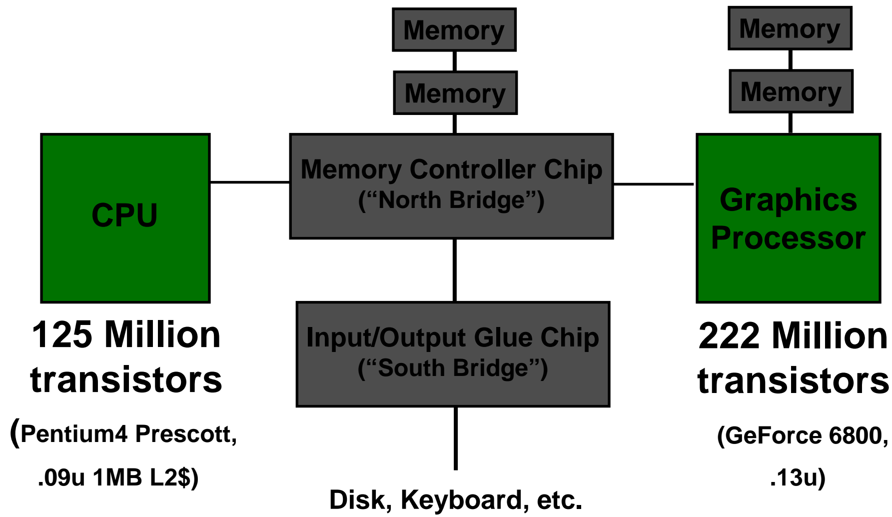
Real-Time 3D Graphics Architecture

Guest Lecture for CS 382m
Bill Mark, Oct. 26, 2005

My background

- 13 years of work in real-time graphics:
  - UNC Chapel Hill, Stanford
  - NVIDIA, SGI, Intel
- Technical lead at NVIDIA
  - Cg – a programming language for graphics HW
- Current research:
  - Future real-time graphics algorithms
  - Single-chip highly-parallel hardware architectures

Dedicated graphics chip in modern PCs



GPU has more bandwidth too

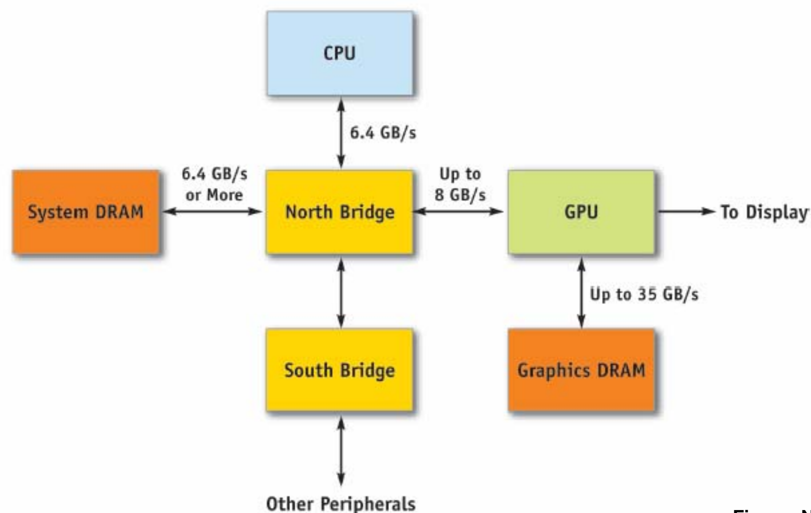


Figure: NVIDIA

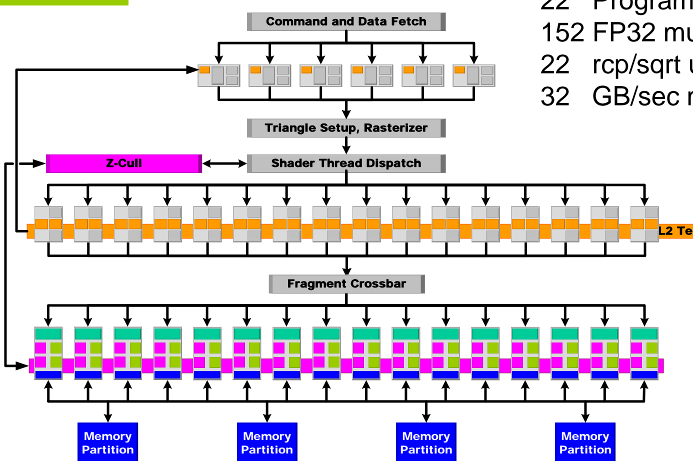
CPU vs. Graphics Peak Performance

	Pentium 4 1.06 GHz FSB	ATI Radeon X800
Clock rate	3.8 GHz	0.5 GHz
Peak GFLOPS	15.2	63.7 (fragment unit)
Memory BW	8.4 GB/sec	32 GB/sec

