

An aerial photograph of the University of Colorado Boulder campus. The foreground is filled with trees showing vibrant autumn foliage in shades of yellow, orange, and red. In the middle ground, several university buildings are visible, including a prominent red brick building with a tall, ornate clock tower. The background features a large, rugged mountain with a mix of green forest and rocky, light-colored slopes under a clear blue sky.

MLOps, ML Hardware

CSCI 5942

Lecture 3



College of Engineering & Applied Science

UNIVERSITY OF COLORADO **BOULDER**

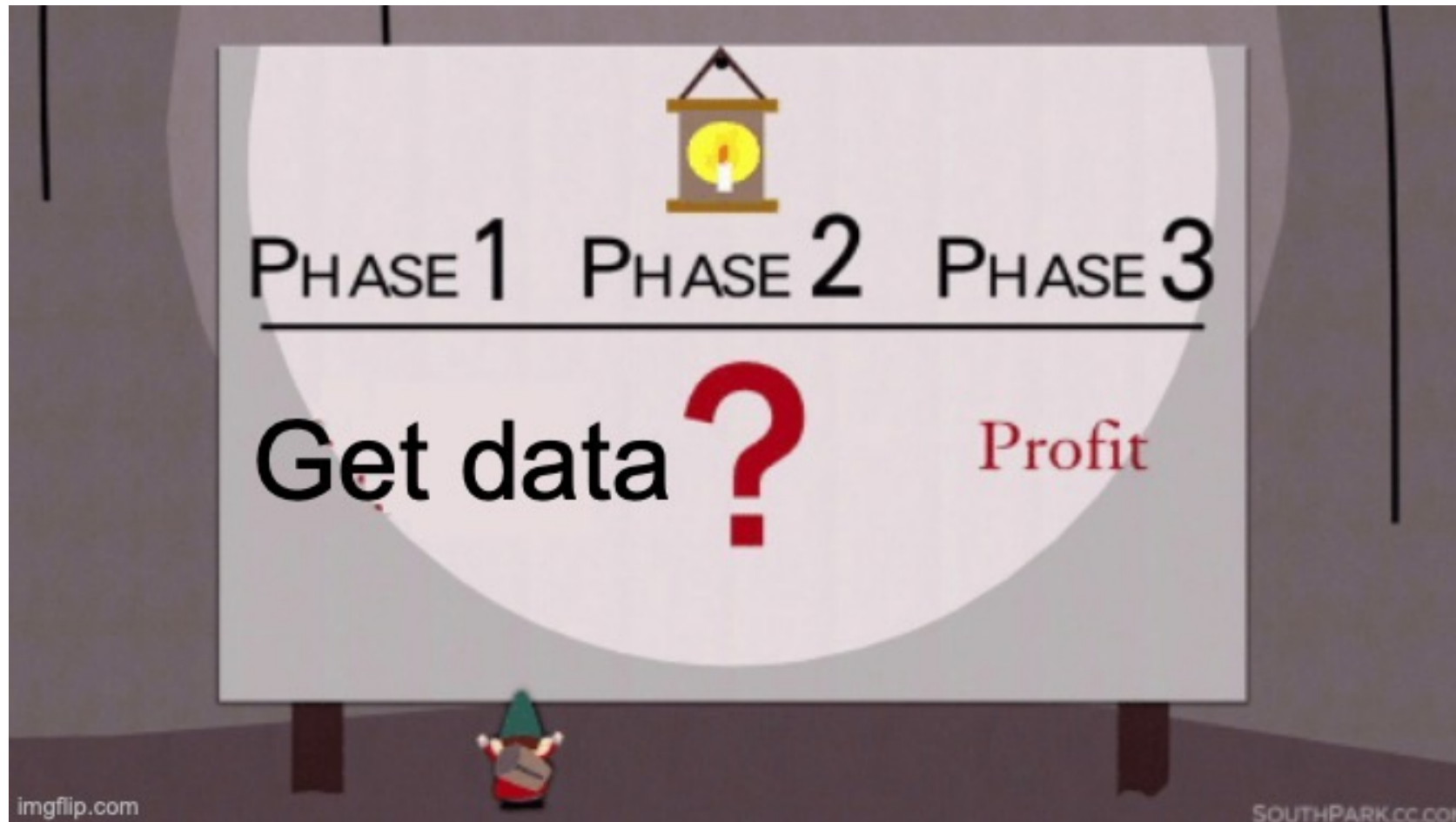
Announcements

- A1 due Thursday at midnight
 - In-class “quiz” based on A1 on Thursday during lecture, so please try to complete the assignment before class!

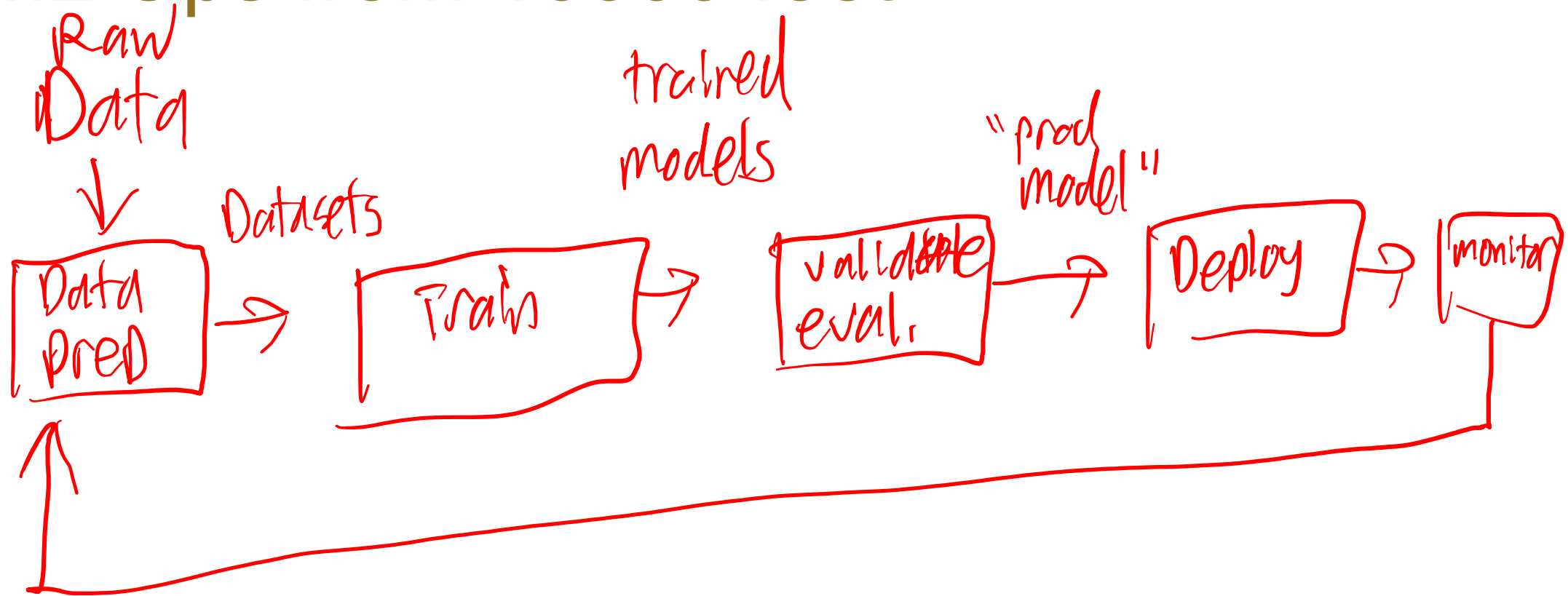
Agenda

- Overview of MLOps* as a field
 - Reading: “We Have No Idea How Models will Behave in Production until Production”: How Engineers Operationalize Machine Learning (Proc. of the ACM on Human Computer Interaction, 2024)
- Crash course on HW acceleration, GPUs
 - Reading: Roofline: An insightful Visual Performance model for multicore Architectures (Communications of the ACM, 2009)

