ML Engineering Fundamentals

김응서

February 11, 2026

SNU BI Lab

목차 i

1. Part 1. Key Insights

2. Part 2. Hardware

3. Part 3. Training

4. Part 4. Inference

5. Part 5. Containers

Part 1. Key Insights

What's Important in the AI Race?

Training:

1. How fast one can train a better model
   (first to market advantage)
2. How much $$ was spent
   (do we still have money left?)

Inference:

1. Fast latency
   (msec response times)
2. Fast throughput
   (concurrent queries)
3. How much $$ per user
   (can we scale?)

What are the Needs of LLM Training?

1. **Fast compute** massively dominated by matrix multiplications
2. **Fast enough** memory, IO, network and CPU to feed the compute

생활 지혜

가장 빠른 가속기에 돈 부으면서 다른 구성요소는 싼걸로 때우면, 돈만 낭비하는 걸 수도

What are the Workhorses of ML?

- An **accelerator** or **processing unit** does most of the work
- ML does a lot of parallel processing (SIMD)

Available accelerators:

- GPUs (NVIDIA, AMD)
- TPUs (Google)
- IPUs, FPGAs, HPUs, QPUs, RDUs
- Recent CPUs (especially for inference)

Can You Feed the Furnace Fast Enough?

Steam locomotive fireman

The engine is great, but if the fireman isn't fast enough to shovel the coal in, the train won't move fast.

Figure 1: *

Source: Wikimedia Commons

The bottleneck is in moving bits, not compute!

- Accelerators: ~2x faster every 2 years
- Network & memory: NOT!
- IO can be another bottleneck
- CPU is fine (enough cores)

생활 지혜

계산 빠르다고 다는 아녀

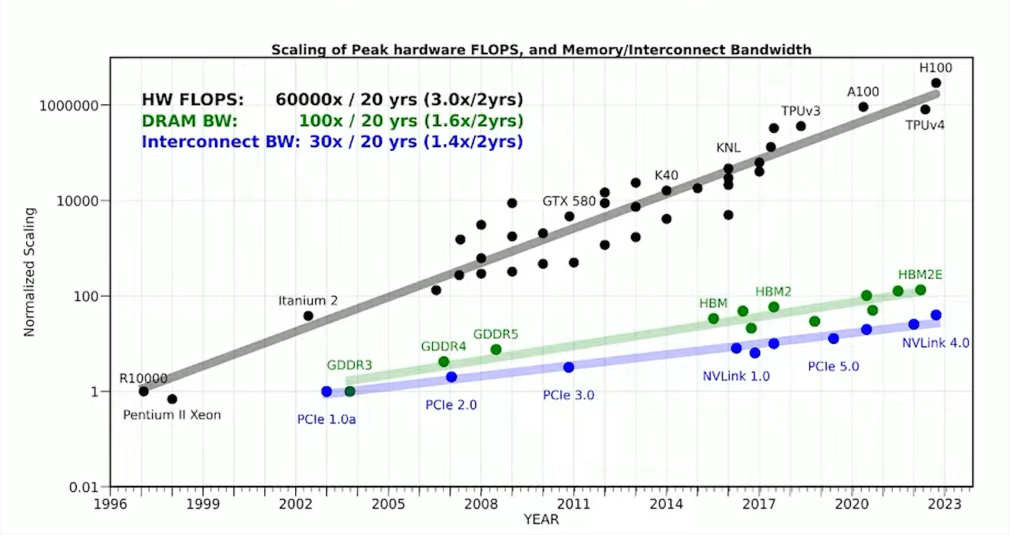

Figure 2: *

FLOPS vs Bandwidth: Compute grows faster than memory bandwidth

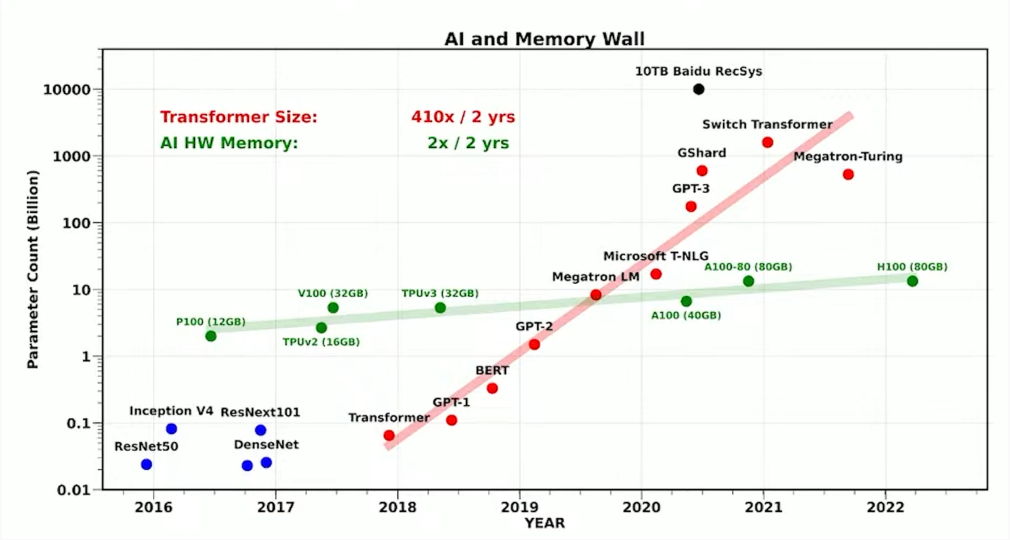

Figure 3: *

AI and Memory Wall: The growing gap between compute and memory

What is TFLOPS?

