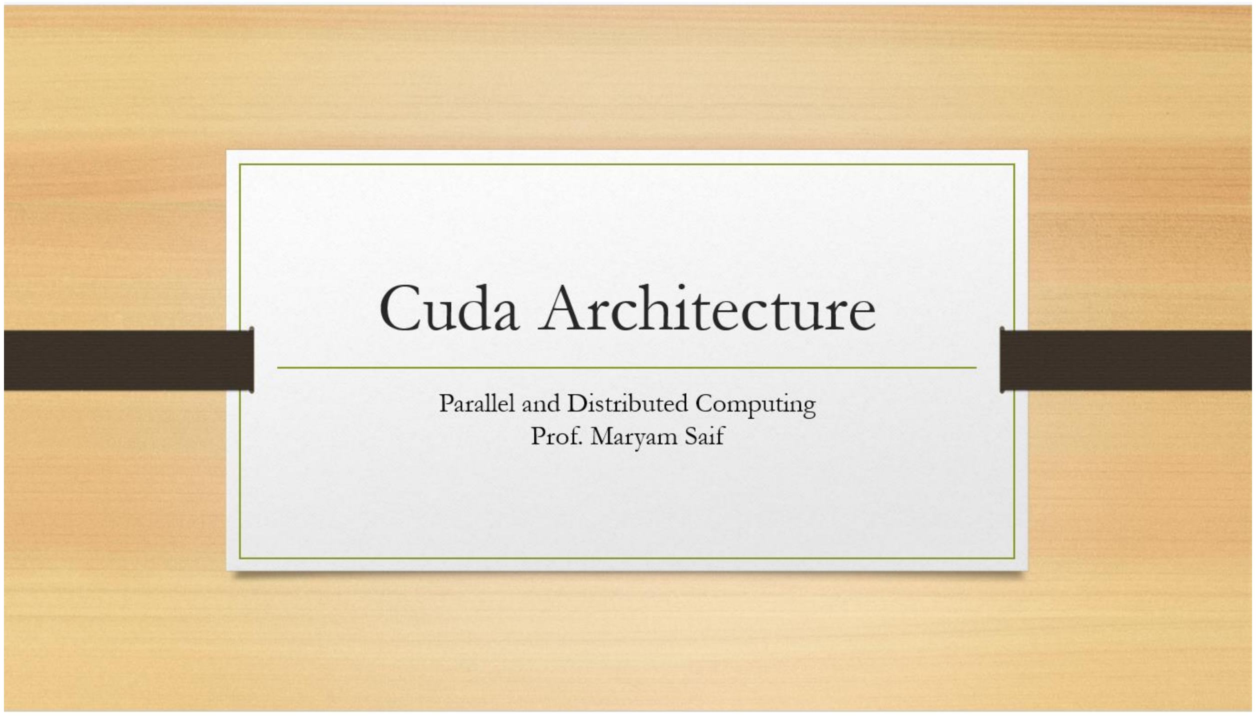
reg # 18, 19


Cuda Architecture

Parallel and Distributed Computing
Prof. Maryam Saif

CUDA Architecture in Parallel and Distributed Computing

1. Introduction

CUDA (Compute Unified Device Architecture) is **NVIDIA's parallel computing platform and programming model**. It allows developers to use **GPUs (Graphics Processing Units)** for general-purpose computing tasks, leveraging thousands of cores to achieve massive parallelism. CUDA is widely used in scientific computing, AI/ML, image processing, and other high-performance tasks.

---

2. CUDA Programming Model

CUDA follows a **heterogeneous computing model**:

- **Host (CPU):** Executes serial parts of the code and coordinates GPU tasks.
- **Device (GPU):** Executes highly parallel tasks (kernels) on multiple threads simultaneously.

Programmers write **kernels** (functions) that are executed in **parallel by multiple GPU threads**.

3. CUDA Thread Hierarchy

CUDA organizes parallelism using a **hierarchical structure**:

1. **Thread:** The smallest execution unit.
2. **Warp:** A group of 32 threads executed simultaneously on a GPU's streaming multiprocessor (SM).
3. **Block:** Threads are grouped into **thread blocks**. Each block has its own **shared memory**.
4. **Grid:** A collection of blocks. Grids allow scaling to very large numbers of threads.

Diagram (conceptual):

```
Grid → Blocks → Threads
```

4. CUDA Memory Model

CUDA uses **different types of memory** for performance optimization:

Memory Type	Scope & Latency
Registers	Private to each thread, very fast
Shared Memory	Shared within a block, fast
Global Memory	Accessible by all threads, slow
Constant / Texture	Read-only memory, cached

Memory hierarchy and coalesced access patterns are crucial for high performance.

5. Parallelism in CUDA

- Each GPU thread works **independently** on data.
- Threads within a block can **synchronize** using `__syncthreads()`.
- Massive parallelism allows GPUs to execute thousands of threads simultaneously, making them ideal for **data-parallel tasks** like matrix multiplication, image filtering, and deep learning.

