EE382V: Principles in Computer Architecture
Parallelism and Locality
Fall 2008

Lecture 11 – The Graphics Processing Unit (II)

Mattan Erez

UT ECE

The University of Texas at Austin

Outline

- 3D graphics pipeline
- Programmable graphics pipeline
- General GPU architecture
  - And high-level examples
- Some slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - From The University of Illinois ECE 498AI class
- Some slides courtesy Massimiliano Fatica (NVIDIA)
- Many slides courtesy Kayvon Fatahalian (Stanford)

A GPU Renders 3D Scenes

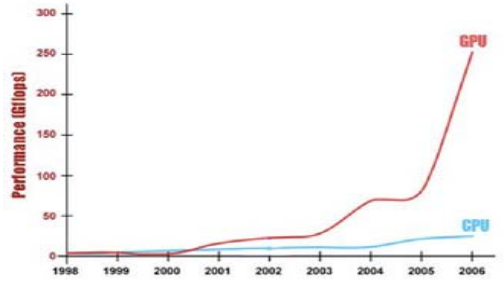
- A ***Graphics Processing Unit (GPU)*** accelerates rendering of 3D scenes
  - Input: description of scene
  - Output: colored pixels to be displayed on a screen
- Input:
  - Geometry (triangles), colors, lights, effects, textures
- Output:

GPU Scene Complexity Defined by Standard Interfaces (DirectX and OpenGL)

- DirectX and OpenGL define the interface between applications and the GPU
- **Geometry** describes the objects and layout
  - Triangles (vertices) describe all objects
    - Can have millions of triangles per scene
  - Can modify triangle surfaces
    - Bumps, ripples, …
  - Lights are part of the scene geometry
- **Pixel Shaders** describe how to add color
  - Colors of triangle vertices
  - Textures (patterns)
  - How to determine color of pixels within a triangle
  - …

Complexity and Quality are Orders of Magnitude Better

GPU Performance is Increasing Much Faster than CPUs



GPU and CPU Architectures are Starting to Converge

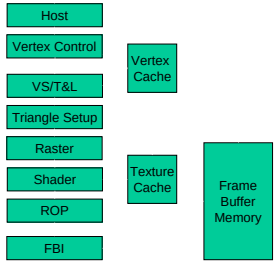
	CPUs	GPUs
1997	no explicit parallelism	not programmable
2000	explicit short vectors	emerging programmability (2001 – 2002), "infinite" DP
2003	explicit short vectors explicit threading (~2)	fully programmable explicit "infinite" DP no scatter
2006	explicit short vectors explicit threading (~4)	explicit vectors explicit threading (~16)
2009?	explicit vectors explicit threading (>16)	explicit vectors explicit threading (>16)

Outline

- What is a GPU?
- Why should we care about GPUs?
- 3D graphics pipeline
- Programmable GPUs

- Many slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - From The University of Illinois ECE 498AL class
- Other slides courtesy Massimiliano Fatica (NVIDIA)

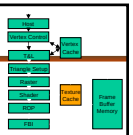
The NVIDIA GeForce Graphics Pipeline



Filtering techniques

- Point sampling:
  - pixel values are calculated by choosing one texture pixel (texel) color
- Bilinear filtering:
  - interpolating colors from 4 neighboring texels. This gives a smoothing (if somewhat blurry) effect and makes the scene look more natural and prevents abrupt transitions between neighboring texels.
- Trilinear filtering:
  - interpolating bilinearly filtered samples from two mip-maps. Trilinear mip-mapping prevents moving objects from displaying a distracting "sparkle" caused by abrupt transitions between mipmaps.
- Anisotropic filtering:
  - interpolating and filtering multiple samples from one or more mip-maps to better approximate very distorted textures. Gives a sharper effect when severe perspective correction is used. Trilinear mipmapping blurs textures more.

