GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Tracing IO in Large Language Model Pretraining on HPC

Karim Elezabawy

MatrNr: 24175359

Supervisor: Zoya Masih

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2026

Abstract

This report presents an end-to-end Input/Output (IO) characterization of Large Language Model (LLM) pre-training using NVIDIA's Megatron-LM framework on a multi-GPU HPC cluster. We integrate DFTracer, an analysis-friendly data flow tracer, at both the Portable Operating System Interface (POSIX) and application levels into seven Megatron-LM modules to capture fine-grained IO events across data loading, batch delivery, and checkpointing. A parameter sweep over three sequence lengths, three batch sizes, and two file access modes (memory-mapped IO and direct POSIX reads) yields 18 configurations, which are analyzed offline with DFAnalyzer. Results show the two access modes exhibit fundamentally different IO tracers: Memory-Mapped IO (MMap) reduces data loader time and improves per-sample bandwidth but renders the data loading path invisible to POSIX-level tracers, while Direct IO exposes every read operation as an explicit system call at the cost of higher overhead. Batch size emerges as the primary driver of data loading overhead, while checkpoint IO remains configuration-invariant. The study highlights the importance of multi-layer tracing for understanding IO behavior in large-scale language model training and provides a reusable instrumentation methodology for future optimization efforts.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☑ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High-Performance Computing

IO Input/Output

LLM Large Language Model

DL Deep Learning

ML Machine Learning

GPU Graphics Processing Unit

CPU Central Processing Unit

NCCL NVIDIA Collective Communications Library

SLURM Simple Linux Utility for Resource Management

TE Transformer Engine

POSIX Portable Operating System Interface

MMap Memory-Mapped IO

BF16 Brain Floating Point 16

FP8 8-bit Floating Point

RoPE Rotary Position Embedding

GQA Grouped-Query Attention

NHR North German Supercomputing Alliance

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

RAM Random Access Memory

VRAM Video Random Access Memory

SSD Solid-State Drive

CUDA Compute Unified Device Architecture

cuDNN CUDA Deep Neural Network Library

JSONL JSON Lines

1 Introduction

1.1 Motivation

Training LLMs on High-Performance Computing (HPC) systems involves far more than raw Graphics Processing Unit (GPU) throughput. Data continuously moves through storage, caches, data loaders, tokenizers, host memory, device memory, and checkpoint files, and any of these stages can silently bottleneck the pipeline. As models, datasets, and checkpoint volumes grow, IO overhead becomes harder to observe and costlier to ignore: a system can scale well in compute while wasting significant time on data movement that conventional tracers do not expose.

This project addresses the gap by integrating **DFTracer**[1] into the **Megatron-LM**[2] training stack and tracing **LLaMA-3-8B**[3] training end to end. Rather than treating training as a black box, we instrument the major IO phases (dataset access, index construction, tokenizer loading, batch delivery, and checkpoint save/load) so that high-level slowdowns can be traced back to concrete code locations.

The study is non-trivial because it spans multiple software layers that are usually examined in isolation. Effective tracing requires placing instrumentation at the right points so that traces reflect real training behavior rather than incomplete fragments.

1.2 Goal and Scope

The goal is to build an end-to-end view of IO behavior during LLM training by integrating DFTracer into Megatron-LM and tracing LLaMA-3-8B training. The project does not aim to redesign Megatron-LM but rather to identify where tracing should be placed, what those points reveal, and how this information supports later optimization.

1.3 Literature Review

Existing work shows that Machine Learning (ML) IO differs fundamentally from traditional HPC IO and requires dedicated study:

- Lewis et al. survey IO in ML on HPC, argues that ML workloads shift pressure from large sequential writes to diverse, framework-driven access patterns across data preparation, training, and inference [LBB25].
- **DLIO** provides a data-centric benchmark that reproduces representative Deep Learning (DL) IO patterns and exposes pipeline bottlenecks [Dev+21]. Kogiou et al. use DFTracer with configurable storage to show that IO behavior depends on deployment, network, and how compute overlaps with IO [Kog+24].
- **DFTracer** supports low-overhead, analysis-friendly tracing across software layers, including dynamically spawned workers common in AI pipelines [Dev+24]. Chris et al. explores LLM-assisted diagnosis of HPC IO from collected traces [Ege+24].

[1] `https://github.com/llnl/dftracer`
[2] `https://github.com/NVIDIA/Megatron-LM`
[3] `https://github.com/meta-llama/llama3`

However, LLM-specific literature focuses on scaling compute and parallelism rather than end-to-end IO characterization. Megatron-LM enables efficient distributed training but does not analyze IO in detail [Nar+21]. ByteCheckpoint targets checkpoint overhead specifically [Wan+25], covering only one segment of the training IO path. There remains a gap in fine-grained studies that trace IO through a real LLM training framework from dataset ingestion to checkpointing.

1.4 Report Structure

The report is organized as follows. Section 2 introduces the tools and frameworks used in this study, including Megatron-LM, DFTracer, DFAnalyzer, and the Pile dataset. Section 3 describes the experimental methodology: hardware environment, software setup, dataset preprocessing, model configuration, parameter sweep design, and the DFTracer integration strategy. Section 4 presents the IO characterization results across all 18 configurations through six quantitative metrics. Section 5 discusses the implications of these results, comparing access modes, identifying primary IO drivers, and offering practical recommendations. Finally, Section 6 summarizes the key findings and Section 7 outlines directions for future work.

2 Tools and Frameworks

2.1 Megatron-LM

Megatron-LM is NVIDIA's distributed training framework for billion-parameter transformer models. It partitions weight matrices across GPUs via intra-layer model parallelism with minimal changes to standard PyTorch[4], requiring no custom compiler or framework rewrite [Nar+21]. In this project, Megatron-LM's data pipeline (dataset loading, tokenized batch construction, host-to-device transfer, and checkpoint writing) constitutes the IO lifecycle under investigation.

2.2 DFTracer

DFTracer is an IO tracing tool from Lawrence Livermore National Laboratory that captures data flow events from AI workflows on HPC systems [Dev+24]. Unlike tracers such as Darshan [Sny+16] and Recorder [Wan+20] that only intercept POSIX calls from the master process, DFTracer also traces dynamically spawned worker processes common in PyTorch pipelines. It operates at two levels:

- **POSIX level:** transparently intercepts file opens, reads, writes, and metadata operations. However, MMap delegates data movement to the kernel via page faults, which no user-space tracer can intercept.
- **Application level:** Python decorators and context managers annotate code regions with metadata (e.g. training step, epoch). This is essential for capturing data loading activity under MMap, where POSIX tracing sees nothing.

[4] `https://pytorch.org`

The combination of both levels ensures that the resulting traces capture a complete picture of data movement, regardless of whether the underlying access mechanism is visible at the syscall layer. This dual-level approach is what distinguishes DFTracer from traditional HPC IO tracers and makes it suitable for instrumenting modern DL training pipelines.

2.3 DFAnalyzer

DFAnalyzer[5] is the companion offline analysis tool for DFTracer. It reads `.pfw` files which are Chrome Trace–compatible JSONs, aligns events across processes, and aggregates them into DL-oriented metrics using the `analyzer/preset=dlio` Hydra preset [Hyd19]. In this work, traces were merged per job into a single file and analyzed for derived metrics. DFAnalyzer therefore closes the loop between raw events collected at runtime and the high-level quantities (times, operation counts, bandwidth) reported in the results. The exact commands are listed in Appendix F.

2.4 The Pile

The Pile[6] is an 825 GiB English-language corpus by EleutherAI, composed of 22 diverse subsets (PubMed, ArXiv, GitHub, Stack Exchange, and others) designed for LLM training [Gao+21]. The motivation behind its construction was the observation that dataset diversity improves cross-domain generalization in large language models, as opposed to training on a single large source such as raw Common Crawl. Higher-quality subsets are upsampled during training, resulting in an effective dataset size of approximately 1.25 TiB after weighting.

