

CHALMERS

Graphics Hardware



Ulf Assarsson

Graphics hardware – why?

- About 100x faster!
- Another reason: about 100x faster!
- Simple to pipeline and parallelize

- Current hardware based on triangle rasterization with programmable shading (e.g., OpenGL acceleration)
- Ray tracing: there are research architectures, and few commercial products
 - Renderdrive, RPU, (Gelato), NVIDIA OptiX
 - Or write your own GPU ray-tracer



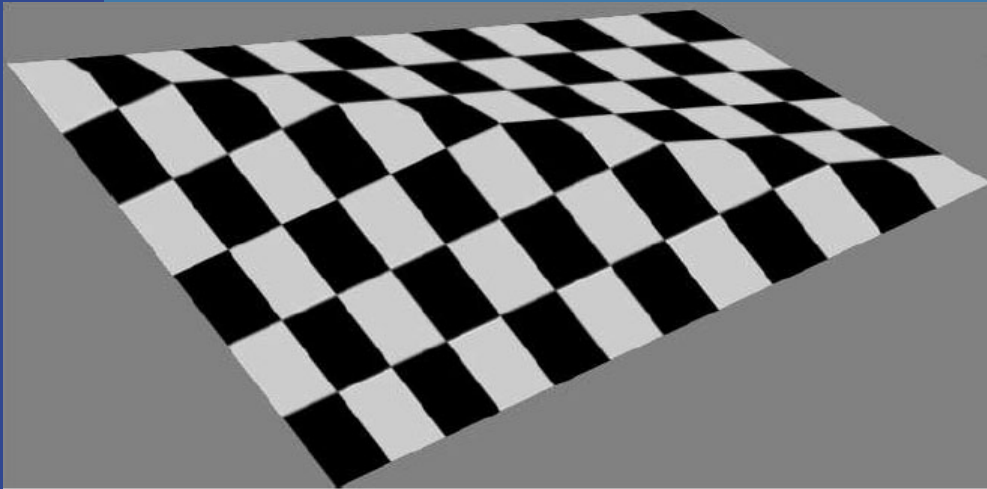
**Steel
Monkeys**



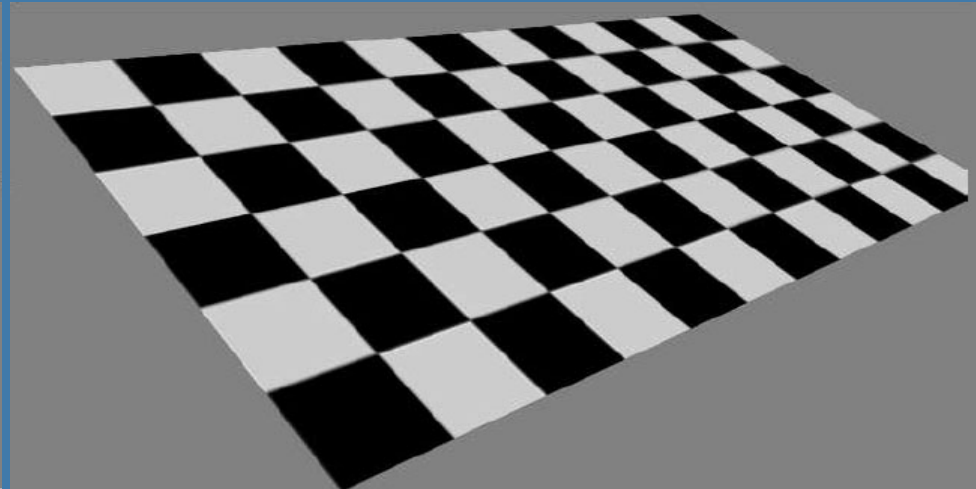
**CLASS 1
Game product**

Perspective-correct texturing

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

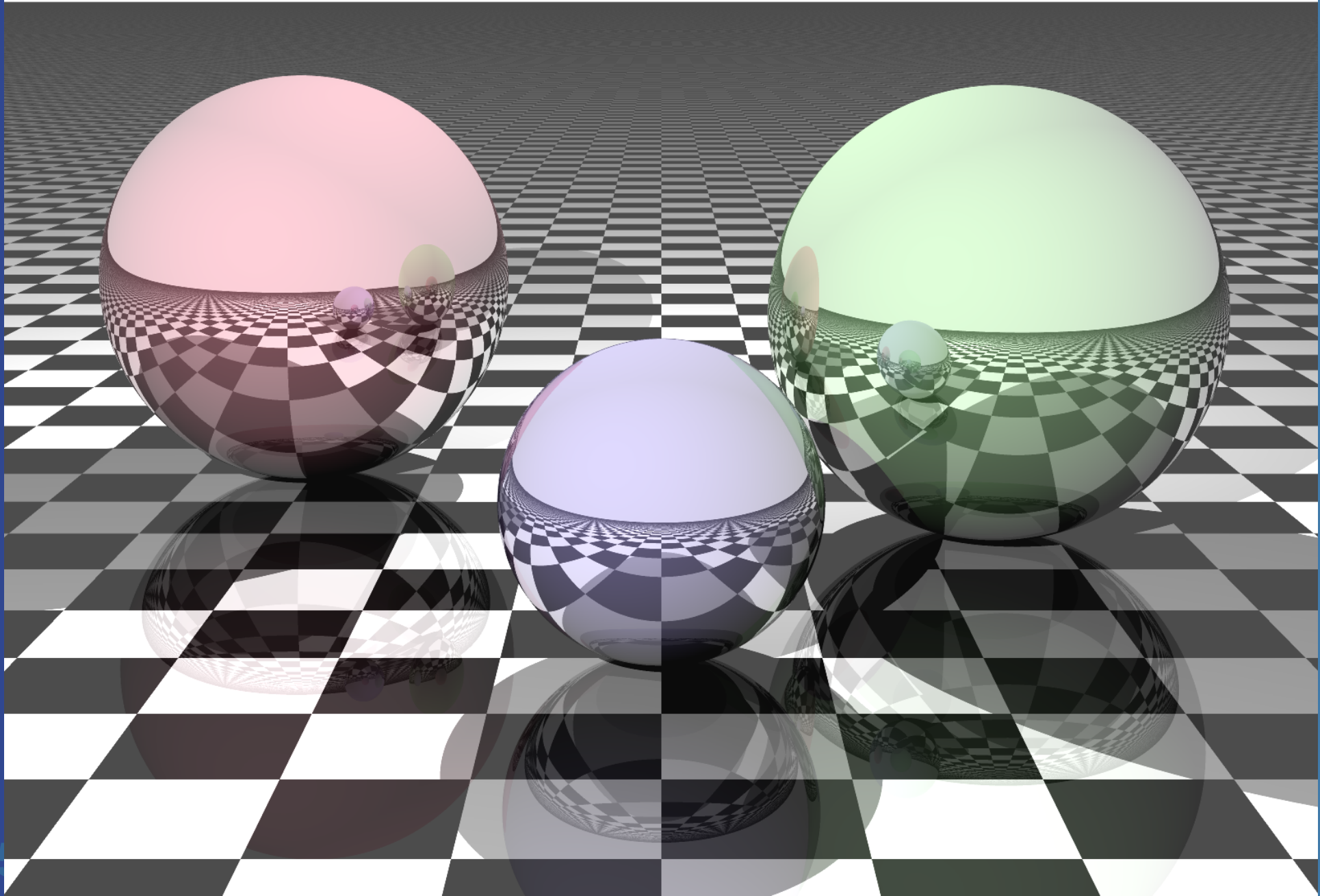


Linear interpolation



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. Thus, tex coords cannot be linearly interpolated (just like a 3D-position cannot be).

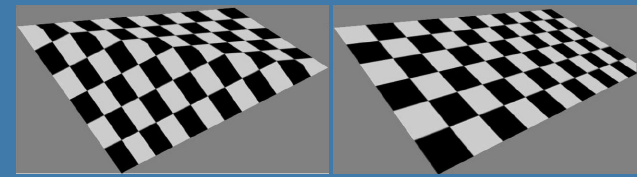
- Before projection, \mathbf{v} , and after \mathbf{p} ($\mathbf{p}=\mathbf{M}\mathbf{v}$)
- After projection p_w is not 1!
- Homogenization: $(p_x/p_w, p_y/p_w, p_z/p_w, 1)$
- Gives $(p_x', p_y', p_z', 1)$

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

Texture coordinate interpolation

- Linear interpolation does not work
- Rational linear interpolation does:
 - $u(x) = (ax+b) / (cx+d)$ (along a scanline where $y = \text{constant}$)
 - a, b, c, d are computed from triangle's vertices (x, y, z, w, u, v)
- Not really efficient to compute a, b, c, d per scan line
- Smarter:
 - Compute $(u/w, v/w, 1/w)$ per vertex
 - These quantities can be linearly interpolated!
 - Then at each pixel, compute $1/(1/w) = w$
 - And obtain: $(w * u/w, w * v/w) = (u, v)$
 - The (u, v) are perspective-correctly interpolated
- Need to interpolate shading this way too
 - Though, not as annoying as textures
- Since linear interpolation now is OK, compute, e.g., $\Delta(u/w) / \Delta x$, and use this to update u/w when stepping in the x -direction (similarly for other parameters)

Put differently:



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

For a formal proof, see Jim Blinn, "W Pleasure, W Fun", IEEE Computer Graphics and Applications, p78-82, May/June 1998

Need to interpolate shading this way too, though, not as annoying as textures

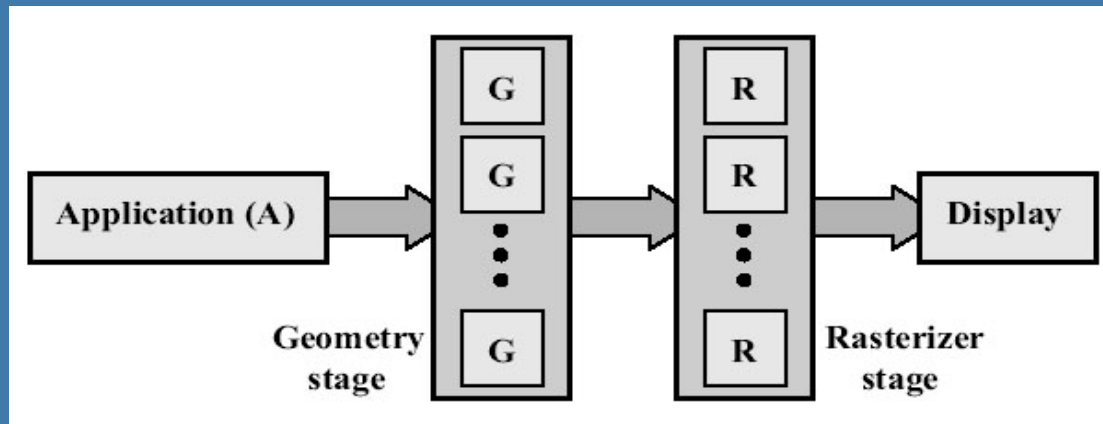
Background:

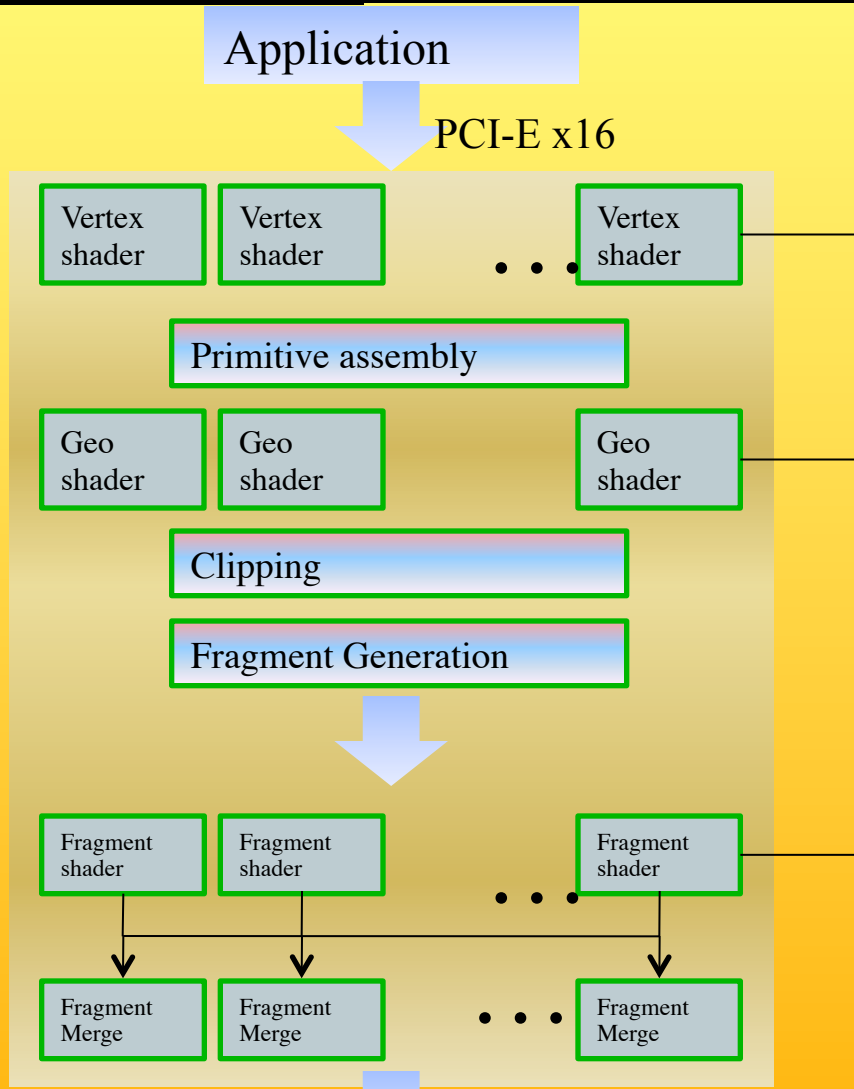
Graphics hardware architectures

- 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:

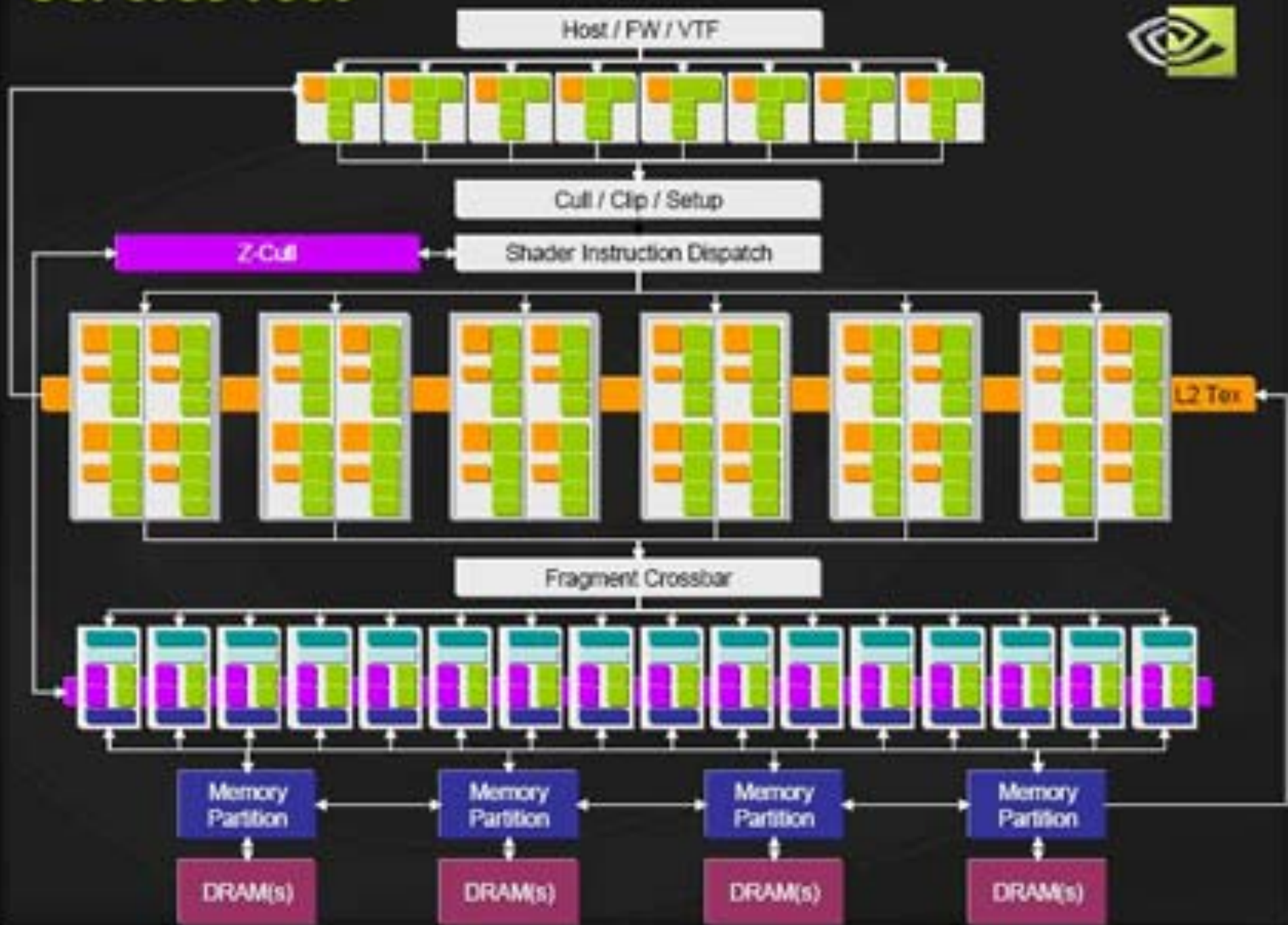




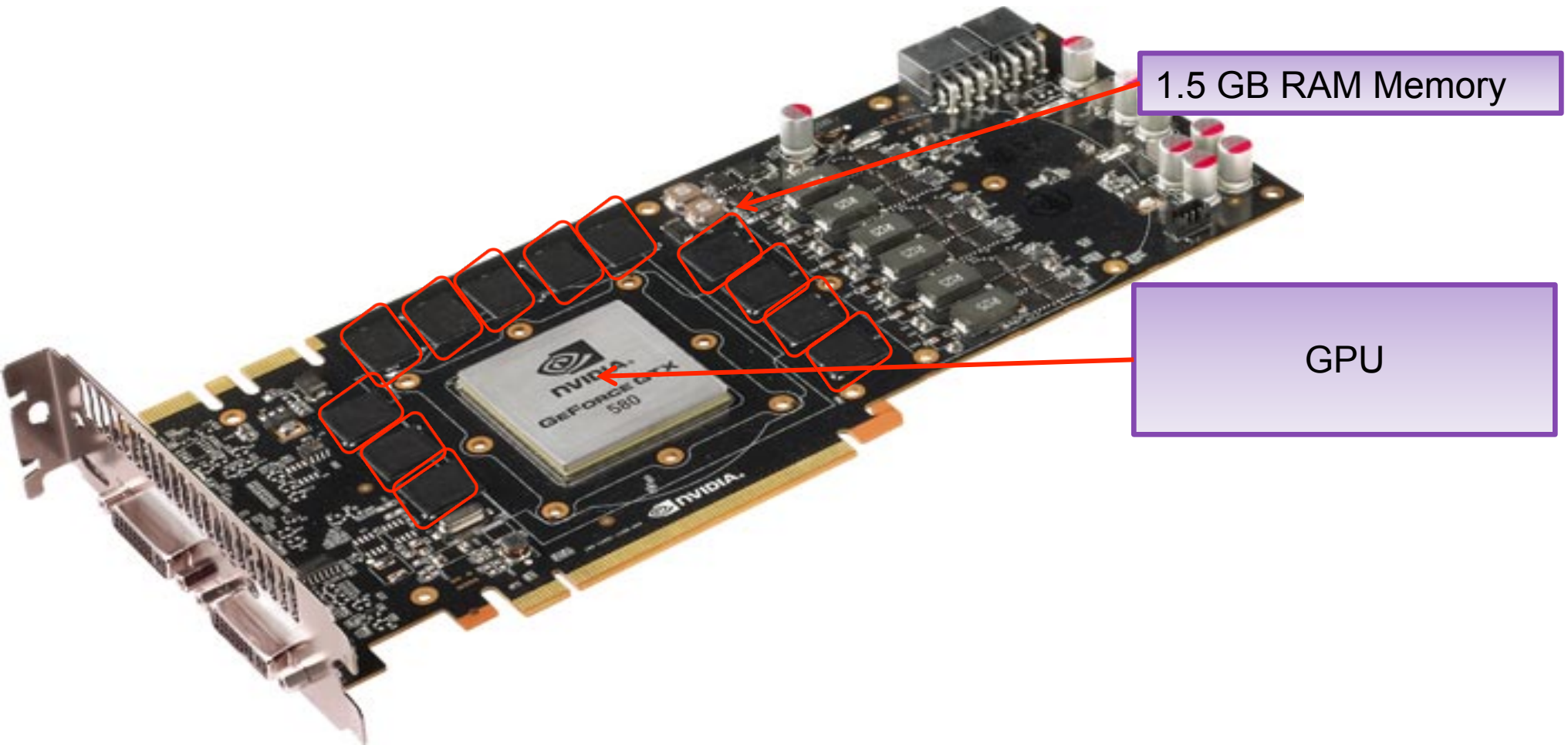
On NVIDIA
8000/9000/200/400/500/6
00/700/TITAN/900-series:
Vertex-, Geometry- and Fragment
shaders allocated from a pool of
up to thousands of ALUs



GeForce 7800

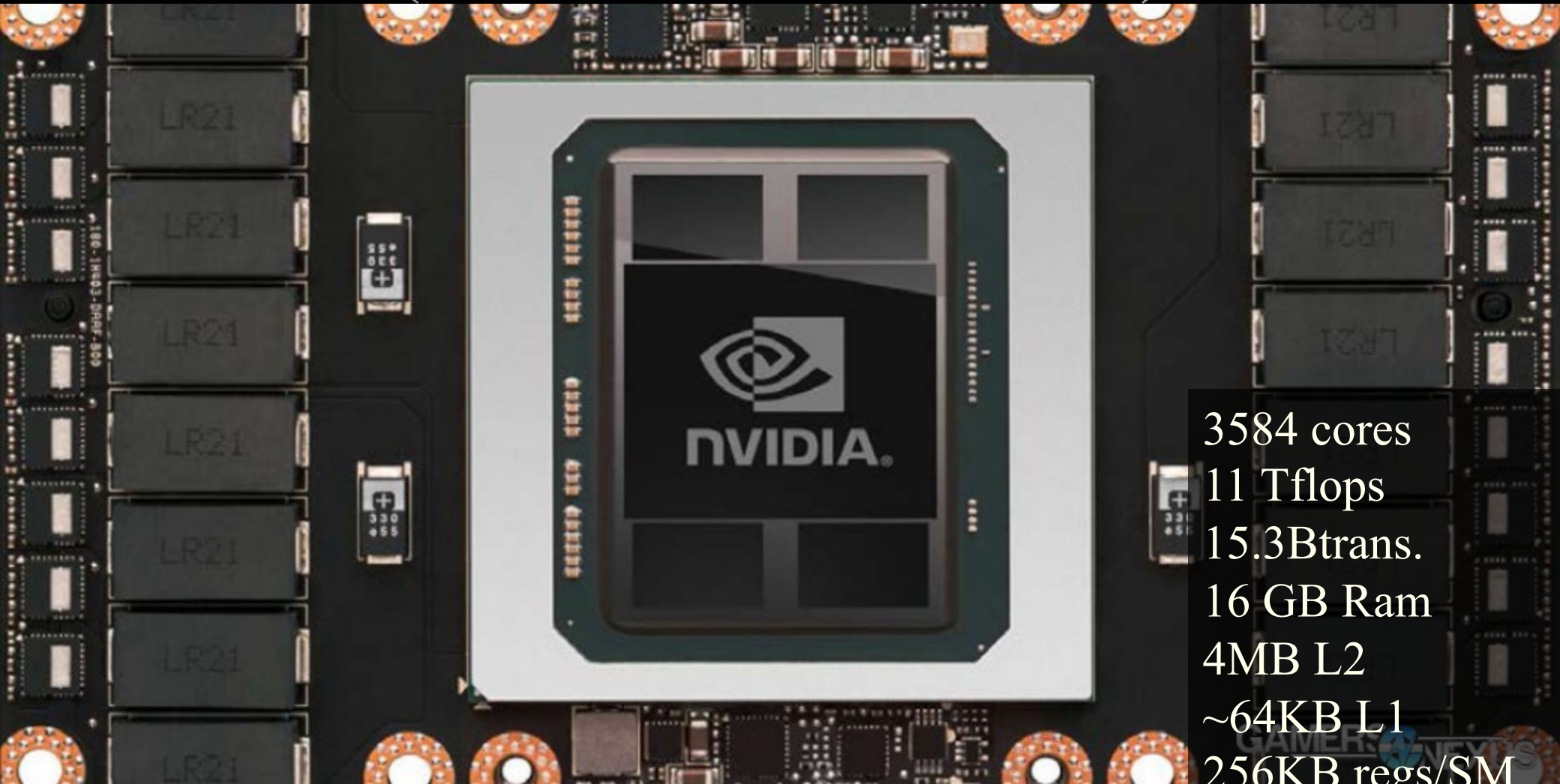


Graphics Processing Unit - GPU



- NVIDIA Geforce GTX 580

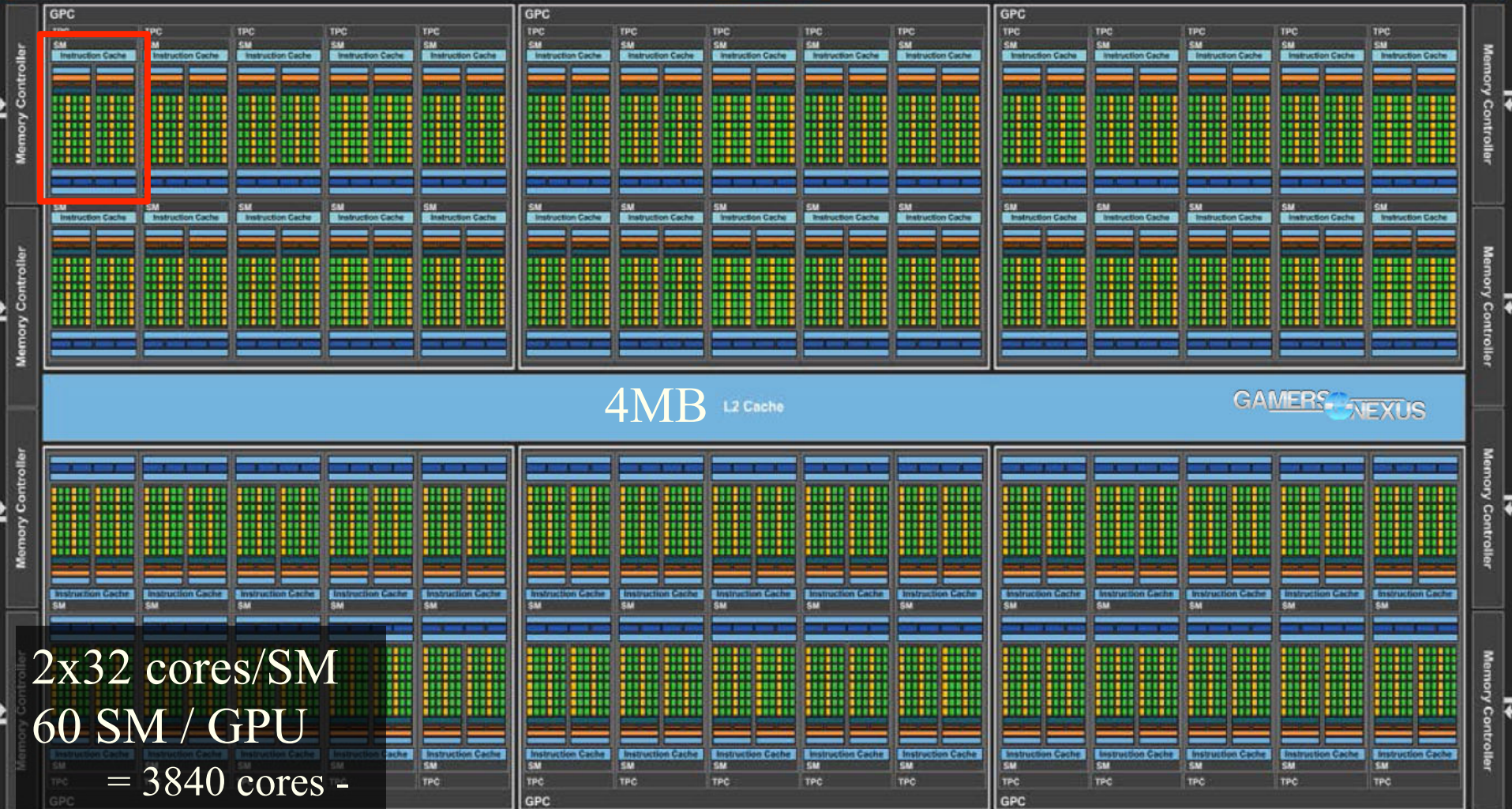
NVIDIA Pascal GP100 (GTX 1080 / Titan X)



