INAF
ISTITUTO NAZIONALE
DI ASTROFISICA

GPU Programming

Lecture 1
UniTS Advance HPC Course 2024/2025

Agenda

- GPU Architecture
- GPU vs CPU
- Parallel programming with GPUs
- Brief introduction to CUDA
- OpenMP for GPUs
- Examples and Exercises

But not everything today….

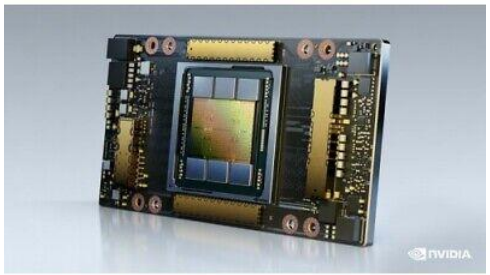
GPU Computing

"If you were plowing a field, which would you rather use?...
Two strong oxen or 1024 chickens?"

Seymour Cray (http://en.wikiquote.org/wiki/Seymour_Cray)

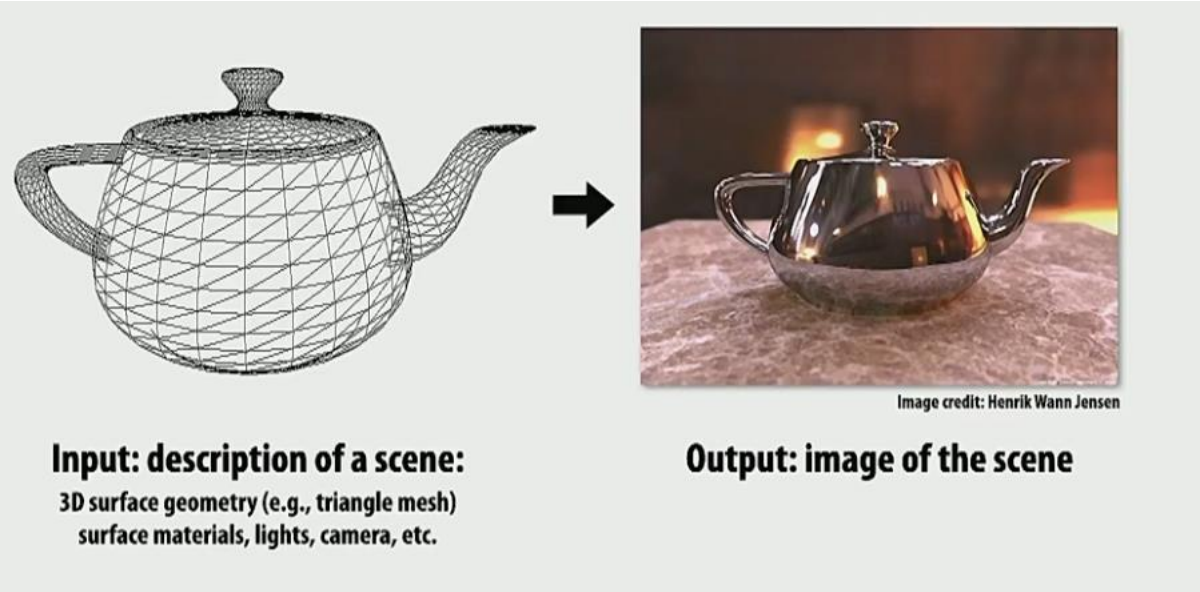
1024 *chickens*!!!

What is a GPU?



The GPU Chronicles: Back to the origins.

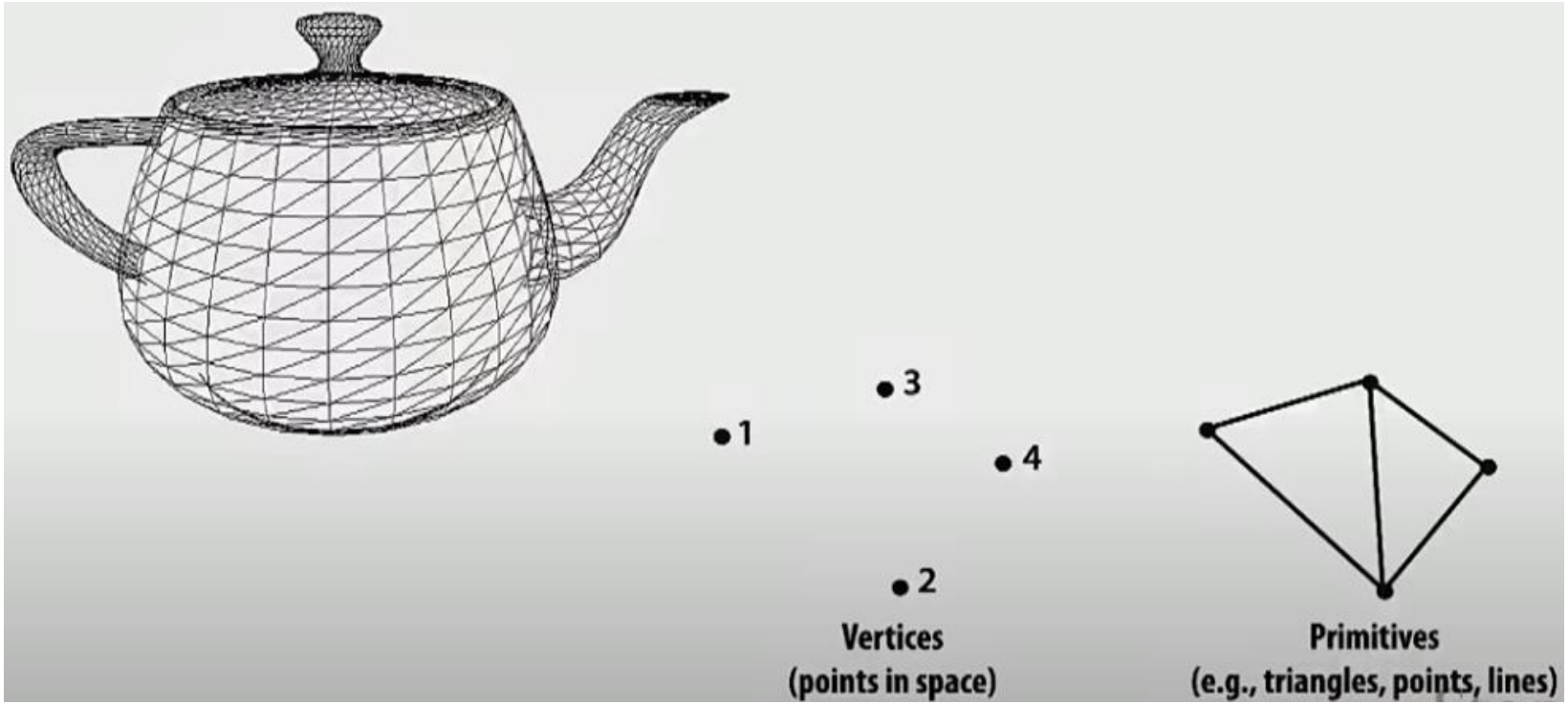
3D RENDERING…



What were GPUs were originally designed to do?

Real-time graphics primitives

Represent surfaces ad 3D triangle meshes

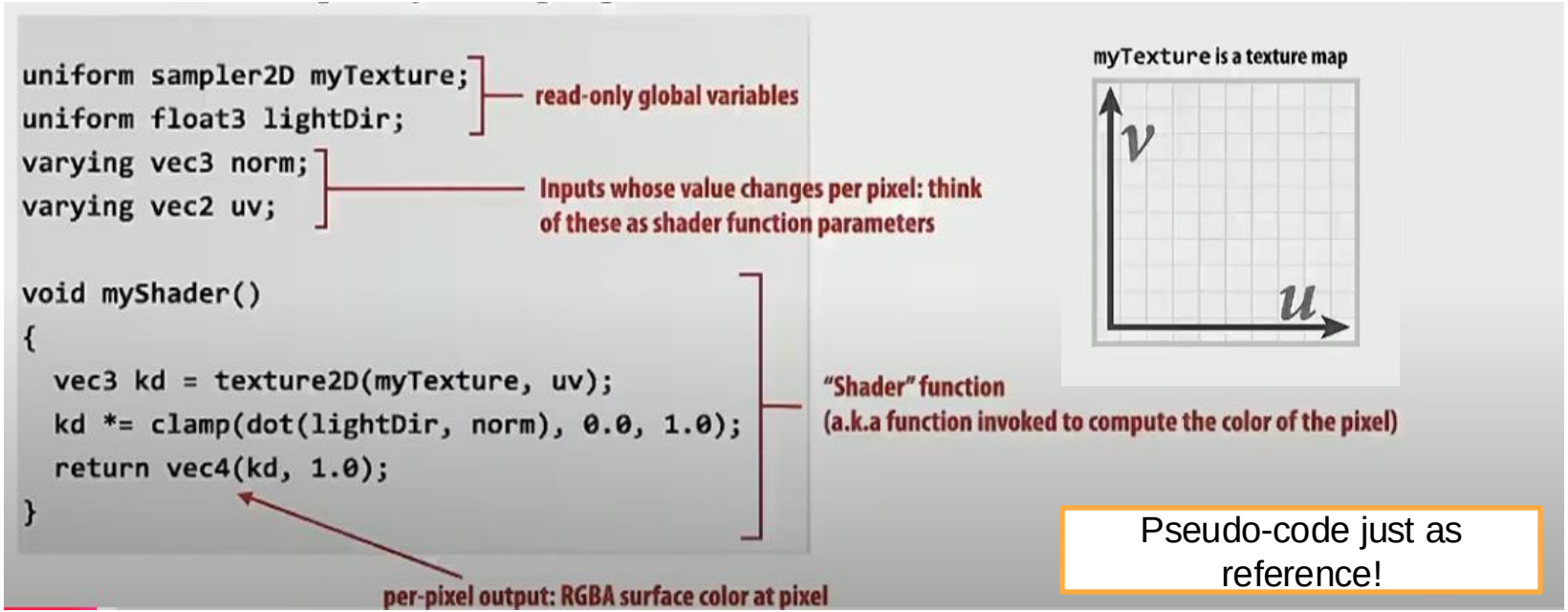


Real-time graphics workload…in one slide

- **Given a tringle, determine where it lies on screen given the position of a virtual camera**
- **For all output image pixels covered by the triangle, compute the color of the surface at that pixel.**

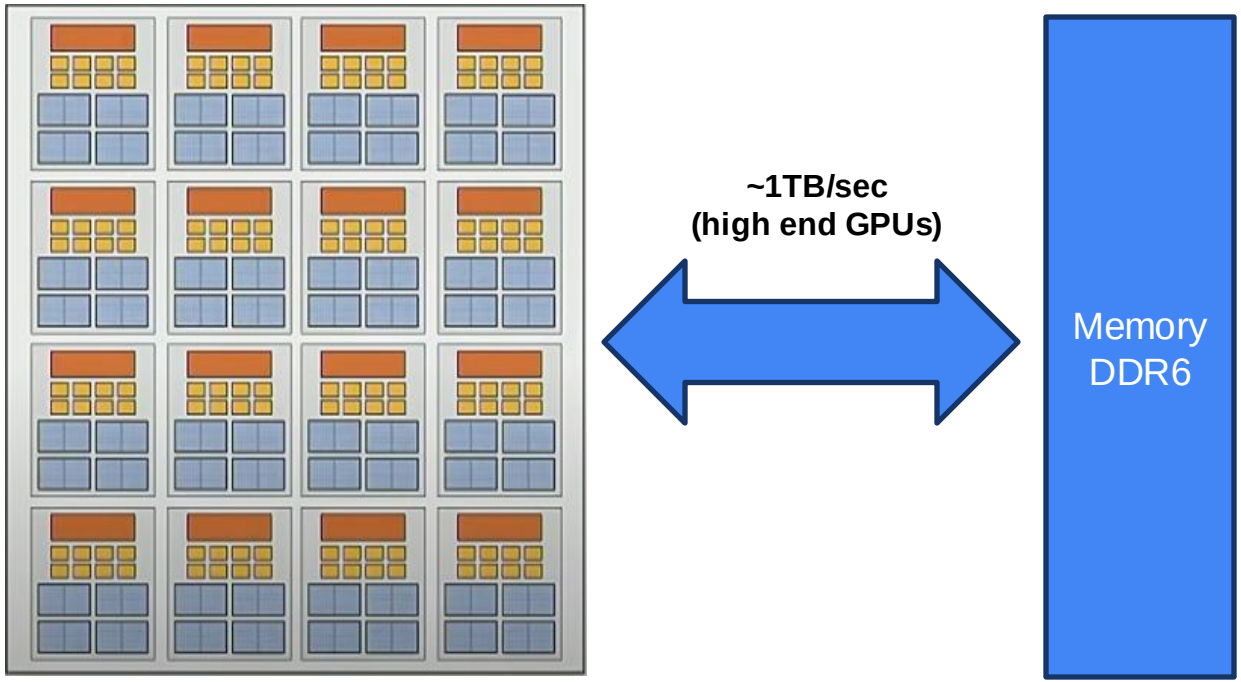
Example “shader program”…

- Run once per fragment (per pixel covered by a tringle)
- OpenGL shading language (GLSL) shader program



Why do GPUs have many high throughput cores?

