SlimPack: Fine-Grained Asymmetric Packing for Balanced and Efficient Variable-Length LLM Training

Yuliang Liu*†
liuyuliang@kuaishou.com
Kling Infra, Kuaishou Technology

Guohao Wu*
wuguohao03@kuaishou.com
Kling Infra, Kuaishou Technology

Shenglong Zhang*
zhangshenglong@kuaishou.com
Kling Infra, Kuaishou Technology

Wei Zhang
zhangwei45@kuaishou.com
Kling Infra, Kuaishou Technology

Qianchao Zhu
zhuqianchao@kuaishou.com
Kling Infra, Kuaishou Technology

Zhouyang Li
lizhouyang@kuaishou.com
Kling Infra, Kuaishou Technology

Chenyu Wang
wangchenyu05@kuaishou.com
Kling Infra, Kuaishou Technology

Abstract

The efficient distributed training of Large Language Models (LLMs) is severely hampered by the extreme variance in context lengths. This data heterogeneity, amplified by conventional packing strategies and asymmetric forward-backward costs, leads to critical inefficiencies such as cascading workload imbalances and severe hardware underutilization. Existing solutions attempt to mitigate these challenges, but often at the expense of memory or communication efficiency.

To address these challenges, we introduce SlimPack, a framework that fundamentally rethinks data packing and scheduling by decomposing samples into fine-grained slices. This slice-level decomposition immediately mitigates critical memory and communication bottlenecks by transforming large, volatile workloads into a stream of smaller, manageable units. This flexibility is then harnessed for our core innovation, Asymmetric Partitioning, which assembles balanced scheduling units uniquely optimized for the different demands of the forward and backward passes. Orchestrated by a two-phase solver and a high-fidelity simulator, SlimPack holistically resolves imbalances across all parallel dimensions. Extensive experiments demonstrate that SlimPack achieves up to a 2.8× training throughput improvement over baselines, breaking the conventional trade-off by delivering both superior balance and high resource efficiency.

1 Introduction

The relentless growth in the scale of LLMs [7, 29, 37] has been a primary driver of their remarkable capabilities, yet this expansion has simultaneously intensified the challenges of efficient distributed training. A fundamental, yet often overlooked, bottleneck is the extreme variance in sequence lengths found in real-world training corpora [2, 25]. Datasets exhibit a pronounced long-tail distribution [5, 9, 30], where a small fraction of ultra-long sequences contributes a disproportionately large share of the computational workload. Existing distributed systems [15, 18, 34], however, are typically designed with an assumption of workload homogeneity, employing static parallelism and scheduling strategies that are fundamentally misaligned with this data heterogeneity. This mismatch leads to severe hardware underutilization, manifesting as cascading workload imbalances across multiple parallelism dimensions.

To manage variable-length inputs, a common strategy adopted by training systems is sample-level packing [19, 39], which typically involves combining multiple shorter sequences into a single micro-batch of fixed length. While this reduces padding, it creates two critical challenges in hybrid parallel systems. First, due to quadratic cost of self-attention [38], a single long-sequence sample can become a "straggler", dictating the execution time for an entire micro-batch. And in a hybrid system, this creates a powerful amplifier effect: a delay in one data parallelism rank propagates through the pipeline, creating a massive **Cascading Imbalance Bubble** that stalls expensive hardware across all parallel workers. Second, these strategies fail to account for the **Asymmetric Costs** of the forward and backward passes. The specialized mechanism of memory-efficient attention necessitates the re-computation of the attention score during the backward pass. This results in a forward-to-backward computational ratio of approximately $1:2.5$ for the attention mechanism, which differs from the $1:2$ ratio typical of standard GEMM operations. Consequently, a perfectly balanced forward pass inevitably becomes imbalanced during backpropagation, reintroducing the very straggler problem the packing was meant to solve.

Existing solutions for load imbalance force a critical trade-off, typically sacrificing either memory efficiency or communication efficiency to achieve a balanced workload. The first category, `Workload-Aware Scheduling`, seeks to balance

*Equal contribution.
†Corresponding author.

the execution flow by strategically altering the data scheduling order. For instance, WLB-LLM [41] packs micro-batches to balance FLOPs and employs an outlier-delay mechanism. However, this alteration of the inherent data distribution can interfere with the statistical properties of the training process, potentially compromising model convergence. Another category, `Parallelism Hot- Switching`, dynamically reconfigures the system's parallelism strategy or communication topology at runtime [10, 11, 40]. While effective at matching heterogeneous data workloads, this flexibility comes at the cost of significant **Communication Overhead**. Each reconfiguration can trigger a communication burst to reshard inputs or model parameters across different parallel layouts. This issue is particularly acute when an infrequent, ultra-long outlier forces a costly transition. Furthermore, these solutions inherit the inherent memory limitations of sample-level packing. The coarse-grained packing strategy introduces significant memory inefficiency in its pursuit of workload balance, as grouping samples of disparate lengths substantially inflates the peak activation memory footprint.

To overcome these limitations, we introduce **SlimPack**, a framework that fundamentally rethinks data packing and scheduling for variable-length LLM training. Instead of treating samples as indivisible units, SlimPack decomposes them into fine-grained slices. This flexibility allows our system to assemble perfectly balanced scheduling units, which we term **MicroPacks**. Crucially, SlimPack employs **Asymmetric Partitioning**: it creates entirely separate MicroPack configurations tailored to the distinct computational profiles of the forward and backward passes, directly solving the primary cause of pipeline imbalance. This methodology is applied hierarchically to holistically address inefficiencies in hybrid parallelism. First, to prevent system-wide stragglers, the global batch is partitioned across Data Parallel (DP) ranks to equalize their total FLOPs. Second, within each pipeline, the asymmetrically partitioned MicroPacks eliminate internal bubbles by ensuring every stage processes a consistent workload throughout the entire execution cycle. Furthermore, this slice-level decomposition inherently **Mitigates Memory and Communication** bottlenecks, especially for ultra-long sequences. By serializing a long sequence into a stream of smaller MicroPacks, SlimPack transforms prohibitive memory spikes into a low, stable footprint. This memory efficiency enables deeper pipelines without risking Out-Of-Memory errors and reduces the need for costly, communication-intensive context parallelism [15, 24].

SlimPack's complex scheduling is orchestrated by an **Two-Phase Solver** and a high-fidelity **DAG-based Simulator**. The solver first determines the optimal sample distribution across DP ranks and then formulates the slice-level packing as a Mixed-Integer Linear Program to find the ideal configuration. Crucially, it resolves the inherent imbalance between forward and backward passes, while also handling extreme outliers with a novel **DP-Merge** technique. These candidate schedules are then rigorously evaluated by the simulator. By identifying the critical path, the simulator accurately models performance—predicting throughput and revealing pipeline bubbles, to guide the selection of the optimal configuration.

We summarize our main contributions as follows:

- We identify and analyze critical bottlenecks in variable-length training: the cascading imbalance effect in hybrid parallel systems and the workload skew from asymmetric forward-backward computational costs.
- We propose SlimPack, a novel system using a fine-grained, slice-level packing paradigm. Its core is an asymmetric partitioning solver that creates balanced "MicroPacks" for each pass, guided by a high-fidelity DAG-based simulator to find the optimal schedule.
- We demonstrate through extensive experiments that SlimPack achieves up to a **2.8×** training throughput improvement over the state-of-the-art Megatron-LM framework with sample-packing„ showing superior efficiency and scalability on long context training.

Table 1. Summary of key notations.

Symbol	Description
	Parallelism Dimensions
DP	Data parallelism size
PP	Pipeline parallelism size
CP	Context parallelism size
	Batch and Sequence Dimensions
B	Global batch size (total samples per iteration)
b	Micro-batch size (samples per fwd/bwd pass)
M	Number of micro-batches ($M = B/b$)
L	Sequence length of input sample
	Model Architecture
h	Hidden dimension size
N_l	Number of Transformer layers

2 Background & Challenges

Table 1 summarizes the key notations used in this paper.

2.1 Variable-Length LLM Training

The efficiency of large-scale LLM training is fundamentally undermined by the extreme variance in sequence lengths characteristic of real-world data. While existing distributed systems employ various strategies to manage this workload variance, they lack a holistic solution, typically forcing a trade-off that sacrifices memory or communication efficiency to achieve better load balance. This problem manifests as critical workload imbalances, which in turn create severe memory and communication bottlenecks.

2.1.1 Sources of Workload Imbalance

Skewed Data Distributions. Workload imbalance in LLM training is primarily driven by the statistical nature of the input corpora. Large-scale datasets such as *Common Crawl* [30], *GitHub* [5], and *Wikipedia* [9] are characterized by substantial length heterogeneity, typically following a skewed, long-tail distribution. As illustrated in Figure 1, While the vast majority of samples are short (e.g., nearly 80% of samples may be shorter than 4K tokens), a disproportionately large share of the total token workload originates from a tiny fraction of extremely long sequences. For instance, it is common for 1% of the longest samples—often exceeding 128K tokens—to contribute nearly half of the total computational workload. This heterogeneity is particularly problematic in a distributed setting. The presence of a few outlier long sequences creates a classic straggler effect, where the overall execution time for a batch is dictated by these outliers. Consequently, parallel workers assigned shorter sequences are forced into prolonged idle states, leading to severe hardware under-utilization and a significant degradation in end-to-end training throughput.