It serves as the training dataset in this study, stored in Megatron-LM's preprocessed binary format on the parallel filesystem. Its large size and multi-source composition create a realistic and demanding IO workload suitable for studying data loading behavior at scale.

3 Experimental Methodology

3.1 Environment Setup

All experiments ran on the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG) HPC cluster using a Miniforge3-managed Python 3.12 conda environment. Key software components:

- **Megatron-LM:** installed via `pip install megatron-core[mlm]` (PyTorch + Compute Unified Device Architecture (CUDA), HuggingFace tokenizers).
- **Transformer Engine (TE)**[7]: compiled from source against CUDA 12.6.2, CUDA Deep Neural Network Library (cuDNN) 9.8.0.87, and GCC 13.2.0 for 8-bit Floating Point (FP8) support on H100 GPUs.

Full installation commands are in Appendix A.

[5] `https://github.com/LLNL/dfanalyzer`
[6] `https://huggingface.co/datasets/monology/pile-uncopyrighted`
[7] `https://github.com/NVIDIA/TransformerEngine`

3.2 Dataset Preprocessing

For this project, we used a subset of the full Pile dataset. The subset (approximately 900,000 documents) was streamed from HuggingFace, tokenized with the LLaMA-3-8B tokenizer, and converted into Megatron-LM's binary format: `.bin` (≈5 GB) and `.idx` (≈7 MB). The tokenization step used Megatron-LM's `tools/preprocess_data.py` utility, which reads raw JSON Lines (JSONL) text and produces the indexed binary files consumed by the training data loader. The preprocessing script is in Appendix B.

3.3 Hardware and Software Environment

Experiments ran on the **Grete GPU cluster** (GWDG / North German Supercomputing Alliance (NHR)). Each job used one node with:

- 4× NVIDIA H100 80 GB Video Random Access Memory (VRAM), Intel Sapphire Rapids Xeon Platinum 8468 Central Processing Unit (CPU), 240 GB Random Access Memory (RAM)
- Dataset on Ceph Solid-State Drive (SSD) parallel filesystem; home directories on VAST NHR distributed filesystem
- InfiniBand HDR interconnect for inter-GPU communication
- PyTorch + CUDA 12.6.2, Megatron-LM with TE (FP8)

A key complexity of the environment was a version conflict between PyTorch's bundled CUDA 13 libraries and the system's CUDA 12.6 modules. This was resolved by setting `LD_PRELOAD` to point directly to `libcudart.so.12` and removing CUDA 13 paths from `LD_LIBRARY_PATH` at job startup.

3.4 Model and Training Configuration

The model is **Meta LLaMA-3-8B**, a decoder-only transformer with 8 billion parameters, configured within Megatron-LM as summarised in Table 1. The full training script is listed in Appendix 5.

Two flags were required by the specific system configuration on this cluster:

- `-attention-backend unfused` — disables the Transformer Engine FlashAttention kernel and falls back to the native PyTorch unfused attention path. The fused kernel raised a CUDA illegal-memory-access error during the first forward pass on this cluster's H100 driver stack.
- `-no-gradient-accumulation-fusion` — disables the fused gradient-accumulation kernel, which produced NaN losses when running alongside the FP8 + Brain Floating Point 16 (BF16) gradient-reduction path.

Parameter sweep. To characterize IO across workload sizes, a 3×3 sweep varied:

- **Sequence length:** 512, 1024, 2048 tokens
- **Global batch size:** 128, 256, 512

Table 1: Model and training configuration.

Parameter	Value
Architecture	
Layers	32
Hidden size	4096
Attention heads	32 (GQA, 8 KV heads)
FFN hidden size	14336
Activation	SwiGLU
Normalization	RMSNorm
Position encoding	RoPE (base 500 000)
Training	
Precision	BF16 + FP8 (Transformer Engine)
Tensor parallelism	2
Optimizer	Distributed AdamW
Grad reduce/gather	Overlapping

These parameters were injected via environment variables, allowing the same training script to be reused across all configurations without modification.

Justification for the parameter sweep. Sequence length and global batch size were chosen as the sweep axes for two reasons. First, they are the parameters practitioners most commonly tune when adapting a pretraining recipe to available hardware: sequence length governs the memory footprint per sample and the size of each binary read, while batch size controls how many samples the data loader must deliver per step and therefore how frequently the storage layer is exercised. Second, the chosen values span the practical range used in LLaMA-scale pretraining on a single node with H100 GPUs, where memory constraints place natural upper bounds on both dimensions.

Access mode comparison. A second experimental axis compared two binary file access strategies:

- **MMap:** the kernel maps the file into the process's virtual address space; pages are loaded on demand through the page fault mechanism
- **Direct IO:** explicit POSIX `read()` calls per sequence

This yields 18 total configurations (9 parameter combinations $\times$ 2 access modes). The access mode was controlled entirely at submission time via the `nommapbinfiles` argument.

3.5 DFTracer Integration

DFTracer was integrated at two levels:

1. **POSIX-level (submit-time):** Environment variables enable DFTracer and direct trace output to a per-job directory. When `DFTRACER_INIT=PRELOAD` is set, `libdftracer_preload.so` is injected via `LD_PRELOAD` to intercept all POSIX calls on the monitored data directory. This was disabled during the parameter sweep to reduce overhead and re-enabled for dedicated IO characterization runs.

2. **Application-level:** DFTracer's Python AI tracing API was integrated into seven Megatron-LM modules. Full instrumentation code is in Appendix E.

Each run produces two distinct trace streams: a POSIX-level trace per job (capturing every `open`, `read`, `write`, and `close` on the data directory) and an application-level trace per GPU rank (capturing data item fetch, preprocessing, and index read events with per-sample metadata). These are correlated offline with DFAnalyzer (Section 2.3) to connect storage-level IO events with training-loop semantics. Analysis commands are in Appendix F.

4 Results

This section presents IO characterization results across all 18 configurations, covering six metrics: POSIX event count, data loader time, IO proportion, data layer breakdown, POSIX bandwidth/transfer size, and per-sample read bandwidth.

4.1 POSIX IO Visibility

Figure 1 shows the total number of POSIX events recorded per experiment under MMap and Direct IO.

- Under **MMap**, the POSIX event count is fixed at approximately **9 846** across all nine configurations. It does not vary with sequence length or batch size. This near-constant count consists primarily of initialization events, metadata lookups, and checkpoint writes rather than data reads, confirming that the actual data access path under MMap is invisible to POSIX-level tracing.
- Under **Direct IO**, the count grows directly with the product of batch size and sequence length:
  - 133,037 at sequence length 512, batch size 128
  - 906,600 at sequence length 2048, batch size 512
  - A factor of approximately **92**× relative to MMap at the largest configuration

  Each sample access under Direct IO involves a cycle of `open()`, `lseek()`, `read()`, `fstat()`, and `close()` calls, which explains the rapid growth in event count as more samples are processed.
- The **write count is constant at ≈5 089** across all 18 experiments in both modes. Writes are checkpoint-driven and are completely independent of training configuration.

This confirms that MMap delegates data access to the OS virtual memory subsystem, producing almost no visible POSIX activity, while Direct IO issues explicit syscalls for every sequence retrieved from the dataset.

