AN INTRODUCTION TO GPUs

Modern computing for physics

J.Pazzini
Padova University

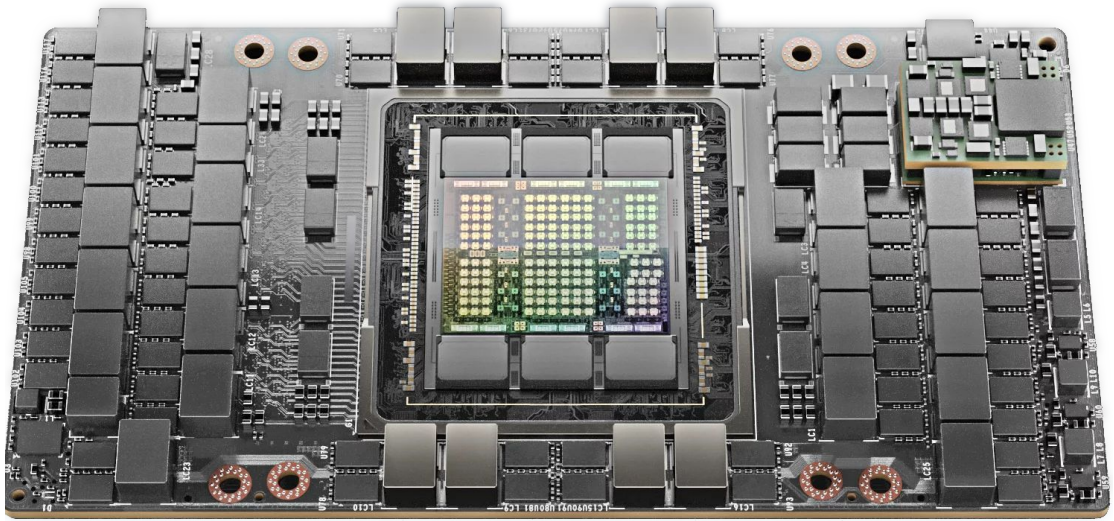
Physics of Data
AA 2024-2025

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

GPUs

Graphics Processing Units (GPUs) are devices equipped with:

- highly parallel microprocessors (from multi-cores to ➔ many-cores)
- dedicated high-bandwidth memory

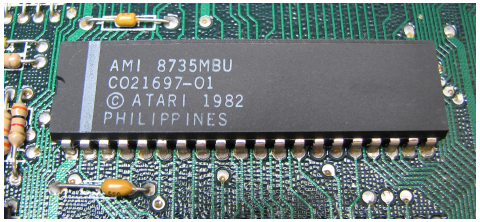


But it wasn’t the case at the beginning…

A historical overview

Originally... in medieval times

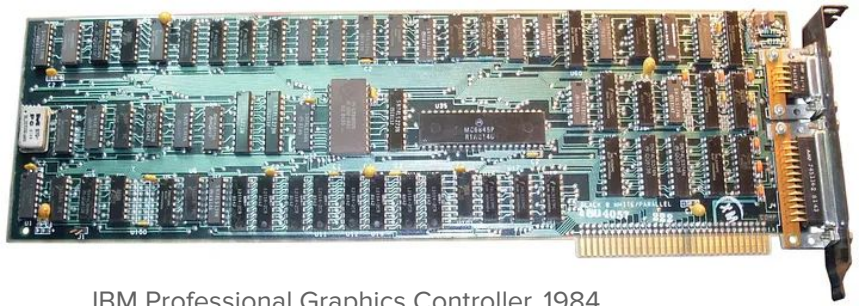
- In the 70s earlier “video-controller” were not dedicated micro-processors, but rather a specific HW component, integrally part of the processing device aimed to output specific video signals
- Most of the work for rendering graphics was still on the CPU



Alphanumeric Television Interface Controller (ANTIC), 1979
ASIC for 2D graphics on Atari 8-bit computers

The 80s

- Around the mid-80s the first discrete graphics “devices” saw the light as dedicated micro-processors, for specific graphics-oriented applications (e.g. CAD)
- Very specific applications and still relegated to large and dedicated systems

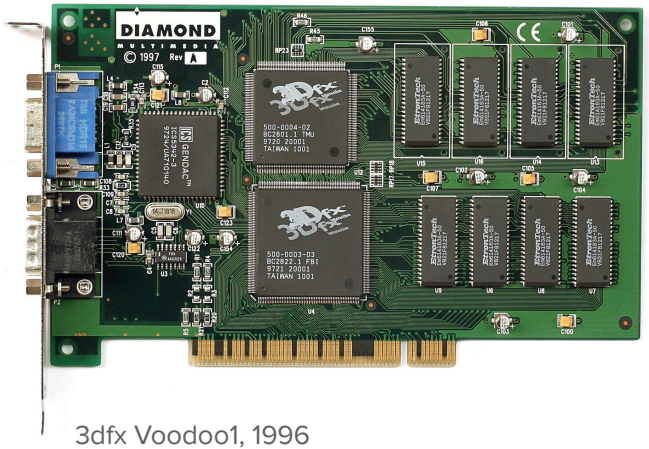


IBM Professional Graphics Controller, 1984
2D and 3D graphics for CAD-dedicated workstations
320kB of dedicated memory!

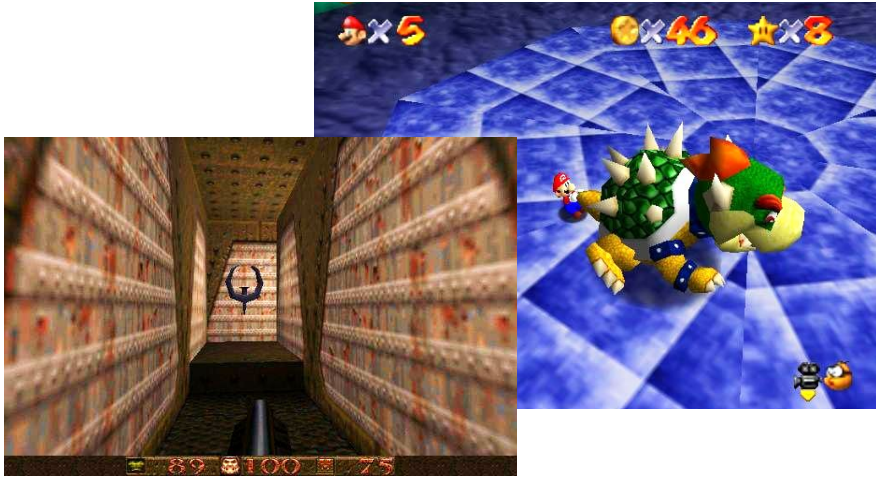
A historical overview

The 90s

- Timed with the explosion of graphical OSs (started in the mid-80s), a transition happened towards a fairly-modern concept of “video-card”. An independent board with a micro-processor dedicated to render 2D graphics, to be slotted into workstation PCs over BUS connections
- With the increasing popularity of 2D and the beginning of 3D graphics (videogames!) specific APIs and libraries for writing graphics applications targeted to multiple graphics devices
- Fixed-function graphics pipelines were used render graphics, but the CPU was still used massively for heavy duty-computations



3dfx Voodoo1, 1996
4MB of memory, 50MHz of clock, Win95 compatible!



A historical overview

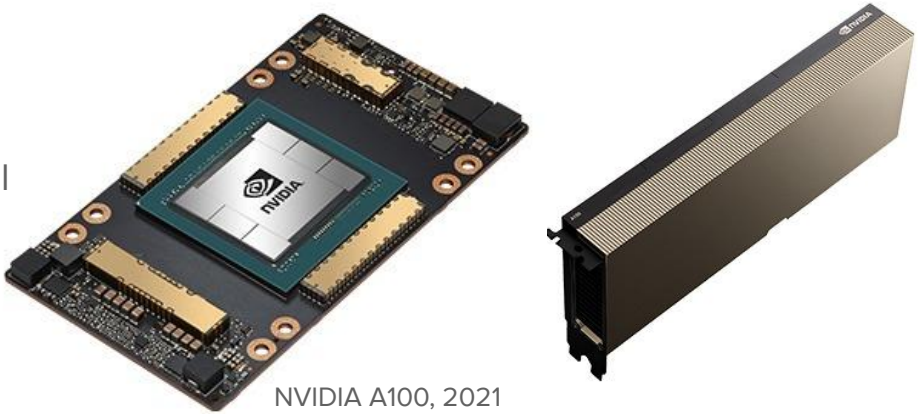
The 2000s

- The first graphics card fully-capable of running the entire graphics pipeline without leveraging complex computation on the CPU saw the light
- Transition toward full-shader flexible graphics cards, with the possibility of running fully customizable computations
- The first generalization from GPU to General Purpose GPU
- CUDA, OpenMP, OpenACC, ...

NVIDIA GeForce3, 2001
64MB of memory, 200MHz of clock
The first with fully-programmable shader

From then until today

- Large expansion of the GPU market, with several technological improvements
- More cores, throughput and energy efficiency
- Development of dedicated libraries for linear algebra, neural networks, tensor applications, ...
- Embedded systems with CPU+GPU



NVIDIA A100, 2021
80GB of memory, ~1GHz of clock
Only for computing - No Video output

From graphics to processing

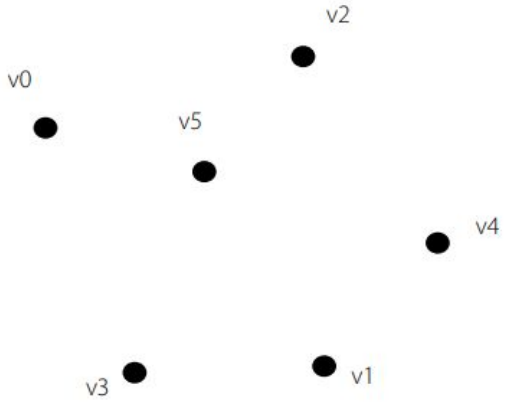
Images are broken down into triangles, and Vertices are located

Vertices are placed in the “screen space”

All “primitives” (triangles) are pixelized in the screen space (rasterization)

Shader apply functions (usually coloring) to each pixel

Every primitive is assembled together in the final frame



From graphics to processing

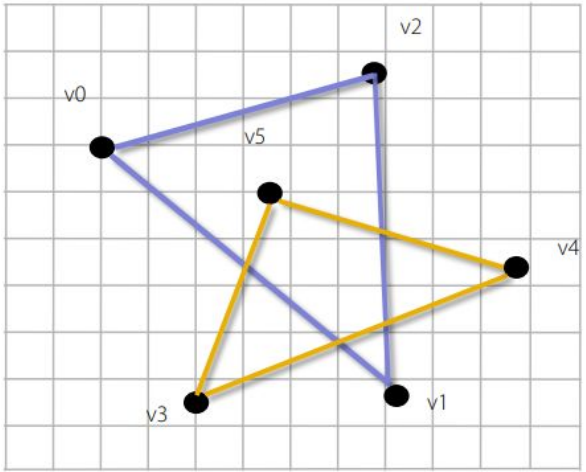
Images are broken down into triangles, and Vertices are located

