Graphics Hardware

Ulf Assarsson

CHALMERS

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 OptiX
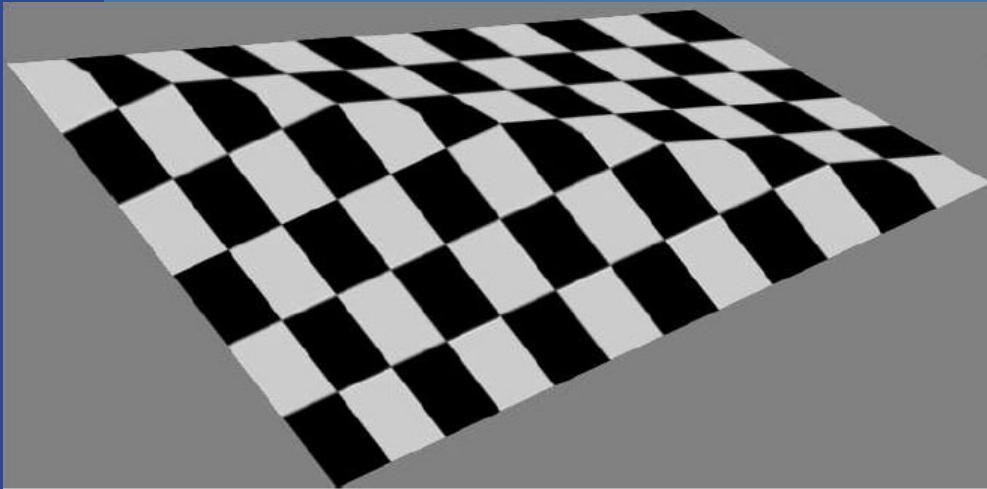 - Or write your own GPU ray-tracer

POSSESSION

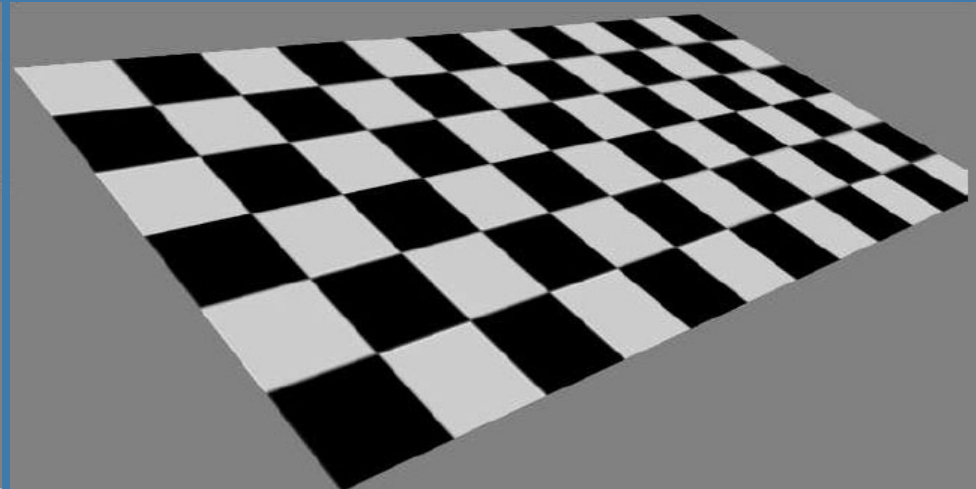Possession is a registered trademark of Blitz Games Limited © Blitz Games 2005

PS3 PORTAL

Perspective-correct texturing

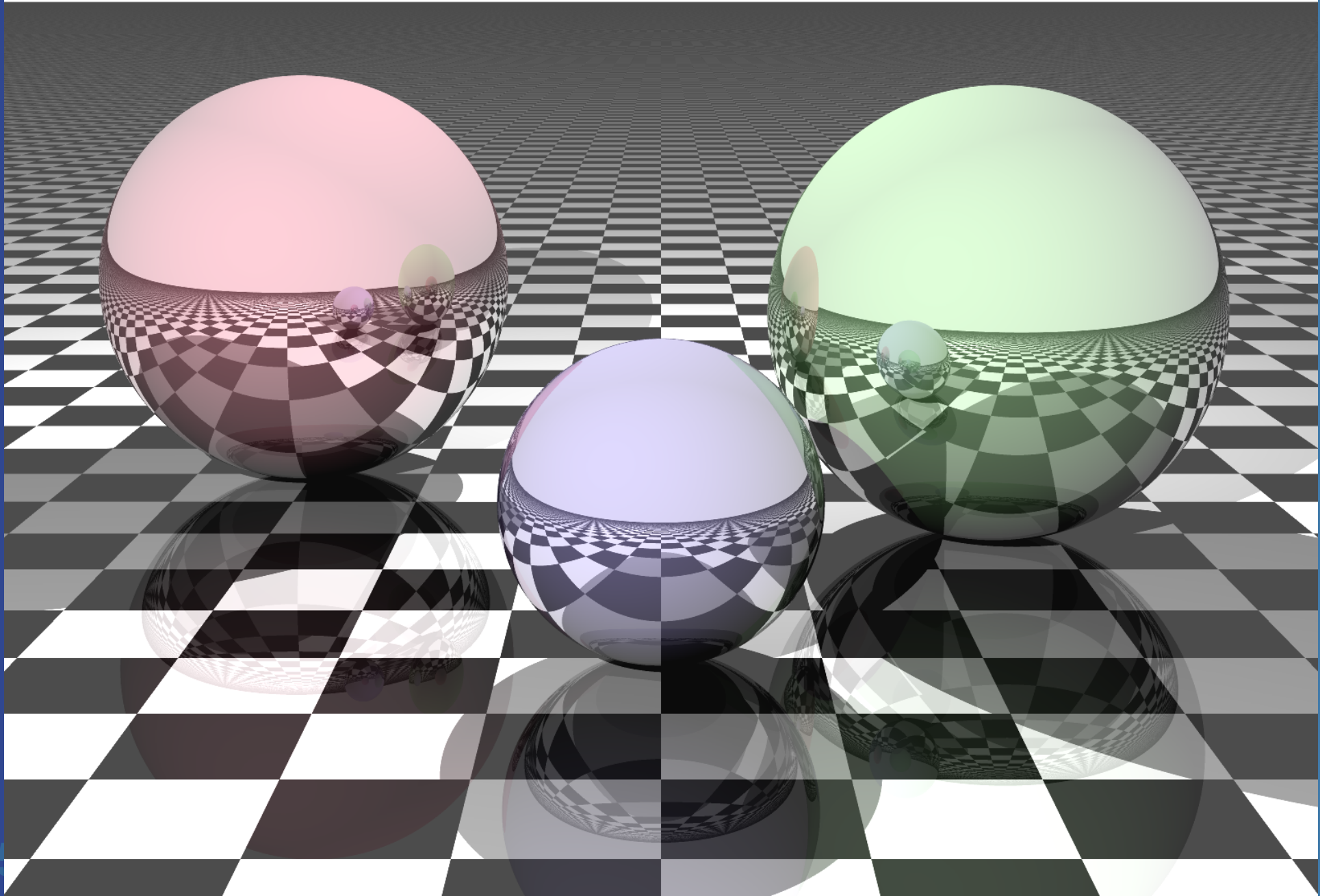- How is texture coordinates interpolated over a triangle?
- Linearly?

