Fine-Tuning Large Language Models on AMD MI250X

Practical Lessons from LUMI-G Sovereign AI Training

Philipp Horn
kontakt@philipp-horn.dev
Independent Researcher, MoE Sovereign Project

August 2026

Abstract

We report practical lessons from fine-tuning large language models (LLMs) on the LUMI-G supercomputer, equipped with AMD MI250X GPUs running ROCm. Despite LUMI-G being one of Europe's largest AI-capable HPC clusters, comprehensive documentation for the specific combination of ROCm + DeepSpeed + HuggingFace TRL for LoRA-based SFT remains fragmented. This paper fills that gap by documenting two parallel fine-tuning campaigns: a *Judge* model (dense 7B and sparse 35B MoE) and a *Planner* model (dense 8B), spanning more than 20 SLURM job iterations. We derive closed-form memory budget estimates for ZeRO stages 2 and 3 on 64 GiB HBM2e GCDs, characterise the quadratic attention cost of eager (non-Flash) attention on ROCm, explain why a nominal 35B sparse MoE model trains faster and with fewer OOM failures than a dense 8B model, and provide a concrete, reproducible configuration checklist including ROCm-specific environment variables that are absent from general PyTorch documentation. All results and job histories are derived from real EuroHPC allocation runs on project `project_465003058`.

Contents

1 Introduction

LUMI-G is one of Europe's highest-ranked supercomputers and among the largest GPU clusters accessible to academic and research projects through EuroHPC JU. Its compute nodes are equipped with AMD Instinct MI250X accelerators, each physical GPU housing two Graphics Compute Dies (GCDs) with 64 GiB HBM2e each, yielding 128 GiB of on-package memory per physical card and 3.2 TB/s aggregate memory bandwidth [1]. This hardware profile makes LUMI-G theoretically well-suited for fine-tuning large language models (LLMs): the memory capacity alone can accommodate models that exceed the VRAM of consumer hardware by an order of magnitude.

1.1 The Documentation Gap

In practice, the path from theory to a working training run is obstructed by a poorly documented software stack. The dominant LLM training frameworks — HuggingFace TRL, DeepSpeed, PEFT (LoRA), and PyTorch — are primarily developed and tested on NVIDIA hardware with CUDA. Their AMD/ROCm ports exist, are maintained, and generally work, but corner-case behaviour deviates from CUDA in ways that are documented only in scattered GitHub issues and community forum threads:

- The `PYTORCH_HIP_ALLOC_CONF=expandable_segments:True` allocator flag is essential for avoiding memory fragmentation on MI250X but is not mentioned in the official PyTorch memory management documentation for ROCm.
- Flash Attention 2, the standard remedy for the quadratic attention cost at long sequence lengths, is not unconditionally available on ROCm. Its availability depends on the exact ROCm, container, and kernel version.
- GPU discovery during model loading emits a watchdog-timeout warning on MI250X whenever the NVML library version does not match the kernel module version — a situation that arises after driver updates without a reboot and that occurs silently in shared HPC environments.
- DeepSpeed's ZeRO stage selection (2 vs. 3) has different performance and memory implications for dense versus sparse Mixture-of-Experts (MoE) models, which current documentation does not address explicitly.

No single document addresses the full stack from SLURM job submission through container configuration, environment variable tuning, ZeRO stage selection, dataset preparation, and memory budget estimation, specifically for AMD MI250X on LUMI-G.

1.2 Contributions

This paper makes the following practical contributions:

1. **Memory budget formulas** for ZeRO-2 and ZeRO-3 on 64 GiB GCDs, covering dense and sparse MoE architectures (Section 3).
2. **Attention complexity analysis** and a closed-form step-time estimate for eager (non-Flash) attention on ROCm, including the dataset sequence-length analysis required to predict training duration (Section 4).
3. **Empirical training campaign** — a full job history of two parallel fine-tuning campaigns (Judge and Planner models) with 20+ SLURM runs, failure modes, and the interventions that resolved them (Section 5).

4. **ROCm-specific configuration checklist** — environment variables, DeepSpeed JSON settings, and SLURM batch script patterns that are not covered by upstream documentation (Section 6).
5. **Reproducible setup** — the complete software stack, container images, and job scripts are published alongside this paper.

1.3 Context: Sovereign AI Training on EuroHPC

The work reported here was conducted as part of the *MoE Sovereign* project [2], which develops a self-hosted, data-sovereign multi-expert LLM orchestrator. Two LLMs required specialised fine-tuning: a *Judge* model that evaluates orchestrator outputs using a four-class label schema, and a *Planner* model that generates tool-calling sequences for domain routing. Both were trained via LoRA-based SFT (Supervised Fine-Tuning) using HuggingFace TRL's `SFTTrainer` and DeepSpeed, inside a Singularity container provided by the LUMI AI environment. The EuroHPC allocation (`project_465003058`) provides 4 500 node-hours (18 000 GPU-hours) under grant `EHPC-DEV-2026D06`, spanning six months on LUMI-G's `standard-g` and `small-g` partitions.

2 Hardware and Software Stack

2.1 LUMI-G Hardware

Each LUMI-G compute node contains one AMD EPYC 7A53 CPU and four AMD Instinct MI250X modules. Crucially, each physical MI250X module contains *two* Graphics Compute Dies (GCDs) connected by Infinity Fabric. SLURM and ROCm expose GCDs as independent GPU devices, so a job requesting `-gpus-per-node=8` receives eight independent 64 GiB devices, not four. This distinction matters for memory budgeting and parallelism strategy.

Table 1: LUMI-G node specification (per compute node).

Component	Specification	Training relevance
GPU modules	4× AMD Instinct MI250X	8 GCDs per node
GCDs per module	2 (Infinity Fabric coupled)	SLURM reports as separate GPUs
HBM2e per GCD	64 GiB	Per-rank memory budget
HBM2e bandwidth	1.6 TB/s per GCD	Attention memory-bound cost
Inter-GCD bandwidth	200 GB/s (Infinity Fabric)	ZeRO all-reduce within node
Inter-node network	HPE Slingshot 200 Gb/s	Multi-node communication
CPU	AMD EPYC 7A53, 64 cores	Data loading, tokenisation
System memory	512 GiB DDR5	Dataset caching

2.2 Software Stack

Training runs in a Singularity container provided by the LUMI AI environment [3] (`lumi-multitorch-latest.si`
The container bundles a ROCm-compatible PyTorch build together with HuggingFace Transformers, TRL, PEFT, DeepSpeed, and supporting libraries.

Container binding on LUMI

All job scripts bind `/pfs`, `/scratch`, `/projappl`, and `/project` into the container. The HuggingFace cache (`HF_HOME`) must point to a `/scratch` path; the home directory (`/home`) is too small (20 GiB) for model weights.

Table 2: Software stack used in all training runs.