Many SIMD, multi-threaded cores provide efficient execution of shader programs



What were GPUs were originally designed to do?

…in summary:

- Graphics pipeline: huge amount of arithmetic on independent data:
  - Transforming positions
  - Generating pixel colors
  - Applying material properties and light situation to every pixel

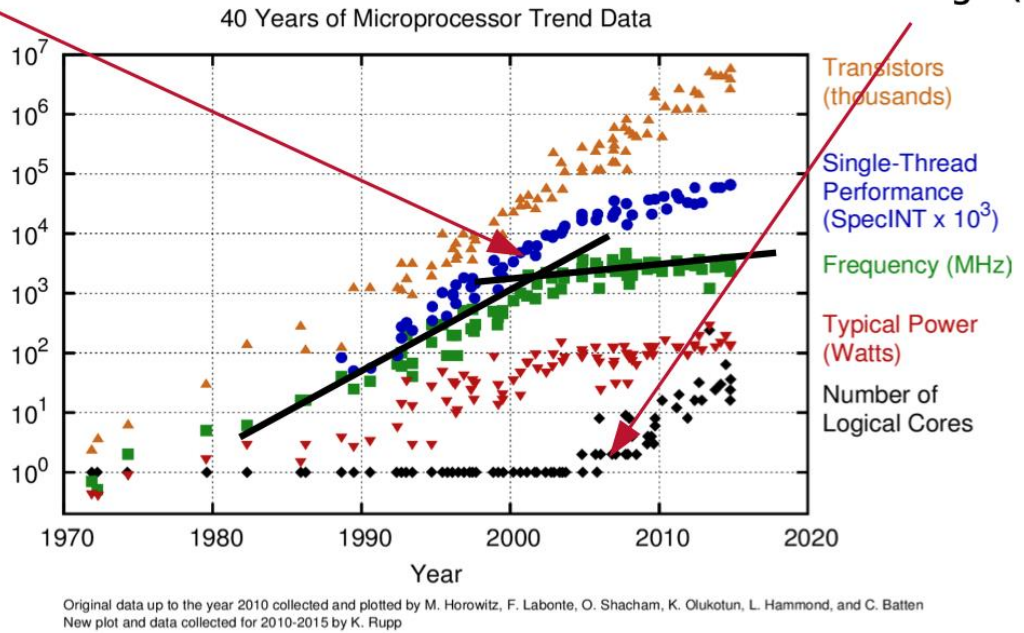
Hardware needs

- Access memory simultaneously and contiguously
- Bandwidth more important than latency
- Floating point and fixed-function logic

Moore's law Today

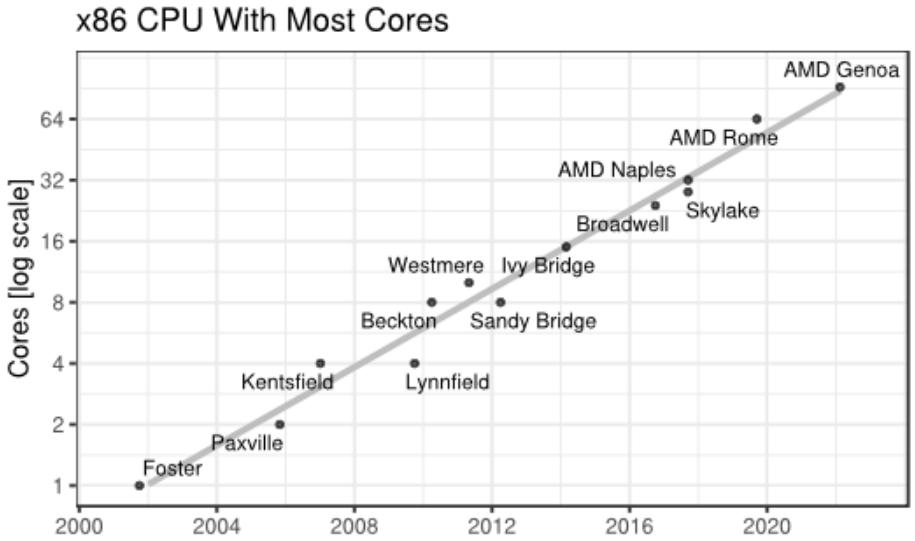
Clock speed stopped increasing due to heat limit

Multiple core processors emerge (Intel i7: 4 cores)



CPU stagnation

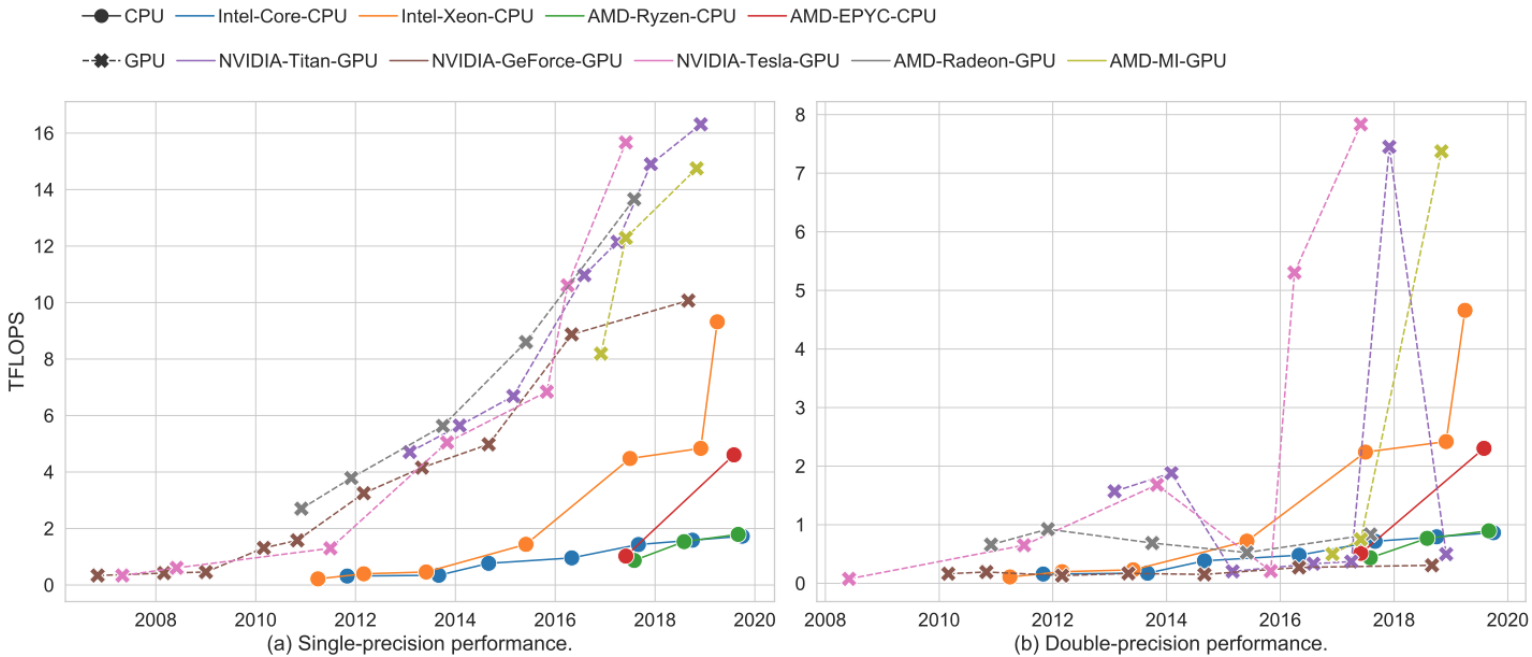
- Core counts double roughly every three years.
- When considering cost per core, this trend appears to have stalled.



Evolution of peak FLOPs

Gaming industry evolves steady → continuous high demand for consumer GPUs

AI growth in many areas → continuous demand for professional GPUs



What were GPUs were originally designed to do?

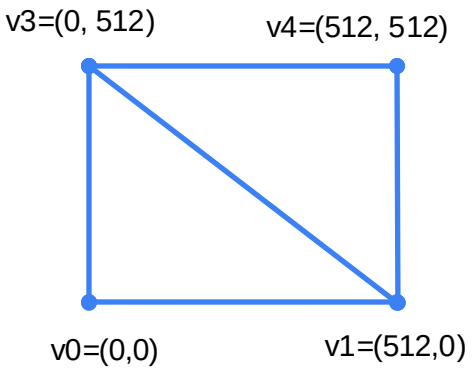
Early 2000. Hacking the system to run scientific computation….

- → **Say you want to run a function on all elements of a 512x512 array**
- → **Set output image size to be array size (512 × 512)**
- → **Render two triangles that exactly cover screen**

(one shader computation per pixel = one shader computation output image element)

We now can use the GPU like a data-parallel programming system.

Fragment shader function is mapped over 512 x 512 element collection.



…and then the NVIDIA Tesla (2007)

First alternative, non-graphics-specific ("compute mode") interface to GPU hardware

Let's say a user wants to run a non-graphics program on the GPU's programmable cores…

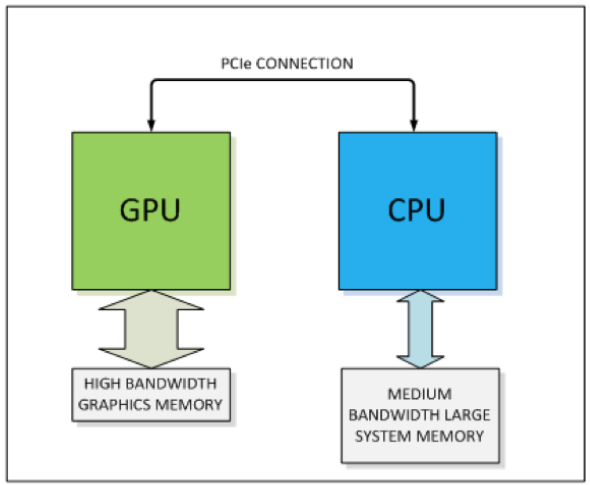
- ✓ **Application can allocate buffers in GPU memory and copy data to/from buffers**
- ✓ **Application (via graphics driver) provides GPU a single kernel program binary**
- ✓ **Application tells GPU to run the kernel in an SPMD fashion**
  ("run N instances of this kernel")

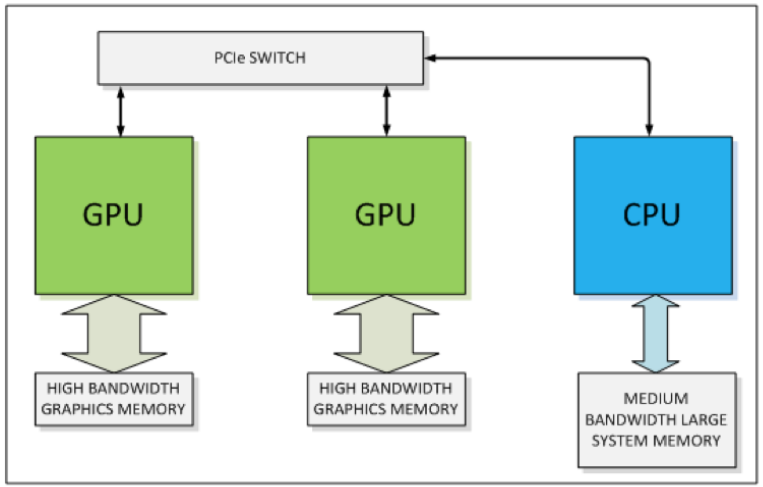
```
launch (myKernel, N)
```

This is a far simpler operation than the graphics operation `drawPrimitives()`

NVIDIA Tesla S870

GPU Architecture: CPU and GPUs





PCIe generation	1 lane	16 lanes	Year
3.0	985 MB/s	15.75 GB/s	2010
4.0	1.97 GB/s	31.5 GB/s	2017

