Optimizing SLO-oriented LLM Serving with PD-Multiplexing

Weihao Cui*
Shanghai Jiao Tong University

Yukang Chen*
Shanghai Jiao Tong University

Han Zhao
Shanghai Jiao Tong University

Ziyi Xu
Shanghai Jiao Tong University

Quan Chen†
Shanghai Jiao Tong University

Xusheng Chen
Huawei Cloud

Yangjie Zhou
Shanghai Jiao Tong University

Shixuan Sun
Shanghai Jiao Tong University

Minyi Guo
Shanghai Jiao Tong University

Abstract

Modern LLM services demand high throughput and stringent SLO guarantees across two distinct inference phases–prefill and decode–and complex multi-turn workflows. However, current systems face a fundamental tradeoff: out-of-place compute partition enables per-phase SLO attainment, while in-place memory sharing maximizes throughput via KV cache reuse. Moreover, existing in-place compute partition also encounters low utilization and high overhead due to phase-coupling design. We present Drift, a new LLM serving framework that resolves this tension via PD multiplexing, enabling in-place and phase-decoupled compute partition. Drift leverages low-level GPU partitioning techniques to multiplex prefill and decode phases spatially and adaptively on shared GPUs, while preserving in-place memory sharing. To fully leverage the multiplexing capability, Drift introduces an adaptive gang scheduling mechanism, a contention-free modeling method, and a SLO-aware dispatching policy. Evaluation shows that Drift achieves an average 5.1× throughput improvement (up to 17.5×) over state-of-the-art baselines, while consistently meeting SLO targets under complex LLM workloads.

Keywords: LLM Serving, SLO, Spatial multiplexing

1 Introduction

Recently, large language models (LLMs)[18, 35, 36] have significantly expanded their capacity. LLM services now perform well not only in simple natural language processing but also in more complex tasks [19, 28, 31]. At the single-request level, LLMs process inputs through a prefill phase, followed by multiple decode phases to generate responses. At the application level, a complete serving workflow may involve multiple turns of requests. Interactions between these requests enable LLMs to support tasks such as multi-turn conversations and agent-based workloads.

These complex LLM services inherently have stringent Service Level Objective (SLO) requirements. To achieve both SLO guarantees and high throughput, managing GPU compute and memory resources has become critical. First, appropriate compute allocation [2, 30, 45] between the two distinct LLM inference phases–prefill and decode–is essential for meeting their respective SLO targets[1]. This is because the two phases interleave during execution and exhibit distinct performance characteristics. Second, efficiently reusing intermediate results [22, 24, 44] through a memory pool, known as the KV cache, is crucial for further improving throughput under SLO constraints. In complex serving workflows like multi-turn conversations, cross-request interactions are enabled by sharing the KV cache among requests. Reusing the KV cache directly accelerates inference by trading memory for reduced redundant computation.

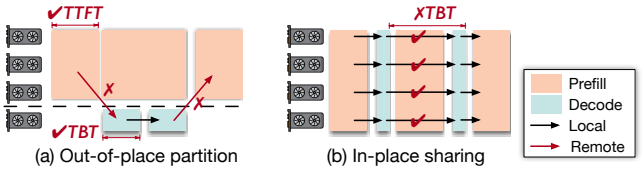


Figure 1. Examples of processing multi-turn conversations in modern LLMs with out-of-place partition and in-place sharing.

Unfortunately, current LLM serving systems tend to favor one aspect over the other due to the misalignment of management requirements, as shown in Figure 1. *Out-of-place compute partition is widely adopted in LLM serving system to ensure SLO guarantees.* In this case, GPUs are disaggregated into prefill and decode instances for fulfilling the distinct SLOs of prefill and decode phases. *Conversely, in-place memory sharing is critical to trade memory for higher throughput.* Consistent KV memory allocation across phases and requests boosts throughput by directly accessing the KV cache.

Integrating both approaches necessitates frequent KV cache transfers between instances; otherwise, additional GPUs must be reserved to recompute the corresponding KV cache. E.g., prefilling a request with 1K new tokens and 127K cached

*Both authors contributed equally to this research.
†Quan Chen is the corresponding author.

[1]We follow prior work [30] by using time to first token (TTFT) as the SLO for prefill and time between tokens (TBT) as the SLO for the decode.

tokens using LLaMA-70B [18] requires either migrating over 40GB of memory from the decode instance (133ms) or wasting at least 128× FLOPs on recomputation (15.9s). In contrast, prefilling the same request with direct KV cache sharing takes only 205ms. The situation worsens as modern LLM services involve complex cross-request interactions.

Such misalignment calls for an in-place compute partition to avoid interfering with memory sharing. Prior works [2, 17] attempt to pool compute resources separately from memory. Rather than instance-level disaggregation, they split the long-context prefill phase into chunks and tightly couple each chunk's execution with the execution of a decode phase. In this monolithic design, the chunk size governs both the SLO attainment of the decode phase and the ability to saturate the GPUs with the split prefill phase. Our investigations in §2.3 show that finding the sweet chunk size in practice is infeasible: increasing the chunk size violates the SLO, while the SLO-compliant chunk size fails to saturate the GPU. Worse still, chunking incurs overhead from extra memory accesses to the KV caches of previous chunks [45]. This overhead is amplified in complex workloads, where KV caches are shared not only across phases but also between requests (§5.2.1). Ultimately, chunking-based method significantly limits the system's ability to maximize throughput while meeting SLO targets.

In this paper, we propose DRIFT, an LLM serving framework that simultaneously ensures SLO attainment and achieves high peak throughput for today's complex LLM services. The key insight of DRIFT is that intra-GPU spatial multiplexing is better suited for online LLM serving, enabling both in-place and phase-decoupled partitioning. Based on this, DRIFT introduces a novel *PD multiplexing approach* that allows adaptive compute partitioning and migration-free memory sharing across LLM inference phases and requests. Specifically, DRIFT integrates the low-level compute partitioning technique GreenContext [8] to multiplex the prefill and decode phases spatially and independently on each GPU. This integration enables fast switching between different spatial sharing partitions for SLO-aware scheduling. To fully leverage this capability, DRIFT introduces an adaptive gang scheduling mechanism, a contention-free modeling method, and an SLO-aware dispatching policy to increase flexibility and minimize GPU bubbles when scheduling the prefill and decode phases concurrently. Consequently, compute resources can be judiciously and dynamically allocated to different phases with minimal overhead, effectively meeting SLO requirements.

We implement DRIFT on top of SGLang [44], extending it with the proposed multiplexing approach. DRIFT is evaluated comprehensively on both small and large LLMs using real-world workloads of complex LLM services. With flexible spatial multiplexing, experiments show that DRIFT delivers an average 5.1× throughput improvement (up to 17.5×) over state-of-the-art solutions, while meeting SLO guarantees.

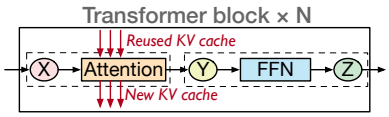


Figure 2. Main architecture of most LLMs.

Despite the notable performance improvement, DRIFT also introduces a simple yet effective design for current LLM serving systems. Unlike approaches based on disaggregation, DRIFT requires neither intensive engineering effort to transfer the KV cache nor complex coordination between prefill and decode instances. Unlike approaches based on prefill chunking, DRIFT avoids tightly coupling the prefill and decode phases, along with the associated overhead. Its design choice reduces complexity and improves the robustness of existing LLM serving systems. We plan to open-source DRIFT after publication.

In summary, we make the following contributions:

- We identify that efficient SLO-oriented LLM serving are hindered either by the misalignment between out-of-place partitioning and in-place sharing, or by phase-coupled designs in in-place partitioning.
- We propose a PD multiplexing approach with a clean design that effectively meets the in-place compute partition requirements of modern LLM inferences.
- We extensively evaluate DRIFT, demonstrating its effectiveness in enhancing SLO-oriented LLM serving compared to state-of-the-art solutions.

2 Background & Motivation

2.1 Modern LLM Services

Architecture of LLMs Most LLMs [1, 5, 18, 35, 36] are built upon the transformer architecture [37], with model-specific modifications. Figure 2 illustrates a typical transformer block, which is replicated multiple times to form an LLM model. Each transformer block consists of two consecutive layers: an attention layer followed by a feed-forward network (FFN) layer. Specifically, the attention computation requires access to all keys and values from previously processed tokens. To avoid redundant computations, modern LLM inference systems store this information in a KV-cache. We omit the discussion of other layers such as embedding, as they contribute minimally to the overall inference time.

