CANZONA: A UNIFIED, ASYNCHRONOUS, AND LOAD-BALANCED FRAMEWORK FOR DISTRIBUTED MATRIX-BASED OPTIMIZERS

Anonymous authors
Paper under double-blind review

ABSTRACT

The scaling of Large Language Models (LLMs) drives interest in matrix-based optimizers (e.g., Shampoo, Muon, SOAP) for their convergence efficiency; yet their requirement for holistic updates conflicts with the tensor fragmentation in distributed frameworks like Megatron. Existing solutions are suboptimal: synchronous approaches suffer from computational redundancy, while layer-wise partitioning fails to reconcile this conflict without violating communication constraints. We propose Canzona, a Unified, Asynchronous, and Load-Balanced framework that decouples logical optimizer assignment from physical parameter distribution. For Data Parallelism, we introduce an α-Balanced Static Partitioning strategy that respects atomicity while neutralizing the load imbalance. For Tensor Parallelism, we design an Asynchronous-Compute pipeline utilizing Micro-Group-Scheduling to batch fragmented updates and hide reconstruction overhead. Experiments on the Qwen3 family (up to 32B parameters) on 256 GPUs demonstrate that our approach preserves the efficiency of established parallel architectures, achieving a 1.57$\times$ speedup in end-to-end iteration time and reducing optimizer latency by 5.8$\times$.

Keywords: llms, optimizer, distributed training

1 INTRODUCTION

The rapid scaling of large language models (LLMs) (Yang et al., 2025b;a; Grattafiori et al., 2024; Guo et al., 2025; Kaplan et al., 2020; Hoffmann et al., 2022) increases the demand for more efficient training algorithms. While element-wise optimizers such as AdamW (Loshchilov & Hutter, 2017) and SGD (Ruder, 2016) remain the standard choice, recent studies (Liu et al., 2025; Anil et al., 2020; Osawa et al., 2019; Wen et al., 2025) suggest that **matrix-based optimizers**, including Shampoo (Gupta et al., 2018), SOAP (Vyas et al., 2024), and Muon (Jordan et al., 2024), can offer better convergence efficiency. Unlike element-wise methods, which update each parameter independently, these optimizers operate on whole tensors or matrices and exploit their structure through matrix operators such as SVD (Golub & Van Loan, 2013) or Newton-Schulz iterations (Higham, 1997).

Improved convergence efficiency does not by itself guarantee faster distributed training. The cost of each matrix update, the communication needed to make its inputs available, and the time spent waiting for other ranks all contribute to end-to-end iteration time. A practical execution strategy must address these system costs while preserving the optimizer's update rule.

The first challenge is reconciling whole-tensor updates with sharded training frameworks such as Megatron (Shoeybi et al., 2019). ZeRO-1-style data parallelism (DP) (Rajbhandari et al., 2020; Zhao et al., 2023) partitions optimizer states and gradient buffers across ranks, while tensor parallelism (TP) partitions individual parameter tensors. These layouts support element-wise updates, but can fragment the tensors required by a matrix operator. We refer to the requirement for complete tensor-level inputs as the **Atomicity Constraint**. In DP, satisfying this constraint must also respect the physical bucket layout used for coalesced collective communication. In TP, complete matrix updates must be reconstructed from shards. Section 2 and Appendix A provide the background.

Existing approaches resolve this conflict with significant trade-offs. Replicating optimizer states or reconstructing tensors on every rank preserves whole-tensor updates but repeats expensive matrix

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

computations. Assigning whole layers to DP optimizer owners avoids state replication, yet can misalign optimizer ownership with Megatron's bucket layout. The resulting mismatch sacrifices efficient bucket-based `Reduce-Scatter` and forward–backward communication overlap, so savings in optimizer computation can be offset by communication overhead. Other approaches change the optimization procedure through shard-local or approximate updates (Khaled et al., 2025; Ahn & Xu, 2025). Such modifications may change training behavior and are tied to particular optimizers. These trade-offs motivate a general execution framework that preserves the original optimizer while retaining the communication efficiency of the training system.

We introduce **Canzona**, a unified, asynchronous, and load-balanced framework that *decouples logical optimizer ownership from physical parameter placement*. Canzona distinguishes the rank responsible for a tensor's matrix update from the ranks holding its parameter replicas or shards. Canzona treats each tensor update as a task assigned to an optimizer owner, while retaining the distributed parameter layout required by the model. The execution mechanism follows the communication constraints of each parallelism dimension, without modifying the optimizer's internal mathematics.

For DP, which can span bandwidth-constrained inter-node links, Canzona uses static partitioning to align whole-tensor optimizer ownership with the bucket geometry. Each owner receives complete local gradients and performs its assigned matrix updates without additional DP communication during the optimizer step. Variable-size bucket collectives preserve communication–computation overlap in the forward and backward passes. This design retains the efficient training pipeline while avoiding replicated optimizer work; preserving atomicity does not require falling back to redundant execution.

For TP, typically deployed within a node, high-bandwidth interconnects make reconstruction of tensor-level updates viable. Canzona assigns each update to a Host Rank, keeps its optimizer states there, and routes gradient shards to that host. Hosts compute different matrix updates asynchronously and return update shards to the original parameter owners. Batching these tasks into micro-groups fuses communication, amortizing small-message overhead without gathering all model gradients at once. Only gradients and update tensors are exchanged during this workflow.

Restoring atomicity alone, however, is not enough. Tensor sizes and matrix-operation costs vary, so naive atomic assignment can concentrate work and optimizer-state memory on a few ranks. The slowest ranks then determine completion time while others wait. Canzona addresses this second challenge with two offline scheduling strategies: α-Balanced Greedy LPT (Graham, 1969) balances DP ownership while trading off global load balance against per-bucket communication uniformity; Micro-Group Scheduling balances TP tasks subject to a per-rank capacity limit. The resulting plans are reused throughout training, avoiding repeated scheduling decisions and preserving the original tensor-level update rules.

Our contributions are:

- **Problem formulation.** We identify and formalize the joint atomicity and communication-layout constraints that limit exact distributed execution of matrix-based optimizers.
- **Execution framework.** We develop an optimizer-agnostic framework that reconciles these constraints through decoupled ownership and load-balanced scheduling across DP and TP.
- **Evaluation.** Experiments on the Qwen3 family (1.7B–32B) with Muon, Shampoo, and SOAP demonstrate Canzona's efficiency and generality. The main 256-GPU comparison against `layerwise_optimizer` achieves a 1.57× end-to-end speedup and a 5.8× optimizer-step speedup.

2 Preliminary

2.1 Data Parallelism and ZeRO

Data parallelism (DP) distributes training examples across ranks. Standard DDP replicates model parameters and optimizer states (Li et al., 2020), whereas ZeRO-1 shards optimizer states (Rajbhandari et al., 2020). Megatron (Shoeybi et al., 2019) packs parameters and gradients into contiguous buckets and assigns rank r the interval $[(r-1)S, rS)$, with $S = |B|/R$ for bucket B and R ranks. This positional mapping enables coalesced `Reduce-Scatter`/`All-Gather` but can split individual tensors. We call the layout requirement the **ZeRO-1 Geometric Constraint**. Hereafter, DP refers

to this ZeRO-1 setting; buffer organization and the standard training workflow are detailed in Appendix A.1.

2.2 Tensor Parallelism

Tensor parallelism (TP) partitions individual parameters, gradients, and optimizer states across ranks. It is orthogonal to DP and is typically deployed within a node to exploit high-bandwidth interconnects.

2.3 Matrix-based Optimizers

Element-wise optimizers such as AdamW (Loshchilov & Hutter, 2017) can update these shards independently. Matrix-based optimizers such as Muon (Jordan et al., 2024) and Shampoo (Gupta et al., 2018) instead operate on whole tensors. For a parameter $W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$, their update can be written as

$$W_{t+1} = W_t - \eta \cdot \text{MatrixOp}(W_t, \nabla W_t) \tag{1}$$

Such operations impose an **Atomicity Constraint**: the complete tensor must be available for its matrix update. A bucket cut through a tensor therefore prevents a shard-local matrix update. DP must preserve tensor boundaries while retaining bucket-compatible communication (Section 3); TP must reconstruct tensor-level updates from shards (Section 4).

3 Load-Balanced Asynchronous Compute for Data Parallelism Matrix-based Optimizer

In DP, Canzona uses static, tensor-aligned ownership to execute matrix updates locally without additional optimizer-step collectives, while preserving Megatron's bucketed communication. Figure 1 contrasts this design with redundant and unbalanced execution. TP reconstruction and scheduling are addressed in Section 4.

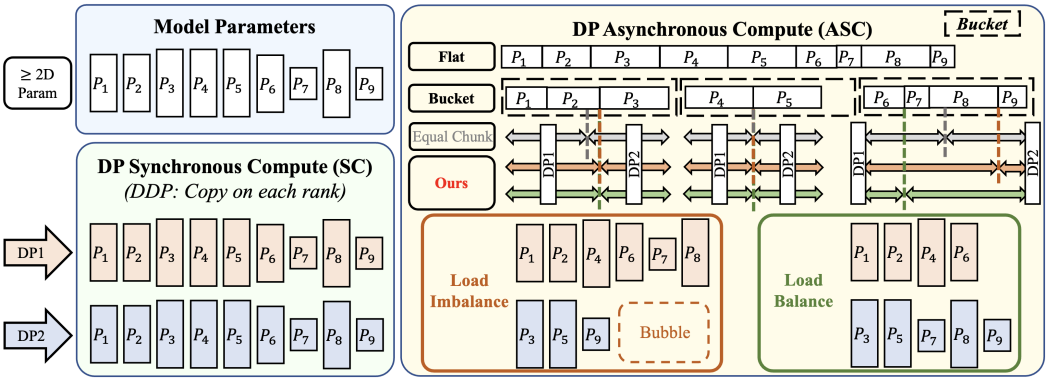


Figure 1: **DP partitioning strategies. Left (DP-SC):** Replicated optimizer states lead to redundant computation. **Right (DP-ASC):** Partitioned states eliminate redundancy. Gray dashed cuts show equal ZeRO-1 chunks, which can split tensors. Orange arrows, lines, and boxes mark unbalanced atomic ownership and the resulting computation bubbles; green marks our α-balanced layout. The dashed box highlights idle time. Blocks labeled P denote optimizer states and associated update computation; model parameters remain replicated during forward and backward passes.

3.1 Design Paradigm Analysis: Why Static Layout?

Standard ZeRO-1 can split tensors at bucket shard boundaries, violating the Atomicity Constraint (Section 2.3). We compare three ways to preserve whole-tensor updates.

Paradigm 1: Synchronous/Redundant Compute (SC). Replicating optimizer states as in DDP preserves atomicity, but every rank performs the same matrix operations, limiting scalability.

Paradigm 2: Layerwise Partitioning. NVIDIA's `layerwise_optimizer`[1] assigns whole layers to ranks, preserving atomicity but misaligning task ownership with Megatron's physical bucket layout. This **Geometric Incompatibility** prevents bucket-coalesced `Reduce-Scatter`, requiring gradient `All-Reduce` or per-parameter communication, plus redistribution of updated weights (Appendix D.2, Figure 16). In the All-Reduce fallback, gradient synchronization transfers twice the volume of Reduce-Scatter; redistributing updated weights adds exposed optimizer-step communication. Alternatively, routing tensors individually loses bucket coalescing and increases kernel-launch overhead. These costs explain why balancing layer assignments alone does not recover the efficient forward–backward pipeline.

Paradigm 3: Static Partitioning (Ours). We anchor ownership to physical buffer position. Given stride $S = |B|/R$, rank r owns the complete optimizer states of parameter p if and only if:

$$(r-1) \cdot S \leq \text{Start_Index}(p) < r \cdot S \tag{2}$$

This **Atomic Ownership Rule** keeps each state tensor intact and preserves sequential rank ordering, satisfying the ZeRO-1 geometric constraints needed for coalesced `Reduce-Scatter`. Appendix D.2 details the alignment requirement.

Remaining challenge: load imbalance. Unequal tensor costs can still create computational stragglers and memory peaks under atomic ownership. We therefore optimize bucket boundaries to balance rank loads while retaining physical parameter order. Per-bucket communication imbalance can be hidden by forward/backward overlap (Appendix C.5).

3.2 Load-Balance Optimization

Consider a distributed training setting with R ranks and a `param_and_grad_buffer` chunked into N logical buckets $\mathcal{B} = \{B_1, \ldots, B_N\}$. Each bucket B_i consists of an ordered sequence of non-splitable parameters P_i. We assign a function $\mathcal{W}(p)$ (Typically $\mathcal{W}(p) = numel(p)$) to each parameter p, representing its primary resource cost (e.g., memory usage, computation, or communication). Note that while we assume linear costs here for simplicity, we provide a generalized explanation for nonlinear optimizer costs in Appendix D.5.

The goal is to determine slicing vectors $\mathbf{s}_i$ for each bucket B_i. The cumulative load assigned to rank r for bucket B_i is $L_{i,r} = \sum_{p \in \text{range}(s_{i,r-1}, s_{i,r})} \mathcal{W}(p)$. We aim to optimize two objectives:

(1) Global DP Balance (Minimize Stragglers): Minimize the maximum deviation from the ideal mean load μ_{load}:

$$\mathcal{J}_{\text{DP}} = \max_r \left| \sum_{i=1}^{N} L_{i,r} - \mu_{\text{load}} \right| \tag{3}$$

(2) Fwd-Bwd Bucket Communication Balance (Minimize Idle Time): Ensure data volume within each bucket is distributed evenly:

$$\mathcal{J}_{\text{Comm}} = \sum_{i=1}^{N} \sum_{r=1}^{R} \left| S_{i,r} - \frac{|B_i|}{R} \right| \tag{4}$$

Crucially, those optimization objectives respect the ZeRO-1 Geometric Constraints (Section 3.1) by **shifting the slice boundaries** $s_{i,r}$ within the bucket. Because the parameters are never physically reordered, the sequential, monotonic physical ordering is preserved. This "boundary-shifting" mechanism serves as the logical bridge that allows us to neutralize load imbalance while retaining the capability to launch optimal, coalesced `Reduce-Scatter` primitives.

Optimizing this discrete partitioning problem under strict atomicity constraints is NP-hard. Our α-Balanced Greedy LPT heuristic processes buckets in descending order of total load, handling heavy buckets first (Algorithm 1; full pseudocode in Appendix D.3). For each bucket, the target allocation is $\mathbf{v}^* = (1-\alpha)\mathbf{v}_{\text{even}} + \alpha \mathbf{v}_{\text{fill}}$, where $\mathbf{v}_{\text{even}} = [1/R, \ldots, 1/R]$ and $\mathbf{v}_{\text{fill}}$ normalizes the deficits $\max(0, \mu_{\text{load}} - L_r)$, with L_r denoting accumulated rank load (falling back to $\mathbf{v}_{\text{even}}$ when all deficits are zero). The algorithm places each cut at the parameter boundary whose cumulative load

[1] https://github.com/NVIDIA/Megatron-LM/pull/2241

is closest to the corresponding cumulative target, then updates the accumulated rank loads. Thus, $\alpha \to 1$ prioritizes global load balance by filling underloaded ranks, while $\alpha \to 0$ prioritizes uniform per-bucket communication volume. Both settings preserve tensor atomicity and physical parameter order.

3.3 System Workflow: Static-Layout Enforcement

Offline planning. The α-Balanced Greedy LPT algorithm constructs a partition map Π assigning rank r the tensor-aligned interval $[s_{i,r-1}, s_{i,r})$ of bucket B_i. We replace uniform shard registration with allocation proportional to the assigned volume $S_{i,r}$, matching physical storage to this plan.

Runtime. During backward, variable-size `Reduce-Scatter` sends each rank its gradient interval according to Π and overlaps with subsequent backward computation. Each rank then performs matrix updates using complete local parameters, gradients, and optimizer states, with no additional optimizer-step communication. Variable-size `All-Gather` reconstructs the updated parameter buckets for forward execution and overlaps with preceding forward computation. Both collectives use the planned shard sizes, preserving bucket-level communication and overlap despite unequal allocations.

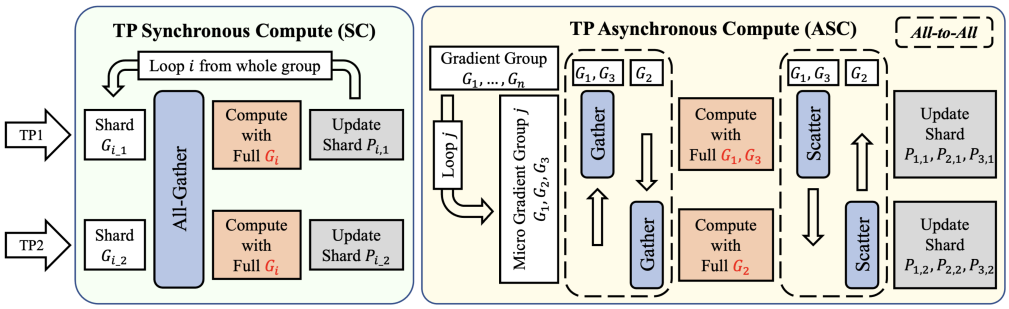


Figure 2: **TP optimizer-update workflows. Left (TP-SC):** All-Gather reconstructs each tensor on every rank, causing redundant computation. **Right (TP-ASC):** Gradients G are batched into micro groups; dashed boxes mark fused All-to-All communication. Offline scheduling assigns tensor updates to Host Ranks, which compute them asynchronously and return update shards. Different compute-block lengths represent varying tensor costs, balanced across ranks by the scheduler.

4 Tensor Parallelism with Load-Balanced Asynchronous Compute

While our DP strategy (Section 3) strictly enforces a zero-communication layout during optimizer step to protect the global network, Intra-node Tensor Parallelism (TP) presents a different challenge. Figure 2 compares the workflow of standard TP and our proposed strategy. In the standard **TP Synchronous Compute (TP-SC)** (Figure 2 Left, NVIDIA's `layerwise_optimizer` implementation), every rank must communicate and compute the full tensor for every parameter. This results in the *Redundant Compute* highlighted in the figure, where all devices perform identical operations (e.g., "Compute with Full G_1") regardless of the tensor ownership.

In contrast, our **TP Asynchronous Compute (TP-ASC)** (Figure 2 Right) eliminates this redundancy through a statically planned asynchronous computation pipeline. As shown in the "Micro Gradient Group j" block, we batch multiple tensors (e.g., G_1, G_2, G_3) into a single fused communication group. Crucially, our offline algorithm (Algorithm 2) assigns these tensors to specific Host Ranks to equalize the load $\mathcal{W}$. We emphasize that this reconstruction communication is viable specifically because TP typically operates within the high-bandwidth *Intra-node* domain (e.g., via NVLink). This stands in sharp contrast to DP, which spans bandwidth-constrained *Inter-node* connections, thereby necessitating the strict zero-communication strategy we enforced in Section 3.

4.1 Task Abstraction: The Asynchronous Compute Unit

Each TP-split parameter defines an atomic update task assigned offline to a Host Rank. Its complete optimizer states are initialized and retained there. Since the update is computed from gradients and states, only gradients and update tensors need communication; weights remain on their original ranks. Each micro group follows three stages (Figure 2):

(1) Gather gradients. A fused All-to-All routes gradient shards to their designated hosts. Grouping amortizes small-message overhead while avoiding the memory peak of gathering all model gradients at once. Group size therefore mediates a communication–memory trade-off: small groups incur frequent collective launches, whereas larger groups improve bandwidth utilization but require larger temporary buffers. The capacity constraint in Section 4.2 controls this trade-off, with empirical sensitivity reported in Appendix C.6. **(2) Compute updates.** Hosts asynchronously execute matrix operations on complete gradients and local states, e.g., $\Delta W = \text{Muon}_{\text{step}}(G, \text{States}_{local})$. **(3) Scatter and apply updates.** A second fused All-to-All returns update shards to the original parameter owners, which apply $P_{i,local} \leftarrow P_{i,local} + \Delta W_{i,local}$. Neither model weights nor optimizer states are transmitted during this workflow.

4.2 Workload Scheduling: Hierarchical Partitioning

While Figure 2 illustrates the example ideal execution flow for a single instance (Micro Group j), the system-level challenge lies in generalizing this balance across the entire model. We must rigorously partition the full parameter set $\mathcal{P}$ into a sequence of such Micro Groups $\mathbb{M} = \{M_1, \ldots, M_K\}$ and determine the optimal Host Rank assignments to minimize the overall execution time. We formulate this as a hierarchical partitioning problem with **Lexicographical Objectives**:

Priority 1: Minimize Computational Imbalance. Within each micro group M_k, we minimize the makespan across the R ranks. Let $\mathcal{L}_{k,r}$ be the load on rank r for group M_k:

$$\min \Phi_1(M_k) = \max_r(\mathcal{L}_{k,r}) - \min_r(\mathcal{L}_{k,r})$$

Priority 2: Maximize Group Saturation. To reduce the total number of communication steps, we pack as much load as possible into each micro group without violating the memory/buffer capacity C_{max}:

$$\max \Phi_2(M_k) = \sum_{r=1}^{R} \mathcal{L}_{k,r}, \quad \text{s.t. } \max_r(\mathcal{L}_{k,r}) \leq C_{max}$$

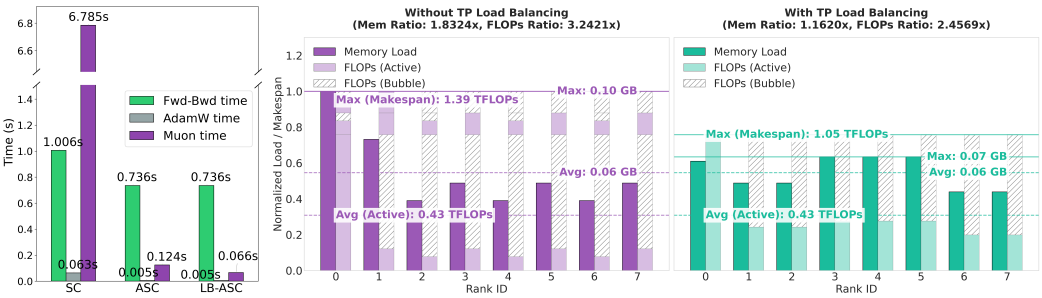
Algorithm: Global LPT with Greedy Rollback. We propose a deterministic heuristic (Algorithm 2; full pseudocode in Appendix D.3) to solve this optimization problem offline. First, we use a **Global Sort** based on computational cost (LPT rule) to ensure determinism across ranks and prioritize heavy tasks (which are harder to balance). Second, we employ a **Greedy Packing with Rollback** strategy. We iteratively add parameters to a current micro group and simulate LPT assignment using a Min-Heap, assigning each tensor to the currently least-loaded rank. If adding a parameter violates the C_{max} constraint (indicating the makespan or memory has exceeded limits), we trigger a *Rollback*: the current group is finalized, and the parameter initiates a new group.

Note that, similar to Section 3.2, while we assume a simplified linear cost model $\mathcal{W}(p) = \text{numel}(p)$ in this section for clarity, our framework formulation fully supports the generalized nonlinear complexity (detailed in Appendix D.5) inherent to optimizers like Muon and Shampoo.

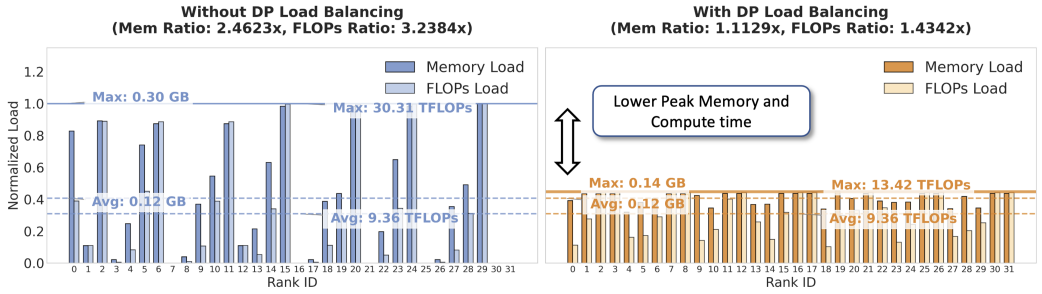
At runtime, the system iterates over the precomputed groups $\mathbb{M}$ and executes the gather–compute–scatter workflow in Section 4.1, without repeating scheduling decisions.

5 Experiment

We evaluate Canzona on up to 512 GPUs, measuring end-to-end efficiency, load balance, and preservation of optimizer training behavior. Appendix C provides additional baseline comparisons, scaling experiments, Shampoo/SOAP results, and scheduling ablations.



(a) Efficiency Comparison (b) Tensor Parallelism Load Balancing Analysis



(c) Data Parallelism Load Balancing Analysis

Figure 3: **Main Results: (a) Efficiency Comparison:** Our LB-ASC strategy outperforms baselines by effectively eliminating computational bubbles and maximizing device utilization. **(b) & (c) Load Balancing Analysis:** The visualized load distributions demonstrate that our proposed scheduling algorithms successfully flatten the workload variance for both Tensor Parallelism (b) and Data Parallelism (c), significantly mitigating the straggler problem compared to naive partitioning.

5.1 Experiment Setup

Models and optimizers We evaluate Canzona on the **Qwen3** (Yang et al., 2025a) model family, ranging from 1.7B to 32B parameters. Unless otherwise specified, our main analysis uses **Qwen3-32B** with the **Muon** (Jordan et al., 2024) optimizer. To verify optimizer generality, we further evaluate **Shampoo** (Gupta et al., 2018) and **SOAP** (Vyas et al., 2024) (Appendix C.4). The default training configuration uses sequence length 4096 and batch size 1 per DP rank.

Baselines We compare four strategies: **SC**, a synchronous/redundant baseline; **NV-layerwise**, NVIDIA's `layerwise_optimizer`; **ASC**, our asynchronous architecture without load-aware scheduling; and **LB-ASC**, the full Canzona design with both decoupled execution and load balancing.

Metrics We report **runtime breakdown** and **load-balance ratio**. Runtime breakdown includes the forward–backward time and target optimizer-step latency. Load-balance ratio is defined as the max-over-average value across ranks for either peak memory or computational FLOPs, where values closer to 1 indicate better balance. Full experimental details are provided in Appendix C.1.

5.2 Main Results

All experiments in this section are conducted on a large-scale cluster of 256 GPUs. The distributed topology is configured with a DP size of 32 and a TP size of 8.

Effectiveness of Load Balancing Strategies (Figure 3). We analyze how our proposed strategies mitigate the severe load imbalance inherent in matrix-based optimizers. For Data Parallelism (Figure 3c), standard static partitioning ("Without DP Load Balancing") results in extreme stragglers, exhibiting a FLOPs imbalance ratio (Max/Avg) of **3.24**$\times$ and a Memory imbalance ratio of **2.46**$\times$. In contrast, our α-Balanced Partitioning drastically reduces these ratios to **1.43**$\times$ and **1.11**$\times$, effectively

flattening the workload distribution. For Tensor Parallelism (Figure 3b), the naive baseline similarly suffers from high variance (FLOPs Ratio: 3.24×). By introducing our Micro-Group Scheduling, we improve the FLOPs ratio to **2.46**× and the Memory ratio to **1.16**×. Crucially, as shown in the Makespan comparison (Figure 3a), this improved balance directly translates to efficiency: LB-ASC achieves the lowest maximum step time (1.05 TFLOPS equivalent) compared to SC and ASC, effectively eliminating the computation bubbles.

Comparison with `layerwise_optimizer` (Figure 4). We compare the end-to-end training iteration time against NVIDIA's `layerwise_optimizer` ("NV-layerwise"), the current best practice for preserving atomicity. As shown in Figure 4, our approach achieves a **1.57**× **speedup** in total iteration time (0.877s vs. 1.381s). This performance gain stems from two critical sources: (1) **Drastic Reduction in Optimizer Latency (5.8**× **Speedup):** Our optimizer step time drops from **0.383s** to **0.066s**. While NV-layerwise necessitates extra runtime `Broadcast` or `All-Gather` to redistribute updated parameters, our zero-communication DP strategy and asynchronous TP pipeline effectively hide the optimization overhead. (2) **Improved Fwd-Bwd Efficiency (1.23**× **Speedup):** Our method also reduces the Forward-Backward pass time from **0.998s** to **0.811s**. This highlights a structural advantage: NV-layerwise suffers from the geometric conflict described in Section 3.1, which structurally forces it to rely on gradient All-Reduce (incurring 2× communication volume), whereas our approach utilizes the more efficient, bucket-overlapped `Reduce-Scatter`.

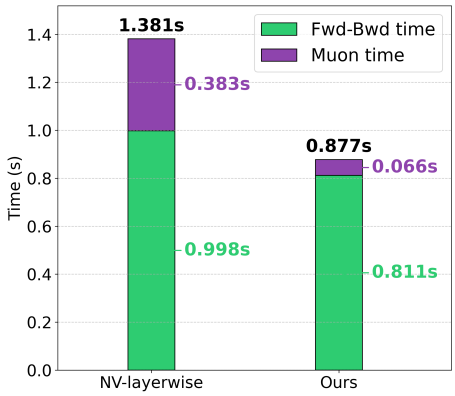


Figure 4: **End-to-End Iteration Time Comparison:** Our framework significantly outperforms `layerwise_optimizer`. The performance advantage is driven by two factors: (1) the elimination of runtime communication during the optimizer step via our decoupled design, and (2) the preservation of the ZeRO-1 Geometric Constraint mentioned in Section 3.1.

5.3 Precision Verification

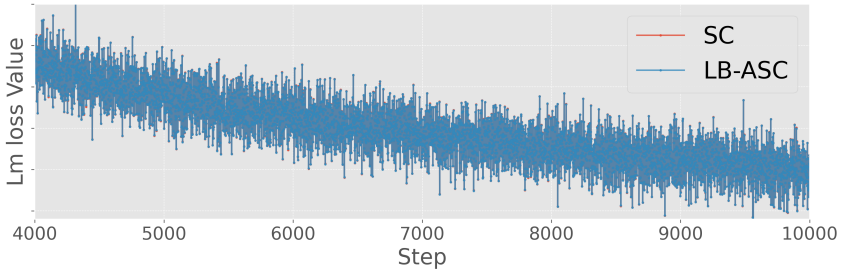


Figure 5: **Precision Verification:** SC Baseline & LB-ASC (Ours)

To ensure algorithmic correctness, we trained a Qwen3-1.7B model on 400B tokens using the Muon optimizer (DP=8, TP=4). As shown in Figure 5, the training loss trajectory of our strategy LB-ASC is indistinguishable from the standard synchronous baseline (SC). This confirms that our decoupled partitioning and scheduling strategies act as purely system-level optimizations, preserving the exact convergence behavior of the original matrix-based optimizer. More precision verification experiments can be found in Appendix C.4

6 Related Work

Existing efforts on distributed matrix-based optimization can be broadly grouped into two categories. The first category preserves the original optimizer mathematically, but compromises system efficiency; the second improves efficiency by modifying the optimizer itself, often at the cost of fidelity or generality.

System-level exact approaches. Recent industrial and distributed implementations, such as layer-wise partitioning strategies and distributed Shampoo-style systems (Shi et al., 2023), preserve tensor atomicity by assigning optimizer work at a coarse granularity, e.g., whole layers. While mathematically exact, these approaches can conflict with the bucketed geometric layout assumed by ZeRO-style communication, making it difficult to retain efficient `Reduce-Scatter`-based execution and overlap communication with computation. As a result, they often preserve correctness at the cost of system throughput.

Algorithmic approximations. Another line of work modifies the optimizer itself to better fit distributed constraints. Representative examples include block-diagonal approximations in distributed second-order methods (Anil et al., 2020; Osawa et al., 2019), shard-local orthogonalization for Muon-style updates (Khaled et al., 2025), and low-rank or subspace approximations (Ahn & Xu, 2025; Gong et al., 2025). These methods can significantly reduce communication and memory overhead, but they alter the original optimization procedure and may sacrifice either convergence fidelity or cross-optimizer generality.

Our position. Canzona differs from both lines of work. Unlike system-level exact methods, it preserves efficient distributed execution by reconciling tensor atomicity with Megatron's DP and TP execution structure. Unlike algorithm-specific approximations, it does not alter the optimizer mathematics. Instead, Canzona provides a unified, optimizer-agnostic system abstraction for executing tensor-level updates efficiently in distributed training.

Please refer to Appendix E for a comprehensive review.

7 Conclusion

We present a unified distributed optimization framework that resolves the fundamental conflict between matrix-based optimizers and modern parallel training strategies. By introducing the α-Balanced Static Partitioning for DP and the Asynchronous Micro-Group Scheduling for TP, we successfully reconcile strict atomicity requirements with massive parallelism. Our extensive experiments on the Qwen3 model family demonstrate that our approach not only eliminates the heavy communication overhead of existing solutions but also effectively neutralizes the load imbalance inherent in second-order optimization, achieving a 1.57$\times$ end-to-end speedup.

AI use statement

We used generative AI tools solely for manuscript editing and polishing, including language refinement, organization, and LaTeX formatting. These tools were not used to develop the research methodology, implement the proposed methods, conduct experiments, analyze results, or formulate scientific claims. The authors take full responsibility for the final content of this paper.

References

Kwangjun Ahn and Byron Xu. Dion: A communication-efficient optimizer for large models. *arXiv e-prints*, pp. arXiv–2504, 2025. doi: 10.48550/arXiv.2504.05295. URL https://doi.org/10.48550/arXiv.2504.05295.

Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020. doi: 10.48550/arXiv.2002.09018. URL https://doi.org/10.48550/arXiv.2002.09018.

Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013. doi: 10.1137/1.9781421407944. URL https://doi.org/10.1137/1.9781421407944.

Wenbo Gong, Meyer Scetbon, Chao Ma, and Edward Meeds. Towards efficient optimizer design for LLM via structured fisher approximation with a low-rank extension. *arXiv preprint arXiv:2502.07752*, 2025. doi: 10.48550/arXiv.2502.07752. URL https://doi.org/10.48550/arXiv.2502.07752.

Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969. doi: 10.1137/0117039. URL https://doi.org/10.1137/0117039.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. doi: 10.48550/arXiv.2407.21783. URL https://doi.org/10.48550/arXiv.2407.21783.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. doi: 10.48550/arXiv.2501.12948. URL https://doi.org/10.48550/arXiv.2501.12948.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pp. 1842–1850. PMLR, 2018.

Wei He, Kai Han, Hang Zhou, Hanting Chen, Zhicheng Liu, Xinghao Chen, and Yunhe Wang. Root: Robust orthogonalized optimizer for neural network training. *arXiv preprint arXiv:2511.20626*, 2025. doi: 10.48550/arXiv.2511.20626. URL https://doi.org/10.48550/arXiv.2511.20626.

Nicholas J Higham. Stable iterations for the matrix square root. *Numerical Algorithms*, 15(2): 227–242, 1997. doi: 10.1023/A:1019150005407. URL https://doi.org/10.1023/A:1019150005407.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022. doi: 10.48550/arXiv.2203.15556. URL https://doi.org/10.48550/arXiv.2203.15556.

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. doi: 10.5555/3454287.3454297. URL https://doi.org/10.5555/3454287.3454297.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. doi: 10.48550/arXiv.2001.08361. URL https://doi.org/10.48550/arXiv.2001.08361.

Ahmed Khaled, Kaan Ozkara, Tao Yu, Mingyi Hong, and Youngsuk Park. MuonBP: Faster muon via block-periodic orthogonalization. *arXiv preprint arXiv:2510.16981*, 2025. doi: 10.48550/arXiv.2510.16981. URL https://doi.org/10.48550/arXiv.2510.16981.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. PyTorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020. doi: 10.48550/arXiv.2006.15704. URL https://doi.org/10.48550/arXiv.2006.15704.

Xi-Lin Li. Preconditioned stochastic gradient descent. *IEEE Transactions on Neural Networks and Learning Systems*, 29(5):1454–1466, 2018. doi: 10.1109/TNNLS.2017.2672978. URL https://doi.org/10.1109/TNNLS.2017.2672978.

Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training. *arXiv preprint arXiv:2305.14342*, 2023. doi: 10.48550/arXiv.2305.14342. URL https://doi.org/10.48550/arXiv.2305.14342.

Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for LLM training. *arXiv preprint arXiv:2502.16982*, 2025. doi: 10.48550/arXiv.2502.16982. URL https://doi.org/10.48550/arXiv.2502.16982.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. doi: 10.48550/arXiv.1711.05101. URL https://doi.org/10.48550/arXiv.1711.05101.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pp. 2408–2417. PMLR, 2015. doi: 10.5555/3045118.3045374. URL https://doi.org/10.5555/3045118.3045374.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017. doi: 10.48550/arXiv.1710.03740. URL https://doi.org/10.48550/arXiv.1710.03740.

Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using Kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12359–12367, 2019. doi: 10.1109/CVPR.2019.01264. URL https://doi.org/10.1109/CVPR.2019.01264.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2020. doi: 10.1109/SC41405.2020.00024. URL https://doi.org/10.1109/SC41405.2020.00024.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016. doi: 10.48550/arXiv.1609.04747. URL https://doi.org/10.48550/arXiv.1609.04747.

Hao-Jun Michael Shi, Tsung-Hsien Lee, Shintaro Iwasaki, Jose Gallego-Posada, Zhijing Li, Kaushik Rangadurai, Dheevatsa Mudigere, and Michael Rabbat. A distributed data-parallel PyTorch implementation of the distributed shampoo optimizer for training neural networks at-scale. *arXiv preprint arXiv:2309.06497*, 2023. doi: 10.48550/arXiv.2309.06497. URL https://doi.org/10.48550/arXiv.2309.06497.

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. doi: 10.48550/arXiv.1909.08053. URL https://doi.org/10.48550/arXiv.1909.08053.

Nikhil Vyas, Depen Morwani, Rosie Zhao, Mujin Kwun, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. Soap: Improving and stabilizing shampoo using Adam. *arXiv preprint arXiv:2409.11321*, 2024. doi: 10.48550/arXiv.2409.11321. URL https://doi.org/10.48550/arXiv.2409.11321.

Junjie Wang, Pan Zhou, Yiming Dong, Huan Li, Jia Li, Xun Zhou, Qicheng Lao, Cong Fang, and Zhouchen Lin. Conda: Column-normalized Adam for training large language models faster. *arXiv preprint arXiv:2509.24218*, 2025. doi: 10.48550/arXiv.2509.24218. URL https://doi.org/10.48550/arXiv.2509.24218.

Kaiyue Wen, David Hall, Tengyu Ma, and Percy Liang. Fantastic pretraining optimizers and where to find them. *arXiv preprint arXiv:2509.02046*, 2025. doi: 10.48550/arXiv.2509.02046. URL https://doi.org/10.48550/arXiv.2509.02046.

Tian Xie, Haoming Luo, Haoyu Tang, Yiwen Hu, Jason Klein Liu, Qingnan Ren, Yang Wang, Wayne Xin Zhao, Rui Yan, Bing Su, et al. Controlled LLM training on spectral sphere. *arXiv preprint arXiv:2601.08393*, 2026. doi: 10.48550/arXiv.2601.08393. URL https://doi.org/10.48550/arXiv.2601.08393.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a. doi: 10.48550/arXiv.2505.09388. URL https://doi.org/10.48550/arXiv.2505.09388.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025b. URL https://doi.org/10.48550/arXiv.2412.15115.

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. doi: 10.48550/arXiv.2304.11277. URL https://doi.org/10.48550/arXiv.2304.11277.

A MEGATRON PARALLELISM OVERVIEW

Megatron (Shoeybi et al., 2019) partitions training states across data parallelism (DP), tensor parallelism (TP), and pipeline parallelism (PP) (Huang et al., 2019), as illustrated in Figure 6. In DP, DDP fully replicates parameters, gradients, and optimizer states, while ZeRO-1 (Rajbhandari et al., 2020; Zhao et al., 2023) shards optimizer states to reduce memory. In TP, individual tensors are partitioned across devices; in PP, different devices hold different subsets of the model along its depth.

These partitioning strategies work naturally for element-wise optimizers, because each scalar can be updated independently. Matrix-based optimizers instead require whole-tensor operations. The default ZeRO-1 and TP partitioning patterns can split a tensor across multiple ranks, violating this atomicity requirement. DDP and PP preserve whole tensors within each rank's assigned model portion and are therefore naturally compatible with these updates. This distinction motivates Canzona's separate mechanisms for the DP and TP planes.

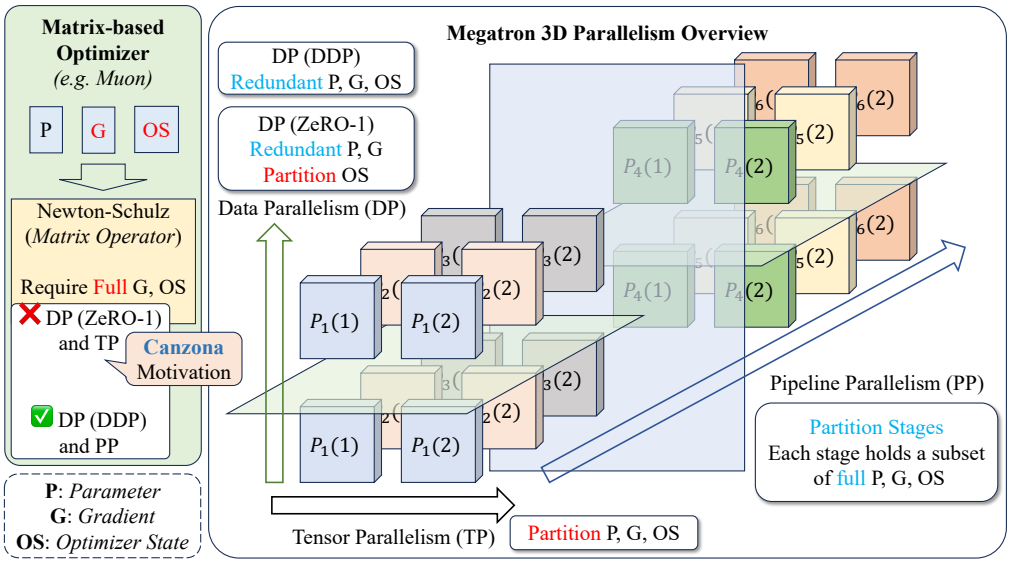


Figure 6: Motivation of Canzona in Megatron's 3D parallel training. Megatron partitions training states differently across its three parallelism dimensions. In Data Parallelism (DP), DDP fully replicates parameters, gradients, and optimizer states, while ZeRO-1 shards optimizer states to reduce memory. In Tensor Parallelism (TP), parameters, gradients, and optimizer states are partitioned across devices, whereas in Pipeline Parallelism (PP), different devices hold different subsets of model states along the model depth. Matrix-based optimizers, however, require holistic tensor-level operations and are therefore incompatible with the default ZeRO-1 and TP partitioning patterns, while remaining naturally compatible with DDP and PP. This mismatch motivates Canzona, which is designed to restore efficient support for matrix-based optimizers in Megatron's distributed training stack.

A.1 MEGATRON BUFFERS AND TRAINING WORKFLOW

Data Parallelism (DP) is the dominant paradigm for training Large Language Models (LLMs). In standard Distributed Data Parallelism (DDP) (Li et al., 2020) (Figure 1), the model parameters and optimizer states are replicated across all parallel data ranks. While this enables parallel gradient computation, the redundancy of optimizer states consumes a significant portion of GPU memory—often 2–3$\times$ the model size for mixed-precision training (e.g., FP32 master weights (Micikevicius et al., 2017) and momentum).

Zero Redundancy Optimizer (ZeRO-1) (Rajbhandari et al., 2020) addresses this redundancy by partitioning the model states across data-parallel ranks. Specifically, ZeRO-1 (Optimizer State Partitioning) shards the optimizer states such that each rank is responsible for updating only a fraction $(1/R)$ of the total parameters. This linear reduction in memory footprint is crucial for scaling typically massive models.

Frameworks like Megatron (Shoeybi et al., 2019) implement ZeRO-1 through its `DistributedOptimizer`, which relies on a unified memory management structure known as the `param_and_grad_buffer` ("Flat" and "Bucket" boxes in Figure 1).

Continuous Memory Buffer and Bucketing of `param_and_grad_buffer`. Instead of managing memory for each parameter tensor individually, Megatron flattens all model parameters (and their gradients) into several continuous memory buffers. To facilitate communication overlapping of forward and backward, each continuous buffer is logically divided into N *buckets*. As parameters are registered, they are packed into these buckets sequentially. This contiguous layout is essential for high-performance collective communications, allowing a single large operation (e.g., `Reduce-Scatter`) to process multiple parameters simultaneously rather than launching many small kernels.

Uniform Partition Rule and ZeRO-1 Geometric Constraint. Crucially, Megatron imposes a strict geometric partition rule. For a bucket of size B, the buffer is rigidly divided into R equal contiguous segments of size $S = |B|/R$, entirely agnostic to the boundaries of individual parameter tensors. Consequently, rank r is hard-wired to process the r-th segment. This rigid positional mapping forms a fundamental **ZeRO-1 Geometric Constraint** that is essential for efficient communication primitives like `Reduce-Scatter`.

The typical training workflow with this buffer structure proceeds as follows:

- **Backward Pass:** As the backward pass proceeds, gradients are computed and accumulated into the buckets. Once the gradients for all parameters in a bucket are ready, a `Reduce-Scatter` operation is triggered across all DP ranks. This operation aggregates the gradients and shards them, such that each rank receives only its assigned continuous segment S. Importantly, this communication is overlapped with the backward computation of the subsequent bucket.
- **Optimizer Step:** Rank r performs the optimizer step (updating parameters) strictly on its assigned memory range $[(r-1)S, rS)$.
- **Forward Pass:** Before the forward computation for a bucket can begin, the updated parameter shards must be gathered from all ranks. An `All-Gather` operation is triggered in the first micro-step to reconstruct the full parameters for that bucket. Similar to the backward pass, this communication is overlapped with the forward computation of the previous bucket.

B SUMMARY OF NOTATIONS

Table 1: Summary of Notations

Symbol	Description
	System Settings & Preliminaries
R	Total number of ranks (DP ranks (Section 3) or TP ranks (Section 4))
$\mathcal{B}$	The set of logical buckets $\{B_1, B_2, \ldots, B_N\}$ in the `param_and_grad_buffer`
B_i	The i-th logical bucket consisting of ordered parameters
P_i	The sequence of non-splitable parameters $(p_{i,1}, \ldots, p_{i,M_i})$ in B_i
S	Uniform shard size ($\lvert B\rvert/R$) in standard ZeRO partition
$\mathcal{I}_r$	The memory interval $[(r-1)S, rS)$ assigned to rank r in ZeRO
o_p	The starting memory offset (index) of parameter p in the flattened buffer
	Load Cost Metrics
$\mathcal{C}_{\text{mem}}(p)$	Memory usage cost of parameter p
$\mathcal{C}_{\text{comp}}(p)$	Computational cost of parameter p (FLOPs)
$\mathcal{C}_{\text{comm}}(p)$	Communication cost of parameter p (parameter size, $= numel(p)$)
$\mathcal{W}_{\text{load}}(p)$	Primary metric for load balancing ($\mathcal{C}_{\text{mem}}$ or $\mathcal{C}_{\text{comp}}$)
$\mathcal{W}_{\text{size}}(p)$	Secondary metric for communication grouping (usually $\mathcal{C}_{\text{comm}}$)
$\mathcal{W}(p)$	A general cost metric assigned to parameter p for load balancing
	Problem Formulation & Load-Balance Algorithm for DP LB-ASC
$\mathbf{s}_i$	Slicing vector $[s_{i,0}, \ldots, s_{i,R}]$ containing cut points for bucket B_i
$\mathcal{U}_i$	Set of feasible (atomic) cut points for bucket B_i
$L_{i,r}$	The cumulative load assigned to rank r in bucket B_i
$S_{i,r}$	The data size assigned to rank r in bucket B_i. Here, $S_{i,r} = s_{i,r} - s_{i,r-1}$
α	Control parameter $\in [0, 1]$ trading off between $\mathcal{J}_{\text{DP}}$ and $\mathcal{J}_{\text{Comm}}$
$\mathbf{L}$	Vector tracking the global accumulated load for each rank
$\mathbf{v}_{\text{even}}$	Basis vector for perfectly even distribution
$\mathbf{v}_{\text{fill}}$	Basis vector for deficit-filling distribution
$\mathbf{v}^*$	Blended target allocation vector for the current bucket
$\Phi_i(u)$	Cumulative load function of bucket B_i up to cut point u
	Problem Formulation & Load-Balance Algorithm for TP LB-ASC
$\mathbb{M}$	The sequence of Micro Groups $\{M_1, ..., M_K\}$ generated by the scheduler
C_{max}	The maximum capacity constraint (memory or FLOPs) for a single Micro Group

C MORE EXPERIMENTS

C.1 FULL EXPERIMENT SETUP

Models and Optimizers. We evaluate the scalability and efficiency of our framework using the **Qwen3** (Yang et al., 2025a) model family, scaling from 1.7B to 32B parameters. To verify the generality of our approach across different matrix-based algorithms, we conduct experiments with three representative second-order optimizers: **Muon** (Jordan et al., 2024), **Shampoo** (Gupta et al., 2018), and **SOAP** (Vyas et al., 2024). Most of the experiments primarily report the detailed performance analysis using **Qwen3-32B** trained with the **Muon** optimizer, unless otherwise specified. The training configuration utilizes a batch-size per DP rank of 1 and a sequence length of 4096.

Hardware and software environment. All experiments are conducted on the same production HPC cluster with 8-GPU nodes. Each node provides high-bandwidth intra-node interconnects, while cross-node communication uses a high-speed network fabric. For all reported results, tensor parallelism is restricted within a node, whereas data parallelism spans nodes when needed. For confidentiality reasons, we omit the exact accelerator model, but all compared methods are evaluated under identical hardware conditions. Our implementation is built on Megatron-LM (NVIDIA Megatron Core 0.16.1) with Python 3.12, PyTorch 2.7, and CUDA 12.8.

Baselines and Methods. We compare three distinct optimization strategies to demonstrate the effectiveness of our decoupled load-balancing framework:

- **SC (Synchronous Compute, Baseline, Paradigm 1 in Section 3.1):** Represents the naive practice (e.g., DDP for DP and `All-Gather` for TP). It employs naive non-partitioning (DP-SC) and synchronous collective communication (TP-SC), resulting in redundant computation and blocking synchronization.
- **NV-layerwise (Baseline, Paradigm 2 in Section 3.1):** NVIDIA's `layerwise_optimizer` implementation. It assigns optimizer states at the granularity of whole layers to respect tensor boundaries. However, as analyzed in Section 3 and Section 3.1, this approach introduces a Geometric Incompatibility between task assignment and parameter partition. This mismatch precludes efficient bucket-based `Reduce-Scatter` overlapping and necessitating expensive `All-Reduce` operations.
- **ASC (Asynchronous Compute, Paradigm 3 in Section 3.1):** Adopts our proposed decoupled architecture to enable asynchronous execution (DP-ASC + TP-ASC) but utilizes naive partitioning without load-aware scheduling.
- **LB-ASC (Load-Balance Asynchronous Compute, Our Framework's Core Strategy):** The complete implementation of our proposed strategies (DP-LB-ASC + TP-LB-ASC), which integrates the α-Balanced Static Partitioning for DP and the Greedy LPT scheduling for TP to minimize stragglers.

Evaluation Metrics. We analyze system performance using two primary metrics:

1. **Runtime Breakdown:** We report the wall-clock time for different training phases. To provide a comprehensive performance context, we include the *Fwd-Bwd time* (Forward and Backward pass) and standard *AdamW optimizer time* as references. The primary metric for comparison is the *Target Optimizer time* (e.g., Muon step latency). All reported timings are averaged over 10 runs and 10 steps within each run to ensure stability.
2. **Load-Balance Ratio:** To quantify the severity of the long-tail straggler problem, we define the Load-Balance Ratio (R_{LB}) for both memory footprint and computational FLOPs:

$$R_{LB} = \frac{\max_r(\sqsubseteq_r)}{\text{avg}_r(\sqsubseteq_r)} \tag{5}$$

where $\sqsubseteq_r$ represents the metric value (Peak Memory or FLOPs) on rank r. An ideal balanced system yields $R_{LB} \approx 1.0$, while a higher ratio indicates severe imbalance.

C.2 Full Performance Comparison with `layerwise_optimizer`

To rigorously evaluate the superiority of our proposed framework over the current state-of-the-art solution for atomicity preservation, we conduct a comprehensive comparison against NVIDIA's `layerwise_optimizer` on a cluster with up to 256 GPUs. Figure 7 presents the step latency breakdown across the entire Qwen3 model family (1.7B, 4B, 8B, 14B, and 32B) under various Data Parallel (DP) and Tensor Parallel (TP) configurations.

Consistent Superiority. The results demonstrate that our proposed framework consistently outperforms NV-layerwise in every tested configuration. The total iteration time reduction ranges from significant to dramatic, driven primarily by the optimization of the optimizer step (Purple bar). For instance, in the Qwen3-32B (DP16-TP8) setting, our method reduces the specific optimizer latency by approximately 8.3x compared to NV-layerwise.

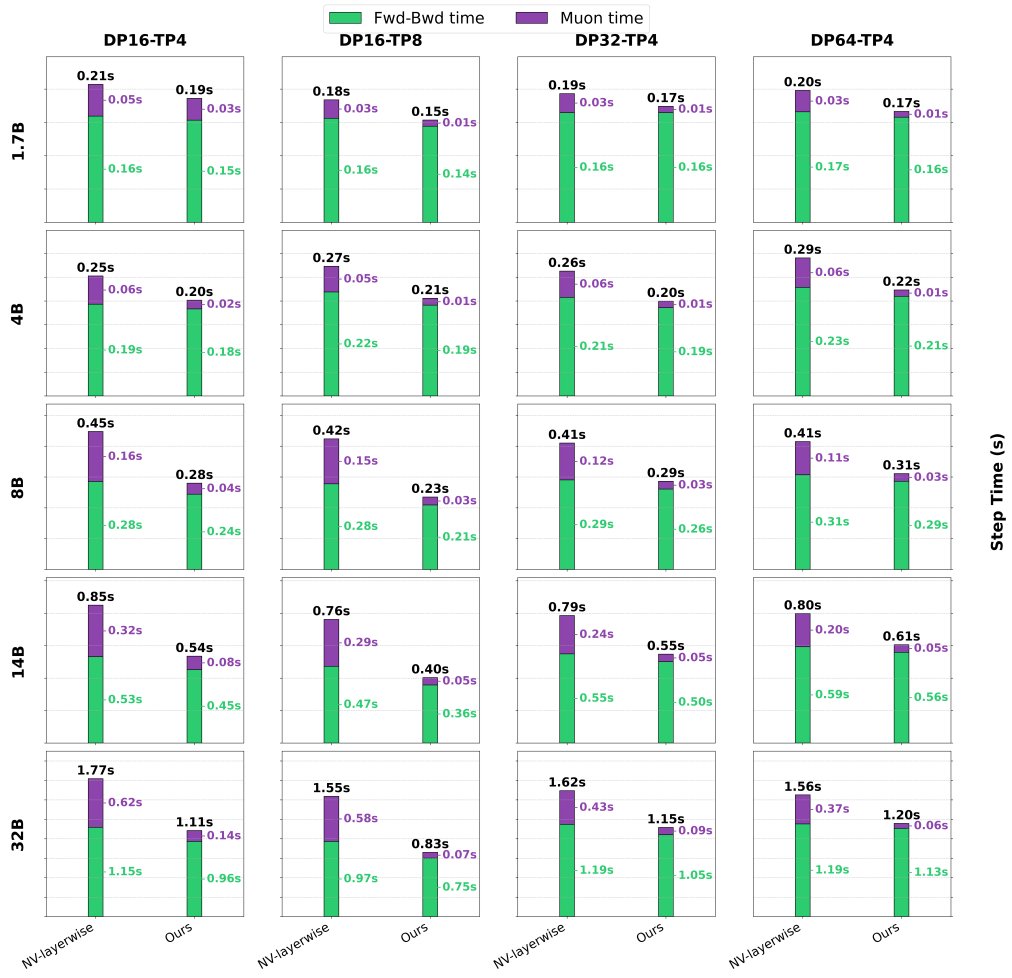


Figure 7: End-to-End Performance Comparison with NVIDIA `layerwise_optimizer`. We compare the breakdown of step latency between `layerwise_optimizer` (NV-layerwise) and our proposed framework across the Qwen3 model family (ranging from 1.7B to 32B parameters) under various parallelism configurations. (Top Row) Results for smaller models (1.7B, 4B) showing consistent latency reduction. (Bottom Row) Results for larger models (14B, 32B) where the computational complexity of the optimizer increases. Our method demonstrates robust scalability, with the performance gap widening as the model size and optimizer complexity grow, primarily due to the significant reduction in Optimizer Time (Purple bar).

Robustness to Model Size. A key observation is that the performance gap widens as the model size increases. For smaller models like Qwen3-1.7B, the absolute time savings are noticeable but bounded by the relatively low computational cost of the optimizer. However, as we scale to Qwen3-32B, the matrix-based operations become computationally dominant. NV-layerwise, constrained by expensive collective `All-Reduce` and extra `All-gather`, exposes this latency fully. In contrast, our asynchronous pipeline effectively hides these expensive communication operators, resulting in a more pronounced speedup for larger models.

Robustness to Parallelism Strategies. We further observe that our advantage remains robust regardless of the specific parallelism split (e.g., comparing DP16-TP8 vs. DP32-TP4). While NV-layerwise shows sensitivity to the communication patterns induced by different TP sizes, our static partitioning and micro-group scheduling adapt to the changing topology, maintaining high throughput. This confirms that our load-balancing algorithms are agnostic to the specific dimension of parallelism employed.

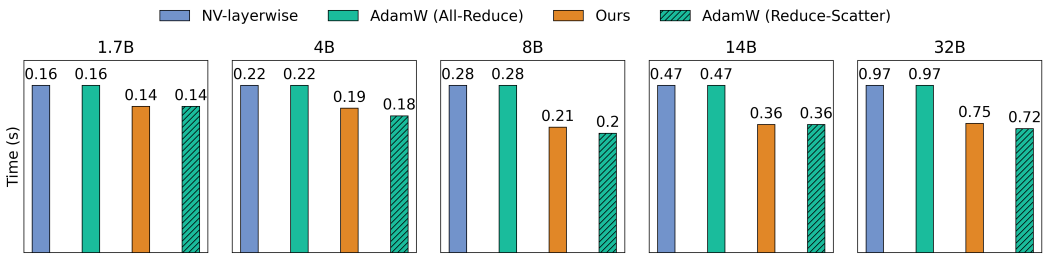
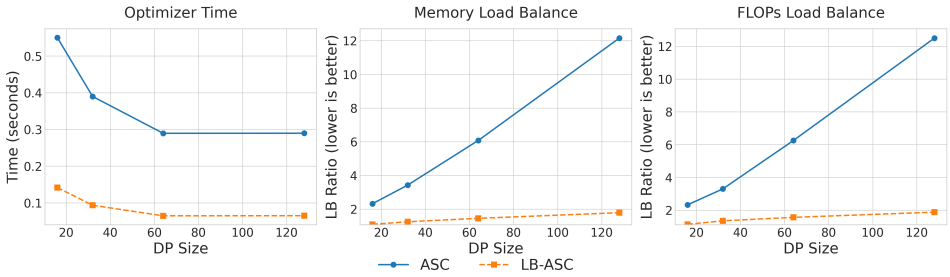


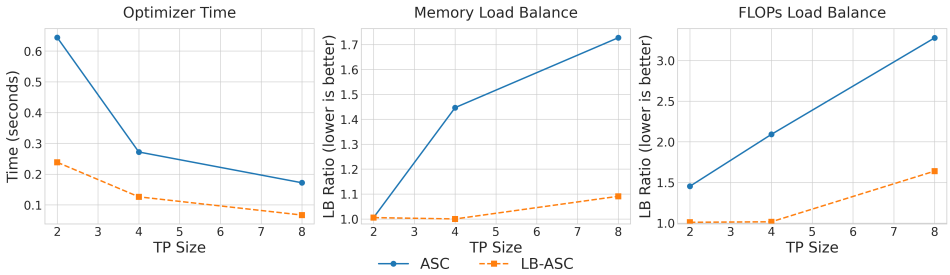
Figure 8: **Verification of Communication Efficiency in Forward-Backward Pass.** We compare the Fwd-Bwd step latency of our framework and the NV-layerwise against two controlled baselines: AdamW with `All-Reduce` (simulating the heavy communication volume necessitated by layer-wise partitioning) and `Reduce-Scatter` (representing the optimal ZeRO-1 communication volume). The results show that the NV-layerwise (blue) consistently aligns with the AdamW `All-Reduce` baseline (teal solid), confirming that its performance is bottlenecked by the 2x communication volume of `All-Reduce` operations despite overlapping. Conversely, our method (orange) closely tracks the AdamW `Reduce-Scatter` baseline (teal hatched), confirming that our static partitioning strategy successfully preserves the efficient communication capabilities of the underlying Megatron framework.

Supplementary Experiment of Fwd-Bwd Communication Efficiency To quantitatively verify the effectiveness of communication overlap during the Forward-Backward (Fwd-Bwd) pass, we designed a comparative experiment. Since isolating precise communication and computation times within the complex pipelining of Fwd-Bwd is challenging, we established two reference baselines using the standard AdamW optimizer training in Megatron: (1) AdamW (`All-Reduce`, DDP), where gradients are collected via overlapping `All-Reduce`, representing the latency upper bound; and (2) AdamW (`Reduce-Scatter`, ZeRO-1), where `Reduce-Scatter` communication is optimally overlapped with computation (standard ZeRO-1 behavior), representing the theoretical lower bound.

Figure 8 demonstrates that our proposed framework successfully preserves the efficient bucket-based communication mechanism, achieving Fwd-Bwd latencies that closely track the AdamW (`Reduce-Scatter`) baseline. In contrast, the `layerwise_optimizer` exhibits latencies matching the AdamW (`All-Reduce`) baseline, confirming that its violation of ZeRO-1 geometric constraints prevents effective hiding of communication overhead. We note that in certain configurations, our Fwd-Bwd time is marginally higher than the ideal AdamW (`Reduce-Scatter`) setting. This minor discrepancy arises because our load-balancing strategy necessitates variable-sized parameter chunks (Section 3.3), introducing slight communication imbalances compared to AdamW's perfectly equal chunks. However, this impact is negligible compared to the significant performance penalty of the inefficient `All-Reduce` primitive.



(a) Data Parallelism Size Scaling Analysis (Fix TP=4)



(b) Tensor Parallelism Size Scaling Analysis (Fix PP=4 and DP=4)

Figure 9: **Parallelism Scaling Analysis.** We evaluate the scalability of our load balancing algorithms as the degree of parallelism increases. (a) Data Parallelism Scaling: We fix the model (Qwen3-32B, Muon) and scale DP size from 16 to 128. While our proposed ASC suffers from increasing load imbalance (rising load-balance ratio) and slower convergence, our α-Balanced strategy (DP LB-ASC) maintains a near-optimal load-balance ratio (≈ 1.0) and stable optimizer time. (b) Tensor Parallelism Scaling: Similarly, scaling TP size from 2 to 8 reveals that our Micro-Group Scheduling (TP LB-ASC) effectively neutralizes the straggler effect that plagues the non-load-balanced ASC.

C.3 Scaling Analysis

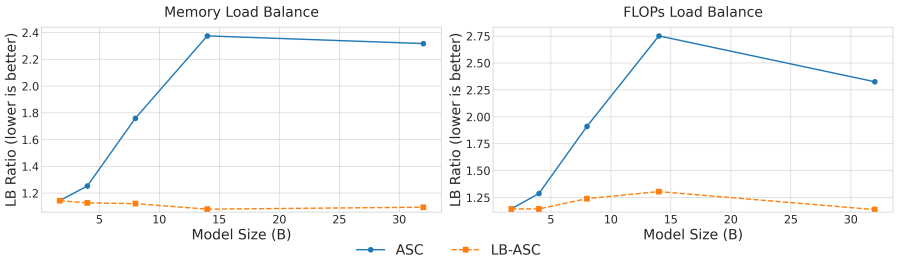
We investigate the scalability of our approach on a large-scale cluster of up to 512 GPUs from two dimensions: system size (Parallelism Scaling) and workload complexity (Model Size Scaling).

Parallelism Scaling (Figure 9). We fix the model size (Qwen3-32B) and scale the number of GPUs.

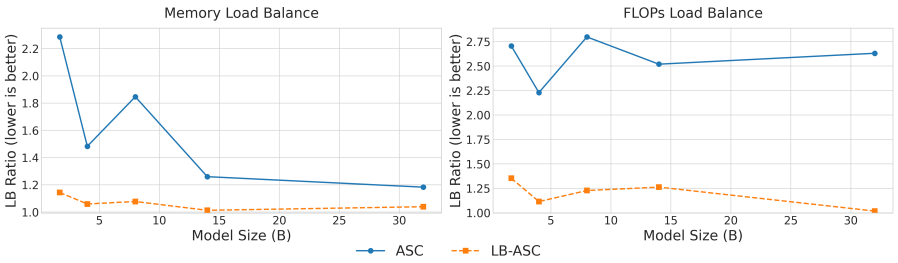
- **Data Parallelism (Figure 9a):** With TP fixed at 4, as we scale the DP size from 16 to 128, the statistical variance in parameter sizes naturally leads to severe load imbalance in naive strategies. The ASC baseline exhibits a linear degradation in the load-balance ratio for both memory and FLOPs, leading to increased optimizer time due to stragglers. Conversely, our DP LB-ASC (α-Balanced) strategy maintains a load-balance ratio close to the ideal 1.0, effectively neutralizing the straggler effect even at large scale (128 DP ranks).
- **Tensor Parallelism (Figure 9b):** Similarly, with PP fixed at 4 and DP fixed at 4, increasing TP size typically exacerbates the fragmentation of weight matrices. The baseline shows a sharp increase in computational imbalance. Our Micro-Group Scheduling (TP LB-ASC) successfully mitigates this, keeping the FLOPs load-balance ratio significantly lower than the baseline, thereby preserving low optimizer latency as the system scales.

Model Size Scaling (Figure 10). We analyze how parameter heterogeneity affects load balancing as models grow from 1.7B to 32B. For these experiments, we fix the parallelism configuration to DP=16 and TP=4.

- **DP Load Balance (Figure 10a):** Larger models typically contain a wider variance of tensor shapes (e.g., larger embedding layers vs. smaller projection heads). This heterogeneity



(a) Model Size Scaling with DP-load-balance Analysis



(b) Model Size Scaling with TP-load-balance Analysis

Figure 10: **Model Size Scaling Analysis.** We evaluate the robustness of our load balancing algorithms across different model sizes (1.7B to 32B). (a) Data Parallelism: Larger models introduce greater parameter heterogeneity, causing the load-balance ratio of the baseline (ASC) to degrade significantly. Our LB-ASC strategy maintains a consistent balance. (b) Tensor Parallelism: The load imbalance fluctuates based on the specific hidden dimension alignment, yet our Greedy Scheduling consistently optimizes the packing.

causes the naive partitioning's load-balance ratio to increase significantly with model size. Our LB-ASC strategy adapts to this heterogeneity, maintaining a flat LB profile.

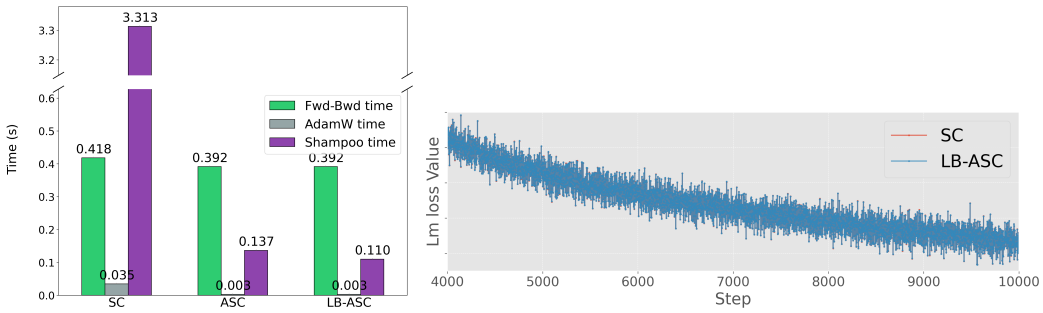
- **TP Load Balance (Figure 10b):** Interestingly, for TP, the baseline imbalance does not strictly increase with model size but fluctuates based on the specific architecture (e.g., hidden dimension size). However, our greedy scheduling algorithm consistently finds a near-optimal packing, ensuring that our performance advantage is sustained across all model sizes.

C.4 Generalization to Other Optimizers (Shampoo & SOAP)

To verify the generality of our framework beyond Muon, we evaluate its performance on Shampoo and SOAP, two other representative matrix-based optimizers. These algorithms involve distinct cubic-complexity operations (SVD and Eigendecomposition, respectively), serving as robust test cases for our system's adaptability.

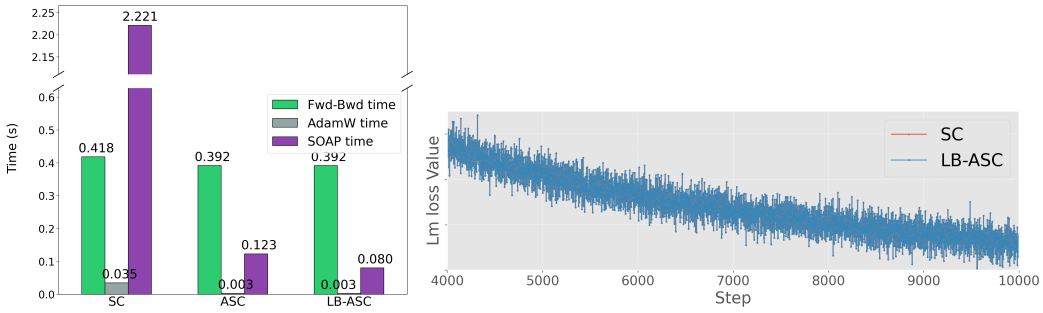
Efficiency (Figures 11a & 12a). These experiments are conducted on 256 GPUs. Due to the significant memory overhead of Shampoo and SOAP preconditioners, we utilize Qwen3-14B with a configuration of PP=2, DP=32, and TP=4 to accommodate the state requirements. The matrix operations in Shampoo and SOAP incur high computational costs, as evidenced by the high latency in the SC baseline (e.g., Shampoo step takes 3.313s). The experimental results confirm that our proposed framework remains highly effective in this context. By applying our unified partitioning and scheduling strategy, we reduce the optimizer step time to 0.110s for Shampoo and similarly for SOAP. This represents a speedup of over 30x, demonstrating that our framework can successfully handle diverse matrix-based workloads without requiring algorithm-specific tuning.

Precision (Figures 11b & 12b). We validate correctness by training a Qwen3-1.7B model for 400B tokens using a DP=8, TP=4 configuration. The loss curves for both Shampoo and SOAP under our framework overlap perfectly with the standard synchronous baseline. It is important to emphasize that



(a) Efficiency Validation with Qwen3-14B (b) Precision Validation with Qwen3-1.7B

Figure 11: **Generality Validation with Shampoo Optimizer.** (a) Efficiency: Comparison of step time for Qwen3-14B using the Shampoo optimizer. The results demonstrate that our proposed LB-ASC strategy is equally effective for Shampoo, achieving a drastic speedup compared to both the synchronous (SC) and asynchronous (ASC) methods. (b) Precision: Training loss curves for Qwen3-1.7B confirm that our system optimizations do not alter the convergence trajectory compared to the synchronous baseline.



(a) Efficiency Validation with Qwen3-14B (b) Precision Validation with Qwen3-1.7B

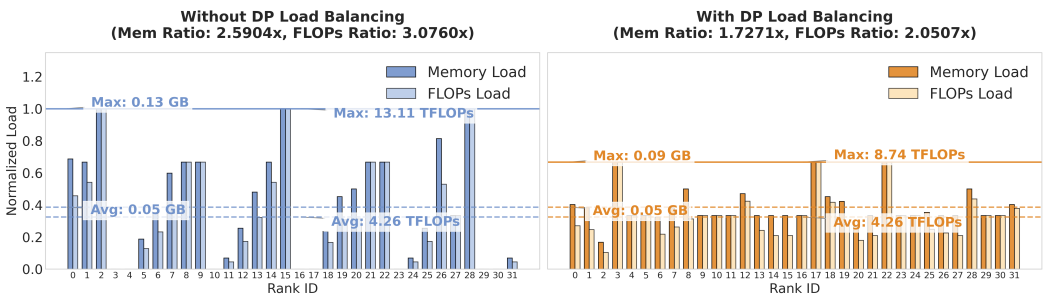
Figure 12: **Generality Validation with SOAP Optimizer.** (a) Efficiency: Step time comparison for Qwen3-14B trained with the SOAP optimizer. Similar to the Shampoo results, our framework generalizes well to SOAP, significantly reducing step latency. (b) Precision: The loss curve for Qwen3-1.7B confirms that our implementation introduces no algorithmic deviation, ensuring mathematical equivalence with standard synchronous training.

LB-ASC strategy is a purely system-level optimization. Unlike methods that approximate curvature or skip updates to save time, our approach strictly adheres to the original optimizer's mathematical definition. Consequently, there is zero mathematical precision loss, and the convergence behavior is identical to that of the baseline.

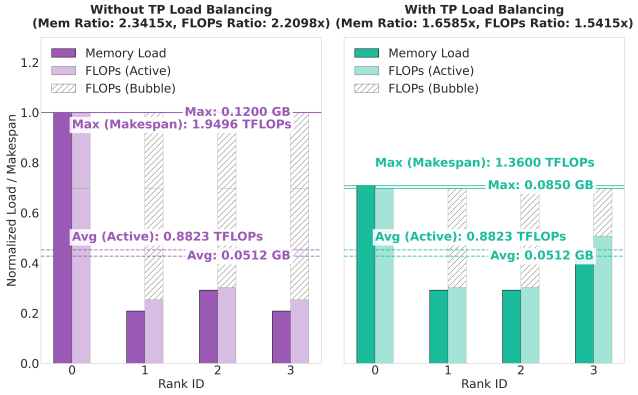
Load Balance (Figure 13). The load distribution analysis for Shampoo / SOAP further corroborates the effectiveness of our proposed LB-ASC strategy. Despite the different computational characteristics of Shampoo compared to Muon, our scheduler successfully flattens the workload variance (reducing the FLOPs load-balance ratio from > 2.0 to ≈ 1.05), ensuring efficient hardware utilization across all ranks.

C.5 DP Load Balance α Analysis

We conduct an ablation study on 128 GPUs (configured as PP=8, DP=16) to determine the optimal value of the control parameter α in our α-Balanced Greedy LPT algorithm (Section 3.2). Recall that $\alpha \to 0$ prioritizes uniform communication sizes (matching standard ZeRO), while $\alpha \to 1$ prioritizes balanced optimizer computational load. As shown in Figure 14, as α increases from 0.0 to 1.0, the Muon time (Purple bar) decreases monotonically. This confirms that computational stragglers are the primary bottleneck during the optimizer step. Crucially, while increasing α introduces slight non-



(a) Data Parallelism Load Balance Ratio



(b) Tensor Parallelism Load Balance Ratio

Figure 13: **Load Balance Analysis for Shampoo/SOAP.** Visualization of the load distribution (Memory and FLOPs) across ranks for the Shampoo optimizer. Similar to the Muon results, the naive partitioning (Left) leads to significant computational bubbles and memory peaks, while our proposed load-balancing strategy (Right) effectively flattens the distribution, achieving a near-perfect load-balance ratio.

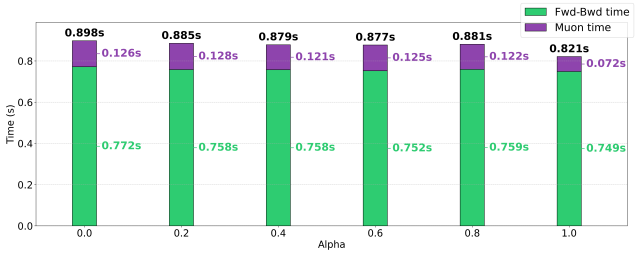


Figure 14: **Sensitivity to DP Load-Balance Factor α.** We analyze the impact of the control parameter α (ranging from 0.0 to 1.0) on the total step latency for Qwen3-32B. $\alpha = 0$ prioritizes communication size uniformity, while $\alpha = 1$ prioritizes optimizer computational load balance. The results indicate that setting $\alpha = 1.0$ yields the lowest total time, suggesting that computational stragglers in the optimizer step are the dominant bottleneck, while communication imbalance in the Fwd-Bwd pass is effectively hidden by overlap.

uniformity in the communication bucket sizes (potentially affecting Fwd-Bwd time), the Fwd-Bwd time (Green bar) remains relatively stable. This stability suggests that the efficient communication overlapping in Megatron effectively masks the minor communication imbalances introduced by our sizing strategy. Consequently, setting $\alpha = 1.0$ yields the best end-to-end performance, justifying our design choice to prioritize computational load balancing in the final implementation.

C.6 TP Communication Fusion Analysis

We analyze the impact of the Micro-Group Fusion strategy for Tensor Parallelism (Section 4.1) using 128 GPUs with DP=16 and TP=8. We vary the maximum capacity constraint (C_{max}) of the micro-groups, which dictates the buffer size for the fused All-to-All communication.

Figure 15 illustrates the Optimizer Time as a function of C_{max}. The "No-Fuse" baseline, which performs communication for each tensor individually, suffers from high latency ($\approx 0.11s$) due to the overhead of launching many small kernels and poor bandwidth utilization. Enabling fusion immediately drops the latency to $\approx 0.073s$. We observe that performance improves slightly as C_{max} increases, stabilizing around 512MB. Beyond this point (1024MB, 2048MB), the latency plateaus, indicating that the All-to-All bandwidth is fully saturated. This result confirms that our fusion strategy is essential for efficiency and that the system is robust to the specific choice of C_{max} provided it is sufficiently large to saturate the interconnect.

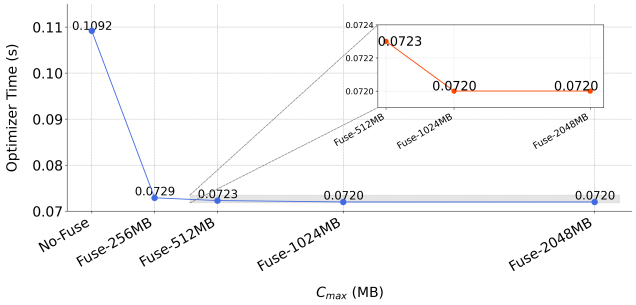


Figure 15: **TP Micro-Group Fusion.** We evaluate the impact of fusing tensor updates into Micro Groups with varying capacity constraints (C_{max}). "No-Fuse" represents treating each tensor individually. The results demonstrate that fusing communication significantly reduces optimizer time by saturating All-to-All bandwidth. Performance plateaus once C_{max} exceeds 1024MB, indicating the optimal granularity for overlap.

D Supplementary Explanation

D.1 Load-Balance Algorithm Latency

It is worth noting that the load-balancing algorithms described in Section 3.2 and Section 4.2 are executed strictly as a **one-time offline planning step** during model initialization. Consequently, they introduce almost zero overhead to the subsequent training iterations. Furthermore, the computational cost of this initialization phase itself is negligible: both algorithms are designed as efficient greedy heuristics with a time complexity dominated by sorting ($O(N \log N)$), where N is the number of parameter tensors or buckets (typically in the scale of thousands).

In our large-scale experiments with Qwen3-32B on 256 GPUs, the offline planning phase completes in milliseconds. This latency is orders of magnitude smaller than the heavy system initialization procedures, such as CUDA context creation, memory allocation, and NCCL communicator setup (which typically span tens of seconds to minutes). Consequently, the scheduling overhead is effectively imperceptible and cannot be distinguished from system noise during the model initialization stage.

Given that this overhead is effectively imperceptible and indistinguishable from system noise, we omit specific latency experiments for the load-balancing phase in our reported results.

D.2 The Geometric Incompatibility of Layer-wise Partitioning

Mechanism (The "Hybrid" ZeRO-1). Existing layer-wise strategies (e.g., NVIDIA's `layerwise_optimizer`) operate as a hybrid system to reconcile atomicity with distribution. During the Forward and Backward passes, they function in a DDP mode and utilize standard `All-Reduce` (incurring $2\times$ communication volume compared to ZeRO-1's Reduce-Scatter) to synchronize gradients. However, for the Optimizer Step, they shard the optimizer states and employ

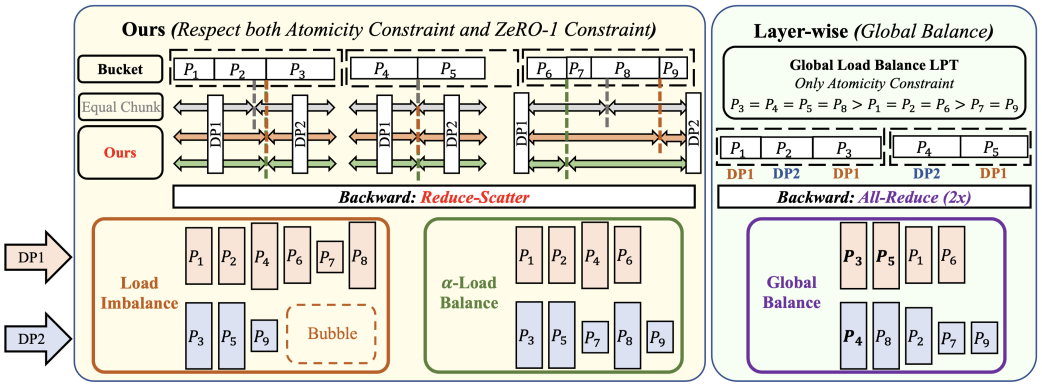


Figure 16: **Visualization of Geometric Incompatibility. (Left) Ours:** Our approach respects both Atomicity and the strict Geometric Constraints of ZeRO-1 primitives. By optimizing slice sizes (shifting boundaries) inside bucket without scrambling the physical parameter sequence, we preserve the contiguous bucket abstraction. This alignment enables the use of the optimal `Reduce-Scatter` primitive. **(Right) Layer-wise:** Strategies relying on Global Load Balancing (e.g., Global LPT) assign tasks purely by weight (e.g., assigning heavy parameter P_3 to DP1) regardless of their physical location. This Data-Task Mismatch (visualized by the interleaved DP1/DP2 assignments in the bucket view) breaks bucket coalescing. Consequently, under Megatron ZeRO-1's fixed geometric slicing and bucket coalescing assumption, the system must abandon bucket-based `Reduce-Scatter` in favor of bucket-based `All-Reduce` or per-parameter-based `Reduce-Scatter`, incurring 2x communication volume and higher latency. (Note: The visualization of the third bucket (P_6, P_7, P_8, P_9) in the right sub-figure is omitted here because its load-based assignment ($P_6 \in$ DP1, $\{P_7, P_8, P_9\} \in$ DP2) coincidentally aligns with the geometric rank order (monotonic), thus not violating the ZeRO-1 Geometric Constraint.)

a **Global Load Balancing** strategy (Global LPT) to assign the ownership of optimizer states. This assignment is based strictly on computational weights (e.g., placing heavy layers on specific ranks) to equalize execution time, disregarding the physical memory layout of the parameters.

The Conflict (ZeRO Geometric Incompatibility in Megatron). This design introduces a fundamental conflict between the logical task assignment and the physical communication primitives used in modern training stacks like ZeRO-1.

- **ZeRO-1 Primitive (Position-Based):** The efficient `Reduce-Scatter` primitive (Appendix A.1) relies on a strict **Geometric Partitioning** logic (Figure 16 (Right)). For a contiguous bucket of size B across R ranks, the primitive rigidly dictates that the r-th geometric slice (interval $[\frac{r-1}{R}B, \frac{r}{R}B)$) is sent to Rank r. The destination is determined solely by the data's *physical position* in the buffer.
- **Layer-wise Logic (Weight-Based):** In contrast, the layer-wise allocator assigns parameter update tasks based on *load*. As illustrated in Figure 16 (Right), a heavy parameter such as P_3 might be assigned to **DP1** to balance the global workload, even though its physical position in the buffer falls within a region that geometrically belongs to or is interleaved with **DP2**. This creates a clear **Data-Task Mismatch**.

The Consequence: The "Lose-Lose" Dilemma. Due to this data-task mismatch, layer-wise approaches are forced to choose between two suboptimal communication paths:

- **Option A: The Bandwidth Penalty (Forced All-Reduce).** This is the default choice for current `layerwise_optimizer` implementations. Since the system cannot coalesce the geometrically-fragmented task assignments into a single geometry-based `Reduce-Scatter`, it defaults to `All-Reduce`. This broadcasts full gradients to all ranks, ensuring that every rank receives the necessary data regardless of its assignment. However, this incurs **2x the communication volume** compared to ZeRO-1's `Reduce-Scatter`, significantly increasing the bandwidth requirement.

- **Option B: The Latency Penalty (Forced Reduce-Scatter).** Alternatively, one might attempt to enforce a `Reduce-Scatter` (sending only the specific parameter data to its assigned rank). However, this fundamentally breaks the **Bucket Coalescing** abstraction. Because the destination ranks are no longer geometrically aligned—visualized in Figure 16 as the interleaved assignment sequence $P_1(\text{DP1}) \rightarrow P_2(\text{DP2}) \rightarrow P_3(\text{DP1})$—the system cannot launch a single monolithic kernel for the entire bucket. Instead, it must dismantle the bucket and trigger **discrete, per-parameter communication kernels** to route specific tensors to their arbitrary destinations. This introduces massive kernel launch overhead and prevents the system from saturating the interconnect bandwidth, shifting the bottleneck from bandwidth to latency.

Compounded Penalty on Parameter Update. Crucially, this geometric violation extends beyond gradient synchronization to parameter reconstruction. Under ZeRO-1 geometric constraints, the updated parameters are gathered via a bucket-based `All-Gather` that is efficiently overlapped with the Forward pass computation. However, because layer-wise partitioning misaligns the ownership of the updated parameters with their geometric shards, a standard coalesced `All-Gather` is impossible. Consequently, implementations are forced to either: (1) perform an explicit `All-Gather` or `Broadcast` *within* the Optimizer Step to redistribute weights (adding significant exposed latency, as observed in our baseline experiments); or (2) resort to inefficient per-parameter `All-Gather` operations during the Forward pass, which destroy the overlap efficiency.

In summary, our framework outperforms layer-wise approaches because our α-Balanced Partitioning respects not only the **Atomicity Constraint** but also the **Geometric Constraints of ZeRO-1 primitives**. By optimizing slice sizes without breaking the sequential rank ordering, we retain the efficient, coalesced `Reduce-Scatter` capability in backward, and the corresponding overlapped, coalesced `All-Gather` in forward, that layer-wise strategies are structurally forced to abandon.

D.3 Detailed Walkthrough of Load-Balancing Algorithms

In this section, we provide a rigorous walkthrough of the two core scheduling algorithms proposed in the paper. We explain the mathematical intuition behind the variable definitions and the algorithmic decisions.

Algorithm 1: α-Balanced Static Partitioning (DP) The core challenge in DP is determining the cut points $s_{i,r}$ for a bucket B_i such that we balance two objectives: **DP Computational Load** vs. **Fwd-Bwd Bucket Communication Uniformity**.

Intuition of the Control Parameter α: The algorithm constructs a *Target Allocation Vector* v^* for the current bucket by blending two basis vectors:

- v_{fill} **(The History Basis):** Represents a compensatory strategy. It assigns more work to ranks that are currently "underloaded" (have a load deficit) based on previous allocations.
- v_{even} **(The Geometry Basis):** Represents a perfectly uniform partition based on data size ($1/R$). This corresponds to the standard ZeRO-1 strategy.

The parameter $\alpha \in [0, 1]$ controls this blend. If $\alpha = 1$, the algorithm aggressively fills deficits (prioritizing compute balance). If $\alpha = 0$, it ignores history (prioritizing communication uniformity).

Step-by-Step Execution:

1. **Global Sorting (LPT Rule):** We process buckets in descending order of their total cost $\mathcal{W}$. This "Longest Processing Time" (LPT) rule is crucial because large rocks (heavy buckets) are harder to balance and should be placed when the "deficit voids" are most flexible.
2. **Deficit Calculation:** At step t, we calculate the current load vector L. The mean load μ represents the ideal average. The deficit vector $d = \max(0, \mu - L_r)$ identifies which ranks are lagging behind the ideal average.
3. **Target Construction:** We calculate a "fill proportion" vector $v_{fill} = d/\sum d$, which dictates how much of the current bucket should be given to each rank to help them catch up. The final target proportion is $v^* = (1 - \alpha)v_{even} + \alpha v_{fill}$.

Algorithm 1 α-Balanced Greedy LPT Partitioning

Require: Buckets $\mathcal{B}$, Ranks R, Balance Factor α
Require: Load function $\mathcal{W}(\cdot) = numel(p)$
Ensure: Partition vectors $\{\mathbf{s}_i\}$ for all i
1: **Define:** Let $\mathcal{W}^i = \sum_{p \in B_i} \mathcal{W}(p)$ be the total load of bucket i.
2: **Define:** Let $\Phi_i(u) = \sum_{p \in B_i[0:u]} \mathcal{W}(p)$ be the cumulative load up to cut point u.
3: **Initialize:** Global load vector $\mathbf{L} \leftarrow \mathbf{0}_R$
4: **Target:** $\mu \leftarrow (\sum_i \mathcal{W}^i)/R$
5: **Sort:** Virtual reorder $\mathcal{B}$ such that $\mathcal{W}^{\pi(1)} \geq \mathcal{W}^{\pi(2)} \geq \cdots \geq \mathcal{W}^{\pi(N)}$ (LPT based on Load, Virtual Inter-Bucket reorder)
6: **for** $t = 1$ to N **do**
7: Let current bucket index $k = \pi(t)$
8: **Step (1): Calculate Deficits (in Load domain)**
9: $\mathbf{d} \leftarrow [\max(0, \mu - L_r)]_{r=1}^{R}$; $D_{\text{total}} \leftarrow \sum_r \mathbf{d}$
10: **Step (2): Define Basis Vectors**
11: $\mathbf{v}_{\text{even}} \leftarrow [1/R, \ldots, 1/R]$
12: $\mathbf{v}_{\text{fill}} \leftarrow \mathbf{d}/D_{\text{total}}$ if $D_{\text{total}} > 0$ else $\mathbf{v}_{\text{even}}$
13: **Step (3): Blended Target Allocation**
14: $\mathbf{v}^* \leftarrow (1 - \alpha)\mathbf{v}_{\text{even}} + \alpha \mathbf{v}_{\text{fill}}$
15: $\mathbf{target_alloc} \leftarrow \mathcal{W}^k \cdot \mathbf{v}^*$ {*Target load for ranks*}
16: **Step (4): Discretization (Project Load to Valid Cuts, following Atomicity Constraint)**
17: Init $s_{k,0} \leftarrow 0$; $C \leftarrow 0$
18: **for** $r = 1$ to $R - 1$ **do**
19: $C \leftarrow C + \mathbf{target_alloc}[r]$
20: {*Find cut u where cumulative load $\Phi_k(u)$ is closest to target C*}
21: $s_{k,r} \leftarrow \arg\min_{u \in \mathcal{U}_k} |\Phi_k(u) - C|$ {*Update global load with actual load of the slice*}
22: $L_r \leftarrow L_r + (\Phi_k(s_{k,r}) - \Phi_k(s_{k,r-1}))$
23: **end for**
24: $s_{k,R} \leftarrow \text{Size}(B_k)$
25: $L_R \leftarrow L_R + (\Phi_k(s_{k,R}) - \Phi_k(s_{k,R-1}))$
26: **end for**
27: **Return** $\{\mathbf{s}_i\}$ sorted by original indices

4. **Atomic Discretization:** The target vector v^* gives us a continuous "ideal" load. We iterate through the parameter list and place the cut point u strictly at parameter boundaries to minimize the distance to the target cumulative sum, thereby enforcing the **Atomicity Constraint**.

Algorithm 2: Micro-Group Scheduling with Greedy Rollback (TP) For TP, we must partition a stream of tensor updates into discrete Micro Groups (for fused communication) and assign them to Host Ranks (for computation). This is modeled as a Bin Packing Problem with an embedded Multiprocessor Scheduling Problem. Based on our implementation, we define a Two-Level Greedy Strategy detailed in Algorithm 3 and Algorithm 4.

Algorithmic Logic Explanation:

1. **Global vs. Local Sorting:** We perform a Global Sort (Line 5, Alg 3) of all parameters first. This ensures that we attempt to pack the largest, most difficult tensors into groups first, preventing "straggler tensors" from accumulating at the end. Inside the MINHEAPSOLVER (Line 3, Alg 4), we perform a Local Sort as part of the standard Longest Processing Time (LPT) heuristic, which provides a reliable approximation for makespan minimization.
2. **The Simulation & Rollback Mechanism:** The core innovation is the feedback loop between the Bin Packing (Outer loop) and the Scheduling (Inner solver). Instead of estimating the cost of a group by simple summation ($\sum Cost/R$), we run the actual scheduling simulation (MINHEAPSOLVER) at every step (Line 13, Alg 3). This calculates the *exact* makespan L_{max} given the current items. If adding a new item causes L_{max} to exceed C_{max} (due to fragmentation or imbalance), we strictly enforce the limit by triggering a Rollback (Lines

Algorithm 2 Micro-Group Construction for TP load balance with Greedy Rollback

Input: Parameters $\mathcal{P}$, Cost Model $\mathcal{W}(\cdot)$, Ranks R, Cap C_{max}
Output: Sequence of Micro Groups $\mathbb{M}$
{*Phase 1: Deterministic Global LPT Sort*}
$\mathcal{P}_{sorted} \leftarrow \text{Sort}(\mathcal{P}, \text{key} = (\mathcal{W}(p), \text{ID}_p), \text{descending})$
$M_{curr} \leftarrow \emptyset$; $\mathbb{M} \leftarrow \emptyset$
{*Phase 2: Partitioning with Rollback*}
for p **in** $\mathcal{P}_{sorted}$ **do**
  $M_{test} \leftarrow M_{curr} \cup \{p\}$
  {*Min-Heap Solver to find optimal Host Ranks*}
  $L_{max} \leftarrow \text{MinHeapBalance}(M_{test}, R)$
  if $L_{max} \leq C_{max}$ **then**
    $M_{curr} \leftarrow M_{test}$ {Valid: Accept p}
  else
    {*Constraint Violated: Rollback & Finalize*}
    $\mathbb{M}.\text{append}(M_{curr})$
    $M_{curr} \leftarrow \{p\}$ {Start new micro group with p}
  end if
end for
$\mathbb{M}.\text{append}(M_{curr})$
return $\mathbb{M}$

18-26). The current group is finalized, and the overflow item becomes the seed for the next group.

3. **Efficiency:** Although simulating the solver at every step seems computationally intensive, the MINHEAPSOLVER operates in $O(K \log R)$ for K items. Since the number of items per Micro Group (K) is small (typically < 50), this overhead is negligible during the offline planning phase.

Algorithm 3 Micro-Group Construction with Greedy Rollback

Input: Parameters $\mathcal{P}$, Shapes $\mathcal{S}$, Capacity C_{max}, Ranks R
Output: List of Micro Groups $\mathbb{M}_{opt}$ where each $M \in \mathbb{M}_{opt}$ contains assignments $\{(p, rank)\}$
{Phase 1: Deterministic Global Sort}
$\mathcal{I}_{meta} \leftarrow [(\text{Cost}(p), p, s)$ for p, s in zip$(\mathcal{P}, \mathcal{S})]$
Sort $\mathcal{I}_{meta}$ descending by Cost
$\mathbb{M}_{opt} \leftarrow []$
$M_{curr} \leftarrow []$ {Current candidate items}
$idx \leftarrow 0$
while $idx <$ Length$(\mathcal{I}_{meta})$ **do**
  $item \leftarrow \mathcal{I}_{meta}[idx]$
  M_{curr}.append$(item)$
  {Simulation Step: Try to solve partition for current candidate list}
  $Assignments, L_{max} \leftarrow$ MinHeapSolver(M_{curr}, R)
  if $L_{max} \leq C_{max}$ **then**
    {Valid: Item fits. Continue accumulating.}
    $idx \leftarrow idx + 1$
  else
    {Constraint Violated: Trigger Rollback}
    M_{curr}.pop() {Remove the item that caused overflow}
    if M_{curr} is empty **then**
      Error: Single item exceeds C_{max}.
    end if
    {Finalize the previous valid group}
    $FinalAlloc, _ \leftarrow$ MinHeapSolver(M_{curr}, R)
    $\mathbb{M}_{opt}$.append$(FinalAlloc)$
    {Start new group with the current item}
    $M_{curr} \leftarrow []$
    {Do not increment idx; retry item in next iteration}
  end if
end while
{Handle remaining items}
if M_{curr} is not empty **then**
  $FinalAlloc, _ \leftarrow$ MinHeapSolver(M_{curr}, R)
  $\mathbb{M}_{opt}$.append$(FinalAlloc)$
end if
Return $\mathbb{M}_{opt}$

Algorithm 4 Subroutine: MinHeapSolver (LPT)

Input: Items I, Ranks R
Output: Allocation $\mathcal{A}$, MaxLoad L_{max}
Sort I descending by Cost {Local LPT Sort}
PriorityQueue $PQ \leftarrow [(0, r)$ for $r \in 0..R-1]$ {Min-Heap of (load, rank)}
$\mathcal{A} \leftarrow [[]$ for $r \in 0..R-1]$
for each item (c, p, s) in I **do**
  $min_load, r_{best} \leftarrow$ heapq.heappop(PQ)
  $\mathcal{A}[r_{best}]$.append$((p, s))$
  $new_load \leftarrow min_load + c$
  heapq.heappush$(PQ, (new_load, r_{best}))$
end for
$L_{max} \leftarrow \max_{(l,r)\in PQ}(l)$
Return $\mathcal{A}, L_{max}$

D.4 Discussion on the Choice of Training Framework

In the realm of large-scale distributed training, **Megatron** (Shoeybi et al., 2019) and **PyTorch FSDP** (Fully Sharded Data Parallel) (Zhao et al., 2023) represent the two dominant frameworks. In this work, we chose to implement and evaluate our strategies primarily within Megatron for two key reasons: (1) it remains the de facto standard for training massive-scale foundational models (e.g., GPT-3, Qwen) due to its highly optimized tensor parallelism and pipelining capabilities; and (2) its reliance on strict geometric partitioning provided the ideal testbed to demonstrate how our Static Partitioning respects ZeRO-1 Geometric Constraints to preserve coalesced communication overlapping, a feature structurally abandoned by other atomicity-preserving methods.

However, the principles of our proposed framework, particularly the **Load-Balanced Asynchronous Compute (LB-ASC)**, are not limited to Megatron and can be generalized to FSDP. From a high-level system abstraction, the optimizer step in FSDP shares fundamental structural similarities with the Tensor Parallelism (TP) challenge we address in Section 4.

Specifically, both FSDP and TP fragment parameters across devices. To perform a matrix-based optimizer update (which requires strict atomicity), both paradigms necessitates an `All-Gather` operation to reconstruct the full parameter, followed by the computation, and a shard update. Therefore, our Micro-Group Scheduling and Asynchronous Compute pipeline are directly transferable to FSDP. By treating FSDP shards as logical units similar to TP splits, our load-balancing algorithm can similarly effectively neutralize stragglers and maximize throughput in an FSDP-based training stack.

It is crucial to note that adapting our TP LB-ASC strategy to FSDP requires distinct handling depending on the specific ZeRO stage employed.

- **FSDP-ZeRO1 Scenario:** In this configuration, gradients are typically not sharded (they are replicated across ranks within the sharding group). Since our TP LB-ASC primarily serves as a gradient reconstruction mechanism, the `All-to-All` communication depicted in Figure 2 can be skipped.
- **FSDP-ZeRO2/3 Scenario:** Here, gradients are sharded across devices. Consequently, the full execution flow of our TP LB-ASC (including the fused `All-to-All` reconstruction) is necessary within the optimizer step to assemble the required data.

Communication Overhead and Engineering Complexity in Hybrid Parallelism: Our choice is driven by the distinct communication characteristics of Tensor Parallelism (TP) versus Data Parallelism (DP). TP typically leverages high-bandwidth intra-node interconnects (e.g., NVLink); thus, while parameter sharding is unavoidable, performing reconstruction communication within the TP group during the optimizer step is acceptable. In contrast, DP communication usually spans lower-bandwidth inter-node networks, making any synchronization within the DP group prohibitively expensive. (Notably, this bottleneck constitutes the primary drawback of the `layerwise_optimizer` baseline, which necessitates an expensive `All-Reduce` and an additional `All-Gather` operation over these slow links during the step.) A significant challenge arises when superimposing TP onto FSDP-ZeRO2/3: this hybrid setting necessitates a coupled 2D (DP+TP) communication mesh to reconstruct tensors fragmented across both dimensions. This not only forces expensive synchronization over the slower inter-node DP links but also introduces substantial engineering complexity to manage the hybrid topology. Megatron's architecture avoids this by inherently decoupling DP and TP, enabling us to isolate synchronization strictly within the high-speed TP domain while keeping the DP dimension communication-free (via ZeRO-1).

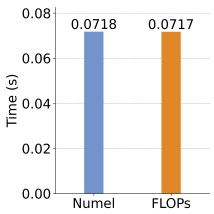


Figure 17: **Cost Metric Ablation.** Comparison of Muon step latency using Numel vs. exact FLOPs cost function. The difference is negligible ($\approx 10^{-4}$s).

D.5 Discussion about Load-Balance Generalization with Nonlinear Cost

While our framework mathematically supports arbitrary cost functions $\mathcal{W}(p)$ to handle the cubic complexity of matrix-based optimizers, in our practical implementation and experiments, we adopted the simplified linear metric $\mathcal{W}(p) = \text{numel}(p)$ primarily to enforce the Unified design philosophy of our framework.We justify this choice based on three key rationales:

- Optimizer-Agnostic Universality: A core contribution of this work is decoupling the system architecture from the specific optimization algorithm. By using numel as the universal cost metric, our partitioner remains strictly agnostic to the optimizer's internal complexity (whether $\mathcal{O}(N)$ or $\mathcal{O}(N^3)$), ensuring that the same infrastructure seamlessly supports Muon, Shampoo, SOAP, and future algorithms without modification.
- Shape-Cost Correlation: For standard Transformer architectures (e.g., Qwen), parameter tensors typically fall into grouped shapes where computational cost correlates strongly with parameter count, making numel an effective proxy.
- Implementation Robustness: This simplification avoids the engineering fragility of maintaining distinct cost models for every new optimizer, reducing system complexity while retaining high load-balancing efficiency as demonstrated in our experiments.

To quantitatively validate the effectiveness of this simplification, we conducted a controlled experiment using the Qwen3-32B model with a parallelism configuration of DP=16 and TP=8.

We compared the execution makespan of the optimizer step when scheduling with the exact computational complexity ($\mathcal{W}(p) = \text{FLOPs}$) versus the simplified parameter count ($\mathcal{W}(p) = \text{numel}(p)$).

As illustrated in Figure 17, the performance difference is negligible: using FLOPs yields a latency of **0.0717s**, while using Numel yields **0.0718s**. The discrepancy is approximately 10^{-4}s. In the context of our broader results in Section C, where the performance gaps between different methods typically range from 10^{-2}s to 10^{-1}s, this minute deviation confirms that the shape-based cost metric (numel) serves as a highly accurate proxy for load balancing. It captures the dominant workload characteristics of Transformer blocks without introducing the engineering overhead of calculating exact FLOPs for every tensor.

E Extended Related Work

To situate our contributions, we classify existing distributed optimization strategies into three categories: (1) **System-Level Compromises** (e.g., Layer-wise Partitioning), which preserve mathematical exactness but sacrifice pipeline efficiency; (2) **Algorithmic Approximations** (e.g., Blocking/Shard-local), which sacrifice mathematical fidelity to fit system constraints; and (3) our proposed Canzona, which reconciles both without compromise.

E.1 The Fundamental Conflict: Matrix-based Optimizers vs. Distributed Atomicity

Modern large-scale training relies on Geometric Sharding (e.g., ZeRO-1, Tensor Parallelism (§ 2)) to distribute memory. Frameworks like Megatron flatten parameters into a contiguous `param_and_grad_buffer` and slice them uniformly (e.g., into $1/N$ shards) regardless of tensor boundaries.

- **The Conflict:** Matrix-based optimizers (e.g., Muon (Jordan et al., 2024), Shampoo (Gupta et al., 2018), SOAP (Vyas et al., 2024), Conda (Wang et al., 2025), ROOT (He et al., 2025), PSGD (Li, 2018), Sophia (Liu et al., 2023), K-FAC (Martens & Grosse, 2015; Osawa et al., 2019), and (Xie et al., 2026)) are inherently non-separable. They require holistic access to the full weight matrix $W \in \mathbb{R}^{d_{in} \times d_{out}}$ to perform operations like SVD or Newton-Schulz iterations.
- **The Gap:** Standard geometric sharding violates this Atomicity Constraint. A single weight matrix is often fragmented across multiple ranks, making local execution of these mathematical operations impossible without costly reconstruction.

E.2 System-Level Attempts: Exactness at the Cost of Efficiency

To preserve the mathematical exactness of the optimizer, recent industrial implementations have adopted coarser-grained partitioning.

Layer-wise Partitioning (e.g., NVIDIA's `layerwise_optimizer`, Distributed Shampoo (Shi et al., 2023)). This approach assigns optimizer states at the granularity of whole layers to ensure atomicity. While mathematically correct, it introduces severe system-level bottlenecks driven by the Geometric Incompatibility (detailed in §3.1):

1. **Structural Fallback to `All-Reduce`:** Because the logical assignment conflicts with the geometric slicing of ZeRO primitives, the system cannot coalesce parameters into a single `Reduce-Scatter` kernel. It is structurally forced to fallback to `All-Reduce` operations ($2\times$ communication volume) and an additional `Broadcast`/`All-Gather` step during the optimizer update to synchronize the new parameters across ranks (as shown in Figure 4 and 7).

E.3 Algorithmic Attempts: Efficiency at the Cost of Fidelity

To avoid the heavy communication of exact reconstruction, another line of research modifies the optimizer algorithm itself, introducing approximations.

Block-Diagonal Approximations (Small-Block Methods). Methods like Distributed Shampoo (Anil et al., 2020) and K-FAC (Osawa et al., 2019) approximate the full preconditioner with a block-diagonal matrix (e.g., approximating a 4096×4096 correlation with smaller blocks).

- **Limitation:** This ignores off-diagonal correlations. While effective for smaller models or specific architectures (e.g., CNNs), this "Small-Block" assumption may degrade convergence speed or solution quality for Large Language Models where dense correlations matter.

Shard-Local Orthogonalization (e.g., MuonBP). Recent proposals like MuonBP (Khaled et al., 2025) attempt to bypass global communication by performing orthogonalization strictly on the local shard of the weight matrix.

- **Limitation:** This constitutes a Mathematical Approximation. The "Local Newton-Schulz" update differs from the "Global Newton-Schulz" update. This creates a Directional Drift between the true gradient geometry and the applied update, which can lead to training instability or require frequent, expensive synchronization steps to correct.

Low-Rank and Subspace Approximations. To circumvent the prohibitive cost of full-rank matrix operations, a recent wave of optimizers proposes constraining the update trajectory to lower-dimensional manifolds to reduce communication and memory overheads. **Dion** (Ahn & Xu, 2025) introduces a distributed orthonormalization technique that replaces global Newton-Schulz iterations with decoupled momentum buffers and low-rank approximations, allowing states to evolve independently on shards without expensive synchronization. Similarly, methods utilizing dynamic subspace selection, such as (Gong et al., 2025), project second-order information onto predefined orthogonal bases of low-rank structures.

- **Limitation:** While these approaches drastically improve efficiency, they impose strong structural assumptions—effectively assuming weight updates lie strictly in a low-rank subspace. This constitutes a fundamental *fidelity trade-off*: unlike system-level attempts, which support the exact holistic update required by the original algorithm, these methods alter the optimization path, potentially degrading convergence quality in regimes where full-rank curvature information is critical.

Overall Limitation: Beyond the potential degradation in convergence fidelity and the need for further verification at scale, these algorithmic approximations fundamentally lack a Unified design

philosophy. Instead of offering a general system solution, they often resort to ad-hoc, algorithm-specific modifications—such as tailoring strategies strictly for Muon (e.g., MuonBP)—which restricts their extensibility to other emerging or future matrix-based optimizers.

E.4 Our Solution: Canzona (System-Level Exact, Efficient, and Unified Optimization)

Our framework, **Canzona**, distinguishes itself by simultaneously achieving Zero-Fidelity-Loss, high system throughput, and broad universality. Unlike prior works that compromise on one or more of these axes, our framework delivers a comprehensive solution:

- **System-Level Exactness (vs. Algorithmic Approximations):** We strictly adhere to the original mathematical definition of the optimizer. By handling the atomicity constraint purely through memory layout and scheduling, rather than altering the update rule (e.g., via blocking or shard-local approximations), we guarantee Zero-Fidelity-Loss, ensuring convergence behaviors identical to single-device baselines.
- **Straggler-Free Efficiency (vs. Layer-wise Partitioning):** We address the system bottlenecks across both parallelism dimensions. For Data Parallelism, our α-Balanced Static Partitioning preserves the computation balance during the optimizer update step. By successfully reconciling the Atomicity Constraint with ZeRO-1 Geometric Constraints, it allows us to fully inherit the efficient, coalesced communication overlapping of the standard ZeRO-1 pipeline and zero communication during optimizer step. Simultaneously, for Tensor Parallelism, our Asynchronous Micro-Group Scheduling eliminates the redundancy of synchronous execution. By batching fragmented updates and balancing computational loads, we effectively hide the expensive reconstruction overhead that limits existing layer-wise approaches.
- **Unified and Agnostic Framework (vs. Specific Hacks):** Crucially, our framework is designed as a Unified framework decoupled from specific optimizer logic. We treat tensor updates as generic computational tasks defined solely by cost metrics (e.g., numel). This makes our system Optimizer-Agnostic, allowing it to support not only Muon but also Shampoo, SOAP, and future matrix-based algorithms without requiring algorithm-specific ”hacks” or re-engineering.