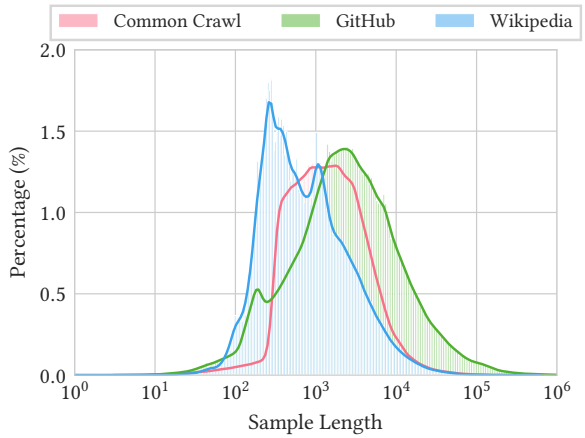

Figure 1. Distribution of sample lengths across pre-training datasets. The pronounced long-tailed pattern is a primary source of workload imbalance.

Quadratic Cost of Attention. The workload imbalance from skewed data is dramatically amplified by the core characteristic of the Transformer: the quadratic complexity of the self-attention mechanism $O(L^2h)$. This property is the primary architectural source of imbalance. It makes the total number of tokens in a micro-batch a poor proxy for its actual workload. While feed-forward networks scale linearly ($O(Lh^2)$), the quadratic cost of attention dominates. Consequently, a micro-batch with a single long sequence has a far greater computational cost than another of the same final length composed of multiple packed shorter sequences. This disparity is a direct cause of the straggler effect.

2.1.2 Training with Long Context

Training LLMs with extended context samples presents unique system-level challenges. To manage the demands, distributed training frameworks rely on a combination of parallelism strategies and specialized techniques.

Data Parallelism (DP). Data Parallelism partitions the data batch across multiple devices, each holding a model replica, and synchronizes gradients via an `AllReduce` operation after local computation. This approach is memory-intensive due to the replicated model and large activation memory from long sequences. To mitigate this, sharded data parallelism techniques like ZeRO [32] partition model states across devices at the cost of increased communication.

Context Parallelism (CP). Context Parallelism [3, 15, 24] directly addresses the activation memory bottleneck by partitioning a single long sequence and its activations across multiple devices. This intra-data partitioning requires substantial communication, such as `Ring`-based [24] or `All-to-All` [15] collectives, to exchange key and value tensors for the distributed attention computation. This communication overhead is a primary performance bottleneck, especially for shorter sequences that are forced to participate.

Pipeline Parallelism (PP). Pipeline Parallelism [8, 12–14, 23, 27, 28] partitions a model's layers across devices (stages) and processes a batch as a stream of smaller micro-batches, requiring only minimal `Peer-to-Peer` communication. While communication-efficient, its performance is extremely sensitive to workload imbalance, as a single straggler micro-batch can create execution bubbles that propagate through all pipeline stages and stall expensive hardware.

Hybrid Parallelism. Modern frameworks [15, 21, 28] combine strategies like DP, PP, and CP into a hybrid approach, often organizing devices into a multi-dimensional mesh. A key limitation of existing systems is their reliance on a static mesh like a fixed DP × CP grid, which is fundamentally misaligned with the heterogeneous workloads of variable-length data. This mismatch motivates recent work [10, 11, 40, 41] on adaptive and flexible parallelism topologies.

Activation Memory Management. Storing activations for the backward pass is a primary memory bottleneck in long-context training. To manage this, activation checkpointing (or recomputation) [4, 16] is a critical technique that trades computation for memory. It frees activations after the forward pass and recomputes them during the backward pass, reducing peak memory usage at the cost of increased computational overhead. Advanced systems [36] can even apply this adaptively based on the workload of each micro-batch or pipeline stage. An alternative approach is activation offloading [10, 32, 33], which moves activations to more abundant host memory. Although constrained by limited PCIe bandwidth, the data transfer cost is often masked by the substantial computation of long sequences.

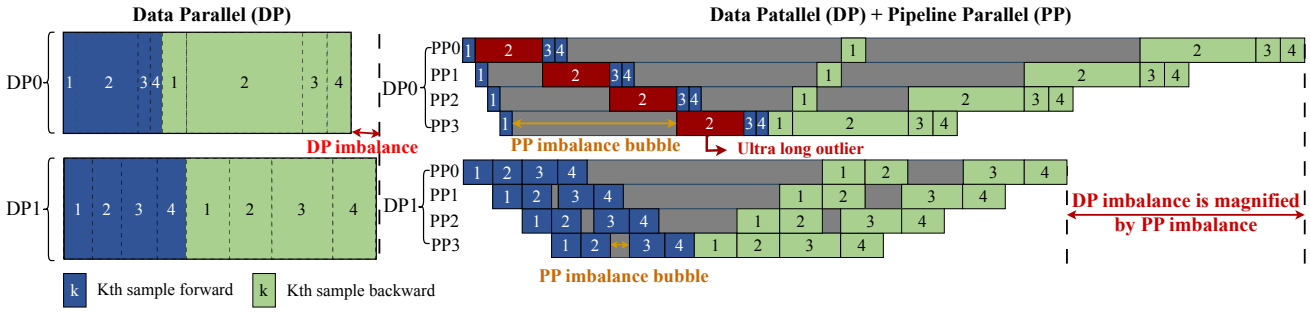

Figure 2. Illustration of the **amplifier effect** in hybrid parallelism. In a pure DP system (Left), minor workload variations between workers create a small, manageable imbalance. In a hybrid DP+PP system (Right), the strict micro-batch synchronization required by PP magnifies these minor delays. A single straggler (e.g., the "ultra long outlier" in DP0) forces subsequent pipeline stages to wait, creating a large **cascading imbalance bubble** and significant hardware idleness.

2.2 Obervation & Challenges

Current training frameworks face significant challenges when handling variable-length inputs, with workload imbalance-induced device idling emerging as the primary bottleneck. This issue is severely exacerbated by the presence of extremely long outliers. While sample-level packing has been adopted as a common solution, its coarse-grained nature fails to fully resolve imbalance issues. Moreover, it introduces substantial memory pressure and communication bottlenecks. Beyond data heterogeneity, we also identify that the computational asymmetry between forward and backward passes in LLM training introduces additional pipeline bubbles, further compounding the imbalance problem.

2.2.1 Challenges from Sample-Level Packing

The conventional approach of treating each packed sample as an indivisible unit of work presents significant challenges in cascading imbalance bubbles, memory consumption, and communication overhead.

The Amplifier Effect of Cascading Imbalance. While a pure Data Parallelism system can tolerate some workload variance between workers, the strict, fine-grained synchronization required by Pipeline Parallelism creates a powerful amplifier effect. As illustrated in Figure 2, the presence of an ultra-long outlier sample induces a substantial processing delay. This delay propagates downstream through the pipeline, starving subsequent stages of input and culminating in a severe pipeline imbalance bubble. Furthermore, such a straggler micro-batch does not only stall its own Data Parallel group. It forces all peer devices to idle during the gradient synchronization phase, waiting for the straggler to complete its computation. This mandatory synchronization induces severe resource underutilization, thereby amplifying the performance penalty across the entire system. Coarse-grained sample-level packing lacks an efficient mechanism to partition extreme outliers without resorting to costly cross-machine communication. The resulting localized pipeline bubbles then propagate globally via the aforementioned amplifier effect, causing a dramatic drop in end-to-end throughput.

Memory Pressure and Communication Bottlenecks. This coarse granularity also exacerbates memory and communication challenges. A significant memory trade-off stems from the pipeline parallelism requirement to maintain p computational units concurrently in flight (where p denotes the pipeline parallelism size), which directly determines the peak memory footprint. As sample-level packing inherently constructs larger composite samples, it leads to substantial memory expansion. The situation becomes critical with ultra-long outliers: matching the outlier's length strains memory, but matching its computational FLOPs under quadratic-attention complexity can create prohibitively long, memory-catastrophic packs. Furthermore, the extreme memory footprint may necessitate the use of sequence parallelism to process a single outlier. This can force the corresponding parallel group to span multiple nodes, thereby triggering substantial and inefficient cross-node communication, which further degrades overall training throughput.

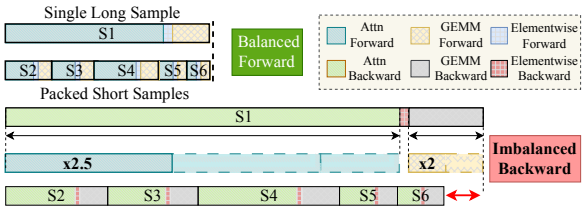

Figure 3. Imbalanced backward pass from a balanced forward pass. Despite a balanced forward pass achieved by packing shorter sequences (S2–S6) with a long sequence (S1), the backward pass exhibits significant imbalance. This is due to the higher computational cost of attention (×2.5) and GEMM (×2) operations during backward pass.

2.2.2 Imbalance from Asymmetric Fwd/Bwd Costs

A more subtle but equally critical challenge arises from the fact that balancing the forward pass does not guarantee a balanced backward pass.

The Asymmetric Cost Profile. The computational costs of the forward and backward passes are inherently asymmetric. While memory-bound operations are balanced, compute-bound kernels are heavier in the backward pass. Specifically, GEMM operations are roughly twice as expensive in backward due to dual gradient computations for activations and weights, while attention kernels like FlashAttention [6] are roughly 2.5× more expensive because they must recompute some activations that were discarded during the forward pass. This creates a fundamental asymmetric cost for any given micro-batch, where the backward pass consistently demands a larger share of the computational resources.