6. CUDA and Distributed Computing

While CUDA is primarily **shared memory parallelism** on a GPU, it can be extended to **distributed systems**:

- **Multi-GPU systems:** Multiple GPUs in the same node communicate via **PCIe** or **NVLink**.
- **Clustered GPUs:** Distributed memory across nodes, communication handled via **MPI + CUDA-aware libraries**.
- **Applications:** HPC simulations, AI model training on clusters, weather/climate modeling.

Key idea: CUDA accelerates **intra-node parallelism**, while MPI or other message-passing methods handle **inter-node communication** in distributed computing.

---

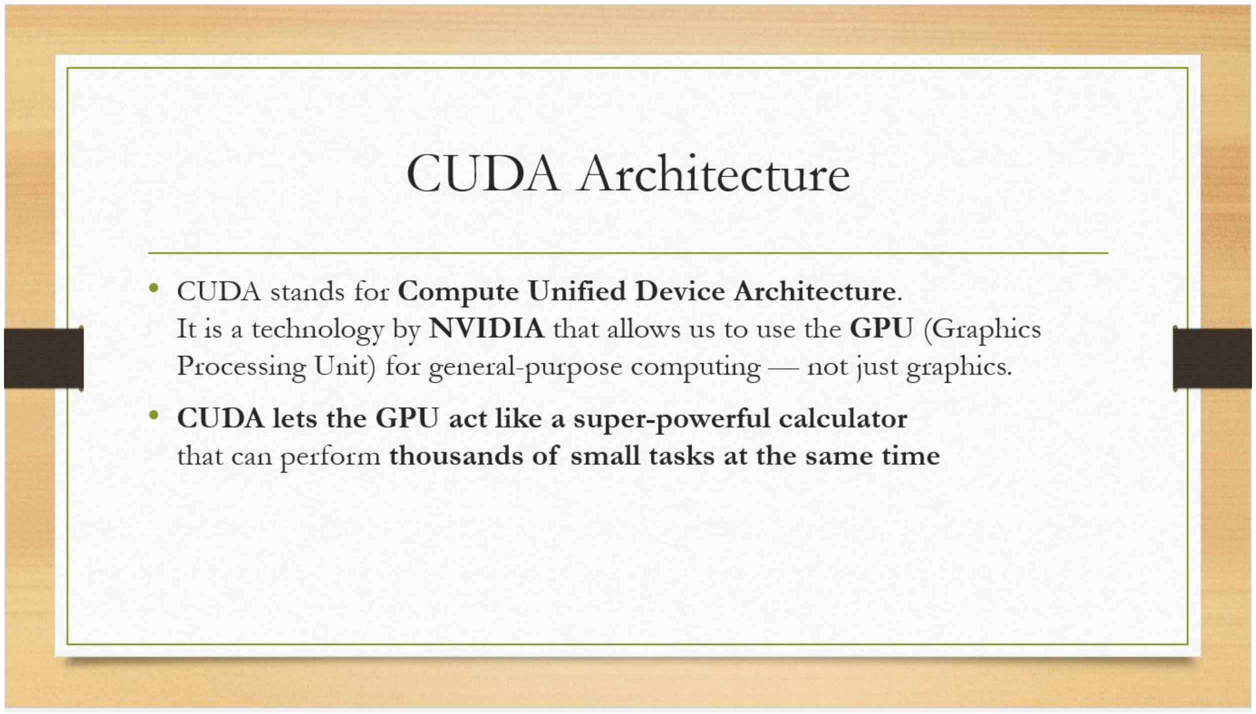
7. Advantages

- Massive parallelism using thousands of threads.
- Efficient memory hierarchy for high performance.
- Scalable from single GPU to multi-GPU clusters.
- Well-supported by libraries like **cuBLAS**, **cuDNN**, **Thrust** for AI/ML and scientific computing.

---

8. Use Cases

- **Machine Learning / Deep Learning:** Training large neural networks.
- **Scientific Simulations:** Fluid dynamics, molecular modeling.
- **Image/Video Processing:** Real-time rendering, filtering, transformations.
- **High-Performance Computing Clusters:** Combined CUDA + MPI for multi-node GPU computing.