Definition

TFLOPS = **T**era **FLO**ating point **O**perations **P**er **S**econd

- One trillion (10^{12}) floating point operations per second
- Key metric for measuring accelerator (GPU/TPU) performance

Examples:

- NVIDIA A100 (BF16): 312 TFLOPS
- NVIDIA H100 (BF16): 989 TFLOPS
- NVIDIA B200 (BF16): 2,250 TFLOPS

TFLOPS: Estimating Training Cost

Calculate time needed:

$$\text{time\_in\_secs} = \frac{\text{total\_tflops\_required}}{\text{tflops\_of\_compute\_unit}}$$

Example: 604,800 secs = 7 days

1. Look at the cost for 7 days ⇒ total $$ to train
2. Compare other proposals
3. Choose the best option

생활 지혜

최대 TFLOPS는 보통 달성 불가능! MFU를 보는게 나음.

Model FLOPS Utilization (MFU)

How well is the accelerator utilized?

$$\text{MFU} = \frac{\text{actual\_TFLOPS}}{\text{advertised\_TFLOPS}}$$

Example: 80B model on BF16@A100

- Single iteration: 624 TFLOPS in 4 secs $\Rightarrow$ 156 actual TFLOPS
- A100 advertised: 312 TFLOPS
- MFU: $156/312 = 0.5 =$ **50%**

생활 지혜

- NVIDIA GPUs: >50% MFU on multi-node = 굿
- If MFU is 50%, training takes 2x longer than estimated

Moving Bits: The Real Bottleneck

Why can't we achieve advertised TFLOPS?

- Takes time to move data between accelerator memory and compute
- Even more time from disk and other GPUs to accelerator memory

Solutions:

- Fused and custom kernels (`torch.compile`, flash attention)
- Single GPU: only accelerator memory matters
- Multiple GPUs: network becomes the bottleneck
- Efficient frameworks overlap compute and comms

Network Speed Comparison

NVIDIA NVLink (intra-node, unidirectional):

GPU	Compute (TFLOPS)	Speedup	Network (GBps)	Speedup
V100	125	1x	150	1x
A100	312	2.5x	300	2x
H100	989	8x	450	3x
B200	2250	18x	900	6x

Recap: Key Issue

Compute grows faster than network! Network is the bottleneck.

Intra-node vs Inter-node Network

Intra-node (within node):

- Fast: NVIDIA NVLink 900 GBps (B200), 450 GBps (H100)
- Intel Gaudi2: 2.1 TBps total
- For Tensor/Sequence parallelism

Inter-node (between nodes):

- 200 Gbps to 6400 Gbps range
- Originally 10x slower than intra-node
- Now approaching intra-node speeds (800 GBps EFA)
- Expect $\sim$80% of advertised speed

Storage Needs

Three distinct IO needs:

1. **DataLoader feeding** - super fast read, sustained for hours/days
2. **Checkpoint writing** - super fast write, burst mode
3. **Codebase** - medium speed read/write, shared across nodes

생활 지혜

You're being sold 80% of what you paid. For reliable 100TB, rent 125TB!

CPU and CPU Memory (8 GPUs 기준)

CPU Memory needs:

- 2-3 DL workers per accelerator (16-24 processes for 8 GPUs)
- More memory if pulling data from cloud
- Enough to load model if not loading to accelerator directly
- For accelerator memory offloading - more is better!

CPU:

- 대충 at least 30 cores (2-3 DL workers + 1 per GPU)
- More cores for OS thread, Tokenization, Unzipping, etc.
- Most compute happens on GPUs anyway

How Many GPUs Do You Need? (5초컷)

Training in half mixed-precision:

$$\text{GPUs} = \frac{\text{model\_size\_in\_B} \times 18 \times 1.25}{\text{gpu\_size\_in\_GB}}$$

Inference in half precision:

$$\text{GPUs} = \frac{\text{model\_size\_in\_B} \times 2 \times 1.25}{\text{gpu\_size\_in\_GB}}$$

Example: 80B model on 80GB GPUs

- Training: $80 \times 18 \times 1.25/80 = 23$ GPUs
- Inference: $80 \times 2 \times 1.25/80 = 3$ GPUs

* 이유는 후술할 예정

Part 2. Hardware

Accelerators - The Workhorses of ML

Evolution:

- Started with GPUs
- Now: TPUs, IPUs, FPGAs, HPUs, QPUs, RDUs, and more

Two main workloads:

- **Training** - one huge matmul per sequence
- **Inference** - thousands of small matmuls one token at a time
- **Finetuning** - same as training (unless LORA-style)

Key Difference

Training requires 3-4x more matmuls than inference (forward + backward + recompute)

High-End Accelerator Reality (Q1 2026)

NVIDIA: B200s/B300s/GB200 emerging, H100/H200 widely available

AMD: MI325X widely available (Tier-2 clouds), MI355X emerging

Intel: Gaudi2/Gaudi3 on Intel cloud

Amazon: Trainium2 on AWS

Google: TPUs (cloud only, vendor lock-in)

On-premises: Cerebras WSE, SambaNova DataScale

Physical Reality: 8xH100 Node

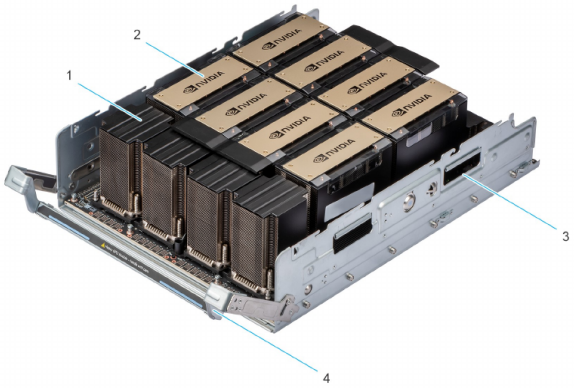

Figure 8. GPU H100/H800 Inside view

1. NVSwitch Heatsink
2. GPU Heatsink
3. Chassis
4. Handle

Figure 4: *

The Most Important Thing

Not enough to buy the fastest accelerators!

High ROI requires: (1) Fast training time, (2) Low total $$

Critical components for ROI:

1. Network (most critical!)
2. Storage
3. CPU & CPU memory (least critical)

생활 지혜

If other components don't match the workload, expensive accelerators will idle!

NVIDIA A100 Spec Reference

	A100 80GB PCIe	A100 80GB SXM
FP64	9.7 TFLOPS	
FP64 Tensor Core	19.5 TFLOPS	
FP32	19.5 TFLOPS	
Tensor Float 32 (TF32)	156 TFLOPS | 312 TFLOPS*	
BFLOAT16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
FP16 Tensor Core	312 TFLOPS | 624 TFLOPS*	
INT8 Tensor Core	624 TOPS | 1248 TOPS*	
GPU Memory	80GB HBM2e	80GB HBM2e
GPU Memory Bandwidth	1,935 GB/s	2,039 GB/s
Max Thermal Design Power (TDP)	300W	400W ***
Multi-Instance GPU	Up to 7 MIGs @ 10GB	Up to 7 MIGs @ 10GB
Form Factor	PCIe Dual-slot air-cooled or single-slot liquid-cooled	SXM
Interconnect	NVIDIA® NVLink® Bridge for 2 GPUs: 600 GB/s ** PCIe Gen4: 64 GB/s	NVLink: 600 GB/s PCIe Gen4: 64 GB/s
Server Options	Partner and NVIDIA-Certified Systems™ with 1-8 GPUs	NVIDIA HGX™ A100-Partner and NVIDIA-Certified Systems with 4,8, or 16 GPUs NVIDIA DGX™ A100 with 8 GPUs

Figure 5: *

TFLOPS: Key Performance Metric

- Most ML work = matrix multiplication (multiply + sum)
- **TFLOPS** = Trillions of Floating Point Operations Per Second
- Higher = better (but beware of marketing!)

Different precisions, different TFLOPS:

- FP32 ¡ TF32 ¡ BF16/FP16 ¡ FP8 ¡ INT8
- BF16 is 2x faster than TF32
- FP8 is 2x faster than BF16

생활 지혜

TFLOPS also depend on matrix size! See next slide.

Matrix Size Matters!

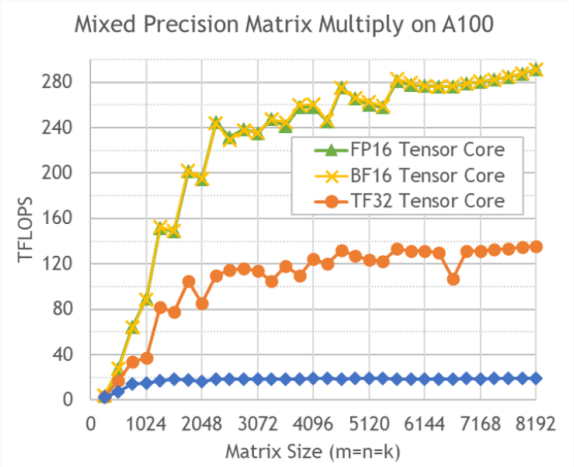

Figure 6. Mixed-precision matrix multiply on A100 with cuBLAS.

