

# **CS 380 - GPU and GPGPU Programming**

## **Lecture 8: GPU Architecture, Pt. 6**

Markus Hadwiger, KAUST

# Reading Assignment #4 (until Sep 29)



## Read (required):

- Get an overview of NVIDIA Ampere (GA102 and A100) GPU white papers:

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

<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-hopper-architecture/nvidia-h100-tensor-c>

- Get an overview of NVIDIA Blackwell (GB202) GPU white papers:

<https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf>

<https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/NVIDIA-RTX-Blackwell-PRO-GPU-Architecture-v1.0.pdf>

## Read (optional):

- Look at the “Tuning Guides” for different architectures in the CUDA SDK
- PTX Instruction Set Architecture (9.0): <https://docs.nvidia.com/cuda/parallel-thread-execution/>  
Read Chapters 1 – 3; get an overview of Chapter 9;  
browse through the other chapters to get a feeling for what PTX looks like
- CUDA SASS ISA (13.0), Chap. 6: [https://docs.nvidia.com/cuda/pdf/CUDA\\_Binary\\_Uutilities.pdf](https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Uutilities.pdf)

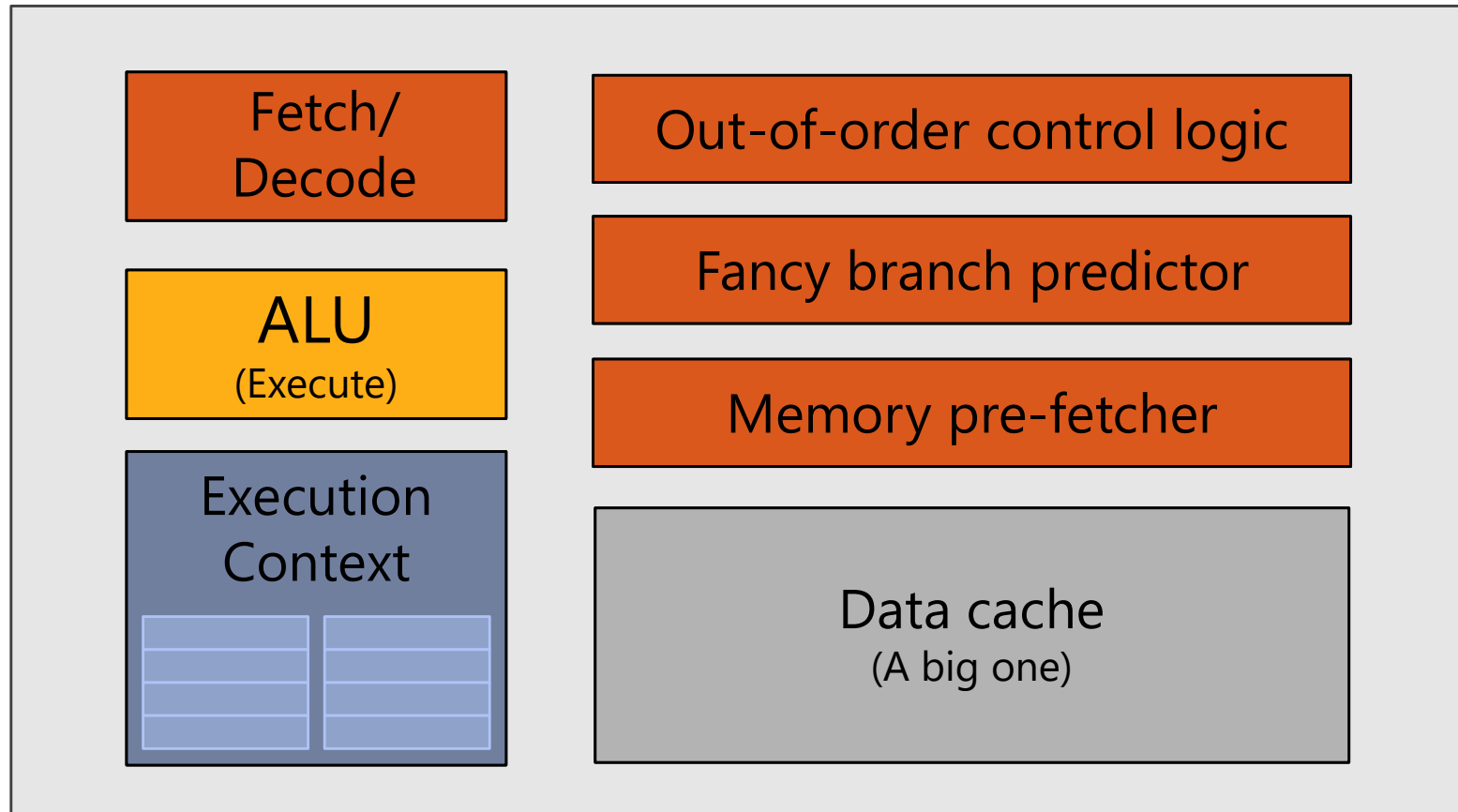
# **GPU Architecture: General Architecture**