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

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

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

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: $(w*u/w,w*v/w)=(u,v)$
    - 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:
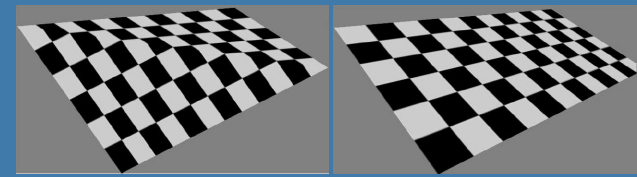
- Linear interpolation in screen space does not work for u,v

- Solution:
  - We have applied a non-linear transform to each vertex (x/w, y/w, z/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:
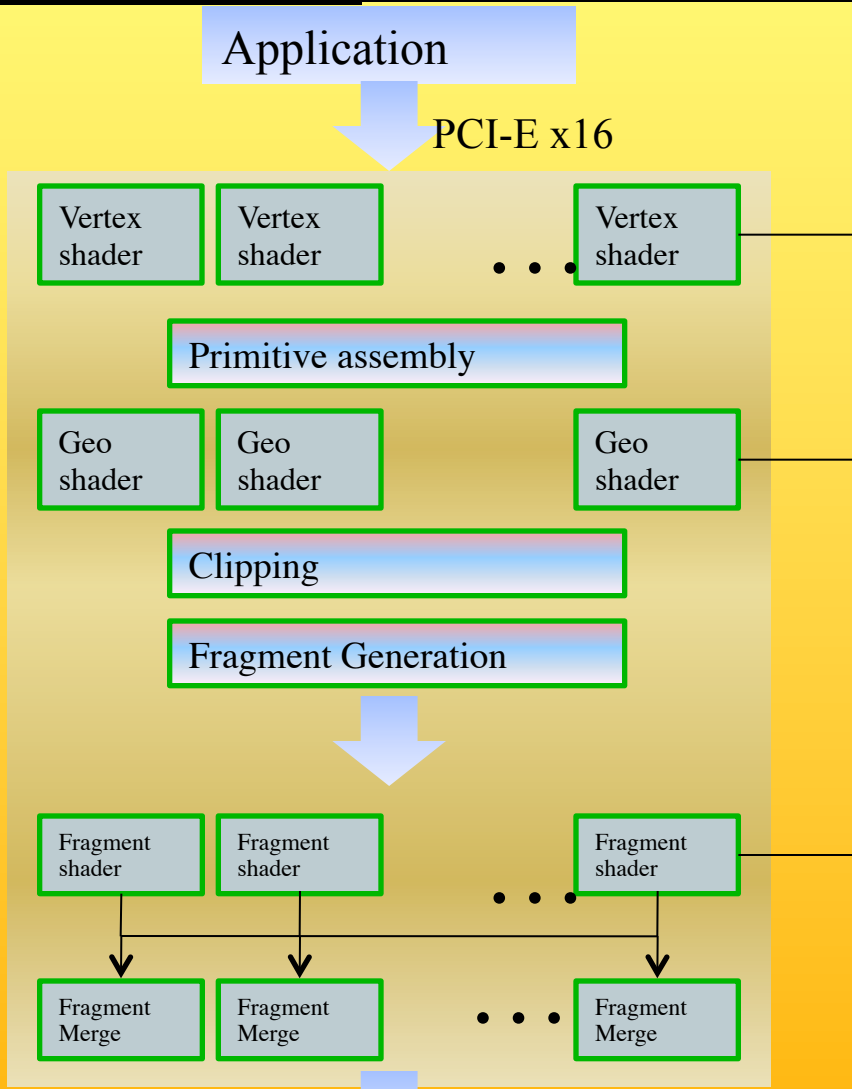   - u = (u/w) / (1/w)
      - v = (v/w) / (1/w)

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 hardware (?)
- Two major ways of getting better performance:
  - Pipelining
  - Parallellization
 - Combinations of these are often used

