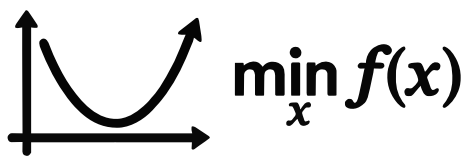
Find the Bottleneck

Speed Up ML Pipelines by 10% – 800%

Michal Šustr Martin Dvořák



Prague, May 4, 2026

Introduction / Motivation

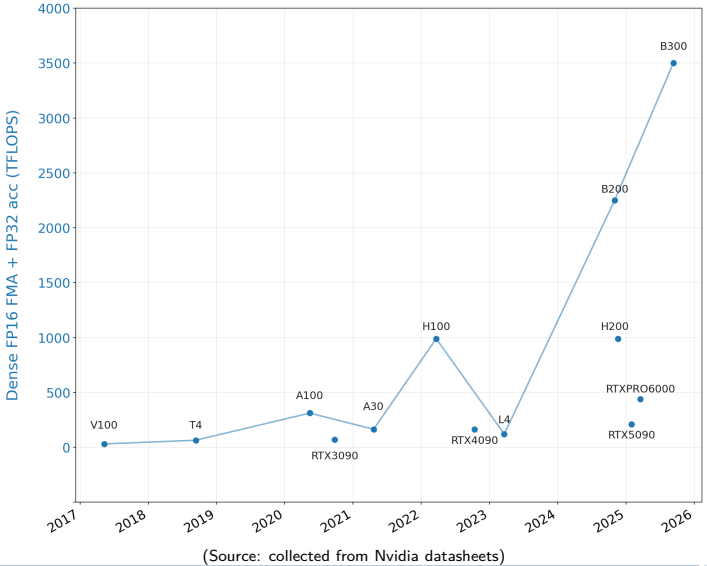
Estimated time: 30 min

GPU performance over time

GPUs are getting faster over time.

Can we keep up feeding them data?

GPU performance over time



(Source: collected from Nvidia datasheets)

Verifying Manufacturer's Claims

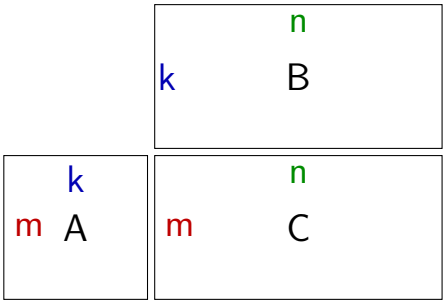
PyTorch FP16 GEMM sanity check

```
A = torch.randn(shape.m, shape.k)
B = torch.randn(shape.k, shape.n)
C = torch.empty(shape.m, shape.n)

torch.mm(A, B, out=C)
```

Verifying Manufacturer's Claims

PyTorch FP16 GEMM sanity check

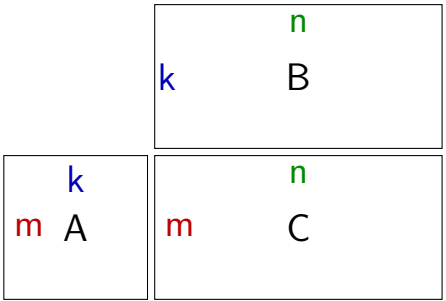
```
A = torch.randn(shape.m, shape.k)
B = torch.randn(shape.k, shape.n)
C = torch.empty(shape.m, shape.n)

torch.mm(A, B, out=C)
```



Verifying Manufacturer's Claims

PyTorch FP16 GEMM sanity check

```
A = torch.randn(shape.m, shape.k)
B = torch.randn(shape.k, shape.n)
C = torch.empty(shape.m, shape.n)

torch.mm(A, B, out=C)
```



`torch.nn.Linear(in_features=k, out_features=m, bias=False)`
is equivalent to matmul with matrix **A** and input matrix **B** with batch size n.

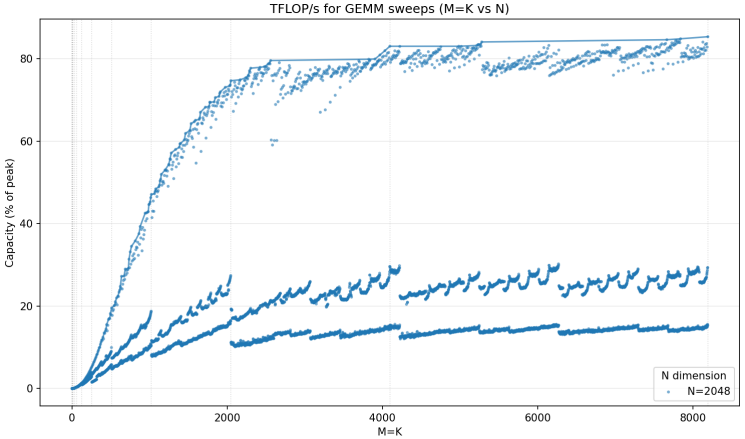
Matmul benchmark - H200 GPU

$$\mathrm{OPS} = 2 \cdot m \cdot n \cdot k$$

$$\mathrm{TFLOPS} = \frac{\mathrm{OPS}/t}{10^{12}}$$

$$\text{OPS} = 2 \cdot m \cdot n \cdot k$$

$$\text{TFLOPS} = \frac{\text{OPS}/t}{10^{12}}$$



Matmul benchmark - H200 GPU

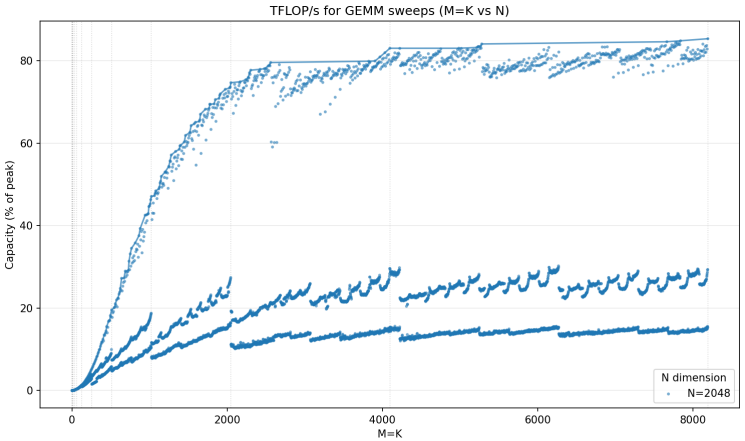
Claimed peak = 835 TFLOPS
- close with 712 TFLOPS and
85% ✓

Matmul benchmark - H200 GPU

Claimed peak = 835 TFLOPS - close with 712 TFLOPS and 85% ✓

"The published specifications for devices are generally peak theoretical numbers. They are not achievable in practice. In practice you may get in the range of 70%-90% of these numbers with well designed tests."

NVIDIA Forum discussion



Throughput of matmul - H200 GPU

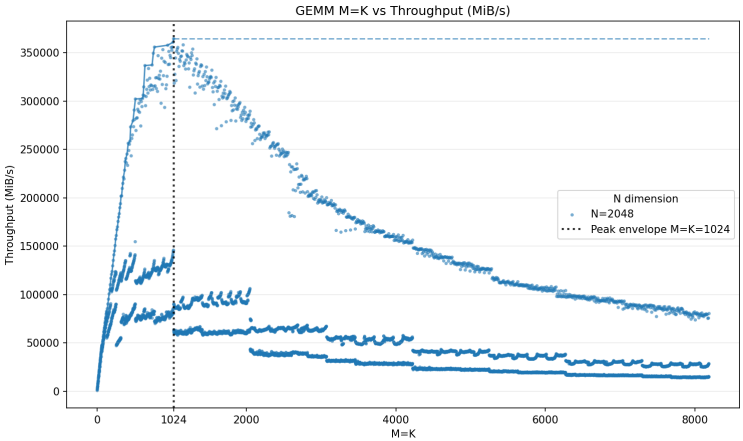
$$\text{InputSize} = n \cdot k \cdot 2 \text{ bytes}$$

$$\text{Throughput} = \frac{\text{InputSize}/t}{2^{20}}$$

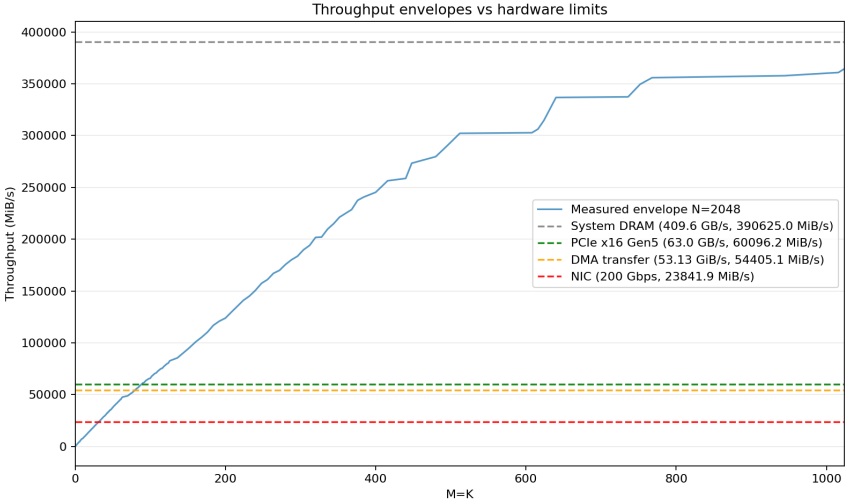
Throughput of matmul - H200 GPU

$$\text{InputSize} = n \cdot k \cdot 2 \text{ bytes}$$

$$\text{Throughput} = \frac{\text{InputSize}/t}{2^{20}}$$



Hardware Limits - H200 GPU

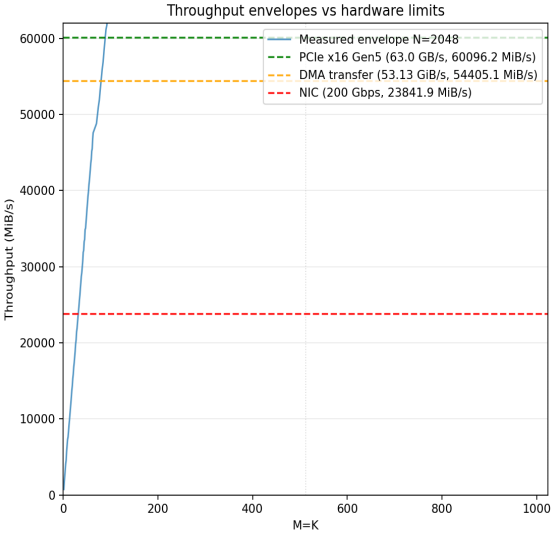


Naive torch dataloader (pseudocode)

```
A = randn(m, k).cuda()

loader = DataLoader(
  dataset=in_memory_dataset,
  batch_size=n, shuffle=True, num_workers=0,
  pin_memory=False, persistent_workers=False)

for _ in range(iters):
  B = next(loader).cuda();
  _ = matmul(A, B)
```

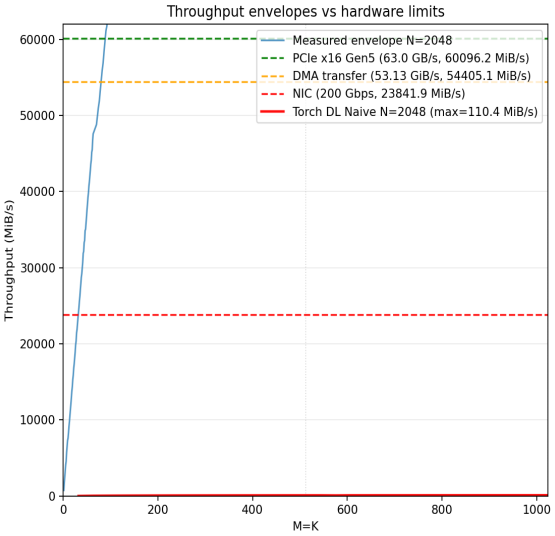
Naive torch dataloader (pseudocode)

```
A = randn(m, k).cuda()

loader = DataLoader(
  dataset=in_memory_dataset,
  batch_size=n, shuffle=True, num_workers=0,
  pin_memory=False, persistent_workers=False)

for _ in range(iters):
  B = next(loader).cuda();
  _ = matmul(A, B)
```



Naive torch dataloader (pseudocode)

```
A = randn(m, k).cuda()

loader = DataLoader(
  dataset=in_memory_dataset,
  batch_size=n, shuffle=True, num_workers=0,
  pin_memory=False, persistent_workers=False)

for _ in range(iters):
  B = next(loader).cuda();
  _ = matmul(A, B)
```

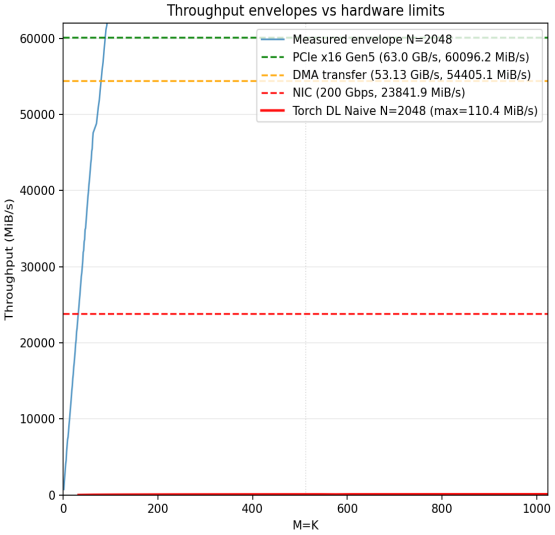


Optimized torch dataloader (pseudocode)

```
A = randn(m, k).cuda()

shuffle_rows = k * workers * 4
loader = DataLoader(
  dataset=in_memory_dataset,
  batch_size=shuffle_rows, num_workers=workers,
  pin_memory=True, persistent_workers=True)
loader = async_cuda_prefetcher(loader)

p = randperm(shuffle_rows, device="cuda")
for _ in range(iters):
  p.shuffle()
  buffer = next(loader)  # already on GPU
  for i in range(shuffle_rows // k):
    idx = p[i*k:(i+1)*k]
    B = buffer[idx, :]
    _ = matmul(A, B)
```

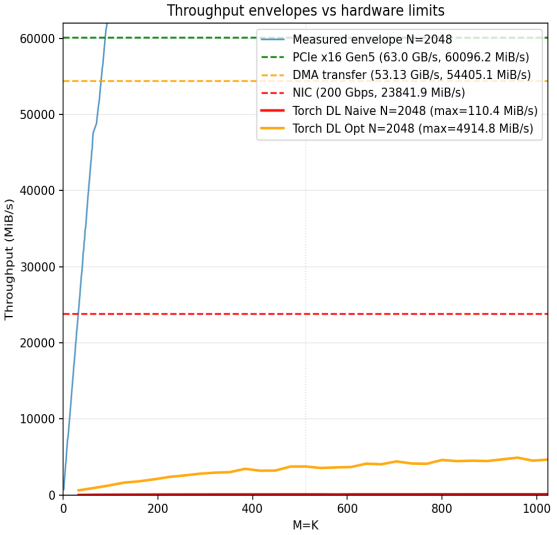
Optimized torch dataloader (pseudocode)

```
A = randn(m, k).cuda()

shuffle_rows = k * workers * 4
loader = DataLoader(
  dataset=in_memory_dataset,
  batch_size=shuffle_rows, num_workers=workers,
  pin_memory=True, persistent_workers=True)
loader = async_cuda_prefetcher(loader)

p = randperm(shuffle_rows, device="cuda")
for _ in range(iters):
  p.shuffle()
  buffer = next(loader)  # already on GPU
  for i in range(shuffle_rows // k):
    idx = p[i*k:(i+1)*k]
    B = buffer[idx, :]
    _ = matmul(A, B)
```



Optimized torch dataloader (pseudocode)

```
A = randn(m, k).cuda()

shuffle_rows = k * workers * 4
loader = DataLoader(
  dataset=in_memory_dataset,
  batch_size=shuffle_rows, num_workers=workers,
  pin_memory=True, persistent_workers=True)
loader = async_cuda_prefetcher(loader)

p = randperm(shuffle_rows, device="cuda")
for _ in range(iters):
  p.shuffle()
  buffer = next(loader)  # already on GPU
  for i in range(shuffle_rows // k):
    idx = p[i*k:(i+1)*k]
    B = buffer[idx, :]
    _ = matmul(A, B)
```

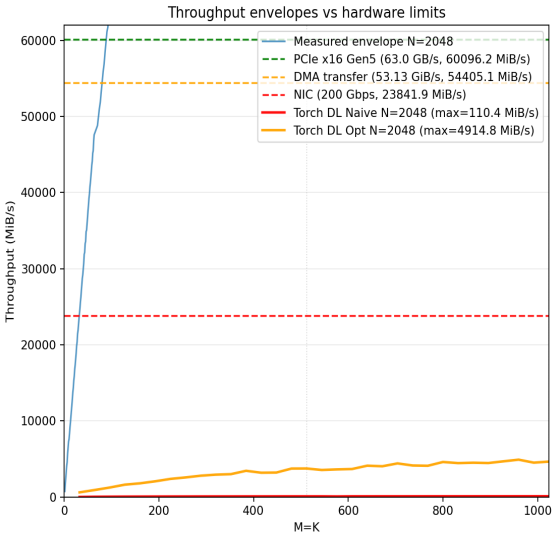


Minfx's RapidTensor (pseudocode)

```
A = randn(m, k).cuda()

cfg = Config(window_size=n,
             buffer_size=pow2(k * workers * 4),
             n_producers=workers)

with Session(cfg) as session:
  loader = session.make_loader(
    urls=["file:///dev/shm/dataset"] * workers,
    batch_size=k
  )

  for _ in range(iters):
    B = next(loader)  # already on GPU
    _ = matmul(A, B)
```

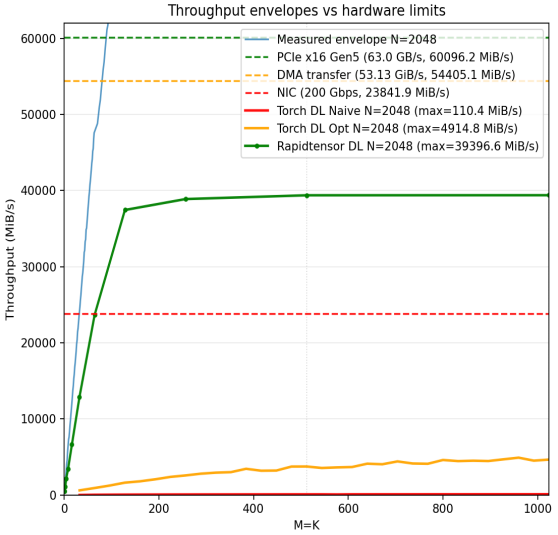
Minfx's RapidTensor (pseudocode)

```
A = randn(m, k).cuda()

cfg = Config(window_size=n,
             buffer_size=pow2(k * workers * 4),
             n_producers=workers)

with Session(cfg) as session:
  loader = session.make_loader(
    urls=["file:///dev/shm/dataset"] * workers,
    batch_size=k
  )

  for _ in range(iters):
    B = next(loader)  # already on GPU
    _ = matmul(A, B)
```



Minfx's RapidTensor (pseudocode)

```
A = randn(m, k).cuda()

cfg = Config(window_size=n,
             buffer_size=pow2(k * workers * 4),
             n_producers=workers)

with Session(cfg) as session:
  loader = session.make_loader(
    urls=["file:///dev/shm/dataset"] * workers,
    batch_size=k
  )

  for _ in range(iters):
    B = next(loader)  # already on GPU
    _ = matmul(A, B)
```



MLP benchmark setup

```
model = MlpModel(p, input_dim=1024, num_classes=10)
# p controls model size:
#  width ~ 1024 * 2^p (fractional interpolation)
#  width is rounded to nearest multiple of 16
#  depth ~ floor(p) + 1 (+ transition layer for frac p)
```

Benchmark parameters

- num_samples = 1,000,000
- batch_size = 512
- workers = 8
- epochs = 5
- warmup_steps = 10

MLP benchmark setup

```
model = MlpModel(p, input_dim=1024, num_classes=10)
# p controls model size:
#  width ~ 1024 * 2^p (fractional interpolation)
#  width is rounded to nearest multiple of 16
#  depth ~ floor(p) + 1 (+ transition layer for frac p)
```

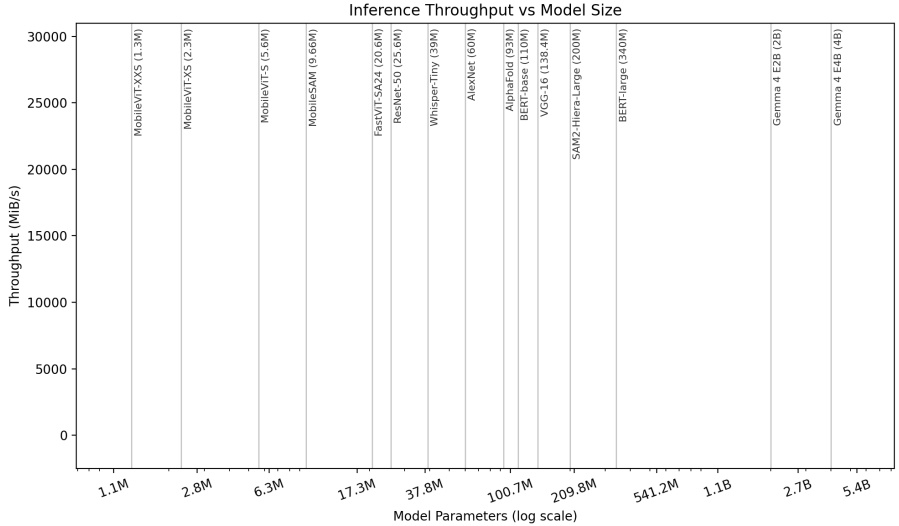
Benchmark parameters

- num_samples = 1,000,000
- batch_size = 512
- workers = 8
- epochs = 5
- warmup_steps = 10

