FSDP Fine-Tuning of Mistral Medium 3.5 128B

Roman Dolgopolyi

July 17, 2026

We acknowledge EuroHPC JU for awarding the project ID EHPC-DEV-2026D04-003 access to MeluXina on the gpu partition hosted by LuxProvide.

Project overview & Agenda

Central question

What is required for training/fine-tuning a huge LLM (100B+ parameters)?

Project overview

- **Model:** Mistral Medium 3.5 128B (full `bf16` weights)
- **Task:** Instruction tuning on OpenAssistant (OASST1)
- **Method:** LoRA adapters + FSDP sharding
- **Platform:** 3 nodes × 4 NVIDIA A100-40GB = 12 GPUs
- **Stack:** Slurm → HuggingFace Accelerate → PyTorch FSDP → PEFT

Agenda

1. **Why FSDP?** — limits of single-GPU and DDP training
2. **How FSDP works** — sharding, use cases, trade-offs
3. **Our project** — GPU memory budget and Meluxina constraints
4. **Code implementation**
5. **Results** — loss curve, GPU memory and power
6. **Q/A session**

Why Fully Sharded Data Parallel (FSDP)?

Training a 128B model in `bf16` means ~255 GiB of *weights alone* — a quick estimate for `bf16`: GiB ≈ billions of params × 2 (e.g. 128 × 2 ≈ 256 GiB). Even the largest widely deployed datacenter GPU today (NVIDIA H200 with 141 GB of HBM3e) cannot hold the full model on a single device; training also needs activations and gradients beyond the weights. **So we need a way to distribute (shard) weights, activations, and gradients across N GPUs, so that the memory required for training can be allocated.**

What are the trade-offs of FSDP?

FSDP saves memory by sharding, but it adds frequent *heavy* communication between GPUs. That traffic can become a training bottleneck when the network is slow. The impact depends on how FSDP is configured and on where GPUs sit:

- **Intra-node** (GPUs on the same server): high bandwidth, lower latency.
- **Inter-node** (GPUs across servers): lower bandwidth and higher latency.

So, when do we use FSDP?

When the memory required for training does not fit on any single GPU. If a single GPU can handle the workload, training on one device is much more efficient and faster, because there is no communication overhead between devices.

Why FSDP? Other training approaches

What other approaches exist for training?

Single GPU

- **Pros:** simplest setup; no communication overhead; fastest when the model fits.
- **Cons:** limited to one GPU's memory; cannot handle huge models (may change with the next generations of GPUs).

Distributed Data Parallel (DDP)