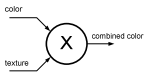
Texture Cache



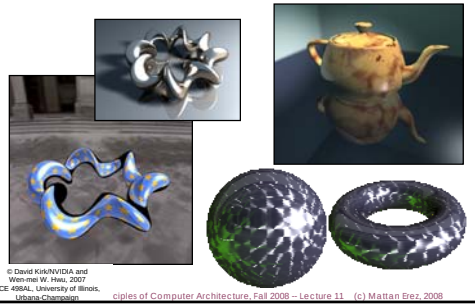
- Stores temporally local texel values to reduce bandwidth requirements
- Due to nature of texture filtering high degrees of efficiency are possible
- Efficient texture caches can achieve 75% or better hit rates
- Reduces texture (memory) bandwidth by a factor of four for bilinear filtering

Pixel Shading

- 1999 (DirectX 7)
  - Application could select from a few simple combinations of texture and interpolated color
    - Add
    - Decal
    - Modulate
- Next (DirectX 9)
  - Write a general program that executes for every pixel with a nearly unlimited number of interpolated inputs, texture lookups and math operations
  - Can afford to perform sophisticated lighting calculations at every pixel

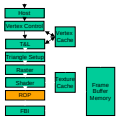


GeForce FX Fragment/Pixel Program Examples



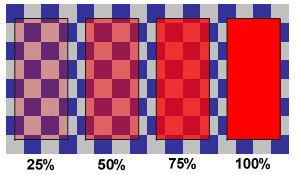
ROP (from Raster Operations)

- C-ROP performs frame buffer blending
  - Combinations of colors and transparency
  - Antialiasing
  - Read/Modify/Write the Color Buffer
- Z-ROP performs the Z operations
  - Determine the visible pixels
  - Discard the occluded pixels
  - Read/Modify/Write the Z-Buffer
- ROP on GeForce also performs
  - "Coalescing" of transactions
  - Z-Buffer compression/decompression



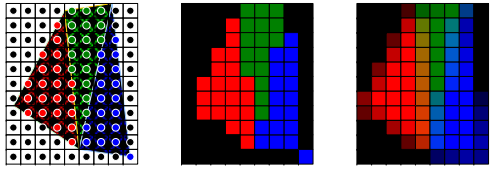
Alpha Blending

- *Alpha Blending* is used to render translucent objects.
- The pixel's alpha component contains its *opacity*.
- Read-modify-write operation to the color framebuffer
- Result = alpha * Src + (1-alpha) * Dst

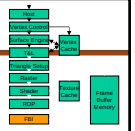


Anti-Aliasing

- Aliased rendering: color sample at pixel center is the color of the whole pixel
- Anti-aliasing accounts for the contribution of all the primitives that intersect the pixel

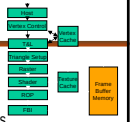


Frame Buffer Interface (FBI)



- Manages reading from and writing to frame buffer
- Perhaps the most performance-critical component of a GPU
- GeForce's FBI is a crossbar
- Independent memory controllers for 4+ independent memory banks for more efficient access to frame buffer

The Frame Buffer



- The primary determinant of graphics performance other than the GPU
- The most expensive component of a graphics product other than the GPU
- Memory bandwidth is the key
- Frame buffer size also determines
  - Local texture storage
  - Maximum resolutions
  - AA resolution limits

Z Buffer

- A Z buffer is a 2-D array of Z values with the same (x,y) dimensions as the color framebuffer
- Every candidate pixel from the shader has a calculated Z value along with its R,G,B,A color.
- Before writing the color, perform the Z-buffer test:
  - Read the Z value from memory
  - Compare the candidate Z to the Z from memory; if the candidate Z is NOT in front of the previous Z, discard the pixel
  - Otherwise, write the new Z value to the Z buffer and write (or blend) the new color to the color framebuffer

Summary, so far…

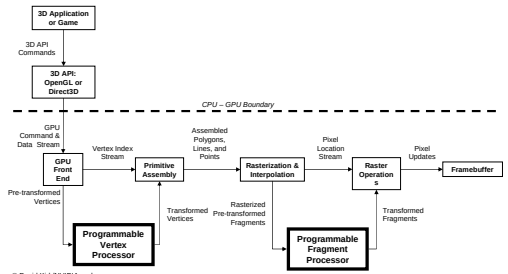
- Introduction to several key 3D graphics concepts:
  - Framebuffers
  - Object Representation
  - Vertex Processing
  - Lighting
  - Rasterization
  - Gouraud Interpolation
  - Texture Mapping
  - Pixel Shading
  - Alpha Blending
  - Anti-Aliasing
  - Z-Buffering

Outline

- What is a GPU?
- Why should we care about GPUs?
- 3D graphics pipeline
- Programmable GPUs