Latest generation Nvidia & AMD GPUs have PCIe 4.0

One-Minute Foundation: Latency and Throughput

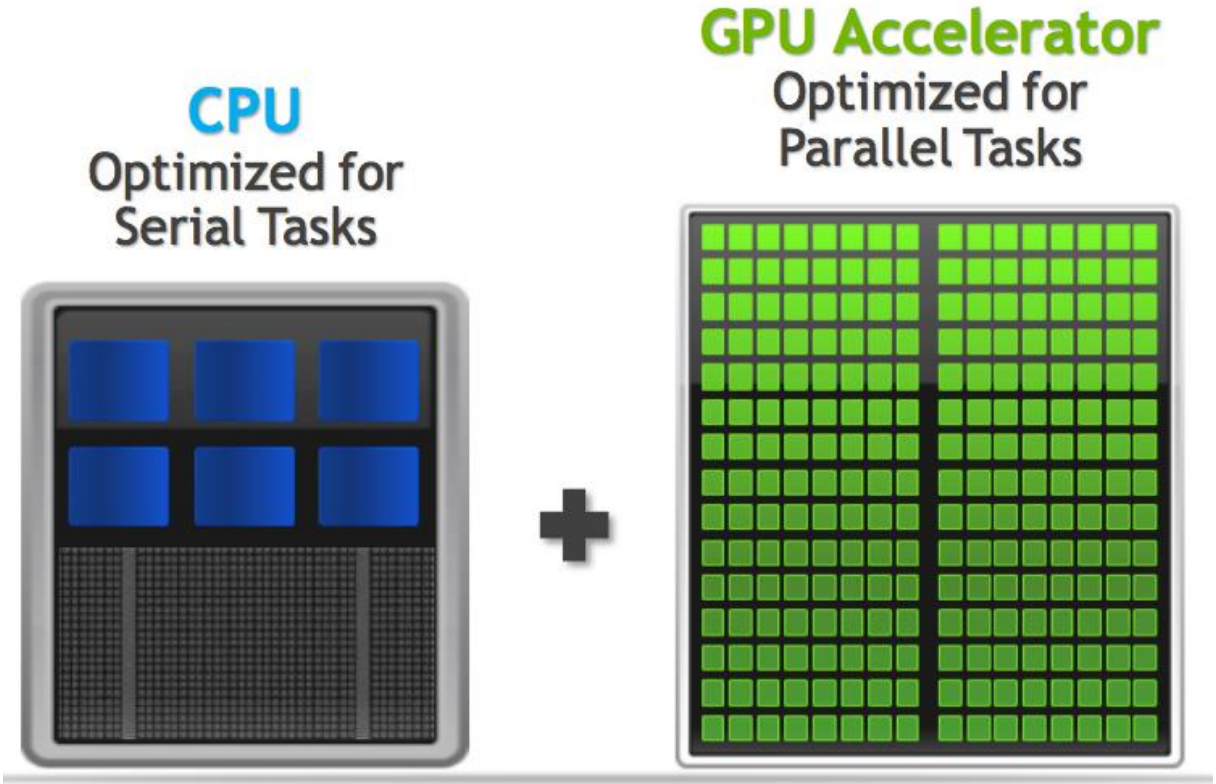
Latency refers to the delay in processing data in a computer system after the original instruction.

Throughput refers to the number of items and instructions that a compute unit can process simultaneously.

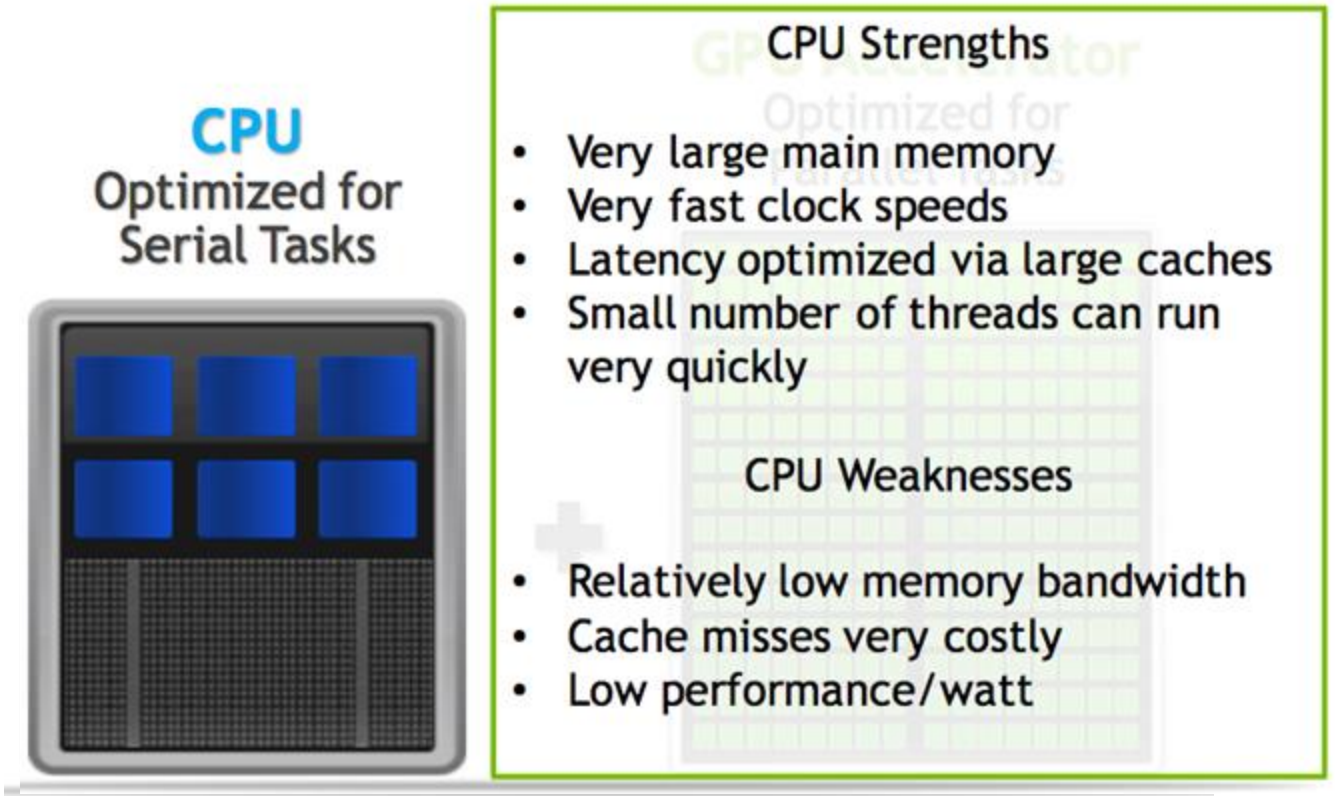
Throughput

Latency

CPU vs GPU


CPU
Optimized for
Serial Tasks
GPU Accelerator
Optimized for
Parallel Tasks

CPU vs GPU


CPU
Optimized for
Serial Tasks
CPU Strengths
• Very large main memory
• Very fast clock speeds
• Latency optimized via large caches
• Small number of threads can run very quickly
CPU Weaknesses
• Relatively low memory bandwidth
• Cache misses very costly
• Low performance/watt

CPU vs GPU

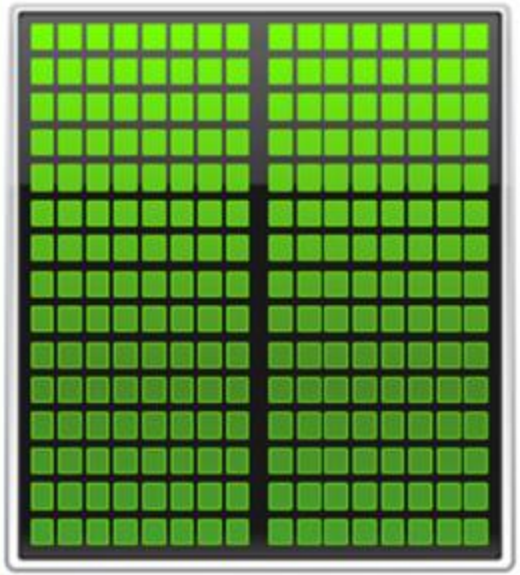
GPU Strengths

- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance

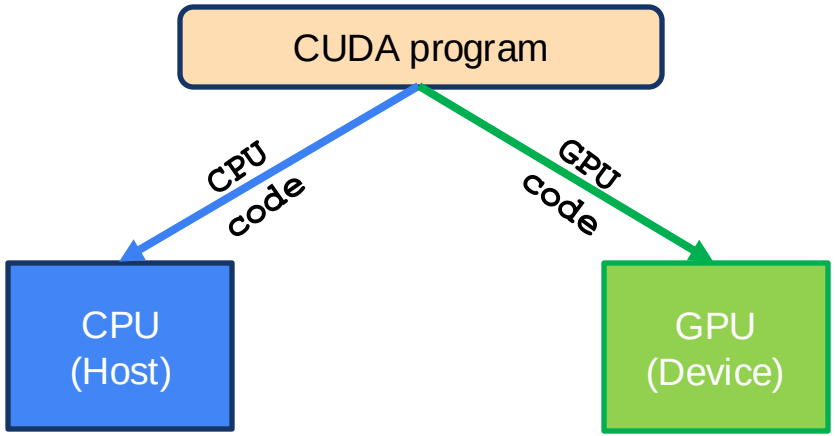
GPU Accelerator
Optimized for
Parallel Tasks



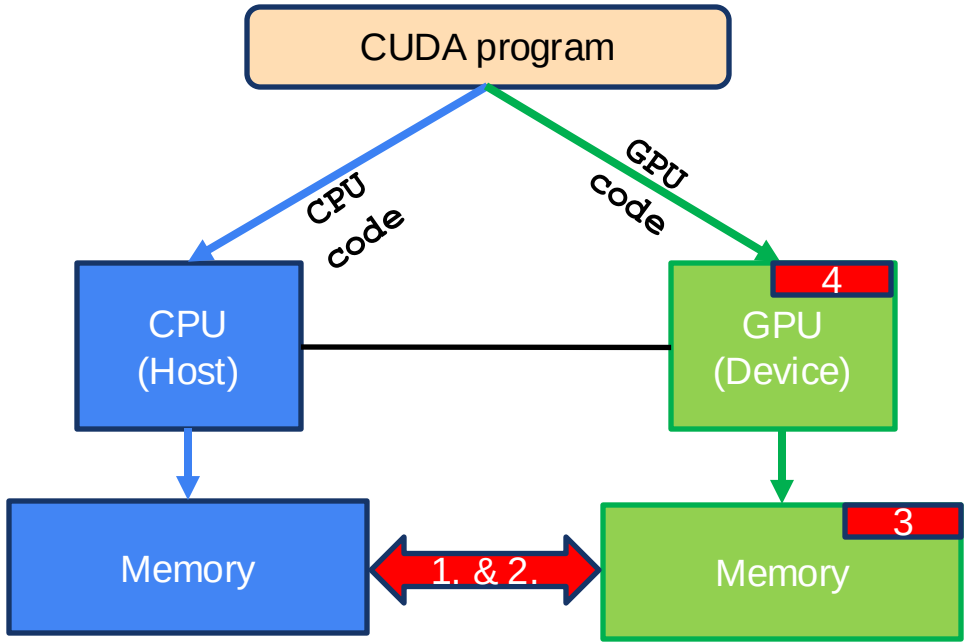
How to program heterogenous HW…

In 2007 NVIDIA introduced CUDA programming language

- **“C-like” language to express programs that run on GPUs using the compute mode hw interface**
- **Relatively low-level: CUDA’s abstractions closely match the capabilities/performance characteristics of modern NVIDIA GPUs**



How to program heterogenous HW…



1. Copy data CPU → GPU
2. Copy data GPU → CPU

   `cudaMemcpy()`

3. Allocate GPU memory

   `cudaMalloc()`

4. Launch code (kernel) on GPU

GPU: how does it work?

