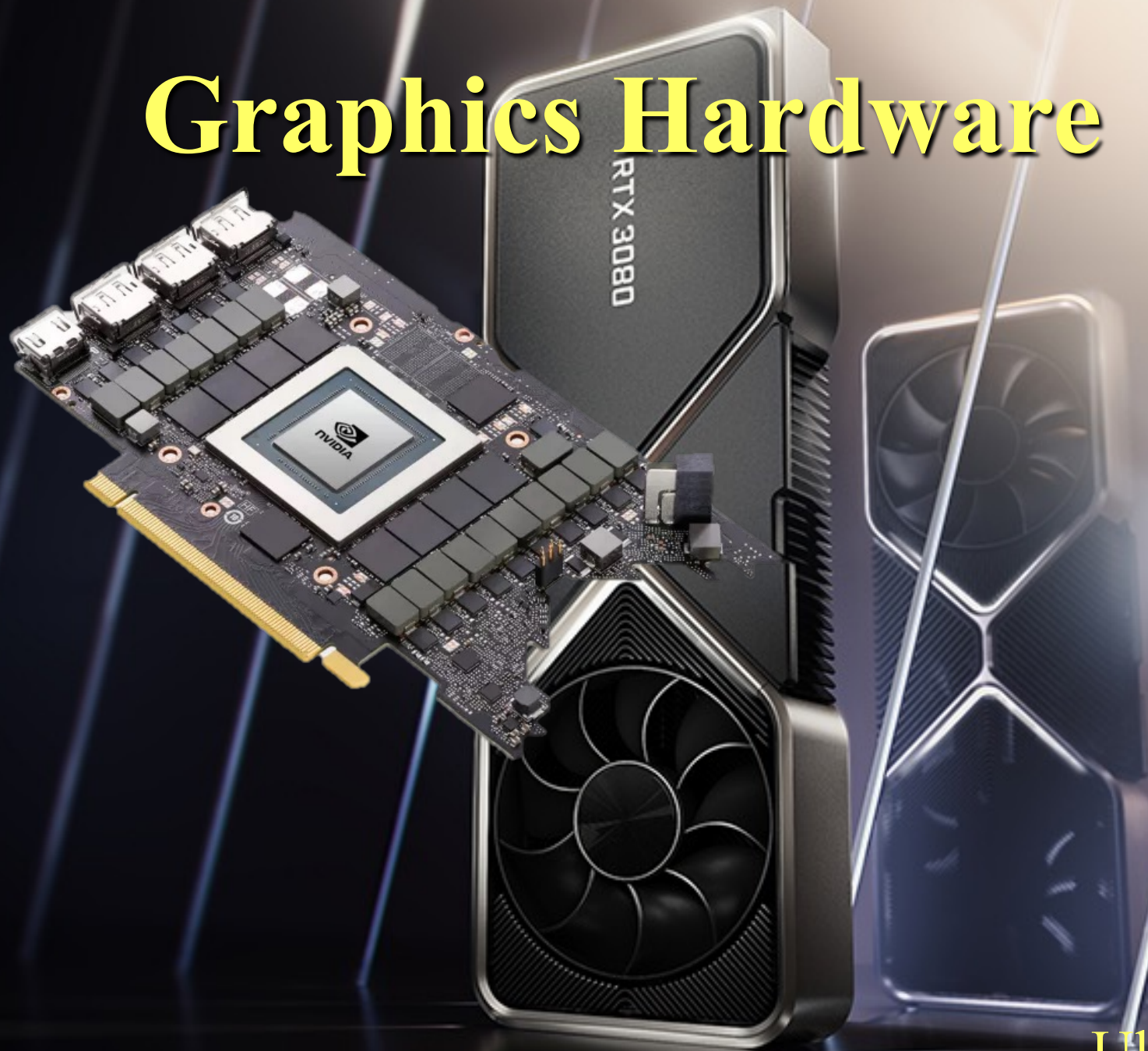


Graphics Hardware



Ulf Assarsson

Graphics hardware – why?

- Often said to be "100x" faster than CPU.
 - Reason: Simple to parallelize triangle rendering :
 - over individual triangles, pixels, (even over x,y,z,w, and r,g,b,a)
 - Hardware fixed functions: clipping, rasterizer, texture filtering, fragment-merge, ...
- Current hardware:
 - Triangle rasterization with programmable shading.
 - Massive parallel general-purpose computations:
 - CUDA/OpenCL/Compute Shaders (~10.000 ALUs)
 - AI computations:
 - ~500 tensor cores, each performing a 4x4-matrix mul+add.
 - GPU Ray tracing:
 - NVIDIA RTX (via OptiX, Vulkan, Microsoft DXR api)
 - Although, can write your own GPU ray-tracer (e.g., CUDA or shader based)
 - or even WebGPU

Perspective-correct interpolation of texture coordinates

(and actually all screen-space-interpolated per-vertex data)



**Steel
Monkeys**

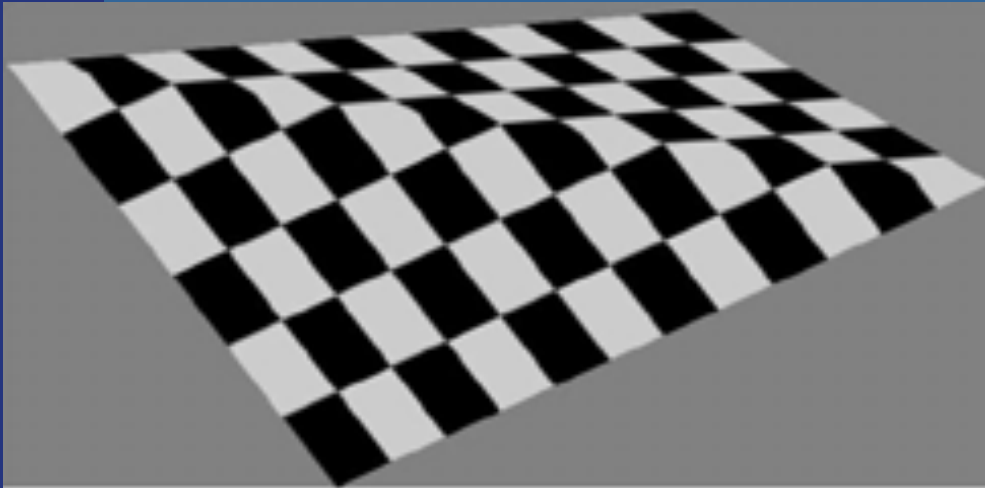


CLASS 1
Game product

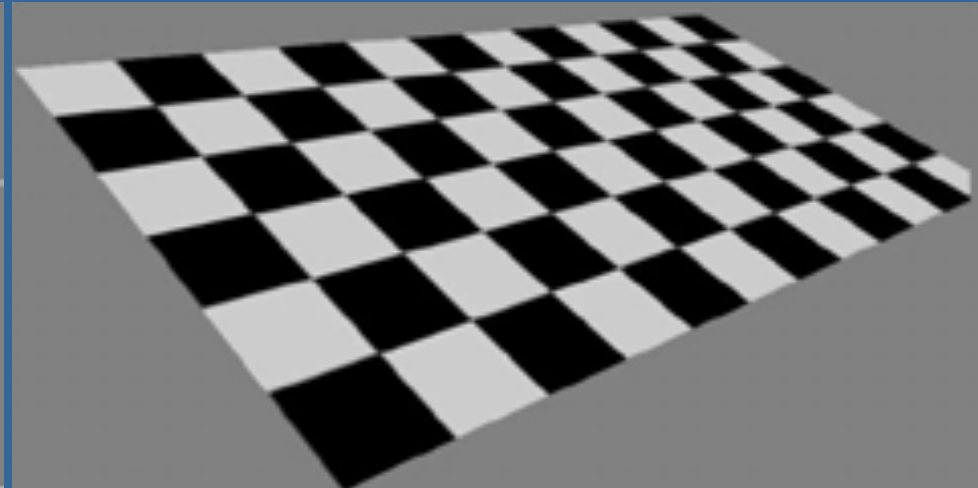
FEEL PORTAL

Perspective-correct texturing

- How is texture coordinates interpolated over a triangle?
- Linearly?



Linear interpolation



Perspective-correct interpolation

- Perspective-correct interpolation gives foreshortening effect!
- Hardware does this for you, but you need to understand this anyway!

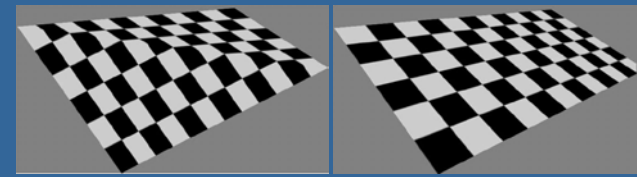
Recall the following

Vertices are projected onto screen by non-linear transform. Hence, tex coords cannot be linearly interpolated in screen space (just like a 3D-position cannot be).

- Perspective projection introduces a non-linear transform by the homogenization step:
 - Projection: $\mathbf{p} = \mathbf{M}\mathbf{v}$
 - After projection p_w is not 1!
 - Homogenization: $(p_x/p_w, p_y/p_w, p_z/p_w, 1)$
 - Gives $(x, y, z, 1)$, where x, y are the screen-space coordinates and z is depth

$$\mathbf{p} = \mathbf{M}\mathbf{v} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ -v_z/d \end{pmatrix}$$

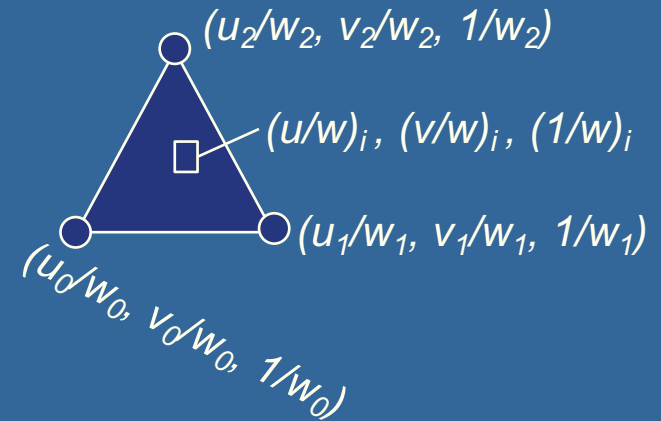
Perspective-correct interpolation



- Linear interpolation in screen space does not work for u, v
- Why:
 - We have applied a non-linear transform to each vertex position $(x/w, y/w, z/w, w/w)$.
 - Non-linear due to $1/w$ – factor from the homogenisation
 - Surprisingly, we can screen-space interpolate any vertex attribute a/w (including $1/w$) perspective correctly.
 - For a proof, see Jim Blinn, "W Pleasure, W Fun", IEEE Computer Graphics and Applications, p78-82, May/June 1998

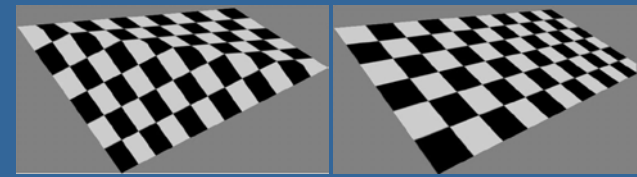
- **Solution:**

- Interpolate $(u/w, v/w, 1/w)$, from each vertex, where w is from homogeneous coordinate (x, y, z, w) . (Screen-space coord is $(x/w, y/w, z/w, 1)$)
 - Then at each pixel, get u_i, v_i as:
 - $w_i = 1 / (1/w)_i$
 - $u_i = (u/w)_i * w_i$
 - $v_i = (v/w)_i * w_i$



Shading is automatically interpolated this way too (though, not as annoying as textures). Perspective correct interpolation nowadays handled automatically by the GPU.

Perspective-correct interpolation



”Intuitive explanation” (but not proof):

- Linear interpolation in screen space does not work for u, v
- Why:
 - We have applied a non-linear transform to each vertex position $(x/w, y/w, z/w, w/w)$.
 - Non-linear due to $1/w$ – factor from the homogenisation
- Solution:
 - We must apply the same non-linear transform to u, v as for x, y, z
 - E.g. $(u/w, v/w)$. This can now be correctly screenspace interpolated since it follows the same non-linear $(1/w)$ transform (and interpolation) as $(x/w, y/w, z/w)$.
 - So, linearly interpolate $(u/w, v/w, 1/w)$, which is computed in screenspace at each vertex.
 - Then at each pixel:
 - $u_i = (u/w)_i / (1/w)_i$
 - $v_i = (v/w)_i / (1/w)_i$

Overview of GPU architecture

- History / evolution
- GPU design: Several **cores** consisting of many **ALUs**
(NVIDIA terminology: **Streaming Multiprocessors (SMs)** of many **cores**)
- GPU vs CPU

Take-away: bandwidth (cost of memory accesses)
is often the main problem

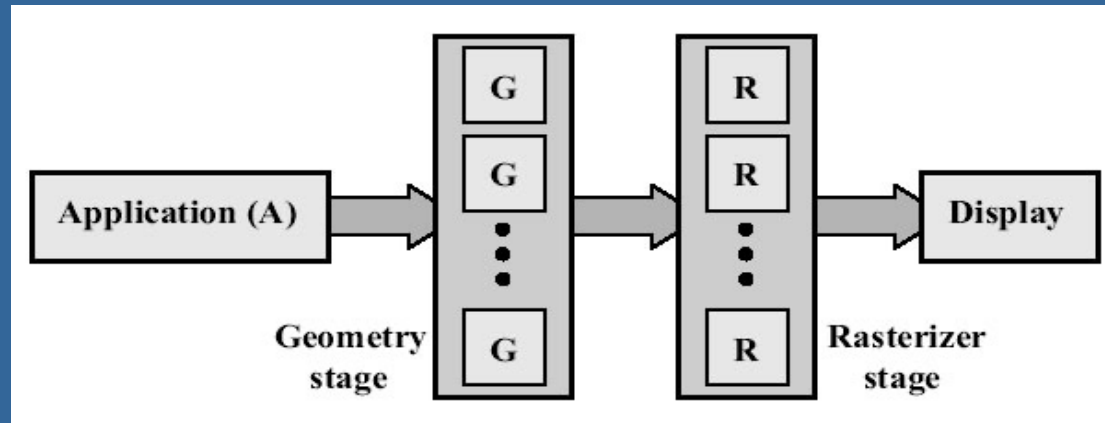
Background:

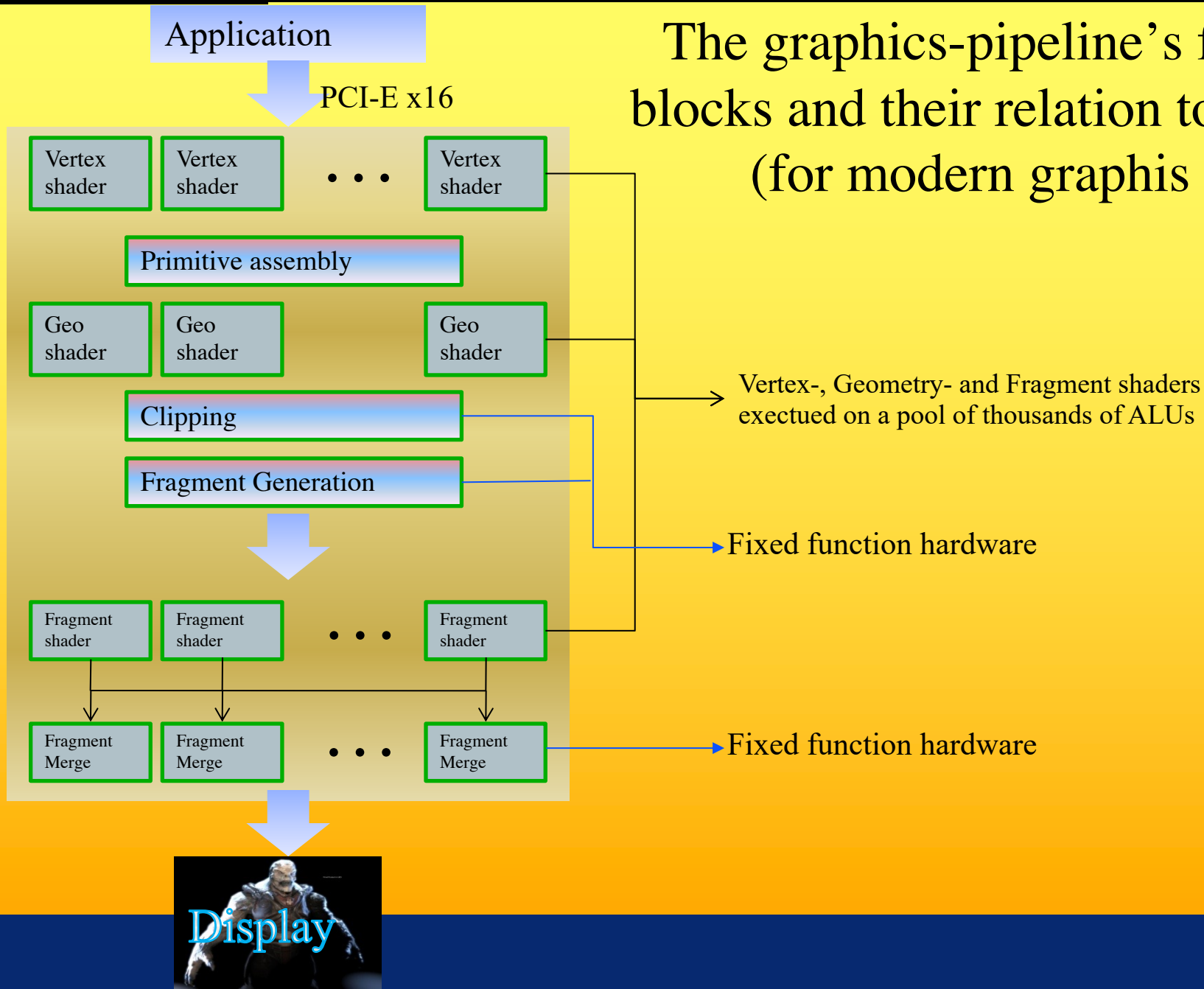
Graphics hardware architectures

- Evolution of graphics hardware has started from the end of the pipeline
 - Rasterizer was put into hardware first (most performance to gain from this)
 - Then the geometry stage
 - Application will not be put into GPU hardware (?)
- Two major ways of getting better performance:
 - Pipelining
 - Parallelization
 - Combinations of these are often used