Figure 6: *

Theoretical TFLOPS Comparison (BF16)

Accelerator	BF16 TFLOPS
NVIDIA GB300/GB200	2500
AMD MI355X	2300
NVIDIA B300/B200	2250
Intel Gaudi3	1677
AMD MI325X/MI300X	1300
NVIDIA H200/H100	989
Intel Gaudi2	432
NVIDIA A100	312

* w/o sparsity. Most marketing uses w/ sparsity (2x higher) - beware!

Reality: Maximum Achievable TFLOPS (MAMF)

Problem: Theoretical peak TFLOPS are unachievable!

Accelerator	MAMF	Theory	Efficiency
Intel Gaudi2	419	432	96.9%
NVIDIA A100	271	312	86.9%
NVIDIA H100	795	989	80.3%
NVIDIA B200	1745	2250	77.6%
Intel Gaudi3	1243	1677	74.1%
AMD MI325X	785	1300	60.4%

Use `mamf-finder.py` to measure your actual hardware!

Accelerator Efficiency Trend

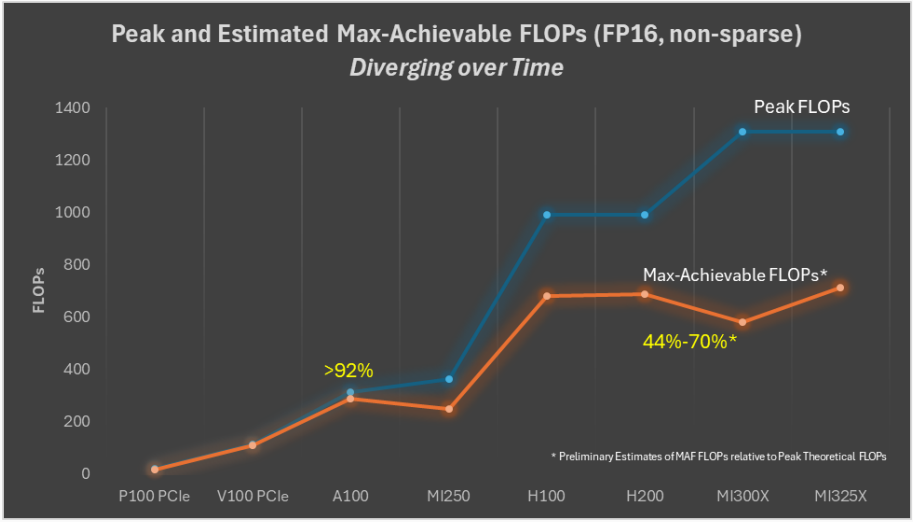

Figure 7: *

Memory: Size and Speed

HBM (High Bandwidth Memory): 3D SDRAM

Accelerator	Memory (GiB)	Bandwidth (TBps)
AMD MI355X	288	8.00
AMD MI325X	256	6.00
AMD MI300X	192	5.30
NVIDIA B200	180	8.00
NVIDIA H200	141	4.80
Intel Gaudi3	128	3.70
NVIDIA H100	80	3.35
NVIDIA A100	80	2.00

More memory = better! Faster bandwidth = less compute idling!

Power Consumption (TDP/TBP)

Higher TDP = more efficient sustained compute!

Accelerator	Power (Watts)
NVIDIA GB300	1400
AMD MI355X	1400
NVIDIA GB200	1200
AMD MI325X	1000
NVIDIA B200	1000
Intel Gaudi3	900
NVIDIA H200/H100	700
AMD MI300X	750

Example: MI325X ¿ MI300X efficiency due to 250W higher TDP

Key Vendors & APIs

NVIDIA	CUDA ecosystem - most mature, widest support
AMD	ROCm - can run most CUDA code as-is! Easy switch from NVIDIA
Intel Gaudi	Habana SynapseAI SDK - PyTorch/TensorFlow support
Google TPU	Cloud-only, vendor lock-in, requires PyTorch XLA
Amazon Trainium	AWS-only, PyTorch XLA

생활 지혜

AMD MI-series with ROCm = easiest alternative to NVIDIA!

Machine Learning IO Needs

3 distinct IO needs for training:

1. **DataLoader feeding** - super fast read, sustained for hours/days
2. **Checkpoint writing** - super fast write, burst mode
3. **Codebase** - medium speed read/write, shared across nodes

생활 지혜

제일 좋고 가장 빠른 걸로 통일한 솔루션은 비싸니까, 2-3개 다른 타입의 파티션으로 나눠서 구성하면 효율적

Storage Reality Check

You're being sold 80% of what you paid!

- Physical disks get full before combined storage gets full
- Rebalancing is costly and infrequent
- At 90% full, programs can fail at any time

Solution:

- Want 100TB reliable? Buy 125TB! (80% of 125TB = 100TB)
- GPFS doesn't have this issue (can use 100%)

IO Concepts That Change Results (and Benchmarks)

Sequential vs Random:

- **Sequential**: checkpoints, codebase load (streaming)
- **Random**: shuffled dataset access, DB-like access, KV-cache offload

Direct vs Buffered IO:

- **Buffered (default)**: OS page cache can make benchmarks look great
- **Direct IO** (`O_DIRECT`): bypass cache to measure real device/FS performance

생활 지혜

IO 벤치마크는 캐시 끄고 돌려야 함. 페이지캐시빨 결과로 의사결정하면 안됨.

