CHALMERS

Graphics Hardware

Ulf Assarsson

Graphics hardware – why?

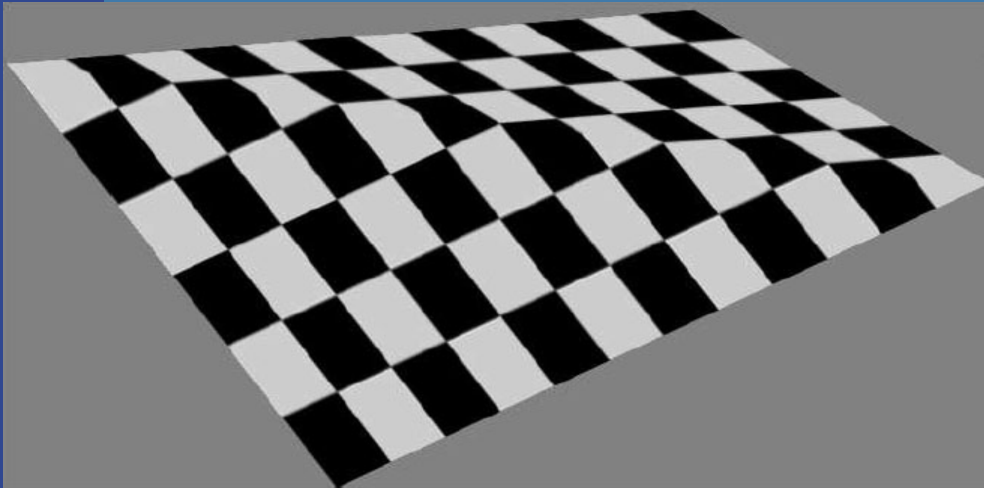
- About 100x faster!
- Another reason: about 100x faster!
- Simple to pipeline and parallelize

- 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 OptiX

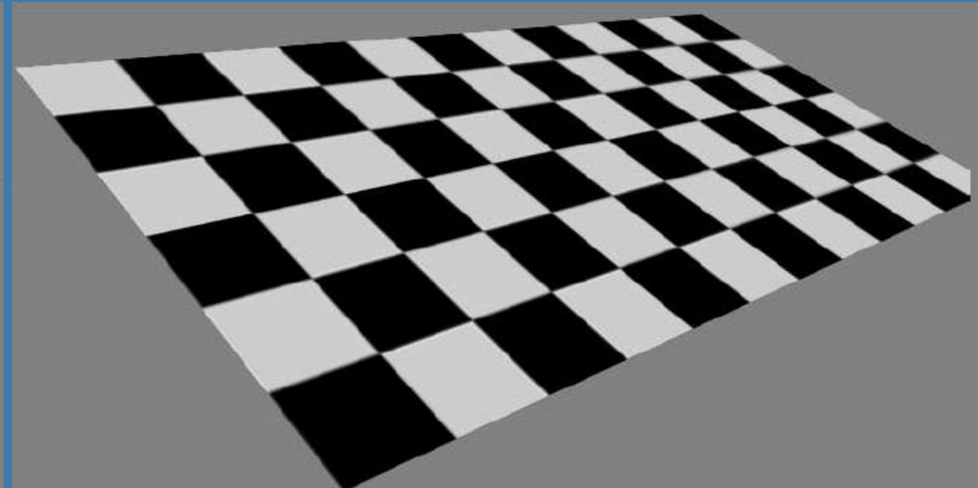
POSSESSION
P
HOTEL
Possession is a registered trademark of Blitz Games Limited © Blitz Games 2005
PS3 PORTAL

Perspective-correct texturing

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

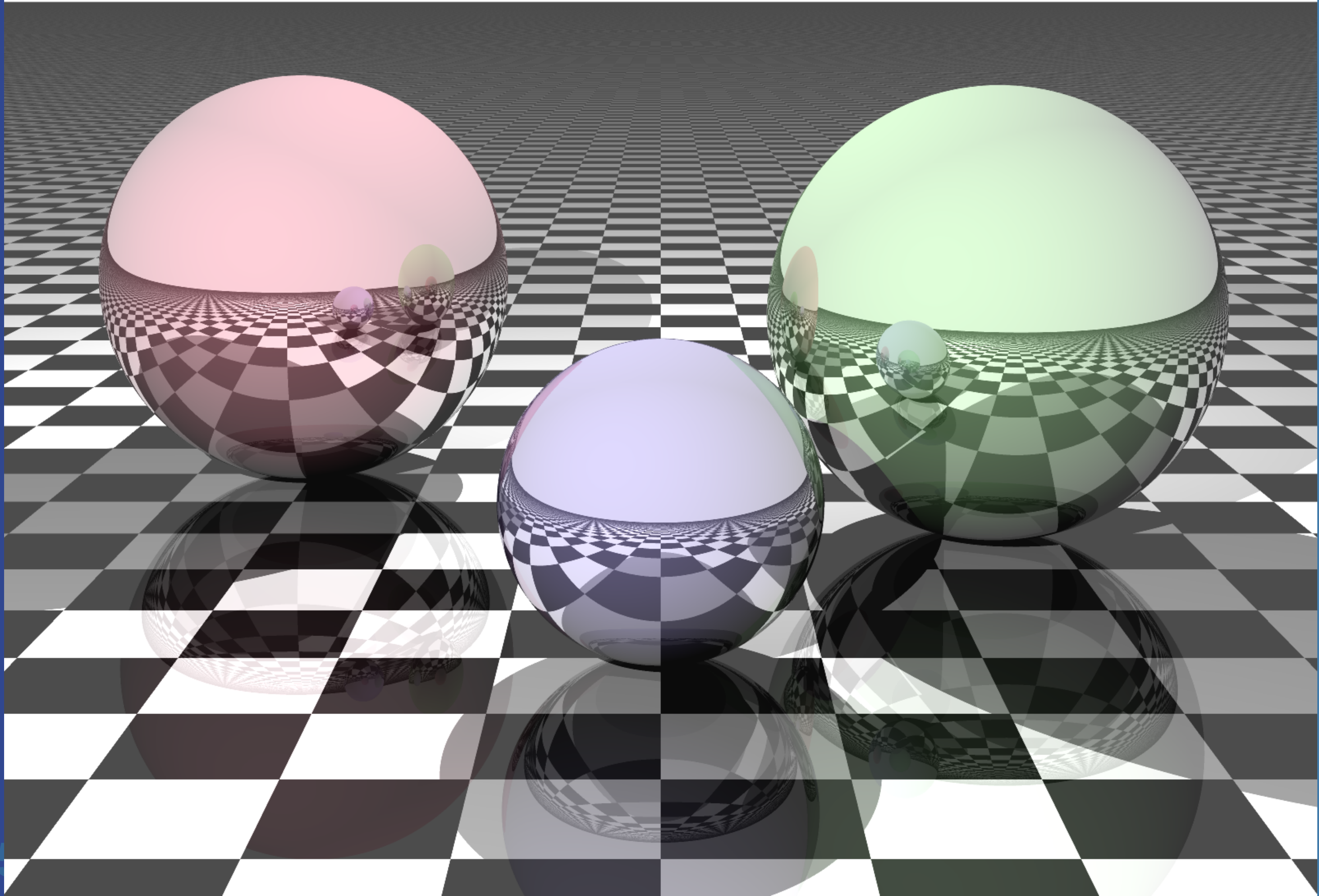


Linear interpolation



Perspective-correct interpolation

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



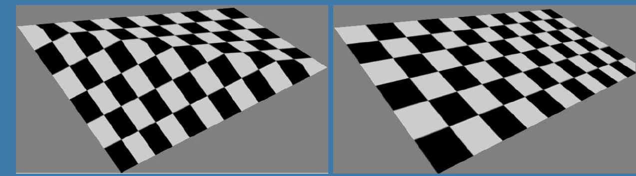
Recall the following

- 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 $(p_x', p_y', p_z', 1)$

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
- 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:



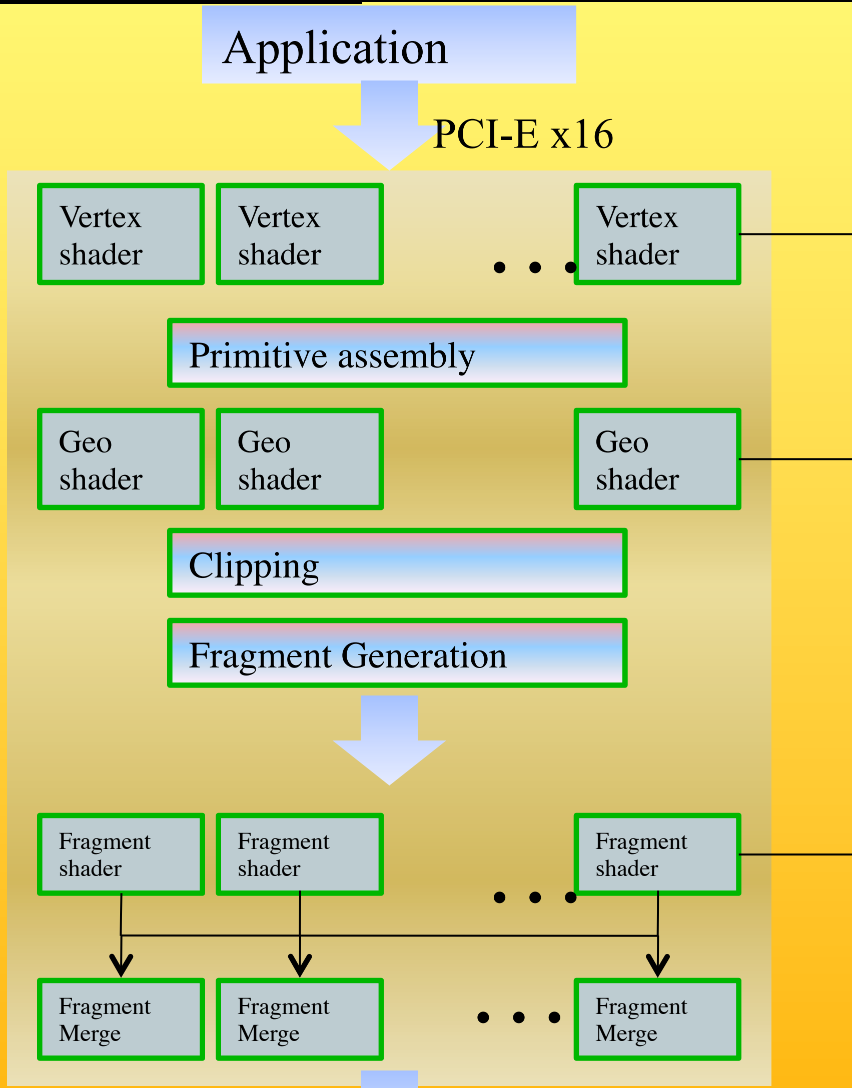
- Linear interpolation in screen space does not work for u,v
- Solution:
  - We have applied a non-linear transform to each vertex (x/w, y/w, z/w).
    - Non-linear due to 1/w – factor from the homogenisation
  - We must apply the same non-linear transform to u,v
    - E.g. (u/w, v/w). This can now be 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

