arXiv:2504.14489v3 [cs.OS] 7 Feb 2026

Towards High-Goodput LLM Serving with Prefill-decode Multiplexing

Yukang Chen*
chenyukang@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Weihao Cui*
weihao@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China
National University of Singapore
Singapore

Han Zhao*
zhao-han@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Ziyi Xu
xzy2022@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Xiaoze Fan
jasonfxz@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Xusheng Chen
michael.xschen@gmail.com
Researcher
Shanghai, China

Yangjie Zhou
yj_zhou@nus.edu.sg
National University of Singapore
Singapore

Shixuan Sun
sunshixuan@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Bingsheng He
dcsheb@nus.edu.sg
National University of Singapore
Singapore

Quan Chen†
chen-quan@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Abstract

Large Language Model (LLM) serving must meet stringent Service Level Objectives (SLOs) for both the prefill and decode phases. Some existing solutions disaggregate the two phases, causing potential resource idleness or compute redundancy. Others split the prefill phase into chunks and fuse it with decode iteration, creating a dilemma between SLO compliance and high utilization. To address these issues, an efficient serving system should dynamically adapt compute allocation, decouple compute from memory management, and execute prefill and decode independently. We present MuxWise, an LLM serving framework that adopts a new paradigm, intra-GPU prefill-decode multiplexing, to meet these requirements. To fully exploit the paradigm, MuxWise integrates a bubble-less multiplex engine, a contention-tolerant estimator, and an SLO-aware dispatcher. Evaluation shows that MuxWise improves peak throughput under SLO guarantees by an average of 2.20× (up to 3.06×) over state-of-the-art baselines.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**; **Cloud computing**; • **Software and its engineering → Process management**.

Keywords: LLM Serving, PD-Multiplexing, Goodput

ACM Reference Format:
Yukang Chen, Weihao Cui, Han Zhao, Ziyi Xu, Xiaoze Fan, Xusheng Chen, Yangjie Zhou, Shixuan Sun, Bingsheng He, and Quan Chen. 2026. Towards High-Goodput LLM Serving with Prefill-decode Multiplexing. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3779212.3790236

*Equal contribution.
†Corresponding author.

This work is licensed under a Creative Commons Attribution 4.0 International License.
ASPLOS '26, Pittsburgh, PA, USA
© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790236

1 Introduction

Large language models (LLM) services now perform well across diverse workloads [19, 30, 33]. At the request level, an LLM processes input in two phases: a prefill phase that produces the first token, followed by a decode phase that iteratively generates the remaining tokens. The ratio of input length (prefill) to output length (decode) varies across tasks [6, 43]. At the application level, tasks such as chatbot services or agent-based workloads [34] often consist of multiple turns of requests with shared context.

本文档由 funstory.ai 的开源 PDF 翻译库 BabelDOC v0.6.2 (https://github.com/funstory-ai/BabelDOC) 翻译，本仓库正在积极的建设当中，欢迎 star 和关注。

arXiv:2504.14489v3 [cs.OS] 7 Feb 2026

面向高吞吐量大语言模型服务与预填充解码多路复用

陈宇康*
chenyukang@sjtu.edu.cn 上海交通大学 上海，中国

崔伟豪*
weihao@sjtu.edu.cn 上海交通大学 上海，中国 新加坡国立大学 新加坡

赵汉*
zhao-han@cs.sjtu.edu.cn 上海交通大学 上海，中国

徐子怡
xzy2022@sjtu.edu.cn
上海交通大学 上海，中国

范晓泽
jasonfxz@sjtu.edu.cn
上海交通大学 上海，中国

陈旭生
michael.xschen@gmail.com 研究员 上海，中国

周阳杰
yj_zhou@nus.edu.sg 新加坡国立大学 新加坡

孙世玄
sunshixuan@sjtu.edu.cn
上海交通大学 上海，中国

何冰生
dcsheb@nus.edu.sg 新加坡国立大学 新加坡

全晨†
chen-quan@cs.sjtu.edu.cn 上海交通大学 上海，中国

摘要

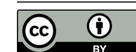
大语言模型（LLM）服务必须满足预填充和解码阶段的服务等级目标（SLO）。一些现有解决方案将这两个阶段分离，导致潜在的资源闲置或计算冗余。其他人将预填充阶段分成块并与解码迭代融合，在满足SLO和保持高利用率之间产生困境。为了解决这些问题，高效的系统应该动态调整计算分配，将计算与内存管理解耦，并独立执行预填充和解码。我们提出了MuxWise，一个采用新范式——GPU内部预填充-解码多路复用的大语言模型服务框架，以满足这些要求。为了充分利用这些范式，MuxWise集成了一个无气泡多路复用引擎、一个抗争容忍估计器和一个SLO感知调度器。评估显示研究表明 MuxWise 在 SLO 保证下，相较于现有最佳基线平均提升了 2.20× (最高可达 3.06×) 的峰值吞吐量。

CCS 概念： •**计算机系统组织 →单指令多数据**; **云计算**; • **软件及其工程 →进程管理**。

关键词： LLM 服务，PD-多路复用，吞吐量

ACM参考格式:
陈宇康，崔伟豪，赵汉，徐子奕，范晓泽，陈旭生，周阳杰，孙世玄，何冰生，及陈全。2026。面向高吞吐量大语言模型预填充-解码多路复用服务。发表于第*31*届*ACM*国际编程语言与操作系统架构研讨会论文集（*ASPLOS '26*），第二卷，*2026*年*3*月*22*日至*26*日，美国宾夕法尼亚州匹兹堡，宾夕法尼亚州，美国。ACM，纽约，纽约，美国，18 页。https://doi.org/10.1145/3779212.3790236

*贡献均等。
†通讯作者。



本作品根据知识共享署名 4.0 国际许可协议授权。*ASPLOS '26,* 匹兹堡，宾夕法尼亚州，美国©2026 版权所有者/作者。

ACM ISBN 979-8-4007-2359-9/2026/03https://doi.org/10.1145/3779212.3790236

1 引言

大语言模型（LLM）服务现在在各种工作负载上表现良好 [19, 30, 33]。在请求级别，LLM 处理输入分为两个阶段：预填充阶段生成第一个标记，然后是解码阶段迭代生成剩余标记。输入长度（预填充）与输出长度（解码）的比率因任务而异 [6, 43]。在应用级别，聊天机器人服务或基于代理的工作负载等任务通常由具有共享上下文的多轮请求组成。 [34] 。

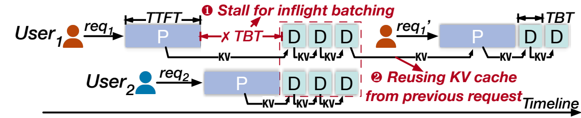


Figure 1. A typical workflow in LLM serving systems: `User1` sends two consecutive requests, with the second reusing the context from the first. `User2` sends 1 request.

To achieve high throughput for serving these workloads, existing LLM serving systems employ several optimizations. Figure 1 presents a typical workflow. While requests arrive at different times, inflight batching stalls the ongoing decode phase to prefill new requests and then processes all decode iterations together in a single batch. It greatly improves compute utilization for the memory-intensive decode phase [47]. Since multi-turn requests share context, LLM serving systems reuse intermediate results (i.e., the KV cache) both within and across requests through a KV cache pool [26, 52].

LLM services also impose stringent Service Level Objectives (SLOs). For instance, chatbot typically requires Time-To-First-Token (TTFT) under 500 ms for prefill and Time-Between-Tokens (TBT) under 100 ms for decode. Since prefill and decode interleave in an LLM serving system, SLO violations may arise. In Figure 1, inflight batching stalls ongoing decode. A long prefill can thus delay decode, potentially violating its SLO. To sustain high goodput–peak throughput with SLO guarantees–existing methods fall into two categories: disaggregated serving [32, 53] and chunked prefill [1].

As for disaggregated serving, Splitwise [32] separates the prefill and decode phases into distinct instances for SLO guarantees, which has two drawbacks. Firstly, it cannot adapt to serving dynamics. In Splitwise, GPUs are statically allocated at initialization. Under fluctuating request loads and diverse serving patterns, it often leads to resource underutilization. Secondly, it decreases goodput due to shrinking the KV cache pool. With the same number of GPUs, disaggregation allocates a separate cache pool for each instance, reducing the effective cache pool size. This lowers cache hit rate [45] (e.g., from 36.6% to 4.2%), leading to unnecessary recomputation and degraded goodput. Furthermore, while LoongServe [44] supports dynamic GPU allocation based on the request sequence length and execution phase, it cannot support the cross-request KV cache reuse, incurring significant recomputation overhead in multi-turn workloads.

Chunked-prefill [1] is another approach to meet decode SLOs. It splits the prefill phase into chunks within each GPU and fuses each chunk with a decode iteration. To ensure computational equivalence, each chunk reads the KV cache generated by all previous chunks. It ensures the decode SLOs by capping the token budget, defined as the sum of new tokens from the prefill chunk and the decode batch. By tuning the chunk and decode batch sizes, it adapts to serving dynamics. Since it avoids disaggregation, it also prevents goodput loss from a reduced KV cache pool.

Unfortunately, chunking is not a free lunch. It creates a dilemma between SLO compliance and high utilization. Because prefill chunk and decode iteration must execute together, the token budget governs both decode SLO attainment and GPU saturation. Yet, finding a sweet budget in practice is infeasible. E.g., deploying a 70B LLM on 8 A100 GPUs requires a 4K budget to saturate the GPU, which is 8× larger than the SLO-compliant budget (256 for a 100 ms TBT SLO). Moreover, TBT in chunk-prefill is inflated by repetitive KV cache access from the prefill chunk. With extremely long reused context, common in multi-turn workloads, chunked-prefill may even fail to meet SLO guarantees (§2.3.2). Ultimately, chunked-prefill cannot sustain high goodput.

Achieving high-goodput LLM serving requires more flexible compute management. We propose intra-GPU prefill-decode (PD) multiplexing as a promising new serving paradigm. Specifically, the prefill and decode phases are executed on different streaming multiprocessors (SMs) within the GPUs. In the new paradigm, 1) compute partitions can be reconfigured with low overhead to adapt to serving dynamics; 2) multiplexed phases share GPU memory, keeping the KV cache pool efficient; 3) with spatial sharing, prefill and decode execute independently, avoiding the tradeoff between SLO compliance and utilization.

Realizing this paradigm is non-trivial. Firstly, phase coordination is still required to enable inflight batching and improve compute utilization. However, since prefill and decode latencies differ significantly, naive coordination often leaves GPU bubbles. Secondly, spatial multiplexing introduces unpredictable contention. Although existing approaches [10, 12, 13] partition compute, they provide little control over shared resources such as memory bandwidth.

To this end, we propose MuxWise, an LLM serving framework that achieves high goodput across diverse workloads. MuxWise comprises three modules: a *bubble-less multiplex engine*, a *contention-tolerant estimator*, and an *SLO-aware dispatcher*. The engine partitions prefill into layers with negligible overhead, aligning execution latencies for bubble-less multiplexing. The contention-tolerant estimator provides worst-case latency predictions by combining a solo-run predictor with a contention guard derived from one-time offline profiling. Built atop the engine and estimator, the SLO-aware dispatcher schedules diverse LLM requests efficiently by selecting multiplexing plans to maximize goodput.

We implement MuxWise on top of SGLang [52], extending it with PD multiplexing. MuxWise is evaluated extensively on both small and large LLMs using real-world workloads. Experiments show that MuxWise achieves an average 2.20× goodput improvement (up to 3.06×) over state-of-the-art solutions. In summary, our contributions are:

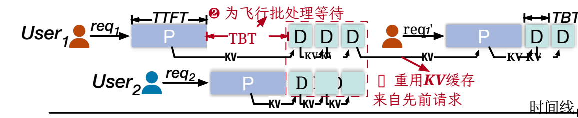


图1。 典型的LLM服务系统工作流：用户1发送两个连续的请求，第二个请求重用第一个请求的上下文。用户2发送1个请求。

为了实现这些工作负载的高吞吐量服务，现有的LLM服务系统采用多种优化。图1展示了一个典型的工作流程。虽然请求在不同时间到达，但飞行批处理会暂停正在进行的解码阶段以预填充新请求，然后一起处理所有解码迭代在一个批次中。这大大提高了内存密集型解码阶段的计算利用率 [47]。由于多轮请求共享上下文，LLM服务系统通过KV缓存池在请求内部和跨请求中重用中间结果（即KV缓存） [26, 52]。

LLM服务也对服务等级目标（SLO）提出了严格要求。例如，聊天机器人通常要求预填充的首次token时间（TTFT）低于500毫秒，解码的token间隔时间（TBT）低于100毫秒。由于预填充和解码在LLM服务系统中交替进行，可能会出现SLO违规。在图1中，飞行批处理会阻塞正在进行的解码。因此，长时间的预填充会延迟解码，可能违反其SLO。为了在保证SLO的前提下实现高吞吐量-峰值吞吐量，现有方法分为两类：解耦服务 [32, 53] 和分块预填充 [1]。

关于解耦服务，Splitwise [32] 将预填充和解码阶段分离为不同的实例以保证SLO，这有两个缺点。首先，它无法适应服务动态。在Splitwise中，GPU在初始化时静态分配。在波动的工作负载和服务模式下，这通常导致资源利用率低下。其次，由于KV缓存池缩小，吞吐量降低。使用相同数量的GPU，解耦为每个实例分配一个独立的缓存池，导致有效缓存池大小减小。这降低了缓存命中率 [45] (例如，从36.6%降至4.2%)，导致不必要的重新计算和吞吐量下降。此外，虽然LoongServe [44]支持根据请求序列长度和执行阶段动态分配GPU，但它不支持跨请求的KV缓存重用，在多轮工作负载中会带来显著的重新计算开销。

分块预填充 [1] 是另一种满足解码SLO的方法。它将预填充阶段在每个GPU内分块，并将每个块与一次解码迭代融合。为确保计算等价性，每个块读取所有先前块生成的KV缓存。通过限制令牌预算——定义为预填充块的新令牌和解码批次的和——来确保解码SLO。通过调整块和解码批大小，它能适应服务动态。由于它避免了解聚合，因此也防止了因KV缓存池减少而导致的吞吐量损失。

不幸的是，分块并非免费的午餐。它在SLO合规性和高利用率之间创造了困境。由于预填充块和解码迭代必须一起执行，令牌预算决定了解码SLO的达成和GPU饱和。然而，在实践中找到一个理想预算是不可行的。例如，在8个A100 GPU上部署70B大语言模型需要4K预算才能使GPU饱和，这 8×大于合规性预算（对于100 ms TBT SLO为256）。此外，分块预填充中的TBT因预填充块重复的KV缓存访问而膨胀。对于多轮工作负载中常见的极长重用上下文，分块预填充甚至可能无法满足SLO保证（§2.3.2）。最终，分块预填充无法维持高吞吐量。

实现高吞吐量大语言模型服务需要更灵活的计算管理。我们提出了GPU内部预填充-解码（PD）多路复用作为一种有前景的新服务范式。具体来说，预填充和解码阶段在不同的GPU内的流多处理器（SMs）上执行。在新范式下，1）计算分区可以以低开销重新配置以适应服务动态；2）多路复用阶段共享GPU内存，保持KV缓存池高效；3）通过空间共享，预填充和解码独立执行，避免了SLO合规性和利用率之间的权衡。

实现这一范式并不简单。首先，为了实现飞行批处理并提高计算利用率，仍然需要阶段协调。然而，由于预填充和解码延迟差异很大，简单的协调经常导致GPU气泡。其次，空间多路复用引入了不可预测的容争。尽管现有方法 [10, 12, 13] 对计算进行分区，但它们对内存带宽等共享资源几乎没有控制力。

为此，我们提出了MuxWise，一个能够在不同工作负载下实现高吞吐量的LLM服务框架。MuxWise包含三个模块：一个无气泡多路复用引擎、一个抗争容忍估计器和一个*SLO*感知调度器。该引擎将预填充分层，开销极小，使执行延迟对齐以实现无气泡多路复用。抗争容忍估计器通过结合一个Solo-run预测器和一个来自一次性离线分析的抗争保护，提供最坏情况延迟预测。建立在引擎和估计器之上的SLO感知调度器通过选择多路复用计划以最大化吞吐量，高效地调度各种LLM请求。

我们在SGLang [52], 的基础上实现了MuxWise，并使用PD多路复用对其进行扩展。MuxWise在真实工作负载下对小型和大型大语言模型进行了广泛评估。实验表明，与最先进解决方案相比，MuxWise实现了平均 2.20×吞吐量提升（最高可达 3.06×）。总之，我们的贡献包括：

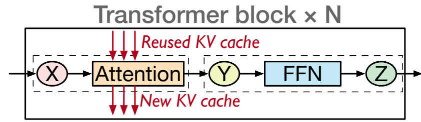


Figure 2. Main architecture of most LLMs.

Table 1. Diverse patterns of typical LLM tasks. Minimum, mean, and maximum values for each metric are reported. The input length includes the length of new and reused context.

	Input length	Output length	Reused length
ShareGPT [4]	4/226/1024	4/195/1838	\
LooGLE [25]	3380/30k/81k	2/15/326	\
OpenThoughts [17]	311/709/4633	684/8374/32k	243
Conversation [34]	891/7538/123k	1/342/2000	0/4496/120k
Tool&agent [34]	891/8596/123k	1/182/2000	0/4905/120k

- We identify key requirements for LLM serving with high goodput through a detailed analysis of prior works.
- We propose a new LLM serving paradigm–PD multiplexing–aligned with these requirements, and present a clean design to effectively serve LLMs with high goodput.
- We evaluate MuxWise under diverse workloads, demonstrating its superiority over state-of-the-art solutions.

2 Background & Motivation

2.1 LLM Services

Architecture of LLMs. Most LLMs [5, 15, 37, 38] are built upon the transformer architecture [39], with model-specific modifications. Figure 2 illustrates a typical transformer layer, which is replicated multiple times to form an LLM model. Each transformer layer contains an attention layer and a feed-forward network (FFN) layer.

Attention computation requires access to all keys and values from processed tokens and also generates the keys and values of new tokens. To avoid redundant computation, LLM serving systems store this data in a KV cache. In the prefill phase, the KV cache is populated from the requests in previous turns. In each decode iteration, the KV cache is derived from earlier prefill and decode iterations.

Diverse workload patterns. Table 1 illustrates the diverse patterns of five typical LLM tasks. The first three are single-turn requests: ShareGPT [4] is a chatbot task, LooGLE [25] is a long-context understanding task, and OpenThoughts [17] is a reasoning task. LooGLE has a long input length due to long documents. Reasoning often requires long thought processes, so OpenThoughts tends to have a longer output length than others. Requests in OpenThoughts share the same system prompt, which is a constant input context (i.e., reused length in the table). Conversion and Tool&agent [34] are two real-world multi-turn tasks. The output tokens from earlier requests become the input context for later requests

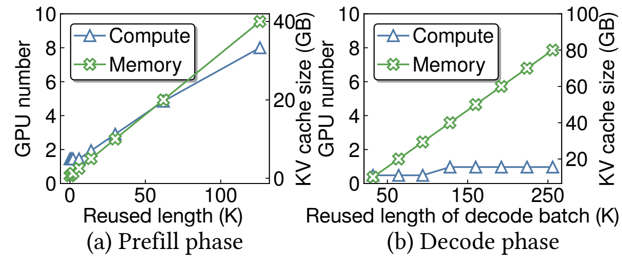


Figure 3. Required compute and memory for processing different phases under SLO constraints with varied reused context lengths. For prefill (a), the batch size is fixed at 1, the new context length is set to $2K$, and TTFT is set to $400ms$. For decode (b), the batch size is fixed at 32, and TBT is set to $100ms$. These settings are commonly seen in online serving.

in the same session. We use these workloads to conduct experiments that both motivate and evaluate our design.

2.2 Characterization under SLO constraint

Many prior works [32, 44, 53] have investigated the relationship between resource requirements and SLO attainment concerning input length and batch size. Their experiments show that the prefill phase is compute-intensive, with compute demand growing linearly with input length, while the decode phase is memory-intensive. However, they mainly focus on the simple single-turn case, which does not consider the effect of reused input length.

Under these circumstances, we further study how the reused length impacts the compute and memory demands of prefill and decode. In our experiment, the reused length spans the range shown in Table 1, and LlaMA-70B [15] is deployed with tensor parallelism [51] on a server with 8 A100 GPUs. All GPUs are configured with the same partial compute resource, defined by the SM number. For each reused length, we determine the best-fit GPU partition ratio (denoted as GPU_{ratio}) to satisfy the SLO target. Figure 3 reports the total compute demand of LlaMA-70B under different reused lengths, computed as $GPU_{num} = GPU_{ratio} \times 8$.

As shown in Figure 3-(a), prefill phase requires increasingly more compute resources to meet SLO targets as the reused length grows. In contrast, the compute demand of the decode phase shows less sensitivity. Thus, it is also critical to allocate more compute to the prefill phase as the reused length increases. Further, the distinct compute requirements of two phases necessitate a runtime compute resource partition for SLO attainment and high utilization.

Figure 3-(b) shows that the KV cache required by both the prefill and decode easily reaches tens or even hundreds of gigabytes. This is common in multi-turn LLM services, which produce ultra-long reused contexts. It is preferable to keep the KV cache in the same memory space (aggregated serving) for efficient reuse across phases and requests.

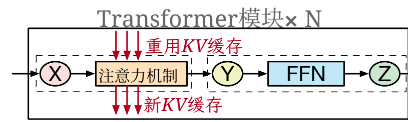


图2。 大多数大语言模型的主架构。

表1。 典型大语言模型任务的多样化模式。报告了每个指标的最小值、平均值和最大值。输入长度包括新和重用上下文的长度。

	输入长度	输出长度	重用长度
ShareGPT [4]	4/226/1024	4/195/1838	\
LooGLE [25]	3380/30k/81k	2/15/326	\
OpenThoughts[17]	311/709/4633	684/8374/32k	243
对话 [34]	891/7538/123k	1/342/2000	0/4496/120k
工具与代理 [34]	891/8596/123k	1/182/2000	0/4905/120k

- 我们通过详细分析现有工作，确定了高吞吐量LLM服务的关键需求。
- 我们提出了一种新的LLM服务范式——PD多路复用——以满足这些要求，并呈现了一种简洁的设计，以高效地服务LLM并实现高吞吐量。
- 我们在多样化的工作负载下评估了MuxWise，证明了其优于最先进解决方案。

2 背景 & 动机

2.1 LLM服务

***LLM*的架构。** 大多数LLM [5, 15, 37, 38] 都是基于Transformer架构 [39], 并进行了模型特定的修改。 图2 展示了一个典型的Transformer层，该层被复制多次以形成LLM模型。每个Transformer层包含一个注意力机制层和一个前馈网络（FFN）层。

注意力机制计算需要访问所有已处理token的键和值，并且还会生成新token的键和值。为了避免冗余计算，LLM服务系统将此数据存储在KV缓存中。在预填充阶段，KV缓存从先前回合的请求中填充。在每次解码迭代中，KV缓存是从更早的预填充和解码迭代中派生的。

多样化的工作负载模式。 表1 展示了五种典型LLM任务的多样化模式。前三种是单轮请求：ShareGPT [4] 是一个聊天机器人任务，LooGLE [25]是一个长上下文理解任务，而OpenThoughts [17]是一个推理任务。LooGLE由于长文档输入长度较长。推理通常需要较长的思考过程，因此OpenThoughts的输出长度往往比其他任务更长。OpenThoughts中的请求共享相同的系统提示，这是一个固定的输入上下文（即表格中的重用长度）。转换和Tool&Agent [34]是两个现实世界的多轮任务。早期请求的输出token成为后期请求的输入上下文

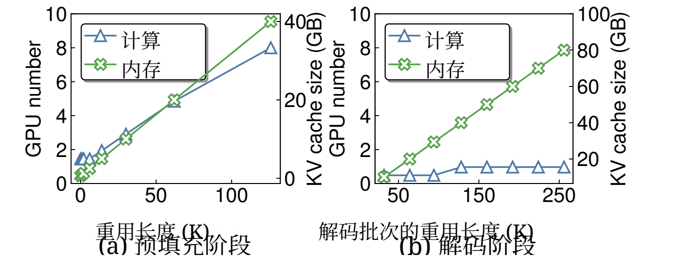


图3。 在SLO约束下，不同重用上下文长度下处理不同阶段所需的计算和内存。对于预填充 (a)，批次大小固定为1，新上下文长度设置为 $2K$，TTFT设置为 $400ms$。对于解码 (b)，批次大小固定为32，TBT设置为$100ms$。这些设置在在线服务中很常见。

在同一个会话中。我们使用这些工作负载进行实验，这些实验既激励又评估了我们的设计。

2.2 在SLO约束下的特性分析

许多先前工作 [32, 44, 53] 已经研究了资源需求与SLO达成之间的关系，这涉及到输入长度和批次大小。他们的实验表明，预填充阶段是计算密集型的，计算需求随着输入长度线性增长，而解码阶段是内存密集型的。然而，他们主要关注简单的单轮情况，这没有考虑重用输入长度的影响。

在这种情况下，我们进一步研究了重用长度如何影响预填充和解码的计算与内存需求。在我们的实验中，重用长度跨越了表1中所示的范围，并且LlaMA-70B被部署在具有8个A100 GPU的服务器上，采用张量并行计算 [51] 。所有GPU都配置了相同的部分计算资源，该资源由SM编号定义。对于每个重用长度，我们确定了最佳匹配的GPU分区比例（记为 GPU_{ratio}），以满足SLO目标。图3报告了LlaMA-70B在不同重用长度下的总计算需求，该需求计算为 $GPU_{num} = GPU_{ratio} \times 8$。

如图3-(a)所示，随着重用长度的增长，预填充阶段需要越来越多的计算资源来满足SLO目标。相比之下，解码阶段的计算需求敏感性较低。因此，随着重用长度的增加，也需要向预填充阶段分配更多的计算资源。此外，两个阶段不同的计算需求需要为SLO达成和高利用率而进行运行时计算资源分区。

图3-(b)显示，预填充和解码所需的KV缓存很容易达到数十GB甚至数百GB。这在多轮LLM服务中很常见，这些服务会产生超长重用上下文。为了跨阶段和请求高效重用，最好将KV缓存保持在同一内存空间（聚合服务）中。

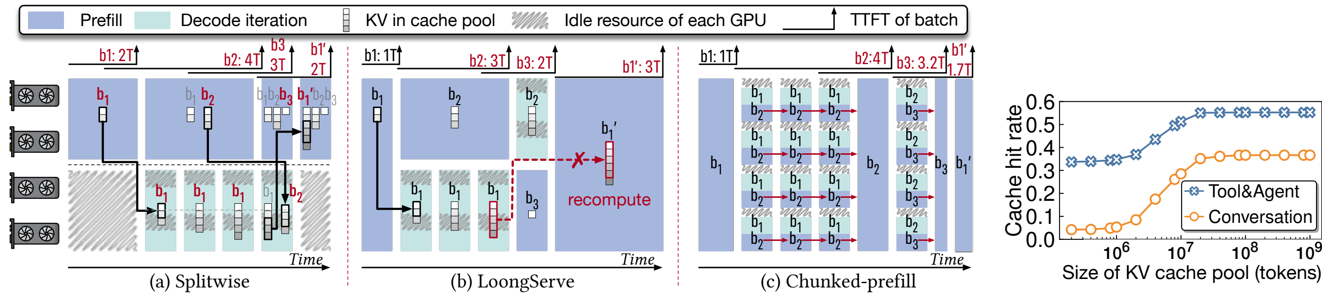


Figure 4. Processing four LLM request batches on 4 GPUs using (a) Splitwise, (b) LoongServe (c) chunked-prefill. All methods satisfy the TBT SLO (T per decode iteration). Specifically, b_1 arrives at $0T$, b_2 at $1T$, b_3 at $3T$, and b_1' at $5T$. b_1' denotes a subsequent request batch that reuses the KV cache of b_1. Inefficient TTFTs are marked in red for each method. KV cache management is shown only for the two disaggregated methods, as they require migration or recomputation. In (a) and (b), solid black arrows represent migration, while dashed red arrows with cross markers denote recomputation. In (a), the KV cache column with a red b_i indicates the active batch. In (c), the red arrow denotes KV cache reads from earlier chunks.

Figure 5. Cache hit rates under varying capacities of the KV cache pool. The eviction policy is Least Recently Used. For serving a 70B LLM, achieving the optimal hit rate requires 3.3 TB of memory. Workload trace details are shown in Table 1.

In a nutshell, we make two observations: *1) Appropriate and dynamic compute partition is essential for meeting the distinct SLO targets of different phases under diverse workloads. 2) Reusing the KV cache across phases and requests is critical for reducing redundant computation and improving goodput.*

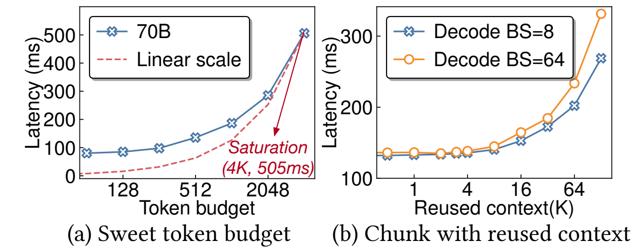
2.3 Deficiencies of Existing Works

2.3.1 Disaggregated Serving. Disaggregating approaches partition GPUs across phases to meet the SLO targets in LLM serving and can be further divided into static and dynamic disaggregation methods. Figure 4-(a) illustrates the static approach (Splitwise [32]), while Figure 4-(b) shows the dynamic approach (LoongServe [44]).

Static disaggregation. As shown in Figure 4-(a), there is a prefill instance and a decode instance with Splitwise [32]. Each instance occupies two GPUs statically and has its own KV cache pool. The GPU number is static after the instance is initialized. In this case, Splitwise suffers from two problems.

First, Splitwise does not adapt to serving dynamics. For example, when batch b1 arrives, only two GPUs process the prefill while the other two GPUs for decoding remain idle. In online serving, such idle periods are common as request loads fluctuate. *Second, the coupled management of compute and memory introduces further inefficiencies.* For instance, if the decode phase of b1 in Figure 4-(a) requires two GPUs to store the KV cache, the system must also allocate two GPUs for computation. Since compute and memory requirements are misaligned, as shown in Figure 3-(b), the GPUs' compute resources may be underutilized.

In addition, each instance must maintain its own model weights and KV cache pool. As a result, the KV cache pool in Figure 4-(a) is at most half the size of that with four GPUs under non-disaggregated execution. Furthermore, experimental results in Figure 5 show that this reduced capacity



(a) Sweet token budget (b) Chunk with reused context

Figure 6. (a) Sweet spot of the token budget in chunk-prefill. The decode uses a fixed batch size of 32, with each request having a reused context length of 1K tokens. (b) Latencies with varied reused context of the fused prefill chunk in chunk-prefill. The token budget is fixed at 512, and the reused context length of decode phase is the same as in (a).

sharply lowers the KV cache hit rate in multi-turn workloads, ultimately degrading the system's goodput.

Dynamic disaggregation. LoongServe [44] supports dynamic GPU partitioning across the two phases. Specifically, it scales GPU resources based on the sequence length and execution phase. As shown in Figure 4-(b), when batch b1 arrives, the scheduler assigns four GPUs to prefill. After prefill, it scales down to two GPUs for the decode iterations.

However, LoongServe still causes idleness due to coupled management, and worse, it trades KV cache reuse for adaptiveness needed in serving dynamics. To avoid duplication, it immediately releases the KV cache on original GPUs. Thus, KV caches are reused only from prefill to decode within a single request and cannot be reused across multi-turn requests. In Figure 4-(b), when b1' needs to reuse the KV cache generated by b1, LoongServe recomputes the entire KV cache.

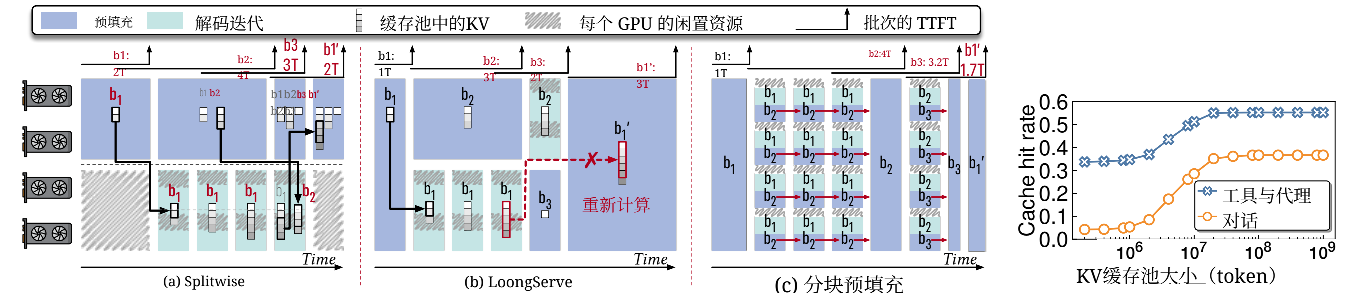


图4。 在4个GPU上处理四个LLM请求批次，使用(a)Splitwise，(b)LoongServe (c)分块预填充。所有方法都满足TBT SLO（每个解码迭代T）。具体来说，b_1达到0T，b_2达到1T，b_3达到3T，以及b_1'达到5T。b_1'表示一个重用b_1的KV缓存的后续请求批次。每种方法中低效的TTFT用红色标记。仅对两个解耦方法显示KV缓存管理，因为它们需要迁移或重新计算。在(a)和(b)中，实心黑箭头表示迁移，而带叉号的红色虚线箭头表示重新计算。在(a)中，带有红色b_i的KV缓存列表示活动批次。在(c)中，红色箭头表示从早期块读取的KV缓存。

图5。 KV缓存池不同容量下的缓存命中率。驱逐策略为最近最少使用。为服务70B大语言模型，实现最佳命中率需要3.3TB内存。工作负载跟踪细节显示在表1。

简而言之，我们有两个观察结果：*1)* 适当且动态的计算分区对于在不同负载下满足不同阶段的不同*SLO*目标至关重要。*2)* 跨阶段和请求重用*KV*缓存对于减少冗余计算和提高吞吐量至关重要。

2.3 现有工作的缺陷

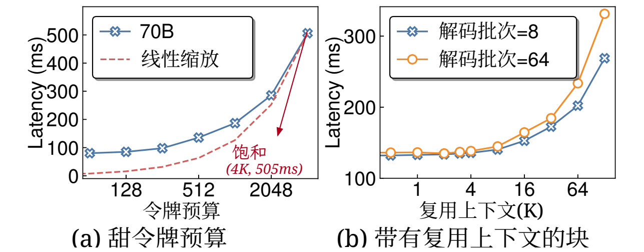
2.3.1 分离式服务。 分离式方法将GPU分配到不同阶段以满足LLM服务的SLO目标，并可进一步分为静态和动态分离方法。图4-(a)展示了静态方法（Splitwise [32]），而图4-(b)显示了动态方法（LoongServe [44]）。

静态分离。如图图4-(a)所示，存在预填充实例和解码实例，使用Splitwise [32]。每个实例静态占用两个GPU，并拥有自己的KV缓存池。实例初始化后GPU数量固定。在这种情况下，Splitwise存在两个问题。

首先，*Splitwise*无法适应服务动态。例如，当批次b1到达时，只有两个GPU处理预填充，而另外两个用于解码的GPU则处于空闲状态。在在线服务中，由于请求负载波动，这种空闲时段很常见。其次，计算和内存的耦合管理引入了进一步的低效。例如，如果b1 在图4-(a)中的解码阶段需要两个GPU来存储KV缓存，系统也必须分配两个GPU

用于计算。由于计算和内存需求不匹配，如图3-(b)所示，GPU的计算资源可能未被充分利用。

此外，每个实例必须维护自己的模型权重和KV缓存池。因此，图4-(a)中的KV缓存池最大仅为非解耦执行时四个GPU的缓存池大小的一半。此外，图5所示实验结果表明，这种容量减少会显著降低多轮工作负载中的KV缓存命中率，最终降低系统的吞吐量。图4-(a)图5显示，这种容量减少



(a) 甜令牌预算 (b) 带有复用上下文的块

图6。 (a) 分块预填充中令牌预算的甜点。解码使用固定32的批次大小，每个请求具有1K的复用上下文长度。(b) 分块预填充中融合预填充块的不同复用上下文延迟。令牌预算固定为512，解码阶段的复用上下文长度与(a)相同。

显著降低了多轮工作负载中的KV缓存命中率，最终降低了系统的吞吐量。

动态解聚合。LoongServe [44] 支持在两个阶段之间动态进行GPU分区。具体来说，它会根据序列长度和执行阶段来扩展GPU资源。如图 4-(b) 所示，当批次b1到达时，调度器会分配四个GPU进行预填充。预填充后，它会缩减到两个GPU进行解码迭代。

然而，LoongServe 由于耦合管理仍会导致空闲，更糟糕的是，它以牺牲 KV 缓存重用来换取服务动态中所需的适应性。为了避免重复，它在原始 GPU 上立即释放 KV 缓存。因此，KV 缓存仅在单个请求的预填充到解码期间被重用，而无法跨多轮请求重用。在 图 4-(b) 中，当 b1'需要重用由b1 生成的 KV 缓存时，LoongServe 重新计算整个 KV 缓存。

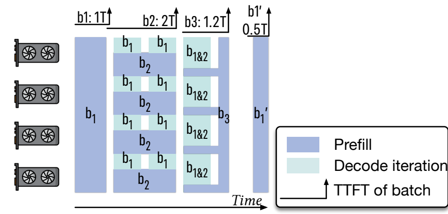


Figure 7. An ideal solution: prefill-decode multiplexing.

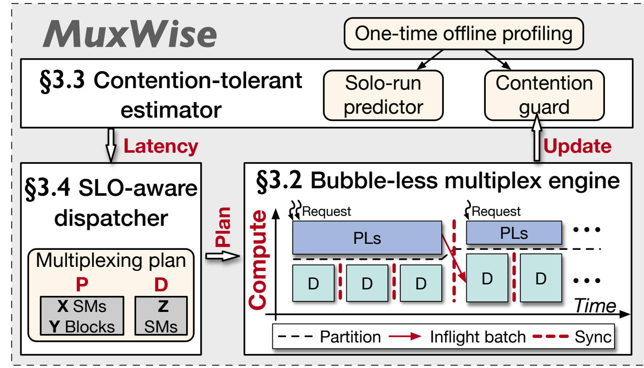


Figure 8. Architecture overview of MuxWise.

2.3.2 Chunked-prefill.

Chunked-prefill [1] adopts intra-GPU compute fusion. As shown in Figure 4-(c), it splits prefill into chunks and fuses each chunk with a decode iteration. To guarantee decode SLOs, chunked-prefill caps the token budget, which is the sum of new tokens from the prefill chunk and the decode batch. While chunked-prefill has known drawbacks such as quadratic memory overhead [53], we find another drawback. Specifically, chunking introduces a dilemma between SLO attainment and utilization.

Figure 6-(a) presents TBT in Chunked-prefill of varying token budget. In this experiment, the decode iteration for fusion has a static batch size of 32 and a reused context length of 1K tokens, and Llama3-70B is deployed on a server with 8 A100 GPUs. As shown, the latency does not increase linearly with the token budget until it reaches $4K$. This indicates that saturating the GPUs requires a prefill chunk with input length of $(4K - 32)$. However, the corresponding latency is $505ms$, far above the typical TBT SLO target ($< 100ms$).

Figure 6-(b) presents TBT in Chunked-prefill with varying reused context lengths of the prefill. In this experiment, the token budget is fixed at 512, and the reused context length of decode iteration is 1K. As shown, TBT increases noticeably after the reused context exceeds $4K$. This reused context length is common in long-context understanding and multi-turn workloads, as shown in Table 1. In such cases, Chunked-prefill easily leads to SLO violations.

2.4 New Paradigm & Challenges

As shown in Figure 7, we propose an intra-GPU prefill-decode (PD) multiplexing paradigm to overcome the above limitations. Specifically, prefill and decode dynamically share the compute resources (SMs) within each GPU. By reserving sufficient SMs to satisfy decode SLOs and assigning the remaining SMs to prefill, high-goodput LLM serving is achieved. PD multiplexing overcomes the limitations of prior methods, benefiting from the following abilities.

First, multiplexing enables dynamic and adaptive compute management. As shown, compute resources can be flexibly allocated between the two phases to maximize system goodput while guaranteeing SLOs. Second, multiplexing decouples compute from memory management. Although the two phases partition compute resources, they share the memory space on each GPU, enabling efficient KV cache reuse. Third, multiplexing allows prefill and decode to run independently without stalling one another, avoiding the dilemma between SLO attainment and system goodput.

However, integrating intra-GPU multiplexing into existing LLM serving systems is non-trivial. There are two challenges to realizing this paradigm. **C-1: GPU bubbles from naive integration.** Current systems have frequent prefill–decode interactions due to the inflight batching mechanism. One phase can easily block the other, creating GPU bubbles. **C-2: Unmanaged contention in spatial multiplexing.** Existing techniques [10, 12, 13] partition only SMs while leaving memory bandwidth unmanaged. As a result, memory bandwidth contention can lead to SLO violations.

3 MuxWise's Design

3.1 Architecture Overview

Based on the PD multiplexing paradigm, we propose MuxWise, an LLM serving framework that achieves high goodput on diverse workloads. Figure 8 shows the overview of MuxWise that comprises (1) a bubble-less multiplex engine, (2) a contention-tolerant estimator, and (3) an SLO-aware dispatcher.

To enable PD multiplexing with bubble-less coordination, the engine partitions prefill into layer-wise execution, aligning its latency with decode execution. Notably, layer-wise execution incurs negligible overhead and avoids the inefficiencies of chunk-prefill. In addition, it provides an extra benefit: preempting ultra-long prefills to prevent the SLO violations caused by them.

To avoid SLO violations caused by unpredictable contention, the estimator provides worst-case latency estimates by combining a solo-run latency predictor with a contention guard. Both components are built from one-time offline profiling for each LLM on a given hardware. They consider five key factors: reused length, input length, output length, decoding batch size, and partition configuration. LLM-specific factors are extracted from workload traces to guide profiling.

As requests arrive, the SLO-aware dispatcher leverages the engine and estimator to schedule prefill layers and decode iterations dynamically for high goodput. Specifically, the

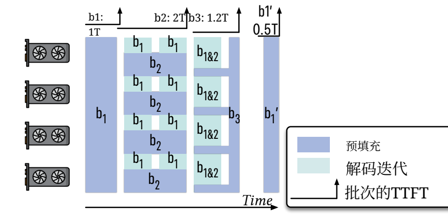


图7。 理想的解决方案：预填充解码多路复用。

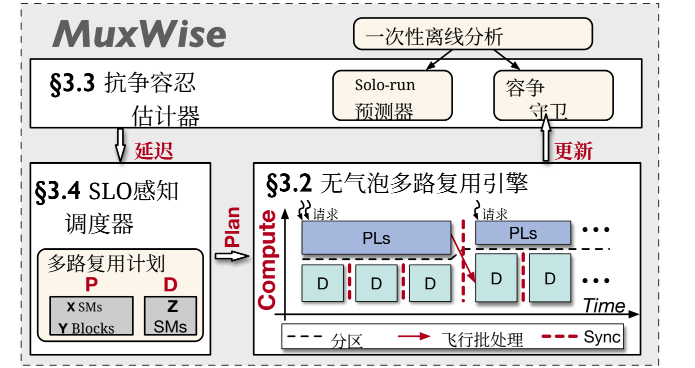


图8。 MuxWise架构概述

2.3.2 分块预填充。

分块预填充 [1]采用 GPU 内部计算融合。如图 图 4-(c) 所示，它将预填充分块，并将每个块与解码迭代融合。为了保证解码 SLO，分块预填充限制令牌预算，即预填充块的新 tokens 与解码批次 tokens 的总和。虽然分块预填充已知存在平方级内存开销 [53], 我们发现另一个缺点。具体来说，分块引入了 SLO 达成与利用率之间的矛盾。

图6-(a)展示了不同令牌预算下的分块预填充中的TBT。在此实验中，融合的解码迭代具有32的静态批次大小和1K令牌的复用上下文长度，Llama3-70B部署在具有8个A100 GPU的服务器上。如图所示，延迟在令牌预算达到 $4K$之前并非线性增加。这表明饱和GPU需要预填充块具有输入长度为$(4K - 32)$。然而，相应的延迟是$505ms$，远高于典型的TBT SLO目标（$< 100ms$）。

图6-(b)展示了预填充中不同复用上下文长度的TBT。在此实验中，令牌预算固定为512，解码迭代的复用上下文长度为1K。如图所示，当复用上下文超过 $4K$后，TBT明显增加。这种复用上下文长度在长上下文理解和多轮工作负载中很常见，如表1所示。在这种情况下，分块预填充很容易导致SLO违规。

2.4 新范式与挑战

如图7所示，我们提出了一个GPU内部预填充-解码（PD）多路复用范式，以克服上述限制。具体来说，预填充和解码动态共享每个GPU内的计算资源（SMs）。通过预留足够的SMs以满足解码SLO，并将剩余的SMs分配给预填充，从而实现了高吞吐量大语言模型服务。PD多路复用克服了先前方法的局限性，具有以下优势。

首先，多路复用支持动态和自适应的计算管理。如图所示，计算资源可以在两个阶段之间灵活分配，以在保证SLO的同时最大化系统吞吐量。其次，多路复用将计算与内存管理解耦。尽管这两个阶段划分了计算资源，但它们共享内存

每块GPU上的空间，实现高效的KV缓存重用。第三，多路复用允许预填充和解码独立运行而不会互相阻塞，避免了SLO达成与系统吞吐量之间的困境

然而，将GPU内部多路复用集成到现有的LLM服务系统中并非易事。实现这一范式存在两个挑战。**C-1：从简单集成导致的GPU气泡。**当前系统由于飞行中批处理机制，频繁出现预填充-解码交互。一个阶段很容易阻塞另一个阶段，从而产生GPU气泡。**C-2：空间多路复用中的未管理容争。**现有技术 [10, 12, 13]仅对SMs进行分区，而内存带宽则未得到管理。结果，内存带宽容争可能导致SLO违规。

3 MuxWise 的设计

3.1 架构概述

基于PD多路复用范式，我们提出了MuxWise，一个在各种工作负载上实现高吞吐量的LLM服务框架。图8 展示了MuxWise的架构概述，它由(1)一个无气泡多路复用引擎、(2)一个容争耐受估计器以及(3)一个SLO感知调度器组成。

要实现无气泡协调的PD多路复用，引擎将预填充分区为逐层执行，使其延迟与解码执行对齐。值得注意的是，逐层执行产生的开销可以忽略不计，并避免了分块预填充的低效性。此外，它还提供了一个额外的优势：通过抢占超长预填充来防止其造成的SLO违规。

为避免由不可预测的容争引起的SLO违规，估计器通过结合Solo-run延迟预测器和抗争保护来提供最坏情况延迟估计。这两个组件都是基于给定硬件上每个LLM的一次性离线分析构建的。它们考虑了五个关键因素：重用长度、输入长度、输出长度、解码批次大小和分区配置。LLM特定的因素从工作负载跟踪中提取出来，以指导分析。

随着请求到达，SLO感知调度器利用引擎和估计器动态调度预填充层和解码迭代以实现高吞吐量。具体来说，

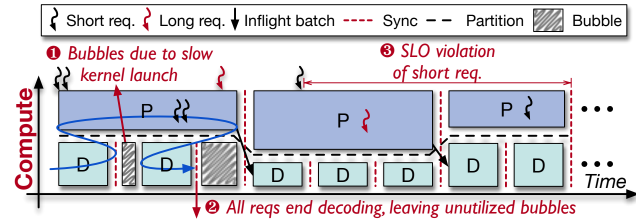


Figure 9. Bubbles and SLO violation of naive intra-process multiplexing. → represents the kernel launch order.

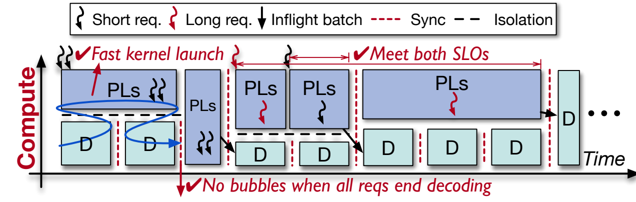


Figure 10. Bubble-less coordination using layer-wise scheduling for the prefill phase and graph-level scheduling for the decode phase. PLs is the short for prefill layers.

dispatcher reserves best-fit SMs to satisfy decode SLOs based on the worst-case estimation, and assigns the remaining SMs to prefill. Meantime, during online serving, it further refines the contention guard using runtime execution data.

3.2 Bubble-less Multiplex Engine

3.2.1 Spatial Multiplexing Technique. According to the analysis in §2.4, MuxWise imposes two requirements for PD multiplexing. First, the compute resources must be dynamically partitioned between the two phases with low overhead. Second, the memory space must be shared across the phases to enable efficient KV cache reuse. To meet these requirements, we examine existing approaches for spatially partitioning GPU compute across tasks.

We categorize these approaches into two types: inter- and intra-process partitioning. Inter-process approaches, such as CUDA MIG [12] and CUDA MPS [13], cannot provide flexible compute resource adjustment, let alone the introduced cross-process communication between prefill and decode. In contrast, the intra-process approach GreenContext [10] enables low-overhead resource adjustment by binding CUDA streams to specific SMs, with reconfiguration costing only a stream synchronization (on the order of microseconds). Furthermore, because both phases reside in the same process under GreenContext, they can directly share the same memory space for maintaining a single KV cache pool.

3.2.2 Inefficiencies from Naive Integration. With intra-process compute partitioning, a naive way to support PD multiplexing is to launch the ongoing decode iteration before the prefill phase of new request. This ordering is motivated by launch latency difference: launching a decode iteration takes less than 0.5 ms, whereas launching a prefill phase takes tens of milliseconds.

Ideally, the launch latency of either a prefill phase or decode iteration can be reduced to a single CUDA graph launch (~0.5*ms*). However, CUDA graph requires offline construction with static configuration and incurs memory overhead. In prefill phase, both batch size and input length vary, whereas decode iteration varies only in batch size. Thus, single-graph optimization is feasible only for decode phase with several selected batch sizes [40], while applying it to prefill phase would require capturing much more graphs, incurring unacceptable memory overhead.

Prefill phase can be optimized through piecewise CUDA graph [40], which splits the prefill phase into multiple layer-wise CUDA graphs. It still incurs ~**10 ms** launch overhead for Llama-70B on 8 A100 GPUs. Fortunately, prefill phases consists of long-duration kernels, which are typically longer than the launch time. It does not suffer noticeable performance degradation from launch overhead in most cases.

To this end, when both a prefill phase and a decode iteration are pending, MuxWise prioritizes launching the decode iteration; otherwise, the SMs allocated for the decode phase would remain idle for tens of milliseconds. However, this naive approach still introduces two types of GPU bubbles and can also lead to SLO violations.

Firstly, when the prefill launch time exceeds the execution time of a decode iteration, next decode iteration cannot launch in time, and GPU bubbles occur. As shown on the left of Figure 9, a bubble appears between two decode iterations because the serving system must return newly generated tokens after each iteration.

Secondly, bubbles can arise from unpredictable termination of decode. As depicted in the middle of Figure 9, all requests in a decode batch may finish token generation while a concurrent prefill phase has already been launched. Due to the non-preemptive nature of GPU execution, the launched prefill cannot be interrupted to reclaim compute resources.

Thirdly, SLO violations may occur due to workload skew among requests, as illustrated in the upper-right of Figure 9. Context lengths can vary significantly, with short conversations coexisting alongside long-text summarizations. In such cases, a short request may suffer long queuing delays while waiting for the prefill of an ultra-long request. If the short request has limited SLO slack, it is likely to miss its deadline.

3.2.3 Bubble-less Coordination. The above inefficiencies stem from the large latency discrepancy between the prefill and decode phases. This is because prefill phases typically take longer to launch and execute, and the execution time of both phases is highly variable at runtime. To address this, we propose layer-wise execution for prefill and query-based synchronization.

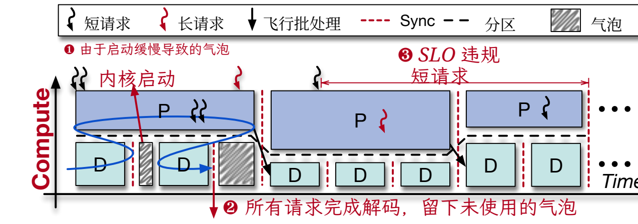


图9。 传统进程内多路复用的气泡和SLO违规。 → 表示内核启动顺序。

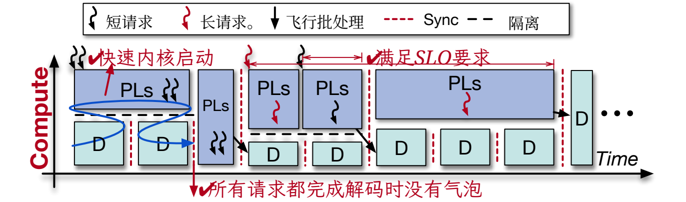


图10。 无气泡协调使用逐层调度预填充阶段和图级调度解码阶段。PLs是预填充层的缩写。

调度器根据最坏情况估计预留最佳匹配的SMs以满足解码SLOs，并将剩余的SMs分配给预填充。同时，在在线服务过程中，它进一步使用运行时执行数据优化抗争保护。

3.2 无气泡多路复用引擎

3.2.1 空间多路复用技术。 根据§2.4的分析，MuxWise 对PD多路复用提出了两个要求。首先，计算资源必须在两个阶段之间以低开销动态分区。其次，内存空间必须跨阶段共享以实现高效的KV缓存重用。为了满足这些要求，我们考察了现有的GPU计算空间分区的任务方法。

我们将这些方法分为两类：进程间和进程内分区。进程间方法，如CUDA MIG [12] 和CUDA MPS [13],，无法提供灵活的计算资源调整，更不用说预填充和解码之间引入的跨进程通信。相比之下，进程内方法GreenContext [10] 通过将CUDA流绑定到特定SMs实现低开销资源调整，重新配置只需一个流同步（微秒级）。此外，由于GreenContext下两个阶段都位于同一进程，它们可以直接共享相同的内存空间以维护单个KV缓存池。

3.2.2 简单集成带来的低效。 在进程内计算分区中，一种支持PD多路复用的简单方法是在新请求的预填充阶段之前启动正在进行的解码迭代。这种顺序是由启动延迟差异驱动的：启动解码迭代的时间不到0.5毫秒，而启动预填充阶段则需要数十毫秒。

理想情况下，预填充阶段或解码迭代的启动延迟可以降低到单个CUDA图启动(~0.5*ms*)。然而，CUDAgraph需要离线构建静态配置并产生内存开销。在预填充阶段，批次大小和输入长度都会变化，而解码迭代仅批次大小变化。因此，单图优化仅适用于解码阶段且仅对几个选定的批次大小 [40], 有效，将其应用于预填充阶段将需要捕获许多更多图，导致不可接受的内存开销。

预填充阶段可以通过分段CUDA图 [40], 进行优化，该图将预填充阶段拆分为多个逐层CUDA图。它仍然会导致Llama-70B在8 A100 GPU上产生 ~**10 毫秒**启动开销。幸运的是，预填充阶段由长时序内核组成，这些内核通常比启动时间更长。在大多数情况下，它不会因启动开销而出现明显的性能下降。

为此，当预填充阶段和解码迭代都待处理时，MuxWise优先启动解码迭代；否则，分配给解码阶段的SMs会空闲几十毫秒。然而，这种简单的方法仍然引入了两种类型的GPU气泡，也可能导致SLO违规。

首先，当预填充启动时间超过解码迭代执行时间时，下一个解码迭代无法及时启动，并出现GPU气泡。如图左侧所示图9，由于服务系统必须在每次迭代后返回新生成的标记，因此两个解码迭代之间会出现气泡。

其次，气泡可能源于解码的不确定终止。如图9中间所示，解码批次中的所有请求可能在并发预填充阶段已经启动时完成token生成。由于GPU执行的不可抢占特性，已启动的预填充无法被中断以回收计算资源。

第三，SLO违规可能由于请求间的负载倾斜而发生，如图9右上角所示。上下文长度可能差异很大，简短对话与长文本摘要并存。在这种情况下，简短请求可能在等待超长请求预填充时遭受较长的排队延迟。如果简短请求的SLO余量有限，它很可能错过截止时间。

3.2.3 无气泡协调。 上述低效性源于预填充和解码阶段之间较大的延迟差异。这是因为预填充阶段通常需要更长时间来启动和执行，并且两个阶段的执行时间在运行时高度可变。为了解决这个问题，我们提出了逐层执行预填充和基于查询的同步。

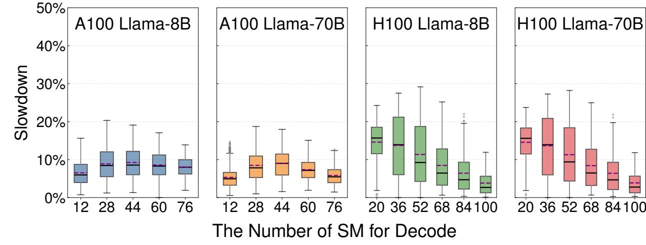


Figure 11. Slowdowns in decode due to contention with different multiplexing configurations of prefill and decode across models and GPUs.

Layer-wise execution for prefill. As shown in Figure 10, MuxWise splits the prefill phase into layers (PLs). Based on this new granularity, MuxWise eliminates GPU bubbles and prevents SLO violations. For the first type of bubble, MuxWise can launch enough PBs to occupy compute resources for prefill, and return in time before the decode phase finishes computation. For the second type, MuxWise switches the execution of later prefill layers into a new GreenContext, just after the decode phase terminates. For SLO violations caused by long requests, layer-wise execution enables preemption, allowing short requests to be prioritized, thereby meeting the SLO targets of both. Importantly, layer-wise execution incurs negligible overhead, since LLMs are inherently structured as multiple transformer layers.

Query-based synchronization. In addition to the above bubbles, inflight batching can also introduce GPU bubbles. Specifically, when the last prefill layer completes, it must block the next decode iteration to merge requests into the decode batch. This blocking creates small bubbles, since prefill and decode rarely finish simultaneously. To address this, MuxWise employs query-based synchronization that periodically polls CUDA events. MuxWise continues launching decode batches and prefill layers asynchronously, and when an event is observed complete, the corresponding prefill request is immediately merged into the current decode batch.

3.3 Contention-tolerant Estimator

When the prefill and decode phases are spatially multiplexed, contention can arise, particularly from unmanaged resources such as memory bandwidth. We begin by analyzing contention between the two phases under spatial multiplexing and then introduce our modeling method.

3.3.1 Contention Analysis. While GreenContext supports precise compute resource allocation, it cannot manage memory or network bandwidth. In particular, efficient techniques for bandwidth management are lacking. Worse, current GPUs do not expose runtime monitoring of bandwidth usage, and both prefill and decode phases can heavily consume bandwidth, making contention hard to predict.

Table 2. Compute analysis for prefill and decode phases.

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

To evaluate the impact on execution slowdown, we extensively profile prefill and decode under multiplexing using Llama-8B and Llama-70B. Figure 11 reports the decode slowdown on servers with 8 A100 and 8 H100 GPUs. The x-axis denotes the number of SMs allocated to the decode batch, with the remaining SMs assigned to the prefill batch. For each configuration, the prefill batch's total context length (reused + new) ranges from 1,024 to 128K tokens, whereas the decode batch's reused length ranges from 1,024 to 1,024K tokens. This profiling takes one week.

As shown in Figure 11, contention-induced slowdown ranges from nearly zero to about 30% across different partition configurations and GPUs. The high variation across hardware partitions indicates the inherent unpredictability of contention slowdown. Although both models exhibit similar slowdown trends on the same GPUs due to their architectural similarity, this observation does not aid contention modeling. Meanwhile, results for prefill are similar but omitted due to space constraints.

3.3.2 Worst-case Estimation for SLO guarantee. To mitigate the risk of SLO violations caused by unpredictable contention in online serving, MuxWise introduces a worst-case latency estimation method tailored for SLO guarantees. *The key observation is that precise latency prediction is not the only way to guarantee SLO. What matters is ensuring that the latency of a scheduled phase, given its allocated compute resources, does not exceed the predefined target.* Thus, MuxWise performs worst-case estimation by first predicting its solo-run latency, and then applying a maximum slowdown factor.

Solo-run predictor. To predict solo-run latency, we analyze the compute complexity of prefill and decode, and construct a predictor using offline profiling. The latency of each prefill or decode iteration is determined by the token lengths of the reused and new context. Table 2 summarizes the compute analysis of the prefill and decode phases with a batch size of 1. The key factors are as follows:

- 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.

Based on the complexity analysis in Table 2, we build latency prediction models for the prefill and decode phases. The prefill model is given in Equation 1, and the decode model in Equation 2, where all θ terms are coefficients. We

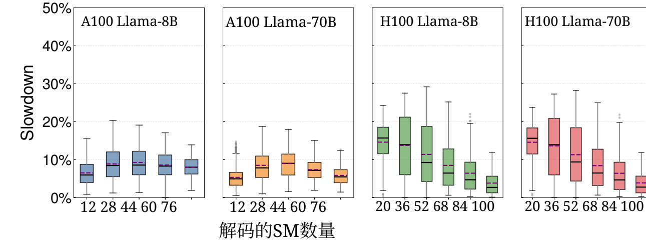


图11. 由于预填充和解码在不同模型和GPU上的多路复用配置之间发生容争，导致解码延迟。

预填充的逐层执行。如图10所示，MuxWise 将预填充阶段分层（PLs）。基于这种新的粒度，MuxWise消除了GPU气泡并防止SLO违规。对于第一种气泡，MuxWise 可以启动足够的PBs来占用预填充的计算资源，并在解码阶段完成计算之前及时返回。对于第二种，MuxWise将后续预填充层的执行切换到解码阶段终止后的新Green-Context。对于由长请求引起的SLO违规，逐层执行允许抢占，使短请求能够优先处理，从而满足两个阶段的SLO目标。重要的是，逐层执行的开销可以忽略不计，因为大语言模型本质上由多个Transformer层组成。

基于查询的同步。除了上述气泡外，飞行批处理也可能引入GPU气泡。具体来说，当最后一个预填充层完成时，它必须阻塞下一个解码迭代以将请求合并到解码批次中。这种阻塞会创建小气泡，因为预填充和解码很少同时完成。为了解决这个问题，MuxWise采用基于查询的同步，该同步定期轮询CUDA事件。MuxWise继续异步地启动解码批次和预填充层，当观察到事件完成时，相应的预填充请求会立即合并到当前解码批次中。

3.3 抗争容忍估计器

当预填充和解码阶段空间多路复用时，可能会出现容争，特别是来自未管理资源（如内存带宽）。我们首先分析空间多路复用下两个阶段之间的容争，然后介绍我们的建模方法。

3.3.1 容争分析。虽然 GreenContext 支持精确的计算资源分配，但它无法管理内存或网络带宽。特别是，带宽管理的有效技术缺乏。更糟的是，当前的 GPU 没有公开带宽使用的运行时监控，并且预填充和解码阶段都可能大量消耗带宽，使得容争难以预测。

表2。 预填充和解码阶段的计算分析。

	注意力机制	FFN
预填充无缓存	$O(Ld^2 + L^2d)$	$O(Ld^2)$
预填充 w/ 缓存	$O(nd^2 + Lnd)$	$O(nd^2)$
解码	$O(d^2 + (r+1)d)$	$O(d^2)$

为了评估执行速度下降的影响，我们使用 Llama-8B 和 Llama-70B 对多路复用下的预填充和解码进行了广泛的分析。图11报告了在配备 8 个 A100 和 8 个 H100 GPU 的服务器上的解码速度下降情况。x 轴表示分配给解码批次的 SM 数量，其余的 SM 分配给预填充批次。对于每种配置，预填充批次的总上下文长度（重用 新）范围从 1,024 到 128K 个 token，而解码批次的重用长度范围从 1,024 到 1,024K 个 token。这项分析持续了一周。

如图11所示，容争引起的延迟从接近零到大约30%在不同分区配置和GPU之间变化。硬件分区之间的高差异表明容争延迟的固有不可预测性。尽管由于架构相似性，这两个模型在相同GPU上表现出相似的延迟趋势，但这一观察结果并不能帮助容争建模。同时，预填充的结果相似但出于篇幅限制而省略。

3.3.2 最坏情况估计用于SLO保证。为了减轻在线服务中由不可预测的容争引起的SLO违规风险，MuxWise引入了一种针对SLO保证的最坏情况延迟估计方法。关键观察是，精确的延迟预测不是保证*SLO*的唯一方法。重要的是确保在分配的计算资源下，调度阶段的延迟不超过预定义的目标。因此，MuxWise通过首先预测其单次运行延迟，然后应用最大延迟因子来进行最坏情况估计。

Solo-run 预测器。要预测 Solo-run 延迟，我们分析预填充和解码的计算复杂度，并使用离线分析构建预测器。每个预填充或解码迭代的延迟由重用和新上下文的 token 长度决定。表 2 总结了批次大小为 1 的预填充和解码阶段的计算分析。关键因素如下：

- d: 每个 token 表示的隐藏维度。
- L: 总 token 长度。
- r: 重用（缓存）上下文的 token 长度。
- $n = L - r$: 新上下文的 token 长度。

根据表2的复杂度分析，我们为预填充和解码阶段构建了延迟预测模型。预填充模型给出在公式1中，解码模型在公式2中，其中所有 θ 项都是系数。我们

separate the models because state-of-the-art serving frameworks adopt different execution paths for these two phases, including distinct GPU kernel implementations and launch methods.

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

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

The trained models achieve high accuracy, with a maximum deviation of 8.16% for prefill and 8.84% for decode, effectively supporting MuxWise's online scheduling. Meanwhile, the offline profiling for training the solo-run predictor can be completed within a few hours, which is acceptable. It is a one-time effort per LLM–machine pair and has been widely adopted in prior LLM serving work [1, 44, 53].

Contention guard. To provide the maximum slowdown factor, MuxWise introduces the contention guard. Specifically, the contention guard provides slowdown factors only for decode, which will be explained in §3.4.1. The contention guard is built using data collected through grid-sampling-based profiling. This profiling spans five variables: the number of new and reused tokens in prefill, the batch size, and total reused tokens in decode, and the partition configuration. For each pair of prefill and decode iterations to be estimated, the contention guard returns the maximum slowdown factor of the grid cell they fall into as the estimation result.

Building such a contention guard incurs much higher offline profiling overhead than the solo-run predictor, since pairwise profiling is needed to obtain slowdown data. Fortunately, extensive profiling shows that slowdown remains within a limited range, with a maximum of 20% on A100 GPUs and 30% on H100 GPUs. This indicates that even with coarse-grained profiling, worst-case latency inflation does not exceed 30%.

To this end, we initialize the contention guard using coarse-grained grid sampling. Specifically, we sample variables such as new and reused tokens in prefill, as well as single-request reused tokens in decode batches, at powers-of-4 granularity, ranging from 2K to 128K. The sampled decode batch sizes follow SOTA serving frameworks (around 20 batch sizes). We partition GPUs at the granularity of 16 SMs, yielding 6 configurations for A100 and 7 for H100. In total, the number of samples per LLM–machine pair is calculated as $(4\times4-1)\times4\times20\times6 \approx 7K$, which can be collected within 12 hours. In the equation, we exclude the case with 128K new and 128K reused tokens in prefill, since 128K is the maximum context window supported by mainstream LLMs [5, 15].

The reason for using 16 SMs as the granularity are twofold: (1) kernels on H100 and newer GPUs requires 16 SMs, due to using new features like thread block cluster [3], and (2)

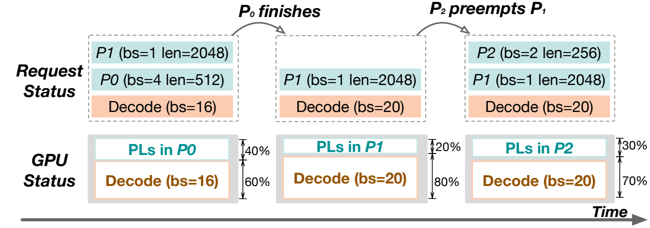


Figure 12. The dispatching policy of MuxWise.

experiments indicate that 16 SMs already deliver strong performance improvements. Finer-grained scheduling offers little benefit while increasing memory overhead.

Furthermore, MuxWise leverages runtime execution data to continuously update the contention guard, thus refining its SLO guarantees. Even with the coarse-grained contention guard, MuxWise already outperforms existing baselines in both SLO attainment and system goodput (§4).

3.4 SLO-aware Dispatcher

With bubble-less multiplexing and contention-tolerant modeling, we introduce MuxWise's detailed dispatching policy.

3.4.1 Priorities of prefill and decode. In this work, we focus on the scheduling within a single serving instance. In MuxWise, 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.

This is also why the contention guard in the contention-tolerant estimator only provides a maximum slowdown factor for decode. When predicting the prefill phase, MuxWise does not need an accurate or worst-case estimate. It only requires that the predicted latency of the launched prefill layers exceeds that of the corresponding decode iteration, ensuring full utilization of the allocated compute resources.

3.4.2 Dispatching policy. Building on the above analysis, Figure 12 illustrates MuxWise's SLO-aware dispatching policy. The system makes scheduling decisions after each prefill batch completes and at the end of each decode iteration. Specifically, the dispatcher selects prefill layers either from the ongoing prefill batch or from a new batch in the request queue, and allocates compute resources between the prefill and decode phases.

As shown in Figure 12, the dispatcher allocates a best-fit number of SMs (60%) to decode and assigns the remaining

将模型分开，因为最先进的推理框架对这两个阶段采用不同的执行路径，包括不同的GPU内核实现和启动方法。

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

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

训练好的模型达到高精度，最大偏差为8。16%，用于预填充和8。84%，用于解码，有效支持MuxWise的在线调度。同时，为训练solo-run预测器的离线分析可以在几小时内完成，这是可以接受的。这是每个LLM-机器对的一次性工作，并在先前的LLM服务工作中被广泛采用[1, 44, 53]。

抗争保护。为了提供最大的减速因子，MuxWise 引入了抗争保护。具体来说，抗争保护仅提供解码的减速因子，这将在§3.4.1中解释。抗争保护是使用基于网格采样的分析收集的数据构建的。这种分析跨越了五个变量：预填充中新增和重用token的数量、批次大小以及解码中重用的总token数和分区配置。对于要估计的每一对预填充和解码迭代，抗争保护会返回它们所在的网格单元的最大减速因子作为估计结果。

构建这种抗争保护会带来比 Solo-run 预测器高得多的离线分析开销，因为需要成对分析才能获取减速数据。幸运的是，广泛的分析表明减速保持在有限范围内，在 A100 GPU 上最高为 20%，在 H100 GPU 上最高为 30%。这表明即使使用粗粒度分析，最坏情况延迟膨胀也不会超过 30%。

为此，我们使用粗粒度网格采样初始化抗争守卫。具体来说，我们在预填充中采样新和重用的 token，以及在解码批次中采样单请求重用的 token，以 4 的幂次粒度进行采样，范围从 2K 到 128K。采样的解码批大小遵循 SOTA 服务框架（约 20 个批大小）。我们将 GPU 分区粒度为 16 个 SM，为 A100 提供 6 种配置，为 H100 提供 7 种配置。总共，每个 LLM-机器对的样本数量计算为 $(4\times4-1)\times4\times20\times6 \approx 7K$，可以在 12 小时内收集。在公式中，我们排除了 128K 新的案例。
并且预填充中重用了128K个token，因为128K是主流大语言模型 [5, 15]支持的最大上下文窗口。

使用 16 个 SM 作为粒度的原因有两个：(1) H100 和更新的 GPU 上的内核需要 16 个 SM，因为它们使用了新特性，如线程块集群 [3]，，以及 (2)

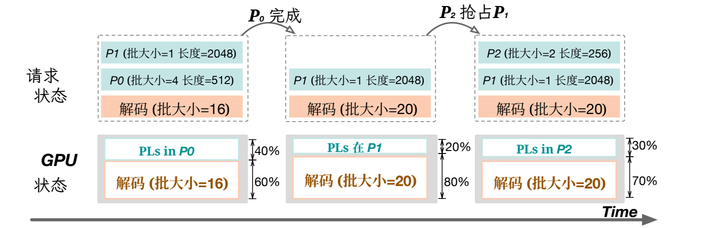


图12。MuxWise的调度策略。

实验表明，16 个 SM 已经能带来显著的性能提升。更细粒度的调度几乎没有收益，同时会增加内存开销。

此外，MuxWise利用运行时执行数据来持续更新抗争保护，从而优化其SLO保证。即使使用粒度化的抗争保护，MuxWise 在SLO达成和系统吞吐量方面（§4）已经优于现有基线。

3.4 SLO感知调度器

通过无气泡多路复用和容争容忍建模，我们介绍了MuxWise的详细调度策略。

3.4.1 预填充和解码的优先级。 在本工作中，我们关注单个服务实例内的调度。在MuxWise中，我们优先保证解码阶段的SLO达成，并尽可能早地处理预填充阶段。预填充阶段的SLO达成不能直接保证，有以下几个原因。首先，尽管我们优先考虑解码阶段，但我们只为它分配了刚刚足够的计算资源。由于剩余的计算资源被分配给预填充阶段，因此通常预期其SLO能够被满足。其次，当预填充阶段的SLO违规发生时，这表明推理负载已超过当前LLM服务实例的峰值容量。在这种情况下，进一步的调度努力将不再提升性能。

这也是为什么在容争耐受估计器中的容争守卫仅对解码提供最大减速因子。在预测预填充阶段时，MuxWise 不需要准确或最坏情况的估计。它只需要启动的预填充层的预测延迟超过相应的解码迭代，以确保完全利用分配的计算资源。

3.4.2 调度策略。 基于以上分析，图12展示了MuxWise的SLO感知调度策略。系统在每个预填充批次完成后以及每次解码迭代结束时做出调度决策。具体而言，调度器从正在进行的预填充批次或请求队列中的新批次中选择预填充层，并在预填充和解码阶段之间分配计算资源。

如图12所示，调度器为解码分配了最佳匹配数量的SMs（60%），并将剩余

SMs (40%) to prefill phase $P0$. The resource partition satisfies the decode SLO and maximizes prefill throughput guided by the contention-tolerant estimator. To support layer-wise execution in the multiplex engine, the estimator computes the number of prefill layers to launch as $N_{PL} = \lceil (T_d \times N_T)/T_P \rceil$, where T_d is the estimated decode latency, T_P is the estimated prefill latency, and N_T is the number of transformer layers in the served LLM.

Once $P0$ finishes computation, it is merged into the decode batch, increasing the batch size to 20. The scheduler then retrieves a new prefill batch $P1$ and adjusts the partition to 20% SMs for prefill and 80% for decode to meet SLO targets.

Later, when a new prefill batch $P2$ arrives, it would normally wait for $P1$ to complete. However, because $P1$ has a long input length, this delay risks violating P2's SLO. To avoid this, MuxWise allows $P2$ to preempt $P1$, provided that preemption does not cause $P1$ to miss its own TTFT SLO.

MuxWise does not allow recursive preemption. For example, after $P2$ preempts prefill batch $P1$, no other batch may preempt $P2$. This design is reasonable, as short requests typically preempt long ones, and preempting a short request in turn would likely cause it to miss its SLO. MuxWise checks SLO attainment only when a prefill batch is preempted; otherwise, it prioritizes processing the active prefill batch as quickly as possible. Notably, preemption in MuxWise is optional. Even when disabled, MuxWise still delivers substantial performance improvements over the baselines.

4 Evaluation

4.1 Experimental Setup

Testbed. We mainly evaluate MuxWise on a server equipped with 8 A100-80GB GPUs. The GPUs are interconnected via NVLINK, providing 600 GB/s of bandwidth. We also evaluate MuxWise on two additional servers to demonstrate its effectiveness on newer GPUs and larger LLMs: one with 8 H100-SMX5-80GB GPUs and another with 8 H200-SMX5-141GB GPUs. These servers offer higher compute capability and larger GPU memory. All experiments are conducted with PyTorch 2.6.0 [31]. MuxWise is implemented using SGLang [52] version 0.4.10post2. The GPU driver version is 570.124.06, and the CUDA version is 12.8.

Models. We primarily evaluate MuxWise using two LLMs from the Llama family [15, 37, 38]: Llama-8B and Llama-70B. These models differ in size and represent the most commonly hosted LLMs in the cloud. We also evaluate a larger MoE model, Qwen3-235B with 22B activated, to demonstrate MuxWise's generality.

Baselines. We compare MuxWise against 3 state-of-the-art solutions for efficient LLM serving. Model parallelism techniques such as tensor parallelism [51] are employed to parallelize the deployed models. For MuxWise, we fix the tensor parallelism degree to 8. Details of each baseline's model parallelism configuration are provided when the baseline is introduced. For all systems, the KV cache memory pool is configured as large as possible to maximize throughput.

- **Chunked-prefill in SGLang [1]**: This version of SGLang is equipped with chunked-prefill, as proposed by SARATHI-Serve [1]. We follow SARATHI-Serve's methodology to calculate the token budget for each workload prior to experiments. It is offline tuned under specific TBT targets for each model. Unlike SARATHI-Serve, which serially executes prefill and decode attention kernels, SGLang leverages Flashinfer [46], a high-performance inference kernel library, to fuse them into a single kernel. It is expected to deliver performance similar to POD-attention [21].
- **NanoFlow [54]**: This is an enhanced version of chunked-prefill with operator-level intra-GPU multiplexing, targeting near-optimal throughput under a relatively loose SLO requirement (200 ms). It requires a large token budget (at least 1024) to achieve this goal. However, such a token budget cannot meet the SLO requirements of modern LLM serving (≤ 100 ms). We use the same token budget as chunked-prefill for NanoFlow.
- **LoongServe [44]**: This is a dynamic disaggregated serving system. We adopt its model-parallelism configuration. For Llama-70B, sequence parallelism is set to 2 and tensor parallelism to 4. For Llama-8B, sequence parallelism is set to 4 and tensor parallelism to 2. It does not support new LLMs like MoE models.
- **Disaggregated serving in SGLang (SGLang-PD in short)**: This is the latest implementation of static disaggregation with KV-cache sharing across phases and requests. The P:D ratio is 1:1, with tensor parallelism set to 4 for each instance. DistServe [53] does not support KV-cache sharing across requests, making it unsuitable for modern LLM services. We also evaluated Dynamo [14], which performed substantially worse than SGLang-PD. Therefore, we use SGLang-PD as the state-of-the-art baseline of static disaggregation for evaluation.

Metrics. MuxWise targets goodput improvement. So, following prior works [1, 32, 34, 53], we use tail latency (e.g., P99) to assess SLO attainment. Meanwhile, there is also another metric to measure the SLO guarantee during decode phase: TPOT (timer per output token). In comparison, TBT accounts the latency of each individual token, whereas TPOT is an average metric that may mask the poor performance of some tokens [42]. Thus, we choose TBT over TPOT for a stricter SLO metric. We set the TBT SLO target to $50ms$ for Llama3-8B and $100ms$ for Llama3-70B, following prior works [1, 34]. We regard MuxWise's ability to deliver better TTFTs under skewed workloads as an additional benefit of the new serving paradigm. Moreover, it breaks the first-come-first-serve model used in other baselines. Thus, we only evaluate TTFT per token in §4.4.3.

SMs（40%）用于预填充阶段 $P0$。资源分区满足解码SLO，并通过抗争容忍估计器指导最大化预填充吞吐量。为了支持在多路复用引擎中的分层执行，估计器计算启动预填充层的数量为 $N_{PL} = \lceil (T_d \times N_T)/T_P \rceil$，其中 T_d 是估计的解码延迟，T_P 是估计的预填充延迟，N_T 是服务中LLM的Transformer层数量。

一旦 $P0$ 完成计算，它将被合并到解码批次中，将批次大小增加到20。调度器随后获取一个新的预填充批次 $P1$，并将分区调整为20%的SMs用于预填充和80%用于解码，以满足SLO目标。

稍后，当一个新的预填充批次 $P2$ 到达时，它通常会等待 $P1$ 完成。然而，因为 $P1$ 具有较长的输入长度，这种延迟有风险违反P2的SLO。为了避免这种情况，MuxWise允许 $P2$ 抢占 $P1$，前提是抢占不会导致 $P1$ 错过自己的TTFT SLO。

MuxWise不允许递归抢占。例如，在 $P2$ 抢占预填充批次 $P1$后，没有其他批次可以抢占 $P2$。这种设计是合理的，因为短请求通常会抢占长请求，而反过来抢占短请求可能会使其错过自己的SLO。MuxWise仅在预填充批次被抢占时检查SLO达成情况；否则，它会优先尽可能快地处理活动的预填充批次。值得注意的是，MuxWise中的抢占是可选的。即使禁用，MuxWise仍然在基线之上提供了显著的性能提升。

4 评估

4.1 实验设置

测试平台。我们主要评估 MuxWiseona 服务器，该服务器配备了 8 台 A100-80GB GPU。这些 GPU 通过 NVLINK 互连，提供 600 GB/s 的带宽。我们还评估 MuxWiseon 两台额外的服务器，以展示其在更新的 GPU 和更大的 LLM 上的有效性：一台配备 8 台 H100-SMX5-80GB GPU，另一台配备 8 台 H200-SMX5- 141GB GPU。这些服务器提供更高的计算能力和更大的 GPU 内存。所有实验均使用 PyTorch 2.6 进行。0 [31]。MuxWiseis 使用 SGLang [52] 版本 0.4.10post2 实现。GPU 驱动程序版本为 570.124.06，CUDA 版本为 12.8。

模型。我们主要使用来自 Llama 家族的两个 LLM [15, 37, 38] 评估 MuxWise：Llama-8B 和 Llama-70B。这些模型在大小上有所不同，代表了云中托管的最常见的LLM。我们还评估了一个更大的 MoE 模型 Qwen3-235B（激活 22B），以展示 MuxWise 的通用性。

基线。我们将 MuxWise 与 3 种高效 LLM 服务方案进行对比。模型并行技术（如张量并行 [51]）被用于并行化部署的模型。对于 MuxWise，我们将张量并行度固定为 8。当基线方案被介绍时，会提供每个基线模型并行配置的详细信息。

引入。对于所有系统，KV缓存内存池配置尽可能大以最大化吞吐量。

- **SGLang中的分块预填充 [1]**: 这个版本的SGLang配备了分块预填充，由SARATHI-Serve [1]提出。我们在实验之前遵循SARATHI-Serve的方法来计算每个工作负载的令牌预算。它在特定的TBT目标下针对每个模型进行离线调整。与SARATHI-Serve不同，后者串行执行预填充和解码注意力核，SGLang利用Flashinfer [46]，高性能推理内核库将它们融合成一个内核。预计将提供与POD-attention [21]相似的性能。
- **NanoFlow [54]**: 这是 chunked-预填充的增强版本，具有 GPU 级别的内部多路复用，旨在相对宽松的 SLO 要求（200 毫秒）下实现接近最优的吞吐量。它需要较大的令牌预算（至少 1024）才能实现这一目标。然而，这样的令牌预算无法满足现代 LLM 服务 (≤ 100 毫秒) 的 SLO 要求。我们为 NanoFlow 使用与 chunked-预填充相同的令牌预算。
- **LoongServe [44]**: 这是一个动态解耦的 serving 系统。我们采用其模型并行配置。对于 Llama-70B，序列并行设置为 2，张量并行设置为 4。对于 Llama-8B，序列并行设置为 4，张量并行设置为 2。它不支持新的 LLM，如 MoE 模型。
- **SGLang 中的解耦 serving（简称 SGLang-PD）**：这是最新实现的静态解耦，跨阶段和请求共享 KV 缓存。P:D 比例为 1:1，每个实例的张量并行设置为 4。DistServe [53] 不支持跨请求的 KV 缓存共享，因此不适用于现代 LLM 服务。我们还评估了 Dynamo [14], ，其表现远不如 SGLang-PD。因此，我们使用 SGLang-PD 作为静态解耦的最先进基线进行评估。

指标。MuxWise 旨在提升吞吐量。因此，遵循先前工作 [1, 32, 34, 53], 我们使用尾部延迟（例如，P99）来评估SLO达成。同时，还有一个指标用于衡量解码阶段的SLO保证：TPOT（每个输出token的计时器）。相比之下，TBT计算每个单独token的延迟，而TPOT是一个平均指标，可能会掩盖某些token的糟糕性能 [42]。因此，我们选择TBT而不是TPOT作为更严格的SLO指标。我们根据先前工作，将TBT SLO目标设置为 $50ms$对于Llama3-8B和 $100ms$ 对于Llama3-70B [1, 34]。我们认为MuxWise的能力在非均匀负载下提供更好的任务完成时间（TTFT）是新的服务范式的额外优势。此外，它打破了其他基线中使用的先到先得模式。因此，我们仅在§4.4.3.

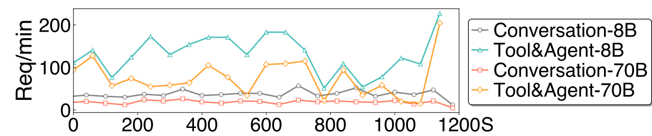


Figure 13. The two real-world workload traces after scaling.

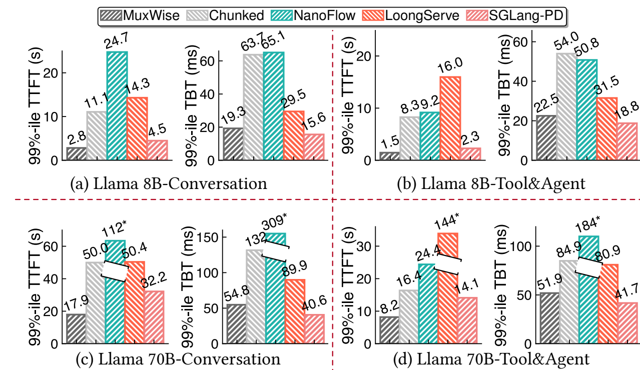


Figure 14. 99%-*ile* TTFT and TBT for Llama-8B and Llama-70B on real-world Conversation and Tool&Agent workloads. Chunked represents chunked-prefill in SGLang. Values marked with ∗ are too large; we clip their corresponding bars, so the bar height only decodes their relative size.

4.2 Evaluation on Real-world Workloads

4.2.1 End-to-end Performance. We begin by evaluating MuxWise with Llama-8B and Llama-70B under real-world workload traces. Figure 13 shows their request rates after scaling down, as they are originally from a large cluster. As illustrated, they show bursty request patterns (up to 13× spike within 1min).

Figure 14 shows the latency distribution of TTFT and TBT. Although the real-world traces are scaled down to a modest level, some baselines still easily reaches its peak throughput and enters an unstable state (NanoFlow in Figure 14-(c) and LoongServe in Figure 14-(d)). After omitting unstable results, MuxWise achieves average 99%-*ile* TTFT speedups of 3.57×, 5.98×, 4.65×, and 1.66× over chunked-prefill, NanoFlow, LoongServe, and SGLang-PD, respectively. MuxWise and the two disaggregated solutions consistently meet the TBT SLO, whereas chunked-prefill and NanoFlow fails in most cases. SGLang-PD achieves shorter TBT than MuxWise, as it statically reserves more compute resources for the decode instance.

Compared to chunked-prefill, MuxWise avoids the dilemma between SLO compliance and high utilization, bringing better performance for both prefill and decode phases. While tuning the token budget, we observe that either increasing or reducing it fails for SLO guarantee. This is because the reused length in prefill phase in the two workloads can reach up to 50K tokens. Further splitting the prefill into smaller chunks

Table 3. Results of other metrics for Llama-70B on Conversation workloads in Figure 14-(c).

	TTFT (s)		TBT (ms)		E2E (s)		TPOT (ms)	
	Avg.	P50	Avg.	TBT	Avg.	P50	Avg.	P50
MuxWise	**3.1**	**1.4**	**30.2**	**25.0**	**13.1**	**12.2**	**31.1**	**28.3**
Chunked	12.0	7.2	45.3	49.3	27.0	23.3	46.9	45.7
NanoFlow	51.4	52.3	105.6	98.9	83.4	84.2	120.2	103.2
LoongServe	17.7	14.7	61.0	58.3	38.4	35.8	62.3	60.6
SGLang-PD	7.38	3.95	32.9	32.5	18.4	16.4	33.5	33.2

does not help control the TBT. MuxWise's PD multiplexing avoids this issue entirely.

NanoFlow performs worse than the original chunked-prefill. This is because, built atop chunked-prefill, NanoFlow is designed to overlap compute-bound kernels with memory-bound or communication kernels. To achieve this, it requires a large token budget (1024 in its paper) to ensure that chunked-prefill as a whole remains compute-bound. However, in Figure 14, the token budget has to be reduced to 256 to meet TBT SLO targets, where chunked-prefill is no longer compute-bound. The long reused context length in the two evaluated real-world traces further makes chunked-prefill harder to be compute-bound. Thus, NanoFlow degrades due to overlapping memory-bound kernels.

The situation worsen for NanoFLow with Llama-70B in Figure 14-(c&d). This could be attributed to the inherent model weight reload of intra-GPU overlapping. NanoFlow split each chunk into 2 nano batches, thus duplicating loading for each decode iteration [54]. When evaluating with Llama-70B, the reloading overhead is amplified due to the larger model size. Conversely, MuxWise duplicates loading only once during prefill, which typically co-runs with tens of decode iterations. Because prefill is compute-intensive and the reload is amortized over the entire phase, MuxWise imposes negligible bandwidth pressure, which is marginal relative to its overall benefits. While NanoFlow performs poorly on the two real-world workloads, it outperforms chunked-prefill for short input sequences without cross-request context length reuse(§4.3).

Against the two disaggregated solutions, MuxWise achieves significantly better TTFT. In LoongServe, instance scaling releases the KV cache needed for reuse in the prefill phase, causing redundant recomputation. In SGLang-PD, static disaggregation often leaves decode instances idle under fluctuating real-world workloads. In contrast, MuxWise avoids KV migration and adapts to dynamic workloads through intra-GPU compute partition reconfiguration.

4.2.2 Other Latency Metrics. In this experiment, we report results for other metrics, such as end-to-end latency and TPOT. We also present these results using other statistical measures, including average and P50 values. Table 3 shows the results on the Conversation workload with Llama-70B,

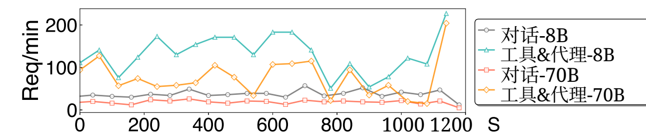


图13。 缩放后的两个真实工作负载跟踪数据。

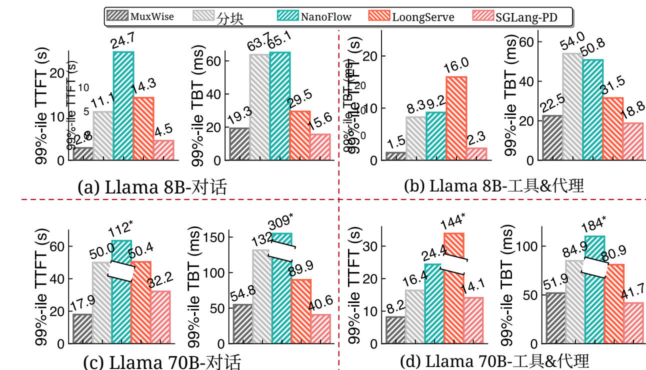


图14。 在真实对话和工具与代理工作负载上，Llama-8B和Llama-70B的99%分位数TTFT和TBT。分块表示SGLang中的分块预填充。用 *标记的值过大；我们截断了它们对应的条形图，因此条形图高度仅解码它们的相对大小。

4.2 实际工作负载上的评估

4.2.1 端到端性能。 我们首先在真实世界工作负载跟踪下评估MuxWise，使用Llama-8B和Llama-70B。图13 显示了它们在规模缩小后的请求率，因为它们原本来自一个大型集群。如图所示，它们表现出突发请求模式（最高 13×在1分钟内出现峰值）。

图14 展示了TTFT和TBT的延迟分布。虽然实际工作负载的轨迹被缩小到适度水平，但一些基线仍然轻易达到其峰值吞吐量并进入不稳定状态（NanoFlow在图14-(c)和LoongServe在图14-(d)中）。在忽略不稳定结果后，MuxWise平均达到99%-百分位 TTFT加速的 3.57×, 5.98×, 4.65×，和 1.66× 分块预填充、NanoFlow、LoongServe和SGLang-PD，分别。MuxWise 和两个解耦方案始终满足TBT SLO，而分块预填充和NanoFlow在大多数情况下都失败。SGLang-PD实现了比MuxWise更短的TBT，因为它为解码实例静态预留了更多的计算资源。

与分块预填充相比，MuxWise避免了SLO合规性与高利用率之间的两难困境，为预填充和解码阶段都带来了更好的性能。在调整令牌预算时，我们发现无论增加还是减少它，都无法保证SLO。这是因为在这两种工作负载的预填充阶段中，重用长度可以达到50K个令牌。进一步将预填充拆分为更小的块

表3. Llama-70B在对话工作负载上其他指标的结果，如图14-(c)所示。

	端到端延迟（秒）		任务批处理时间（毫秒）		端到端时间（秒）		任务处理时间（毫秒）	
	平均	P50	平均	TBT	平均	P50	平均	P50
MuxWise	**3.1**	**1.4**	**30.2**	**25.0**	**13.1**	**12.2**	**31.1**	**28.3**
分块	12.0	7.2	45.3	49.3	27.0	23.3	46.9	45.7
NanoFlow	51.4	52.3	105.6	98.9	83.4	84.2	120.2	103.2
LoongServe	17.7	14.7	61.0	58.3	38.4	35.8	62.3	60.6
SGLang-PD	7.38	3.95	32.9	32.5	18.4	16.4	33.5	33.2

并不能帮助控制TBT。MuxWise的PD多路复用完全避免了这个问题。

NanoFlow表现不如原始分块预填充。这是因为，建立在分块预填充之上，NanoFlow旨在重叠计算密集型内核与内存密集型或通信内核。为此，它需要较大的令牌预算（论文中为1024），以确保分块预填充作为一个整体保持计算密集型。然而，在图14中，令牌预算必须减少到256以满足TBT服务水平目标，此时分块预填充不再计算密集型。在两个评估的真实世界追踪中，长重用上下文长度进一步使分块预填充更难保持计算密集型。因此，NanoFlow由于重叠内存密集型内核而性能下降。

对于Llama-70B，NanoFlow在图14-(c&d)中的情况更糟。这可能是由于GPU内部重叠固有的模型权重重载所致。NanoFlow将每个块分成2个纳米批次，因此每个解码迭代都重复加载 [54]。在评估Llama-70B时，由于模型尺寸更大，重载开销被放大。相反，MuxWise 在预填充期间仅重复加载一次，这通常与数十个解码迭代并发运行。由于预填充计算密集型且重载在整个阶段摊销，MuxWise对带宽压力影响微乎其微，相对于其整体收益来说微不足道。虽然NanoFlow在两个真实世界工作负载上的表现不佳，但在没有跨请求上下文长度重用的短输入序列中（§4.3），它优于分块预填充。

在两种解耦方案中，MuxWise实现了显著更优的TTFT。在LoongServe中，实例扩展会释放用于预填充阶段重用的KV缓存，导致冗余重计算。在SGLang-PD中，静态解耦常在波动的实际工作负载下使解码实例闲置。相比之下，MuxWise避免了KV迁移，通过GPU内部计算分区重构来适应动态工作负载。

4.2.2 其他延迟指标。 在本实验中，我们报告了其他指标的结果，例如端到端延迟和TPOT。我们还使用其他统计指标呈现这些结果，包括平均值和P50值。 表3 显示了Llama-70B在对话工作负载上的结果，

Table 4. Results of other metrics for Llama-70B on Tool&Agent workloads in Figure 14-(d).

	TTFT (s)		TBT (ms)		E2E (s)		TPOT (ms)	
	Avg.	P50	Avg.	P50	Avg.	P50	Avg.	P50
MuxWise	**1.3**	**1.0**	**27.2**	24.1	**4.0**	**1.6**	**33.1**	**27.7**
Chunked	2.4	1.1	30.5	**21.2**	5.5	2.3	45.9	47.1
NanoFlow	2.8	1.2	58.8	42.4	7.4	2.5	70.3	62.6
LoongServe	59.9	56.0	52.4	50.8	65.2	61.0	56.2	54.6
SGLang-PD	2.1	1.5	31.6	31.0	5.2	2.3	37.1	33.7

while Table 4 shows the results on the Tool&Agent workload with Llama-70B in §4.2.1. Results in other settings are similar and are omitted due to space constraints.

As shown in the two tables, while MuxWise focuses on improving high goodput, it also consistently outperforms the baselines across the reported metrics. There is only one outlier in the P50 TBT of Table 4, and the values are very close. This can occur because the P50 TBT in chunked-prefill may correspond to the latency of a pure decode iteration.

4.2.3 SLO Attainment and Goodput. We also measure the SLO attainment of TBT and the corresponding goodput to evaluate the effectiveness of MuxWise in meeting SLO compliance. 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 [44]. We stop testing once the serving system becomes unstable or fails to meet the TBT SLO target.

Figure 15 shows the SLO attainment results under gradually increasing workloads. Under the constraint of meeting the *99%-ile* SLO guarantees, MuxWise achieves 2.6×, 5.2×, 2.0×, and 1.3× higher goodput than chunked-prefill, NanoFlow, LoongServe, and SGLang-PD, respectively for Llama-8B; and 3.06×, 2.62×, and 1.62× higher than chunked-prefill, LoongServe, and SGLang-PD for Llama-70B. NanoFlow never meets the SLO even with a small chunk size of 64 for Llama-70B; therefore, the corresponding goodput improvement is omitted. Table 5 further shows the corresponding token thorughput and GPU utilization of MuxWise and baselines. GPU utilization is an aggregated metric reported by NVIDIA Nsight Systems, that reflects the fraction of active SMs as well as the utilization of intra-SM resources.

Chunked-prefill, and NanoFlow fails to meet the TBT SLO even at lower request rates than the other two baselines. This is because chunking is largely ineffective at reducing TBT in real-world LLM services, where cross-request interactions are common. Compared to LoongServe, MuxWise achieves higher goodput by avoiding recomputation in multi-turn requests. Compared to SGLang-PD, MuxWise achieves higher goodput through a larger KV-cache pool and reduced idleness caused by static disaggregation. Meanwhile, MuxWise achieves shorter TTFT across all cases (up to 9.16×).

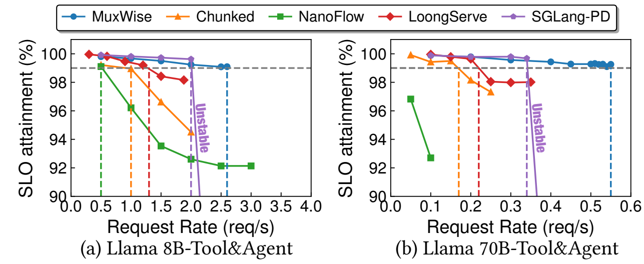


Figure 15. SLO attainment for Llama-8B and Llama-70B on Tool&Agent workload with varied request rates.

Table 5. Token throughput and GPU utilization for Llama-8B and Llama-70B on Tool&Agent workload under Goodput.

Model	Llama-8B		Llama-70B	
Metrics	Token/s	GPU Util.	Token/s	GPU Util.
MuxWise	25397	88.1	7430	84.0
Chunked	9768	63.8	2269	66.1
NanoFlow	4884	55.1	–	–
LoongServe	12698	75.3	2936	70.1
SGLang-PD	19535	P(72.4)/D(83.4)	4538	P(67.1)/D(81.9)

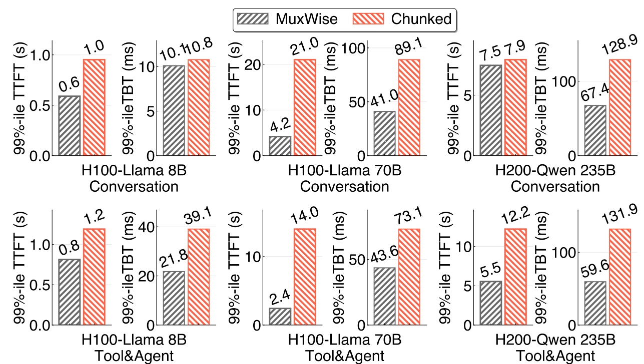


Figure 16. *99%-ile* TTFT and TBT for Llama-8B and Llama-70B on a server with 8 H100 GPUs and *99%-ile* TTFT and TBT for Qwen-235B on a server with 8 H200 GPUs.

4.2.4 More Advanced GPUs and Larger LLM. To demonstrate MuxWise's effectiveness on other GPUs and LLMs, we evaluate it with Llama-8B and Llama-70B on a server with 8 H100 GPUs, and with Qwen-235B on a server with H200 GPUs. In this experiment, we only compare MuxWise with chunked prefill. LoongServe does not support new MoE models like Qwen-235B, and disaggregated serving solutions are also infeasible for Qwen-235B, even though each H200 has 141 GB of GPU memory. Figure 16 shows the experimental results. Across all cases, MuxWise achieves an average 2.28× speedup on *99%-ile* TTFT and an average 1.81× speedup on *99%-ile* TBT. These consistent improvements demonstrate the generality of MuxWise's serving paradigm across diverse hardware and larger, newer LLMs.

表4。 Llama-70B在工具与代理工作负载上的其他指标结果图14-(d)。

	端到端延迟（秒）（毫秒）		任务批处理时间		端到端时间（秒）		任务处理时间（毫秒）	
	平均	P50	平均	P50	平均	P50	平均	P50
MuxWise	**1.3**	**1.0**	**27.2**	24.1	**4.0**	**1.6**	**33.1**	**27.7**
分块	2.4	1.1	30.5	**21.2**	5.5	2.3	45.9	47.1
NanoFlow	2.8	1.2	58.8	42.4	7.4	2.5	70.3	62.6
LoongServe	59.9	56.0	52.4	50.8	65.2	61.0	56.2	54.6
SGLang-PD	2.1	1.5	31.6	31.0	5.2	2.3	37.1	33.7

而 表4 显示了Llama-70B在工具与代理工作负载上的结果，位于§4.2.1。其他设置中的结果相似，由于篇幅限制而省略。

如表所示，虽然 MuxWise 侧重于提升高吞吐量，但它也始终在各项报告指标上优于基线。P50 TBT 在表4中只有一个异常值，且数值非常接近。这可能是因为分块预填充中的P50 TBT 对应于纯解码迭代的延迟。

4.2.3 SLO达成与吞吐量。 我们还测量了TBT的SLO达成情况以及相应的吞吐量，以评估MuxWise在满足SLO合规性方面的有效性。在这个实验中，我们从工具与代理跟踪中提取请求，但用泊松过程生成的不同速率的时间戳替换它们到达的时间戳，遵循先前的工作 [44]。一旦服务系统变得不稳定或无法满足TBT SLO目标，我们就停止测试。

图15 显示了在逐渐增加的工作负载下的SLO达成结果。在满足 *99%-ile*SLO保证的约束下，MuxWise， 2.6×, 5.2×, 2.0×, 1.3×和更高的吞吐量，对于Llama-8B，分别优于分块预填充、NanoFlow、LoongServe和SGLang-PD；以及 3.06×, 2.62×, 1.62×和高于分块预填充、LoongServe和SGLang-PD，对于Llama-70B。对于Llama-70B，即使块大小为64，NanoFlow也从未满足SLO；因此，相应的吞吐量提升被省略。 表5 进一步显示了MuxWise 和基线相应的token吞吐量和GPU利用率。GPU利用率是一个聚合指标，由NVIDIA Nsight Syste ms报告，反映了活动SM的比例以及SM内部资源的利用率。

分块预填充，并且 NanoFlow 在请求率低于其他两个基线的条件下也无法满足 TBT SLO。这是因为分块在真实世界的 LLM 服务中对于减少 TBT 效果不佳，其中跨请求交互

都是常见的。与LoongServe相比，MuxWise通过避免多轮请求中的重新计算来获得更高的吞吐量。与SGLang-PD相比，MuxWise通过更大的KV缓存池和减少静态解耦造成的闲置来实现更高的吞吐量。同时，MuxWise在所有情况下都实现了更短的TTFT（高达 9.16×）。

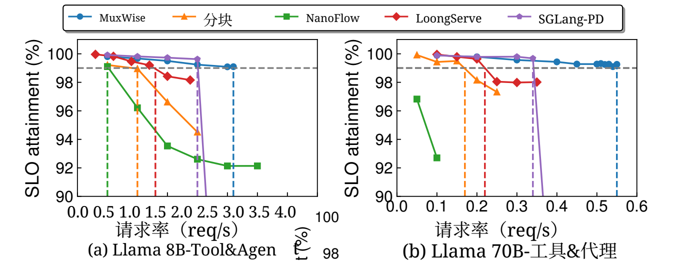


图15。 Llama-8B和Llama-70B在Tool&Agent工作负载下、不同请求率下的SLO达成情况。

表5。吞吐量下Llama-8B和Llama-70B在工具与代理工作负载中的Token吞吐量和GPU利用率。

模型	Llama-8B		Llama-70B	
指标	Token/s	GPU利用率	Token/秒	GPU利用率
Muxwise	25397	88.1	7430	84.0
分块	9768	63.8	2269	66.1
NanoFlow	4884	55.1	–	–
LoogServe	12698	75.3	2936	70.1
SGLang-PD	19535	P(72.4)/D(83.4)	4538	P(67.1)/D(81.9)

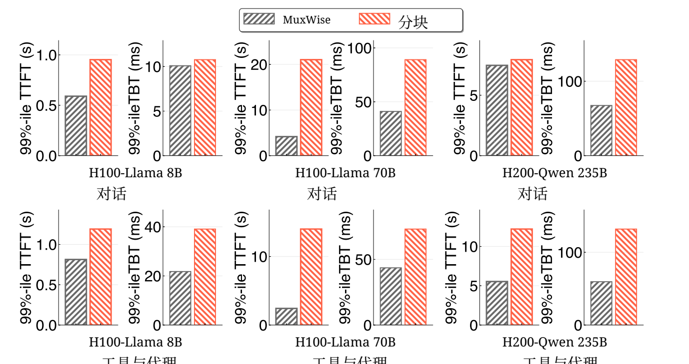


图16。 99%-分位TTFT和TBT用于Llama-8B和Llama-70B的服务器，配备8个H100 GPU，以及99%-分位 TTFT和TBT用于Qwen-235B的服务器，配备8个H200 GPU。

4.2.4 更高级的 GPU 和更大的大语言模型。为了展示 MuxWise 的有效性，我们在配备 8 块 H100 GPU 的服务器上使用 Llama-8B 和 Llama-70B，以及在配备 H200 GPU 的服务器上使用 Qwen-235B 对其进行评估。在这个实验中，我们仅将 MuxWise 与分块预填充进行比较。LoongServe 不支持新的 MoE 模型（如 Qwen-235B），即使每个 H200 GPU 拥有 141 GB 的显存，解耦服务解决方案对 Qwen-235B 也同样不可行。图16 显示了实验结果。在所有情况下，MuxWise 实现了平均 2.28×在 99%-*ile*TTFT 上的平均加速在99%-*ile*TBT 上的平均加速。这些持续的性能提升展示了 MuxWise 的服务范式在不同硬件和更大、更新的大语言模型上的普适性。

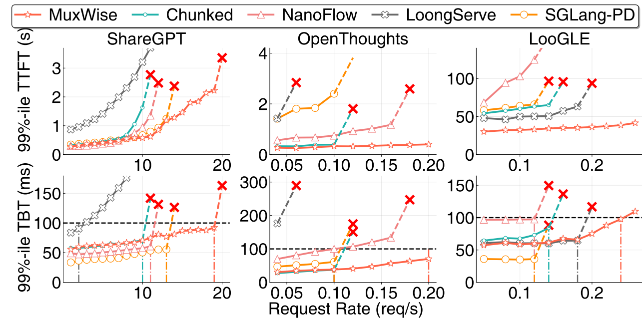


Figure 17. 99%*-ile* TTFT and TBT with Llama-70B on three types of synthetic workloads.

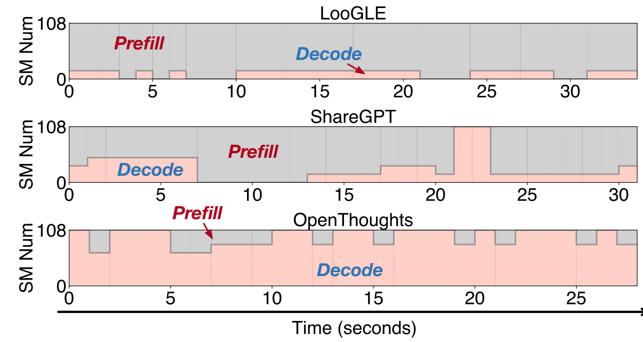


Figure 18. Change in compute partition between prefill and decode on LooGLE, ShareGPT, and OpenThoughts. Figures are sorted in descending order of prefill compute demand.

4.3 Evaluation on Diverse Synthetic Workloads

To better demonstrate MuxWise's effectiveness, we further evaluate it under three synthetic workloads. In the rest of the evaluation, we focus on Llama-70B due to space constraints, as results on other models are similar. Requests are generated by sampling inputs from ShareGPT [4], Openthoughts [17], and LooGLE [25], with arrival rates gradually increased following a Poisson process. Among these, only Openthoughts requests share a short system prompt. We select these workloads because they represent three typical patterns: moderate input and output, short input with ultra-long output, and ultra-long input with short output.

Figure 17 shows 99%*-ile* TTFT and TBT of MuxWise and three baselines. On ShareGPT, MuxWise achieves goodput improvements of 1.9×, 1.73×, 9.5×, 1.46× over chunked-prefill, NanoFlow, LoongServe, and SGLang-PD, respectively. On LooGLE, it achieves 1.71×, 2×, 1.33×, 2× over the four baselines. On Open-Thoughts, it achieves the same 2× improvement over chunked-prefill, NanoFlow and SGLang-PD, while Loongserve never meets SLO.

On ShareGPT, MuxWise, chunked-prefill, NanoFlow, and SGLang-PD all provide SLO guarantees at the beginning. SGLang-PD even achieves better TBT than MuxWise, as it statically reserves more compute. In contrast, MuxWise delivers shorter TTFT by reserving only best-fit SMs for decode. On OpenThoughts, LoongServe performs worse than the others, as it is designed for long-context workloads rather than requests with short inputs and long outputs.

NanoFlow outperforms chunked-prefill only on ShareGPT. On OpenThoughts, the system spends most of the time in the decode phase. Therefore, NanoFlow splits decode iterations to enable overlapping, leading to higher TBT than chunked-prefill. On LooGLE, it performs worse due to the small token budget used for long requests.

We also observe that SGLang-PD performs much worse on OpenThoughts and LooGLE than on ShareGPT. The causes differ across workloads. For OpenThoughts, since requests share little context, the system must still reserve slots for KV caches during prefill and decode. As the request rate increases, prefill stalls once the KV cache pool runs out of space. For LooGLE, only four GPUs are available for prefill, causing requests to queue in the prefill instance.

4.3.1 Short Requests and Single GPU.

Running Llama-8B on an A100 with ShareGPT, MuxWise improves goodput by 1.2× over chunked-prefill while maintaining similar TBT. This is because even when chunking rarely happens, satisfying a strict TBT SLO still forces chunked-prefill to use a small token budget, limiting GPU utilization and peak goodput. Notably, real-world conversation inputs are becoming significantly longer (e.g., 1.2K and 2.3K tokens in two recent conversation traces from cloud vendors [35, 41], compared with 226 tokens in the older ShareGPT dataset). The conversation in our evaluation is a multi-turn real-world trace, whose average length approaches to 7.5K. This trend is driven by the widespread adoption of techniques such as RAG [24], which increase the effective model input length by appending retrieved sequences to the user input.

4.4 Ablation Study

4.4.1 Scheduling details of different tasks.

We further evaluate MuxWise's dynamic scheduling of compute partitions by extracting scheduling details from runtime serving in §4.3. As shown in Figure 18, MuxWise makes different scheduling decisions for different workloads. On LooGLE, most SMs are allocated to prefill, while on OpenThoughts, MuxWise allocates the majority of SMs to decode. Results on ShareGPT lie between LooGLE and OpenThoughts. Overall, however, more SMs are allocated to prefill on ShareGPT, since decode is typically memory-bound and does not require as many SMs as prefill. Notably, we use Figure 18 to show that different partitions are required for different workloads. They are relatively static because the request rate is stable. In real-world traces, the workload could be bursty. Experimental results show that, during a bursty interval in Figure 13, MuxWise activated all the six configurations within 30s.

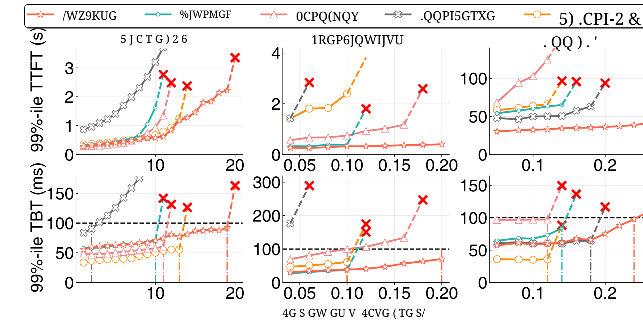


图17. 99%-分位 TTFT和TBT在Llama-70B上针对三种合成工作负载的性能表现。

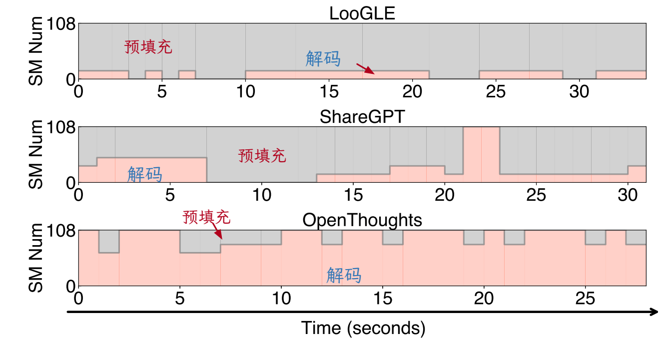


图18。在 LooGLE、ShareGPT 和 OpenThoughts 上，预填充与解码之间的计算分区变化。图按预填充计算需求降序排列。

4.3 在多样化合成工作负载上的评估

为了更好地展示 MuxWise 的有效性，我们在三种合成工作负载下进一步评估它。在后续评估中，由于篇幅限制，我们主要关注 Llama-70B，因为其他模型的结果相似。请求通过从 ShareGPT [4], OpenThoughts [17], 和 LooGLE [25], 中采样输入生成，其到达率逐渐增加，遵循泊松过程。在这些请求中，只有 OpenThoughts 请求共享一个简短的系统提示。我们选择这些工作负载，因为它们代表了三种典型模式：中等输入和输出、短输入与超长输出、超长输入与短输出。

图17展示了99%-ileTTFT和TBT的MuxWise和三个基线。在ShareGPT上，MuxWise 在分块预填充、NanoFlow、LoongServe和SGLang-PD上分别实现了 1.9×、1.73×、9.5×、1.46× 的吞吐量提升。在LooGLE上，它实现了 1.71×、2×、1.33×、2× 的吞吐量提升，超过四个基线。在Open-Thoughts上，它在分块预填充、NanoFlow和SGLang-PD上实现了相同的提升，而LoongServe从未满足SLO要求。

在 ShareGPT、MuxWise、分块预填充、NanoFlow 和 SGLang-PD 中，所有这些都从一开始就提供 SLO 保证。SGLang-PD 甚至比 MuxWise 实现了更好的 TBT，因为它静态预留了更多的计算资源。相比之下，MuxWise 通过仅预留最适合的 SMs 来解码，从而提供了更短的 TTFT。在 OpenThoughts 上，LoongServe 的表现比其他方案更差，因为它是为长上下文工作负载设计的，而不是为具有短输入和长输出的请求设计的。

NanoFlow 仅在 ShareGPT 上优于分块预填充。在 OpenThoughts 上，系统大部分时间都花在解码阶段。因此，NanoFlow 将解码迭代拆分以实现重叠，导致其 TBT 高于分块预填充。在 LooGLE 上，由于用于长请求的令牌预算较小，其表现更差。

我们还观察到SGLang-PD在OpenThoughts和LooGLE上的表现远不如ShareGPT。其原因因工作负载而异。对于OpenThoughts，由于请求共享很少的上下文，系统在预填充和解码期间仍需为KV缓存保留槽位。随着请求率的增加，当KV缓存池空间耗尽时，预填充就会停滞。对于LooGLE，只有四个GPU可用于预填充，导致请求在预填充实例中排队。

对于LooGLE，只有四个GPU可用于预填充，导致请求在预填充实例中排队。

4.3.1 短请求和单 GPU。

在 A100 上使用 ShareGPT 运行 Llama-8B 时，MuxWise 相比分块预填充将吞吐量提高了 1.2×，同时保持了相似的 TBT。这是因为即使分块很少发生，满足严格的 TBT SLO 仍然会迫使分块预填充使用较小的令牌预算，从而限制 GPU 利用率和峰值吞吐量。值得注意的是，现实世界的对话输入正变得显著更长（例如，来自云服务提供商的两个最近对话跟踪中分别为 1.2K 和 2.3K 个令牌，而旧的 ShareGPT 数据集中只有 226 个令牌）。我们评估中的对话是一个多轮现实世界跟踪，其平均长度接近 7.5K。这一趋势是由 RAG [24], 等技术的广泛应用推动的，这些技术通过将检索到的序列附加到用户输入中来增加有效模型输入长度。

4.4 消融研究

4.4.1 不同任务的调度细节。

我们进一步评估了MuxWise的动态计算分区调度，通过从§4.3的运行时服务中提取调度细节。如图18所示，MuxWiseWise针对不同的工作负载做出不同的调度决策。在LooGLE上，大多数SMs被分配给预填充，而在OpenThoughts上，MuxWise 将大多数SMs分配给解码。ShareGPT上的结果介于LooGLE和OpenThoughts之间。然而，总体而言，ShareGPT上分配给预填充的SMs更多，因为解码通常是内存绑定的，并且不需要像预填充那样多的SMs。值得注意的是，我们使用图18 来展示不同工作负载需要不同的分区。它们相对静态，因为请求率是稳定的。在真实世界的跟踪中，工作负载可能是突发的。实验结果表明，在图13的突发间隔内，MuxWise 在30秒内激活了所有六个配置。

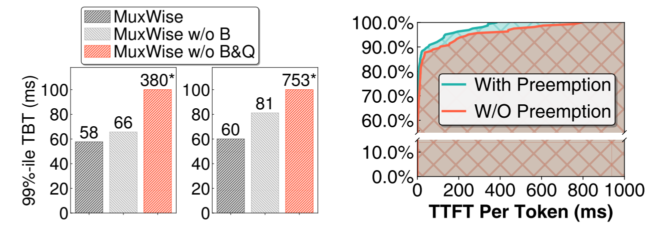


Figure 19. MuxWise with and without bubble-less multiplexing.

Figure 20. CDF of TTFT per token with and without preemption.

4.4.2 Effectiveness of Bubble-less Multiplex Engine.

As shown in Figure 9, bubbles commonly occur in the green context created for decode. In this experiment, we compare the TBT of MuxWise against its two variants. First, we disable layer-wise scheduling. Second, we further disable the query-based synchronization optimization. The workloads used are Tool&Agent under two different request rates.

Figure 19 presents the experimental results. As shown in the figure, disabling layer-wise execution slightly increases the TTFT of decode by approximately 10ms, which aligns with the typical kernel launch time for the prefill phase of Llama-70B. When query-based synchronization is further disabled, MuxWise suffers a significant degradation, 314ms for Llama-8B and 672ms for Llama-70B, due to frequent stalls waiting for the prefill phase to complete.

For further evaluation, we also collect the bubble ratio of MuxWise and chunked-prefill for the goodput results in Figure 15-a by profiling them with the NVIDIA Nsight Systems. The interval of the CUDA stream in the profiled timeline is treated as a bubble when it is not occupied by any GPU kernel. The bubble ratio is then defined as the proportion of all such bubbles in the compute stream. Since MuxWise has two active concurrent streams, we compute the bubble ratio for each stream and report their average as the final result. Notably, the bubble ratio is a temporal metric and does not reflect how GPU kernels utilize the parallel GPU resources they occupy.

MuxWise has a slightly higher bubble ratio (7.7% vs. 4.5%) due to its fine-grained kernel scheduling. These extra bubbles occurs when the system is purely processing decode iterations and all prefill layers are completed. Fortunately, these bubble do not degrade goodput, as there are no pending prefill launches and the decode iteration SLO is not violated. The reported GPU utilization in §4.2.3 also prove this.

4.4.3 Preemptive Scheduling for Long Request.

We evaluate the benefit of the bubble-less multiplex engine for preemptive scheduling by mixing requests from ShareGPT and LooGLE (50% each). Requests are generated at a rate of 0.5 per second following a Poisson process. Figure 20 shows the CDF of TTFT per token with and without preemptive scheduling. As shown, MuxWise achieves a 1.96× speedup on the 99%-*ile* TTFT per token, demonstrating that it can also be configured to support more advanced SLO-aware scheduling policies.

4.5 Overhead for Realizing PD-Multiplexing

Memory. MuxWise introduces some memory overhead by integrating GreenContext into existing serving systems. Creating a group of green contexts requires only 4MB, which is negligible compared to the total memory of modern GPUs. However, integrating it with CUDA Graph incurs a 6.2% overhead for both Llama-8B and Llama-70B on servers with 8 A100 or 8 H100 GPUs. This arises because the serving system records kernel launches for each decode-phase batch size into a CUDA Graph, consuming extra GPU memory. In MuxWise, there are six partition configurations in total, and each decode-phase compute partition created by GreenContext adds memory usage for all recorded batch sizes. Given the impressive performance gains of MuxWise, this overhead is acceptable.

Runtime. MuxWise splits the prefill phase into multiple prefill layers to enable bubble-less scheduling. This may introduce extra overhead due to fine-grained kernel launches. We conduct an experiment to compare full prefill launching with layer-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%.

5 Discussion

Generality of MuxWise. MuxWise generalizes to accelerators that support intra-process spatial sharing with lightweight dynamic adjustment, such as GreenContext [10] on NVIDIA GPUs (supported since the Pascal architecture) and `hipExtStreamCreateWithCUMask()` on AMD GPUs.

Contexts where MuxWise excels. MuxWise targets scenarios with strict SLO guarantees (e.g., a decode-phase SLO below 100*ms*). This is also the prevailing trend for achieving Model-as-a-Service in LLM serving. In this setting, MuxWise excels over existing works due to its efficient, fine-grained, and dynamic resource management between the prefill and decode phases. When the SLO target is loose or absent, such as in offline serving, MuxWise has no opportunity to outperform baselines such as chunked-prefill [55] or NanoFlow [54].

Large-scale deployment. While MuxWise is a single-instance optimization for high-goodput LLM serving, it can still benefit large-scale distributed deployments. In such deployments, MuxWise is complementary to disaggregated serving, as it optimizes each individual instance. Specifically, low-utilization decode instances could be replaced

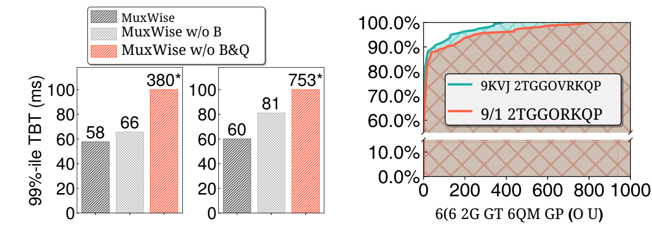


图19。 MuxWise带和无气泡多路复用。

图20。 每token的TTFT的CDF带和没有抢占。

4.4.2 无气泡多路复用引擎的有效性。如图9所示，气泡通常出现在为解码创建的绿色上下文中。在这个实验中，我们比较了MuxWise 的TBT与其两个变体。首先，我们禁用分层调度。其次，我们进一步禁用基于查询的同步优化。使用的工作负载是Tool&Agent在两种不同的请求率下。

图19展示了实验结果。如图所示，禁用逐层执行会略微增加解码的TTFT，约10毫秒，这与Llama-70B预填充阶段的典型内核启动时间一致。当基于查询的同步进一步禁用时，MuxWise 性能显著下降， 314毫秒对于Llama-8B和672毫秒对于Llama-70B，这是由于频繁等待预填充阶段完成的停顿。

为了进一步评估，我们还收集了MuxWise 和分块预填充在图15-a中的吞吐量结果的气泡率，通过NVIDIA Nsight Systems进行性能分析。在分析的CUDA流时间线中，当它没有被任何GPU内核占用时，该间隔被视为气泡。气泡率然后定义为所有此类气泡在计算流中的比例。由于MuxWise 有两个活跃并发流，我们为每个流计算气泡率，并报告它们的平均值作为最终结果。值得注意的是，气泡率是一个时间指标，并不反映GPU内核如何利用它们占用的并行GPU资源。

MuxWise 具有略微更高的气泡率（7.7% vs. 4.5%），这是由于其细粒度的内核调度。这些额外的气泡发生在系统仅处理解码迭代且所有预填充层都完成时。幸运的是，这些气泡不会降低吞吐量，因为没有任何待处理的预填充启动，且解码迭代SLO没有被违反。§4.2.3中报告的GPU利用率也证明了这一点。

4.4.3 长请求的抢占式调度。 我们通过混合 ShareGPT 和 LooGLE 的请求（各50%）来评估无气泡多路复用引擎在抢占式调度中的优势。请求按照泊松过程以每秒0.5个的速率生成。 图20 展示了每token任务完成时间（TTFT）在有和没有抢占式调度时的CDF。如图所示，MuxWise 实现了 1.96× 加速在99%-*ile* 每token任务完成时间上，证明了它也可以配置为支持更高级的SLO感知调度策略。

4.5 实现PD多路复用的开销

内存。 MuxWise通过将GreenContext集成到现有的服务系统中引入了一些内存开销。创建一组greencontexts只需要4MB，与现代GPU的总内存相比可以忽略不计。然而，将其与CUDA图集成会导致在具有8个A100或8个H100 GPU的服务器上，对于Llama-8B和Llama-70B，开销均为6.2%。这是因为服务系统将每个解码阶段批大小对应的内核启动记录到CUDA图中，从而消耗了额外的GPU内存。在MuxWise中，总共有六种分区配置，GreenContext创建的每个解码阶段计算分区都会为所有记录的批大小增加内存使用。鉴于MuxWise令人印象深刻的性能提升，这个开销是可以接受的。

运行时。 MuxWise 将预填充阶段拆分为多个预填充层以实现无气泡调度。这可能会由于细粒度内核启动引入额外的开销。我们进行了一项实验，比较了完整预填充启动与逐层启动，其中预填充阶段被拆分为最细的粒度。在各种具有不同批大小和上下文长度的配置下，总开销保持在 1。5% 以内。

5 讨论

MuxWise。 MuxWise可推广到支持进程内空间共享且具有轻量级动态调整功能的加速器，例如NVIDIA GPU上的GreenContext [10]（自Pascal架构起即支持）和AMD GPU上的hipExtStreamCreateWithCUMask()。

MuxWise在这些场景中表现出色。 MuxWise针对具有严格SLO保证的场景（例如，解码阶段的SLO低于 100*ms*）。这也是实现LLM服务中Model-as-a-Service的主流趋势。在这个设置中，MuxWise由于其在预填充和解码阶段之间的高效、细粒度和动态资源管理，优于现有工作。当SLO目标宽松或不存在时，例如在离线服务中，MuxWise没有机会超越chunked-prefill [55]或NanoFlow [54]等基线。

大规模部署。 虽然 MuxWise 是针对高吞吐量 LLM 服务的一个实例优化，但它仍然可以受益于大规模分布式部署。在这样的部署中，MuxWise 与解耦服务互补，因为它优化每个单独的实例。具体来说，低利用率的解码实例可以被替换

with MuxWise instances to exploit idle resources via spatially multiplexing prefill. If prefill instance serves short requests—resulting in low utilization–or multiplexing decode on it does not violate TTFT SLO, it can be also utilized for higher efficiency. However, when prefill instances consistently handle long requests (e.g., using chunked pipeline parallelism [34]) or decode multiplexing violates TTFT SLO, MuxWise offers limited benefit.

6 Related Work

Multiplexing in LLM Serving. There are also prior works [16, 29] that multiplex the prefill and decode phases in LLM serving. WindServe [16] multiplexes prefill and decode using a normal CUDA stream, which leads to uncontrollable contention. It also does not address bubbles during scheduling. Our prototype implementation of WindServe shows that, on ShareGPT, MuxWise achieves a 1.61× goodput improvement under a 50*ms* TBT SLO on an A100 with Llama-8B. Tropical [29] replaces the decode instance in disaggregated serving with temporally multiplexed prefill and decode. It launches a full prefill only when sufficient slack exists. When developing MuxWise, we implemented an enhanced temporal-only variant that splits prefill into layers to fit small slacks. It performs at least 20% worse than MuxWise because it cannot spatially leverage wasted resources. There are also two similar community works [18]. Semi-PD [18] utilizes MPS for multiplexing. MPS enables *inter-process* spatial sharing but requires process restarts to adjust SM allocations. Semi-PD mitigates this by introducing a resident process and two additional inference engines, which adds significant complexity to existing frameworks. Bullet [28] relies on `libsmctrl` [7] to control SM allocation. While Bullet claims that it can dynamically change the SMs allocated to each CUDA graph, our trials with Bullet's open-sourced implementation show that the SM allocation for each CUDA graph does not change. This also aligns with the claim in `libsmctrl` [7] that it does not work with CUDA graph.

Compute management in LLM serving. There are two main approaches to compute management for improving system throughput under SLO constraints: disaggregation-based and fusion-based methods. On the one hand, DistServe [53] and Splitwise [32] disaggregate LLM serving into separate prefill and decode instances, while LoongServe [44] improves adaptability by enabling dynamic switching between the two at runtime. On the other hand, chunk-prefill [2] splits the prefill phase into chunks and fuses each chunk with a decode iteration for execution. However, disaggregation-based methods incur significant resource waste due to the coupled management of compute and memory, whereas fusion-based methods fail to fully maximize system throughput under SLO constraints. In contrast, our work decouples compute and memory management and maximizes goodput through spatial multiplexing.

Memory management in LLM serving. To improve system throughput in multi-turn or context-heavy LLM workloads, several systems propose memory management techniques. PagedAttention [23] introduces a paged memory pool to enable KV cache reuse between prefill and decode phases. Parrot [26] and SGLang [52] leverage context-aware caching to maximize reuse of KV segments across requests. MuxWise enhances these approaches by preserving memory sharing across phases and requests.

Execution time modeling. Performance modeling under spatial sharing is highly challenging, and prior efforts[22, 36, 48, 49] mainly focus on predicting interference for specific operators. GPUlet[9] uses linear regression with L1 cache utilization and DRAM bandwidth as input features to estimate performance interference among colocated operators. HSM[50] and GDP[20] also adopt linear regression based on low-level metrics in the simulator for operator slowdown prediction.

Compute partition techniques. Existing GPU partitioning techniques can be broadly categorized into time-sharing and space-sharing approaches. Time-sharing is typically implemented via API remoting [8, 27]. However, time-sharing alone is insufficient to meet MuxWise's requirements, as the prefill and decode phases already interleave in a time-sharing manner. In contrast, NVIDIA provides MPS [13], MIG [12], and GreenContext [10] for spatial sharing. MPS and MIG support inter-process spatial multiplexing, while GreenContext [10] enables intra-process spatial multiplexing with precise SM partition [11]. MuxWise builds on GreenContext to implement its PD multiplexing approach.

7 Conclusion

LLM services requires high goodput, yet existing serving systems struggle due to various deficiencies. To address these issues, we present MuxWise, an LLM serving framework with high goodput. MuxWise leverages a promising new serving paradigm, intra-GPU PD multiplexing, to achieve more flexible compute management for prefill and decode phases in LLM serving. Experiments show that MuxWise improves goodput by 2.2× on average over state-of-the-art baselines. Despite the notable performance improvement, MuxWise also introduces a simple yet effective design for current LLM serving systems. We plan to open-source MuxWise after publication.

Acknowledgments

This work is partially sponsored by the National Key Research and Development Program of China (2024YFB4505700), National Natural Science Foundation of China (62232011) and Natural Science Foundation of Shanghai Municipality (24ZR1430500). We thank the anonymous reviewers for their constructive feedback and suggestions.

以MuxWise 实例来利用空闲资源，通过空间多路复用预填充。如果预填充实例服务短请求——导致低利用率——或者在其上进行的解码多路复用不违反 TTFT SLO，它也可以用于更高的效率。然而，当预填充实例持续处理长请求（例如，使用分块流水线并行 [34]）或解码多路复用违反 TTFT SLO 时，MuxWise提供有限的益处。

6 相关工作

***LLM*服务中的多路复用。** 也有先前的工作 [16, 29] 在LLM服务中多路复用预填充和解码阶段。WindServe [16] 使用普通的CUDA流来多路复用预填充和解码，这会导致不可控的容争。它也没有解决调度过程中的气泡。我们的WindServe原型实现表明，在ShareGPT上，MuxWise在A100上使用Llama-8B时，在 50*ms* TBT SLO下实现了 1.61× 吞吐量提升。Tropical [29] 用时间多路复用的预填充和解码替换了分散服务的解码实例。它只有在存在足够松弛时才会启动完整的预填充。在开发MuxWise时，我们实现了一个增强的时间仅变体，它将预填充分层以适应小松弛。它的性能至少比MuxWise差20%，因为它无法空间利用浪费的资源。还有两个类似的社区工作 [18]。Semi-PD [18] 使用MPS进行多路复用。MPS支持进程间 空间共享，但需要进程重启来调整 SM 分配。Semi-PD 通过引入一个驻留进程和两个额外的推理引擎来缓解这个问题，这给现有框架增加了显著复杂性。Bullet [28] 依赖于 libsmctrl [7] 来控制SM分配。虽然Bullet声称它可以动态更改分配给每个CUDA图 的SM，但我们使用Bullet开源实现进行的试验表明，每个CUDA图的SM分配没有改变。这也与libsmctrl [7] 中声称它不与CUDA图兼容的说法一致。

***LLM*服务中的计算管理。** 在SLO约束下提高系统吞吐量的计算管理主要有两种方法：解耦方法和融合方法。一方面，Dist- Serve [53] 和Splitwise [32] 将LLM服务解耦为独立的预填充和解码实例，而LoongServe [44]通过在运行时启用两者之间的动态切换来提高适应性。另一方面，chunk-prefill [2]将预填充阶段拆分为块，并将每个块与解码迭代融合以执行。然而，解耦方法由于计算和内存的耦合管理导致资源浪费严重，而融合方法在SLO约束下无法充分最大化系统吞吐量。相比之下，我们的工作将计算和内存管理解耦，并通过空间多路复用最大化吞吐量。

***LLM*服务中的内存管理。** 为了提高多轮或上下文密集型LLM工作负载中的系统吞吐量，一些系统提出了内存管理技术。PagedAttention [23] 引入了一个分页内存池，以在预填充和解码阶段之间实现KV缓存重用。Parrot [26] 和SGLang [52] 利用上下文感知缓存来最大化跨请求的KV段重用。MuxWise通过跨阶段和请求保留内存共享来增强这些方法。

执行时间建模。 在空间共享下的性能建模极具挑战性，且先前工作[22, 36,48, 49] 主要集中于预测特定操作员的干扰。GPUlet[9] 使用以L1缓存利用率和DRAM带宽为输入特征的线性回归来估计共位操作员之间的性能干扰。HSM[50] 和GDP[20] 也采用基于模拟器中低级指标的线性回归进行操作员减速预测。

计算分区技术。 现有的GPU分区技术可以大致分为时间共享和空间共享方法。时间共享通常通过API远程调用 [8, 27] 实现。然而，仅靠时间共享不足以满足MuxWise的要求，因为预填充和解码阶段已经以时间共享的方式交错进行。相比之下，NVIDIA提供了MPS [13],MIG [12],和GreenContext [10]用于空间共享。MPS和MIG支持进程间空间多路复用，而GreenContext [10] 则通过精确的SM分区 [11] 实现进程内空间多路复用。MuxWise基于GreenContext实现了其PD多路复用方法。

7 结论

大语言模型服务需要高吞吐量，但现有的服务系统由于各种缺陷而难以满足要求。为了解决这些问题，我们提出了MuxWise，一个具有高吞吐量的大语言模型服务框架。MuxWise利用一种有前景的新服务范式——GPU内PD多路复用，为LLM服务中的预填充和解码阶段实现更灵活的计算管理。实验表明，MuxWise在平均上比当前最先进基线提高了 2.2× 的吞吐量。尽管性能提升显著，MuxWise还为当前的大语言模型服务系统引入了一种简单而有效的设计方案。我们计划在发表后开源Mux Wise。

致谢

这项工作得到了中国国家重点研发计划（2024YFB4505700）、国家自然科学基金（62232011）和上海市自然科学基金（24ZR1430500）的部分资助。我们感谢匿名审稿人为其提出的建设性反馈和建议。

A Artifact Appendix

A.1 Abstract

MuxWise is an LLM serving framework adopting intra-GPU prefill-decode multiplexing, which is built on the top of SGLang[52]. We provide the source code of MuxWise and scripts to reproduce comparison of chunked-prefill. This appendix includes instructions for reproducing similar data in Figure 16 and Figure 17.

A.2 Artifact check-list (meta-information)

- **Model:** CodeLlama-34b-Instruct-hf.
- **Data set:** ShareGPT [4] and LooGLE [25].
- **Hardware:**
  NVIDIA H200 NVL (140 GB, 132 SMs)
  NVIDIA driver: 580.65.06 (must be greater than 570)
- **Experiments:** This appendix provides instrucitons for comparing 99%-*ile* TTFT and 99%-*ile* TBT MuxWise between MuxWise and chunked-prefill under various workload.
- **Metrics:** 99%-*ile* TTFT, 99%-*ile* TBT
- **Output:** Jsonl files containing metrics from MuxWise and chunked-prefill with different chunk size.
- **How much disk space required (approximately)?:** Approximately 200GB
- **How much time is needed to prepare workflow (approximately)?:** About 10 minutes to build from source code.
- **How much time is needed to complete experiments (approximately)?:** About 2 hours for ShareGPT workload and 4 hours for LooGLE workload.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to access.
The source code of MuxWise is available for download on Zenodo: https://zenodo.org/records/18062118. The pre-built Docker image can be found in: https://hub.docker.com/layers/combathhhhhh/pdmux/sglpr_torch2.6_bench

A.3.2 Hardware dependencies.
Requires an x86-64 Linux host with at least 200 GB of free disk space, and an NVIDIA H200 NVL GPU (140 GB, 132 SMs).

A.3.3 Software dependencies.
NVIDIA driver: 580.65.06 (must be greater than 570).

A.3.4 Data sets.
ShareGPT: chatbot tasks, with an average input length of 226 and average output length of 195.

LooGLE: long-context understanding tasks, with an average input length of 30k and average output length of 15.

A.3.5 Models.
CodeLlama-34b-Instruct-hf.

A.4 Installation

Please follow the instructions below, which are adapted from our GitHub repository (https://github.com/ykcombat/sglang/tree/slo_config):

```
# 1. Clone the repository and switch to the slo_config branch
git clone https://github.com/ykcombat/sglang.git
cd sglang
git checkout slo_config

# 2. Build SGLang
pip install --upgrade pip
pip install -e "python"
```

A.5 Experiment workflow

Our experiments focus on comparison between MuxWise and chunked-prefill.

1. Download the required LLM model(CodeLlama-34b-Instruct-hf) to /workspace/data.
2. Start MuxWise or chunked-prefill server. You can change the environment virable $CHUNK_SIZE to start chunked-prefill server with different token budgets.

```
# 1. Start MuxWise Server
./start_pdmux.sh

# 2. Start Chunked-prefill Server
./start_chunk.sh
```

3. Start evaluating in another terminal.

```
# 1. Evaluate MuxWise on ShareGPT and LooGLE
./bench_pdmux.sh

# 2. Evaluate Chunked-prefill on ShareGPT and LooGLE
./bench_chunk.sh
```

A.6 Evaluation and expected results

When all experiments done, you will obtain jsonl files containing detailed metrics under /workspace/sglang. To visualize the results, run plot.ipynb; this will generate figures similar to the reference plot provided at /workspace/sglang/H200_result.png. Please note that results may vary depending on the specific hardware used. You can refer to https://github.com/ykcombat/sglang/blob/slo_config/README.md for more information.

A.7 Notes

When serving different workloads, different configurations are used, which can be found in our repository.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 7, 18 pages.

A 工具附录

A.1 摘要

MuxWise是一个采用GPU内部预填充-解码多路复用的大语言模型服务框架，它构建在SGLang之上[52]。我们提供了MuxWise 的源代码以及用于重现分块预填充比较的脚本。本附录包含在图16和图17中重现类似数据的说明。

A.2 工具检查清单（元信息）

- 模型: CodeLlama-34b-Instruct-hf。
- **数据集：** ShareGPT [4]和 LooGLE [25].
- **硬件：** NVIDIA H200 NVL (140 GB, 132 SMs)
  NVIDIA驱动程序： 580.65.06 (必须大于570)
- **实验：** 本附录提供了在各种工作负载下比较99%-分位数 TTFT和99%-分位数 TBT MuxWise与MuxWise和分块预填充的说明。
- **指标：** 99%-分位数TTFT,99%-分位数TBT
- **输出：** 包含不同块大小下MuxWise和分块预填充指标的jsonl文件。
- **需要多少磁盘空间（大约）？：** 大约200GB
- **需要多长时间来准备工作流（大约）？：** 从源代码构建大约需要 10 分钟。
- **完成实验需要多长时间（大约）？：** ShareGPT 工作负载大约需要 2 小时，LooGLE 工作负载大约需要 4 小时。
- **是否公开可用？：** 是。

A.3 描述

A.3.1 如何访问。 MuxWise的源代码可在 Zenodo 上下载：https://zenodo.org/records/18062118。预构建的 Docker 镜像可在以下地址找到：https://hub.docker.com/layers/combathhhhhh/pdmux/sglpr_torch2.6_bench_

A.3.2 硬件依赖。 需要 x86-64 Linux 主机，至少 200 GB 的可用磁盘空间，以及一个 NVIDIA H200 NVL GPU（140 GB，132 个 SMs）。

A.3.3 软件依赖。 NVIDIA驱动：580.65.06（必须大于570）。

A.3.4 数据集。 ShareGPT：聊天机器人任务，平均输入长度为226，平均输出长度为195。

LooGLE：长文本理解任务，平均输入长度为30k，平均输出长度为15。

A.3.5 模型。 CodeLlama-34b-Instruct-hf。

A.4 安装

请遵循以下说明，这些说明改编自我们的GitHub仓库（https://github.com/ykcombat/sglang/tree/slo_配置）：

1. 克隆仓库并切换到 slo_config 分支
git clone https://github.com/ykcombat/sglang.git
cd sglang git checkout slo_config

2. 安装 SGLang pip install --upgrade pip pip install -e " python"

A.5 实验工作流程

我们的实验重点比较 MuxWise 和分块预填充。

1. 将所需的 LLM 模型（CodeLlama-34b-Instruct-hf）下载到 /workspace/data。 2. 启动 MuxWise 或分块预填充服务器。您可以更改环境变量 $CHUNK_SIZE 以使用不同的 token 预算启动分块预填充服务器。

1. 启动 MuxWise 服务器 ./start_pdmux.sh

2. 启动分块预填充服务器 ./start_chunk.sh

3. 在另一个终端开始评估。

1. 在 ShareGPT 和 LooGLE 上评估 MuxWise ./bench_pdmux.sh

2. 在 ShareGPT 和 LooGLE 上评估分块预填充 ./bench_chunk.sh

A.6 评估和预期结果

当所有实验完成后，你将在 /workspace/sglang 下获得包含详细指标的 jsonl 文件。要可视化结果，请运行 plot.ipynb；这将生成类似于 /workspace/sglang/H200_result.png 提供的参考图的图像。请注意，结果可能会因使用的具体硬件而异。你可以参考 https://github.com/ykcombat/sglang/blob/slo_config/README.md获取更多信息。

A.7 备注

在服务不同工作负载时，会使用不同的配置，这些配置可以在我们的仓库中找到。

参考文献

[1] Amey Agrawal、Nitin Kedia、Ashish Panwar、Jayashree Mohan、Nipun Kwatra、Bhargav S. Gulavani、Alexey Tumanov 和 Ramachandran Ramjee。2024年。使用 sarathi-serve 调控 LLM 推理中的吞吐量-延迟权衡。在 第18届*USENIX*操作系统设计与实现会议论文集(圣克拉拉，CA，USA) *(OSDI'24)*。USENIX协会，美国，第7篇论文，18页。

[2] 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]

[3] Michael Andersch, Greg Palmer, Ronny Krashinsky, Nick Stam, Vishal Mehta, Gonzalo Brito, and Sridhar Ramaswamy. 2025. NVIDIA Hopper Architecture In-Depth – Thread block clusters. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/#thread_block_clusters. Accessed 2026-01-10.

[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, Xiaodeng 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). arXiv:2309.16609 [cs] doi:10.48550/arXiv.2309.16609

[6] Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, et al. 2024. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv preprint arXiv:2412.15204* (2024).

[7] Joshua Bakita and James H Anderson. 2023. Hardware compute partitioning on NVIDIA GPUs. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 54–66.

[8] 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. doi:10.1145/2872362.2872368

[9] 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.

[10] 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.

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

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

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

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

[15] 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

[16] Jingqi Feng, Yukai Huang, Rui Zhang, Sicheng Liang, Ming Yan, and Jie Wu. 2025. WindServe: Efficient Phase-Disaggregated LLM Serving with Stream-based Dynamic Scheduling. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1283–1295. doi:10.1145/3695053.3730999

[17] Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, Marianna Nezhurina, Jean Mercat, Trung Vu, Zayne Sprague, Ashima Suvarna, Benjamin Feuer, Liangyu Chen, Zaid Khan, Eric Frankel, Sachin Grover, Caroline Choi, Niklas Muennighoff, Shiye Su, Wanjia Zhao, John Yang, Shreyas Pimpalgaonkar, Kartik Sharma, Charlie Cheng-Jie Ji, Yichuan Deng, Sarah Pratt, Vivek Ramanujan, Jon Saad-Falcon, Jeffrey Li, Achal Dave, Alon Albalak, Kushal Arora, Blake Wulfe, Chinmay Hegde, Greg Durrett, Sewoong Oh, Mohit Bansal, Saadia Gabriel, Aditya Grover, Kai-Wei Chang, Vaishaal Shankar, Aaron Gokaslan, Mike A. Merrill, Tatsunori Hashimoto, Yejin Choi, Jenia Jitsev, Reinhard Heckel, Maheswaran Sathiamoorthy, Alexandros G. Dimakis, and Ludwig Schmidt. 2025. OpenThoughts: data recipes for reasoning models. doi:10.48550/arXiv.2506.04178 arXiv:2506.04178 [cs].

[18] Ke Hong, Lufang Chen, Zhong Wang, Xiuhong Li, Qiuli Mao, Jianping Ma, Chao Xiong, Guanyu Wu, Buhe Han, Guohao Dai, Yun Liang, and Yu Wang. 2025. semi-PD: Towards Efficient LLM Serving via Phase-Wise Disaggregated Computation and Unified Storage. arXiv:2504.19867 [cs.CL] https://arxiv.org/abs/2504.19867

[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] Aditya K. Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. 2025. POD-attention: unlocking full prefill-decode overlap for faster LLM inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 897–912. doi:10.1145/3676641.3715996

[22] 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. doi:10.1007/s10586-021-03429-7

[23] 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. doi:10.1145/3600006.3613165

[24] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[25] 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

[26] 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.

[27] 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.

[28] Zejia Lin, Hongxin Xu, Guanyi Chen, Zhiguang Chen, Yutong Lu, and Xianwei Zhang. 2025. Boosting LLM Serving through Spatial-Temporal GPU Resource Sharing. arXiv:2504.19516 [cs.DC] https://arxiv.org/abs/2504.19516

[29] Jinming Ma, Jiefei Chen, Xiuhong Li, Jiangfei Duan, Haojie Duanmu, Xingcheng Zhang, Chao Yang, and Dahua Lin. 2025. *Tropical: Enhancing SLO Attainment in Disaggregated LLM Serving via SLO-Aware Multiplexing*. IEEE Press. https://doi.org/10.1109/DAC63849.2025.11132617

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

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein,

[2] Amey Agrawal、Ashish Panwar、Jayashree Mohan、Nipun Kwatra、Bhargav S. Gulavani以及Ramachandran Ramjee。2023年。SARATHI：通过分块预填充搭便车解码实现高效的LLM推理。arXiv:2308.16369（2023年8月）。arXiv:2308.16369 [cs][3] Michael Andersch、Greg Palmer、Ronny Krashinsky、Nick Stam、Vishal Mehta、Gonzalo Brito以及Sridhar Ramaswamy。2025年。NVIDIA Hopper架构深度解析——线程块集群。https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/#thread_block_clusters。访问于2026-01-10。[4] anon8231489123。2023年。ShareGPT Vicuna未过滤——清理分割（v3）。https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna unfiltered/resolve/main/ShareGPT V3 unfiltered cleaned _split.json。访问于：2025-04-16.[5] Jinze Bai、Shuai Bai、Yunfei Chu、Zeyu Cui、Kai Dang、Xiaodeng Deng、Yang Fan、Wenbin Ge、Yu Han、Fei Huang、Binyuan Hui、Luo Ji、Mei Li、Junyang Lin、Runji Lin以及等人。2023年。Qwen技术报告。arXiv:2309.16609（2023年9月）。arXiv:2309.16609 [cs] doi:10.48550/arXiv. 2309.16609[6] Yushi Bai、Shangqing Tu、Jiajie Zhang、Hao Peng、Xiaozhi Wang、Xin Lv、Shulin Cao、Jiazheng Xu、Lei Hou、Yuxiao Dong以及等人。2024年。LongBench v2：迈向对现实长上下文多任务更深入的理解和推理。arXiv预印本arXiv:2412.15204（2024年）。[7] Joshua Bakita 和 James H Anderson。2023 年。NVIDIA GPU 上的硬件计算分区。在2023 IEEE 29th实时与嵌入式技术与应用会议（RTAS）。IEEE，54–66。[8] Quan Chen、Hailong Yang、Jason Mars以及Lingjia Tang。2016年。Bay- max：为仓库规模计算机中非抢占式加速器提供QoS感知和利用率提升。在编程语言与操作系统架构支持第二十一届国际会议论文集。ACM，亚特兰大，乔治亚，美国，681–696。doi:10.1145/2872362.2872368[9] Seungbeom Choi、Sunho Lee、Yeonjae Kim、Jongse Park、Youngjin Kwon以及Jaehyuk Huh。2022年。在{多GPU}服务器上使用{时空}共享服务异构机器学习模型。在2022 USENIX年度技术会议（USENIX ATC 22）。199– 216.[10] NVIDIA Corporation。2025年。CUDA驱动程序API：绿色上下文。https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA_ GREEN CONTEXTS.html。访问于：2025-03-29.[11] NVIDIA Corporation。2025年。CUDA运行时API：流管理。https://docs.nvidia.com/cuda/cuda-runtime-api/group_ CUDART STREAM.html。访问于2025-03-30.[12] NVIDIA Corporation。2025年。多实例GPU（MIG）。https://www.nvidia.com/en-sg/technologies/multi-instance-gpu/. 访问于：2025-03-30.[13] NVIDIA Corporation。2025年。多进程服务。版本570。[14] NVIDIA Corporation。2025年。NVIDIA Dynamo：一个数据中心规模分布式推理服务框架。https://github.com/ai- dynamo/dynamo。访问于：2025-04-07。[15] Abhimanyu Dubey、Abhinav Jauhri、Abhinav Pandey、Abhishek Ka- dian、Ahmad Al-Dahle、Aiesha Letman、Akhil Mathur、Alan Schelten、Amy Yang、Angela Fan、Anirudh Goyal、Anthony Hartshorn、Aobo Yang、Archi Mitra、Archie Sravankumar以及等人。2024年。Llama 3模型群。arXiv:2407.21783（2024年8月）。arXiv:2407.21783[16] Jingqi Feng、Yukai Huang、Rui Zhang、Sicheng Liang、Ming Yan以及Jie Wu。2025年。WindServe：基于流的动态调度分阶段解耦LLM服务。在计算机架构第52届年度国际会议论文集（ISCA '25）。Association for Computing Machinery，纽约，纽约，美国，1283–1295。doi:10.1145/3695053.3730999[17] Etash Guha、Ryan Marten、Sedrick Keh、Negin Raoof、Georgios Smyr- nis、Hritik Bansal、Marianna Nezhurina、Jean Mercat、Trung Vu、Zayne Sprague、Ashima Suvarna、Benjamin Feuer、Liangyu Chen、Zaid Khan、Eric Frankel、Sachin Grover、Caroline Choi、Niklas Muennighoff、Shiye

苏婉佳，赵，约翰·杨，Shreyas·Pimpalgaonkar，Kartik·Sharma，Charlie·Cheng-Jie·Ji，邓一川，Sarah·Pratt，Vivek·Ramanujan，Jon·Saad-Falcon，Jeffrey·Li，Achal·Dave，Alon·Albalak，Kushal·Arora，Blake·Wulfe，Chinmay·Hegde，Greg·Durrett，Oh·Sewoong，Bansal·Mohit，Saa-dia·Gabriel，Aditya·Grover，Chang·Kai-Wei，Shankar·Vaishaal，Gokaslan·Aaron，Merrill·Mike·A.，Hashimoto·Tatsunori，Choi·Yejin，Jitsev·Jenia，Heckel·Reinhard，Sathiamoorthy·Maheswaran，Dimakis·Alexandros·G.，Schmidt·Ludwig。2025。OpenThoughts：推理模型的数據配方。doi:

10 48550 2506 04178 2506 04178 。/arXiv。.

[ ]

arXiv: . cs。[18] Ke·Hong，Chen·Lufang，Wang·Zhong，Li·Xiuhong，Mao·Qiuli，Ma·Jian-ping，Xiong·Chao，Wu·Guanyu，Han·Buhe，Dai·Guohao，Liang·Yun，Wang·Yu。2025。semi-PD：通過階段式分離計算和統一存儲實現高效的LLM服務。arXiv:

2504 19867 [ ] 2504 19867 。 cs.CL

[19] 2025

https://arxiv.org/abs/ . Anysphere·Inc。 . Cursor：AI代碼編輯器。https://www.cursor.com/。訪問：2025-04-05.[20] Jahre·Magnus和Eeckhout·Lieven。[n。]。Gdp：利用數據流屬性在運行時精確估計無干擾性能。在IEEE高性能計算機架構國際會議（HPCA 2018）中。296–309.[21] Kamath·Aditya·K.，Prabhu·Ramya，Mohan·Jayashree，Peter·Simon，Ramjee·Ramachandran，Panwar·Ashish。2025。POD-attention：釋放完整的預填充-解碼重疊以實現更快的LLM推理。在ACM國際計算機結構支持程式設計語言和操作系統會議第30屆（ASPLOS '25）文獻集中，第2卷（Volume 2）。Association for Computing Machinery，New York，NY，USA，897–912。doi:10.1145/3676641.3715996[22] Kim·Sejin和Kim·Yoonhee。2022。K-Scheduler：基於GPU的執行配置文件動態內部SM多任務管理。Cluster Computing 25，1（2022年2月），597–617。doi:10.1007/s10586-021-03429-7[23] Kwon·Woosuk，Li·Zhuohan，Zhuang·Siyuan，Sheng·Ying，Zheng·Lianmin，Yu·Cody·Hao，Gonzalez·Joseph，Zhang·Hao，Sto- ica·Ion。2023。針對LLM服務的高效內存管理。在操作系統原理第29屆會議文獻集中。ACM，Koblenz Germany，611–626。doi:10.1145/3600006.3613165[24] Lewis·Patrick，Perez·Ethan，Piktus·Aleksandra，Petroni·Fabio，Karpukhin·Vladimir，Goyal·Naman，Küttler·Heinrich，Lewis·Mike，Yih·Wen-tau，Rocktäschel·Tim，等。2020。知識密集型NLP任務的檢索增強生成。神經信息處理系統進展33（2020年），9459–9474。[25] Li·Jiaqi，Wang·Mengmeng，Zheng·Zilong，Zhang·Muhan。2024。LooGLE：長文本語言模型能否理解長文本？arXiv:2311.04939 [cs.CL] https://arxiv.org/abs/2311.04939[26] Lin·Chaofan，Han·Zhenhua，Zhang·Chengruidong，Yang·Yuqing，Yang·Fan，Chen·Chen，Qiu·Lili。2024。Parrot：使用語義變量高效服務LLM基應用程序。在USENIX操作系統設計與實施第18屆會議（OSDI 24）。929–945。[27] Lin·Yu-Shiang，Lin·Chun-Yuan，Lee·Che-Rung，Chung·Yeh-Ching。2019。qcuda：Gpgpu虛擬化以實現高帶寬效率。在2019 IEEE雲計算技術與科學國際會議（CloudCom）中。IEEE，95–102。[28] Lin·Zejia，Xu·Hongxin，Chen·Guanyi，Chen·Zhiguang，Lu·Yutong，Zhang·Xianwei。2025。通過空間-時間GPU資源共享加強LLM服務。arXiv:2504.19516 [cs.DC] https: // arxiv.org/abs/2504.19516[29] Ma·Jinming，Chen·Jiefei，Li·Xiuhong，Duan·Jiangfei，Duanmu·Haojie，Zhang·Xingcheng，Yang·Chao，Lin·Dahua。2025。Tropical：通過SLO感知多路復用提高分離LLM服務的SLO達成。IEEE Press。https://doi/

10 1109 63849 2025 11132617 。 /

[30] 2025 2025 04 05

DAC 。 . OpenAI。 . ChatGPT。https://chatgpt.com/。訪問： - - 。[31] Paszke·Adam，Gross·Sam，Massa·Francisco，Lerer·Adam，Bradbury·James，Chanan·Gregory，Killeen·Trevor，Lin·Zeming，Gimelshein·Natalia，

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*. Curran Associates Inc., Red Hook, NY, USA, 8026–8037.

[32] 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. doi:10.1109/ISCA59077.2024.00019

[33] 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

[34] 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.

[35] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. DynamoLLM: designing LLM inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1348–1362. doi:10.1109/HPCA61900.2025.00102 ISSN: 2378-203X.

[36] 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. doi:10.1145/3627703.3629578

[37] 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). arXiv:2302.13971 doi:10.48550/arXiv.2302.13971

[38] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[39] 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. arXiv:1706.03762 doi:10.48550/arXiv.1706.03762

[40] vLLM Team. [n. d.]. CUDA Graphs — vLLM Design Documentation. https://docs.vllm.ai/en/stable/design/cuda_graphs/. Accessed: 2026-01-09.

[41] Jiahao Wang, Jinbo Han, Xingda Wei, Sijie Shen, Dingyan Zhang, Chenguang Fang, Rong Chen, Wenyuan Yu, and Haibo Chen. 2025. KVCache cache in the wild: characterizing and optimizing KVCache cache at a large cloud provider. 465–482. https://www.usenix.org/conference/atc25/presentation/wang-jiahao

[42] Zhibin Wang, Shipeng Li, Yuhang Zhou, Xue Li, Zhonghui Zhang, Nguyen Cam-Tu, Rong Gu, Chen Tian, Guihai Chen, and Sheng Zhong. 2025. Revisiting Service Level Objectives and System Level Metrics in Large Language Model Serving. arXiv:2410.14257 [cs.LG] https://arxiv.org/abs/2410.14257

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[44] 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. doi:10.1145/3694715.3695948

[45] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. 2025. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 94–109. doi:10.1145/3689031.3696098

[46] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. 2025. FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving. In *Eighth Conference on Machine Learning and Systems*.

[47] 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.

[48] 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. doi:10.1145/3689031.3696070

[49] 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. doi:10.1145/3330345.3330351

[50] 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.

[51] 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.

[52] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 2000, 27 pages.

[53] 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

Luca Antiga，Alban Desmaison，Andreas Köpf，Edward Yang，Zach DeVito，Martin Raison，以及等人。2019年。PyTorch：一种命令式风格、高性能的深度学习库。在神经信息处理系统国际会议第33届论文集。Cur- ran Associates Inc.，美国纽约州Red Hook，8026–8037.[32] Pratyush Patel，Esha Choukse，Chaojie Zhang，Aashaka Shah，Íñigo Goiri，Saeed Maleki，以及Ricardo Bianchini。2024年。Split wise：使用阶段分割进行高效的生成式大语言模型推理。在2024年ACM/IEEE第51届年度国际计算机架构 symposium (ISCA)。118–132。doi:10.1109/ISCA59077.2024.00019[33] Zhenting Qi，Mingyuan Ma，Jiahang Xu，Li Lyna Zhang，Fan Yang，以及Mao Yang。2024年。相互推理使更小的大语言模型成为更强的解决问题者。arXiv:2408.06195 (2024年8月)。arXiv:2408.06195[34] Ruoyu Qin，Zheming Li，Weiran He，Jialei Cui，Feng Ren，Mingxing Zhang，Yongwei Wu，Weimin Zheng，以及Xinran Xu。2025年。Mooncake：用更多存储空间换取更少计算——一种以KVCache为中心的大语言模型聊天机器人服务架构。在23rd USENIX文件和存储技术会议 (FAST 25)。155–170.[35] Jovan Stojkovic，Chaojie Zhang，Íñigo Goiri，Josep Torrellas，以及Esha Choukse。2025年。DynamoLLM：为性能和能效设计的大语言模型推理集群。在2025年IEEE高性能计算机架构 symposium (HPCA)。1348–1362。doi:10.1109/HPCA61900.2025.00102 ISSN: 2378-203X.[36]Foteini Strati，Xianzhe Ma，以及Ana Klimovic。2024年。Orion：针对机器学习应用的感知干扰、细粒度GPU共享。在欧洲计算机系统会议第19届论文集(EuroSys '24)。Association for Computing Machinery，美国纽约州纽约，1075–1092。doi:10.1145/3627703.3629578[37] 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，以及Guillaume Lample。2023年。LLaMA：开放且高效的基础语言模型。arXiv:2302.13971 (2023年2月)。arXiv:2302.13971 doi:10.48550/arXiv.2302.13971[38] Hugo Touvron，Louis Martin，Kevin Stone，Peter Albert，Amjad Alma- hairi，Yasmine Babaei，Nikolay Bashlykov，Soumya Batra，Prajwal Bhargava，Shruti Bhosale，Dan Bikel，Lukas Blecher，Cristian Canton Ferrer，Moya Chen，Guillem Cucurull，David Esiobu，Jude Fernandes，Jeremy Fu，Wenyin Fu，Brian Fuller，Cynthia Gao，Vedanuj Goswami，Na man Goyal，Anthony Hartshorn，Saghar Hosseini，Rui Hou，Hakan Inan，Marcin Kardas，Viktor Kerkez，Madian Khabsa，Isabel Kloumann，Artem Korenev，Punit Singh Koura，Marie-Anne Lachaux，Thibaut Lavril，Jenya Lee，Diana Liskovich，Yinghai Lu，Yuning Mao，Xavier Martinet，Todor Mihaylov，Pushkar Mishra，Igor Molybog，Yixin Nie，Andrew Poulton，Jeremy Reizenstein，Rashi Rungta，Kalyan Saladi，Alan Schelten，Ruan Silva，Eric Michael Smith，Ranjan Subramanian，Xiaoqing Ellen Tan，Binh Tang，Ross Taylor，Adina Williams，Jian Xiang Kuan，Puxin Xu，Zheng Yan，Iliyan Zarov，Yuchen Zhang，Angela Fan，Melanie Kambadur，Sharan Narang，Aurelien Rodriguez，Robert Stojnic，Sergey Edunov，以及Thomas Scialom。2023年。Llama 2：开放基础和微调聊天模型。arXiv:2307.09288 [cs.]https://arxiv.org/abs/2307.09288[39] Ashish Vaswani，Noam Shazeer，Niki Parmar，Jakob Uszkoreit，Llion Jones，Aidan N. Gomez，Lukasz Kaiser，以及Illia Polosukhin。2023年。注意力机制是你所需要的全部。在神经信息处理系统进展。NeurIPS，美国加利福尼亚州长滩。arXiv:1706.03762 doi:10.48550/arXiv.1706.03762[40] vLLM团队。[n. d.]。CUDA图——vLLM设计文档。https://docs.vllm.ai/en/stable/design/cuda_graphs/。访问时间：2026-01-09.[41] Jiahao Wang，Jinbo Han，Xingda Wei，Sijie Shen，Dingyan Zhang，Chenguang Fang，Rong Chen，Wenyuan Yu，以及Haibo Chen。2025年。KVCache在野外：表征和优化大型云服务提供商处的KVCache缓存。465–482。https://www.usenix.org/conference/atc25/presentation/wang-jiahao

[42] 王志斌, 李世鹏, 周宇航, 李雪, 张中会, 阮文图, 郭荣, 陈晨, 陈桂海, 和钟胜. 2025. 重新审视大语言模型服务中的服务等级目标与系统级指标. arXiv:2410.14257 [cs.LG] https: //arxiv.org/abs/2410.14257[43] 魏建, 王学志, Schuurmans Dale, Bosma Maarten, Ichter Brian, 夏菲, Chi Ed, Le Quoc V, 和 周登. 2022. 思维链提示激发大语言模型的推理能力. 2022年神经信息处理系统进展 35, 24824– 24837.[44] 吴炳阳, 刘胜宇, 钟银民, 孙鹏, 刘玄哲, 和 金欣. 2024. LoongServe: 通过弹性序列并行高效服务长上下文大语言模型. ACM SIGOPS 第30届操作系统原理研讨会 (SOSP '24). ACM, 美国纽约, 纽约, 640–654. doi:10.1145/3694715.3695948[45] 姚佳怡, 李汉辰, 刘宇涵, Ray Siddhant, Cheng Yihua, 张启正, 杜昆泰, 陆山, 和 姜俊辰. 2025. CacheBlend: 快速用于RAG的缓存知识融合大语言模型服务. 第20届欧洲计算机系统会议 (EuroSys '25). ACM, 美国纽约, 纽约, 94–109. doi:10.1145/3689031.3696098[46] 叶志浩, 陈乐群, 赖瑞航, 林无伟, 张寅eng, 王斯婷, 陈天齐, Kasikci Baris, Grover Vinod, Krishnamurthy Arvind, 和 Ceze Luis. 2025. FlashInfer: 用于LLM推理服务的高效且可定制的注意力引擎. 第八届机器学习与系统会议. [47] 余景仁, 姜俊成, Kim Geon-Woo, Kim Soojeong, 和 Chun Byung-Gon. 2022. Orca: 基于Transformer的生成模型的分布式服务系统. 第16届USENIX操作系统设计与实现研讨会 (OSDI 22). 521–538.[48] 张舒莱, 陈全, 崔伟豪, 赵汉, 薛春宇, 郑振, 林伟, 和 郭明毅. 2025. 通过自适应无气泡时空共享提升GPU共享性能. 第20届欧洲计算机系统会议 (ACM会议). 573–588. doi:10.1145/3689031.3696070[49] 张伟, 崔伟豪, 傅凯华, 陈全, Mawhirter Daniel Edward, 吴波, 李超, 和 郭明毅. 2019. Laius: 迈向数据中心空间多任务加速器延迟感知与性能提升. ACM国际超级计算会议 (ICS '19). ACM, 美国纽约, 纽约,

58 68 10 1145 3330345 3330351 –

[50] [ ]

. doi: . / . . 赵夏 , Jahre Magnus, 和 Eeckhout Lieven. n. d. . HSM: 多任务GPU的混合减速模型. 第25届国际计算机体系结构支持编程语言与操作系统会议 (ASPLOS 2022). 1371–1385.[51] 郑连民, 李朱浩, 张浩 , 庄永浩 , 陈志锋 , 黄艳平 , 王逸达 , 许元忠 , 朱丹阳 , Xing Eric P., Gonzalez Joseph E., 和 Stoica Ion. 2022. Alpa: 自动化分布式深度学习的跨与 intra-算子并行. 第16届USENIX操作系统设计与实现研讨会 (OSDI 22). 559–578.[52] 郑连民, 殷良胜, 谢志强, 孙楚月, 黄杰, 余曹祎豪, 曹诗怡, Kozyrakis Christos, Stoica Ion, Gonzalez Joseph E., Barrett Clark, 和 生颖. 2024. SGLang: 结构化语言模型程序的高效执行. 第38届神经信息处理系统大会 (Vancouver, BC, Canada) (NIPS '24). Curran Associates Inc., 纽约红钩, 纽约, USA, Article 2000, 27页. [53] 钟银民, 刘胜宇, 陈君达, 胡建波, 朱伊波, 刘玄哲, 金欣, 和 张浩. 2024. DistServe: 为吞吐量优化的大语言模型分离预填充与解码

Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 193–210.

[54] Kan Zhu, Yufei Gao, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Zihao Ye, Keisuke Kamahori, Chien-Yu Lin, Ziren Wang, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. 2025. NanoFlow: Towards Optimal Large Language Model Serving Throughput. 749–765. https://www.usenix.org/conference/osdi25/presentation/zhu-kan

[55] Simiao Zuo, Xiaodong Liu, Jian Jiao, Young Jin Kim, Hany Hassan, Ruofei Zhang, Tuo Zhao, and Jianfeng Gao. 2022. Taming Sparsely Activated Transformer with Stochastic Experts. arXiv:2110.04260 (Feb. 2022). arXiv:2110.04260 [cs]

服务. 在 第*18*届*USENIX*操作系统设计与实现研讨会 *(OSDI 24)*. 193–210.

[54] 朱侃, 高宇飞, 赵一龙, 赵良宇, 左歌斐, 古一勒, 谢德东, 叶志浩, Kamahori Keisuke, Lin Chien-Yu, 王子仁, 王斯婷, Krishnamurthy Arvind, 和 Kasikci Baris. 2025. NanoFlow: 通过吞吐量优化实现大语言模型服务. 749–765. https://www.usenix.org/conference/osdi25/ presentation/zhu-kan

[55] 左思妙, 刘晓东, 焦健, 金泳镇, 哈桑·汉尼, 张若飞, 赵拓, 高建锋. 2022. 勘定稀疏激活的Transformer与随机专家. arXiv:2110.04260 (2022年2月). arXiv:2110.04260 [cs]