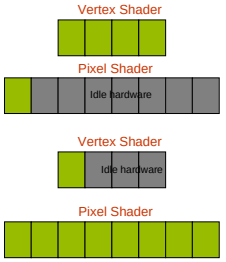
- Many slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - From The University of Illinois ECE 498AI class
- Other slides courtesy Massimiliano Fatica (NVIDIA)

Adding Programmability to the Graphics Pipeline



Vertex and Fragment Processing Share Unified Processing Elements

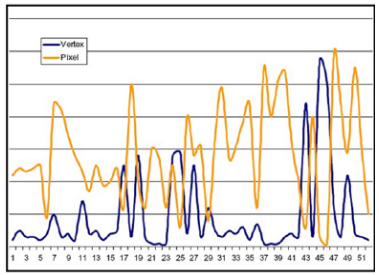
- Load balancing HW is a problem



Heavy Geometry
Workload Perf = 4

Heavy Pixel
Workload Perf = 8

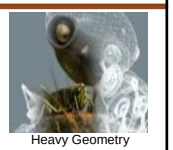
Vertex and Fragment Processing Share Unified Processing Elements



Vertex and Fragment Processing Share Unified Processing Elements

- Load balancing SW is easier

Unified Shader



Heavy Geometry Workload Perf = 11

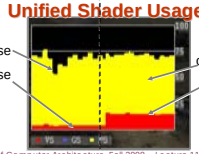
Unified Shader

Heavy Pixel Workload Perf = 11

Vertex and Fragment Processing is Dynamically Load Balanced

Less Geometry — More Geometry

Unified Shader Usage

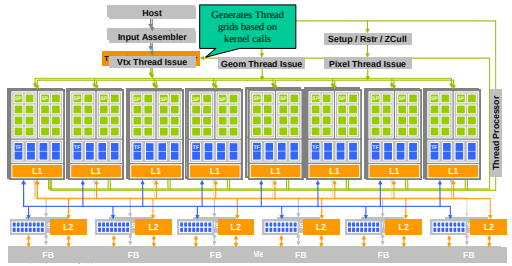


High pixel shader use
Low vertex shader use

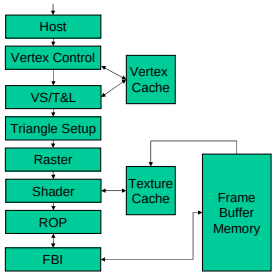
Balanced use of pixel shader and vertex shader

Make the Compute Core The Focus of the Architecture

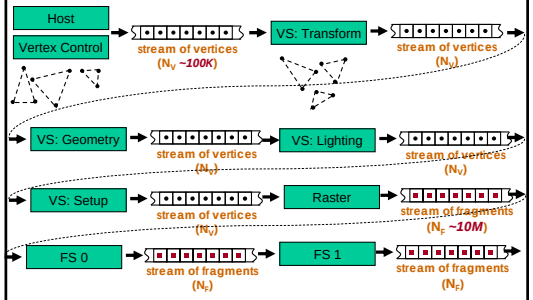
- The future of GPUs is programmable processing
- So – build the architecture around the processor
- Processors execute computing threads
- Alternative operating mode specifically for computing



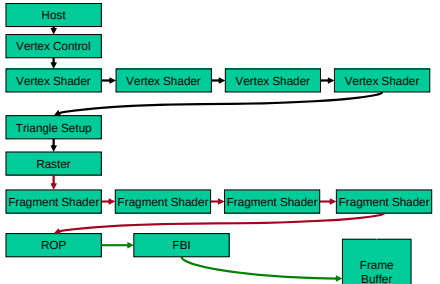
The NVIDIA GeForce Graphics Pipeline



Another View of the 3D Graphics Pipeline



The NVIDIA GeForce Graphics Pipeline



Stream Execution Model

- Data parallel ***streams*** of data
- Processing ***kernels***
  - Unit of Execution is processing of one stream element in one kernel – defined as a ***thread***

Stream Execution Model

- Can partition the streams into chunks
  - Streams are very long and elements are independent
  - Chunks are called ***strips*** or ***blocks***

- Unit of Execution is processing one block of data by one kernel – defined as a ***thread block***

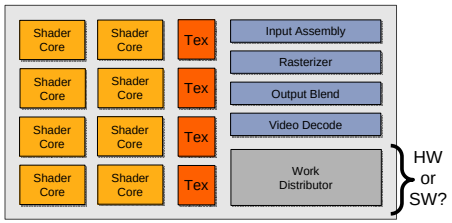
From Shader Code to a Teraflop: How Shader Cores Work

Kayvon Fatahalian
Stanford University

This talk

- Three key concepts behind how modern architectures run "shader" code
- Knowing these concepts will help you:
  1. Understand space of GPU shader core (and throughput CPU processing core) designs
  2. Optimize shaders/compute kernels
  3. Establish intuition: what workloads might benefit from the design of these architectures?

