硕士学位论文

面向大模型推理任务的批式调度和 KV 缓存优化方法研究

BATCH SCHEDULING AND KV CACHE OPTIMIZATION FOR LARGE LANGUAGE MODEL INFERENCE

研　　究　　生：何忆源

指　导　教　师：徐敏贤副研究员

南方科技大学

二〇二六年二月

国内图书分类号：XXxxx.x　　学校代码：14325
国际图书分类号：xx-x　　密级：公开

工学硕士专业学位论文

面向大模型推理任务的批式调度和 KV 缓存优化方法研究

学 位 申 请 人：何忆源
指　导　教　师：徐敏贤副研究员
专　业　类　别：计算机技术
答　辩　日　期：2026 年 4 月
培　养　单　位：深圳理工大学
学位授予单位：南方科技大学

BATCH SCHEDULING AND KV CACHE OPTIMIZATION FOR LARGE LANGUAGE MODEL INFERENCE

A dissertation submitted to
Southern University of Science and Technology
in partial fulfillment of the requirement
for the professional degree of
Master of Engineering

by
He Yiyuan

Supervisor: Associate Researcher Minxian Xu

April, 2026

学位论文公开评阅人和答辩委员会名单

公开评阅人名单

刘 XX	教授	南方科技大学
陈 XX	副教授	XXXX 大学
杨 XX	研究员	中国 XXXX 科学院 XXXXXXX 研究所

答辩委员会名单

主席	赵 XX	教授	南方科技大学
委员	刘 XX	教授	南方科技大学
	杨 XX	研究员	中国 XXXX 科学院 XXXXXXX 研究所
	黄 XX	教授	XXXX 大学
	周 XX	副教授	XXXX 大学
秘书	吴 XX	助理研究员	南方科技大学

南方科技大学学位论文原创性声明和使用授权说明

南方科技大学学位论文原创性声明

本人郑重声明：所提交的学位论文是本人在导师指导下独立进行研究工作所取得的成果。除了特别加以标注和致谢的内容外，论文中不包含他人已发表或撰写过的研究成果。对本人的研究做出重要贡献的个人和集体，均已在文中作了明确的说明。本声明的法律结果由本人承担。

作者签名：　　日期：

南方科技大学学位论文使用授权书

本人完全了解南方科技大学有关收集、保留、使用学位论文的规定，即：

1. 按学校规定提交学位论文的电子版本。

2. 学校有权保留并向国家有关部门或机构送交学位论文的电子版，允许论文被查阅。

3. 在以教学与科研服务为目的前提下，学校可以将学位论文的全部或部分内容存储在有关数据库提供检索，并可采用数字化、云存储或其他存储手段保存本学位论文。

（1）在本论文提交当年，同意在校园网内提供查询及前十六页浏览服务。

（2）在本论文提交 □ 当年/□ __ 年以后，同意向全社会公开论文全文的在线浏览和下载。

4. 保密的学位论文在解密后适用本授权书。

作者签名：　　日期：

指导教师签名：　　日期：

摘 要

随着大语言模型（LLMs）在云计算环境中的广泛部署，其推理服务面临资源消耗巨大、服务等级目标（SLO）要求严格、预填充（Prefill）与解码（Decode）阶段资源需求异构以及负载动态多变等多重挑战。据统计，云平台中约 90% 的人工智能计算资源用于模型推理而非训练，而现有系统受限于粗粒度的静态资源配置，GPU 利用率通常仅为 20%–40%，且在突发流量下因扩容滞后导致严重的 SLO 违约。针对这些问题，本文围绕” 面向大模型的推理任务批式调度和 KV 缓存优化” 这一主题，构建了从静态优化到动态调控的技术体系，实现了对 LLM 推理服务资源管理的系统性优化。

首先，针对静态场景下批处理策略粗放、部署配置忽视网络拓扑异构性、缺乏服务质量（SLO）感知等问题，本文提出了 UELLM（Unified and Efficient LLM Inference Serving）统一高效批式调度与部署框架。该框架通过微调 ChatGLM3-6B 模型对请求输出长度进行预测，分桶准确率达到 99.51%，为后续批处理优化提供关键先验知识；设计了 SLO 与输出长度驱动的动态批处理算法（SLO-ODBS），通过权重化目标函数平衡总延迟与总输出长度，采用贪心策略将长度相近的请求组合成批，有效减少 KV Cache 冗余填充和计算浪费；提出了基于动态规划的高效低延迟资源分配算法（HELR），综合考虑集群网络拓扑异构性（NVLink/PCIe 带宽差异）和 GPU 计算能力差异，自动优化层到 GPU 的映射策略，支持高利用率（HE）与低延迟（LR）两种模式灵活切换。在 4-GPU 异构集群上的验证表明，UELLM 相比 Morphling 和 S^3 等先进方案，降低推理延迟 72.3% 至 90.3%，提升 GPU 利用率 1.2 倍至 4.1 倍，吞吐量提高 1.92 倍至 4.98 倍，并实现零 SLO 违约。

其次，面向 PD（Prefill-Decode）分离架构中固有的计算–内存负载失衡、静态资源配置无法适应动态负载变化、前缀缓存感知路由导致的热点倾斜等问题，本文提出了 BanaServe 细粒度弹性资源管理与 KV 缓存优化框架。该框架创新性地引入了模块级细粒度迁移机制，包括层级（Layer-level）权重迁移与注意力级（Attention-level）KV Cache 迁移：前者支持连续 Transformer 层的动态重配置，实现粗粒度负载均衡；后者通过按注意力头维度切分 KV Cache，选择性将部分注意力头状态迁移至辅助 GPU，实现细粒度计算卸载，且无需传输模型权重，迁移开销极低；设计了全局 KV Cache 存储与层间流水线重叠传输机制，通过解耦缓存状态与计算位置，消除前缀缓存对调度决策的约束，使路由器可基于纯负载指标进行调度，同

时利用 Transformer 逐层计算特性，将第 i 层计算与第 $i+1$ 层 KV Cache 预取重叠，隐藏通信延迟；提出了基于实时负载感知的请求调度算法，周期性地测量各 Prefill 实例的综合负载（计算 + 内存利用率），将新请求分发至负载最轻的实例，配合动态迁移机制实现快速负载均衡。在 13B 参数模型（LLaMA-13B/OPT-13B）和公开基准（Alpaca/LongBench）上的评估表明，BanaServe 相比主流的开源推理系统 vLLM 吞吐量提升 1.2 倍至 3.9 倍、总处理时间降低 3.9% 至 78.4%；相比主流的 pd 分离系统 DistServe 吞吐量提升 1.1 倍至 2.8 倍、延迟降低 1.4% 至 70.1%，并在突发流量场景下展现出优异的鲁棒性。

本文所提出的方法在真实 GPU 集群和公开基准数据集上得到了充分验证，构建的” 静态批处理优化–动态层级资源调控” 技术体系，为大模型推理服务的高效部署与资源管理提供了新的理论依据和技术路径，对推动大模型技术在智能客服、内容创作、知识检索等延迟敏感场景的规模化应用具有重要意义。

关键词：大语言模型推理；批式调度；PD 分离架构；动态批处理；异构部署优化

Abstract

With the widespread deployment of Large Language Models (LLMs) in cloud computing environments, inference services face critical challenges including massive resource consumption, strict Service Level Objective (SLO) requirements, heterogeneous resource demands between the prefill and decode phases, and highly dynamic workload variations. Statistics indicate that approximately 90% of AI computing resources in cloud platforms are dedicated to model inference rather than training. However, existing systems are constrained by coarse-grained static resource configurations, resulting in GPU utilization typically ranging merely between 20%–40%, and severe SLO violations due to delayed scaling during bursty traffic. To address these issues, this thesis focuses on "Batch Scheduling and KV Cache Optimization for Large Language Model Inference," constructing a technical architecture spanning from static optimization to dynamic scaling, thereby achieving systematic optimization of resource management for LLM inference services.

Firstly, to address coarse-grained batching strategies, deployment configurations that neglect network topology heterogeneity, and the lack of Service Level Objective (SLO) awareness in static scenarios, this thesis proposes UELLM (Unified and Efficient LLM Inference Serving), a unified batch scheduling and deployment optimization framework. The framework employs a fine-tuned ChatGLM3-6B model to predict request output lengths, achieving a bucketing accuracy of 99.51% to provide critical prior knowledge for subsequent batching optimization. It introduces the SLO and Output-Driven Dynamic Batch Scheduler (SLO-ODBS), which balances total latency and total output length through a weighted objective function, and adopts a greedy strategy to group requests with similar lengths into batches, effectively reducing KV Cache redundancy and computational waste. Additionally, the High-Efficiency Low-Latency Resource Allocation (HELR) algorithm based on dynamic programming is proposed, which automatically optimizes layer-to-GPU mapping strategies by comprehensively considering cluster network topology heterogeneity (NVLink/PCIe bandwidth variations) and GPU computational capability differences, supporting flexible switching between High Efficiency (HE) and Low Latency (LR) modes. Experimental validation on a 4-GPU heterogeneous cluster demonstrates that UELLM reduces inference latency by 72.3% to 90.3%, improves GPU utilization by 1.2× to 4.1×, and increases throughput by 1.92× to 4.98× compared to

state-of-the-art methods such as Morphling and S^3, while achieving zero SLO violations.

