Lect. 10: Vector and SIMD Processors

- Many real-world problems, especially in science and engineering, map well to computation on arrays
- RISC approach is inefficient:
  - Based on loops → require dynamic or static unrolling to overlap computations
  - Indexing arrays based on arithmetic updates of induction variables
  - Fetching of array elements from memory based on individual, and unrelated, loads and stores
  - Instruction dependences must be identified for each individual instruction
- Idea:
  - Treat operands as whole vectors, not as individual integer or float-point numbers
  - Single machine instruction now operates on whole vectors (e.g., a vector add)
  - Loads and stores to memory also operate on whole vectors
  - Individual operations on vector elements are independent and only dependences between whole vector operations must be tracked

Execution Model

```
for (i=0; i<64; i++)
        a[i] = b[i] + s;
```

- Straightforward RISC code:
  - F2 contains the value of s
  - R1 contains the address of the first element of a
  - R2 contains the address of the first element of b
  - R3 contains the address of the last element of a + 8

```
loop: L.D    F0,0(R2)   ;F0=array element of b
      ADD.D  F4,F0,F2   ;main computation
      S.D    F4,0(R1)   ;store result
      DADDUI R1,R1,8    ;increment index
      DADDUI R2,R2,8    ;increment index
      BNE    R1,R3,loop ;next iteration
```

Execution Model

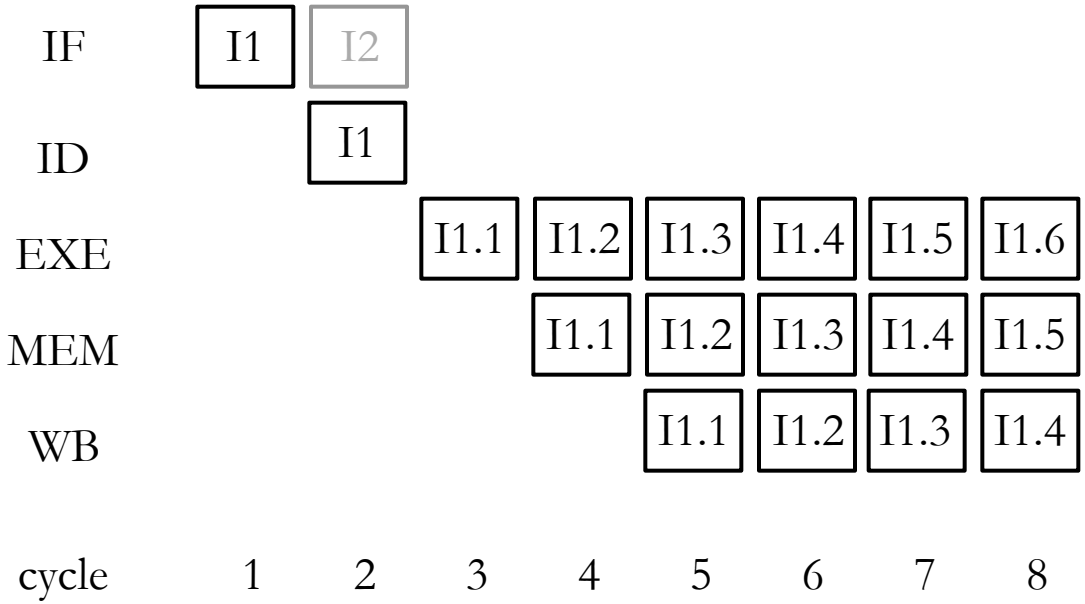
```
for (i=0; i<64; i++)
        a[i] = b[i] + s;
```

- Straightforward vector code:
  - F2 contains the value of s
  - R1 contains the address of the first element of a
  - R2 contains the address of the first element of b
  - Assume vector registers have 64 double precision elements

```
LV      V1,R2 ;          V1=array b
ADDVS.D V2,V1,F2 ;  main computation
SV      V2,R1;           store result
```

  - Notes:
    - In practice vector registers are not of the exact size of the arrays
    - Only 3 instructions executed compared to 6*64=384 executed in the RISC

Execution Model (Pipelined)



- In practice, the vector units takes several cycles to operate on each element, but is pipelined
- With multiple vector units, I2 can execute together with I1

Pros of Vector Processors

- Reduced pressure on instruction fetch
  - Fewer instructions are necessary to specify the same amount of work
- Reduced pressure on instruction issue
  - Reduced number of branches alleviates branch prediction
  - Much simpler hardware for checking dependences
- More streamlined memory accesses
  - Vector loads and stores specify a regular access pattern
  - High latency of initiating memory access is amortized

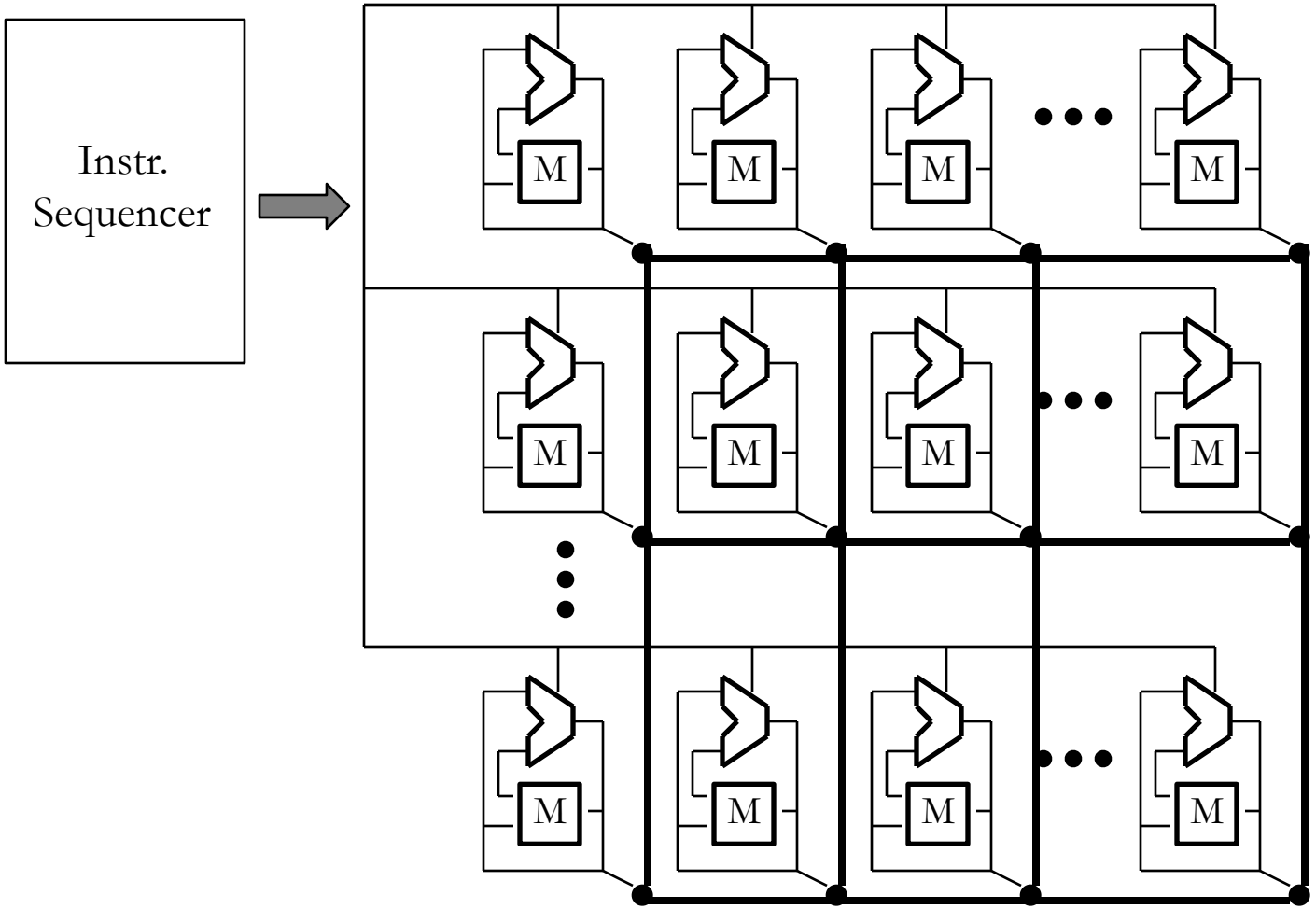
Cons of Vector Processors

- Still requires a traditional scalar unit (integer and FP) for the non-vector operations
- Difficult to maintain precise interrupts (can't rollback all the individual operations already completed)
- Compiler or programmer has to vectorize programs
- Not suitable/efficient for many different classes of applications
- Requires a specialized, high-bandwidth, memory system
  - Usually built around heavily banked memory with data interleaving