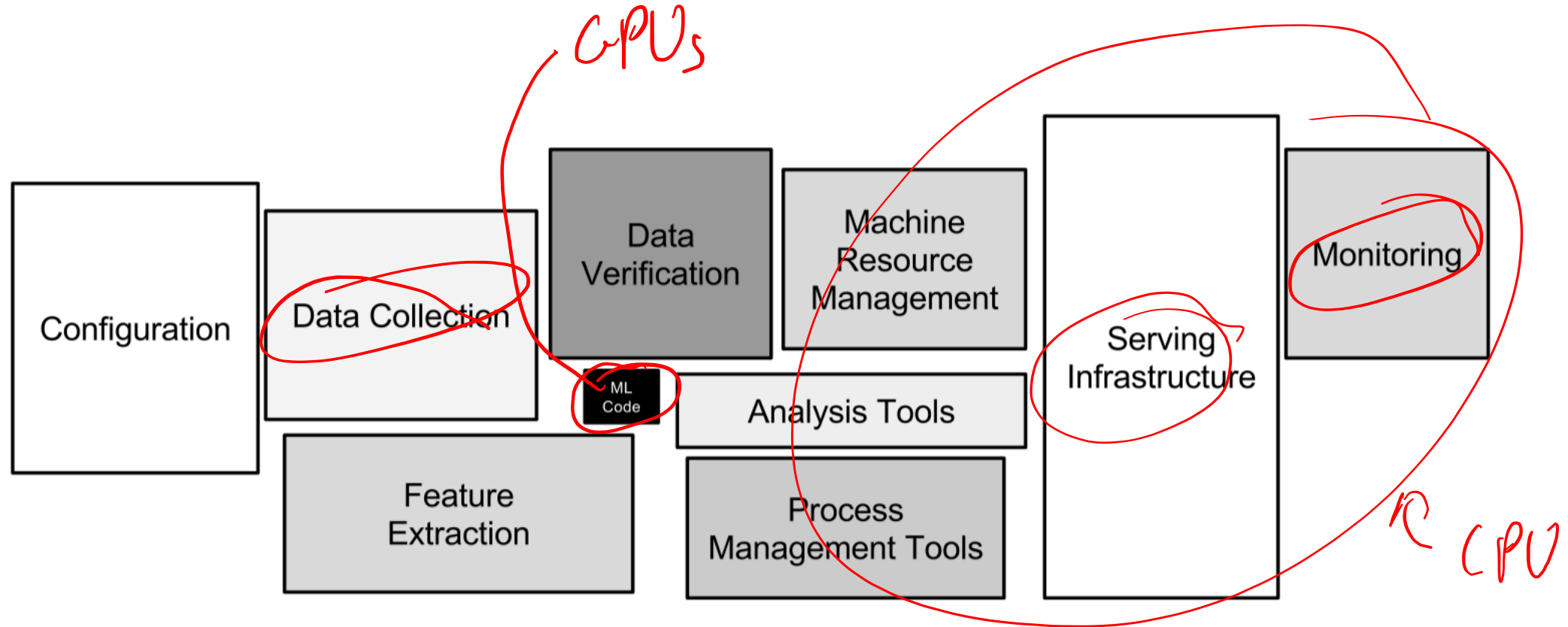
ML Ops from 10000 feet



ML Ops from 10000 feet

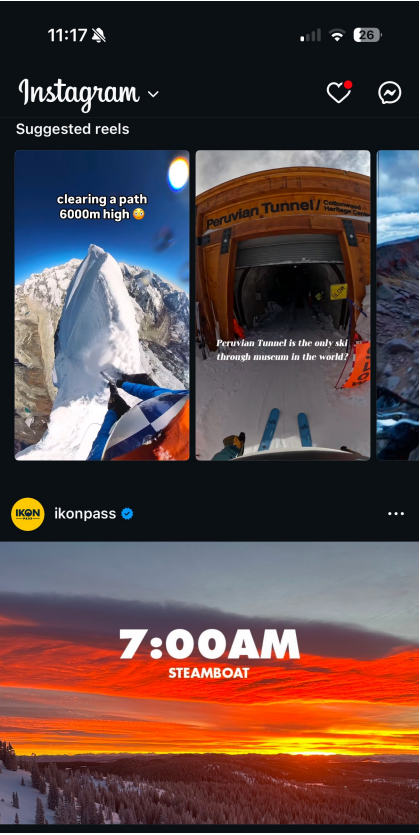
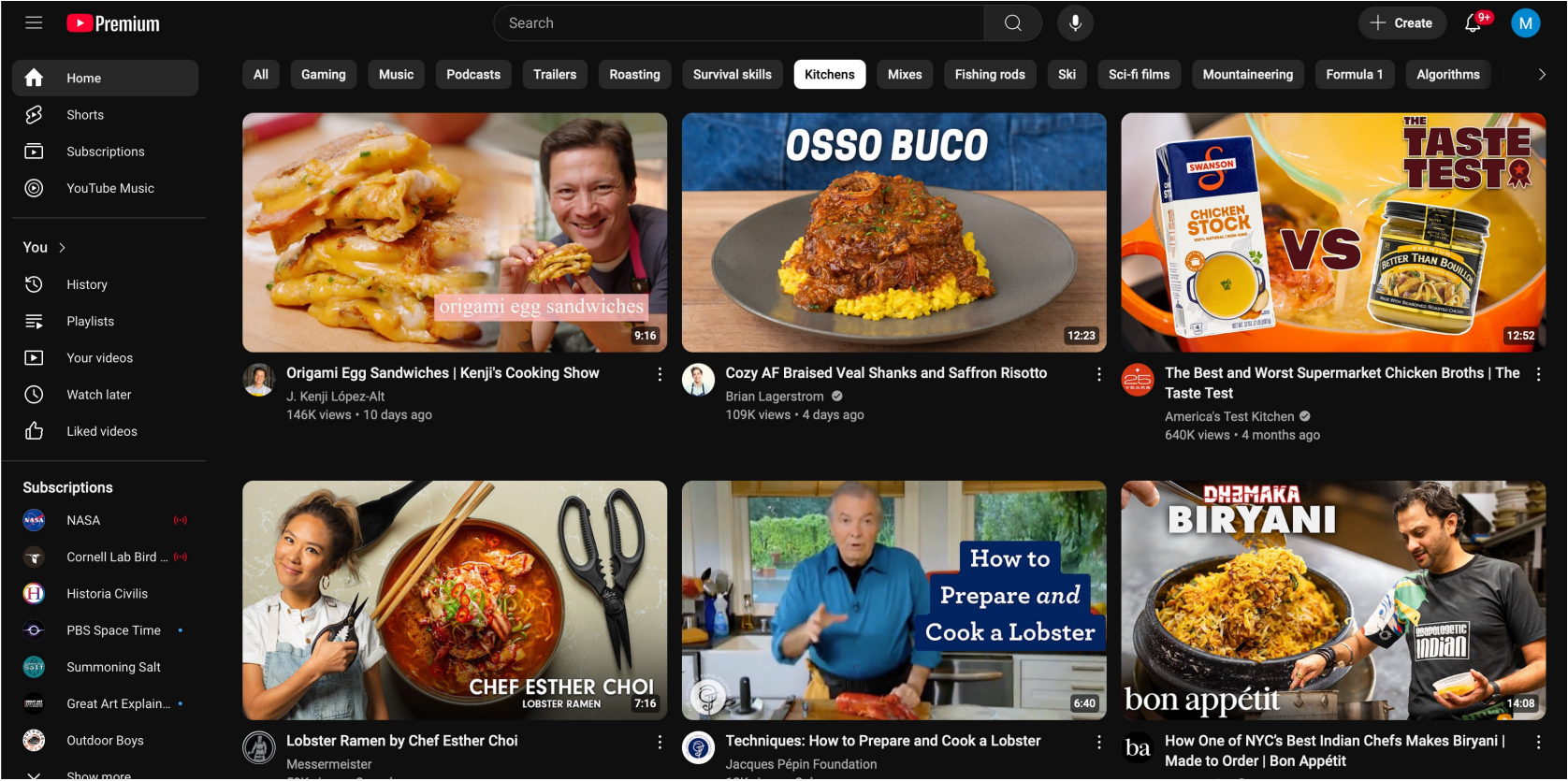


ML Systems Support the *AI Lifecycle*



Only a small fraction of real-world ML systems is composed of ML code

End-to-End example

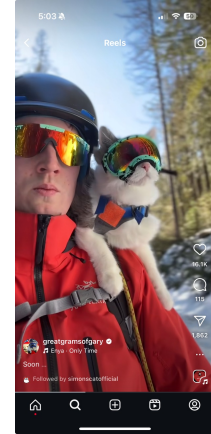


Generate Data

- Step 1: Capture raw interactions —————> Ellie liked

Who! App developer

How! Segment.io, Snowplow, RudderStack, ...



Generate Data

- Step 1: Capture raw interactions
- Step 2: Ingest distributed events

billions of devices



user 153 liked post_id 192

...

...



user 91 liked post_id 52

~~How?~~ Who? Data Platform Engineer

How? Kafka, Kinesis, Pub/Sub

Generate Data

- Step 1: Capture raw interactions
- Step 2: Ingest distributed events
- Step 3: Transform into features/training samples



```
sample: {  
  user: 153,  
  rec_post: 15,  
  label: like,  
  features: {  
    last_n_liked: [192, ...],  
    last_n_comment: ...,  
    ...}  
}
```

Who! Data engineer

How! Spark & Databricks

Generate Data

- Step 1: Capture raw interactions
- Step 2: Ingest distributed events
- Step 3: Transform into features/training samples
- Step 4: Store training data



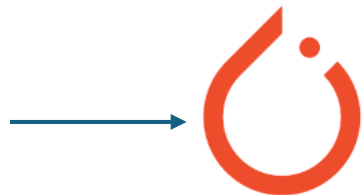
User	Features	Rec. Item	Like?
Alice	A: [31, 55, ...], B: [11, 13, ...], ...	512	0
Bob	A: [23, 11, ...], B: [98, 5, ...], ...	83	1
Carol	A: [17, 43, ...], B: [1, 7, ...], ...	15	1

Generate Data

- Step 1: Capture raw interactions
- Step 2: Ingest distributed events
- Step 3: Transform into features/training samples
- Step 4: Store training data
- Step 5: Preprocess and ingest data

Who! ML Engineer
How! DataLoader

User	Features	Rec. Item	Like?
Alice	A: [31, 55, ...], B: [11, 13, ...], ...	512	0
Bob	A: [23, 11, ...], B: [98, 5, ...], ...	83	1
Carol	A: [17, 43, ...], B: [1, 7, ...], ...	15	1



Train Model

> python ./train.py ?...

- Workload:

- 100s of model types
- 1000s of training jobs
- 100000s of GPUs
- ... all of the time

one for each "task" / "surface"
→ Hyperparam Search

Evaluate Model

- Thousands of model variants, which one to deploy?

Offline

Accuracy
Loss
F1 score
...

every model

top performers

Online

How well the model does in prod.