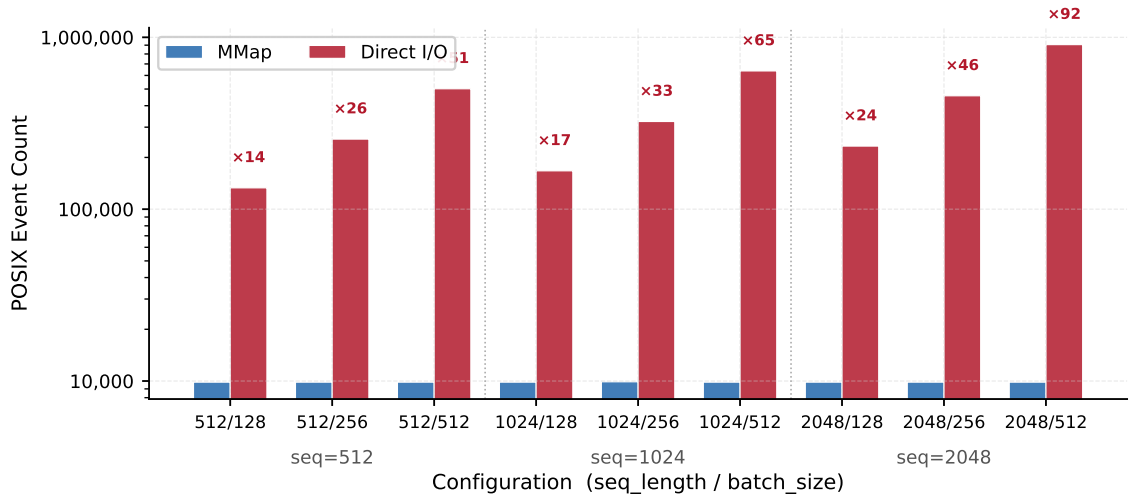

Figure 1: Total POSIX event count per experiment under MMap and Direct IO, shown on a log scale.

4.2 Data Loader Time

Figure 2 presents data loader time for all configurations as a pair of 3×3 heatmaps.

- Under **MMap**, data loader time ranges from **12.8 s** (sequence length 1024, batch size 128) to **43.1 s** (sequence length 2048, batch size 512). The relatively narrow range reflects the fact that MMap access is dominated by CPU-bound index lookups rather than storage latency.
- Under **Direct IO**, data loader time ranges from **26.1 s** (sequence length 512, batch size 128) to **255.5 s** (sequence length 2048, batch size 512). The wide range demonstrates that explicit POSIX read calls scale linearly with the number and size of samples accessed.
- **Batch size** is the dominant factor in both modes:
  - Under MMap, moving from batch size 128 to 512 increases data loader time by a factor of ≈3.
  - Under Direct IO, the same increase raises data loader time by a factor of up to **9**.
- **Sequence length** has a secondary effect that is more visible under Direct IO: at batch size 512, increasing sequence length from 512 to 2048 raises data loader time from 139.9 s to 255.5 s under Direct IO, compared to 39.7 s to 43.1 s under MMap.

4.3 IO Proportion of Job Time

Figure 3 shows data loader time as a percentage of total job time.

- Under **MMap**, the proportion remains **below 1%** across all configurations with no clear dependence on batch size or sequence length. This is consistent with the low absolute data loader times reported above and confirms that under MMap, the data loading path is not a meaningful bottleneck at this scale.

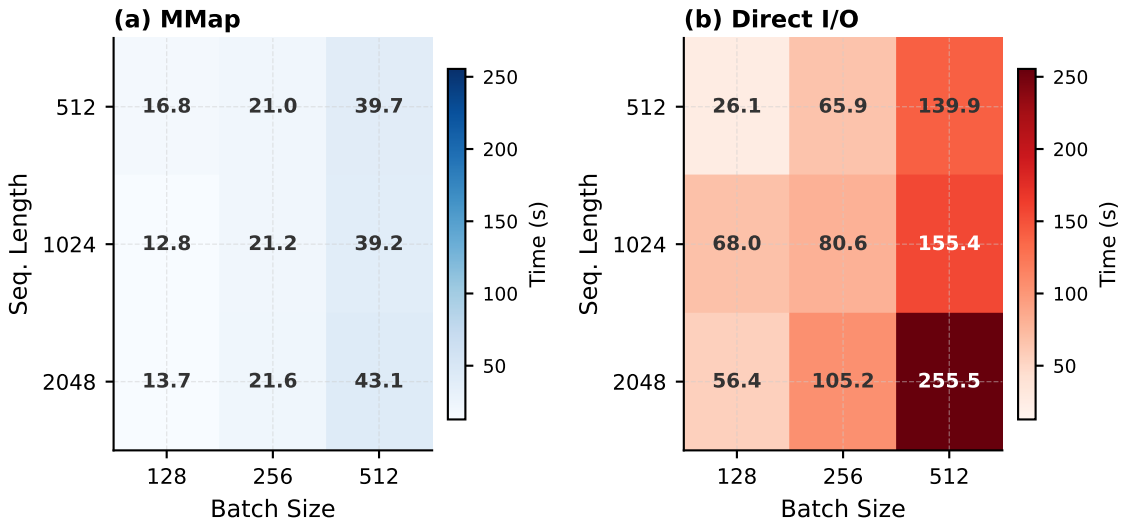

Figure 2: Data loader time (seconds) as a 3×3 heatmap for each access mode.

- Under **Direct IO**, the proportion ranges from **1.5% to 3.9%**. Key observations:
  - At batch size 128, increasing sequence length from 512 to 2048 raises the proportion from 1.5% to 3.1%.
  - At sequence length 512, doubling batch size from 128 to 256 raises the proportion from only 1.5% to 1.9%.
  - The highest value, **3.9%**, occurs at sequence length 1024 and batch size 128.
- Sequence length has a stronger effect on IO proportion than batch size at fixed total token volume, because longer sequences increase per-sample read cost without proportionally increasing compute time.

Although these percentages are modest, they represent consistent, avoidable overhead that accumulates across many training steps. Over the course of a multi-day pretraining campaign, even a 2% overhead translates to hours of wasted compute time.

4.4 Data Layer Operation Breakdown

Figure 4 shows the cumulative time per instrumented data layer operation across all configurations.

- Under **Direct IO**, the dominant component is `preprocess.read_raw`:
  - At batch size 512, sequence length 2048, it reaches **160.3 s**.
  - `preprocess.file_read` reaches **152.1 s** at the same configuration, indicating that nearly all raw read time is spent in direct binary file reads.
  - The close alignment between `read_raw` and `file_read` confirms that under Direct IO, the dominant cost is storage access rather than in-memory processing.

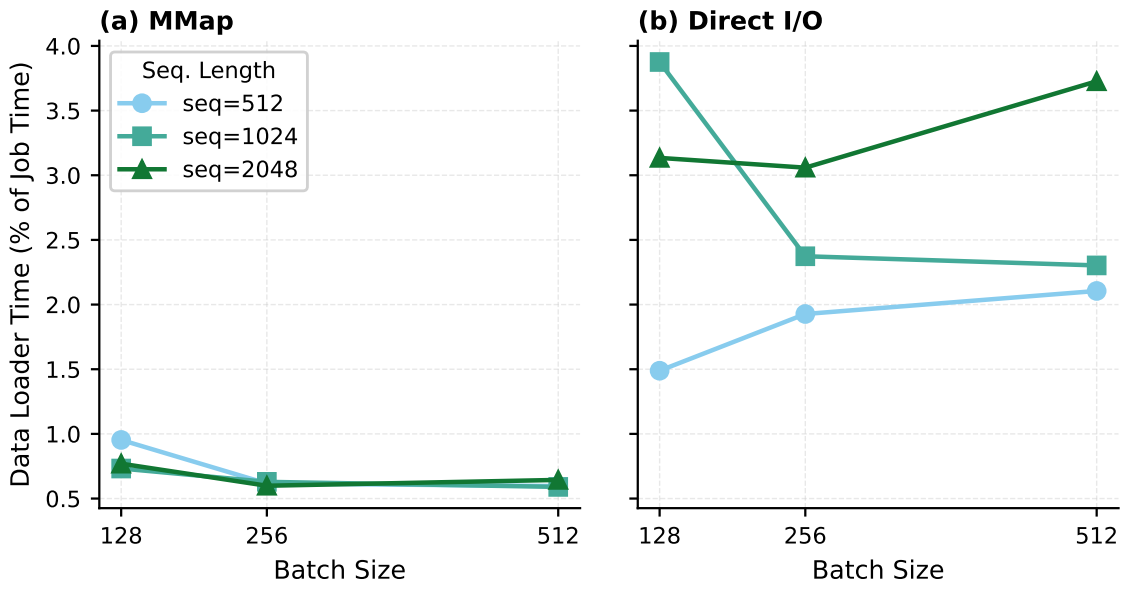

Figure 3: Data loader time as a percentage of total job time.