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 |

| Fragment Merge | Fragment Merge | ... | Fragment Merge |

Display

On NVIDIA
8000/9000/200/400/500/
600-series:
Vertex-, Geometry- and
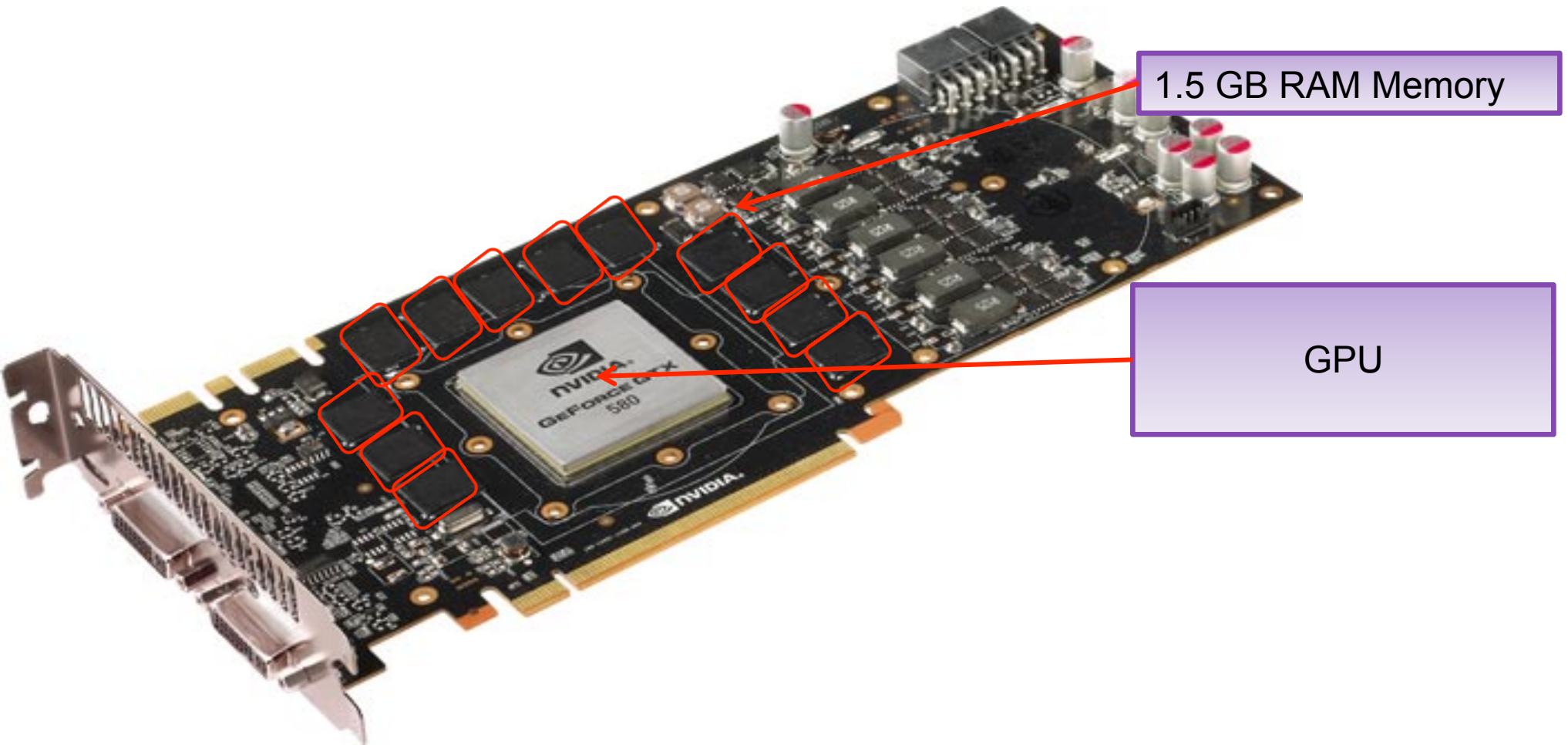Fragment shaders allocated
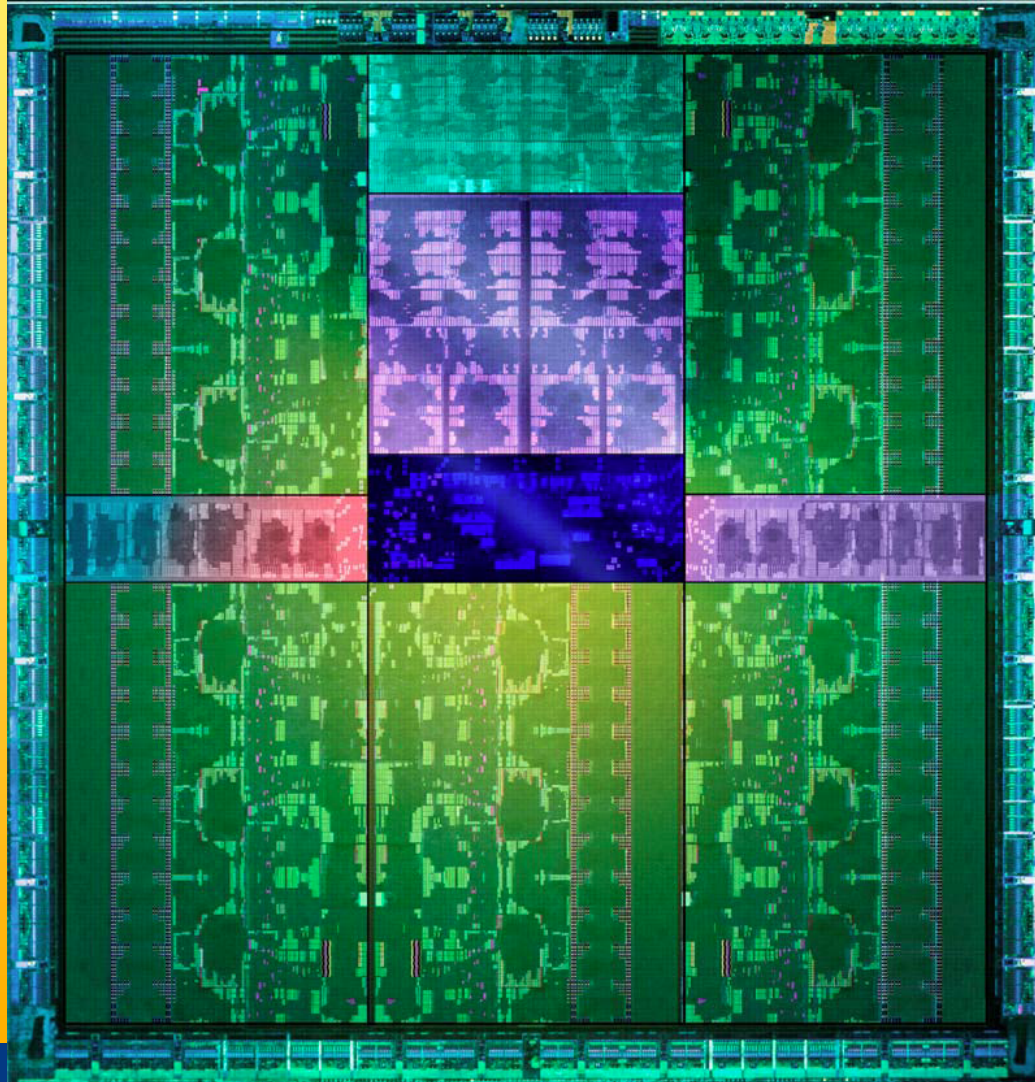from a pool of
128/240/480/512/1536/2880
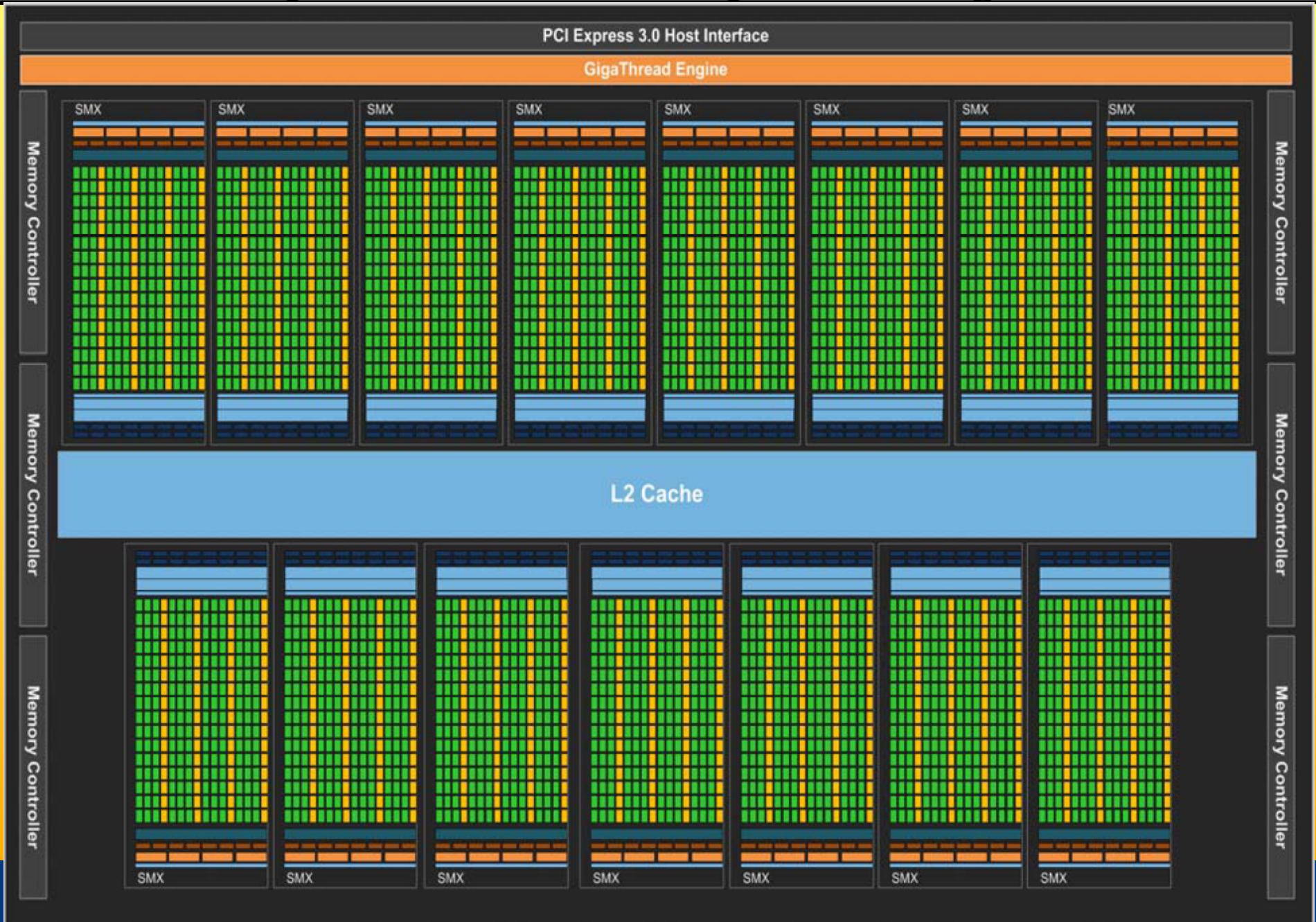ALUs