GFLOPS source: Fatahalian et al, GH2004

Highly parallel, single chip architecture

GeForce 6800



22 Programmable Cores
152 FP32 mult/add units
22 rcp/sqrt units
32 GB/sec memory BW

Figure: NVIDIA

Task is computationally intensive

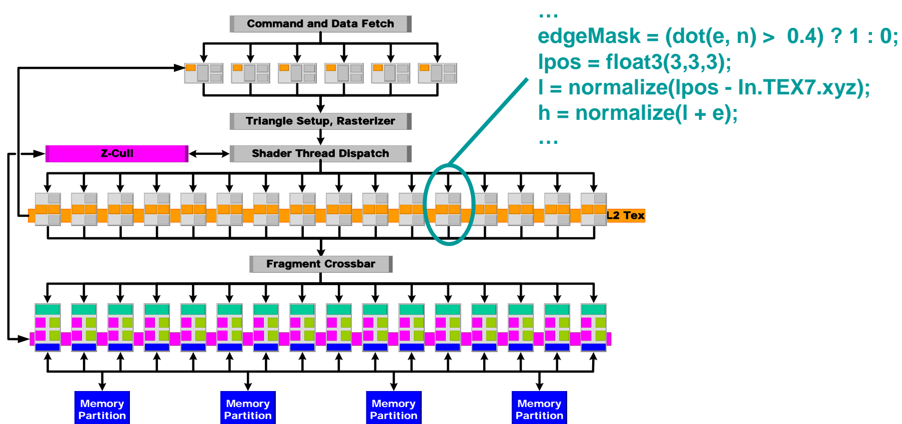
1 million pixels
@ 60 frames/sec:

60 million pixels/sec.

Lots of work for each pixel.

Half Life 2
Valve Software
Nov. 2004

HW is programmable (for some units)




```
…
edgeMask = (dot(e, n) >  0.4) ? 1 : 0;
lpos = float3(3,3,3);
l = normalize(lpos - In.TEX7.xyz);
h = normalize(l + e);
…
```

"Mainstream" architects can learn from GPUs

- Parallelism is becoming more important
  - Single thread vs. FLOPS/$ and FLOPS/Watt
- GPUs are first highly-parallel processors in PCs
  - And now they're programmable
- Games are major driver of PC performance
  - Large market
  - Performance is not yet "good enough"
  - Innovative and talented software developers
    - Willing to experiment
    - Willing to endure (some) pain to get performance

Outline

- Fundamentals of 3D graphics
- Overall architecture of graphics processor
- Details of particular hardware units
- Questions

Please interrupt at any time to ask questions.

Fundamentals of 3D Graphics

Motivation for learning graphics fundamentals

- Q: I'm an architect. I do hardware, not algorithms. Can't we just skip ahead to the architecture?

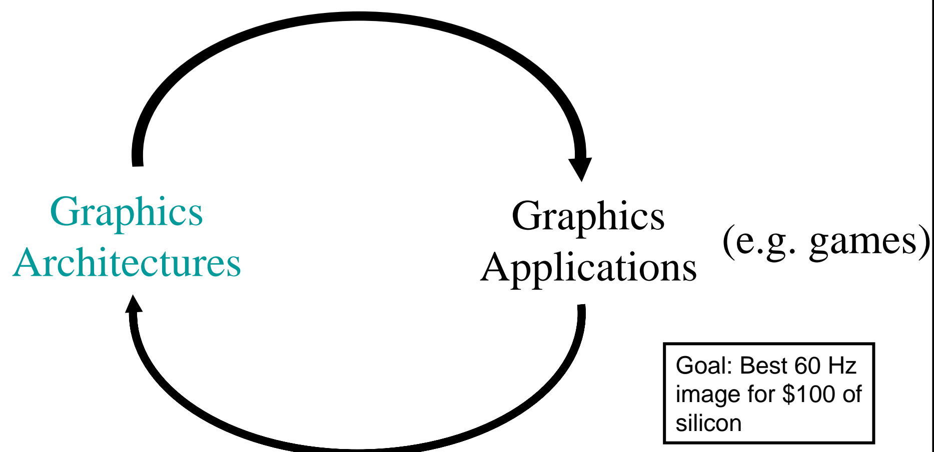
  A: Not really. You can't understand 3D graphics architectures without understanding 3D graphics algorithms.

- Q: Could I design my new Acme FlexiGPU architecture by optimizing for current graphics applications/traces/benchmarks?

  A: No, not if you want your architecture to be relevant when it's done.

Graphics applications and HW co-evolve

Architecture strongly influences applications



Goal: Best 60 Hz image for $100 of silicon

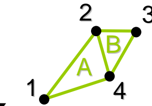
The rendering problem

- Given:
  - 3D world (objects and materials)
  - Light locations
  - A viewpoint

- Compute:
  - 2D image seen from the viewpoint

Geometry is modeled using triangle meshes

Image: Hughes Hoppe, Microsoft Research



Vertex Array
(x1, y1, z1)
(x2, y2, z2)
…

Index Array
V1, V2, V4 – represents Triangle A
V4, V2, V3 – represents Triangle B

Vertices are only stored once.
Triangles point to their vertices.

