Here's a **complete, in-depth explanation of GPU** in the context of **Parallel and Distributed Computing**, with examples, architecture, and applications.

GPU: Parallel & Distributed Computing

1. Definition of GPU

A **GPU (Graphics Processing Unit)** is a **specialized processor designed for performing large-scale, highly parallel computations**, originally for graphics rendering but now widely used for general-purpose computing (GPGPU).

Exam-ready one-line definition:

A GPU is a specialized processor optimized for parallel processing, capable of executing thousands of threads simultaneously, making it ideal for high-performance and distributed computing tasks.

---

2. Why GPU is Important

- CPUs have **few cores optimized for sequential tasks**.
- GPUs have **thousands of cores optimized for parallel tasks**.
- Ideal for:
    - Scientific simulations
    - Data analytics
    - Machine learning
    - Image/video processing

Key Idea: GPU accelerates tasks that can be **broken into thousands of small, independent computations**.

---

3. GPU vs CPU

Feature	CPU	GPU
Cores	Few (4–32)	Thousands
Optimization	Sequential processing	Parallel processing
Clock Speed	High	Moderate
Memory	Cache-based, small	High-bandwidth memory (VRAM)

Feature	CPU	GPU
Example Tasks	OS, web browsing	Matrix multiplication, ML, simulations

---

4. GPU Architecture

4.1 Core Components

1. **Streaming Multiprocessors (SMs):**
   - Each SM contains multiple cores.
   - Executes multiple threads simultaneously.
2. **Global Memory:**
   - Accessible by all SMs.
   - High capacity, moderate speed.
3. **Shared Memory:**
   - Local to each SM.
   - Very fast, used for inter-thread communication.
4. **Registers:**
   - Fastest memory per thread.
5. **Warp / Thread Blocks:**
   - Threads organized into **blocks**.
   - Blocks organized into **grids**.

4.2 Parallel Execution Model

- GPU uses **SIMD (Single Instruction, Multiple Data)** model.
- Executes the **same instruction across multiple threads** simultaneously.
- Ideal for **data-parallel tasks**.

---

5. GPU in Parallel & Distributed Computing

5.1 Parallel Computing

- Break computation into thousands of threads.
- Each thread computes a small part of the task.
- Example: Matrix multiplication, vector addition.

5.2 Distributed Computing

- Multiple GPUs can work together across nodes.

- Example: GPU clusters for training deep learning models.
- Communication via frameworks like **NCCL, MPI, or CUDA-aware MPI**.

---

6. GPU Programming Examples

6.1 CUDA (NVIDIA GPU)

Vector Addition Example in CUDA (C++)

```
#include <cuda_runtime.h>
#include <stdio.h>

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

int main(){
    int n = 1024;
    int size = n * sizeof(int);
    int *a, *b, *c;
    cudaMallocManaged(&a, size);
    cudaMallocManaged(&b, size);
    cudaMallocManaged(&c, size);

    for(int i=0;i<n;i++){
        a[i] = i;
        b[i] = i*2;
    }

    vectorAdd<<<(n+255)/256, 256>>>(a, b, c, n);
    cudaDeviceSynchronize();

    printf("Result[0] = %d\n", c[0]);
    cudaFree(a); cudaFree(b); cudaFree(c);
    return 0;
}
```

Explanation:

- `__global__` → function executed on GPU.
- Threads are organized in **blocks**.
- GPU executes **thousands of threads in parallel**.

6.2 Python Example (PyTorch)

```python
import torch

# Create tensors on GPU
a = torch.randn(1000, device='cuda')
b = torch.randn(1000, device='cuda')

# Parallel vector addition on GPU
c = a + b
print(c[:5])
```

- Automatically uses GPU cores for parallel computation.
- Ideal for ML, simulations, and scientific computing.

---

7. Advantages of GPU

- Massive parallelism → thousands of threads
- High throughput for vector/matrix operations
- Ideal for **data-parallel workloads**
- Accelerates ML, AI, simulations, graphics

---

8. Limitations of GPU

- Not good for sequential tasks
- High power consumption
- Memory size is limited compared to CPU
- Requires specialized programming (CUDA, OpenCL, etc.)

---

9. GPU vs CPU vs TPU

Feature	CPU	GPU	TPU
Cores	Few	Thousands	Thousands (ML optimized)
Best For	Sequential tasks	Parallel data tasks	Matrix operations / Deep learning
Memory	General RAM	VRAM	High bandwidth memory
Example	OS, apps	ML, simulations	TensorFlow training

---

10. GPU in Distributed Systems

- Multi-GPU nodes connected via **NVLink / PCIe**

- Clusters communicate via **MPI or NCCL**
- Example: **Distributed deep learning training**:
    - Each GPU computes gradients on a data shard
    - Gradients aggregated across nodes
    - Model updated in parallel

---

11. GPU Use Cases in Parallel & Distributed Computing

- Deep learning / AI training (TensorFlow, PyTorch)
- Scientific simulations (weather, molecular modeling)
- Image/video processing
- Big data analytics
- Cryptocurrency mining

---

12. GPU vs Hadoop / AWS / OpenStack / Condor

Feature	GPU	Hadoop	AWS	OpenStack	Condor
Type	Processor	Framework	Cloud	Cloud	Job scheduler
Parallelism	Thread-level parallelism	MapReduce tasks	EC2/Lambda tasks	VMs, containers	Independent tasks
Distributed	Multi-GPU clusters	Across cluster nodes	Across regions	Across nodes	Across idle machines
Fault Tolerance	Via software frameworks	Built-in replication	Multi-zone	Multi-zone replication	Job migration

---

13. Exam-Oriented Summary

Keywords:

- GPU, parallel cores, SIMD, CUDA, VRAM, thread blocks, multi-GPU cluster

One-Paragraph Answer:

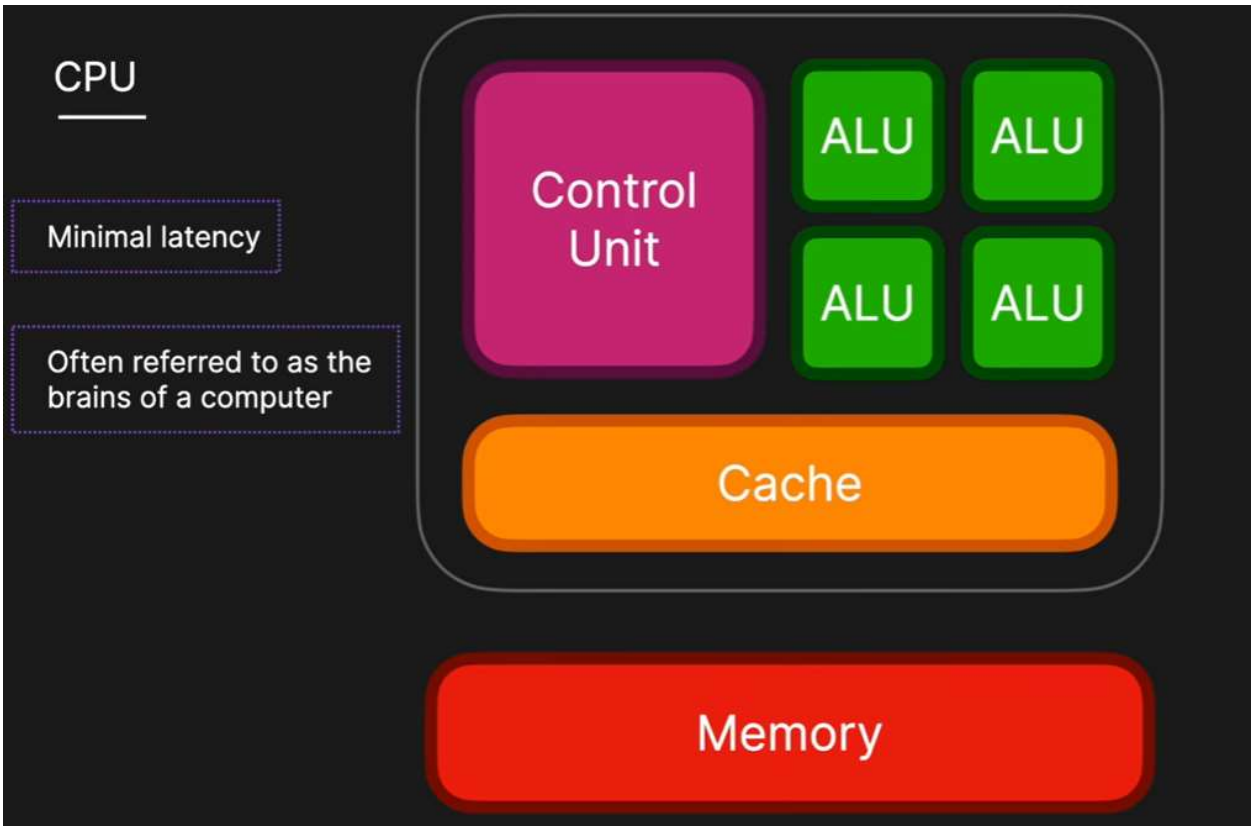
A GPU is a specialized processor designed for highly parallel computations, capable of executing thousands of threads simultaneously. It is widely used in parallel and distributed computing for scientific simulations, machine learning, and big data processing. GPUs are organized into cores, thread blocks, and grids, supporting SIMD execution. Multiple GPUs can be clustered for distributed tasks, achieving high throughput and scalability.

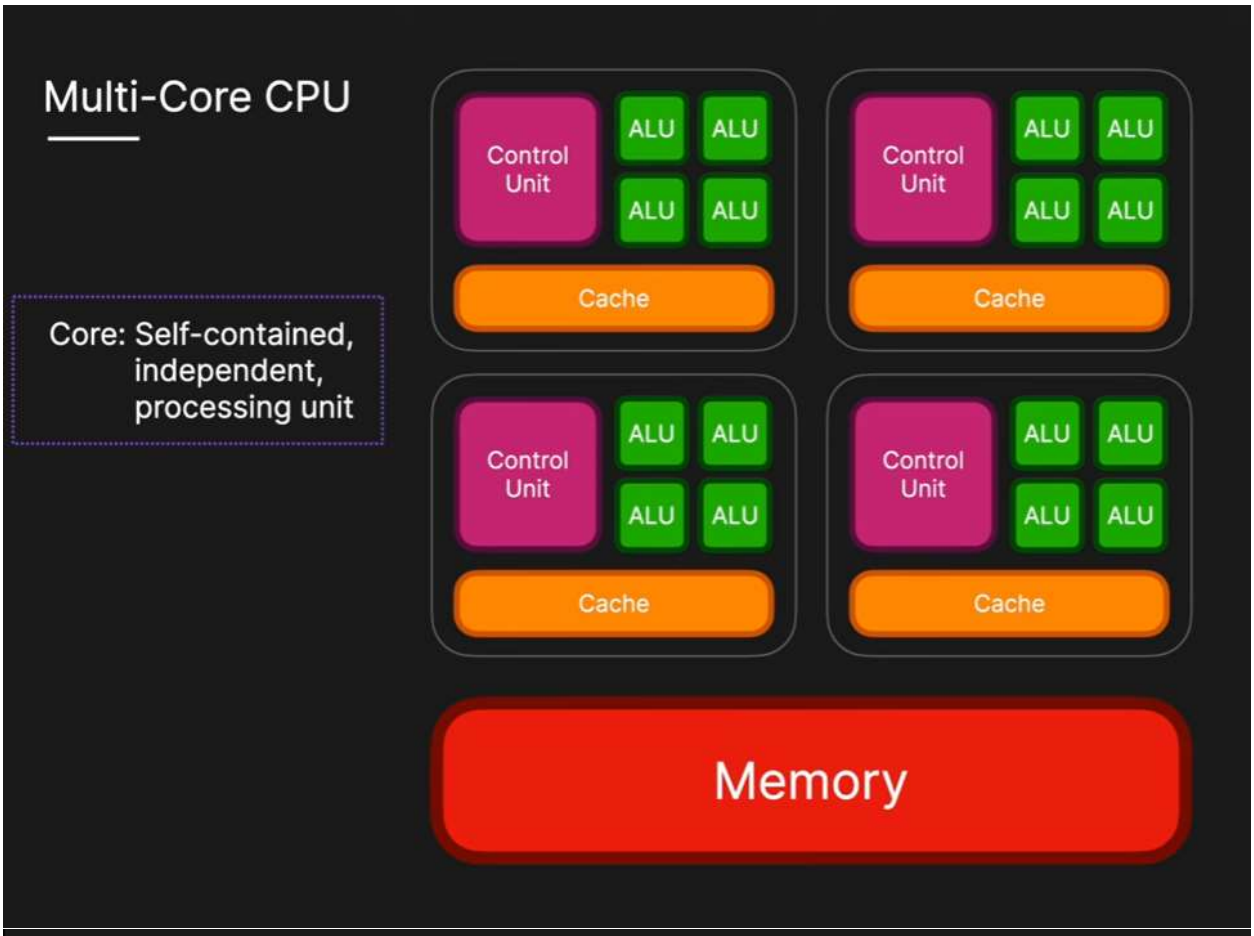
The Central Processing Unit (CPU) and Graphics Processing Unit (GPU) serve different purposes and are optimized for specific tasks.

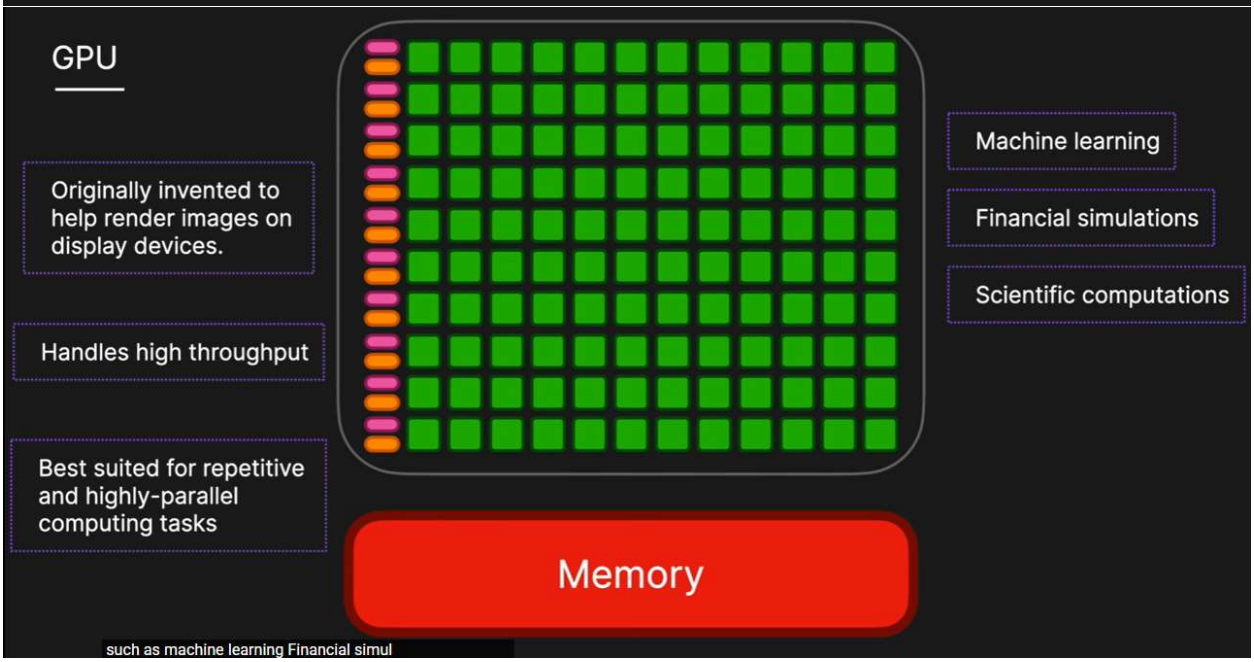
CPU (Central Processing Unit):
The CPU is designed for general-purpose tasks and can handle a variety of different processes. It excels at sequential processing, such as calculating a single matrix quickly. You can think of the CPU like a smartphone—it is capable of performing many different tasks but is not specialized for high-end production work.

