The Challenges of Scaling

Training Time ◂

Memory Footprint ◂

Solutions ◂

Energy efficiency ◂

Training / Inference

Training

- Forward → loss → backward → update
- Stores: parameters, gradients, optimizer states, activations
- Key metrics: throughput (images/s, tokens/s), time to target accuracy

Inference

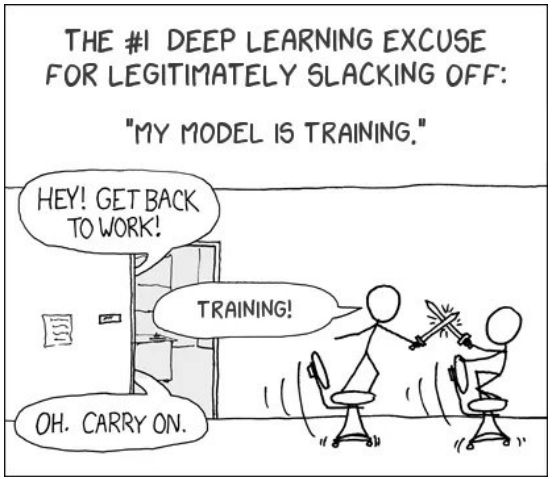
- Forward only → much less memory
- Model can be compressed: quantization, pruning, distillation…→ Day 4
- Key metrics: latency, throughput, memory (+ KV cache for LLMs)

Jean Zay: mostly training (+ evaluation), no model serving
→ DLO-JZ focuses on training

Constraints of Deep Learning

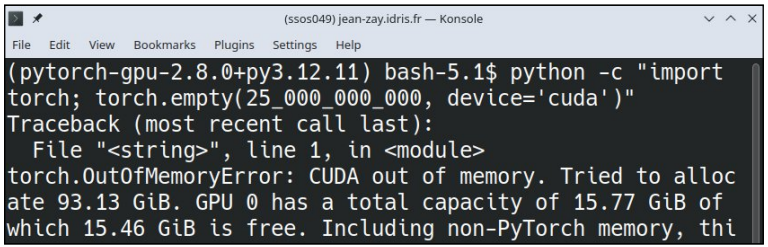
Two problems to deal with

Training Time



Adapted from xkcd #303

GPU memory (Out Of Memory)





Training Time

Training Resnet-50 on Imagenet



Goal:

Classification (1000 classes)

Model :

25 M parameters

Dataset:

1.2 M labeled images

2017

14 Days

1 NVIDIA M40 GPU

Training Time

⚠ **Wall-clock time ≠ GPU-hours**
2017: 1 M40, 14 days → 336 GPU-h
2019: 2 048 V100, 1.2 min → 41 GPU-h
Time ÷ 17 000, GPU-hours ÷ 8

Training Resnet-50 on Imagenet

Facebook Caffe2	**UC Berkeley, TACC, UC Davis** Tensorflow	**Preferred Network** ChainerMN	**Tencent** TensorFlow	**Sony** Neural Network Library (NNL)	**Fujitsu** MXNet
1 hour	31 mins	15 mins	6.6 mins	2.0 mins	1.2 mins
Tesla P100 x 256	1,600 CPUs	Tesla P100 x 1,024	Tesla P40 x 2,048	Tesla V100 x 3,456	Tesla V100 x 2,048
Apr	Sept	Nov	July	Nov	Apr

2017 2018 2019

Fast.ai tips and engineering

2017 → 14 Days (on 1 GPU) → 2018

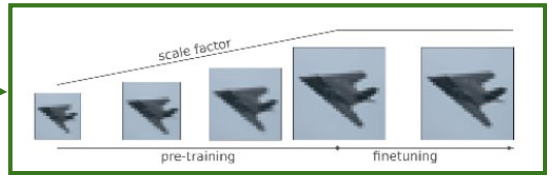
« Now anyone can train Imagenet in 18 Minutes »

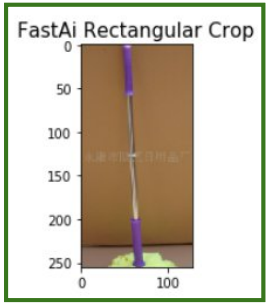
fast.ai - 2018/08/10

From fast.ai & Defense Innovation Unit (DIU) team

- 16 AWS instances × 8 V100 = 128 GPU × 18 min ~ 38 GPU-h => **~40 $**
- Combination of techniques :
  - progressive resizing (fast.ai)
  - rectangular image validation (fast.ai)
  - NCCL/all-reduce (NVIDIA) → Day 2
  - weight decay tuning (Tencent) → Day 3
  - dynamic batch sizes (inspired by Google Brain) → Day 3
  - gradual learning rate warm-up → Day 3

  (Goyal et al 2018, and Leslie Smith 2018)





Training Time: BLOOM

2022

a BigScience initiative

BLOOM

176B params · 59 languages · Open-access

Goal:

Text Generation Foundation Model (LLM)

Model :

176 B parameters

Dataset:

366 B tokens

117 Days

384 A100 GPUs



1.08 M GPU-hours

~ 4×10^{23} FLOP

Training Time: Llama 3.1



Llama 3.1 405B



Goal:

Text Generation Foundation Model (LLM)

Model :

405 B parameters

Dataset:

15 T tokens

72 Days (est.)

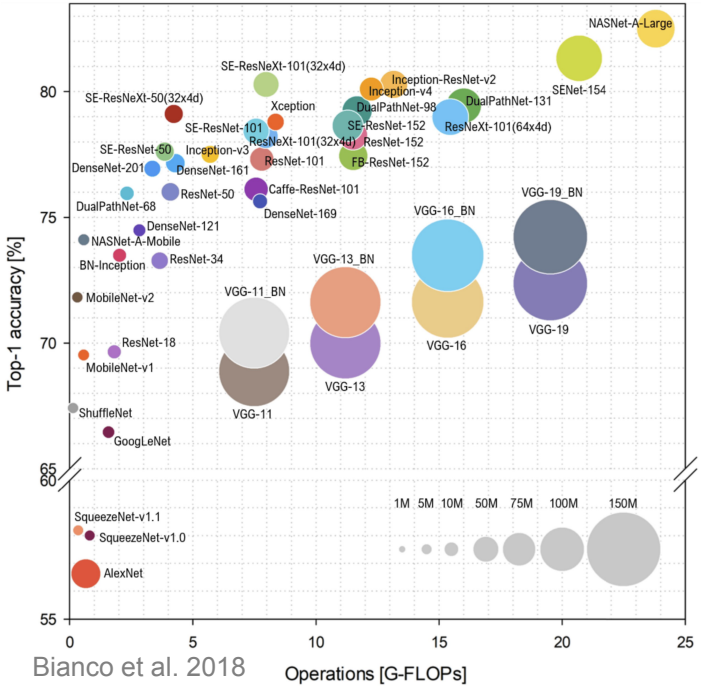
up to 16 384 H100 GPUs

30.8 M GPU-hours

3.8×10^{25} FLOP

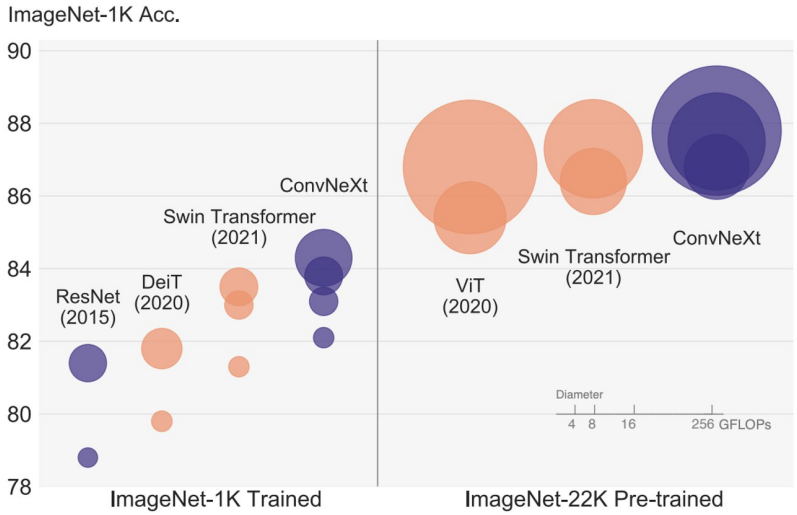
Large Model (Vision)

