UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Programming

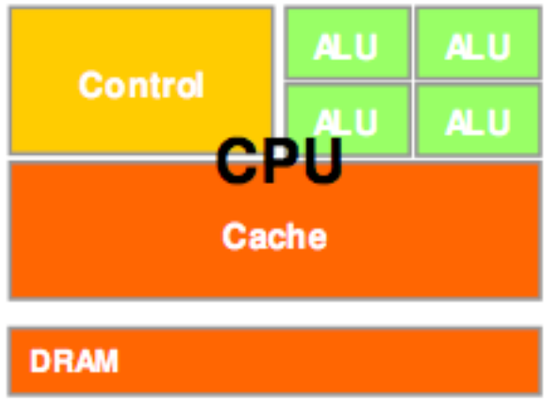
Prof. Marco Bertini

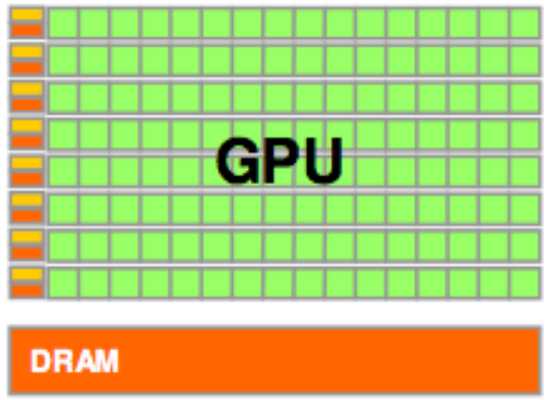
UNIVERSITÀ
DEGLI STUDI
FIRENZE

Data parallelism: GPU computing

CPUs vs. GPUs

- The design of a CPU is optimized for sequential code performance.
  - out-of-order execution, branch-prediction
  - large cache memories to reduce latency in memory access
  - multi-core
- GPUs:
  - many-core
  - massive floating point computations for video games
  - much larger bandwidth in memory access
  - no branch prediction or too much control logic: just compute





CPUs have latency oriented design:

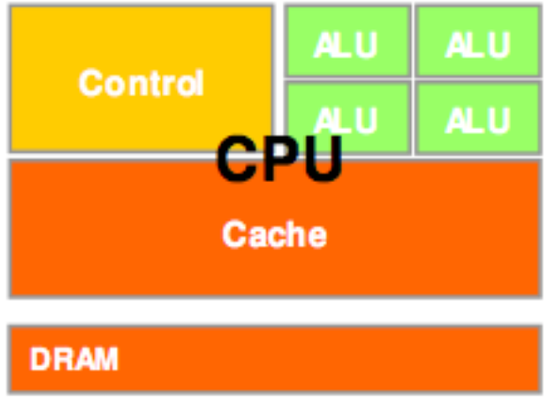
- Large caches convert long latency RAM access to short latency
- Branch pred., OoOE, operand forwarding reduce instructions latency
- Powerful ALU for reduced operation latency

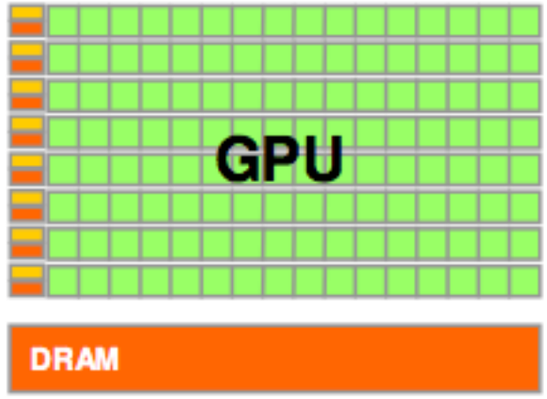
GPUs have a throughput oriented design:

- Small caches to boost RAM throughput
- Simple control (no operand forwarding, branch prediction, etc.)
- Energy efficient ALU (long latency but heavily pipelined for high throughput)
- Require massive # threads to tolerate latencies

- multi-core

- GPUs:
  - many-core
  - massive floating point computations for video games
  - much larger bandwidth in memory access
  - no branch prediction or too much control logic: just compute





CPUs and GPUs

- GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well;

- One should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.

- We are going to deal with **heterogenous architectures**: CPUs + GPUs.

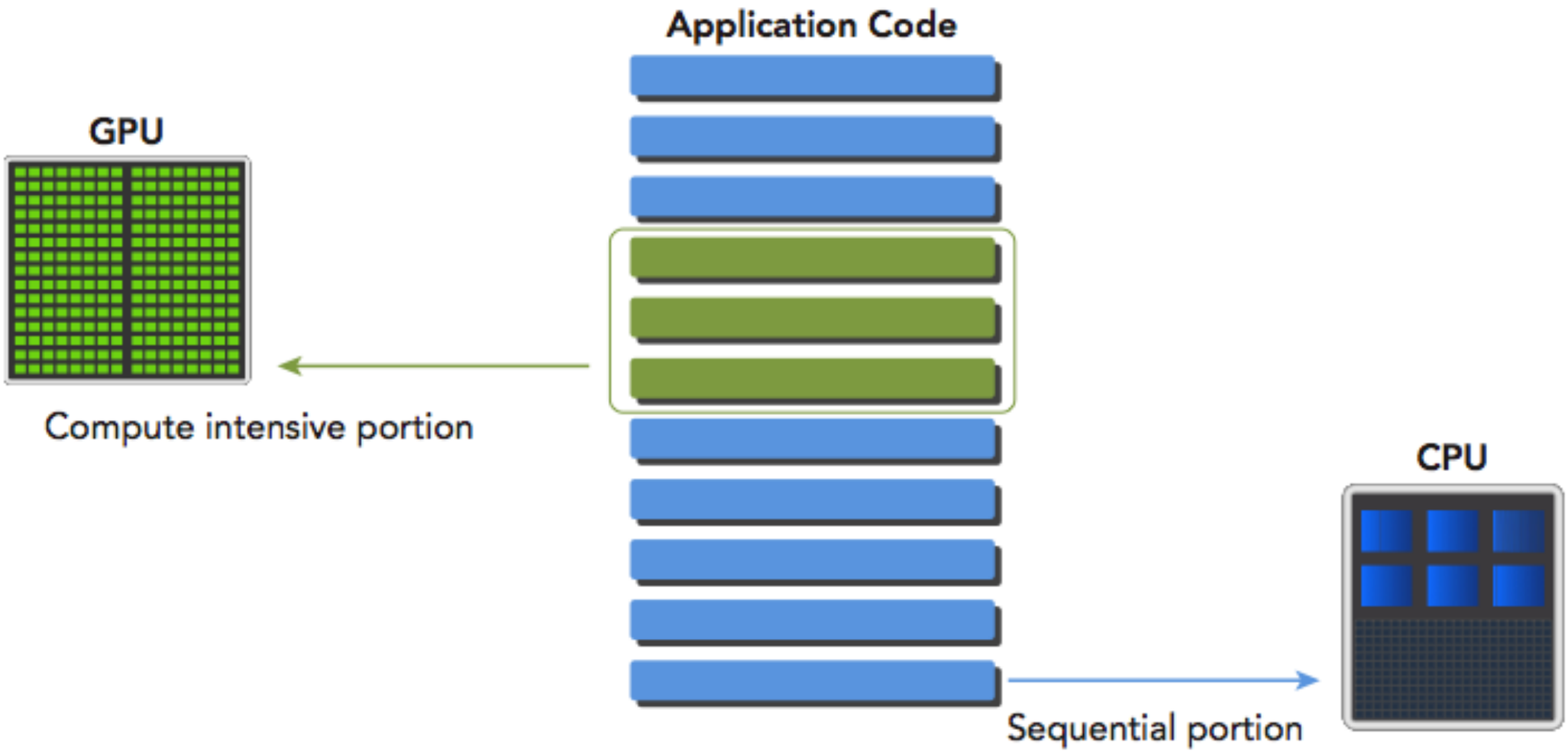
Heterogeneous Computing

- CPU computing is good for control-intensive tasks, and GPU computing is good for data-parallel computation-intensive tasks.

- The CPU is optimized for dynamic workloads marked by short sequences of computational operations and unpredictable control flow;

- GPUs aim at workloads that are dominated by computational tasks with simple control flow.

Heterogeneous Computing

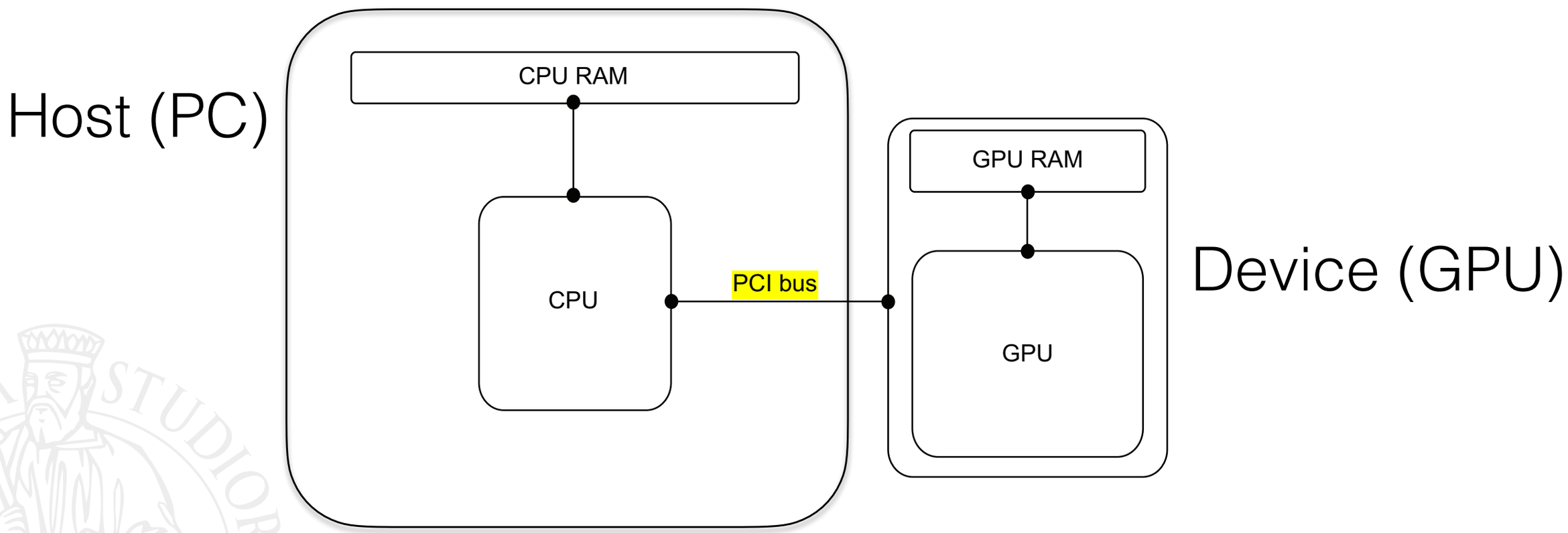


Heterogeneous Computing

- A heterogeneous application consists of two parts:
  - Host code
  - Device code
- **Host** code runs on **CPUs** and **device** code runs on **GPUs**.

GPU systems

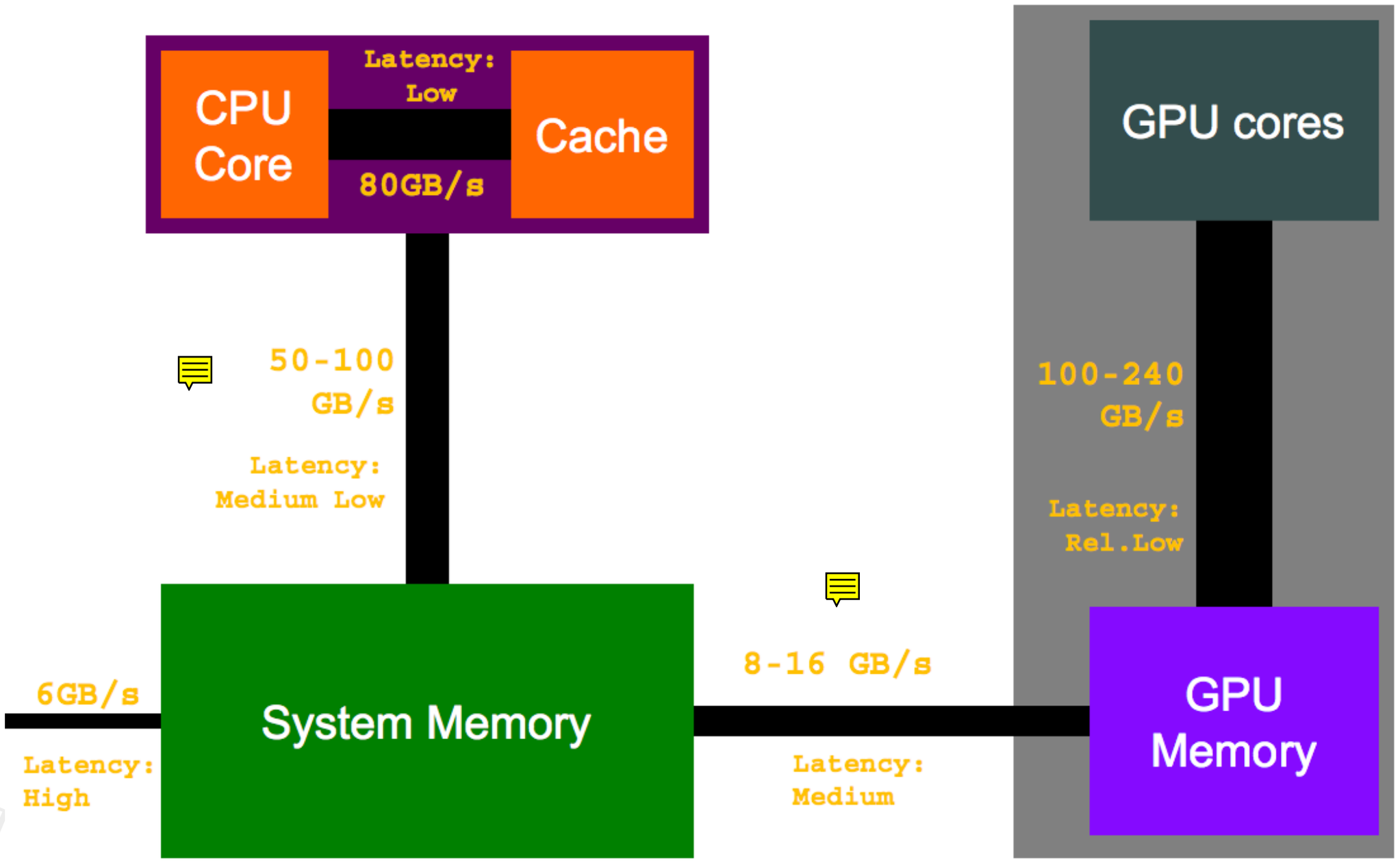
- HPC systems are increasingly coming equipped with multiple GPUs
- Workstation s may accomodate two GPUs, servers can go up to 8.



