SiPipe: Bridging the CPU–GPU Utilization Gap for Efficient Pipeline-Parallel LLM Inference

Yongchao He
SciTiX AI

Bohan Zhao
SciTiX AI

Zheng Cao
SciTiX AI

Abstract

As inference workloads for large language models (LLMs) scale to meet growing user demand, pipeline parallelism (PP) has become a widely adopted strategy for multi-GPU deployment, particularly in cross-node setups, to improve key-value (KV) cache capacity and inference throughput. However, PP suffers from inherent inefficiencies caused by three types of execution bubbles—*load-imbalance*, *intra-stage*, and *inter-stage*—which limit pipeline saturation. We present SiPipe, a heterogeneous pipeline design that improves throughput by leveraging underutilized CPU resources to offload auxiliary computation and communication. SiPipe incorporates three key techniques—*CPU sampling*, a *token-safe execution model*, and *structure-aware transmission*—to mitigate pipeline bubbles and improve execution efficiency. Across diverse LLMs, SiPipe achieves up to 2.1× higher throughput, 42.7% lower per-token latency, and up to 23% higher average GPU utilization compared to the state-of-the-art vLLM under the same PP configuration, demonstrating its generality across LLMs and deployment scenarios.

1 Introduction

Large language models (LLMs), e.g., GPT [11, 35] and Llama [2, 14], have achieved remarkable success in inference tasks such as code generation and question answering, enabling widely used applications like ChatGPT [34]. However, this success has come with rapidly increasing model sizes—often reaching hundreds of billions of parameters—which far exceed the memory capacity of a single GPU—and with it, a growing user base that demands ever higher inference throughput [3].

Scaling LLMs across multiple GPUs is essential due to memory limits [8]. For example, while a 72B LLM's weights may fit on two 80GB GPUs, the key-value (KV) [38] caches used to accelerate inference typically require 2–3× more memory, increasing the practical GPU requirement to eight or more [2, 6, 41]. To handle such large-scale deployments, *tensor parallelism (TP)* is commonly employed to distribute computation across GPUs. However, TP's reliance on frequent *all-reduce* [36] communications creates a performance bottleneck beyond 4–8 GPUs, as inter-GPU communication during *forward* passes often dominates computation time [41].

To scale LLM inference, *pipeline parallelism (PP)* complements TP by dividing the model into sequential stages, each mapped to one or more GPUs. Within each stage, TP [1] is used to further parallelize the *forward* of that stage—i.e., a portion of the model—across the assigned GPUs. This hybrid strategy reduces inter-GPU communication and enables efficient scaling across large GPU clusters. As discussed in prior works [9, 16, 41, 48], let p and t denote the degrees of PP and TP, respectively, and let $N = p \times t$ be the total number of GPUs used. The overall inference throughput scales as $T(p,t) \propto \frac{1}{\frac{k_1}{pt}+\frac{k_2 \log t}{p}+b}$, while the per-token latency follows $D(p,t) \propto \frac{k_1}{t} + k_2 \log t + k_3(p-1)$, where k_i and b are constants. A detailed derivation is provided in Appendix A. Therefore, given a user-facing inference service with a latency budget D_e (i.e., a service-level objective, or SLO), the inference engine must select an empirical configuration (p_e, t_e) such that $D(p_e, t_e) \le D_e$, subject to the constraint $p_e \times t_e = N$, while maximizing throughput. This trade-off typically requires a carefully tuned PP degree p_e—large enough to exploit parallelism, yet small enough to satisfy the latency target. Figure 1 validates this conclusion. On a 16-GPU, 2-node deployment, recent LLMs achieve peak throughput with PP degrees $p = 2$ or 4, using state-of-the-art (SOTA) inference engine vLLM [27]. For example, on DeepSeek V3 [31], the PP-based configuration vLLM (P_4^4) delivers up to 3.22× throughput of the pure TP configuration vLLM (P_{16}^1).

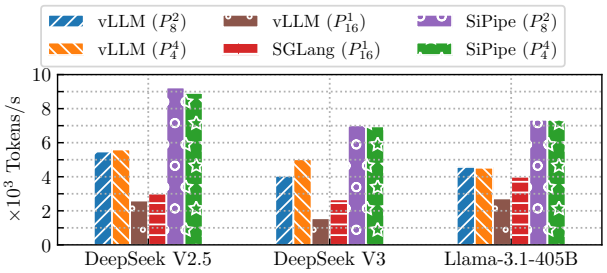


Figure 1. Throughput comparison of different engines under various parallel configurations on 16 H100 GPUs. Each configuration is denoted as P_j^i, where the PP degree $p = i$ and the TP degree $t = j$. See §7.1 for detailed experimental settings.

Despite recent progress, mainstream inference engines [27, 33, 47] offer limited support for PP, missing opportunities to scale LLM inference efficiently across multiple GPUs. Furthermore, their naïve PP designs [27, 33, 41] introduce several fundamental bottlenecks (§3.1):

Stage-wise compute load imbalance. PP introduces a compute imbalance absent in pure TP: although LLM layers are

[1] Data parallelism is treated as TP degree $t = 1$.

evenly divided across stages, the last stage incurs extra compute overhead from the *sampling* task (§2.1), resulting in uneven stage durations and pipeline bubbles.

Stage-wise input preparation overhead. Before each *forward* pass, the CPU prepares input tensors and transfers them to GPU memory, incurring a preparation overhead t_p, while the *forward* itself costs t_f. In PP, this preparation is redundantly repeated at every stage, even though each stage computes only a $1/p$ fraction of the *forward*. As a result, the preparation-to-computation ratio increases from $\frac{t_p}{t_f}$ to $\frac{t_p}{t_f/p}$, leading to greater GPU idle time as the PP degree p increases.

Unpredictable cross-stage communication overhead. In PP, strict data dependencies between adjacent stages—where the size of the transferred data vary with each iteration—require synchronized communication. This communication pattern exhibits unique characteristics of *communication stalls* and *multi-round metadata exchanges*, resulting in overheads that far exceed the cost of the data transfer itself.

Collectively, these factors highlight the growing imperative for pipeline-aware architecture in LLM serving systems. Although recent studies explore diverse pipelined strategies [12, 32, 42, 45], they leave core pipeline bottlenecks unresolved. Likewise, specialized designs for serverless inference [20] or Mixture-of-Expert (MoE) models [13, 49] address narrow scenarios but fall short of providing a general-purpose, inference-oriented pipeline framework.

To tackle these bottlenecks, we identify a key inefficiency in LLM inference: while GPU computation is almost saturated and KV caching with inter-node communication imposes heavy memory and network pressure, CPUs on the same nodes are often underutilized (typically below 10%, see §7.4). SiPipe exploits this *CPU slack* by offloading parts of inter-stage communication and auxiliary tasks to underutilized CPUs, mitigating pipeline stalls and improving throughput. This insight motivates the design of SiPipe, a ***S**aturated **I**nference **Pipe**line* that rebalances computation across CPUs and GPUs for more efficient PP.

SiPipe decouples *sampling* from GPU execution through *asynchronous, column-wise CPU sampling*. SiPipe performs sampling on the CPU to mitigate GPU-side computation imbalance across stages. It adopts a column-wise layout for tensors and performs incremental updates, enabling in-place computation with minimal memory allocation. This design reduces CPU latency and eliminates sampling-induced stalls in PP, thereby improving overall inference efficiency. (§5.1)

SiPipe decouples and overlaps CPU and GPU execution with the *token-safe execution model (TSEM)*. SiPipe introduces the TSEM to overlap CPU-side input preparation with GPU-side *forward*. For a given batch size, TSEM pre-captures two versions of the CUDA graph, which alternately bind to two shared buffers. This design allows the CPU to fill one buffer while the GPU reads the other, enabling seamless parallelism without modifying the execution graph. By decoupling input preparation from *forward*, TSEM eliminates intra-stage GPU stalls and improves overall throughput. (§5.2)

SiPipe introduces *structure-aware transmission (SAT)* for efficient and decoupled stage communication. Based on the insight that *hidden states* (i.e., the output tensors of each stage) exhibit structural stability across inference iterations, SAT infers tensor layouts in advance, enabling early memory allocation and eliminating the need to transmit metadata between stages. This significantly reduces communication rounds and removes the overhead of metadata serialization and deserialization. By allowing receivers to predict tensor layouts ahead of time, SAT eliminates synchronous metadata exchange, naturally enabling asynchronous communication and mitigating *communication stalls*. (§5.3)

Our evaluation demonstrates that under the same PP configurations, SiPipe achieves a throughput improvement of 1.6×-2.1× and 1.4×-1.7× over vLLM [27] on the 8×H100 and 16×H100 testbeds, respectively, along with latency reductions of up to 30.5% and 42.7%. As shown in Figure 1, SiPipe delivers up to 4.5× throughput improvement over the pure TP approach in a cross-node setup with 16 H100 GPUs.

In summary, the contributions of this paper are:

(1) We identify three types of PP bubbles caused by pipeline-agnostic designs in existing LLM inference engines that significantly degrade performance. (§3.1)

(2) We analyze PP characteristics and propose leveraging underutilized CPU resources to assist communication and computation, mitigating these bubbles. (§3.2)

(3) We design and implement SiPipe based on these insights; the code will be open-sourced after publication. (§5)

(4) We conduct extensive experiments on representative LLMs, demonstrating that SiPipe outperforms SOTA systems such as vLLM [27] and SGLang [47]. (§7)

2 Background

2.1 LLM Inference

Figure 2 illustrates the *autoregressive generation* process of LLM inference, where a pretrained LLM (e.g., DeepSeek-R1 [19]) generates **output** tokens conditioned on a **prompt**. A tokenizer [17, 25, 40] *tokenizes* the prompt into discrete **token IDs** using a fixed *vocabulary table*, which maintains the mapping between IDs and tokens. An inference *engine* [27, 47] maintains the generation state for each request as a **sequence**, which consists of the token IDs of both the input prompt and all output token IDs. Generation proceeds in discrete **iterations**, each producing one new token ID per scheduled sequence. To improve hardware efficiency, sequences at the same iteration are grouped into a **batch** for parallel inference on GPUs. Each decoding iteration consists of two parts[2]:

[2]We focus on the *decoding* stage in this paper, where token generation proceeds one step at a time, in contrast to the initial *prefill* [38] stage that encodes the full prompt in a single pass.

