Systems for ML

CSEE 4121 · Spring 2026

Logistics

- **HW3 deadline extended**
  - Due 04/21 at 11:59 PM
  - Arya will hold office hours to help students
  - Office hour time slot to be announced
- **HW4**
  - Will be released this week
  - Similar to HW2. Due 04/28
- **Final exam**
  - Will be held on 04/29. 2 hours.
  - Make up session will be held afterwards. Time slot to be announced

Last Week: Security & Privacy

- Basics of Cryptography
- Public key encryption and certificates
- Authentication & Authorization
- Anonymity in datasets
- Equifax case study
    - In scope for assignment/exams unlike other case studies

ML Models & Hardware

The Success of Machine Learning Today

Chatbots & Agents

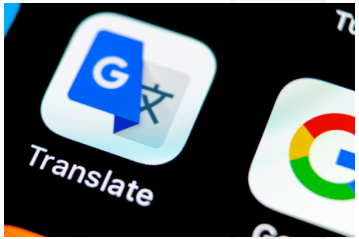
ChatGPT, Claude, Gemini, Grok, many more

Autonomous Vehicles

Tesla, Waymo

Machine Translation

Google Translate, DeepL

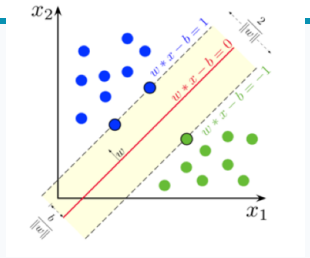


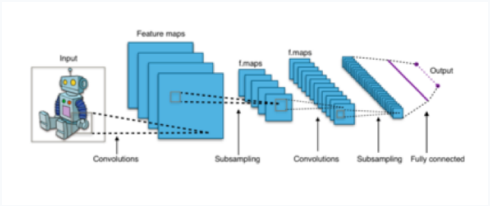
Code Generation

Copilot, Cursor, Claude Code, Antigravity

All powered by massive compute infrastructure — the systems are what made the algorithms practical

ML Techniques: A Brief Timeline





Perceptron — 1958

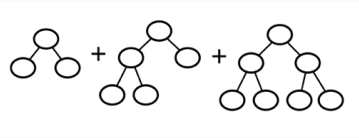
Backprop — 1986

SVM — 1992

ConvNet — 1998

GBM — 1999

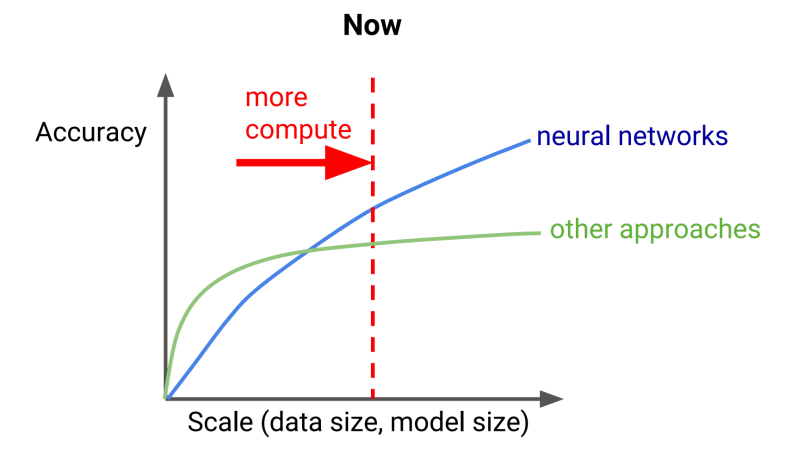
Transformer ★ — 2017



- Most techniques existed by the 1990s — so why didn't ML take off then?
- The Transformer's attention mechanism being parallel was a systems-relevant architectural choice that enabled scaling in ways RNNs could not.

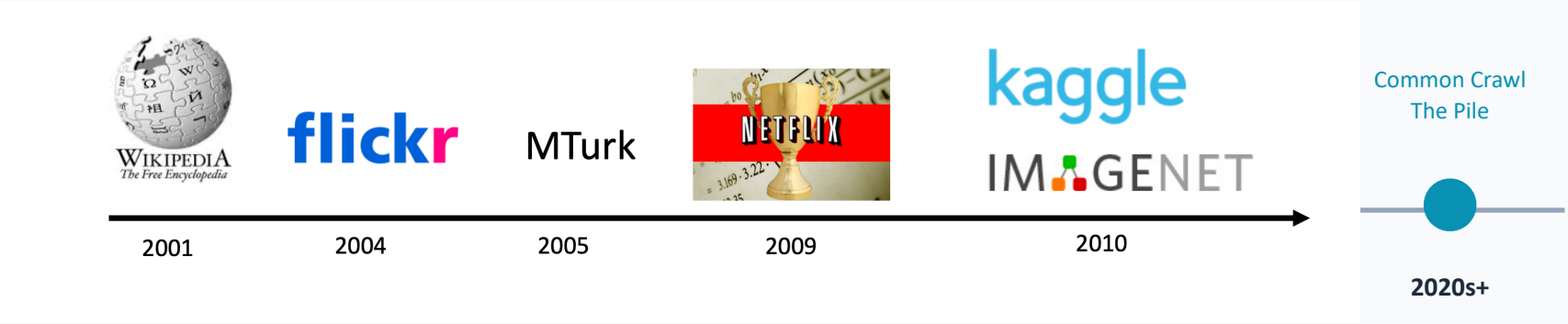
The Rise of Neural Networks

- At small scale (1990s), other approaches (SVMs, random forests) outperformed neural networks
- Neural networks have a higher accuracy ceiling — but only with enough data and compute
- Key shift: with modern hardware and internet-scale data, we moved past the crossover point
- Now neural networks dominate on almost every benchmark