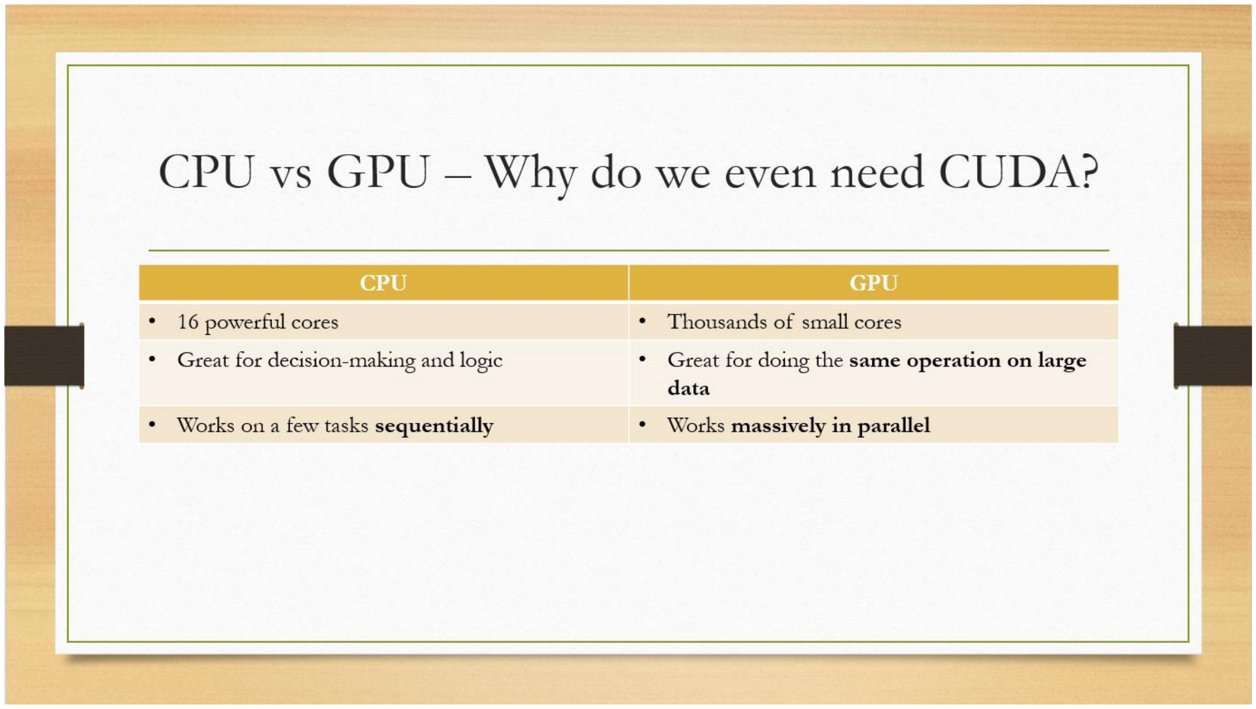

CUDA Architecture

- CUDA stands for **Compute Unified Device Architecture**.
  It is a technology by **NVIDIA** that allows us to use the **GPU** (Graphics Processing Unit) for general-purpose computing — not just graphics.
- **CUDA lets the GPU act like a super-powerful calculator** that can perform **thousands of small tasks at the same time**

CUDA Architecture

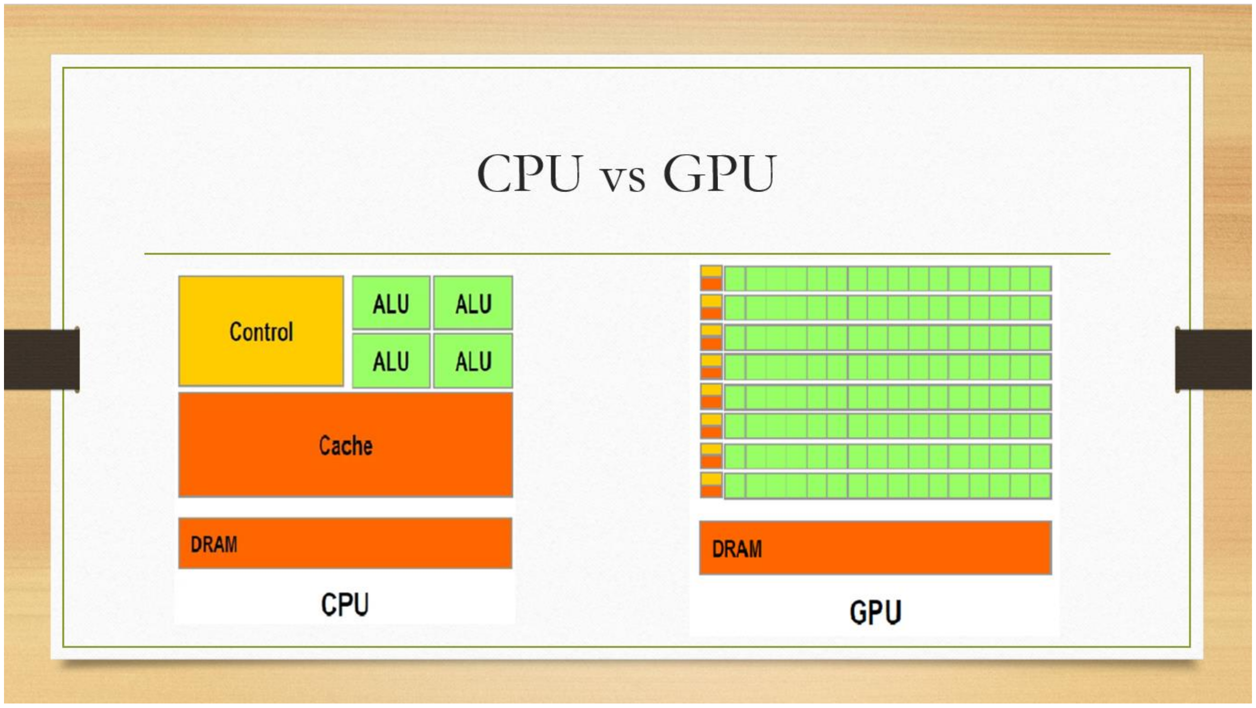
CUDA stands for **Compute Unified Device Architecture**. It is a parallel computing platform and programming model developed by **NVIDIA** that allows the **GPU (Graphics Processing Unit)** to be used for **general-purpose computing**, beyond just graphics rendering.