Outside the stages, the engine performs two key tasks. First, it selects a batch of active sequences (e.g., seq_0 and seq_1 at iteration 0) and generates a *scheduling output*, which is broadcast to all GPUs. Completed sequences are removed, and new ones are added to maintain batch occupancy and reduce queuing latency. Second, after the batch finishes traversing all stages, the engine appends the generated token IDs to each sequence, checks for completion, and *detokenizes* the outputs into human-readable text.

Inside each stage, three steps are applied in every iteration. ① *Input preparation.* Each GPU constructs a *model input* using the received *scheduling output*. This step encompasses tasks such as model and attention metadata computation, sequence caching and updates, tensor allocation, and CPU–GPU data transfers. The resulting model input is a collection of tensors that are subsequently transferred from CPU to GPU. ② *Forward pass.* In PP, each stage processes model inputs and returns an *activation* vector for each sequence. Non-final stages produce intermediate activations of shape $B \times H$, where B is the microbatch size and H the hidden dimension, and forward them to the next stage. Only the final stage transforms these into *logits* of shape $B \times V$, where V is the vocabulary size. Each row of the logits corresponds to a sequence, and each column to a vocabulary token.

③ *Sampling.* A *softmax* in the final stage converts logits into probabilities. We denote the *logits* from the final layer of a LLM at time step s as $\mathbf{z}_s \in \mathbb{R}^{B\times V}$, where B is the batch size and V is the vocabulary size. For each request $b \in \{1, \ldots, B\}$, $\mathbf{z}_s^{(b)} \in \mathbb{R}^V$ represents the unnormalized log-probabilities over the vocabulary. Let $\mathbf{y}_{<s}^{(b)} = (y_1^{(b)}, \ldots, y_{s-1}^{(b)})$ denote the sequence of previously generated tokens for request b up to step $s-1$. The sampling process can be defined as follows.

(1) Logits adjustment. Modify logits using penalties [4, 26] (e.g., frequency) based on $\mathbf{y}_{<s}^{(b)}$ to promote diversity.

(2) Probability computation. Scale logits by temperature τ [7], apply *softmax*, and optionally restrict candidates using top-k [15] or top-p [23] filtering:

$$\mathbf{p}_i^{(b)} = \text{Filter}\left(\text{softmax}\left(\frac{\text{ApplyPenalty}(\mathbf{z}_s^{(b)}, \mathbf{y}_{<s}^{(b)})}{\tau}\right); k, p\right)$$

(3) Sampling. Draw a token ID from the categorical distribution $y_s^{(b)} \sim \text{Categorical}(\mathbf{p}_s^{(b)}), \quad \mathbf{y}_s \in \mathbb{N}^B$.

For example, at the 0_{th} iteration, top-k sampling may select token IDs 8 and 14 for seq_0 and seq_1 respectively, yielding updated sequences $seq_0 = \langle 6, 7, 9, 11, 3, \underline{8}\rangle$ and $seq_1 = \langle 13, 7, 4, 5, 12, 3, \underline{14}\rangle$. The engine proceeds to the next iteration, repeating until an END-OF-SEQUENCE (EOS) token ID is generated or the maximum sequence length is reached.

2.2 Pipeline Parallelism in LLM Inference

LLMs typically consist of stacked transformer [44] layers, each performing attention and feed-forward computations. Pipeline parallelism (PP) partitions the model along its depth, with each *stage* responsible for a subset of layers. During inference, unfinished sequences are grouped into *microbatches* and processed in a pipelined fashion. For example, with four sequences $\{seq_0, seq_1, seq_2, seq_3\}$ and pipeline degree $p = 2$, one schedule assigns microbatch $\{seq_0, seq_1\}$ to iteration $2k$ and $\{seq_2, seq_3\}$ to $2k + 1$, alternating until completion. Each stage sequentially performs *input preparation* and a *forward pass*, while only the final stage executes *sampling*.

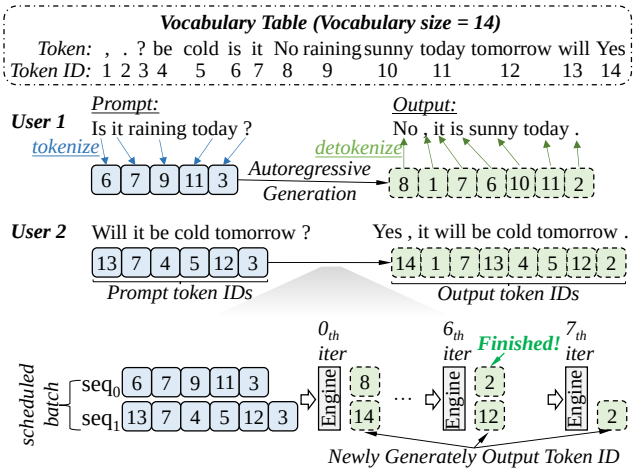


Figure 2. Example of iterative autoregressive generation in LLM serving. Note that the scheduled batch may changed from one iteration to the next—for instance, certain sequences may be preempted by others. The diagram omits these dynamics for clarity.

PP enables inference of LLMs that do not fit on a single GPU and improves throughput. Unlike TP, whose *allreduce* operations scale poorly across nodes, PP restricts communication to adjacent stages and is more bandwidth-efficient —especially in cross-node deployments using Ethernet or InfiniBand. Let N, H, B, t, and p denote the number of layers, hidden size, batch size, TP degree, and PP degree, respectively. The per-GPU communication volume is $C(p,t) = BH\left(\frac{4N(t-1)}{pt} + p - 1\right)$. When $p^2t < 4N$—a condition commonly satisfied in LLM deployments that require 8 or more GPUs under PP (e.g., $N > 50$, $pt \le 16$) [2, 31, 43]—we have $C(p,t) > 2 \times C(2p, t/2)$, suggesting that increasing p and reducing t can substantially lower communication. For small models[3] (e.g., 32B LLMs that require $pt \le 4$), PP brings limited benefit and may introduce pipeline bubbles. But for larger models ($pt \ge 8$), hybrid PP+TP (e.g., $p = 2$, $t = 4$) significantly reduces communication versus pure TP ($p = 1$, $t = 8$). As shown in Figure 1, PP yields up to 4.5× throughput improvement for 16-GPU inference of DeepSeek V3.

[3]Not in the scope of this paper.

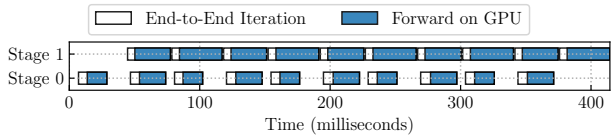


Figure 3. Per-iteration execution breakdown using vLLM on 8 H100 GPUs with Qwen-2.5-72B ($t = 2$, $p = 4$). Bars denote *iteration time*; filled regions represent GPU *forward time*.

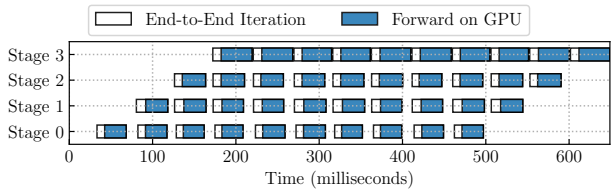


Figure 4. Per-iteration execution breakdown using vLLM on 16 H100 GPUs with DeepSeek V3 ($t = 4$, $p = 4$).

3 Motivation

This section presents key observations on PP and identifies core challenges in achieving high throughput. These observations reveal fundamental inefficiencies in PP inference and motivate the design of SiPipe.

3.1 Observations of PP Inefficiencies in LLM Inference

Figure 3 and Figure 4 show the per-stage breakdown of each iteration during 8-GPU and 16-GPU PP inference with vLLM [27]. These results lead to the following observations:

Observation 1: Load-imbalance bubble. The overall throughput of PP inference is limited by the slowest stage. Profiling across various LLMs and GPU counts consistently shows the final stage has a significantly higher computation load ($22\% - 40\%$)—mainly due to the extra cost of *sampling*. This imbalance causes earlier stages to idle: with PP degree $p = x$, stage 0 at iteration $n + x$ must wait for stage $x - 1$ to complete iteration n. Consequently, the overloaded final stage creates persistent pipeline bubbles, reducing overall efficiency.

Observation 2: Intra-stage bubble. In each iteration, we observe a gap ($12\% - 19\%$) at the beginning of every stage's execution, caused by the CPU-side *input preparation* before launching the GPU-side *forward*. This delay arises from the use of CUDA graphs [18], which require inputs to reside at fixed GPU memory addresses. As a result, CPU-prepared tensors must be synchronously copied into pre-allocated buffers before each CUDA graph replay.

In frameworks like vLLM, input preparation is deferred until the previous *forward* completes to ensure correctness, avoiding race conditions in graph execution. This forces the GPU to wait for the CPU to finish preparation, creating intra-stage bubbles. While more asynchronous designs are possible, they risk correctness issues (see Section 3.2).