- Under **MMap**, the same operation reaches only **10.1 s** at the equivalent configuration, as the OS page cache absorbs most accesses. The dominant component under MMap is instead `preprocess.read_indexed`, which performs the CPU-bound index lookup that maps a shuffled sample index to a byte offset and length in the binary file.
- The `preprocess` component (tokenization and mask construction) remains **nearly constant** across all configurations in both modes, confirming it is compute-bound rather than IO-bound.

4.5 POSIX-Level IO Performance

Figure 5 presents three POSIX-level metrics across the parameter grid.

POSIX read time (Figures 5 a, b).

- Under **Direct IO**, read time grows from **10.2 s** to **93.8 s** as batch size and sequence length increase. The growth is approximately linear in the product of the two parameters, consistent with a per-sample read cost model.
- Under **MMap**, read time remains **below 0.025 s** in all cases, a difference of up to three orders of magnitude. The small residual corresponds to the few explicit reads Megatron-LM performs for index and metadata files, not for training data.

POSIX aggregate bandwidth (Figures 5 c, d).

- Under **MMap**: **4,581 to 5,515 MB/s** across all configurations, stable and high. The reported bandwidth under MMap reflects only checkpoint writes and metadata reads, since data reads are invisible to POSIX.

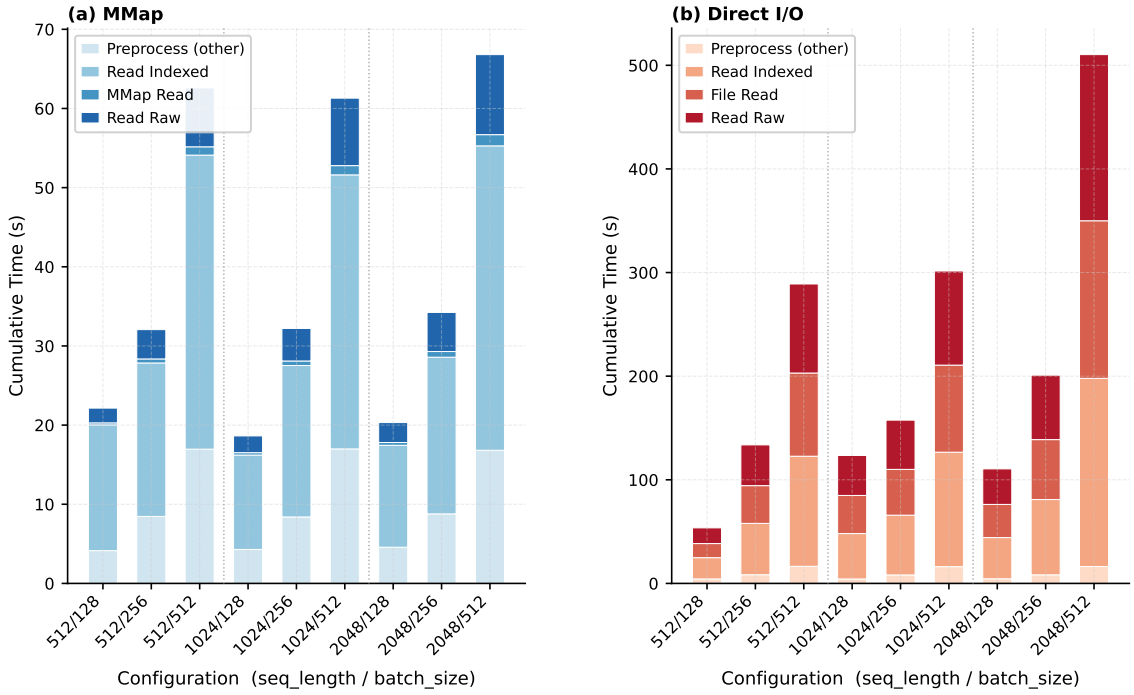

Figure 4: Cumulative time per data layer operation across all nine configurations.

- Under **Direct IO**: **1,670 to 4,677 MB/s**, degrading as batch size and sequence length increase, due to the overhead of many small, per-sequence read calls competing with the Ceph parallel filesystem's access patterns.

POSIX average transfer size (Figures 5 e, f).

- Under **MMap**: approximately **12 MB** throughout, corresponding to large sequential checkpoint writes (the only significant POSIX operations).
- Under **Direct IO**: falls from **0.89 MB** at sequence length 512, batch size 128 to **0.13 MB** at sequence length 2048, batch size 512, reflecting an increasing number of small per-sequence reads that reduce the average transfer size.

4.6 Per-Sample Read Bandwidth

Figure 6 shows the mean per-sample read bandwidth recorded by the application-level metric, annotated inside `__getitem__` (refer to Listing 6).

- Under **MMap**, bandwidth rises monotonically with sequence length:
  - ≈**29.6 MB/s** at sequence length 512
  - ≈**111.4 MB/s** at sequence length 2048

  This increase is expected because longer sequences produce larger contiguous reads that benefit from the OS page cache's sequential prefetching behavior.

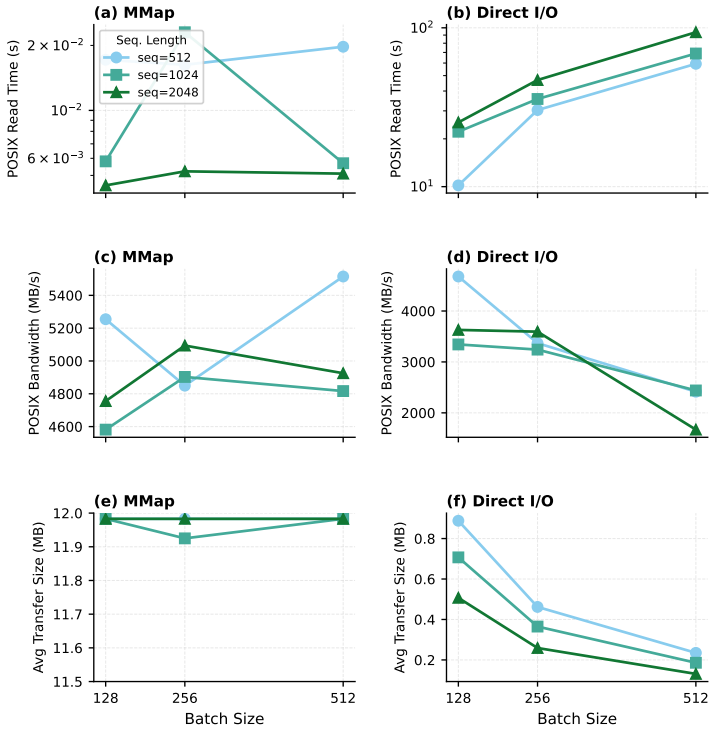

Figure 5: POSIX-level IO performance metrics across the 3×3 parameter grid. (a/b) POSIX read time (s), log scale. (c/d) POSIX aggregate bandwidth (MB/s). (e/f) POSIX average transfer size (MB).

- Under **Direct IO**, bandwidth is consistently lower and scales more weakly with sequence length, reaching only **26.6 MB/s** at the largest configuration. The per-sequence overhead of opening and closing files limits throughput regardless of how many bytes are transferred per call.
- The **MMap advantage grows with sequence length**: from a factor of **2.6**× at sequence length 512 to **4.4**× at sequence length 2048. This suggests that the page cache's prefetching effectiveness improves as individual read sizes grow.
- **Batch size has little effect** on per-sample bandwidth in either mode, confirming that the dominant factor is the number of tokens per sample, not the number of samples per batch.

5 Discussion

5.1 Effect of Access Mode on IO

MMap and Direct IO differ not only in performance but also in observability:

- Under MMap, POSIX tracing captures almost no data activity (near-constant 9 846 events, negligible read time), because page-fault-driven access bypasses the POSIX layer entirely.