With CUDA, the GPU behaves like a **highly parallel super-computer**, capable of executing **thousands of small tasks simultaneously**, making it ideal for applications requiring massive parallelism such as scientific simulations, AI/ML workloads, and image processing.



CPU vs GPU – Why do we even need CUDA?

CPU	GPU
• 16 powerful cores	• Thousands of small cores
• Great for decision-making and logic	• Great for doing the **same operation on large data**
• Works on a few tasks **sequentially**	• Works **massively in parallel**



CPU vs GPU

CPU vs GPU – Why Do We Need CUDA?

1. CPU (Central Processing Unit):

- Designed for **general-purpose computing**.

- Has **few cores** (typically 4–32) optimized for **sequential task execution**.
- Excellent at **complex logic and branching tasks**, but limited for tasks that require **massive parallelism**.

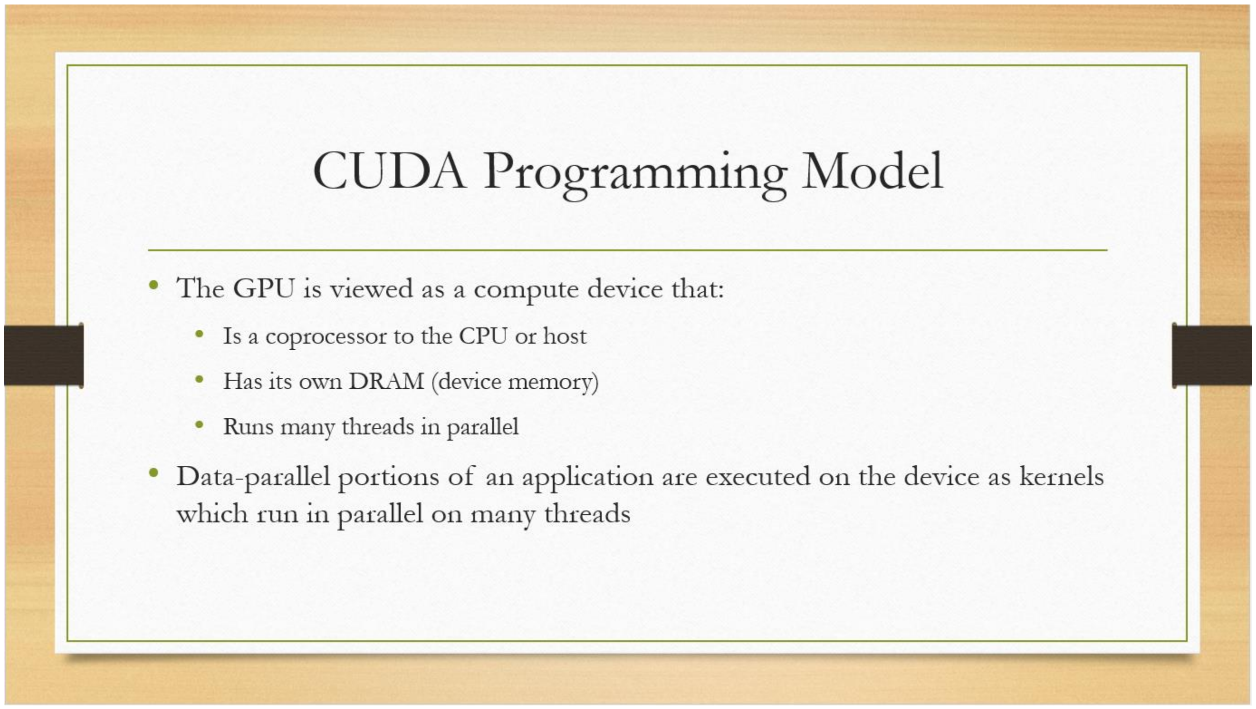
2. GPU (Graphics Processing Unit):

- Designed for **highly parallel workloads**, originally for graphics rendering.
- Has **thousands of smaller cores** capable of executing many tasks simultaneously.
- Ideal for **data-parallel computations**, such as matrix operations, image processing, and deep learning.

3. Why CUDA?
CPUs alone cannot handle the **extreme parallelism** required by modern scientific simulations, AI/ML training, or large-scale data processing efficiently. CUDA provides:

- **Access to GPU cores:** Lets programmers offload heavy parallel tasks to the GPU.
- **Massive parallel execution:** Thousands of threads can compute simultaneously.
- **Optimized memory management:** Shared, global, and constant memory for high-speed computations.
- **Integration with HPC frameworks:** Works with MPI or multi-GPU clusters for distributed computing.

Analogy:
Think of a CPU as a **single expert worker** solving complex problems one by one, while a GPU with CUDA is like a **factory with thousands of workers**, each performing simple tasks simultaneously to finish large computations much faster.