Observation 3: Inter-stage bubble. *inter-stage bubbles* become non-negligible after an even p-stage partition, which consist of two components: synchronization overhead due to data dependencies of adjacent stages (1.4–2.6 ms) and transmission overhead of activations (1–2 ms). Figures 3 and 4 reveal that even with an even p-stage partition, the *forward* execution time varies across stages with a standard deviation of $3\% - 7\%$. This variation amplifies synchronization cost.

For example, suppose stage 0 takes t_0^{n+1} to complete iteration $n + 1$, while stage 1 takes t_1^n to complete iteration n, where $t_0^{n+1} < t_1^n$. If both begin at time t, stage 0 becomes idle since $t + t_0^{n+1}$, waiting to send hidden states to stage 1. The interval $[t + t_0^{n+1},\ t + t_1^n]$ is a *communication stall* induced solely by stage-wise synchronization on data dependencies.

3.2 Challenges and Solutions

Eliminating bubbles in PP requires addressing challenges in computation efficiency, dependency management, and overlapping communication. We next detail each challenge and show how underutilized CPUs are leveraged to mitigate them.

Challenge 1: *Decoupling sampling to preserve pipeline balance.* To eliminate load-imbalance bubbles, we first identify their root cause: *The final pipeline stage performs an additional sampling step*. In typical PP, sampling is naturally colocated with the final stage, as this stage produces the logits. However, this design introduces imbalance: while other stages proceed to the next microbatch after *forward*, the final stage is stalled by the additional, sequential sampling step. This creates a serialization point that degrades overall throughput.

A natural idea is to offload sampling to a dedicated GPU. However, in large-scale deployments where all GPUs are fully utilized (e.g., $t = 2$, $p = 4$ on an 8-GPU node), these options are impractical. Sampling is too light and sequential to warrant an exclusive GPU, and reshaping the pipeline to hide its latency (such as giving the final stage fewer layers) forces hardware-specific workload balancing, reducing portability. Because these strategies entangle sampling with model partitioning, they also impede modular scaling. Efficiently decoupling sampling thus remains a key challenge.

Solution 1: *SiPipe adopts a column-wise layout and incremental computation to accelerate CPU-based sampling.* SiPipe offloads the sampling process to multiple CPUs to balance GPU workload in PP. To avoid the CPU becoming a straggler during high-throughput LLM inference, SiPipe introduces a column-wise data layout to enable efficient in-place updates. Built on this layout, SiPipe incrementally updates cross-iteration metadata, avoiding redundant computation and memory allocation. These optimizations exploit inter-batch similarity, reduce CPU latency, and eliminate sampling-induced load-imbalance bubbles. (§5.1)

Challenge 2: *Cross-iteration input conflicts under static execution models.* A natural way to eliminate intra-stage bubbles is to decouple CPU and GPU execution—preparing inputs asynchronously on the CPU while the GPU performs the *forward* pass. However, to minimize runtime overhead, modern inference engines often adopt static execution models,

i.e., CUDA graphs [18], which fix input bindings and kernel sequences. While efficient, this approach leads to subtle *write-after-read (WAR) hazards*: when the CPU and GPU run asynchronously, inputs for iteration j may overwrite buffers that the GPUs is still reading from iteration $i < j$. These conflicts arise from temporal asymmetry between CPU and GPU, rigid memory bindings in CUDA graphs, and the evolving nature of model inputs in autoregressive generation.

A common workaround—staging model input tensors in temporary buffers and copying them into static regions before launch *forward*—avoids conflicts but incurs extra memory copies, reintroducing synchronization bottlenecks and offsetting the benefits of CUDA Graphs. These limitations call for a principled solution that enables safe decoupling without sacrificing the efficiency of static execution.

Solution 2: ***SiPipe adopts token-safe execution model (TSEM) to decouple CPU and GPU execution.*** To eliminate intra-stage bubbles caused by GPU stalls during CPU-side input preparation, SiPipe introduces the *TSEM*, which safely decouples input preparation from *forward* while preserving static graph constraints. TSEM pre-captures two CUDA graphs per batch size, each bound to a distinct shared buffer. Iterations alternate between these graphs, allowing the CPU to prepare inputs in one buffer while the GPU reads the other. This *shadow buffering*-based scheme [22] ensures version isolation without requiring runtime graph modifications. By enabling asynchronous CPU-GPU execution, TSEM eliminates intra-stage bubbles. (§5.2)

Challenge 3: ***Efficient transmission of dynamic hidden states across pipeline stages.*** In PP, each stage transmits a tensor dictionary of *hidden states* (activations and residuals [21]) to the next stage every iteration. Since tensor sizes vary dynamically and cannot be statically pre-allocated, packing and transmission must wait until *forward* passes complete, delaying communication. This uncertainty causes *communication stall* (§3.1) and requires *multi-round metadata exchanges* to determine tensor sizes, inflating inter-stage bubbles and reducing pipeline efficiency.

Solution 3: ***SiPipe adopts a structure-aware transmission (SAT) strategy to enable asynchronous communication between stages.*** Although hidden states change across iterations, their structure remains mostly stable. SiPipe exploits this property via SAT, which captures the invariant structure and infers per-iteration changes from scheduling output—eliminating the need to treat each tensor dictionary as fully dynamic. Once the structure is identified, an asynchronous CPU thread in the receiving process handles data pre-allocation, management, and transmission for each iteration. As a result, downstream stages can directly access the prepared data without waiting, eliminating transmission latency. SAT reduces multi-round metadata exchanges, avoids redundant metadata processing, shortens inter-stage bubbles, and removes communication stalls through its asynchronous design. (§5.3)

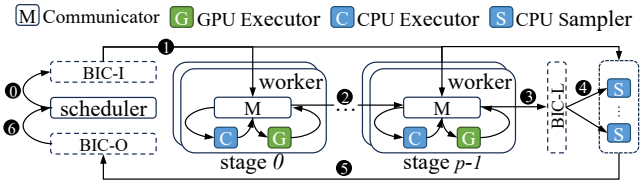


Figure 5. Architecture and workflow of SiPipe.

4 Overview

4.1 System Architecture

Figure 5 shows the pipelined parallel inference architecture of SiPipe, comprising four key components: the *scheduler*, *workers*, *CPU samplers*, and the *Buffered IPC Channel (BIC)*. In each iteration, the scheduler selects a microbatch from unfinished sequences and generates a *scheduling output*, which is broadcast to all workers. The pipeline consists of p stages, each with t *workers* executing TP inference over a model shard. The output of stage i (i.e., the hidden states) becomes the input to stage i+1. Each worker includes three modules: (1) a *communicator* that handles inter-stage data transfer and internal coordination; (2) a *CPU executor* for input preparation and cached state management; (3) a *GPU executor* for the *forward* pass. After the final stage, a pool of *CPU samplers* receives the logits and performs parallel sampling. To enable low-latency communication between components, SiPipe introduces the BIC (detailed in §6), which serves as a message queue for different data types: *BIC-I* for scheduling outputs, *BIC-L* for logits, and *BIC-O* for sampling outputs.

4.2 Execution Flow

We illustrate the SiPipe workflow with an example. Suppose we process 4 sequences ($\{seq_0, seq_1, seq_2, seq_3\}$) with a PP degree of p=2, resulting in a microbatch size of $\frac{4}{p} = 2$. At initialization, the scheduler dispatches p consecutive scheduling outputs to workers via *BIC-I* (❶). Thereafter, upon receiving sampling output for iteration n, it immediately dispatches iteration $n + p$. This pipelined approach keeps all workers busy, maximizing resource utilization. For example, the scheduler may assign microbatches $\{seq_0, seq_1\}$ and $\{seq_2, seq_3\}$ to iteration $2k$ and iteration $2k + 1$, respectively.

Each worker handles sequences as follows: the *CPU executor* asynchronously updates the sequence cache, generates model input, and forwards it to the *communicator*. Concurrently, the *GPU executor* polls the communicator and launches *forward* passes upon receiving new model input. For stage-0 workers, execution triggers on model input availability alone (❶). Subsequent stages wait for both model input and hidden states from the previous stage. Intermediate stages forward hidden states downstream (❷), while the final stage sends logits to CPU samplers via *BIC-L* (❸❹).

Each CPU sampler then generates sampling metadata for the microbatch and performs sampling using the logits. Sampling outputs are sent back to the scheduler via *BIC-O* (❺), triggering the next iteration (❻).

5 Design

In this section, we present SiPipe's core designs for eliminating pipeline inefficiencies, focusing on load-imbalance bubbles (§5.1), intra-stage bubbles (§5.2), and inter-stage bubbles (§5.3).

5.1 Column-Wise CPU Sampling via Metadata Reuse

Offloading sampling to the CPU reduces load-imbalance bubbles by balancing GPU workloads but introduces two main challenges. First, sampling relies on dynamic metadata (e.g., output token IDs) that change each iteration. Intermediate steps—such as penalty application, softmax, and filtering—produce many temporary tensors, causing frequent memory allocations and stressing CPU memory management. Second, sampling is compute-intensive and requires careful CPU-side optimization to maintain high throughput.

For instance, given logits $\mathbf{Z} \in \mathbb{R}^{B\times V}$ and previously output token IDs $\mathbf{Y} \in \mathbb{N}^{B\times(s-1)}$, we compute a penalty tensor $\mathbf{f} = \text{CalcPenalty}(\mathbf{Y})$, where each row $\mathbf{f}_{i,:}$ encodes token frequency, presence, or repetition in sequence i. The penalized logits are $\mathbf{Z}' = \mathbf{Z} - \alpha \cdot \mathbf{f}$, with α as a tunable hyperparameter. Then temperature scaling is applied: $\mathbf{Z}'' = \mathbf{Z}'/\tau$, $\tau > 0$, followed by a row-wise softmax $\mathbf{P}_{i,j} = \frac{\exp(\mathbf{Z}''_{i,j})}{\sum_{v=1}^{V}\exp(\mathbf{Z}''_{i,v})}$. Filtering methods such as top-k and top-p sampling optionally restrict candidate tokens before sampling. Since $\mathbf{Y}$ changes every iteration, these computations and memory allocations must be repeated.