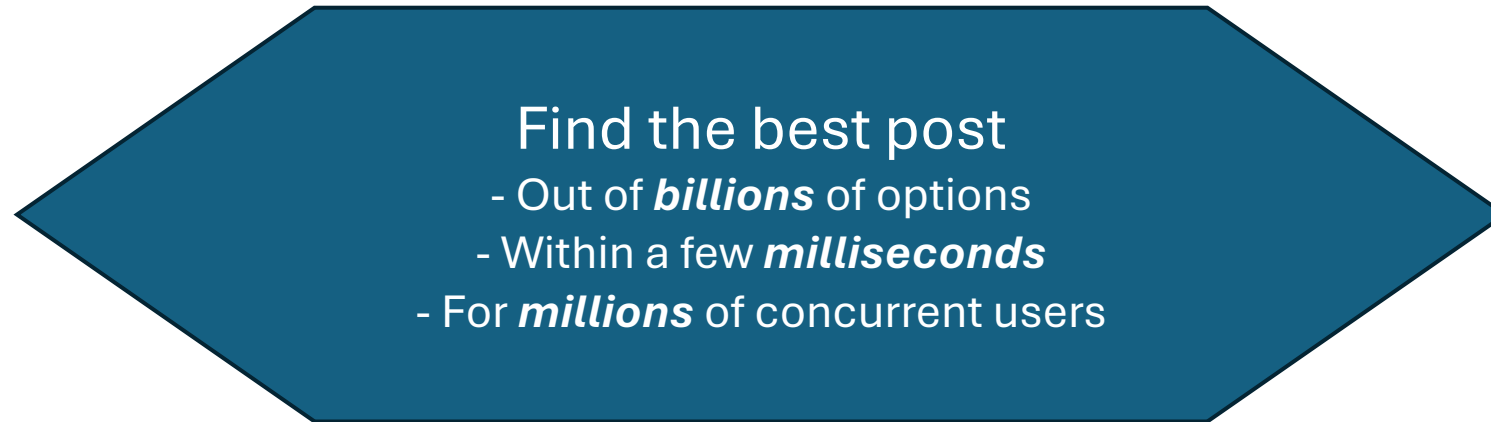
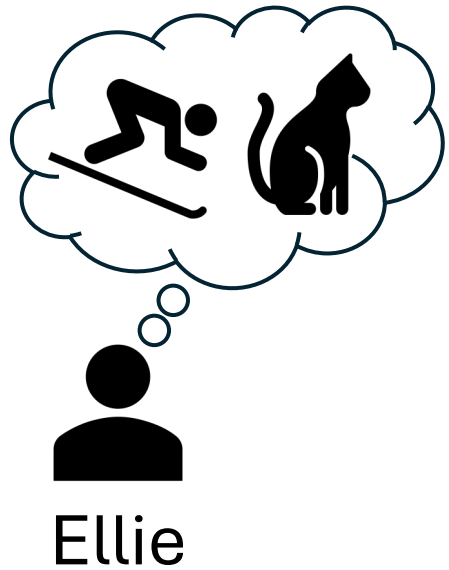
A/B testing

Replay/shadowing

Gradual Rollout

↳ 0.1% → 1% → 100%

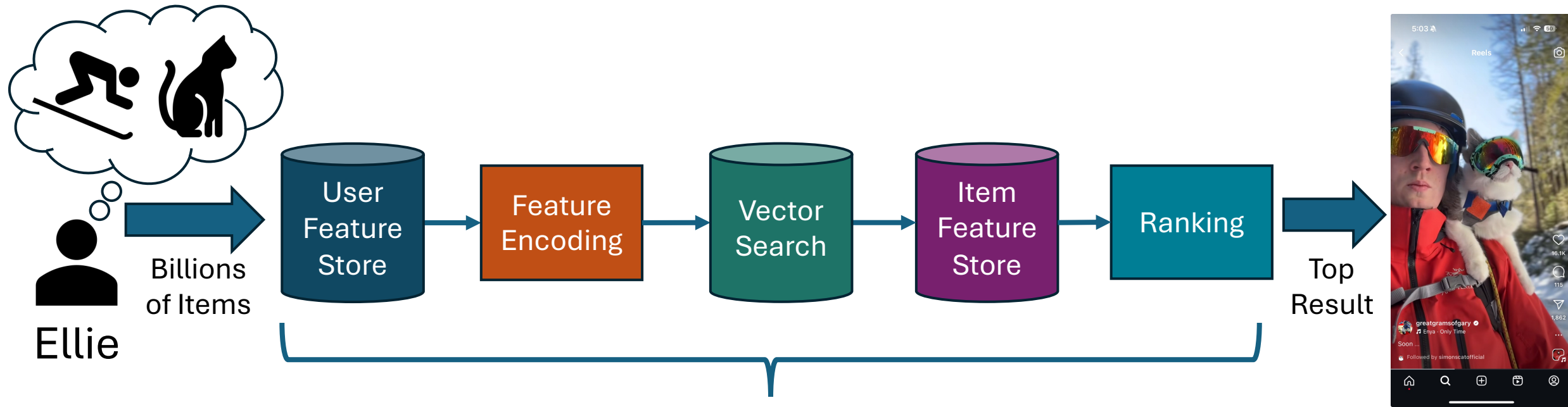
Deploy Model



Inference System

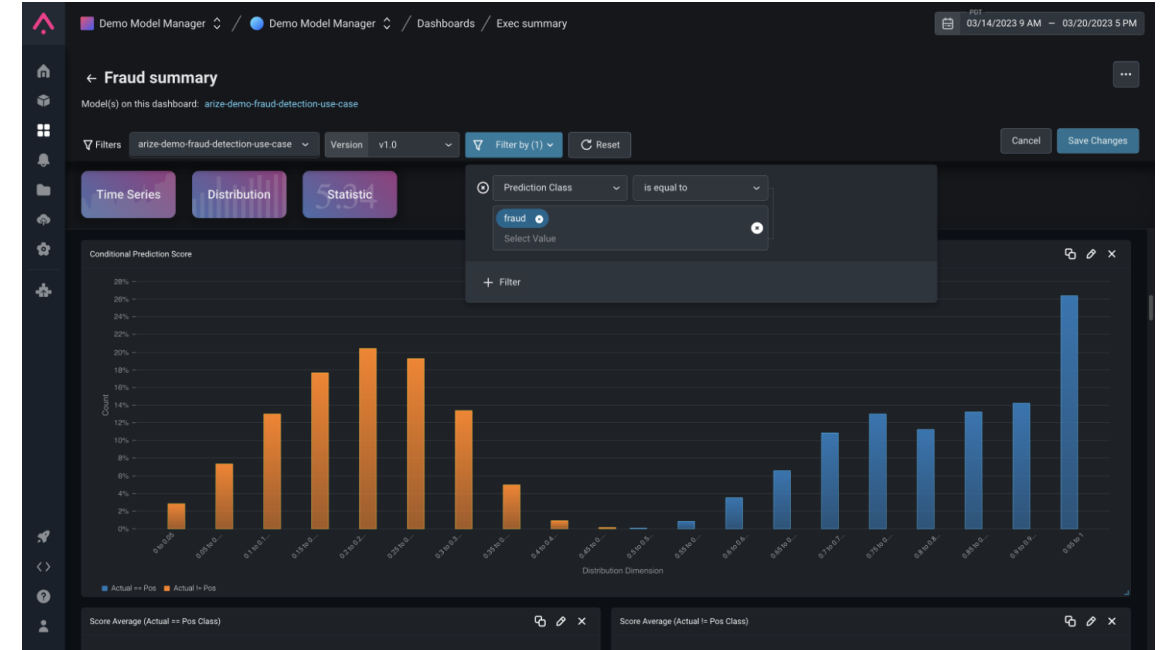
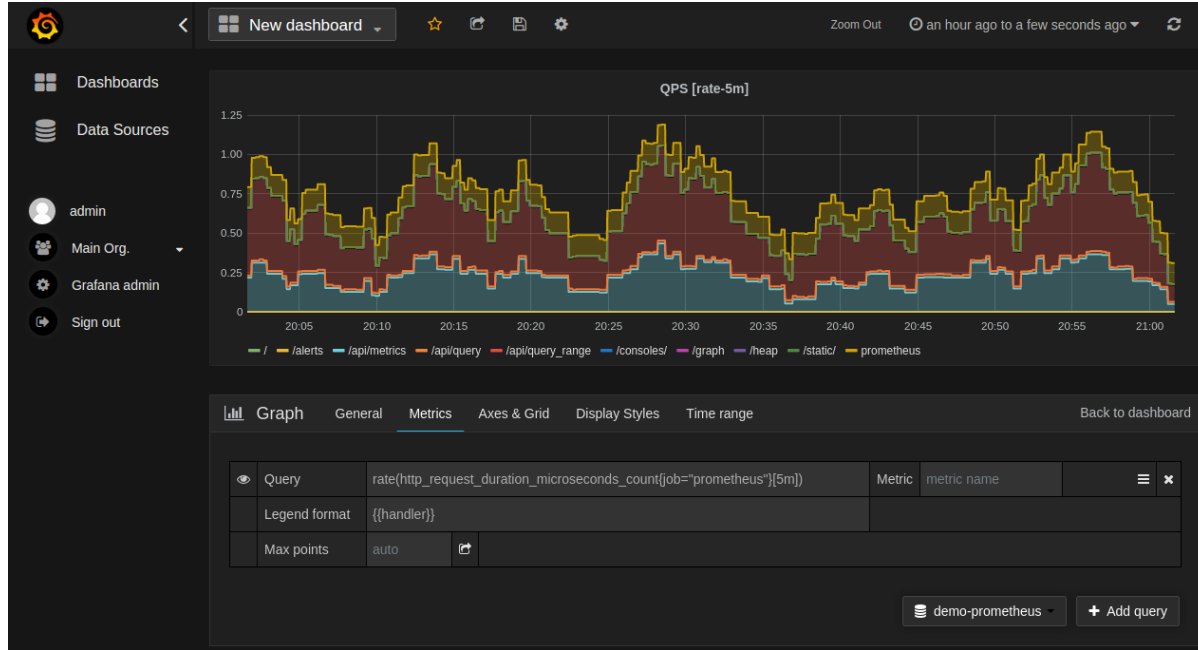