Vertices are placed in the “screen space”

All “primitives” (triangles) are pixelized in the screen space (rasterization)

Shader apply functions (usually coloring) to each pixel

Every primitive is assembled together in the final frame



From graphics to processing

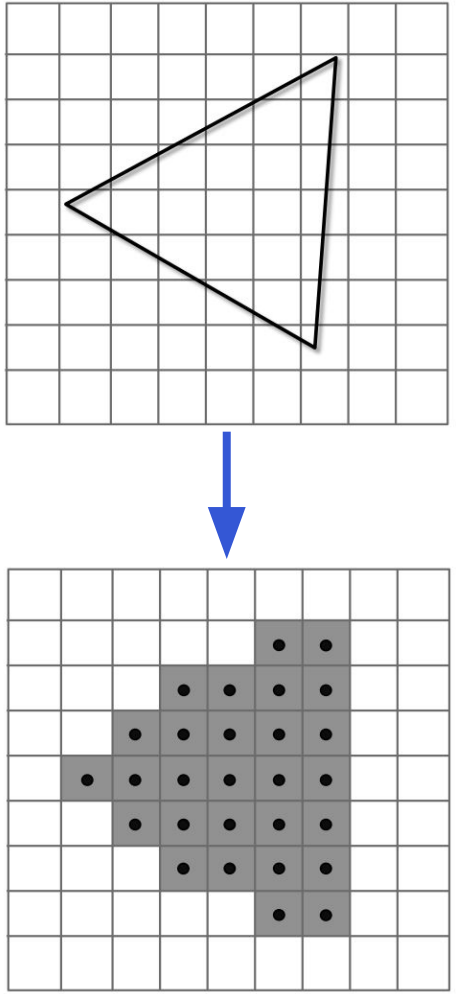
Images are broken down into triangles, and Vertices are located

Vertices are placed in the “screen space”

All “primitives” (triangles) are pixelized in the screen space (rasterization)

Shader apply functions (usually coloring) to each pixel

Every primitive is assembled together in the final frame



From graphics to processing

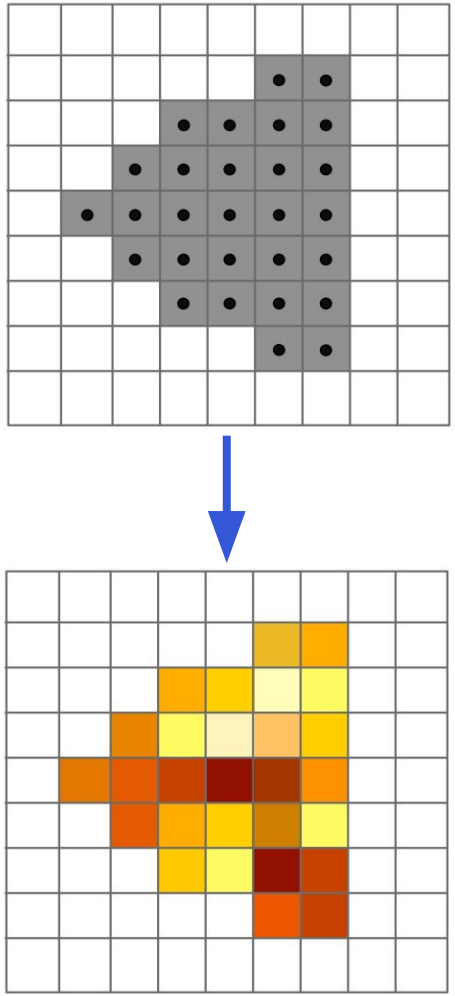
Images are broken down into triangles, and Vertices are located

Vertices are placed in the “screen space”

All “primitives” (triangles) are pixelized in the screen space (rasterization)

Shader apply functions (usually coloring) to each pixel

Every primitive is assembled together in the final frame



From graphics to processing

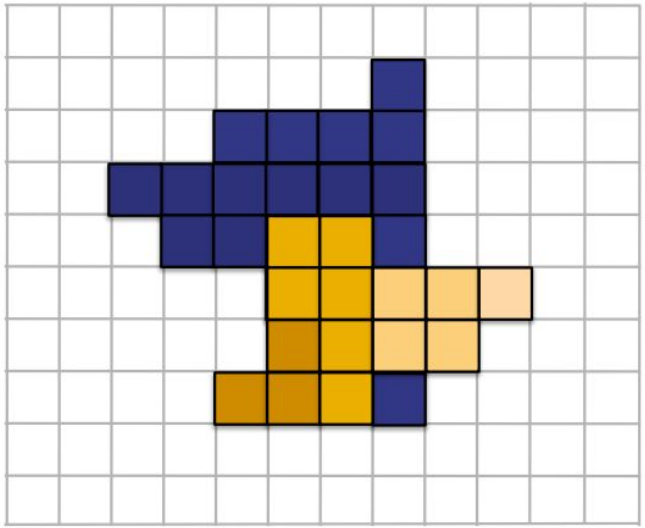
Images are broken down into triangles, and Vertices are located

Vertices are placed in the “screen space”

All “primitives” (triangles) are pixelized in the screen space (rasterization)

Shader apply functions (usually coloring) to each pixel

Every primitive is assembled together in the final frame



From graphics to processing

Images are broken down into triangles, and Vertices are located

Vertices are placed in the “screen space”

All “primitives” (triangles) are pixelized in the screen space (rasterization)

Shader apply functions (usually coloring) to each pixel

Every primitive is assembled together in the final frame

All of this, at ~60 frames per second, for 1 million triangles and 6 million pixels per frame

From graphics to processing

Images are broken down into triangles, and Vertices are located

Vertices are placed in the “screen space”

All “primitives” (triangles) are pixelized in the screen space (rasterization)

Shader apply functions (usually coloring) to each pixel

Every primitive is assembled together in the final frame

All of this, at ~60 frames per second, for 1 million triangles and 6 million pixels per frame

How much parallelization is potentially in this process?

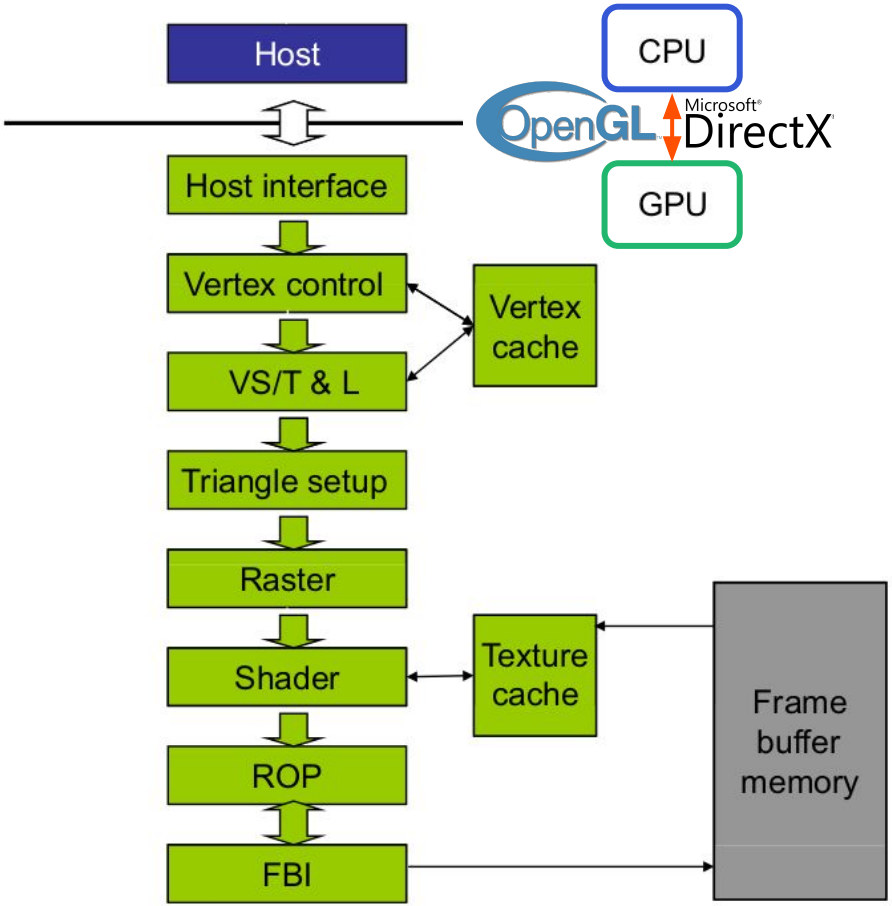
EACH VERTEX IS TRANSFORMED INDEPENDENTLY

EACH PRIMITIVE IS RASTERIZED INDEPENDENTLY

EACH FRAGMENT IS PROCESSED INDEPENDENTLY TO SHADE EACH PIXEL

POTENTIALLY, A LOT!

