gLLM: Global Balanced Pipeline Parallelism System for Distributed LLM Serving with Token Throttling

Tianyu Guo
guoty9@mail2.sysu.edu.cn
Sun Yat-sen University
Guangzhou, China

Xianwei Zhang[#]
zhangxw79@mail.sysu.edu.cn
Sun Yat-sen University
Guangzhou, China

Jiangsu Du
dujiangsu@mail.sysu.edu.cn
Sun Yat-sen University
Guangzhou, China

Zhiguang Chen
chenzhg29@mail.sysu.edu.cn
Sun Yat-sen University
Guangzhou, China

Nong Xiao
xiaon6@mail.sysu.edu.cn
Sun Yat-sen University
Guangzhou, China

Yutong Lu
luyutong@mail.sysu.edu.cn
Sun Yat-sen University
Guangzhou, China

Abstract

Pipeline parallelism has emerged as a predominant approach for deploying large language models (LLMs) across distributed nodes, owing to its lower communication overhead compared to tensor parallelism. While demonstrating high throughput in request serving, pipeline parallelism often suffers from performance limitations caused by pipeline bubbles, which are primarily resulted from imbalanced computation delays across batches. Existing methods like Sarathi-Serve attempt to address this through hybrid scheduling of chunked prefill and decode tokens using a fixed token budget. However, such methods may experience significant fluctuations due to either insufficient prefill tokens or uneven distribution of decode tokens, ultimately leading to computational imbalance. To overcome these inefficiencies, we present gLLM, a globally balanced pipeline parallelism system incorporating Token Throttling to effectively mitigate the pipeline bubbles. Our Token Throttling mechanism is a fine-grained scheduling policy that independently regulates the quantities of prefill and decode tokens, thus enabling balanced computation by leveraging global information from the inference system. Specifically, for decode tokens, gLLM maintains near-consistent token count across processing batches. For prefill tokens, it dynamically adjusts batch sizes based on both total pending tokens and the memory utilization rates of key-value cache (KV cache). Furthermore, gLLM runtime adopts an asynchronous execution and message passing architecture specifically optimized for pipeline parallelism characteristics. Experimental evaluations with representative LLMs show that gLLM achieves significant performance improvements, delivering 11% to 398% higher maximum throughput compared to state-of-the-art pipeline or tensor parallelism systems, while simultaneously maintaining lower latency.

1 INTRODUCTION

Large language models (LLMs) have demonstrated remarkable capabilities in performing complex tasks like logical reasoning [1–3], mathematical problem solving [4–9] and agent acting [10–15]. As model parameters scale to hundreds of billion or even trillions [16–21], distributed serving of LLMs [22–25] has become essential due to GPU memory constraints. Among distributed approaches, pipeline parallelism has emerged as a predominant method for training and serving models like deep nerual networks (DNN) [26–28] and LLMs
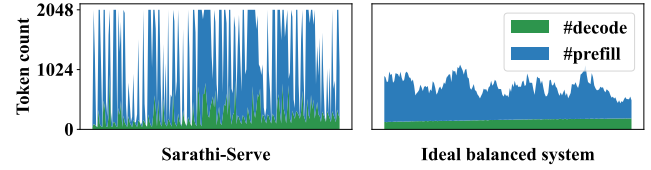

[#]Corresponding author.

Figure 1: A comparison of the scheduled token counts of prefill and decode stage between Sarathi-Serve and an ideal balanced system per iteration (horizontal axis), with the maximum batched token count (token budget) set to 2048 for both systems.

[29–38], primarily due to its low communication overhead that effectively mitigates bandwidth limitations. However, this method often incurs pipeline bubbles, i.e., periods of GPU inactivity where subsequent pipeline stages must wait for prior ones to complete micro-batch processing. These inefficient bubbles mainly stem from computational imbalance between micro-batches. While existing researches have focused on optimizing pipeline bubbles in training scenarios [29–36], recent methods like Sarathi-Serve [36] and prefill-decode disaggregated architectures [37, 38] aim to address computational imbalances specifically between prefill and decode stages. Our analysis reveals that these solutions remain insufficient for effectively resolving the bubble issues.

Each LLM serving request undergoes two distinct stages. The first stage, prefill, processes the prompt tokens to compute keys and values (KV cache), then generating the initial output token. The second stage, decode, produces the remaining output tokens. The two stages exhibit markedly different computational characteristics, where prefill operations typically saturate GPU capacity, while decode operations demonstrate significantly lower compute utilization. Therefore, to enhance hardware efficiency, multiple decode tokens are commonly batched together for parallel processing. However, the interleaved execution of prefill and decode operations may create mutual interference. To mitigate the unbalanced batching policy, Sarathi-Serve [36] proposes to apply a hybrid scheduling that allocates a fixed token budget between chunked prefill and decode operations. However, this solution fails to properly account for the critical prefill-to-decode ratio. In practice, decode tokens often lack sufficient corresponding prefill tokens for

effective co-scheduling. Figure 1 illustrates the disparity in batched token counts per iteration between Sarathi-Serve and an optimally balanced system. It can be observed that Sarathi-Serve's scheduling results in substantially greater token count volatility compared to the balanced counterpart. These fluctuations can be attributed to two primary factors: (1) missed opportunities for batching decode tokens with prefill tokens, and (2) uneven distribution of decode tokens across batches. Unfortunately, they frequently induce pipeline bubbles that significantly degrade system performance. Whereas reducing token budget could theoretically smooth these fluctuations, such approach would disproportionately penalize prefill rates, ultimately constraining the overall system throughput.

To address the divergent computational demands between prefill and decode stages, prefill-decode disaggregated architectures [37, 38] have been recently proposed to alleviate prefill-decode interference and achieve low cost budget for each stage respectively. Generally, the designs assign prefill and decode computation to different nodes connected via KV cache transmission. Despite separate computation of the prefill and decode stages, computational imbalance persists across either prefill batches or decode batches. Moreover, determining the optimal ratio of GPUs allocated to the prefill stage versus the decode stage becomes challenging under dynamically fluctuating request rates. The tight coupling between prefill and decode nodes also raises fault tolerance concerns. A failure in one stage could cascade to the other.

To effectively address the unbalanced problems and minimize pipeline bubbles, we design gLLM, a globally efficient pipeline parallelism system that balances computation across micro-batches using Token Throttling. This mechanism intelligently regulates the number of tokens processed in the prefill and decode stage based on real-time system states. For decode tokens, it distributes tokens evenly across micro-batches. For prefill tokens, it balances batch tokens based on both the pending prefill tokens and KV cache utilization rates. By dynamically throttling tokens count, gLLM achieves better load balancing across micro-batches and greatly alleviates the pipeline bubble issue. Moreover, considering the characteristic of pipeline parallelism, gLLM runtime adopts an asynchronous execution and message passing architecture for reducing data dependency and CPU overhead.

The contributions of this paper are:

- We highlight the observations that pipeline bubbles caused by unbalanced computation in prefill and decode stage significantly degrade the performance of inference systems.
- We present gLLM, a distributed serving system with Token Throttling for effectively balancing the computation across batches to reduce pipeline bubbles.
- Token Throttling dynamically adjusts the batch size of the prefill and decode stage separately to achieve balanced schedule based on real-time inference system feedback.
- Experiments on representative LLMs show that gLLM enhances maximum throughput by 11% to 398% compared to state-of-the-art pipeline or tensor parallelism systems, while simultaneously achieving lower latency.

2 BACKGROUND AND MOTIVATION
2.1 The Transformer Architecture
The predominant architecture in contemporary LLMs adopts the decoder-only transformer [39]. In which, the input begins with an embedding layer, which converts token IDs into hidden states while incorporating positional encoding to keep sequence order. These hidden states then pass through multiple decoder layers, each consisting of a self-attention mechanism (with causal masking to ensure autoregressive properties), layer normalization, and a multi-layer perceptron (MLP). Between them, self-attention is the core of the transformer, enabling LLMs to capture long-range contextual dependencies. After processing through the decoder layers, a linear projection layer transforms the final hidden states into logits, representing the probability distribution over the vocabulary. Finally, a sampling strategy (e.g., greedy, top-k, or nucleus sampling) selects the next token for generation.

2.2 LLM Inference Procedure
Autoregressive Decoding. The procedure of LLM inference is autoregressive where each token is generated based on all preceding onces through the computation of attention scores between their keys and values. To avoid the recomputation of keys and values, they are retained for subsequent steps [40, 41]. Base on that, LLM inference can be divided into two distinct phases: 1. Prefill: The prompt's tokens are processed in parallel to populate the KV cache and generate the first output token. This phase fully utilizes GPU compute resources due to high parallelism. 2. Decode: Each new token is generated sequentially by attending to the last token and the stored KV cache. This phase often underutilizes the GPU, as it processes only one token per step and incurs frequent memory bandwidth bottlenecks from KV cache accesses. The unbalanced computation characteristic between prefill and decode stage may impose inefficiency in LLM inference [36–38].