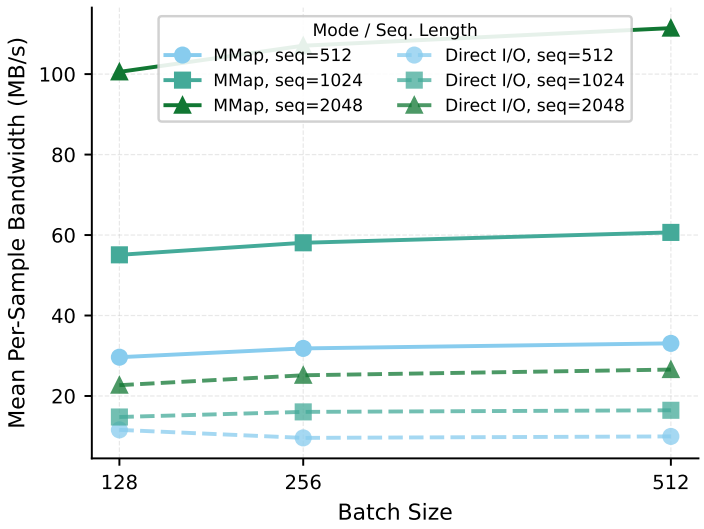

Figure 6: Mean per-sample read bandwidth (MB/s) as a function of batch size for both access modes. Lines: sequence length (512, 1024, 2048).

- Application-level DFTracer instrumentation is therefore essential. Without it, `preprocess.read_raw` and `preprocess.read_indexed` time would be invisible under MMap.

This finding has broader implications beyond the specific tools used in this study. Any IO tracing effort targeting MMap-based data pipelines must include application-level tracing; relying solely on syscall interception will produce a fundamentally incomplete picture of data movement.

5.2 Batch Size as the Primary Driver of Data Loading Time

Batch size is the dominant driver of data loader time across both modes. However, the direction of the effect depends on the access mode. Under Direct IO, large batches increase the total number of read calls because each sequence requires a separate read cycle, raising per-sample cost proportionally. Under MMap, the effect is more moderate: the dominant overhead is the CPU-bound index lookup in `preprocess.read_indexed`, which scales with sample count regardless of file access mode. This suggests that under MMap, optimizing index construction or pre-staging shuffled indices would reduce data loader time more effectively than changing the access mode itself.

5.3 Sequence Length and Read Volume

Sequence length has a secondary but consistent effect:

- Under Direct IO, longer sequences increase bytes per sample and per-access syscall count.
- The MMap bandwidth advantage grows from 2.6× (seq. 512) to 4.4× (seq. 2048), as the page cache amortizes larger sequential reads more efficiently.
- Despite this, **total job time is dominated by batch size** and remains nearly constant across sequence lengths at fixed batch size, since compute absorbs the IO

difference at this scale. At batch size 128, job time is approximately 1,750 s across all three sequence lengths in MMap mode.

The IO overhead of longer sequences would become visible at larger scale or with faster compute, where the compute-to-IO ratio shifts and data loading can no longer hide behind the training step.

5.4 Checkpoint IO is Configuration-Invariant

Checkpoint IO is constant across all 18 experiments: 5 089 POSIX writes, ≈118 GB write volume, and ≈50 s per run (out of total job times of 1 750 to 6 855 s). This confirms checkpointing depends only on model size and frequency, not training parameters. Optimizations like asynchronous or distributed checkpointing [Wan+25] can be evaluated independently of the data loading study and would benefit all configurations equally.

5.5 DFTracer Overhead

DFTracer overhead was not directly measurable from traces but is estimated at 1 to 5% based on prior characterization in the original paper [Dev+24].

5.6 Practical Implications

These results lead to three practical recommendations for LLM pretraining on HPC systems:

1. **Include application-level instrumentation** alongside POSIX tracing. Under MMap, POSIX tracing alone captures almost nothing of interest in the data loading path. DFTracer decorators inside the dataset class are necessary to expose the actual cost of index lookups and preprocessing.
2. **Account for batch size when using Direct IO.** The largest batch sizes tested produce data loader times up to ten times those of the smallest. Workloads constrained to Direct IO should prefer larger batch sizes to amortize per-sample file access cost across more tokens per step.

6 Conclusion

This work integrated DFTracer into Megatron-LM and traced the IO behavior of LLaMA-3-8B training across 18 configurations varying batch size, sequence length, and file access mode. The key findings are:

- MMap consistently outperforms Direct IO in data loader time and bandwidth, but renders data loading invisible to POSIX-level tracers, making application-level instrumentation indispensable.
- Batch size is the primary driver of data loading overhead; sequence length has a secondary effect that grows with Direct IO.
- Checkpoint IO is entirely configuration-invariant, determined solely by model size and save frequency.

These results demonstrate that effective IO tracing of LLM training requires multi-layer instrumentation and that even modest IO overheads (1 to 5% of job time) become significant when accumulated over long pretraining campaigns.

7 Future Work

Several directions can extend this study:

- **Multi-node scaling:** Extend tracing to multi-node distributed training to capture cross-node communication and parallel filesystem contention.
- **Multi-worker data loading:** The current experiments use a fixed number of DataLoader workers. Varying the worker count would reveal how parallelism in the prefetch pipeline affects data loader time and POSIX event volume, and whether worker-level IO contention becomes a bottleneck at larger batch sizes.
- **Larger datasets and models:** Evaluate whether IO proportions change at larger dataset and model scales where compute-to-IO ratios shift.
- **IO optimization evaluation:** Use the tracing infrastructure to measure the impact of optimizations such as asynchronous checkpointing, data pre-staging, and prefetching strategies.
- **Theoretical bandwidth validation:** Compare the measured bandwidth against a theoretical estimate derived from average bytes per sample and observed training throughput. Any systematic gap between the two would reveal unmeasured overheads such as framework scheduling or memory copies that the current instrumentation does not capture.

References

[Dev+21] H. Devarajan et al. "DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications". In: 2021, pp. 81–91.

[Dev+24] Hariharan Devarajan et al. "DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2024. URL: https://grc.iit.edu/publications/devarajan-2024-dftracer-8fcb/.

[Ege+24] Chris Egersdoerfer et al. "ION: Navigating the HPC I/O Optimization Journey using Large Language Models". In: *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*. 2024, pp. 86–92. DOI: 10.1145/3655038.3665950.

[Gao+21] Leo Gao et al. *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*. https://pile.eleuther.ai/. 2021.

[Hyd19] Hydra Contributors. *Hydra*. https://hydra.cc/. Open-source project. 2019.

[Kog+24] Olga Kogiou et al. "Understanding Highly Configurable Storage for Diverse Workloads". In: *IEEE International Conference on Cluster Computing*. 2024. URL: https://www.osti.gov/servlets/purl/2449727.

[LBB25] Noah Lewis, Jean Luca Bez, and Suren Byna. *I/O in Machine Learning Applications on HPC Systems: A 360-degree Survey*. https://escholarship.org/uc/item/8tc828xw. Survey of ML I/O patterns, profilers, and optimizations on HPC systems. 2025.

[Nar+21] Deepak Narayanan et al. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–15. DOI: 10.1145/3458817.3476209.

[Sny+16] Shane Snyder et al. "Modular HPC I/O Characterization with Darshan". In: *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*. 2016, pp. 9–17. DOI: 10.1109/ESPT.2016.006.

[Wan+20] Chen Wang et al. "Recorder 2.0: Efficient Parallel I/O Tracing and Analysis". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2020, pp. 1–8. DOI: 10.1109/IPDPSW50202.2020.00176.

[Wan+25] Borui Wan et al. *ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development*. 2025. arXiv: 2407.20143 [cs.AI]. URL: https://arxiv.org/abs/2407.20143.

A Installation Commands

The commands below reproduce the full software environment used in this work. They are listed here for completeness; the prose description in the main text is in Section 3.1.

