Training workloads profiling

Internal Development Report 02

Report Date: December 23, 2025
Engineer: Dmitrii Nikolaev
Project: General intro into profiling

Executive Summary

This report documents the first week and a hlaf of development activities focused on establishing a training environment using Primus on AMD Instinct MI300X GPUs. The week involved initial setup, environment configuration, and baseline training runs for two large language models: Llama 3.1 8B and Qwen 2.5 7B.

Key Achievements:

- Checked the all-hands presentation for general intro into profiling
- Learned Primus and Nemo documentation and design principles about profiling
- Repeated training for Llama 3.1 8B and Qwen 2.5 7B with profiling for AMD MI300X against Nvidia H100
- Done resource monitoring software (htop, nvidia-smi, rocm-smi, nvitop)
- Done training logs with Tracelense and Perfetto (https://github.com/AMD-AGI/TraceLens)
- Researched performance metrics and future experiment cases for AMD and NVIDIA

AMD Training

Running Primus Container [1]

```
docker run -it \
    --device /dev/dri \
    --device /dev/kfd \
    --device /dev/infiniband \
    --network host --ipc host \
    --group-add video \
    --cap-add SYS_PTRACE \
    --security-opt seccomp=unconfined \
    --privileged \
    -v $HOME:$HOME \
    --shm-size 128G \
    -e HSA_NO_SCRATCH_RECLAIM=1 \
    -e HSA_ENABLE_SDMA=1 \
    -e HF_TOKEN=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 \
    -e TRACELENS_ENABLED=1 \
    -e PROFILE_ITERS="0,50" \
    --name primat \
    rocm/primus:v25.10
```

Pretraining

LLama 3.1 8B

```
EXP=examples/megatron/configs/MI300X/llama3.1_8B-pretrain.yaml \
bash ./examples/run_pretrain.sh \
    --train_iters 50 \
    --fp8 hybrid \
    --micro_batch_size 2 \
    --global_batch_size 256 \
    --profile=True \
    --use_pytorch_profiler=True \
    --profile_ranks="[0,1,2,3,4,5,6,7]"
```

Qwen 2.5 7B

```
EXP=examples/megatron/configs/MI300X/qwen2.5_7B-pretrain.yaml \
bash ./examples/run_pretrain.sh \
    --train_iters 50 \
    --fp8 hybrid \
    --micro_batch_size 4 \
    --global_batch_size 512 \
    --profile=True \
    --use_pytorch_profiler=True \
    --profile_ranks="[0,1,2,3,4,5,6,7]"
```

# Profile Log Structure

Primus introduces structured logging that organizes output by experiment and rank, making it easier to isolate issues and trace behavior in large-scale runs. Distributed training often produces scattered and overwhelming logs across multiple ranks and nodes—making debugging difficult and time-consuming. Primus introduces a structured, hierarchical logging system that organizes logs by experiment, module, rank, and severity level to streamline analysis and issue tracking.

```
{workspace}/
└── {work_group}/{user_name}/{exp_name}/logs/pre_trainer/
    ├── rank-0/
    │   ├── debug.log
    │   ├── error.log
    │   ├── info.log
    │   └── warning.log
    ├── rank-1/
    │   └── ...
    ...
    └── rank-7/
```

Log root follows workspace hierarchy which ensures logs are cleanly separated across teams, users, and experiments.

```
{workspace}/{work_group}/{user_name}/{exp_name}/logs/
```

# NVIDIA Training

Pulling training containder

```
docker pull nvcr.io/nvidia/nemo:25.04
```

Running the container

```
docker run --gpus all -it --rm \
  --name primat \
  --shm-size=64g \
  --ulimit memlock=-1 \
  --ulimit stack=67108864 \
  -v /path/to/your/data:/data \
  -v /path/to/your/checkpoints:/checkpoints \
  nvcr.io/nvidia/nemo:25.04 bash
```

Loggin into Hugging Face account

```
huggingface-cli login
```

LLAMA

Converting llama to nemo format

```
python3 -c "
from nemo.collections import llm
import torch

llm.import_ckpt(
    model=llm.LlamaModel(llm.Llama3Config8B()),
    source='hf://meta-llama/Llama-3.1-8B',
    output_path='/checkpoints/llama3_1_8b.nemo'
)
"
```

Sussesfully converted output

```
✓ Checkpoint imported to /checkpoints/llama3_1_8b.nemo
Imported Checkpoint
├── context/
│   ├── artifacts/
│   │   └── generation_config.json
│   ├── nemo_tokenizer/
│   │   ├── special_tokens_map.json
│   │   ├── tokenizer.json
│   │   └── tokenizer_config.json
│   ├── io.json
│   └── model.yaml
└── weights/
    ├── .metadata
    ├── __0_0.distcp
    ├── __0_1.distcp
    ├── common.pt
```

```
from nemo.collections import llm
import nemo_run as run

def run_pretrain():
    recipe = llm.llama3_8b.pretrain_recipe(  # Try different model
        name="llama3_test_fp8",
        dir="/checkpoints",
        num_nodes=1,
        num_gpus_per_node=8,
    )
```

```



    recipe.data.micro_batch_size = 1
    recipe.data.global_batch_size = 8
    recipe.trainer.max_steps = 10

    recipe.model.config.fp8 = "hybrid"

    # Even more parallelism
    recipe.trainer.strategy.tensor_model_parallel_size = 8  # Split
across all GPUs

    recipe.trainer.enable_checkpointing = False
    recipe.resume = None

    run.run(recipe, direct=True)

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

Execute the training

```
torchrun --nproc-per-node=8 /data/start_train.py
```

QWEN

write convert_qwen.py

from nemo.collections import llm
import torch

llm.import_ckpt(
   model=llm.Qwen2Model(llm.Qwen2Config7B()),
   source='hf://Qwen/Qwen2.5-7B',
   output_path='/checkpoints/qwen2_5_7b.nemo'
)

Execute qwen converter
python3 convert_qwen.py

pretrain_qwen.py

```
from nemo.collections import llm
import nemo_run as run

def run_pretrain():
    # Use the Qwen2 7B recipe factory
    recipe = llm.qwen2_7b.pretrain_recipe(
        name="qwen2_5_7b_pretrain_fp8",
        dir="/checkpoints",
        num_nodes=1,
        num_gpus_per_node=8,
    )
```

Use the Qwen2 7B recipe factory

recipe = llm.qwen2_7b.pretrain_recipe(
  name="qwen2_5_7b_pretrain_fp8",
  dir="/checkpoints",
  num_nodes=1,
  num_gpus_per_node=8,
)

```
# Batch size configuration
recipe.data.micro_batch_size = 2
recipe.data.global_batch_size = 256

# Set training duration
recipe.trainer.max_steps = 50

# FP8 CONFIGURATION
# DO NOT set recipe.trainer.precision - it conflicts with the
MegatronMixedPrecision plugin
# Instead, configure FP8 through the model config only
recipe.model.config.fp8 = "hybrid"
recipe.model.config.fp8_param = True

# Execute
run.run(recipe, direct=True)
```

if **name** == "**main**":
  run_pretrain()

Execute the script

```
torchrun --nproc-per-node=8 pretrain_qwen.py
```

Here's the updated table with Llama 3.1 8B and Qwen 2.5 7B added:

MODEL CAPACITY		
Max Model Size (single GPU) - FP16	70-150B parameters	~40B parameters
Max Model Size (single GPU) - FP8	140-300B parameters	~80B parameters
Example: Llama 3.1 8B (FP16)	Fits on 1 GPU (~16GB)	Fits on 1 GPU (~16GB)
Example: Llama 3.1 8B (FP8)	Fits on 1 GPU (~8GB)	Fits on 1 GPU (~8GB)
Example: Qwen 2.5 7B (FP16)	Fits on 1 GPU (~14GB)	Fits on 1 GPU (~14GB)
Example: Qwen 2.5 7B (FP8)	Fits on 1 GPU (~7GB)	Fits on 1 GPU (~7GB)
Example: OPT-66B (FP16)	Fits on 1 GPU (~145GB)	Requires 2+ GPUs
Example: LLaMA-70B (FP16)	Fits on 1 GPU (~140GB)	Requires 2 GPUs (~80GB each)
Example: LLaMA-70B (FP8)	Fits on 1 GPU (~70GB)	Fits on 1 GPU (~70GB)

Memory Calculation Details:

Model	Parameters	FP16 Memory	FP8 Memory
Llama 3.1 8B	8 billion	~16GB (8B × 2 bytes)	~8GB (8B × 1 byte)
Qwen 2.5 7B	7 billion	~14GB (7B × 2 bytes)	~7GB (7B × 1 byte)
OPT-66B	66 billion	~132GB base + overhead ≈ 145GB	~66GB + overhead ≈ 75GB
LLaMA-70B	70 billion	~140GB (70B × 2 bytes)	~70GB (70B × 1 byte)

AMD Tracelense

Install TraceLens directly from GitHub:

```
pip install git+https://github.com/AMD-AGI/TraceLens.git
```

To apply **AMD TraceLens** to your Llama 3.1 8B pretraining run on MI300X, you need to integrate the TraceLens profiling hooks.

```
pip install orjson pandas tqdm
```

TraceLens is typically used in conjunction with the **PyTorch Profiler**.

```
TraceLens_generate_perf_report_pytorch --profile_json_path
output/amd/root/<filename>.pt.trace.json
```

Conclusion

The MI300X has superior hardware specs but often underperforms H100 in practice due to AMD's less mature software stack (ROCm vs CUDA). As a result, the on-paper specifications do not directly translate to real-world training performance, creating a **divergence between theoretical and observed performance** driven primarily by software stack maturity differences. The key challenge is to design experiments that isolate **hardware capability** from ***software efficiency*** [2, 3]. This requires carefully controlled benchmarking where model architecture, parallelism strategy, numerical format (BF16/FP8), kernel selection, communication patterns, and compiler settings are held constant, while low-level profiling both compute utilization and system-level bottlenecks. Only by combining low-level profiling (kernel time, memory bandwidth, occupancy) with end-to-end metrics (tokens/sec, convergence speed, stability) can one fairly attribute performance gaps to ROCm limitations rather than architectural constraints, and identify which parts of the training stack—kernels, collectives, compiler, or framework integration—are responsible for the divergence between theoretical and observed performance.

The key challenge is to design experiments that can attribute performance gaps to specific bottlenecks—whether compiler optimization quality, library coverage, kernel fusion efficiency, or memory management strategies—enabling targeted improvements rather than accepting aggregate performance deficits as inherent hardware limitations. The gap between AMD and NVIDIA accelerators isn't just about floating-point operations; it’s about how efficiently the software translates code into silicon action. **CUDA (NVIDIA):** A decade-plus of optimization has led to highly tuned libraries (cuDNN, NCCL) that are the "gold standard" for deep learning. **ROCm (AMD):** While rapidly improving and open-source, it often requires more manual tuning or "shimming" to achieve performance parity with native CUDA kernels.

Designing the Benchmark Experiment

Based on the performance results page from the AMD ROCm Hub, the terms **Throughput** and **Tokens/sec/GPU** are used to measure different types of workloads and represent different levels of hardware aggregation:

1. **Throughput**. Throughput generally means **how much work the system (or benchmark setup) completes per unit time**. In LLM inference benchmarks this is usually expressed in terms of **tokens per second** — how many tokens the model processes (reads and/or generates) per second under load. In many benchmarks, throughput refers to the system’s ability to process *a stream of requests continuously with maximum load saturation*. High throughput means the system is efficiently using the GPU and can handle requests with high concurrency. **Tokens/sec/GPU**. This is a **specific measurement of throughput *per GPU***. It’s literally the number of tokens processed by a *single GPU* in one second. When benchmarking multi-GPU systems, dividing total tokens processed by the number of GPUs gives you this per-GPU rate. Because training

often scales across hundreds or thousands of GPUs, engineers use the per-GPU throughput to evaluate the efficiency of the software stack and the hardware's raw power, independent of the total cluster size. If a training table shows **13,763 Tokens/sec/GPU** on a 1-node (8-GPU) system, the total system throughput would be $13,763 \times 8$.

2. **GEMM Performance.** For this reason, **GEMM performance is a good proxy for how well frontier transformers, such as ChatGPT, Llama, Claude, Grok, etc. will train on the hardware**. This is a strange bug as torch.matmul and F.Linear are both wrappers around the hardware vendor GEMM libraries, so they should achieve the same level of performance. F.Linear, in particular, is important, as this is the way most end users in PyTorch launch the GEMM kernels.
3. **Time-to-convergence.** It doesn't just measure raw throughput (tokens per second); it measures how long it takes for a model to reach a specific **Target Quality** (accuracy / log perplexity). **Convergence For** GPT-3 175B**, the metric was** Log Perplexity**.**
4. **Non-causal attention.** Standard causal attention (used in most GPT-style models) has highly optimized, fused kernels in both CUDA and ROCm. Non-causal variants like Sliding Window Attention require custom masks and potentially different kernel paths. MI300X delivered **close to half the training throughput** of H100 on Mistral 7B due to poorly optimized or buggy implementations of these non-causal layers in ROCm/PyTorch. Lack of full Flash Attention support and unoptimized backwards passes. Frontier models increasingly adopt variants like Sliding Window Attention/Grouped Query Attention for efficiency, so this was a notable weakness.
5. **Multi-node and agentic setup.** A huge challange is to make a scaling up the training in a modern setup [5,6].

Sources

1. NVIDIA NeMo Framework User Guide https://docs.nvidia.com/nemo-framework/user-guide/latest/overview.html
2. Dylan Patel, Daniel Nishball, and Reyk Knuhtsen MI300X vs H100 vs H200 Training Benchmark: CUDA Moat Still Alive https://newsletter.semianalysis.com/p/mi300x-vs-h100-vs-h200-benchmark-part-1-training?utm_source=chatgpt.com
3. MI300X vs H100 for AI Inference: Benchmarks, Cost & Best GPU Choice https://www.clarifai.com/blog/mi300x-vs-h100

4. MoE Training Best Practices on AMD GPUs
https://rocm.blogs.amd.com/software-tools-optimization/primus-moe-package/README.html
5. Coding Agents on AMD GPUs: Fast LLM Pipelines for Developers
https://rocm.blogs.amd.com/artificial-intelligence/coding-agent/README.html