Sum of two arrays

```
for (int i=0; i<6; i++) a(i)=a(i)+b(i)
```

The CPU would step through the loop 6 times, adding array elements one by one

Also CPU are “parallel” (Cores, AVX, threads..)

Array a		Array b
32	+	7
4	+	1
48	+	11
68	+	56
5	+	2
63	+	3

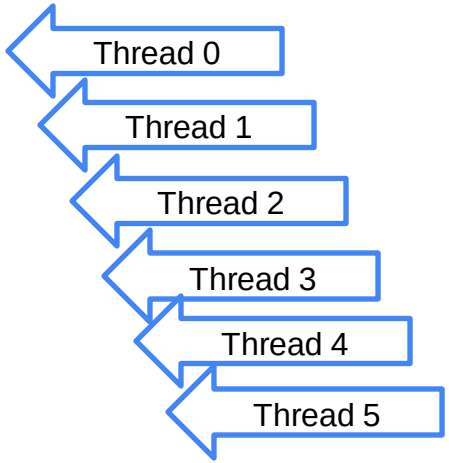
GPU: how does it work?

Sum of two arrays

```
for (int i=0; i<6; i++) a(i)=a(i)+b(i)
```

The GPU instead will start 6 threads at the same time, summing one element each

Array a		Array b
32	+	7
4	+	1
48	+	11
68	+	56
7	+	2
66	+	3



GPU: how does it works

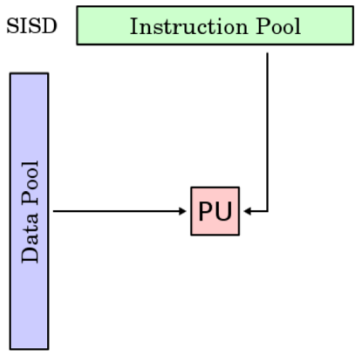
```
for (int i=0; i<N; i++) a(i)=a(i)+b(i)
```