The Z-buffer algorithm

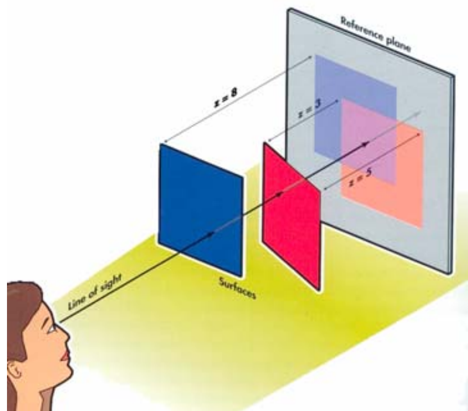
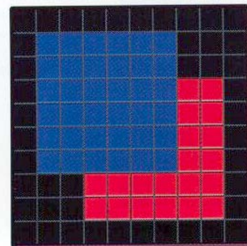


Figure: Prosise, How Computer Graphics Work



Z-Buffer

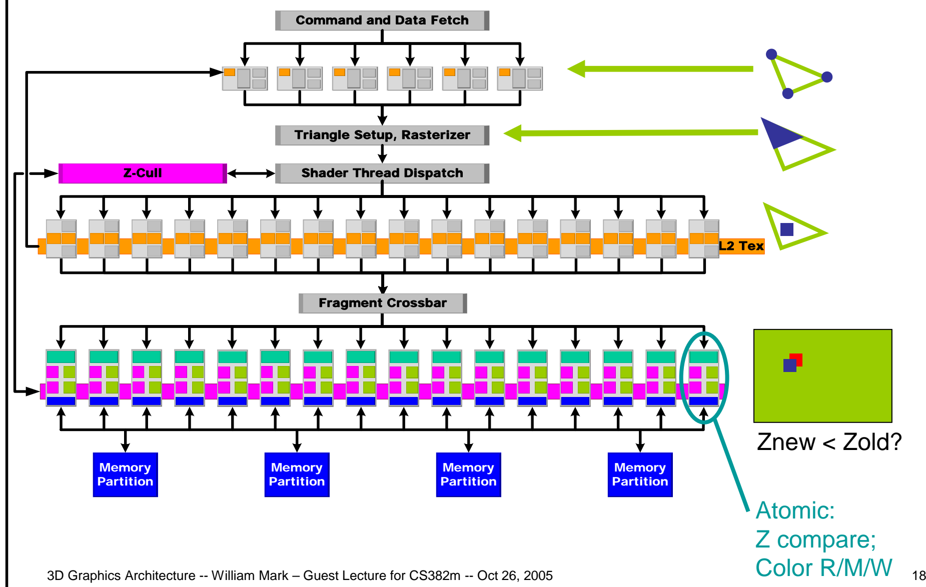
0	0	0	0	0	0	0	0	0	0
0	8	8	8	8	8	8	0	0	0
0	8	8	8	8	8	8	0	0	0
0	8	8	8	8	8	8	5	5	0
0	8	8	8	8	8	8	5	5	0
0	8	8	8	8	8	8	5	5	0
0	8	8	8	8	8	8	5	5	0
0	0	0	3	3	4	4	5	5	0
0	0	0	3	3	4	4	5	5	0
0	0	0	0	0	0	0	0	0	0

Z-buffer algorithm uses “brute force”

for each triangle *A* **do**
  for each pixel in *A* **do**
    Compute depth *z* and shade *s* of *A* at *(i,j)*
    if z > *Z-buffer [i,j]* **then**
      Z-buffer [i,j] ← *z*
      Color-buffer[i,j] ← *s*
    end if
  end for
end for

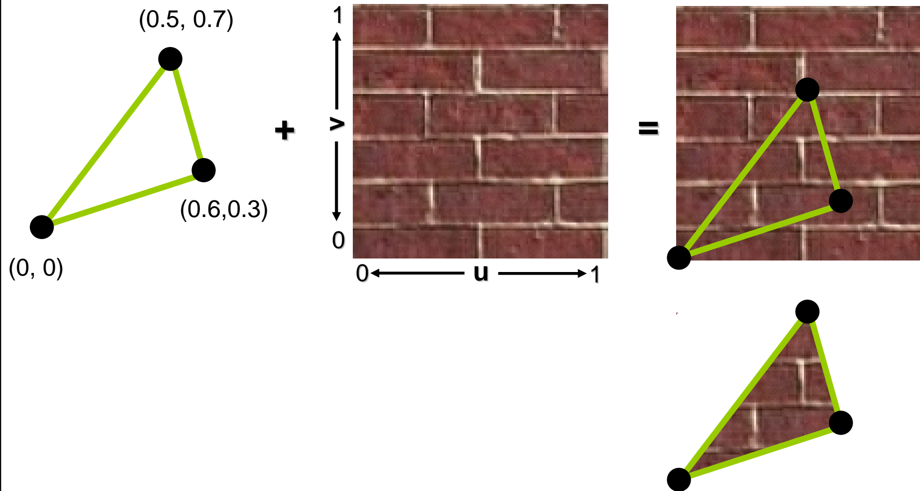
- Touches each triangle exactly once.
- Application can choose triangle order. (and associate meaning with it)
- But: Nearly-random accesses to memory.