SIMD Processors: Original Idea

- Network of simple processing elements (PE)
  - PEs operate in lockstep under the control of a master sequencer
  - PEs can exchange results with a small number of neighbours via special data-routing instructions
  - Each PE has its own local memory
  - Very large (up to 64K) number of PEs
  - Usually operated as co-processors with a host computer to perform I/O and to handle external memory
- Intended for use as supercomputers
- Programmed via custom extensions of common HLL

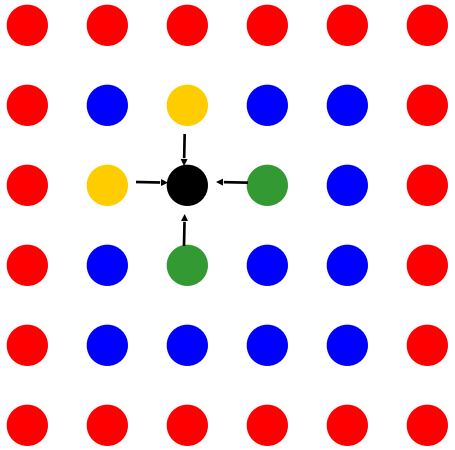
Original SIMD Idea



Example: Equation Solver Kernel

- The problem:
  - Operate on a (n+2)x(n+2) matrix

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$



- SIMD implementation:
  - Assign one node to each PE
  - Step 1: all PE's send their data to their east neighbours and simultaneously read the data sent by their west neighbours
  - Steps 2 to 4: same as step 1 for west, south, and north (again, appropriate nodes are masked out)
  - Step 5: all PE's compute the new value using equation above

Multimedia SIMD Extensions

- Key ideas:
  - No network of processing elements, but an array of ALU's
  - No memories associated with ALU's, but a pool of relatively wide (64 to 128 bits) registers that store several narrower operands
  - No direct communication between ALU's, but via registers and with special shuffling/permutation instructions
  - Not co-processors or supercomputers, but tightly integrated into CPU pipeline
  - Still lockstep operation of ALU's

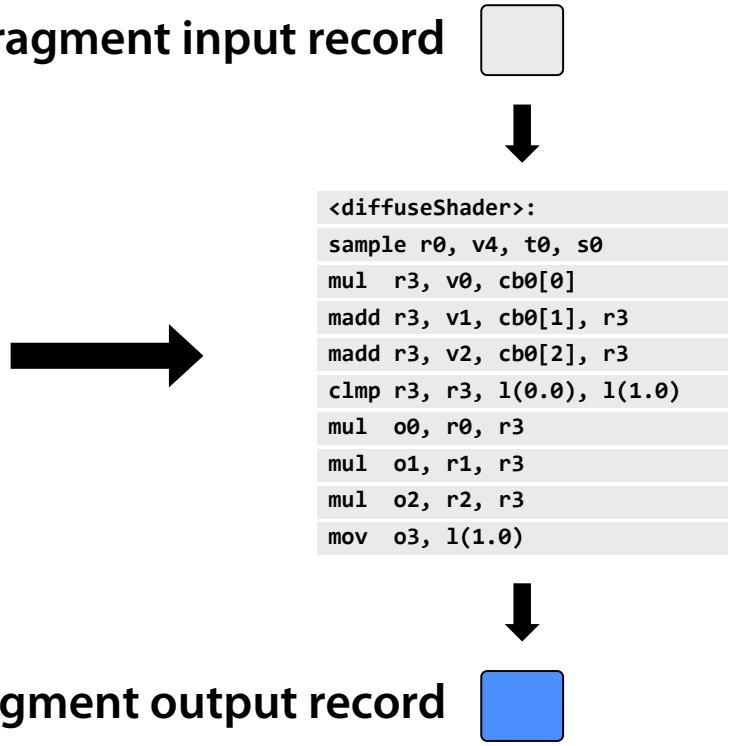
Graphics Processing Unit (GPU)

- Graphics apps have lot of parallelism
- Take advantage of hardware invested to do graphics well
- GPU is now ubiquitous!
- Several supercomputers in top500 use GPUs
  - Titan uses Nvidia tesla

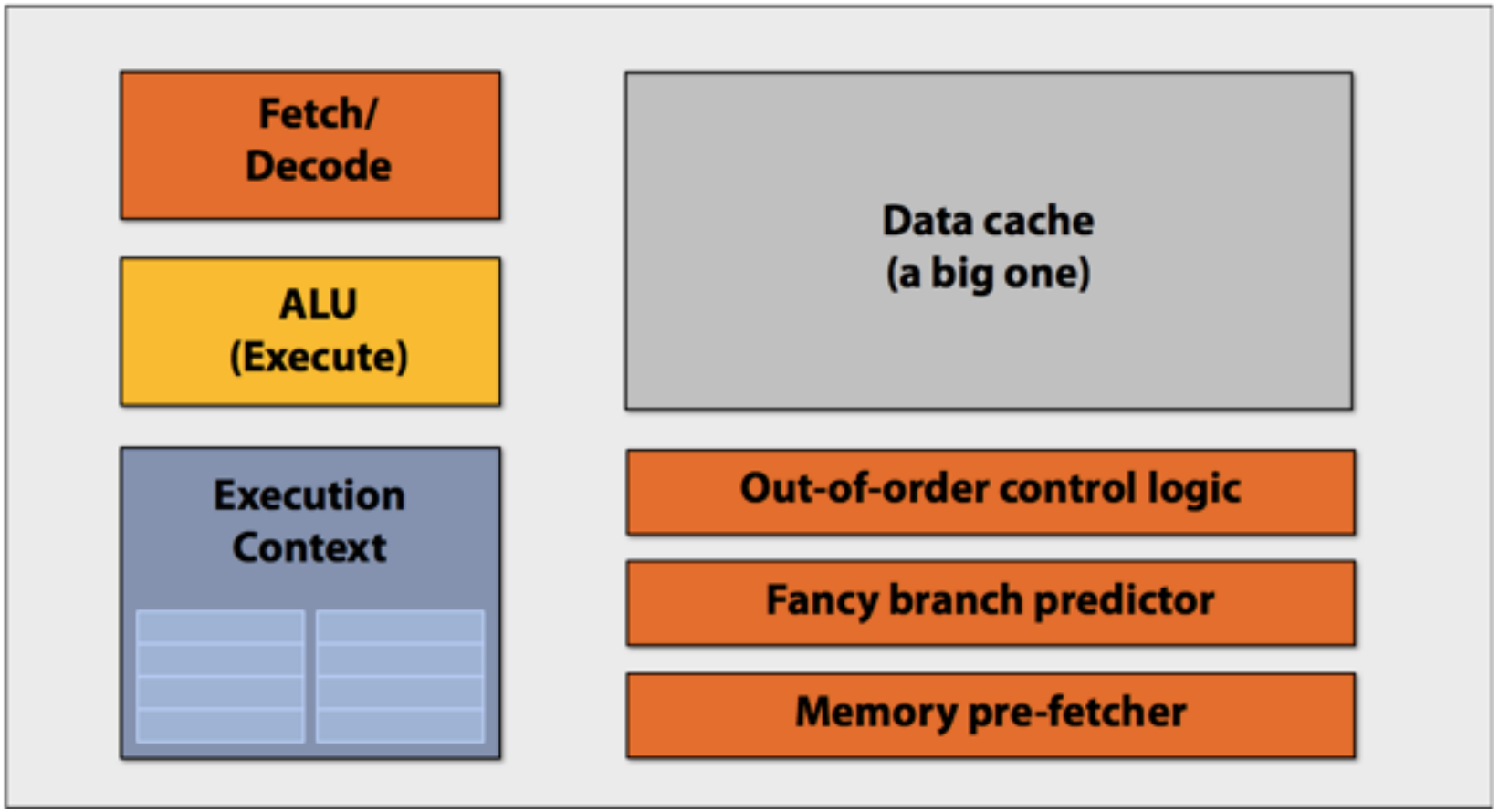
Thanks: Slides from Beyond programmable shading course ACM Siggraph '10 by Fatahalian