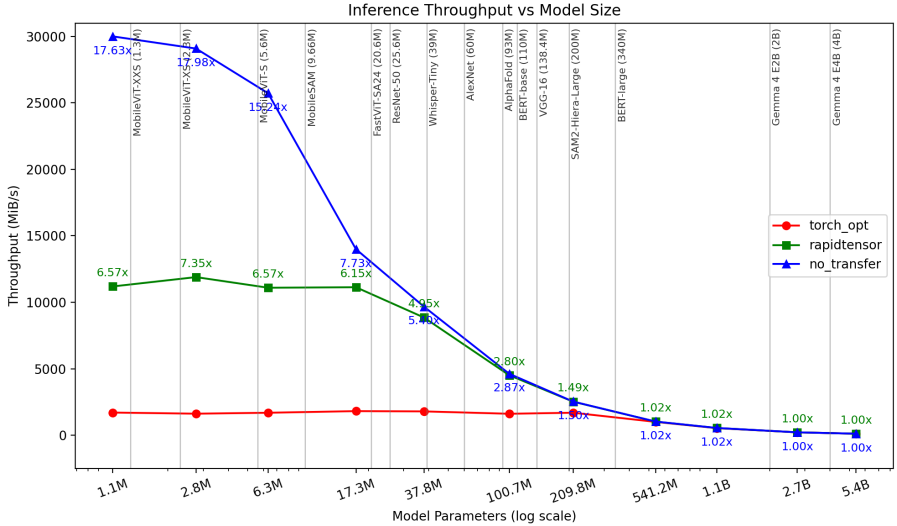
Backends compared

- no_transfer
- torch_opt
- rapidtensor

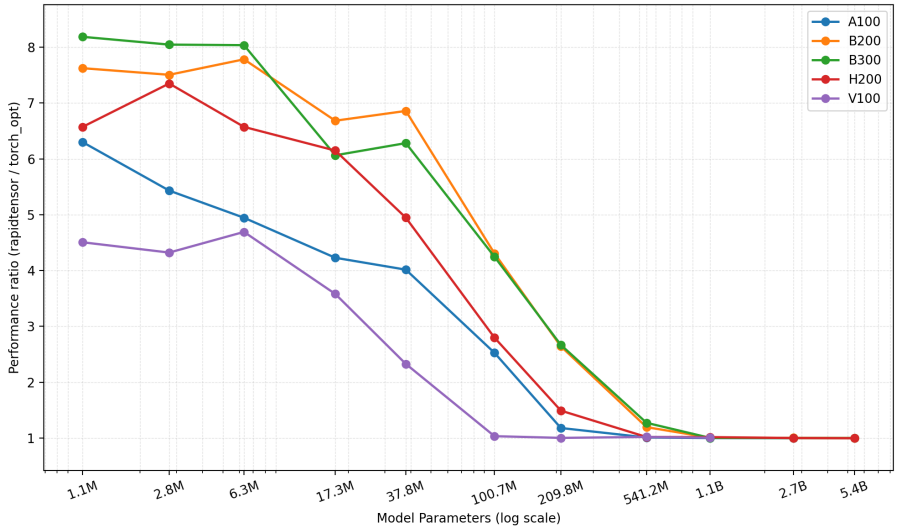
Benchmark: MLP - H200 GPU



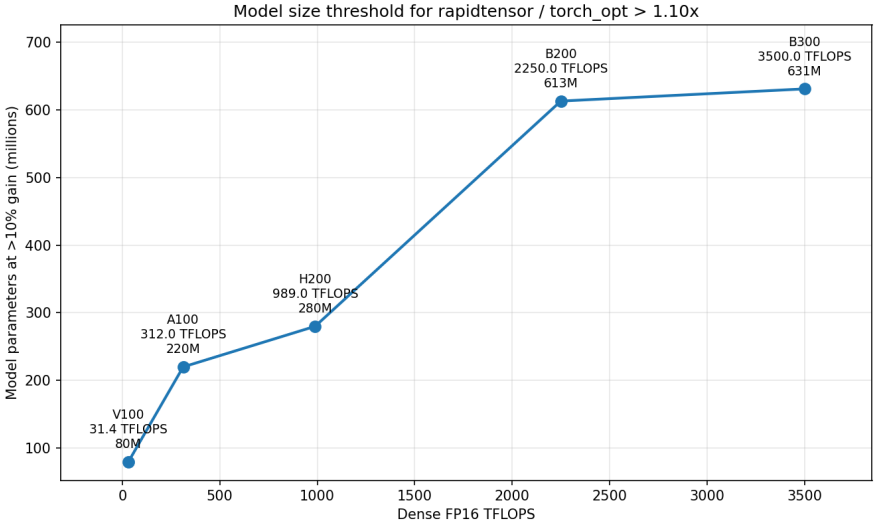
Benchmark: MLP - H200 GPU



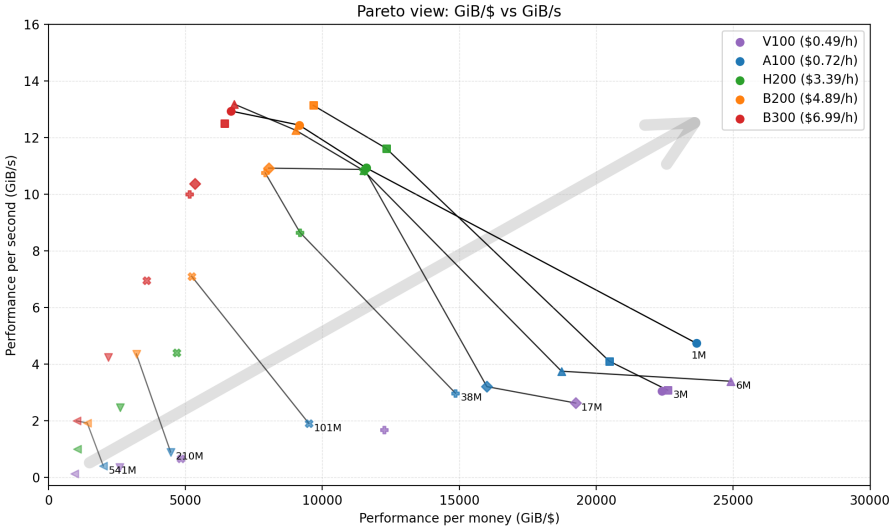
Benchmark: MLP - Performance Gap Ratio



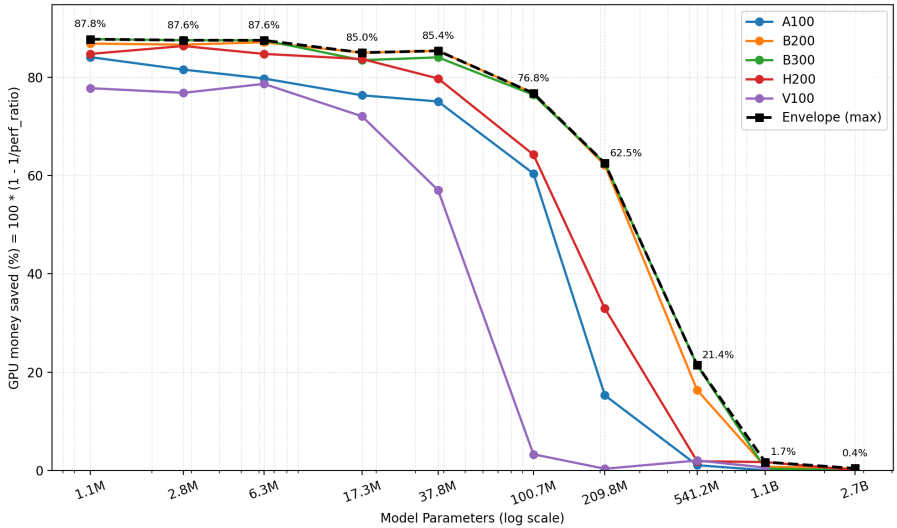
Benchmark: MLP across GPUs



Benchmark: MLP - Pareto Optimal Points



Benchmark: MLP - GPU % Money Saved



Other `rapidtensor` Benefits

- Very simple to use with non-local storage (S3/GCS, other micro-services)

Other `rapidtensor` Benefits

- Very simple to use with non-local storage (S3/GCS, other micro-services)
- Very low system resource usage: few CPU cores (~ 8), low memory usage ($\sim 1 - 2$GB)

Other rapidtensor Benefits

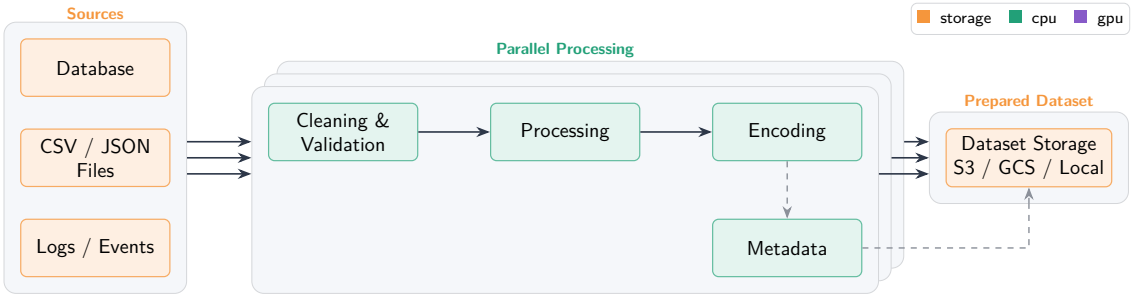
- Very simple to use with non-local storage (S3/GCS, other micro-services)
- Very low system resource usage: few CPU cores (~ 8), low memory usage ($\sim 1 - 2$GB)
- No CPU/GPU OOMs in the middle of the job

Typical Pipeline Setup: Data Preparation

Typical ML pipeline has three steps:

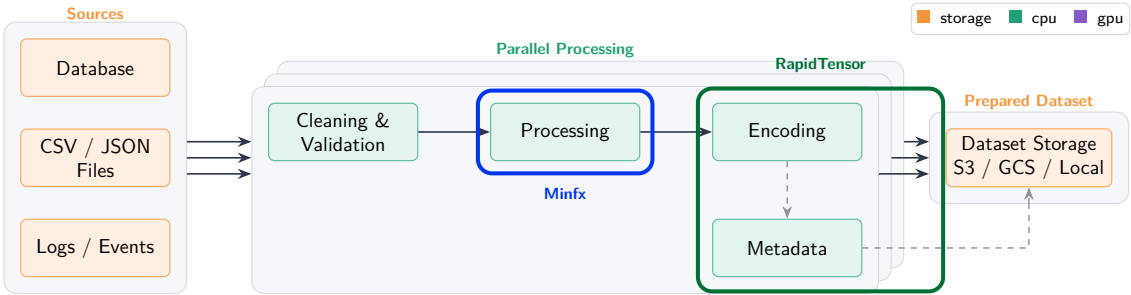
1. Data preparation
2. Training
3. Inference

Typical Pipeline Setup: Data Preparation



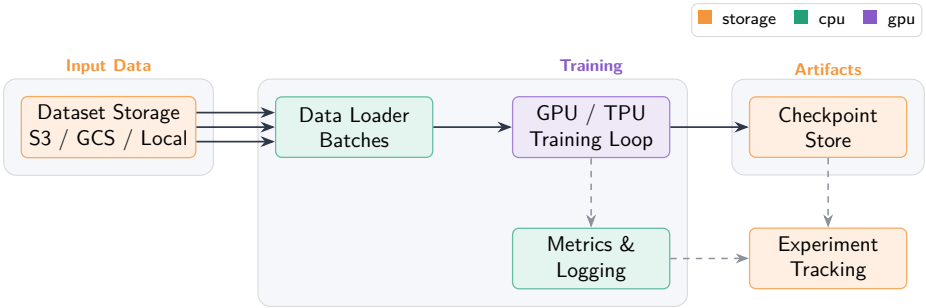
- Processing pipeline can be on cheap CPU nodes, rather than expensive GPU nodes.
- Massive parallelism, processing many TiBs / PiBs at the same time.
- Technologies: Ray, Dask, Flyte, Spark, etc.

Typical Pipeline Setup: Data Preparation

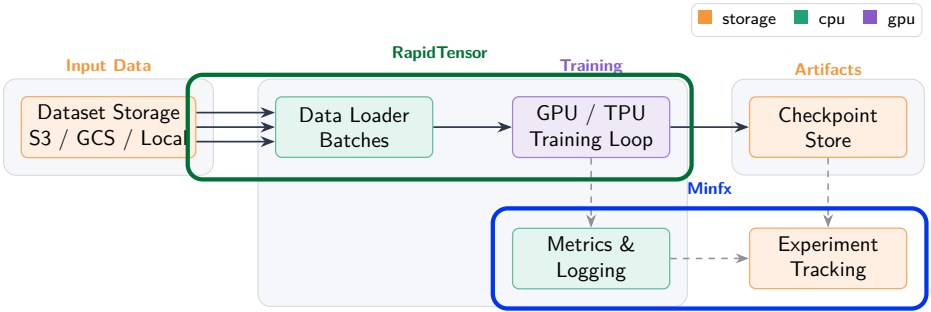


- Save data in a file format for efficient loading
- Track progress and throughput/latency of the pipeline - like distributed `tqdm`

Typical Pipeline Setup: Training

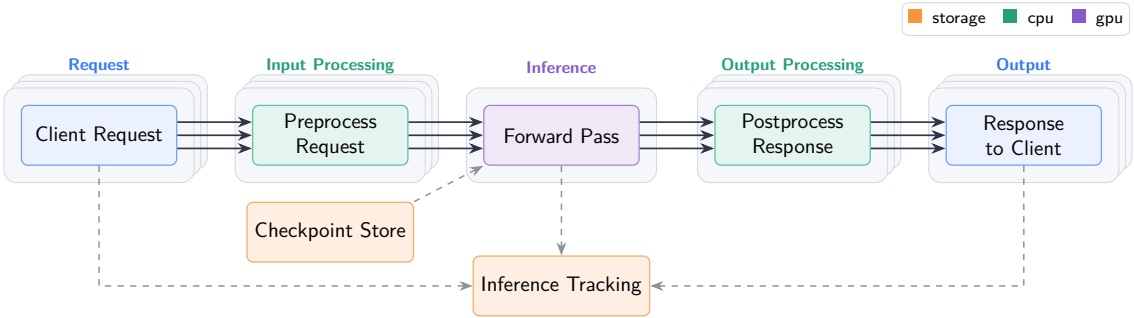


Typical Pipeline Setup: Training

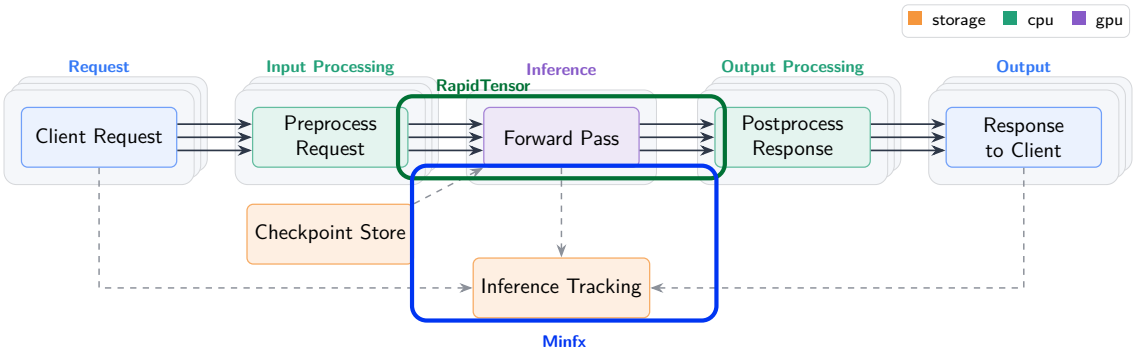


- Load data straight to GPU with high throughput
- Track experiment metrics (loss, accuracy, etc.) - like `wandb`

Typical Pipeline Setup: Inference

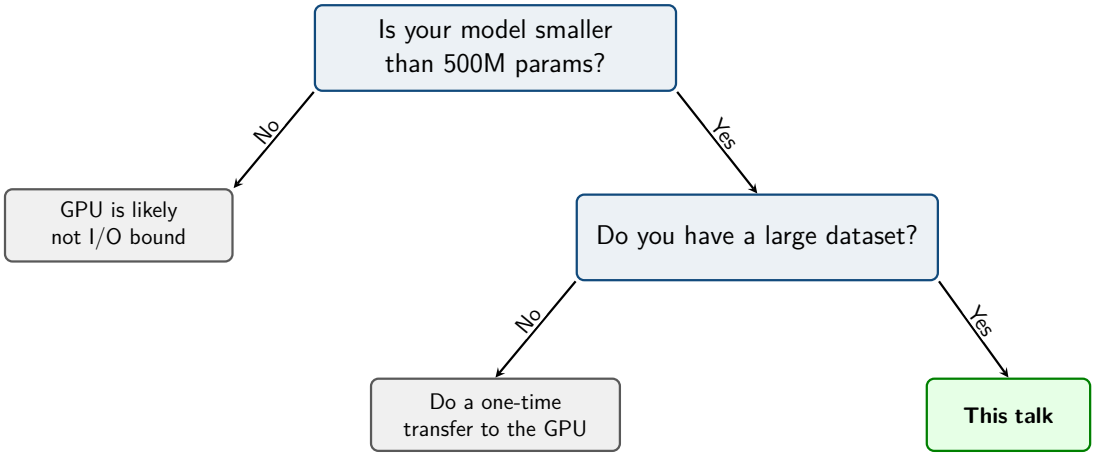


Typical Pipeline Setup: Inference



- Load data straight to GPU with high throughput, easily plug in with microservices
- Track inference metrics

Who Is This Talk For?



Not covered in this talk

- LLM specific optimizations (KV cache, etc.)

Not covered in this talk

- LLM specific optimizations (KV cache, etc.)
- Model optimization (we will mention basic things)

Not covered in this talk

- LLM specific optimizations (KV cache, etc.)
- Model optimization (we will mention basic things)
- Multi-node/multi-GPU setups (we will talk about related hardware)

Not covered in this talk

- LLM specific optimizations (KV cache, etc.)
- Model optimization (we will mention basic things)
- Multi-node/multi-GPU setups (we will talk about related hardware)
- Preprocessing-specific optimizations (but what we present is generally applicable)

LinkedIn

Michal Šustr

- Founder of min $f(x)$
- Engineer at Equilibre.ai / Tower Research
- Intern at DeepMind
- PhD from Czech Technical University

LinkedIn

Martin Dvořák

- Founder of min $f(x)$
- Founder of DataChine
- Performant data processing systems
- Strong focus on low-level optimization

Why Are We Doing This Talk

- Understanding systems and doing perf. optimizations is very fun for us
- We will give you knowledge and tools to do it yourself
- Or you can buy it from us and save time and money

Questions?

https://mlprague2026.minfx.ai/

Schedule

14:00 - 15:30

- (30 min) Introduction / Motivation — THEORY
- (20 min) Performance Basics — THEORY
- (35 min) Modern Hardware — THEORY
- (5 min) Profiling tools (start) — PRACTICE

15:30 - 16:00 *Break for 30 minutes. Good time to setup your laptop for the next part.*

16:00 - 17:30

- (25 min) Programming with Hardware in Mind — THEORY
- (15 min) Profiling tools (continue) — PRACTICE
- (50 min) Dataloading and Profiling — THEORY PRACTICE

$\sim 2-5$ min questions/answers at the end of each block. Please mark slide number.

Performance Basics

Estimated time: 20 min

Thinking About Performance

"Let's write down the code in as simple a way as possible and deal with performance later, when we can profile."

Thinking About Performance

> *"Let's write down the code in as simple a way as possible and deal with performance later, when we can profile."*

However:

- Flat profile with no obvious hotspots
- For a library: users don't understand its code
- Harder to change a deployed system
- Bad expensive work-arounds like over-replication or overprovisioning.

J. Dean & S. Ghemawat: https://abseil.io/fast/hints.html

Performance: Dev loop. *Most important slide.*

1. Build features
2. Check for correctness / tests
3. Goto 1 until application is complete
4. Optimize for performance

Performance: Dev loop. *Most important slide.*

1. Build features
2. Check for correctness / tests
3. Goto 1 until application is complete
4. Optimize for performance

Performance: Dev loop. *Most important slide.*

1. Build features
2. Check for correctness / tests
3. Goto 1 until application is complete
4. Optimize for performance

1. Understand target hardware

Performance: Dev loop. *Most important slide.*

1. Build features
2. Check for correctness / tests
3. Goto 1 until application is complete
4. Optimize for performance

1. Understand target hardware
2. Squeeze maximum in a simple representative setup. **Build harness!**

Performance: Dev loop. *Most important slide.*

1. Build features
2. Check for correctness / tests
3. Goto 1 until application is complete
4. Optimize for performance

1. Understand target hardware
2. Squeeze maximum in a simple representative setup. **Build harness!**
3. Iteratively add features **while keeping performance**

Performance: Dev loop. *Most important slide.*

1. Build features
2. Check for correctness / tests
3. Goto 1 until application is complete
4. Optimize for performance

1. Understand target hardware
2. Squeeze maximum in a simple representative setup. **Build harness!**
3. Iteratively add features **while keeping performance**

Latency vs. Throughput

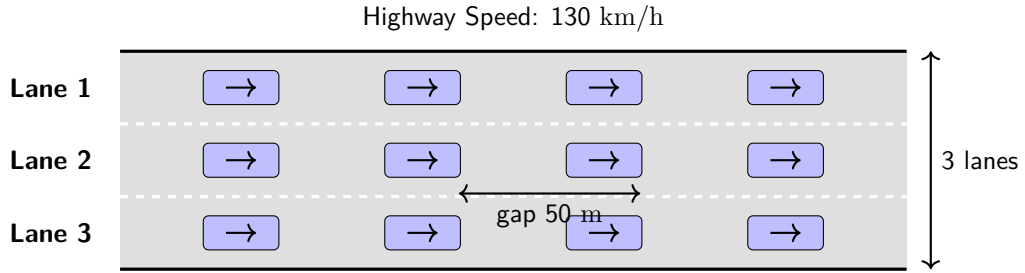
- Latency: It takes 18-24 hours to build a car.
- Throughput: A finished car comes out of the factory every minute on average.

Throughput vs. Bandwidth

- Throughput: Actual achieved rate
- Bandwidth: Maximum possible rate

Throughput vs. Bandwidth

- Throughput: Actual achieved rate
- Bandwidth: Maximum possible rate



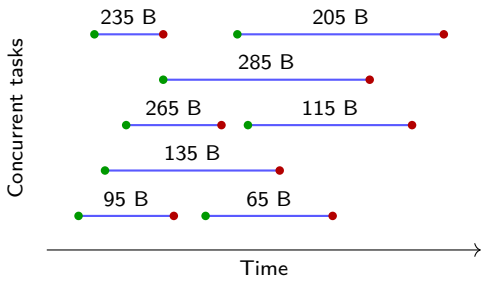
Per lane: $\frac{130{,}000\ \mathrm{m/h}}{50\ \mathrm{m}} = 2{,}600\ \mathrm{cars/h}$

Total: $3 \times 2{,}600 = 7{,}800\ \mathrm{cars/h}$

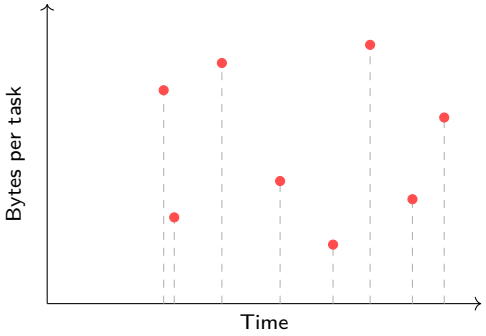
Tasks and Output over Time

- Each segment is a task: start → finish
- A task produces some amount of data.
- We deal with a large amount of tasks.

Tasks over Time



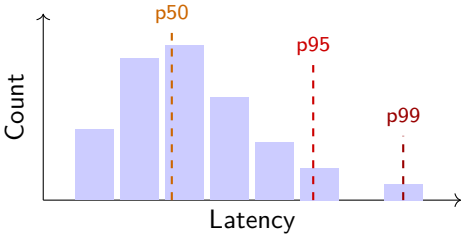
Output over Time



Latency Distribution

Latency is a distribution, not one number

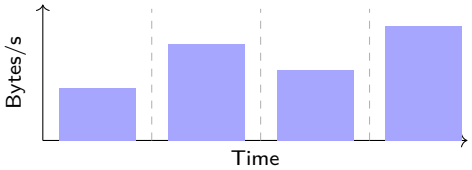
- Track percentiles: $p50$, $p95$, $p99$.
- $p50$: typical request; $p99$: tail behavior users feel.
- Tail growth often comes from queueing and contention.
- Implementation: online histogram with predetermined bins (fp16 trick)



Throughput over Time

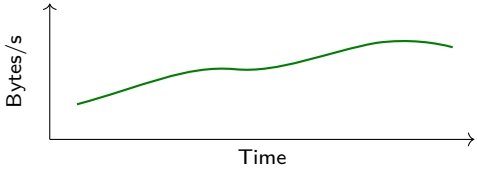
Discrete windows (fixed bins)

- Split time into fixed intervals (e.g. $\Delta t = 1s$).
- Throughput per bin: bytes/Δt.
- CPU utilization: active time / Δt.
- Implementation: simple counters with periodic resets.