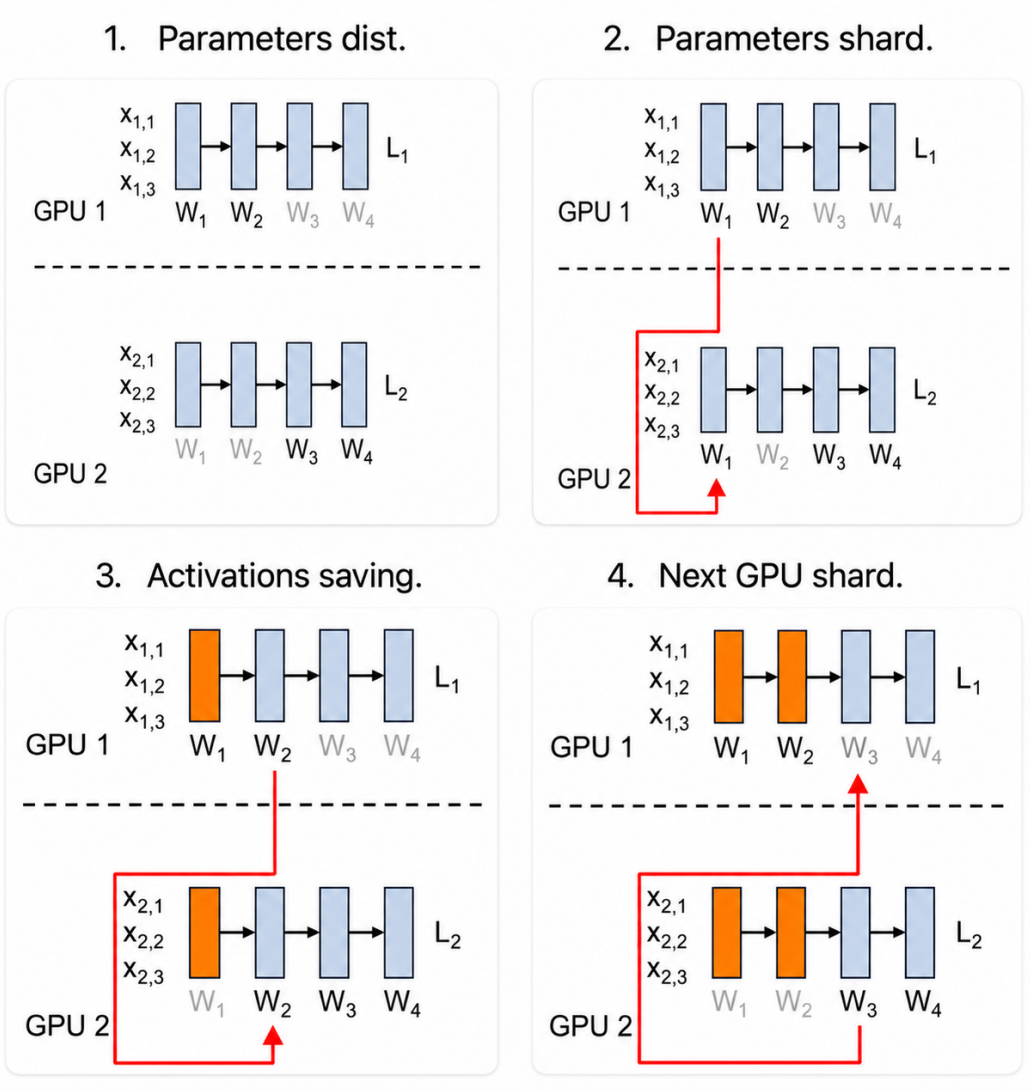
- **Pros:** allows larger batches (more data) and longer sequence lengths during training, which may improve accuracy; straightforward when each GPU holds the full model.
- **Cons:** each GPU must hold a full model copy, which is costly and redundant; does not support huge models (may change with the next generations of GPUs).

"The best choices are the ones that have more pros than cons, not those that don't have any cons at all."

— Ray Dalio

There is no one-size-fits-all setup: iterate between single GPU, DDP, and FSDP depending on your task at hand (model size, available hardware, and batch requirements).

How FSDP works?



Source: Stanford CS231N | Spring 2025 | Lecture 11, Prof. Justin Johnson.

https://www.youtube.com/watch?v=9MvD-XsowsE&t=3495s.

Our Project: task at hand

Fine-tune Mistral Medium 3.5 128B with LoRA on the Meluxina cluster. Find the **optimal number of GPUs** (NVIDIA A100-40GB, 4 per node) so the full `bf16` model plus training overhead fits in memory without OOM.

Memory requirements

- **Weights:** $127.7 \times 2 \approx 255\,\text{GiB}$ cluster-wide ($255/N\,\text{GiB/GPU}$ sharded)
- **LoRA + optimizer:** ~6.7M params (attention only); $6.7 \times 10^6 \times 12 \approx 0.08\,\text{GiB}$
- **Activations:** $88 \times 128 \times 12{,}288 \times 48$ bytes $\approx 6.3\,\text{GiB/GPU}$ (batch 1). Scales with layers, sequence length, hidden size; 48 counts saved forward activations in `bf16` per layer-step
- **FSDP overhead:** full decoder block ($127.7/88 \times 2 \approx 2.9\,\text{GiB}$) + NCCL comm. buffers ($\approx 1\,\text{GiB}$) $\approx 3.9\,\text{GiB/GPU}$

Per-GPU peak: $\mathbf{255/N + 6.3 + 3.9 \leq 40 \Rightarrow N \geq 9}$.

Meluxina cluster limits

Memory needs $N \geq 9$, but Meluxina allocates *full nodes* only (4 A100-40GB GPUs each), so the smallest valid job is **3 nodes = 12 GPUs**. We must take all 12 GPUs on those nodes; that is acceptable and even preferable: peak use is only $\approx 31\,\text{GiB/GPU}$, leaving $\approx 9\,\text{GiB/GPU}$ headroom for unplanned spikes (fragmentation, longer sequences, extra FSDP gathers).

