Lecture 2:
Systems for ML

Edouard Oyallon
edouard.oyallon@cnrs.fr
CNRS, ISIR

What you will learn today

- Latency vs throughput (and why it matters)
- Roofline intuition: when you're compute-bound vs memory-bound
- Parallelism 101: the main ways to scale
- MFU: a simple "are we using the GPU well?" metric

Why LLMs Run on Multiple GPUs

- **Training large language models is extremely compute-intensive**: most of the work is in linear algebra operations (especially large matrix multiplications) in Transformer layers.

- These operations are highly parallelizable, so they can be executed concurrently on many small compute units.

- Accelerators like GPUs were originally designed for graphics and dense linear algebra, making them ideal for training and running LLMs.

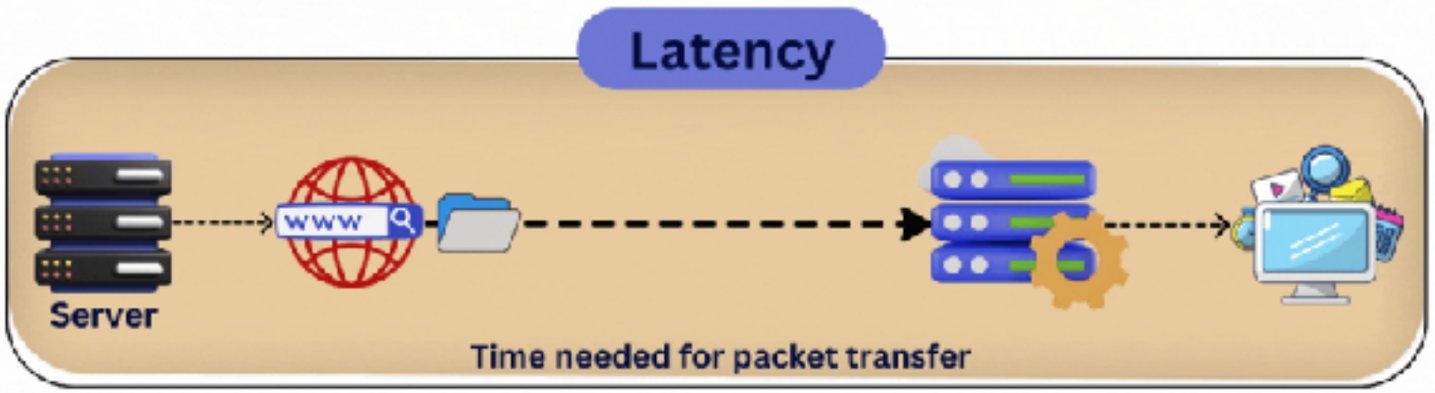
- Modern models are too large for a single GPU, so we distribute them across many GPUs. (other accelerators exist, e.g., TPUs)

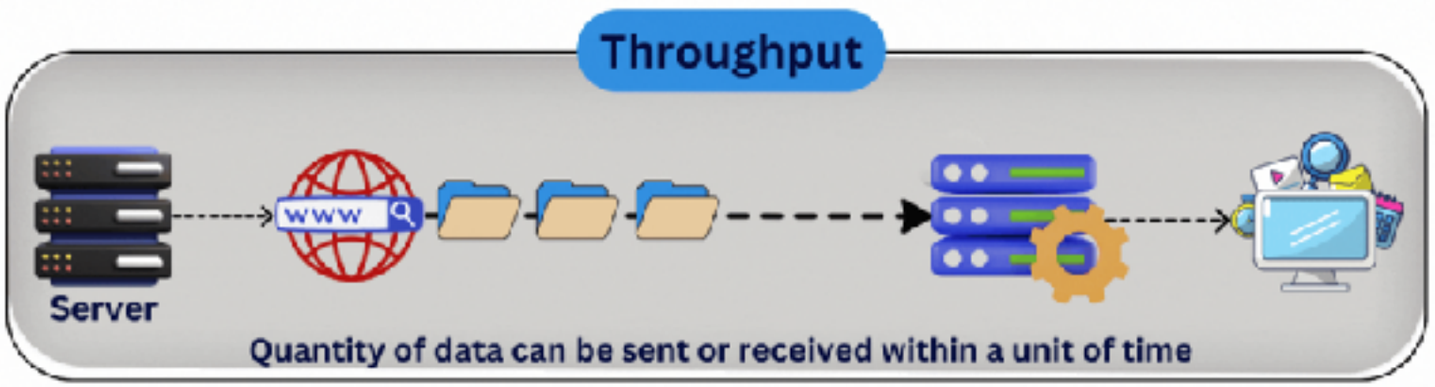
- A key challenge is to make multiple GPUs behave like one big accelerator, coordinating communication and synchronization efficiently.

Understanding GPUs

Throughput vs Latency: System Fundamentals

- **Throughput** is the sustained rate of completed work (items/second)
- **Latency** is the elapsed time for one item to finish (request → response).
- Throughput is bounded by the slowest stage and available parallelism.
- When each task is small, startup dominates: speeding up the task itself barely changes end-to-end time, and throughput won't improve unless you run many tasks in parallel.





Is training related
to latency
or throughput?

Inference?

Type of memories

- Memory is where data resides during computation. Key aspects are **latency** (access time), **bandwidth** (bytes/s) and **capacity.**

- **Caches** are *transparent* copies managed by hardware/OS to exploit temporal/spatial locality. They're in general small (because expensive).

- **Buffers** are *explicit* programmer- or runtime-managed regions that stage data for being re-used.

- For example: on a GPU, the on-chip SRAM cache has about 10× higher bandwidth than HBM main memory, but is roughly 1000× smaller in capacity**:** it's *very* fast but tiny, versus HBM which is slower but huge.

GPU vs CPUs

- **CPUs** aim for **low latency:** a small number of cores with low-bandwidth/low-memory to finish one job quickly.

- **GPUs** aim for **high throughput**: many simpler cores running many similar tasks at once with high-bandwidth/high-latency memory.

- How they hide waiting:
  - CPUs keep one task moving with fast caches and smart reordering;
  - GPUs keep the machine busy by switching to another task whenever one is waiting on data.

- This is why GPUs are excellent for performing linear algebra.

- Once parallelism is high, **memory bandwidth** and **collective sync costs** dominate end-to-end time.

Roofline model

- Suppose we have a black-box routine that does some amount of computation (W, measured in FLOPs) and reads/writes M data in slow memory. Its **Arithmetic Intensity (AI) in FLOPs/bytes is:**

