

---

# MEGATRONAPP: EFFICIENT AND COMPREHENSIVE MANAGEMENT ON DISTRIBUTED LLM TRAINING

---

**Bohan Zhao, Yongchao He**  
 Suanzhi Future  
 contact@siflow.cn

**Guang Yang, Shuo Chen, Ruitao Liu, Tingrui Zhang, Wei Xu**  
 Shanghai Qi Zhi Institute  
 xuwei@sqz.ac.cn

## ABSTRACT

The rapid escalation in the parameter count of large language models (LLMs) has transformed model training from a single-node endeavor into a highly intricate, cross-node activity. While frameworks such as Megatron-LM successfully integrate tensor (TP), pipeline (PP), and data (DP) parallelism to enable trillion-parameter training, they simultaneously expose practitioners to unprecedented systems-level challenges in performance optimization, diagnosis, and interpretability. MegatronApp is an open-source toolchain expressly designed to meet these challenges. It introduces four orthogonal, yet seamlessly composable modules—MegaScan, MegaFBD, MegaDPP, and MegaScope—that collectively elevate the reliability, efficiency, and transparency of production-scale training. This paper presents the motivation, architecture, and distinctive contributions of each module, and elucidates how their synergistic integration augments the Megatron-LM ecosystem.

## 1 Introduction

Large-scale transformer [1] training now operates at cluster scale, where thousands of accelerators coordinate over bandwidth-constrained networks and failure-prone hardware. Minor perturbations such as transient GPU throttling or link jitter can cascade into significant slowdowns, while the opaqueness of distributed execution hampers root-cause analysis. At the same time, the interpretability community increasingly demands fine-grained introspection [2, 3, 4, 5, 6] of model states during training—yet naïvely capturing activations or attention maps in trillion-parameter models [7, 8, 9] quickly saturates I/O subsystems.

Existing profiling, monitoring, and visualization tools [10, 11, 12] provide partial relief but suffer three key limitations: (i) lack of temporal causality modeling, rendering them ineffective at distinguishing sources from victims of performance anomalies; (ii) insufficient awareness of deep-learning communication semantics, resulting in high false-positive rates under structured parallelism [13, 14]; and (iii) rigid data-collection pipelines that either couple tightly to user code or impose prohibitive overheads. MegatronApp addresses these deficiencies through a suite of lightweight extensions that require only minimal code changes yet expose rich, semantically meaningful telemetry.

We summarize our contributions as follows:

**MegaScan:** a CUDA-event-driven tracing engine that performs on-line slow-node detection and root-cause localization at operator granularity.

**MegaFBD:** a forward-backward decoupling layer that reallocates the two phases across heterogeneous resources, mitigating memory contention and improving utilization.

**MegaDPP:** a dynamic pipeline scheduler that adapts traversal order and communication overlap to network and compute volatility.

**MegaScope:** a pluggable visualization and perturbation framework that enables low-overhead capture, compression, and interactive rendering of intermediate tensors.

Each component is optional, self-contained, and activated via simple runtime flags, preserving compatibility with upstream Megatron-LM. MegatronApp is publicly available at <https://github.com/OpenSQZ/MegatronApp>.

## 2 Background

### 2.1 The Era of Trillion-Parameter LLMs

Model size has grown exponentially since the original transformer, vaulting from millions to *trillions* of parameters in under a decade. NVIDIA’s Megatron project [15, 13, 14] first demonstrated that a GPT-style model with one trillion parameters could be trained in practicable wall-clock time by combining tensor, pipeline, and data parallelism across 3072 A100 GPUs. This escalation in scale makes distributed training not a luxury but a necessity: even if a single accelerator had enough memory, compute time would be prohibitive (e.g., 36 years on eight V100s for GPT-3 [16].

### 2.2 Distributed Training Fundamentals

Modern LLM training relies on three orthogonal parallelism strategies:

- **Data parallelism (DP)** [16, 17] replicates the model and splits the batch across devices, synchronizing gradients with AllReduce. Its conceptual simplicity yields near-linear scaling until memory limits bite.
- **Tensor parallelism (TP)** [13] partitions weight matrices so that a single layer spans multiple GPUs, reducing per-device memory at the expense of tighter communication coupling.
- **Pipeline parallelism (PP)** [18, 19, 20] allocates consecutive layer blocks to different devices and overlaps micro-batch execution; bubble overhead and activation storage dominate its efficiency.

State-of-the-art systems combine all three dimensions—commonly dubbed *3-D parallelism*—to scale well beyond 10 B parameters.

### 2.3 Ecosystem Landscape

**Megatron-LM.** Originally introduced by NVIDIA, Megatron-LM [13] remains the de-facto reference for GPT-style training with tight integration of TP, PP, and DP.

**DeepSpeed and ZeRO.** Microsoft’s DeepSpeed [21] library tackles optimizer and activation memory via ZeRO-3/Infinity offloading [22] and NVMe staging, enabling 500 B-scale models on modest clusters.

**PyTorch FSDP and Others.** PyTorch 2.3 stabilised Fully-Sharded Data Parallel (FSDP) [23], bringing zero-redundancy sharding into the core framework, while projects such as MosaicML Composer and Hugging Face Accelerate target ease-of-use for sub-100 B models.

#### 2.3.1 LLM Interpretability

A growing body of work seeks to open the “black box” of Transformer-based LLMs. Zhao *et al.* offer one of the earliest systematic taxonomies, covering local- and global-explanation techniques, attribution methods, and probing tasks [24]. Luo and Specia extend this line with a 2024 survey that also catalogues practical uses of explanations—e.g., model editing and controllable generation—highlighting the tension between faithfulness and usability [25]. Mechanistic-interpretability efforts [26, 27, 28] (e.g. neuron activation patching, causal tracing) complement these surveys but remain nascent for trillion-parameter scales, motivating the interactive visualization that MegaScope supplies.

#### 2.3.2 Pipeline-Parallel Scheduling

Pipeline parallelism amortises memory across stages but suffers from bubble overhead and load imbalance. A 2024 comprehensive review contrasts synchronous (GPIPE [18]) and asynchronous (PIPEDREAM [19]) schedules, emphasising stage-wise profiling and recomputation to minimise idle time [29]. More recently, DAWNPIPE [30] introduces a compilation-based profiler and dynamic chunking to shrink stage memory while keeping utilization high in multi-GPU clusters. Qi *et al.* propose Zero Bubble Pipeline Parallelism (ZBPP) to (almost) eliminate pipeline bubbles by carefully interleaving forward/backward steps and weight-gradient computations [31]. DualPipe introduces a *bidirectional, dual-channel* pipeline that *fully overlaps* forward/backward computation with communication [7]. These studies underscore the need for runtime visibility into per-stage latency—functionality built into MegaScope and exploited by MegaDPP for adaptive scheduling.