GPU (Graphics Processing Unit):
The GPU, on the other hand, is designed for specific, graphics-related tasks like gaming. It is optimized for parallel processing, meaning it contains many cores that can perform independent calculations simultaneously. This makes the GPU better suited for tasks that involve repeated, similar calculations, such as rendering pixels in games where values continuously change and are recalculated. An analogy for a GPU is a professional cinema camera built for high-quality video production, unlike a smartphone. Demonstrations from Nvidia show that a GPU can paint images much faster than a CPU due to its parallel processing capabilities.

CPU vs. GPU Interchangeability:
CPUs and GPUs cannot currently be used interchangeably because of compatibility issues with motherboards and other peripherals. Even though it is theoretically possible with redesigned hardware, it would be highly inefficient, as a CPU would perform poorly as a GPU and vice versa.


CPU
Minimal latency
Often referred to as the brains of a computer
Control Unit
ALU
ALU
ALU
ALU
Cache
Memory


Multi-Core CPU
Core: Self-contained, independent, processing unit
Control Unit
ALU
ALU
ALU
ALU
Cache
Control Unit
ALU
ALU
ALU
ALU
Cache
Control Unit
ALU
ALU
ALU
ALU
Cache
Control Unit
ALU
ALU
ALU
ALU
Cache
Memory


GPU
Originally invented to help render images on display devices.
Handles high throughput
Best suited for repetitive and highly-parallel computing tasks
Machine learning
Financial simulations
Scientific computations
Memory
such as machine learning Financial simul

Feature	CPU (Central Processing Unit)	GPU (Graphics Processing Unit)	Similarities
Definition / Role	The core computational unit in a server or computer. Handles all types of computing tasks required for operating systems and applications.	A specialized hardware component designed to efficiently handle complex mathematical operations that run in parallel.	Both are hardware units that make a computer work and are often considered the "brain" of a computing device.
Purpose / Origin	General-purpose computing; required to run essential software tasks.	Created to offload compute-intensive workloads like graphics and animation from CPUs, especially for parallel processing.	Both perform computations and logical operations.
Processing	Excels at sequential processing; handles a wide variety of tasks.	Excels at parallel processing; breaks repetitive tasks into smaller components to process simultaneously.	Both have cores that execute computations and logical functions.
Cores	Multi-core CPUs exist today (initially single-core).	Multi-core GPUs designed for thousands of parallel computations.	Both have cores to perform computations.
Memory	Uses internal memory (cache) to improve performance and access data quickly.	Uses internal memory (cache) for fast data access and processing efficiency.	Both utilize internal memory to speed up processing.
Control Unit	Synchronizes processing tasks and determines processing speed through electric pulse frequency.	Synchronizes processing tasks and contributes to high-speed parallel computations.	Both have control units to manage and coordinate tasks.
Workloads	General-purpose; runs operating systems, applications, and standard computing tasks.	Compute-intensive workloads like graphics, AI, and machine learning.	Both are essential for computing performance in their respective roles.
Dependency	A computer cannot operate without a CPU.	Supports the CPU by handling parallel computations.	Both are integral hardware components of a computer system.

Core Differences

	CPU	GPU
Function	Generalized component that deals with the main processing functions of a computer	Specialized component that is great for parallel computing
Processing	Runs processes serially	Runs processes in parallel
Design	Fewer more powerful cores	More cores (less powerful than CPU cores)
Emphasis	Low latency	High throughput
Use case	General purpose computing devices	High-performance computing devices

use cases CPUs are used for general

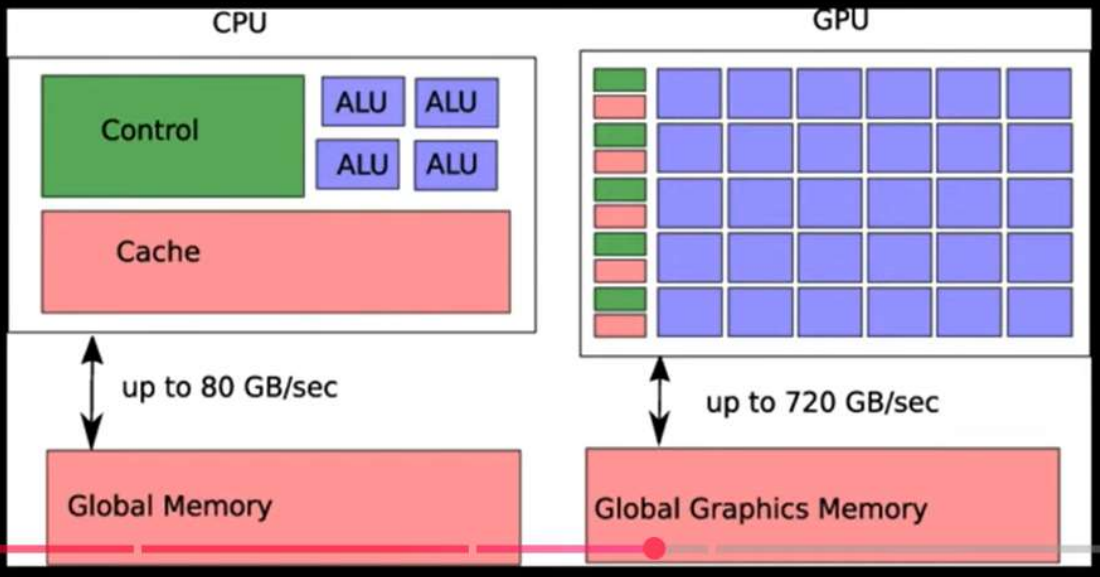
Key Understandings

- A CPU can never be fully replaced by a GPU
- Fundamental difference
  - CPUs are ideal for performing sequential tasks
  - GPUs use parallel processing

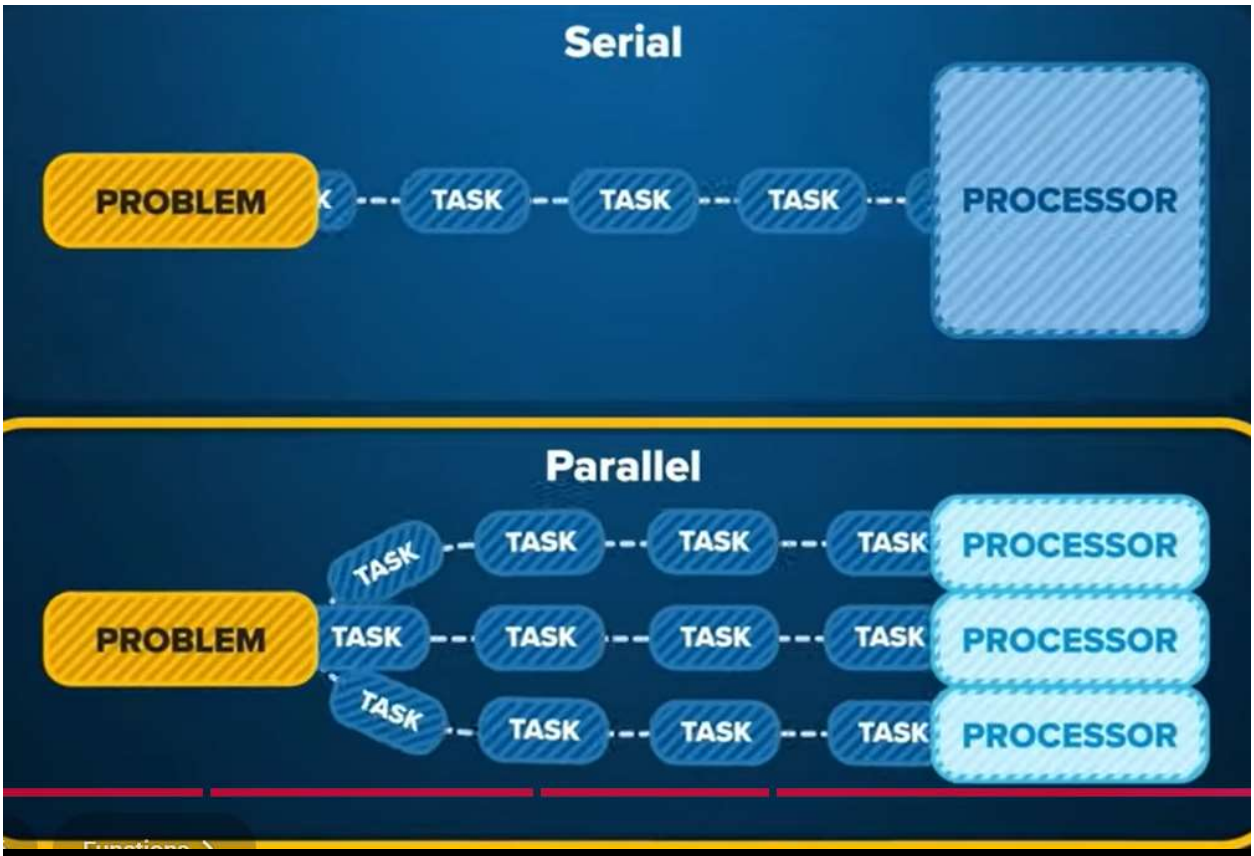
Similarities between GPU and CPU

Core, Memory & Control Unit

Control Unit



4:26 Control Unit >



Key Differences: CPU and GPU

	CPU	GPU
Function	Generalised component that handles main processing functions of a server	Specialized component that excels at parallel computing
Processing	Designed for serial instruction processing	Designed for parallel instruction processing
Design	Fewer, more powerful cores	More cores than CPU, but less powerful than CPU cores
Application	General purpose computing applications	High-performance computing applications

Comprehensive Overview of How Graphics Cards Work

Computational Power of Graphics Cards

Modern graphics cards perform an immense number of calculations. For example, the NVIDIA

3090 can execute around 36 trillion calculations per second, which illustrates the vast computational power of these devices.

GPU vs. CPU Differences
GPUs and CPUs are designed for different purposes. A GPU can be thought of as a cargo ship—it performs massive numbers of calculations across large datasets at a slower rate. In contrast, a CPU is like a jumbo jet—it processes smaller amounts of data faster and is more flexible for running operating systems and various applications.

GPU GA102 Architecture
The GA102 GPU, found in NVIDIA's 30-series graphics cards, has a hierarchical architecture. It consists of Graphics Processing Clusters (GPCs), Streaming Multiprocessors (SMs), warps, CUDA cores, tensor cores, and ray tracing cores, each organized to efficiently handle complex computational tasks.

GPU Manufacturing and Binning
Different graphics card models, such as the 3080 and 3090, can use the same GA102 chip design. This is achieved through a process called "binning," where chips with manufacturing defects have certain areas deactivated, leading to variations in core counts and overall performance between models.

CUDA Core Design
A CUDA core performs fundamental operations such as fused multiply and add (FMA), which are essential for graphics rendering. Each CUDA core is carefully designed to execute these operations efficiently across large datasets.

Graphics Card Components
Beyond the GPU itself, graphics cards include essential components such as the printed circuit board (PCB), display ports, power connectors, PCIe interface, voltage regulator module, and a heat sink system for cooling.

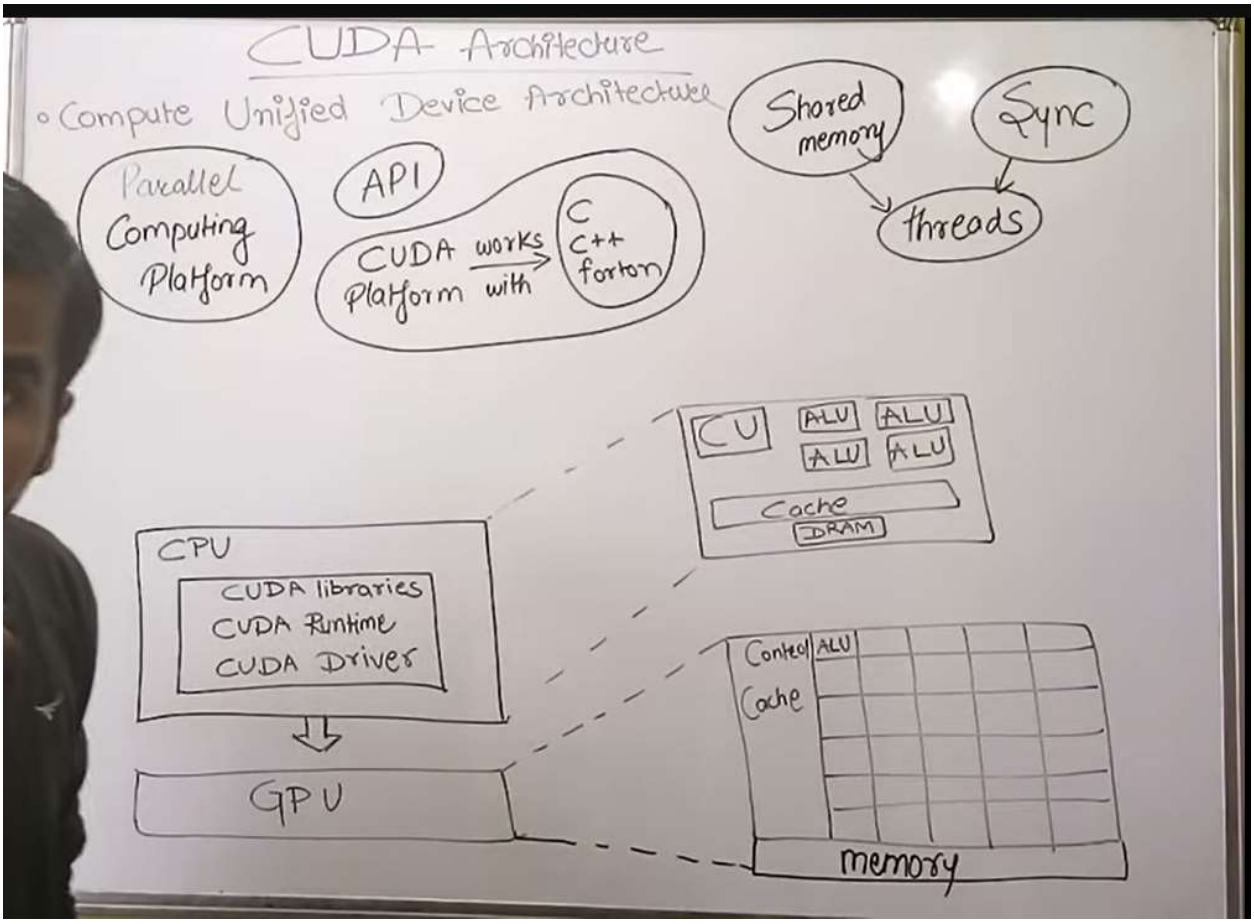
Graphics Memory (GDDR6X & GDDR7)
Graphics memory is critical for feeding data to the GPU. The GDDR6X SDRAM in the 3090 continuously supplies terabytes of data, while the newer GDDR7 memory uses multi-voltage level encoding schemes (PAM3) to further increase data transfer rates.

Micron's Role in Memory Innovation
Micron contributes to advancements in memory technology, producing High Bandwidth Memory (HBM) like HBM3E, which is especially important for AI-focused chips.

Computational Architecture: SIMD and SIMT
GPUs handle "embarrassingly parallel" problems using computational principles like Single Instruction Multiple Data (SIMD) and Single Instruction Multiple Threads (SIMT). SIMD applies the same instruction to large amounts of data, ideal for video game rendering, while SIMT allows individual threads to progress at different rates, providing greater flexibility.

Bitcoin Mining with GPUs
GPUs were initially used for Bitcoin mining because they efficiently executed thousands of iterations of the SHA-256 hashing algorithm, taking advantage of their parallel processing capability.

Tensor Cores for AI and Neural Networks
Tensor cores are specialized GPU cores designed for matrix multiplication and addition, operations that are fundamental to neural networks and generative AI.



Accelerator-Based Computing and GPU Applications

Accelerator-based computing focuses on improving application performance by using specialized hardware or software, with GPUs being a primary example. The approach builds on concepts from parallel and distributed computing, including sequential programming and multi-core CPU architectures. While multi-core CPUs increased computational power, they faced limitations due to memory bottlenecks and communication overhead between threads, leading to diminishing returns in speedup.