Scheduling Policies. Traditional inference engine like Faster-Transformer [42] employs batch-level scheduling which selects a group of requests and executes it until the completion of all the sequences. Without considering variable length characteristic of transformer architecture, this method delays early-finished and late-joining requests. Orca [40] overcomes this issue by proposing iteration-level scheduling, which allows requests to dynamically enter or exit a batch before each model forward. Whereas Orca can batch requests from both prefill and decode stages, it introduces generation stalls for ongoing decode requests due to high prefill computation latency. To solve the problem, Sarathi-Serve [36] allows computing large prefills in small chunks across several iterations and hybrid scheduling of chunked prefill and decode tokens to achieve stall-free batching. Specifically, Sarathi-Serve first schedules all decode tokens, then maximizes chunked prefill tokens within the fixed token budget. However, in actual situation, decode tokens often lack the opportunity to mix with adequate prefill tokens.

2.3 Parallelism Strategies of LLM Inference
The basic parallelism strategies in LLM inference primarily consist of data parallelism and model parallelism (as shown in Figure 2).
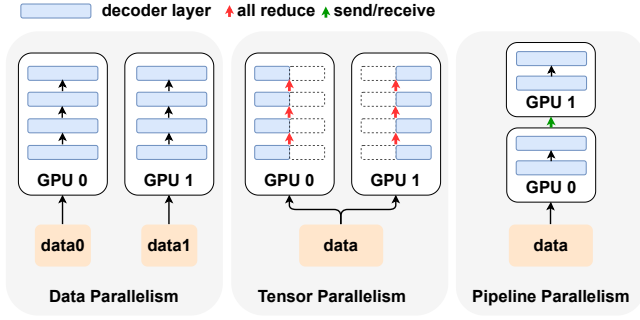
Figure 2: Comparison of data parallelism, tensor parallelism and pipeline parallelism.

Data parallelism splits the input data into multiple parts, sending each part to the corresponding GPU for parallel processing. Model parallelism partitions the model itself, with each GPU responsible for a different portion of the model. Model parallelism can be further classified into tensor parallelism and pipeline parallelism, depending on how the model is partitioned. Tensor parallelism employs intra-layer parallelism, splitting individual operations across GPUs and requiring frequent communication to synchronize results. Pipeline parallelism adopts inter-layer parallelism, assigning different layers to different GPUs and only requiring communication to pass intermediate activations between stages. In addition, pipeline parallelism employs multiple micro-batches to saturate GPUs at different pipeline stages. Pipeline depth refers to the number of sequential stages in a pipeline, where each stage performs a specific part of a task. Due to these differences, tensor parallelism can reduce forward latency at the cost of higher communication overhead than pipeline parallelism. In online serving scenarios, tensor parallelism is more suitable for low request rates, while pipeline parallelism better handles high-throughput demands.

2.4 Challenges in Pipeline Parallelism

Pipeline Bubbles. Pipeline Parallelism reduces the need for high-bandwidth communication but may incur load-balancing issues and low GPU utilization. These issues manifest as pipeline bubbles, periods of GPU idle time caused by two types of dependencies: (1) inter-stage dependency, where a stage cannot begin computation until the preceding stage completes, and (2) inter-batch dependency, where the number of concurrent micro-batches is limited by the pipeline depth. The load imbalance stems from: (1) inter-stage imbalance due to uneven computation distribution across pipeline stages, and (2) inter-batch imbalance caused by variation in computation requirements across different micro-batches. Figure 3 illustrates how these imbalances create pipeline bubbles. In this paper, we focus on solving inter-batch pipeline bubbles, while the inter-stage bubbles are left for future works.

Insufficient GPU Utilization. Pipeline parallelism has long been suffered from low GPU utilization. Although Sarathi-Serve attempts to mitigate this issue, under-utilization persists in current systems. As shown in Figure 4 when serving a 32B model with 4 GPUs, their utilization follows a two-stage pattern: an initial phase with high fluctuations followed by a stable but suboptimal phase. By
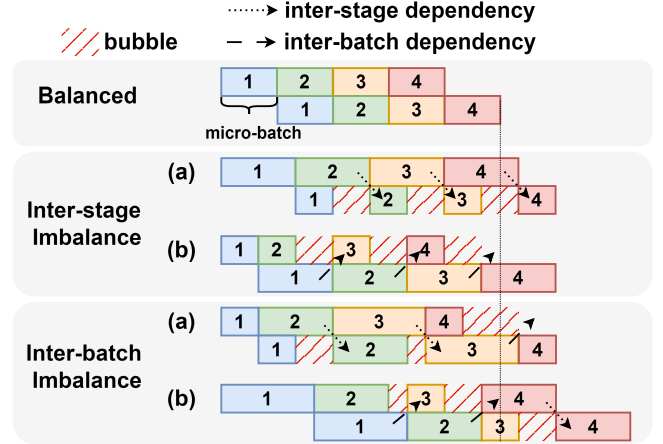

Figure 3: Pipeline bubbles caused by two types of imbalance, i.e., inter-stage and inter-batch. (the number indicates the ordinal position of the micro-batch)

(a) Variation of GPU utilization
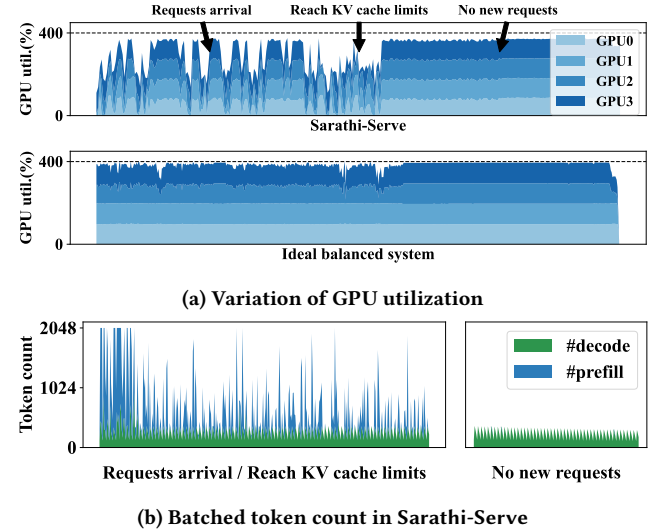

(b) Batched token count in Sarathi-Serve

Figure 4: Under-utilized GPU caused by unbalanced scheduling.

correlating these observations with request timing, we see that the initial stage coincides with incoming requests, forcing the system to handle both prefill and decode tokens. Once no new requests arrive, the system shifts to decoding only, leading to steadier but still inefficient utilization. Notably, batched token counts (Figure 4b) fluctuate throughout execution, particularly upon requests arrival or reaching KV cache limits. These irregular batch sizes introduce severe pipeline bubbles, which directly contribute to poor GPU utilization in both stages.

2.5 Scheduling Demands

Balanced Scheduling. Pipeline parallelism relies on the balanced computation across micro-batches, but current scheduling strategy
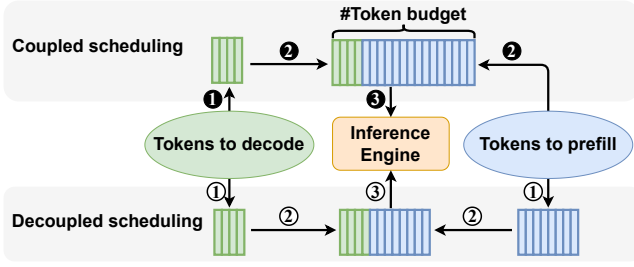
Figure 5: Comparison between coupled and decoupled scheduling.

Figure 6: Overview of gLLM system. Inference procedure in gLLM consists of tokens flowing through its components.

fails to meet this requirement: (1) Prefill Imbalance. The fluctuations of batched prefill tokens count within the current scheduling strategy primarily stems from two key factors. First, prefill tokens depend on waiting requests. When no requests are available to prefill, the batched token count fluctuates. Second, prefill operations are constrained by KV cache utilization. If insufficient space exists to store the computed KV cache, the system halts the scheduling of prefill tokens. However, current scheduling method fails to account for these factors, resulting in unbalanced prefill scheduling. (2) Decode Imbalance. To achieve balanced processing during the decode stage, we aim to distribute the total decode requests as evenly as possible across all available micro-batches. While current scheduling strategy lacks explicit balancing considerations and may induce uneven workload distribution across micro-batches.

Decoupled Scheduling. As shown in Figure 5, current method first schedules all decode tokens ❶, then maximizes chunked prefill tokens within the fixed token budget ❷. However, we should separately schedule balanced count of prefill and decode tokens ①, rather than limiting their total to a fixed value. This is because the scheduled tokens count may not reach the maximum budget and often causes the unbalanced schedule. Besides, the optimal batch tokens count tends to vary dynamically. Moreover, the tight coupling between prefill and decode scheduling often leads to interference between these stages. For instance, when numerous decode requests are in progress, the available token capacity for prefill becomes limited. Nevertheless, maximizing inference throughput requires processing large batches of prefill tokens. This fundamental conflict reveals that tightly coupling prefill and decode scheduling under a fixed total token budget cannot effectively satisfy their respective requirements. Thus there is a critical need to develop decoupled scheduling mechanisms.

Dynamically Scheduling. The number of prefill tokens per batch should adapt dynamically to the inference system's state. For example, at low KV cache utilization (i.e., when few requests are in decode), we should increase prefill speed to maximize GPU utilization. Conversely, during periods of high KV cache utilization, we are expected to reduce the prefill rate to prevent preemption of sequences for insufficient KV cache. Another concern is that if there are few tokens to prefill, we should prefill smoothly to avoid sudden fluctuation in batched tokens count. In the contrast, if there are abundant tokens to prefill, we are expected to maintain high prefill rate. However, current prefill batching is constrained by the fixed token budget and the number of active decode requests,
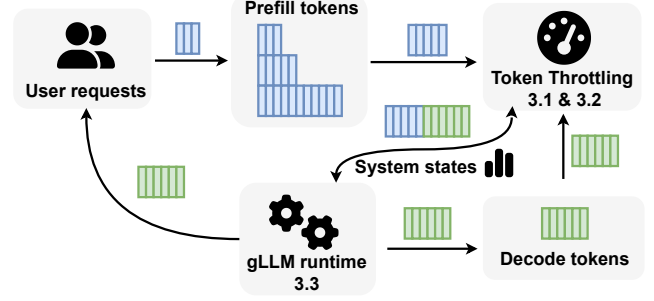
rather than adapting to actual demand. This inflexibility leads to suboptimal scheduling policy, highlighting the need for a more adaptive scheduling approach.

3 DESIGN

In this section, we present gLLM, a global balanced pipeline parallelism systems with Token Throttling. As shown in Figure 6, gLLM dynamically schedules prefill and decode tokens separately by throttling batched token counts according to real-time inference system states. After scheduling, gLLM merges scheduled prefill and decode tokens into a single batch. In addition, gLLM runtime adopts asynchronous architectures, hiding CPU operations overhead.

3.1 Prefill Token Throttling

The prefill operation is the first step in LLM inference, where the prompt's KV cache is computed and the first output token is generated. Therefore, prefill scheduling depends primarily on two factors: the number of tokens waiting for prefill and the system's KV cache utilization. First, if no requests are pending prefill, the operation cannot be scheduled. Second, if the KV cache is near capacity, prefill cannot proceed due to insufficient memory. Beyond these constraints, the scale of these factors also influences scheduling decisions. When few requests are waiting or KV cache usage is high, the system should reduce the prefill rate to avoid pipeline bubbles or requests preemption. Conversely, when many requests are queued or KV cache availability is high, the system should increase the prefill rate to maximize throughput. To balance these factors, we throttle the prefill token count based on both the volume of pending tokens and current KV cache pressure.

3.1.1 Throttling by Tokens Count Awaiting Prefill (WT). During each schedule, gLLM collects the number of tokens awaiting prefill ($\#WP$) to determine the batched prefill token count ($\#P$). This decision relies on three hyperparameters, minimum/maximum batched token count of prefill ($\#MinP$/$\#MaxP$) and the number of iterations ($\#T$) required to process all tokens waiting for prefill. The batched prefill token count can be calculated as follows:

$$\#P = min(max(\frac{\#WP}{\#T}, \#MinP), \#MaxP) \quad (1)$$

3.1.2 Throttling by KV Cache Utilization Rate (UT). At the beginning of each schedule, gLLM also collects the KV cache idle rate
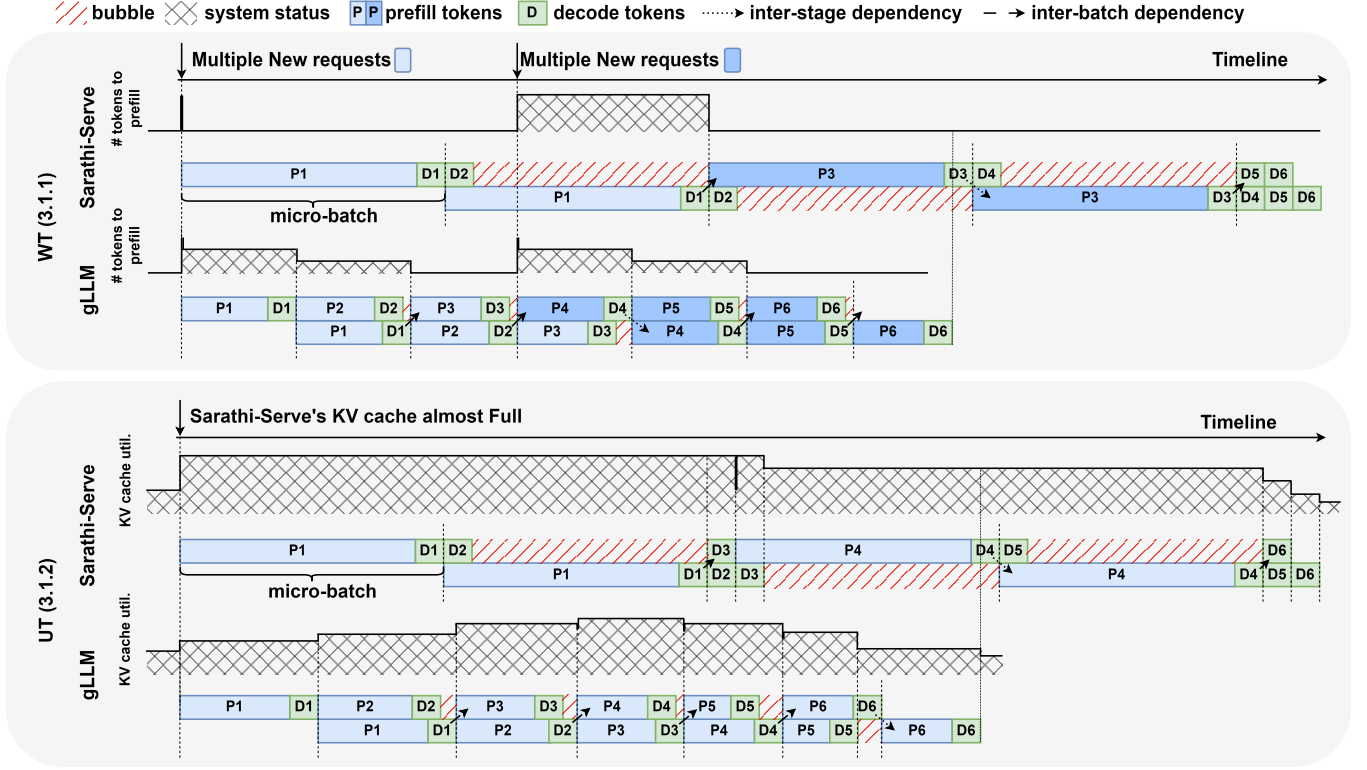
Figure 7: Case study (the pipeline depth is 2) of prefill Token Throttling. The number indicates the ordinal position of the micro-batch. For the second situation, the KV cache is allocated for prefill tokens prior to the execution of each micro-batch (at the begin of first stage). After processing a micro-batch (at the end of last stage), the KV cache usage may increase, decrease or stay roughly the same depending on how many requests meet the termination criterion. The KV cache usage is consistent across all GPUs since they share unified page tables.

$(KV_{free} \in [0, 1])$ to determine batched token count (#P). This decision depends on two hyperparameters, minimum/maximum tokens count batched in prefill (#MinP/#MaxP). The batched token count is then computed as:

$$\#P = max(\#MaxP \times KV_{free}, \#MinP) \qquad (2)$$

3.1.3 Threshold. Besides dynamically adjusting prefill tokens count, we also introduce a KV cache idle threshold ($KV_{thresh} \in [0, 1]$) to regulate scheduling decisions. When current KV cache idle rate is less than KV_{thresh}, the system automatically suspends prefill token processing to prevent KV cache overflow. This safeguard mechanism addresses the critical issues observed in unrestricted prefill operations: Premature preemption of ongoing decode requests causes costly recomputation time. By maintaining buffer headroom through the threshold mechanism, we ensure adequate resource allocation for active decode requests while maintaining stable system.

Combining aforementioned factors (WT, UT and threshold), we compute the batched token count as:

$$\#P = max(min(\frac{\#WP}{\#T}, \#MaxP \times \frac{KV_{free} - KV_{thresh}}{1 - KV_{thresh}}), \#MinP) \quad (3)$$

3.1.4 Case Study. Figure 7 compares the prefill scheduling strategies of Sarathi-Serve and gLLM. In the first scenario, there are no tokens waiting to prefill at the initial state and #T is set to 3. Upon requests arrival, Sarathi-Serve eagerly processes prefill tokens. This leads to no prefill tokens batched in the following batch, resulting in large fluctuation in the computation. The significant pipeline bubbles also prevent Sarathi-Serve from processing the next batch of requests in a timely manner. Instead, gLLM evenly distributes new requests across the following batches, achieving better load balance.

For the second situation, there are sufficient tokens waiting to prefill at the start. Sarathi-Serve schedules prefill tokens without considering KV cache utilization rates. This results in the remaining KV cache space being insufficient to support prefill computation after allocating the room to the prefill tokens. And the following batch can only handle decode tokens until some requests finish. In the contrast, gLLM dynamically adjust prefill scheduling based on KV cache occupancy, ensuring more balanced computation. Moreover, gLLM can achieve more efficient KV cache turnover, enabling timely processing of decode requests and preventing their KV cache from occupying space for extended periods.
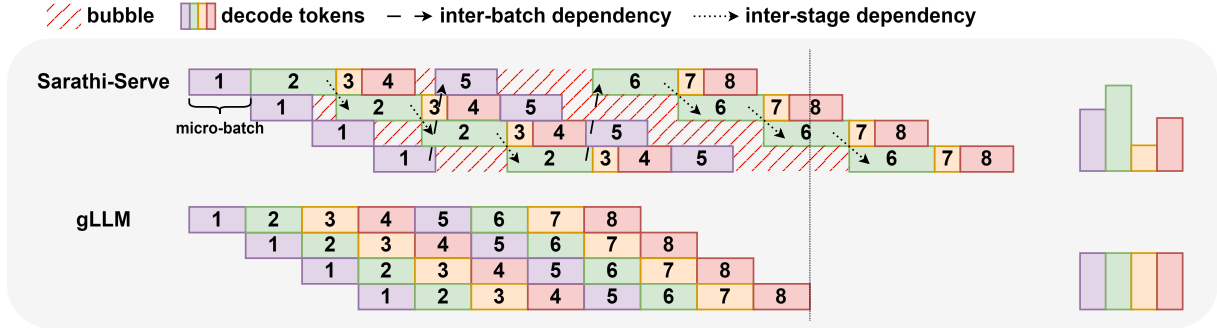
Figure 8: Case study (the pipeline depth is 4) of decode Token Throttling. The number indicates the ordinal position of the micro-batch.

3.2 Decode Token Throttling

3.2.1 Throttling by Tokens Count Under Decode. The scheduling for decode Token Throttling is straightforward, since decode operations require multiple iterations (equal to the output sequence length) while prefill operations typically complete in a single iteration. The variation in decode requests is relatively small, as these requests either originate from completed prefill phase or have reached their termination condition. Therefore, our objective is to distribute the total decode tokens evenly across all available micro-batches. Given that the maximal number of active micro-batches equals the pipeline depth ($\#PP_{depth}$), we can compute the batched decode tokens count ($\#D$) as:

$$\#D = \frac{\#RD}{\#PP_{depth}} \quad (4)$$

where $\#RD$ is total running tokens count under decode. If the remaining decode tokens are fewer than $\#D$, we schedule all of them; otherwise, we schedule exactly $\#D$ tokens.

3.2.2 Case Study. Figure 8 compares the decode scheduling strategies of Sarathi-Serve and gLLM. Sarathi-Serve fails to balance workloads evenly across micro-batches, leading to significant pipeline bubbles. For inter-stage dependency, the second/sixth micro-batch has to wait for the completion of the previous stage. For inter-batch dependency, the fifth/sixth micro-batch has to wait for the finish of the first/second micro-batch. These dependencies induce significant pipeline bubbles. In contrast, gLLM optimizes scheduling by dynamically adjusting token counts based on the total tokens under decode, ensuring a balanced workload distribution and reducing pipeline inefficiencies.

3.3 gLLM Runtime

Compared to tensor parallelism, pipeline parallelism involves more sophisticated control flow to orchestrate micro-batches across different pipeline stages. To enable Token Throttling, we design an asynchronous runtime with its architecture shown in Figure 9.

To support concurrent execution of multiple stages, gLLM runtime adopts a multi-process architecture where each pipeline stage is assigned a dedicated worker process, along with a separate front-end process for user interaction. The workers are divided into two roles: a driver worker and ordinary workers. The driver worker oversees other ordinary workers, handling tasks such as receiving
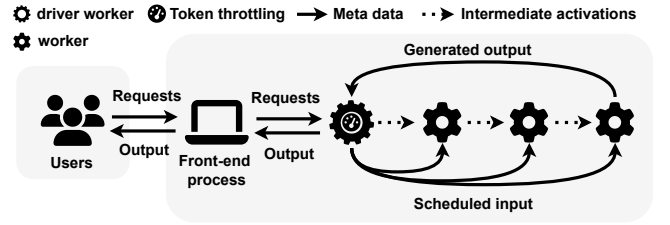

Figure 9: Architecture of gLLM runtime.

new requests from the front-end, scheduling micro-batches, broadcasting metadata for each schedule and streaming output back to the front-end. Meanwhile, all the workers focus on model execution, receiving activations from the previous stage, performing forward computations, and sending activations to the next stage. The driver worker is responsible for the KV cache management and all the workers share the page tables like vLLM.

The asynchronous of gLLM runtime is achieved through three coordinated design principles:

(1) Non-blocking pipeline operations. All workers processes employ non-blocking mechanisms for core operations including request reception, metadata exchange, and activation transmission, forming a continuous processing pipeline that eliminates idle waiting between computational stages.

(2) Decoupled frontend-backend processing. We design a dedicated frontend process that handles user-facing operations (request intake and response streaming), enabling full parallelism with backend model execution on worker processes. This separation allows continuous user interaction while maintaining uninterrupted model computation.

(3) Preemptive metadata scheduling. The runtime utilizes a dual-phase data transmission[1] strategy: (a) Driver workers broadcast metadata packets to all workers (b) Workers receive corresponding intermediate data streams. This decoupling enables workers to perform critical path preparation (input or attention metadata tensor creation) using early-arrived metadata, effectively overlapping data preparation latency with active computation cycles. The proactive

---

[1]In gLLM, metadata is transmitted via ZeroMQ, while intermediate activations are exchanged using NCCL.

scheduling mechanism ensures computation resources remain fully utilized throughout the execution timeline.

3.4 Implementation

Upon implementation, we identify a critical design limitation in vLLM's pipeline parallelism architecture: it tightly couples the transmission of intermediate activations with input scheduling metadata. This integration introduces significant CPU overhead during input preparation for model forwarding, accounting for approximately 17% of the total execution time and substantially diminishing the potential benefits of pipeline parallelism.

To overcome this limitation, we have implemented essential part of gLLM with 4K lines of Python code on top of CUDA ecosystem. The system features a RESTful API frontend and offers core OpenAI-compatible APIs. It incorporates vLLM's manually optimized CUDA kernels for key operations including activation functions, layer normalization, position encoding, and KV cache management, while also adopting flash attention [43–45]. We also integrates several recent advanced LLM optimizations, including iteration-level scheduling [40], PagedAttention [41], chunked prefill [36] and prefix caching [46]. The system supports leading open-source LLMs like Llama [17], Qwen [47] and ChatGLM [48] with varying sizes. To validate its output quality, we evaluate gLLM on the MMLU-pro benchmark [49], with detailed results presented in Section 4.7.

4 EVALUATION

4.1 Experimental Setup

Models and Environment. We evaluate gLLM using the Qwen2.5 series [47] (14B and 32B parameter variants) and Llama-3.1-100B[2] [17] for their strong multi-task performance. All models utilize bfloat16 data type. Our experiments employ three node configurations: (1) 4× NVIDIA L20-48GB GPUs (2) 4× NVIDIA A100-40GB GPUs (3) 4× NVIDIA A800-80GB GPUs. All GPUs are connected via PCIe across three configurations. We primarily evaluated two types of scenarios: intra-node deployment and cross-node deployment. For intra-node experiments, we use the L20 configuration. Cross-node evaluations leverage the A100 and A800 setup with simulated network conditions achieved by disabling both P2P communication (PCIe-based) and shared memory access (by setting environment variable $NCCL\_SHM\_DISABLE = 1$ and $NCCL\_P2P\_DISABLE = 1$). This configuration forces all inter-GPU communication through the network stack. Testing results show that the simulated network communication bandwidth achieves 73.28 Gbps, whereas the PCIe-based communication bandwidth attains 20.79 GB/s.

Workloads. We synthesize workloads based on ShareGPT [50] and Azure [37], which both comes from real LLM services. The ShareGPT dataset is a collection of user-shared conversations with ChatGPT. The Azure dataset is from production traces taken from Azure LLM inference services, including the arrival time, input size and output size. Figure 11 displays the distribution of input and output lengths across the sampled datasets, revealing that the Azure dataset has a notably longer average input (5.21×) and output (1.66×) length compared to ShareGPT. We mimic the cloud

service scenario and generate request arrival times using Poisson distribution with different request rates.

Schemes. To evaluate the effectiveness of our proposed design, we compare gLLM against the following systems[3].

- **vLLM** [41]. We leverage vLLM (v0.8.1) as the fastest inference engines for pipeline parallelism. The framework offers two backend engine versions, V0 and V1. While V1 demonstrates superior speed to V0, our testing revealed it to be less stable. As a result, we employ V1 as the default choice but fall back to V0 when encountering engine crashes.
- **SGLang** [46]. We leverage SGLang (v0.4.3.post2) as the most efficient inference engine for tensor parallelism implementations. While SGLang has lower CPU overhead than vLLM, it currently lacks pipeline parallelism support.
- **gLLM**. Proposed global balanced pipeline parallelism systems with Token Throttling.
- **gLLM w/o WT**. gLLM without WT (section 3.1.1).
- **gLLM w/o UT**. gLLM without UT (section 3.1.2).
- **gLLM w/ CK**. gLLM with the scheduling policy used in Sarathi-Serve.

vLLM and SGLang both follows Sarathi-Serve's scheduling strategy (token budget is set to 2048). To ensure fairness comparisons, we disable KV cache reuse across requests and cuda graph for each system. While vLLM and gLLM employ pipeline parallelism, SGLang utilizes tensor parallelism. The GPU memory utilization of each system is set to the maximum without encountering out of memory error. In the main experiment, we benchmark gLLM against vLLM and SGLang, while the ablation study compares gLLM with its variations. For gLLM, we set the hyperparameters as follows: $\#T = 8$, $\#MaxP = 2048$, $\#MinP = 32$ and $KV_{thresh} = 0.05$. We examine the impact of these parameter settings in the sensitivity study.

Metrics. We use the following metrics to measure the performance of the systems.

- **Time to first token (TTFT)**: Average time taken from when a user sends a prompt to the LLM until the first token of the response is generated.
- **Time per output token (TPOT)**: Average time required to generate each subsequent token after the first one.
- **End to end latency (E2EL)**: Average total time from prompt submission to the completion of the full response.
- **Throughput**: Average input and output tokens processing throughput.
- **SLO Attainment**: The SLO fulfillment rate under the given TTFT and TPOT constraints.

4.2 Latency and Throughput Evaluation

To compare the performance of vLLM, SGLang and gLLM, we conduct experiments based on ShareGPT and Azure dataset using models from 14B to 100B with the results illustrated on Figure 10[4] (intra-node) and Figure 13 (cross-node). We can summarize a few key points from the figure: As request rates increase, (1) latency rises while throughput gradually plateaus. This plateau represents the system's maximum processing capacity. (2) TTFT

---

[2]This model is downscaled from Llama3.1-405B to fit in GPU memory.

[3]We do not add prefill-decode disaggregated architectures to the baselines because gLLM can be applied into it as a superior backend system.

[4]↑ indicates higher values are better, while ↓ means lower values are preferable.

Tianyu Guo, Xianwei Zhang[#], Jiangsu Du, Zhiguang Chen, Nong Xiao, and Yutong Lu

(a) Qwen2.5-14B, ShareGPT

(b) Qwen2.5-14B, Azure

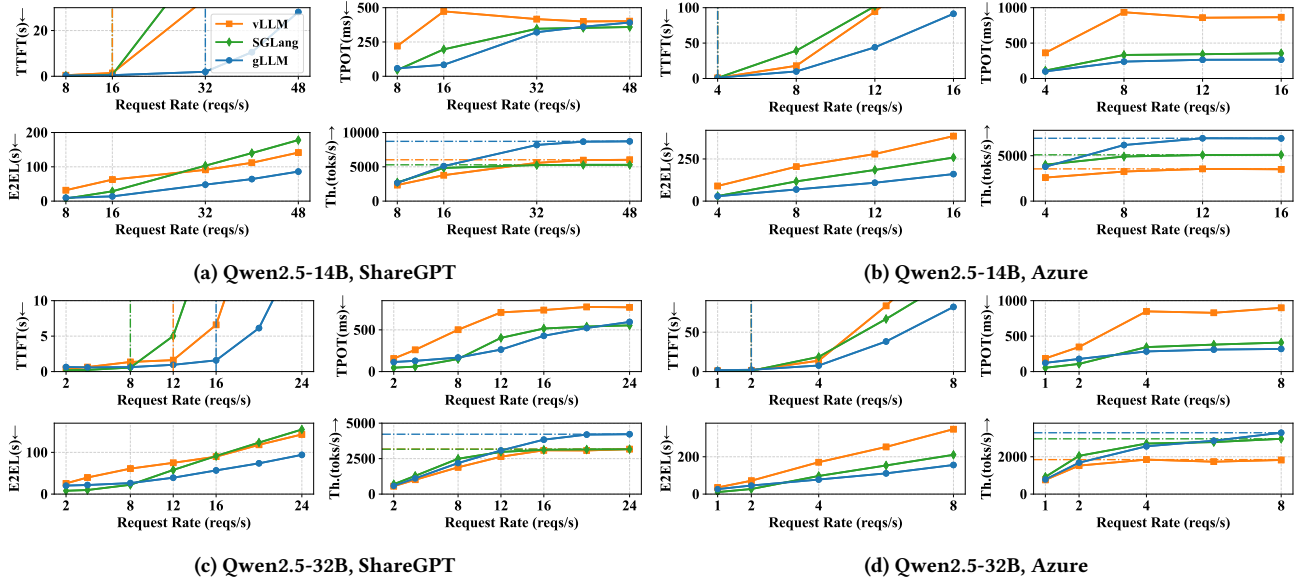(c) Qwen2.5-32B, ShareGPT

(d) Qwen2.5-32B, Azure

Figure 10: The latency and throughput comparison between vLLM, SGLang and gLLM (1 node with 4×L20).
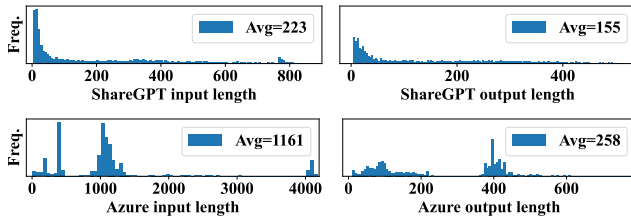
Figure 11: Distribution of input and output length of the sampled dataset.

(a) Intra-node scalability with L20

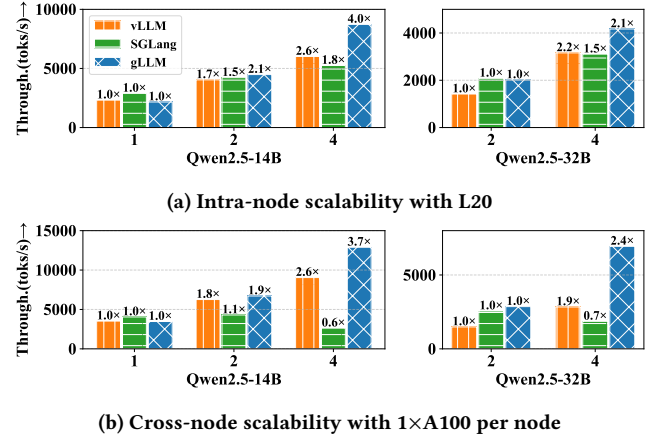(b) Cross-node scalability with 1×A100 per node

Figure 12: Variation of maximum throughput as the number of GPUs/nodes (horizontal axis) increases. The numbers (×) on the bar illustrate the multiples compared to single/two-card/node(s) performance.

will rise significantly at some point due to requests queuing. At the most situations, gLLM reaches its turning point at 2-6× higher request rates compared to other systems. (3) E2EL shows an approximately linear increase trend. At the most situations, gLLM achieves 0.14-0.92× lower slope compared to the other two systems. (4) In most cases, benefiting from the more efficient scheduling policy and runtime, gLLM significantly outperforms vLLM in both latency and throughput across the tested scenarios (processing capacity improves 0.29-1.50×). While gLLM performs slightly worse than vLLM when serving Llama3.1-100B on the Azure dataset at a request rate of 4, this is due to the abundance of tokens for prefilling and sufficient memory to hold the KV cache, which minimizes variations of scheduled tokens in Sarathi-Serve. However, such conditions are uncommon in real-world serving environments. (5) Tensor parallelism is well-suited for scenarios with low request rates and high bandwidth connection. For intra-node experiment, SGLang achieves lower latency compared to pipeline parallelism systems under low request rates. But as the request rate increases, the advantage of SGLang diminishes and even less than gLLM. For inter-node experiment, the performance of gLLM is significantly better than SGLang due to its high communication overhead.

(processing capacity improves 0.11-3.98×) (6) Compared to Azure, gLLM is better at handling ShareGPT. The reason is that gLLM can balance prefill tokens across different requests. While sequences in Azure encompass longer input, weakening inter-request parallelism.

4.3 Scalability Study

To evaluate the scalability of gLLM, we conduct maximum throughput tests against vLLM and SGLang by incrementally increasing request rates until system throughput stabilizes. Our results shown in Figure 12 reveal distinct scaling patterns across systems. While gLLM demonstrates marginally lower throughput than vLLM and
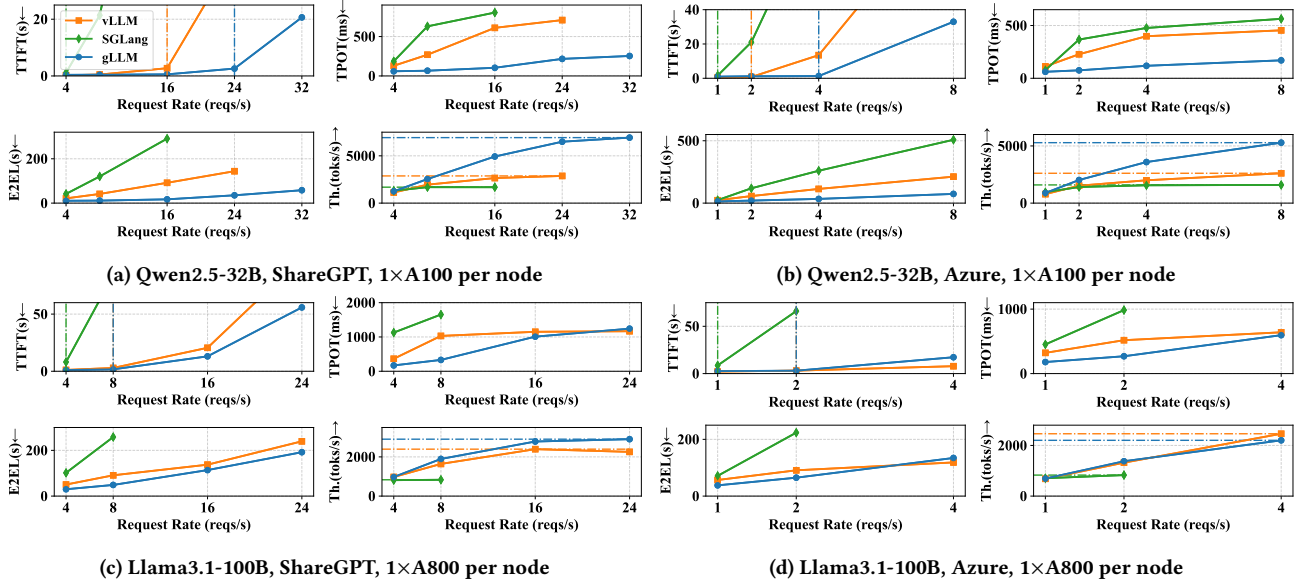
(a) Qwen2.5-32B, ShareGPT, 1×A100 per node

(b) Qwen2.5-32B, Azure, 1×A100 per node

(c) Llama3.1-100B, ShareGPT, 1×A800 per node

(d) Llama3.1-100B, Azure, 1×A800 per node

Figure 13: The latency and throughput comparison between vLLM, SGLang and gLLM (4 nodes).

SGLang when serving 14B model on single GPU configurations (attributed to incomplete optimizations), it achieves near-linear scaling efficiency as GPU counts increase. In contrast, vLLM exhibits sub-linear scaling with the 14B model but maintains linear scaling with the 32B variant. SGLang demonstrates sub-linear scaling within a single node, but experiences performance degradation as GPUs count increase in cross-node deployments due to high inter-GPU communication overhead. Notably, gLLM's performance advantage becomes progressively more pronounced with more GPUs, suggesting superior architectural scalability.

4.4 SLO Attainment

(a) ShareGPT with SLO TTFT:3000ms and TPOT:150ms

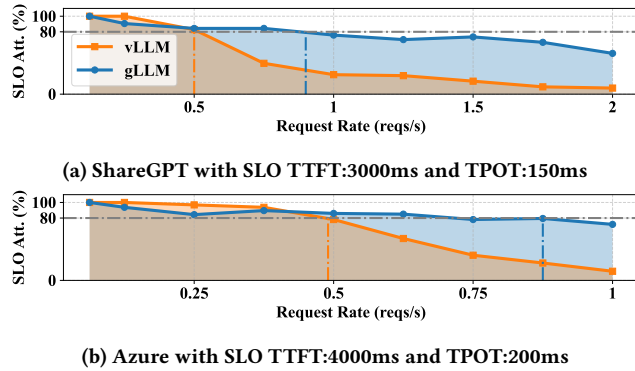(b) Azure with SLO TTFT:4000ms and TPOT:200ms

Figure 14: SLO attainment in cross-node deployments of Llama3.1-100B with A800.

To evaluate SLO attainment, we compare gLLM and vLLM in cross-node deployments of Llama3.1-100B. As shown in Figure 14, gLLM achieves 64% higher SLO attainment coverage than vLLM across

various request rates. Under 80% SLO attainment, gLLM can sustain 79% higher request rate compared to vLLM. While at low request rate, gLLM exhibits slightly lower SLO attainment due to a marginal increase in TTFT caused by Token Throttling, which occasionally exceeds the time constraint. To address this, we can fine-tune the hyperparameter $\#T$ to balance TTFT and TPOT performance.
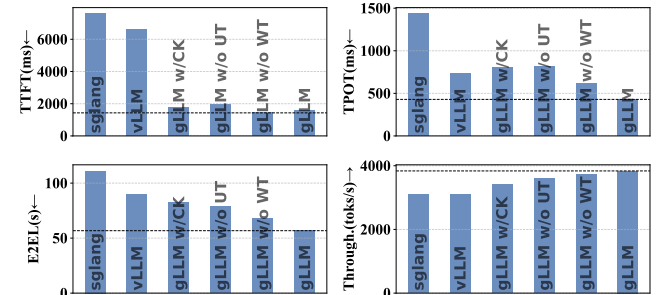
4.5 Ablation Study

Figure 15: An ablation study on the design choices of gLLM. The dashed line marks the optimal value under each metric.

To evaluate the effectiveness of our design, we conduct ablation experiments on gLLM, with the results shown in Figure 15. Notably, gLLM w/o WT achieves 10% lower TTFT compared to gLLM, as WT's balanced scheduling approach slightly compromises prefill speed. However, this comes at the cost of 44% higher TPOT and 20% longer E2EL. The absence of UT leads to even more significant performance degradation, increasing TTFT by 22%, TPOT by 91%, and E2EL by 38%, demonstrating UT's crucial role in balancing computation. Meanwhile, even with Sarathi-Serve's basic scheduling

strategy, gLLM w/CK achieves 10% higher throughput and 8% lower E2EL compared to vLLM. This demonstrates that gLLM runtime is more efficient than that of vLLM.
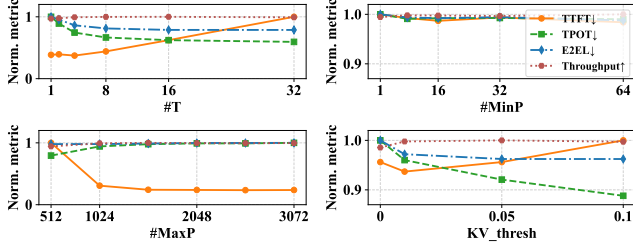
4.6 Sensitivity Study

Figure 16: Normalized metric under different setting of hyperparameters #T, #MaxP, #MinP and KV_{thresh}.

To examine the effect of the hyperparameters used in gLLM, we conduct a sensitivity study and the normalized results are shown in Figure 16.

4.6.1 Impact of #T. The parameter $#T$, representing the number of iterations to process all pending prefill tokens. As $#T$ increases, the number of prefill tokens per micro-batch decreases, which generally leads to longer TTFT due to reduced prefill rate. However, when $#T$ grows from 1 to 4, the TTFT remains stable because gains in computational parallelism, such as better GPU utilization and reduced idle time, offset the smaller batch sizes. Concurrently, TPOT improves as smaller micro-batches enable faster decoding rates. Throughput increases and E2EL decreases with larger $#T$, proving Token Throttling improve overall processing efficiency through more evenly distributed computations.

4.6.2 Impact of #MaxP. The hyperparameter $#MaxP$ governs the maximum number of batched prefill tokens processed per scheduling iteration. Increasing $#MaxP$ creates a dual effect: Higher prefill rates accelerate initial token generation (TTFT), while expanded batch sizes prolong per-token processing latency (TPOT). However, excessively conservative $#MaxP$ settings (e.g., 512) degrade system throughput due to suboptimal prefill rate, which constrains the system's capacity to handle concurrent requests during decode phases.

4.6.3 Impact of KV_{thresh}. KV_{thresh} defines the idle rate threshold for KV cache utilization. Setting KV_{thresh} to zero introduces inefficiencies: TTFT, TPOT and E2EL exhibit slight increases, while throughput declines. This occurs because a zero threshold pushes the system toward KV cache capacity limits more frequently, forcing preemption of ongoing requests to accommodate remaining decode requests. Such preemption waste computational resources and degrade overall performance.

4.6.4 Impact of #MinP. The minimum batched prefill token count ($#MinP$) exhibits limited impact on overall system performance across most configurations (within 2% performance fluctuations).

Table 1: Comparison of the number of lines of code and the score of MMLU-Pro [49] (evaualted on Qwen2.5-32B-Instruct) between gLLM, SGLang and vLLM.

Framework	gLLM	SGLang	vLLM(V1)	vLLM(V0)
Lines of code	3874	65097	226874	
MMLU-pro↑	68.86	68.85	NA	69.17

4.7 Functionality Study

Table 1 presents a comparison in lines of code and output quality across different frameworks. Notably, gLLM achieves comparable output quality to vLLM[5] and SGLang while maintaining a superior inference speed.

5 RELATED WORK

Scheduling in LLMs. Traditional DNN inference frameworks primarily employ batch-level scheduling [42], which struggles to handle the variable sequence lengths inherent in LLMs. To address this limitation, Orca [40] introduces iteration-level scheduling, enabling dynamic request admission and early exit before model execution. However, this approach faces challenges when processing lengthy prefill requests, as they can delay subsequent decode requests. To mitigate this imbalance, recent work proposes Sarathi-Serve [36], which runs prefill and decode operations together by splitting long sequences into smaller chunks. Despite these advancements, computational imbalance persists across each batch, significantly degrading the efficiency of pipeline parallelism.

LLMs serving systems. To efficiently serve LLMs, researchers have developed various systems. Orca [40], for instance, introduces a distributed serving system with iteration-level scheduling to improve throughput. For memory optimization, vLLM [41] employs paged attention, imitating virtual memory mechanisms to reduce fragmentation, while SGLang [46] leverages radix attention to eliminate redundant KV cache computations across requests. To address the divergent computational demands of the prefill and decode stages, Splitwise [37] and DistServe [38] adopt a disaggregated architecture, allocating specialized hardware configurations for each phase. Additionally, frameworks like FlexGen [51] and InfiniGen [52] tackle GPU memory constraints through adaptive techniques such as memory compression and intelligent offloading, ensuring scalable performance under resource limitations. Nevertheless, these systems fail to achieve balanced schedule between each batch, leading to significant performance bottleneck.

Model Parallelism for LLMs training and serving. Model parallelism has become essential for distributed training and serving of LLMs as their size increases. Tensor parallelism, which demands frequent communication between devices, is primarily employed in environments with high-bandwidth interconnects. Recent advancements [53–56] address communication idling by strategically overlapping communication operations with computation. For pipeline parallelism, research focuses on solving unbalanced memory consumption [29, 30, 34], pipeline bubbles [30, 33], communication optimization [31] and activation checkpointing [29, 32]. Hybrid strategies combining tensor pipellelism and pipeline parallelism leverage

---

[5]Due to the stability problem, vLLM with V1 crashes during the test.

automated search algorithms [57–59] or utilize the heterogeneous characteristics [58–61], while frameworks like Megatron-LM [35] demonstrate empirically validated configurations for massive-scale deployment. However, those methods focus on optimization during training. In LLM serving, specialized optimizations target hardware heterogeneity through GPU-aware allocation [22, 23] and reduce pipeline bubbles using chunked prefill mechanisms [36]. Nevertheless, pipeline bubbles are not efficiently solved by chunked prefill and become a bottleneck for efficient deploying.

6 CONCLUSION

This paper presents gLLM, a global balanced pipeline parallelism system for distributed LLM serving incorporating with Token Throttling method. Token Throttling dynamically adjusts the token count for prefill or decode stages respectively, ensuring balanced computation across batches and minimizing pipeline bubbles. To enable Token Throttling, we design an asynchronous runtime tailored for pipeline parallelism. Experiments with leading LLMs demonstrate that gLLM delivers 11% to 398% higher maximum throughput over state-of-the-art pipeline or tensor parallelism systems, while simultaneously maintaining lower latency.

References

[1] H. Sun, W. Xu, W. Liu, J. Luan, B. Wang, S. Shang, J. Wen, and R. Yan, "Determlr: Augmenting llm-based logical reasoning from indeterminacy to determinacy," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (L. Ku, A. Martins, and V. Srikumar, eds.), pp. 9828–9862, Association for Computational Linguistics, 2024.

[2] Y. Lei and R. Huang, "Boosting logical fallacy reasoning in llms via logical structure tree," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024* (Y. Al-Onaizan, M. Bansal, and Y. Chen, eds.), pp. 13157–13173, Association for Computational Linguistics, 2024.

[3] A. Toroghi, W. Guo, A. Pesaranghader, and S. Sanner, "Verifiable, debuggable, and repairable commonsense logical reasoning via llm-based theory resolution," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024* (Y. Al-Onaizan, M. Bansal, and Y. Chen, eds.), pp. 6634–6652, Association for Computational Linguistics, 2024.

[4] Y. Zhao, Y. Long, H. Liu, R. Kamoi, L. Nan, L. Chen, Y. Liu, X. Tang, R. Zhang, and A. Cohan, "Docmath-eval: Evaluating math reasoning capabilities of llms in understanding financial documents," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (L. Ku, A. Martins, and V. Srikumar, eds.), pp. 16103–16120, Association for Computational Linguistics, 2024.

[5] Q. Li, L. Cui, X. Zhao, L. Kong, and W. Bi, "Gsm-plus: A comprehensive benchmark for evaluating the robustness of llms as mathematical problem solvers," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (L. Ku, A. Martins, and V. Srikumar, eds.), pp. 2961–2984, Association for Computational Linguistics, 2024.

[6] H. Liu, Z. Zheng, Y. Qiao, H. Duan, Z. Fei, F. Zhou, W. Zhang, S. Zhang, D. Lin, and K. Chen, "Mathbench: Evaluating the theory and application proficiency of llms with a hierarchical mathematics benchmark," in *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024* (L. Ku, A. Martins, and V. Srikumar, eds.), pp. 6884–6915, Association for Computational Linguistics, 2024.

[7] Z. Lu, A. Zhou, H. Ren, K. Wang, W. Shi, J. Pan, M. Zhan, and H. Li, "Mathgenie: Generating synthetic data with question back-translation for enhancing mathematical reasoning of llms," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (L. Ku, A. Martins, and V. Srikumar, eds.), pp. 2732–2747, Association for Computational Linguistics, 2024.

[8] A. Didolkar, A. Goyal, N. R. Ke, S. Guo, M. Valko, T. P. Lillicrap, D. J. Rezende, Y. Bengio, M. C. Mozer, and S. Arora, "Metacognitive capabilities of llms: An exploration in mathematical problem solving," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024* (A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, eds.), 2024.

[9] A. Setlur, S. Garg, X. Geng, N. Garg, V. Smith, and A. Kumar, "RL on incorrect synthetic data scales the efficiency of LLM math reasoning by eight-fold," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024* (A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, eds.), 2024.

[10] R. Schumann, W. Zhu, W. Feng, T. Fu, S. Riezler, and W. Y. Wang, "VELMA: verbalization embodiment of LLM agents for vision and language navigation in street view," in *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada* (M. J. Wooldridge, J. G. Dy, and S. Natarajan, eds.), pp. 18924–18933, AAAI Press, 2024.

[11] A. Martin, C. Pande, H. F. Witschel, and J. Mathez, "Chedbot: Designing a domain-specific conversational agent in a simulational learning environment using llms," in *Proceedings of the AAAI 2024 Spring Symposium Series, Stanford, CA, USA, March 25-27, 2024* (R. P. A. Petrick and C. W. Geib, eds.), pp. 180–187, AAAI Press, 2024.

[12] C. Toukmaji and A. Tee, "Retrieval-augmented generation and LLM agents for biomimicry design solutions," in *Proceedings of the AAAI 2024 Spring Symposium Series, Stanford, CA, USA, March 25-27, 2024* (R. P. A. Petrick and C. W. Geib, eds.), pp. 273–278, AAAI Press, 2024.

[13] S. Deng, W. Xu, H. Sun, W. Liu, T. Tan, J. Liu, A. Li, J. Luan, B. Wang, R. Yan, and S. Shang, "Mobile-bench: An evaluation benchmark for llm-based mobile agents," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (L. Ku, A. Martins, and V. Srikumar, eds.), pp. 8813–8831, Association for Computational Linguistics, 2024.

[14] H. Shi, Z. Sun, X. Yuan, M. Côté, and B. Liu, "Opex: A component-wise analysis of llm-centric agents in embodied instruction following," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (L. Ku, A. Martins, and V. Srikumar, eds.), pp. 622–636, Association for Computational Linguistics, 2024.

[15] Q. Yang, Z. Wang, H. Chen, S. Wang, Y. Pu, X. Gao, W. Huang, S. Song, and G. Huang, "Psychogat: A novel psychological measurement paradigm through interactive fiction games with LLM agents," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (L. Ku, A. Martins, and V. Srikumar, eds.), pp. 14470–14505, Association for Computational Linguistics, 2024.

[16] DeepSeek-AI, A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Guo, D. Yang, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Zhang, H. Ding, H. Xin, H. Gao, H. Li, H. Qu, J. L. Cai, J. Liang, J. Guo, J. Ni, J. Li, J. Wang, J. Chen, J. Chen, J. Yuan, J. Qiu, J. Li, J. Song, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Xu, L. Xia, L. Zhao, L. Wang, L. Zhang, M. Li, M. Wang, M. Zhang, M. Zhang, M. Tang, M. Li, N. Tian, P. Huang, P. Wang, P. Zhang, Q. Wang, Q. Zhu, Q. Chen, Q. Du, R. J. Chen, R. L. Jin, R. Ge, R. Zhang, R. Pan, R. Wang, R. Xu, R. Zhang, R. Chen, S. S. Li, S. Lu, S. Zhou, S. Chen, S. Wu, S. Ye, S. Ye, S. Ma, S. Wang, S. Zhou, S. Yu, S. Zhou, S. Pan, T. Wang, T. Yun, T. Pei, T. Sun, W. L. Xiao, and W. Zeng, "Deepseek-v3 technical report," *CoRR*, vol. abs/2412.19437, 2024.

[17] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Rozière, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. M. Kloumann, I. Misra, I. Evtimov, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Upasani, K. Plawiak, K. Li, K. Heafield, K. Stone, and et al., "The llama 3 herd of models," *CoRR*, vol. abs/2407.21783, 2024.

[18] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Debbah, É. Goffinet, D. Hesslow, J. Launay, Q. Malartic, D. Mazzotta, B. Noune, B. Pannier, and G. Penedo, "The falcon series of open language models," *CoRR*, vol. abs/2311.16867, 2023.

[19] OpenAI, "GPT-4 technical report," *CoRR*, vol. abs/2303.08774, 2023.

[20] M. Reid, N. Savinov, D. Teplyashin, D. Lepikhin, T. P. Lillicrap, J. Alayrac, R. Soricut, A. Lazaridou, O. Firat, J. Schrittwieser, I. Antonoglou, R. Anil, S. Borgeaud,

A. M. Dai, K. Millican, E. Dyer, M. Glaese, T. Sottiaux, B. Lee, F. Viola, M. Reynolds, Y. Xu, J. Molloy, J. Chen, M. Isard, P. Barham, T. Hennigan, R. McIlroy, M. Johnson, J. Schalkwyk, E. Collins, E. Rutherford, E. Moreira, K. Ayoub, M. Goel, C. Meyer, G. Thornton, Z. Yang, H. Michalewski, Z. Abbas, N. Schucher, A. Anand, R. Ives, J. Keeling, K. Lenc, S. Haykal, S. Shakeri, P. Shyam, A. Chowdhery, R. Ring, S. Spencer, E. Sezener, and et al., "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," *CoRR*, vol. abs/2403.05530, 2024.

[21] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *J. Mach. Learn. Res.*, vol. 23, pp. 120:1–120:39, 2022.

[22] J. Stojkovic, C. Zhang, Í. Goiri, J. Torrellas, and E. Choukse, "Dynamollm: Designing LLM inference clusters for performance and energy efficiency," *CoRR*, vol. abs/2408.00741, 2024.

[23] Y. Mei, Y. Zhuang, X. Miao, J. Yang, Z. Jia, and R. Vinayak, "Helix: Distributed serving of large language models via max-flow on heterogeneous gpus," *CoRR*, vol. abs/2406.01566, 2024.

[24] A. Borzunov, M. Ryabinin, A. Chumachenko, D. Baranchuk, T. Dettmers, Y. Belkada, P. Samygin, and C. A. Raffel, "Distributed inference and fine-tuning of large language models over the internet," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023* (A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, eds.), 2023.

[25] B. Zhang, H. Zhu, F. Gao, Z. Yang, and X. S. Wang, "Moirai: Towards optimal placement for distributed inference on heterogeneous devices," *CoRR*, vol. abs/2312.04025, 2023.

[26] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, and P. B. Gibbons, "Pipedream: Fast and efficient pipeline parallel DNN training," *CoRR*, vol. abs/1806.03777, 2018.

[27] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel DNN training," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event* (M. Meila and T. Zhang, eds.), vol. 139 of *Proceedings of Machine Learning Research*, pp. 7937–7947, PMLR, 2021.

[28] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. X. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada* (H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, eds.), pp. 103–112, 2019.

[29] Z. Sun, H. Cao, Y. Wang, G. Feng, S. Chen, H. Wang, and W. Chen, "Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024* (R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, eds.), pp. 86–100, ACM, 2024.

[30] Z. Liu, S. Cheng, H. Zhou, and Y. You, "Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023* (D. Arnold, R. M. Badia, and K. M. Mohror, eds.), pp. 56:1–56:13, ACM, 2023.

[31] J. Lin, Z. Liu, Y. You, J. Wang, W. Zhang, and R. Zhao, "Weipipe: Weight pipeline parallelism for communication-effective long-context large model training," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2025, Las Vegas, NV, USA, March 1-5, 2025*, pp. 225–238, ACM, 2025.

[32] W. Liu, M. Li, G. Tan, and W. Jia, "Mario: Near zero-cost activation checkpointing in pipeline parallelism," in *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2025, Las Vegas, NV, USA, March 1-5, 2025*, pp. 197–211, ACM, 2025.

[33] P. Qi, X. Wan, G. Huang, and M. Lin, "Zero bubble (almost) pipeline parallelism," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*, OpenReview.net, 2024.

[34] T. Kim, H. Kim, G. Yu, and B. Chun, "Bpipe: Memory-balanced pipeline parallelism for training large language models," in *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA* (A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, eds.), vol. 202 of *Proceedings of Machine Learning Research*, pp. 16639–16653, PMLR, 2023.

[35] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia, "Efficient large-scale language model training on GPU clusters using megatron-lm," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021* (B. R. de Supinski, M. W. Hall, and T. Gamblin, eds.), p. 58, ACM, 2021.

[36] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in LLM inference with sarathi-serve," in *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (A. Gavrilovska

and D. B. Terry, eds.), pp. 117–134, USENIX Association, 2024.

[37] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative LLM inference using phase splitting," in *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024*, pp. 118–132, IEEE, 2024.

[38] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (A. Gavrilovska and D. B. Terry, eds.), pp. 193–210, USENIX Association, 2024.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, eds.), pp. 5998–6008, 2017.

[40] G. Yu, J. S. Jeong, G. Kim, S. Kim, and B. Chun, "Orca: A distributed serving system for transformer-based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022* (M. K. Aguilera and H. Weatherspoon, eds.), pp. 521–538, USENIX Association, 2022.

[41] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023* (J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, eds.), pp. 611–626, ACM, 2023.

[42] "Faster Transformer." https://github.com/NVIDIA/FasterTransformer.

[43] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022* (S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds.), 2022.

[44] T. Dao, "Flashattention-2: Faster attention with better parallelism and work partitioning," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*, OpenReview.net, 2024.

[45] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024* (A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, eds.), 2024.

[46] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, C. W. Barrett, and Y. Sheng, "Sglang: Efficient execution of structured language model programs," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024* (A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, eds.), 2024.

[47] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu, Z. Cui, Z. Zhang, and Z. Qiu, "Qwen2.5 technical report," *CoRR*, vol. abs/2412.15115, 2024.

[48] A. Zeng, B. Xu, B. Wang, C. Zhang, D. Yin, D. Rojas, G. Feng, H. Zhao, H. Lai, H. Yu, H. Wang, J. Sun, J. Zhang, J. Cheng, J. Gui, J. Tang, J. Zhang, J. Li, L. Zhao, L. Wu, L. Zhong, M. Liu, M. Huang, P. Zhang, Q. Zheng, R. Lu, S. Duan, S. Zhang, S. Cao, S. Yang, W. L. Tam, W. Zhao, X. Liu, X. Xia, X. Zhang, X. Gu, X. Lv, X. Liu, X. Liu, X. Yang, X. Song, X. Zhang, Y. An, Y. Xu, Y. Niu, Y. Yang, Y. Li, Y. Bai, Y. Dong, Z. Qi, Z. Wang, Z. Yang, Z. Du, Z. Hou, and Z. Wang, "Chatglm: A family of large language models from GLM-130B to GLM-4 all tools," *CoRR*, vol. abs/2406.12793, 2024.

[49] Y. Wang, X. Ma, G. Zhang, Y. Ni, A. Chandra, S. Guo, W. Ren, A. Arulraj, X. He, Z. Jiang, T. Li, M. Ku, K. Wang, A. Zhuang, R. Fan, X. Yue, and W. Chen, "Mmlu-pro: A more robust and challenging multi-task language understanding benchmark," in *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024* (A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, eds.), 2024.

[50] "ShareGPT." https://sharegpt.com/.

[51] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single GPU," in *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA* (A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, eds.), vol. 202 of *Proceedings of Machine Learning Research*, pp. 31094–31116, PMLR, 2023.

[52] W. Lee, J. Lee, J. Seo, and J. Sim, "Infinigen: Efficient generative inference of large language models with dynamic KV cache management," in *18th USENIX*

Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024 (A. Gavrilovska and D. B. Terry, eds.), pp. 155–172, USENIX Association, 2024.

[53] A. Jangda, J. Huang, G. Liu, A. H. N. Sabet, S. Maleki, Y. Miao, M. Musuvathi, T. Mytkowicz, and O. Saarikivi, "Breaking the computation and communication abstraction barrier in distributed machine learning workloads," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, eds.), pp. 402–416, ACM, 2022.

[54] S. Wang, J. Wei, A. Sabne, A. Davis, B. Ilbeyi, B. Hechtman, D. Chen, K. S. Murthy, M. Maggioni, Q. Zhang, S. Kumar, T. Guo, Y. Xu, and Z. Zhou, "Overlap communication with dependent computation via decomposition in large deep learning models," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023* (T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, eds.), pp. 93–106, ACM, 2023.

[55] C. Chen, X. Li, Q. Zhu, J. Duan, P. Sun, X. Zhang, and C. Yang, "Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024-1 May 2024* (R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, eds.), pp. 178–191, ACM, 2024.

[56] J. Du, J. Wei, J. Jiang, S. Cheng, D. Huang, Z. Chen, and Y. Lu, "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, PPoPP 2024, Edinburgh, United Kingdom, March*

2-6, 2024 (M. Steuwer, I. A. Lee, and M. Chabbi, eds.), pp. 42–54, ACM, 2024.

[57] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022* (M. K. Aguilera and H. Weatherspoon, eds.), pp. 559–578, USENIX Association, 2022.

[58] S. Zhang, L. Diao, C. Wu, Z. Cao, S. Wang, and W. Lin, "HAP: SPMD DNN training on heterogeneous GPU clusters with automated program synthesis," in *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*, pp. 524–541, ACM, 2024.

[59] T. Um, B. Oh, M. Kang, W. Lee, G. Kim, D. Kim, Y. Kim, M. Muzzammil, and M. Jeon, "Metis: Fast automatic distributed training on heterogeneous gpus," in *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024* (S. Bagchi and Y. Zhang, eds.), pp. 563–578, USENIX Association, 2024.

[60] X. Jia, L. Jiang, A. Wang, W. Xiao, Z. Shi, J. Zhang, X. Li, L. Chen, Y. Li, Z. Zheng, X. Liu, and W. Lin, "Whale: Efficient giant model training over heterogeneous gpus," in *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022* (J. Schindler and N. Zilberman, eds.), pp. 673–688, USENIX Association, 2022.

[61] M. Ryabinin, T. Dettmers, M. Diskin, and A. Borzunov, "SWARM parallelism: Training large models can be surprisingly communication-efficient," in *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA* (A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, eds.), vol. 202 of *Proceedings of Machine Learning Research*, pp. 29416–29440, PMLR, 2023.