### 2.3.3 Straggler Detection and Mitigation

Stragglers—GPUs that throttle or experience link jitter—can slash throughput by cascading delays through tightly coupled collectives. A 2025 ByteDance trace study quantifies this impact, using what-if analysis to show that removing the slowest 5% of iterations yields up to 38% speed-ups [32]. Earlier, DPRO-SM proposed LSTM-based runtime prediction to pre-emptively migrate work away from prospective stragglers [33]. **Greyhound** takes a complementary, production-driven angle: it first *characterises* more than 10 000-GPU jobs on an industrial cluster and finds that *fail-slow-transient*, sub-minute-to-multi-hour stragglers—inflate job runtime by  $1.34\times$  on average [34]. These works reinforce the importance of fine-grained latency telemetry, which **MegaScan** captures via CUDA-event tracing and analyses efficiently using its multi-stage heuristic algorithm.

### 2.3.4 Heterogeneous Computing for LLM Training

As accelerator diversity widens, frameworks must seamlessly span GPUs, CPUs, NPUs, and emerging ISAs. Jiang *et al.* outline a unified architecture that shards model states across heterogeneous GPU/CPU clusters, leveraging smart replica placement and prioritized gradient staging to sustain bandwidth parity [35]. NVIDIA’s 2025 announcement of CUDA support for RISC-V CPUs signals further heterogeneity at the system-software layer, opening the door to bespoke edge chips orchestrating large-model workloads [36]. Prior arts like FlashFlex [35, 37] solves training LLMs on heterogeneous clusters treat device diversity as a first-class scheduling signal, rather than forcing homogeneous parallelism across all participants. **MegaFBD** builds on these insights by decoupling forward and backward passes, thereby freeing the scheduler to map lighter compute onto CPUs or other devices with poorer performance while reserving the fastest GPUs for compute-dense stages.

Collectively, these four research streams motivate the design choices in **MegatronApp**: trace-driven interpretability hooks, adaptive pipeline orchestration, fast straggler detection, and flexible device placement—features essential for pushing LLM training into the heterogeneous, trillion-parameter era.

## 3 **MegaScan**: Operator-Granular Tracing and Straggler Detection

### 3.1 Motivation and Goals

As the parameter scale of Transformer-based large language models (LLMs) grows exponentially, model training has evolved from single-machine computation to a distributed, cross-node cooperative task. Training frameworks exemplified by *Megatron-LM* make trillion-parameter model training possible through an organic combination of tensor parallelism, pipeline parallelism, and data parallelism.

Although these parallel strategies significantly boost training throughput, they also greatly increase system-level complexity; the training system becomes extremely sensitive to hardware stability and communication reliability as the number of nodes rises. In real-world deployments, transient faults (such as GPU down-clocking or link jitter) often trigger cascading delays that degrade training performance. More critically, because the communication topology is highly coupled, fault signals propagate along hidden paths with uncertain impact scopes, making root-cause localization particularly challenging.

Current mainstream performance-monitoring and anomaly-detection methods face three key challenges:

1. **Lack of temporal-dependency modeling.** Traditional metrics (e.g., GPU utilization) only reflect local states and cannot capture causal chains between nodes. For instance, although NVIDIA’s DCGM tool [38] can collect metrics such as memory usage and bandwidth, it struggles to tell influencers from those affected.
2. **Missing communication-pattern modeling.** Most methods target general distributed systems and therefore fail to capture the highly structured communication semantics of training workloads. This results in high false-positive rates, especially under specific configurations such as tensor or pipeline parallelism.
3. **Confounding of hardware and software anomalies.** Hardware issues-like GPU down-clocking or power fluctuations-exhibit temporal patterns similar to software factors such as data-distribution skew or batch-size changes, yet the detection strategies required for each are fundamentally different.

**MegaScan** tackles these challenges by introducing a temporal-analysis and slow-node-detection framework at the granularity of application-level training operators. The system builds a lightweight tracing infrastructure centered on CUDA Events [39], unifies visualization via the Chrome Tracing [40] format, and performs root-cause diagnosis through cross-rank dependency analysis, clock alignment, and bandwidth assessment. Compared with traditional approaches, **MegaScan** aligns more closely with actual training semantics while offering extremely low intrusiveness and strong scalability.