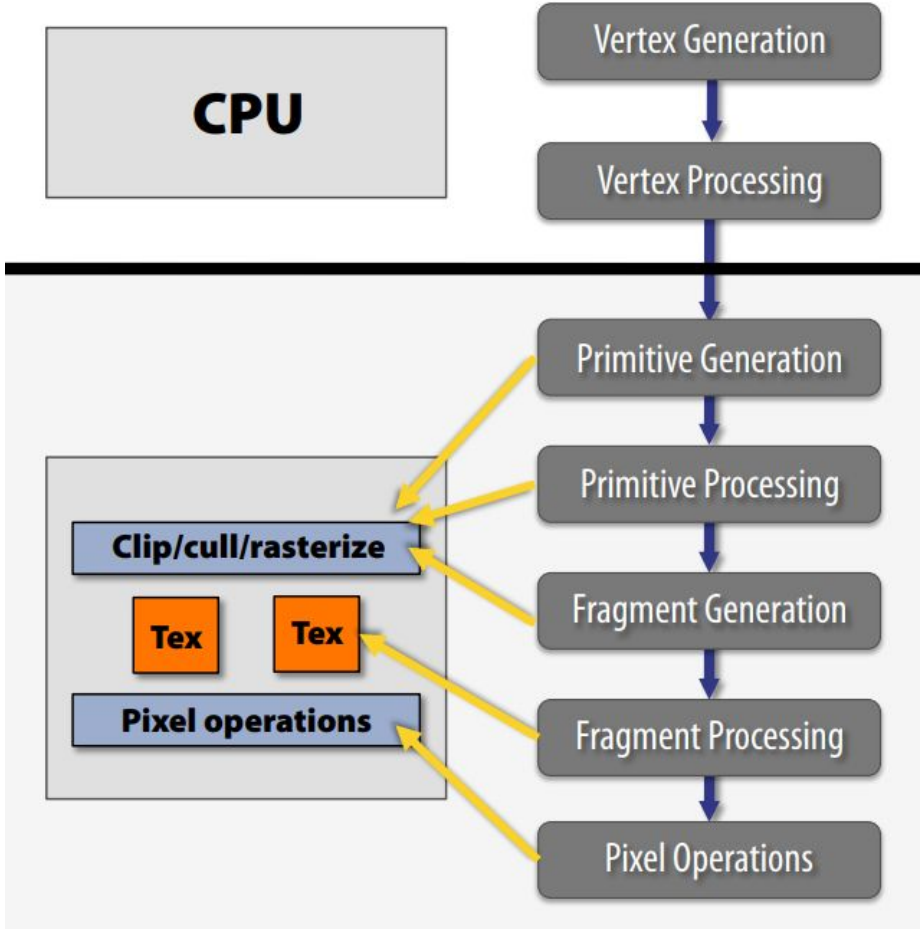
GPUs pre-2000… mostly fixed-function pipelines



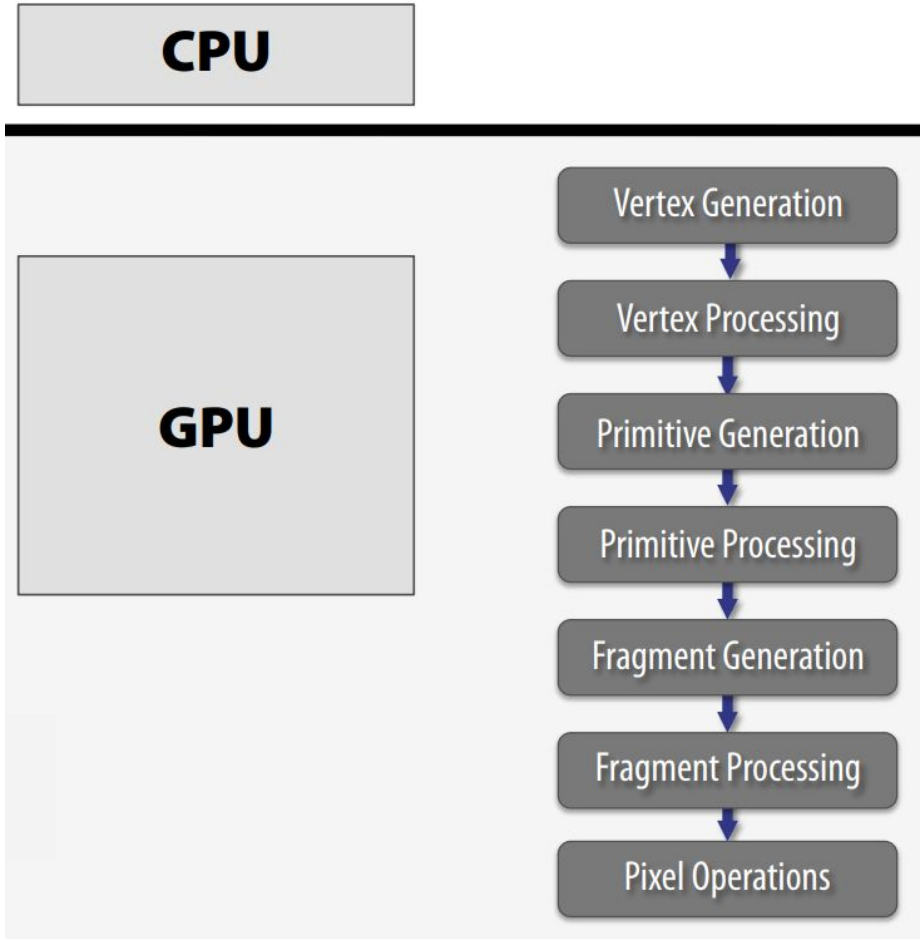
Configurable *(but not programmable)* fixed-function pipelines:

- Host interface receives data and graphics instructions from the CPU
- APIs “translate” and “ship” them to the GPU via DMA transfers
  - Direct Memory Access → bulk data transfer from memory to memory w/o burdening the CPU
- Vertex control, Rastering, and Shading set values to be applied at the
- A final set of ROP (Raster OPeration) actions are performed onto the entire frame (e.g. anti-aliasing)
- The FBI (Frame Buffer Interface) stage manages the frames read/write to memory

GPUs pre-2000... mostly fixed-function pipelines

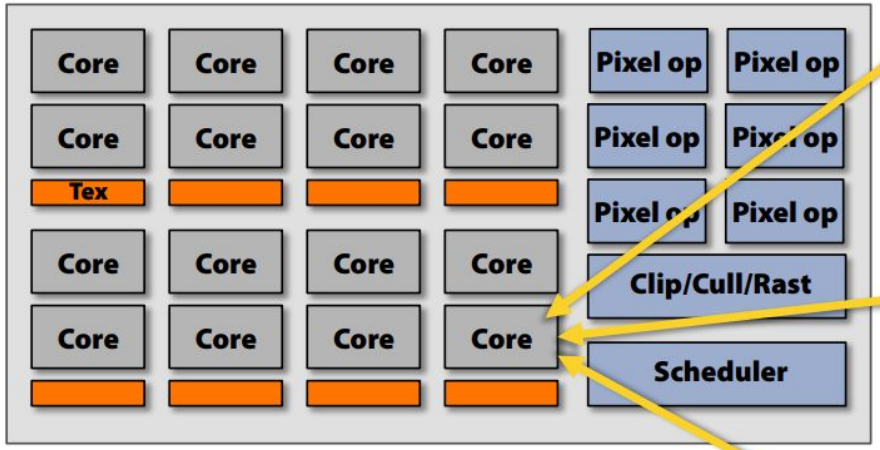


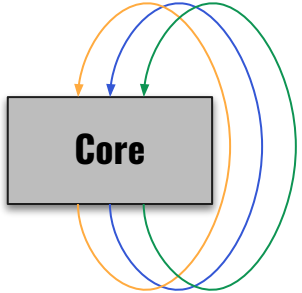
GPUs around 2000... moving vertexing onto GPU



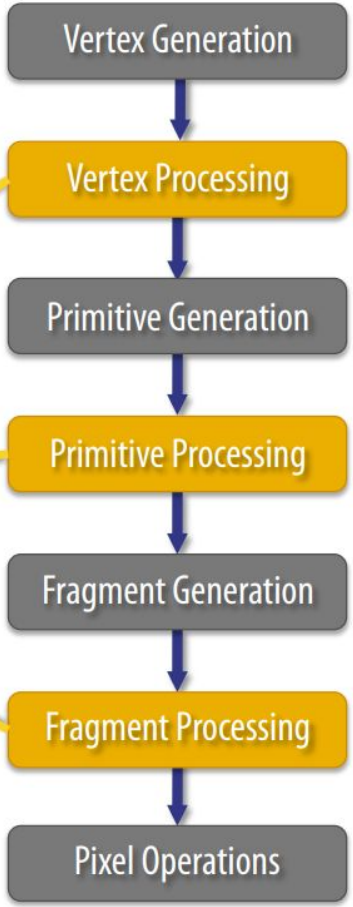
Fast forward to circa 2006… unified processors

Separate **programmable graphics stages** are sent to an **array of unified processors**





The logical graphics pipeline is physically a recirculating path that visits these processors three times



Post-2006… General Purpose GPU for parallel computing

The use of unified processors for general shaders opened the door to (*outstandingly smart and madly skilled*) researchers to use this parallel architecture to their advantage

V

Intel Xeon 7100 (from 2006)

2 cores (4 threads) @ 3.4 GHz
16kB L1 cache
1MB L2 cache
16MB shared L3 cache

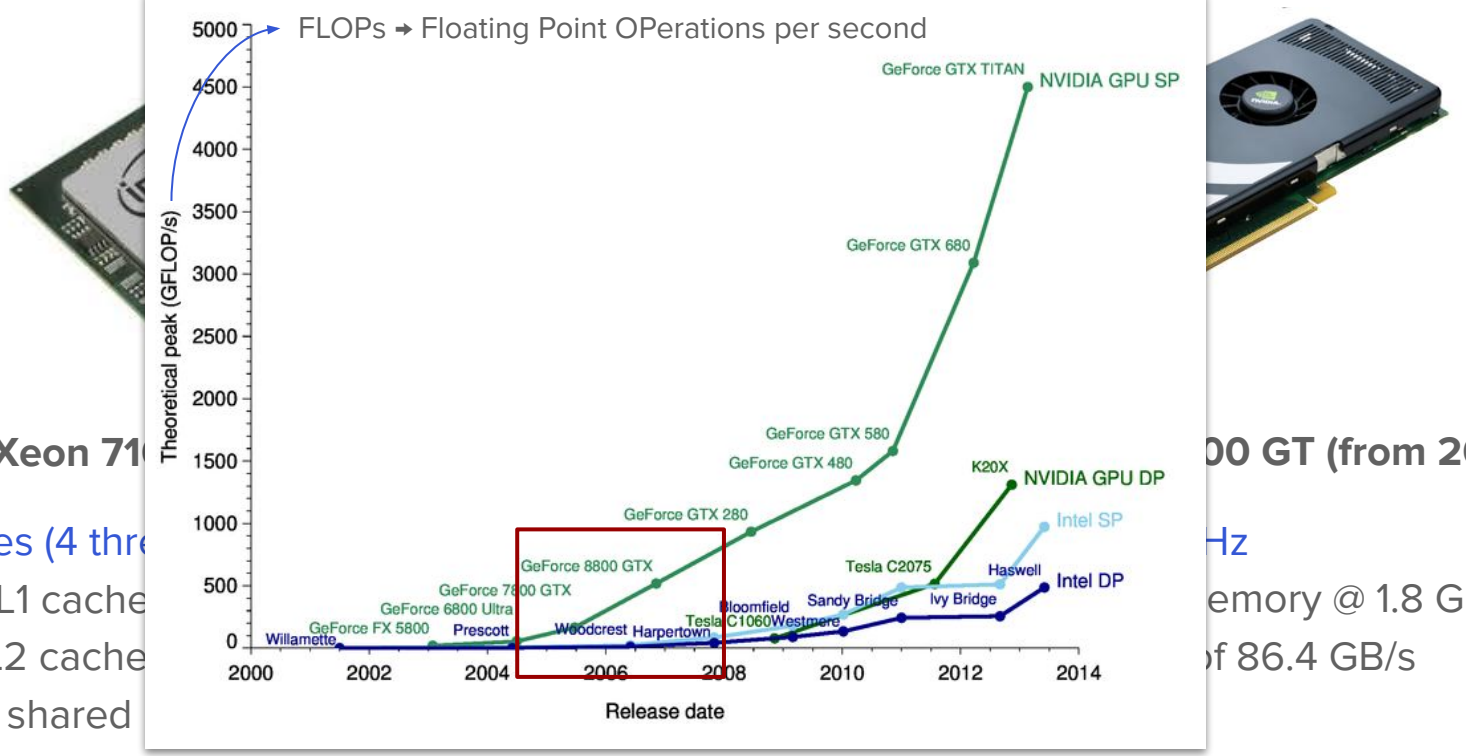
NVIDIA GeForce 8800 GT (from 2006)