Listing 1: Step 1 — Python environment and Megatron-LM.

```
module load miniforge3/24.3.0-0
conda create --name megatron_env python=3.12 -y
conda activate megatron_env

pip install "setuptools<80.0.0,>=77.0.0" "packaging>=24.2"
pip install --no-build-isolation megatron-core[mlm]

git clone https://github.com/NVIDIA/Megatron-LM.git
cd Megatron-LM
pip install pybind11 psutil
pip install --no-build-isolation .[mlm]
```

Listing 2: Step 2 — Transformer Engine (compiled from source).

```
module load gcc/13.2.0
module load cuda/12.6.2
module load cudnn/9.8.0.87-12

export CUDA_HOME=${CUDA_HOME:-$(dirname $(dirname $(which nvcc)))}

export NCCL_INCLUDE_DIR=$(python -c "import torch, os; \
    print(os.path.join(os.path.dirname(torch.__file__), \
    'include', 'torch', 'csrc', 'cuda'))")

export C_INCLUDE_PATH="${C_INCLUDE_PATH}:${NCCL_INCLUDE_DIR}"
export CPLUS_INCLUDE_PATH="${CPLUS_INCLUDE_PATH}:${NCCL_INCLUDE_DIR}"
export CPATH="${CPATH}:${NCCL_INCLUDE_DIR}"

pip install --no-build-isolation transformer-engine[pytorch]
python -c "import transformer_engine as te; \
    print(f'Transformer Engine version: {te.__version__}')"
```

B Dataset Preprocessing Script

Listing 3 shows the full SLURM batch script (`examples/llama/prepare_pile_5gb.sh`) used to download and tokenize the Pile dataset subset. The script is idempotent: it skips the download phase if the JSONL file already exists and skips tokenization if the `.bin`/`.idx` pair is already present.

Listing 3: SLURM job for downloading and tokenizing Pile dataset.

```
#!/bin/bash
#SBATCH --job-name=pile_prep
#SBATCH --partition=grete-h100:shared
#SBATCH --gres=gpu:H100:1        # GPU node needed so TE can load libcudart
#SBATCH -C inet
#SBATCH --mem=32G
#SBATCH --time=02:00:00
#SBATCH --cpus-per-task=16
#SBATCH --output=logs/pile_prep_%j.out
#SBATCH --error=logs/pile_prep_%j.err

set -euo pipefail

#  Environment
module load gcc/13.2.0; module load cuda/12.6.2; module load cudnn/9.8.0.87-12
module load miniforge3/24.3.0-0
conda activate megatron_env

WORKSPACE_DIR="/mnt/ceph-ssd/workspaces/.../u25131-llm-io-workspace/"
OUTPUT_PREFIX="${WORKSPACE_DIR}/pile_subset"
JSONL_FILE="${WORKSPACE_DIR}/pile_5gb.jsonl"
TOKENIZER="meta-llama/Meta-Llama-3-8B"
TARGET_SAMPLES=900000      #  5 GB tokenized

#  Step 1: Download from HuggingFace (streaming, resumes if partial)
if ! [ -f "$JSONL_FILE" ] || [ "$(wc -l < "$JSONL_FILE")" -lt "$TARGET_SAMPLES" ];
↪  then
    python3 -c "
import json
from datasets import load_dataset
target = $TARGET_SAMPLES; jsonl_path = '$JSONL_FILE'
ds = load_dataset('monology/pile-uncopyrighted', split='train', streaming=True)
written = 0
with open(jsonl_path, 'w', buffering=8*1024*1024) as f:
    for sample in ds:
        text = sample.get('text', '')
        if text.strip():
            f.write(json.dumps({'text': text}) + '\n')
            written += 1
            if written >= target: break
print(f'Done: {written:,} samples')
"
fi

#  Step 2: Tokenize into Megatron binary format (.bin + .idx)
if ! [ -f "${OUTPUT_PREFIX}_text_document.bin" ]; then
    python tools/preprocess_data.py \
```

```
        --input           "$JSONL_FILE" \
        --output-prefix   "${OUTPUT_PREFIX}" \
        --tokenizer-type HuggingFaceTokenizer \
        --tokenizer-model "$TOKENIZER" \
        --workers         "${SLURM_CPUS_PER_TASK:-16}" \
        --append-eod \
        --json-keys       text
fi

echo "DATA_PATH for training: ${OUTPUT_PREFIX}_text_document"
```

C Parameter-Sweep Submit Script

Listing 4 shows the SLURM job-array submit script for Megatron training. The two environment variables `SEQ_LENGTH` and `GLOBAL_BATCH_SIZE` are exported and consumed by the core training script.

Listing 4: SLURM job-array sweep script for the mmap campaign (`examples/llama/submit_4gpu_h100_sweep.sh`).

```
#!/bin/bash
#SBATCH --job-name=llama3_8b_h100_sweep_mmap
#SBATCH --partition=grete-h100:shared
#SBATCH --account=nhr_ni_test
#SBATCH --gres=gpu:H100:4
#SBATCH -C inet
#SBATCH --mem=240G
#SBATCH --time=48:00:00
#SBATCH --cpus-per-task=32
#SBATCH --output=logs/h100_sweep_mmap_%A_%a.out
#SBATCH --error=logs/h100_sweep_mmap_%A_%a.err
#SBATCH --array=0-8   # 3 seq lengths x 3 batch sizes = 9 combinations

#  Parameter grid
SEQ_LENGTHS=(512 1024 2048)
BATCH_SIZES=(128 256 512)
SEQ_IDX=$(( SLURM_ARRAY_TASK_ID / 3 ))
BS_IDX=$(( SLURM_ARRAY_TASK_ID % 3 ))
export SEQ_LENGTH=${SEQ_LENGTHS[$SEQ_IDX]}
export GLOBAL_BATCH_SIZE=${BATCH_SIZES[$BS_IDX]}
export NO_MMAP_BIN_FILES=0   # 0 = memory-mapped; 1 = direct read

#  Runtime environment
unset LD_PRELOAD
module load gcc/13.2.0; module load cuda/12.6.2; module load cudnn/9.8.0.87-12
module load miniforge3/24.3.0-0
conda activate megatron_env

# Force CUDA 12 runtime to avoid conflict with PyTorch-bundled CUDA 13
export LD_PRELOAD=$(find "${CUDA_HOME}/lib64" -name "libcudart.so.12.*" \
    ! -type l | head -1)

#  DFTracer configuration
```

```
WORKSPACE_DIR="/mnt/ceph-ssd/workspaces/.../u25131-llm-io-workspace/RedPajama"
export DFTRACER_ENABLE=1
export DFTRACER_INC_METADATA=1
export DFTRACER_LOG_DIR="${WORKSPACE_DIR}/dftracer_logs/\
${SLURM_JOB_NAME}_${SLURM_ARRAY_JOB_ID}_task${SLURM_ARRAY_TASK_ID}_\
mmap_seq${SEQ_LENGTH}_bs${GLOBAL_BATCH_SIZE}"
export DFTRACER_DATA_DIR="${WORKSPACE_DIR}"
mkdir -p "$DFTRACER_LOG_DIR"

#  Launch
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
RUN_TAG="mmap_seq${SEQ_LENGTH}_bs${GLOBAL_BATCH_SIZE}_${TIMESTAMP}"
CHECKPOINT_DIR="${WORKSPACE_DIR}/checkpoints/llama3_8b_4h100_${RUN_TAG}"
TENSORBOARD_DIR="${WORKSPACE_DIR}/tensorboard_logs/llama3_8b_4h100_${RUN_TAG}"
DATA_PATH="${WORKSPACE_DIR}/pile_subset_text_document"

bash examples/llama/train_llama3_8b_4gpu_h100.sh \
    "$CHECKPOINT_DIR" "$TENSORBOARD_DIR" \
    meta-llama/Meta-Llama-3-8B "$DATA_PATH"
```

D Core Training Script

Listing 5 shows the core training script (`examples/llama/train_llama3_8b_4gpu_h100.sh`). Two arguments deserve specific attention:

- `-no-gradient-accumulation-fusion` disables the CUDA kernel that fuses gradient accumulation into a single pass. On this system the fused kernel produced incorrect results when combined with the FP8 Transformer Engine path and BF16 gradient reduction, manifesting as NaN losses after the first iteration. Disabling the fusion restores numerically correct behaviour at a small throughput cost.
- `-attention-backend unfused` disables Transformer Engine's fused attention kernel (FlashAttention) and falls back to the native PyTorch unfused path. The fused kernel raised a CUDA illegal-memory-access error during the first forward pass on this cluster's H100 driver stack; switching to the unfused backend eliminated the crash while keeping FP8 computation active for all other operations.