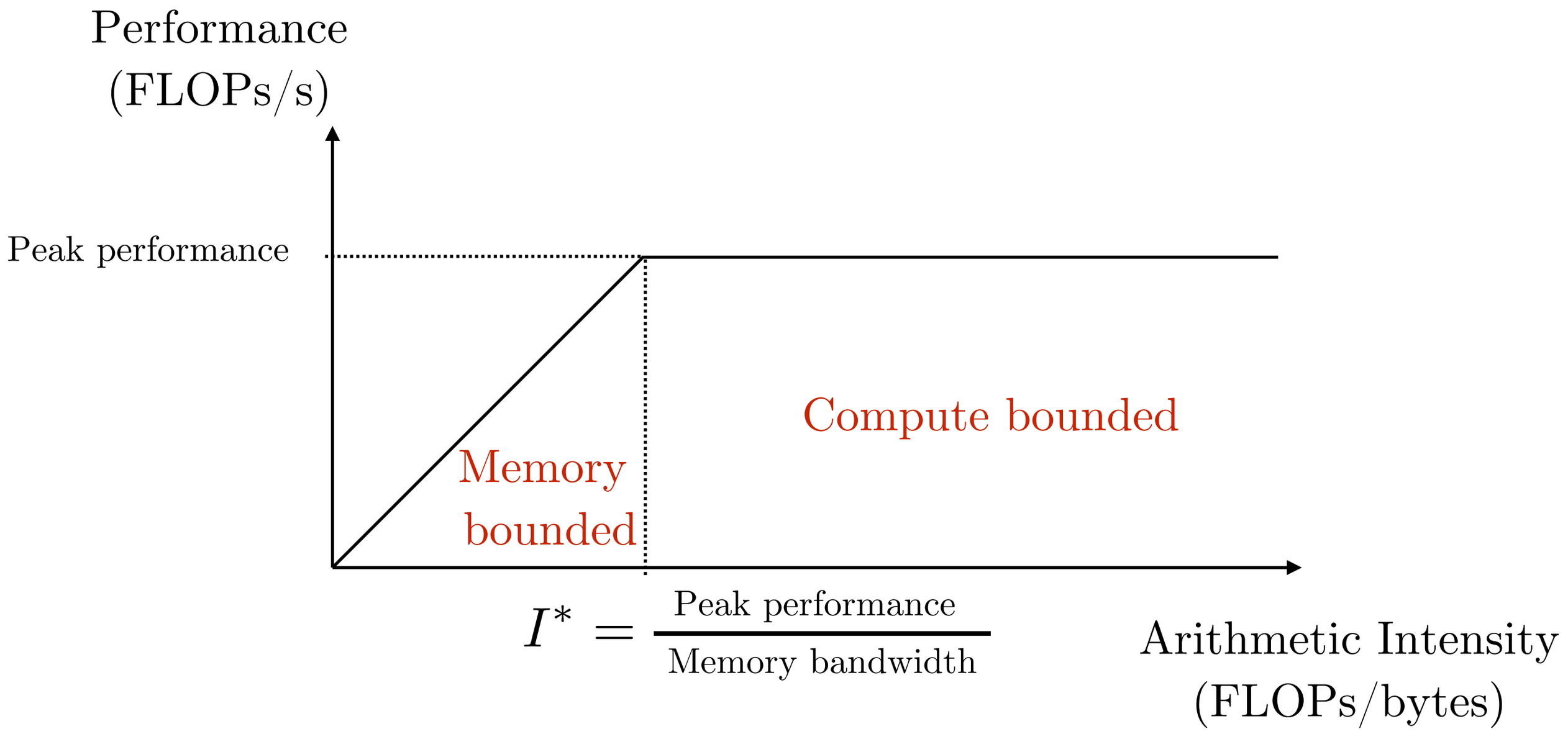
$$I = \frac{W}{M} = \frac{\text{FLOPs}}{\text{memory accesses}}.$$

- **Note:** cache hits are not counted here, only off-chip (HBM/DRAM) memory accesses.(/**!**\)

- Then, the possible performance depends on the hardware as:

$$\underset{\text{FLOPs/s}}{\text{AttainablePerformance}(I)} = \min(\underset{\text{FLOPs/s}}{\text{Peak Performance}}, \; I \cdot \underset{\text{Bytes/s}}{\text{Memory Bandwidth}})$$

Roofline Diagram



Understanding the limits of a GPU (or any other types of hardware) can be simply understood via this diagram.

Matrix multiplication on GPUs

- **Dense matrix multiplications are simple, uniform, and massively parallel**, which makes them easy to implement efficiently in CUDA (the GPU programming model).

- **Matrix multiplications have high arithmetic intensity**: with proper tiling, each value loaded from memory can be reused many times, so we do lots of FLOPs per byte of I/O.

- **Modern GPUs expose Tensor Core primitives** that perform small matrix multiplications (e.g. 16×16) extremely efficiently, especially in low/mixed precision.

- **Libraries (cuBLAS, PyTorch) and compilers rewrite large matrix multiplications** into sequences of these Tensor Core instructions to reach peak GPU throughput.

Examples of GPUs spec

- Memory capacity, memory bandwidth (BW), and peak arithmetic intensity (AI), peak FLOPs, all measured in half precision (FP16):

GPU	**Mem [GB]**	**Mem BW [TB/s]**	**FP16 peak [pFLOPs]**	**AI_FP16 [FLOPs/B]**
V100	32	1.1	0.13	110
A100	80	2.0	0.31	160
H100	80	3.3	2.0	590
H200	141	4.8	2.0	410
B200	180	8.0	2.3	280

Now let's apply the Roofline model to a transformer

Application: Matrix multiplication

- Consider the multiplication of a matrix A with L tokens:

$A \in \mathbb{R}^{d \times n}$
read once A

$x_1 ... x_L \in \mathbb{R}^n$
read every x_l

$\longrightarrow$

$y_1 = Ax_1, ..., y_L = Ax_L$
write every y_l

- Let's compute the arithmetic intensity:

  - FLOPs : $2Lnd$

  - I/O:

$$dn + nL + dL$$

$$AI_{\text{Matrix}} = \frac{2Lnd}{dn + nL + dL}$$

Application: Attention

$$q_1 ... q_L \in \mathbb{R}^d$$
$$k_1 ... k_L \in \mathbb{R}^d$$
$$v_1 ... v_L \in \mathbb{R}^d$$
$$\longrightarrow \quad o_1 ... o_L \in \mathbb{R}^d$$

$$\text{and} \quad \sigma : \mathbb{R}^L \to \mathbb{R}^L, \quad \sigma(x)_i = \frac{\exp(x_i - \max_k x_k)}{\sum_{j=1}^{L} \exp(x_j - \max_k x_k)}$$

- Attention is performed via:

$$qk^T \in \mathbb{R}^{L \times L} \longrightarrow \sigma(qk^T) \longrightarrow o = \sigma(qk^T)v$$