Z-buffer algorithm maps to giant pipeline



Texture mapping adds detail to polygons

Use **texture coordinates** to map image to geometry



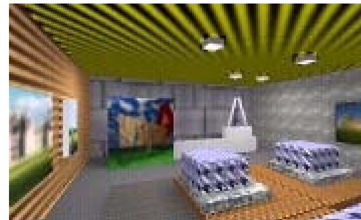
3D Graphics Architecture -- William Mark – Guest Lecture for CS382m -- Oct 26, 2005 19

Avoid texture artifacts with filtering operations

Simple texture sampling

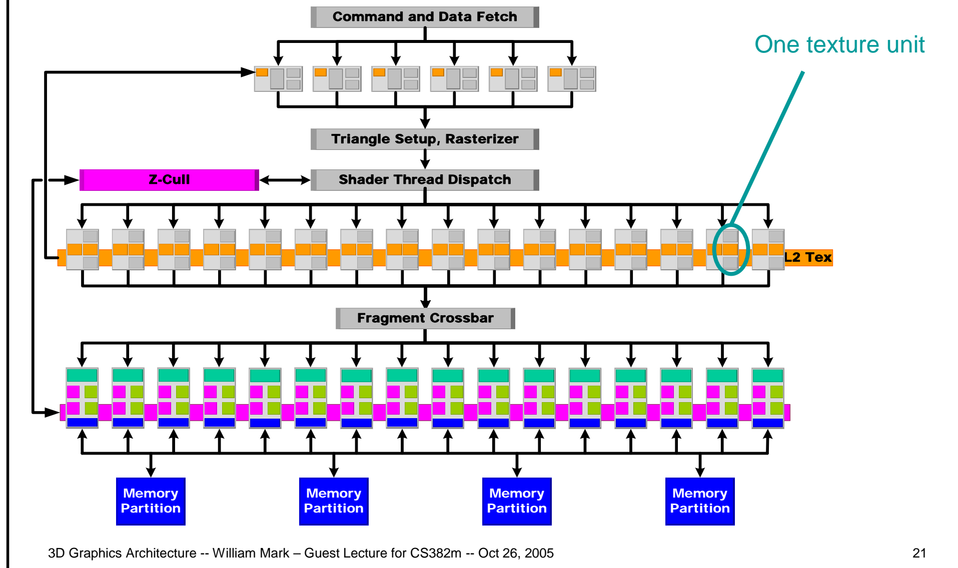
Better: use MIP-mapping
(a form of filtered sampling)

Each pixel:
18 adds, 24 mul, 8 loads!
Use 16-bit arithmetic.
8 pixels/clock!

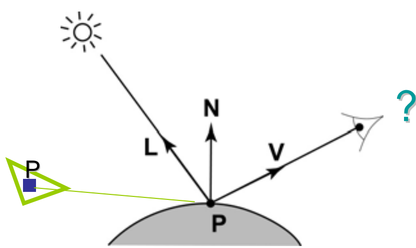


3D Graphics Architecture -- William Mark – Guest Lecture for CS382m -- Oct 26, 2005 20

Where texture HW lives in Z-buffer pipeline



Computing color at a pixel can be complex



P = point on surface

- Compute color of surface
- Consider:
  - Light positions
  - Surface properties
  - Surface texture
  - Etc.

Lighting adds realism to single-color surface

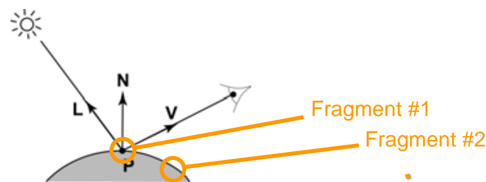
Variety in materials → programmable shaders

- Real world has infinite variety of materials
  - Need programmable shaders to describe them
- Example fragment program in Cg/HLSL:

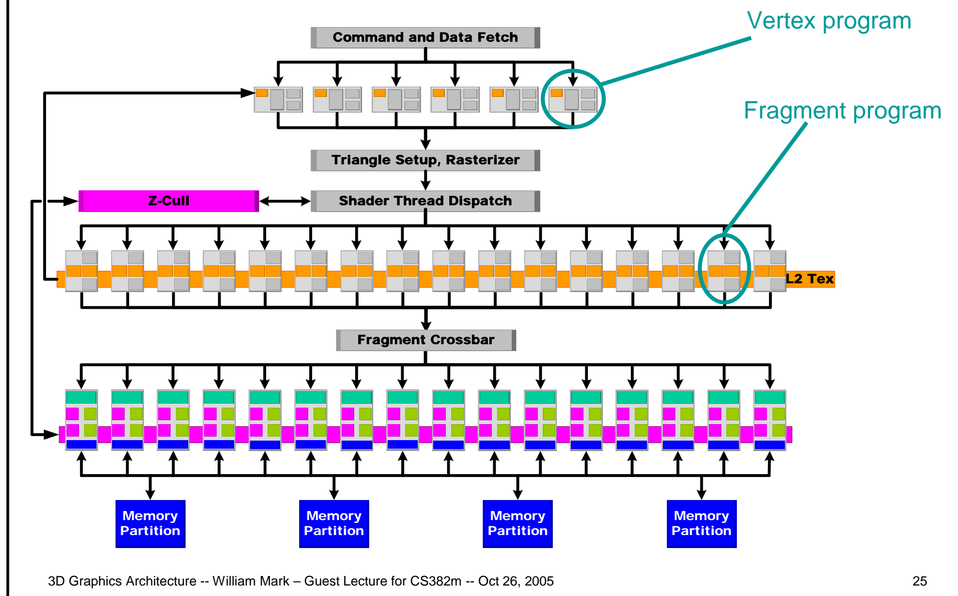
```
void normalmapped(float2 normalMapTexCoord : TEXCOORD0,
                                    …
                  out float4 color : COLOR,
                       uniform float ambient,
                       …)
{
  float3 normalTex, …;
  normalTex = tex2D(normalMap, normalMapTexCoord).xyz;
  …
  diffuse = saturate(dot(normal, normLightDir);
  …
  color = Kd * (ambient + diffuse ) +
          Ks * pow(specular, specularExponent;
}
```


