Computação em Larga Escala

CUDA Programming

Eurico Pedrosa

António Rui Borges

Universidade de Aveiro - DETI

2025-05-18

CUDA

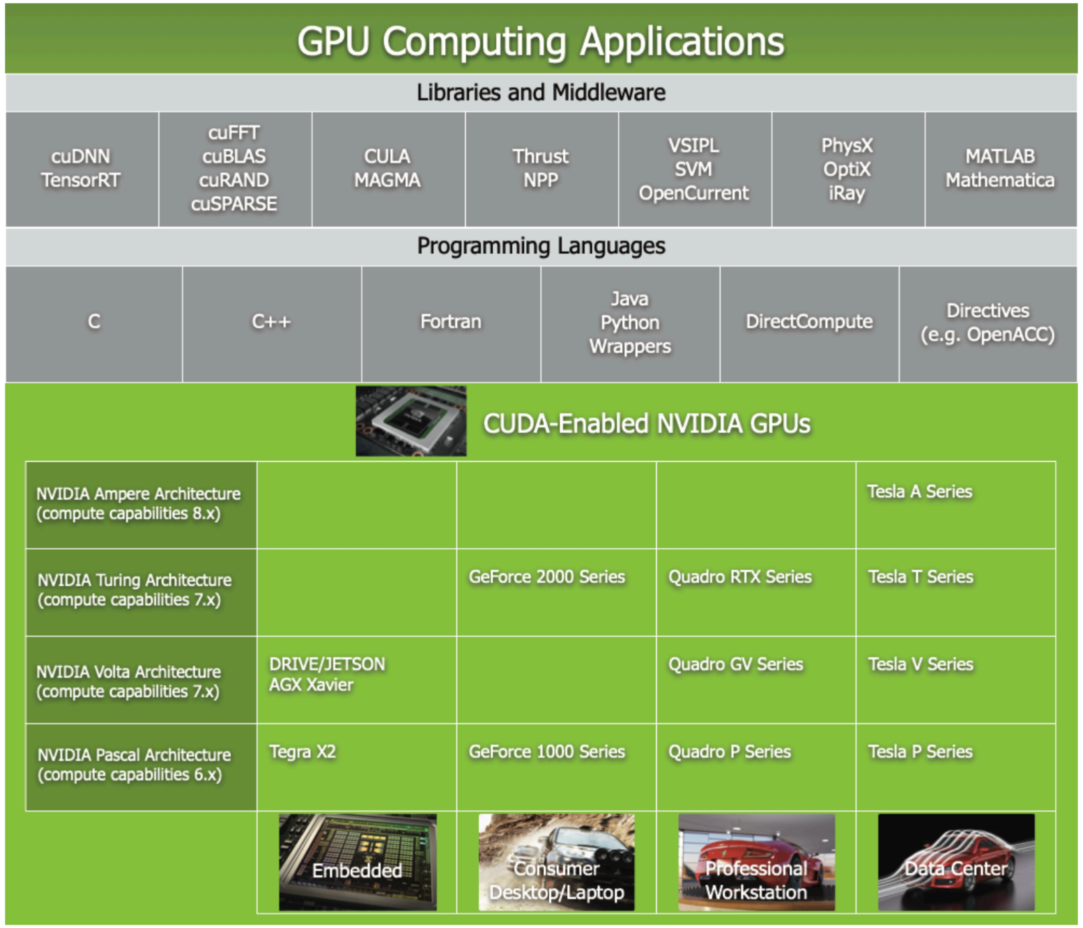
Overview

- **CUDA** is a **general-purpose parallel computing platform** and **programming model** developed by **NVIDIA**.
- It leverages the **parallel processing power** of NVIDIA **GPUs** to efficiently solve **computation-intensive problems**.

Overview

- The CUDA platform is accessible via:
  - **CUDA-accelerated libraries**
  - **Compiler directives**
  - **Application Programming Interfaces (APIs)**
  - **Extensions** to industry-standard languages such as **C, C++, Fortran, Java**, and **Python**
- **CUDA C** is an **extension of ANSI C** that includes:
  - Language features for **heterogeneous programming**
  - APIs to manage **devices**, **memory**, and **other GPU tasks**

Overview


GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

	Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center
NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series

Figure 1: GPU Computing Applications. Source: CUDA C Programming Guide, NVIDIA

API Levels

- CUDA provides **two API levels** for managing the GPU and organizing threads:
  - **CUDA Driver API** (docs.nvidia.com/cuda/cuda-driver-api)
  - **CUDA Runtime API** (docs.nvidia.com/cuda/cuda-runtime-api)

API Levels

- **Driver API**:
  - A **low-level API** that gives **fine-grained control** over GPU behavior
  - **Harder to program**, but offers **more flexibility**
- **Runtime API**:
  - A **high-level API** built **on top of the Driver API**
  - Easier to use; each runtime function maps to one or more driver-level operations

API Levels

- **Key notes**:
  - There is **no significant performance difference** between the two APIs
  - **Thread organization** and **memory usage** impact performance more than API choice
  - **Mutually exclusive**: you must use **either** the Driver API **or** the Runtime API in your code — **no mixing allowed**

Compilation Model

- The **CUDA `nvcc` compiler** separates **host code** and **device code** during compilation
- **Host code**:
  - Written in **standard C/C++**
  - Compiled using a **regular C/C++ compiler**
- **Device code**:
  - Written in **CUDA C** with special **keywords** to label **kernels** (data-parallel functions)
  - Compiled by **`nvcc`** for execution on the **GPU**

Compilation Model

- During the **linking stage**:
  - **CUDA runtime** or **driver libraries** are linked in
  - Enables **kernel calls** and **explicit GPU management**
- The **CUDA Toolkit** includes:
  - The **nvcc compiler**
  - **Mathematical libraries**
  - **Debugging** and **performance optimization tools**

Core Abstractions

- CUDA exposes three core abstractions through minimal language extensions:
  - **Hierarchy of thread groups**
  - **Shared memory**
  - **Barrier synchronization**
- These abstractions support:
  - **Fine-grained data parallelism** and **thread parallelism**
  - Nested within **coarse-grained data parallelism** and **task parallelism**

Core Abstractions

- **Problem decomposition** strategy:
  - A problem is partitioned into **subproblems**, each solved in parallel by **thread blocks**
  - Each subproblem is further subdivided and solved cooperatively by **threads within a block**

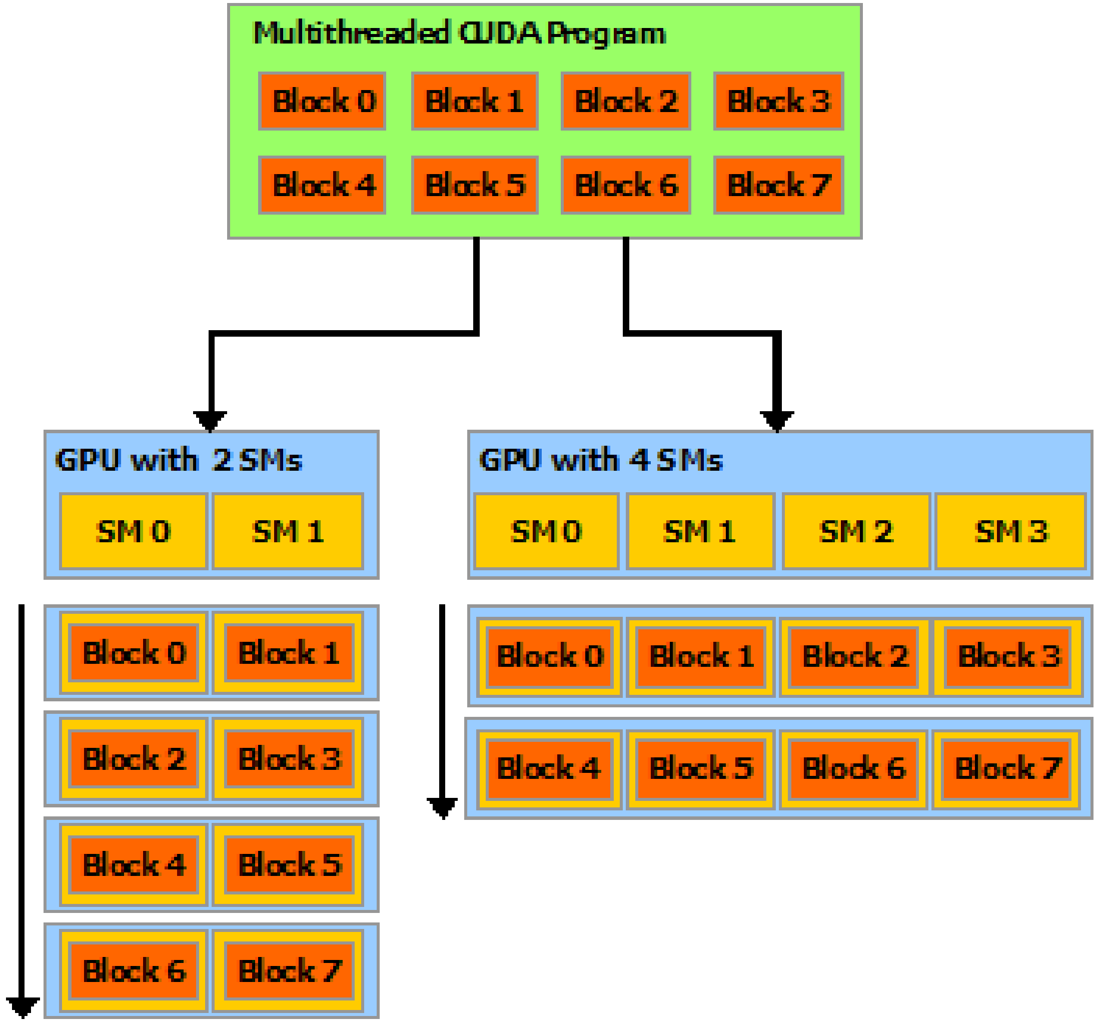
Core Abstractions

- **Benefits**:
  - Preserves **language expressiveness** through intra-block **thread cooperation**
  - Enables **scalable execution**:
    - Thread blocks are **independent units of execution**
    - The **runtime system** schedules blocks on **any available multiprocessor**, in **any order**
    - Allows execution on **any GPU**, regardless of the number of multiprocessors

GPU Hardware Architecture

- A **GPU** consists of an array of **Streaming Multiprocessors (SMs)**
- Internally organized as a **MIMD** (Multiple Instruction, Multiple Data) topology of **SIMD** (Single Instruction, Multiple Data) processors
- A **multithreaded CUDA program** is divided into **blocks of threads**
  - **Thread blocks** execute **independently** of each other
  - This allows for **scalable execution** across different GPU hardware
- **Scalability benefit**:
  - A GPU with **more multiprocessors** can execute **more blocks concurrently**
  - This leads to **faster execution** without requiring changes to the code

GPU Hardware Architecture



Automatic Scalability

Source: CUDA C Programming Guide, NVIDIA

Program Structure

Typical CUDA Program Structure

1. **Allocate GPU memory**
2. **Copy data** from **CPU (host) memory** to **GPU (device) memory**
3. **Invoke CUDA kernel** to perform program-specific computation
4. **Copy results back** from **GPU memory** to **CPU memory**
5. **Free (destroy) GPU memory**

Program Structure

```
#include "../common/common.h"
#include <stdio.h>
/* A simple introduction to programming in CUDA. This program
* prints "Hello World from GPU! from 10 CUDA threads running
* on the GPU.
*/
__global__ void helloFromGPU()
{
  printf("Hello World from GPU!\n");
}

int main(int argc, char **argv)
{
  printf("Hello World from CPU!\n");
  helloFromGPU<<<1, 10>>>();
  CHECK(cudaDeviceReset());
  return 0;
}
```

CUDA Programming Model

Kernels and Threading Model

- **CUDA C** extends C by allowing the definition of **kernels** — special functions executed **concurrently by N CUDA threads**
  - In contrast to regular C functions, which execute only once
- A **kernel** is defined using the `__global__` specifier
- A kernel is launched with an **execution configuration**:
  - Syntax: `<<<gridDim, blockDim>>>`
  - Defines a **grid** of **thread blocks** running concurrently

Kernels and Threading Model

- Each thread is assigned a **unique thread ID** within its block:
  - Accessible via the built-in variable **`threadIdx`**
- Each thread has:
  - Its own **registers** and **private memory**
  - Access to **input data**
  - Responsibility to **compute and output results**

Thread Blocks and Thread Indexing

- **`threadIdx`** is a **3-component vector**:
  - Threads can be indexed in **1D**, **2D**, or **3D**
  - Forms a **1D, 2D, or 3D thread block**
  - Ideal for computations over **vectors, matrices, or volumes**
- **Thread ID computation** (assuming block dimensions `Dx`, `Dy`, `Dz`):
  - **1D block**:
    - Thread ID = `x`
  - **2D block**:
    - Thread ID = `x + y × Dx`
  - **3D block**:
    - Thread ID = `x + y × Dx + z × Dx × Dy`

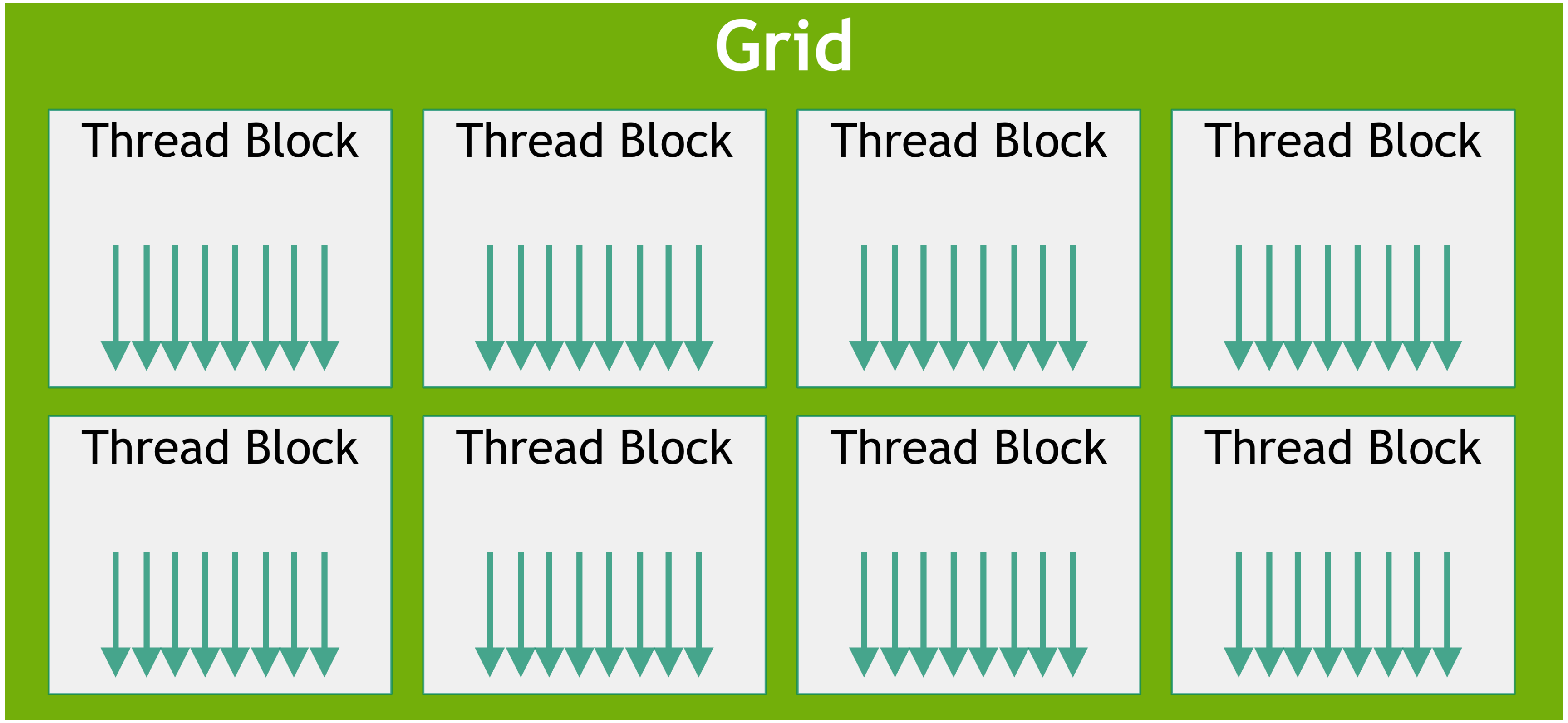
Thread Blocks and Thread Indexing

- **Hardware constraints**:
  - A **thread block** can have up to **1024 threads** (current GPUs)
  - All threads in a block **reside on the same SM (Streaming Multiprocessor)**
  - Threads **share memory resources** (registers, shared memory) of that SM
- **Scalability**:
  - A **kernel** can be launched with **multiple blocks**
  - Total number of threads = **threads per block** × **number of blocks**

Grid of Thread Blocks

- Thread **blocks are organized** into a **1D, 2D, or 3D grid**
- The number of **blocks per grid** is usually determined by:
  - The **size of the data set**, or
  - The **number of available GPU processors** (can exceed both)
- Execution configuration (`<<<...>>>`) supports:
  - **Thread block dimensions** and **grid dimensions** of type `int` or **`dim3`**
- Within a kernel:
  - Each block is identified by **`blockIdx`**
    - A **1D, 2D, or 3D index** depending on grid shape
  - The size of each block is given by **`blockDim`**

Grid of Thread Blocks



Grid Of Thread Blocks

Source: CUDA C Programming Guide, NVIDIA

Thread Block Independence & Synchronization

- **Thread blocks must execute independently**:
  - Can be scheduled **in any order**, **in parallel**, or **in series**
  - Enables **scalability** across GPUs with varying numbers of cores
- This independence allows blocks to be **assigned dynamically** to **any core**, supporting scalable parallel execution

Thread Block Independence & Synchronization

- **Threads within a block can cooperate** through:
  - **Shared memory** (fast, low-latency memory near the processor core)
  - **Barrier synchronization** using **`__syncthreads()`**
- **`__syncthreads()`**:
  - Acts as a **barrier** — all threads in the block must reach it before any can proceed
  - Enables **coordinated memory access**
  - Expected to be a **lightweight operation**, like L1 cache synchronization in CPUs

Warps and Execution

It’s important to know that warps are the unit of actual execution on the GPU. Although we write kernel code for one thread, the GPU executes threads in batches of 32 called a warp.

- **Warp = 32 threads**:
  - The GPU scheduler groups threads of a block into warps of 32
  - A warp executes instructions in SIMD fashion (Single Instruction, Multiple Threads a.k.a. SIMT).
- **Lock-step execution**: All threads in a warp execute the same instruction concurrently on different data. If threads in a warp take different paths (branch divergence), the warp serializes those paths (reducing efficiency)

Warps and Execution

- **Warp scheduling**: Each SM has a warp scheduler. Warps that are ready to execute are interleaved to hide latency (while one warp waits for memory, another can execute).
- **Implication for block size**: Using multiples of 32 threads per block (e.g., 128, 256, 512) is often best to avoid partial warps. The block is essentially cut into warps internally.
- **Independent thread scheduling** (new GPUs): Since Volta, threads in a warp can diverge more flexibly, but the cost of divergence (inactive threads) still exists

Memory Model

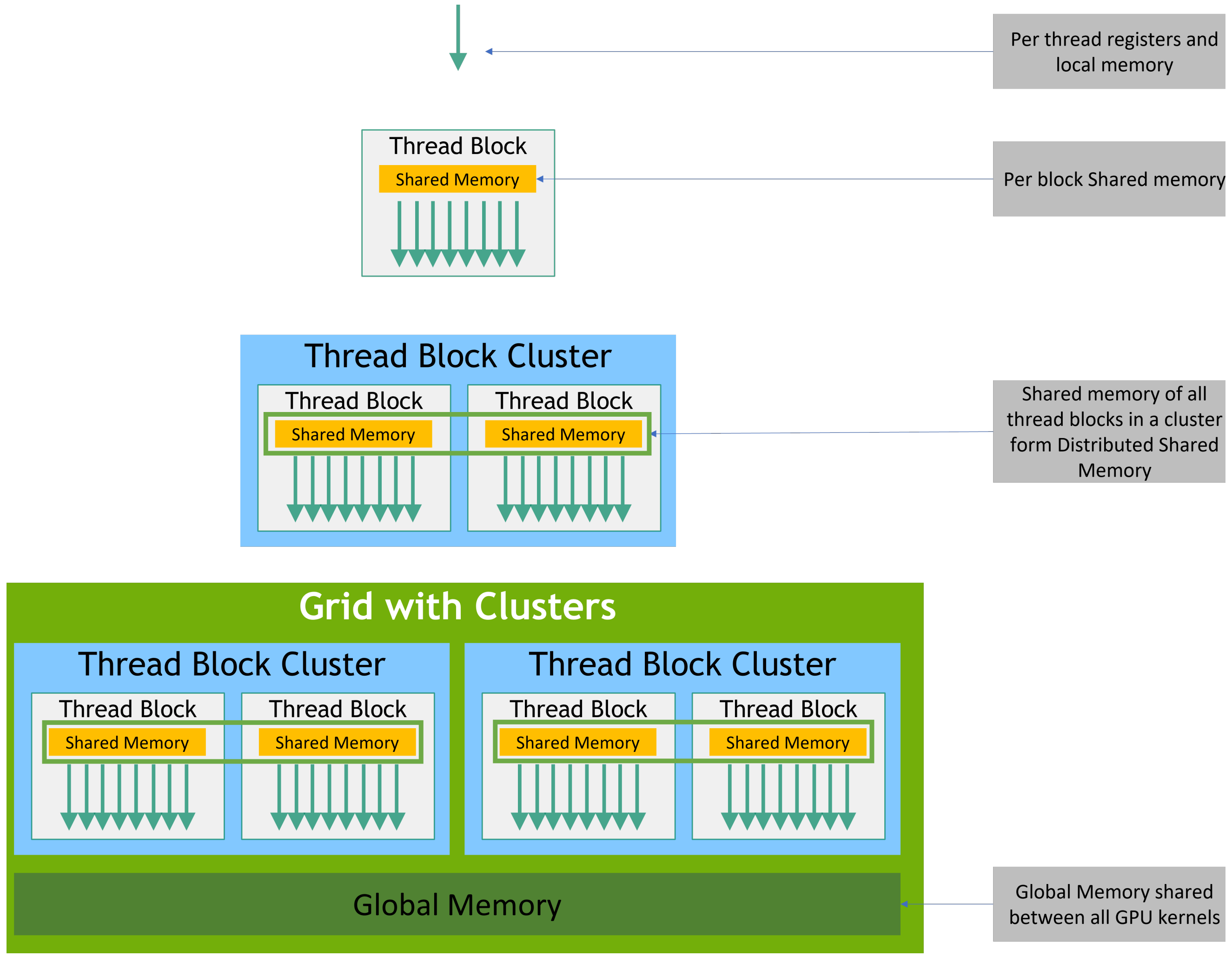
- **CUDA threads access multiple memory spaces** during execution:
  - **Local memory**: private to each **thread**
  - **Shared memory**: accessible by all **threads in a block**, with block lifetime
  - **Global memory**: accessible by **all threads** across all blocks
  - **Constant memory**: **read-only**, accessible by all threads
  - **Texture memory**: **read-only**, supports **special addressing** and **data filtering**

- **Global**, **constant**, and **texture memory**:
  - Optimized for **different usage patterns**
  - Are **persistent across kernel launches** within the same application

Memory Model

- **Host vs. Device memory**:
  - CUDA assumes **separate DRAM** for **host (CPU)** and **device (GPU)**
  - Requires explicit **memory management** via the **CUDA runtime API**, including:
    - **Device memory allocation/deallocation**
    - **Data transfers** between **host memory** and **device memory**

Memory Model



Memory Hierarchy

Source: CUDA C Programming Guide, NVIDIA

Heterogeneous Programming

Host–Device Memory and Unified Memory in CUDA

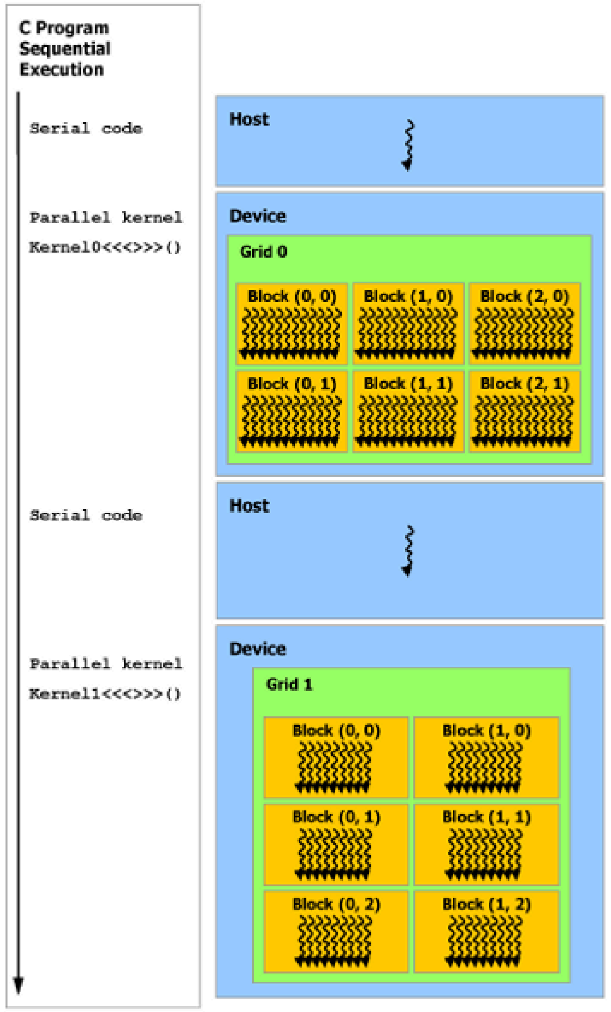
- CUDA assumes **separate memory spaces** for:
  - **Host (CPU)** $\longrightarrow$ **Host memory**
  - **Device (GPU)** $\longrightarrow$ **Device memory**

- Programs must manage:
  - **Global**, **constant**, and **texture memory spaces** via the **CUDA Runtime API**
  - This includes:
    - **Device memory allocation/deallocation**
    - **Explicit data transfers** between **host** and **device**

Heterogeneous Programming

- **Unified Memory** (also called **Managed Memory**):
  - Provides a **single, coherent memory space** shared between **CPU and GPU**
  - Enables:
    - **Automatic data migration**
    - **Oversubscription** of device memory
    - **Simplified programming** — no need for manual memory copying
    - Easier **application porting** without explicit host–device mirroring

Heterogeneous Programming



Heterogeneous Programming

Source: CUDA C Programming Guide, NVIDIA

CUDA Function Type Qualifiers

Qualifier	Executed On	Callable From	Notes
`__global__`	Device	Host	Must return `void`; launches a kernel
`__device__`	Device	Device	Used for device-only functions
`__host__`	Host	Host	Optional; default for host code

- `__device__` and `__host__` can be **used together**:
  - This compiles the function for **both host and device use**
- Common usage:
  - **Kernel functions** use `__global__`
  - **Helper functions** shared by host and device may use `__host__ __device__`

Kernel Restrictions

CUDA **kernels** are special functions with several important restrictions:

- Must **only access device memory**
- Must have a **`void` return type**
- **Variable argument lists** (e.g., ...) are **not supported**
- **Static variables** are **not allowed**
- **Function pointers** are **not supported**
- Kernel launches are **asynchronous**
  - Control returns to the host **before** the kernel finishes execution

Suggested Reading

- CUDA C Programming Guide, NVIDIA
  - (https://docs.nvidia.com/cuda/cuda-c-programming-guide/)