

EE382N (20): Computer Architecture - Parallelism and Locality  
Fall 2011

## Lecture 16 – GPUs (I)

Mattan Erez



The University of Texas at Austin

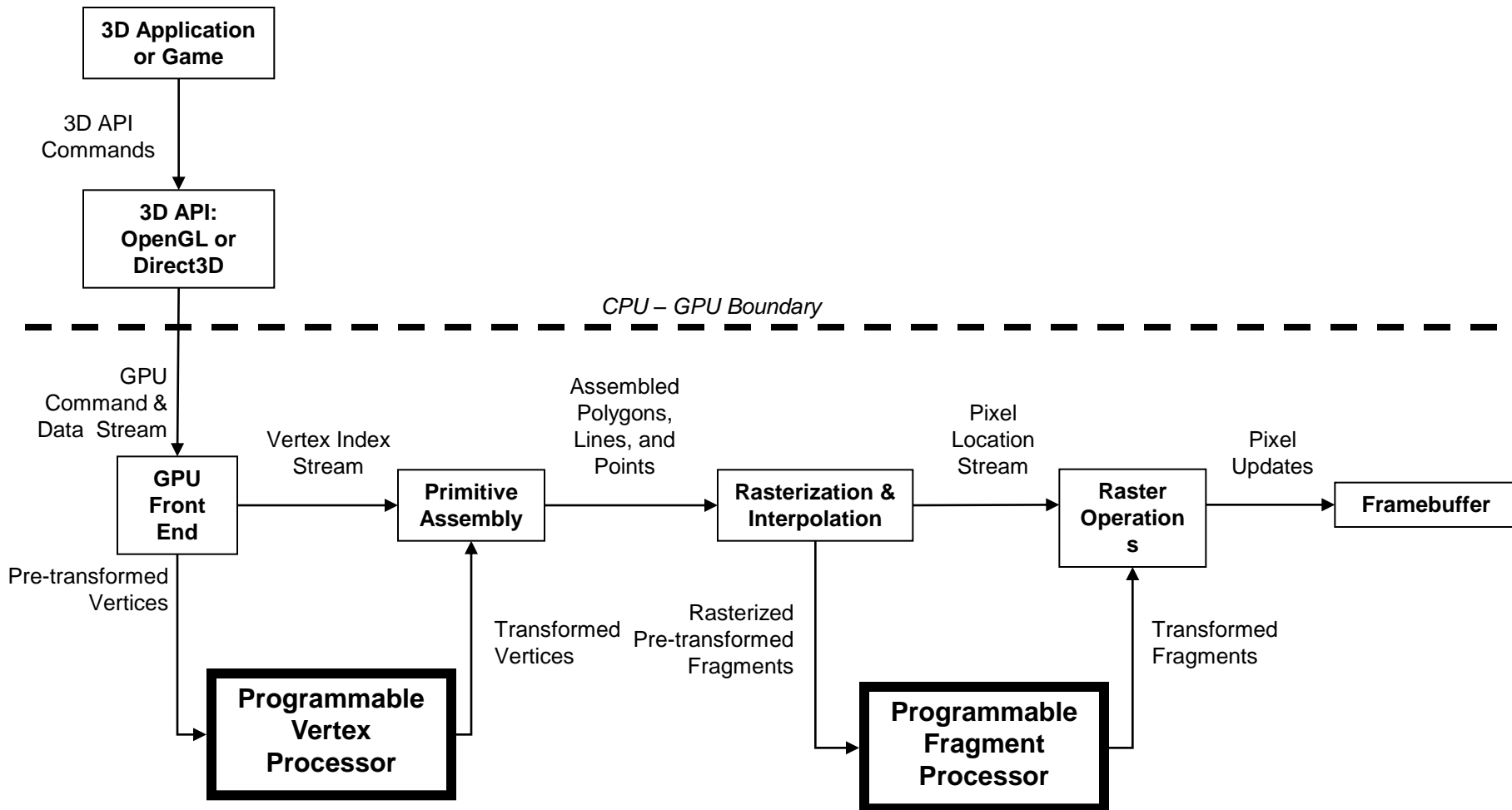


# A GPU Renders 3D Scenes

- 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:



# Adding Programmability to the Graphics Pipeline



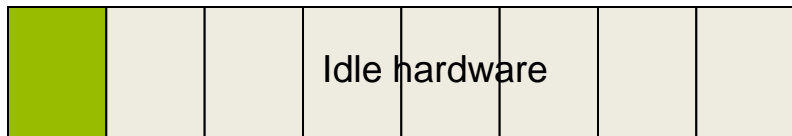
# Vertex and Fragment Processing Share Unified Processing Elements

- Load balancing HW is a problem

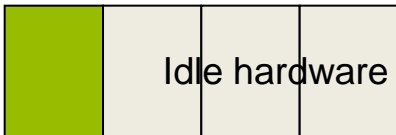
Vertex Shader



Pixel Shader



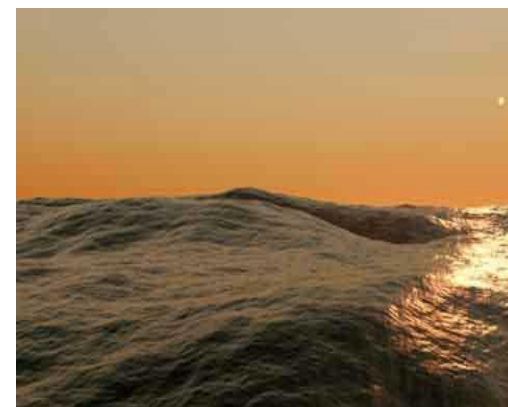
Vertex Shader



Pixel Shader



Heavy Geometry  
Workload Perf = 4



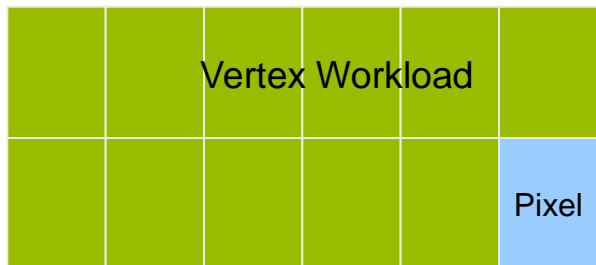
Heavy Pixel  
Workload Perf = 8



# Vertex and Fragment Processing Share Unified Processing Elements

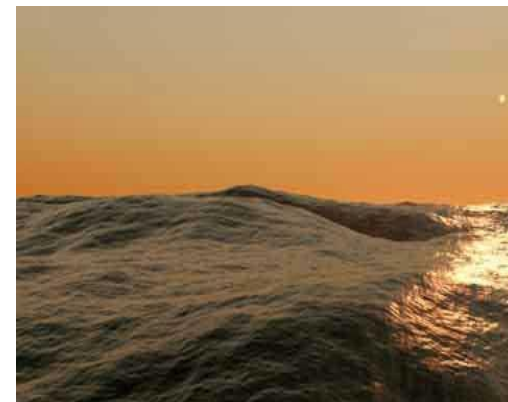
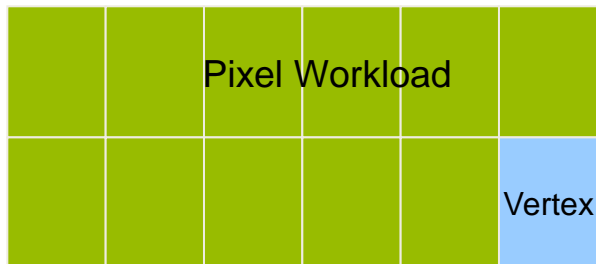
- Load balancing SW is easier

## Unified Shader



Heavy Geometry  
Workload Perf = 11

## Unified Shader

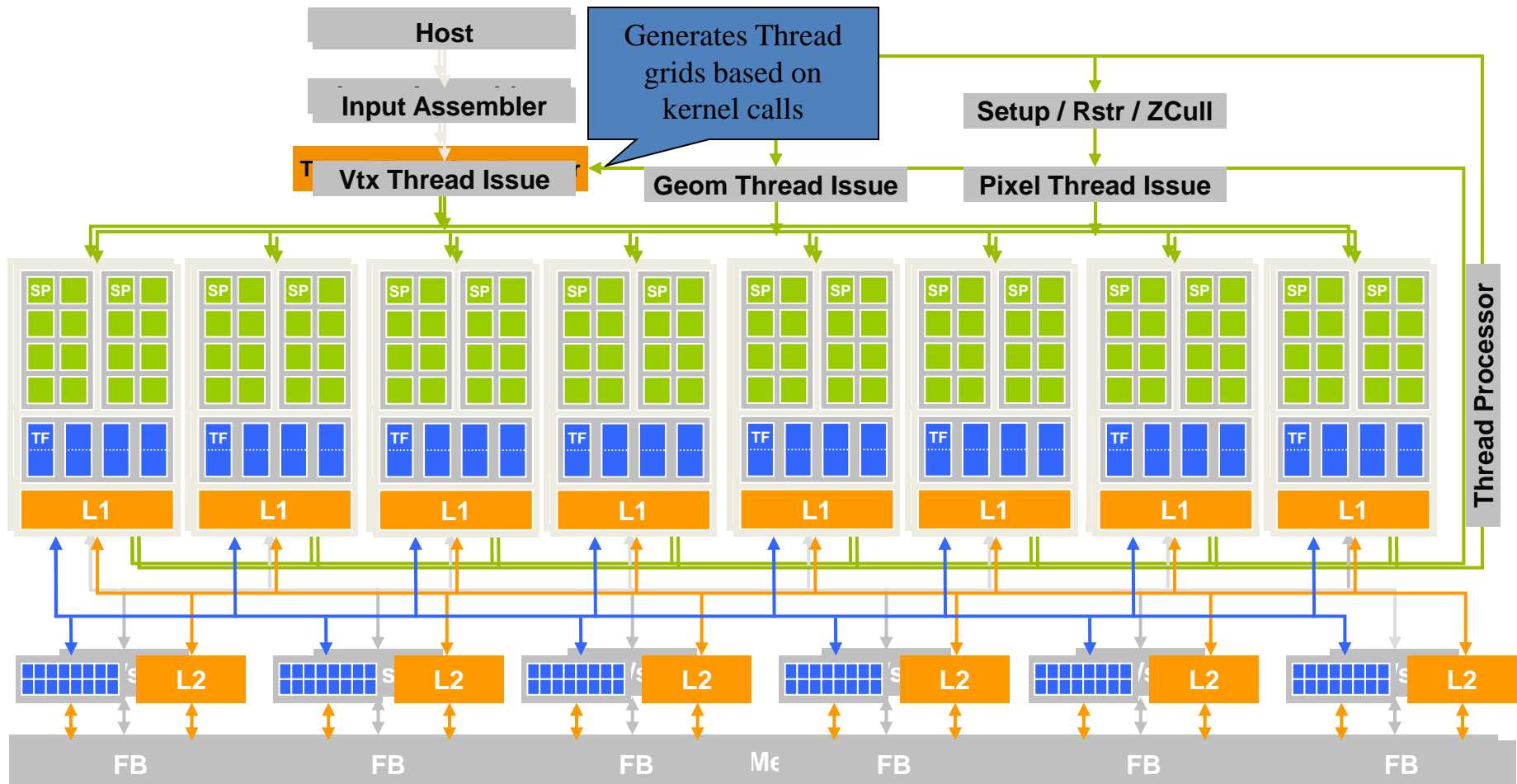


Heavy Pixel  
Workload Perf = 11

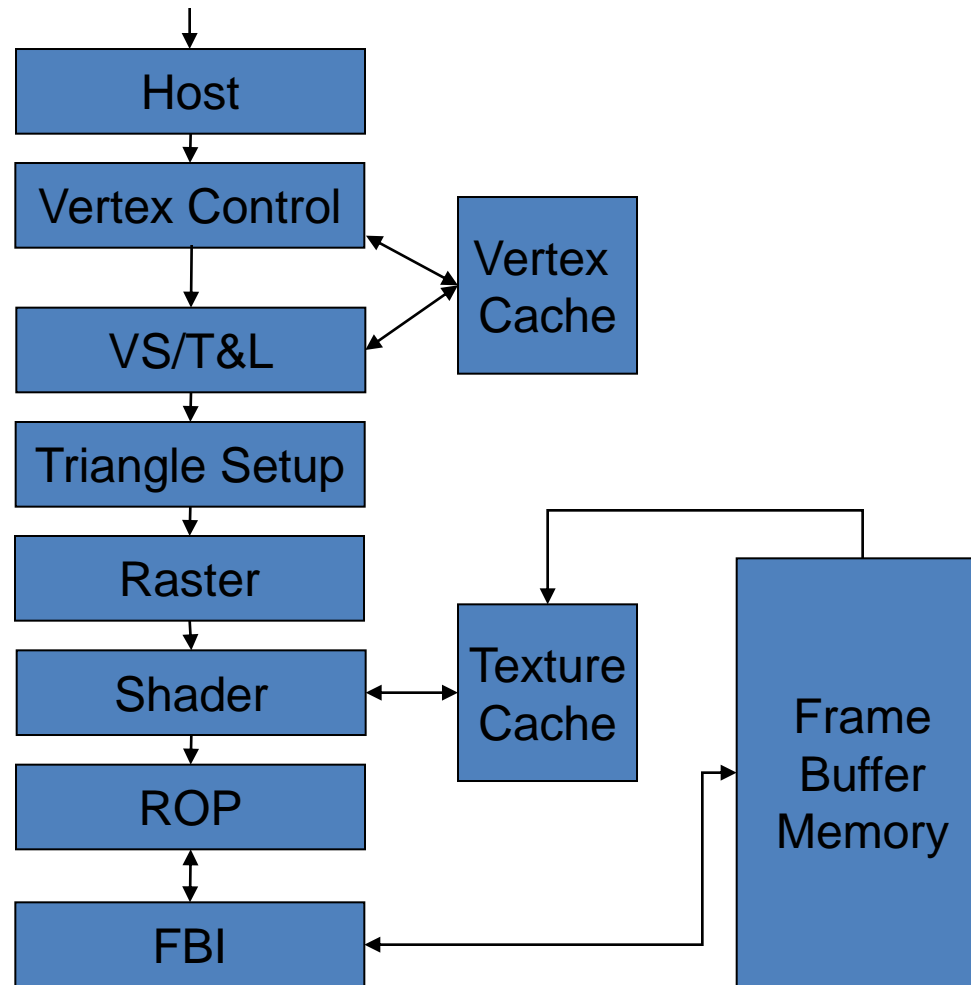


# Make the Compute Core The Focus of the Architecture

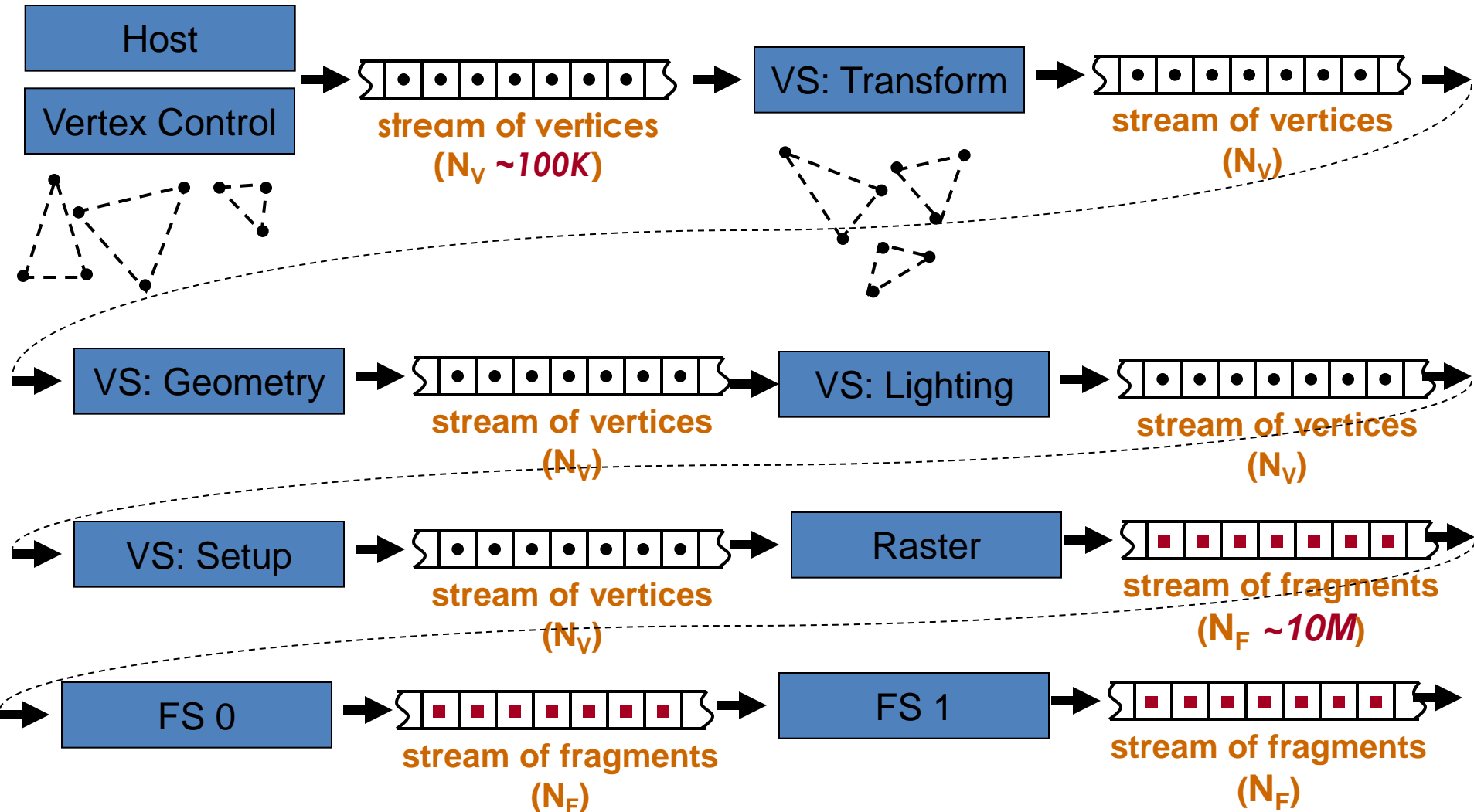
- The future of GPUs is programmable processing
- Processors execute computing threads
- So – build the architecture around the processor
- Alternative operating mode specifically for computing