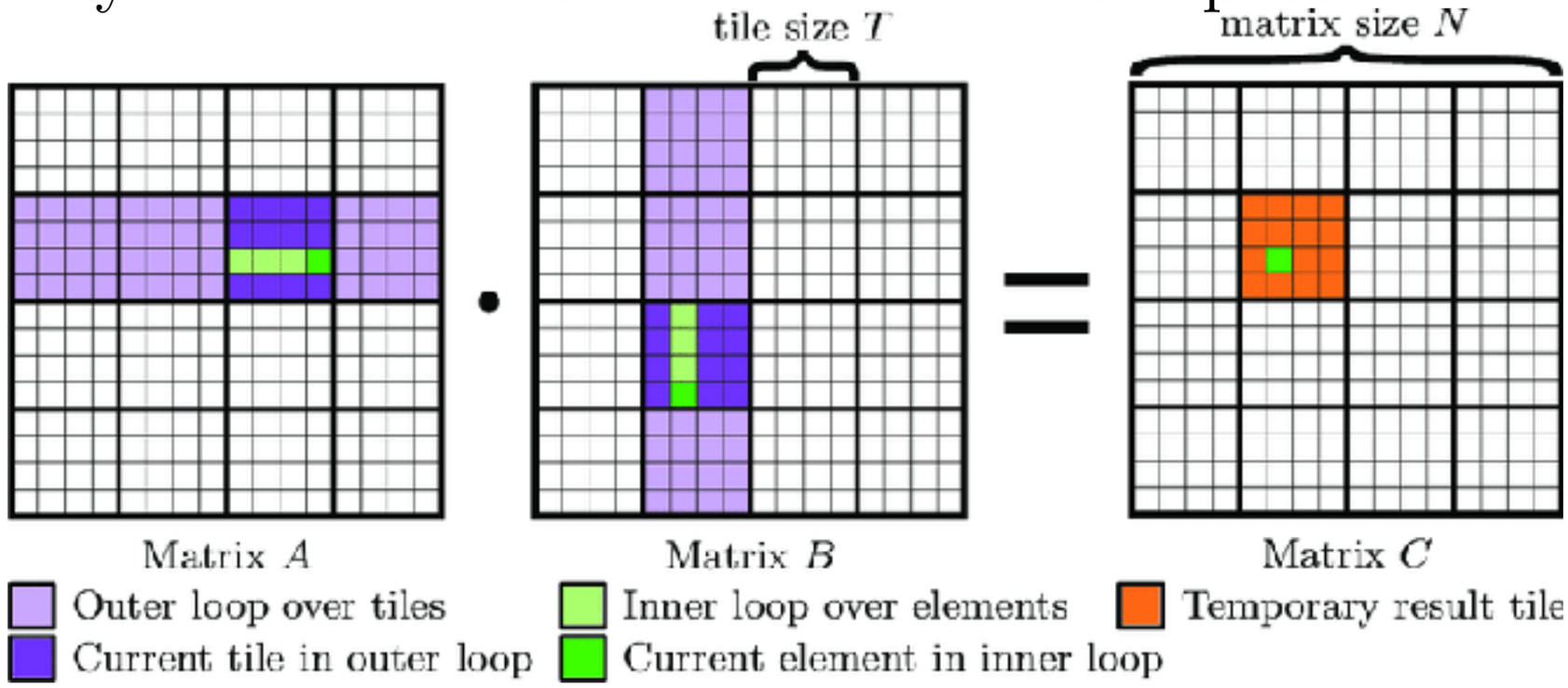
- FLOPs: $L^2(2d + 5 + 2d)$

- I/O: $(2Ld + L^2) + 2L^2 + (Ld + L^2 + Ld)$

$$AI_{\text{Attention}} = \frac{L(4d + 5)}{4(d + L)}$$

The challenge to speed-up attention

- In practice, attention is **memory-bound**: the main cost is I/O.
- Speedups come from reducing I/O..
- **Tiling strategy:** split Q/K/V/O into blocks that fit in on-chip memory and use a recursive relation to compute the output O.



- For each tile: load once, do all computations, write only the final outputs and never materialize the full attention matrix.

Application: Flash attention

$q = [q_1; ...; q_B]$ $k = [k_1; ...; k_C]$ $v = [v_1; ...; v_C]$ $o = [o_1; ...; o_B]$

and $o = \sigma(qk^T)v$

for $c \leq C$ **do**

Load k_c, v_c

for $b \leq B$ **do**

Load $q_b, o_{b,c-1}, l_{b,c-1}, m_{b,c-1}$

$w_{b,c} = q_b k_c^T \longrightarrow m_{b,c} = \max_{c' \leq c}(w_{b,c'})$ and $l_{b,c} = \sum_{c' \leq c}(e^{w_{b,c'} - m_{b,c}})$

Computed using a recursive relation

$o_{b,c} = \sigma(w_{b,1:c})z_{1:c}$

Write $o_{b,c}, m_{b,c}, l_{b,c}$

$o_b = o_{b,C}$

- Assume the existence of a fast cache of size $m = 5\frac{Ld}{C}$ (to load variables)
- The computation of the attention remains stable while being recursive.

I/O: $2Ld + 2CLd + 2CL + CLd + 2CL = 2Ld + 3CLd + 4CL = 2Ld(1 + \frac{3Ld + 4L}{10m})$

FLOPs: $CB(2\frac{L^2d}{BC} + 5\frac{L^2}{BC} + 2\frac{L^2d}{BC} + 10\frac{L^2}{BC} + 10\frac{L^2d}{BC}) = 14L^2d + 15L^2$

what are k_c, v_c, q_b, o_b sizes?

$$AI_{\text{FlashAttention}} = \frac{14Ld + 15L}{2d(1 + L\frac{4+3d}{10m})}$$

Arithmetic Intensity Comparison

- Let us consider a multi-head self-attention layer with $\frac{n}{d}$ heads and token dimension n. This leads to

$$AI_{\text{Matrix}} = \frac{2Ln^2}{n^2 + 2nL} \quad \Big| \quad AI_{\text{Attention}} = \frac{L(4d+5)}{4(d+L)} \quad \Big| \quad AI_{\text{FlashAttention}} = \frac{14Ld + 15L}{2d(1 + L\frac{4+3d}{10m})}$$

and in typical applications, the cache, dimensionality values are

$$n \approx L \approx 10^4 \qquad d \approx 10^2 \qquad m \approx 10^7$$

- This leads to:

$$AI_{\text{Matrix}} \approx \frac{2}{3}L \quad \Big| \quad AI_{\text{Attention}} \approx d \quad \Big| \quad AI_{\text{FlashAttention}} \approx 7L$$

- Avoids materializing the full attention matrix significantly reduces the number of I/Os and makes attention *compute-bounded*.

- In addition, FlashAttention is more memory efficient...

Fused CUDA kernels, foreach

- **Autograd is modular:** models run as graphs of many small primitive operations
- **Backpropagation needs intermediates** values, which must be stored (or recomputed) for gradients
- **GPU downside:** many small kernel launches + high memory traffic → bottlenecks
- This motivates the combination of several operations into a single kernel to be more I/O efficient and reduce launch overhead (*but we lose per-operation visibility*). This leads to three notions:

- **Fused CUDA kernels:** hand-written or library-provided kernels that implement multiple operations together (e.g. bias addition + activation, FlashAttention), reducing memory reads/writes.
- **Compiler-based fusion:** graph compilers that analyze the computation graph and automatically generate fused kernels, generalizing manual fusion
- **Foreach CUDA kernels:** specialized kernels that apply the same operation to a list of tensors in one go (e.g. optimizer steps), amortizing kernel launch overhead.

Strategy to reduce memory footprint: Quantization, activation checkpointing

Reducing Memory Footprint

- In large-scale training, GPU memory is one of the main bottleneck: it limits the model size, batch size, and context length we can use, and can force us onto more (or more expensive) GPUs.

- For reducing the memory footprint, several techniques are commonly used:

  - **Fused kernels** (as we just saw): e.g., FlashAttention

  - **Quantization**: Trading numerical precisions for lower memory footprint (and faster computations)

  - **Activation recomputations**: Recomputing activations during backpropagation, trading extra compute for a significant reduction in activation memory.

  - **Sharding/chunking/splitting** resources across multiple GPUs (next lecture)

Mixed-Precision Training: Benefits and Pitfalls

- **FP32 (32-bit, 4 bytes)** is the standard format for training and is usually numerically accurate enough for deep networks.

- To **reduce FLOPs and memory footprint**, it is attractive to use **16-bit formats (FP16)** for most computations and activations.

- In mixed-precision training:

  - The **forward and backward passes** store activations and gradients in **FP16**.

  - Many operations (e.g., matmuls, reductions) still **accumulate in FP32** to keep numerical accuracy.

  - A set of master weights and gradients is kept in FP32 to reliably update parameters.

- Another **Challenge:** FP16 has a **much smaller dynamic range**, so some weights and gradients can **underflow to zero**, hurting training stability.

Automating Mixed PrecisionExample

- AMP (Automatic Mixed Precision) chooses the datatype per operation to balance speed and numerical stability during forward *and* backward.

- With **FP32**, typical training ranges rarely cause numerical issues.

- Numerically sensitive ops (stay in FP32): **sigmoid**, **softmax**, **exp**/**log**, and some reductions are computed in **FP32** to avoid underflow/overflow and preserve accuracy

- Throughput-oriented ops (use FP16 + FP32 accumulate):
  - Accumulation in FP32 (partial sums), eg $\sum_{i=1}^{n} x_i y_i$
  - Output often **cast back to FP16** if the next layer expects FP16.