Parallellism

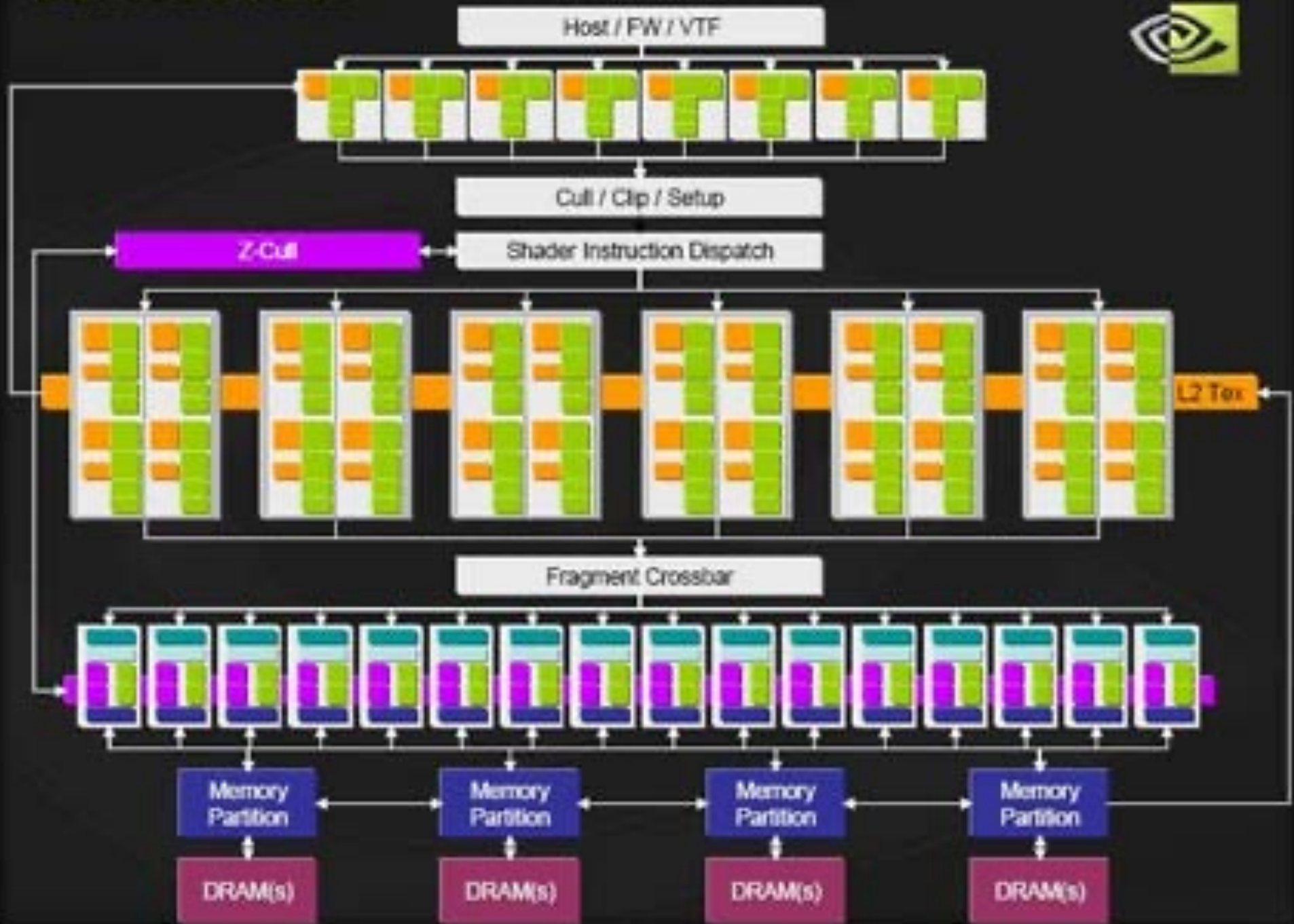
- "Simple" idea: compute n results in parallel, then combine results
- Not always simple
 - Try to parallelize a sorting algorithm...
 - But vertices are independent of each other, and also pixels, so simpler for graphics hardware
- Can parallelize both geometry and rasterizer stage:



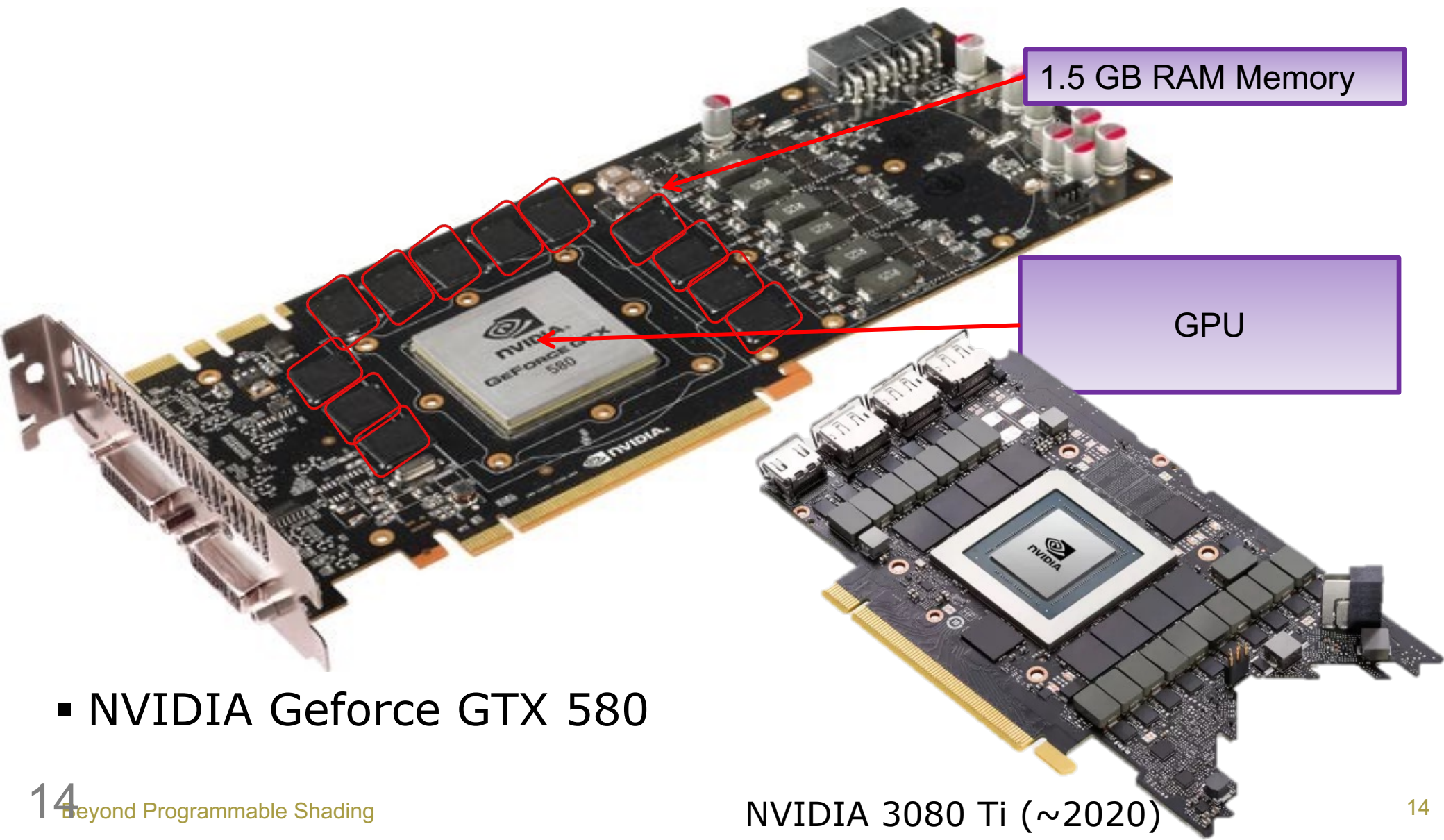


GeForce 7800

Older architecture (2006)



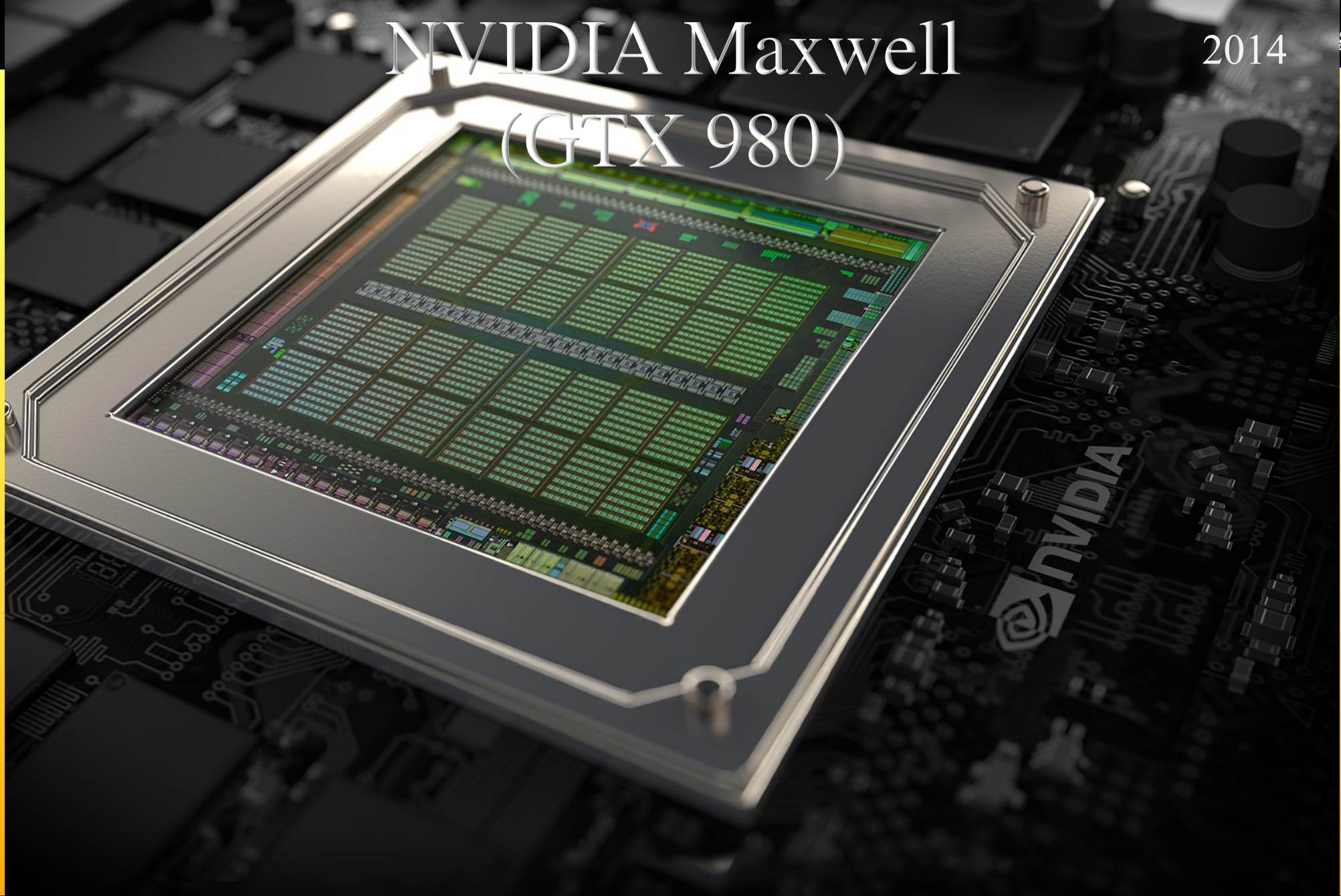
Graphics Processing Unit - GPU

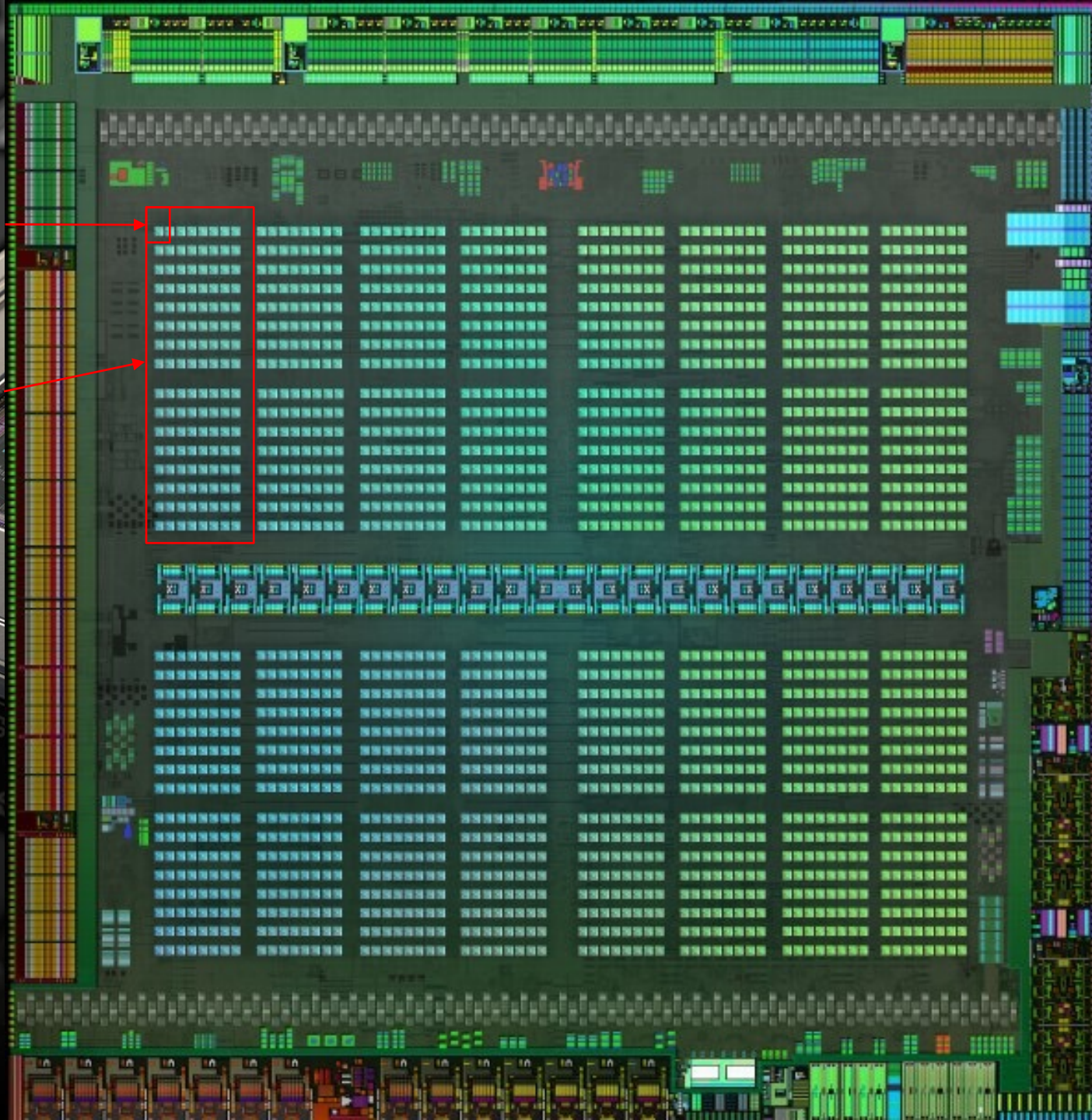


NVIDIA Maxwell (GTX 980)

2014

g





ALU:



SMM
(core):

