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 architetures, and few commercial products
  - Renderdrive, RPU, (Gelato), NVIDIA RTX (via OptiX or DXR api)
  - Or write your own GPU ray-tracer

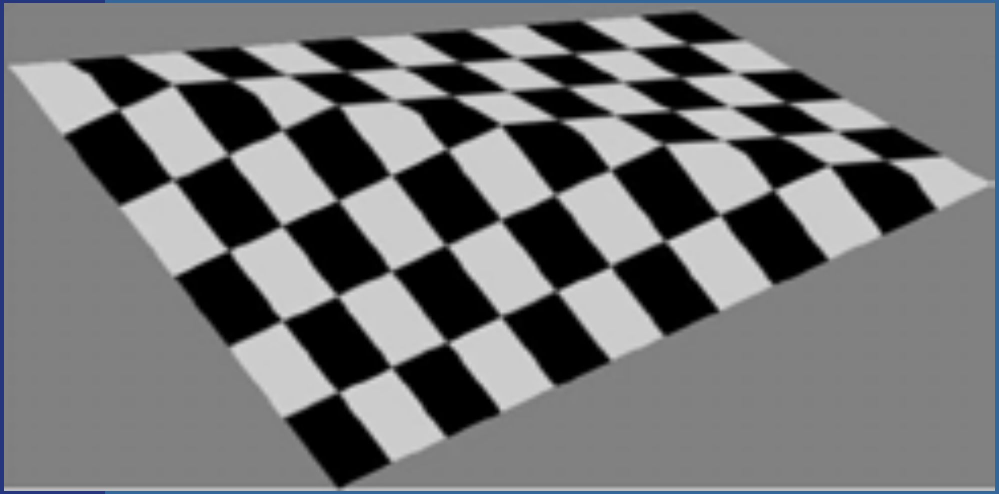
Perspective-correct interpolation of texture coordinates

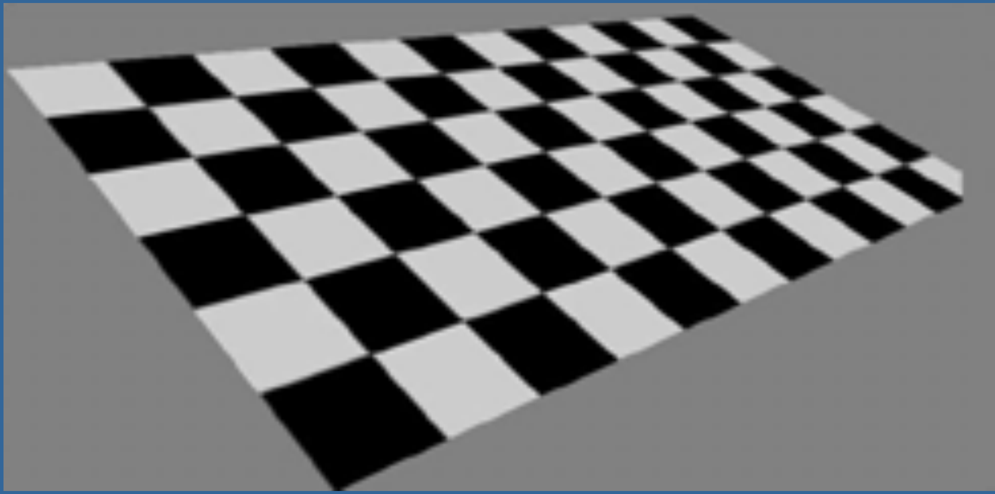
(and actually all screen-space-interpolated per-vertex data)

DELI - GROCERY
Steel Monkeys
CLASS 1
Game product
PS3 PORTAL

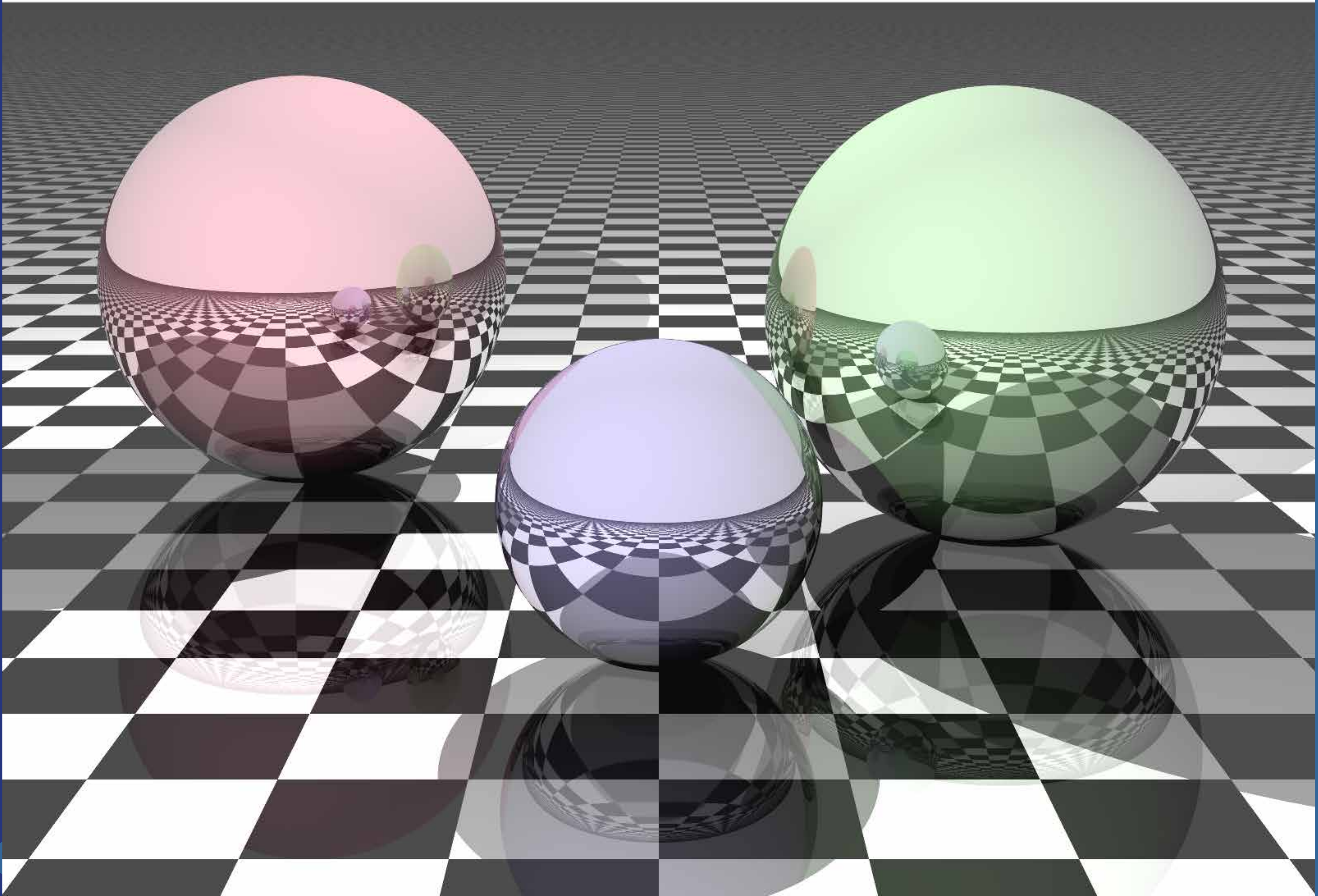
Perspective-correct texturing

- 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: $\mathbf{p} = \mathbf{Mv}$
  - 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

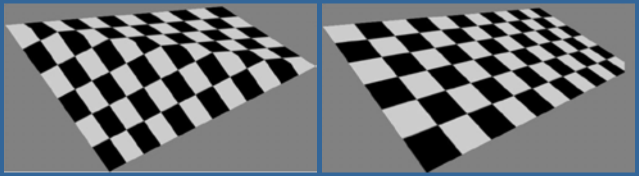
$$\mathbf{p} = \mathbf{Mv} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ -v_z/d \end{pmatrix}$$

Mathematic derivation:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3.211&rep=rep1&type=pdf

Texture coordinate interpolation

- Linear interpolation does not work
- Rational linear interpolation does:
  - $u(x)=(ax+b)\,/\,(cx+d)$ (along a scanline where y=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: $(\mathrm{w}*u/w,\mathrm{w}*v/w)=(u,v)$
  - The (u,v) are perspectively-correct 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
- 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
    - 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).
    - 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:
      - $w_i = 1/(1/w)_i$
      - $u_i = (u/w)_i * w_i$
      - $v_i = (v/w)_i * w_i$

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

Overview of GPU architecture

-History / evolution

- GPU design: Several **cores** consisting of many **ALU**s
  (NVIDIA terminology: **Streaming Multiprocessors (SMMs)** of many **cores**
- GPU vs CPU

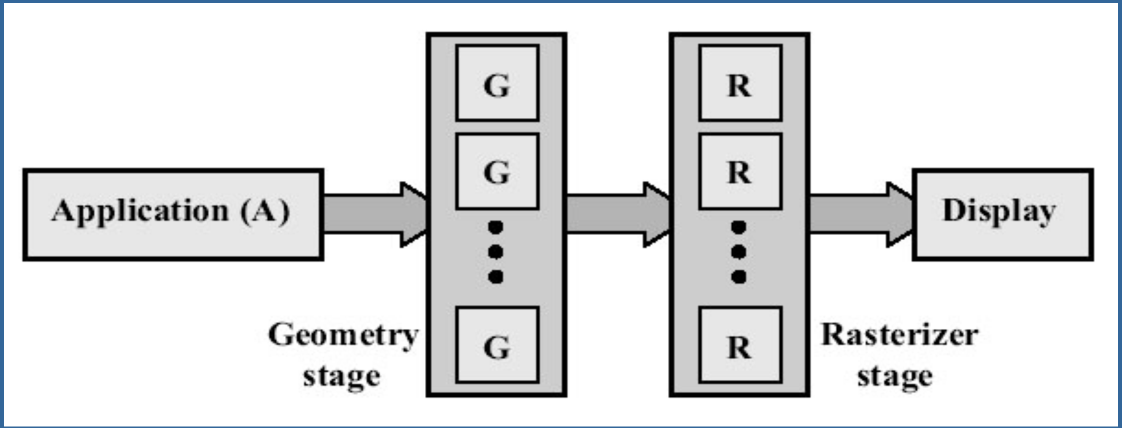
Take-away: bandwidth (cost of memory accesses) is a major problem

Background: Graphics hardware architectures

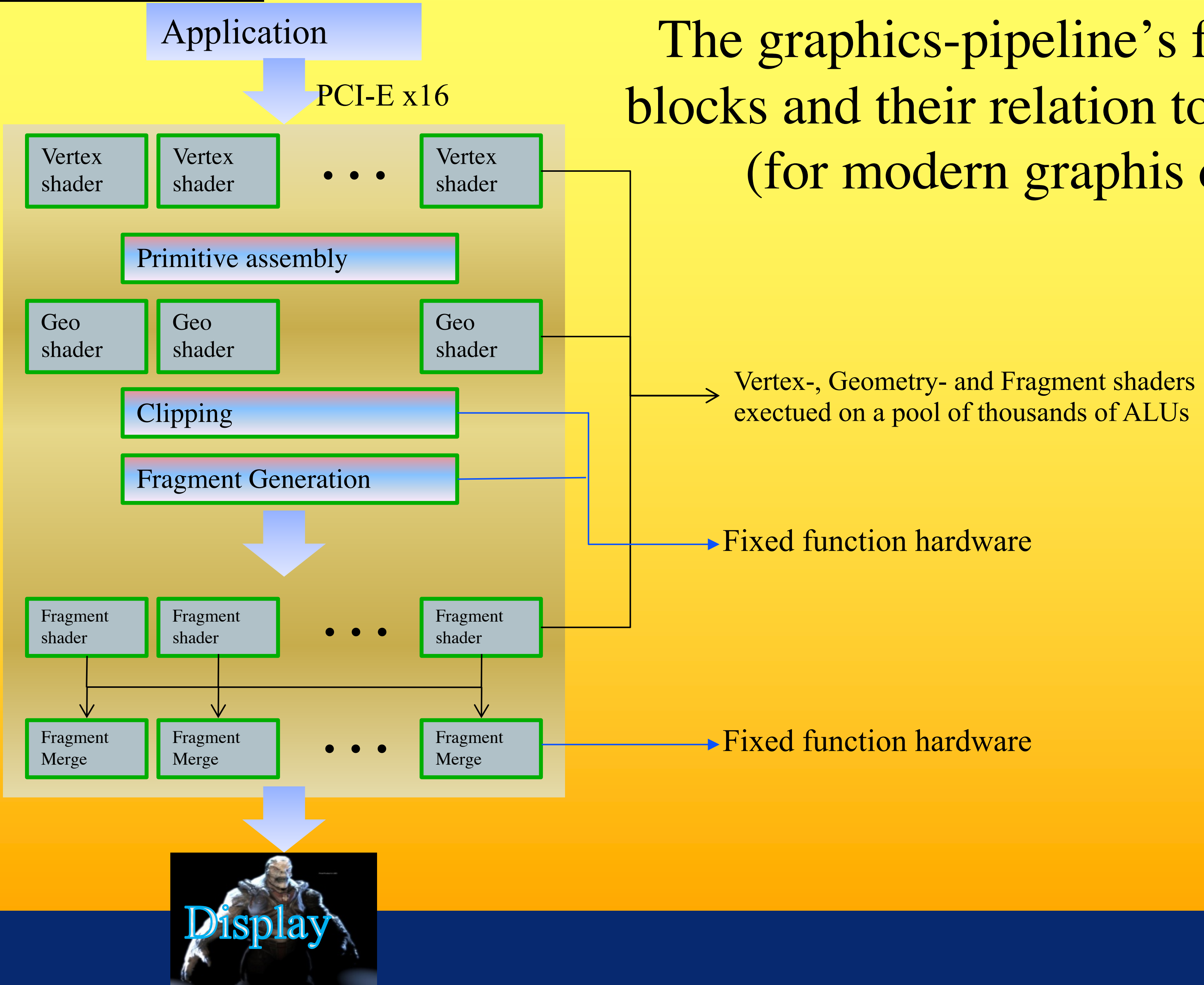
- Evolution of graphics hardware has started from the end of the pipeline
  - Rasterizer was put into hardware first (most performance to gain from this)
  - Then the geometry stage
  - Application will not be put into GPU hardware (?)
- Two major ways of getting better performance:
  - Pipelining
  - Parallellization
  - 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 parallellize both geometry and rasterizer stage:

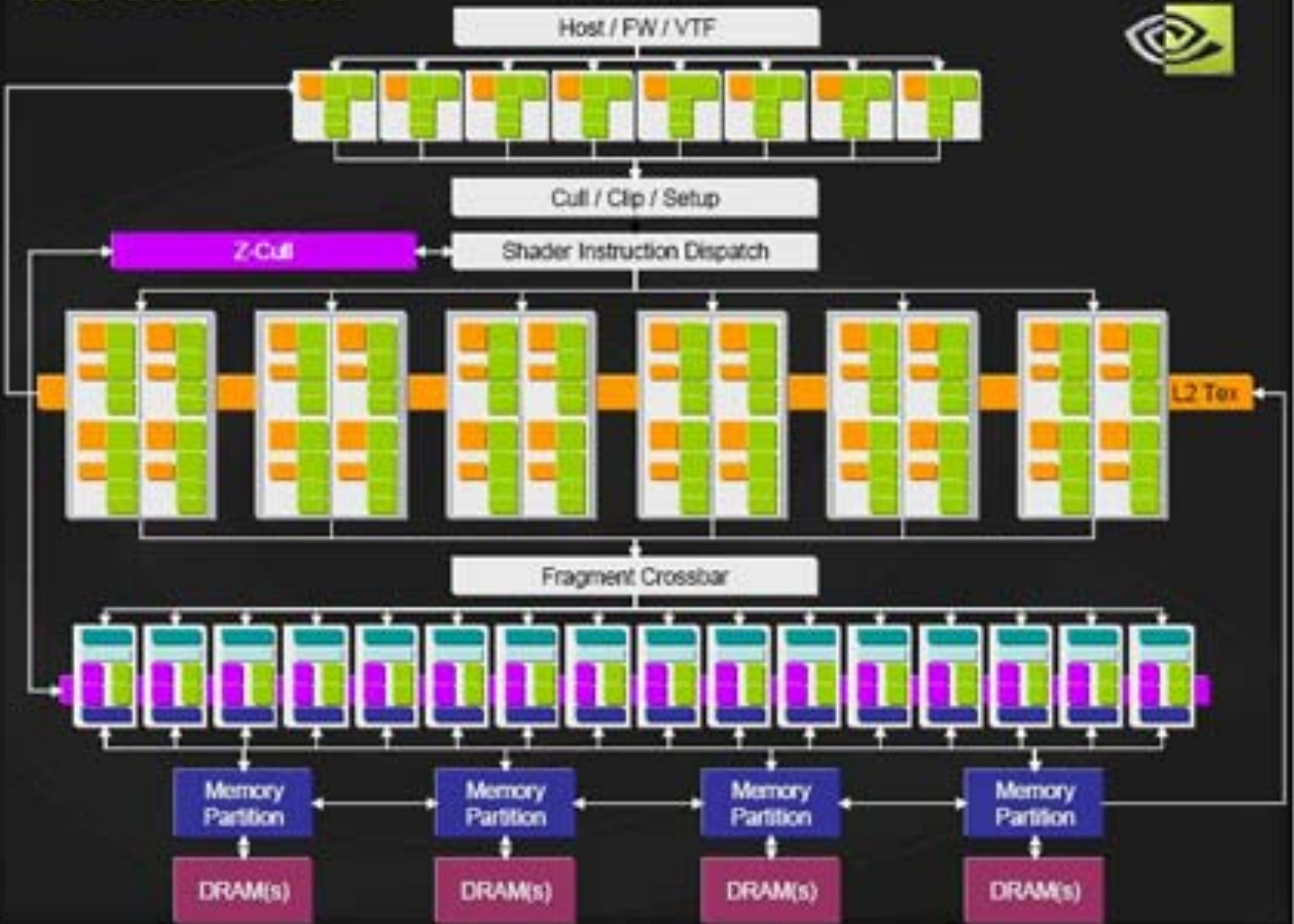


The graphics-pipeline's funcional blocks and their relation to hardware (for modern graphis card)