Shading a fragment

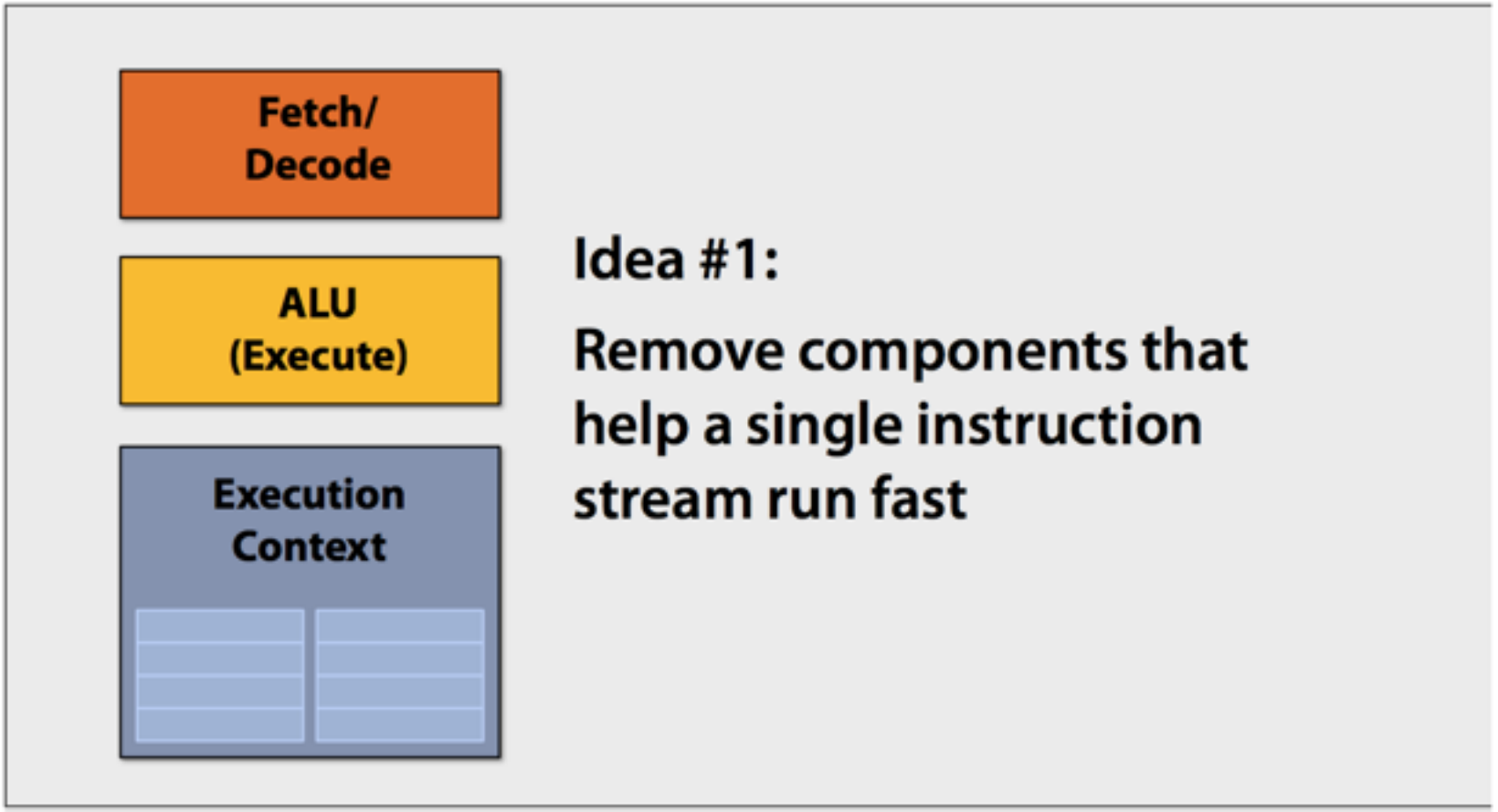
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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);
}
```



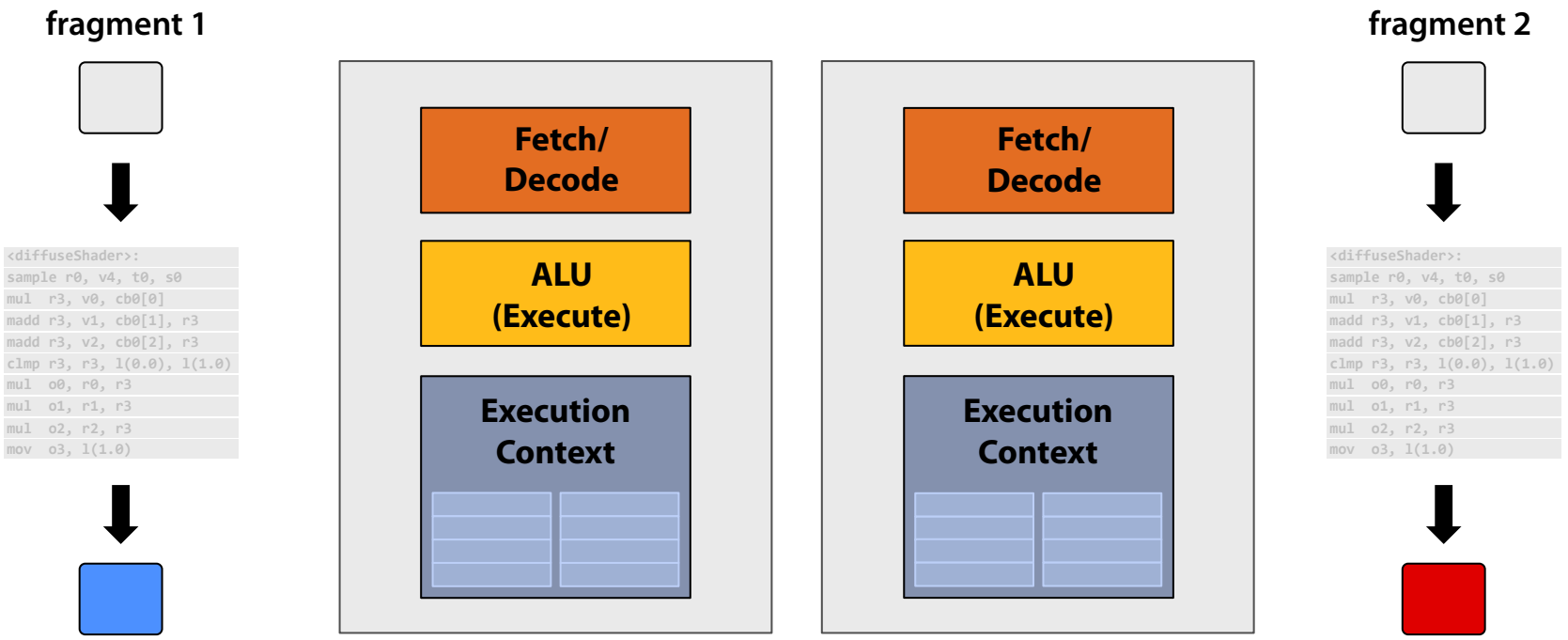
CPU-style Core



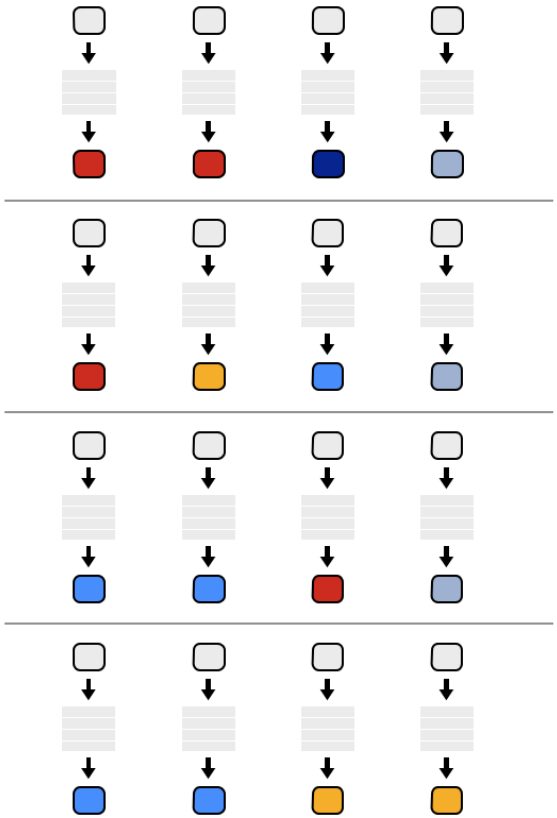
Slimming down…

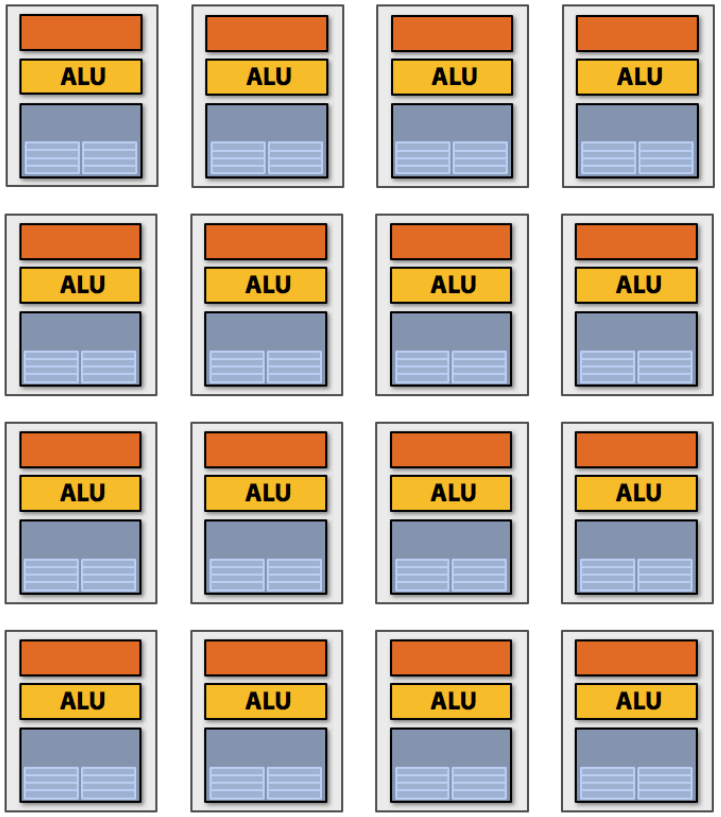


Add cores!


fragment 1
<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)
Fetch/ Decode
ALU (Execute)
Execution Context
Fetch/ Decode
ALU (Execute)
Execution Context
fragment 2
<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)

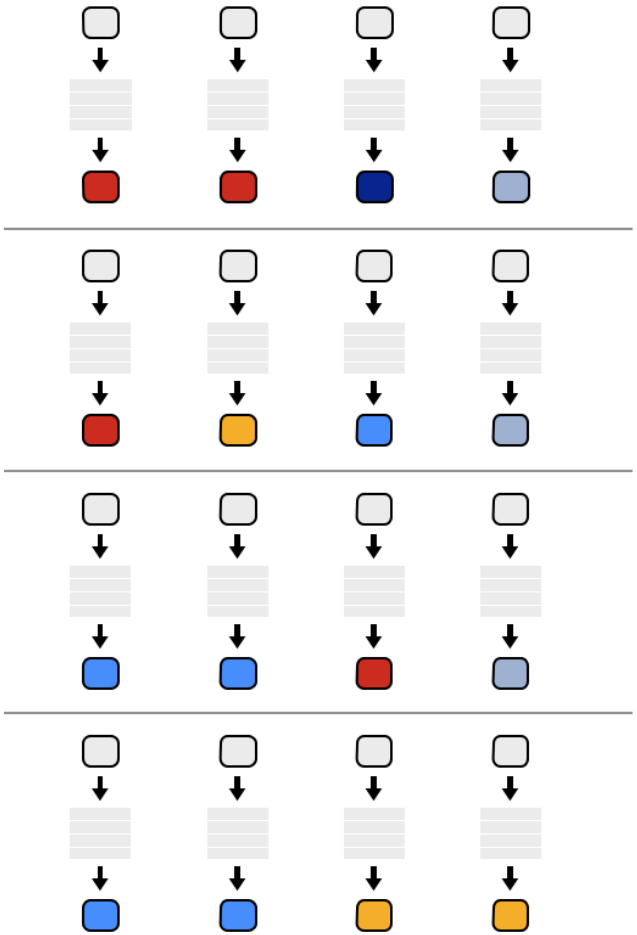
16 cores: 16 fragments in parallel





16 cores = 16 simultaneous instruction streams

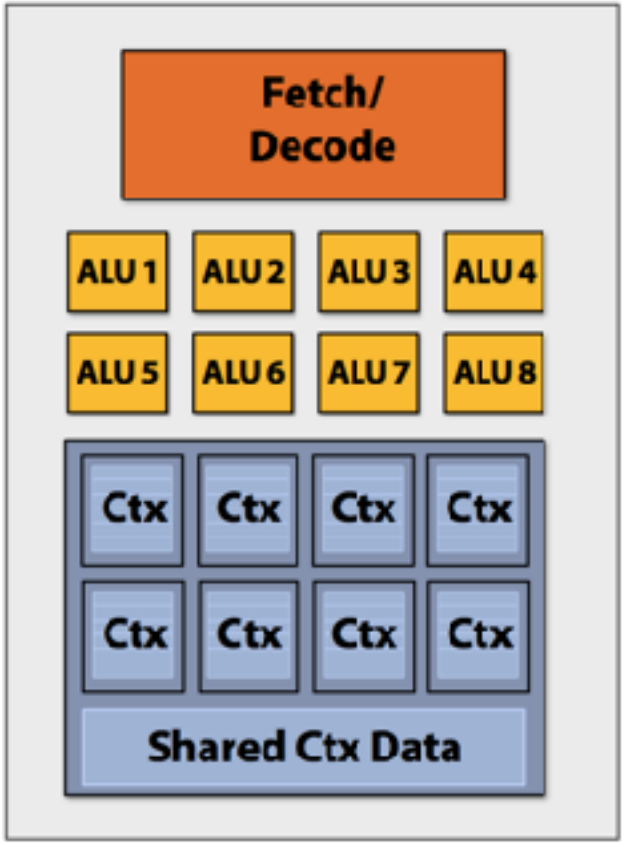
Instruction stream sharing



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)
```

Add ALUs (SIMD)…



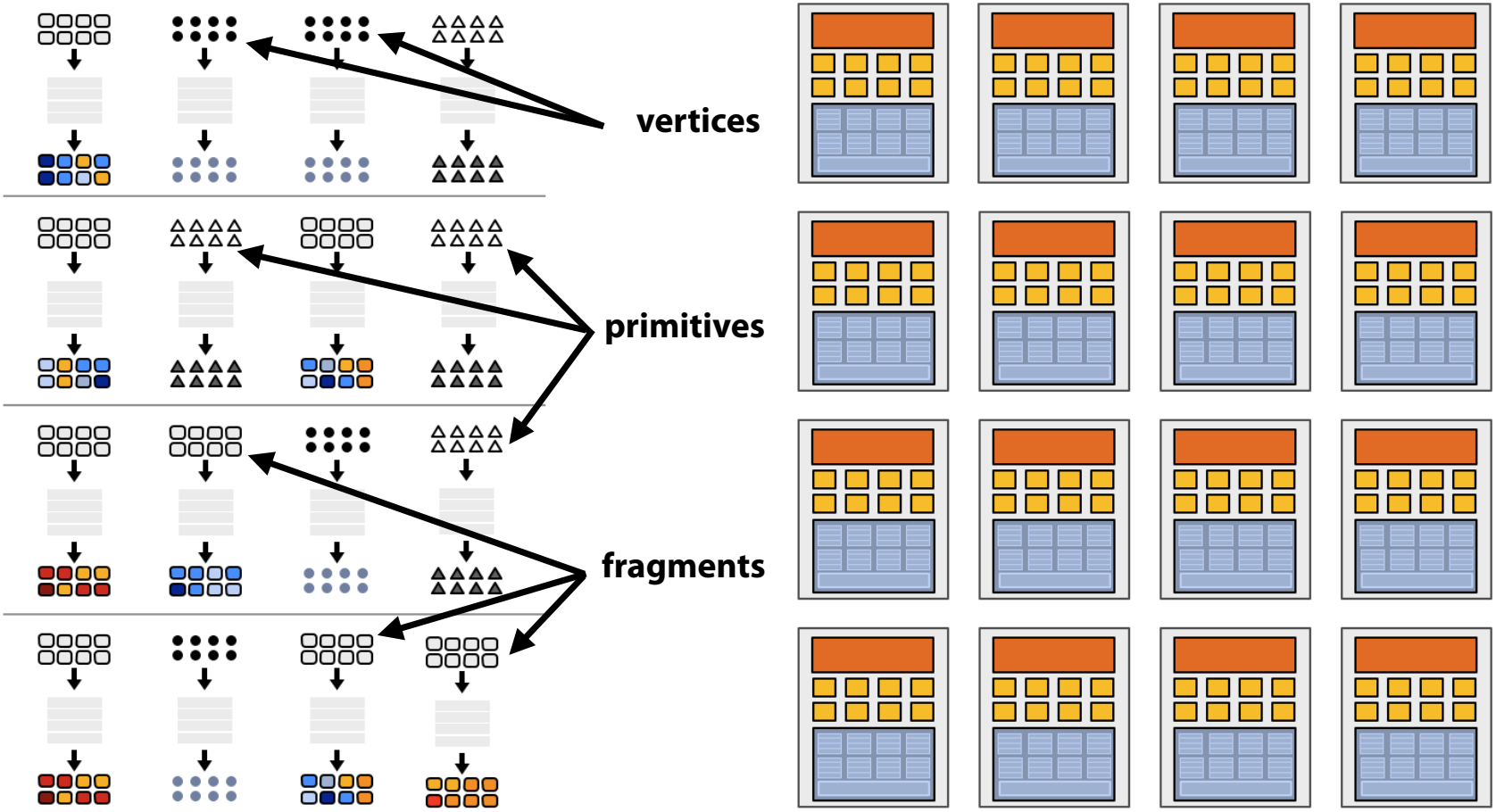
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

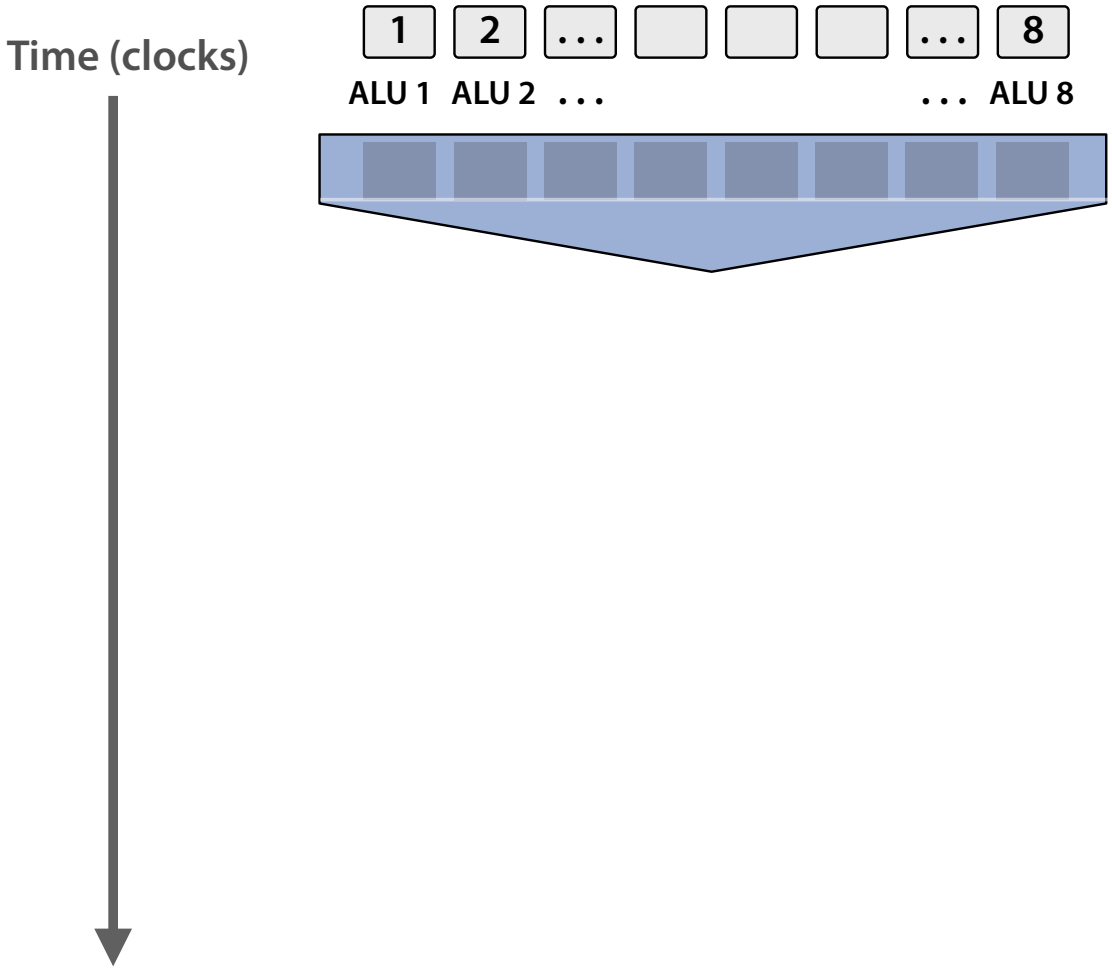
128 fragments in parallel!

16 cores = 128 ALUs, 16 simultaneous instruction streams

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

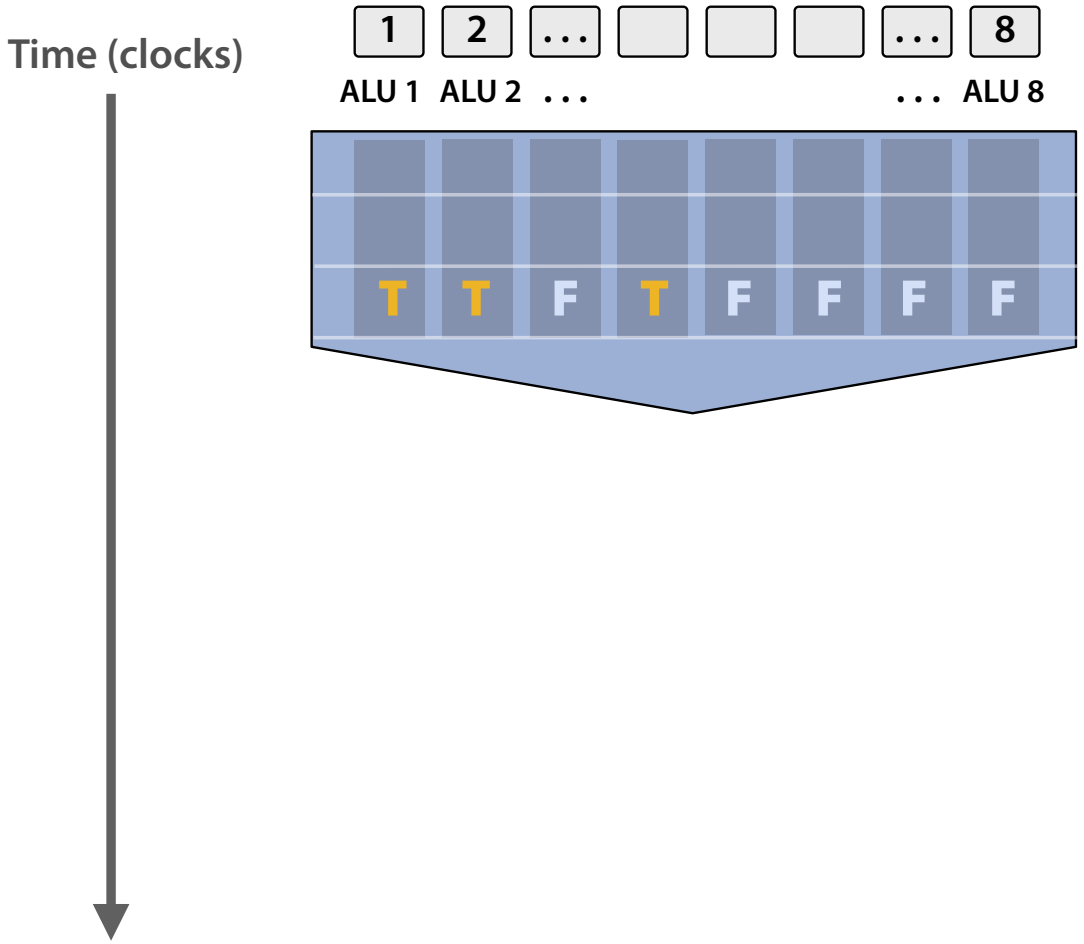


But what about branches?



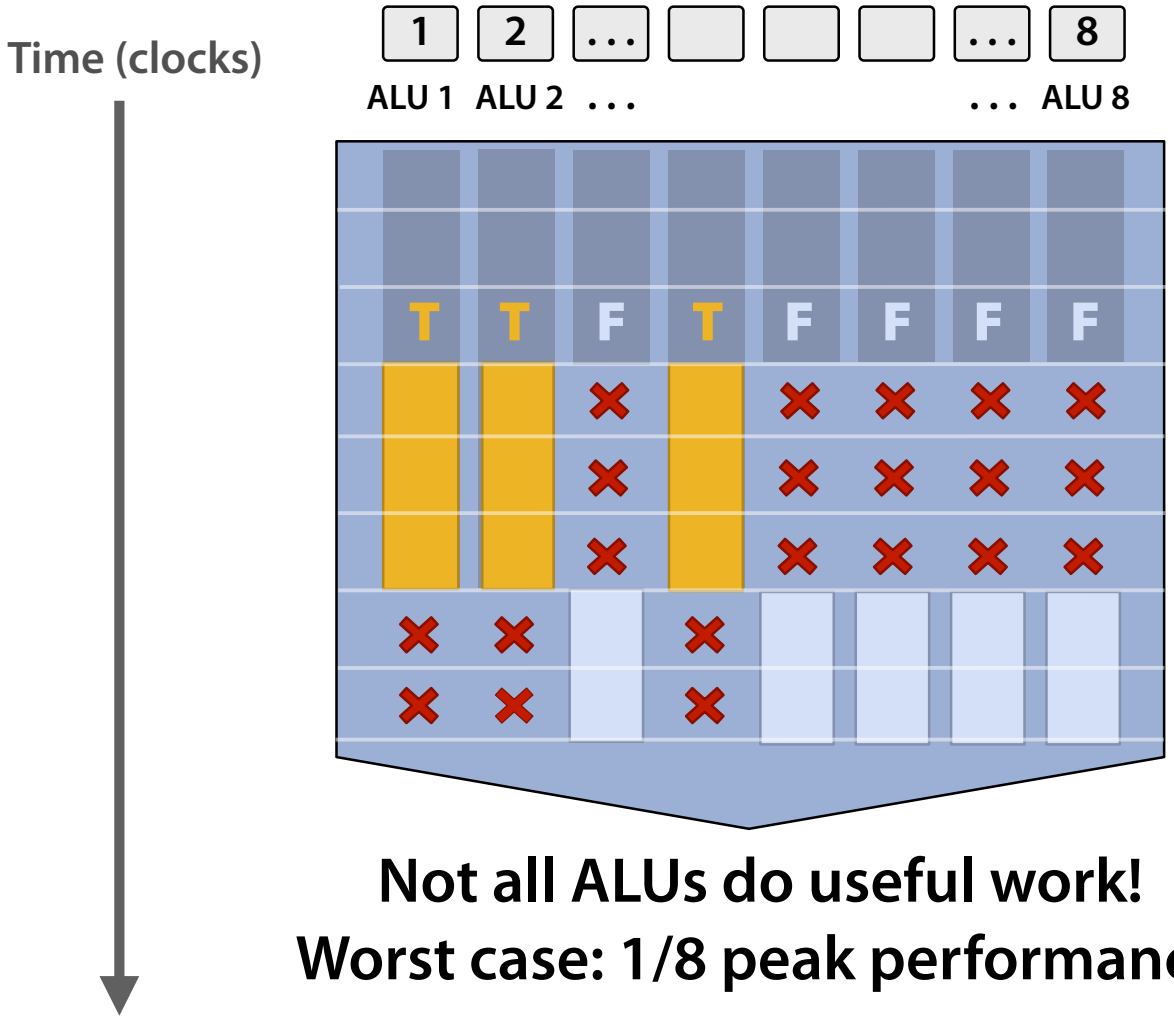

```
<unconditional
 shader code>

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>
```

Not all ALUs do useful work!

Worst case: 1/8 peak performance

Stalls!

- No caches
- No dynamic scheduling
- Dependencies and memory accesses can cause stalls!

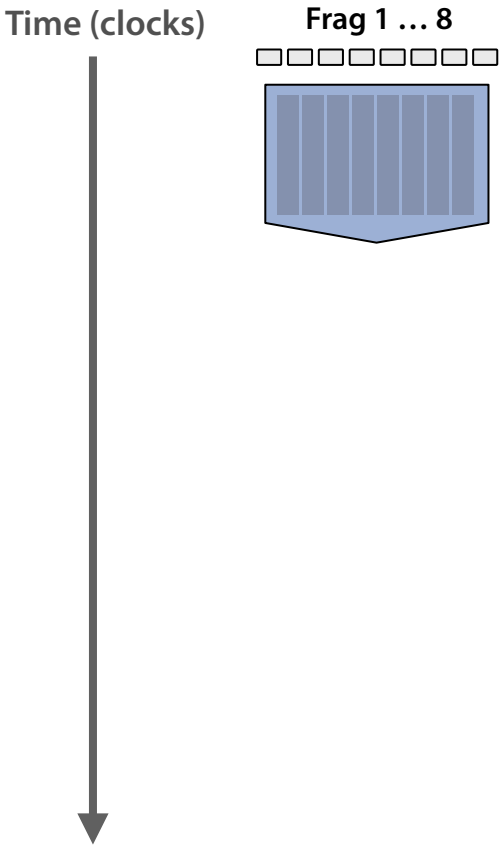
Solution: Multithreading

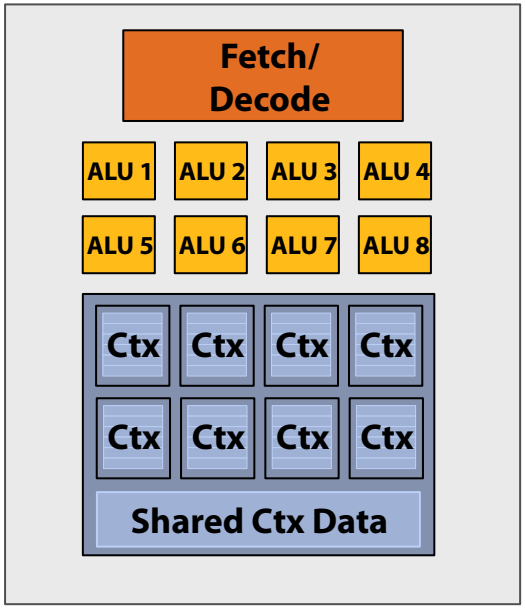
But we have LOTS of independent fragments.

Idea #3:

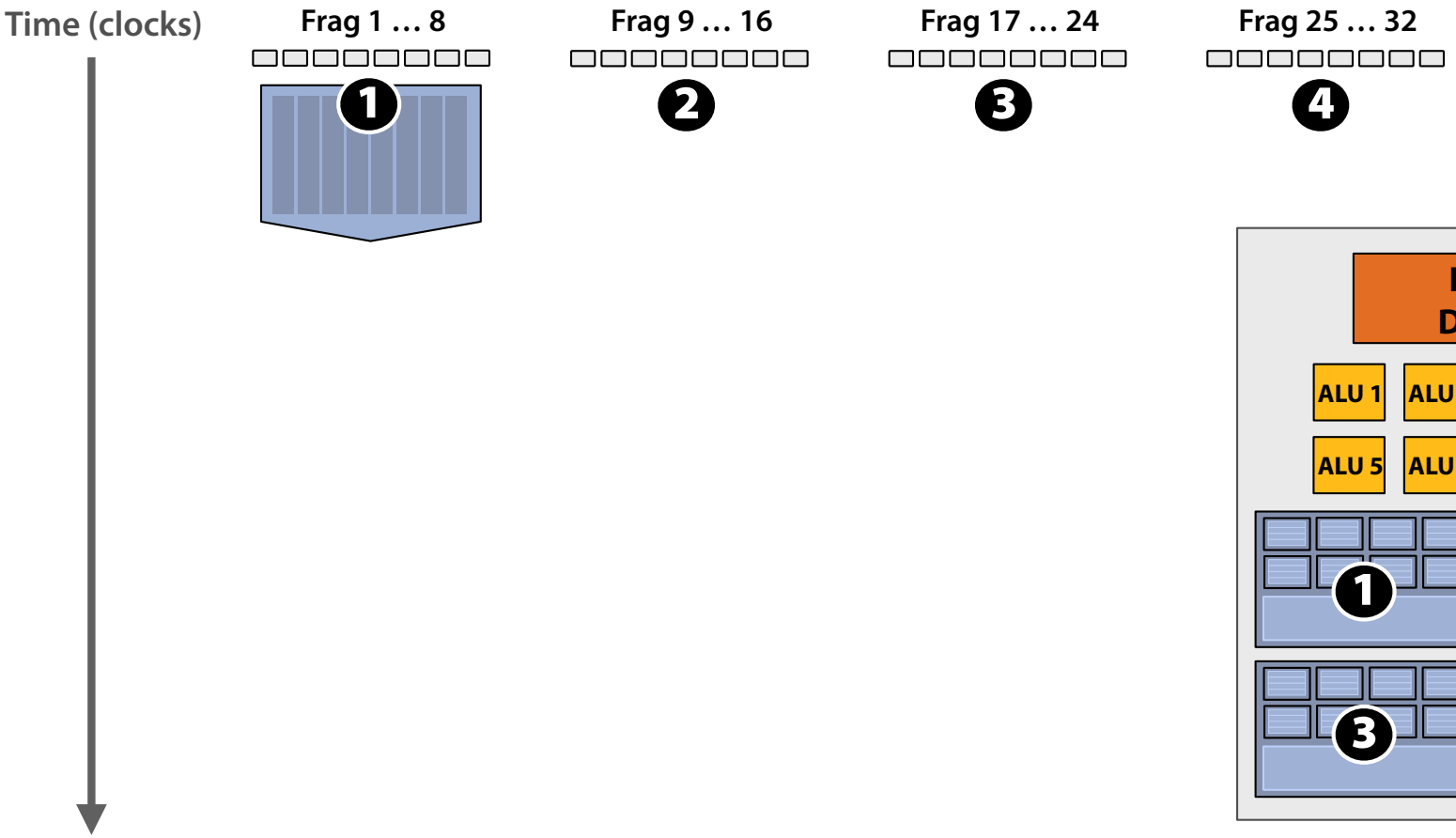
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