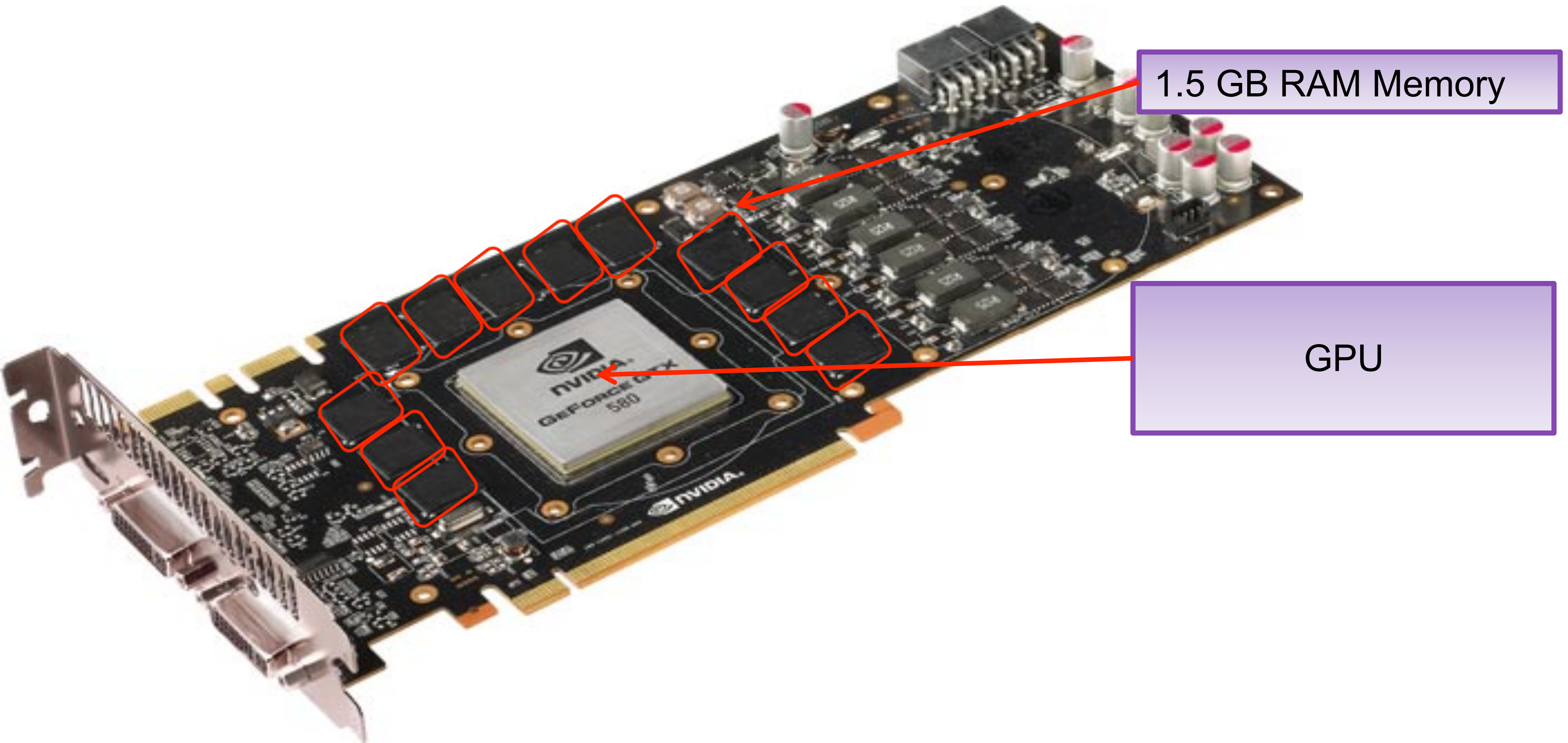


On NVIDIA 8000/9000/200/ 400/500-series:

Vertex-, Geometry- and Fragment shaders allocated from a pool of 128/240/480/512 processors

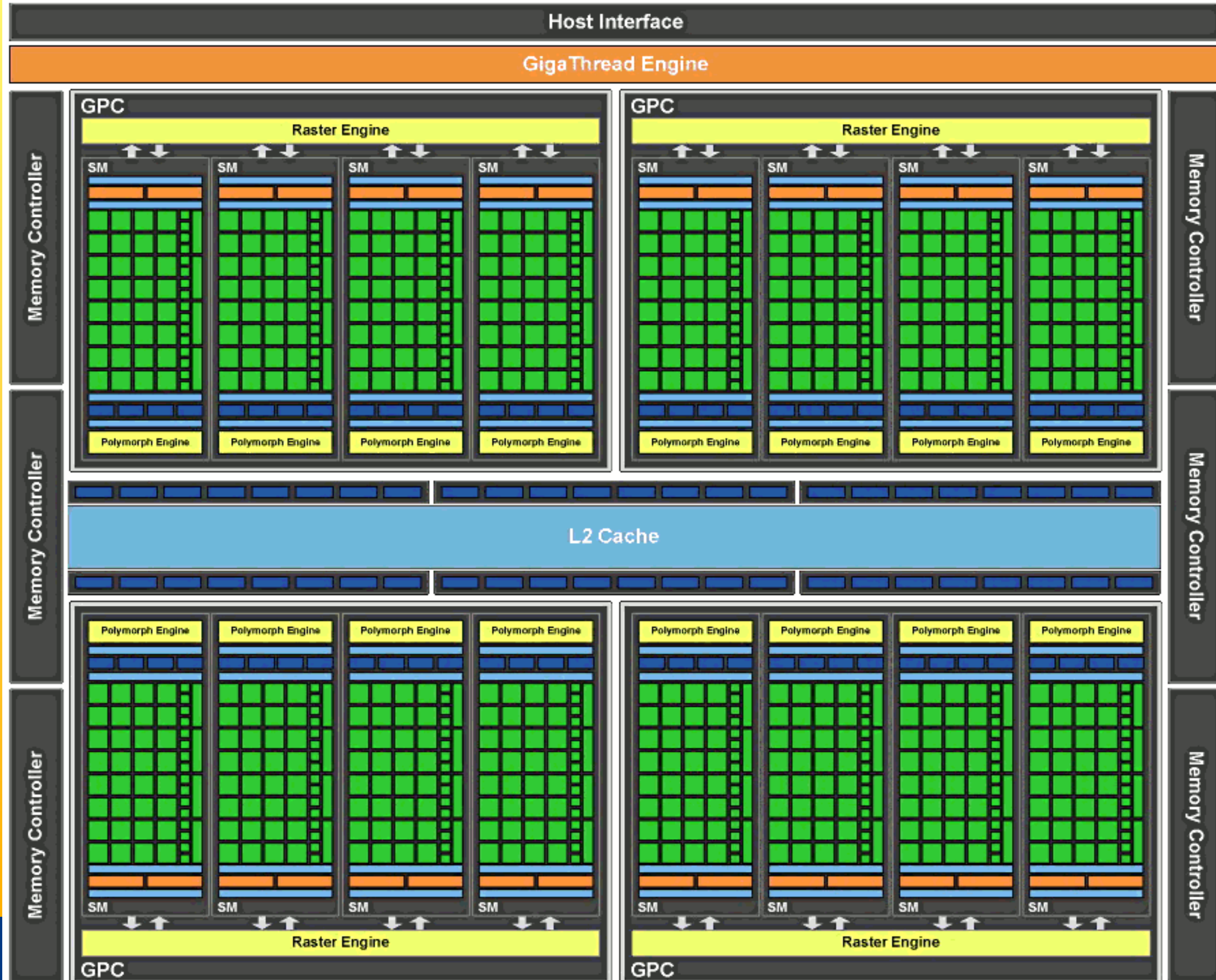
GeForce 7800

Graphics Processing Unit – GPU



- NVIDIA Geforce GTX 580

NVIDIA GT300 (GF100)

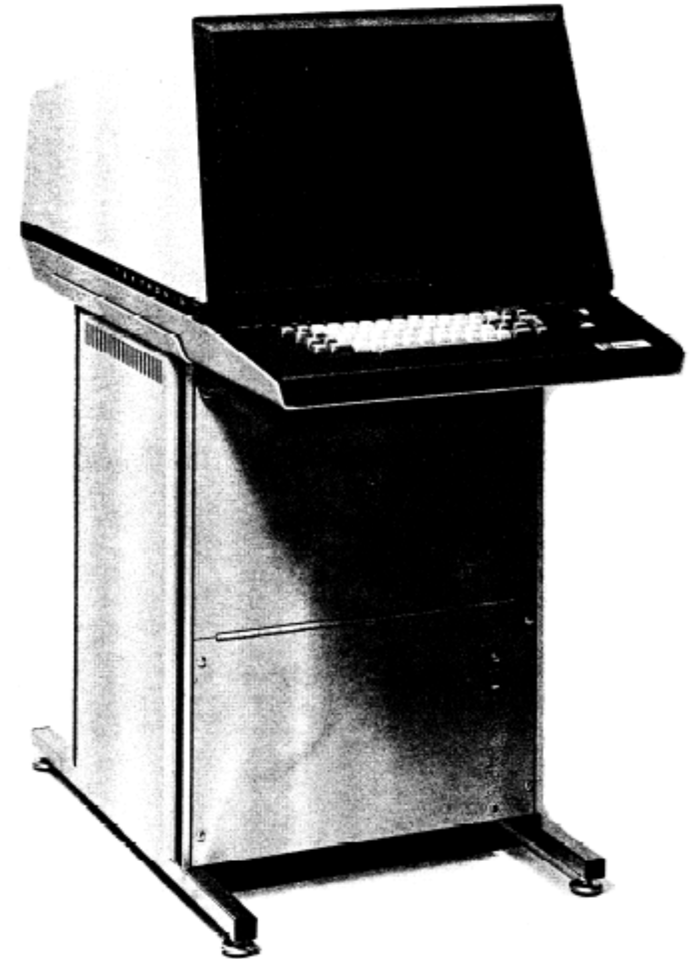


Graphics Hardware History

- 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
- 10' NVIDIA's Fermi / ~~Intel's Larrabee~~
  - More general multiprocessor systems, ~16-24 proc à 16 SIMD, L1/L2 cache

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.

