EECS 570

Lecture 14

GPUs

Winter 2025

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/co4urses/eecs570/

Slides adapted from instructional material with D. Kirk and W. Hwu, Programming Massively Parallel Processors: A Handson Approach, Third Edition.
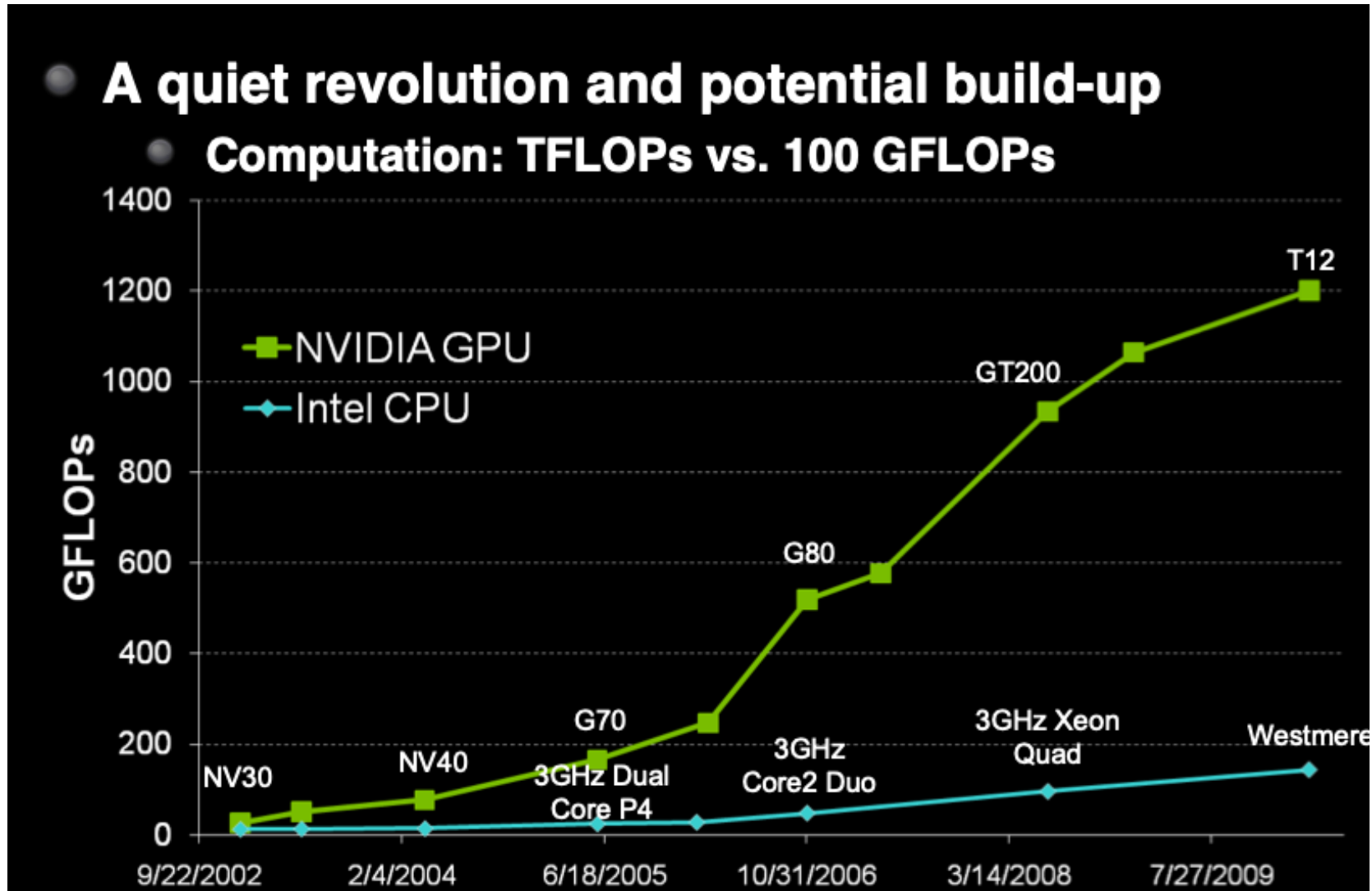Credits to Nikos Hardavellas (Northwestern), Reetu Das (UM), Thomas Wenisch
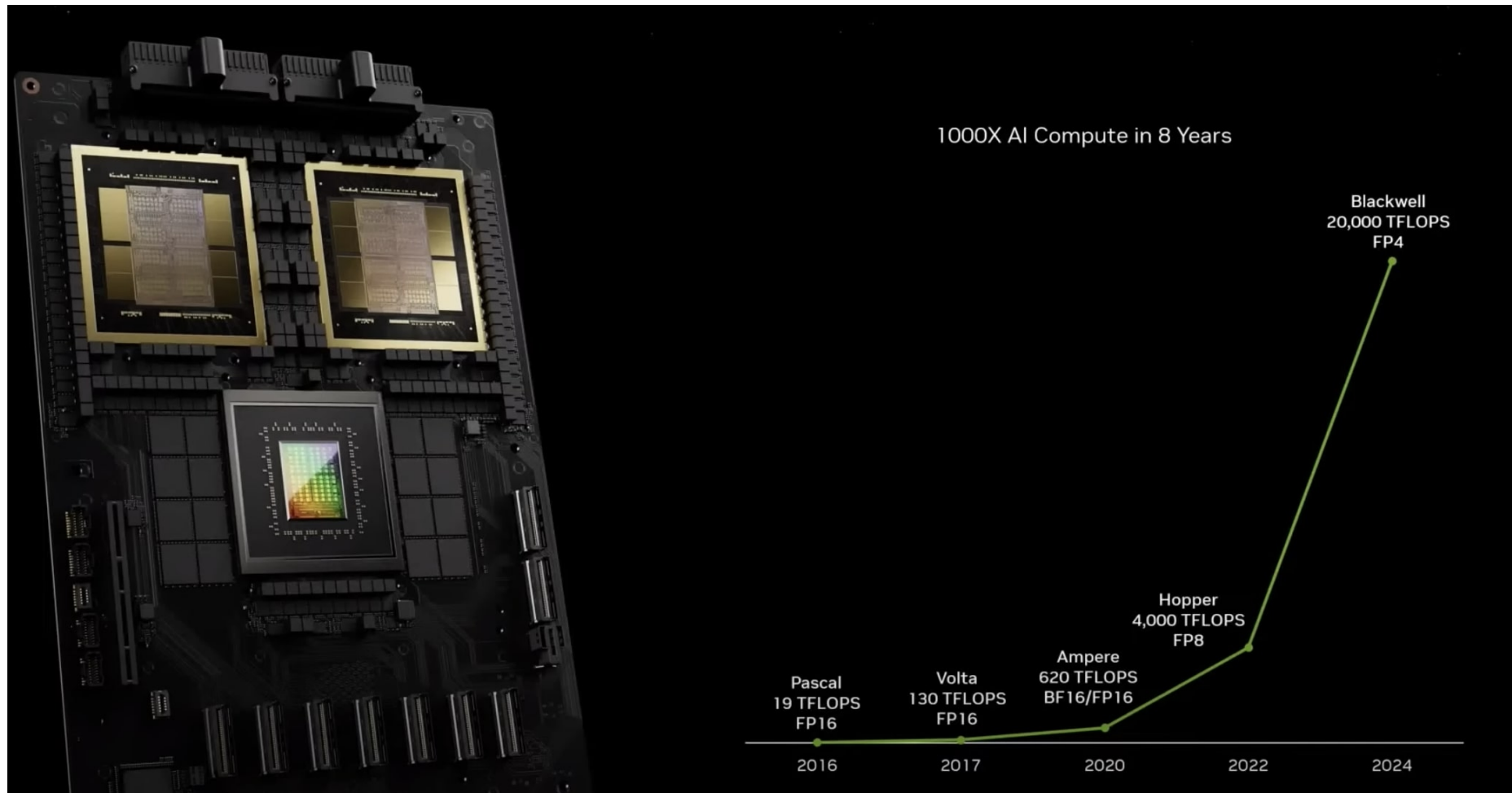
Readings

This week:

- Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Ch. 3.1-3.3, 4.1-4.3

Growth in GPUs

A quiet revolution and potential build-up

Computation: TFLOPs vs. 100 GFLOPs

Revolution in GPUs

1000X AI Compute in 8 Years

Blackwell
20,000 TFLOPS
FP4

Hopper
4,000 TFLOPS
FP8

Ampere
620 TFLOPS
BF16/FP16

Pascal
19 TFLOPS
FP16

Volta
130 TFLOPS
FP16

2016   2017   2020   2022   2024

Nvidia Increased Compute Power 1000X in 8 Years to 20 Petaflops in the Blackwell GPU | NextBigFuture.com
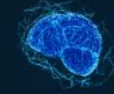
A major paradigm shift

18 future arenas of competition

These industries could yield **$29-48 trillion** in revenues and **$2-6 trillion** in profits by 2040.

E-commerce
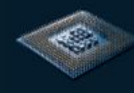
AI software and services

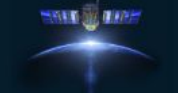Cloud services

Electric vehicles
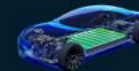
Digital advertising

Semiconductors

Shared autonomous vehicles

Space

Cybersecurity

Batteries

Modular construction

Streaming video

Video games

Robotics

Industrial and consumer biotechnology

Future air mobility

Drugs for obesity and related conditions

Nuclear fission power plants

McKinsey Global Institute

Domain-Specific and Generative AI Application Systems

Model Customization, Evaluation, Safety, and Explainability

Model Architecture and Techniques

Systems Optimization

Systems and Applications
Ecosystem: LangChain, LlamaIndex, Weights & Biases
NVIDIA: AI Workbench, NeMo Guardrails

Services and Microservices
Ecosystem: AWS Bedrock, AzureML, Cohere,
Google Vertex AI, OpenAI APIs
NVIDIA: NIM, Avatar Cloud Engine (ACE), BioNeMo, NeMo, Picasso

Models
Ecosystem: BLOOM, Llama, Mistral, MPT, OPT, Phi-2,
Getty Images AI Generator, Shutterstock 3D Generator
NVIDIA: BioMegatron, Edify, Nemotron

SDKs and Frameworks
Ecosystem: Colossal-AI, HuggingFace Transformers, PyTorch
NVIDIA: A2X, Megatron-LM, NeMo Framework, Riva, Picasso

Libraries
Ecosystem: XLA
NVIDIA: CUDA, CUTLASS, CV-CUDA, Megatron-Core, Megatron-LM,
NCCL, RAFT, Transformer Engine, TensorRT-LLM, Ray

Management and Orchestration
Ecosystem: Kubernetes, Nephele, Slurm, VMware
NVIDIA: Base Command Platform

Accelerated Infrastructure
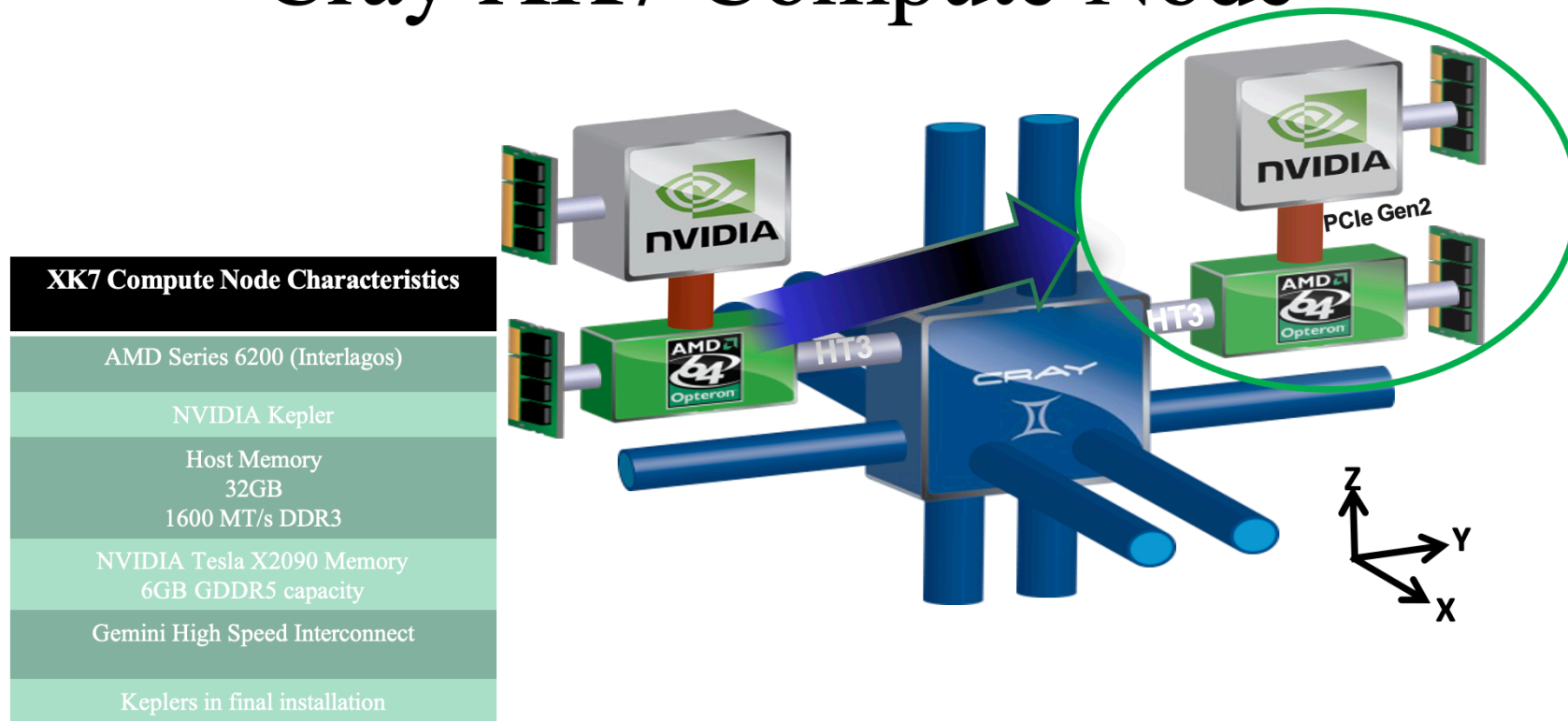NVIDIA: DPUs, GPUs, InfiniBand

AI computing stack

Computing at Exascale

El Capitan at Lawrence Livermore
National Laboratory (LLNL)

Performance is expected
to exceed 2 exaFLOPS,
which comes with a $600
million price tag.

Cray XK7 Compute Node

XK7 Compute Node Characteristics
AMD Series 6200 (Interlagos)
NVIDIA Kepler
Host Memory 32GB 1600 MT/s DDR3
NVIDIA Tesla X2090 Memory 6GB GDDR5 capacity
Gemini High Speed Interconnect
Keplers in final installation

CPUs: Latency Oriented Design

High clock frequency

Large caches
- Convert long latency memory accesses to short latency cache accesses
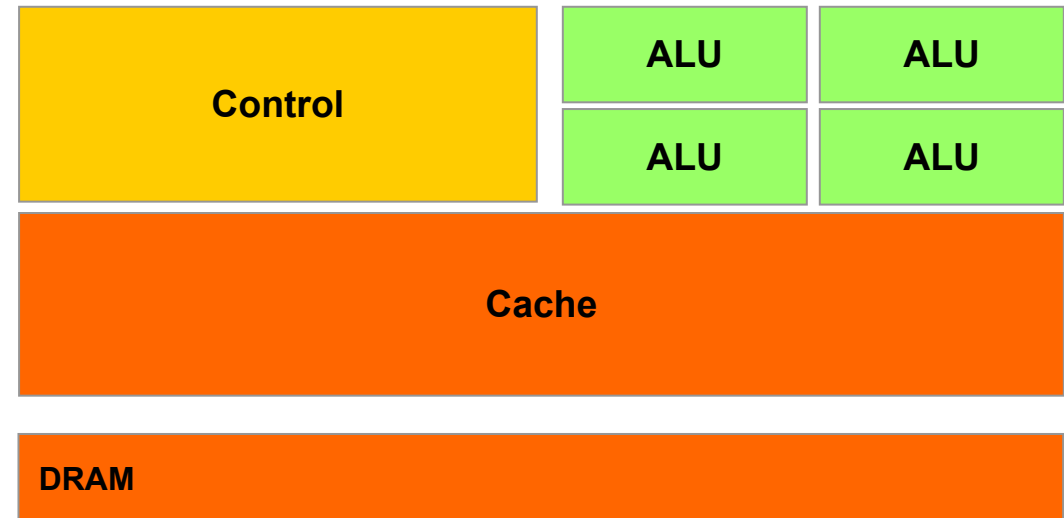
Sophisticated control
- Branch prediction for reduced branch latency
- Data forwarding for reduced data latency

Powerful ALU
- Reduced operation latency

CPU

| Control	ALU	ALU
	ALU	ALU

Cache

DRAM

GPUs: Throughput Oriented Design

- Moderate clock frequency
- Smaller caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
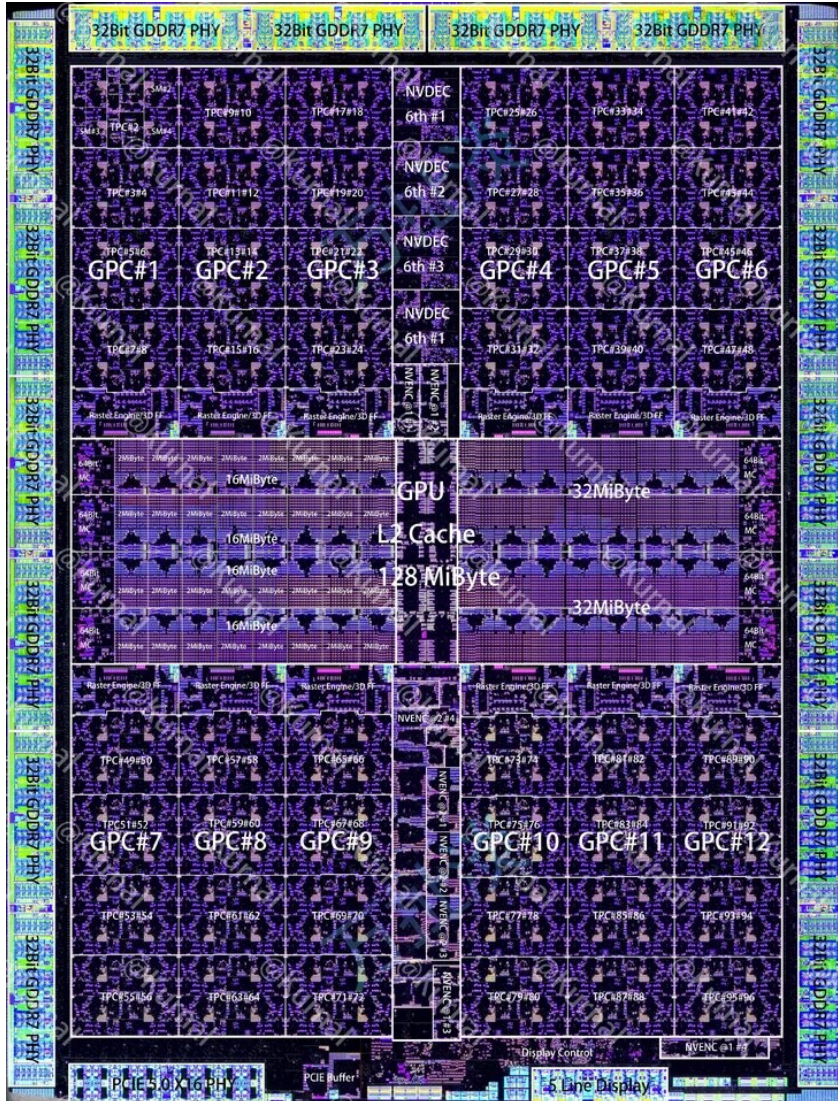- Require massive number of threads to tolerate latencies

GPU

DRAM
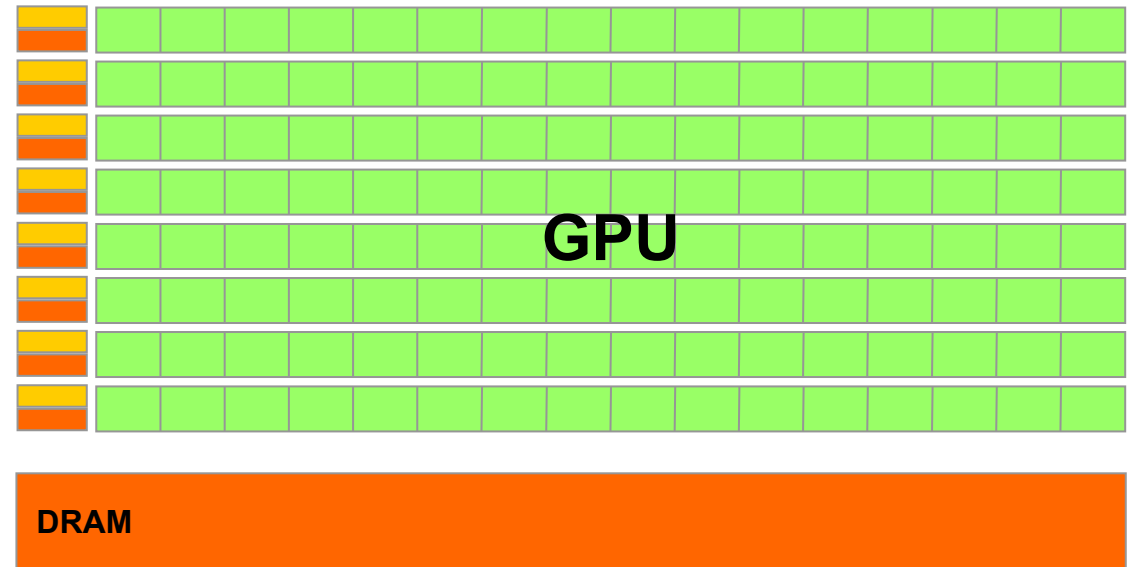
CPU vs. GPU

- Different design philosophies
  - CPU: A few out-of-order cores
  - GPU: Many in-order SIMD cores

CPU

| Control | ALU | ALU |
| | ALU | ALU |

Cache

DRAM

GPU

DRAM
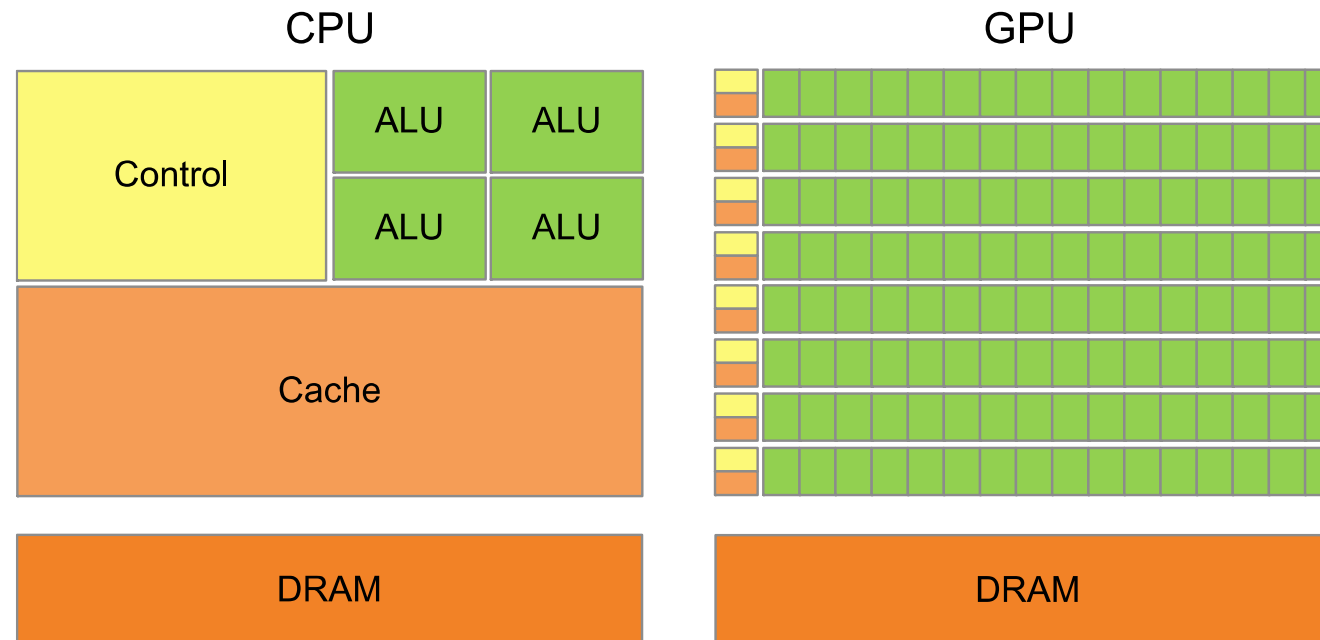
NVIDIA B100 (2024-25)

B100:

- **Streaming Multiprocessors (SMs):** 192

- **L1 Cache:** 128 KB per SM

- **L2 Cache:** 50 MB

- **Transistor Count:** 104 billion

- **Memory Size:** 192 GB HBM3e

- **Memory Bandwidth:** 8 TB/s

- **Power Consumption:** 700W

Each SM can execute 32 threads at a time

12

Exponential growth continues …

Supercharging Next-Generation AI and Accelerated Computing

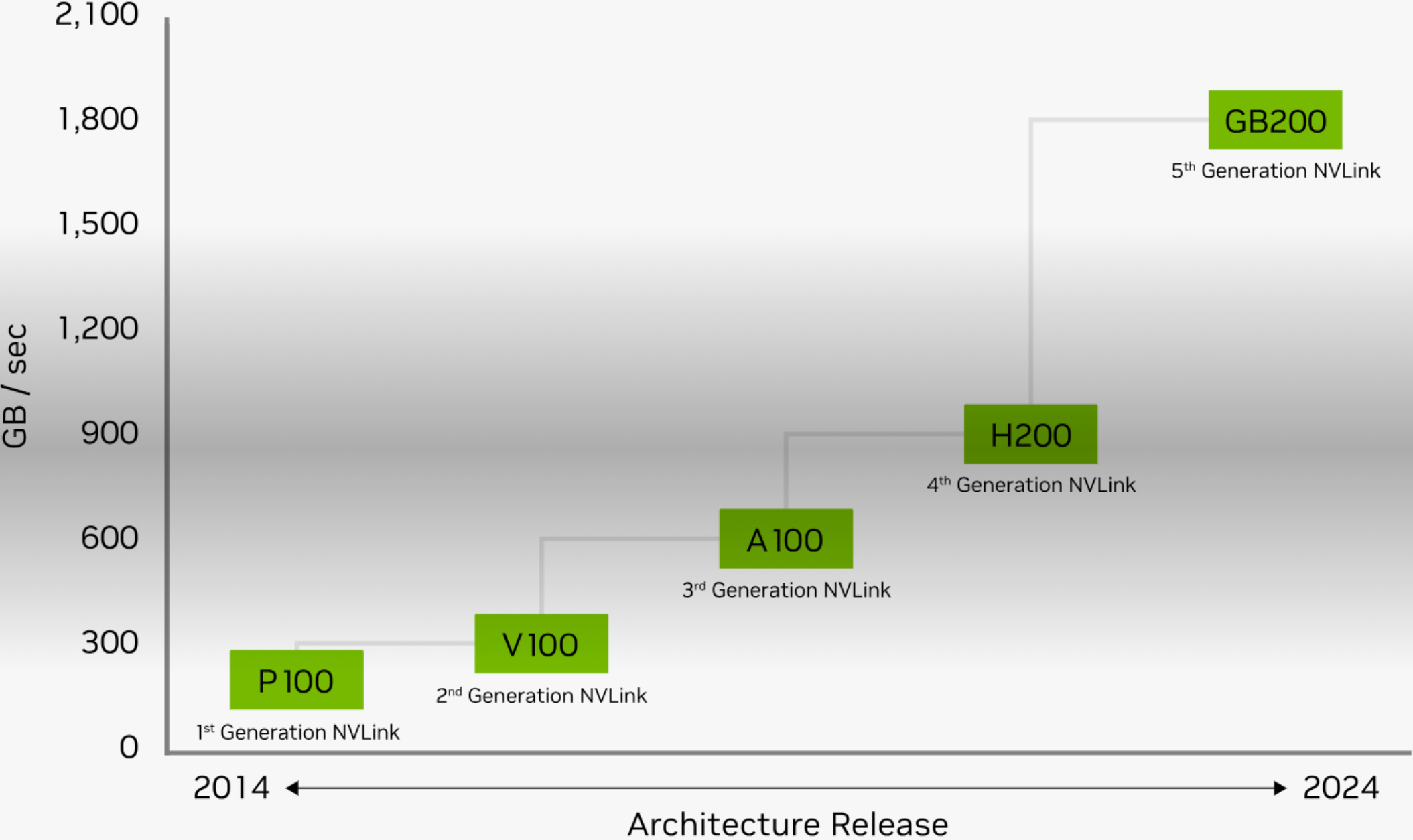| **LLM Inference**	**LLM Training**	**Energy Efficiency**	**Data Processing**
30X vs. NVIDIA H100 Tensor Core GPU	4X vs. H100	25X vs. H100	18X vs. CPU

NVLink At-Scale Performance

Massive
Parallelism -
Regularity

Applications Benefit from Both CPU and GPU
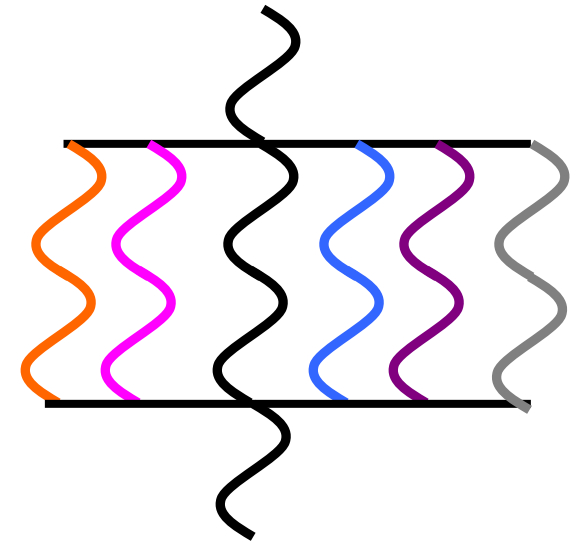
CPUs for sequential parts where latency matters

GPUs for parallel parts where throughput wins

CPUs can be 10+X faster than GPUs for sequential code

GPUs can be 10+X faster than CPUs for parallel code

Amdahl's Law

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

Speeding Up Real Applications

Big Idea: Amdahl's Law

$$\text{Speedup} = \frac{1}{(1 - F) + \dfrac{F}{S}}$$

Non-speed-up part → (1 - F)

Speed-up part → $\dfrac{F}{S}$

Example: the execution time of half of the program can be accelerated by a factor of 2.
What is the program speed-up overall?

$$\frac{1}{\dfrac{0.5 + 0.5}{2}} = \frac{1}{0.5 + 0.25} = 1.33$$

Load Balance

The total amount of time to complete a parallel job is limited by the thread that takes the longest to finish
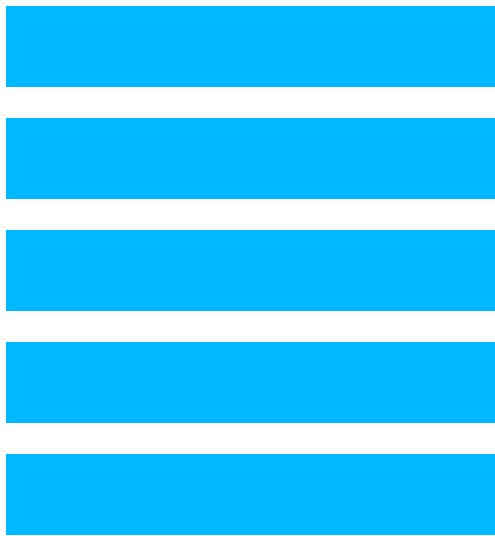
good
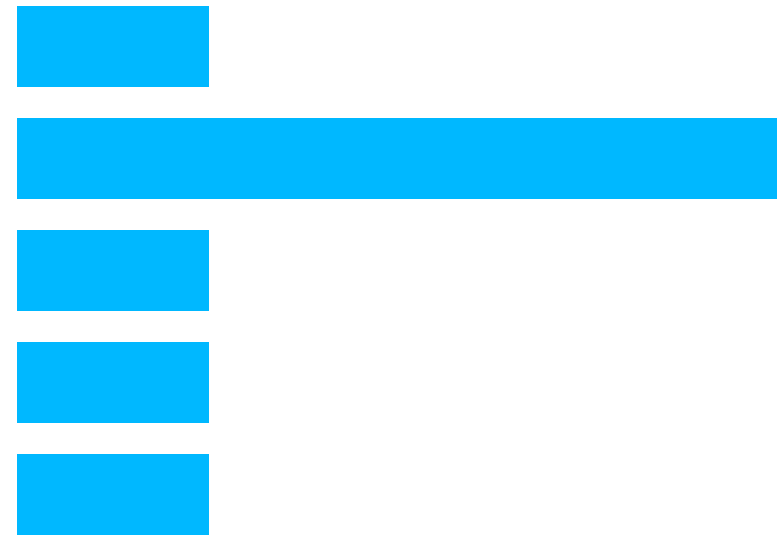
bad!

Memory Bandwidth Constraint

Memory Contentions in accessing critical dat
serialization

Massively parallel execution cannot afford se

Computation – Communication

> Slowest of the two determines performance

Global Memory Bandwidth

Ideal

Reality

GPUs and SIMD/Vector Data Parallelism

- Graphics processing units (GPUs)
  - ❏ How do they have such high peak FLOPS?
  - ❏ Ans: exploit massive data parallelism

- "SIMT" execution model
  - ❏ Single instruction multiple threads
  - ❏ Similar to both "vectors" and "SIMD"
  - ❏ A key difference: better support for conditional control flow

- Program it with CUDA or OpenCL (or Vulkan or Metal or …)
  - ❏ Extensions to C (or Objective-C in the case of Metal)
  - ❏ Perform a "shader task" (a snippet of scalar computation) over many elements
  - ❏ Internally, GPU uses scatter/gather and vector mask operations

Context: History of Programming GPUs

- "GPGPU"
  - Originally could only perform "shader" computations on images
  - So, programmers started using this framework for computation
  - Puzzle to work around the limitations, unlock the raw potential

- As GPU designers notice this trend…
  - Hardware provided more "hooks" for computation
  - Provided some limited software tools

- GPU designs are now fully embracing compute
  - More programmability features to each generation
  - Industrial-strength tools, documentation, tutorials, etc.
  - Can be used for in-game physics, etc.
  - Many application targets:
    - AI, graphics, data analytics, scientific computation, genomics

Throughput Computing:
Hardware Basics

Justin Hensley

Advanced Micro Devices, Inc

Graphics Product Group

What does a modern graphics API do?

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk
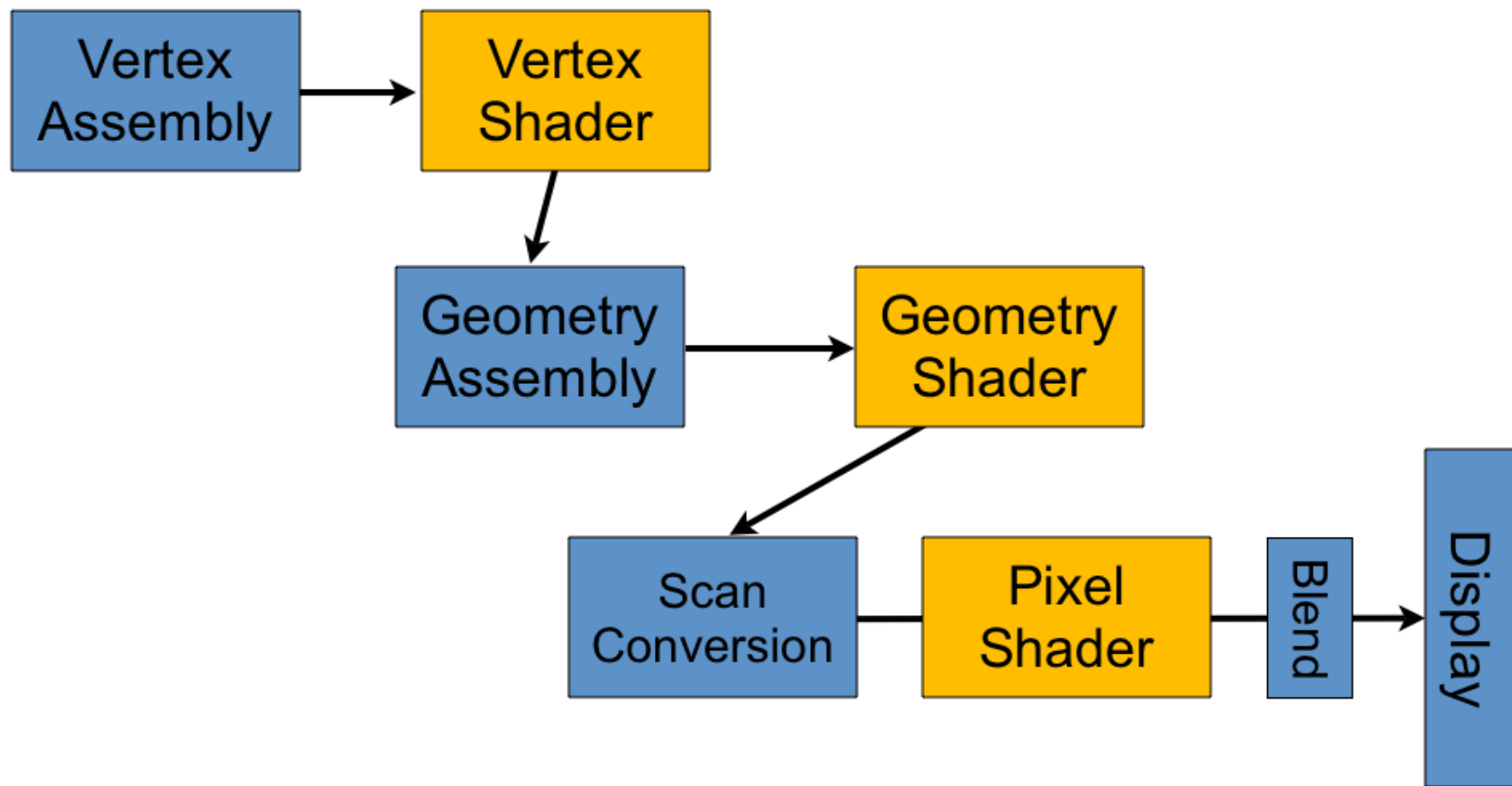
A Simple Program - Diffuse Shader

```
sampler mySamp;

Texture2D<float3> myTex;

float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)

{

  float3 kd;

  kd = myTex.Sample(mySamp, uv);

  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);

  return float4(kd, 1.0);

}
```

Each invocation is independent, but no explicitly exposed parallelism

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

SIGGRAPHASIA2008
NEW HORIZONS

Shader is compiled

1 Unshaded fragment in

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

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

1 Shaded fragment out

SIGGRAPHASIA2008
NEW HORIZONS

Exploit data parallelism! - add two cores

```
<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

Execute
(ALU)

Execution context

Fetch

Decode

Execute
(ALU)

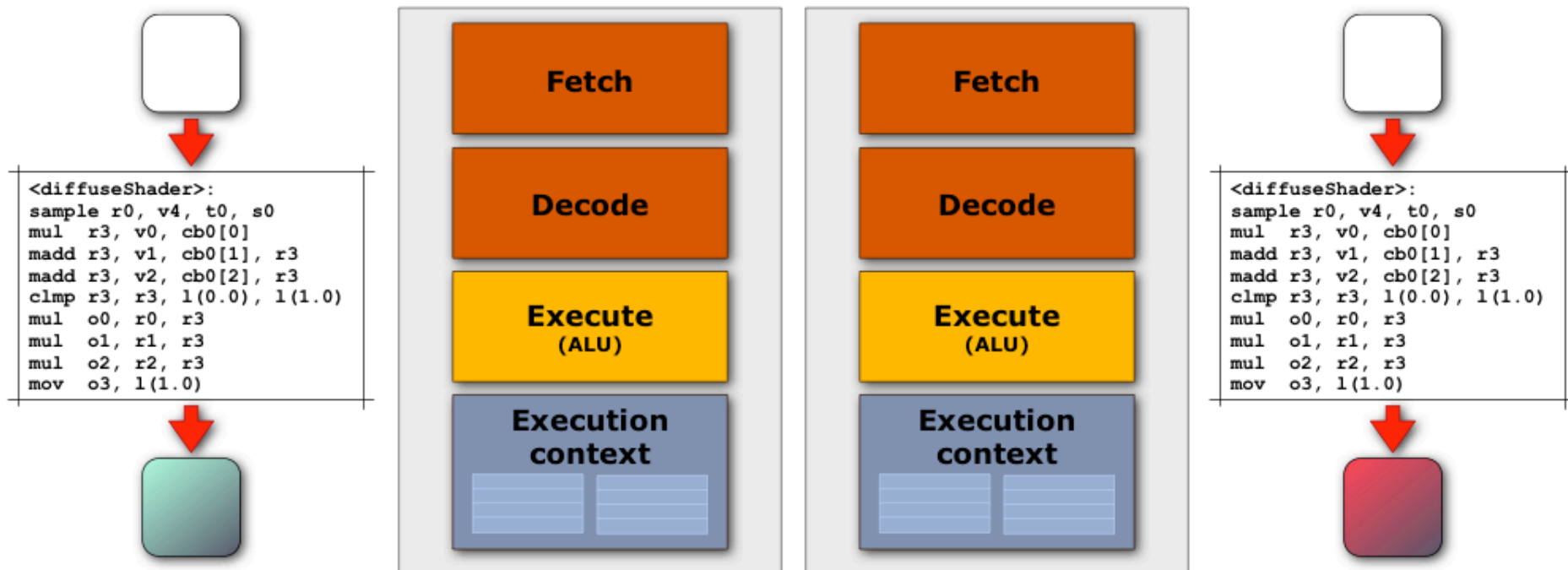**Execution context**

```
<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)
```
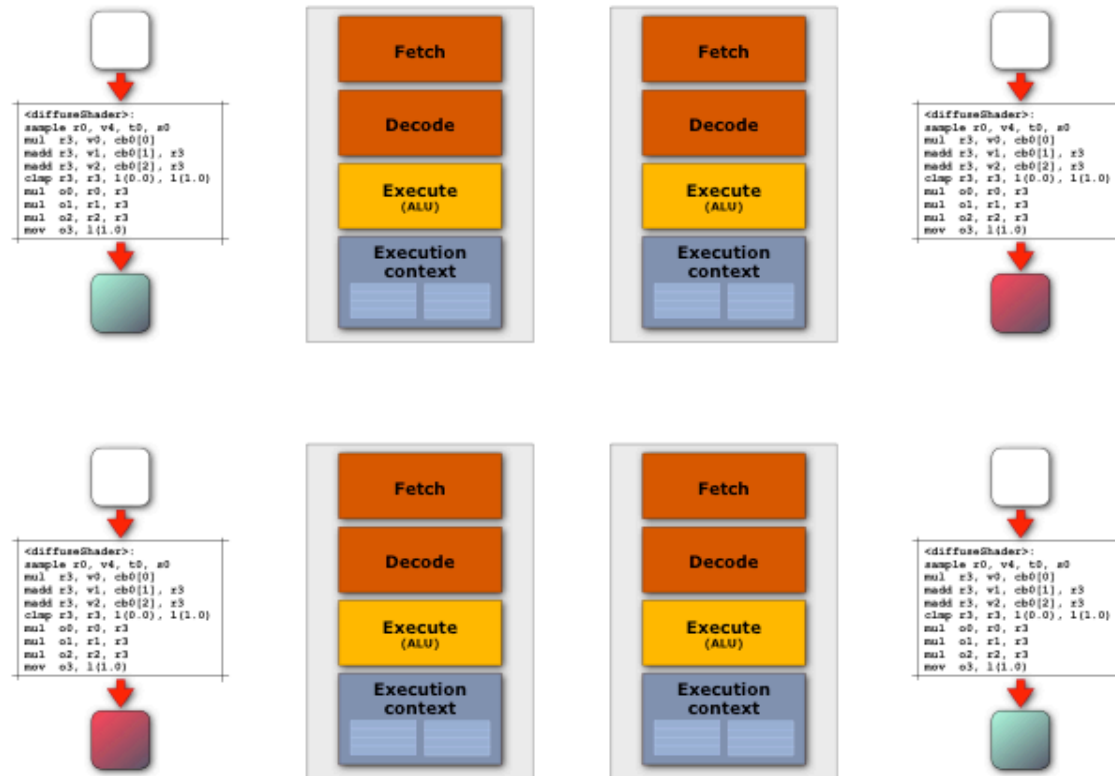
Each invocation is independent!

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

SIGGRAPH ASIA 2008
NEW HORIZONS

Add even more cores - four cores

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

SIGGRAPHASIA2008
NEW HORIZONS

How about even more cores - 16 cores

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk
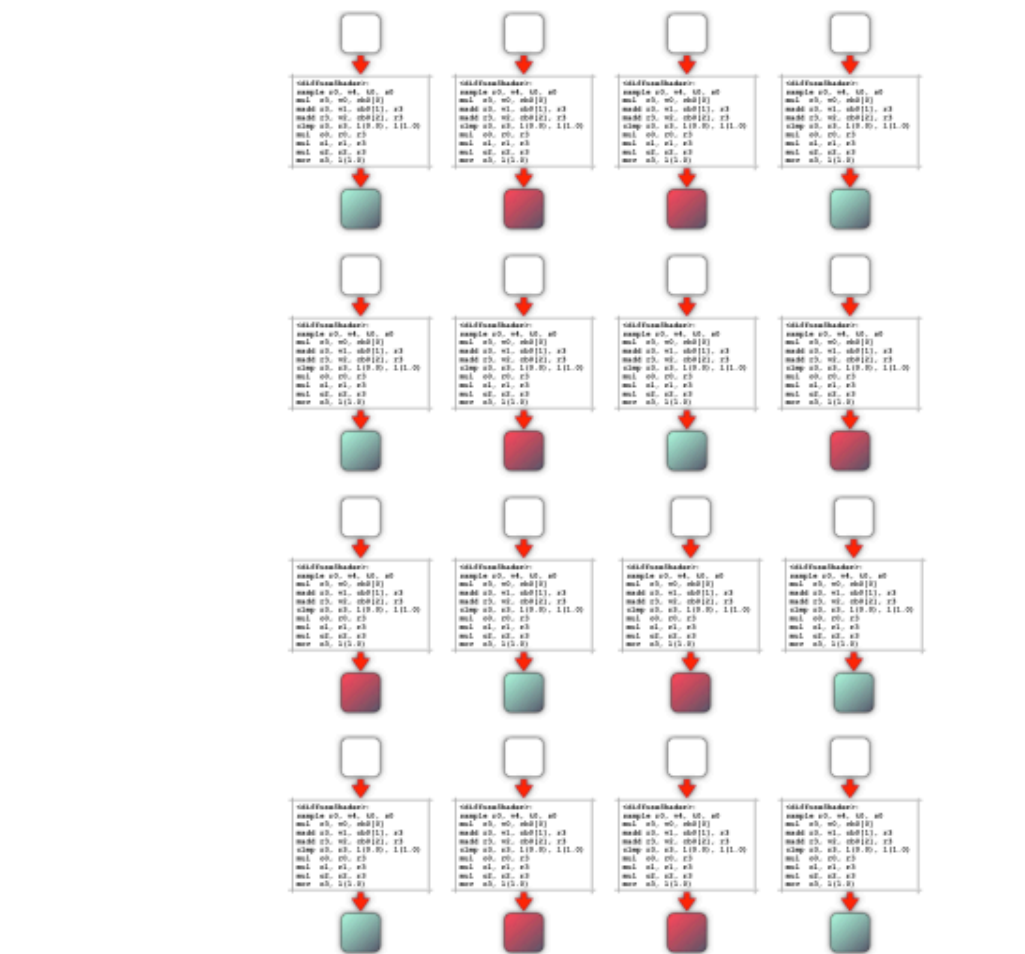
128 cores?

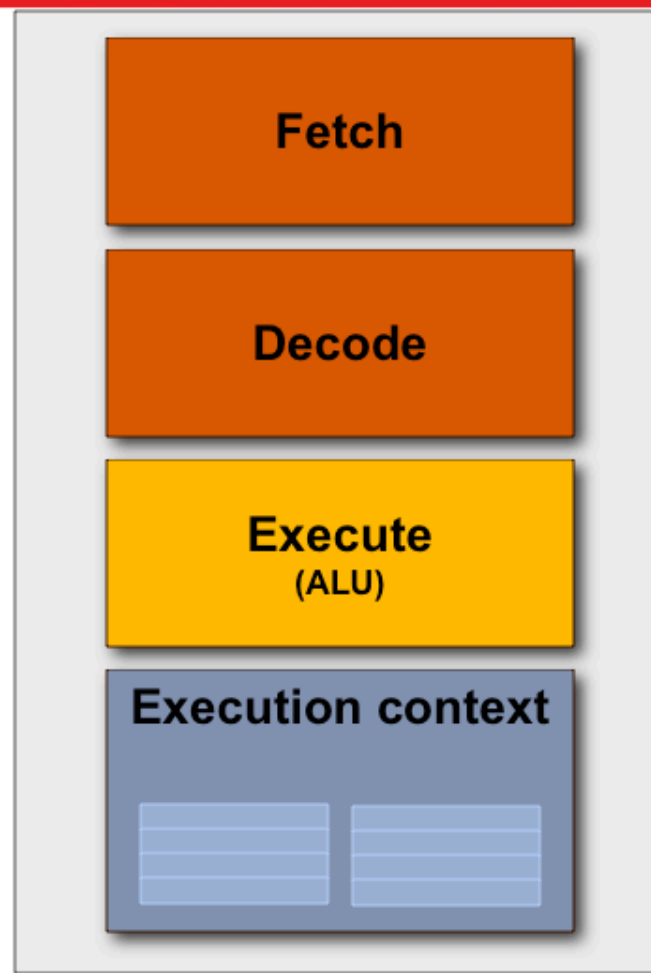How do you feed all these cores?

Think data parallel! - Graphics requires hardware process *lots* of "items" that share the same shader

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

Back to the simple core...

- **How do you feed all these cores?**

- Share cost of fetch / decode across many ALUs
- **SIMD** Processing

Fetch
Decode
Execute (ALU)
Execution context

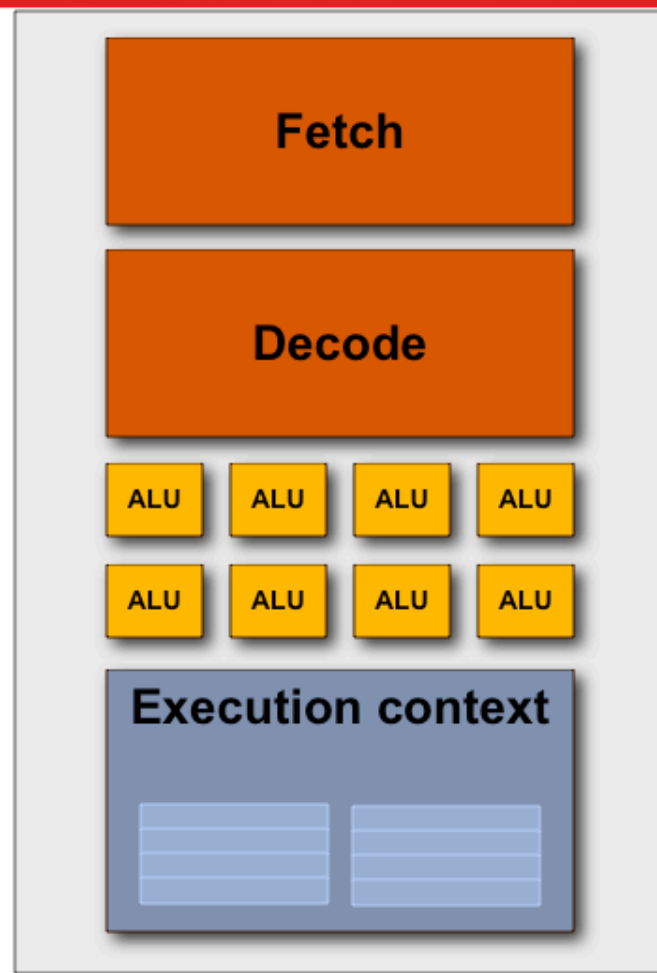adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

SIGGRAPH ASIA 2008
NEW HORIZONS
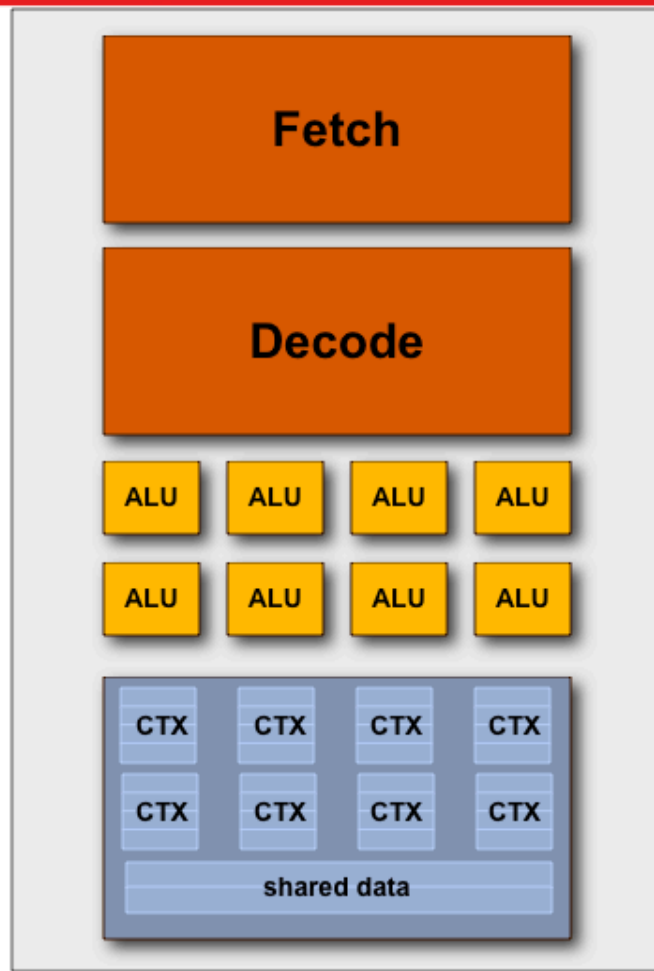
Back to the simple core...

- **How do you feed all these cores?**

- Share cost of fetch / decode across many ALUs
- **SIMD** Processing
  - **S**ingle
  - **I**nstruction
  - **M**ultiple
  - **D**ata

Fetch

Decode

ALU ALU ALU ALU

ALU ALU ALU ALU

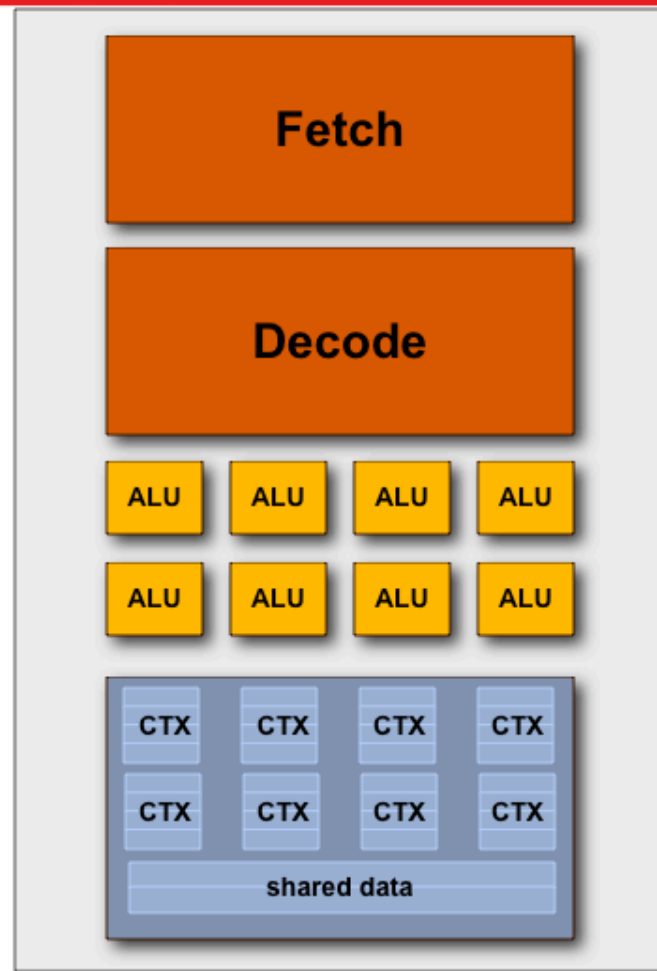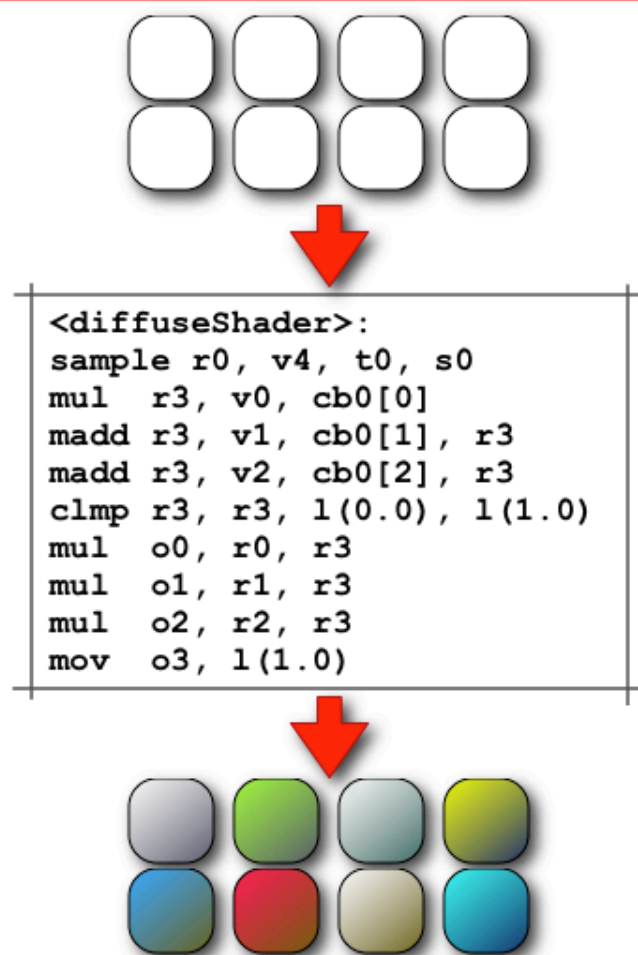Execution context

SIGGRAPHASIA2008
NEW HORIZONS

Back to the simple core...

- **How do you feed all these cores?**

- Share cost of fetch / decode across many ALUs
- **SIMD** Processing
  - **S**ingle

SIMD Processing does not imply SIMD instructions!

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

Back to a single core...

```
<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 | ALU | ALU | ALU |
| ALU | ALU | ALU | ALU |

| CTX | CTX | CTX | CTX |
| CTX | CTX | CTX | CTX |

shared data

128-Fragments in parallel

16 cores ➜ 128 ALUs (16 cores * 8 ALUs)
➜ 16 independent instruction streams

SIGGRAPHASIA2008
NEW HORIZONS

128-things in parallel

- **X** cores can work on primitives (triangles)
    - "geometry shader"
- **Y** cores can work on vertices
    - "vertex shader"
- **Z** cores can work on fragments
    - "pixel shader"
- **N cores can work on data/work/etc**
    - **"compute kernels"/"compute shaders"**
- Which cores working on what data changes over time

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

SIGGRAPHASIA2008
NEW HORIZONS

What about branching?
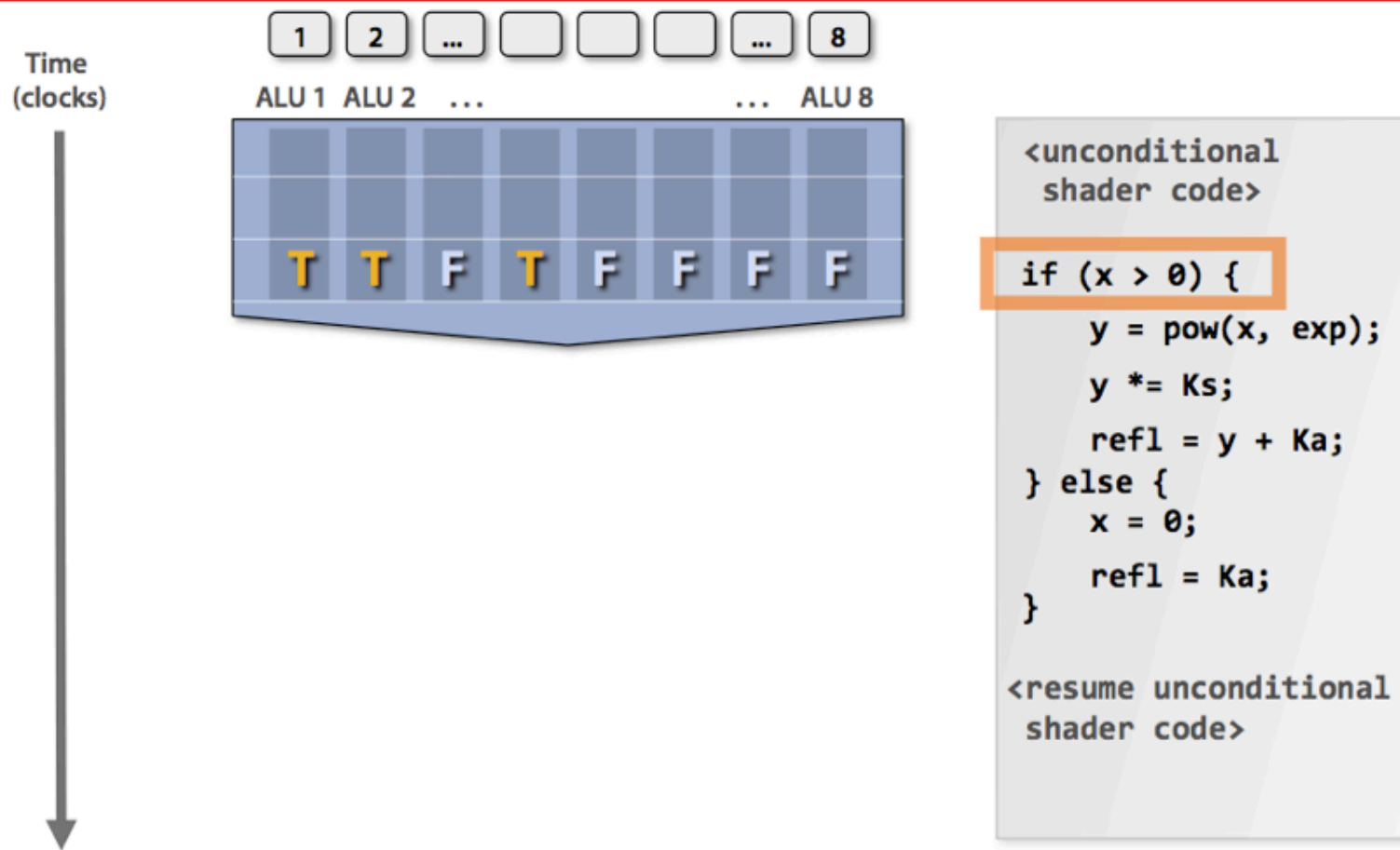


```
<unconditional
  shader code>

if (x > 0) {
    y = pow(x, exp);

    y *= Ks;

    refl = y + Ka;
} else {
    x = 0;

    refl = Ka;
}

<resume unconditional
  shader code>
```

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

What about branching?

Time (clocks)

1 2 8

ALU 1 ALU 2 ... ... ALU 8

T T F T F F F F

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);

    y *= Ks;

    refl = y + Ka;
} else {
    x = 0;

    refl = Ka;
}

<resume unconditional
 shader code>
```

SIGGRAPHASIA2008
NEW HORIZONS

What about branching?

Not all ALUs do useful work!
Worst case: 1/8 performance
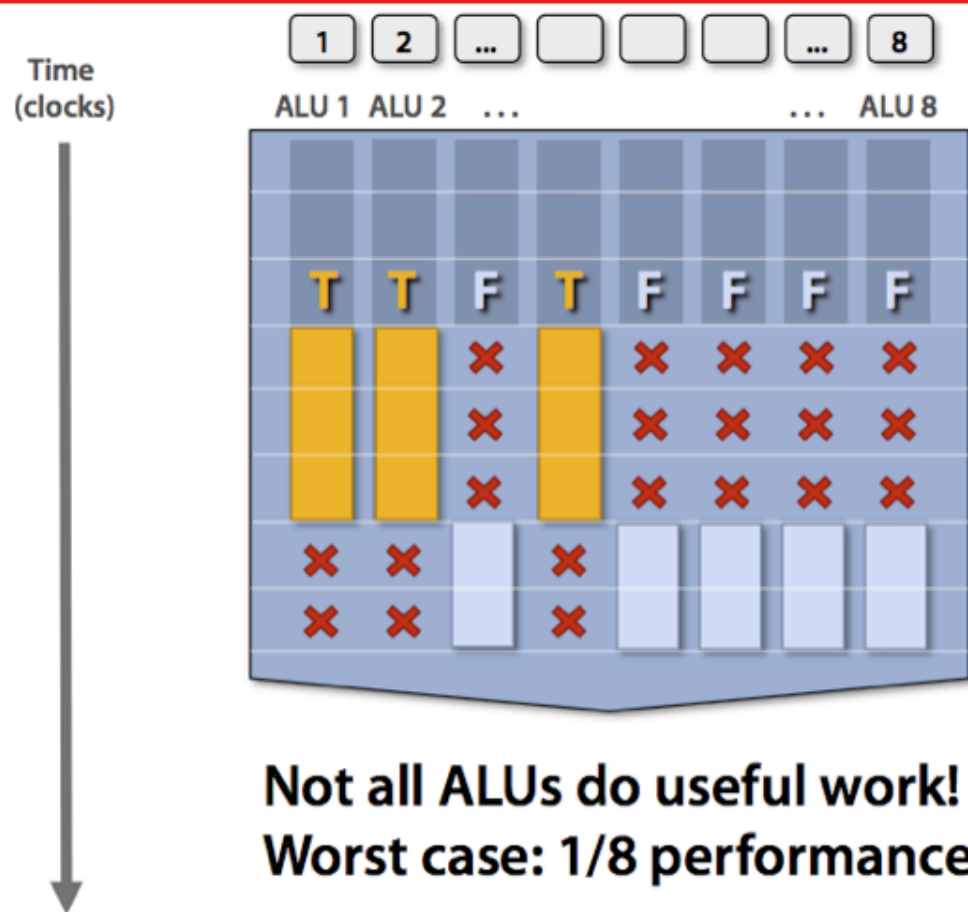
```
<unconditional
 shader code>

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

<resume unconditional
 shader code>
```

What about branching?

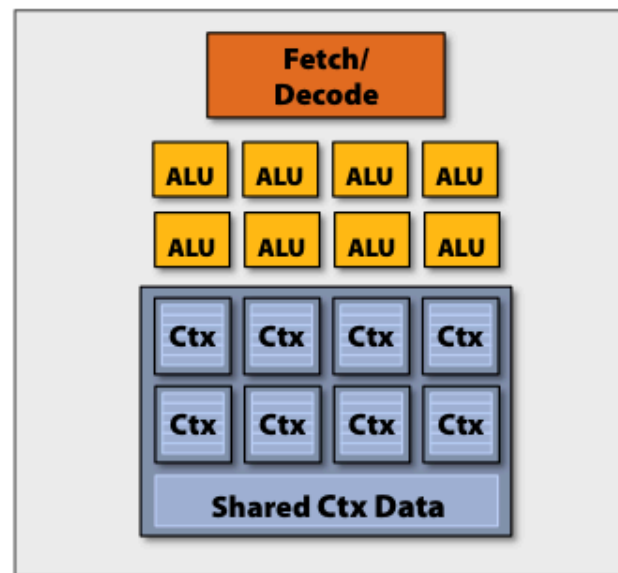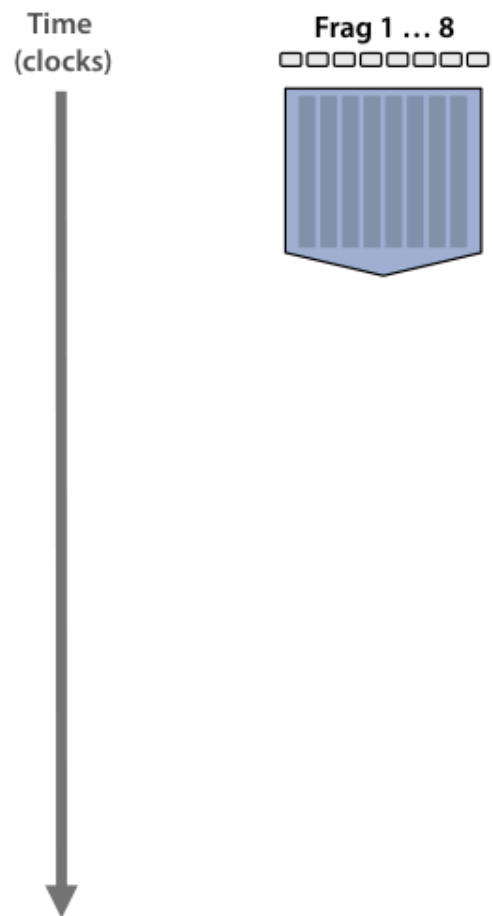adapted from Kayvon Fatahalian's SIGGRAPH'08 talk
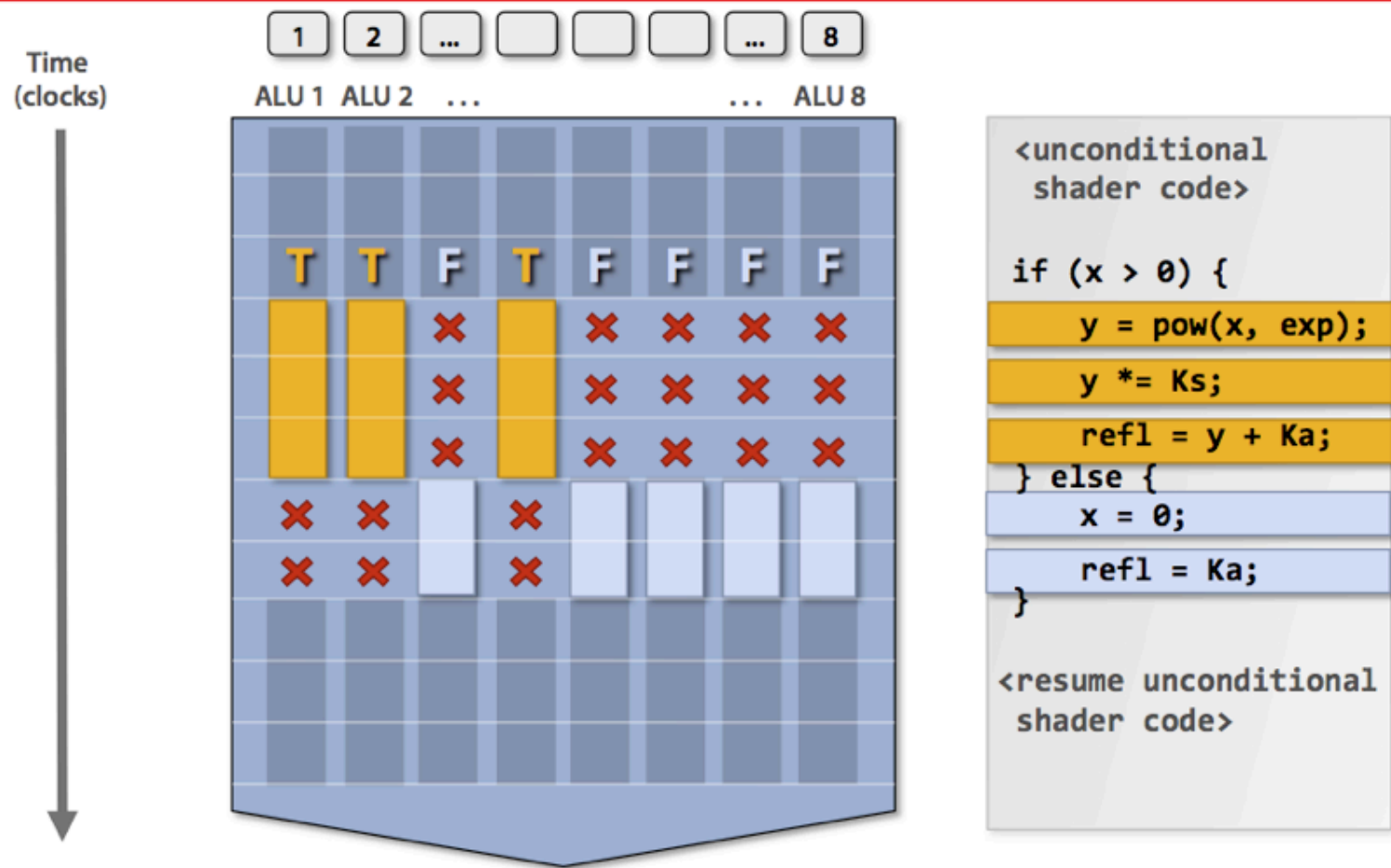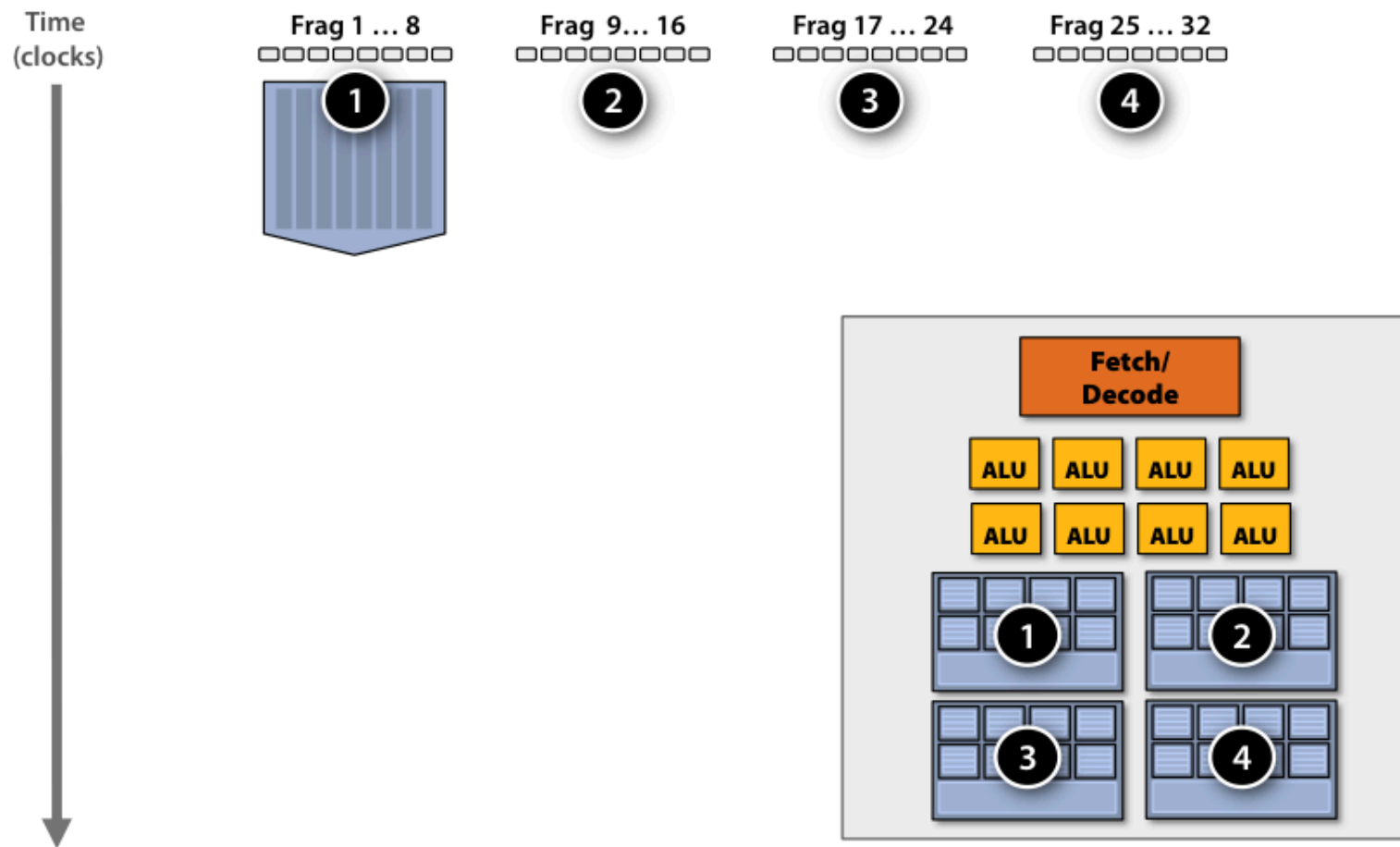
How to handle stalls?

- Memory access latency = 100's to 1000's of cycles
  - Stalls occur when a core cannot run the next instruction

- GPUs don't have the large / fancy caches and logic that helps avoid stall because of a dependency on a previous operation.

- But we have **LOTS** of independent fragments.
  - Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

SIGGRAPHASIA2008
NEW HORIZONS

Hiding Memory Stalls

Time
(clocks)

Frag 1 ... 8

Fetch/
Decode

ALU ALU ALU ALU

ALU ALU ALU ALU

Ctx Ctx Ctx Ctx

Ctx Ctx Ctx Ctx

Shared Ctx Data

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

SIGGRAPHASIA2008
NEW HORIZONS

Hiding Memory Stalls

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

Hiding Memory Stalls

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

Hiding Memory Stalls

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

Hiding Memory Stalls

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

Throughput computing

adapted from Kayvon Fatahalian's SIGGRAPH'08 talk

Latency Hiding with "Thread Warps"

- Warp: A set of threads that execute the same instruction (on different data elements)

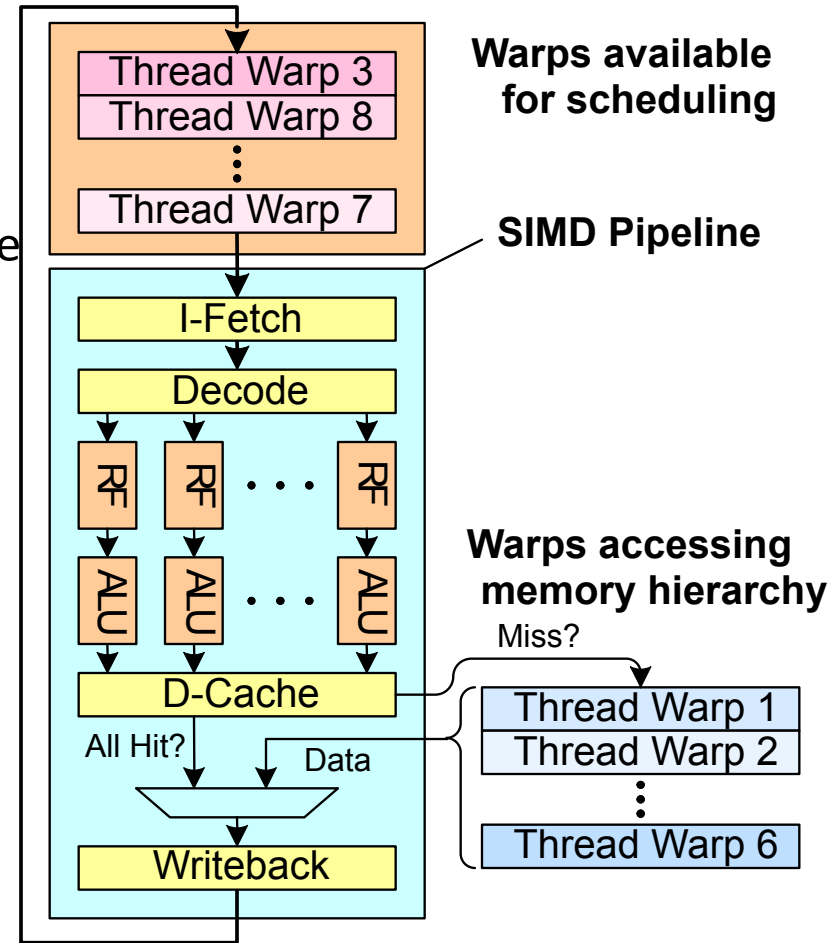- Fine-grained multithreading

  ❐ One instruction per thread in pipeline at a time (No branch prediction)

  ❐ Interleave warp execution to hide latencies

- Register values of all threads stay in register file

- No OS context switching

- Memory latency hiding

  ❐ Graphics has millions of pixels

Warps available for scheduling

Thread Warp 3
Thread Warp 8
⋮
Thread Warp 7

SIMD Pipeline

I-Fetch

Decode

RF RF · · · RF

ALU ALU · · · ALU

D-Cache

Miss?

Warps accessing memory hierarchy

All Hit?   Data

Thread Warp 1
Thread Warp 2
⋮
Thread Warp 6

Writeback

Slide credit: Tor Aamodt

Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - ❐ Lock step
  - ❐ Programming model is SIMD (no threads)
  - ❐ ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - ❐ Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - ❍ Enables memory and branch latency tolerance
  - ❐ ISA is scalar → vector instructions formed dynamically

CUDA In One Slide
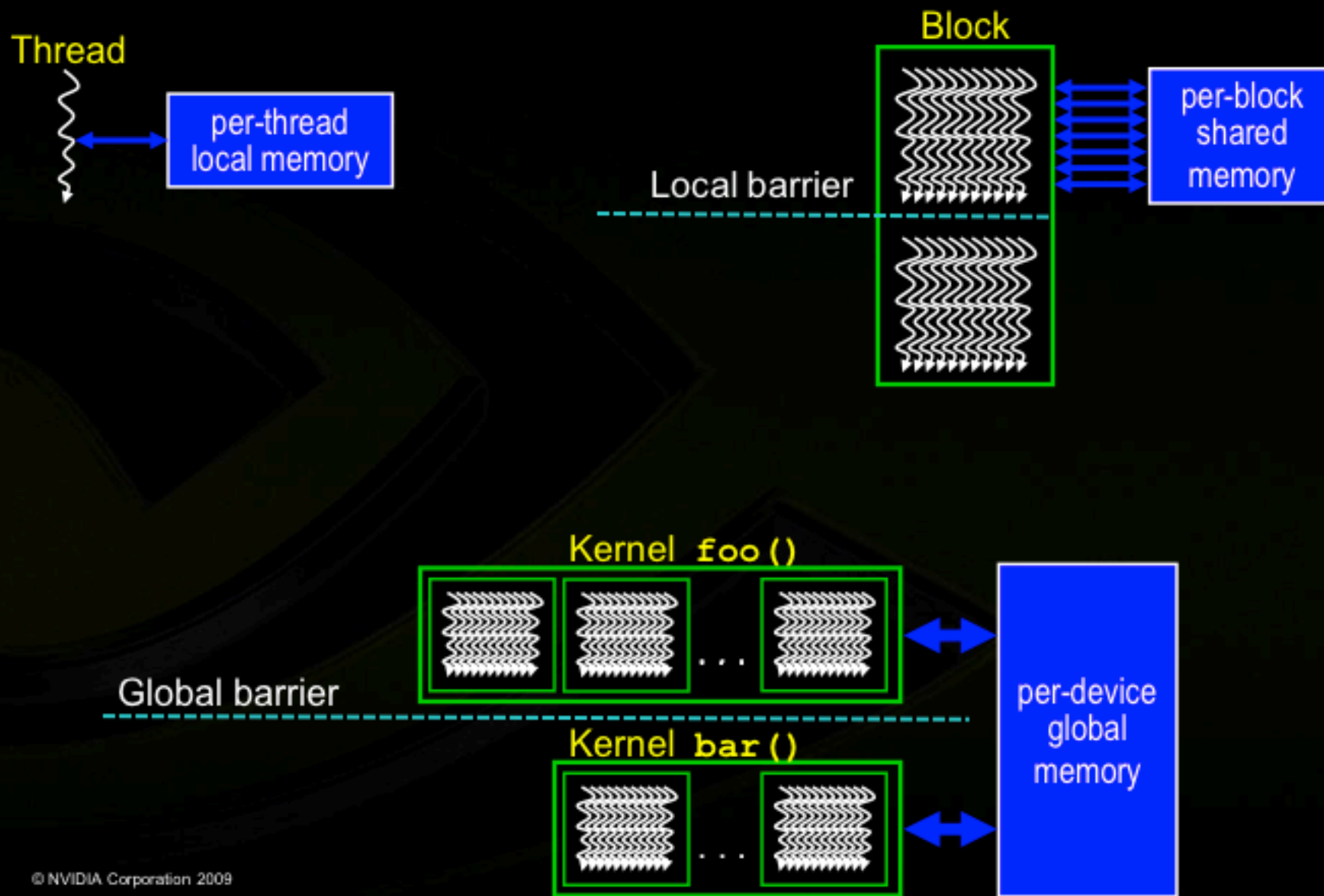
CUDA Devices and Threads

- A compute device
  - ❏ Is a coprocessor to the CPU or host
  - ❏ Has its own DRAM (device memory)
  - ❏ Runs many threads in parallel
  - ❏ Is typically a GPU but can also be another type of parallel processing device

- Data-parallel portions of an application are expressed as device kernels which run on many threads

- Differences between GPU and CPU threads
  - ❏ GPU threads are extremely lightweight
    - ❍ Very little creation overhead
  - ❏ **GPU needs 1000s of threads for full efficiency**
    - ❍ **Multi-core CPU needs (relatively) only a few**

Thread Batching: Grids and Blocks

- A kernel is executed as a
  grid of thread blocks

  - ❐ All threads share data memory space

- A thread block is a batch of threads that
  can cooperate with each other by:

  - ❐ Synchronizing their execution
    - ○ For hazard-free shared memory accesses
  - ❐ Efficiently sharing data through a low
    latency shared memory

- Two threads from two different blocks
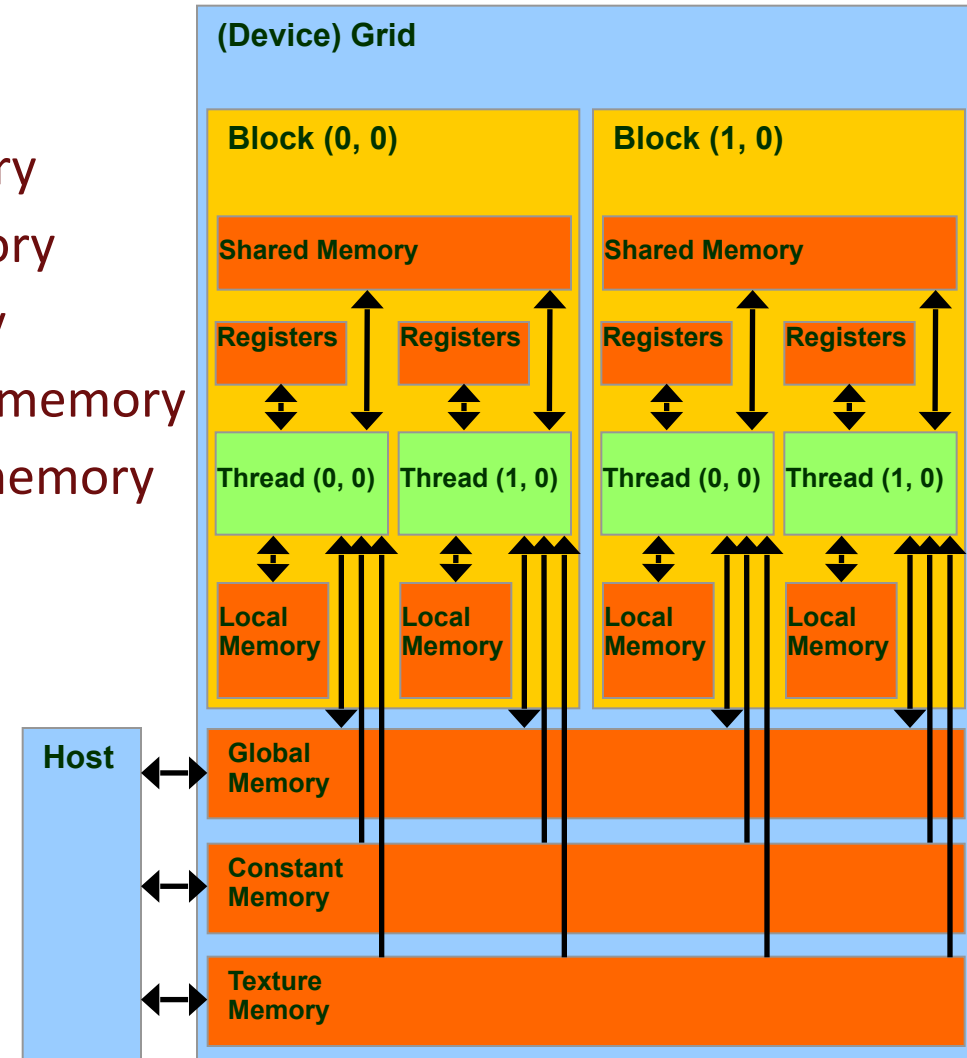 cannot cooperate
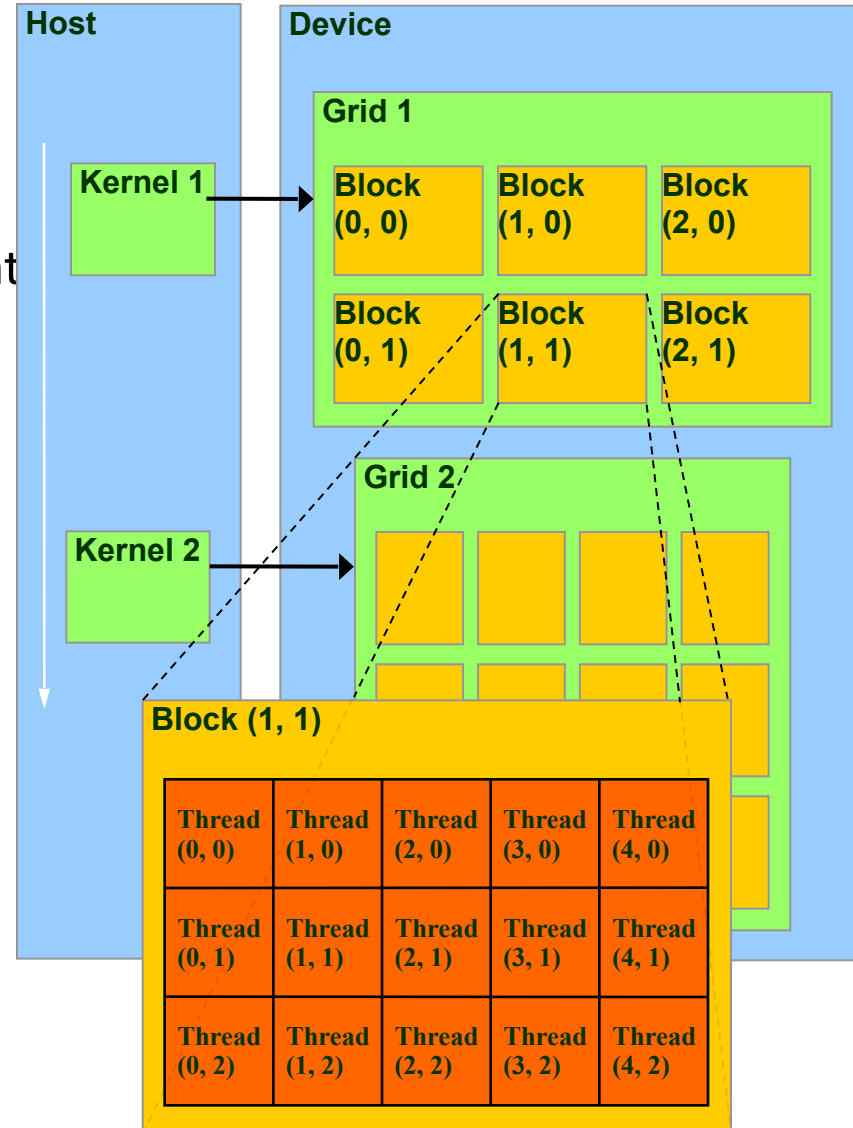
Execution Model

- Each thread block is executed by a single multiprocessor
  - ❏ Synchronized using shared memory

- Many thread blocks are assigned to a single multiprocessor
  - ❏ Executed concurrently in a time-sharing fashion
  - ❏ Keep GPU as busy as possible

- Running many threads in parallel can hide DRAM memory latency
  - ❏ Global memory access : 2~300 cycles

CUDA Device Memory Space Overview

- Each thread can:
  - ❑ R/W per-thread registers
  - ❑ R/W per-thread local memory
  - ❑ R/W per-block shared memory
  - ❑ R/W per-grid global memory
  - ❑ Read only per-grid constant memory
  - ❑ Read only per-grid texture memory

- The host can R/W global, constant, and texture memories

(Device) Grid

Block (0, 0)

Shared Memory

Registers   Registers

Thread (0, 0)   Thread (1, 0)

Local Memory   Local Memory

Block (1, 0)

Shared Memory

Registers   Registers

Thread (0, 0)   Thread (1, 0)

Local Memory   Local Memory

Host

Global Memory

Constant Memory

Texture Memory

Example: Vector Addition Kernel

```
// Pair-wise addition of vector elements
// One thread per addition

__global__ void
vectorAdd(float* iA, float* iB, float* oC)
{
    int idx = threadIdx.x
        + blockDim.x * blockId.x;
    oC[idx] = iA[idx] + iB[idx];
}
```

Courtesy NVIDIA

Example: Vector Addition Host Code

```
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
// ... initalize h_A and h_B

// allocate device memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float) );
cudaMalloc( (void**) &d_B, N * sizeof(float) );
cudaMalloc( (void**) &d_C, N * sizeof(float) );

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
  cudaMemcpyHostToDevice );
cudaMemcpy( d_B, h_B, N * sizeof(float),
  cudaMemcpyHostToDevice );

// execute the kernel on N/256 blocks of 256 threads each
vectorAdd<<< N/256, 256>>>( d_A, d_B, d_C);
```

CUDA-Strengths

- (Relatively) easy to program (small learning curve)

- Success with several complex applications
  - At least 7X faster than CPU stand-alone implementations

- Allows us to read and write data at any location in the device memory

- More fast memory close to the processors (registers + shared memory)

CUDA-Limitations

- Some hardwired graphic components are hidden

- Better tools are needed
    - Profiling
    - Memory blocking and layout
    - Binary Translation

- Difficult to find optimal values for CUDA execution parameters
    - Number of thread per block
    - Dimension and orientation of blocks and grid
    - Use of on-chip memory resources including registers and shared memory

- Working with GPUs is an active area of research