Gradient underflowing

- Consider this simple function to optimize:

$$f(\theta) = \frac{1}{2} a\theta^2 \quad \text{with} \quad f'(\theta) = a\theta$$

the a gradient descent should write:

$$\theta_{t+1} = (1 - \eta a)\theta_t = (1 - \eta a)^t \theta_1$$

Unit roundoff:

$$1 + \epsilon = 1$$

- If a is very small, i.e., $|\eta a| < \varepsilon_{\text{min}}^{\text{float16}}$, then, there are three solutions:
  - Forcing the operations in *float32*.
  - Rescaling the loss $f_s(\theta) = sf(\theta)$ with $s > 0$ so that $|\eta s a| > \varepsilon_{\text{min}}^{\text{float16}}$
  - Or, changing the range: that's why *brain-float16* are made for.

Float types and precision

- If stable, lower precision brings speed, reduced latency and memory saving, even if master weights are needed.

- BF16 keeps (almost) FP32's dynamic range while having the benefits of FP16, thus removing the need for loss scaling.

Type	**Bytes**	**Exp/Mant**	**Range**	**Rel. precision**	**Stable**
FP32	4	8 / 23	$10^{\pm 38}$	$\approx 2^{-23} \approx 1.19 \times 10^{-7}$	High
BF16	2	8 / 7	$10^{\pm 38}$	$\approx 2^{-7} \approx 7.81 \times 10^{-3}$	Medium-High
FP16	2	5 / 10	$10^{\pm 5}$	$\approx 2^{-10} \approx 9.77 \times 10^{-4}$	Medium
FP8	1	4 / 3	$\sim 10^{\pm 9}$	$\approx 2^{-3} \approx 1.25 \times 10^{-1}$	Low

Gradient accumulation in full precision

- Each parameter update is the sum of **many small gradient contributions**. Accumulating these directly in **BF16/FP16 can cause numerical *instability*** (rounding/underflow), especially with large effective batch sizes or many accumulation steps.

- Update flow:
  - Forward / backward in **BF16/FP16.**
  - Cast grads to FP32 (and unscale if using loss scaling)
  - Update FP32 master weights and FP32 optimizer states (/!\)
  - Cast updated master weights back to **BF16/FP16** for the next step

- Using FP8 is still an on-going research topic, as well as getting rid of the master weights.

Activation Checkpointing

- **Idea:** Store activations only at a few **checkpoints**, and **recompute** the missing ones during backward → big **memory savings** for extra compute.

- In forward, save activations only at selected points (e.g. **FFN blocks**); in backward, **re-run forward between checkpoints** to recover them.

- In Transformers, recompute mainly self-attention, which is memory-bound, while FC layers are more FLOP-heavy but less memory-hungry. This is the idea exploited by FlashAttention-style kernels.

- Benefit: Enables deeper / larger models or bigger batches on the same GPU, especially when combined with mixed precision and activation offloading.

Reducing Memory Overhead via Activation checkpointing

Forward connexion

Input layer

Intermediary layer

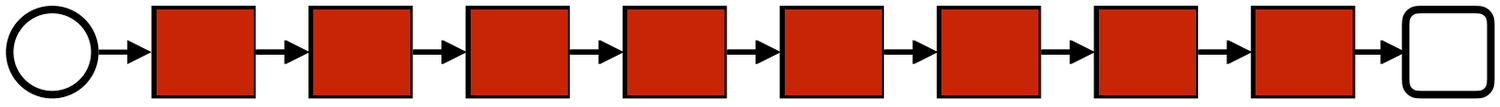
Final layer

Checkpoints

Activations materialized during forward

Activations re-computed during backward

Backprop without Activation Checkpointing:



The total computational cost of the backprop pass is: $F+B+W$

Multiple GPUs

How to make GPUs collaborate?

- A single GPU is already massively parallel, but we can scale even further by distributing computations across multiple GPUs.

- Each GPU is an isolated accelerator; combined, they can deliver more FLOPs, host larger and deeper models, and process more samples in parallel.

- The key challenge is to minimize and overlap inter-GPU communication —data transfers and synchronization—so bandwidth and latency don't eat into compute time; this is very similar to I/O bottlenecks on a single GPU.

- Before diving into concrete strategies, we'll first look at the possible interaction patterns between GPUs and the infrastructure needed to support them.

Programmation for Multi-GPU applications

Single Program, Multiple Data

- *Single Program, Multiple Data*: The same program runs concurrently on multiple devices (organised as a *mesh*); each device operates on its own slice of data and may take device-specific branches.

- **Same code, different views:** Program text is identical; behavior can diverge by device (e.g., different data/tensor partitions).

- **Local compute, global coordination:** Do work locally; synchronize via **collectives** (e.g., all-reduce, reduce-scatter, all-gather, broadcast).

- **Rank & world size:** The total number of devices is the **world size**. Devices are distinguished by an integer `rank` (e.g., `if rank == 0: #do something on device #0`).

"hello world"

```
# hello.py
import socket
import torch.distributed as dist


def main():
    dist.init_process_group(backend="gloo")  # CPU-friendly; use "nccl" on
GPUs
    rank = dist.get_rank()
    world = dist.get_world_size()
    host = socket.gethostname()
    print(f"Hello from rank {rank}/{world} on host {host}", flush=True)
    dist.barrier()  # simple synchronization point
    dist.destroy_process_group()


if __name__ == "__main__":
    main()
```

Run in bash:

```
torchrun --standalone --nnodes=1 --nproc_per_node=4 hello.py
```

Distributed Instructions

- `dist.barrier()`: all ranks wait here until everyone arrives. Don't put it inside `if rank == 0:` or any rank-conditional branch, or you'll deadlock.

- Point-to-point communications (`dist.send()`, `dist.receive()`): Let two specific ranks talk directly without involving everyone; useful for custom protocols or pipeline-style setups.

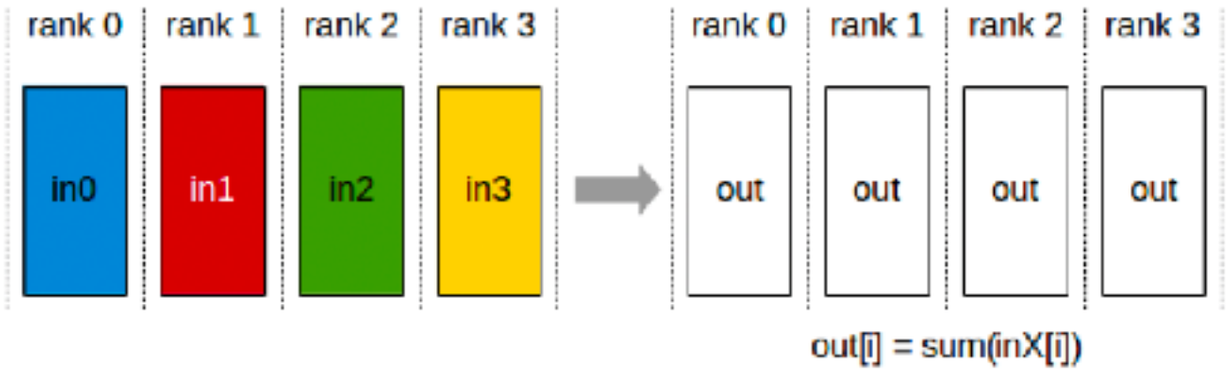
- Process groups & subgroups (`dist.new_group()`): You can form smaller groups of ranks and then use specific instruction, e.g `dist.barrier(…, group=…)` which is a selective synchronization

- And collective primitives, that we'll discuss now: One-to-One, One-to-Many, Many-to-One, Many-to-Many.