Secondly, targeting the inherent compute-memory load imbalance, inability of static resource configurations to adapt to dynamic workload changes, and hotspot skews caused by prefix cache-aware routing in PD (Prefill-Decode) disaggregated architectures, this thesis presents BanaServe, a fine-grained elastic resource management and KV cache optimization framework. The framework introduces novel module-level migration mechanisms, including layer-level weight migration and attention-level KV Cache migration: the former supports dynamic reconfiguration of consecutive Transformer layers for coarse-grained load rebalancing; the latter achieves fine-grained computation offloading by partitioning KV Cache along the attention head dimension and selectively migrating partial attention head states to auxiliary GPUs without transferring model weights, incurring minimal migration overhead. It designs a Global KV Cache Store with layer-wise pipelined transmission to eliminate constraints imposed by prefix caching on scheduling decisions by decoupling cache state from compute placement, enabling purely load-based routing decisions, while simultaneously hiding communication latency by overlapping the computation of layer i with the prefetching of KV Cache for layer $i + 1$ leveraging the layer-wise execution characteristic of Transformers. Furthermore, a real-time load-aware request scheduling algorithm is proposed, which periodically measures the combined load (compute and memory utilization) of each prefill instance and dispatches new requests to the least-loaded instance, working in coordination with dynamic migration for rapid load balancing. Evaluations on 13B-parameter models (LLaMA-13B/OPT-13B) and public benchmarks (Alpaca/LongBench) demonstrate that BanaServe achieves 1.2×–3.9× higher throughput and 3.9%–78.4% lower total processing time compared to the mainstream open-source inference system vLLM, and 1.1×–2.8× higher throughput with 1.4%–70.1% lower latency compared to the mainstream prefill-decode disaggregation system DistServe, while also exhibiting superior robustness under bursty traffic scenarios.

The proposed methods have been extensively validated on real GPU clusters and public benchmarks. The technical architecture of “Static Batching Optimization – Dynamic Hierarchical Resource Scaling” provides novel theoretical foundations and technical pathways for efficient deployment and resource management of LLM inference services, and holds significant importance for promoting the large-scale application of LLM technologies in latency-sensitive scenarios such as intelligent customer service, content

creation, and knowledge retrieval.

Keywords: Large Language Model Inference; Batch Scheduling; PD Disaggregated Architecture; Dynamic Batching; Heterogeneous Deployment Optimization

目 录

符号和缩略语说明

$A^{(j)}$	注意力权重矩阵，设备 j 上的注意力分数指数值
B	有效网络带宽（Gbps）
B_{net}	节点间有效互联带宽
B_{sz}	批处理大小（Batch Size）
C_d	设备 d 当前计算使用量
$C_d^{\max}$	设备 d 的峰值计算能力上限
C_ℓ	每层每 token 的计算开销
C_{gpu}	GPU 峰值计算能力
CM	当前批次综合优化指标
D	硬件设备节点集合，$D = \{d_1, d_2, \cdots, d_n\}$
E	硬件网络拓扑中节点间连接边集合
G	硬件网络拓扑图，$G = (D, E)$
H	注意力头总数
K	输出长度分桶数量；键矩阵（Key Matrix）
$K^{(j)}$	分配至设备 j 的键矩阵子集
K_{acc}	Decode 实例中解码过程累积的 KV Cache 大小
K_{init}	Prefill 实例的初始 KV Cache 分配大小
L	输入序列长度（token 数）
L_1	并行计算引入的延迟系数（与 SLO 相关）
L_2	并行计算引入的延迟系数（与输出长度相关）
L_{CM}	当前批次中的最大 SLO 值
M	LLM 显存总需求
M_0	进程基础内存开销
M_d	设备 d 当前显存使用量
$M_d^{\max}$	设备 d 的显存容量上限
M_ℓ	每层显存占用
N	当前批次中的请求数量；Transformer 总层数；并发请求总数
N_m	模块选择和开销评估的计算量
O	批次内最大输出长度；注意力输出矩阵
$O^{(j)}$	设备 j 上计算得到的注意力输出子集
O_{CM}	当前批次中的最大输出长度

P	Prefill 实例集合
Q	查询矩阵（Query Matrix）；请求队列
S	设备分配方案，$S \subseteq D$
S_{kv}	单 token 每层 KV Cache 大小（字节）
S_ℓ^{kv}	第 ℓ 层对应 KV Cache 大小
S_ℓ^{total}	层级迁移总数据量（权重 +KV Cache）
S_ℓ^{w}	第 ℓ 层权重大小
T	KV Cache 预留显存；延迟阈值
T_{attn}	注意力级 KV Cache 迁移延迟
T_{budget}	迁移延迟预算约束
T_c	缓存访问延迟
T_d	基础解码计算时间
T_F	Prefill 阶段总前向计算时间
$T_{F,\text{layer}}$	每层前向计算时间
T_{KV}	每层 KV Cache 传输时间
T_l	批次总延迟优化目标
T_{layer}	层级权重迁移延迟
T_m	内存带宽停顿时间
$T_{\text{mem\_realloc}}$	缓冲区重分配时间
T_o	批次总输出长度优化目标
T_p	Prefill 计算时间
T_q	Decode 前的排队延迟
T_{sync}	迁移同步开销
T_{TPOT}	每输出 token 时间（Time Per Output Token）
T_{TTFT}	首 token 延迟（Time To First Token）
T_x	KV Cache 传输时间（加载 + 获取）
U_{avg}	集群平均 GPU 利用率
U_d	设备 d 的归一化综合利用率
U_p	Prefill 实例平均利用率
U_{p_i}	Prefill 实例 p_i 的归一化负载
V	值矩阵（Value Matrix）
$V^{(j)}$	分配至设备 j 的值矩阵子集
W_ℓ	第 ℓ 层的模型权重矩阵
b	批次中的请求数量
d	注意力头维度

d_{head}	每个注意力头的维度
d_{model}	模型隐藏层维度
h_{kv}	GQA 下 KV 头数量
h_{total}	总注意力头数量
m	每层所需显存（$M/\text{Layer}(M)$）
n_d	Decode 实例分配的 Transformer 层数
n_p	Prefill 实例分配的 Transformer 层数
p	性能-时间关系调节系数
q_i	第 i 个推理请求
r	平均前缀缓存命中率
AI	人工智能（Artificial Intelligence）
API	应用程序接口（Application Programming Interface）
BF16	Brain Float 16 位浮点数格式
CPU	中央处理器（Central Processing Unit）
CUDA	统一计算设备架构（Compute Unified Device Architecture）
DistServe	预填充-解码分离推理系统[1]
DP	动态规划（Dynamic Programming）
FIFO	先进先出调度策略（First-In-First-Out）
FP16	16 位浮点数格式（Float Point 16）
GPU	图形处理单元（Graphics Processing Unit）
GQA	分组查询注意力机制（Grouped Query Attention）
HBM	高带宽显存（High Bandwidth Memory）
HE	高效利用率模式（High-Efficiency Mode）
HELR	高效低延迟资源分配算法（High-Efficiency Low-Latency Resource Allocation）
HFT	Hugging Face Transformers 推理框架
INT4	4 位整数量化格式
INT8	8 位整数量化格式
KV Cache	键值缓存（Key-Value Cache）
LJF	长作业优先调度（Longest Job First）
LLM	大语言模型（Large Language Model）
LongBench	长文本理解基准测试数据集[2]
LoRA	低秩自适应微调（Low-Rank Adaptation）
LR	低延迟优先模式（Low-Latency Mode）
MDP	马尔可夫决策过程（Markov Decision Process）