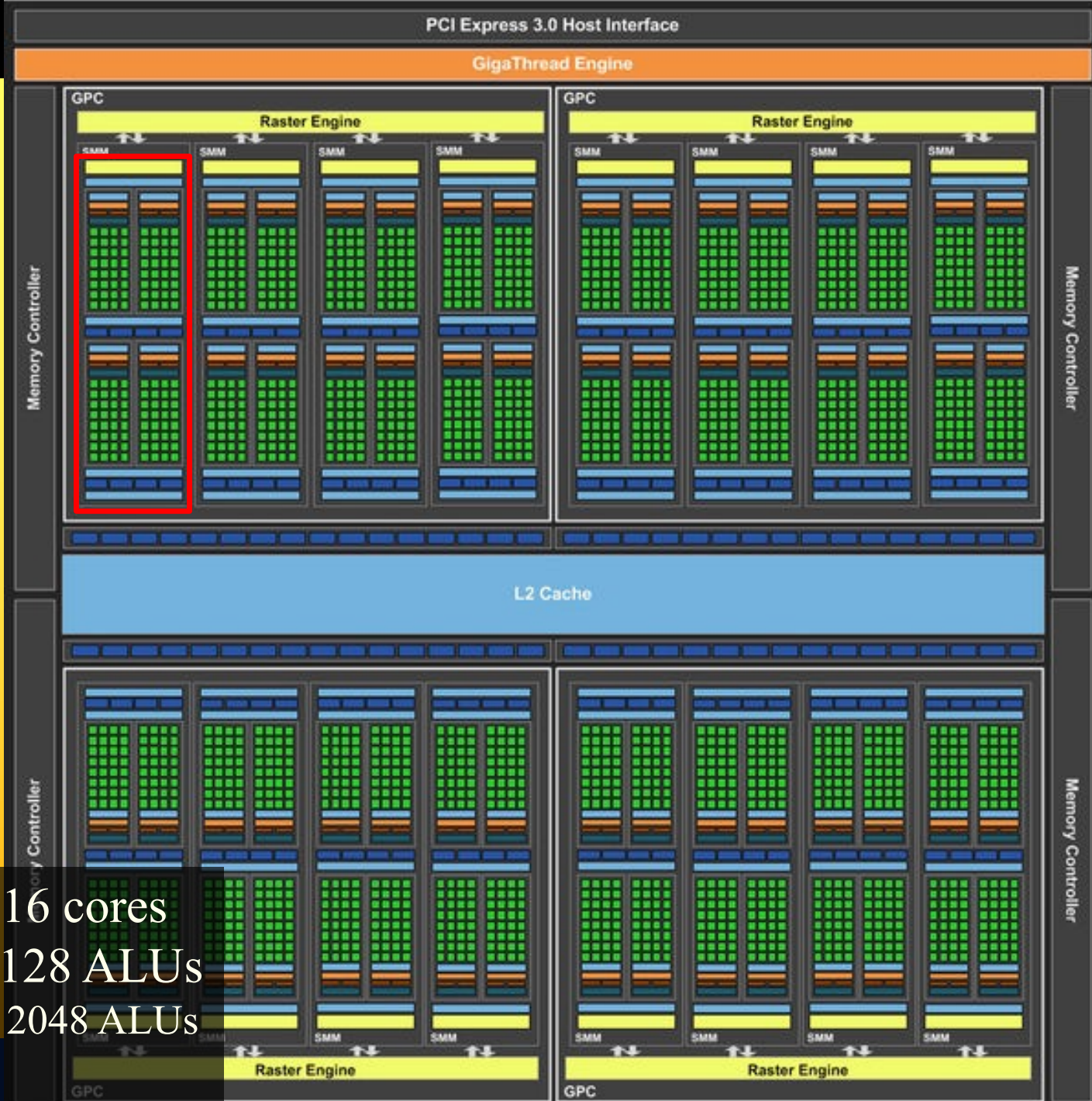


16 Cores (“SMM”)
 2MB L2 cache
 64 output pixels / clock
 (i.e., 64 ROPs)
 2048 ALUs (“cores”)
 ~6 Tflops

Each Core:

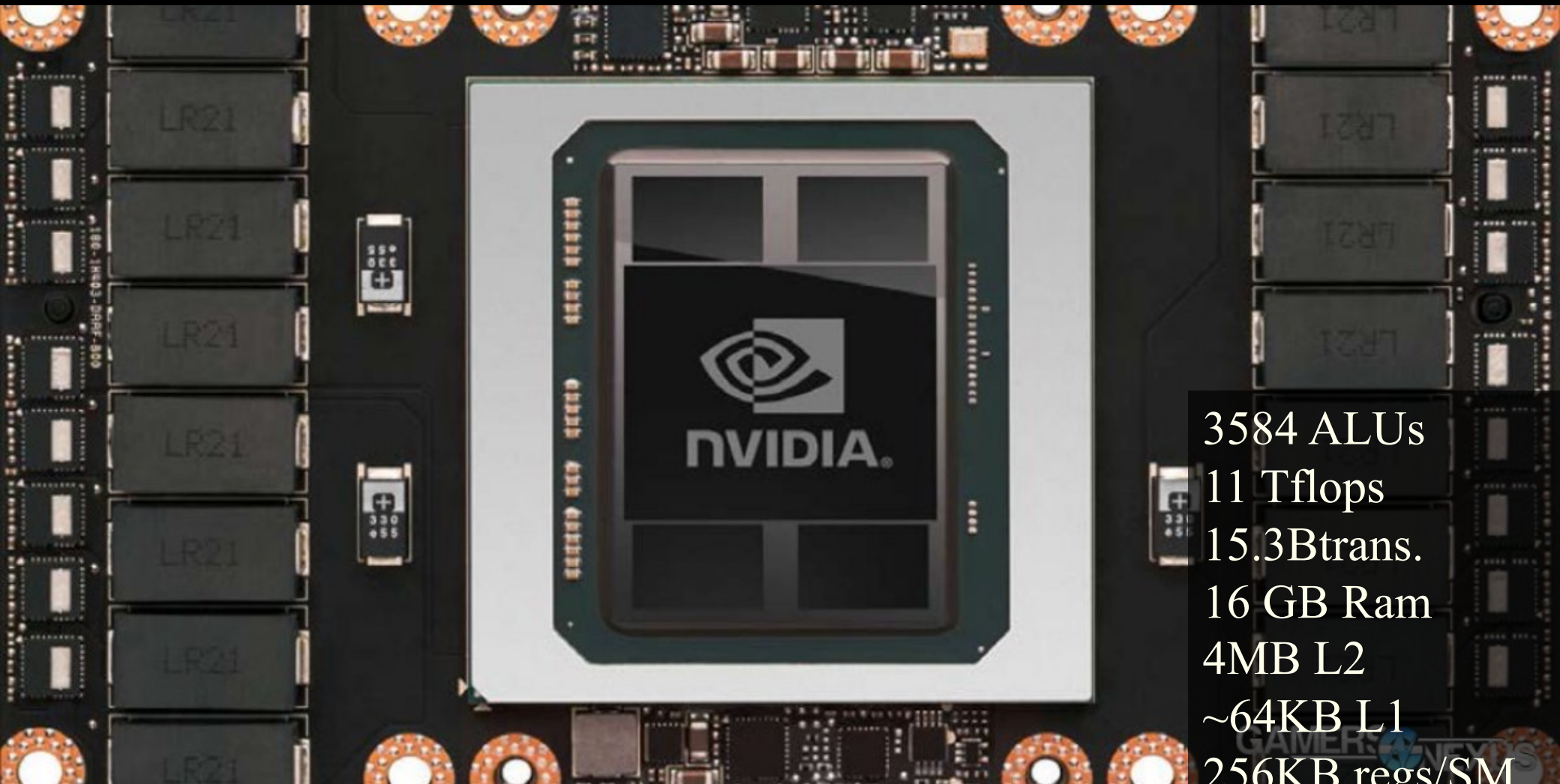
- 128 ALUs
- 96KB L1 cache
- 8 TexUnits
- 32 Load/Store units for access to global memory

GPU: 16 cores
 Core: 128 ALUs
 => 2048 ALUs

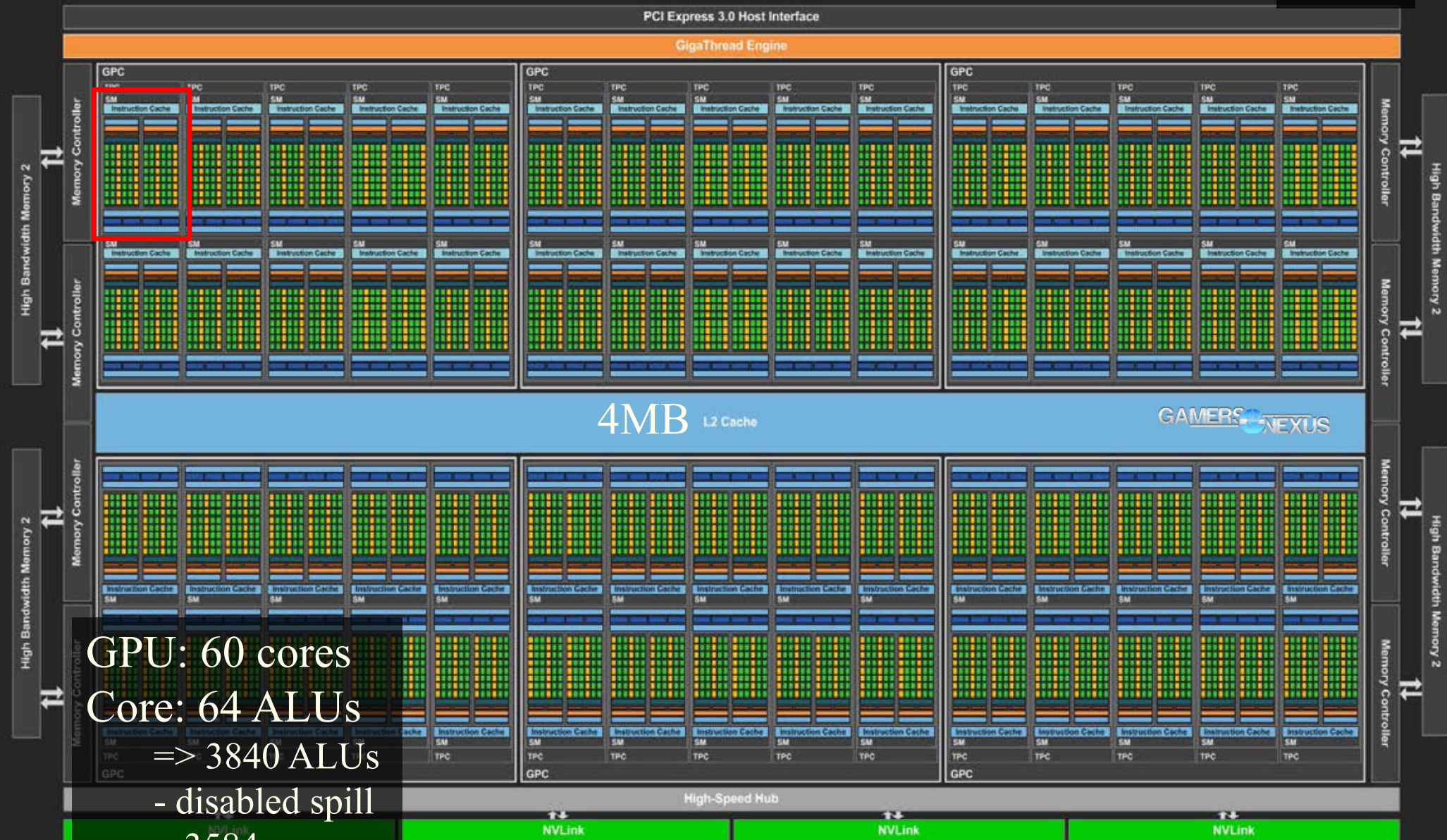


NVIDIA Pascal GP100 (GTX 1080 / Titan X)

2016



3584 ALUs
11 Tflops
15.3Btrans.
16 GB Ram
4MB L2
~64KB L1
256KB regs/SM
224 tex units



Instruction Cache

Instruction Buffer

Warp Scheduler

Dispatch Unit

Dispatch Unit

Register File (32,768 x 32-bit)



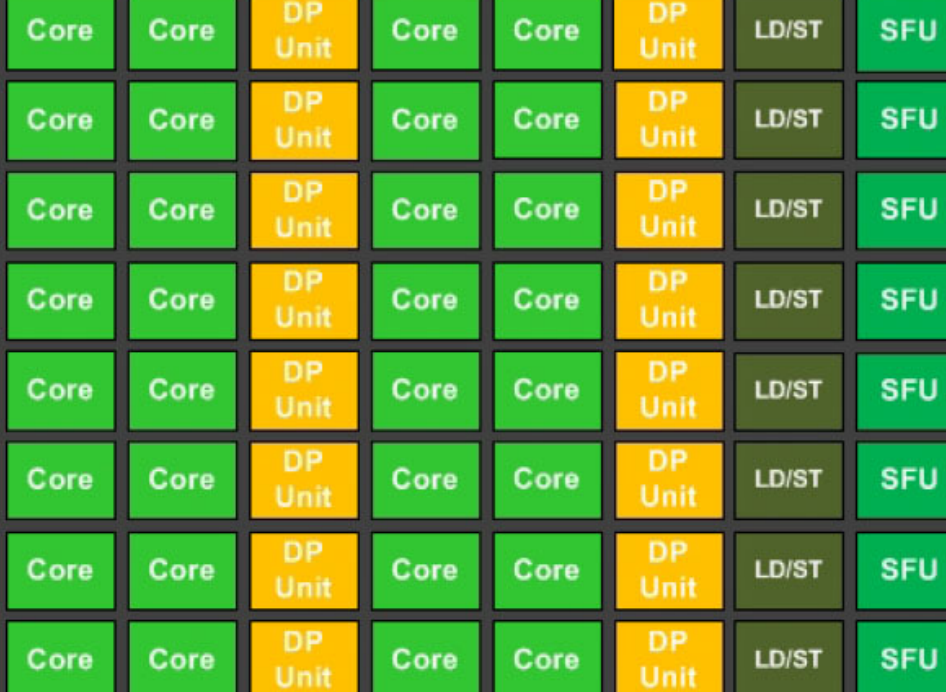
Instruction Buffer

Warp Scheduler

Dispatch Unit

Dispatch Unit

Register File (32,768 x 32-bit)



Texture / L1 Cache

Tex

Tex

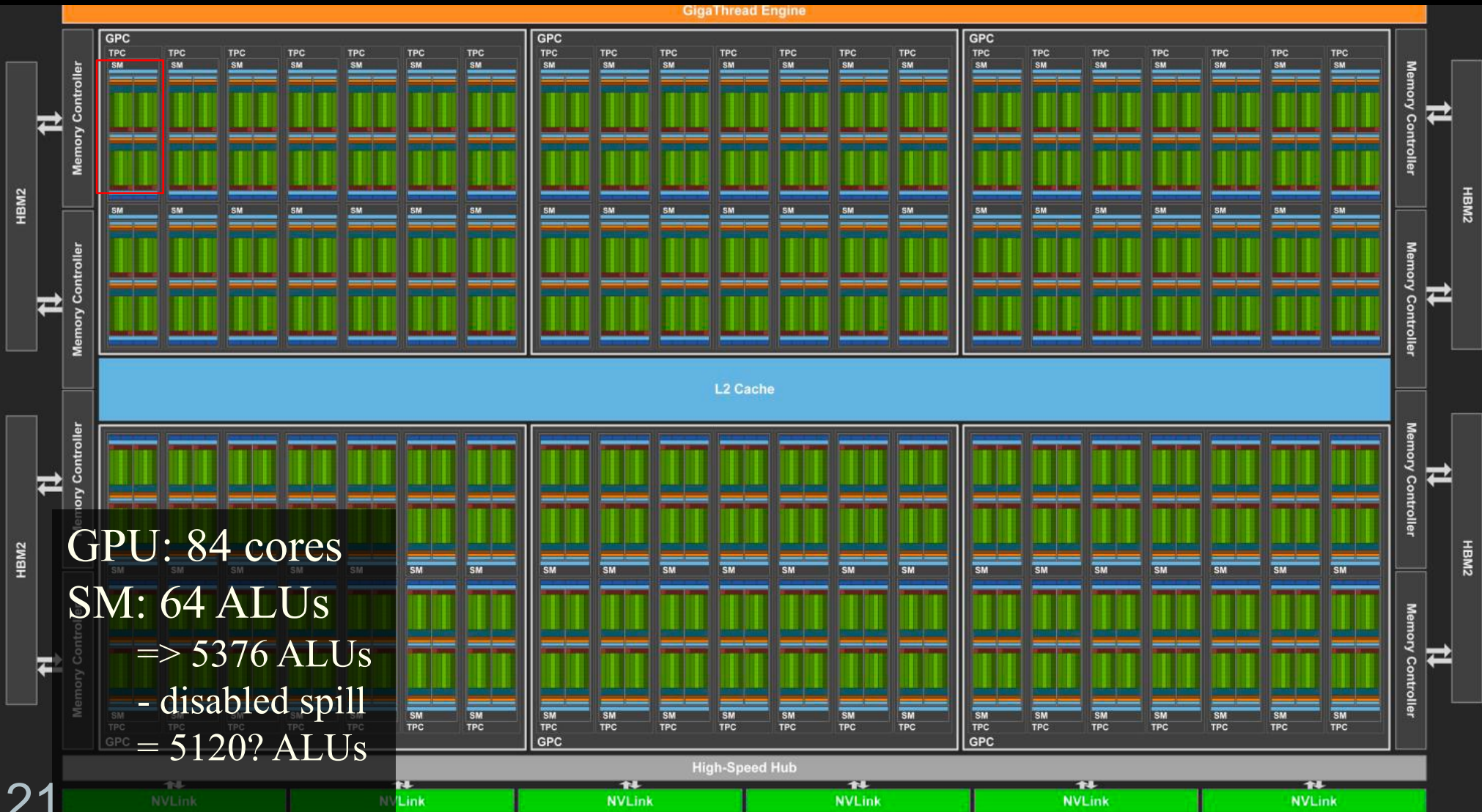
Tex

Tex

64KB Shared Memory

NVIDIA Volta GV100

2018
(Dec.
2017)



NVIDIA Volta GV100

2018

SM



- Core:
- 64 32-bit fp/int ALUs
 - 512 16-bit ALUs

Tensor core
per clock:

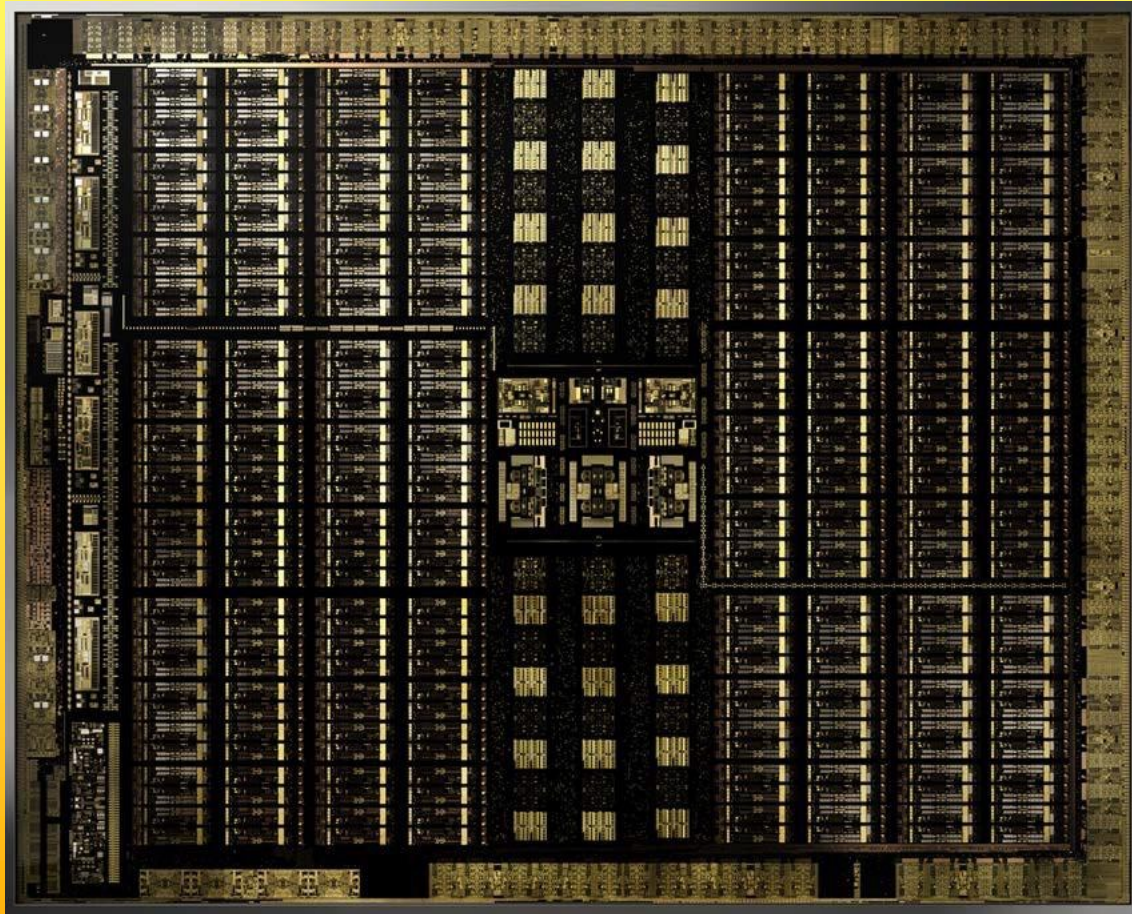
$D =$

$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 or FP32

NVIDIA Turing TU102

2018

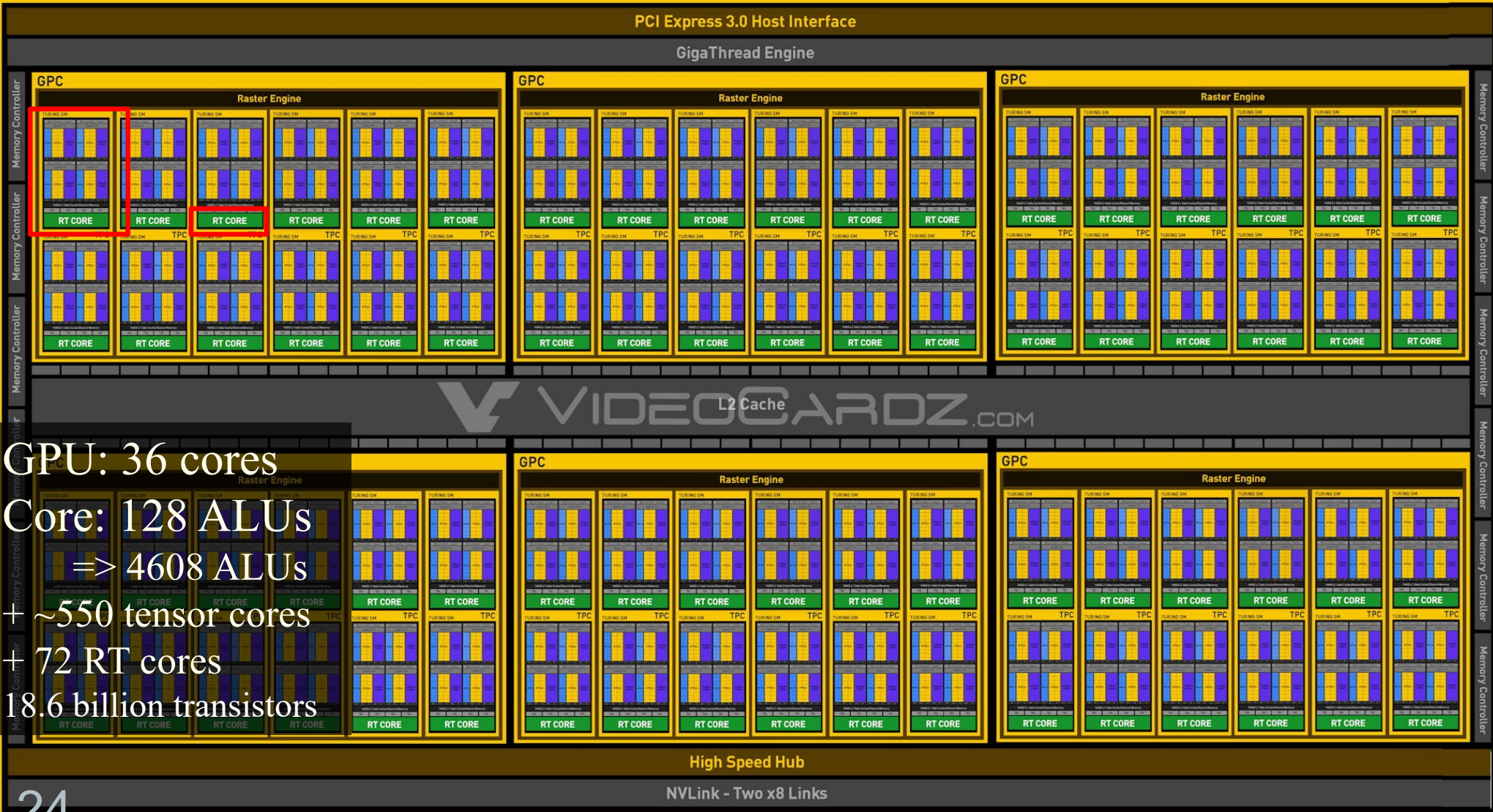


GPU: 36 cores
Core: 128 ALUs
=> 4608 ALUs
+ ~550 tensor cores
+ 72 RT cores
18.6 billion transistors

NVIDIA Turing TU102

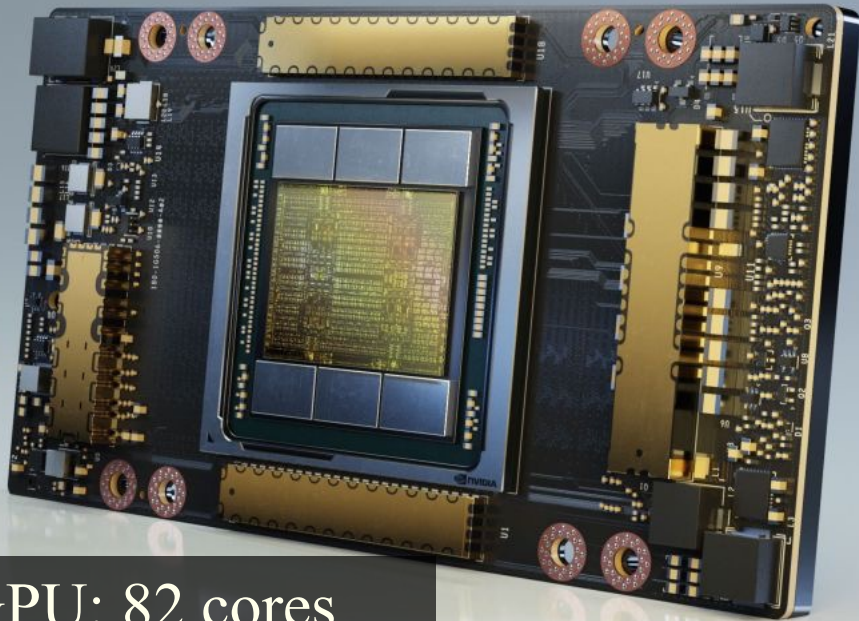
2018

TURING TU102

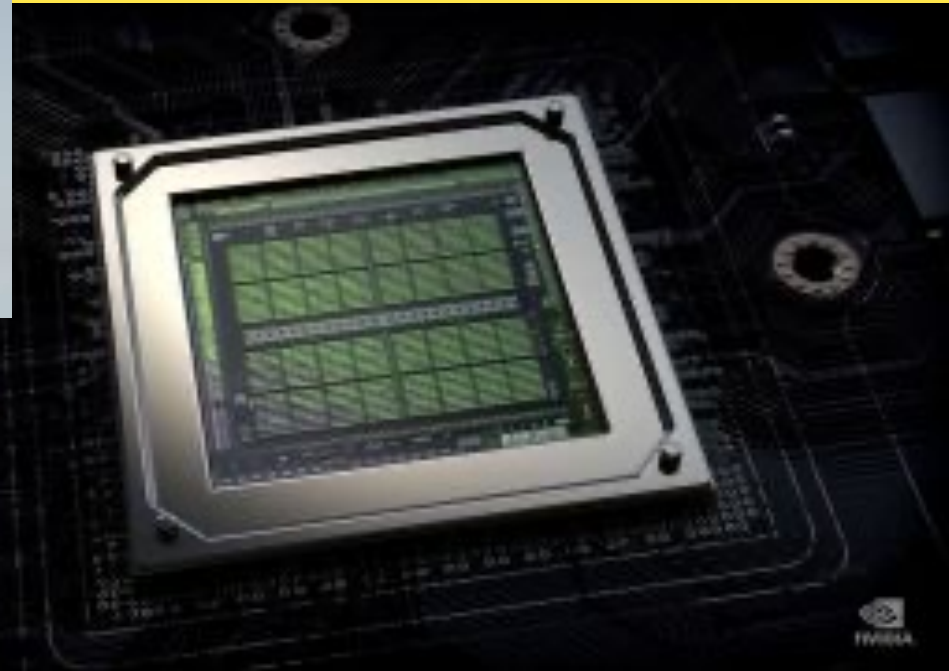


NVIDIA Ampere

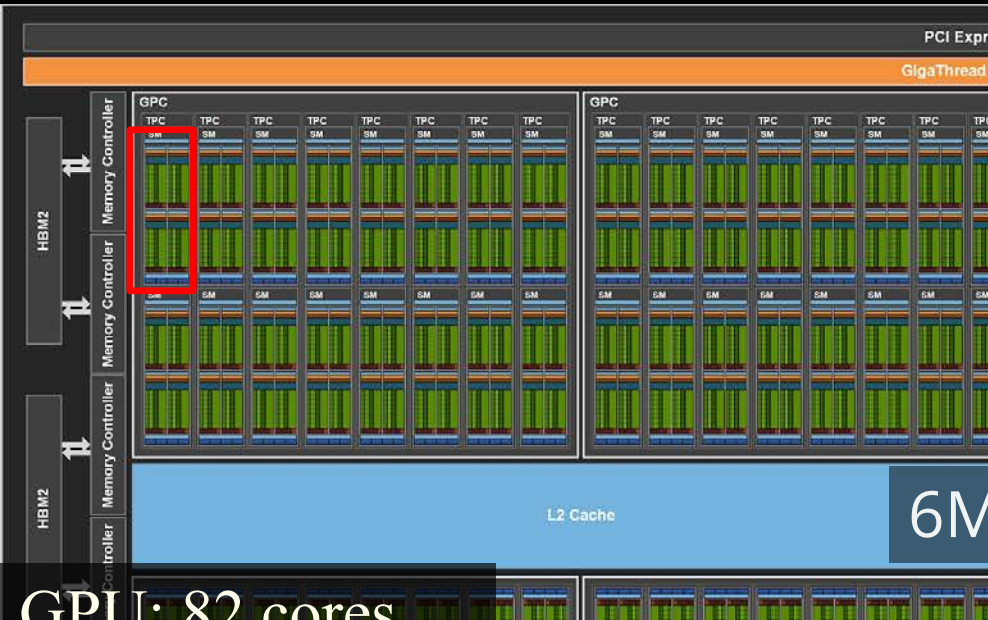
2020



GPU: 82 cores
Core: 128 ALUs
=> 10496 ALUs
+ ~328 tensor cores
+ 82 RT cores
28.3 billion transistors



NVIDIA



GPU: 82 cores
 Core: 128 ALUs
 $\Rightarrow 10496 \text{ ALUs}$
 $\sim 128\text{KB L1\$}$
 + ~ 328 tensor cores
 + 82 RT cores
 28.3 billion transistors

26

SM: 35 Tflops
 Tensor: 143 Tflops:
 RT: 69 Tflops
 Sum: ~ 250 Tflops.

- INT8 allows for 624 TOPS (1248 TOPS with sparsity).
- INT4 doubles that to 1248 / 2496 TOPS.

SM

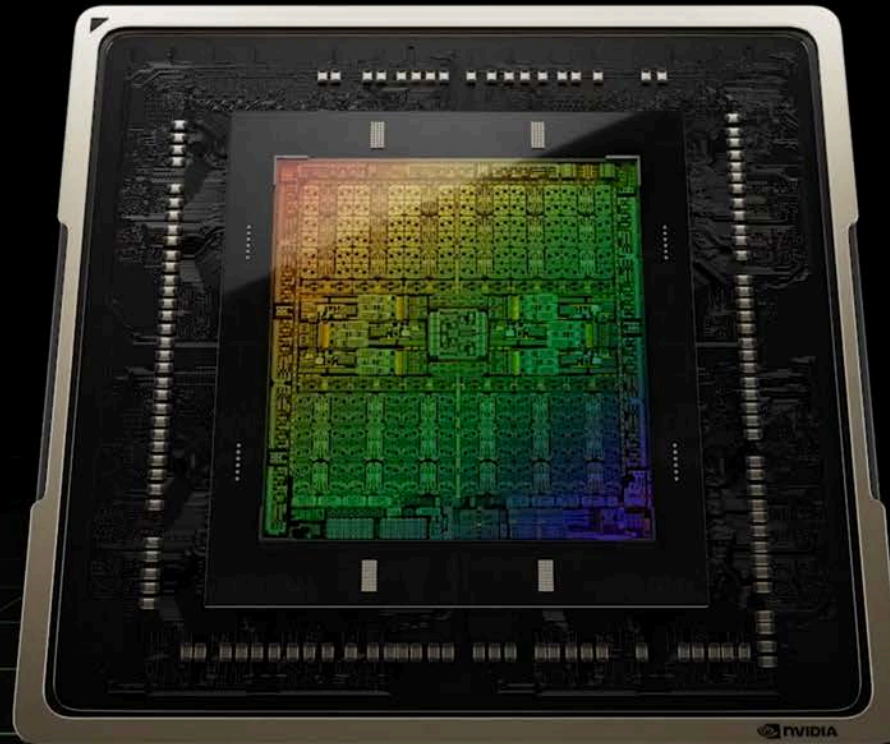


NVIDIA Ada Lovelace

2022

NVIDIA ADA LOVELACE

76 Billion Transistors | TSMC 4N Process | Micron G6X Memory



Shaders

New Streaming Multiprocessor
90 Shader TFLOPs
2X Power Efficiency

Ray Tracing

3rd-Gen RT Cores
200 RT TFLOPs
2X Ray-Triangle Intersection

Deep Learning

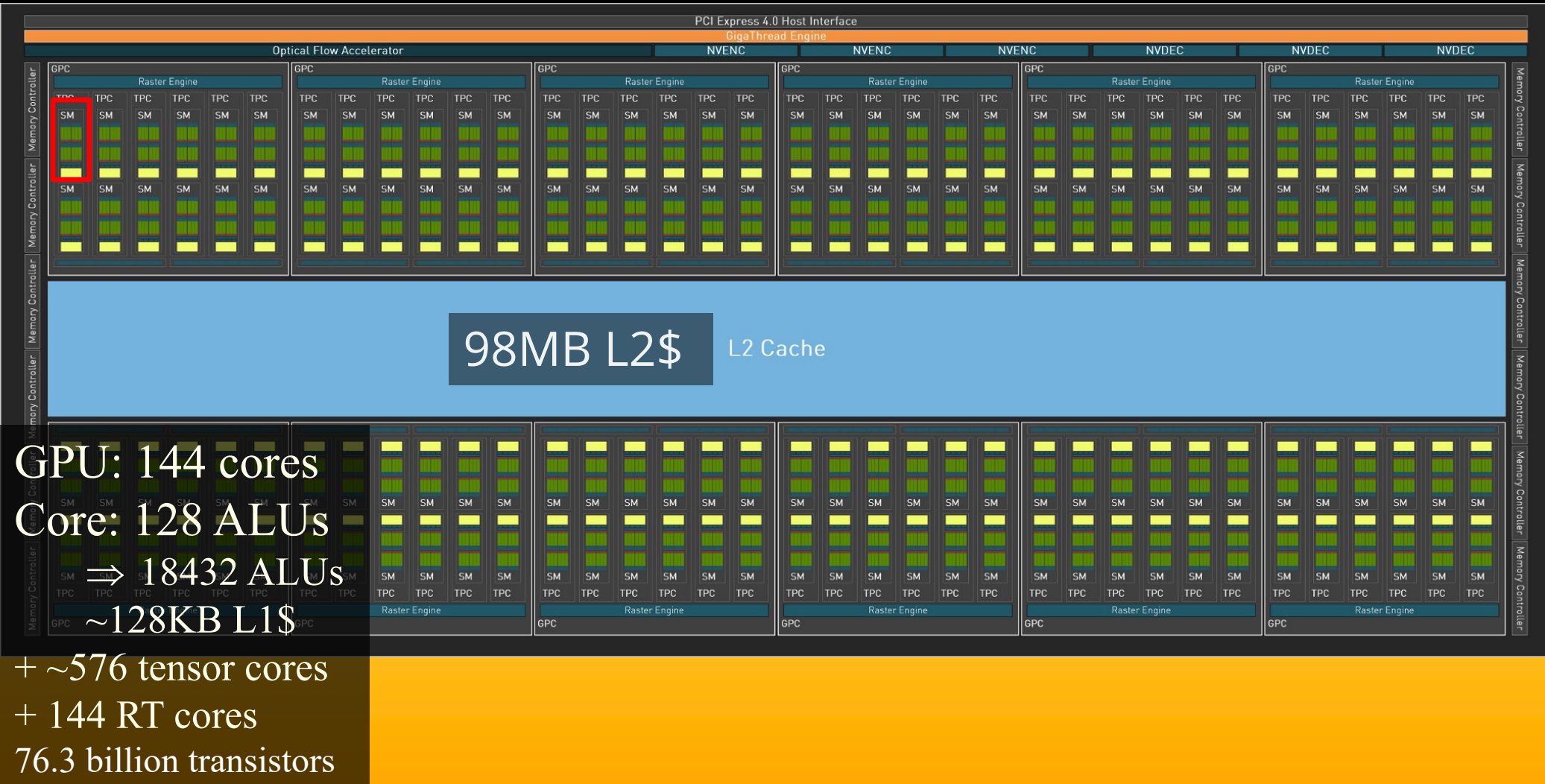
4th-Gen Tensor Cores
1,400 Tensor TFLOPs
Optical Flow Accelerator