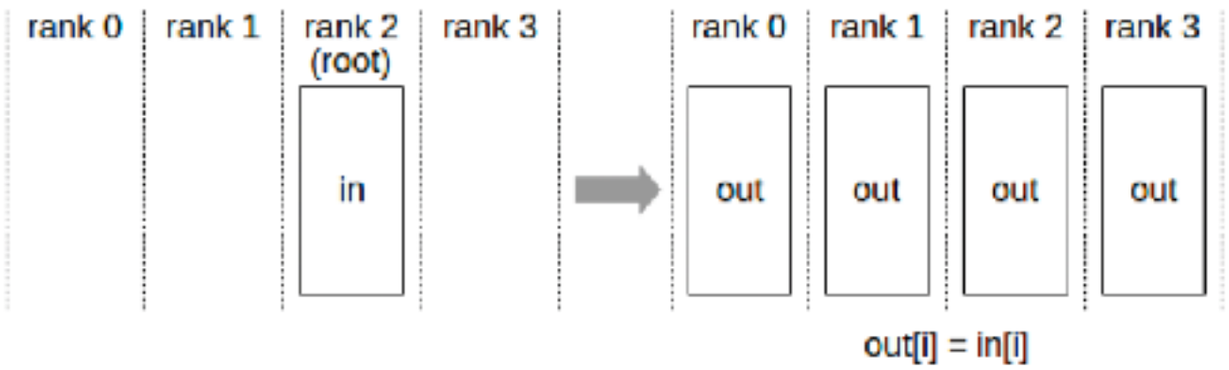
Example of collectives

The goal of collectives is to efficiently orchestrate common communication patterns across all workers with a single optimized operation.

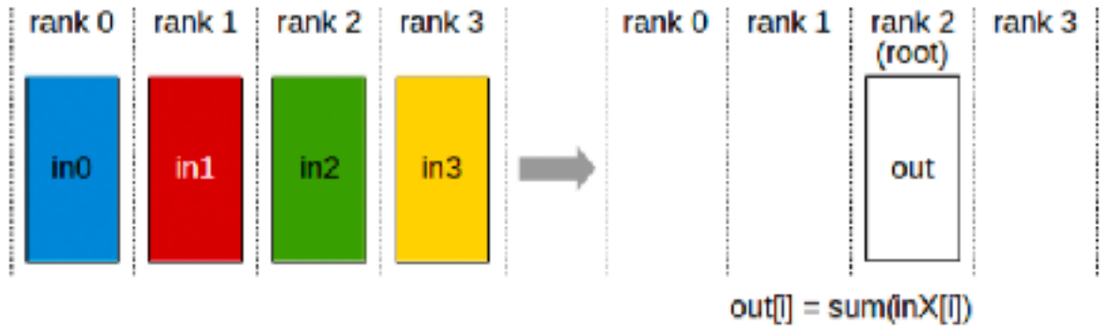
All reduce



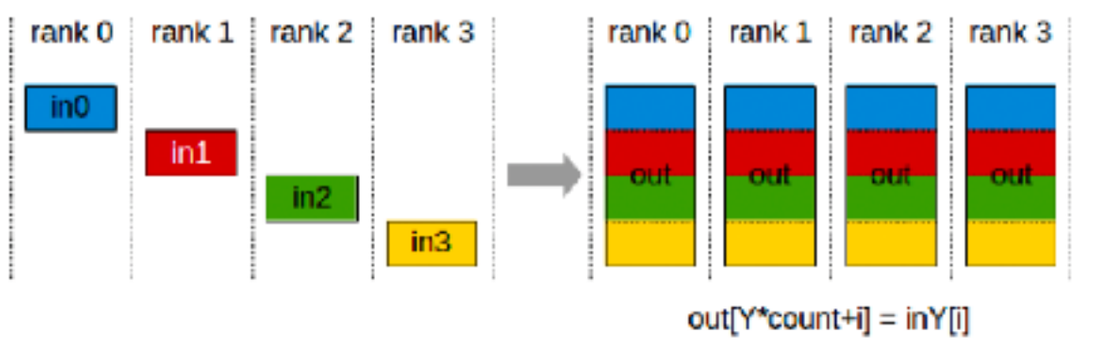
Broadcast



Reduce



All Gather

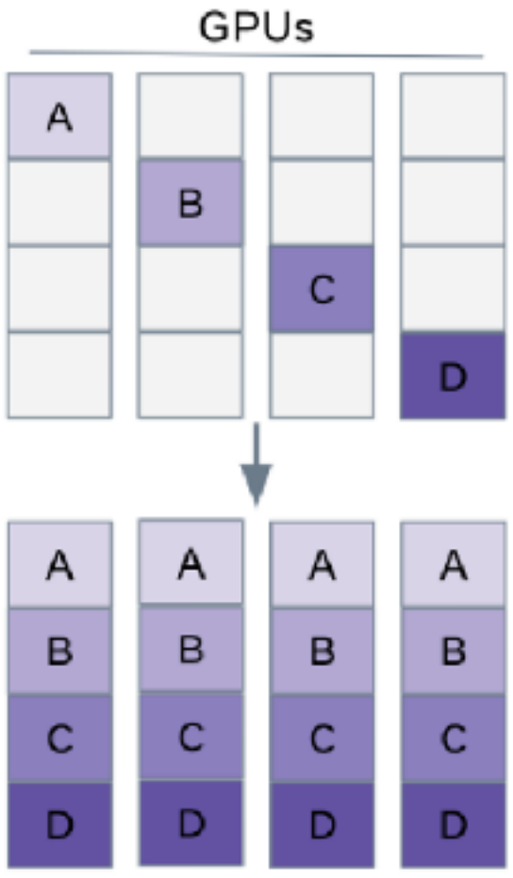


Reduce Scatter

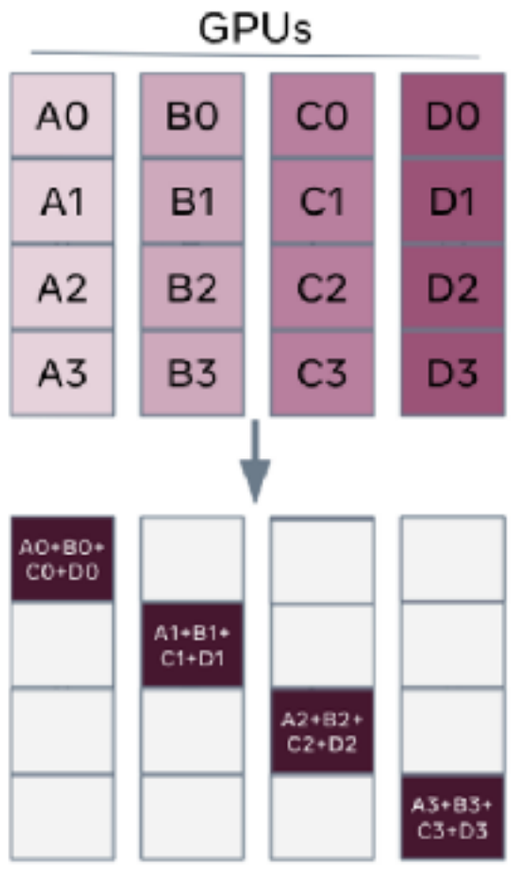
Compatibility of Distributed Collectives with Autograd

- Collective routines are compatible with Autograd!