MLaaS	机器学习即服务（Machine Learning as a Service）
MPS	多进程服务（Multi-Process Service）
MIG	多实例 GPU（Multi-Instance GPU）
NVLink	NVIDIA 高速 GPU 互联技术
ODBS	输出长度驱动的动态批处理调度算法（Output-Driven Batch Scheduler）
OOM	显存溢出（Out-of-Memory）
PCIe	高速串行计算机扩展总线标准（Peripheral Component Interconnect Express）
PD 分离	预填充-解码分离架构（Prefill-Decode Disaggregation）
PPO	近端策略优化算法（Proximal Policy Optimization）
QoS	服务质量（Quality of Service）
QLoRA	量化低秩自适应微调（Quantized Low-Rank Adaptation）
RAG	检索增强生成（Retrieval-Augmented Generation）
RDMA	远程直接内存访问（Remote Direct Memory Access）
RPS	每秒请求数（Requests Per Second）
S3	基于输出长度感知的调度框架[3]
SAC	软演员-评论家算法（Soft Actor-Critic）
SGLang	结构化语言模型程序执行框架[4]
SJF	短作业优先调度（Shortest Job First）
SLO	服务等级目标（Service Level Objective）
SLO-DBS	SLO 感知的动态批处理调度算法（SLO-aware Dynamic Batch Scheduler）
SLO-ODBS	SLO 感知与输出长度驱动的动态批处理调度算法（SLO and Output-Driven Dynamic Batch Scheduler）
SSD	固态硬盘（Solid State Drive）
TPOT	每输出 token 时间（Time Per Output Token）
TTFT	首 token 延迟（Time To First Token）
UA	UELLM-all，同时采用 HELR 和 SLO-ODBS 的完整版本
UB	UELLM-batch，仅使用 SLO-ODBS 批处理调度算法的版本
UD	UELLM-deploy，仅使用 HELR 模型部署算法的版本
UELLM	统一高效大语言模型推理服务框架（Unified and Efficient LLM Inference Serving）
vLLM	基于 PagedAttention 的高效 LLM 推理系统[5]
WAN	广域网（Wide Area Network）

α	联合优化目标中资源利用率的权重系数
β	联合优化目标中延迟的权重系数
γ	联合优化目标中吞吐量的权重系数
δ	负载不均衡阈值
δ_L	负载感知调度中的负载阈值
$\delta_{\uparrow}$	迁移触发的上升阈值（滞后控制）
$\delta_{\downarrow}$	迁移恢复的下降阈值（滞后控制）
Δ	过载设备与欠载设备之间的瞬时负载差
η_{static}	静态批处理下 GPU 有效利用率
Θ	系统有效吞吐量（tokens/s 或 requests/s）
κ	迁移收益-开销比阈值（效率比 ρ）
$\mathcal{B}$	输出长度分桶集合
$\mathcal{D}$	Decode GPU 实例集合
$\mathcal{K}_{\ell}$	第 ℓ 层的 KV Cache
$\mathcal{P}$	Prefill GPU 实例集合
$\ell^{(j)}$	设备 j 上注意力归一化因子（softmax 分母）
π	模块迁移方案
π^*	最优模块迁移方案
ρ	迁移效率比阈值（可配置参数）
w_1	SLO-ODBS 算法中总延迟的权重系数
w_2	SLO-ODBS 算法中总输出长度的权重系数

第 1 章　绪论

1.1　研究背景

近年来, 以 GPT-4[6]、LLaMA[7]、Claude[8]等为代表的大语言模型 (Large Language Models, LLMs) 在自然语言处理、代码生成、知识检索和内容创作等领域展现出卓越的性能, 推动了人工智能技术的跨越式发展。这些模型通过数百亿乃至万亿级参数的规模化效应, 涌现出强大的上下文学习与推理能力, 已成为智能时代的关键基础设施。然而, 随着模型参数规模的指数级增长, 其训练和推理过程对计算资源提出了极高的要求, 由此产生的资源密集型特征与高效部署需求之间的矛盾日益尖锐。

在机器学习即服务 (MLaaS) 的范式下, 大模型的生命周期呈现出" 训练集中, 推理分散" 的显著特征。据统计, 云平台中约 90% 的人工智能计算资源被用于模型推理服务而非训练[9], 且推理成本随着模型规模呈超线性增长。以 OpenAI 的 ChatGPT 为例, 为维持其日均 7000 万次访问的服务规模, 需要部署超过 6 万张 NVIDIA A100 GPU, 初始投资成本高达 16 亿美元, 每日电费支出约 10 万美元[10]。这些数字清晰地揭示了大模型推理阶段对计算资源的巨额消耗及其带来的沉重经济负担。

大模型推理过程具有独特的两阶段执行特性：预填充 (Prefill) 阶段与解码 (Decode) 阶段[11-12], 这种内在的计算模式差异构成了资源优化的核心挑战。Prefill 阶段计算密集, 需并行处理整个输入序列以计算注意力键值 (KV Cache) 并生成首个令牌 (Time to First Token, TTFT), 其延迟直接影响用户感知的响应速度[13]。Decode 阶段则受限于自回归生成机制, 需逐令牌迭代输出, 呈现显著的内存密集型特征, 其关键指标为每输出令牌时间 (Time Per Output Token, TPOT)。两个阶段在计算强度、内存访问模式和延迟敏感度上的固有不对称性, 使得传统单一架构难以同时优化, 往往导致严重的资源利用率低下和负载不均衡问题[14]。

此外, 大模型推理面临着严峻的显存墙 (Memory Wall) 挑战。以千亿级参数模型为例, 其参数本身需占用数百 GB 显存, 而 KV Cache 的动态增长进一步加剧了显存压力。由于单张 GPU 显存容量有限 (通常为 40GB 或 80GB), 必须采用张量并行 (Tensor Parallelism)[15]或流水线并行 (Pipeline Parallelism) 等分布式部署策略。然而, 随着部署 GPU 数量的增加, 节点间通信开销和同步延迟显著上升。反之, 若 GPU 资源配置不足, 则会导致显存溢出 (Out-of-Memory, OOM) 错误或极长的推理延迟。这种资源配置的刚性约束与服务质量 (Quality of Service, QoS) 要求之间的张

力, 构成了大模型部署的核心难点。

更为复杂的是, 生产环境的推理负载具有高度动态性和不可预测性。请求到达率 (Requests Per Second, RPS) 随时间呈现剧烈波动, 输入/输出序列长度分布呈现显著的重尾特性 (Heavy-tailed), 即少数长序列请求占据了大量资源。传统静态资源配置策略在负载低谷期造成严重的资源浪费 (GPU 利用率常仅 20%-40%), 而在突发流量 (Bursty Traffic) 下又因扩容滞后导致服务等级目标 (Service Level Objective, SLO) 违约甚至服务中断。现有系统如 vLLM[5] 通过 PagedAttention 优化显存管理,DistServe[1] 采用 PD 分离架构消除阶段间干扰, 但这些方案仍受限于粗粒度的资源配置 (实例级或 GPU 池级) 和缺乏前瞻性的静态调度策略, 无法适应快速变化的负载模式。因此, 如何在保障严格 SLO 的前提下, 实现计算与内存资源的细粒度调度, 成为大模型推理服务从实验室走向规模化产业应用必须攻克的关键瓶颈。

1.2　研究意义

大语言模型推理服务的快速普及推动了 AI 基础设施的深度变革，但其计算密集、显存敏感、负载动态多变的固有特性对资源管理与服务质量保障提出了严峻挑战。当前，静态资源配置导致的 GPU 利用率低下、粗粒度批处理引发的 SLO 违约，以及 PD 分离架构下 Prefill 与 Decode 阶段的资源失衡等问题，不仅制约了大模型推理服务的规模化部署，也显著推高了企业的运营成本。因此，研究面向大模型推理任务的高效批式调度与 KV 缓存优化方法，是解决当前 AI 基础设施核心瓶颈、推动大模型技术规模化落地的重要基础。

本研究的意义体现在以下几个方面：首先，从理论层面，本研究通过建立考虑 KV Cache 动态增长、网络拓扑异构性和 SLO 约束的数学模型，深入探索了离散配置空间中最优部署策略的算法边界与计算复杂度，揭示了 Prefill 与 Decode 阶段分离场景下资源分配与延迟约束之间的权衡机理，为大规模分布式推理系统的资源管理提供了新的理论依据，丰富了云计算与人工智能交叉领域的研究框架。其次，从实践层面，本研究设计的批处理调度与部署优化方法能够将集群 GPU 利用率从 20%–40% 提升至 70% 以上，有效降低推理延迟并消除 SLO 违约，为云服务提供商和企业用户提供高效稳定的技术支撑，在智能客服、金融风控、内容创作等延迟敏感场景中具有直接的工程应用价值。此外，本研究提出的统一优化框架不仅适用于云端数据中心，其核心理念还可延伸至边缘计算场景，促进 GPU、TPU、CPU 等异构计算资源的协同调度技术发展，推动人工智能技术在医疗、教育、制造等行业的深度落地。

在经济效益与社会影响方面，本研究通过优化计算资源的调度与使用，预计

可降低推理成本 30% 以上，同时减少闲置 GPU 的能源浪费，降低数据中心整体功耗[9]，兼顾了经济效益与绿色计算目标。面对日益增长的大模型服务需求与技术迭代压力，本研究为构建更加高效、弹性、可持续的 AI 推理基础设施提供了创新思路与可行路径，具有重要的学术价值和产业应用前景。