Dynamic $\mathbf{Y}$ and $\mathbf{Z}$ impose significant overhead when batch size B and vocabulary size V are large—a common setting in LLM serving. For instance, Qwen-2.5-72B (V = 151,643, B = 256) requires approximately 300 MB per penalty tensor (double precision). To sustain high inference throughput, sampling must finish within the decoding slack—typically 1–2 ms per sampling strategy—demanding over 150 GB/s memory bandwidth solely for penalty operations. While this falls below DDR5's peak bandwidth, contention under high concurrency reduces effective utilization. Additionally, element-wise computations on large matrices further increase computational cost as B and V grow. Without memory reuse or incremental updates, penalty calculation becomes a critical CPU bottleneck, limiting overall throughput.

To mitigate memory allocation overhead and facilitate reuse across iterations, we propose a *column-wise data layout* that transposes the model logits $\mathbf{Z} \in \mathbb{R}^{B\times V}$ and $\mathbf{Y} \in \mathbb{N}^{B\times(s-1)}$ into $\mathbf{Z}^\top \in \mathbb{R}^{V\times B}$ and $\mathbf{Y}^\top \in \mathbb{N}^{(s-1)\times B}$, respectively. This reordering forms the basis of a redesigned CPU sampling that directly improves both *computational efficiency* and *memory access locality*. Specifically, SiPipe reuses this transposed structure to incrementally construct penalties and optimize downstream sampling steps:

(1) Memory reuse for output sequences. We preallocate a maximum-length output buffer $\mathbf{Y} \in \mathbb{N}^{L_{\max}\times B}$. In the transposed layout, new token IDs are appended as rows, enabling fast in-place update and avoiding tensor reshaping. This also ensures that all subsequent metadata (e.g., frequency matrices) can be updated incrementally.

(2) Incremental penalty construction. For penalties such as frequency, repetition, and presence, we maintain a shared buffer[4] $\mathbf{f} \in \mathbb{R}^{V\times B}$ aligned with the transposed logits $\mathbf{Z}^\top$. At each iteration, only the B elements of $\mathbf{f}$ that correspond to the newly generated token IDs are incrementally updated in-place across all sequences in the microbatch. This approach enables efficient computation of $\alpha \cdot \mathbf{f}$ with minimal overhead, while being cache-friendly and avoiding reconstruction of the entire penalty tensor. Consequently, penalty adjustment reduces to a single vectorized subtraction $\mathbf{Z}^\top := \mathbf{Z}^\top - \alpha \cdot \mathbf{f}$, eliminating the need to regenerate $\mathbf{f}$ from scratch at every iteration.

(3) Low-cost pipelined integration. In each stage, each worker typically generates a logits shard of shape $B \times \frac{V}{t}$. By employing a column-wise layout, these shards can be locally transposed to $\frac{V}{t} \times B$ and directly concatenated along the row dimension to form the global matrix $\mathbf{Z}^\top \in \mathbb{R}^{V\times B}$. This approach avoids costly all-gather operations or row-wise tensor reconstructions.

By redesigning the sampling process to leverage the column-major layout, we unify key sampling operations into in-place transformations on the logits tensor $\mathbf{Z}^\top$. This eliminates repeated intermediate tensor allocations and data copying. All operations run in a memory- and cache-efficient manner, enabling vectorized computation and high throughput. The resulting design forms a lean, scalable CPU-side sampling pipeline that is both elegant and efficient in practice.

SiPipe relies on the assumption that consecutively scheduled batches are identical or exhibit high similarity (i.e., the microbatch size B remains constant across iterations), thereby permitting incremental updates through minor adjustments to $\mathbf{Y}$. Although this assumption may appear restrictive, it is well-justified in large-scale LLM inference scenarios employing scheduling strategies such as *contiguous batching* or *iteration batching* [8, 16, 27, 46]. In particular, within TP, adjacent batches are highly likely to be identical. Similarly, in PP with parallelism degree p, batches indexed by n and $n + p$ are expected to be identical, while batches indexed by n and $n + j$ for $0 < j < p$ are orthogonal. Consequently, maintaining only p distinct column-wise data replicas—corresponding to the pipeline stages—is sufficient to support efficient incremental computation without violating this assumption.

The memory overhead on the host for each replica is bounded by $n \times B \times V + B \times L_{\max}$, where B typically does not exceed 1024, and $n = 3$ corresponds to the frequency, presence, and repetition penalty buffers. Moreover, $L_{\max}$ and V for most SOTA LLMs rarely exceed 128K and 200K, respectively. Consequently, the total memory consumption remains well within reasonable limits ($\approx$ 3-4 GB), making this overhead acceptable for typical host memory capacities.

[4]Each penalty type has its own buffer and follows a similar processing routine.

5.2 Token-Safe Execution Model with CPU-Assist

To eliminate intra-stage bubbles, SiPipe decouples CPU and GPU executors for asynchronous execution. In PP, microbatches scheduled across successive iterations are non-overlapping sequence sets. Therefore, the CPU executor can safely prepare inputs for iteration $i+1$ while the GPU executor performs *forward* on iteration i. Without data dependencies between adjacent microbatches, CPU-GPU concurrency is both safe and effective. This insight underpins our decoupled execution strategy, enabling full utilization of CPU and GPU resources without compromising correctness.

However, this asynchronous CPU-GPU execution conflicts with CUDA graphs, which rely on static buffer bindings that assume inputs remain unchanged across iterations. While CUDA graphs reduce launch overhead by capturing fixed kernel execution sequences and their memory bindings, this static model causes WAR hazards when the CPU overwrites data still in use by the GPU, threatening correctness (see Challenge 2, §3.2).

To address cross-iteration WAR hazards under static execution models, we propose the *token-safe execution model (TSEM)*, a mechanism that ensures conflict-free pipeline inference. TSEM introduces an asynchronous CPU executor dedicated to input preparation, which operates in parallel with the GPU executor while maintaining compatibility with statically captured CUDA graphs. To coordinate these components without introducing blocking or direct dependencies, TSEM adopts a message-driven architecture (Figure 6), where a dedicated *communicator* orchestrates execution by dispatching events to the CPU and GPU executors. This design decouples input preparation from *forward* and eliminates idle time due to executor synchronization, allowing both components to make forward progress independently.

The GPU executor employs a versioned buffer mechanism, where each type of input tensor maintains two physical versions (v_0 and v_1). For each version, we capture a set of static CUDA graphs, each corresponding to a different batch size. A graph is selected based on a tuple $<v_0/v_1,\ batch\_size>$, enabling the executor to alternate between versions across iterations and avoid WAR hazards. The CPU executor maintains a *SequenceCache* that maps each sequence to its cached metadata, including prompt/output token IDs and KV-related state, minimizing redundant data transfer when sequences are repeatedly scheduled. It also prepares per-iteration *BatchMetadata*, which includes preprocessed CPU tensors (e.g., attention inputs) for each microbatch. To reduce recomputation, p versions of BatchMetadata are maintained (where p is PP degree), leveraging the high similarity between microbatches in iteration i and $i+p$ (§5.1). The communicator coordinates execution by managing two message queues: one for scheduling outputs and one for model inputs. It buffers scheduling outputs from the scheduler and model inputs from

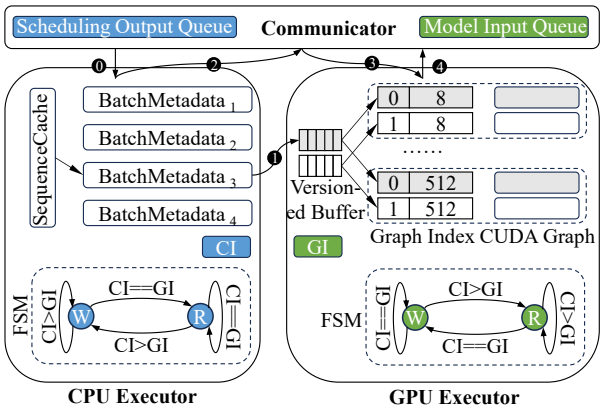


Figure 6. Illustration of the token-safe execution model for conflict-free GPU execution.

the CPU executor, and asynchronously transmits GPU executor outputs to the next stage while receiving hidden states from the previous one. To drive progress, it maintains two indicators—the CPU indicator (CI) and GPU indicator (GI)—which track the latest completed iteration of each executor.

Each executor operates as a finite-state machine (FSM) with two states: `wait (W)` and `running (R)`. Initially, both CI and GI are set to -1. The CPU executor transitions from W to R—or remains in R—whenever `CI == GI`, indicating that all previously generated model inputs have been consumed. In state R, it consumes a scheduling output from the communicator's *scheduling output queue* (❶), updates the SequenceCache and the corresponding BatchMetadata (indexed by i mod p, where i = `CI`), and writes the prepared input tensors into the versioned input buffer (version i mod 2, ❶). It then enqueues a lightweight model input descriptor (e.g., batch size) into the model input queue and increments CI (❷). The GPU executor follows a similar FSM structure, with a key distinction: it increments GI immediately after entering or remaining in state R, thereby allowing the CPU executor to asynchronously prepare the next iteration's input. Upon entering R, it fetches a model input from the model input queue (❸), locates the correct versioned buffer and CUDA graph using the index `<(GI-1) mod2, batch_size>`, executes the graph, and forwards the output to the next stage via the communicator (❹). This FSM-driven design decouples CPU and GPU progress while maintaining strict correctness, enabling efficient overlap of input preparation and *forward* pass.