128 “cores” @ 1.35 GHz
768 MB of GDDR3 memory @ 1.8 GHz
Memory bandwidth of 86.4 GB/s

https://www.nvidia.co.uk/content/PDF/Geforce_8800/GeForce_8800_GPU_Architecture_Technical_Brief.pdf

Post-2006… General Purpose GPU for parallel computing

The use of unified processors for general shaders opened the door to (*outstandingly smart and madly skilled*) researchers to use this parallel architecture to their advantage



Intel Xeon 71

2 cores (4 thr

16kB L1 cache

1MB L2 cache

16MB shared

00 GT (from 2006)

Hz

emory @ 1.8 GHz

of 86.4 GB/s

Post-2006... General Purpose GPU for parallel computing

However, despite being comprised of a set of (now *relatively* general purpose) processors, GPUs were still designed to perform image processing and image manipulation

⇒ To issue a parallelizable scientific problem to a GPU at the time, one had to:

- reshape the input data as an “image” (pixels, triangles, vertices)
- program the GPU cores with appropriate “custom shader” functions to operate on data inputs
- memorize intermediate stages of the computation as image frames in the frame memory buffers
- output the result of the task as the set of pixels of a 2D image

From 2010s to today... GPUs as computing tools

Generalization on how GPUs can be programmed (CUDA, OpenMP, OpenACC, etc) lead to the possibility of treating GPUs as “standard” processors, not anymore necessarily tied to graphics-related workloads.

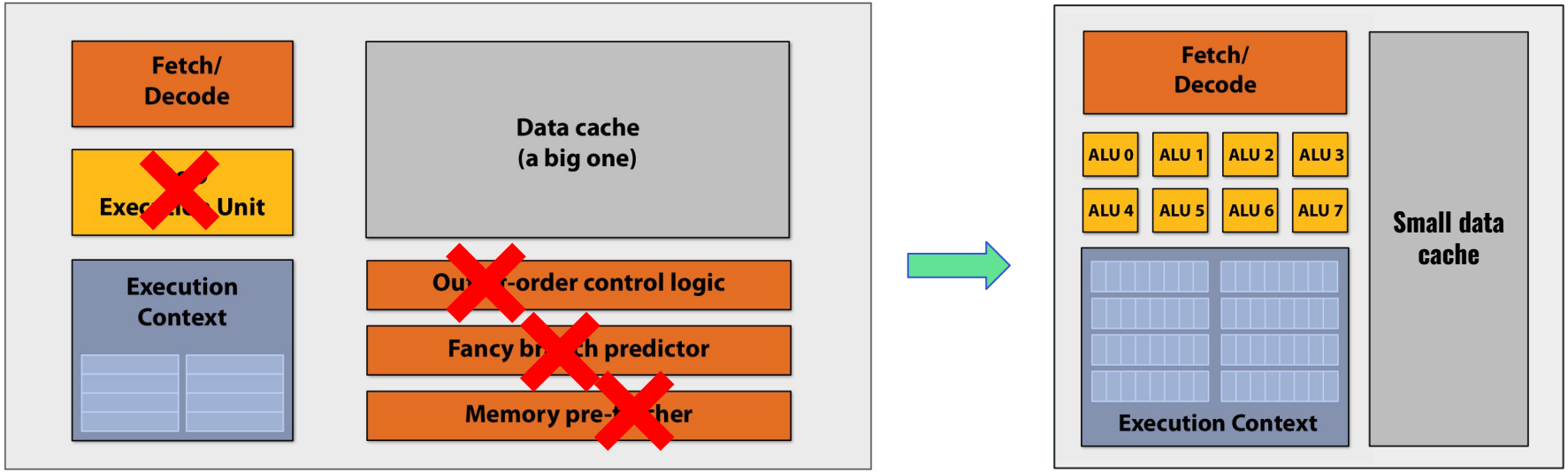
➜ No more need to program via the OpenGL or DirectX *graphics* APIs

From 2010s to today... GPUs as computing tools

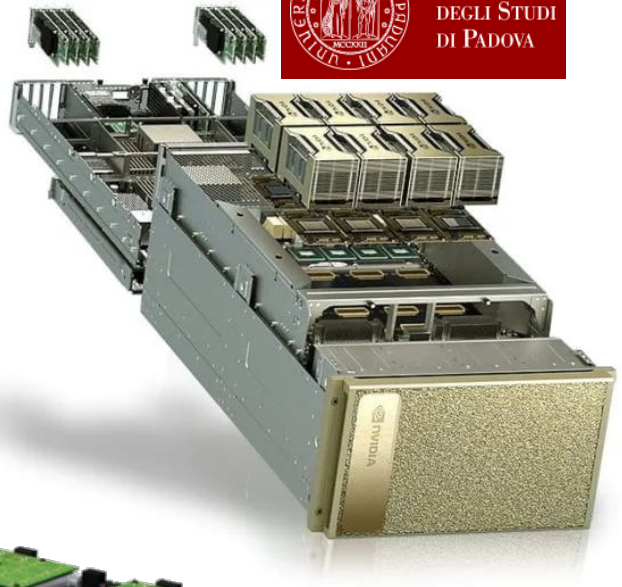
Memory on board to store intermediate and final results not as images, but as “memory reserved locations”, as in standard CPUs

GPU cores became very similar to CPUs, but with simpler logic: no OoO execution or branch prediction, and smaller cache

➜ but **many simpler cores (with SIMD ALUs)** instead of a few complex and performant ones



And nowadays, GPUs are ~everywhere...



cloudveneto

GPU Booking Calendar

Available GPUs:

- Nvidia Tesla V100_1
- Nvidia Tesla V100_2
- Nvidia Tesla V100_3
- Nvidia Tesla V100_4
- Nvidia Tesla T4_1
- Nvidia Tesla T4_2
- Nvidia Tesla T4_3
- Nvidia Tesla T4_4
- Nvidia Tesla T4_5
- Nvidia Tesla T4_6
- Nvidia Tesla T4_7
- Nvidia Tesla T4_8
- Nvidia Tesla T4_9
- Nvidia Tesla T4_10
- Nvidia Tesla T4_11
- Nvidia Tesla T4_12
- Nvidia Quadro
- Nvidia TitanXP_1
- Nvidia TitanXP_2
- Nvidia Geforce GTX

Tesla T4 GPUs marked in orange can b

Tesla T4 GPUs marked in blue can be used thro

today

Sun

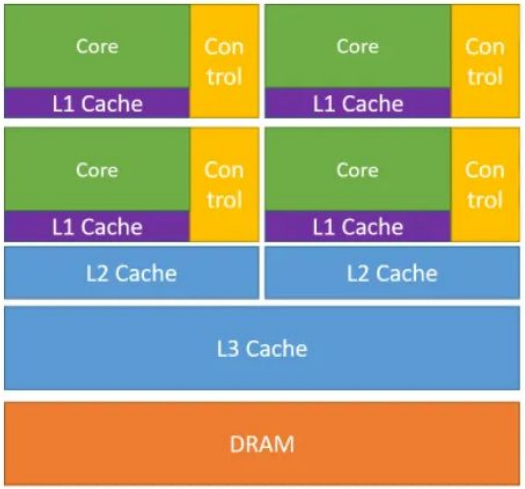
28

- Nvidia Tesla T4_5
- Nvidia Tesla T4_6
- Nvidia Tesla V100_1
- Nvidia Tesla T4_3
- Nvidia Tesla T4_7
- Nvidia Tesla T4_8
- Nvidia Tesla T4_10
- Nvidia Tesla T4_2
- Nvidia Tesla V100_2
- Nvidia Tesla V100_4
- Nvidia Tesla T4_4
- Nvidia Tesla T4_1
- Nvidia Tesla T4_9
- Nvidia Tesla T4_11
- Nvidia Tesla T4_12

LEONARDO CINECA

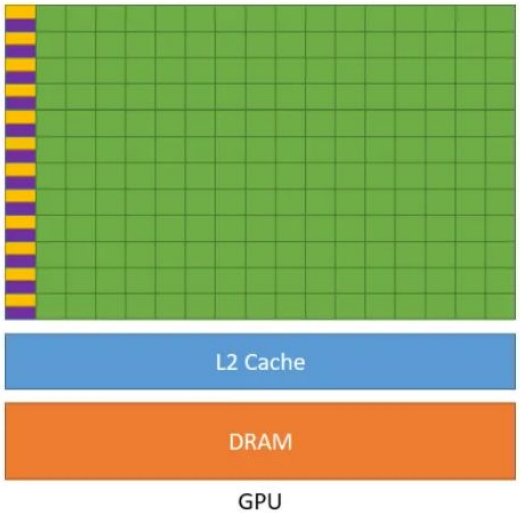
Let’s understand how to exploit them

CPU v GPU



CPU

- Multi-core
- Complex and powerful ALUs
- Large cache to limit CPU ↔ Memory latency
- OoO execution to reduce instruction latency



GPU

- Massively parallel architecture (many cores)
- Simple ALUs, but efficient for pipelining
- Small cache but large memory bandwidth
- Simple control (no OoO execution, simply compute in order)

CPU v GPU

Latency oriented

⇒ Optimised for serial code performance (single thread)