Vision models on ImageNet (bubble size = nb of params)



Bianco et al. 2018

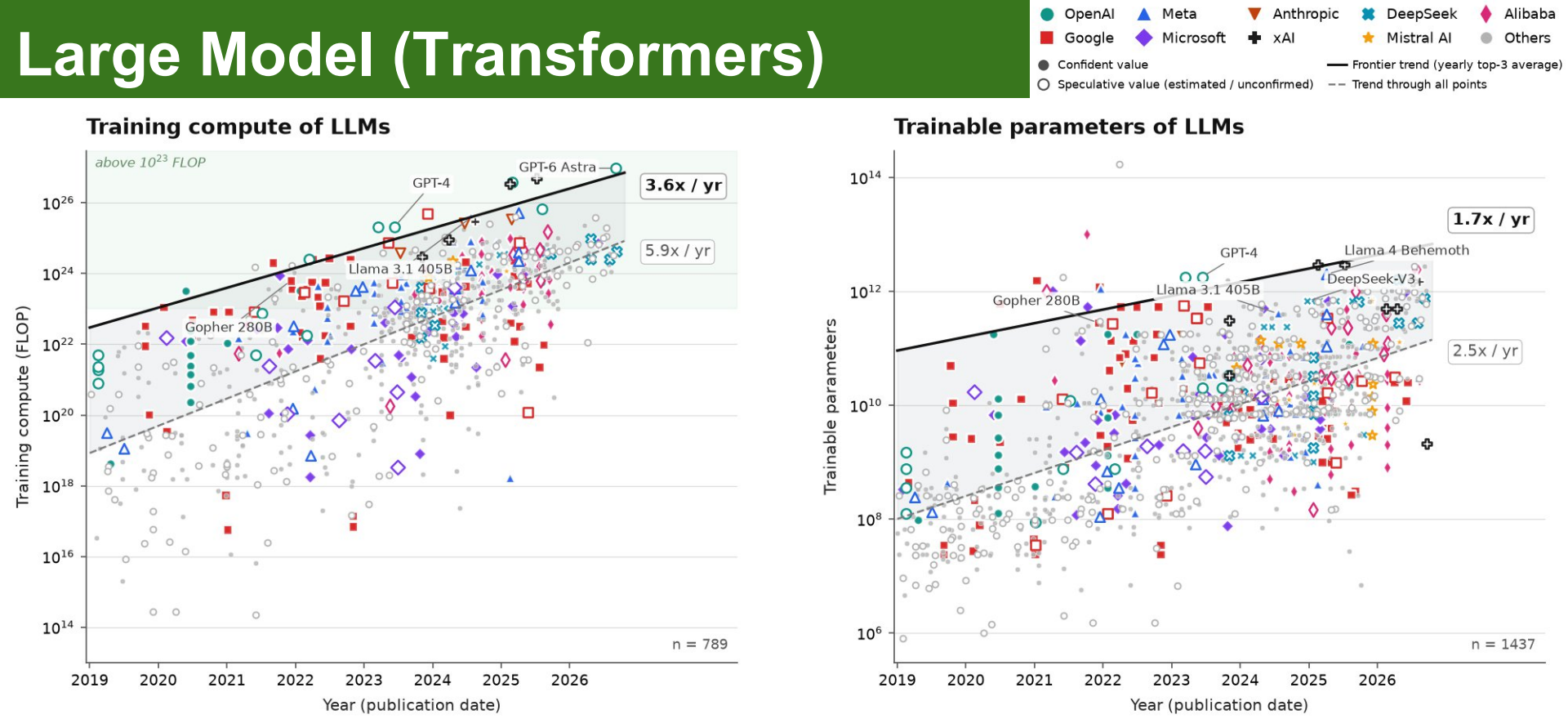
Effect of training set size (bubble size = FLOPs)



ConvNeXt (Liu et al. 2022)

Bigger models → better accuracy
(with enough data)… but architecture matters:
VGG-16 (138M params) < ResNet-50 (25M)

Large Model (Transformers)

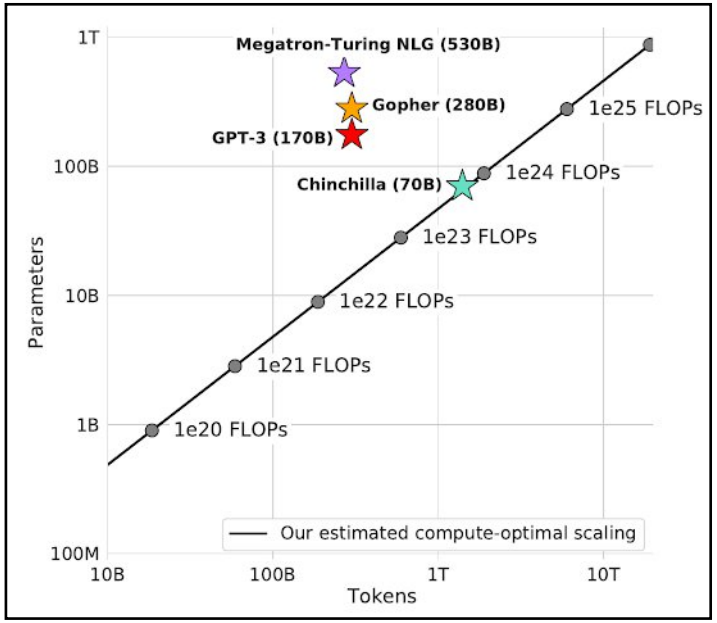


Huge models → long training times and large memory footprint:
1 B parameters = 4 GB in FP32 (2 GB in BF16)
…weights only!

Data from epoch.ai

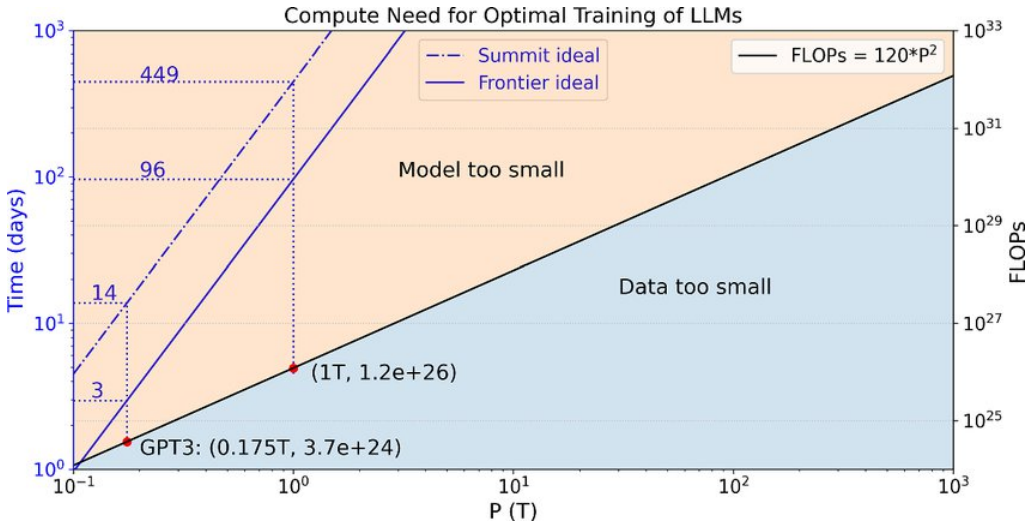
Large Model (Transformers)