Forward



Backward



cc: PyTorch

AllGather: $\{x_0, \ldots, x_{p-1}\} \longrightarrow \left\{ [x_0, \ldots, x_{p-1}] \right\}_{r=0}^{p-1}$

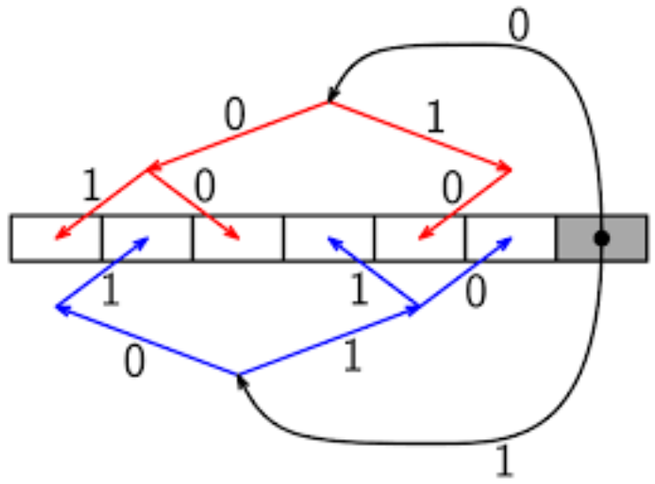
(the same vector is present on every rank).

Backward: $\left\{ [g_0^r, \ldots, g_{p-1}^r] \right\}_{r=0}^{p-1} \longrightarrow \left\{ \sum_{r=0}^{p-1} g_i^r \right\}_{i=0}^{p-1}$

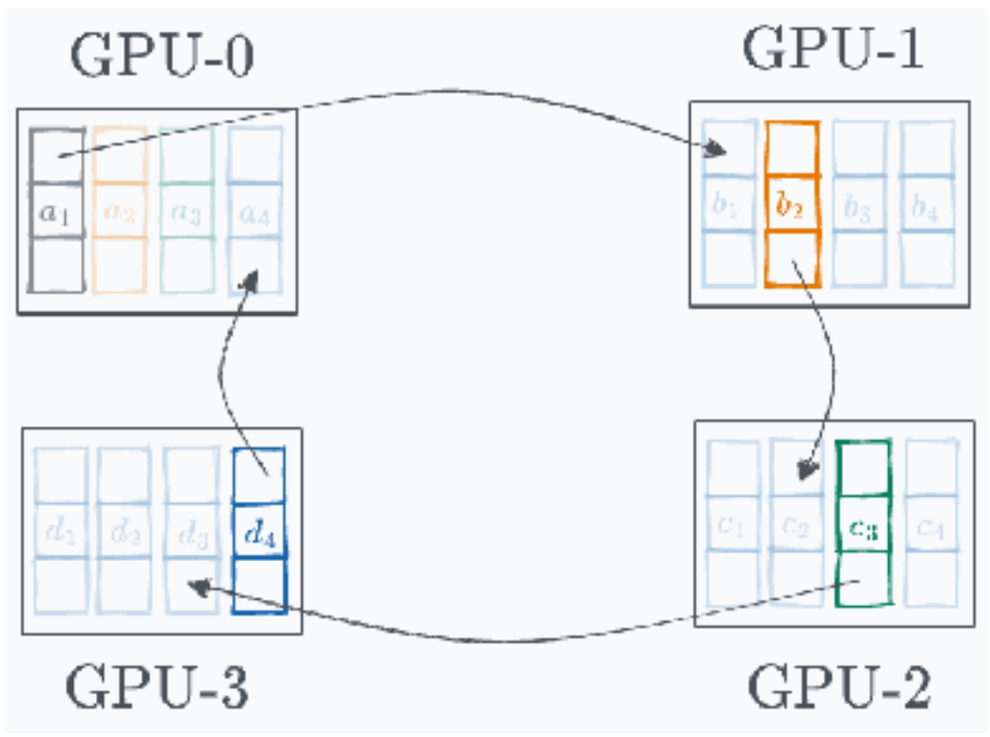
(rank i receives the sum of the i-th entries).

An implementation of All-Reduce

- Tree all-reduce wins for small tensors / many ranks (latency-bound).

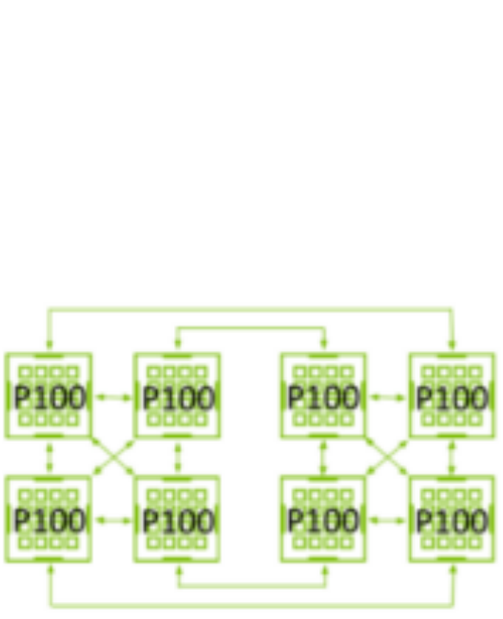


- **Ring all-reduce** wins for **large tensors** (bandwidth-bound).



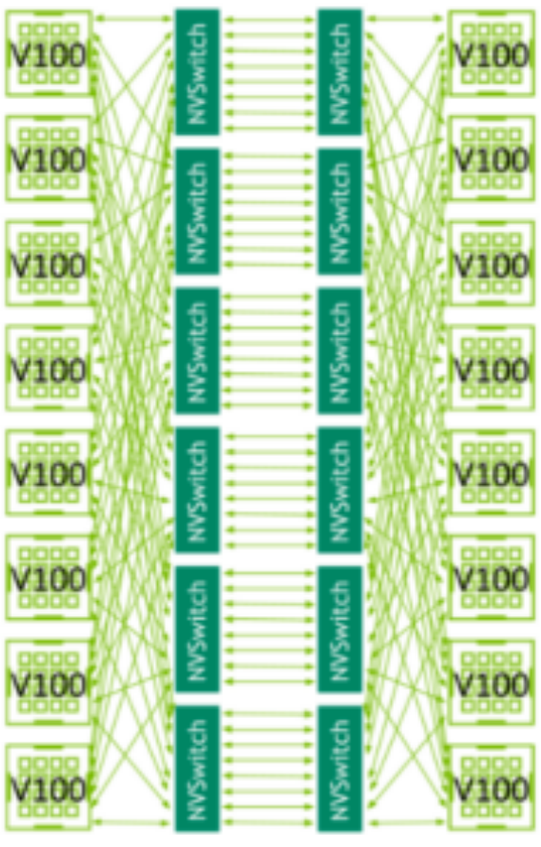
Distributed Setting on a cluster

Big picture of a cluster



2016

DGX-1 (P100)

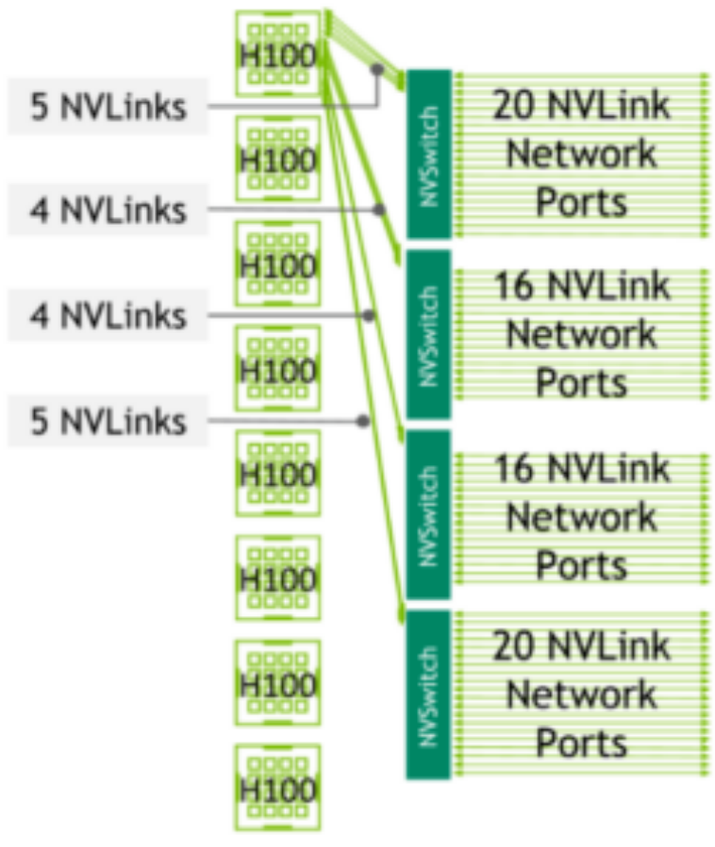


2017

DGX-2 (V100)

2020

DGX A100



2022

DGX H100

What is a (GPU) cluster?