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

PCI Express 3.0 Host Interface

GigaThread Engine



2x32 cores/SM
60 SM / GPU
= 3840 cores -
disabled spill
= 3584 cores

High-Speed Hub

NVLink

NVLink

NVLink

GAMERS NEXUS

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

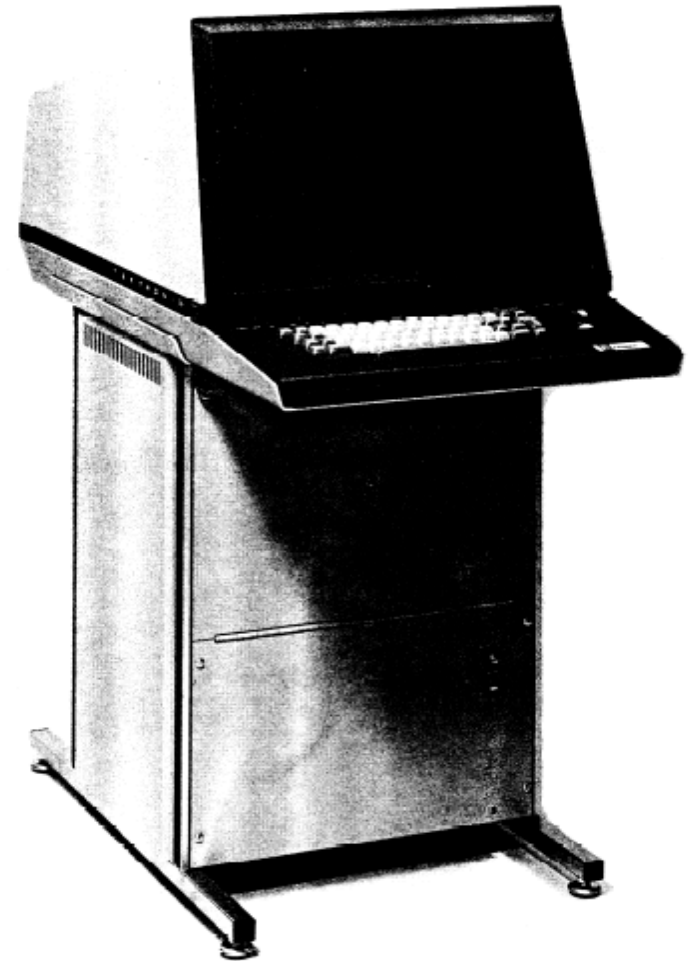
Graphics Hardware History

- 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



Direct View Storage Tube

- Created by Tektronix
 - 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



Tektronix 4014

Fig. 1-1. 4014 Computer Display Terminal.

Briefly about Graphics HW pipelining

2001 • In GeForce3: 600-800 pipeline stages!

- 57 million transistors
- First Pentium IV: 20 stages, 42 million transistors,

• Evolution of cards:

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, bw 54GB/s, 430 MHz core, mem 650MHz(1.3GHz)

2006 - GeForce 8800: 681M trans, 39.2Gpix/s, 10.6Gverts/s, bw:103.7 GB/s, 612 MHz core (1500 for shaders), 1080 MHz mem (effective 2160 GHz)

2008 - Geforce 280 GTX: 1.4G trans, 65nm, 602/1296 MHz core, 1107(*2)MHz mem, 142GB/s, 48Gtex/s

2007 - ATI Radeon HD 5870: 2.15G trans, 153GB/s, 40nm, 850 MHz, GDDR5, 256bit mem bus,

2010 - Geforce GTX480: 3Gtrans, 700/1401 MHz core, Mem (1.848G(*2)GHz), 177.4GB/s, 384bit mem bus, 40Gtexels/s

2011 - GXT580: 3Gtrans, 772/1544, Mem: 2004/4008 MHz, 192.4GB/s, 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

- Lesson learned: #trans doubles ~per 2 years. Core clock increases slowly. Mem clock -increases with new technology DDR2, DDR3, GDDR5, and with more memory busses (à 64-bit). Now stacked mem.

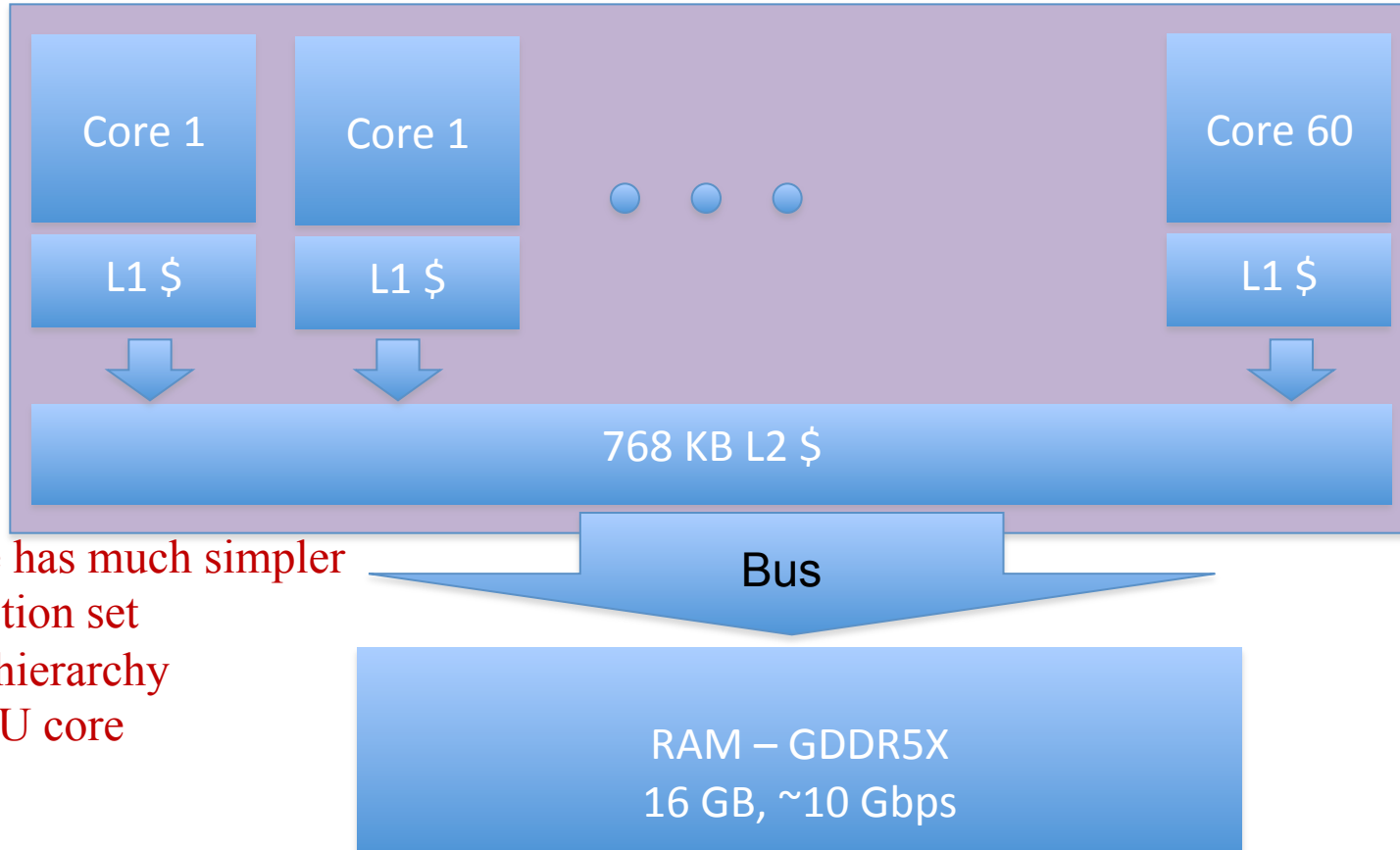
- We want as fast memory as possible! Why?

- Parallelization can cover for slow clock. Parallelization more energy efficient than high clock frequency. Powerconsumption prop. to freq².
- Memory transfers often the bottleneck

GPU- Nvidia's Pascal 2016



Overview:



60 cores à
64-SIMD width
(2*4*8)

~64 KB per each
64 SIMD

Bandwidth
~480 GB/s

Bus:
256/384/4096
bits

Compare to
ATI 2900:

- 2x512bits

Larrabee:

- 2x512bits

GPU core has much simpler

- instruction set
 - cache hierarchy
- than a CPU core

Wish:

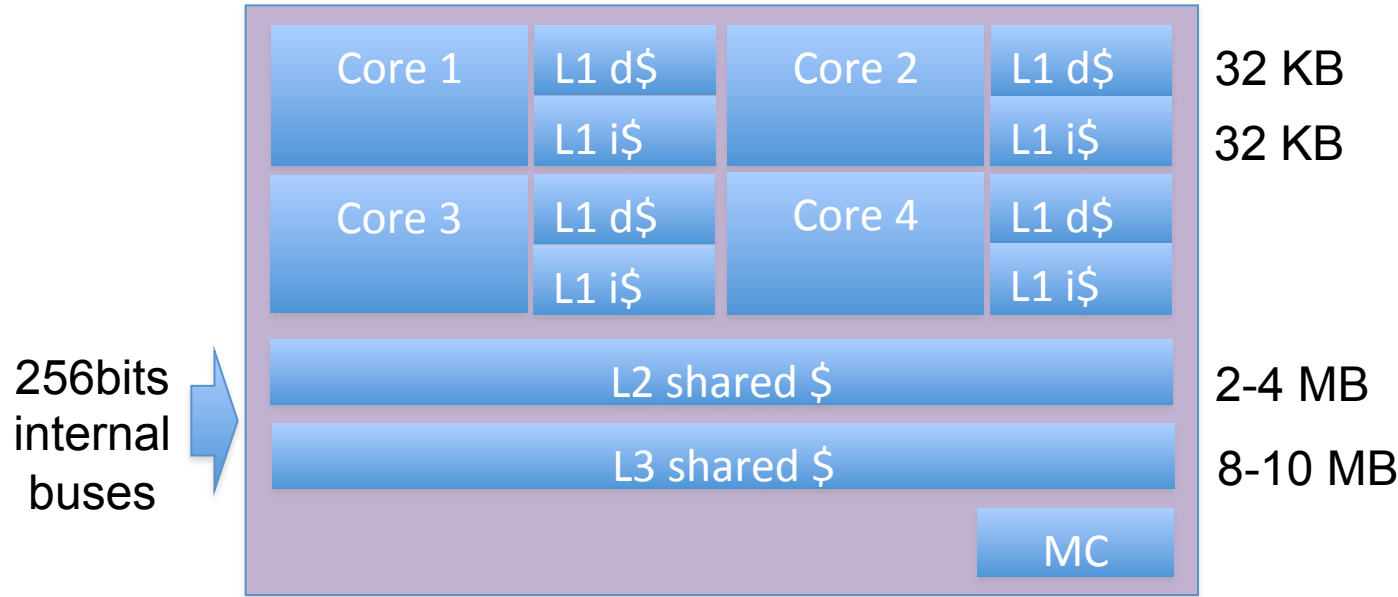
3584 ALUs à 1 float/clock => 14KB/clock

~1.5GHz core clock => 21500 GB/s request

We have ~480GB/s. In reality we can do 20-40 instr. between each RAM-read/write. Solved by L1\$ + L2\$ + latency hiding (warp switching)

CPU - 2011

AVX:
Intel's Sandybridge
AMD's Bulldozer



1 – 8 cores à
4 SIMD floats
(16 SIMD for
bytes)



- 8 cores à 4 floats
⇒ We want 128 bytes/clock (e.g. from RAM)
⇒ 128GByte/s, 1GHz CPU

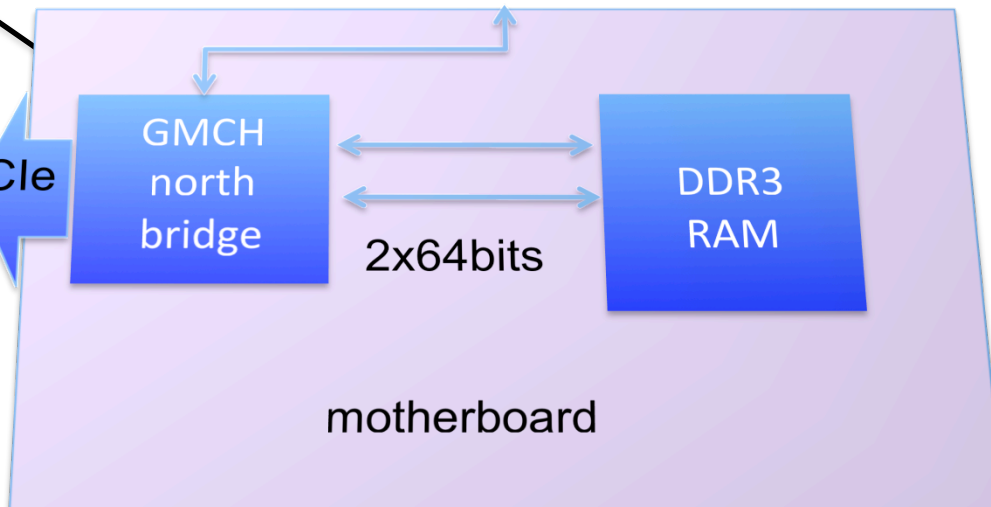
- In addition, x3, since:
 $r1 = r2 + r3;$

