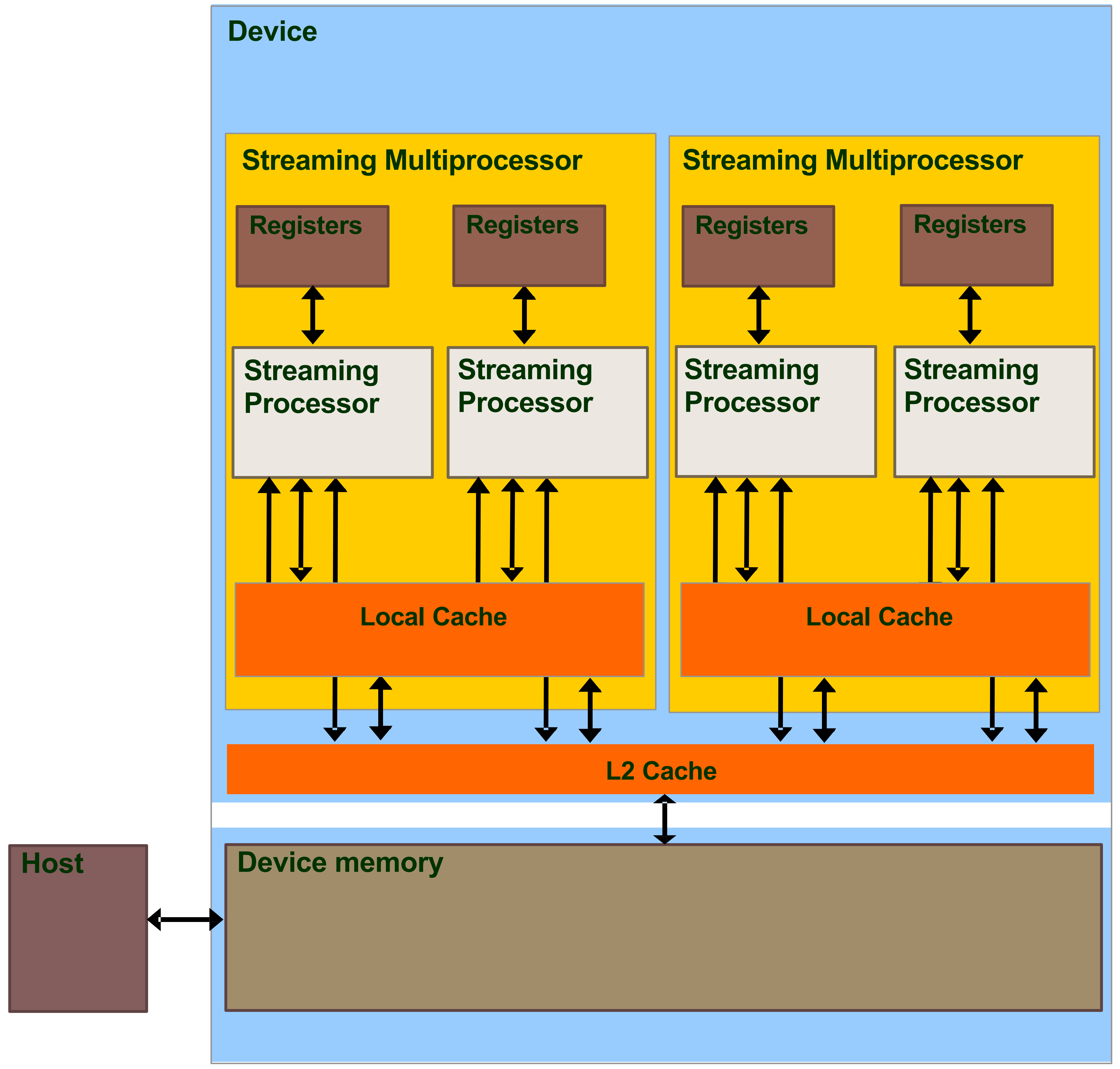
- **Off-chip** memory
- Accessible by all threads on SMs
- Accessible by the host



GPU Memory Hierarchy

Memory latencies:

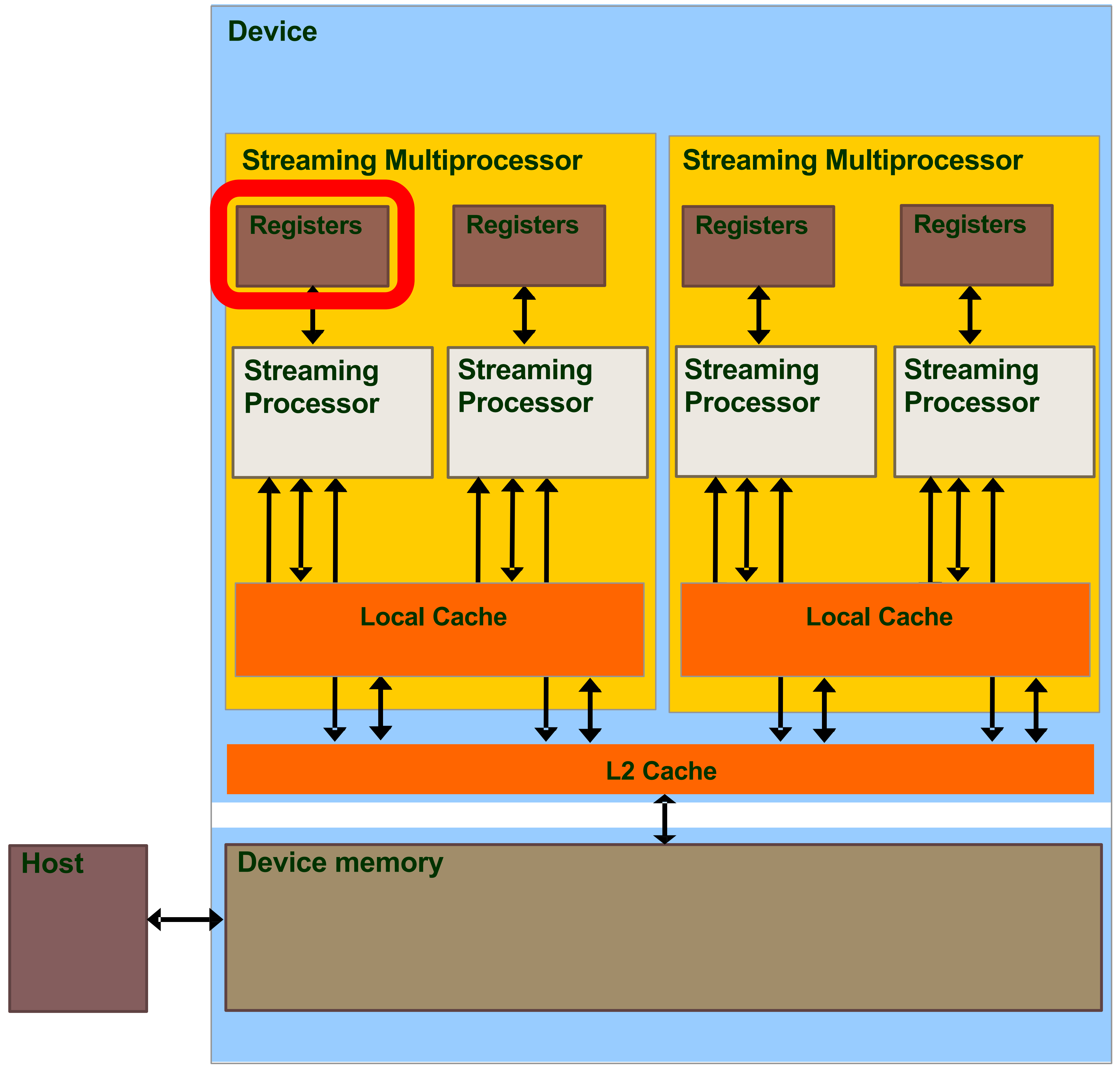
	Cost (cycles)
Registers	1
Device memory	200-800
Shared memory	~1
L1	1
Constant cache	~1
Read-only	1



CUDA Memory Model

Key components:

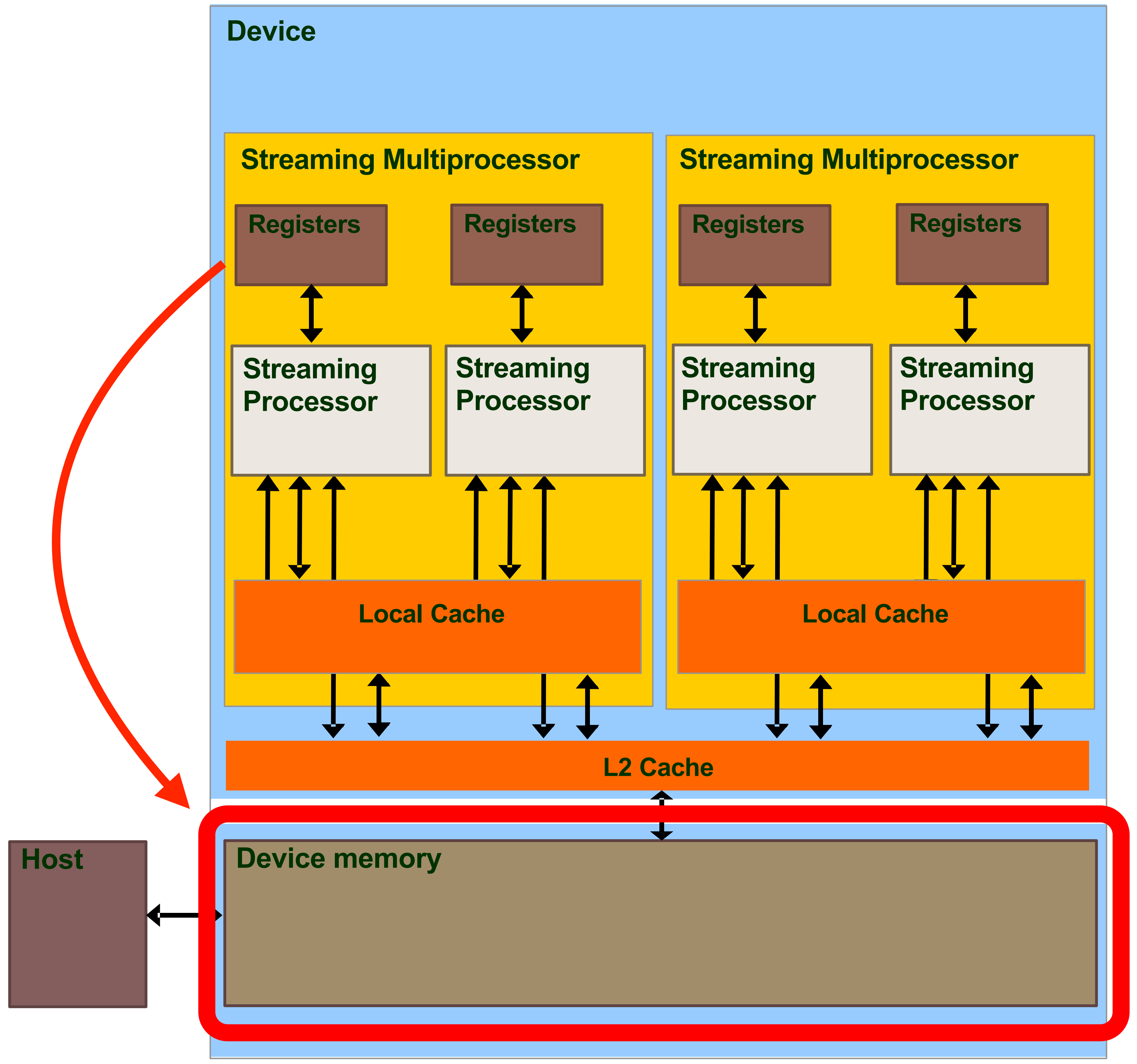
- **Registers (per-thread)**: fastest form of memory and are private to each thread, they store local variables used within a thread, making access almost instantaneous. The number of registers per thread is limited, and excessive register usage can lead to **register spilling** into slower local memory (see next slide).