# The NVIDIA GeForce Graphics Pipeline



# Another View of the 3D 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**

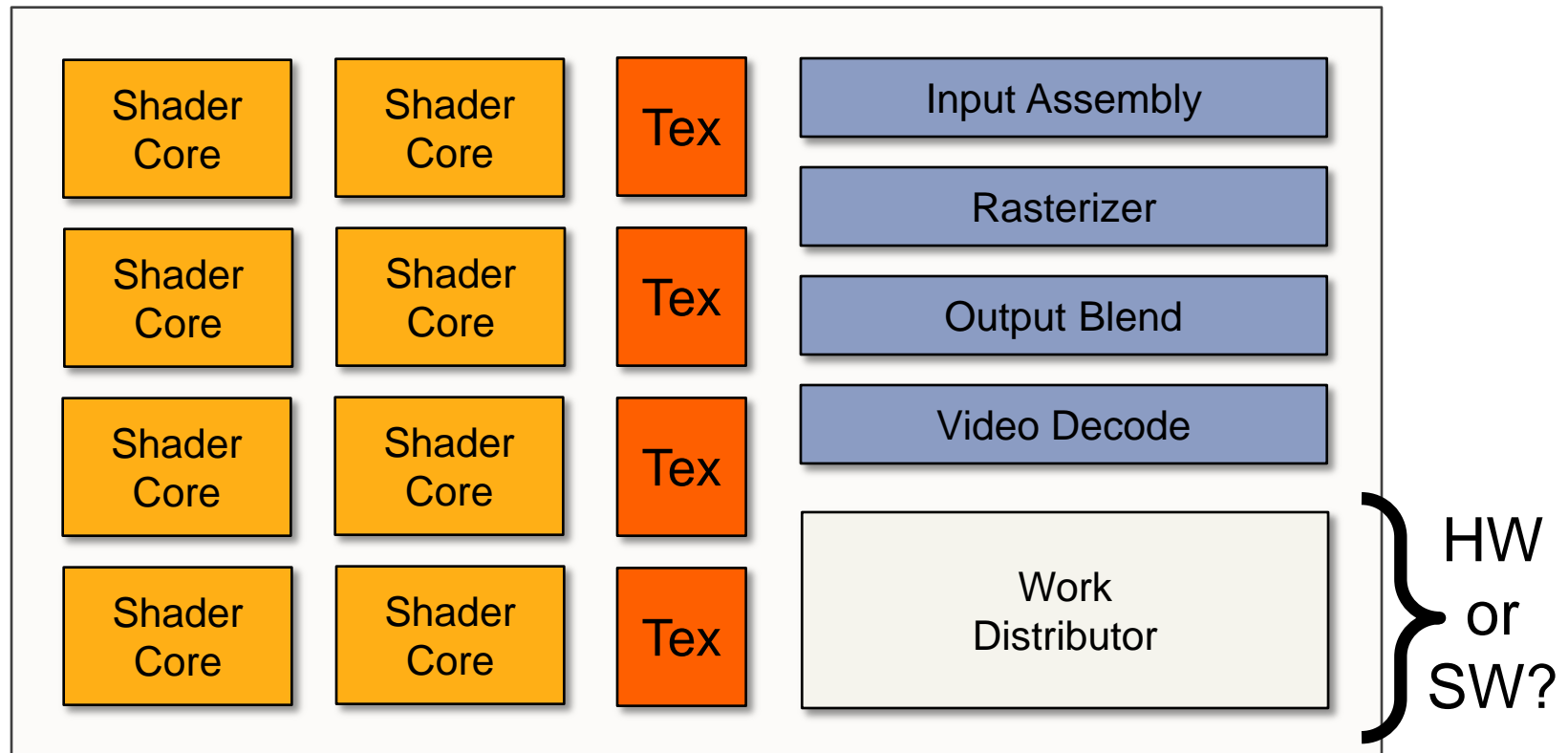


# From Shader Code to a **Teraflop**: How Shader Cores Work

Kayvon Fatahalian  
CMU



# 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);
}
```



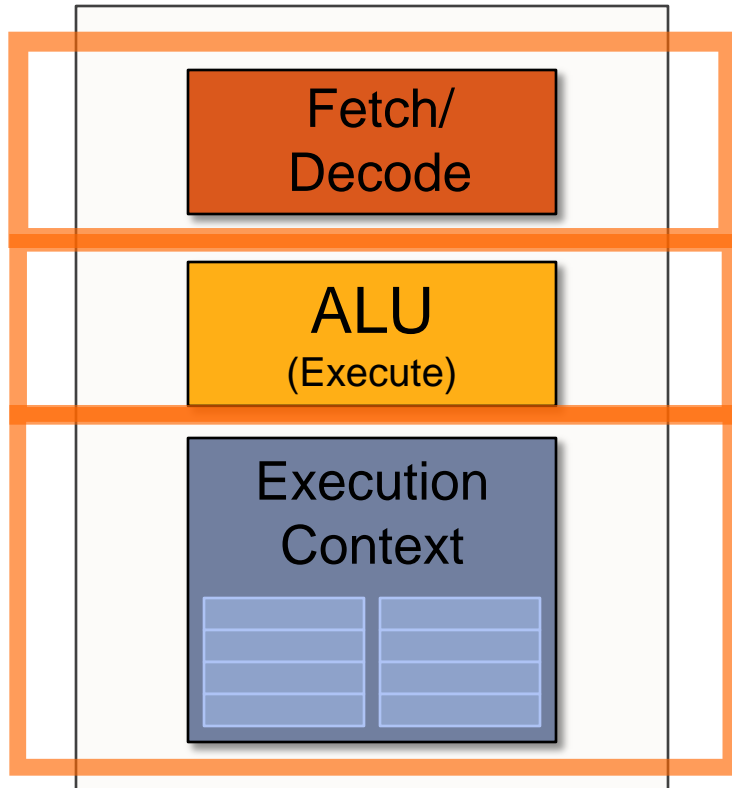
```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
```



1 shaded fragment output record



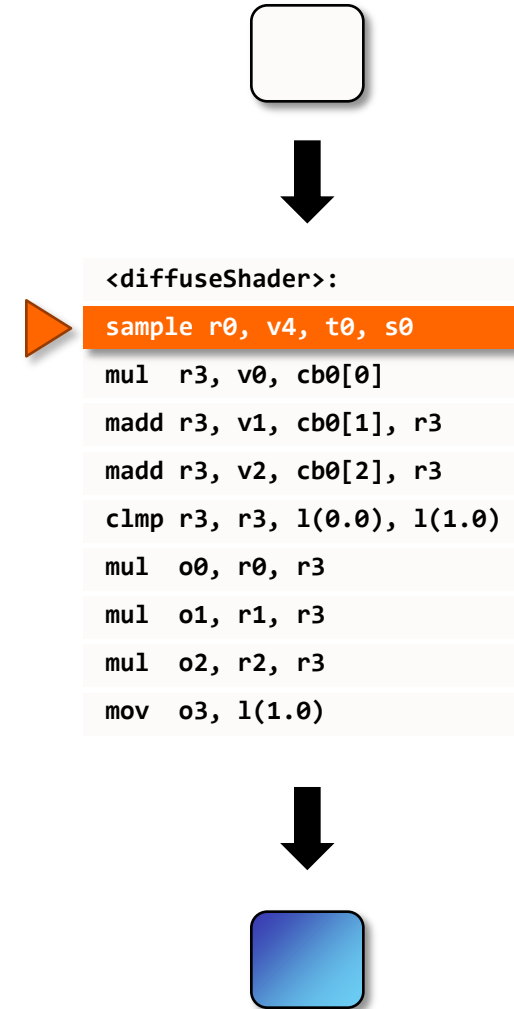
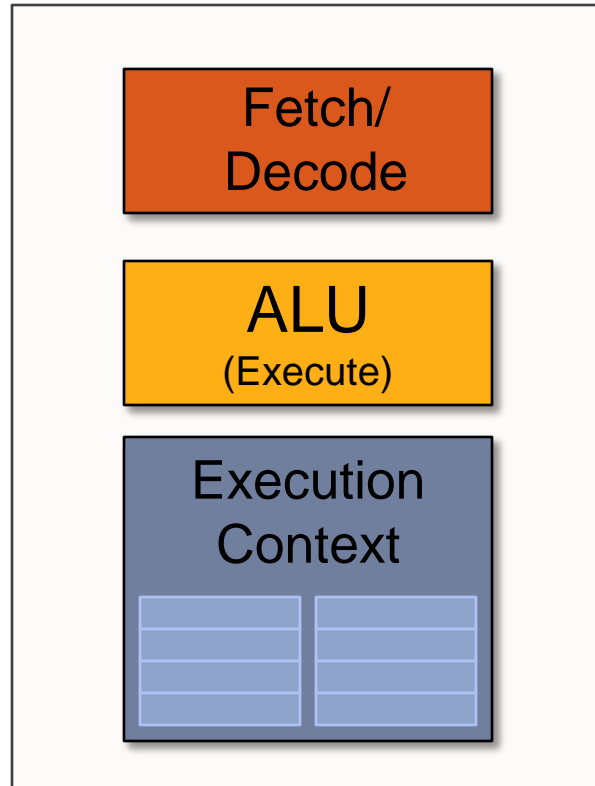
# Execute shader