GPU: 144 cores
Core: 128 ALUs
=> 18432 ALUs
+ ~576 tensor cores
+ 144 RT cores
76.3 billion transistors



NVIDIA Ada Lovelace

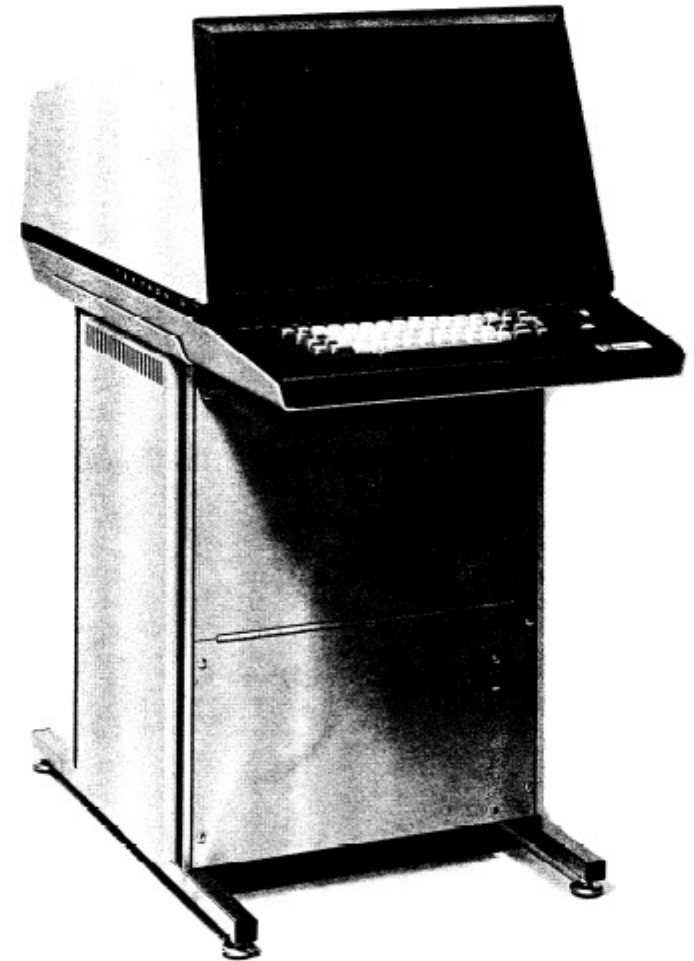
2020



Graphics Hardware History

Direct View Storage Tube:

- Created by Tektronix (early 70's)
 - First with "frame buffer" (moveto/lineto)
 - Did not require constant refresh
 - Standard interface to computers
 - Allowed for standard software
 - Plot3D in Fortran
 - Relatively inexpensive
 - Opened door to use of computer graphics for CAD community
 - 4096 * 4096 addressable points (4096 * 3120 viewable).



Tektronix **4014**

Graphics Hardware History - functionality

- 80's:
 - linear interpolation of color over a scanline
 - Vector graphics
- 91' Super Nintendo, Neo Geo,
 - Rasterization of 1 single 3D rectangle per frame (FZero)
- 95-96': Playstation 1, 3dfx Voodoo 1
 - Rasterization of whole triangles (Voodoo 2, 1998)
- 99' Geforce (256)
 - Transforms and Lighting (geometry stage)
- 02' 3DLabs WildCat Viper, P10
 - Pixel shaders, integers,
- 02' ATI Radion 9700, GeforceFX
 - Vertex shaders and **Pixel shaders** with floats
- 06' Geforce 8800
 - Geometry shaders, integers and floats, logical operations
- Then: — More general multiprocessor systems, higher SIMD-width, more cores
- 09' Tessellation Shaders (Direct3D '09, OpenGL '10)
- 17' Tensor cores
- 18' RT cores, Mesh Shaders



Graphics Hardware History - specs

E.g., since 2004:

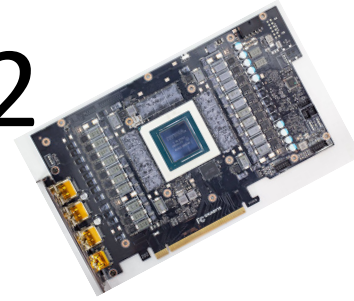
- #trans: ~200x increase
- Bandwidth: ~30x
- Clock freq: ~2x

- 2001 – GeForce3: 600-800 pipeline stages! 57 million transistors.
- First Pentium IV: 20 stages, 42 million transistors,
- 2004 – X800 – 165M transistors
- 2005 – X1800 – 320M trans, 625 MHz, 750 Mhz mem, 10Gpixels/s, 1.25G verts/s
- 2004 – GeForce 6800: 222 M transistors, 400 MHz, 400 MHz core/550 MHz mem
- 2005 – GeForce 7800: 302M trans, 13Gpix/s, 1.1Gverts/s, 430 MHz core, mem bw 54GB/s, 650MHz(1.3GHz)
- 2006 – GeForce 8800: 681M trans, 39.2Gpix/s, 10.6Gverts/s, 612/1500 MHz core, 103.7 GB/s, 1080/2160GDDR3
- 2008 – Geforce 280 GTX: 1.4G trans, 65nm, 602/1296 MHz core, 142GB/s, 1107(*2)MHz mem, 48Gtex/s
- 2007 – ATI Radeon HD 5870: 2.15G trans, 40nm, 850 MHz, 153GB/s, GDDR5, 256bit mem bus,
- 2010 – Geforce GTX480: 3Gtrans, 700/1401 MHz core, Mem 177.4GB/s, 1.848G(*2)GHz, 384bit mem bus, 40Gtex/s
- 2011 – GTX580: 3Gtrans, 772/1544, Mem: 192.4GB/s, 2004/4008 MHz, GDDR5, 384bit mem bus, 49.4 Gtex/s
- 2012 – GTX680: 3.5Gtrans (7.1 for Tesla), 1006/1058, 192.2GB/s, 6GHz GDDR5, 256-bit mem bus.
- 2013 – GTX780: 7.1G, core clock: 837MHz, 336 GB/s, Mem clock: 6GHz GDDR5, 384-bit mem bus
- 2014 – GTX980: 7.1G?, core clock: ~1200MHz, 224GB/s, Mem clock: 7GHz GDDR5, 256-bit mem bus
- 2015 – GTX Titan X: 8Gtrans, core clock: ~1000MHz, 336GB/s, Mem clock: 7GHz GDDR5, 384-bit mem bus
- 2016 – Titan X: 12/15Gtrans, core clock: ~1500MHz, 480GB/s, Mem clock: 10Gbps GDDR5X, 4096-HBM2
- 2018 – Nvidia Volta: 21.1Gtrans, core clock: ~1500MHz, 600GB/s, Mem: 14Gbps 4096-bit HBM2 (or GDDR6)
- 2020 – Nvidia Ampere: 54 Gtrans, ~1500MHz, 1008 GB/s, Mem: 21Gbps, 4096-bit HBM2, (or GDDR6)
- 2022 – Nvidia Ada Lovelace: 76 Gtrans, 2.2-2.5 GHz, 1008 GB/s, Mem: 21Gbps, GDDR6X, 384-bit mem bus

Lesson learned: #trans doubles ~per 2 years. Core clock increases slowly. Mem clock –increases with new technology DDR2, DDR3, GDDR5/6, HBM2, GDDR6X.

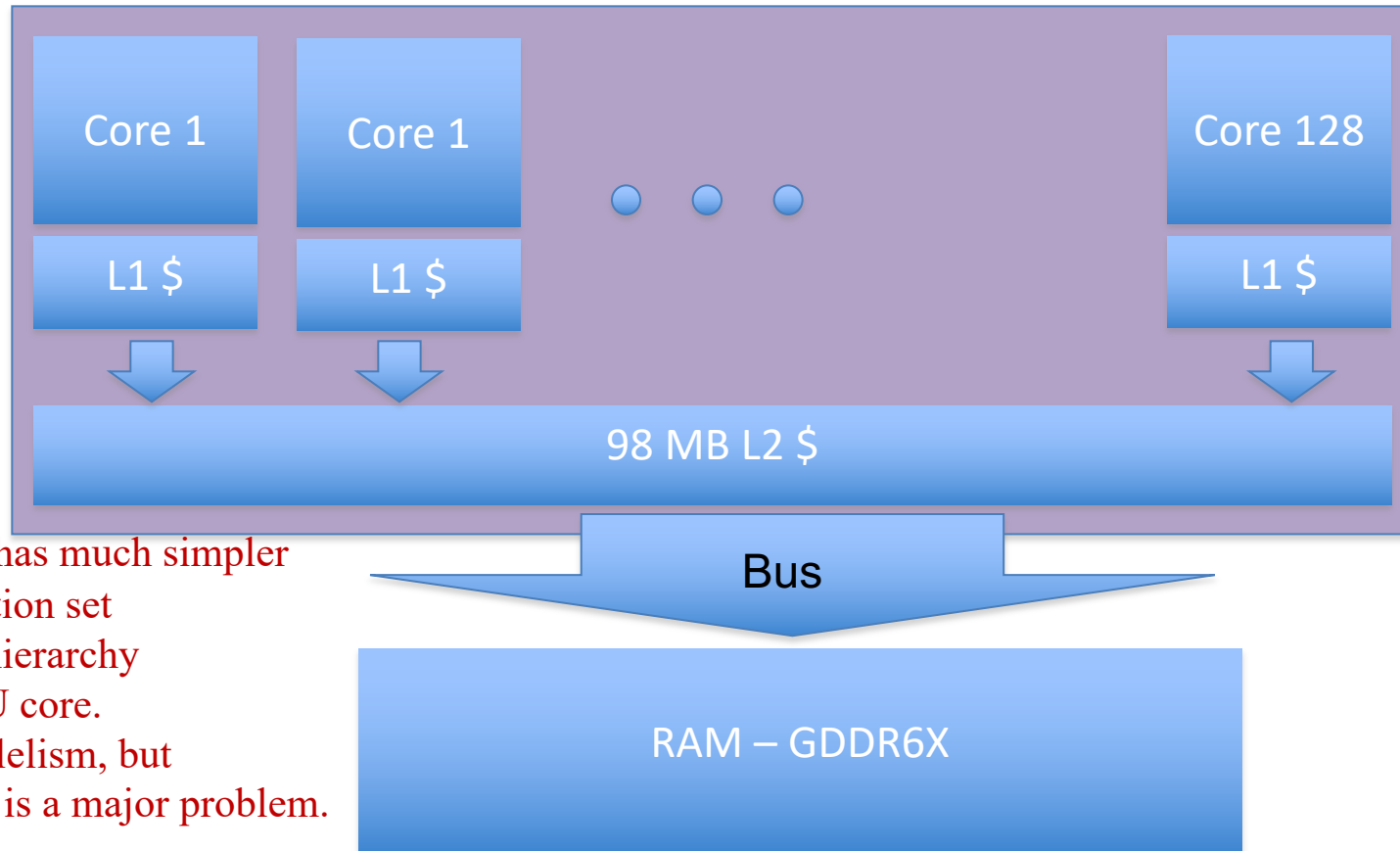
- We want as fast memory as possible! Why?
 - Parallelization can cover for slow core clock. Parallelization more energy efficient than high clock frequency; power consumption proportional to freq^2 .
 - Memory transfers are often the bottleneck

GPU- Nvidia's Ada Lovelace 2022



RTX 4090:

Overview:



128 cores à
128 ALUs

~128 KB L1\$ per
core

Bandwidth
~1 TB/s

Bus: 384 bits

GPU core has much simpler

- instruction set
- cache hierarchy

than a CPU core.

High parallelism, but
bandwidth is a major problem.

Wish:

~16.384 ALUs à 1 float.op/clock => 64KB/clock cycle

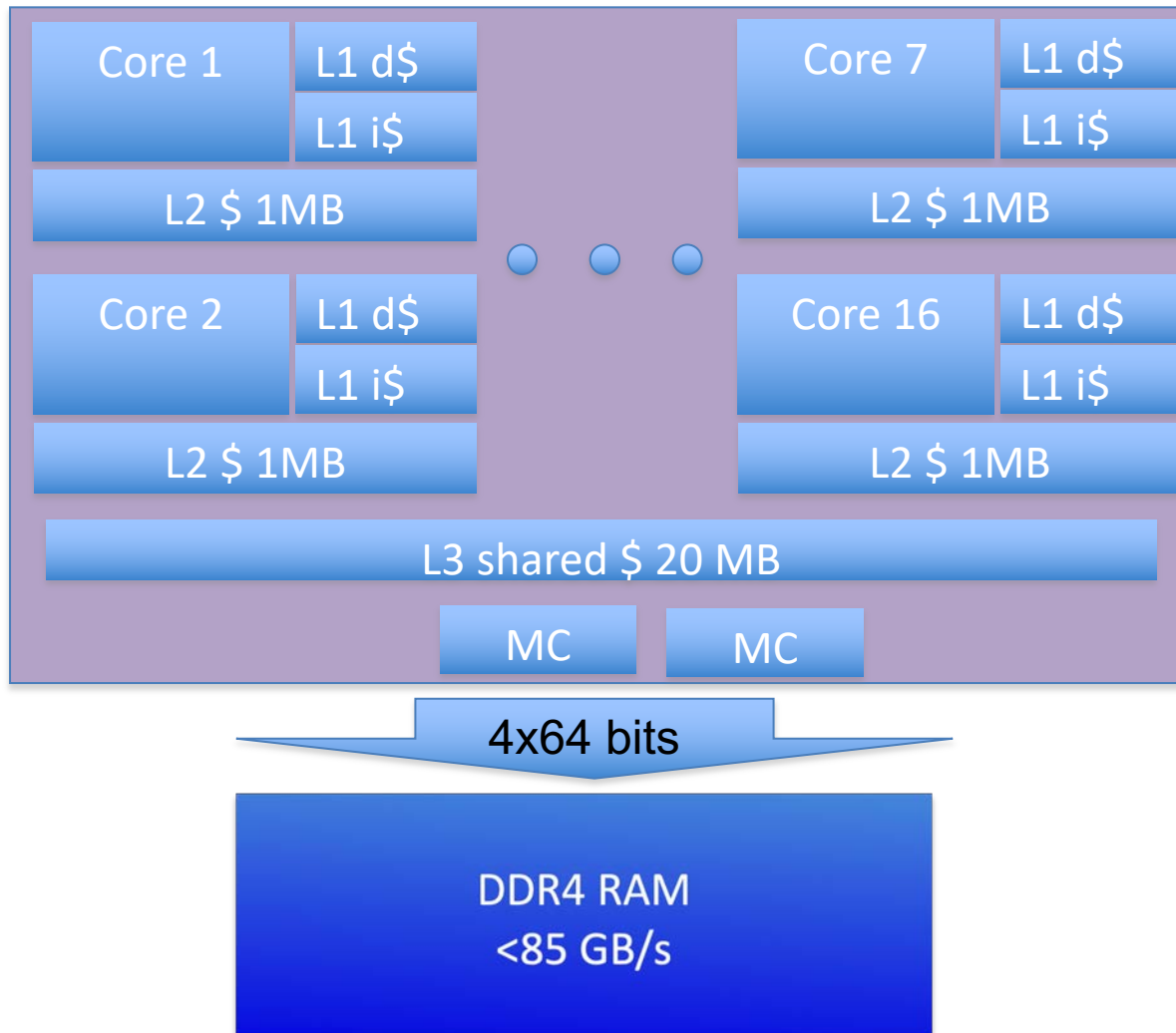
~2.2GHz core clock => 144 TB/s request

We have ~1TB/s. Hence, would need to do ~144 computations between each RAM-read/write.

Ameliorated by L1\$ + L2\$ + latency hiding (warp switching) but still a main problem!

CPU – 2021

Roughly Intel i9



64 KB

64 KB

~16 cores à
8 SIMD floats

- With 16 cores à 8 floats
⇒ We want **512** bytes/clock
(e.g. from RAM).
- 3GHz CPU => **1.5** TByte/s.
(In addition x2, both for GPU
& CPU, since:
 $r1 = r2 + r3;$)

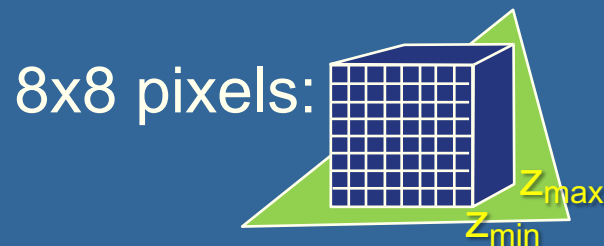
We only have **85** GB/s due to
cheaper DDR4 RAM.