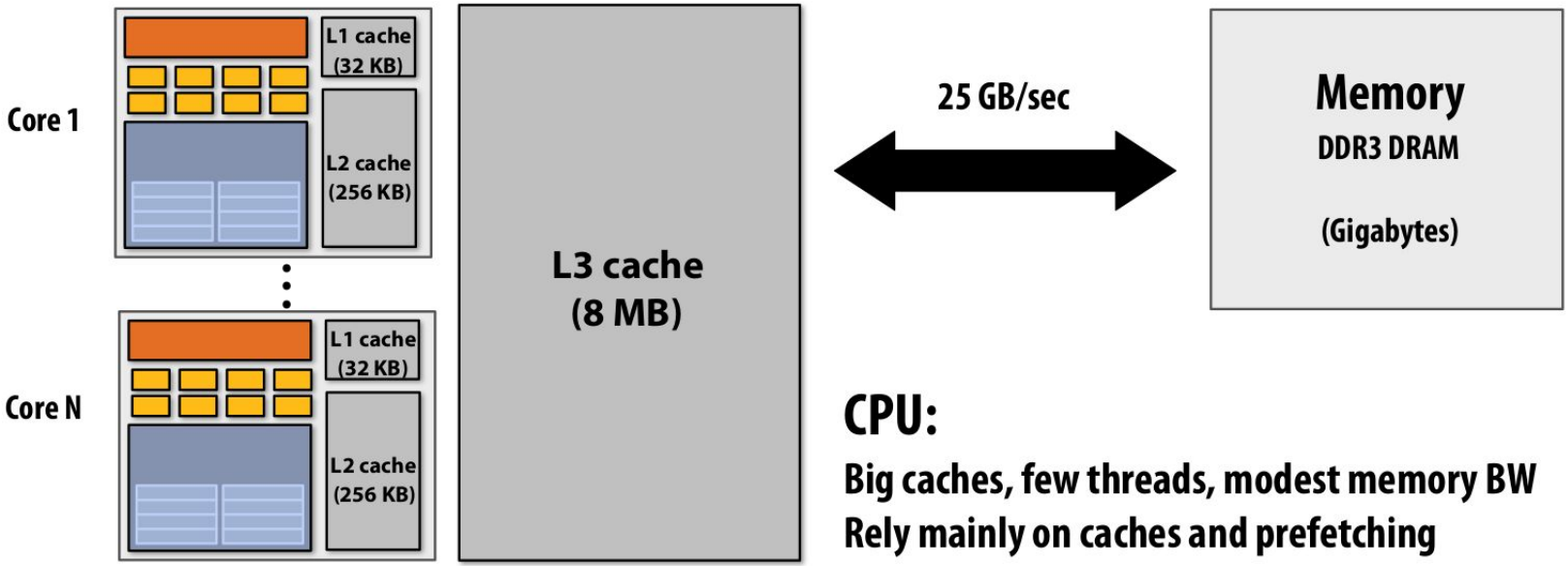
⇒ Good for single, complex, branching tasks

Throughput oriented

⇒ Optimised for performing many similar tasks simultaneously (data parallel)

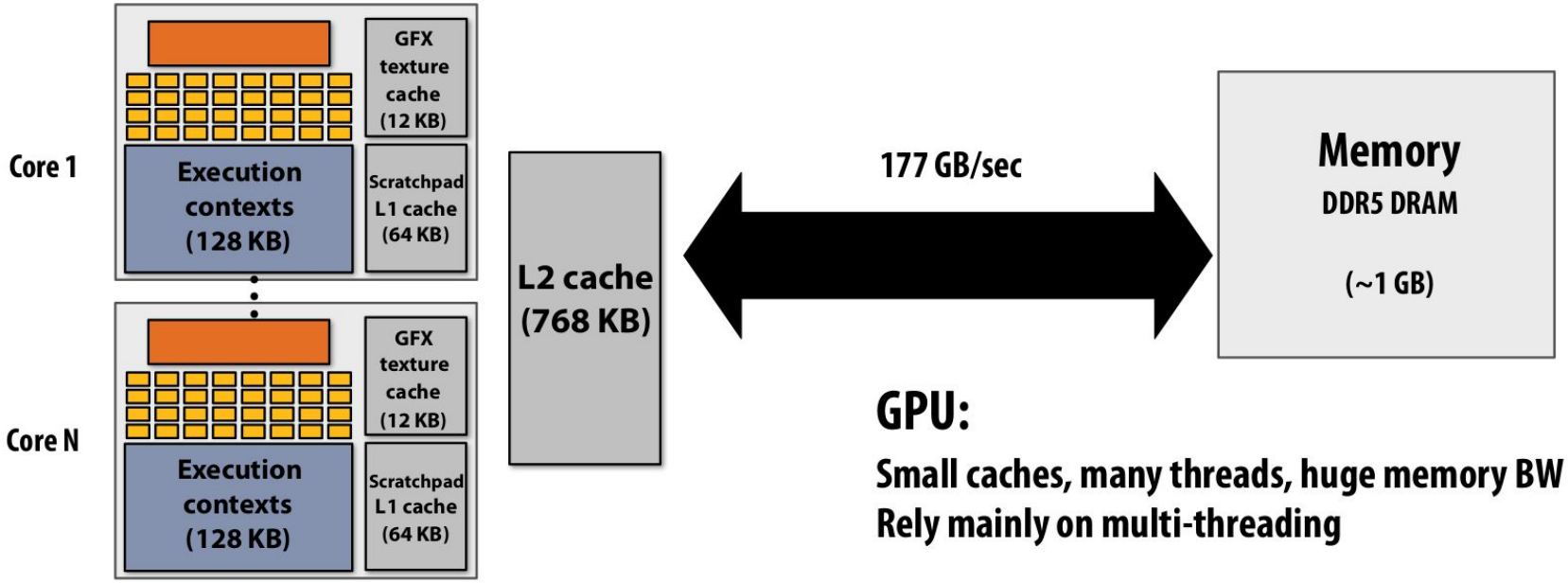
⇒ Good for lots of simple, deterministic tasks

CPU v GPU



May result in being (e.g.) 10x faster than a GPU for complex sequential tasks

CPU v GPU



May result in being (e.g.) 10x faster than a CPU for heavily parallel tasks

Taxonomy of a GPU

- GPUs are designed to tackle heavily data-parallel problems
- But what (Flynn) taxonomy best describes data parallelism with a GPU?
  - SISD
  - SIMD
  - MISD
  - MIMD
  - SPMD

Taxonomy of a GPU

- GPUs are designed to tackle heavily data-parallel problems
- But what (Flynn) taxonomy best describes data parallelism with a GPU?
  - SISD ➜ absolutely not
  - SIMD ➜ kinda, but not all processors are synched to run the same instruction at the same time
  - MISD ➜ definitely not
  - MIMD ➜ it's way more parallel than simply this
  - SPMD ➜ kinda, but not exactly the same "program"; definitely multiple data though

Taxonomy of a GPU

- GPUs are designed to tackle heavily data-parallel problems
- But what (Flynn) taxonomy best describes data parallelism with a GPU?
  - SISD ➜ absolutely not
  - SIMD ➜ kinda, but not all processors are synched to run the same instruction at the same time
  - MISD ➜ definitely not
  - MIMD ➜ it's way more parallel than simply this
  - SPMD ➜ kinda, but not exactly the same "program"; definitely multiple data though
- It looks like NVIDIA calls the modern GPU taxonomy **SIMT**
  - **Single Instruction Multiple Threads**
  - Same instruction run simultaneously by multiple independent threads (on different data inputs)

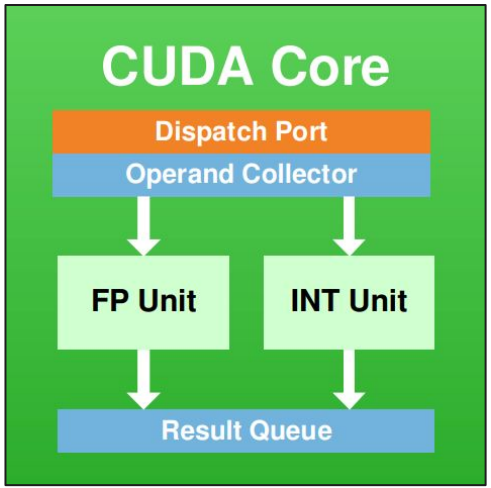
    e.g. $c_i = \eta * a_i + b_i$ in each thread, with $0<i<1000$, i.e. 1000 independent threads

  - It's like having SIMD combined with multithreading

HW view of a (NVIDIA) GPU

- The GPU architecture is built around a 3-level hierarchy

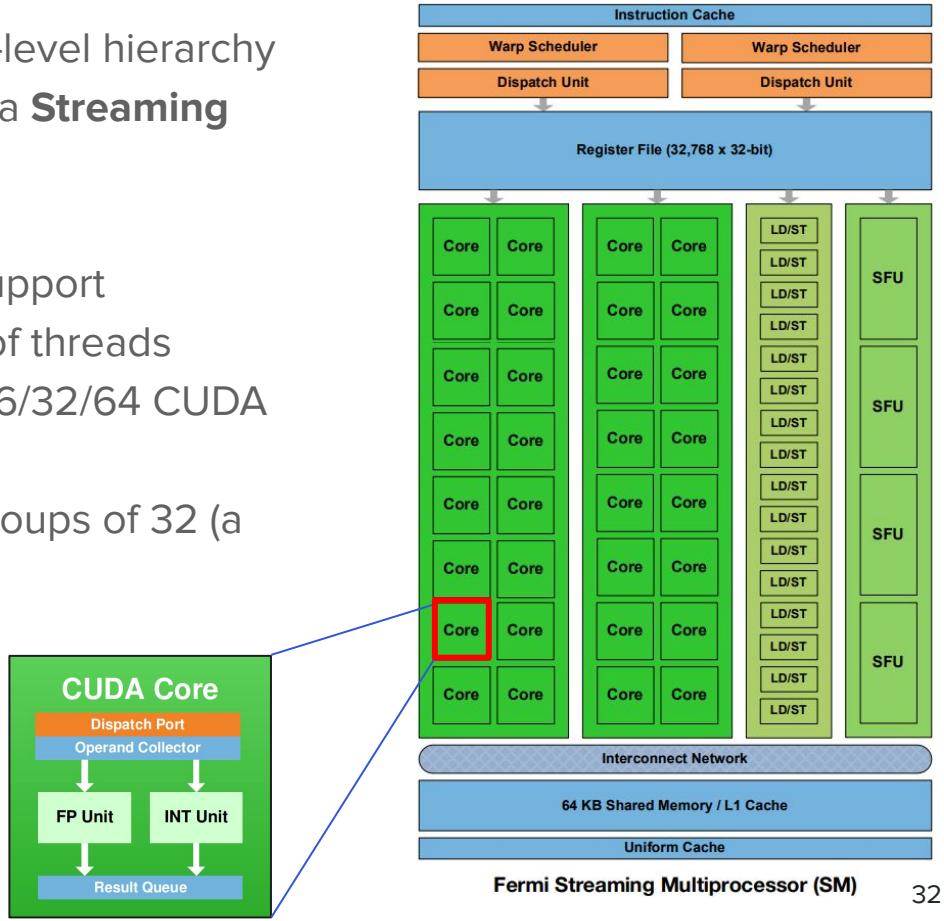
1. The NVIDIA **CUDA Core** (or Streaming Processor, **SP**) is the innermost processing unit
   - Usually scalar processors (i.e. SISD) with no OoO ex., branch pred., etc...
     - In more modern GPU CORES also vector instructions (SIMD) inside the SP
   - ALUs to operate on Integers, Float16 (half-precision), Float32 (single-precision), Float64 (double-precision) depending on the architecture
   - Works on a single operation per clock cycle, and fully pipelinable