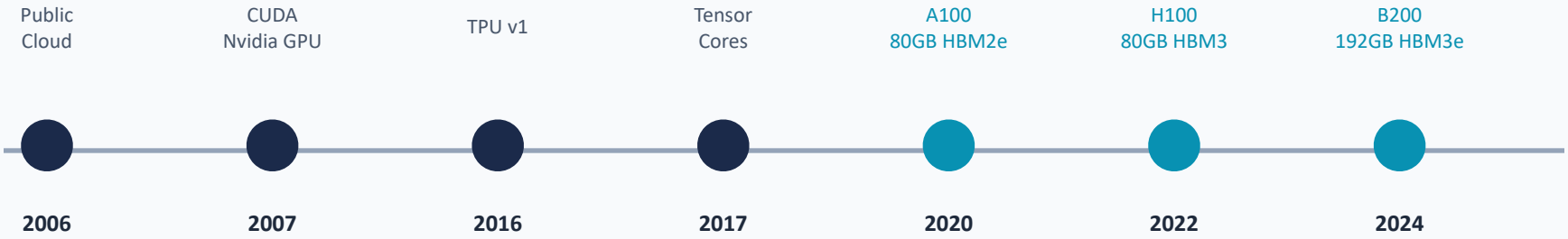
From Jeff Dean, HotChips 2017

Big Data Arrives



- Earlier: curated, labeled datasets (ImageNet = 14M labeled images)
- LLM era: massive unsupervised web scrapes — trillions of tokens from the internet
- The nature of 'big data' changed: from 'labeled examples for one task' to 'all text on the internet'

AI Hardware Timeline

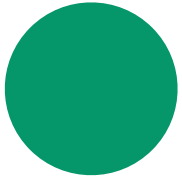


- Each generation: ~2-3x more compute, more memory, higher bandwidth
- H100 vs A100: same 80GB capacity but 3.35 TB/s vs 2 TB/s bandwidth — 67% faster memory
- B200: 192GB HBM3e — first single GPU that can hold a 70B model in FP16
- AMD offerings and NVIDIA Vera Rubin — very hot!

The Three Ingredients of ML Success

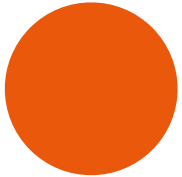
Models

ResNet, Transformers, MoE, ...



Data

ImageNet, Common Crawl, ...



Hardware

GPUs, TPUs, HBM, ...

All three had to come together — no single ingredient was sufficient

Traditional ML Still Matters

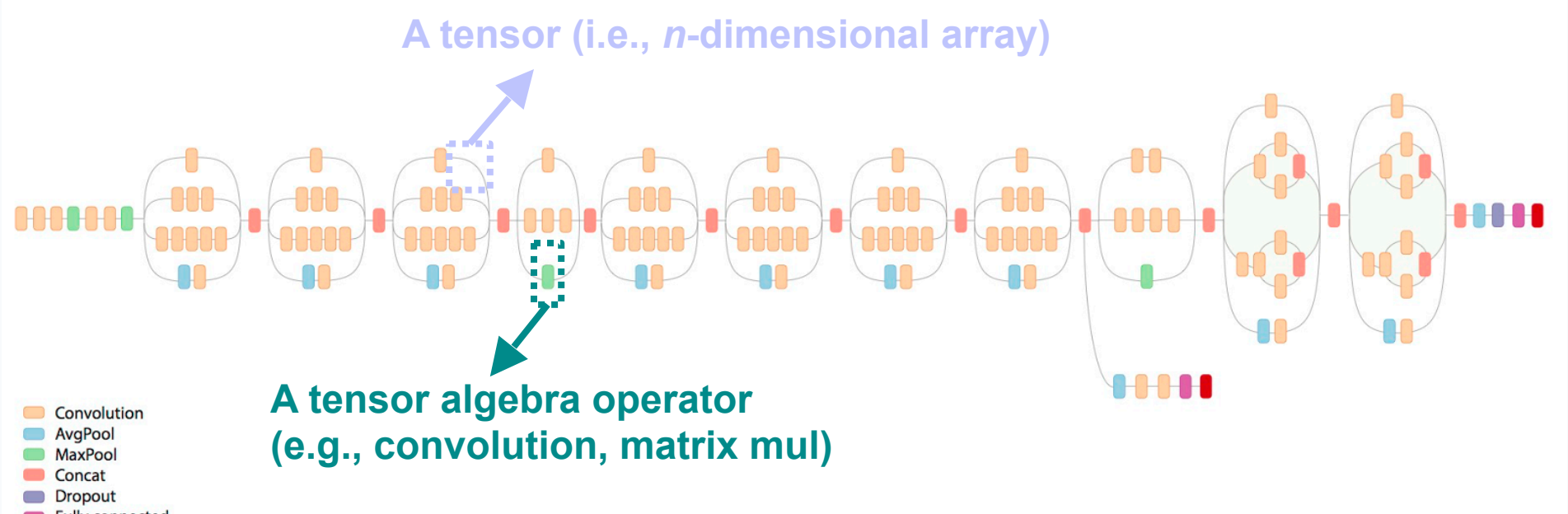
- Regressions, random forests, SVM, XGBoost, LightGBM — extremely widely used
- Very cheap: train on a single CPU in seconds to minutes
- Inference: microseconds per prediction
- XGBoost/LightGBM dominate Kaggle competitions on tabular data
- Used everywhere: fraud detection, monitoring, networking, ad ranking, recommender systems
- Frameworks: scikit-learn (classic), XGBoost/LightGBM (gradient boosting)

Always ask:

Does my task actually require a deep neural network?