Solved by \$-hierarchy +
registers + thread switching

- Wish: GPU 144TB/s vs CPU 1.5 TB/s \approx 100x diff.
- You could say bandwidth is 2 orders of magnitude more important on GPU than CPU, due to parallelism.

Memory bandwidth usage is huge!!

- On top of that bandwidth usage is never 100%.
- However, there are many techniques to reduce bandwidth usage:
 - Texture caching with prefetching
 - Texture compression
 - Hierarchical Z-occlusion testing
 - E.g., for every 8x8 pixel block of frame buffer, store its Z_{\min} , Z_{\max} .
 - If triangle is behind pixel block, skip rasterize it.
 - If triangle is in front, skip accessing 8x8 individual z-values.



Taxonomy of hardware design

for how to resynchronize (sort) parallelized work.

Outputs to frame buffers must respect incoming triangle order.

Take-aways: Sort-first, Sort-middle, Sort-Last Fragment,
Sort-Last Image

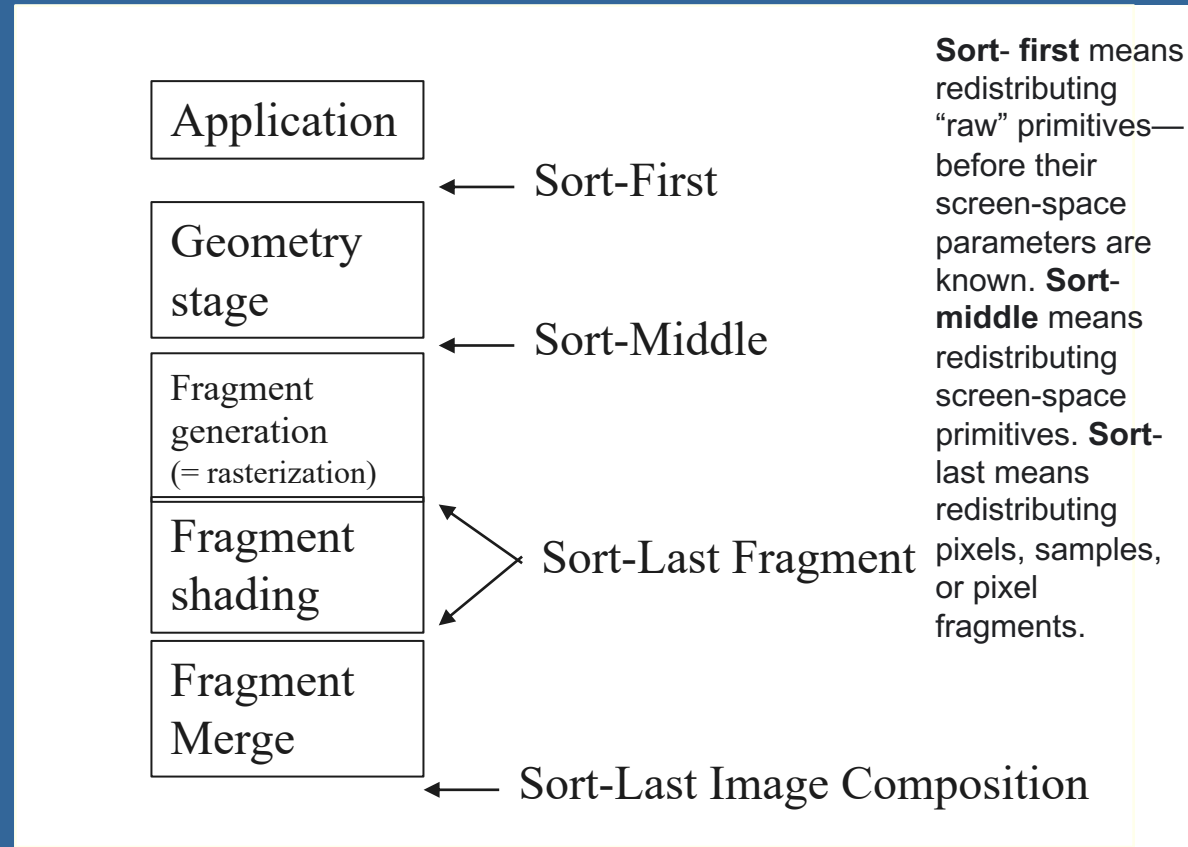
Taxonomy of Hardware

- We can do many computations in parallel:
 - Pixel shading, vertex shading, geometry shading
- But result on screen must be as if each triangle were rendered one by one in their incoming order (according to OpenGL spec)
 - I.e., for every pixel, the rasterized fragments must be merged to the buffers in the original input triangle order
 - E.g., for blending/transparency, (z-culling + stencil test)
- Hence, results need to be sorted somewhere before reaching the screen...

Taxonomy of hardware

- Need to sort the results of the parallelization

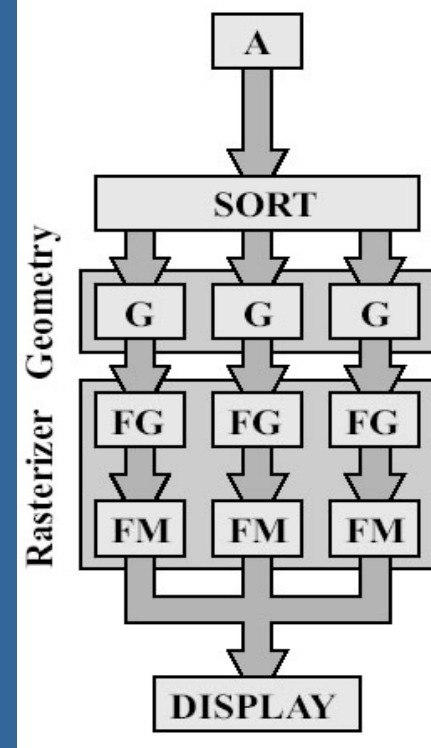
- Gives four major architectures:
 - Sort-first
 - Sort-middle
 - Sort-Last Fragment
 - Sort-Last Image



- Will describe these briefly. Sort-last fragment (and sort middle) are most common in commercial hardware

Sort-First

- Sorts primitives before geometry stage
 - Screen is divided into large regions
 - Blocks or scanlines
 - A separate pipeline is responsible for each region (or many)
- Not explored much at all, since:
 - Poor load balancing if uneven triangle distribution between regions.
 - Vertex shader can change triangle position

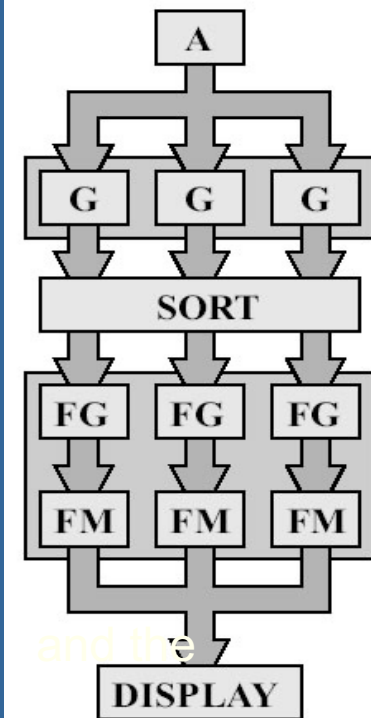


Explanation of image: G is geometry, FG & FM is part of rasterizer (R)

- A fragment is all the generated information for a pixel on a triangle
- FG is Fragment Generation (finds which pixels are inside triangle)
- FM is Fragment Merge (merges the created fragments with various buffers (Z, color))

Sort-Middle

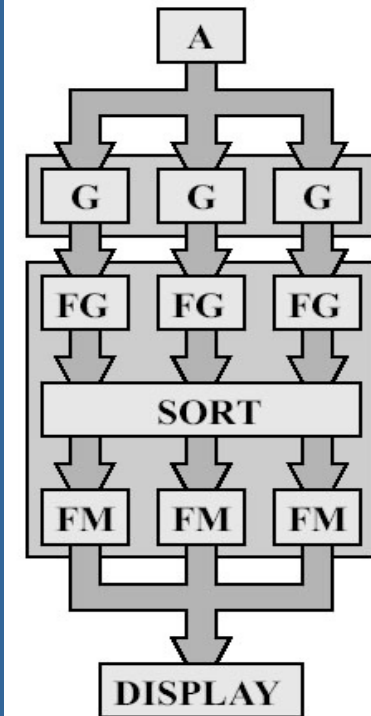
- Sorts between G and R
- Pretty natural, since after G, we know the screen-space positions of the triangles
- Older/cheaper hardware uses this
 - Examples include InfiniteReality (from SGI)
 - KYRO architecture (from Imagination)
- Spread work arbitrarily among G's
- Then depending on screen-space position, sort to different R's
 - Screen can be split into "tiles". For example:
 - Rectangular blocks (8x8 pixels)
 - Every n scanlines
- The R is responsible for rendering inside tile
- Bads (same as Sort-First):
 - A triangle can be sent to many FG's depending on overlap (over tiles)
 - May give poor load balancing if triangles are unevenly distributed over the screen tiles



Sort-Last Fragment

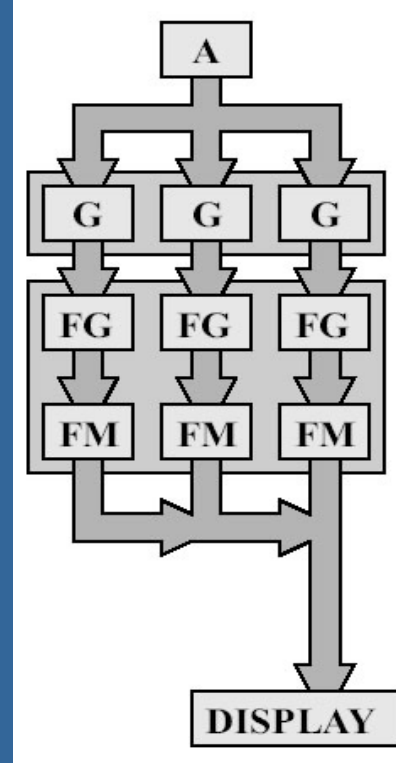


- Sorts between FG and FM
 - After rasterization
- Most graphics cards use this somehow.
 - Each pixel block is responsible for sorting its fragments according to original triangle render order.
 - One typical block size: 4x8 pixels
- Example how it could work:
 - Take pixel block from queue, based on triangle order
 - test hierarchical z-culling
 - Execute shaders
 - Merge
- Good load balancing for all stages before FM.
- Small pixel blocks give good load balancing on screen
- With triangle sizes roughly similar to block sizes, there are not so many more blocks to sort vs sorting triangles in Sort-First and Sort-Middle.



Sort-Last Image

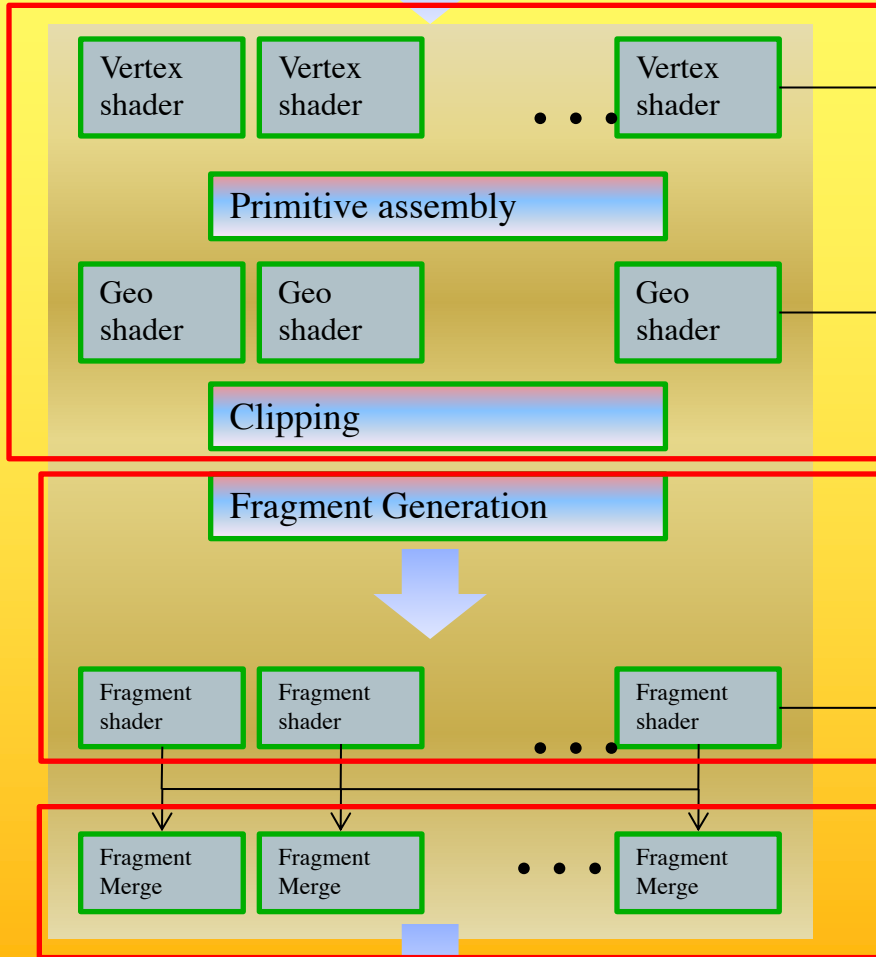
- Sorts after entire pipeline
- So each FG & FM has a separate frame buffer for entire screen (Z and color)
 - Typically: one whole graphics card per pipeline.
- After all primitives have been sent to the pipeline, the z-buffers and color buffers are merged into one color buffer
- Can be seen as a set of independent pipelines
- Huge memory requirements!
- Used in research, but not much commercially.
- Problematic for transparency.



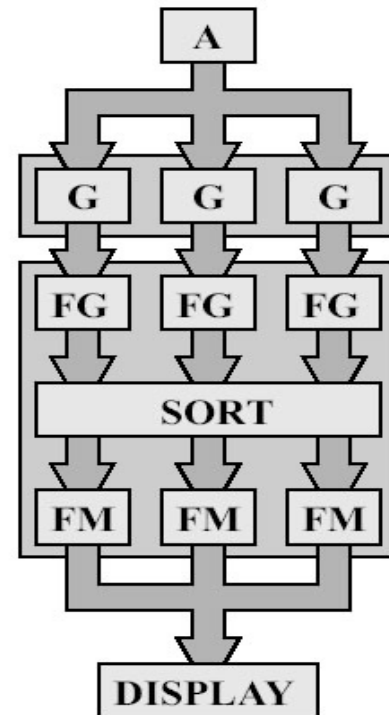
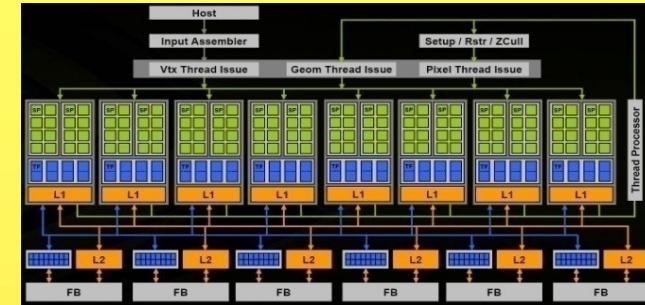
Functional layout of the graphics pipeline and relation to a graphics card:

Application

PCI-E x16



Vertex-, Geometry- and Fragment shaders allocated from a pool of processors (cores and ALUs)



Display

The history implies the future

- Cell – 2005, Sony Playstation 3
 - 8 cores à 4-float SIMD, 256KB L2 cache/core, 3.2 GHz
- NVIDIA 8800 GTX – Nov 2006
 - 16 cores à 8-float SIMD ([GTX 280 - 30 cores à 8, june '08](#))
 - 16 KB L1 cache, 64KB L2 cache
 - 1.2-1.625 GHz
- NVIDIA Fermi GF100 – 2010, (GF110 2011)
 - 16 cores à 2x16-float SIMD (1x16 double SIMD)
 - 16/48 KB L1 cache, 768 KB L2 cache
- NVIDIA Kepler 2012 - 16 cores à 2x3x16=96 float SIMD
- NVIDIA Kepler 2013 - 16 cores à 2x6x16=192 float SIMD
- NVIDIA Titan X 2016 – 60 cores à 2x4x8=64 float SIMD
- NVIDIA Volta 2018 – 84 cores à 64 float SIMD + 640 tensor cores (16-bit matrix mul+add)
- NVIDIA Turing 2018 – 36 cores à 128 float SIMD + ~550 tensor cores (16-bit matrix mul+add) + 72 RT cores
- NVIDIA Ampere 2020 – 82 cores à 128 ALUs + ~328 tensor cores + 82 RT cores
- NVIDIA Ada Lovelace 2022 – 144 cores à 128 ALUs + ~576 tensor cores + 144 RT cores

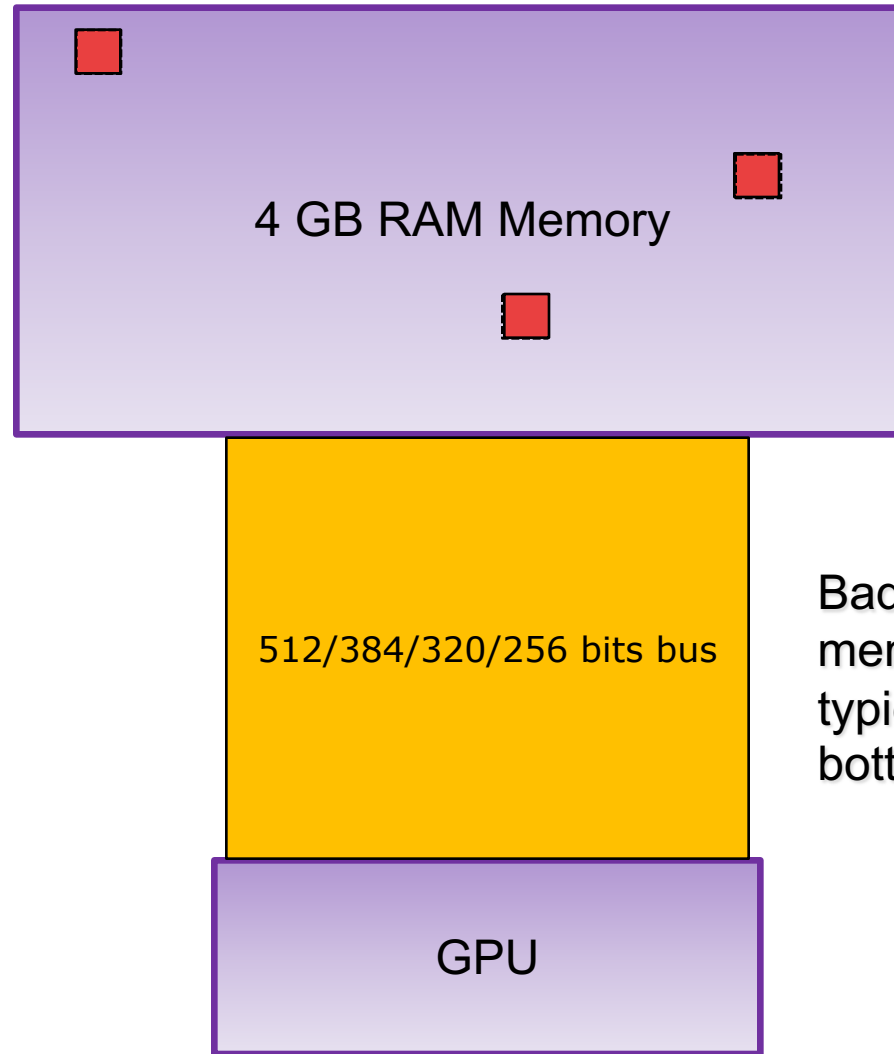
If we have time...