HW view of a (NVIDIA) GPU

- The GPU architecture is built around a 3-level hierarchy

2. An array of CUDA cores are arranged in a **Streaming MultiProcessor** (**SM**, or **SMP**)
   - Similarly to a multi-core processor
   - Each SM in a GPU is designed to support concurrent execution of hundreds of threads
   - In a SM are arranged a number of 16/32/64 CUDA cores
   - NVIDIA GPUs execute threads in groups of 32 (a *warp*, more on that later).
   - All threads in a *warp* execute:
     - the same instruction
     - at the same time



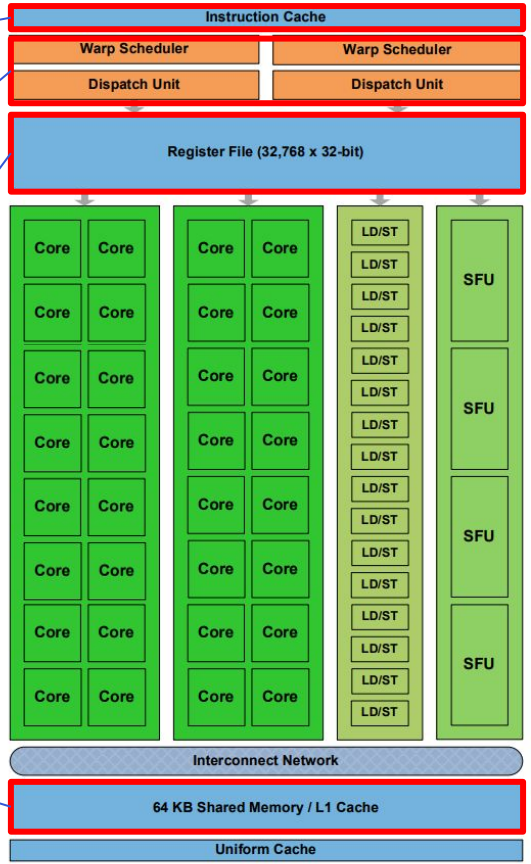
HW view of a (NVIDIA) GPU

Instruction cache (common across all SPs in a SMP)

Scheduler / dispatcher

1 scheduler per warp (32 cores)

Registers (common across all SPs in a SMP)



Fermi Streaming Multiprocessor (SM)

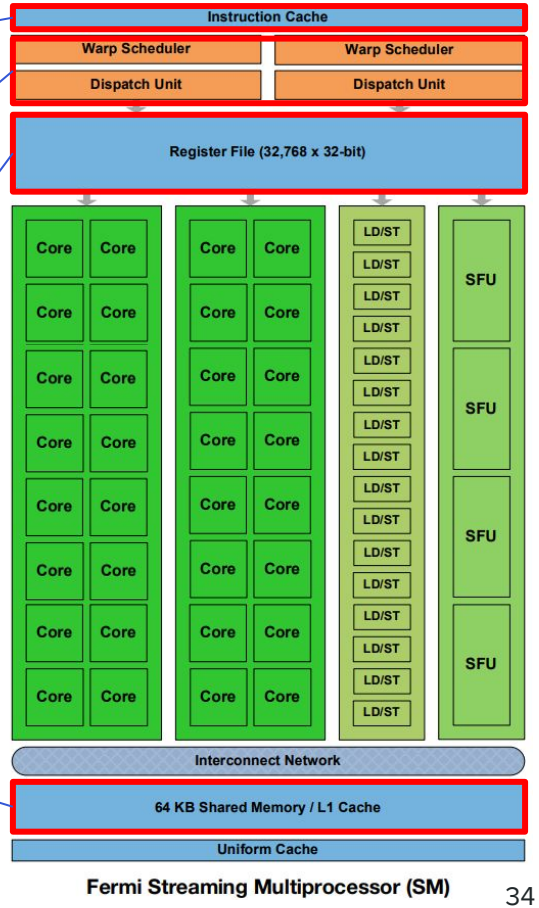
Small L1-like cache, but shared across all SPs in a SMP

HW view of a (NVIDIA) GPU

Instruction cache (common across all SPs in a SMP)

Scheduler / dispatcher
1 scheduler per warp (32 cores)

Registers (common across all SPs in a SMP)



Fermi Streaming Multiprocessor (SM)

shared cache and control logic across all SPs in a SMP
acting as a multi-core CPU with SIMD "inner" processing units

REMEMBER → task scheduling and cache is common to all SPs!

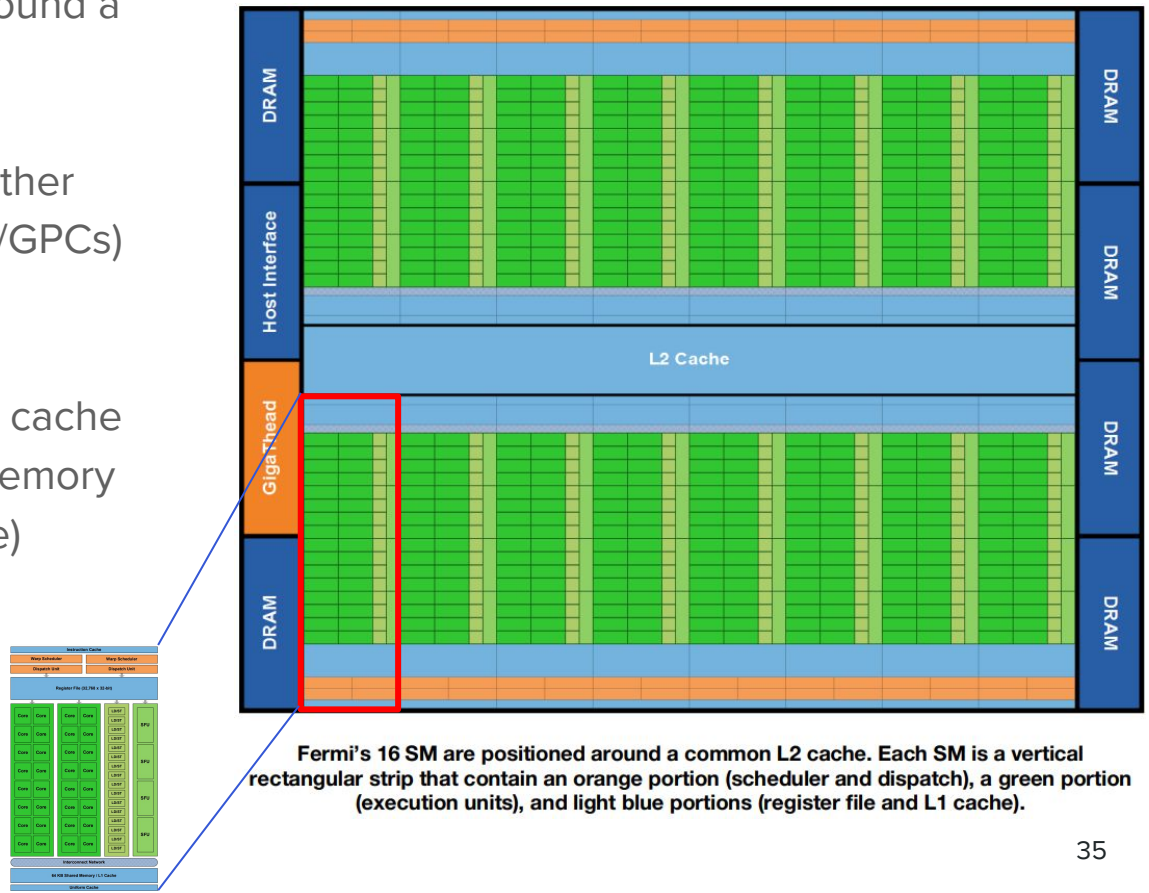
⇒ All CUDA cores do the same thing (on different data inputs)

Small L1-like cache, but shared across all SPs in a SMP

HW view of a (NVIDIA) GPU

- The GPU architecture is built around a 3-level hierarchy

3. Multiple SMPs per GPU
    - Sometimes arranged in further Processing Clusters (TPCs/GPCs)
    - All SMPs are *independent*
    - They only share:
        - Larger, but slower, L2 cache
        - Access to the GPU memory (slower than L2 cache)



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

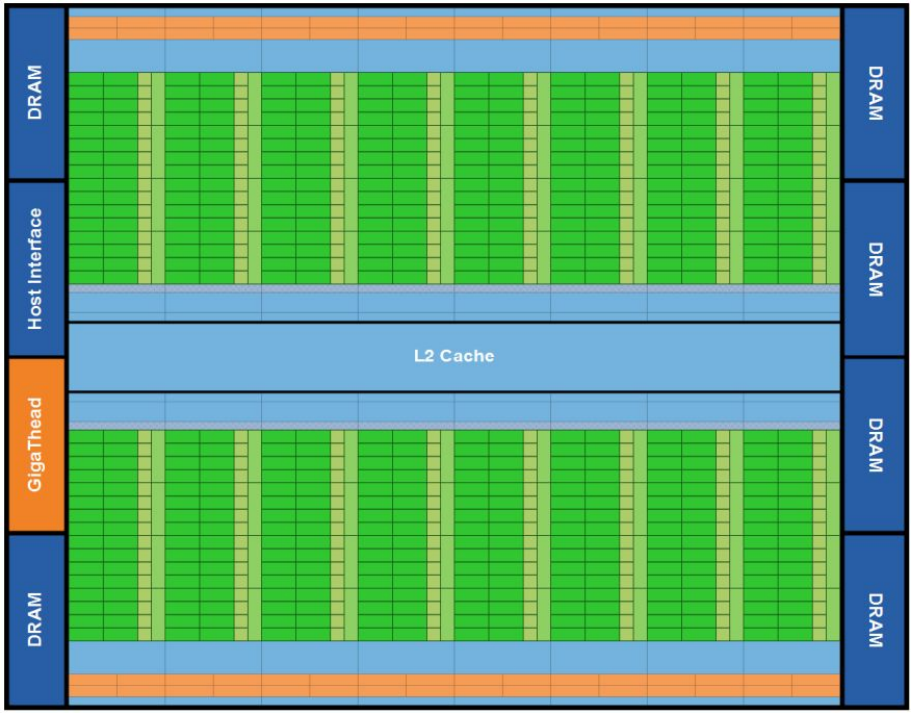
Fermi Family (2010)

16 SMPs
32 SP per SMP
⇒ 512 CUDA cores

up to 48kB L1 Cache (per SM)