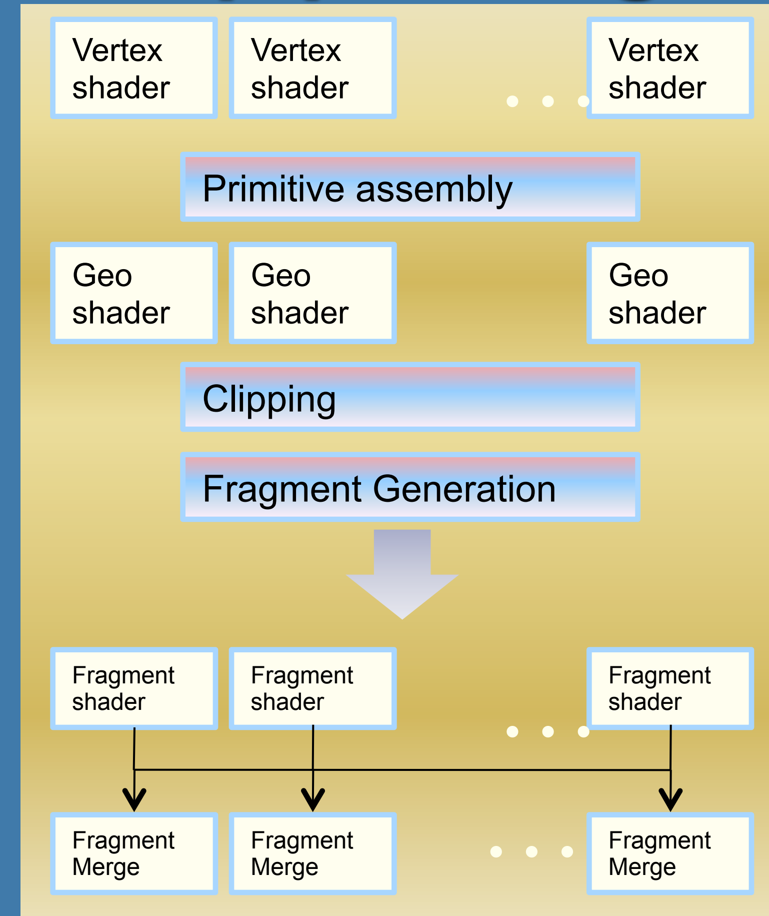
Briefly about Graphics HW pipelining

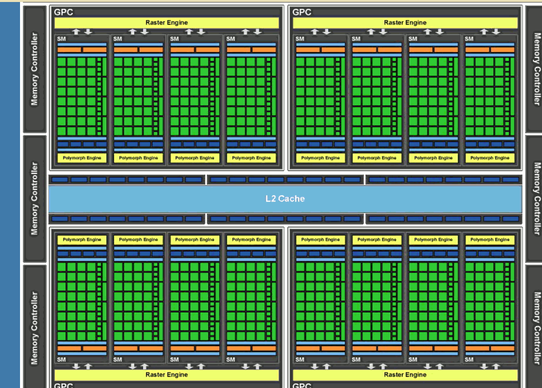
2001
- In GeForce3: 600-800 pipeline stages!
  - 57 million transistors
  - First Pentium IV: 20 stages, 42 million transistors,
  - Core2 Duo, 271 Mtrans, Intel Core 2 Extreme QX9770 – 820Mtrans.
  - Intel Pentium D 900, 376M trans, Intel i7 (quad): 731Mtrans, 10-core Xeon Westmere: 2.6Gtrans
- Evolution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

  - Lesson learned: #trans doubles ~per year. Core clock increases slowly. Mem clock –increases with new technology DDR2, DDR3, GDDR5

Briefly about Graphics HW pipelining

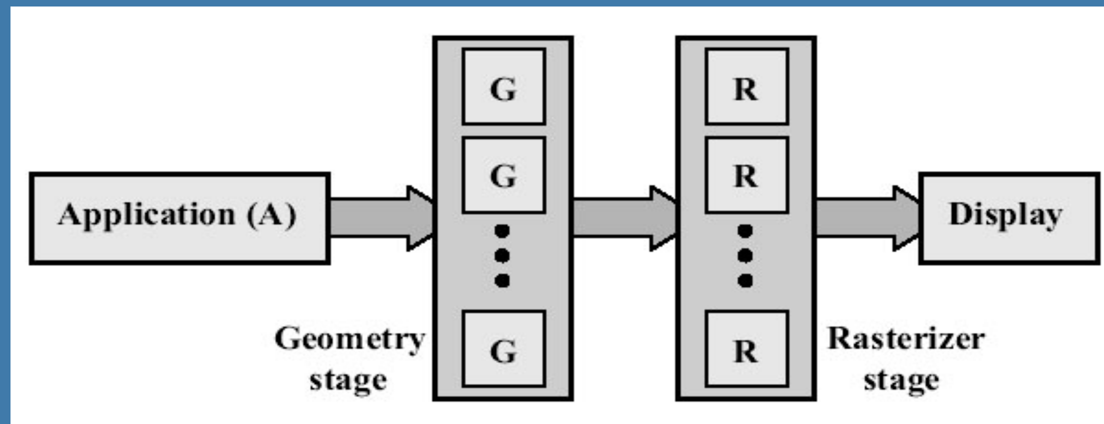
- Ghw speed doubles ~6-12 months,
  CPU speed doubles ~18 months
- Ideally: n pipeline stages → n times throughput
  - But latency is high (may also increase)! However, not a problem here:
    - Chip runs at about 500 MHz (2ns per clock)
    - 2ns*700=1.4 μs
    - We got about 20 ms per frame (50 fps)
- Graphics hardware is simpler to pipeline and parallelize because:
  - Pixels are (most often) independent of each other
  - Few branches and much fixed functionality
  - Don't need high clock freq: bandwidth to memory is bottleneck
    - This is changing with increased programmability
  - Simpler to predict memory access pattern (do prefecthing!)





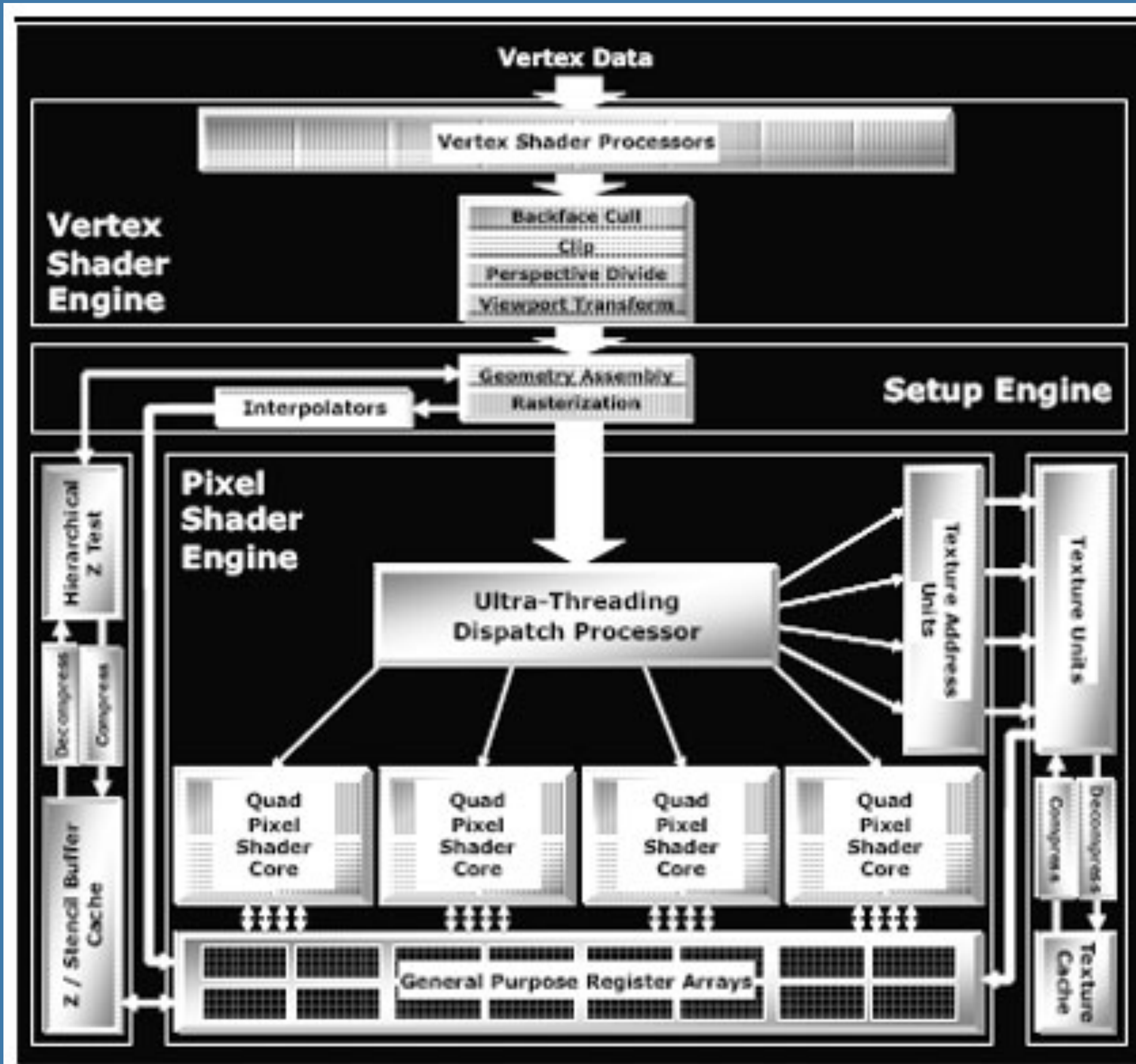
Parallellism

- ”Simple” idea: compute n results in parallel, then combine results
- NVIDIA GTX580: ≤ 512 pixels/clock
  - Many pixels are processed simultaneously
- Not always simple!
  - Try to parallelize a sorting algorithm…
  - But pixels are independent of each other, so simpler for graphics hardware
- Can parallellize both geometry and rasterizer:



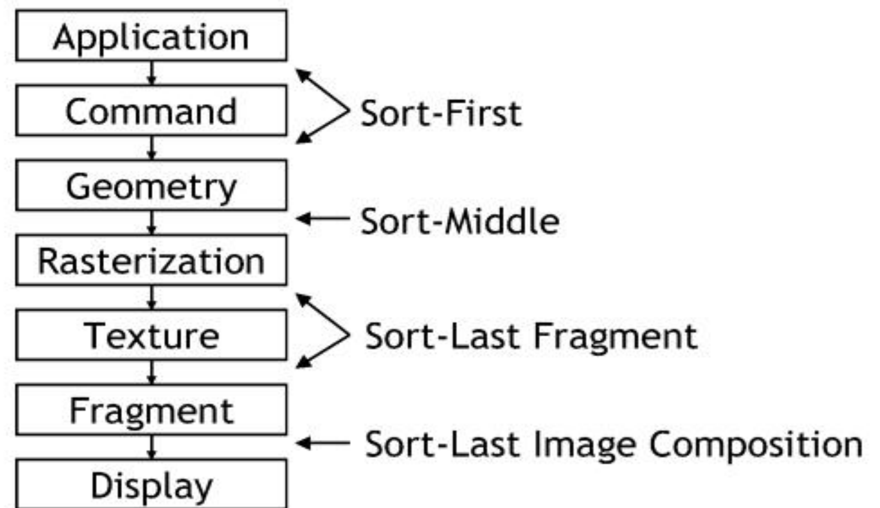
Example: ATI X1800

- 8 vertex shaders
- 16 pixel shaders
- SIMD rgba,xyzw



Taxonomy of hardware

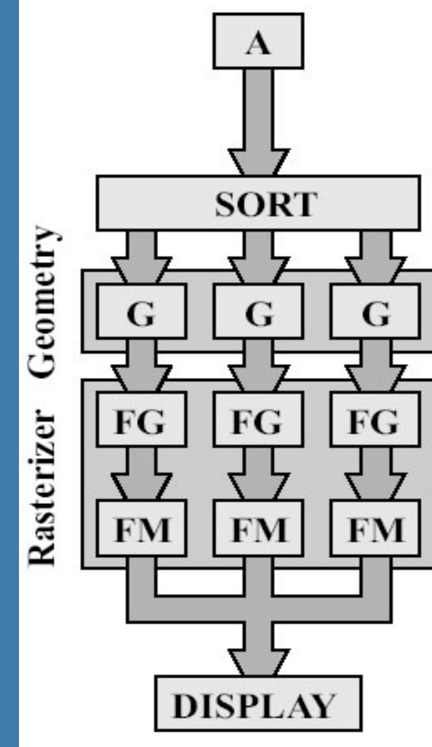
- Need to sort from model space to screen space
- Gives four major architectures:
  - Sort-first
  - Sort-middle
  - Sort-Last Fragment
  - Sort-Last Image