How create efficient GPU
programs?


Answer: coalesced memory
accesses

Graphics Processing Unit - GPU

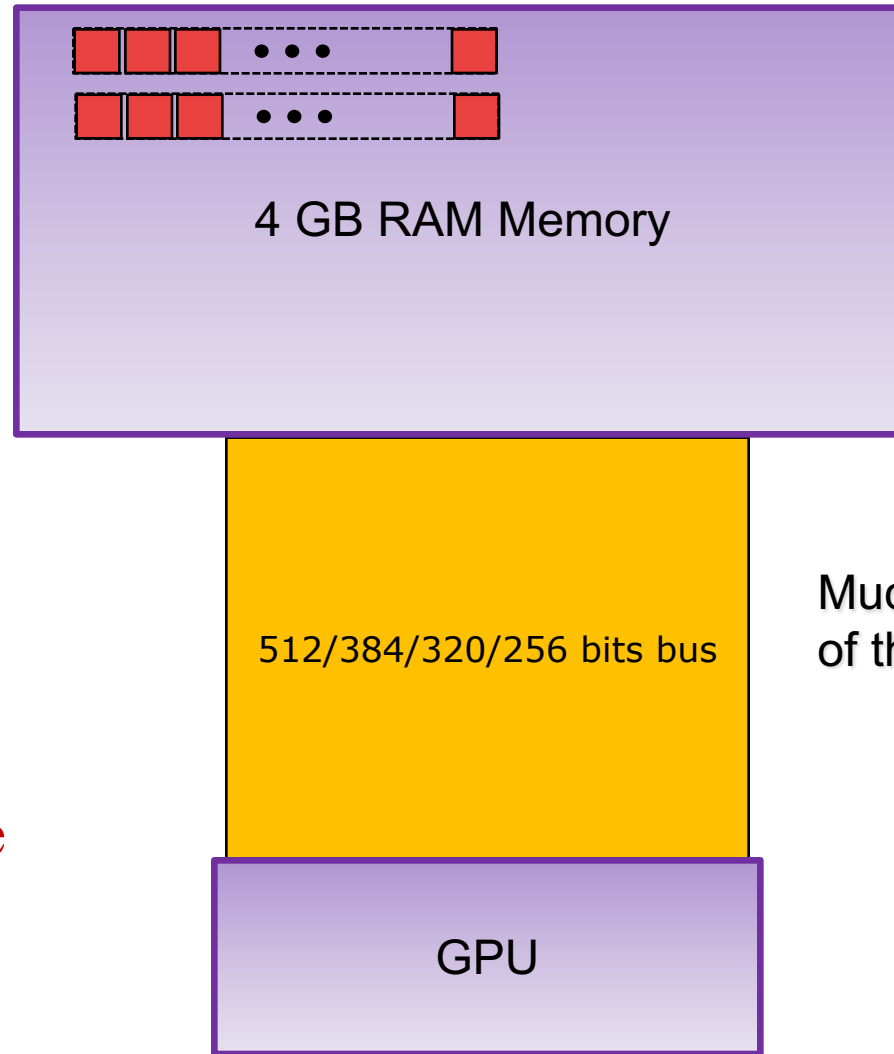
Conceptual layout:



Bad utilization of the memory bus, which typically is the bottleneck!

 = memory element (32 bits)

Graphics Processing Unit - GPU

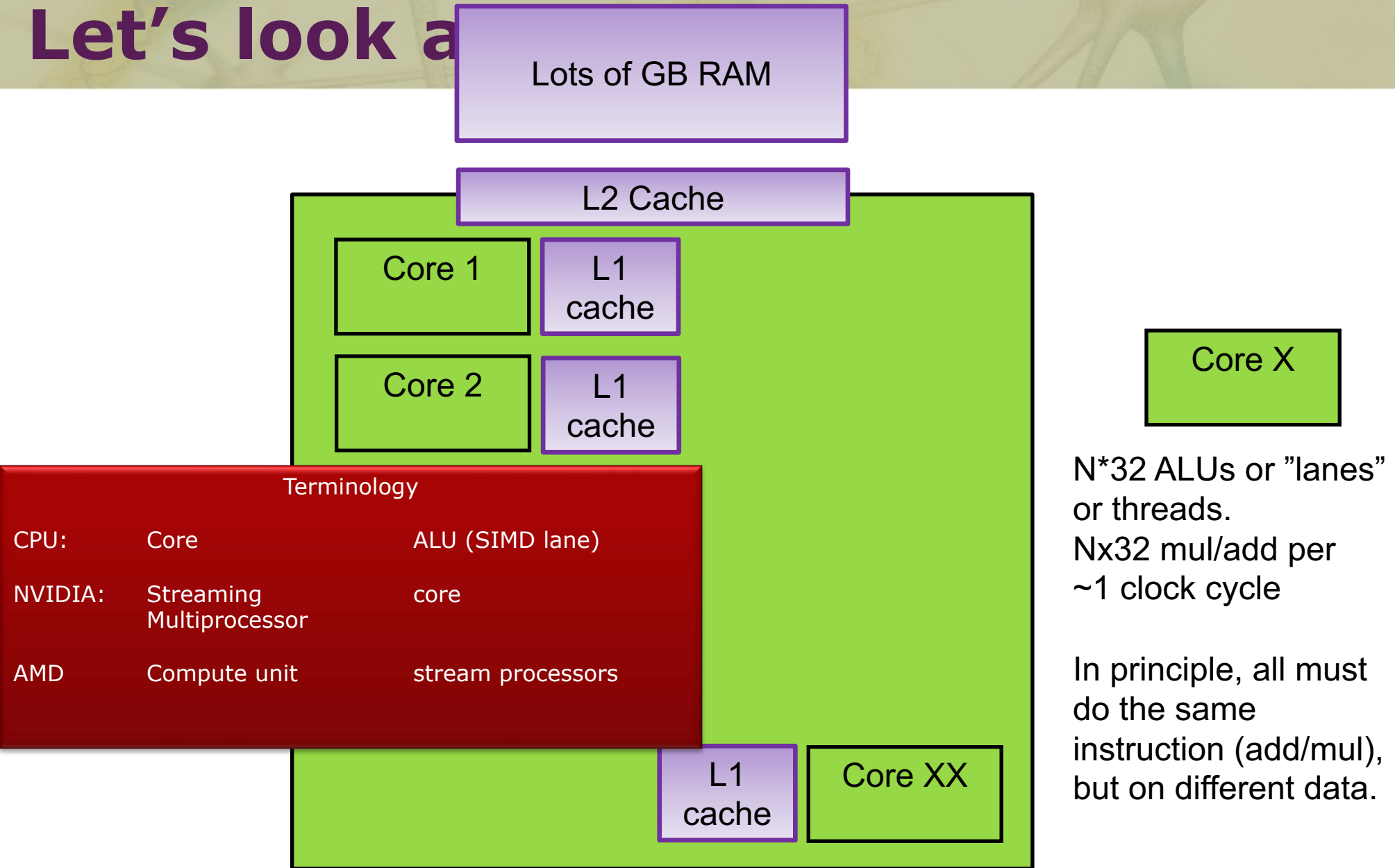


Read 32
coalesced floats
for max
bandwidth usage

Much better utilization
of the memory bus!

■ = memory element (32
bits)

Let's look at



Let's look at the GPU

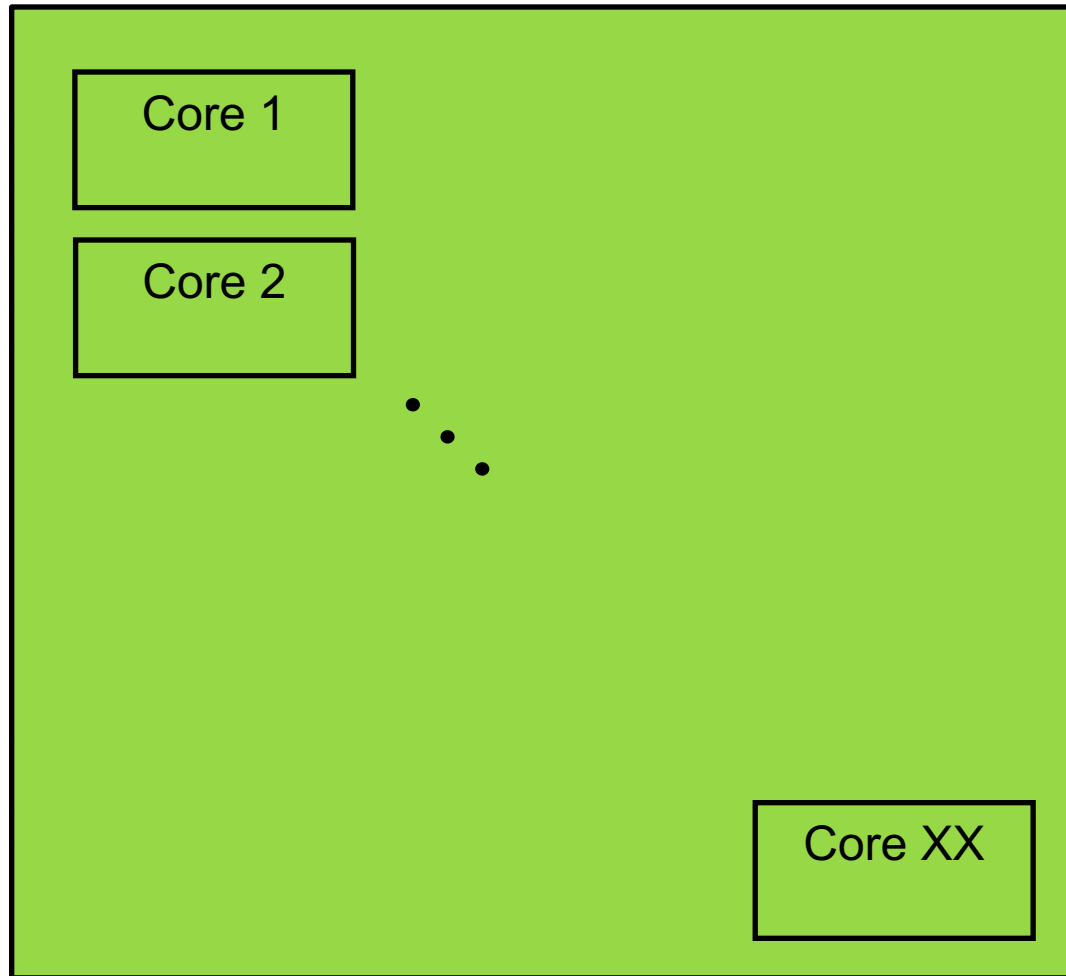
Each core:

- executes one program (=shader).

Each cycle:

- $N \times 32$ flops

These days, can be a few different instr.



From RAM or
L1/L2 cache



32 add/mul etc
in a clock cycle*



To RAM or
L1/L2 cache

Low level APIs for GPU programming

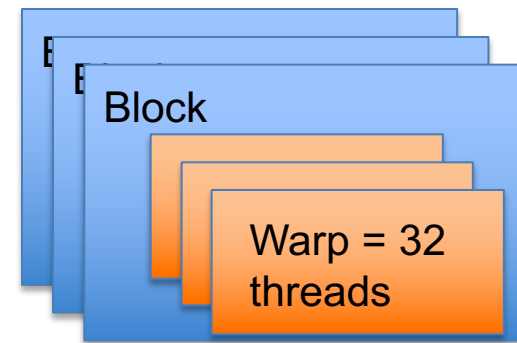
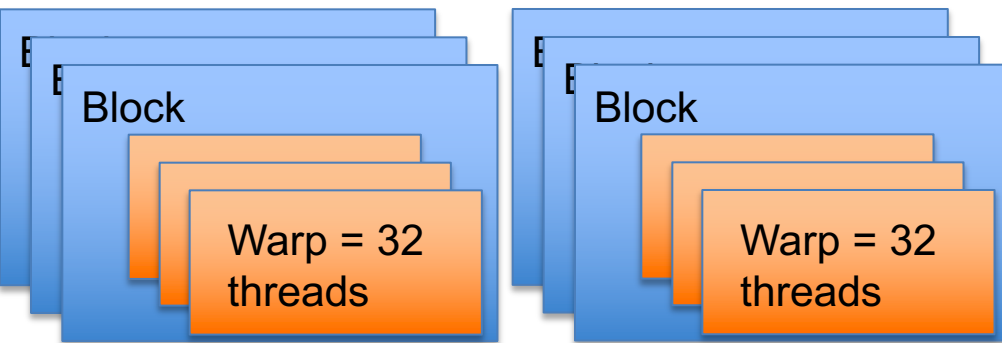
- CUDA
 - C++ compiler
 - Works best for NVIDIA GPUs
 - CUDA SDK
 - Numerous examples and documentation (most for single GPU)
 - Has most functionality
- OpenCL
 - C compiler
 - Platform independent
 - AMD
 - NVIDIA
 - Less control/functionality than CUDA
- Compute Shaders (DirectX, OpenGL).

CUDA

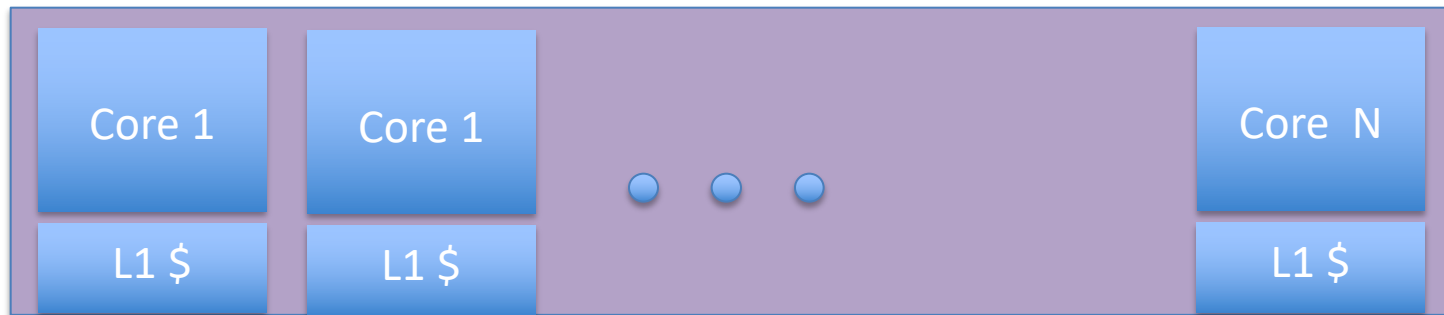
- A kernel (=CUDA program) is executed by 100:s-1M:s threads
 - A "warp" = 32 threads, one thread per ALU
 - Warps (one to ~32) are grouped into one block
 - Block: executed on one core
 - One to 48 warps execute on a core



1 core



Max one program
per block.
One program
counter per warp.