CUDA Memory Model

Key components:

- **Registers (per-thread)**: fastest form of memory and are private to each thread, they store local variables used within a thread, making access almost instantaneous. The number of registers per thread is limited, and excessive register usage can lead to **register spilling** into slower local memory.
- **Local memory (per-thread):** resides in device memory rather than being on-chip. It's used when there are not enough registers for all thread-local data. Local memory is primarily accessed when *register spilling* occurs, which should be minimized for performance reasons.



CUDA Memory Model

Key components:

- **Registers (per-thread)**: fastest form of memory and are private to each thread, they store local variables used within a thread, making access almost instantaneous. The number of registers per thread is limited, and excessive register usage can lead to **register spilling** into slower local memory.
- **Local memory (per-thread):** resides in device memory rather than being on-chip. It's used when there are not enough registers for all thread-local data. Local memory is primarily accessed when *register spilling* occurs, which should be minimized for performance reasons.

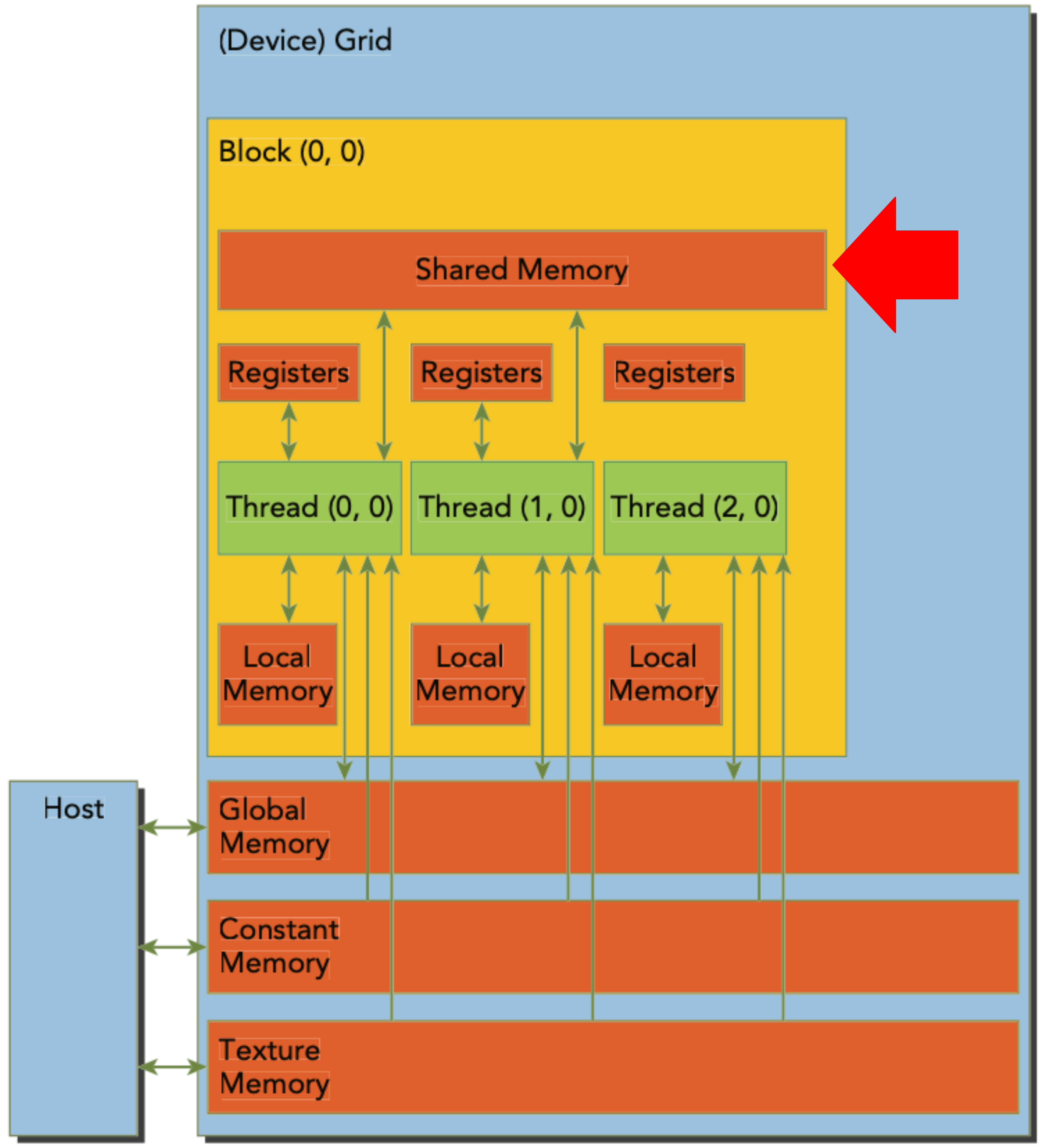
- **Shared memory (per-block)**

Shared Memory

The shared memory is a very fast **scratchpad** memory integrated in the SM (**L1 cache**)

- Intra-block data sharing and thread cooperation
- Reduce accesses to the global memory

Together with registers, shared memory represents a constraint on the number of blocks assignable to a SM