```
<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)
```

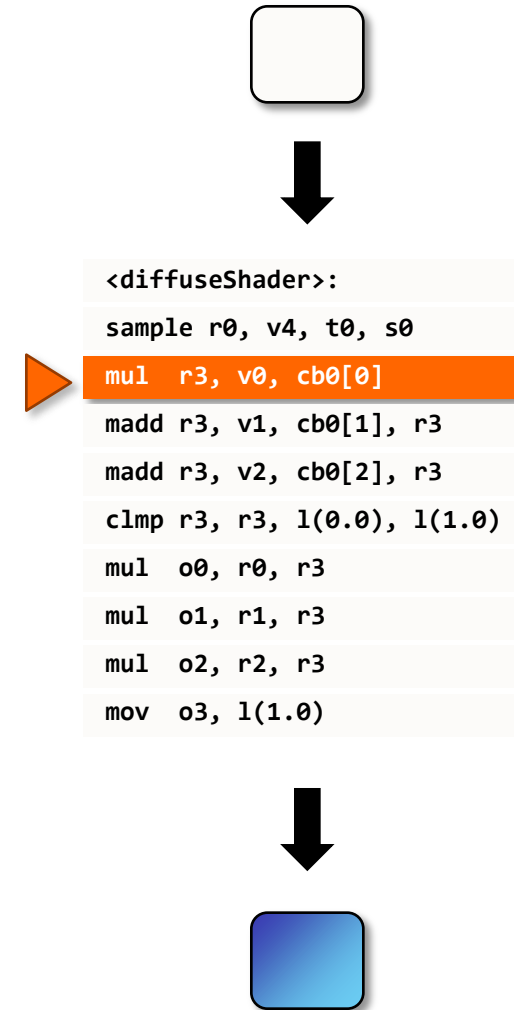
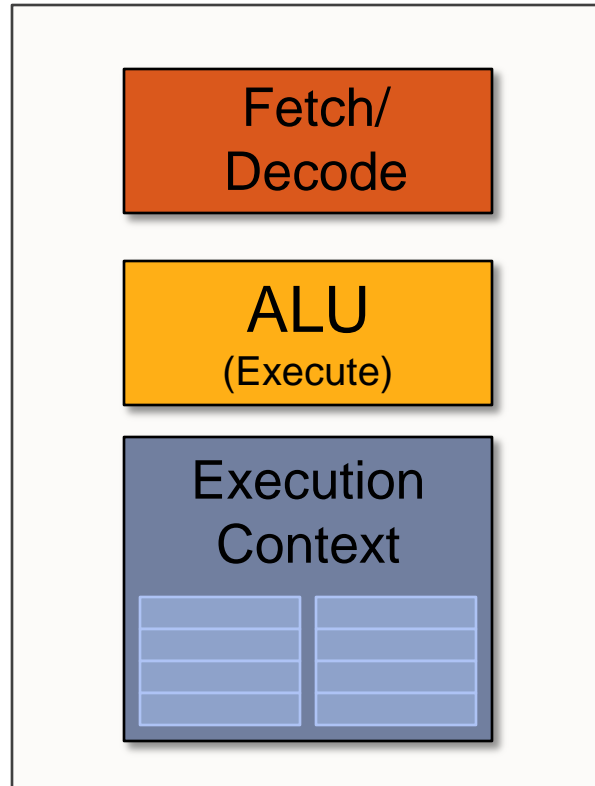


# Execute shader

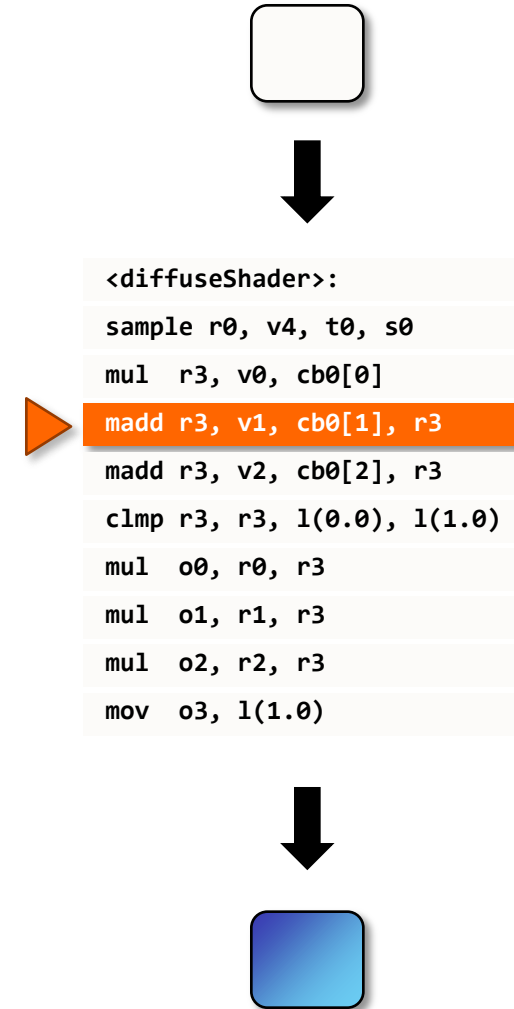
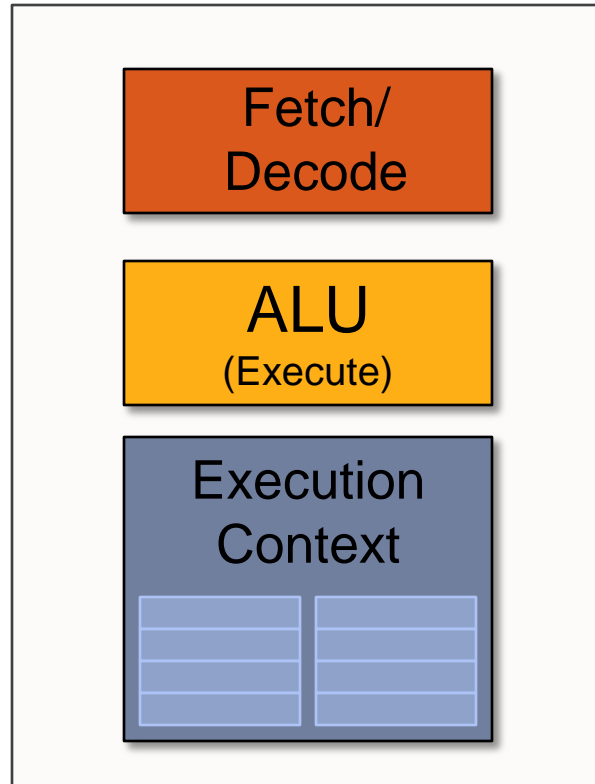




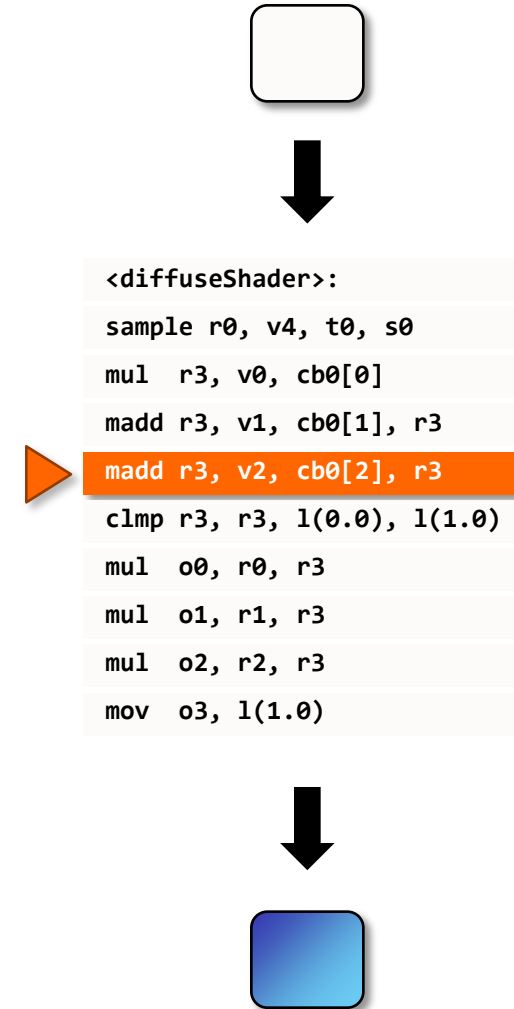
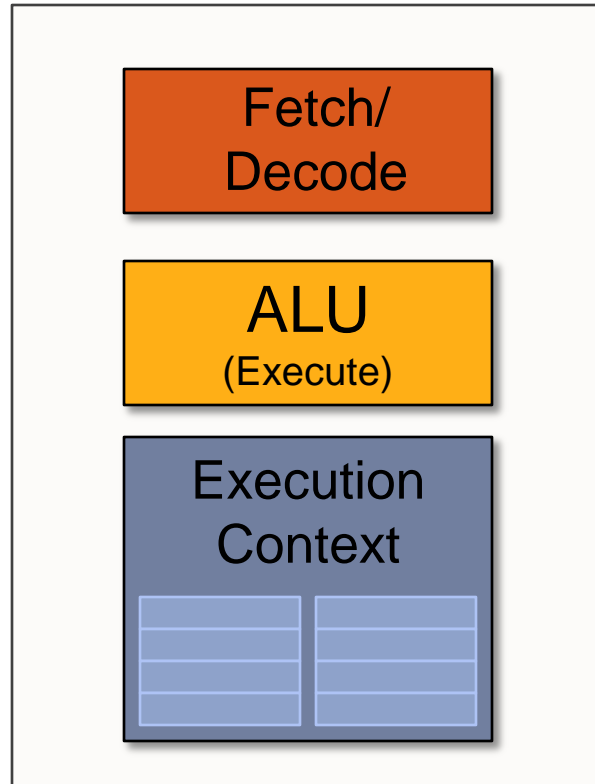
# Execute shader



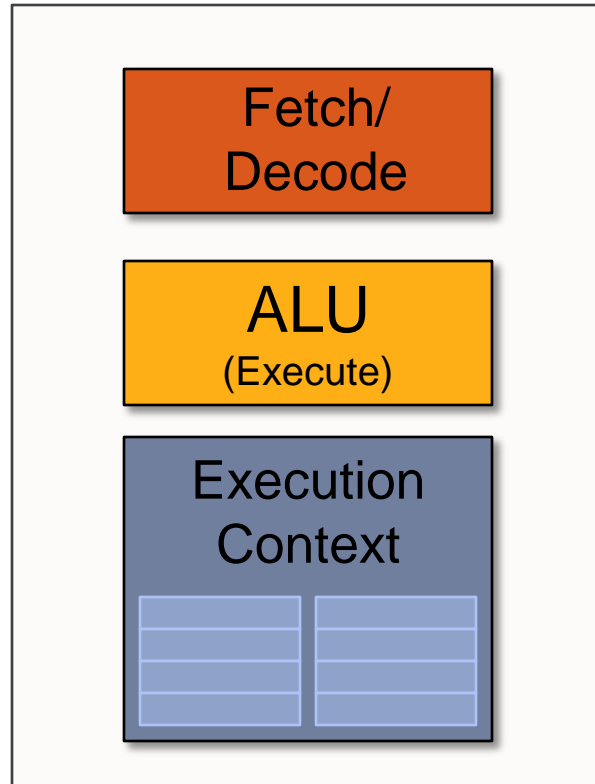
# Execute shader



# Execute shader



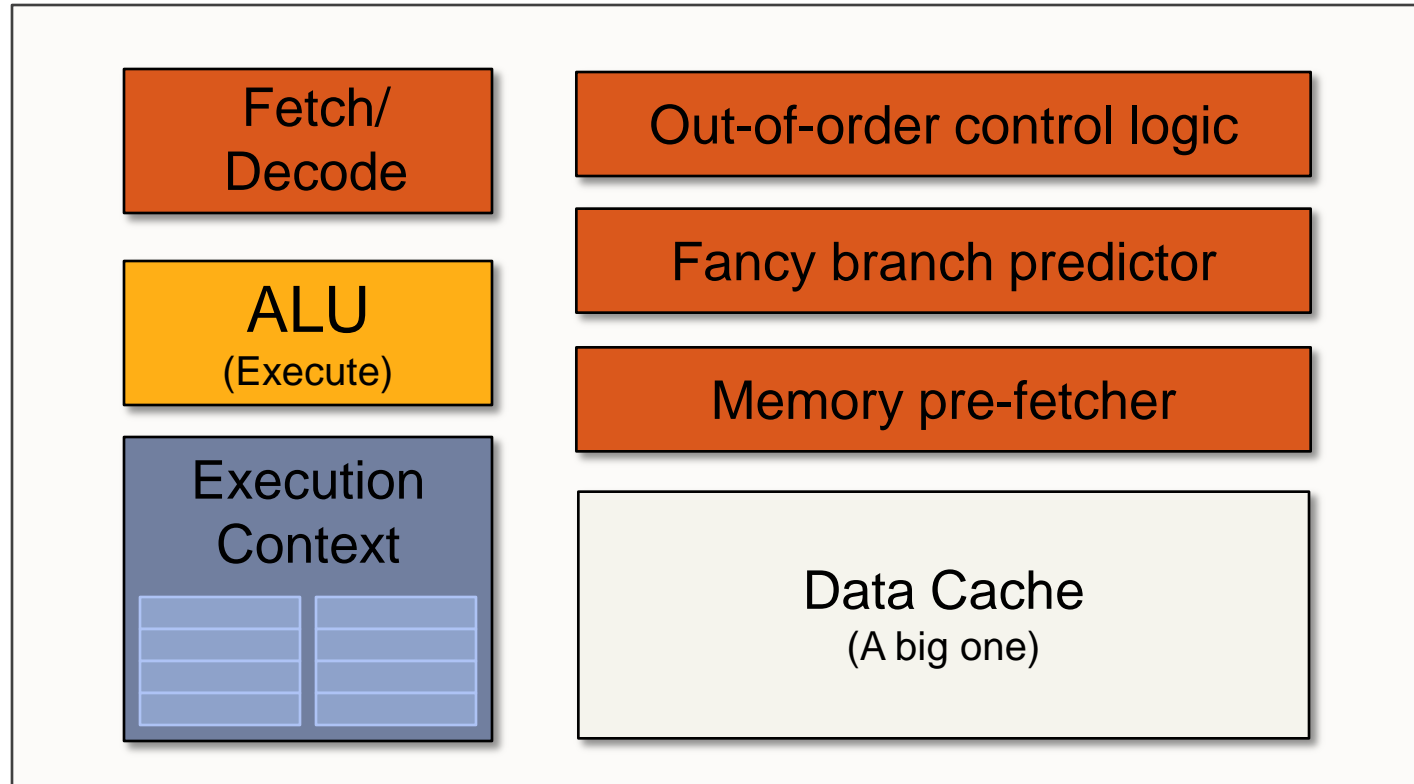
# Execute shader



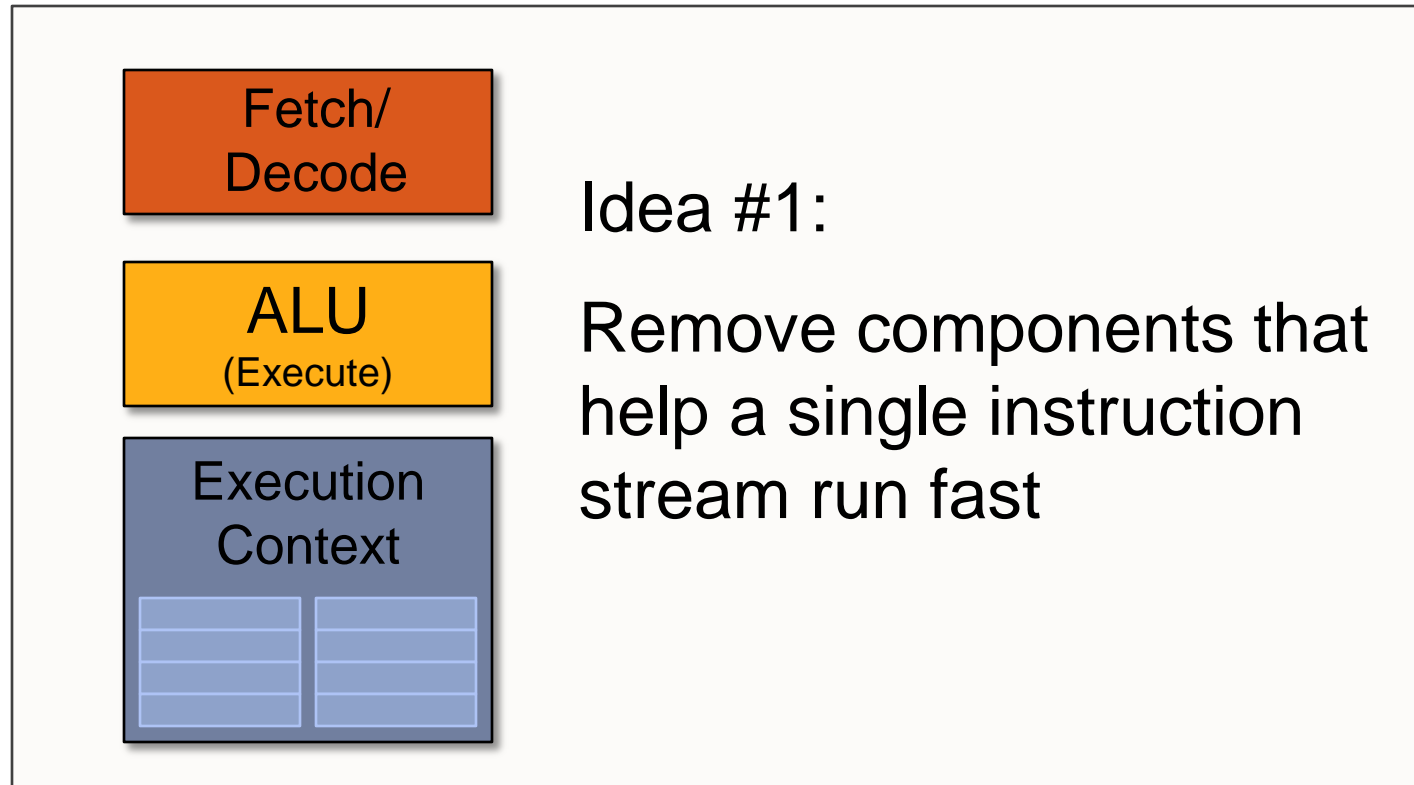
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```



# CPU-“style” cores

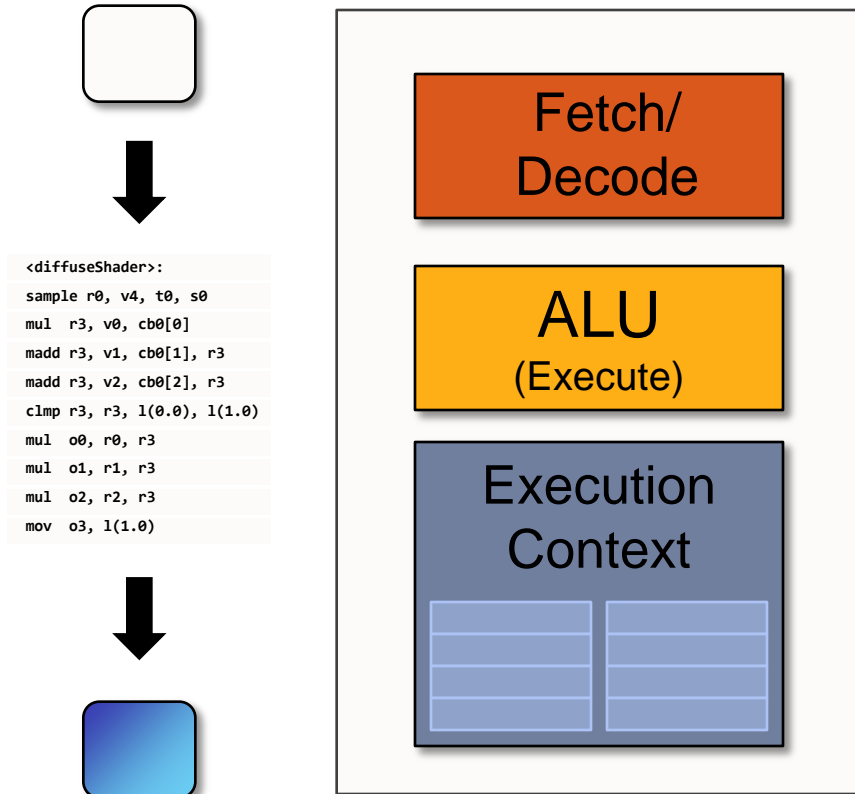


# Slimming down

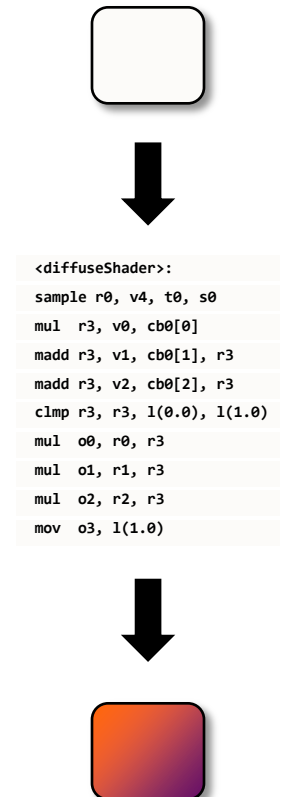


# Two cores (two fragments in parallel)

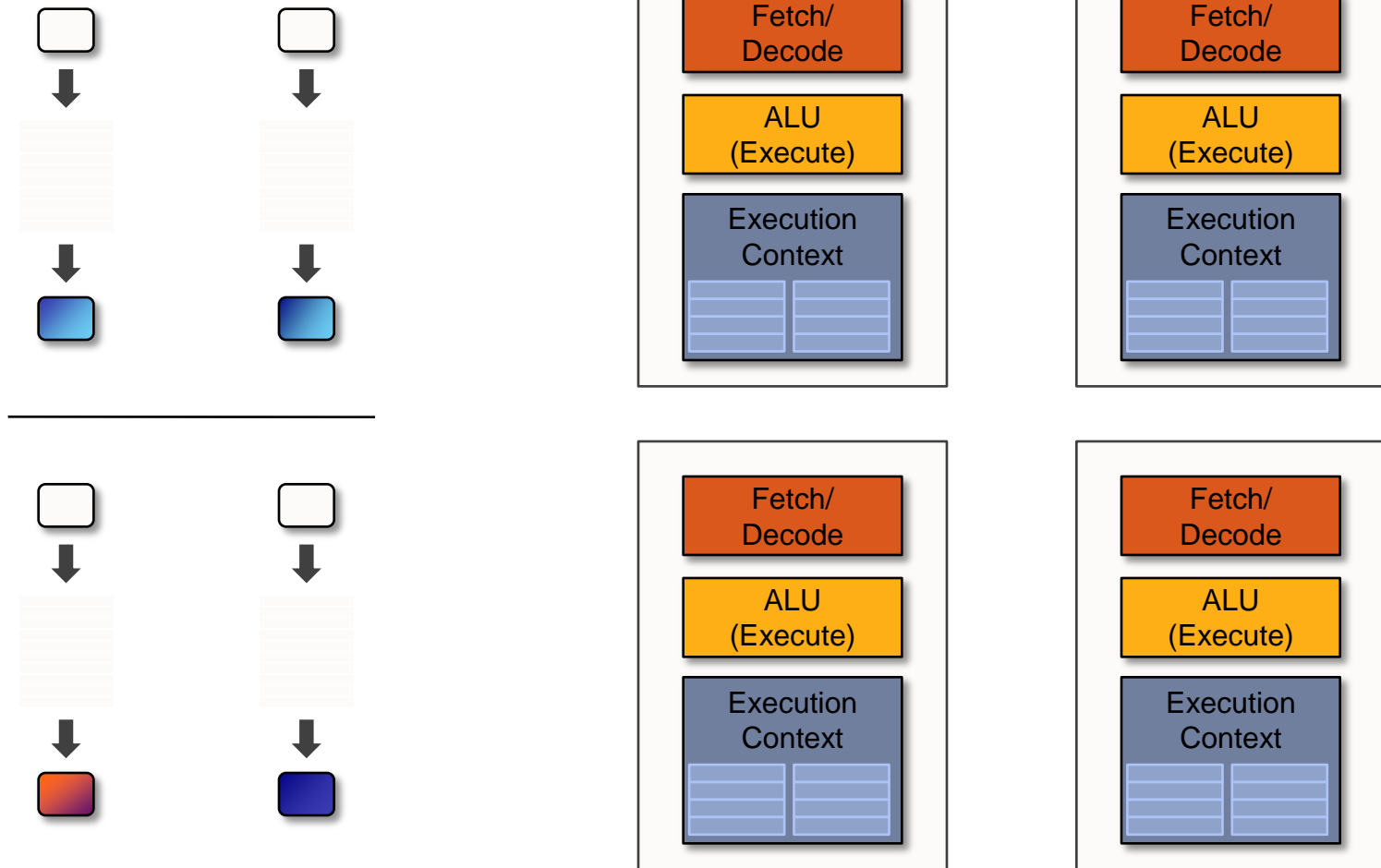
fragment 1



fragment 2



# Four cores (four fragments in parallel)





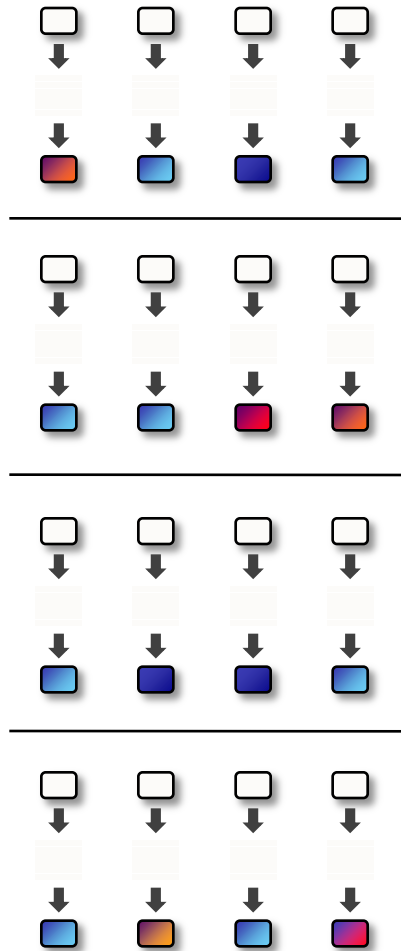
# Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams



# Instruction stream coherence

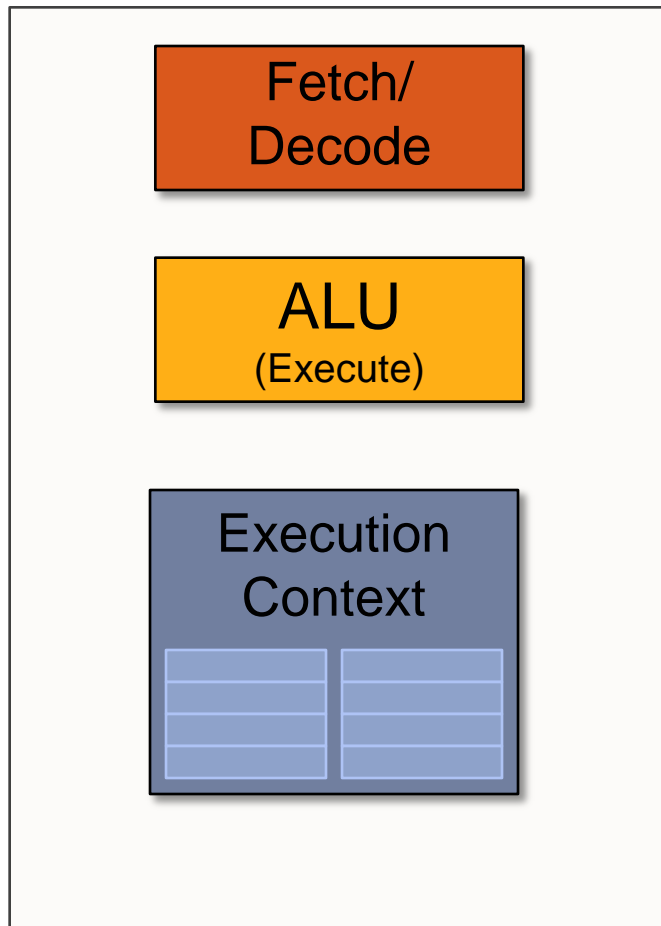


But... many fragments should be able to share an instruction stream!

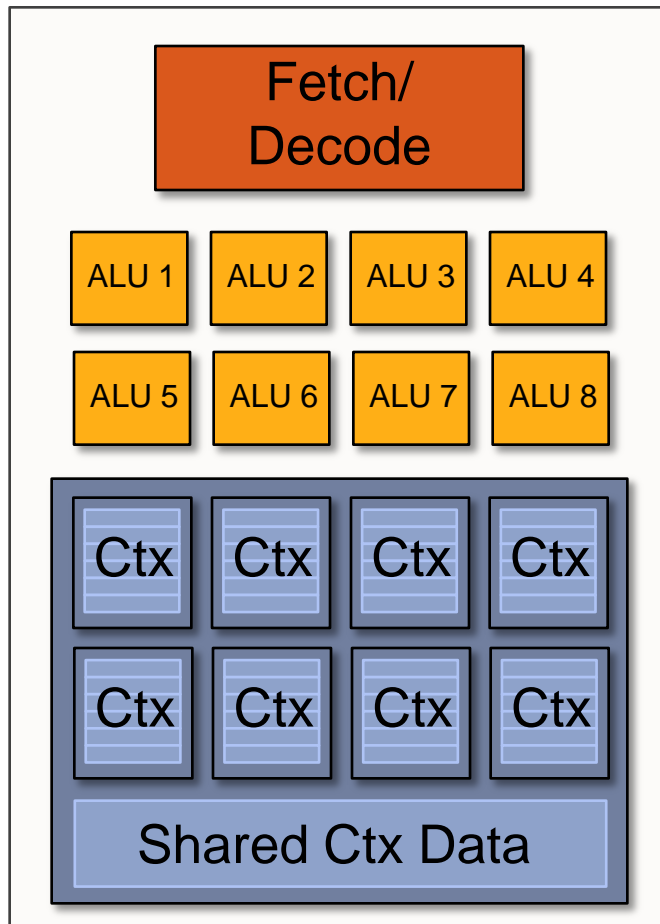
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```



# Recall: simple processing core



# Add ALUs



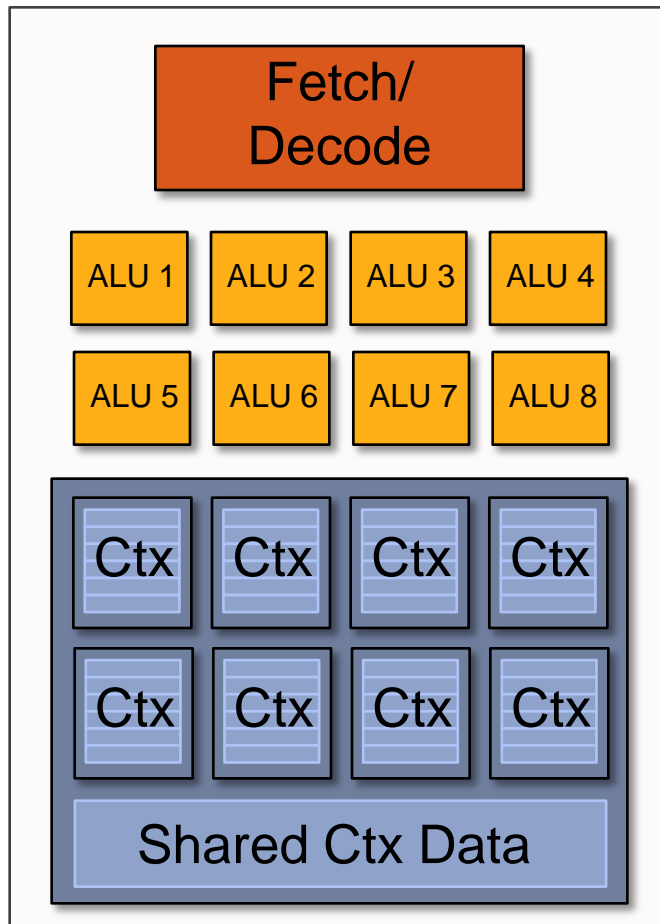
Idea #2:

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

## SIMD processing



# Modifying the shader



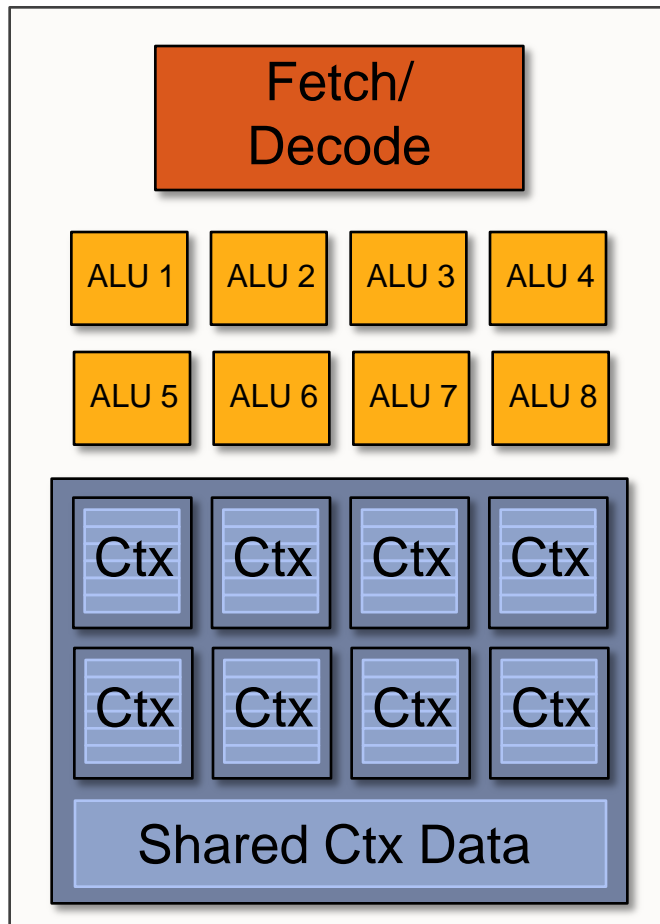
```