Chinchilla Law



$N_{\text{processed tokens}} \sim 20 \times N_{\text{params}}$

Impact



Common approx. for a dense Transformer:
computing cost ~ $6 \times N_{\text{params}} \times N_{\text{processed tokens}}$

Computing cost ~ $120 \times N_{\text{params}}^2$
→ 10× more parameters = 100× more compute

Training compute and wall-clock time

time ≈ Compute needed (FLOP) ÷ Compute speed (FLOP/s)

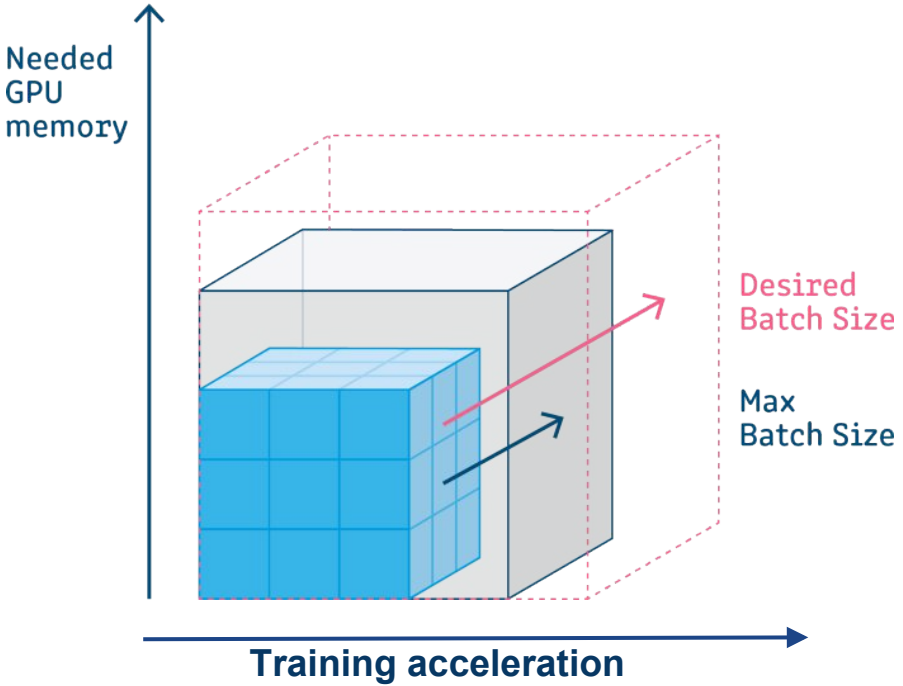
Compute needed grows with:

- model size (parameters, depth)
- data seen = dataset size × number of epochs
- input size: image resolution, sequence length (attention $\propto L^2$)

Compute speed = nb of GPUs × peak FLOP/s per GPU × efficiency
(efficiency < 100 %: data loading, I/O, communications, syncs…)

How to speed up? 1) Less compute 2) Better GPUs
3) More GPUs 4) Better GPU usage ← DLO-JZ

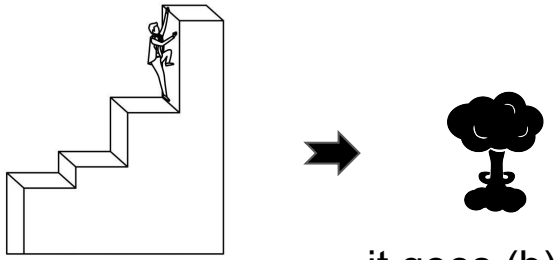
Batch Size & Memory Usage



Bigger batch

- GPU better filled
- more images/s (and fewer steps/epoch)

But memory grows with the batch, until...



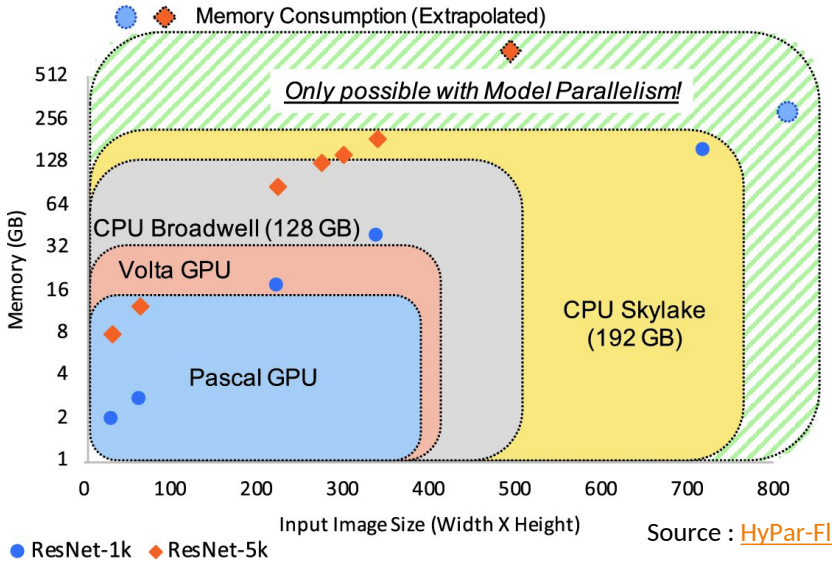
...it goes (b)OOM

Effect on convergence: not free → Day 3

Large inputs / long sequences

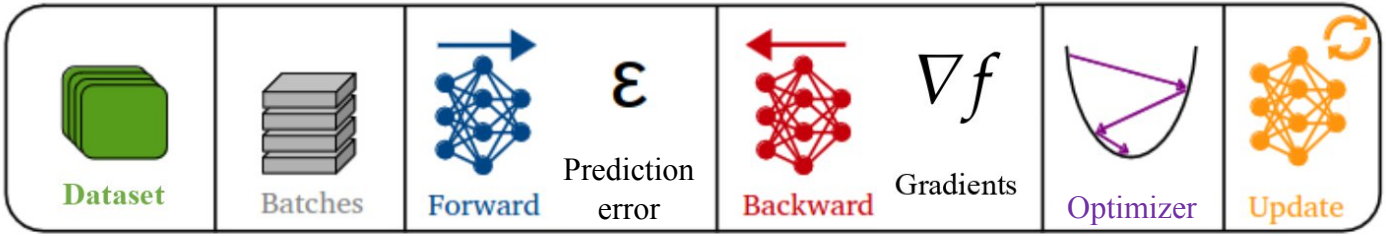
Large input data causes **serious memory occupancy problems** during training, **accentuated by the depth of the model**.