GeForce 7800
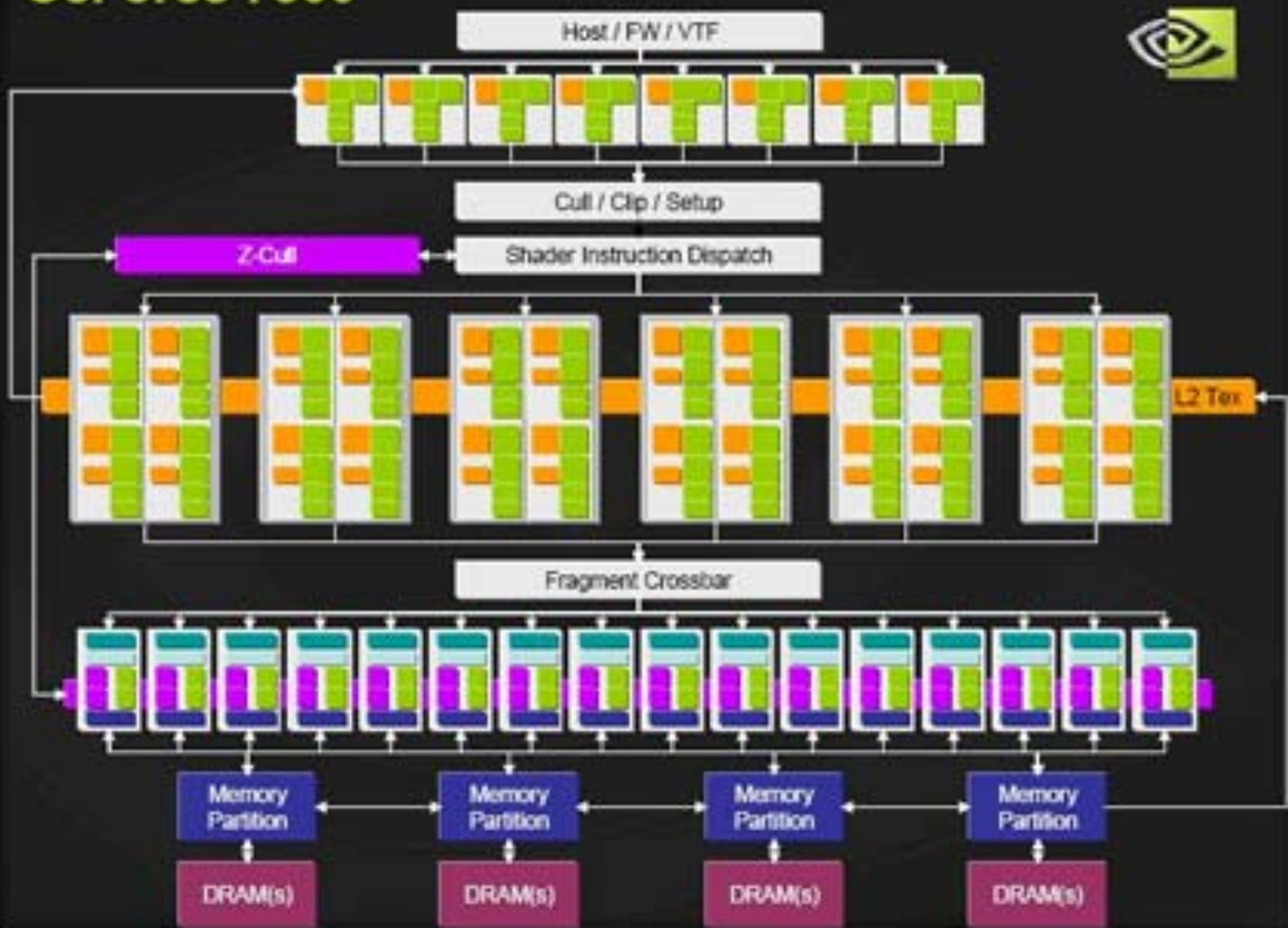
Graphics Processing Unit – GPU

1.5 GB RAM Memory

GPU

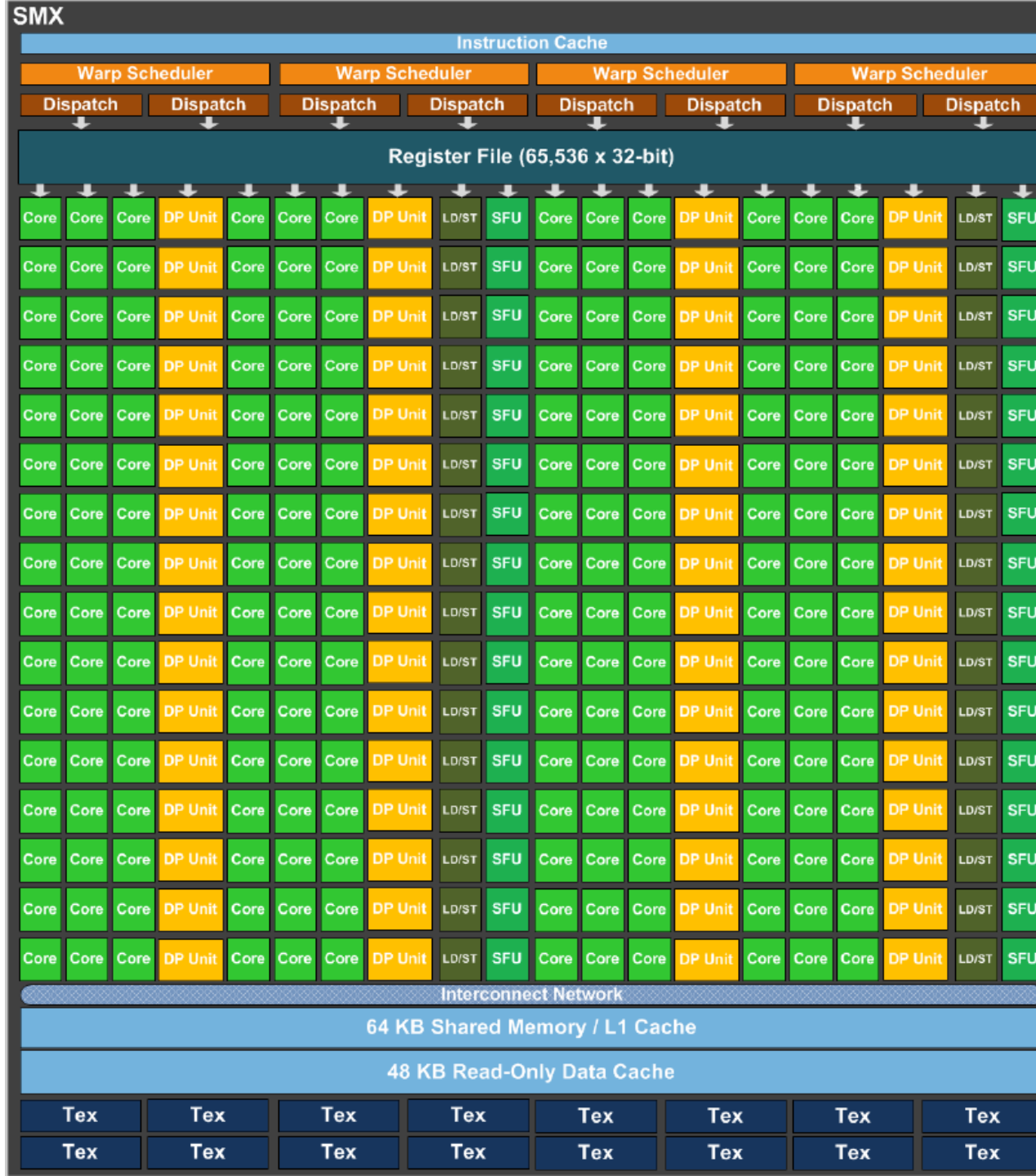- NVIDIA Geforce GTX 580

Kepler GK110 Die Photo

Kepler GK110 Full chip block diagram

SMX:

- **192 single-precision CUDA cores,**
- **64 double-precision units,**
- **32 special function units (SFU), and**
- **32 load/store units (LD/ST).**

