King Abdullah University of Science and Technology

KAUST

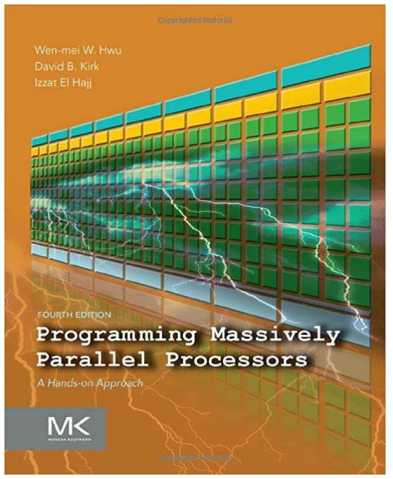
CS 380 - GPU and GPGPU Programming
Lecture 5: GPU Architecture, Pt. 3

Markus Hadwiger, KAUST

Reading Assignment #3 (until Sep 22)

Read (required):

- Programming Massively Parallel Processors book, 4th edition, **Chapter 4** (*Compute architecture and scheduling*)



- NVIDIA CUDA C++ Programming Guide
  (current: v13.0.1, Sep 2, 2025)

`https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

*Read **Chapter 5.6*** (Compute Capability);
*"Read" **Chapter 20.1** and **20.2*** (Compute Capabilities);
*Browse all of **Chapter 20*** (Compute Capabilities)
*Browse all of **Chapter 8.2*** (Maximize Utilization) and
Chapter 8.4 (Maximize Instruction Throughput)

CUDA C++ Programming Guide Chapter 8.4 now (since CUDA 13) actually refers to:
NVIDIA CUDA C++ Best Practices Guide, ***Chapter 12*** (Instruction Optimization)

`https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`

NVIDIA Architectures (since first CUDA GPU)

Tesla [CC 1.x]: 2007-2009

- G80, G9x: 2007 (Geforce 8800, ...)
  GT200: 2008/2009 (GTX 280, ...)

Fermi [CC 2.x]: 2010 (2011, 2012, 2013, …)

- GF100, ... (GTX 480, ...)
  GF104, ... (GTX 460, ...)
  GF110, ... (GTX 580, ...)

Kepler [CC 3.x]: 2012 (2013, 2014, 2016, …)

- GK104, ... (GTX 680, ...)
  GK110, ... (GTX 780, GTX Titan, ...)

Maxwell [CC 5.x]: 2015

- GM107, ... (GTX 750Ti, ...); [Nintendo Switch]
  GM204, ... (GTX 980, Titan X, ...)

Pascal [CC 6.x]: 2016 (2017, 2018, 2021, 2022, …)

- GP100 (Tesla P100, ...)
- GP10x: x=2,4,6,7,8, ...
  (GTX 1060, 1070, 1080, Titan X *Pascal*, Titan Xp, ...)

Volta [CC 7.0, 7.2]: 2017/2018

- GV100, ...
  (Tesla V100, Titan V, Quadro GV100, ...)

Turing [CC 7.5]: 2018/2019

- TU102, TU104, TU106, TU116, TU117, ...
  (Titan RTX, RTX 2070, 2080 (Ti), GTX 1650, 1660, ...)

Ampere [CC 8.0, 8.6, 8.7, 8.8]: 2020

- GA100, GA102, GA104, GA106, ...; [Nintendo Switch 2]
  (A100, RTX 3070, 3080, 3090 (Ti), RTX A6000, ...)

Hopper [CC 9.0], **Ada Lovelace** [CC 8.9]: 2022/23

- GH100, AD102, AD103, AD104, AD106, AD107, ...
  (H100, L40, RTX 4080 (12/16 GB), RTX 4090,
  RTX 6000 (Ada), ...)

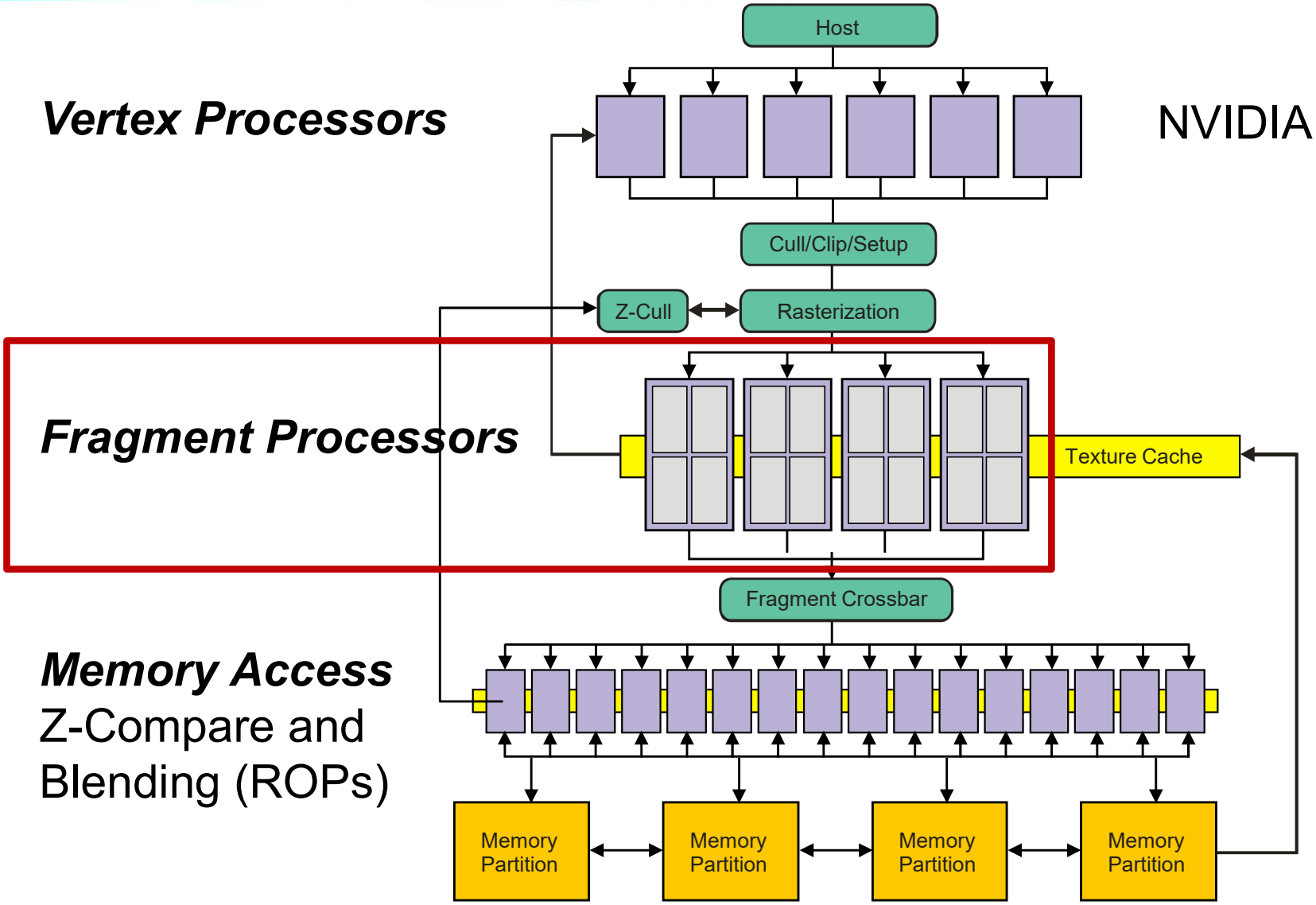
Blackwell [CC 10.0, 10.1(11.0), 10.3, 12.0, 12.1]: 2024/2025

- GB100/102, GB200, GB202/203/205/206/207, ...
  (RTX 5080/5090, GB200 NVL72, HGX B100/200,
  RTX 6000 PRO Blackwell, ...)

see `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`
and `https://en.wikipedia.org/wiki/CUDA`

GPU Architecture
Fast Forward to Today

GPU Structure Before Unified Shaders

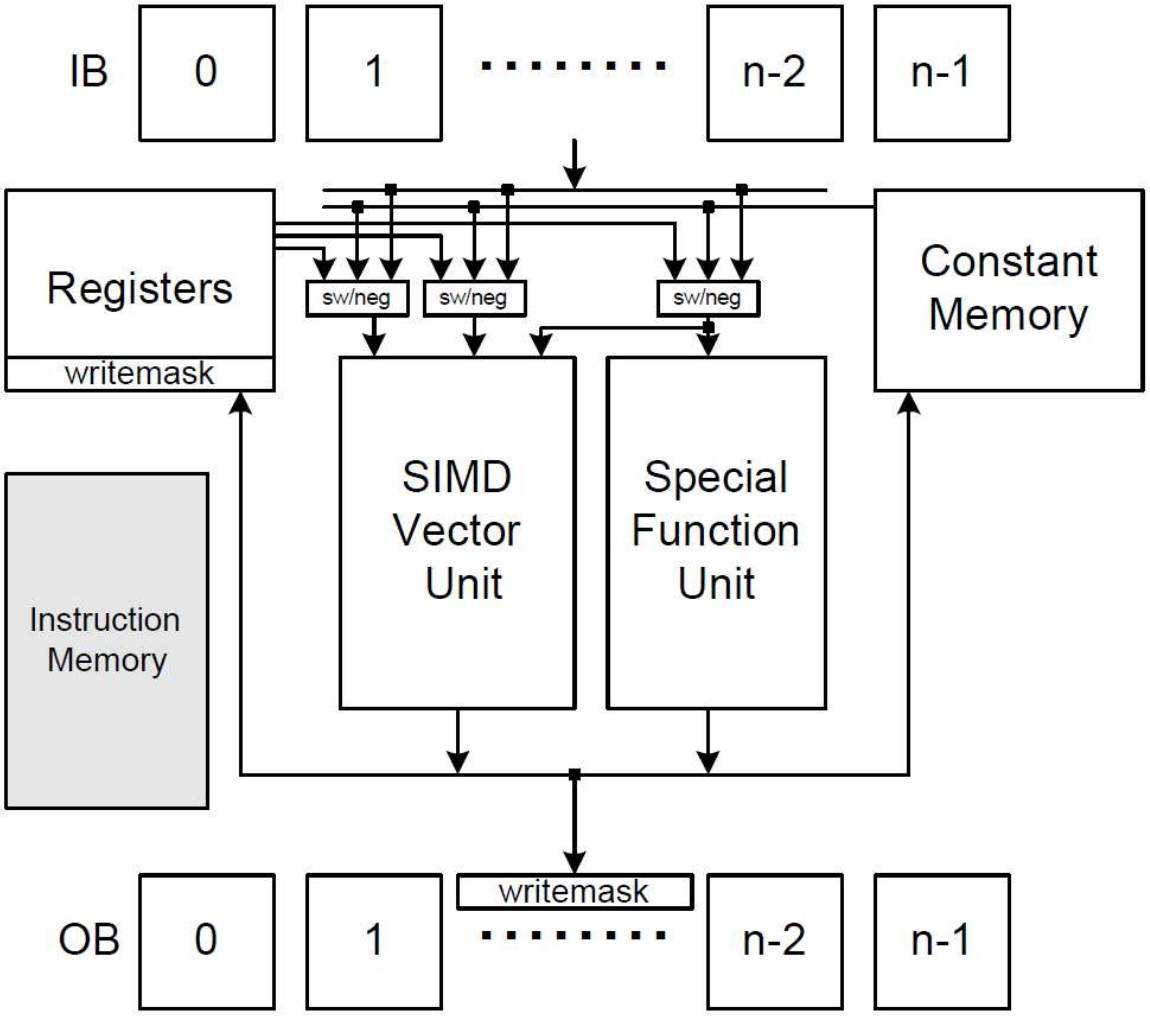


NVIDIA Geforce 6/7
(NV40)
2004, 2005

Legacy Vertex Shading Unit (1)