- Will describe these briefly, and then focus on sort-middle and sort-last fragment (used in commercial hardware)

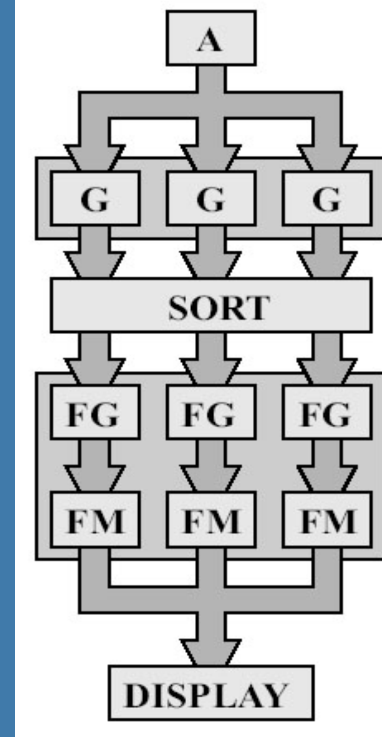
Sorting/dividing work to parallel execution units.

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

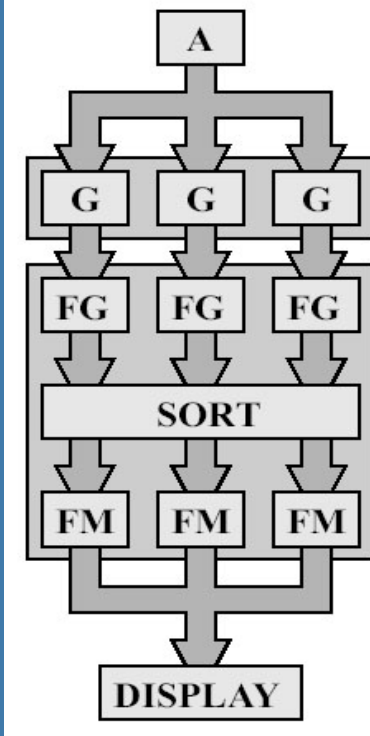
Sort-Middle



- 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)

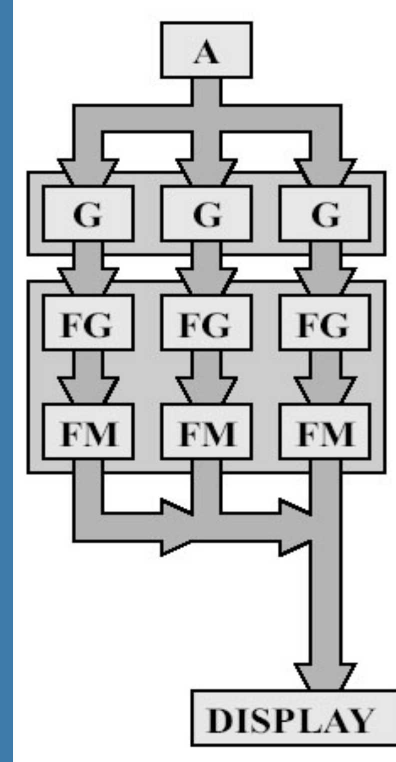
Sort-Last Fragment

- 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)



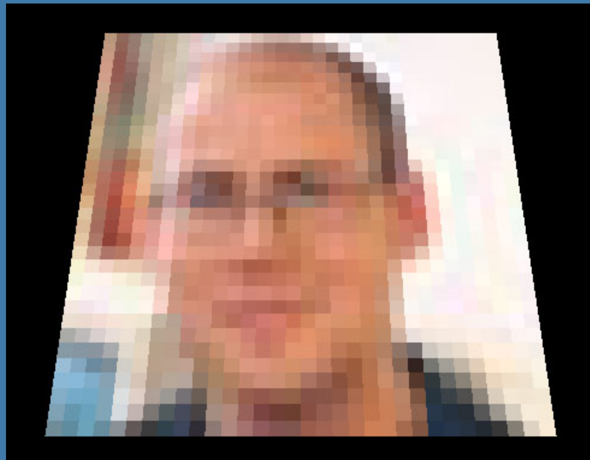
- After all primitives have been sent to the pipeline, the z-buffers and color buffers are merged into one color buffer
- Can be seen as a set of independent pipelines
- Huge memory requirements!
- Used in research, but probably not commerically

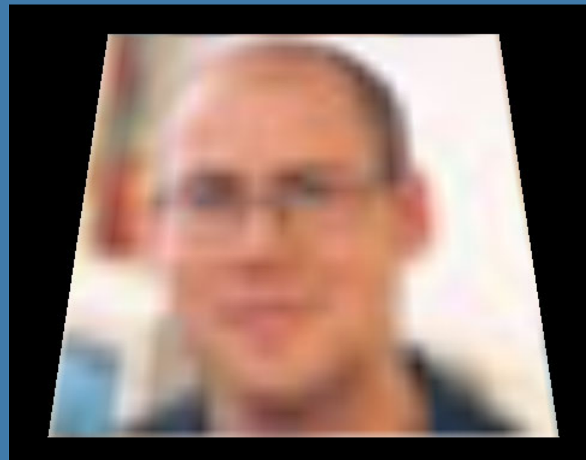
Memory bandwith usage is huge!!

Mainly due to texture reads

FILTERING:

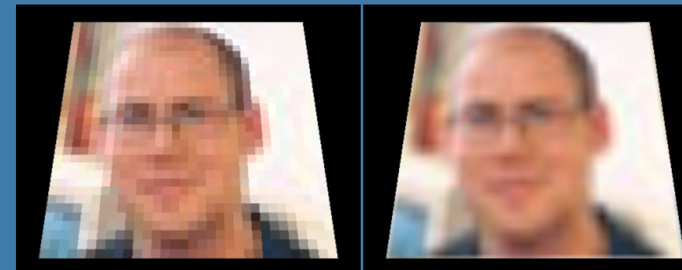
- For magnification: Nearest or Linear (box vs Tent filter)



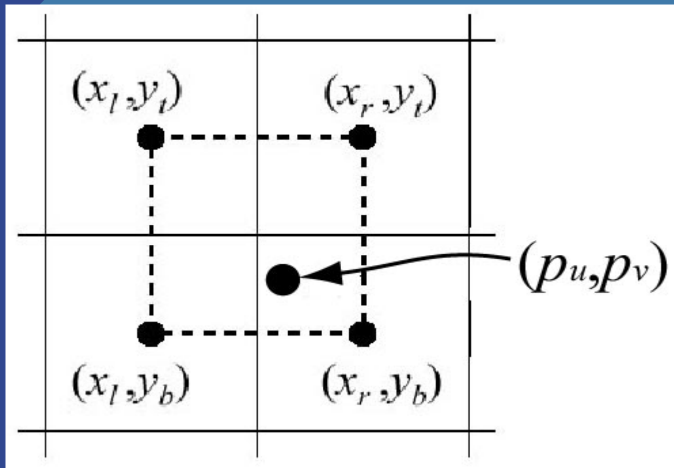


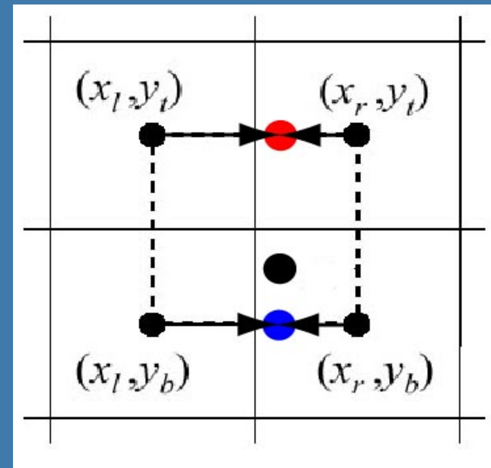
- For minification:
  - Bilinear – using mipmapping
  - Trilinear – using mipmapping
  - Anisotropic – some mipmap lookups along line of anisotropy