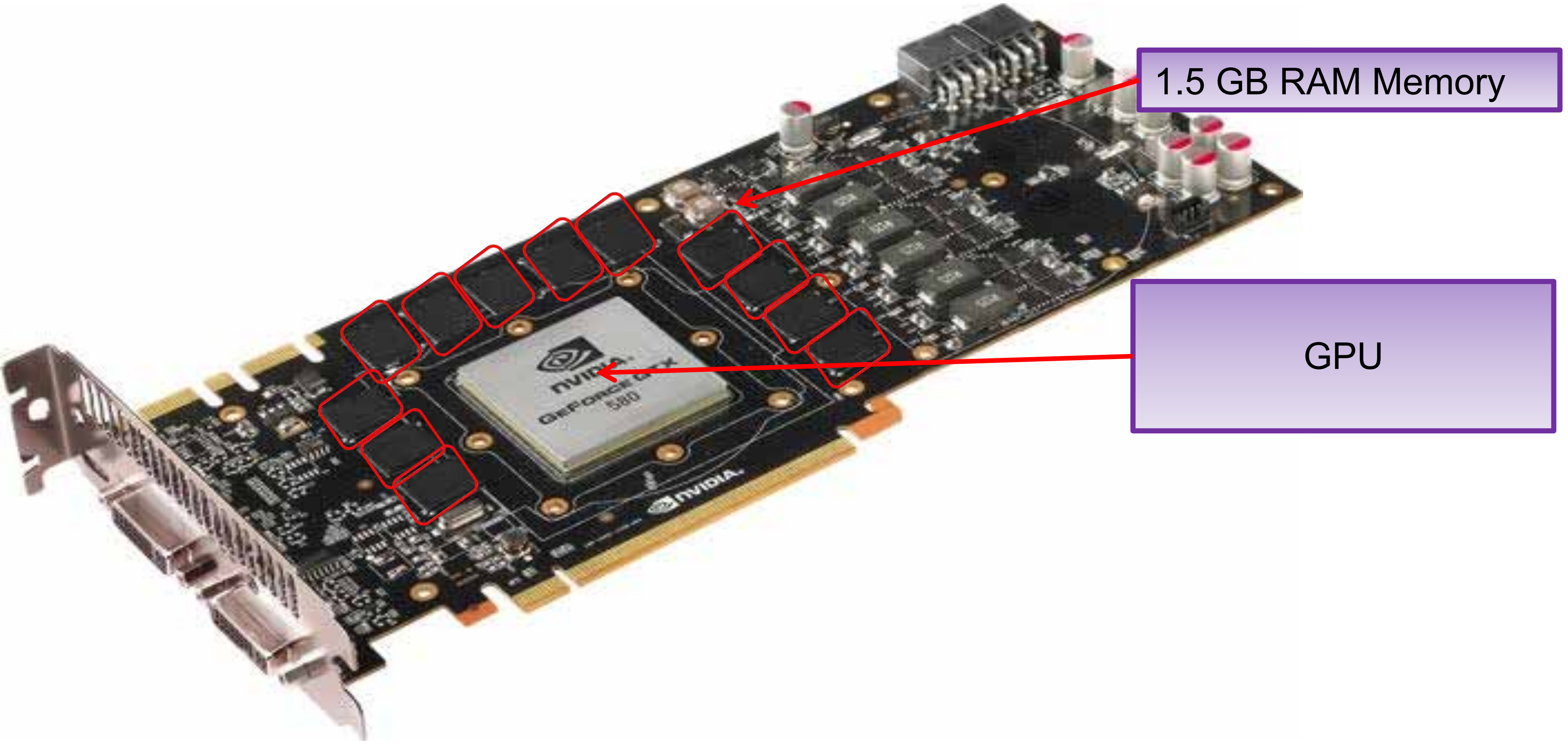


GeForce 7800

Older architecture (2006)



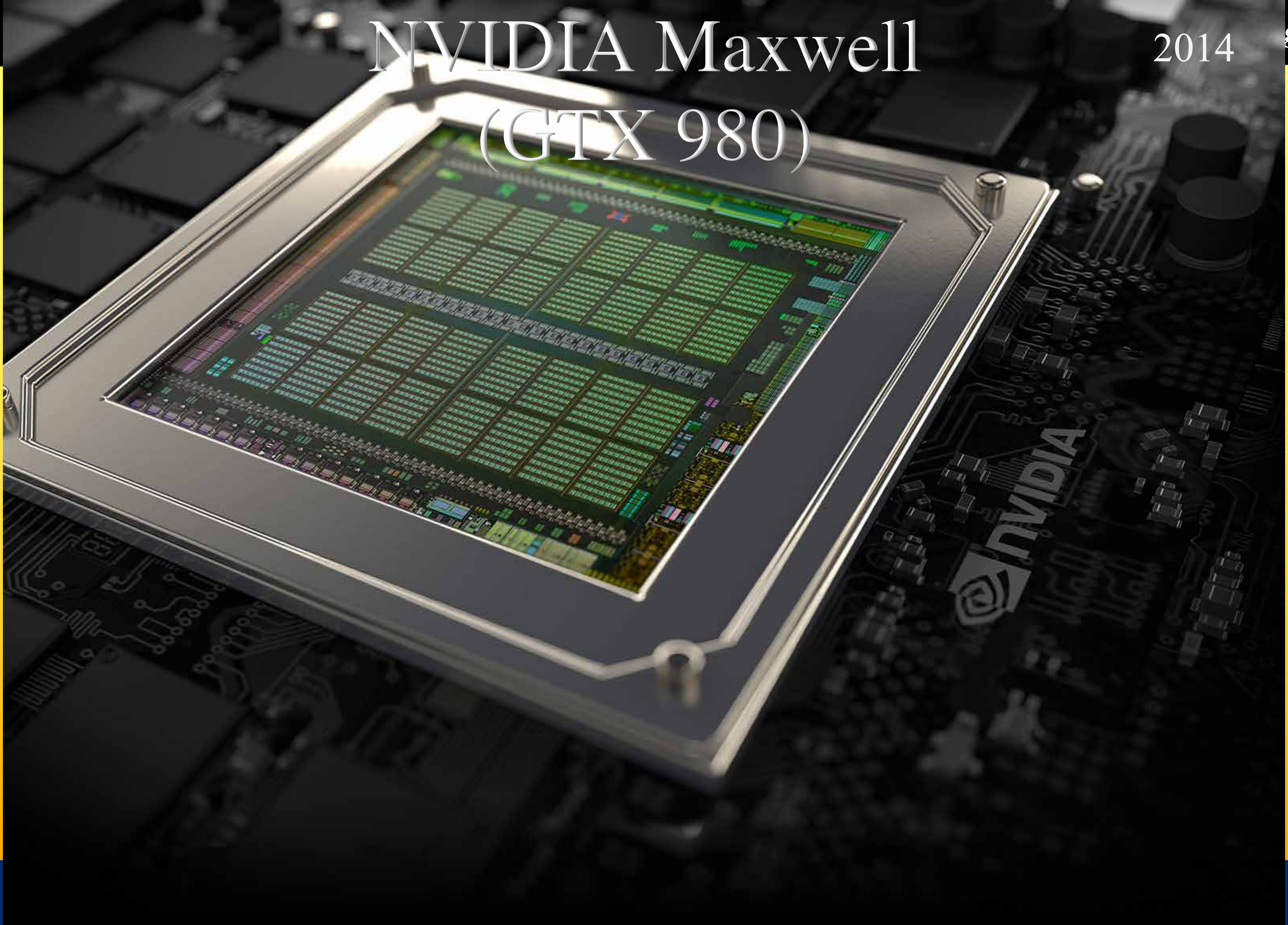
Graphics Processing Unit - GPU



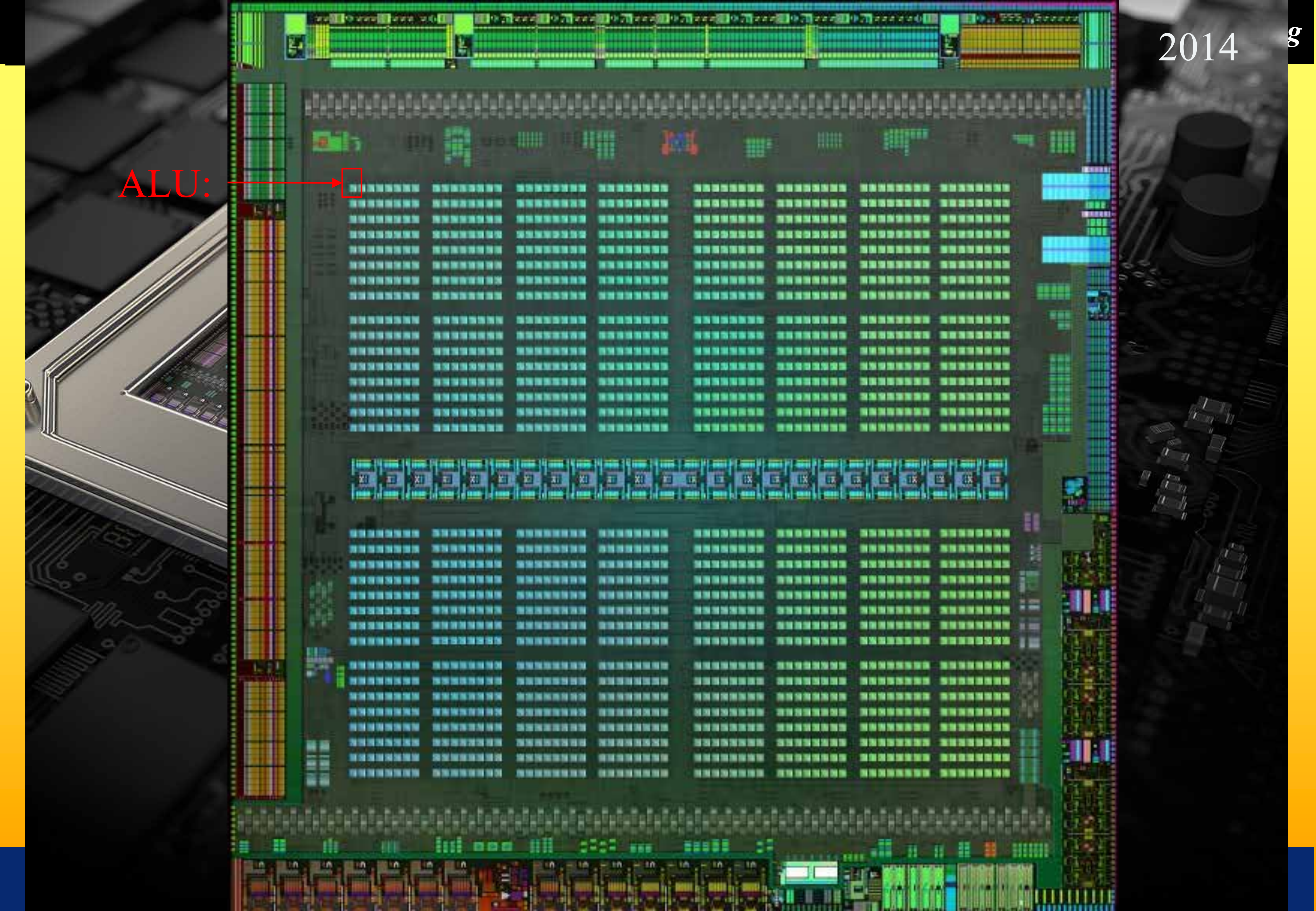
- NVIDIA Geforce GTX 580

NVIDIA Maxwell (GTX 980)

2014



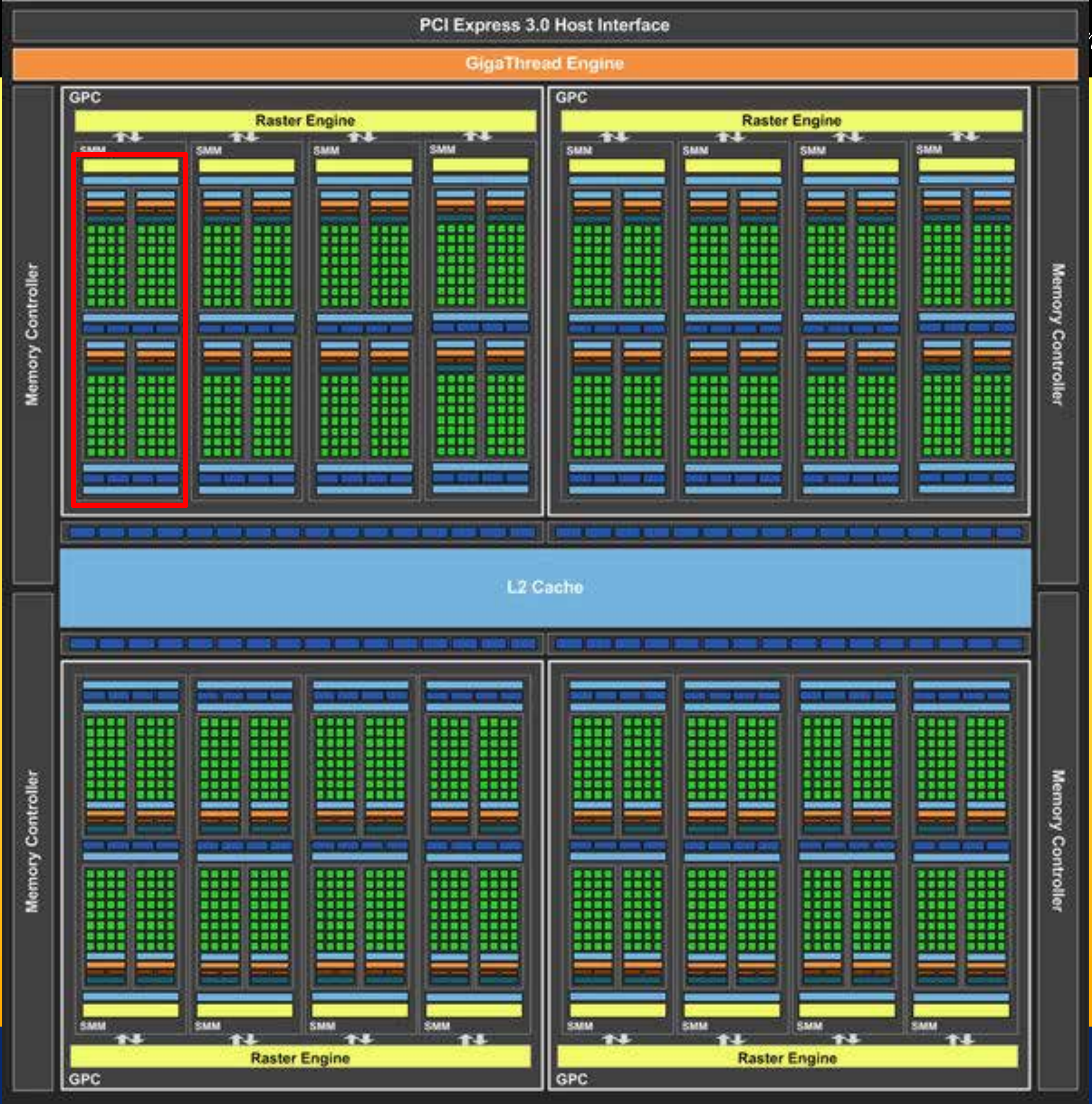
2014



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

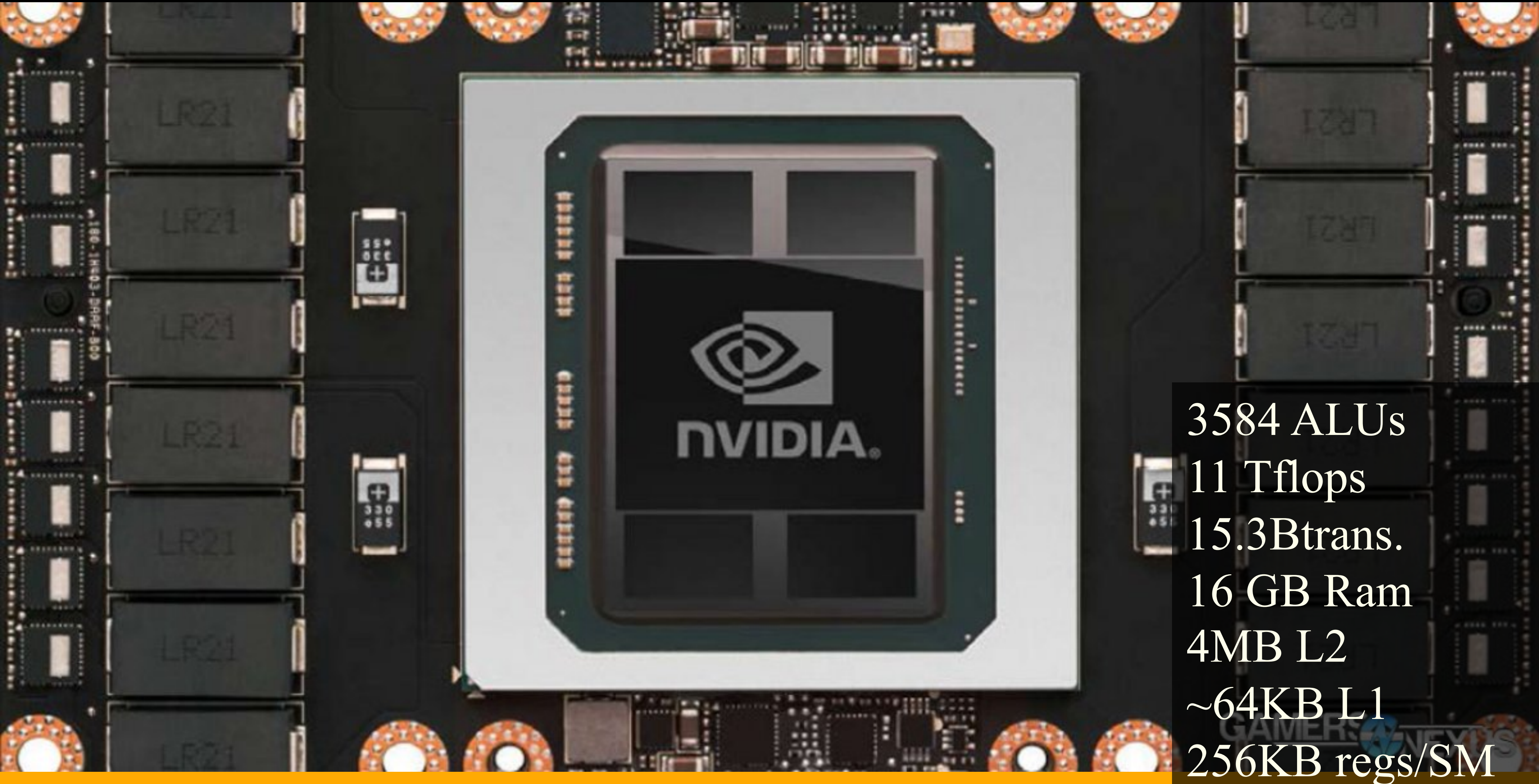
Each Core:

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



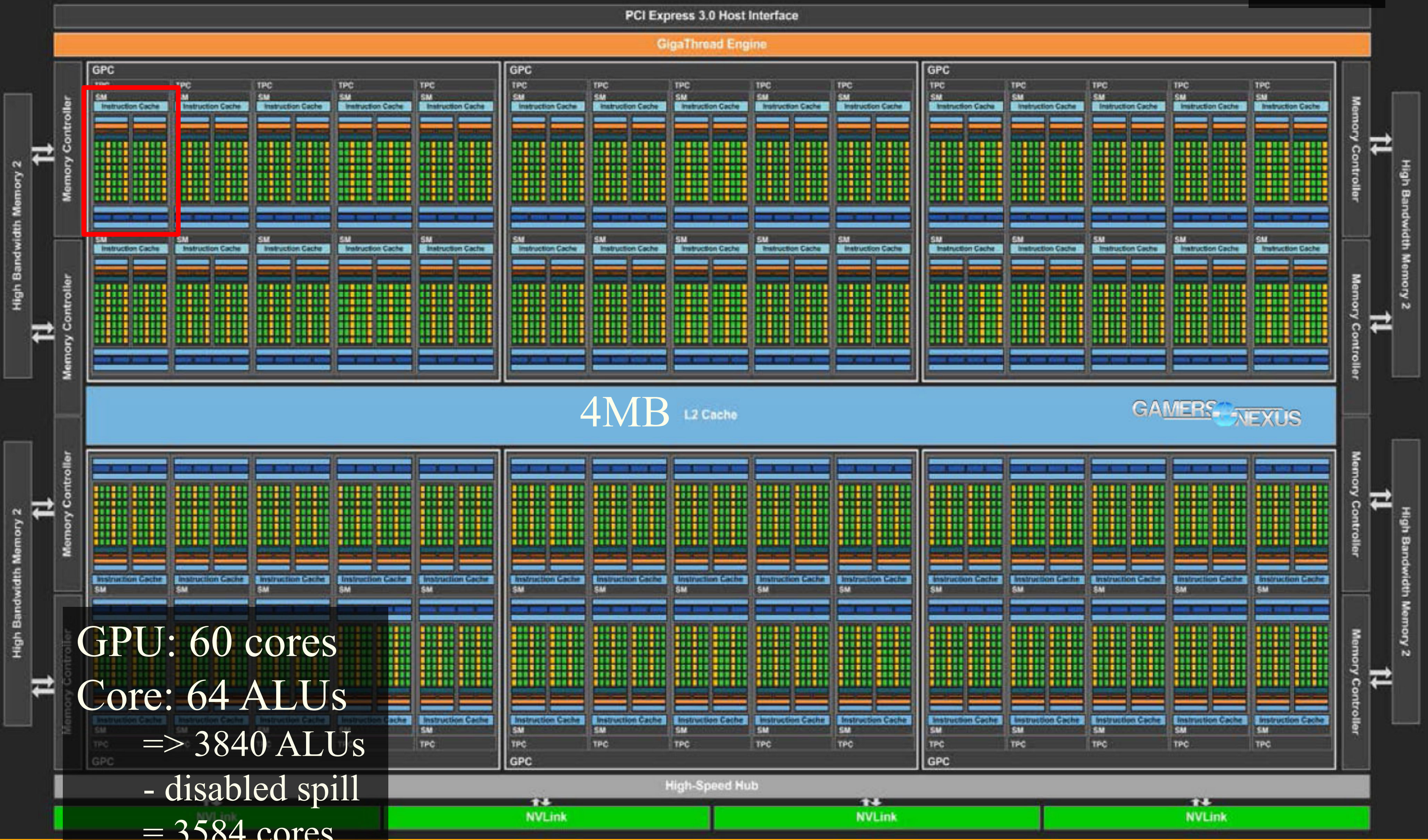
NVIDIA Pascal GP100 (GTX 1080 / Titan X)

2016



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

2016



GPU: 60 cores

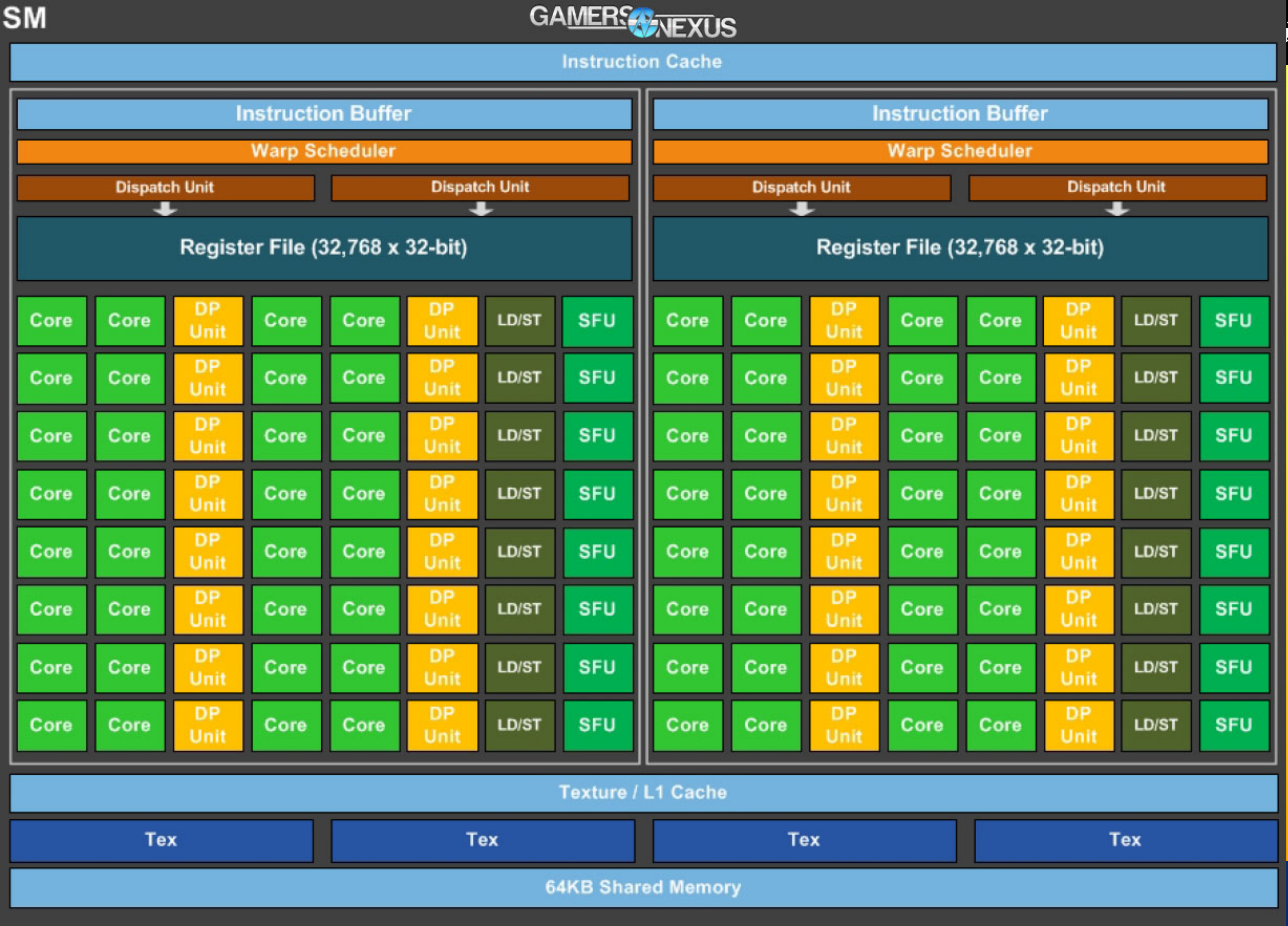
Core: 64 ALUs

=> 3840 ALUs

- disabled spill

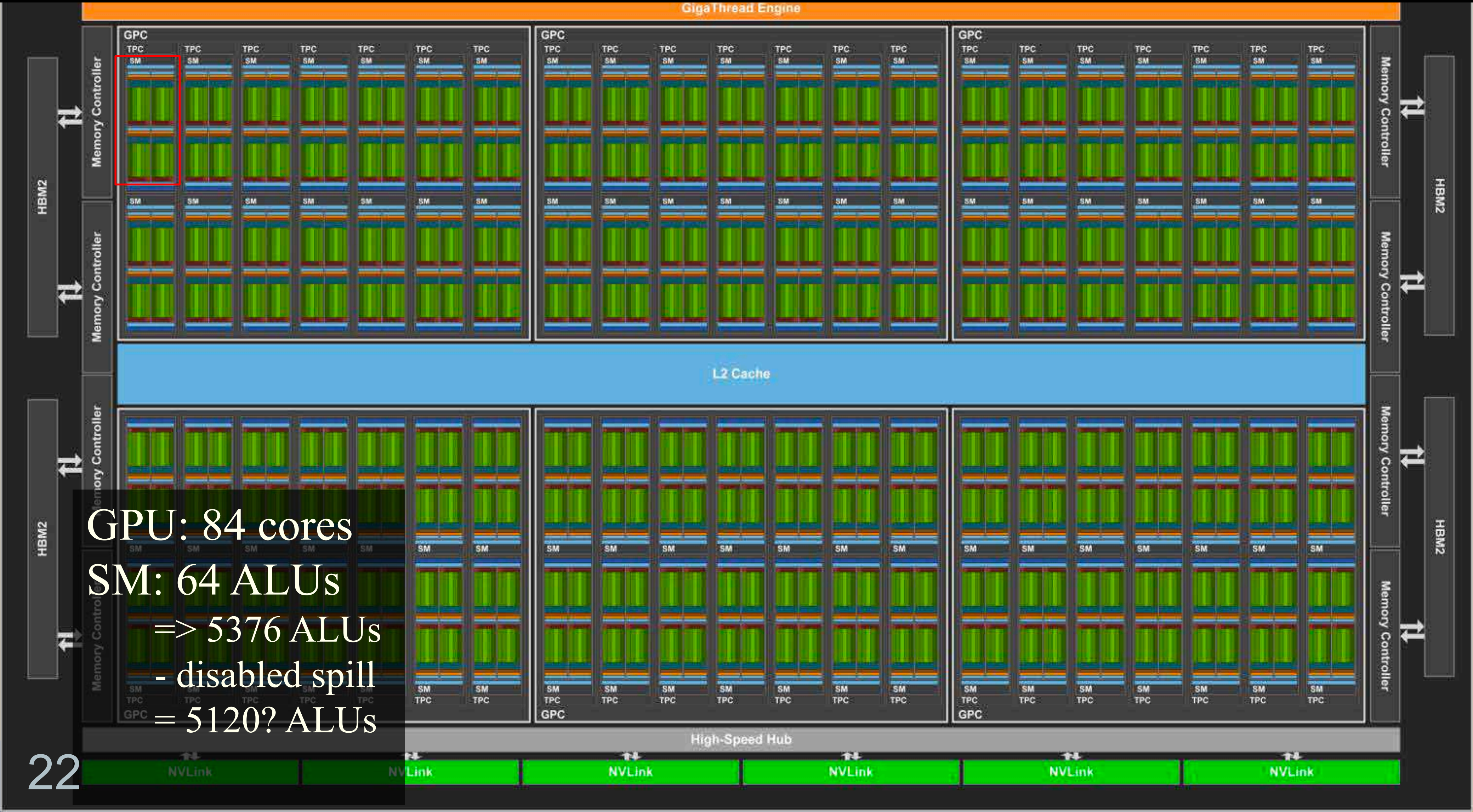
= 3584 cores

ing


SM
GAMERS NEXUS
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
Core
DP Unit
LD/ST
SFU
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
Core
DP Unit
LD/ST
SFU
Texture / L1 Cache
Tex
Tex
Tex
Tex
64KB Shared Memory

NVIDIA Volta GV100

2018 (Dec. 2017)



GPU: 84 cores

SM: 64 ALUs

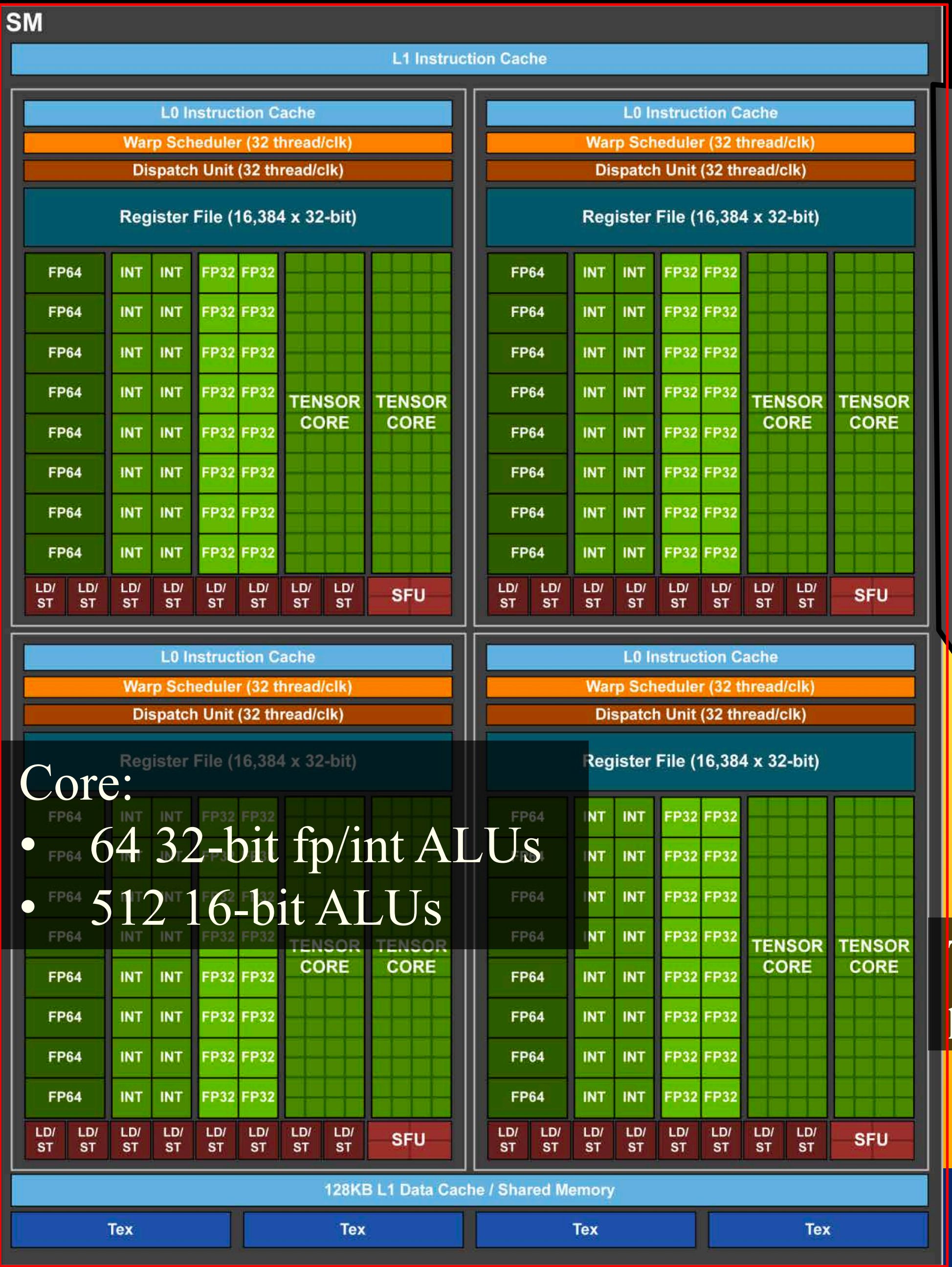
=> 5376 ALUs

- disabled spill

= 5120? ALUs

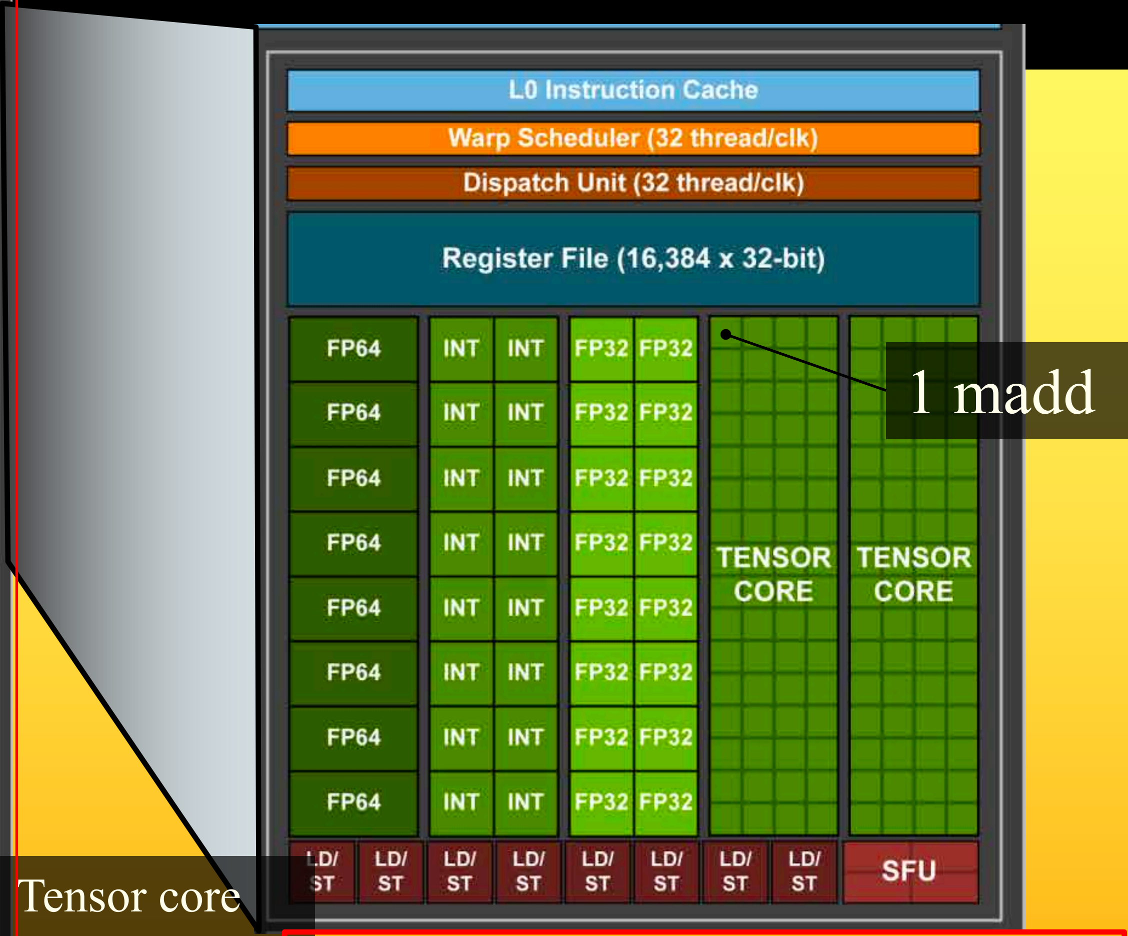
NVIDIA Volta GV100

2018

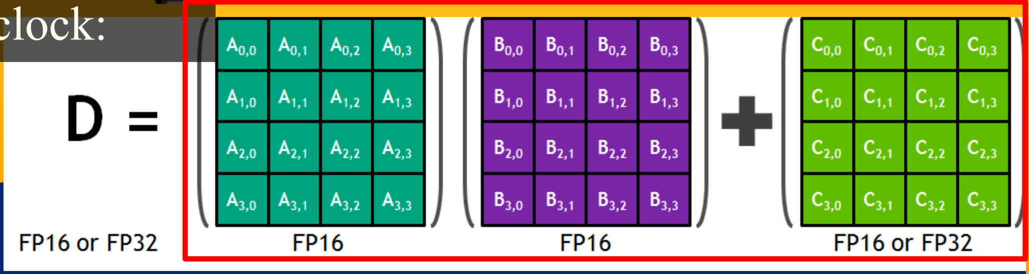

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Core:

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


L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
1 madd

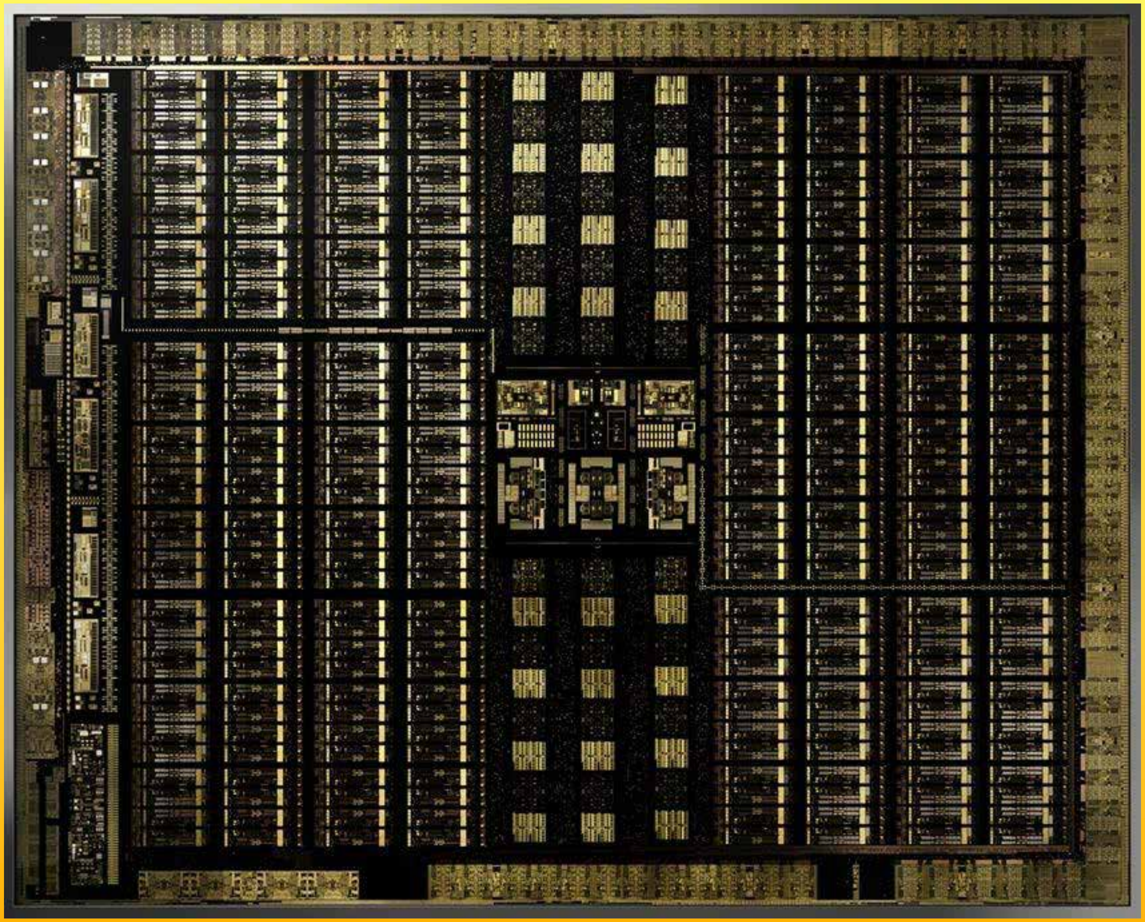
Tensor core per clock:



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

NVIDIA Turing TU102

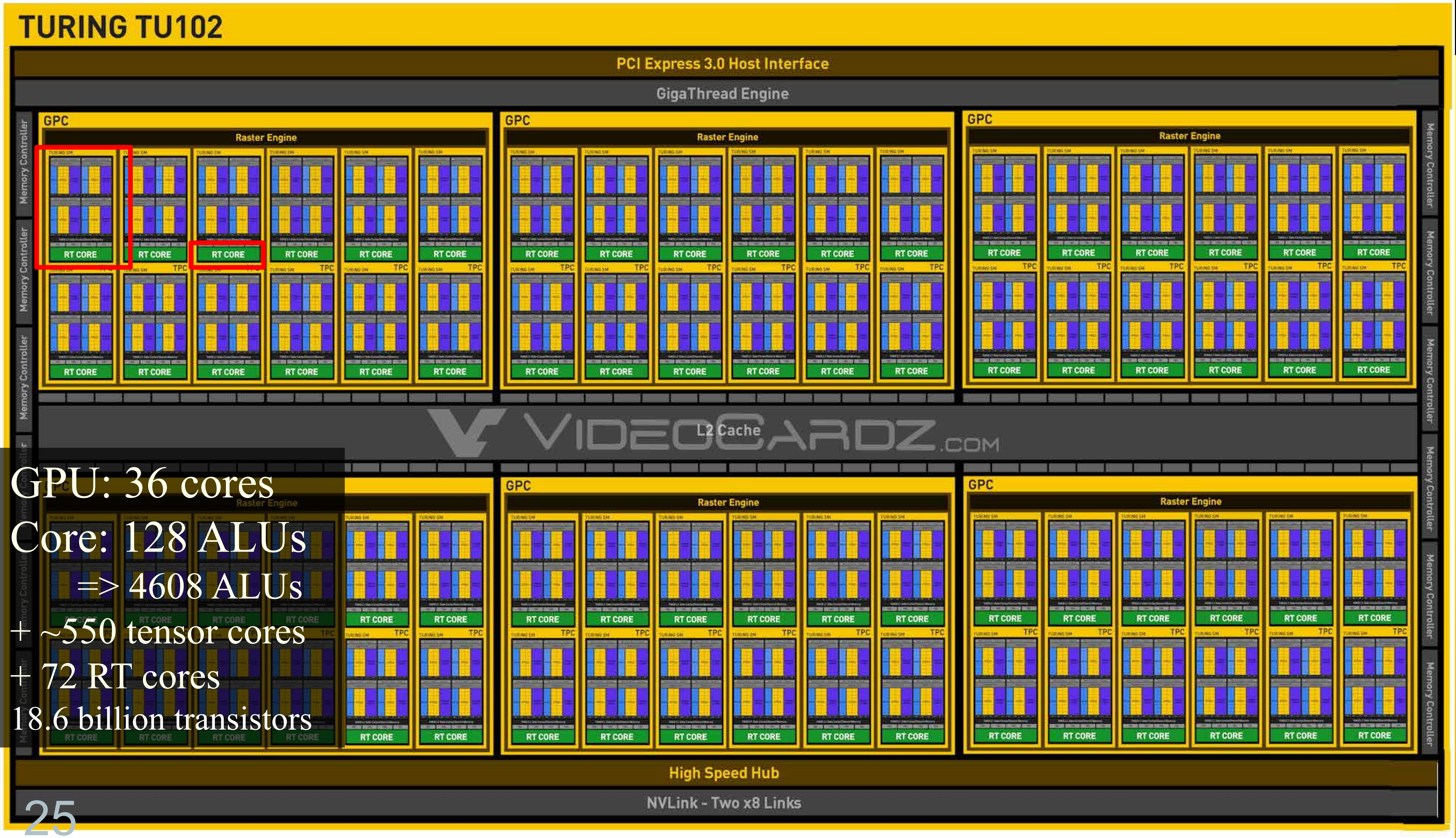
2018



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

NVIDIA Turing TU102

2018



GPU: 36 cores

Core: 128 ALUs

=> 4608 ALUs

+ ~550 tensor cores

+ 72 RT cores

18.6 billion transistors

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
- 09' Tesselation Shaders (Direct3D '09, OpenGL '10)

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

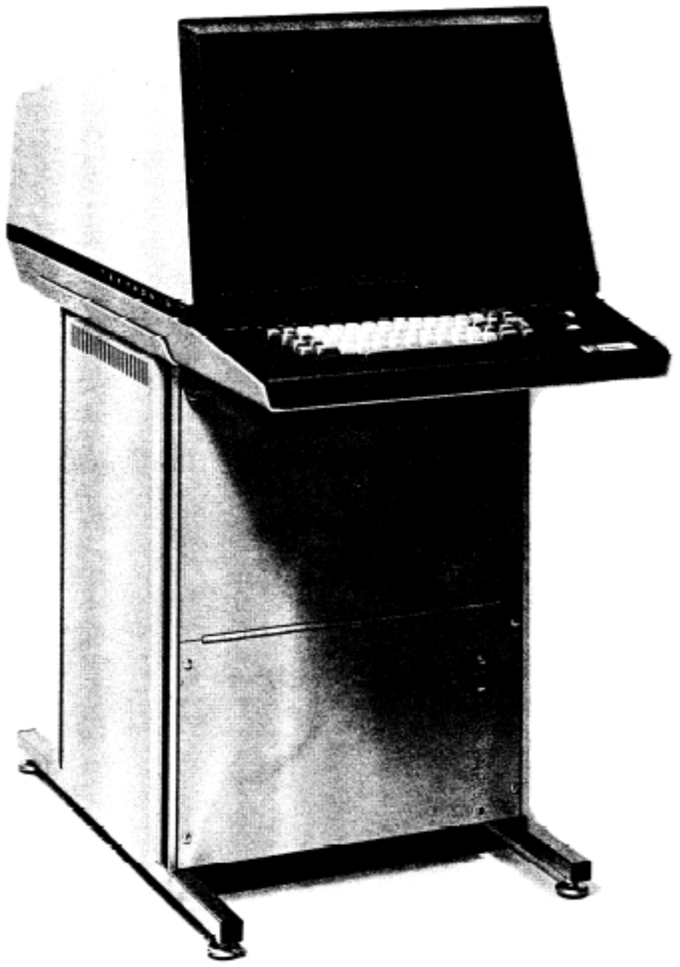


Fig. 1-1. 4014 Computer Display Terminal.

Tektronix **4014**

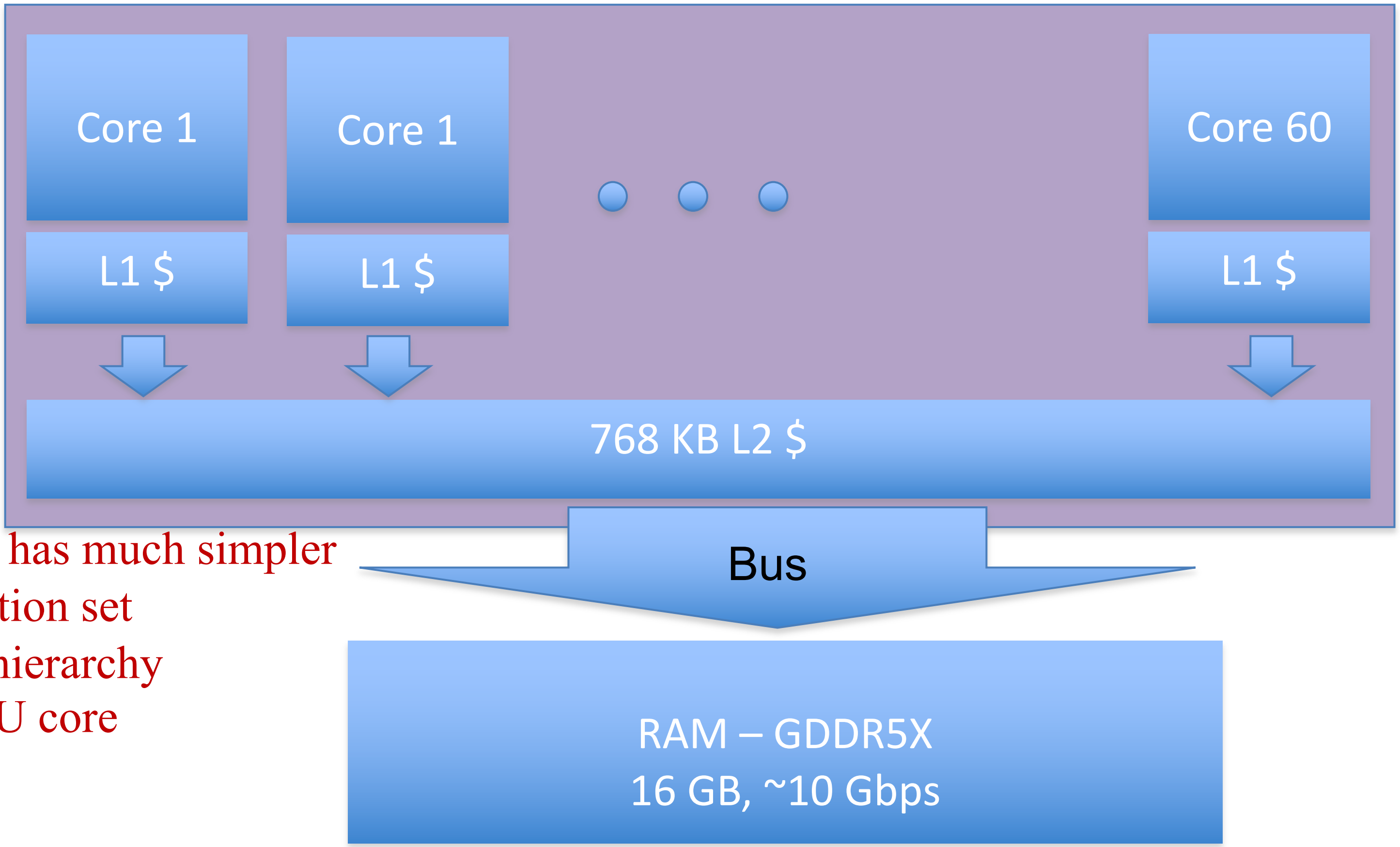
Graphics Hardware History

- 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 MHz), GDDR3
  - 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
  - 2018 Nvidia Volta: 21.1Gtrans, core clock: ~1500MHz, 900GB/s, Mem: 4096-bit HBM2, (or GDDR6)

  Lesson learned: #trans doubles ~per 2 years. Core clock increases slowly. Mem clock –increases with new technology DDR2, DDR3, GDDR5, HBM2 and with more memory busses (à 64-bit). Now stacked.
  - 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 often the bottleneck

GPU- Nvidia's Pascal 2016

Overview:



60 cores à 64-SIMD width (2*4*8)

~64 KB per each 64 SIMD

GPU core has much simpler

- instruction set
- cache hierarchy

than a CPU core

Bandwidth ~480 GB/s

Bus: 256/384/4096 bits

Compare to ATI 2900:
- 2x512bits

Larrabee:
- 2x512bits

Wish:

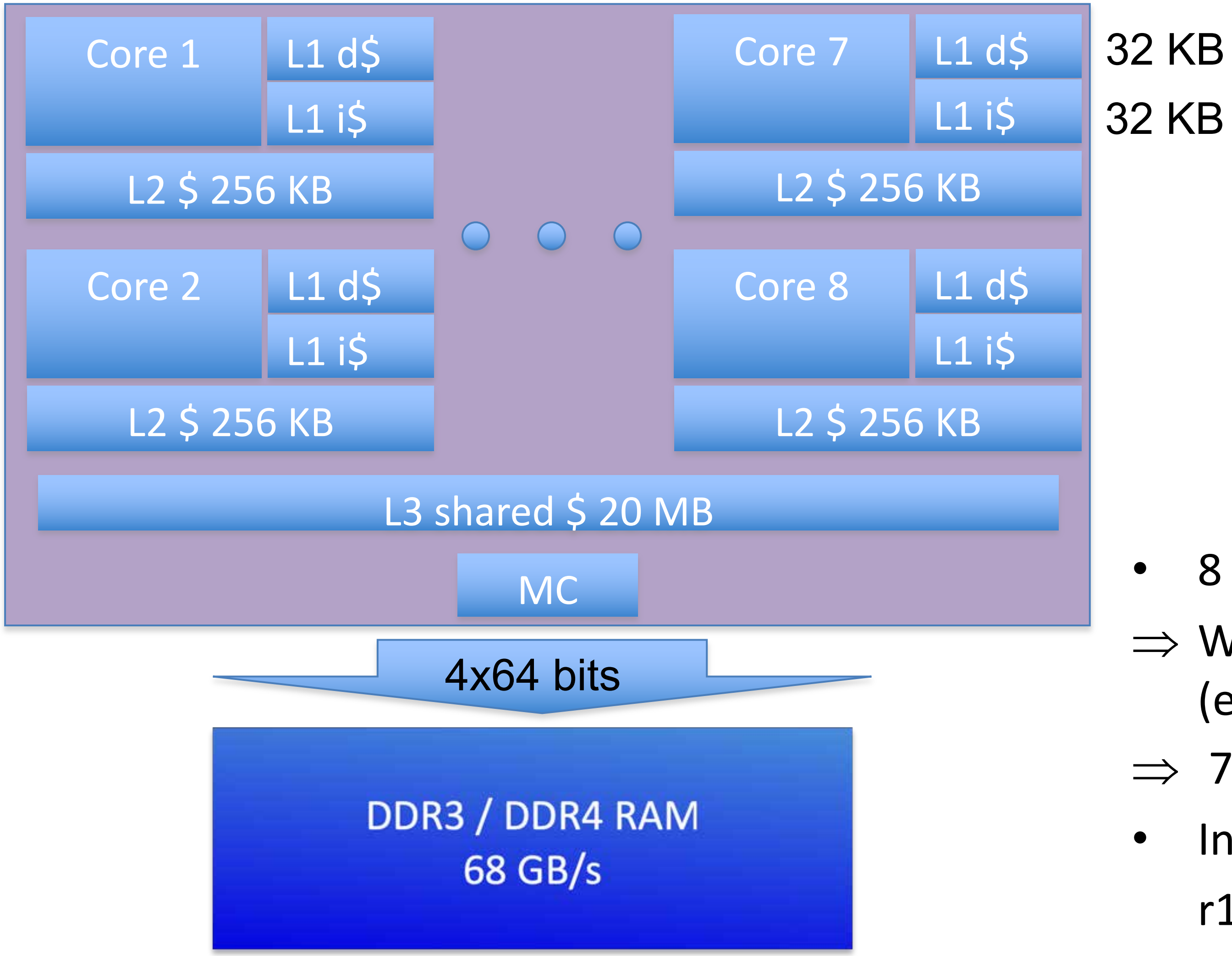
3584 ALUs à 1 float/clock => 14KB/clock

~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 – 2014-2016

Intel's Sandybridge / Haswell / Broadwell



1 – 8 cores à 8 SIMD floats

- 8 cores à 8 floats

⇒ We want 256 bytes/clock (e.g. from RAM)

⇒ 768 GByte/s, 3GHz CPU

- In addition, x2, since:

  r1 = r2 + r3;

In reality: 30-68 GB/s

Solved by $-hierarchy + registers + thread switching

Memory bandwidth usage is huge!!

