

GPU Architecture

**Robert Strzodka (MPII),
Dominik Göttsche (TU Do), Dominik Behr (AMD)**

**PPAM 2009 - Conference on Parallel Processing and
Applied Mathematics
Wrocław, Poland, September 13-16, 2009**

www.gpgpu.org/ppam2009

Acknowledgements

- **Original author of this talk: Kayvon Fatahalian (Stanford University)**
 - <http://graphics.stanford.edu/~kayvonf/>
 - Special thanks for allowing us to present his talk
- ***"From shader code to a Teraflop: How shader cores work"***
 - SIGGRAPH Beyond Programmable Shading Courses 2008 and 2009
- **Corresponding paper:**
 - Kayvon Fatahalian and Mike Houston: *"A Closer Look at GPUs"*, Communications of the ACM 51(10), October 2008

Original acknowledgements

- **Mike Houston**
- **Jeremy Sugerman**
- **Kurt Akeley**
- **Pat Hanrahan**
- **(Stanford University)**

Goals and terminology

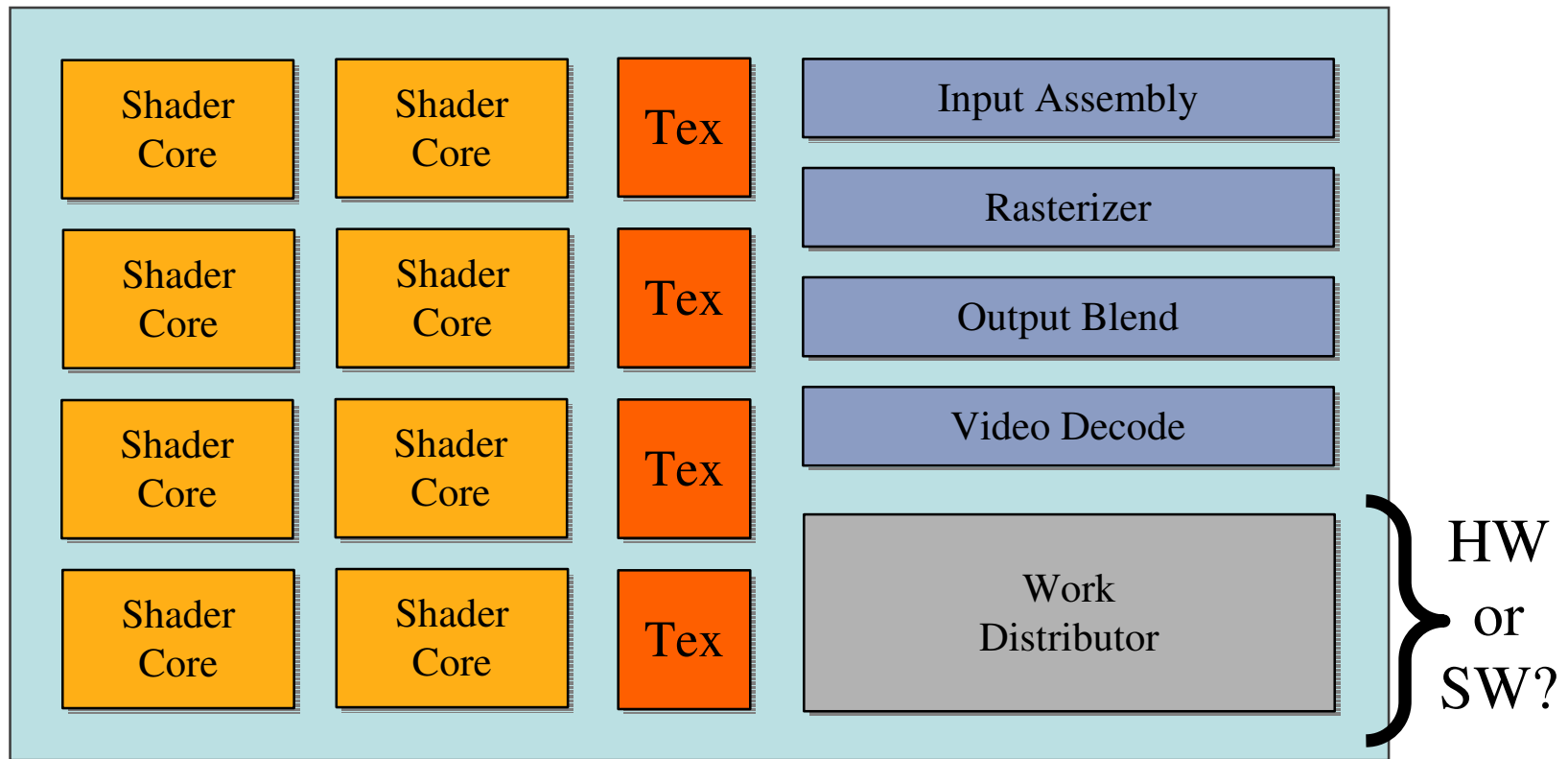
- **Identify three concepts of GPU architecture**

- Know design space
- Establish intuition: Which algorithms / workloads are suitable for these architectures?
- Best results obtained when "thinking data parallel"
- Not about: Detailed optimization for one architecture

- **Terminology**

- Not entirely technically correct, but we use the following synonymously (pick your language):
 - Vertices / fragments / primitives / shaders (graphics)
 - Work items (OpenCL)
 - Compute threads (DX11 Compute)
 - CUDA threads

What's in a GPU?



Heterogeneous chip multi-processor (highly tuned for graphics)

Example operation

```
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 unprocessed work-item input record



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

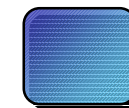
float4 diffuseShader(float3 norm, float2
uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp ( dot(lightDir, norm), 0.0,
1.0 );
    return float4(kd, 1.0);
}
```



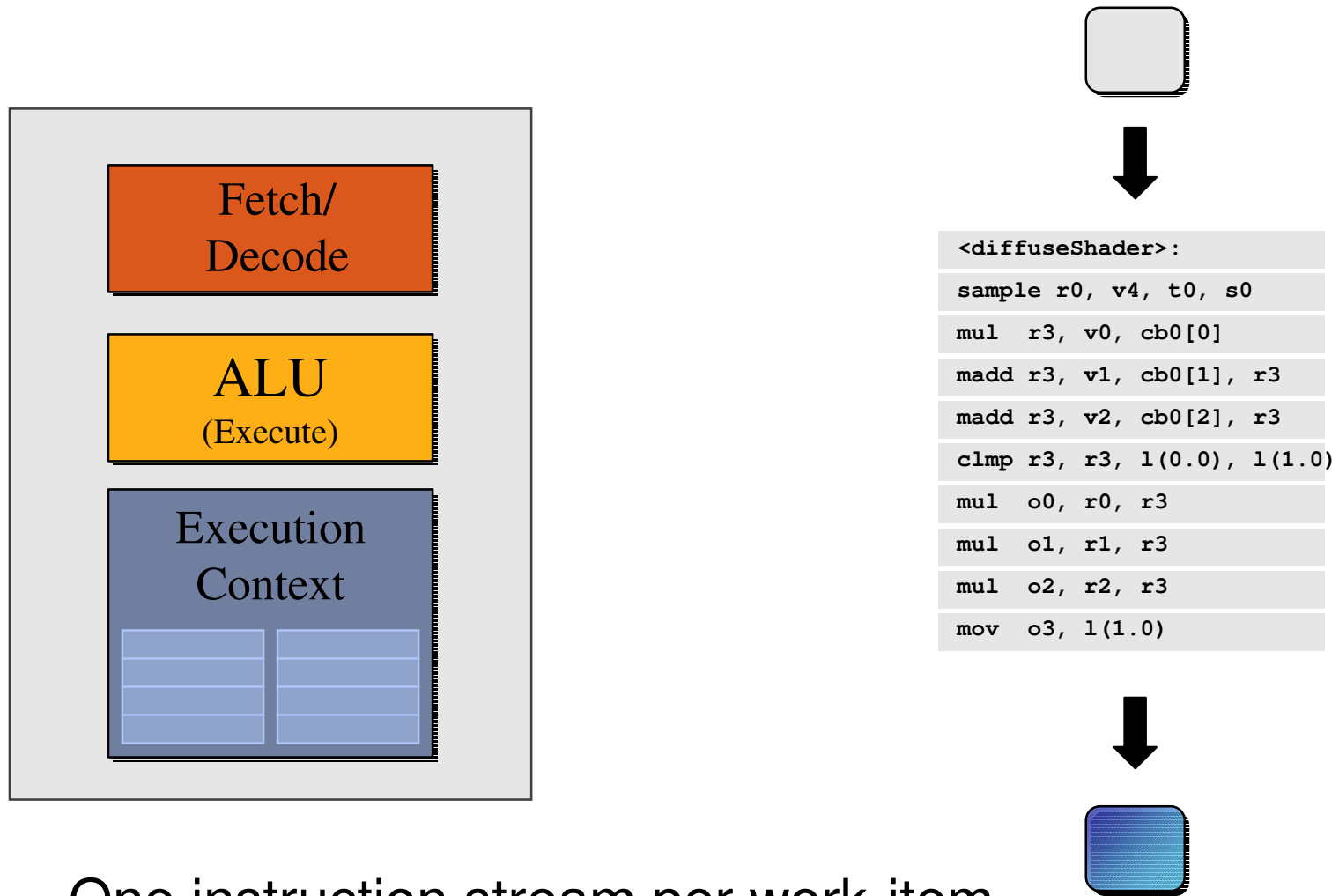
```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
```



1 processed work-item output record

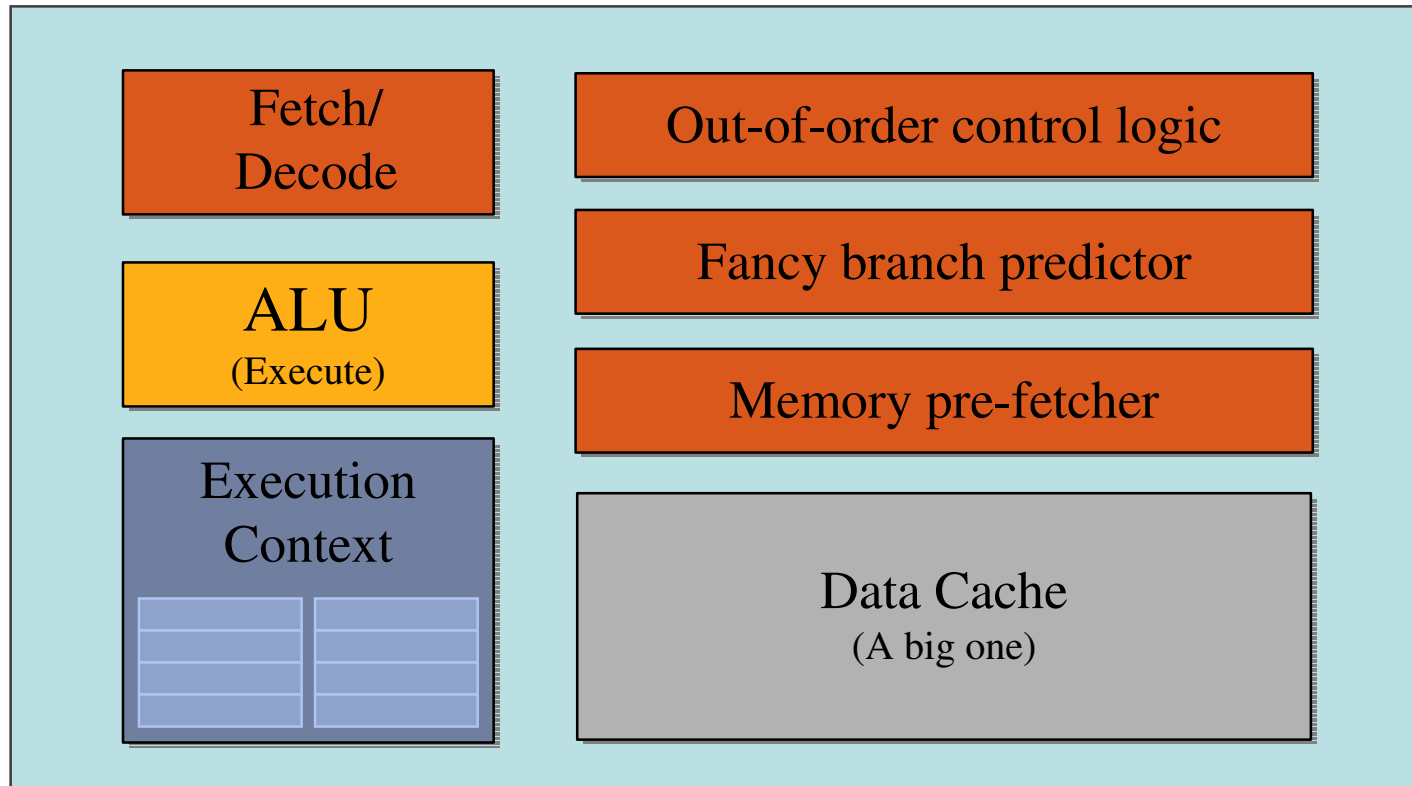


Execute shader

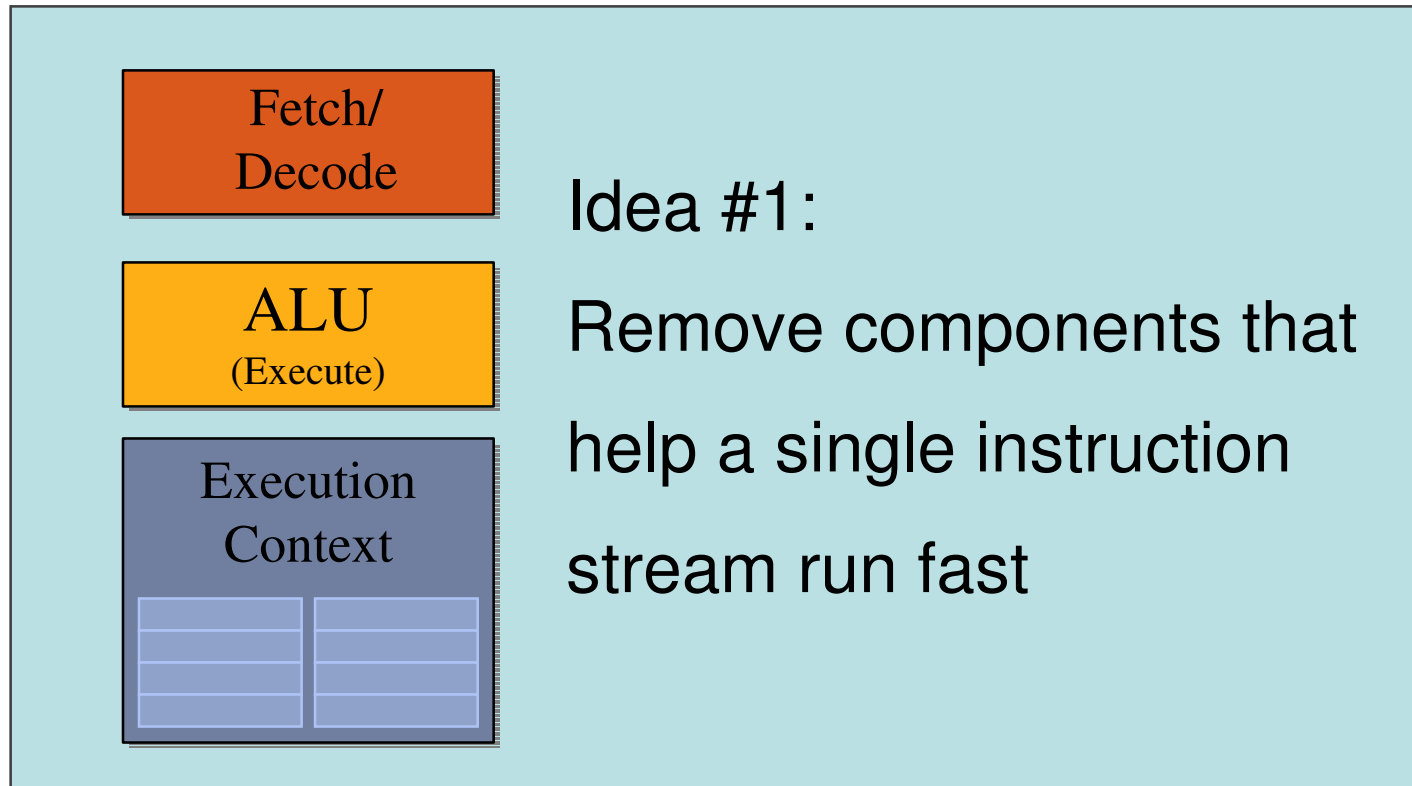


One instruction stream per work-item

CPU-“style” cores

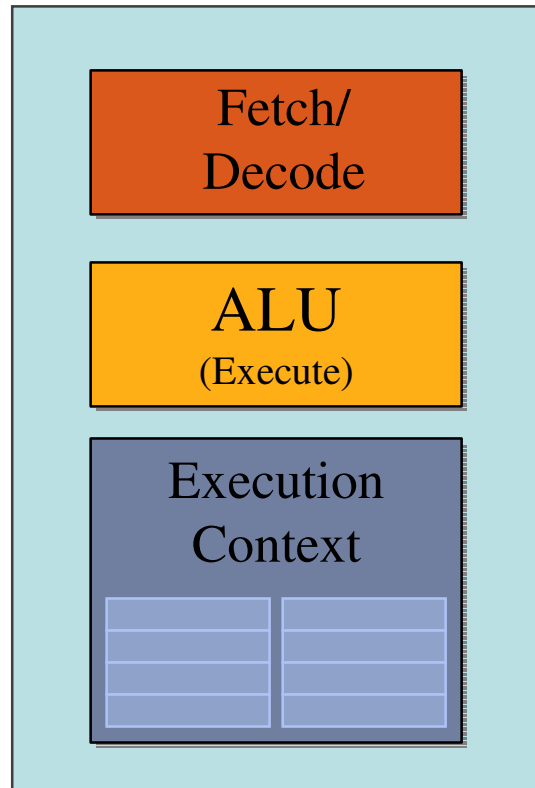
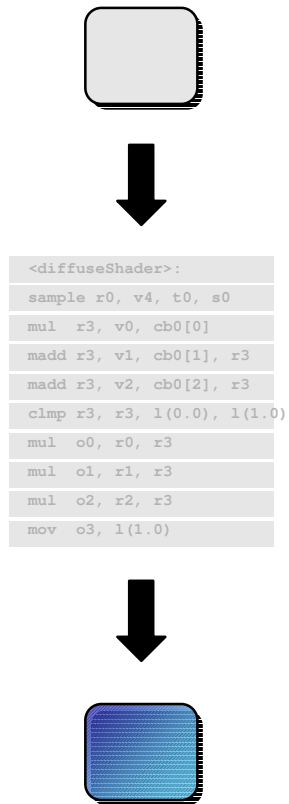


Slimming down

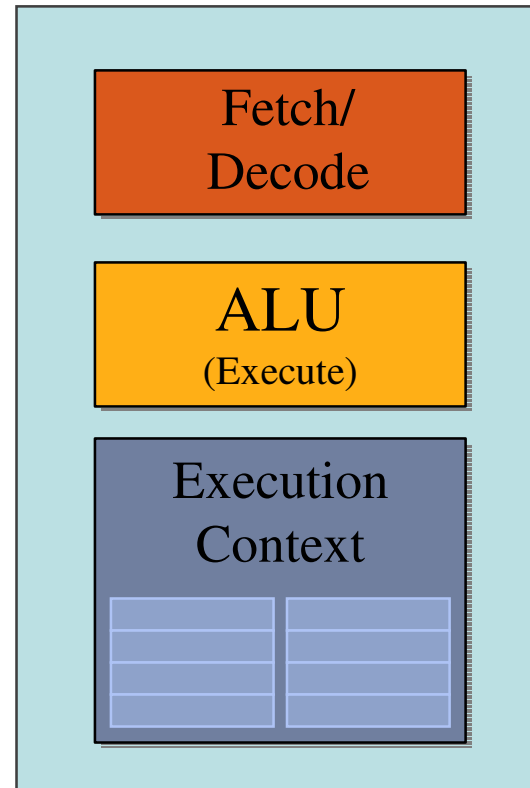
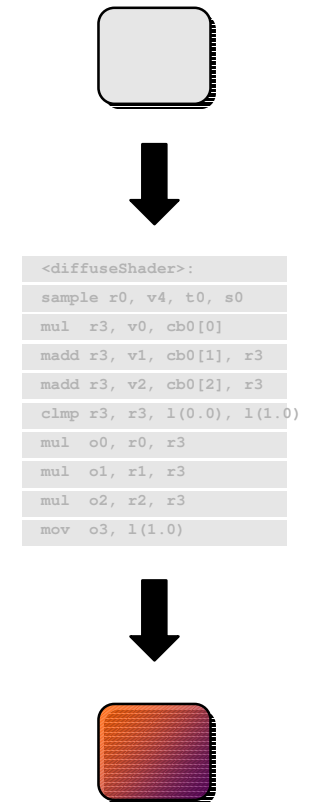


Two cores

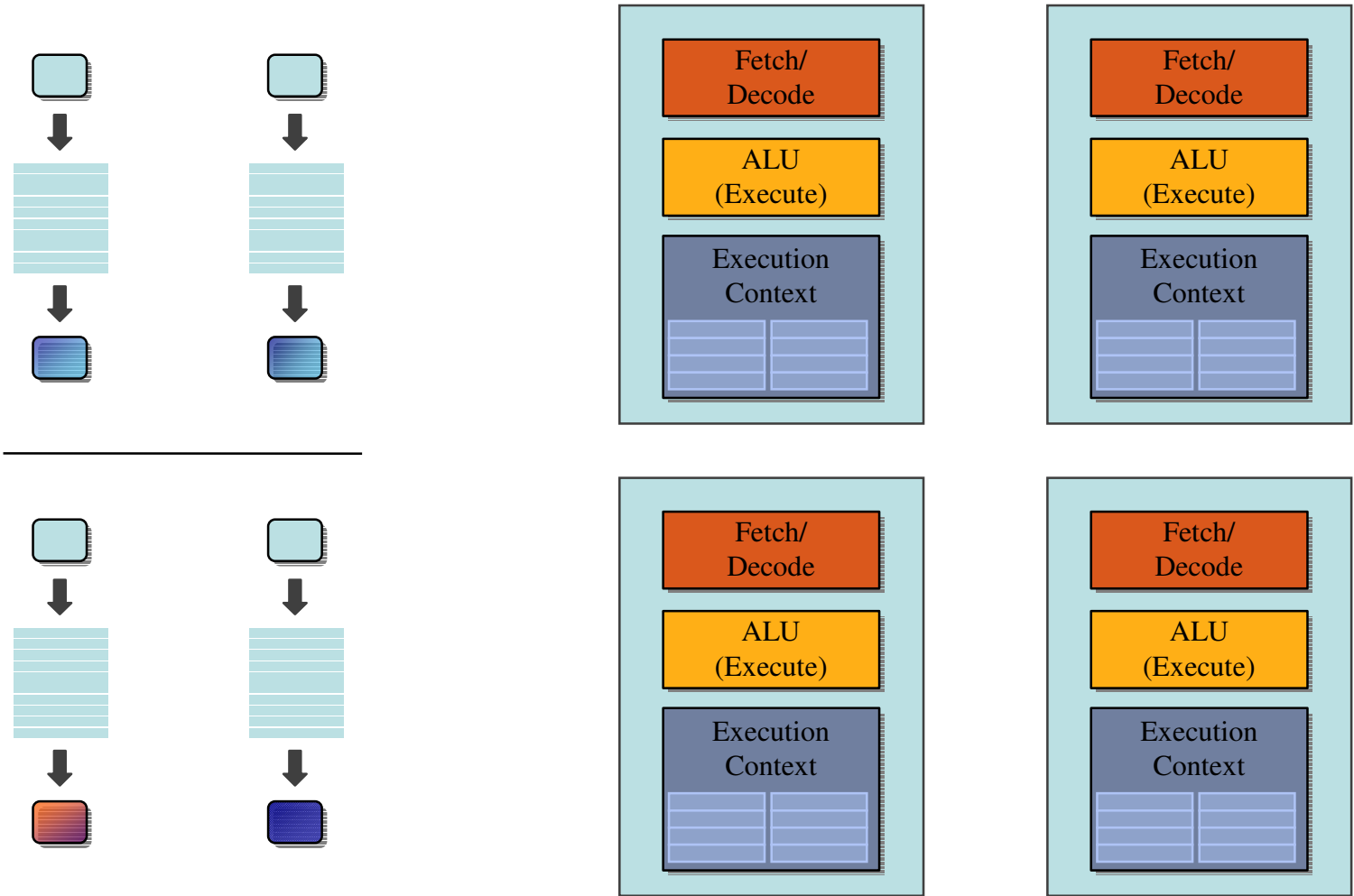
work-item 1



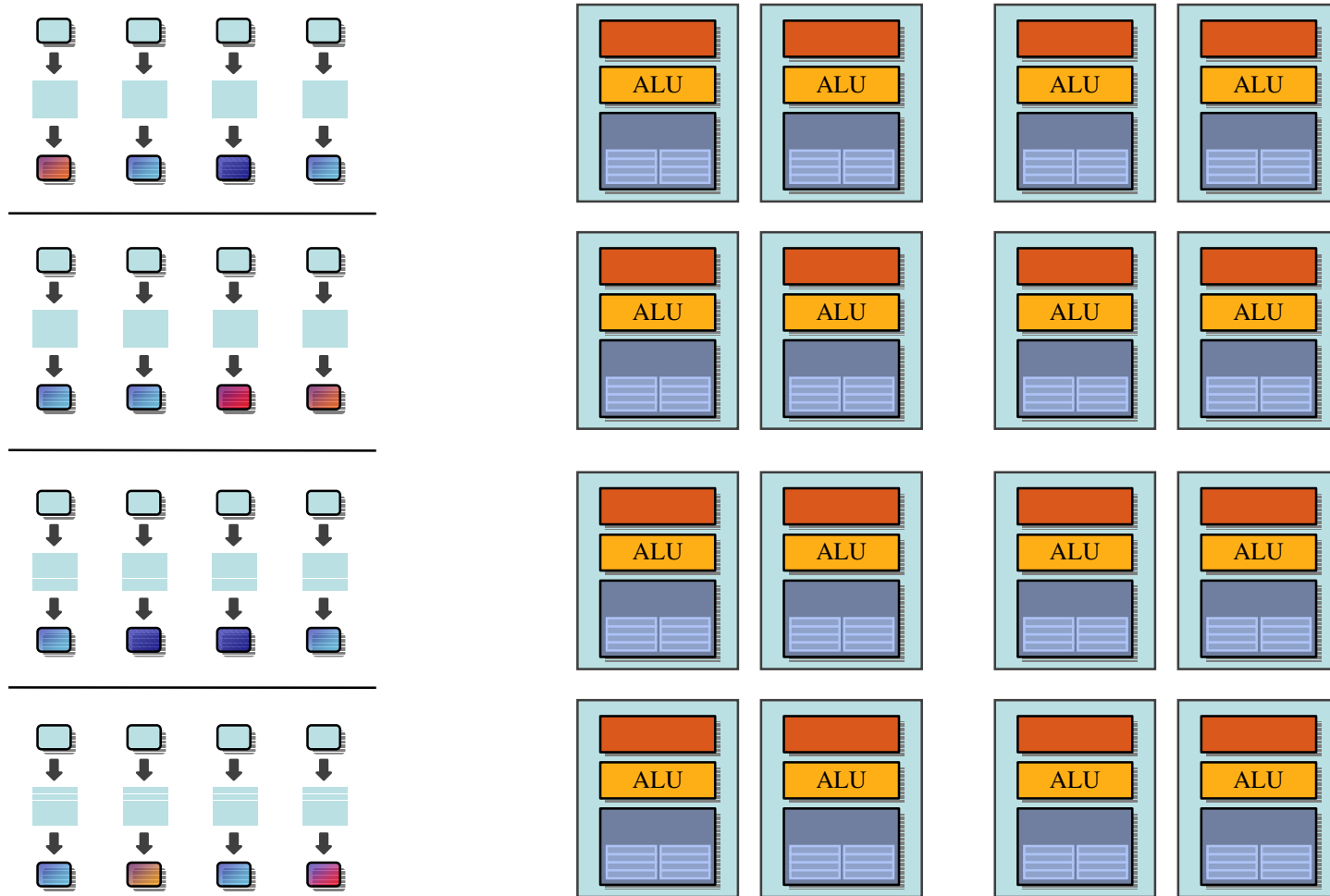
work-item 2



Four cores

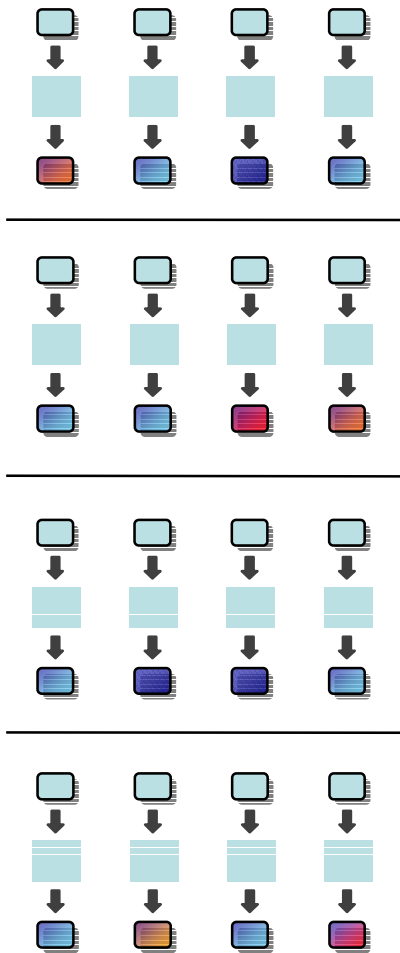


Sixteen cores



16 cores = 16 simultaneous instruction streams

Instruction stream sharing

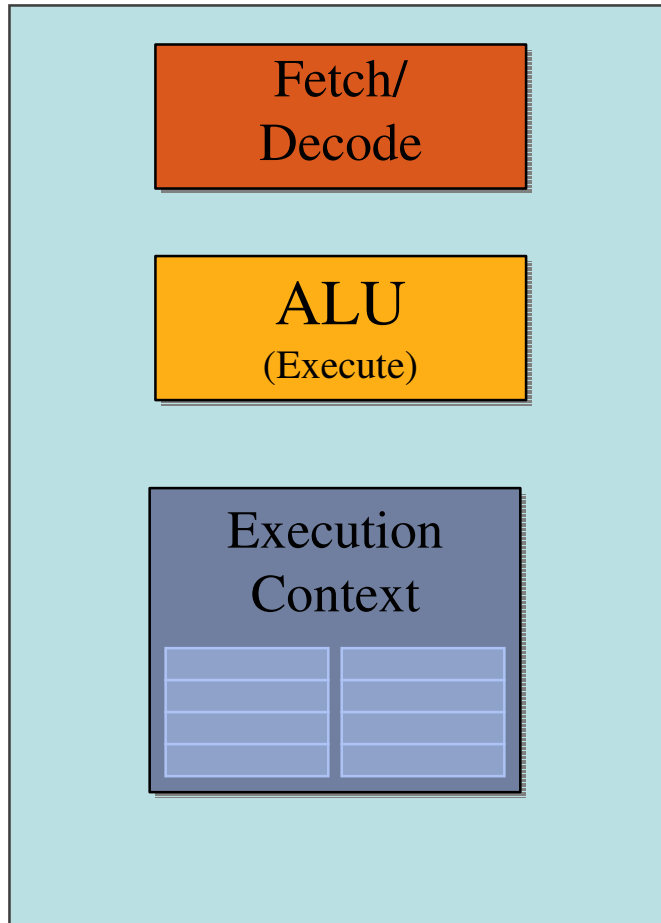


But... many items should be able to share an instruction stream!

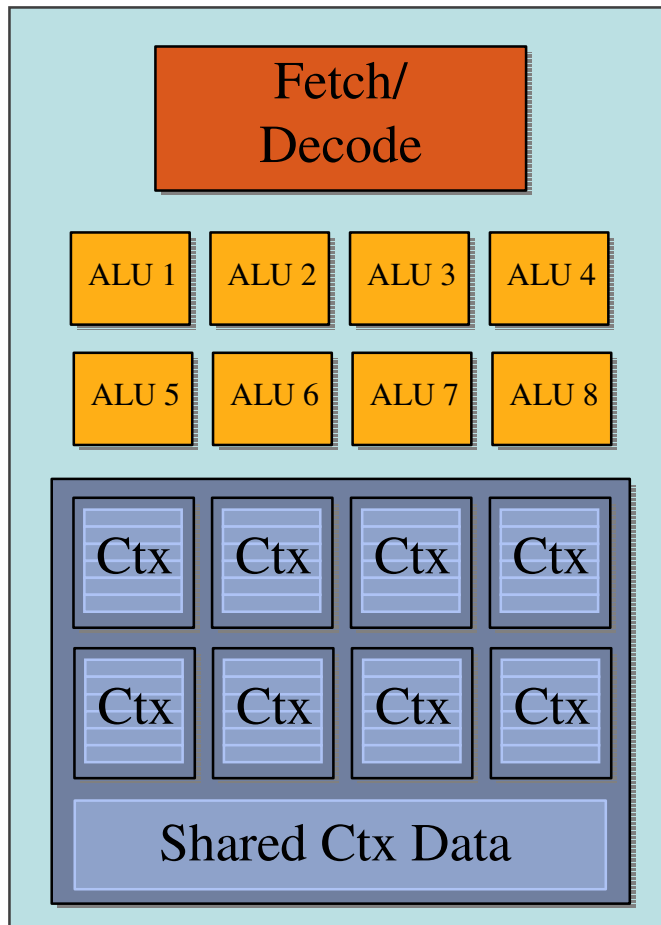
```

<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, 1(0.0), 1(1.0)
mul   o0, r0, r3
mul   o1, r1, r3
mul   o2, r2, r3
mov   o3, 1(1.0)
    
```

Recall: simple processing core



Add ALUs

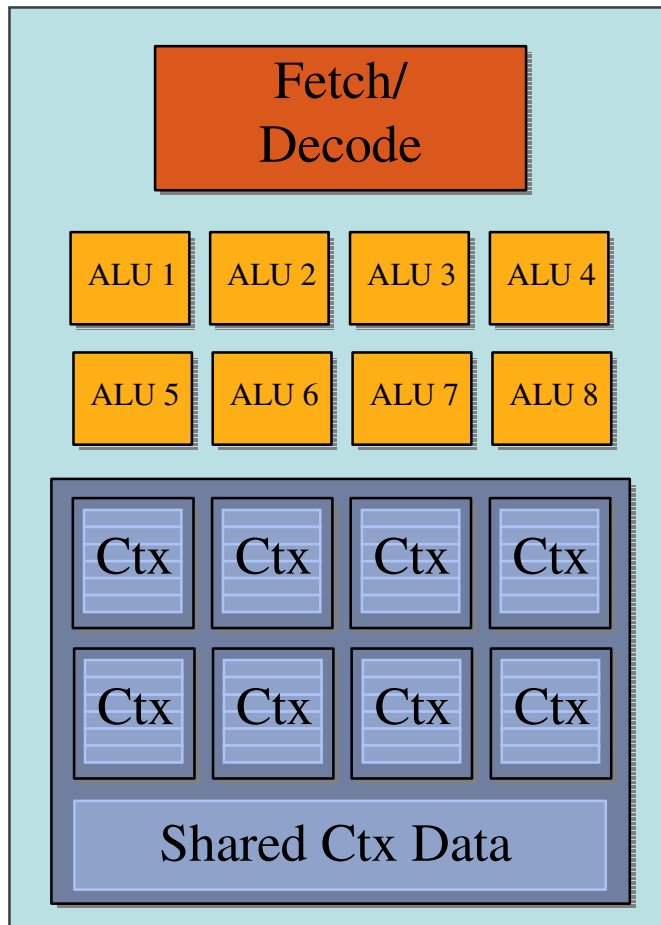


Idea #2:

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

SIMD processing

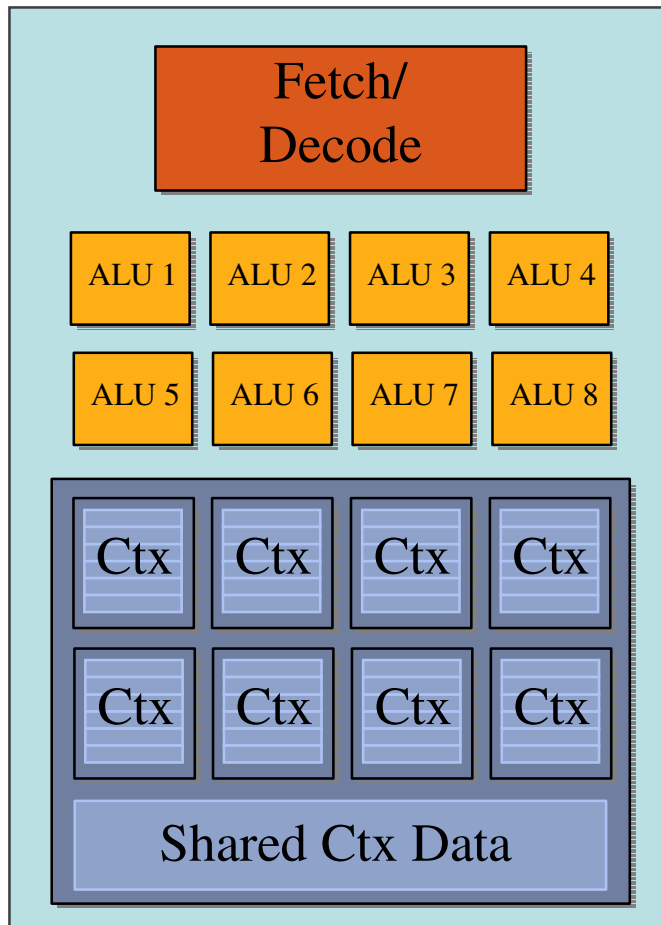
Modifying the shader



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

Original compiled shader:
Processes one work-item
using scalar ops on scalar
registers

Modifying the shader

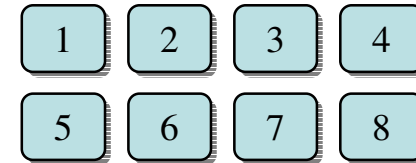
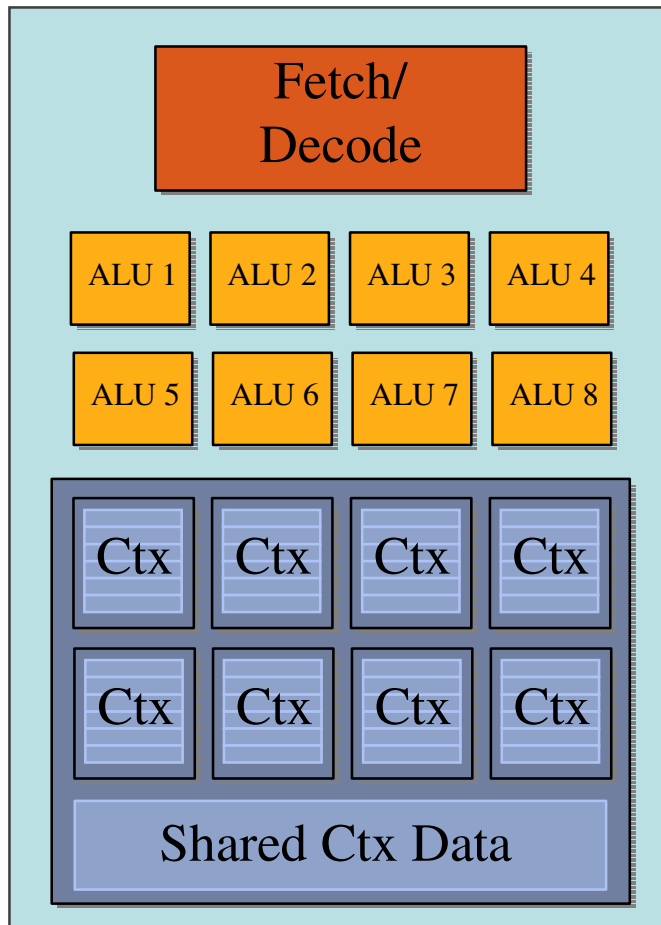


```
<VEC8_diffuseShader>:  
VEC8_sample vec_r0, vec_v4, t0, vec_s0  
VEC8_mul   vec_r3, vec_v0, cb0[0]  
VEC8_madd  vec_r3, vec_v1, cb0[1], vec_r3  
VEC8_madd  vec_r3, vec_v2, cb0[2], vec_r3  
VEC8_clmp  vec_r3, vec_r3, 1(0.0), 1(1.0)  
VEC8_mul   vec_o0, vec_r0, vec_r3  
VEC8_mul   vec_o1, vec_r1, vec_r3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

New compiled shader:

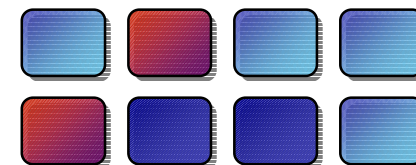
Processes 8 work-items
using vector ops on vector
registers

Modifying the kernel

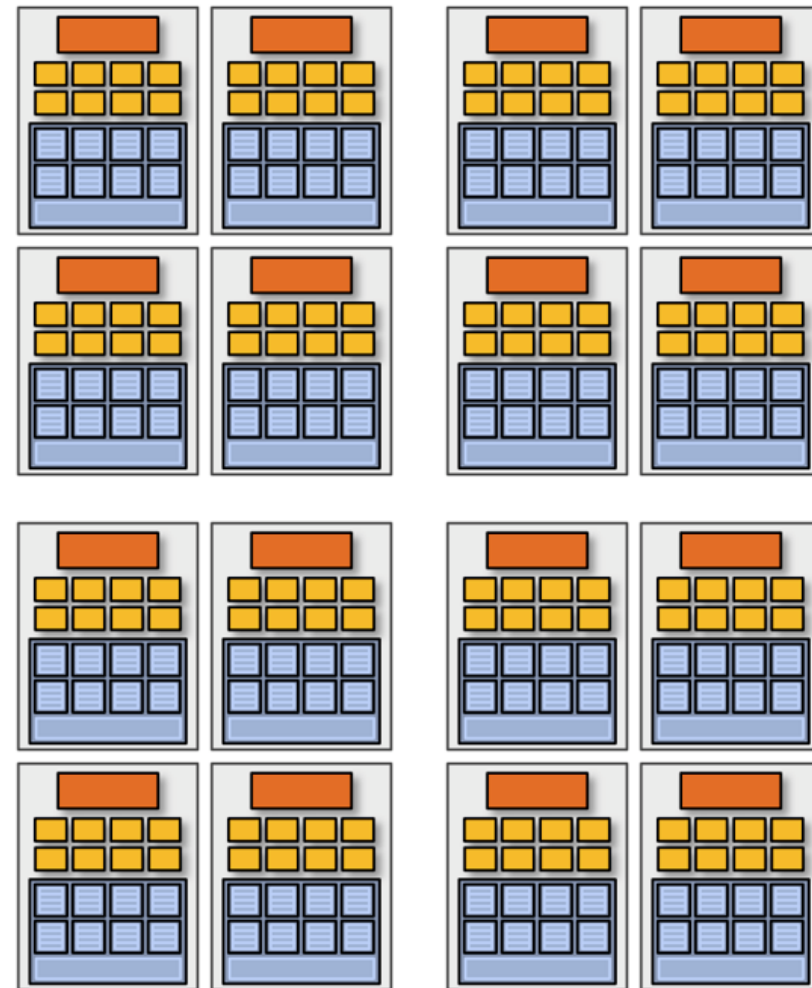
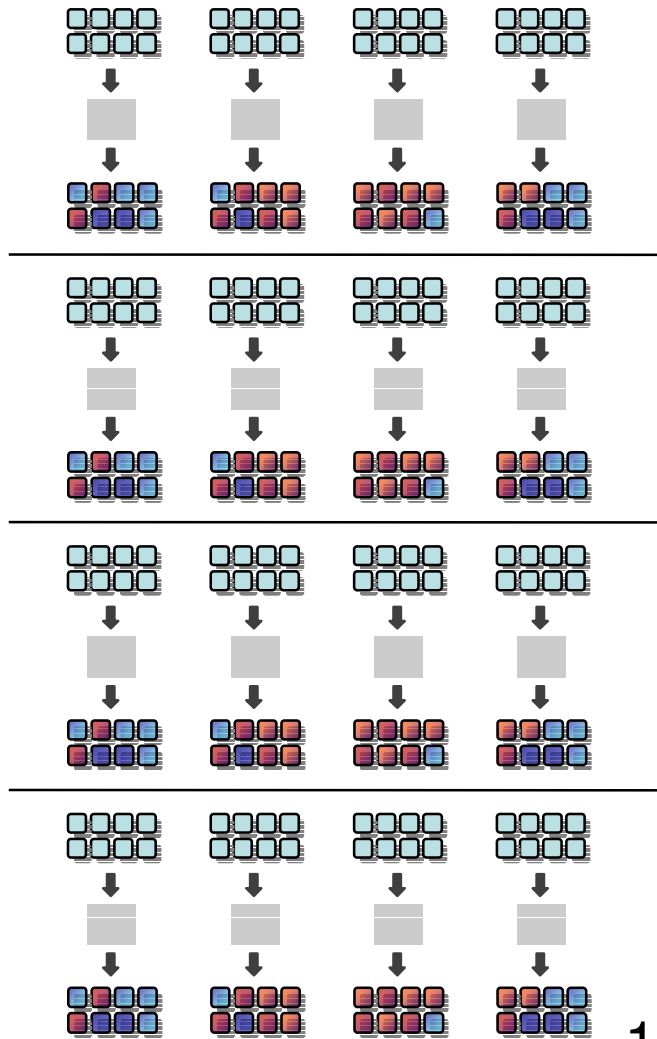


<VEC8_diffuseShader>:

```
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, 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)
```

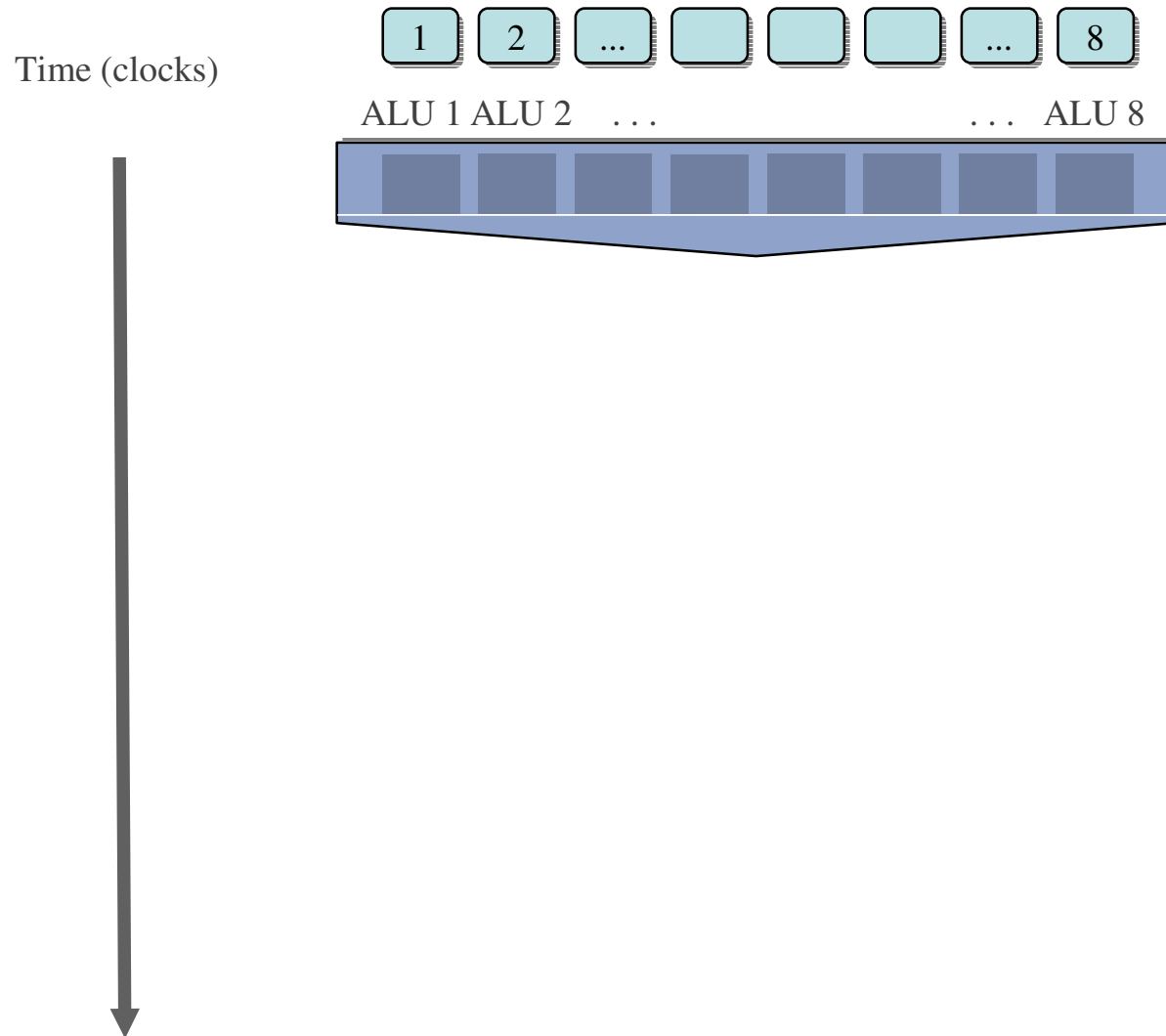


128 elements in parallel



16 cores = 128 ALUs
= 16 simultaneous instruction streams

But what about branches?

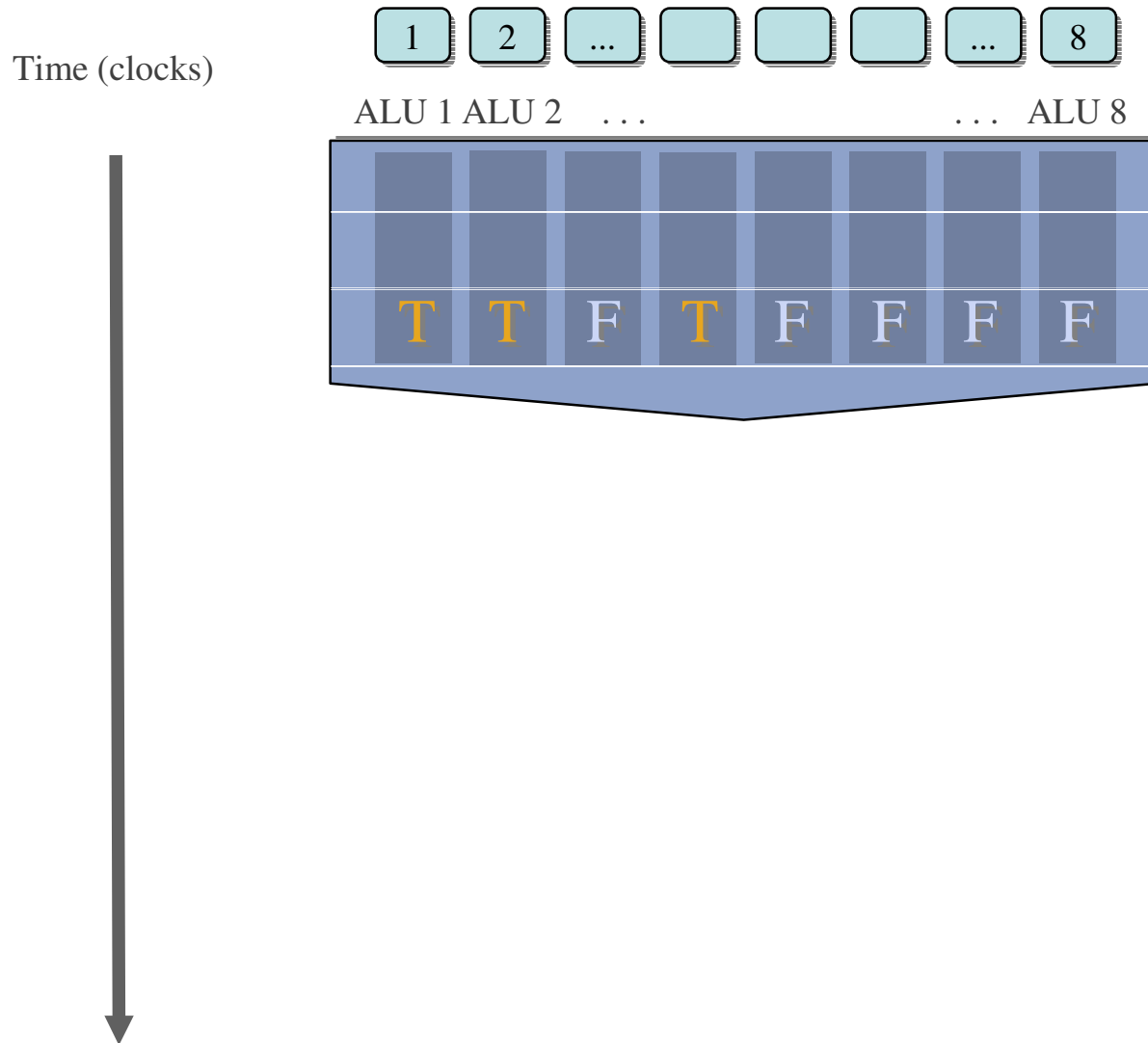


<unconditional shader code>

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

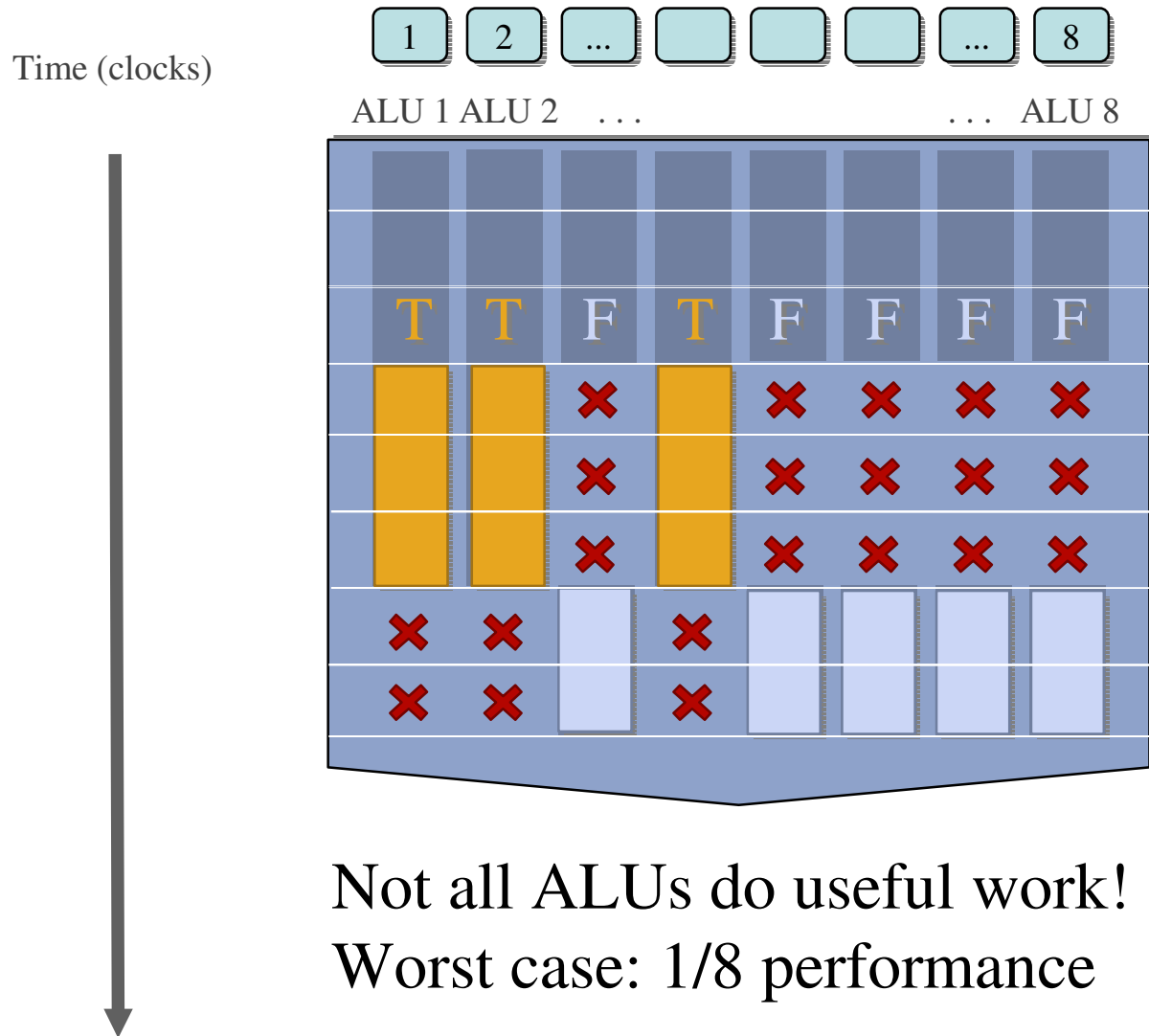
<resume unconditional
shader code>

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), ATI Radeon architectures



In practice: 16 to 64 work-items share an instruction stream

Stalls!

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

Memory 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 work items.

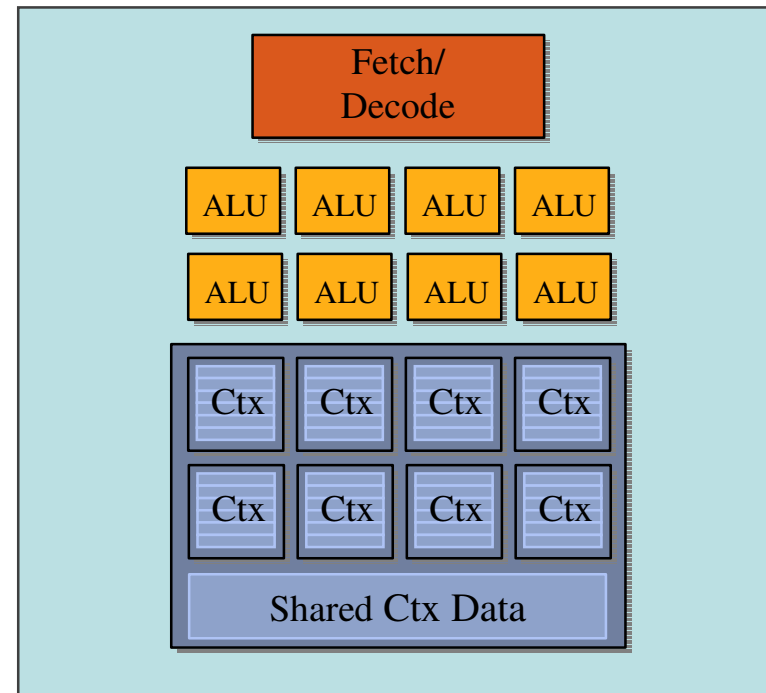
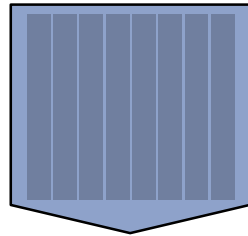
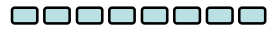
Idea #3:

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

Hiding stalls

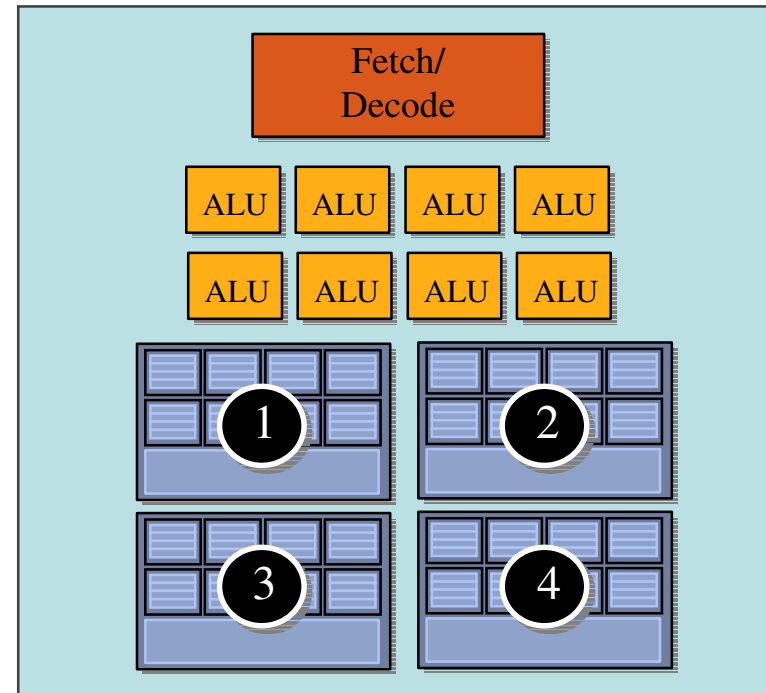
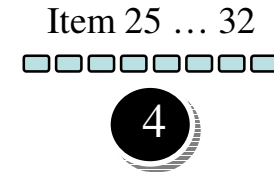
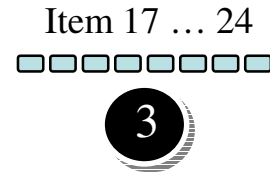
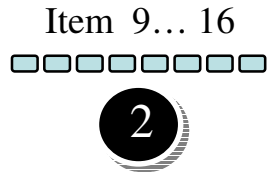
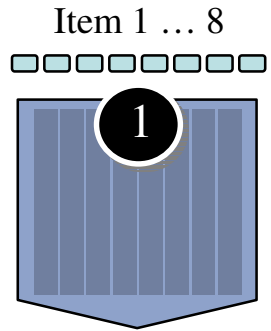
Time (clocks)

Item 1 ... 8

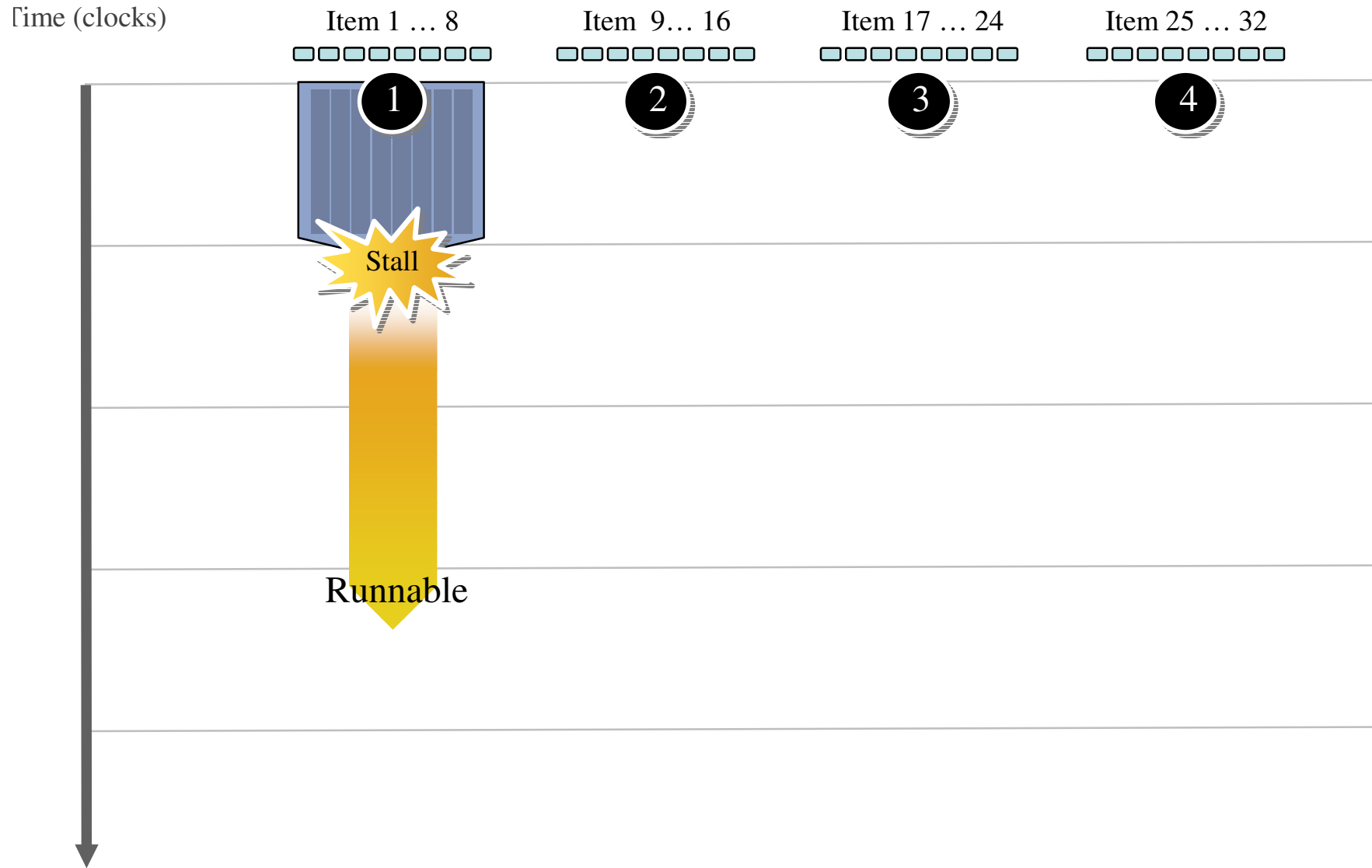


Hiding stalls

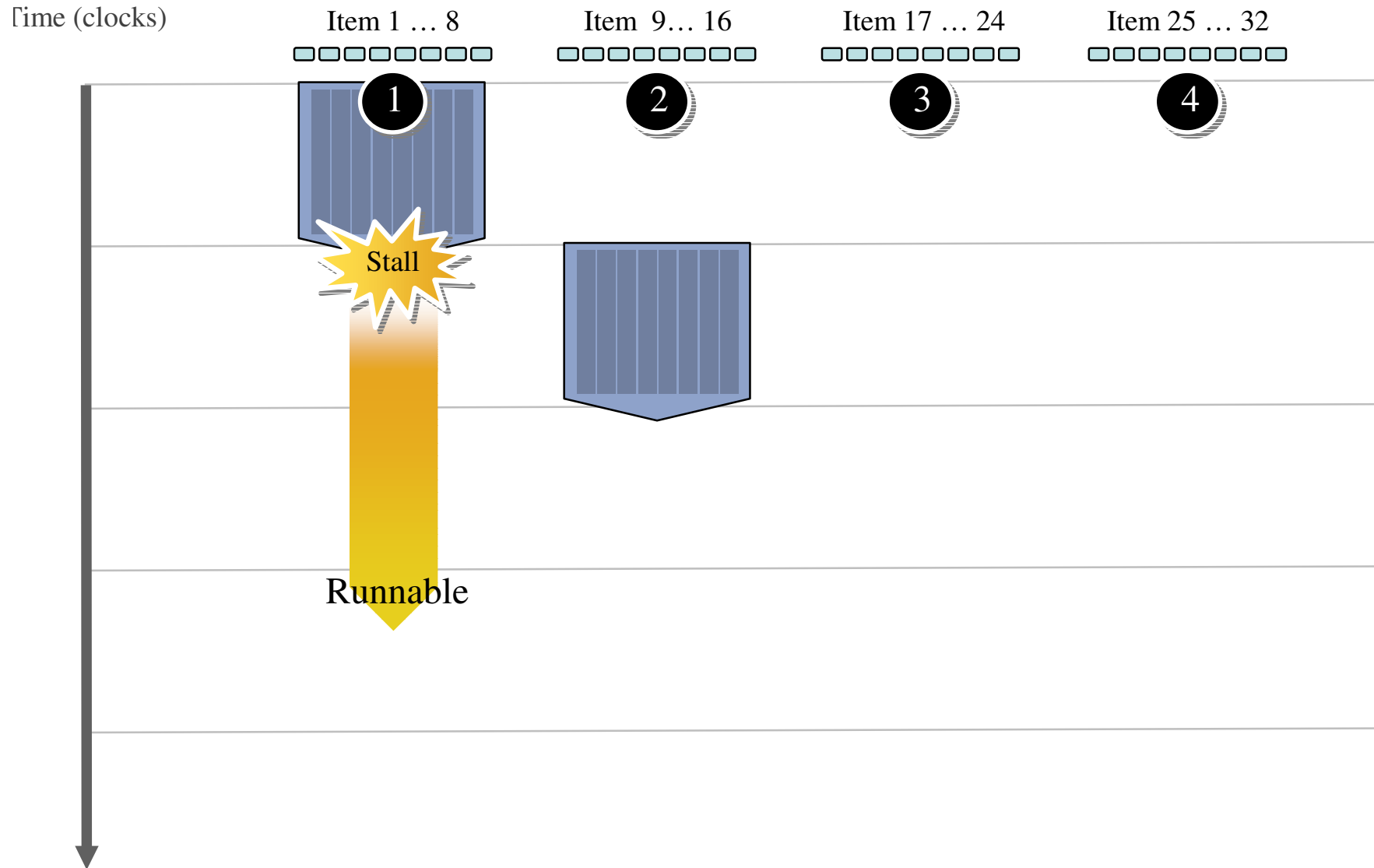
Time (clocks)



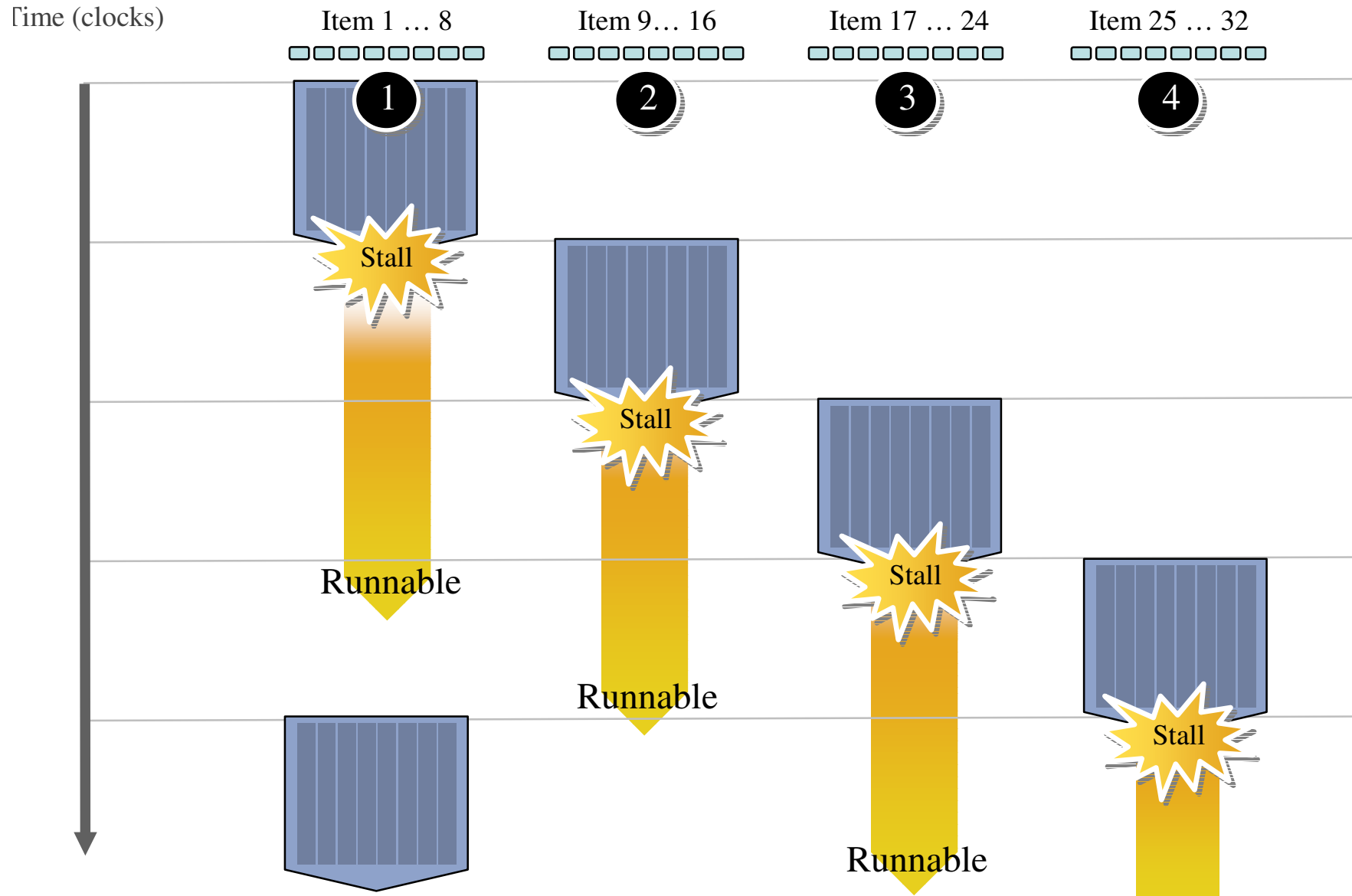
Hiding stalls



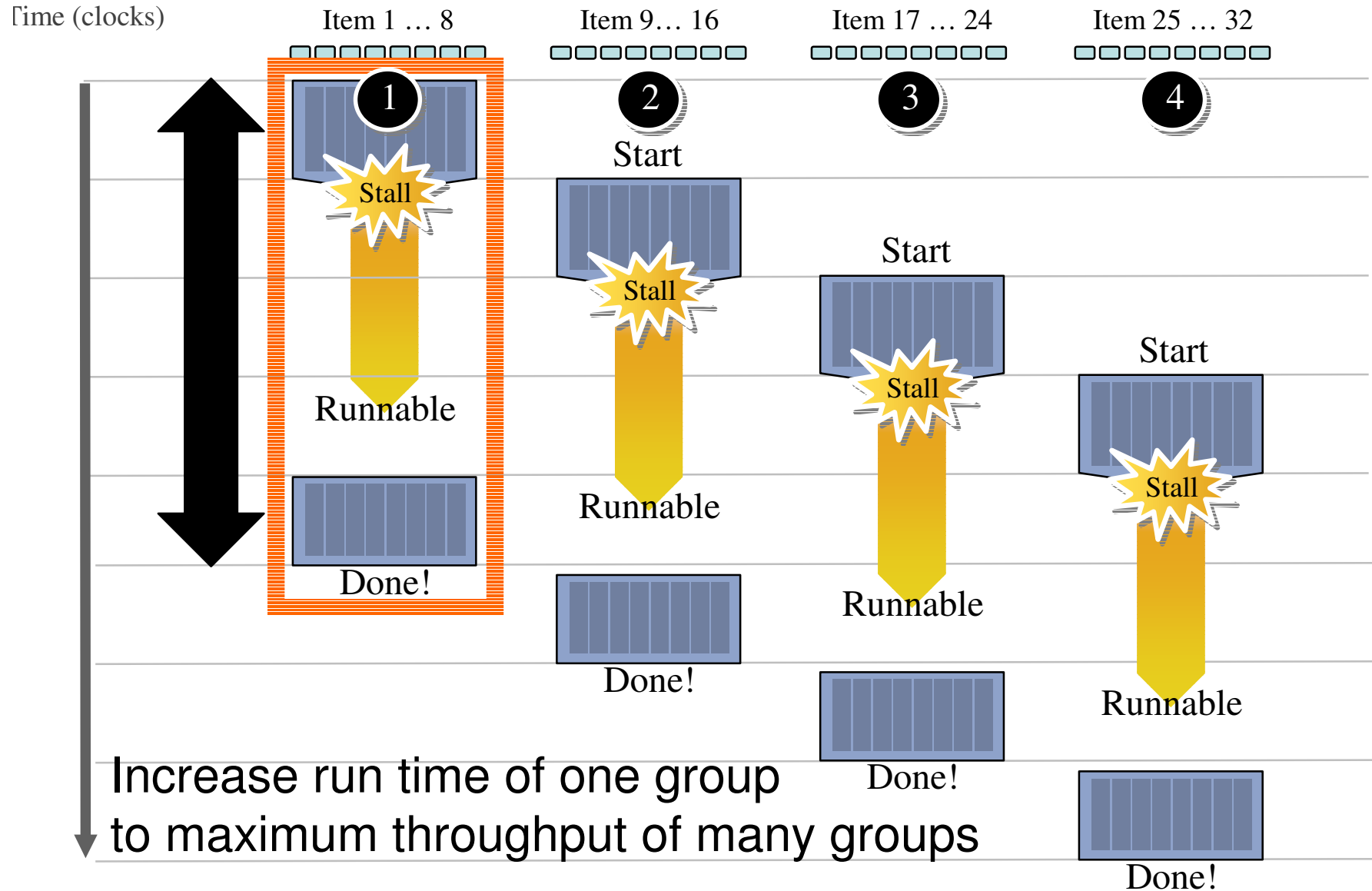
Hiding stalls



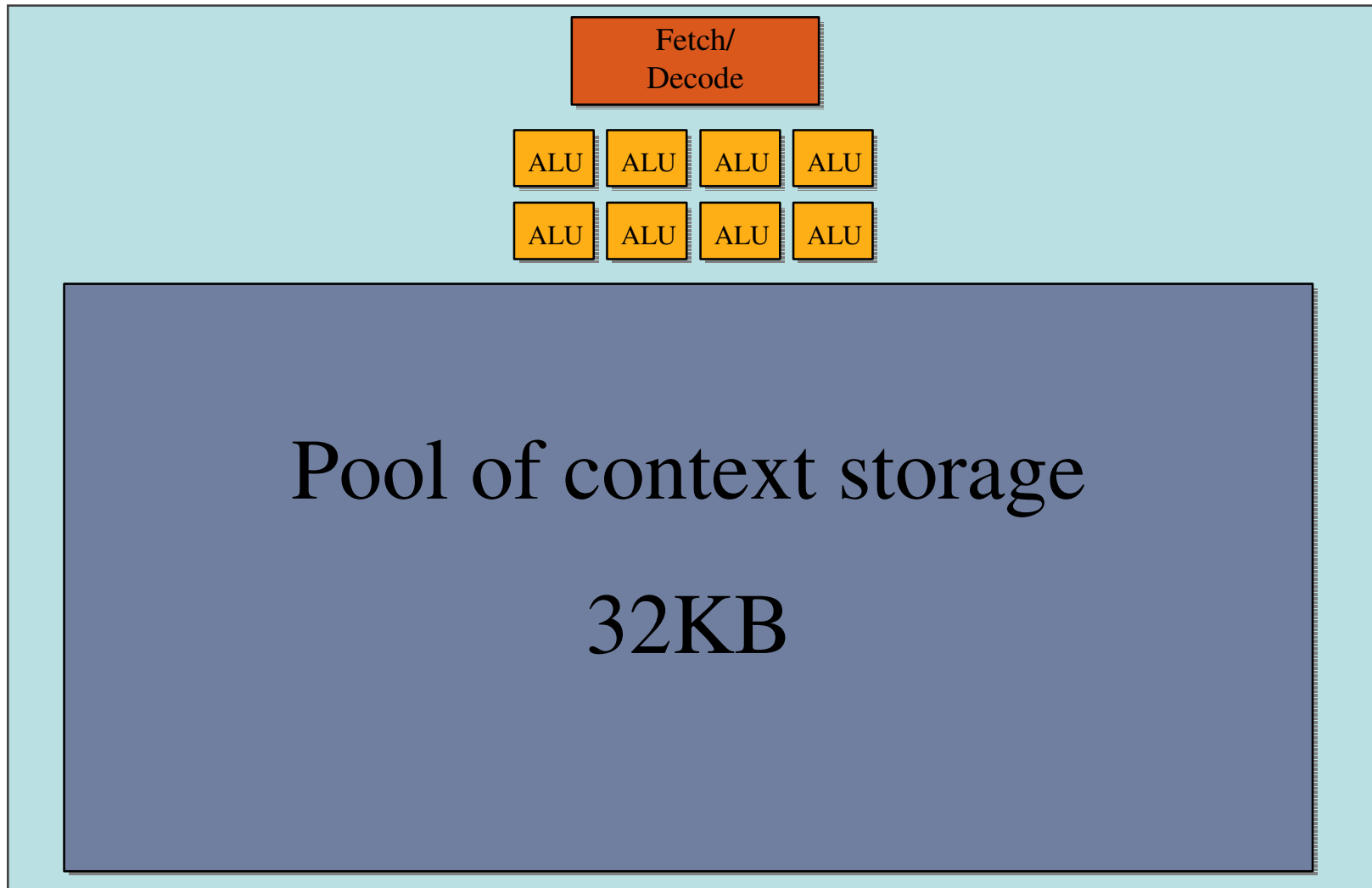
Hiding stalls



Throughput!

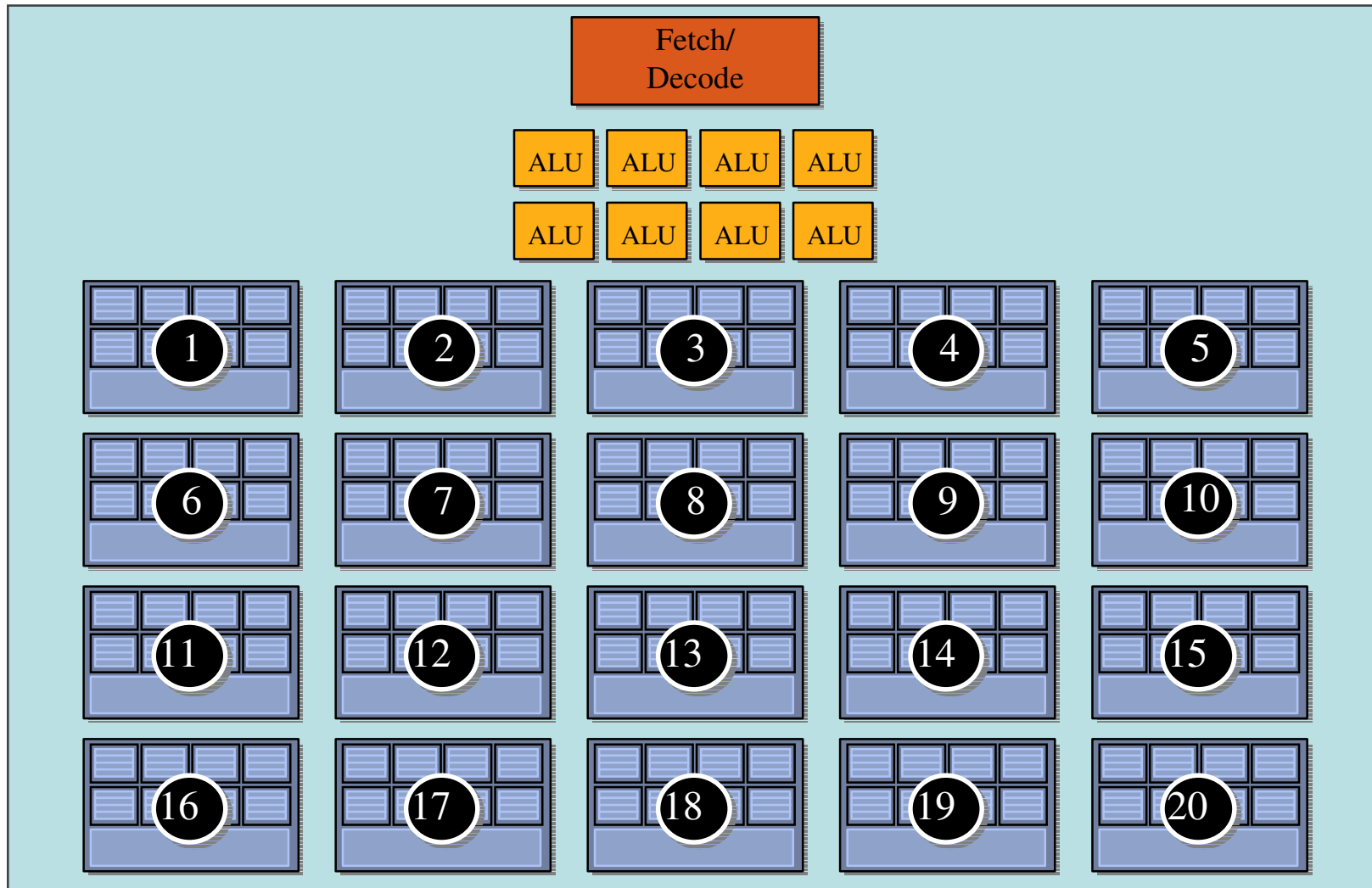


Storing contexts

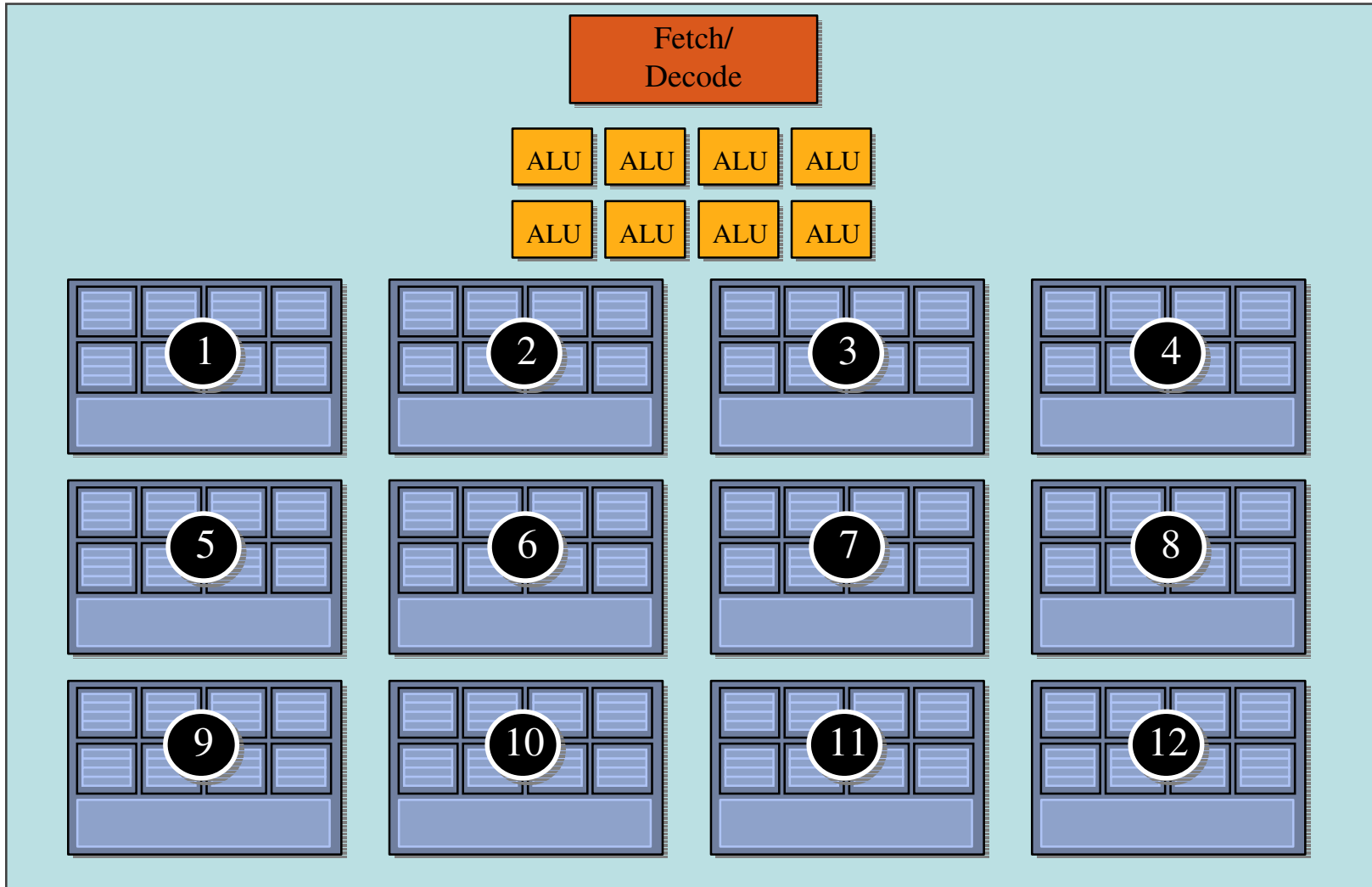


Twenty small contexts

(maximal latency hiding ability)

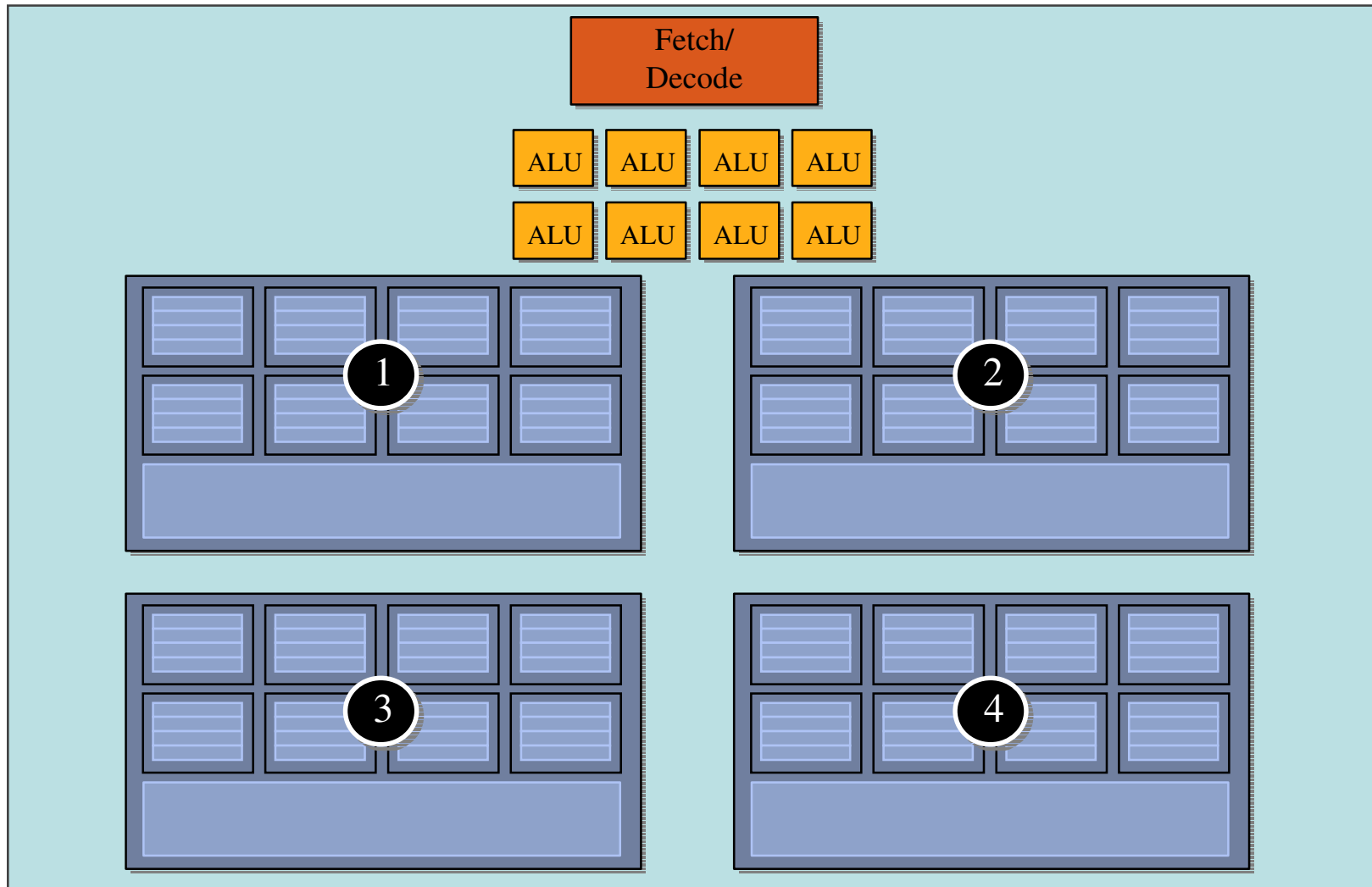


Twelve medium contexts



Four large contexts

(low latency hiding ability)



Scalable design: Mid-range

16 cores

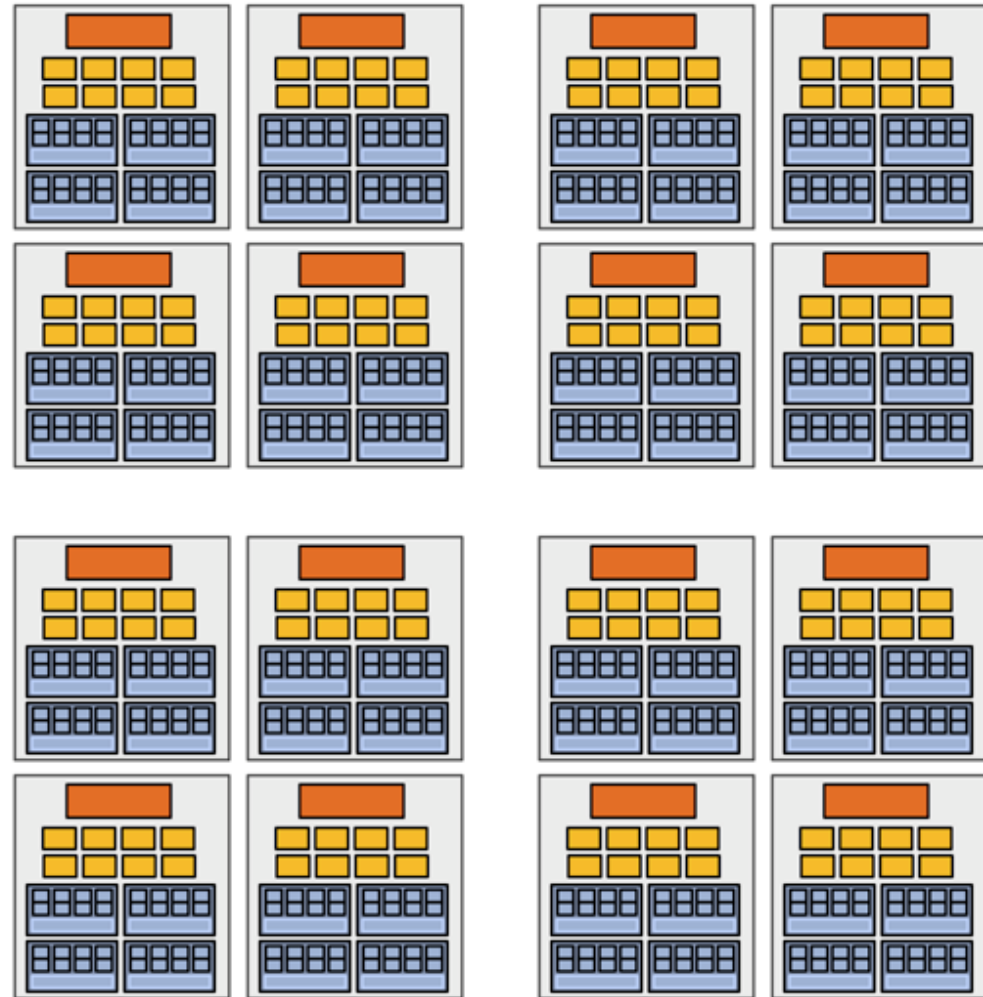
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

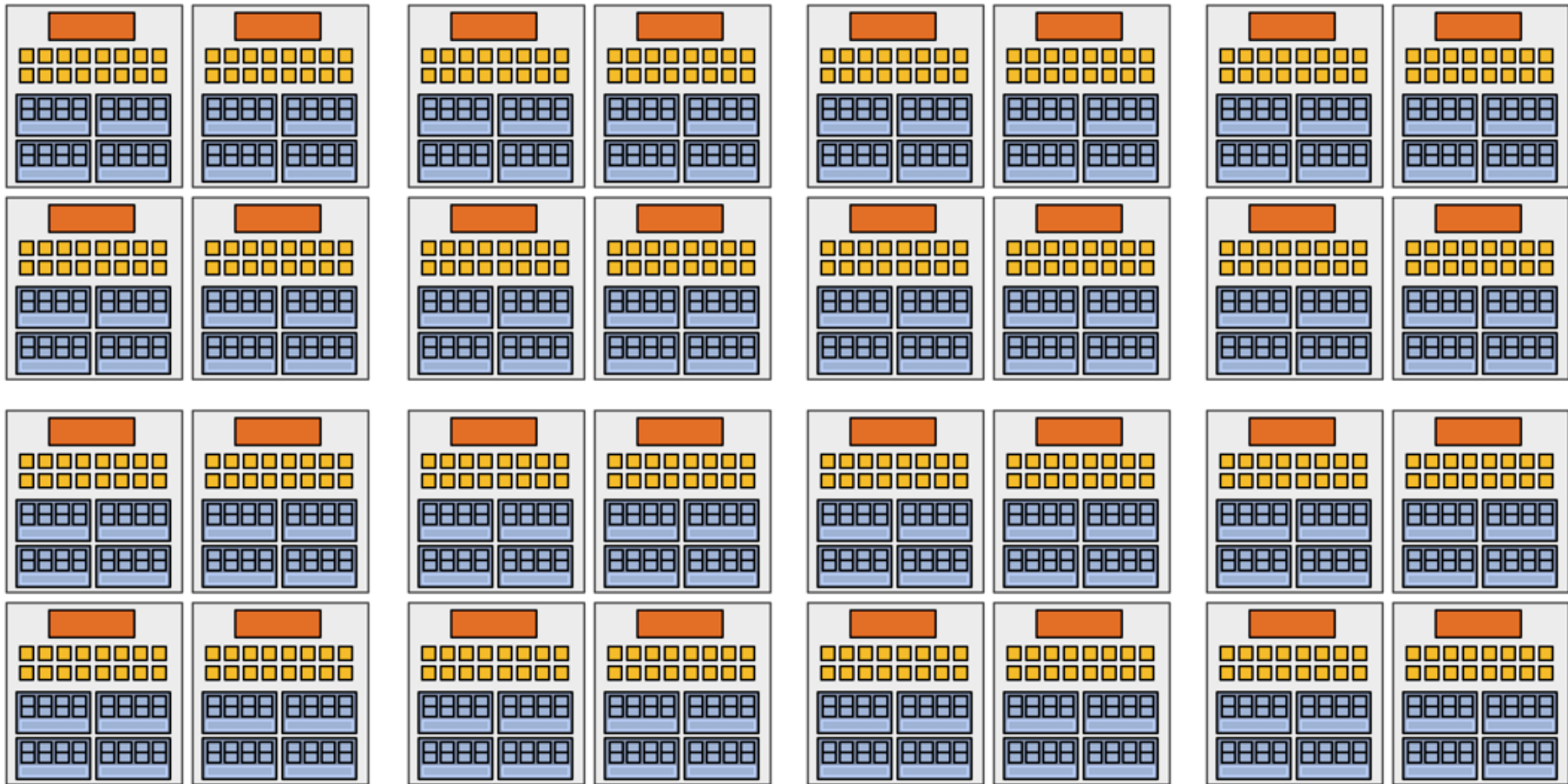
64 concurrent (but interleaved)
instruction streams

512 concurrent work items

= 256 GFLOPs (@ 1GHz)



Scalable design: High-end



32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)

Summary: Three key ideas

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

Roundup

- **How do real designs fit into this model architecture?**
- **For details, see**
 - Kayvon Fatahalian and Mike Houston: "*A Closer Look at GPUs*", Communications of the ACM 51(10), October 2008

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

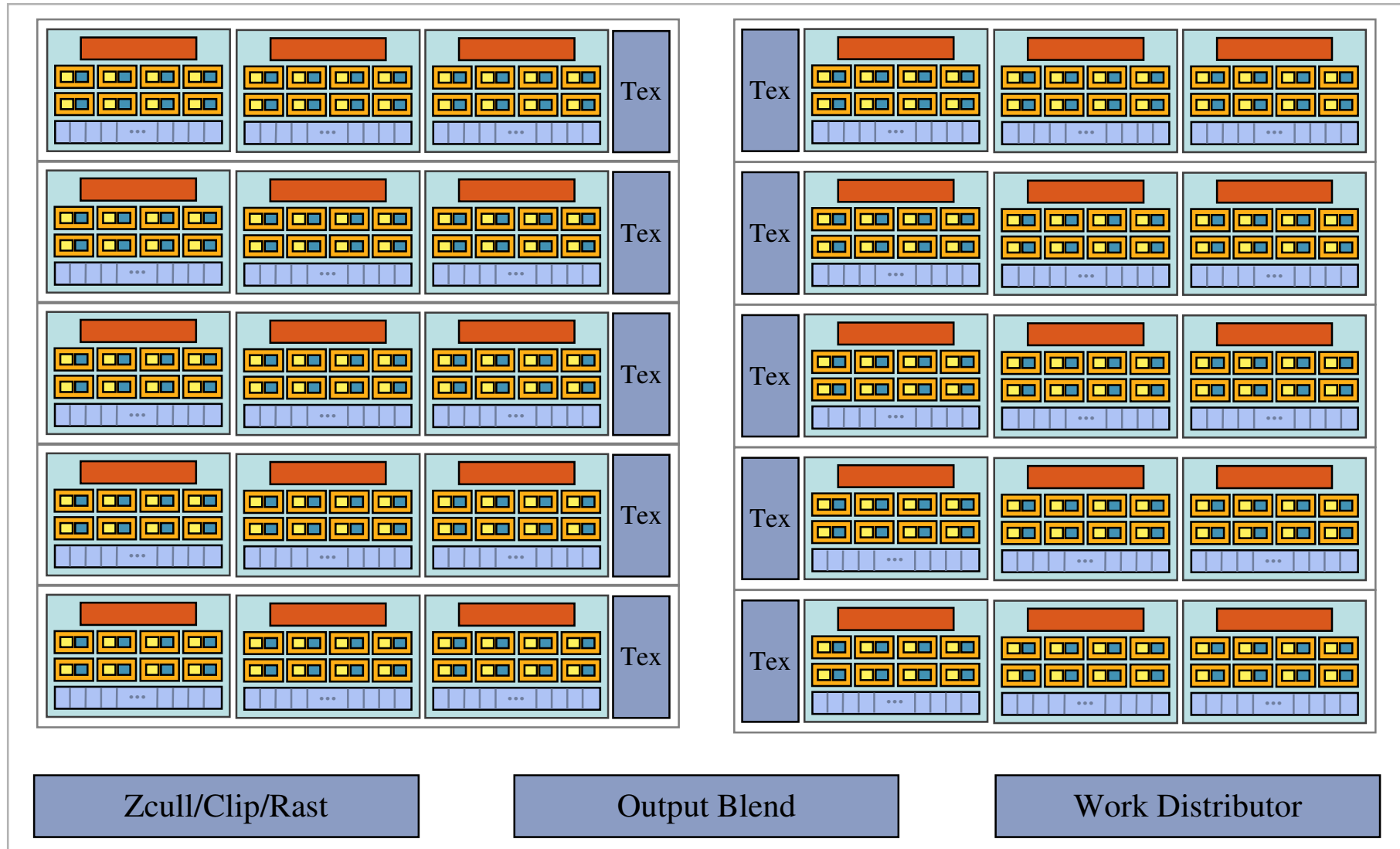


= fixed function unit

NVIDIA GeForce 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 work-items (16 for vertices)
 - 8 work-items run on 8 SIMD functional units in one clock
 - Instruction repeated for 4 clocks (2 clocks for vertices)

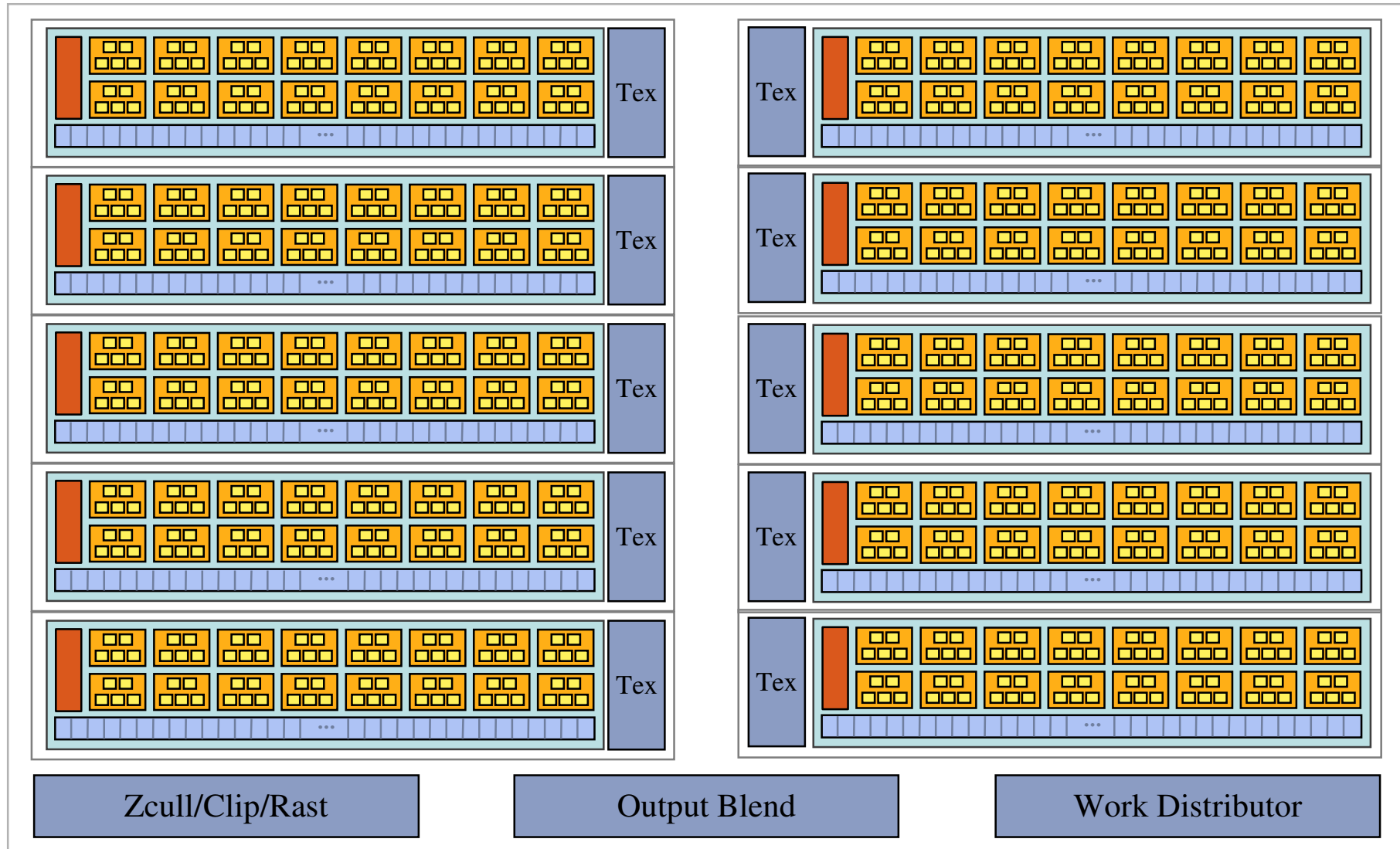
NVIDIA GeForce GTX 280



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 work-items
 - 16 work-items run on 16 SIMD functional units in one clock
 - Instruction repeated for 4 consecutive clocks

ATI Radeon 4870



Intel Larrabee

- **Intel speak:**

- We won't say anything about the number of cores or clock rate of cores
- Explicit 16-wide vector ISA
- If 1GHz clock (then 1 core = 1 LRB unit = 32 GFLOPS -- from SIGGRAPH 08 paper)

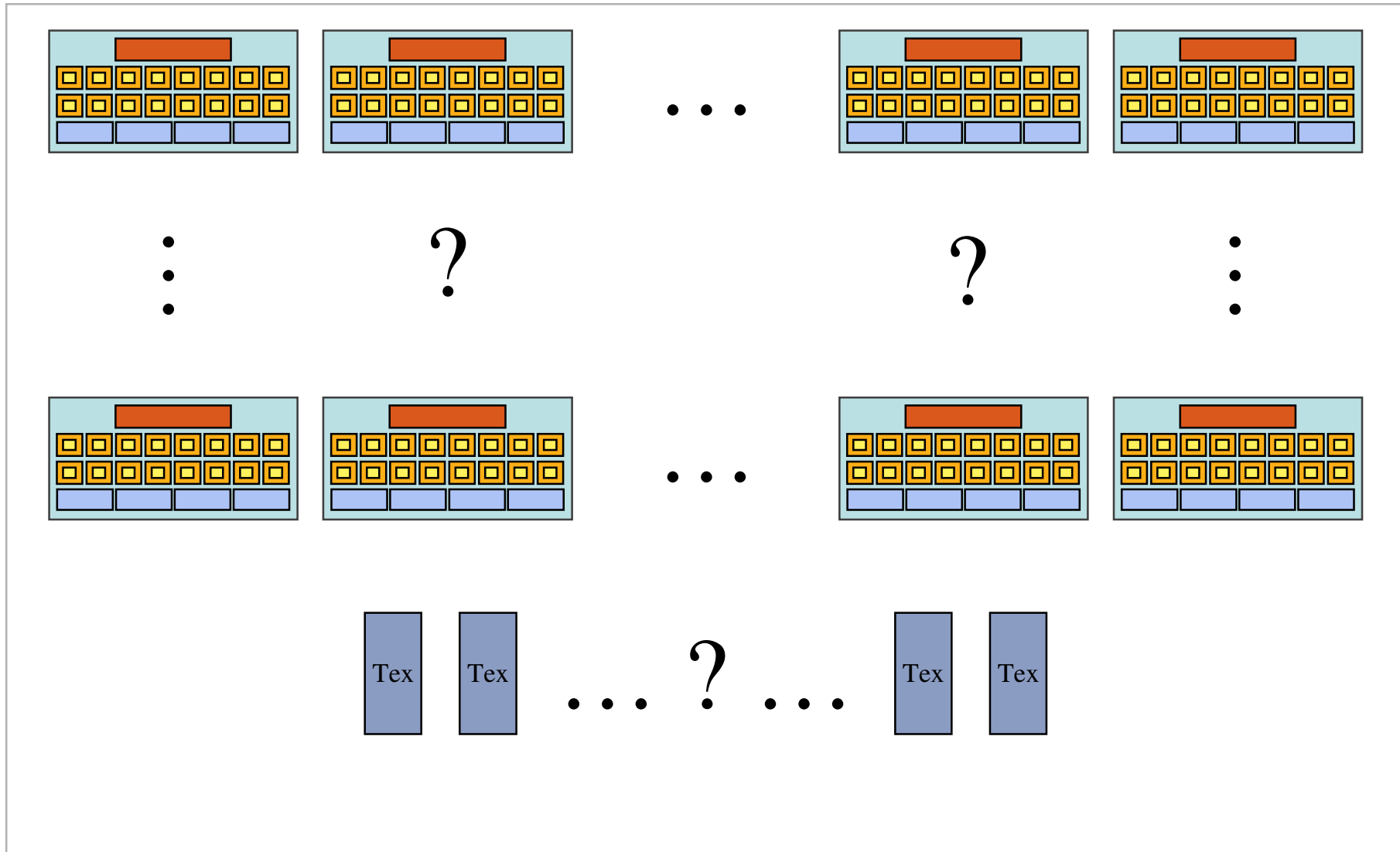
- **Generic speak:**

- X processing cores
- 16 SIMD functional units per core
- 1 mul-add per functional unit (2 flops/clock)
- Best case: 16X mul-adds per clock
- If you wanted to compete with current GPUs (~ 1 TFLOP), you need about 32 Larrabee units
 - $32 * 16 * 2 = 1$ TFLOP

- **Mapping data-parallelism to chip:**

- Compilation options determine instruction stream sharing across work-items (a multiple of 16)
- 16 work-items run on 16 SIMD functional units in one clock

Intel Larrabee (SIGGRAPH08 paper)



GPUs and other multithreaded devices

Type	Processor	Cores/Chip	ALUs/Core ³	SIMD width	Max T ⁴
GPUs	AMD Radeon HD 4870	10	80	64	25
	NVIDIA GeForce GTX 280	30	8	32	128
CPUs	Intel Core 2 Quad ¹	4	8	4	1
	STI Cell BE ²	8	4	4	1
	Sun UltraSPARC T2	8	1	1	4

¹ SSE processing only, does not account for traditional FPU

² Stream processing (SPE) cores only, does not account for PPU cores.

³ 32-bit floating point operations

⁴ Max T is defined as the maximum ratio of hardware-managed thread execution contexts to simultaneously executable threads (not an absolute count of hardware-managed execution contexts). This ratio is a measure of a processor's ability to automatically hide thread stalls using hardware multithreading.