What's in a GPU?



Heterogeneous chip multi-processor (highly tuned for graphics)

A diffuse reflectance shader

```
sampler mySamp;
Texture2D<float3>myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
kd = myTex.Sample(mySamp, uv);
kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

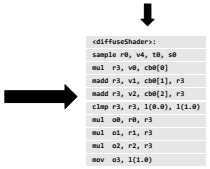
Independent – data parallelism

Compile shader

1 unshaded fragment input record

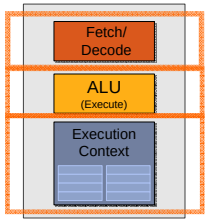
```
sampler mySamp;
Texture2D<float3>myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
kd = myTex.Sample(mySamp, uv);
kd *= clamp ( dot(lightDir, norm), 0.0, 1.0 );
  return float4(kd, 1.0);
}
```

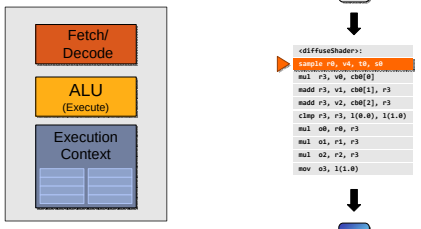


1 shaded fragment output record

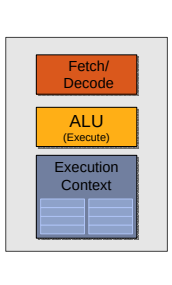
Execute shader

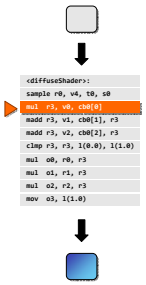


Execute shader

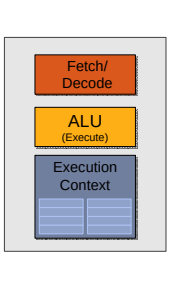


Execute shader

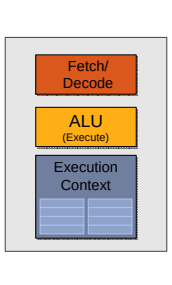


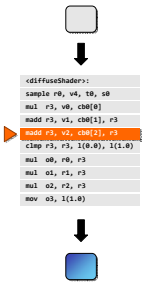


Execute shader

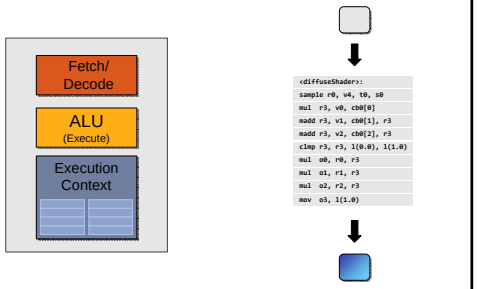


Execute shader

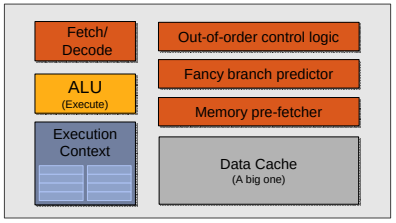




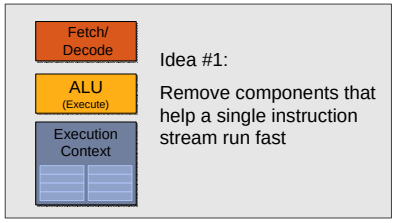
Execute shader



CPU-"style" cores



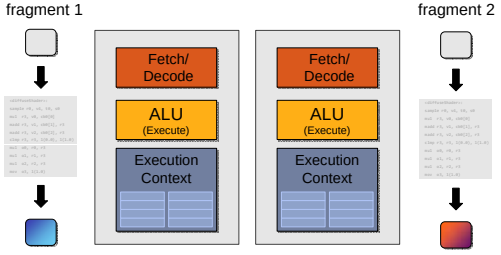
Slimming down



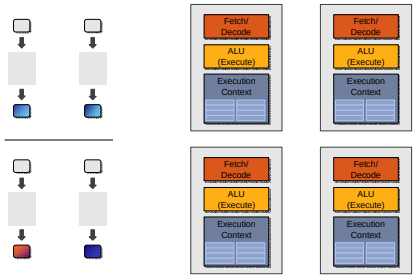
Idea #1:

Remove components that help a single instruction stream run fast

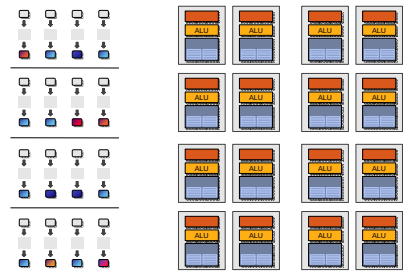
Two cores (two fragments in parallel)



Four cores (four fragments in parallel)

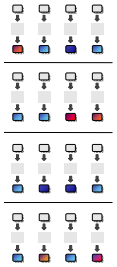


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

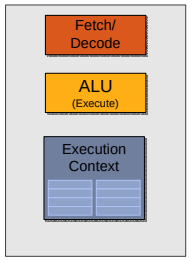
Instruction stream coherence



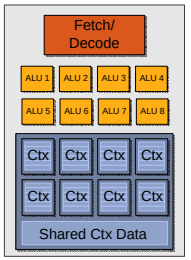
But… many fragments should be able to share an instruction stream!

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```


Recall: simple processing core



Add ALUs

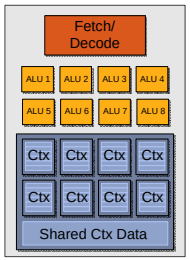


Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Modifying the shader

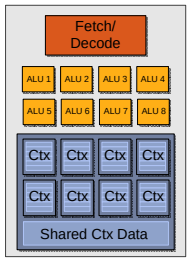



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using scalar ops on scalar registers

Modifying the shader

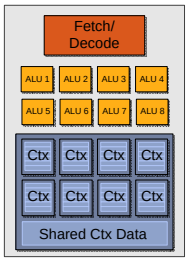


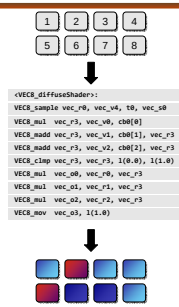

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
```