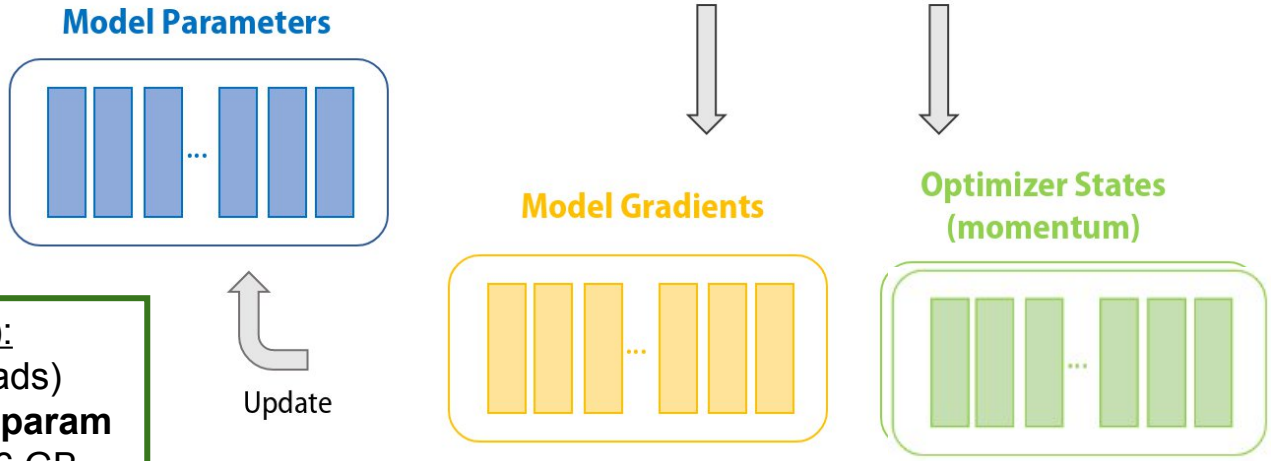
- ➔ Text (N, 100, 500) ***~x1***
- ➔ Image 2D (N, 224, 224, 3) ***~x3***
- ➔ Image 3D (N, 224, 224, 100, 3) ***~x300***
- ➔ Video (N, 100, 224, 224, 3) ***~x300***
- ➔ LLM, long context (N, 128k, 4096) ***~x10 000***



Source : HyPar-Flow

Model Memory Occupancy

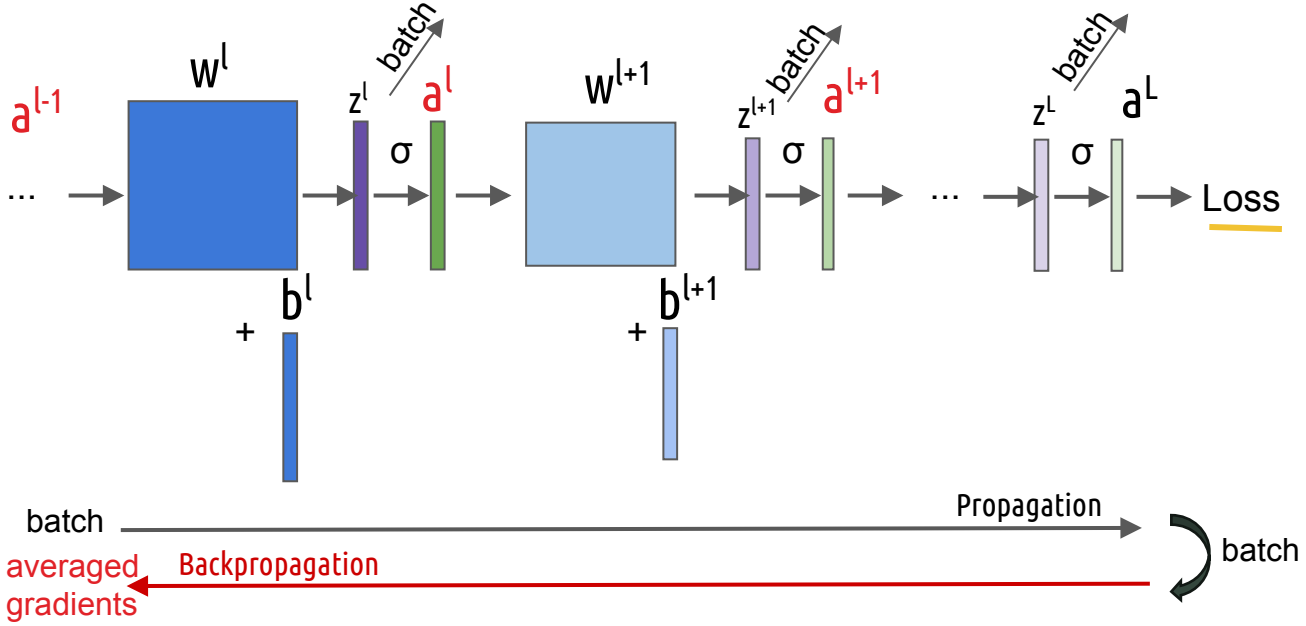




Rule of thumb (Adam):
4 bytes (weights) + 4 (grads)
+ 8 (optimizer) = **16 bytes / param**
thus 1 billion params → 16 GB

× 7 B params **≈ 112 GB > one H100**
...before activations!

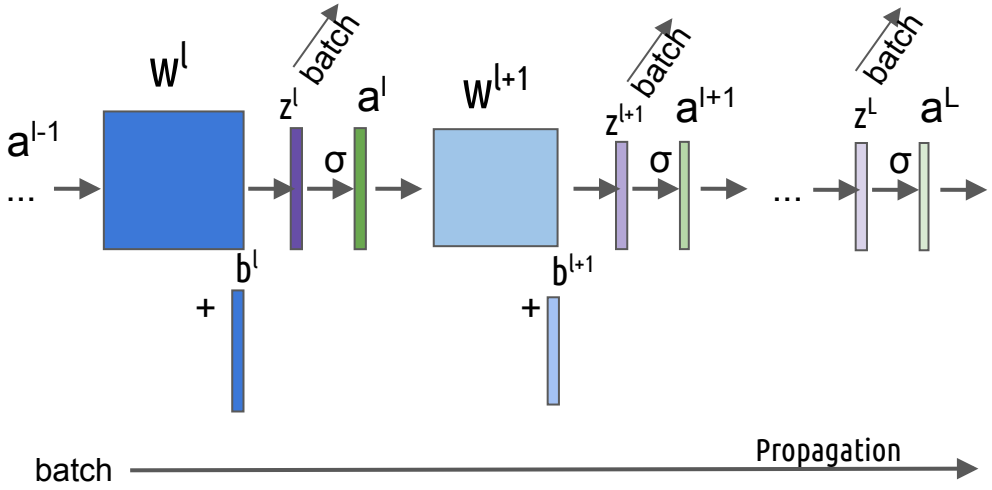
Activations Memory Occupancy



Note: For backpropagation, it is necessary to save intermediate activations.

Activation memory ∝ batch size × input size × depth

Inference & evaluation



Propagation

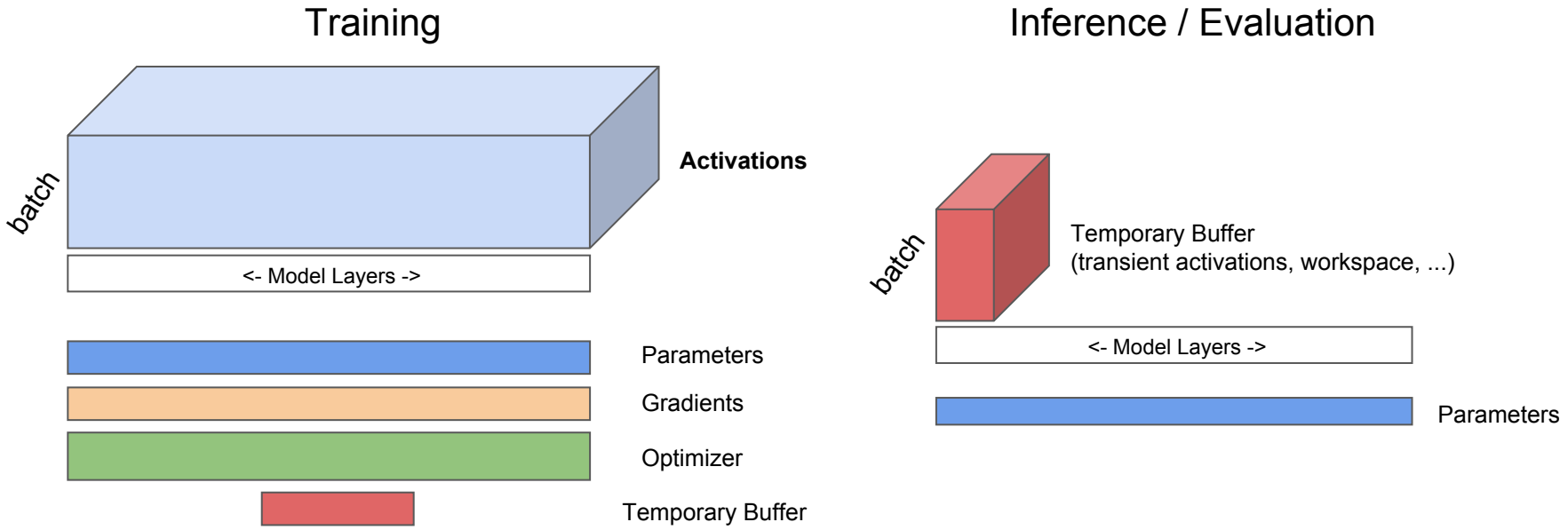
$$a^l = \sigma(w^l a^{l-1} + b^l) = \sigma(z^l)$$

```
…
model.eval()                        # dropout / BatchNorm
                                      in eval mode
with torch.inference_mode():        # no autograd graph →
                                      activations not kept
    val_outputs = model(val_images)
    loss = criterion(val_outputs, val_labels)
model.train()                       # back to training mode
…
```