Failure of Forward-Balanced Packing. This cost asymmetry has a critical implication: any sample-level packing strategy that perfectly balances the forward pass will inevitably become imbalanced during backpropagation. The initial balance achieved during the forward pass is merely a temporary veneer that shatters once gradient computation begins. As illustrated in Figure 3, a micro-batch `Pack1` containing a long sequence `S1` can be perfectly balanced with `Pack2` (containing shorter sequences `S2`–`S6`) during the forward pass. However, during the backward pass, the 2.5× cost multiplier is applied to the already quadratically larger workload of `S1`. This causes the absolute increase in `S1`'s computational cost to far exceed the combined increase from all shorter sequences, turning `Pack1` into a severe straggler. Since the composition of a micro-batch is static and cannot change between passes, this "hidden" imbalance reintroduces the very inefficiency the packing was designed to solve, becoming a new, unaddressed source of pipeline bubbles and hardware idleness.

3 Existing Solutions and Limitations

The landscape of existing solutions reflects a sustained effort to contain the inefficiencies caused by variable-length inputs. As summarized in Table 2, these approaches can be broadly categorized by their primary intervention, ranging from workload-aware scheduling to parallelism hot switching. However, none fully resolves the complex interplay between memory pressure, communication cost, and load imbalance.

3.1 Sample Packing

Length-Based Strategies. Length-based sample packing groups sequences of comparable or bounded maximum length to minimize padding. This strategy mitigates the issue of non-uniform computational load across samples, as the workload is positively correlated with the total sequence length. Nonetheless, due to the quadratic computational complexity of the self-attention mechanism, a significant disparity in computational load persists.

Table 2. Comparison of existing methods for variable-length LLM training. We evaluate each approach across memory efficiency, communication overhead, sampling order losslessness, and multi-level load balancing capabilities. A detailed discussion can be found in §3. (✓: Effective, X: Ineffective/Unaddressed)

Category	Work	Memory Effic.	Comm. Effic.	Lossless	Load Balancing		
					DP	PP	Fwd/ Bwd
Workload-Aware Scheduling	WLB-LLM	X	✓	X	✓	✓	X
	DynaPipe	X	✓	✓	✓	✓	X
Parallelism Hot Switching	FlexSP	X	X	✓	✓	X	X
	ByteScale	X	X	✓	✓	✓	X
	HotSPa	X	X	✓	✓	✓	X
Ours Method	**SlimPack**	✓	✓	✓	✓	✓	✓

TFLOPs-Based Strategies. An intuitive solution to workload imbalance is TFLOPs-based sample packing, which aims to equalize the theoretical floating-point operations per batch, accepting memory imbalance as a trade-off. The naive application of this method fails on real-world datasets characterized by long-tailed length distributions. The quadratic complexity of attention means that accommodating extreme length outliers forces shorter sequences to be packed into impractically long composites, violating hardware memory limits. Additionally, a key theoretical limitation arises when a single outlier's workload exceeds the capacity of a standard batch, rendering perfect balance unattainable.

Although both packing strategies improve hardware utilization, they operate at a coarse, sample-level granularity. By treating each sample as an indivisible unit, they cannot prevent a single long sequence from becoming a straggler, which then triggers the cascading imbalance in hybrid DP+PP systems. Furthermore, these packing strategies are suboptimal as they are oblivious to the asymmetry between forward and backward computation costs, using a single strategy for both phases. The following solutions are grounded in the TFLOPs-based packing framework. These solutions can be broadly categorized into two complementary strategies: **Parallelism Hot Switching** and **Workload-Aware Scheduling**.

3.2 Workload-Aware Scheduling

This class of solutions focuses on the data-preparation stage, reconstructing micro-batches to mitigate imbalance.

The FLOPs-only packing objective does not ensure phase balance or memory efficiency with ultra-long outliers. In response, **WLB-LLM** [41] employs an outlier-delay mechanism that departs from the standard data order, caching outliers for later processing to preserve training efficiency. While effective for load balancing, this strategy introduces

a risk to statistical convergence by altering data stochasticity, presenting a fundamental trade-off between system performance and model integrity.

DynaPipe [17] introduces an adaptive scheduling mechanism to mitigate pipeline bubbles caused by imbalanced micro-batches. It achieves this by dynamically reordering the execution sequence of micro-batches within a single training iteration, a design that preserves model correctness. The core idea involves trading off memory for throughput by warming up with more forward passes. However, this very trade-off limits its efficacy against outliers, as it becomes constrained by hardware memory capacity and cannot fully eliminate the bubbles induced by severe computational imbalance.

3.3 Parallelism Hot Switching

This category of solutions dynamically alters the parallelism configuration or execution strategy at runtime in response to workload variations.

The core innovation of **FlexSP** [40] lies in its creation of heterogeneous Sequence Parallelism groups tailored to sample length. By allocating longer sequences to larger SP groups, it strives to balance the computational load among packed samples. To handle extreme outliers, FlexSP employs cross-node sequence parallelism to match their workload to that of other samples, albeit at the cost of significant communication overhead. In terms of parallel strategy, a notable drawback is its lack of support for Pipeline Parallelism, restricting its applicability to broader model architectures.

ByteScale [10] proposes Hybrid Data Parallelism (HDP) to address the challenge of load imbalance in training scenarios that incorporate Pipeline Parallelism. The framework provides two balancing strategies—DP Balance and PP Balance—as trade-off solutions. To handle outliers that induce significant computational skew, ByteScale increases the degree of Sequence Parallelism, a strategy similarly adopted by FlexSP. However, this approach for managing extreme cases introduces substantial communication overhead, mirroring a key limitation observed in FlexSP.

HotSPa [11] advances beyond adaptive data parallelism by dynamically switching the entire hybrid parallelism strategy—including model parallelism dimensions like Tensor Parallelism—at runtime. It dynamically applies distinct hybrid parallel strategies to sequence length based mini-batch partitions, supported by a graph compiler and a hot-switch planner that manages real-time parameter/gradient re-sharding across strategies from a shared model storage. Although this approach maximizes the theoretical optimization space by assigning an ideal parallel strategy to each data subset, it does so at the cost of substantial communication overhead. The frequent reconfiguration of the hybrid parallel plan, especially when it involves cross-machine Pipeline Parallelism, introduces inter-node communication costs that dramatically exceed those inherent to the methods of FlexSP and ByteScale.

In summary, as summarized in Table 2, no existing method holistically resolves the core challenges outlined in Section 2.2 without introducing significant trade-offs. Critically, it must be emphasized that parallelism hot switching aims to balance computational cost, not memory usage. This often results in allocating a larger sequence parallel groups to an individual long sequence that are computationally intensive but shorter in their packed form. Consequently, hot switching does not address the problem of memory usage.

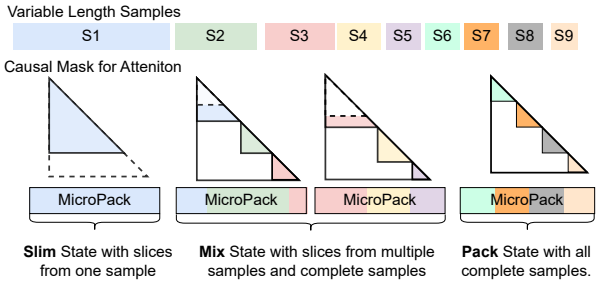

Figure 4. MicroPack formation under a uniform FLOPs budget. Variable-length samples (S1–S9) are transformed into three formation states: Slim—consecutive slices from one long sample; Mix—leftover slices from a slimmed sample co-packed with complete short samples or additional slices (from other samples, order preserved) to meet the budget; Pack—all complete short samples. Note: The attention masks and slice areas are drawn solely to illustrate intra-sample slice dependencies; they are not to scale.

4 SlimPack System

In this section, we present SlimPack, a zero-communication-overhead, and memory-efficient framework for training LLMs with highly heterogeneous, variable-length inputs. SlimPack introduces a principled, aggressively optimized scheduling–packing co-design that (i) dramatically reduces pipeline idle time, (ii) rigorously preserves slice-level packing dependencies, and (iii) substantially lowers peak activation memory.