New compiled shader:

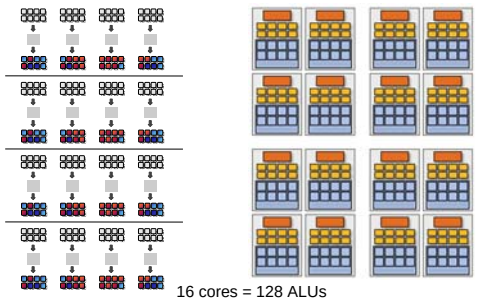
Processes 8 fragments using vector ops on vector registers

Modifying the shader



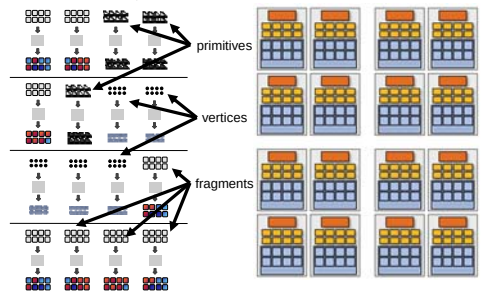


128 fragments in parallel

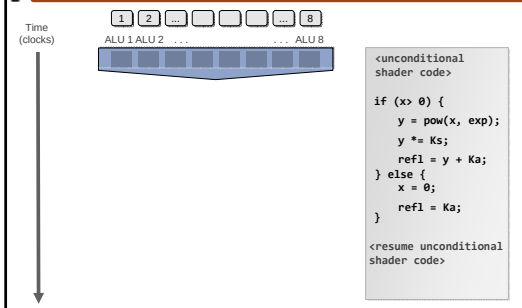


16 cores = 128 ALUs
= 16 simultaneous instruction streams

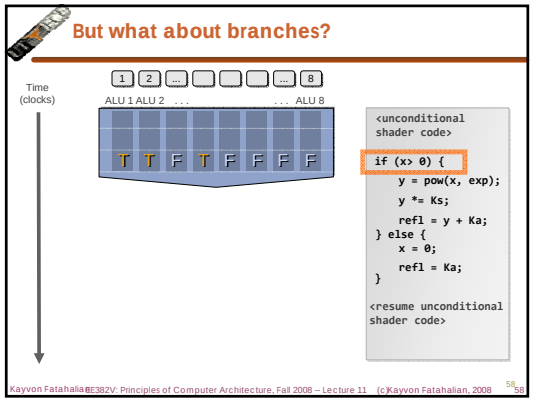
128 [vertices / fragments / primitives / CUDA threads / OpenCL work items / compute shader threads] in parallel



