Lecture 7:

An Introduction to CUDA/OpenCL and GPUs

Guest Lecture, UC Berkeley CS 267 Spring 2019
John Owens, UC Davis

“If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?”—Seymour Cray

From Bill Dally, 2015 talk …

Its not about the FLOPs

- DFMA 0.01mm^2 10pJ/OP – 2GFLOPs

16nm chip, 10mm on a side, 200W

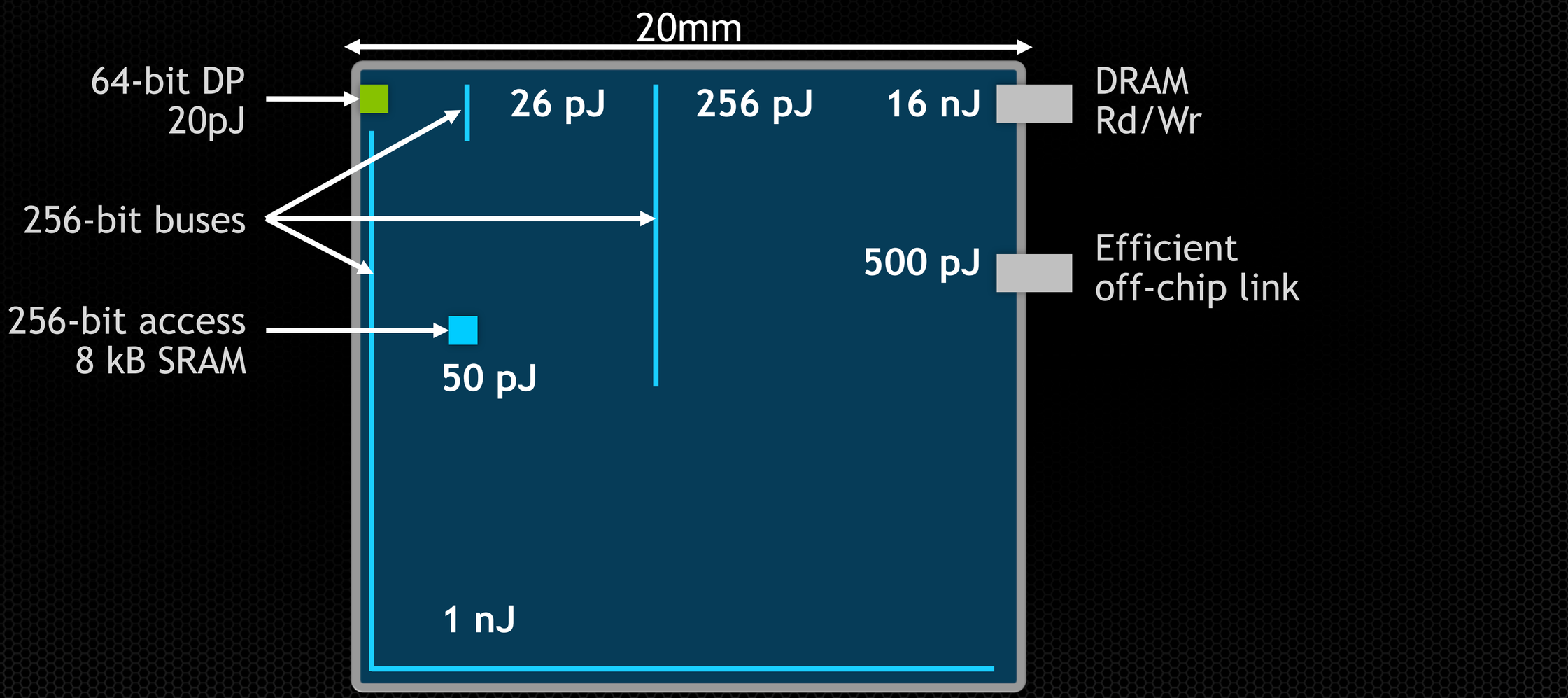
A chip with 10^4 FPUs:
100mm^2
200W
20TFLOPS

Pack 50,000 of these in racks
1EFLOPS
10MW

NVIDIA

From Bill Dally, 2015 talk …

Communication Dominates Arithmetic

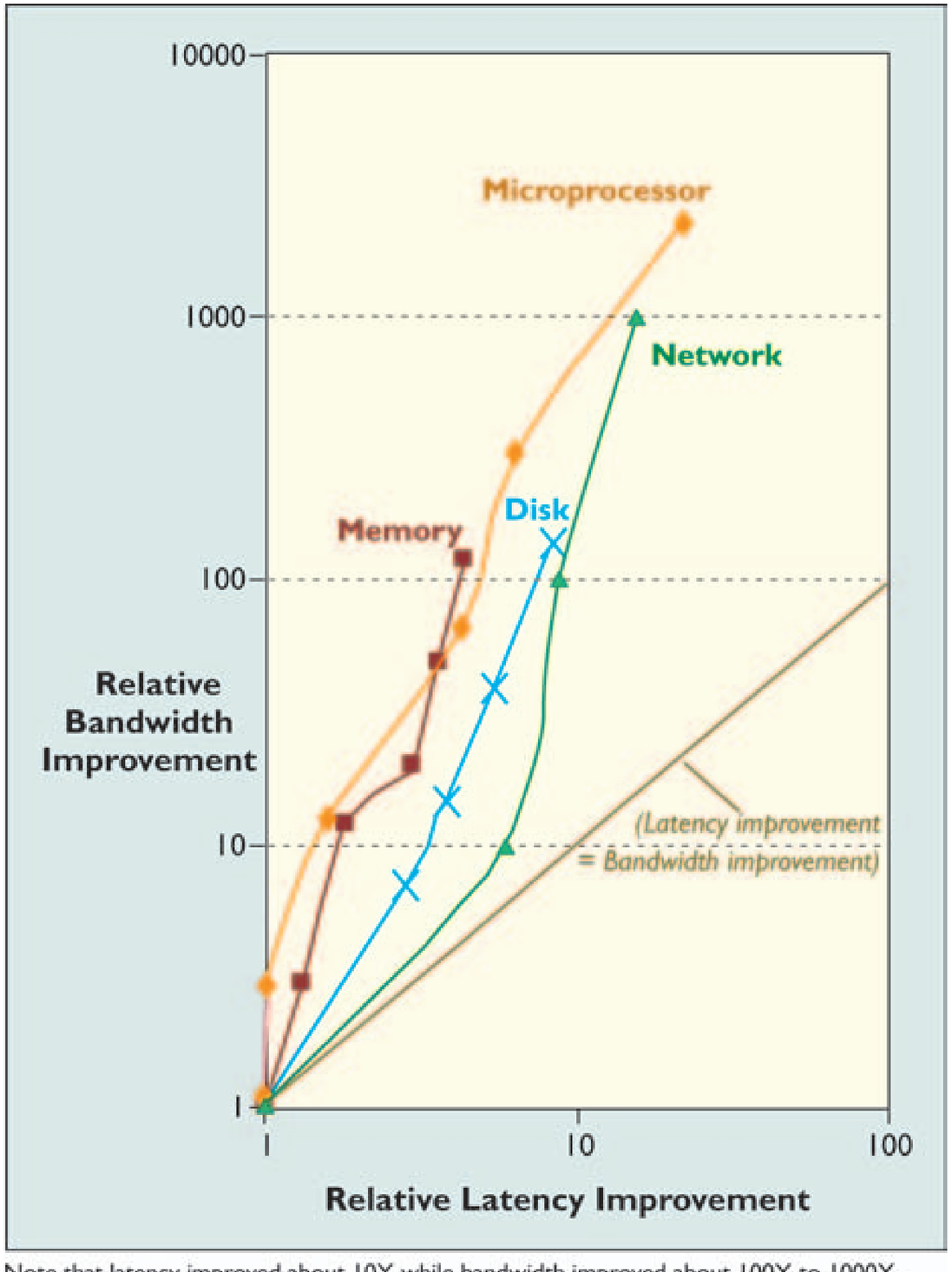


Challenges

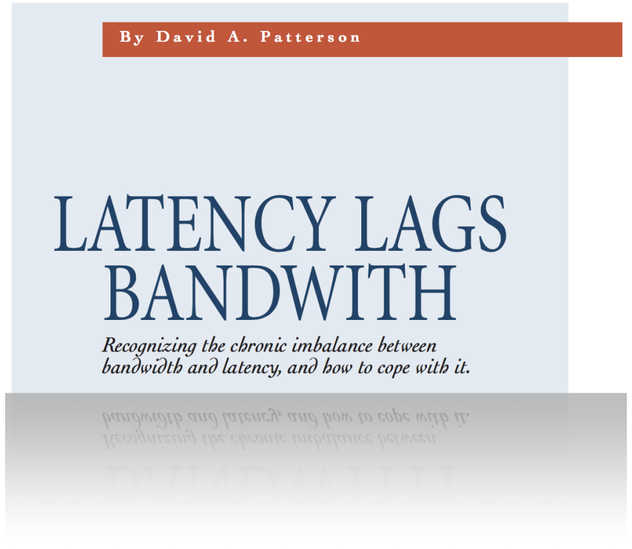
- *Not* how to pack enough arithmetic on a chip but instead …
  - How to do so in an energy-efficient way
    - *"Perf/W is Perf"—Bill Dally*
  - How to design a programming model that allows you to use that arithmetic (today's focus)
- Communication, power, and latency
  - Dominant power cost
  - Must exploit locality (for power and latency)
  - Other side of chip is many cycles away
  - Off-chip is hundreds of cycles away

Major idea today: GPUs are designed to maximize throughput as opposed to minimize latency

Latency Lags Bandwidth

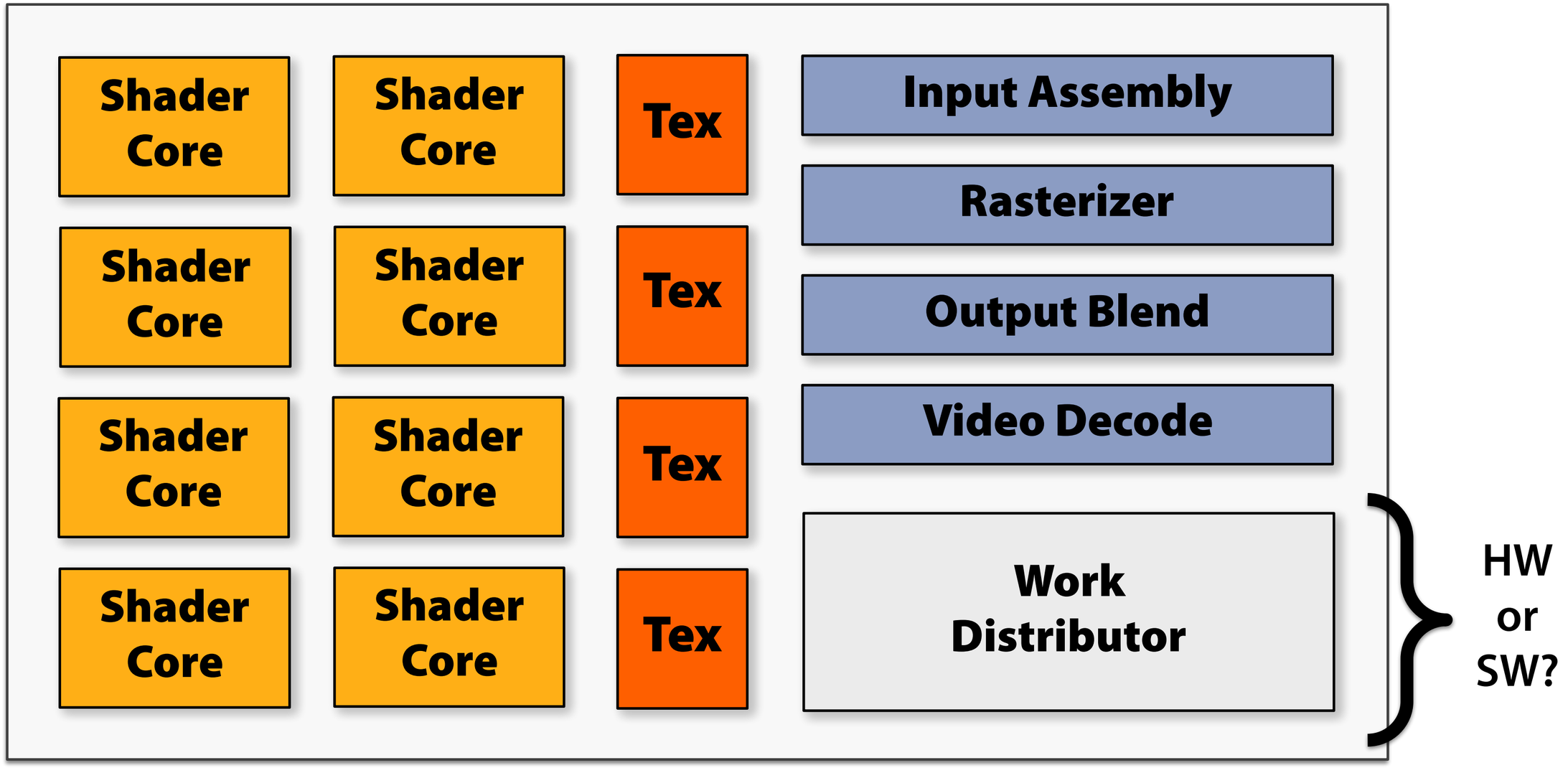


Note that latency improved about 10X while bandwidth improved about 100X to 1000X.



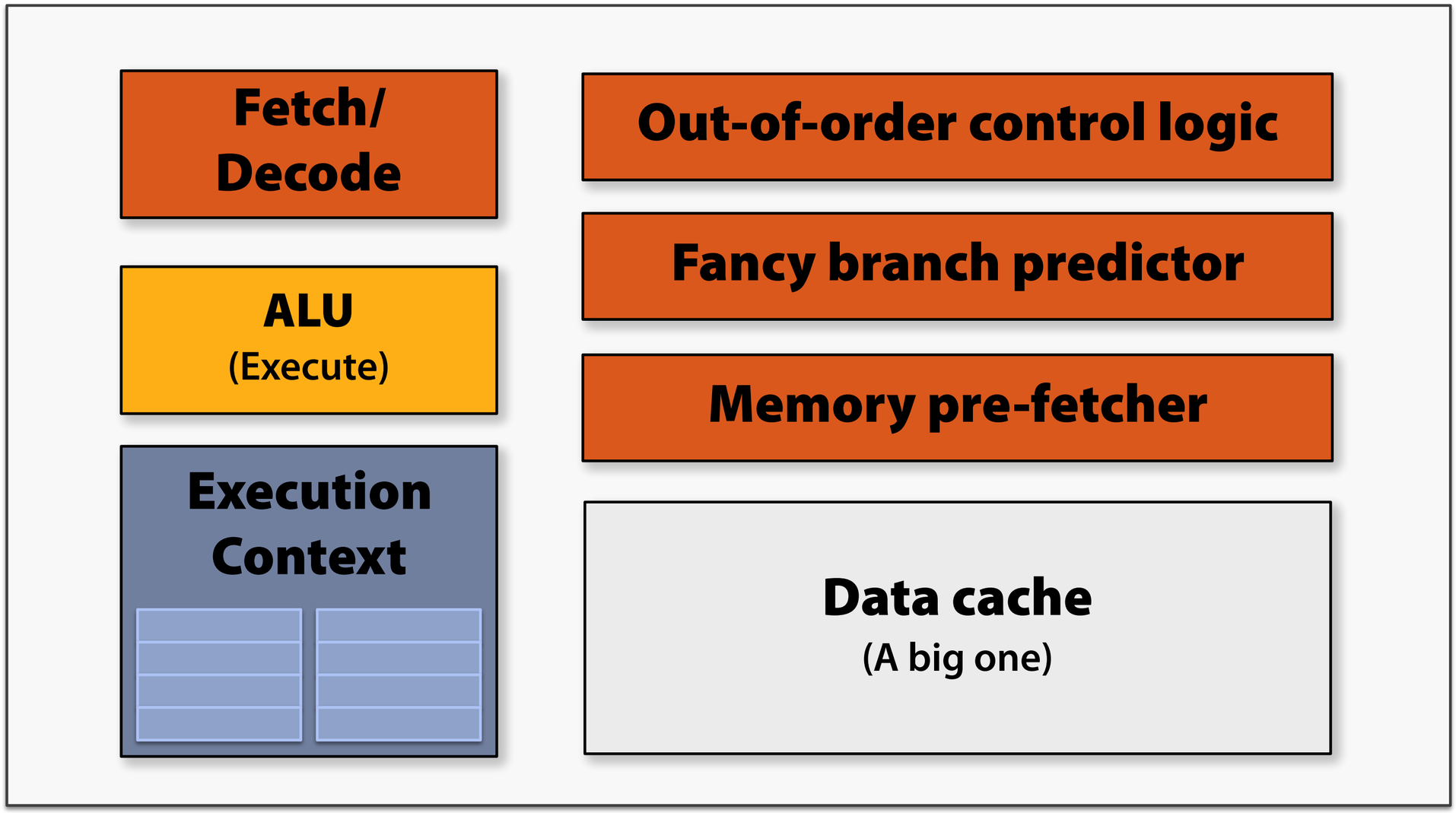
D. Patterson, CACM October 2004

What's in a GPU?

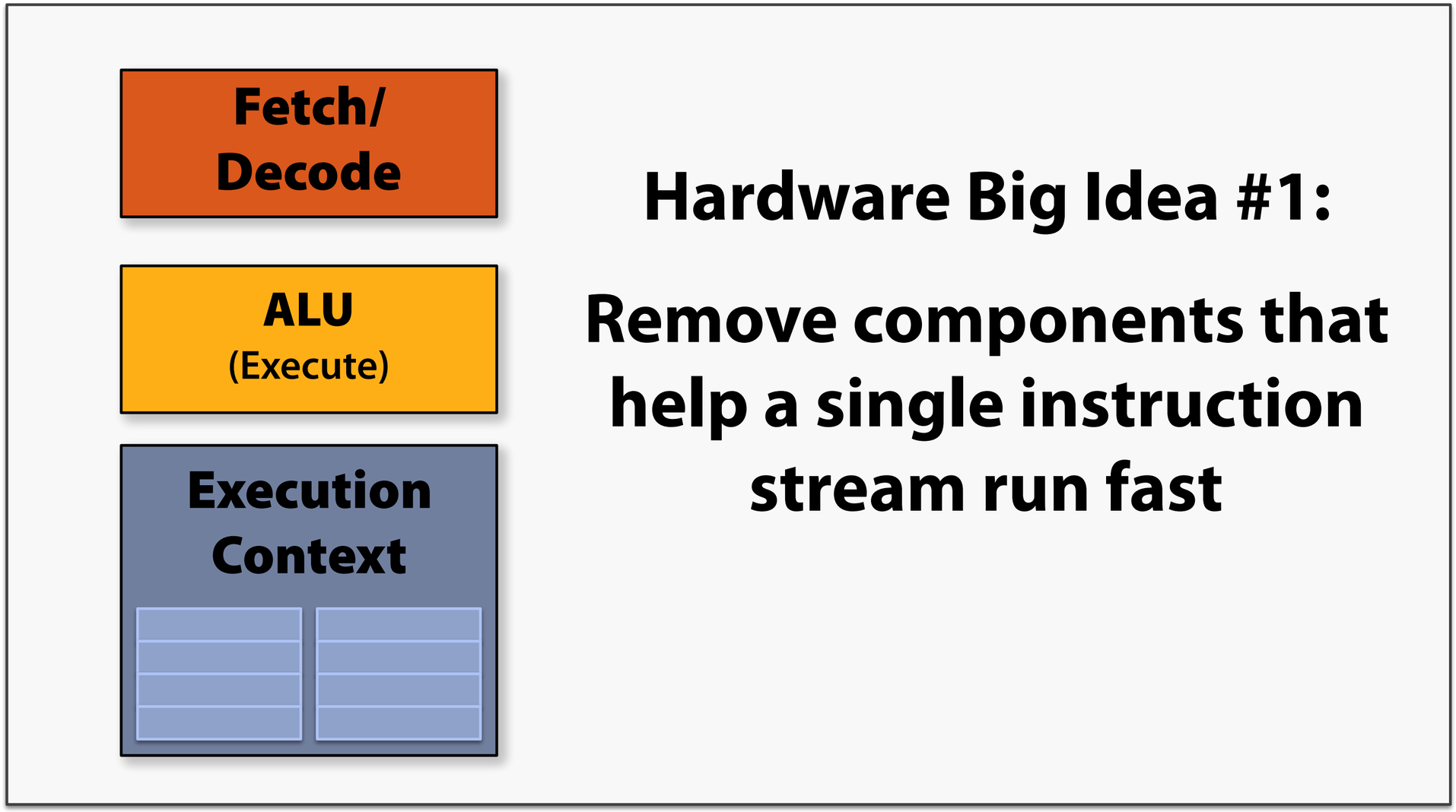


Heterogeneous chip multi-processor (highly tuned for graphics)

CPU-"style" cores



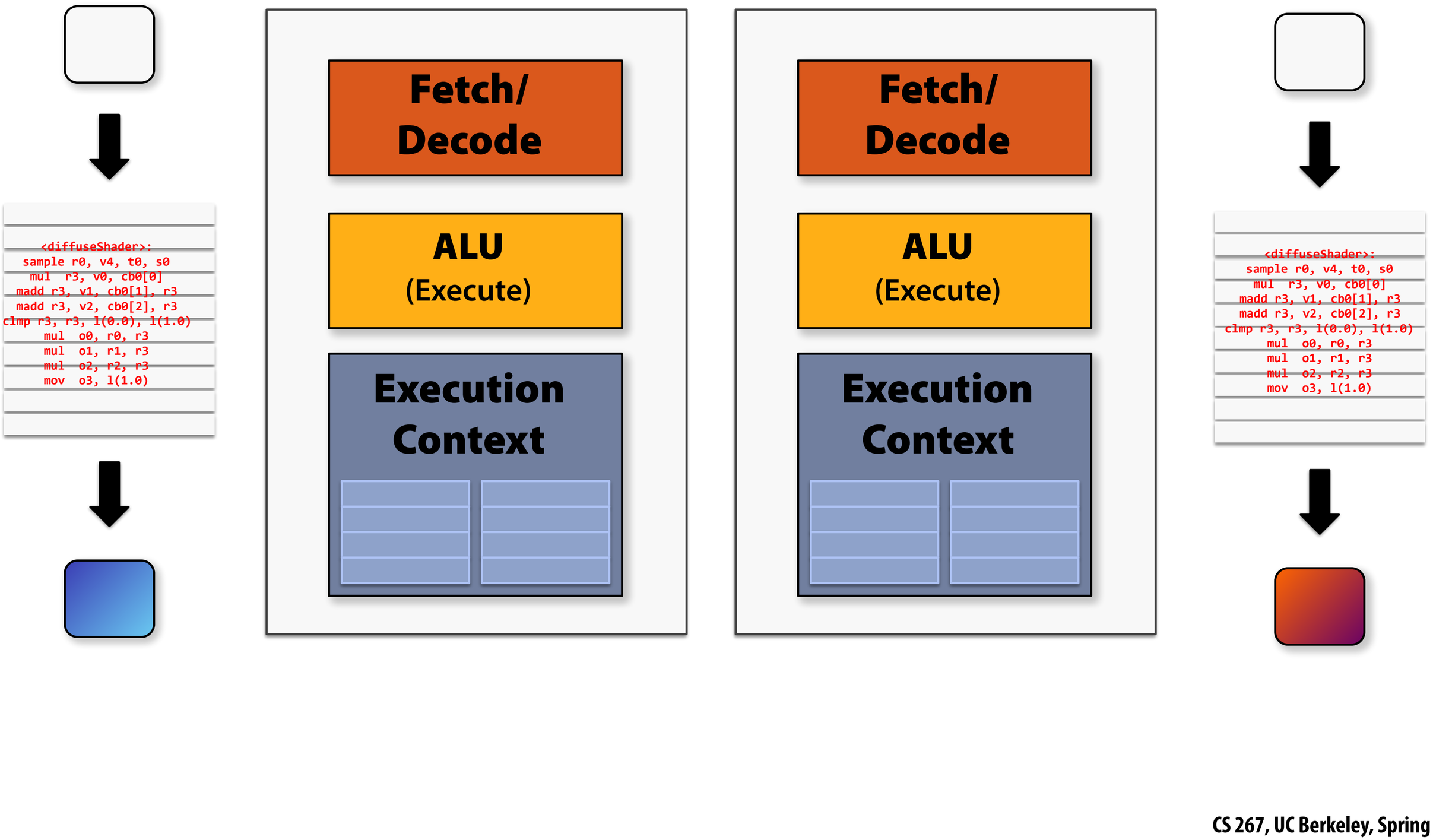
Slimming down



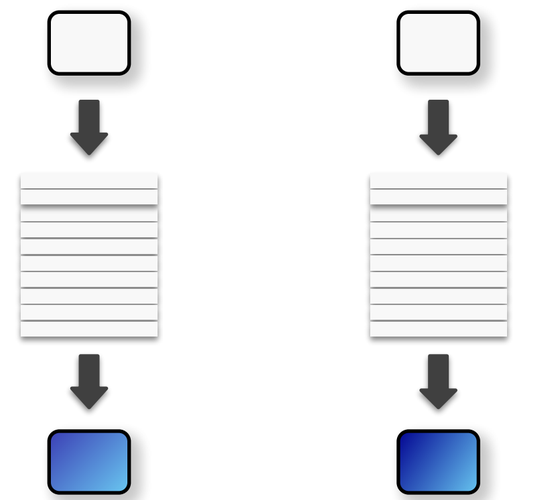
Two cores (two threads in parallel)

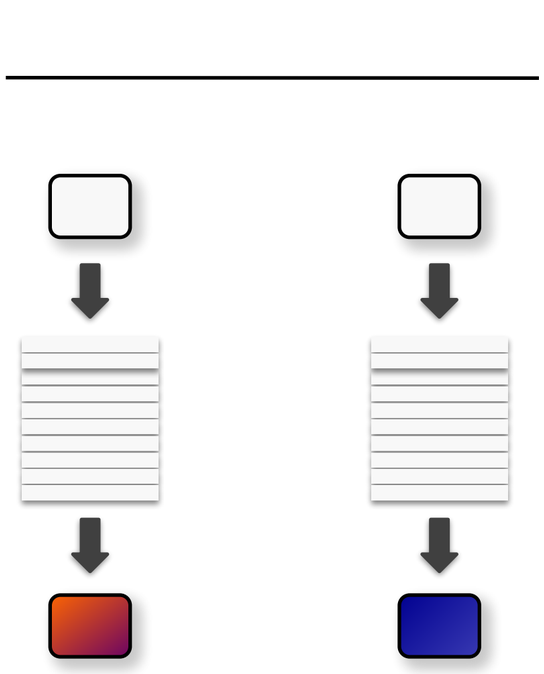
thread 1

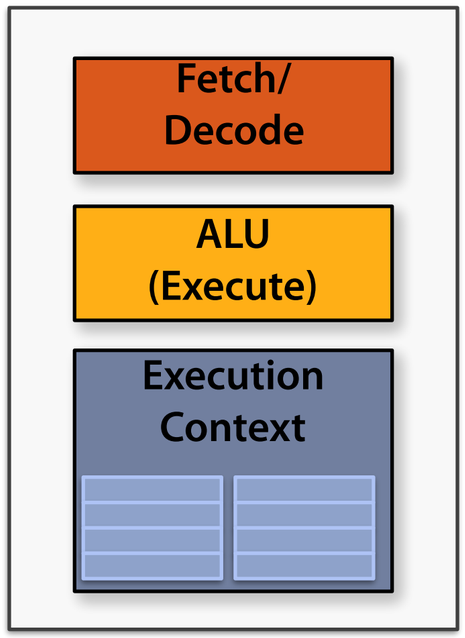
thread 2

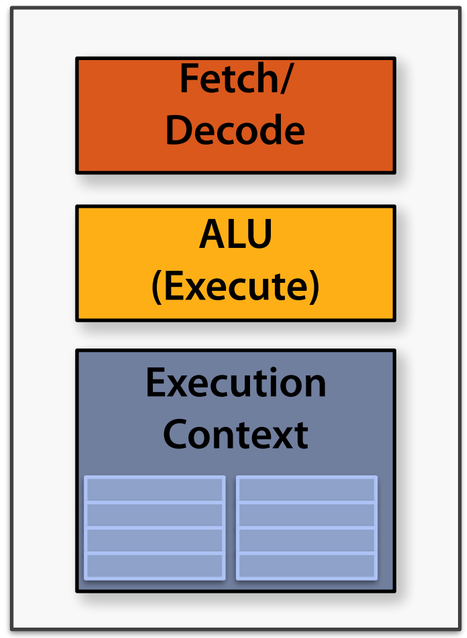


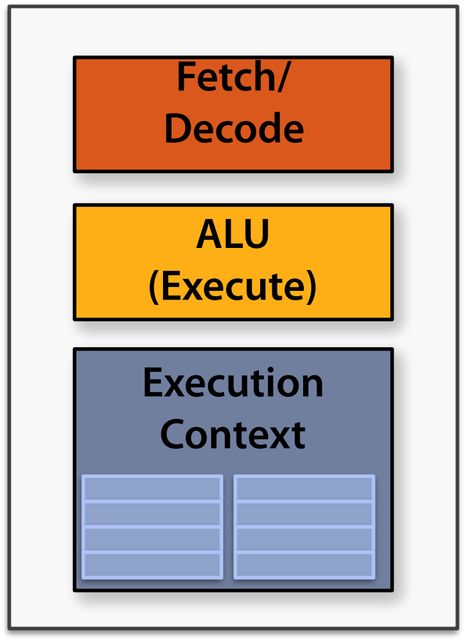
Four cores (four threads in parallel)

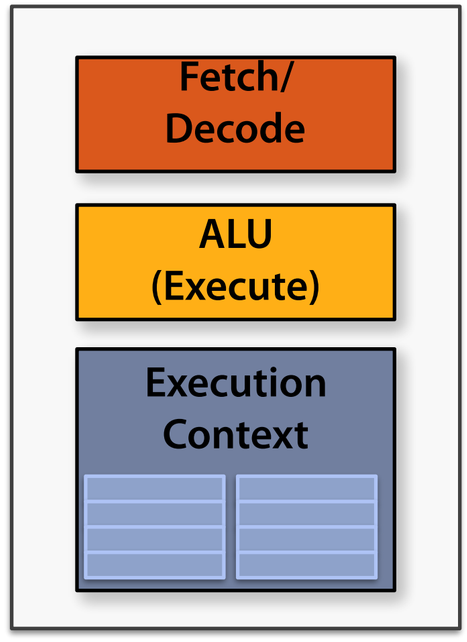




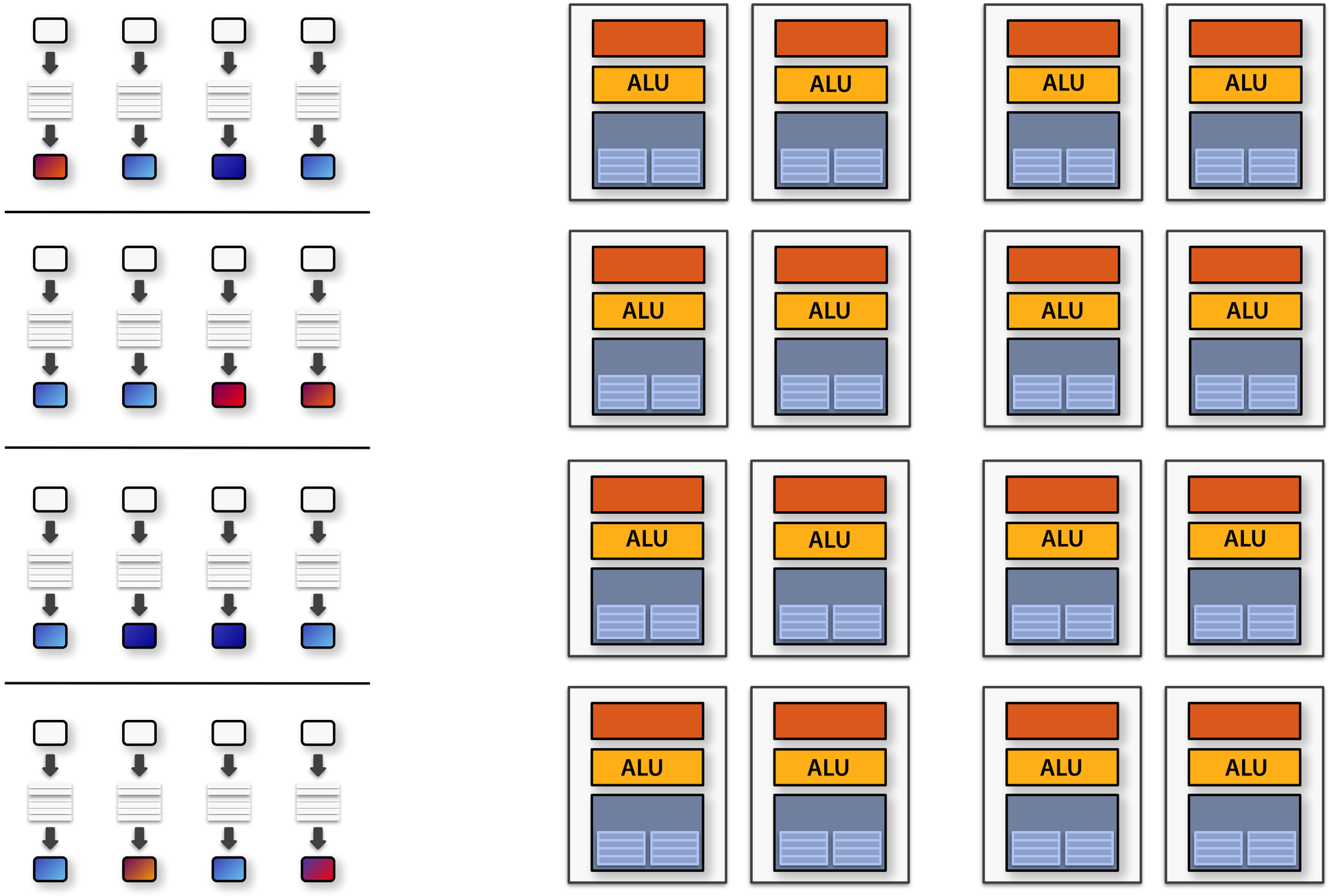






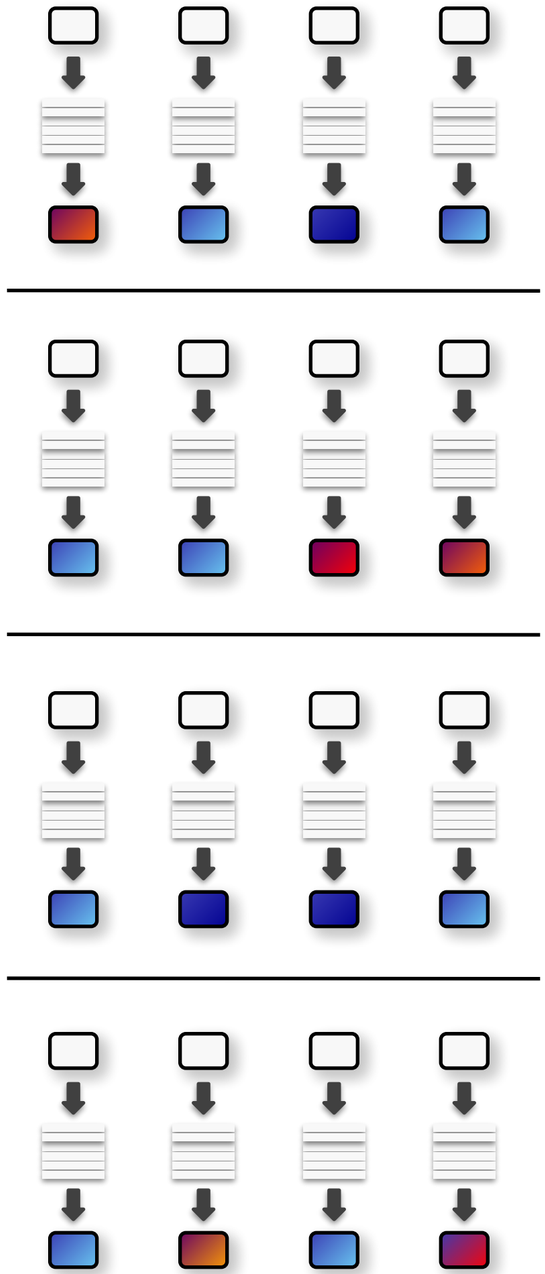


Sixteen cores (sixteen theads in parallel)



16 cores = 16 simultaneous instruction streams

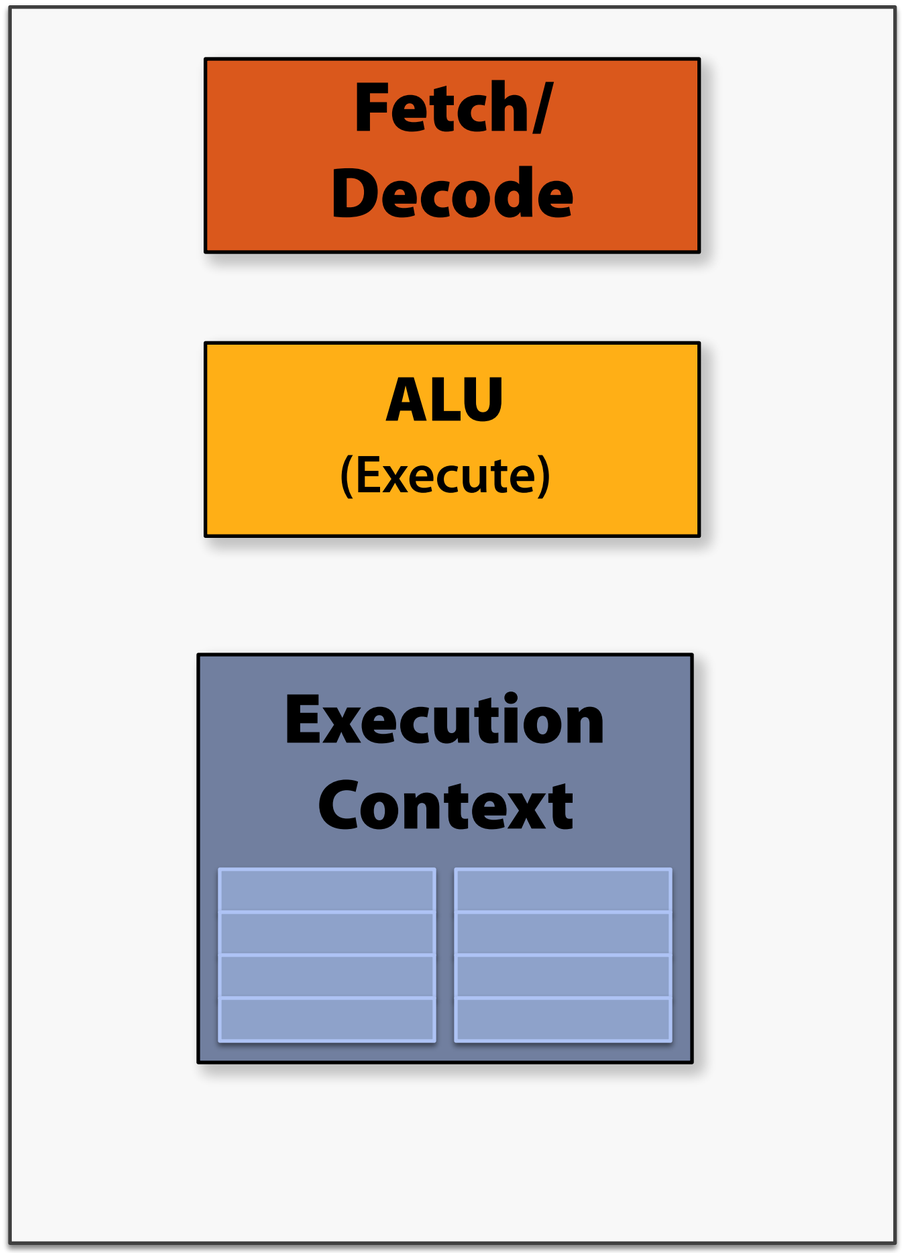
Instruction stream sharing



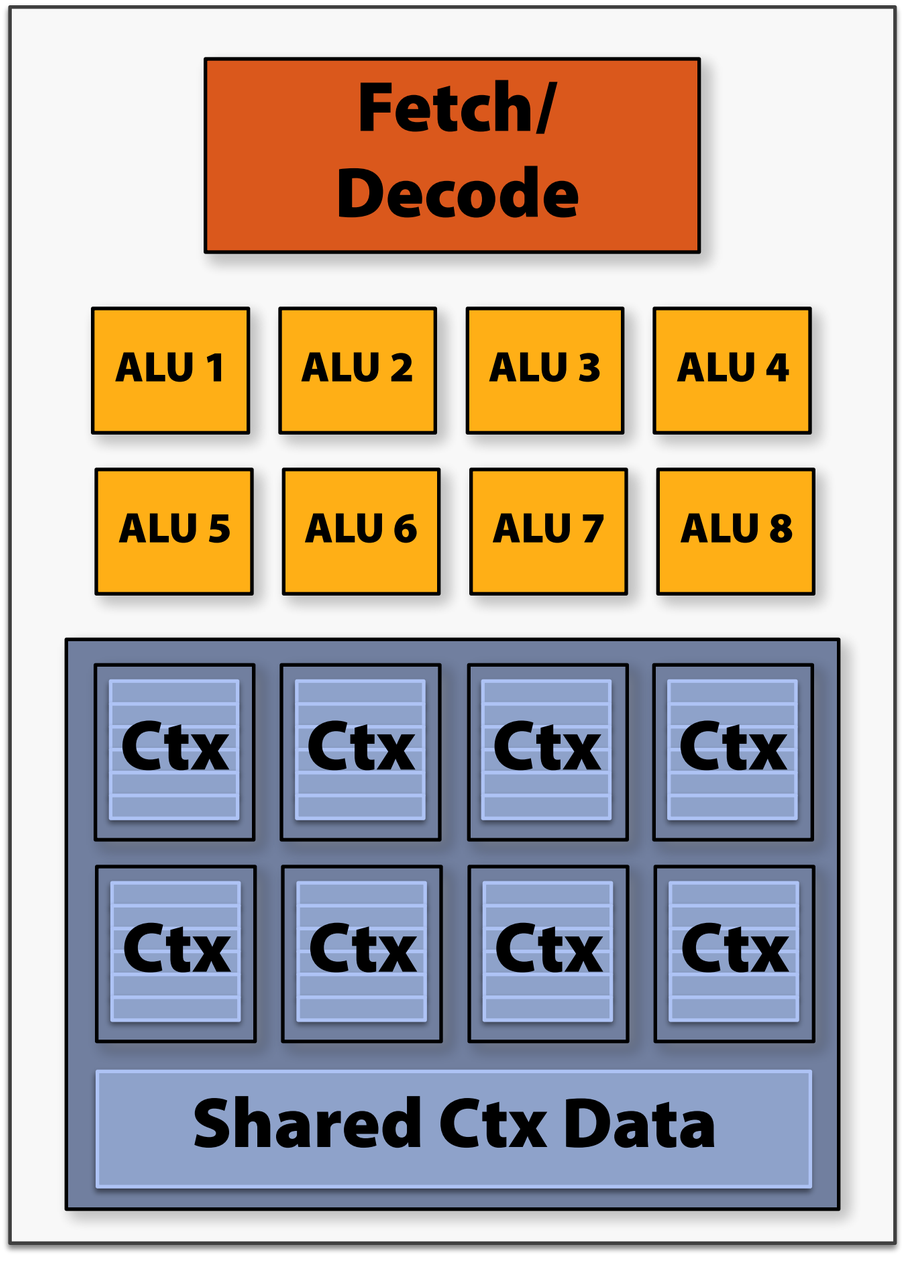
But… many threads *should* be able to share an instruction stream!

```
<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



Hardware Big Idea #2:

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

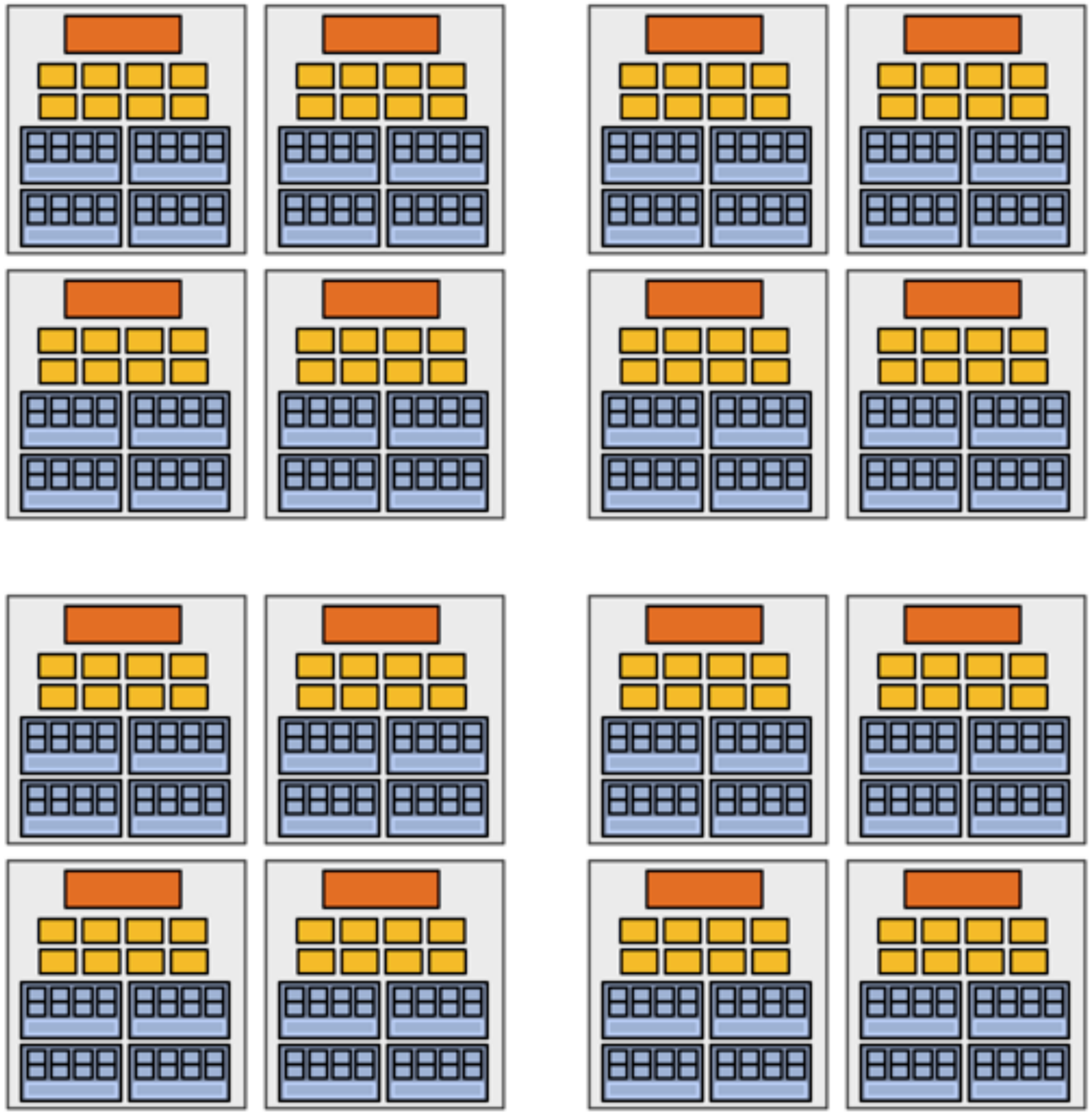
SIMD processing

We would call this “8-wide” SIMD.
NVIDIA would say this is a “warp” of size 8.

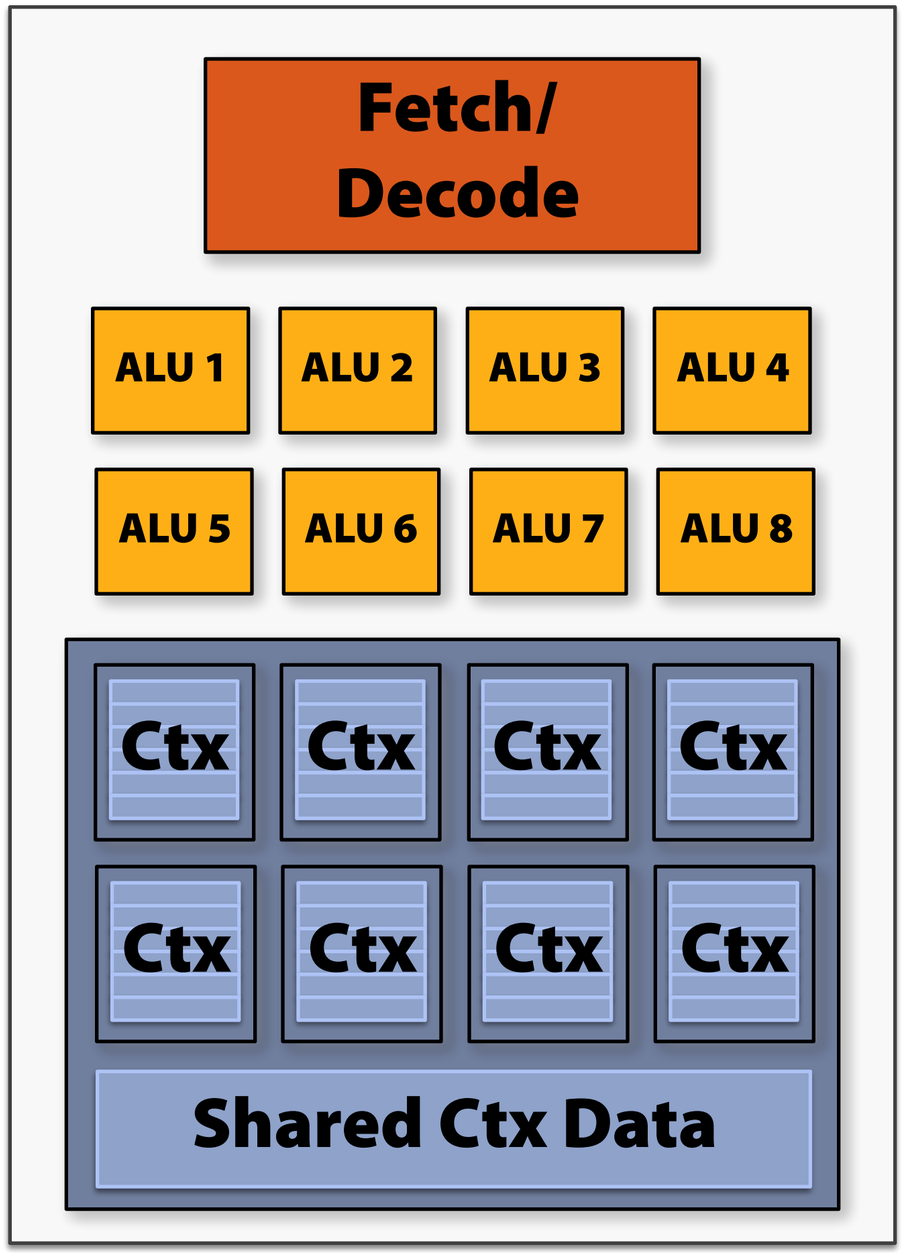
Now we have two ways to exploit parallelism

- 1: SIMD within a core
- 2: Multiple cores

- This chip has:
  - 16 cores
  - 8 MAD ALUs/core
  - 16 simultaneous instruction streams
  - 64 "resident" (but interleaved) instruction streams
  - 512 resident threads
  - 256 GFLOPS @ 1 GHz



Do we have to modify the program?



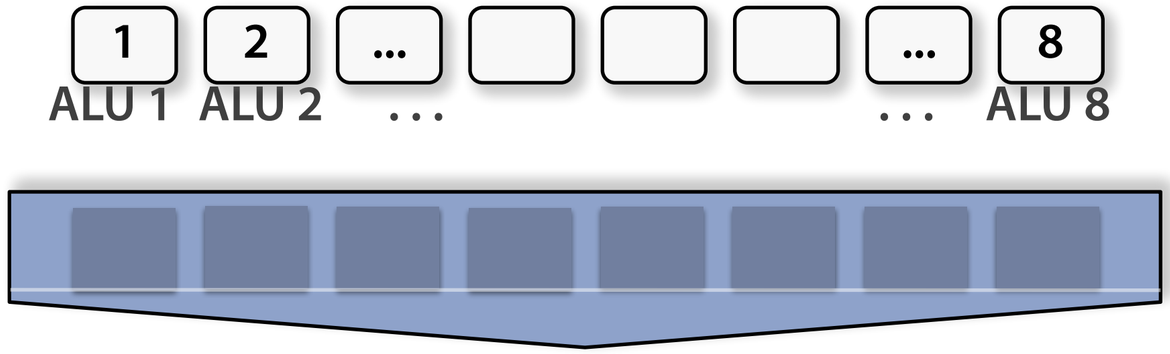

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

No! The programmer writes a "scalar" program and the hardware+driver map that scalar program to multiple SIMD lanes.

*—This is also the idea behind Intel's ISPC ("Intel SPMD Program Compiler")

But what about branches?

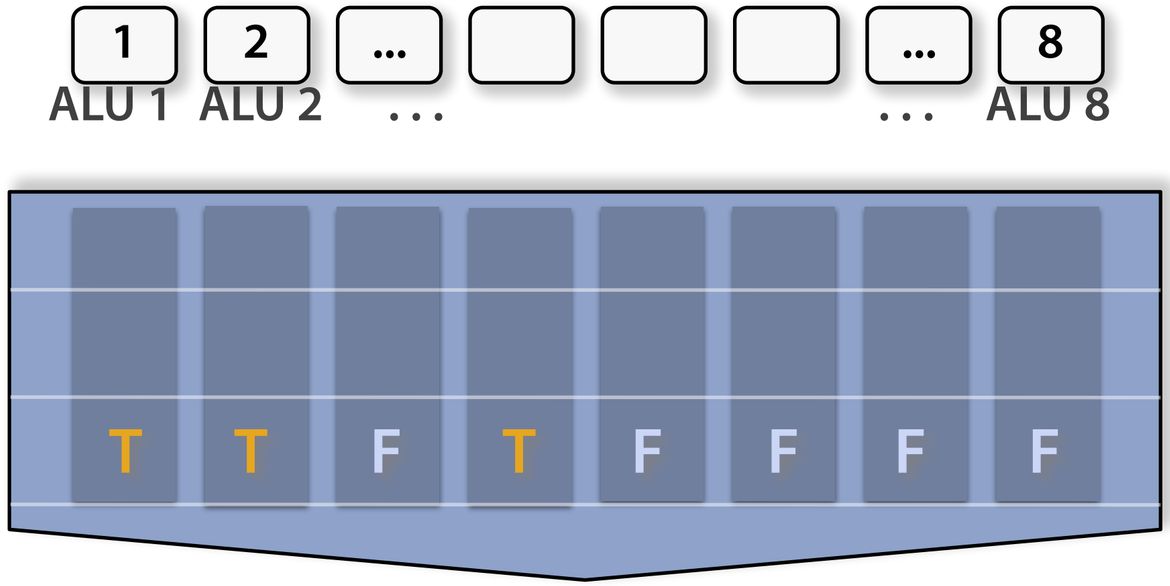
Time (clocks)