Complex workloads Modern LLM services efficiently handle complex workloads through multi-level interactions. Figure 1 illustrates a typical example in multi-turn conversational workloads. At the request level, input tokens are first processed in the *prefill* phase to generate the initial output token. This is followed by a sequence of *decode* phases, where each subsequent token is generated based on all previously generated tokens, including the most recent one. At

Table 1. Summary of theoretical compute analysis for prefill and decode phases.

	Attention	FFN
Prefill phase w/o cache	$O(Ld^2 + L^2d)$	$O(Ld^2)$
Prefill phase w/ cache	$O(nd^2 + Lnd)$	$O(nd^2)$
Decode phase	$O(d^2 + (r+1)d)$	$O(d^2)$

the application level, multiple turns of LLM service invocations collaborate to complete the conversation with the user. During each phase, either prefill or decode, the serving system retrieves KV cache generated in previous phases and requests, and generates new KV cache as well.

2.2 Characterizing the Complex LLM Services

We now characterize complex LLM services to underscore the urgency of integrating compute partitioning with efficient memory sharing. We first present a theoretical analysis, followed by an experimental study under SLO constraints.

Theoretical analysis of compute complexity In complex workloads, the latency of each prefill or decode phase is determined by the token length of the reused context and new context. Since the decode phase can be viewed as a special prefill case with the new context including only one token, we unify our theoretical analysis by focusing solely on the prefill phase. We also treat the batch size as 1 for simplicity. The key factors are listed as below:

- d: The hidden dimension of each token's representation.
- L: The total token length.
- r: The token length of the reused (cached) context.
- $n = L - r$: The token length of the new context.

As for attention, each input token is linearly projected into Query (Q), Key (K), and Value (V) vectors. Attention scores are computed by taking the dot product of Q and K, followed by a weighted sum over V. For the FFN, it is typically a two-layer feed-forward network with a hidden size of $4d$. Without KV cache reuse, attention complexity is often approximated as $O(Ld^2 + L^2d)$: generating the Q/K/V vectors costs roughly $O(Ld^2)$, and the $QK^T \cdot V$ operation costs $O(L^2d)$. With KV cache reuse, generating the Q/K/V vectors of n new tokens costs roughly $O(nd^2)$, and the $QK^T \cdot V$ operation costs $O(Lnd)$. Similarly, the attention complexity of decode phase is $O(d^2 + (r+1)d)$ as $L = r + n$ and $n = 1$.

Meantime, without KV cache reuse, the FFN complexity under prefill phase can be approximated as $O(Ld^2)$. When KV cache is enabled, the FFN complexity also reduces linearly to $O(nd^2)$. Also, for the decode phase, the FFN complexity is $O(d^2)$ as n equals 1.

Table 1 summarizes the compute analysis of the prefill phase (without and with cache reuse) and the decode phase. From the table, first, efficient KV cache reuse is critical in the prefill phase with shared context, as it reduces compute complexity linearly in both the attention and FFN layers. Second, even with reused context, the prefill phase remains significantly higher compute complexity than the decode phase. As a result, in-place interleaved execution easily leads to SLO violations, since prefill phase blocks decode phase.

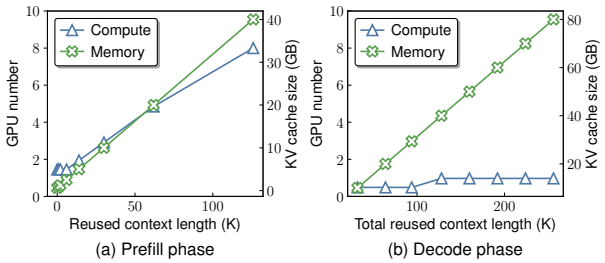
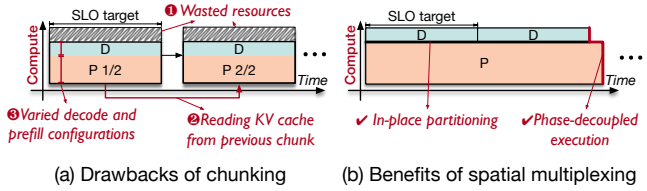


Figure 3. Required compute resources and accessed memory for processing different inference phases under SLO constraints with varying reused context lengths. In the prefill phase, the batch size is fixed at 1, the new context length is set to $2K$, and TTFT is set to $400ms$. In the decode phase, the batch size is fixed at 32, and TBT is set to $80ms$. These values are commonly seen in online serving.

Experimental analysis under SLO constraint We further conduct experiments to investigate how context length affects the compute and memory resource demands of both the prefill and decode phases. Specifically, we run LLaMA-70B [18] on a server equipped with 8 A100 GPUs, using tensor parallelism [43] across all GPUs. In all experiments, the model runs under spatial partition mode, where each GPU allocates only a portion of its compute resources using GreenContext [8]. All GPUs are configured with the same partition ratio. Based on this setup, we identify the best-fit GPU partition ratio, denoted as GPU_{ratio}, that satisfies the SLO target for a given context length. Then, we compute the overall GPU resource demand as $GPU_{num} = GPU_{ratio} \times 8$. Figure 3 shows the detailed results.

As shown in Figure 3, LLM inference in the prefill phase requires increasingly more compute resources to meet SLO targets as the reused context length grows. In contrast, the compute resource demand of the decode phase shows minimal sensitivity. Therefore, it is critical to allocate more compute resources to the prefill phase as context length increases. Further, the distinct compute requirements of prefill and decode phases necessitate partitioning compute resources between them at runtime. While prior works [30, 45] have also recognized this need in simple LLM workloads, our findings show that the context length variation in complex services amplifies the necessity of appropriate compute partition.

Meantime, Figure 3 shows that the KV cache required by both the prefill and decode phases can easily grow to tens or even hundreds of gigabytes. This is primarily due to complex LLM services involving multi-turn interactions, which result in ultra-long context lengths. To avoid redundant computation, it is preferable to keep the KV cache in the same



(a) Drawbacks of chunking (b) Benefits of spatial multiplexing

Figure 4. Execution timeline comparison between chunking-based solution and DRIFT.

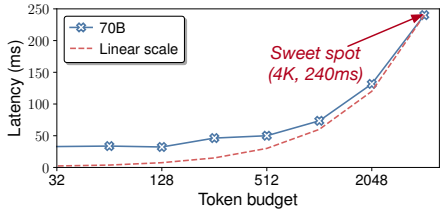


Figure 5. Sweet spot of the token budget in chunk-based solutions. The decode phase uses a fixed batch size of 32, with each request having a reused context length of 1K tokens.

memory space through persistent allocation. This memory sharing strategy is critical for maintaining high throughput.

Summary *We make two key observations in characterization: 1) Appropriate compute partition is essential for meeting the distinct SLO targets of different phases. 2) Reusing the shared KV cache with persistent allocation is critical for reducing redundant computation and improving throughput.*

2.3 Inefficiencies of Existing Works

Misaligned requirements To fulfill the above observations, the straightforward solution is to integrate current compute partition methods [30, 38, 45] with memory sharing support. However, we find that doing so leads to inefficiencies. This is because there is a misalignment of management requirements between appropriate compute partition and efficient memory sharing.

Current methods [30, 38, 45] generally adopt out-of-place compute partition to ensure SLO guarantees. Specifically, they partition GPUs into prefill and decode instances, which not only compute but also memory resources. However, KV cache reuse requires in-place memory sharing to trade memory for higher throughput. The out-of-place compute partition prevents direct cache sharing across phases and requests for reuse.

Faced with the above misalignment, integrating compute partition methods with memory sharing necessitates frequent KV cache transfers between instances. Indeed, transfers from a prefill instance to a decode instance can be partially optimized using techniques like layer-wise transfer [30]. However, transfers in the reverse direction, from decode to prefill, lack such optimization opportunities. As a result, the latency of an LLM request with reused context increases due to either waiting for cache transfer or recomputing the KV cache.