Programming model makes parallelism easy

- Program is re-run for every fragment (or vertex)
- Perfect parallelism:
  - Program cannot communicate with other fragments
  - No persistent state (each fragment is independent)
  - In some respects, a “stream programming” model:
    - Each fragment gets one input record and one output record
    - Fragment program = “stream kernel” or “filter”



Where shader programs execute



Data type (precision) summary

	Old	New
Fragment processor	fixed10-12	float32
Framebuffer color, blend unit	fixed8	fixed8, float16
Textures, texture filter	fixed8	fixed8, float16
Vertex processor (positions)	float32	float32
Rasterizer	Various float	Various float

Increasing precision driven by:
- Programmable shading -- [fragment processor]
- High-dynamic-range rendering and framebuffers -- [texture, framebuffer, blend]
- Global illumination (mostly for future) -- [fragment processor, framebuffer, textures]

Graphics HW supports 4 x FP32 register SIMD

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{new} = \begin{bmatrix} R_{xx} & R_{xy} & R_{xz} & T_x \\ R_{yx} & R_{yy} & R_{yz} & T_y \\ R_{zx} & R_{zy} & R_{zz} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{old}$$

4x4 rotation/translation/etc. matrix

Lots of 4x4 matrix operations, as well as 3-vector math.

ISA support:
- 4-wide SIMD
- Dot product
- Multiply-accumulate

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{final} = \frac{1}{w} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{new}$$

Division is also important, but only need **scalar reciprocal** instruction.

Architecture Discussion & Details

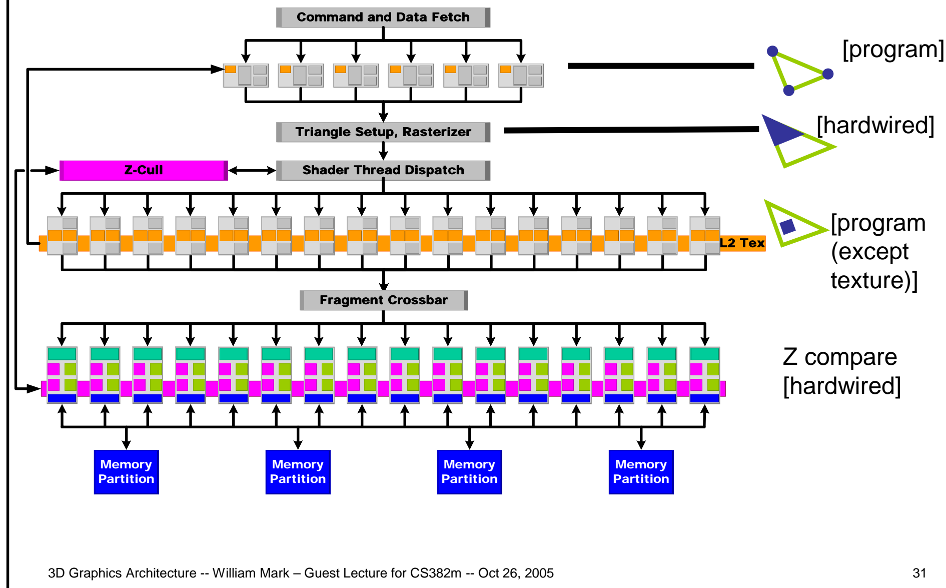
Why is graphics hardware fast?

- Specialization
  - Gradually becoming less important (esp. for FLOPS)
  - But still matters a lot
- Parallelization
  - Rapidly becoming more important
  - Two kinds:
    - Task parallelism – pipeline of operations
    - Data parallelism
  - **Architecture is optimized for throughput, not latency**

Specialization

- Memory system
  - Organize accesses for temporal and spatial locality
    - Interacts strongly with parallel work scheduling
  - Specialized compression of most memory traffic
  - Specialized pre-fetching and caching
- Dedicated HW for hard-to-parallelize operations
  - E.g. rasterization
- Dedicated HW for texture filtering
  - Most computationally intensive task
- Overall data pipeline
  - Esp. enforcement of ordering; culling optimizations

Lots of data parallelism – at *most* stages



[program]

[hardwired]

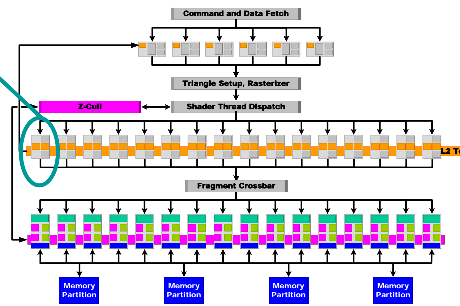
[program (except texture)]

Z compare [hardwired]

What is irrelevant to GPU's being fast?