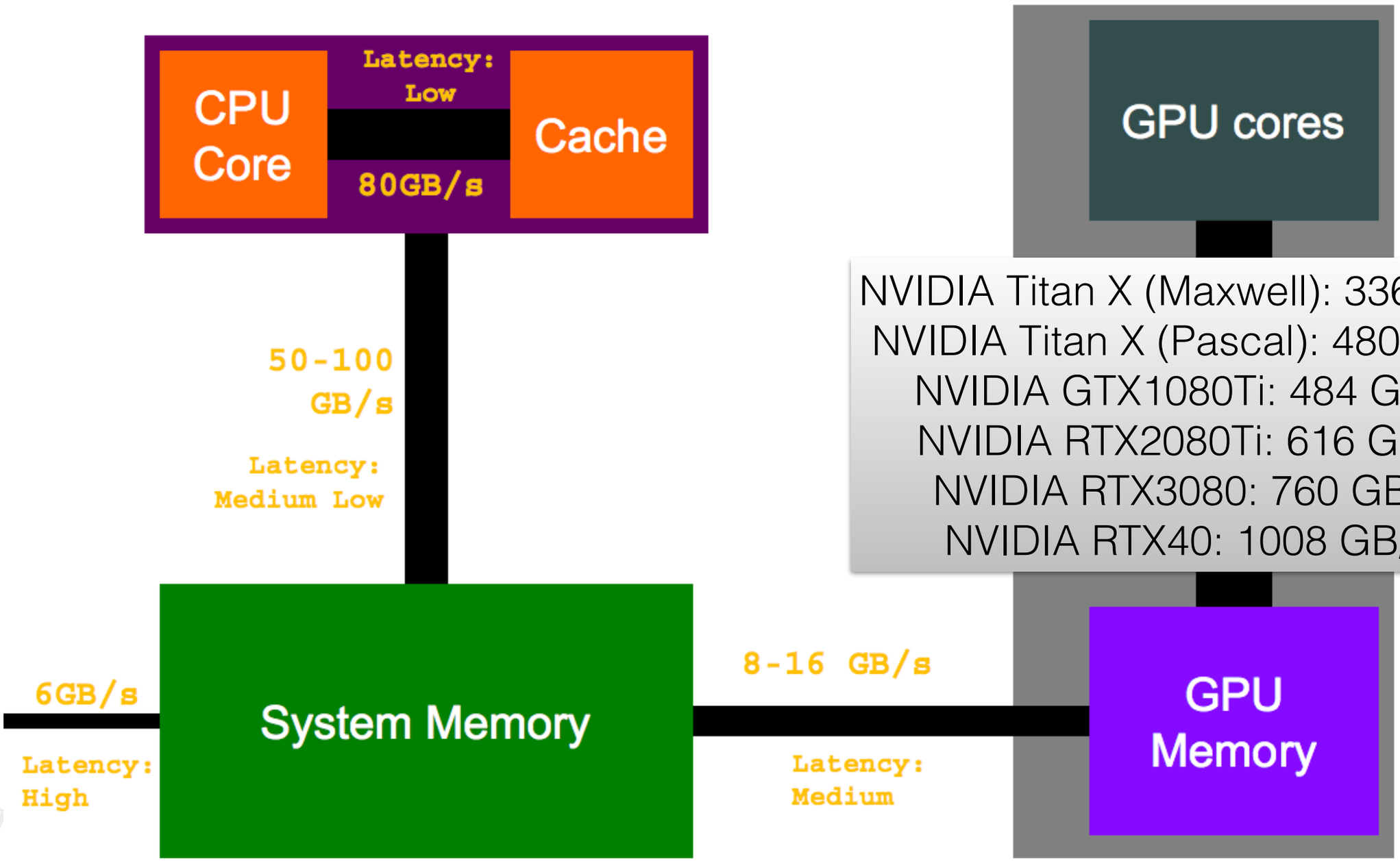
Integrated vs. dedicated GPUs

- GPUs are best described as accelerators, long used in the computing world.
  - e.g. 8087 math co-processor that helped 8086
  - An accelerator (hardware) is a special-purpose device that supplements the main general-purpose CPU in speeding up certain operations.
- *Integrated* GPUs — A graphics processor engine that is contained on the CPU. E.g. Apple Ax and M*x* processors have them, Intel and AMD provide them.
- *Dedicated*/*Discrete* GPUs — A GPU contained on a separate peripheral card. E.g. NVIDIA cards. Attached using a Peripheral Component Interconnect (PCI) slot.

Bandwidth in a CPU-GPU System



Bandwidth in a CPU-GPU System



Threads

- Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off CPU execution channels to provide multithreading capability. Context switches are slow and expensive.

  We deal with a few tens of threads per CPU, depending on HyperThreading.

- Threads on GPUs are extremely lightweight. In a typical system, thousands of threads are queued up for work. If the GPU must wait on one group of threads, it simply begins executing work on another.

  We deal with tens of thousands of threads per GPU.

SIMT

- GPU is a SIMD (Single Instruction, Multiple Data) device → it works on “streams” of data
  - Each “GPU thread” executes one general instruction on the stream of data that the GPU is assigned to process
  - NVIDIA calls this model SIMT (single instruction multiple thread)
- The SIMT architecture is similar to SIMD. Both implement parallelism by broadcasting the same instruction to multiple execution units.

  A key difference is that SIMD requires that all vector elements in a vector execute together in a unified synchronous group, whereas SIMT allows multiple threads in the same group to execute independently.

SIMT

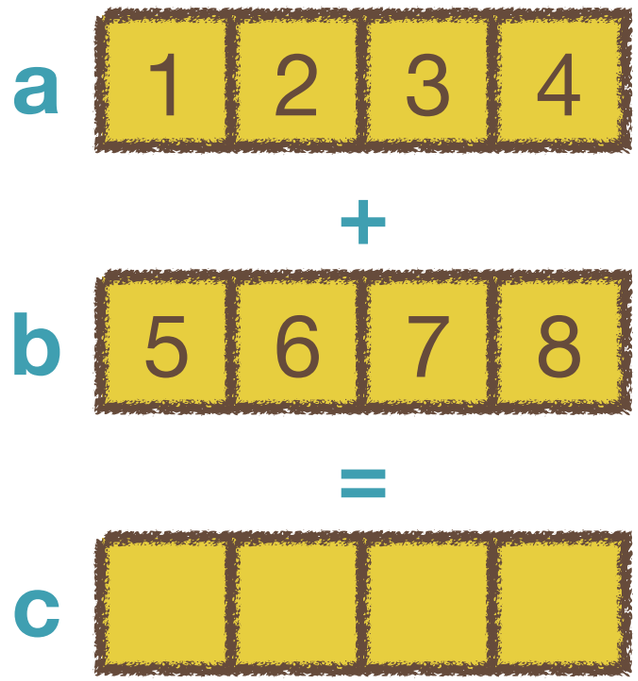
- The SIMT model includes three key features that SIMD does not:

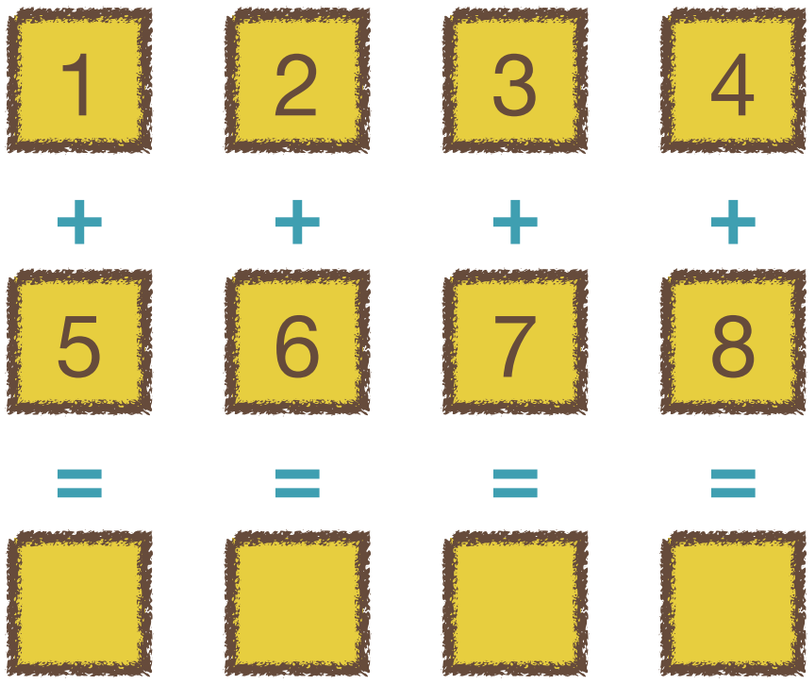
- Each thread has its own instruction address counter.

- Each thread has its own register state, i.e. it has a register set.

- Each thread can have an independent execution path.

SIMD (SSE) view vs. SIMT (CUDA) view






```
__m128 a = _mm_set_ps (4, 3, 2, 1);
__m128 b = _mm_set_ps (8, 7, 6, 5);
__m128 c = _mm_add_ps (a, b);
```

```
float a[4] = {1, 2, 3, 4},
      b[4] = {5, 6, 7, 8}, c[4];

// …
// Define a compute kernel, which
// a fine-grained thread executes.
{
  int id = … ;  // my thread ID
  c[id] = a[id] + b[id];
}
```

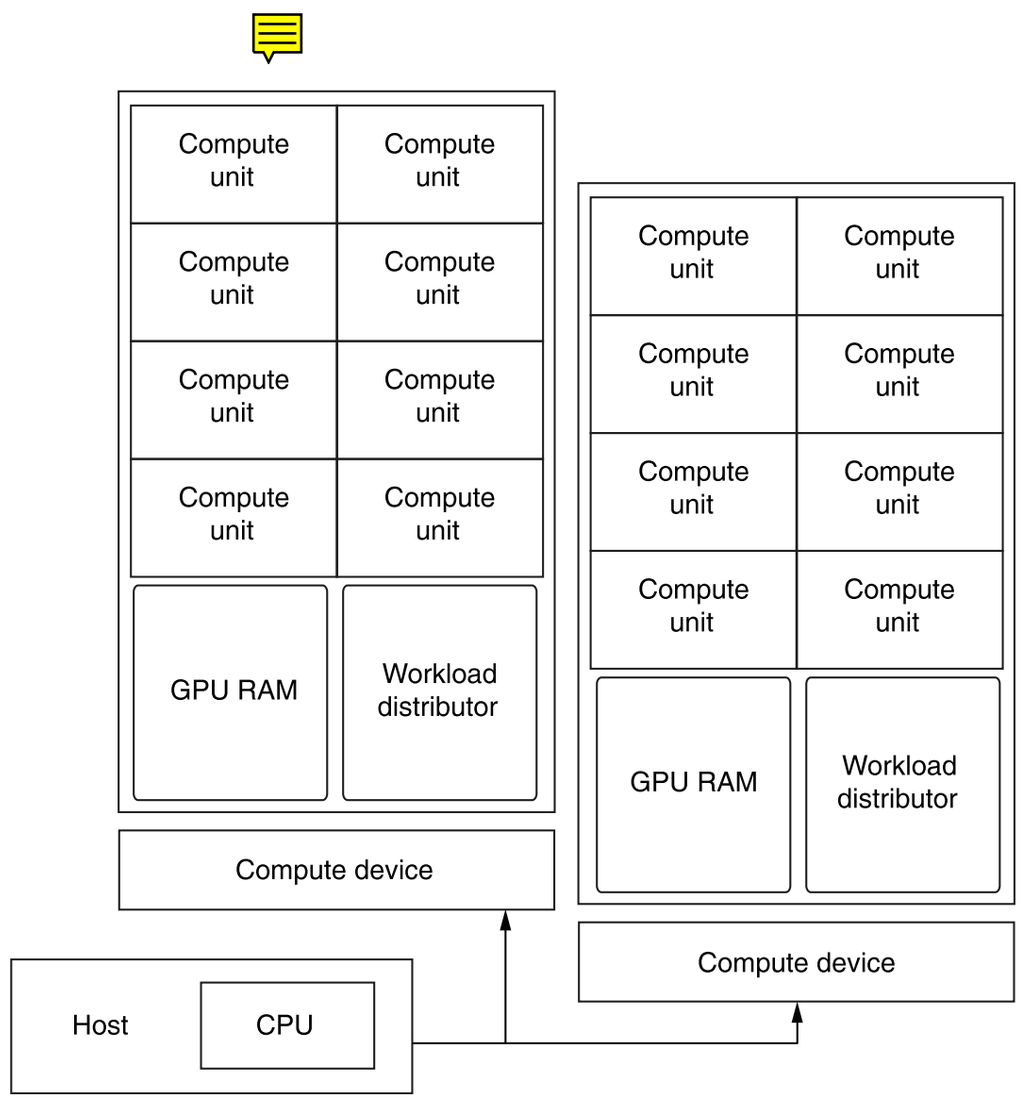
UNIVERSITÀ DEGLI STUDI FIRENZE

H/w terminology

Host / CPU	OpenCL / AMD	CUDA / NVIDIA
CPU	Compute device / GPU	GPU
Multiprocessor	Compute Unit (CU)	Streaming Multiprocessor (SM)
Processing core (core)	Processing Element (PE)	CUDA core / Compute core
Thread	Work Item	Thread
Vector / SIMD	Vector & subgroups (SIMT warp)	SIMT warp