In reality: 6-12GB/s

Solved by \$-hierarchy + registers

Graphics
Memory
Controller
HUB

Gfx card PCIe



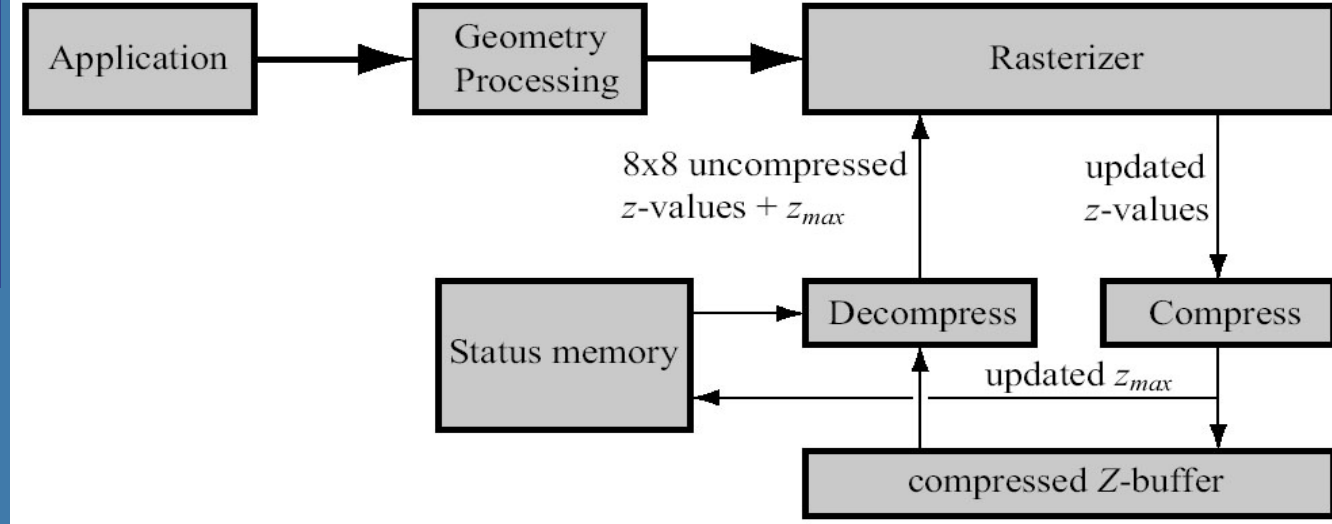
Memory bandwidth usage is huge!!

- On top of that bandwidth usage is never 100%.
- However, there are many techniques to reduce bandwidth usage:
 - Texture caching with prefetching
 - Texture compression
 - Z-compression
 - Z-occlusion testing (HyperZ)

Z-occlusion testing and Z-compression

- One way of reducing bandwidth
 - ATI Inc., pioneered with their HyperZ technology
- Very simple, and very effective
- Divide screen into tiles of 8x8 pixels
- Keep a status memory on-chip
 - Very fast access
 - Stores additional information that this algorithm uses
- Enables occlusion culling on triangle basis, z-compression, and fast Z-clears

Architecture of Z-cull and Z-compress



- Store z_{max} per tile, and a flag (whether cleared, compressed/uncompressed)
- Rasterize one tile at a time
- Test if z_{min} on triangle is farther away than tile's z_{max}
 - If so, don't do any work for that tile!!!
 - Saves texturing and z-read for entire tile – huge savings!
- Otherwise read compressed Z-buffer, & unpack
- Write to unpacked Z-buffer, and when finished compress and send back to memory, and also: update z_{max}
- For fast Z-clears: just set a flag to "clear" for each tile
 - Then we don't need to read from Z-buffer, just send cleared Z for that tile

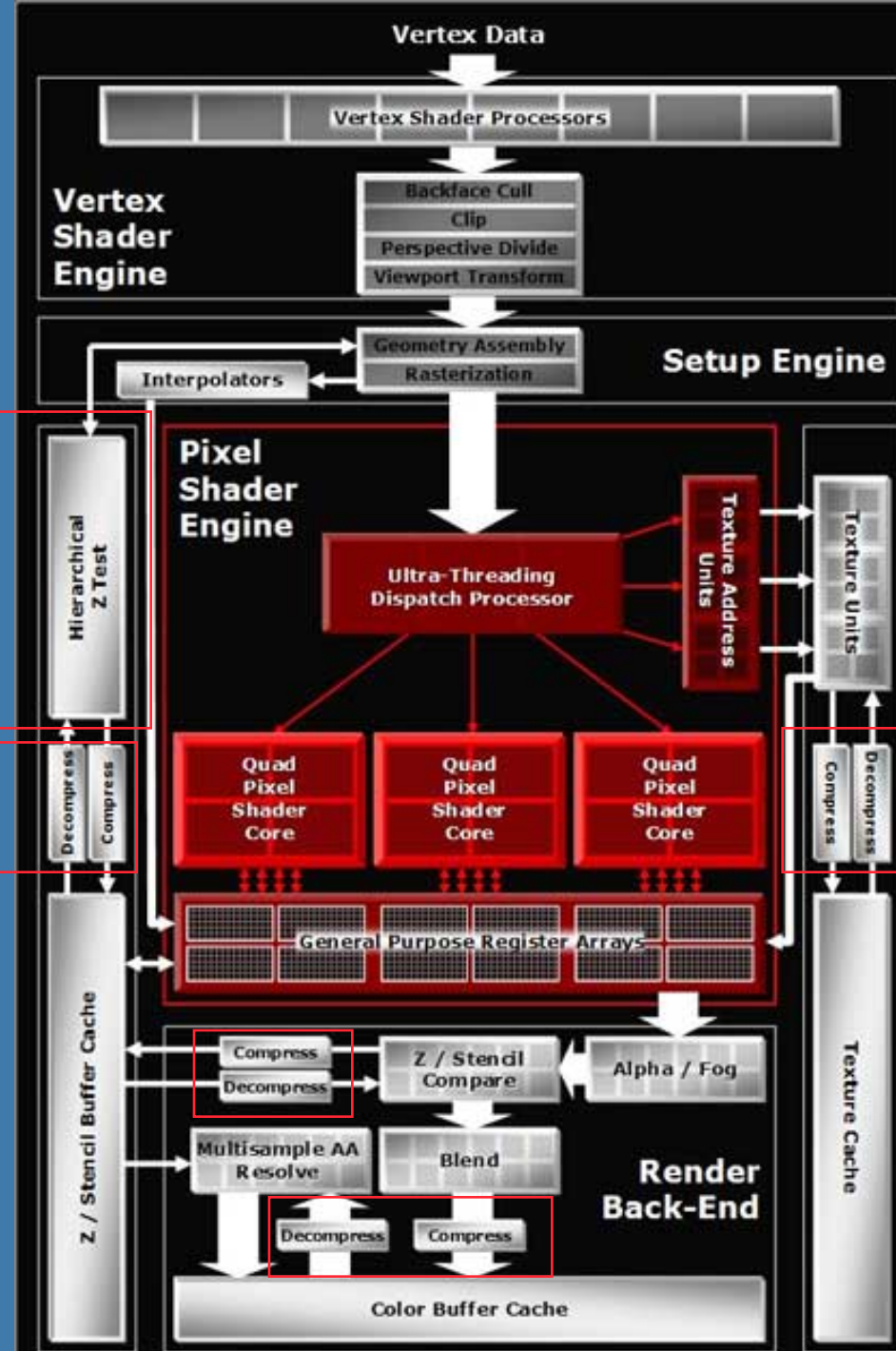
X1800 GTO

- Real example

Z-cull

Z-compress

Also note texture compress and color compress



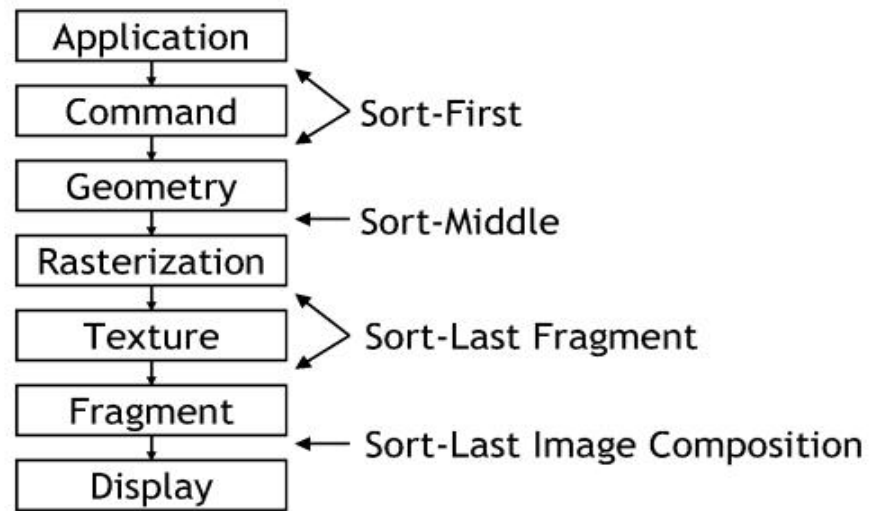
Taxonomy of Hardware

- We can do many computations in parallel:
 - Pixel shading, vertex shading, geometry shading
 - x, y, z, w r, g, b, a
- But results need to be sorted somewhere before reaching the screen.
 - Operations can be parallelized but result on screen must be as if each triangle were rendered one by one in their incoming order (according to OpenGL spec)
 - E.g., for blending (transparency), (z-culling, stencil test)

Taxonomy of hardware

- Need to sort from model space to screen space
- Gives four major architectures:
 - Sort-first
 - Sort-middle
 - Sort-Last Fragment
 - Sort-Last Image

Sorting Taxonomy



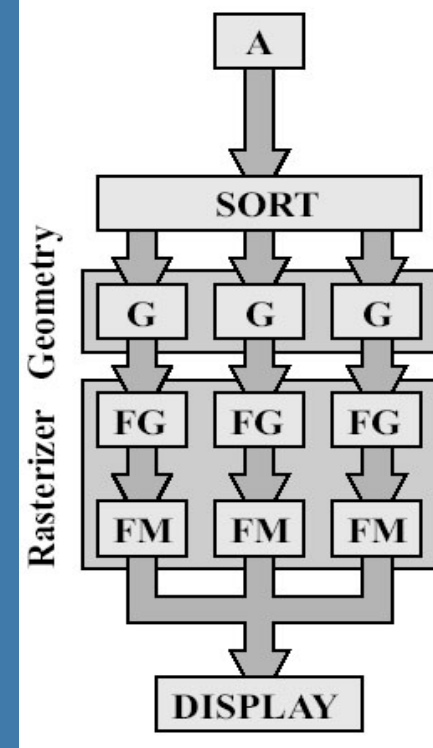
CS448 Lecture 9

Kurt Akeley, Pat Hanrahan, Fall 2001

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

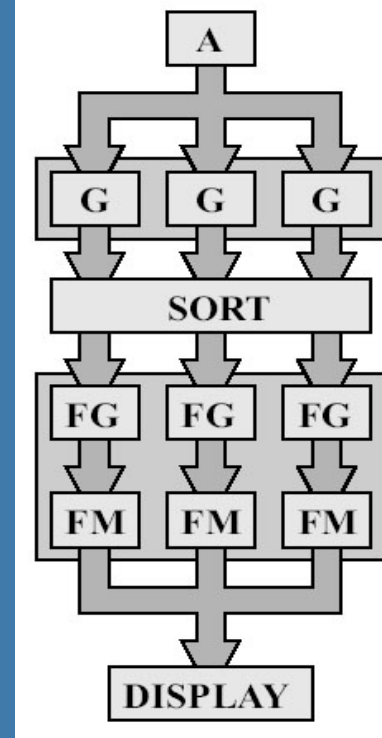
Sort-First

- Sorts primitives before geometry stage
 - Screen is divided into large regions
 - A separate pipeline is responsible for each region (or many)
 - But vertex shader can change screen location!
- G is geometry, FG & FM is part of rasterizer
 - 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))
- Not explored much at all



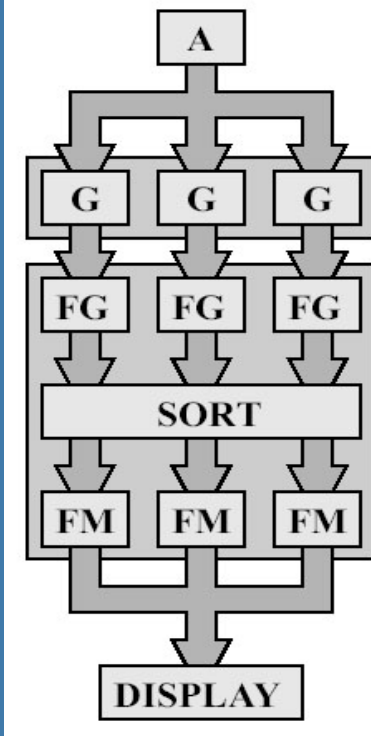
Sort-Middle

- 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) and the 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
- A triangle can be sent to many FG's depending on overlap (over tiles)