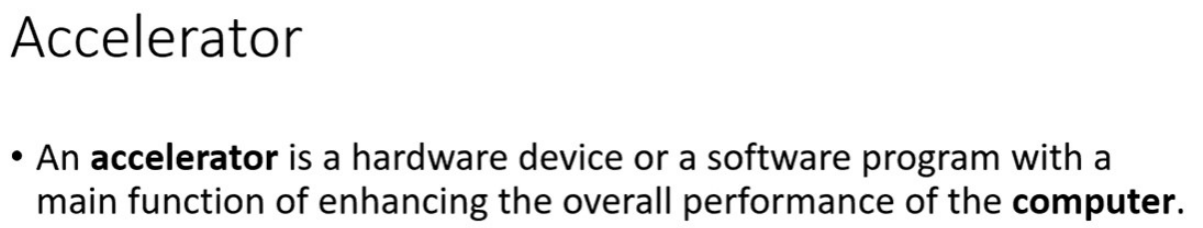
Accelerators
Accelerators are designed to handle computationally intensive parts of an application, offloading these tasks from the CPU. Common types of accelerators include GPUs, FPGAs, and DSPs. In a heterogeneous system, the CPU handles sequential tasks efficiently, while the GPU processes parallel tasks using its numerous cores. This combination leverages the strengths of both types of processors.

GPU Advantages
GPUs provide significantly higher computational performance than CPUs for parallelizable applications. Examples include:

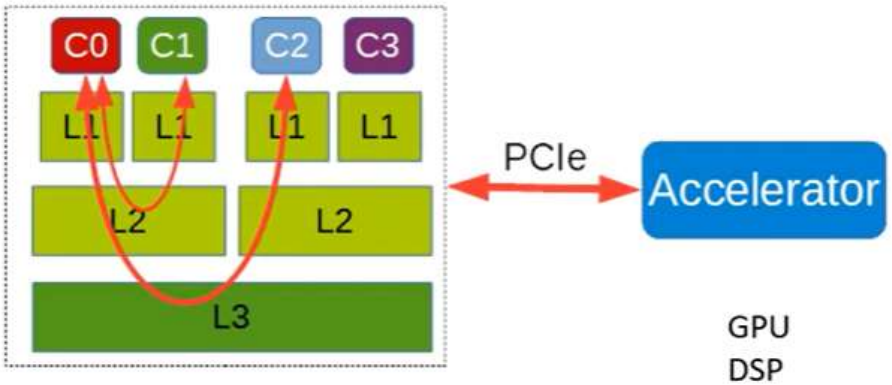
- **Vector Addition:** GPUs perform these operations much faster than CPUs, even when accounting for communication overhead.
- **Matrix Multiplication:** As matrix sizes grow, GPUs show a greater performance advantage over CPUs.
- **Planet and Particle Simulations:** GPUs provide massive speedups compared to CPUs due to their parallel processing capabilities.

Applications
Due to their high performance, GPUs are widely used in supercomputers and are highly recommended for graphics-intensive applications, video games, and machine learning models that require processing large datasets.

Future Work
Programming these accelerators involves specialized libraries and directives such as CUDA and OpenACC, which enable developers to efficiently harness GPU power for high-performance computing tasks.

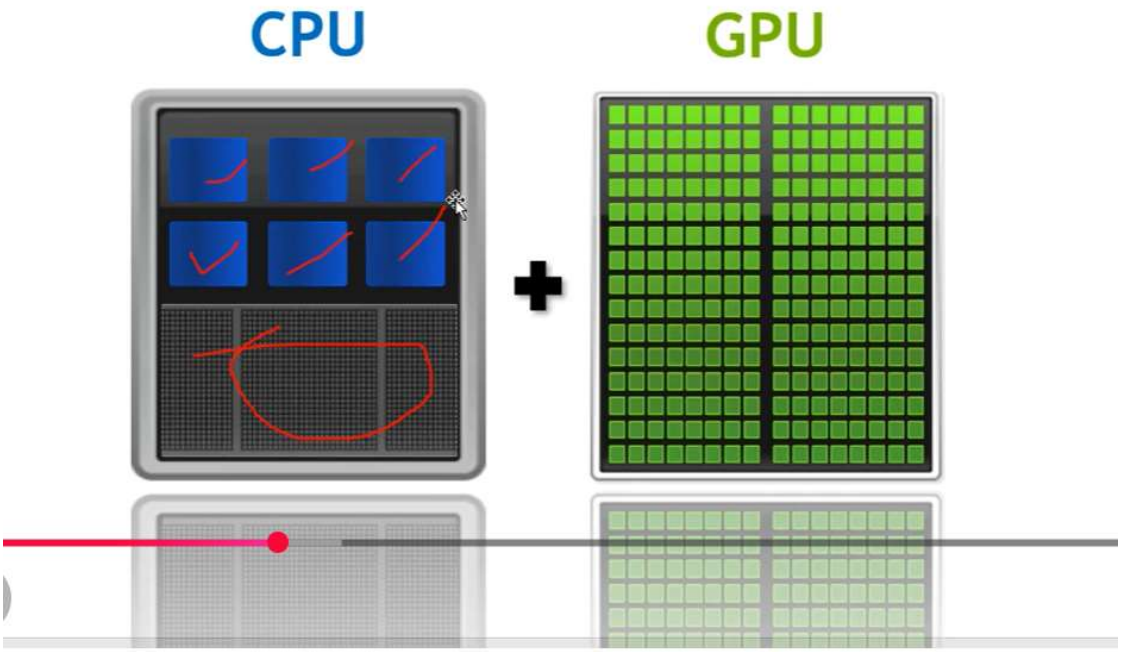


Accelerator-based computing

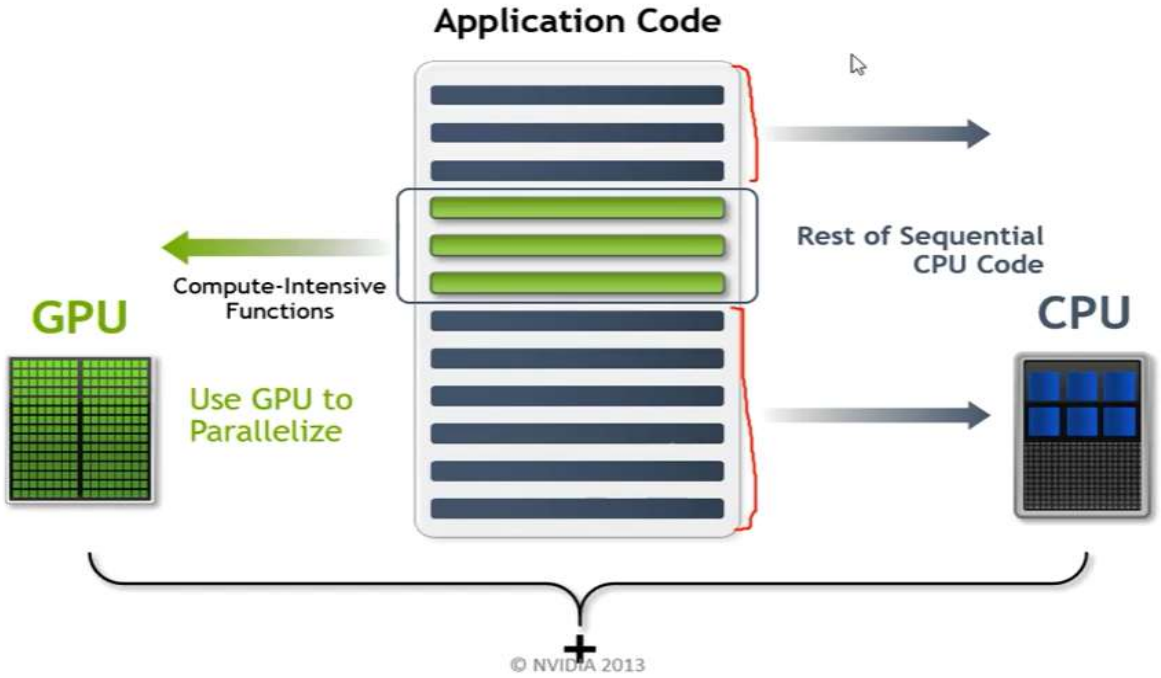


GPU
DSP
FPGA
QPU
TPU

Add GPUs: Accelerate Science Applications

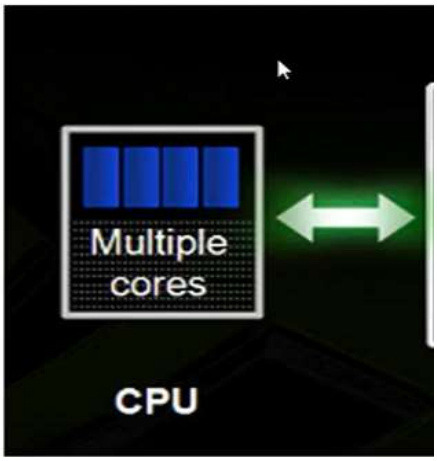


Small Changes, Big Speed-up



CPU vs GPU

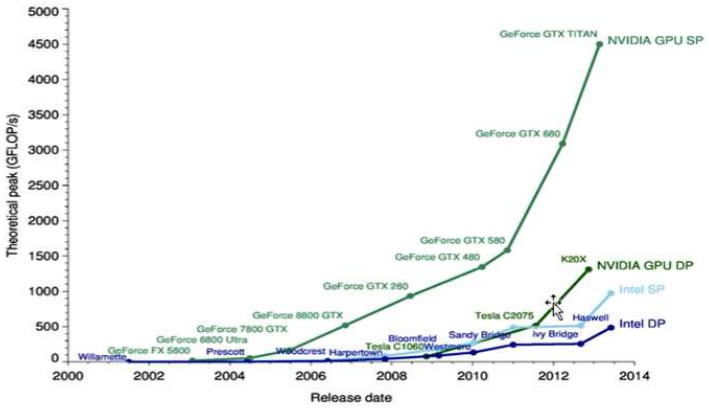
- GPU has higher parallelism than CPU
- CPU has better serial processing capabilities
- CPU-GPU comprise a heterogeneous system



- Best performance is using both CPU & GPU

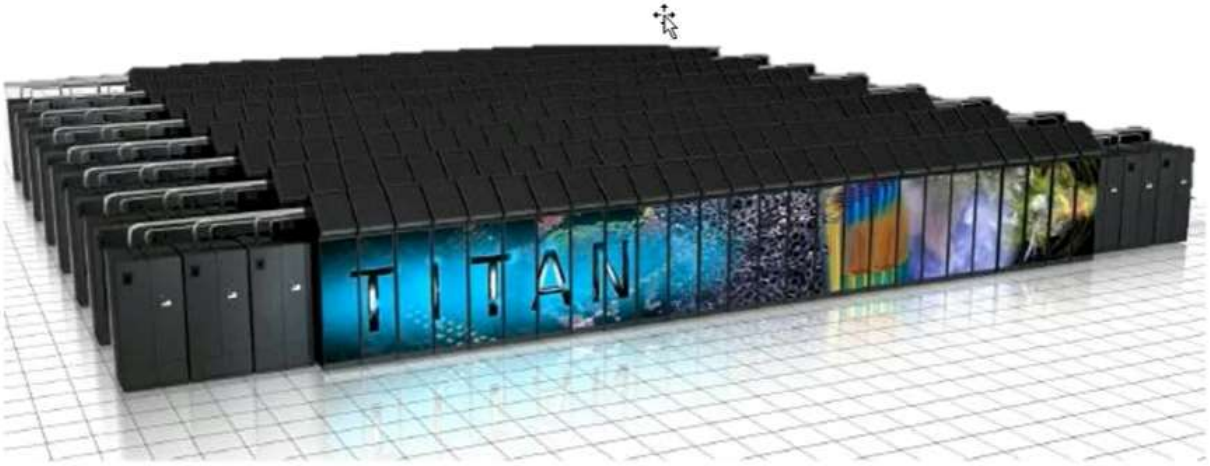
CPU vs GPU

- GPU peak performance much higher than CPU
- Only achievable for highly parallel applications
  - Graphics
  - Scientific
  - Many others
- Made possible by many small GPU cores



CPU vs GPU

- Many of the Top 10 supercomputers use GPUs

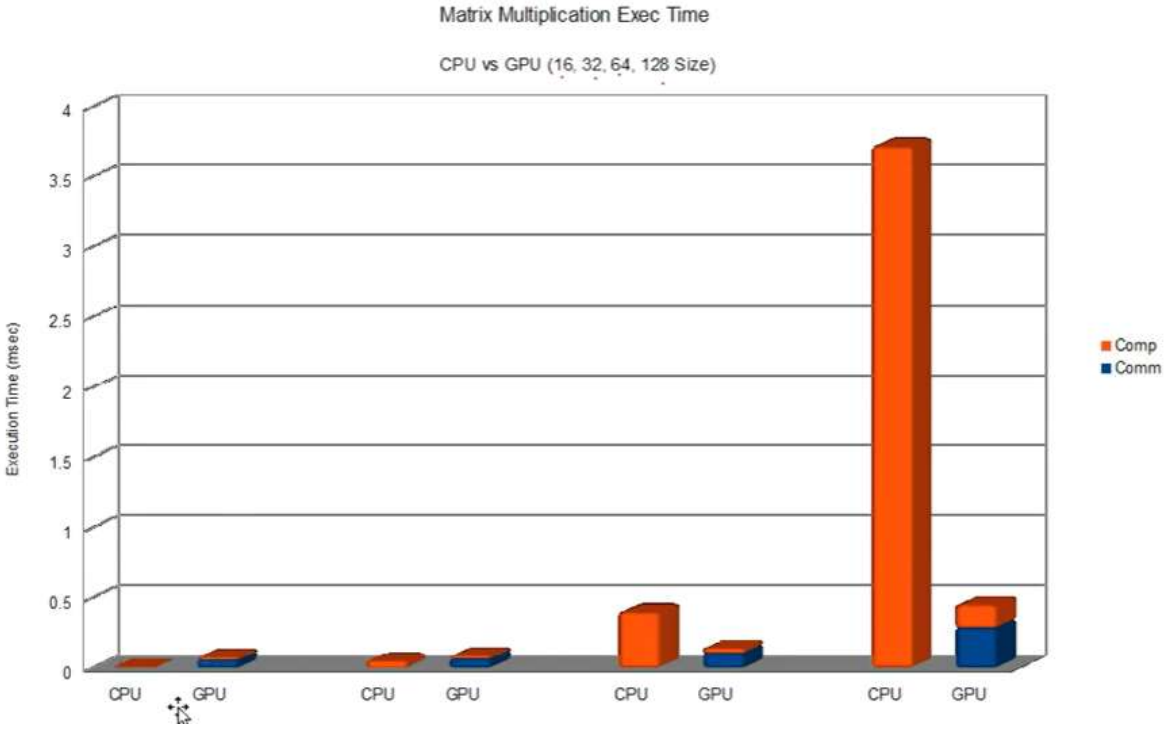


Titan supercomputer

Vector Add Example: Timings

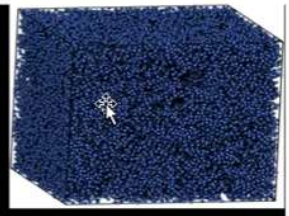
- CPU : 93 msec
- GPU (CUDA) : 51 msec (4 + 47)