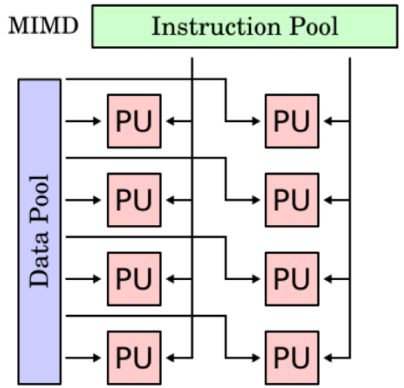
```
#include <cuda_runtime.h>
#include <iostream>


__global__ void addVectors(float *A, float *B, float *C, int
n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        C[idx] = A[idx] + B[idx];
    }
}
// Copy vectors to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

addVectors<<<numBlocks, blockSize>>>(d_A, d_B, d_C, n);

// Copy result back to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

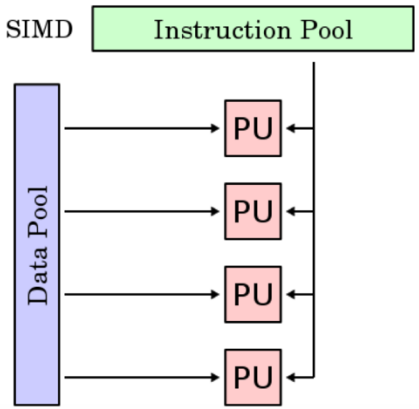
Kernel

One-Minute Foundation: SIMD and SIMT

SISD	MIMD	SIMD
Single Instruction Single Data	Multiple Instruction Multiple Data	Single Instruction Multiple Data
Uniprocessor machines	Multi-core, grid-, cloud-computing	e.g. vector processors

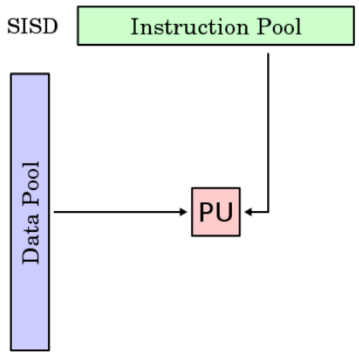


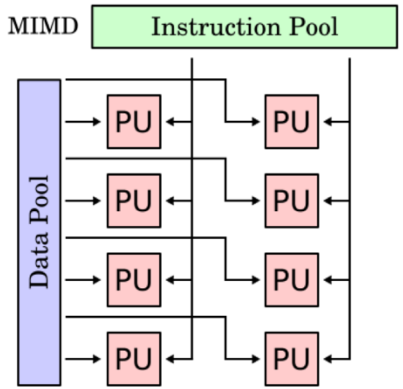


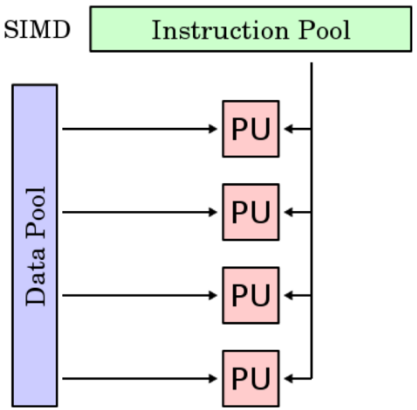


One-Minute Foundation: SIMD and SIMT

SISD	MIMD	SIMT
Single Instruction Single Data	Multiple Instruction Multiple Data	Single Instruction Multiple Threads
Uniprocessor machines	Multi-core, grid-, cloud-computing	GPUs







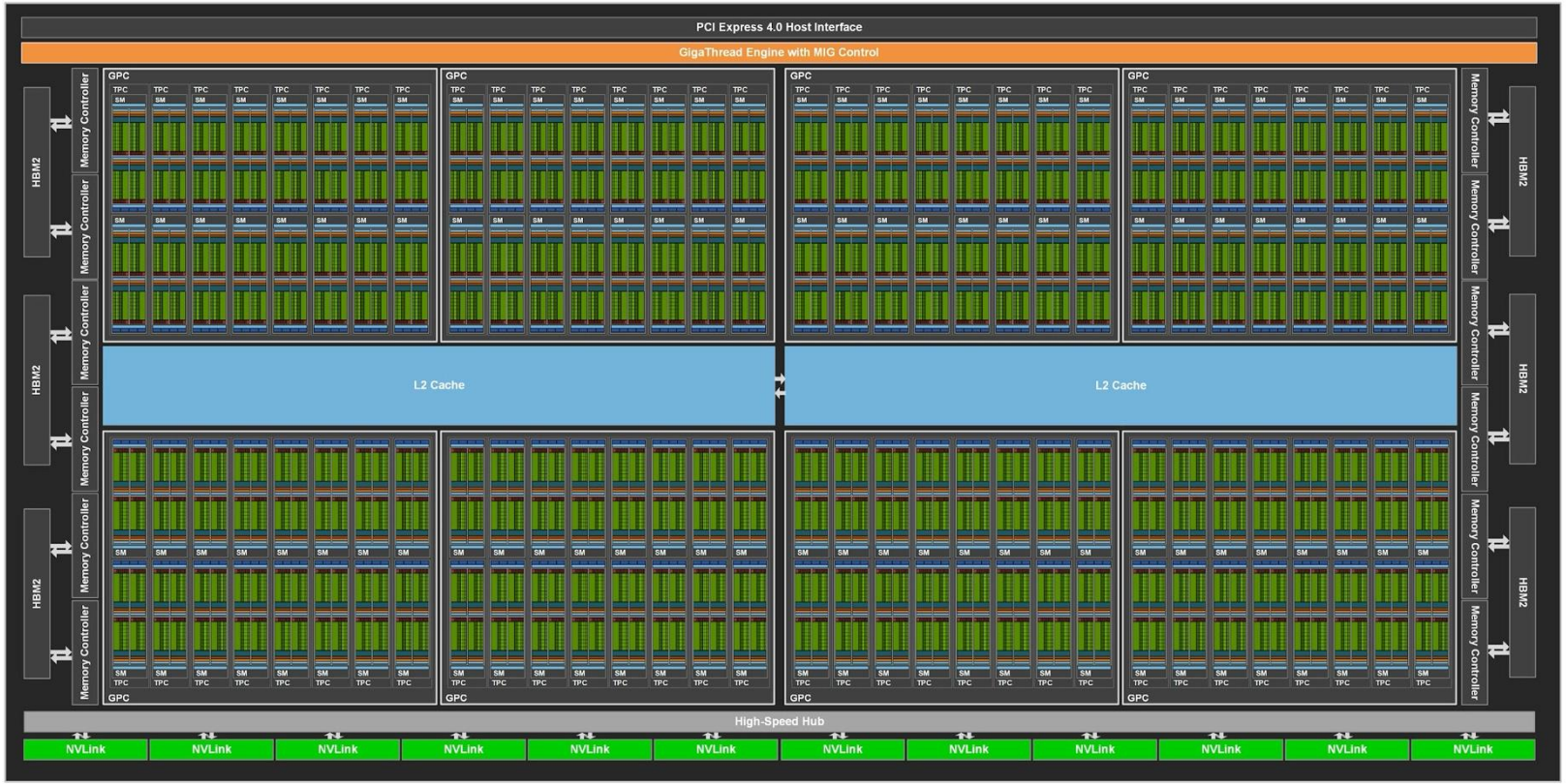
One-Minute Foundation: Single Instruction Multiple Threads

SIMT

- Similar to programming a vector processor
- Use threads instead of vectors
- No need to read data into vector register
- Only one instruction decoder available
  → all threads have to execute the same instruction

- Abstraction of vectorization:
  - Each element of vector is processed by an independent thread
  - One instruction fills entire vector
  - # of threads = vector size

GPU architecture details: NVIDIA A100



GPU architecture details: NVIDIA A100

PCI interface

Memory controller

Graphic Processing Cluster

Texture Processing Cluster

Streaming Multiprocessors

NVLINK interface

GPU architecture details: GPC

A **Graphics Processing Cluster** is a logical and physical division within the GPU.

- **TPC (Texture Processing Clusters)** A GPC is subdivided into multiple **TPCs**, each of which contains **two SMs (Streaming Multiprocessors)**. TPCs act as intermediate units that manage and optimize workloads for their SMs.
- **SMs (Streaming Multiprocessors)**: The **SM** is the fundamental compute unit of the GPU. It contains CUDA cores, Tensor Cores, and other resources for parallel execution.
- **Geometry and Raster Engines (in graphics-focused GPUs)**: For consumer GPUs, GPCs also handle geometry processing and rasterization for rendering. This aspect is less emphasized in compute-oriented GPUs like the NVIDIA A100.
- **Memory and Cache Hierarchy**: Each GPC is tightly connected to shared cache levels (such as L2 cache) and memory controllers for efficient data access.

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

GPU architecture details: GPC

A **Graphics Processing Cluster** is a logical and physical division within the GPU.

Organizing Compute Workloads: The GPC serves as a logical group that organizes the execution of threads across its SMs. This improves parallelism and resource management.

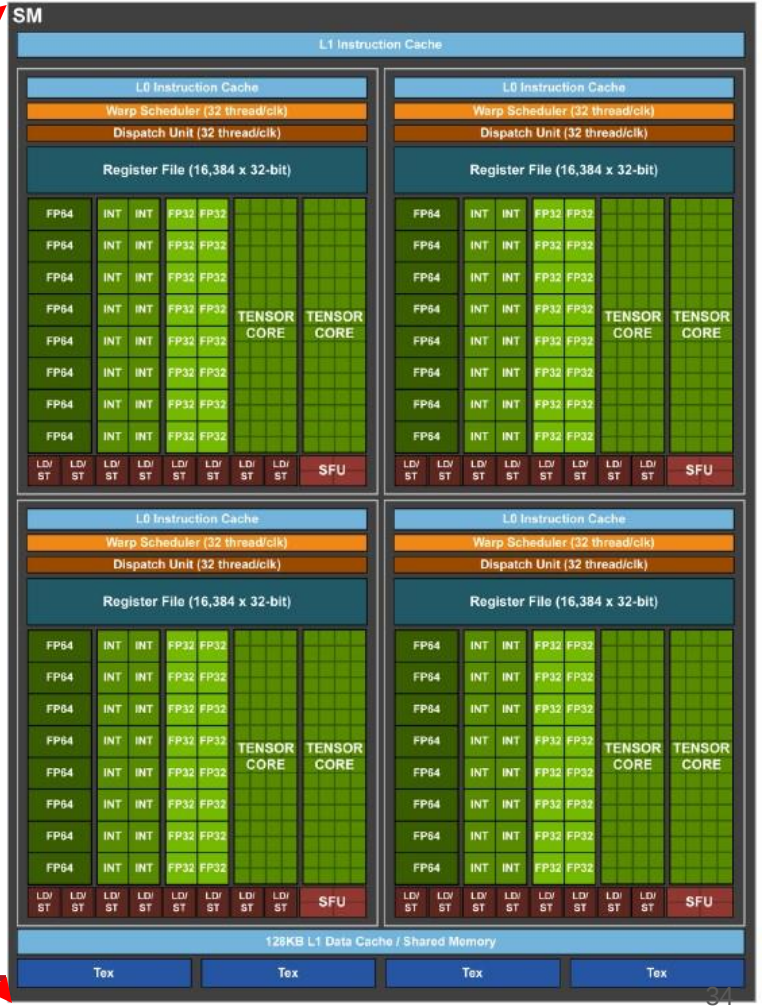
Scaling Performance: By replicating GPCs across the GPU, NVIDIA can scale computational and memory resources efficiently to meet performance demands.

Balancing Workloads: The GPU's hardware scheduler uses GPCs to distribute workloads evenly across the hardware to maximize efficiency.

- The NVIDIA Ampere A100 GPU has **7 GPCs**, as shown in the diagram.
- Each GPC contains several TPCs and a total of **108 SMs** are distributed across these GPCs.
- GPCs are connected to global memory (HBM2) and shared L2 cache for efficient data movement and computation.

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

GPU architecture details: SMs



- Each SM contains multiple **CUDA cores**, special-purpose units (like **Tensor Cores**), and resources for memory management (e.g., shared memory and L1 cache).

- It can execute multiple **warps** (groups of 32 threads) simultaneously.

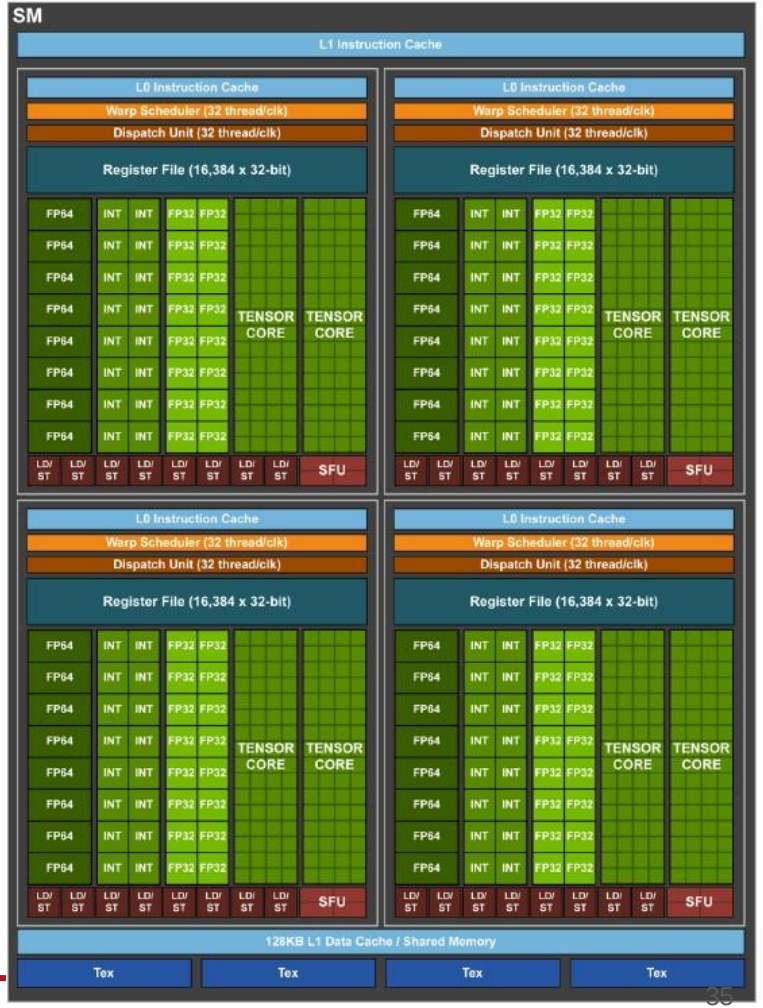
- A **CUDA core** is the basic computation unit of a GPU

GPU architecture details: SMs

• A **warp** is a group of **32 threads** that are executed together in lockstep (all threads in a warp execute the same instruction but on different data).

Warp scheduling:
• An SM can have up to **64 active** warps at any time
• In each clock cycle, the SM's scheduler selects up to **4 warps** for execution simultaneously.

Warps are a core mechanism for managing and hiding memory latency, as idle warps waiting for data can be swapped out for ready warps.



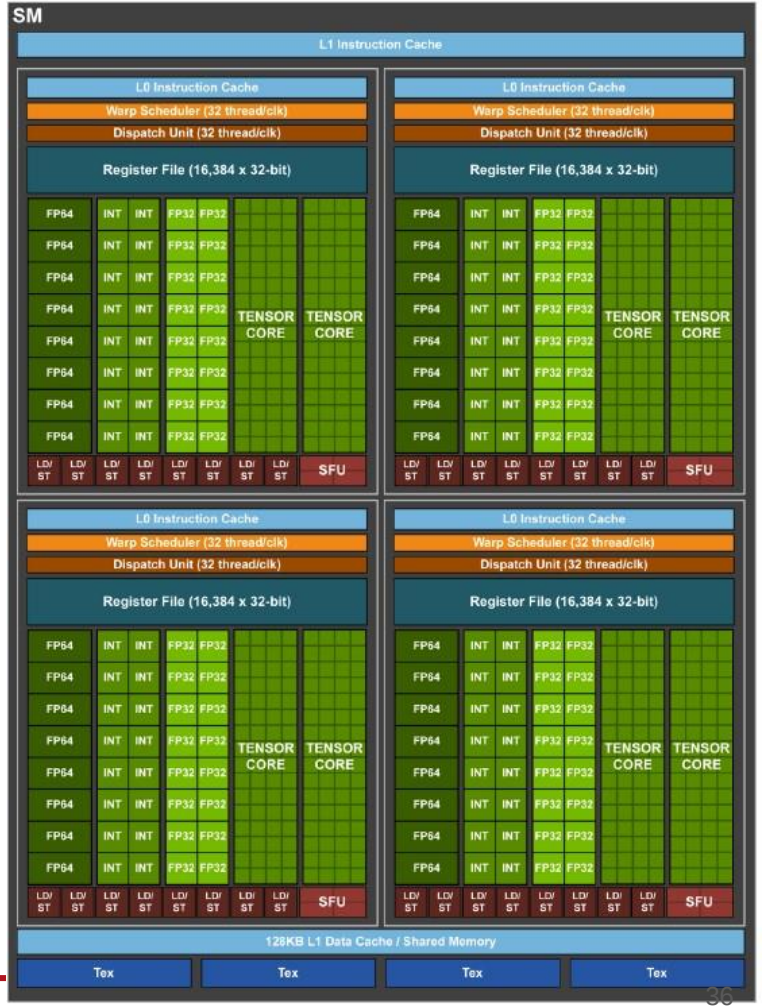
GPU architecture details: SMs

- A **CUDA core** is the basic computation unit inside an SM.

- It can execute floating-point (FP32), double precision (FP64) or integer (INT32) operations.

Historically, **Streaming Processor** (SP) referred to individual cores within an SM (essentially equivalent to CUDA cores). In modern terminology, SP is often used interchangeably with CUDA cores, representing the hardware responsible for executing basic arithmetic instructions (FP32 or INT32).

FP32/INT32 is the value that distinguish gaming vs computing GPUs



GPU architecture details: memory hierarchy

• A **CUDA core** is the basic computation unit inside an SM.

• It can execute floating-point (FP32), double precision (FP64) or integer (INT32) operations.

Historically, **Streaming Processor** (SP) referred to individual cores within an SM (essentially equivalent to CUDA cores).
In modern terminology, SP is often used interchangeably with CUDA cores, representing the hardware responsible for executing basic arithmetic instructions (FP32 or INT32).

GPU architecture details: high level components

Hardware Scheduler

The **GigaThread Engine** is a hardware scheduler in NVIDIA GPUs that manages the distribution of workloads (warps and kernels) across the Streaming Multiprocessors (SMs).

Key Functions:

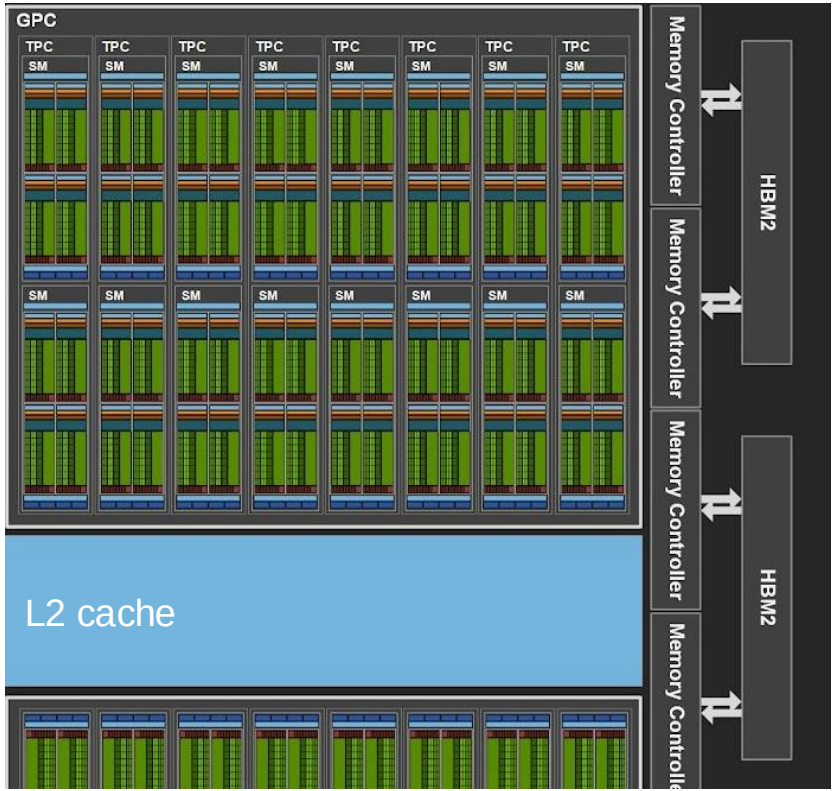
- Warp Scheduling: Determines which warps to execute at any given time on an SM.
- Kernel Overlap: Enables multiple kernels to execute concurrently on the GPU, maximizing resource utilization.
- Latency Hiding: Switches between warps to hide memory access latency or other delays.

Significance:

- It optimizes GPU performance by keeping the hardware fully utilized and minimizing idle time.

Multi Instance GPU (MIG) is a feature introduced in NVIDIA's Ampere architecture that allows a single physical GPU to be partitioned into multiple independent GPU instances. Each instance operates as a smaller, virtual GPU with its own dedicated resources (e.g., SMs, memory, cache).

GPU architecture details: memory hierarchy



Registers (Per-Thread Local Memory)
Shared Memory (Per-SM Memory)
L1 Cache
L2 Cache
Global Memory (HBM2 Memory)

Constant Memory
Texture and Surface Memory

GPU architecture details: memory hierarchy

Per-Thread Local Memory: Registers

Description:

• Registers are the fastest and most limited memory resource.
• Each thread in an SM has its private set of registers for storing local variables.

Key Features:

• Extremely low latency.
• Used for storing data that only the thread accessing it will use.

Limitations:

• If a thread uses more registers than available, it spills data into the local memory, which is slower.

Per-SM Memory: Shared Memory

Description:

• Shared memory is on-chip memory that is accessible to all threads within a block.
• It is part of the L1 cache/shared memory unit and can be dynamically partitioned.

Key Features:

• Low latency and high bandwidth.
• Ideal for thread collaboration, such as when threads in a block need to share intermediate results.

Capacity:

• In Ampere, each SM has up to 164 KB of combined shared memory and L1 cache (configurable).

GPU architecture details: memory hierarchy

L1 Cache Per-SM

Description:

- L1 cache is located within each SM and is used for frequently accessed data.
- It is tightly coupled with shared memory and dynamically partitioned between them.

Key Features:

- Acts as a fast-access layer for global and texture memory requests.
- Reduces the need to access slower memory types.

Capacity:

- Up to 164 KB per SM (shared with shared memory).

L2 Cache across SMs

Description:

- L2 cache is shared across all SMs in the GPU.
- It serves as an intermediate layer between the SMs and the **global memory** (HBM2).

Key Features:

- Reduces contention and improves memory access efficiency for all SMs.
- Optimized for data reuse across different SMs.

Capacity:

- In the A100, the L2 cache size is 40 MB, which is significantly larger than previous architectures to improve memory bandwidth efficiency.

GPU architecture details: memory hierarchy

Global Memory (HBM2 Memory)

Description:

- The **High Bandwidth Memory 2** is the main off-chip memory used to store the data needed by the GPU.
- It is an advanced DRAM (Dynamic Random Access Memory)
- Ampere GPUs like the A100 use HBM2 memory for its high bandwidth, low latency and energy efficiency.

Key Features:

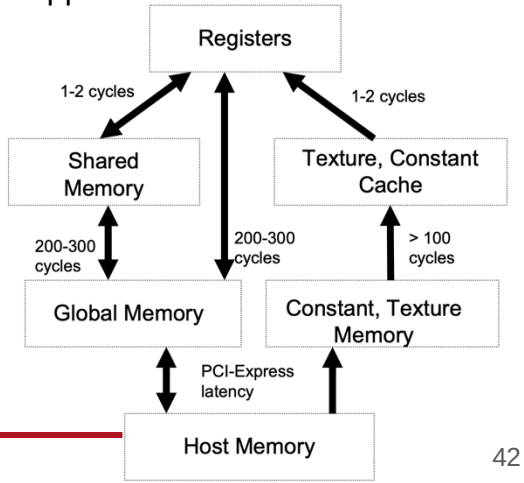
- Very large capacity, typically **40 GB** or more in the A100.
- Slower than on-chip memory (registers, shared memory, and caches) but provides a large storage space.

Bandwidth:

- Can reach up to **1.6 TB/s**, enabling the GPU to handle massive datasets for HPC and AI applications.

Memory Access Flow

1. **Thread registers** are accessed first for thread-local data.
2. If a thread needs to access data shared among other threads in the same block, it uses **shared memory**.
3. If the data isn't in shared memory, the request goes to **L1 cache**.
4. If not found in L1 cache, the request moves to the **L2 cache**.
5. If the data isn't in L2 cache, the request goes to the global memory (**HBM2**).



Programming a GPU in CUDA