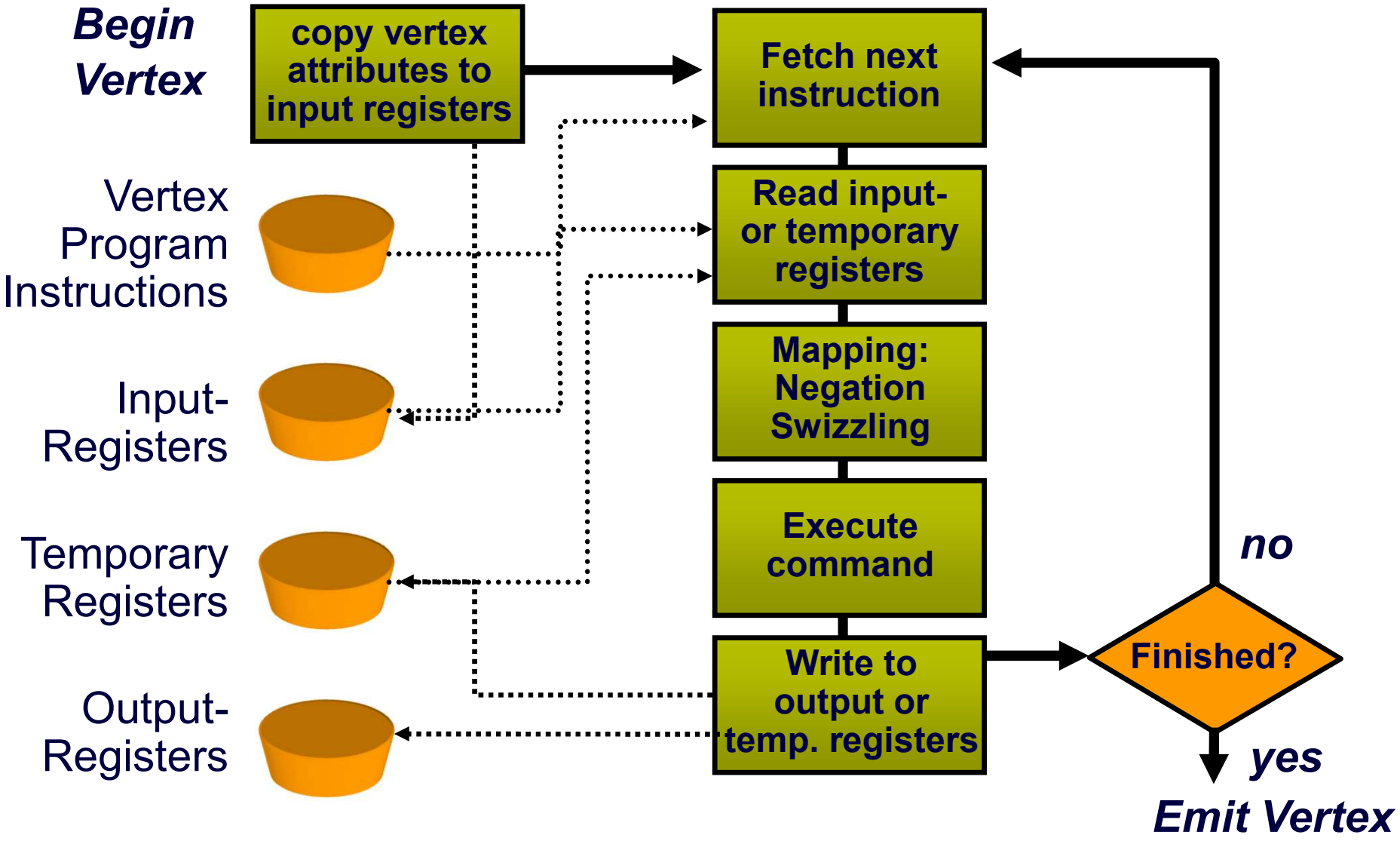
Geforce 3 (NV20), 2001

- floating point 4-vector vertex engine
- still very instructive for understanding GPUs in general



Lindholm et al., A User-Programmable Vertex Engine, SIGGRAPH 2001

Vertex Processor



Legacy Vertex Shading Unit (2)

Input attributes

Vertex Attribute Register	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0	Vertex position	`glVertex`	*x,y,z,w*
1	Vertex weights	`glVertexWeightEXT`	*w,0,0,1*
2	Normal	`glNormal`	
3	Primary color	`glColor`	*r,g,b,a*
4	Secondary color	`glSecondaryColorEXT`	*r,g,b,1*
5	Fog coordinate	`glFogCoordEXT`	*f,0,0,1*
6	-	-	-
7	-	-	-
8	Texture coord 0	`glMultiTexCoordARB(GL_TEXTURE0…)`	*s,t,r,q*
9	Texture coord 1	`glMultiTexCoordARB(GL_TEXTURE1…)`	*s,t,r,q*
10	Texture coord 2	`glMultiTexCoordARB(GL_TEXTURE2…)`	*s,t,r,q*
11	Texture coord 3	`glMultiTexCoordARB(GL_TEXTURE3…)`	*s,t,r,q*
12	Texture coord 4	`glMultiTexCoordARB(GL_TEXTUER4…)`	*s,t,r,q*
13	Texture coord 5	`glMultiTexCoordARB(GL_TEXTUER5…)`	*s,t,r,q*
14	Texture coord 6	`glMultiTexCoordARB(GL_TEXTUER6…)`	*s,t,r,q*
15	Texture coord 7	`glMultiTexCoordARB(GL_TEXTUER7…)`	*s,t,r,q*

Code examples

```
DP4 o[HPOS].x, c[0], v[OPOS];
MUL R1, R0.zxyw, R2.yzxw ;
MAD R1, R0.yzxw, R2.zxyw, -R1;
```

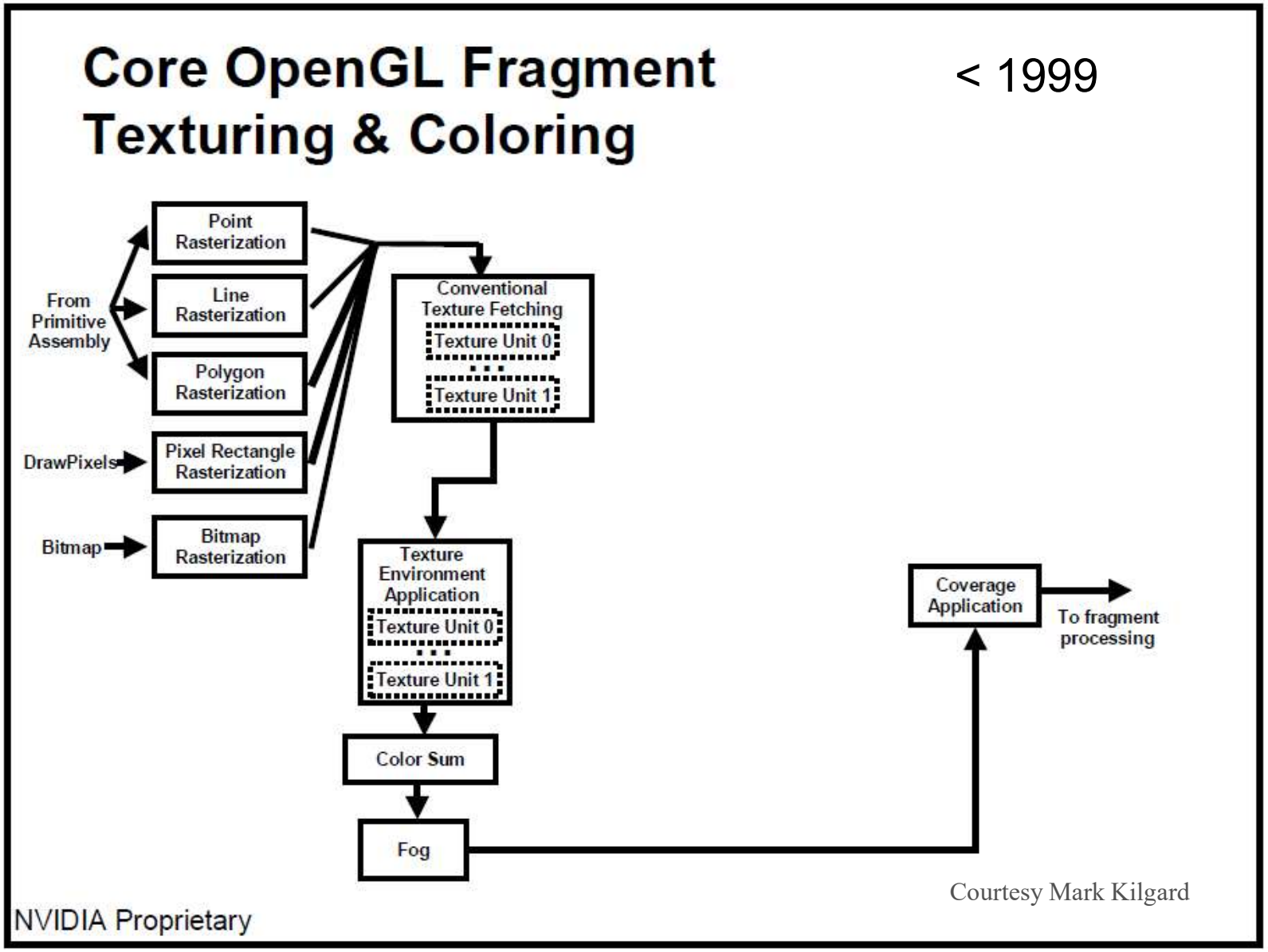
swizzling!