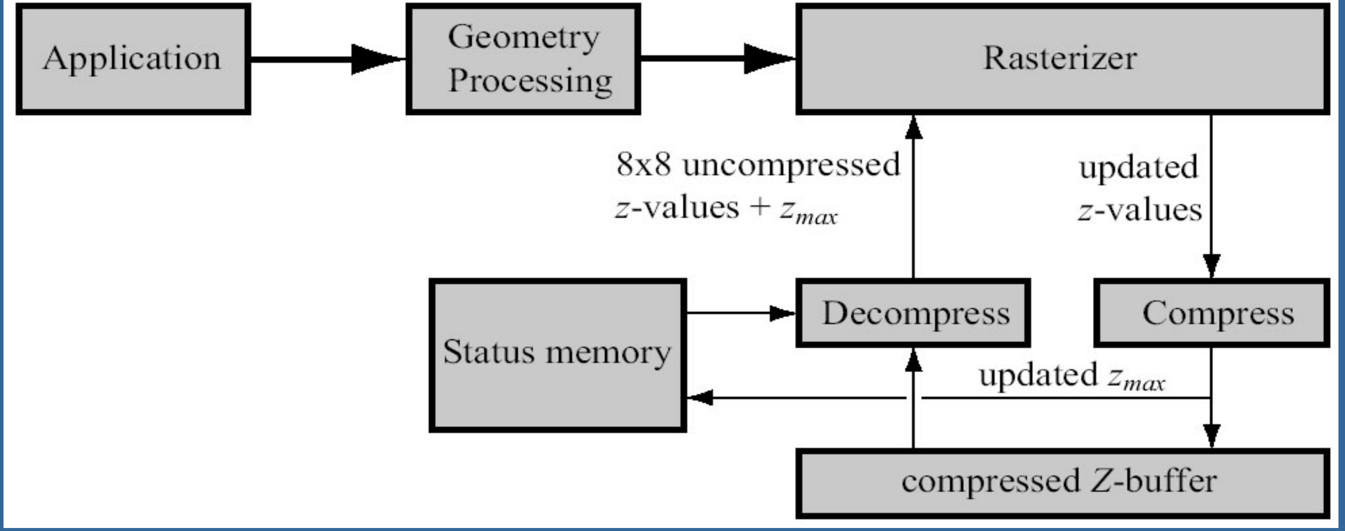
- On top of that bandwith usage is never 100%.
- However, there are many techniques to reduce bandwith usage:
  - Texture caching with prefetching
  - Texture compression
  - Z-compression
  - Z-occlusion testing (HyperZ)

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

Bonus

Architecture of Z-cull and Z-compress



- Store zmax per tile, and a flag (whether cleared, compressed/uncompressed)
- Rasterize one tile at a time
- Test if zmin on triangle is farther away than tile's zmax
  - If so, don't do any work for that tile!!!
  - Saves texturing and z-read for entire tile – huge savings!
- Otherwize read compressed Z-buffer, & unpack
- Write to unpacked Z-buffer, and when finished compress and send back to memory, and also: update zmax
- 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

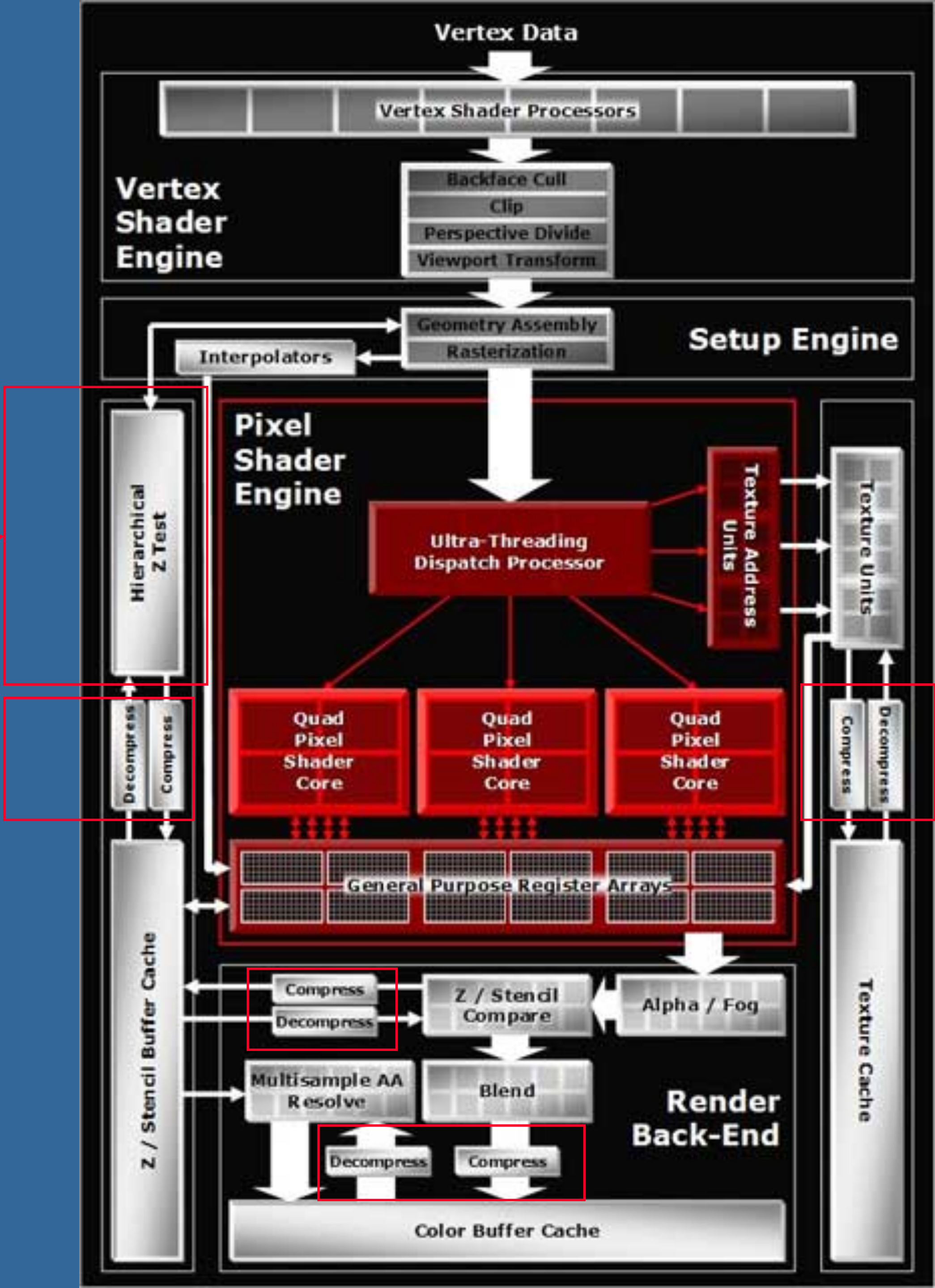
X1800 GTO

- Real example

Z-cull

Z-compress

Also note texture compress and color compress



Taxonomy of hardware design

for how to resynchronize (sort) parallelized work.

Outputs to frame buffers must respect incoming triangle order.

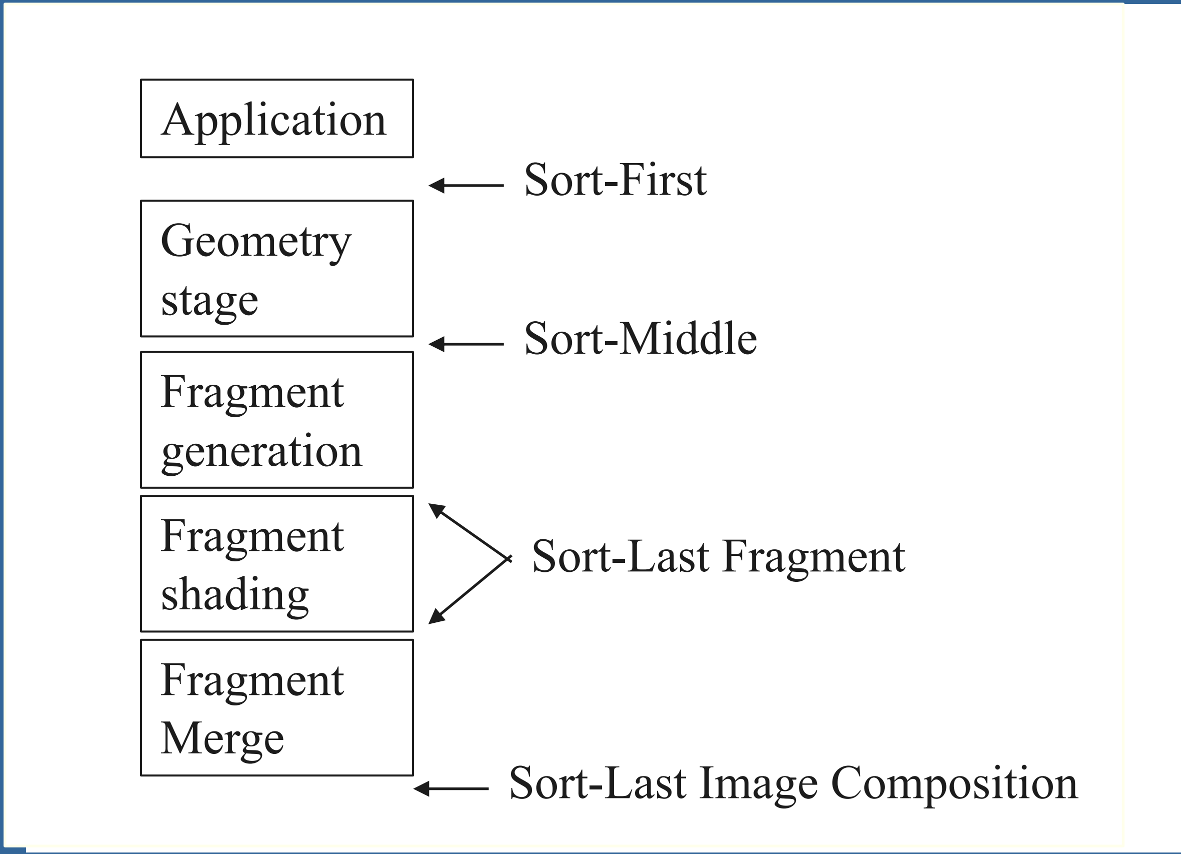
Take-aways: Sort-first, Sort-middle, Sort-Last Fragment,
Sort-Last Image

Taxonomy of Hardware

- We can do many computations in parallel:
  - Pixel shading, vertex shading, geometry shading
    - 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 where 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)

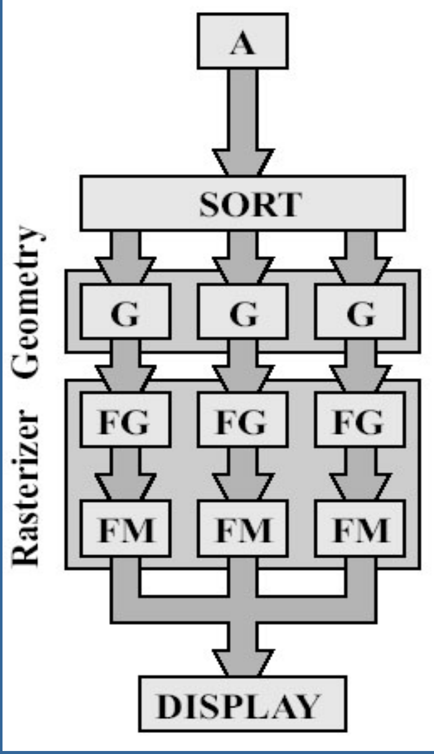
Taxonomy of hardware

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



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

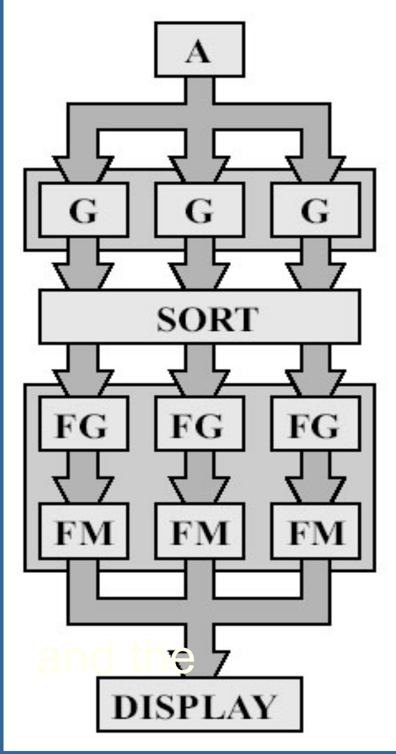
Sort-First



- Sorts primitives before geometry stage
  - Screen in 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 (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))
- Not explored much at all, since:
  - Poor load balancing if uneven triangle distribution between regions.
  - Vertex shader can change triangle position

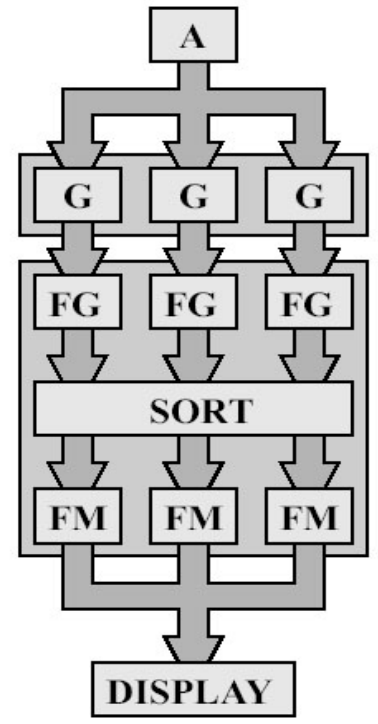
Sort-Middle

- Sorts betwen 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:
  - 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



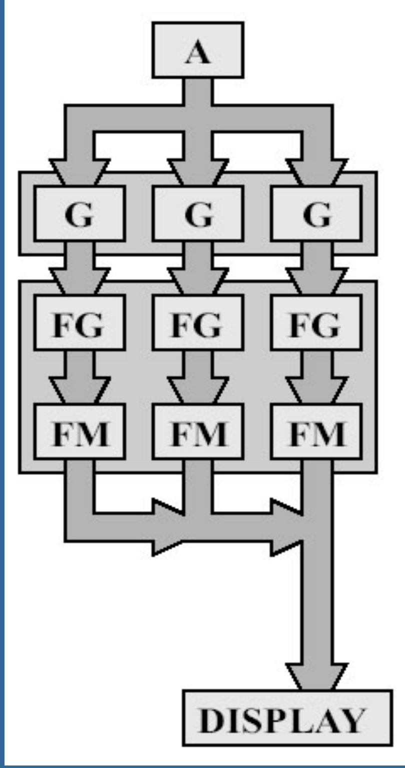
Sort-Last Fragment

- Sorts betwen 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 fragment is only sent to one FG, so this avoids doing the same work twice
- (Bad: many more fragments to sort than triangles)

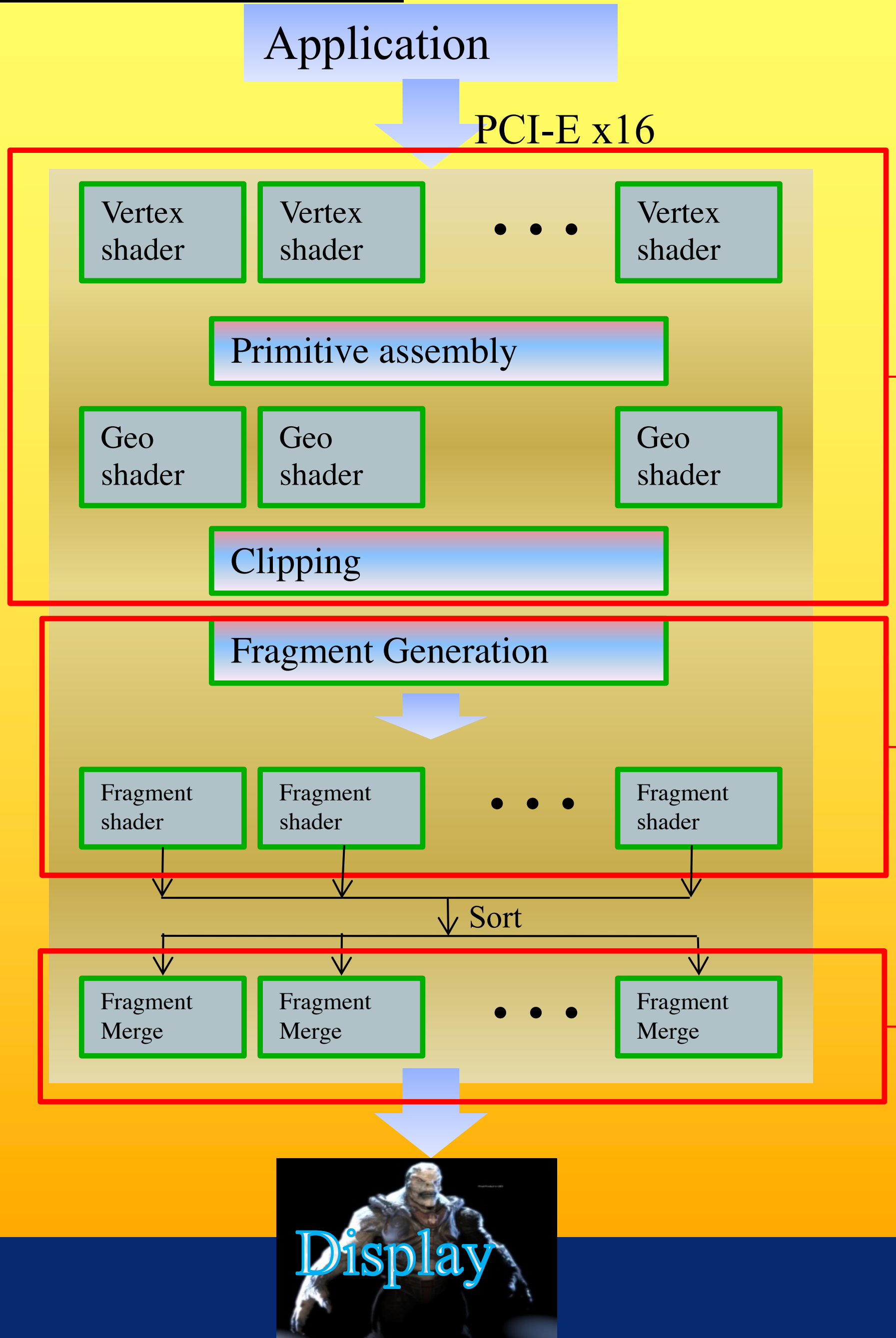


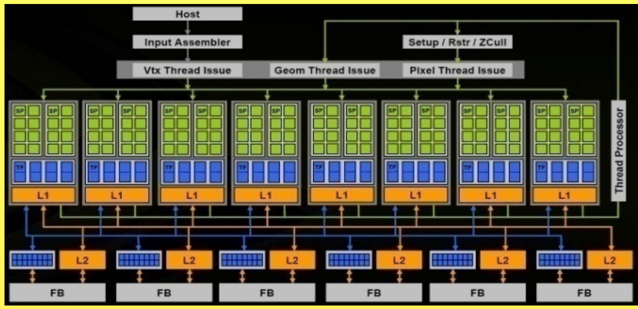
Sort-Last Image

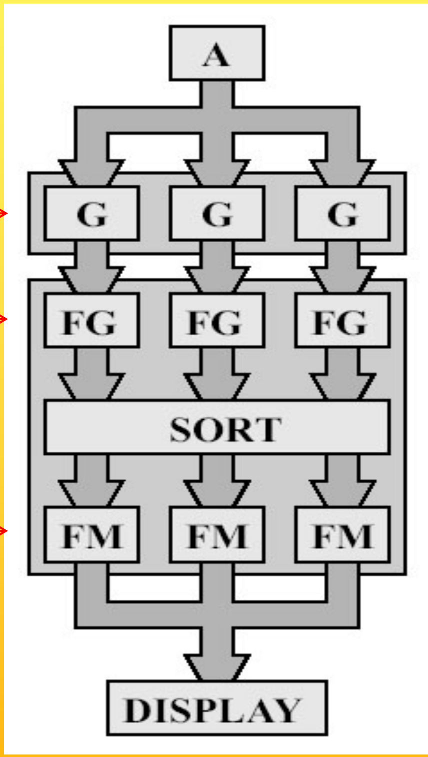
- Sorts after entire pipeline
- So each FG & FM has a separate frame buffer for entire screen (Z and color)
  - Typically: one whole graphics card per pipeline.