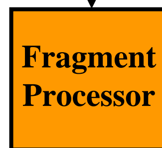
- Aggressive ILP
  - Out-of-order execution
  - Speculative execution
  - Ultra-deep, ultra-high-frequency processor pipelines
  - These techniques do not give good !/$
- Most architectural optimizations for low latency

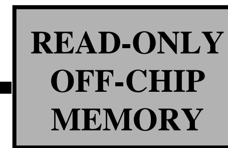
Fragment processor



Fragment programming model: A stream of tasks

Input stream (from rasterizer)

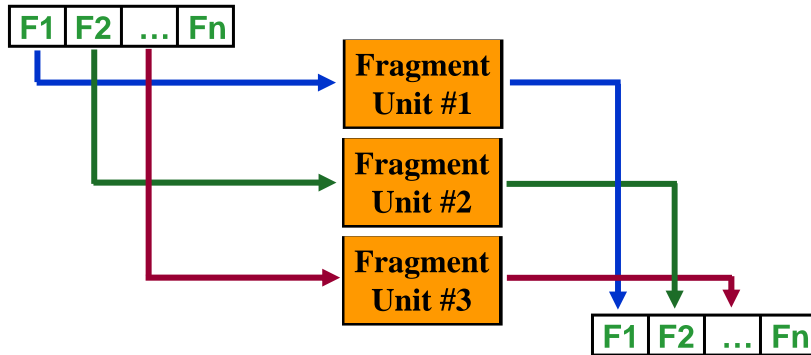




Output stream (to Z compare)

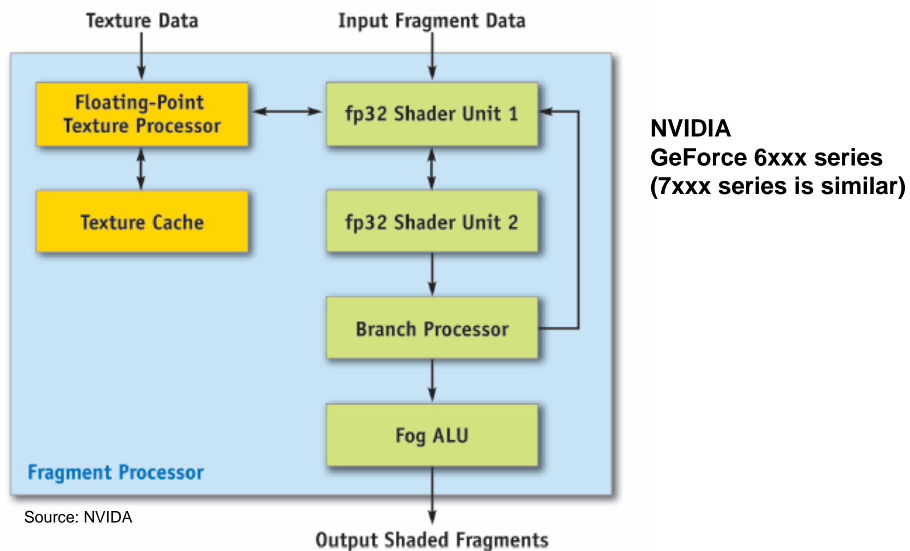
Architecture uses:
- Multiple cores
- Massive multithreading within each core

Stream model supports data parallelism



- Communication between elements is prohibited

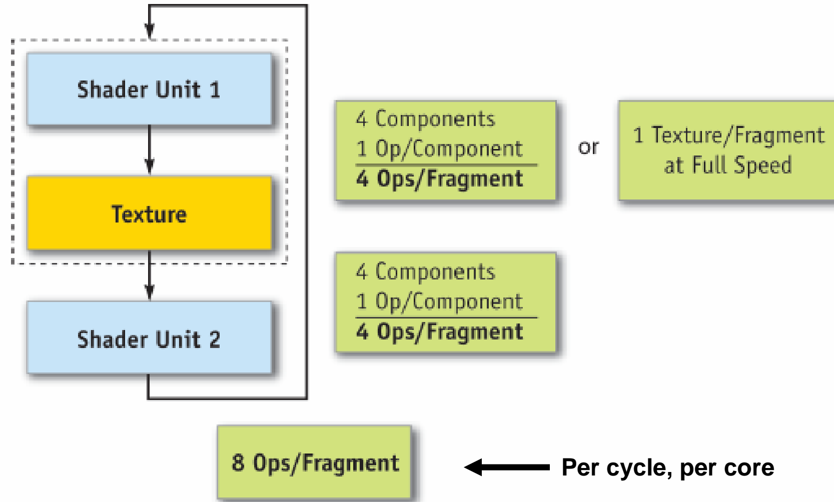
High-level arch of one fragment core



Source: NVIDA

NVIDIA
GeForce 6xxx series
(7xxx series is similar)

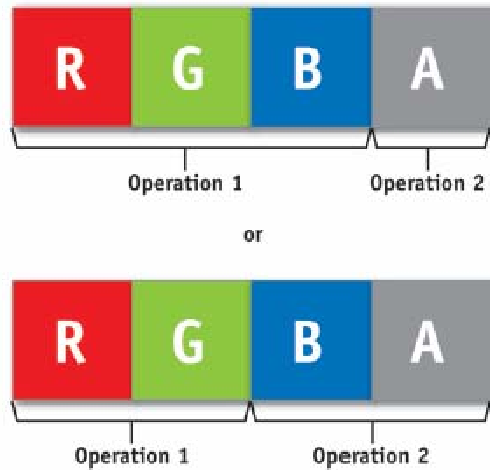
Slightly more detailed and realistic arch