```
for (int i=0; i<N; i++) a(i)=a(i)+b(i)
```

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


__global__ void addVectors(float *A, float *B, float *C, int
n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        C[idx] = A[idx] + B[idx];
    }
}
// Copy vectors to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

addVectors<<<numBlocks, blockSize>>>(d_A, d_B, d_C, n);

// Copy result back to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Kernel

NVIDIA GPU: the programmer view on an SIMT device.

In CUDA programming, the GPU's computational resources are organized hierarchically into **grids**, **blocks**, **threads.**

Grids, **blocks**, **threads** maps on Warps, SMs, TPC and GPC … and of course memory.

Threads:
• The smallest unit of execution. Each thread executes the kernel (GPU program) independently.
• Each thread has its own ID and can access its own portion of the data.

Blocks:
• A block is a group of threads that execute together and share resources like shared memory.
• Threads within a block can synchronize and communicate with each other using shared memory and thread sync primitives.
• Each block has a unique **Block ID** and contains a **1D, 2D, or 3D** grid of threads**.**

Grid:
• A grid is a collection of blocks.
• The grid organizes the execution of multiple blocks, and blocks within the grid are assigned to different SMs.
• Each grid has a unique **Grid ID** and can have a 1D, 2D, or (rarely) 3D layout of blocks.

NVIDIA GPU: the programmer view on an SIMT device.

In CUDA programming, the GPU's computational resources are organized hierarchically into **grids**, **blocks**, **threads.**

Grids, **blocks**, **threads** maps on Warps, SMs, TPC and GPC … and of course memory.

Threads per Block:
• Each block can have up to 1,024 threads (this limit varies with GPU architecture).
• Threads are indexed as `threadIdx.x, threadIdx.y, threadIdx.z,` enabling 1D, 2D, or 3D layouts.

Blocks per Grid:
• The grid can consist of multiple blocks, each identified by `blockIdx.x, blockIdx.y, blockIdx.z.`
• This allows for grids with 1D, 2D, or 3D layouts.

Mapping Dimensions:
• The combination of thread and block indices determines **the global thread ID**, which allows each thread to access a unique portion of data.

NVIDIA GPU: the programmer view on an SIMT device.

In CUDA programming, the GPU's computational resources are organized hierarchically into **grids**, **blocks**, **threads.**

Grids, **blocks**, **threads** maps on Warps, SMs, TPC and GPC … and of course memory.

Imagine we want to add two large arrays using CUDA:

- Threads per block: blockDim.x = 256 (1D block of 256 threads).
- Blocks per grid: gridDim.x. = 1024 (1D grid with 1024 blocks).

Each thread computes one element of the result:

- Thread t 's global index

$$globalIdx = blockIdx.x \times blockDim.x + threadIdx.x$$

- Thread t computes:

$$C[globalIdx] = A[globalIdx] + B[globalIdx]$$

This configuration launches `256 * 1024 = 262,144 threads` to handle the computation in parallel.

Programming a GPU in CUDA

```
for (int i=0; i<N; i++) a(i)=a(i)+b(i)
```

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


__global__ void addVectors(float *A, float *B, float *C, int
n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        C[idx] = A[idx] + B[idx];
    }
}
// Copy vectors to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

addVectors<<<numBlocks, blockSize>>>(d_A, d_B, d_C, n);

// Copy result back to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Thread Global ID

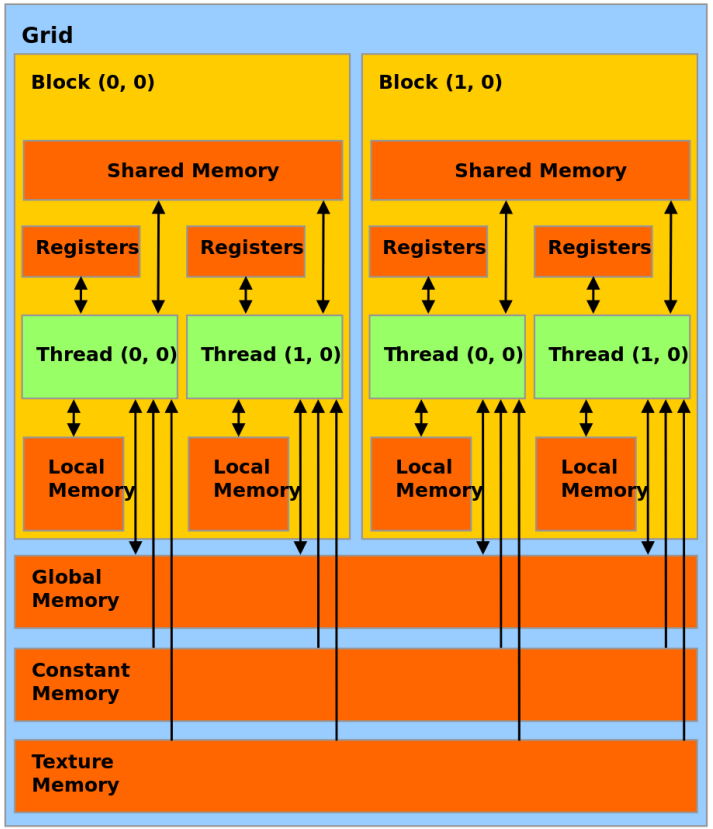
NVIDIA GPU: the programmer view on memory

Memory Sharing:

- Threads within the same block can communicate through shared memory, which has low latency.
- Threads in different blocks cannot directly share memory, so they must use global memory, which is slower.

Synchronization:

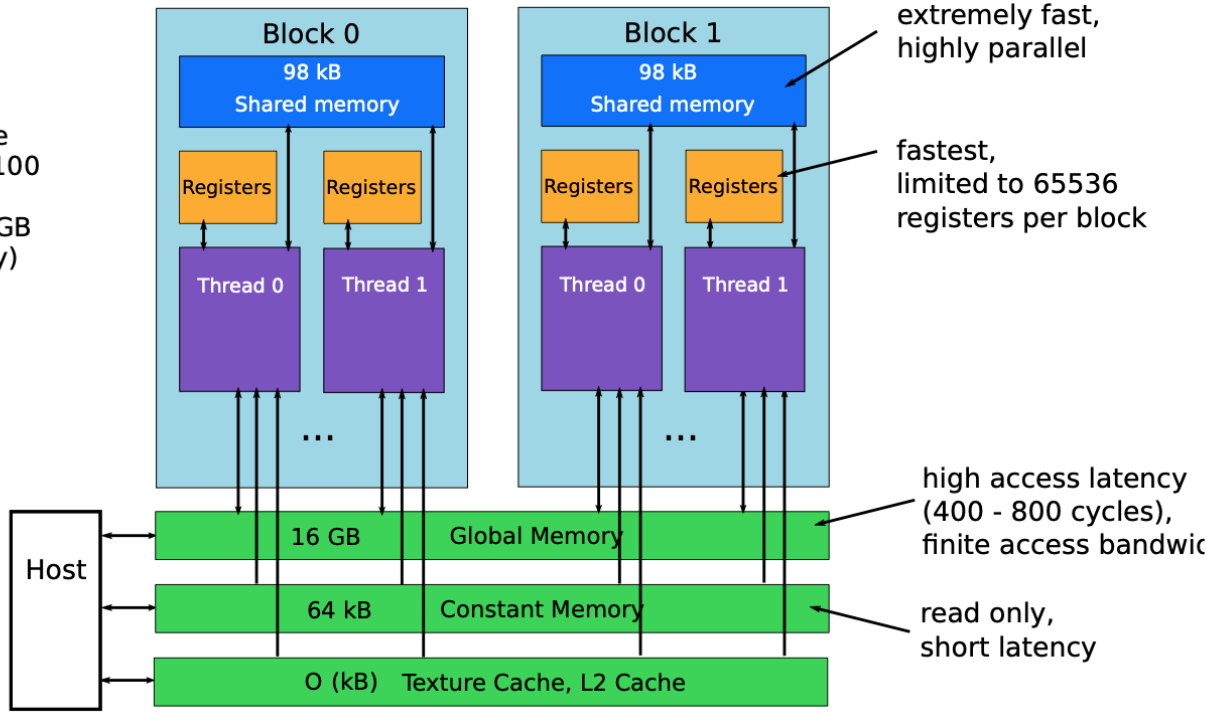
- Threads in a block can synchronize with each other using `__syncthreads()`, ensuring all threads reach a certain point before continuing.
- Blocks are independent and cannot synchronize directly.



NVIDIA GPU: the programmer view on memory

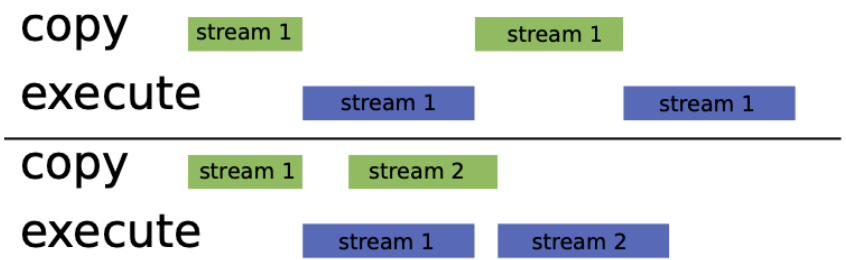
Specs from the
16 GB Tesla V100

(A100 has 40 GB
global memory)



Programming a GPU in CUDA

- Independent tasks can be executed concurrently:
  - Computation on host
  - Computation on device
  - Memory transfers
- “Async” function calls → pipeline of tasks only synchronized within one stream
- `cudaDeviceSynchronize()`: waits for all streams to finish



NVIDIA GPU Arch: Summing up

Execution Structure

- ✓ Threads (CUDA core)
- ✓ Warp
- ✓ Streaming Multiprocessor
- ✓ Grid and Blocks (GPC/TPC)

Memory layout

- ✓ Registers
- ✓ Shared Memory
- ✓ L1 Cache
- ✓ L2 Cache
- ✓ Global Memory (HBM2)

Component	Value
L1 Cache / Shared Memory	Up to **164 KB** per SM (configurable).
L2 Cache	**40 MB** (shared across all SMs).
Global Memory	**40 GB** HBM2.
Memory Bandwidth	Up to **1.6 TB/s**.
CUDA Cores per SM	**128** CUDA cores.
Tensor Cores per SM	**4** Tensor Cores.
Total SMs	**108** SMs.
Maximum Threads per SM	**2048** threads (64 warps × 32 threads).
Maximum Threads per Block	**1024** threads.
Maximum Warps per SM	**64** warps (managed by warp scheduler).
Maximum Active Warps per SM	**4** warps

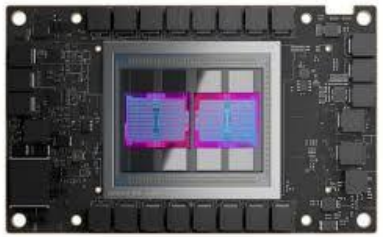
It achieves massive parallelism with **108 SMs**, each equipped with **128 CUDA cores**, scheduling up to **64 warps per SM** and executing multiple warps concurrently to hide latency.

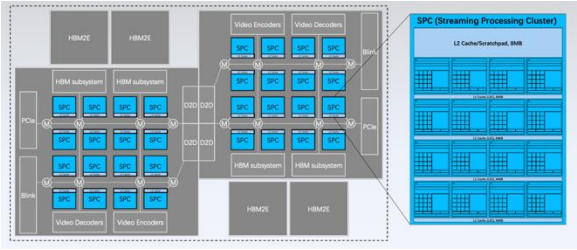
NVIDIA GPU: CUDA tools

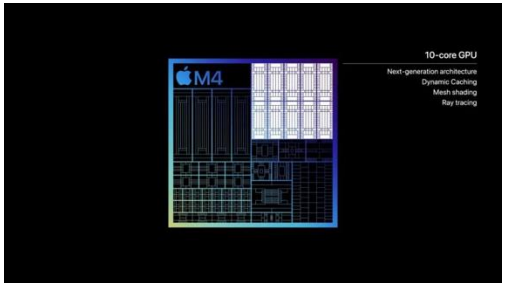
```
[gtaffoni@len61 ~]$ nvidia-smi
Mon Dec  2 23:45:30 2024
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.103.01   Driver Version: 470.103.01   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:83:00.0 Off |                    0 |
| N/A   24C    P0    60W / 149W |      0MiB / 11441MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K80           Off  | 00000000:84:00.0 Off |                    0 |
| N/A   35C    P0    75W / 149W |      0MiB / 11441MiB |     98%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