- 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 commercically.
- Problematic for transparency.


Application
PCI-E x16
Vertex shader
Vertex shader
• • •
Vertex shader
Primitive assembly
Geo shader
Geo shader
Geo shader
Clipping
Fragment Generation
Fragment shader
Fragment shader
• • •
Fragment shader
Sort
Fragment Merge
Fragment Merge
• • •
Fragment Merge
Display


Host
Input Assembler
Setup / Rstr / ZCull
Vtx Thread Issue
Geom Thread Issue
Pixel Thread Issue
Thread Processor
SP
TF
L1
L2
FB


A
G
G
G
FG
FG
FG
SORT
FM
FM
FM
DISPLAY

Near-future GPUs

The history implies the future

- Cell – 2005, Sony Playstation 3
  - 8 cores à 4-float SIMD
  - 256KB L2 cache/core
  - 128 entry register file
  - 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 (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
- NVIDIA Volta 2018 – 84 cores à 64 float SIMD + 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

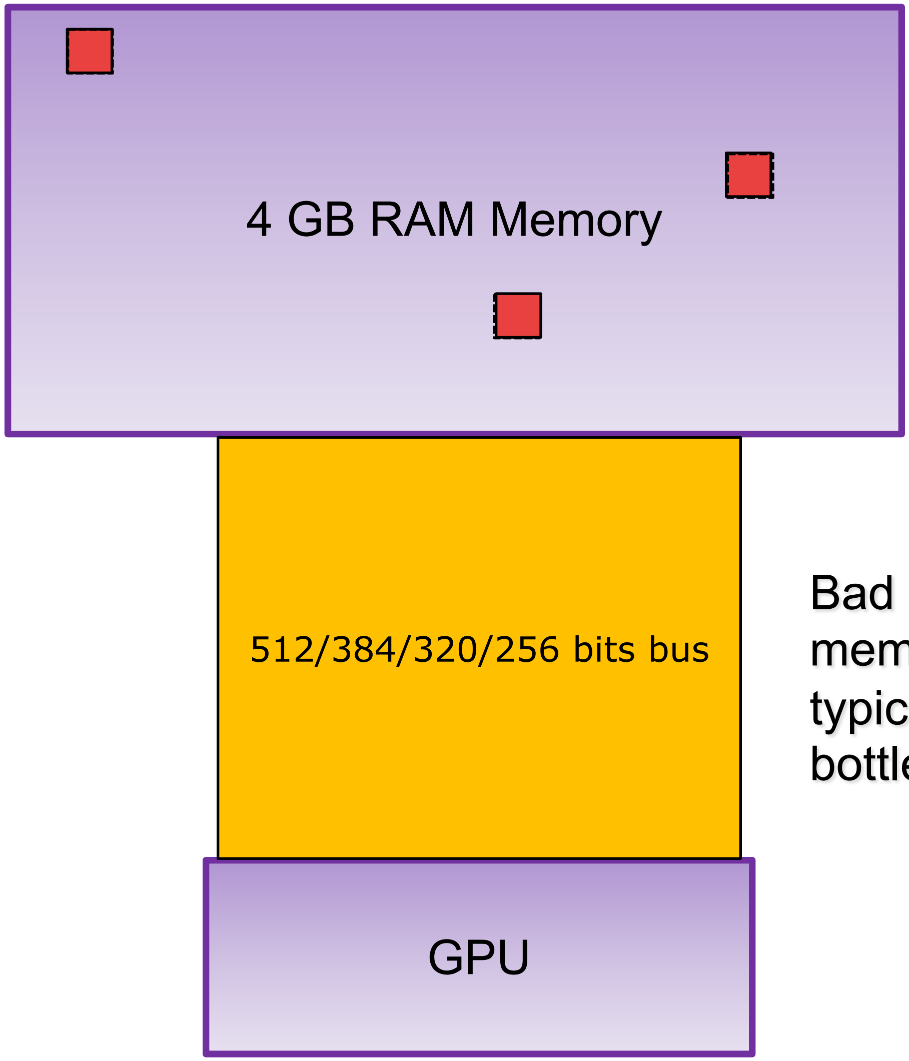
If we have time…

How create efficient GPU programs?

Answer: coallesced memory accesses

Graphics Processing Unit - GPU

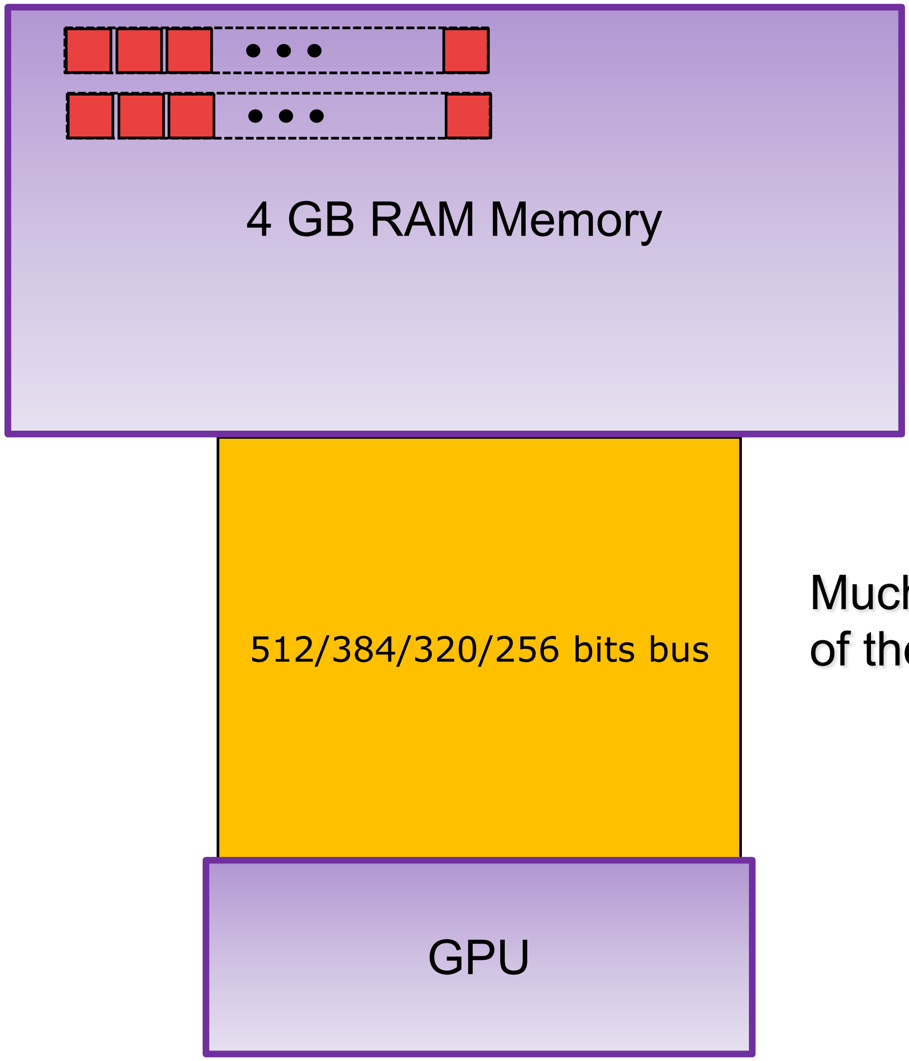
Conceptual layout:



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

🟥 = memory element (32 bits)

Graphics Processing Unit - GPU

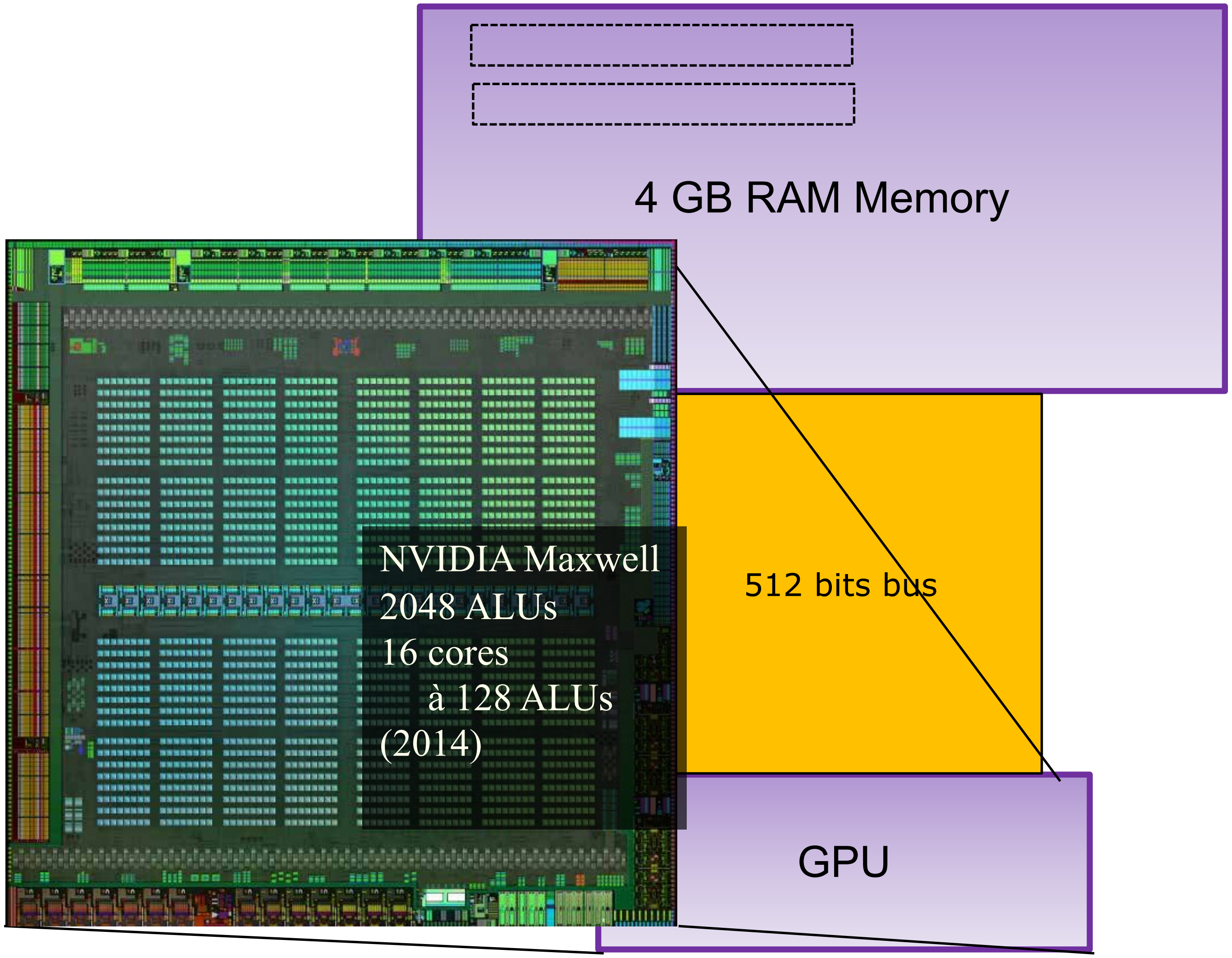


Read 32 coallesced floats for max bandwidth usage

Much better utilization of the memory bus!

🟥 = memory element (32 bits)

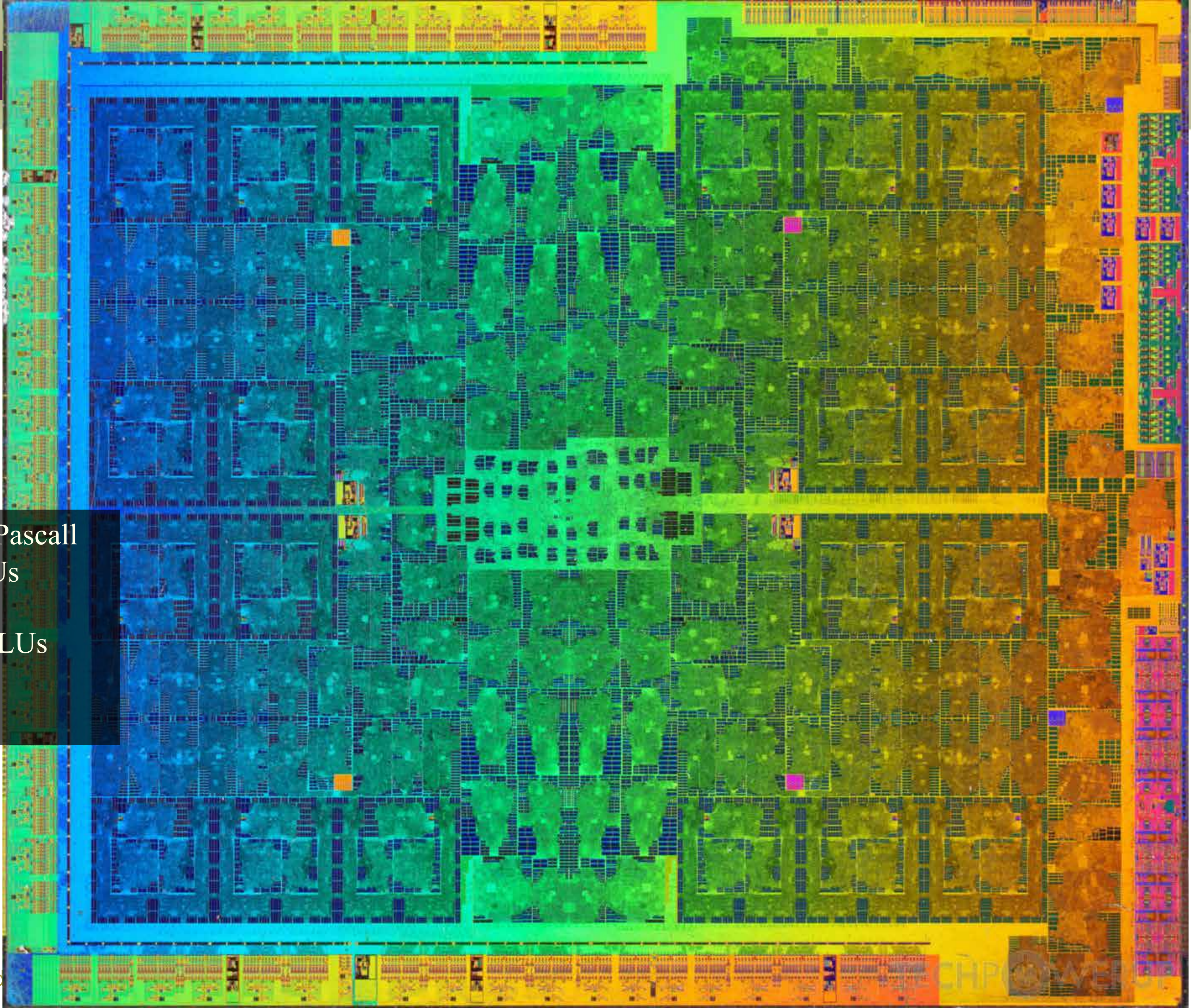
Let's look at the GPU



NVIDIA Maxwell
2048 ALUs
16 cores
à 128 ALUs
(2014)