1.3　国内外研究现状及分析

大语言模型推理优化的研究呈现出从算法创新到系统架构、从静态配置到动态调度、从单机优化到分布式协同的演进趋势。本节系统梳理了算法层面的解码优化、系统层面的内存管理与批处理技术、架构层面的资源配置策略, 并对比分析国内外研究特点, 最后指出现有研究的局限性与本文切入点。

1.3.1　算法层面的推理优化

（1）解码策略相关研究

为突破自回归生成的序列依赖瓶颈, 学术界提出了多种非传统的解码范式。推测性解码 (Speculative Decoding) 通过小规模草稿模型 (Draft Model) 快速生成候选序列, 再由目标模型并行验证并修正, 在保持输出质量不变的前提下实现 2-3 倍的加速比[16-17]。非自回归解码 (Non-autoregressive Decoding) 则彻底打破逐 token 生成的限制, 通过迭代精化 (Iterative Refinement)[18]或掩码预测 (Masked Prediction) 机制并行输出生成序列, 虽然牺牲部分质量, 但在实时性要求极高的场景 (如实时翻译) 展现潜力。

针对生成过程的不确定动态性, 早退出机制 (Early Exiting) 和级联推理 (Cascade Inference) 根据中间层置信度自适应决定计算深度。例如,CALM[19]通过动态层级退出减少简单查询的计算量。CascadeBERT[20]采用模型级联策略, 对简单样本使用轻量级模型, 仅将困难样本路由至大模型。这类方法通过计算量与任务难度的自适应匹配, 显著提升了平均推理效率。

（2）模型压缩与量化

为缓解显存墙 (Memory Wall) 问题, 模型压缩技术成为研究热点。低比特量化 (Low-bit Quantization) 将 FP16/FP32 权重压缩至 INT8 甚至 INT4, 配合权重量化 (Weight-only Quantization) 和激活量化 (Activation Quantization) 策略, 使千亿级模型可在单卡消费级 GPU 上部署[21-22]。近期研究还探索了 1-bit 量化 (如 BitNet[23]) 和混合精度量化, 通过细粒度分组量化 (Group-wise Quantization) 减少精度损失。

知识蒸馏 (Knowledge Distillation) 通过将大模型 (教师) 的知识迁移至小模型 (学生), 在保持性能的同时显著降低推理成本。MiniLLM[24]和 LaMini-GPT[25]等工

作表明, 经过针对性蒸馏的 7B 参数模型在特定任务上可逼近 70B 模型的性能。国内研究机构在此领域成果显著, 清华大学的 QLoRA[26] 和 Knowledge Fusion 方法在低资源场景下的压缩效率达到国际领先水平。

1.3.2　系统架构与内存管理优化

（1）注意力机制与显存优化

Transformer 的注意力机制计算复杂度高, 显存占用随序列长度平方增长。PagedAttention[5] 借鉴操作系统虚拟内存管理思想, 将 KV Cache 划分为非连续的物理块 (Block), 通过块表 (Block Table) 映射实现动态分配, 消除了传统连续存储导致的显存碎片, 使 GPU 显存利用率从 40-50% 提升至 90% 以上。在此基础上,vLLM 实现了连续批处理 (Continuous Batching), 通过动态合并新到达请求并移除已完成请求, 最大化吞吐率。

针对长上下文 (Long Context) 场景, 滑动窗口注意力 (Sliding Window Attention) 和稀疏注意力 (Sparse Attention) 通过限制注意力范围降低计算复杂度。StreamingLLM[27] 发现注意力汇点 (Attention Sinks) 现象, 仅需保留初始 token 的 KV Cache 即可维持长序列生成稳定性, 将显存消耗从序列长度的二次方降至线性。

（2）内核优化与硬件协同

为充分发挥 GPU Tensor Core 的计算能力, 算子融合 (Operator Fusion) 和自动编译 (Automatic Compilation) 技术被广泛采用。FlashAttention[28] 系列算法通过 IO 感知的精确计算调度和分块策略 (Tiling), 将注意力计算中的 HBM 访问次数从 $O(N)$ 降至 $O(N^2/\text{block\_size})$, 在 A100 GPU 上实现 2-4 倍的加速。TVM[29] 和 MLIR 等编译器框架通过自动生成针对特定硬件的微内核 (Micro-kernel), 进一步优化了矩阵乘法和内存访问模式。

1.3.3　批处理与请求调度策略

批处理 (Batching) 是提升 LLM 推理吞吐量的核心手段, 但静态批处理 (Static Batching) 的填充浪费 (Padding Waste) 问题严重。近期研究聚焦于**输出长度感知调度**和 **SLO 约束优化**。

输出长度预测是优化批处理的关键。S^3 [3] 将请求调度建模为多维装箱问题, 利用轻量级预测器估计输出长度, 将相似长度的请求归入同一批次, 减少填充开销。现有方案多基于历史数据静态预测, 难以适应输出长度分布的实时变化。

前缀缓存与数据局部性是另一优化维度。SGLang[4] 和 RadixAttention 通过前缀树 (Trie) 结构缓存共享系统提示 (System Prompt) 的 KV Cache, 对多轮对话场景实现显著加速。然而, 缓存感知路由 (Cache-aware Routing) 引入的负载不均衡问题

尚未得到有效解决, 热点前缀可能导致特定 GPU 过载。

SLO 感知调度方面,ClockWork[30]针对传统模型提出基于完成时间预测的早期退出策略, 但难以适应 LLM 自回归生成的动态延迟特性。Splitwise[31]尝试在保证延迟 SLO 的前提下优化吞吐量, 但缺乏对 Prefill 与 Decode 阶段差异性的细粒度建模。

1.3.4　计算和存储资源配置策略

（1）PD 分离架构的演进

大模型推理的 Prefill 阶段 (计算密集) 与 Decode 阶段 (内存密集) 具有截然不同的资源需求特征, 催生了预填充-解码分离 (Prefill-Decode Disaggregation) 架构 (PD 分离)。DistServe[1]和 Splitwise[31]将两个阶段部署在不同 GPU 实例上, 消除阶段间干扰, 使各自可独立优化。然而, 现有方案多采用静态配比 (如 1:1 或固定比例), 无法适应动态负载中 Prefill 与 Decode 需求的实时变化[32-35]。

分布式 KV Cache 管理是支撑 PD 分离的关键。Mooncake[36]提出全局 KV Cache 池, 通过 RDMA 网络实现跨节点缓存迁移。MemServe[37]探索了缓存预取 (Prefetching) 和冗余消除技术。但这些方案引入了显著的跨节点通信开销, 且缺乏对网络拓扑异构性的感知。

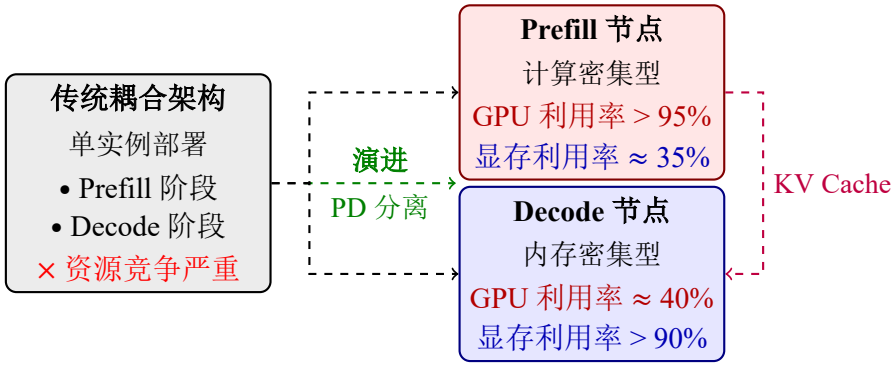


图 1-1　传统耦合架构与 PD 分离架构对比

（2）部署配置优化

传统模型部署采用静态配置方式, 在部署时固定 GPU 数量和层到设备的映射关系, 无法在运行期间根据实际负载动态调整。Morphling[38]提出基于元学习的配置搜索框架, 但需要对每个候选配置进行压力测试, 产生额外计算负担。

在异构资源管理方面,AntMan[39]和 Allox[40]提出针对深度学习任务的 GPU 时间片调度, 但主要针对训练场景, 未考虑 LLM 推理中 KV Cache 的状态保持需求。NVIDIA MPS 和 MIG(Multi-Instance GPU) 支持硬件级的 GPU 切分, 但配置刚性且缺乏软件层面的灵活性。