CUDA Programming Model

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads

---

CUDA Programming Model

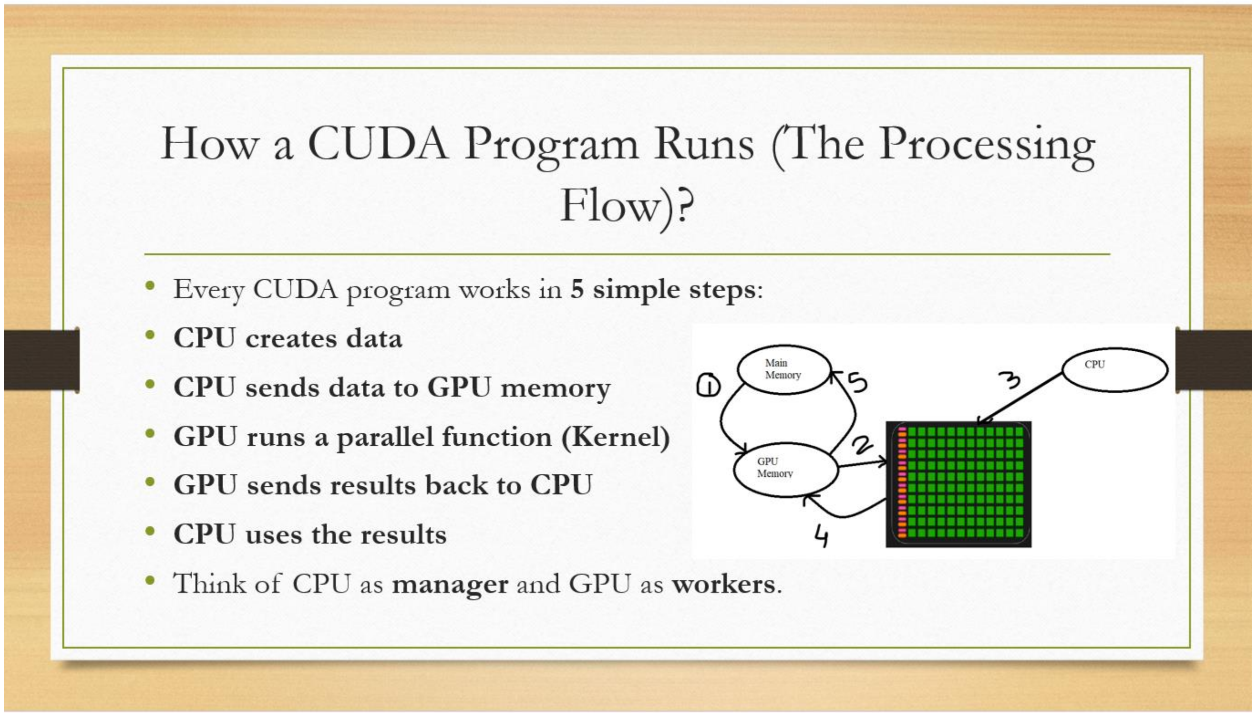
In the CUDA programming model, the **GPU is treated as a compute device** that works alongside the CPU (host) as a **coprocessor**. Key characteristics include:

- **Independent Device Memory:** The GPU has its own DRAM, called **device memory**, separate from the CPU’s memory.
- **Massive Parallelism:** The GPU can run **thousands of threads simultaneously**, allowing highly parallel execution.
- **Kernels:** Data-parallel portions of an application are implemented as **kernels**, which are functions executed **in parallel across many threads** on the GPU.

Workflow:

1. The CPU (host) launches kernels on the GPU (device).
2. Threads in the GPU execute the kernel in parallel, often organized in **blocks and grids**.
3. Results are transferred back from device memory to host memory as needed.

This model allows developers to **accelerate computationally intensive tasks** by exploiting the GPU’s parallel architecture.



How a CUDA Program Runs (The Processing Flow)?

- Every CUDA program works in **5 simple steps**:
- **CPU creates data**
- **CPU sends data to GPU memory**
- **GPU runs a parallel function (Kernel)**
- **GPU sends results back to CPU**
- **CPU uses the results**
- Think of CPU as **manager** and GPU as **workers**.