GPU are not only NVIDIA

- AMD's Instinct MI200
- Intel Ponte Vecchio
- Apple Silicon (M1/M2/M3)



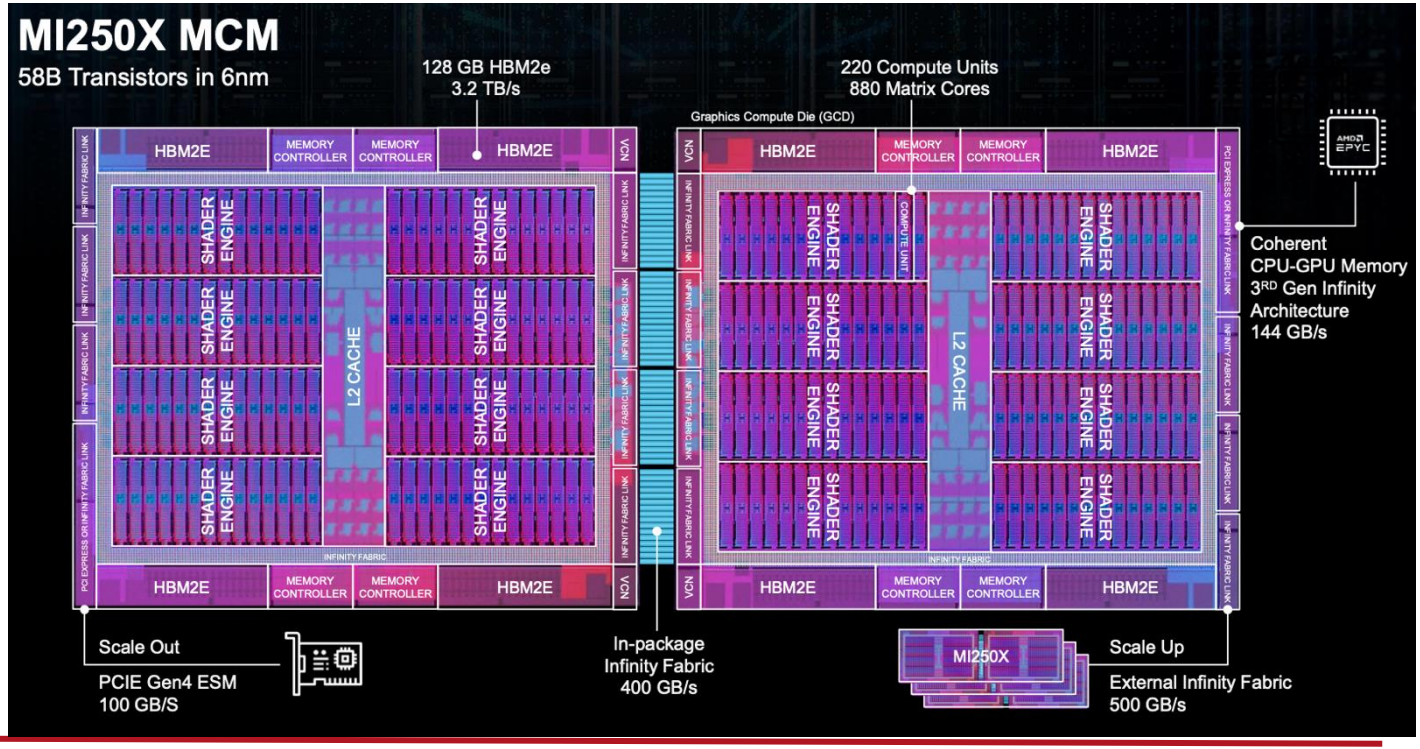




- *Biren BR100 GPGPU*

AMD GPU Architecture

AMD GPUs are designed around the RDNA architecture (for consumer GPUs) and the **CDNA** architecture (for compute GPUs like the MI200 series).



AMD GPU Architecture

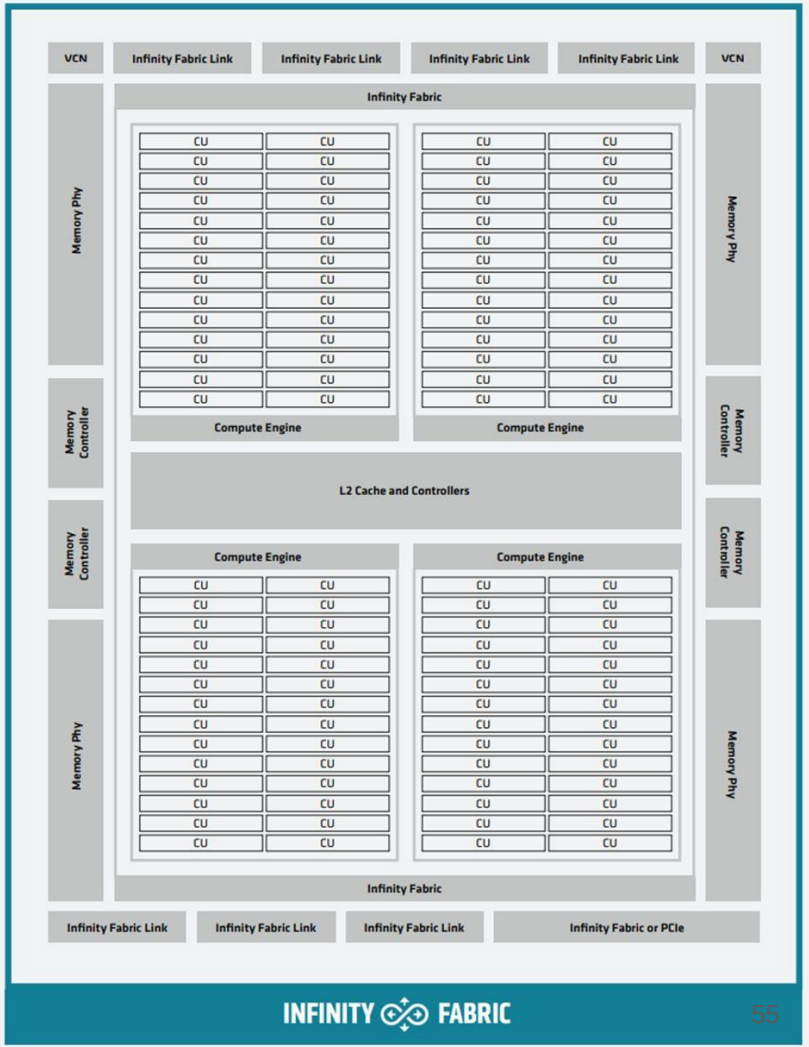
Compute Units (CUs)

Wavefronts

Workgroups

Graphics Processing Clusters (GPC) / Shader Engines

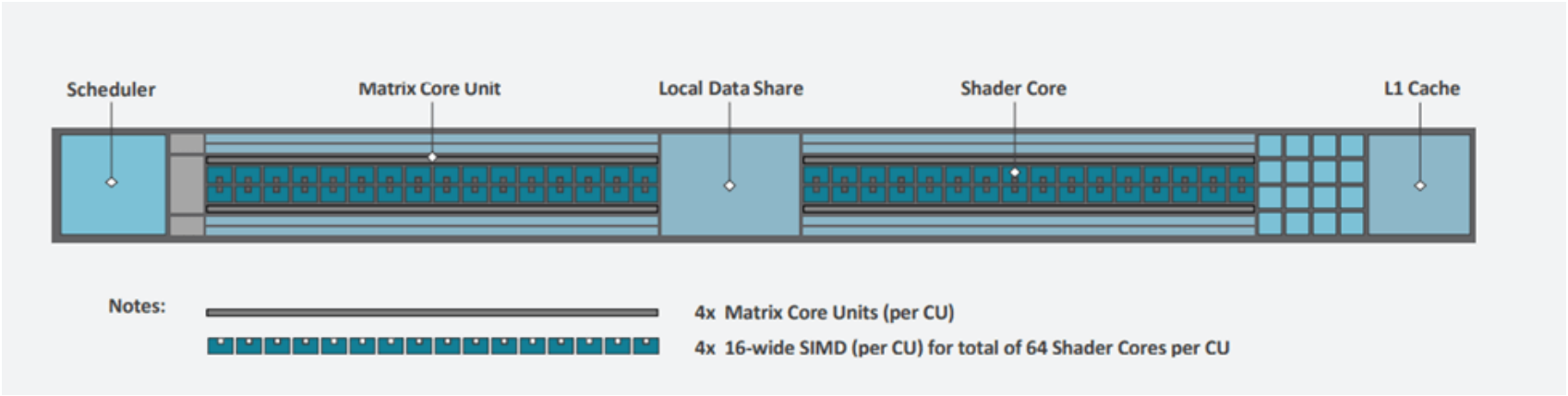
Infinity Cache



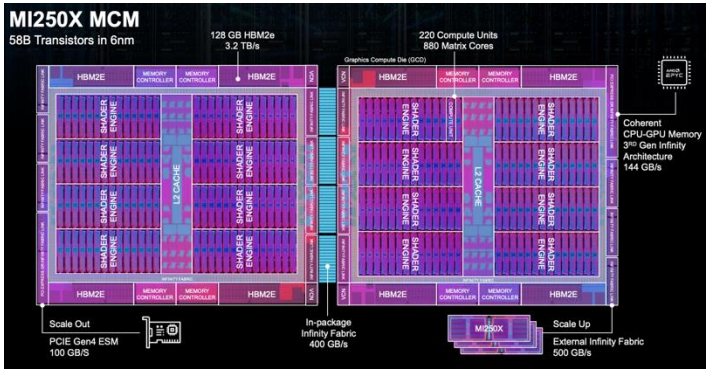
AMD GPU Architecture

Compute Units (CUs)
The Compute Unit (CU) is the fundamental processing block of AMD GPUs, similar to NVIDIA's Streaming Multiprocessor
Each CU consists of:

- **Stream Processors** responsible for executing the actual computations.
- **Scalar Units**: Handle non-vector operations.
- **Vector Units**: Optimized for parallel SIMD (Single Instruction, Multiple Data) operations.
- **L1 Cache** and Local Data Store (LDS): Provides fast, localized memory access.