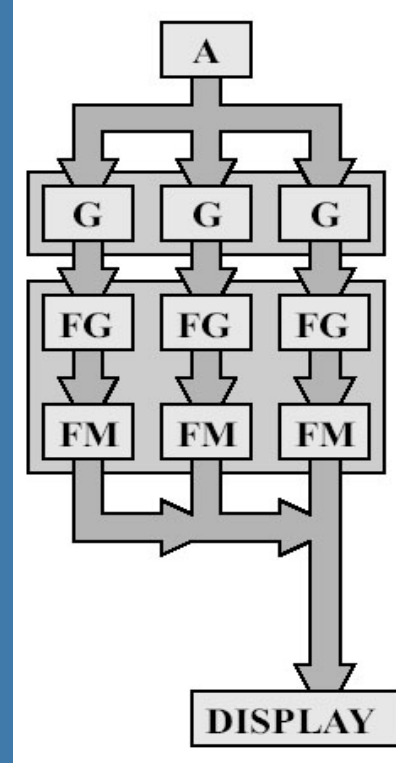
Sort-Last Fragment

- Sorts between FG and FM
- XBOX, PS3, nVidia use this
- Again spread work among G's
- The generated work is sent to FG's
- Then sort fragments to FM's
 - An FM is responsible for a tile of pixels
- A triangle is only sent to one FG, so this avoids doing the same work twice
 - Sort-Middle: If a triangle overlaps several tiles, then the triangle is sent to all FG's responsible for these tiles. Results in extra work

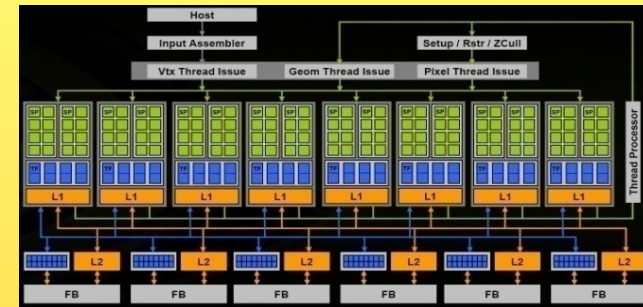
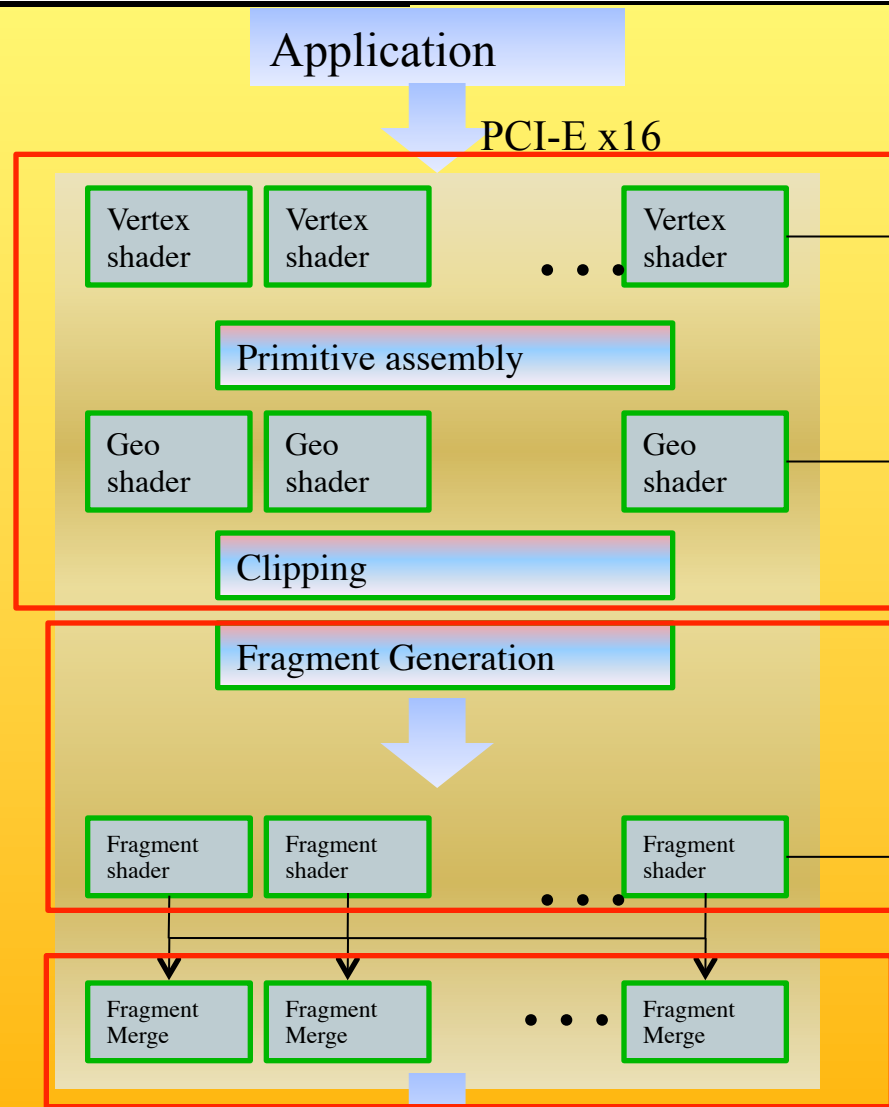


Sort-Last Image

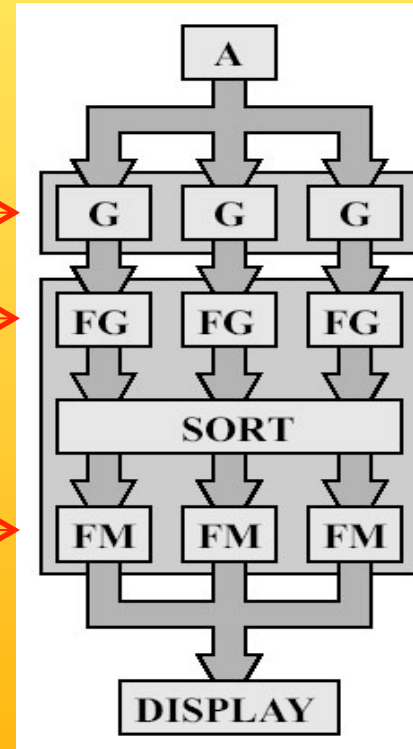
- Sorts after entire pipeline
- So each FG & FM has a separate frame buffer for entire screen (Z and color)
- After all primitives have been sent to the pipeline, the z-buffers and color buffers are merged into one color buffer
- Can be seen as a set of independent pipelines
- Huge memory requirements!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



Logical layout of a graphics card:



On NVIDIA
8000/9000/200-1000
/Titan-series:
→ Vertex-, Geometry- and
Fragment shaders allocated
from a pool of
128/240/480/1536/3072/
3584 processors (=ALUs)

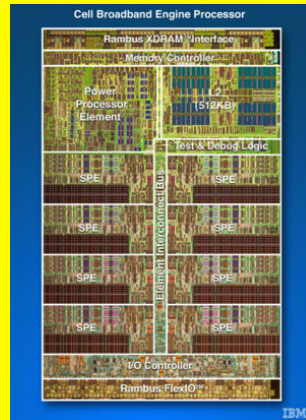


Current and Future Multicores in Graphics

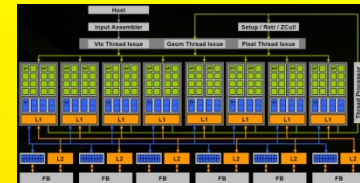
- Cell – 2005
 - 8 cores à 4-float SIMD
 - 256KB L2 cache/core
 - 128 entry register file
 - 3.2 GHz

PowerXCell 8i Processor – 2008

- 8 cores à 4-float SIMD
- 256KB L2 cache
- 128 entry register file
- but has better double precision support



- 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 (rumour)
 - 1.2-1.625 GHz



- Larrabee – "2010"
 - 16-24 cores à 16-float SIMD (Xeon Phi: 61 cores, 2012)
 - Core = 16-float SIMD (=512bit FPU) + x86 proc with loops, branches + scalar ops, 4 threads/core
 - 32KB L1cache, 256KB L2-cache (512KB/core)
 - 1.7-2.4 GHz (1.1 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



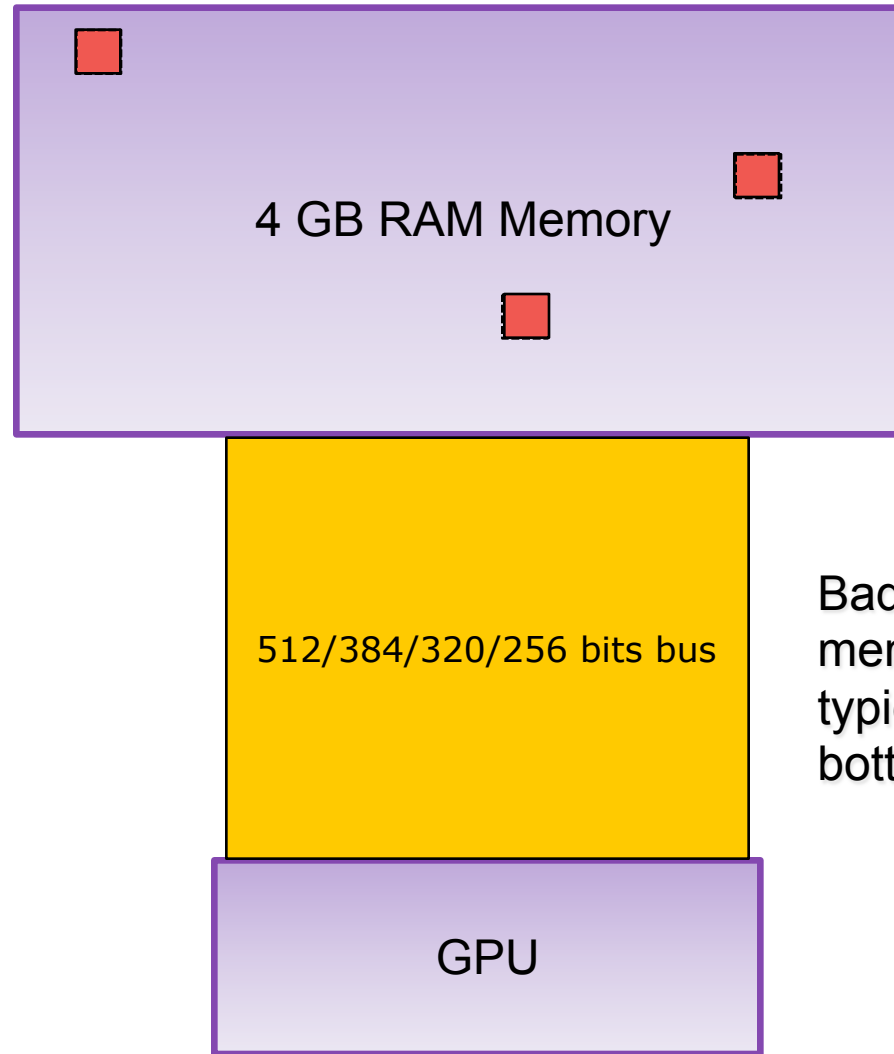
If we have time...

How create efficient GPU
programs?


Answer: coalesced memory
accesses

Graphics Processing Unit - GPU

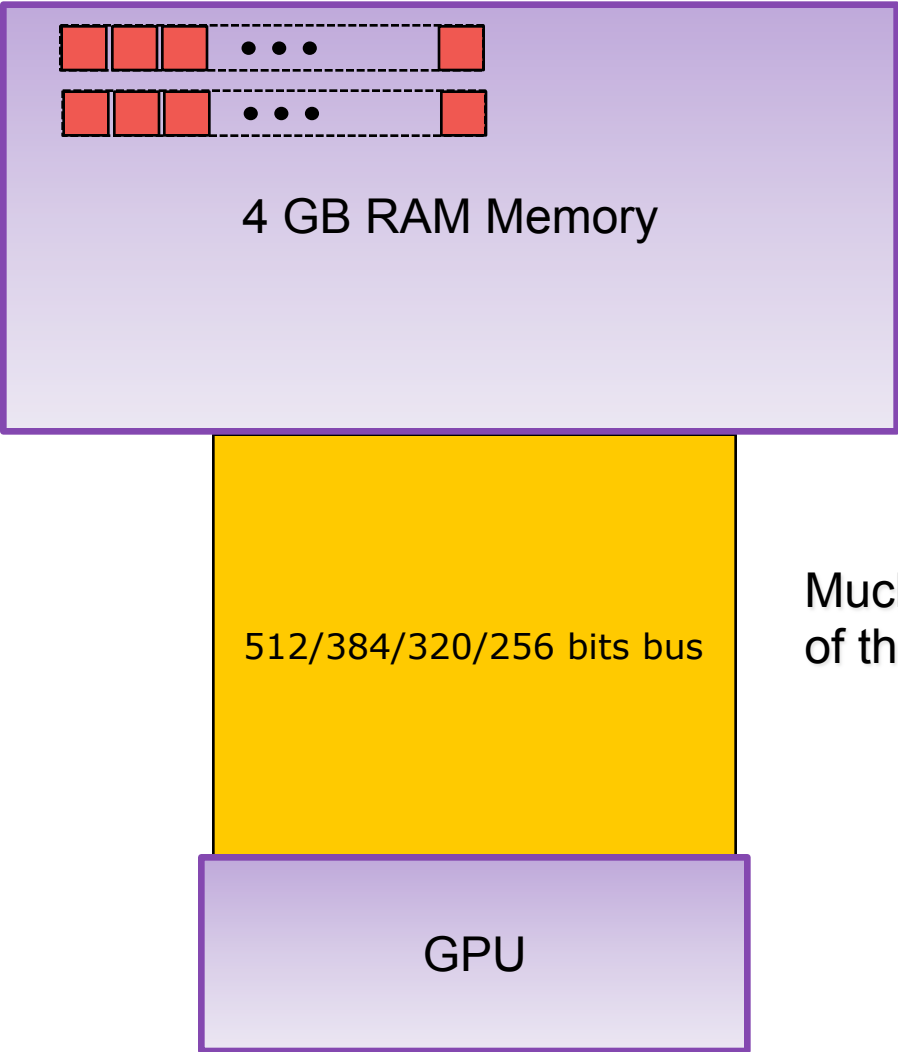
Conceptual layout:



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


 = memory element (32 bits)