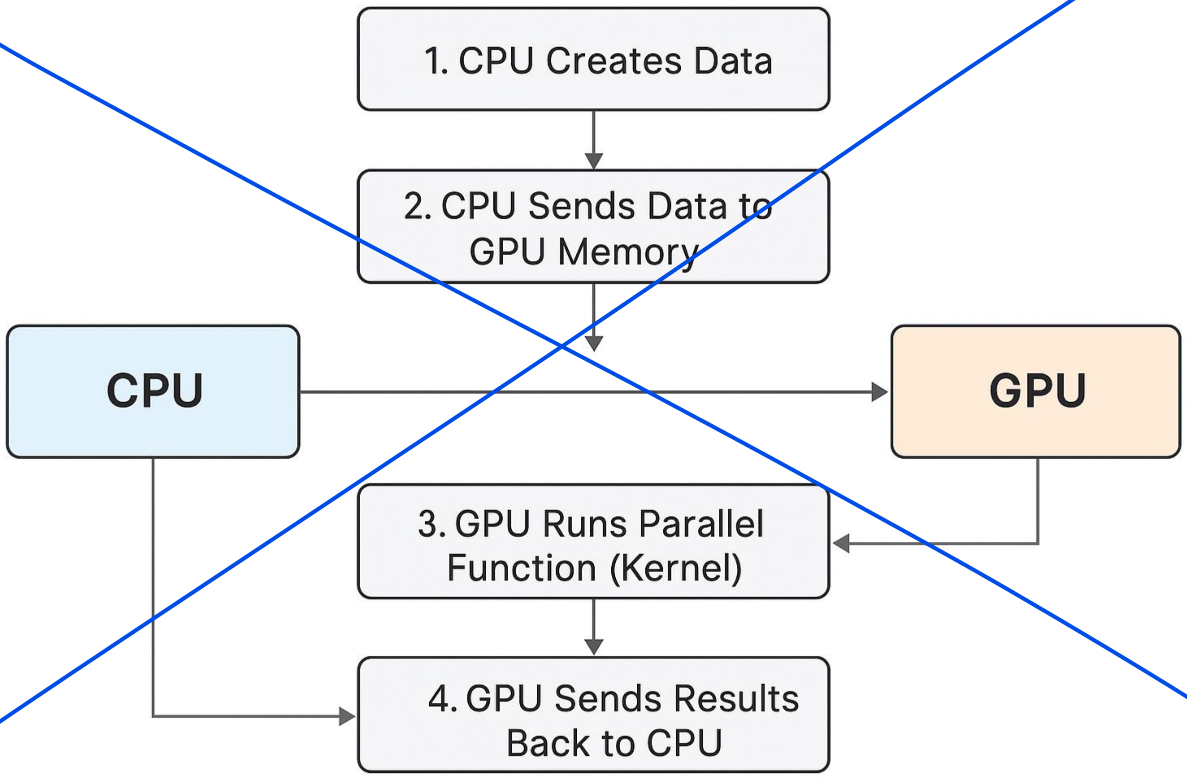
Main Memory
CPU
GPU Memory
1
2
3
4
5

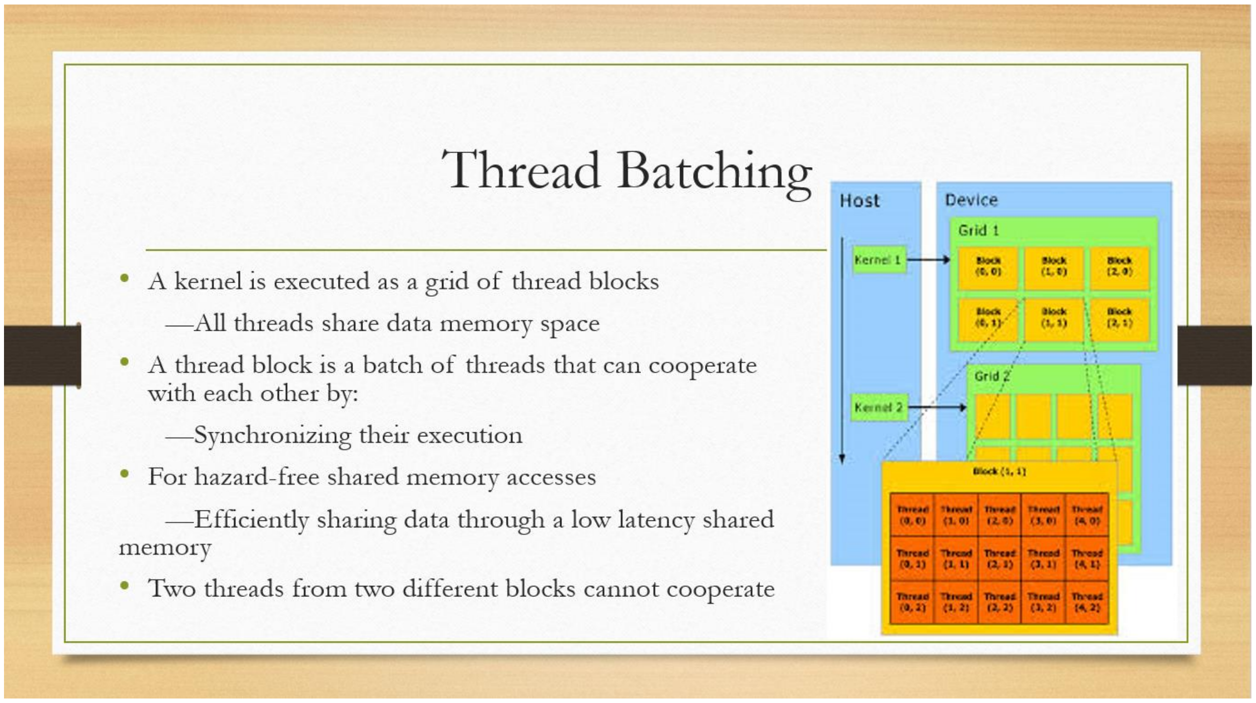
How a CUDA Program Runs (Processing Flow)

Every CUDA program follows a **simple five-step workflow**:

1. **CPU Prepares Data:** The host (CPU) creates and initializes the data required for computation.
2. **Data Transfer to GPU:** The CPU sends this data to the GPU's **device memory**.
3. **Kernel Execution on GPU:** The GPU runs a **parallel function (kernel)** across thousands of threads simultaneously.
4. **Results Transfer Back:** Once computation is complete, the GPU sends the results back to the CPU memory.
5. **CPU Uses the Results:** The host processes or displays the results as needed.