Per cycle, per core

Figure: NVIDIA

Each pipeline stage does mini-VLIW



Result:
High throughput on scalar operations as well as 4-wide SIMD

Figure: NVIDIA

More architectural details

Massive multithreading can hide memory latency

(a.k.a. "Never stall on a cache miss")

- Consider texture mapping:

```
for each fragment {
    compute_texture_addresses();
    texels = memory_read(texaddress1, 2, 3, 4, 5, 6, 7, 8);
    compute_color(texels);
}
```

- Each fragment is a thread
- Context switch on texture fetch
  - Must hide memory latency – cache miss rate > 10%
  - Trend is towards dynamic scheduling
- Need 64+ live fragments per fragment processor!
  - Fortunately, thread context is small (<< 100 bytes, typ.)

Multithreaded prefetch also used for framebuffer

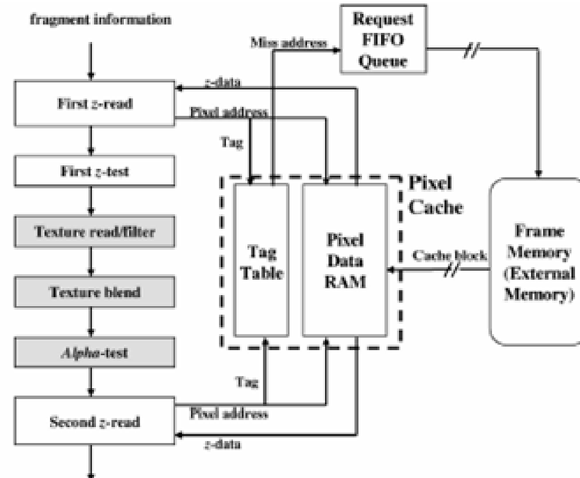


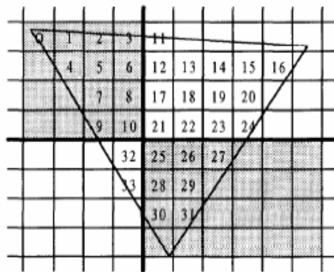
Figure: W. Park et al,
An Effective Pixel Rasterization Pipeline Architecture for 3D Rendering Processors, 2003
Also, see NVIDIA and ATI patents, e.g. US #6,734,861, filed Oct 2000.

Framebuffer has R/W hazards

- Semantics say:
  - – Preserve ordering
  - – Atomic R/M/W for Z compare
- In practice:
  - – Semantics only matter for two fragments at same pixel
  - – Detect this and special case it
    - Conceptually, 1 million locks (one for each pixel)
    - Hash instead!
- All of this is hardwired
  - – Needs very high throughput
  - – One of the big “tar pits” for general purpose hardware

Maximize cache hit rates with 2D tiling

- Framebuffer and textures organized into tiles
  - Allows capture of 2D spatial coherence by caches
- Rasterizer generates fragments in tile order
- All of this is hardwired



Reference:
McCormack et al,
Neon: a single-chip 3D workstation graphics accelerator, 1998

Other important optimizations

- Early fragment kill
  - Perform Z and/or stencil test before shading, texturing
  - Be careful, since semantically it occurs afterward
- Hierarchical (low-res) Z/stencil buffers
  - Keep low-res buffers on-chip
  - Improves performance of early-discard tests
  - Annoying interactions with other features
    - E.g. Turn this stuff off if fragment shader can modify Z

Miscellaneous

Yield tricks

- Top-of-the-line HW has 16 fragment units
  - But it's quite hard to find these parts
- Almost-top HW has 12 fragment units
  - Much easier to find these parts
- Why might that be?

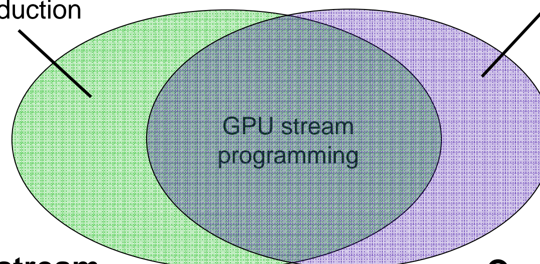
General purpose computation on GPUs

- Use fragment processors as stream processors
- Specialized languages for this purpose
  - Brook for GPU's [Buck et al., 2003]
  - Sh [McCool et al, 2004]
- Applications include:
  - Image processing
  - Some BLAS routines (single-precision only)
  - Ray casting
- Hype exceeds reality
  - But reality is slowly catching up

Full stream proc ≢ GPU

- Scatter to memory
- Conditional kernel outputs
- Efficient reduction
- etc.

- Tagged caches
- R/M/W blend,Z
- etc.



