Internal Development Report 03

Report Date: January 15, 2026
Engineer: Dmitrii Nikolaev
Project: General intro into profiling

Executive Summary

- **Explored PyTorch Trace Collection in NeMo**
  Investigated how to enable and extract PyTorch-level execution traces within **NVIDIA NeMo**, focusing on operators, kernels, and data pipeline stages relevant for model training profiling.
- **Generated Trace Reports from NeMo Traces, but not Compared with MI300X Results**
  Extracted NeMo-generated PyTorch traces using **TraceLens**, produced full profiling reports, due to power maintainence shutdowns some of the logs were destroyed.
- **Performed Baseline Benchmarking on MI300X vs. H100**
  Ran benchmarking sessions to measure **tokens/sec per GPU** on **Llama 3.1 8B** and **Qwen 2.5 7B**, capturing throughput on both AMD MI300X and NVIDIA H100 using identical training configurations for an unbiased baseline.
- **Conducted Device-Specific Tuning for Optimized Performance**
  Adjusted batch size and related performance-sensitive hyperparameters for each device independently (MI300X and H100). Measured and recorded improved throughput metrics after tuning to determine the optimal performance envelope on each platform.

Nemo pofiling

There are two convenient ways to profile Nemo. First one is now implemented in my benchmark.

1. PyTorch Profiler.

PyTorch Profiler is built on top of **Kineto**, which is Facebook/Meta's open-source profiling library. Kineto provides low-level instrumentation for collecting GPU/CPU traces, CUDA events, and performance metrics. It captures kernel launch times, memory allocations, and operator-level statistics across PyTorch operations. When you enable PyTorch Profiler, Kineto automatically instruments the execution pipeline and generates Chrome trace-compatible JSON output for visualization.

Add the PyTorch Profiler to your trainer configuration:

```
from nemo.collections import llm
import nemo_run as run
from lightning.pytorch.profilers import PyTorchProfiler

def run_pretrain():
    recipe = llm.llama31_8b.pretrain_recipe(
        name="llama3_1_8b_pretrain_fp8",
        dir="/checkpoints",
        num_nodes=1,
        num_gpus_per_node=8,
    )

    # ... your existing configuration ...

    # ADD PROFILER
    recipe.trainer.profiler = PyTorchProfiler(
        dirpath="./profile_logs",
        filename="perf_logs",
        export_to_chrome=True,  # Creates JSON for chrome://tracing
        profile_memory=True,
        with_stack=True,
        with_flops=True,
        # Profile specific steps
        schedule={
            "wait": 1,      # Skip first step
            "warmup": 1,    # Warmup for 1 step
            "active": 3,    # Profile 3 steps
            "repeat": 2     # Repeat cycle 2 times
        }
    )

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

2. NVIDIA Nsight Systems

Run your training with `nsys`:

```
nsys profile \
    --output=llama_profile \
    --trace=cuda,nvtx,osrt,cudnn,cublas \
    --force-overwrite=true \
    --capture-range=cudaProfilerApi \
    python pretrain_llama.py
```

Then add NVTX markers in your code:

```
import torch.cuda.nvtx as nvtx

# In your training script
recipe.trainer.callbacks = recipe.trainer.callbacks or []
# NeMo automatically adds NVTX markers, but you can add custom ones
```

Analyzing Results

- **PyTorch Profiler**: View the generated JSON in `chrome://tracing` or use TensorBoard
- **Nsight Systems**: Open `.nsys-rep` files in NVIDIA Nsight Systems GUI
- **AMD**: The `.pt.trace_perf_report.xlsx` files you already have contain profiling data

Would you like me to modify one of your training scripts to add profiling, or would you like more details on any specific profiling method?

Baseline Banchmarking

Looking at my actual benchmark results, I've aligned configuration mismatches, batch size and sequence length.

```
"global_batch_size": 128,
"sequence_length": 2048,
"num_gpus": 8
```

Later on I've configured training seed and learning rate.

Training Convergence Analysis

During initial benchmarking, I observed a significant disparity in training convergence behavior between platforms. AMD training showed rapid loss reduction (11.9 → 0.02 within 30 steps), while NVIDIA training exhibited minimal progress (11.65 → 10.83 over 100 steps).

My initial hypothesis centered on potential issues with gradient accumulation in parallelized configurations or insufficient hardware utilization on the NVIDIA platform. Through systematic experimentation with various parallelization strategies, I successfully increased NVIDIA GPU memory utilization to 90-95%, maximizing resource usage. However, the loss curve continued to plateau around 11.64—a value consistent with cross-entropy for random predictions.

This investigation revealed a fundamental difference in framework configurations: **NeMo is designed out-of-the-box to work with synthetic random data** (suitable for performance testing), while **Primus is preconfigured for real datasets**. This explains the convergence disparity entirely—the frameworks were training on fundamentally different data distributions.

Future Work

GPU Memory Profiling Analysis
Conduct comprehensive side-by-side memory profiling using both **Primus** and **NeMo** frameworks. This includes collecting detailed memory traces, analyzing allocation patterns, and evaluating memory efficiency across different parallelization strategies to better understand the performance characteristics of each platform.

Real Dataset Integration
Prepare and configure real-world training datasets for NeMo to enable meaningful convergence studies. This will allow for fair comparison of model training quality and convergence behavior between platforms, complementing the existing throughput benchmarks.

Automated Benchmark Infrastructure
Implement fully automated benchmark orchestration with robust logging and checkpoint mechanisms. This system will prevent data loss during unexpected interruptions (e.g., power maintenance) and ensure reproducible, reliable benchmark results across multiple runs.