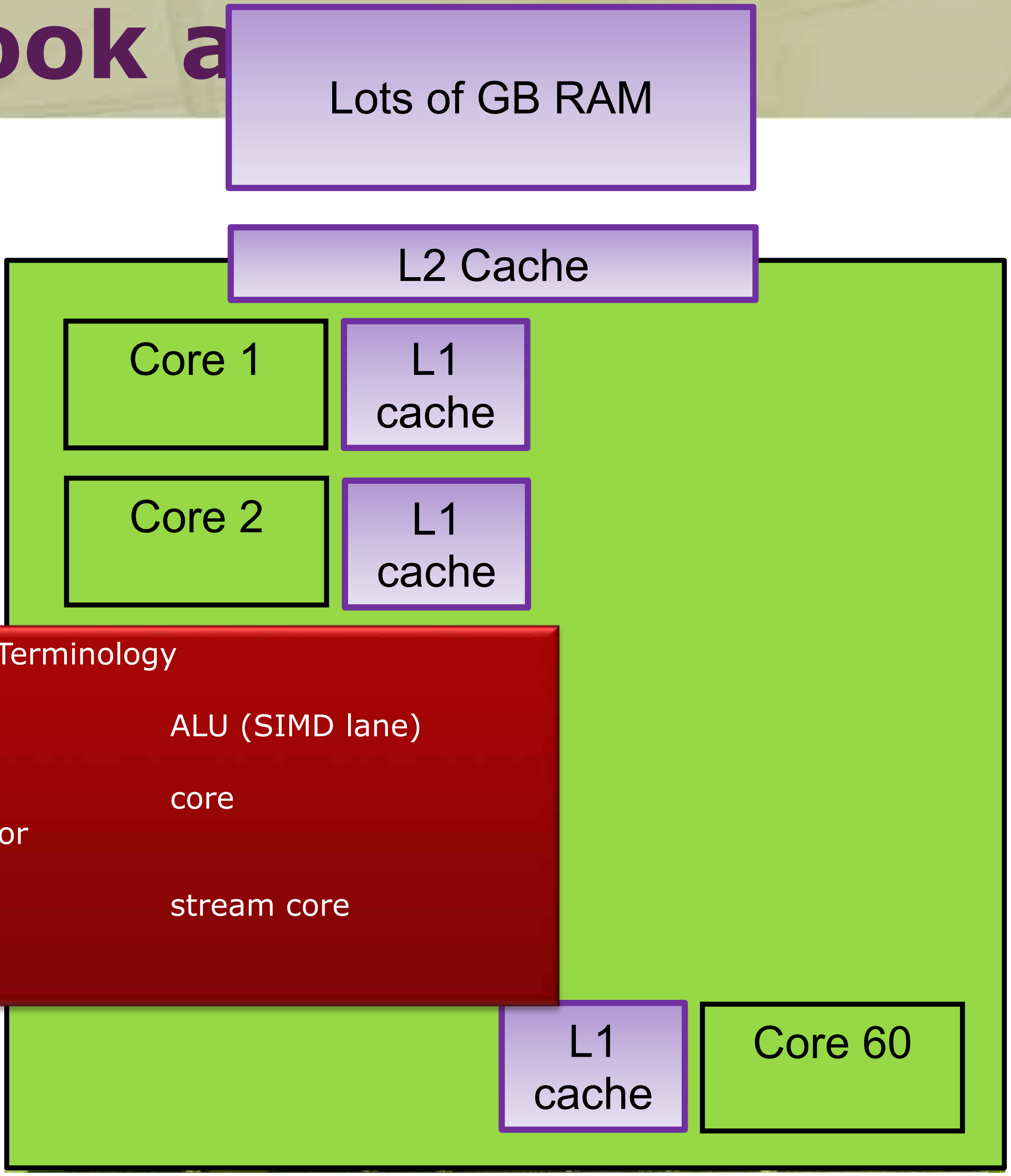
NVIDIA Fermi – GTX480, 2010. 16 cores

Let



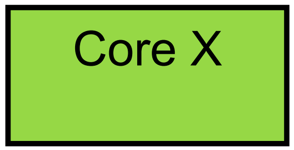
NVIDIA Pascall
3584 ALUs
60 cores
à 64 ALUs
(2016)

Let’s look a



Terminology

CPU:	Core	ALU (SIMD lane)
NVIDIA:	Streaming Multiprocessor	core
ATI	SIMD core	stream core



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

SIMD = single instruction multiple data

Pascal: 60 multi-processors

Let’s look at the GPU

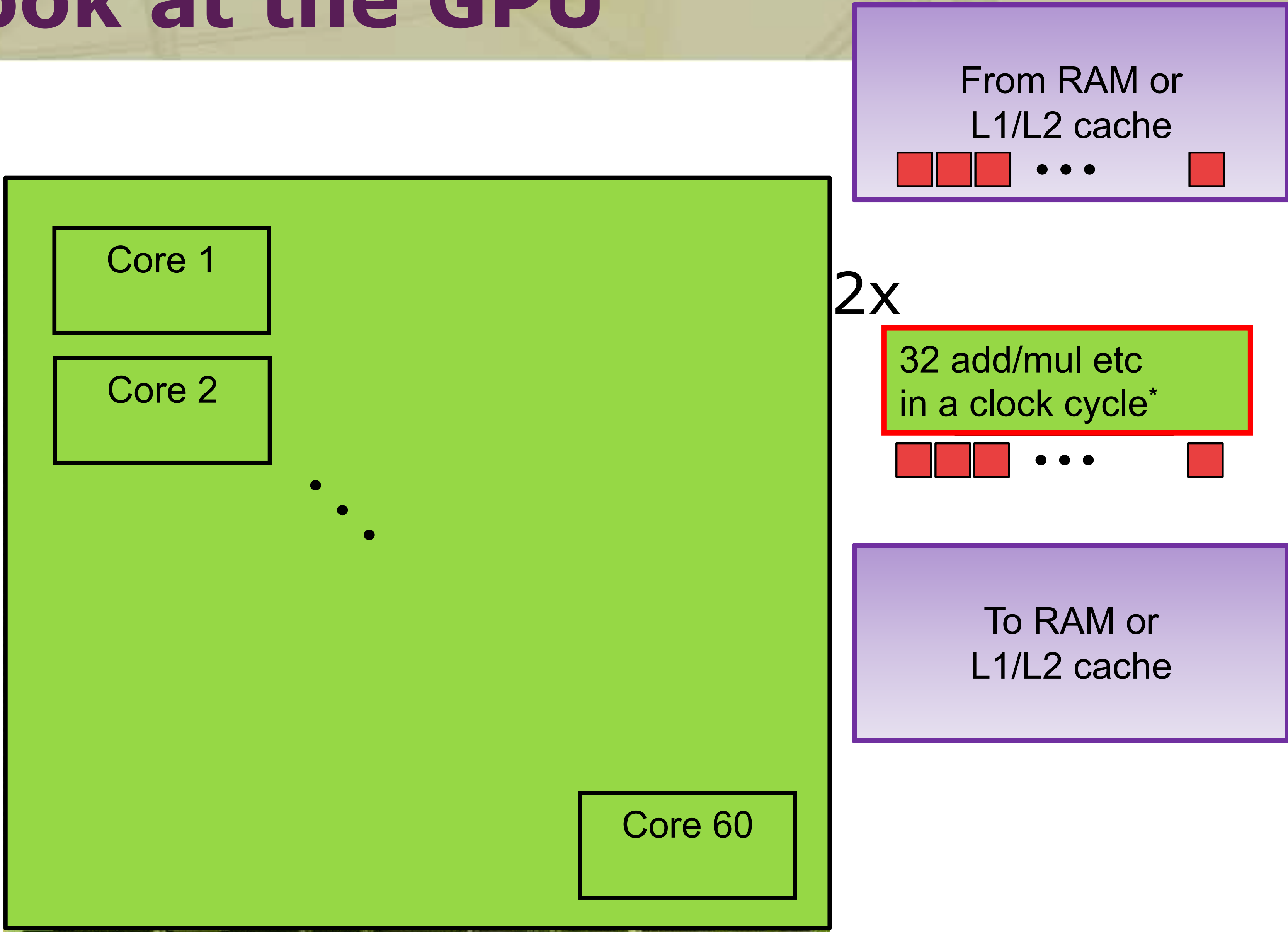
Each core:

- executes one program (=shader).

Each cycle:

- 64 flops
- 2x32 SIMD

can be up to 4 different instr.



NVIDIA Pascal: 60 multi-processors

*1-2 clock cycles depending on architecture

Let’s look at the GPU

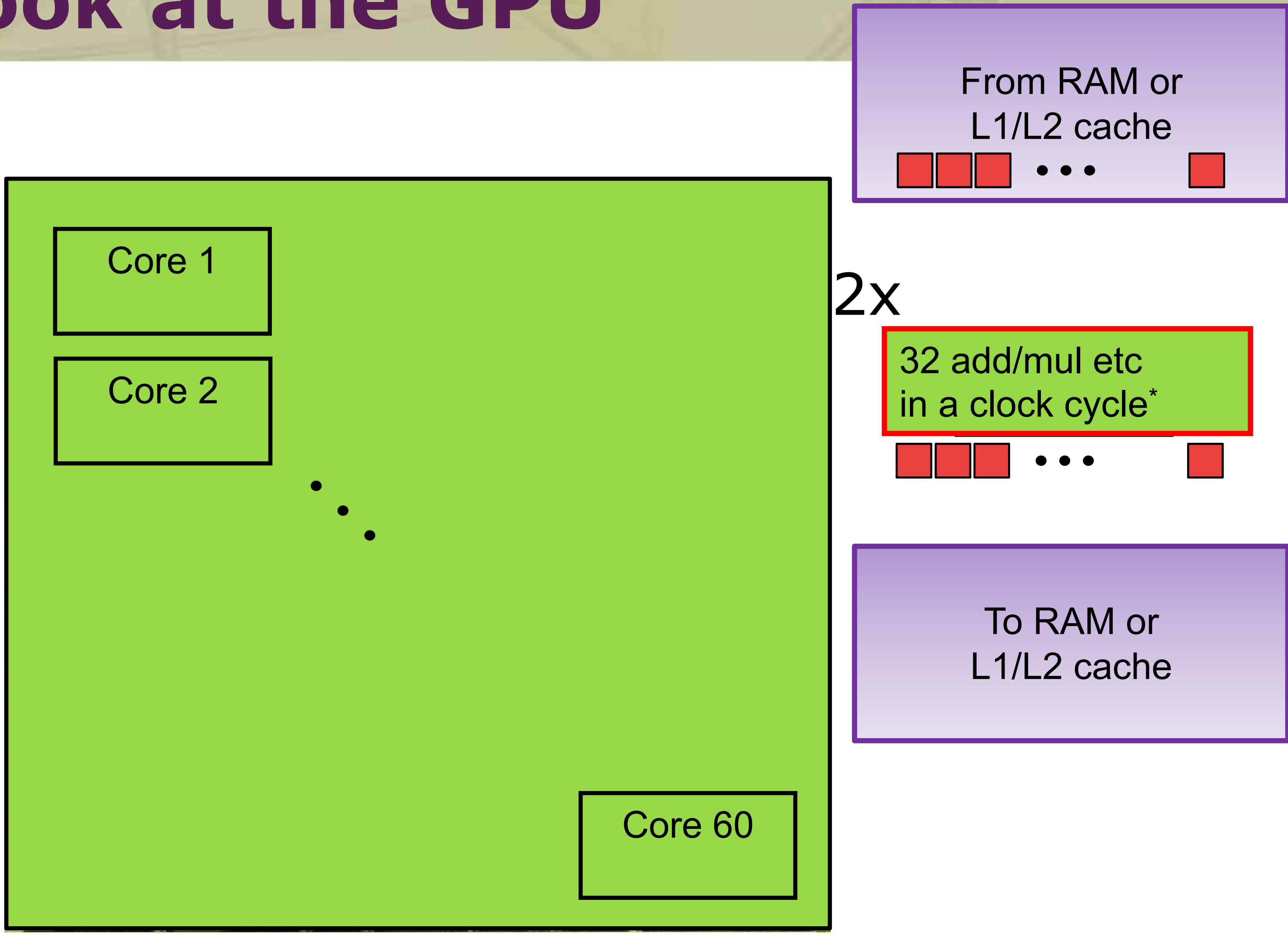
Each core:

- executes one program (=shader).

Each cycle:

- 64 flops
- 2x32 SIMD

can be up to 4 different instr.



NVIDIA Pascal: 60 multi-processors

*1-2 clock cycles depending on architecture

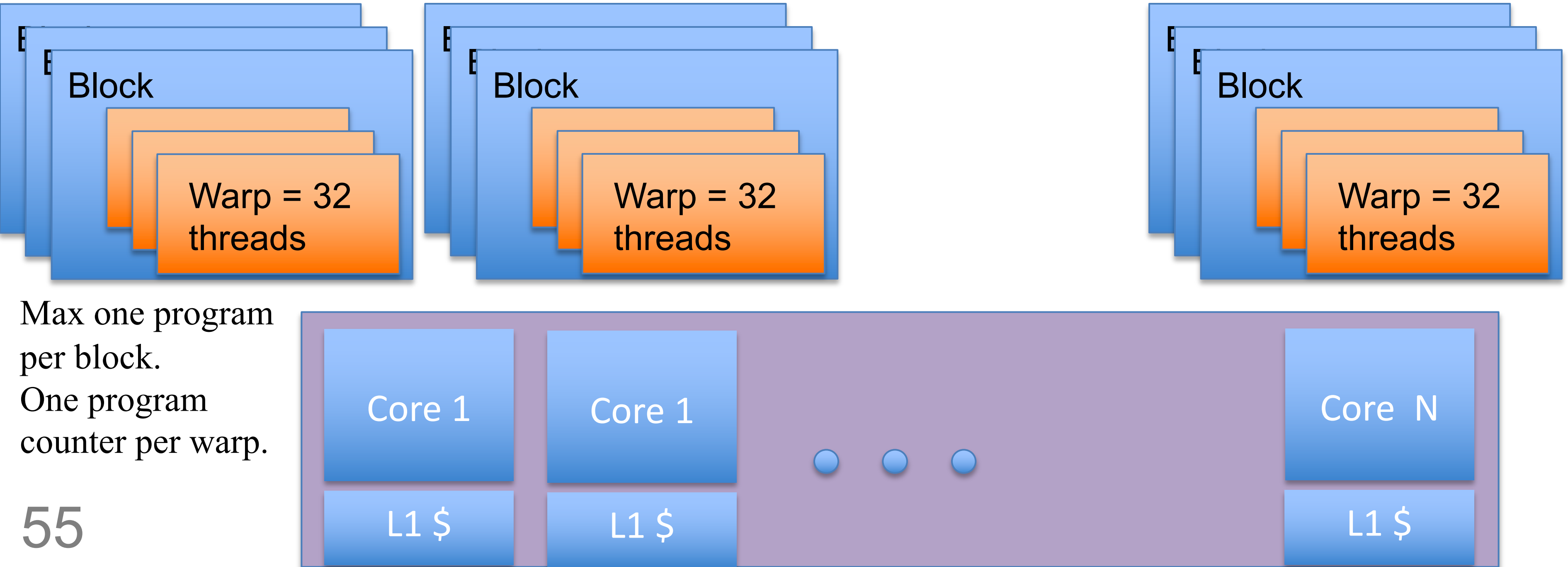
Low level APIs for GPU programming

- 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

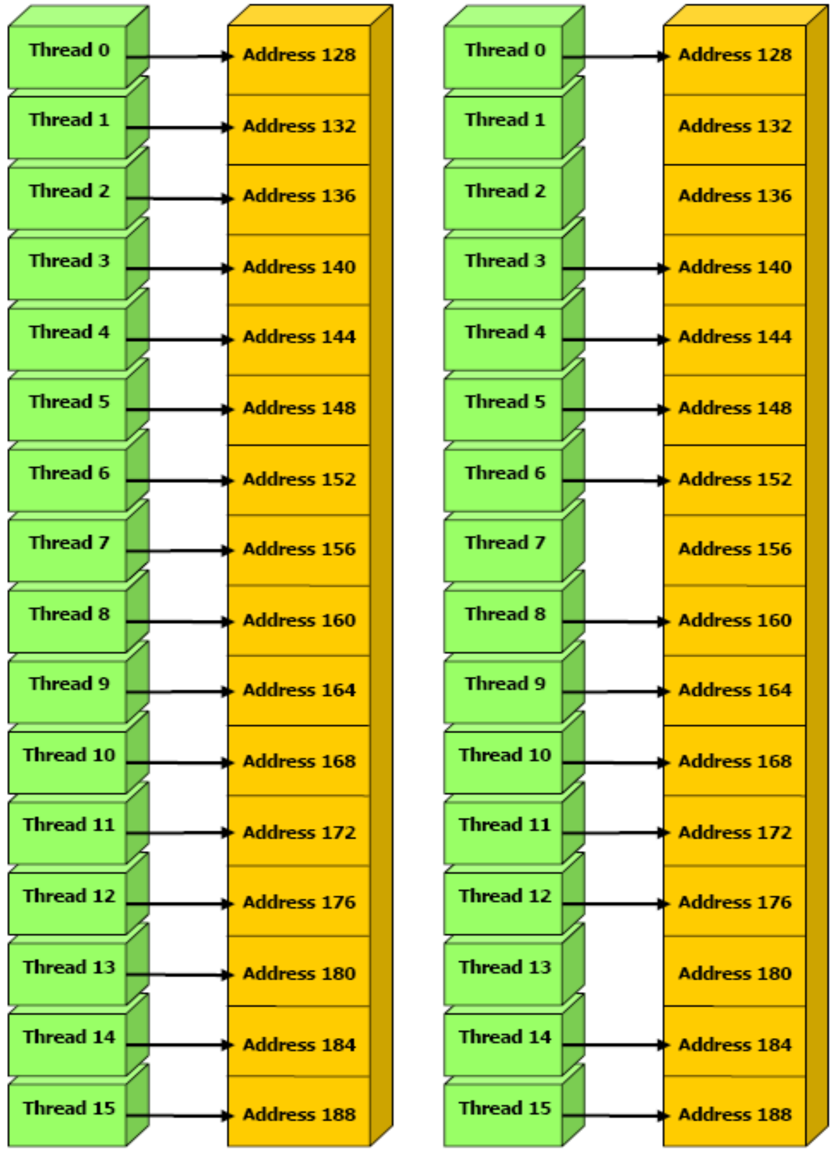
- A kernel (=CUDA program) is executed by 100:s-1M:s threads
  - A ”warp” = 32 threads, one thread per ALU
  - Warps (one to ~32) are grouped into one block
  - Block: executed on one core
    - One to 48 warps execute on a core

1 core



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

Memory Acceses – 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

Read whole cache blocks (128 bytes)

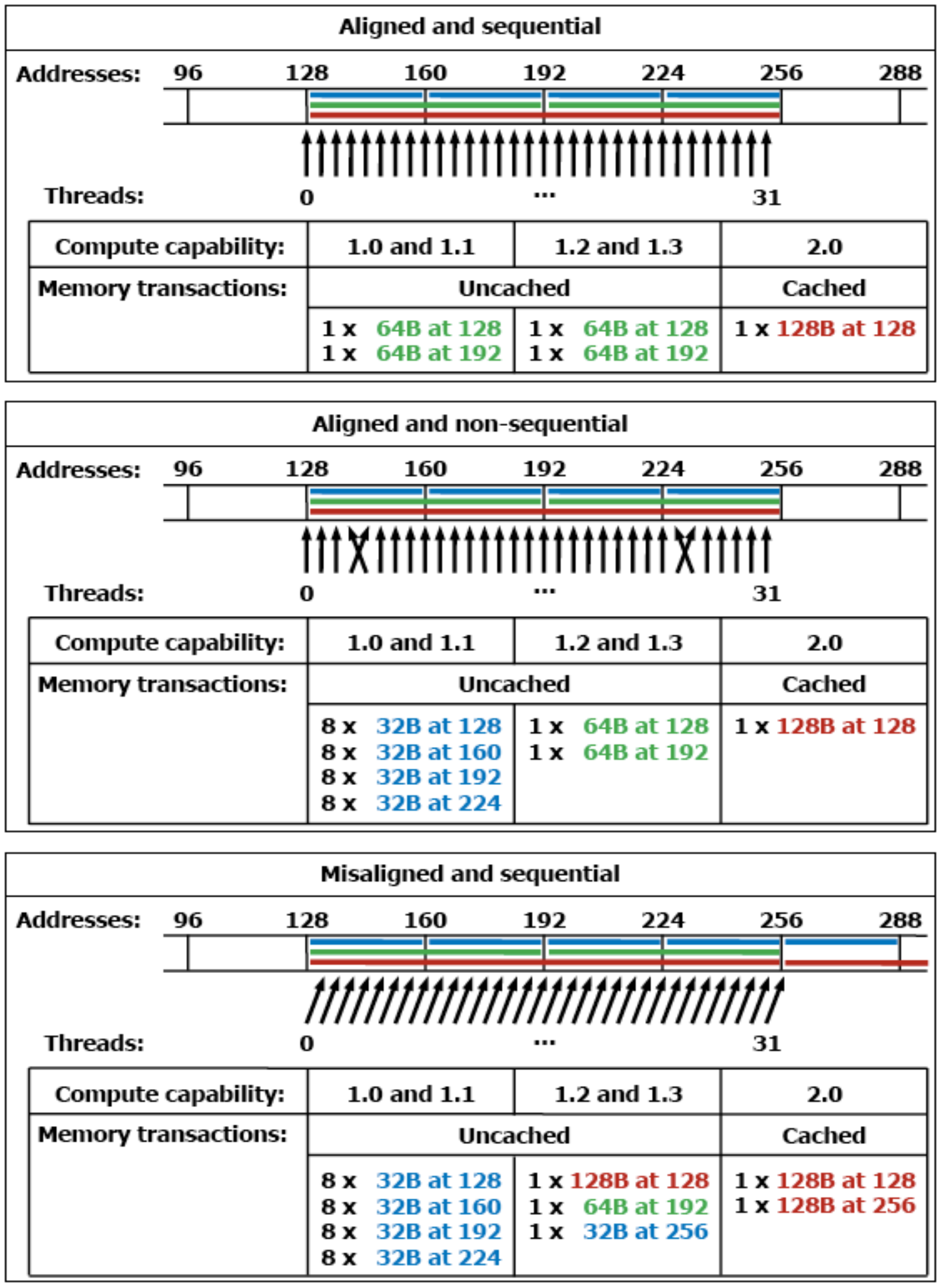
- Global mem accesses.