Graphics Processing Unit - GPU

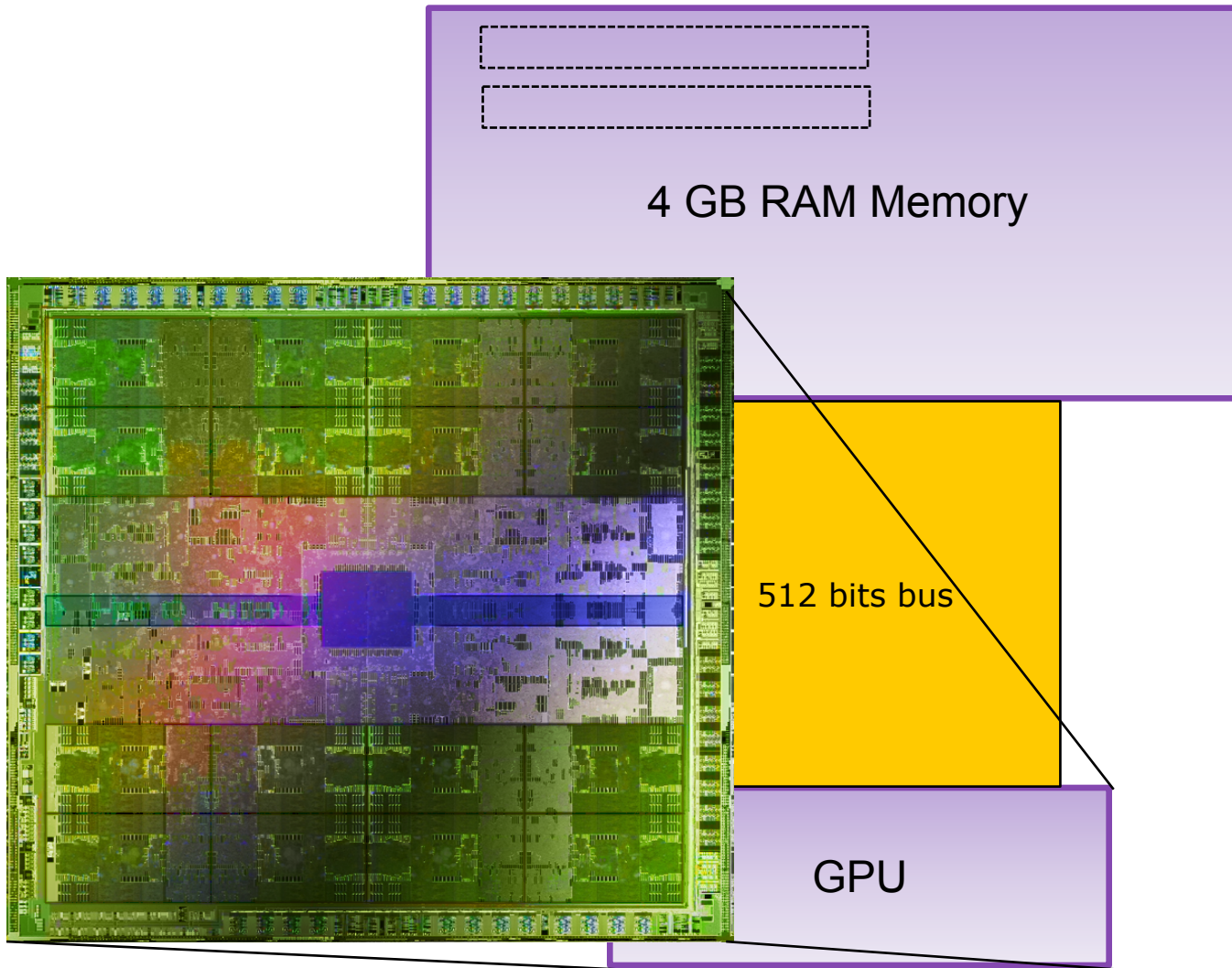


Read 32
coalesced floats
for max
bandwidth usage

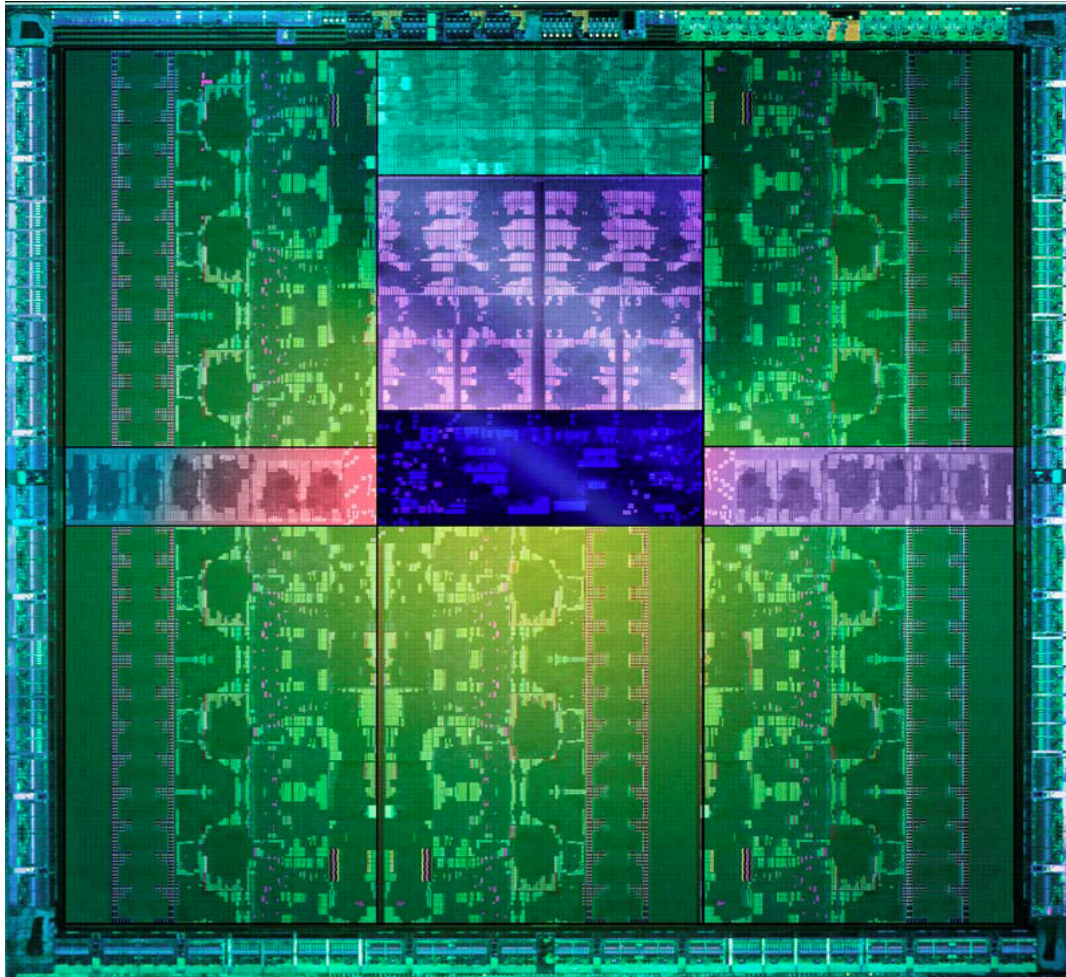
Much better utilization
of the memory bus!

 = memory element (32 bits)

Let's look at the GPU

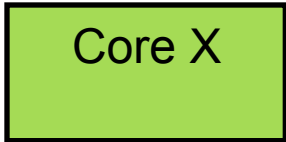
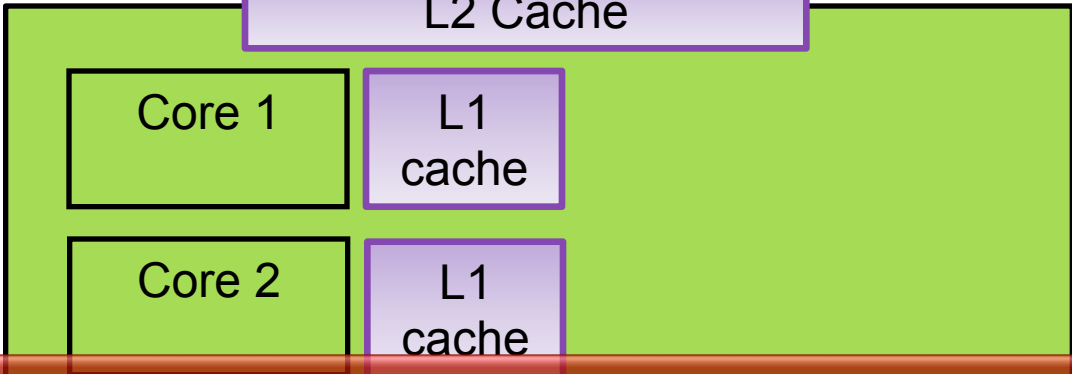


Let's look at the GPU

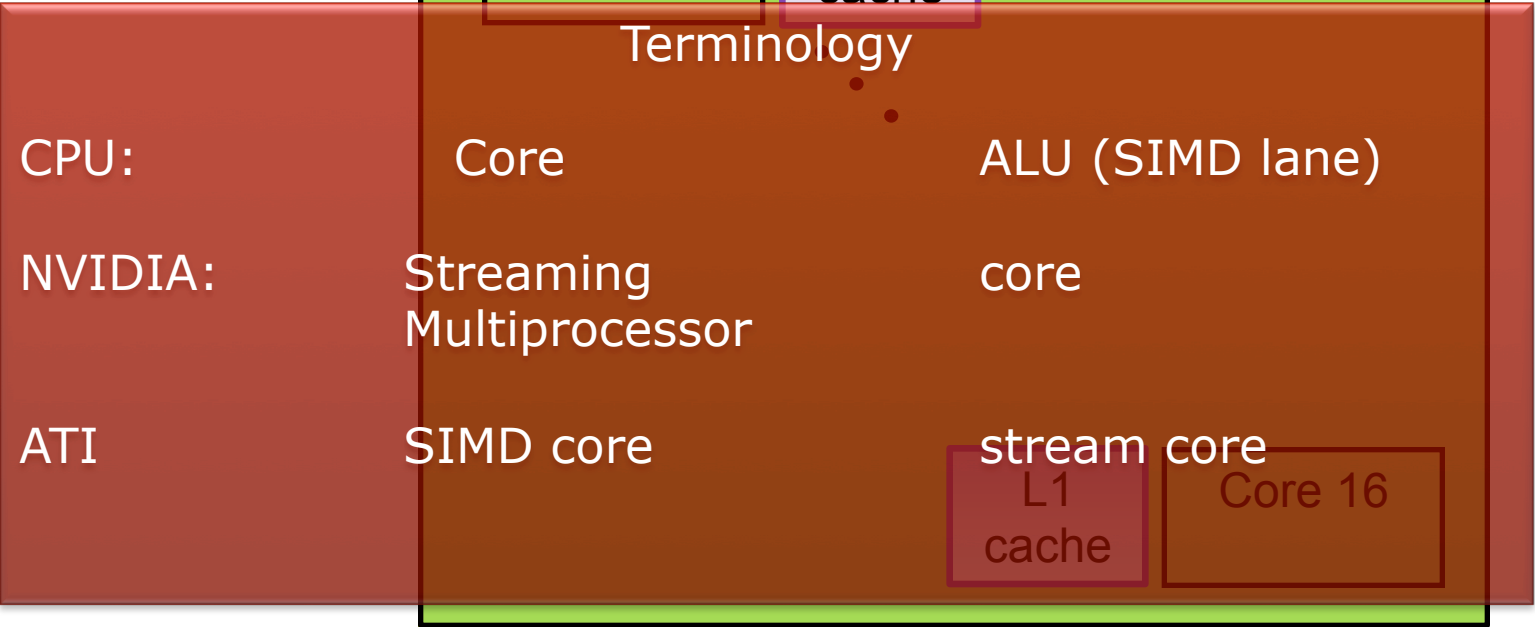


NVIDIA Kepler: 15-16 multi-processors (GTX 680, ~2012)