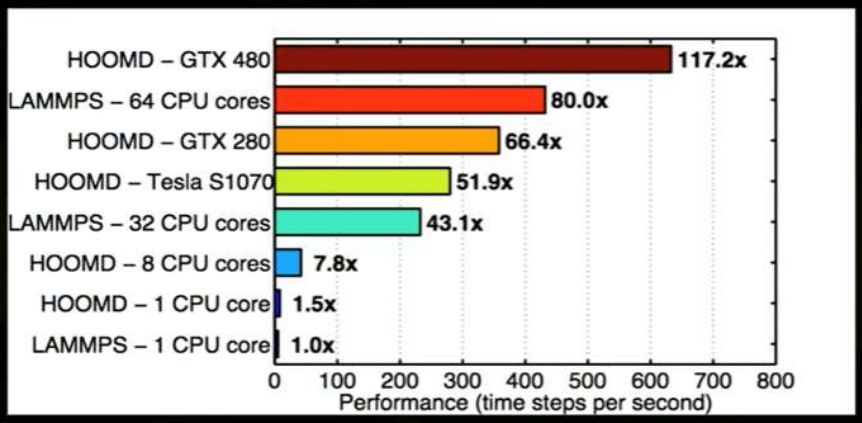
Matrix Multiplication Results



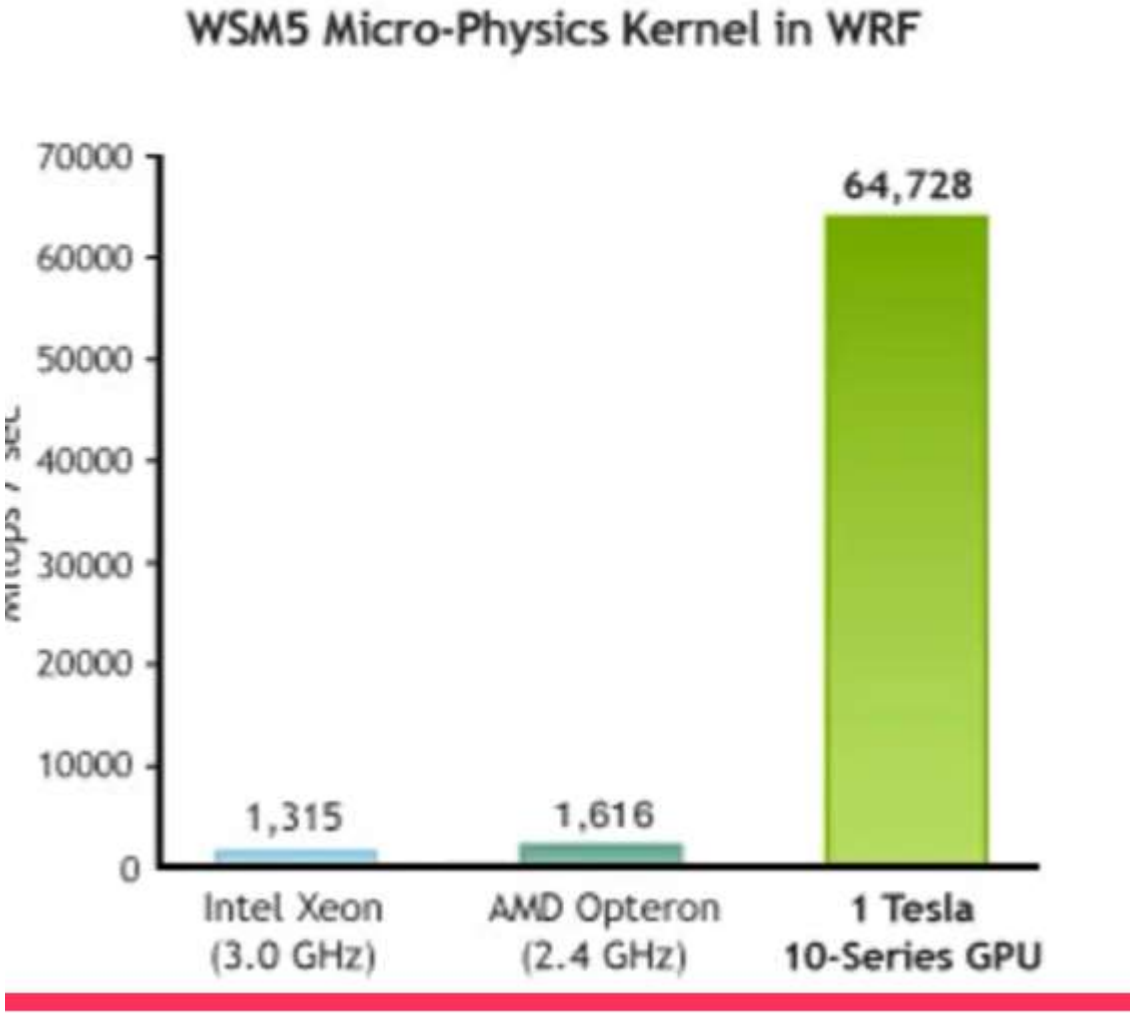
HOOMD-blue Benchmark

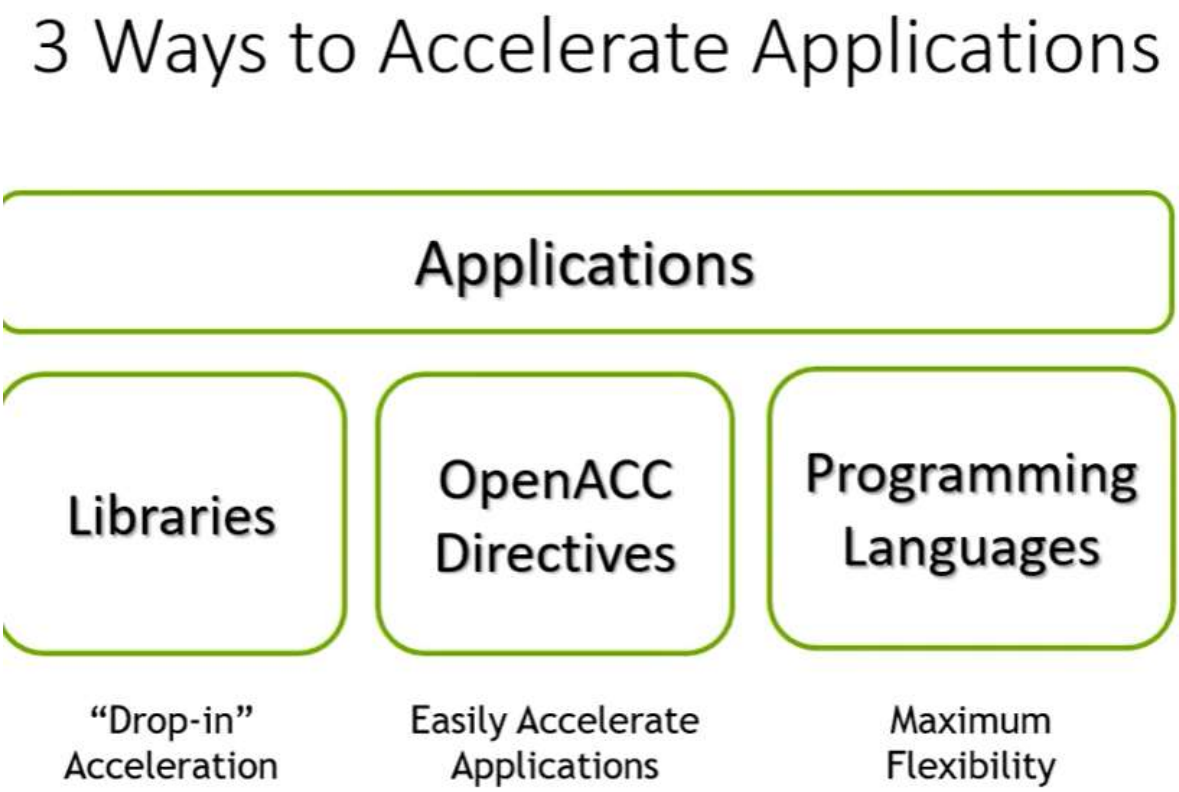
- 64,000 particle Lennard-Jones fluid simulation
- representative of typical performance gains





*CPU: Intel Xeon E5540 @ 2.53GHz


WSM5 Micro-Physics Kernel in WRF
70000
60000
50000
40000
30000
20000
10000
0
64,728
1,315
1,616
Intel Xeon
(3.0 GHz)
AMD Opteron
(2.4 GHz)
1 Tesla
10-Series GPU



Understanding GPUs and Their Functionality

What is a GPU?
A GPU (Graphics Processing Unit) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate image creation for display. GPUs are typically placed on a video card and include their own dedicated memory.

Why GPUs Are Needed
Conventional CPUs often struggle with high-intensity graphics applications, such as modern video games and HD movies, due to the massive number of pixels and frames that must be processed. Graphics can be raster-based, where images lose clarity when zoomed, or vector-based, where images maintain clarity through rule-based definitions. Modern applications increasingly use vector graphics, which demand more computational power.

Processing Requirements for Graphics
High-intensity graphics require immense processing capabilities beyond what even advanced CPUs can provide. Shader programs, which use specialized programming languages, define and transform objects, vertices, and pixels within an image. GPUs are essential for applications where **high throughput**—the ability to process many data units simultaneously—is more critical than latency.

CPU vs. GPU

- **CPU:** Contains a few cores optimized to reduce instruction latency. CPUs are excellent for serial processing and managing data dependencies.
- **GPU:** Contains hundreds to thousands of simpler cores focused on high throughput. GPUs excel at parallel processing, breaking jobs into independent tasks. Most programs use both CPU and GPU code, with the CPU offloading parallelizable tasks to the GPU.

Flynn's Classification of Computer Architectures
Flynn's taxonomy categorizes architectures based on instruction and data streams:

- **SISD (Single Instruction, Single Data):** Conventional uniprocessors.
- **MISD (Multiple Instruction, Single Data):** Rarely used.
- **SIMD (Single Instruction, Multiple Data):** Vector processors perform the same operation on many data items.
- **MIMD (Multiple Instruction, Multiple Data):** Multi-core computers that execute multiple instructions on multiple datasets concurrently.

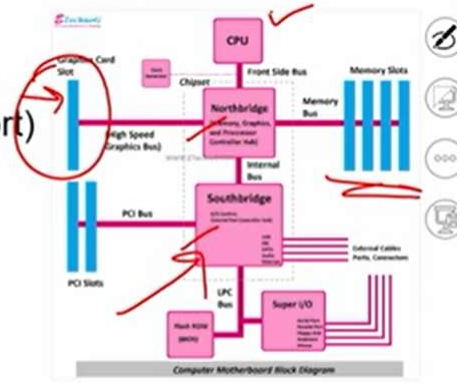
Exploiting Parallelism and SIMT
GPUs primarily operate using the **SIMT (Single Instruction, Multiple Threads)** model, which extends SIMD with multi-threading. In SIMT, each thread executes the same code but on a different piece of data. Sequential loops, such as matrix addition, can be vectorized so that independent iterations run in parallel, taking advantage of the GPU’s architecture.

Warp and Fine-Grained Multi-Threading
A warp is a group of threads, typically 32, that execute the same instruction and are dynamically grouped by the GPU hardware. GPUs use warp-level, fine-grained multi-threading to interleave the execution of multiple warps on the same pipeline. This maximizes utilization of functional units and enables efficient parallel processing of numerous data elements, such as pixels during image transformations.

-

What is a GPU?

- Graphics Processing Unit is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to display.
- Typically placed on a video card, which contains its own memory and display interfaces(HDMI, DVI, VGA, etc)
- Video card connected to motherboard through PCI-Express or AGP(Accelerated Graphics Port)
- Northbridge chip enables data transfer between the CPU and GPU



Why GPU ?

Motivation Example-1:

- Consider a 1920 x 1080 HD display, Refresh rate: 60 frames/second
- Roughly 125 million pixels have to be processed per second
- A 3 GHz processor with an IPC of 2 (medium to high ILP) can process 6 billion instructions per second.
- 48 instructions per pixel.
- Not enough for most high-intensity games
- Graphics effects, HD movies

Why GPU ?

Motivation Example-2:

Vector Graphics

Raster Graphics

Raster graphics

- The image was represented as an array of pixels.
- Very simple rules were used to create it.
- Earlier systems were simply showing images on the screen.

Why GPU ?

Motivation Example-2:

Vector Graphics

Raster Graphics

Vector graphics

- The programmer creates high level objects: shades, textures, characters,
- Images are created from basic rules specified by programmer
- The system generates the images: vector graphics
- If we zoom the image, clarity is not lost

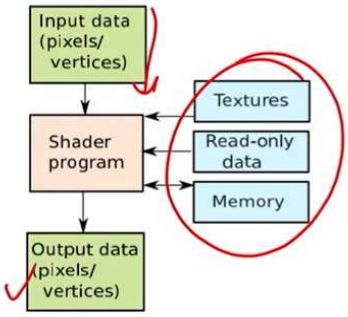
Why GPU ?

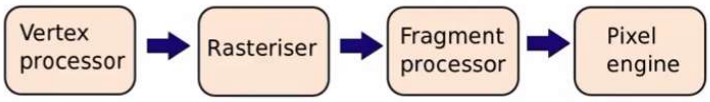
Processing Requirements

- The processing requirements for games, high-intensity graphics is huge
- Aggressive OOO processors (even multicore ones) are insufficient.
- We need shader programs.
  - Custom language to work on objects, vertices, and pixels
  - Apply transformations to images: rotation, skewing, etc.
  - Apply effects: textures, shading, and illumination

Shader Programs

- We need shader programs.
  - Custom language to work on objects, vertices, and pixels
  - Apply transformations to images: rotation, skewing, etc.
  - Apply effects: textures, shading, and illumination





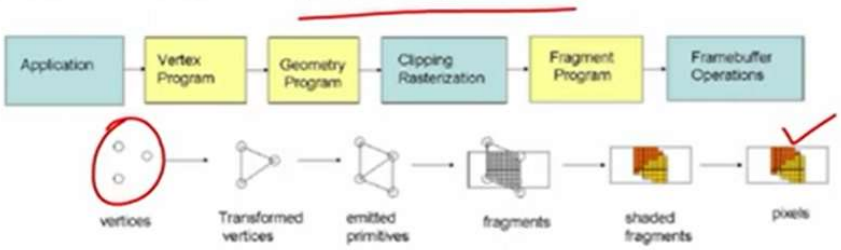
Graphics Pipeline

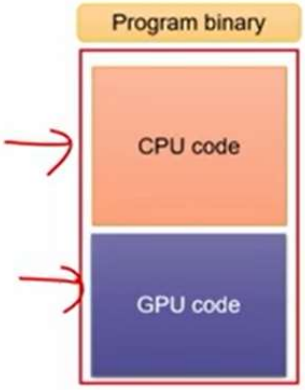
Why GPU ?

- Throughput more important than latency
- High throughput needed for the huge amount of computations required for graphics
- Extremely parallel- different pixels and elements of the image can be operated on independently
- Hundreds of cores executing at the same time to take advantage of this fundamental parallelism

CPU – GPU Interaction

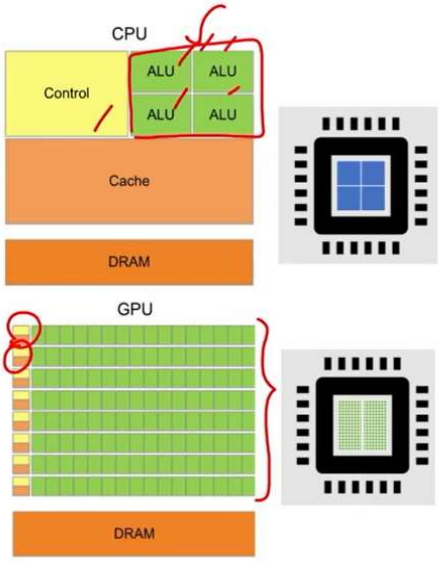
- GPU houses a graphics pipeline





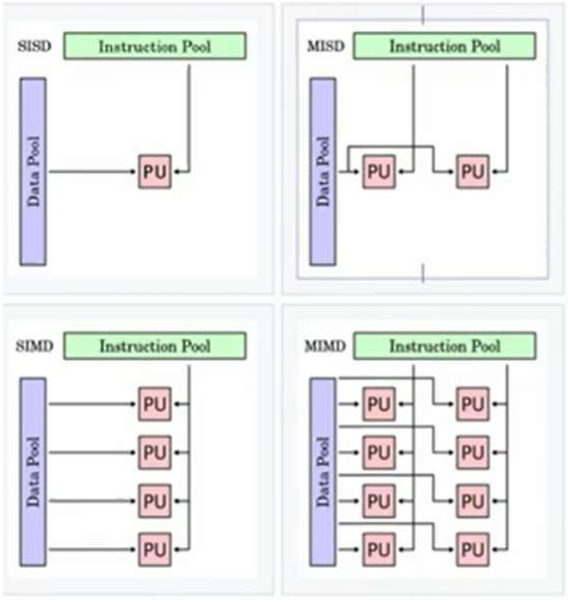
- GPU programs are written in two parts
  - One part runs on a normal CPU
  - Other part runs on the GPU (compiled in a GPU-specific ISA)

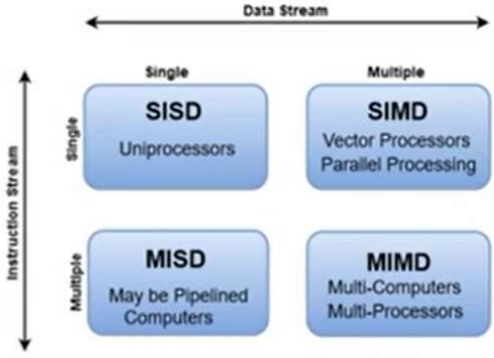
CPU vs GPU



CPU	GPU
Central Processing Unit	Graphics Processing Unit
4-8 Cores	100s or 1000s of Cores
Low Latency	High Throughput
Good for Serial Processing	Good for Parallel Processing
Quickly Process Tasks That Require Interactivity	Breaks Jobs Into Separate Tasks To Process Simultaneously
Traditional Programming Are Written For CPU Sequential Execution	Requires Additional Software To Convert CPU Functions to GPU Functions for Parallel Execution

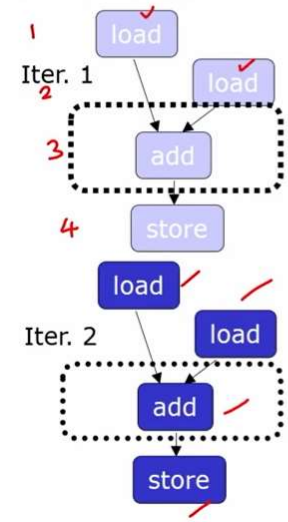
Flynn's Classification





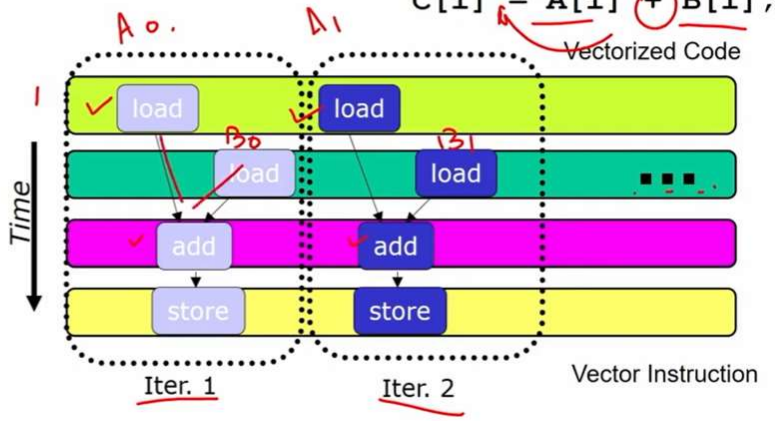
Exploiting Parallelism

Scalar Sequential Code




```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

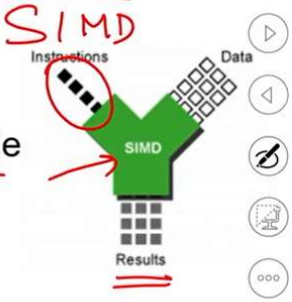
Vectorized Code



- Vectorization : Compile-time reordering of operation sequencing
- Requires extensive loop dependence analysis

Vector Machine - Summary

- Vector/SIMD machines are good at exploiting regular data-level parallelism
  - Same operation performed on many data elements
  - Improve performance, (no intra-vector dependencies)
- Performance improvement limited by vectorizability of code
  - Scalar operations limit vector machine performance
  - Ref: Amdahl's Law
- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD



Programming vs Execution Model

Programming Model refers to how the programmer expresses the code

- Ex: Sequential (von Neumann), Data Flow (SPMD)

Execution Model refers to how the hardware executes the code underneath

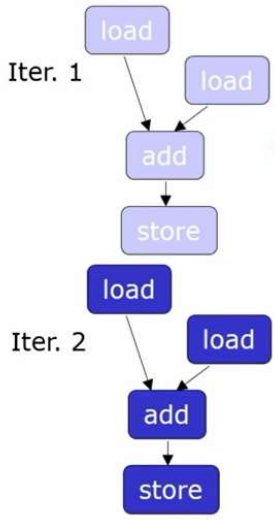
- Ex: Out-of-order execution, Vector processor, Multithreaded processor

Execution Model can be very different from the Programming Model

- Ex: von Neumann model implemented by an OoO processor
- Ex: SPMD model implemented by a SIMD processor (a GPU)

Programming vs Execution Model

Scalar Sequential Code



Model 1: Sequential (SISD)

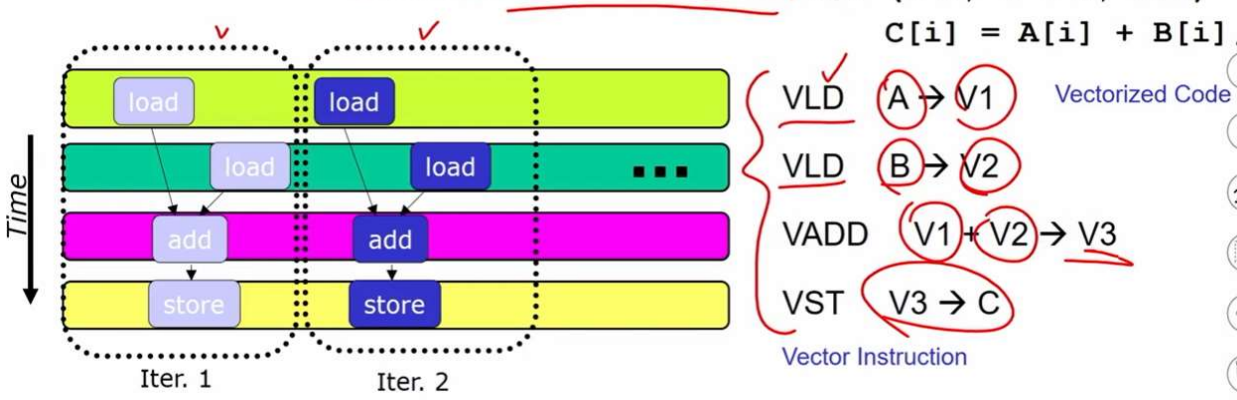
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i]
```

- Pipelined processor
- Out-of-order execution processor
- Independent instructions executed when ready
- Different iterations are present in the instruction window and can execute in parallel in multiple functional units
- Loop is dynamically unrolled by the hardware
- Superscalar or VLIW processor
- Can fetch and execute multiple instructions per cycle

Programming vs Execution Model

Model 2: Data Parallel (SIMD)

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



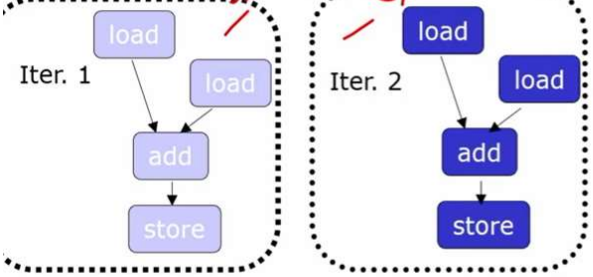
- Each iteration is independent
- Compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Programming vs Execution Model

Model 3: Multithreadded

Scalar Sequential Code

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



....

This can be realized on

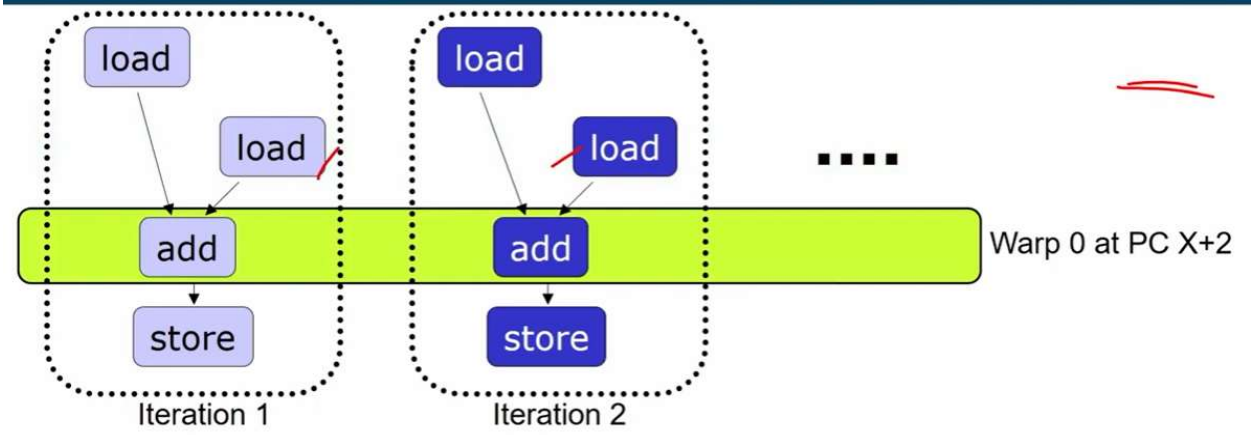
- SPMD: Single Program Multiple Data
- Single Instruction Multiple Thread

- Each iteration is independent
- Programmer or compiler generates a thread to execute each iteration.
- Each thread does the same thing (but on different data)

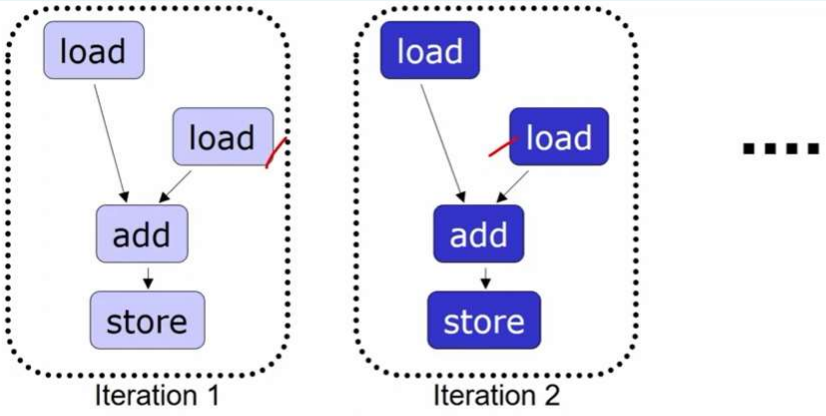
GPU is a SIMT Machine

- Single instruction, multiple threads (SIMT) is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading.
- Each thread executes the same code but operates a different piece of data
- Each thread has its own context (can be /restarted/executed independently)
- A set of threads executing the same instruction are dynamically grouped into a warp (wavefront) by the hardware

SIMT Illustration



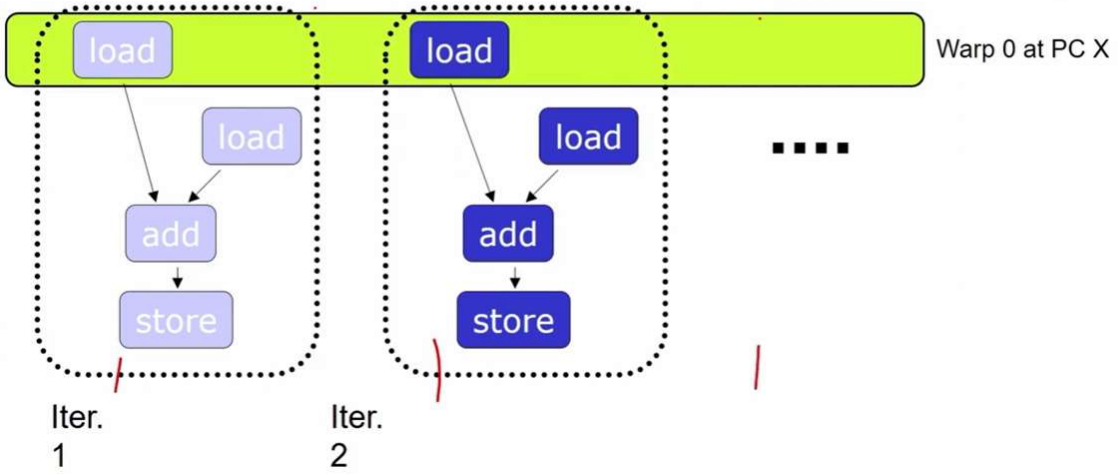
SIMT Illustration



- **Warp:** A set of threads that execute the same instruction at the same PC.
- Programmer or compiler generates a thread to execute each iteration.
- Each thread does the same thing (but on different data)

Multithreaded Warps

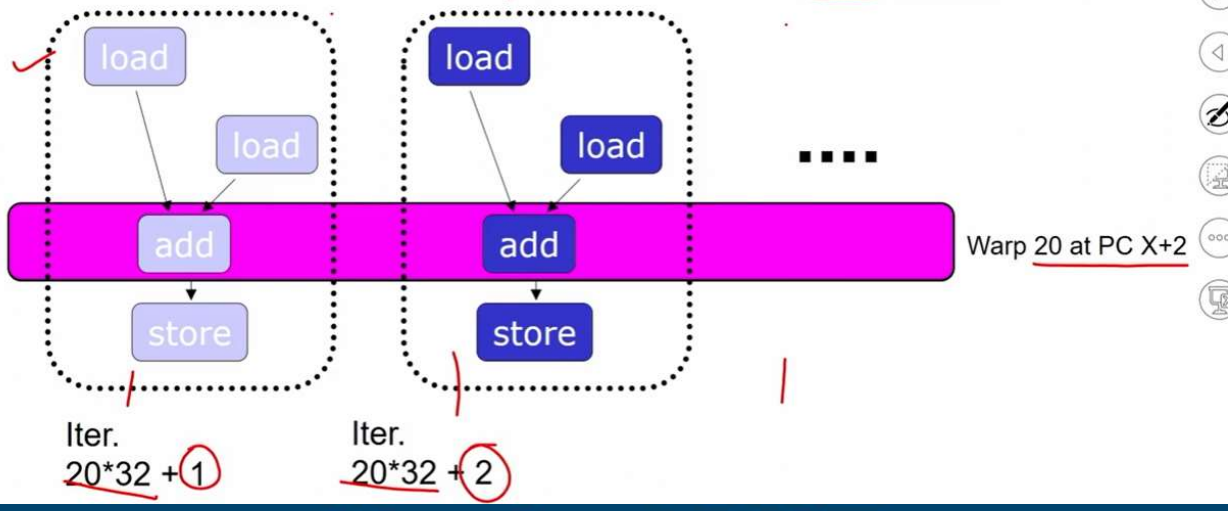
- Warp has 32 threads, 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading



Multithreaded Warps

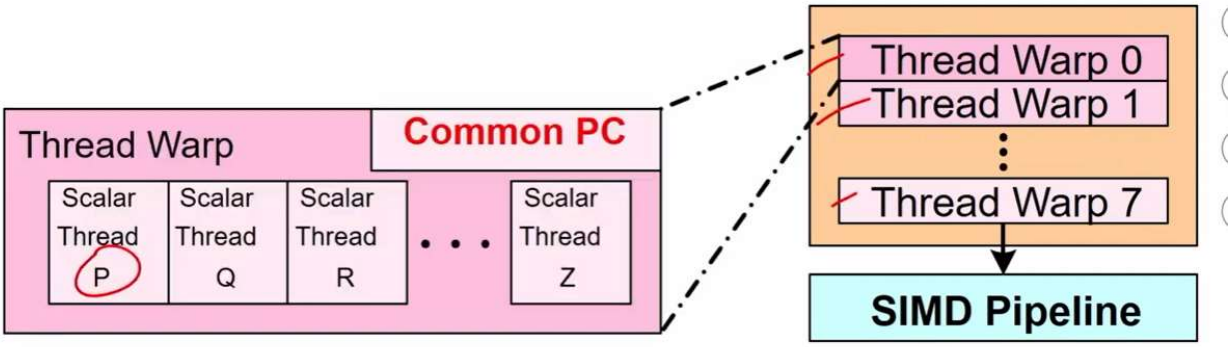
Warp has 32 threads, 32K iterations, and 1 iteration/thread → 1K warps

Warps can be interleaved on the same pipeline → Fine grained multithreading



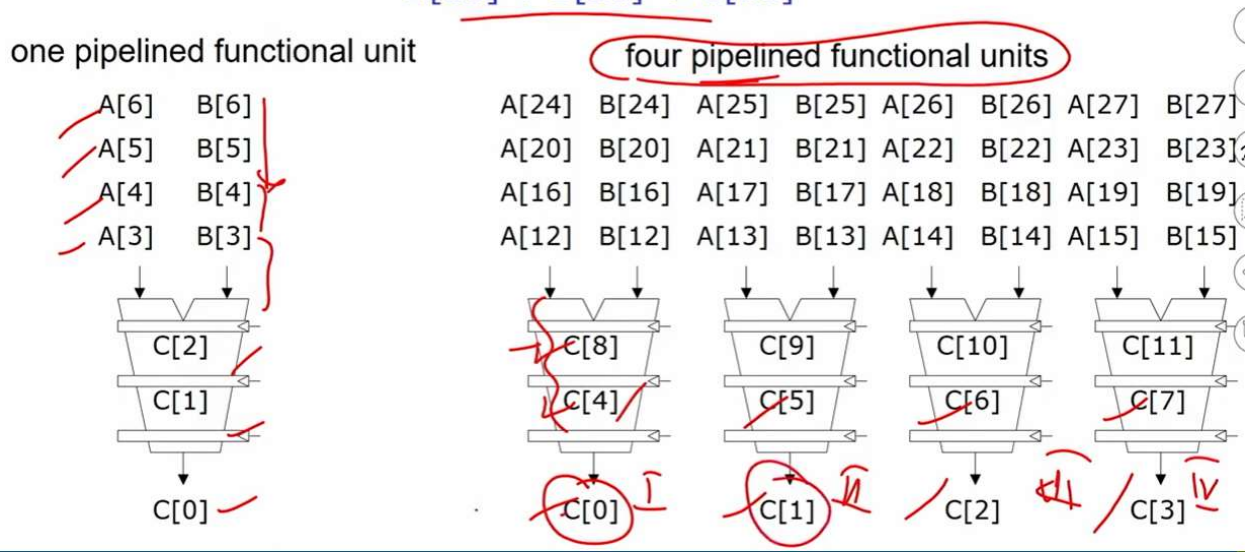
Warp-Level FGMT

- Warp: A set of threads that execute the same instruction on different data elements → SIMT
- All threads run the same code



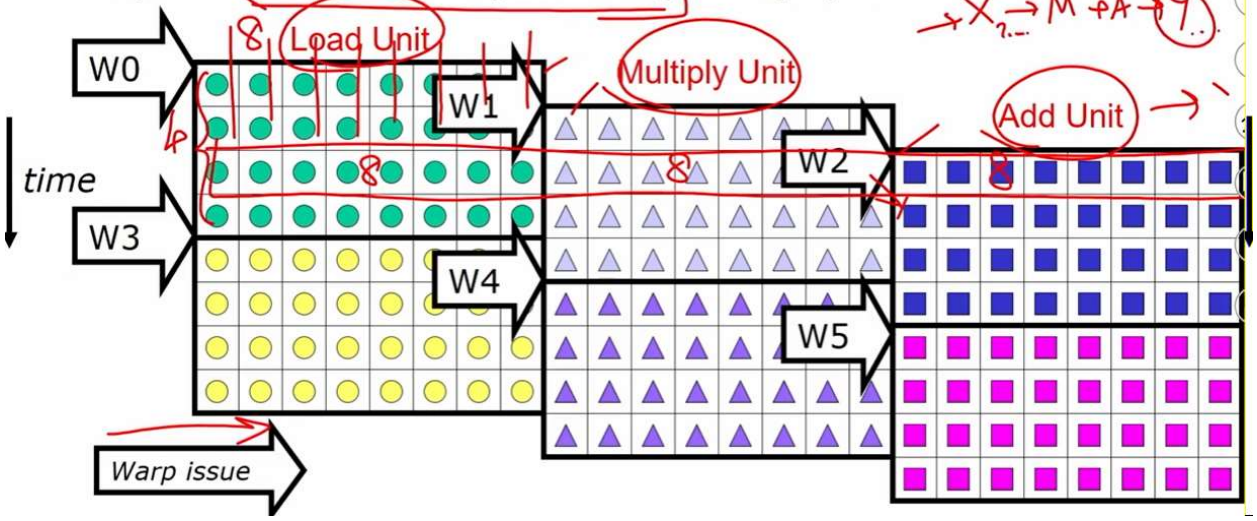
Warp-Level FGMT

32-thread warp
A[tid] + B[tid] → C[tid]



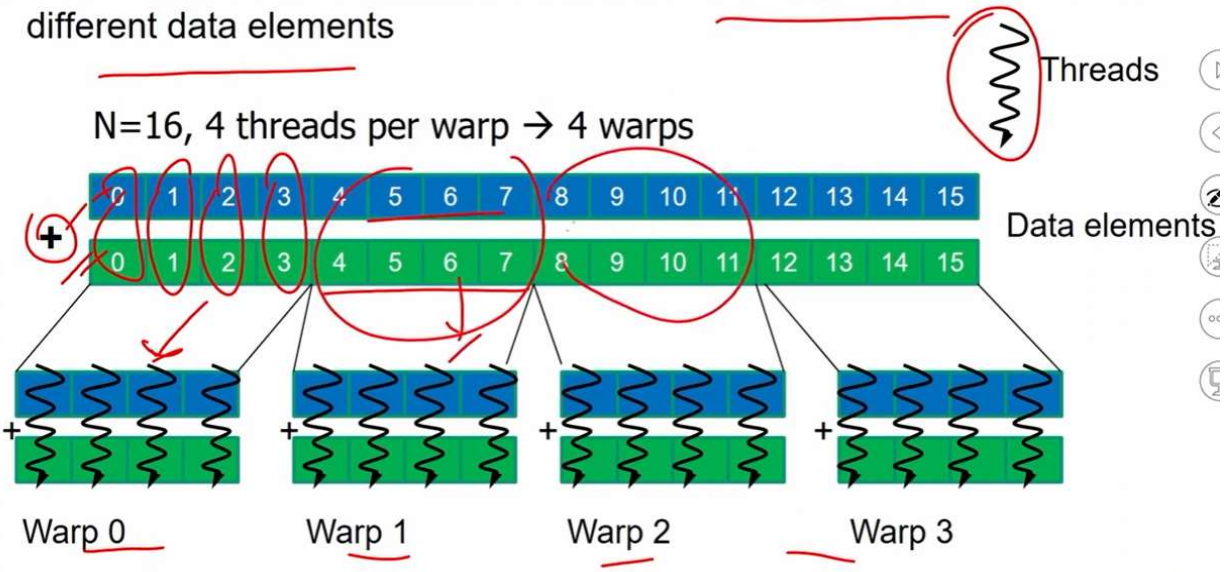
Warp Instruction Level Parallelism

- Overlap execution of multiple instructions [32 threads per warp, 8 lanes]
- Completes 24 operations/cycle issuing 1 warp/cycle



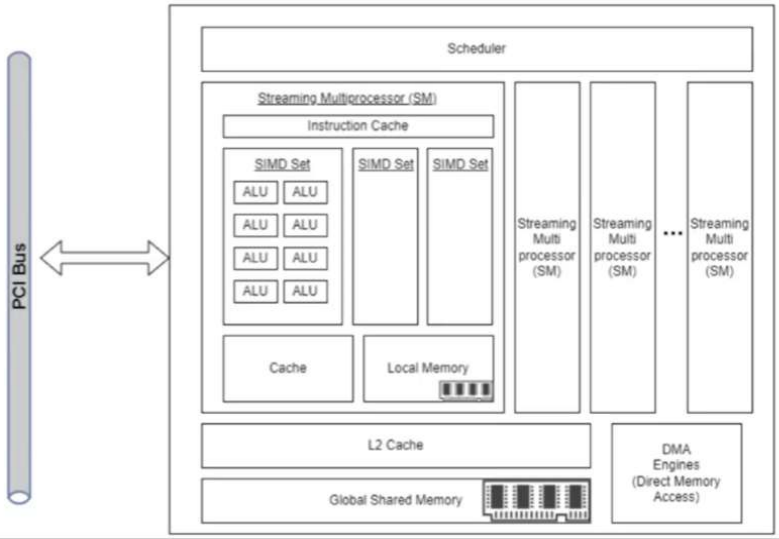
SIMT Memory Access

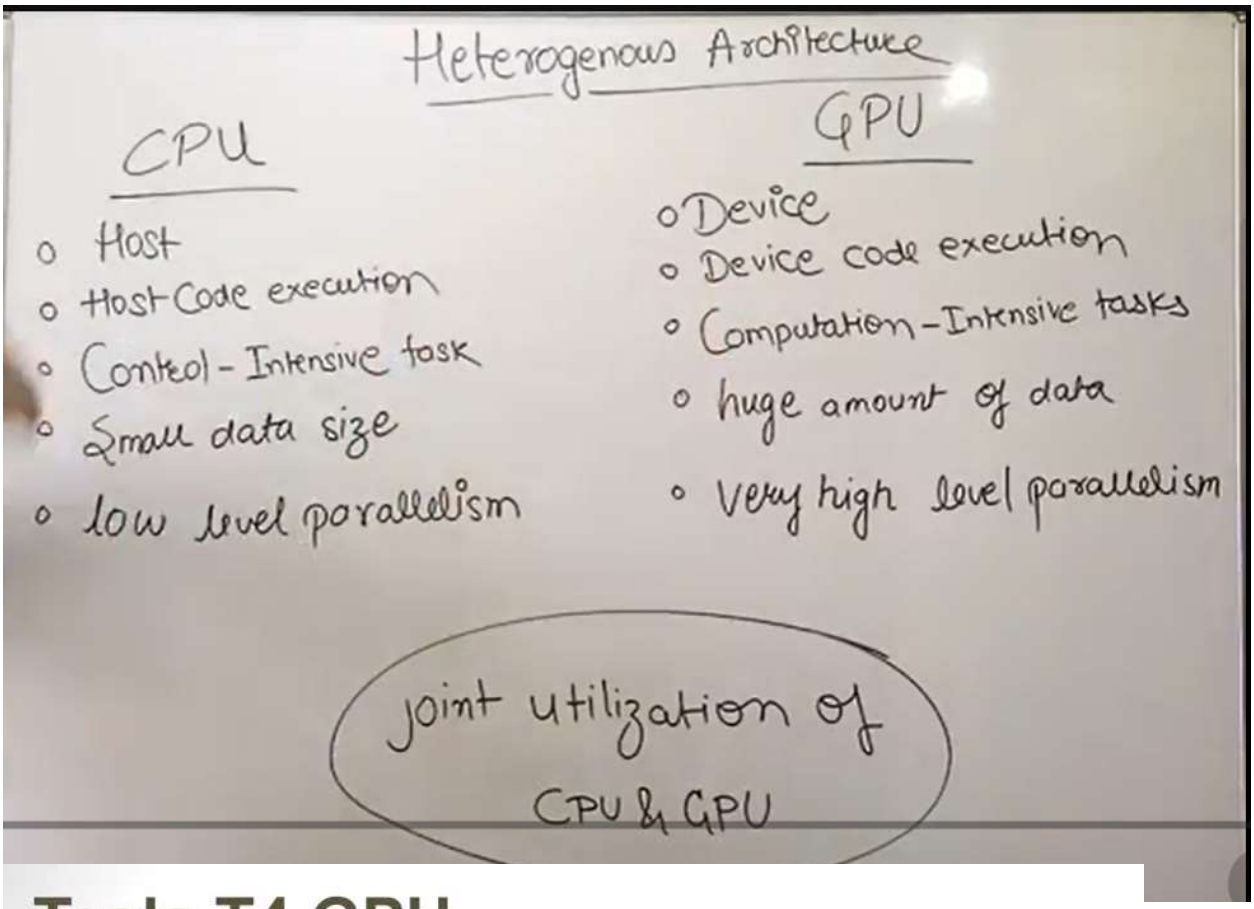
- Same instruction in different threads uses thread id to index and access different data elements



GPU Architecture

- 100s or 1000s of cores
- High Throughput
- Parallel Processing
- Data Parallelism





Tesla T4 GPU

- **SM's => 40 (Streaming Multiprocessors)**

- **Each SM have**
  - 64 CUDA Cores (64x40) = 2560
  - 8 Tensor Cores
  - 4 Warp Schedulers
  - Others (Registers, Shared Memory, L1 Cache)

Note: L2 Cache, Global Memory belongs to whole GPU not per SM

- **Others**
  - Max Threads per Block: 1024 (**32 - 1024**)
  - Warp Size: 32
  - Tensor
  - Max Threads per SM (Resident): 2048/32 = 64

NVIDIA CUDA and GPU Programming

What is CUDA?

CUDA (Compute Unified Device Architecture) is a parallel computing platform developed by NVIDIA in 2007. It allows GPUs to perform general-purpose computations beyond graphics rendering, enabling the parallel processing of large datasets. CUDA has been crucial for advancements in artificial intelligence, deep learning, and scientific computing.

CPU vs. GPU in CUDA

CPUs are designed for versatility, with relatively few cores optimized for serial processing and low-latency tasks. GPUs, on the other hand, contain hundreds to thousands of cores, optimized for high-throughput parallel operations, such as matrix multiplication and vector transformations. In CUDA, the CPU is referred to as the **host**, and the GPU as the **device**. The host executes control-intensive, sequential code, while the device handles computation-intensive tasks involving large datasets in parallel. Together, they form a **heterogeneous architecture**, leveraging the strengths of both processors.

How CUDA Works

Developers write **CUDA kernels**, which are functions executed on the GPU. Data is copied from main memory (RAM) to GPU memory. The CPU instructs the GPU to execute the kernel in parallel, organizing threads into a multi-dimensional grid. Once execution completes, results are copied back to the main memory.

Key concepts in CUDA programming include:

- **cudaDeviceSynchronize()**: Pauses CPU execution until all previously launched GPU kernels complete.
- **Modular Programming with Multiple Kernels**: Different kernels can be launched to perform distinct tasks efficiently.
- **GPU Streams and Parallel Execution**: Multiple streams allow kernels to execute concurrently on the GPU, potentially leading to non-deterministic execution order.
- **Race Conditions**: Occur when multiple kernels access shared memory concurrently in different streams. Serial execution in Stream Zero avoids race conditions.

GPU Architecture

GPU architecture is optimized for data parallelism, unlike CPUs, which are better for task parallelism. Key components include:

- **PCIe Connection**: Connects GPU to the host system.
- **Scheduler**: Assigns tasks to GPU cores.
- **Direct Memory Access (DMA) Engine**: Facilitates direct transfer of data between host and GPU memory.
- **L2 Cache**: Stores data for fast access by cores.
- **Streaming Multiprocessors (SMs)**: The core building blocks of GPUs. SMs are called **CUDA Cores** in NVIDIA GPUs, **Xe Cores** in Intel GPUs, and **Compute Units** in AMD GPUs.