### 3.2 Design

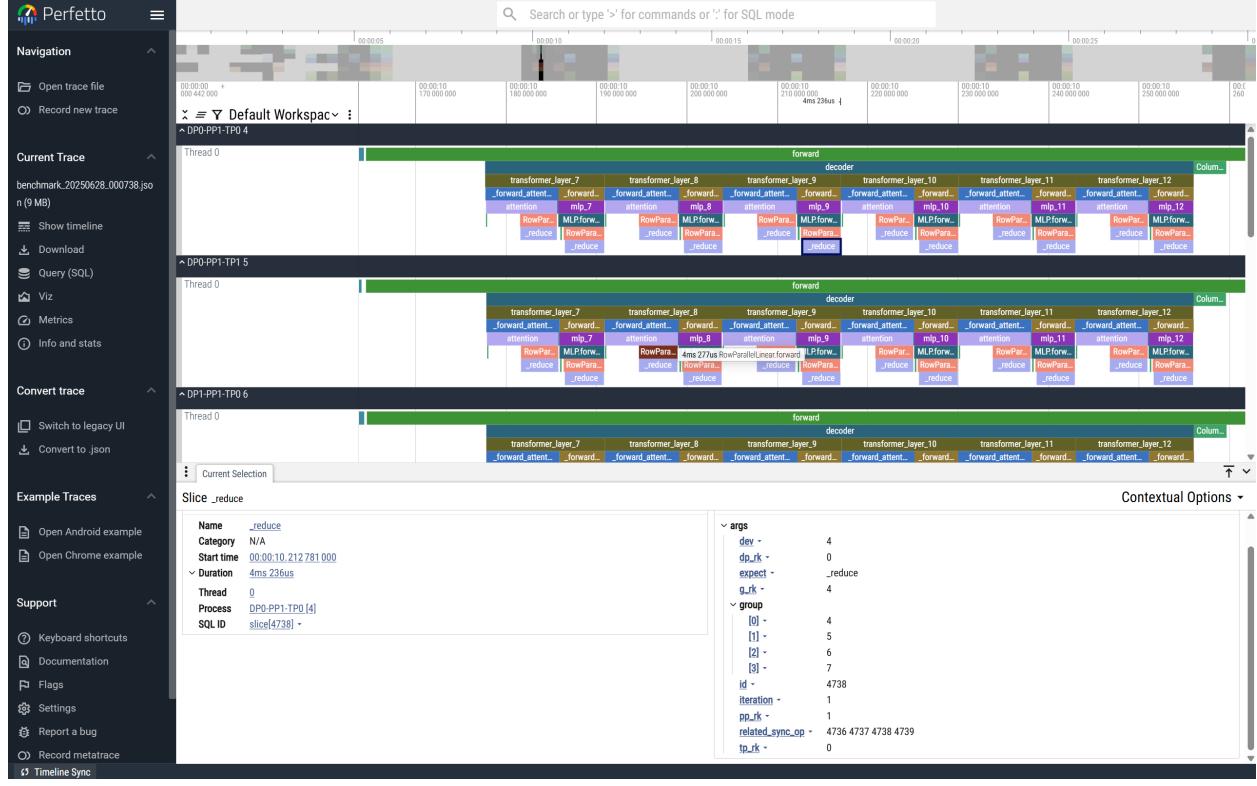


Figure 1: MegaScan visualizes the trace file using Chrome Tracing (or Perfetto UI).

**Workload tracing.** MegaScan adopts a timing mechanism based on *CUDA Events*. A CUDA event is a special marker, injected by the host into a CUDA stream, that is essentially an empty kernel. When the GPU’s execution flow reaches this marker, it records the current timestamp. By inserting a *start* event and an *end* event in the same CUDA stream immediately before and after an operation under test (e.g. a compute kernel or a communication call), we can later query the elapsed time between the two events asynchronously via `cudaEventElapsedTime`. This time difference precisely reflects the true execution time of that operation on the GPU. Recording a CUDA event is lightweight, and fetching the timestamps as well as computing the difference can be done asynchronously, avoiding any synchronization overhead on the critical execution path. Consequently, the event-based approach captures the real latency of asynchronous operations with high fidelity while imposing negligible performance impact on the original training workload, making it an ideal foundation for the low-intrusion tracing framework required by our system.

Beyond recording event names and timestamps, the tracing framework allows additional *metadata* (Arguments) to be attached to each event—such as the current micro-batch index, the amount of data involved in a communication, or the peer rank—by passing extra parameters to `tracers.scope`. These metadata are essential for later constructing a global dependency graph, understanding system behavior, and performing fault diagnosis.

**Log pre-processing.** During distributed training, each GPU process (*rank*) independently produces its own trace, and rank 0 gathers all traces and persists them to disk. The gathering and persistence operations are conducted in a separate thread to alleviate training stalls. After training, every rank has its own recorded event sequence as a JSON file. Because each rank timestamps events with its local GPU clock, directly merging these files does not yield an accurate global timeline. To obtain a consistent global view for cross-rank analysis, we aggregate and align the independent trace files. We implement an aggregation script that reads all per-rank JSON files and merges them—ordered by time—into a single JSON file conforming to the *Chrome Tracing Format*. Originally designed for analyzing asynchronous events and performance bottlenecks inside the Chrome browser, this open JSON-based standard is now widely used for visualizing temporal event data. Users can load the merged trace in the built-in viewer (`chrome://tracing`) without additional software (as shown in Figure 1). The viewer clearly displays, along a timeline, the start/end times, exact durations, concurrency, hierarchical nesting, and logical flow of events across different processes or threads (each rank is mapped to a separate process in our scenario). Users can zoom, pan, and search for specific events, examining the detailed

parameters we carefully recorded—such as micro-batch IDs, communication volume, and rank lists—directly in the GUI. Moreover, thanks to its broad adoption, Chrome Tracing enjoys mature community support and toolchains for data sharing, conversion, and further development.

**Dependency reconstruction.** After obtaining the aggregated—yet not fully aligned—trace, a key step is to explicitly link synchronous communication operations executed on different ranks. In distributed training, synchronous collectives such as `AllReduce`, `AllGather`, and `Broadcast`, as well as point-to-point (`Send/Recv`) operations, require all participating ranks to reach a common synchronization point before the operation can complete. Identifying which concrete event instances form the *same* communication operation is therefore crucial for subsequent analyses. To enable this, our tracer records, in addition to latency, the *process-group* or *peer-rank* information for every traced communication call. For collective operations we log the global ID list of all participating ranks; a single pass over the events then matches those that belong to the same communication instance.

**Timeline alignment.** Given the collections of events that logically constitute the same synchronous communication (stored in the `related_sync_op` attribute), we can exploit the fact that all participants in a synchronous call must *logically* finish at the same moment before proceeding. This provides anchor points for aligning different ranks’ timelines. We choose a reference rank (e.g. Rank 0) and iteratively align the other ranks to it, using the identified communication instances as calibration points. Because collectives occur frequently, the calibration points are dense, limiting clock-drift errors to the interval between two consecutive anchors and preventing long-term error accumulation. Under high-frequency communication patterns—such as in tensor parallelism or during intensive pipeline-stage exchanges—the alignment accuracy is correspondingly high.

**Anomaly analysis.** We devise a multi-stage heuristic algorithm to detect and localize down-clocked GPUs. The core insight is that the *true* fault source should appear as the slowest member in *every* synchronous group it joins, whereas ranks that merely suffer collateral slowdown will only lag because they are waiting for the faulty peer.