Full stream programming
(e.g. Imagine processor)

Current GPU's
(efficient Z-buffer rendering, with programmable shading)

Historical trends

Year	NVIDIA Product	Mtri/ sec	Mfrag/ sec (*)	BW GB/sec	Clk MHz	Trnst cnt (M)	Proc (um)
1998	Riva ZX	3	100	1.6	100	4	.35
1999	Riva TNT2	9	350	3.2	175	9	.22
2000	GeForce2 GTS	25	664	5.3	166	25	.18
2001	GeForce3	30	800	7.4	200	57	.18
2002	GeForce4 Ti 4600	60	1200	10.4	300	63	.15
2003	GeForce FX	167	2000	16.0	500	121	.13
2004	GeForce 6800 Ultra	170	6800	35.2	425	222	.13

* Fragment fill rate for 1 texture.

Source: Mark Kilgard, NVIDIA

- Yearly growth rates well above CPU rate of ~1.5
  - While adding substantial new functionality!
- But growth rates for BW & die area probably unsustainable

Recap

Why is graphics hardware fast?

- Specialization
  - Serial bottlenecks such as rasterization
  - Memory access, caching, compression, addressing
  - Ordering of parallel memory writes
  - Shepherding of parallelism, data flows, communication
  - Smart work avoidance: early Z tests, etc.
  - Texture filtering
- Parallelism
  - Multithreaded vertex processor
  - Multithreaded fragment/texture processor
  - “Multithreaded” ROP unit (Z test, etc)

Advantages of Z-buffer algorithm

- Reasonable computational cost
- Each polygon touched just once
  - Application can feed polygons in any order.
  - Works well for moving objects.
- Producer-consumer locality within HW pipeline
- Good spatial locality of memory accesses
  - Texture
  - Framebuffer
- Most parts of algorithm easily parallelized

The “tar pits” for conventional architects

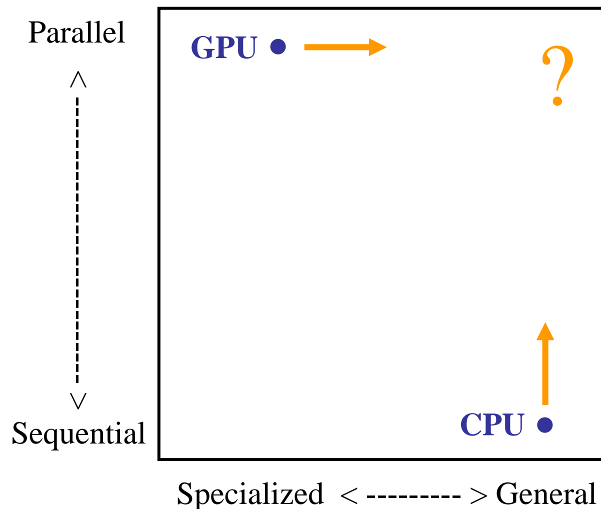
- GPUs must optimize for throughput, not latency
  - We see this trend emerging elsewhere, too.
- 3D graphics computations != signal processing
  - Surprisingly irregular and complicated
  - Especially with optimizations like compression
- Functionality changes rapidly
  - Big mistake to design for old benchmarks
  - Challenging for academics to keep up
- Specialized HW still critical to Z-buffer performance
- Architects must understand the application
  - Perhaps generally true for parallel systems?

The Future

Near term (next two years)

- Additional pipeline stages become programmable
  - E.g. Geometry subdivision/tesselation
- Additional flexibility in data flow, communication
  - Easier to implement innovative graphics algorithms
  - Easier to use GPU as “general purpose” parallel processor.
- First real successes for using GPU as “general purpose” processor
  - But limitations of stream programming model will also become apparent.

Longer term



Forces driving long-term evolution

- Desire to accelerate other computations
  - Collision detection and response, AI, etc.
  - Scene management
- Desire for more realistic images
  - Better shadows, indirect illumination, antialiasing, etc.
  - Z buffer has trouble with needed visibility computations
  - Possibilities include:
    - Enhancements to Z buffer
    - Raycasting visibility algorithms
- Work smarter, not harder
  - Trend away from brute-force, one-size-fits-all algorithms

Long-term predictions

- Graphics algorithms continue to evolve rapidly
  - End of Z-buffer as we know it
- Graphics is major driver of single-chip parallelism
  - Return to “software rendering”
  - Two parallel programming models: Streams and CSP
- One chip combines “CPU” and “GPU”
  - Fine grained throughput-optimized cores
  - Coarse grained latency-optimized cores
  - Specialized HW for certain tasks
  - Who makes it?
  - What are details of its architecture?

Game consoles as innovation platform

- Clean slate design
  - Minimal need for backward compatibility
- One company controls entire system design
  - Graphics processor
  - CPU
  - APIs and programming languages
  - Operating system
  - Application software
- But economics still discourage radical designs

Open research questions

- How should real-time graphics algorithms and architectures co-evolve?
  - What new/enhanced algorithms? What HW?
- Specialized vs. General HW?
  - What is the right balance?
  - Is semi-specialized HW useful? (e.g. R/M/W)
- What programming model for parallel units?
  - Stream, CSP, both, other?
- What granularity of parallel units?
  - Lots of little ones vs. a few big ones vs. hybrids
- Can HW for graphics also accelerate other apps?

If you only remember one thing…

Thread-level parallelism will be the most important technique for achieving performance goals in future commodity computer architectures.

But exploiting it requires more interaction between application and architecture than we're accustomed to.

The End

Questions?