<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



```

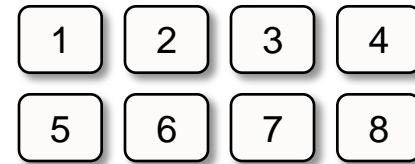
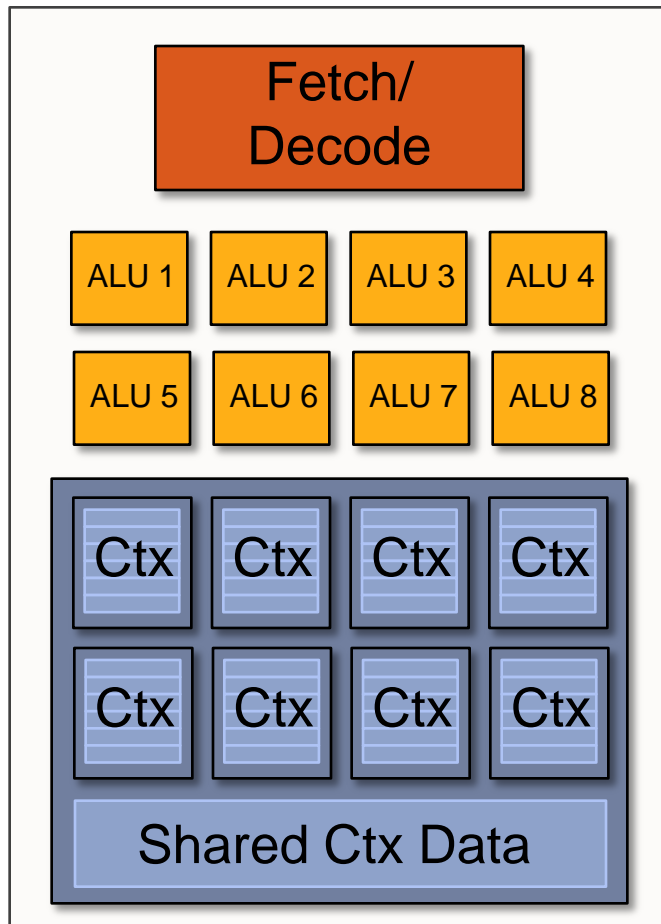
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_cImp vec_r3, vec_r3, 1(0.0), 1(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, 1(1.0)
  
```

New compiled shader:

Processes 8 fragments  
using vector ops on vector  
registers



# Modifying the shader



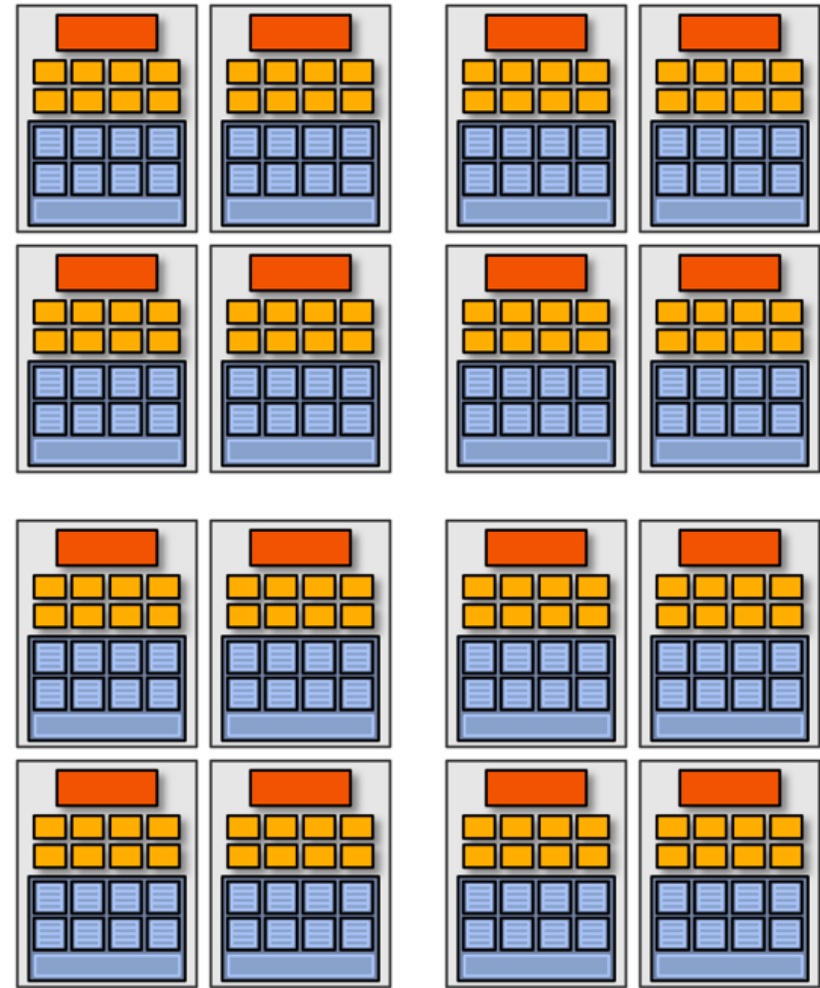
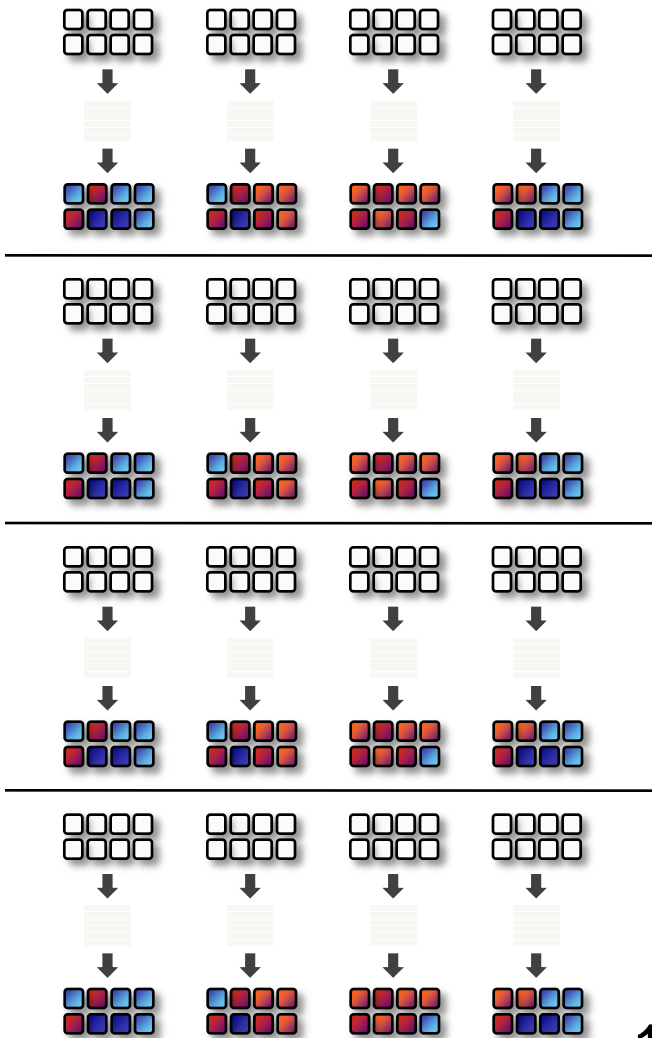
<VEC8\_diffuseShader>:

```

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)
  