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
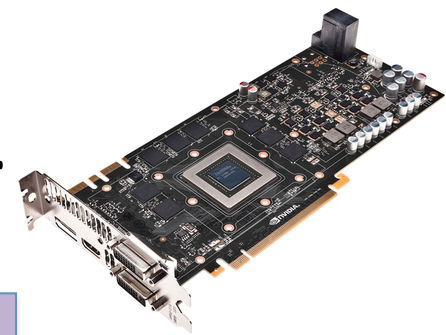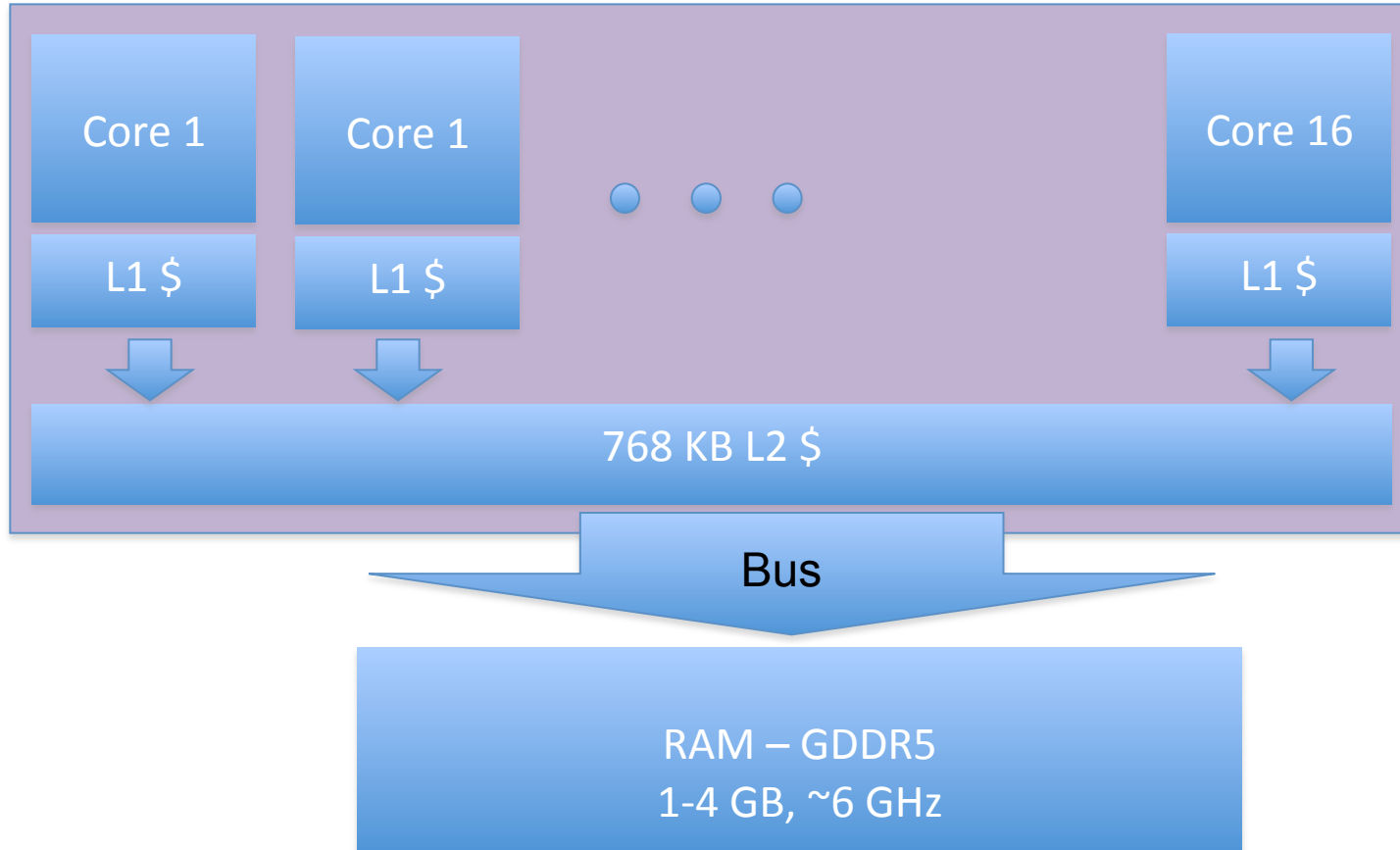   - Opened door to use of computer graphics for CAD community

Tektronix **4014**

Fig. 1-1. 4014 Computer Display Terminal.
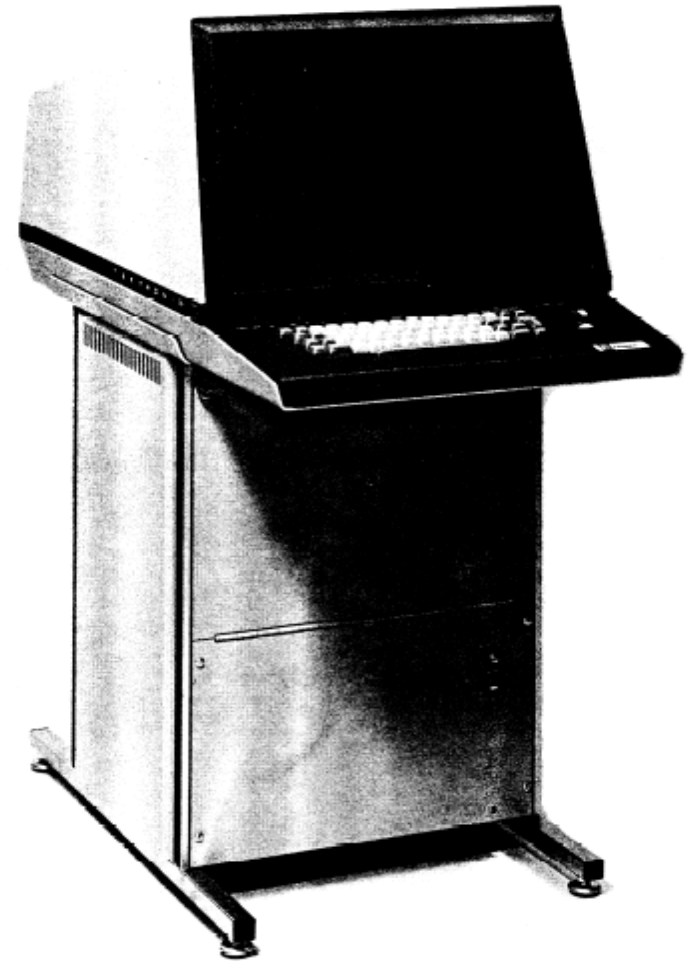
Briefly about Graphics HW pipelining

2001 ● In GeForce3: 600-800 pipeline stages!
  – 57 million transistors
  – First Pentium IV: 20 stages, 42 million transistors,
  – Core2 Duo, 271 Mtrans, Intel Core 2 Extreme QX9770 – 820Mtrans.
  – Intel Pentium D 900, 376M trans, Intel i7 (quad): 731Mtrans, 10-core Xeon Westmere: 2.6Gtrans

● 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.

  – Lesson learned: #trans doubles ~per 2 years. Core clock increases slowly. Mem clock –increases with new technology DDR2, DDR3, GDDR5
  – 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 freq^2.
    ● Memory transfers often the bottleneck

18

GPU- Nvidia's Kepler 2012

Overview:

| Core 1 | Core 1 | • • • | Core 16 |
| L1 $ | L1 $ | | L1 $ |

768 KB L2 $

Bus

RAM – GDDR5
1-4 GB, ~6 GHz

16 cores à
96-SIMD width
(2*3*16)