Deep Neural Networks

- Collection of trainable mathematical units organized in layers
- Core operations: matrix multiplication, convolution, attention
- Data structures: tensors (n-dimensional arrays — vectors, matrices, higher)
- A DNN is essentially a pipeline of tensor algebra operations



The ML Development Cycle

- Training gets the hype, but data cleaning is where most time actually goes
- Post-training: RLHF, DPO, distillation, constitutional methods — much broader than just 'RL'
- Most practitioners: start with a pretrained model, apply fine-tuning or RAG
- Inference cost grows with users — often dominates total spend over time

Pretrained Models vs. Training from Scratch

Train from Scratch	Fine-tune	Use an API
Thousands of GPUs Months of training $10M — $500M+	1-8 GPUs Hours to days $10 — $10,000	Zero GPUs Seconds Pay per token

The hardware you need varies by 1000x depending on which path you choose

Hardware for ML: Lots of Options

Server CPU

General purpose
64-192 cores
100s GB RAM

Server GPU

Parallel compute
1000s of cores
24-192 GB HBM

Cloud TPU

Google's custom
Matrix operations
JAX/TF optimized

Phone CPU/GPU

Low power
Inference focused
On-device ML

Edge TPU/NPU

Tiny, efficient
Inference only
IoT/sensors

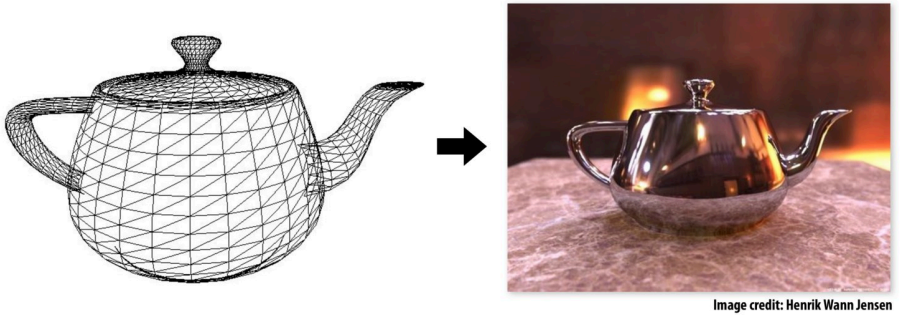
CPU

- General-purpose processor, originally optimized for serial execution
- Server CPUs: 64-192 cores (increasingly parallel over time)
- Large memory: 100s of GB to TBs of DDR5 DRAM
- Memory bandwidth: 200-600+ GB/s for modern dual-socket servers
- Strengths: versatile, huge memory capacity, good at branching and control flow
- Main companies: Intel, AMD, ARM
- Still the right choice for some ML workloads

Key specs

Cores: 64-192
Memory: 100s GB - TBs
Bandwidth: ~200-600 GB/s
Strength: versatility

GPU



Input: triangle mesh

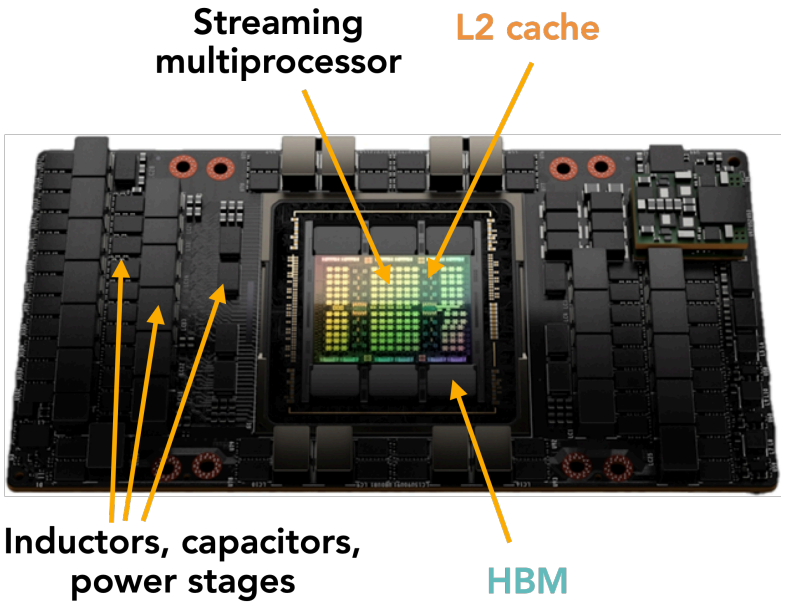
Output: image

- Originally designed for graphics (rendering triangles into pixels)
- Thousands of small cores optimized for parallel computation
- Organized into Streaming Multiprocessors (SMs), each with many cores
- Cores grouped into warps — 32 threads executing the same instruction simultaneously
- Since 2007 (CUDA): repurposed as a general computing platform
- Memory: 24-192 GB of specialized HBM (current generation)
- Memory bandwidth: 2-8 TB/s — 10-30x higher than CPU
- Dominant company: Nvidia (>90% market share for ML training)

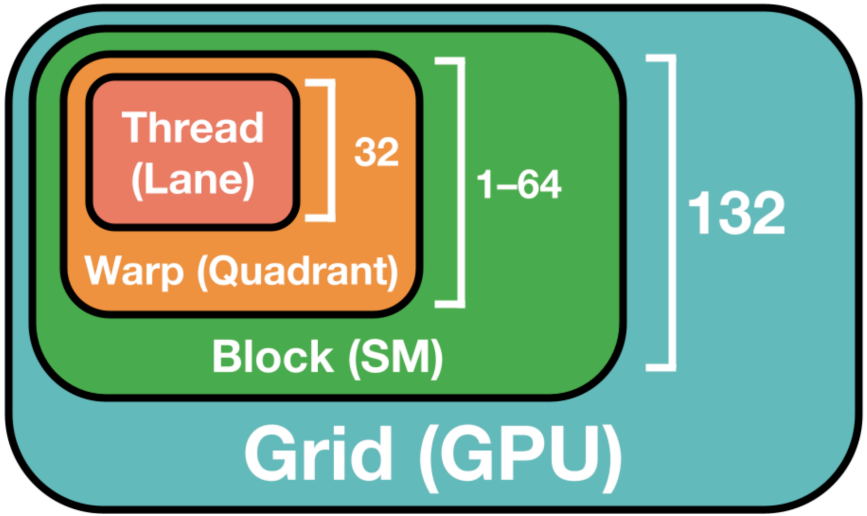
Key specs (H100)