```



# 128 fragments in parallel



16 cores = 128 ALUs  
 = 16 simultaneous instruction streams

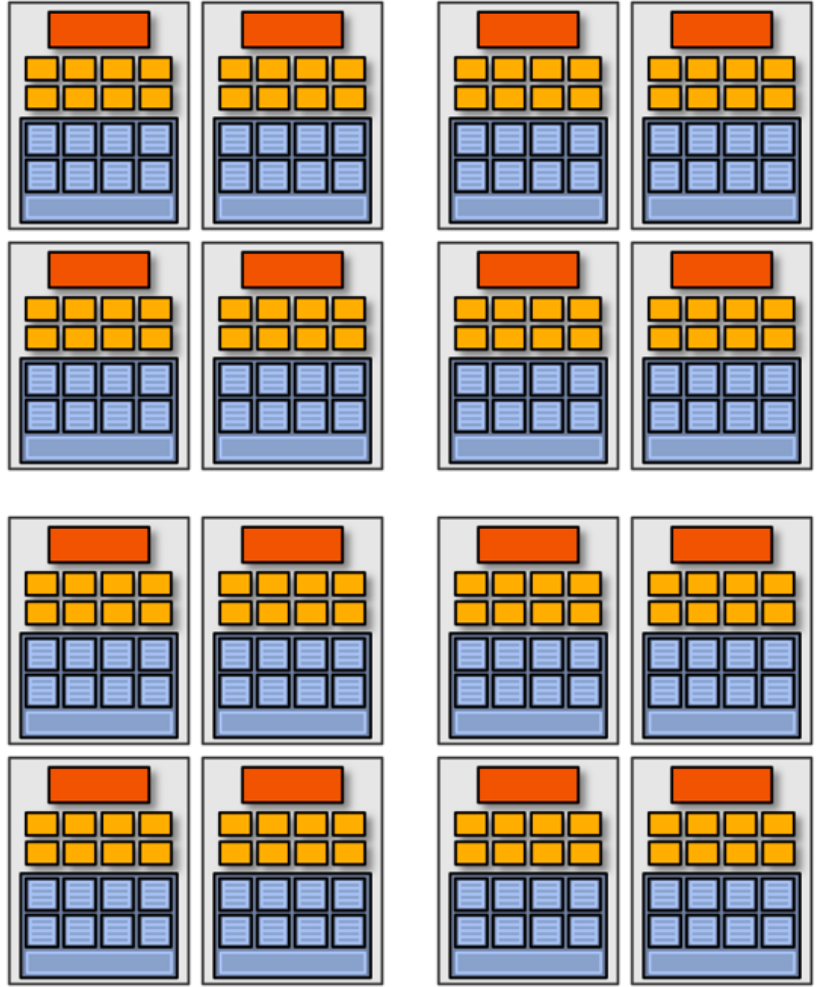
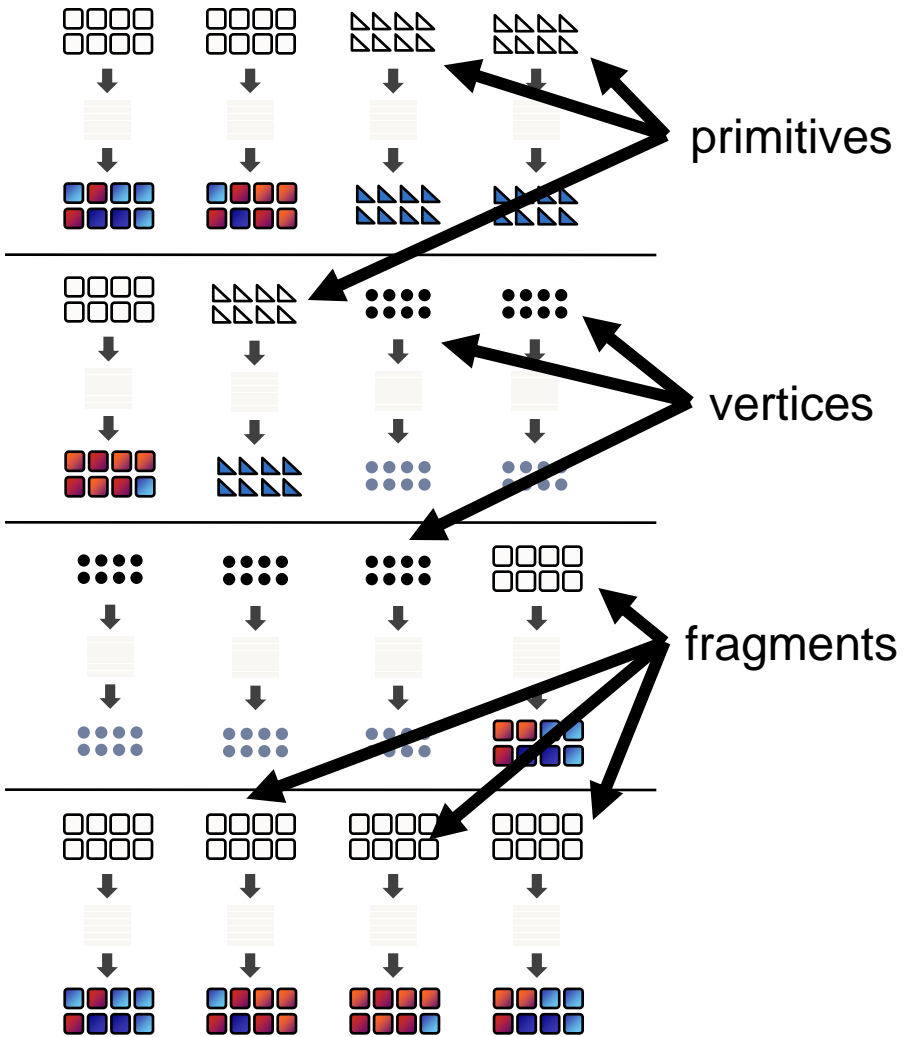




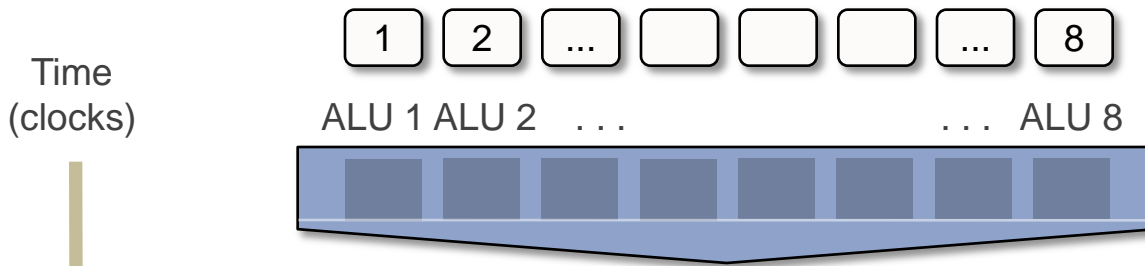
# 128 [

vertices / fragments  
primitives  
CUDA threads  
OpenCL work items  
compute shader threads

# ] in parallel



# But what about branches?



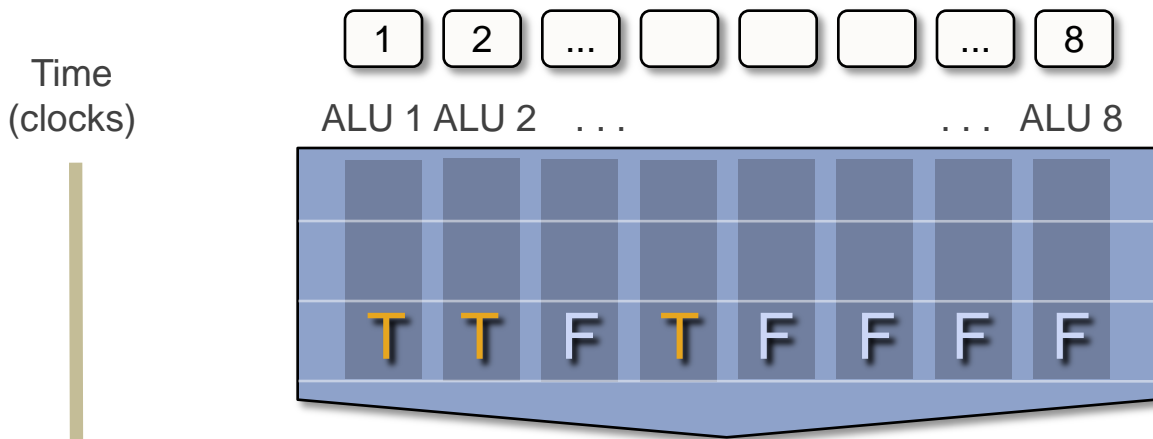
<unconditional  
shader code>

```
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

<resume unconditional  
shader code>



# But what about branches?



<unconditional  
shader code>

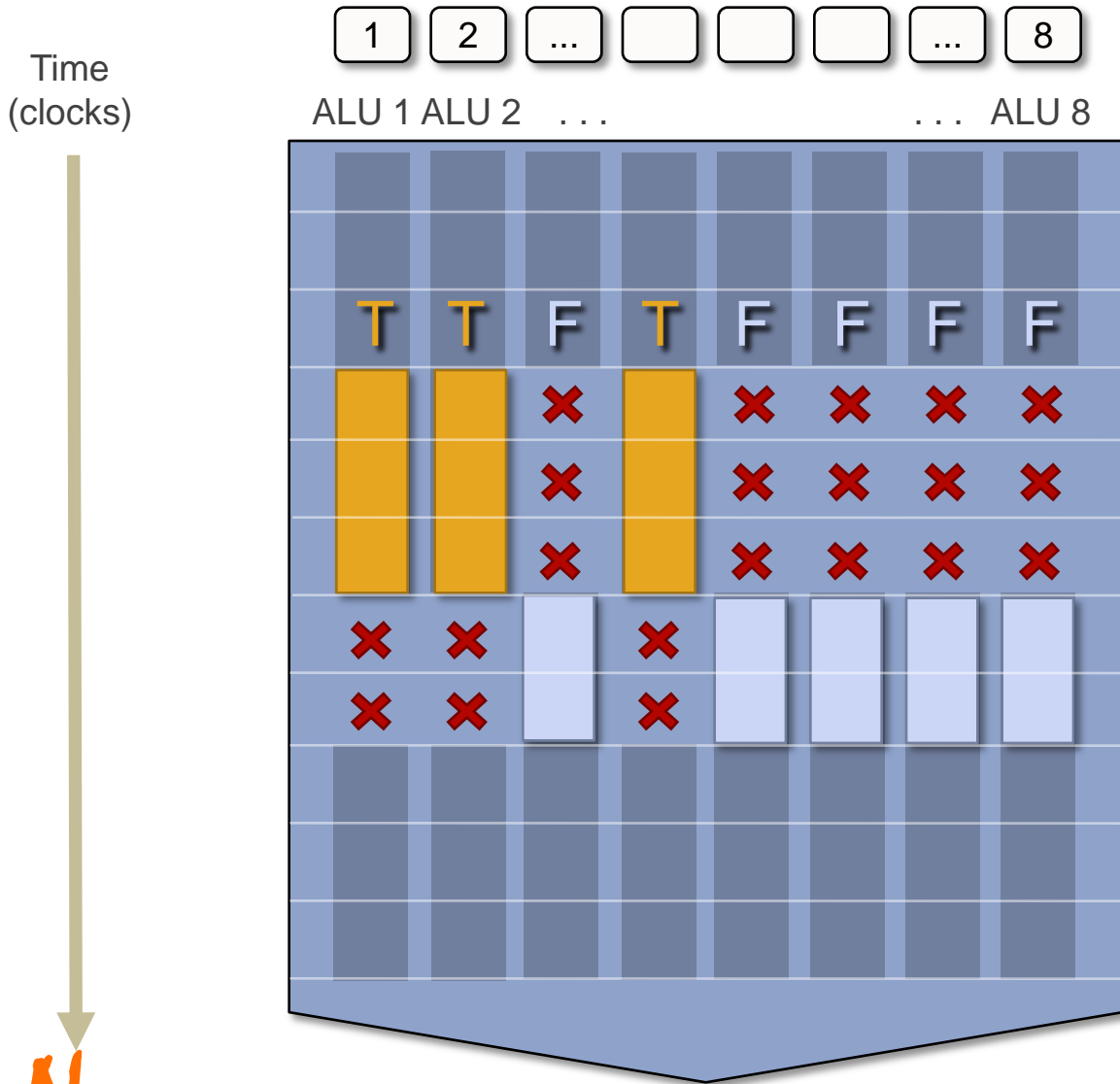
```
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

<resume unconditional  
shader code>





# But what about branches?



```

<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

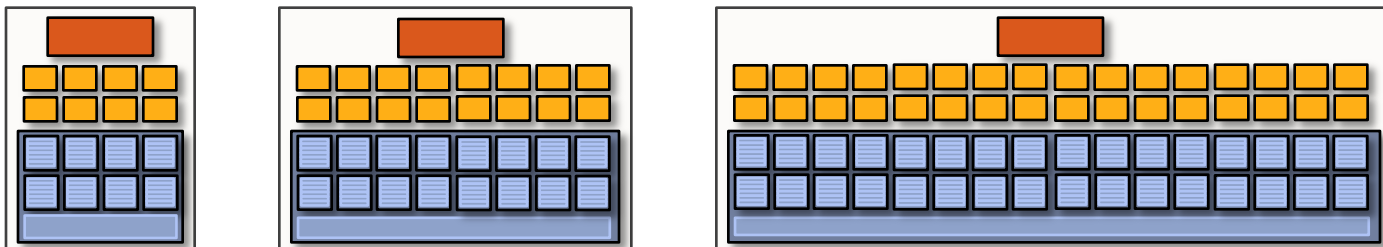
<resume unconditional
shader code>
    
```



# 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.

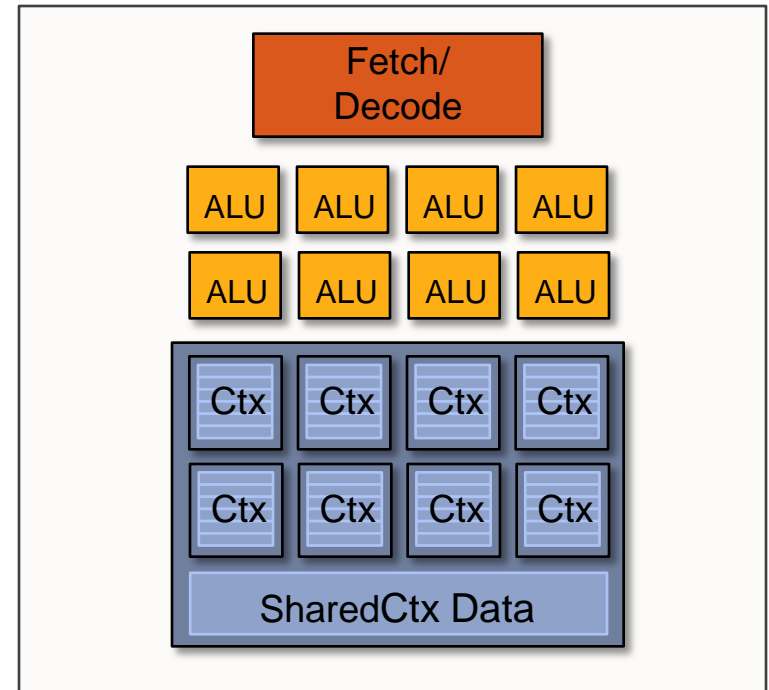
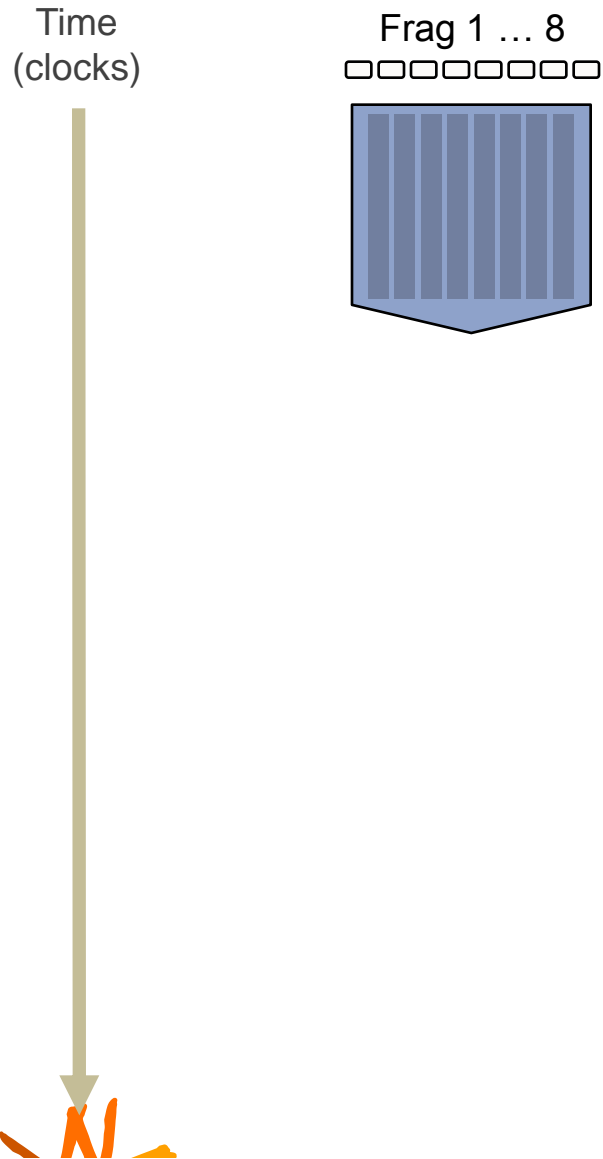
## Idea #3:

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



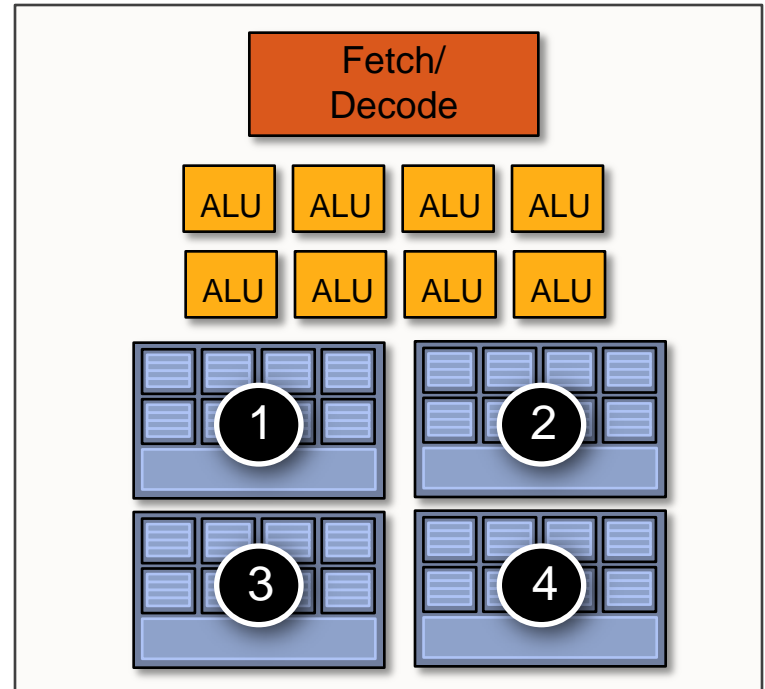
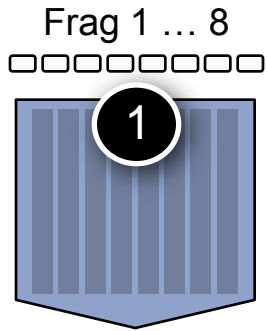