Analogy: Think of the **CPU as the manager** who assigns tasks and coordinates work, while the **GPU acts as a team of workers** executing many tasks in parallel.


1. CPU Creates Data
2. CPU Sends Data to GPU Memory
CPU
GPU
3. GPU Runs Parallel Function (Kernel)
4. GPU Sends Results Back to CPU



Thread Batching

- A kernel is executed as a grid of thread blocks
  —All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  —Synchronizing their execution
- For hazard-free shared memory accesses
  —Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate

Thread Batching in CUDA

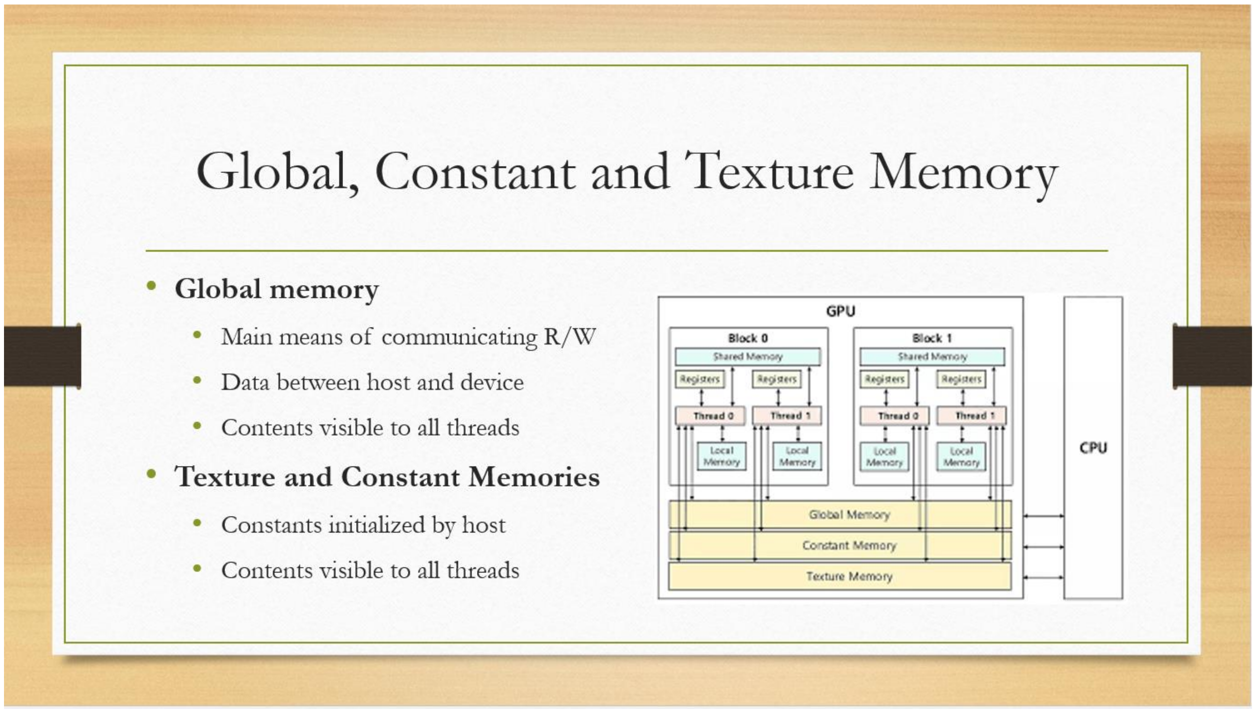
In CUDA, a **kernel is executed as a grid of thread blocks**, where threads within a block can work together efficiently:

- **Thread Block:** A batch of threads that can **cooperate with each other**.
- **Shared Memory:** Threads in the same block can share data using **low-latency shared memory**, allowing fast communication within the block.
- **Synchronization:** Threads within a block can **synchronize their execution** to ensure hazard-free access to shared memory.

Important Note:

- Threads **from different blocks cannot directly cooperate** or share memory. Communication across blocks must occur through **global memory**, which is slower than shared memory.

This model allows CUDA to balance **massive parallelism** with **efficient intra-block communication**, while maintaining independence between blocks for scalability.



Global, Constant and Texture Memory

- Global memory
  - Main means of communicating R/W
  - Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads

GPU
Block 0
Block 1
Shared Memory
Registers
Thread 0
Thread 1
Local Memory
Global Memory
Constant Memory
Texture Memory
CPU

---

1. Global Memory

- **Purpose:** Main memory for reading/writing data between **host (CPU)** and **device (GPU)**.

- **Accessibility:**
  - Visible to **all threads** in all thread blocks.
  - Each thread can read/write to global memory.
- **Location:** On the GPU's device memory (off-chip DRAM).
- **Speed:** Relatively **slow** compared to shared or registers.
- **Persistence:** Exists for the lifetime of the application (until freed).