No backward

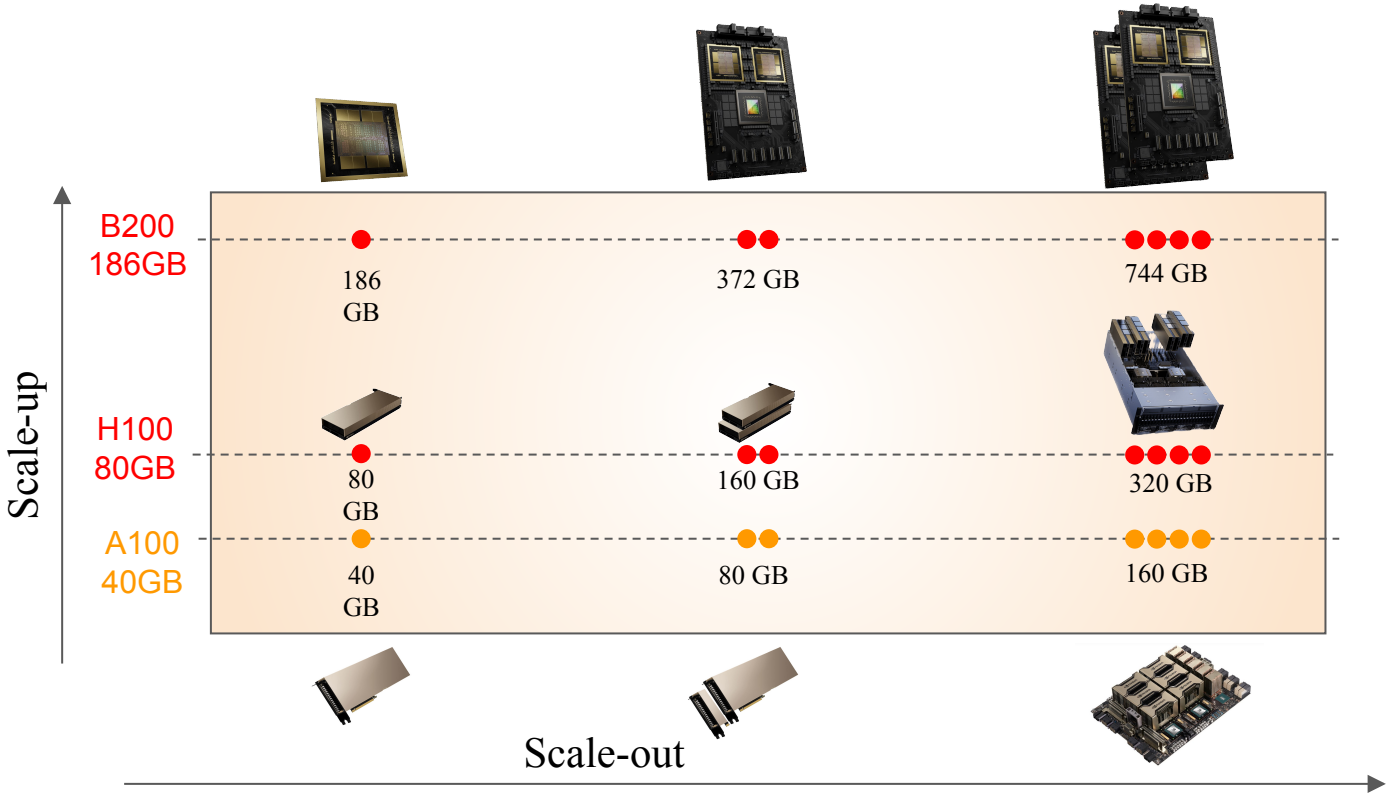
- no stored activations
- much less memory
- (bigger validation batch possible)

Memory Footprint



Small models: activations dominate
Huge models: parameters + gradients + optimizer alone can exceed GPU memory

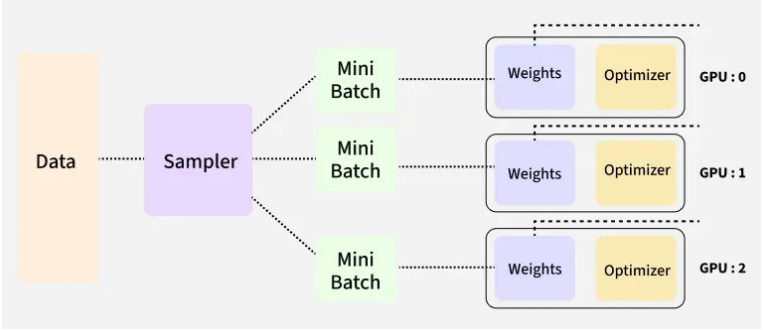
System Solutions



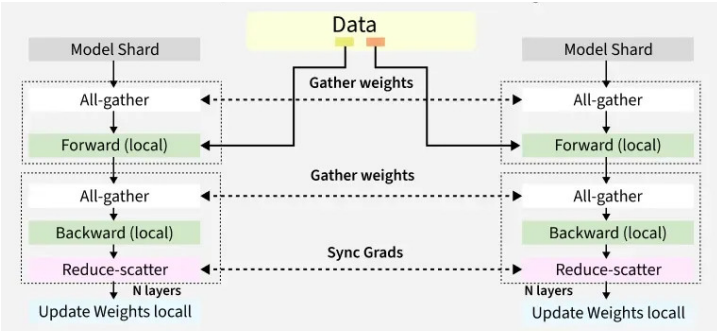
Scale-up = better GPUs ; Scale-out = more GPUs

Distribution / Scale-out

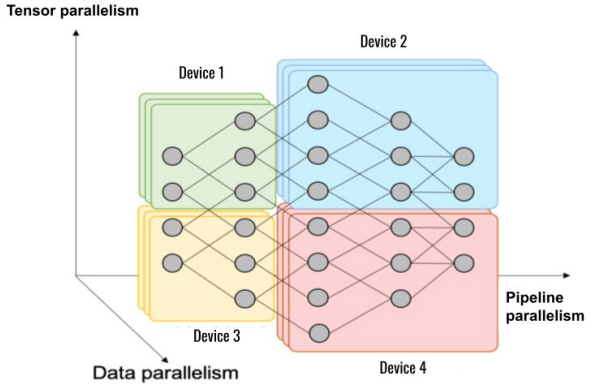
Distributed Data Parallelism (DDP) - Day 2