MicroPack in SlimPack system serves as the minimal schedulable, budgeted container. Under a uniform FLOPs budget per MicroPack, it admits three formation states: Slim — only consecutive slices from a single long sample; Mix — leftover slices from a previously slimmed sample co-packed with complete short samples or additional slices (i.e., consecutive slices drawn from other samples, preserving their per-sample order) to meet the budget; and Pack — filled entirely with complete short samples, akin to regular sample-level packing. A single sample may span multiple MicroPacks, and its slice order is always preserved as shown in Figure 4. Note:

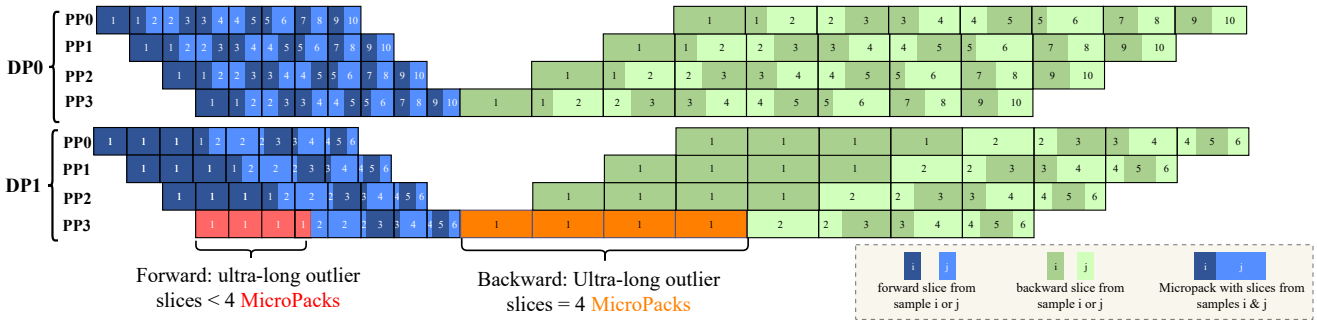

Figure 5. SlimPack's PP schedule for 16 samples across two pipelines (10 for DP1 and 6 for DP2), each using 8 MicroPacks. During backpropagation, slices are reallocated for asymmetric partitioning to rebalance workloads. For instance, sample 1 is split differently during forward and backward passes to balance MicroPack loads. Decomposing a long sequence into slices transforms its high-variance compute cost into many low-variance chunks, ensuring each pipeline stage processes near homogeneous workloads and thus prevents straggler-induced bubbles.

a MicroPack may re-group slices during the backward pass to address forward–backward load asymmetry.

the overall SlimPack framework comprises three MicroPack-oriented components: **Asymmetric Partitioning Solver** distributes global batch samples across DP groups, conditionally applies DP-Merge for extreme outliers, and determines each sample's slim/pack state, thereby producing a family of feasible MicroPack configurations. Building on the chosen configuration, the **PackFlow Pipeline Parallelism scheduling** parameterizes the mapping and ordering of MicroPacks across stages—equalizing per-stage compute, limiting GPU memory peaks, and reusing existing DP/PP/CP collectives with zero additional communication. Given a PackFlow-instantiated schedule, **DAG-based Simulator** faithfully captures data dependencies, memory pressure, and pipeline bubbles, quantitatively evaluate and compare the scheduled candidates, enabling principled selection of the configuration that maximizes end-to-end efficiency under highly variable sequence lengths and finally dispatches the selected schedule for runtime execution.

4.1 Balanced Packing & Asymmetric Partitioning Solver

We formulate LLM packing as a step-time minimization problem under per-rank memory and compute budgets. Exploiting autoregressive causality, sequences are sliced into primitives and assembled into fixed-budget MicroPacks. Because memory scales roughly linearly with sequence length while attention/compute grow superlinearly, the solver balances packing granularity against end-to-end efficiency by providing equal memory budgets across DP ranks and packing compute locally, keeping model/optimizer states stationary, and controlling pipeline bubbles via the MicroPack count rather than inflating to the global worst-case sequence length.

4.1.1 Two-Phase Objective Formulation

Leveraging the observations from §2.1.1, we split the solving process into two phases with corresponding objectives.
Phase 1. Uniform per-DP capacity assignment.

We initialize a compute-balanced target by assigning each DP rank the same forward-FLOPs capacity:

$$C_{\text{dp}_i}^{\text{flops}} = \tfrac{1}{DP} \sum_{x \in \mathcal{B}} f(x), \quad f(x) = f^{\text{fwd}}(x)$$

where $f^{\text{fwd}}(x)$ denotes the forward FLOPs of sample x. Samples in the global batch are sorted by required FLOPs and then distributed across DP groups to approximate this uniform capacity.

Phase 2. Intra-DP slim/pack strategy to reach local workload balance.

After each DP group receives its assigned samples, this DP forward-pass budget is split across m MicroPacks, giving each MicroPack a target $\tau_{\text{MicroPack}}$ of

$$\tau_{\text{MicroPack}} = \frac{\sum_{x \in dp_i} \text{f}^{\text{fwd}}(x)}{m}, \quad m = i \times p, \quad i \in \mathbb{Z}^+.$$

The solver apples *slim* (slice-level partitioning) to long sequences, distributing slices from the sample across multiple MicroPacks so that each pack's workload is close to $\tau_{\text{MicroPack}}$; for short samples, we *pack* (sample-level packing) multiple samples into a single MicroPack until the per-pack budget is met.

Note that each DP replica may choose a different m; accordingly, the per-replica MicroPack target can vary across replicas. Since our workload analysis relies on forward FLOPs, we model it as a Mixed-Integer Linear Program (MILP) problem to translate FLOPs targets into sequence lengths and thereby determine the exact slice boundaries.

DP-Merge: On-Demand Execution Regime for Ultra-Long Samples.

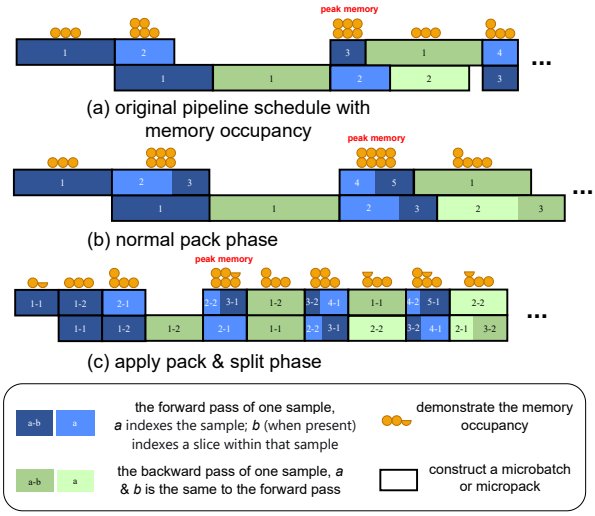

Figure 6. The illustration of the memory consumption of Microbatch/Micropack under three scheduling strategies. In the original 1F1B schedule (a), Samples 1 and 2 hold 3 and 2 memory units, whereas Samples 3-5 use only 1. An optimized schedule with a packing phase (b) reduces idle time but increases peak memory. In contrast, applying a split and pack phase (c) achieves both reduced idle time and lower peak memory consumption.

While the *slim* and *pack* mechanisms work well for typical workloads, rare extreme long outliers $x^\star$ with FLOPs $f(x^\star) \gg C^{\text{flops}}_{\text{dp}_i}$ can behave as stragglers that inflate the system step time. Importantly, in the packing regime, the MicroPack closest to the per-GPU memory limit is not necessarily the one incurring the largest FLOPs; memory and compute outliers need not coincide. To neutralize such computation outliers, we propose a novel technique called **DP Merge**. When the outlier $x^\star$ appear on dp_i , we form a group $G \subseteq DP$ with size $|G| = g$ $(1 \le g \le DP)$. The merged ranks operate as a *super-DP* and a context-parallel with $CP = g$ is applied as it provides the $1/g$ per-rank scaling. The addition context parallel dim g_{cp} spread $x^\star$ accross rank, reducing the *per-rank* costs to

$$f_{\text{eff}}(x^\star) = \frac{f(x^\star)}{g_{\text{cp}}} \le \min_{j \in \text{DP}\setminus G} C^{\text{flop}}_{\text{DP}_j}.$$

Because model and optimizer states remain local (no cross-rank state reshuffling), DP Merge is practical as an on-demand mitigation for extreme outliers and it mitigates the straggler effect, bringing the affected rank's per-step compute into near alignment with stage-level and DP-level capacities.

4.1.2 Asymmetric Partition.

Although slice-level scheme equalizes forward-pass variance, FlashAttention's backward recomputation obeys a steeper

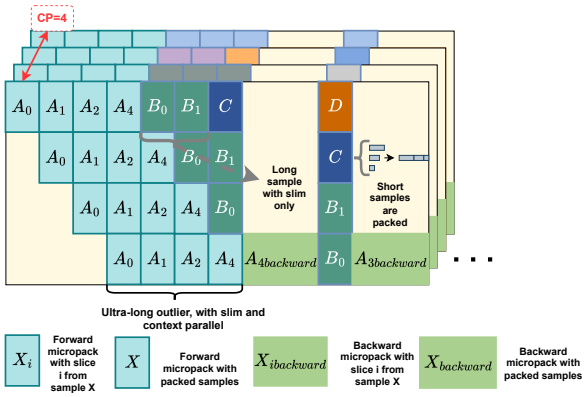

Figure 7. Under DP-Merge, a super-DP with context-parallel size 4 (CP=4) slices the ultra-long outlier into MicroPacks across four devices. All other samples follow SlimPack (*slim+pack*); long sequences may be *slim*-sliced into multiple MicroPacks and executed sequentially on the local device, whereas short sequences are packed into a single MicroPack as the per-pack budget permits.

cost curve and reintroduces imbalance as revealed in sec 2.2.2. To address this imbalanced backward workload, we precompute each sample's backward FLOPs and store its backward optimal slice-level partitioning strategy associated with the given number of MicroPacks without breaking the relative order of MicroPacks. At runtime, when the pipeline transitions to the backward phase, MicroPacks will dynamically apply this strategy, re-dividing slices and re-selecting samples so that each MicroPack carries an equal backward workload. The asymmetric partition and slice-level packing scheme breaks down high-variance cost of large samples into numerous low-variance chunks, distributes them across balanced MicroPacks, and thus eliminates pipeline imbalance bubbles.