Component	**Version / Details**
ROCm	7.0
PyTorch	2.10.0+rocm7.0
HuggingFace Transformers	4.51.x
TRL	0.17.x (`SFTTrainer`)
PEFT	0.14.x (LoRA)
DeepSpeed	0.16.x
Container runtime	Singularity/Apptainer
Job scheduler	SLURM

```
singularity exec \
  --bind /pfs,/scratch,/projappl,/project \
  --env HF_HOME=/scratch/$PROJECT/$USER/hf_cache \
  $SIF python3 train.py ...
```

2.3 Parallelism Frameworks

Two distinct parallelism strategies were used across the two model campaigns:

Data Parallel (DDP). PyTorch DistributedDataParallel with `torchrun`. Each rank holds a complete copy of model weights and LoRA adapters. Gradients are averaged across ranks via all-reduce after each backward pass. Used for the Judge 7B model, where a full BF16 copy fits in 64 GiB.

DeepSpeed ZeRO. ZeRO-2 shards optimizer states and gradients across ranks, keeping a full parameter copy on every rank. ZeRO-3 additionally shards parameters, reducing peak parameter memory to $1/N$ but introducing all-gather communication during forward and backward passes. Used for the 35B MoE Judge and the 8B Dense Planner.

3 Memory Budget Analysis

Predicting whether a given model and configuration fits into 64 GiB per GCD is the first gate before submitting a job. The following formulas derive static and dynamic memory contributions and explain why a nominally larger sparse model can consume less memory than a smaller dense model.

3.1 Static Memory: Parameters and Optimizer States

For a model with P parameters trained in BF16 with the Adam optimizer:

$$M_{\text{params}}^{\text{BF16}} = P \times 2 \text{ bytes} \tag{1}$$

$$M_{\text{optim}}^{\text{Adam}} = P \times 12 \text{ bytes} \quad \text{(fp32 master + 1st + 2nd moment)} \tag{2}$$

The ZeRO stage determines how these are partitioned across N ranks:

Table 3: Memory per GPU (GCD) for static tensors under different ZeRO stages. P = total parameters, N = number of GCDs.

ZeRO stage	**BF16 params**	**BF16 grads**	**Optimizer states**
DDP (no ZeRO)	$2P$	$2P$	$12P$
ZeRO-1	$2P$	$2P$	$12P/N$
ZeRO-2	$2P$	$2P/N$	$12P/N$
ZeRO-3	$2P/N$	$2P/N$	$12P/N$

For Qwen3-8B (BF16, Adam, $N = 8$, ZeRO-2):

$$M_{\text{static}} = 2 \times 8\text{B} + \frac{2 \times 8\text{B}}{8} + \frac{12 \times 8\text{B}}{8} = 16\,\text{GiB} + 2\,\text{GiB} + 12\,\text{GiB} = 30\,\text{GiB}$$

This leaves approximately $64 - 30 = 34\,\text{GiB}$ for activations, communication buffers, and peak computation overhead.

3.2 Why Sparse MoE Models Are Cheaper Per Forward Pass

Sparse Mixture-of-Experts models like Qwen3.6-35B-A3B store 35 B parameters but activate only $P_{\text{act}} \approx 3\text{B}$ parameters per token. The "A3B" suffix denotes this: 35 B total, 3 B *active* per token.

This has a critical consequence for activation memory and attention cost: the *active* hidden dimension — which governs attention head count, embedding size, and FFN width per token — corresponds to a $\sim 3\,\text{B}$-class dense model. Concretely, Qwen3.6-35B-A3B has $d_{\text{model}} = 2048$, not the 4096 of a standard 8B dense model.

Table 4: Effective model dimensions relevant to activation memory and attention cost. "Active" refers to parameters actually computed per token.

Model	**Total params**	**Active params**	d_{model}	**Attn heads**
Qwen3-8B (Dense)	8 B	8 B	4096	32
Qwen3.6-35B-A3B (MoE)	35 B	3 B	2048	16
Qwen2.5-7B (Dense)	7 B	7 B	3584	28

Under ZeRO-3 with $N = 8$ for the 35B MoE:

$$M_{\text{static}} = \frac{2 \times 35\text{B}}{8} + \frac{2 \times 35\text{B}}{8} + \frac{12 \times 35\text{B}}{8} \approx 8.75\,\text{GiB} + 8.75\,\text{GiB} + 52.5\,\text{GiB} = 70\,\text{GiB}$$

This exceeds 64 GiB if all 35B parameters contribute equally — which is why ZeRO-3 is mandatory for 35B models. However, with $N = 8$: $70/8 \approx 8.75\,\text{GiB}$ per rank for parameters alone, leaving substantial headroom for a $d_{\text{model}} = 2048$ active attention.

The practical summary: **the 35B MoE model has smaller attention matrices per forward pass than the 8B dense model** because the attention computation scales with d^2_{model}, not total parameter count.

3.3 Dynamic Memory: Activations and Gradient Checkpointing

Without gradient checkpointing, every intermediate activation needed for the backward pass must be retained in memory. For a single micro-batch, this includes all layer inputs, attention weights, and FFN intermediates across all L layers — easily exceeding available HBM2e for long sequences.

Gradient checkpointing (GC) discards intermediate activations after each forward layer and recomputes them during the backward pass. This trades compute (roughly doubles the number of forward passes) for memory: peak dynamic memory reduces from $O(L \times T^2)$ to $O(T^2)$, where T is the sequence length.

Gradient checkpointing is not optional for eager attention on LUMI-G

Without GC, the peak activation requirement at $B = 4$, $T = 8192$, $H = 32$ exceeds 1 TiB across 36 layers — far beyond the 64 GiB per GCD. All successful runs in our campaign used GC. Disabling it for throughput experiments caused immediate OOM at step 0 in every case.

The empirical static + peak activation memory observed across our runs was approximately:

$$M_{\text{total}}^{\text{observed}}(\text{Qwen3-8B, ZeRO-2, GC=ON}) \approx 55\text{–}60\,\text{GiB}$$

leaving only 4–9 GiB of headroom — enough for normal steps but insufficient when a single long-sequence batch triggers a peak allocation.

4 Attention Complexity on ROCm

The attention mechanism is the dominant memory and throughput bottleneck when training on long-context datasets without Flash Attention. This section characterises the problem analytically and derives the step-time estimate used to predict whether a training run can complete within a SLURM time limit.

4.1 Eager Attention: The T^2 Problem

In standard (eager) attention, computing the attention weights for a single layer requires materialising the score matrix:

$$A = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) \in \mathbb{R}^{B\times H\times T\times T} \tag{3}$$

PyTorch upcasts this to `float32` for numerical stability before applying softmax, so the peak memory for a single layer's attention computation is:

$$M_{\text{attn}} = B \times H \times T^2 \times 4 \text{ bytes} \tag{4}$$

With gradient checkpointing, this matrix is materialised once per layer during the recomputed forward pass in the backward step, but is immediately discarded after — so the *peak* memory at any moment is for a single layer.

For Qwen3-8B ($H = 32$, $B = 4$, $T = 8192$):

$$M_{\text{attn}} = 4 \times 32 \times 8192^2 \times 4 \text{ B} = 34.4\,\text{GiB}$$

Adding the ≈ 30 GiB static budget: $30 + 34.4 = 64.4\,\text{GiB} > 64\,\text{GiB}$, which explains the observed OOM when individual sequences approach the maximum length. The run *does* succeed in the average case (where $T \approx T_{p50} = 2647$ tokens), because:

$$M_{\text{attn}}^{T=2647} = 4 \times 32 \times 2647^2 \times 4 \text{ B} \approx 3.6\,\text{GiB} \ll 34\,\text{GiB}$$

OOM is triggered by the rare long-sequence batch near $T_{\max} = 8192$.