---

2. Constant Memory

- **Purpose:** Store **read-only constants** initialized by the **host**.
- **Accessibility:**
  - Visible to **all threads**, but **cannot be modified by threads**.
  - Efficient for broadcasting the same value to multiple threads.
- **Location:** On-chip cache (fast access if cached properly).
- **Speed: Faster than global memory** when accessed uniformly.
- **Persistence:** Lifetime of the application.

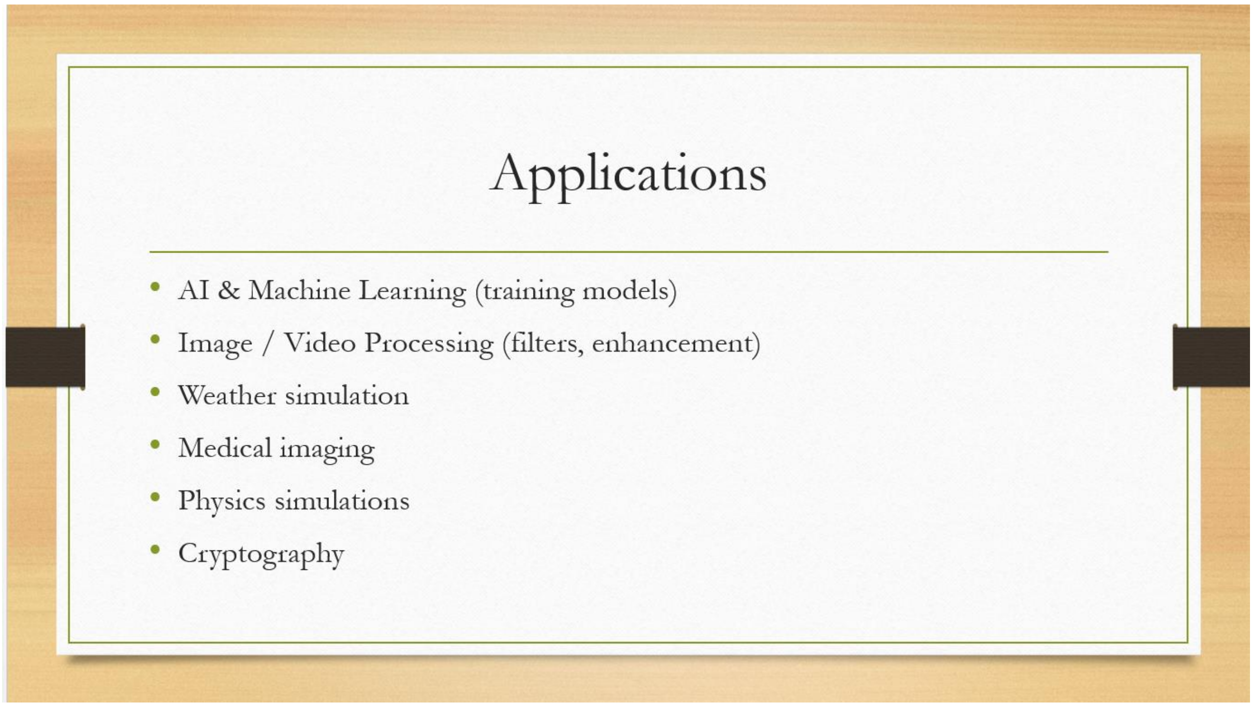
---

3. Texture Memory

- **Purpose:** Optimized for **2D spatial locality** and image processing.
- **Accessibility:**
  - Read-only for threads (usually bound to global memory).
  - Allows hardware interpolation for graphics-like operations.
- **Location:** Cached on-chip.
- **Speed:** Faster than global memory due to caching for spatial locality.
- **Special Use:** Graphics, image filters, or lookup tables.

✅ Key Points

- **Global memory:** R/W by all threads, slow, off-chip.
- **Constant memory:** Initialized by host, read-only, fast if uniformly accessed.
- **Texture memory:** Read-only, optimized for 2D/3D access patterns, cached.



Applications

- AI & Machine Learning (training models)
- Image / Video Processing (filters, enhancement)
- Weather simulation
- Medical imaging
- Physics simulations
- Cryptography

---

1. AI & Machine Learning

- **Use:** Training deep learning models, neural networks.
- **Why CUDA:** Parallel processing speeds up matrix multiplications, backpropagation, and large dataset computations.

---

2. Image & Video Processing

- **Use:** Filters, image enhancement, real-time video editing, computer vision tasks.
- **Why CUDA:** Handles millions of pixels simultaneously; texture memory and shared memory speed up operations.

---

3. Weather Simulation

- **Use:** Climate modeling, storm prediction, computational fluid dynamics.
- **Why CUDA:** Massive parallelism helps simulate multiple regions or data points in parallel.

---

4. Medical Imaging

- **Use:** MRI reconstruction, CT scan processing, 3D visualization.
- **Why CUDA:** Parallel processing accelerates image reconstruction and high-resolution data rendering.

---

5. Physics Simulations

- **Use:** Particle systems, fluid dynamics, molecular simulations.
- **Why CUDA:** Thousands of particles or atoms can be computed in parallel for realistic simulations.

---

6. Cryptography

- **Use:** Hashing, encryption/decryption, brute-force attacks for testing.
- **Why CUDA:** Parallel threads can process multiple encryption keys or blocks simultaneously.