```
  <unconditional
   program code>

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

     <resume
   unconditional
   program code>
```

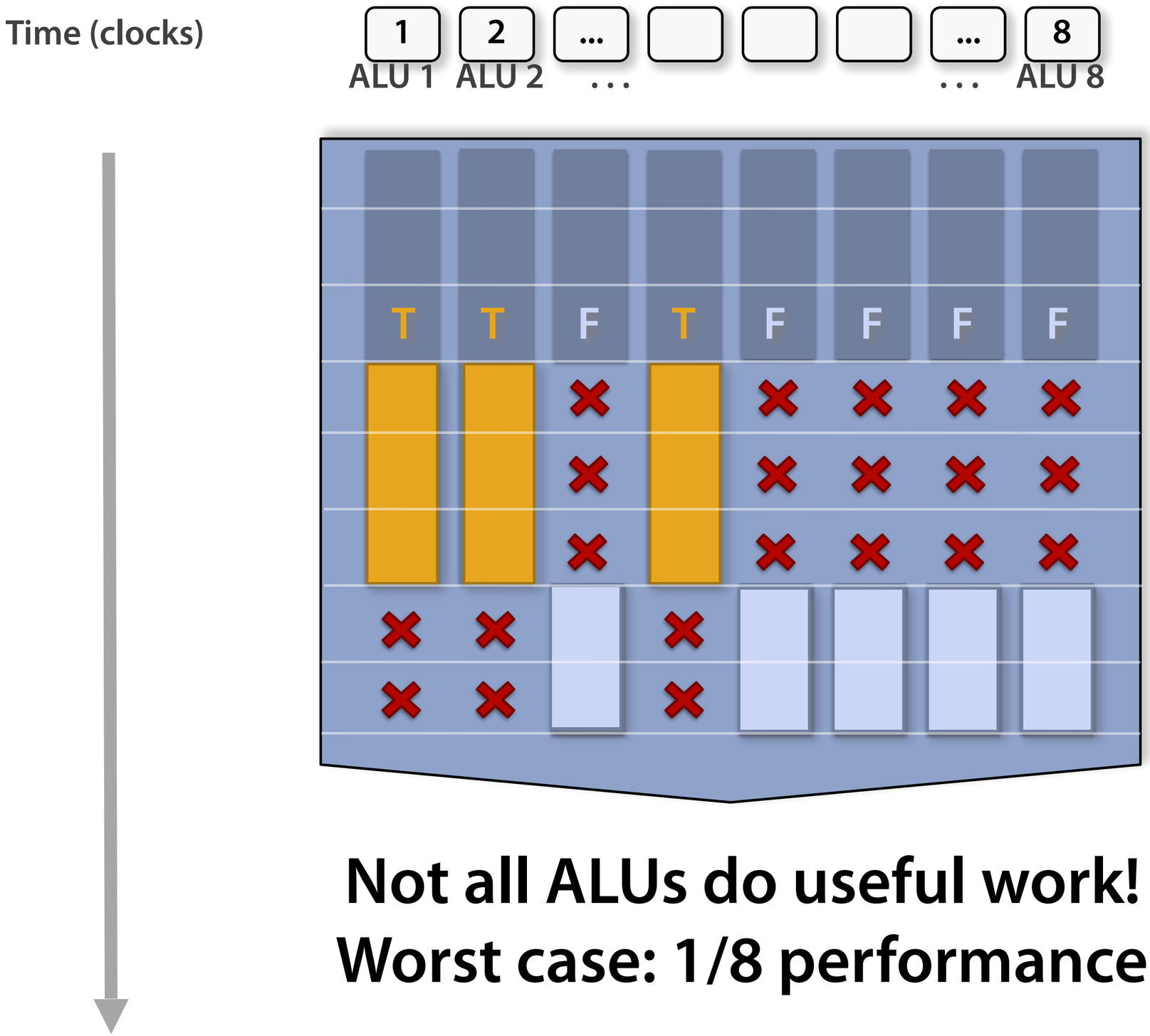
But what about branches?

Time (clocks)




```
<unconditional
 program code>

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

    <resume
  unconditional
  program code>
```

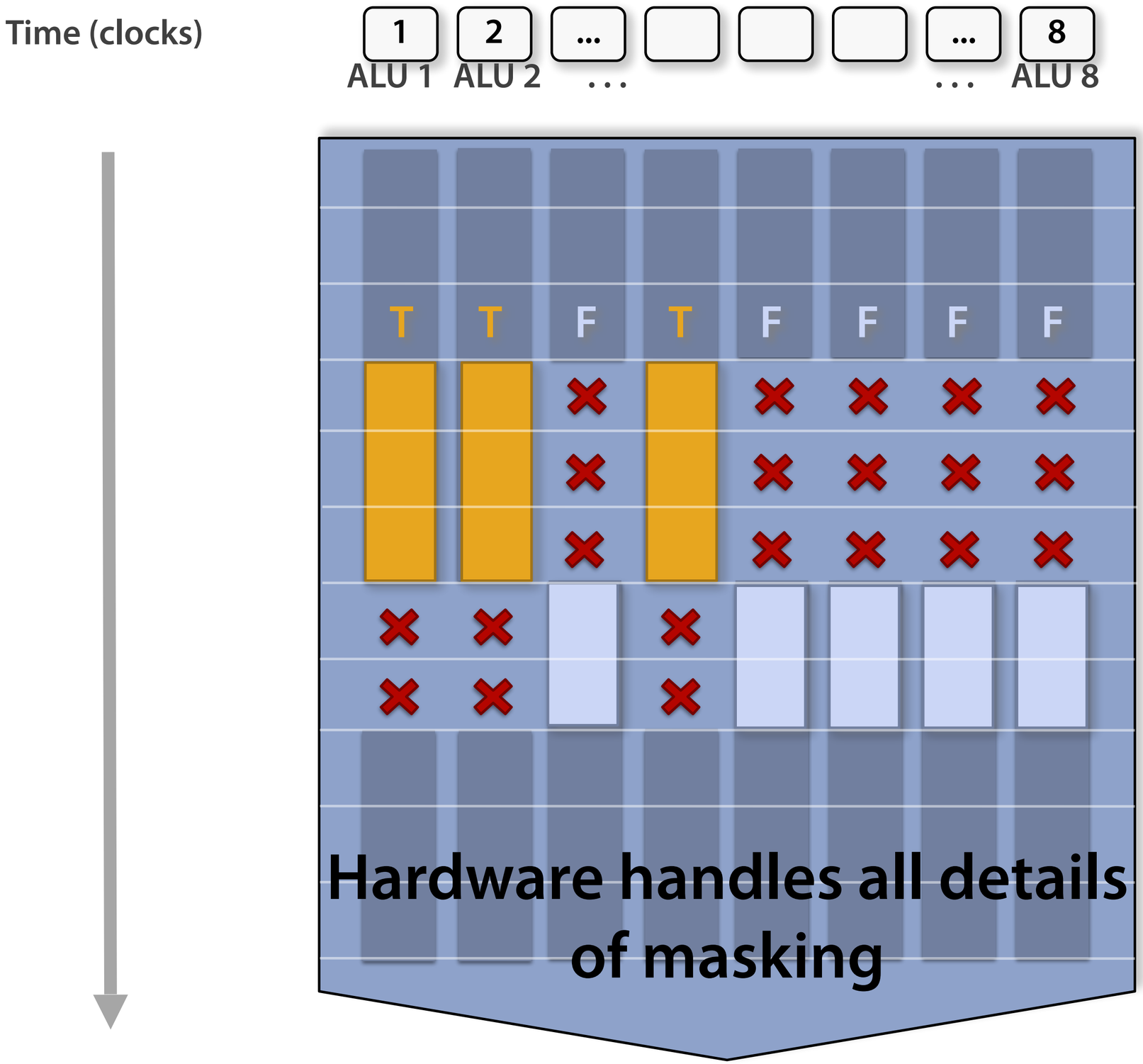
But what about branches?




```
<unconditional
 program code>

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

<resume
 unconditional
 program code>
```

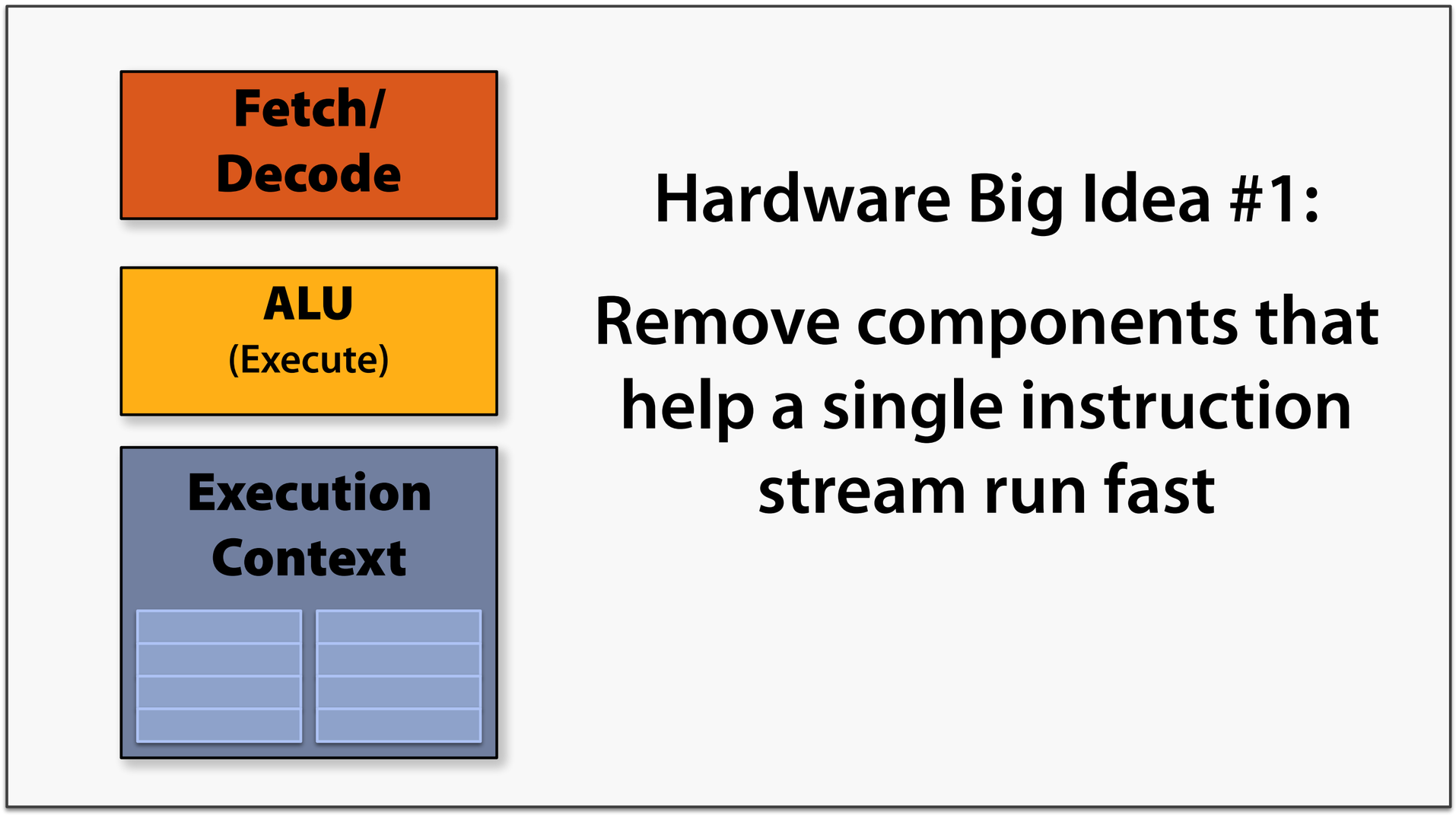
Not all ALUs do useful work!
Worst case: 1/8 performance

But what about branches?

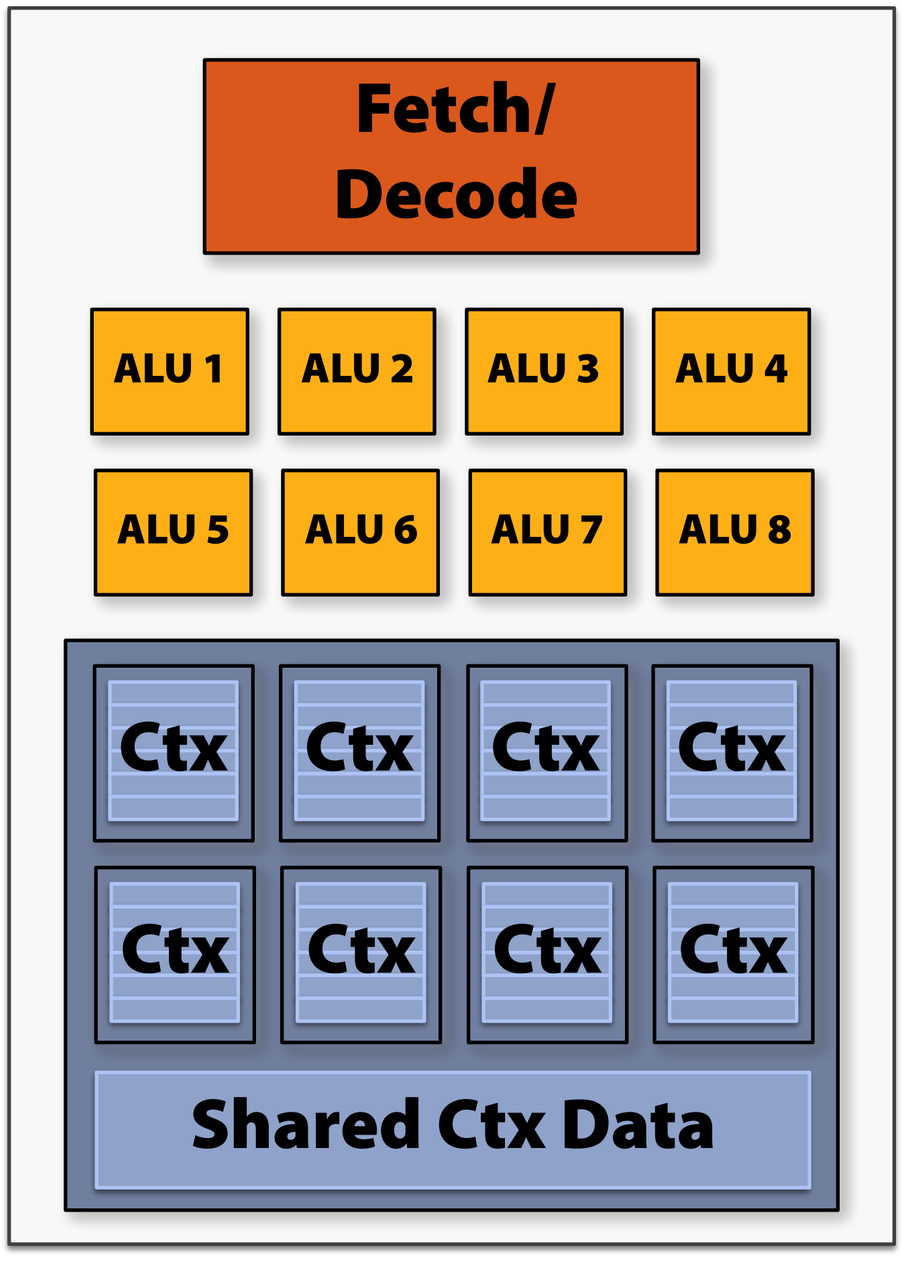



```
<unconditional
 program code>

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

   <resume
  unconditional
  program code>
```

Slimming down



Add ALUs



Hardware Big Idea #2:

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

SIMD processing

Stalls!

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

Memory access latency = 100s to 1000s of cycles

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

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

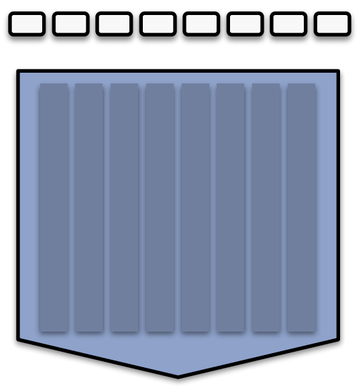
Hardware Big Idea #3:

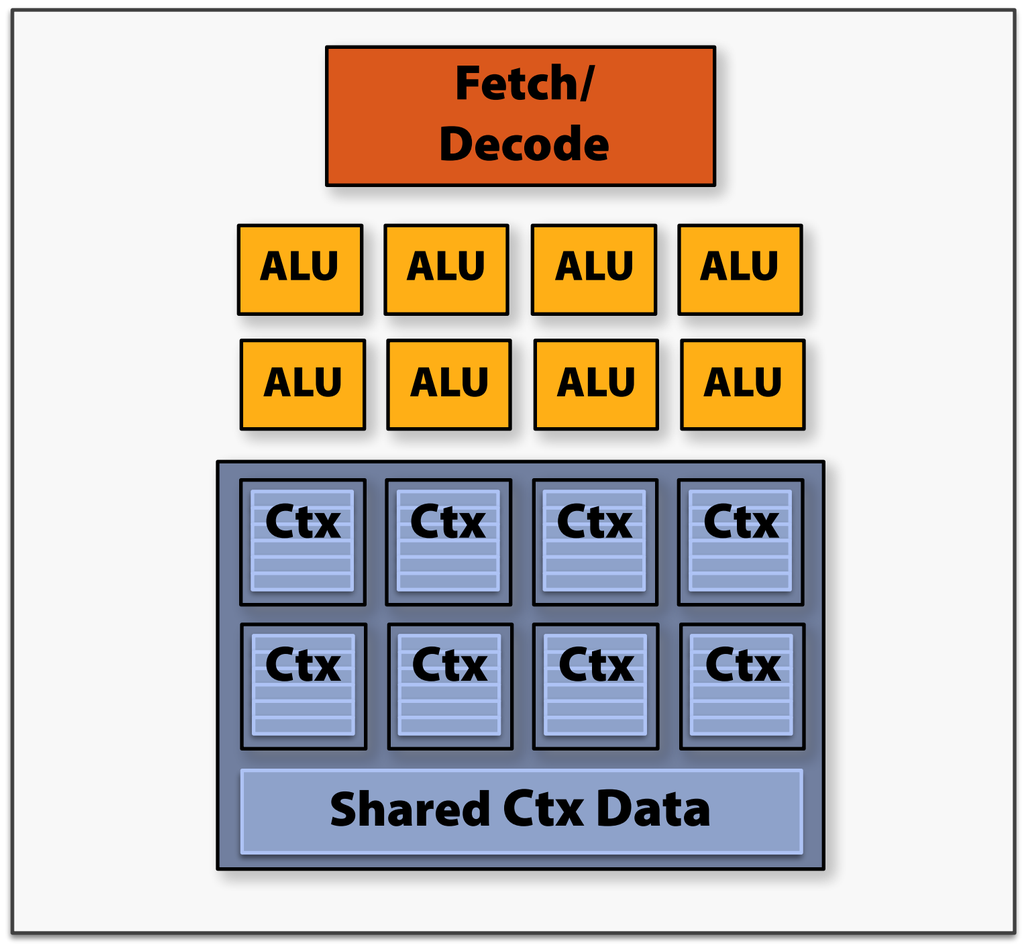
Interleave processing of many threads on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

Time (clocks)

Threads 1 … 8





Hiding shader stalls

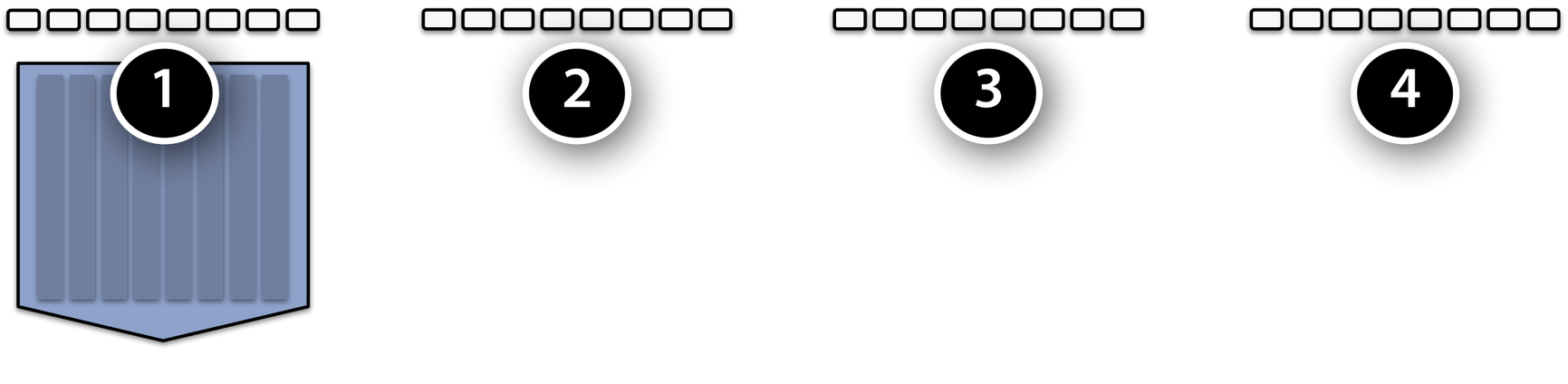
Time (clocks)

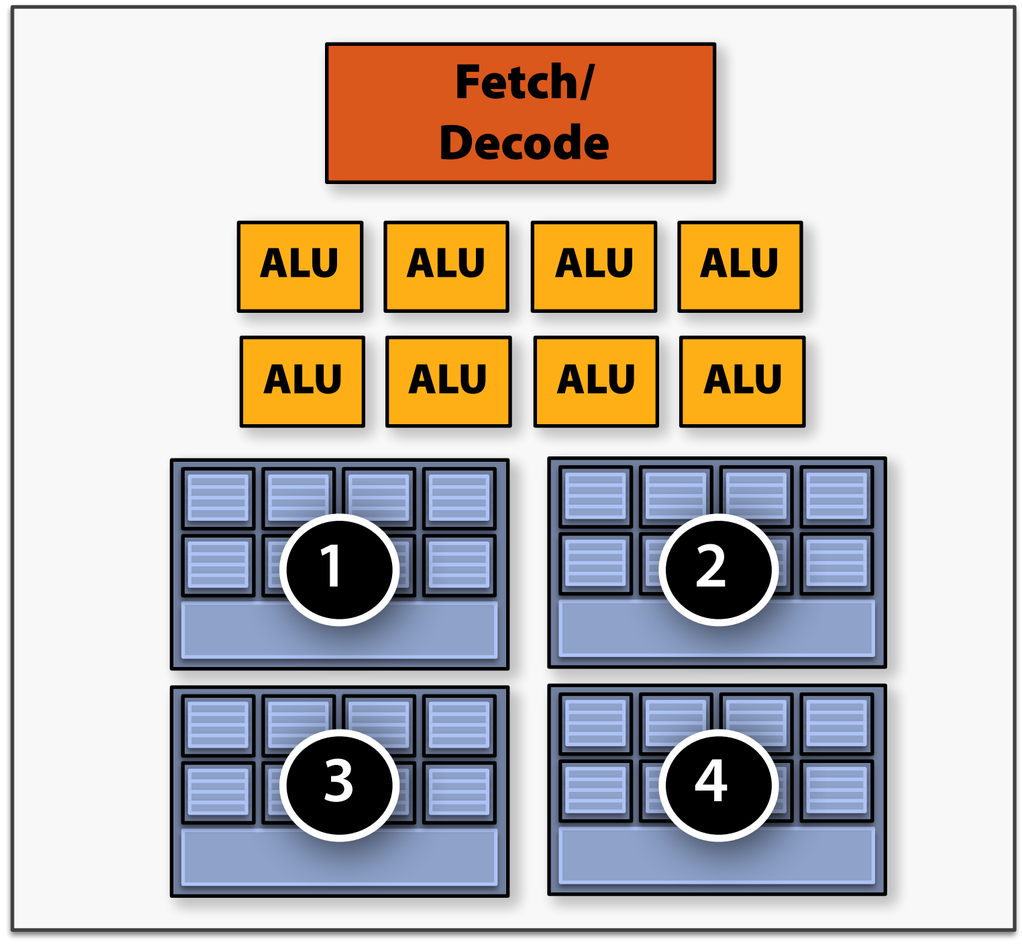
Threads 1 … 8

Threads 9… 16

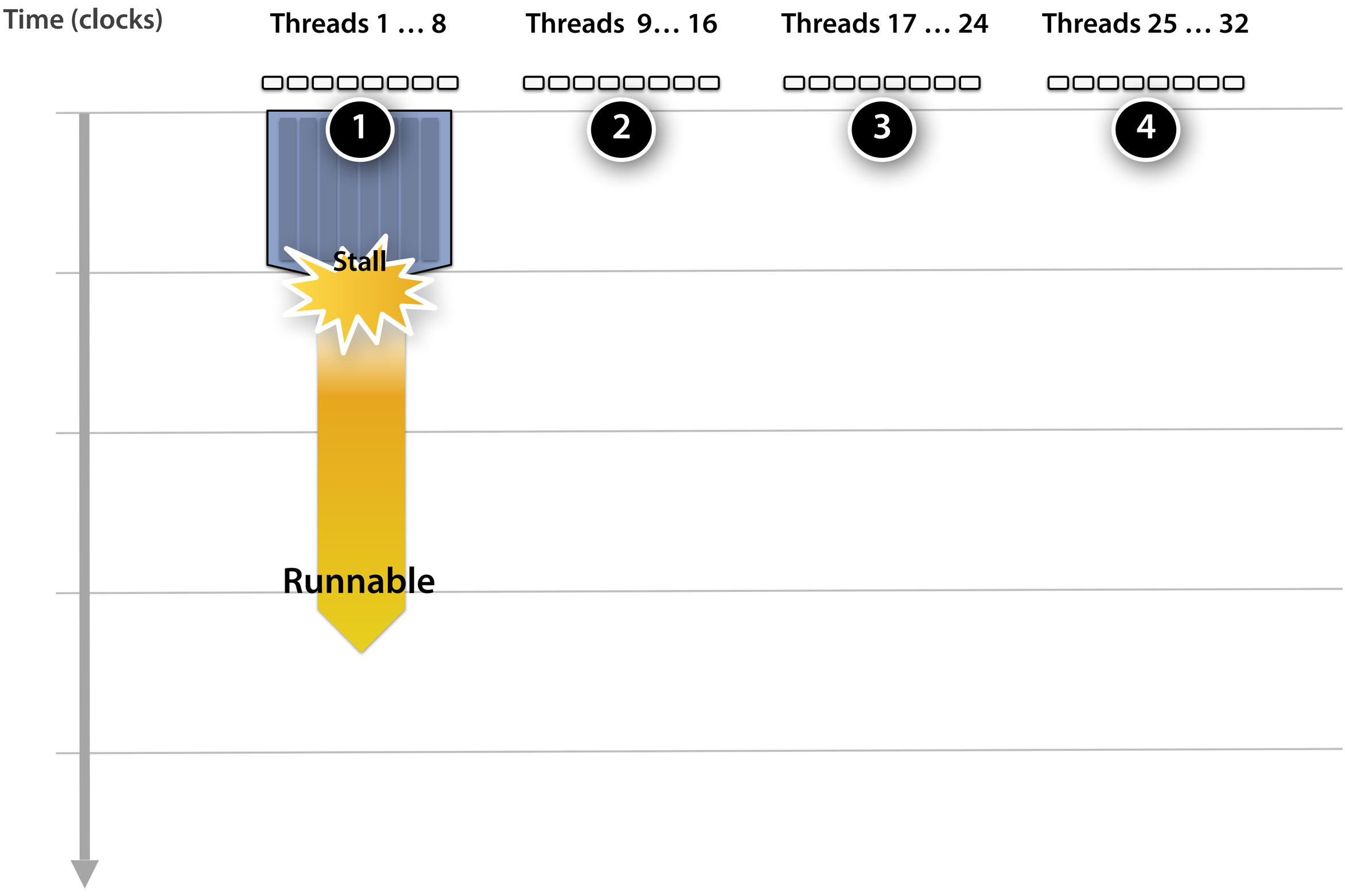
Threads 17 … 24

Threads 25 … 32

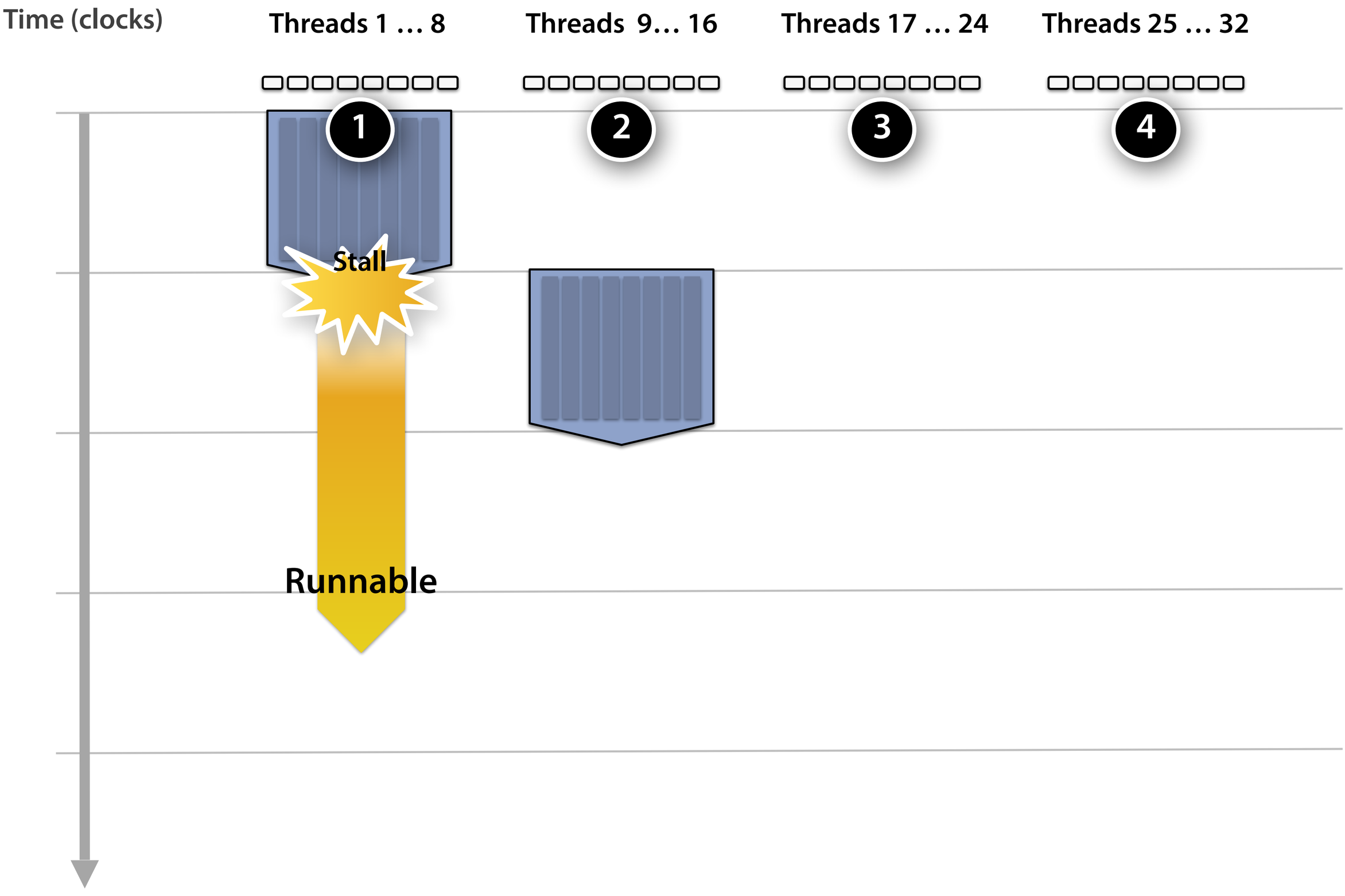




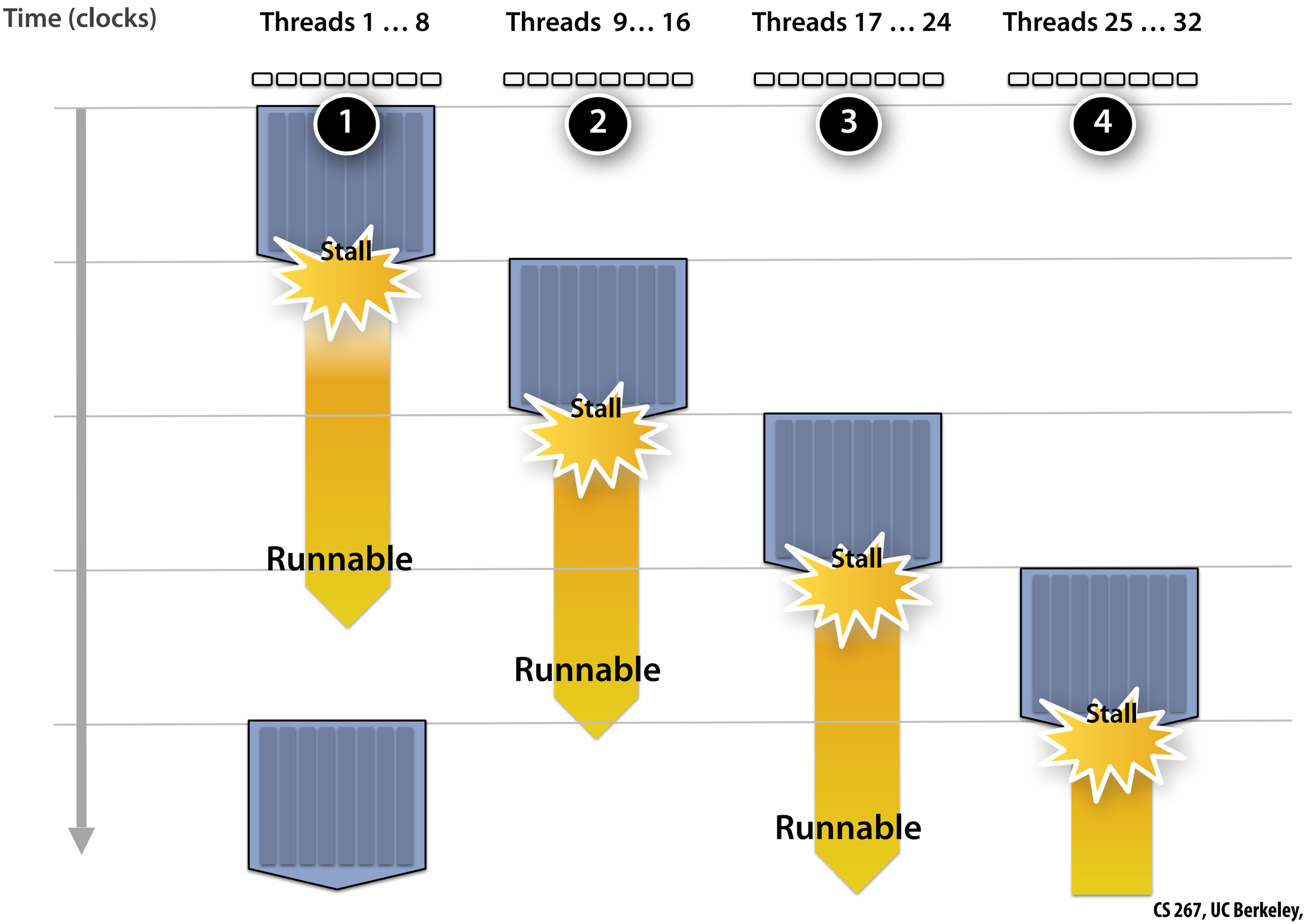
Hiding shader stalls



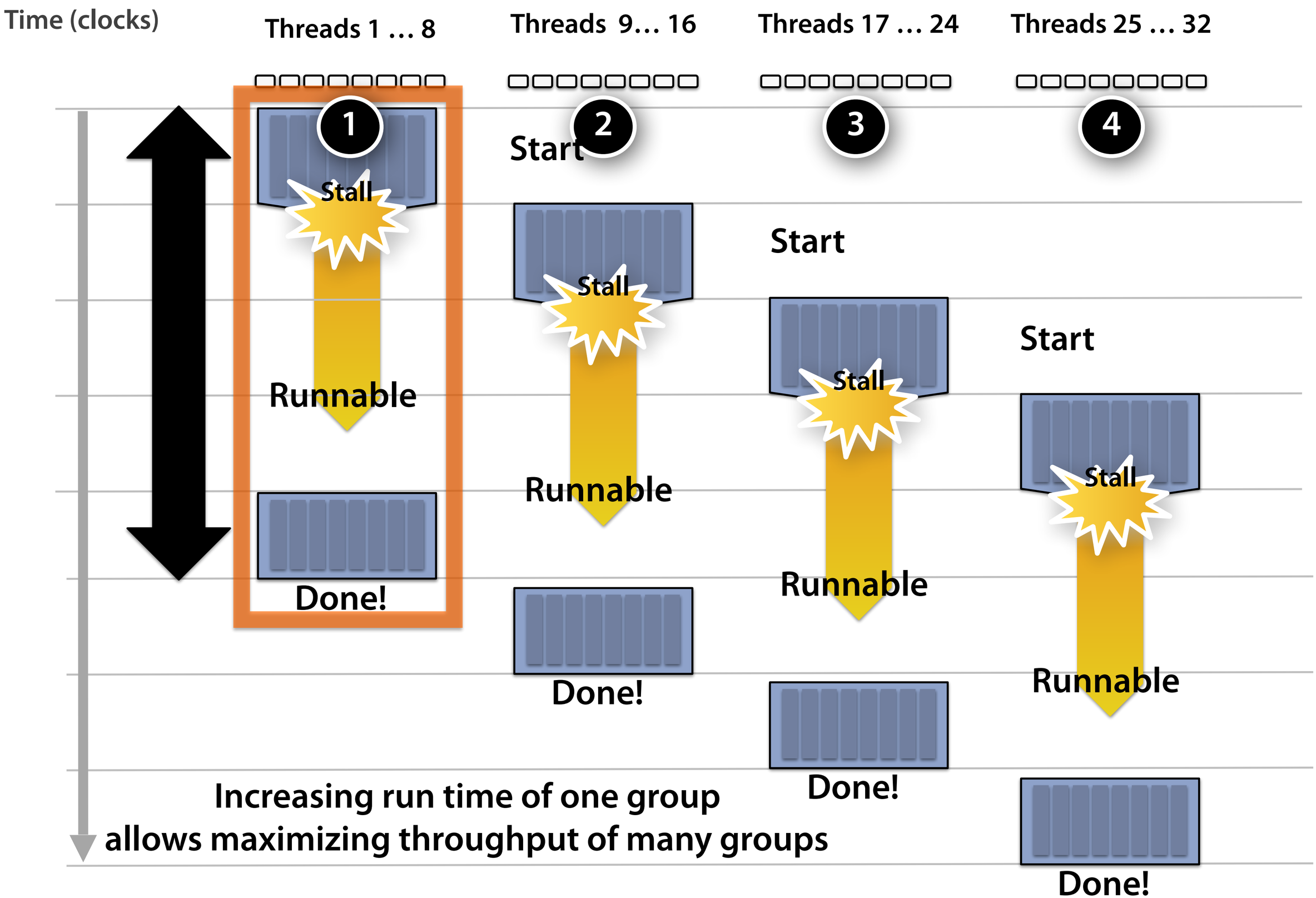
Hiding shader stalls



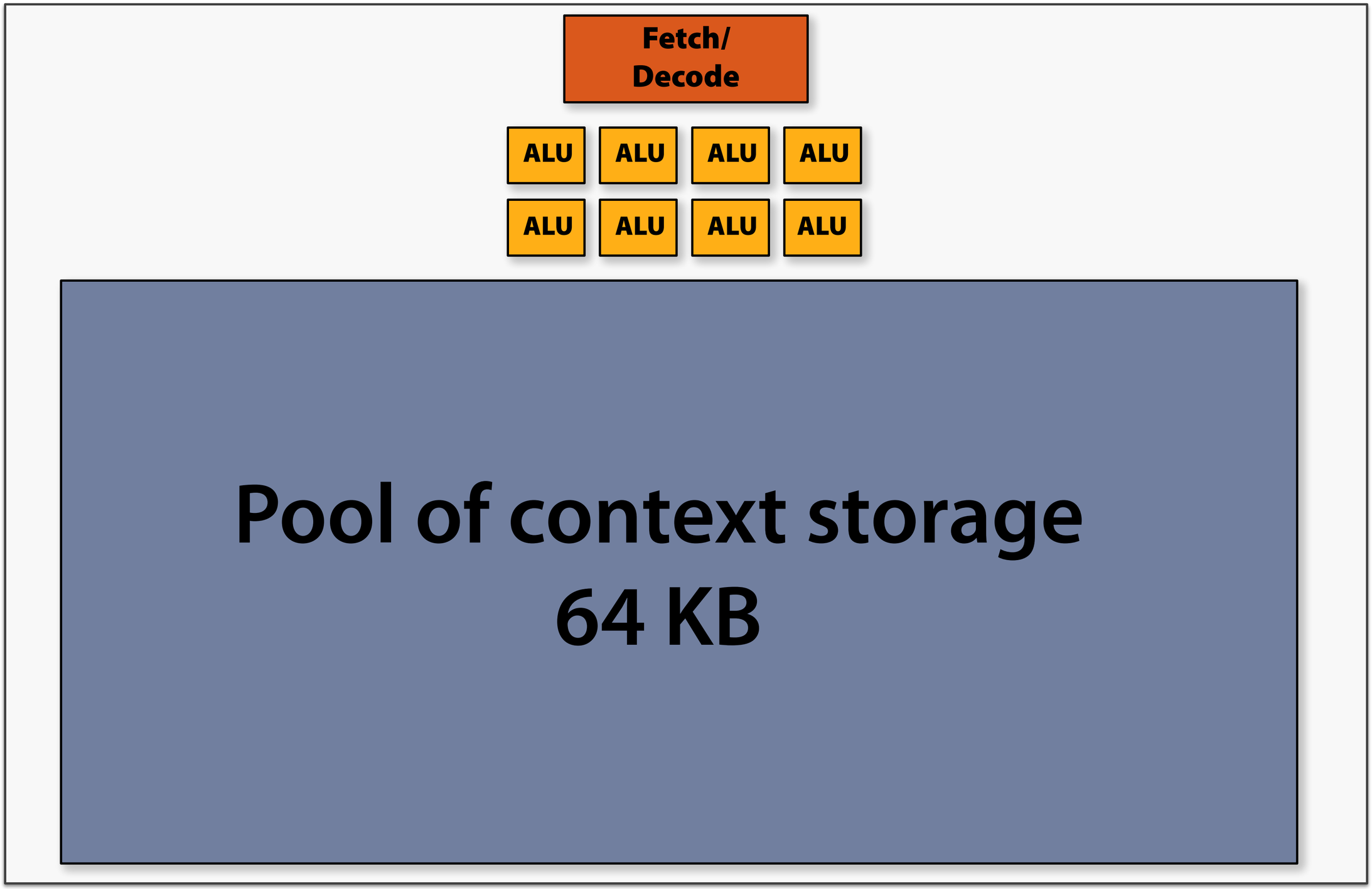
Hiding shader stalls



Throughput!

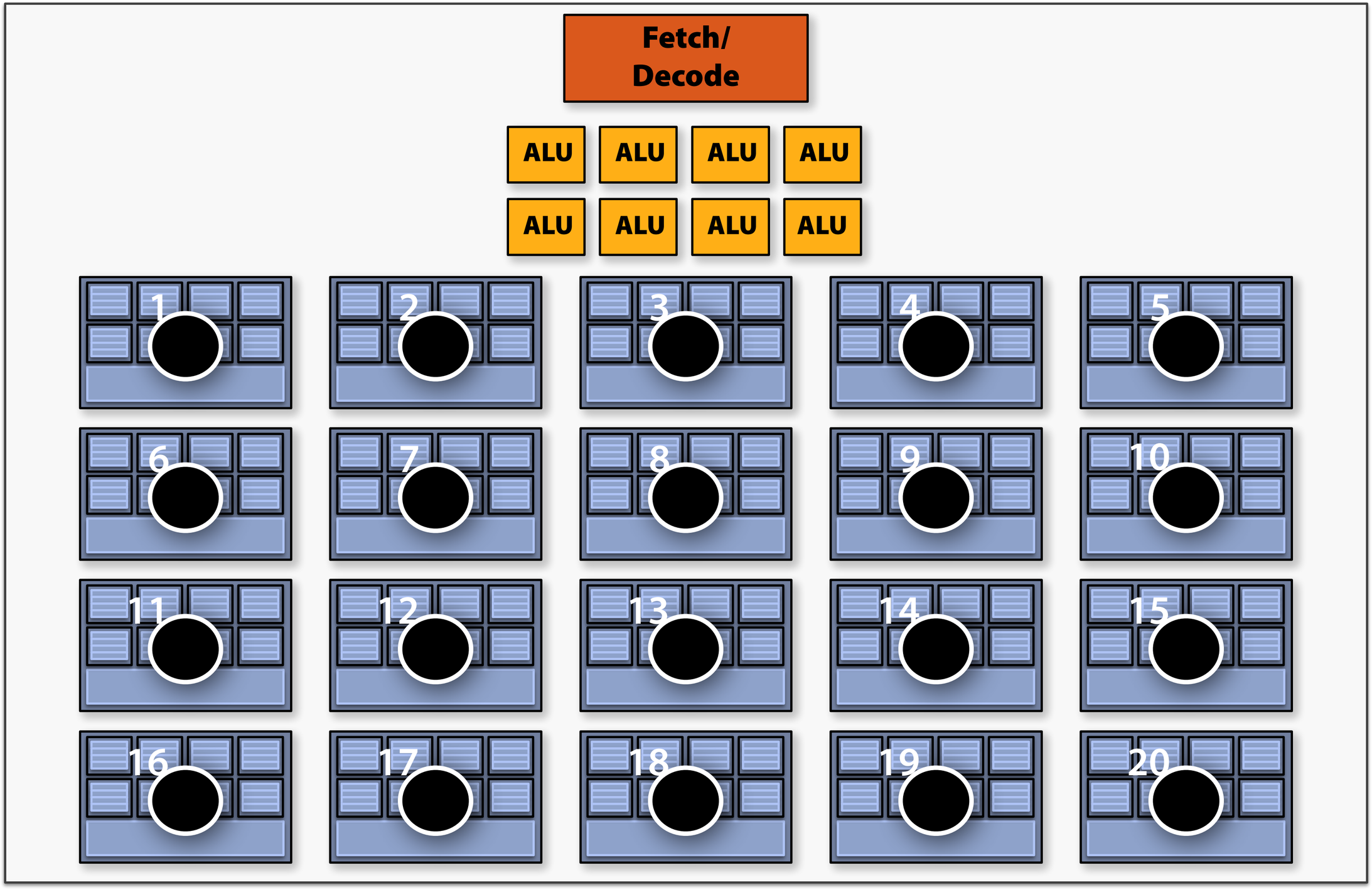

Time (clocks)
Threads 1 … 8
Threads 9… 16
Threads 17 … 24
Threads 25 … 32
1
2
3
4
Stall
Start
Runnable
Done!
Stall
Start
Runnable
Done!
Stall
Start
Runnable
Done!
Stall
Runnable
Done!
Increasing run time of one group
allows maximizing throughput of many groups

Storing contexts

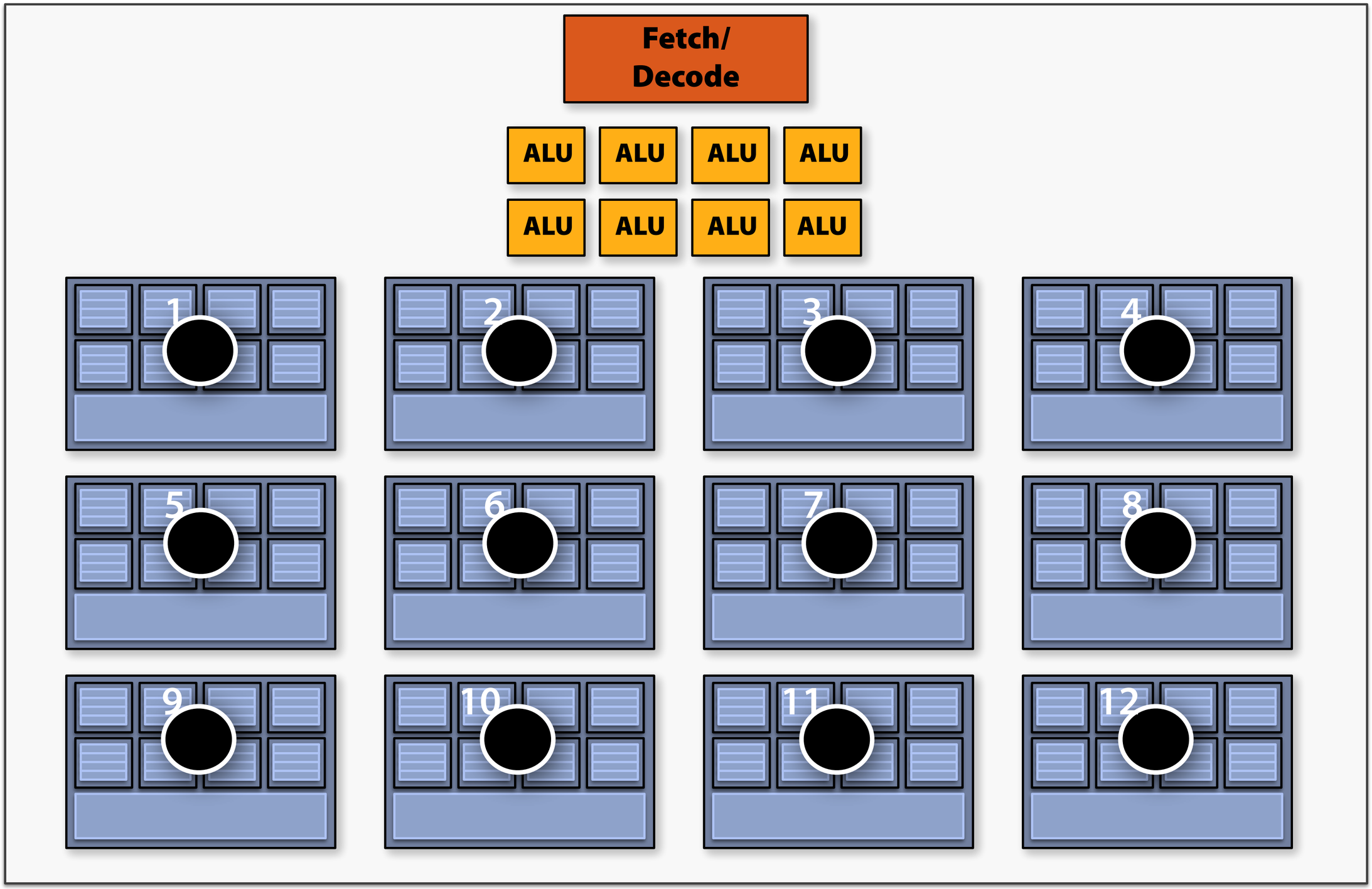


Twenty small contexts

(maximal latency hiding ability)

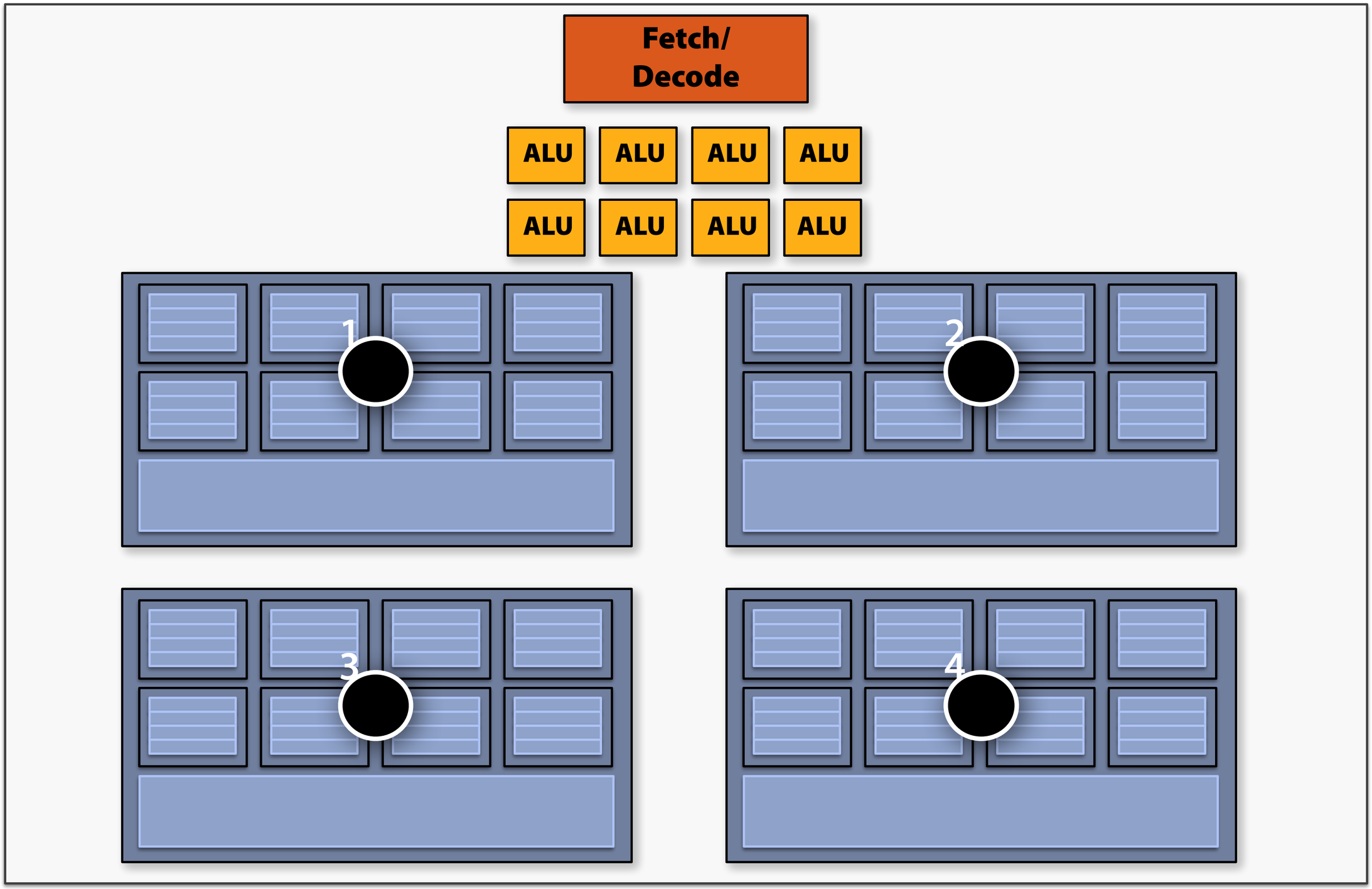


Twelve medium contexts



Four large contexts

(low latency hiding ability)



Clarification

- Interleaving between contexts can be managed by HW or SW (or both!)
- NVIDIA / AMD Radeon GPUs
  - HW schedules / manages all contexts (lots of them)
  - Special on-chip storage holds thread state
- Intel Xeon Phi
  - HW manages four x86 (big) contexts at fine granularity
  - SW scheduling interleaves many groups of threads on each HW context
  - L1–L2 cache holds thread state (as determined by SW)

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 fragments)
  - The CUDA/OpenCL model expresses programs as scalar programs and has implicit sharing managed by hardware

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

Computation/Memory Hierarchy (v100)

Level	Computation	Memory
Global	Kernels	DRAM (32 GB), L2 cache (6 MB)
Per-block	Blocks (MIMD within a kernel) (84)	Shared/L1: 96 kB/SMX x 84 SMs = 7.7 MB)
Per-warp	Warps (MIMD within a block)	(no explicit storage)
Per-thread	Threads (32-wide SIMD within a thread) (≥100k)	Registers (64 kB/SM x 84 SMs = 5 MB)

An efficient GPU workload …

- Has thousands of independent pieces of work
  - Uses many ALUs on many cores
  - Supports massive interleaving for latency hiding

- Is amenable to instruction stream sharing
  - Maps to SIMD execution well
  - Minimizes *branch divergence*

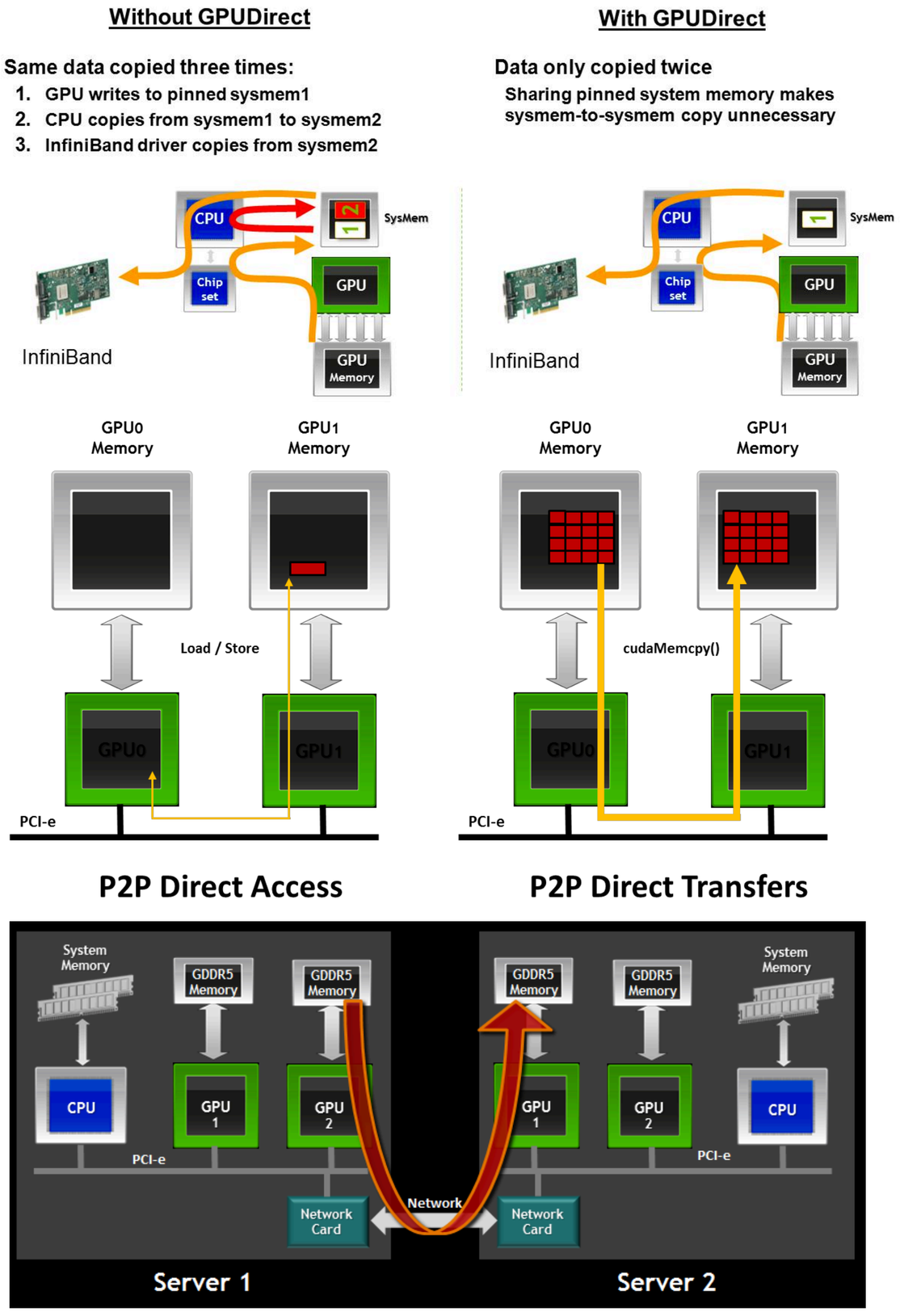
- Is compute-heavy: the ratio of math operations to memory access is high
  - Not limited by bandwidth
  - But even if it is, GPUs have high main memory bandwidth

Structuring a GPU Program

- CPU assembles input data
- CPU transfers data to GPU (GPU "main memory" or "device memory")
  - This used to be a manual copy, but now it's a single address space
- CPU calls GPU program (or set of kernels). GPU runs out of GPU main memory.
- When GPU finishes, CPU copies back results into CPU memory
- Recent interfaces allow overlap between communicate/ compute

Multi-GPU with GPUDirect

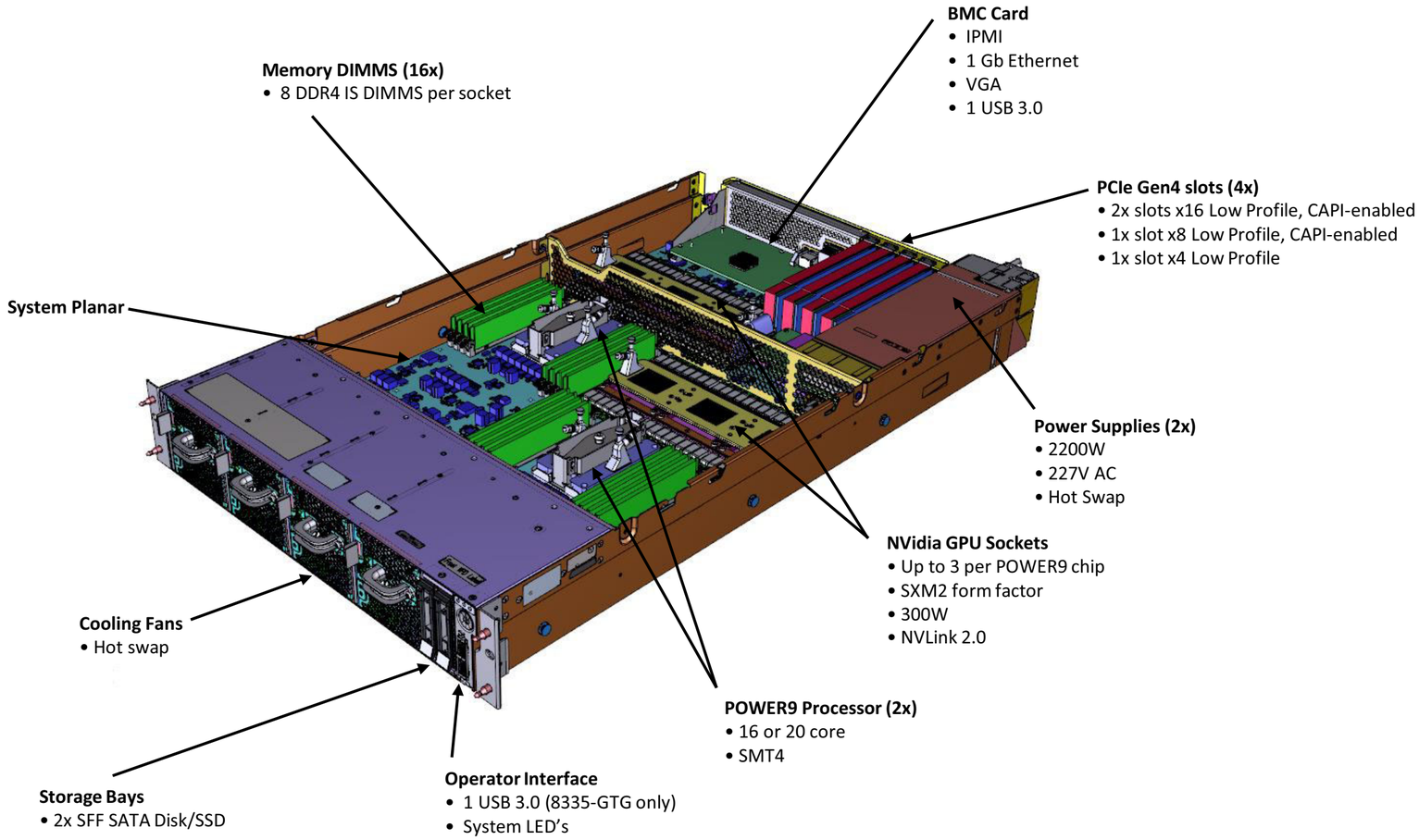
- MPI is dominant
- GPUDirect Shared Memory (2010)
- GPUDirect Peer-to-Peer (P2P) Communication between GPUs on the same PCIe bus (2011)
- GPUDirect Support for RDMA, introduced with CUDA 5 (2012)

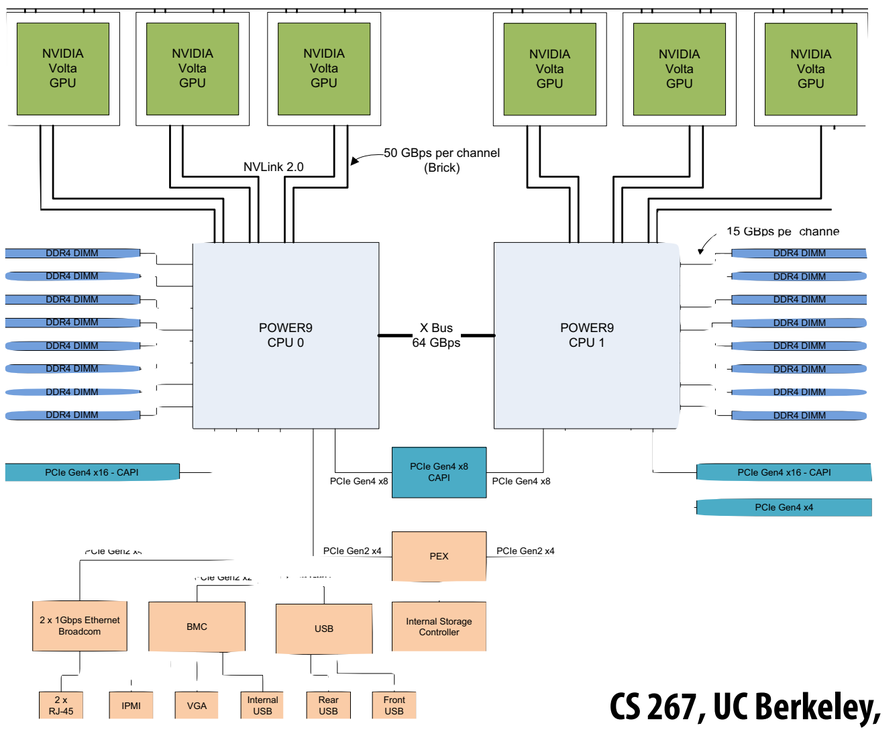


[https://developer.nvidia.com/gpudirect]

IBM Power Systems AC922 (Mar. 2018)

- 2U chassis
- 2 POWER9 processors (16 or 20 core)
- up to 1 TB DRAM
- 6 × NVIDIA V100 GPUS
- 4 × PCIe 4 connections
- NVLink 2.0





https://www.redbooks.ibm.com/redpapers/pdfs/redp5472.pdf

Programming Model Big Idea #1

- One thread per data element.
- Doesn't this mean that large problems will have millions of threads?

Programming Model Big Idea #2

- Write one program.
- That program runs on ALL threads in parallel.
- Software terminology here is "SPMD": single-program, multiple-data.
- Hardware terminology here is "SIMT": single-instruction, multiple-thread.
  - Roughly: SIMD means many threads run in lockstep; SIMT means that some divergence is allowed and handled by the hardware

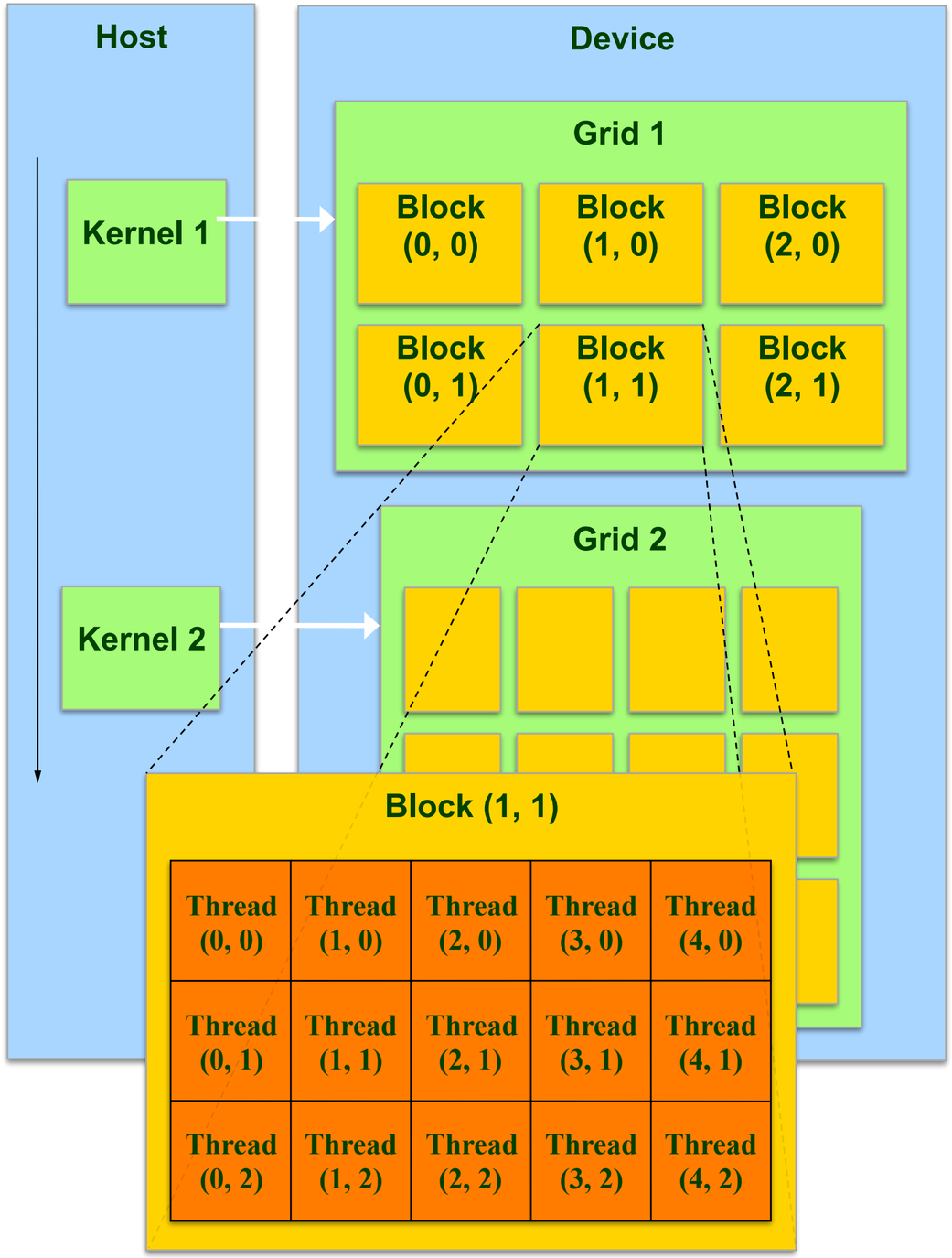
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One SIMT kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA must use 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions:
Device = GPU;
Host = CPU;
Kernel = function that runs on the device

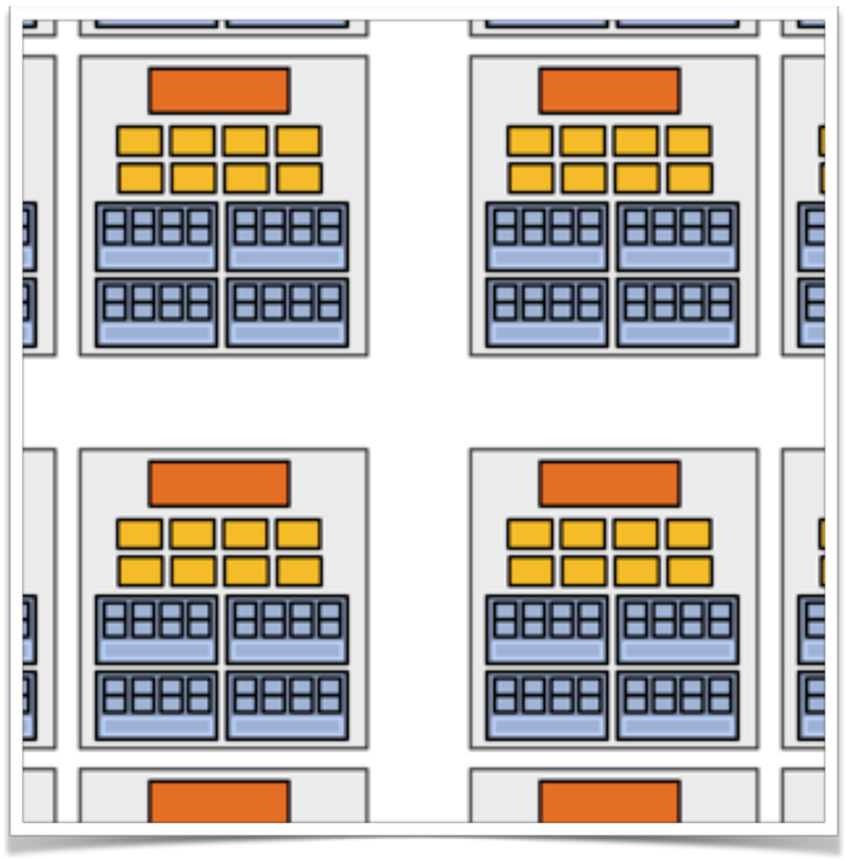
Programming Model Overview

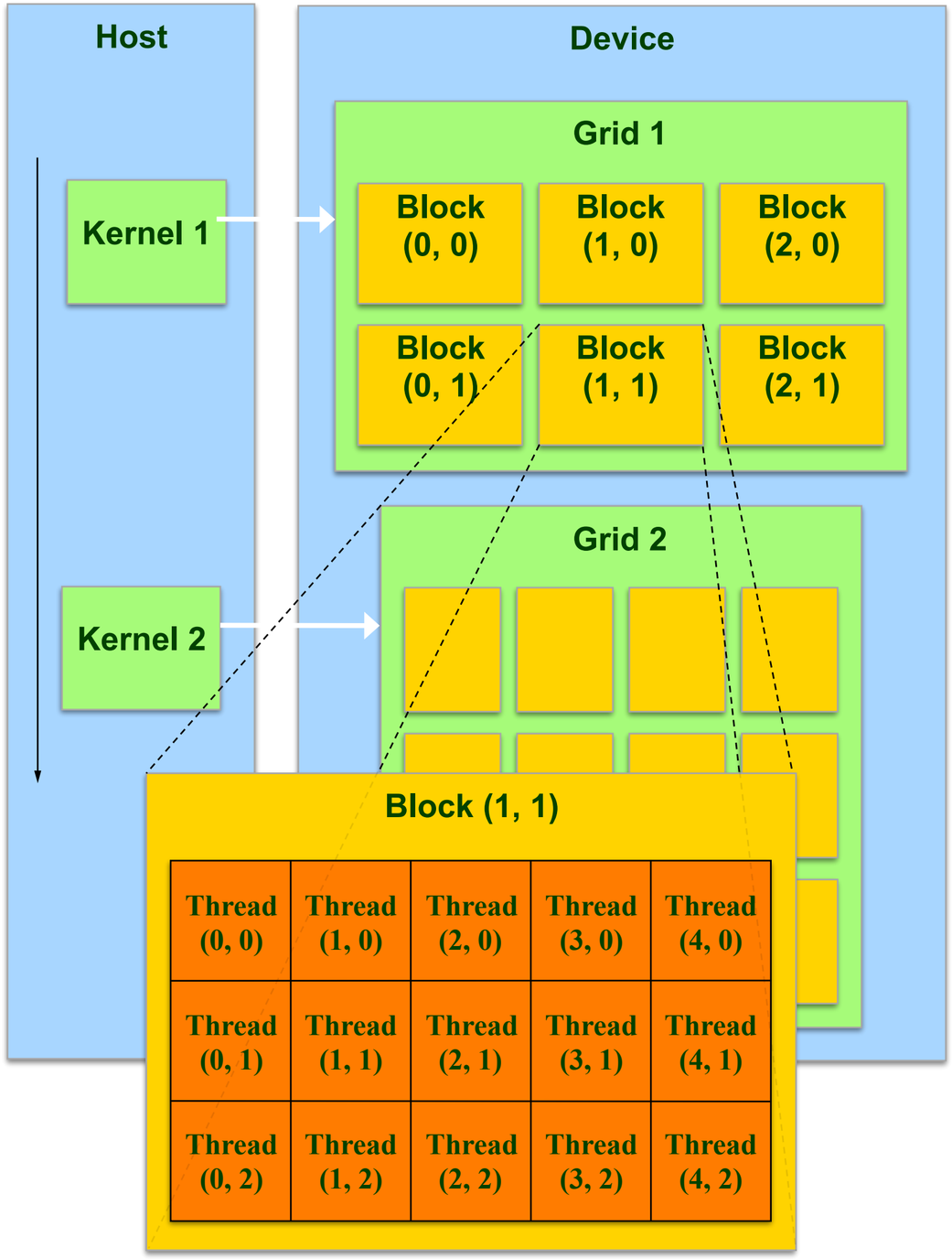
- A kernel is executed as a grid of thread blocks
- The programmer specifies the dimensions of the grid and thread block
- A thread block is a batch of threads that can cooperate with each other (shared memory, synchronization)
- Two threads from two different blocks cannot cooperate
  - Blocks are independent



What The Hardware Does

- Grids run on the entire machine
- Blocks are mapped to cores
- Threads are mapped to scalar processors





TESLA V100

21B transistors
815 mm²

80 SM
5120 CUDA Cores
640 Tensor Cores

16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink

*full GV100 chip contains 84 SMs

[INSIDE THE VOLTA GPU ARCHITECTURE AND CUDA 9]

Axel Koehler, GTC Europe 2017

VOLTA GV100 SM

Redesigned for Productivity

Completely new ISA
Twice the schedulers
Simplified Issue Logic
Large, fast L1 cache
Improved SIMT model
Tensor acceleration

	GP100	GV100
FP32 units	64	64
FP64 units	32	32
INT32 units	NA	64
Tensor Cores	NA	8
Register File	256 KB	256 KB
Unified L1/Shared memory	L1: 24KB Shared: 64KB	128 KB
Active Threads	2048	2048

CUDA vs. OpenCL terminology

CUDA term	OpenCL term
GPU	Device
(Streaming) Multiprocessor	Compute unit
Scalar core	Processing element
Global memory	Global memory
Shared (per-block) memory	Local memory
Local memory (automatic, or local)	Private memory
Block	Work-group
Thread	Work-item

[Matt Harvey, "Experiences porting from CUDA to OpenCL"]

Let's look at CUDA code!

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

(1) Use __global__ to indicate a GPU function