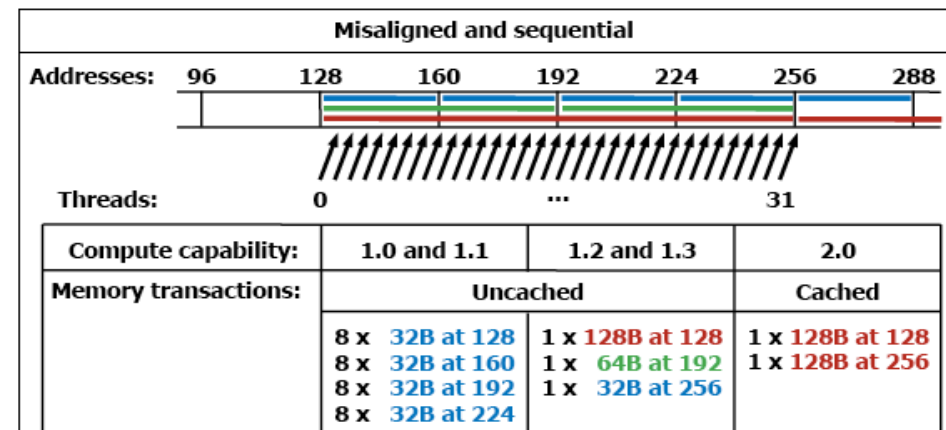
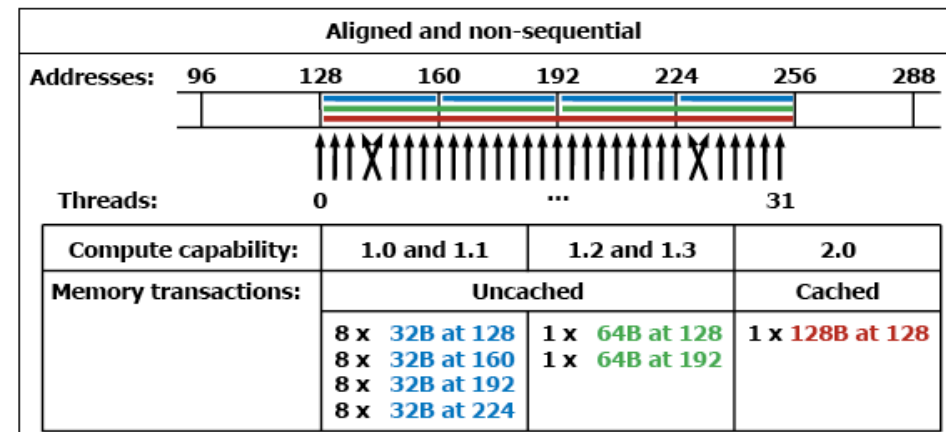
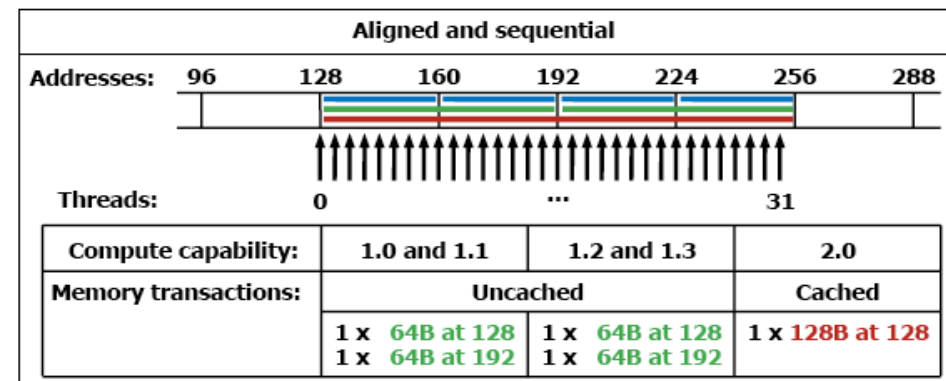
Read whole cache blocks (128 bytes)

- Global mem accesses.

- One transaction:

Bandwidth to GPU RAM is the most precious resource, so two transactions is often bad.

- Two transactions:

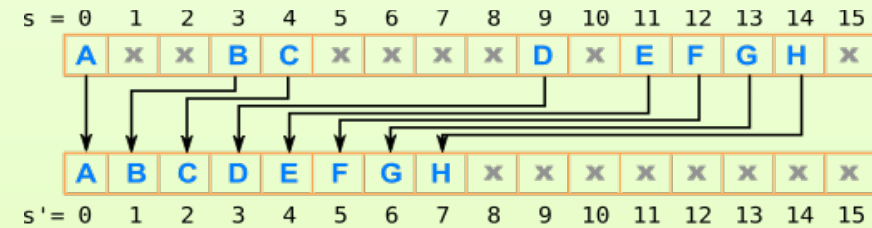


Fermi:

Figure G-1. Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions Based on Compute Capability

Efficient Programming

- If your program can be constructed this way, you are a winner!
- More often possible than anticipated
 - Stream compaction
 - Prefix sums
 - Sorting



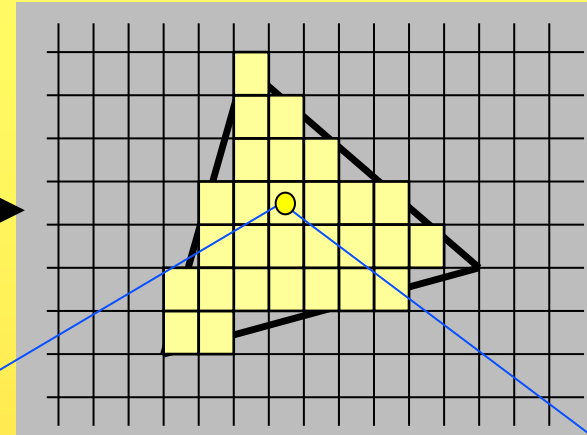
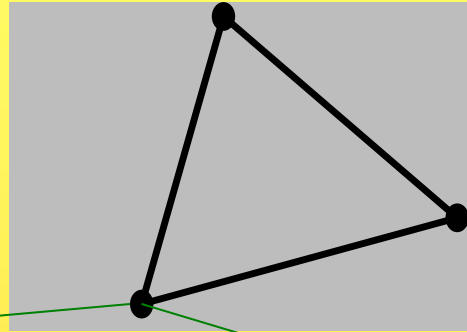
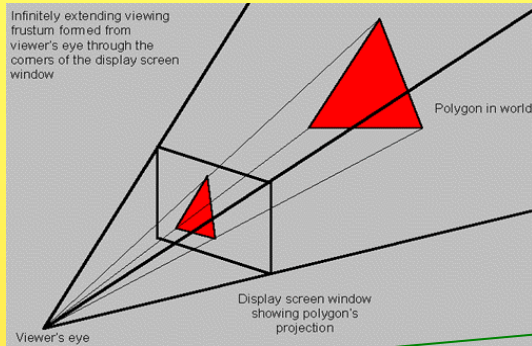
input	1	3	9	4	2	5	7	1	8	4	5	9	3
output	0	1	4	13	15



19 5 100 1 63 79
 ↓
 1 5 19 63 79 100

Fermi: 16 multi-processors à 2x16 SIMD width

Shaders



```
// Vertex Shader
#version 130

in vec3 vertex;
in vec3 color;
out vec3 outColor;
uniform mat4 modelViewProjectionMatrix;

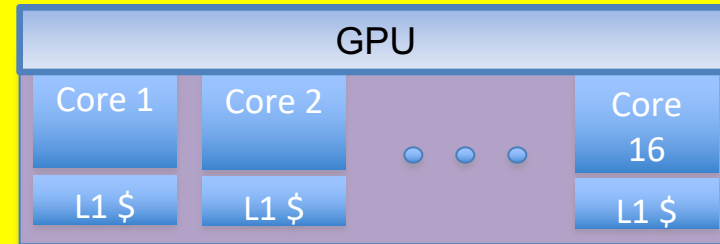
void main()
{
    gl_Position = modelViewProjectionMatrix*vec4(vertex,1);
    outColor = color;
}
```

```
// Fragment Shader:
#version 130
in vec3 outColor;
out vec4 fragColor;

void main()
{
    fragColor = vec4(outColor,1);
}
```

Shaders and coalesced memory accesses

- Each core (e.g. 128-SIMD) typically executes the same instruction per clock cycle for either a:



- Vertex shader:

- E.g. 128 vertices

- Geometry shader

- E.g. 128 triangles

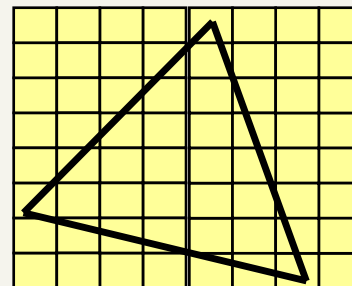
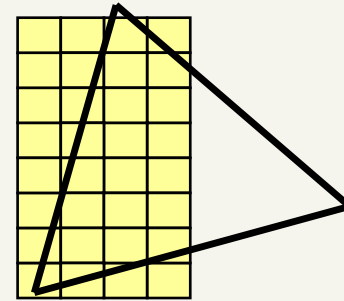
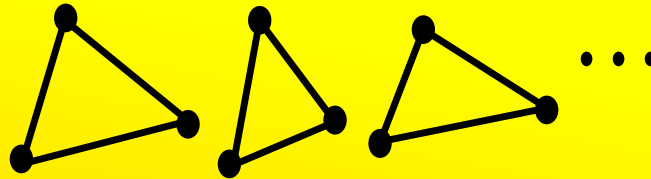
- or Fragment shader:

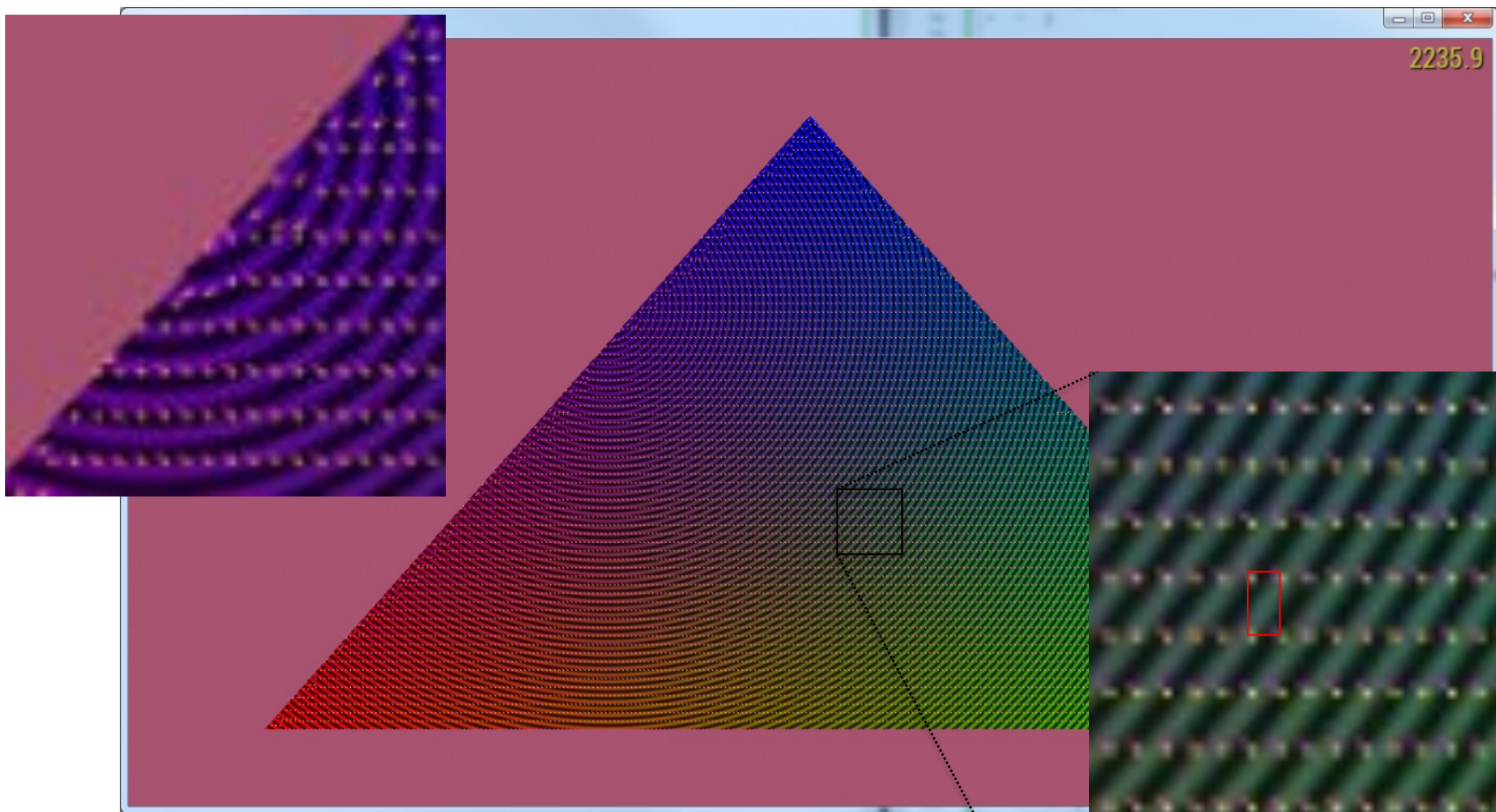
- E.g. 128 pixels

in blocks of at least 2x2 pixels
(to compute texture filter derivatives) .

Here is an example of blocks
 $4 \times 8 = 32$ pixels:

- However, many architectures can execute different instructions, of the same shader, for different warps (warp = group of 32 ALUs)





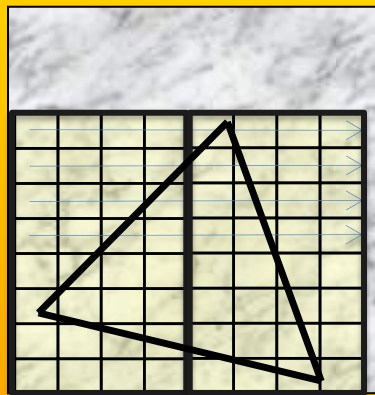
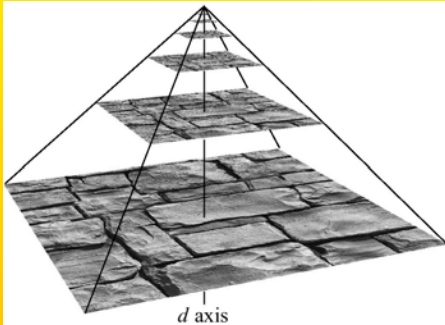
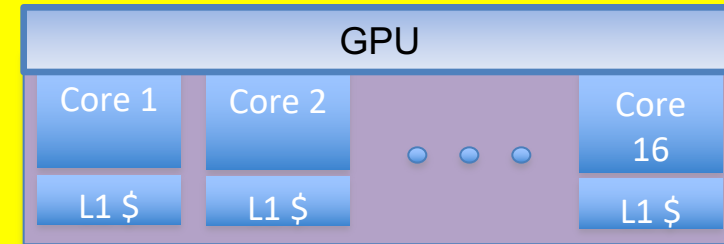
NVIDIA:

yellow = lane 0

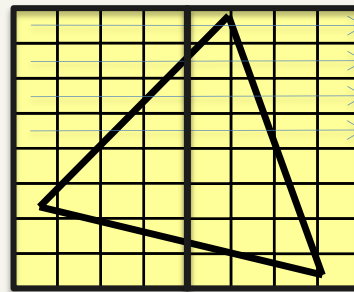
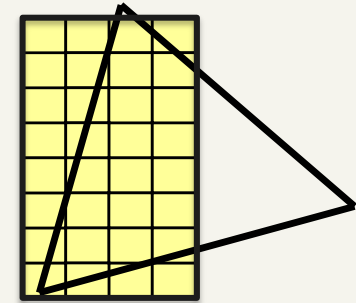
purple = lane 31

Shaders and coalesced memory accesses

- For mipmap-filtered texture lookups in a fragment shader, this can provide decently coalesced memory accesses.



Texture space



Screen space

Thread utilization

- Each core executes one program (=shader)
- Each of the 192 ALUs execute one "thread" (a shader for a vertex or fragment)
- Since the core executes the same instruction for at least 32 threads (as far as the programmer is concerned)...
- If (...) ...the core must
 - Then, $a = b + c$; execute both paths
 - ... if any of the 32 threads need the if and else-path.
- Else
 - $a = c + d$; But not if all need the same path.

Summary

Need to know:

- Perspective correct interpolation (e.g. for textures)

- Taxonomy:

- Sort first
- sort middle
- sort last fragment
- sort last image

- Bandwidth

- Why it is a problem and how to "solve" it
 - L1 / L2 caches
 - Texture caching with prefetching, (warp switching)
 - Texture compression, Z-compression, Z-occlusion testing (HyperZ)

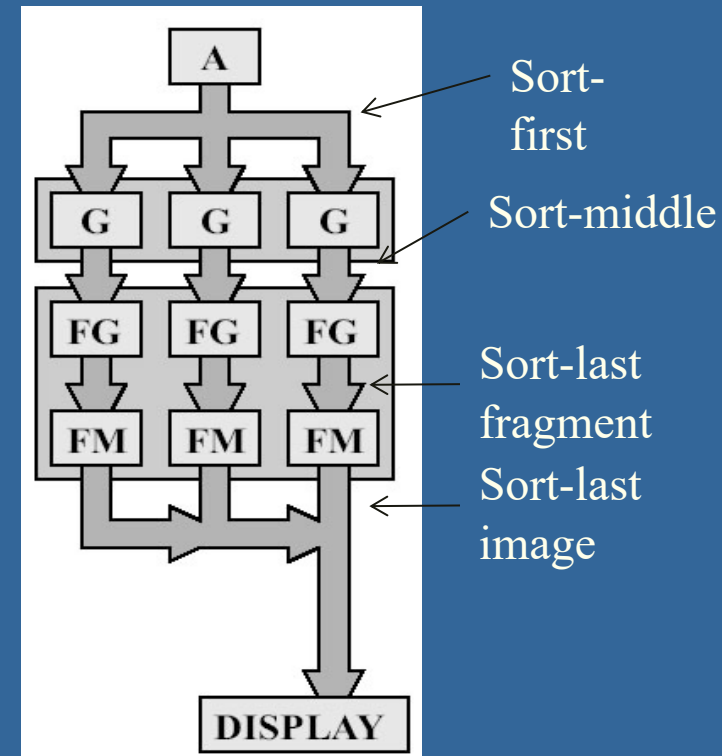
- Be able to sketch the functional blocks and relation to hardware for a modern graphics card (next slide→)

Linearly interpolate (u_i/w_i , v_i/w_i , $1/w_i$) in screenspace from each triangle vertex i .

Then at each pixel:

$$u_{ip} = (u/w)_{ip} / (1/w)_{ip}$$
$$v_{ip} = (v/w)_{ip} / (1/w)_{ip}$$

where ip = screen-space interpolated value from the triangle vertices.



The graphics-pipeline's functional blocks and their relation to hardware