16/48 KB per
each 48 SIMD

Bandwidth
~150-200 GB/s

Bus: 256/384
bits
Compare to
ATI 2900:
  - 2x512bits
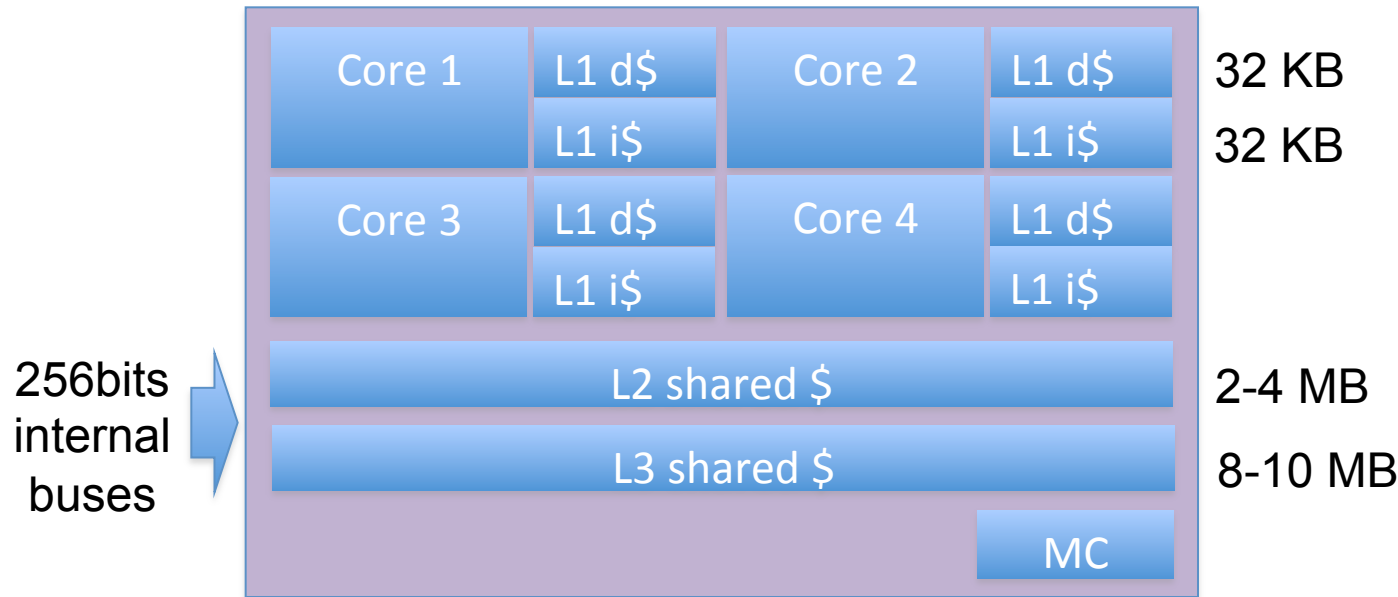Larrabee:
  - 2x512bits

Wish:
    1536 ALUs à 1 float/clock => 6KB/clock
    ~1GHz core clock => 6000 GB/s request
We have ~192 GB/s. In reality we can do 20-40 instr. between each RAM–
read/write. Solved by L1$ + latency hiding (warp switching)

CPU - 2011

Core 1	L1 d$	Core 2	L1 d$	32 KB
	L1 i$		L1 i$	32 KB
Core 3	L1 d$	Core 4	L1 d$	
	L1 i$		L1 i$	

L2 shared $ 2-4 MB

L3 shared $ 8-10 MB

MC

256bits internal buses

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

64 bits   FSB

Graphics Memory Controller HUB

Gfx card PCIe

GMCH north bridge

2x64bits

DDR3 RAM

motherboard

- 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
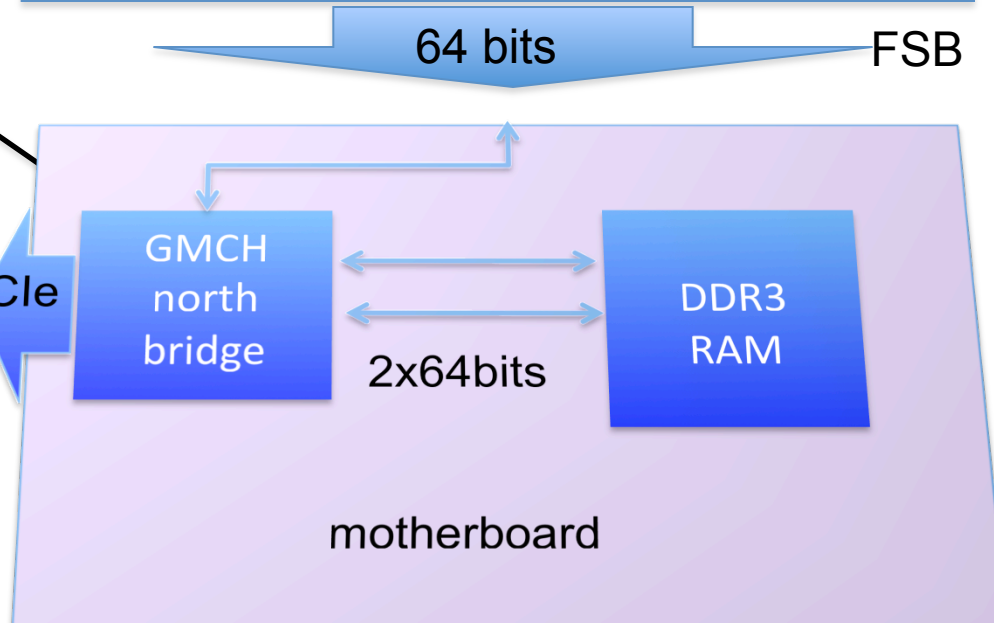
Memory bandwidth usage is huge!!

- On top of that, effective utilization of bandwith is seldom 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