1. **Cross-data-parallel comparison of peer operations.** Data parallelism guarantees that ranks playing the same role—i.e. having identical PP and TP indices—execute identical sequences of compute kernels. If a rank’s execution time for a given kernel is significantly longer than that of its peer ranks, we mark that kernel instance as a *slow operation*. We then compute the proportion of slow operations per rank; ranks whose proportion exceeds a threshold become *candidate* ranks.
2. **Root-cause identification via collective synchronization.** We further analyze candidates within their TP and DP collectives. If a rank consistently *starts* collective calls noticeably later than its peers—because its preceding computation is slower—and this pattern repeats across many collectives, it is likely the genuine fault source rather than a victim.
3. **Root-cause identification via P2P latency.** Pipeline parallelism complicates matters: the start times of `Send/Recv` between adjacent stages naturally differ due to pipeline scheduling and micro-batch asynchrony, even when the system is healthy. Start-time comparison therefore cannot readily flag slow nodes. Instead, we leverage the payload size (tensor size) and the observed latency recorded for each P2P transfer to compute the *effective bandwidth*. A true slow node often exhibits degraded bandwidth, either because its PCIe-NIC data path is impaired or because it prepares/consumes data slowly, especially during the *warm-up* phase (1-forward-1-backward) where forward activations are sent downstream with few confounding interactions.

**Fast data retrieval.** To efficiently query and analyze bandwidth data we use *Perfetto SQL* [41]. Perfetto, developed by Google, supports Chrome Tracing Format and offers a built-in SQLite-like query engine. We can import the aligned trace into Perfetto and write SQL queries to extract communication events, compute their effective bandwidths, and perform statistical analyses.

## 4 MegaFBD: Heterogeneity-Aware Forward-Backward Decoupling

### 4.1 Motivation and Goals

In the three-dimensional parallel paradigm adopted by Megatron-LM—tensor, pipeline, and data parallelism—the forward and backward phases are, by default, scheduled on the same GPU. Such binding ignores their structural differences in resource usage: the forward pass has more network transfers, whereas the backward pass involves gradient computation, activation recomputation, and optimizer updates. Consequently, the two phases exhibit highly dissimilar memory footprints, communication patterns, FLOP distributions, and power profiles.

Problems of co-locating forward and backward include:

- **Resource contention:** overlapped memory/bandwidth requests from the forward and backward passes can block the critical path.
- **Scheduling entanglement:** activations cannot be released early in the forward pass because the backward pass follows immediately.
- **Heterogeneous-resource under-utilization:** when CPUs, NPUs, or other devices are available, they cannot be flexibly reused.

To overcome these problems, we propose four core strategies in MegaFBD:

- **Instance-level decoupled scheduling:** split the forward and backward phases into two logical processes, assign distinct ranks, and bind them to different resources to reduce coupling.
- **Heterogeneous resource mapping:** deploy the forward phase on devices with higher computation power.
- **Differential parallelism configuration:** assign a smaller parallel degree to the forward pass—leveraging lower GPU memory usage—to cut communication cost.
- **Thread-level coordination mechanism:** employ a communication coordinator to ensure necessary data synchronization between forward and backward, preventing deadlocks and redundant transfers.

Implemented as a low-intrusion enhancement atop vanilla Megatron-LM, MegaFBD adds a new dimension of resource scheduling, making it particularly suitable for heterogeneous clusters and resource-constrained environments.

## 4.2 Design

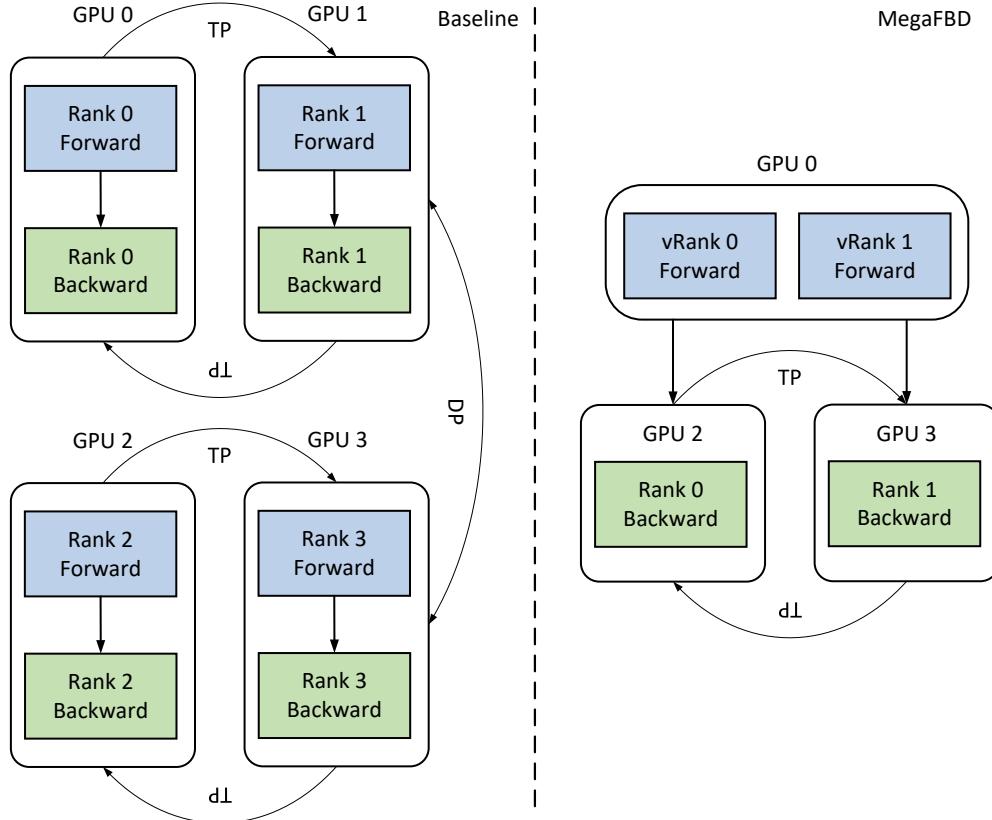


Figure 2: Forward/backward instance deployment of MegaFBD compared with that of existing training frameworks.

**Thread-level workers and virtual ranks.** To minimise changes to the original framework, MegaFBD keeps the fact that “the forward and backward phases use different parallel configurations” transparent to the training framework itself. Consequently, MegaFBD maintains *two* parallel naming systems as shown in Figure 2:

- **Virtual ranks:** every training thread owns a unique rank ID, and the forward and backward instances have the *same* number of ranks. Virtual ranks follow Megatron’s original allocation rules, ensuring that model partitioning and computation proceed according to the framework’s existing logic.
- **Physical ranks:** the rank IDs actually held by each GPU. Because the forward and backward phases may differ in their degrees of parallelism and compute capability, multiple training threads may reside on the *same* GPU. These intra-GPU threads need no heavy collective communication (they share the same device memory); any external communication is uniformly managed by the control thread of MegaFBD’s *communication coordinator*.