SMs: 132
Memory: 80 GB HBM3
Bandwidth: 3.35 TB/s
Strength: matrix math

GPU components

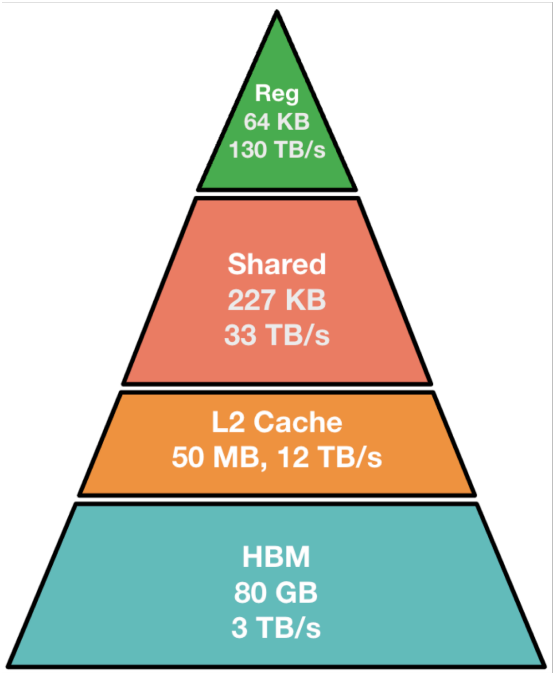


GPU processor hierarchy



All threads in a warp execute the same instruction (SIMT) — this is why GPUs excel at 'same operation on many data elements'

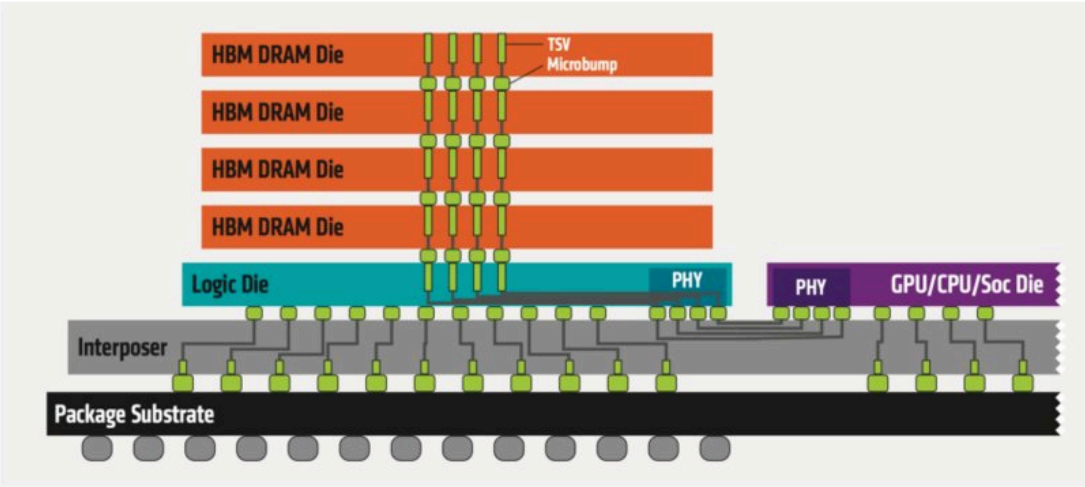
GPU memory hierarchy



40x bandwidth difference between top and bottom — performance depends on keeping data in fast memory

GPU memory: not so commodity

- AI models require very high memory bandwidth
- How is this achieved? HBM (High-Bandwidth Memory)
  - Need to keep memory very close to processor (GPU in this case)
  - Key idea: stack memory chips one on top of the other
  - “Through-Silicon-Vias” (TSV) provide very high bandwidth, interposer connects DRAM stack with GPU
  - Requires smart memory controller which is like a mini processor just for managing the memory
- Result: 100X higher bandwidth than regular DRAM, but can’t pack many DRAM dies so close to processor → overall capacity is 10X or more lower, much more expensive



- This is why GPU memory is precious — every GB matters for model fitting

CPU vs GPU: When to Use Which

Use GPU when...

- Large matrix multiplications
- Same operation on lots of data
- Model has millions+ parameters
- Deep neural networks (CNNs, Transformers)

Use CPU when...

- Small models (trees, regressions)
- Irregular data structures
- Lots of branching/control flow
- Data transfer overhead > computation

Rule of thumb: if your model fits in a spreadsheet, use a CPU. GPU instances cost 5-10x more per hour.

TPUs and Other Accelerators

- Google TPU: purpose-built for matrix ops, tightly integrated with JAX
- AMD MI300X: 192GB HBM3, gaining real traction for training — not just inference anymore
- AWS Trainium, Meta MTIA: custom silicon for specific workloads
- Intel Gaudi: limited adoption, uncertain future

- **Reality: Nvidia GPUs dominate (~70-80% of training)**
  - ~20-25% on TPUs (mostly Google internal)
  - Everything else is a small fraction

- Why? CUDA ecosystem + PyTorch integration = massive software moat
- Even if competitor hardware matches specs, the software ecosystem matters more

Edge Hardware

- GPUs/NPUs on phones and edge devices for ~10 years
- Much weaker than server versions — optimized for low power consumption
- Focused on inference, not training (more data + compute in the cloud)
- Examples: Apple Neural Engine, Qualcomm Hexagon, Google Edge TPU
- Growing use case: on-device models for privacy and latency
  - Voice recognition, keyboard prediction, camera features — all run locally
- Training on edge is rare (maybe will change with federated learning?)