# Hiding shader stalls

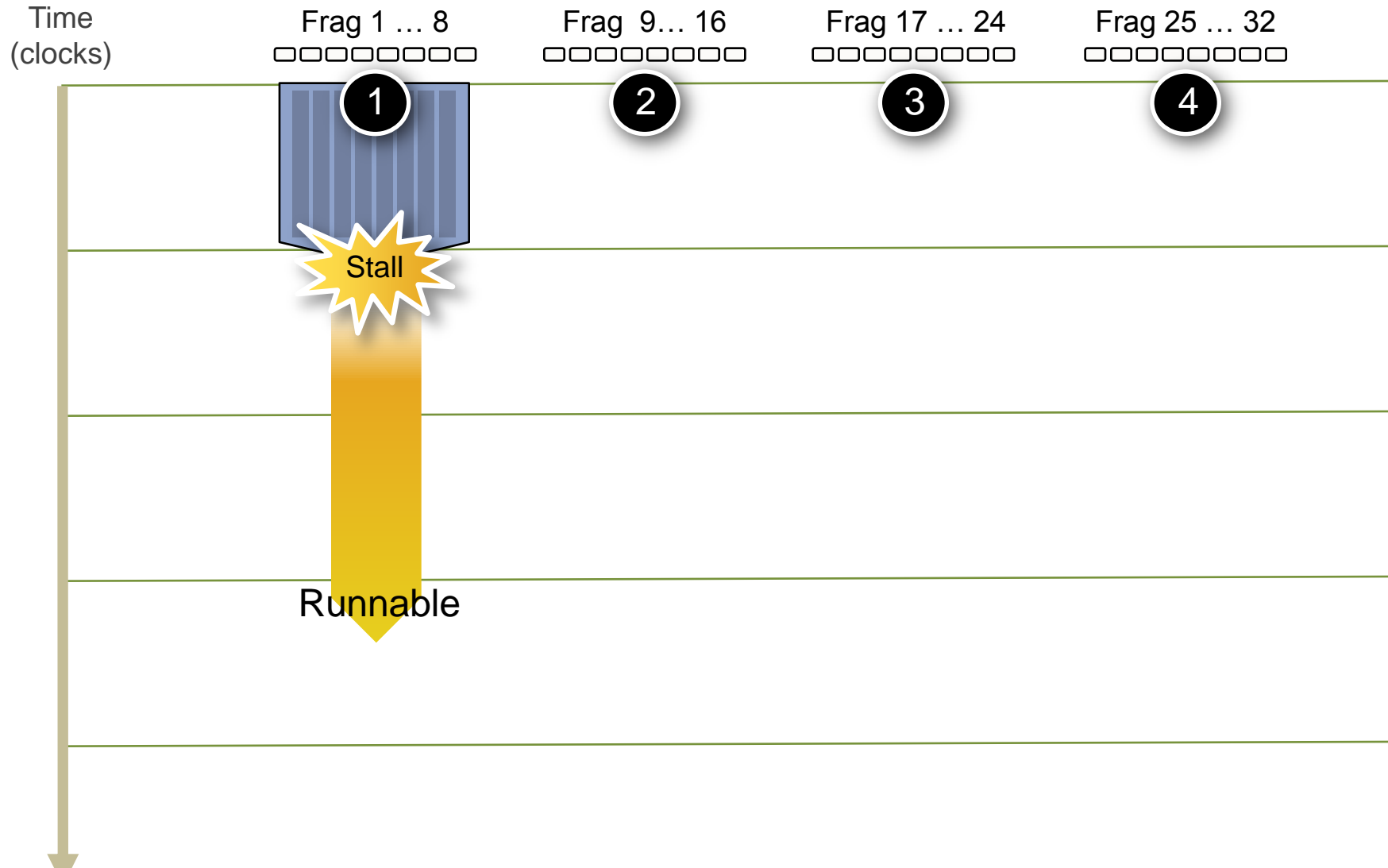


# Hiding shader stalls

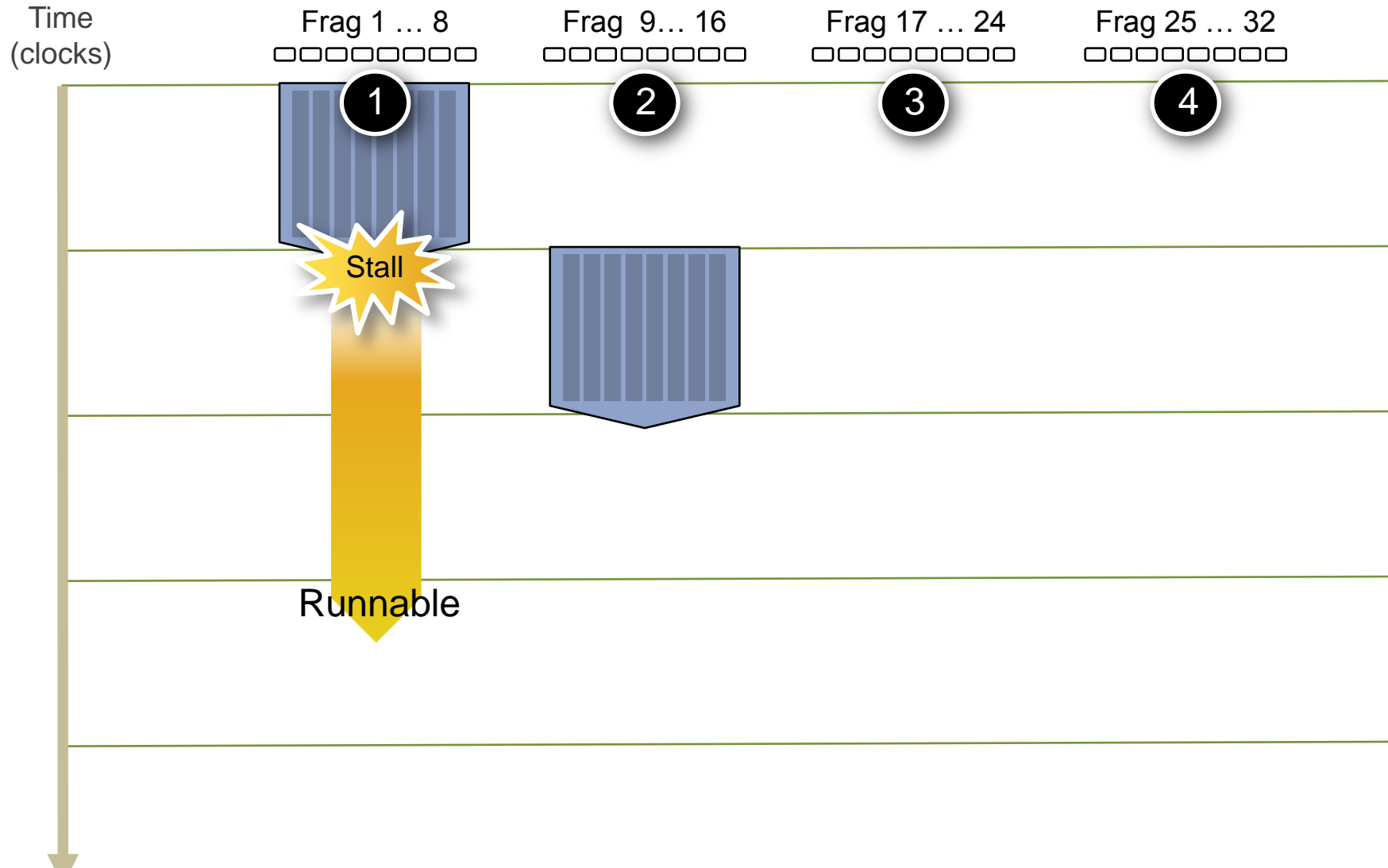
Time (clocks)