768 kB L2 Cache

4-6 GB Memory



https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

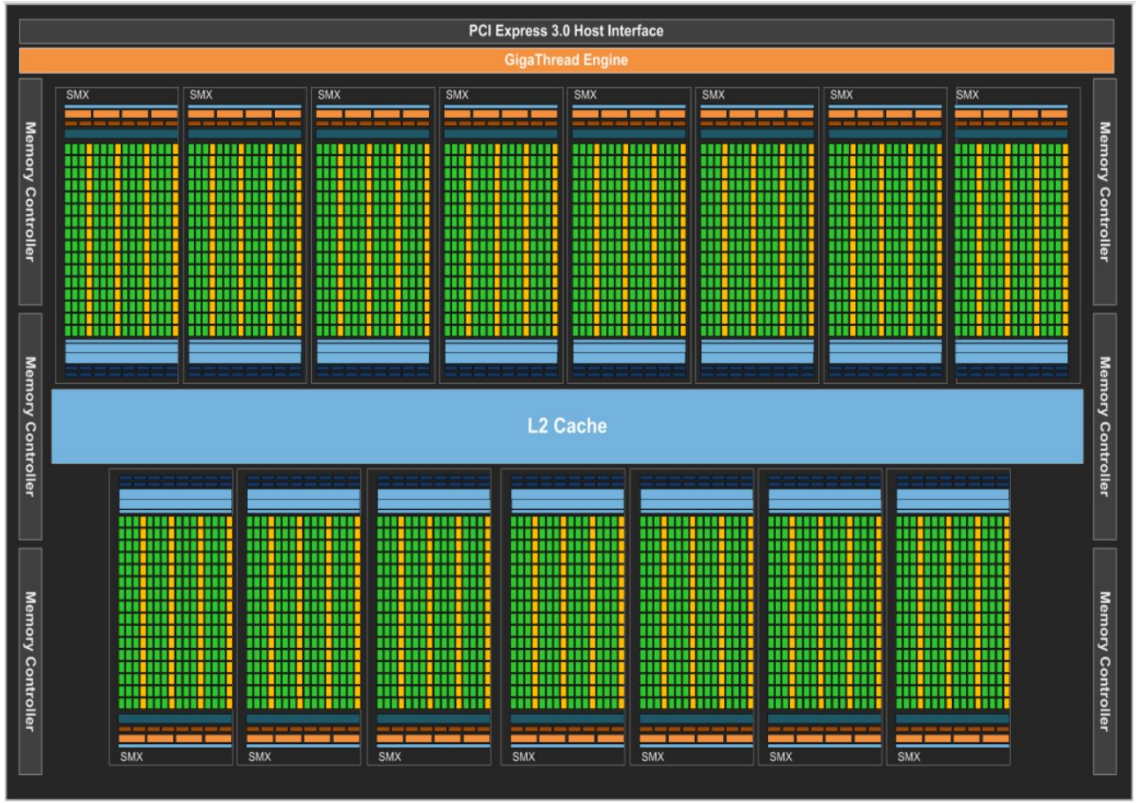
Kepler Family (2012)

15 SMPs
192 SP per SMP
⇒ 2880 CUDA cores

up to 48kB L1 Cache (per SM)

1536 kB L2 Cache

12 GB GDDR5 Memory



https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

Maxwell Family (2014)

4 PCs

8 SMPs per PC
⇒ 32 SMPs

128 SP per SMP
⇒ 4096 CUDA cores (@1.1 GHz)

2MB L2 Cache

https://www.techpowerup.com/gpu-specs/docs/nvidia-gtx-980.pdf

Maxwell Family (2014)

4 PCs

8 SMPs per PC
⇒ 32 SMPs

128 SP per SMP
⇒ 4096 CUDA cores (@1.1 GHz)

2MB L2 Cache

https://www.techpowerup.com/gpu-specs/docs/nvidia-gtx-980.pdf

Pascal Family (2016)

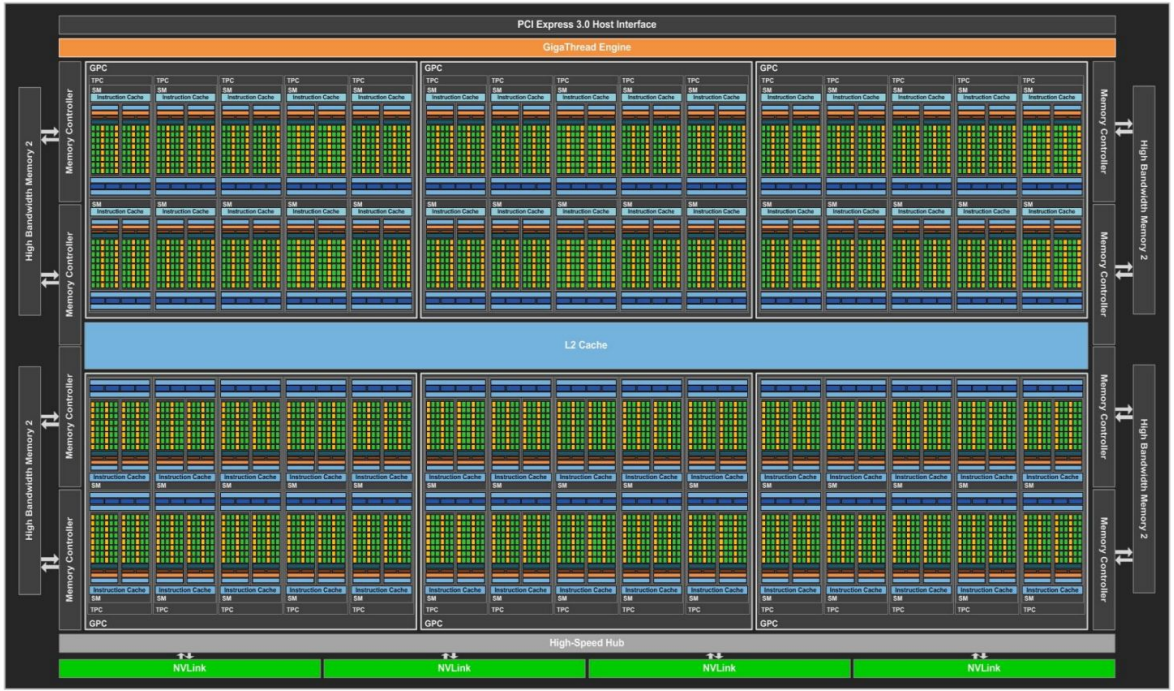
6 GPCs w/ 5 TPCs per GPG (!?)
⇒ 30 GPC…(?)

10 SMPs per PC
⇒ up to 60 SMPs in GPU

64 SP per SMP
⇒ 3840 CUDA cores (@1.3 GHz)

Different types of specialized CUDA cores

4MB L2 Cache
16GB of HBM2 (High Bandwidth Memory)



NVLINK for GPU ↔ GPU connectivity

https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

Pascal Family (2016)

6 GPCs w/ 5 TPCs per GPG (!?)
⇒ 30 GPC…(?)

10 SMPs per PC
⇒ up to 60 SMPs in GPU

64 SP per SMP
⇒ 3840 CUDA cores (@1.3 GHz)

Different types of specialized CUDA cores

4MB L2 Cache
16GB of HBM2 (High Bandwidth Memory)

NVLINK for GPU ↔ GPU connectivity

https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

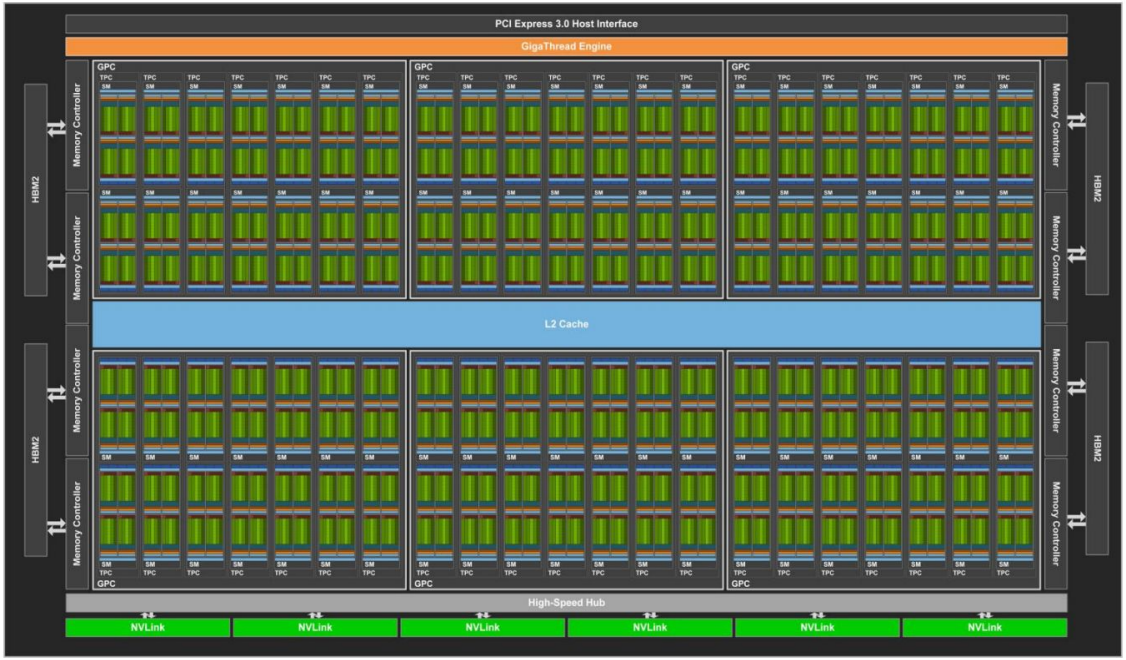
Volta Family (2017)

[...]