GPU generic architecture

- A GPU is composed of
  - GPU RAM (also known as global memory)
  - Workload distributor
  - Compute units (CUs), called Streaming Multiprocessors (SMs) in CUDA



Compute Units (CUs)

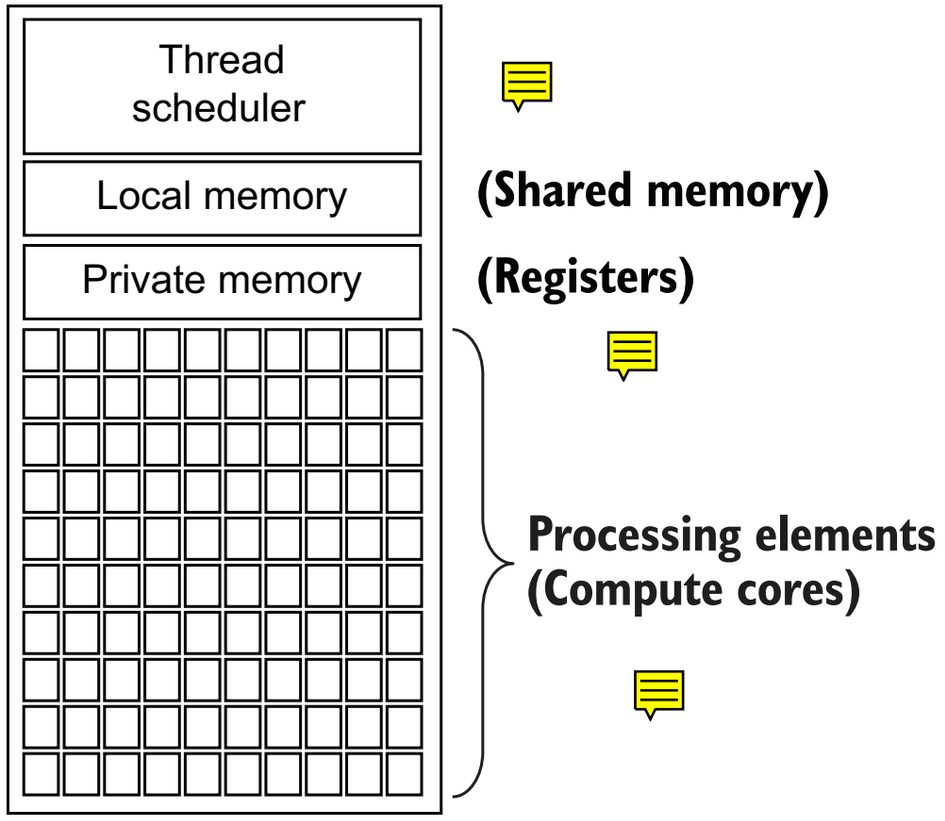
- CUs have their own internal architecture, often referred to as the microarchitecture. Instructions and data received from the CPU are processed by the workload distributor. The distributor coordinates instruction execution and data movement onto and off of the CUs.

- The achievable performance of a GPU depends on:
  - Global memory bandwidth
  - Compute unit bandwidth
  - The number of CUs

Compute Units (CUs)

- Each CU contains multiple graphics processors called processing elements (PEs) in OpenCL, or CUDA cores (or Compute Cores) in NVIDIA.

- PEs are not equivalent to a CPU processor; they are simpler designs, needing to perform graphics operations. But the operations needed for graphics include nearly all the arithmetic operations that a programmer uses on a regular processor.



SIMT/SIMD and PE

- Within each PE, it might be possible to perform an operation on more than one data item.
- Depending on the details of the GPU microprocessor architecture and the GPU vendor, these are referred to as SIMT, SIMD, or vector operations.
  A similar type of functionality can be provided by ganging PEs together (OpenCL subgroups).

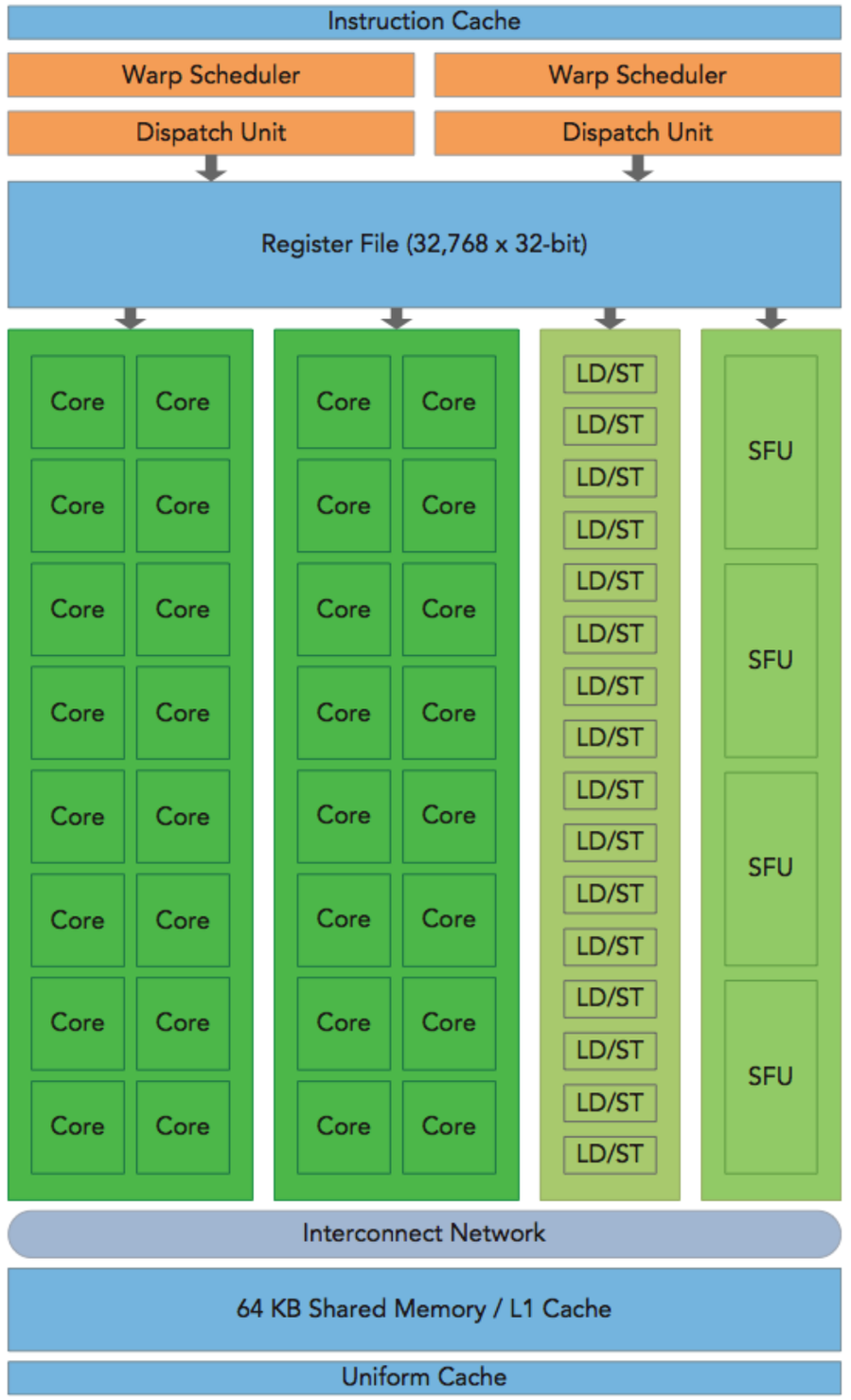
GPU Architecture Overview

- The GPU architecture is built around a scalable array of Streaming Multiprocessors (SM).

- Each SM in a GPU is designed to support concurrent execution of hundreds of threads, and there are multiple SMs per GPU

- NVIDIA GPUs execute threads in groups of 32 called warps. All threads in a warp execute the same instruction at the same time.

- GPU H/Ws are differentiated based on their "compute capabilities". The higher the better.
  Maxwell architecture (e.g. GTX980) have 5.2.
  Pascal architecture (e.g. GTX1080) GPUs has 6.0-6.2.
  Ampere architecture has 8.0, the latest Hopper has 9.0.



LD/ST: load/store data from cache and DRAM
SFU: Execute transcendental instructions such as sin, cosine, reciprocal, and square root.

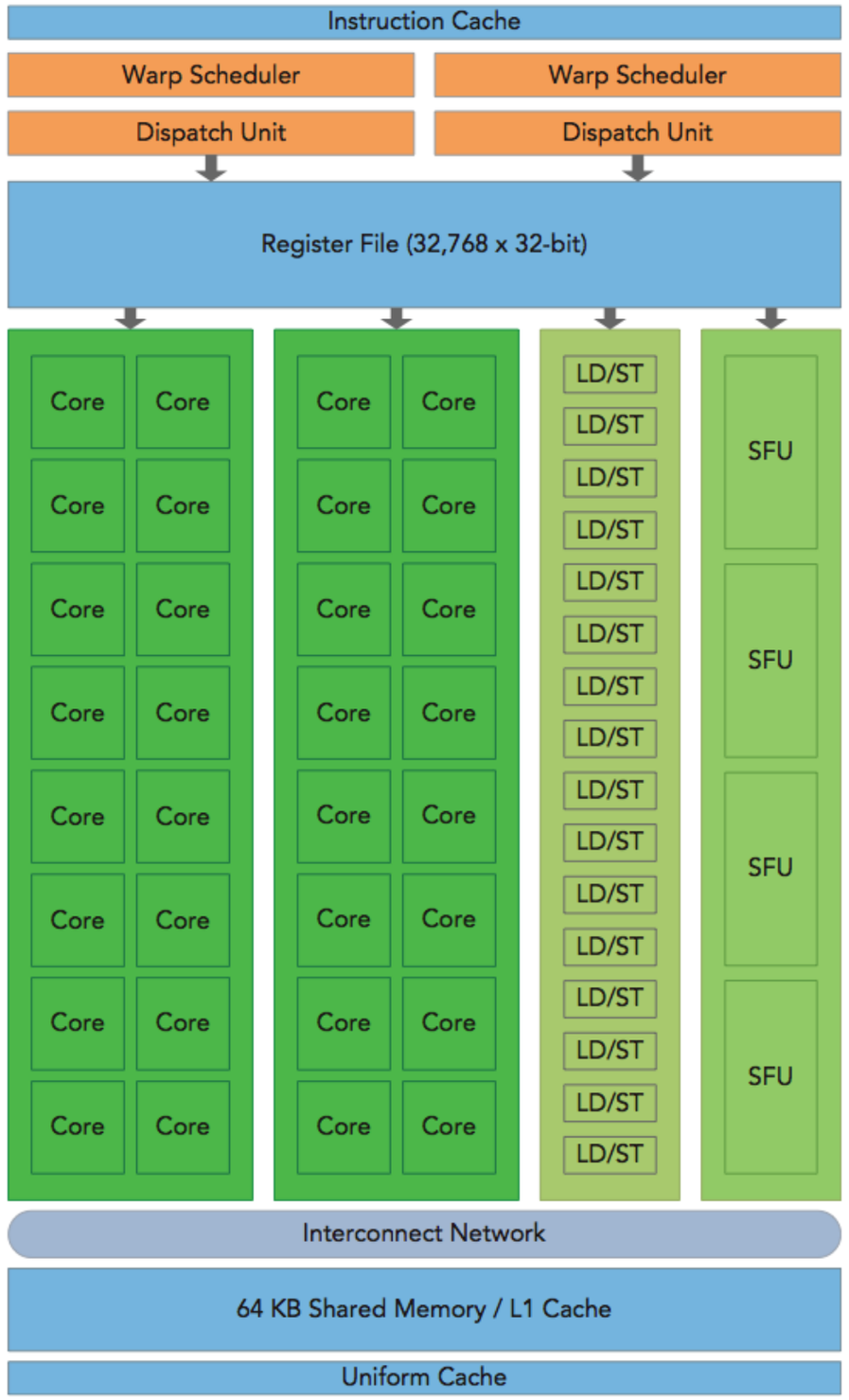
Overview

- The GPU architecture is built around a scalable array of Streaming Multiprocessors (SM).

- Each SM in a GPU is designed to support concurrent execution of hundreds of threads, and there are multiple SMs per GPU

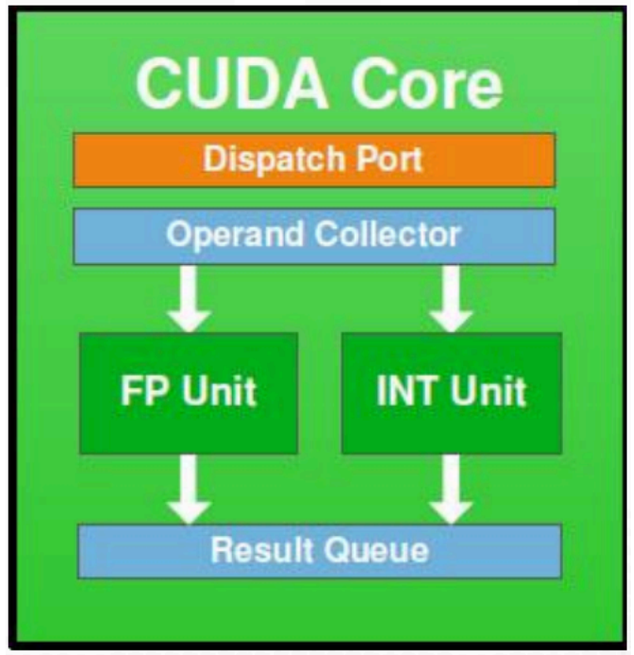
- NVIDIA GPUs execute threads in groups of 32 called warps. All threads in a warp execute the same instruction at the same time.

- GPU H/Ws are differentiated based on their “compute capabilities”. The higher the better. Maxwell architecture (e.g. GTX980) have 5.2. Pascal architecture (e.g. GTX1080) GPUs has 6.0-6.2.
  Ampere architecture has 8.0, the latest Hopper has 9.0.



CUDA core

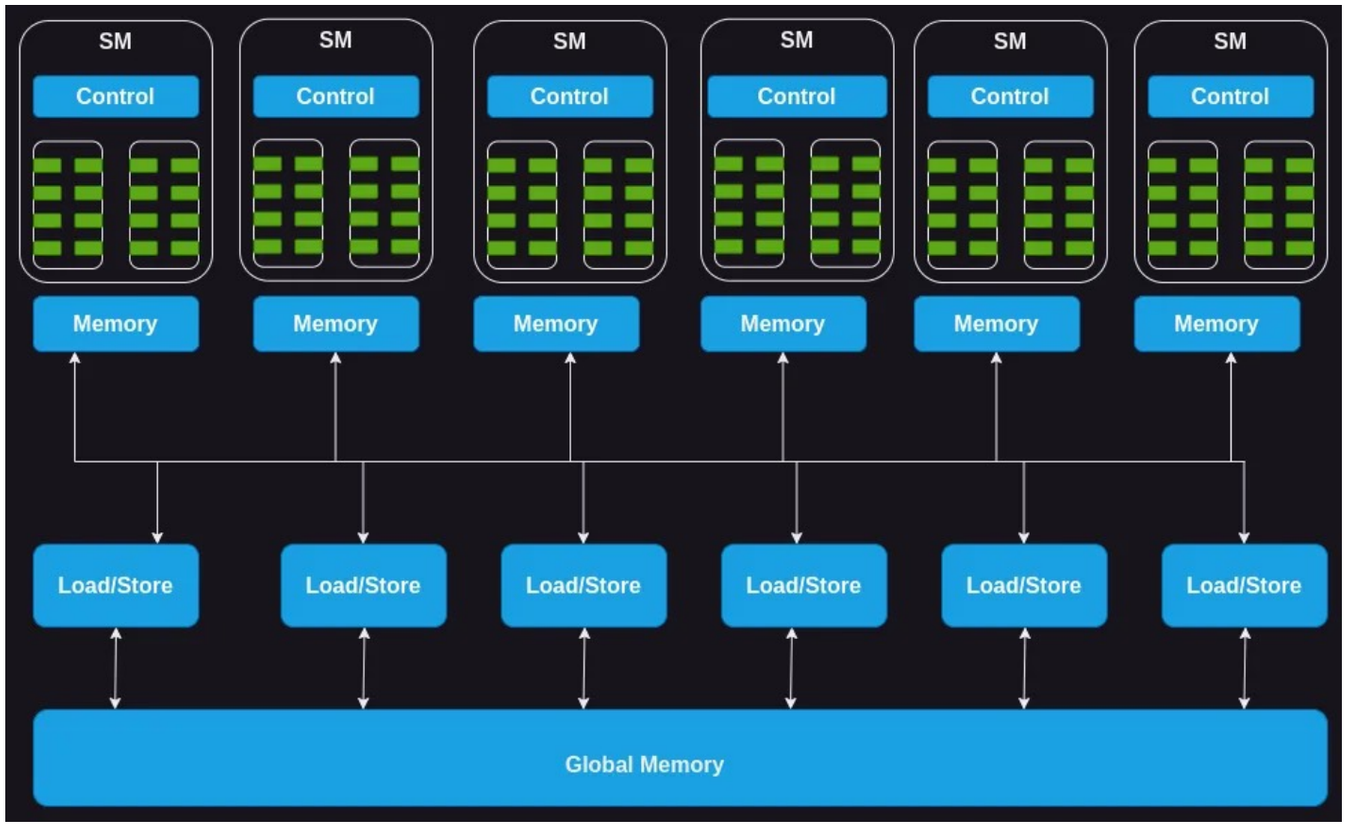
- It’s a vector processing unit
- Works on a single operation
- It’s the building block of SM
- As the process reduces them (e.g. 28nm) they increase in number per SM



Overview

- A GPU consists of an array of streaming multiprocessors (SM). Each of these SMs in turn consists of several streaming processors or cores. E.g., the Nvidia H100 GPU has 132 SMs with 64 cores per SM (8448 cores total).

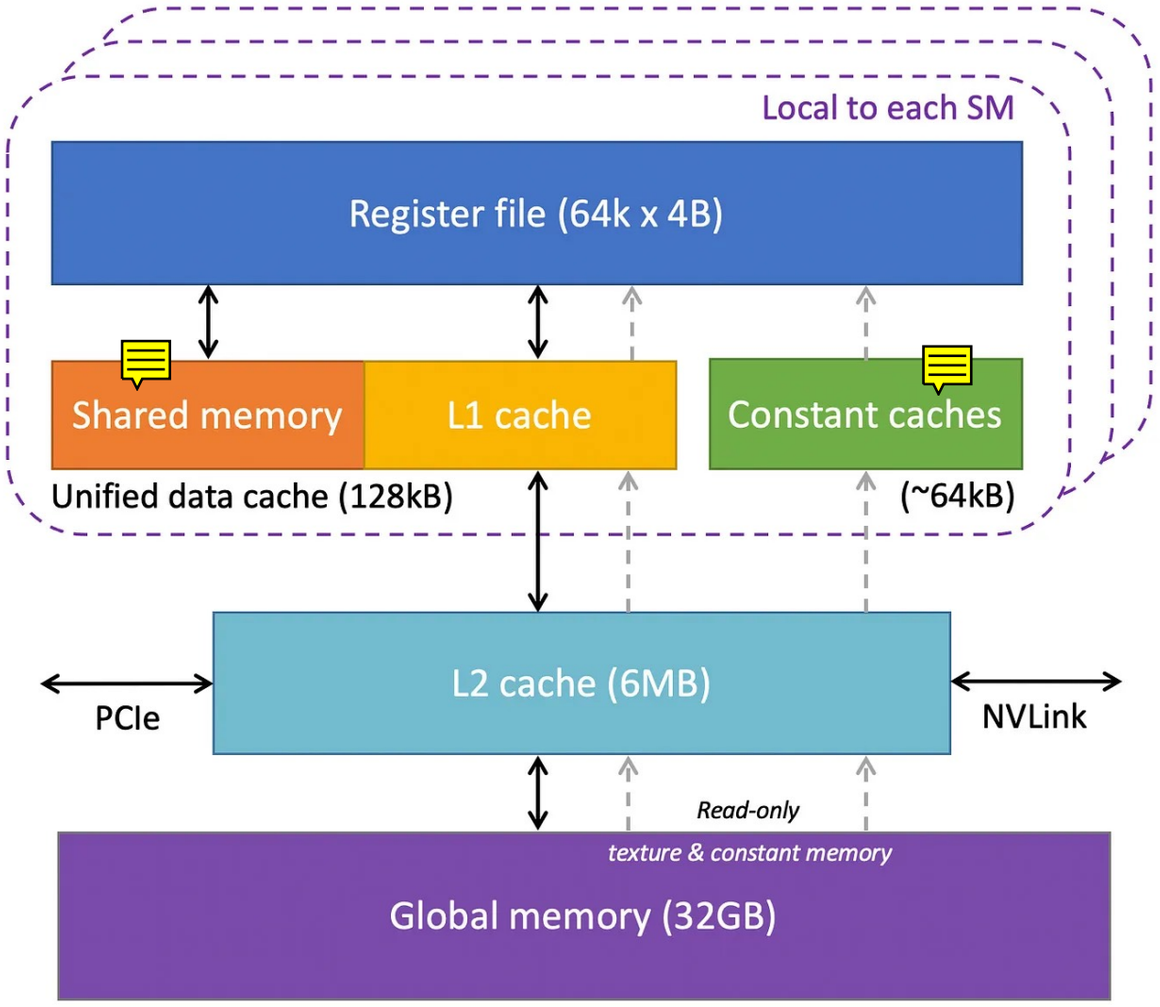
- Each SM has a limited amount of on-chip memory, often referred to as shared memory or a scratchpad, which is shared among all the cores. Likewise, the control unit resources on the SM are shared by all the cores. Additionally, each SM is equipped with hardware-based thread schedulers for executing threads.



Memory overview

- GPUs have several layers of different kinds of memories, with each having their specific use case.

- Some memories are programmable (e.g. shared memory, constant cache)

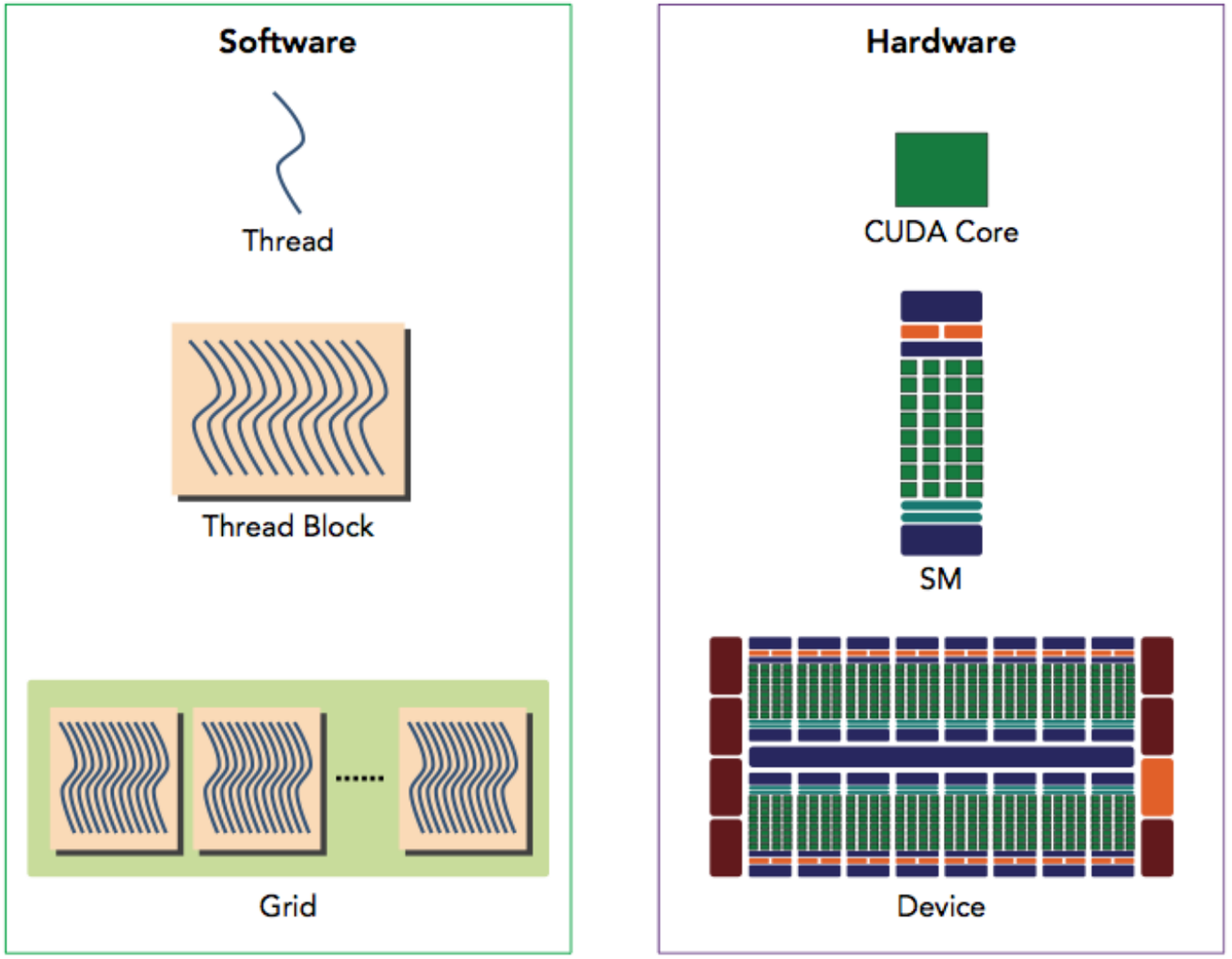


CPU vs GPU

- CPUs have several cores, each one with several ALUs (for SIMD, e.g. SSE for 128bit instructions, AVX for 256bit instructions, etc.)
  - Parallelization at compile time: the compiler generates vectorized instructions
- GPUs have several Streaming Multiprocessors, each one with hundreds of ALUs (i.e. CUDA cores, for SIMT)
  - The binary is scalar, executed in parallel on the CUDA cores. It's the hardware that is responsible fr parallelism

Execution

- A thread block (a “warp”) is scheduled on only one SM. Once a thread block is scheduled on an SM, it remains there until execution completes. An SM can hold more than one thread block at the same time.



Maxwell SMM

- Four 32-core processing blocks each with a dedicated warp scheduler that can dispatch 2 instructions per clock
- Larger shared memory (dedicated to SM)
- Larger L2 cache (shared by SMs)

Pascal SM

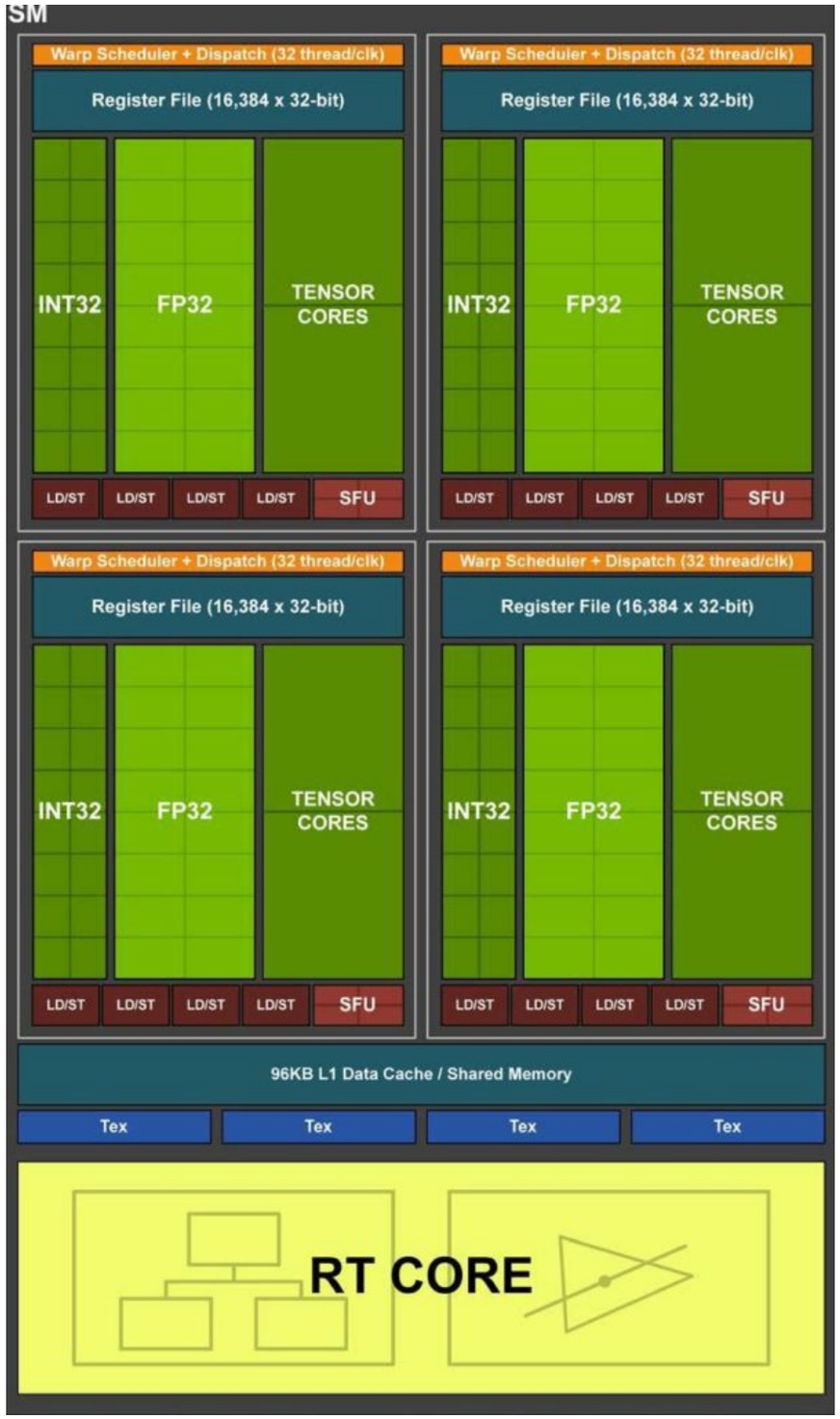
- More SMs per GPU
- FP16 computation (2× faster than FP32)
- Less cores but same # registers: more registers per core
- Fast HBM2 memory interface
- Fast NVLink bus
- Unified memory: programs can access both CPU and GPU RAM

Volta SM

- New tensor cores
- Unified L1 / shared memory
- Independent FP32 and INT32 cores
- More SMs per GPU
- Larger L2 cache
- New L0 instruction cache (accessed directly from functional units)

Turing SM

- New L0 instruction cache, ~2x L1 capacity, 2x L2 capacity
- More SMs per GPU
- Independent FP32 and INT32 cores (like Volta)
- New mixed-precision Turing Tensor cores
- New faster GDDR6 memory, bandwidth up to 600Gb/sec
- Ray Tracing core



Ampere SM

- 3rd generation Tensor Cores with TF32 precision: 2x AI throughput w/ less cores (4 vs. 8)
- Third-generation NVIDIA NVLink high-speed interconnect
- Larger and faster L1 cache (1.5x = 192KB) + shared memory unit; 40 MB Level 2 (L2) cache - ~7x larger than V100
- Async copy instruction to loads data directly from global memory into SM shared memory
- More SMs, more CUDA cores

Hopper SM

- 4th generation tensor cores (double the performance of Ampere tensor cores at the same clock frequency)
- Tensor Memory Accelerators (TMA) sit between the global memory and shared memory in the CUDA programming model hierarchy. They accelerate memory transfers by asynchronously transferring data to shared memory while the CUDA threads do some other work.
- New instructions or dynamic programming (e.g. edit distances)
- A new extension to the CDA programming model (Thread Block Cluster)
- A new Transformer Engine: hardware to a accelerate NN based on transformer architectures (for LLMs)

Blackwell SM

- 208B transistors (2.5x Hopper)
- Decompression engine (LZ4, Deflate, Snappy) to accelerate data pipelines
- Faster and wider NVLink (5th gen.) to provide 1.8 TB/sec total bandwidth. Faster (HBM3e) global memory.
- 2nd generation Transformer Engine: hardware to a accelerate NN based on transformer architectures (for LLMs), use also FP4 / FP6 support

UNIVERSITÀ DEGLI STUDI FIRENZE

NVIDIA GPUs

Blackwell Architecture (compute capabilities 10.x)		GeForce RTX 50 Series		Tesla B Series
Hopper / Ada Architecture (compute capabilities 9.x / 8.9)		GeForce RTX 40 Series	Quadro RTXSeries	Tesla H / L Series
Ampere Architecture (compute capabilities 8.x)	Drive AGX Jetson Orin	GeForce RTX 30 Series	Quadro RTXSeries	Tesla A Series
Turing Architecture (compute capabilities 7.x)		GeForce RTX 20 Series	Quadro RTXSeries	Tesla T Series
Volta Architecture (compute capabilities 7.x)	Jetson Xavier Jetson TX2	Titan V		Tesla V Series
Pascal Architecture (compute capabilities 6.x)	Jetson TX1	GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (compute capabilities 5.x)	Tegra X1 Jetson Nano	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center

Example: GPU specs

GPU	NVIDIA V100 (Volta)	NVIDIA A100 (Ampere)
Compute units (CUs)	80	108
FP32 cores/CU	64	64
FP64 cores/CU	32	32
GPU clock nominal/boost	1290/1530 MHz	1410 MHz
Subgroup or warp size	32	32
Memory clock	876 MHz	1215 MHz
Memory type	HBM2 (32 GB)	HBM2(40 GB)
Memory data width	4096 bits	5120 bits
Memory bus type	NVLink or PCIe 3.0x16	NVLink or PCIe Gen 4
Design Power	300 watts	400 watts

CU = NVIDIA CUDA SMM

PE (CUDA core) per CU.
We can roughly double your performance by reducing your precision requirements from double precision to single.
Same is true for AMD GPUs.

Peak theoretical flops

- Peak Theoretical Flops (GFlops/s)
  = Flops/cycle × Clock rate MHZ × Compute Units × Processing units
  The flops per cycle accounts for the fused-multiply add (FMA), which does two operations in one cycle.

- Theoretical Peak Flops for NVIDIA V100:
  - 2 × 1530 × 80 × 64 /10^6 = 15.6 TFlops (single precision)
  - 2 × 1530 × 80 × 32 /10^6 = 7.8 TFlops (double precision)
- Theoretical Peak Flops for NVIDIA Ampere:
  - 2 × 1410 × 108 × 64 /10^6 = 19.5 TFlops (single precision)
  - 2 × 1410 × 108 x× 32 /10^6 = 9.7 TFlops (double precision)

Back to bandwidth

- A NVIDIA GTX1080Ti has 28 SM and 3584 CUDA cores (128 cores per SM). It's clocked at 1.48GHz and memory bandwidth is 484GB/s. This means:
- ~327 bytes/cycle for thew whole GPU.
- 11.7 bytes/cycle per SM (~4× of Intel i7-7700K)
- 0.09 bytes/cycle per CUDA core, i.e. only one byte every 11 instructions !

Back to bandwidth

- A NVIDIA GTX1080Ti has 28 SM and 3584 CUDA cores (128 cores per SM). It's clocked at 1.48GHz and memory bandwidth is 484GB/s. This means:
- ~327 bytes/cycle for thew whole GPU.
- 11.7 bytes/cycle per SM (~4× of Intel i7-7700K)
- 0.09 bytes/cycle per CUDA core, i.e. only one byte every 11 instructions !

Remind that a 40-years old MOS 6502 got 4 bytes/instruction !

Back to bandwidth

Absolute memory bandwidths in consumer devices have gone up by several orders of magnitude from the ~1MB/s of early 80s home computers, but available compute resources have grown much faster still.
The only way to stop bumping into bandwidth limits all the time is to make sure your workloads have reasonable locality of reference so that the caches can do their job.

- A
- co
- an
- ~3
- 11.7 bytes/cycle per SM (~4× of Intel i7-7700K)
- 0.09 bytes/cycle per CUDA core, i.e. only one byte every 11 instructions !

L2 caches of NVIDIA GPUs are going up from 1536KB in Kepler (K40), to 4096KB in Pascal (GP100) and 6144KB in Volta (GV100)

Remind that a 40-years old MOS 6502 got 4 bytes/instruction !

Back to bandwidth

Absolute memory bandwidths in consumer devices have gone up by several orders of magnitude from the ~1MB/s of early 80s home computers, but available compute resources have grown much faster still.
The only way to stop bumping into bandwidth limits all the time is to make sure your workloads have reasonable locality of reference so that the caches can do their job.

- A
- co
- an
- ~3

L2 caches of NVIDIA GPUs are going up from 1536KB in Kepler (K40), to 4096KB in Pascal (GP100) and 6144KB in Volta (GV100)

Pascal and Volta GPUs use High Bandwidth Memory 2 (HBM2) high-perf. RAM interface to achieve higher bandwidth (900 GB/s).
HBM2e on A100 goes up to 1.9 TB/s.
Hopper uses HBM3 (~3 TB/s) and Blackwell uses HBM3e going up to 8TB/s!

- 0.09 bytes/cycle per CUDA core, i.e. only one byte every 11 instructions !

Remind that a 40-years old MOS 6502 got 4 bytes/instruction !

Back to bandwidth

Absolute memory bandwidths in consumer devices have gone up by several orders of magnitude from the ~1MB/s of early 80s home computers, but available compute resources have grown much faster still.
The only way to stop bumping into bandwidth limits all the time is to make sure your workloads have reasonable locality of reference so that the caches can do their job.

- A
- co
- an
- ~3

L2 caches of NVIDIA GPUs are going up from 1536KB in Kepler (K40), to 4096KB in Pascal (GP100) and 6144KB in Volta (GV100)

Pascal and Volta GPUs use High Bandwidth Memory 2 (HBM2) high-perf. RAM interface to achieve higher bandwidth (900 GB/s).
HBM2e on A100 goes up to 1.9 TB/s.
Hopper uses HBM3 (~3 TB/s) and Blackwell uses HBM3e going up to 8TB/s!

NVLink bus connects CPU and GPU (or multiple GPUs) at 80-200 GB/s - it's an alternative to PCI Express

every 11 instructions !

Remind that a 40-years old MOS 6502 got 4 bytes/instruction !

Theoretical peak memory bandwidth

- Theoretical Bandwidth = Memory Clock Rate (GHz) × Memory bus (bits) × (1 byte/8 bits) × transaction multiplier

or

- Theoretical Bandwidth = Memory Transaction Rate (Gbps) × Memory bus (bits) × (1 byte/8 bits)

- GPUs use specialized global memory (RAM), which provides higher bandwidth, whereas current CPUs use DDR4 memory and are just now moving to DDR5.
  GPUs use a special version called GDDR5 that gives higher performance.
  The latest GPUs are now moving to High-Bandwidth Memory (HBMx) that provides even higher bandwidth (and reduces power consumption).

Memory specs

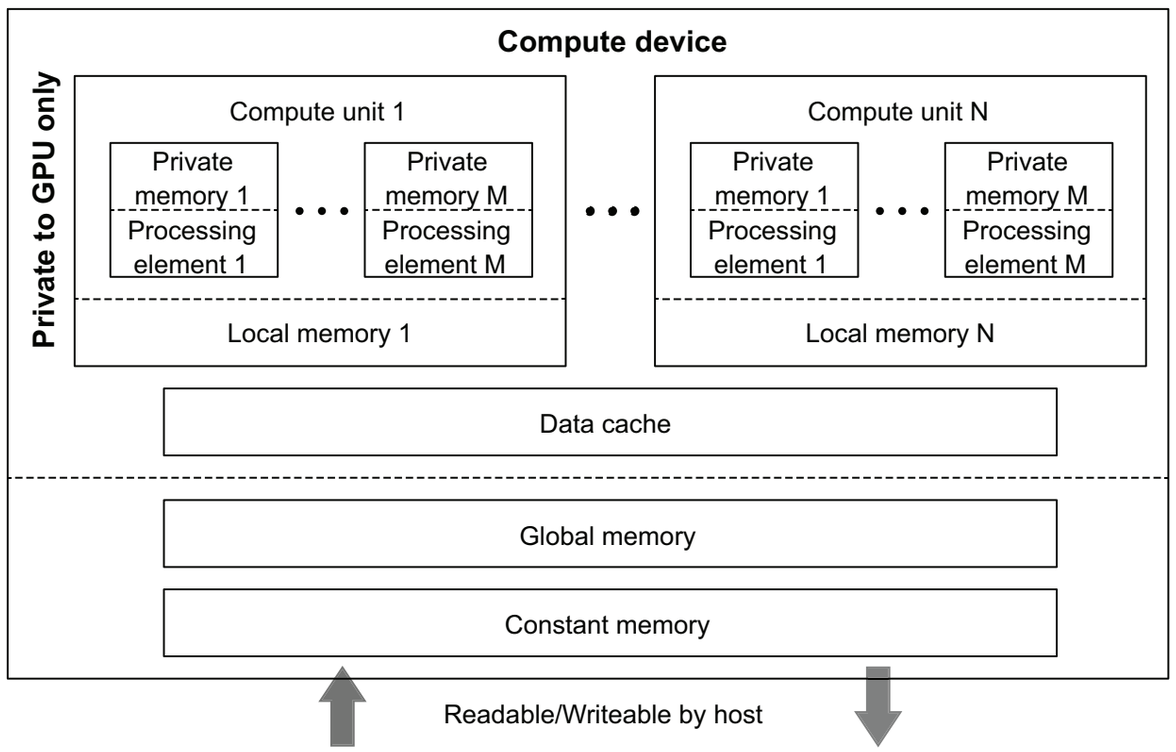
Graphics Memory Type	Memory Clock (MHz)	Memory Transactions (GT/s)	Memory Bus Width (bits)	Transaction Multiplier	Theoretical Bandwidth (GB/s)
GDDR3	1000	2.0	256	2	64
GDDR4	1126	2.2	256	2	70
GDDR5	2000	8.0	256	4	256
GDDR5X	1375	11.0	384	8	528
GDDR6	2000	16.0	384	8	768
HBM1	500	1000.0	4096	2	512
HBM2	1000	2000.0	4096	2	1000

- NVIDIA V100 operates at 876 MHz memory clock and uses HBM2 memory:

- Theoretical Bandwidth = 0.876 × 4096 × 1/8 × 2 = 897 GB/s

GPU memories

- A typical GPU has different types of memory. Using the right memory space can make a big impact on performance.
  - Private memory (register memory) — Immediately accessible by a single PE and only by that PE.
  - Local memory — Accessible to a single CU and all of the PEs on that CU. Local memory can be split between a scratchpad that can be used as a programmable cache and, by some vendors, a traditional cache on GPUs. Local memory is around 64-96 KB in size.
  - Constant memory — Read-only memory accessible and shared across all of the CUs.
  - Global memory—Memory that's located on the GPU and accessible by all of the CUs



CPU to GPU data transfer

- The PCI bus is needed to transfer data from the CPU to the GPU and back. The cost of the data transfer can be a significant limitation on the performance of operations moved to the GPU.

- The current version of the PCI bus is called PCI Express (PCIe). It has been revised several times in generations from 1.0 to 6.0
  - Gen6 final draft (v0.9) available since 6th Oct. 2021

- Communication occurs over multiple PCIe lanes.

- Uses an encoding scheme to ensure data integrity, this slows down effective transfer rate.

Theoretical PCI bus bandwidth

- Theoretical Bandwidth (GB/s) = Lanes × TransferRate (GT/s) × OverheadFactor(Gb/GT) × (1 byte/8 bits)

- OverheadFactor accounts for the encoding scheme overhead

- Get PCI lanes using lspci command on Linux, e.g.:
  `lspci -vmm | grep "PCI bridge" -A2`
  or dmidecode, e.g.:
  `dmidecode | grep "PCI"`

- Use `lspci` to get the transfer rate (look for LnkCap - link capacity)

PCIe specs

PCIe Generation	Maximum Transfer Rate (bi-directional)	Encoding Overhead	Overhead factor (100%-encoding overhead)	Theoretical Bandwidth 16 lanes - GB/s
Gen1	2.5 GT/s	20%	80%	4
Gen2	5.0 GT/s	20%	80%	8
Gen3	8.0 GT/s	1.54%	98.46%	15.75
Gen4	16.0 GT/s	1.54%	98.46%	31.5
Gen5 (2019)	32.0 GT/s	1.54%	98.46%	63
Gen6 (2021)	64.0 GT/s	1.54%	98.46%	126

- Note Gen1 and 2 transmitted 2 additional bytes every 8 payload bytes. Starting from Gen3 they transmit 2 additional bytes every 128 payload bytes.

Theoretical PCI bus bandwidth example

- Suppose we have a Gen3 PCIe system with 16 lanes.
  Gen3 systems have a maximum transfer rate of 8.0 GT/s and an overhead factor of 0.985. With 16 lanes, the theoretical bandwidth is 15.75 GB/s.

- Theoretical Bandwidth (GB/s)
  = 16 lanes × 8.0 GT/s × 0.985 (Gb/GT) × (1 byte/8 bits) = 15.75 GB/s

Moving data between GPUs

- The PCI bus is a major limitation for compute nodes with multi-GPUs. It impacts heavy workloads such as in machine learning on smaller clusters.

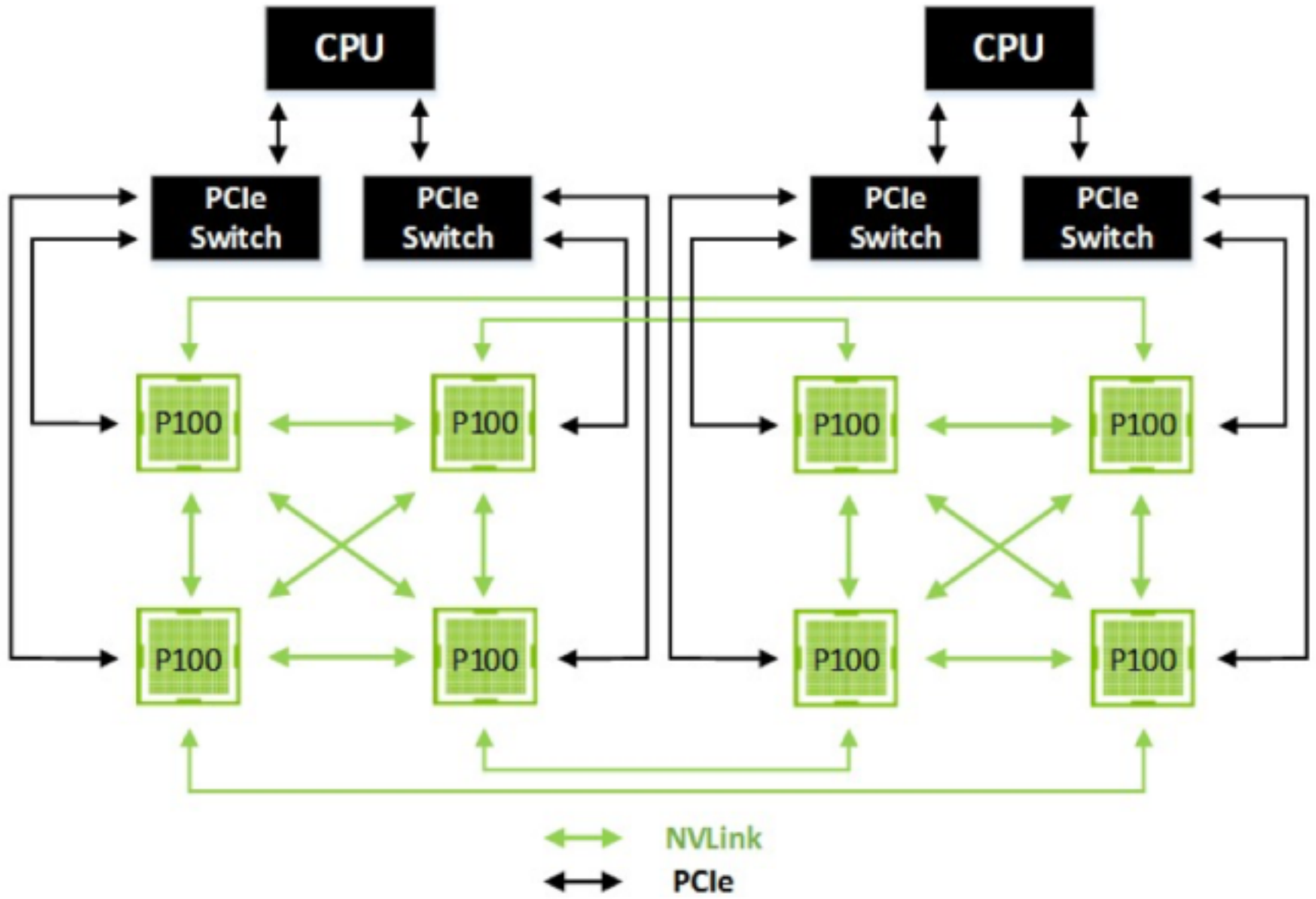
- NVIDIA introduced NVLink to replace previous GPU-to-GPU and GPU-to-CPU connections since Pascal (Tesla P100). Used in NVIDIA DGX-1 (up to 8 P100).

- NVLink enables each GPU to directly execute read, write, and atomic memory operations out of the memory of the other GPUs.

- With NVLink 2.0 (Volta), the data transfer rates can reach 300 GB/sec. NVLink 3.0 (Ampere) goes to 600 GB/sec.

- Similarly AMD uses Infinity Fabric to speed up data transfers.

DGX-1 connections schema



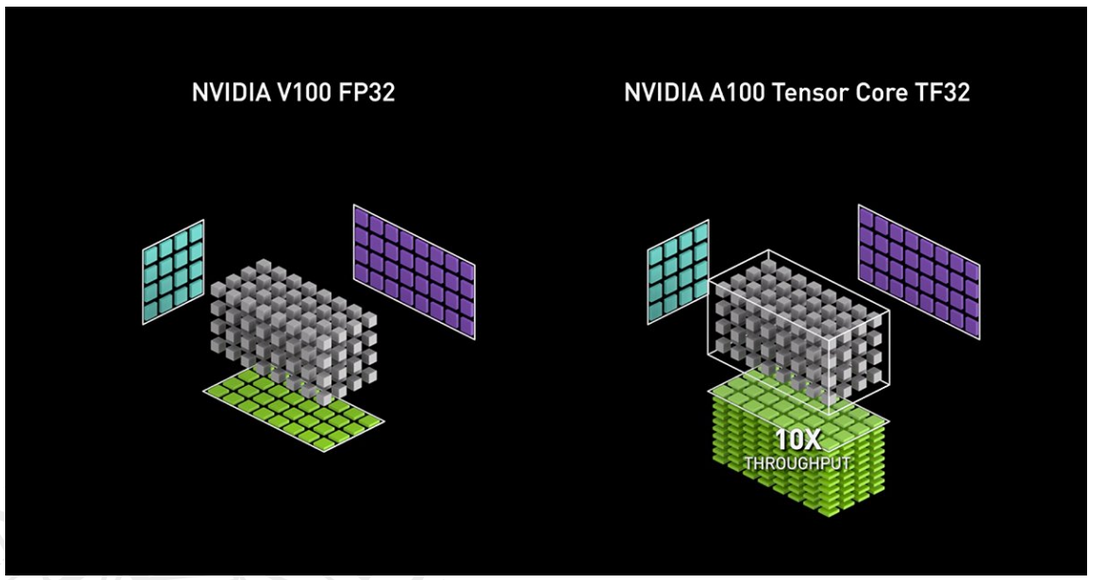
- The mesh is a Non-Uniform Memory Access (NUMA) style design: not every GPU is linked to every other GPU. When two GPUs aren't linked, the next GPU is no more than 1 hop away. The GPUs connect back to their x86 CPU host over standard PCI Express.

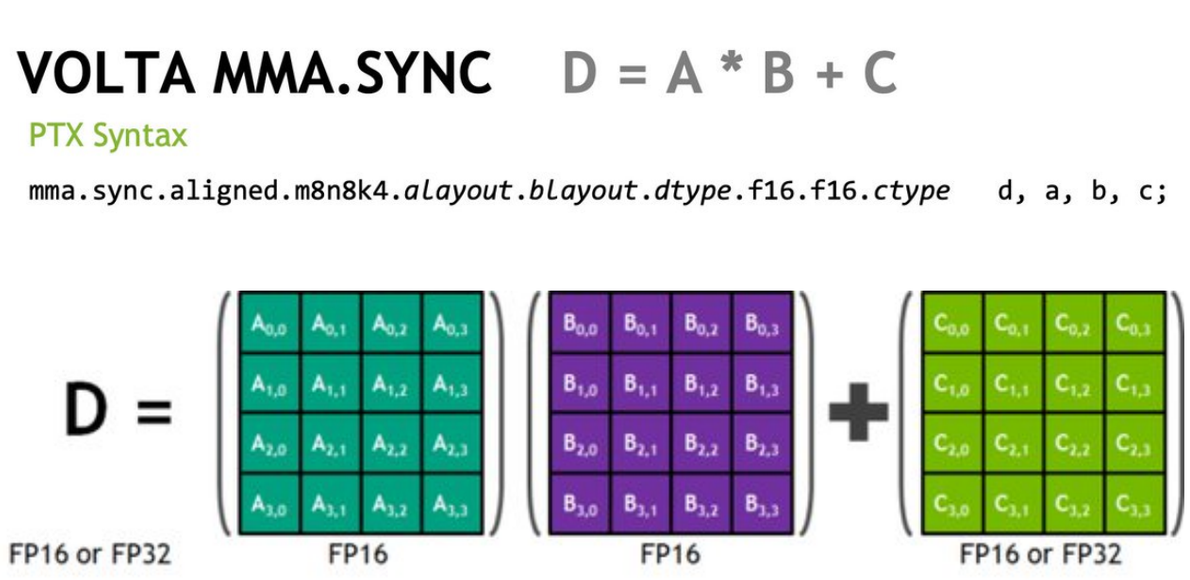
UNIVERSITÀ
DEGLI STUDI
FIRENZE

Tensor cores

What are tensor cores

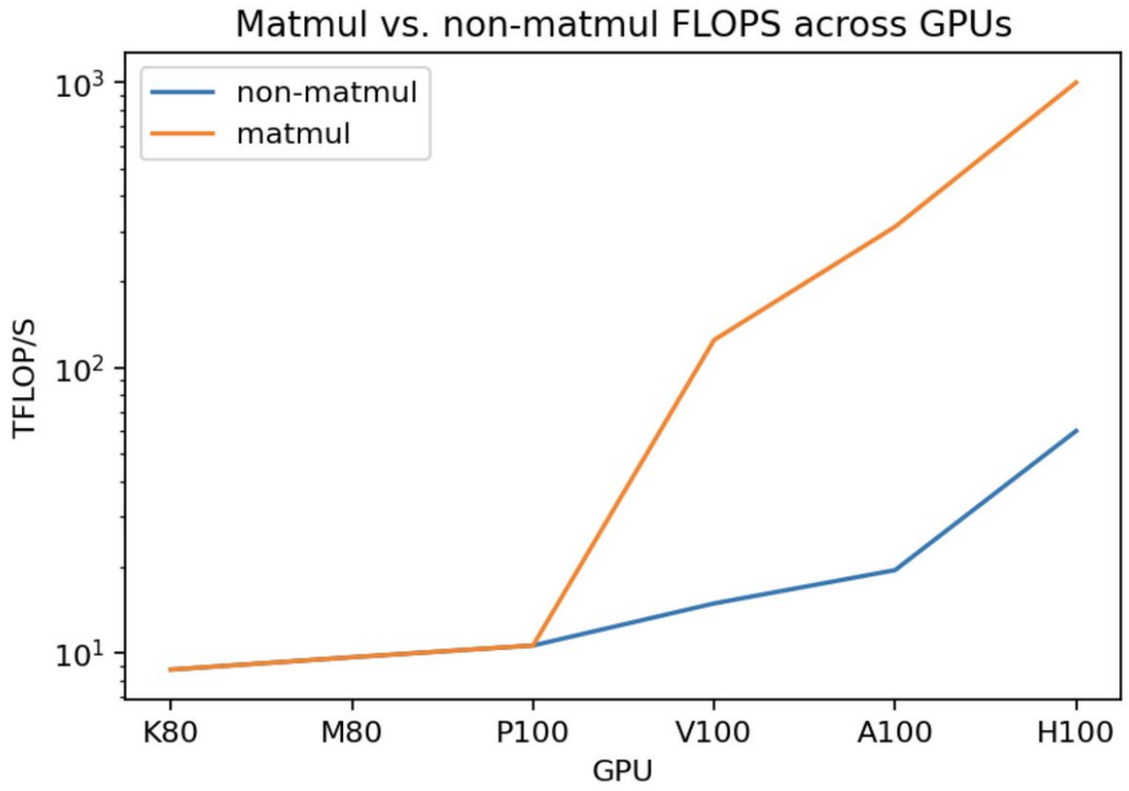
- These cores are specifically designed for matrix multiplication
  - Work with mixed precision
  - Use fused multiply-add operations
- They add a new instruction that allows to use them
  - Work on 4×4 matrices (that would need 64 multiplications + 48 additions) in 1 cycle





Why is it useful ?

- The D = A x B + C operation is used heavily in deep learning (output = weight * input + bias)
- Such specialized hardware leads to TFLOPS improvements



Their evolution

- Newer versions allow to use different types of precision
  - e.g. INT4, INT8, BF16
- Since Ampere they support 256 GEMM instead of 64
- Since Ampere they support sparse matrices (i.e. matrices with many 0s)

UNIVERSITÀ
DEGLI STUDI
FIRENZE

GPU programming model

Motivations and goals

- Let's briefly review an abstract GPU programming model to understand the essential aspects

  - We need to change the approach used on CPUs

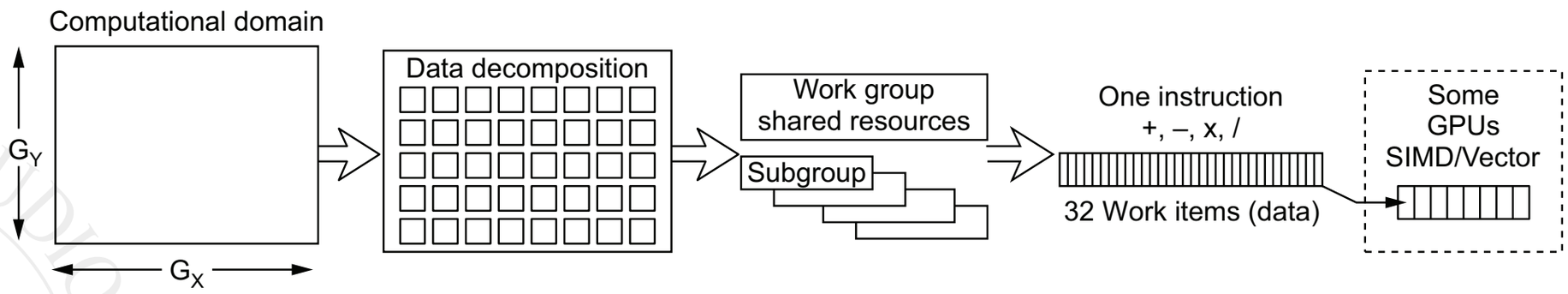
    - It's simpler than the real model

    - It adapts to different GPUs and real-world languages

- Learn how to express parallel loops, move data from host CPU to device GPU and coordinate threads

Massive parallelism

- GPUs are designed to handle graphics operations, that have massive parallelism
  - For a high frame rate and high-quality graphics, there are lots of pixels, triangles, and polygons to process and display.
  - The operations on the data are generally identical, so GPUs apply a single instruction to multiple data items
- The common programming abstractions across various vendors and GPU models are:
  - Data decomposition
  - Chunk-sized work for processing with some shared, local memory
  - Operating on multiple data items with a single instruction
  - Vectorization (on some GPUs)



Coordination

- Graphics workloads do not require much coordination within the operations, so GPUs do not provide much beyond basic functionalities like barriers.

- If we need to implement algorithms, such as reductions, that require coordination, we will have to develop (more or less complicated) schemes to handle these situations.

Terminology

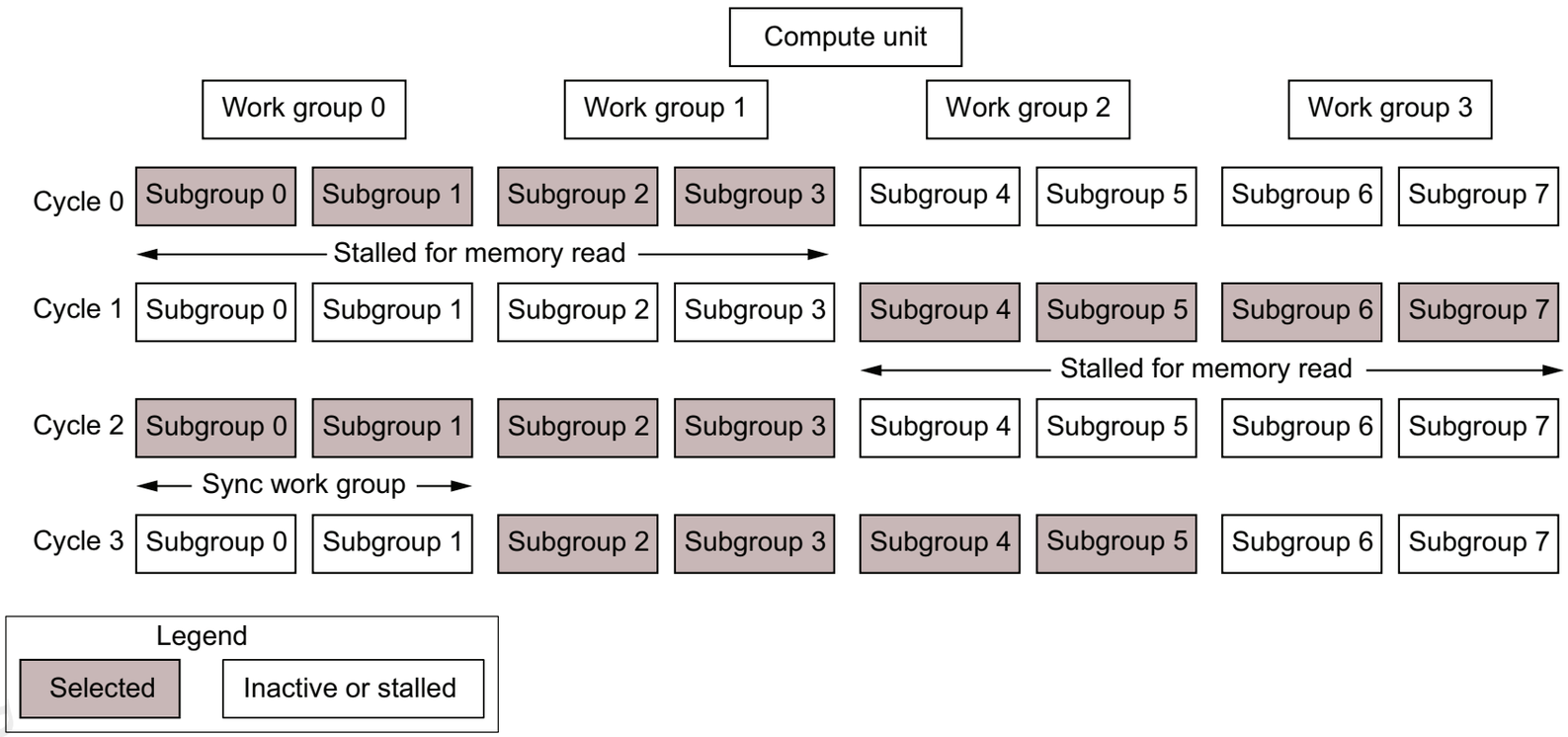
CPU	OpenCL	CUDA
Standard loop bounds or index sets with loop blocking	NDRange (*N*-dimensional range)	grid
loop block	Work Group	block or thread block
SIMD length	Subgroup or Wavefront	warp
thread	Work Item	thread

Data decomposition

- It is the heart of how GPUs obtain performance. GPUs break up the problem into many smaller blocks of data. Then they break it up again, and again.

- Graphic operations are completely independent from each other. For this reason, the top-level data decomposition for computational work on a GPU also generates independent and asynchronous work.

- With lots of work to do, GPUs hide latency (stalls for memory loads) by switching to another work group that is ready to compute.



Subgroup scheduler

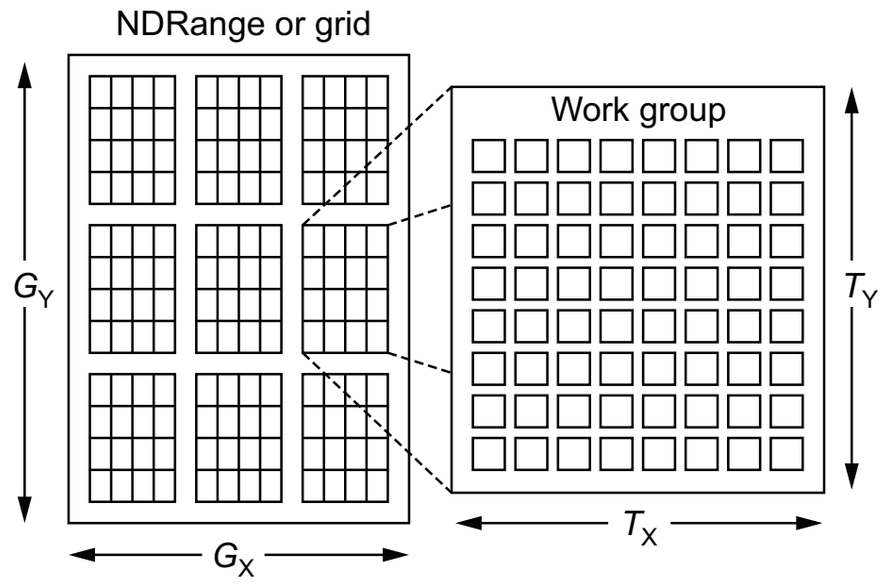
- The execution switch, also called a context switch, is hiding latency with computation rather than with a deep cache hierarchy.

- Use chip silicon for ALUs rather than cache levels

	NVIDIA Volta and Ampere	AMD MI50
Active number of subgroups per compute unit	64	40
Active number of work groups per compute unit	32	40
Selected subgroups for execution per compute unit	4	4
Subgroup (warp or wavefront) size	32	64

Decomposition example

- Example of 2D computational domain is split into smaller 2D blocks of data. In OpenCL, this is called an *NDRange* (N-dimensional range, in CUDA it is a *grid*).
  The NDRange in this case is a 3×3 set of tiles of size 8×8. The data decomposition process breaks up the global computational domain, G_y by G_x, into smaller blocks or tiles of size T_y by T_x.



Decomposition examples

	1D	Small 2D	Large 2D	3D
Global size	1,048,576	1024 × 1024	1024 × 1024	128 × 128 × 128
$T_z \times T_y \times T_x$	128	8 × 8	8 × 16	4 × 4 × 8
Tile size	128	64	128	128
$NT_z \times NT_y \times NT_x$	8,192	128 × 128	128 × 64	32 × 32 × 16
NT (number of work groups)	8,192	16,384	8,192	16,384

Decomposition tricks

- In the original graphics use case, there is not much of a need to go beyond 2 or 3 dimensions and the corresponding number of parallelization levels. If your algorithm has more dimensions or levels, you must combine some computational loops to fully parallelize your problem.

- The fastest changing tile dimension, T_x, should be a multiple of the cache line length, memory bus width, or subgroup (wavefront or warp) size for best performance. The number of tiles, NT, overall and in each dimension, results in a lot of work groups (tiles, CUDA blocks) to distribute across the GPU compute engines and processing elements.

- For algorithms that need neighbor information, the optimum tile size for memory accesses needs to be balanced against getting the minimum surface area for the tile. Neighbor data must be loaded more than once for adjacent tiles - can we share it ?

Chunk-sized work

- The work group spreads out the work across the threads on a compute unit. Each GPU model has a maximum work group size specified for the hardware, usually between 256 and 1,024. It can be much smaller (and typically is so), so that there are more memory resources per work item or thread.

- The work group is subdivided into subgroups/wavefront (OpenCL/AMD) or warps (CUDA). A subgroup is the set of threads that execute in lockstep. For NVIDIA, the warp size is 32 threads, for AMD the size is usually 64 work items.

- Trick: the work group size must be a multiple of the subgroup size.

Workgroups

- Cycle through processing each subgroup
- Have local memory (shared memory) and other resources shared within the group
- Can synchronize within a work group or a subgroup

Work item

- The basic unit of operation is called a work item in OpenCL. This work item can be mapped to a thread or to a processing core, depending on the hardware implementation.

- In CUDA, it is called a thread because that is how it is mapped in NVIDIA GPUs. This term is mixing the programming model with how it is implemented in the hardware (but is clearer to the programmer).

- A work item can invoke another level of parallelism on GPUs with vector hardware units.

SIMD / Vector h/w

- Some GPUs also have vector hardware units and can do SIMD (vector) operations in addition to SIMT operations.

- Vector operations are exposed in the OpenCL language and AMD languages.

- CUDA hardware does not have vector units, therefore the same level of support is not present in CUDA languages.

Code structure

- The code written to run on the GPU (called kernel) is to be applied as a C++ Functor or Lambda expression to the elements of the data chunks obtained by our data decomposition.

- In OpenMP or other CPU-based code we would have a (parallelized) loop and the loop body contains the operation applied to each element of the chunk.

- In GPU kernels we simply have body loop and some code that computes the index to associate each thread to the elements of the data chunks.

Code structure

- From:

```
// stream triad loop
for (int i=0; i<STREAM_ARRAY_SIZE; i++) {
  c[i] = a[i] + scalar*b[i];
}
```

- To:

```
size_t gid = get_global_id(0); // OpenCL
size_t gid = blockIdx.x *blockDim.x + threadIdx.x; // CUDA
if (gid >= STREAM_ARRAY_SIZE) return;
c[gid] = a[gid] + scalar*b[gid]; // loop body
```

Code structure

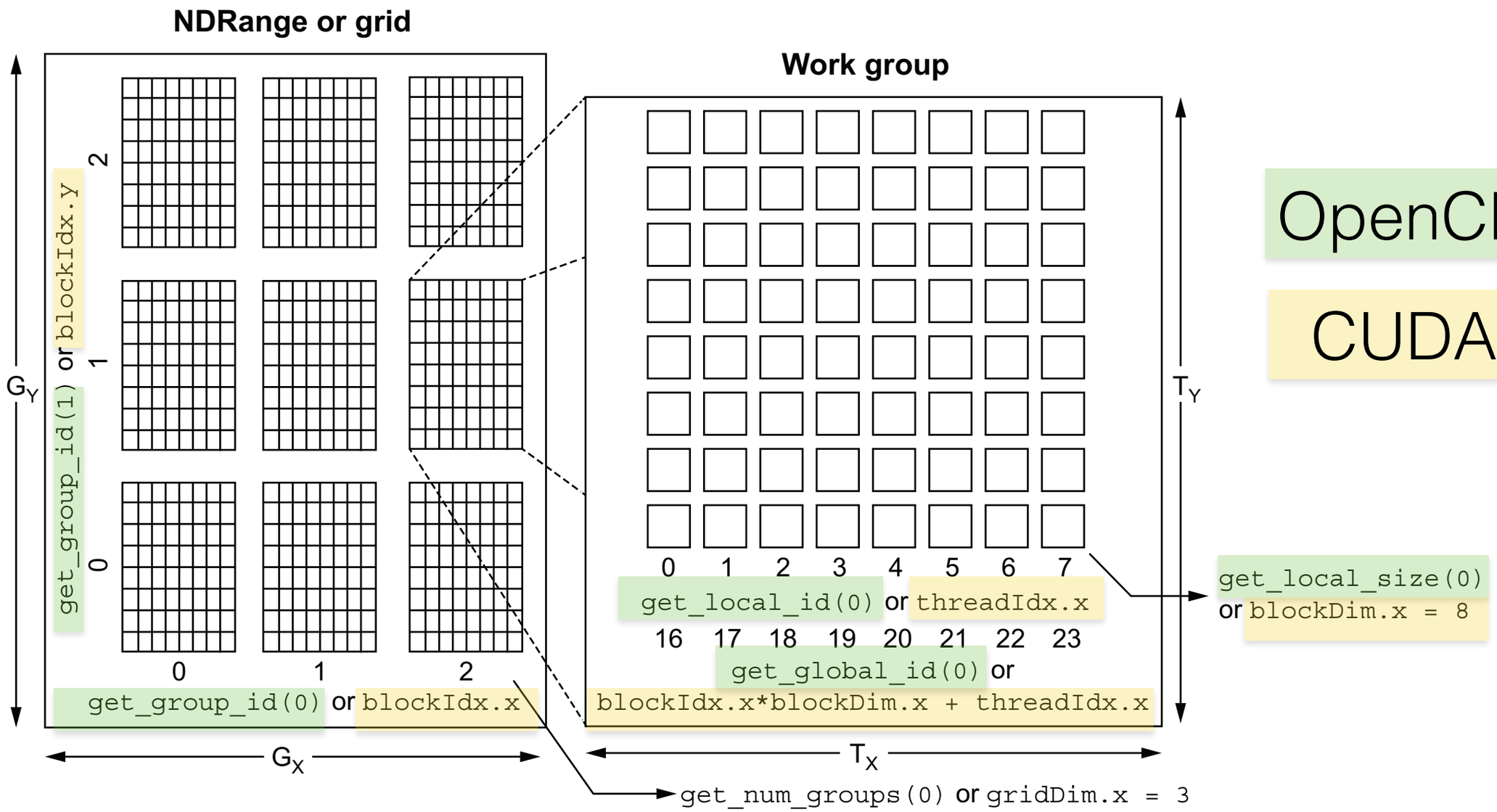
- How to move from loop-based code to GPU style kernels ?

1. Extract the parallel kernel
2. Map from the local data tile to global data
3. Calculate data decomposition on the host into blocks of data
4. Allocate any required memory

Thread indices

- We are dealing with a data decomposition approach, we need to compute an appropriate mapping between local tiles and global data, using available information provided by OpenCL/CUDA.
- OpenCL
  - Dimension — Gets the number of dimensions, either 1D, 2D, or 3D, for this kernel from the kernel invocation
  - Global information — Global index in each dimension, which corresponds to a local work unit, or the global size in each dimension, which is the size of the global computational domain in each dimension
  - Local (tile) information —The local size in each dimension, which corresponds to the tile size in this dimension, or the local index in each dimension, which corresponds to the tile index in this dimension
  - Group information — The number of groups in each dimension, which corresponds to the number of groups in this dimension, or the group index in each dimension, which corresponds to the group index in this dimension
- CUDA
  - Similar information is available in CUDA, but the global index must be calculated from the local thread index plus the block (tile) information:

OpenCL and CUDA indices



OpenCL

CUDA

- The size of the indices for each work group should be identical. This is done by padding the global computational domain out to a multiple of the local work group size.

Using GPU Memory

- Memory allocation for the GPU has to be done on the CPU.
- Try to re-use local (shared) memory for data that is used more than once.
  - But improvement in GPU caches are reducing this need.

Optimize GPU resources usage

- The most important control available to the GPU programmer is the work group size.

- Computational kernels need more compute resources than graphics kernels: this is the so called memory pressure or register pressure. Reducing the work group size gives each work group more resources to work with.

- Another benefit is allowing the scheduler to choose between more work groups for context switching.

Optimize GPU resources usage

Resource limit	NVIDIA compute capability 7.0	AMD Vega 20 (MI50)
Maximum threads per work group	1024	256
Maximum threads per compute unit	2048	
Maximum work groups per compute unit	32	16
Local memory per compute unit	96 KB	64 KB
Register file size per compute unit	64K	256 KB vector
Maximum 32-bit registers per thread	255	

- Occupancy is a measure of how busy the compute units are during the calculation.
  - Occupancy = Number of Active Threads/Maximum Number of Threads Per Compute Unit
- Because the number of threads per subgroup is fixed, an equivalent definition is based on subgroups, also known as wavefronts or warps:
  - Occupancy = Number of Active Subgroups/Maximum Number of Subgroups Per Compute Unit

Optimize GPU resources usage

Resource limit	NVIDIA compute capability 7.0	AMD Vega 20 (MI50)
Maximum threads per work group	1024	256
Maximum threads per compute unit	2048	
Maximum work groups per compute unit	32	16
Local memory per compute unit	96 KB	64 KB

CUDA has a tool to compute occupancy
It is important to have good occupancy but the really important thing is to have enough work groups to switch, to hide latency and memory stall

- Occupancy is a measure of how busy the compute units are during the calculation.
  - Occupancy = Number of Active Threads/Maximum Number of Threads Per Compute Unit
- Because the number of threads per subgroup is fixed, an equivalent definition is based on subgroups, also known as wavefronts or warps:
  - Occupancy = Number of Active Subgroups/Maximum Number of Subgroups Per Compute Unit

Synchronization and cooperation

- The source of the difficulty is that we cannot do cooperative work or comparisons across work groups.

- We have work group barriers and atomics

- No mutexes

- If needed, execute and synchronize successive launches of kernels

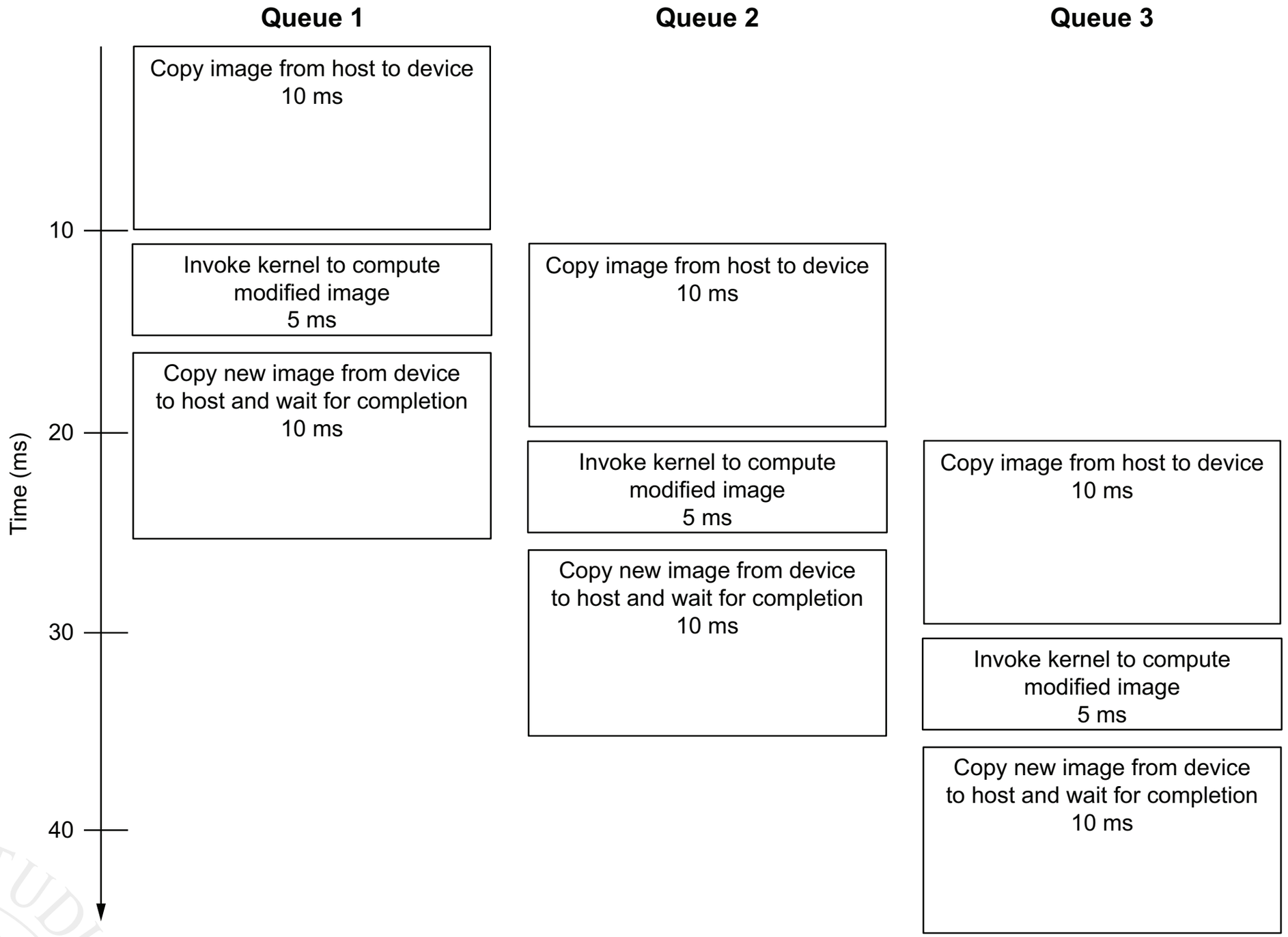
Asynchronous computing

- The basic nature of work on GPUs is asynchronous. Work is queued up on the GPU and, usually, only gets executed when a result or synchronization is requested.

- We can also schedule work in multiple queues (OpenCL) or streams (CUDA) that are independent and asynchronous.

- We can exploit this to speed-up operations that follow the host-to-device copy / compute / device-to-host copy schema, by overlapping communications and computation operations.

Asynchronous computing



Credits

- These slides report material from:
  - Prof. Jan Lemeire (Vrjie Universiteit Brussel)
  - Prof. Dan Negrut (Univ. Wisconsin - Madison)
  - Horace He (Cornell Univ.)
  - NVIDIA GPU Teaching Kit

Books

- Parallel and High Performance Computing, R. Robey, Y. Zamora, Manning - Chapt. 9 and 10

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 3rd edition - Chapt. 1

  or

  Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 2nd edition - Chapt. 1-2

- Professional CUDA C Programming, J. Cheng, M. Grossman and T. McKercher, Wrox - Chapt. 1-2