Shared Memory

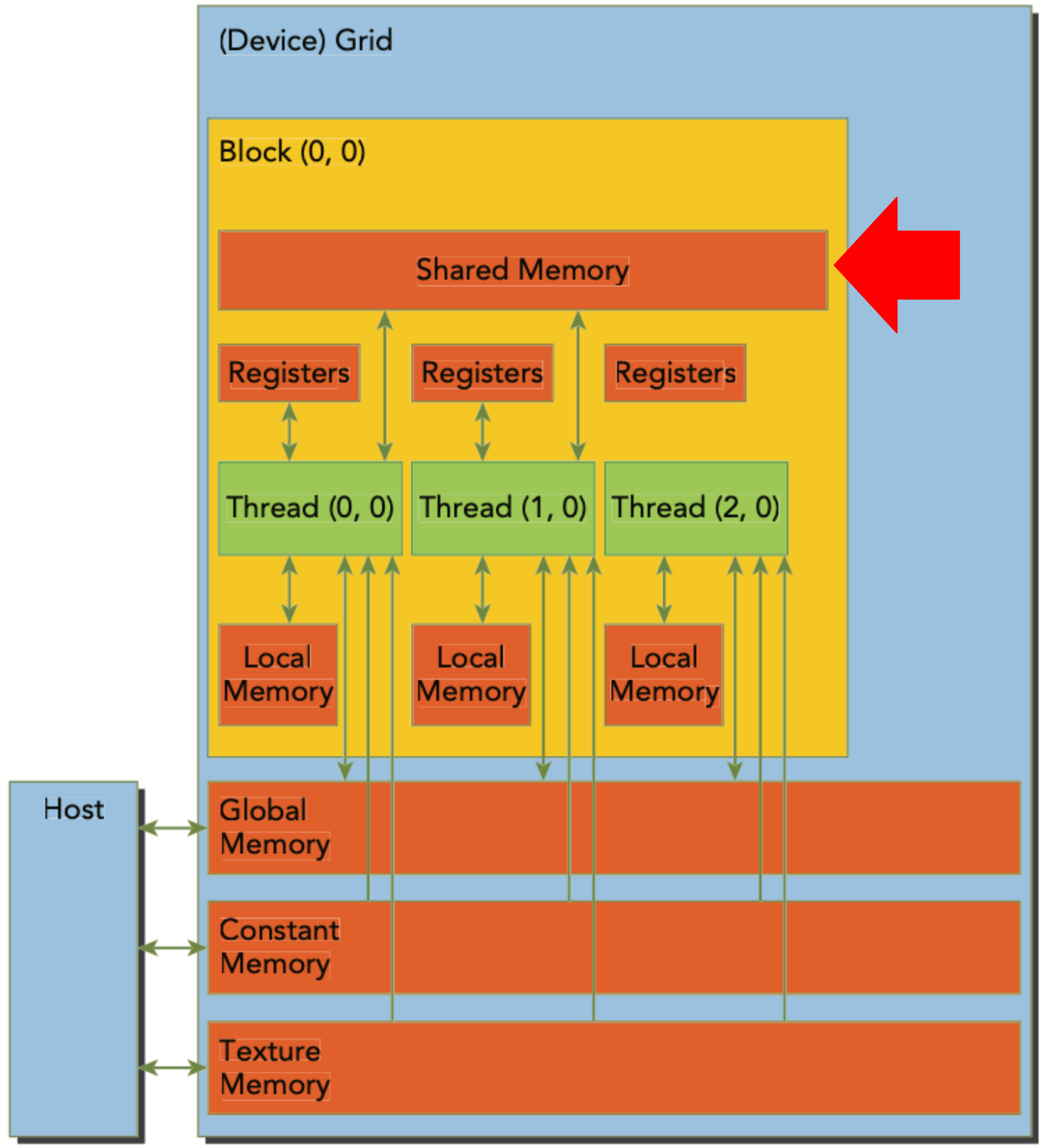
The shared variable is indicated by the `__shared__` qualifier

Two possible ways to allocate shared variables

- Static allocation
- Dynamic allocation

Shared variables are declared in the kernel code or globally (i.e., declared for all the kernels)