During initialization, physical ranks are created and assigned by the PyTorch launcher (process-level setup). Each process then spawns the required number of workers (thread-level setup), inheriting Megatron’s rank-assignment policy.

**Communication coordinator.** A common issue in multi-threaded collective communication is deadlock due to conflicting peer choices. Consider the scenario where Rank 0 wants to communicate with Rank 2, but Rank 2 intends to communicate with Rank 1. If we launch communications without verification, the control thread may start the Rank 0-2 transfer first; because Rank 1 shares the same control thread, its Rank 1-2 request is blocked, causing a deadlock.

The solution is to start a collective *only* after confirming that all ranks in the group have issued the *same* request. MegaFBD therefore introduces a **communication coordinator** that performs process-level synchronization and eliminates deadlocks.

When a worker thread needs to initiate a collective (e.g. all-reduce, broadcast, P2P), it first sends a request to its control thread. Control threads, which run permanently, exchange these requests within the relevant communication group. Once a control thread detects that *all* participants have posted the same request, it launches the operation.

The coordinator handles only *cross-GPU* communication. For each eligible communication group it maintains a bit-vector table of size (number of groups)  $\times$  (bit-vector length), where each bit vector maps one-to-one to all virtual ranks. The table is flattened into a 1-D tensor; different coordinators exchange this tensor via a global communication group, allowing them to determine how many members of each group are ready and to execute collectives in group order. The detailed steps are:

1. **Registration:** every thread marks each planned collective by setting the bit corresponding to its virtual rank in the group’s bit vector to 1.
2. **Alignment:** threads periodically perform an all-reduce with bitwise OR on the flattened vector.
3. **Readiness check:** each thread compares the current bit vector with the group’s *expected* value (e.g. with eight threads, the group {0,1,2,3} expects 11110000). If equal, the collective is ready.
4. **Ordered execution:** threads execute the ready collectives in ascending group order, avoiding contention and starvation.

This design exploits the three-dimensional parallel property of “serial within groups, parallel across groups.” Bit-vector compression reduces scheduling overhead to  $\mathcal{O}(G)$ , where  $G$  is the number of communication groups, and proves more efficient and scalable than traditional lock- or condition-variable-based thread synchronization schemes.

## 5 MegaDPP: Dynamic Pipeline Scheduling

### 5.1 Motivation and Goals

Pipeline parallelism is a key technique for boosting large-model training throughput: the model is partitioned into multiple sub-modules that execute concurrently on several GPUs in a pipeline fashion. The conventional *1F1B* schedule—one forward pass and one backward pass per iteration—minimizes memory footprint but exposes three critical drawbacks:

1. **Weight-update latency:** AllReduce for gradient synchronization can be issued only after *all* backward stages finish, delaying the optimizer step.
2. **Shrunken communication window:** forward outputs must be sent to the next stage immediately, leaving short overlapping windows between communication and subsequent computation.
3. **Rigid scheduling:** the fixed order lacks adaptability to disturbances such as link-bandwidth fluctuations or GPU down-clocking.

Prior studies have proposed various enhancements: *PipeDream-2BW* employs double buffering to shorten pipeline flushes; *ZB1P* compresses bubbles via an optimized schedule graph; *BitPipe* introduces bidirectional interleaving to raise throughput. However, all of them retain the 1F1B assumption, keeping the schedule rigid and ill-suited for dynamic environments.

MegaDPP tackles this challenge with an **elastic, adaptive pipeline-scheduling framework** that supports dynamic traversal orders and resource-aware decisions within the Megatron-LM ecosystem. Its key designs are:

- **Traversal-policy switching:** flexibly alternate between *depth-first* (advancing the same data through multiple model blocks) and *breadth-first* (advancing multiple data items through the same block) strategies.
- **Resource-aware scheduling:** prioritize tasks based on real-time memory usage and network-bandwidth conditions, hiding communication bottlenecks at low overhead.
- **Asynchronous-communication support:** a lightweight parallel scheduler plus an async P2P communication library fuse computation and transfer to maximize device utilization.

MegaDPP delivers a decisive breakthrough over traditional static schedules, granting training systems greater robustness and scalability under complex parallel settings.

## 5.2 Design

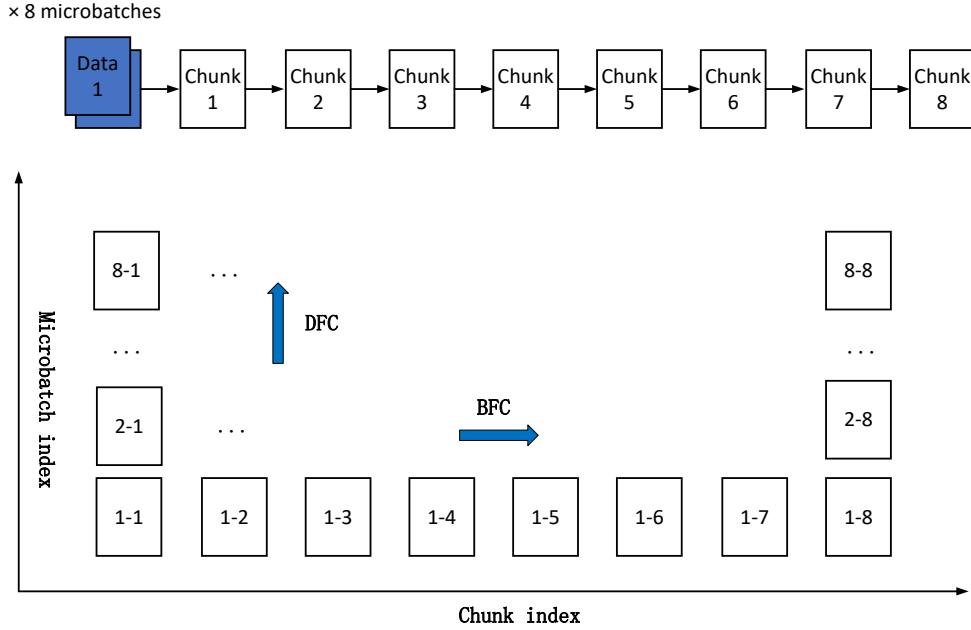


Figure 3: Training models with 8 chunks on 8 microbatches of data require 64 computation tasks per iteration.  $i-j$  denotes the task that processes the  $i$ -th microbatch on chunk  $j$ . MegaDPP scheduler arranges these tasks into an  $8 \times 8$  matrix and executes them successively along either the micro-batch (row) or model-chunk (column) dimensions.

**Parallel scheduler.** In pipeline parallelism each rank runs an independent *parallel scheduler* that decides the execution order of all compute and communication steps. If we view every compute task of one training iteration as a two-dimensional matrix indexed by (model\_chunk\_id, microbatch\_id)—that is, producing an output on a given model shard for a given data shard—then a schedule is simply a *traversal order* over this matrix. **MegaDPP** supports two traversal strategies:

- **Depth-First Computation (DFC):** process the *same* micro-batch across *different* model chunks first. This lets the corresponding backward pass start earlier, releases activation memory sooner, and lowers the GPU-memory peak.

- **Breadth-First Computation (BFC)**: process *different* micro-batches on the *same* model chunk first. Portions of a chunk thus finish all their computation sooner, enabling earlier gradient synchronisation, while significantly delaying downstream consumption of activation outputs—hence reducing network-communication pressure.

MegaDPP allows users to choose either scheme explicitly or to adopt BFC in a *best-effort* fashion as long as it does not cause out-of-memory (OOM) errors (Figure 3).

**Lightweight communication library.** We implement an intra-/inter-node P2P library atop shared memory and RDMA interfaces. Unlike mainstream NCCL, our API permits a single device to issue *concurrent, asynchronous* P2P transfers, so a compute thread can always pick the highest-priority ready input for execution. Concretely:

- **Four buffers** store, respectively, (i) forward tensors received from the network, (ii) forward outputs to be sent, (iii) backward tensors received, and (iv) backward outputs.
- **Two task queues** correspond to the sender and receiver. When the main thread invokes a communication call, the task is enqueued.
- **Dedicated worker threads** dequeue tasks, perform the actual send or receive, and copy tensors to the designated addresses.
- The **main thread** tracks completion of all queued tasks, thus always knows which concurrent transfers have finished.

For collective communications such as `all-reduce`, `all-gather`, and `reduce-scatter`, we continue to rely on NCCL.

## 6 MegaScope: Real-Time Interactive Visualization for LLM Interpretability

### 6.1 Motivation and Goals

Interpretability and training-process analysis for large language models are becoming research hot-spots. Yet once parameter counts exceed the hundreds-of-billions or even trillions, the training state space becomes enormous and highly dynamic, posing unprecedented challenges for visualization systems. On the one hand, researchers struggle to extract meaningful behaviours and patterns from surface-level metrics and to uncover the causal chain between parameter updates and performance evolution. On the other hand, directly logging and displaying internal states—activations, attention matrices, gradients—can incur significant I/O and communication overhead, which may severely harm training efficiency in distributed settings.

Mainstream tools reveal several limitations when applied to LLM visualization:

- **Fixed metric dimensions**: tools such as *TensorBoard* display only framework-defined metrics and hardly support user-defined training signals.
- **Tight coupling and manual export**: solutions like *BertViz* require users to insert extra code to dump tensors, breaking the training code structure; for large models the export cost is prohibitive.
- **Lack of interaction and injection**: most tools provide only static or streaming views, without the ability to inject intervention signals during training—hence no support for behavioural debugging or interpretability experiments.

Key features of MegaScope include:

- **Dynamic sampling and asynchronous processing**: users define observation points (layer / token / metric) via a registration API. The system captures the requested intermediate tensors on demand and caches them asynchronously, minimizing interference with the training path.
- **Hierarchical compression and smart caching**: activations and attention tensors are compressed online on the host side using aggregate statistics (e.g. max, mean, sparsity). Multi-level caching strategies control bandwidth and storage pressure.
- **Explorable interpretability UI**: the front-end presents multi-level, multi-granularity views—including attention heatmaps, representation trajectories, and token evolution—enabling full-chain insight from token behaviour to model mechanisms.

- **Perturbation injection and controlled experiments:** users can inject interventions at specific layers—such as replacing an attention matrix or adding perturbations—for rapid experiment design, enhancing research flexibility.

MegaScope greatly improves the observability of the training process and offers a powerful experimental tool for LLM architecture studies, robustness testing, and behavioural analysis.

## 6.2 Design

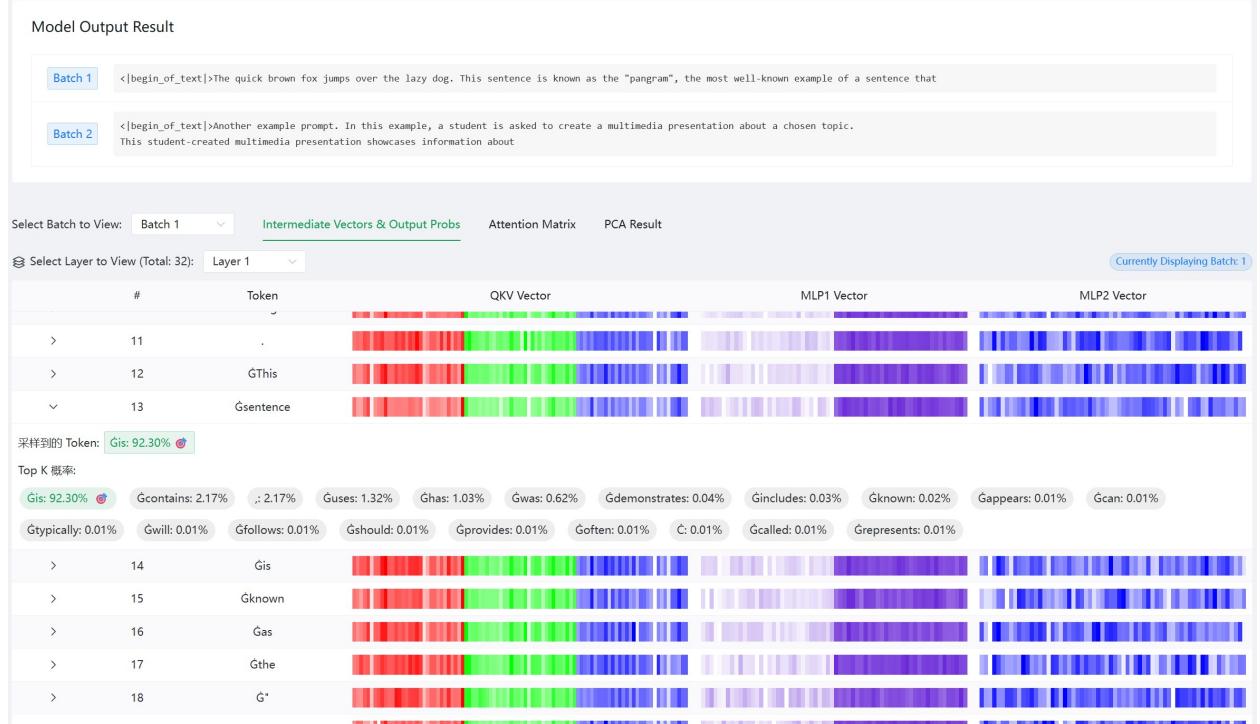


Figure 4: An example of visualizing text generation. MegaScope displays the visualization results token-by-token. In the first tab, the intermediate vector heatmaps are displayed and the output probabilities are shown in the expandable sections.

**Real-time generation and synchronized visualization.** After the user enters any prompt, the front-end displays the model’s decoding process in a *token-by-token refresh* mode while simultaneously presenting the back-end’s captured internal states (activations, attention maps, gradients, *etc.*), achieving a one-to-one correspondence between observed behaviour and underlying mechanisms. Figure 4 displays a visualization example of MegaScope.

**In-Depth analysis of intermediate results.** To highlight potential distribution drifts or outliers, MegaScope renders density heatmaps for crucial intermediate tensors such as the query, key, and value vectors  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$ , the outputs of the MLP sublayers, and the residual branches. For attention diagnostics, the system retrieves weights at both layer and head granularity, then displays their temporal evolution through interactive heatmaps and short animations; researchers can thus juxtapose attention patterns arising from different prompts (Figure 5). During every “predict-next-token” step, the interface shows the chosen token alongside its probability and overlays a top- $k$  bar chart that makes the entire decision distribution visually explicit.

**Interactive exploration tools.** A quick-access panel lets users pivot fluidly across layer, head, token, and batch dimensions; these selections are tied to a timeline component so that the decoding process can be replayed or scrubbed freely. High-dimensional hidden states are also projected onto two dimensions via PCA (Figure 6), and an interactive scatter plot traces each token’s trajectory, exposing geometric relationships between tokens and their originating prompts.

**Pluggable perturbation-injection framework.** Before critical parameter tensors are written back to memory, MegaScope can inject random bit flips or Gaussian noise, enabling systematic studies of how storage faults un-

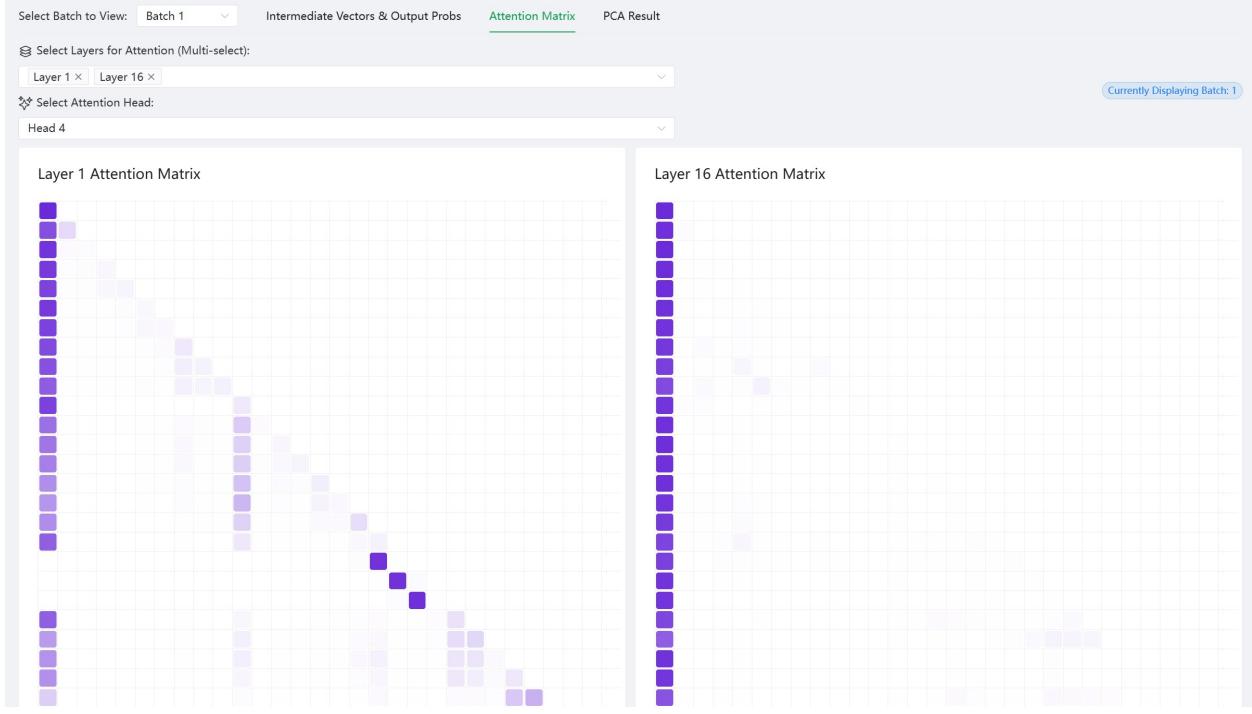


Figure 5: An example of visualizing attention scores. Users can select the layer and attention head they wish to inspect.

determine model robustness. During the forward pass, researchers may add noise fields or masking functions to the output of any chosen layer, making it straightforward to explore the behavioural impact of reduced numerical precision. At the system level, a constant offset can be introduced into the results transferred between layers, which emulates cross-device quantization errors or persistent link jitter and helps quantify the model’s resilience to communication anomalies.

## 7 Conclusion

In this work, we propose **MegatronApp**, a cohesive, production-ready toolchain that augments the training, debugging, and visualization capabilities of Megatron-LM at trillion-parameter scale. By integrating four purpose-built modules—**MegaScan**, **MegaFBD**, **MegaDPP**, and **MegaScope**—the framework transforms large-model development from a monolithic, opaque process into an analyzable, tunable, and highly efficient workflow.

1. **MegaScan**: Fine-grained performance tracing and slow-node detection with near-zero overhead, enabling practitioners to isolate system bottlenecks in minutes rather than hours.
2. **MegaFBD**: Decouples forward and backward passes to unlock heterogeneous resource utilization, reducing peak GPU memory and increasing overall throughput in mixed-CPU/GPU clusters.
3. **MegaDPP**: Introduces adaptive pipeline scheduling that reacts to runtime imbalances, flattening straggler effects and improving hardware utilization across deep pipeline stages.
4. **MegaScope**: Provides customizable metrics collection and real-time visualization, turning billions of training events into actionable insights through an intuitive dashboard.

Together, these components establish a new operational baseline for large-scale language-model training:

- **Higher efficiency**: Double-digit gains in throughput and cluster utilization.
- **Deeper observability**: Unified, operator-level visibility into computation and communication behavior, together with real-time and customized metrics collection on intermediate results.
- **Rapid diagnosis**: Automated anomaly detection and root-cause analysis that shorten mean time to recovery.



Figure 6: An example of visualizing the PCA dimensionality reduction feature. Users can visually inspect the clustering of tokens and understand how the model groups similar concepts. The displayed layer can also be selected.

- **Future-proof extensibility:** A modular, open-source architecture ready for emerging hardware and novel parallelism strategies.

Looking forward, we plan to extend MegatronApp with:

- Native support for fat failover after anomaly detection.
- New features and optimizations inherited from Megatron-LM.
- Additional tracing and diagnosis support for inference scenarios.

By lowering the barrier to efficient, transparent, and fault-tolerant training, **MegatronApp** positions itself as an indispensable companion for researchers and engineers pushing the frontier of large language models.

## References

- [1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [2] Wei Shi, Sihang Li, Tao Liang, Mingyang Wan, Guojun Ma, Xiang Wang, and Xiangnan He. Route sparse autoencoder to interpret large language models, 2025.
- [3] Luke Marks, Amir Abdullah, Clement Neo, Rauno Arike, David Krueger, Philip Torr, and Fazl Barez. Interpreting learned feedback patterns in large language models, 2024.
- [4] Hoagy Cunningham, Aidan Ewart, Logan Riggs, Robert Huben, and Lee Sharkey. Sparse autoencoders find highly interpretable features in language models, 2023.
- [5] David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Thomas McGrath, and Vladimir Mikulik. Tracr: Compiled transformers as a laboratory for interpretability, 2023.
- [6] Roland S. Zimmermann, David Klindt, and Wieland Brendel. Measuring per-unit interpretability at scale without humans. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. Paper ID 56ed2bd15b66f709cd81cb1aaa0496b9, official BibTeX pending.
- [7] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.
- [8] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [9] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengan Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- [10] Jesse Vig. Visualizing attention in transformer-based language representation models. *CoRR*, abs/1904.02679, 2019.
- [11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorBoard: Tensorflow’s visualization toolkit, 2015. Software available from tensorflow.org.
- [12] Andrew Chen, Andy Chow, Aaron Davidson, Arjun D’Cunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. Developments in MLflow: A system to accelerate the machine learning lifecycle. In *Proc. International Workshop on Data Management for End-to-End Machine Learning (DEEM)*, pages 1–4, 2020.
- [13] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, 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*, pages 1–15, 2021.
- [14] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.
- [15] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- [16] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, 2014.

- [17] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.
- [18] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.
- [19] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.
- [20] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.
- [21] Microsoft. DeepSpeed, 2024. <https://github.com/microsoft/DeepSpeed>.
- [22] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload : Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.
- [23] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.
- [24] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. Explainability for large language models: A survey. *arXiv preprint*, 2023.
- [25] Haoyan Luo and Lucia Specia. From understanding to utilization: A survey on explainability for large language models. *arXiv preprint*, 2024.
- [26] Neel Nanda, Collin Burns, Lawrence Chan, et al. A gentle introduction to mechanistic interpretability of neural networks. *arXiv preprint*, 2023.
- [27] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in gpt. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.
- [28] Kevin Meng, Max Nadeau, Alex Andonian, and David Bau. Mass editing memory in a transformer. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [29] Lei Guan, Dong-Sheng Li, Ji-Ye Liang, Wen-Jian Wang, Ke-Shi Ge, and Xi-Cheng Lu. Advances of pipeline model parallelism for deep learning training: An overview. *Journal of Computer Science and Technology*, 39(3):567–584, May 2024.
- [30] Xuan Peng, Xuanhua Shi, Haolin Zhang, Yunfei Zhao, and Xuehai Qian. Dawnpiper: A memory-scalable pipeline parallel training framework. *arXiv preprint*, 2025.
- [31] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024. Poster.
- [32] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, Xin Liu, Aurojit Panda, and Jinyang Li. Understanding stragglers in large model training using what-if analysis. *arXiv preprint*, 2025.
- [33] Aswathy Ravikumar and Harini Sriraman. DPro-SM: A distributed framework for proactive straggler mitigation using LSTM. *Heliyon*, 10(1):e23567, 2024.
- [34] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. Greyhound: Hunting fail-slows in hybrid-parallel training at scale. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC '25)*, pages 731–747, 2025.
- [35] Ran Yan, Youhe Jiang, Wangcheng Tao, Xiaonan Nie, Bin Cui, and Binhang Yuan. FlashFlex: Accommodating large language model training over heterogeneous environment. *arXiv preprint*, 2024.
- [36] Chris Mellor. NVIDIA CUDA gets RISC-V support. [https://www.theregister.com/2025/07/21/nvidia\\_cuda\\_riscv/](https://www.theregister.com/2025/07/21/nvidia_cuda_riscv/), 2025. Accessed: 2025-07-23.
- [37] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.
- [38] NVIDIA Corporation. NVIDIA Data Center GPU Manager (DCGM). <https://developer.nvidia.com/dcgm>, 2024. Version 3.x, Accessed: 2025-07-25.

- [39] NVIDIA Corporation. *CUDA Toolkit Documentation*. NVIDIA, 2024. Version 12.x, Accessed: 2025-07-25.
- [40] Google Inc. Chrome Tracing. <https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/>, 2013. Accessed: 2025-07-25.
- [41] Google Inc. Perfetto Open-Source Tracing Project. <https://perfetto.dev>, 2024. Accessed: 2025-07-25.