Finally, the solver generates multiple candidate configurations with different number of microbatches, and each of them is evaluated by a high-fidelity simulator modeling per-slice runtimes data dependency, load-imbalance bubbles, warmup or cooldown overhead, and overall GPU memory usage. The configuration with the highest estimated end-to-end throughput under the memory constraints is chosen for production-scale training runs.

4.1.3 System-Level Analysis of SlimPack

Alleviating Memory Pressure. Due to the scaling disparity of memory $(O(Lh^2))$ and computation/attention $(O(L^2h))$, naïve global packing forces short sequences to match the longest sequence's FLOPs within a microbatch, as shown in Figure 6(b) , driving per-stage peak activations toward the global worst case. Under SlimPack system, each DP rank constrains its own activation footprint by tuning its MicroPack budget and MicroPack count.

Long samples are sliced into MicroPacks and executed sequentially, as shown in Figure 6(c) . From the pipeline perspective, this ensures that peak memory in the warm-up phase is accumulated for only a single sample at a time, rather than retaining multiple samples—including outliers—as in conventional microbatch execution.

Eliminating DP/PP Bubble. The primary source of DP-level imbalance is ultra-long samples that overshoot a rank's FLOPs/memory budget and become step-time stragglers. When such an outlier is detected, DP-Merge aggregates multiple DP ranks into a temporary super-DP and applies context (Ring/Ulysses) parallelism, spreading the outlier's sequence across g ranks so each rank's exposed FLOPs/activations drop by $\approx 1/g$. This neutralizes the straggler and restores global DP compute balance without moving model/optimizer states. On the pipeline side, SlimPack keeps bubbles controllable along two axes: (i) warm-up/cool-down bubbles shrink as long samples are slimed into multiple MicroPacks thereby increasing the MicroPack count m and reducing the warm-up fraction, and (ii) stage-imbalance bubbles are mitigated by balanced packing/partitioning. Our asymmetric partitioning solver, introduced in 4.1.2, and fixed-budget MicroPacks equalize forward/backward work across stages. In combination, DP-Merge handles outlier-induced DP skew, while SlimPack's MicroPack granularity and asymmetric packing suppress both warm-up and imbalance bubbles.

Reducing Communication Overhead. Splitting a long sequence into multiple MicroPacks does not introduce extra communication beyond the baseline inter-pp-stage activation transfers. Slicing raises the number of stage-to-stage send/recv events, but the aggregate bytes are invariant; under steady-state 1F1B, the extra per-message latency is overlapped and is negligible for practical m. When an extreme outlier triggers DP-Merge, context (Ring/Ulysses) parallelism is applied only to the outlier's MicroPacks, while all other samples continue under the standard SlimPack schedule — introducing no additional communication and keeping the pattern at the baseline minimum as shown in Figure 7.

4.2 PackFlow Pipeline Parallelism Scheduling

Figure 5 illustrates the overall SlimPack pipeline schedule. Both our simulator's analysis and the real run-time training adhere to this execution plan. To facilitate clarity and understanding of the SilmPack pipeline parallel workflow, we establish the following basic conventions:

- **MicroPack Execution Granularity:** Each MicroPack is the smallest scheduling unit in the pipeline, and the sum of FLOPs within each MicroPack is roughly equal across all MicroPacks, ensuring load-balanced forward and backward computation.
- **Intra-Pass Forward Dependency:** Within the forward pass of one sample, slices must be computed in their original order to preserve the causal property of the model.
- **Inter-Pass Backward Dependency:** The backward pass for a sample can only begin after all of its slices have completed their forward pass.

Echoing the concept of microbatches, MicroPacks traverse the entire PP workflow, covering both forward and backward passes. Each stage computes activations and sends them downstream by point-to-point communications. The solver ensures that every MicroPack carries a balanced forward workload, and the pipeline enforces strict preservation of each sample's slice order, guaranteeing correct autoregressive semantics. We employ the KV cache technique [22, 31] to achieve high-throughput slice-level causal attention calculation. However, reusing the forward-pass packing during the backward pass would reintroduce imbalance as described in 3. Instead, once the forward pass completes, we dynamically regroup sample slices into new backward MicroPacks according to the solver's strategy (detailed in §4.1), ensuring each MicroPack meets its backward-FLOP target.

The MicroPack concept, built from asymmetrically partitioned and packed slices, seamlessly integrates into the 1F1B pipeline with minimal modifications. In a standard 1F1B setup, *microbatches* are issued in FIFO (first-in, first-out) order so that each backward pass immediately follows its forward pass, eliminating activation buffering and reducing memory overhead. In SlimPack, we replace microbatches with *MicroPacks*. MicroPacks preserve FIFO order through the pipeline; nevertheless, within each sample's slices, they follow FILO (first-in, last-out) order for correct gradient calculation, as present in Figure 8. Before each iteration starts, samples assigned to a data-parallel replica are sorted in descending order of their forward-pass FLOP cost and then sliced or grouped into MicroPacks such that, once the sample slices' *backward dependency* is met, the backward pass can launch immediately under the 1F1B protocol. However, because backward MicroPacks must satisfy specific backward-FLOP targets to guarantee balance, they can occasionally still falls short of backward-pass FLOPs requirement. In this scenario, we inject an extra forward MicroPack to increase the workload and maintain uninterrupted pipeline flow as shown in Figure 9.

4.3 DAG-based Pipeline Simulator

To evaluate diverse packing solutions and determine optimal pipeline scheduling strategies, we developed a sophisticated pipeline simulator as our performance model. Traditional analytical performance models [20, 42] often simplify calculations by neglecting inter-batch execution time variations and disparities between forward and backward pass durations, potentially yielding inaccurate estimations. While recent simulation-based approaches employing techniques like dynamic programming [26] offer improved accuracy, they

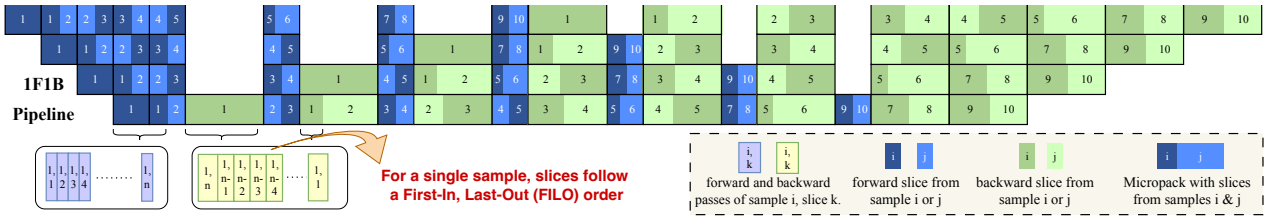

Figure 8. The 1F1B pipeline schedule of SlimPack system. Ten samples are partitioned into slices and sorted into eight MicroPacks for both forward and backward passes. MicroPacks enter the pipeline in FIFO order at the sample level, but within each sample, slices follow FILO order for correct gradient computation.

typically operate with a microbatch-level granularity, limiting their applicability to finer-grained pipeline schemes [23]. Our DAG-based simulator addresses aforementioned issues, achieving higher fidelity in estimating both execution time and memory footprints.

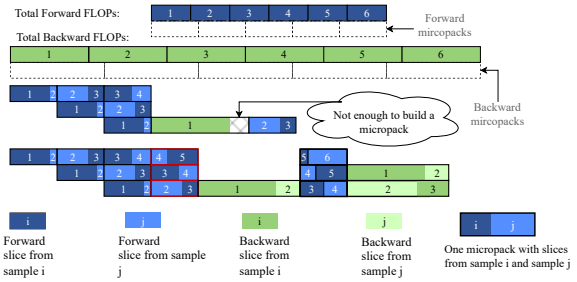

Figure 9. Pipeline schedule illustrating the injection of an extra (red) MicroPack to satisfy backward load-balancing requirements. Six samples are divided into five forward MicroPacks and five backward MicroPacks. The backward flops from sample 1 are not enought to build one backward MicroPack and therefore, the backward of sample 1 is delayed (after forward MicroPack 2 shown in red box for last pp stage)

4.3.1 DAG-based pipeline representation

We represent the entire pipeline execution as a graph $G = (V, E)$. This representation explicitly captures task dependencies, which are crucial for accurately modeling fine-grained schedules.

Vertices (V): tasks. Each vertex $v \in V$ represents a computation or communication task, defined by a tuple `(model_id, action, data_id)`:

- `model_id`: Identifies the set of model parameters used in the task. For layer-partitioned pipeline parallelism, this typically corresponds to the pipeline stage ID (`stage_id`) assigned to a specific device rank.

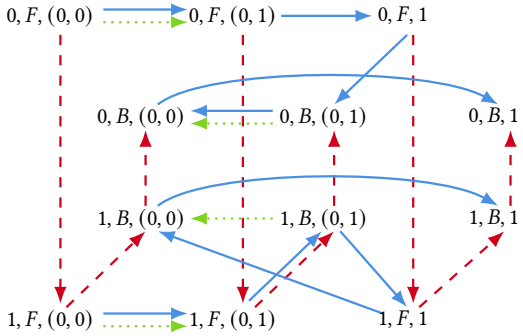

Figure 10. An DAG-based representation of a 1F1B pipeline with 2 stages and 2 input microbatches, with the first microbatch divided into 2 slices. Each node represents a pass `(stage_id, action, data_id)`. Red dashed lines represent **inter-stage dependencies**, green dotted lines represent **inter-slice dependencies** and blue solid lines represent **schedule dependencies**.