Let's look at



Terminology



192 ALUs or "lanes"
 (logically: 6 x 32-SIMD width)
 6x32 mul/add per 1-2 clocks
 (6x32 "threads")

SIMD = single instruction multiple data

Kepler: 15-16 multi-processors

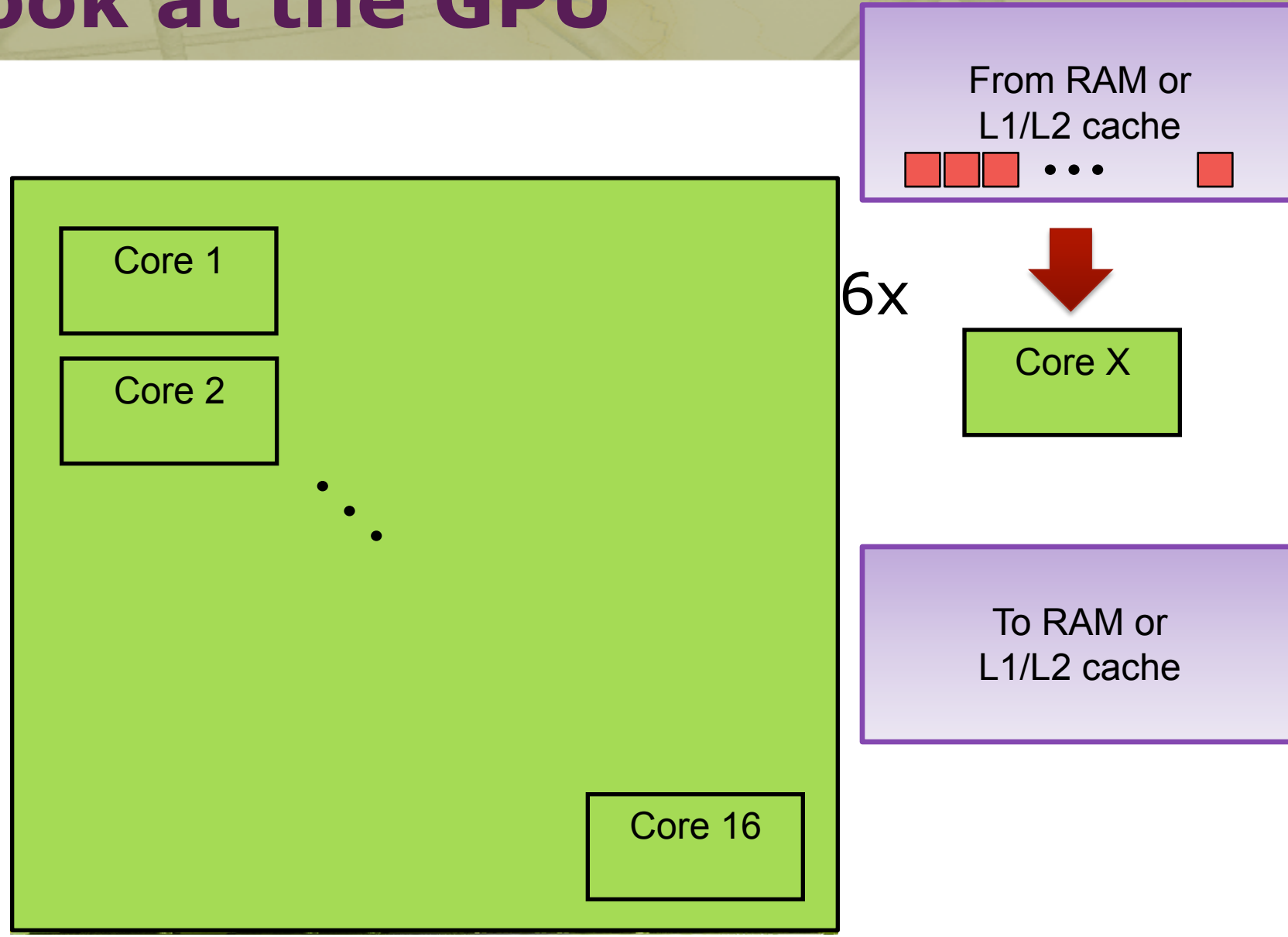
Let's look at the GPU

Each core:

- executes one program (=shader).

Each cycle:

- 192 flops
- 6x32 SIMD for up to 4 different instr.



Kepler: 15-16 multi-processors

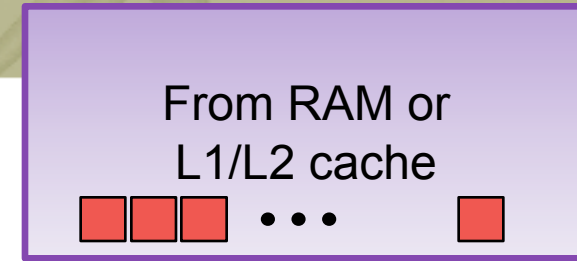
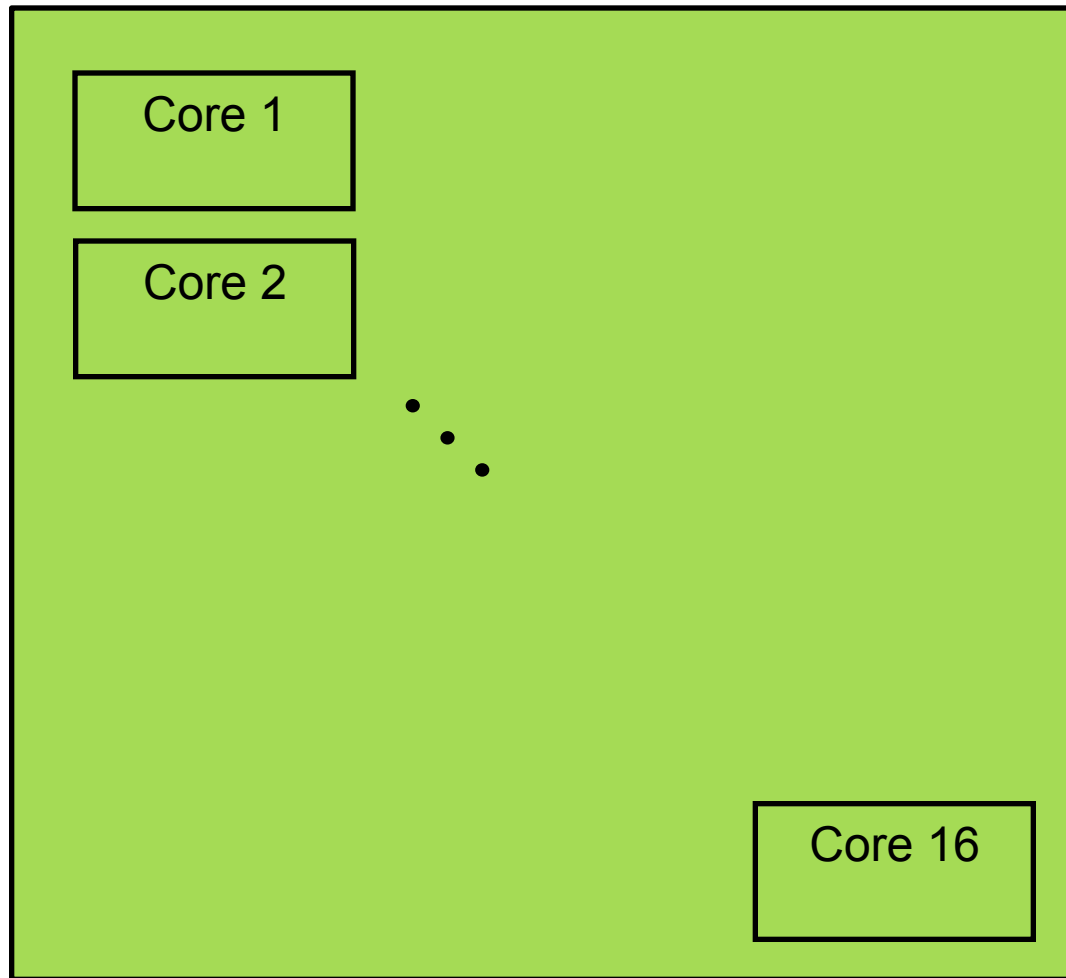
Let's look at the GPU

Each core:

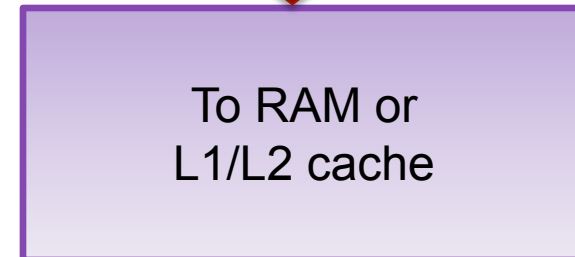
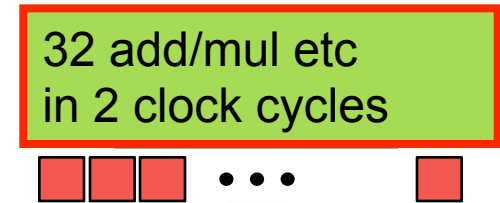
- executes one program (=shader).

Each cycle:

- 192 flops
- 6x32 SIMD for up to 4 different instr.



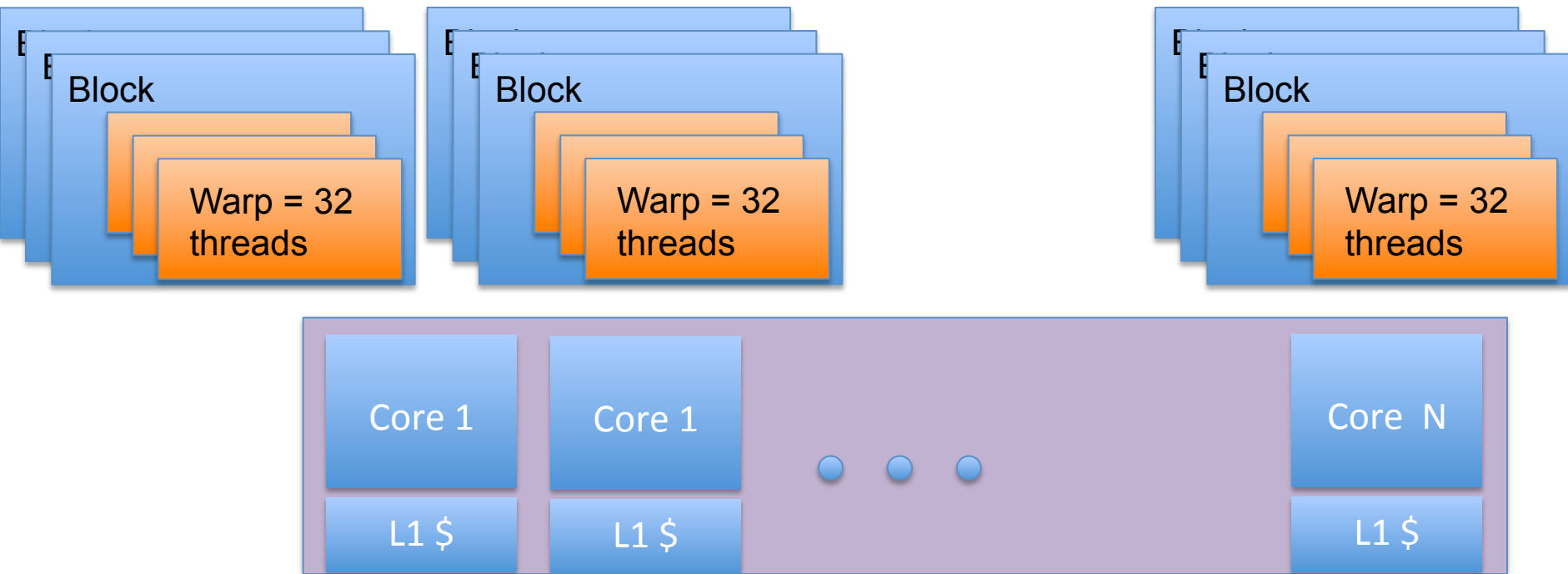
6x



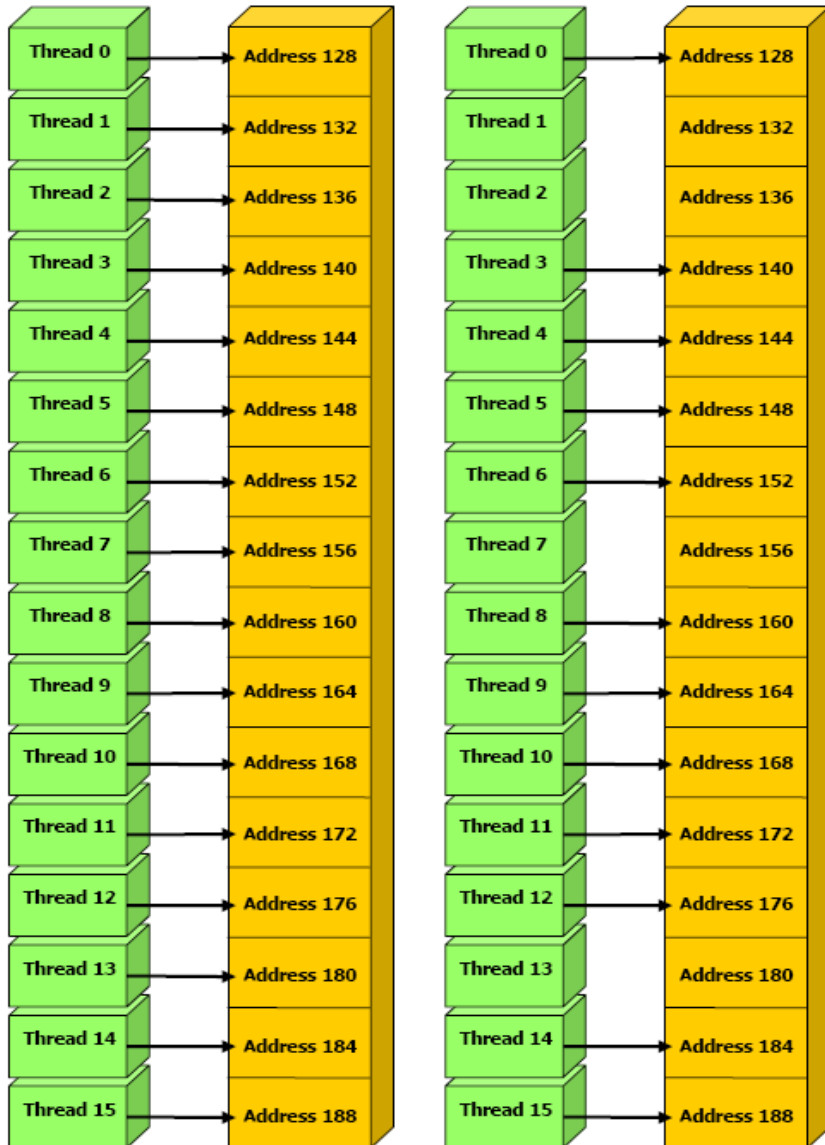
Kepler: 15-16 multi-processors

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



Memory Accesses – Global Memory



4 GB RAM

- Coalesced reads and writes
- For maximum performance, each thread should read from the same 16-float block (128 bytes)
 - i.e., the same cache-line

Fermi

- Global mem accesses.
- One transaction:
- Two transactions:

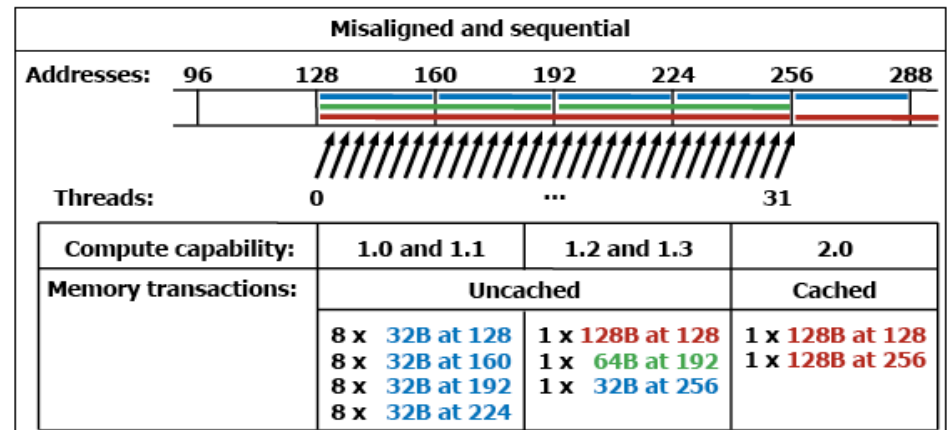
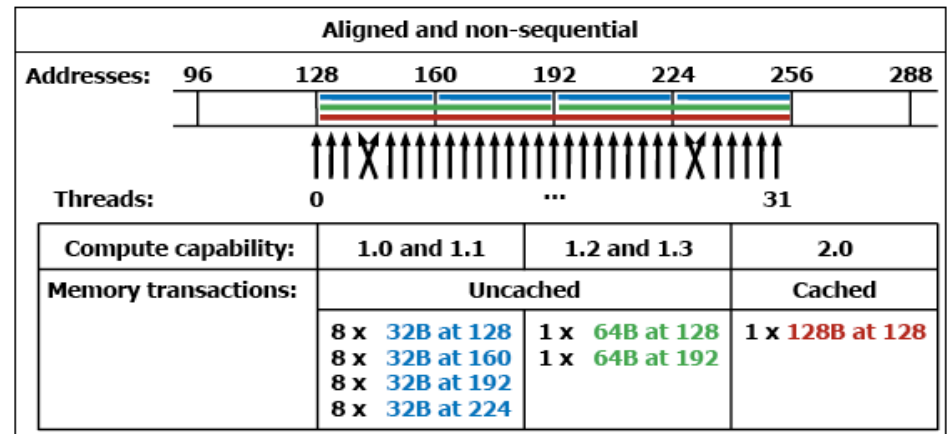
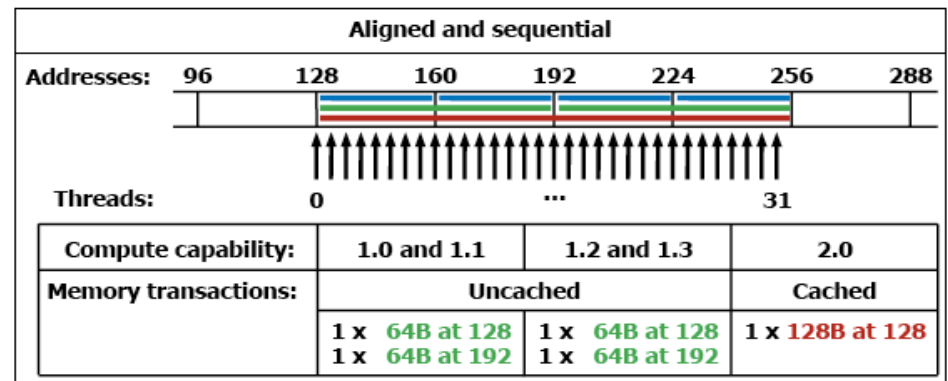
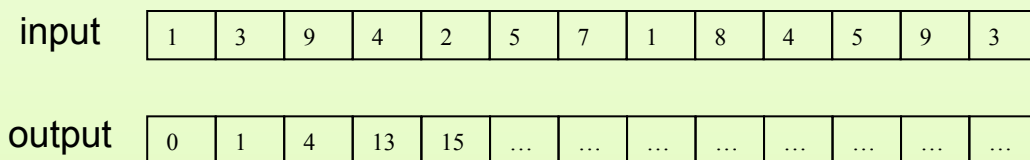
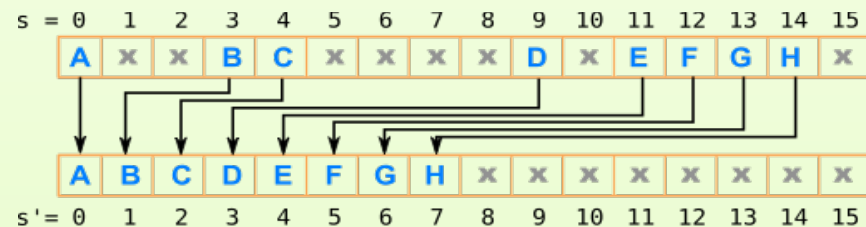


Figure G-1. Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions Based on Compute Capability

Efficient Programming

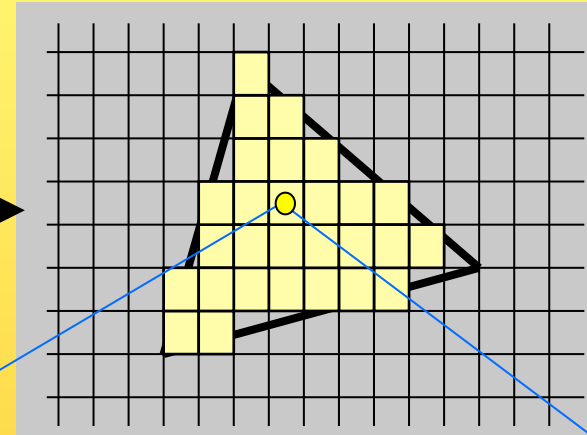
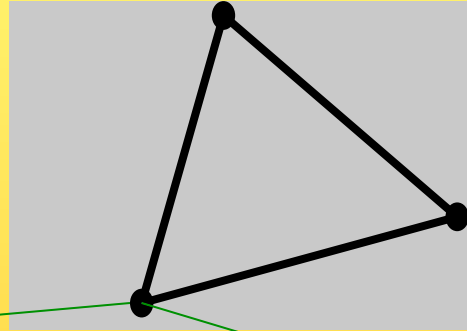
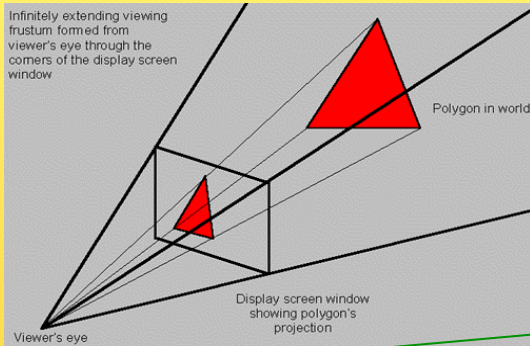
- If your program can be constructed this way, you are a winner!
- More often possible than anticipated
 - Stream compaction
 - Prefix sums
 - Sorting



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;

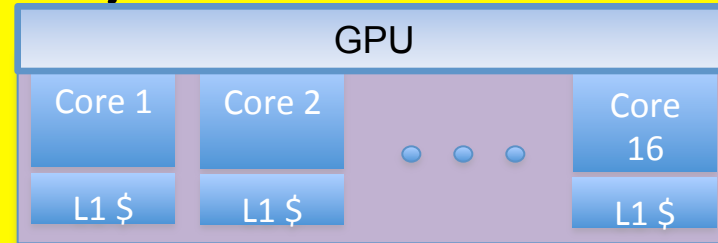
void main()
{
    gl_Position = modelViewProjectionMatrix*vec4(vertex,1);
    outColor = color;
}
```

```
// 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. 192-SIMD) executes the same instruction per clock cycle for either a:



- Vertex shader:

- E.g. 192 vertices

- Geometry shader

- E.g. 192 triangles

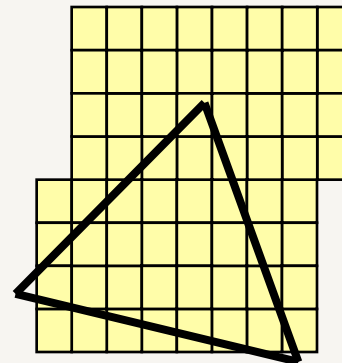
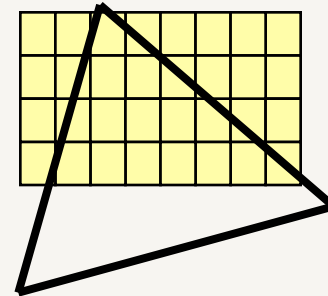
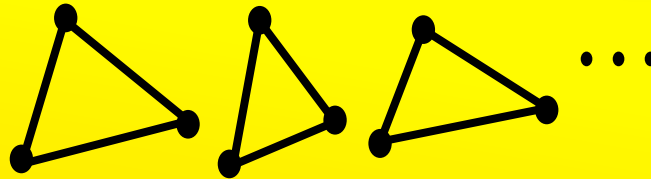
- Fragment shader:

- E.g. 192 pixels

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

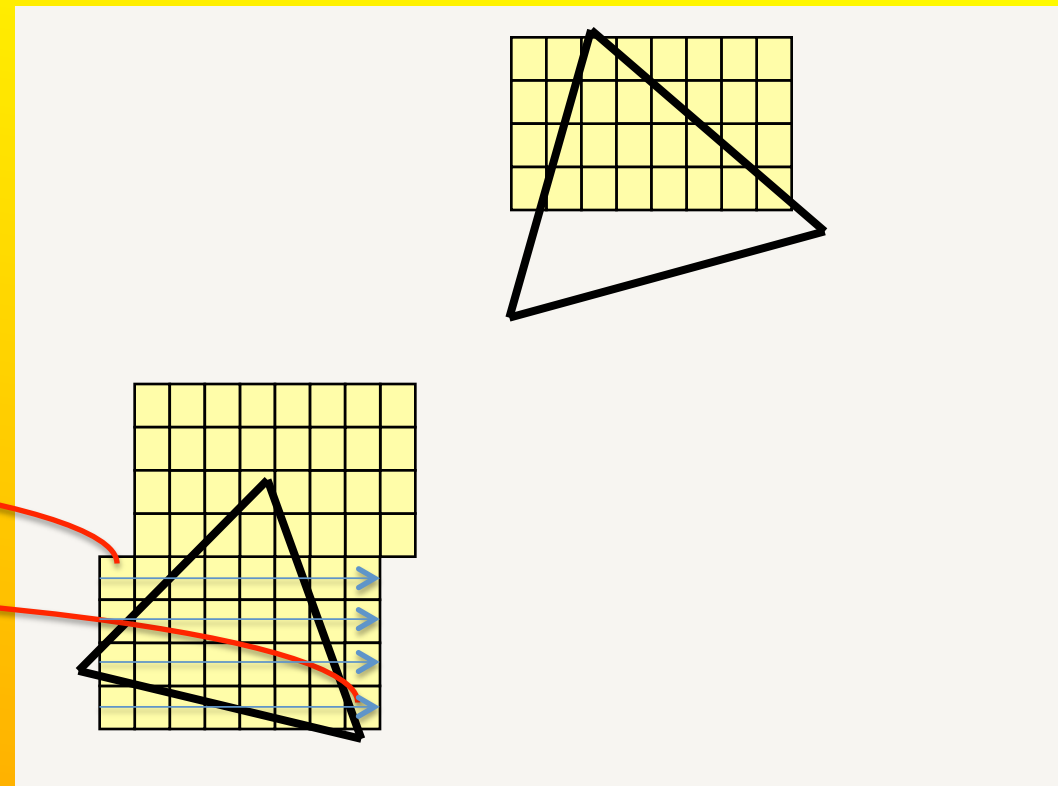
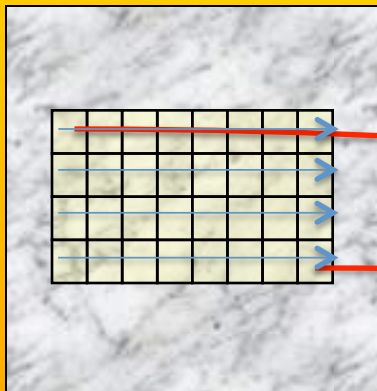
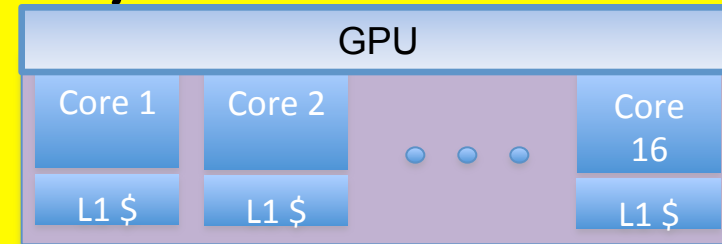
Here is an example of blocks
4x8 = 32 pixels:

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



Shaders and coalesced memory accesses

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



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.

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
 - Texture caching with prefetching
 - Texture compression
 - Z-compression
 - Z-occlusion testing (HyperZ)

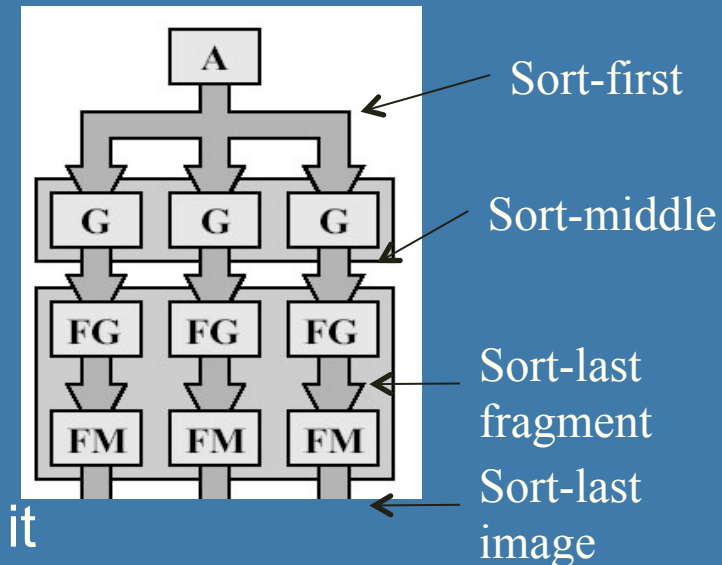
- Be able to sketch the architecture of a moder graphics card

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.



Need to know:

