King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 7: GPU Architecture, Pt. 4

Markus Hadwiger, KAUST

Reading Assignment #4 (until Sep 25)

Read (required):

- Read:
  https://en.wikipedia.org/wiki/Instruction_pipelining
  https://en.wikipedia.org/wiki/Classic_RISC_pipeline
- Get an overview of NVIDIA Ampere (GA102) white paper:

https://www.nvidia.com/content/PDF/
nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

- Get an overview of NVIDIA Ampere (A100) Tensor Core GPU white paper:

https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/
nvidia-ampere-architecture-whitepaper.pdf

- Get an overview of NVIDIA Hopper (H100) Tensor Core GPU white paper:

https://resources.nvidia.com/en-us-tensor-core

NVIDIA GTC

GTC (GPU Technology Conference)

- Sep 19 – 22, 2022
- Keynote: Sep 20 (18:00 – 19:30)
  - Announcement of Ada/Lovelace architecture
- Future of AI chat with Turing Award winners: Sep 20 (20:00 – 20:50)

https://www.nvidia.com/gtc/

Next Lectures

no lecture on Sep 21 !

Lecture 8: Sunday, Sep 25

Quiz #1: Sep 28

Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assignments
- Programming assignments (algorithms, methods)
- Solve short practical examples

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?
- 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)
- To exploit ILP: Execute independent instructions in parallel (e.g., 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)
- To exploit TLP: Execute operations/instructions from multiple threads in parallel
- **On GPUs: main type of parallelism**

(more types:

- Bit-level parallelism (processor word size: 64 bits instead of 32, etc.)
- Data parallelism (SIMD, but also SIMT), task parallelism, …)

Concepts: Latency Hiding

It's 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

Where this is going...

Summary: three key ideas for high-throughput execution

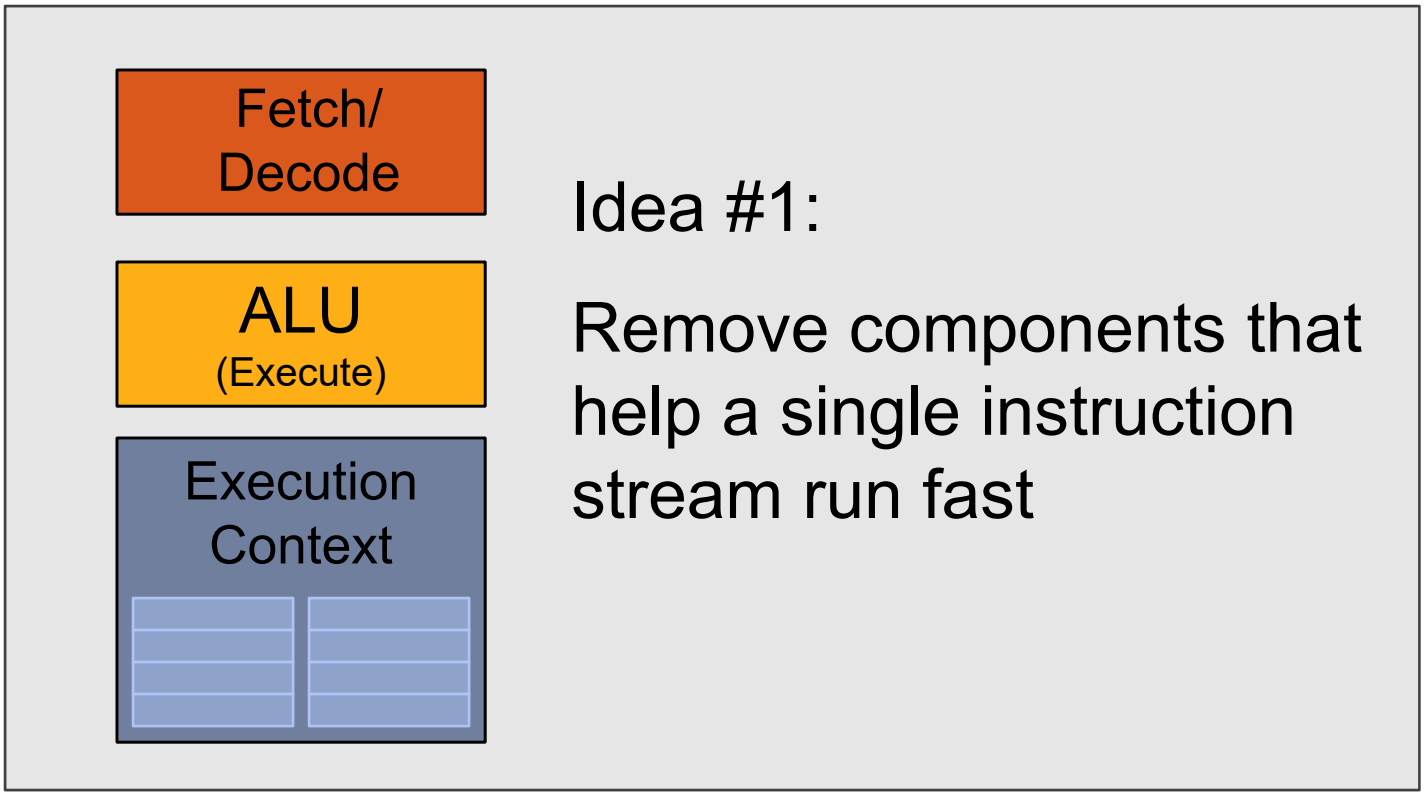
1. Use many "slimmed down cores," run them in parallel

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

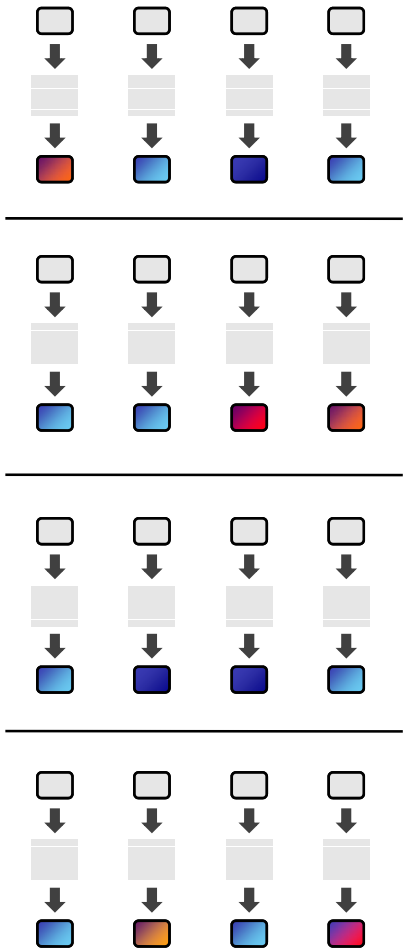
Idea #1: Slim down



Sixteen cores (sixteen fragments in parallel)

16 cores = 16 simultaneous instruction streams

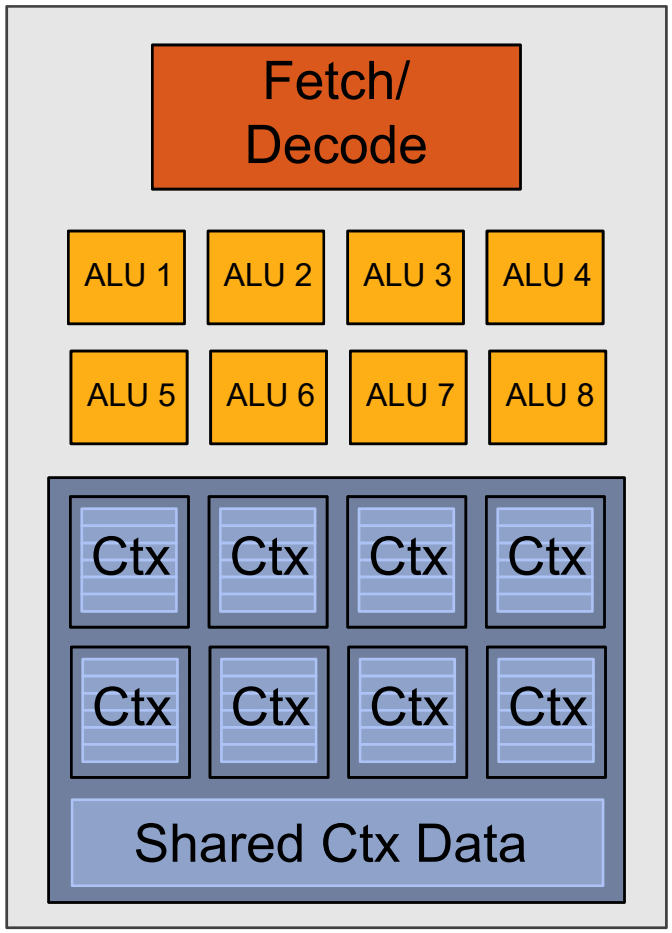
Instruction stream sharing



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)
```

Idea #2: Add ALUs



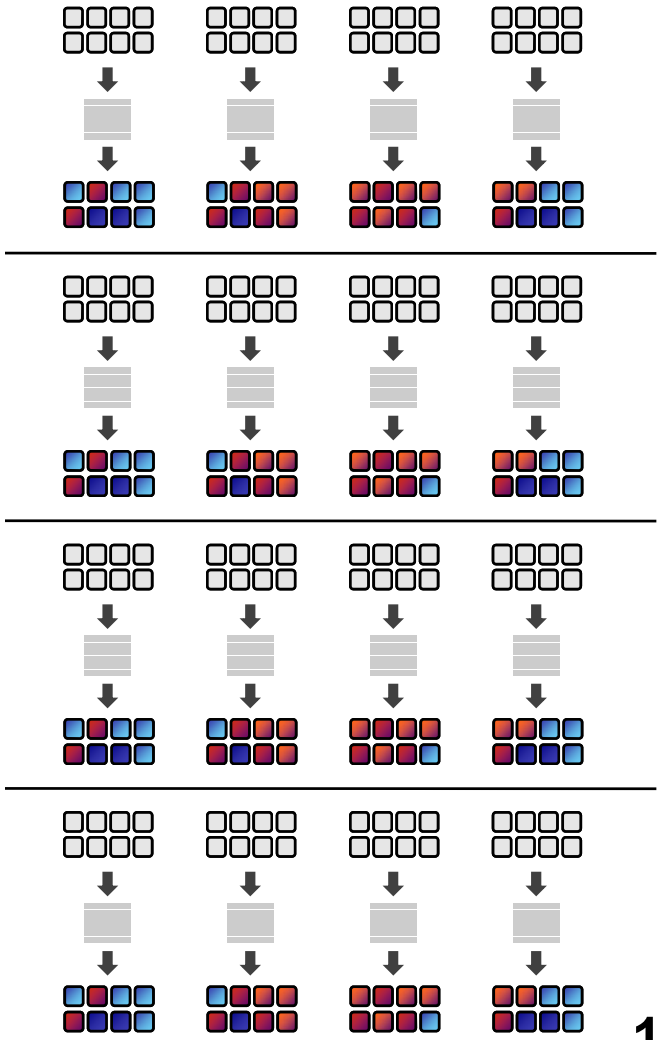
Idea #2:

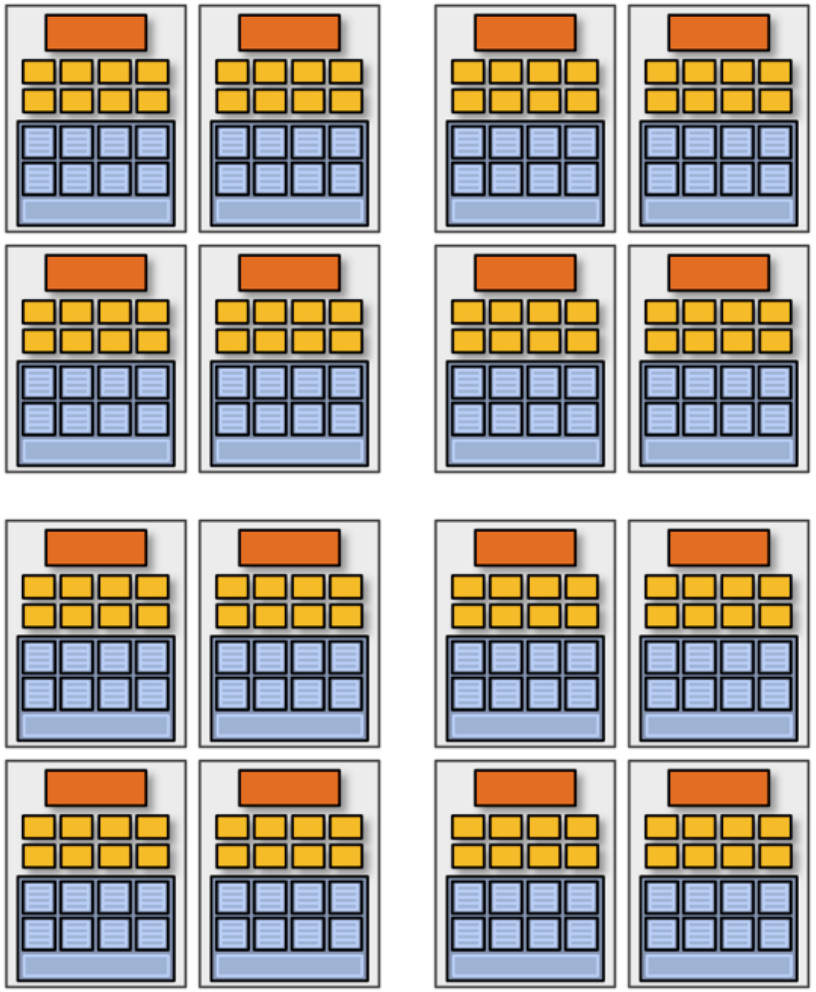
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

(or SIMT, SPMD)

128 fragments in parallel





16 cores = **128** ALUs

= **16** simultaneous instruction streams

Next Problem: Stalls!

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

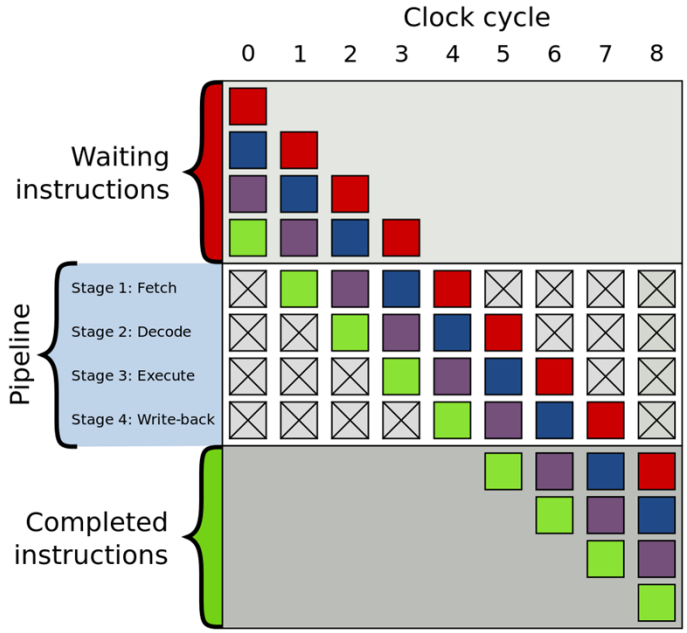
Texture access latency = 100's to 1000's of cycles
(also: instruction pipelining hazards, …)

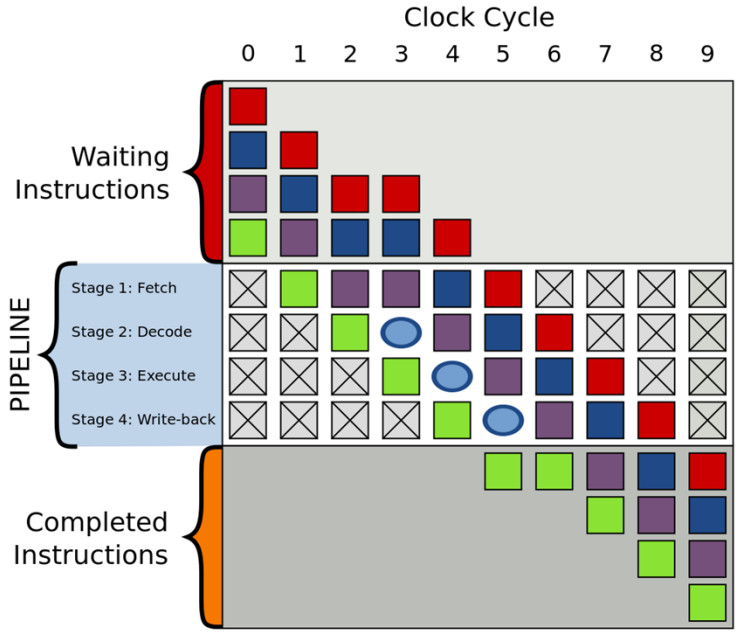
We've removed the fancy caches and logic that helps avoid stalls.

Interlude: Instruction Pipelining

Most common way to exploit *instruction-level parallelism* (ILP)

Problem: hazards (different solutions: bubbles, forwarding, …)





wikipedia

https://en.wikipedia.org/wiki/Instruction_pipelining
https://en.wikipedia.org/wiki/Classic_RISC_pipeline

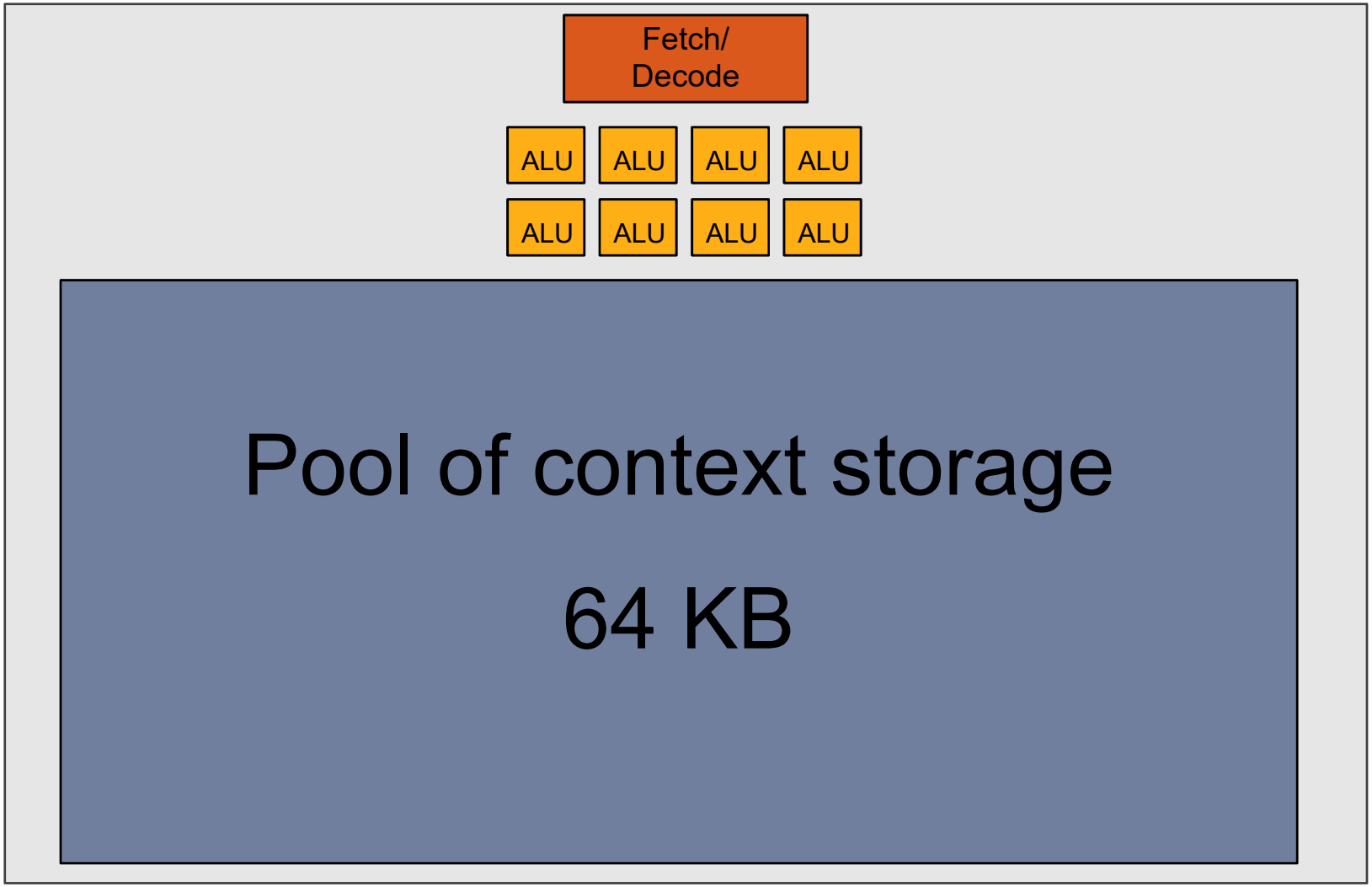
Idea #3: Interleave execution of groups

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.

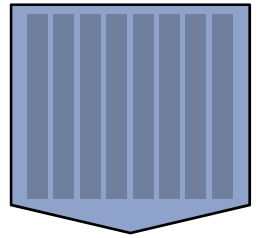
Idea #3: Store multiple group contexts

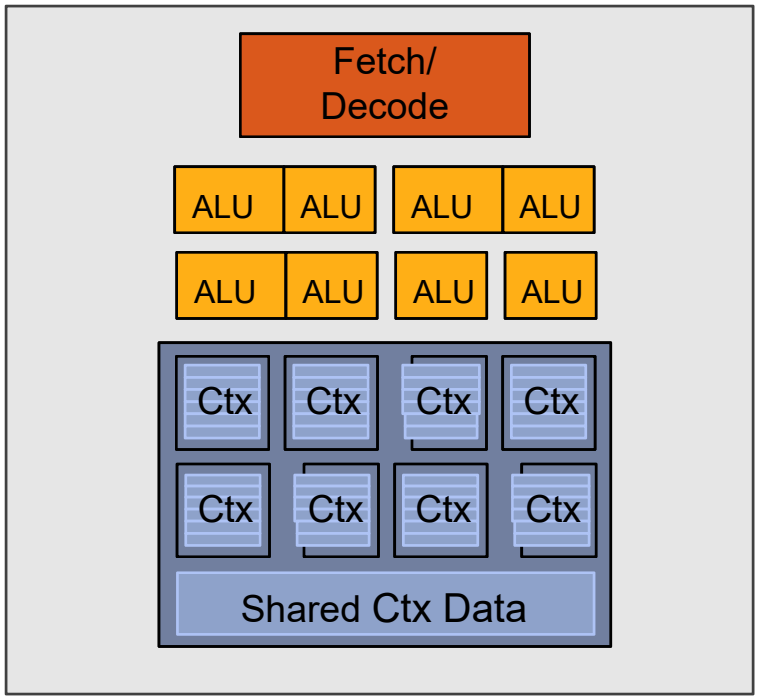


Hiding shader stalls

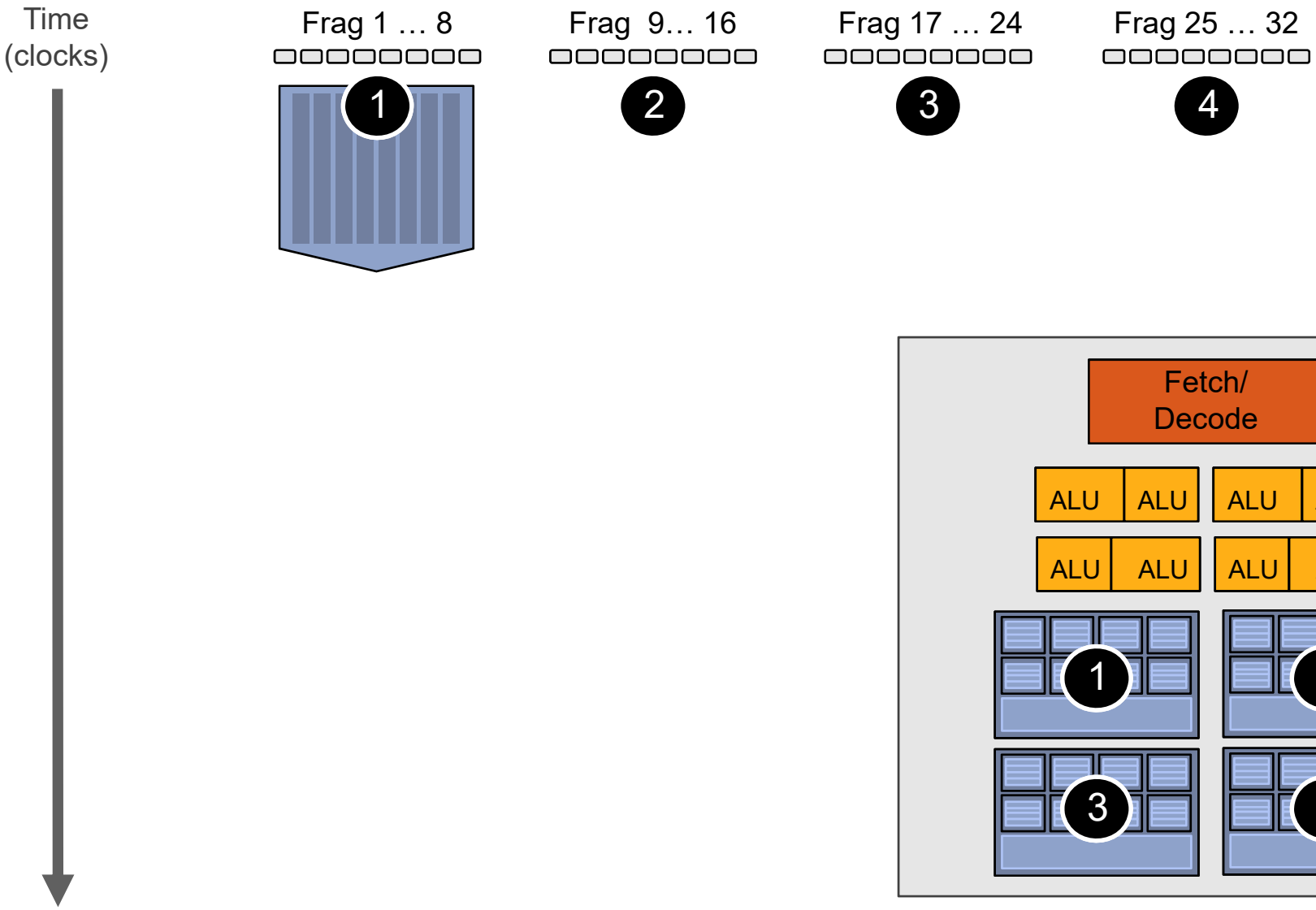
Time
(clocks)

Frag 1 … 8

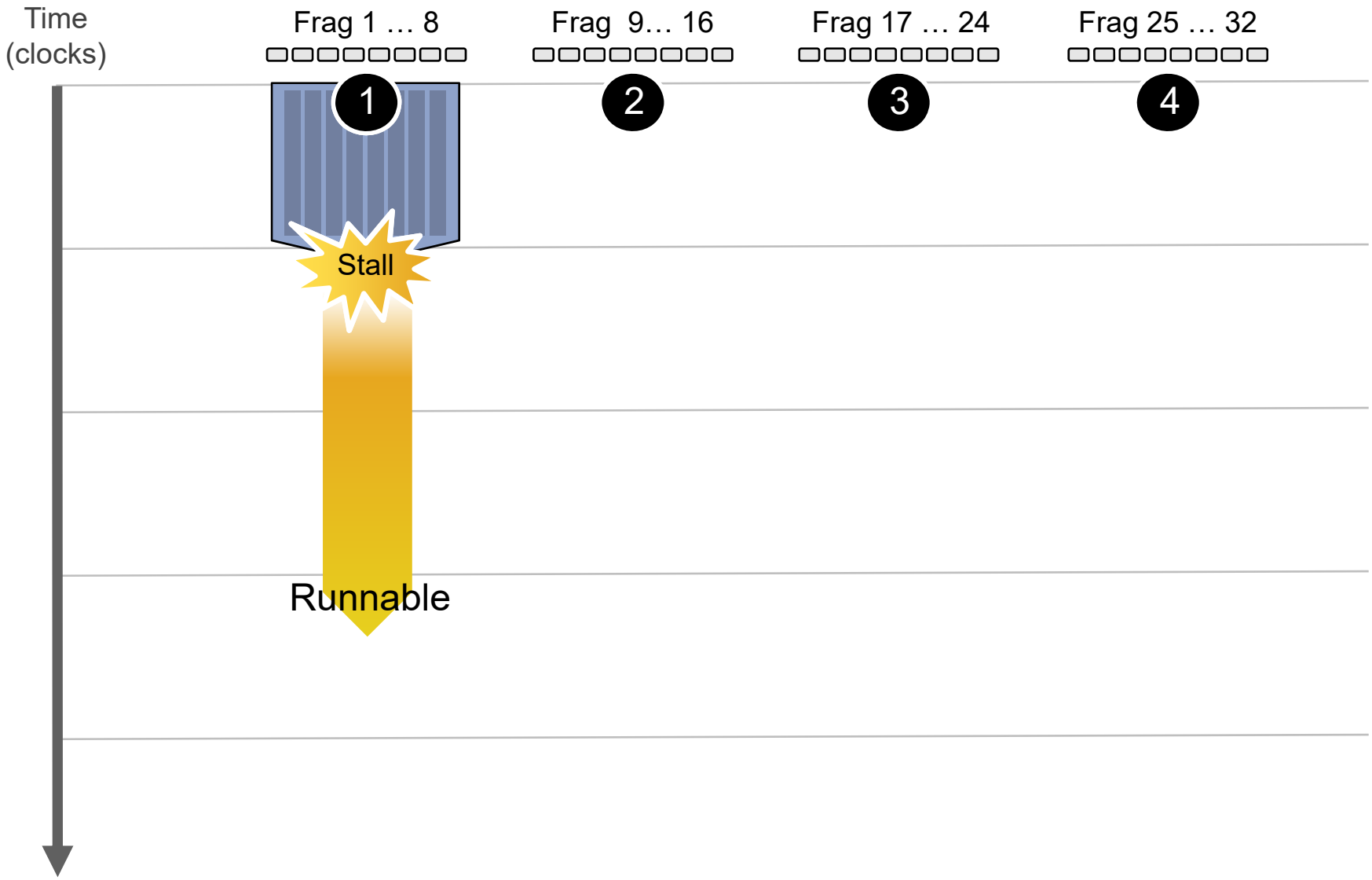




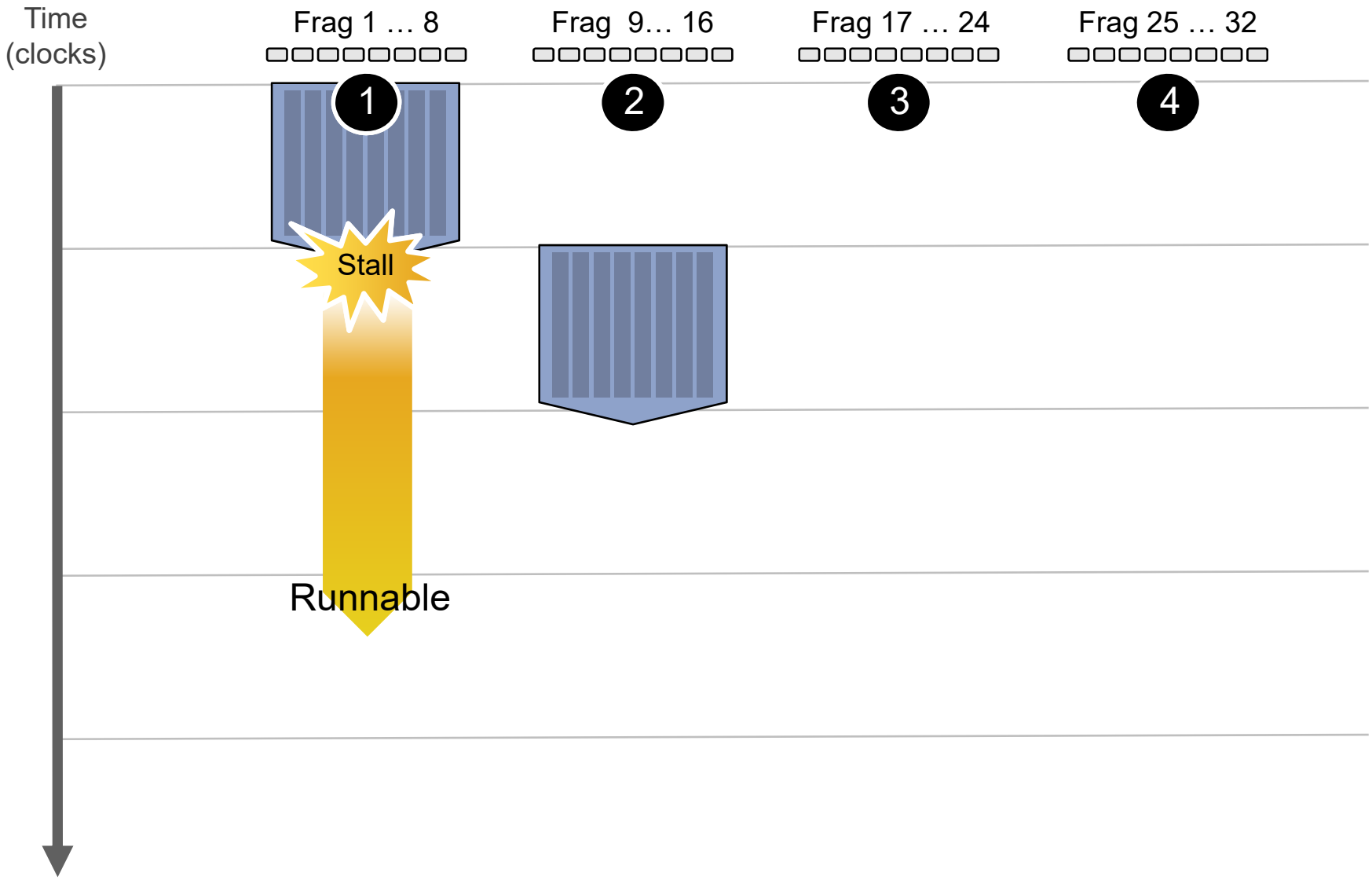
Hiding shader stalls



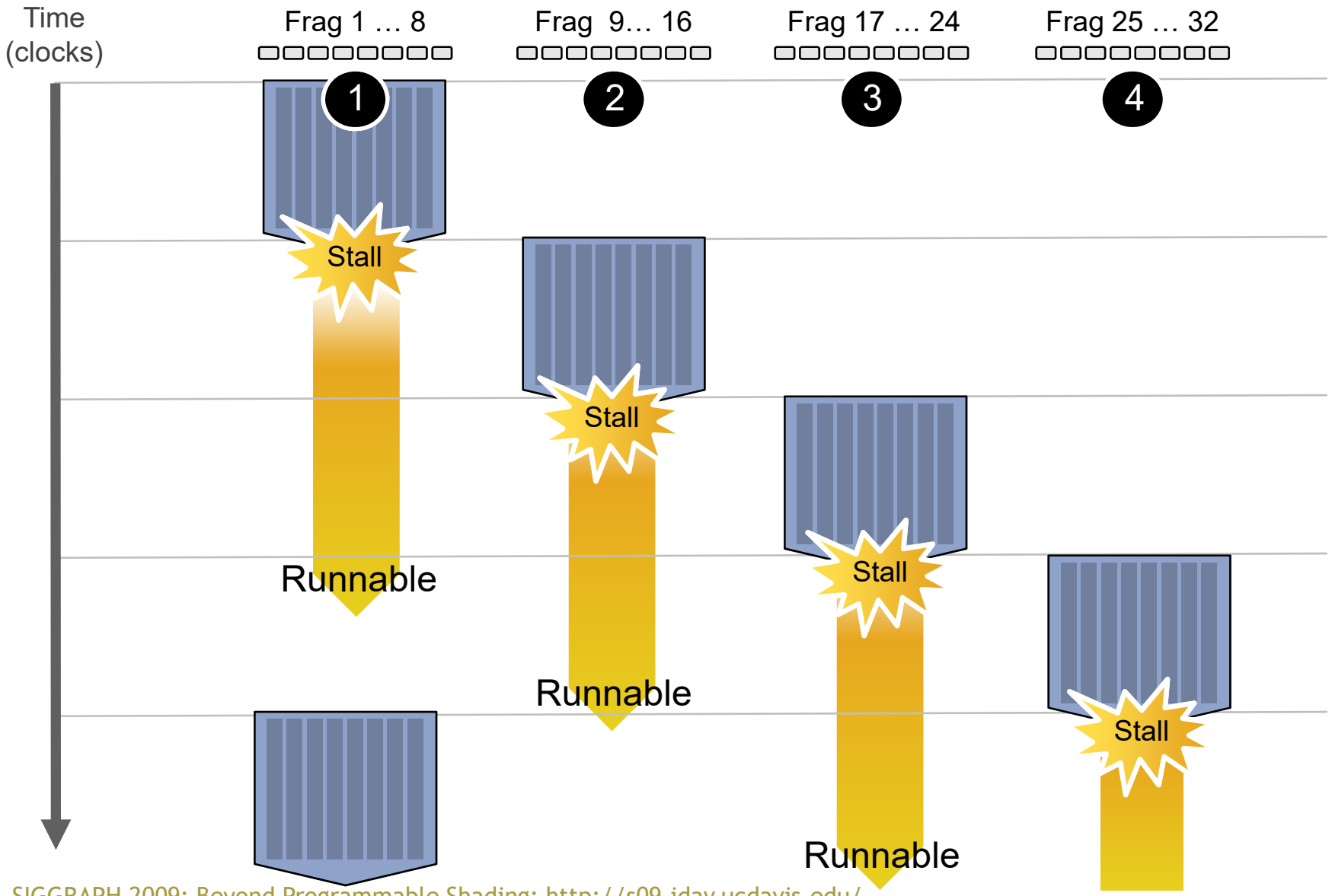
Hiding shader stalls



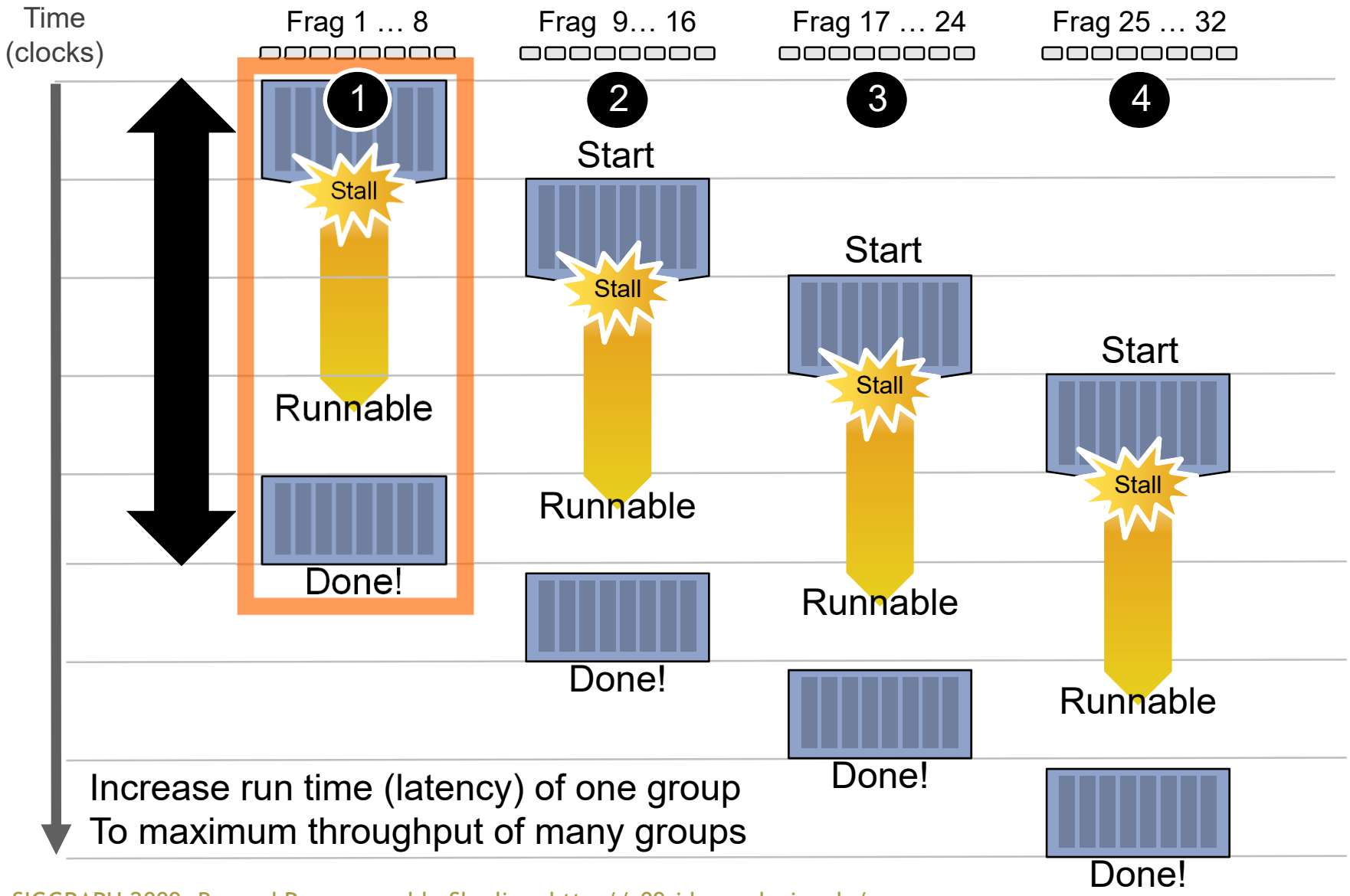
Hiding shader stalls



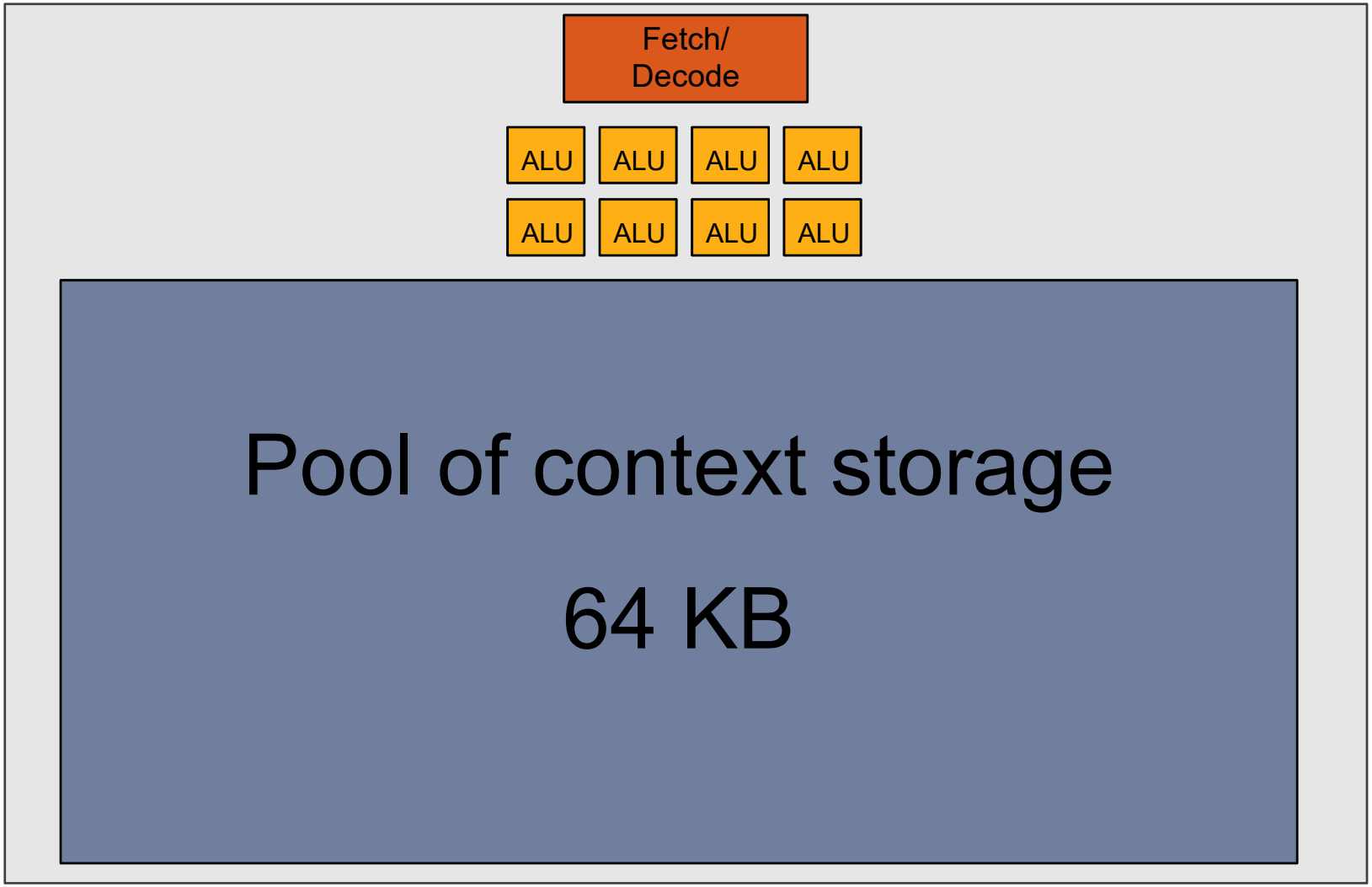
Hiding shader stalls



Throughput!

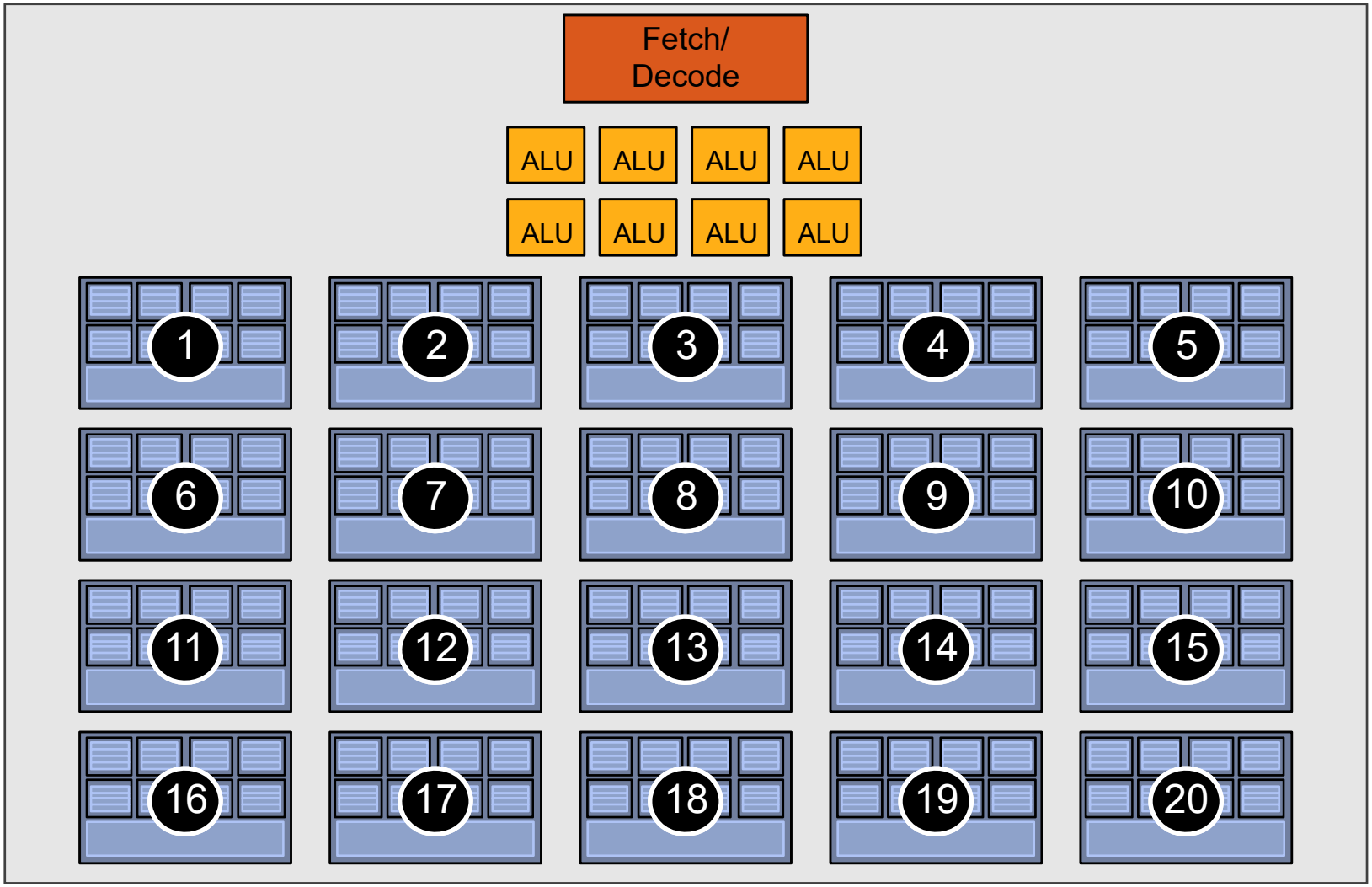

Time
(clocks)
Frag 1 … 8
Frag 9… 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Runnable
Done!
Increase run time (latency) of one group
To maximum throughput of many groups

Storing contexts

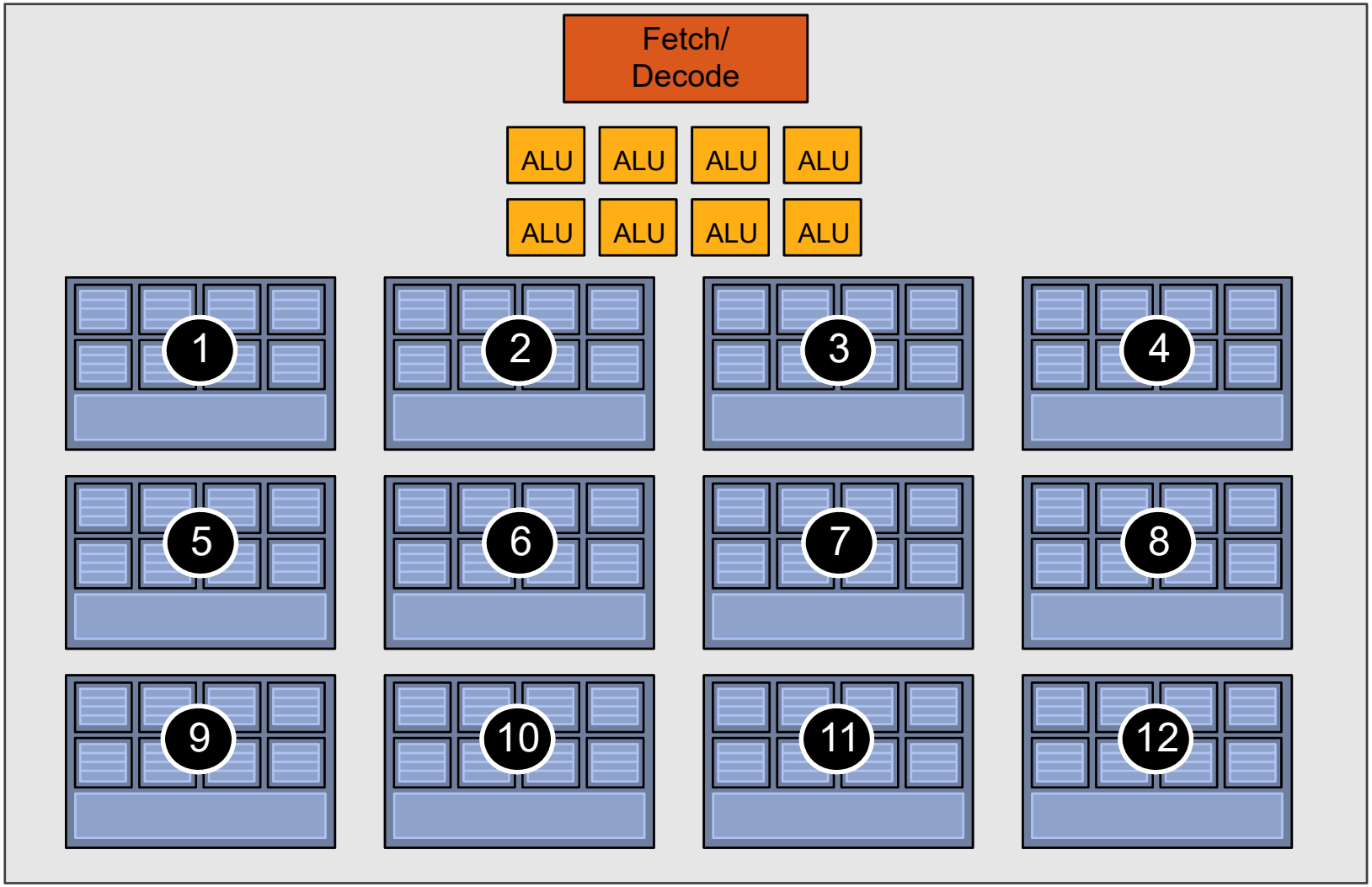


Twenty small contexts (few regs/thread)

(maximal latency hiding ability)

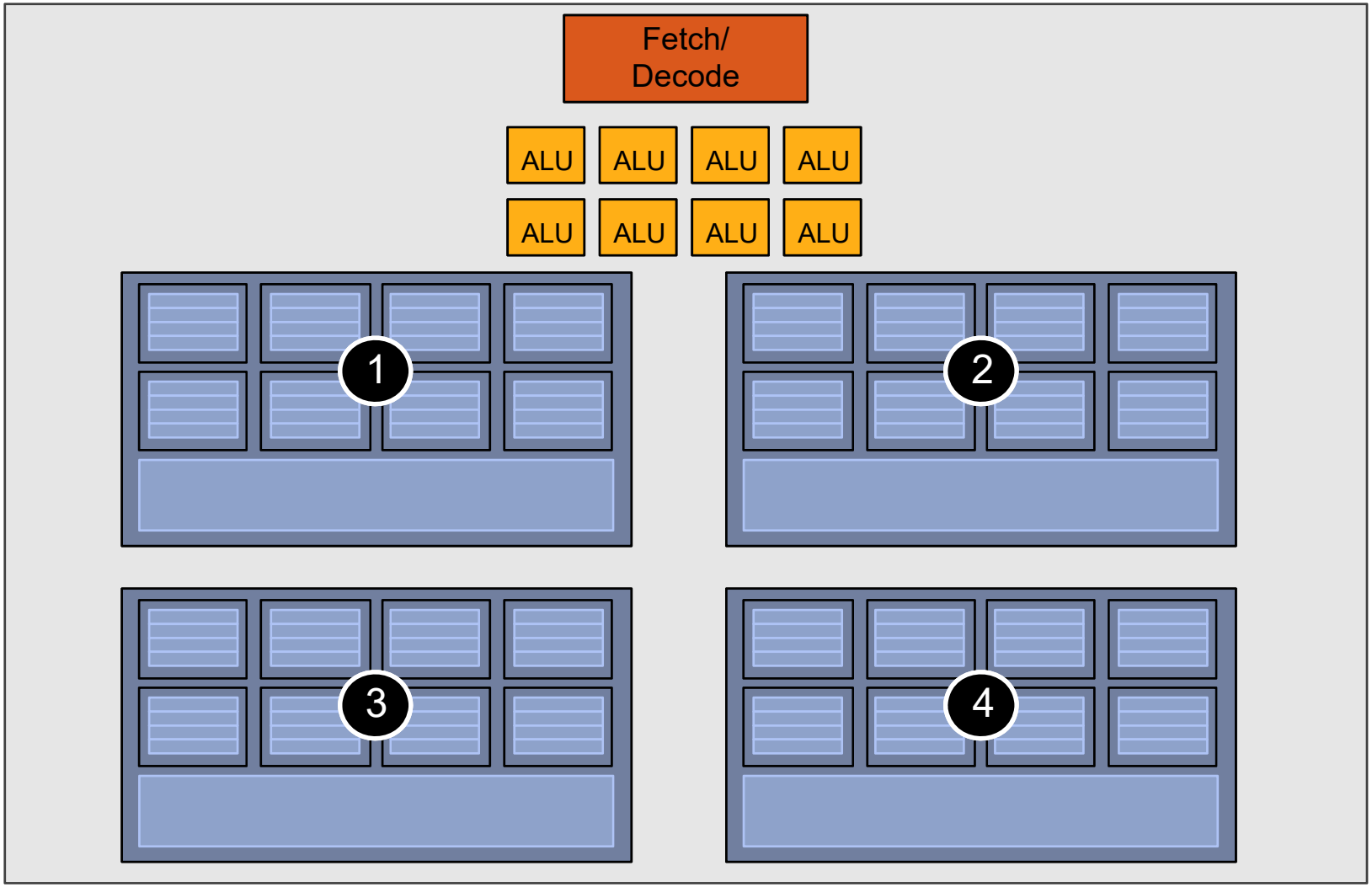


Twelve medium contexts (more regs/th.)



Four large contexts (many regs/thread)

(low latency hiding ability)



Complete GPU

16 cores

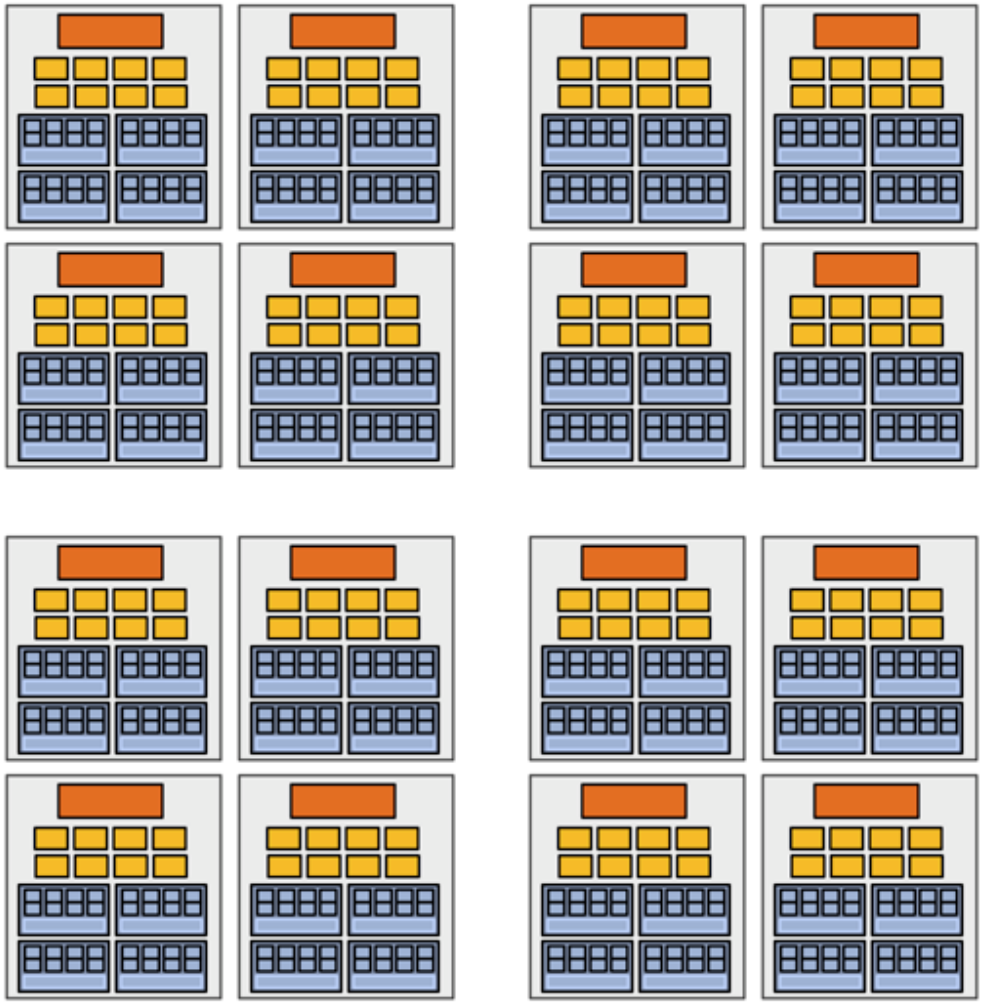
8 mul-add [mad] ALUs per core
(8*16 = **128** total)

16 simultaneous
instruction streams

64 (4*16) concurrent (but
interleaved) instruction streams

512 (8*4*16) concurrent
fragments (resident threads)

= 256 GFLOPs (@ 1GHz)
(**128** * 2 [mad] * 1G)



Complete GPU

16 cores

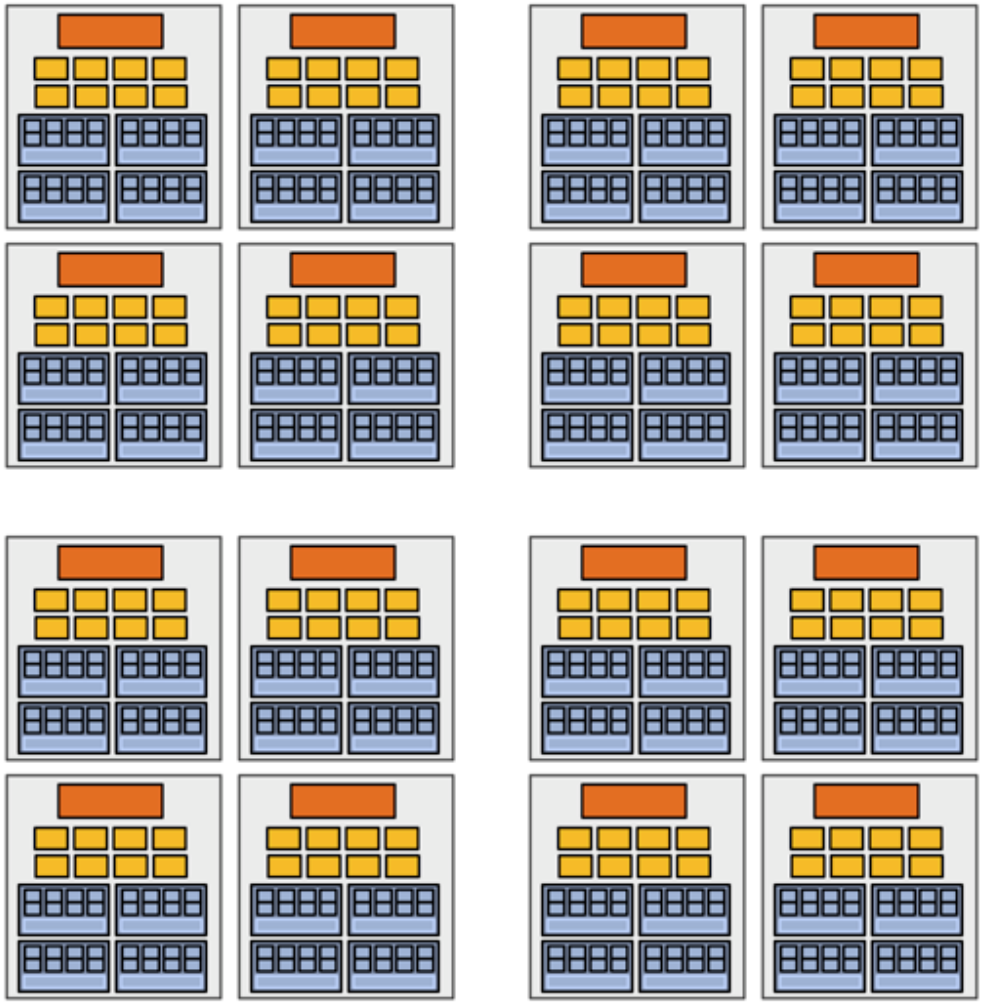
8 mul-add [mad] ALUs per core (8*16 = **128** total)

16 simultaneous instruction streams

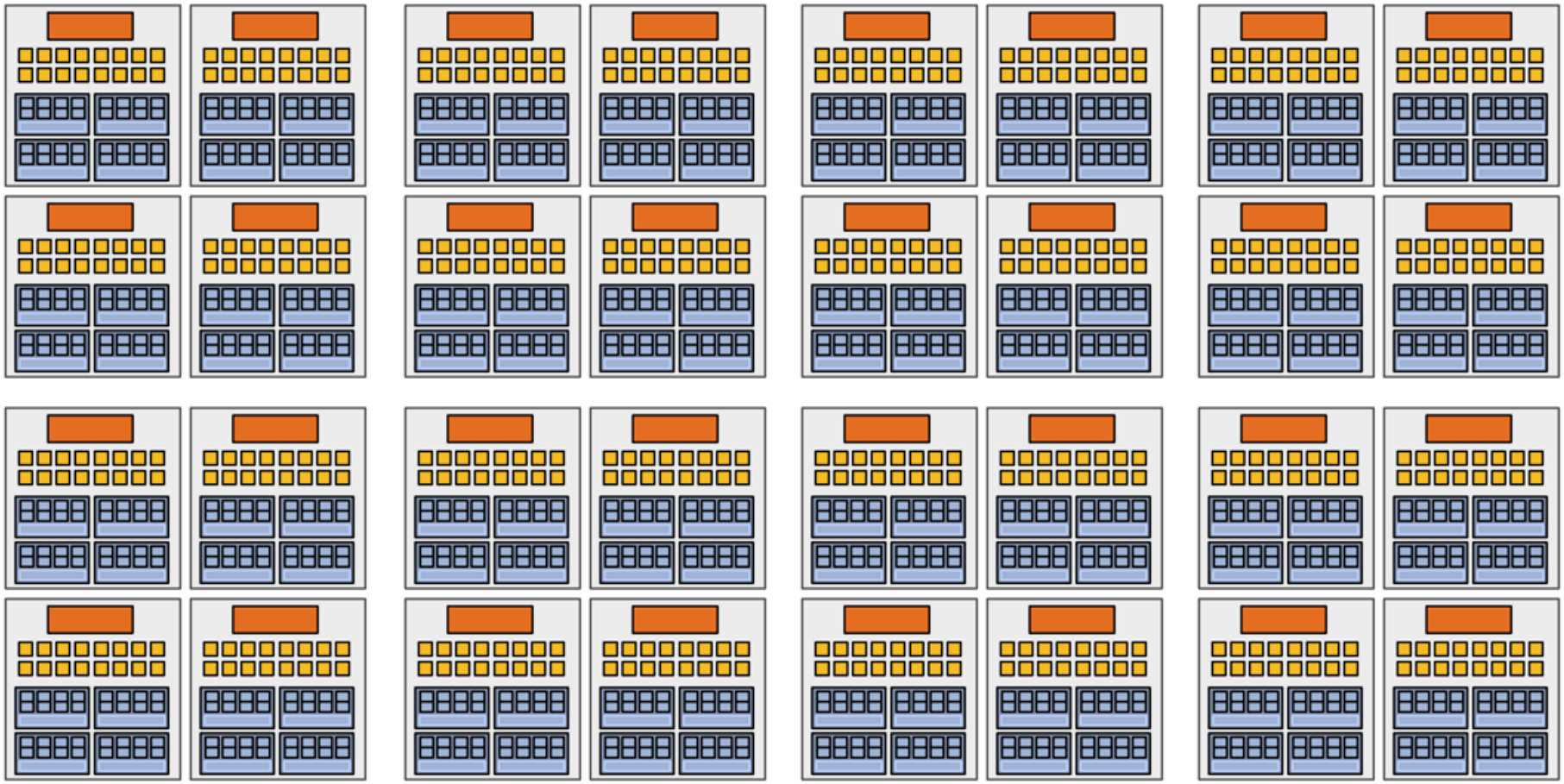
64 (4*16) concurrent (but interleaved) instruction streams

512 (8*4*16) concurrent fragments (resident threads)

= **256 GFLOPs** (@ 1GHz)
(**128** * 2 [mad] * 1G)



“Enthusiast” GPU (Some time ago :)



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

Where We've Arrived...

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

GPU Architecture: Real Architectures

NVIDIA Architectures (since first CUDA GPU)

Tesla: 2007-2009

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

Fermi: 2010

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

Kepler: 2012

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

Maxwell: 2015

- GM107, ... (GTX 750Ti, ...)
  GM204, ... (GTX 980, Titan X, ...)

Pascal: 2016

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

Volta: 2017/2018

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

Turing: 2018/2019

- TU102, TU104, TU106, TU116, ...
  (Titan RTX, RTX 2070, 2080, 2080Ti, ...)

Ampere: 2020

- GA100, GA102, GA104, ...
  (A100, RTX 3070, 3080, 3090, ...)

Hopper, Ada/Lovelace: 2022/23

- GH100,...
  (H100,...)

Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 5.4)

	Compute Capability										
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	**9.0**	**9.?**
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128	256[3]		256	?
32-bit floating-point add, multiply, multiply-add	192	128		64	128		64		128	128	?
64-bit floating-point add, multiply, multiply-add	64[4]	4		32	4		32[5]	32	2	64	?

3 128 for __nv_bfloat16
4 8 for GeForce GPUs, except for Titan GPUs
5 2 for compute capability 7.5 GPUs

Concepts: Latency Hiding

It's 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

Concepts: SM Occupancy in CUDA

We need to hide latencies from

- Instruction pipelining hazards
- Memory access latency

First type of latency: Definitely need to hide! (it is always there)

Second type of latency: only need to hide if it does occur (of course not unusual)

Occupancy: How close are we to *maximum latency hiding ability?*
(how many threads are resident vs. how many could be)

ALU Instruction Latencies and Instructs. / SM

CC	2.0 (Fermi)	2.1 (Fermi)	3.x (Kepler)	5.x (Maxwell)	6.0 (Pascal)	6.1/6.2 (Pascal)	7.x (Volta, Turing)	8.x (Ampere)	9.x (Hopper/Ada)
# warp sched. / SM	2	2	4	4	2	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	?
SM busy with # warps	**22**	**22 + ILP**	**44 + ILP**	**36**	**12**	**24**	**16**	**16**	**4*?**

see NVIDIA CUDA C Programming Guides (different versions) performance guidelines/multiprocessor level; compute capabilities

ALU Instruction Latencies and Instructs. / SM

CC	2.0 (Fermi)	2.1 (Fermi)	3.x (Kepler)	5.x (Maxwell)	6.0 (Pascal)	6.1/6.2 (Pascal)	7.x (Volta, Turing)	8.x (Ampere)	9.x (Hopper/Ada)
# warp sched. / SM	2	2	4	4	2	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	?
SM busy with # warps	**22**	**22 + ILP**	**44 + ILP**	**36**	**12**	**24**	**16**	**16**	**4*?**

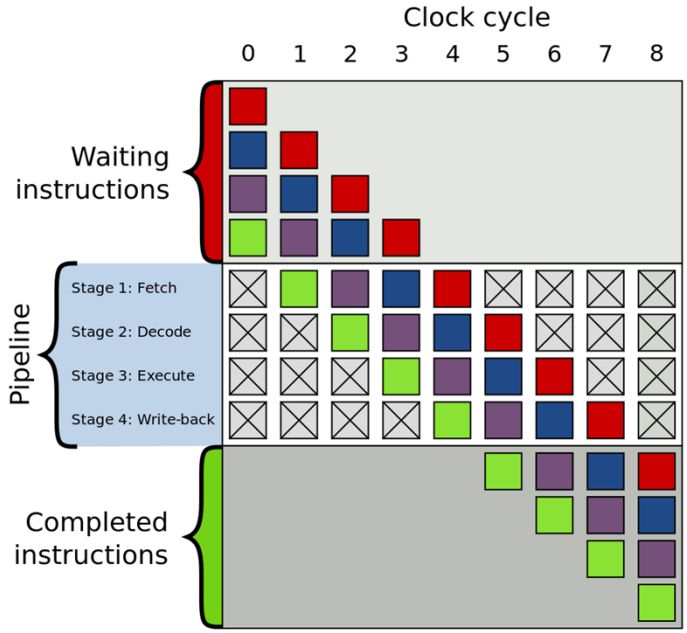
IF no other stalls occur!
(i.e., except inst. pipe hazards)

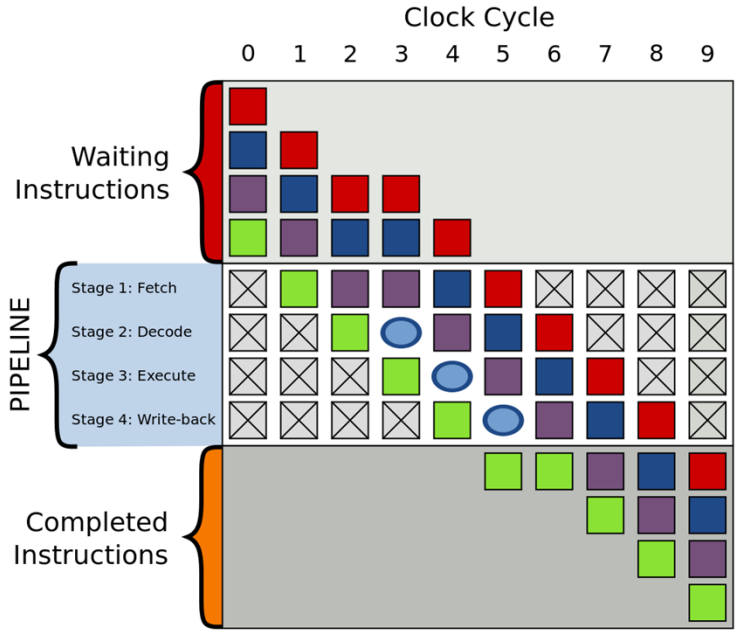
see NVIDIA CUDA C Programming Guides (different versions) performance guidelines/multiprocessor level; compute capabilities

Instruction Pipelining

Most basic way to exploit instruction-level parallelism (ILP)

Problem: hazards (different solutions: bubbles, …)





wikipedia

https://en.wikipedia.org/wiki/Instruction_pipelining
https://en.wikipedia.org/wiki/Classic_RISC_pipeline

Thank you.