Architecture of Z-cull and Z-compress
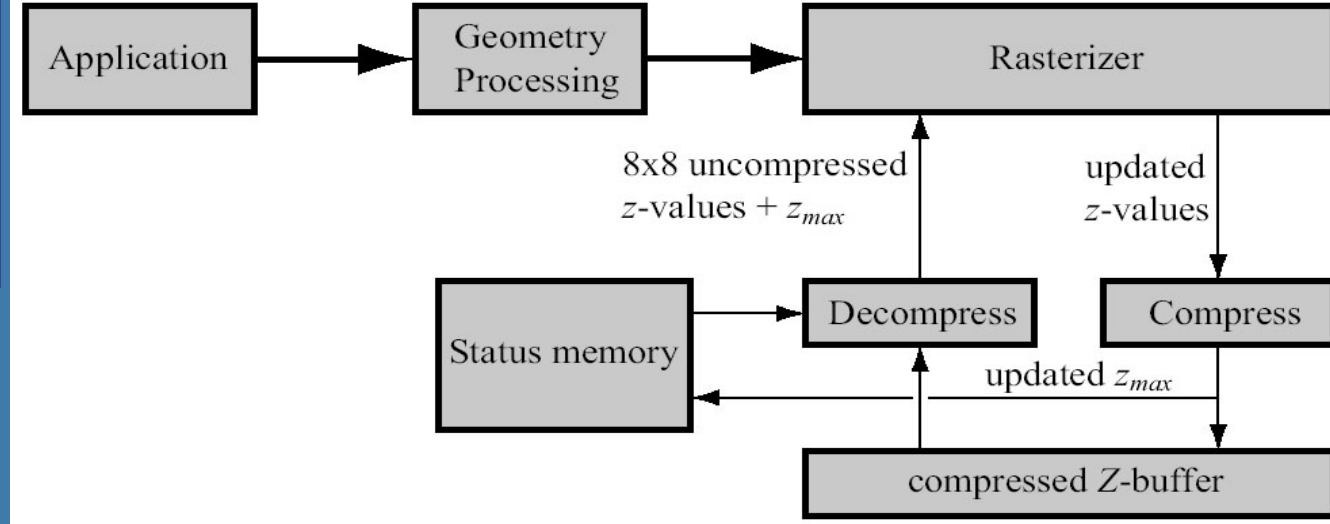
- Store zmax per tile, and a flag (whether cleared, compressed/uncompressed)
- Rasterize one tile at a time
- Test if zmin on triangle is farther away than tile's zmax
  - If so, don't do any work for that tile!!!
  - Saves texturing and z-read for entire tile – huge savings!
- Otherwize read compressed 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
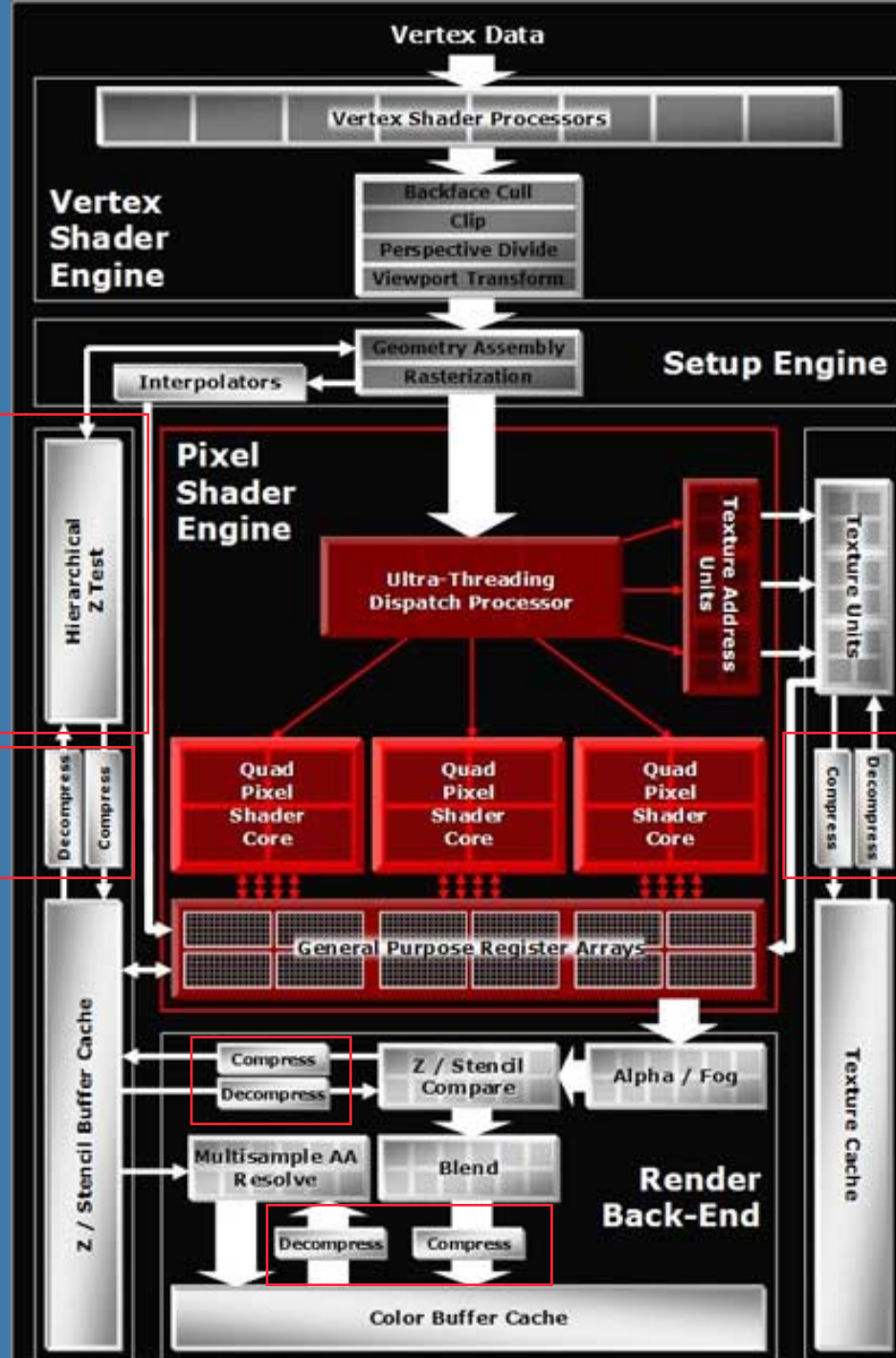 - Then we don't need to read from Z-buffer, just send cleared Z for that tile

23

X1800 GTO

- Real example

Z-cull

Z-compress

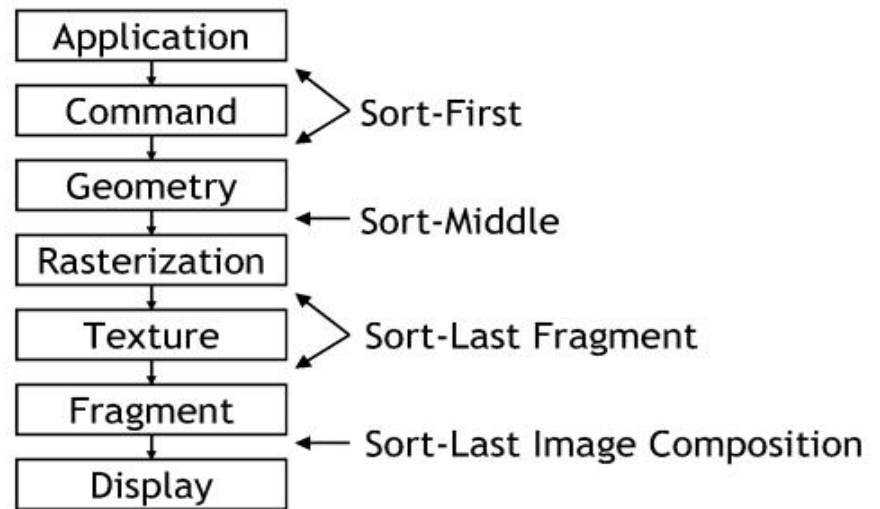Also note texture compress
and color compress

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 where 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

Application
Command → Sort-First
Geometry ← Sort-Middle
Rasterization
Texture → Sort-Last Fragment
Fragment ← Sort-Last Image Composition
Display

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 in divided into large regions
  - A separate pipeline is responsible for each region (or many)

- 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

27
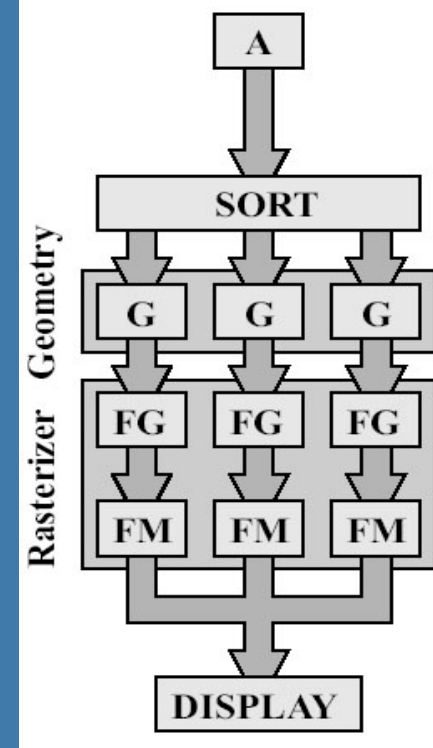
Sort-Middle

- 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) 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
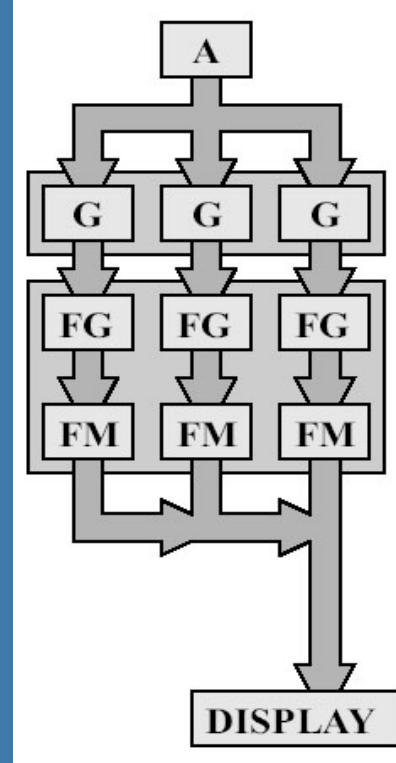- A triangle can be sent to many FG's depending on overlap (over tiles)
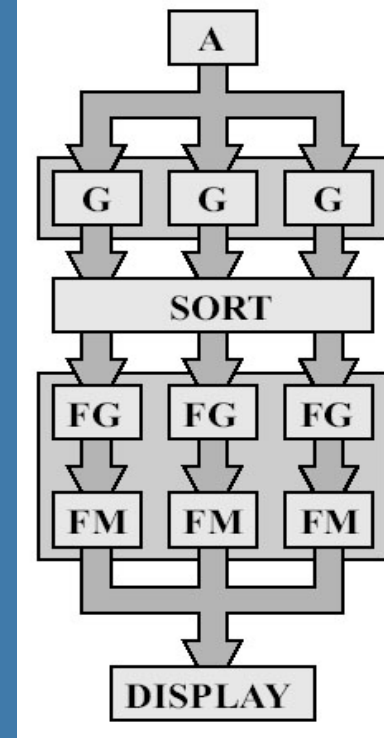
Sort-Last Fragment