Phase coupling of prefill chunking Prefill chunking [2] is a dynamic compute partition technique that preserves KV cache reuse. It splits a long-context prefill phase into smaller chunks and couples the execution of each chunk with that of a decode phase. However, as illustrated in Figure 4-(a), this phase-coupling design makes it difficult to maximize the throughput under SLO constraints due to three drawbacks.

The first is resource waste caused by the infeasibility to achieve the sweet spot between SLO attainment and high utilization. Figure 5 presents the execution time of varying chunk sizes, paired with a decode phase using a fixed configuration. The token budget in the figure is the sum of the decode batch size (32 in this experiment) and the prefill chunk size. This budget is used in the paper for determining SLO guarantee [2]. As shown, the latency does not increase linearly with the token budget until it reaches $4K$. This indicates that the sweet spot for fully utilizing GPU resources requires a prefill chunk with a context length of $(4K - 32)$. However, the latency of this coupled execution is $240ms$, significantly exceeding the typical TBT SLO target ($< 100ms$). In this case, to meet a small TBT SLO target, the prefill chunk and decode phases often run inefficiently.

The second is that prefill chunking requires repeated reads of the KV cache from previous chunks. This can result in two inefficiencies: 1) It incurs quadratic memory overhead [45]. 2) It may fail to meet SLO targets of the decode phase when the reused context length is too large. Each chunked prefill can be viewed as a prefill with cache. As shown in Table 1, its compute complexity increases with the length of the reused context. When the reused context becomes large, like thousands of tokens in a chunk, attention computation can dominate the chunk's latency. In such cases, the chunk's latency already exceeds the TBT SLO target of the coupled decode phase. Our evaluation in §5.2.1 further confirms this.

The last is its incompatibility with acceleration techniques like CUDA Graph [9], which requires static configuration. The large space formed by varying chunk sizes and decode batch sizes makes it infeasible to apply CUDA Graph for acceleration.

2.4 Opportunity

Given the fundamental limitations of misaligned requirements between compute partition and memory sharing, it is desirable to find a lightweight and adaptive way to manage the compute resources. Without modifying the execution mode of LLM inference phases, we find that intra-GPU spatial multiplexing offers a new opportunity.

For example, Figure 4-(b) shows an ideal execution timeline of the prefill and decode phases scheduled with GreenContext [8], a low-level compute partitioning technique provided by NVIDIA. Firstly, since GreenContext can partition

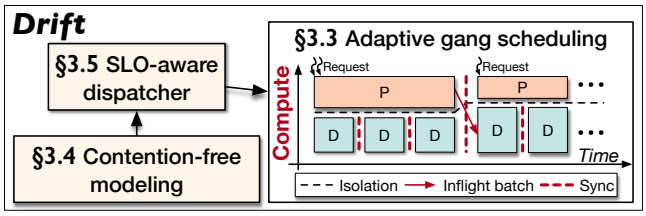


Figure 6. Architecture overview of DRIFT.

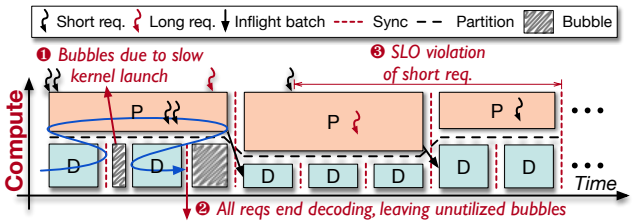


Figure 7. Bubbles and SLO violation of phase-level gang scheduling. → represents the kernel launch order.

the compute resources of a single GPU across concurrent GPU kernels, it enables in-place partitioning between the prefill and decode phases. Secondly, GPU kernels running in different green contexts execute independently, thereby allowing phase-decoupled execution. Thirdly, the low-level partitioning technique preserves the native execution mode of LLM inference and remains compatible with existing optimizations such as CUDA Graph [9].

Therefore, intra-GPU spatial multiplexing motivates us to integrate memory sharing and compute partition for achieving high throughput and SLO guarantee at the same time.

3 DRIFT's Design

3.1 Architecture Overview

The observations in §2.4 motivate DRIFT, an LLM serving framework that introduces a novel spatial prefill-decode (PD) multiplexing approach. DRIFT is designed to achieve both SLO compliance and high throughput in complex LLM workloads. Figure 6 presents the system architecture of DRIFT. DRIFT consists of: (1) an offline modeling component that captures the concurrent behavior of prefill and decode phases to enable SLO-aware scheduling, and (2) an online serving component that spatially multiplexes these phases to maximize throughput while meeting SLO targets.

(1) Offline modeling DRIFT first profiles the prefill and decode latencies under representative workloads across various compute partitions. Using these profiling results, DRIFT constructs latency predictors for decode and prefill phases via a contention-free modeling method.

(2) Online serving As requests arrive, DRIFT's SLO-aware dispatcher groups them into prefill batches and determines compute partition between the prefill and decode phases, adhering to SLO constraints (§3.5). These decisions are guided by the latency predictor. Subsequently, the prefill and ongoing decode batches execute concurrently using an adaptive gang scheduling mechanism designed to minimize execution bubbles (§3.3). Beyond compute partition, this mechanism also incorporates existing optimizations such as inflight batching [39] and CUDA Graphs [9].

3.2 Strawman Integration

A strawman approach to integrating GreenContext [8], the low-level compute partition mechanism, into LLM serving is to allocate separate green contexts for the prefill and decode phases. Then, the LLM serving system can switch between these contexts to perform appropriate compute partition between these phases.

In the runtime scheduling, several aspects must be handled carefully. First, the prefill and decode phases must synchronize to enable inflight batching, which is the key technique for increasing decode-phase parallelism and achieving high throughput. As these phases run concurrently and periodically synchronize to move requests from prefill to decode, we refer to this scheduling approach as phase-level gang scheduling. Second, SLO-aware modeling and dispatching are required to manage compute partition between the prefill and decode phases. Without proper partition, SLO violations may occur or throughput may not be maximized.

In the rest of this section, we first address the inefficiencies of phase-level gang scheduling, and then introduce the contention-aware modeling and SLO-aware dispatching mechanisms in DRIFT.

3.3 Adaptive Gang Scheduling

Phase-level gang scheduling in strawman integration can suffer from several inefficiencies when handling dynamic online serving workloads. Figure 7 illustrates these inefficiencies with detailed scheduling cases.

Firstly, bubbles exist due to slow GPU kernel launch. While GreenContext enables intra-process compute partitioning, DRIFT must decide whether to launch prefill or decode kernels first within the process. In state-of-the-art LLM serving systems, the prefill phase launches kernels sequentially on the CPU, while the decode phase launches all kernels at once using CUDA Graph [22, 44]. Since graph launching takes less than 0.5 millisecond, whereas launching prefill kernels takes tens of milliseconds, launching the prefill kernels first would introduce a bubble. As a result, it is preferable to launch the decode phase first.

Unfortunately, launching the decode phase first still incurs bubbles. An LLM serving system must return the newly generated tokens after each decode phase, preventing multiple decode phases from being launched simultaneously without host synchronization. As illustrated on the left of Figure 7, DRIFT must interleave kernel launches for two phases. This leads to bubbles during the second decode phase.

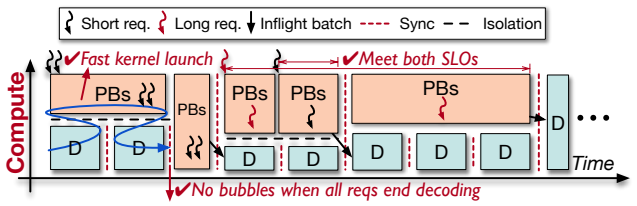


Figure 8. Adaptive gang scheduling using a block-level scheduling unit for the prefill phase and a graph-level scheduling unit for the decode phase. PBs is the short for prefill blocks. → indicates the kernel launch order.

Secondly, bubbles arise due to the unpredictable termination of the decode phase, as illustrated in the mid-bottom of Figure 7. The completion of decoding cannot be determined in advance, as it depends on the outcome of the current decode operation. Consequently, all requests in a decode batch may finish token generation while a concurrent prefill phase has already been launched with preconfigured compute partition. Worse, due to the non-preemptive nature of the GPU execution model, the launched GPU kernels cannot be interrupted to incorporate newly released compute resources from the decode phase.

Thirdly, SLO violations also arise due to workload skew among accepted requests, as shown in the upper right of Figure 7. New context lengths may vary significantly, like short conversations and long-text summarizations co-existing. In such cases, a request with a short new context may experience long queuing delays, waiting for the prefill phase of a request with an ultra-long context to complete. When the short request has tight SLO headroom, it is highly likely to miss its deadline.

Block-wise kernel launching for prefill The aforementioned inefficiencies mainly stem from the discrepant scheduling units of the prefill and decode phases. The prefill phase typically takes longer to launch and execute. To bridge this gap, DRIFT introduces the adaptive gang scheduling by slicing the entire prefill phase into multiple prefill blocks (PBs). Importantly, block-wise slicing does not alter the internal execution model of the LLM, since LLMs are inherently composed of multiple transformer blocks.

As shown in Figure 8, DRIFT performs scheduling at block-level granularity for the prefill phase and graph-level granularity for the decode phase. As for kernel launching, DRIFT can now launch enough PBs to fully occupy the compute resource partitioned to the prefill phase, and return in time before the completion of the launched decode phase to process its results. As for decode termination, DRIFT can now switch the execution of later prefill blocks into a new green context, just after the end of the terminated decode phase. As for short requests, DRIFT can now preempt the prefill execution of long requests to prioritize short ones, thereby meeting the SLO targets of both. In this way, bubbles are eliminated and SLO attainment is ensured.

Query-based synchronization To enable inflight batching in adaptive gang scheduling, the prefill phase must synchronize with the decode phase. A straightforward approach is to insert a CUDA event after the last prefill block, blocking the launch of the next decode phase until the event completes. Once the event signals completion, the prefill request is added to the ongoing decode batch.

However, this blocking approach introduces execution bubbles, as the prefill and decode phases typically do not finish at the same time. Instead, DRIFT adopts a query-based synchronization strategy that periodically polls the status of CUDA events. Rather than waiting, DRIFT continues launching decode batches and prefill blocks asynchronously. Once a CUDA event is confirmed as complete, the prefill request is immediately added to the ongoing decode batch.

3.4 Contention-free Modeling

GreenContext [8] enables compute partition by setting the number of SMs a GPU kernel can use. When the prefill and decode phases run concurrently, contention may arise from other resources, such as memory bandwidth. We start with contention analysis of the spatial multiplexing between two phases and then provide our modeling method.

Contention-awareness Analysis. Contention between concurrent GPU kernels under spatial multiplexing is difficult to model [26, 41], primarily due to three factors.

The first issue is compute unit contention. Previous works rely on techniques like CUDA MPS [12] or CUDA streams [10] for compute partition. However, neither of these techniques guarantees that scheduled kernels can run concurrently with a precise number of GPU compute units (SMs), leading to contention. GreenContext resolves this by precisely controlling the number of compute units assigned to each kernel. There is no overlap among green contexts created within the same group.

The second issue is bandwidth contention. While GreenContext partitions compute units, it does not guarantee bandwidth partition along the critical memory access path. To address this, we first present a widely accepted principle suggesting that multiplexing the prefill and decode phases under high load is contention-free, and then validate it with theoretical analysis. We focus on the high-load scenario, as the multiplexing of two phases under low load does not saturate the GPU and incurs less performance interference. The widely accepted principle in GPU spatial multiplexing [21, 34, 40, 41] is as follows:

Principle 1. *When there is no contention over compute units, only two GPU kernels that are both memory-bound will exhibit significant contention, leading to noticeable slowdown.*

Table 2. Ratio of theoretical memory access time to compute time for key GPU kernels in modern LLMs.

Kernel	BS	New context	Reused context	Compute (ms)	Memory (ms)	Ratio
QKV	256	-	-	0.0412	0.0275	0.666
O	256	-	-	0.0138	0.0105	0.765
UG	256	-	-	0.0964	0.0605	0.628
D	256	-	-	0.0482	0.0317	0.659
Extend Attn	1	1024	8196	0.124	0.00334	0.027
Decode Attn	256	1	1024	0.00344	0.0661	**19.2**

Based on this principle, we investigate the compute intensity of today's LLMs on commonly used GPUs. Specifically, we evaluate LLaMA-70B [18] on our testbed server described in §5.1, which is equipped with 8 A100-SXM4-80GB GPUs. The derived conclusion generalizes to both smaller models and newer GPUs. We begin by comparing the ratio of theoretical memory access time to compute time to identify the bottlenecks in key GPU kernels. While this ratio does not directly capture actual compute intensity, we later validate our findings through extensive profiling.

Given the peak compute of 320 TFLOPs and memory bandwidth of 2039 GB/s on the A100-SXM4-80GB, we report the resulting ratio in Table 2. In the table, QKV denotes the GEMM kernel for generating Query, Key, and Value; O denotes the output GEMM kernel of attention; UG represents the up-gating GEMM kernel in the FFN; D denotes the down-projection GEMM kernel in the FFN. Extend Attn refers to the attention kernel in the prefill phase with context reuse, while Decode Attn refers to the attention kernel in the decode phase. The batch size is set to 256, which is commonly the upper bound in online serving. A larger batch size further reduces the ratio. The configuration of `QKV, O, UG, D` is set according to the decode phase. The settings in the prefill phase are significantly more compute-bound than those in the decode phase.

As shown in the table, only the attention kernel in the decode phase exhibits a ratio greater than 1, indicating it is memory-bound. All other kernels have ratios below 1, suggesting they are compute-bound. This result can be attributed to two main reasons. First, modern LLMs widely adopt new attention algorithms, such as grouped query attention, which significantly reduce memory access. Second, the FFN dimensions have been substantially increased. Modern LLMs are more likely to become compute-bound on new advanced GPUs [13, 15]. This conclusion has also been validated by existing works [46].

The third issue is the network. In online serving, the communication overhead of the decode phase is relatively low, as it depends only on the batch size. In contrast, the prefill phase dominates bandwidth consumption, with its communication cost scaling with the product of batch size and new context length. Given the high bandwidth provided by NVLink [16] (600GB/s on our testbed), network contention between prefill and decode phases remains low.

In conclusion, we find that multiplexing the prefill and decode phases of modern LLMs on today's advanced GPUs incurs minimal contention. To further validate this, we directly profile the prefill and decode phases under spatial multiplexing using Llama-8B and Llama-70B. The total context length for the prefill batch (reused + new) ranges from 1024 to 32,768 tokens, while the product of decode batch size and per-request reused context length spans from 1024 to 4,194,304. The prefill phase is assigned no fewer than half of the A100's SMs, as we find this rarely happens in online serving. Extensive profiling across over 8600 cases shows that the slowdown caused by contention in both phases remains below 7% at the $90\%-ile$, with a maximum of 17%. For online serving, such deviation is acceptable for guiding scheduling decisions. Therefore, Drift directly uses predictors trained from solo-run profiling results.

Solo-run The space for prefill and decode batches formed online is vast. Given the dynamic assignments of compute partition, profiling all possible configurations to support online decision-making is infeasible. Based on the complexity analysis in Table 1, we borrow the modeling method from previous work [38] to build the latency prediction model for prefill phase and decode phase in complex LLM serving workloads. The prediction model of prefill phase is formulated as Equation 1, while the prediction model of decode phase is formulated as Equation 2. In the two equations, all θ are coefficients.

$$T_{Prefill} = \theta_1 \cdot \sum_{i}^{bs} n_i^2 + \theta_2 \cdot \sum_{i}^{bs} n_i \cdot r_i + \theta_3 \cdot \sum_{i}^{bs} n_i + \theta_4 \quad (1)$$

$$T_{Decode} = \theta_1 \cdot \sum_{i}^{bs} r_i + \theta_2 \cdot bs + \theta_3 \quad (2)$$

For each compute partition configuration, a prediction model is trained using offline profiled data. Although the decode phase is a special case of prefill, Drift still trains two separate sets of models. This is because, under the serving framework, the execution paths of prefill and decode phases differ entirely, including their GPU kernel implementations and the methods used to launch them. The offline profiling for model training is a one-time effort per model and can be completed in several hours, which is acceptable. The trained models achieve high accuracy, with a maximum deviation of 8.16% for prefill and 8.84% for decode, effectively supporting Drift's online scheduling.

3.5 SLO-aware Dispatching

With adaptive gang scheduling and contention-free modeling, we introduce Drift's detailed dispatching policy.

Priorities of prefill and decode In Drift, we prioritize SLO attainment for the decode phase and process the prefill phase as early as possible. SLO attainment for the prefill phase is not directly guaranteed for two reasons. Firstly,

although we prioritize the decode phase, we only allocate just-enough compute resources for it. Since the remaining compute resources are allocated to the prefill phase, its SLO is generally expected to be met. Secondly, when SLO violations occur for the prefill phase, it indicates that the inference load has exceeded the peak capacity of the current LLM serving instance. In such cases, further scheduling efforts would no longer improve performance.

Dispatching policy With above analysis, Algorithm 1 describes Drift's SLO-aware dispatching policy. The system operates in a loop, continuously dispatching requests from the queue and adaptively allocating compute resources between the prefill and decode phases (lines 1-5).

Specifically, each iteration begins by invoking `GeneratePB` to determine the next prefill batch. It then calls `Partition` to divide compute resources between prefill and decode phases based on estimated time budgets, and splits the current prefill batch into blocks. Finally, `Process` executes the prefill blocks and the decode batch concurrently. Once the prefill batch completes, it is merged into the decode batch to enable in-flight batching.

As for the `GeneratePB` function, it is responsible for selecting the next prefill batch (lines 6-20). If there are preempted batches on the stack, it resumes them with priority. Otherwise, it retrieves a new batch from the request queue. If there is no ongoing prefill batch, the new batch is returned directly. If a batch is already in progress, the system estimates the combined execution time of the current and new batches using the contention-aware model (§3.4). If their total predicted time remains within the SLO budget, the current batch is preempted and pushed onto the stack, and the new batch is adopted. Otherwise, the new batch is returned to the queue, and processing continues with the current batch.

When preempting an ongoing prefill batch, Drift only allows a preemption stack size of one, meaning a prefill batch can be preempted at most once. This design is reasonable, as in most cases only a short request preempts the execution of a long one. Preempting the short request in turn would likely cause it to violate its SLO. Drift checks SLO attainment only when a prefill batch is preempted; otherwise, it focuses solely on processing the prefill batch as quickly as possible.

After determining the next prefill batch, the `Partition` function then determines how to allocate compute resources between two phases under the SLO constraints (lines 21-24). It returns the number of compute units assigned to each phase and the set of prefill blocks to be processed in this round. When generating prefill blocks, Drift estimates their latency based on the total latency of the current prefill phase and the assigned compute units.

Algorithm 1 Drift's SLO-aware dispatching policy

Q_{req}: request queue.
$Block_{PB}$: prefill blocks to be processed.
PB: the ongoing prefill batch.
$Stack_{PB}$: the preempted prefill blocks
DB: the ongoing decode batch.
C_{PB}: the compute partitioned to prefill batch.
C_{DB}: the compute partitioned to decode batch.

1: **while true do**
2: $PB \leftarrow$ GeneratePB(PB, DB, C_{PB}, C_{DB})
3: $Block_{PB}, C_{PB}, C_{DB} \leftarrow$ Partition(PB, DB, SLO_{TBT})
4: Process$(Block_{PB}, DB, C_{PB}, C_{DB})$
5: **if** PB.is_finished() **then** $DB.merge(PB)$
6: **function** GeneratePB(PB, DB, C_{PB}, C_{DB})
7: **if** !$Stack_{PB}$.is_empty() **then**
8: **if** PB.is_empty() **then return** $Stack_{PB}$.pop()
9: **elsereturn** PB
10: $PB_{new} \leftarrow Q_{req}.get\_new\_batch()$
11: **if** PB.is_empty() **then return** PB_{new}
12: **else**
13: $T_{PB_{new}} \leftarrow predicting(PB_{new}, DB, C_{PB}, C_{DB})$
14: $T_{PB} \leftarrow predicting(PB, DB, C_{PB}, C_{DB})$
15: **if** $T_{PB} + T_{PB_{new}} \leq PB.SLO\_headroom()$ **then**
16: $Stack_{PB}.push(PB)$
17: **return** PB_{new},
18: **else**
19: $Q_{req}.put\_back(PB_{new})$
20: **return** PB
21: **function** Partition(PB, DB, SLO_{TBT})
22: $C_{PB}, C_{DB} \leftarrow modeling(PB, DB, SLO_{TBT})$
23: $Block_{PB} \leftarrow PB.new\_blocks(DB, C_{DB})$
24: **return** $Block_{PB}, C_{PB}, C_{DB}$

4 Implementation

We implement Drift and integrate it with SGLang [44] and PyTorch [29]. The implementation consists of approximately 10k lines of Python and 100 lines of C++ and CUDA code. Since PyTorch does not natively support GreenContext, the spatial multiplexing mechanism provided by CUDA [8], we first develop a PyTorch extension to enable its use for all GPU operations launched by PyTorch.

While GreenContext is a lightweight mechanism for spatial multiplexing, it still introduces runtime overhead when created on demand. Drift pre-creates multiple groups of green contexts before serving begins and switches to the required configurations based on guidance from the SLO-aware dispatcher. This approach introduces some additional memory overhead due to internal data structures and joint use with CUDA Graph [9]. Our evaluation in §5.3.3 shows that this overhead is acceptable. Moreover, Drift is equipped with four groups of green contexts. The SM partitions for each group are configured as $(108, 0)$, $(84, 24)$, $(72, 36)$, and $(0, 108)$, where the first number denotes the SMs allocated to the prefill phase, and 108 is the total number of SMs on the A100 GPU. Our practical experience shows that this

is sufficient for online scheduling. Finer granularity does not improve performance and instead introduces additional memory overhead.

5 Evaluation

5.1 Experimental Setup

Testbed We evaluate Drift on a server equipped with an Intel(R) Xeon(R) Gold 6346 CPU with 64 cores and 8 A100-SXM4-80GB GPUs. The GPUs are interconnected via NVLINK, providing a total bandwidth of 600 GB/s. The server has 1 TB of host memory. To ensure correct operation of GreenContext [8], we use GPU driver version 570.124.06 and CUDA 12.8. All experiments are conducted with PyTorch 2.6.1 [29]. Drift is implemented using SGLang [44] version 0.3.6post3.

Models We primarily evaluate Drift using two LLMs from the Llama family [18, 35, 36]: Llama-8B and Llama-70B. These models, differing in size, represent the most commonly hosted LLMs in the cloud.

Workloads To evaluate Drift for complex LLM services, we conduct experiments using two real-world LLM workload traces from production [32], referred to as `Conversation` and `Tool&Agent`. `Conversation` consists of multi-turn dialogues, representing the most common use case for LLMs. Requests from the same user within a dialogue often share context. `Tool&Agent` captures workloads where LLMs are deployed as tools or agents to perform tasks such as coding [19]. These tasks typically follow pre-defined workflows, where context is also shared. As these two traces are collected from a large serving cluster, Drift scales them down to better fit a single serving instance. To evaluate Drift with requests that do not share context, we follow prior works [2, 38] and generate synthetic workloads using datasets like ShareGPT [4] and LooGle [23] with a Poisson process. While ShareGPT exhibits short context reuse, LooGLE features long contexts with many requests sharing them.

Baselines We compare Drift against four state-of-the-art solutions for efficient LLM serving. Since the evaluation uses 8 GPUs, model parallelism techniques such as tensor parallelism [43] are employed to parallelize the deployed models. For Drift, we fix the tensor parallelism degree to 8. Details of each baseline's model parallelism configuration are provided when the baseline is introduced, as disaggregation affects the choice of model parallelism. For all systems, the KV cache memory pool is configured as large as possible to maximize throughput.

- **SGLang [44]**: This is one of the most popular LLM serving systems. It also introduces RadixCache to efficiently share context across requests. The tensor parallelism is set to 8. We only compare with SGLang on peak throughput, as it does not provide SLO guarantees.
- **Prefill Chunking in SGLang [2]**: This is a version of SGLang equipped with prefill chunking, as proposed by SARATHI-Serve [2]. Similar to SARATHI-Serve, it splits long requests into chunks to provide SLO guarantees for the decode phase based on a predefined token budget. We follow SARATHI-Serve's methodology to calculate the token budget for each workload prior to experiments. The best token budget is 512.
- **LoongServe [38]**: This is a disaggregation-based solution with an elastic P:D ratio. Specifically, it proposes elastic sequence parallelism to dynamically scale the number of GPUs used for processing ongoing batches of requests. We follow LoongServe's model parallelism configuration. For Llama-70B, the sequence parallelism is set to 2, and tensor parallelism to 4. For Llama-8B, the sequence parallelism is set to 4, and tensor parallelism to 2. LoongServe's scheduling policy provides SLO guarantees for the decode phase through elastic scaling.
- **Dynamo [14]**: This is NVIDIA's latest implementation of a disaggregation-based solution. Its backend is vLLM [22], a widely used LLM serving system. Dynamo introduces a high-performance communication framework, NIX [27], to transfer the KV cache between prefill and decode instances. For Llama-8B, we set the P:D ratio to 2:1, with tensor parallelism set to 2 for prefill instances and 1 for decode instances. For Llama-70B, we set the P:D ratio to 1:1 due to memory constraints, and both prefill and decode instances use tensor parallelism of 4. This configuration follows the practices established by state-of-the-art disaggregation-based systems.

Metrics We focus on efficient SLO-oriented LLM serving. Therefore, throughout the evaluation, we measure the *99%-ile* of TTFT and TBT for all processed requests. We also measure the peak throughput within the constraints of a pre-defined SLO targets. When measuring SLO attainment, we set the TBT SLO target to 50ms for Llama3-8B and 100ms for Llama3-70B, following prior works [2, 32]. Moreover, the new context length processed during prefill depends on whether the KV cache is hit. In Drift, the TTFT SLO target is set per request upon arrival, once the new context length is determined by the existing caching mechanism [44]. Given the varying new context lengths, Drift sets the TTFT to 1 second per 1K tokens for each accepted request. We do not impose such constraints on the baselines.

5.2 End-to-end Performance

5.2.1 Real-world Workloads. We begin by evaluating Drift with Llama-8B and Llama-70B under real-world workload traces, compared to baseline systems. Figure 9 shows the latency distribution of TTFT and TBT. Although the real-world traces are scaled down to a modest level, disaggregation-based solutions still easily reach their peak throughput and enter an unstable state, where requests experience significant

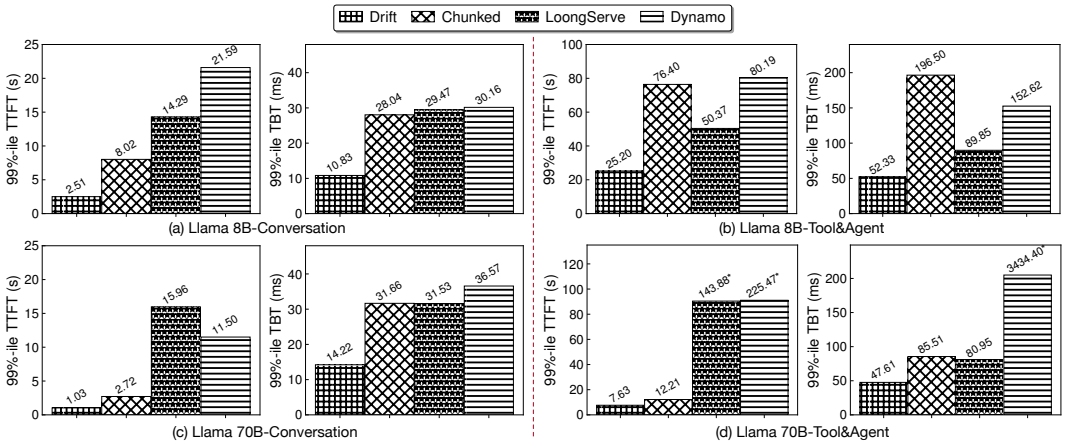


Figure 9. 99% − *ile* TTFT and TBT for Llama-8B and Llama-70B on real-world Conversation and Tool&Agent workloads. Chunked represents prefill chunking in SGLang. Values marked with ∗ are too large; we clip their corresponding bars, so the bar height only reflects their relative size.

queuing or frequent drops. After omitting unstable results of baselines in Figure 9-d, Drift achieves average *99%-ile* TTFT speedups of 2.62×, 7.72×, and 9.88× over prefill chunking, LoongServe, and Dynamo, respectively, and corresponding TBT speedups of 2.59×, 2.22×, and 2.67×.

Compared to in-place prefill chunking, Drift avoids resource waste and prefill overhead, bringing better performance for both prefill and decode phases. Notably, the token budget used for chunking is already finely tuned. We observe that adjusting the budget—either increasing or reducing it—fails to improve performance, especially in terms of TBT. This is because reused context length in prefill phase in complex LLM services can reach up to 50K KV tokens, which dominates the latency of chunked prefill. Further spliting the prefill into smaller chunks does not help control the TBT of the coupled decode phase. Drift's phase-decoupled multiplexing avoids this issue entirely.

Against the two out-of-place disaggregation-based solutions, Drift performs significantly better on TTFT. In LoongServe, due to instance scaling, KV cache required for reuse in the prefill phase cannot be transferred back from the decode phase, leading to redundant recomputation. In Dynamo, the KV cache is transferred between prefill and decode instances repeatedly. Although the transfer framework is highly optimized, it still incurs substantial prefill waiting time. Moreover, due to its static disaggregation, Dynamo often suffers from idle resources in either the prefill or decode instance under fluctuating real-world workloads. Moreover, while both LoongServe and Dynamo are disaggregation-based systems, LoongServe outperforms Dynamo in most cases due to its scalable instance design. This is particularly evident when serving smaller models, as LoongServe supports a wider scaling range (i.e., a higher degree of sequence parallelism). This observation further underscores the need for dynamic compute partitioning in online LLM serving, which also motivates the design of Drift.

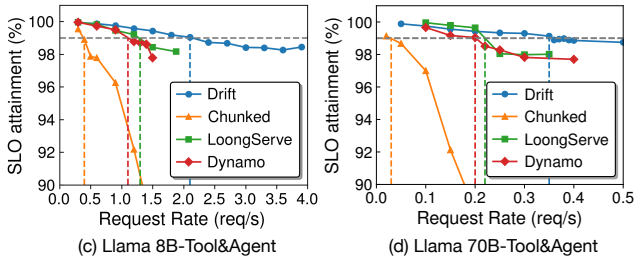


Figure 10. SLO attainment for Llama-8B and Llama-70B on Tool&Agent workload with varied request rates.

5.2.2 SLO Attainment and Peak Throughput. We also measure the SLO attainment of TBT to evaluate the effectiveness of Drift in meeting SLO guarantees. In this experiment, we extract requests from the Tool&Agent trace but replace their arrival timestamps with those generated by a Poisson process at varying rates, following prior work [38]. Figure 10 shows the SLO attainment results under gradually increasing workloads. Compared to all baselines, Drift performs significantly better in providing SLO guarantees for TBT. Under the constraint of meeting the *99%-ile* SLO guarantees, Drift achieves 6.2×, 1.63×, and 2.10× higher peak supported throughput than prefill chunking, LoongServe, and Dynamo, respectively, for Llama-8B; and 17.5×, 1.4×, and 1.75× for Llama-70B. Meanwhile, it is worth noting that Drift achieves shorter TTFT across all cases (1.46 × ~8.74×), except for prefill chunking with LLaMA-70B. This exception arises due to the significant gap between the peak supported loads of Drift and prefill chunking.

It is also evident that prefill chunking fails to meet the TBT SLO even under lower request rates than Dynamo and

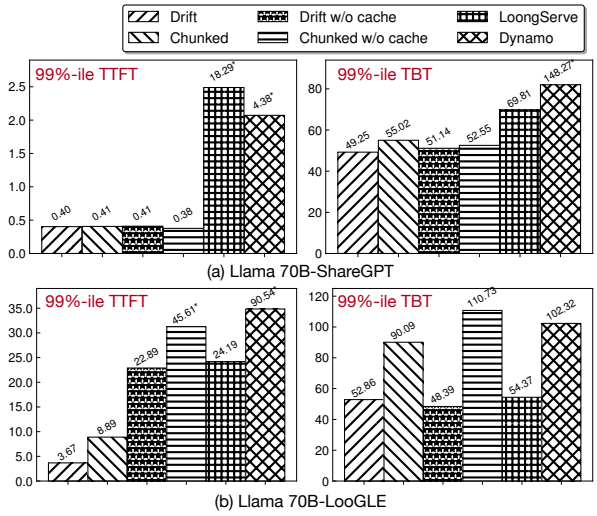


Figure 11. Comparison of *99%-ile* TTFT and TBT on synthetic workload from ShareGPT and LooGLE.

LoongServe, not to mention when compared to Drift. This is because chunking is ineffective at reducing TBT in complex LLM services, where cross-request interactions frequently occur. In contrast, out-of-place partitioning allows LoongServe and Dynamo to control TBT more effectively.

5.2.3 Performance on Synthetic Workloads. In the rest of the evaluation, we primarily focus on experiments with LLaMA-70B due to space constraints, as other results are similar. We also evaluate Drift using commonly adopted synthetic workloads from prior works [2, 24, 38, 45]. In this experiment, requests are generated by sampling inputs from ShareGPT and LooGLE, and are fed to the serving systems at a modest request rate following a Poisson process. Drift just enables seamless integration of in-place partition and sharing. The cache reused across the requests is not Drift's contribution. Thus we also disable the cross-request sharing for prefill chunking and Drift for further comparison. All frameworks run in stable state.

Figure 11 compares the *99%-ile* TTFT and TBT across Drift, three baselines, and two additional variants. In ShareGPT, Drift, prefill chunking, and their two variants perform similarly in both TTFT and TBT. This is expected, as requests from ShareGPT rarely share KV cache, and the request rate is modest. However, LoongServe and Dynamo still perform worse in terms of TTFT, as they continue to migrate the KV cache generated during the prefill phase to the decode instance. LoongServe performs slightly better because it can partially hide the transfer cost within the attention computation. As for TBT, both LoongServe and Dynamo show slightly higher values than Drift and prefill chunking. This is because Drift and prefill chunking benefit from a higher degree of tensor parallelism. When the decode batch size is small, a larger tensor parallelism increases the parallelism of underlying computations. With the high bandwidth provided by NVLink [16], Drift and prefill chunking achieve better TBT performance.

Table 3. Peak throughput of SGlang and Drift with LooGLE.

Model	SGLang	Drift	Speedup
ShareGPT	9.05	11.13	1.23×
LooGLE	1.94	2.21	1.14×

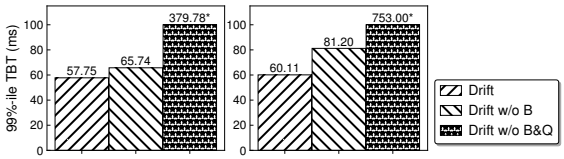


Figure 12. Comparison of Drift's TBT with and without adaptive gang scheduling.

5.2.4 Peak Throughput without SLO Constraints. While Drift is designed for efficient SLO-oriented LLM serving, we also evaluate it against the original SGLang to demonstrate its robustness. In this experiment, we lift the TBT constraints in Drift and use ShareGPT and LooGle to generate synthetic requests. During the experiment, we also run versions of both Drift and SGLang with cache sharing between requests disabled to further investigate the impact. Table 3 shows the peak throughput comparison without SLO constraints. Drift outperforms SGLang, achieving 1.23× and 1.14× higher throughput on ShareGPT and LooGLE. Drift performs better by consistently utilizing GPU resources more efficiently. When a decode batch underutilizes resources, Drift avoids stalling the prefill batch and instead allocates resources to enable both phases to run concurrently.

5.3 Ablation Study

5.3.1 Effectiveness of Adaptive Gang Scheduling. As shown in Figure 7, bubbles commonly occur in the green context created for decode. In this experiment, we compare the TBT of Drift against its two variants. First, we disable block-wise scheduling in adaptive gang scheduling. Second, we further disable the query-based synchronization optimization. The workloads used are Tool&Agent under two different request rates.

Figure 12 presents the experimental results. As shown in the figure, disabling block-wise slicing slightly increases the TTFT of decode by approximately 10ms, which aligns with the typical kernel launch time for the full prefill phase of LLaMA-70B. When query-based synchronization is further disabled, Drift suffers a significant degradation—314ms for LLaMA-8B and 672ms for LLaMA-70B—due to frequent stalls waiting for the entire prefill phase to complete.

5.3.2 Different Partition Groups. To investigate the performance of Drift under different compute partition group

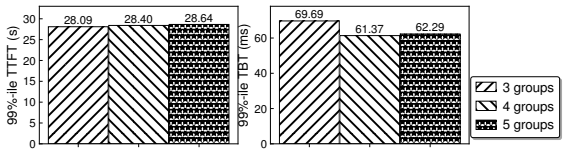


Figure 13. Comparison of Drift's TBT with different number of partition groups.

configurations, we conduct experiments by varying the number of partition groups. For 3 groups, the configurations are (108, 0), (80, 28), and (0, 108). For 4 groups, the configurations are (108, 0), (84, 24), (72, 36), and (0, 108). For 5 groups, the configurations are (108, 0), (80, 28), (84, 24), (72, 36), and (0, 108). As shown in Figure 13, the 4-group and 5-group settings achieve similar performance. In contrast, the 3-group configuration is less effective in controlling TBT, as it offers only two options for the decode phase and may fail to allocate appropriate partitions.

5.3.3 Overhead of Spatial Multiplexing.

Memory Drift introduces additional memory overhead due to integrating GreenContext into existing serving systems. Creating a group of green contexts incurs only 4MB, which is small compared to the total memory of modern GPUs. However, integrating it with CUDA Graph results in 743MB of memory usage for both LLaMA-8B and LLaMA-70B on 8 A100 GPUs. This is because the serving system records kernel launches for each decode-phase batch size into a CUDA Graph, which consumes extra GPU memory. In this case, each newly created green context for the decode phase adds memory usage for all recorded batch sizes. Due to the closed-source nature of CUDA, this cannot be addressed on our side. Fortunately, with just 4 groups of green contexts, Drift already outperforms all baselines.

Runtime Drift slices the prefill phase into multiple prefill blocks to enable adaptive gang scheduling. This may introduce extra overhead due to fine-grained kernel launches. We conduct an experiment to compare full prefill launching with block-wise launching, where the prefill phase is split into the finest granularity. Across various configurations with different batch sizes and context lengths, the total overhead remains within 1.5%.

6 Related Work

Compute partition for SLO attainment in LLM serving. To guarantee SLO, prior work adopts two types of compute partitioning solutions. The first is out-of-place partition, where the two phases are assigned to separate GPU instances to isolate interference. DistServe [45] and Splitwise [30] disaggregate LLM serving into distinct prefill/decode instances to meet the SLO target of their respective phases. LoongServe [38] further improves adaptability by enabling dynamic runtime switching between prefill and decode instances. The second approach, chunking-based methods [3], adopts in-place partitioning by splitting the long-running prefill phase into chunks, each coupled with a decode phase. The tight coupling forces the chunk size to align with the decode-phase SLO, limiting flexibility and peak utilization. Both approaches fall short of supporting efficient KV reuse while maintaining phase-aware SLO guarantees. In contrast, our work enable in-place and phase-decoupled partition within a single GPU through spatial multiplexing, enabling dynamic compute partitioning and migration-free KV reuse, achieving both reuse efficiency and SLO compliance.

Memory sharing for high throughput in LLM serving. In order to improve throughput in multi-turn or context-heavy LLM workloads, several systems adopt memory sharing as a primary design objective. PagedAttention [22] introduces a paged memory pool to reuse KV cache between prefill and decode phases while minimizing fragmentation. Parrot [24] and SGLang [44] leverage context-aware caching to maximize reuse of KV segments across requests. Drift enhances these approaches by preserving memory sharing across phases and requests. This allows existing reuse techniques to remain effective even under tight SLO constraints.

Compute partition techniques. Existing GPU resource partitioning techniques can be broadly categorized into time-sharing and space-sharing approaches. Time-sharing is typically implemented via API remoting [6, 25, 33**?**], which intercepts CUDA APIs to control kernel launch timing and allocate GPU time slices. However, time-sharing alone is insufficient to meet Drift's requirements, as the prefill and decode phases already interleave in a time-sharing manner. In contrast, NVIDIA provides several mechanisms for spatial sharing, including MPS [12], MIG [11], and GreenContext [8]. MPS and MIG support inter-process spatial multiplexing, while GreenContext [8] enables intra-process spatial multiplexing through precise SM partitioning [10]. Drift builds on GreenContext to implement its PD multiplexing approach.

Execution time modeling. Performance modeling under spatial sharing is highly challenging, and many prior efforts[21, 34, 40, 41] focus only on predicting interference for specific operators. GPUlet[7] uses linear regression with L1 cache utilization and DRAM bandwidth as input features to estimate performance interference among colocated operators. HSM[42] and GDP[20] also adopt linear regression based on low-level metrics in simulators to predict the slowdown of colocated GPU operators. Other works

leverage profiling to identify operator sets with low interference. Orin[34] classifies operators as compute- or memory-intensive and finds their spatial co-location generally acceptable. In parallel, a large body of work[6, 41] focuses on performance modeling for solo-run tasks or under time-sharing, which also typically rely on linear regression models.

7 Conclusion

Modern LLM services demand both high throughput and strict SLO guarantees, yet existing serving systems struggle due to gap between out-of-place compute partitioning and in-place memory sharing. Moreover, in-place compute partition fail to work efficiently due to phase-coupled design. To address these issues, we present Drift, a serving framework that enables adaptive compute partitioning and migration-free memory sharing simultaneously. Drift leverages intra-GPU spatial multiplexing to concurrently execute the prefill and decode phases, allowing a in-place and phase-decoupled design without sacrificing KV cache locality. Experiments show that Drift improves peak throughput by 5.1× on average over state-of-the-art baselines while consistently meeting SLO targets.

References

[1] Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, and et al. 2024. Phi-4 Technical Report. arXiv:2412.08905 (Dec. 2024). https://doi.org/10.48550/arXiv.2412.08905 arXiv:2412.08905 [cs]

[2] 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.

[3] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramjee. 2023. SARATHI: Efficient LLM Inference by Piggybacking Decodes with Chunked Prefills. arXiv:2308.16369 (Aug. 2023). arXiv:2308.16369 [cs]

[4] anon8231489123. 2023. ShareGPT Vicuna Unfiltered – Cleaned Split (v3). https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/main/ShareGPT_V3_unfiltered_cleaned_split.json. Accessed: 2025-04-16.

[5] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, and et al. 2023. Qwen Technical Report. arXiv:2309.16609 (Sept. 2023). https://doi.org/10.48550/arXiv.2309.16609 arXiv:2309.16609 [cs]

[6] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Atlanta Georgia USA, 681–696. https://doi.org/10.1145/2872362.2872368

[7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 199–216.

[8] NVIDIA Corporation. 2025. CUDA Driver API: Green Contexts. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__GREEN__CONTEXTS.html. Accessed: 2025-03-29.

[9] NVIDIA Corporation. 2025. CUDA Runtime API: Graph Management. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__GRAPH.html. Accessed: 2025-03-30.

[10] NVIDIA Corporation. 2025. CUDA Runtime API: Stream Management. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html. Accessed: 2025-03-30.

[11] NVIDIA Corporation. 2025. Multi-Instance GPU (MIG). https://www.nvidia.com/en-sg/technologies/multi-instance-gpu/. Accessed: 2025-03-30.

[12] NVIDIA Corporation. 2025. *Multi-Process Service*. Version 570.

[13] NVIDIA Corporation. 2025. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-sg/data-center/a100/. Accessed: 2025-04-16.

[14] NVIDIA Corporation. 2025. NVIDIA Dynamo: A Datacenter Scale Distributed Inference Serving Framework. https://github.com/ai-dynamo/dynamo. Accessed: 2025-04-07.

[15] NVIDIA Corporation. 2025. NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-sg/data-center/h100/. Accessed: 2025-04-16.

[16] NVIDIA Corporation. 2025. NVIDIA NVLink and NVSwitch: Fastest HPC Data Center Platform. https://www.nvidia.com/en-sg/data-center/nvlink/. Accessed: 2025-04-16.

[17] DeepSpeedAI. 2025. DeepSpeed-MII. https://github.com/deepspeedai/DeepSpeed-MII. Accessed: 2025-03-28.

[18] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, and et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 (Aug. 2024). arXiv:2407.21783

[19] Anysphere Inc. 2025. Cursor: The AI Code Editor. https://www.cursor.com/. Accessed: 2025-04-05.

[20] Magnus Jahre and Lieven Eeckhout. [n. d.]. Gdp: Using dataflow properties to accurately estimate interference-free performance at runtime. In *IEEE International Symposium on High Performance Computer Architecture (HPCA 2018)*. 296–309.

[21] Sejin Kim and Yoonhee Kim. 2022. K-Scheduler: Dynamic Intra-SM Multitasking Management with Execution Profiles on GPUs. *Cluster Computing* 25, 1 (Feb. 2022), 597–617. https://doi.org/10.1007/s10586-021-03429-7

[22] 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*. ACM, Koblenz Germany, 611–626. https://doi.org/10.1145/3600006.3613165

[23] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2024. LooGLE: Can Long-Context Language Models Understand Long Contexts? arXiv:2311.04939 [cs.CL] https://arxiv.org/abs/2311.04939

[24] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. 2024. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 929–945.

[25] Yu-Shiang Lin, Chun-Yuan Lin, Che-Rung Lee, and Yeh-Ching Chung. 2019. qcuda: Gpgpu virtualization for high bandwidth efficiency. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 95–102.

[26] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 881–897.

[27] NVIDIA. 2025. NVIDIA Inference Xfer Library (NIXL). https://github.com/ai-dynamo/nixl. Accessed: 2025-04-16.

[28] OpenAI. 2025. ChatGPT. https://chatgpt.com/. Accessed: 2025-04-05.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, and et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Number 721. Curran Associates Inc., Red Hook, NY, USA, 8026–8037.

[30] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient Generative LLM Inference Using Phase Splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 118–132. https://doi.org/10.1109/ISCA59077.2024.00019

[31] Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. 2024. Mutual Reasoning Makes Smaller LLMs Stronger Problem-Solvers. arXiv:2408.06195 (Aug. 2024). arXiv:2408.06195

[32] 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.

[33] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2011. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.* 61, 6 (2011), 804–816.

[34] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-Aware, Fine-Grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1075–1092. https://doi.org/10.1145/3627703.3629578

[35] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 (Feb. 2023). https://doi.org/10.48550/arXiv.2302.13971 arXiv:2302.13971

[36] Hugo Touvron, Louis Martin, and Kevin Stone. [n. d.]. Llama 2: Open Foundation and Fine-Tuned Chat Models. ([n. d.]).

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. In *Advances in Neural Information Processing Systems*. NeurIPS, Long Beach, CA, USA. https://doi.org/10.48550/arXiv.1706.03762 arXiv:1706.03762

[38] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. 2024. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 640–654. https://doi.org/10.1145/3694715.3695948

[39] 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.

[40] Shulai Zhang, Quan Chen, Weihao Cui, Han Zhao, Chunyu Xue, Zhen Zheng, Wei Lin, and Minyi Guo. 2025. Improving GPU Sharing Performance through Adaptive Bubbleless Spatial-Temporal Sharing. In *Proceedings of the Twentieth European Conference on Computer Systems (ACM Conferences)*. 573–588. https://doi.org/10.1145/3689031.3696070

[41] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. 2019. Laius: Towards Latency Awareness and Improved Utilization of Spatial Multitasking Accelerators in Datacenters. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. Association for Computing Machinery, New York, NY, USA, 58–68. https://doi.org/10.1145/3330345.3330351

[42] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. [n. d.]. HSM: A Hybrid Slowdown Model for Multitasking GPUs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. 1371–1385.

[43] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[44] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2023. Efficiently Programming Large Language Models Using SGLang. arXiv:2312.07104 (Dec. 2023). arXiv:2312.07104 [cs]

[45] 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)*. 193–210.

[46] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Stephanie Wang, Arvind Krishnamurthy, Baris Kasikci, and et al. [n. d.]. NanoFlow: Towards Optimal Large Language Model Serving Throughput. ([n. d.]).