- `action`: Specifies the type of computation, such as Forward (F) or Backward (B). Other actions like communication, offloading and recomputation could also be incorporated as needed.
- `data_id`: Uniquely identifies the data unit being processed. This can be a `microbatch_id` or a composite identifier like `(microbatch_id, slice_id)` for finer granularities.

Each vertex v has a weight $w(v)$ representing its estimated execution cost (time), obtained through profile-based performance modeling.

Edges (E): dependencies. An edge $e = (u, v) \in E$ represents a dependency, indicating that task v cannot begin until task u completes. The graph must be acyclic—a cycle would imply a circular dependency, leading to a deadlock in the execution.

The set of edges encapsulates the ordering constraints imposed by the data flow and the pipeline schedule itself. We construct E by considering the union of the following 2 types of dependencies.

1. Data dependencies (E_{data}): These arise from the inherent data flow in LLM training.

- **Inter-stage dependencies:** Data flows sequentially through stages (shown as red dashed lines in Figure 10). For a given data unit identified by `data_id` id and a pipeline with s stages, we recognize the following dependency chain:

$$(0, F, id) \rightarrow (1, F, id) \rightarrow \cdots \rightarrow (s-1, F, id) \rightarrow (s-1, B, id) \rightarrow \cdots \rightarrow (1, B, id) \rightarrow (0, B, id)$$

- **Inter-slice dependencies:** Causal attention mechanism introduces dependencies within a microbatch when sliced (shown as green dotted lines in Figure 10). Forward pass of a slice depends on KV caches from prior slices, while its backward pass depends on KV gradients from subsequent slices.

2. Schedule dependencies (E_{schedule}): The pipeline schedule dictates the execution order of tasks assigned to a certain pipeline rank (shown as blue solid lines in Figure 10). These can be generated according to the scheduling strategy of the specific pipeline scheme. Taking 1F1B schedule as an example, it first fills the pipeline with a sequence of forward passes, then enters a steady-state phase, where it interleaves the execution of a single backward pass with a forward pass. Finally, a cool-down phase drains the pipeline by executing all remaining backward passes for in-flight microbatches.

The full dependency set is $E = E_{\text{data}} \cup E_{\text{schedule}}$. This formulation ensures the DAG accurately models all ordering constraints.

4.3.2 DAG resolution

Once $G = (V, E)$ is constructed with task execution times $w(v)$ associated with each vertex v, we can estimate the total pipeline execution time. This corresponds to finding the *critical path* in the DAG. We use a standard algorithm based on topological sorting and edge relaxation.

Let $Start(v)$ be the start time for task v, and $Finish(v)$ be its finish time. We have:

$$Start(v) = \max(\{0\} \cup \{Finish(u) \mid (u, v) \in E\}) \quad (1)$$

$$Finish(v) = Start(v) + w(v) \quad (2)$$

The total execution time is the maximum finish time over all tasks: $T_{\text{total}} = \max_{v \in V}\{Finish(v)\}$. Algorithm 1 outlines the procedure, which runs in $O(|V| + |E|)$ time. It effectively computes the start and end time of every task in the schedule.

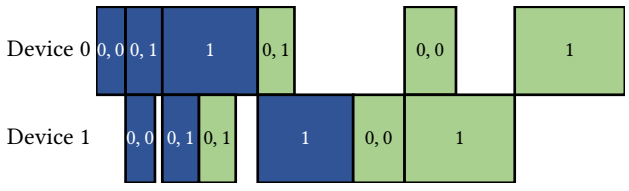

Figure 11. A pipeline schedule generated by the simulator via solving the DAG illustrated in Figure 10. Time costs of passes are intentionally set to introduce extra bubbles.

Algorithm 1 Execution timeline calculation on the DAG

```
procedure CalculateExecutionTimeline(G, w)
    Start(v) ← 0 for v ∈ V
    topological_order ← TopologicalSort(G)
    for u in topological_order do
        Finish(u) ← Start(u) + w(u)
        for each (u, v) originating from u do
            Start(v) ← max(Start(v), Finish(u))
        end for
    end for
    T_total ← max_{v∈V}{Finish(v)}
    return Start, Finish, T_total
end procedure
```

Figure 11 visualizes a schedule derived from solving the DAG in Figure 10. The task costs $w(v)$ in this example were constructed to create pipeline bubbles, which the simulator correctly identifies by computing the critical path.

4.3.3 Memory footprint simulation

Accurate memory estimation is crucial for identifying feasible schedules, especially under long-context scenarios. Prior models often operate at a microbatch granularity and assume a stable memory footprint during the pipeline steady phase, failing to capture the dynamics introduced by slice-level scheduling, imbalanced partitions, and techniques like activation checkpointing and offloading.

Our DAG-based simulator enables fine-grained memory tracking. Using the execution timeline obtained previously, we simulate memory allocation and deallocation events over time. The simulation tracks the live size of:

- Model weights and optimizer states.
- Activations allocated during forward tasks, held until consumption by the corresponding backward tasks.
- KV cache and gradients managed on a per-slice basis according to the pipeline schedule.
- Temporary buffers used during recomputation and offloading.

The simulation proceeds chronologically. Memory for activations is partially freed based on recomputation schedules or explicit offloading events. This event-driven approach allows precise tracking of instantaneous memory usage, thus enabling adaptive offloading strategies as well.

5 Implementation

Our SlimPack is built on top of Megatron-LM [35], the prevalently used parallel training framework for transfromer-based models. Computation is carried out using PyTorch as its backend.

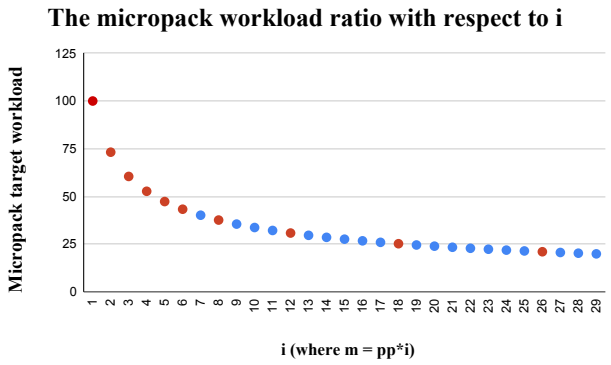

Figure 12. Given a fixed total workload W, the number of micropacks m sets the per-pack budget to W/m. Changing m therefore alters pack granularity and, consequently, the steady-state pipeline-bubble fraction.To prune the search space, we quantize m to multiples of the pipeline-parallel width PP, i.e.,$m = i \cdot \text{PP}$ with $i \in \mathbb{Z}_{>0}$. We then evaluate only these candidates (the red dots in Figure)

5.1 Primitive Measurements and Search Space

As introduced in §4.1, given the theoretical FLOPs per iteration, the SlimPack solver converts them into estimated runtime so the simulator can estimate the end-to-end time cost. However, this conversion's accuracy hinges on each operator's hardware utilization. To compensate, we run preliminary benchmarks to characterize the peak compute-throughput utilization for compute-bound kernels (e.g., GEMM and Flash Attention) and profile the memory-bandwidth utilization for memory-bound operators (e.g., RMSNorm), then use those profiles to calibrate our runtime estimates.

When sweeping the micropack factor i to determine the number of micropack for a given DP workload, we observe that the budget per microbatch decreases roughly logarithmically as i increases. To narrow the search space and avoid evaluating configurations with negligible impact, we sample i values using a modified logarithmically spaced scheme as shown in Figure 12.

5.2 Zero-Overhead Runtime Integration

The SlimPack solver is integrated directly into the PyTorch data sampler, allowing the packing plan to be computed on the CPU side. To achieve low-latency and high-throughput multiple micropack assignments, the core MILP formulation and sequence-to-FLOPs conversion routines are implemented in C++, with a lightweight thread pool. The two phase design described in 4.1.1 substantially reduces the search space. We further overlap the solver exceution with data loading and prefetching so that GPU compute stream never stalls waiting for a new packing plan. As a result, SlimPack introduces minimal overhead while delivering optimal MicroPack partitions in real time.

Model	#heads	#groups	hidden dim	FFN dim	#Layers
Llama 7B	32	–	4096	11008	32
Llama 13B	40	–	5120	13824	40
Llama 70B	64	8	8192	28672	80
Llama 150B	96	8	12288	32768	96

Table 3. Model specifications.

6 Experiment

6.1 Experimental Settings

Enviroment. All experiments were performed on a cluster of GPU-equipped nodes, each housing two Intel Xeon Platinum CPUs, 1000 GiB of RAM, and eight NVIDIA Hopper 80GB GPUs, interconnected via NVLink at 400 GB/s per GPU. For inter-node communication, every GPU is also paired with a 400 Gbps NIC. Tensor Parallelism (TP)—always combined with Sequence Parallelism (SP) are confined to individual nodes, whereas Pipeline Parallelism (PP) and Data Parallelism (DP) may span multiple nodes. Context Parallelism (CP) only crosses node boundaries when memory capacity demands it.

Baseline. Megatron-LM [35] serves as our baseline framework and incorporates the same implementation of partial-recompute and activation-offload features as SlimPack. Both the baseline and SlimPack adopt the 1F1B pipeline schedule. We leverage parallelism (TP, DP, PP, CP) according to model scale and training setup, and enable memory optimizations (partial/full recompute, activation offload) as needed. Additionally, when handling variable-length inputs, the baseline employs the Best-Fit Packing strategy [19].