Code implementation

Slurm

Allocates nodes/GPUs; does not train.

```
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=1
#SBATCH --gpus-per-node=4
#SBATCH --partition=gpu
```

$3 \times 4 = 12$ GPUs (one process each)

```
MASTER_ADDR=$(scontrol show \
  hostnames "$SLURM_JOB_NODELIST" \
  | head -n1)
MASTER_PORT=22222
NUM_PROCESSES=$((SLURM_NNODES \
  * SLURM_GPUS_ON_NODE))  # = 12
```

Why `MASTER_ADDR`**?** All 12 processes meet on the first allocated node (rendezvous host).
Why `MASTER_PORT`**?** TCP port for process discovery; must be free and identical on every rank (22222).
Why `NUM_PROCESSES`**?** Accelerate needs the world size: how many GPU workers to start ($3 \times 4 = 12$).
Why pass them to Accelerate? Slurm only reserves hardware; Accelerate launches one process per GPU, wires NCCL, then enables FSDP.

HF Accelerate

Launches one process per GPU; turns on FSDP.

```
python -m accelerate.commands.launch \
  --num_machines "$SLURM_NNODES" \
  --num_processes "$NUM_PROCESSES" \
  --machine_rank "$SLURM_NODEID" \
  --main_process_ip "$MASTER_ADDR" \
  --main_process_port "$MASTER_PORT" \
  --config_file fsdp_config.yml \
  mistral_guanaco_fsdp.py
```

fsdp_config.yml

```
distributed_type: FSDP
fsdp_config:
  fsdp_sharding_strategy: FULL_SHARD
  # weights + grads + optimizer
  fsdp_backward_prefetch: BACKWARD_PRE
  fsdp_cpu_ram_efficient_loading: true
  # rank 0 loads; then broadcast
  fsdp_sync_module_states: true
  # all ranks start identical
```

`-config_file` passes this YAML into Accelerate so it wraps the model with PyTorch FSDP.

PyTorch FSDP

Shards the 128B base across GPUs.

```
ps = PartialState()
process_index = ps.process_index

if process_index == 0:
  model = Mistral3ForConditionalGeneration \
    .from_pretrained(
      MODEL_DIR, dtype=torch.bfloat16)
else:
  with init_empty_weights():
    model = Mistral3ForConditionalGeneration \
      ._from_config(
        config, dtype=torch.bfloat16)
```

Rank 0 loads bf16 on CPU; others use a meta skeleton. FSDP shards onto the 12 GPUs.

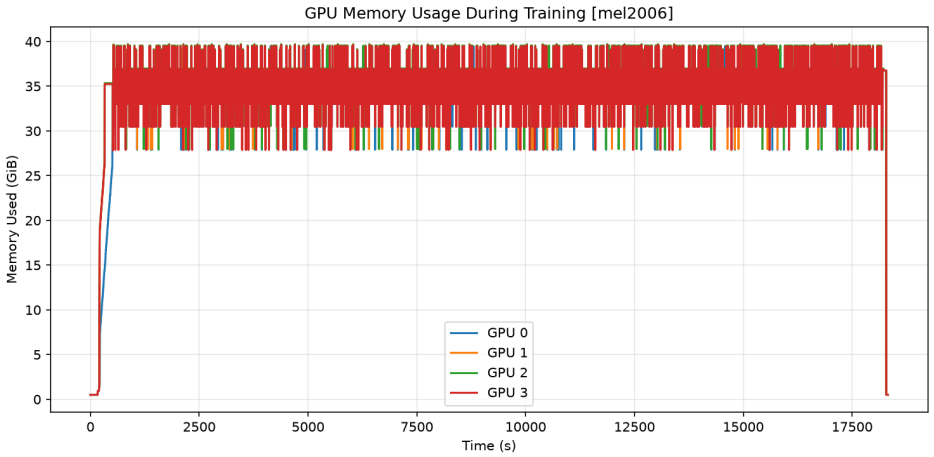
PEFT

Train tiny LoRA adapters, not the base.

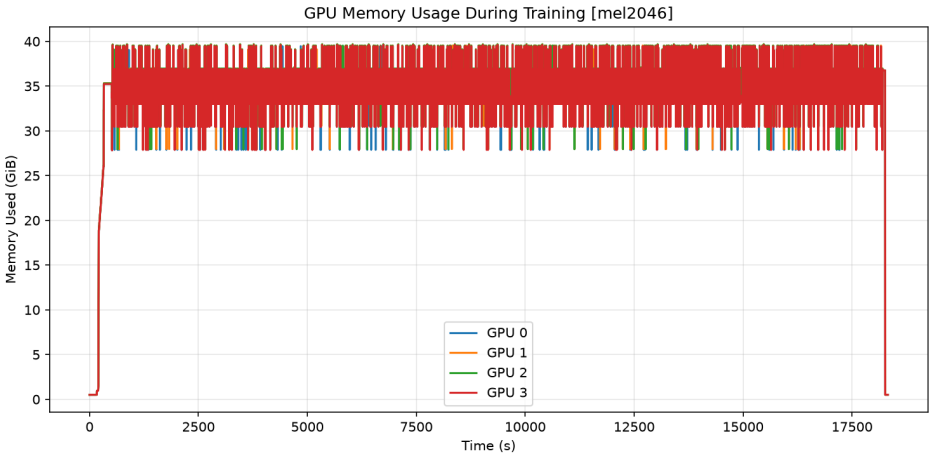
```
peft_config = LoraConfig(
  task_type="CAUSAL_LM",
  r=2,
  lora_alpha=4,
  target_modules=
    r".*language_model.*self_attn\."
    r"(q_proj|v_proj)",
)
```

`task_type="CAUSAL_LM"` — PEFT task tag for next-token language modeling (decoder LLMs). Tells PEFT this is causal LM fine-tuning, not seq2seq/classification.
`r` / `lora_alpha` — `r` is the LoRA rank (adapter size). `lora_alpha` scales the update; effective scale is α/r (here $4/2 = 2$).
`target_modules` — Adapters only on self-attention query/value projections in the language model. Attention is where most instruction-following adaptation happens.

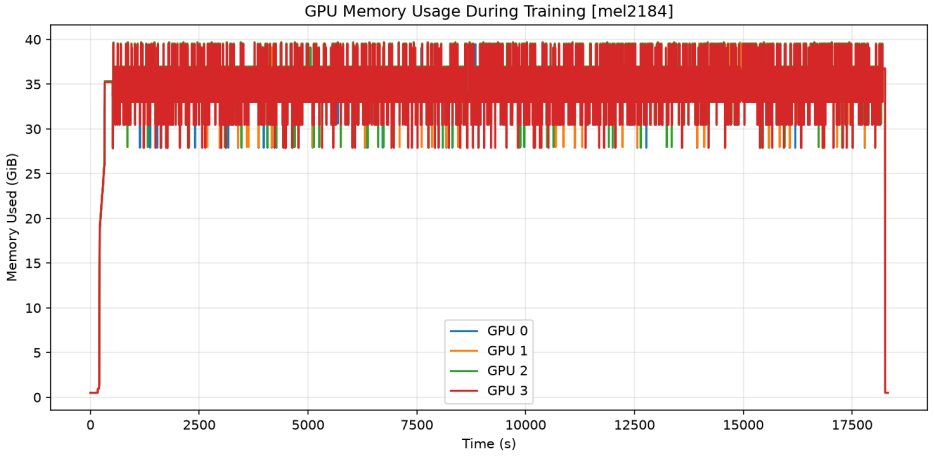
Results



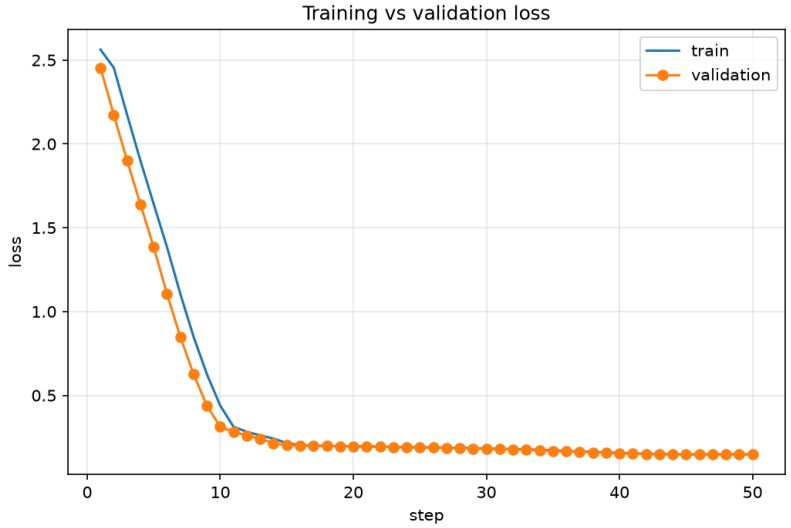
mel2006



mel2046



mel2184



loss curve

Q&A

Thank you!

Any questions?