1.3.5　国内外研究现状对比

（1）**国外研究**

以 OpenAI、Google、Meta、Stanford 等为代表的国外研究机构在系统架构创新和硬件-软件协同设计方面处于领先地位。OpenAI 的 GPT 系列和 Google 的 Gemini 推动了超大规模模型的工程化实践, 其内部推理系统 (如 Triton、Pathways) 虽未完全开源, 但通过 vLLM、TensorRT-LLM 等开源工具影响了行业标准。学术机构如 Stanford 的 FlexGen[41] 和 Berkeley 的 SkyPilot[42] 在资源受限场景下的推理优化提出系统性解决方案。

在新兴硬件适配方面, 国外研究积极探索 TPU、AWS Inferentia、Groq LPU 等专用芯片的优化策略, 以及 FPGA/ASIC 的定制化推理加速。PagedAttention、FlashAttention 等底层优化多源自国外顶尖实验室。

（2）**国内研究**

国内以百度、阿里、华为、清华、中科院等为代表的研究力量在垂直场景优化和算法效率提升方面表现突出。百度的 ERNIE 系列、阿里的通义千问 (Qwen)、华为的盘古 (Pangu) 模型在中文语境和多模态推理方面具有特色; 清华的 FastTransformer 实验室在稀疏化、量化方面贡献显著。

国内研究更注重产业落地与边缘部署。针对国内云计算和边缘计算的特定需求, 研究重点集中在移动端部署 (如小米、OPPO 的端侧大模型)、多模态推理优化 (文生图、文生视频的高效推理) 以及国产 AI 芯片 (昇腾、寒武纪、海光) 的适配优化。此外, 国内在长文本处理 (如月之暗面的 Kimi 模型) 和智能体 (Agent) 推理调度方面形成了特色研究方向。

表 1-1　国内外大模型推理优化研究特点对比

维度	**国外研究**	**国内研究**
核心优势	系统架构、硬件协同	场景驱动、算法创新
代表工作	vLLM[5], DistServe[1]	QLoRA[26], Mooncake[36]
技术重点	底层系统、内存管理	模型压缩、端侧部署
产业落地	云基础设施	端侧 AI、国产芯片适配

1.3.6　现有研究的局限性与启示

综合上述分析, 当前 LLM 推理优化研究在以下维度存在显著局限：

（1）**调度粒度与资源状态的紧耦合**：现有系统 (如 SGLang、Mooncake) 的调度决策严重依赖 KV Cache 的物理位置, 前缀缓存感知路由导致负载热点倾斜。

路由器被迫在计算负载均衡与缓存命中率之间做困难权衡 (Cache-Load Balancing Trade-off), 缺乏计算-缓存联合优化的全局视角。

（2）**资源配置的静态化与刚性约束**：现有 PD 分离系统 (DistServe、Splitwise) 在部署时固定 Prefill 与 Decode 实例比例, 无法在运行期间根据实际负载动态调整。传统部署方法的配置搜索开销巨大, 且模型加载时间较长, 难以处理突发流量下的资源重分配需求。

（3）**缺乏跨阶段的细粒度资源协同机制**：Prefill 与 Decode 阶段的资源需求 (计算 vs 内存) 在时域和空域上互补, 但现有系统缺乏在两个阶段之间实时优化资源配置的细粒度机制, 导致严重的资源利用率失衡 (Prefill 实例计算饱和但显存闲置,Decode 实例反之)。

（4）**对异构性和动态性的适应性不足**：现有方案多假设同构的 GPU 集群和稳态负载, 对混合型号 GPU、网络拓扑差异以及重尾分布 (Heavy-tailed) 的动态负载缺乏有效建模, 导致实际部署中资源碎片化严重。

针对上述局限, 本文拟从请求级批处理优化与层级资源分配两个层面, 构建大模型推理服务的统一资源管理框架, 突破静态配置与动态需求间的结构性矛盾, 实现计算-内存资源的细粒度协同优化。

1.4　论文主要工作

针对大语言模型 (LLM) 推理服务中**资源需求难以预测、静态配置效率低下、批处理策略粗放**等关键挑战, 本文围绕**大模型推理服务的批式调度与 KV 缓存优化**这一核心主题, 构建批处理优化-层级资源分配的完整技术体系。如图 1-2所示, 研究工作涵盖静态场景下的批处理与部署优化, 以及动态场景下的 PD 分离架构资源管理, 形成从任务接入到资源释放的全生命周期管理闭环。

1.4.1　面向 LLM 推理任务的高效批式调度和部署方法

针对传统系统批处理策略粗放 (FIFO)、部署配置静态化、缺乏服务质量 (SLO) 感知等缺陷, 本文提出 UELLM(Unified and Efficient LLM Inference Serving) 框架, 实现从资源画像到调度决策的端到端优化。

（1）**基于微调大模型的输出长度预测**

突破传统静态分析方法的局限, 认识到输出长度是决定 KV Cache 大小和计算量的关键变量。采用 ChatGLM3-6B 作为基础模型, 在 Alpaca[43] 和 Natural Questions 数据集上进行 LoRA 微调, 将输出长度预测建模为分类任务 (分桶策略：8/16/32/64/128/.../2048 tokens)。微调后模型在测试集上达到 **99.51%** 的分桶准确率,

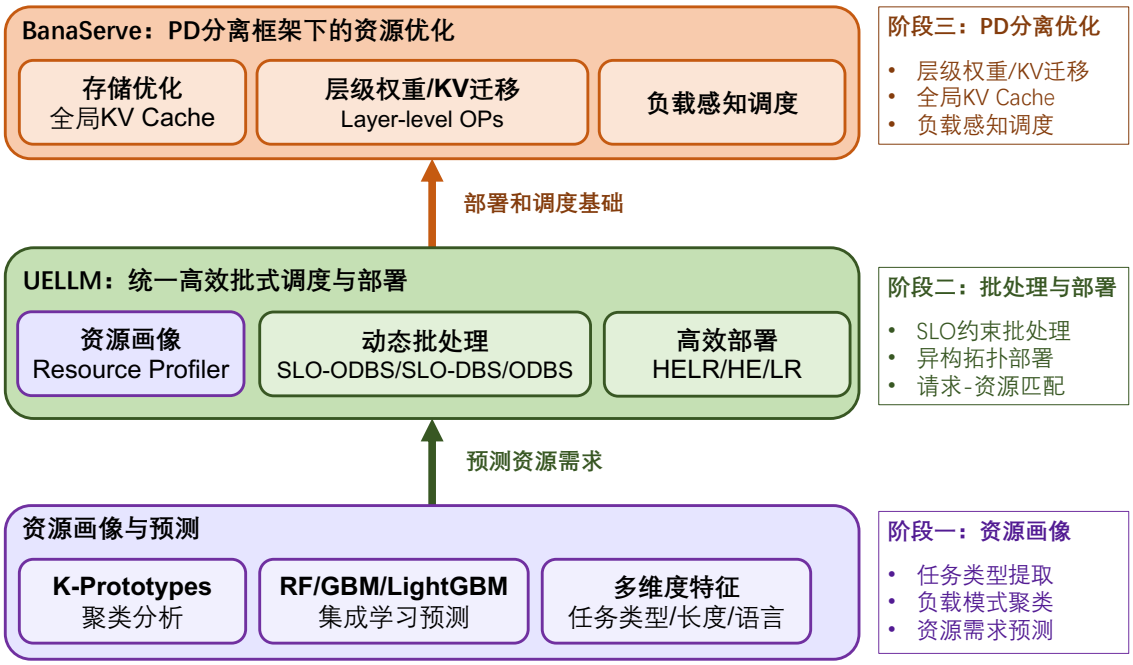


图 1-2　论文整体研究框架

相比基于规则的方法显著提升了预测精度。

（2）**SLO 与输出长度驱动的动态批处理算法 (SLO-ODBS)**

针对传统 FIFO 批处理策略导致的 KV Cache 冗余填充和 SLO 违约问题, 建立了综合考虑请求 SLO 约束和输出长度差异的批处理优化模型。算法通过权重化目标函数平衡总延迟与总输出长度, 采用贪心策略将长度相近的请求组合成批, 避免为短输出请求不必要的填充计算。实验表明,SLO-ODBS 在保持低延迟的同时, 将 SLO 违约率降低至接近 0%。

（3）**面向异构拓扑的高效资源分配算法 (HELR)**

针对 LLM 分布式部署中设备映射 (Device Map) 搜索空间巨大、静态配置性能次优的问题, 提出了基于动态规划的高效资源分配算法。算法综合考虑集群网络拓扑 (NVLink、PCIe 带宽异构)、GPU 计算能力差异和模型层间依赖关系, 自动求解最优的层到设备映射策略。通过调整权重系数 α_1 和 α_2, 可在高利用率模式 (HE) 和低延迟模式 (LR) 间灵活切换, 适应不同服务等级需求。