AMD GPU Architecture



Wavefronts
The wavefront is AMD's equivalent of a warp in NVIDIA GPUs.
A wavefront typically consists of **64 threads**.
Threads in a wavefront execute in lockstep, following the SIMD model.

Workgroups
- Workgroups in AMD are equivalent to blocks in NVIDIA.
- A workgroup consists of multiple wavefronts and executes on a single CU.

Graphics Processing Clusters (GPC) / Shader Engines
- AMD divides its GPUs into Shader Engines, which are similar to NVIDIA's Graphics Processing Clusters (GPC).
- Each Shader Engine contains multiple CUs and manages workloads across them.

Infinity Cache
- Introduced in RDNA 2 and CDNA architectures, the Infinity Cache is a large, high-speed cache shared across the GPU.
- It minimizes latency and reduces the need to access slower global memory, improving bandwidth efficiency.

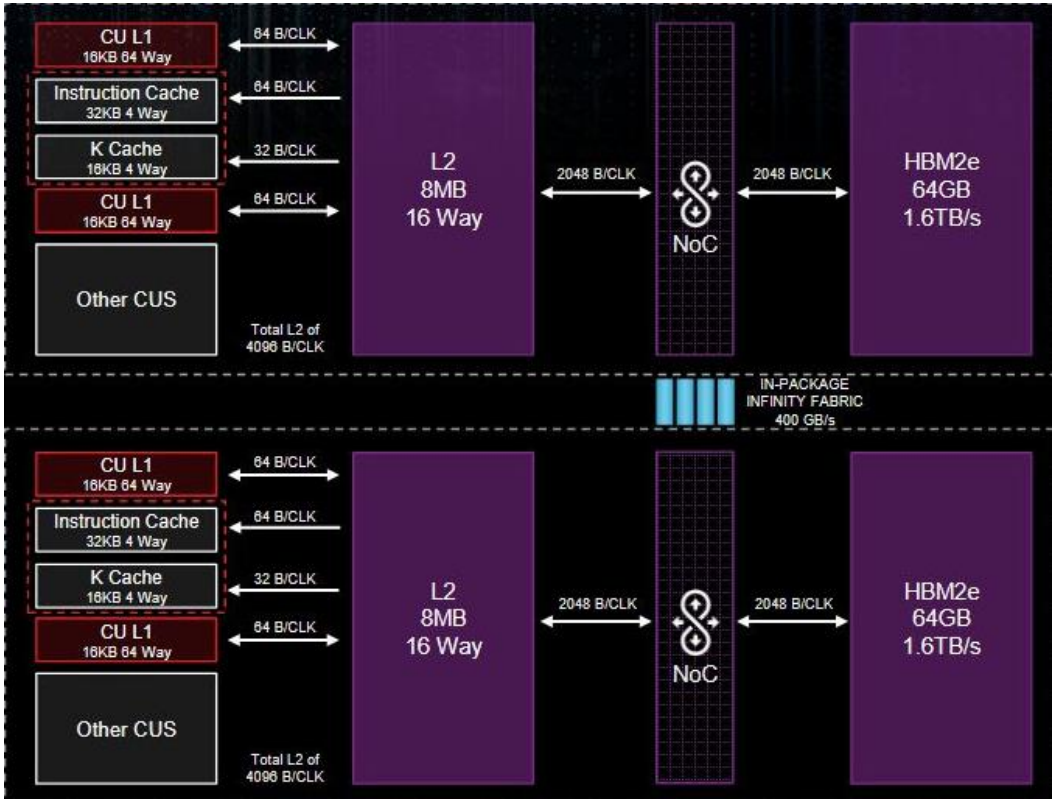
AMD GPU Architecture

AMD GPUs have a hierarchical memory system optimized for high-throughput parallelism

Registers:
- Local to each thread in a wavefront, used for storing thread-private data.
- Fastest memory layer but limited in size.

LDS (Local Data Store):
- Equivalent to shared memory in NVIDIA GPUs.
- Provides fast, low-latency memory shared by all threads in a CU.

L1 Cache:
- Local to each CU, serving threads within the CU.
- Optimized for frequently accessed data.



AMD GPU Architecture

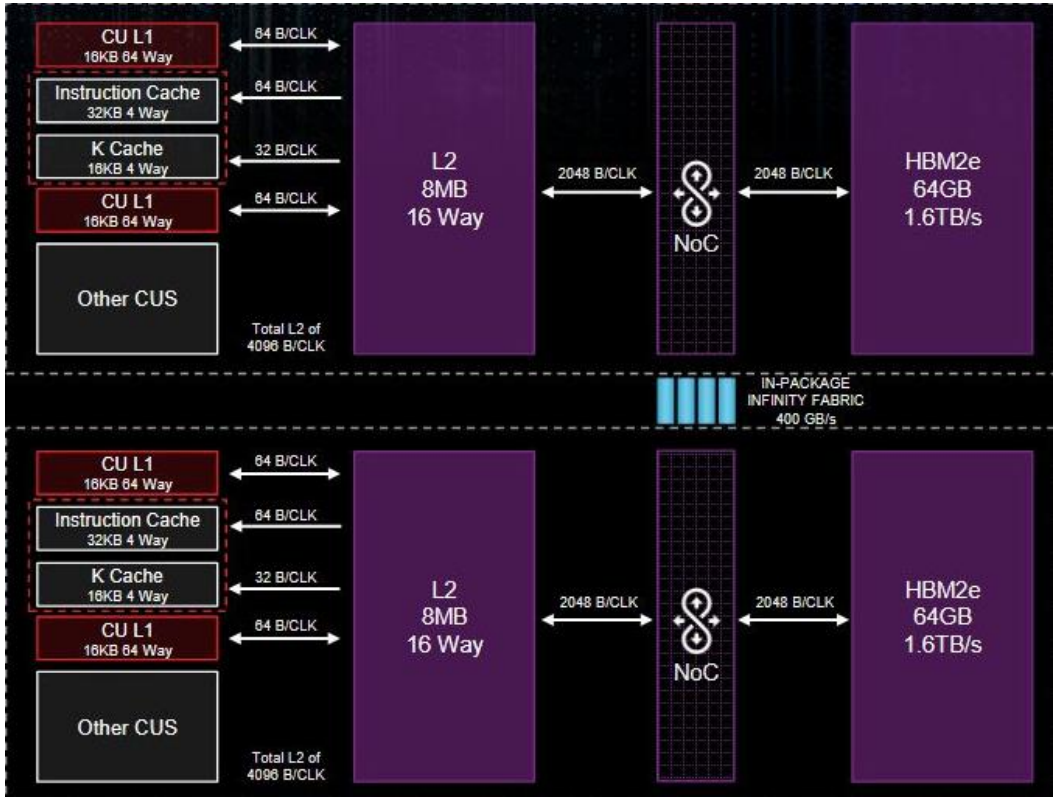
AMD GPUs have a hierarchical memory system optimized for high-throughput parallelism:

L2 Cache:

- Shared across Shader Engines.
- Serves as an intermediary between the CUs and global memory.

Global Memory (HBM2/Infinity Fabric):

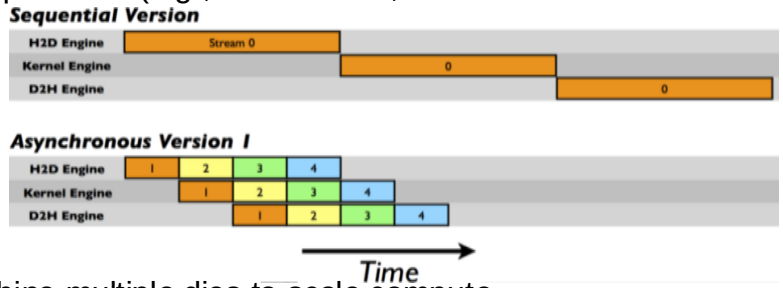
- AMD GPUs, particularly in the CDNA architecture, use HBM2 memory for high bandwidth.
- For RDNA GPUs, global memory is typically GDDR6.
- The **Infinity Cache** reduces reliance on global memory by caching frequently accessed data.



AMD GPU Architectural features

SIMD Execution
- AMD GPUs use SIMD execution extensively:
- Each wavefront executes a single instruction across its 64 threads.
- This makes them highly efficient for vectorized operations.

Asynchronous Compute
- AMD GPUs are designed for asynchronous compute, allowing multiple tasks (e.g., AI inference, rendering) to run concurrently without interfering with each other.
- Overlapping Computations and Data Movements



Scalability with Chiplets
- Recent AMD GPUs (like the MI200) use chiplet designs, which combine multiple dies to scale compute and memory resources efficiently.
- The Infinity Fabric interconnects chiplets, enabling high-speed communication.

AMD GPUs vs NVIDIA GPUs

Feature	AMD GPUs	NVIDIA GPUs
Processing Unit	Compute Unit (CU)	Streaming Multiprocessor (SM)
Threads per Warp/Wavefront	64 threads per wavefront	32 threads per warp
Shared Memory Equivalent	Local Data Store (LDS)	Shared Memory
Cache System	Infinity Cache, L1,L2	L1/L2 Caches
Global Memory	HBM2 or GDDR6 (Infinity Fabric)	HBM2 or GDDR6
Parallelism	SIMD Wavefront Execution	SIMD Warp Execution
Scalability	Chiplet Design, Infinity Fabric	Monolithic GPU (in most designs)

Programming an AMD GPU: ROCm(HIP)

```
for (int i=0; i<N; i++) a(i)=a(i)+b(i)
```

```
#include <hip/hip_runtime.h>
#include <iostream>

__global__ void addVectors(float *A, float *B, float *C, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        C[idx] = A[idx] + B[idx];
    }
}
```

```
// Allocate device memory
float *d_A, *d_B, *d_C;
hipMalloc(&d_A, size);
hipMalloc(&d_B, size);
hipMalloc(&d_C, size);

// Copy vectors to device
hipMemcpy(d_A, h_A, size, hipMemcpyHostToDevice);
hipMemcpy(d_B, h_B, size, hipMemcpyHostToDevice);
```

Programming an Apple Silicon (Metal)

```
for (int i=0; i<N; i++) a(i)=a(i)+b(i)
```

```
#include <metal_stdlib>
using namespace metal;

kernel void addVectors(const device float* A [[ buffer(0) ]],
                       const device float* B [[ buffer(1) ]],
                       device float* C [[ buffer(2) ]],
                       uint id [[ thread_position_in_grid ]]) {
    C[id] = A[id] + B[id];
}
```

```
// Creazione dei buffer
auto bufferA = device->newBuffer(A.data(), n * sizeof(float), MTL::ResourceStorageModeShared);
auto bufferB = device->newBuffer(B.data(), n * sizeof(float), MTL::ResourceStorageModeShared);
auto bufferC = device->newBuffer(C.data(), n * sizeof(float), MTL::ResourceStorageModeShared);
```

```
computeEncoder->setBuffer(bufferB, 0, 1);
computeEncoder->setBuffer(bufferC, 0, 2);

// Configurazione dei thread
MTL::Size threadsPerThreadgroup(256, 1, 1);
MTL::Size threadgroups((n + 255) / 256, 1, 1);
computeEncoder->dispatchThreadgroups(threadgroups, threadsPerThreadgro

// Concludere l'encoding e avviare i comandi
computeEncoder->endEncoding();
commandBuffer->commit();
commandBuffer->waitUntilCompleted();
```

MPSoC == Shared Memory

How can I get all info about my GPU?

```
$ nvaccelinfo
```

```
CUDA Driver Version:           12010
NVRM version:                  NVIDIA UNIX x86_64 Kernel

Device Number:                 0
Device Name:                   NVIDIA A100-SXM-64GB
Device Revision Number:        8.0
Global Memory Size:            68099571712
Number of Multiprocessors:     124
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    1395 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   Yes
Memory Clock Rate:             1593 MHz
Memory Bus Width:              4096 bits
L2 Cache Size:                 33554432 bytes
Max Threads Per SMP:           2048
Async Engines:                 5
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     Yes
Preemption Supported:          Yes
Cooperative Launch:            Yes
Default Target:                cc80
```

INAF
ISTITUTO NAZIONALE
DI ASTROFISICA

THANK YOU.

Now, it’s your time
QUESTIONS?