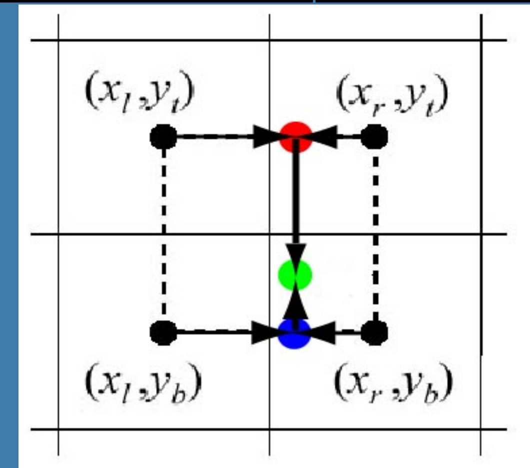
Interpolation



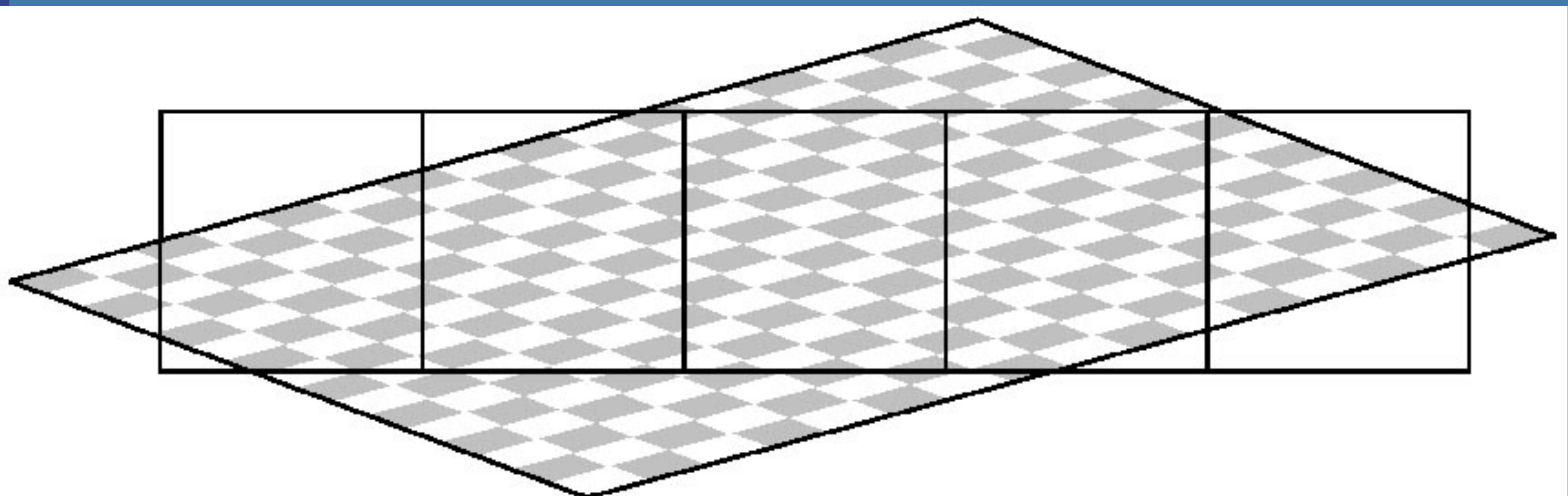
Magnification



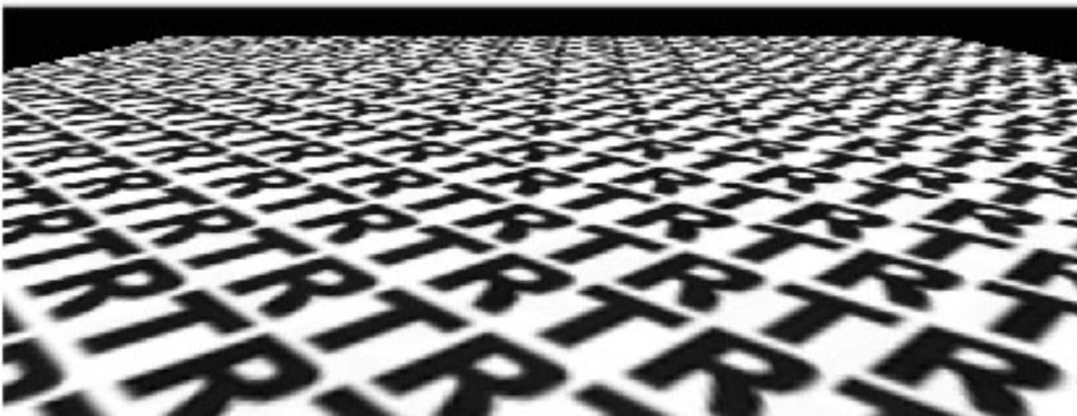


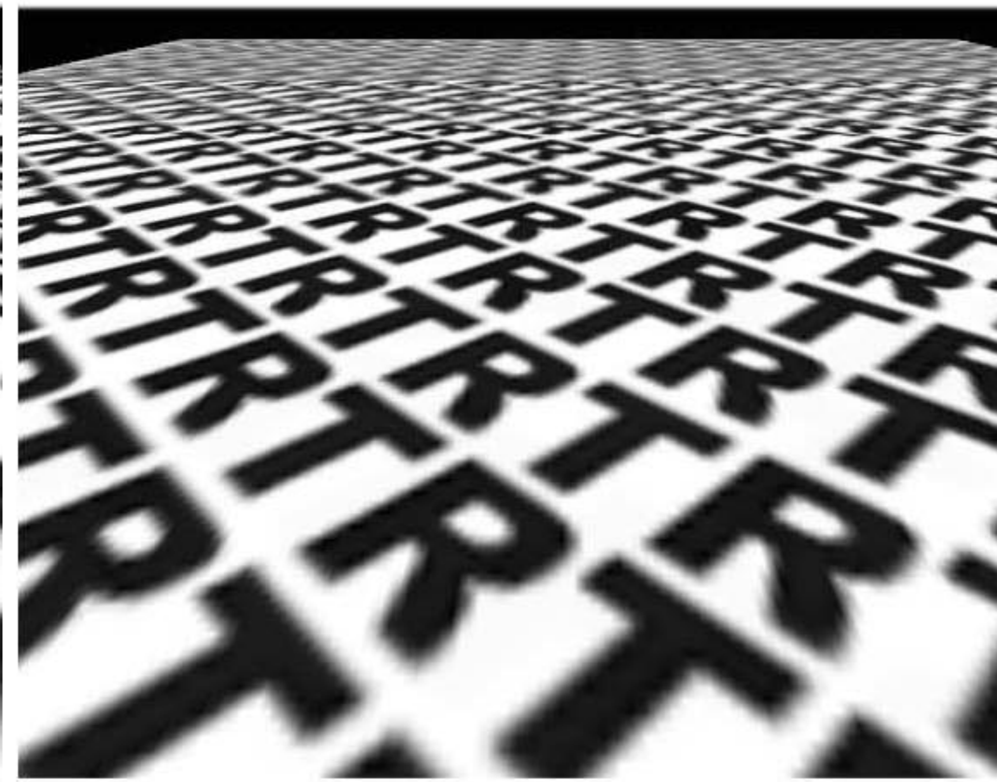


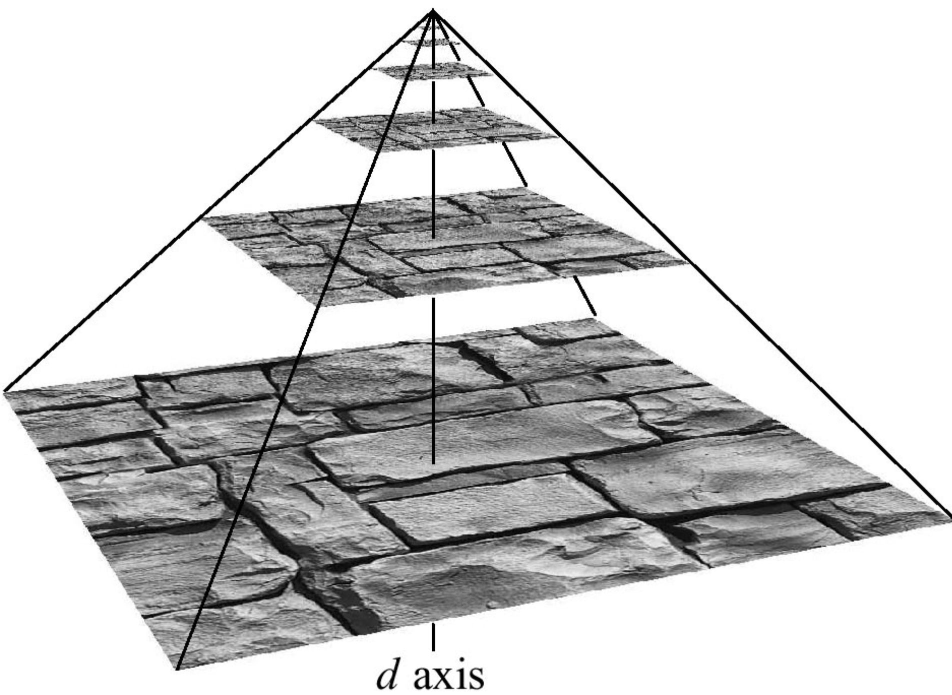
Minification



Bilinear filtering using Mipmapping







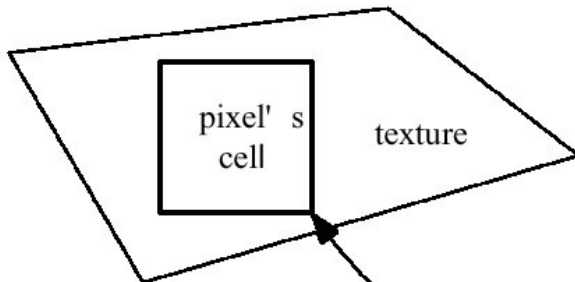
Anisotropic texture filtering

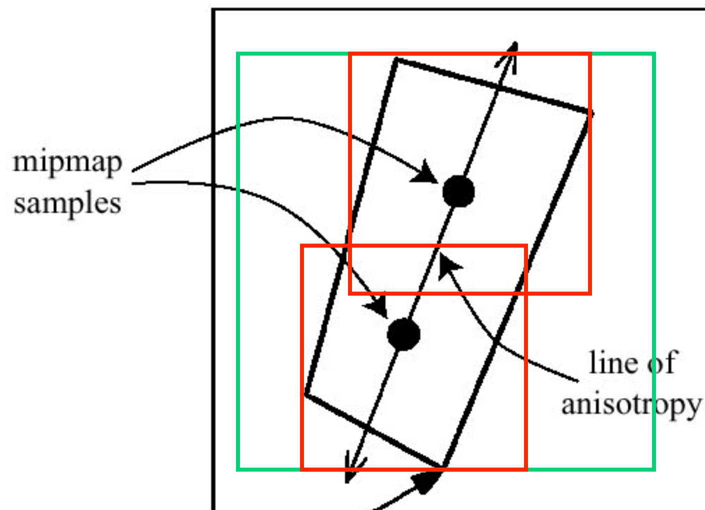
pixel space

texture space

And we haven't even used float-textures yet...

nor 3D textures…





Wish list:

1 sample = 32 bytes (or 512 for 16x ani. filter.)

512/4(rgba) proc * 1544MHz * 32 bytes = 6.3 GB/s (101TB/s)

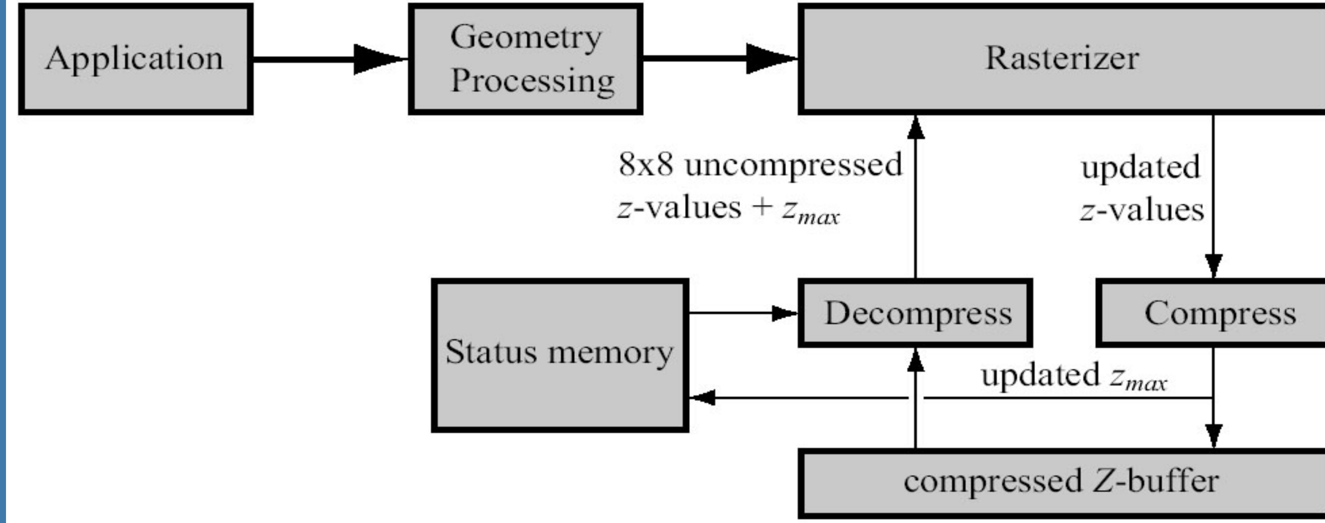
Memory bandwidth usage is huge!!