- Sorts 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 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
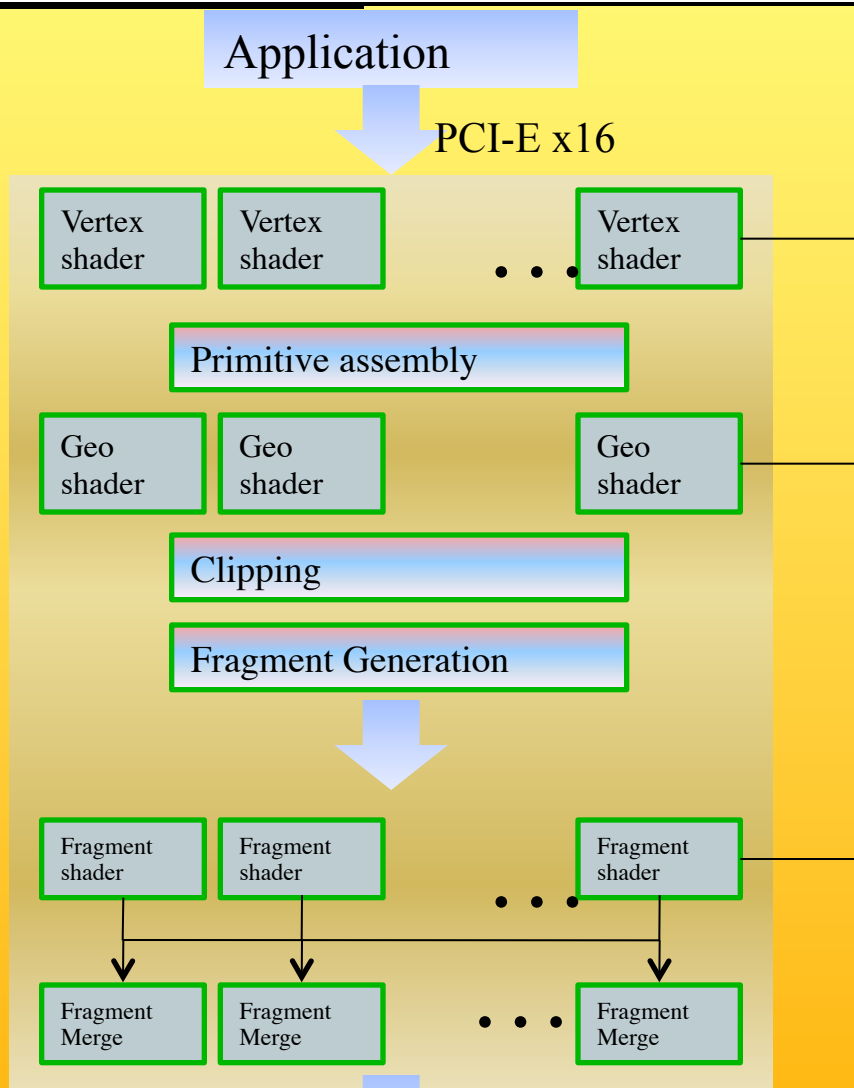   - Results in extra work
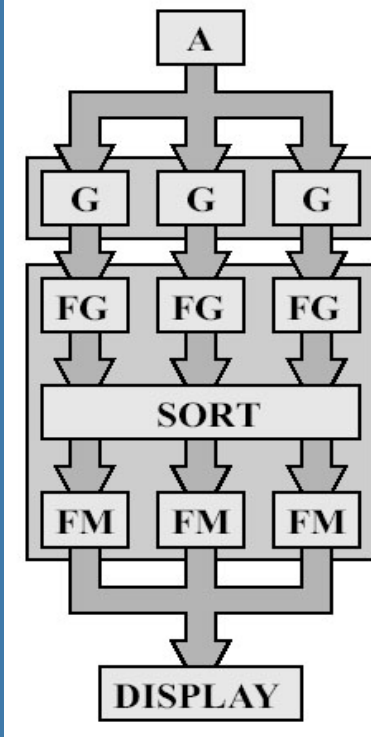
Sort-Last Image

- 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 commerically

Logical layout of a graphics card:

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 |

| Fragment Merge | Fragment Merge | . . . | Fragment Merge |

Display

On NVIDIA 8000/9000/200/400/600-series: Vertex-, Geometry- and Fragment shaders allocated from a pool of 128/240/480/1536 processors
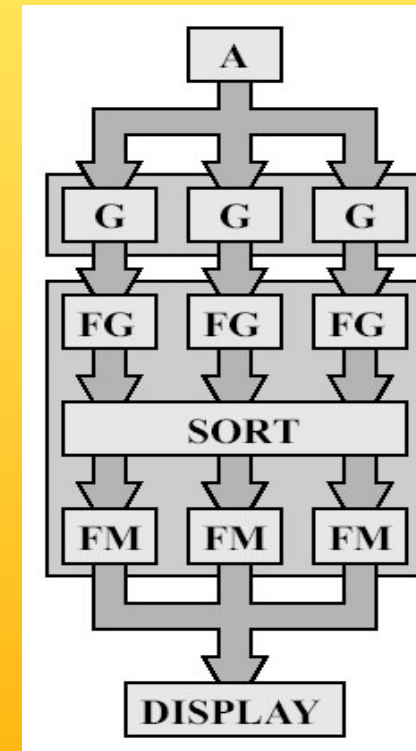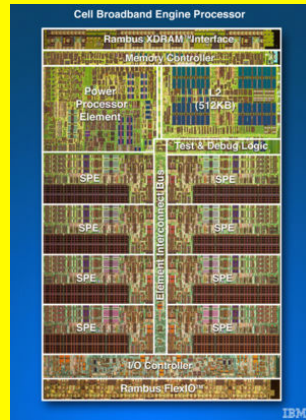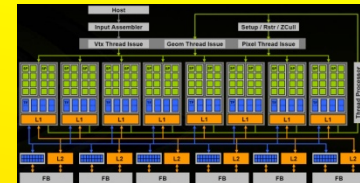
Current and Future Multicores in Graphics

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

- 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

PowerXCell 8i Processor – 2008
- 8 cores à 4-float SIMD
- 256KB L2 cache
- 128 entry register file
- but has better double precission support

Intel Xeon Phi

- ## Knights Corner

Figure 4: Knights Corner Core

Figure 5: Knights Corner Microarchitecture

http://www.tomshardware.com/reviews/xeon-phi-larrabee-stampede-hpc,3342-3.html

Need to know:

Linearly interpolate $(u_i/w_i, v_i/w_i, 1/w_i)$ in screenspace from each triangle vertex i.
Then at each pixel:
$$u_{ip} = (u_{ip}/w_{ip}) / (1/w_{ip})$$
$$v_{ip} = (v_{ip}/w_{ip}) / (1/w_{ip})$$

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

- Perspective correct texturing
- Taxonomy:
  - Sort first
  - sort middle
  - sort last fragment
  - sort last image
- Bandwidth
  - Why it is a problem
  - How to "solve" it
- Be able to sketch the architecture of a moder graphics card
  - As visualized on the next slide

Sort-first

Sort-middle

Sort-last fragment

Sort-last image

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

Fragment Merge   Fragment Merge   • • •   Fragment Merge

Display

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