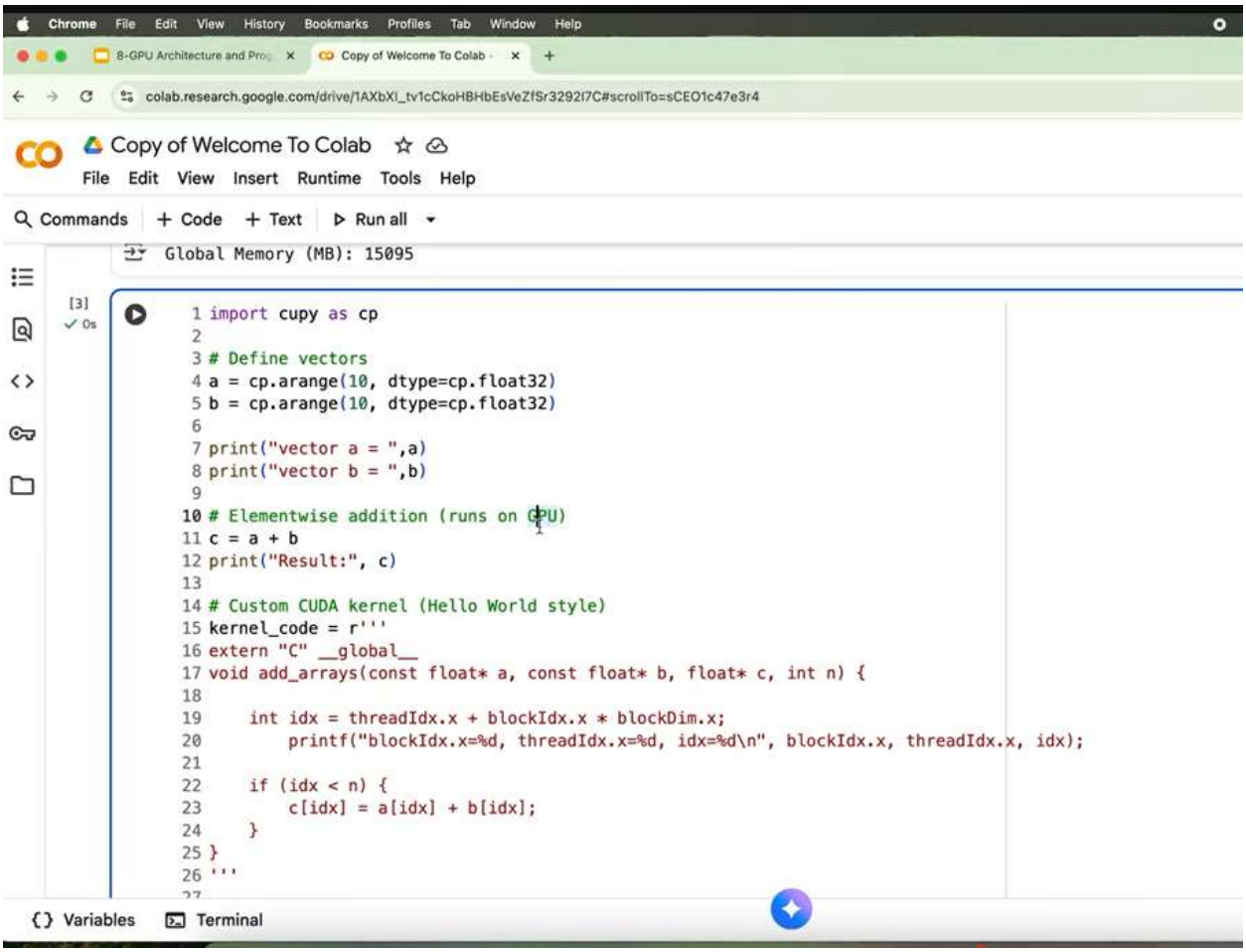
- **Execution Within SMs**: Each SM contains multiple SIMD units and ALUs, enabling multiple threads to run in parallel.
- **Job Scheduling**: The scheduler manages task allocation to SMs, each of which has local memory and cache to optimize performance.

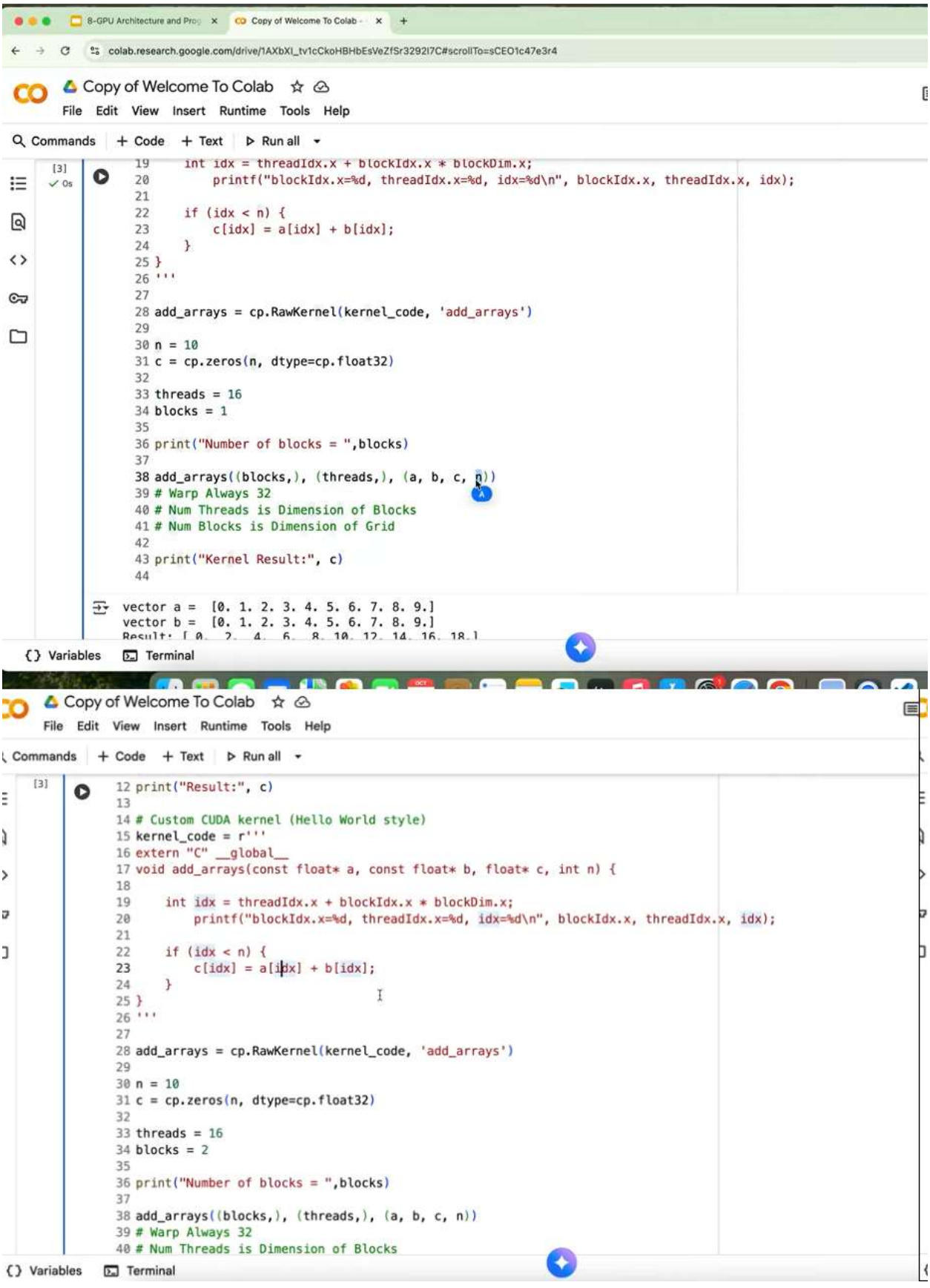
GPU Programming Model

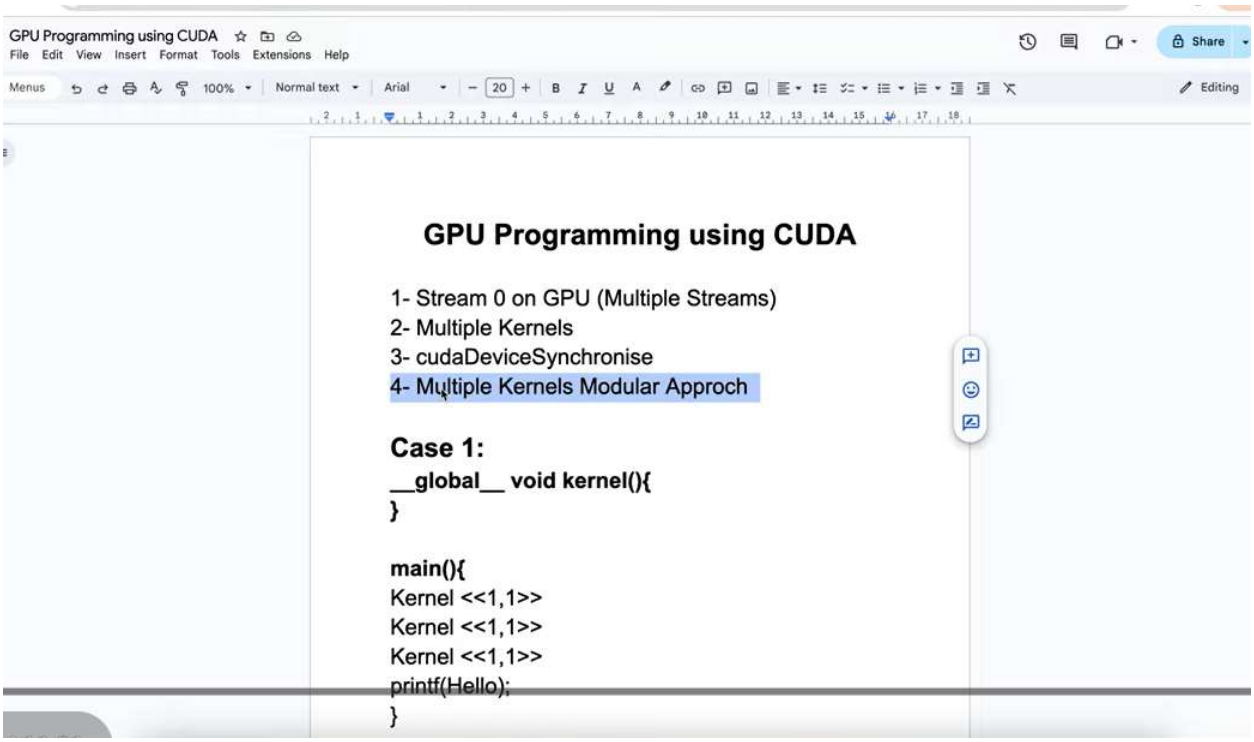
- **SIMT (Single Instruction, Multiple Threads)**: Combines SIMD with multithreading. Each thread executes the same instruction on a different piece of data.
- **Warps and Fine-Grained Multithreading**: A warp is a group of threads (commonly 32) executing the same instruction simultaneously. GPUs interleave multiple warps on the same pipeline to maximize utilization of functional units, allowing efficient processing of data elements like pixels in graphics transformations.

Practical Example
CUDA can be implemented in C++ or Python (e.g., using CuPy). Kernels are launched with configuration syntax specifying the number of blocks and threads. Developers calculate each thread's global index to handle portions of data in parallel. Synchronization ensures correct execution order and prevents race conditions.

Applications
CUDA-enabled GPUs are widely used in AI, deep learning, graphics-intensive applications, games, and scientific computing, where large datasets and parallel computations are common. The heterogeneous CPU-GPU architecture ensures that both sequential and parallel workloads are efficiently handled.


Chrome File Edit View History Bookmarks Profiles Tab Window Help
8-GPU Architecture and Prog
Copy of Welcome To Colab -
colab.research.google.com/drive/1AXbXI_tv1cCkoHBHbEsVeZfSr3292l7C#scrollTo=sCEO1c47e3r4
Copy of Welcome To Colab
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Global Memory (MB): 15095
[3]
0s
1 import cupy as cp
2
3 # Define vectors
4 a = cp.arange(10, dtype=cp.float32)
5 b = cp.arange(10, dtype=cp.float32)
6
7 print("vector a = ",a)
8 print("vector b = ",b)
9
10 # Elementwise addition (runs on GPU)
11 c = a + b
12 print("Result:", c)
13
14 # Custom CUDA kernel (Hello World style)
15 kernel_code = r'''
16 extern "C" __global__
17 void add_arrays(const float* a, const float* b, float* c, int n) {
18
19 int idx = threadIdx.x + blockIdx.x * blockDim.x;
20 printf("blockIdx.x=%d, threadIdx.x=%d, idx=%d\n", blockIdx.x, threadIdx.x, idx);
21
22 if (idx < n) {
23 c[idx] = a[idx] + b[idx];
24 }
25 }
26 '''
Variables
Terminal


8-GPU Architecture and Prog
Copy of Welcome To Colab
colab.research.google.com/drive/1AXbXI_tv1cCkoHBHbEsVeZfSr3292l7C#scrollTo=sCEO1c47e3r4
Copy of Welcome To Colab
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
[3]
0s
int idx = threadIdx.x + blockIdx.x * blockDim.x;
printf("blockIdx.x=%d, threadIdx.x=%d, idx=%d\n", blockIdx.x, threadIdx.x, idx);
if (idx < n) {
c[idx] = a[idx] + b[idx];
}
}
'''
add_arrays = cp.RawKernel(kernel_code, 'add_arrays')
n = 10
c = cp.zeros(n, dtype=cp.float32)
threads = 16
blocks = 1
print("Number of blocks = ",blocks)
add_arrays((blocks,), (threads,), (a, b, c, n))
# Warp Always 32
# Num Threads is Dimension of Blocks
# Num Blocks is Dimension of Grid
print("Kernel Result:", c)
vector a = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
vector b = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
Variables
Terminal
Copy of Welcome To Colab
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
[3]
print("Result:", c)
# Custom CUDA kernel (Hello World style)
kernel_code = r'''
extern "C" __global__
void add_arrays(const float* a, const float* b, float* c, int n) {
int idx = threadIdx.x + blockIdx.x * blockDim.x;
printf("blockIdx.x=%d, threadIdx.x=%d, idx=%d\n", blockIdx.x, threadIdx.x, idx);
if (idx < n) {
c[idx] = a[idx] + b[idx];
}
}
'''
add_arrays = cp.RawKernel(kernel_code, 'add_arrays')
n = 10
c = cp.zeros(n, dtype=cp.float32)
threads = 16
blocks = 2
print("Number of blocks = ",blocks)
add_arrays((blocks,), (threads,), (a, b, c, n))
# Warp Always 32
# Num Threads is Dimension of Blocks
Variables
Terminal


GPU Programming using CUDA
File Edit View Insert Format Tools Extensions Help
Share
Menus
100%
Normal text
Arial
20
Editing
GPU Programming using CUDA
1- Stream 0 on GPU (Multiple Streams)
2- Multiple Kernels
3- cudaDeviceSynchronise
4- Multiple Kernels Modular Approch
Case 1:
__global__ void kernel(){
}
main(){
Kernel <<1,1>>
Kernel <<1,1>>
Kernel <<1,1>>
printf(Hello);
}

Case 2:

```
__global__ void kernel(){

}
```

```
main(){
Kernel <<1,1>>
Kernel <<1,1>>
Kernel <<1,1>>
cudaDeviceSynchronize()
printf(Hello);
}
```

Case 3:

```
__global__ void kernel1(){
 ////
}
```

Case 3:

```
__global__ void kernel1(){
////
}

__global__ void kernel2(){
////
}

main(){
kernel1 <<1,1>>
kernel2 <<1,1>>
}
```

Case 4:

For Parallel Launching a Kernel

```
kernel<<<1,1,0,stream1>>>();
```

Case 4:

For Parallel Launching a Kernel

```
kernel<<<1,1,0,stream1>>>();
kernel<<<1,1,0,stream2>>>();
kernel<<<1,1,0,stream3>>>();
```

RACE conditions

Case 1: (No Race Condition)

```
Kernel <<1,1>>
Kernel <<1,1>>
Kernel <<1,1>>
```

Case 2: (No Race Condition)

```
Kernel <<1,1>>
Kernel <<1,1>>
Kernel <<1,1>>
```

RACE conditions

Case 1: (No Race Condition)

```
Kernel <<1,1>>
Kernel <<1,1>>
Kernel <<1,1>>
```

Case 2: (No Race Condition)

```
Kernel <<1,1>>
Kernel <<1,1>>
Kernel <<1,1>>
cudaDeviceSynchronize()
```