- Assume GDDR5 (read/write twice per clock) at 2.004 MHz, (256+128=384) bits per access: => 192.4 Gb/s
- On top of that bandwith usage is never 100%, and anti-aliasing (supersampling), will use up bandwidth to frame buffers as well as texture mem.
- 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

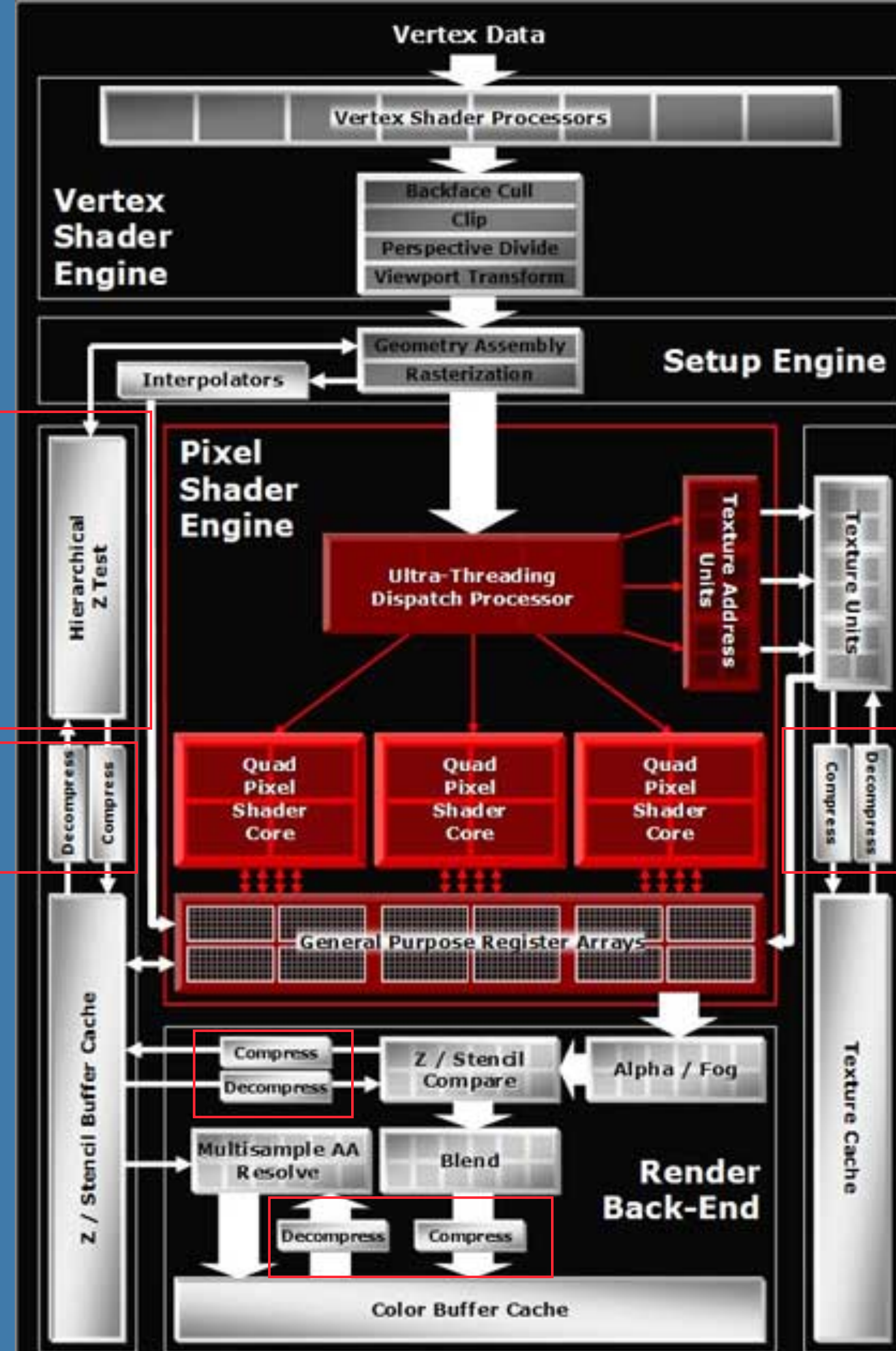
X1800 GTO

- Real example

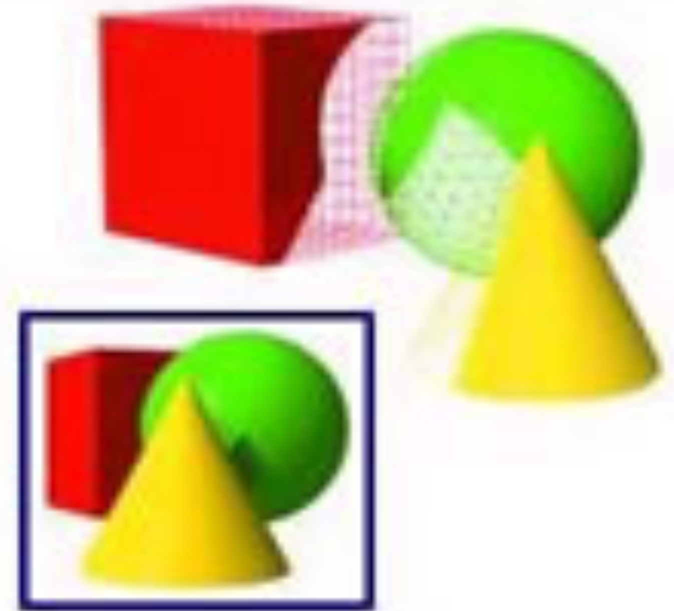
Z-cull

Z-compress

Also note texture compress and color compress



KYRO II



Added by Ulf Assarsson 2004

KYRO II

32x16 Pixel

Added by Ulf Assarsson 2004

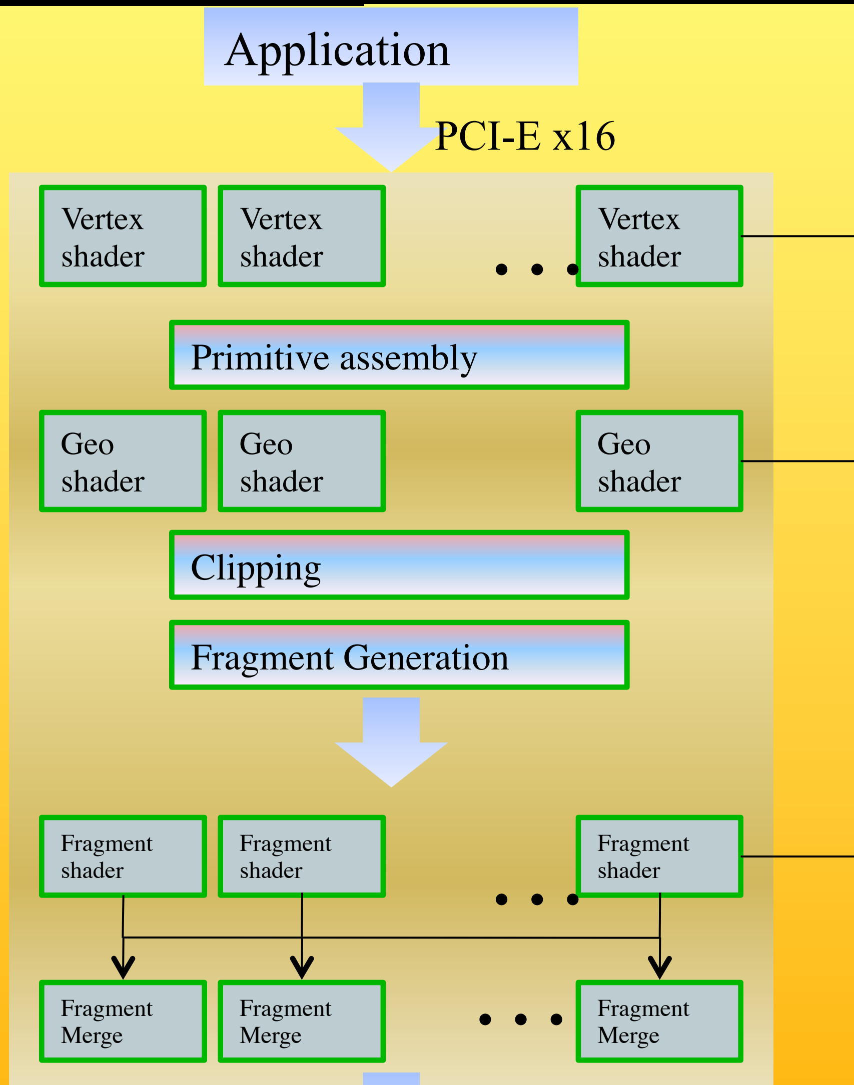
KYRO – a different architecture

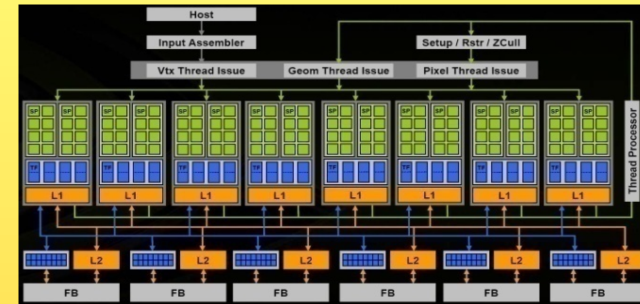
- Based on cost-effective PowerVR architecture
- Tile-based
  - For KYRO II: 32x16 pixels
- Fundamental difference
  - For entire scene, do this:
  - Find all triangles inside each tile
  - Render all triangle inside tile
- Advantage: can implement temporary color, stencil, and Z-buffer in fast on-chip memory
- Saves memory and memory bandwidth!
  - Claims to save 2/3 of bandwidth compared to traditional architecture (without Z-occlusion testing) thanks to deferred shading
- Disadvantage: Need to store scene in local card memory. 3 MB can handle a little over 30,000 triangles

KYRO: pros and cons

- Uses a small amount of very fast memory
  - Reduces bandwidth demands greatly
  - Reduces frame buffer memory greatly
- But more local memory is needed
  - For tile sorting
  - Amount of local memory places a limit on how many triangles can be rendered
  - 3 MB can handle a little over 30,000 triangles