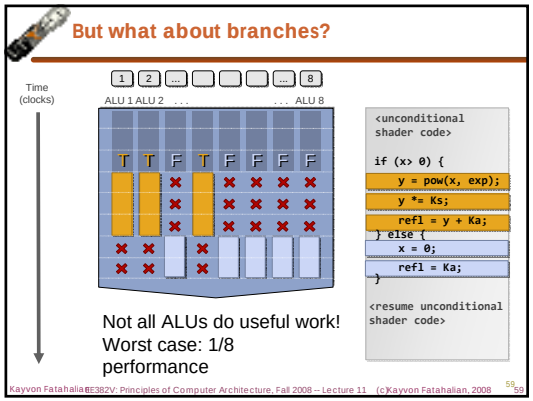
But what about branches?



But what about branches?

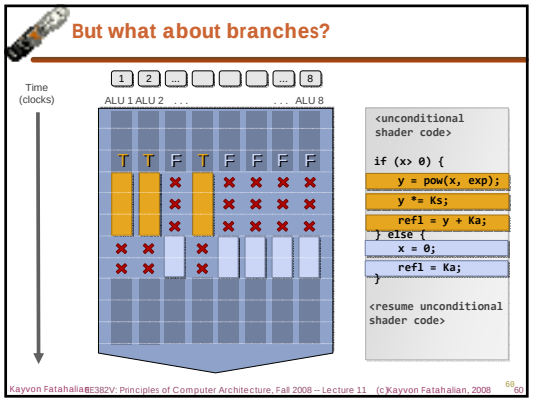


But what about branches?



Not all ALUs do useful work!
Worst case: 1/8 performance

But what about branches?



Clarification

SIMD processing does not imply SIMD instructions

- Option 1: Explicit vector instructions
  - Intel/AMD x86 SSE, Intel Larrabee
- Option 2: Scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps), ATIRadeon architectures

In practice: 16 to 64 fragments share an instruction stream

Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

Texture access latency = 100's to 1000's of cycles

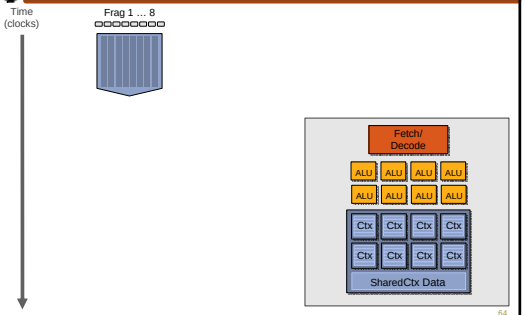
We've removed the fancy caches and logic that helps avoid stalls.

But we have LOTS of independent fragments.

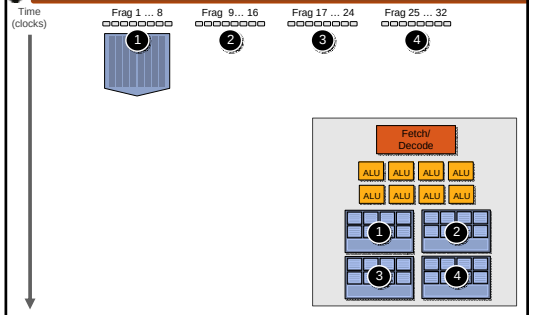
Idea #3:

Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

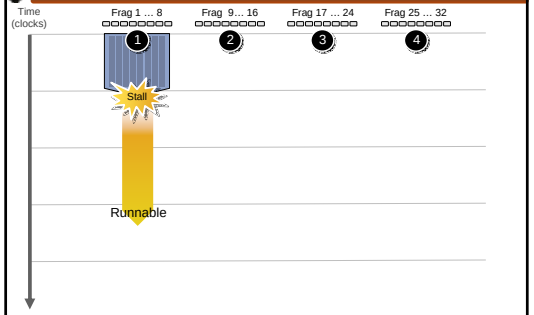
Hiding shader stalls



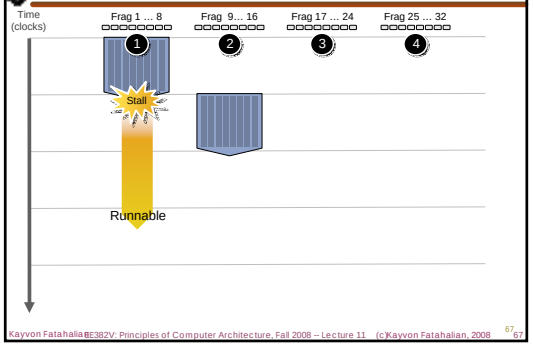
Hiding shader stalls



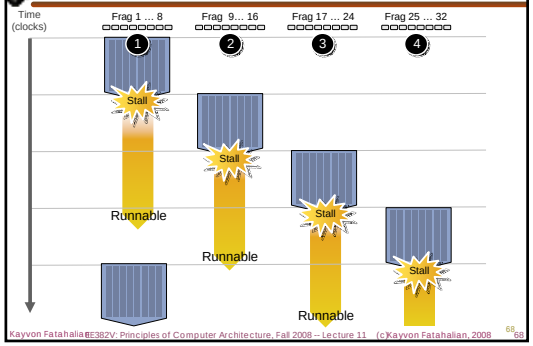
Hiding shader stalls



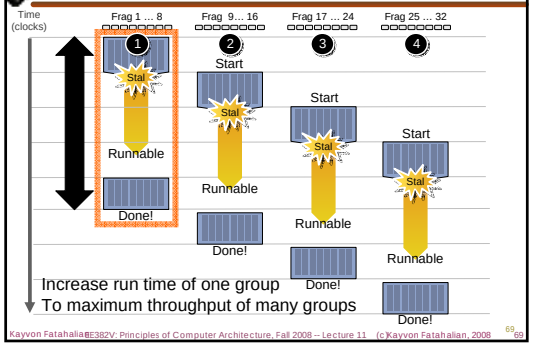
Hiding shader stalls



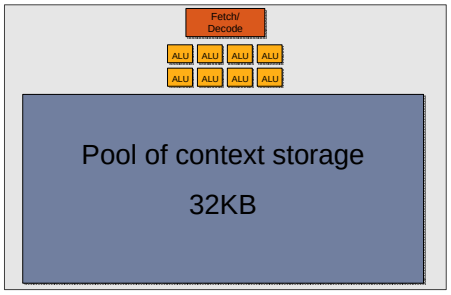
Hiding shader stalls



Throughput!

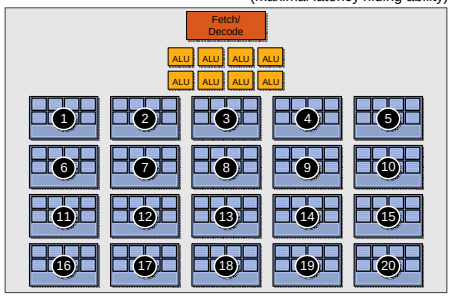


Storing contexts

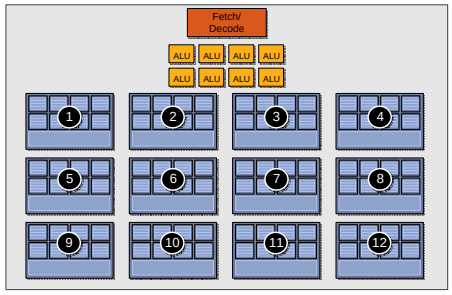


Twenty small contexts

(maximal latency hiding ability)

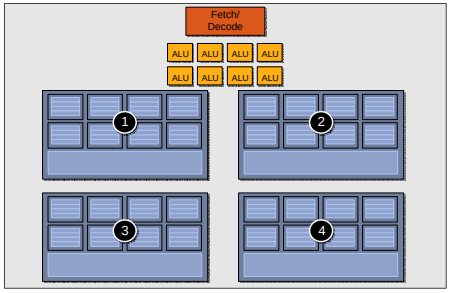


Twelve medium contexts



Four large contexts

(low latency hiding ability)



Summary: three key ideas

1. Use many "slimmed down cores" to run in parallel
2. Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware
3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group