TSEM eliminates intra-stage bubbles by decoupling CPU input preparation from GPU execution, ensuring that each new iteration can immediately begin once the previous *forward* pass completes. This tight coordination maximizes pipeline utilization and sustains high throughput. The overhead of TSEM is modest, requiring only an additional static input buffer and a few CUDA graphs. This cost is effectively neutralized by offloading sampling to the CPU, which reclaims GPU memory previously allocated for sampling metadata, e.g., penalty tensor $\mathbf{f}$.

5.3 Structure-Aware Hidden State Transmission

In PP, consecutive stages exhibit strict data dependencies: the output (i.e., hidden states) of stage i serves as the input to stage i+1 for every microbatch. This mandates a data handoff between adjacent stages at each iteration. Cross-stage communication is inherently serialized. Stage i+1 must wait until it fully receives and deserializes the hidden states from stage i before initiating *forward* pass. This strict dependency leads to *inter-stage bubbles*, where communication latency hinders overall pipeline utilization. Since tensor shapes and sizes of hidden states may vary across microbatches, the receiving stage cannot initiate partial *forward* and must wait for full transmission and deserialization.

Consider transmitting hidden states structured as {key1: tensor_1, key2: tensor_2}. As shown in Figure 7(a), state-of-the-art inference engines typically adopt a *structure-unaware transmission* strategy. In this approach, the receiver lacks prior knowledge of the dictionary layout or tensor sizes, preventing asynchronous communication and enforcing strict synchronization. The sender first serializes metadata containing the number of entries, the list of keys, and per-tensor attributes such as shape, dtype, and device (❶). Because the receiver does not know the size of this metadata blob, it must first allocate a temporary buffer to receive the blob size (❶), then perform a communication round to obtain it (❷). Once the size is known, it allocates a full metadata buffer (❸) and receives the actual blob in a second communication round (❹). After deserialization (❺), the receiver sequentially allocates memory and receives each tensor one by one (❻–❽). Finally, it reconstructs the hidden state by pairing each key with its corresponding tensor (❾). This serialized, step-wise process requires multiple tightly coupled communication rounds and memory allocations, significantly delaying stage activation and exacerbating inter-stage bubbles.

For a running engine with the fixed model type and GPU assignment, the structure of the hidden state dictionary remains largely static across iterations. In typical models, this dictionary contains a small, fixed set of tensors, such as the output of the final feed-forward layer and the residual connection. As a result, both the number of entries and their string keys are predictable. Each tensor typically has a shape of $x \times y$, where y (the hidden size) is model-dependent and fixed, whereas x (the batch size) may vary across iterations. All other attributes, including data type and device, remain unchanged.

Figure 7(b) illustrates SiPipe's structure-aware transmission mechanism, which avoids redundant metadata communication by separating static and dynamic components of the hidden states. The mechanism consists of two key components: static structure capture and dynamic batch size retrieval. In the first iteration, the receiver performs a full transmission using the structure-unaware transmission. During the reconstruction step, however, it additionally extracts and stores

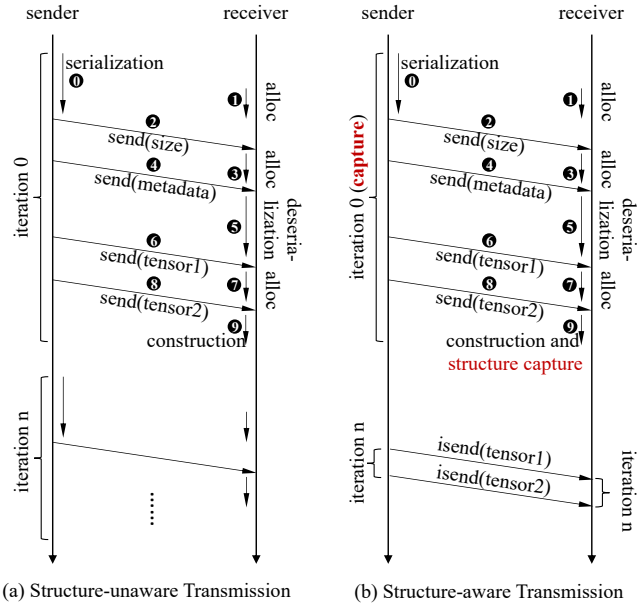


Figure 7. Effect of structure-aware vs. structure-unaware transmission when transferring variable-size tensors across multiple iterations.

the static structure of the dictionary (❾). Since the scheduler sends per-iteration scheduling output to each worker, the receiver can directly extract the only dynamic factor—the batch size—based on the number of sequences assigned to that stage.

With both static structure and batch size available, the receiver can pre-allocate memory and post asynchronous `irecv` operations before transmission begins, avoiding *communication stalls* caused by unfinished *forward* on GPU. The sender transmits hidden states using asynchronous `isend`, overlapping communication with ongoing computation. As shown in Figure 7(b), metadata-related serialization, deserialization, and synchronization are skipped entirely during steady-state iterations.

Our structure-aware mechanism provides two key advantages over the structure-unaware baseline. First, by removing synchronous metadata communication, it decouples adjacent pipeline stages. In the structure-unaware setting, stage i may stall due to blocking communication. In contrast, our design enables pre-buffering and asynchronous receives to eliminate these communication stalls. Second, although hidden states are small—typically requiring only hundreds of microseconds to transfer over RDMA or NVLink—the overhead from metadata handling and control synchronization in the structure-unaware design dominates communication cost. By bypassing these steps, our approach brings inter-stage latency down from 2–5 ms to the pure data transfer overhead of less than 500μs, which is further hidden by overlapping communication with computation asynchronously (reducing the effective inter-stage bubble to ~100μs).

6 Implementation

SiPipe is implemented in pure Python to maximize compatibility with existing inference frameworks. Specifically, it is implemented as a plugin for `vLLM` [27], inheriting from key Python classes and overriding selected methods to integrate seamlessly.

Unlike conventional frameworks where most computation is offloaded to GPUs, SiPipe performs sampling on the CPU. Due to Python's Global Interpreter Lock (GIL), we adopt a multi-process architecture to fully exploit CPU parallelism.[5] This leads to substantial inter-process communication, which we categorize into two patterns: *dispatch* and *combine*.

For dispatch—broadcasting scheduling outputs from the scheduler to all worker and sampler processes, and broadcasting logits from the final-stage GPUs to all samplers—we design *Buffered IPC Channels (BIC)* using shared-memory ring buffers with a *lock-ahead* mechanism. We instantiate *BIC-I* for scheduling output and *BIC-L* for logits transfer. Each BIC maintains N slots, each guarded by a mutex implemented using `fcntl` file locks, enabling lightweight, cross-process synchronization without busy-waiting. In iteration n, the producer pre-acquires an exclusive (write) lock on slot $(n+1) \bmod N$, writes data to slot $n \bmod N$, and then releases the $(n \bmod N)_{th}$ slot's exclusive lock—ensuring lock-ahead progression without contention. Consumers poll slots sequentially, acquiring a shared (read) lock when a slot becomes available, which allows concurrent read access without blocking other consumers. For cross-host communication (e.g., between scheduler and remote workers), each host maintains a BIC replica, and a background daemon handles data synchronization across hosts.

For combine—aggregating sampling results from all samplers back to the scheduler—we design *BIC-O*, a TCP-based multi-producer ring buffer. Each slot in BIC-O contains multiple subslots, one per sampler. In iteration n, sampler i writes its output to subslot i of slot $n \bmod N$. Once all subslots are filled, the scheduler detects that the microbatch is completed. Since each sampling result is typically just a token ID (i.e., an integer), this TCP-based approach is lightweight and avoids the complexity of emulating shared memory across hosts.

7 Evaluation

This section answers four key questions:
(1) Does SiPipe deliver performance gains across diverse parallel inference configurations? (§7.2)
(2) How does SiPipe impact end-to-end inference latency across diverse configurations? (§7.3)
(3) How does SiPipe affect compute resource utilization across diverse configurations? (§7.4)
(4) What is the contribution of each component in SiPipe's design? (§7.5)

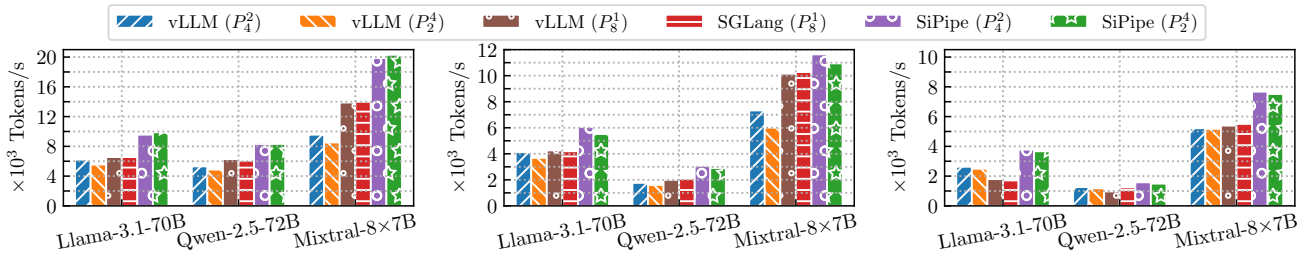
[5]Future versions will explore Python 3.13's *free-threaded* mode to simplify the implementation.