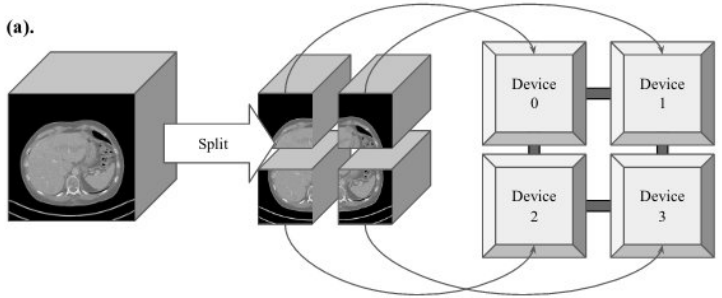
Sharded data parallelism (FSDP) - Day 4



Model Parallelism (Tensor + Pipeline) - Day 4

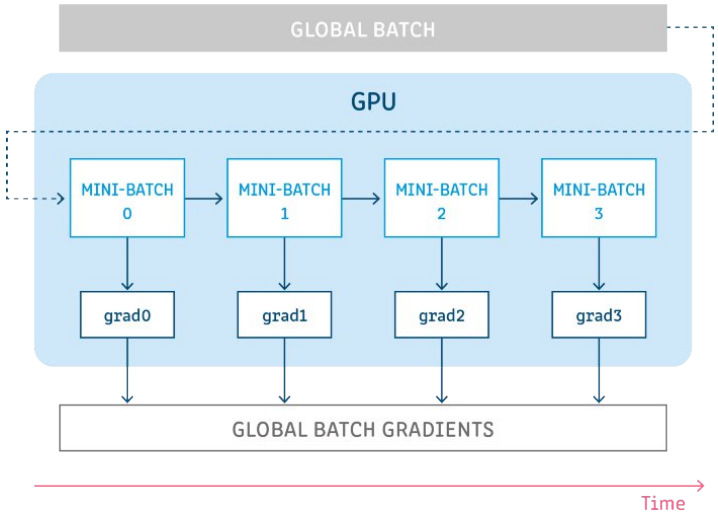


Spatial/domain parallelism



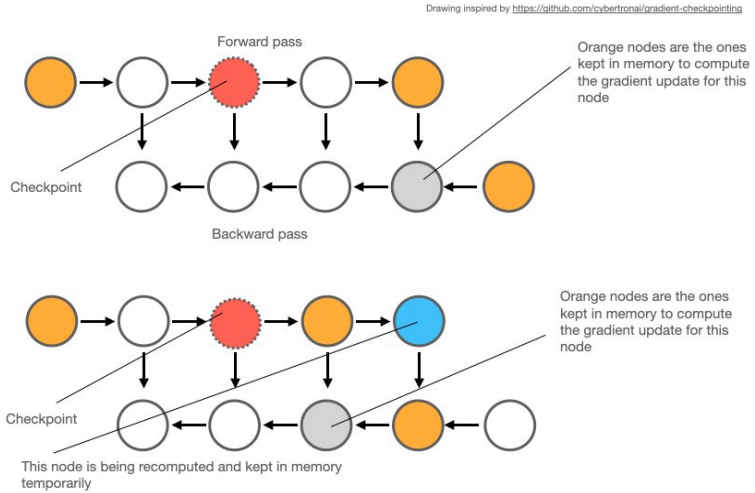
Workaround Solutions

Gradient accumulation



Large effective batch with little memory but sequential (no speed-up)

Gradient / Activation checkpointing

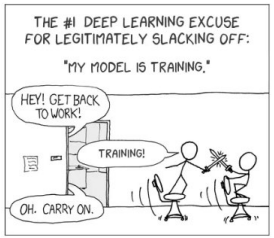


Keep only some activations, recompute the others in backward
≈ +30 % compute

A 3rd problem to deal with ...

Two problems to deal with

Training Time



Adapted from xkcd #303

GPU memory (Out Of Memory)

```
(pytorch-gpu-2.8.0+py3.12.11) bash-5.1$ python -c "import
torch; torch.empty(25_000_000_000, device='cuda')"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
torch.OutOfMemoryError: CUDA out of memory. Tried to alloc
ate 93.13 GiB. GPU 0 has a total capacity of 15.77 GiB of
which 15.46 GiB is free. Including non-PyTorch memory, thi
```



Energy consumption

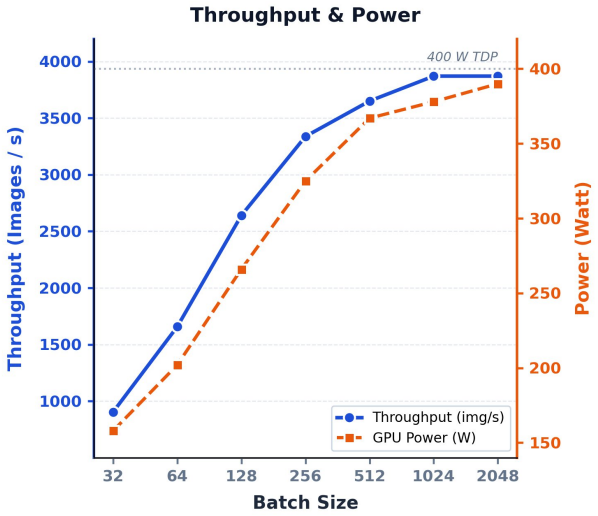


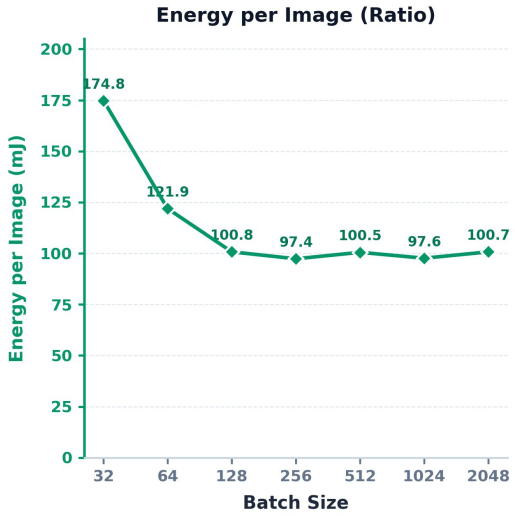
Power draw & energy efficiency

	H100 PCIe	H100 SXM5	A100 PCIe	A100 SXM4	V100 PCIe	V100 SXM2
Max Power	350W	700W	250W	400W	250W	300W
Idle Power	~30W	~55W	~30W	~60W	~40W	~45W
Perf. (vs V100 SXM2)	x3.5	x4	x1.75	x2	x0.9	x1

For a node:
The CPU (often 2 processors) consumes what approximately 1 GPU consumes.