When NOT to Use a GPU

- **Scenario: fraud detection at a bank**
- XGBoost on tabular data — 50 features, 1,000 trees
- Training: 30 seconds on a CPU
- Inference: microseconds per row on a CPU

- **Why a GPU would be slower:**
  - Data transfer overhead (CPU → GPU) exceeds the computation time
  - Model is too small to utilize thousands of GPU cores
  - Tree-based models don't map well to GPU's parallel matrix math

- **Lesson: GPUs accelerate large matrix operations. If your workload doesn't have those, a CPU is better and cheaper.**

Programming ML Hardware

CUDA: Programming GPUs

- CUDA: Nvidia's programming language for GPUs, similar to C/C++
- Very low-level: programmer explicitly manages threads, blocks, memory transfers
- Launched in 2007 — this head start is a major reason Nvidia dominates
- One of Nvidia's key 'moats': even if competitor hardware matches specs, the CUDA ecosystem is massive
- Most ML developers never write CUDA directly
- But CUDA is what makes PyTorch fast under the hood

CUDA Example: Matrix Addition

Regular application thread running on CPU (the "host")

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3);
dim3 numBlocks(Nx/threadsPerBlock.x, Ny/
threadsPerBlock.y);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will launch 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

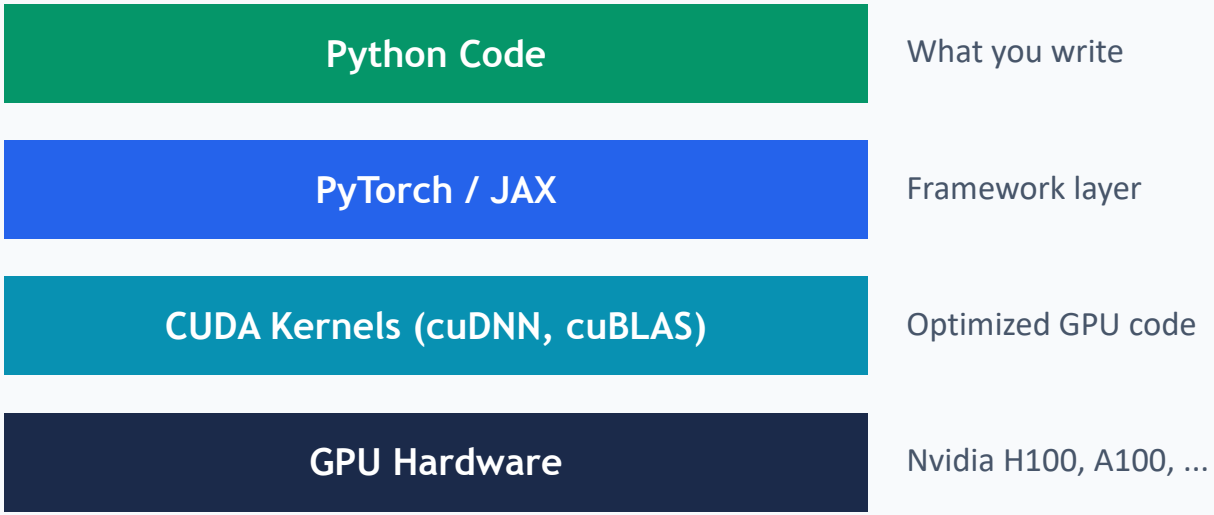
CUDA kernel definition

```
// kernel definition (runs on GPU)
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + B[j][i];
}
```

Key takeaways

- CPU and GPU code clearly separated
- Programmer specifies thread count and block size
- Each thread knows its position via blockIdx + threadIdx
- Thread blocks execute in any order → scalable

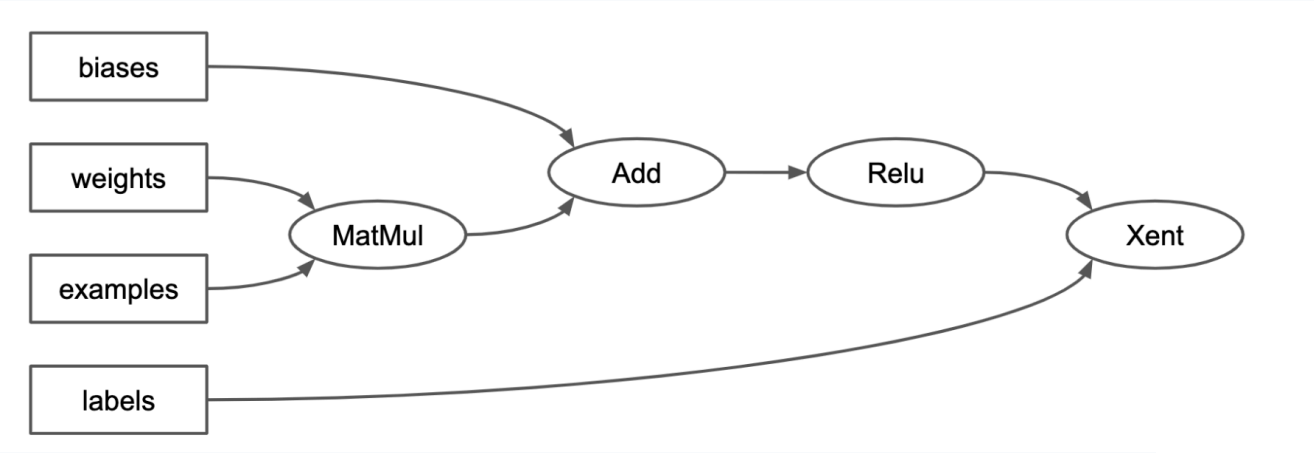
The Software Stack: CUDA is Too Low Level