4.2 Step-Time Estimation

Beyond memory, eager attention also dominates *wall-clock time* on MI250X. The attention computation is memory-bandwidth-bound rather than compute-bound, because the score matrix must be read from and written to HBM2e. A per-step time estimate under gradient checkpointing (GC) is:

$$t_{\text{step}} \approx \underbrace{N_\mu \cdot \frac{2 \cdot B \cdot H \cdot T_{\text{avg}}^2 \cdot 4\ \text{B}}{\text{BW}_{\text{HBM}}}}_{\text{attention (GC recompute} \times 2)} + \underbrace{N_\mu \cdot \frac{C_{\text{FFN}}}{F_{\text{peak}}}}_{\text{FFN (compute-bound)}} + t_{\text{comm}} \tag{5}$$

where N_μ is the number of gradient accumulation micro-steps per effective training step, T_{avg} is the average token length of a micro-batch, $\text{BW}_{\text{HBM}} = 1.6\,\text{TB/s}$ is the HBM2e bandwidth per GCD, and F_{peak} is the peak BF16 FLOP/s throughput.

In our empirical campaign, the measured step time at ($B = 4, T_{\text{avg}} \approx 2647, N_\mu = 4$, GC = ON) was:

$$t_{\text{step}}^{\text{obs}} \approx 148\ \text{s/step}$$

For 6060 total effective training steps, this yields:

$$T_{\text{total}} \approx 6060 \times 148\ \text{s} = 897{,}000\ \text{s} \approx 249\ \text{h}$$

which far exceeds the 38-hour wall-time limit of the `standard-g` partition used in this campaign. (The largest LUMI-G SLURM partition caps at 72 hours, but `standard-g`, which provides guaranteed GPU availability, enforces a 38-hour limit; all jobs in this work used `standard-g`.)

4.3 Flash Attention on ROCm

Flash Attention [4] avoids materialising the full $T \times T$ score matrix by computing attention in tiled blocks, reducing memory complexity to $O(T)$ and dramatically improving throughput. On NVIDIA hardware, `torch.nn.functional.scaled_dot_product_attention` dispatches to a Flash Attention 2 kernel automatically.

On ROCm with this container (`torch 2.10.0+rocm7.0`), the situation is as follows:

- `scaled_dot_product_attention` dispatches to an eager fallback for Qwen3's GQA configuration on the MI250X in this build.
- The HuggingFace Transformers Qwen3 implementation explicitly requests `attn_implementation="eager"` when Flash Attention is not detected, confirmed by the log line: `Loading model ... (BF16, eager attention)`.
- A dedicated `flash_attn` wheel for ROCm exists and may activate the tiled kernel, but requires compilation against the exact ROCm version and is not included in the LUMI container image.

Flash Attention availability is container-dependent

Do not assume Flash Attention is active because the model supports it. Check `model.config._attn_implementation` after loading to confirm. If it reads `"eager"`, the T^2 cost applies and sequence length is the dominant performance driver.

Table 5: Token length percentiles for the Planner training dataset (Qwen3-8B tokeniser).

Percentile	**Tokens**	**Attention peak (GiB),** $B = 4$**,** $H = 32$
p_{50}	2,647	3.6
p_{95}	3,607	6.7
p_{99}	3,628	6.8
p_{100}	≈4,300	9.6
`max_seq_len=8192`	8,192	34.4
`max_seq_len=4096`	4,096	8.6

4.4 Dataset Sequence Length Analysis

Before submitting a training job, profiling the sequence length distribution of the tokenised dataset is essential for predicting both memory behaviour and training duration.

For our Planner dataset (259 829 samples of chat-formatted tool-calling sequences): The key observation: $p_{99} = 3628$ tokens, meaning 99% of all samples fit within 4096 tokens with margin. Setting `max_seq_len=8192` provides headroom for only 1% of samples while quadrupling the worst-case attention peak relative to `max_seq_len=4096`.

Practical recommendation. Set `max_seq_len` to the nearest power of two above p_{99} of your tokenised dataset. For this dataset, `max_seq_len=4096` covers 99% of samples and reduces the worst-case attention peak from 34.4 GiB to 8.6 GiB, providing 25 GiB of margin on a 64 GiB GCD under the static 30 GiB budget.

5 Empirical Training Campaign

This section documents two parallel fine-tuning campaigns: a *Judge* model trained to evaluate orchestrator outputs, and a *Planner* model trained to generate tool-calling sequences. Both campaigns ran on LUMI-G under EuroHPC project `project_465003058` between July and August 2026.

5.1 Judge Model Training

Task and dataset. The Judge model must classify orchestrator response sequences into four labels: True (T), False (F), Borderline (B), and None (N), with a short paraconsistent reasoning chain. The training dataset (`paraconsistent_large.jsonl`) contains approximately 10 000 samples of structured debate-format sequences. Average tokenised length is well below 1 000 tokens per sample.

First iteration: Qwen2.5-7B-Instruct. The initial Judge training used Qwen2.5-7B-Instruct with DDP, LoRA ($r = 16$, $\alpha = 32$), BF16, `max_seq_len=2048`, $B = 4$, and 3 epochs. With 10 000 samples and short sequences, the attention peak per layer at $(B = 4, T = 2048)$ is:

$$M_{\text{attn}}^{(B=4,T=2048)} = 4 \times 28 \times 2048^2 \times 4 \text{ B} \approx 2.4 \text{ GiB}$$

This fits comfortably within 64 GiB alongside the 7B model's static budget. Three identical jobs (#19682379–#19682381) each completed within 4 hours.

Second iteration: Qwen3.6-35B-A3B MoE. A larger Judge was trained on the same dataset using the sparse Qwen3.6-35B-A3B model with DeepSpeed ZeRO-3, DDP-style torchrun, and BF16 (no quantisation). Despite the nominally larger model, the active hidden dimension of $d_{\text{model}} = 2048$ kept attention peaks comparable to the 7B version. ZeRO-3 sharded parameters to 35B $\times$ 2 B/8 $\approx$ 8.75 GiB per GCD for parameters alone — well within budget. After several configuration fixes (Section 6), a 19-hour job completed successfully through checkpoint-704.

Outcome. Both Judge variants trained to convergence. The final deployed model (`sovereign-judge:35b-q4km` GGUF Q4_K_M quantised from the 35B MoE LoRA merge) achieved training loss 0.3032 and token-accuracy 86.72 %.

Two Judge Systems in MoE Sovereign

The Judge described in this section is the *General Paraconsistent Judge* (Qwen3.6-35B-A3B, 10 000-sample dataset, T/F/B/N classification, deployed as `sovereign-judge:35b-q4km`). A separate *Conflict-Arbitration Judge* (base: `granite4.1:30b`, 90 000-sample adversarial debate dataset generated with GPT-4 / Mixtral-8x22B / Qwen2.5-32B) is described in the companion whitepaper [2] (§10.8) and handles only the `resolve_conflicts_node` for safety-critical expert disagreements. Both judges use the same four-valued Belnap-Dunn label set {T, F, B, N}; the adversarial dataset uses I (Inconsistent) and U (Undecided) as implementation labels that map to Belnap's B and N respectively.

5.2 Planner Model Training

Task and dataset. The Planner model must generate structured tool-calling JSON sequences for domain routing. The training dataset contains 259 829 samples of chat-formatted multi-turn sequences with five system prompt variants and 15% negative (rejection) samples ($p_{50} = 2647$, $p_{99} = 3628$ tokens, see Table 5).

Base model: Qwen3-8B Dense. Qwen3-8B was selected for its strong tool-calling capability, Apache 2.0 licence, small inference footprint ($\approx$ 5 GiB at Q4), and confirmed ROCm/TRL/DeepSpeed compatibility. Previous attempts with phi-4:14B failed due to loss collapse from low dataset entropy in the v1 dataset [2].

5.3 Planner Job History and Failure Analysis

Table 6 summarises the full job iteration history for the Planner campaign. Each failed job produced a diagnostic that directly informed the next run.

5.3.1 Phase 1: OOM from Fragmentation (Jobs 20276483–20276792)

The first hint that raw memory pressure was not the sole cause of OOM came from the error message in jobs 20276483 and 20276792:

Listing 1: Fragmentation OOM pattern.

```
torch.OutOfMemoryError: HIP out of memory. Tried to allocate 6.28 GiB.
GPU 7 has total capacity 63.98 GiB of which 2.50 GiB is free.
Of the allocated memory: 40.63 GiB allocated by PyTorch,
and 19.58 GiB reserved by PyTorch but unallocated.
If reserved but unallocated memory is large try setting
PYTORCH_ALLOC_CONF=expandable_segments:True.
```

Table 6: Planner SFT job history on LUMI-G (2026-07-26 to 2026-07-29). GC = gradient checkpointing; ES = `expandable_segments`.

Job ID	Duration	State	B_μ	GC	ES	Failure / Note
20269376	0:20h	FAILED	8	ON	OFF	OOM step 0: allocate 12.69 GiB (only 540 MiB free)
20271148	3:50h	CANCELLED	2	ON	OFF	66 s/step → ETA 109 h; speed insufficient
20276483	0:07h	FAILED	4	ON	OFF	OOM: 19 GiB reserved-but-unallocated (fragmentation)
20276792	0:01h	FAILED	4	ON	OFF	OOM: same fragmentation pattern
20278004	0:05h	FAILED	4	ON	ON	OOM: ES active but insufficient (first ES-enabled run)
20323741	0:00h	FAILED	4	ON	ON	Immediate exit (dataset path error)
20328849	0:03h	FAILED	4	ON	ON	OOM
20329106	5:03h	CANCELLED	4	ON	ON	Investigating resume behaviour
20333367	**19:14h**	**CANCELLED**	**4**	**ON**	**ON**	**Step 450/6060 @ 0.006 steps/s → ETA 249 h**
20391589	0:07h	FAILED	4	**OFF**	ON	OOM step 0: 59.1 GiB allocated + 6.28 GiB request

With 19.58 GiB *reserved but unallocated*, the allocator holds memory it cannot coalesce into the requested 6.28 GiB contiguous block. This is the textbook fragmentation pattern that `expandable_segments` addresses.

5.3.2 Phase 2: `expandable_segments` Activation (Jobs 20278004–20333367)

After enabling `PYTORCH_HIP_ALLOC_CONF=expandable_segments:True`, the fragmentation disappeared. Job 20391589 (post-ES, GC=OFF) shows:

Listing 2: Raw-memory OOM after fragmentation was fixed.

```
torch.OutOfMemoryError: HIP out of memory. Tried to allocate 6.28 GiB.
GPU 7 has total capacity 63.98 GiB of which 3.45 GiB is free.
Of the allocated memory: 59.10 GiB allocated by PyTorch,
and 229.99 MiB reserved by PyTorch but unallocated.
```

The reserved-but-unallocated dropped from 19.58 GiB to 0.23 GiB, confirming ES is active and effective. The OOM is now pure raw-memory pressure: $59.1 + 6.28 = 65.4\,\text{GiB} > 64\,\text{GiB}$ — which is caused by the absence of gradient checkpointing storing all 36 attention matrices.

5.3.3 Phase 3: The Speed Problem (Job 20333367)

Job 20333367, with GC=ON and ES=ON, ran continuously for 19 hours and 14 minutes, reaching step 450 of 6060 before being cancelled. Step timing was measured at multiple checkpoints:

Table 7: Measured step progression in job 20333367.

Step	Wall time	s/step	Loss	ETA
25	16:28 UTC	152	0.9711	14234 min (237 h)
50	17:29 UTC	122	0.4938	14457 min
400	07:54 UTC	148	0.0550	13930 min
425	08:56 UTC	148	0.0521	13886 min
450	09:58 UTC	148	0.0505	13830 min

At a stable 148 s/step, completing 6060 steps requires ≈ 249 hours. No checkpoint was saved (first checkpoint at step 500, job cancelled at step 450), so no forward progress was retained.

Loss trajectory. The loss curve ($0.97 \rightarrow 0.49 \rightarrow 0.055$) shows healthy learning across the observed range, confirming that the model is training correctly. Speed, not convergence, is the blocking issue.

5.4 Root-Cause Summary

The 249-hour ETA is explained by two compounding factors:

1. **Eager attention at** $T_{\max} = 8192$: the attention recompute during GC backward dominates per-step time. Average $T_{p50} = 2647$ already yields $M_{\text{attn}} \approx 3.6\,\text{GiB}$ per layer per micro-step — but this must be computed $L = 36$ times during the backward recompute, for each of $N_\mu = 4$ micro-steps.
2. **No Flash Attention kernel available**: the MI250X eager path is memory-bandwidth-bound for this attention size, not compute-bound. At $\text{BW}_{\text{HBM}} = 1.6\,\text{TB/s}$, reading and writing the recomputed attention matrix accounts for a large fraction of the observed step time.

The corrective action identified was to reduce `max_seq_len` to 4096 — covering 99% of dataset samples — which was analytically expected to reduce the worst-case attention peak by a factor of $(8192/4096)^2 = 4$ and bring the total ETA within the 38-hour SLURM limit. Section 5.5 documents the empirical outcome of this change.

5.5 Planner v4: Corrective Campaign

Configuration changes. The v4 campaign applied the corrective measure and introduced one additional stability change:

- `max_seq_len` reduced from 8192 to 4096 (covers $p_{99} = 3628$ tokens with no sample truncation).
- `save_steps=100` with a fixed `OUTPUT_DIR` enables resume across job boundaries without data-pipeline re-initialisation.
- Jobs chained with `afterany` dependency: six 38-hour SLURM jobs (IDs 20469441–20470181) provide a continuous 228-hour budget.

Measured throughput. Step timing measured at step 10 (elapsed: 21 minutes):

Table 8: Planner v4 step performance at step 10 (LUMI-G, 2026-07-31).

Metric	**Value**	**Note**
`max_seq_len`	4096	(reduced from 8192)
s/step	≈ 125	measured
ETA for 6060 steps	≈ 210 h	within 228 h budget
Effective batch / step	128	$2 \times 8 \times 8$, no packing

Unexpected result: minimal speedup from `max_seq_len` reduction. The analytical prediction was 148 s/step → 37 s/step (factor 4). The measured result was 148 s/step → 125 s/step (factor 1.2).

The discrepancy is explained by the *no-packing* configuration: in `-no-packing` mode, the collator pads every micro-batch to the length of its *longest sample*, not to `max_seq_len`. When the dataset $p_{99} = 3628$ tokens, most micro-batches already operate near 3628 tokens even at `max_seq_len=4096`. Reducing `max_seq_len` from 8192 eliminates padding waste on rare long outliers, but the dominant cost is the median micro-batch length, which is unaffected.

Implication: `max_seq_len` reduction is only effective when combined with packing (`-packing`), which fills each position up to `max_seq_len` regardless of individual sample lengths. Packing was disabled here to avoid cross-sample attention leakage in safety-critical training data.

expandable_segments availability. During the v4 run, all eight ranks emitted the warning:

Listing 3: expandable_segments not supported on this PyTorch/ROCm build.

```
expandable_segments not supported on this platform
```

The `PYTORCH_HIP_ALLOC_CONF=expandable_segments:True` flag is present in the job script but has no effect: the PyTorch/ROCm build on LUMI-G does not expose the `C10_DRIVER_API_SUPPORTED` symbol. Fragmentation protection is therefore *inactive*. No OOM has occurred in v4; the absence is likely due to ZeRO-2's coarser sharding keeping individual tensor allocations smaller than in v3 runs.

5.6 Phase 2: Teacher-Student Distillation

In parallel with the Planner v4 SFT campaign, a second architectural approach was configured: teacher-student distillation rather than direct SFT on the target model.

Rationale. The Planner's primary deployment constraint is inference latency on a single 24 GB GPU (`N04-RTX`). Qwen3-8B requires ≈ 5 GiB at Q4 quantisation; a 4B-class model would require ≈ 2.5 GiB, leaving substantially more headroom for the KV-cache at 262k-token context windows. Teacher-student distillation allows the smaller model to absorb the larger model's output distribution rather than raw human labels alone.

Job 2 — Distillation SFT (Qwen3.5-4B).

- **Teacher:** Qwen/Qwen3.5-35B-A3B (MoE, 3B active parameters)
- **Student:** Qwen/Qwen3.5-4B (dense)
- **Dataset:** `moe_system_knowledge_sft.jsonl`
- **Framework:** TRL `SFTTrainer` + PEFT LoRA + DeepSpeed ZeRO-2, BF16

- **Key hyperparameters:** `max_seq_len=4096`, $B_\mu = 2$, `grad_accum=8`, `lr=2e-4`, LoRA $r = 16/\alpha = 32$, no 4-bit, no packing
- **Container:** Singularity `lumi-multitorch-latest.sif` (avoids LUMI-G system Python 3.6.15 incompatibility)

The effective global batch size is $2 \times 8 \times 8 = 128$. ZeRO-2 shards optimizer states and gradients across 8 GCDs; parameters remain replicated.

Job 3 — DPO Alignment. After the distillation checkpoint converges, a DPO (Direct Preference Optimisation) alignment pass is applied using `moe_rule_based_rl_dpo.jsonl` as the preference dataset. Key differences from the SFT stage:

- Learning rate reduced to 1×10^{-5} (preserves learned knowledge while shifting the preference distribution).
- Base model is the merged LoRA output of Job 2, not a pre-trained checkpoint.
- Same hardware configuration and ZeRO-2 setup as Job 2.

6 ROCm-Specific Configuration

This section documents configuration details that are specific to AMD MI250X / ROCm and are absent from or inadequately covered by upstream documentation.

6.1 HIP Allocator: `expandable_segments`

PyTorch's CUDA memory allocator maintains a pool of memory blocks. When a new allocation request cannot be satisfied by an existing free block (fragmentation), the allocator normally requests a new block from the OS. Under memory pressure, this fails with an OOM error even when the pool contains enough total free memory — just not in a contiguous chunk.

The `expandable_segments` option changes the allocator's strategy: instead of allocating fixed-size blocks, it can *extend* an existing segment using the virtual memory API. This eliminates the contiguous-allocation requirement for most fragmentation scenarios.

Setting the flag on ROCm. The ROCm allocator reads `PYTORCH_HIP_ALLOC_CONF` as its primary configuration source. Set *both* environment variables for safety, as some build configurations read only one:

```
export PYTORCH_HIP_ALLOC_CONF=expandable_segments:True
export PYTORCH_ALLOC_CONF=expandable_segments:True
```

Build-dependent availability

`expandable_segments` in the ROCm HIP allocator requires `PYTORCH_C10_DRIVER_API_SUPPORTED` to be set at compile time. If the flag is not compiled in, setting the environment variable has no effect and emits a `TORCH_WARN_ONCE` warning — it does not crash. Verify that the reserved-but-unallocated figure drops (compare OOM messages before and after) to confirm the flag is active.

Diagnostic: is `expandable_segments` active? Compare the "reserved by PyTorch but unallocated" value in OOM messages:

- **Before**: 19.58 GiB reserved-but-unallocated → flag inactive
- **After**: 0.23 GiB reserved-but-unallocated → flag active

6.2 NVML Library/Kernel Version Mismatch

A common situation in long-running HPC clusters: the NVIDIA (or AMD ROCm) kernel module version is updated during a system maintenance window, but nodes are not rebooted immediately. This creates a mismatch between the userspace library (`librocm-smi.so`) and the kernel module version that causes `rocm-smi` and `nvidia-smi` to fail with a diagnostic like:

```
Failed to initialise NVML: Driver/library version mismatch
```

Impact on training. Inside the Singularity container, the ROCm libraries are container-bundled and match each other. GPU training itself is unaffected. However, *monitoring tools running on the host* that call `nvidia-smi` directly (cron jobs, Prometheus node exporters, dashboard backends) will fail silently.

Workaround for host-side GPU monitoring. The Linux kernel exposes GPU information via `/proc` without requiring NVML:

```
# Read GPU UUID and name from kernel sysfs (no NVML)
for f in /proc/driver/nvidia/gpus/*/information; do
    uuid=$(grep 'GPU UUID' "$f" | awk '{print $NF}')
    name=$(grep 'Model'    "$f" | cut -d: -f2- | xargs)
    echo "$uuid $name"
done
```

For live metrics (utilisation, memory, temperature), `docker exec` into a container whose bundled `nvidia-smi` matches its internal libraries:

```
docker exec ollama nvidia-smi \
  --query-gpu=uuid,utilization.gpu,memory.used \
  --format=csv,noheader,nounits
```

6.3 GPU Discovery Watchdog on MI250X

During model loading in the Ollama inference engine (and in some VLLM configurations), a GPU discovery probe uses an NVML call that times out when the kernel/library version is mismatched. The resulting warning is:

```
WARN: llama-server GPU discovery watchdog timed out
      ... context deadline exceeded
```

Despite this warning, the model *does* load on GPU correctly (the library mismatch only affects the discovery probe, not the actual compute path). The warning can be safely ignored until the next host reboot.

6.4 DeepSpeed ROCm Environment Variables

The following environment variables are required or strongly recommended for DeepSpeed training on LUMI-G. Standard PyTorch and DeepSpeed documentation does not include `ROCR_VISIBLE_DEVICES` or `PYTORCH_HIP_ALLOC_CONF`:

```
# GPU visibility (AMD equivalent of CUDA_VISIBLE_DEVICES)
export ROCR_VISIBLE_DEVICES=0,1,2,3,4,5,6,7
export HIP_VISIBLE_DEVICES=0,1,2,3,4,5,6,7

# NCCL network interface (LUMI Slingshot fabric)
export NCCL_SOCKET_IFNAME=hsn0
```

```
export NCCL_NET_GDR_LEVEL=3

# Distributed training setup
export MASTER_ADDR=$(hostname)
export MASTER_PORT=29500
export OMP_NUM_THREADS=7    # 56 CPU cores / 8 GPUs

# Allocator (see Section 6.1)
export PYTORCH_HIP_ALLOC_CONF=expandable_segments:True
export PYTORCH_ALLOC_CONF=expandable_segments:True
```

6.5 DeepSpeed ZeRO Configuration Files

Two DeepSpeed configuration files were used, one for ZeRO-2 (dense 8B model) and one for ZeRO-3 (sparse 35B MoE model). The key differences are highlighted:

Listing 4: deepspeed_zero2_qwen3.json (key fields).

```
{
  "zero_optimization": {
    "stage": 2,
    "overlap_comm": true,
    "contiguous_gradients": true,
    "reduce_bucket_size": "auto",
    "allgather_bucket_size": "auto"
  },
  "bf16": { "enabled": true },
  "gradient_accumulation_steps": 4,
  "train_micro_batch_size_per_gpu": 4
}
```

Listing 5: deepspeed_zero3.json (key fields).

```
{
  "zero_optimization": {
    "stage": 3,
    "offload_optimizer": { "device": "none" },
    "offload_param":     { "device": "none" },
    "overlap_comm": true,
    "contiguous_gradients": true,
    "sub_group_size": 1e8,
    "reduce_scatter": true,
    "stage3_max_live_parameters": 1e8,
    "stage3_max_reuse_distance":  1e8,
    "stage3_gather_16bit_weights_on_model_save": true
  },
  "bf16": { "enabled": true }
}
```

ZeRO-3 + gradient accumulation mismatch

When using `SFTTrainer` with a DeepSpeed ZeRO-3 config and `gradient_accumulation_steps > 1`, a warning appears:

```
[RANK 0] Gradient accumulation steps mismatch:
```

```
GradientAccumulationPlugin has 1, DeepSpeed config has 4.  Using
DeepSpeed's value.
```

This is benign — DeepSpeed's value takes precedence and is correct. Suppress it by removing `gradient_accumulation_steps` from the JSON and relying solely on the Trainer argument.

6.6 Checkpoint Resume Under ZeRO Parallelism

When resuming from a checkpoint with ZeRO, each rank's optimizer state shard must be loaded. The memory peak during resume *exceeds* the peak during steady-state training because the Trainer temporarily holds both the loaded checkpoint tensors and the initialised model parameters before consolidating them. This transient peak caused OOM in job 20391589 when attempting to resume (without GC) from a checkpoint saved by a different job.

Safe resume pattern.

- Ensure the same `gradient_checkpointing` setting between the checkpoint job and the resume job.
- Verify the output directory contains a valid checkpoint before `trainer.train(resume_from_checkpoint=.` is called.
- Add at least 10% memory headroom (target < 58 GiB steady state on a 64 GiB GCD) to absorb the resume peak.

7 Practical Recommendations

The following checklist distils the lessons from our campaign into concrete, actionable decisions for a practitioner planning to fine-tune an LLM on LUMI-G.

7.1 Pre-Submission Checklist

Step 1 — Analyse your dataset before choosing `max_seq_len`

```python
import json, numpy as np
from transformers import AutoTokenizer

tok = AutoTokenizer.from_pretrained("your/model")
lengths = []
with open("dataset.jsonl") as f:
    for line in f:
        msgs = json.loads(line)["messages"]
        text = tok.apply_chat_template(msgs, tokenize=False)
        lengths.append(len(tok.encode(text)))

a = np.array(lengths)
print(f"p50={np.percentile(a,50):.0f}  "
      f"p95={np.percentile(a,95):.0f}  "
      f"p99={np.percentile(a,99):.0f}  "
      f"max={a.max()}")
```

Set `max_seq_len` to the next power of two above your p_{99}. Never set it to an arbitrary round number like 8192 if your actual data is concentrated below 4096.
Packing caveat: with `-no-packing`, each micro-batch is padded to its longest sample, not to `max_seq_len`. In this configuration, reducing `max_seq_len` eliminates waste only for rare

long-tail outliers; the dominant cost is the median micro-batch length. Expect a factor-4 speedup only when `-packing` is enabled.

Step 2 — Compute your memory budget

Apply the formulas from Section 3:

$$M_{\text{static}} = 2P + \frac{2P}{N} + \frac{12P}{N} \quad \text{(ZeRO-2)}$$
$$M_{\text{attn,peak}} = B \times H \times T_{p99}^2 \times 4\ \text{B} \quad \text{(one layer, GC=ON)}$$

If $M_{\text{static}} + M_{\text{attn,peak}} > 64\,\text{GiB}$: either reduce B, reduce `max_seq_len`, or switch to ZeRO-3.

Step 3 — Estimate training duration before submitting

$$T_{\text{total}} \approx N_{\text{steps}} \times t_{\text{step}}^{\text{ref}} \times \left(\frac{T_{\text{avg}}}{T_{\text{avg}}^{\text{ref}}}\right)^2$$

Use a reference point from a short (1–2%) pilot run. Report the ETA after step 25 and cancel immediately if it exceeds your time limit. **Do not submit a 38-hour job based solely on estimated throughput** — measure it.

7.2 Configuration Reference

ZeRO stage selection

Scenario	ZeRO stage	Reason
Dense model $\leq$ 13B, 8 GCDs, 64 GiB	ZeRO-2	Params fit per-rank
Dense model $>$ 13B, 8 GCDs, 64 GiB	ZeRO-3	Params sharding needed
Sparse MoE, $P_{\text{total}} > 30$B	ZeRO-3	Total params too large
Sparse MoE, $P_{\text{total}} \leq 30$B	ZeRO-2	Active params small

Required environment variables (LUMI-G SLURM script)

```
# Must-have for ROCm training on LUMI-G
export ROCR_VISIBLE_DEVICES=0,1,2,3,4,5,6,7
export HIP_VISIBLE_DEVICES=0,1,2,3,4,5,6,7
export NCCL_SOCKET_IFNAME=hsn0
export NCCL_NET_GDR_LEVEL=3
export MASTER_ADDR=$(hostname)
export MASTER_PORT=29500
export OMP_NUM_THREADS=7

# Critical for fragmentation avoidance
export PYTORCH_HIP_ALLOC_CONF=expandable_segments:True
export PYTORCH_ALLOC_CONF=expandable_segments:True

# Suppress tokeniser parallelism warning
export TOKENIZERS_PARALLELISM=false

# HuggingFace cache must be on /scratch (home too small)
export HF_HOME=/scratch/$PROJECT/$USER/hf_cache
```

LoRA hyperparameters for instruction-following SFT

The following values are a proven starting point for both Judge and Planner tasks. They are conservative by design — reduce r first if overfitting is suspected; increase LR only if loss stalls above 0.5 after 100 steps.

```
lora_config = LoraConfig(
    r=16, lora_alpha=32,
    lora_dropout=0.05,
    target_modules=["q_proj","k_proj","v_proj","o_proj",
                    "gate_proj","up_proj","down_proj"],
    task_type="CAUSAL_LM",
)
training_args = SFTConfig(
    learning_rate=2e-4,             # standard for LoRA
    num_train_epochs=3,
    per_device_train_batch_size=4, # adjust based on Step 2
    gradient_accumulation_steps=4, # effective_batch = 128
    gradient_checkpointing=True,   # mandatory (Section 3.3)
    bf16=True,
    logging_steps=25,
    save_steps=250,                # checkpoint before first
       possible OOM
)
```

7.3 Decision Tree: OOM on LUMI-G

The following decision flow covers the three OOM failure modes encountered in our campaign:

1. **Check reserved-but-unallocated in the error message.** If > 2 GiB: fragmentation. Enable `expandable_segments` (Section 6).
2. **If reserved-but-unallocated is low (< 1 GiB) and OOM still occurs:** raw memory pressure. Verify gradient checkpointing is ON. If ON: reduce `max_seq_len` by one step ($8192 \rightarrow 4096 \rightarrow 2048$) or reduce B_μ.
3. **If OOM occurs during checkpoint resume:** the resume-load peak is higher than steady-state. Reduce `max_seq_len` or add `per_device_eval_batch_size=1` to reduce eval memory before the resume consolidation.
4. **If Flash Attention is not active and training is slow:** reduce `max_seq_len` to p_{99} of the tokenised dataset. This is the single most impactful change for eager-attention runs: a $2\times$ reduction in `max_seq_len` yields a $4\times$ reduction in worst-case attention cost.

7.4 Handling Multi-Job Runs: SLURM Dependency Chains

LUMI-G enforces a hard wall-time limit of 38 hours per job (`#SBATCH -time=1-14:00:00`). Any training run that requires more time must be split across multiple jobs and resume automatically from the last checkpoint. This section documents the three bugs encountered during our campaign that prevented automatic resume, the fixes applied, and the chain-submission helper that makes the pattern robust.

Three Bugs That Silently Prevented Resume

All three bugs coexist quietly: the job starts, trains, and terminates on timeout without producing an error message. The next job starts from step 0, discarding all previous work.

Bug 1 — Output directory contains the job ID.

```
# Wrong: each job creates a fresh directory
OUTPUT_DIR="${SCRATCH}/run_${SLURM_JOB_ID}"

# Fixed: all jobs in one training round share one directory
TRAIN_RUN="${TRAIN_RUN:-qwen3_planner_v4}"
OUTPUT_DIR="${SCRATCH}/${TRAIN_RUN}"
```

When `OUTPUT_DIR` includes `$SLURM_JOB_ID`, every new job starts in an empty directory. `get_last_checkpoint()` returns `None` and training restarts from scratch. The fix is a fixed run name exported as an environment variable so that all jobs in one training round write to and read from the same path.

Bug 2 — `save_steps` larger than the job's step budget.

```
# Wrong: first checkpoint at step 500
# SAVE_STEPS="${SAVE_STEPS:-500}"

# Fixed: checkpoint every ~100 steps (~3.5 h at 130 s/step)
SAVE_STEPS="${SAVE_STEPS:-100}"
```

With `save_steps=500` and a step rate of 148 s/step, the first checkpoint appears after $500 \times 148 = 20.7$ h. A job cancelled at step 450 (after 18.5 h of work) produces no checkpoint. The fix is to reduce `save_steps` so that at least one checkpoint is written well before the 38-hour limit. A useful heuristic: target a checkpoint every 3–4 hours.

Bug 3 — Gradient checkpointing disabled by a speed experiment. A prior job attempted to speed up training by disabling gradient checkpointing (`gradient_checkpointing=false`). This caused an immediate OOM (Section 3), meaning the job produced neither a checkpoint nor useful work. The fix is to lock the setting in the SLURM script so that speed experiments cannot accidentally remove a configuration that is required for correctness:

```
GRADIENT_CHECKPOINTING="${GRADIENT_CHECKPOINTING:-true}"
```

Automatic Resume with `get_last_checkpoint`

HuggingFace Transformers provides `get_last_checkpoint(output_dir)`, which scans `output_dir` for checkpoint directories and returns the path of the most recent one (by step number), or `None` if none exist. Used correctly, the training script requires no changes between an initial run and a resume:

```
from transformers.trainer_utils import get_last_checkpoint

last_ckpt = get_last_checkpoint(str(output_dir))
if last_ckpt:
    logger.info("Resuming from %s", last_ckpt)
else:
    logger.info("No checkpoint found -- starting from scratch.")

trainer.train(resume_from_checkpoint=last_ckpt)
```

Add a diagnostic log line before training so that the output file confirms the correct checkpoint path for every job in the chain:

```
# In the SLURM script, before srun:
LAST_CKPT=$(ls -d "${OUTPUT_DIR}/checkpoint-"* 2>/dev/null \
            | sort -V | tail -1 || true)
if [ -n "$LAST_CKPT" ]; then
```

```
    STEP=$(basename "$LAST_CKPT" | grep -oP '\d+')
    echo "Resuming from: $LAST_CKPT (step ${STEP})"
else
    echo "No checkpoint -- training starts from scratch."
fi
```

Chain Submission Helper

Submit N jobs in a dependency chain with a single command:

Listing 6: submit_planner_chain.sh

```
#!/bin/bash
# Usage: TRAIN_RUN=my_run ./submit_planner_chain.sh <N>
N="${1:-3}"
SCRIPT="$(dirname "$0")/lumi_sft_qwen3_8b.sh"
TRAIN_RUN="${TRAIN_RUN:-qwen3_planner_v4}"
PREV_JOB_ID=""

for i in $(seq 1 "$N"); do
    if [ -z "$PREV_JOB_ID" ]; then
        OUT=$(TRAIN_RUN="$TRAIN_RUN" sbatch "$SCRIPT")
    else
        OUT=$(TRAIN_RUN="$TRAIN_RUN" sbatch \
                --dependency="afterany:${PREV_JOB_ID}" \
                --kill-on-invalid-dep=yes \
                "$SCRIPT")
    fi
    ID=$(echo "$OUT" | grep -oP '(?<=Submitted batch job )\d+')
    echo "Job $i: $ID"
    PREV_JOB_ID="$ID"
done
```

`afterany` triggers the next job regardless of whether the previous job succeeded, timed out, or failed. `-kill-on-invalid-dep=yes` prevents the chain from stalling if an earlier job is cancelled manually. The chain can be extended at any time by submitting additional jobs with `-dependency=afterany:<last_job_id>`.

Memory peak during checkpoint resume

Loading a ZeRO checkpoint temporarily holds both the checkpoint tensors and the newly initialised model parameters in memory before consolidation. This transient peak is higher than the steady-state training peak. Ensure that the checkpoint job and the resume job use identical `gradient_checkpointing` and `max_seq_len` settings, and target $\leq 58\,\text{GiB}$ steady-state memory per GCD to leave headroom for the resume peak (Section 6).

Dimensioning the Chain

1. Measure the step rate at step 25 (see Section 7).
2. Compute total training time: $T_{\text{total}} = N_{\text{steps}} \times t_{\text{step}}$.
3. Divide by the SLURM wall-time limit to get the minimum number of jobs: $N_{\text{jobs}} = \lceil T_{\text{total}} / 38\,\text{h} \rceil + 1$ (the $+1$ accounts for checkpoint overhead and scheduling delays).
4. Submit the chain immediately using the helper above. Additional jobs can be appended later if the estimate was too low.

Example from our campaign: The Planner SFT (Qwen3-8B, 6060 steps, 130 s/step) requires $6060 \times 130/3600 \approx 219$ h, so $\lceil 219/38 \rceil + 1 = 7$ jobs. We submitted a chain of 6, observed step timing, and added one more job when needed — a practical workflow for any long training run on LUMI-G.

7.5 Monitoring During Training

Add a step-time log to the training script. The very first reported speed after step 25 is an accurate predictor of the final ETA (the LR warmup affects loss but not throughput):

```
class StepTimerCallback(TrainerCallback):
    def __init__(self):
        self._t0 = None
        self._step0 = 0

    def on_step_end(self, args, state, control, **kw):
        if state.global_step == 1:
            self._t0, self._step0 = time.time(), 1
        if state.global_step % 25 == 0 and self._t0:
            elapsed = time.time() - self._t0
            steps   = state.global_step - self._step0
            rate    = steps / elapsed
            eta_h   = (state.max_steps - state.global_step) / rate /
                3600
            logger.info(
                "Step %d/%d | loss=%.4f | %.4f steps/s | ETA %.0f min",
                state.global_step, state.max_steps,
                state.log_history[-1].get("loss", 0),
                rate, (state.max_steps - state.global_step) / rate / 60,
            )
```

Cancel and reconfigure if the ETA at step 25 exceeds the SLURM time limit. **Do not wait 5 hours to discover that a job will run for 250 hours.**

8 Conclusion

We have documented a complete LLM fine-tuning campaign on AMD MI250X hardware (LUMI-G) from the practitioner's perspective, covering two model types (dense and sparse MoE), two training roles (Judge and Planner), and 20+ SLURM job iterations spanning failure, diagnosis, and partial resolution.

The principal findings are:

1. **The 64 GiB-per-GCD budget is tight for dense models under ZeRO-2 with eager attention.** A dense 8B model consumes ≈ 30 GiB in static tensors alone, leaving only 34 GiB for activations. The float32-upcast attention matrix for a single layer at $B = 4$, $T = 8192$ requires 34.4 GiB — exactly at the limit. Gradient checkpointing is mandatory; it reduces the activation high-water mark to a single layer at a time.

2. **A sparse MoE model with lower d_{model} can be cheaper to train than a dense model with a smaller parameter count.** Qwen3.6-35B-A3B (35B total, 3B active, $d_{\text{model}} = 2048$) has smaller attention matrices per forward pass than Qwen3-8B ($d_{\text{model}} = 4096$, all parameters active). Users should compare *active* model dimensions, not total parameter counts, when estimating training cost.

3. **Memory fragmentation is a distinct failure mode from memory pressure.** The OOM error message exposes the reserved-but-unallocated figure. Values above 2 GiB indicate fragmentation; the fix is `PYTORCH_HIP_ALLOC_CONF=expandable_segments:True`, which is not documented in the standard PyTorch memory management guide for ROCm.

4. **Flash Attention availability on ROCm must be verified, not assumed.** Without it, eager attention at long sequence lengths ($T > 2048$) is the dominant training cost. Setting `max_seq_len` to the next power of two above p_{99} of the tokenised dataset is the highest-impact single configuration change available without replacing the container.

5. **Measure throughput at step 25 and compute the ETA before trusting a job to run to completion.** The step-time signal is stable within the first 25 steps and accurately predicts total training duration. A job with ETA > SLURM time limit should be cancelled and reconfigured immediately.

6. **Reducing `max_seq_len` only helps when combined with packing.** The analytical prediction of a factor-4 speedup from halving `max_seq_len` assumed that the dominant cost scales as $T_{\max}^2$. In practice, with `-no-packing`, each micro-batch is padded to its *longest sample*, not to `max_seq_len`. When $p_{99} = 3628$ tokens, the actual micro-batch length is near 3628 regardless of `max_seq_len`. The measured speedup from $8192 \to 4096$ was only $1.2\times$ (148 s/step $\to$ 125 s/step), not the expected $4\times$.

7. **Verify `expandable_segments` availability at runtime.** The flag `PYTORCH_HIP_ALLOC_CONF=expandable` may silently have no effect if the PyTorch/ROCm build does not expose `C10_DRIVER_API_SUPPORTED`. The LUMI-G build in use (July 2026) emits a per-rank warning and leaves fragmentation protection inactive. Inspect job logs for the string `expandable_segments not supported` before assuming fragmentation protection is active.

Status update (August 2026). The Planner v4 campaign (Section 5.5) ran as a six-job chain (Jobs 20469441–20470181) with a combined 228-hour budget. The Judge model (`sovereign-judge:35b-q4km`) completed in July 2026 and is deployed in production as of 2026-07-19. A second distillation campaign (Section 5.6) targets a smaller 4B student model (Qwen3.5-4B) via teacher-student distillation from Qwen3.5-35B-A3B, followed by DPO alignment. LoRA merging, GGUF quantisation, and on-device validation on `N04-RTX` (24 GiB VRAM) are the remaining steps before deployment.

Reproducibility. All SLURM job scripts, DeepSpeed configuration files, the dataset generation pipeline, and the training scripts are available as part of the MoE Sovereign repository [2]. The Singularity container (`lumi-multitorch-latest.sif`) is available through the LUMI AI environment module system [3].

References

[1] AMD Inc. *AMD Instinct MI250X Accelerator*. Tech. rep. Product datasheet, https://www.amd.com/en/products/accelerators/instinct/mi200/mi250x.html. Advanced Micro Devices, 2022.

[2] P. Horn. *MoE Sovereign: A Self-Hosted Multi-Model Orchestrator with Template-Based Expert Routing for Sovereign AI Infrastructure*. Technical Whitepaper, v2.8, August 2026. 2026.

[3] CSC – IT Center for Science. *LUMI AI Guide*. Online documentation, https://docs.lumi-supercomputer.eu/. 2024.

[4] T. Dao et al. “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2022.