Legacy Vertex Shading Unit (3)

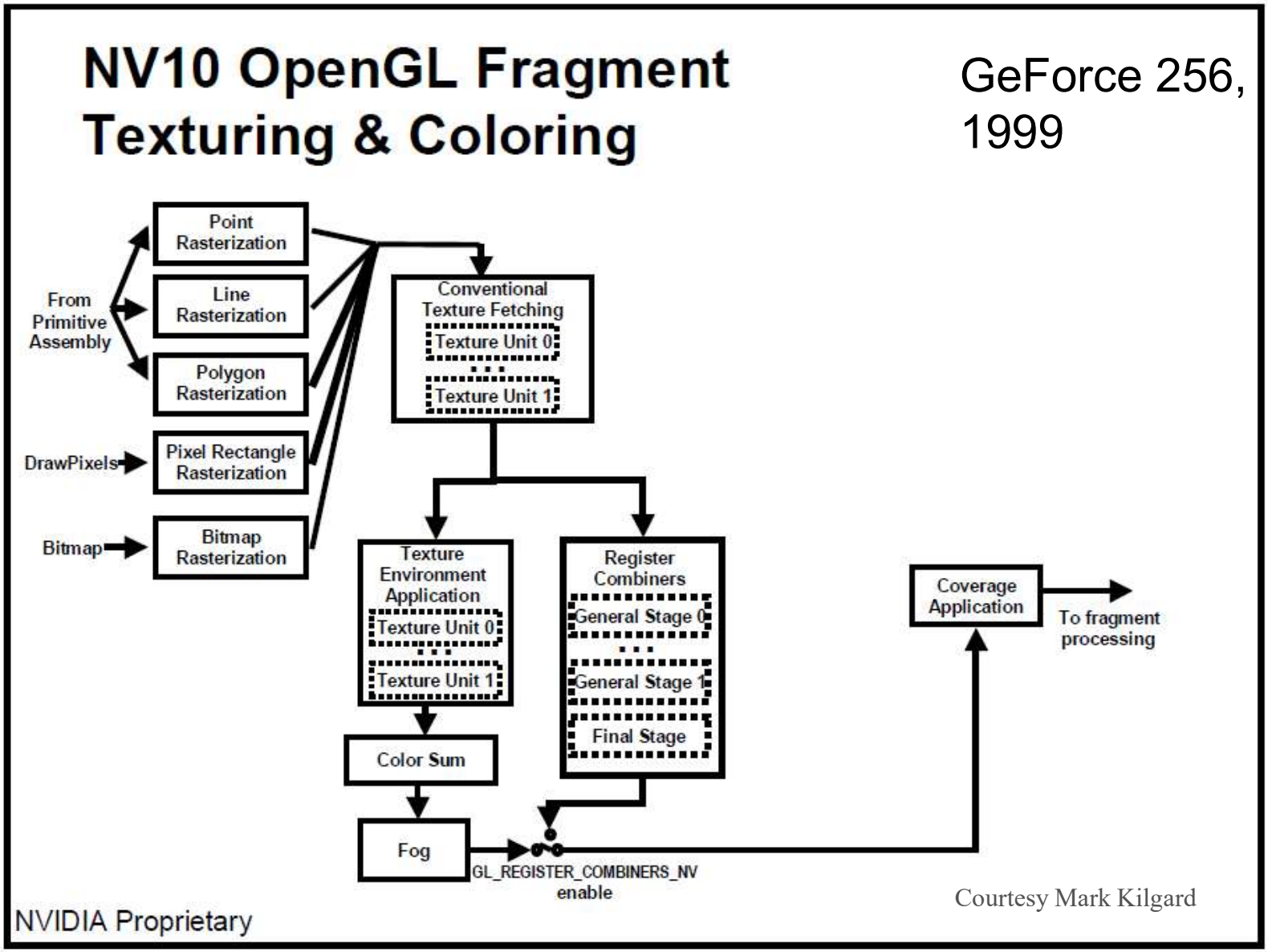
Vector instruction set, very few instructions; **no branching** yet!

OpCode	Full Name	Description
MOV	Move	vector -> vector
MUL	Multiply	vector -> vector
ADD	Add	vector -> vector
MAD	Multiply and add	vector -> vector
DST	Distance	vector -> vector
MIN	Minimum	vector -> vector
MAX	Maximum	vector -> vector
SLT	Set on less than	vector -> vector
SGE	Set on greater or equal	vector -> vector
RCP	Reciprocal	scalar-> replicated scalar
RSQ	Reciprocal square root	scalar-> replicated scalar
DP3	3 term dot product	vector-> replicated scalar
DP4	4 term dot product	vector-> replicated scalar
LOG	Log base 2	miscellaneous
EXP	Exp base 2	miscellaneous
LIT	Phong lighting	miscellaneous
ARL	Address register load	miscellaneous

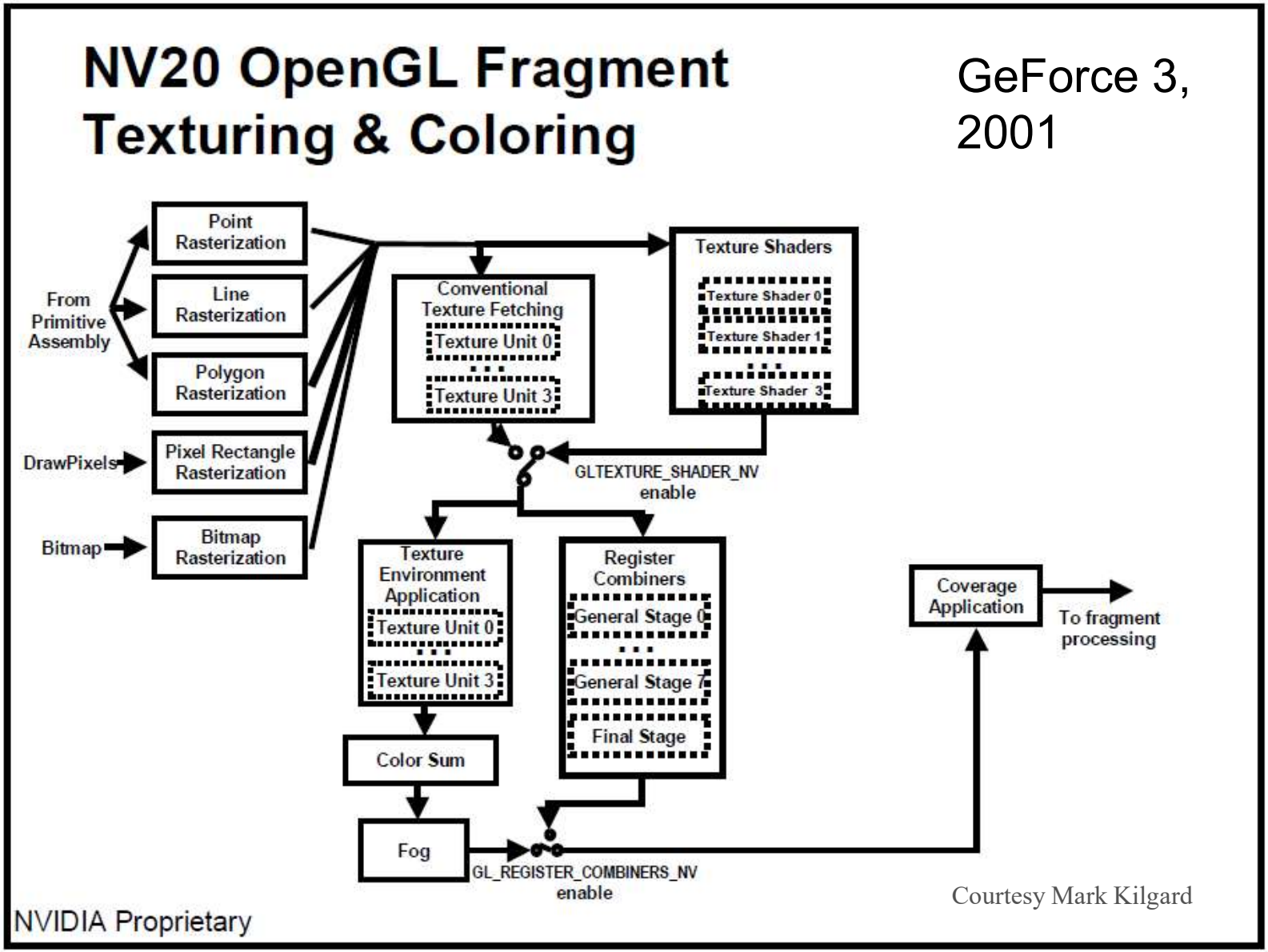
Fast Forward to Programm. Fragment Shading



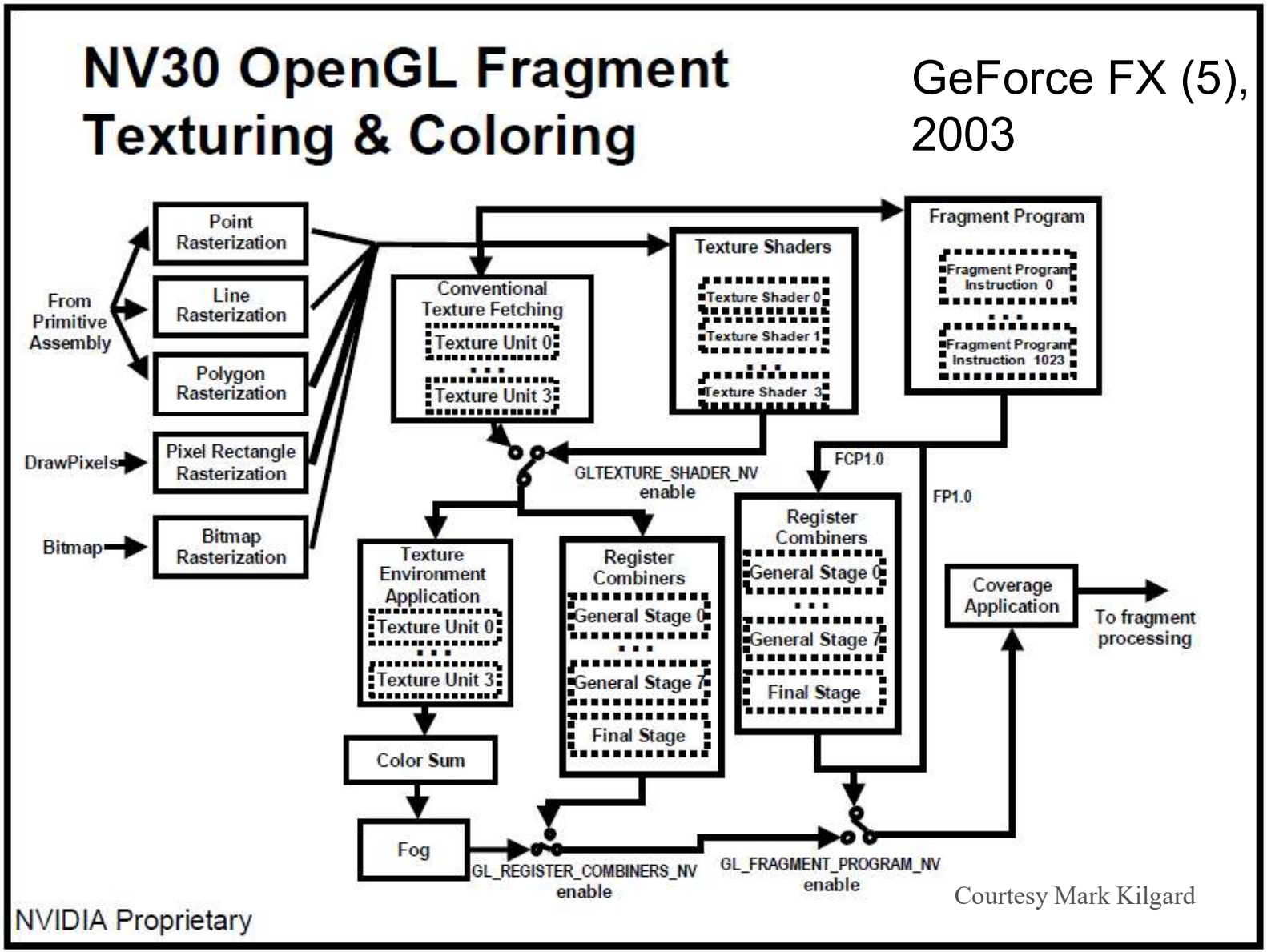
Fast Forward to Programm. Fragment Shading