- PyTorch (Meta): de facto standard — 'Pythonic', used by vast majority
- JAX (Google): functional style, used by Google DeepMind for large-scale training
- TensorFlow (Google): historically important, now largely legacy

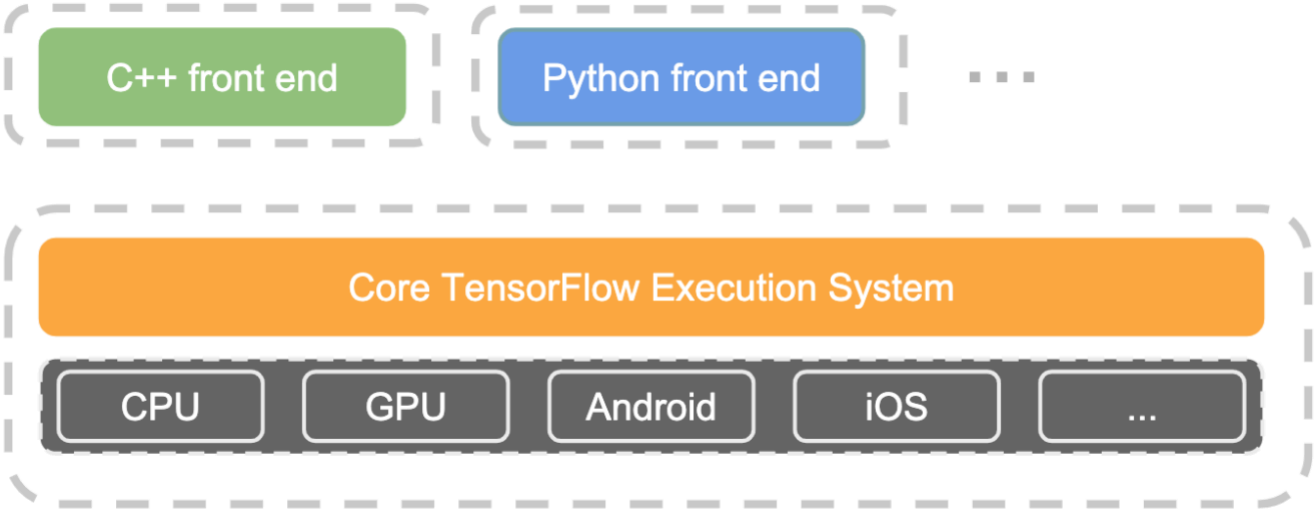
Computation as a Graph

- Both PyTorch and TensorFlow model computation as a directed graph
- Nodes = operations (MatMul, Add, ReLU, Softmax)
- Edges = tensors (n-dimensional arrays flowing between operations)
- **Similar to Spark:**
  - Spark: nodes are transformations, edges are RDDs/DataFrames
  - PyTorch: nodes are tensor ops, edges are tensors
- Graph structure enables: operator fusion, memory planning, automatic differentiation
- Graphs can be run repeatedly (one execution per training mini-batch)



Tensorflow

- A system for training ML models, developed by Google
- Primarily motivated by deep neural networks (DNN)
- Can train DNN models, taking into account hardware on the target device/s



PyTorch

- 'Eager execution' — code runs line by line like normal Python (easy to debug)
- torch.autograd — automatic differentiation, computes gradients for you
- .to('cuda') — one line to move computation to GPU
- Strong CUDA integration via cuDNN, cuBLAS under the hood
- torch.compile() — newer graph-mode optimizations for speed
- Huge ecosystem: Hugging Face, torchvision, torchaudio, ...

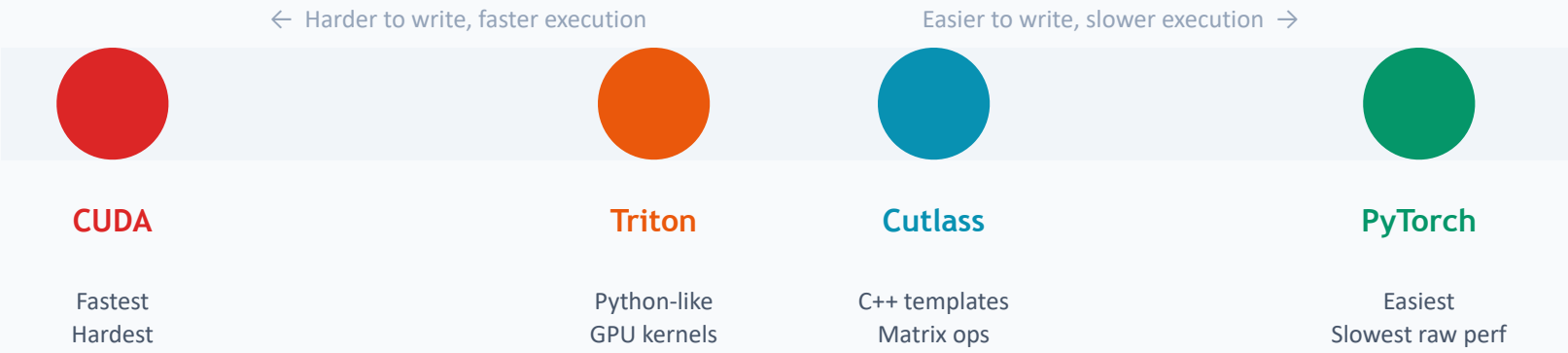
Tensorflow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a =x * y
b =a + z
c=torch.sum(b)

c.backward()
```

Usability vs. Performance



- In production at large AI labs, engineers often drop to lower levels for critical operations
- Triton (OpenAI): now mainstream — integrated into PyTorch via torch.compile
- Key insight: this tradeoff exists because GPU hardware is complex and general frameworks can't always exploit it optimally

Does My Model Fit in Memory?

Model memory = # parameters × bytes per parameter

Precision	Bytes/Param	Llama 3 8B	Llama 3 70B
FP32	4	32 GB	280 GB
FP16 / BF16	2	16 GB	140 GB
INT8	1	8 GB	70 GB
INT4	0.5	4 GB	35 GB

For training, multiply by ~4x: weights + gradients + optimizer states (Adam stores 2 extra copies per param)
Example: training Llama 8B in FP16 ≈ 16 GB × 4 = ~64 GB → barely fits on an 80GB A100

Mixed Precision Training

FP32	32 bits	Full precision, traditional default
FP16	16 bits	Half memory, 2x throughput on tensor cores Can have numerical instability
BF16	16 bits	Same range as FP32, less precision Best of both worlds — preferred
FP8	8 bits	Quarter memory, 4x throughput Cutting edge (DeepSeek V3)

In practice: torch.cuda.amp handles this automatically. Almost always a free lunch — use it.

Batch Size Tradeoffs

- Batch size = number of examples processed together before updating weights
- Larger batch → more memory (activations scale linearly)
- Larger batch → better GPU utilization (more parallel work)
- Too large → OOM crash
- Too small → GPU cores sit idle, wasting money

- **Practical approach:**
  - Start small, double until you OOM, back off one step

- Sweet spot: largest batch size that fits in memory

Data Loading as a Bottleneck

- If GPU finishes a batch before the next is ready, it sits idle ('data starvation')

- **Common causes:**
  - Reading from slow disk, complex CPU preprocessing, single data loading thread

- **Solutions:**
  - Multiple DataLoader workers (num_workers parameter)
  - Prefetching: load next batch while GPU processes current one
  - Pinned memory: CPU memory the GPU can access faster
  - Fast data formats: WebDataset, pre-tokenized data

GPU Utilization: Are You Wasting Money?

nvidia-smi — command line tool to monitor and manage GPUs

0-10%	Data loading bottleneck, or model too small for GPU
30-50%	Batch size too small, or CPU preprocessing bottleneck
80-100%	Good — GPU is well utilized

Expensive hardware should have highest possible utilization

Other tools: PyTorch Profiler, Nsight Systems, Weights & Biases

Choosing Hardware in the Cloud

GPU	Memory	~Cost/hr	Good for
T4	16 GB	$0.50	Inference, small fine-tuning
A10G	24 GB	$1.50	Medium workloads
A100	40/80 GB	$10-15	Serious training
H100	80 GB	$25-35	Large-scale training, fastest

- **Decision framework:**
  - 1. Do I need a GPU at all? (XGBoost/sklearn → probably not)
  - 2. Does my model fit on one GPU? (Do the memory math)
  - 3. How long will training take? (Estimate before launching)
  - 4. What's my budget? (hours × cost/hr)

- Spot/preemptible instances: 60-70% cheaper but can be interrupted — use checkpointing!

Can You Run an LLM on Your Laptop?

Model	FP16 Size	INT4 Size	Runs on...
Llama 3 8B	16 GB	4 GB	Any modern laptop
Llama 3 70B	140 GB	35 GB	64GB Mac or server GPU
DeepSeek V3 671B	1.3 TB	~170 GB	~$10K server (MoE: only 37B active)

- Tool: ollama — one-line install, downloads and runs quantized models locally
- Quantization makes the impossible possible: 70B model goes from needing 2x H100 to fitting on a MacBook
- Model architecture matters: DeepSeek V3's MoE only activates 37B out of 671B params per token

Training & Inference

What Happens During One Training Step

1. Forward Pass

Input flows through layers → store intermediate activations

2. Loss Computation

Compare model output to ground truth

3. Backward Pass

Compute gradients using stored activations (reverse order)

4. Optimizer Update

Adjust weights using gradients (Adam: running mean + variance)

Memory peaks during backward pass — you need activations + gradients + optimizer states simultaneously

Activations: The Hidden Memory Cost

Activation memory ≈ batch_size × seq_len × hidden_dim × num_layers × bytes

Example: Llama 8B, batch=8, seq_len=2048, hidden=4096, 32 layers, BF16
Per layer: 8 × 2048 × 4096 × 2 bytes ≈ 128 MB
All 32 layers: ~4 GB just for activations (on top of model weights + gradients)

- With larger batch or longer sequences, activation memory grows fast
- **Activation checkpointing (gradient checkpointing):**
  - Don't store all activations — recompute them during backward pass
  - Saves memory but costs ~33% more compute
  - PyTorch: torch.utils.checkpoint
- Batch size and sequence length are the knobs you turn when you hit OOM

Training Time Estimation

Training time ≈ (total tokens) / (throughput in tokens/sec)

Example: Fine-tuning Llama 8B on 1M examples
Measured throughput on A100: ~3,000 tokens/sec
1M examples × 512 avg tokens = 512M tokens
Time: 512M / 3,000 ≈ 170K sec ≈ 47 hours ≈ 2 days
Cost: 47 hours × $15/hr = ~$700

- **Always do a short test run (1% of data) first, then extrapolate**
- If estimate is too high: larger batch, mixed precision, faster GPU, or less data
- Estimate before committing — don't discover a 2-week run after it's started

Checkpointing and Fault Tolerance

- Large training runs take days to weeks
- Hardware fails: GPUs overheat, machines crash, cloud instances get preempted
- Without checkpoints: lose all progress, restart from scratch → waste thousands of dollars

- **Checkpointing: periodically save model + optimizer + training state to disk**
  - Checkpoint size = model + optimizer states (8B model, FP32: ~96 GB per checkpoint)

- **How often? Tradeoff:**
  - Too often → slow (disk I/O overhead)