A100 SXM4 GPU





- Bigger batch → more watts… but even more images/s
- Batch 32 → 1024:
  ≈ 45 % less energy per image
- A fully used GPU is the most energy-efficient

Energy Consumption / GPU-hours

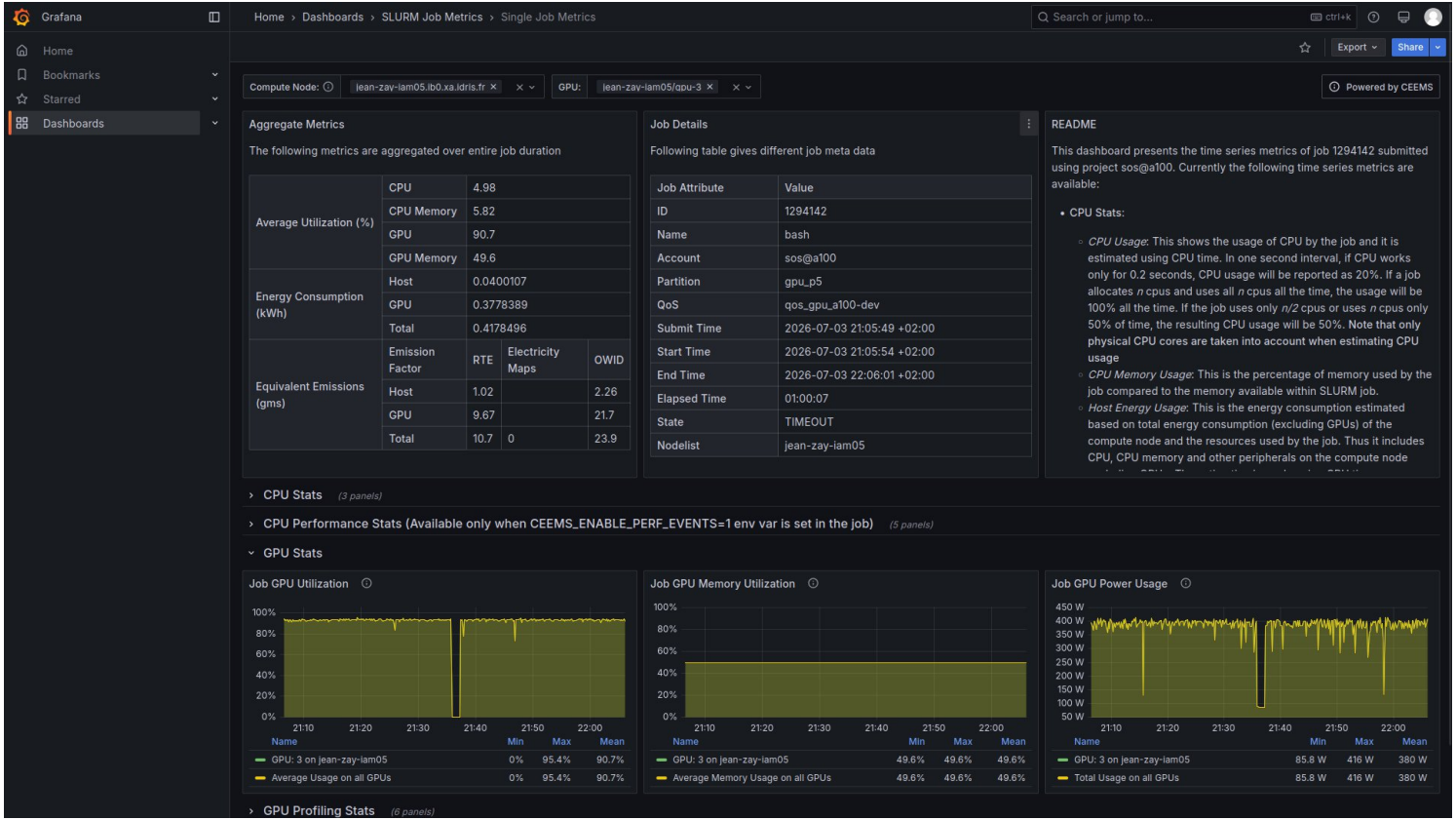
Energy Saving
≅
GPU-hours Saving

System optimization (DLO-JZ)

- Maximize throughput (without losing accuracy)
- Remove GPU idle time (e.g. waiting for data)
- Use the right number of GPUs: neither too many nor too few

Energy Consumption / GPU-hours

Introducing: CEEMS



Energy Consumption / GPU-hours

Energy Saving
≅
GPU-hours Saving

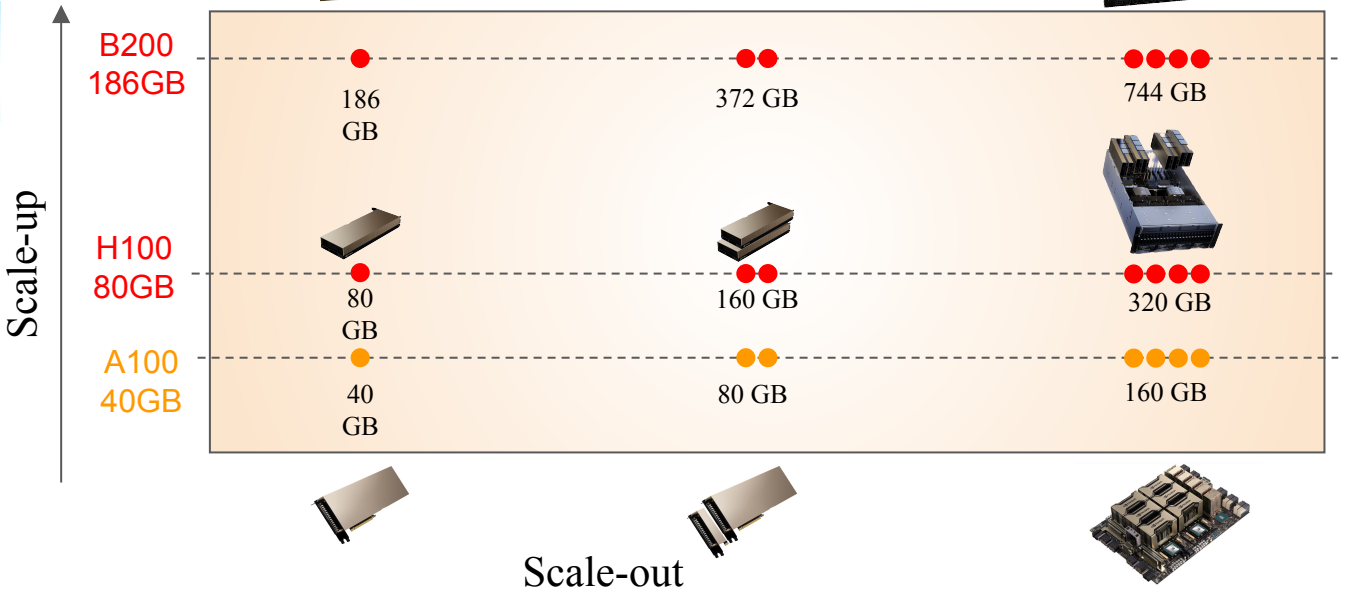
System optimization (DLO-JZ)

- Maximize throughput (without losing accuracy)
- Remove GPU idle time (e.g. waiting for data)
- Use the right number of GPUs: neither too many nor too few