Sliding windows (rolling average)

- Recompute every step over the last T seconds.
- Smoother trend, less aliasing at bin boundaries.
- Implementation: time-ordered event deque with expiry by timestamp.

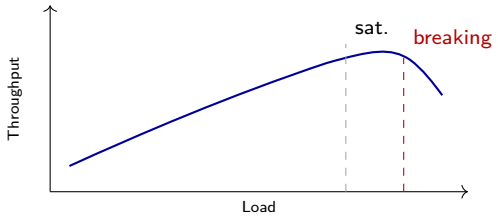


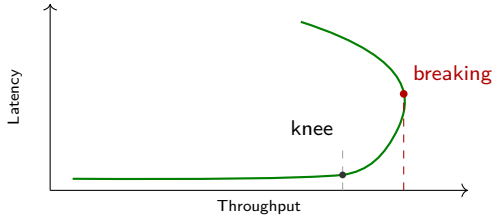
Tradeoff: visibility of bursts vs trends - can ideally report both.

Latency and Throughput Under Load

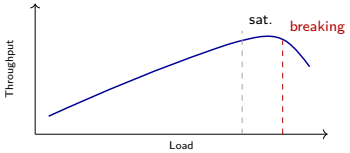
Relation to throughput

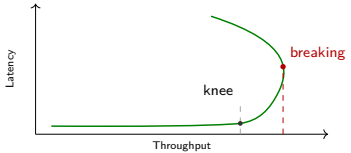
- Increasing load improves throughput *until saturation*.
- Near the knee, queues form and latency rises sharply.
- Past saturation, systems can enter a breaking region.
  - Latency spikes dramatically (timeouts, retries).
  - Throughput may actually drop (due to failures, contention).

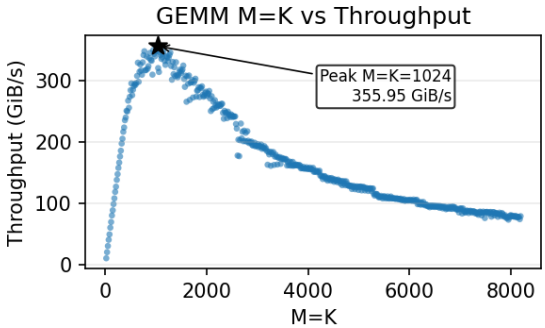


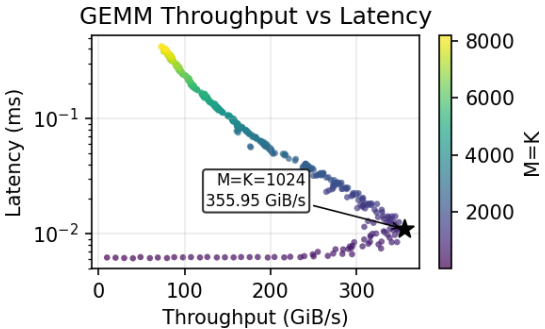


Matmul - Breaking Point

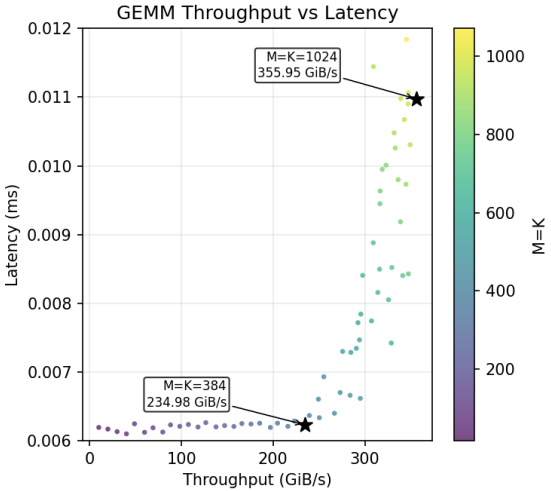


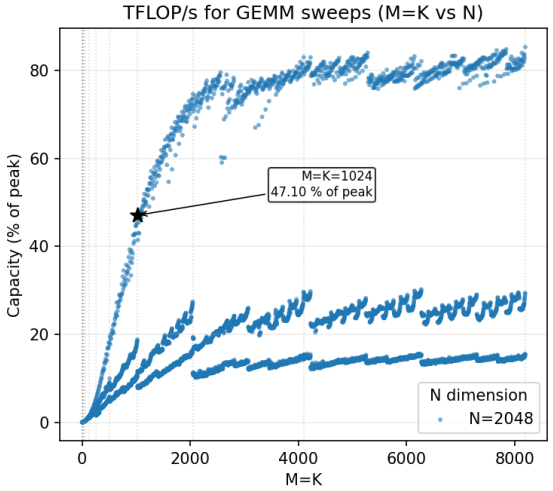






Matmul - Breaking Point and Capacity





Optimizing Latency vs Throughput

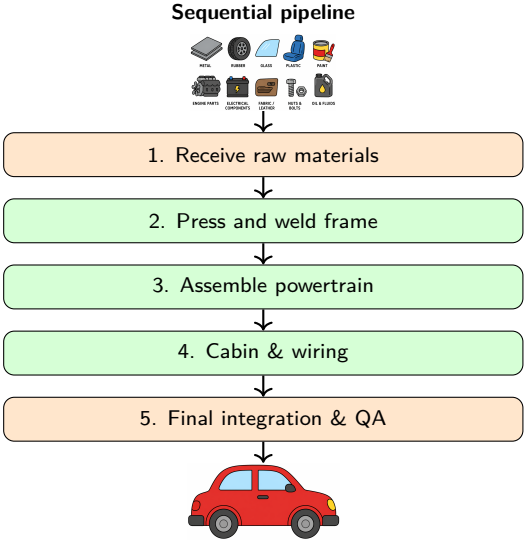
Latency

- Track p50, p95, p99; optimize primarily for tail latency.
- Keep p99 at expected peak load, not only average load.
- Watch queueing delays, retries, and timeouts.
- Reduce contention: shorter critical sections, fewer shared bottlenecks.
- Prefer bounded queues to avoid unbounded waiting time.

Throughput

- Increase load gradually and locate the knee before saturation.
- Use batching carefully: it boosts throughput but can hurt latency.
- Scale parallelism / batch size until useful work plateaus (then stop).

Amdahl's Law


Sequential pipeline
METAL
RUBBER
GLASS
PLASTIC
PAINT
ENGINE PARTS
ELECTRICAL COMPONENTS
FABRIC / LEATHER
NUTS & BOLTS
OIL & FLUIDS
1. Receive raw materials
2. Press and weld frame
3. Assemble powertrain
4. Cabin & wiring
5. Final integration & QA

Amdahl's Law

Sequential pipeline

1. Receive raw materials
2. Press and weld frame
3. Assemble powertrain
4. Cabin & wiring
5. Final integration & QA

Parallelized pipeline

1. Serial intake & planning
2. Body shop
3. Powertrain
4. Cabin & wiring
5. Serial final integration & QA

Amdahl's Law

Sequential pipeline

1. Receive raw materials
2. Press and weld frame
3. Assemble powertrain
4. Cabin & wiring
5. Final integration & QA

Parallelized pipeline

1. Serial intake & planning
2. Body shop
3. Powertrain
4. Cabin & wiring
5. Serial final integration & QA

$$\text{Speedup} = \frac{1}{1 - \mathrm{P} + \frac{\mathrm{P}}{n}}$$

- P: fraction in the parallelizable work section.
- $1 - P$: serial work that cannot be parallelized.

Amdahl's Law

Sequential pipeline

1. Receive raw materials
2. Press and weld frame
3. Assemble powertrain
4. Cabin & wiring
5. Final integration & QA

Parallelized pipeline

1. Serial intake & planning
2. Body shop
3. Powertrain
4. Cabin & wiring
5. Serial final integration & QA

$$\text{Speedup} = \frac{1}{1 - \mathrm{P} + \frac{\mathrm{P}}{n}}$$

- P: fraction in the parallelizable work section.
- $1 - P$: serial work that cannot be parallelized.

Example: $\text{Speedup} = \dfrac{1}{0.4 + \frac{0.6}{3}} = \dfrac{1}{0.6} \approx 1.67\times$

Storage units (bytes)

Name	SI (decimal)	IEC (binary)
kilo /kibi	$1\ \mathrm{kB} = 10^3\ \mathrm{B}$	$1\ \mathrm{KiB} = 2^{10}\ \mathrm{B}$
mega/mebi	$1\ \mathrm{MB} = 10^6\ \mathrm{B}$	$1\ \mathrm{MiB} = 2^{20}\ \mathrm{B}$
giga /gibi	$1\ \mathrm{GB} = 10^9\ \mathrm{B}$	$1\ \mathrm{GiB} = 2^{30}\ \mathrm{B}$
tera /tebi	$1\ \mathrm{TB} = 10^{12}\ \mathrm{B}$	$1\ \mathrm{TiB} = 2^{40}\ \mathrm{B}$

Storage units (bytes)

Name	SI (decimal)	IEC (binary)
kilo /kibi	$1\ \mathrm{kB} = 10^3\ \mathrm{B}$	$1\ \mathrm{KiB} = 2^{10}\ \mathrm{B}$
mega/mebi	$1\ \mathrm{MB} = 10^6\ \mathrm{B}$	$1\ \mathrm{MiB} = 2^{20}\ \mathrm{B}$
giga /gibi	$1\ \mathrm{GB} = 10^9\ \mathrm{B}$	$1\ \mathrm{GiB} = 2^{30}\ \mathrm{B}$
tera /tebi	$1\ \mathrm{TB} = 10^{12}\ \mathrm{B}$	$1\ \mathrm{TiB} = 2^{40}\ \mathrm{B}$

$2^1 = 2$	$2^5 = 32$	$2^9 = 512$	$2^{13} = 8192$
$2^2 = 4$	$2^6 = 64$	$2^{10} = 1024$	$2^{14} = 16384$
$2^3 = 8$	$2^7 = 128$	$2^{11} = 2048$	$2^{15} = 32768$
$2^4 = 16$	$2^8 = 256$	$2^{12} = 4096$	$2^{16} = 65536$

Units

Storage units (bytes)

Name	SI (decimal)	IEC (binary)
kilo /kibi	$1\ \mathrm{kB} = 10^3\ \mathrm{B}$	$1\ \mathrm{KiB} = 2^{10}\ \mathrm{B}$
mega/mebi	$1\ \mathrm{MB} = 10^6\ \mathrm{B}$	$1\ \mathrm{MiB} = 2^{20}\ \mathrm{B}$
giga /gibi	$1\ \mathrm{GB} = 10^9\ \mathrm{B}$	$1\ \mathrm{GiB} = 2^{30}\ \mathrm{B}$
tera /tebi	$1\ \mathrm{TB} = 10^{12}\ \mathrm{B}$	$1\ \mathrm{TiB} = 2^{40}\ \mathrm{B}$

$2^1 = 2$	$2^5 = 32$	$2^9 = 512$	$2^{13} = 8192$
$2^2 = 4$	$2^6 = 64$	$2^{10} = 1024$	$2^{14} = 16384$
$2^3 = 8$	$2^7 = 128$	$2^{11} = 2048$	$2^{15} = 32768$
$2^4 = 16$	$2^8 = 256$	$2^{12} = 4096$	$2^{16} = 65536$

Network rates

- $1\ \mathrm{B} = 8\ \mathrm{b}$
- $1\ \mathrm{Gbps} = 10^9\ \mathrm{b/s}$
- $1\ \mathrm{Gib/s} = 2^{30}\ \mathrm{b/s}$

Useful conversions

$$1\ \mathrm{Gbps} = 125\ \mathrm{MB/s} \approx 119.21\ \mathrm{MiB/s}$$

$$100\ \mathrm{Gbps} = 12.5\ \mathrm{GB/s} \approx 11.64\ \mathrm{GiB/s}$$

Questions?

Modern Hardware

Estimated time: 35 min

Let's understand the limits of modern hardware,
so we know how much performance we can get out of it!

Hardware: About this section

Intro (8 slides):

- Hardware targets
- Hardware diagrams
- Compute vs. memory scaling

CPU-centric (16 slides):

- Programming abstractions
- Speed comparisons
- Caches: locality, lines
- Memory: addresses, page tables, pinning, DMA

Interconnects for ML Systems (28 slides):

- Ethernet
- InfiniBand
- PCIe
- NVLink

GPUs (not covered):

- We are optimizing how to get the data to the GPU, not how to implement efficient GPU kernels.

Hardware Targets

NVIDIA GPUs:

- V100, A100, H200, B200, B300
- Multi-GPU per node, multi-node per system

Hardware Targets

NVIDIA GPUs:

- V100, A100, H200, B200, B300
- Multi-GPU per node, multi-node per system

CPUs that usually come with the GPUs:

- Intel Xeons
- AMD EPYCs
- NVIDIA Grace CPUs

Hardware Targets

NVIDIA GPUs:

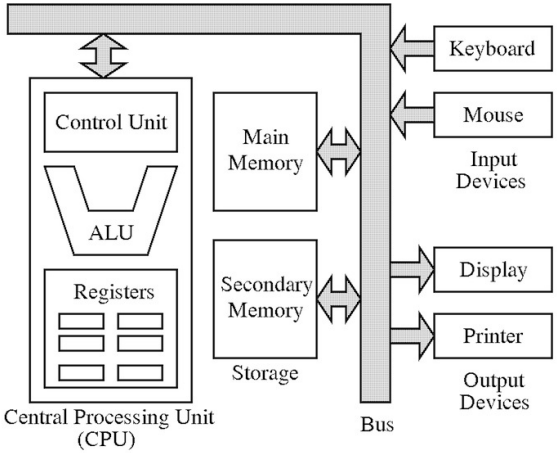
- V100, A100, H200, B200, B300
- Multi-GPU per node, multi-node per system

CPUs that usually come with the GPUs:

- Intel Xeons
- AMD EPYCs
- NVIDIA Grace CPUs

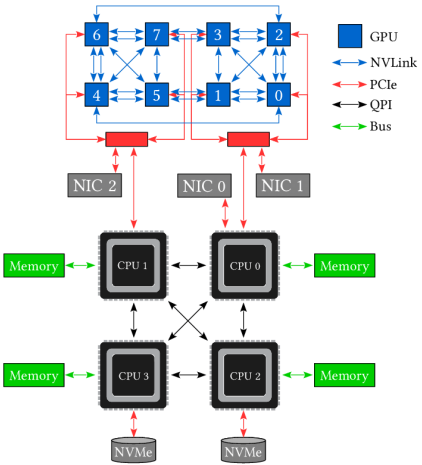
- ISA: x86 (Intel/AMD), Arm (NVIDIA Grace)
- Cores: tens to 192+ per socket
- Host memory: DDR4/5, LPDDR5X, commonly from hundred GiB to multiple TiB per node
- GPU memory: HBM / HBM3e, tens to hundreds of GB per GPU
- NUMA groups: 1-8
- SSD: NVMe (3-14 GB/s read)
- Interconnects: Ethernet / InfiniBand, PCIe / NVLink
- PCIe: commonly Gen4 / Gen5 x16 for GPUs, NICs, and NVMe

Hardware Diagram: Textbook CPU Model



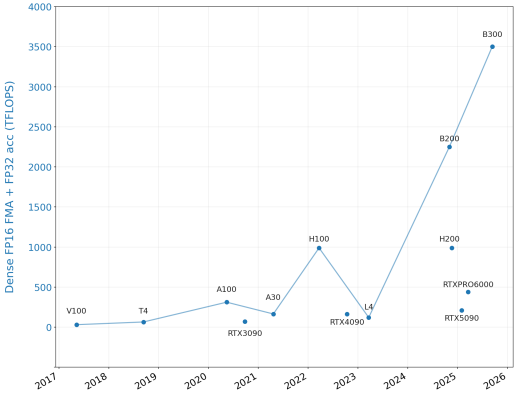
The von Neumann model is not wrong; it is just too flat to explain modern performance.

Hardware Diagram: Modern Heterogeneous Node

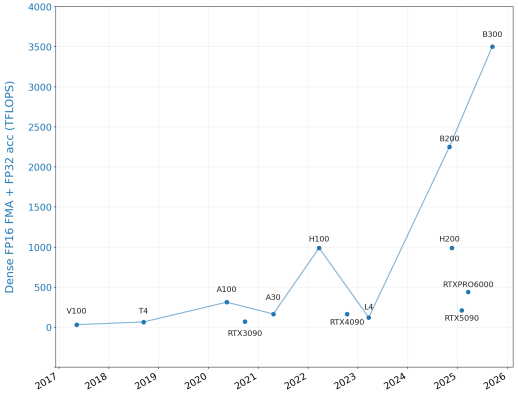


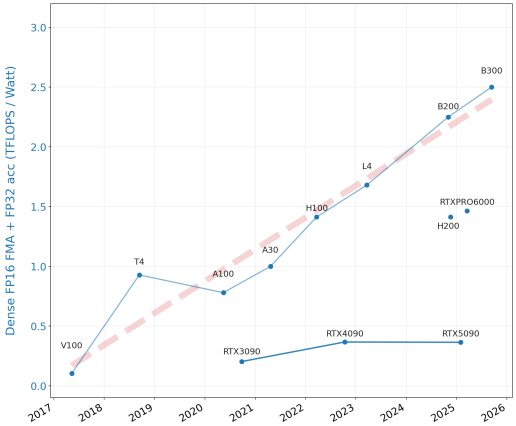
Source: Zhao et al. (2022)

GPU performance over time



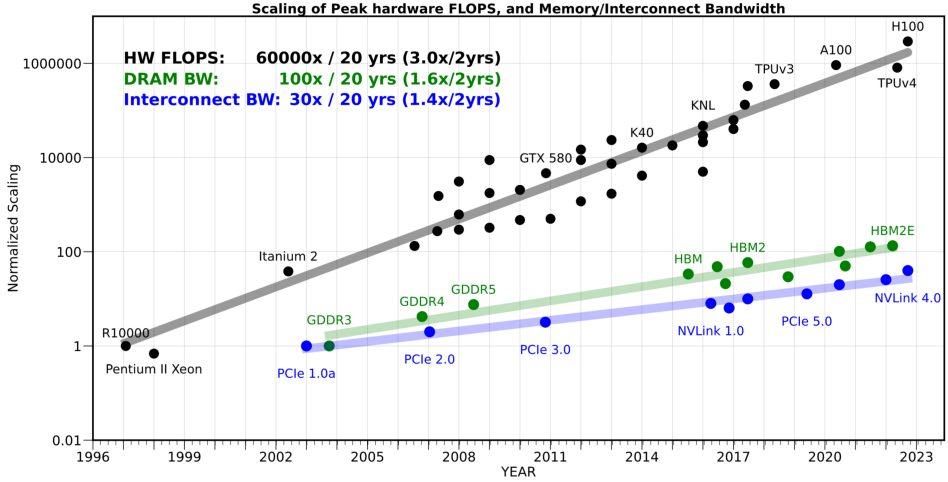
GPU performance over time



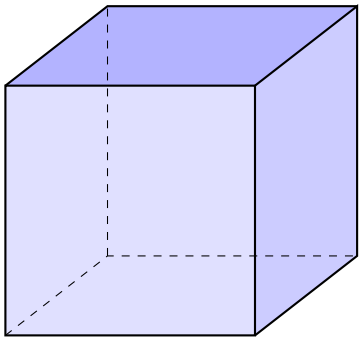


(Source: Nvidia datasheets)

Flops vs Memory / Interconnect Scaling



Growing bottleneck: Moving the data, not compute! Source: Gholami et al. (2024)

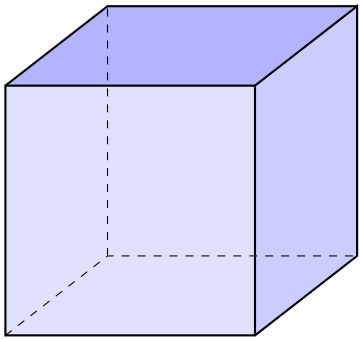


Volume: $O(N^3)$

- mass, energy capacity, compute

Surface area: $O(N^2)$

- heat removal, information transfer



Volume: $O(N^3)$

- mass, energy capacity, compute

Surface area: $O(N^2)$

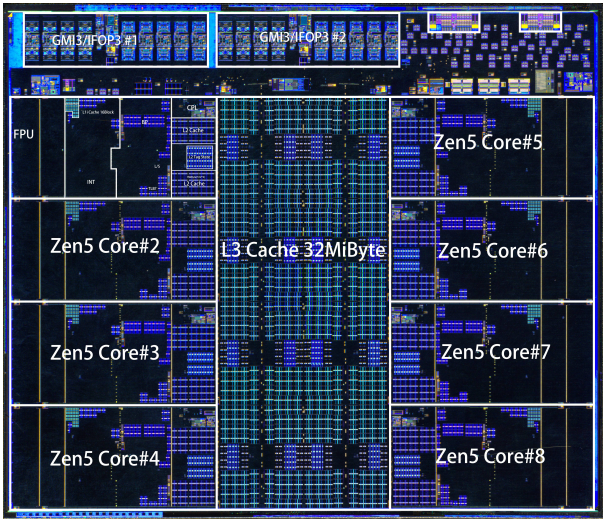
- heat removal, information transfer

$$\frac{\text{volume}}{\text{surface area}} \propto \frac{N^3}{N^2} = N$$

As systems grow, cooling and communication become relatively harder than adding compute.

$\Rightarrow$ Transfers (memory access) are likely to continue to be bottlenecks!

CPU Die Shot - AMD Zen5 CPU-Die

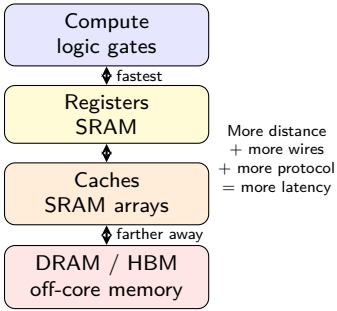


Cache: >50 % of the die

Source: Kurnal Insights [CN]

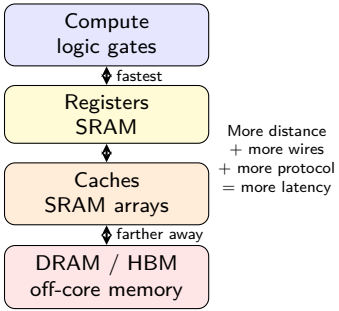
Compute

- Logic gates implement adders, multipliers, FMAs, tensor cores, control logic, etc.
- A simple operation may use only tens to thousands of transistors.
- Arithmetic is local: nearby wires, short distances, **deeply pipelined**.