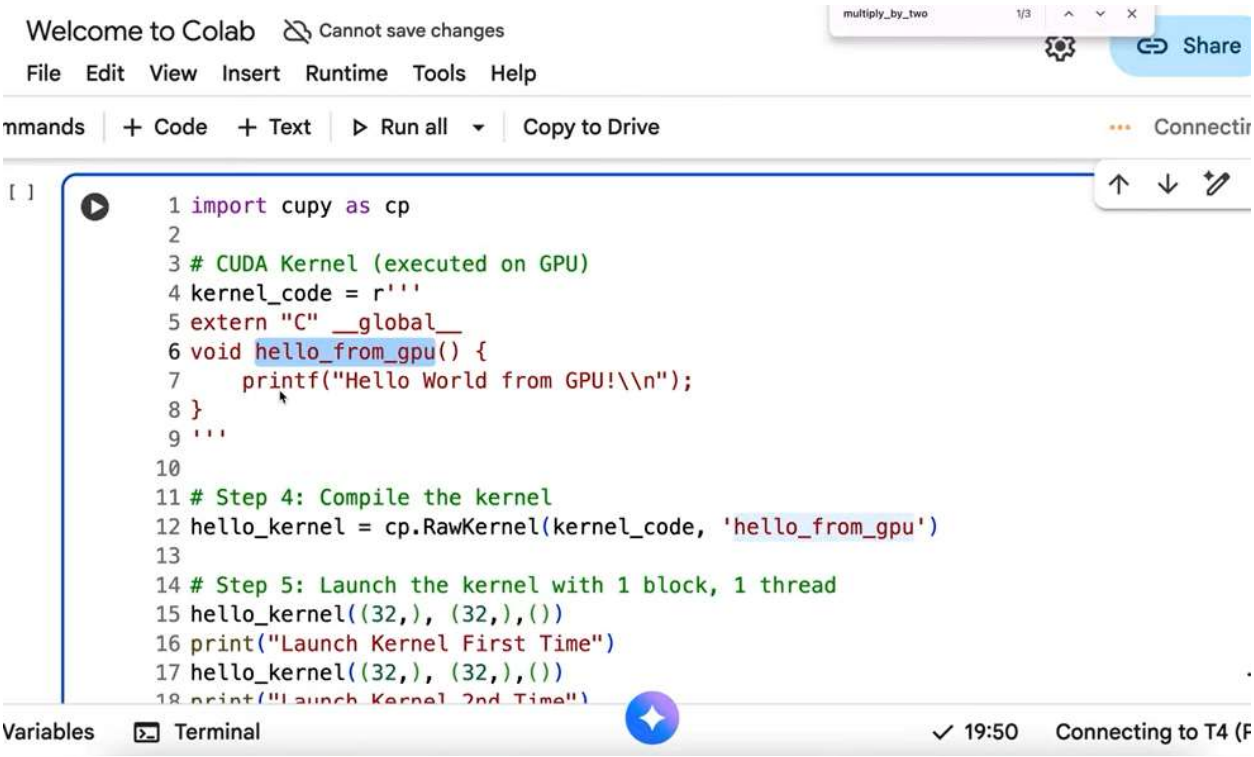
Case 3: (No Race Condition)

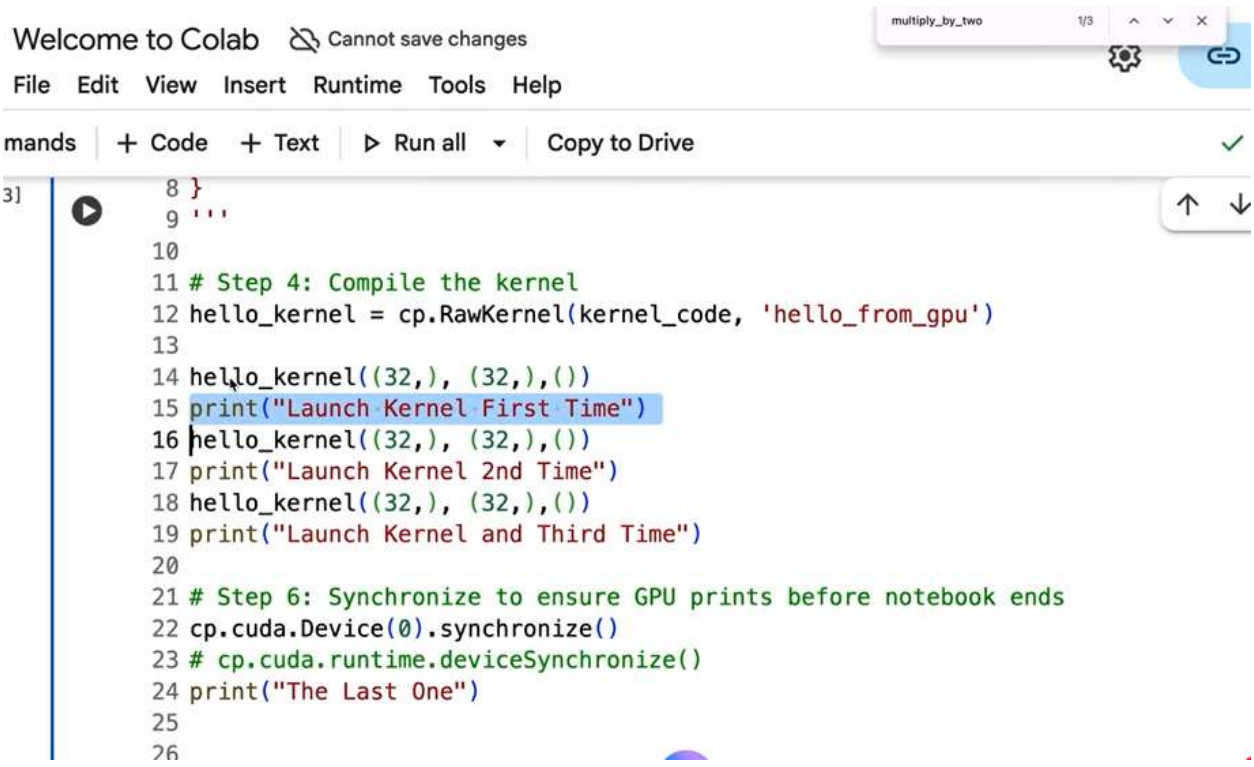
```
Kernel1 <<1,1>>
Kernel2 <<1,1>>
Kernel3 <<1,1>>
```

Case 4: (Could have a race condition)

For Parallel Launching a Kernel

```
kernel<<<1,1,0,stream1>>>();
kernel<<<1,1,0,stream2>>>();
kernel<<<1,1,0,stream3>>>();
```


Welcome to Colab
Cannot save changes
File Edit View Insert Runtime Tools Help
Share
+ Code + Text Run all Copy to Drive
Connecti
[ ]
1 import cupy as cp
2
3 # CUDA Kernel (executed on GPU)
4 kernel_code = r'''
5 extern "C" __global__
6 void hello_from_gpu() {
7 printf("Hello World from GPU!\\n");
8 }
9 '''
10
11 # Step 4: Compile the kernel
12 hello_kernel = cp.RawKernel(kernel_code, 'hello_from_gpu')
13
14 # Step 5: Launch the kernel with 1 block, 1 thread
15 hello_kernel((32,), (32,),())
16 print("Launch Kernel First Time")
17 hello_kernel((32,), (32,),())
Variables
Terminal
19:50
Connecting to T4


Welcome to Colab
Cannot save changes
File Edit View Insert Runtime Tools Help
+ Code + Text Run all Copy to Drive
3]
8 }
9 '''
10
11 # Step 4: Compile the kernel
12 hello_kernel = cp.RawKernel(kernel_code, 'hello_from_gpu')
13
14 hello_kernel((32,), (32,),())
15 print("Launch Kernel First Time")
16 hello_kernel((32,), (32,),())
17 print("Launch Kernel 2nd Time")
18 hello_kernel((32,), (32,),())
19 print("Launch Kernel and Third Time")
20
21 # Step 6: Synchronize to ensure GPU prints before notebook ends
22 cp.cuda.Device(0).synchronize()
23 # cp.cuda.runtime.deviceSynchronize()
24 print("The Last One")
25
26

```
print("Launch Kernel and Third Time")

# Step 6: Synchronize to ensure GPU prints before notebook ends
cp.cuda.Device(0).synchronize()
# cp.cuda.runtime.deviceSynchronize()
print("The Last One")
```

```
Launch Kernel First Time
Launch Kernel 2nd Time
Launch Kernel and Third Time
The Last One
```

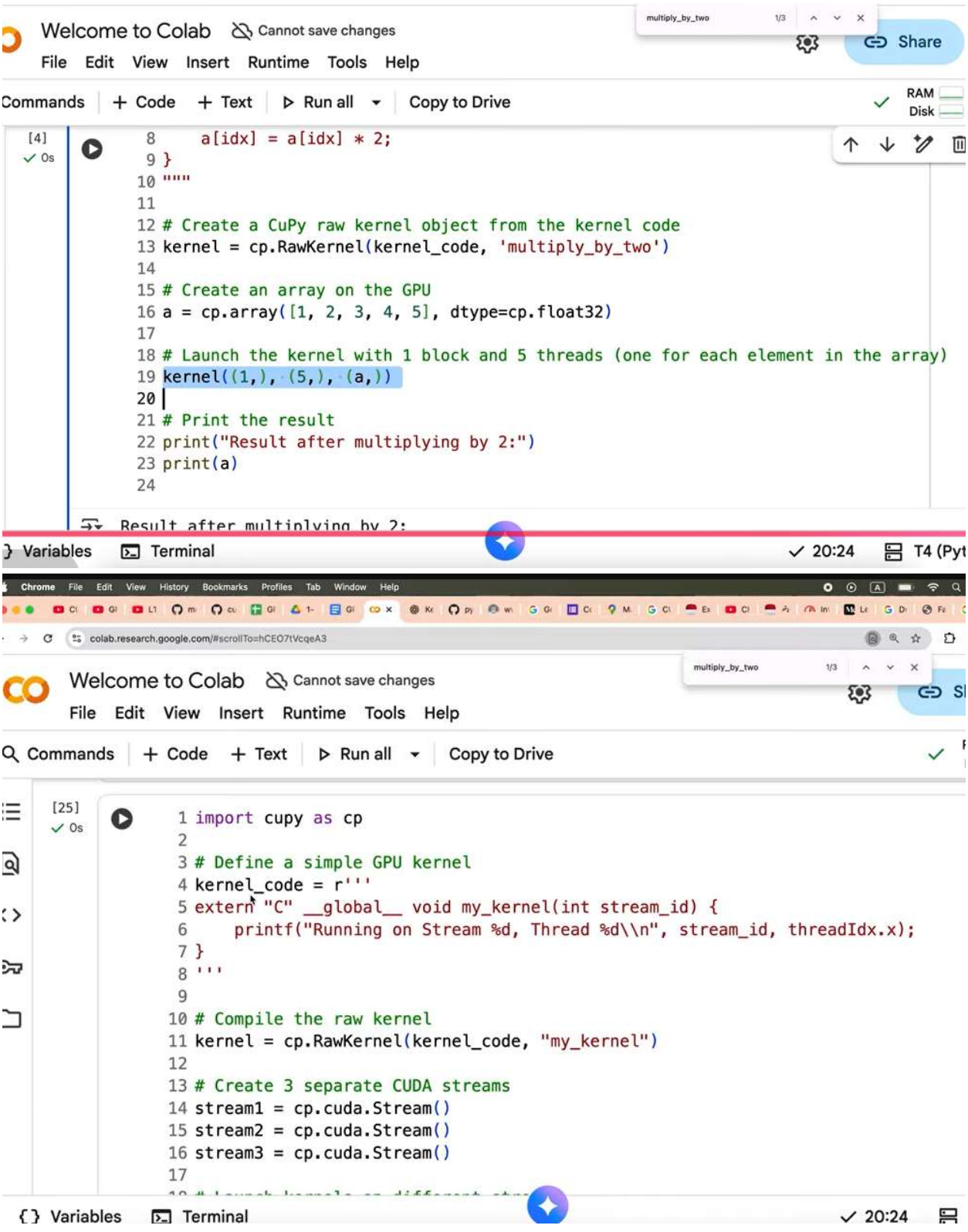
```
Start coding or generate with AI.
```

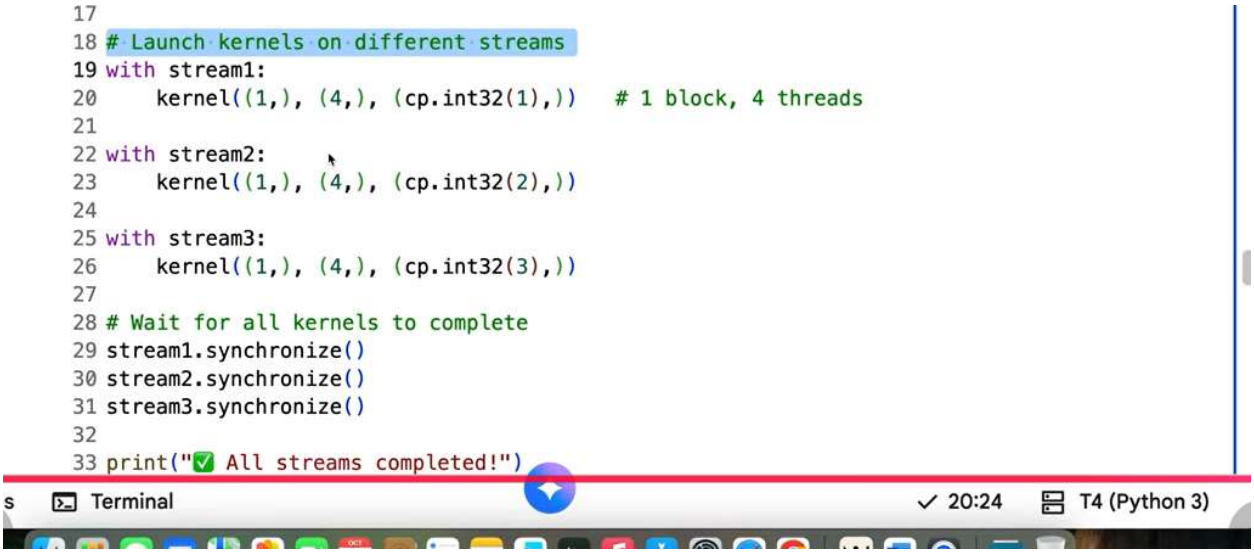
```
import cupy as cp

# Define the CUDA kernel (C-style code)
kernel_code = """
extern "C" __global__ void multiply_by_two(float *a)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    a[idx] = a[idx] * 2;
}
"""

# Create a CuPy raw kernel object from the kernel code
kernel = cp.RawKernel(kernel_code, 'multiply_by_two')

# Create an array on the GPU
a = cp.array([1, 2, 3, 4, 5], dtype=cp.float32)
```


Welcome to Colab
Cannot save changes
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all Copy to Drive
RAM
Disk
multiply_by_two 1/3
Share
[4]
0s
a[idx] = a[idx] * 2;
}
"""
# Create a CuPy raw kernel object from the kernel code
kernel = cp.RawKernel(kernel_code, 'multiply_by_two')
# Create an array on the GPU
a = cp.array([1, 2, 3, 4, 5], dtype=cp.float32)
# Launch the kernel with 1 block and 5 threads (one for each element in the array)
kernel((1,), (5,), (a,))
# Print the result
print("Result after multiplying by 2:")
print(a)
Result after multiplying by 2:
Variables
Terminal
20:24
T4 (Pyt
colab.research.google.com/#scrollTo=hCEO7tVcqeA3
Welcome to Colab
Cannot save changes
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all Copy to Drive
[25]
0s
import cupy as cp
# Define a simple GPU kernel
kernel_code = r'''
extern "C" __global__ void my_kernel(int stream_id) {
printf("Running on Stream %d, Thread %d\\n", stream_id, threadIdx.x);
}
'''
# Compile the raw kernel
kernel = cp.RawKernel(kernel_code, "my_kernel")
# Create 3 separate CUDA streams
stream1 = cp.cuda.Stream()
stream2 = cp.cuda.Stream()
stream3 = cp.cuda.Stream()
Variables
Terminal
20:24


17
18 # Launch kernels on different streams
19 with stream1:
20 kernel((1,), (4,), (cp.int32(1),)) # 1 block, 4 threads
21
22 with stream2:
23 kernel((1,), (4,), (cp.int32(2),))
24
25 with stream3:
26 kernel((1,), (4,), (cp.int32(3),))
27
28 # Wait for all kernels to complete
29 stream1.synchronize()
30 stream2.synchronize()
31 stream3.synchronize()
32
33 print("✅ All streams completed!")
s
Terminal
20:24
T4 (Python 3)