- Access: read/write for all block threads
- Scope: block
- Lifetime: block

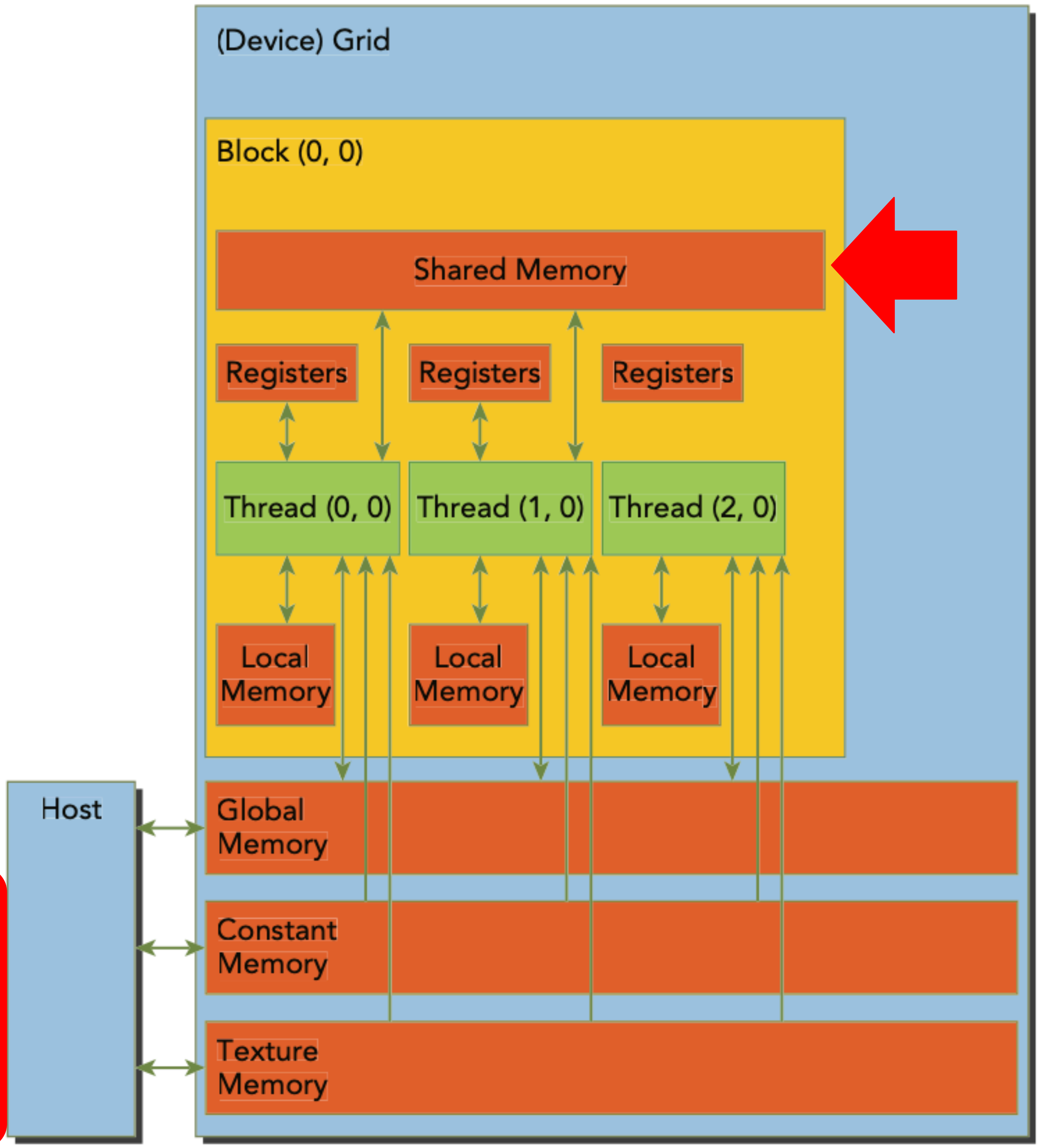


Shared Memory

Static allocation:

```
__global__ void foo(int* a, int* b) {
   __shared__ int sv[N];
   int i = blockIdx.x * blockDim.x +
           threadIdx.x;
   sv[i] = a[i];
   __syncthreads();
   /*...elaborate...*/
   __syncthreads();
   b[i] = sv[i];
}
```

Multiple accesses of threads of the same block to shared memory must be **synchronized** through barriers (`__syncthreads()`)

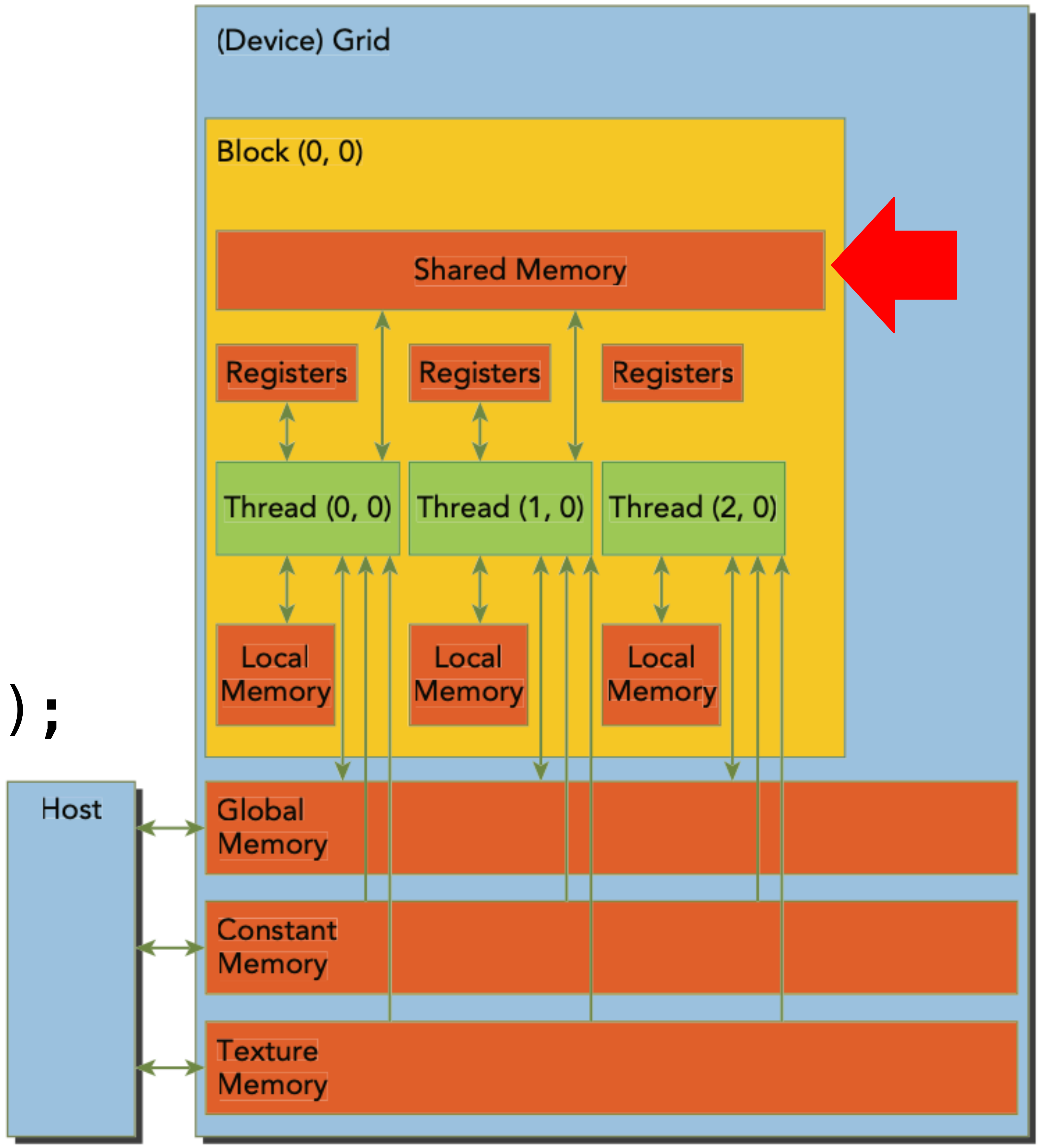


Shared Memory

Dynamic allocation:

```
__global__ void foo(int* a, int* b) {
  extern __shared__ int sv[];
  /*...*/
}

int main(){
  /*...*/
  foo<<<gdim, bdim, sizeof(int)*N>>>(va, vb);
}
```