7.1 Experimental Setup

Testbed. Our evaluation is conducted on following testbeds: `H100-NVL`, `A100-NVL`, and `A100-PCIe`. Each server in H100-NVL and A100-NVL is equipped with 8 H100 or A100 GPUs connected via NVLink, respectively, while each A100-PCIe server contains 8 A100 GPUs connected via PCIe. All GPUs have 80 GB memory. Each server also features a 192-logical-core Intel(R) Xeon(R) Platinum 8468 CPU and 2 TB of host memory. Servers within the same testbed are connected via 400 Gbps ConnectX-7 NICs.
Baseline. We compare SiPipe against two of the most popular inference engines: `vLLM` [27] and `SGLang` [47]. Since SGLang does not support PP, we evaluate only its TP mode. For vLLM, we evaluate both TP and PP modes.
Models. We evaluate SiPipe using six LLMs, categorized into two groups: *large* LLMs—Llama-3.1-70B [37], Qwen-2.5-72B [43], and Mixtral-8×7B [24]; and *ultra-large* LLMs —DeepSeek V2.5 [30], DeepSeek V3 [31], and Llama-3.1-405B [37]. Based on deployment experience, *large* LLMs are typically served on 8 GPUs [1, 2], whereas the *ultra-large* models require 16 GPUs because their weights alone saturate the memory of 8 GPUs. We focus on LLMs of this scale because PP is primarily intended for multi-GPU deployments. In contrast, for smaller LLMs (e.g., 14B), pure TP is a more suitable choice. Optimizing TP performance for small LLMs is beyond the scope of this work. Data parallelism is also omitted: instances with fewer GPUs lack sufficient memory for both model weights and KV caches.
Configuration. The default batch size is 512 for 8-GPU and 1024 for 16-GPU experiments. In pure TP mode, PP is disabled ($p = 1$), and the TP degree t equals the total number of GPUs. In PP mode, each stage forms a TP group assigned to a contiguous block of LLM layers, with pt equal to the GPU count. Empirically, setting $t = 4$ per stage delivers near-optimal intra-stage efficiency [6]. Samplers are colocated with the final stage. On this node, each worker occupies an isolated CPU core for *input preparation* (8 cores total), while the remaining cores are dedicated to *sampling*.
Workload. To ensure fair and reproducible comparison, we randomly select a fixed set of prompts from the ShareGPT dataset [5] and reuse them across all experiments. Decoding is performed using all common sampling strategies, including top-p, top-k, min-p, temperature, and penalties for repetition, presence, and frequency.
Metrics. We report throughput, per-token latency, and resource utilization. Throughput is measured in tokens per second, and per-token latency reflects the average generation time per token. Utilization metrics reflect hardware efficiency.

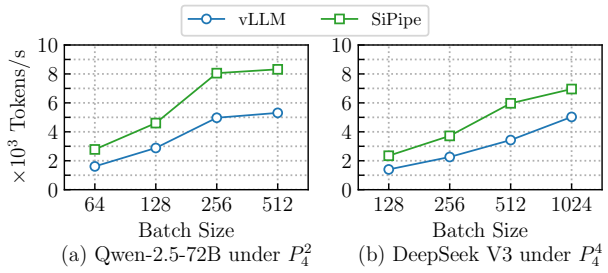
7.2 Throughput Gains under Diverse Parallel Settings

Throughput across LLMs and testbeds. Figure 8(a) shows the throughput of *large* LLMs on the `H100-NVL` testbed using different inference engines and configurations. Compared



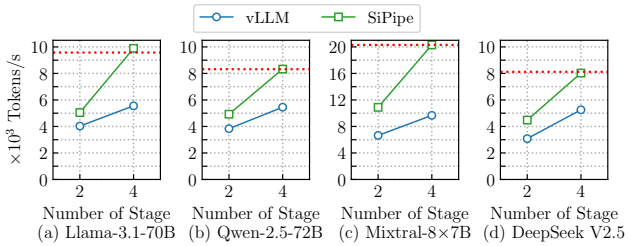
(a) Using 8×H100 on the H100-NVL testbed (b) Using 8×A100 on the A100-NVL testbed (c) Using 8×A100 on the A100-PCIe testbed

Figure 8. Throughput comparison of different engines under various parallel configurations, evaluated on multiple testbeds. Each configuration is denoted as P_j^i, where PP degree $p = i$ and the TP degree $t = j$. Specifically, P_t^1 denotes the pure TP mode.



(a) Qwen-2.5-72B under P_4^2 (b) DeepSeek V3 under P_4^4

Figure 9. Effect of batch size on inference throughput.



(a) Llama-3.1-70B (b) Qwen-2.5-72B (c) Mixtral-8×7B (d) DeepSeek V2.5

Figure 10. Effect of the number of pipeline stages (i.e., PP degree p) on throughput. In the first three subfigures, each stage uses a TP degree of t=2, while the last subfigure uses t=4. The dashed line denotes the performance of SiPipe under the default configuration.

to vLLM (P_4^2), SiPipe achieves a performance gain of 1.6× to 2.1× by eliminating pipeline bubbles that degrade vLLM's PP mode, especially as the number of stages increases (e.g., from P_4^2 to P_2^4 in vLLM). In this single-node scenario, we also observe that both vLLM and SGLang in pure TP mode outperform vLLM's PP mode due to NVLink-enabled efficient GPU communication and better TP scalability. However, as each PP stage has a smaller TP degree (2 or 4) compared to the pure TP modes ($t = 8$), using PP cuts *all-reduce* overhead [10, 41] and is inherently more efficient. Building on this, SiPipe further addresses the inefficiencies of inter-stage coordination in prior PP implementation, enabling it to outperform pure TP modes in overall throughput.

Figure 8(b) and Figure 8(c) show results on the `A100-NVL` and `A100-PCIe` testbeds, respectively. SiPipe consistently outperforms existing frameworks across GPU architectures and interconnects, demonstrating strong generality and robustness in both NVLink- and PCIe-based environments. Notably, the speedup on A100 is smaller than on H100, as expected. H100's 3× higher compute capability makes pipeline bubbles a more significant bottleneck relative to the *forward* pass. On the A100, by contrast, the *forward* pass dominates the runtime. This highlights an important trend—future accelerators will further reduce the relative cost of *forward*, making pipeline inefficiencies more pronounced. SiPipe aligns with this trend by minimizing stalls and maximizing pipeline efficiency.

Next, we revisit the 16-GPU inference results for *ultra-large* LLMs, previously shown in Figure 1. In this cross-node setting, SiPipe consistently outperforms all other inference engines, achieving 1.4×-1.7× improvement over the best baseline. The speedup is relatively lower than that observed for *large* LLMs, primarily because the *forward* pass in *ultra-large* models takes longer to compute, which reduces the relative impact of pipeline bubbles. Notably, pure TP configurations perform worse than PP in this scenario. As confirmed in prior studies [41], the *allreduce* communication in TP scales poorly across machines, making them a major performance bottleneck.

Impact of batch size on throughput. Because production workloads rarely use a fixed batch size, we swept the batch size to gauge SiPipe's robustness and scalability. Figure 9 presents throughput comparison for both *large* and *ultra-large* LLMs, each with a representative LLM. Under these batch sizes, SiPipe consistently delivers stable performance gains. Compared to vLLM, SiPipe achieves a speedup of 1.6-1.7× on Qwen-2.5-72B and 1.4-.17× on DeepSeek V3, demonstrating that the optimizations in SiPipe remain effective even under low-load scenarios (i.e., small batch sizes).

Scalability under multi-GPU settings. Figure 10 demonstrates the scalability of SiPipe with varying number of GPUs. In LLM inference, it is typical to reserve KV cache memory amounting to 2–3× the LLM size [27, 41]. For example, deploying a 70B LLM generally requires > 4 GPUs [2, 6]. However, due to poor scalability in existing inference engines, adding GPUs to a single instance often yields less benefit than running multiple smaller ones–an approach that only works when all sequences are short, which is rarely the case in production.

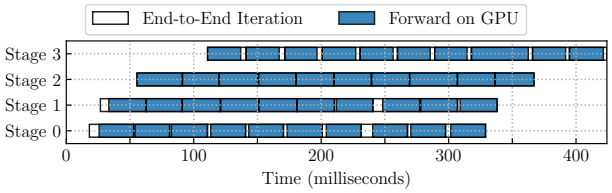


Figure 11. Per-iteration execution breakdown of each pipeline stage using SiPipe (P_4^4) on 16 H100 GPUs in the H100-NVL testbed with the DeepSeek V3.

We therefore compare one-instance scaling directly. *large* LLMs use $t = 2$ while *ultra-large* models use $t = 4$. As shown in Figure 10, vLLM consistently falls short of linear scalability, achieving only 1.4× to 1.7× speedup. In contrast, SiPipe achieves 1.8× to 2.0× scaling. The improvement arises because doubling GPUs expands KV-cache capacity super-linearly, relieving memory contention; SiPipe's pipeline optimizations convert this headroom into higher effective throughput. Improved scalability lets more memory be devoted to KV caches, cutting request preemption and expensive CPU–GPU cache transfers in large workloads. It also reduces the number of instances required to saturate the cluster, easing inter-instance load imbalance [47].

Pipeline efficiency and bubble analysis. We analyze the execution timeline to evaluate the pipeline efficiency of SiPipe. Figure 11 presents a per-iteration execution breakdown of each stage using DeepSeek V3—the largest LLM in our evaluation and the one with the lowest relative performance gain. As shown, most pipeline bubbles have been eliminated, and workload is reasonably balanced across stages. The last stage is no longer a performance bottleneck. Some bubbles remain in the pipeline due to intrinsic workload asymmetry arising from the MoE architecture, which is orthogonal to the pipeline optimizations proposed in this paper.

7.3 Latency Reduction under Different Configurations

Time-to-First-Token (TTFT). TTFT measures the delay from request arrival to the first output token and is mainly affected by scheduling and queuing, which are outside the scope of SiPipe.

Time-per-Output-Token (TPOT). TPOT measures the average latency incurred for generating each output token during the decoding phase. It directly reflects the efficiency of per-iteration execution in the pipeline. Figure 12 and Figure 13 present the cumulative distribution (CDF) of per-iteration TPOT across *large* and *ultra-large* LLMs, respectively. For *large* LLMs, SiPipe reduces the average TPOT by up to 31%, while for *ultra-large* LLMs the reduction reaches 43%. These improvements stem from SiPipe's elimination of pipeline bubbles, which significantly shortens the execution time of each iteration.

We observe more substantial gains in *ultra-large* models, where deeper pipelines amplify the impact of PP bubbles. In PP with degree p, a stall of duration Δt in any single stage

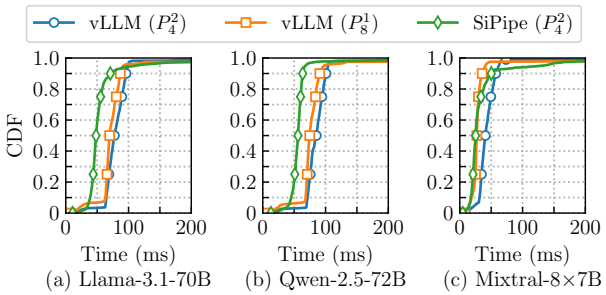


Figure 12. TPOT comparison of different enginess, evaluated with 8×H100 on H100-NVL testbed.

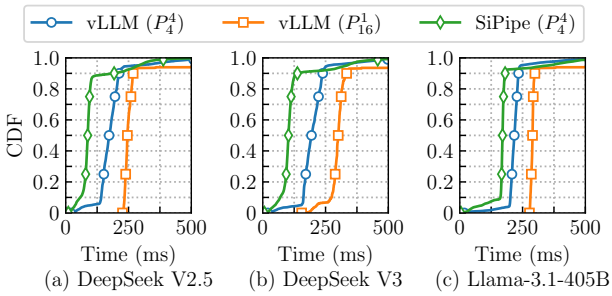


Figure 13. TPOT comparison of different enginess, evaluated with 16×H100 on H100-NVL testbed.

can delay the entire pipeline by up to $p\Delta t$, as all p stages are sequentially blocked. Notably, for Mixtral-8×7B, SiPipe's TPOT is comparable to that of the pure TP baseline. This model features faster *forward* execution than others in its category due to its sparse MoE architecture. In single-node settings, pure TP benefits from avoiding inter-stage communication, while SiPipe's design introduces coordination overhead that offsets its efficiency gains at the TPOT level.

7.4 Resource Utilization Analysis

GPU utilization. We report GPU utilization to assess how effectively each approach leverages hardware resources. Figure 14 presents the average GPU utilization across all devices for both *large* and *ultra-large* LLMs.

In the 8-GPU *large* model setting (Figure 14(a)), SiPipe achieves an average utilization of approximately 85%, 23% higher than that of vLLM (PP). While SiPipe and the pure TP baseline exhibit similar utilization, pure TP consumes more streaming multiprocessor (SM) resources for *all-reduce* communication. Consequently, SiPipe delivers higher throughput under comparable GPU utilization.

In the 16-GPU *ultra-large* setting (Figure 14(b)), SiPipe achieves over 95% average utilization, surpassing all baselines. This improvement stems from eliminating pipeline bubbles in PP baselines and avoiding the underutilization caused by slow cross-node *all-reduce* in pure TP.

CPU utilization. A key distinction of SiPipe compared to other inference engines is its effective use of otherwise underutilized CPU resources to accelerate inference. Figure 15 SiPipe shows that this strategy lifts average CPU utilization by

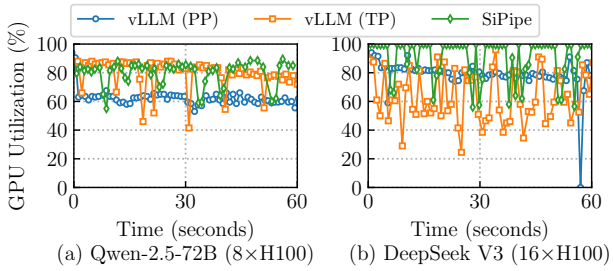


Figure 14. GPU utilization over time on H100-NVL testbed. vLLM (TP) and SiPipe adopt the default configuration, while vLLM (TP) using pure TP configuration.

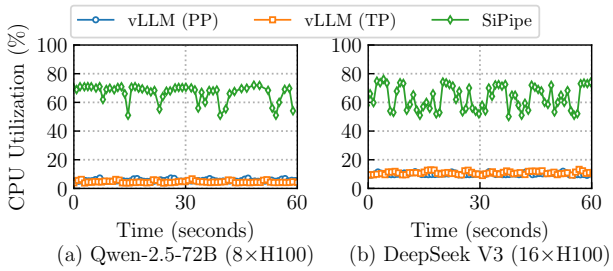


Figure 15. CPU utilization over time on H100-NVL testbed. vLLM (TP) and SiPipe adopt the default configuration, while vLLM (TP) using pure TP configuration.

approximately 60%, confirming that the CPU now contributes meaningfully across the inference process.

We also observe differences in CPU utilization patterns across LLMs. In Figure 15(a), SiPipe exhibits higher and more stable CPU usage compared to Figure 15(b). While the sampling workload per iteration is similar across LLMs, the fraction of time spent on *forward* varies significantly —*ultra-large* LLMs have longer *forward*. Since sampling tasks are executed after *forward*, faster *forward* leads to more frequent sampling tasks, driving higher and steadier CPU utilization. Conversely, the longer *forward* in *ultra-large* LLMs result in more bursty CPU utilization patterns.

7.5 Ablation Studies

Figure 16 presents the performance improvements contributed by each design component of SiPipe under the default configuration on the H100-NVL testbed. The results show that most gains come from CPU sampling (16%–38%) and TSEM (80% for Mixtral, as its fast *forward* makes input preparation more dominant, and 10%–24% for others), consistent with the per-iteration breakdown analysis in Section 3.1, which identifies *intra-stage bubbles* and *load-imbalance bubbles* as the main overheads in pipeline parallelism. Additionally, SAT contributes a 3%–6% improvement by eliminating stalls that arise when an upstream stage finishes its forward pass but must wait for the downstream stage to become ready to receive hidden states.

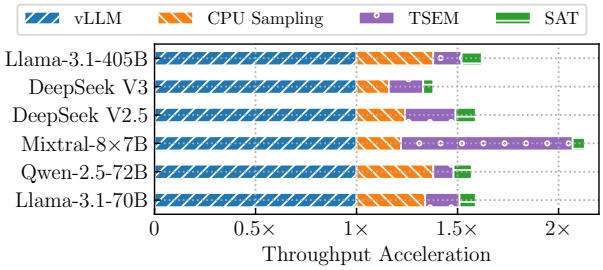


Figure 16. Incremental performance improvements contributed by each individual design.

8 Related Work

Besides the classic implementation of parallelism in open-source inference engines [27, 47], recent developments in PP for LLM inference reveal a dynamic shift from simple layer-based designs toward rich, multi-dimensional pipelined architectures that adapt to model structure and deployment scenarios. Classic vertical pipelining— partitioning consecutive layers across devices—has evolved through micro-batching into more nuanced systems that support sequence slicing (PipeLLM [42], HPipe [32]), patch-level parallelism for vision models (PipeFusion [45]), and token-level speculative execution (PipeInfer [12]), all tailored to minimize pipeline bubbles and boost utilization. Parallel efforts such as PISeL [39], PLANCK [29], Quart [28], and gLLM [20] underscore the importance of latency-awareness, cold-start avoidance, and SLO-aligned scheduling, signaling an industry-wide pivot toward dynamic, runtime-sensitive pipeline orchestration.

Furthermore, MoE models are catalyzing a new frontier of pipeline design by leveraging expert-disaggregated pipelining. Systems like MegaScale–Infer [49] and Pipeline MoE [13] decouple attention and FFN modules or experts, orchestrating ping-pong microbatches to overlap computation and communication efficiently and reflexively across heterogeneous GPUs—achieving significant throughput and cost gains. SiPipe can be compatible with these techniques as they parallel and accelerate the inference in different dimensions while meeting the same targets of strict latency and utilization.

9 Conclusion

This paper identifies key scalability bottlenecks in PP for LLM inference, including load-imbalance bubbles, intra-stage bubbles, and inter-stage bubbles. To overcome these challenges, we leverage the underutilized CPU resources on the host by offloading the sampling process to the CPU, introducing an asynchronous CPU executor, and designing a CPU-assisted communication module. These optimizations collectively yield up to 2× improvement in GPU inference throughput on large-scale LLMs. Our findings demonstrate that carefully orchestrating CPU-GPU collaboration is critical for unlocking the full potential of PP inference.

References

[1] [n. d.]. Llama 2. https://infohub.delltechnologies.com/ja-jp/l/llama-2-inferencing-on-a-single-gpu/introduction-3976/.

[2] 2024. Llama 2 follow-up: too much RLHF, GPU sizing, technical details. https://www.interconnects.ai/p/llama-2-part-2.

[3] 2025. AI Inference Market Size, Share and Trends Report. https://www.grandviewresearch.com/industry-analysis/artificial-intelligence-ai-inference-market-report.

[4] 2025. OpenAI API Documentation. https://platform.openai.com/docs.

[5] 2025. ShareGPT Datasets. https://huggingface.co/collections/bunnycore/sharegpt-datasets-66fa831dcee14c587f1e6d1c.

[6] 2025. vLLM–Optimization and Tuning. https://docs.vllm.ai/en/latest/configuration/optimization.html.

[7] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. 1985. A learning algorithm for Boltzmann machines. *Cognitive science* 9, 1 (1985), 147–169.

[8] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[9] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming Throughput-Latency Tradeoff in LLM Inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 117–134. https://www.usenix.org/conference/osdi24/presentation/agrawal

[10] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255* (2022).

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[12] Branden Butler, Sixing Yu, Arya Mazaheri, and Ali Jannesari. 2024. PipeInfer: Accelerating LLM Inference using Asynchronous Pipelined Speculation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE, 1–19. doi:10.1109/SC41406.2024.00046

[13] Xin Chen, Hengheng Zhang, Xiaotao Gu, Kaifeng Bi, Lingxi Xie, and Qi Tian. 2023. Pipeline MoE: A Flexible MoE Implementation with Pipeline Parallelism. *arXiv preprint arXiv:2304.11414* (April 2023).

[14] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[15] Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical neural story generation. *arXiv preprint arXiv:1805.04833* (2018).

[16] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-latency serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, 135–153.

[17] Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal* 12, 2 (1994), 23–38.

[18] Design Guide. 2020. Cuda c++ programming guide. *NVIDIA, July* (2020).

[19] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[20] Tianyu Guo, Xianwei Zhang, Jiangsu Du, Zhiguang Chen, Nong Xiao, and Yutong Lu. 2025. gLLM: Global Balanced Pipeline Parallelism System for Distributed LLM Serving with Token Throttling. *arXiv preprint arXiv:2504.14775* (April 2025).

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[22] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. A generic service to provide in-network aggregation for key-value streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 33–47.

[23] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751* (2019).

[24] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[25] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Eduardo Blanco and Wei Lu (Eds.). Association for Computational Linguistics, Brussels, Belgium, 66–71. doi:10.18653/v1/D18-2012

[26] Ilya Kulikov, Alexander H Miller, Kyunghyun Cho, and Jason Weston. 2018. Importance of a search strategy in neural dialogue modelling. *arXiv preprint arXiv:1811.00907* 2 (2018).

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[28] Yanying Lin, Yanbo Li, Shijie Peng, Yingfei Tang, Shutian Luo, Haiying Shen, Cheng-Zhong Xu, and Kejiang Ye. 2024. Quart: Latency-Aware FaaS System for Pipelining Large Model Inference. In *Proceedings of the 44th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 1–12. doi:10.1109/ICDCS.2024.00010

[29] Yanying Lin, Shijie Peng, Shuaipeng Wu, Yanbo Li, Chengzhi Lu, Chengzhong Xu, and Kejiang Ye. 2024. Planck: Optimizing LLM Inference Performance in Pipeline Parallelism with Fine-Grained SLO Constraint. In *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 1306–1313.

[30] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).

[31] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[32] Ruilong Ma, Xiang Yang, Jingyu Wang, Qi Qi, Haifeng Sun, Jing Wang, Zirui Zhuang, and Jianxin Liao. 2024. HPipe: Large Language Model Pipeline Parallelism for Long Context on Heterogeneous Cost-Effective Devices. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL) / Industry Track*. Association for Computational Linguistics, 1–9. doi:10.18653/v1/2024.naacl-industry.1

[33] Microsoft. 2024. DeepSpeed. https://github.com/microsoft/DeepSpeed.

[34] OpenAI. 2023. ChatGPT application. https://chat.openai.com/.

[35] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina

Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[36] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[37] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2022. The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink. arXiv:2204.05149 [cs.LG] https://arxiv.org/abs/2204.05149

[38] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation—A {KVCache-centric} Architecture for Serving {LLM} Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. 155–170.

[39] Maryam Rahimi Jafari et al. 2024. PISeL: Pipelining DNN Inference for Serverless Computing. In *Proceedings of the CIKM Conference*. ACM.

[40] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Katrin Erk and Noah A. Smith (Eds.). Association for Computational Linguistics, Berlin, Germany, 1715–1725. doi:10.18653/v1/P16-1162

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[42] Yifan Tan, Cheng Tan, Zeyu Mi, and Haibo Chen. 2025. Pipellm: Fast and confidential large language model services with speculative pipelined encryption. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 843–857.

[43] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[45] Jiannan Wang, Jinyang Fang, Aoyu Li, and Pengcheng Yang. 2024. PipeFusion: Displaced Patch Pipeline Parallelism for Inference of Diffusion Transformer Models. *arXiv preprint arXiv:2405.14430* (May 2024).

[46] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[47] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104* (2024).

[48] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 193–210. https://www.usenix.org/conference/osdi24/presentation/zhong-yinmin

[49] Ruidong Zhu, Ziheng Jiang, Chao Jin, Peng Wu, Cesar A. Stuardo, Dongyang Wang, Xinlei Zhang, Huaping Zhou, Haoran Wei, Yang Cheng, Jianzhe Xiao, Xinyi Zhang, Lingjun Liu, Haibin Lin, Li-Wen Chang, Jianxi Ye, Xiao Yu, Xuanzhe Liu, Xin Jin, and Xin Liu. 2025. MegaScale-Infer: Serving Mixture-of-Experts at Scale with Disaggregated Expert Parallelism. *arXiv preprint arXiv:2504.02263* (April 2025).

Table 1. Notations and their meanings.

Notation	Meaning
L	Number of transformer layers
C	Per-layer computation time of models
N	Total number of GPUs
p	Pipeline parallelism degree
t	Tensor parallelism degree
α	Launch delay of communication
s	Sequence length
b	Batch size
h	Hidden layer size
B_1	Intra-node bandwidth
B_2	Inter-node bandwidth
m	Number of microbatches
n	Number of hosts

A Performance Modeling

This section derives analytic expressions for throughput and per-token delay in model-parallel LLM inference deployments. All symbols are defined in Table 1. We first analyze a single-node configuration, where a single inference engine is partitioned across multiple GPUs using either tensor parallelism (TP) or pipeline parallelism (PP), and then extend the LLM to cross-node settings.

A.1 Pure Tensor Parallelism

Each transformer layer performs two *allreduce* communication operations under TP. Including start-up latency, the communication cost on N GPUs is

$$t_{\text{comm-TP}}(N) = 2L \times (\alpha \log_2 N + 2sbh/B_1). \tag{1}$$

Therefore, the overall throughput of the inference system is:

$$\begin{aligned} T(N) &= \frac{b}{t_{\text{comp-TP}}(N) + t_{\text{comm-TP}}(N)} \\ &= \frac{b}{\frac{LC}{N} + 2L\alpha \log_2 N + 4sbhL/B_1}. \end{aligned} \tag{2}$$

Correspondingly, the per-token delay—the reciprocal of throughput—is:

$$D(N) = \frac{LC}{N} + 2L\alpha \log_2 N + 4sbhL/B_1. \tag{3}$$

A.2 Pure Pipeline Parallelism

Pipeline parallelism splits a batch of b sequences into m microbatches , so each stage produces b/m tokens per iteration. For one stage the latency is:

$$t_{\text{stage}} = \frac{LC}{N} + sbh/B_1. \tag{4}$$

Hence overall throughput is:

$$T(N) = \frac{b/m}{t_{\text{stage}}} = \frac{b/m}{\frac{LC}{N} + sbh/B_1}. \tag{5}$$

A single token experiences N compute phases and $N-1$ inter-stage transfers, so its end-to-end delay is

$$D(N) = N \times \frac{LC}{N} + (N-1) \times sbh/B_1. \tag{6}$$

A.3 Hybrid Model Parallelism

In production, tensor and pipeline parallelism are often enabled together to fit larger models, so the system incurs communication overheads from each. Let t be the tensor-parallel degree and p the pipeline-parallel degree ($p \times t = N$). Combining the two previous models yields:

$$\begin{aligned} T(p,t) &= \frac{b/m}{\frac{LC}{N} + sbh/B_1 + \frac{2L}{p}(\alpha \log_2 t + 2sbh/B_1)}; \\ D(p,t) &= p\left(\frac{LC}{N} + \frac{2L}{p}(\alpha \log_2 t + 2sbh/B_1)\right) \\ &\quad + (p-1)sbh/B_1. \end{aligned} \tag{7}$$

A.4 Multi-node Model Parallelism

When an inference instance spans multiple nodes, part of the communication occurs over a slower inter-node network with bandwidth B_2. Assuming that n hosts are connected through this network, then the performance of pure tensor parallelism is:

$$\begin{aligned} T(N) &= \frac{b}{\frac{LC}{N} + 2L(\alpha \log_2 N + 2sbh/B_2)}, \\ D(N) &= \frac{LC}{N} + 2L(\alpha \log_2 N + 2sbh/B_2). \end{aligned} \tag{8}$$

For pure pipeline parallelism, only the $n-1$ inter-stage transfers that cross hosts use bandwidth B_2; the remaining $N-n$ transfers stay on the faster intra-node link B_1:

$$\begin{aligned} T(N) &= \frac{b/m}{\frac{LC}{N} + sbh/B_2}, \\ D(N) &= LC + (n-1)sbh/B_2 + (N-n)sbh/B_1. \end{aligned} \tag{9}$$

Put them together, the combined throughput and per-token delay are:

$$\begin{aligned} T(p,t) &= \frac{b/m}{\frac{LC}{N} + sbh/B_2 + 2L(\alpha \log_2 N + 2sbh/B_2)}, \\ D(p,t) &= LC + (n-1)sbh/B_2 + (N-n)sbh/B_1 \\ &\quad + 2L(\alpha \log_2 N + 2sbh/B_2). \end{aligned} \tag{10}$$

From the models above we note two insights:

(1) **Throughput scaling.** Pipeline parallelism scales almost linearly with the number of GPUs, whereas tensor parallelism hits a throughput ceiling: although computation is divided further, the communication volume stays constant (or even grows), so GPU utilization plateaus.

(2) **Latency trends.** Increasing the tensor-parallel degree quickly lowers per-token latency, while deeper pipeline parallelism lengthens it because each token must traverse more stages.