⇒ 5120 CUDA cores

New *Tensor Cores* for faster Matrix-to-Matrix operations

6MB L2 Cache



16/32GB of HBM2 (High Bandwidth Memory)

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

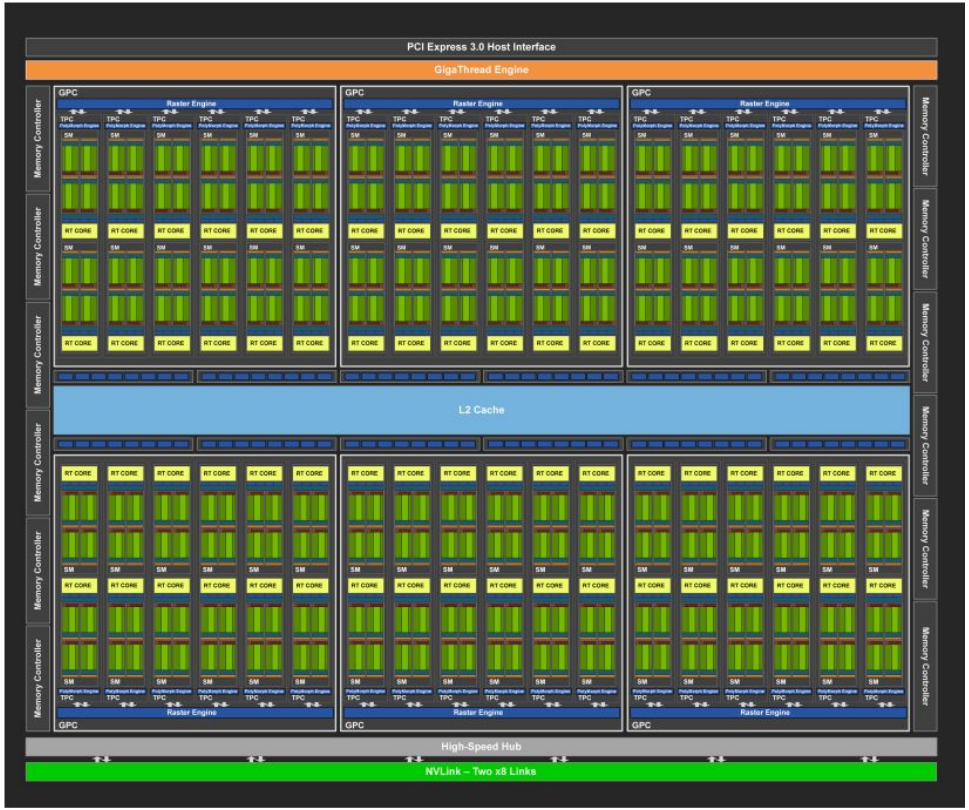
Turing Family (2018)

[...]

⇒ 4608 CUDA cores (~1.4 GHz)

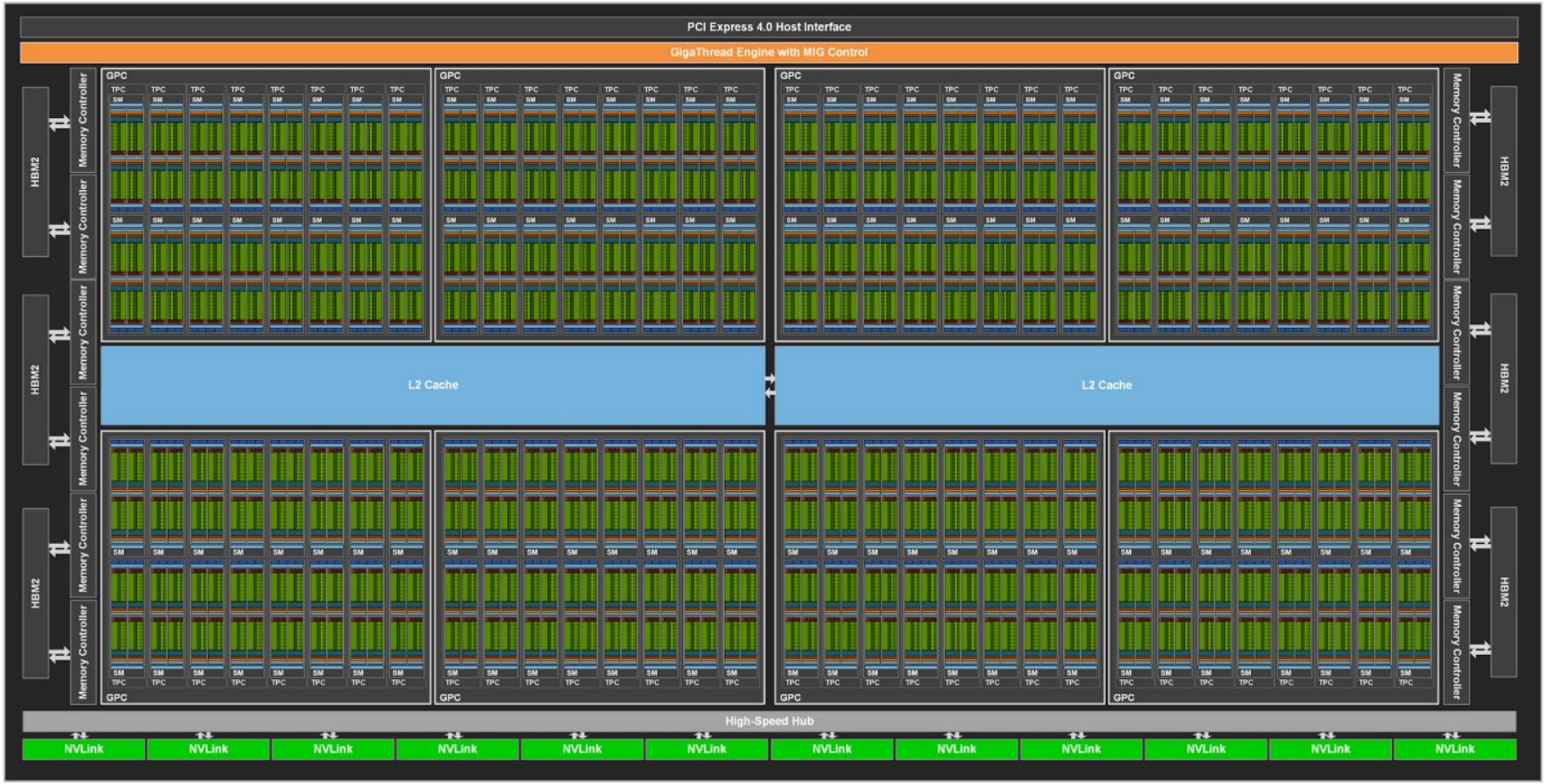
⇒ 576 Tensor cores

[...]



https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

Ampere Family (2020)



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

Hopper Family (2022)

https://resources.nvidia.com/en-us-tensor-core?ncid=no-ncid

It’s not that there is only NVIDIA on the market…

Also AMD (with the Radeon Instinct line) and INTEL (with the Intel Data Center GPU Max Series) offer a portfolio of GPUs for computing applications

If we looked at the diagrams of AMD’s or INTEL’s GPUs, we would see an almost identical hierarchical structure, although with a different naming convention wrt the one used by NVIDIA, e.g.: Stream MultiProcessor ➔ Compute Units

We are (mostly) going to target NVIDIA in this course as:

- all GPUs you can access on CloudVeneto are NVIDIA
- we have access to small developer boards from NVIDIA
- NVIDIA is currently experiencing a dominant position on the market

However, a very similar logic applies for the programming of most GPUs

One consideration on GPU memory

- Given that GPUs are designed for extremely data-parallel applications, the data transfer to-from CUDA cores and Memory is a key factor in the GPU performance

⇒ GPUs can process at most what can transfer to the SPs!

⇒ GPU works efficiently when exploiting the memory bandwidth (high throughput)

An extremely simplified example:

- A NVIDIA V100 is equipped with 16 GB HBM2 memory, with a memory bandwidth 900 GB/s, and 5120 SPs clocked at max 1.53 GHz
- Assume we don't rely on L1/L2 cache or SPs registers
  ➡ the SPs have to access every Byte of data directly from memory
- Assume each SP is performing a simple Floating-Point (4Bytes) operation, in parallel over all SPs

The maximum *ideal* rate of FP OPerations per Second (FLOPS) before exceeding the memory bandwidth would be:

FLOPS = 900 GB/s / 4B = 225 GFLOPS

One consideration on GPU memory

- Given that GPUs are designed for extremely data-parallel applications, the data transfer to-from CUDA cores and Memory is a key factor in the GPU performance

⇒ GPUs can process at most what can transfer to the SPs!
⇒ GPU works efficiently when exploiting the memory bandwidth (high throughput)

The maximum data transfer for the whole GPU over a single clock cycle of the SPs would be:

900 GB/s / 1.53 GHz = 588 Bytes per cycle (per the entire GPU!)

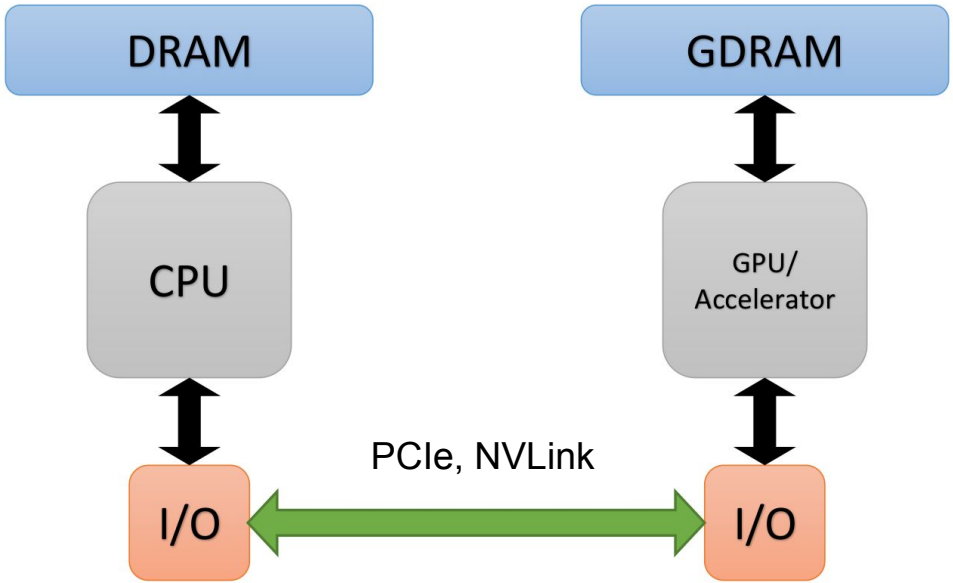
Per single CUDA core:

588 Bytes per cycle / 5120 CUDA cores = 0.11 Bytes per cycle per CUDA core

i.e. ~ 1 Byte processed per every 9 Clock cycles!

GPUs don’t live in a void

- CPUs and GPUs are used together
  - CPUs for generic, complex, and sequential tasks
  - GPUs for heavily data-parallel parts
- CPU ↔ GPU communication via:
  - PCIe bus
    - PCIe 3.0
      up to 8 GT/s (~16GB/s)
    - PCIe 4.0
      up to 16 GT/s (~32GB/s)
    - PCIe 5.0
      up to 32 GT/s (~64GB/s)
  - NVIDIA NVLink bus
    - NVLink 1.0
      up to 20 GT/s (~80 GB/s)
    - NVLink 2.0
      up to 25 GT/s (~150 GB/s)
    - NVLink 3.0
      up to 50 GT/s (~300 GB/s)



From now on we'll focus on how to program a GPU for parallel processing