Model and Datasets. We benchmark various LLaMA-style dense models ranging from 7B to 150B; their detailed specs appear in Table 3. GQA [1] is applied to the 70B and 150B models. All models are equipped with a 32,000 sized vocabulary. We evaluate our system on three datasets (*Common Crawl*, *GitHub*, and *Wikipedia*) as introduced in §2.1.1

Workload and Metrics. Three configuration axes are evaluated: context length, GPU count, and global batch size. Context length ranges from 64 K to 256 K tokens, GPU count per run scales from 128 to 256, and global batch size increases from 512 to 2048—reflecting typical large-scale LLM training setups. We use tokens per second per GPU (TPS) as our primary performance metric and report results averaged over 20 measured iterations after a 5-iteration warmup.

6.2 End-to-End Performance

We conduct extensive experiments to evaluate the efficiency of SlimPack across three datasets, four model configurations, and three context lengths (see Table 3). The combined results are plotted in Figure 13. Specifically, we benchmark

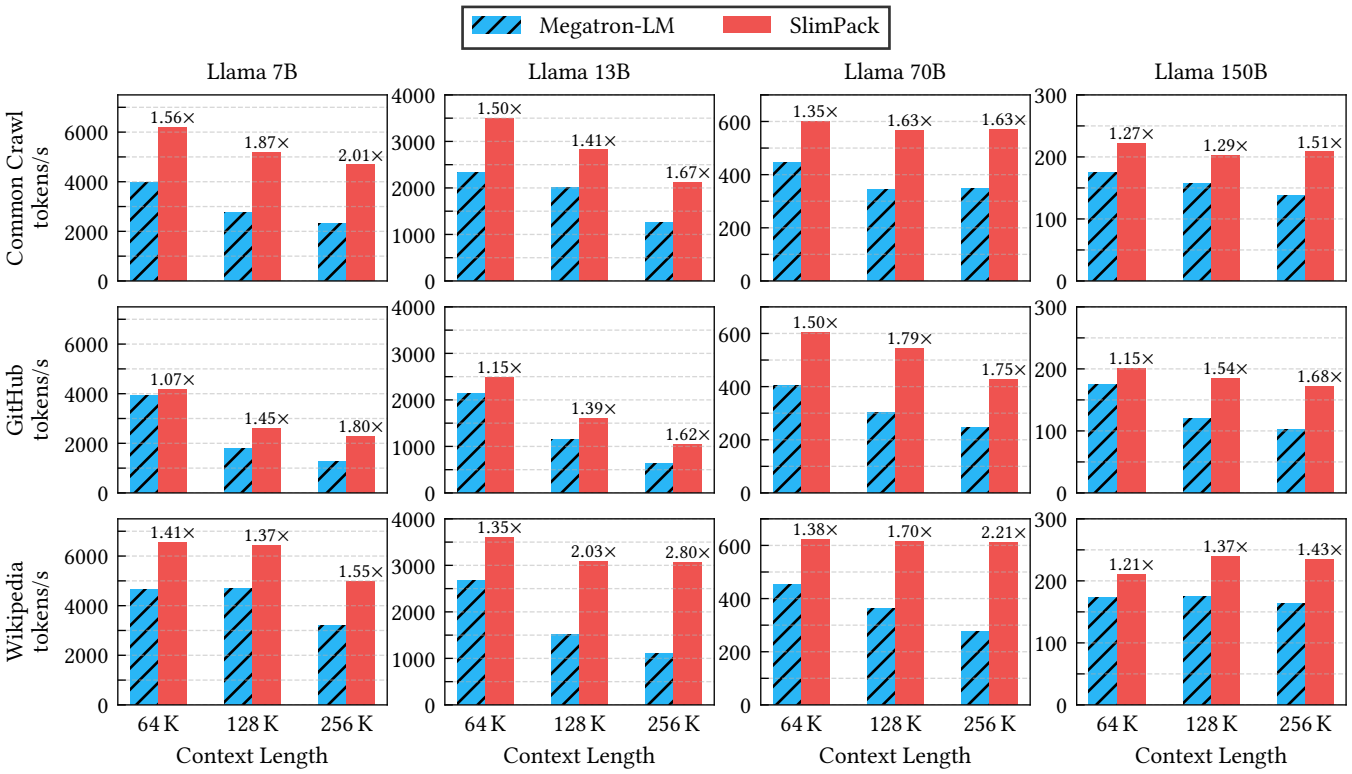

Figure 13. End-to-end performance comparison between Megatron-LM and SlimPack across different models and datasets. Annotations above the bars show the relative speedup of SlimPipe over Megatron-LM.

Llama models at different scales: Llama 7B and Llama 13B are trained on 128 devices, while Llama 70B and Llama 150B are trained on 256 devices.

SlimPack demonstrates significant efficiency improvements across all tested sequence lengths, datasets, and model sizes. Notably, for Llama 13B at a context length of 256K, SlimPack achieves a 2.8× throughput improvement over the baseline. Furthermore, the performance gains of SlimPack scale progressively with longer sequences. For instance, in the Llama 150B case, SlimPack delivers a 1.15× throughput improvement at 64K context length. This advantage increases to 1.54× at 128K and further to 1.68× at 256K, highlighting SlimPack's superior scalability for long-context workloads. Our experiments reveal an important distinction in the scaling behavior of smaller models (Llama 7B and Llama 13B) on the GitHub dataset. While longer context lengths lead to decreased tokens per second per GPU, this stems from fundamental computational properties rather than systemic inefficiencies, such as Model FLOPS Utilization (MFU) dropping:

1. Higher Attention Overhead: Attention layers dominate computation in smaller models , causing quadratic complexity to disproportionately impact total FLOPs at longer sequences. For example, at identical context lengths, attention operations consume 3.0× greater share of total FLOPs in Llama 7B compared to Llama 150B.
2. Dataset Characteristics: The GitHub corpus exhibits longer average sequence lengths than other two datasets. As evidenced by Figure 1, the GitHub dataset exhibits a right-shifted length distribution compared to Common Crawl and Wikipedia benchmarks, indicating systematically longer sequence lengths.

In a nutshell, across all setups, SlimPack delivers throughput gains to up to 2.8× over the baseline.

6.3 Workload Balance Studies

Difference in Datasets. The three violin plots highlight how SlimPack's asymmetric slice-level packing adapts to different data distributions. On the GitHub dataset, which displays a relatively moderate spread of sequence lengths, baseline's stragegy already shows some clustering but still suffers from occasional outliers; SlimPack compresses this spread into a tight band, virtually eliminating the ourlier and balanced the overall workload. For the long-tailed CommonCrawl data, the benefits are even more pronounced: sample-level packing produces a heavy tail of slow microbatches, whereas slice-level packing confines nearly all latencies within a narrow range, eradicating bottlenecks. Finally, on the Wikipedia

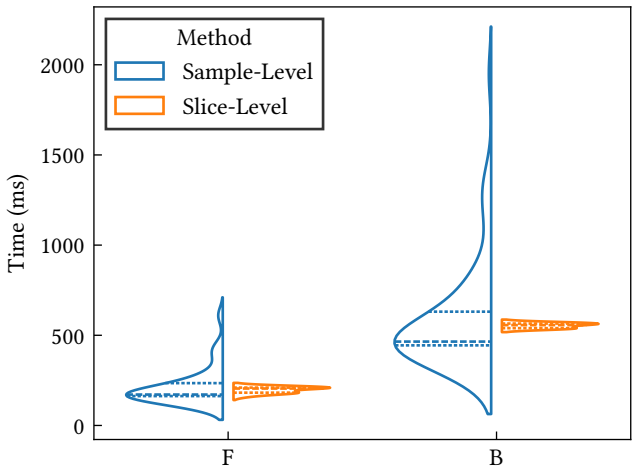

Figure 14. Violin plot comparing computation times for forward (F) and backward (B) passes between a sample-level packing method [19] and our proposed slice-level packing strategy. Execution times are measured on a specific device at the same iteration of two separate runs on the GitHub dataset. Each density function is normalized so the violins have equal width, with inner lines indicating quartiles.

distribution, where short and long sequences create two distinct performance clusters under sample-level packing, our system merges them nearly into a single, uniform mode, demonstrating its ability to smooth out both mild and extreme length variability.

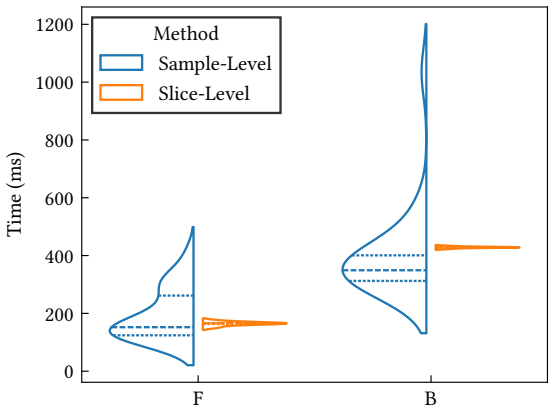

Figure 15. Comparison of forward (F) and backward (B) pass computation times between sample-level and slice-level packing on the Common Crawl dataset. The presentation style and annotations are the same as in Figure 14.

