# Modern Computer Architecture

with sources that included UPenn & University of Wisconsin slides
by Joe Devietti, Milo Martin & Amir Roth , Mark Hill, Guri Sohi, Jim Smith, and David Wood

## Schedule

- Introduction and Transistors
- Parallel computing (Isaac D. Scherson, University of California, Irvine,  in October)
- ISAs
- Performance
- Pipelining Basic
- Branch Prediction
- Caches
- Virtual Memory
- Out-of-Order Execution
- Multicore multi-thread
- Vectors/GPUs for data parallelism.

## How to Compute This Fast?

- Performing the **same** operations on **many** data items
    - Example: SAXPY

```
for (I = 0; I < 1024; I++) {
  Z[I] = A*X[I] + Y[I];
}
```

```
L1: ldf [X+r1]->f1   // I is in r1
    mulf f0,f1->f2   // A is in f0
    ldf [Y+r1]->f3
    addf f2,f3->f4
    stf f4->[Z+r1}
    addi r1,4->r1
    blti r1,4096,L1
```

- Instruction-level parallelism (ILP) - fine grained
    - Loop unrolling with static scheduling –or– dynamic scheduling
    - Wide-issue superscalar (non-)scaling limits benefits
- Thread-level parallelism (TLP) - coarse grained
    - Multicore
- Can we do some “medium grained” parallelism?

# Data-Level Parallelism

- **Data-level parallelism (DLP)**
    - Single operation repeated on multiple data elements
        - SIMD (**S**ingle-**I**nstruction, **M**ultiple-**D**ata)
    - Less general than ILP: parallel insns are all same operation
    - Exploit with **vectors**
- Old idea: Cray-1 supercomputer from late 1970s
    - Eight 64-entry x 64-bit floating point “vector registers”
        - 4096 bits (0.5KB) in each register!  4KB for vector register file
    - Special vector instructions to perform vector operations
        - Load vector, store vector (wide memory operation)
        - Vector+Vector or Vector+Scalar
            - addition, subtraction, multiply, etc.
        - In Cray-1, each instruction specifies 64 operations!
    - ALUs were expensive, so one operation per cycle (not parallel)

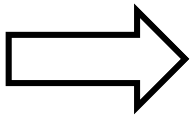
# Example Vector ISA Extensions (SIMD)

- Extend ISA with vector storage …
  - **Vector register**: fixed-size array of FP/int elements
  - **Vector length**: For example: 4, 8, 16, 64, …
- … and example operations for vector length of 4
  - Load vector: `ldf.v [X+r1]->v1`
    - `ldf [X+r1+0]->`$v1_0$
    - `ldf [X+r1+1]->`$v1_1$
    - `ldf [X+r1+2]->`$v1_2$
    - `ldf [X+r1+3]->`$v1_3$
  - Add two vectors: `addf.vv v1,v2->v3`
    - `addf` $v1_i$`,`$v2_i$`->`$v3_i$ `(where i is 0,1,2,3)`
  - Add vector to scalar: `addf.vs v1,f2,v3`
    - `addf` $v1_i$`,f2->`$v3_i$ `(where i is 0,1,2,3)`
- Today's vectors: short (128-512 bits), but fully parallel

# Example Use of Vectors – 4-wide

```
ldf  [X+r1]->f1
mulf f0,f1->f2
ldf  [Y+r1]->f3
addf f2,f3->f4
stf  f4->[Z+r1]
addi r1,4->r1
blti r1,4096,L1
```

7x1024 instructions



```
ldf.v  [X+r1]->v1
mulf.vs v1,f0->v2
ldf.v  [Y+r1]->v3
addf.vv v2,v3->v4
stf.v  v4,[Z+r1]
addi r1,16->r1
blti r1,4096,L1
```

7x256 instructions
(4x fewer instructions)

- Operations
  - Load vector: `ldf.v [X+r1]->v1`
  - Multiply vector to scalar: `mulf.vs v1,f2->v3`
  - Add two vectors: `addf.vv v1,v2->v3`
  - Store vector: `stf.v v1->[X+r1]`
- Performance?
  - Best case: 4x speedup
  - But, vector instructions don’t always have single-cycle throughput
    - Execution width (implementation) vs vector width (ISA)

## Vector Datapath & Implementatoin

- Vector insn. are just like normal insn… only "wider"
  - Single instruction fetch (no extra $N^2$ checks)
  - Wide register read & write (not multiple ports)
  - Wide execute: replicate floating point unit (same as superscalar)
  - Wide bypass (avoid $N^2$ bypass problem)
  - Wide cache read & write (single cache tag check)
- Execution width (implementation) vs vector width (ISA)
  - Example: Pentium 4 and "Core 1" executes vector ops at half width
  - "Core 2" executes them at full width
- Because they are just instructions…
  - …superscalar execution of vector instructions
  - Multiple n-wide vector instructions per cycle

## Vector Insn Sets for Different ISAs

- x86
  - Intel and AMD: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2
  - currently: AVX 512 offers 512b vectors
- PowerPC
  - AltiVEC/VMX: 128b
- ARM
  - NEON: 128b
  - Scalable Vector Extensions (SVE): up to 2048b
    - implementation is narrower than this!

# Other Vector Instructions

- These target specific domains: e.g., image processing, crypto
  - Vector reduction (sum all elements of a vector)
  - Geometry processing: 4x4 translation/rotation matrices
  - Saturating (non-overflowing) subword add/sub: image processing
  - Byte asymmetric operations: blending and composition in graphics
  - Byte shuffle/permute: crypto
  - Population (bit) count: crypto
  - Max/min/argmax/argmin: video codec
  - Absolute differences: video codec
  - Multiply-accumulate: digital-signal processing
  - Special instructions for AES encryption
- More advanced (but in Intel's Xeon Phi)
  - Scatter/gather loads: indirect store (or load) from a vector of pointers
  - Vector mask: predication (conditional execution) of specific elements

# Vector Masks (Predication)

- **Vector Masks**: 1 bit per vector element
  - Implicit predicate in all vector operations

    ```
    for (I=0; I<N; I++) if (mask_I) { vop… }
    ```
  - Usually stored in a "scalar" register (up to 64-bits)
  - Used to vectorize loops with conditionals in them

    `cmp_eq.v`, `cmp_lt.v`, etc.: sets vector predicates

```
for (I=0; I<32; I++)
   if (X[I] != 0.0) Z[I] = A/X[I];

ldf.v [X+r1] -> v1
cmp_ne.v v1,f0 -> r2       // 0.0 is in f0
divf.sv {r2} v1,f1 -> v2   // A is in f1
stf.v {r2} v2 -> [Z+r1]
```

# Scatter Stores & Gather Loads

- How to vectorize:

```
for(int i = 1, i<N, i++) {
    int bucket = val[i] / scalefactor;
    found[bucket] = 1;
```

  - Easy to vectorize the divide, but what about the store?
- Solution: hardware support for vector "scatter stores"
    - `stf.v v2->[r1+v1]`
  - Each address calculated from $r1+v1_i$

$stf\ v2_0\text{->}[r1+v1_0]$, $stf\ v2_1\text{->}[r1+v1_1]$,

$stf\ v2_2\text{->}[r1+v1_2]$, $stf\ v2_3\text{->}[r1+v1_3]$

- Vector "gather loads" defined analogously
  - `ldf.v [r1+v1]->v2`
- Scatter/gathers slower than regular vector load/store ops
  - Still provides a throughput advantage over non-vector version

## Using Vectors in Your Code

- Write in assembly
  - Ugh
- Use "intrinsic" functions and data types
  - For example: _mm_mul_ps() and "__m128" datatype
- Use vector data types
  - typedef double v2df __attribute__ ((vector_size (16)));
- Use a library someone else wrote
  - Let them do the hard work
  - Matrix and linear algebra packages
- Let the compiler do it (automatic vectorization, with feedback)
  - GCC's "-ftree-vectorize" option, -ftree-vectorizer-verbose=**n**
  - Limited impact for C/C++ code (old, hard problem)

# By the numbers: CPUs vs GPUs

| | Intel Xeon Platinum 8168 "Skylake" | Nvidia Tesla P100 | Intel Xeon Phi 7290F |
|---|---|---|---|
| frequency | 2.7 GHz | 1.3 GHz | 1.5 GHz |
| cores / threads | 24 / 48 | 56 ("3584") / 10Ks | 72 / 288 |
| RAM | 768 GB | 16 GB | 384 GB |
| DP TFLOPS | 1.0 | 4.7 | 3.5 |
| Transistors | >5B ? | 15.3B | >5B ? |
| Price | $5,900 | $6,000 | $3,400 |

## GPUs and SIMD/Vector Data Parallelism

- How do GPUs have such high peak FLOPS & FLOPS/Joule?
  - Exploit massive data parallelism – focus on total throughput
  - Remove hardware structures that accelerate single threads
  - Specialized for graphics: e.g., data-types & dedicated texture units
- “SIMT” execution model
  - Single instruction multiple threads
  - Similar to both “vectors” and “SIMD”
  - A key difference: better support for conditional control flow
- Program it with CUDA or OpenCL
  - Extensions to C/C++
  - Perform a “shader task” (a snippet of scalar computation) over many elements
  - Internally, GPU uses scatter/gather and vector mask operations

- following slides “Beyond Programmable Shading” course

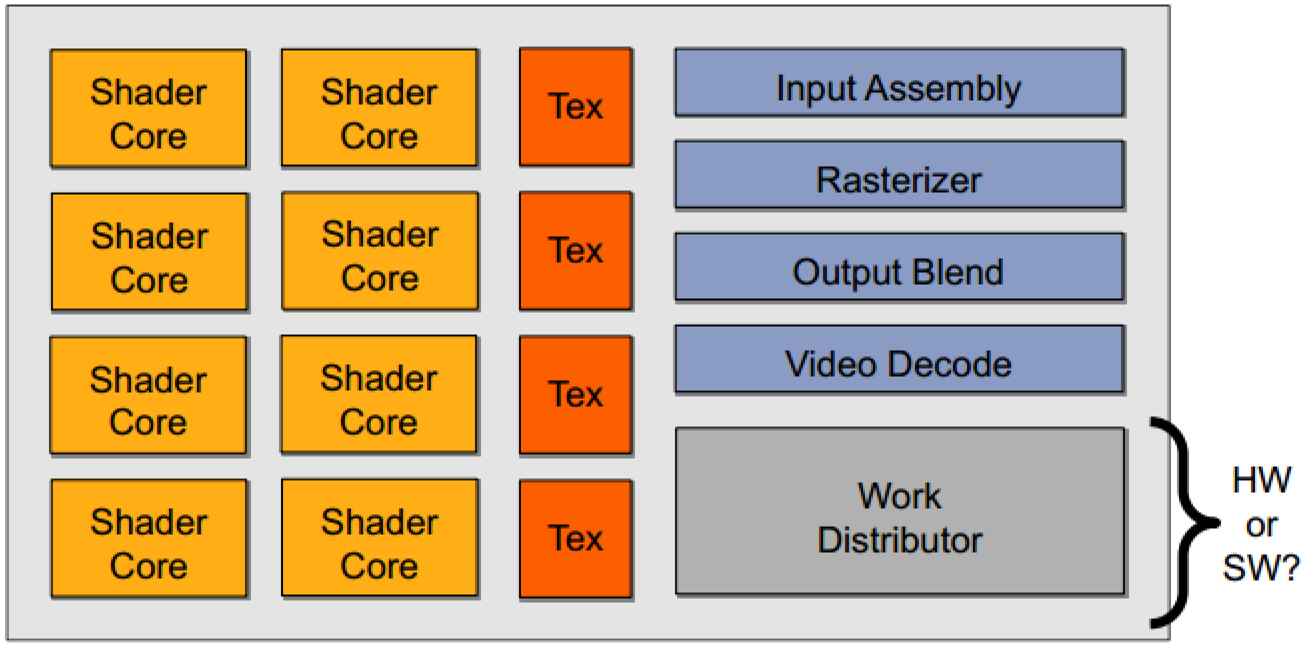
**“Real-Time Rendering Architectures”**

**http://bps11.idav.ucdavis.edu/**

Mike Houston, AMD

## What's in a GPU?

A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)



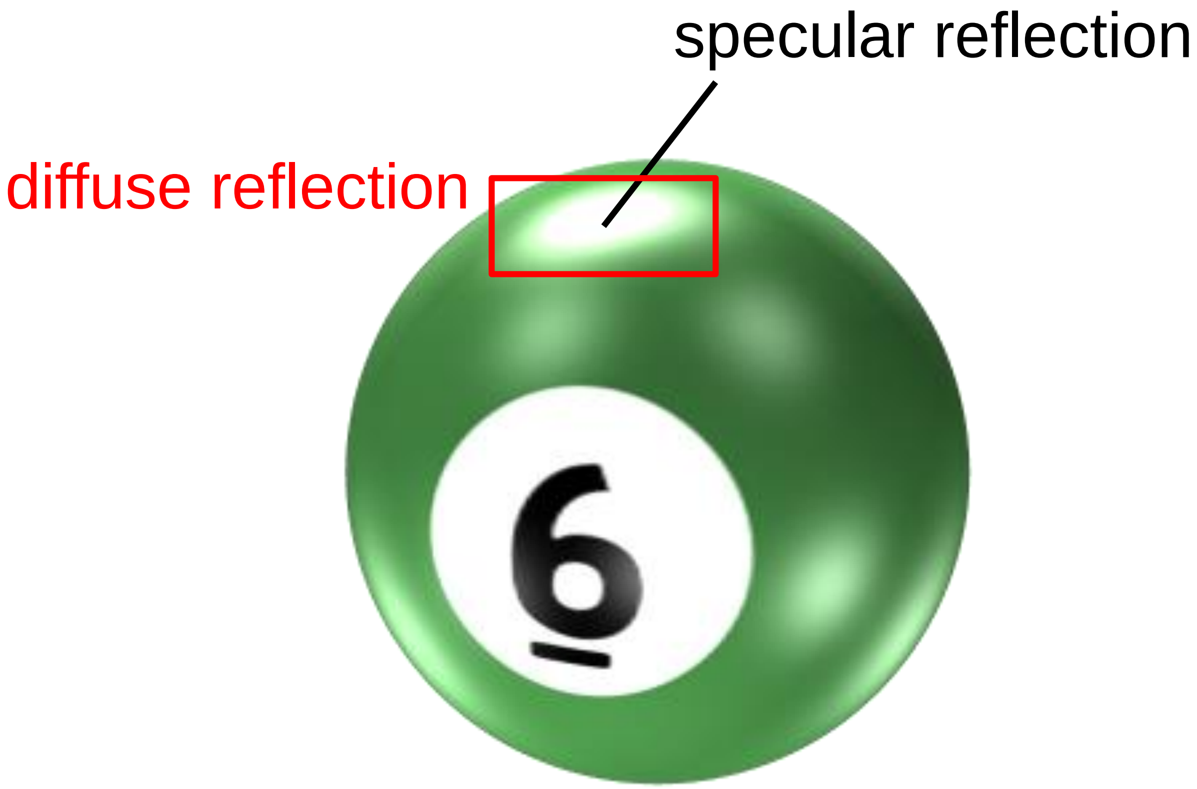
## A diffuse reflectance shader

```
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);
}
```

Shader programming model:

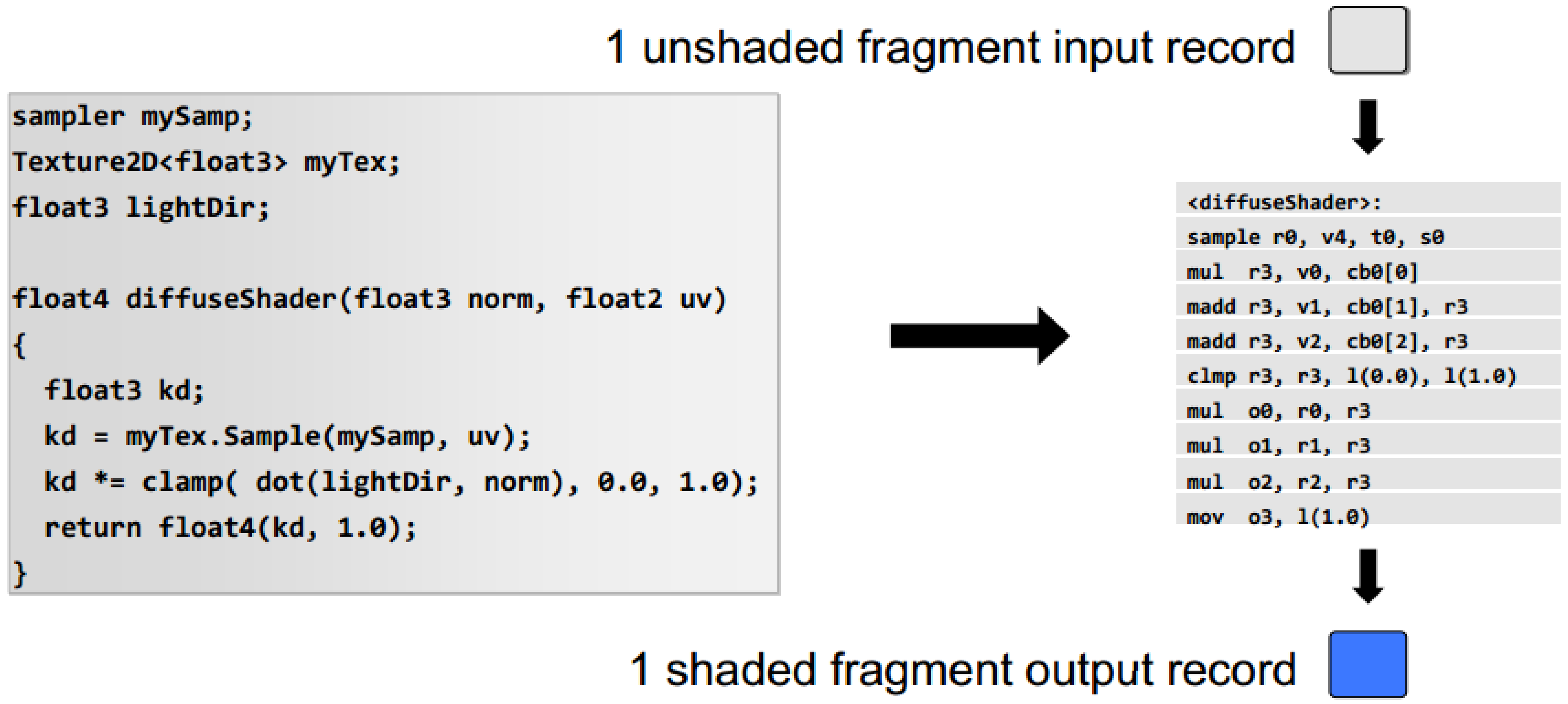
Fragments are processed independently,
but there is no explicit parallel programming

# diffuse reflection example



c/o http://www.iconarchive.com/show/pool-ball-icons-by-barkerbaggies/Ball-6-icon.html

## Compile shader



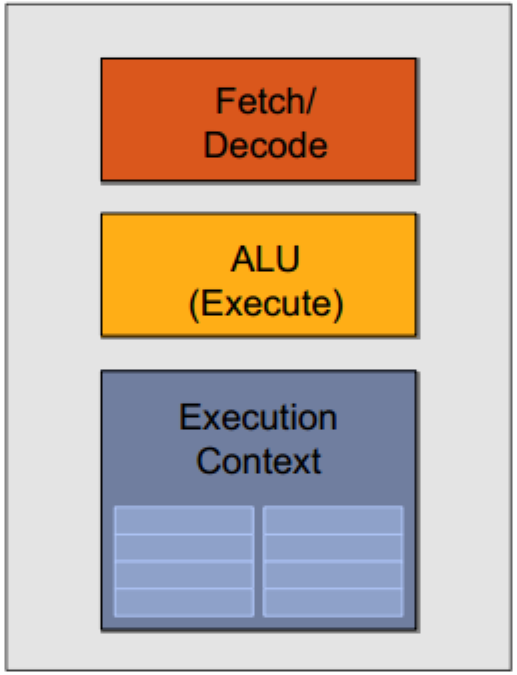
1 unshaded fragment input record

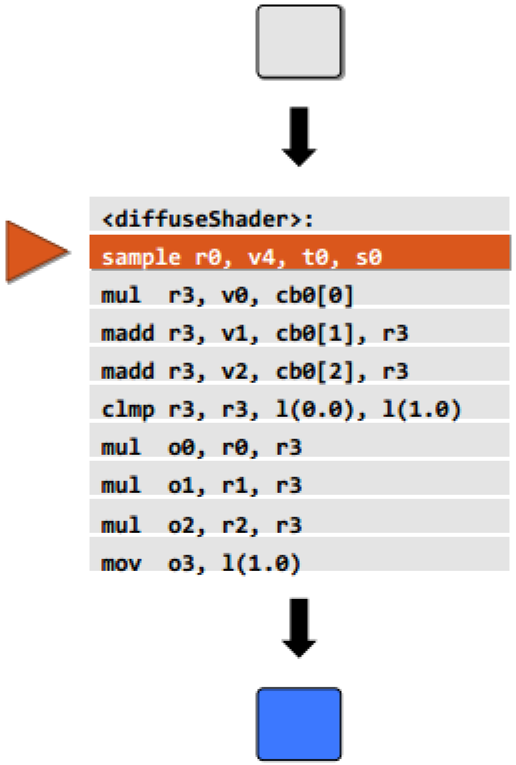
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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

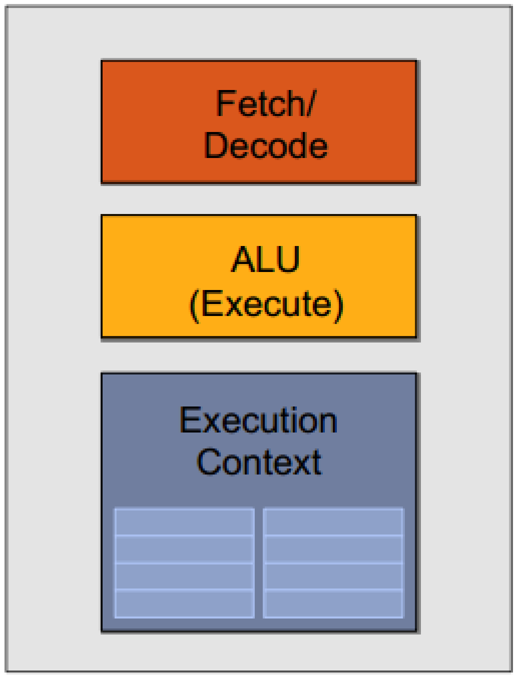
## Execute shader

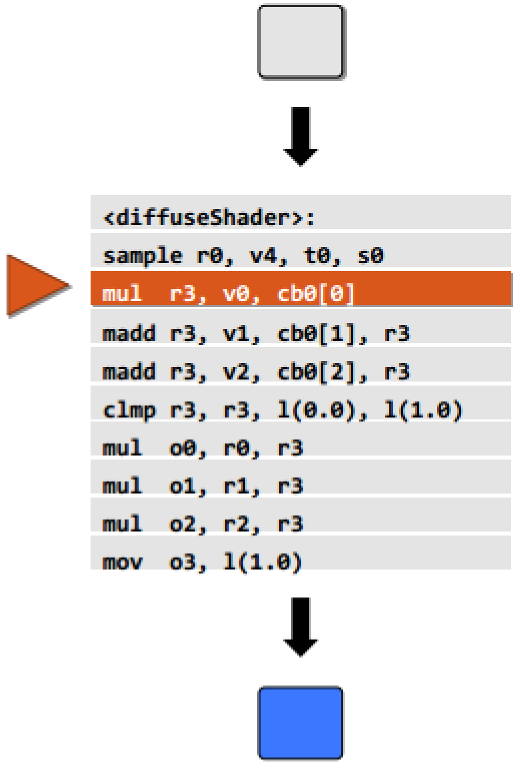




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

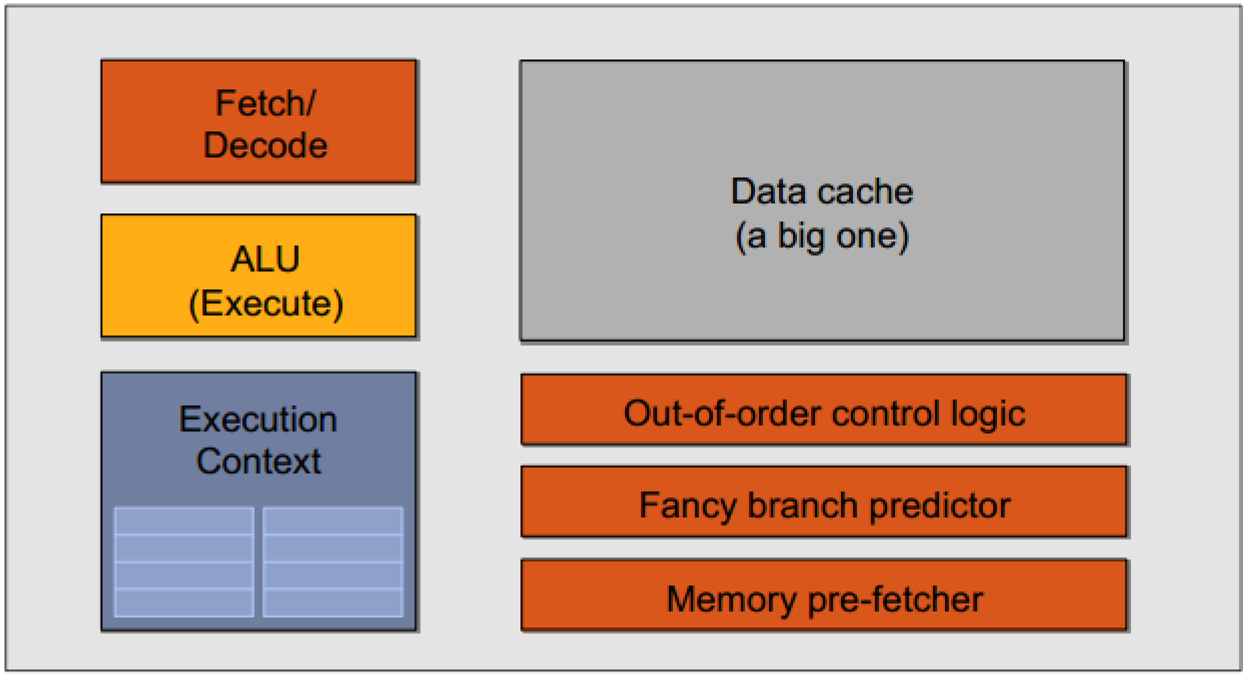
## Execute shader



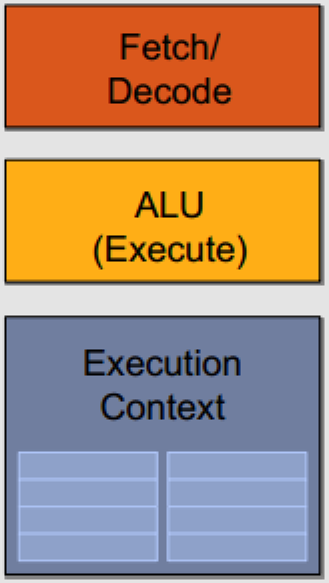


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

## “CPU-style” cores



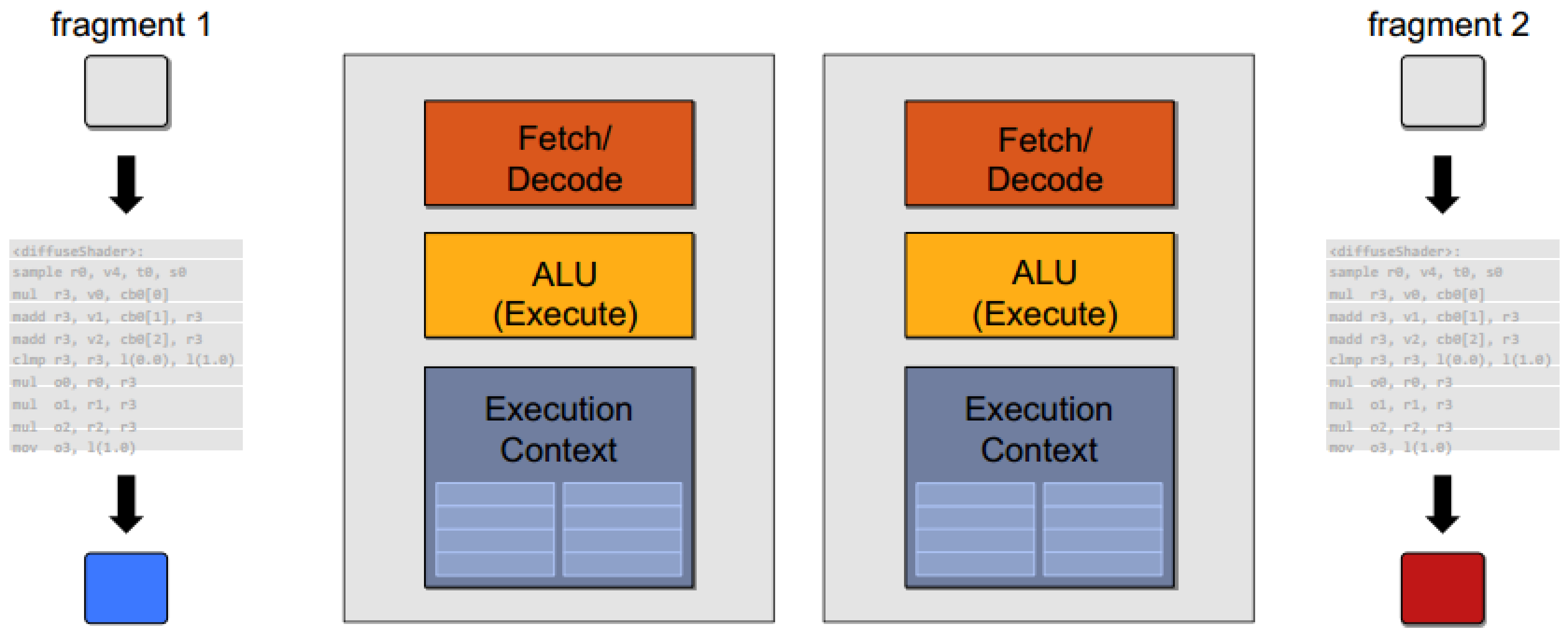
## Slimming down



Idea #1:

Remove components that help a single instruction stream run fast

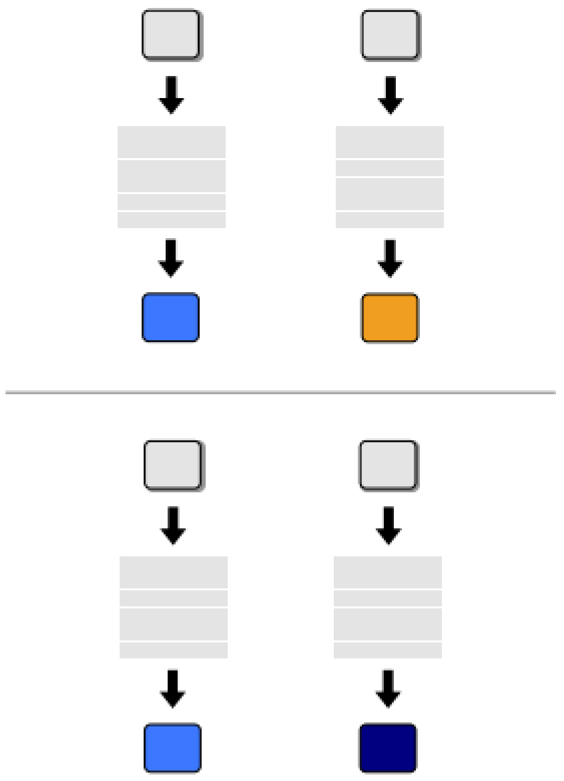
## Two cores (two fragments in parallel)

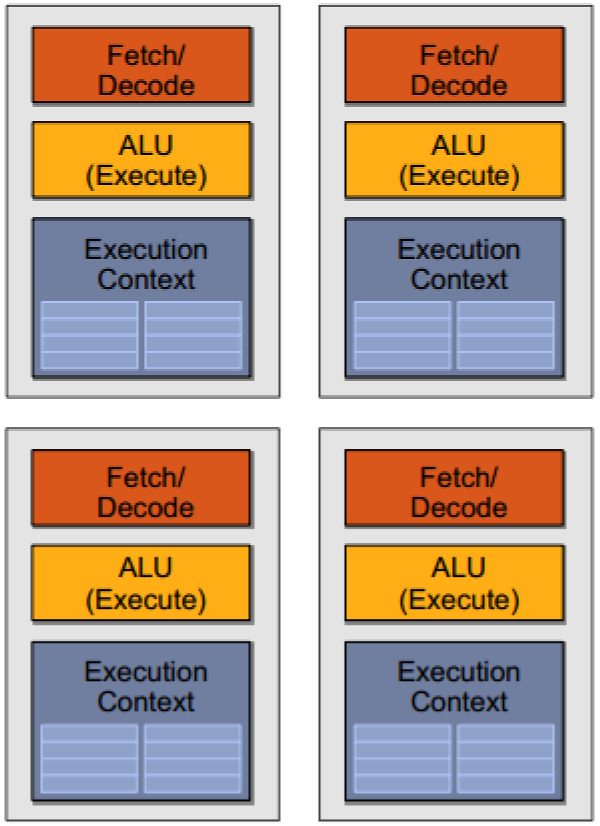


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

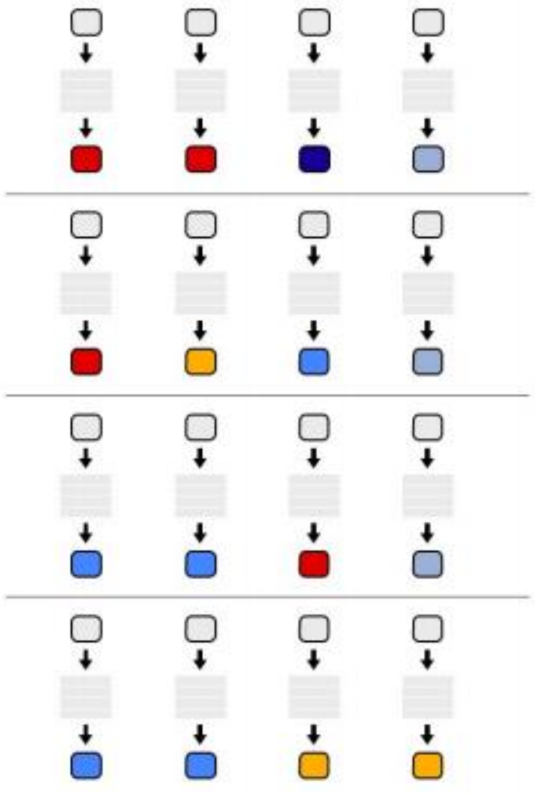
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

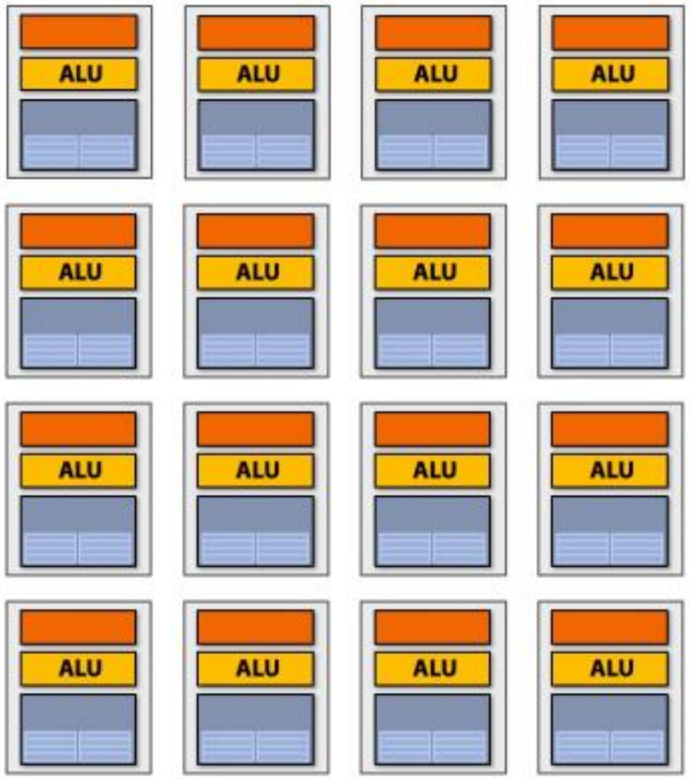
## Four cores (four fragments in parallel)





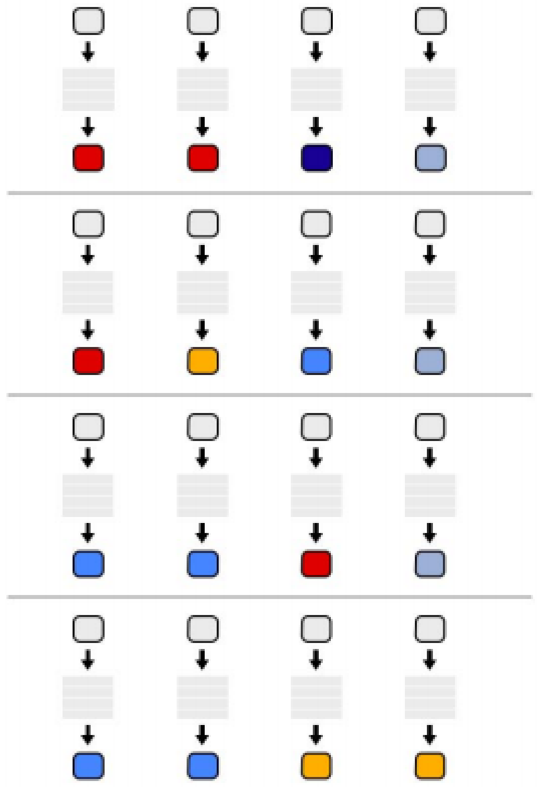
# Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

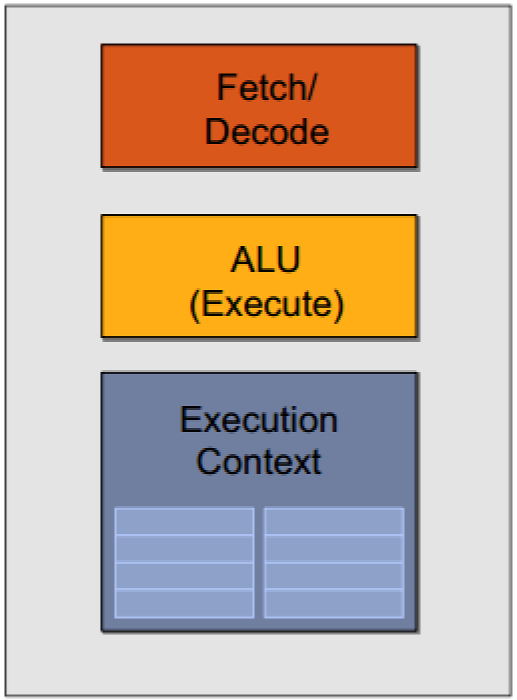
## Instruction stream sharing



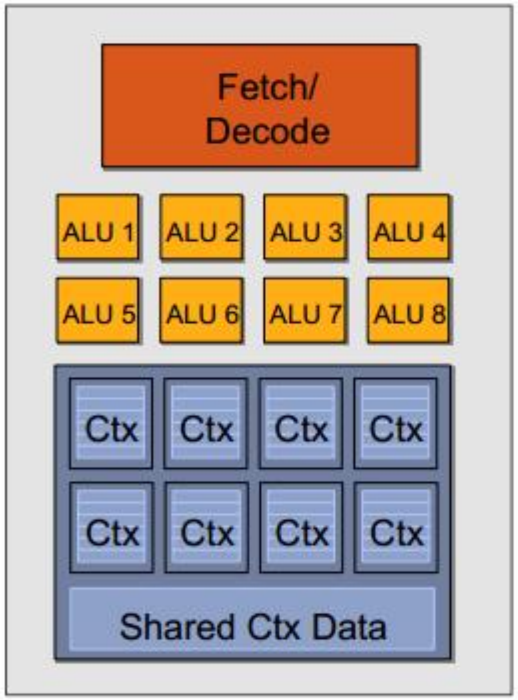
But ... many fragments should be able to share an instruction stream!

```
<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



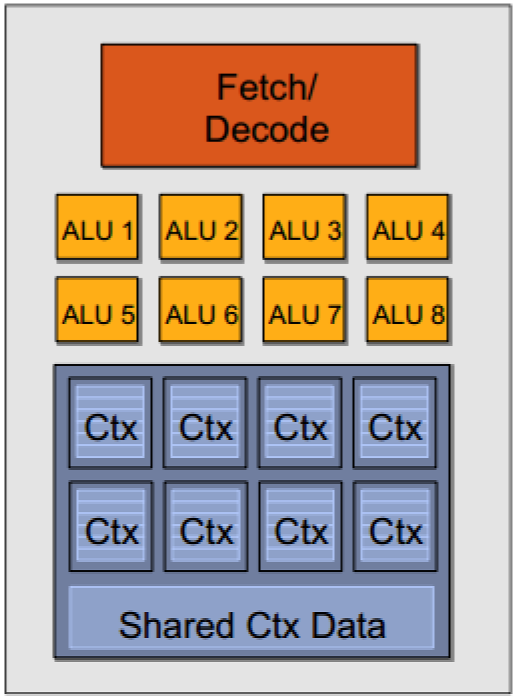
## Add ALUs

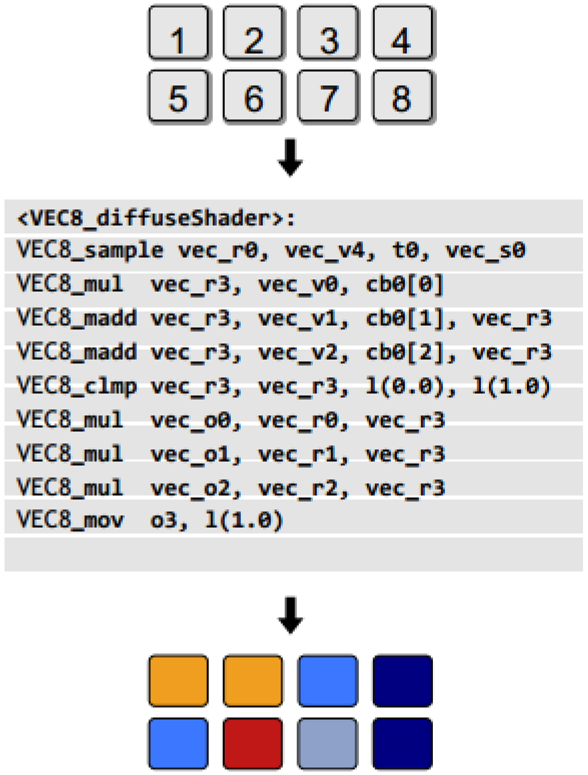


Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

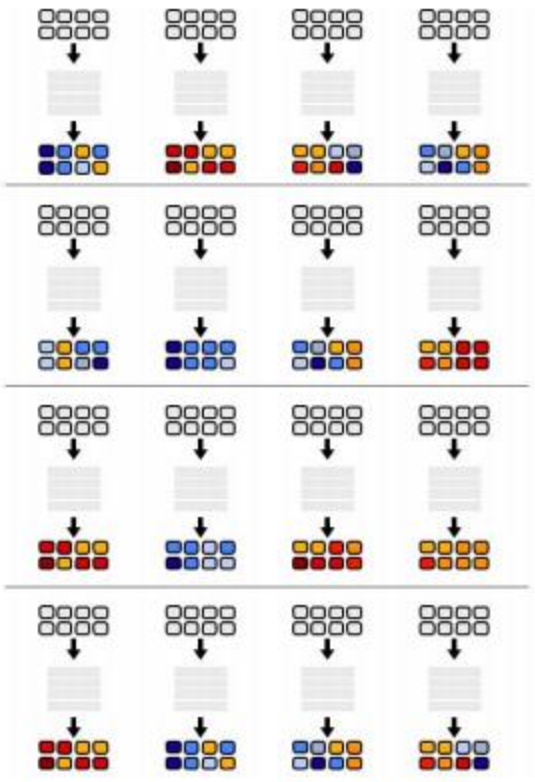
## Modifying the shader



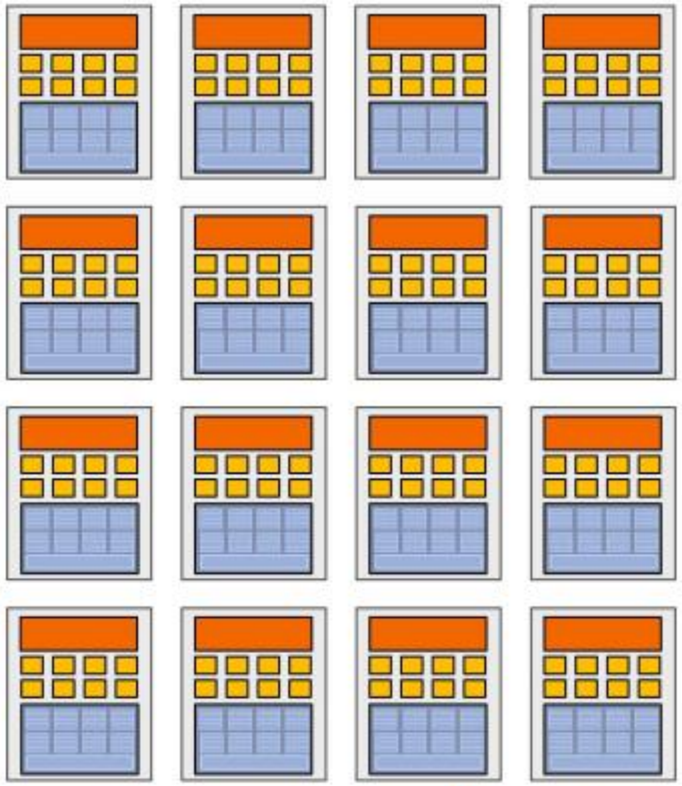


```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  o3, l(1.0)
```

## 128 fragments in parallel

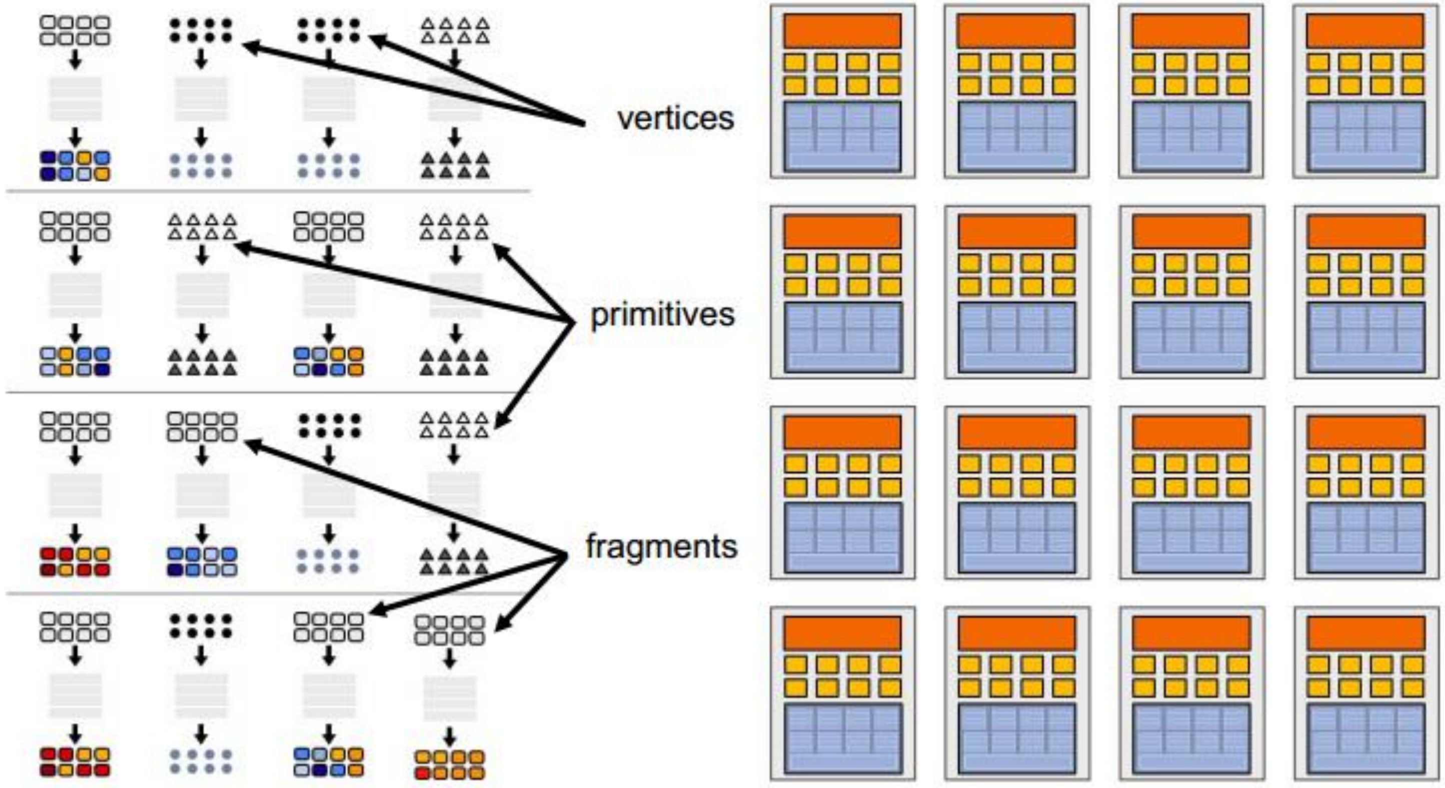


16 cores = 128 ALUs

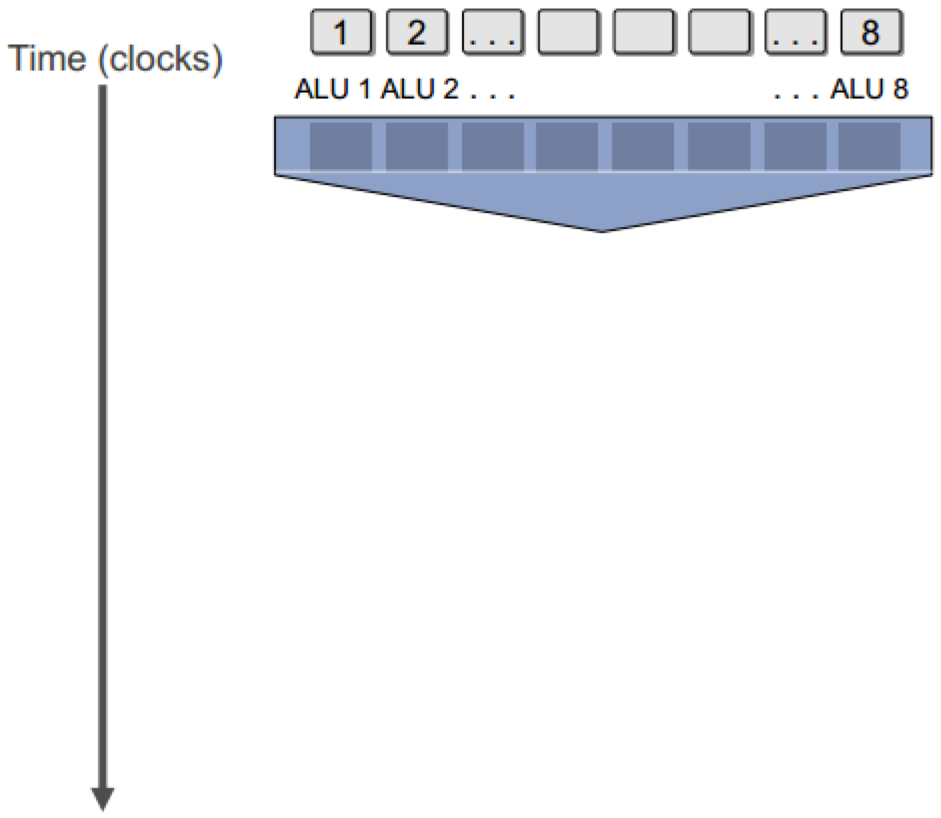


, 16 simultaneous instruction streams

## 128 [ vertices/fragments primitives OpenCL work items ] in parallel

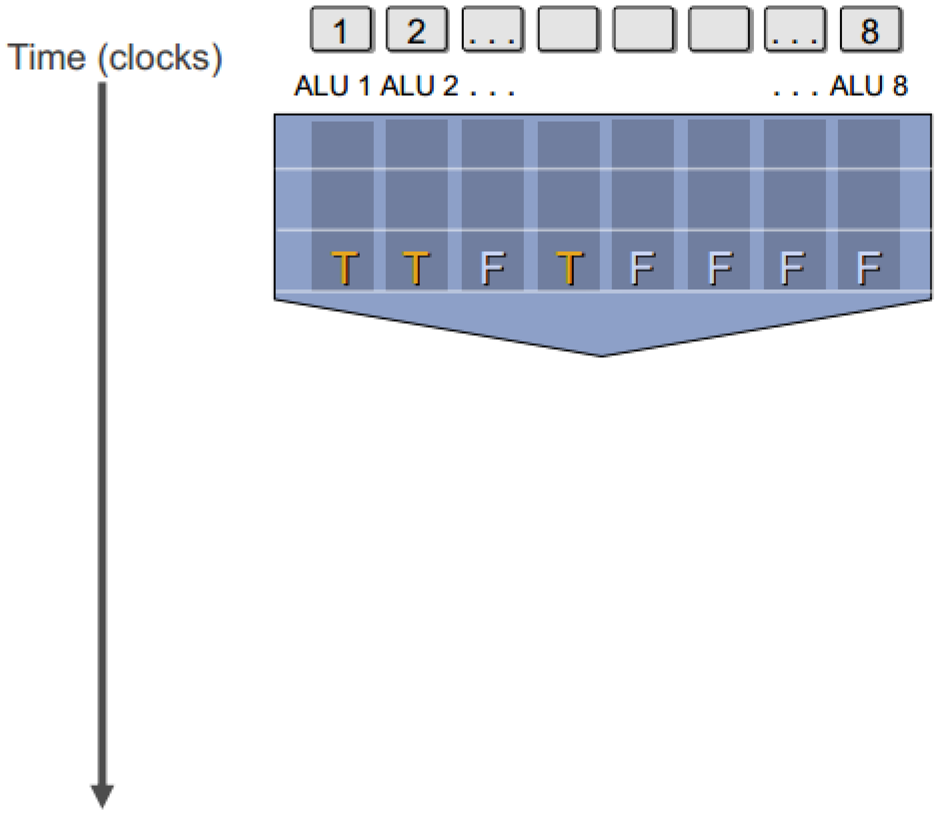


## But what about branches?



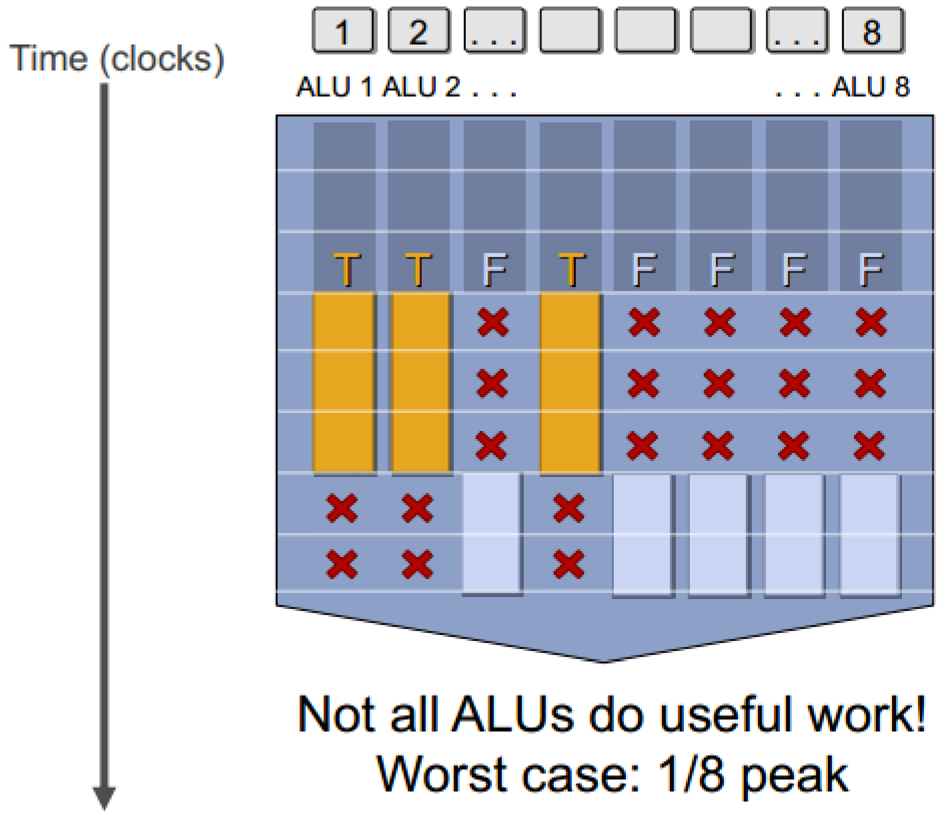
```
<unconditional
 shader code>

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?



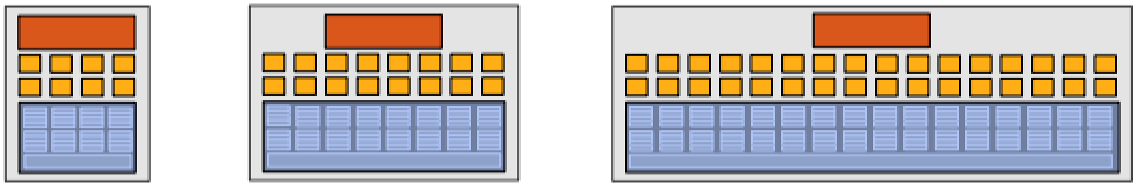
```
<unconditional
 shader code>

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

<resume unconditional
 shader code>
```

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

## Clarification

SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
  - x86 SSE, AVX, 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 (“wavefronts”)



In practice: 16 to 64 fragments share an instruction stream.

## SIMD vs SIMT

- SIMD: single insn multiple **data**
  - write 1 insn that operates on a vector of data
  - handle control flow via explicit masking operations
- SIMT: single insn multiple **thread**
  - write 1 insn that operates on scalar data
  - each of many threads runs this insn
  - compiler+hw aggregate threads into groups that execute on SIMD hardware
  - compiler+hw handle masking for control flow

## 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

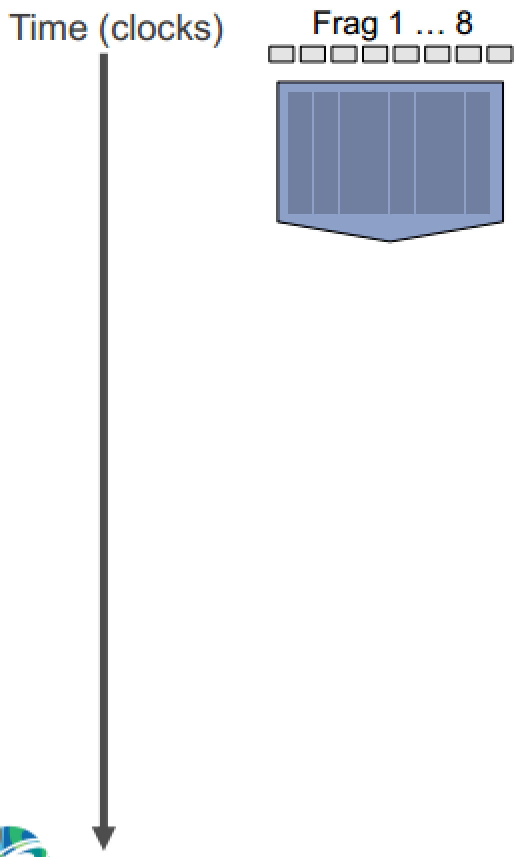
We’ve removed the fancy caches and logic that helps avoid stalls.

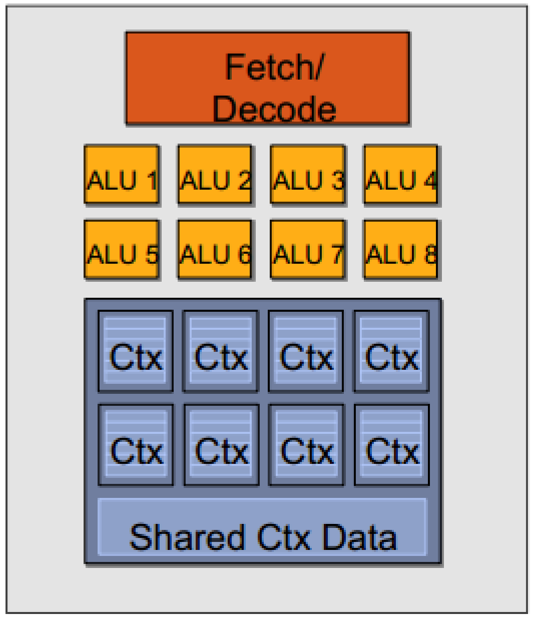
But we have LOTS of independent fragments.

## Idea #3:

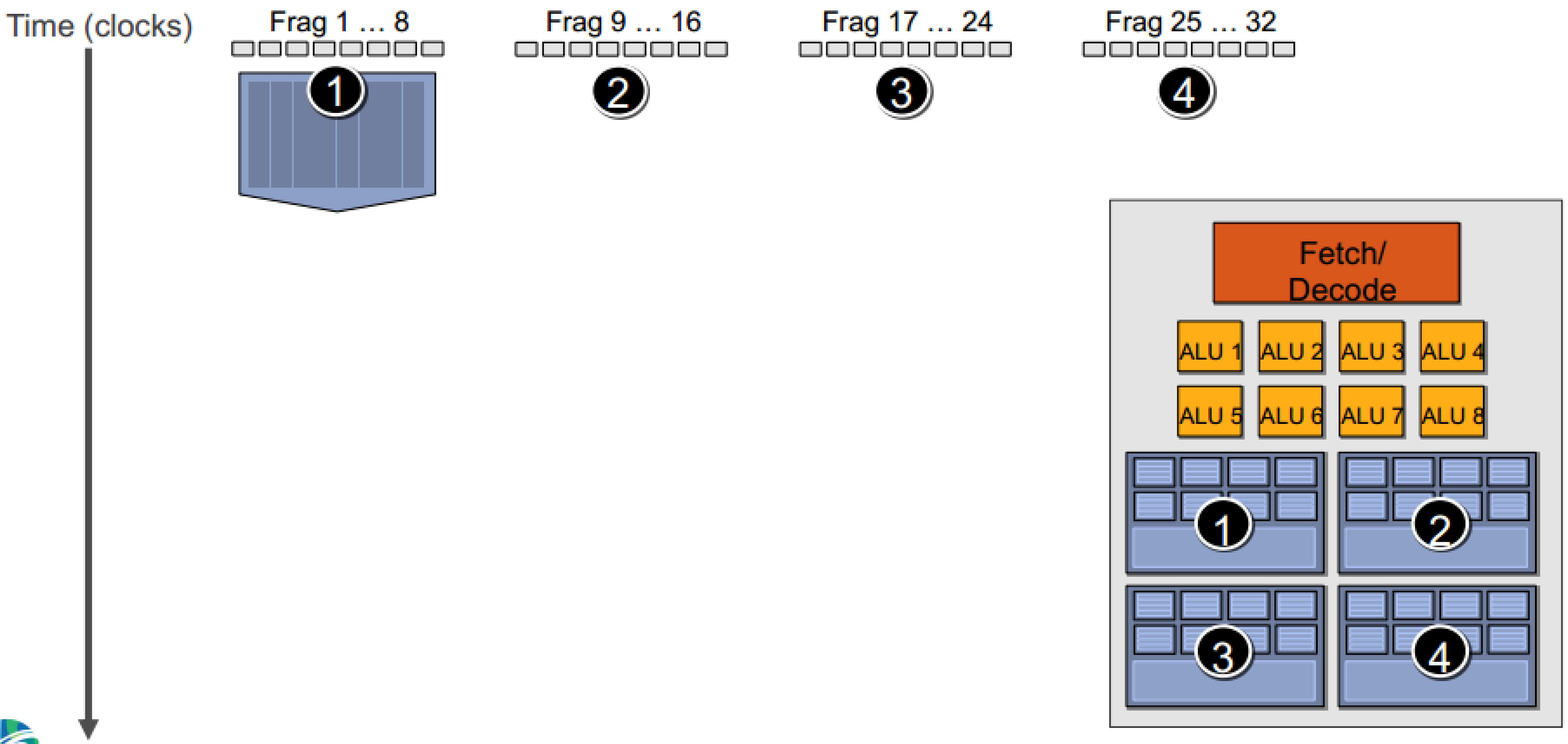
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

## Hiding shader stalls

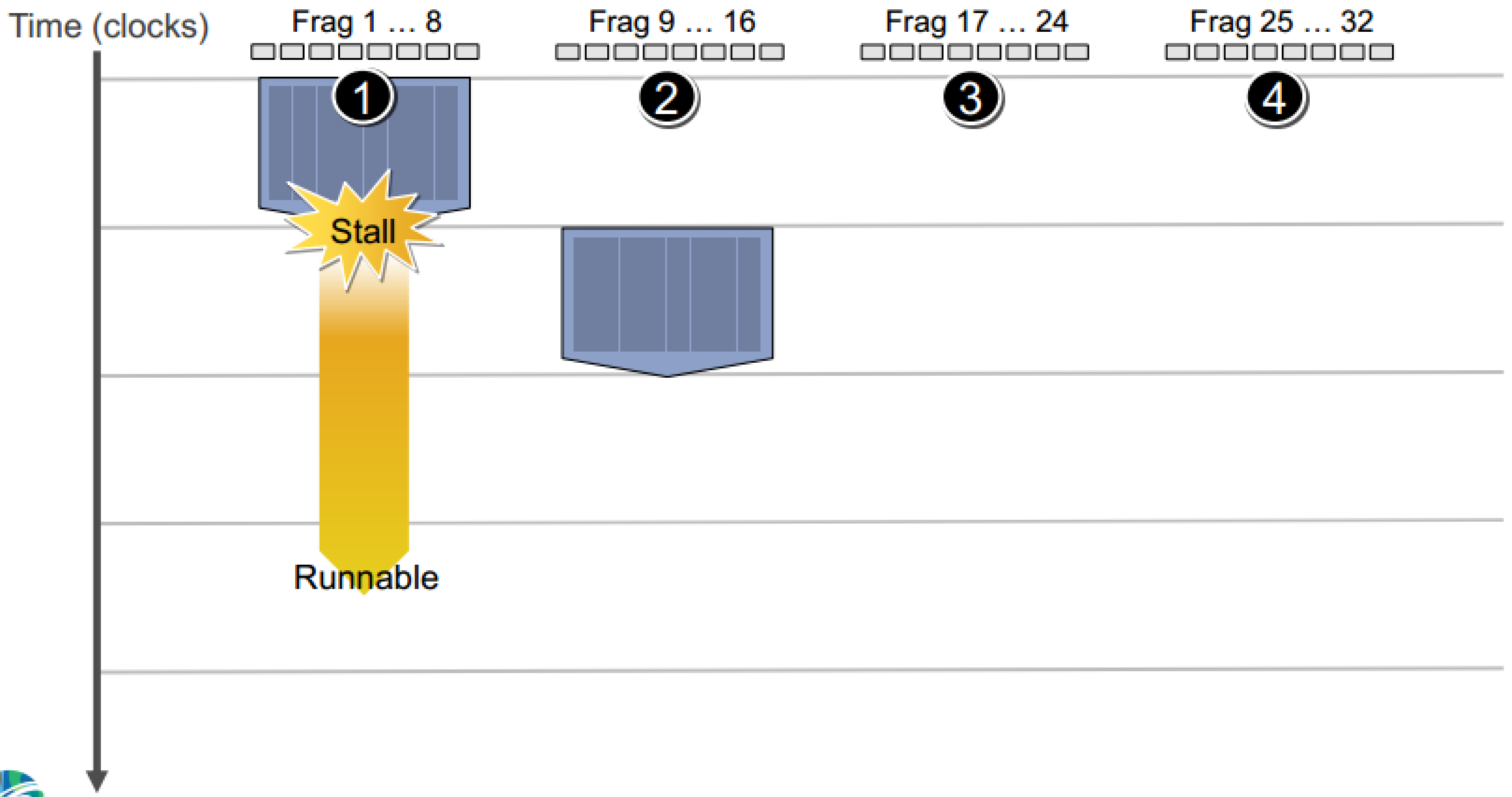




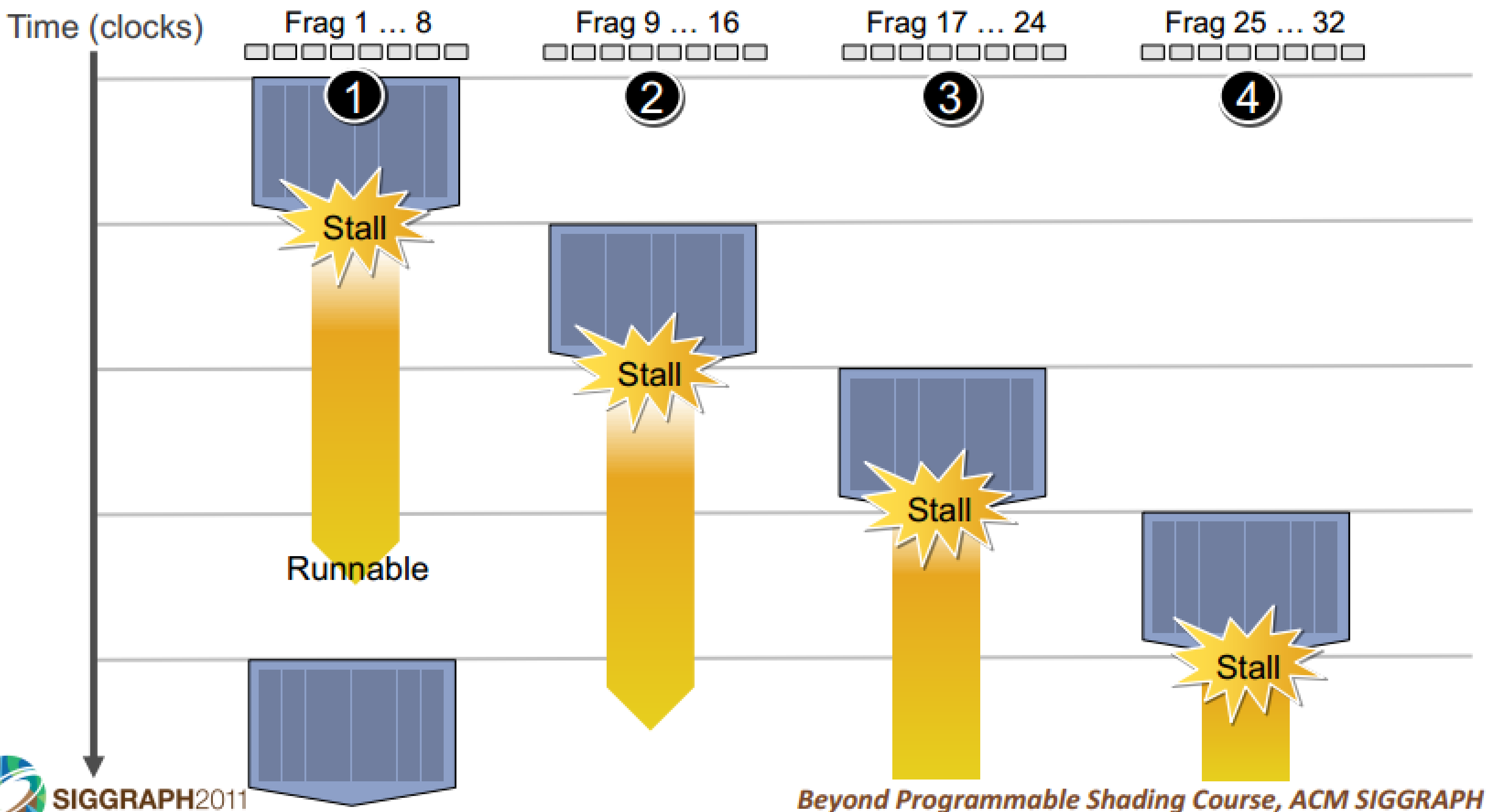
# Hiding shader stalls



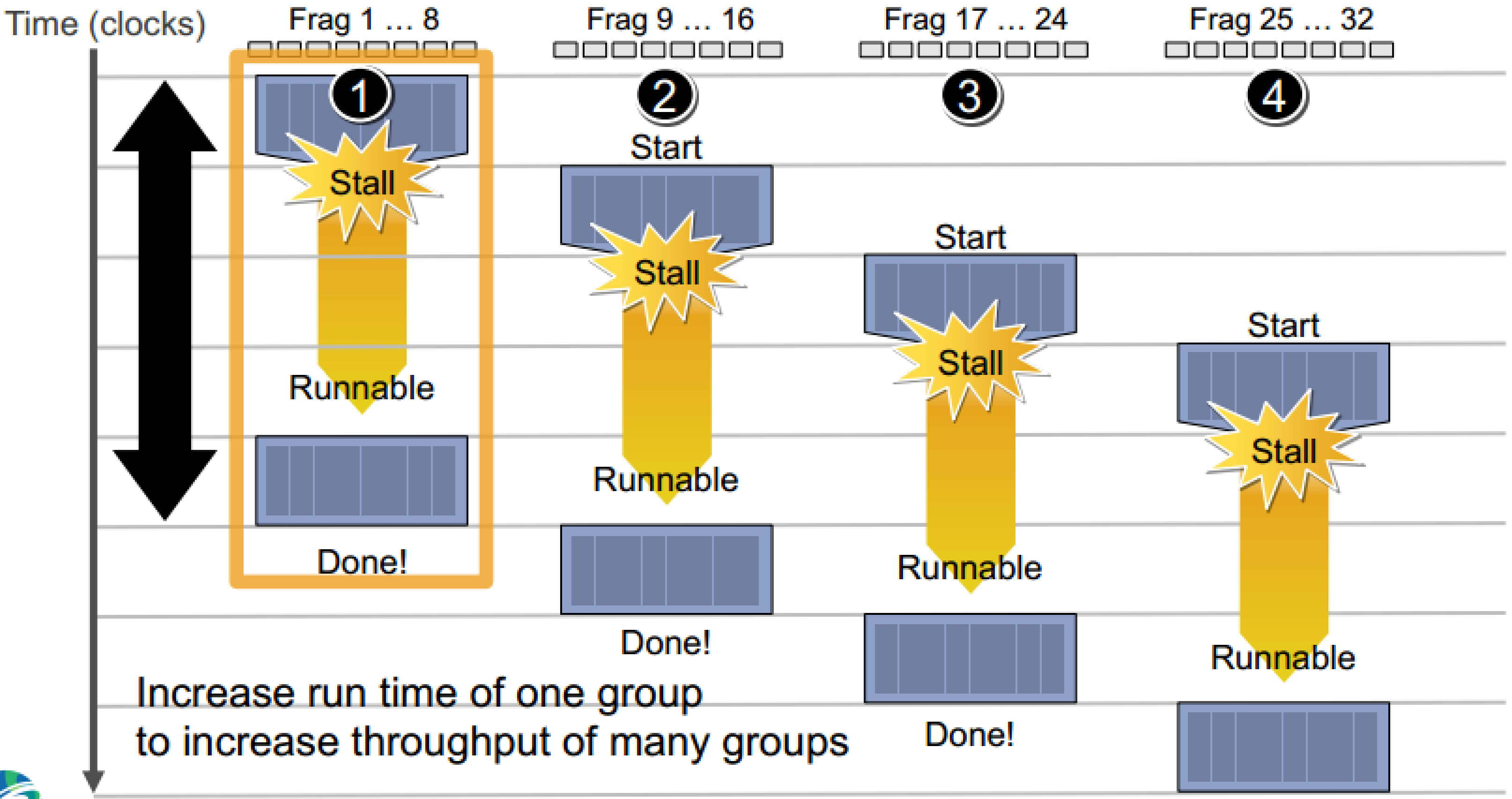
# Hiding shader stalls



## Hiding shader stalls

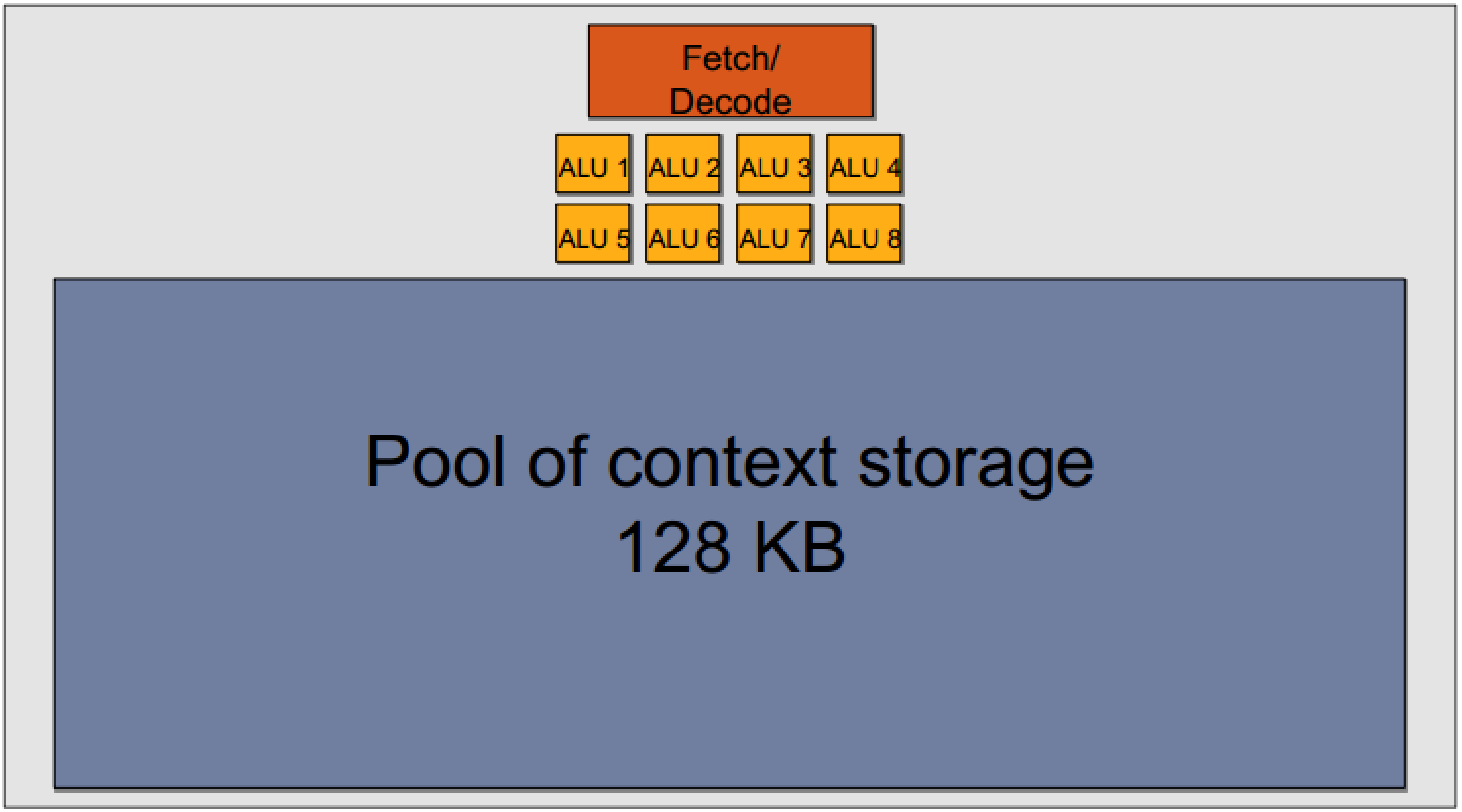


## Throughput!

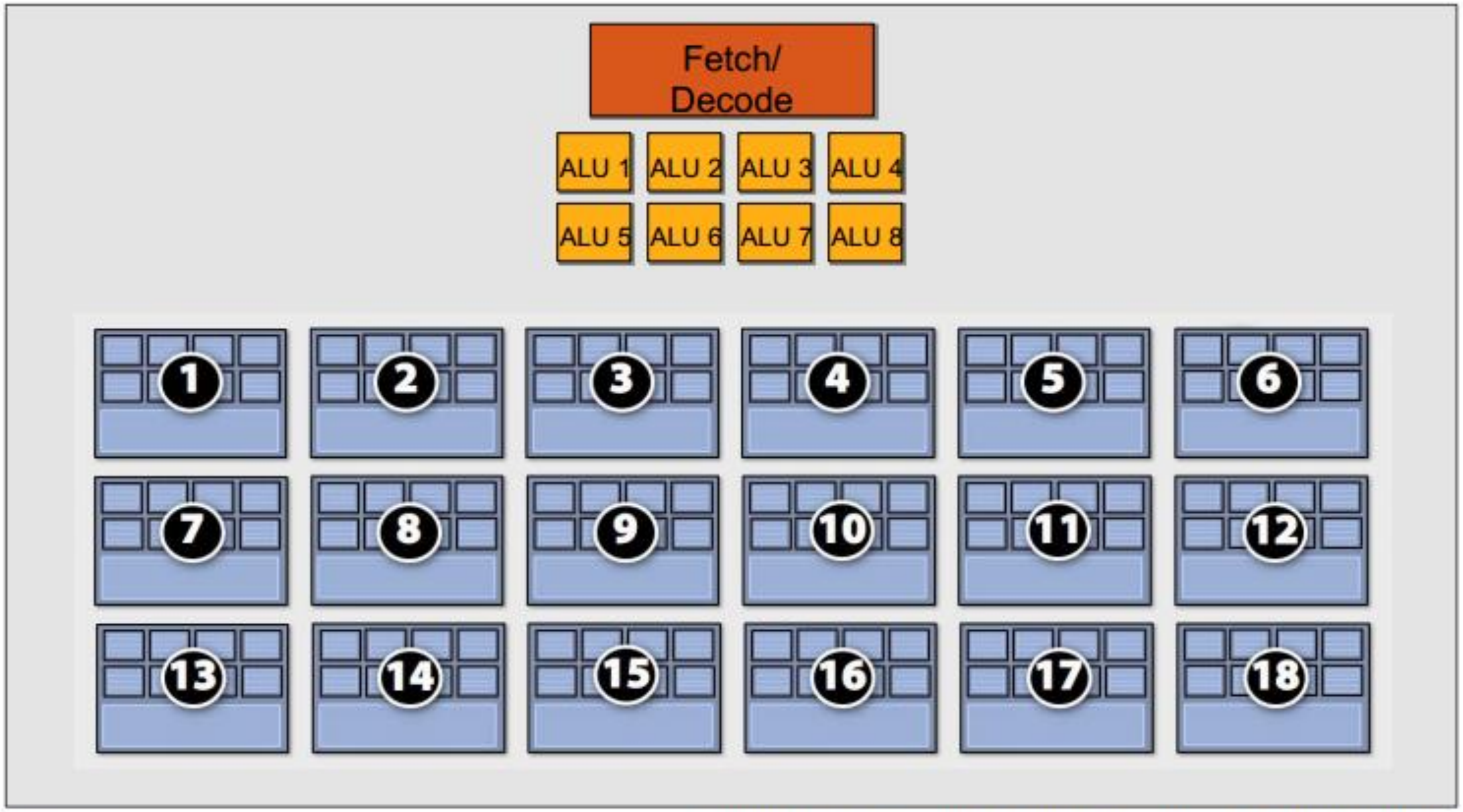


Increase run time of one group
to increase throughput of many groups

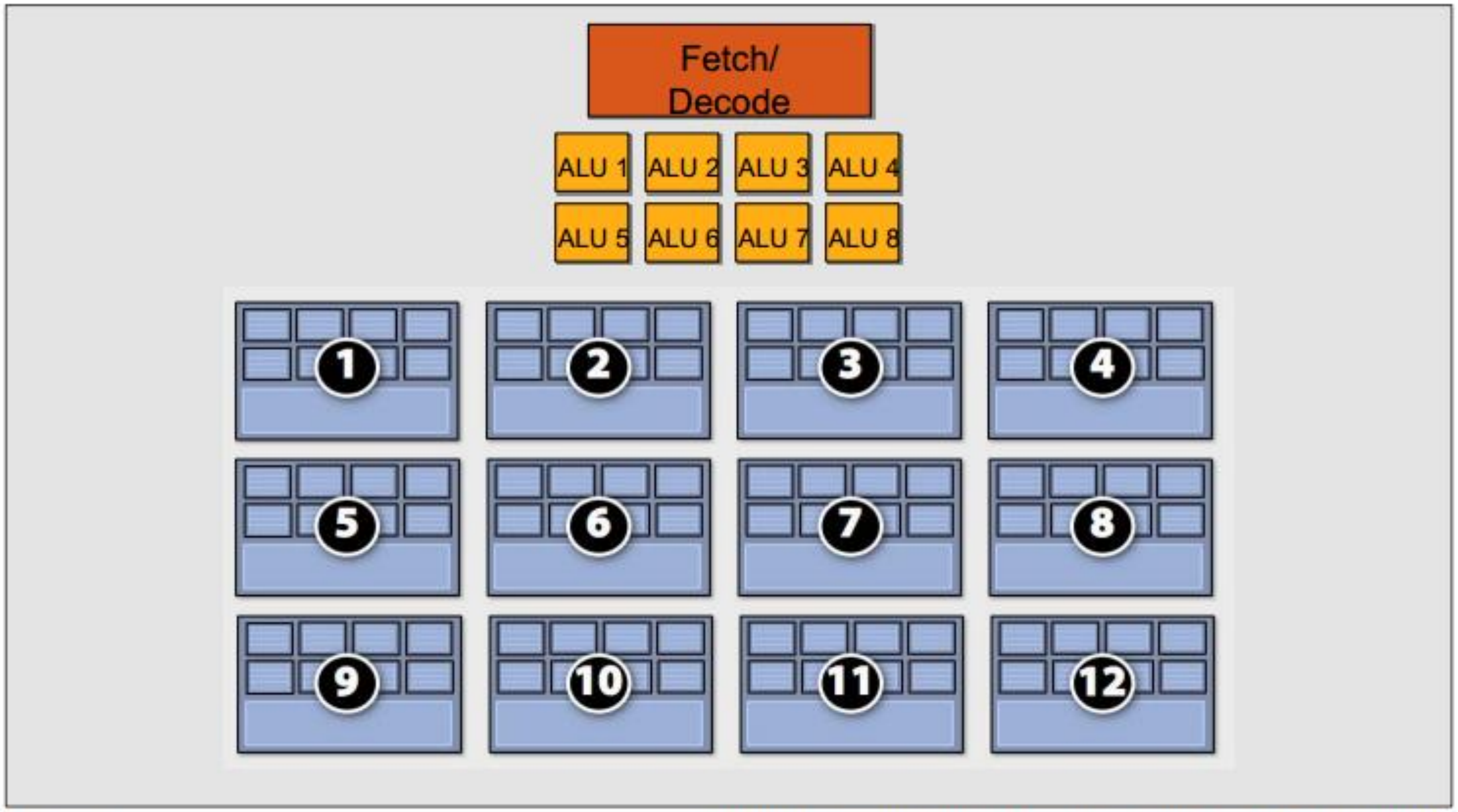
## Storing contexts



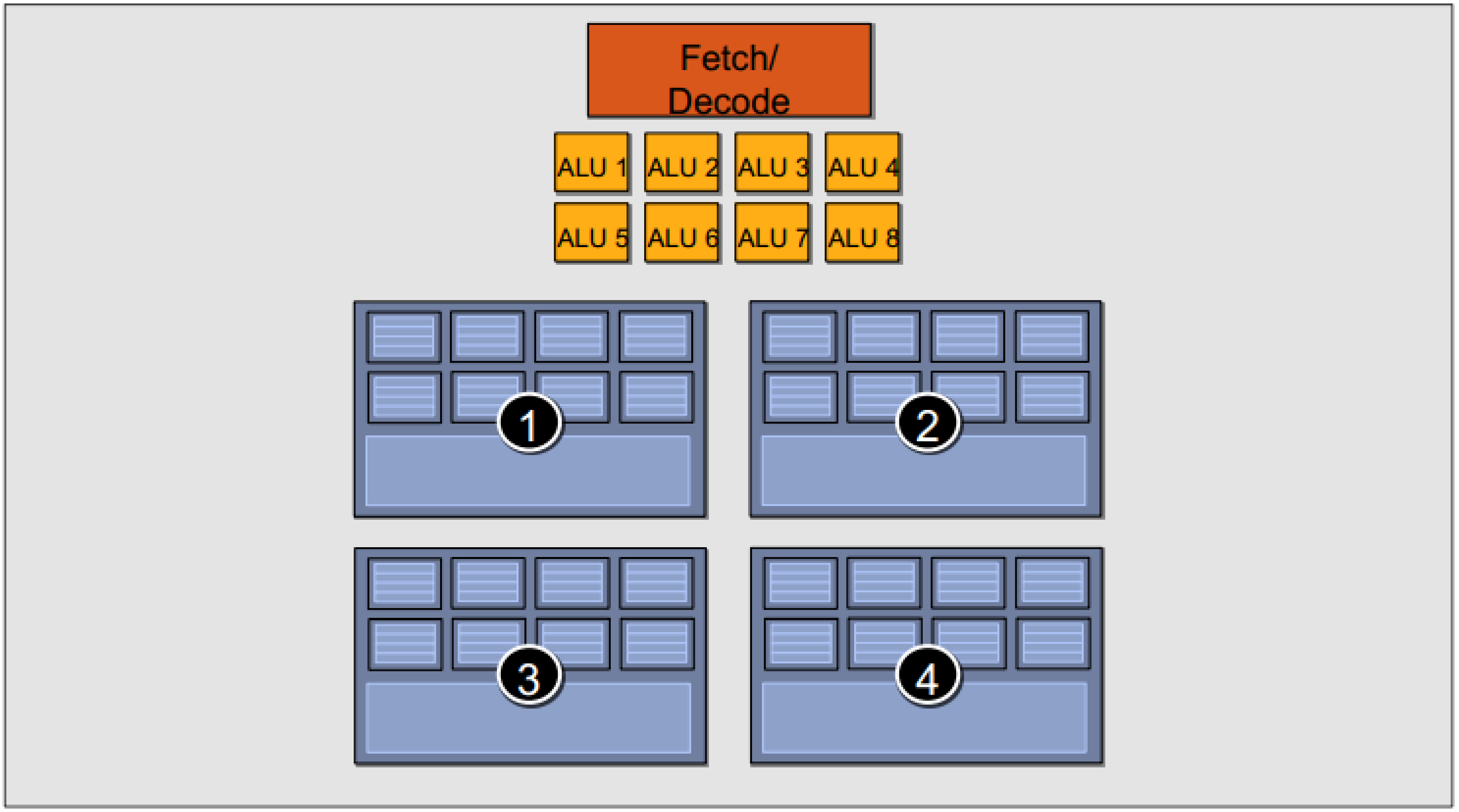
## Eighteen small contexts (maximal latency hiding)



## Twelve medium contexts



## Four large contexts (low latency hiding ability)



## Clarification

Interleaving between contexts can be managed by hardware or software (or both!)

- NVIDIA / ATI Radeon GPUs
  - HW schedules / manages all contexts (lots of them)
  - Special on-chip storage holds fragment state
- Intel Larrabee
  - HW manages four x86 (big) contexts at fine granularity
  - SW scheduling interleaves many groups of fragments on each HW context
  - L1-L2 cache holds fragment state (as determined by SW)

# Example chip

16 cores

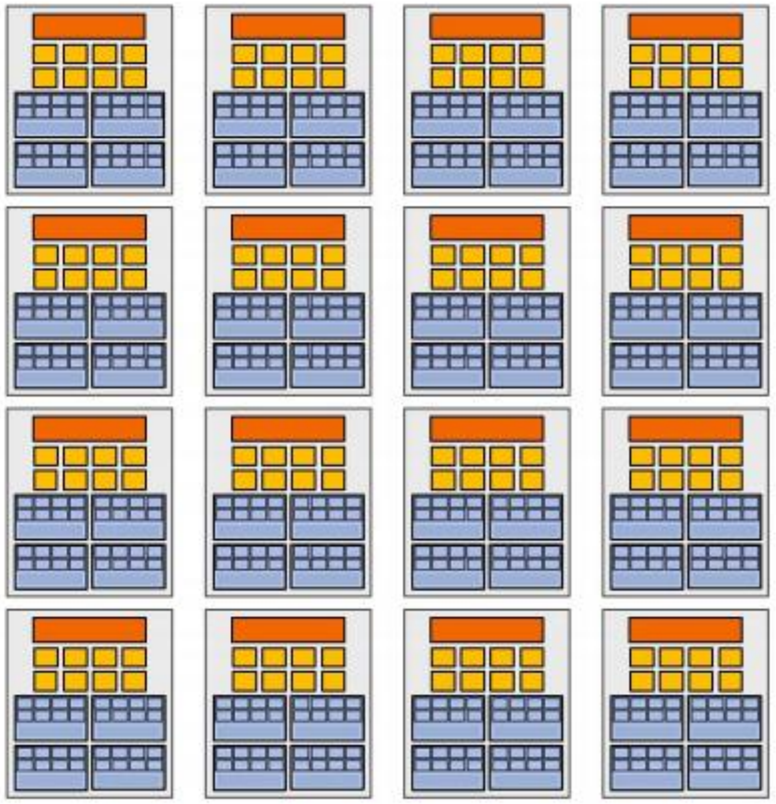
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



## 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

# Data Parallelism Summary

- Data Level Parallelism
  - "medium-grained" parallelism between ILP and TLP
  - Still one flow of execution (unlike TLP)
  - Compiler/programmer must explicitly expresses it (unlike ILP)
- Hardware support: new "wide" instructions (SIMD)
  - Wide registers, perform multiple operations in parallel
- Trends
  - Wider: 64-bit (MMX, 1996), 128-bit (SSE2, 2000), 256-bit (AVX, 2011), 512-bit (Xeon Phi, 2013)
  - More advanced and specialized instructions
- GPUs
  - Embrace data parallelism via "SIMT" execution model
  - Becoming more programmable all the time
- Today's chips exploit parallelism at **all** levels: ILP, DLP, TLP