Why Network Speed Matters

The Real Bottleneck:

- Training requires syncing gradients between GPUs
- For 2B param model: need to send 16GB of data per iteration!
- Network speed can make or break your training speed

Two types of networking:

- **Intra-node** (within node): Super fast! NVLink, XGMI
- **Inter-node** (between nodes): Used to be 10x slower, now catching up

Key Issue

The whole ensemble communicates at the speed of the slowest link!

Intra-node Network Comparison

All-to-all bandwidth (GBps, unidirectional):

Interconnect	Accelerator	GBps
NVIDIA NVLink 5	B200	900
Intel Gaudi3	Gaudi3	600
NVIDIA NVLink 4	H100	450
AMD XGMI	MI325X	448
NVIDIA NVLink 3	A100	300
Intel Gaudi2	Gaudi2	300
PCIe 5		63

PCIe is 5-10x slower than specialized interconnects!

Inter-node Network Comparison

Total bandwidth per node (GBps, unidirectional):

Interconnect	Config	Total GBps
AWS EFA v4	16x400 Gbps	800
InfiniBand XDR800	8x800 Gbps	800
Intel Gaudi3	24x200 Gbps	600
NVIDIA Quantum-2 IB	8x400 Gbps	400
AWS EFA v3	16x200 Gbps	400
AWS EFA v2	32x100 Gbps	400
Intel Gaudi2	24x100 Gbps	300

Inter-node is catching up to intra-node speeds!

Network Reality: It's Not What You Paid For

Real throughput ≠ Advertised spec

- Best case: 80-90% of advertised speed
- Depends heavily on payload size
- Small payloads = much lower throughput

Example: A100 NVLink (300 GBps advertised)

- 32KB payload: 1.31 GBps (0.4%)
- 536MB payload: 222.72 GBps (74%)
- 17GB payload: 234.89 GBps (78%)

Network Collectives: The Building Blocks

How do GPUs communicate?

Broadcast	1 GPU → All GPUs (copy same data)
Scatter	1 GPU → All GPUs (each gets different chunk)
Gather	All GPUs → 1 GPU (collect all chunks)
All-Gather	All GPUs → All GPUs (everyone gets all data)
Reduce	All GPUs → 1 GPU (sum/avg gradients)
All-Reduce	All GPUs → All GPUs (sum + broadcast)
Reduce-Scatter	All GPUs → All GPUs (reduce + distribute)

TMI

DDP uses **all-reduce** for gradients. ZeRO uses **all-gather** + **reduce-scatter**.

All-Reduce: The Most Important Collective

Used everywhere in distributed training!

What it does:

1. Each GPU sends its gradient data
2. Sum (or avg) all gradients together
3. Result is sent back to every GPU

Part 3. Training

Model Parallelism: Why We Need It

Problem: Models are too big to fit on single GPU!

Three main strategies:

1. **Data Parallelism (DP/DDP)** - replicate model, split data
2. **Tensor Parallelism (TP)** - split model layers horizontally
3. **Pipeline Parallelism (PP)** - split model layers vertically

생활 지혜

Most large model training uses a combination of all three!

Data Parallelism (DP)

How it works:

- Each GPU has a full copy of the model
- Each GPU processes different data
- Sync gradients at the end of each iteration

Pros & Cons:

Pros:

- Simple to implement
- Good GPU utilization
- Minimal overhead

Cons:

- Model must fit on 1 GPU
- Gradient sync overhead
- Memory replicated

생활 지혜

DP is the simplest - use it if your model fits on 1 GPU!

ZeRO: Memory-Efficient Data Parallelism

ZeRO (Zero Redundancy Optimizer) - DeepSpeed's innovation

Three stages of memory savings:

- **ZeRO-1**: Partition optimizer states (4x memory reduction)
- **ZeRO-2**: Partition gradients (8x memory reduction)
- **ZeRO-3**: Partition model weights (linear with #GPUs)

Example: 7B model on 8x 80GB GPUs

- DDP: Can't fit! (needs 126GB per GPU)
- ZeRO-3: Easy fit! (16GB per GPU)

생활 지혜

ZeRO-3 is magic for fitting large models! But adds comms overhead.

Tensor Parallelism (TP)

Split model weights across GPUs horizontally

- Split attention and MLP layers across GPUs
- Each GPU computes part of the layer
- Requires all-reduce after each layer
- **TP degree** = number of GPUs splitting each layer

Pros & Cons:

Pros:

- Fits larger models
- Good for inference too
- Can reduce memory

Cons:

- High comms overhead
- Needs fast intra-node
- Limited by node size

생활 지혜

TP degree usually ≤ 8 due to comms overhead. Needs NVLink speed!

Pipeline Parallelism (PP)

Split model layers across GPUs vertically

- GPU 0: layers 0-7, GPU 1: layers 8-15, etc.
- Process multiple micro-batches in parallel
- Reduces memory per GPU

Pros & Cons:

Pros:

- Low comms (P2P only)
- Scales to many GPUs
- Works inter-node

Cons:

- Pipeline bubble waste
- Complex to tune
- Needs micro-batching

생활 지혜

PP is great for very large models across many nodes!

Combining Parallelism Strategies

Real world: Use all three together!

Example: GPT-3 175B on 1024 GPUs

- TP=8 within each node (NVLink fast)
- PP=16 across nodes (slower inter-node)
- DP=8 data parallel replicas
- Total: $8 \times 16 \times 8 = 1024$ GPUs

생활 지혜

- TP: within node (need fast network)
- PP: across nodes (tolerate slower network)
- DP/ZeRO: for scaling and memory efficiency

Anatomy of Model's Memory: 18 Bytes Rule

Why $\times 18$ in the GPU formula?

Mixed precision (AdamW) training needs **18 bytes per parameter**:

Component	Bytes per param
Model weights (fp32 master copy)	4
Model weights (bf16 working copy)	2
Gradients (fp32)	4
AdamW state: momentum (fp32)	4
AdamW state: variance (fp32)	4
Total	**18**

Example: 7B model $\Rightarrow 7 \times 18 = 126$ GB just for weights+optim+grads!

Activation Memory: The Hidden Cost

Forward activations also consume GPU memory!

- Grows with: batch size $\times$ sequence length $\times$ hidden size
- Per layer: $\sim$20-50 copies of hidden states tensor
- Logits: 4$\times$ bs $\times$ seqlen $\times$ vocab_size (huge!)
- Activations scale with batch size and sequence length

Example: Llama-3.1-8B (bf16, bs=1, seqlen=32K)

- W/o gradient checkpointing: $\sim$240 GB (impossible!)
- W/ gradient checkpointing: $\sim$31 GB (manageable)

생활 지혜

Gradient checkpointing은 거의 항상 켜야 함! Forward를 2번 돌리는 대신 메모리 8x 절약.

Key Training Concepts

Speed vs Memory optimization:

Method	Speed	Memory
Gradient accumulation	✓	✓
Gradient checkpointing	(×)*	✓
Mixed precision (BF16)	✓	–
Flash Attention	✓	✓
DeepSpeed ZeRO	–	✓
Larger batch size	✓	–

* Gradient checkpointing slows down per-iteration, but enables larger BS $\Rightarrow$ net speedup

생활 지혜

Training optimization is all about balancing: speed vs memory vs cost!

Gradient Accumulation

Simulate larger batch size without more memory

How it works:

- Run forward + backward for N micro-batches
- Accumulate gradients without optimizer step
- After N steps, apply accumulated gradients once
- Effective batch size = micro_batch $\times$ N

Code example:

```
accumulation_steps = 4
optimizer.zero_grad()
for step, (input_data, target_data) in enumerate(dataloader):
    outputs = model(input_data)
    loss = criterion(outputs, target_data)
    loss = loss / accumulation_steps  # Scale loss
    loss.backward()                   # Accumulate gradients

    if (step + 1) % accumulation_steps == 0:
        optimizer.step()              # Update weights
        optimizer.zero_grad()         # Reset gradients
```

Gradient Checkpointing

Trade compute for memory (recomputation)

How it works:

- Don't save all activations during forward pass
- Only save checkpoints at certain layers
- During backward, recompute activations on-the-fly
- Saves 8-10x memory at cost of 20-30% slower iteration

When to use:

- Model doesn't fit in memory (OOM errors)
- Want to use longer sequences or larger batch size
- Compute is cheap, memory is expensive

생활 지혜

거의 항상 켜야 함! 20% slower per iteration이지만, 2-4x larger batch로 net speedup!

Fault Tolerance & Checkpointing

Large training = guaranteed failures!

- Hardware failures: GPU, network, storage
- Software crashes: OOM, NaN, bugs
- The larger the cluster, the more frequent the failures

Checkpointing strategies:

- Save model + optimizer + scheduler + RNG states
- Save to fast local NVME, then async offload to cloud
- Keep last N checkpoints locally for fast resume
- Typical: checkpoint every 10-60 min

생활 지혜

체크포인트 안 하면 하루 돌린 학습을 다 날릴 수 있음! Super fast write가 필요한 이유.

Training Dtypes: Speed vs Stability

Common dtype progression:

- FP32 (stable, slow)
- TF32 (Ampere+): FP32-like workflow, much faster matmul
- BF16 mixed precision: **LLM training default** (more stable than FP16)
- FP8 mixed precision: faster, but needs careful recipes (e.g., TE/AMAX scaling)

A100 theoretical (no sparsity)	TFLOPS
FP32	19.5
TF32	156
BF16/FP16	312
FP8/INT8	624

Part 4. Inference

Inference: Two Stages

Prefill and Decode

1. Prefill Stage:

- Process entire prompt at once (parallel)
- Cache intermediate states (KV cache)
- Fast! Even 1k tokens processed quickly

2. Decode Stage:

- Generate new tokens one at a time (sequential)
- Based on all previous tokens
- Slow! This is the bottleneck

Key Insight

Decoding can't be parallelized - it's inherently sequential!

Key Performance Metrics

User Experience Metrics:

TTFT Time To First Token - how fast user sees response

TPOT Time Per Output Token - generation speed per token

TPS Tokens Per Second - inverse of TPOT (easier to think about)

Human reading speed:

- Subvocal: 250 WPM $\approx$ 6 TPS (TPOT: 0.16s)
- Auditory: 450 WPM $\approx$ 11 TPS (TPOT: 0.089s)
- Visual: 700 WPM $\approx$ 19 TPS (TPOT: 0.057s)

생활 지혜

Target 20 TPS to keep up with fast readers!

KV Cache: The Memory Hog

Why KV cache?

- Don't want to recalculate past tokens each time
- Cache Key and Value for all previous tokens
- Grows with sequence length and batch size

Example: Llama-3.1-8B (bf16)

- 1 token = 0.131 MB
- 1024 tokens (batch=1) = 134 MB
- 1024 tokens (batch=128) = 17.2 GB

생활 지혜

KV cache is memory-bound! Smaller cache = faster generation.

Attention Mechanisms: MHA vs GQA vs MQA

Evolution to save KV cache memory:

MHA	Multi-Head Attention - original, uses most memory
GQA	Grouped-Query Attention - groups share KV (4x less memory)
MQA	Multi-Query Attention - all share 1 KV (32x less memory)
MLA	Multi-Latent Attention - compress KV into latent (DeepSeek v3)

Trade-off:

- MHA: Best quality, most memory
- GQA: Good balance (Llama-3 uses this!)
- MQA: Fastest, least memory, slightly lower quality

Batching Strategies

Static Batching (naive):

- Batch N queries together, process in parallel
- All must wait for longest to finish
- Example: 3 queries (10, 50, 100 tokens) $\Rightarrow$ all wait 100 steps

Continuous Batching (smart):

- Still process batch in parallel each step!
- But dynamically adjust batch: remove done, add new
- Example: Step 10 (query 1 done) $\Rightarrow$ add query 4 to batch
- GPU stays full, no wasted slots waiting

Paged Attention

Inspired by OS virtual memory paging

- Treat GPU memory like OS memory
- Dynamic allocation of KV cache
- Prevents memory fragmentation
- Much better GPU utilization

Benefits:

- Fit more requests in same memory
- Higher throughput
- More efficient batching

생활 지혜

vLLM pioneered this - now standard in inference frameworks!

Speculative Decoding

Use small draft model to speed up large model

How it works:

1. Small model generates N tokens (fast)
2. Large model verifies all N at once (batch)
3. If matches: saved time! If not: retry

When it works best:

- Translation, summarization, document QA
- Tasks with predictable outputs
- Greedy decoding or low temperature

생활 지혜

Can achieve 2-3x speedup for input-grounded tasks!

Decoding Methods

How does the model choose the next token?

Greedy Always pick highest probability token
Fast but repetitive, can loop

Beam Search Track top-K paths simultaneously
Better quality but K× slower and more memory

Top-K Sampling Pick randomly from top K tokens
More diverse, controlled randomness

Top-p (Nucleus) Pick from tokens summing to probability p
Adaptive K – fewer candidates when model is confident

생활 지혜

Top-p + Temperature = 가장 많이 쓰는 조합!

Temperature: Controlling Randomness

scaled_logits $=$ **logits**$/$**temperature**

$$
\begin{aligned}
t = 0 &= \text{Greedy decoding (no randomness)} \\
0 < t < 1 &\ \text{Sharper distribution (more focused, precise)} \\
t = 1 &\ \text{Original distribution (balanced)} \\
t > 1 &\ \text{Flatter distribution (more creative, random)}
\end{aligned}
$$

Use cases:

- Code generation: $t \approx 0.0$–0.2 (precise)
- General chat: $t \approx 0.7$–0.9 (balanced)
- Creative writing: $t \approx 1.0$–1.5 (diverse)

생활 지혜

Temperature는 Top-p sampling에만 실질적 영향! Greedy/Beam/Top-K에는 순서가 안 바뀜.

Guided / Structured Generation

Goal: JSON, SQL, schema-constrained output을 “수정”이 아니라 “처음부터” 맞게 생성

How it works (conceptually):

- 다음 token 후보 중에서 **스키마에 맞는 token subset**만 허용
- 그 subset 안에서 확률이 가장 높은 token을 선택

Trade-offs:

- 장점: invalid JSON 같은 실패를 크게 줄임
- 단점: decode가 느려질 수 있음 (제약이 복잡할수록)

Inference Memory Anatomy

Three components:

1. **Model Weights:**
   - fp32: 4 bytes $\times$ params
   - bf16: 2 bytes $\times$ params
   - int8: 1 byte $\times$ params
   - int4: 0.5 bytes $\times$ params
2. **KV Cache:** grows with batch $\times$ seq length
3. **Activation Memory:** temporary processing memory

Example: Llama-3.1-8B in bf16

- Weights: $2 \times 8\text{B} = 16$ GB
- KV cache (batch=128, 1K tokens): $\sim$17 GB
- Total: $\sim$33+ GB (fits on 1x 80GB GPU)

Online vs Offline Inference

Online (Interactive):

- Real-time user queries
- Chatbots, search, APIs
- Needs: low TTFT, low latency
- Always uses inference server

Offline (Batch):

- Hundreds/thousands of prompts
- Benchmarks, synthetic data
- Needs: high throughput
- Server often not needed

생활 지혜

Online: TTFT와 latency 최적화! Offline: throughput 최적화!
같은 하드웨어에서도 세팅이 완전히 다름.

Popular Inference Frameworks

Top choices as of 2025:

- **vLLM** - most popular, great performance, paged attention
- **SGLang** - fast, good for structured output
- **TensorRT-LLM** - NVIDIA optimized, C++ based, complex
- **TGI** - HuggingFace, easy integration
- **DeepSpeed-FastGen** - from DeepSpeed team

How to choose?

1. Does it support your model & features?
2. Permissive license? (Apache 2.0 recommended)
3. Active community & contributors?
4. Can you modify it if needed? (Python vs C++)
5. Does it support your accelerator? (NVIDIA lock-in?)

Part 5. Containers

Container?

격리되고 독립된 리눅스 환경을 보장하는 프로세스

- Isolated Linux environment (pivot-root, namespace, overlay filesystem, cgroup)
- Required files for application (binaries, libraries, configs, etc.)
- Can run anywhere with container runtime (on-premises/cloud)

Why Containers for ML?

Reproducibility and Portability

- Package code + dependencies + environment together
- "Works on my machine" $\Rightarrow$ works everywhere
- Easy to share and deploy
- Isolate different projects with conflicting dependencies

Why containers matter for ML:

- CUDA, cuDNN, PyTorch versions must match
- System libraries (NCCL, MPI) are critical
- Moving between dev/cloud/production environments
- Multi-node training requires identical environments

NVIDIA Container Toolkit

GPU access in containers

- Containers need special setup to access GPUs
- NVIDIA Container Toolkit provides GPU support
- Automatically mounts CUDA drivers and libraries
- Works with Docker and Kubernetes

Running with GPU:

- Docker: `docker run --gpus all ...`
- Automatically detects and uses available GPUs
- No need to manually install CUDA inside container

생활 지혜

항상 NVIDIA 공식 base image 사용! (nvcr.io/nvidia/pytorch:xx.xx-py3)

Docker Overlay Filesystem

Layered storage system for efficient image management

How it works:

- Images are built in layers (each Dockerfile command = 1 layer)
- Layers are stacked on top of each other
- Lower layers are read-only, top layer is writable
- Multiple containers can share the same base layers

Benefits:

- Space efficient: shared layers across images
- Fast builds: only rebuild changed layers
- Fast deployment: pull only new/changed layers

Example

10 containers from same PyTorch image $\Rightarrow$ base image stored once!

Docker Image Layers in Practice

Layer structure example:

1. **Base OS layer** (Ubuntu 22.04) - 80 MB
2. **CUDA layer** (CUDA 12.1) - 3 GB
3. **cuDNN layer** (cuDNN 8.9) - 500 MB
4. **PyTorch layer** (PyTorch 2.1) - 2 GB
5. **Your dependencies** (pip install) - 500 MB
6. **Your code** (COPY train.py) - 10 MB
7. **Container runtime layer** (writable) - changes only

생활 지혜

코드 변경 시 layer 6만 rebuild! 1-5는 캐시 재사용 ⇒ 빠른 빌드!

Container Best Practices

Image size optimization:

- Use official NVIDIA base images (already optimized)
- Multi-stage builds to reduce final size
- Clean up pip cache: `pip install --no-cache-dir`
- Don't include datasets in image (mount as volume)

생활 지혜

Base image 20GB는 정상! CUDA + cuDNN + PyTorch 다 포함이라 큼.

참고문헌 i

- Bekman, Stas. (2023–2026). **Machine Learning Engineering Open Book**. Stasosphere Online Inc. https://github.com/stas00/ml-engineering
- 정환열. (2026). **Container Deep Dive**. IT Medical Center, JADECROSS. coordinatorj@jadecross.com
- PyTorch Documentation. **Reproducibility (Randomness)**. https://pytorch.org/docs/stable/notes/randomness.html
- PyTorch Documentation. **TensorFloat-32 (TF32) on Ampere and later devices**. https://pytorch.org/docs/stable/notes/cuda.html
- Hugging Face Accelerate Documentation. **Accelerator.main_process_first**. https://huggingface.co/docs/accelerate/