  - Too rarely → lose too much work on crash
  - Typical: every 30 min to few hours, or every N training steps

Monitoring Training Runs

- **Loss curve: should decrease over time**
  - Plateaus early, spikes, or diverges → something is wrong
- **GPU utilization: should be consistently high**
  - Drops indicate data loading or batch size bottlenecks
- **Memory usage: watch for gradual leaks (increasing over time)**
- **Gradient norms: sudden spikes = numerical instability (esp. with FP16)**

- Tools: Weights & Biases (wandb), TensorBoard, nvidia-smi, PyTorch Profiler

- **Don't launch a multi-day run and walk away — monitor it, especially in the first hours**

DeepSeek V3 vs. Llama 3

2.7M	30.8M	~12x	$5.5M
GPU hours (DeepSeek V3)	GPU hours (Llama 3 405B)	fewer GPU hours for larger model	reported cost (GPU rental only)

- **How did DeepSeek use ~12x fewer GPU hours for a larger model?**
  - FP8 mixed precision: ~80% of computation at FP8, halving bandwidth needs
  - Mixture of Experts: 671B total but only 37B active per token
  - Custom pipeline parallelism (DualPipe): better GPU utilization
- **The '$5.5M' only counts GPU rental at ~$2/hr**
  - Excludes R&D, failed experiments, salaries, 50K total GPUs (~$1.3B in hardware)
  - 'training cost' is ambiguous — always ask what's included

When One GPU Isn't Enough — Preview

- **Model doesn't fit on one GPU → model parallelism**
  - Split model across multiple GPUs
- **Training too slow on one GPU → data parallelism**
  - Same model on each GPU, different data, synchronize gradients
- **Both → hybrid parallelism**
  - Combine multiple strategies simultaneously

- The largest models require 3-4 types of parallelism across thousands of GPUs

- **This is what we'll cover in the next two lectures**

Training vs. Inference: Different Bottlenecks

Training

- Large batches (1000s of examples)
- Lots of matrix multiplications
- Compute-bound
- GPU cores are the bottleneck

Inference (text generation)

- One token at a time per user
- Weight × tiny vector each step
- Memory-bound
- Bandwidth is the bottleneck

Different bottlenecks → different hardware specs matter → different optimizations help

Why Inference is Memory-Bound

- To generate one token: load ALL model weights from HBM, multiply by one small vector

- **Example: 70B model in FP16**
  - Load 140 GB of weights from memory
  - At 3 TB/s bandwidth (H100): takes ~47 ms just to read weights
  - The actual computation (matrix-vector multiply): tiny fraction of that time

- **Arithmetic intensity = FLOPs / bytes loaded = very low for single-token generation**

- The GPU's compute cores are mostly idle, waiting for data from memory
- Compare to training: batch of 1024 → 1024x more useful work per weight load

The KV Cache

- During text generation, each new token 'attends' to all previous tokens
- Naive approach: recompute everything from scratch each step → very wasteful
- **KV cache: store Key and Value matrices from previous tokens**

- **The problem: KV cache grows with seq_length × num_layers × hidden_dim × 2**

- **Example: Llama 70B, seq length 4096, FP16**
  - KV cache per request: ~10 GB
  - Serving 8 users simultaneously: 80 GB just for KV caches
  - That equals or exceeds the model weight memory!

- Managing KV cache memory is THE central systems challenge of modern inference

Batching for Inference

- **The fix for memory-boundedness: serve multiple users simultaneously**
- Loading 140 GB of weights anyway → multiply by 32 users' vectors instead of 1
- Same memory traffic, 32x more useful work → much higher throughput

- **Static batching:**
  - Wait for N requests, process together, return all results
  - Problem: fast requests wait for slow ones

- **Continuous batching:**
  - As one request finishes, immediately slot in a new one
  - Like an OS scheduler — no wasted GPU time
  - Used by modern serving: vLLM, TensorRT-LLM, SGLang

Quantization for Deployment

- **Quantization: represent weights with fewer bits after training**
- FP16 → INT8: half the memory, nearly same quality
- FP16 → INT4: quarter the memory, small quality loss, huge speedup

- **Why it helps inference so much:**
  - Inference is memory-bound → halving weight size ≈ doubling speed
  - Less data to load from HBM → less waiting → higher throughput

- Popular tools: GPTQ, AWQ, bitsandbytes, llama.cpp
- When to quantize: almost always for deployment (unless max precision needed)
- Post-training quantization is simpler and usually sufficient

GPU Costs in the Real World

Scenario: Your startup deploys Llama 70B to serve 10,000 requests/day

Average request: 500 input + 200 output tokens
Throughput (INT8, batched, H100): ~100 tokens/sec → each request ≈ 7 sec
10,000 requests × 7 sec = 70,000 GPU-sec/day ≈ 19.4 GPU-hours/day
Monthly: ~580 GPU-hours × $30/hr = ~$17,500/month (one GPU)
With redundancy: 2-3 GPUs → $35-50K/month

- **Training was a one-time cost. Serving is ongoing and scales with users.**
- This is why inference optimization (batching, quantization) matters more than training for most companies
- Also why many companies just use APIs (OpenAI, Anthropic) — let someone else handle infrastructure

Key Takeaways

- GPUs are powerful but expensive — use them only when the workload justifies it
- Memory is the binding constraint: know your model size math
- Mixed precision is (almost) always a free lunch
- Training is compute-bound; inference is memory-bound — different optimizations for each
- Quantization makes large models accessible on smaller hardware
- Infrastructure costs are dominated by inference at scale, not training
- Always estimate costs before committing to a training or serving strategy