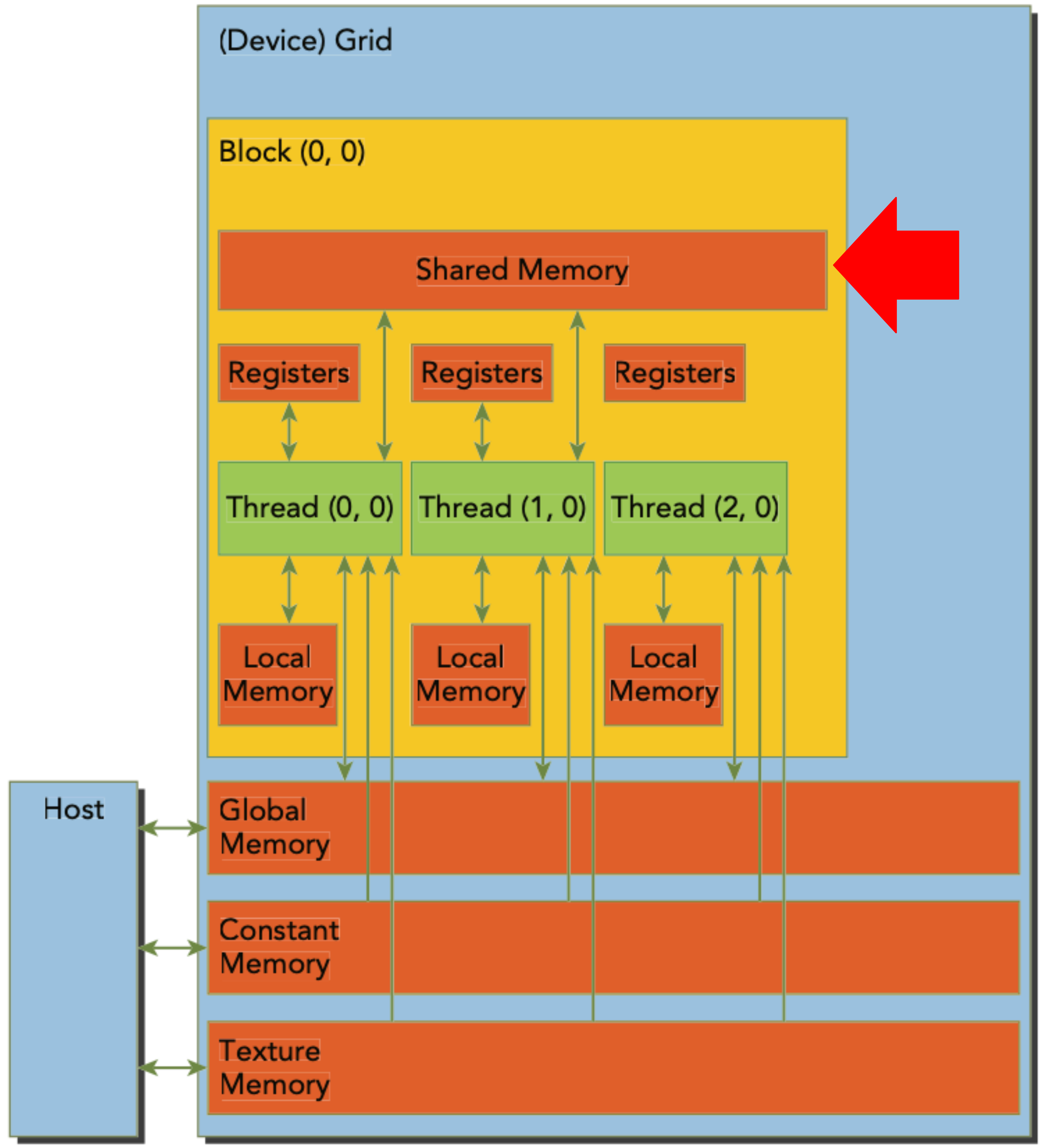
CUDA supports dynamic allocation of only 1D array

Allocation of multiple variables is tricky

Shared Memory

Typical shared memory usage:

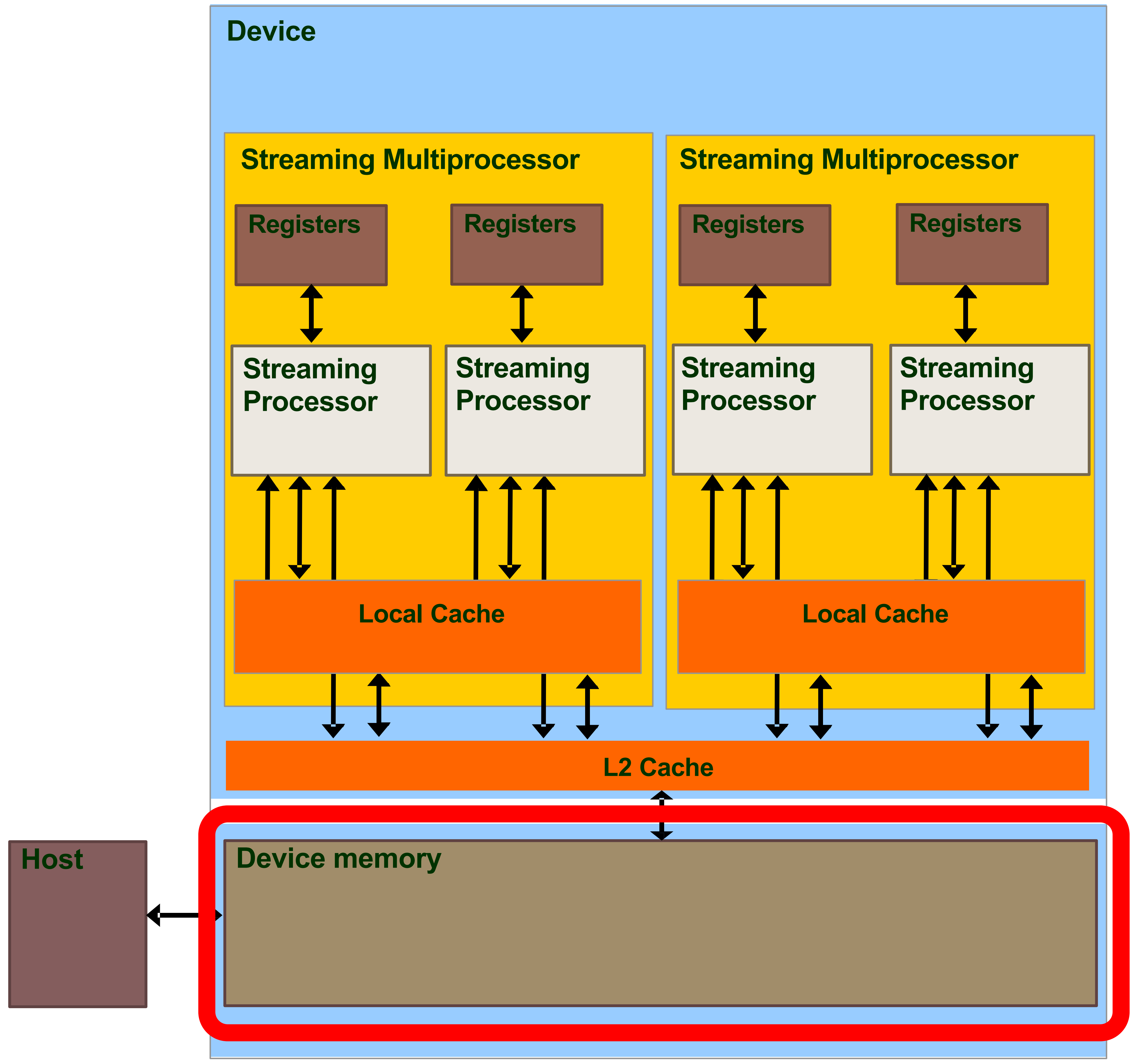
- Parallel load of data from global memory to shared memory
- Synchronization of block threads
- Elaboration of block threads on shared memory data
- Parallel store of data from shared memory to global memory



CUDA Memory Model

Key components:

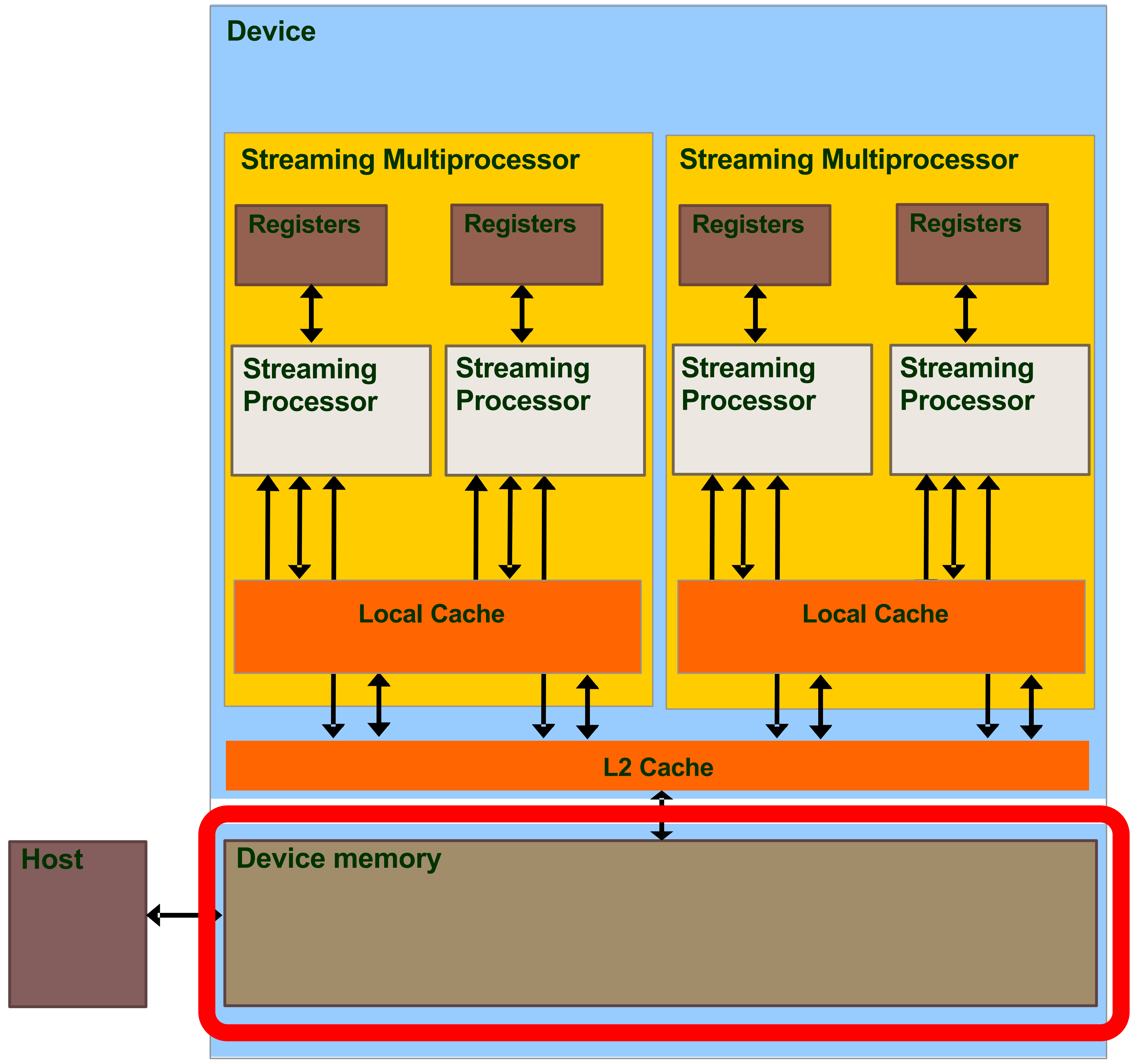
- **Global Memory (device-wide)**: **main memory** accessible by all threads across all blocks on the GPU, it has the largest capacity but is slower than registers and shared memory. It resides in device memory (HBM/DDR) and it is used for host/device communication. *Coalesced Access*: Access patterns in global memory can significantly affect performance. Coalesced memory accesses—where threads in a warp access consecutive memory addresses—optimize memory bandwidth.



CUDA Memory Model

Key components:

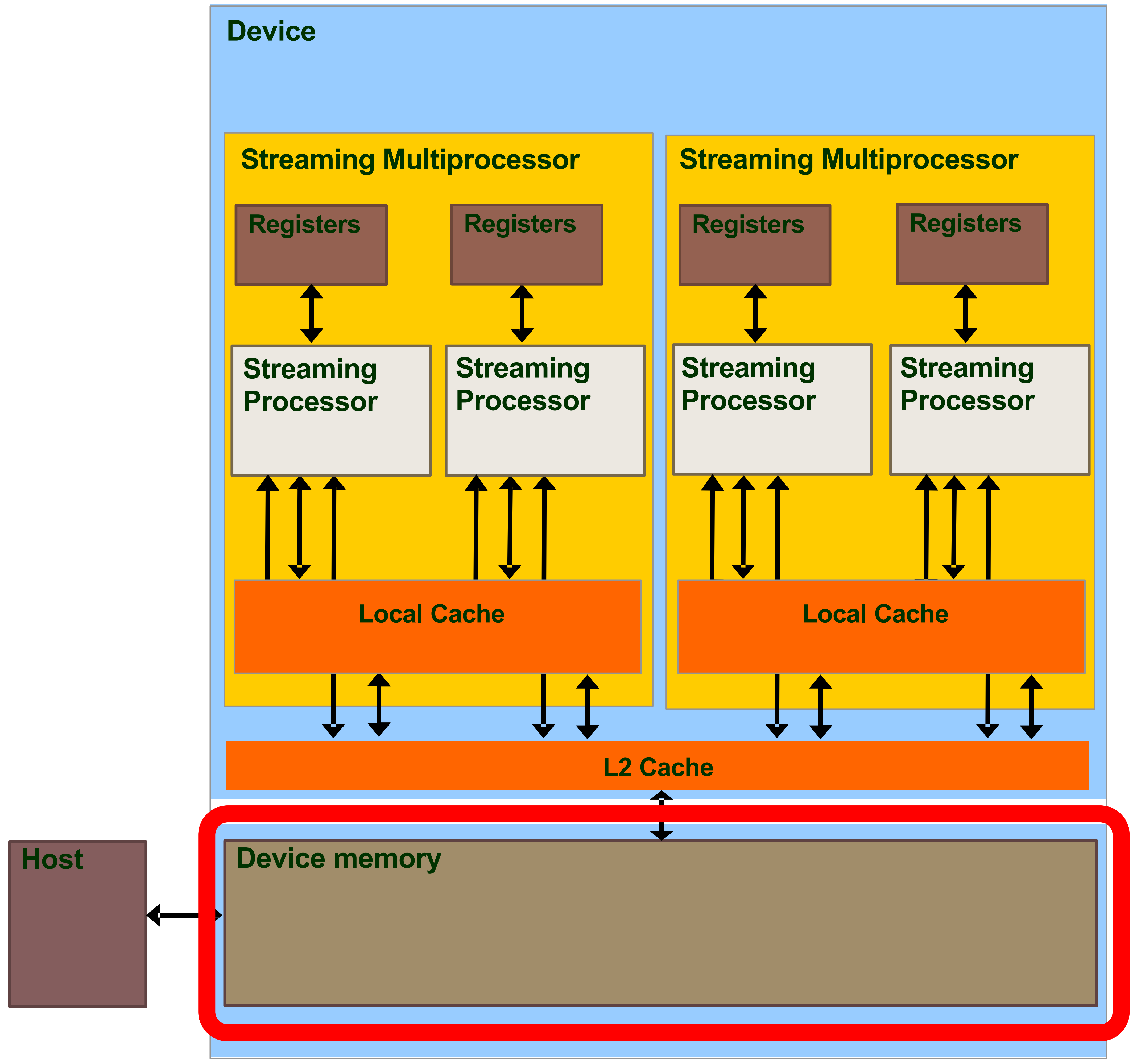
- **Constant Memory (device-wide, read-only)**: Constant memory is a **read-only** memory type that is cached, and it's accessible by all threads. It's mainly used to store data that remains constant throughout the execution of a kernel, such as lookup tables. Access to constant memory is optimized when all threads in a warp access the **<u>same</u> memory location**, as it avoids redundant fetches.



CUDA Memory Model

Key components:

- **Texture memory (device-wide, read-only)**: read-only memory optimized for **spatial locality** and **2D locality**. It's cached, making it more efficient for applications requiring non-uniform access patterns, such as image processing. Texture memory is accessed through the **texture cache**, which is optimized for spatial locality and can handle non-coalesced access patterns better than global memory.



Credits

Slides mainly based on:

- D. B. Kirk, W.-m. W. Hwu, **Programming Massively Parallel Processors: A Hands-on Approach**
- J. Chen, M. Grossman, T. Mckercher, **Professional Cuda C Programming**
- A. Miele, G. Accordi, A. Zeni, **GPUs and Heterogeneous Systems (programming models and architectures)**
- **NVIDIA GPU Teaching Kit**

Thanks for your attention!

Beatrice Branchini
beatrice.branchini@polimi.it

Advanced Computer Architecture Course
T.1.1 - 5/5/2024