UELLM 框架在真实 4-GPU 集群上的验证表明, 相比 Morphling 和 S^3 等先进方案, 系统降低推理延迟 72.3% 至 90.3%, 提升 GPU 利用率 1.2 倍至 4.1 倍, 吞吐量提高 1.92 倍至 4.98 倍, 实现了零 SLO 违约的高质量推理服务。

1.4.2　面向 PD 分离架构的资源管理和 KV 缓存优化方法

当面临**动态变化**的负载时, 静态配置的系统难以维持高效运行。特别是 PD(Prefill-Decode) 分离架构中,Prefill 阶段计算密集而 Decode 阶段内存密集的固

有特性导致资源利用率失衡[14,44-45]。针对该问题, 本文提出 BanaServe 优化框架, 通过**层级权重分配**与**全局 KV Cache 管理**实现 PD 分离架构的性能优化。

（1）**层级权重迁移机制**。

针对 PD 分离架构中 Prefill 阶段计算密集与 Decode 阶段内存密集的资源需求差异, 提出了层级 (Layer-level) 权重迁移机制。该机制支持将连续的 Transformer 层动态分配至不同的 GPU 实例, 实现粗粒度负载重平衡。通过合理的层级划分, 主 GPU 与辅助 GPU 可以并行处理不同的模型层, 在不增加端到端延迟的情况下提升整体吞吐量。数学上证明了层级分解的正确性, 确保分布式计算与单卡计算数值等价。

（2）**注意力头级别的细粒度迁移**。

为应对更细粒度的负载波动, 进一步提出了注意力级别 (Attention-level) 的 KV Cache 迁移策略。该策略沿注意力头维度分割 KV Cache, 选择性将部分注意力头的状态迁移至辅助 GPU, 而无需传输模型权重。由于仅迁移中间激活值且传输量小, 该方法可在毫秒级完成负载调整, 适用于实时性要求高的动态场景。

（3）**全局 KV Cache 存储与层间流水线传输**。

针对传统前缀缓存感知路由导致的负载倾斜问题, 设计了跨 Prefill 实例的全局 KV Cache 存储层, 解耦缓存状态与计算位置, 使调度决策无需受限于缓存局部性。针对全局存储引入的访问延迟, 提出了层间流水线重叠传输机制, 利用 Transformer 逐层计算特性, 将第 i 层计算与第 $i+1$ 层 KV Cache 预取重叠。理论分析与实验验证表明, 当 KV Cache 传输时间 (约 0.082ms) 远小于层计算时间 (约 4.22ms) 时, 可实现近透明的缓存访问。

（4）**负载感知调度算法**。

基于全局 KV Cache 存储, 实现了完全基于实时负载的调度策略。算法周期性地测量各 Prefill 实例的综合负载 (计算 + 内存利用率), 将新请求分发至负载最轻的实例。通过消除缓存局部性对调度的约束, 系统能够实现更均衡的负载分布, 避免热点实例过载而其他实例空闲的情况。

BanaServe 在 13B 参数模型 (LLaMA-13B[7]、OPT-13B[46]) 和公开基准 (Alpaca[43]、LongBench[2]) 上的评估表明, 相比 vLLM, 系统吞吐量提升 1.2 倍至 3.9 倍, 总处理时间降低 3.9% 至 78.4%。相比 DistServe, 吞吐量提升 1.1 倍至 2.8 倍, 延迟降低 1.4% 至 70.1%, 并在突发流量下展现出卓越的鲁棒性。

1.4.3　主要创新点

本文围绕大模型推理服务的资源管理问题, 从批处理优化到层级部署开展了系统性研究, 主要创新点可总结为：

（1）**提出了 SLO 感知的动态批处理与异构部署联合优化方法**：将输出长度预测与 SLO 约束联合建模, 设计了 SLO-ODBS 批处理算法与 HELR 部署优化算法, 突破了传统批处理仅优化吞吐量、传统部署忽视网络拓扑异构性的局限, 实现了延迟、利用率与违约率的联合优化。

（2）**提出了 PD 分离架构下的层级资源分配新方法**：针对 PD 分离架构, 提出了层级权重迁移机制, 实现了计算与内存资源的合理配置和 PD 分离架构下细粒度资源分配。

（3）**提出了全局 KV Cache 共享与计算-通信重叠机制**：通过全局 KV Cache 存储解耦缓存状态与计算位置, 消除了前缀缓存对调度的约束。结合层间流水线传输技术, 解决了全局存储的延迟瓶颈, 实现了 PD 分离架构下的高效负载均衡。

1.5　论文组织架构

本文共分为五章, 各章内容安排如下：

第一章 绪论：介绍大语言模型推理服务的研究背景与意义, 系统梳理国内外在算法优化、系统架构、批处理调度等方面的研究现状, 深入分析现有研究的局限性。阐述本文” 批处理优化-层级部署” 的技术路线与创新点。

第二章 现有 LLM 推理任务的批式调度与 KV 缓存优化方法：系统梳理大模型推理优化领域的现有技术方案。在批式任务调度方面, 介绍 S^3 的输出长度感知调度、Continuous Batching 的动态请求合并、Orca[18] 中的 In-flight Batching 批处理机制, 以及 FIFO、短作业优先、长作业优先等传统调度策略。在 KV 缓存优化方面, 分析 Mooncake 的全局缓存池架构、MemServe 的弹性内存管理、vLLM 的 PagedAttention 显存优化, 以及模型量化等压缩技术。最后总结现有方法在**调度粒度与资源状态紧耦合、资源配置静态化、缺乏跨阶段细粒度协同机制、对异构性和动态性适应性不足**等方面的局限性, 为后续章节提出 UELLM 和 BanaServe 两种优化方案提供技术背景与改进动机。

第三章 面向 LLM 推理任务的高效批式调度和部署方法：UELLM：研究静态场景下的批处理与部署联合优化。介绍基于微调大模型 (ChatGLM3-6B) 的输出长度预测方法 (99.51% 分桶准确率)。详细阐述 SLO 与输出长度驱动的动态批处理算法 SLO-ODBS 的数学模型 (填充浪费建模) 与双阶段贪心策略。论述面向异构拓扑的高效资源分配算法 HELR, 包括基于动态规划的层到 GPU 映射优化、HE/LR 双模式切换机制。介绍 UELLM 系统实现与 4-GPU 集群实验验证, 展示相比 Morphling 和 S^3 在延迟、利用率、吞吐量方面的性能提升。

第四章 面向 PD 分离架构的资源管理和 KV 缓存优化方法：BanaServe：针对

PD 分离架构下的资源配置问题, 研究层级资源分配与 KV 缓存优化机制。阐述层级 (Layer-level) 权重迁移的资源重平衡策略, 给出层级分解的数学正确性证明。介绍全局 KV Cache 存储架构设计与层间流水线重叠传输机制, 分析通信-计算重叠的条件与收益。论述负载感知请求调度算法, 通过 LLaMA-13B 和 OPT-13B 模型在 Alpaca 和 LongBench 基准上的实验, 验证 BanaServe 相比 vLLM 和 DistServe 的性能优势。

第五章 总结与展望：总结本文的主要研究成果, 讨论存在的问题与局限性, 展望未来研究方向, 包括异构硬件感知调度、端到端延迟优化、跨地域分布式推理等。

第 2 章　现有 LLM 推理任务的批式调度与 KV 缓存优化方法

2.1　概述

大语言模型推理优化是一个多维度的系统工程问题，涵盖从请求接入、批处理调度、内存管理到分布式部署的完整技术栈。本章系统梳理现有 LLM 推理优化方法，重点围绕**批式任务调度**与**KV 缓存优化**两条主线展开分析。

在批式任务调度方面，本章介绍传统调度策略（FIFO、短作业优先、长作业优先）、输出长度感知的 S^3 调度框架[3]、连续批处理（Continuous Batching）机制以及 Orca[18]中的 In-flight Batching 技术，分析各方法在吞吐量提升与延迟控制之间的权衡。在 KV 缓存优化方面，本章分析 vLLM[5]的 PagedAttention 显存管理机制、Mooncake[36]的全局 KV Cache 池架构、MemServe[37]的弹性内存管理技术，以及模型量化等压缩手段对显存占用的影响。

最后，本章从**调度粒度与资源状态紧耦合**、**资源配置静态化**、**缺乏跨阶段细粒度协同机制**、**对异构性和动态性适应性不足**四个维度总结现有方法的局限性，为后续章节提出 UELLM 和 BanaServe 优化方案提供技术背景与改进动机。

2.2　批式任务调度方法

批式任务调度（Batch Scheduling）是提升 LLM 推理服务吞吐量、降低单请求服务成本的核心手段。其基本思想是将多个推理请求合并为一个批次（Batch）进行并行处理，通过权重共享减少冗余的模型参数加载，从而充分利用 GPU 的矩阵并行计算能力。然而，LLM 推理的自回归生成特性使得批处理面临独特挑战：不同请求的输入输出长度差异悬殊，且输出长度在请求到达时完全未知。本节从传统调度策略到新型感知调度逐层递进，系统梳理现有方法。