Fast Forward to Programm. Fragment Shading



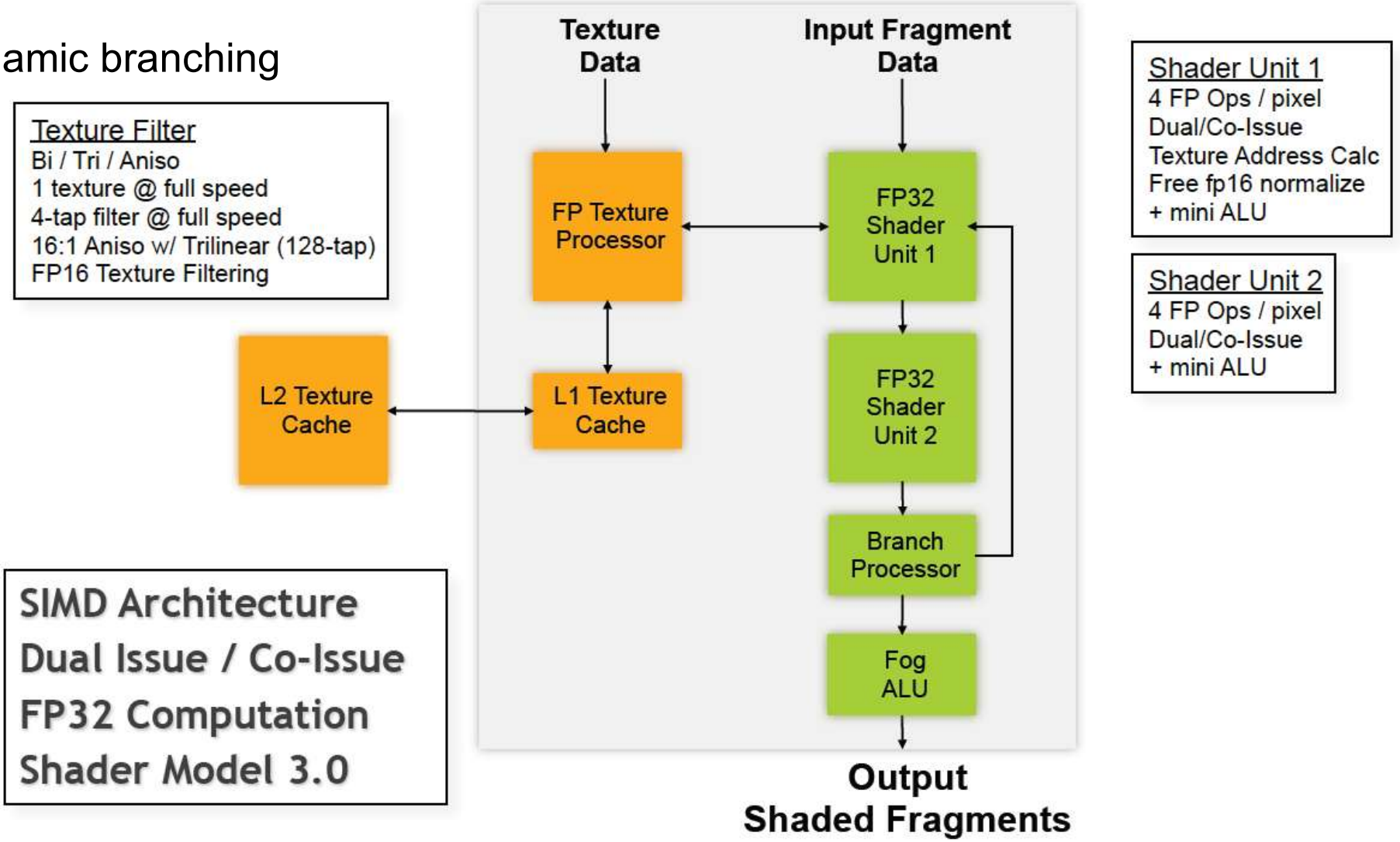
Fast Forward to Programm. Fragment Shading



Legacy Fragment Shading Unit (1)

GeForce 6 (NV40), 2004

- dynamic branching

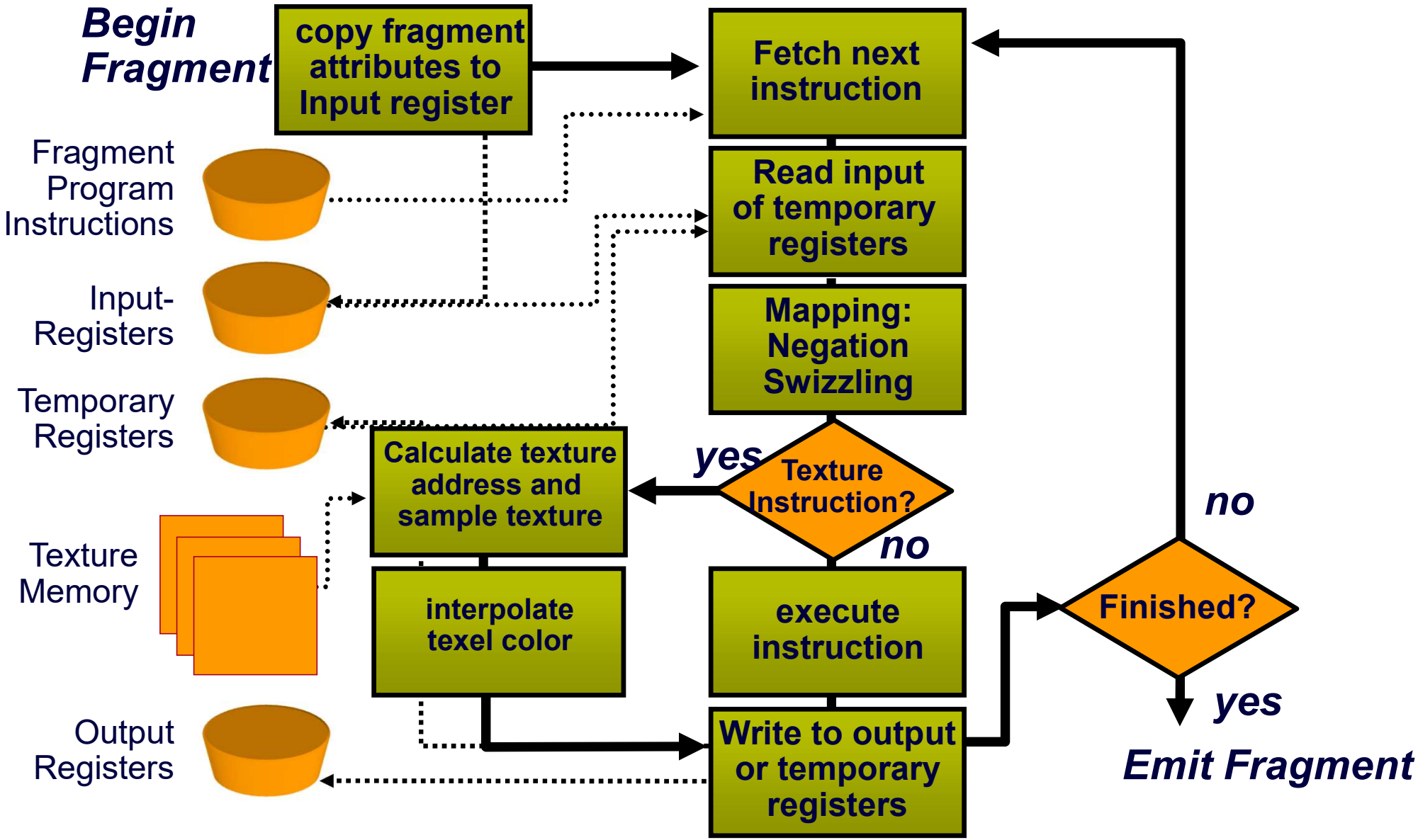


Legacy Fragment Shading Unit (2)

Example code

```
!!ARBfp1.0

ATTRIB unit_tc = fragment.texcoord[ 0 ];
PARAM  mvp_inv[]  = { state.matrix.mvp.inverse };
PARAM  constants  = {0, 0.999, 1, 2};

TEMP pos_win, temp;

TEX pos_win.z, unit_tc, texture[ 1 ], 2D;

ADD pos_win.w, constants.y, -pos_win.z;
KIL pos_win.w;

MOV result.color.w, pos_win.z;

MOV pos_win.xyw, unit_tc;
MAD pos_win.xyz, pos_win, constants.a, -constants.b;

DP4 temp.w, mvp_inv[ 3 ], pos_win;
RCP temp.w, temp.w;

MUL pos_win, pos_win, temp.w;

DP4 result.color.x, mvp_inv[ 0 ], pos_win;
DP4 result.color.y, mvp_inv[ 1 ], pos_win;
DP4 result.color.z, mvp_inv[ 2 ], pos_win;

END
```

Fragment Processor



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, but no explicit 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);
}
```

```
<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)
```

1 shaded fragment output record

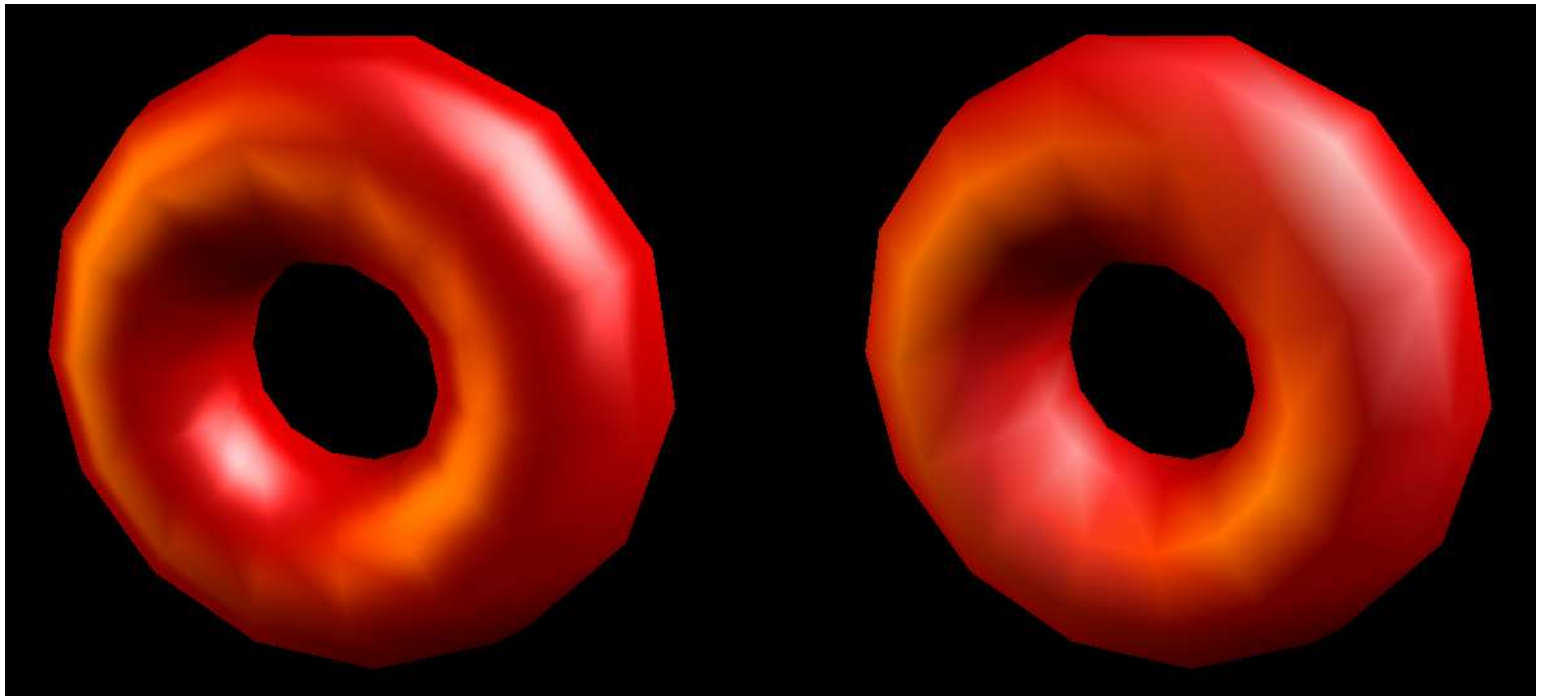
Per-Pixel(Fragment) Lighting

Simulating smooth surfaces by calculating illumination for each fragment

Example: specular highlights (Phong illumination/shading)

Phong shading:
per-fragment evaluation

Gouraud shading:
linear interpolation from vertices



Per-Pixel Phong Lighting (Cg)

```
void main(float4 position  : TEXCOORD0,
          float3 normal    : TEXCOORD1,

      out float4 oColor     : COLOR,

  uniform float3 ambientCol,
  uniform float3 lightCol,
  uniform float3 lightPos,
  uniform float3 eyePos,
  uniform float3 Ka,
  uniform float3 Kd,
  uniform float3 Ks,
  uniform float  shiny)
{
```

Per-Pixel Phong Lighting (Cg)

```
  float3 P = position.xyz;
  float3 N = normal;
  float3 V = normalize(eyePosition - P);
  float3 H = normalize(L + V);

  float3 ambient = Ka * ambientCol;

  float3 L         = normalize(lightPos - P);
  float  diffLight = max(dot(L, N), 0);
  float3 diffuse   = Kd * lightCol * diffLight;

  float specLight = pow(max(dot(H, N), 0), shiny);
  float3 specular = Ks * lightCol * specLight;

  oColor.xyz = ambient + diffuse + specular;
  oColor.w = 1;
}
```

GPU Architecture: General Architecture

Concepts: Latency vs. Throughput

Latency

- What is the time *between start and finish* of an operation/computation?
- How long does it take between starting to execute an instruction until the execution is actually finished / its results are available?
- Examples: 1 FP32 MUL instruction; 1 vertex computation, …

Throughput

- How many computations (operations/instructions) *finish per time unit*?
- How many instructions of a certain type (e.g., FP32 MUL) finish per time unit (per clock cycle, per second)?

GPUs: ***High-throughput execution*** (at the expense of latency)
(but: *hide* latencies to avoid throughput going down)

Concepts: Types of Parallelism

Instruction level parallelism (ILP)

- In single instruction stream: Can consecutive instructions/operations be executed in parallel? (Because they don't have a dependency)
- Exploit ILP: Execute independent instructions (1) via pipelined execution (instr. pipe), or even (2) in multiple parallel instruction pipelines (superscalar processors)
- On GPUs: also important, but much less than TLP (compare, e.g., Kepler with current GPUs)

Thread level parallelism (TLP)

- Exploit that by definition operations in different threads are independent (if no explicit communication/synchronization is used, which should be minimized)
- Exploit TLP: Execute operations/instructions from multiple threads in parallel (which also needs multiple parallel instruction pipelines)
- **On GPUs: main type of parallelism**

more types:
- Bit-level parallelism (processor word size: 64 bits instead of 32, etc.)
- Data parallelism (SIMD/vector instructions), task parallelism, …

Concepts: Latency Hiding

Not about latency of single operation or group of operations:
It's about avoiding that the *throughput* goes below peak

Hide latency that *does* occur for one instruction (group) by *executing a different instruction (group)* as soon as current one stalls:

→ *Total throughput does not go down*

In GPUs, hide latencies via:

- **TLP: pull independent, not-stalling instruction from other thread group**
- ILP: pull independent instruction from down the inst. stream in same thread group
- Depending on GPU: TLP often sufficient, but sometimes also need ILP
- However: If in one cycle TLP doesn't work, ILP can jump in or vice versa

SIGGRAPH2009 NEW ORLEANS

From Shader Code to a Teraflop: How Shader Cores Work

Kayvon Fatahalian
Stanford University

Where this is going...

Summary: three key ideas for high-throughput execution

1. Use many "slimmed down cores," run them in parallel

2. Pack cores full of ALUs (by sharing instruction stream overhead 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

Where this is going...

Summary: three key ideas for high-throughput execution

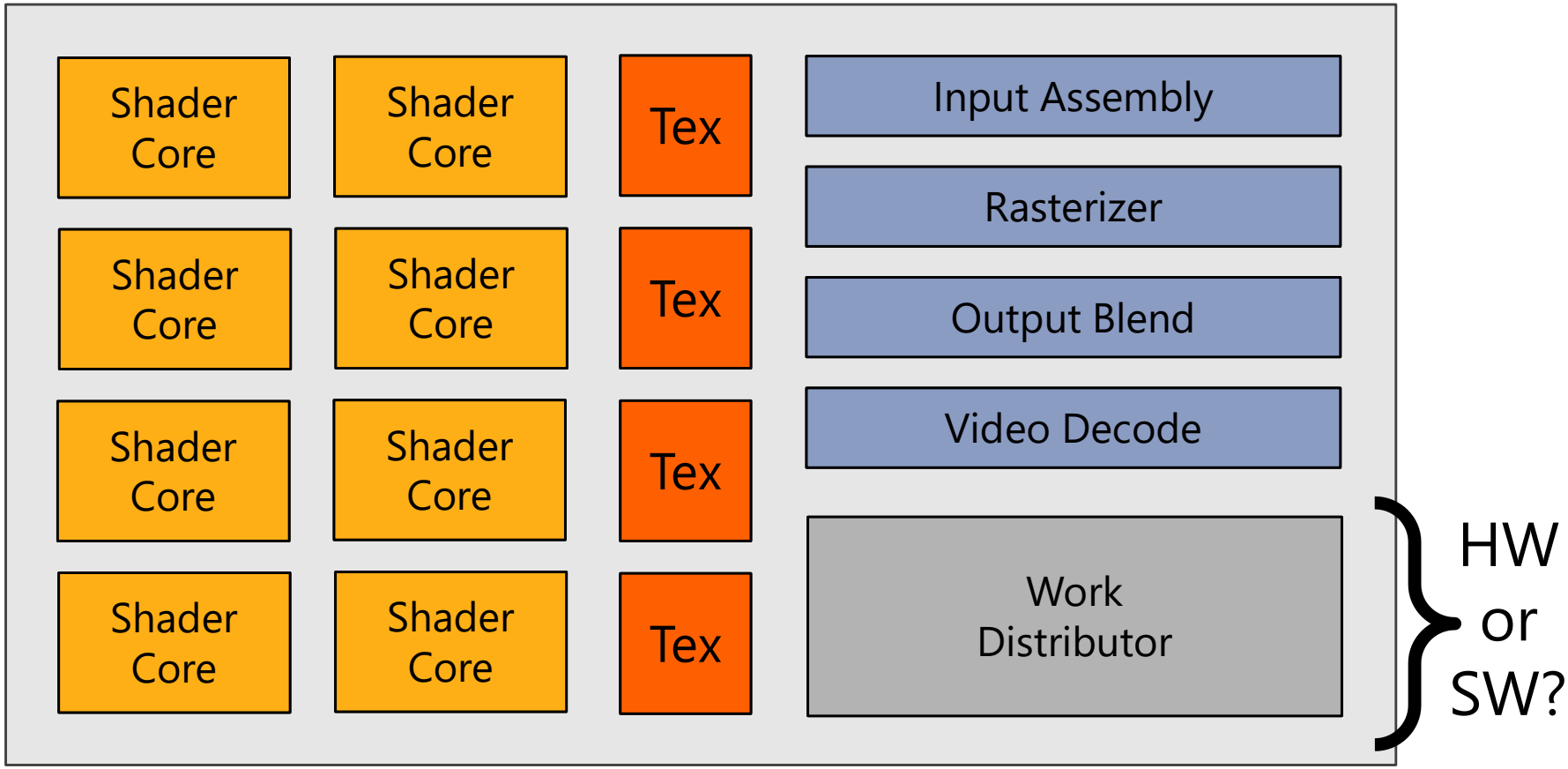
1. Use many "slimmed down cores," run them in parallel

2. Pack cores full of ALUs (by sharing instruction stream overhead across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

   GPUs are here! (usually)

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group

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, but no explicit parallelism

Compile shader

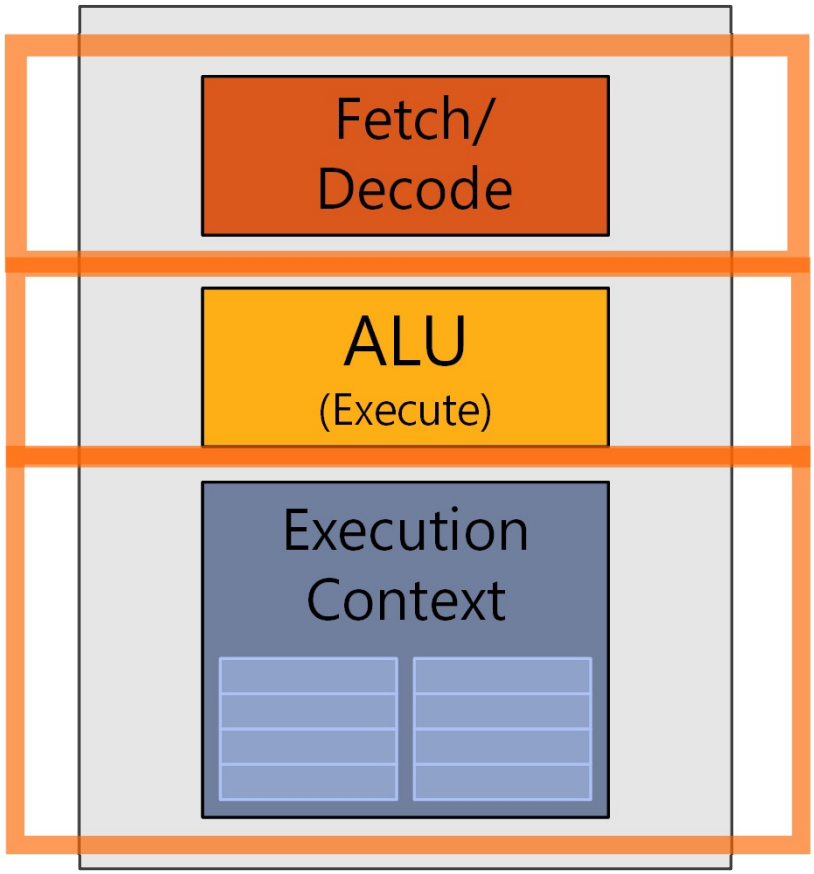
1 unshaded fragment input record

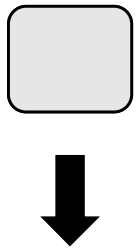
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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);
}
```

```
<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)
```

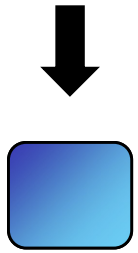
1 shaded fragment output record

Execute shader

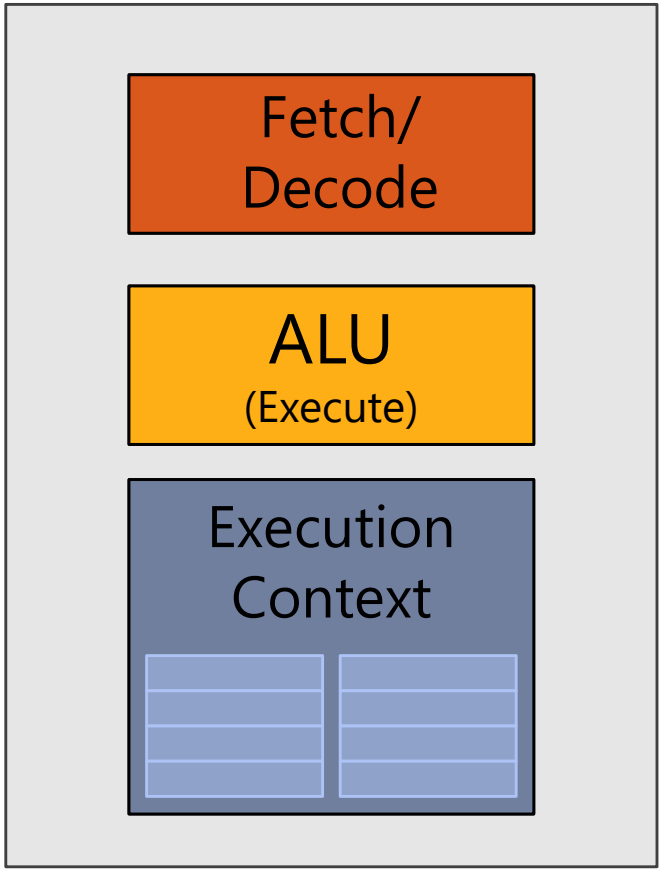


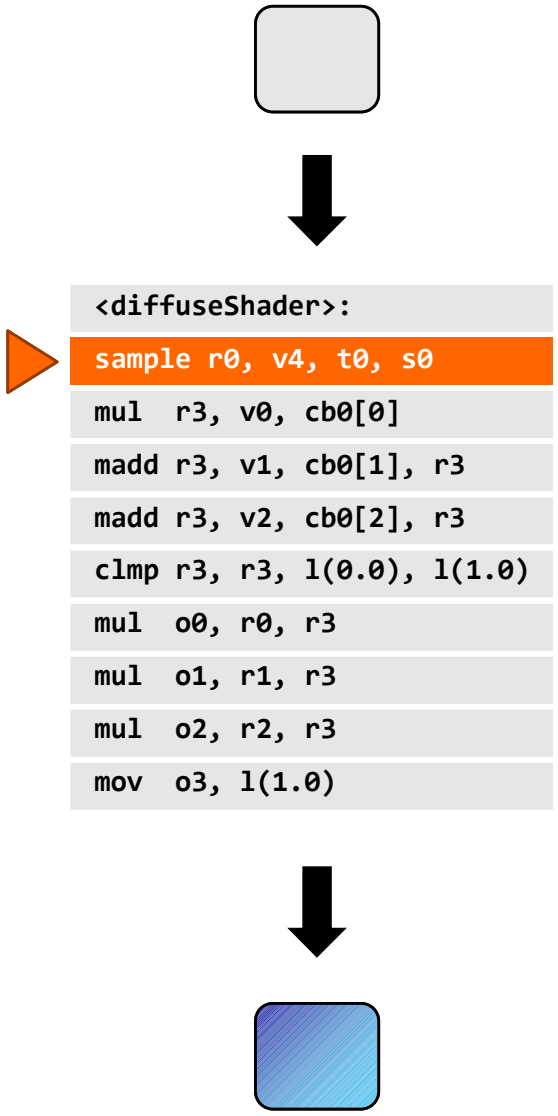


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

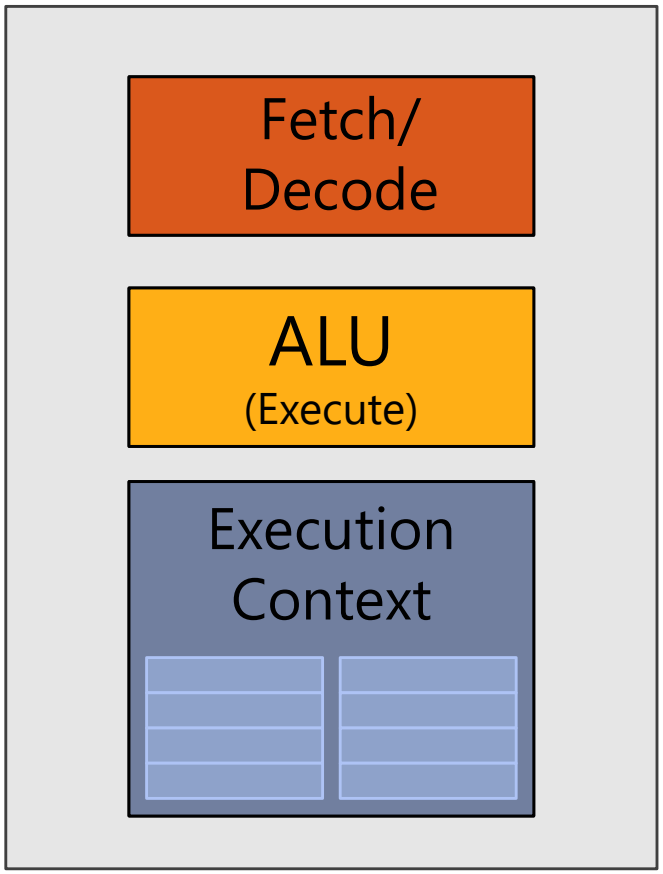


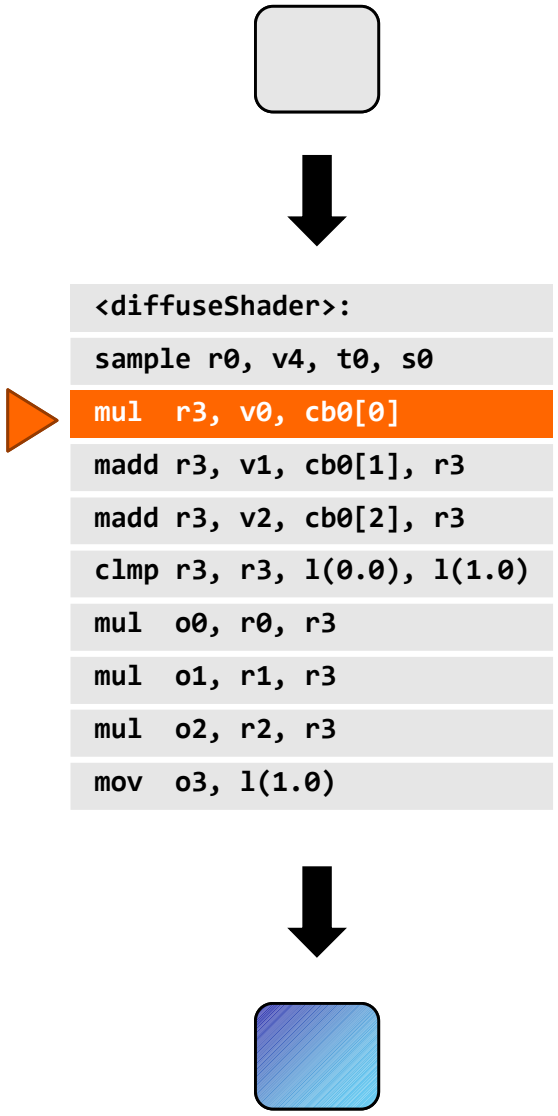
Execute shader



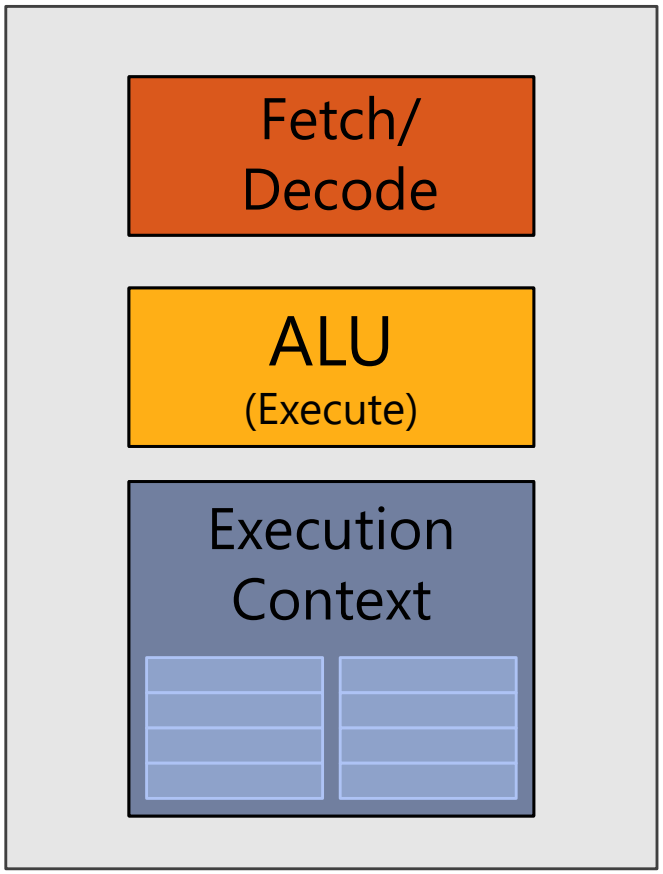


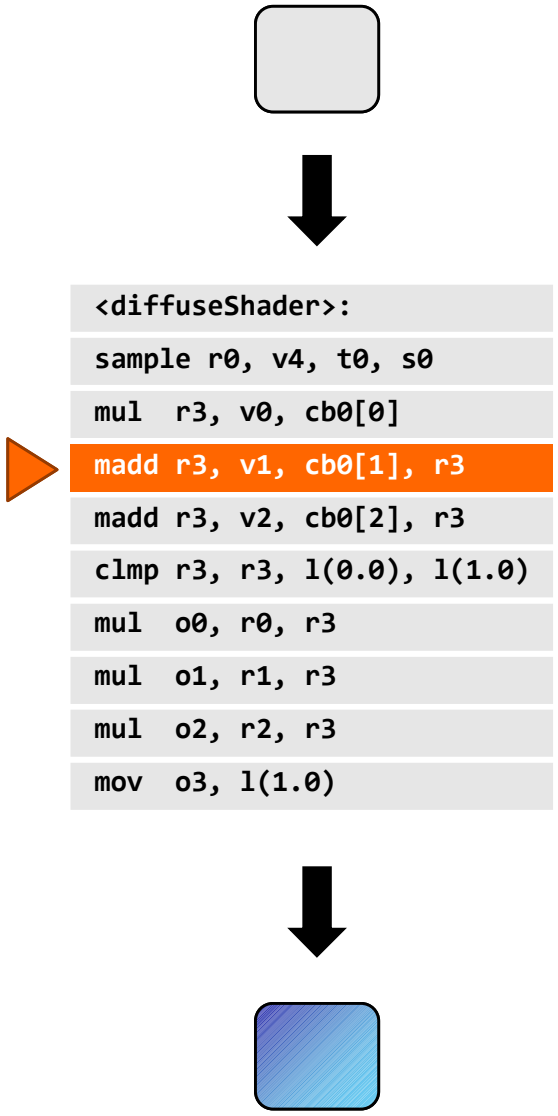
Execute shader



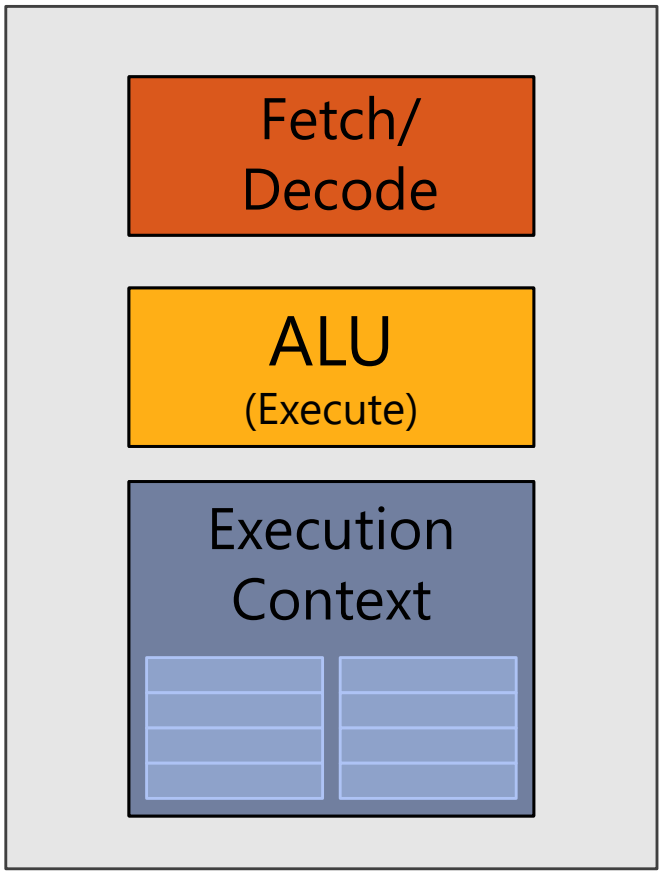


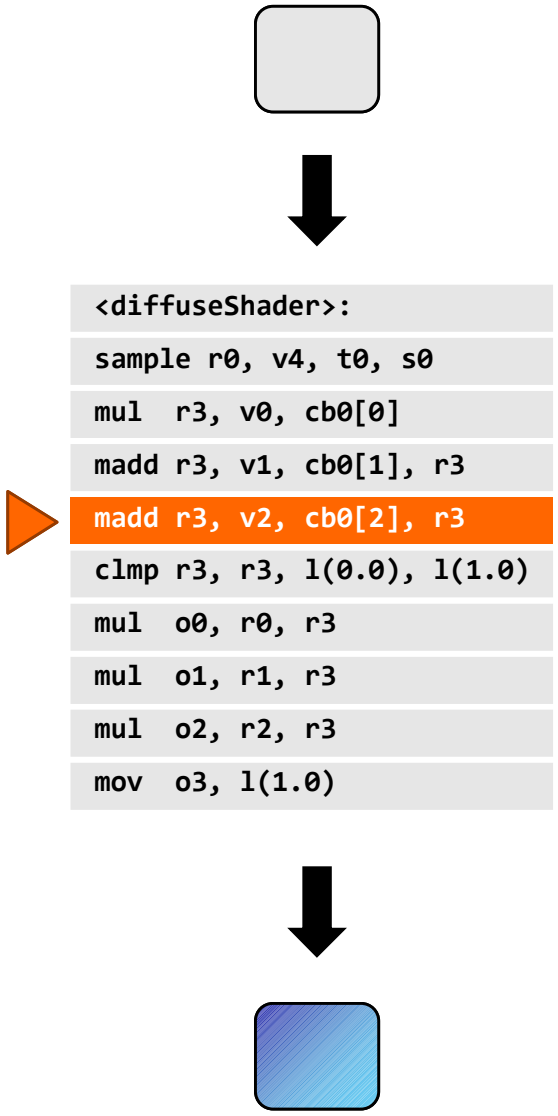
Execute shader



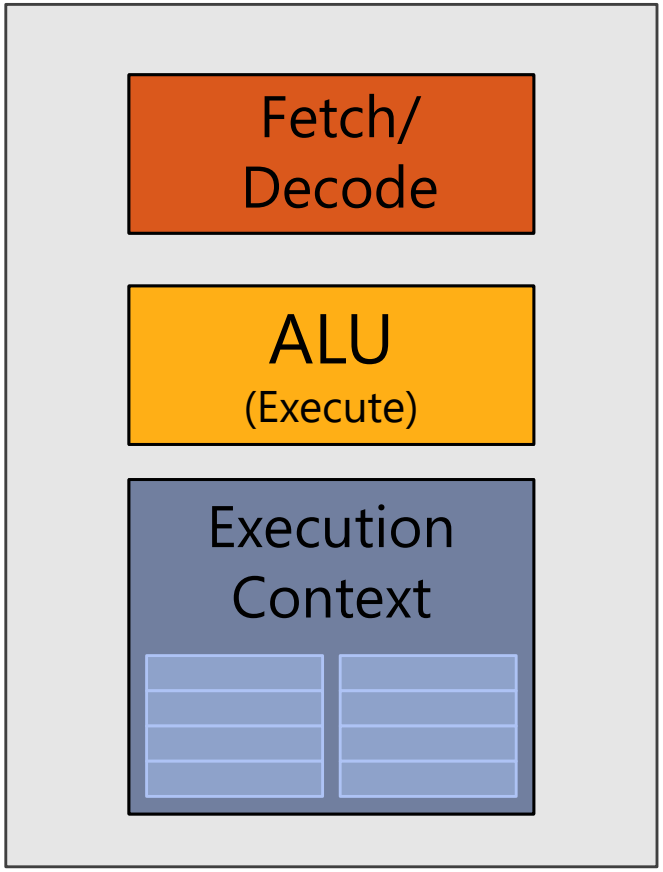


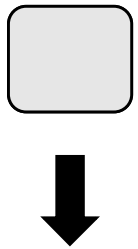
Execute shader



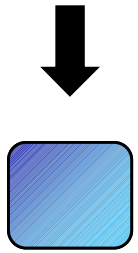


Execute shader



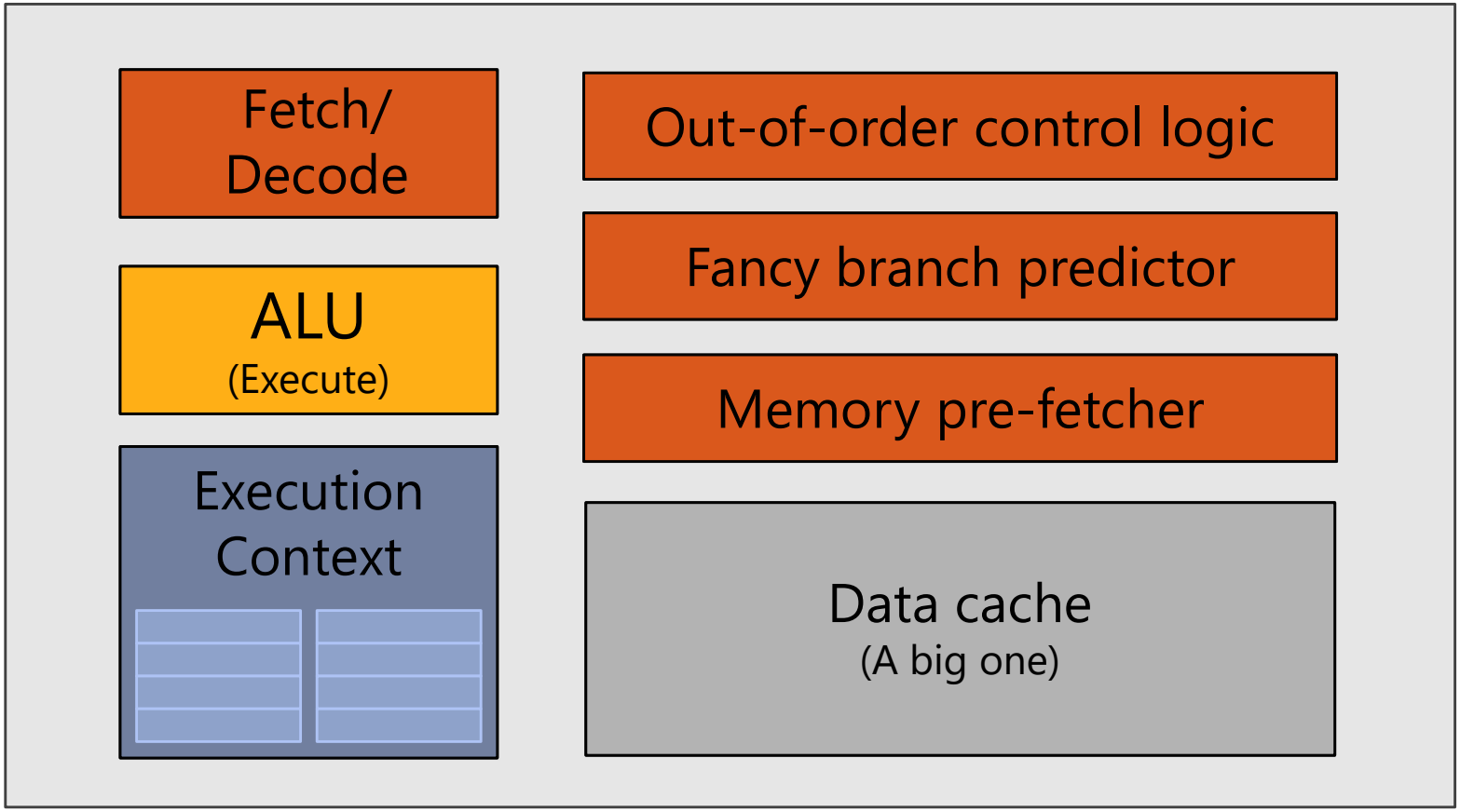


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

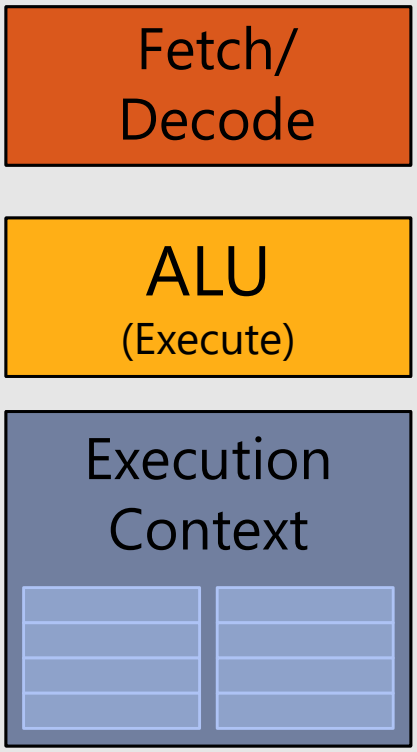


GPU Architecture
Big Idea #1

CPU-"style" cores



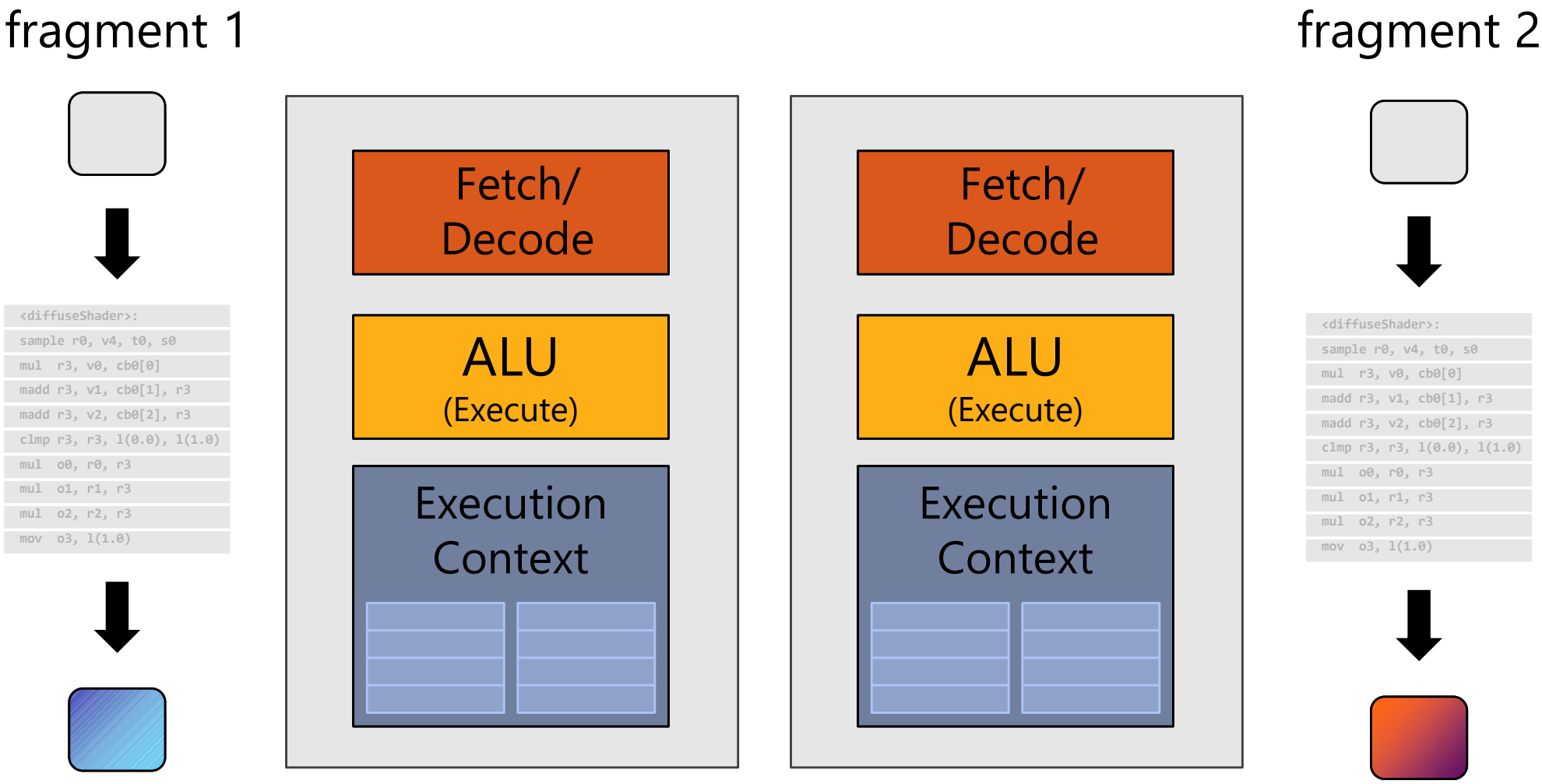
Idea #1: Slim down



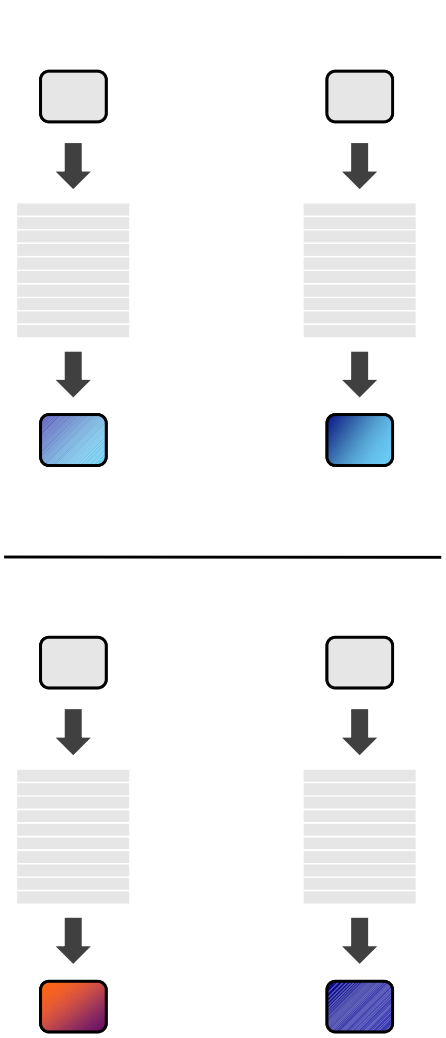
Idea #1:

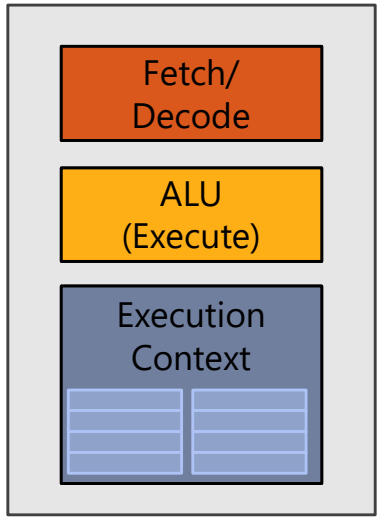
Remove components that help a single instruction stream run fast (i.e., that reduce latency of a single instruction stream)

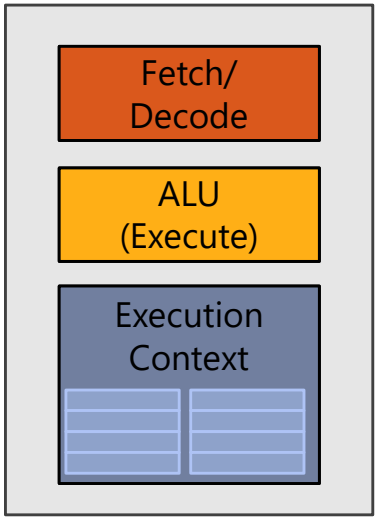
Two cores (two fragments in parallel)

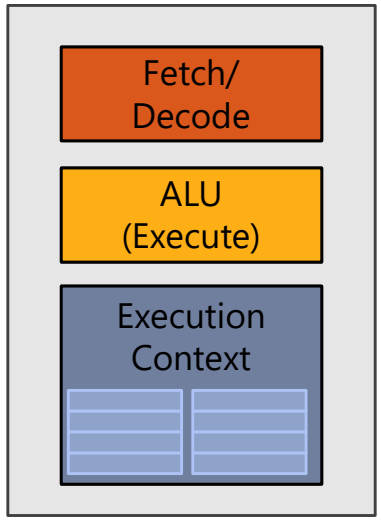


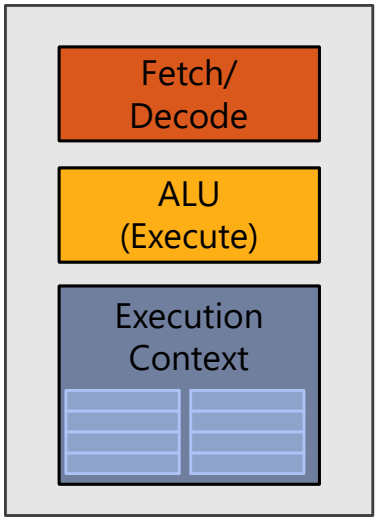
Four cores (four fragments in parallel)











Sixteen cores (sixteen fragments in parallel)

16 cores = 16 simultaneous instruction streams

Thank you.