Listing 5: Core LLaMA-3-8B training script for 4 H100 GPUs.

```
#!/bin/bash
# examples/llama/train_llama3_8b_4gpu_h100.sh
export CUDA_DEVICE_MAX_CONNECTIONS=1

WORKSPACE_DIR="/mnt/ceph-ssd/workspaces/.../u25131-llm-io-workspace/RedPajama"
CHECKPOINT_DIR=$1; TENSORBOARD_DIR=$2; TOKENIZER=$3; DATA_PATH=$4
TRAIN_ITERS=${5:-100}

GPUS_PER_NODE=${GPUS_PER_NODE:-4}; NNODES=${NNODES:-1}
NODE_RANK=${NODE_RANK:-0}
MASTER_ADDR=${MASTER_ADDR:-localhost}; MASTER_PORT=${MASTER_PORT:-6000}
```

```
12
13  NUM_LAYERS=32; HIDDEN_SIZE=4096; NUM_ATTENTION_HEADS=32
14  NUM_KEY_VALUE_HEADS=8; FFN_HIDDEN_SIZE=14336
15
16  SEQ_LENGTH=${SEQ_LENGTH:-512}
17  MAX_POSITION_EMBEDDINGS=${SEQ_LENGTH}
18  MICRO_BATCH_SIZE=1
19  GLOBAL_BATCH_SIZE=${GLOBAL_BATCH_SIZE:-128}
20  TP_SIZE=${TP_SIZE:-2}; PP_SIZE=${PP_SIZE:-1}
21
22  FP8_ARGS="--bf16 --fp8-format hybrid --fp8-amax-history-len 1024 \
23            --fp8-amax-compute-algo max"
24  DATA_CACHE_DIR="${WORKSPACE_DIR}/data_cache"
25
26  DISTRIBUTED_ARGS="
27      --nproc_per_node $GPUS_PER_NODE --nnodes $NNODES
28      --node_rank $NODE_RANK --master_addr $MASTER_ADDR
29      --master_port $MASTER_PORT
30  "
31
32  GPT_MODEL_ARGS="
33      --num-layers $NUM_LAYERS --hidden-size $HIDDEN_SIZE
34      --num-attention-heads $NUM_ATTENTION_HEADS
35      --num-query-groups $NUM_KEY_VALUE_HEADS
36      --ffn-hidden-size $FFN_HIDDEN_SIZE
37      --seq-length $SEQ_LENGTH
38      --max-position-embeddings $MAX_POSITION_EMBEDDINGS
39      --position-embedding-type rope --rotary-base 500000
40      --use-rotary-position-embeddings
41      --swiglu --normalization RMSNorm
42      --disable-bias-linear --untie-embeddings-and-output-weights
43      --no-position-embedding --no-masked-softmax-fusion
44      --attention-softmax-in-fp32
45      # *** REQUIRED: fall back to unfused attention kernel;
46      #     the TE FlashAttention kernel raises a CUDA
47      #     illegal-memory-access on this driver stack ***
48      --attention-backend unfused
49  "
50
51  TRAINING_ARGS="
52      --micro-batch-size $MICRO_BATCH_SIZE
53      --global-batch-size $GLOBAL_BATCH_SIZE
54      --train-iters $TRAIN_ITERS
55      --weight-decay 0.1 --adam-beta1 0.9 --adam-beta2 0.95
56      --init-method-std 0.006 --clip-grad 1.0
57      ${FP8_ARGS}
58      --lr 1.5e-4 --lr-decay-style cosine --min-lr 1.0e-5
59      --lr-decay-iters $(( TRAIN_ITERS * 9 / 10 ))
60      --lr-warmup-iters $(( TRAIN_ITERS / 10 ))
61      # *** REQUIRED: disabling the fused gradient-accumulation kernel
62      #     prevents NaN losses that appear when it runs alongside
63      #     the FP8 + BF16-gradient-reduction path ***
64      --no-gradient-accumulation-fusion
65      --recompute-activations
66  "
67
68  MODEL_PARALLEL_ARGS="
```

```
    --tensor-model-parallel-size $TP_SIZE
    --pipeline-model-parallel-size $PP_SIZE
    --use-distributed-optimizer
    --overlap-grad-reduce --overlap-param-gather
"

# Binary data IO mode (default: no-mmap; set NO_MMAP_BIN_FILES=0 for mmap)
if [ "${NO_MMAP_BIN_FILES:-1}" != "0" ]; then
    NO_MMAP_BIN_ARG="--no-mmap-bin-files"
else
    NO_MMAP_BIN_ARG=""
fi

DATA_ARGS="
    --data-path $DATA_PATH
    --tokenizer-type HuggingFaceTokenizer
    --tokenizer-model $TOKENIZER
    --split 949,50,1
    --data-cache-path $DATA_CACHE_DIR
    ${NO_MMAP_BIN_ARG}
"

EVAL_AND_LOGGING_ARGS="
    --log-interval 1
    --save-interval $TRAIN_ITERS
    --eval-interval $(( TRAIN_ITERS / 2 ))
    --eval-iters 10
    --save $CHECKPOINT_DIR --load $CHECKPOINT_DIR
    --tensorboard-dir $TENSORBOARD_DIR
    --log-throughput --log-params-norm
"

torchrun $DISTRIBUTED_ARGS pretrain_gpt.py \
    $GPT_MODEL_ARGS $TRAINING_ARGS $MODEL_PARALLEL_ARGS \
    $DATA_ARGS $EVAL_AND_LOGGING_ARGS
```

E DFTracer Integration Code

The listings below show every DFTracer instrumentation point added to Megatron-LM. Each listing header names the file that was modified. All files share a common graceful-fallback import block: if DFTracer is not installed, a no-op stub is used so the code runs unchanged in uninstrumented environments.

Listing 6: DFTracer integration in `pretrain_gpt.py`

```python
# File: pretrain_gpt.py
#  DFTracer AI tracing

import dftracer.python as _dftracer_mod
from dftracer.python import ai
_DFTRACER_AVAILABLE = True


# Decorated functions (elsewhere in the file):
@ai.dataloader.fetch
def get_batch(data_iterator, vp_stage=None):
    """Fetch one training batch - traced as a data-load event."""
    ...

@ai.compute.forward
def forward_step(data_iterator, model, ...):
    """One forward pass - traced as a compute-forward event."""
    ...

# In __main__: initialise logger in the main process
if _DFTRACER_AVAILABLE:
    _dft_log_file = os.path.join(DFTRACER_LOG_DIR, f"ai_trace_{os.getpid()}.pfw")
    _df_logger = _dftracer_mod.dftracer.initialize_log(
        _dft_log_file, DFTRACER_DATA_DIR, os.getpid())

# After pretrain() returns: flush and close the trace file
if _df_logger is not None:
    _df_logger.finalize()
```

Listing 7: DFTracer integration in `megatron/core/datasets/gpt_dataset.py`.

```python
# File: megatron/core/datasets/gpt_dataset.py
from dftracer.python import ai
_DFTRACER_AVAILABLE = True


#  __getitem__: outer data-item event + inner preprocess context
@ai.data.item
def __getitem__(self, idx: Optional[int]) -> Dict[str, torch.Tensor]:
    ...
    with ai.data.preprocess:
        text = torch.from_numpy(text).long()
        tokens = text[:-1].contiguous()
        labels = text[1:].contiguous()
        ...

```

```
#  _query_document_sample_shuffle_indices: per-sample read size metadata
@ai.data.preprocess.derive(name="read_indexed")
def _query_document_sample_shuffle_indices(self, idx: int):
    ...
    sizes = (numpy.sum(samples.shape) * samples.dtype.itemsize
           + numpy.sum(document_indices.shape) * document_indices.dtype.itemsize)
    ai.update(image_size=sizes, image_idx=idx)
    return (samples, document_indices)

#  _get_ltor_masks_and_position_ids: mask/position-ID construction
@ai.data.derive(name="read_indexed")
def _get_ltor_masks_and_position_ids(data, eod_token, ...):
    ...
```

Listing 8: DFTracer integration in `megatron/core/datasets/indexed_dataset.py`: both the mmap and direct-read paths are traced, enabling side-by-side comparison of the two IO modes.

```
# File: megatron/core/datasets/indexed_dataset.py

from dftracer.python import ai
_DFTRACER_AVAILABLE = True

#  _MMapBinReader.read: memory-mapped binary read
@ai.data.preprocess.derive(name="mmap_read")
def read(self, dtype, count, offset) -> numpy.ndarray:
    """Memory-mapped read from .bin file."""
    return numpy.frombuffer(self._bin_buffer, dtype=dtype,
                            count=count, offset=offset)

#  _FileBinReader.read: direct-IO binary read
@ai.data.preprocess.derive(name="file_read")
def read(self, dtype, count, offset) -> numpy.ndarray:
    """POSIX read() from .bin file (no-mmap mode)."""
    sequence = numpy.empty(count, dtype=dtype)
    with open(self._bin_path, mode="rb", buffering=0) as f:
        f.seek(offset)
        f.readinto(sequence)
    return sequence
```

Listing 9: DFTracer integration in `megatron/training/training.py`: step and pipeline events bracket individual optimiser steps and the full training loop respectively.

```
# File: megatron/training/training.py
from dftracer.python import ai
_DFTRACER_AVAILABLE = True


#  train_step: one optimiser step
@ai.compute.step
def train_step(forward_step_func, data_iterator, model,
               optimizer, opt_param_scheduler, config, forward_backward_func):
    ...

```

```
#  train: full training loop
@ai.pipeline.train
def train(forward_step_func, model, optimizer, opt_param_scheduler,
          train_data_iterator, valid_data_iterator, ...):
    ...
```

Listing 10: DFTracer integration in `megatron/training/checkpointing.py`: every checkpoint write is bracketed so its IO cost can be separated from training-step IO in the analysis.

```
# File: megatron/training/checkpointing.py
from dftracer.python import ai
_DFTRACER_AVAILABLE = True


#  save_checkpoint: checkpoint write event
@ai.checkpoint.capture
def save_checkpoint(iteration, model, optimizer,
                    opt_param_scheduler, ...):
    """Save model/optimizer checkpoint - traced as a checkpoint event."""
    ...
```

Listing 11: DFTracer worker-process initialisation in `megatron/training/datasets/data_samplers.py`. Without this, DataLoader workers would not emit any AI-level trace events.

```
# File: megatron/training/datasets/data_samplers.py
# DataLoader workers are spawned as separate processes and do NOT
# inherit the DFTracer logger created in the main process.
# _dftracer_worker_init_fn() is passed as worker_init_fn to DataLoader
# so that each worker initialises its own logger on startup.

import dftracer.python as _dftracer_mod
_DFTRACER_AVAILABLE = True


import atexit, os

def _dftracer_worker_init_fn(worker_id):
    if not _DFTRACER_AVAILABLE:
        return
    if not os.environ.get("DFTRACER_ENABLE", "0") == "1":
        return
    log_dir  = os.environ.get("DFTRACER_LOG_DIR",  ".")
    data_dir = os.environ.get("DFTRACER_DATA_DIR", ".")
    pid      = os.getpid()
    log_file = os.path.join(log_dir, f"ai_trace_worker_{pid}.pfw")
    os.makedirs(log_dir, exist_ok=True)
    _dftracer_worker_init_fn._logger = \
        _dftracer_mod.dftracer.initialize_log(log_file, data_dir, pid)
    atexit.register(
        lambda: _dftracer_worker_init_fn._logger.finalize()
        if getattr(_dftracer_worker_init_fn, '_logger', None) else None
    )
```

F Trace post-processing and analysis commands

This appendix lists the shell commands used to merge per-job shards, run DFAnalyzer, and extract additional POSIX-level statistics from `combined.pfw`. All steps assume `conda activate megatron_env` (or an environment where `dfanalyzer` and `dftracer_merge` are on `PATH`) and a working directory at the project root unless noted.

Merge trace shards into one file

Inside each trace directory that contains multiple `.pfw`/`.pfw.gz` files, run:

```
cd /path/to/dftracer_logs/<job_trace_directory>
dftracer_merge    # writes combined.pfw in the current directory
```

If `combined.pfw` already exists and must be rebuilt, use `dftracer_merge -f`.

DFAnalyzer (DLIO preset, console output)

From the project root (so that `overrides.yaml` resolves when passed as `+override=`), with `trace_path` pointing at the trace folder:

```
dfanalyzer \
  "trace_path=./dftracer_logs/<job_trace_directory>" \
  "analyzer/preset=dlio" \
  "output=console" \
  "+override=overrides"
```

The same analysis can be driven entirely from the command line by repeating the override keys that appear in `overrides.yaml`, for example:

```
dfanalyzer \
  "trace_path=./dftracer_logs/<job_trace_directory>" \
  "analyzer/preset=dlio" \
  "output=console" \
  "++analyzer.preset.size_layers=[posix,reader_posix,checkpoint_posix,data_loader]" \
  "++analyzer.preset.size_derived_metrics.data_loader=[preprocess_read_indexed]" \
  "++analyzer.preset.derived_metrics.data_loader.preprocess_read_indexed=\
'func_name==\"preprocess.read_indexed\"'" \
  "++analyzer.preset.additional_metrics.proc_name.image_bw=\
'data_loader_preprocess_read_indexed_size_sum.fillna(0)/\
data_loader_preprocess_read_indexed_time_sum'" \
  "++analyzer.preset.additional_metrics.time_range.image_bw=\
'data_loader_preprocess_read_indexed_size_sum.fillna(0)/\
data_loader_preprocess_read_indexed_time_max'"
```

`zgrep`/`jq` queries on `combined.pfw`

The merged file is line-oriented JSON (and may be searched with `zgrep` even when uncompressed). The following pipelines were used for auxiliary CSV-style summaries; run them from the directory that contains `combined.pfw`.

Per-category event counts (after selecting lines that contain the substring `cat` so that JSON objects expose a `cat` field):

```
zgrep cat combined.pfw \
  | jq -c '.cat' | sort | uniq -c
```

POSIX complete events (`ph=="X"`) — aggregate duration by syscall name (`dur` in microseconds in the trace; divided by 10^6 for seconds):

```
zgrep -h '"ph":"X"' combined.pfw \
  | jq -r 'select(.cat=="POSIX") | [.cat, .name, (try (.dur|tonumber) catch empty)]
  ↪  | @tsv' \
  | awk 'NF==3{sum[$1 FS $2]+=$3} END{for(k in sum){split(k,a,FS); \
      printf "%s\t%s\t%.6f s\n", a[1], a[2], sum[k]/1e6}}' \
  | sort -k1,1 -k3,3nr
```

Data-layer complete events (same pattern, `cat=="data"`):

```
zgrep -h '"ph":"X"' combined.pfw \
  | jq -r 'select(.cat=="data") | [.cat, .name, (try (.dur|tonumber) catch empty)] |
  ↪  @tsv' \
  | awk 'NF==3{sum[$1 FS $2]+=$3} END{for(k in sum){split(k,a,FS); \
      printf "%s\t%s\t%.6f s\n", a[1], a[2], sum[k]/1e6}}' \
  | sort -k1,1 -k3,3nr
```

POSIX operation counts (histogram of `.name` for complete POSIX events):

```
zgrep -h '"ph":"X"' combined.pfw \
  | jq -r 'select(.cat=="POSIX") | .name' \
  | sort | uniq -c | sort -rn \
  > posix_op_count.csv
```