Memory

- Stores bits using physical cells: SRAM, DRAM, Flash, etc.
- HBM is stacked DRAM with a very wide interface, not a separate cell type.
- Large arrays need decoders, sense amplifiers, precharge circuits, ECC, buses, controllers.
- **Access often requires moving data over longer wires than compute**.



The Memory Wall

For ML systems:

- Tensor cores can perform huge numbers of operations per second.
- But operands must arrive from registers, SRAM, HBM, host memory, or other GPUs.
- If data movement cannot keep up, compute units stall.

The Memory Wall

For ML systems:

- Tensor cores can perform huge numbers of operations per second.
- But operands must arrive from registers, SRAM, HBM, host memory, or other GPUs.
- If data movement cannot keep up, compute units stall.

$$\text{performance} \neq \text{only FLOPs}$$

The Memory Wall

For ML systems:

- Tensor cores can perform huge numbers of operations per second.
- But operands must arrive from registers, SRAM, HBM, host memory, or other GPUs.
- If data movement cannot keep up, compute units stall.

$$\text{performance} \neq \text{only FLOPs}$$

$$\text{time} \approx \max\left(\frac{\text{operations}}{\text{compute throughput}}, \frac{\text{bytes moved}}{\text{memory/interconnect bandwidth}}\right)$$

Programming Abstractions

1. High-level programming languages
2. System-level programming languages
3. Assembly (x86, Arm, ...)
4. Microarchitecture (Skylake, Zen, ...)

Abstracts away (mostly):

Programming Abstractions

1. High-level programming languages
2. System-level programming languages
3. Assembly (x86, Arm, ...)
4. Microarchitecture (Skylake, Zen, ...)

Abstracts away (mostly):

1. Memory management

Programming Abstractions

1. High-level programming languages
2. System-level programming languages
3. Assembly (x86, Arm, ...)
4. Microarchitecture (Skylake, Zen, ...)

Abstracts away (mostly):

1. Memory management
2. ISA details through the compiler

Programming Abstractions

1. High-level programming languages
2. System-level programming languages
3. Assembly (x86, Arm, ...)
4. Microarchitecture (Skylake, Zen, ...)

Abstracts away (mostly):

1. Memory management
2. ISA details through the compiler
3. Pipelines, caches, speculation, execution units, power/performance

Programming Abstractions

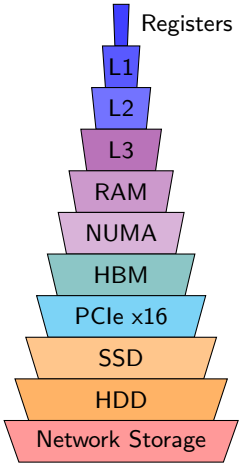
1. High-level programming languages
2. System-level programming languages
3. Assembly (x86, Arm, ...)
4. Microarchitecture (Skylake, Zen, ...)

Abstracts away (mostly):

1. Memory management
2. ISA details through the compiler
3. Pipelines, caches, speculation, execution units, power/performance

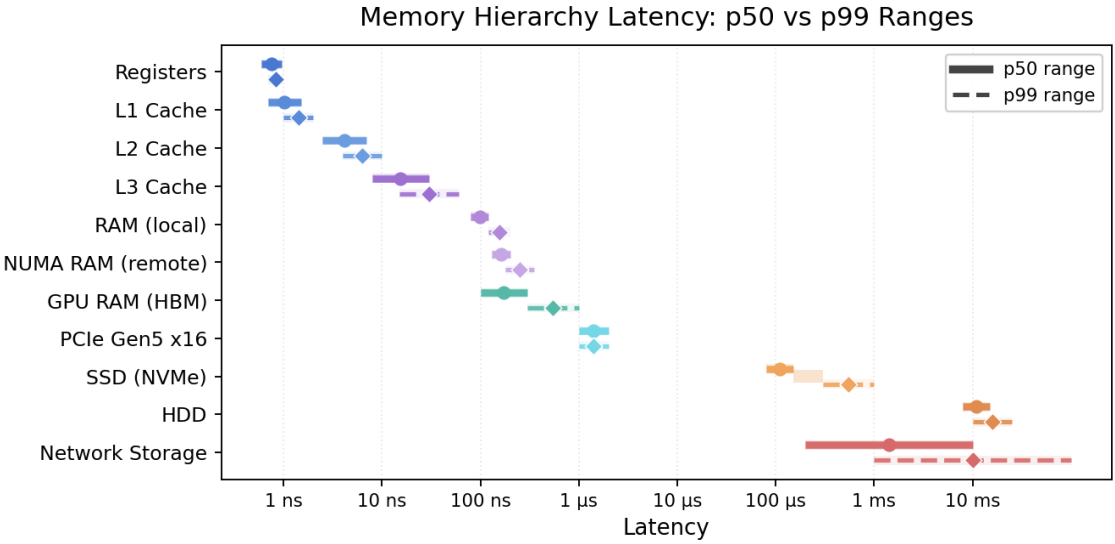
$\Rightarrow$ abstracts away (mostly) memory management: But that's the most important!

CPU-side Memory Access



Scope	Capacity	p50	p99	BW
per core	∼1–4 KiB	< 1 ns	< 1 ns	10–30+ TB/s
per core	∼32–128 KiB	0.7–1.5 ns	1–2 ns	∼1 TB/s
per core	∼0.5–2 MiB	2.5–7 ns	4–10 ns	0.5–1 TB/s
per socket	∼32–512 MiB	8–30 ns	15–60 ns	300–700 GB/s
per socket	∼512 GiB–6 TiB	80–120 ns	120–200 ns	375–615 GB/s
remote socket	∼256 GiB–3 TiB	130–200 ns	180–350 ns	100–200 GB/s
per GPU	∼80–141 GB	100–300 ns	0.3–1 µs	3.35–4.8 TB/s
per direction	n/a (interconnect)	1–2 µs	1–2 µs	55–63 GB/s
per drive	∼3.84–122.88 TB	80–150 µs	0.3–1 ms	10–14 GB/s
per drive	∼20–32 TB	8–15 ms	10–25+ ms	0.27–0.29 GB/s
per zone	PiB+	1–10 ms	1–20+ ms	15–34 GB/s

Memory Latency: p50 and p99 Ranges



Caches: Locality

Spatial locality: accessed memory objects are close to each other

- Code: inner loops
- Data: structures (reading of one field is often followed by reads of other fields)

Spatial locality: accessed memory objects are close to each other

- Code: inner loops
- Data: structures (reading of one field is often followed by reads of other fields)

Temporal locality: The same data will be used multiple times in a short period of time

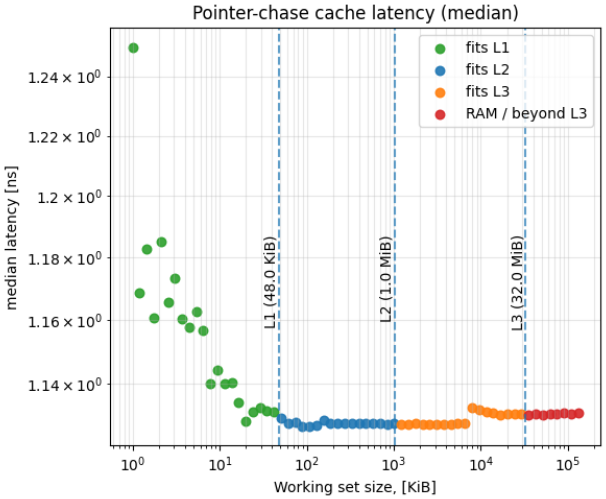
- Code: loops
- Data: e.g. digital filter coefficients are accessed every sampling period

Caches: Terminology

- Cache reference / access
- Cache hit
- Cache miss
- Cache miss rate = cache misses / total cache accesses
- Hit rate = cache hits / total cache accesses
- Cache line eviction

Caches: Linear Access

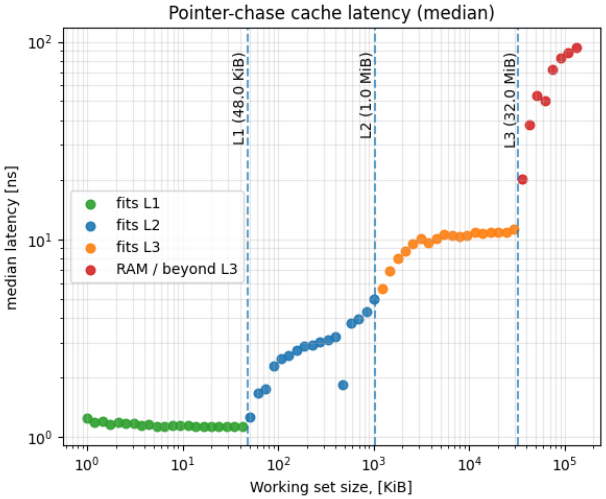
```
next_index = np.arange(1, n)

idx = 0
for _ in range(1, n):
  idx = next_index[idx]
```



Benched on AMD EPYC 9355

Caches: Random Access

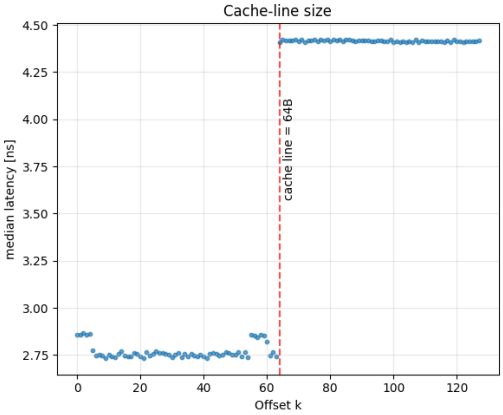
```
next_index = np.arange(1, n)
np.random.shuffle(next_index)
idx = 0
for _ in range(1, n):
  idx = next_index[idx]
```



Benched on AMD EPYC 9355

Cache lines

```
arr = np.ones(10 * l3_size, dtype=np.uint8)
assert arr.ctypes.data % 64 == 0

lines = np.arange(64, 4 * l3_size, 64)
np.random.shuffle(lines)

for k in range(128):
  for line in lines:
    _ = arr[line] + arr[line + k]
```



Benched on Ryzen 5950X

Physical and Virtual Addresses

A pointer `void* p` is usually a **virtual address**.

Physical and Virtual Addresses

A pointer `void* p` is usually a **virtual address**.

CPU translates virtual address through **page tables**.

Physical and Virtual Addresses

A pointer `void* p` is usually a **virtual address**.

CPU translates virtual address through **page tables**.

- Each process has its own virtual address space.

Physical and Virtual Addresses

A pointer `void* p` is usually a **virtual address**.

CPU translates virtual address through **page tables**.

- Each process has its own virtual address space.
- Virtual memory is divided into fixed-size **pages**.

Physical and Virtual Addresses

A pointer `void* p` is usually a **virtual address**.

CPU translates virtual address through **page tables**.

- Each process has its own virtual address space.
- Virtual memory is divided into fixed-size **pages**.
- Page tables map virtual pages to physical page frames.

Physical and Virtual Addresses

A pointer `void* p` is usually a **virtual address**.

CPU translates virtual address through **page tables**.

- Each process has its own virtual address space.
- Virtual memory is divided into fixed-size **pages**.
- Page tables map virtual pages to physical page frames.
- Memory Management Unit (MMU) is hardware unit for translating a virtual address into a physical address.

Physical and Virtual Addresses

A pointer `void* p` is usually a **virtual address**.

CPU translates virtual address through **page tables**.

- Each process has its own virtual address space.
- Virtual memory is divided into fixed-size **pages**.
- Page tables map virtual pages to physical page frames.
- Memory Management Unit (MMU) is hardware unit for translating a virtual address into a physical address.
- Translation Lookaside Buffer (TLB) caches recent virtual-to-physical address translations.

Physical and Virtual Addresses

A pointer `void* p` is usually a **virtual address**.

CPU translates virtual address through **page tables**.

- Each process has its own virtual address space.
- Virtual memory is divided into fixed-size **pages**.
- Page tables map virtual pages to physical page frames.
- Memory Management Unit (MMU) is hardware unit for translating a virtual address into a physical address.
- Translation Lookaside Buffer (TLB) caches recent virtual-to-physical address translations.

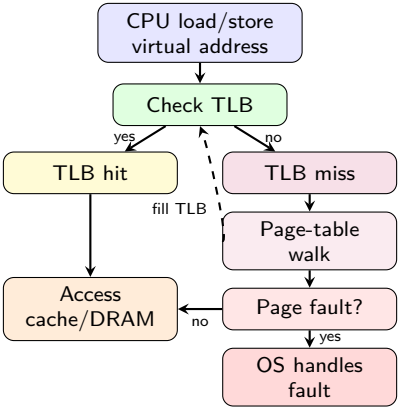
Two processes can use the same pointer value, but refer to different physical memory.

Virtual-to-Physical Mapping

On every CPU load/store, the CPU needs a virtual-to-physical translation.

Fast path: TLB hit

- Translation is already cached in the TLB.
- CPU can access cache/DRAM with little extra delay.

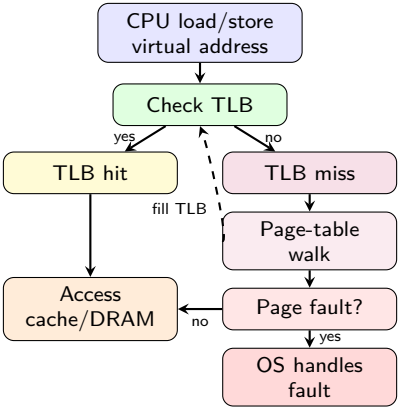


Virtual-to-Physical Mapping

On every CPU load/store, the CPU needs a virtual-to-physical translation.

Slower path: TLB miss

- CPU/MMU walks the page tables to find the mapping.
- The page-table entries themselves may need cache/DRAM reads.
- Translation is inserted into the TLB.

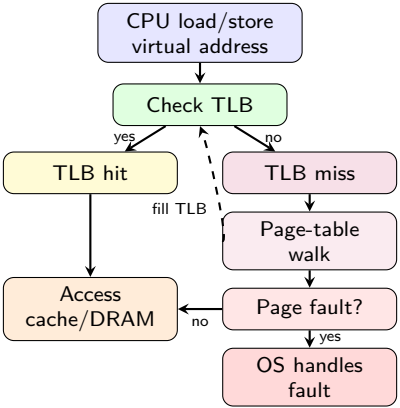


Virtual-to-Physical Mapping

On every CPU load/store, the CPU needs a virtual-to-physical translation.

Very slow path: page fault

- Mapping is missing, invalid, or not currently resident.
- CPU traps into the OS kernel.
- OS fixes the mapping, loads the page, or kills the process.



Huge Pages: Fewer Translations for Large Buffers

Normal pages are often **4 KiB**. For large buffers, that means many pages and many translations.

Huge Pages: Fewer Translations for Large Buffers

Normal pages are often **4 KiB**. For large buffers, that means many pages and many translations.

Huge pages use larger page sizes, commonly **2 MiB** Max size: Gigantic pages (1 GiB)

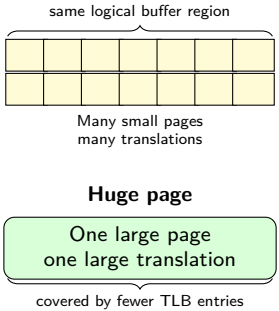
Tradeoff: higher memory consumption (potential waste) due to fragmentation and increased memory management overhead for the kernel.

Huge Pages: Fewer Translations for Large Buffers

Normal pages are often **4 KiB**. For large buffers, that means many pages and many translations.

Huge pages use larger page sizes, commonly **2 MiB** Max size: Gigantic pages (1 GiB)

Tradeoff: higher memory consumption (potential waste) due to fragmentation and increased memory management overhead for the kernel.



Latency: TLB Hit, TLB Miss, Page Fault

Event	What happens	Typical scale	Why it matters
TLB hit	Translation cached	$\sim$1 cycle	Fast path
L1/L2 TLB miss	Page-table walk, entries cached	$\sim$10s of cycles	Small but measurable
Page-table walk to DRAM	Page-table entries fetched from memory	$\sim$100s of cycles	Can stall load/store
Minor page fault	OS updates mapping, page already in RAM	$\sim$1–10 μs	Kernel trap + bookkeeping
Major page fault	OS loads page from disk/swap	$\sim$100 μs–ms+	Storage latency dominates

Latency: TLB Hit, TLB Miss, Page Fault

Event	What happens	Typical scale	Why it matters
TLB hit	Translation cached	$\sim$1 cycle	Fast path
L1/L2 TLB miss	Page-table walk, entries cached	$\sim$10s of cycles	Small but measurable
Page-table walk to DRAM	Page-table entries fetched from memory	$\sim$100s of cycles	Can stall load/store
Minor page fault	OS updates mapping, page already in RAM	$\sim$1–10 μs	Kernel trap + bookkeeping
Major page fault	OS loads page from disk/swap	$\sim$100 μs–ms+	Storage latency dominates

- A **TLB miss** is usually a hardware-managed translation-cache miss.
- A **page fault** is an exception into the OS.
- A major page fault can be **millions of cycles** slower than a TLB hit.

Latency: TLB Hit, TLB Miss, Page Fault

Event	What happens	Typical scale	Why it matters
TLB hit	Translation cached	$\sim$1 cycle	Fast path
L1/L2 TLB miss	Page-table walk, entries cached	$\sim$10s of cycles	Small but measurable
Page-table walk to DRAM	Page-table entries fetched from memory	$\sim$100s of cycles	Can stall load/store
Minor page fault	OS updates mapping, page already in RAM	$\sim$1–10 μs	Kernel trap + bookkeeping
Major page fault	OS loads page from disk/swap	$\sim$100 μs–ms+	Storage latency dominates

- A **TLB miss** is usually a hardware-managed translation-cache miss.
- A **page fault** is an exception into the OS.
- A major page fault can be **millions of cycles** slower than a TLB hit.

$$\text{TLB hit} \ll \text{TLB miss} \ll \text{minor page fault} \ll \text{major page fault}$$

DMA and RDMA

- **DMA = Direct Memory Access**
  - A device moves data to/from host memory without the CPU copying every byte.
  - Example: GPU, NIC, or NVMe device transfers data through PCIe.

DMA and RDMA

- **DMA = Direct Memory Access**
  - A device moves data to/from host memory without the CPU copying every byte.
  - Example: GPU, NIC, or NVMe device transfers data through PCIe.
- **RDMA = Remote Direct Memory Access**
  - A NIC/HCA moves data directly between memory on different machines.
  - Reduces CPU involvement, memory copies, and software overhead.

DMA and RDMA

- **DMA = Direct Memory Access**
  - A device moves data to/from host memory without the CPU copying every byte.
  - Example: GPU, NIC, or NVMe device transfers data through PCIe.
- **RDMA = Remote Direct Memory Access**
  - A NIC/HCA moves data directly between memory on different machines.
  - Reduces CPU involvement, memory copies, and software overhead.

DMA: device $\leftrightarrow$ local memory

DMA and RDMA

- **DMA = Direct Memory Access**
  - A device moves data to/from host memory without the CPU copying every byte.
  - Example: GPU, NIC, or NVMe device transfers data through PCIe.
- **RDMA = Remote Direct Memory Access**
  - A NIC/HCA moves data directly between memory on different machines.
  - Reduces CPU involvement, memory copies, and software overhead.

DMA: device $\leftrightarrow$ local memory

RDMA: memory on node A $\leftrightarrow$ network $\leftrightarrow$ memory on node B

DMA and RDMA

- **DMA = Direct Memory Access**
  - A device moves data to/from host memory without the CPU copying every byte.
  - Example: GPU, NIC, or NVMe device transfers data through PCIe.
- **RDMA = Remote Direct Memory Access**
  - A NIC/HCA moves data directly between memory on different machines.
  - Reduces CPU involvement, memory copies, and software overhead.

DMA: device $\leftrightarrow$ local memory

RDMA: memory on node A $\leftrightarrow$ network $\leftrightarrow$ memory on node B

DMA is local. RDMA extends the idea across the network.

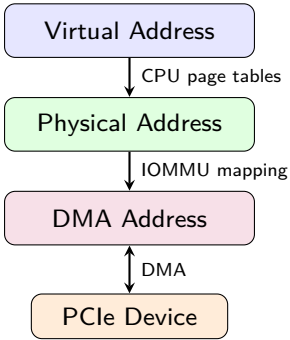
Data Movement: Address Spaces

A PCIe transfer crosses several address spaces.

- **Virtual address**: CPU process
- **Physical address**: CPU memory system
- **DMA address**: PCIe device

Important distinction:

- Ordinary CPU loads/stores in a process use **virtual addresses**.
- PCIe devices usually DMA using **DMA addresses**.



Virtual vs. Physical vs. DMA Addresses

Address type	Used by	Meaning
Virtual address	CPU process	Per-process pointer value
Physical address	CPU / memory controller	Location in system memory
DMA address	PCIe device	Address programmed into device DMA engine
GPU virtual address	GPU kernels / driver	Device-side virtual address space

- A pointer like `void* p` is usually a **CPU virtual address**.
- A PCIe device cannot safely DMA to an arbitrary user pointer.
- The OS/driver must pin pages, translate them, and program DMA mappings.

Why DMA Needs Pinned Memory

DMA lets a PCIe device read/write host memory without the CPU copying every byte.

Why DMA Needs Pinned Memory

DMA lets a PCIe device read/write host memory without the CPU copying every byte.

But normal pageable memory is unsafe for long DMA:

Why DMA Needs Pinned Memory

DMA lets a PCIe device read/write host memory without the CPU copying every byte.

But normal pageable memory is unsafe for long DMA:

- OS may page it out.

Why DMA Needs Pinned Memory

DMA lets a PCIe device read/write host memory without the CPU copying every byte.

But normal pageable memory is unsafe for long DMA:

- OS may page it out.
- OS may migrate or remap pages.

Why DMA Needs Pinned Memory

DMA lets a PCIe device read/write host memory without the CPU copying every byte.

But normal pageable memory is unsafe for long DMA:

- OS may page it out.
- OS may migrate or remap pages.
- Virtual pages may not be physically contiguous.

Why DMA Needs Pinned Memory

DMA lets a PCIe device read/write host memory without the CPU copying every byte.

But normal pageable memory is unsafe for long DMA:

- OS may page it out.
- OS may migrate or remap pages.
- Virtual pages may not be physically contiguous.
- Device needs stable DMA mappings during the transfer.

Why DMA Needs Pinned Memory

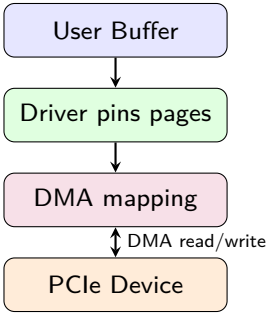
DMA lets a PCIe device read/write host memory without the CPU copying every byte.

But normal pageable memory is unsafe for long DMA:

- OS may page it out.
- OS may migrate or remap pages.
- Virtual pages may not be physically contiguous.
- Device needs stable DMA mappings during the transfer.

Pinned memory gives the driver stable resident pages and DMA mappings for the duration of the transfer or registration.

Pinning too much memory is harmful: it reduces OS flexibility and can hurt system performance.



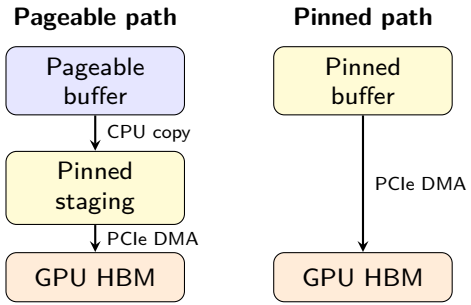
Pageable vs. Pinned H2D Copy

Pageable host memory

- Application buffer can move/page.
- Runtime may stage through temporary pinned memory.
- Extra copy can reduce bandwidth.
- Less predictable for async transfer.

Pinned host memory

- Pages stay resident.
- DMA engine can access stable mappings.
- Better for large PCIe transfers.
- Better for overlapping copy and compute.



How components communicate?

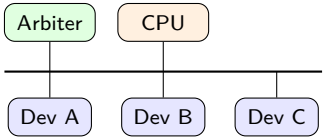
How do hardware components communicate via DMA?

Next: interconnects (PCIe, NVLink, InfiniBand, Ethernet)

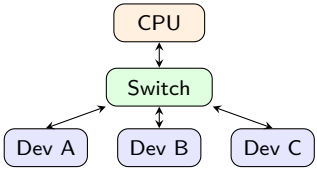
Interconnect

- Hardware/protocol system used to move data between components.
- Includes **physical layer**: wires/traces/lanes, connectors, PHY/SerDes.
- Includes **protocol layer**: addressing, routing, packets, ordering, flow control.

Classic shared bus



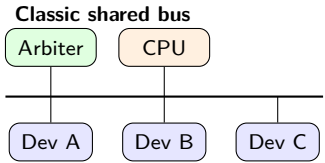
Modern switched interconnect



Interconnect vs. Bus

Bus

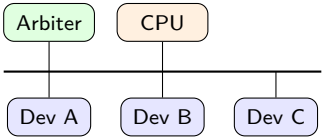
- One specific kind of interconnect with a **shared channel**.
- Multiple devices can contend for the same channel.
- Contention and arbitration often limit scalability.



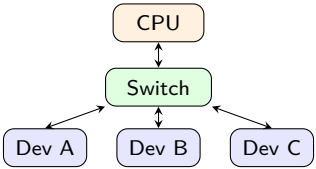
Interconnect vs. Bus Arbitration

- A bus **arbiter** decides who owns the shared bus;
- A (PCIe) **switch** routes packets across independent links and only arbitrates when multiple packets want the same output port.

Classic shared bus



Modern switched interconnect (PCIe-like, simplified)



Interconnects for ML Systems

	PCIe	**NVLink**	**InfiniBand**	**Ethernet**
Scope	Inside one node	Inside one node / GPU island	Between nodes	Between nodes / data center
Main role	General local I/O	GPU-to-GPU	HPC / AI cluster	General network
Bandwidth	∼63 GB/s one direction for PCIe 5.0 x16	100s of GB/s per GPU	200–800 Gb/s	25–400 GbE
ML use	CPU–GPU, GPU–NIC, NVMe	Fast intra-node GPU collectives	Cross-node GPU collectives	Storage, services, ML traffic

GPU ↔ **NVLink**/**PCIe** ↔ NIC/HCA ↔ **Ethernet**/**InfiniBand** ↔ remote NIC/HCA ↔ **PCIe**/**NVLink** ↔ GPU

Collectives: Optimized Operations Across GPUs

Collective	What it does	Used for
broadcast	One GPU sends the same data to all others	Sharing parameters/config/state
reduce	Many GPUs send values that get combined into one result	Summing losses or metrics
all-reduce	Every GPU contributes data, and every GPU receives the combined result	Gradient synchronization in data parallel training
scatter	One GPU splits data and sends different chunks to different GPUs	Distributing shards
gather	Many GPUs send chunks to one GPU	Reassembling shards
all-gather	Every GPU shares its chunk, and every GPU receives all chunks	Tensor/model parallelism
reduce-scatter	Values are reduced, then the reduced result is split across GPUs	Efficient distributed training

Ethernet: Interconnect

- **Ethernet** is the dominant general-purpose networking technology.
- Used in data centers, cloud networks, enterprise networks, storage, and the internet.
- Modern Ethernet supports very high link rates: 100G, 200G, 400G, and 800G.
- Unlike PCIe, Ethernet connects **systems across a network**, not devices inside one host.

Ethernet: Interconnect

- **Ethernet** is the dominant general-purpose networking technology.
- Used in data centers, cloud networks, enterprise networks, storage, and the internet.
- Modern Ethernet supports very high link rates: 100G, 200G, 400G, and 800G.
- Unlike PCIe, Ethernet connects **systems across a network**, not devices inside one host.

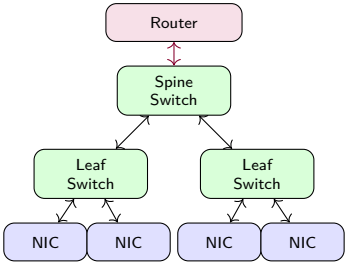
For ML clusters, Ethernet can carry:

- GPU-to-GPU communication across nodes
- Distributed collectives: all-reduce, all-gather, reduce-scatter
- Storage traffic
- Control-plane and service traffic

Ethernet: Topology

Ethernet is organized as a **switched network**.

- **NIC** = Network Interface Card
- **Switch** = forwards Ethernet frames between ports
- **Router** = forwards IP packets between networks
- Data centers commonly use leaf-spine topologies
- AI Ethernet fabrics may use multiple rails for higher aggregate bandwidth



Ethernet: Bandwidth

Standard	IEEE standard year	Decimal raw equivalent
100MbE	802.3u, 1995	12.5 MB/s
1GbE	802.3z, 1998	125 MB/s
10GbE	802.3ae, 2002	1.25 GB/s
25GbE	802.3by, 2016	3.125 GB/s
50GbE	802.3cd, 2018	6.25 GB/s
100GbE	802.3ba, 2010	12.5 GB/s
200GbE	802.3bs, 2017	25 GB/s
400GbE	802.3bs, 2017	50 GB/s
800GbE	802.3df, 2024	100 GB/s

- Nominal link rate is not the same as application-level bandwidth.
- Large messages are usually bandwidth-bound; small messages are latency-bound.

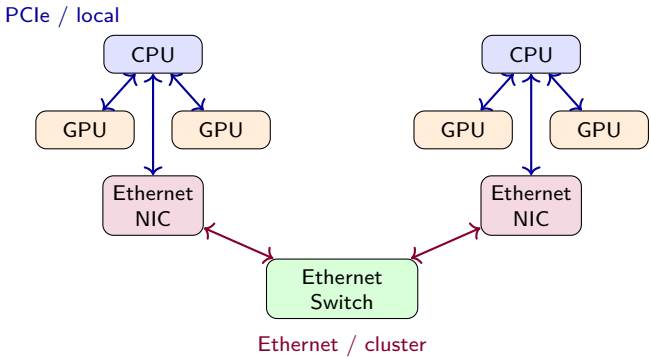
Ethernet: Protocol Stack

Ethernet can carry several different communication styles.

Stack	Typical use	Notes
Ethernet + IP + TCP	General networking	Reliable, widely supported
Ethernet + IP + UDP	Low-overhead datagrams	Used by many distributed systems
Ethernet + RoCEv2	RDMA over UDP/IP/Ethernet	Used for AI/HPC-style traffic; needs ca
Ethernet + storage protocols	NVMe/TCP, NFS, etc.	Common for storage traffic

$$\text{Ethernet} \rightarrow \text{IP} \rightarrow \text{TCP/UDP/RoCE} \rightarrow \text{application}$$

Ethernet: ML Cluster



With RoCE and GPUDirect RDMA support, Ethernet can carry GPU communication with reduced CPU overhead.

InfiniBand: an Ethernet Alternative

Ethernet's primary design goal:

- Universal interoperability and flexibility.
- Best-effort by default: Ethernet does not guarantee delivery, latency, or ordering without higher-layer protocols or extensions.

InfiniBand: an Ethernet Alternative

Ethernet's primary design goal:

- Universal interoperability and flexibility.
- Best-effort by default: Ethernet does not guarantee delivery, latency, or ordering without higher-layer protocols or extensions.

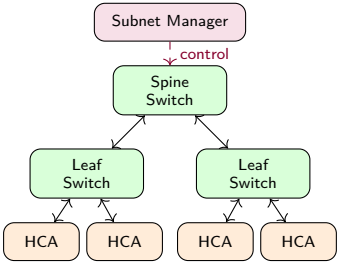
InfiniBand's primary design goal:

- Low latency, high throughput, and scalable many-node communication.
- Designed to avoid packet loss in normal operation, unlike classic Ethernet.
- Widely used in **HPC** and **large AI training clusters**, where cross-node synchronization cost is critical.
- Core capability: **RDMA** (Remote Direct Memory Access), which enables direct memory transfers with minimal CPU overhead.

InfiniBand Topology

InfiniBand is organized as a **switched fabric**.

- **Fabric** = switches, links, HCAs, routing, subnet management
- **HCA** = Host Channel Adapter; the InfiniBand NIC
- **Switch** = forwards packets between ports
- **Subnet Manager** = discovers topology and programs forwarding tables
- Topologies: fat-tree, dragonfly, torus, and rail-optimized AI fabrics
- **Rails** = multiple NIC/fabric paths used to increase aggregate bandwidth and reduce contention



InfiniBand: Bandwidth

Generation	Per-port rate	Decimal raw equivalent
EDR (2014)	100 Gb/s	12.5 GB/s
HDR (2016)	200 Gb/s	25 GB/s
NDR (2021)	400 Gb/s	50 GB/s
XDR (2023)	800 Gb/s	100 GB/s

- At the same nominal rate, raw bandwidth is similar to Ethernet.
- The difference is usually latency, congestion behavior, RDMA maturity, topology, and software stack.
- Application-level bandwidth is lower than raw link bandwidth.

In distributed training, both fabrics can carry the same logical traffic:

GPU gradients $\rightarrow$ all-reduce / reduce-scatter / all-gather $\rightarrow$ updated model state

Concern	InfiniBand	Ethernet
Raw bandwidth	100–800 Gb/s class	100–800 Gb/s class
Latency	Usually lower	Competitive when tuned
Congestion	HPC-oriented controls	Requires careful tuning
Operations	Specialized fabric	Familiar data-center operations
Cost/flexibility	Specialized	Broader ecosystem

Same headline bandwidth does not imply same training performance.

Why PCIe Comes Next

- DMA and pinned memory are the software-visible side of PCIe data movement.
- PCIe is the local interconnect that usually carries host-memory $\leftrightarrow$ GPU, NIC, and NVMe transfers.
- So before comparing cluster fabrics, we need the local device path.

$$\text{CPU memory} \leftrightarrow \text{PCIe} \leftrightarrow \text{GPU} \;/\; \text{NIC} \;/\; \text{NVMe}$$

PCIe: Interconnect

- **PCIe** is point-to-point, packet-based, full-duplex, tree-like interconnect.
- A **link** has multiple **lanes**: x1, x4, x8, **x16** ($\Rightarrow$ more lanes, more bandwidth).
- Initialization cycle auto-negotiates link speed and link width.
- Slot's physical size and actual wiring may differ.
- Can support hot-plug/hot-swap when the platform, slot, device, and OS support it.

For ML training nodes, PCIe is commonly the host path for:

- SSD / NIC $\leftrightarrow$ host memory
- host memory $\leftrightarrow$ GPU HBM (H2D / D2H copies)

PCIe: Communication

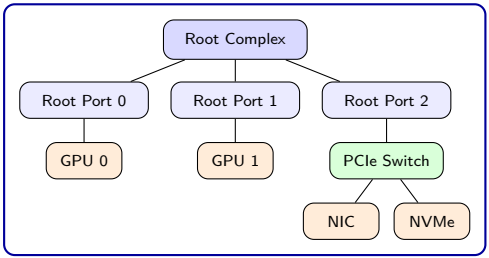
- **Full duplex**: each link can send and receive simultaneously (both H2D and D2H!).
- Not one global bus: different links/ports can carry traffic concurrently.
- PCIe communication is **packet-based**.
- On one link and in one direction, packets are serialized into an ordered stream.
- Across different links/ports, many packets can move concurrently.
- PCIe supports **pipelined split transactions**: many requests may be in flight before completions return.

PCIe: Topology

PCIe is usually organized as a **tree topology**.

- **Fabric** = particular PCIe hierarchy: ports, links, switches, endpoints, and root
- **Root Complex** = bridge between CPU/memory system and PCIe hierarchy
- **Root Port** = downstream-facing PCIe port on the Root Complex
- **Port** = endpoint of one point-to-point PCIe link (upstream or downstream)
- **Switch** = one upstream port + multiple downstream ports
- **Endpoint** = leaf PCIe device

Fabric



PCIe: Bandwidth

Gen	Rate/lane	x16 theoretical, one direction	Typical measured copy payload
PCIe 3.0 (2010)	8 GT/s	$\sim$ 15.8 GB/s	$\sim$ 12–14 GB/s
PCIe 4.0 (2017)	16 GT/s	$\sim$ 31.5 GB/s	$\sim$ 24–28 GB/s
PCIe 5.0 (2019)	32 GT/s	$\sim$ 63.0 GB/s	$\sim$ 48–56 GB/s
PCIe 6.0 (2022)	64 GT/s	$\sim$ 121–128 GB/s	workload/protocol/platform dependent

- **Practical payload** is lower due to protocol overhead, packetization, and software stack effects.
- NUMA placement matters: wrong socket can reduce effective bandwidth and increase latency.

PCIe: Single-stream pinned-memory H2D copy benchmark

CUDA C++ implementation

1. Allocate pinned host buffer (`cudaMallocHost`).
2. Allocate device buffer once (`cudaMalloc`).
3. Create non-blocking stream (`cudaStreamCreateWithFlags`).
4. Warmup with async H2D copies (`cudaMemcpyAsync`).
5. Time measured loop with one stream synchronized after all iterations.
6. Report GB/s throughput.

```cpp
size_t nbytes = 256ull << 20;  // 256 MiB
int warmup = 8, iters = 128;
uint8_t* h = nullptr; uint8_t* d = nullptr;
cudaStream_t s;

cudaMallocHost(reinterpret_cast<void**>(&h), nbytes); // pinned
cudaMalloc(reinterpret_cast<void**>(&d), nbytes);
cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
memset(h, 0xA5, nbytes);  // Fill with a test value

for (int i = 0; i < warmup; ++i) {
  cudaMemcpyAsync(d, h, nbytes, cudaMemcpyHostToDevice, s);
  cudaStreamSynchronize(s);
}

auto t0 = std::chrono::high_resolution_clock::now();
for (int i = 0; i < iters; ++i) {
  cudaMemcpyAsync(d, h, nbytes, cudaMemcpyHostToDevice, s);
}
cudaStreamSynchronize(s);
auto t1 = std::chrono::high_resolution_clock::now();
double dt = std::chrono::duration<double>(t1 - t0).count();
double gbps = (double(nbytes) * iters) / (dt * 1e9);
std::cout << "H2D throughput: " << gbps << " GB/s\n";
```

Source code: mlprague2026/labs/pcie-transfer

NVLink: Interconnect

- **NVLink** is NVIDIA's high-bandwidth, low-latency interconnect for GPU-centric systems.
- Like PCIe, it is **point-to-point**, **packet-based**, and **full-duplex**.
- NVLink is optimized mainly for:
  - GPU ↔ GPU communication
  - GPU ↔ CPU communication in Grace-Hopper / Grace-Blackwell systems
  - Multi-GPU collectives and model-parallel workloads
- NVLink is **not** a general expansion bus for SSDs/NICs/peripherals.

PCIe model

Link = one or more lanes
Width = x1, x4, x8, x16

NVLink model

Each GPU has multiple NVLink ports. Aggregate bandwidth is the sum across active NVLink connections.

$$\mathrm{BW}_{\mathrm{agg}} \approx \sum_{i} \mathrm{BW}_{\mathrm{link},i}$$

NVLink: ML Traffic

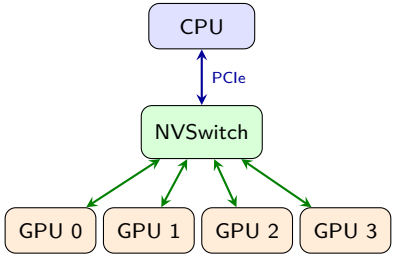
NVLink is commonly the fast path for GPU-to-GPU traffic in training and inference:

- **HBM $\leftrightarrow$ HBM transfers**: activations, tensor-parallel shards, KV-cache movement
- **Parallelism traffic**: tensor, pipeline, and expert-parallel routing
- **Collectives**: graph operations like all-reduce, all-gather, reduce-scatter, etc.

NVLink: Topology

NVLink topologies are more GPU-centric than PCIe.

- **Direct NVLink**: GPUs are connected by dedicated high-speed links.
- **NVSwitch**: switch fabric that gives many GPUs high-bandwidth all-to-all connectivity.
- **NVLink domain**: group of GPUs connected through NVLink/NVSwitch.
- PCIe may still exist in the same node for boot, control, NICs, SSDs, and host access.



NVLink/NVSwitch fabric

Key idea:

- PCIe topology is usually **CPU-rooted**.
- NVLink topology is usually **GPU-fabric-rooted**.

NVLink: Communication

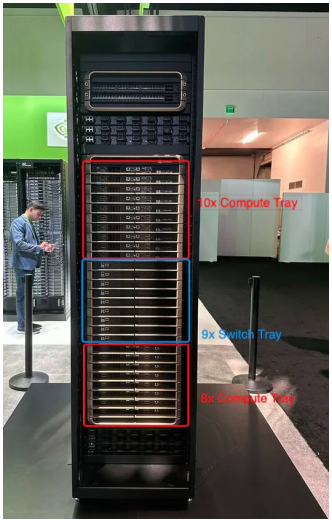
- **Full duplex**: GPUs can send and receive simultaneously.
- **Peer-to-peer**: GPU memory can be accessed by another GPU without staging through host memory.
- **Higher bandwidth than PCIe**: especially important for multi-GPU tensor movement.
- **Lower CPU involvement**: GPU-to-GPU traffic can avoid the CPU/root complex.
- **Collective-friendly**: NVSwitch systems are designed for many simultaneous GPU communication flows.

NVLink: Bandwidth

Platform / generation	Agg. bidir NVLink bandwidth/GPU	Notes
V100 / NVLink 2	$\sim$ 300 GB/s	DGX-1/DGX-2 era
A100 / NVLink 3	$\sim$ 600 GB/s	common in HGX A100
H100 / NVLink 4	$\sim$ 900 GB/s	Hopper generation
B200 / NVLink 5	$\sim$ 1.8 TB/s	Blackwell generation
Rubin / NVLink 6	$\sim$ 3.6 TB/s	announced / future platform

- For reference: PCIe 5.0 x16 is $\sim$ 63 GB/s one direction.
- These are aggregate GPU interconnect bandwidth figures, not host-memory copy bandwidth.
- Exact usable bandwidth depends on topology, GPU model, software stack, transfer size, and communication pattern.
- NVLink bandwidth is most useful when GPU-to-GPU communication is on the critical path.

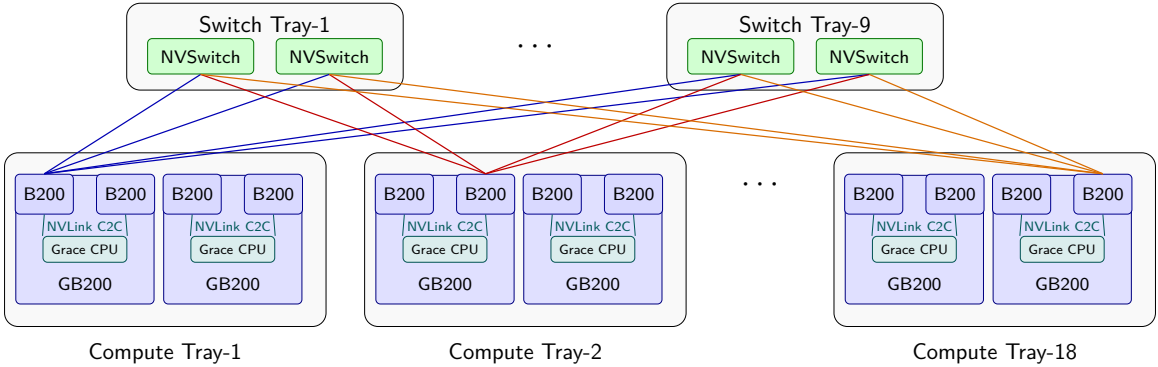
NVIDIA GB200 NVL72 at GTC



Source: naddod.com

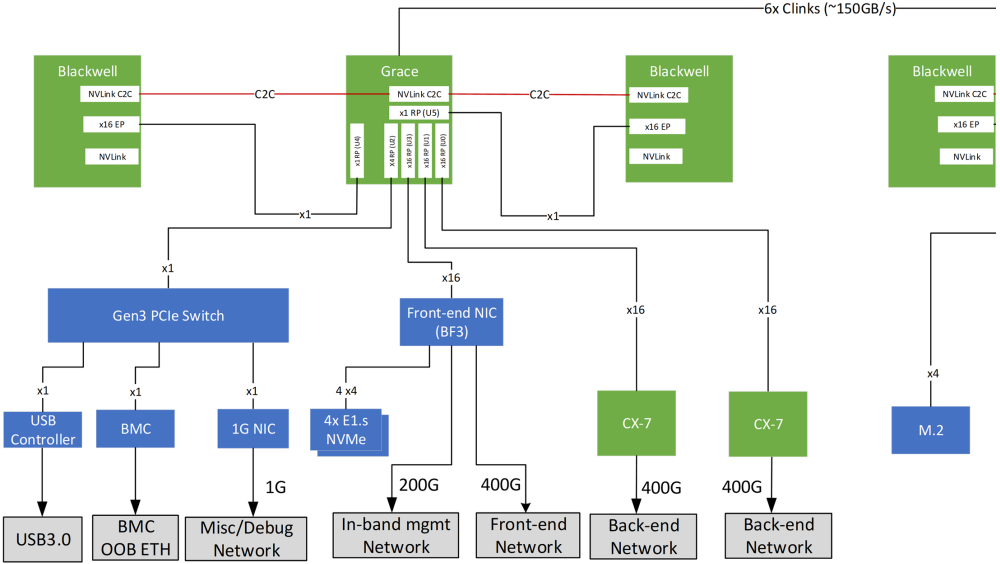
- 72 Blackwell GPUs + 36 Grace CPUs
- GPU HBM memory: up to 15.5 TB
- Total marketed NVLink bandwidth: 130 TB/s
- FP16 360 PFLOPs, FP4 1440 PFLOPs
- Power: $\sim 120 - 132$ kW/rack
- Price: $\sim$ \$3 million

NVIDIA GB200 NVL72: 72-GPU NVLink Domain



- Total 72 B200 GPUs + 36 Grace CPUs.
- Each GPU connects into the NVLink switch fabric with 18 NVLinks.
- Grace CPUs connect to paired Blackwell GPUs through NVLink-C2C.

NVIDIA GB200 NVL72: Other parts of the system



Questions?

Profiling Tools

Estimated time: 5+10 min

Tools we will cover

- VizTracer
- NVIDIA Nsight Systems
- NVIDIA Nsight Compute

VizTracer

- **Tracing profiler** for Python that records function calls, durations, and custom events.

VizTracer

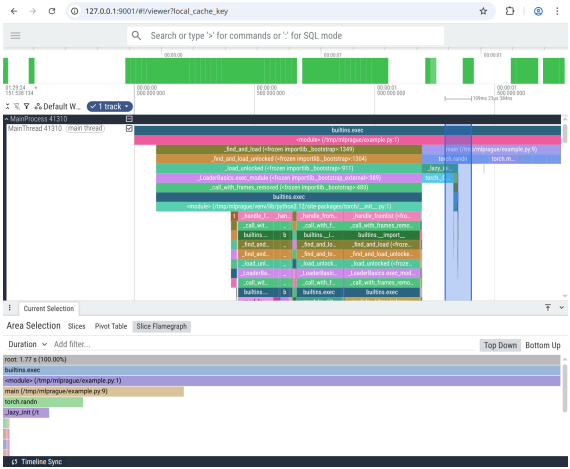
- **Tracing profiler** for Python that records function calls, durations, and custom events.
- **Timeline-first view**: interactive trace helps connect Python control flow with runtime phases.

VizTracer

- **Tracing profiler** for Python that records function calls, durations, and custom events.
- **Timeline-first view**: interactive trace helps connect Python control flow with runtime phases.
- **Simple workflow**: run with `viztracer train.py` and inspect results in browser-based viewer.

VizTracer

- **Tracing profiler** for Python that records function calls, durations, and custom events.
- **Timeline-first view**: interactive trace helps connect Python control flow with runtime phases.
- **Simple workflow**: run with `viztracer train.py` and inspect results in browser-based viewer.
- **Output**: `.json` trace files compatible with Perfetto/Chrome trace-style analysis tools.



Nsight Systems

- **System-wide timeline profiler**: correlates CPU threads, CUDA APIs, kernels, memory transfers, and OS runtime events.

Nsight Systems

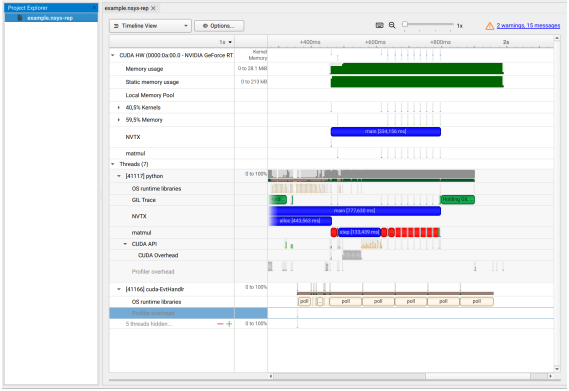
- **System-wide timeline profiler**: correlates CPU threads, CUDA APIs, kernels, memory transfers, and OS runtime events.
- **Primary goal**: find *where time goes* across the full application pipeline, not just inside one kernel.

Nsight Systems

- **System-wide timeline profiler**: correlates CPU threads, CUDA APIs, kernels, memory transfers, and OS runtime events.
- **Primary goal**: find *where time goes* across the full application pipeline, not just inside one kernel.
- **Typical outputs**: `.nsys-rep` timeline report, SQLite export, and summary statistics for scripting.

Nsight Systems

- **System-wide timeline profiler**: correlates CPU threads, CUDA APIs, kernels, memory transfers, and OS runtime events.
- **Primary goal**: find *where time goes* across the full application pipeline, not just inside one kernel.
- **Typical outputs**: `.nsys-rep` timeline report, SQLite export, and summary statistics for scripting.
- **Best use cases**: pipeline bottlenecks, overlap gaps (CPU/GPU), launch latency, synchronization stalls, data-loader inefficiencies.



Nsight Compute

- **Kernel-level performance profiler**: deep analysis of a single CUDA kernel on one GPU architecture.

Nsight Compute

- **Kernel-level performance profiler**: deep analysis of a single CUDA kernel on one GPU architecture.
- **Primary goal**: explain *why a kernel is slow* using occupancy, memory throughput, and instruction metrics.

Nsight Compute

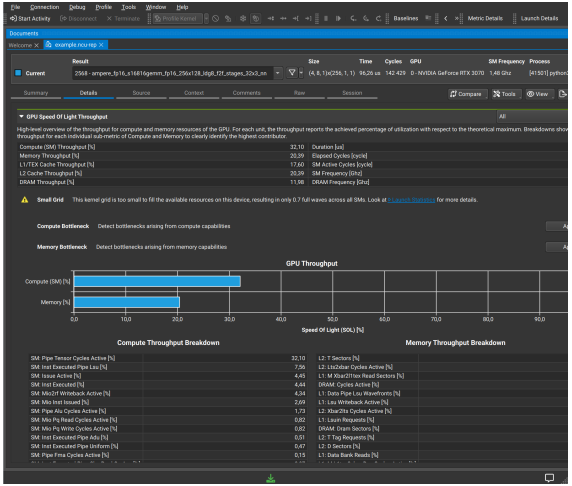
- **Kernel-level performance profiler**: deep analysis of a single CUDA kernel on one GPU architecture.
- **Primary goal**: explain *why a kernel is slow* using occupancy, memory throughput, and instruction metrics.
- **Typical outputs**: section-based reports (`SpeedOfLight`, memory, scheduler, roofline) and source/SASS correlation.

Nsight Compute

- **Kernel-level performance profiler**: deep analysis of a single CUDA kernel on one GPU architecture.
- **Primary goal**: explain *why a kernel is slow* using occupancy, memory throughput, and instruction metrics.
- **Typical outputs**: section-based reports (`SpeedOfLight`, memory, scheduler, roofline) and source/SASS correlation.
- **Best use cases**: low occupancy, memory bottlenecks, warp stalls, register pressure, and launch configuration tuning.

Nsight Compute

- **Kernel-level performance profiler**: deep analysis of a single CUDA kernel on one GPU architecture.
- **Primary goal**: explain *why a kernel is slow* using occupancy, memory throughput, and instruction metrics.
- **Typical outputs**: section-based reports (`SpeedOfLight`, memory, scheduler, roofline) and source/SASS correlation.
- **Best use cases**: low occupancy, memory bottlenecks, warp stalls, register pressure, and launch configuration tuning.
- **Practical workflow**: find hot kernels in Nsight Systems first, then profile only those kernels in Nsight Compute.



Coffee Break

Install the tools, if you haven't already!

https://mlprague2026.minfx.ai/

Programming with Hardware in Mind

Estimated time: 25 min

Programming with Hardware in Mind

- Memory layout: SoA vs AoS
- Memory alignment
- Striding
- Matmuls and caches
- Study

SoA vs AoS: Representation

Representation in memory matters. Two approaches:

SoA vs AoS: Representation

Representation in memory matters. Two approaches:

AoS (Array of Structures)

- Single array of structures
- Each structure is contiguous in memory

```
@dataclass
class Event:
  clicked: float
  weight: float

events = [Event(clicked=1.0, weight=0.1),
          Event(clicked=0.0, weight=0.2),
          Event(clicked=1.0, weight=0.3),
          Event(clicked=0.0, weight=0.4),
          Event(clicked=1.0, weight=0.5)]
```

In memory: `AB AB AB AB AB`

SoA vs AoS: Representation

Representation in memory matters. Two approaches:

AoS (Array of Structures)

- Single array of structures
- Each structure is contiguous in memory

```
@dataclass
class Event:
  clicked: float
  weight: float

events = [Event(clicked=1.0, weight=0.1),
          Event(clicked=0.0, weight=0.2),
          Event(clicked=1.0, weight=0.3),
          Event(clicked=0.0, weight=0.4),
          Event(clicked=1.0, weight=0.5)]
```

In memory: `AB AB AB AB AB`

SoA (Struct of Arrays)

- Separate arrays for each field
- Each field is contiguous in memory

```
clickeds = [1.0, 0.0, 1.0, 0.0, 1.0]
weights = [0.1, 0.2, 0.3, 0.4, 0.5]
```

In memory: `AAAAA BBBBB`

Example: we want to compute the weighted click rate.

$$\frac{\sum_{i=1}^{n} \text{weight}_i \ * \ \text{clicked}_i}{\sum_{i=1}^{n} \text{weight}_i}$$

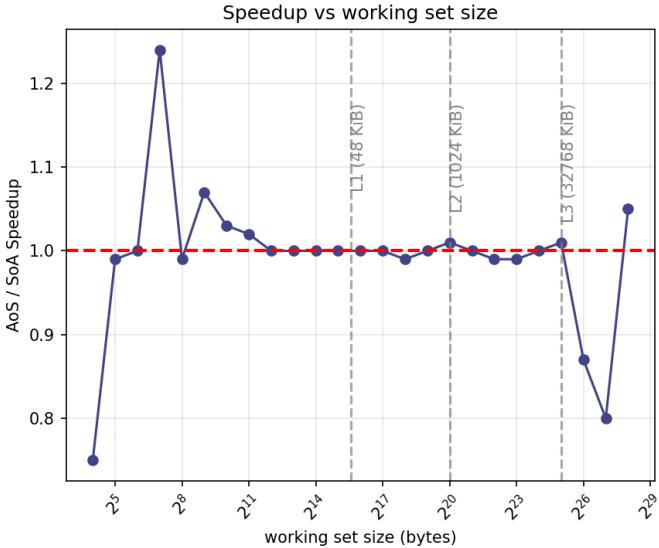
AoS

```
wclick = sum(e.clicked * e.weight for e in events)
wsum = sum(e.weight for e in events)
avg_weighted = wclick / wsum
```

SoA

```
wclick = sum(c * w for c, w in zip(clickeds, weights))
wsum = sum(weights)
avg_weighted = wclick / wsum
```

SoA vs AoS: Benchmark processing Event



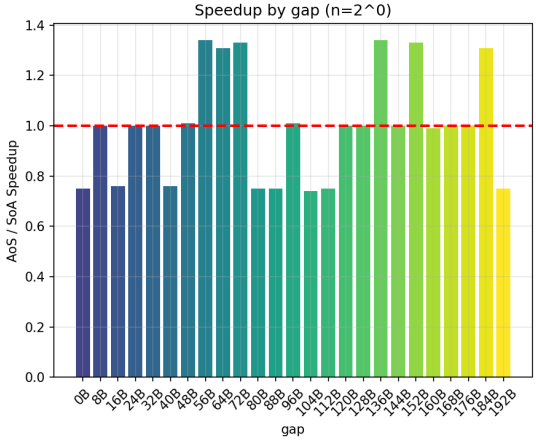
SoA vs AoS: Extra fields

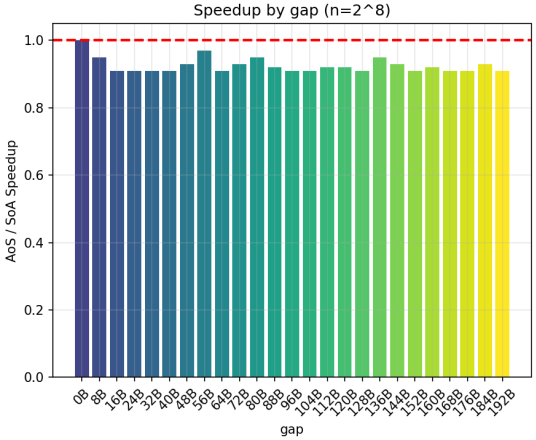
However, we rarely have just exactly two fields.

```
typedef struct {
  double clicked;
#if NUM_ATTRS > 0
  double extra_attrs[NUM_ATTRS];
#endif
  double weight;
} Event;
```

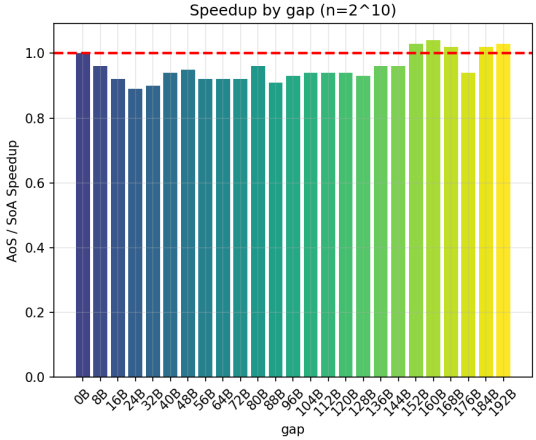
Gap between clicked and weight is 8 bytes * `NUM_ATTRS`

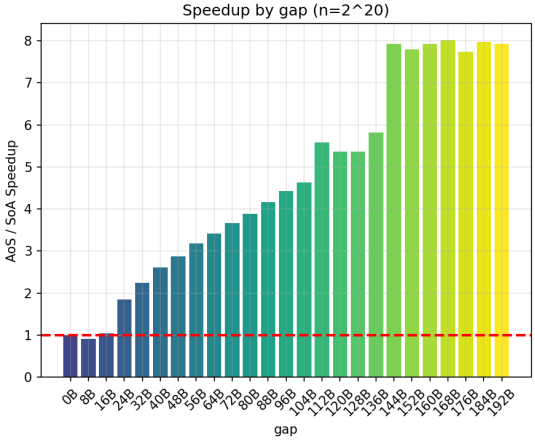
SoA vs AoS: Benchmark with extra fields



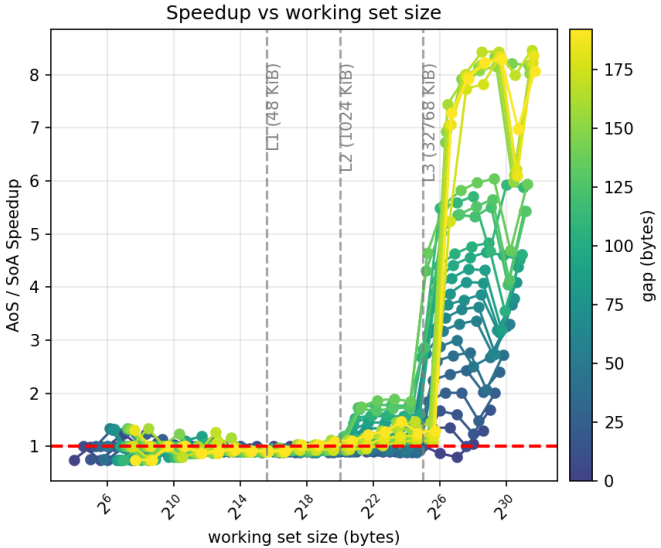


SoA vs AoS: Benchmark with extra fields





SoA vs AoS: Benchmark sweep



SoA vs AoS: Languages

Tradeoff: Usually SoA ergonomics are worse than AoS.

Native SoA support: Odin (2016)

```
package main

Vec3 :: struct { x, y, z: f32 }

main :: proc() {
    // AoS: [x y z] [x y z] [x y z] [x y z] [x y z]
    aos: [5]Vec3
    aos[0] = Vec3{1, 2, 3}

    // SoA: [x x x x x] [y y y y y] [z z z z z]
    soa: #soa[5]Vec3
    soa[0] = Vec3{1, 2, 3}

    soa.x[1] = 10 // access field array directly
}
```

Alignment: page and cache

Rule of thumb: cache-line alignment for hot structs (typically 64B), page alignment for buffers (typically 4096B).

Cache line (64B)

```c
// C
typedef struct __attribute__((aligned(64))) {
  double x[8];
} CacheHot;

// C++
struct alignas(64) CacheHotCpp {
  double x[8];
};

// Rust
#[repr(C, align(64))]
struct CacheHotRust {
    x: [f64; 8],
}
```

Page (4096B)

```c
// C (POSIX)
void* p = NULL;
posix_memalign(&p, 4096, bytes);

// C++17
void* p2 = std::aligned_alloc(4096, bytes);

// Rust
use std::alloc::{alloc, Layout};
let layout = Layout::from_size_align(bytes, 4096)?;
let p3 = unsafe { alloc(layout) };
```

Dot product in C: scalar baseline

Baseline (portable, no SIMD assumptions):

```c
#include <stddef.h>

float dot_scalar(const float* a, const float* b, size_t n) {
  float acc = 0.0f;
  for (size_t i = 0; i < n; ++i) {
    acc += a[i] * b[i];
  }
  return acc;
}
```

Optimization idea: if alignment is guaranteed, switch to wider SIMD loads.

Dot product SIMD128 (SSE, 16B alignment)

```cpp
#include <immintrin.h>

float dot_sse128_aligned(const float* a, const float* b, size_t n) {
  __m128 acc = _mm_setzero_ps();
  size_t i = 0;
  for (; i + 4 <= n; i += 4) {
    __m128 va = _mm_load_ps(a + i);   // needs 16B alignment
    __m128 vb = _mm_load_ps(b + i);   // needs 16B alignment
    acc = _mm_add_ps(acc, _mm_mul_ps(va, vb));
  }
  float tmp[4];
  _mm_storeu_ps(tmp, acc);
  float sum = tmp[0] + tmp[1] + tmp[2] + tmp[3];
  for (; i < n; ++i) sum += a[i] * b[i];
  return sum;
}
```

Dot product SIMD256 (AVX2, 32B alignment)

```cpp
#include <immintrin.h>

float dot_avx256_aligned(const float* a, const float* b, size_t n) {
  __m256 acc = _mm256_setzero_ps();
  size_t i = 0;
  for (; i + 8 <= n; i += 8) {
    __m256 va = _mm256_load_ps(a + i);   // needs 32B alignment
    __m256 vb = _mm256_load_ps(b + i);   // needs 32B alignment
    acc = _mm256_add_ps(acc, _mm256_mul_ps(va, vb));
  }
  float tmp[8];
  _mm256_storeu_ps(tmp, acc);
  float sum = tmp[0] + tmp[1] + tmp[2] + tmp[3]
            + tmp[4] + tmp[5] + tmp[6] + tmp[7];
  for (; i < n; ++i) sum += a[i] * b[i];
  return sum;
}
```

Dot product SIMD512 (AVX-512, 64B alignment)

```
#include <immintrin.h>

float dot_avx512_aligned(const float* a, const float* b, size_t n) {
  __m512 acc = _mm512_setzero_ps();
  size_t i = 0;
  for (; i + 16 <= n; i += 16) {
    __m512 va = _mm512_load_ps(a + i);   // needs 64B alignment
    __m512 vb = _mm512_load_ps(b + i);   // needs 64B alignment
    acc = _mm512_add_ps(acc, _mm512_mul_ps(va, vb));
  }
  float sum = _mm512_reduce_add_ps(acc);
  for (; i < n; ++i) sum += a[i] * b[i];
  return sum;
}
```

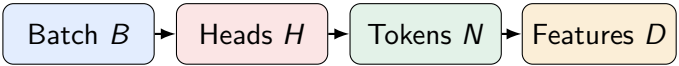
Prefer runtime dispatch: AVX-512 → AVX2 → SSE → scalar.

A tensor is a multi-dimensional array.

$$X \in \mathbb{R}^{B \times H \times N \times D}$$

Example from attention:

$$Q[b, h, n, d]$$



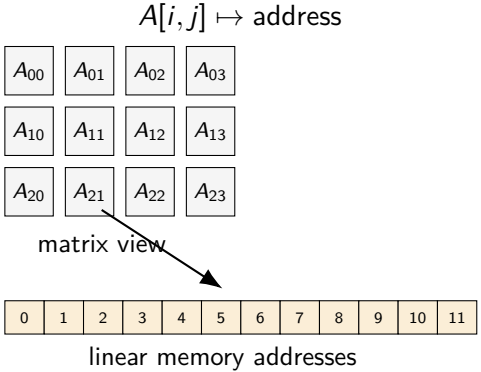
A single element is selected by four indices:

$$Q[b, h, n, d]$$

Memory layout: Linear memory

Memory is physically linear.

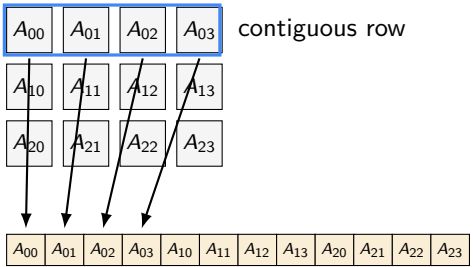
A 2D matrix is mapped onto a 1D address space:



The layout rule determines which element goes to which memory address.

In row-major layout, consecutive elements of a row are contiguous.

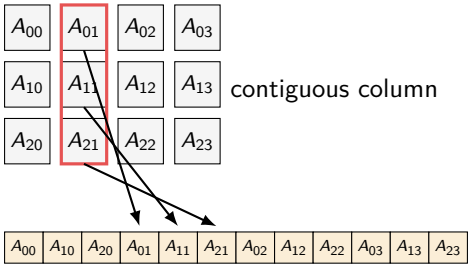
$$\text{offset}(i, j) = iN + j$$



This is the default in C, C++, NumPy, PyTorch, and many GPU kernels.

In column-major layout, consecutive elements of a column are contiguous.

$$\text{offset}(i, j) = jM + i$$



This is common in Fortran, MATLAB, Julia, LAPACK, and BLAS-style APIs.

Memory layout: Strides

A tensor does not only have a shape. It also has strides.

For a tensor

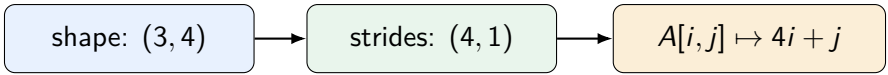
$$X \in \mathbb{R}^{D_0 \times D_1 \times \cdots \times D_{r-1}}$$

with strides

$$s = (s_0, s_1, \ldots, s_{r-1})$$

the memory offset is

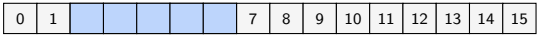
$$\text{offset}(i_0, \ldots, i_{r-1}) = \sum_{k=0}^{r-1} i_k s_k$$



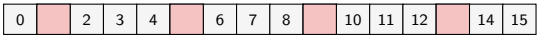
Memory layout: Contiguous vs strided access

A contiguous access touches adjacent memory locations.

A strided access jumps through memory.



contiguous



stride 4

Strided access usually uses cache lines less efficiently.

Transposition as a metadata operation

For a row-major matrix

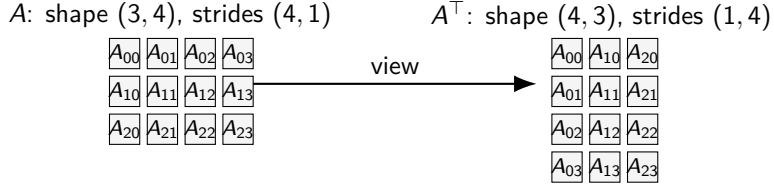
$$A \in \mathbb{R}^{M \times N}$$

the default strides are

$$(N, 1)$$

A transpose view changes shape and strides:

$$A^\top \in \mathbb{R}^{N \times M}, \qquad \text{strides}(A^\top) = (1, N)$$



Transpose view vs materialized transpose

There are two different ideas:

Transpose view

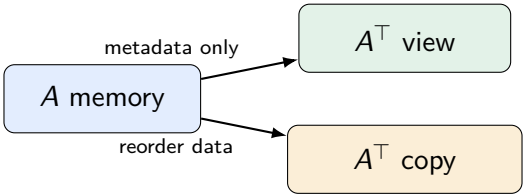
$$B = A^\top$$

- no data copy
- changes shape and strides
- may create non-contiguous access

Materialized transpose

$$B = \text{copy}(A^\top)$$

- data is physically rearranged
- B can be contiguous
- costs extra memory bandwidth



Strides explain many performance surprises

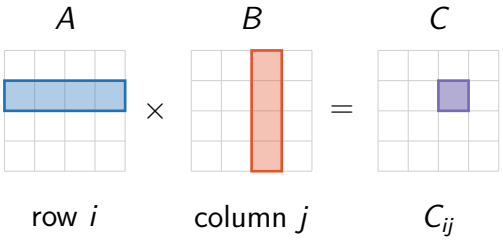
Two tensors can have the same values and shape but different memory behavior.