```
// CUDA Kernel function to add the elements
// of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

```
// CUDA Kernel function to add the elements
// of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

(2) Allocate GPU-accessible memory

```
// Allocate Unified Memory --
// accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

```
// CUDA Kernel function to add the elements
// of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}
```

```
// Allocate Unified Memory --
// accessible from CPU or GPU
float *x, *y;
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
```

(3) Launch the kernel on the GPU

```
// Run kernel on 1M elements
// on the GPU
add<<<1, 1>>>(N, x, y);
// kernel calls are non-blocking

// Wait for GPU to finish
// before accessing on host
cudaDeviceSynchronize();
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// GPU function to add the elements of two arrays
__global__ void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

Non-idealities:

- Runs on one thread. No parallelism.

- Launching more than one thread results in a race condition (all threads would compute the entire array)

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// GPU function to add the elements of two arrays
__global__ void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

(1) Launch 256 threads instead of 1

```
add<<<1, 256>>>(N, x, y);
```

This runs 256 threads in SIMD-parallel fashion on one core

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// GPU function to add the elements of two arrays
__global__ void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

(2) Divide work among *stride* threads

```
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}
```

Each thread knows its thread and block ID, and the dimensions of the grid and block.

```
add<<<1, 256>>>(N, x, y);
```

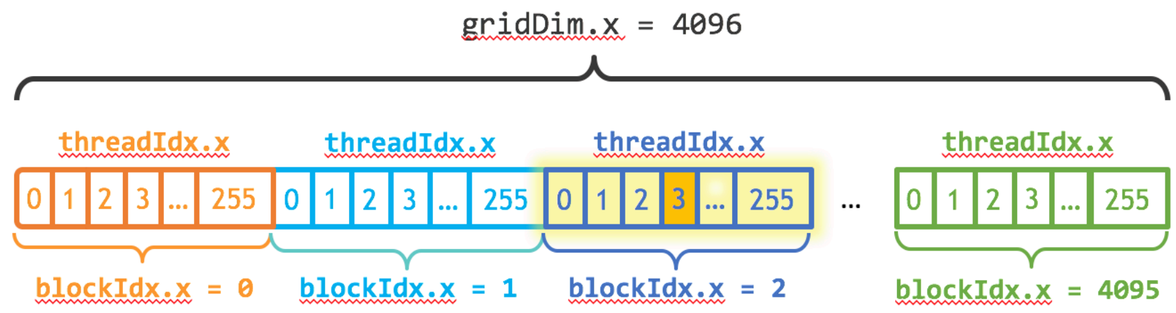
https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

```
#include <iostream>
#include <math.h>

// GPU function to add the elements of two arrays
__global__
void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 256>>>(N, x, y);
  cudaDeviceSynchronize();

  // … for space, remove error checking/free

  return 0;
}
```

(3) Divide work among *more** threads

```
__global__
void add(int n, float *x, float *y)
{
  int index = blockIdx.x *
    blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

Each thread knows its thread and block ID, and the dimensions of the grid and block.




```
index = blockIdx.x  * blockDim.x + threadIdx.x
index =    (2)      *   (256)    +     (3)      = 515
```

* *more* = numBlocks × blockSize = *N*

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

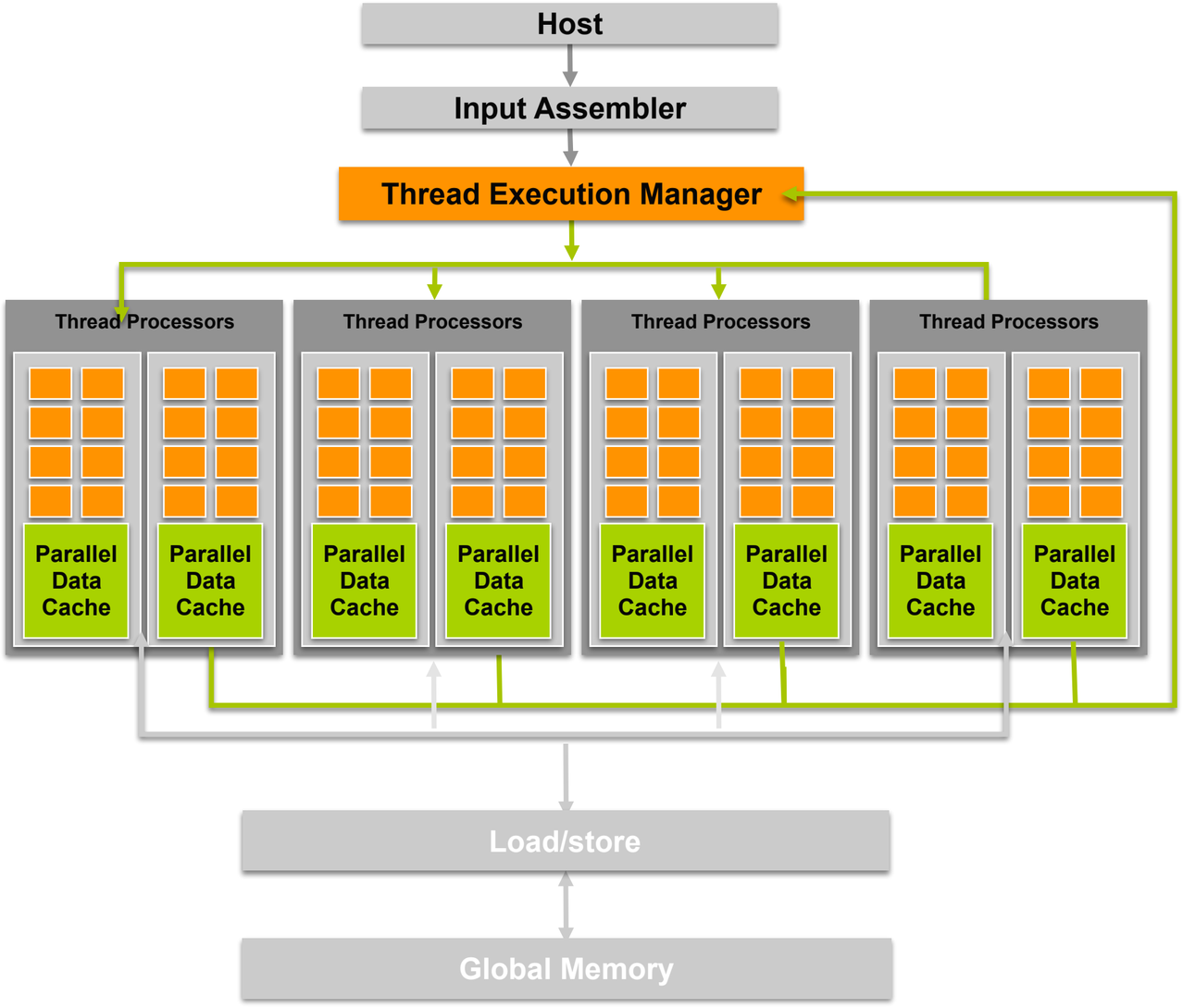
Performance

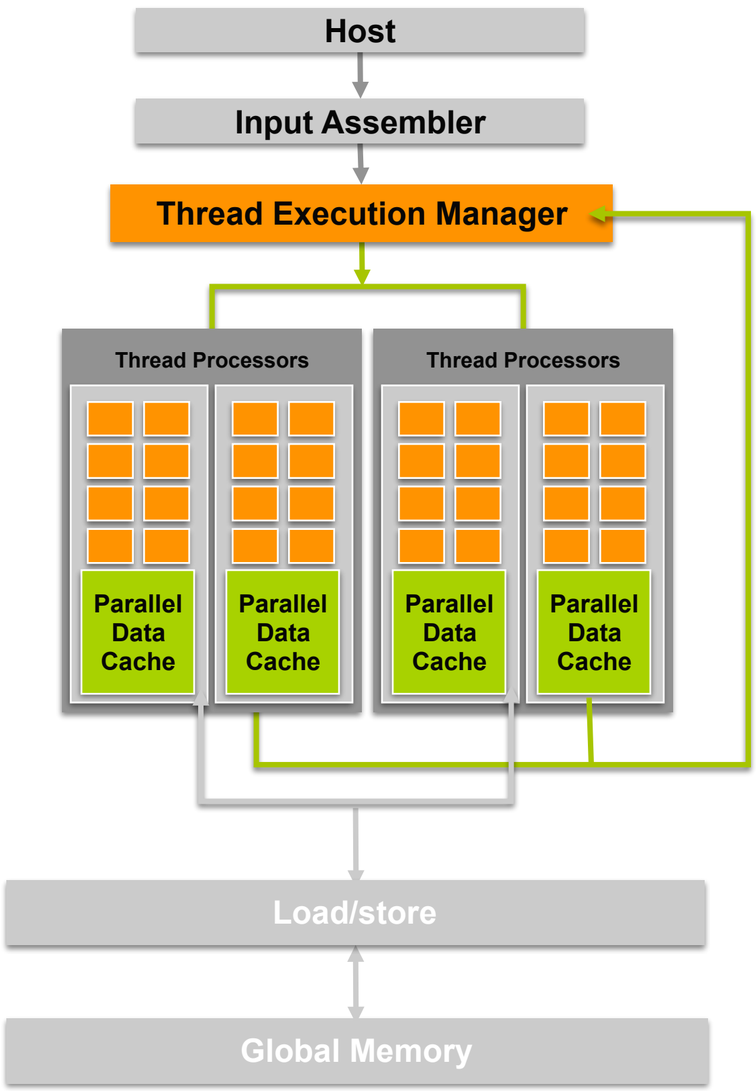
Version	Laptop (GT 750M)		Server (Tesla K80)	
	Time	Bandwidth	Time	Bandwidth
1 CUDA Thread	411 ms	30.6 MB/s	463 ms	27.2 MB/s
1 CUDA Block	3.2 ms	3.9 GB/s	2.7 ms	4.7 GB/s
Many CUDA blocks	0.68 ms	18.5 GB/s	0.094 ms	134 GB/s

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

Scaling the Architecture

- Same program
- Scalable performance





There's lots of dimensions to scale this processor with more resources. What are some of those dimensions? If you had twice as many transistors, what could you do with them?

What *should* you do with more resources? In what dimension do you think NVIDIA will scale future GPUs?

Programming Model Big Idea #3

- *Scalable execution*
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling

- Hierarchical execution model
  - Decompose problem into sequential steps (kernels)
  - Decompose kernel into computing parallel blocks
  - Decompose block into computing parallel threads

This is very important!

- Hardware distributes independent blocks to SMs as available

CUDA Software Development Kit

CUDA Optimized Libraries:
math.h, FFT, BLAS, …

Integrated CPU + GPU
C Source Code

NVIDIA C Compiler (LLVM-based)

NVIDIA Assembly
for Computing (PTX)

CPU Host Code

CUDA
Driver

Debugger
Profiler

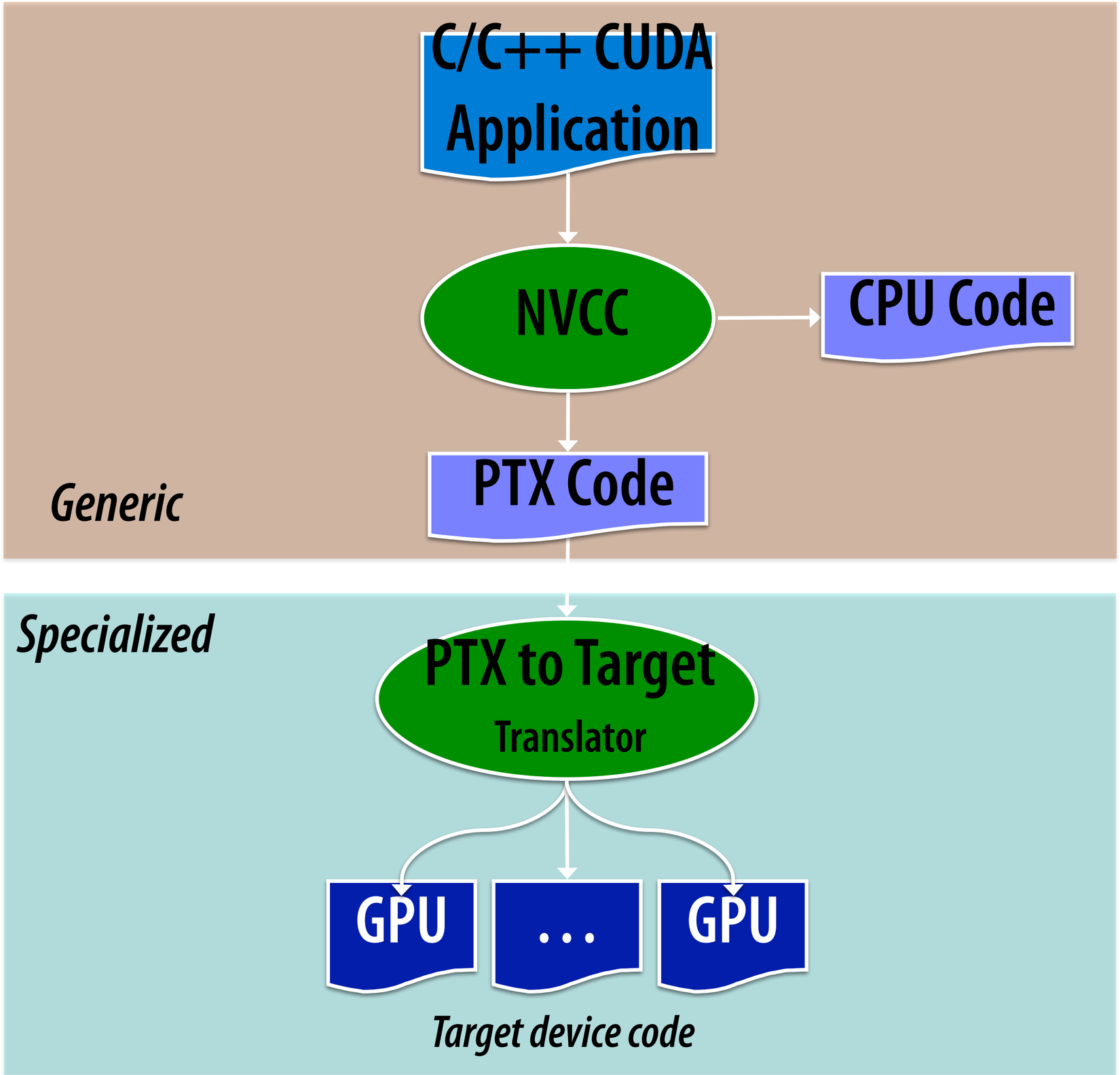
Standard C Compiler

SASS (GPU machine code)

GPU

CPU

Compiling CUDA for GPUs



OpenCL vs. CUDA

- OpenCL: Controlled by Khronos Group (consortium)
- Supported on:
  - GPUs (AMD, Intel, NVIDIA, ~~Apple~~, Qualcomm, Samsung, etc.)
  - CPUs (Intel, AMD, etc.)
  - DSPs, FPGAs
- CUDA has single-program model. OpenCL separates programs into host vs. kernel code.
- Fewer {language, functional} features than CUDA.
- OpenCL programs have more boilerplate (harder for beginners)
- Generally functionally compatible across devices, but not necessarily performance-compatible