Energy Consumption / GPU-hours

More FLOP
per watt ✓

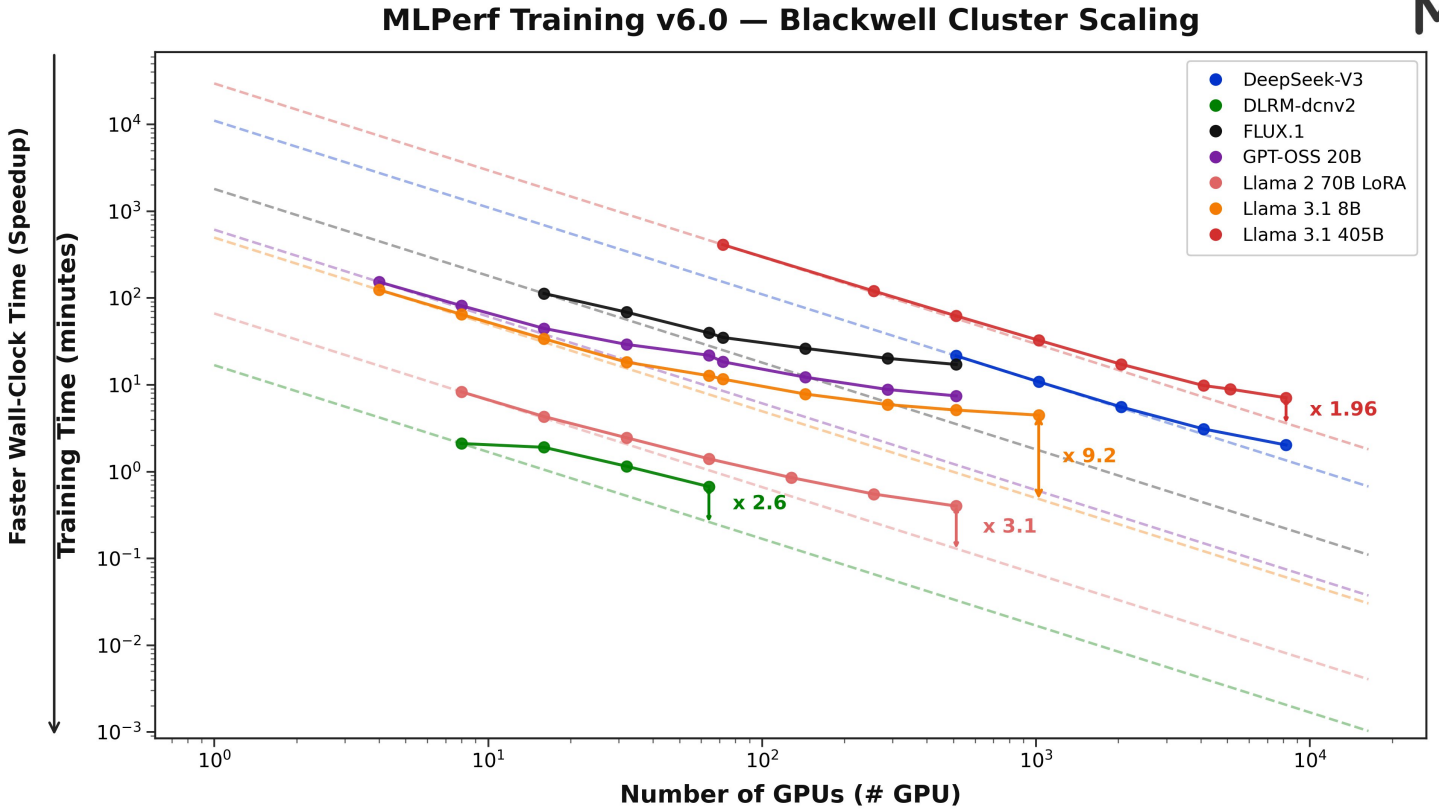




Communication overhead
→ more GPU-hours ✗



ML Perf Result - Scaling



Dashed = ideal scaling · Solid = measured · Gap = wasted GPU-hours
(best case: vendor-optimized code)

ML Perf Result - Scaling

Optimizations fall into four major categories:

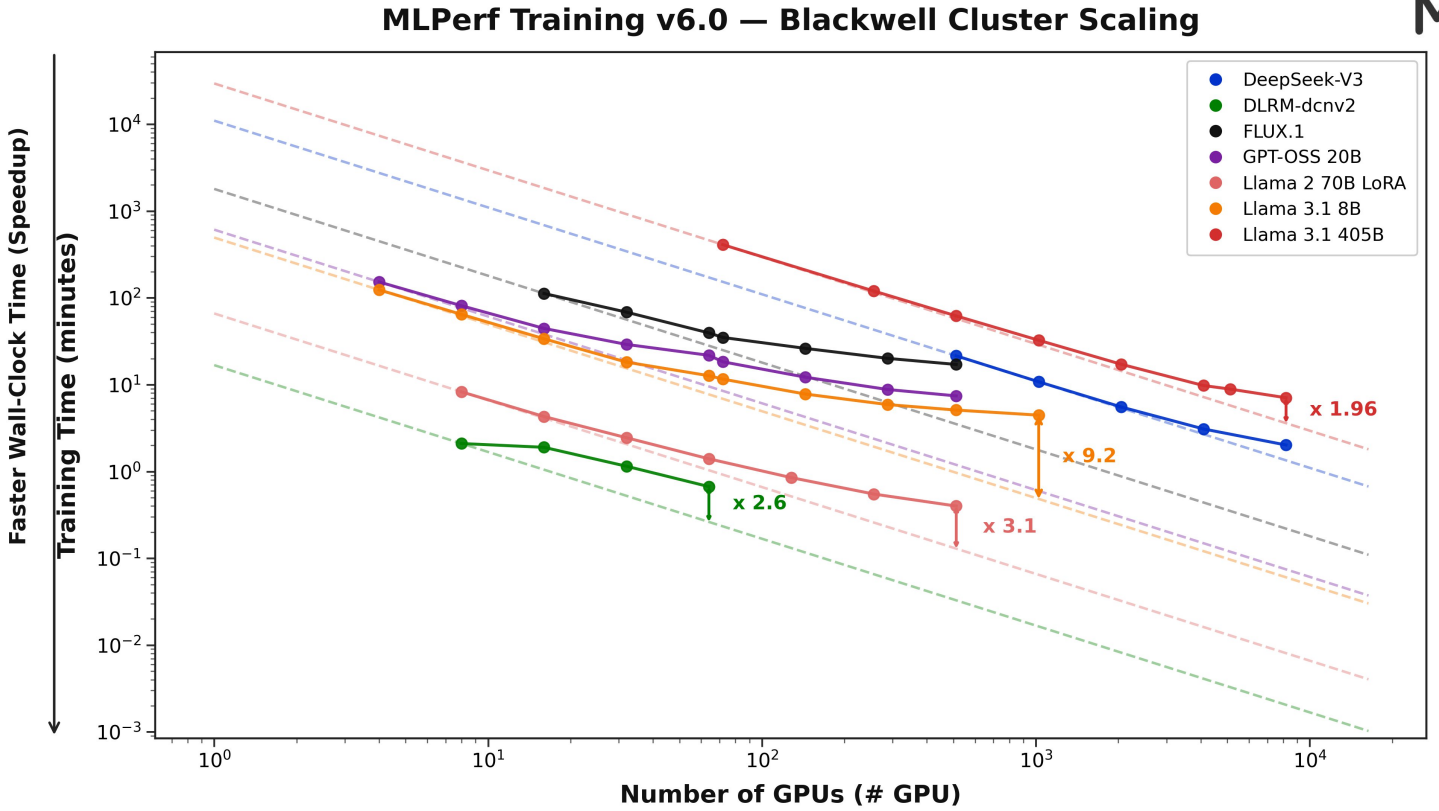
A. Low-Precision Numerics & 2nd-Gen Transformer Engine
- FP8 Mixed-Precision Training (E4M3 & E5M2)
- Block-Level Dynamic Scaling
- Tensor Memory Accelerator (TMA) & Asynchronous Warp Specialization

B. 5D Distributed Parallelism & Hybrid Schedules
1. NVLink-5 Domain Tensor Parallelism (TP):
2. Zero-Bubble Pipeline Parallelism (ZB-PP):
3. Dual-Pipe Expert Parallelism (EP) for MoEs:
4. Context Parallelism (CP):
5. Fully Sharded Data Parallelism (ZeRO-3 / FSDP):

C. Model-Specific Algorithmic & Kernel Breakthroughs

D. In-Network Collective Acceleration
- SHARP (Scalable Hierarchical Aggregation and Reduction Protocol)
- CUDA Graphs

ML Perf Result - Scaling



Dashed = ideal scaling · Solid = measured · Gap = wasted GPU-hours
(best case: vendor-optimized code)

Energy : Methodology

Methodology: avoid unnecessary trainings

1. Reuse hyperparameters from publications (reproduce the state of the art)

2. Tune on small models, then scale up

3. Only then: automatic HPO (→ Day 4)