Operation	Shape effect	Stride effect
reshape	changes dimensions	usually preserves contiguity if possible
slice	smaller view	may introduce larger strides
transpose	swaps axes	swaps strides
permute	reorders axes	reorders strides
contiguous copy	same logical values	physically compacts memory

Shape tells you *what* tensor you have.
Strides tell you *how* it is stored.

For square matrices:

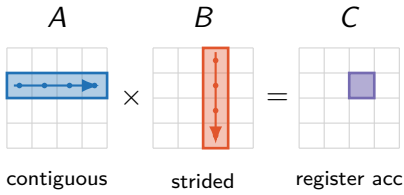
$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}$$



Matmul: Naive implementation

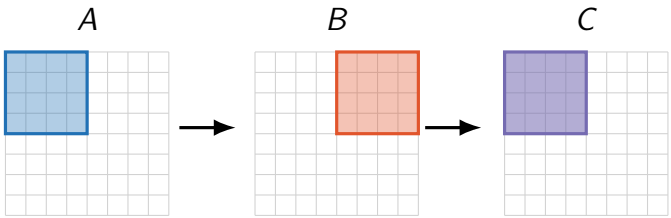
```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    float sum = 0.0f;
    for (int k = 0; k < n; ++k) {
      sum += A[i][k] * B[k][j];
    }
    C[i][j] = sum;
  }
}
```

- Inner loop walks across one row of A.
- Inner loop walks down one column of B.
- One scalar of C stays in a register until the dot product ends.



Instead of completing one scalar at a time, operate on blocks that fit in cache:

$$C_{I,J} \mathrel{+}= A_{I,K} B_{K,J}$$



Matmul: Optimizations

- Tree summation (numerical stability)
- SIMD vectorization
- Fused multiply-add (FMA): `c = c + a * b` is a single instruction
- Multithreading
- Explicit prefetching
- Loop ordering

Matmul: Loop order also matters

Naive i, j, k

- $A[i][k]$: contiguous in k.
- $B[k][j]$: strided by row length.
- $C[i][j]$: scalar accumulation.

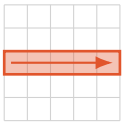
Alternative i, k, j

- $B[k][j]$: contiguous in j.
- $C[i][j]$: contiguous row updates.
- Usually better before adding full tiling.

B: i, j, k

strided column

B: i, k, j



contiguous row

Attention: Basic computation

$$O = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

Naive attention materializes the full score matrix:

$$QK^\top \in \mathbb{R}^{N \times N}$$

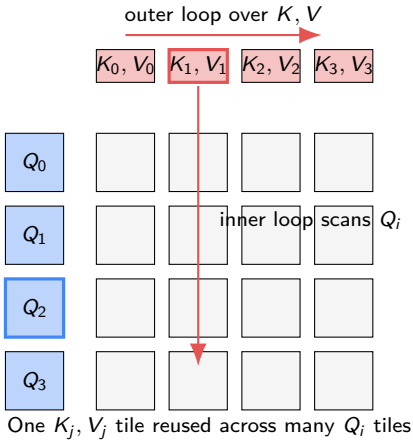
FlashAttention Dao et al. (2022) avoids materializing $QK^\top$ in HBM by computing attention in tiles and maintaining online softmax statistics.

FlashAttention v1: outer loop over K, V blocks

Original FlashAttention loops over blocks of K, V outside, then scans blocks of Q, O inside.

v1 loop order

for $j \in$ KV blocks:
  load K_j, V_j
  for $i \in$ Q blocks:
    load Q_i
    $S_{ij} = Q_i K_j^\top$
    $O_i \leftarrow$ online-update(O_i, S_{ij}, V_j)

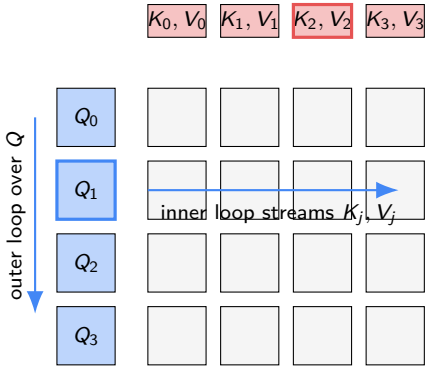


FlashAttention v2: swap the loop order

FlashAttention v2 Dao (2023) swaps the loop order: **2x faster**!

v2 loop order

for $i \in$ Q blocks:
  load Q_i
  for $j \in$ KV blocks:
    load K_j, V_j
    $S_{ij} = Q_i K_j^\top$
    $O_i \leftarrow$ online-update(O_i, S_{ij}, V_j)

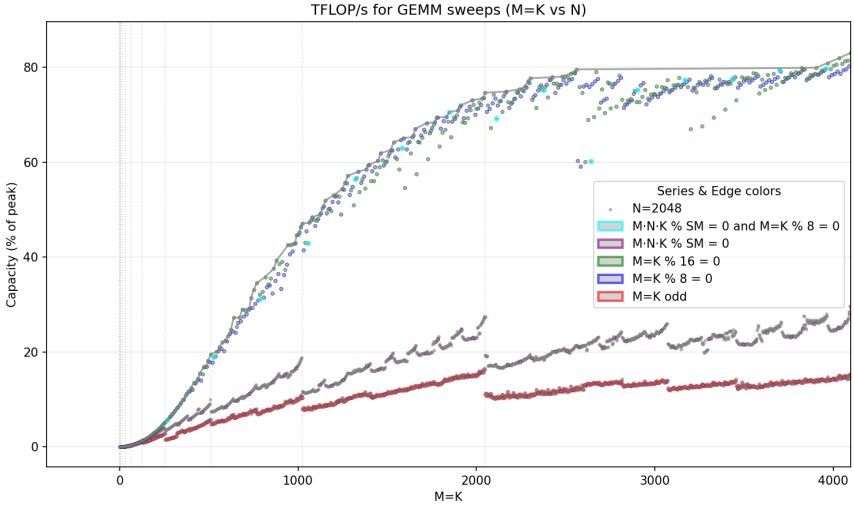


Each Q_i block can be assigned to an independent thread block

Matmul on GPU - H200

Performance differences due to alignment!

- Dimension (K) should be aligned to 8 elements.
- Tiling size: $256 \times 128 = 32768$
- Note: every 4th point satisfies $M \times N \times K \% 32768 = 0$



Source: nVidia perf docs - Recommended!

Further Study

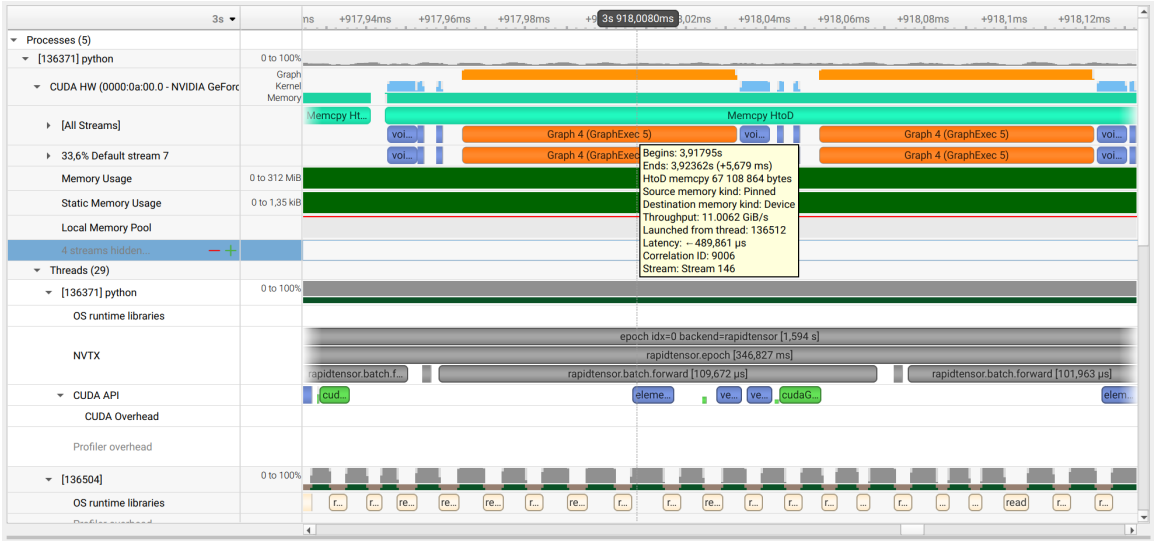
- Philoshophy: Mike Acton "Data-Oriented Design and C++" [YouTube]
- Hardware: What Every Programmer Should Know About Memory [PDF]
- Concurrency: Rust Atomics and Locks [PDF]
- Async runtime: [Blog]
- Software runs slower on newer hardware due to NUMA: [YouTube]

Questions?

Profiling Tools (continued)

Estimated time: 15 min

Nsight Systems Timeline View



Nsight Systems: Practical Profiling Workflow

1. **Focus first**: profile only performance-critical regions (not init/validation code).
2. **Annotate regions**: use NVTX ranges and names for threads/streams/devices.
3. **Choose mode**: one-shot `nsys profile` or interactive `launch/start/stop`.
4. **Control overhead**: enable only traces needed for the current question.
5. **Iterate**: collect short traces, inspect, refine capture scope, repeat.

- Guide emphasis: less data + better labels = faster diagnosis and clearer optimization targets.

Lab 1: Matmul Profiling

/lab/01_matmul/01.a_matmul.py

```
import nvtx
import torch
DIM_M = DIM_K = DIM_N = 1024

@nvtx.annotate("main")
def main():
    with nvtx.annotate("alloc"):
        A = torch.randn(DIM_M, DIM_K, device="cuda", dtype=torch.float16)

    for i in range(1000):
        with nvtx.annotate("init", domain="matmul", payload=i, color="red"):
            B = torch.randn(DIM_K, DIM_N, dtype=torch.float16)

        with nvtx.annotate("move", domain="matmul", payload=i, color="green"):
            B = B.to(device="cuda")

        with nvtx.annotate("step", domain="matmul", payload=i, color="blue"):
            _ = torch.matmul(A, B)
```

Lab 1: Question 1/7

- Q1. What GPU was the profiling done on?

Lab 1: Question 1/7

- Q1. What GPU was the profiling done on?
- A1. RTX 3070

Lab 1: Question 2/7

- Q2. How many physical cores are in the CPU?

Lab 1: Question 2/7

- Q2. How many physical cores are in the CPU?
- A2. 16

Lab 1: Question 3/7

- Q3. Find the throughput of H2D transfer, how long it took, and host memory type.

Lab 1: Question 3/7

- Q3. Find the throughput of H2D transfer, how long it took, and host memory type.
- A3. 15.0278 GiB/s, 259.935 μs, Pageable

- Q4. What sequence of events happens when data is moved?

Lab 1: Question 4/7

- Q4. What sequence of events happens when data is moved?
- A4. `cudaMemcpyAsync`, `cudaStreamSynchronize`

Lab 1: Question 5/7

- Q5. What is the name of the kernel that was doing matrix multiplication?

Lab 1: Question 5/7

- Q5. What is the name of the kernel that was doing matrix multiplication?
- A5. `ampere_fp16_s16816gemm_fp16_256x128_ldg8_f2f_stages_32x3_nn`

Lab 1: Question 6/7

- Q6. How long did `matmul:step` take on avg/med/max/min (ns)?

Lab 1: Question 6/7

- Q6. How long did `matmul:step` take on avg/med/max/min (ns)?
- A6. Run:

```
nsys stats --report nvtx_sum,nvtx_gpu_proj_sum 01.a_matmul.nsys-rep
```

```
** NVTX Range Summary (nvtx_sum):

     Range     Time (%)  Instances      Avg (ns)         Med (ns)         Min (ns)         Max (ns)
 ------------  --------  ---------  ---------------  ---------------  -------------  -------------
 :main             48,0          1  20 217 102 892,0  20 217 102 892,0  20 217 102 892  20 217 102 892
 matmul:init       46,0      1 000      19 444 560,0      19 388 709,0      19 253 811      23 511 313
 GIL Trace          3,0     30 995          44 158,0           1 370,0             150     328 883 066
 :alloc             1,0          1     459 758 972,0     459 758 972,0     459 758 972     459 758 972
 matmul:move        0,0      1 000         202 363,0         200 965,0         197 395         338 788
 matmul:step        0,0      1 000         101 891,0          24 015,0          22 311          77 066 456

** NVTX GPU Projection Summary (nvtx_gpu_proj_sum):

    Range     Total Proj Time (ns)  Proj Avg (ns)    Proj Med (ns)    Proj Min (ns)  Proj Max (ns)
 -----------  --------------------  ---------------  ---------------  -------------  -------------
 :main              19 757 437 061  19 757 437 061,0  19 757 437 061,0  19 757 437 061  19 757 437 061
 matmul:move           119 125 917         119 125,0         118 911,0         118 271         142 047
 :alloc                     10 656          10 656,0          10 656,0          10 656          10 656
 matmul:step             78 876 108          78 876,0          78 848,0          78 720          79 199

# Outputs simplified to fit slide
```

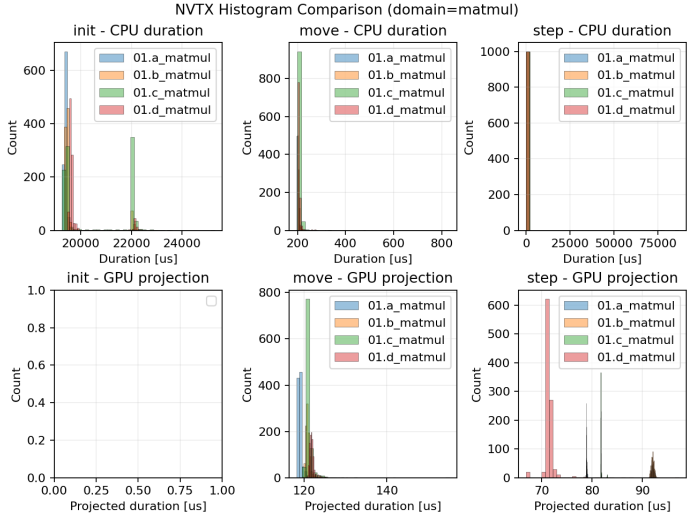
- Q7. Make a histogram of the matmul time distributions for both host and device, across all a/b/c/d variants.

Lab 1: Question 7/7

- Q7. Make a histogram of the matmul time distributions for both host and device, across all a/b/c/d variants.
- A7. Download the analysis guide and use it to derive SQL-based extraction:

```
wget https://docs.nvidia.com/nsight-systems/AnalysisGuide/index.html -O analysis_guide.html
```

Lab 1: Question 7/7



Questions?

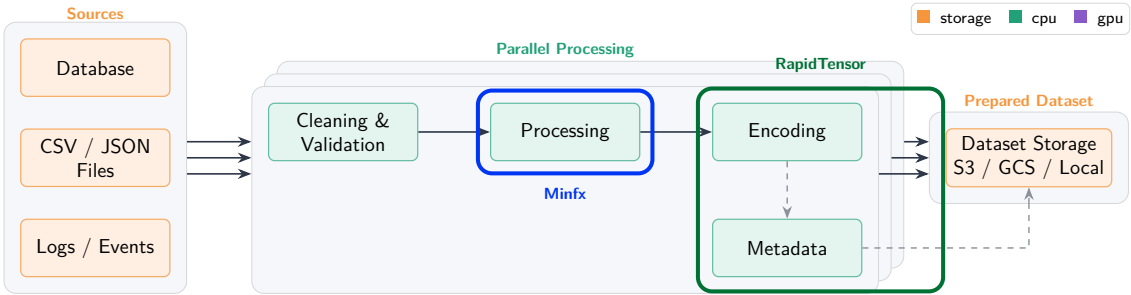
Dataloading and Profiling

Estimated time: 50 min

Overview

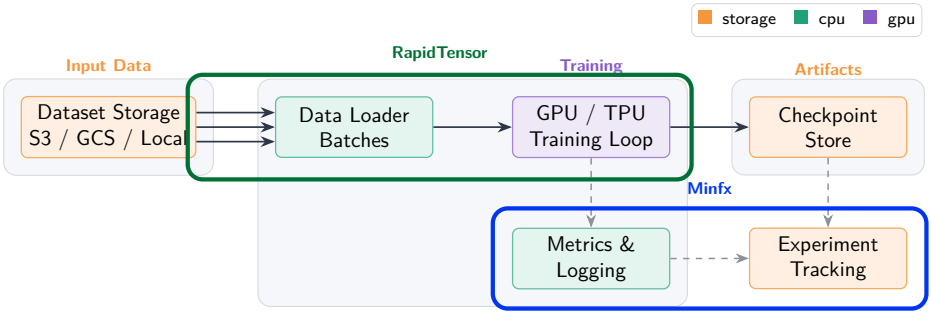
- File formats
- Shuffling
- Moving data: numbers
- Model optimizations
- Lab: going through profiles and code

Typical Pipeline Setup: Data Preparation



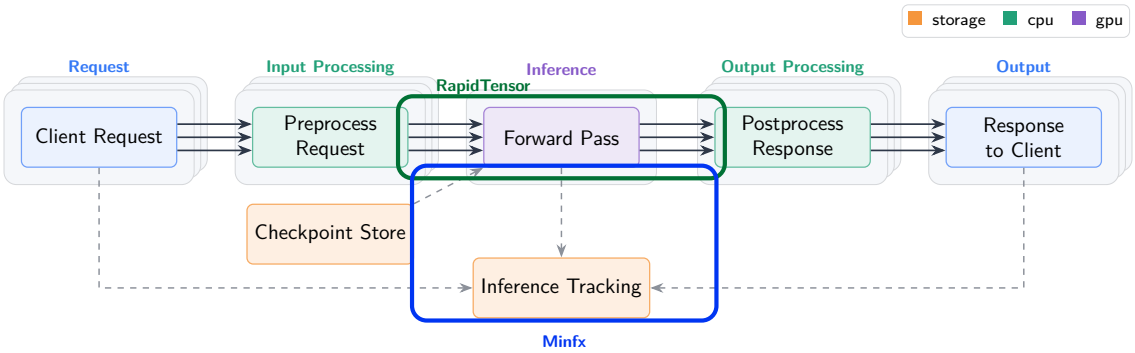
- Save data in a file format for efficient loading
- Track progress and throughput/latency of the pipeline - like distributed `tqdm`

Typical Pipeline Setup: Training



- Load data straight to GPU with high throughput
- Track experiment metrics (loss, accuracy, etc.) - like `wandb`

Typical Pipeline Setup: Inference



- Load data straight to GPU with high throughput, easily plug in with microservices
- Track inference metrics

File formats: Requirements

For efficient dataloading, the storage format is important.

Requirements:

- Standardized versioned format
- Support in many languages
- Small number of files, ideally 1 shard = 1 file
- Streaming read/write
- Random access
- Checksums for integrity
- Compression support
- Multiple data types in a row
- Easy access to metadata
- Ideally chunk reads for efficiency

File formats: Rejected and considered

For efficient dataloading, the storage format is important.

Rejected formats:

- NumPy, Torch (pickle), Safetensors, Lance: no checksums
- TileDB: too many small files, not good for streaming
- HDF5: designed around disk access and metadata can be scattered

Considered formats:

- Parquet
- Zarr
- Arrow IPC

File formats: Parquet, Arrow IPC, Zarr

Property	Parquet	Arrow IPC	Zarr
Streaming read / write	Bad Partial read / Yes-ish write; footer matters!	Yes Designed specifically for streaming	Yes Chunk-wise
Random access	Bad column-order, need to seek a lot!	Yes read footer metadata for batches	Yes read metadata and compute position
Partial/chunk checksums	Yes optional page-level CRC32	Bad checksum only when using compression	Yes e.g. Zarr v3 CRC32C codec
Compression	Yes per page/column	Yes LZ4/ZSTD in implementations	Yes codec pipeline
Multi-type	Yes including nested/list types	Yes schema + field types	Yes many arrays per group/store
Metadata location	Bad Schema in footer, need to jump there	Yes schema/dicts up front	Yes zarr.json / metadata objects beside chunks

Zarr Format

- Open, community-driven format
- Stores chunked, compressed N-dimensional arrays
- Design goal: enable efficient random access to array slices by mapping logical chunk coordinates directly to storage keys

$$\text{array slice} \rightarrow \text{chunk coordinates} \rightarrow \text{store key} \rightarrow \texttt{get(key)}$$

Zarr Format

- Open, community-driven format
- Stores chunked, compressed N-dimensional arrays
- Design goal: enable efficient random access to array slices by mapping logical chunk coordinates directly to storage keys

$$\text{array slice} \rightarrow \text{chunk coordinates} \rightarrow \text{store key} \rightarrow \texttt{get(key)}$$

Ecosystem and maturity. Strongest support is in Python via `zarr-python`, `xarray`, and `dask`, with additional implementations in Rust, JavaScript, C/C++, Java, Julia, and R.

Zarr Format

- Open, community-driven format
- Stores chunked, compressed N-dimensional arrays
- Design goal: enable efficient random access to array slices by mapping logical chunk coordinates directly to storage keys

$$\text{array slice} \rightarrow \text{chunk coordinates} \rightarrow \text{store key} \rightarrow \texttt{get(key)}$$

Ecosystem and maturity. Strongest support is in Python via `zarr-python`, `xarray`, and `dask`, with additional implementations in Rust, JavaScript, C/C++, Java, Julia, and R.

Quick benchmark Ran random-access over sharded data for MNIST dataset with ZSTD + CRC32c and got around 3 GiB/s per core. Also, compression ratio was around 3.7x. (12MiB vs 45MiB original)

Zarr Format

- Open, community-driven format
- Stores chunked, compressed N-dimensional arrays
- Design goal: enable efficient random access to array slices by mapping logical chunk coordinates directly to storage keys

$$\text{array slice} \rightarrow \text{chunk coordinates} \rightarrow \text{store key} \rightarrow \texttt{get(key)}$$

Ecosystem and maturity. Strongest support is in Python via `zarr-python`, `xarray`, and `dask`, with additional implementations in Rust, JavaScript, C/C++, Java, Julia, and R.

Quick benchmark Ran random-access over sharded data for MNIST dataset with ZSTD + CRC32c and got around 3 GiB/s per core. Also, compression ratio was around 3.7x. (12MiB vs 45MiB original)

Comprehensive benchmark is available in the Zarr repository [GitHub].

Shuffling: Permutation

Almost every paper begins with:

Let's take IID samples of the data (without-replacement sampling).

Shuffling: Permutation

Almost every paper begins with:

Let's take IID samples of the data (without-replacement sampling).

```
from random import randint

def fisher_yates_forward_shuffle(seq: list[T]) -> list[T]:
    for i in range(1, len(seq)):
        j = randint(0, i)
        seq[i], seq[j] = seq[j], seq[i]
    return seq
```

Produces an unbiased permutation: every permutation is equally likely.

Shuffling: Permutation

Almost every paper begins with:

Let's take IID samples of the data (without-replacement sampling).

```
from random import randint

def fisher_yates_forward_shuffle(seq: list[T]) -> list[T]:
    for i in range(1, len(seq)):
        j = randint(0, i)
        seq[i], seq[j] = seq[j], seq[i]
    return seq
```

Produces an unbiased permutation: every permutation is equally likely.

```
def random_samples_without_replacement(seq: list[T], k: int) -> list[T]:
    return fisher_yates_forward_shuffle(seq)[:k]
```

Shuffling: Permutation

Almost every paper begins with:

Let's take IID samples of the data (without-replacement sampling).

```python
from random import randint

def fisher_yates_forward_shuffle(seq: list[T]) -> list[T]:
    for i in range(1, len(seq)):
        j = randint(0, i)
        seq[i], seq[j] = seq[j], seq[i]
    return seq
```

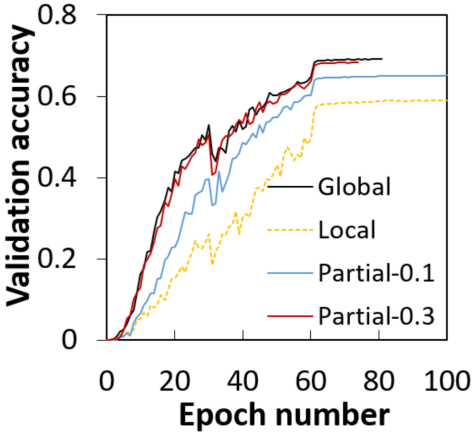
Produces an unbiased permutation: every permutation is equally likely.

```python
def random_samples_without_replacement(seq: list[T], k: int) -> list[T]:
    return fisher_yates_forward_shuffle(seq)[:k]
```

However, both can be hard to realize in distributed settings!

Top-1 accuracy of ResNet50 with ImagetNet-1K (2048 workers)

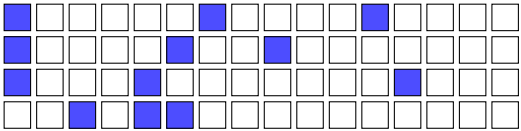
- Global: Permutation
- Local: Sharding to individual workers
- Partial-X: Each epoch fraction of worker's data is redistributed



Source: Nguyen et al. (2022)

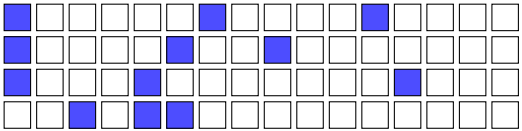
Shuffling: Complications with Sequences

Example: 4 workers, 3 samples/worker

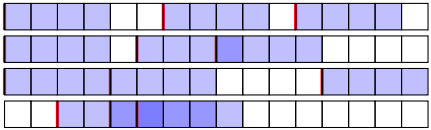


Shuffling: Complications with Sequences

Example: 4 workers, 3 samples/worker

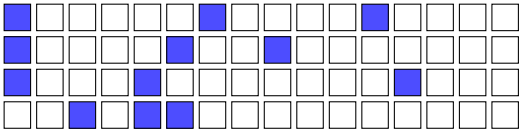


Sequential data, window size 4

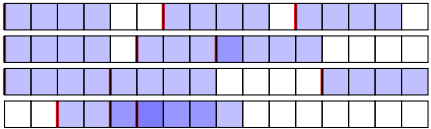


Shuffling: Complications with Sequences

Example: 4 workers, 3 samples/worker



Sequential data, window size 4



Requirements:

- Efficient use of data per worker
- Cover the dataset exactly once per epoch
- Closely approximate random permutations
- No global communication
- Can be (optionally) made deterministic

Shuffling: Detecting Randomness

Goal: test whether a generated ordering "looks like" a random permutation.

n: dataset size, m: observed shuffles, B: simulated null permutations.

Test	What it detects	Cost
Position χ^2	item/position bias	$O(mn)$
Adjacent-pair χ^2	local ordering bias	$O(mn)$
Runs / inversions	too much structure	$O(n)$–$O(n \log n)$
Fixed points / cycles	permutation-level bias	$O(n)$
KS on ranks/gaps	non-uniform spacing	$O(n \log n)$
Monte Carlo p-values	arbitrary statistic	$O(Bn)$

Shuffling: Detecting Randomness

Goal: test whether a generated ordering "looks like" a random permutation.

n: dataset size, m: observed shuffles, B: simulated null permutations.

Test	What it detects	Cost
Position χ^2	item/position bias	$O(mn)$
Adjacent-pair χ^2	local ordering bias	$O(mn)$
Runs / inversions	too much structure	$O(n)$–$O(n \log n)$
Fixed points / cycles	permutation-level bias	$O(n)$
KS on ranks/gaps	non-uniform spacing	$O(n \log n)$
Monte Carlo p-values	arbitrary statistic	$O(Bn)$

Key limitation: testing exact uniformity over all $n!$ permutations is infeasible.
Practical tests are just *projections*: they detect specific failure modes, not perfect randomness.

Shuffling: Sorting Approach

Running these tests is slow. But what is fast: sorting!

Idea: count the number of swaps/inversions required to sort the data and compare the distributions.

Shuffling: Sorting Approach

Running these tests is slow. But what is fast: sorting!

Idea: count the number of swaps/inversions required to sort the data and compare the distributions.

- bubble sort (swap count)
- quick sort (swap count)
- merge sort (inversion count)
- insertion sort (shift count)
- comb sort (swap count)
- bucket sort (internal swap/shift count)
- selection sort (swap count)
- heap sort (swap count)
- shell sort (shift count)
- timsort (shift/inversion count)

Shuffling Test: Permutation vs. Contiguous

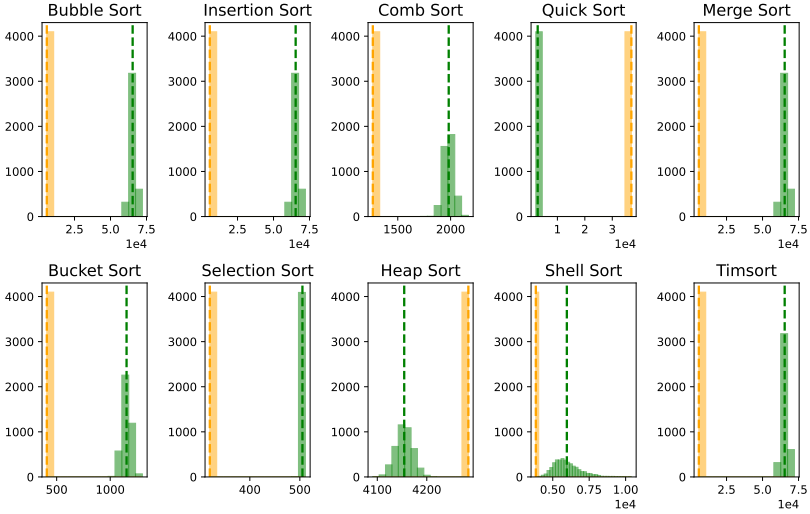
Random Permutation

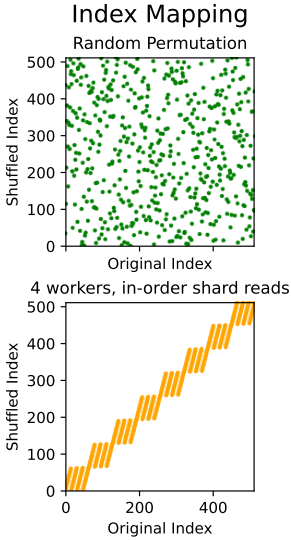
- Build one global permutation of all indices.
- Every batch samples from the whole dataset uniformly.
- No shard-locality bias in index order.

Per-Worker Contiguous

- Split dataset into contiguous shards.
- Each worker reads its shard sequentially (in-order).
- Output interleaves workers, but each worker stream is locally monotonic.

Shuffling Test: Permutation vs. Contiguous





Shuffling Test: Permutation vs. Shard rand

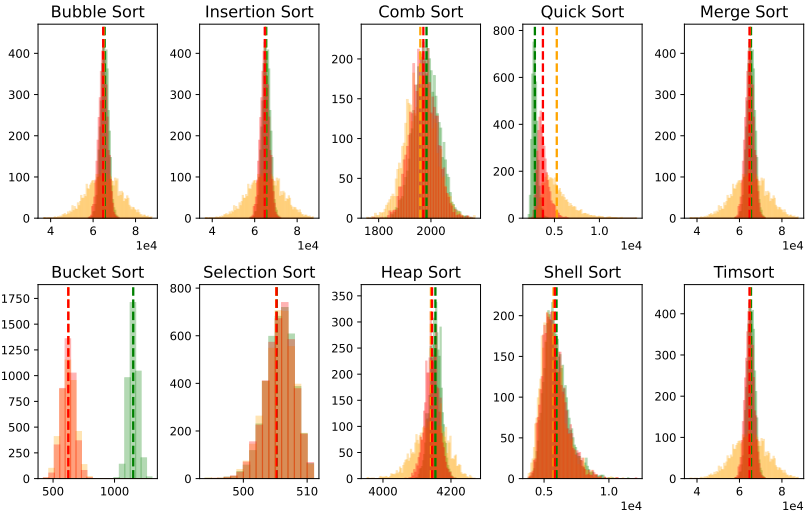
Shard randomization, full read

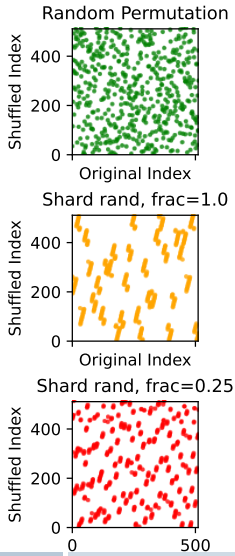
- Variable-size contiguous shards ($\pm 10\%$).
- Workers read assigned shards sequentially.
- Random shard order + per-shard random offset + wrap to beginning.
- Worker take-order is shuffled during emission.
- Very efficient: No need for full in-memory reshuffle.
- Requires ability to seek in a shard (this is usually not a problem).

Shard randomization, partial reads

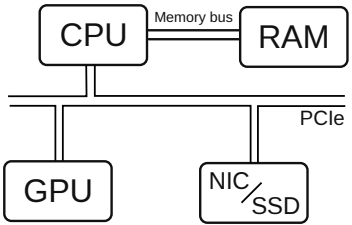
- Same shard/chunked structure.
- Each worker reads only 25% of its current shard before moving to the next shard.
- Workers cycle through shards until all examples are emitted exactly once.
- **Similar to Partial-X scheme.**

Shuffling Test: Permutation vs. Shard rand

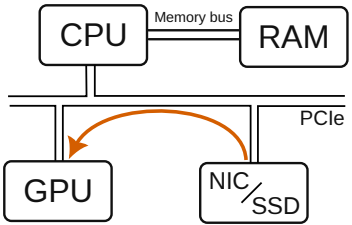




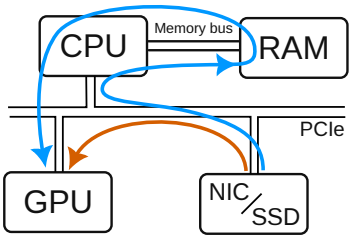
Main Goal: Move Data from NIC/SSD to GPU



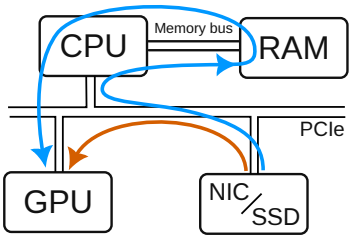
Main Goal: Move Data from NIC/SSD to GPU



Main Goal: Move Data from NIC/SSD to GPU

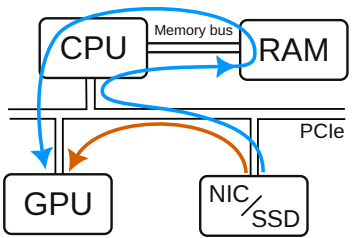


Main Goal: Move Data from NIC/SSD to GPU

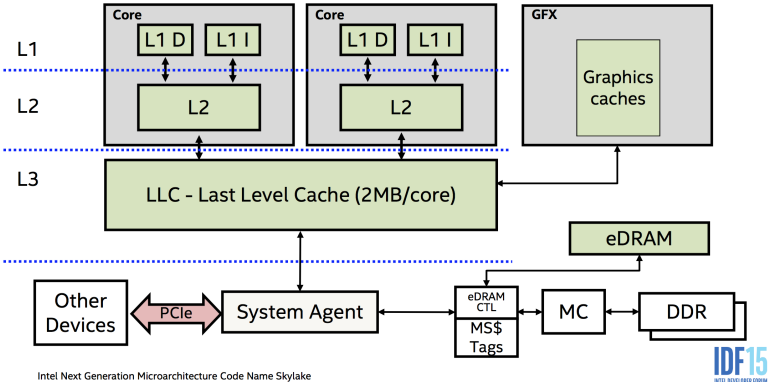


This is not crazy!

Main Goal: Move Data from NIC/SSD to GPU

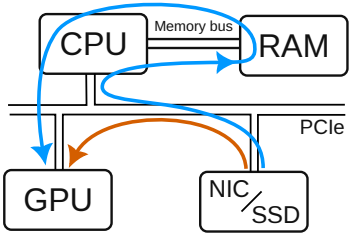


This is not crazy!



Source: Ars Technica [EN]

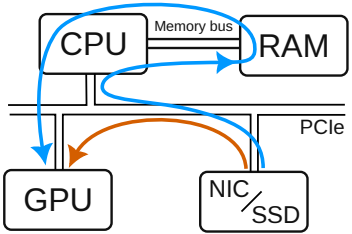
Main Goal: Move Data from NIC/SSD to GPU



This is not crazy!

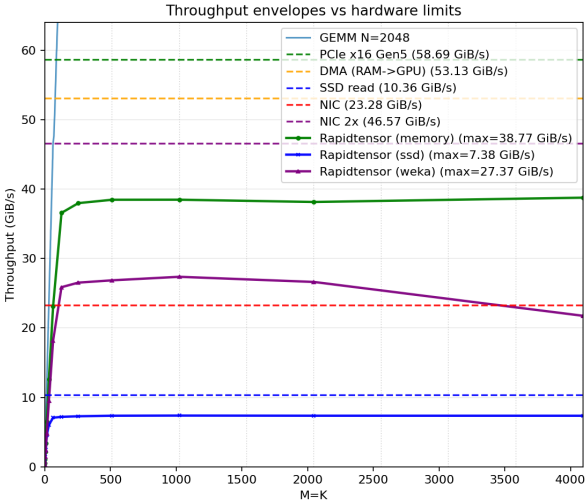
H200 / B200: while 8/16 DMA engines, only 1 (full duplex) for PCIe!

Main Goal: Move Data from NIC/SSD to GPU



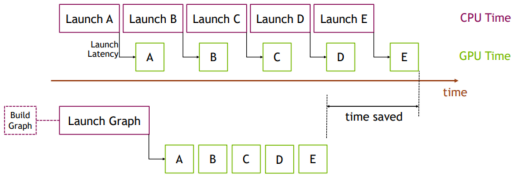
This is not crazy!

H200 / B200: while 8/16 DMA engines, only 1 (full duplex) for PCIe!



Cuda Graphs

- Every kernel launch introduces latency of 1-2us
- We are mostly doing the same work over and over again
- Cuda Graphs are a way to trace the execution of a CUDA kernels and optimize them.
- Tradeoff: fixed buffers (e.g. batch sizes/sequence lengths etc.), fixed graph (no dynamically constructed neural networks)



Source: PyTorch Blog

Cuda Graphs: Compilation and warmup

Torch compilation - in between JIT and full Cuda Graph [docs]:

```
torch.compile(network, mode="reduce-overhead")
```

Two-step passes:

1. warm up each graph, which does things like CuBlas or Triton benchmarking
2. do a CUDA Graph recording, and replays it
3. finally hit the optimized, CUDA Graph replay path

Full CUDA Graph example (more complex): [PyTorch Blog]

Lab 2: Structure

/lab/02_loading/

- 02_loading.py
- dataset.py
- model.py
- utils.py
- loader_torch_naive.py
- loader_no_transfer.py
- loader_torch_opt.py
- loader_rapidtensor.py

Lab 2: Structure

/lab/02_loading/

- 02_loading.py
- dataset.py
- model.py
- utils.py
- loader_torch_naive.py
- loader_no_transfer.py
- loader_torch_opt.py
- loader_rapidtensor.py

/lab/02_loading/results/

Lab 2: `rtx3070.a.nsys.torch_naive`

- How many data transfers per forward pass?
- How long does it take to do the forward pass?
- There is `cudaStreamSynchronize`, even though it's not explicitly called. Explain why, and how it affects the results.

Lab 2: `rtx3070.b.nsys.no_transfer`

- Locate the data transfer in the profiler.
- How come the `cudaStreamSynchronize` disappeared?

Lab 2: `rtx3070.c.nsys.no_transfer`

- How long does it take to do the forward pass?
- What is the difference between this and the previous run?

Lab 2: `rtx3070.d.nsys.no_transfer`

- How long does it take to do the forward pass?
- What is the difference between this and the previous run?

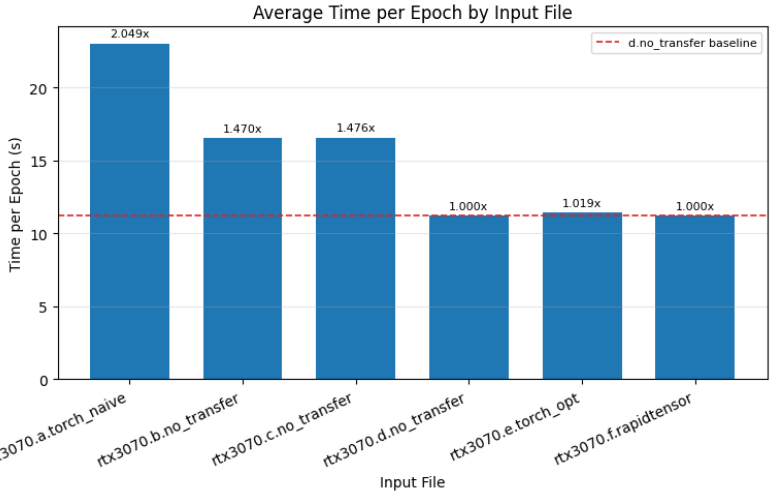
Lab 2: `rtx3070.e.nsys.torch_opt`

- How long does it take to do the forward pass?
- Find where there are any remaining bubbles in the profiler.

Lab 2: `rtx3070.f.nsys.rapidtensor`

- Find where there are any remaining bubbles in the profiler.

Lab 2: Summary Results RTX 3070



Lab 2: h200.d.nsys.no_transfer

- Find where there are any remaining bubbles in the profiler.
- How much could we speed up training/inference if we got rid of them?

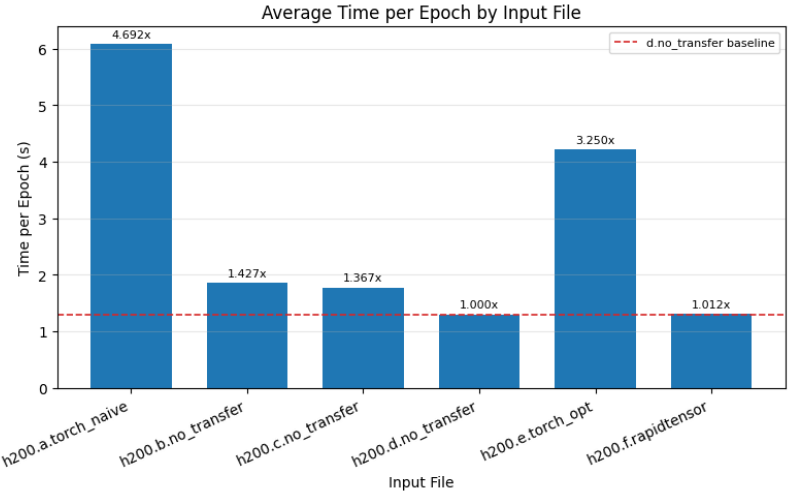
Lab 2: h200.e.nsys.torch_opt

- What are the main issues with this loader?

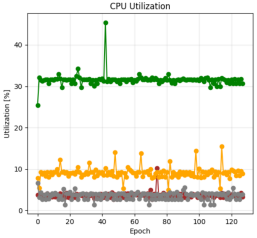
Lab 2: `h200.f.nsys.rapidtensor`

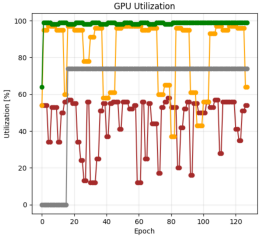
- What are some remaining optimizations?

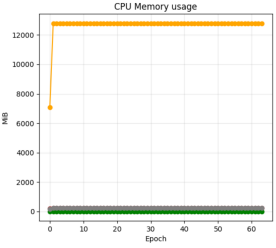
Lab 2: Summary Results H200

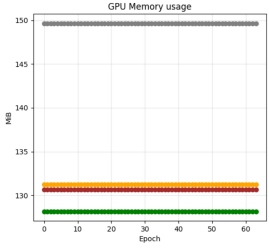


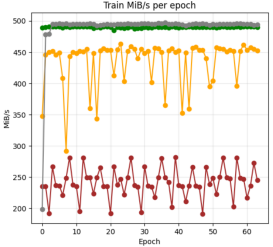
Running for longer (different experiment)

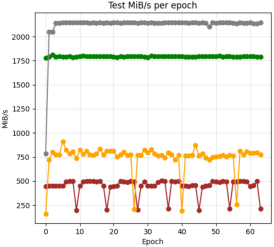












- torch_opt
- torch_naive
- rapidtensor
- no_transfer

Takeaways

RapidTensor

Takeaways

RapidTensor

- Achieves high throughput, similar to one-time transfer to GPU.

Takeaways

RapidTensor

- Achieves high throughput, similar to one-time transfer to GPU.
- Is more stable than the optimized PyTorch dataloaders, due to GC.

Takeaways

RapidTensor

- Achieves high throughput, similar to one-time transfer to GPU.
- Is more stable than the optimized PyTorch dataloaders, due to GC.
- Has lower memory footprint both on CPU and GPU.

Takeaways

RapidTensor

- Achieves high throughput, similar to one-time transfer to GPU.
- Is more stable than the optimized PyTorch dataloaders, due to GC.
- Has lower memory footprint both on CPU and GPU.
- Supports HTTP/HTTPS transport protocols (among others), making it easy to integrate in existing infrastructure.

Takeaways

RapidTensor

- Achieves high throughput, similar to one-time transfer to GPU.
- Is more stable than the optimized PyTorch dataloaders, due to GC.
- Has lower memory footprint both on CPU and GPU.
- Supports HTTP/HTTPS transport protocols (among others), making it easy to integrate in existing infrastructure.
- Has optimizations for dataset shuffling (while other dataloader used just basic shuffling).

Questions?

Thank You for Coming!

Survey: https://forms.gle/UoF1G8aznKUHKzPu8

References I

NVIDIA Performance Documentation [docs]

Find peak arithmetic intensities of each GPU [docs]

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

Truong Thao Nguyen, François Trahay, Jens Domke, Aleksandr Drozd, Emil Vatai, Jianwei Liao, Mohamed Wahib, and Balazs Gerofi. Why globally re-shuffle? revisiting data shuffling in large scale deep learning. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1085–1096. IEEE, 2022.

References II

Weijie Zhao, Xuewu Jiao, Mingqing Hu, Xiaoyun Li, Xiangyu Zhang, and Ping Li. Communication-efficient terabyte-scale model training framework for online advertising. *arXiv preprint arXiv:2201.05500*, 2022.