- One transaction:

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

- Two transactions:

Fermi:


Aligned and sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31
Aligned and non-sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31
Misaligned and sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 ... 31

Aligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Aligned and non-sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

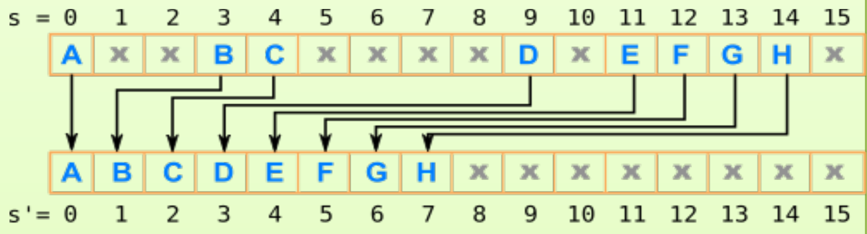
Misaligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

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


s = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A x x B C x x x x D x E F G H x
A B C D E F G H x x x x x x x x
s'= 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

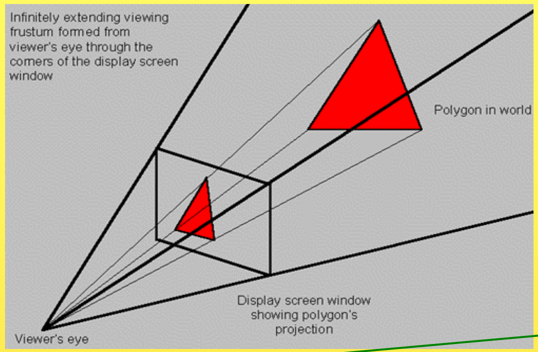
input	1	3	9	4	2	5	7	1	8	4	5	9	3
output	0	1	4	13	15	…	…	…	…	…	…	…	…

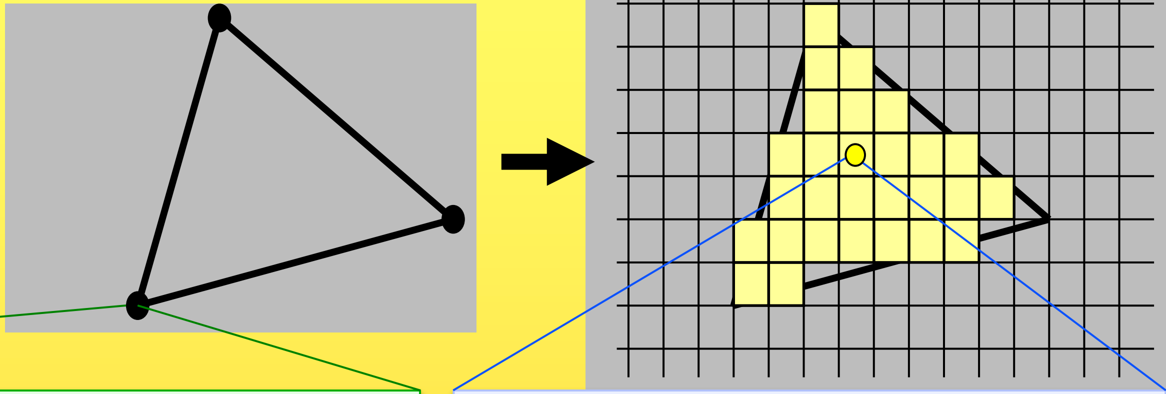
19 5 100 1 63 79

1 5 19 63 79 100

Fermi: 16 multi-processors à 2x16 SIMD width

Shaders

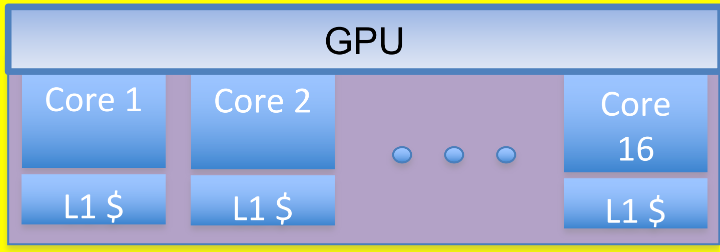

Infinitely extending viewing frustum formed from viewer's eye through the corners of the display screen window
Polygon in world
Display screen window showing polygon's projection
Viewer's eye



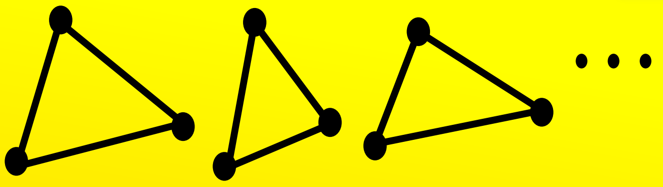
```
// Vertex Shader
#version 130

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

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

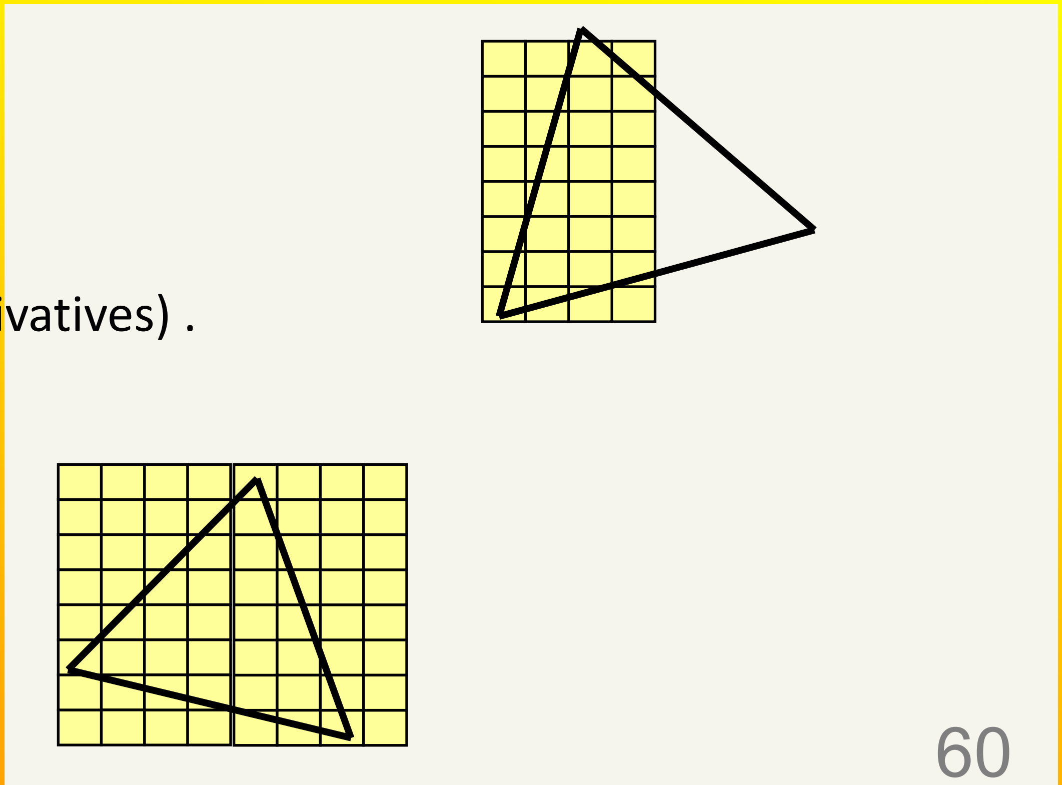
```
// Fragment Shader:
#version 130
in  vec3 outColor;
out vec4 fragColor;

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

Shaders and coallesced 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:

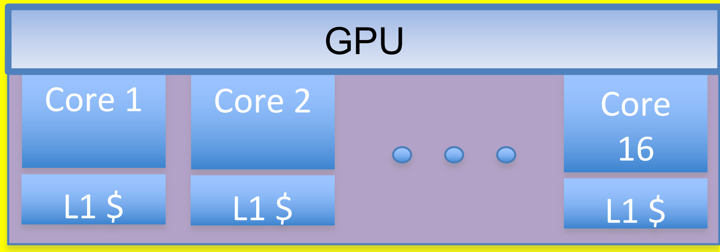


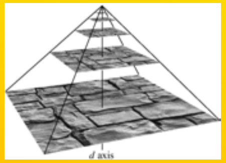


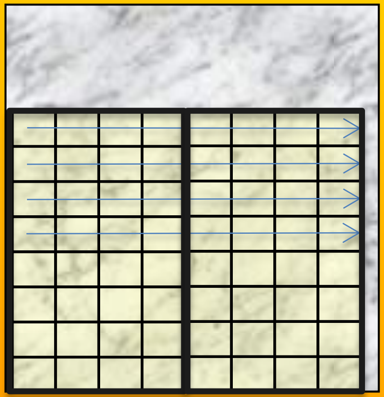
– However, many architectures can execute different instructions, of the same shader, for different warps (warp = group of 32 ALUs)

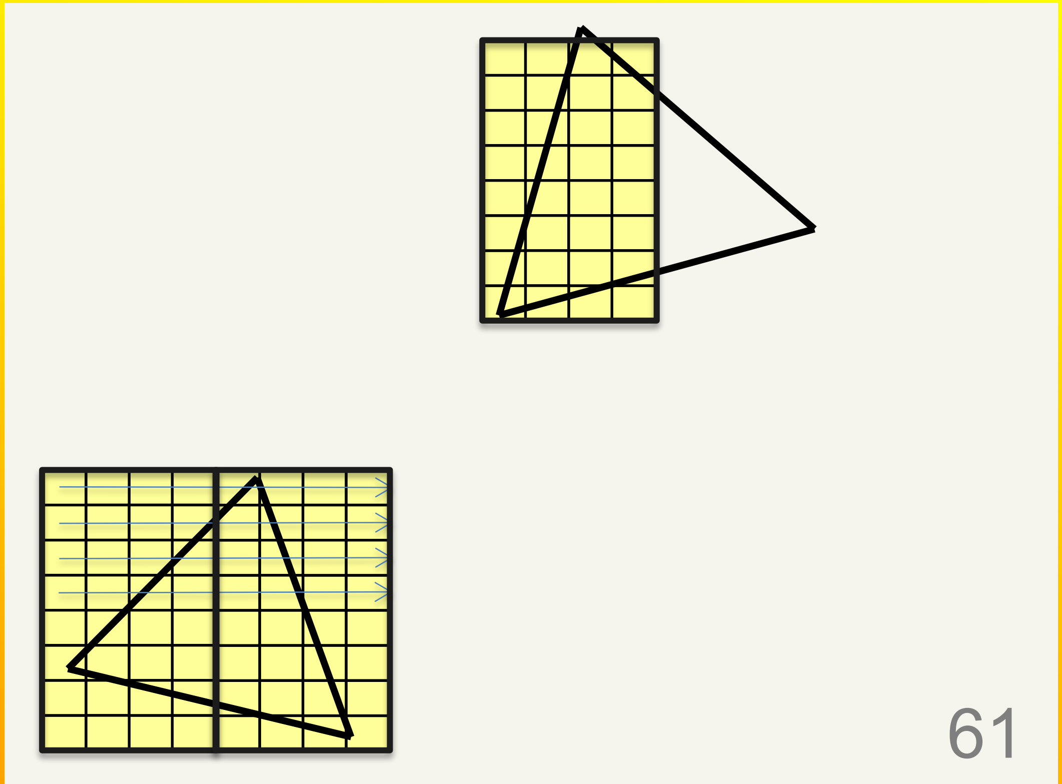
Shaders and coallesced memory accesses

- For mipmap-filtered texture lookups in a fragment shader, this can provide coallesced 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 (…)
  - Then, a = b + c;
  - …
- Else
  - a = c + d;

…the core must execute both paths if any of the 32 threads need the if and else-path.

But not if all need the same path.

Summary

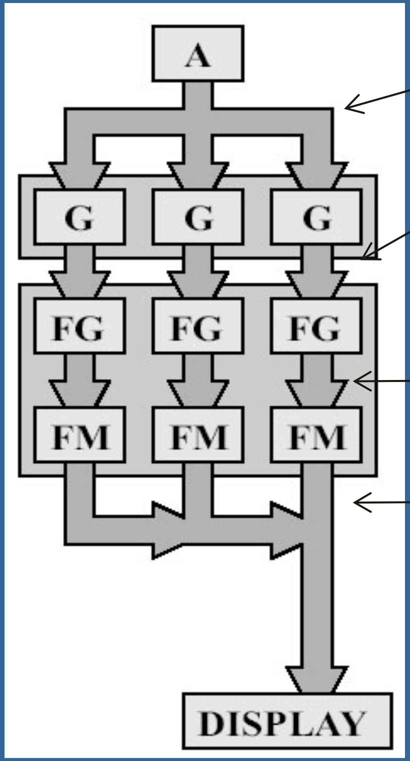
Linearly interpolate $(u_i/w_i, v_i/w_i, 1/w_i)$ in screenspace from each triangle vertex i.
Then at each pixel:

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

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

Need to know:

- 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→)



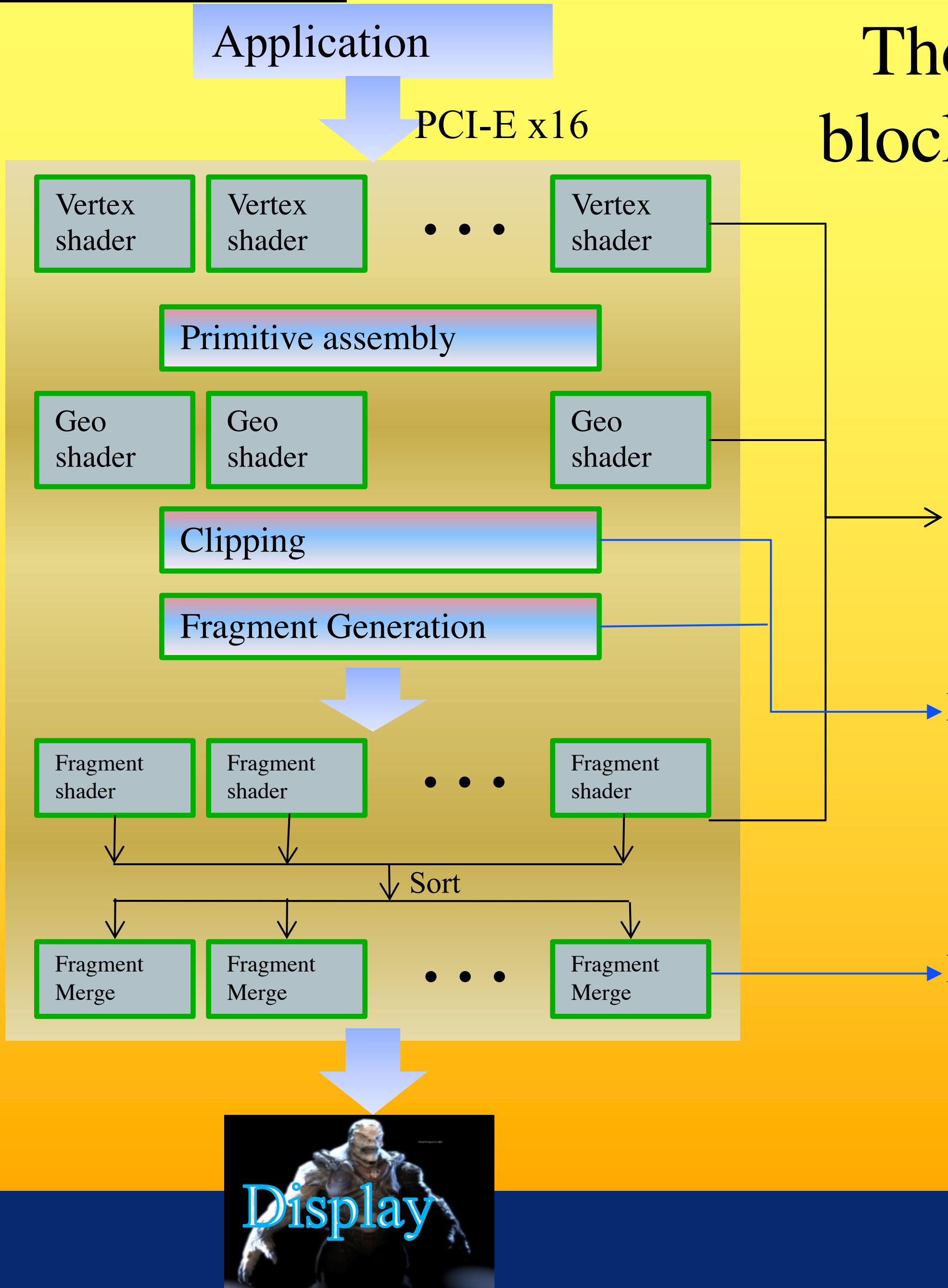
Sort-first

Sort-middle

Sort-last fragment

Sort-last image

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



Vertex-, Geometry- and Fragment shaders exectued on a pool of thousands of ALUs

Fixed function hardware

Fixed function hardware