2.2.1　传统调度策略

传统推理系统主要采用三类基础调度策略，这些策略设计简单，但在 LLM 推理场景下均存在显著局限。

（1）**先进先出调度（FIFO）**

先进先出（First-In-First-Out, FIFO）是最基础的调度策略，按请求到达的时间顺序依次处理。在静态批处理（Static Batching）框架下，系统等待固定数量的请求到达后组成一个批次，再统一提交推理引擎执行。

FIFO 策略的主要问题在于**批内长度异构性**导致的计算浪费。当批次中同时存在短输出请求（如回答” 中国的首都是北京”，仅需 8 个 token）和长输出请求（如” 比较猫和狗的异同”，需 58 个 token）时，短请求完成后必须等待长请求全部生成完毕才能将整个批次返回。在此过程中，短请求的计算槽位被强制填充（Padding）大量无效 token，产生严重的 KV Cache 冗余和计算资源浪费[10,47]。

定量分析表明，对于批大小为 b、最大输入序列长度为 s、最大输出序列长度为 n、隐藏层维度为 h、Transformer 层数为 l 的推理任务，KV Cache 峰值存储字节数为：

$$\text{KV Cache}_{\text{peak}} = 4 \times b \cdot l \cdot h \cdot (s + n) \tag{2-1}$$

可见 KV Cache 随批大小和最大输出长度线性增长[41]。当批次内各请求输出长度差异显著时，短请求的完成不能及时释放其 KV Cache 槽位，导致显存被无效占用，进一步限制系统可接纳的有效批大小。

（2）短作业优先调度（SJF）

短作业优先（Shortest Job First, SJF）将预估执行时间最短的请求优先调度处理，目标是最小化系统平均等待时间和响应时间。在 LLM 推理场景中，” 作业长度” 通常以预期输出 token 数量来衡量[48]。

SJF 的理论优势在于其最优平均等待时间的数学性质：在非抢占式 SJF 下，对于 n 个已知执行时间的独立任务，SJF 可使平均等待时间最小化。然而，该策略在 LLM 实践中面临三个主要挑战：其一，LLM 推理的输出长度在请求开始前无法精确获取，SJF 的实施依赖于长度预测器的准确性，预测偏差会直接导致调度次序错乱；其二，在高负载场景下，持续到来的短请求可能使长请求长期得不到调度，导致服务公平性问题和 SLO 违约；其三，纯 SJF 关注单请求调度顺序，未考虑批次内部的长度匹配问题，难以与批处理机制有效结合。

（3）长作业优先调度（LJF）

长作业优先（Longest Job First, LJF）策略优先调度预期输出最长的请求，其设计初衷是避免长请求因频繁被短请求抢占而造成的队尾延迟（Tail Latency）问题。然而，LJF 在 LLM 场景中同样存在明显缺陷：由于长请求占用 KV Cache 的时间更久，LJF 可能导致系统显存长期处于高水位，限制并发处理能力；同时，大量短请求的响应时间因等待长请求完成而显著增加，严重影响用户体验。

（4）传统调度策略的共同局限

综合来看，上述传统策略均缺乏对 LLM 推理两阶段异构特性的感知：不区分 Prefill 与 Decode 阶段的资源需求差异；调度决策不考虑 SLO 约束；批次构建未能利用输出长度信息消除填充浪费；静态配置无法适应动态变化的请求负载。这些

局限共同导致了实际部署中 GPU 利用率低下（通常仅 20%–40%）和 SLO 违约率偏高的问题[10,49-51]。

2.2.2　输出长度感知的 S^3 调度框架

针对传统策略中输出长度未知导致的批处理低效问题，Jin 等人[3]提出了 S^3（Skipping Scheduling for GPU Sharing）框架，将输出长度预测与批处理调度联合优化，是该领域的重要里程碑工作。

（1）**核心思想：批组合的装箱问题建模**

S^3 将批次构建建模为一个**多维装箱问题**（Multi-dimensional Bin Packing Problem）。给定一组待调度请求 $\mathcal{Q} = \{q_1, q_2, \cdots, q_N\}$，每个请求 q_i 具有输入长度 Input_i 和预测输出长度 $\widehat{\text{Output}}_i$，批次构建的目标是将请求分配至若干批次 $\mathcal{B} = \{B_1, B_2, \cdots, B_K\}$，使得所有批次中因填充产生的冗余 token 总数最小化：

$$\min_{\mathcal{B}} \sum_{k=1}^{K} \left[\max_{q_i \in B_k} \widehat{\text{Output}}_i \cdot |B_k| - \sum_{q_i \in B_k} \widehat{\text{Output}}_i \right] \tag{2-2}$$

S^3 通过将预测输出长度相近的请求聚合为同一批次，有效减少批内的最大输出长度差异，从而降低填充比例，提升有效计算密度。

（2）**轻量级输出长度预测器**

S^3 的预测模块采用轻量级分类模型，将输出长度分桶（Bucketing）为若干离散区间（如 $[0, 8)$、$[8, 16)$、$[16, 32)$、$[32, 64)$、……、$[1024, 2048)$ 等），以请求的输入文本特征为输入，训练一个多分类器预测所属桶编号。分桶策略将连续的长度预测问题转化为分类问题，显著降低了预测复杂度，同时保证了调度决策的快速响应。预测器通常采用 BERT 等小型预训练语言模型进行微调，在保持较高预测精度的同时，推理延迟远低于目标 LLM 本身。

（3）**S^3 的贡献与局限**

S^3 将输出长度预测与 LLM 批处理联合建模，验证了长度感知调度对 GPU 利用率提升的有效性。然而，S^3 存在以下局限：未考虑 SLO 约束，仅优化填充浪费，在混合 SLO 场景下会产生大量违约；预测器基于历史数据离线训练，对输出长度分布的实时变化适应能力有限；未考虑 GPU 拓扑和设备映射（Device Map）的优化；对于预测错误的纠正依赖于周期性的离线重训练，实时性不足。上述局限性正是本文第三章提出 UELLM 框架的直接动机，UELLM 在 S^3 基础上进一步将 SLO 感知调度与异构拓扑感知部署纳入统一优化框架[10]。

2.2.3　连续批处理机制（Continuous Batching）

传统静态批处理（Static Batching）的一个根本缺陷在于：批次中最长的请求完成之前，整个批次不能释放，GPU 必须等待尾部请求生成完毕才能接受新的请求。这导致在动态请求流场景下，GPU 大量时间处于等待状态，有效利用率严重受损。

针对这一问题，**连续批处理**（Continuous Batching）打破了静态批次的边界，实现了请求级别的动态合并与退出，是现代 LLM 推理系统（如 vLLM[5]）的核心调度机制之一。其核心思想是：在每个解码步骤（Decode Step）完成后，检查批次中已生成 EOS（End-of-Sequence）token 的请求，将其立即从批次中移除，同时从等待队列中选取新的请求补充进入批次，使 GPU 在任意时刻均保持接近满载的状态。

设系统批次槽位数为 B，第 i 个请求的生成长度为 L_i。在静态批处理下，GPU 有效利用率可近似表示为：

$$\eta_{\text{static}} = \frac{\sum_{i=1}^{B} L_i}{B \cdot \max_i L_i} \tag{2-3}$$

当请求长度分布呈现重尾特性时，η_{static} 可低至 20% 以下。而连续批处理通过实时替换已完成请求，理论上可将 GPU 利用率提升至接近 100%，显著改善系统吞吐量。

连续批处理的实现需要解决两个关键问题：其一，新请求插入时需即时分配 KV Cache 空间，而传统连续显存分配方案会产生严重碎片，难以支持动态插入，vLLM 通过 PagedAttention 解决了这一问题（详见第 2.3.1节）；其二，动态插入的新请求处于 Prefill 阶段，而已有请求处于 Decode 阶段，两者混合执行会引入阶段间干扰，需要精细的调度策略加以协调。

2.2.4　Orca 中的 In-flight Batching 机制

Orca[18]是最早系统性地提出并实现 In-flight Batching（也称迭代级调度，Iteration-level Scheduling）的 LLM 推理框架，其核心思想与连续批处理高度一致，但在调度粒度和实现机制上进行了更为系统的设计。

（1）**迭代级调度**

Orca[18]将调度粒度从”请求级”细化为”迭代级”，即在每次前向传播（Forward Pass）后重新做出调度决策。具体而言，Orca 维护一个全局请求队列，每次迭代开始时检测当前批次中已完成的请求将其移出，并从等待队列中选取新请求加入批次直到达到显存上限，随后对批次中处于 Prefill 阶段的新请求和处于 Decode 阶段