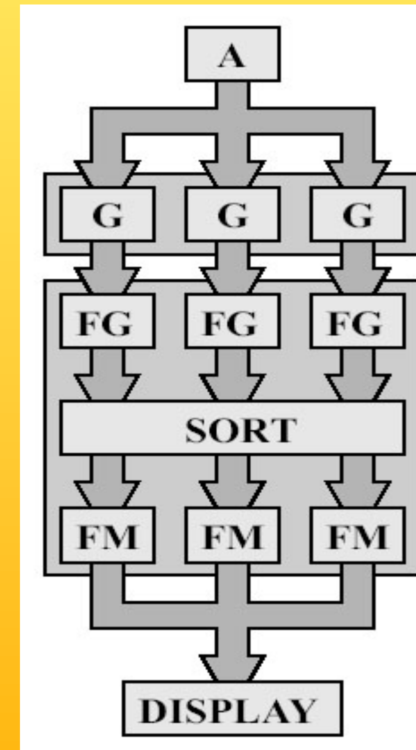
Logical layout of a graphics card:





On NVIDIA 8000/9000/200/ 400-series:

Vertex-, Geometry- and Fragment shaders allocated from a pool of 128/240/480 processors

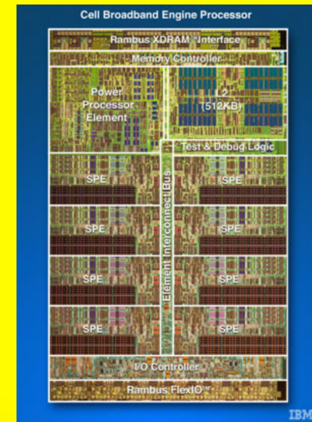


Current and Future Multicores in Graphics

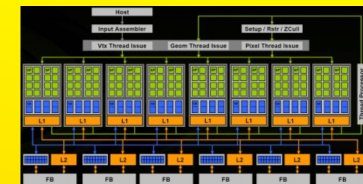
- Cell – 2005
  - 8 cores à 4-float SIMD
  - 256KB L2 cache/core
  - 128 entry register file
  - 3.2 GHz

PowerXCell 8i Processor – 2008

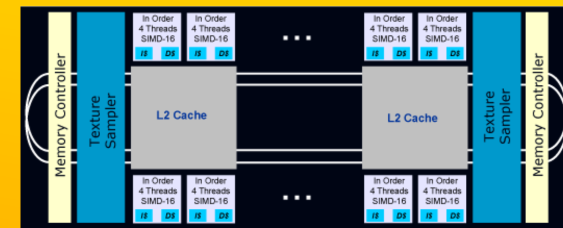
- 8 cores à 4-float SIMD
- 256KB L2 cache
- 128 entry register file
- but has better double precission support



- 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

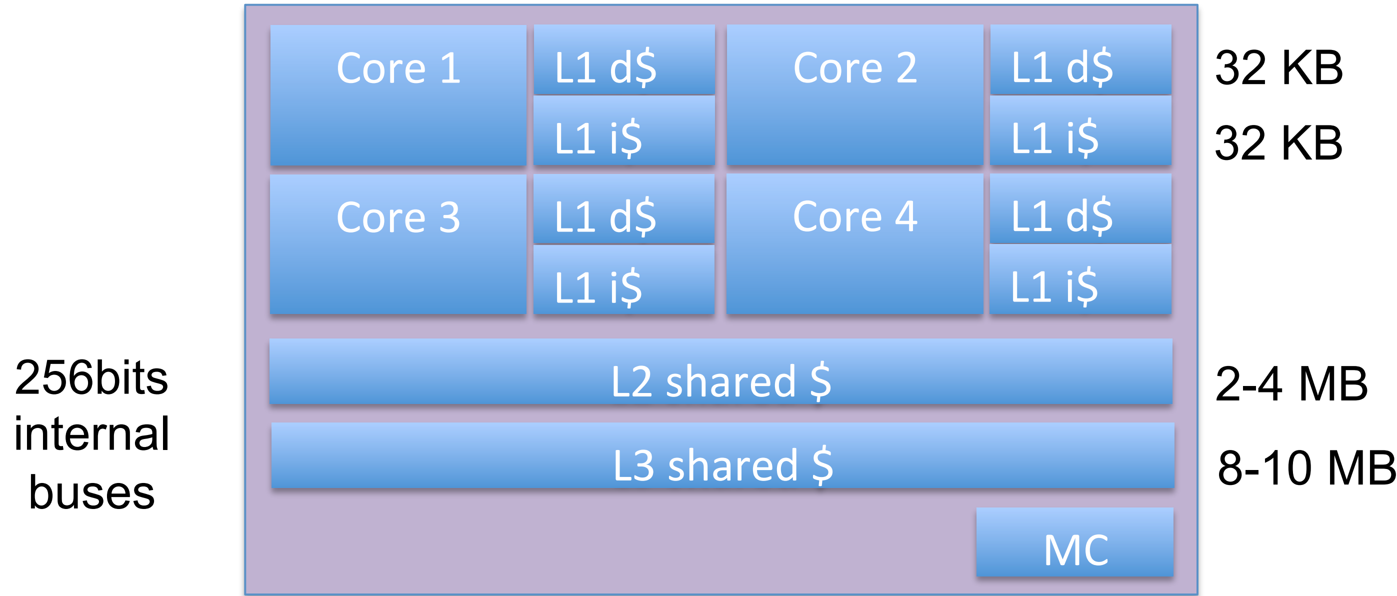


- Larrabee – 2010 ?
  - 16-24 cores à 16-float SIMD
  - Core = 16-float SIMD (=512bit FPU) + x86 proc with loops, branches + scalar ops, 4 threads/core
  - 32KB L1cache, 256KB L2-cache
  - 1.7-2.4 GHz
- NVIDIA Fermi GF100 - 2010
  - 16 cores à 2x16-float SIMD (1x16 double SIMD)
  - 16+48 KB L1 cache, 768 KB L2 cache
  - 1401 - 1544 MHz core



CPU - 2011

AVX:
Intel's Sandybridge
AMD's Bulldozer



32 KB

32 KB

2-4 MB

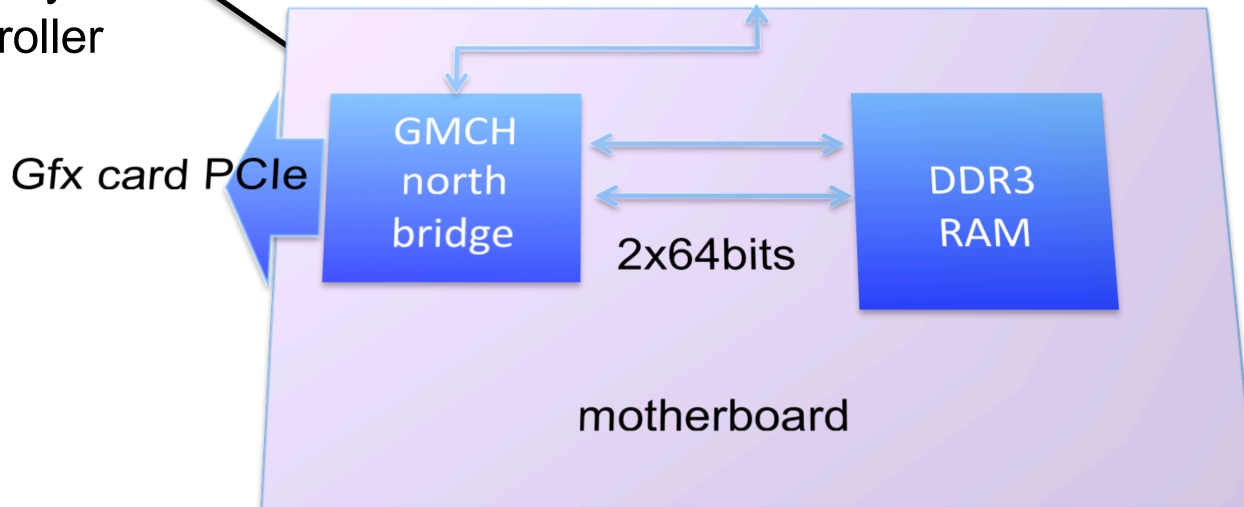
8-10 MB

256bits internal buses

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

FSB

Graphics Memory Controller HUB



Wish list:

- 4-8 cores à 4 floats

⇒ 128 bytes/clock

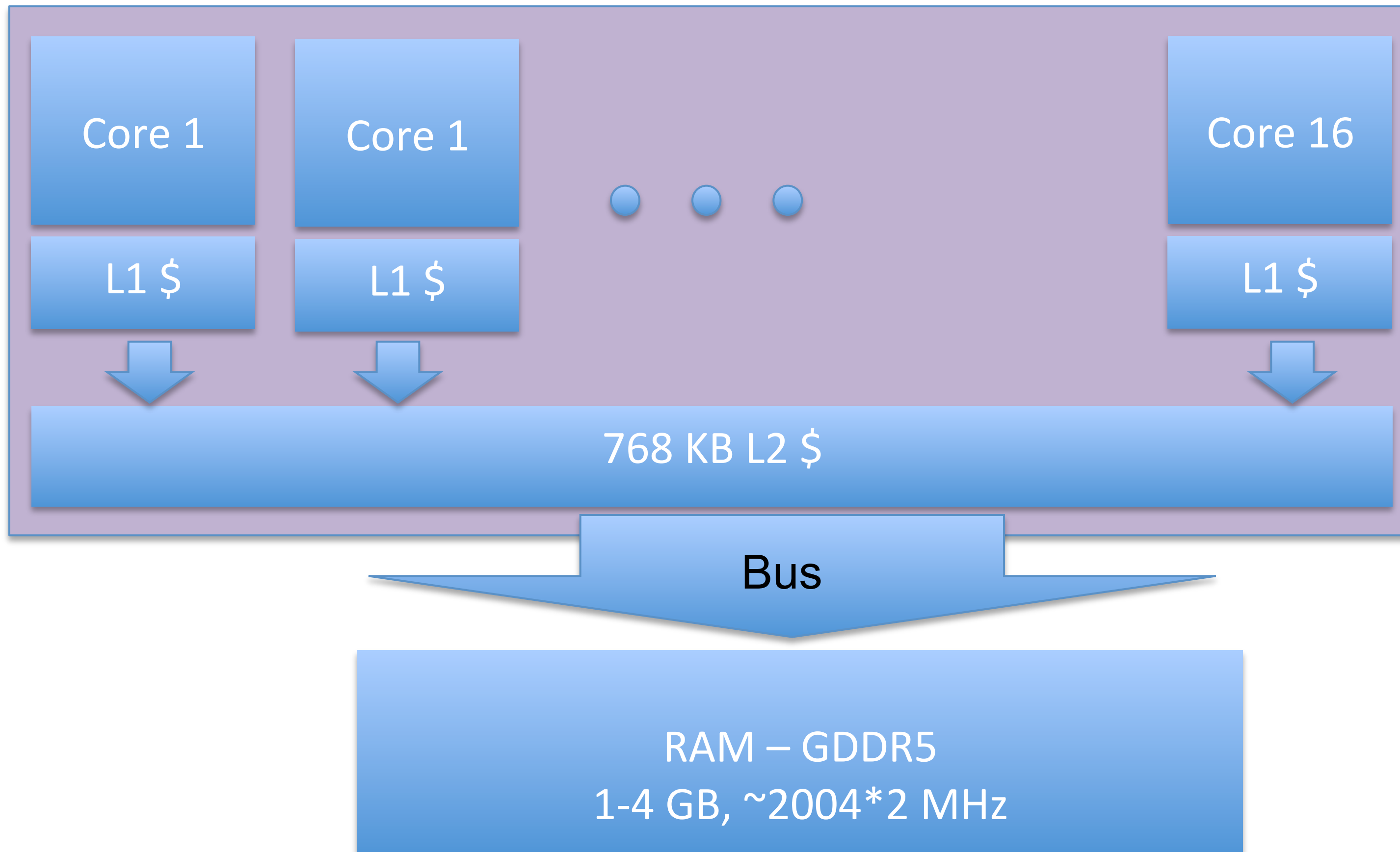
⇒ 128GByte/s

- In addition, x3, since:

r1 = r2 + r3;