Difference in MicroPack vs MicroBatch. We evaluated the effectiveness of our proposed slice-level packing method against our sample-level packing baseline, by looking at forward and backward pass time distribution of individual

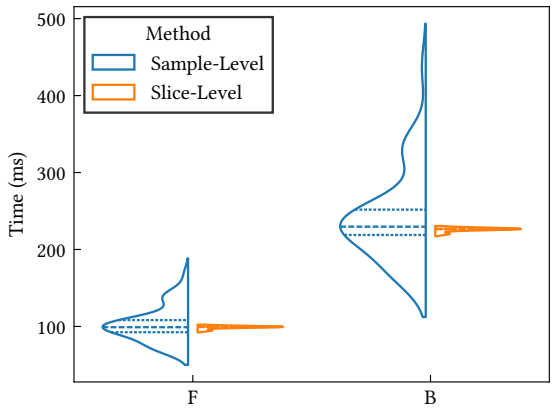

Figure 16. Comparison of forward (F) and backward (B) pass computation times between sample-level and slice-level packing on the Wikipedia dataset. The presentation style and annotations are the same as in Figure 14.

microbatches or micropacks. Figure 14 presents this comparison on the GitHub dataset. The results indicate a notable advantage for slice-level packing, which exhibits a more concentrated distribution, as shown by the narrower violin plot. This implys a consistent latency across micropacks, which helps reducing PP imbalance bubbles.

We extended our analysis to Common Crawl and Wikipedia datasets, with results detailed in Figure 15 and Figure 16. Both results reaffirm the trends observed on the GitHub dataset. Across these diverse datasets, slice-level packing consistently outperforms sample-level packing, showing reduced variance in execution times. The backward pass timings are also highly concatenated, thanks to the asymmetric partitioning scheme.

Collectively, the data presented suggests that slice-level packing offers considerable improvements in terms of workload balance over traditional sample-level packing. This improved efficiency can translate to smaller pipeline imbalance bubbles, particularly for datasets with long-tailed distribution.

6.4 Simulator Accuracy

To evaluate the precision of our memory model, we conducted an experiment using the Llama 13B model. The experiment were run on the *GitHub* dataset with 16 NVIDIA Hopper 80GB GPUs. `torch.cuda.max_memory_allocated` is used to measure peak GPU memory usage at the end of each iteration.

As illustrated in Figure 17, our predictions align well with the actual peak GPU memory consumption. The Mean Absolute Percentage Error (MAPE) between the predicted and measured values is merely 1.6%. The predicted values are consistently lower than the measured peak memory. This underestimation is likely introduced by factors not explicitly

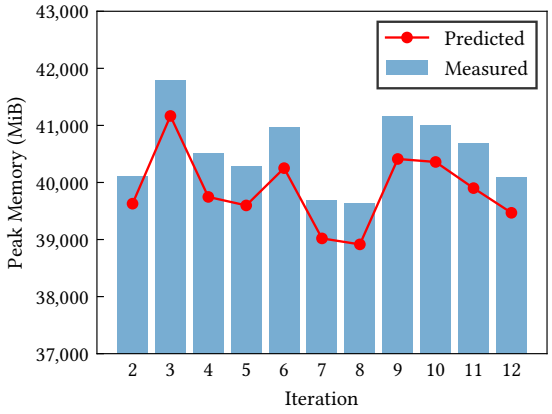

Figure 17. Comparison of measured (bars) and predicted (markers) peak GPU memory consumption in MiB from iterations 2 to 12.

captured in the model (e.g., communication buffers). The systematic under-prediction does not adversely affect our packing resolution because the memory model is primarily used as a conservative filter to discard packing strategies that would lead to out-of-memory (OOM) errors.

7 Conclusion

In this paper, we observe critical inefficiencies in variable-length LLM training, namely the cascading imbalance in hybrid parallel systems and workload skew from asymmetric forward-backward costs, which conventional packing strategies fail to resolve. Therefore, we introduce **SlimPack**, a optimized framework that reimagines data packing by decomposing samples into fine-grained slices. Its core innovation, Asymmetric Partitioning, creates balanced scheduling units (MicroPacks) uniquely tailored for the different demands of the forward and backward passes. Guided by a two-phase solver and a high-fidelity simulator, SlimPack holistically resolves imbalances across all parallel dimensions with minimal communication overhead. Extensive experiments show SlimPack achieves up to a 2.8× training throughput improvement over strong baselines under 256K context length, presenting a robust and scalable solution that breaks the conventional trade-off between workload balance and resource efficiency for long-context model training.

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *The 2023 Conference on Empirical Methods in Natural Language Processing*. https://openreview.net/forum?id=hmOwOZWzYE

[2] Yushi Bai, Xin Lv, Jiajie Zhang, Yuze He, Ji Qi, Lei Hou, Jie Tang, Yuxiao Dong, and Juanzi Li. 2024. LongAlign: A Recipe for Long Context Alignment of Large Language Models. arXiv:2401.18058 [cs.CL] https://arxiv.org/abs/2401.18058

[3] William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. 2023. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431* (2023).

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[5] CodeParrot. [n. d.]. github-code. https://huggingface.co/datasets/codeparrot/github-code

[6] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 16344–16359. https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf

[7] Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[8] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[9] Wikimedia Foundation. [n. d.]. *Wikimedia Downloads*. https://dumps.wikimedia.org

[10] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. 2025. ByteScale: Efficient Scaling of LLM Training with a 2048K Context Length on More Than 12,000 GPUs. arXiv:2502.21231 [cs.DC] https://arxiv.org/abs/2502.21231

[11] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. 2024. Enabling Parallelism Hot Switching for Efficient Training of Large Language Models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 178–194. doi:10.1145/3694715.3695969

[12] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377* (2018).

[13] Mincong Huang, Chao Wang, Chi Ma, Yineng Zhang, Peng Zhang, and Lei Yu. 2024. Re-evaluating the Memory-balanced Pipeline Parallelism: BPipe. *arXiv preprint arXiv:2401.02088* (2024).

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[15] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. arXiv:2309.14509 [cs.LG] https://arxiv.org/abs/2309.14509

[16] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 497–511. https://proceedings.mlsys.org/paper_files/paper/2020/file/0b816ae8f06f8dd3543dc3d9ef196cab-Paper.pdf

[17] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. 2023. DynaPipe: Optimizing Multi-task Training through Dynamic Pipelines. doi:10.48550/arXiv.2311.10418 arXiv:2311.10418 [cs.DC]

[18] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing Activation Recomputation in Large Transformer Models. arXiv:2205.05198 [cs.LG] https://arxiv.org/abs/2205.05198

[19] Mario Michael Krell, Matej Kosec, Sergio P. Perez, and Andrew Fitzgibbon. 2022. Efficient Sequence Packing without Cross-contamination: Accelerating Large Language Models without Impacting Performance. arXiv:2107.02027 [cs.CL] https://arxiv.org/abs/2107.02027

[20] Shigang Li and Torsten Hoefler. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 27, 14 pages. doi:10.1145/3458817.3476145

[21] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. 2023. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*. 766–775.

[22] Zhouyang Li, Yuliang Liu, Wei Zhang, Tailing Yuan, Bin Chen, Chengru Song, and Di Zhang. 2025. SlimPipe: Memory-Thrifty and Efficient Pipeline Parallelism for Long-Context LLM Training. arXiv:2504.14519 [cs.LG] https://arxiv.org/abs/2504.14519

[23] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*. PMLR, 6543–6552.

[24] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2024. RingAttention with Blockwise Transformers for Near-Infinite Context. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=WsRHpHH4s0

[25] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).

[26] Weijian Liu, Mingzhen Li, Guangming Tan, and Weile Jia. 2025. Mario: Near Zero-cost Activation Checkpointing in Pipeline Parallelism. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, USA) *(PPoPP '25)*. Association for Computing Machinery, New York, NY, USA, 197–211. doi:10.1145/3710848.3710878

[27] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[28] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[29] OpenAI, Josh Achiam, Steven Adler, et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[30] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. 2023. The RefinedWeb dataset for Falcon LLM: outperforming curated corpora with web data, and web data only. *arXiv preprint arXiv:2306.01116* (2023). arXiv:2306.01116 https://arxiv.org/abs/2306.01116

[31] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems* 5 (2023), 606–624.

[32] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. arXiv:1910.02054 [cs.LG] https://arxiv.org/abs/1910.02054

[33] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 1–14.

[34] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[35] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL] https://arxiv.org/abs/1909.08053

[36] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. 2024. AdaPipe: Optimizing Pipeline Parallelism with Adaptive Recomputation and Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 86–100. doi:10.1145/3620666.3651359

[37] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, et al. 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL] https://arxiv.org/abs/2312.11805

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*. 5998–6008. https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[39] Shuhe Wang, Guoyin Wang, Yizhong Wang, Jiwei Li, Eduard Hovy, and Chen Guo. 2024. Packing Analysis: Packing Is More Appropriate for Large Models or Datasets in Supervised Fine-tuning. arXiv:2410.08081 [cs.LG] https://arxiv.org/abs/2410.08081

[40] Yujie Wang, Shiju Wang, Shenhan Zhu, Fangcheng Fu, Xinyi Liu, Xuefeng Xiao, Huixia Li, Jiashi Li, Faming Wu, and Bin Cui. 2025. FlexSP: Accelerating Large Language Model Training via Flexible Sequence Parallelism. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 421–436. doi:10.1145/3676641.3715998

[41] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, Yuchen Hao, and Yufei Ding. 2025. WLB-LLM: Workload-Balanced 4D Parallelism for Large Language Model Training. arXiv:2503.17924 [cs.DC] https://arxiv.org/abs/2503.17924

[42] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng