SEESAW: HIGH-THROUGHPUT LLM INFERENCE VIA MODEL RE-SHARDING

Qidong Su [1 2 3] **Wei hao** [3 4] **Xin Li** [3] **Muralidhar Andoorveedu** [3] **Chenhao Jiang** [1 2] **Zhanda Zhu** [1 2 3]
Kevin Song [1 2] **Christina Giannoula** [1 2 3] **Gennady Pekhimenko** [1 2 3]

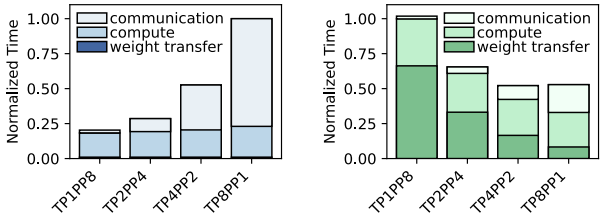
ABSTRACT

To improve the efficiency of distributed large language model (LLM) inference, various parallelization strategies, such as tensor and pipeline parallelism, have been proposed. However, the distinct computational characteristics inherent in the two stages of LLM inference—prefilling and decoding—render a single static parallelization strategy insufficient for the effective optimization of both stages. To address this challenge, in this work we present *Seesaw*, an LLM inference engine optimized for throughput-oriented tasks. The key idea behind Seesaw is *dynamic model re-sharding*, a technique that facilitates the dynamic reconfiguration of parallelization strategies across stages, thereby maximizing throughput at both phases. To mitigate re-sharding overhead and optimize computational efficiency, we employ *tiered KV cache buffering* and *transition-minimizing scheduling*. These approaches work synergistically to reduce the overhead caused by frequent stage transitions while ensuring maximum batching efficiency. Our evaluation demonstrates that Seesaw achieves a throughput increase of up to 1.78× (1.36× on average) compared to vLLM, the most widely used state-of-the-art LLM inference engine, particularly in scenarios with low-bandwidth interconnections.

1 INTRODUCTION

Large language models (LLMs), such as the LLaMA (Touvron et al., 2023a) and GPT (Achiam et al., 2023) families, have demonstrated exceptional performance across a wide range of tasks. Beyond their prevalent use in interactive applications like chatbots (OpenAI, 2024), LLMs are also gaining high interest in throughput-oriented offline inference workloads such as information extraction (Narayan et al., 2022), database querying (Liu et al., 2024c), and knowledge graph processing (Edge et al., 2024). Unlike interactive applications where low latency is crucial, these offline inference tasks *prioritize high throughput over response time*. These offline inference workloads are widely adopted in industry (Kamsetty et al., 2023; Yu et al., 2024; Dell Technologies, 2024; Chan et al.), leading MLPerf to develop benchmarks specifically for them (MLCommons, 2024). In this work, we focus on improving inference efficiency for offline, throughput-oriented LLM inference workloads.

As LLMs often exceed the memory capacity of individual GPUs, parallelization is essential for their deployment (Ben-Nun & Hoefler, 2019; Shoeybi et al., 2019). Several parallelization strategies, including *tensor* parallelism (Shoeybi et al., 2019) and *pipeline* parallelism (Narayanan et al., 2019; Huang et al., 2019), have been proposed, each presenting distinct trade-offs in memory efficiency, inter-device communication, and computational efficiency. Tensor parallelism distributes model weights across devices but suffers from high communication costs due to frequent all-reduce operations at each layer (Pope et al., 2023; Chang et al., 2024). The communication cost becomes particularly severe in systems connected via PCIe (Dell Technologies, 2023) or with partial high-speed connections (NVIDIA Corporation, 2020). In contrast, pipeline parallelism partitions the model into sequential stages, reducing inter-device communication by passing only activations between them. However, to enable pipelining, each data batch needs to be divided



(a) Prefill (b) Decode

Figure 1. Breakdown of execution time for the prefill and decode stages for LLaMA2-13B inference on 8×L4 GPUs connected with PCIe. Communication overhead increases with the degree of tensor parallelism. (The global batch size is 16. Pipeline parallelism further divides the data into micro-batches of size 16/PP to fully utilize pipelining).

[1] University of Toronto [2]Vector Institute [3]CentML Inc [4]Stanford University. Correspondence to: Qidong Su <qdsu@cs.toronto.edu>.

Proceedings of the 8th MLSys Conference, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

into micro-batches, leading to extra execution overheads, since every micro-batch repeatedly loads weights into the compute units (see Section 3.1 for details).

While numerous studies have proposed methods to optimize parallelization strategies for LLMs (Miao et al., 2023; Kwon et al., 2023; Li et al., 2023; Pope et al., 2023; Zhu et al., 2025), prior works typically rely on a single, static configuration throughout the entire generation process. However, our findings indicate that this one-size-fits-all approach is often inefficient for *throughput-oriented* LLM inference because it fails to leverage the distinct patterns between the two stages in LLM generation: the *prefill* stage, where the input sequence is processed at once to produce the initial token, and the *decode* stage, where subsequent tokens are generated sequentially based on prior tokens. These two stages exhibit fundamentally different computational characteristics (Yuan et al., 2024). During the prefill stage, multiple tokens from the input prompt are processed simultaneously, making computation and communication the dominant contributors to runtime. In contrast, the decode stage processes one token at a time for each sequence, increasing the percentage of time spent on weight transfer. This difference indicates that the optimal parallelization strategy for each stage may also vary.

To illustrate the performance limitations of applying a uniform parallelization strategy for both prefill and decode, we measure the execution time of each stage under various combinations of tensor and pipeline parallelism, as shown in Figure 1. In the prefill stage, as the degree of tensor parallelism increases, the communication overhead increases significantly due to additional GPUs participating in all-reduce operations. As a result, tensor parallelism performs significantly worse than pipeline parallelism. In contrast, during the decode stage, pipeline parallelism is slower than tensor parallelism, largely due to increased weight transferring overhead caused by micro-batching required for pipelining (see Section 3.1 for more details). Therefore, we need stage-specific parallelization strategies to provide better LLM inference throughput.

An existing approach is disaggregated prefill-decode (Zhong et al., 2024; Qin et al., 2024), which assigns prefill and decode computation to different GPU instances. The prefill instances and decode instances form a two-stage pipeline to serve inference requests. Therefore, the overall throughput of disaggregated prefill-decode is constrained by the slower of the two stages, and balancing throughput between these two stages is essential. The key drawback of disaggregated prefill-decode is that it can cause large amounts of pipeline bubbles under resource-constrained environments. For example, when deploying a 70B model on 8×40GB GPUs, even the most balanced configuration results in a 6× difference in throughput between the prefill and decode stages. In this setup, the decode stage operates at one-sixth the throughput of the prefill stage, resulting in a significant bottleneck at the decode stage that slows down the entire system (see Section 3.2 for details).

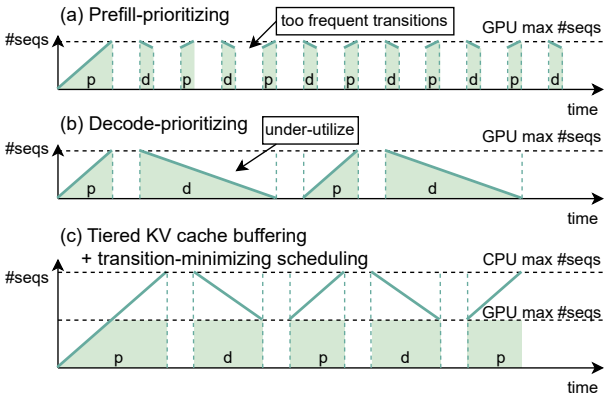


Figure 2. Different scheduling policies considering transition overhead. Decoding throughput is positively correlated with the number of sequences in GPU memory (the maximal batch size), which is highlighted as light green area.

To address these challenges, we present *Seesaw*, a high-throughput LLM inference engine that dynamically reconfigures parallelization strategies between the prefill and decode stages. The key idea behind Seesaw is **model re-sharding**, a novel technique that dynamically re-partitions model weights and KV cache between prefill and decode stages. By tailoring parallelization strategies to the distinct computational demands of each stage, Seesaw reduces communication overhead during the prefill stage, while enhancing memory efficiency in the decode stage, resulting in a substantial increase in overall throughput.

However, the overhead associated with model re-sharding can be high due to frequent transitions between prefill and decode. To maximize throughput, existing systems typically adopt prefill-prioritized scheduling (Yu et al., 2022; Kwon et al., 2023), which interleaves prefill and decode stages across batches to achieve continuous batching. Yet, as illustrated in Figure 2(a), integrating this approach with model re-sharding can result in significant overhead due to frequent transitions between prefill and decode. On the other hand, decode-prioritized scheduling (NVIDIA, 2024a) completes all decode steps for a batch before proceeding to the next, resulting in lower re-sharding overhead. However, as depicted in Figure 2(b), this method suffers from low resource utilization due to smaller batch sizes.

To overcome this constraint and achieve both minimal re-sharding overhead and large batch size, we propose two synergetic techniques: **tiered KV cache buffering** and **transition-minimizing scheduling**. Tiered KV cache buffering leverages CPU memory as auxiliary storage for

the KV cache, enabling Seesaw to process a larger number of prefill requests. Transition-minimizing scheduling reduces re-sharding overhead by minimizing the number of transitions to the decode stage. Seesaw transitions from prefill to decode only after the CPU KV cache is full. As depicted in Figure 2(c), this approach maintains the maximal batch size during the decode stage, while significantly reducing the frequency of stage transitions, thereby minimizing re-sharding overhead. Additionally, to mitigate the overhead of KV cache transfers between CPU and GPU, Seesaw employs asynchronous pipelining to overlap data transfers with computation.

In summary, we make the following contributions.

- We identify and quantitatively analyze the different preferences for parallelisms in the prefill and decode stages of *throughput-oriented* LLM inference tasks. Our analysis comprehensively accounts for data movement, computation, and communication costs.
- We propose *dynamic model re-sharding*, a novel technique that dynamically reconfigures the parallelization strategies for prefill and decode stages. We address the challenge of transition overhead in model re-sharding with continuous batching by introducing *tiered KV cache buffering* and *transition-minimizing scheduling*. Based on these techniques, we implement Seesaw, a high-throughput offline inference system that optimizes parallelization strategies for each LLM inference stage.
- We conduct a comprehensive evaluation of Seesaw across a variety of workloads and hardware configurations. Our results show Seesaw achieves an average speedup of 1.36× and a throughput improvement of up to 1.78× compared to the state-of-the-art LLM inference engines.

2 BACKGROUND

2.1 LLM Inference

Transformer Architecture. Modern large language models are based on the transformer architecture (Vaswani et al., 2017), which typically consists of multiple identical decoder layers (OpenAI, 2024). Each layer includes several linear layers and an attention layer. The weights of the linear layers account for the majority of the model's parameters.

Auto-regressive Generation. LLM inference follows an auto-regressive paradigm (Bengio et al., 2000), which takes an input prompt and generates a sequence of output tokens. This process is divided into two stages: prefilling, which processes the input tokens, and decoding, which generates a token per step. These stages exhibit distinct computational properties (Zhong et al., 2024; Yuan et al., 2024). Prefilling processes the prompts that are typically hundreds to thousands of tokens long. The computation and communication costs, both of which scale with the number of tokens, dominate the runtime during this stage. Since the cost of loading weights is amortized over a larger set of tokens, the overall performance is primarily bound by compute and/or communication. In contrast, Decoding processes only the newly generated tokens in each auto-regressive step and has comparatively smaller computation in each step. Therefore the cost for loading the weight data from off-chip memory to computation units has a relatively higher percentage.

Moreover, in each generation step, the intermediate tensors K and V in each attention operator can be cached for future reuse, which is called Key-value cache (*KV cache*) (Pope et al., 2023). While being able to accelerate computation, it occupies a substantial amount of GPU memory proportional to the total number of tokens.

2.2 LLM Inference Optimization

Parallelism. As the size of LLMs grows, the memory capacity on a single GPU becomes insufficient. Consequently, various techniques are developed to partition models onto multiple GPUs (Zheng et al., 2022). These parallelization strategies can be classified as (1) inter-operator, which places different operators or layers across multiple GPUs, overlapping them with pipelining (known as *Pipeline parallelism*, PP) (Huang et al., 2019; Narayanan et al., 2019; Li et al., 2023), and (2) intra-operator, which partitions different dimensions of tensors involved in computation, including data parallelism (Srivatsa et al., 2024), tensor parallelism (Shoeybi et al., 2019), etc. *Data parallelism* duplicates models on different devices and dispatches requests among them. *Tensor parallelism* shards model weights and each device performs a portion of the computation, then aggregates these partial results to produce the final output.

Batching. Batching more tokens in a single forward pass increases inference efficiency by, for example, amortizing the time required to load model weights (Sheng et al., 2023; Fang et al., 2021). However, its effectiveness differs between the prefilling and decoding stages (Yuan et al., 2024; He & Zhai, 2024; Agrawal et al., 2023). In decoding, where weight-loading overhead occupies a larger portion of the runtime, batching significantly boosts throughput by effectively amortizing this overhead. Conversely, in the prefilling stage, batching has a less pronounced impact since the token count in input prompts is generally sufficient to keep the process compute-bound. Furthermore, optimal batching strategies need to take in consideration other factors such as speculative decoding (Su et al., 2023; Liu et al., 2024d) and prefix sharing (Pan et al., 2024a; Liu et al., 2024b). Overall, larger batch sizes yield higher throughput, though the maximum batch size is limited by available GPU memory, as it requires additional space for activations and the KV cache.

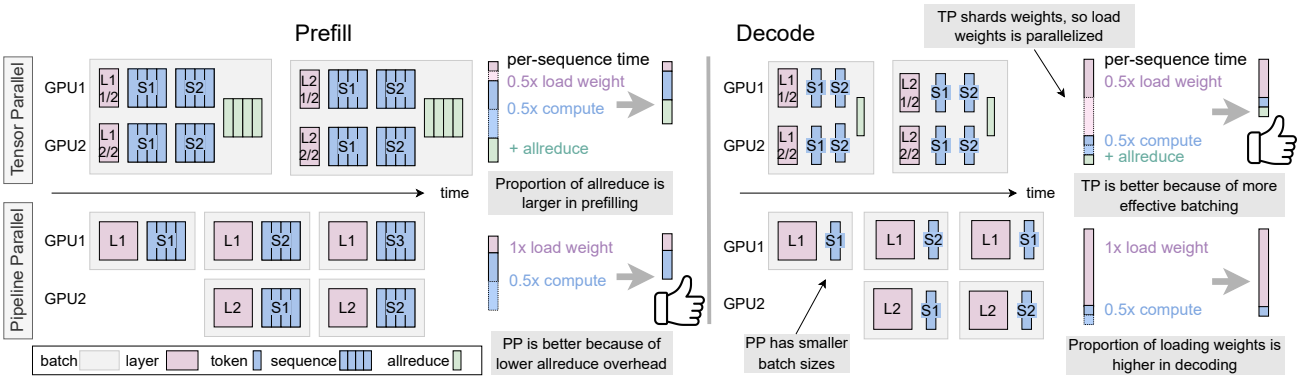


Figure 3. Different effects of tensor and pipeline parallelisms on prefilling and decoding. Tensor parallelism incurs all-reduce overhead, which has a higher percentage in prefilling, therefore pipeline parallelism is better for prefilling. Conversely, pipeline parallelism splits batches into smaller micro-batches, which leads to more forward passes and repetitive loading weights, which is insufficient in decoding.

Continuous Batching and Scheduling. Continuous batching is an essential optimization for throughput-oriented LLM inference (Yu et al., 2022; Kwon et al., 2023). By batching multiple sequences at the token level, it allows the system to onboard new sequences and clear the KV cache of completed sequences at any generation step. This approach enables prefill-prioritizing scheduling, which removes sequences as they finish, frees up their KV cache, and eagerly schedules the prefilling of new sequences whenever GPU memory becomes available. This strategy maximizes the number of concurrent sequences being processed, resulting in higher throughput. Another alternative is to use *decode-prioritizing* scheduling, which minimizes the frequency of transitions. Instead of scheduling to prefilling eagerly, this approach waits until all sequences in a batch have finished decoding before initiating the next round of prefilling. However, this scheduling policy results in suboptimal decoding throughput (Agrawal et al., 2024).

3 Motivation and Analysis

In this section, we provide an in-depth analysis of two key observations we identify from Figure 1 in Section 1: (1) Tensor parallelism often exhibits significantly worse performance than pipeline parallelism during the prefill stage due to its substantial communication overhead; (2) Pipeline parallelism tends to fall short in the decode stage owing to the considerable weight loading overhead it incurs. We then argue that a dynamic parallelization strategy is essential to attain optimal performance across both stages.

Given the importance of batching in throughput-oriented tasks, it can be useful to consider how different parallelization strategies impact the *maximum batch size*, rather than assuming batch size as a tunable parameter, as is often done in online-serving contexts such as DistServe (Zhong et al., 2024) and Sarathi-serve (Agrawal et al., 2024).

3.1 Parallelism Analysis

Observation 1: Tensor parallelism incurs substantial communication overhead during the prefill stage, especially in environments with low-bandwidth interconnections. In Tensor parallelism, each device performs a part of computation and aggregates the partial result. The activations at each layer are synchronized across all GPUs using all-reduce operations. The overhead associated with this operation can be quantified as:

$$\frac{\#\text{tokens} \times \text{activation size}}{\text{all-reduce bandwidth}},$$

where *all-reduce bandwidth* refers to the rate of data transfer during all-reduce operations, calculated as the size of the tensor being all-reduced divided by the all-reduce runtime.

As the degree of tensor parallelism increases, the proportion of execution time of all-reduce operations grows substantially. This growth is attributed to two main factors. First, while model weights are partitioned, activations in tensor parallelism remain fully replicated across GPUs, leading to a constant activation size regardless of the degree of tensor parallelism. Second, all-reduce bandwidth decreases as the number of GPUs grows, due to more complex communication schemes. Therefore, increasing the degree of tensor parallelism not only fails to reduce the traffic of all-reduce operations but further limits the communication bandwidth, resulting in escalated communication overhead. This issue is particularly pronounced in the prefill stage, where a large number of tokens are processed simultaneously, making communication overhead the primary bottleneck. Thus, tensor parallelism tends to perform worse than pipeline parallelism due to its large communication overhead.

Observation 2: Pipeline parallelism suffers from significant weight transferring overhead in the decode stage. Pipeline parallelism distributes model layers sequentially

across devices, with each device responsible for processing a set of consecutive layers before passing the output to the next device. Due to the auto-regressive nature of LLM inference, a sequence cannot enter the pipeline until its preceding token is generated. As a result, at any given time step, a sequence can appear in only one stage of the pipeline, making the batches processed by each device *mutually exclusive*. However, the total number of sequences that the pipeline can handle at a time, referred to as the *global batch size*, is constrained by the size of KV cache. Given the mutual exclusion of batches at each device, pipeline parallelism can process only approximately $1/\text{PP}$ of the global batch per forward pass. We denote this reduced batch size in pipeline parallelism as the *micro-batch size*.

Dividing batches into micro-batches increases the number of LLM forward passes required to process the same amount of requests. Specifically, a pipeline parallelism degree of PP necessitates PP times more forward passes for a given global batch. This repeated execution degrades inference performance, as model weight matrices must be loaded from global memory repeatedly. This inefficiency is especially significant in the decode stage, where weight-loading overhead accounts for a substantial portion of total execution time. As a result, pipeline parallelism generally underperforms relative to tensor parallelism in the decode stage due to the amplified weight loading overhead.

Discussion on Data Parallelism. Unlike tensor and pipeline parallelism, which distribute the model across devices, data parallelism distributes the data while duplicating the model. While data parallelism has minimal communication overhead, it has two key disadvantages: (1) the volume of weight transferring is higher by the number of duplicates compared to tensor parallelism; and (2) it occupies more GPU memory, reducing the available space for the KV cache and thus limiting the maximum batch size resulting in lower throughput. Data parallelism can be applied orthogonally alongside both tensor and pipeline parallelism. We do not dynamically adjust data parallelism, which will be explained in Section 4.1.

Conclusion: No one-size-fits-all When comparing these three parallelism strategies for high-throughput LLM inference, a key observation is that prefilling and decoding stages benefit from different parallelism approaches. This difference arises from the distinct characteristics of each stage, as illustrated in Figure 3. Tensor parallelism is preferred for decoding due to its ability to efficiently accelerate weight matrix loading. However, it incurs significant communication overhead, as it requires all-reduce operations at each layer. In contrast, pipeline and data parallelism have much lower communication overhead, making them preferable for prefilling. However, their decoding throughput is limited by inefficient batching and additional weight-loading overhead.

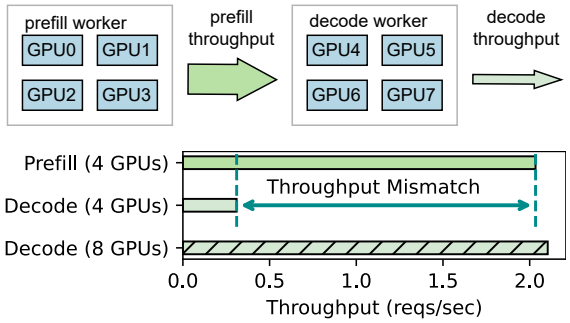


Figure 4. An example of spatially disaggregating prefilling and decoding has a restricted search space. Deploying a 70B model on eight 40GiB GPUs allows only one disaggregation strategy: four GPUs for prefilling and four for decoding. However, this causes severe throughput mismatch between the two stages.

To quantitatively analyze the trade-offs across different parallelisms, we model the average runtime per sequence (the inverse of throughput) as follows. Derivations and further details are provided in the Appendix.

$$T \propto \frac{T_{dm}^{\text{linear}}}{\text{TP}} + \frac{T_{dm}^{\text{attn}} + T_{comp}}{\text{DP} \cdot \text{TP} \cdot \text{PP}} + \frac{T_{comm}(\text{TP})}{\text{PP} \cdot \text{DP}}$$

Here T_{dm}^{linear} represents data movement for linear layers (primarily model weights), T_{dm}^{attn} represents data movement for attention layers (primarily KV cache) , T_{comp} represents computation time, T_{comm} represents communication time. Note that T_{comm} is a monotonically increasing function with respect to TP, as all-reduce operations require more time as TP increases.

Tensor parallelism can effectively accelerate loading model weights, which is T_{dm}^{linear}, while pipeline and data parallelism cannot. On the other hand, pipeline and data parallelism effectively reduce the overhead of communication, while tensor parallelism contrarily increases the communication overhead. In prefilling, T_{dm}^{linear} is negligible, and T_{comm} becomes larger, so pipeline and data parallelisms are more preferred, while in decoding, T_{dm}^{linear} occupies a larger proportion so tensor parallelism is more advantageous.

3.2 Why not Disaggregate Prefilling and Decoding?

Spatially disaggregating prefilling and decoding with separate hardware resources, as done in online serving systems such as DistServe (Zhong et al., 2024) and MoonCake (Qin et al., 2024), is one approach to separately select parallelization strategies for prefilling and decoding. Sequences are first processed by the devices dedicated for prefilling before being transferred to decoding devices.

However, there are two obstacles when applying prefill-decode disaggregation to purely throughput-oriented scenarios. First, since the overall throughput is bound by

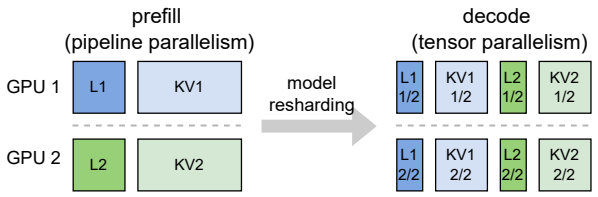


Figure 5. Model weights and KV cache need to be re-sharded when switching between different parallelism.

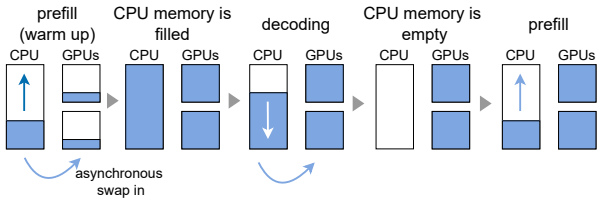


Figure 6. Tiered KV cache buffering and transition-minimizing scheduling, and the change of KV cache occupancy.

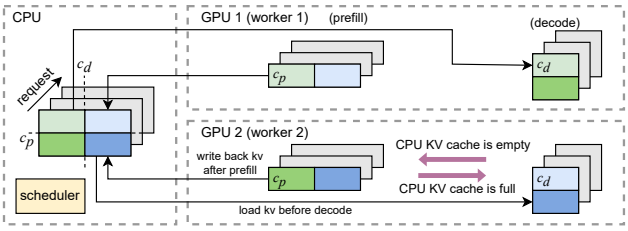


Figure 7. KV cache re-sharding is completed during swapping, leveraging CPU shared memory.

the slower stage, the throughput of prefilling and decoding needs to be matched by adjusting the devices allocated for each stage. However, it can be impractical in resource-constrained scenarios. As shown in Figure 4, to deploy a 70B model (which takes 140GiB memory for model weights) on eight 40GiB GPUs, there is only one disaggregation strategy, that is four GPUs for prefilling and four for decoding (At least four GPUs collectively providing 160 GiB memory are needed to fit the model weights). However, it causes severe throughput mismatch where prefilling has more than 6× higher throughput than decoding. Second, disaggregation duplicates the model weights similarly to data parallelism, bringing similar drawbacks, such as limited KV cache space and increased weight transfer. As a result, decoding throughput with four GPUs is only 15% of that with eight GPUs.

In conclusion, although disaggregation allows selecting different parallelization strategies for each stage, the throughput mismatch (Jin et al., 2024a) between stages and limited resources can lead to suboptimal performance. This calls for a method that offers flexibility in parallelization while maximizing hardware resource utilization.

4 Seesaw: Key Ideas

4.1 Dynamic Model Re-sharding

Observing that prefilling and decoding have distinct preferences for parallelism, we propose a technique called *dynamic model re-sharding*. This technique enables the selection of different parallelism strategies for each stage and automatically transitions between them. This approach expands the configuration space, allowing for separate optimization of the two stages, potentially improving overall throughput compared to using a single configuration. In the following paragraphs, we denote the parallelization strategy used in prefilling as c_p and that in decoding as c_d.

To support transitions between different parallelization configurations, the cluster must rearrange the data stored on each device to align with the new parallelism which involves both model weights and KV cache, as illustrated in Figure 5. In Seesaw, model weights are re-sharded by reloading the required shards from CPU memory, and KV cache re-sharding is performed through CPU shared memory.

The inter-device movement of tensors incurs overhead. To mitigate this re-sharding cost, we design an asynchronous pipeline to overlap data transfer with computation, as detailed in Section 5.2.

Discussion on data parallelism. Unlike switching between tensor and pipeline parallelism, adjusting the degree of data parallelism alters the proportion of GPU memory allocated to model weights versus KV cache. This adjustment increases system complexity or necessitates additional data movement between the CPU and GPU. Therefore, we only dynamically adjust tensor and pipeline parallelism.

4.2 Tiered KV Cache Buffering and Transition-minimizing Scheduling

Challenge: Transition Overhead. In practice, dynamic model resharding encounters an obstacle of transition overhead, which is amplified by the widely-used continuous batching and prefill-prioritizing scheduling. Prefill-prioritizing scheduling eagerly schedules new prefilling tasks, causing frequent transitions between the two stages. As a result, directly applying model re-sharding with this interleaved prefill-decode scheduling policy would introduce significant re-sharding overhead. On the other hand, decode-prioritizing scheduling minimizes the frequency of transitions but results in suboptimal decoding throughput. Other compromise solutions involve setting a threshold-based approach for managing the prefill-decode transition (Cheng et al., 2024). However, they still involve a trade-off between reducing transition overhead and maximizing decod-

ing throughput.

To address this problem, we propose 1) *tiered KV cache buffering*, which leverages CPU memory offloading 2) *transition-minimizing* scheduling policy. These two synergistic techniques prevent frequent stage transitions and maintain a high decoding throughput.

Tiered KV cache buffering uses CPU memory as auxiliary storage for the KV cache, enabling the pre-computation of a large batch of prefilling consecutively. During the prefill stage, the generated KV cache is offloaded to CPU KV cache storage, freeing it from the limitations of GPU memory space. During decoding, continuous batching runs as normal, except that new sequences are on-boarded by swapping in its KV cache from the CPU memory.

Transition-minimizing scheduling controls the transition to only happen when the CPU KV storage is either full or empty. During prefill, once the CPU KV cache storage is fully utilized, re-sharding is triggered, and the cluster transitions to decoding. During decoding, GPUs continue processing requests and loading KV cache from CPU memory, keeping GPU KV cache fully utilized for high decoding throughput. When the entire CPU KV cache has been transferred to GPU memory, the cluster switches back to prefilling. The whole process is illustrated in Figure 6.

KV cache re-sharding occurs throughout this process. As illustrated in Figure 7, in a multi-GPU setup, the CPU KV cache storage is shared among all GPUs. During swap-out, each GPU pushes its shard (based on c_p) of the generated KV cache to the shared CPU storage, where these shards collectively form the complete KV cache. During swap-in, each GPU retrieves its required KV shard (based on c_d) from the shared storage. We implement the shared KV cache using shared memory of the operating system.

5 System Design and Implementation

5.1 Scheduler-worker Architecture

In order to support dynamically switching parallelization configurations for prefilling and decoding, we build Seesaw, a new LLM inference engine designed for high-throughput LLM inference. The overall architecture of Seesaw follows a single-scheduler, multi-worker design. The scheduler manages all generation requests, organizes them into batches, and sends instructions to the workers. To fully utilize pipelining, each decoding step processes $1/\text{PP}$ of the sequences in GPU KV storage. Once a batch is formed, it is sent to workers through shared queues. Each worker is responsible for controlling a single GPU and maintains a task queue to receive and execute instructions sequentially. This architecture facilitates the implementation of asynchronous features, such as pipeline parallelism and the asynchronous pipeline for tiered KV cache buffering.

5.2 Asynchronous Pipeline

While re-sharding and tiered KV cache buffering offer substantial benefits, they also introduce new overhead related to moving model weights and KV cache. The overhead of reloading model weights remains constant relative to batch size, allowing it to be amortized with larger batches. In contrast, swapping the KV cache incurs overhead proportional to batch size, making it harder to amortize. Fortunately, these overheads can be mitigated through computation-communication overlap. We implement an asynchronous pipeline to overlap KV cache transfer with ongoing computation, as illustrated in Figure 8.

Overlap swap-out with computation. The KV cache generated during the prefilling stage is not used until decoding begins, allowing the KV cache swap-out to overlap with other computations during prefilling. Although CPU-GPU data transfer is relatively slow due to PCIe bandwidth limitations, it can still be overlapped with computation, given the high FLOPS involved in prefilling.

In practice, CPU-GPU data transfer can only overlap with computation when using pinned memory, but shared memory cannot be pinned (AlbanD, 2023). To address this, we split the transfer into two stages: GPU to pinned memory (overlapped with computation) and then pinned to shared memory, which is a host-side operation that also runs concurrently with GPU kernels.

Asynchronous swap-in. We implement swap-in using a background thread called the *prefetcher* on each worker, operating in a fully asynchronous paradigm. The prefetcher is controlled directly by the scheduler and runs independently of the main thread, whether the main thread is handling prefilling or decoding. In each iteration, the scheduler creates new prefetching tasks when there are free slots in the GPU KV store. Once the prefetcher completes moving the KV cache for certain sequences, it notifies the scheduler via a shared queue, allowing those sequences to be scheduled for decoding tasks later.

Swap-in can also be well overlapped if the data movement is faster than the speed of the GPU KV cache being consumed. In practice, as long as the output length is not too short, the swap-in can also be well overlapped.

Bandwidth-aware KV cache layout. The data layout of the KV cache significantly impacts the bandwidth efficiency of data movement. There are two common layouts for storing KV cache: *(seq_len, num_heads, head_dim)* (NHD) and *(num_heads, seq_len, head_dim)* (HND). NHD is less optimal for memory access because tensor parallelism shards the KV cache along the H dimension (number of heads), which is the second-to-last dimension, leading to more non-

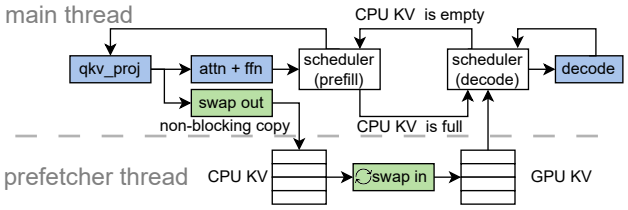


Figure 8. Async pipeline of Seesaw: Swap-in overlaps with prefill computation, while swap-out occurs in a separate asynchronous prefetcher thread.

contiguous memory access. Therefore, we use the HND layout for storing the KV cache in CPU memory.

6 Evaluation

In this section, we evaluate the performance of Seesaw under a variety of hardware configurations and workloads.

6.1 Experiment Settings

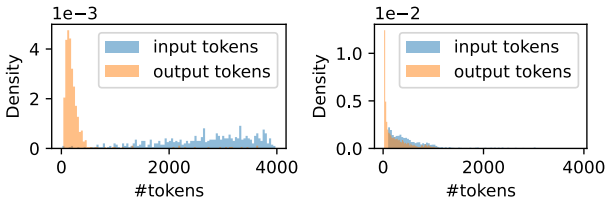
Hardware. We use three types of GPUs: NVIDIA A10, L4, and A100. The A10 and L4 are deployed on AWS EC2 instances (`g5.48xlarge` and `g6.48xlarge` (Amazon Web Services, 2024)), and the A100 is used on GCP (Google Cloud, 2024). GPU specifications are listed in Table 1. The PCIe connection for each GPU is PCIe 4.0 8×, providing 16 GiB/s bandwidth (PCI-SIG, 2017), while NVLink (NVIDIA Corporation, 2024) offers a bandwidth of 600 GiB/s. Additionally, we allocate 80 GiB of CPU memory per GPU.

Model. We use three different LLMs with different sizes: (1) a 15B variety of LLaMA3 (Elinas, 2024); (2) CodeLLaMA-34B (Roziere et al., 2023); (3) LLaMA2-70B (Touvron et al., 2023b). They all use Grouped Query Attention (GQA) (Ainslie et al., 2023). For brevity, we refer to them as 15B, 34B, and 70B, respectively, in the following sections. We use float16 as the data type.

Workload. We use two different datasets in our evaluation, namely `sharegpt` (ShareGPT, 2023) and `arxiv-summarization` (Cohan et al., 2018). They correspond to two different distributions of workload. `sharegpt` is a dataset of chatting history, so its input and output have comparable lengths, while `arxiv-summarization` dataset is a summarization dataset where inputs are much longer than outputs. The characteristics of these two datasets are shown in Figure 9. We sample 2000 requests from the `sharegpt` dataset and 500 requests from `arxiv-summarization`. Additionally, we evaluate constant-length workloads in Section 6.5. Since Seesaw is purely throughput-oriented, we measure the end-to-end throughput as our primary metric.

Table 1. GPU hardware specification

GPU Model	Memory Size	Memory Bandwidth	FLOPS	NVLink
A10	24 GiB	600 GiB/s	125T	✗
L4	24 GiB	300 GiB/s	121T	✗
A100	40 GiB	1,555 GiB/s	312T	✓



(a) arxiv-summarization (b) ShareGPT

Figure 9. Input and output length distributions of the datasets

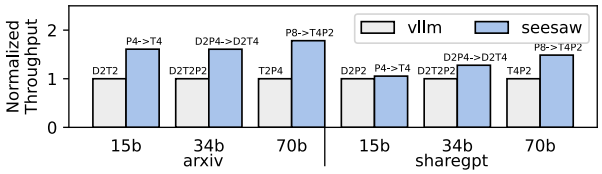
Baselines. We use vLLM 0.5.4 (Kwon et al., 2023) as the baseline. It is the most widely used open-source LLM serving engine with wide support for different parallelisms. We also directly use the vLLM's model implementation for a straightforward comparison. SGLang (Zheng et al., 2023) and DeepSpeed-FastGen (Holmes et al., 2024) do not support pipeline parallelism. TensorRT-LLM (NVIDIA, 2024b) is not included in the comparison because it uses a similar scheduling policy as vLLM, and vLLM demonstrates comparable performance (vLLM Team, 2024) in throughput-oriented tasks. The techniques proposed in Seesaw can also be applied to modifying TensorRT-LLM.

Chunked Prefill. We enable chunked prefill and tune the chunk size for vLLM to get the optimal throughput, following the practice of Sarathi-serve (Agrawal et al., 2024). Otherwise, suboptimal chunk sizes would cause severe throughput degradation.

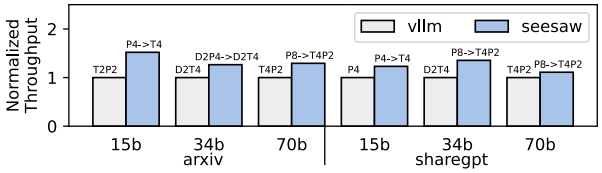
6.2 End-to-end Throughput on PCIe Systems

First, we measure the end-to-end throughput of Seesaw. We sweep over all available single parallelism configurations for vLLM and show the result of the best configuration. We use four GPUs for the 15B model, and eight GPUs for the 34B and 70B models. The result is shown in Figure 10, with the used parallelism labeled above each bar.

On A10, compared with the highest single parallelism baseline, Seesaw achieves a geometrically average speedup of 1.45×, with up to 1.78× speedup. On L4, Seesaw achieves a geometrically average speedup of 1.29×, with up to 1.52× speedup. The overall average speedup is 1.36×. The speedup is more significant on A10, because A10 has better single GPU performance than L4, while they have similar PCIe inter-connection bandwidth, causing a higher percentage of communication overhead.



(a) End-to-end Throughput on A10



(b) End-to-end Throughput on L4

Figure 10. End-to-end throughput comparison on PCIe systems. The used parallelization strategies are labelled above each bar. Labels such as "P4 → D4" represent the parallelization strategies for prefilling and decoding respectively in Seesaw.

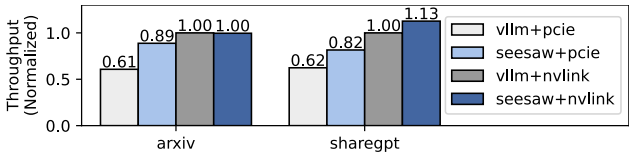


Figure 11. Throughput comparison on A100.

6.3 Speedup Breakdown: An Example

Runtime Breakdown Figure 12 illustrates how Seesaw merges the advantages of different parallelisms. Using CodeLLaMA34B on the `arxiv-summarization` dataset with four A10 GPUs as an example, we measured the runtime of each stage. TP4 is optimal for decoding but significantly slower for prefilling, while PP4 excels at prefilling but is slower during decoding. Seesaw uses a mixed parallelism strategy, applying PP4 for prefilling and TP4 for decoding, achieving performance comparable to the best configuration for each stage.

Compare with Chunked Prefill Compared to the optimal single parallelism configuration (TP2PP2) with chunked prefill, Seesaw achieves higher performance for two reasons: (1) chunked prefill does not piggy-back all decoding steps, leaving some purely decoding steps, and (2) chunked prefill with TP2PP2 is slower than prefilling with PP4. Considering other parallelization strategies with chunked prefill, PP4 is inefficient for purely decoding steps, while TP4 suffers from substantial communication overhead.

6.4 End-to-end Throughput on A100

Speedup on A100 + NVLink The NVLink interconnection across A100 GPUs significantly reduces the all-reduce overhead and further scales tensor parallelism. Tensor parallelism alone typically suffices to achieve optimal performance with up to four GPUs. Nevertheless, there is still a noticeable percentage of all-reduce overhead in prefilling when tensor parallelism scales beyond four GPUs. Seesaw can still provide speedup in this case. As shown in Figure 11, Seesaw still achieves a 13% throughput increase over vLLM for the `sharegpt` dataset on LLaMA3-70B on eight A100s.

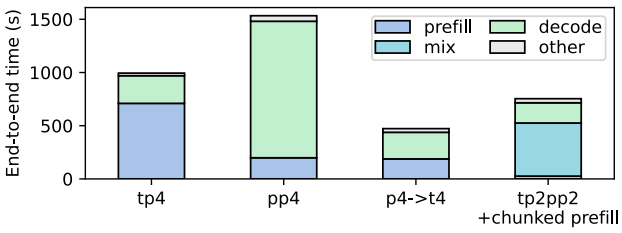


Figure 12. Speedup breakdown. "mix" represents batches containing both prefilling and decoding when chunked prefill is enabled. We disable chunked prefill for TP4 and PP4 in order to show the reference prefilling and decoding time. TP2PP2 with chunked prefill is the optimal parallelism for vLLM.

Speedup on A100 + PCIe Besides A100 SXM with NVLink inter-connection, there is also another version of A100 that is inter-connected with PCIe links, where Seesaw can achieve noticeable speedup. As shown in Figure 11, Seesaw provides 46% speedup on `arxiv-summarization` and 30% speedup on `sharegpt`. Seesaw brings the performance of the A100 PCIe version much closer to the performance level of the NVLink version. vLLM gets roughly 60% throughput on A100 PCIe compared with A100 SXM, while Seesaw boosts it up to 82% – 89%.

6.5 Sensitivity Study

Ratio between Input and Output Length The speedup of Seesaw depends on the ratio between the input and output length, or $P : D$. Model re-sharding has the opportunity to provide speedup when prefilling and decoding have balanced time. To investigate to what extent model re-sharding would be effective, we measure the throughput of various parallelization strategies on synthesized datasets with uniform lengths and different $P : D$ ratios. We fix the input length to 3000 tokens and vary the output length.

As shown in Figure 13, PP8 achieves the highest throughput during prefilling, while TP4PP2 excels in decoding. When the output length equals one (prefilling only), Seesaw and PP8 show similar throughput, and TP4PP2 performs worse due to high communication overhead. As output length increases, the inefficiency of PP in decoding outweighs its advantage in prefilling, causing PP8's throughput to drop rapidly. There is a range where TP2PP4 becomes optimal before decoding dominates the runtime and TP4PP2 takes over as the fastest. Nonetheless, Seesaw consistently outperforms

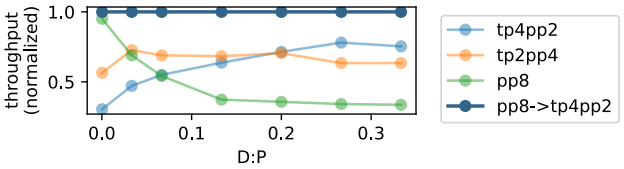


Figure 13. Throughput of various parallelization strategies with different ratios between output and input lengths ($D : P$), measured on 70B model and eight A10 GPUs.

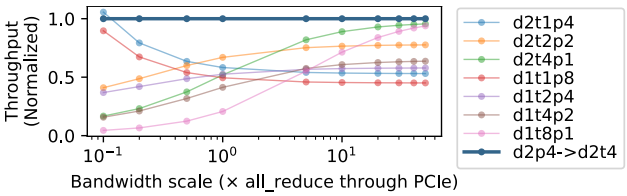


Figure 14. Projected throughput of various parallelization strategies with different inter-connection bandwidth, measured and traced on 34B model and eight A10 GPUs.

other strategies across all input-output length ratios. In real scenarios with variable input and output lengths, Seesaw is even more advantageous due to its adaptive capabilities.

Inter-connection Bandwidth The effectiveness of Seesaw also depends on the inter-connection bandwidth. We investigate this by measuring the runtime and tracing all-reduce operations of running `arxiv-summarization` and 34B model on eight A10s. We then mutate the all-reduce time to project the end-to-end throughput with different inter-connection bandwidths. As shown in Figure 14, when the inter-connection bandwidth is slow (for example, among geographically distributed devices (Borzunov et al., 2022)), pipeline parallelism is optimal; when the bandwidth is very high, tensor parallelism is optimal. The throughput of Seesaw is superior to fixed parallelization strategies on a wide range from $0.1\times$ to $50\times$ of PCIe bandwidth. Our findings indicate that Seesaw primarily is suited for resource-constrained deployments, especially those with relatively slower interconnections, such as clusters consisting of commodity-grade GPUs.

7 Related Work

7.1 Heterogenity between Prefilling and Decoding

Due to the different computational characteristics between prefilling and decoding leading to under-utilization of hardware resources, prior research has investigated two directions to address this problem, namely disaggregating or merging the two stages. Disaggregation places prefilling and decoding onto different devices to avoid their interference while merging processes prefilling and decoding in one batch.

Disaggregate Prefill and Decoding DistServe (Zhong et al., 2024) proposed placing prefilling and decoding on different devices to prevent interference and leverage different characteristics of the two stages. Mooncake (Qin et al., 2024) uses a similar approach via a distributed KV cache pool. P/D-Serve (Jin et al., 2024b) uses the device-to-device network to transfer the KV cache between prefill and decode devices. Splitwise (Patel et al., 2024) proposes using different GPU models for the two stages. TetriInfer (Hu et al., 2024) further disaggregates different downstream tasks to avoid interference. These works are designed for online serving while Seesaw focuses on offline inference. Moreover, they are usually designed for large clusters.

Merge Prefill and Decode (Chunked prefill) Chunked prefill, as proposed by SplitFuse (Holmes et al., 2024), Sarathi (Agrawal et al., 2023), and Sarathi-serve (Agrawal et al., 2024), splits long prompts in the prefilling stage into smaller chunks, combining them with decoding steps to strike a balance between data movement and computation and reduce pipeline bubbles in pipeline parallelism. However, determining the optimal chunk size is challenging. A chunk size that is too large results in excessive decode-only steps, closely resembling traditional prefill-decode scheduling. Conversely, a chunk size that's too small reduces kernel efficiency. Furthermore, chunked prefill cannot always perfectly piggyback decoding with prefilling, leaving separate decode-only steps. For further performance enhancement, model re-sharding can be integrated with chunked prefill to accelerate these decode-only steps. On the other side, the key advantage of chunked prefill is its flexibility in supporting both throughput- and latency-oriented scenarios, whereas Seesaw prioritizes throughput at the cost of increased latency.

7.2 Parallel and Distributed LLM Inference

Aside from tensor parallelism, pipeline parallelism, and data parallelism discussed in Section 2.2, there are also other types of parallelisms, such as sequence parallelism (SP) (Li et al., 2021; Liu et al., 2023; Lin et al., 2024; Brandon et al., 2023; Xue et al., 2024) and fully sharded data parallelism (FSDP) (Zhao et al., 2023; Rajbhandari et al., 2020). Sequence parallelism is especially designed for long sequence lengths, and is orthogonal with our work. FSDP requires frequently transferring weight matrices across GPUs, thus mainly used in training.

HexGen (Jiang et al., 2023), LLM-PQ (Zhao et al., 2024), Helix (Mei et al., 2024) investigate parallelisms in heterogeneous clusters. Intra-device parallelism leverages overlapping functions using different resources within each device, including DeepSeek (Liu et al., 2024a), NanoFlow (Zhu et al., 2024) and Liger (Du et al., 2024). Petals (Borzunov et al., 2022) explores LLM inference in geographically dis-

tributed setups, employing pipeline parallelism to minimize communication costs. SpotServe (Miao et al., 2024) runs LLM inference on preemptible instances. Since most of these works target different scopes than Seesaw, direct comparisons are not applicable.

7.3 Offloading in LLM Inference

Offloading is a widely used technique to run LLM applications in resource-constrained scenarios (Ren et al., 2021). FlexGen (Sheng et al., 2023) swaps tensors across GPU memory, CPU memory, and disks. Fiddler (Kamahori et al., 2024), HeteGen (Xuanlei et al., 2024), PowerInfer (Song et al., 2023) and FastDecoder (He & Zhai, 2024) perform part of computation in CPU, which require CPUs with strong compute capability or external CPU nodes connected with high-bandwidth networking. Instinfer (Pan et al., 2024b) offloads computation to Computational Storage Drives. Unlike these approaches, Seesaw does not assume extremely limited GPU memory that requires frequent CPU-GPU weight matrix transfers. Instead, Seesaw uses CPU memory only as an intermediate stage for model and KV cache resharding.

8 Conclusion

This paper proposes Seesaw, a high-throughput LLM inference engine, to address the inefficiencies of fixed parallelization by selecting different parallelization strategies for the prefilling and decoding stages and switching between them using model re-sharding. It uses tiered KV cache buffering to minimize re-sharding overhead. Our experiments show that Seesaw outperforms widely-used open-source inference engines, with a throughput increase of 1.06-1.78× and an average throughput improvement of 1.36×. These results highlight Seesaw's effectiveness and adaptability.

Acknowledgments

We want to express our sincere gratitude to the anonymous MLSys reviewers for their valuable and constructive feedback and suggestions. The authors with the University of Toronto are supported by the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Agrawal, A., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., and Ramjee, R. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B. S., Tumanov, A., and Ramjee, R. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

AlbanD. Why not multiprocess pin memory in data loader? https://discuss.pytorch.org/t/why-not-multiprocess-pin-memory-in-data-loader/197345/2, 2023.

Amazon Web Services. Amazon EC2 Instance Types, 2024. URL https://aws.amazon.com/ec2/instance-types/.

Ben-Nun, T. and Hoefler, T. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

Bengio, Y., Ducharme, R., and Vincent, P. A neural probabilistic language model. *Advances in neural information processing systems*, 13, 2000.

Borzunov, A., Baranchuk, D., Dettmers, T., Ryabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., and Raffel, C. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022.

Brandon, W., Nrusimha, A., Qian, K., Ankner, Z., Jin, T., Song, Z., and Ragan-Kelley, J. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431*, 2023.

Chan, V., Zhang, H., and Wang, F. Snowflake llm inference: Optimizing gpu capacity for interactive workloads. https://www.snowflake.com/engineering-blog/snowflake-llm-inference-interactive-workloads/.

Chang, L., Bao, W., Hou, Q., Jiang, C., Zheng, N., Zhong, Y., Zhang, X., Song, Z., Jiang, Z., Lin, H., et al. Flux: Fast software-based communication overlap on gpus through kernel fusion. *arXiv preprint arXiv:2406.06858*, 2024.

Cheng, K., Hu, W., Wang, Z., Peng, H., Li, J., and Zhang, S. Slice-level scheduling for high throughput and load balanced llm serving. *arXiv preprint arXiv:2406.13511*, 2024.

Cohan, A., Dernoncourt, F., Kim, D. S., Bui, T., Kim, S., Chang, W., and Goharian, N. A discourse-aware attention

model for abstractive summarization of long documents. *arXiv preprint arXiv:1804.05685*, 2018.

Dell Technologies. Poweredge server gpu matrix, 2023. URL https://www.delltechnologies.com/asset/en-ca/products/servers/briefs-summaries/poweredge-server-gpu-matrix.pdf.

Dell Technologies. Inferencing performance for generative ai in the enterprise with amd accelerators. https://infohub.delltechnologies.com/en-au/l/generative-ai-in-the-enterprise-with-amd-accelerators/inferencing-performance/, 2024.

Du, J., Wei, J., Jiang, J., Cheng, S., Huang, D., Chen, Z., and Lu, Y. Liger: Interleaving intra-and inter-operator parallelism for distributed large model inference. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 42–54, 2024.

Edge, D., Trinh, H., Cheng, N., Bradley, J., Chao, A., Mody, A., Truitt, S., and Larson, J. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

Elinas. Llama-3-15b instruct-zeroed. https://huggingface.co/elinas/Llama-3-15B-Instruct-zeroed, 2024.

Fang, J., Yu, Y., Zhao, C., and Zhou, J. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 389–402, 2021.

Google Cloud. GPU platforms: A100 GPUs, 2024. URL https://cloud.google.com/compute/docs/gpus#a100-gpus.

He, J. and Zhai, J. Fastdecode: High-throughput gpu-efficient llm serving using heterogeneous pipelines. *arXiv preprint arXiv:2403.11421*, 2024.

Holmes, C., Tanaka, M., Wyatt, M., Awan, A. A., Rasley, J., Rajbhandari, S., Aminabadi, R. Y., Qin, H., Bakhtiari, A., Kurilenko, L., et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

Hu, C., Huang, H., Xu, L., Chen, X., Xu, J., Chen, S., Feng, H., Wang, C., Wang, S., Bao, Y., et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

Jiang, Y., Yan, R., Yao, X., Zhou, Y., Chen, B., and Yuan, B. Hexgen: Generative inference of large language model over heterogeneous environment. In *Forty-first International Conference on Machine Learning*, 2023.

Jin, H., Lai, R., Ruan, C. F., Wang, Y., Mowry, T. C., Miao, X., Jia, Z., and Chen, T. A system for microserving of llms. *arXiv preprint arXiv:2412.12488*, 2024a.

Jin, Y., Wang, T., Lin, H., Song, M., Li, P., Ma, Y., Shan, Y., Yuan, Z., Li, C., Sun, Y., et al. P/d-serve: Serving disaggregated large language model at scale. *arXiv preprint arXiv:2408.08147*, 2024b.

Kamahori, K., Gu, Y., Zhu, K., and Kasikci, B. Fiddler: Cpu-gpu orchestration for fast inference of mixture-of-experts models. *arXiv preprint arXiv:2402.07033*, 2024.

Kamsetty, A., Chen, H., and Xie, L. How bytedance scales offline inference with multi-modal llms to 200tb data. https://www.anyscale.com/blog/how-bytedance-scales-offline-inference-with-multi-modal-llms-to-200TB-data, August 2023.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Li, S., Xue, F., Baranwal, C., Li, Y., and You, Y. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120*, 2021.

Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., et al. Alpaserve: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 663–679, 2023.

Lin, B., Peng, T., Zhang, C., Sun, M., Li, L., Zhao, H., Xiao, W., Xu, Q., Qiu, X., Li, S., et al. Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache. *arXiv preprint arXiv:2401.02669*, 2024.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024a.

Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023.

Liu, S., Biswal, A., Cheng, A., Mo, X., Cao, S., Gonzalez, J. E., Stoica, I., and Zaharia, M. Optimizing llm queries in relational workloads. *arXiv preprint arXiv:2403.05821*, 2024b.

Liu, S., Biswal, A., Cheng, A., Mo, X., Cao, S., Gonzalez, J. E., Stoica, I., and Zaharia, M. Optimizing llm queries in relational workloads. *arXiv preprint arXiv:2403.05821*, 2024c.

Liu, X., Daniel, C., Hu, L., Kwon, W., Li, Z., Mo, X., Cheung, A., Deng, Z., Stoica, I., and Zhang, H. Optimizing speculative decoding for serving large language models using goodput. *arXiv preprint arXiv:2406.14066*, 2024d.

Mei, Y., Zhuang, Y., Miao, X., Yang, J., Jia, Z., and Vinayak, R. Helix: Distributed serving of large language models via max-flow on heterogeneous gpus. *arXiv preprint arXiv:2406.01566*, 2024.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Jin, H., Chen, T., and Jia, Z. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023.

Miao, X., Shi, C., Duan, J., Xi, X., Lin, D., Cui, B., and Jia, Z. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 1112–1127, 2024.

MLCommons. Mlperf inference: Datacenter benchmark suite. https://mlcommons.org/benchmarks/inference-datacenter/, 2024.

Narayan, A., Chami, I., Orr, L., Arora, S., and Ré, C. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911*, 2022.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pp. 1–15, 2019.

NVIDIA. Fastertransformer: Transformer related optimization, including bert, gpt. https://github.com/NVIDIA/FasterTransformer, 2024a.

NVIDIA. Tensorrt-llm: Optimized inference for large language models. https://github.com/NVIDIA/TensorRT-LLM, 2024b.

NVIDIA Corporation. Nvidia a100 pcie product brief, 2020. URL https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/A100-PCIE-Prduct-Brief.pdf.

NVIDIA Corporation. NVIDIA NVLink: High-Speed GPU Interconnect, 2024. URL https://www.nvidia.com/en-us/data-center/nvlink/.

OpenAI. Chatgpt (gpt-4), 2024. URL https://www.openai.com/research/gpt-4.

Pan, R., Wang, Z., Jia, Z., Karakus, C., Zancato, L., Dao, T., Wang, Y., and Netravali, R. Marconi: Prefix caching for the era of hybrid llms. *arXiv preprint arXiv:2411.19379*, 2024a.

Pan, X., Li, E., Li, Q., Liang, S., Shan, Y., Zhou, K., Luo, Y., Wang, X., and Zhang, J. Instinfer: In-storage attention offloading for cost-effective long-context llm inference. *arXiv preprint arXiv:2409.04992*, 2024b.

Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., and Bianchini, R. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

PCI-SIG. PCI-SIG Releases PCIe 4.0, Version 1.0, 2017. URL https://pcisig.com/pci-sig-releases-pcie%C2%AE-40-version-10.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

Qin, R., Li, Z., He, W., Zhang, M., Wu, Y., Zheng, W., and Xu, X. Mooncake: Kimi's kvcache-centric architecture for llm serving. *arXiv preprint arXiv:2407.00079*, 2024.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 551–564, 2021.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

ShareGPT. Sharegpt vicuna unfiltered dataset. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered, 2023. Apache 2.0 License.

Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pp. 31094–31116. PMLR, 2023.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Song, Y., Mi, Z., Xie, H., and Chen, H. Powerinfer: Fast large language model serving with a consumer-grade gpu. *arXiv preprint arXiv:2312.12456*, 2023.

Srivatsa, V., He, Z., Abhyankar, R., Li, D., and Zhang, Y. Preble: Efficient distributed prompt scheduling for llm serving. 2024.

Su, Q., Giannoula, C., and Pekhimenko, G. The synergy of speculative decoding and batching in serving large language models. *arXiv preprint arXiv:2310.18813*, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

vLLM Team. Performance update: Bringing vllm to the next level. https://blog.vllm.ai/2024/09/05/perf-update.html, 2024.

Xuanlei, Z., Jia, B., Zhou, H., Liu, Z., Cheng, S., and You, Y. Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices. *Proceedings of Machine Learning and Systems*, 6:162–172, 2024.

Xue, F., Chen, Y., Li, D., Hu, Q., Zhu, L., Li, X., Fang, Y., Tang, H., Yang, S., Liu, Z., et al. Longvila: Scaling long-context visual language models for long videos. *arXiv preprint arXiv:2408.10188*, 2024.

Yu, C., Lee, S., Xu, R., Lin, W., Gorthy, P., and Liaw, R. Batch llm inference on anyscale slashes aws bedrock costs by up to 6x, October 2024. URL https://www.anyscale.com/blog/batch-llm-inference-announcement.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Yuan, Z., Shang, Y., Zhou, Y., Dong, Z., Xue, C., Wu, B., Li, Z., Gu, Q., Lee, Y. J., Yan, Y., et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

Zhao, J., Wan, B., Peng, Y., Lin, H., and Wu, C. Llm-pq: Serving llm on heterogeneous clusters with phase-aware partition and adaptive quantization. *arXiv preprint arXiv:2403.01136*, 2024.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.

Zheng, L., Yin, L., Xie, Z., Huang, J., Sun, C., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023.

Zhong, Y., Liu, S., Chen, J., Hu, J., Zhu, Y., Liu, X., Jin, X., and Zhang, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.

Zhu, K., Zhao, Y., Zhao, L., Zuo, G., Gu, Y., Xie, D., Gao, Y., Xu, Q., Tang, T., Ye, Z., et al. Nanoflow: Towards optimal large language model serving throughput. *arXiv preprint arXiv:2408.12757*, 2024.

Zhu, Z., Giannoula, C., Andoorveedu, M., Su, Q., Mangalam, K., Zheng, B., and Pekhimenko, G. Mist: Efficient distributed training of large language models via memory-parallelism co-optimization. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, 2025.