Deploy Model



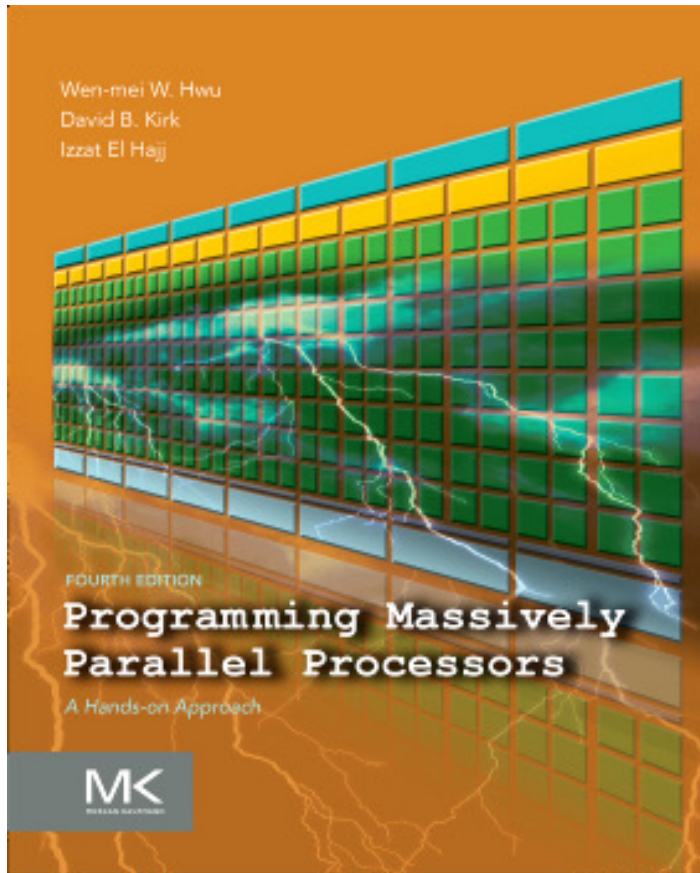
Specialized **data** (e.g., feature stores, vector DBs)
and **inference** (e.g., embedding models, DLRMs) systems

Monitor Model



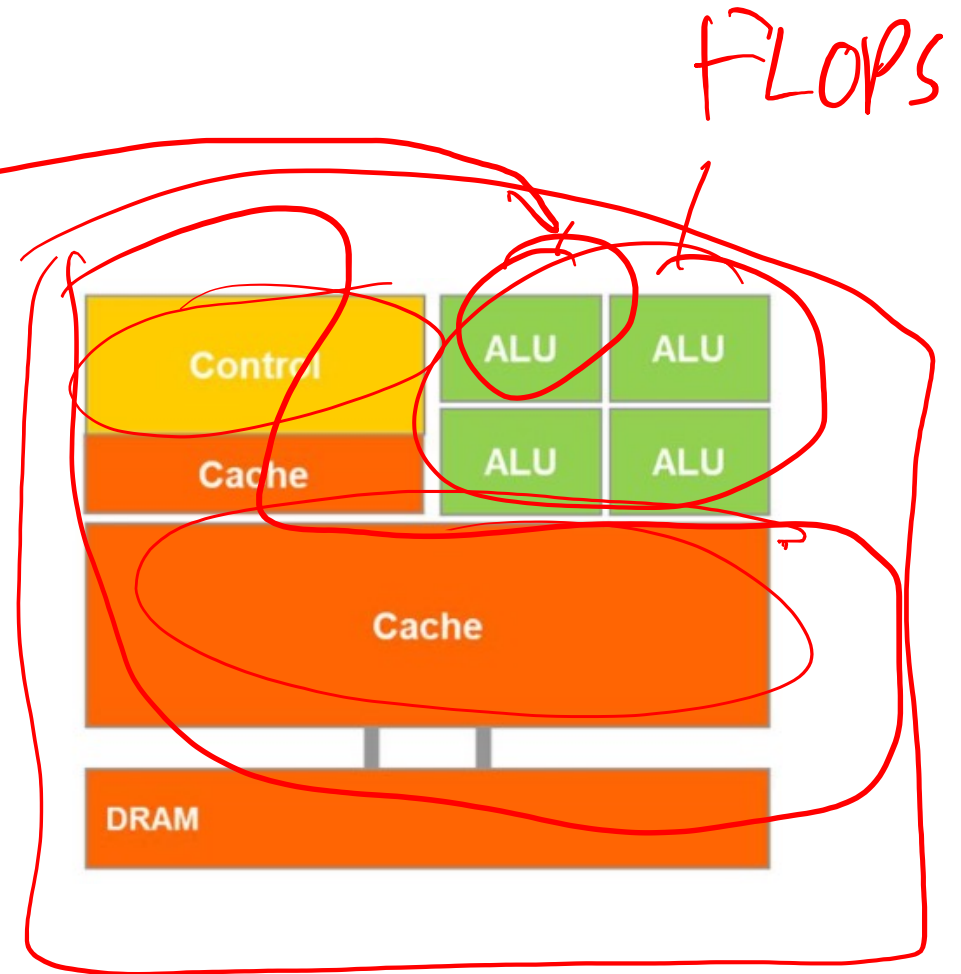
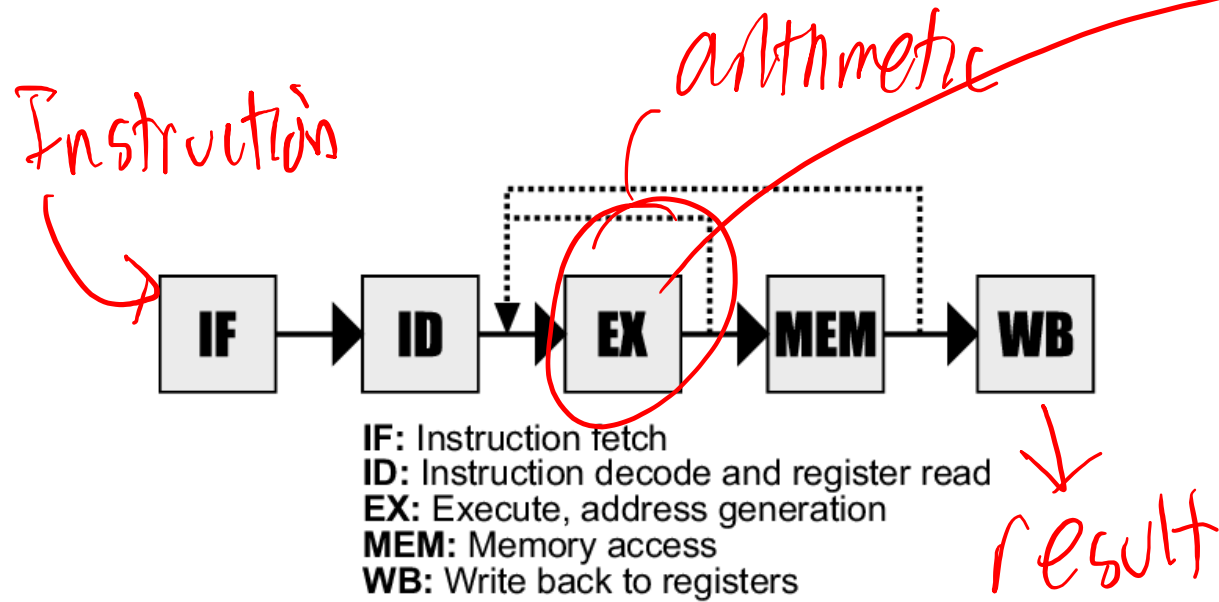
System perf: Latency, throughput,
Model Perf: Data drift, downstream impact

PMPP



- Digital Copy available through CU Libraries
- Chapters 1-6 is essential reading to understand GPU programming

CPU Architecture



Latency - Optimized: CPUs
↳ Gen purpose programs

Amdahl's Law

$$\text{Speedup} = \frac{\text{old exec. time}}{\text{new exec. time}} = \frac{10s}{1s} = 10x$$