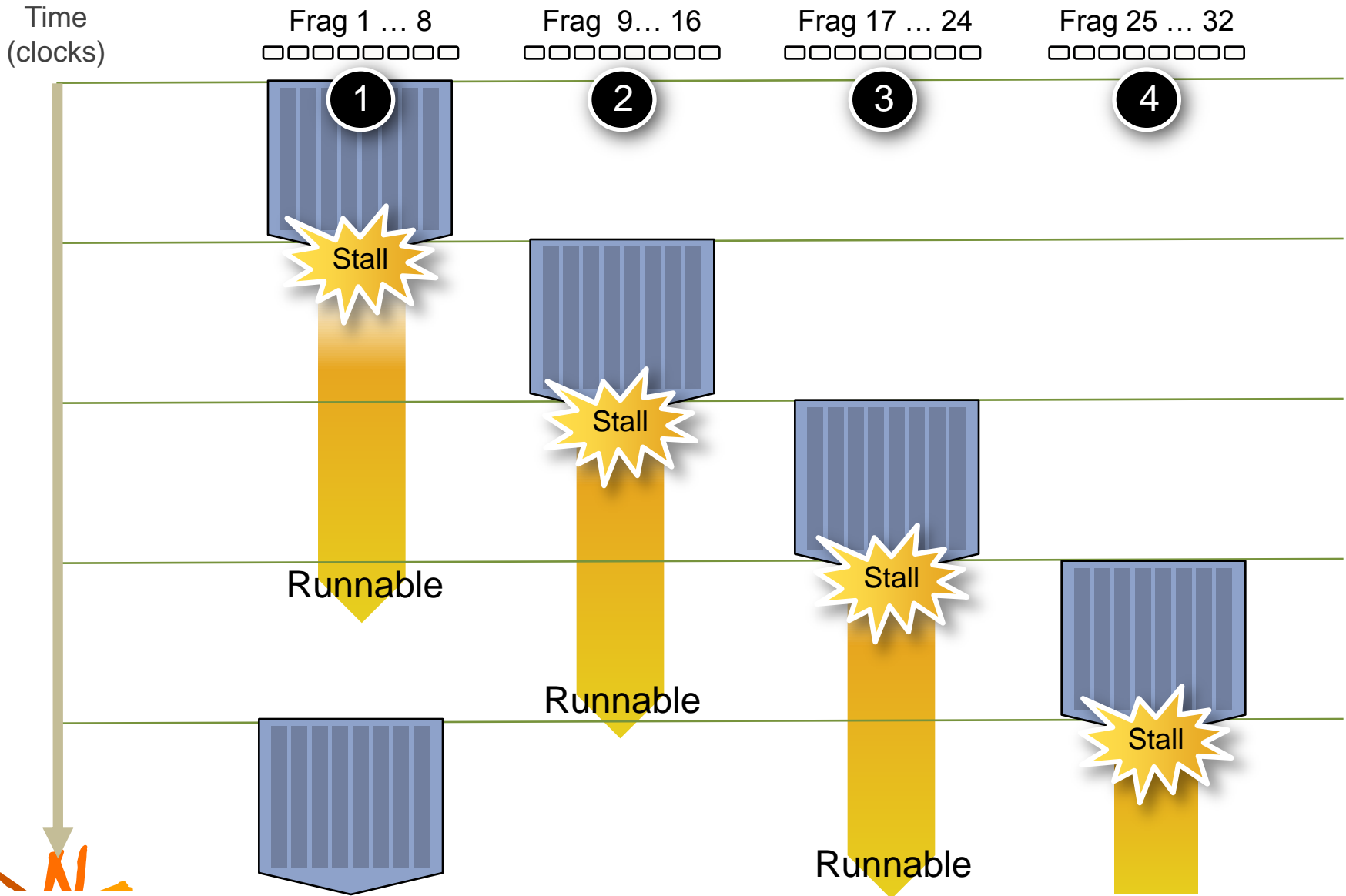
# Hiding shader stalls



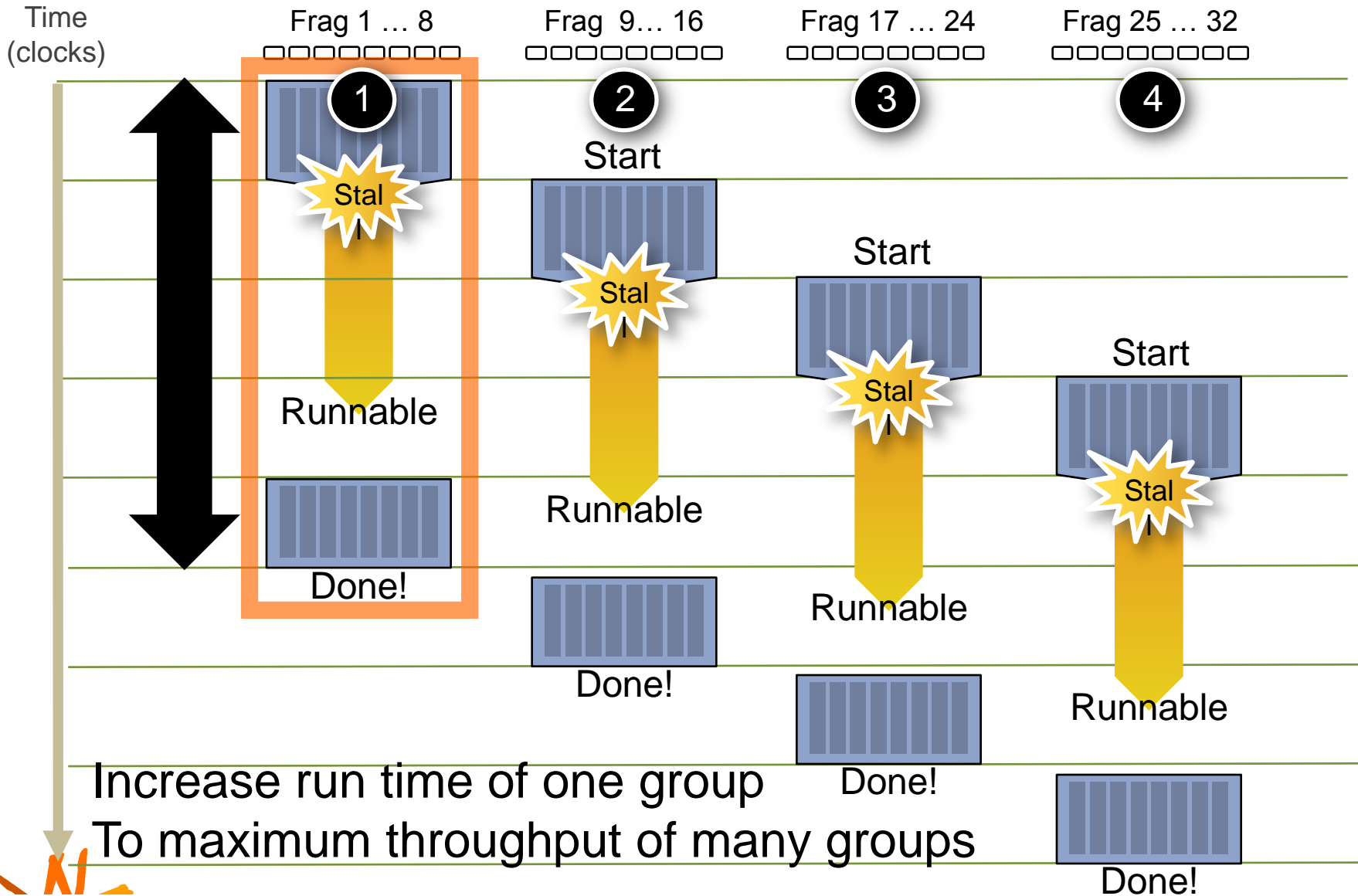
# Hiding shader stalls



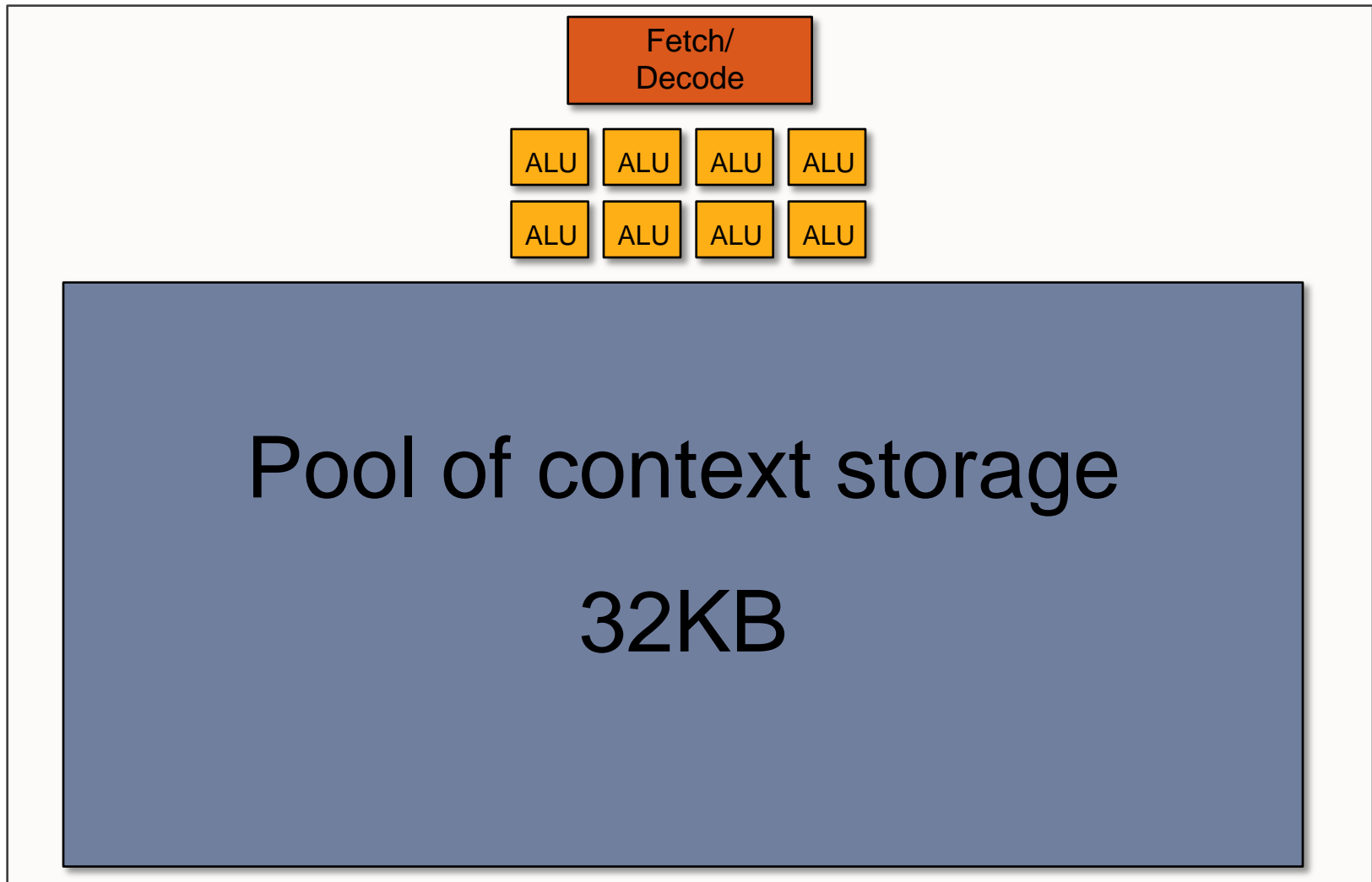
# Hiding shader stalls



# Throughput!

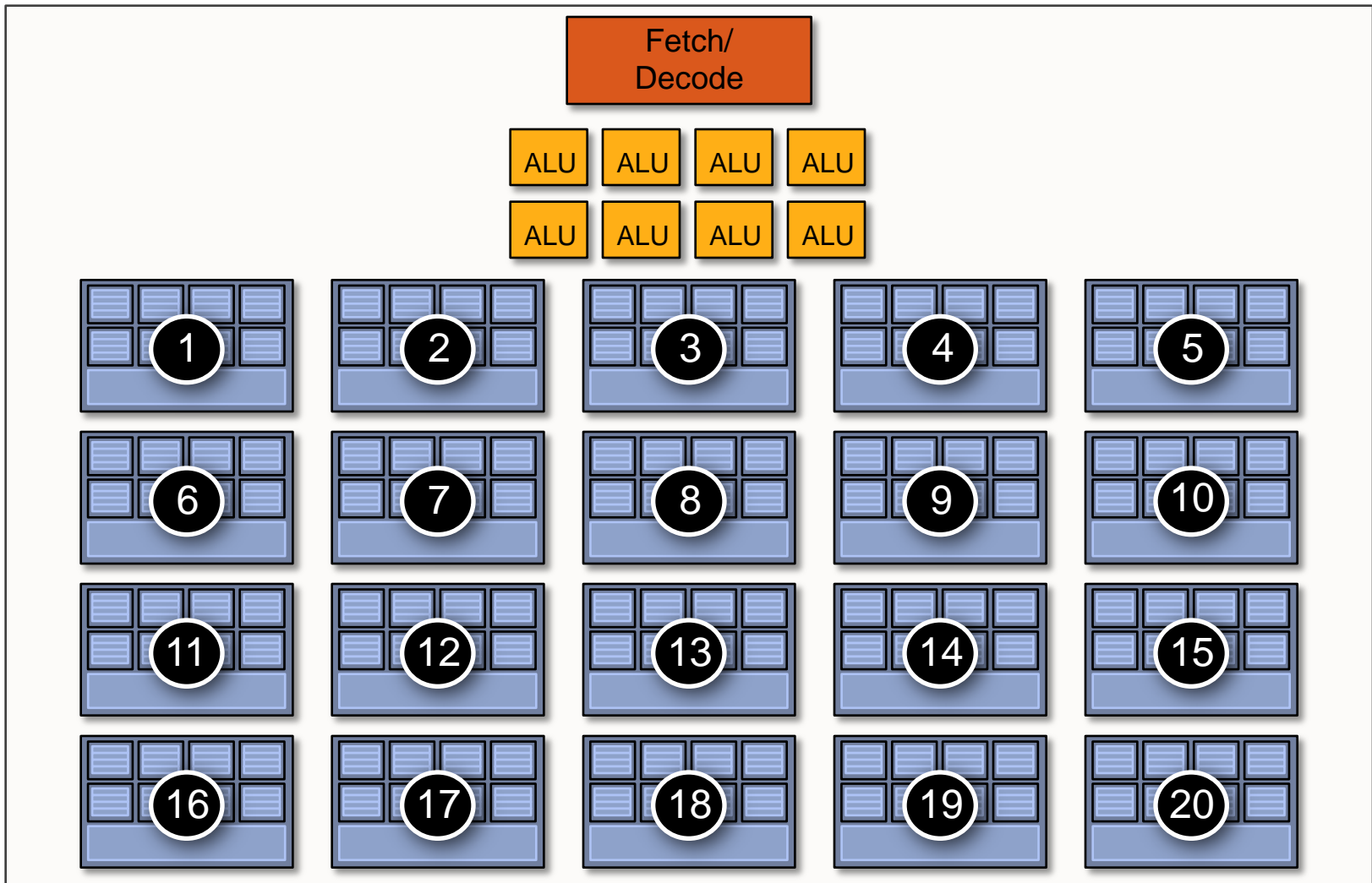


# Storing contexts



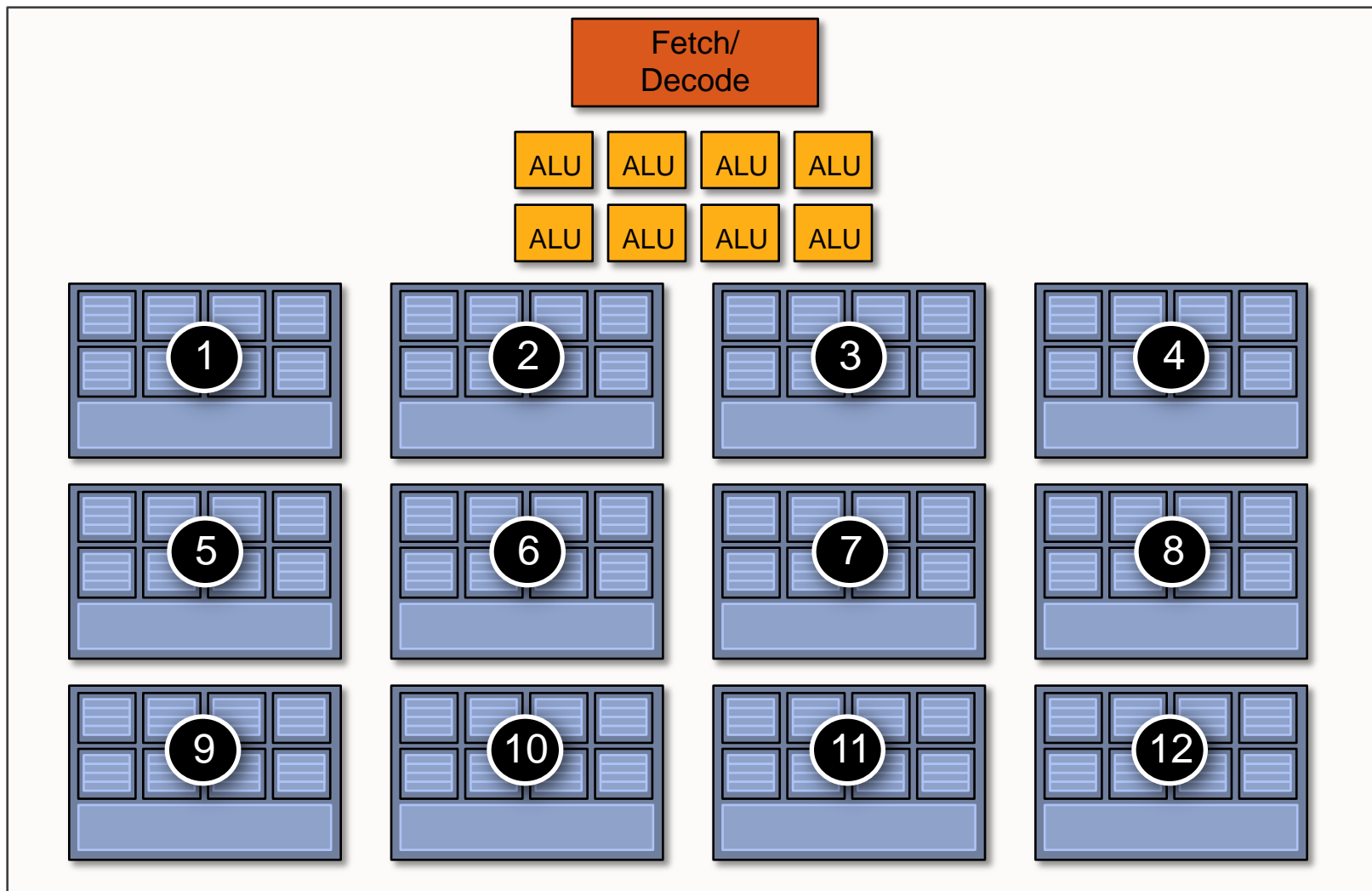
# Twenty small contexts

(maximal latency hiding ability)



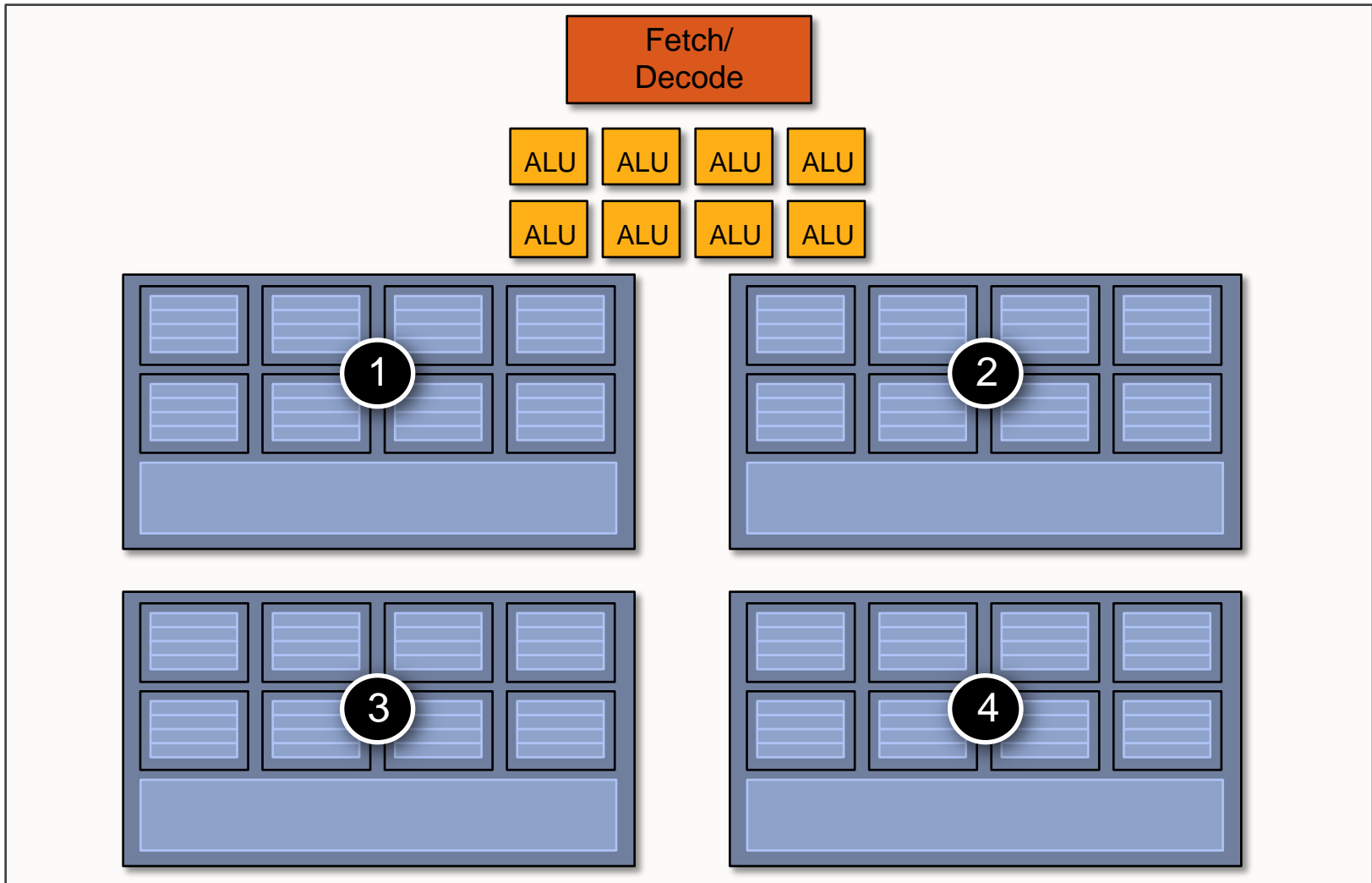


# Twelve medium contexts



# Four large contexts

(low latency hiding ability)



# Summary: three key ideas for GPU architecture

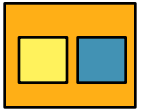
1. Use many “slimmed down cores” to run in parallel
2. Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
  - Option 1: Explicit SIMD vector instructions
  - Option 2: Implicit sharing managed by hardware
3. Avoid latency stalls by interleaving execution of many groups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



# GPU block diagram key



= single “physical” instruction stream fetch/decode  
(functional unit control)



= SIMD programmable functional unit (FU), control shared with other functional units. This functional unit may contain multiple 32-bit “ALUs”



= 32-bit mul-add unit



= 32-bit multiply unit



= execution context storage



= fixed function unit

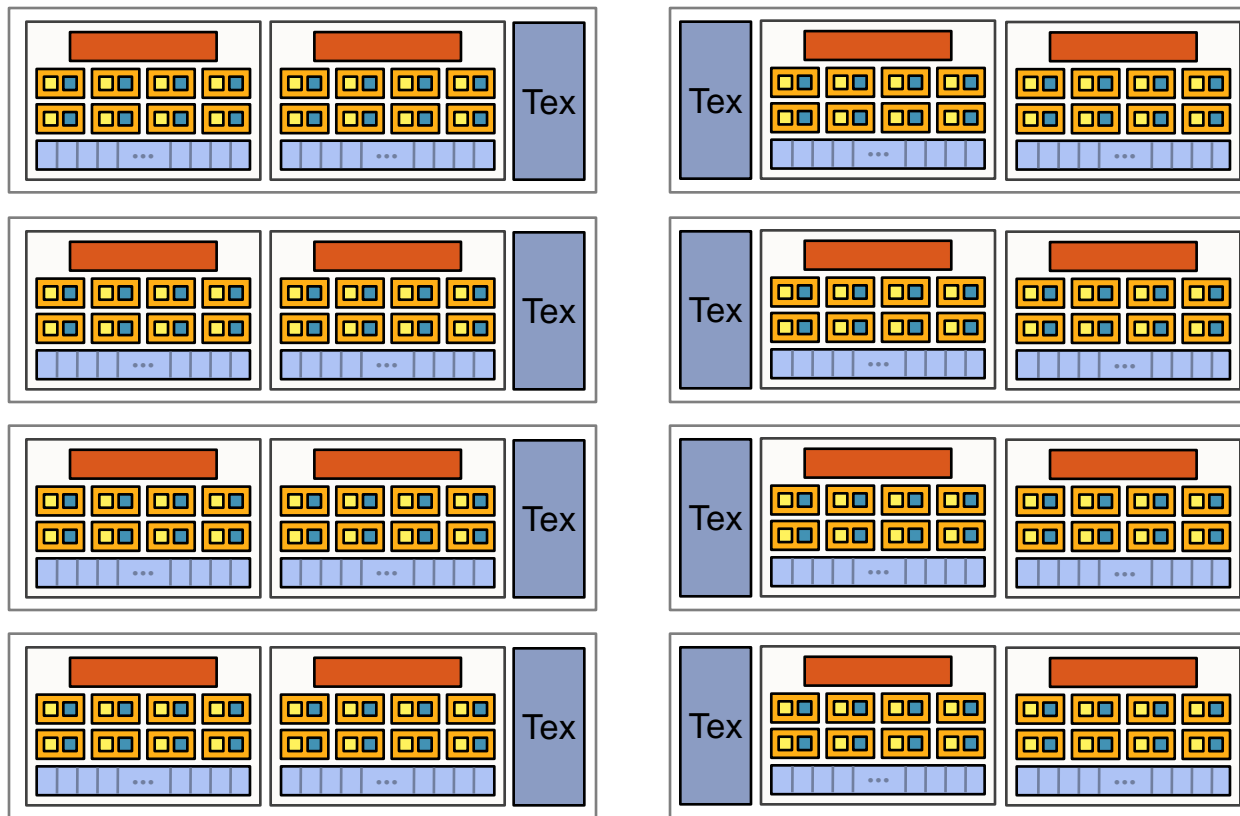


# NVIDIA GeForce 8800/9800

- NVIDIA-speak:
  - 128 stream processors
  - “SIMT execution” (automatic HW-managed sharing of instruction stream)
- Generic speak:
  - 16 processing cores
  - 8 SIMD functional units per core
  - 1 mul-add (2 flops) + 1 mul per functional units (3 flops/clock)
  - Best case: 128 mul-adds + 128 muls per clock
  - 1.3 GHz clock
  - $16 * 8 * (2 + 1) * 1.2 = 460$  GFLOPS
- Mapping data-parallelism to chip:
  - Instruction stream shared across 32 fragments (16 for vertices)
  - 8 fragments run on 8 SIMD functional units in one clock
  - Instruction repeated for 4 clocks (2 clocks for vertices)



# NVIDIA GeForce 8800/9800



Zcull/Clip/Rast

Output Blend

Work Distributor

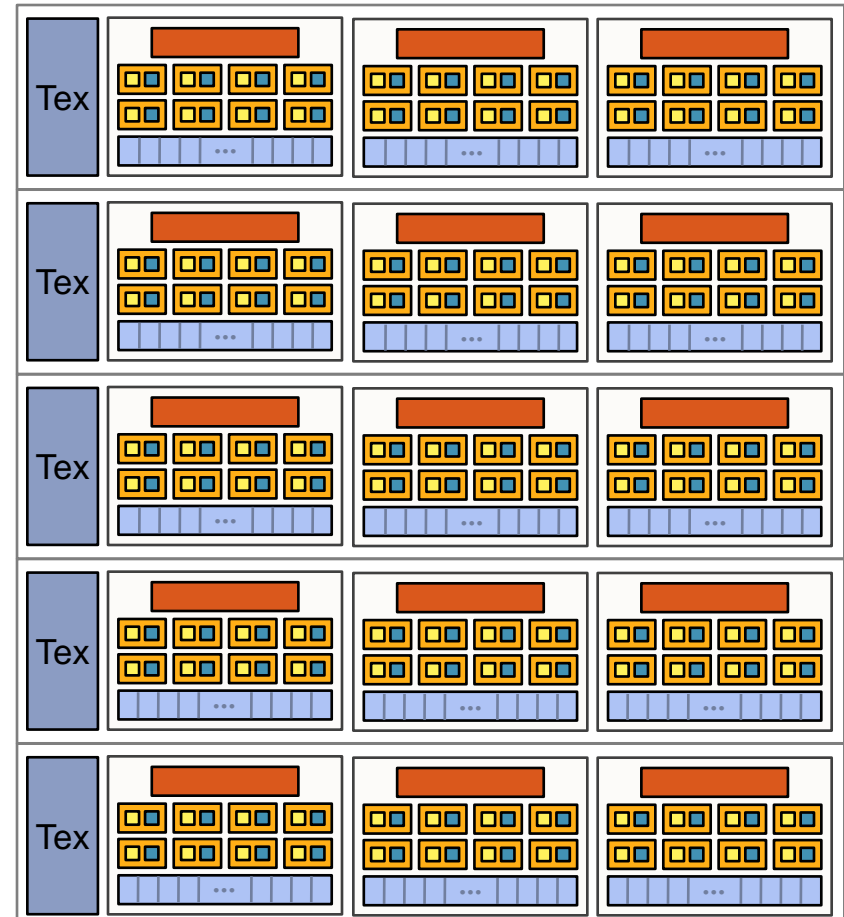
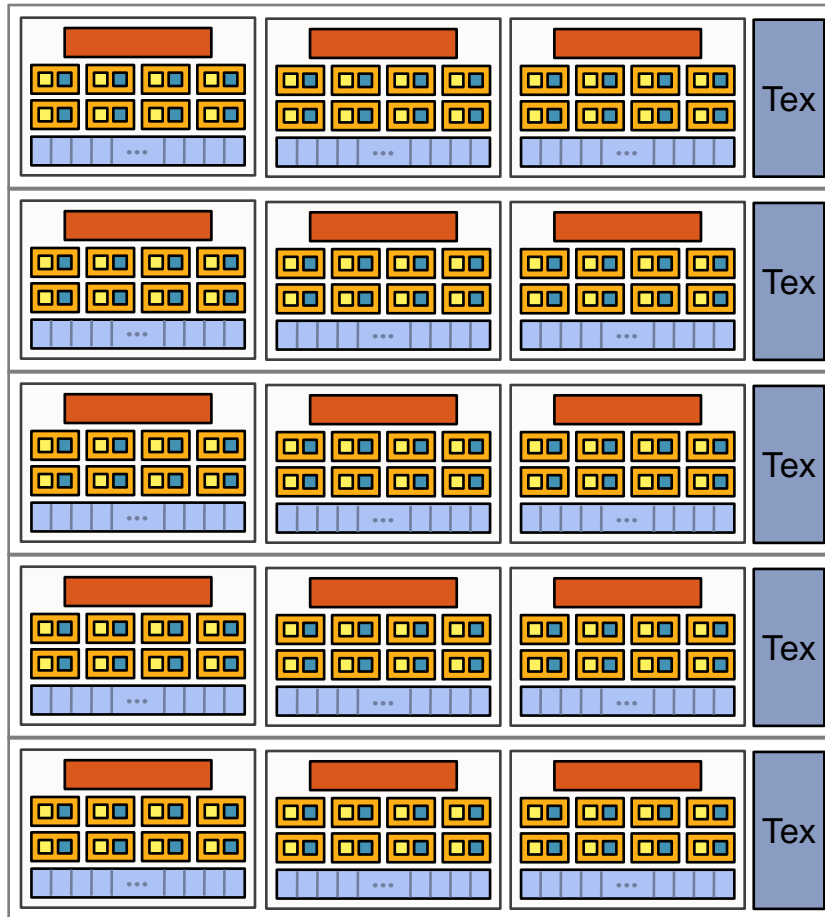


# NVIDIA GeForce GTX 280

- NVIDIA-speak:
  - 240 stream processors
  - “SIMT execution” (automatic HW-managed sharing of instruction stream)
- Generic speak:
  - 30 processing cores
  - 8 SIMD functional units per core
  - 1 mul-add (2 flops) + 1 mul per functional units (3 flops/clock)
  - Best case: 240 mul-adds + 240 muls per clock
  - 1.3 GHz clock
  - $30 * 8 * (2 + 1) * 1.3 = 933$  GFLOPS
- Mapping data-parallelism to chip:
  - Instruction stream shared across 32 “threads”
  - 16 threads run on 8 SIMD functional units in one clock
  - Instruction repeated for 4 clocks (2 clocks for vertices)



# NVIDIA GeForce GTX 280



Zcull/Clip/Rast

Output Blend

Work Distributor





# NVIDIA Fermi

- NVIDIA-speak:
  - 512 CUDA processors (formerly stream processors)
  - “SIMT execution” (automatic HW-managed sharing of instruction stream)
- Generic speak:
  - 26 processing cores
  - 32 SIMD functional units per core
  - 1 FPU (FMA) + 1 INT per functional units (2 flops/clock)
  - Best case: 512 FP mul-adds
  - 1.3 GHz clock
  - $16 * 32 * 2 * 1.5 = 1.5$  GFLOPS (more real than earlier)
- Mapping data-parallelism to chip:
  - Instruction stream shared across 32 threads
  - 32 fragments run on 16 SIMD functional units in one clock
  - Instruction repeated for 2 clocks ?

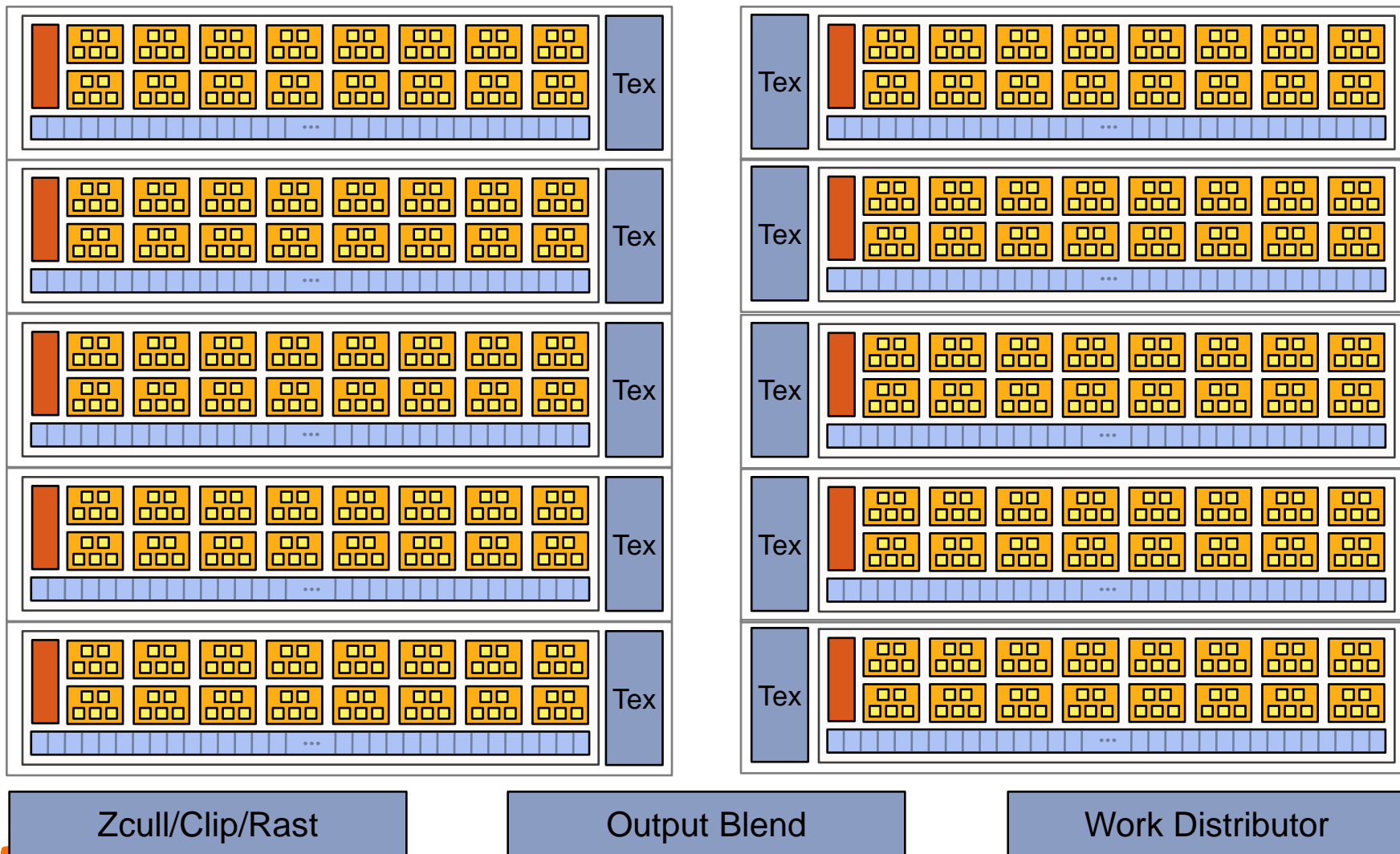


# ATI Radeon 4870

- AMD/ATI-speak:
  - 800 stream processors
  - Automatic HW-managed sharing of scalar instruction stream (like “SIMT”)
- Generic speak:
  - 10 processing cores
  - 16 SIMD functional units per core
  - 5 mul-adds per functional unit ( $5 * 2 = 10$  flops/clock)
  - Best case: 800 mul-adds per clock
  - 750 MHz clock
  - $10 * 16 * 5 * 2 * .75 = 1.2$  TFLOPS
- Mapping data-parallelism to chip:
  - Instruction stream shared across 64 fragments
  - 16 fragments run on 16 SIMD functional units in one clock
  - Instruction repeated for 4 consecutive clocks



# ATI Radeon 4870

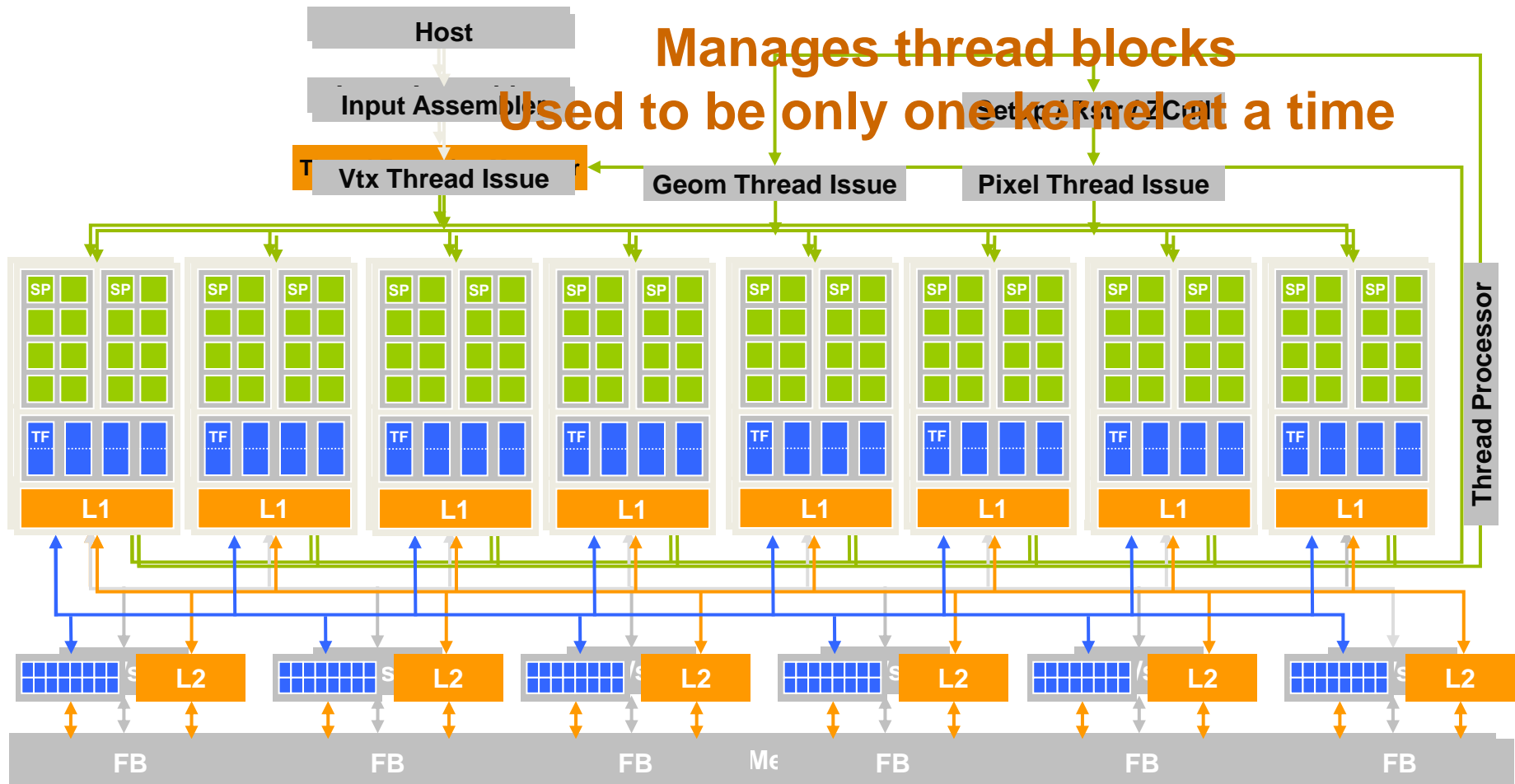


Additional information on “supplemental slides” and at  
<http://graphics.stanford.edu/~kayvonf/gblog>

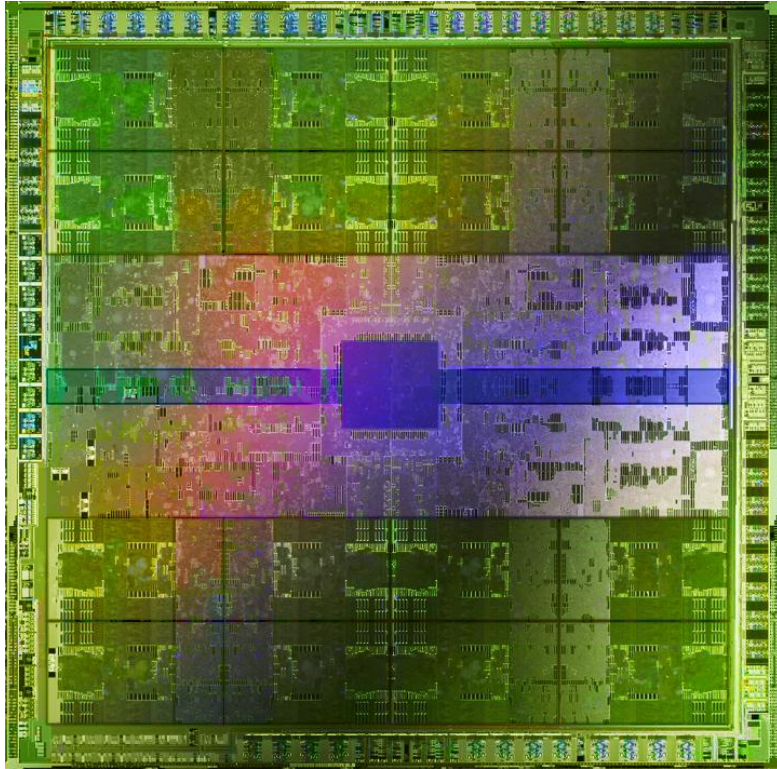


# Make the Compute Core The Focus of the Architecture

- The future of GPUs is programmable processing
- After build the architecture around the processor



# Next-Gen GPU Architecture: *Fermi*



- 3 billion transistors
- Over 2x the cores (512 total)
- ~2x the memory bandwidth
- **L1 and L2 caches**
- 8x the peak DP performance
- ECC
- C++
- **Announced Sept. 2009**



# Fermi Focus Areas

- Expand performance sweet spot of the GPU
  - Caching
  - Concurrent kernels
  - FP64
  - 512 cores
  - GDDR5 memory
- Bring more users, more applications to the GPU
  - C++
  - Visual Studio Integration
  - ECC