# **GPU Architecture**

## **Big Idea #1**

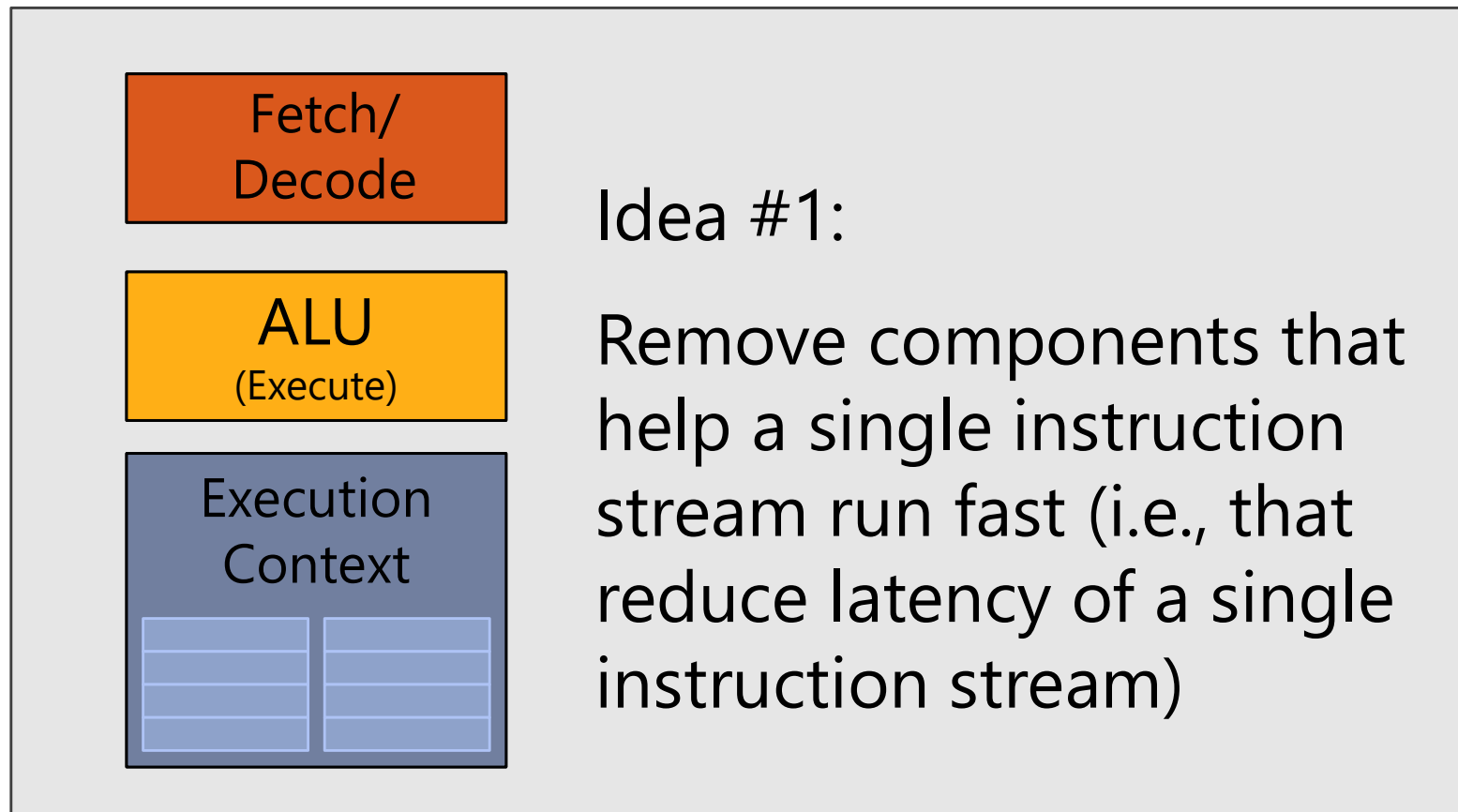
# CPU-“style” cores

---



# Idea #1: Slim down

---



**Goal: Increase (peak) throughput**

# Sixteen cores (sixteen fragments in parallel)

→ **16x peak throughput!**



16 cores = 16 simultaneous instruction streams

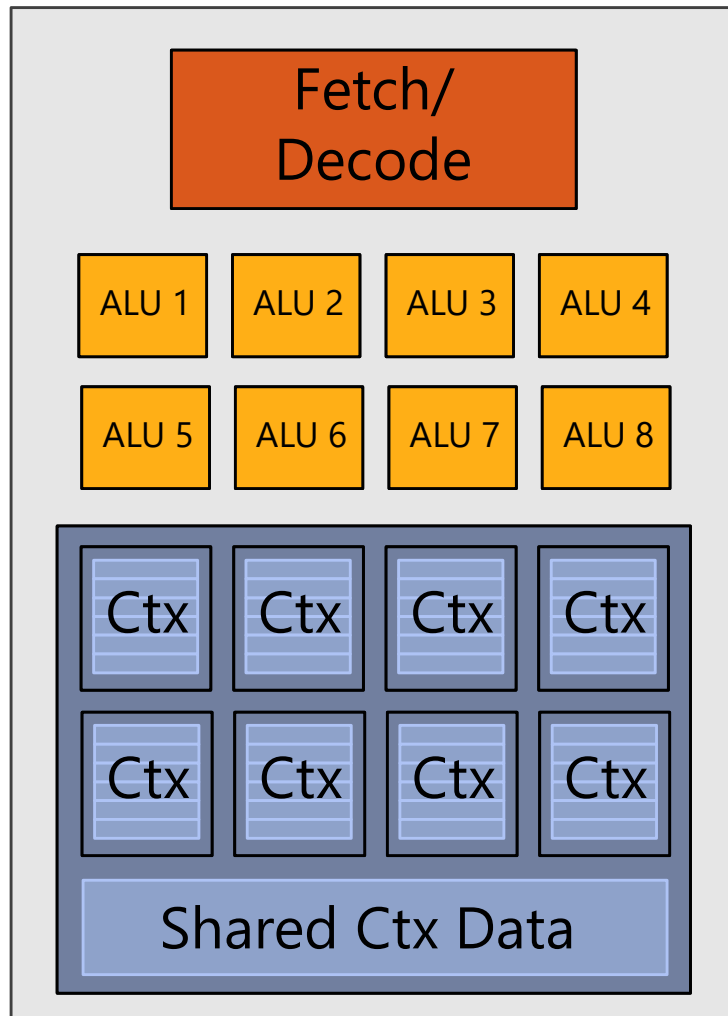
# **GPU Architecture**

## **Big Idea #2**



# Idea #2: Add ALUs

---



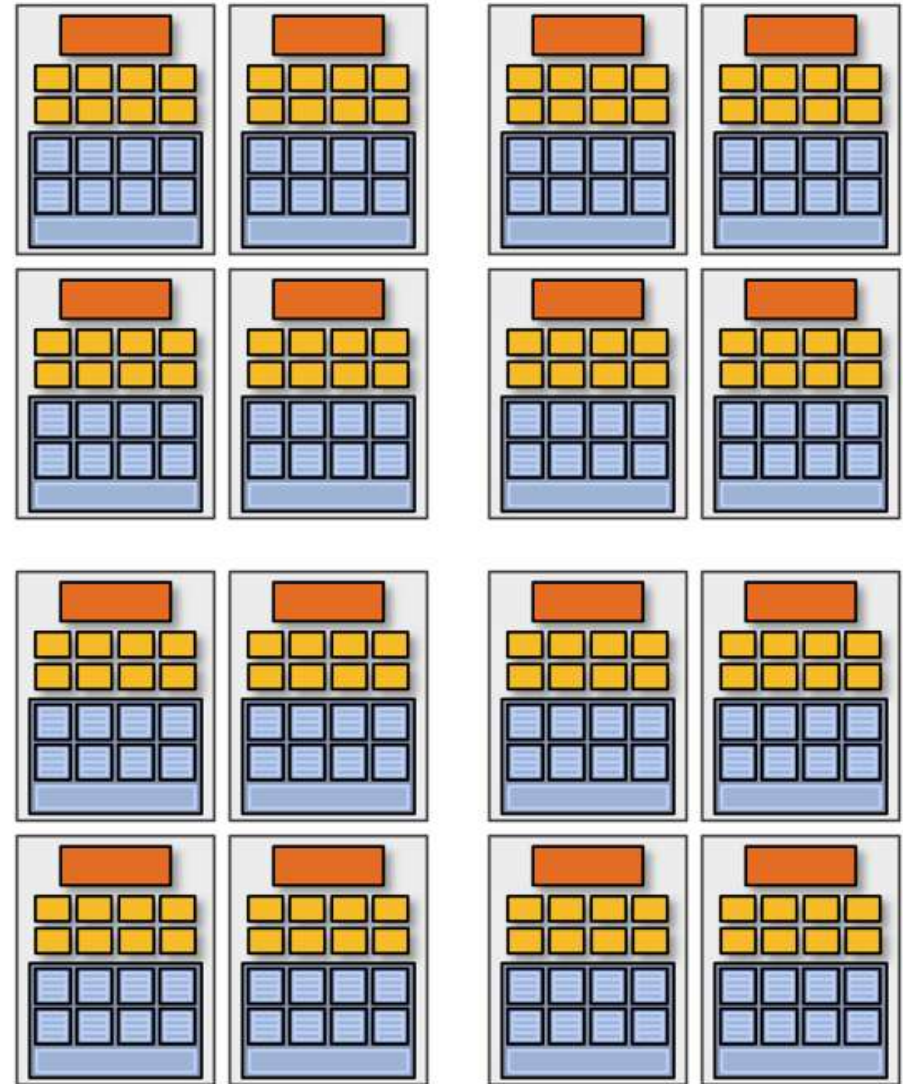
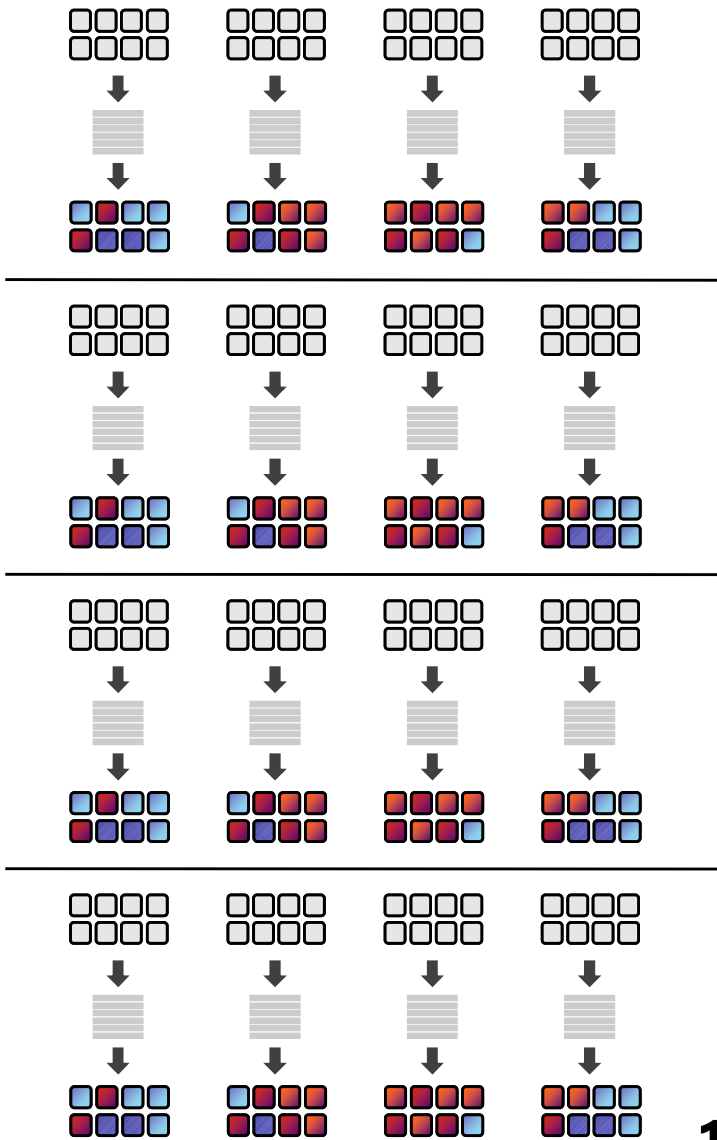
Idea #2:

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

SIMD processing

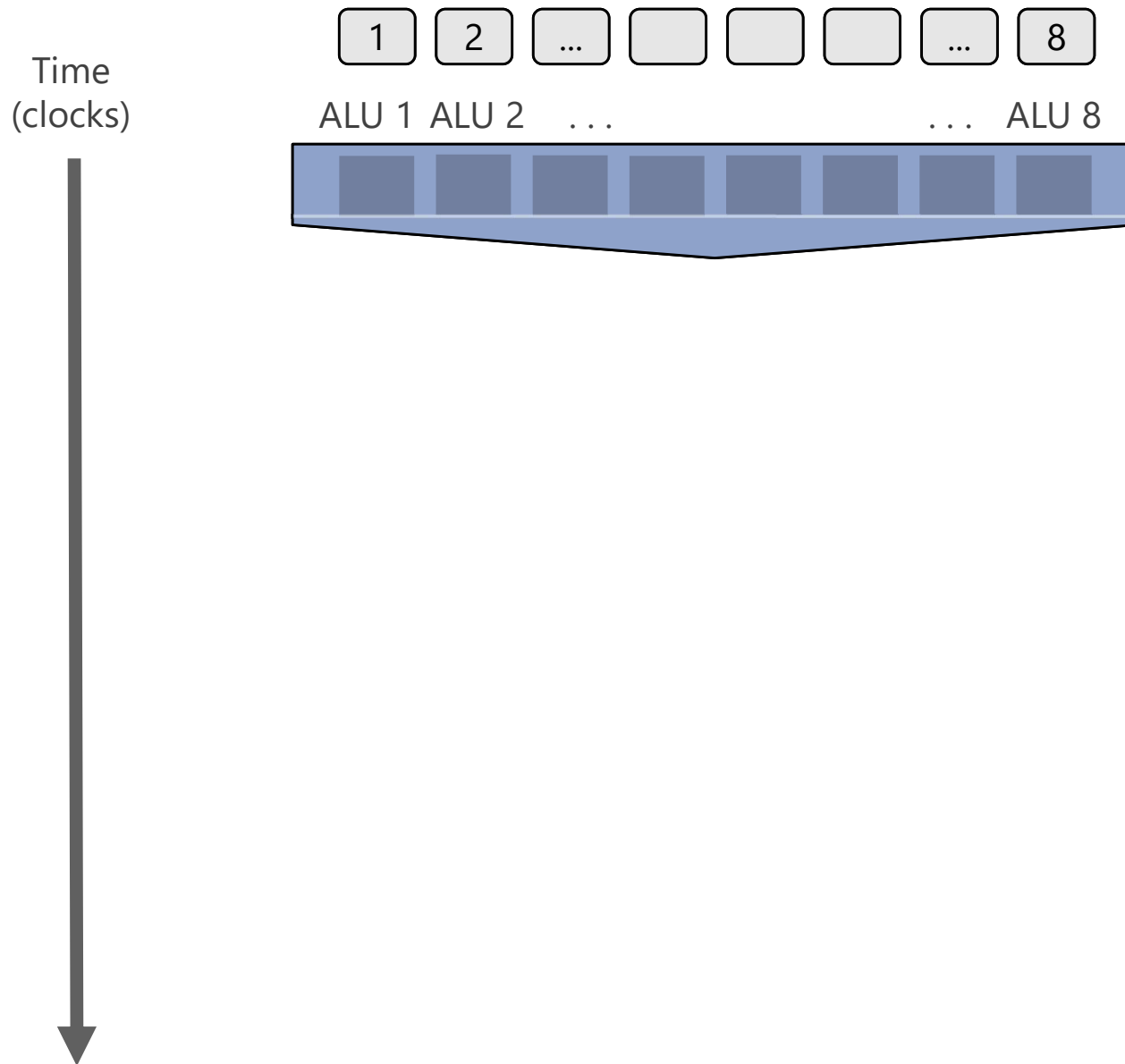
(or **SIMT**, SPMD)

# 128 fragments in parallel → 128x peak throughput!



**16** cores = **128** ALUs  
= **16** simultaneous instruction streams

# But what about branches?

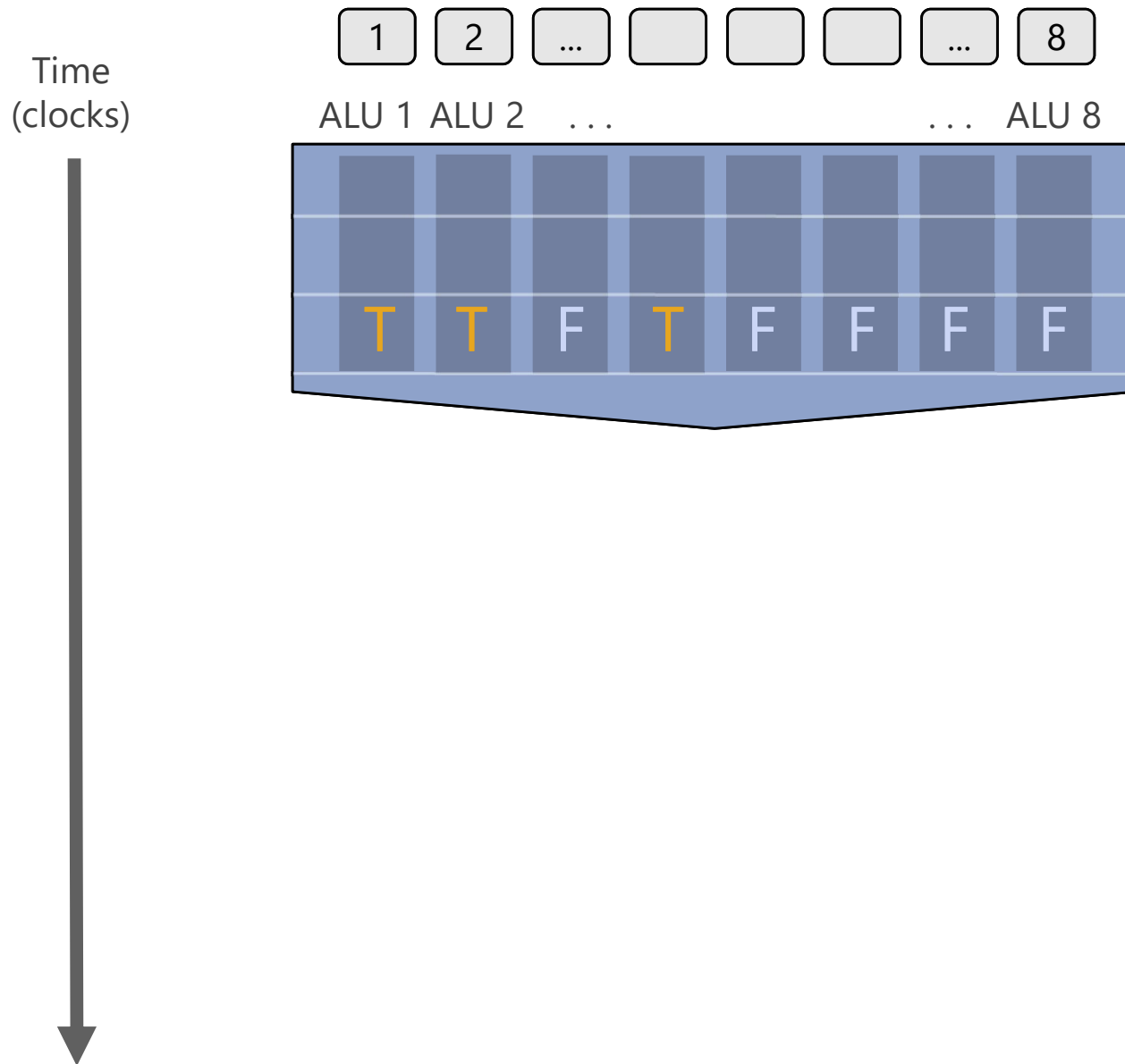


<unconditional  
shader code>

```
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?

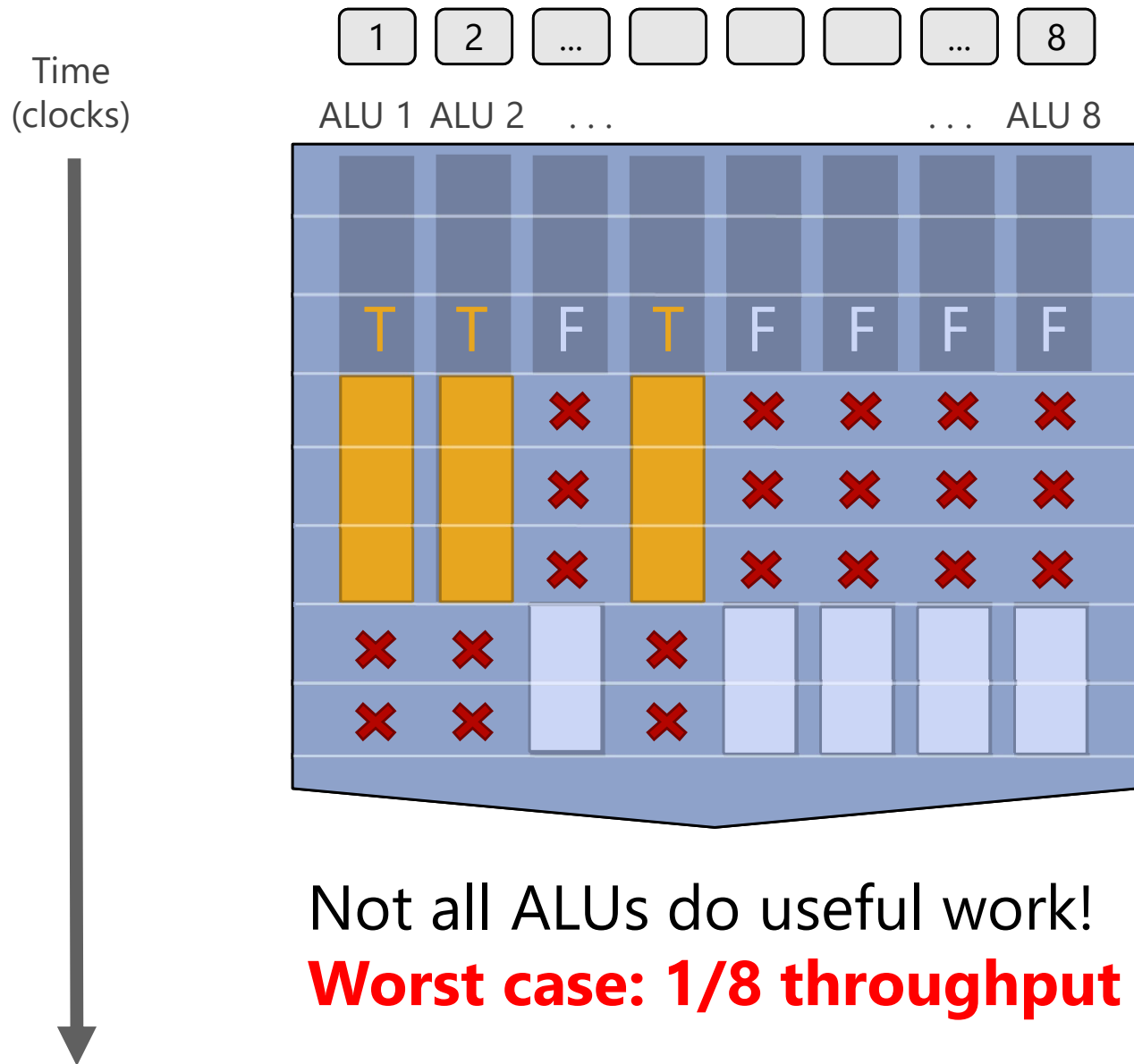


<unconditional  
shader code>

```
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 {
```

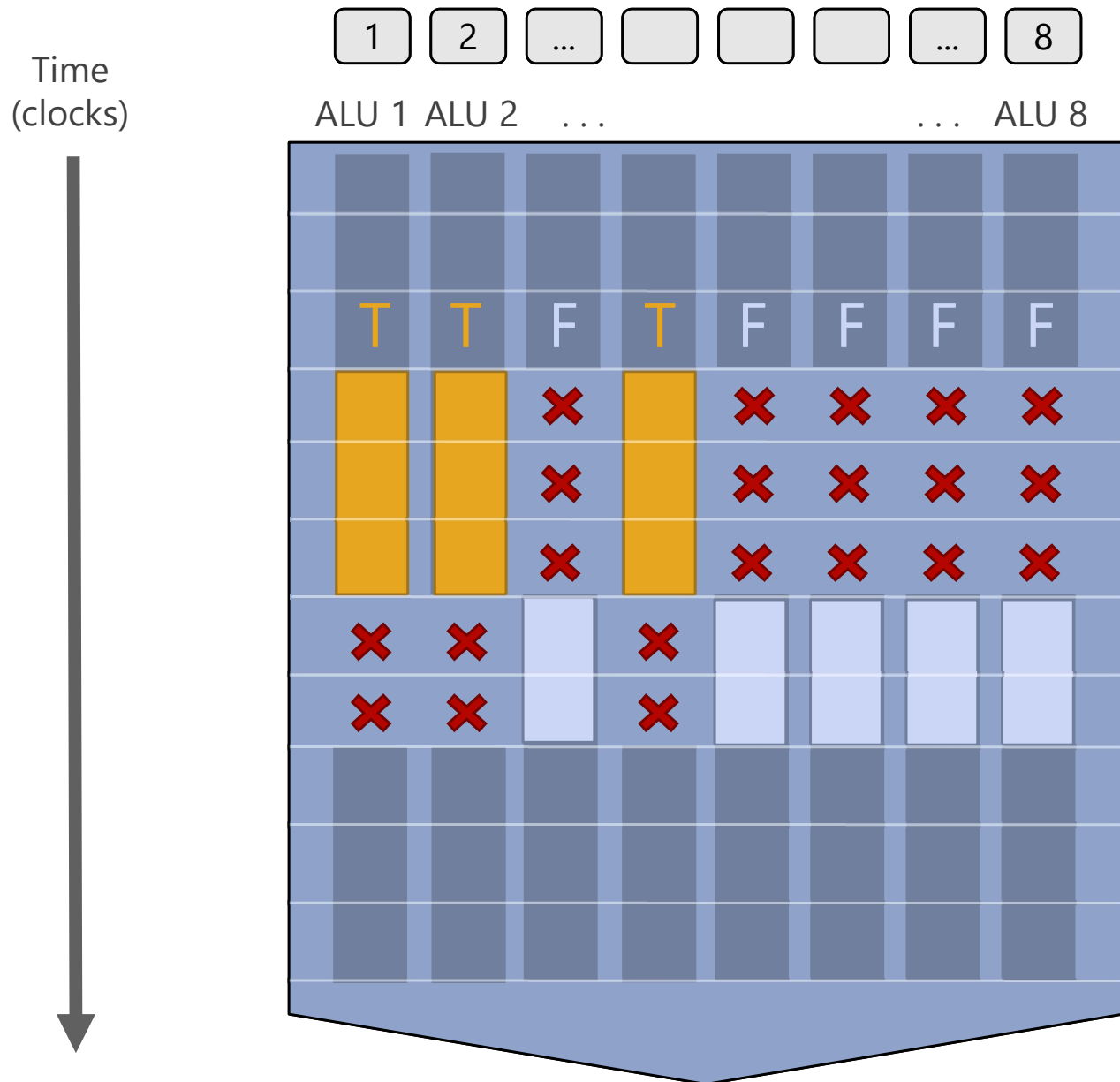
```
    x = 0;
```

```
    refl = Ka;
```

```
}
```

<resume unconditional  
shader code>

# 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>

# **GPU Architecture**

## **Big Idea #3**

# Idea #3: Interleave execution of groups

---

But we have **LOTS** of independent fragments.

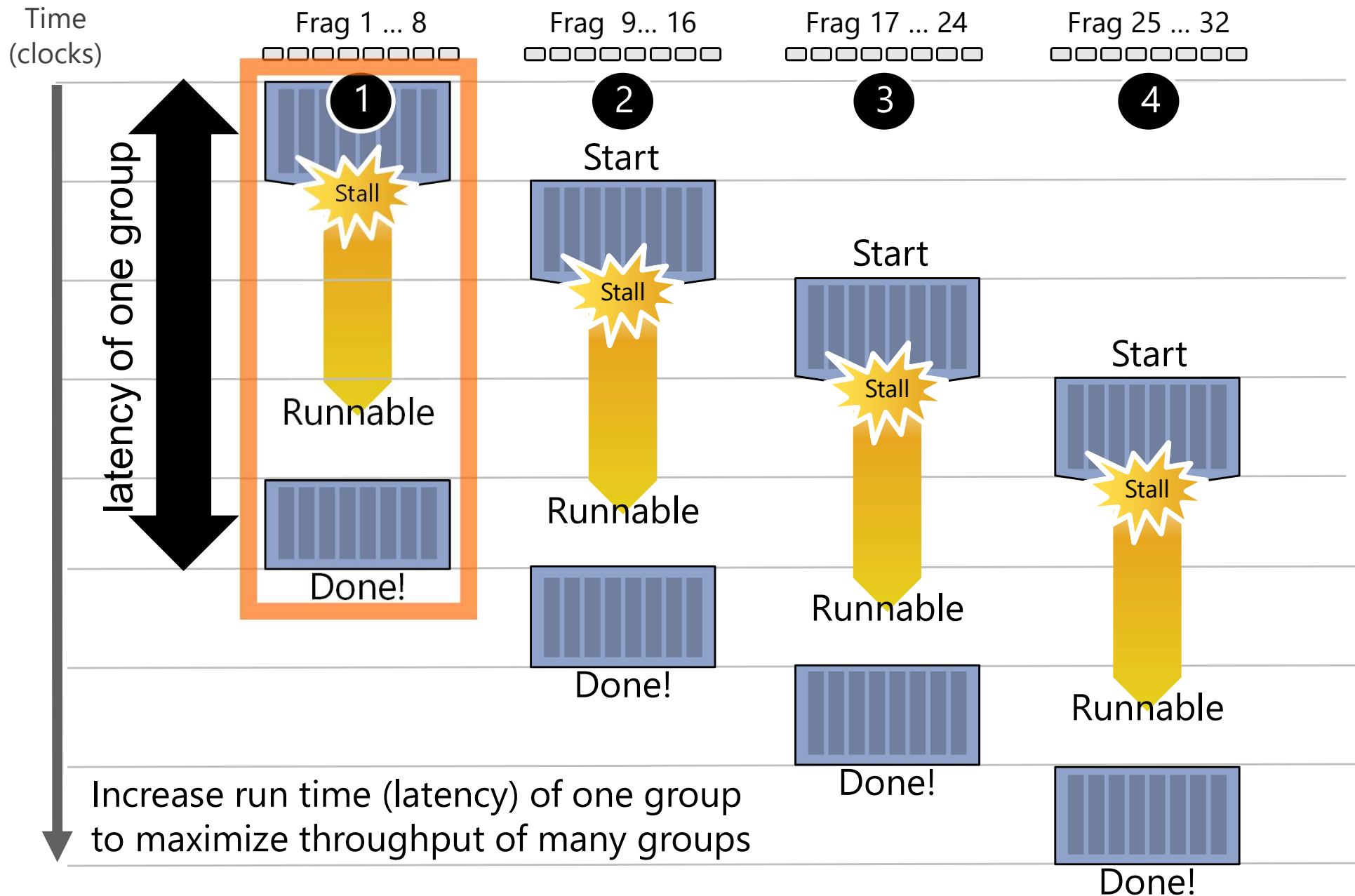
## Idea #3:

Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.  
(including instruction pipeline hazards)

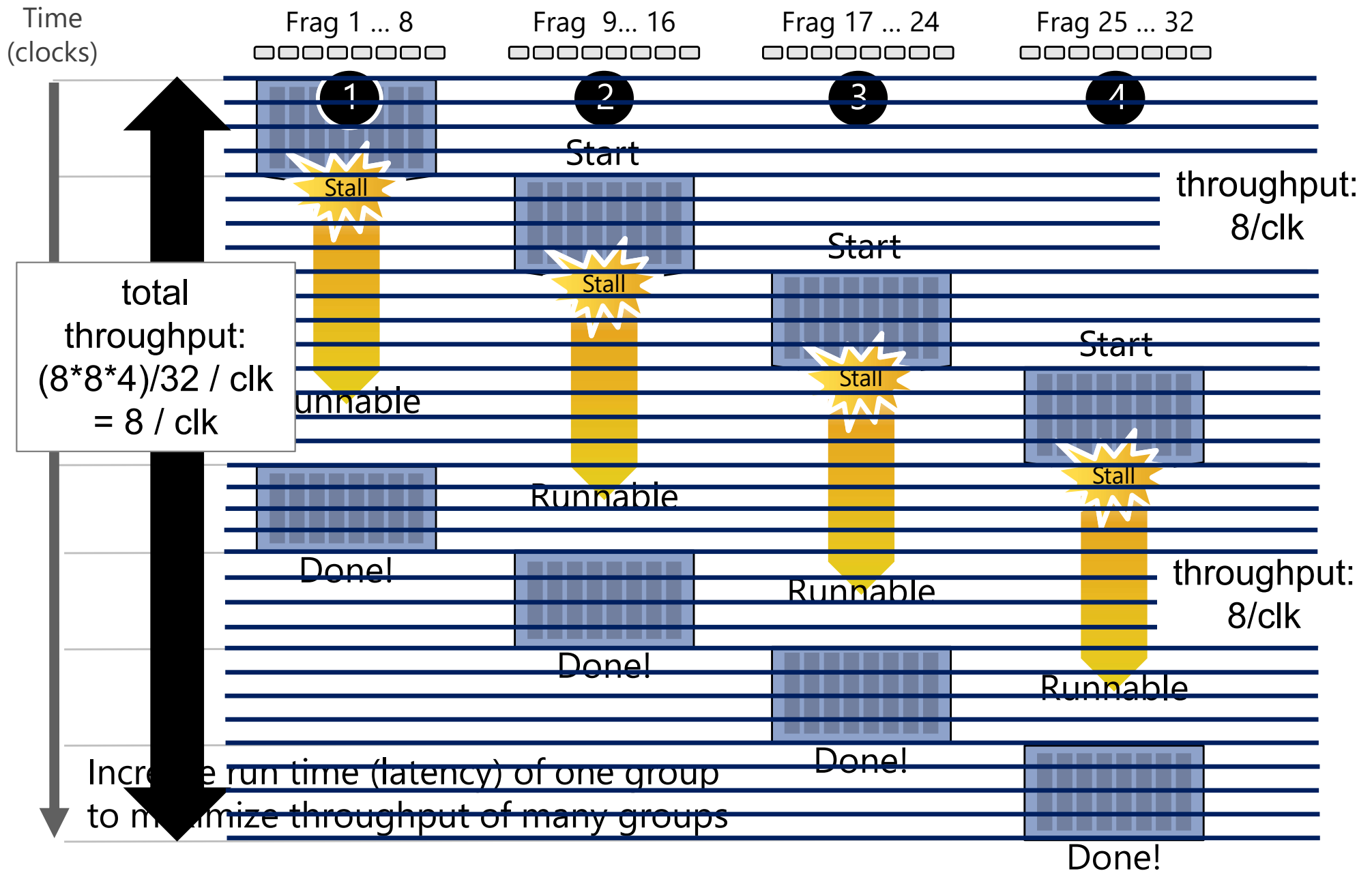
***Increases the latency of a single group of fragments,  
but keeps the throughput as close to peak as possible!***



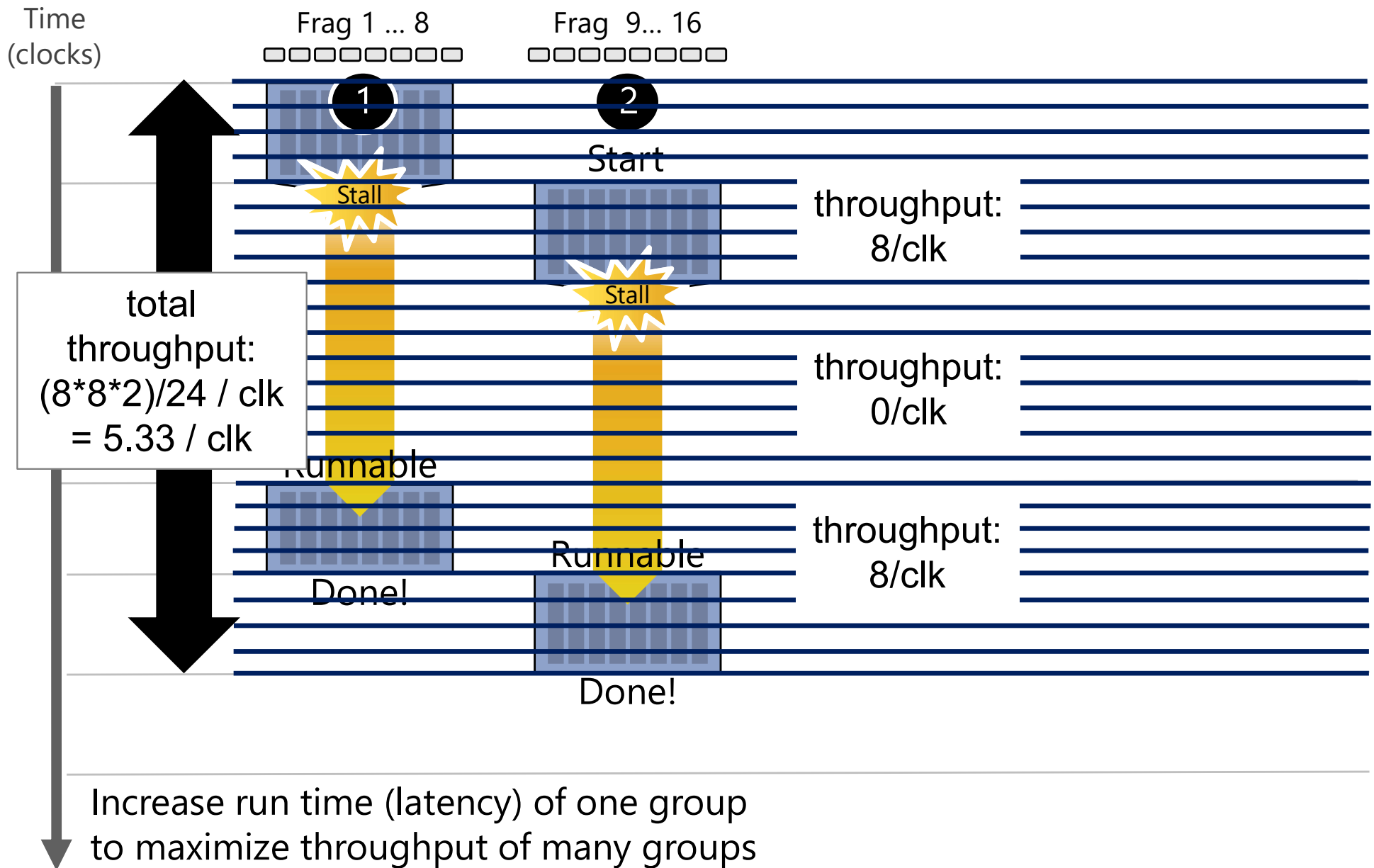
# Hiding shader stalls



# Throughput! (4 groups of threads)



# Throughput! (2 groups of threads)



# 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)
- Exploit ILP: Execute independent instructions (1) via pipelined execution (instr. pipe), or even (2) in multiple parallel instruction pipelines (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, which should be minimized)
- Exploit TLP: Execute operations/instructions from multiple threads in parallel (which also needs multiple parallel instruction pipelines)
- **On GPUs: main type of parallelism**

more types:

- Bit-level parallelism (processor word size: 64 bits instead of 32, etc.)
- Data parallelism (SIMD/vector instructions), task parallelism, ...

# Concepts: Latency Hiding



**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:

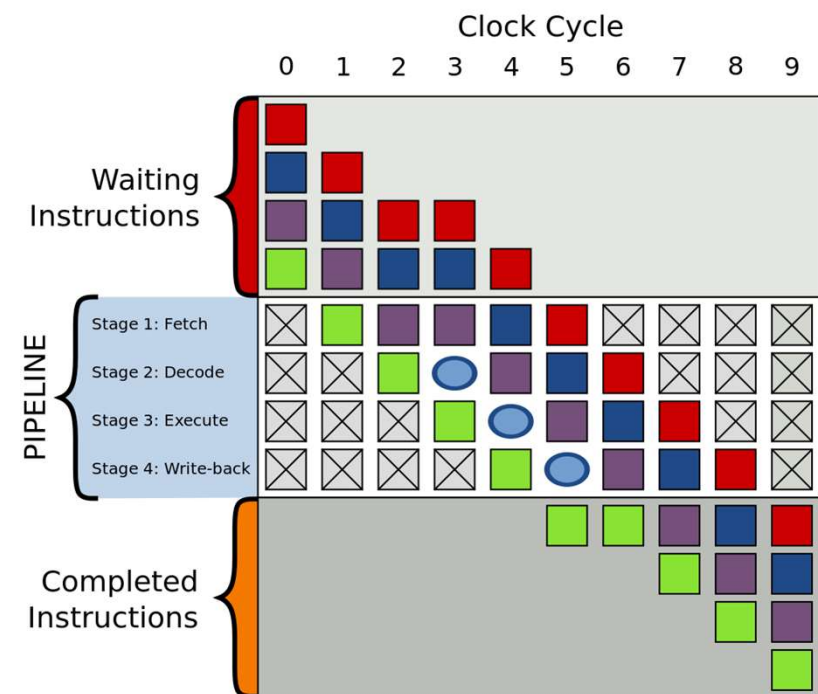
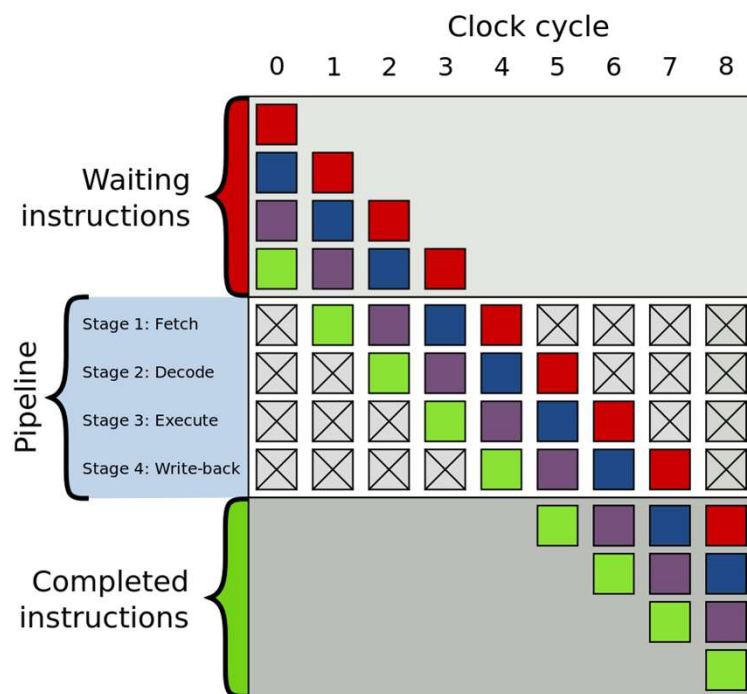
- **TLP: pull independent, not-stalling instruction from other thread group**
- ILP: pull independent instruction from down the inst. stream in same thread group
- Depending on GPU: TLP often sufficient, but sometimes also need ILP
- However: If in one cycle TLP doesn't work, ILP can jump in or vice versa

# 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/Instruction_pipelining)  
[https://en.wikipedia.org/wiki/Classic\\_RISC\\_pipeline](https://en.wikipedia.org/wiki/Classic_RISC_pipeline)

# Concepts: SM Occupancy in CUDA (*TLP!*)



We need to hide latencies from

- Instruction pipelining hazards (RAW – read after write, etc.)  
(also: branches; behind branch, fetch instructions from different instruction stream)
- Memory access latency

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

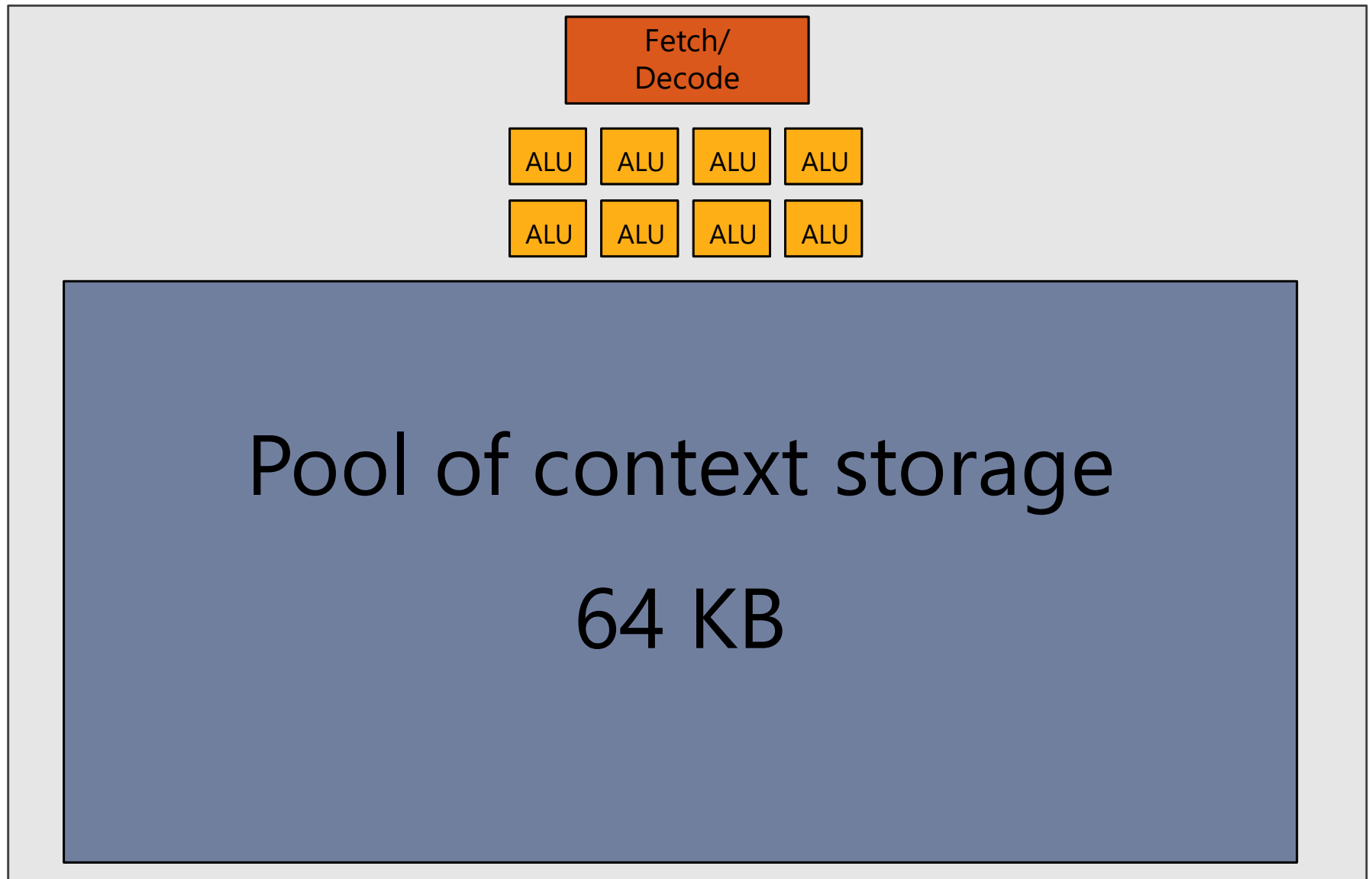
Second type of latency: only need to hide if it does occur (of course not unusual)

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

See run time occupancy API, or Nsight Compute: <https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator>

# Storing contexts

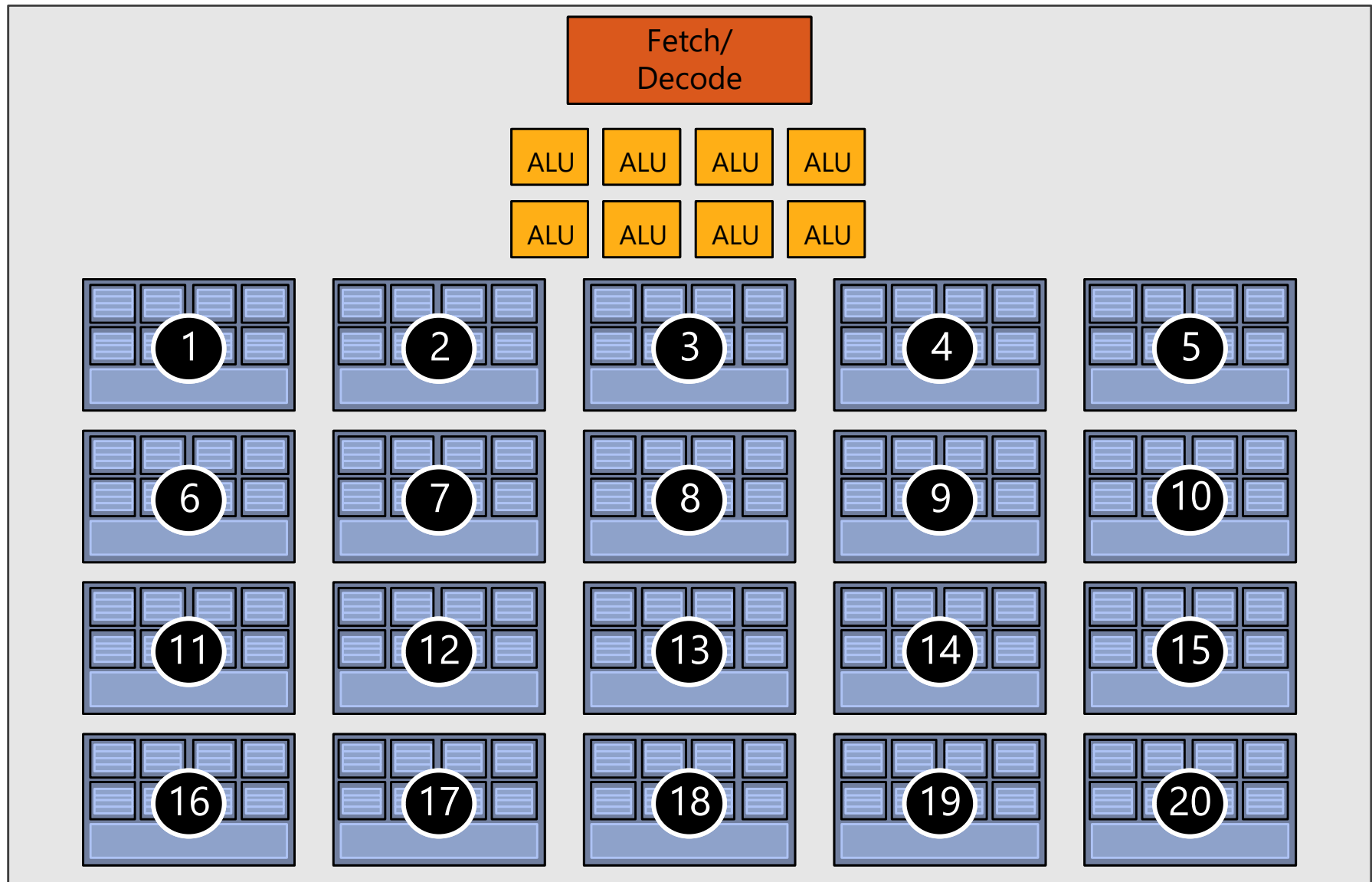
---



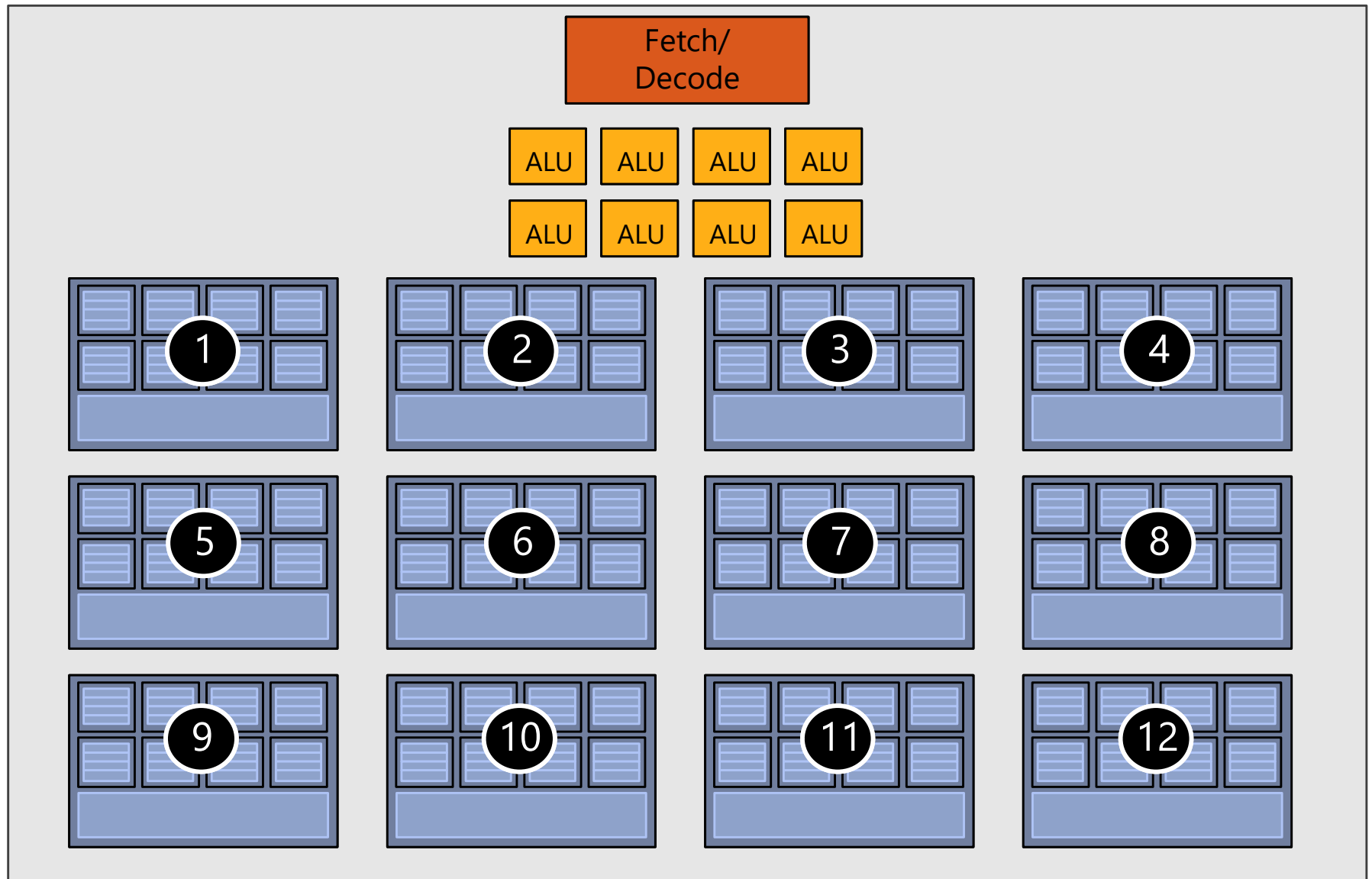


# Twenty small contexts (few regs/thread)

(maximal latency hiding ability)

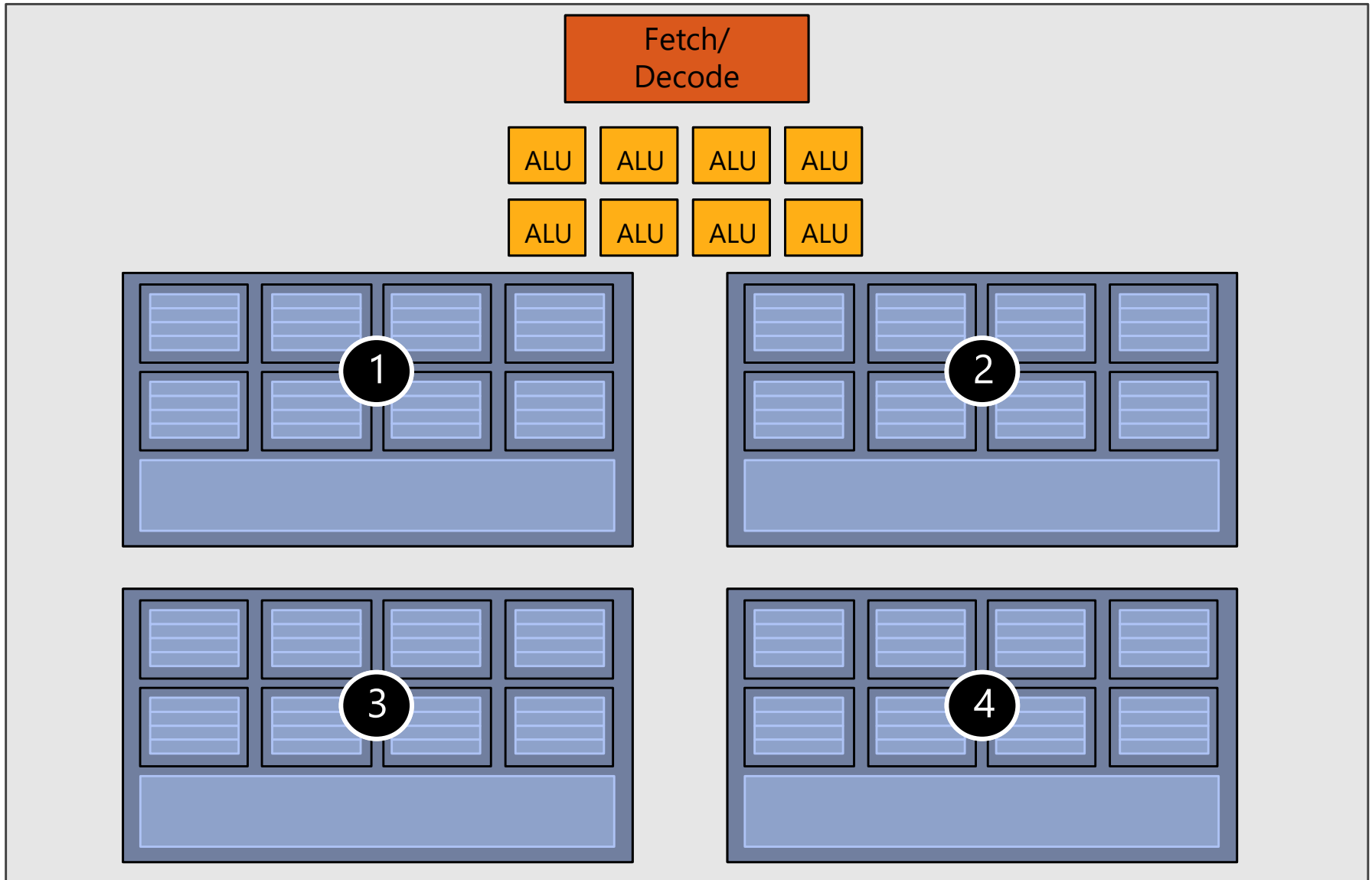


# Twelve medium contexts (more regs/th.)



# Four large contexts (many regs/thread)

(low latency hiding ability)



# Complete GPU

16 cores

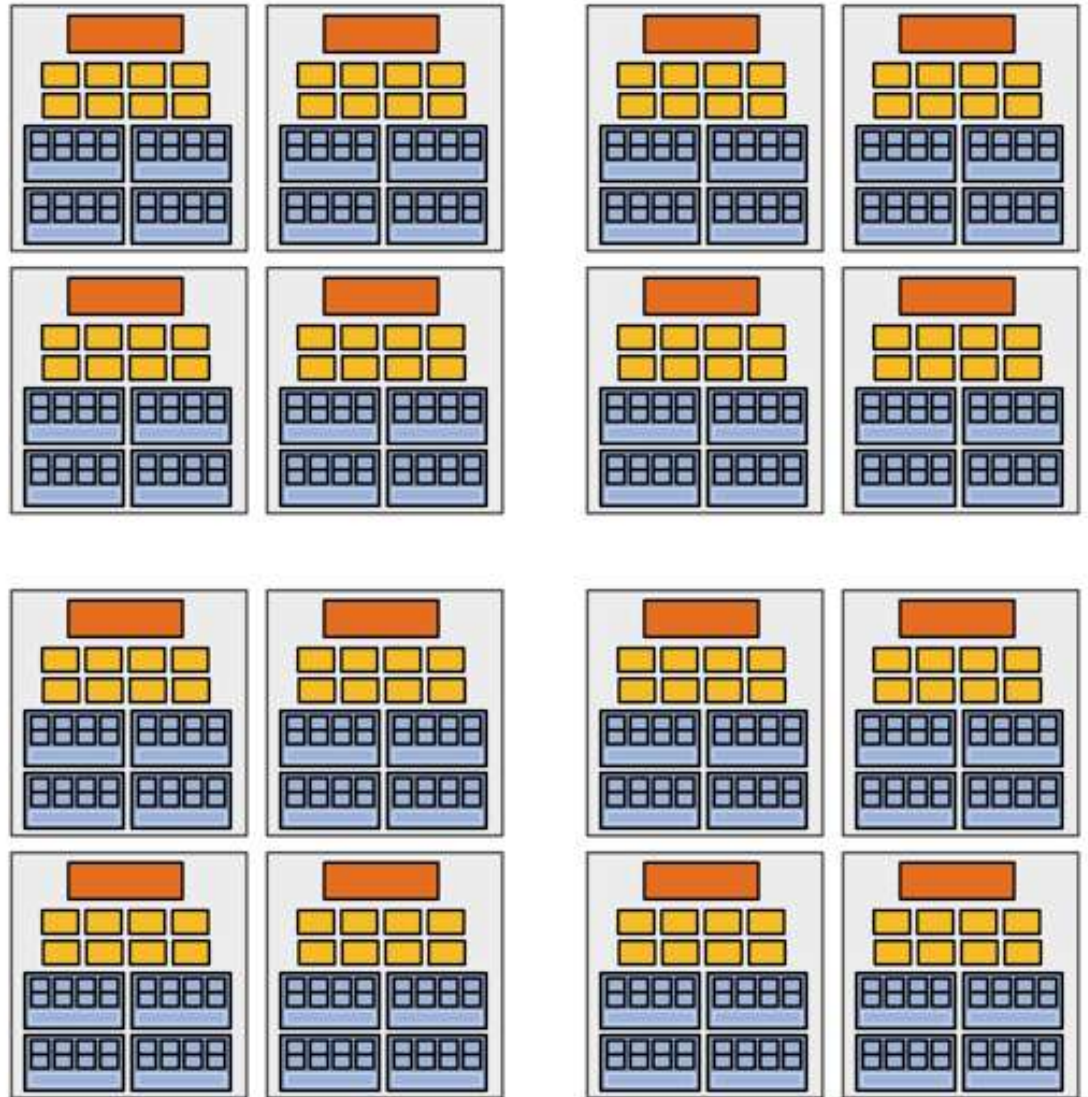
8 mul-add <sub>[mad]</sub> ALUs per core  
(8\*16 = **128** total)

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 <sub>[mad]</sub> \* 1G)



# Complete GPU

16 cores

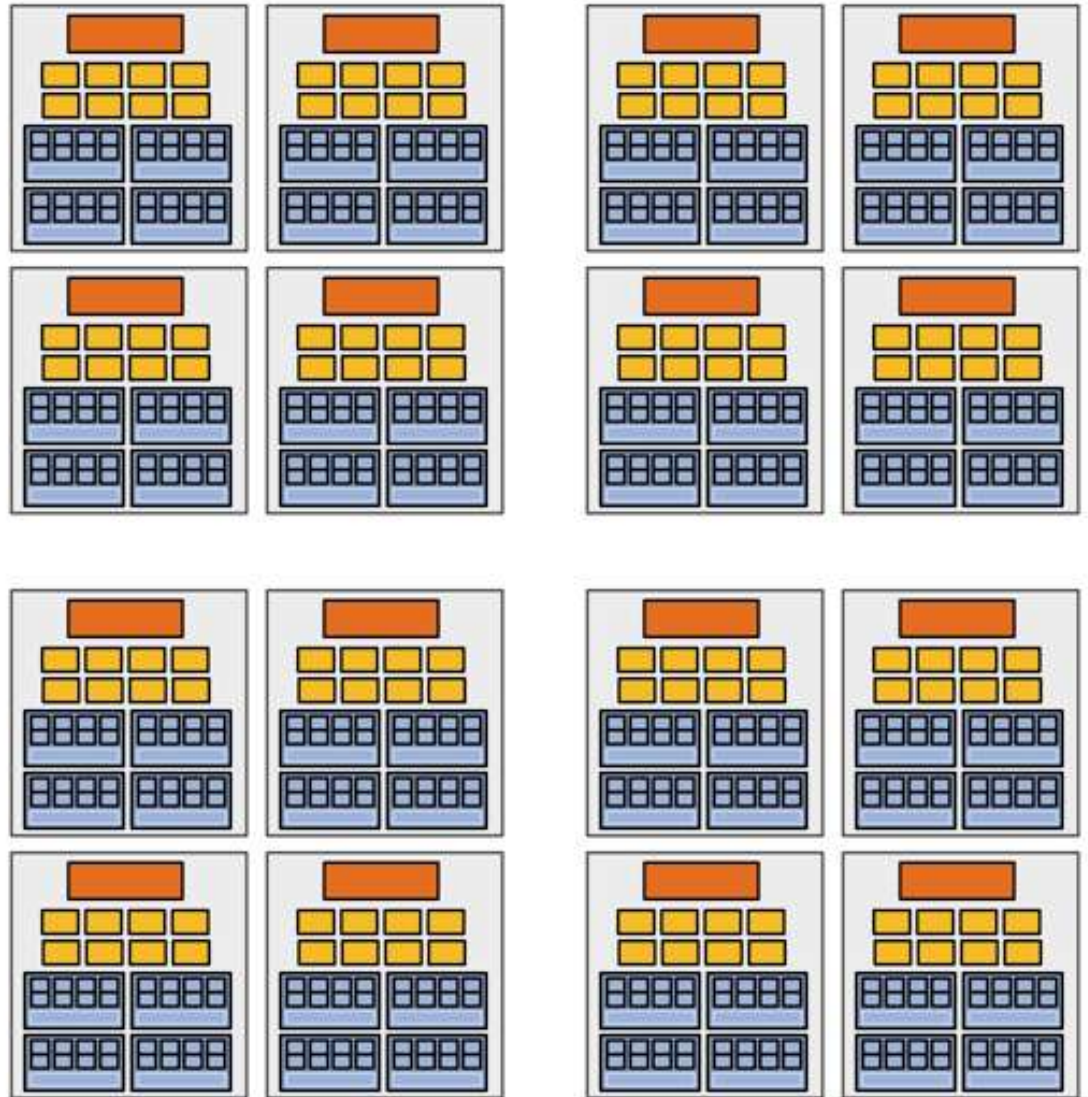
8 mul-add<sub>[mad]</sub> ALUs per core  
(8\*16 = **128** total)

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<sub>[mad]</sub> \* 1G)





# Complete GPU

16 cores

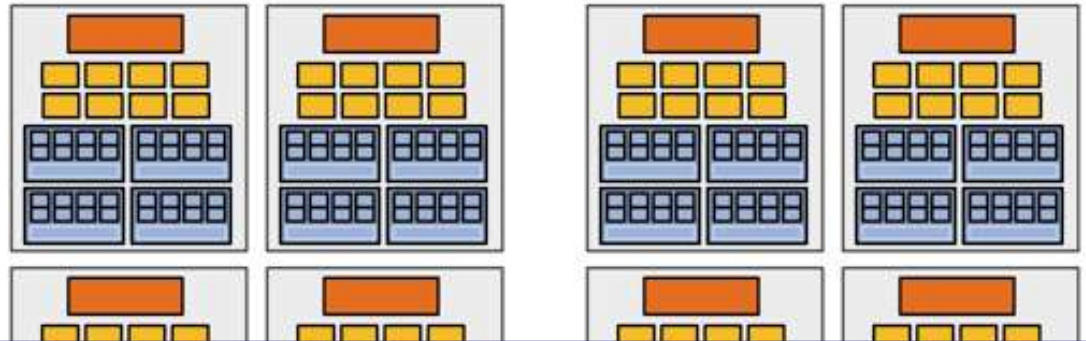
8 mul-add<sub>[mad]</sub> ALUs per core  
(8\*16 = **128** total)

16 simultaneous  
instruction streams

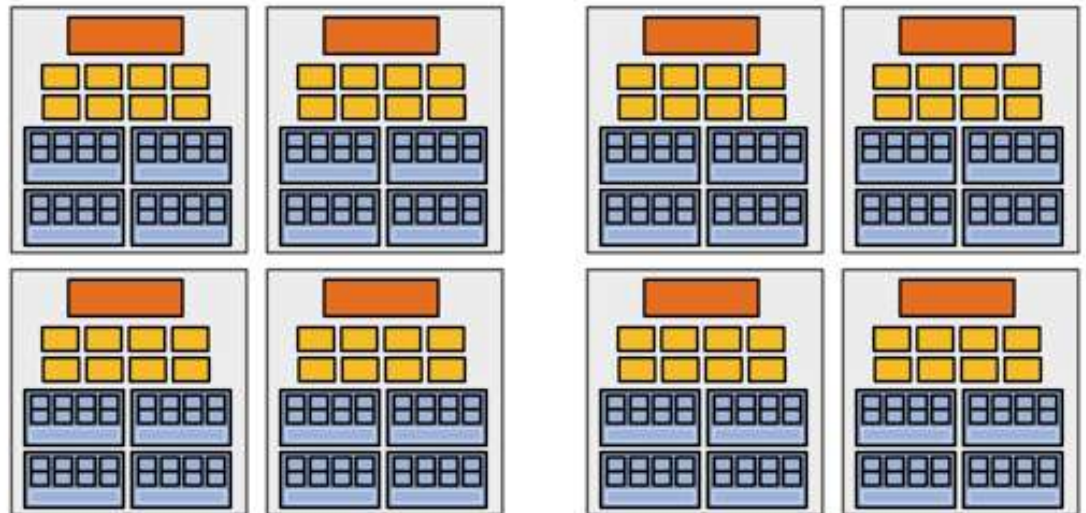
64 (4\*16) concurrent (but  
interleaved) instruction streams

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

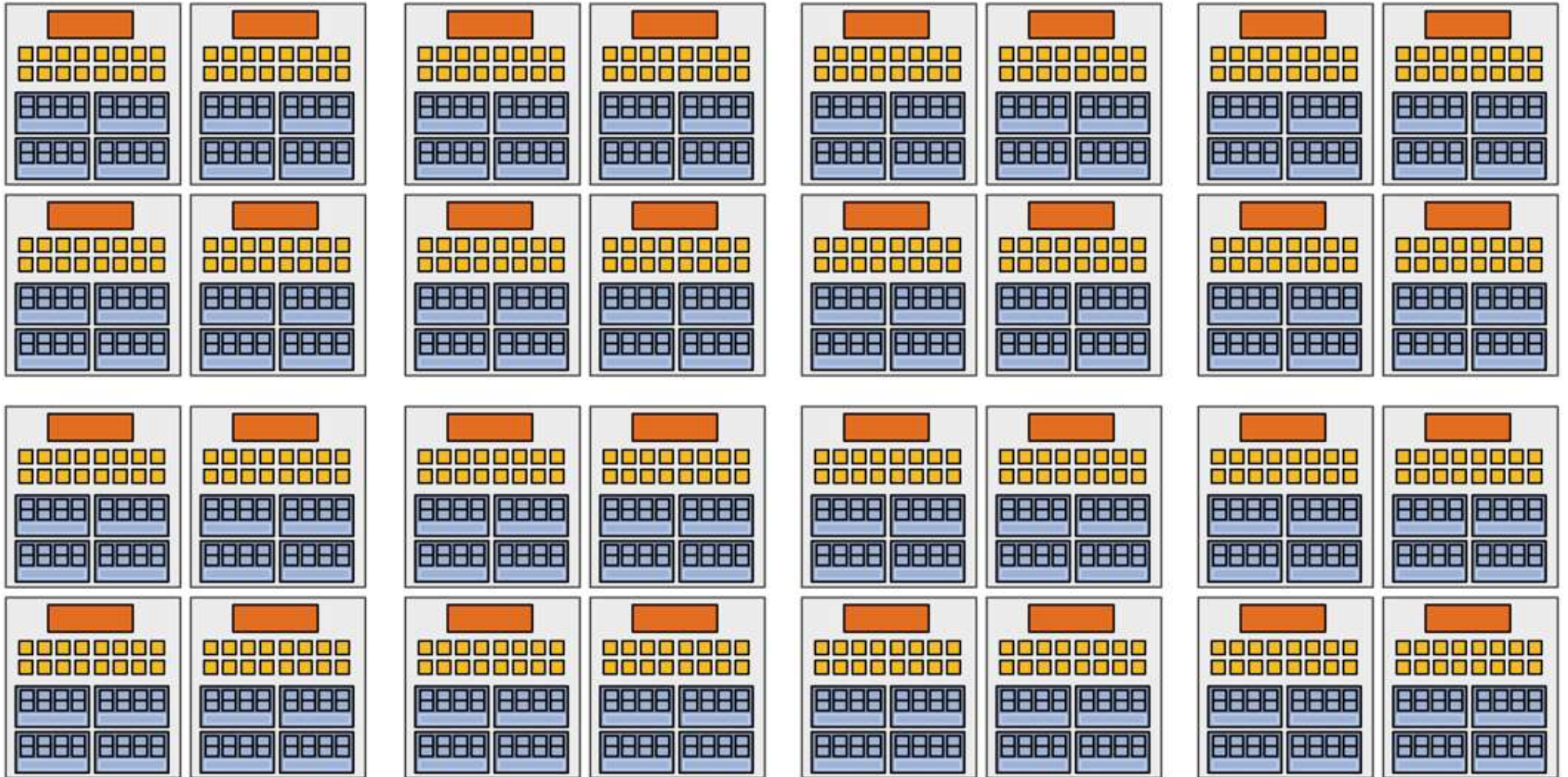
= **256 GFLOPs** (@ 1GHz)  
(**128** \* 2<sub>[mad]</sub> \* 1G)



Beware: this is either for a kernel that happens to have the occupancy shown here (4 groups of threads per core), or: the diagram could show maximum occupancy



# "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
3. Avoid latency stalls by interleaving execution of many groups of fragments
  - When one group stalls, work on another group

**GPUs are here!**  
(usually)



# **GPU Architecture: Real Architectures**

# NVIDIA Architectures (since first CUDA GPU)



## Tesla [CC 1.x]: 2007-2009

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

## Fermi [CC 2.x]: 2010 (2011, 2012, 2013, ...)

- GF100, ... (GTX 480, ...)  
GF104, ... (GTX 460, ...)  
GF110, ... (GTX 580, ...)

## Kepler [CC 3.x]: 2012 (2013, 2014, 2016, ...)

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

## Maxwell [CC 5.x]: 2015

- GM107, ... (GTX 750Ti, ...); [Nintendo Switch]  
GM204, ... (GTX 980, Titan X, ...)

## Pascal [CC 6.x]: 2016 (2017, 2018, 2021, 2022, ...)

- GP100 (Tesla P100, ...)
- GP10x: x=2,4,6,7,8, ...  
(GTX 1060, 1070, 1080, Titan X *Pascal*, Titan Xp, ...)

## Volta [CC 7.0, 7.2]: 2017/2018

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

## Turing [CC 7.5]: 2018/2019

- TU102, TU104, TU106, TU116, TU117, ...  
(Titan RTX, RTX 2070, 2080 (Ti), GTX 1650, 1660, ...)

## Ampere [CC 8.0, 8.6, 8.7, 8.8]: 2020

- GA100, GA102, GA104, GA106, ...; [Nintendo Switch 2]  
(A100, RTX 3070, 3080, 3090 (Ti), RTX A6000, ...)

## Hopper [CC 9.0], Ada Lovelace [CC 8.9]: 2022/23

- GH100, AD102, AD103, AD104, AD106, AD107, ...  
(H100, L40, RTX 4080 (12/16 GB), RTX 4090,  
RTX 6000 (Ada), ...)

## Pascal [CC 6.x]: 2016 (2017, 2018, 2021, 2022, ...) Blackwell [CC 10.0, 10.1(11.0), 10.3, 12.0, 12.1]: 2024/2025

- GB100/102, GB200, GB202/203/205/206/207, ...  
(RTX 5080/5090, GB100/GB200 NVL72, HGX B100/B200,  
RTX 4000/5000/6000 PRO Blackwell, ...)

# Concepts: Latency Hiding (Latency Tolerance)



Main goal: Avoid that instruction *throughput* goes below peak

**ILP:** Hide instruction pipeline latency of one instruction by pipelined execution of *independent* instruction from same thread

**TLP:** Hide any latency occurring for one thread (group/warp/wavefront) by *executing a different thread (group/warp/wavefront)* as soon as current thread (group/warp/wavefront) stalls:

→ *Total throughput does not go down*

## GPUs

- **TLP: pull independent, not-stalling instruction from other thread group**
- ILP: pull independent instruction from same thread group (instruction stream)
- 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\*

(\*depending on actual microarchitecture)

# ILP vs. TLP on GPUs



## Main observations

- Each time unit (usually one clock cycle), a new instruction *without dependencies* should be dispatched to functional units (ALUs, SFUs, ...)
- *Instruction* is a *group of threads* that is executing the same instruction: CUDA warp (32 threads), wavefront (32 or 64 threads), ...
- Where can this instruction come from?
  - TLP: from another runnable warp (i.e., different instruction stream)
  - ILP: from the same warp (i.e., the same instruction stream)

## How many instructions/warps per time unit (clock cycle)?

- “Scalar” pipeline ( $CPI=1.0$ ): **TLP sufficient** (if enough warps); **can exploit ILP** (next instruction either from different warp, or from same warp)
- “Superscalar” ( $CPI<1.0$ ) pipeline: dispatch more than one instruction per cycle, ( $\#dispatchers > \#warp\ schedulers$ ): **need ILP!**

( $CPI = \text{clocks per instruction}$ )

# Example: “Scalar” GF100

Main concept here:

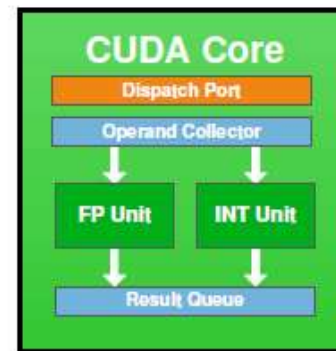
There is one instruction dispatcher  
(dispatch unit / fetch/decode unit)  
per warp scheduler  
(warp selector)

Details later...

Ignore less important subtleties...

GF100 has two warp schedulers, not one,  
and each 32-thread instruction is executed  
over two clock cycles, not one, etc.

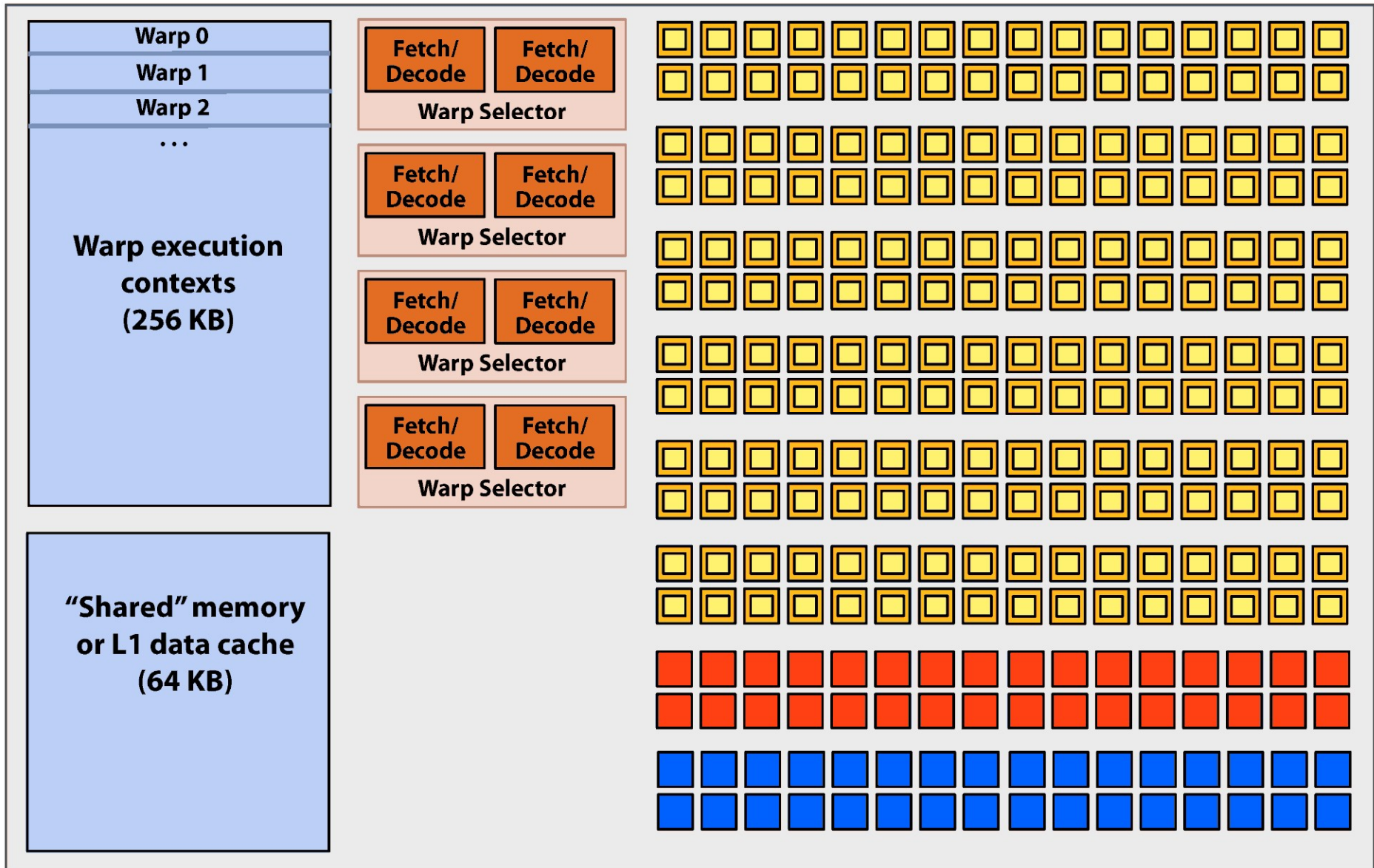
**Caveat on NVIDIA diagrams:** if two dispatchers per warp scheduler are shown, it still doesn't mean that the ALU pipeline is “superscalar” (often, the second dispatcher dispatches to a *non-ALU* pipeline)  
... need to look at CUDA programming guide info, also given  
in our tables in row “# ALU dispatch / warp sched.”






# Example: “Superscalar” ALUs in SM Architecture

NVIDIA Kepler GK104 architecture SMX unit (one “core”)



 = SIMD function unit,  
control shared across 32 units  
(1 MUL-ADD per clock)

 = “special” SIMD function unit,  
control shared across 32 units  
(operations like sin/cos)

 = SIMD load/store unit  
(handles warp loads/stores, gathers/scatters)

# Instruction Throughput



Instruction throughput numbers in older (<13) CUDA C Programming Guide (Chapter 8.4)

	Compute Capability										
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9	9.0
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128	256 <small>128 for __nv_bfloat16</small>		128	256 <small>128 for __nv_bfloat16</small>
32-bit floating-point add, multiply, multiply-add	192	128		64	128		64		128		
64-bit floating-point add, multiply, multiply-add	64 <small>8 for GeForce GPUs, except for Titan GPUs</small>	4		32	4		32 <small>2 for compute capability 7.5 GPUs</small>	32	2	2	64

# Instruction Throughput



Instruction throughput numbers in older (<13) CUDA C Programming Guide (Chapter 8.4)

	Compute Capability									
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9 9.0
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm ( <code>_log2f</code> ), base 2 exponential ( <code>exp2f</code> ), sine ( <code>_sinf</code> ), cosine ( <code>_cosf</code> )	32			16		32			16	
32-bit integer add, extended-precision add, subtract, extended-precision subtract	160	128		64		128			64	
32-bit integer multiply, multiply-add, extended-precision multiply-add	32	Multiple instruct.							64	
									32 for extended-precision	

list continues...



# Instruction Throughput



Instruction throughput numbers in CUDA 13 C Best Practices Guide (Chapter 12.1, Table 5)

Compute Capability	7.5 Turing	8.0 Ampere	8.6 Ada	8.9 Ada	9.0 Hopper	10.0 Blackwell	12.0 Blackwell
16-bit floating-point add, multiply, multiply-add (2-way SIMD): add.f16x2	64 <sup>3</sup>	128 <sup>4</sup> <sup>4</sup> 64 for __nv_bfloat16 <sup>3</sup> multiple instructions for __nv_bfloat16	64		128	64	
32-bit floating-point add, multiply, multiply-add: add.f32	64		128				
64-bit floating-point add, multiply, multiply-add: add.f64	2	32	2		64	64	2

# 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)	8.0/8.6 (Ampere)	8.9/9.0 (Ada, Hopper)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
<b>SM busy with # warps</b>	<b>22</b>	<b>22 + ILP</b>	<b>44 + ILP</b>	<b>36</b>	<b>12</b>	<b>24</b>	<b>16</b>	<b>16</b>	<b>16</b>	<b>16</b>

*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)	8.0/8.6 (Ampere)	8.9/9.0 (Ada, Hopper)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
<b>SM busy with # warps</b>	<b>22</b>	<b>22 + ILP</b>	<b>44 + ILP</b>	<b>36</b>	<b>12</b>	<b>24</b>	<b>16</b>	<b>16</b>	<b>16</b>	<b>16</b>

*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*

# 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)	8.0/8.6 (Ampere)	8.9/9.0 (Ada, Hopper)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
<b>SM busy with # warps</b>	<b>22</b>	<b>22 + ILP</b>	<b>44 + ILP</b>	<b>36</b>	<b>12</b>	<b>24</b>	<b>16</b>	<b>16</b>	<b>16</b>	<b>16</b>

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

*“superscalar”*

*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)	8.0/8.6 (Ampere)	8.9/9.0 (Ada, Hopper)	10.x/12.x (Blackwell)
# warp sched. / SM	2	2	4	4	2	4	4	4	4	4
# ALU dispatch / warp sched.	1 (over 2 clocks)	2 (over 2 clocks)	2	1	1	1	1	1	1	1
SM busy with # warps + inst	L	2L	8L	4L	2L	4L	4L	4L	4L	4L
inst. pipe latency (L)	22	22	11	9	6	6	4	4	4	4
SM busy with # warps	22	22 + ILP	44 + ILP	36	12	24	16	16	16	16

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

*“superscalar”*

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

Thank you.