$$S = \frac{1}{(1 - p) + \frac{p}{n}}$$

$p = .30, n = 100 \rightarrow S = 1.42x$

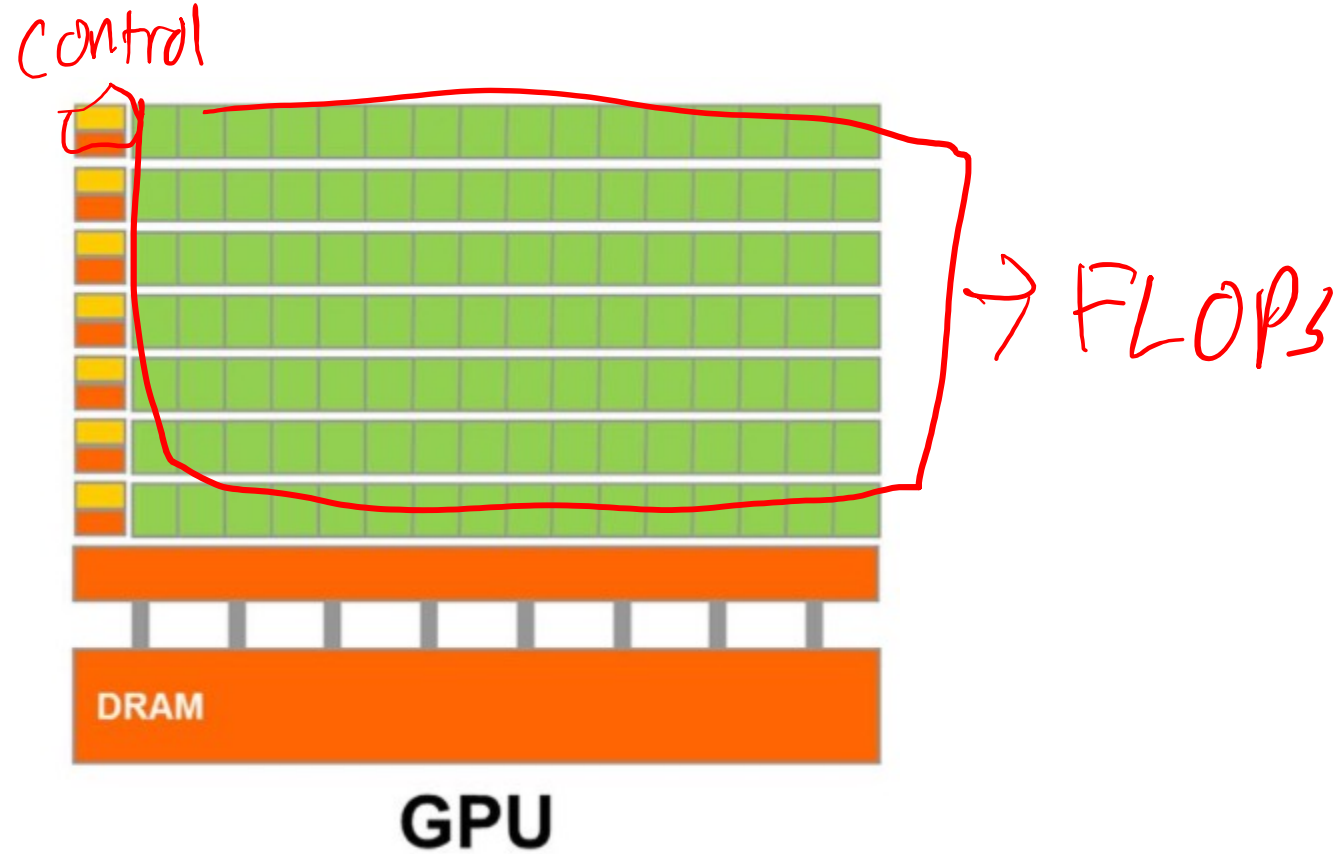
$p = .99, n = 100 \rightarrow S = 50x$

↑
frac. of prog. time spent in
"parallel" portion

← speedup of "parallel" portion

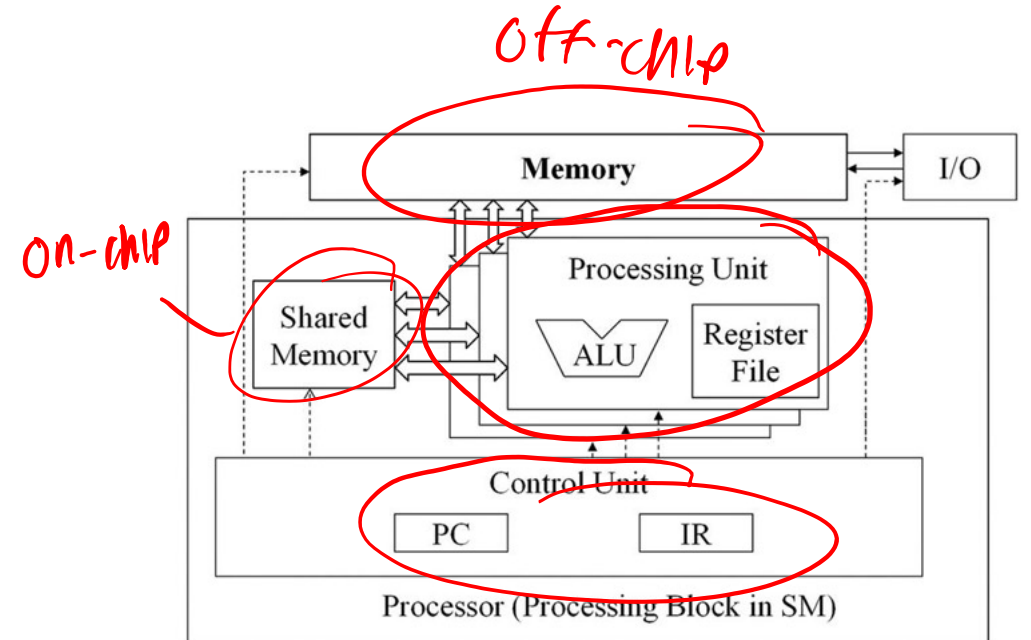
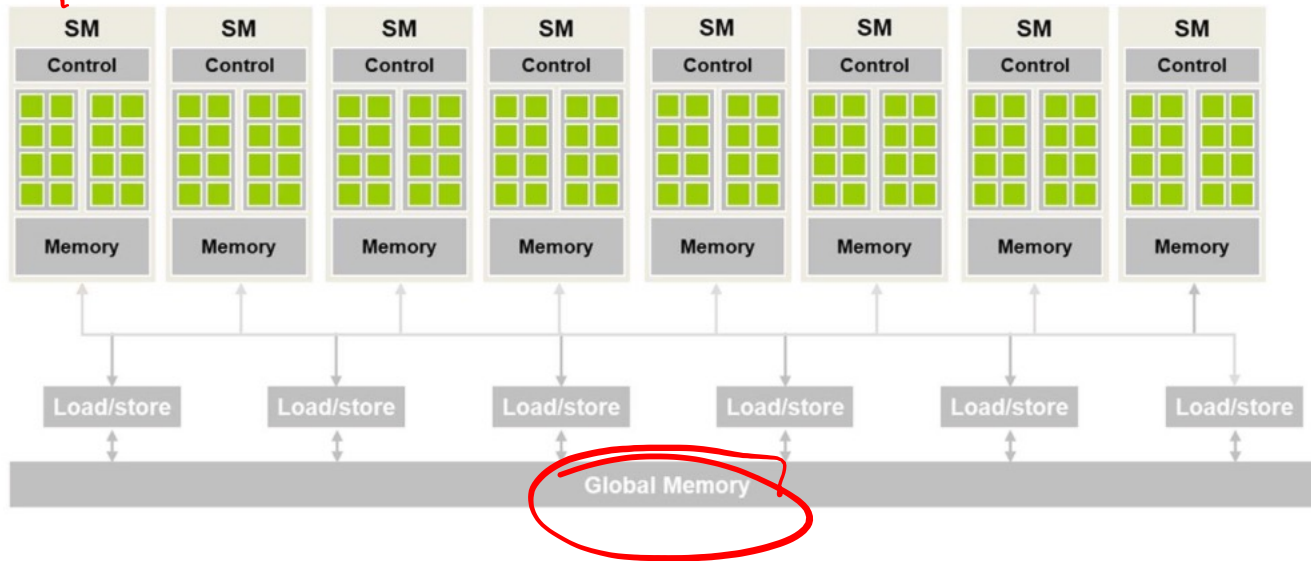
ML, Graphics