- A **cluster** is many compute nodes (machines) connected by a **high-speed network**.
- Each **node** has multiple **GPUs**, **CPUs**, **RAM**, and **local storage**.
- Nodes usually share a global filesystem (e.g. /home, /scratch) accessible from all nodes.
- A **scheduler** (e.g. Slurm) groups nodes into **partitions** (set of node with similar hardware which obey some policy).
- It's often OK to assume a partition is **homogeneous**.
- Example**:** partition `gpu-a100`: **80 nodes**, each with 4× A100 GPUs.

Examples of cluster: JeanZay

- Jean Zay (public French AI/HPC supercomputer)

Partition	Type of node	# GPUs	GPU	CPUs	RAM	Total GPUs
cpu_p1	CPU-only	0	–	40 cores	192 GB	0
gpu_p13	4-GPU	4	V100 16/32GB	40 cores	192 GB	1584
gpu_p2	8-GPU	8	V100 32GB	24 cores	384 GB	248
gpu_p4	8-GPU	8	A100 40/80GB	48+ cores	512 GB	416
gpu_p5	4-GPU	4	H100 80GB	48+ cores	512 GB	1456

In your opinions, which partitions are free?

- For comparison a single "node" Google TPU v4 Pod: Up to **4,096 TPU chips** in a single pod (one giant accelerator island)

Slurm

- **Slurm** is the standard software used to **assign cluster resources** (CPUs, GPUs, memory, time) to users' jobs.

- You do not run jobs directly on nodes: instead, you **submit a job** to Slurm, and Slurm puts it in a **queue** and starts it when resources are free.

- Typically jobs have to finish in a finite time: it's thus good to make job batch-able via checkpoints so they can be re-started periodically

- A typical slurm command looks like this:

number of processes per machine

```
srun --partition=gpu-a100 --nodes=4 --ntasks-per-node=4
--gres=gpu:4 --cpus-per-task=8 --time=02:00:00

torchrun --nnodes=4 --nproc_per_node=4 --rdzv_backend=c10d
--rdzv_endpoint=$MASTER_NODE:29500 train.py --checkpoint ./save
```

run 16 times

Yaml config files in torch titan

```
[job]
dump_folder = "./outputs"
description = "DeepSeek-V3 16B model training"
print_config = false

[profiling]
enable_profiling = false
save_traces_folder = "profile_trace"
profile_freq = 10
enable_memory_snapshot = false
save_memory_snapshot_folder = "memory_snapshot"

[metrics]
log_freq = 10
disable_color_printing = false
enable_tensorboard = false
save_tb_folder = "tb"
enable_wandb = false

[model]
name = "deepseek_v3"
flavor = "16B"
hf_assets_path = "./assets/hf/deepseek-moe-16b-base"
# converters = ["float8"]

[optimizer]
name = "AdamW"
lr = 2.2e-4
eps = 1e-8

[lr_scheduler]
warmup_steps = 200
decay_ratio = 0.8
decay_type = "cosine"
min_lr_factor = 0.1

[training]
local_batch_size = 4
seq_len = 4096
max_norm = 1.0  # grad norm clipping
steps = 1000
dataset = "c4"  # supported datasets: c4_test (2K), c4 (177M)

[parallelism]
data_parallel_replicate_degree = 1
data_parallel_shard_degree = -1
fsdp_reshard_after_forward = "default"
tensor_parallel_degree = 1
enable_async_tensor_parallel = false
pipeline_parallel_degree = 1
pipeline_parallel_schedule = "Interleaved1F1B"
expert_parallel_degree = 8
expert_tensor_parallel_degree = 1

[checkpoint]
enable = false
folder = "checkpoint"
interval = 10
last_save_model_only = true
export_dtype = "float32"
async_mode = "disabled"

[activation_checkpoint]
mode = "selective"

[compile]
enable=true
components = ["loss"] # ["model", "loss"]

[quantize.linear.float8]
enable_fsdp_float8_all_gather = false
precompute_float8_dynamic_scale_for_fsdp = false
filter_fqns = ["output", "router.gate"]

[quantize.grouped_mm.float8]
fqns = ["experts"]
```

Communications between GPUs

Hockney Model for Communications:

- Channels are independent paths that can carry data **in parallel.**

- Typically, the time to send a message of size m si given by

$$T_{\text{comm}}(m) = \alpha + \beta \cdot m$$

  - α **(latency):** fixed time to initiate a message, independent of its size

  - β **(per-byte cost):** time per-byte (approx. 1/bandwidth)

- For tensor I/O, tensors are **chunked over multiple channels** to increase effective bandwidth and then submitted in parallel. Under ideal parallelism for n channels, it writes: $T_{\text{comm}}(m, n) = \alpha + \beta \cdot \dfrac{m}{n}$
- The chunk size $\dfrac{m}{n}$ must be large enough to amortise the cost of α.

Communication bandwidth

- There are multiple hardware links (NVLink) combined with software optimization, so there is a trade-off between few large chunks and many small chunks (amenable to overlap) leading to

$$T_{\text{comm}}(m) = \alpha + \beta_{\text{eff}} \cdot m$$

- For JeanZay, we end up with values in the range of
  - CPU - GPU: $\alpha \approx 5\mu s$ $\quad \frac{1}{\beta_{\text{eff}}} \approx 25\text{GB}/s$
  - GPU - GPU (intra-node): $\alpha \approx 1\mu s$ $\quad \frac{1}{\beta_{\text{eff}}} \approx 100\text{GB}/s$
  - GPU - GPU (extra-node): $\alpha \approx 10\mu s$ $\quad \frac{1}{\beta_{\text{eff}}} \approx 300\text{GB}/s$

NCCL: A Protocol to Make GPUs Communicate

- I'll focus on NVIDIA environments which are more common.

- **NVIDIA Collective Communications Library (NCCL "Nuh-kuh-l")** is a communication library used to make GPUs exchange data efficiently, typically over **NVLink, PCIe, or InfiniBand**.

- It can **treat multiple GPUs as one large accelerator** by hiding link and topology details behind a simple **collective communication API** (e.g., all-reduce, broadcast, all-gather).

- NCCL is a **CUDA-aware library**: communication is implemented with **GPU kernels**, runs in parallel, and can overlap with computation. It's often a backend of libraries like PyTorch.

- Many communication details are hidden from the user (e.g., **gradient bucketing / tensor fusion** in deep learning frameworks), but performance improves when the user keeps **tensors contiguous and well-sized**.

- NCCL can select the best communication algorithm and pattern for the current topology and message size, often by probing/benchmarking available options.