GPU- Nvidia's Fermi 2010

Overview:



16 cores à 2x16-SIMD width

16/48 KB each

Bandwidth ~192 GB/s

Bus: 256/384 bits

Compare ATI 2900:
- 2x512bits

Larrabee:
- 2x512bits

Wish:

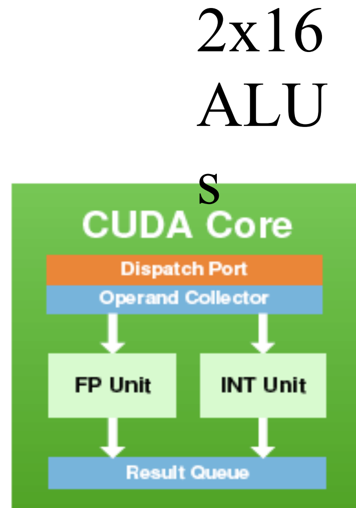
512 ALUs à 1 float/clock => 2KB/clock

~1.5GHz core clock => 3000 GB/s request

We have ~192 GB/s. In reality we can do 20-40 instr. between each RAM–read/write.

One SIMD-processor(NVIDIA)

- 512 “CUDA stream cores” (= ALUs)
- All threads execute the same program
- But each thread knows block and thread ID
  - So we can branch on this
    - Leads to efficiency issues since every thread performs same operation



one SIMD processor

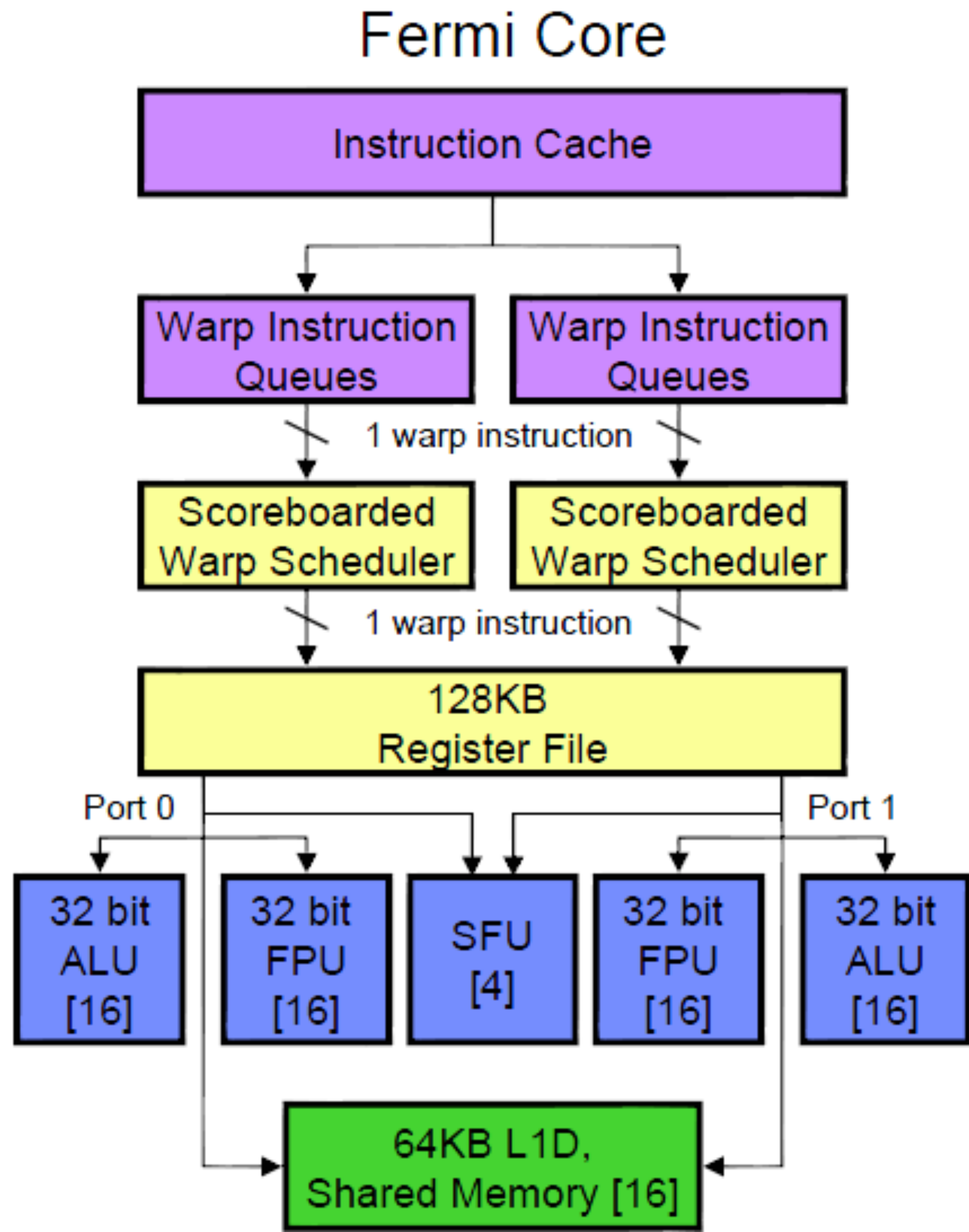
Instruction Cache
Warp Scheduler | Warp Scheduler
Dispatch Unit | Dispatch Unit
Register File (32,768 x 32-bit)
Core (x32) | LD/ST (x16) | SFU (x4)
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache

Fermi Streaming Multiprocessor (SM)

One SIMD processor

Per multiprocessor:

- 8800/GTX280:
  - 8 mul finishes per cycle
  - Takes 4 cycles
  - Result usable after 24 clock cycles
- Fermi:
  - 16 mul per cycle
- Two separate warps (=groups of 32 threads) per 16-SIMD ALU/FPU



The Future

- Faster
- More programmable
- More Multiprocessors
  - Maybe higher SIMD-width (more than 32 stream processors)
- Physics Processing Unit
  - (Havok FX uses the GPU, announced at GDC 2006)

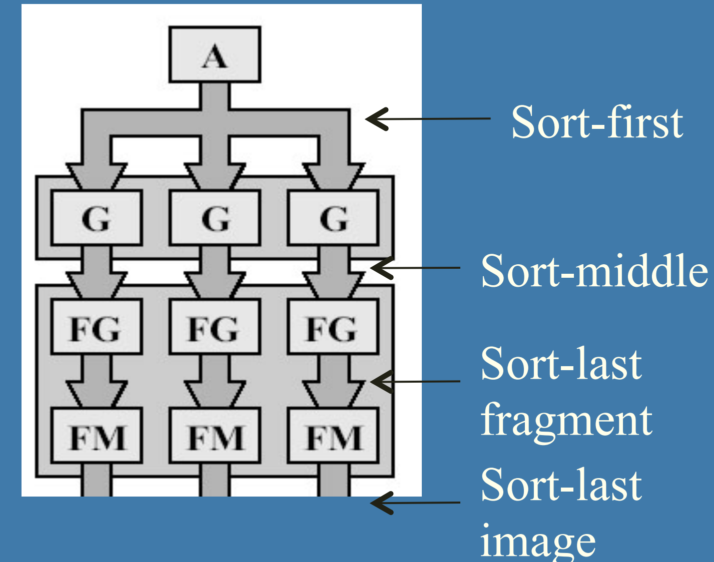
Study:

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

$$u_{ip} = (u_{ip}/w_{ip}) / (1/w_{ip})$$
$$v_{ip} = (v_{ip}/w_{ip}) / (1/w_{ip})$$

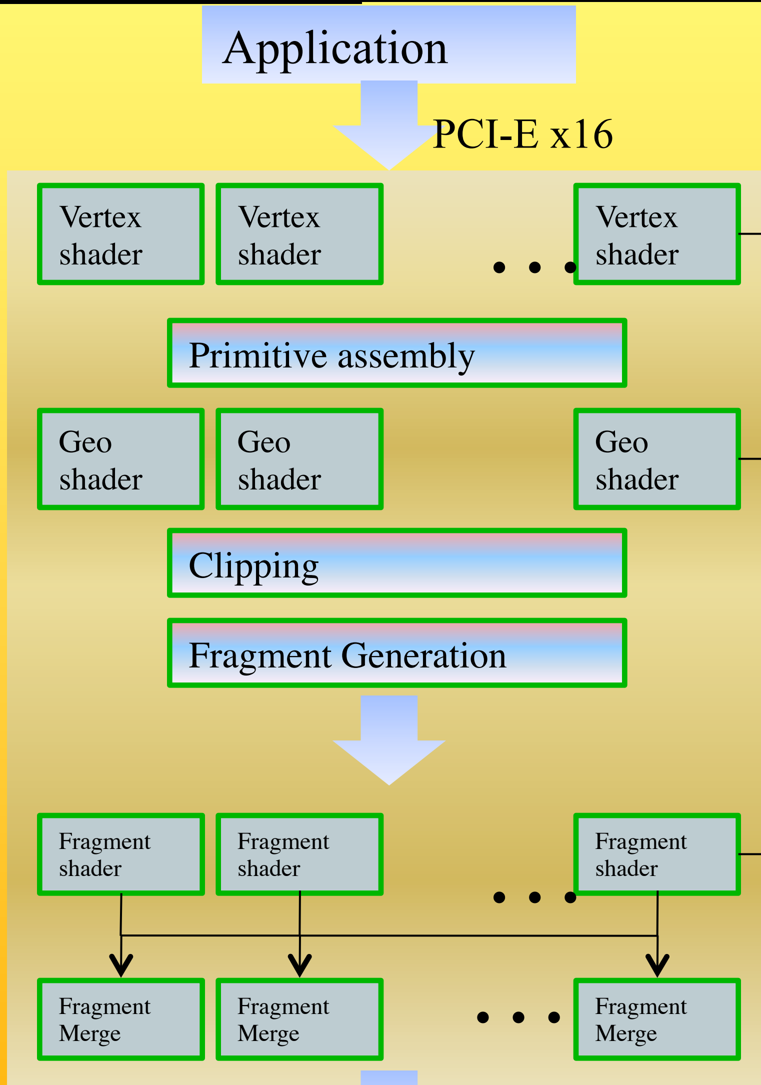
where ip = screen-space interpolated value from the triangle vertices.

- 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

Ulf Assarsson © 2004



On NVIDIA 8000-series:

Vertex-, Geometry- and Fragment shaders allocated from a pool of currently up to 512 processors