Throughput-oriented Hardware



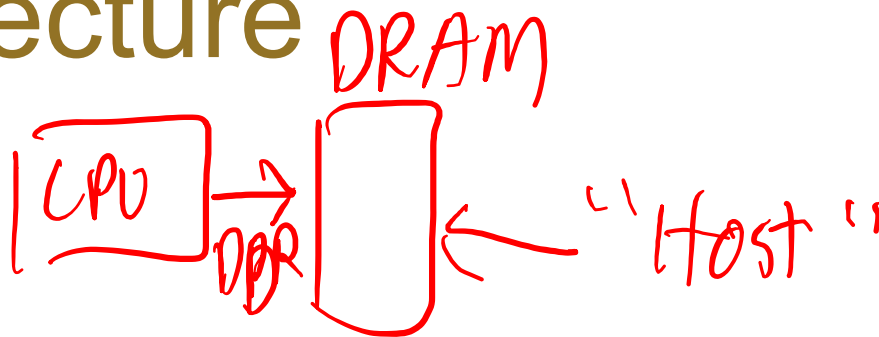
GPU Microarchitecture

Streaming
- multiprocessors



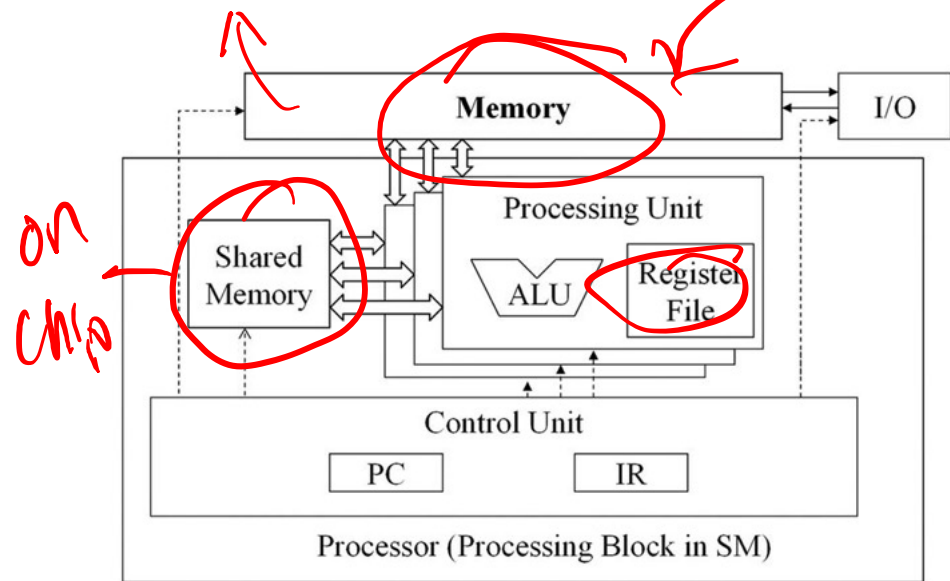
SIMD: Single Instruction Multiple Data

Memory Architecture



Slow $\rightarrow$ fast
Host, Device $\rightarrow$ shared
large
Size

CPU, HBM $\rightarrow$ off chip
PCIe

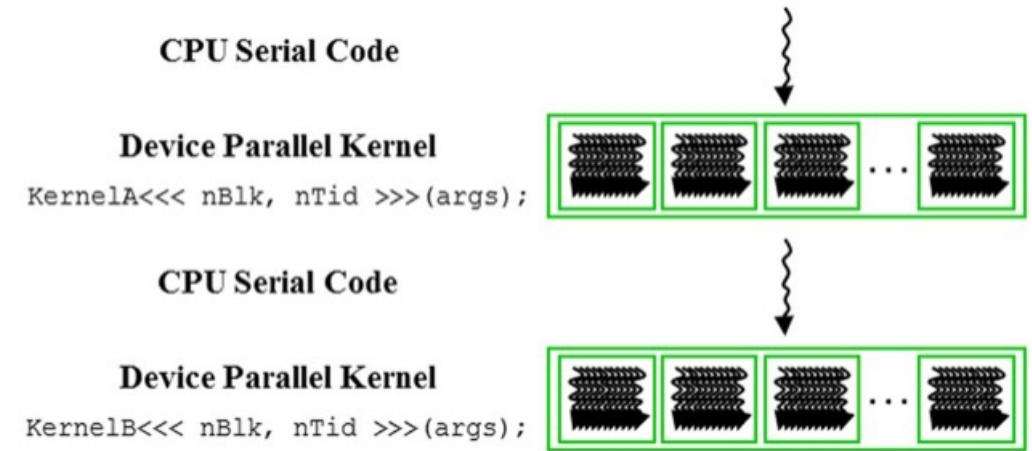


Device

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	<u>Register</u>	Thread	Grid
Automatic array variables	<u>Local</u>	Thread	Grid
<code>__device__ __shared__ int SharedVar;</code>	Shared	Block	Grid
<code>__device__ int GlobalVar;</code>	Global	Grid	Application
<code>__device__ __constant__ int ConstVar;</code>	Constant	Grid	Application

CUDA Programming

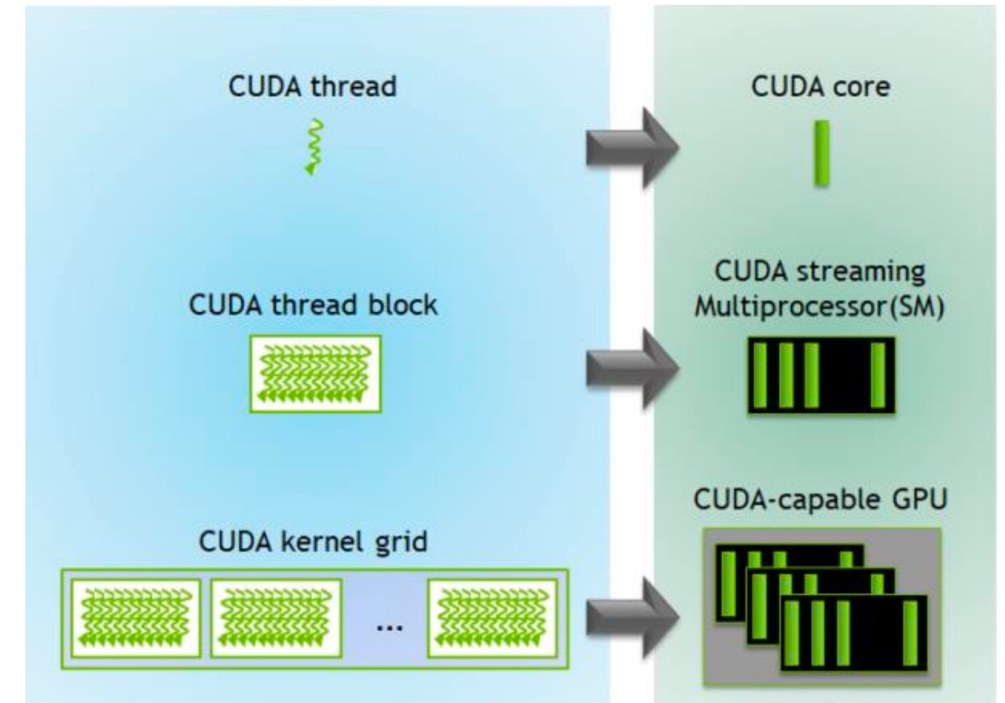
- Mixes *Host* (CPU) and *Device* (GPU) code
- CPU code invokes GPU processing by launching *kernels*
 - <<<...>>> specify kernel config
- CPU transfers data to/from device memory
 - cudaMalloc(), cudaMemcpy(), cudaFree()



Qualifier Keyword	Callable From	Executed On	Executed By
<u>__host__</u> (default)	Host	Host	Caller host thread
<u>__global__</u>	Host (or Device)	Device	New grid of device threads
<u>__device__</u>	Device	Device	Caller device thread

Kernel Execution

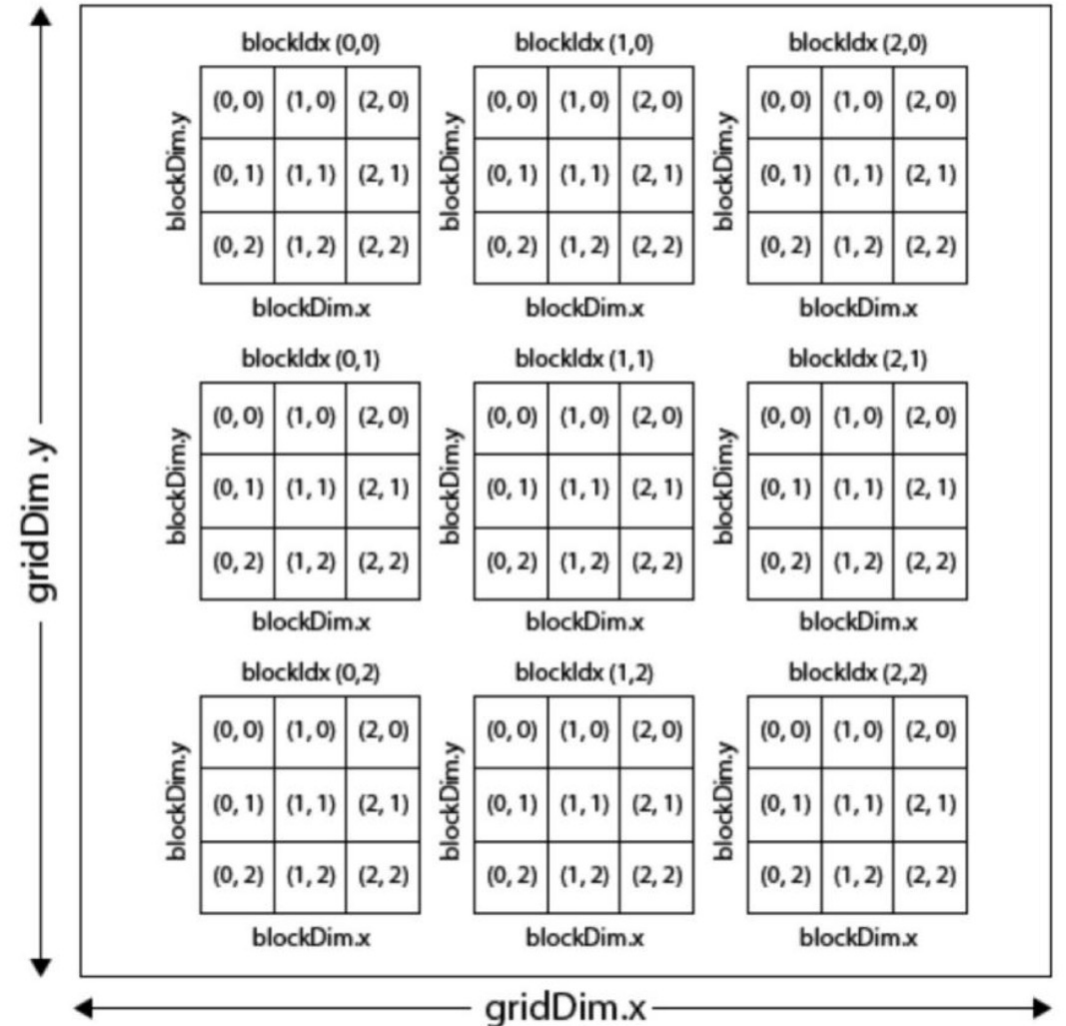
- **Thread:** Simple sequential unit of execution
 - Process a small unit of data (e.g., one pixel)
- **Blocks:** Group of threads that share memory
 - Scheduled onto the same SM
 - Threads in block execute concurrently, use `__syncthreads()`
- **Warps:** 32-thread unit, scheduled by HW
 - Concurrent SIMD execution
- **Grid:** All threads that execute a kernel
 - Blocks can execute in any order relative to other blocks



Thread Organization

(up to) 3D organization of threads per block, blocks per grid

- **threadIdx.<x,y,z>**: Coordinate of thread in block
- **blockDim.<x,y,z>**: Dimension of threads per block
 - Currently, max 1024 threads per block
- **blockIdx.<x,y,z>**: Coordinate of block in grid
- **gridDim.<x,y,z>**: Dimensions of blocks per grid

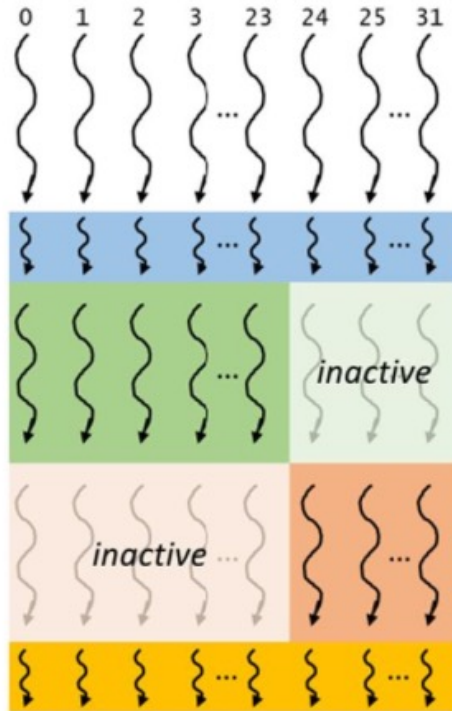


Control Divergence

```

if(threadIdx.x < 24) {
    A
} else {
    B
}
C

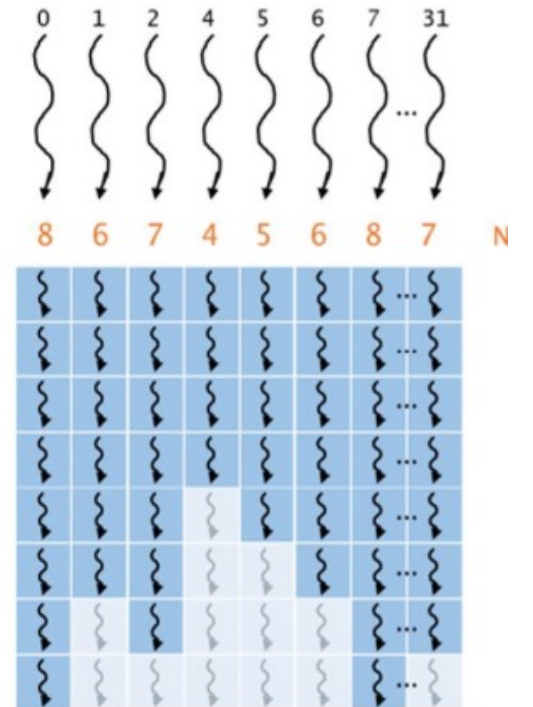
```



```

N = a[threadIdx.x];
for(i = 0; i < N; ++i) {
    A
}

```



Vector Vector Add

```
01  // Compute vector sum  $C_h = A_h + B_h$ 
02  void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
03      for (int i = 0; i < n; ++i) {
04          C_h[i] = A_h[i] + B_h[i];
05      }
06  }
07  int main() {
08      // Memory allocation for arrays A, B, and C
09      // I/O to read A and B, N elements each
10      ...
11      vecAdd(A, B, C, N);
12  }
```


vvadd

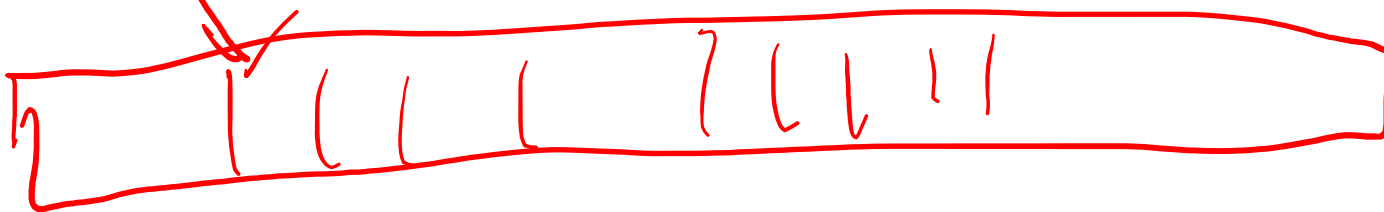
```
01 void vecAdd(float* A, float* B, float* C, int n) {
02     float *A_d, *B_d, *C_d;
03     int size = n * sizeof(float);
04
05     cudaMalloc((void **) &A_d, size);
06     cudaMalloc((void **) &B_d, size);
07     cudaMalloc((void **) &C_d, size);
08
09     cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
10     cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);
11
12     vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
13
14     cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
15
16     cudaFree(A_d);
17     cudaFree(B_d);
18     cudaFree(C_d);
19 }
```

Handwritten annotations:

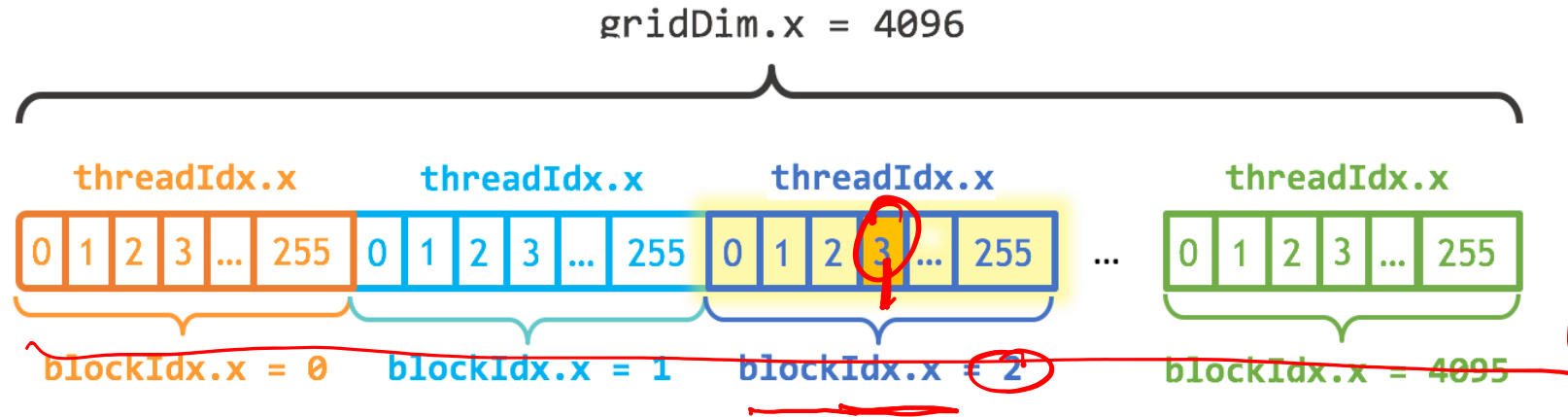
- Input:** A, B (pointing to lines 01 and 02)
- Output:** C (pointing to line 01)
- allocate CPU:** A_d, B_d, C_d (pointing to lines 05, 06, 07)
- D → H:** cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost); (pointing to line 14)

vvadd kernel

```
__global__ void vecAddKernel(float* A, float* B, float* C, int n) {  
    int idx = threadIdx.x + blockDim.x * deviceblockIdx.x; Size  
    if (idx < n) {  
        C[idx] = A[idx] + B[idx];  
    }  
}
```



Thread indexing



$$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

$$\text{index} = (2) * (256) + (3) = 515$$

Hardware bottlenecks

1,000 H100s $\times$ 30 days $\times$ 40% MFU?

MFU is model FLOPs utilization: the fraction of the chips' peak rate the run actually reaches.

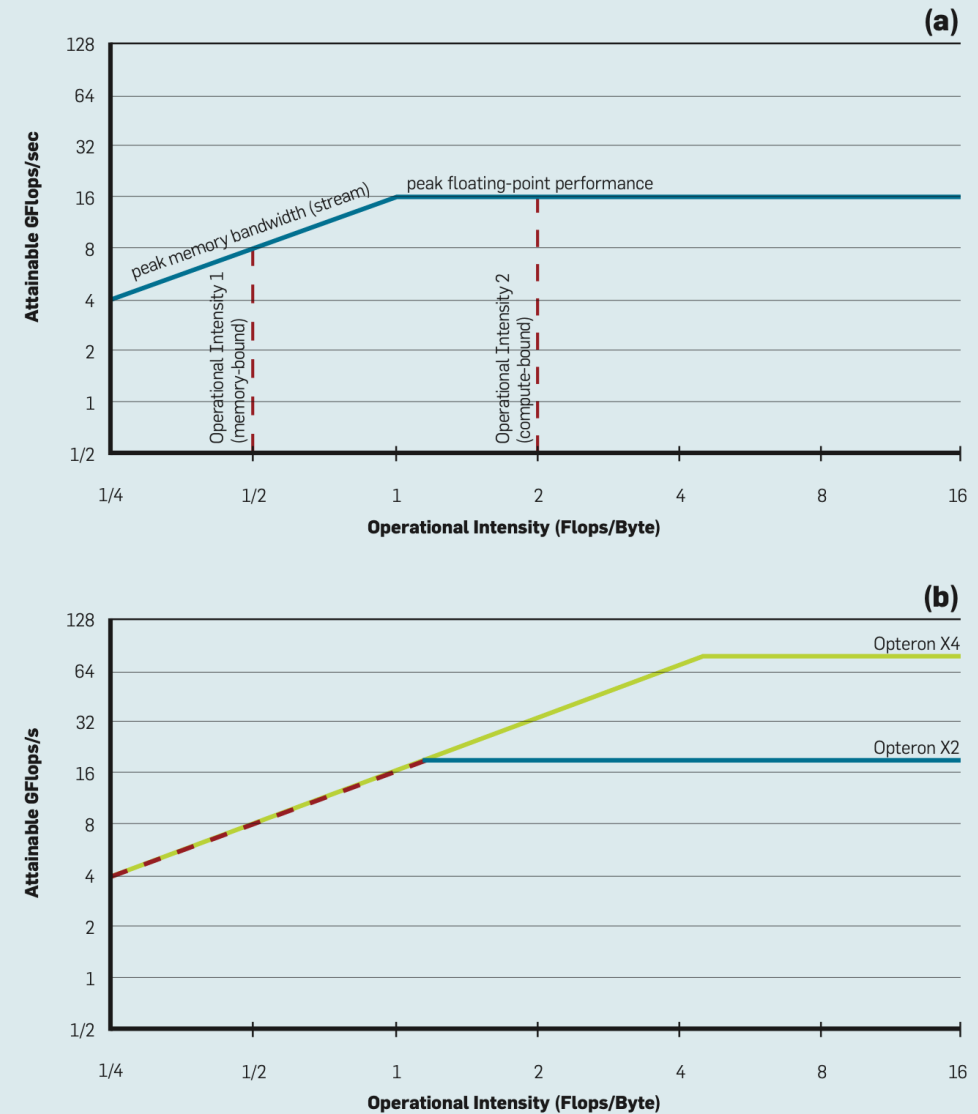
as measured; imagine why?

Hardware bottlenecks

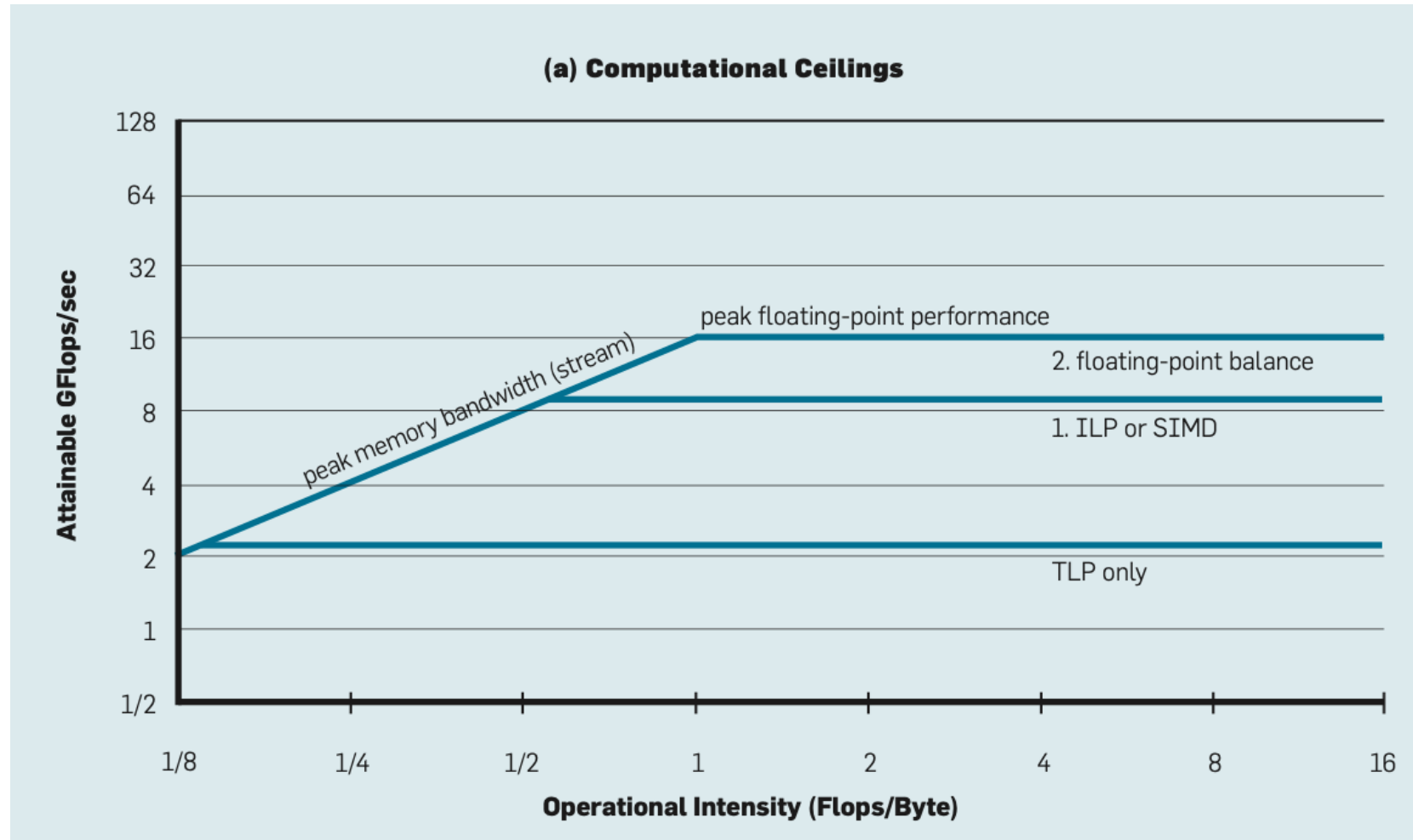
- How can we reason about if a specific model/algorithm/kernel/function is bound by off-chip memory or compute?

Roofline Model

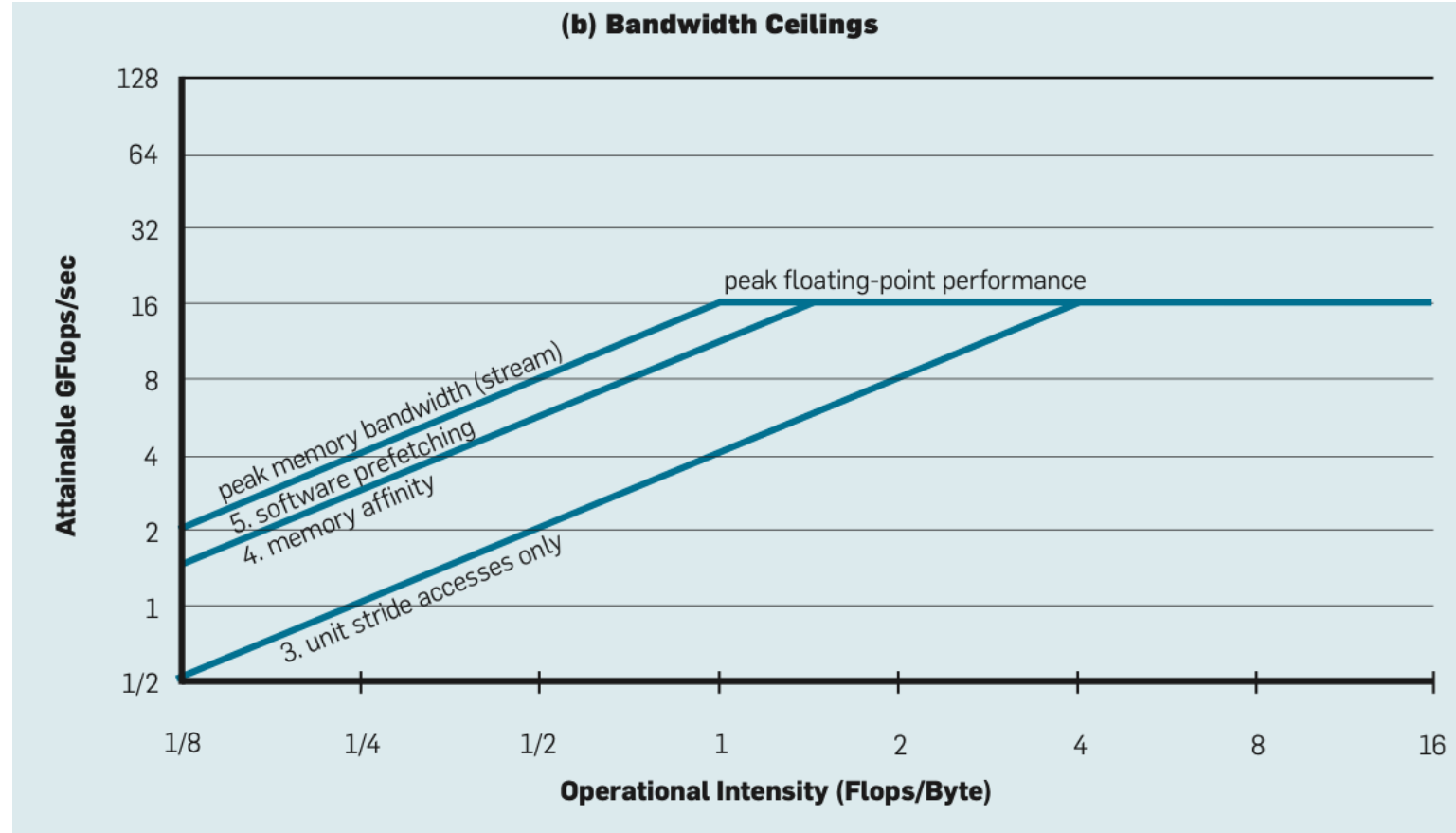
Figure 1: Roofline model for (a) AMD Opteron X2 and (b) Opteron X2 vs. Opteron X4.



Roofline Model



Roofline Model



Roofline Model