One word about the I/O on a cluster

- Can our input pipeline deliver *well-shuffled* (say, i.i.d.) minibatches fast enough to keep GPUs 100% busy?

- There is again a latency/throughput trade-off: we need to **keep batch latency low** (fast time-to-first-byte) so the **prefetch queue stays full** and GPUs never wait; then to use **high throughput** to stream enough data to maintain shuffle quality and utilization.

- Storage options (typical trade-offs), assuming an "offline" shuffling:

  - Object storage (Amazon S3): very high throughput, but higher per-request latency; pair with node-local RAM cache.

  - Shared filesystem (on Network): very high aggregate throughput, low latency

  - **Local RAM: lowest latency** per node; great as a **cache**.

Bonus: Sharing Randomness Is (Almost) Free

- Real data is expensive to send between workers... but shared randomness is cheap.

- How?

- Simply share the random seed once...

Parallelising DNN training

The need for a metric to measure efficiency

- In High-Performance Computing (HPC), we often talk about:
  - Strong scaling: Fix the problem size, increase the number of workers and measure how the time to solution decreases
  - Weak scaling: Increase the **problem size** proportionally to the number of workers and keep the work per worker constant and measure whether the **time to solution** stays constant
- In deep learning, "problem size" is not fixed: more GPUs = larger batch size or bigger models.
  - Classic strong/weak scaling only tell you about **throughput** (and clearly not model quality).
  - It measures only parallel efficiency and it doesn't take in account for the hardware usage.

Model FLOPs Utilization

- A simple, efficient metric which allows to take in account for the efficiency of a ML algorithm is the Model FLOPs Utilisation:

$$\text{MFU} = \frac{\text{FLOPs your model actually executes per second}}{\text{Theoretical peak FLOPs of the hardware}}$$

- MFU tells us how close we are to not wasting FLOPs.

- During the previous lectures, we saw that typically, for a model of size P:

$$\text{MFU} \approx \frac{6P \cdot \text{Tokens per second}}{\text{Theoretical peak FLOPs of the hardware}}$$

Example of MFUs

- Tuning a cluster to maximize MFU is a real art.
- The typical ranges spans 20% to 60%.
- For instance, the Herds of LLama3 models report a MFU of 40%
- Megatron DeepSpeed report on a single worker a MFU of 55%
- DeepSeek-v3 has probably a MFU of 45%.

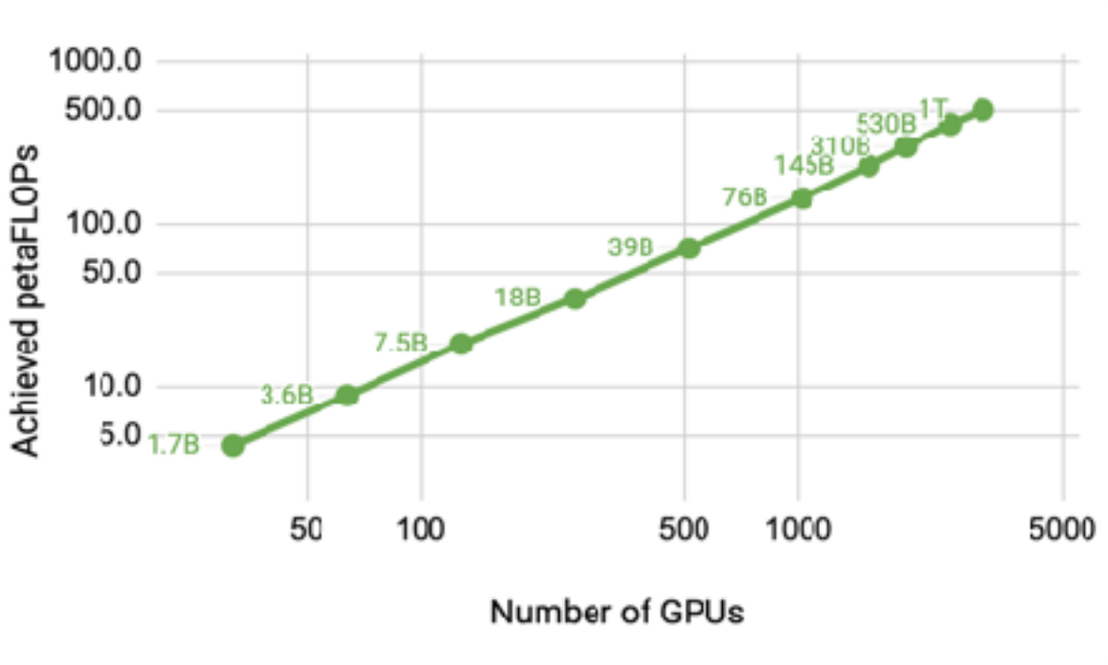
The tradeoffs of a good MFU

- Good MFU has to deal with 3 trade-offs:
  - Memory
  - Communications
  - Statistical efficiency
- We'll see this at the next lecture!

The software eco-system

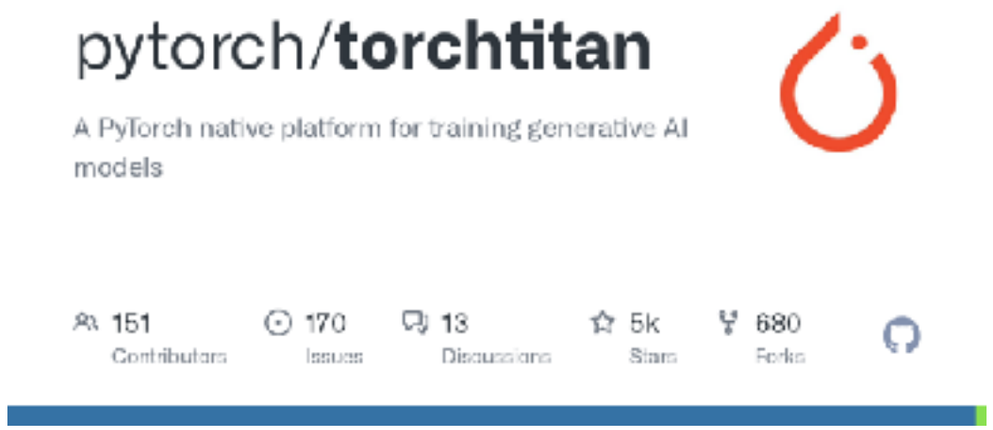
Megatron+DeepSpeed

- Megatron-LM: large-scale Transformer training with tensor, pipeline, and sequence parallelism
- **DeepSpeed**: system-level optimizations (ZeRO-1/2/3, optimizer & activation sharding, offload)
- Combined stack:
  - Megatron handles **model parallelism**
  - DeepSpeed handles memory efficiency & scaling
- Proven at **very large scale** (multi-node, multi-GPU)
- Powerful but **complex to configure** and tightly coupled to Transformer architectures



TorchTitan

- **TorchTitan**: PyTorch-native framework for large-scale LLM training
- Built on **PyTorch** distributed primitives
- Emphasizes:
  - **Composability** over monolithic design
  - Clear separation between **model code** and **parallelism**
- Designed to be:
  - Easier to extend
  - Easier to reason about
  - Less architecture-specific than Megatron



- Still **younger** / **evolving**, fewer battle-tested recipes at extreme scale

Ecosystem Around TorchTitan (Tools & Integrations)

- **PyTorch distributed stack:** `torch.distributed` + FSDP/DTensor …

- Fault tolerance: **TorchFT** for resilient, restart/continue training on failures

- Post-training / finetuning: **TorchTune**

- Ecosystem glue: **Hugging Face** (models, datasets, tooling interoperability)

Conclusion

- Let's try out TorchTitan!