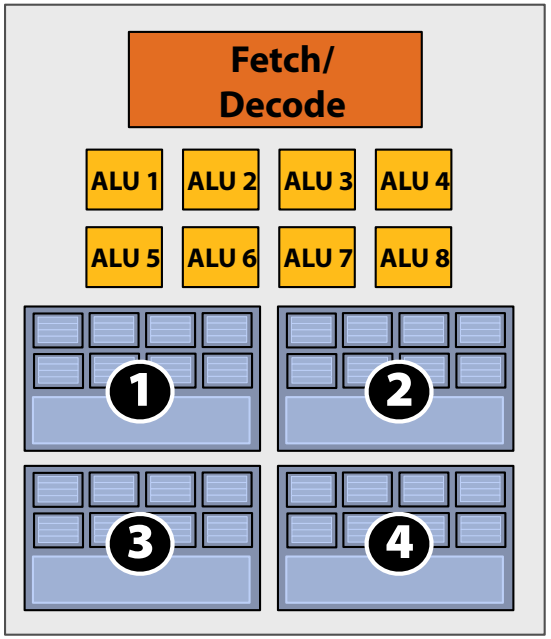
Hiding shader stalls



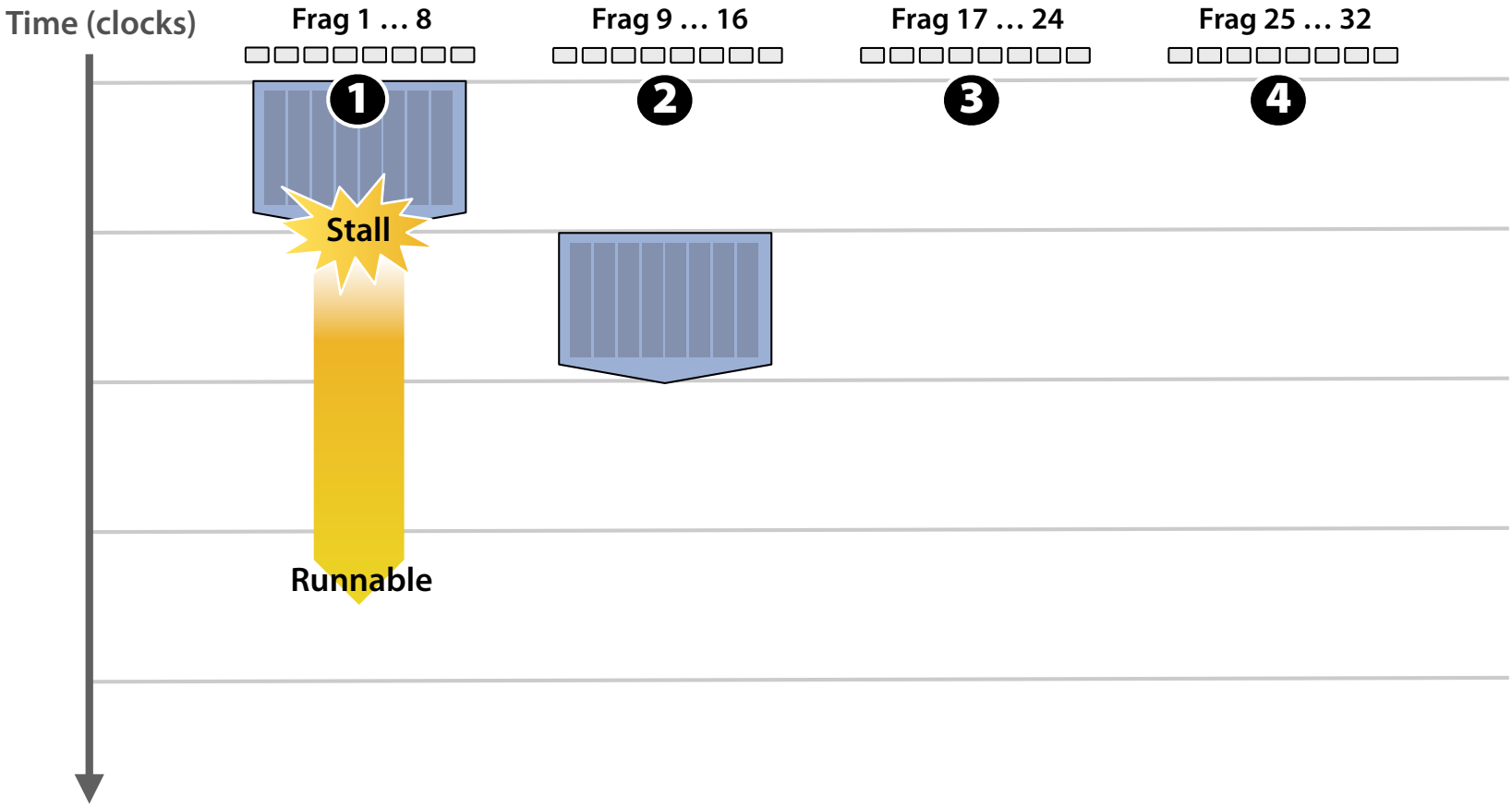


Hiding shader stalls

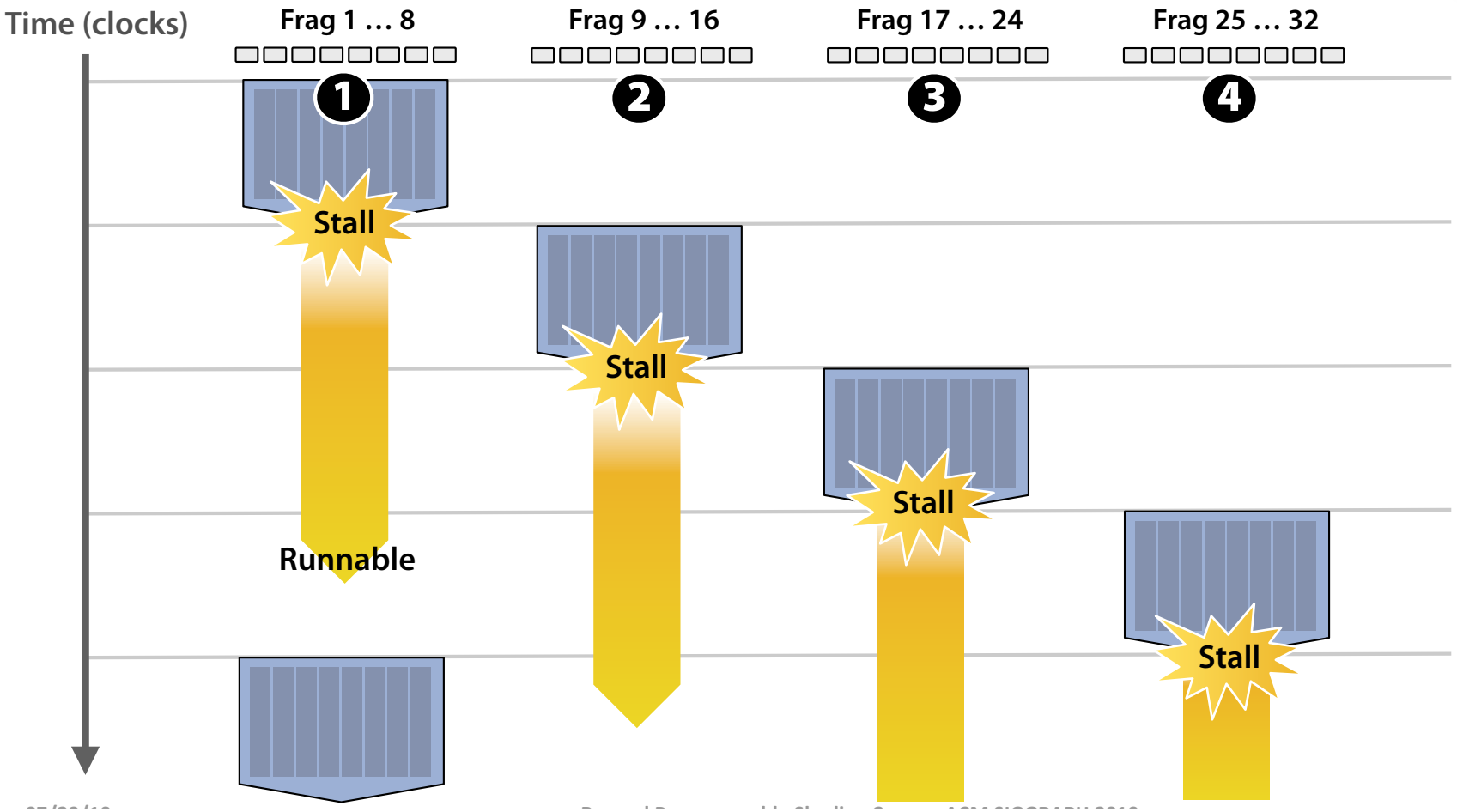




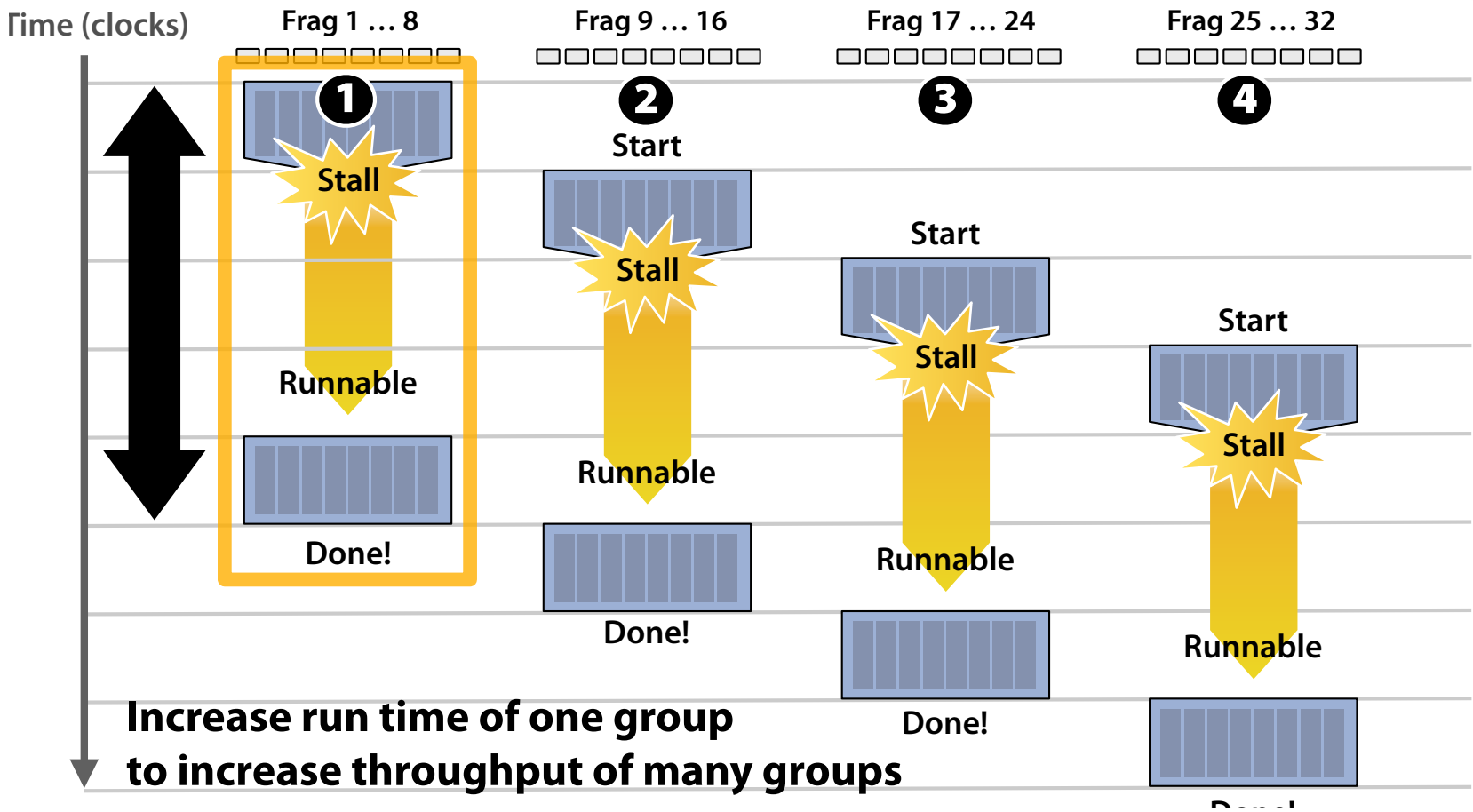
Hiding shader stalls



Hiding shader stalls



Throughput!


Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Stall
Start
Stall
Start
Stall
Runnable
Runnable
Runnable
Runnable
Done!
Done!
Done!
Increase run time of one group
to increase throughput of many groups

GPU…

16 cores

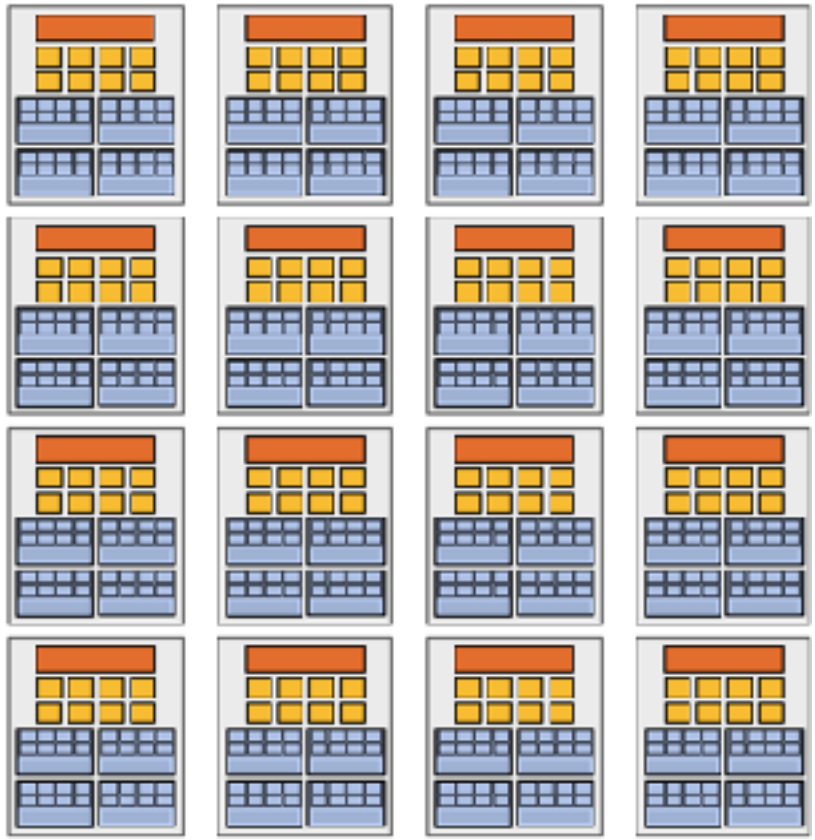
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



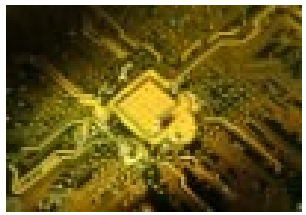
lfcs | Laboratory for Foundations of Computer Science

cisa | Centre for Intelligent Systems and their Applications

icsa | Institute for Computing Systems Architecture

```
lemma resPres:
  fixes P :: pi
  and   Q :: pi
  and   x :: name

  assumes PBiSimQ: "P ~ Q"

  shows "(νx)P ~ (νx)Q"
proof -
  let ?X = "{x. ∃P Q. P ~ Q ∧ (∃a. x = ((νa)P, (νa)Q)))"
  from PBiSimQ have "((νx)P, (νx)Q) ∈ ?X" by blast
  moreover have "⋀P Q a. P ↝[bisim] Q ⟹ (νa)P ↝[(?X ∪ bisim)] (νa)Q"
  proof -
    fix P Q a
    assume PSimQ: "P ↝[bisim] Q"
    moreover have "⋀P Q a. P ~ Q ⟹ ((νa)P, (νa)Q) ∈ ?X ∪ bisim" by blast
    moreover have "bisim ⊆ ?X ∪ bisim" by blast
    moreover have "eqvt bisim" by(rule eqvt)
    moreover have "eqvt (?X ∪ bisim)"
      by(auto simp add: eqvt_def dest: eqvtI)+
    ultimately show "(νa)P ↝[(?X ∪ bisim)] (νa)Q"
      by(rule strongLateSimPres.resPres)
  qed

  ultimately show ?thesis using PBiSimQ
    by(coinduct, blast dest: unfoldE)
qed
```

epcc



Program

Specify

Design

Verify

PhD in Pervasive Parallelism

Optimize

Apply

http://pervasiveparallelism.inf.ed.ac.uk

intel

Agilent Technologies

Microsoft

ALTRAN

criticalblue
Accelerating Embedded Software

QUALCOMM

IBM

codeplay

ACE

SAMSUNG

ARM

freescale™

ORACLE

wolfson microelectronics

SYNOPSYS

amazon

The End!

Further Reading

- The first truly successful vector supercomputer:

  "The CRAY-1 Computer System", R. M. Russel, Communications of the ACM, January 1978.

- Vector processor on a chip:

  "Vector vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks", C. Kozyrakis and D. Patterson, Intl. Symp. on Microarchitecture, December 2002.

- Integrating a vector unit with a state-of-the-art superscalar:

  "Tarantula: A Vector Extension to the Alpha Architecture", R. Espasa, F. Ardanaz, J. Elmer, S. Felix, J. Galo, R. Gramunt, I. Hernandez, T. Ruan, G. Lowney, M. Mattina, and A. Seznec, Intl. Symp. on Computer Architecture, June 2002.

Further Reading

- Seminal SIMD work:
  - "A Model of SIMD Machines and a Comparison of Various Interconnection Networks", H. Siegel, IEEE Trans. on Computers, December 1979.
  - "The Connection Machine", D. Hillis, Ph.D. dissertation, MIT, 1985.
- Two commercial SIMD supercomputers:
  - "The CM-2 Technical Summary", Thinking Machines Corporation, 1990.
  - "The MasPar MP-1 Architecture", T. Blank, Compcon, 1990.
- SIMD co-processor:
  - "CSX Processor Architecture", ClearSpeed, Whitepaper, 2006.