的旧请求采用统一的前向传播流程处理。

（2）**选择性批处理（Selective Batching）**

Orca 观察到 Transformer[11]不同层的计算模式存在差异：注意力层需要访问各请求的 KV Cache，难以对不同长度的请求做统一批处理；而前馈层（FFN Layer）和归一化层则对批内长度不敏感。基于这一观察，Orca 提出**选择性批处理**策略：对 FFN 层采用标准批处理，对注意力层则按请求独立计算，在保证计算正确性的同时最大化批处理收益。Orca 通过迭代级调度相比传统静态批处理实现了最高 36.9 倍的吞吐量提升[18]。然而，Orca 采用连续显存预分配策略，未能有效解决 KV Cache 碎片化问题；批次内同时存在 Prefill 和 Decode 请求时，计算密集的 Prefill 会显著增加 Decode 请求的 TPOT 延迟；此外，Orca 的调度决策以最大化吞吐量为主要目标，未将不同请求的 SLO 差异纳入调度优先级考量。

2.3　KV 缓存优化方法

KV Cache 是 LLM 推理中显存占用的主要来源，其管理效率直接决定了系统的并发处理能力和服务吞吐量。本节系统梳理现有 KV Cache 优化技术，包括 PagedAttention 显存管理、全局 KV Cache 池架构、弹性内存管理以及模型量化压缩等方法。

2.3.1　PagedAttention 与 vLLM 的显存管理

vLLM[5]提出的 PagedAttention 是迄今为止最具影响力的 KV Cache 管理技术之一，其设计思想直接借鉴了操作系统的虚拟内存分页管理机制。

（1）**传统 KV Cache 管理的问题**

在传统 LLM 推理系统中，KV Cache 以连续显存块的形式为每个请求预分配。由于输出长度在请求开始时未知，系统通常按最大可能长度预留显存，导致严重的内部碎片（Internal Fragmentation）。此外，请求间共享前缀的 KV Cache 无法复用，产生大量冗余存储。实测表明，传统方案的 KV Cache 显存利用率通常仅为 40%–60%[5,52]。

（2）**PagedAttention 的核心机制**

PagedAttention 将每个请求的 KV Cache 划分为固定大小的物理块（Physical Block），每个块存储固定数量 token 的键值对。系统维护一张块表（Block Table），记录每个请求的逻辑块编号到物理块地址的映射关系。

这种非连续存储机制带来了三项关键优势：消除内部碎片，KV Cache 按需分配物理块，不再预留最大长度空间，内部碎片从平均 30% 以上降至接近零；支持

动态扩展，请求生成过程中随着输出长度增加按需分配新物理块，无需预先知道最终输出长度；实现前缀共享，多个请求共享相同系统提示时，其对应的 KV Cache 物理块可被多个逻辑块表同时引用，通过写时复制（Copy-on-Write）机制安全共享，显著减少冗余存储。

（3）**PagedAttention 的性能表现与局限**

vLLM 在 PagedAttention 基础上实现了完整的连续批处理推理引擎。实验表明，相比 FasterTransformer 和 Orca 等先进系统，vLLM 在相同硬件条件下吞吐量提升最高达 24 倍，KV Cache 显存利用率从传统方案的 40%–60% 提升至 90% 以上[5]。尽管如此，PagedAttention 仍存在若干局限：其一，基于单实例的显存管理设计在 PD 分离架构下不支持高效的跨节点 KV Cache 共享；其二，块粒度的分配在极端长序列场景下仍存在少量外部碎片；其三，在超高并发场景下块表维护引入的地址转换开销可能成为性能瓶颈。

2.3.2　Mooncake 的全局 KV Cache 池架构

Mooncake[36]由月之暗面（Moonshot AI）提出，是面向超长上下文（Long Context）推理场景的 KV Cache 中心化管理方案，代表了 KV Cache 管理从实例级向集群级演进的重要方向。

（1）**KVCache 中心化架构**

Mooncake[36]的核心设计理念是将 KV Cache 从各推理实例的本地显存中剥离，构建一个全局分布式 KV Cache 池（Global KV Cache Pool），统一管理整个集群的 KV Cache 资源。该池由 CPU 内存和 SSD 共同组成存储层次，通过 RDMA（Remote Direct Memory Access）网络实现跨节点的高速 KV Cache 访问。所有 Prefill 和 Decode 实例均可访问统一的 KV Cache 存储层，实现跨实例的缓存复用与 SLO 感知调度。

（2）**SLO 感知的分层调度**

Mooncake 将请求按 SLO 紧迫程度分为不同优先级，结合 KV Cache 的访问热度（热数据保留在 CPU 内存，冷数据迁移至 SSD），实现差异化的缓存驱逐策略。对于 SLO 紧迫的请求，优先从 CPU 内存加载 KV Cache 以最小化 TTFT；对于 SLO 宽松的请求，允许从 SSD 加载，在降低存储成本的同时保证服务质量。

（3）**前缀缓存的全局复用**

Mooncake 通过全局前缀树（Global Prefix Trie）索引所有已缓存的 KV Cache 前缀，任意推理实例均可查询并复用其他实例历史生成的 KV Cache。这一机制在多轮对话、相似系统提示等场景下显著减少了重复的 Prefill 计算，整体前缀缓存命中率可达 60% 以上。

（4）**Mooncake 的局限**

然而，Mooncake 的分布式设计也引入了新的挑战。首先，跨节点 KV Cache 访问的 RDMA 传输延迟不可忽视，在网络拥塞时会显著增加 TTFT。其次，全局元数据管理器存在单点瓶颈风险，在超大规模集群中的可扩展性有限。最关键的是，Mooncake 的调度仍需感知 KV Cache 的物理位置，缓存局部性约束未被彻底解耦，前缀缓存命中率高的节点依然会吸引更多请求，引发负载倾斜（Load Skew）问题。这一缺陷正是 BanaServe[53] 在设计全局 KV Cache Store 时着重解决的核心问题，详见第四章。

2.3.3　MemServe 的弹性内存管理

MemServe[37] 提出了面向 PD 分离架构的弹性内存池（Elastic Memory Pool）方案，旨在解决 Prefill 实例与 Decode 实例之间内存资源的动态再分配问题。

（1）**弹性内存池设计**

MemServe 的核心创新在于将传统静态绑定于单个实例的内存资源抽象为一个统一的弹性内存池，支持在 Prefill 和 Decode 实例之间动态迁移内存配额。当 Decode 阶段的 KV Cache 累积量快速增长导致显存紧张时，系统可从空闲的 Prefill 实例借用内存页；反之，当 Prefill 阶段计算压力骤增时，可回收 Decode 实例的冗余内存。

（2）**上下文缓存与预取机制**

MemServe 还引入了上下文缓存（Context Caching）机制，对多轮对话中历史轮次的 KV Cache 进行持久化存储，避免每次对话轮次都重新计算相同的历史上下文。结合异步预取（Asynchronous Prefetching）策略，系统可在当前请求处理期间提前将下一轮可能需要的 KV Cache 从 CPU 内存传输至 GPU 显存，有效隐藏数据传输延迟。

（3）**MemServe 的局限**

MemServe 的弹性内存池在一定程度上缓解了 PD 分离架构的内存不均衡问题，但其调度粒度仍停留在实例级（Instance-level），无法实现层级（Layer-level）或注意力头级别（Attention-level）的细粒度资源再分配。此外，MemServe 对网络拓扑异构性（NVLink 与 PCIe 带宽差异）缺乏感知，跨节点内存迁移的开销估算不够精确，在实际异构集群中的表现与理论值存在一定差距。

2.3.4　模型量化与压缩技术

除系统层面的 KV Cache 管理优化外，模型压缩技术通过直接减小模型参数和激活值的存储精度，从根本上降低显存占用，是缓解显存墙问题的重要补充手段。

（1）**低比特权重量化**

权重量化（Weight Quantization）将模型参数从 FP16/FP32 压缩至 INT8、INT4 乃至更低比特表示，在显著降低显存占用的同时，可利用整数计算单元加速矩阵乘法运算。LLM.int8()[21]在十亿级参数模型上实现无损 INT8 量化，通过**混合精度分解**策略将包含离群值（Outlier）的维度保留为 FP16 计算，其余维度采用 INT8 量化，实现了接近 2 倍的显存压缩。GPTQ[22]采用基于二阶近似的逐层量化策略，在 INT4 量化精度下的困惑度损失控制在 1% 以内，量化过程一次性完成，推理时无额外开销。

（2）**KV Cache 专项量化**

KVQuant[54]和 H_2O[55]等工作观察到 KV Cache 中不同 token 的注意力分数分布极不均匀，少数” 重要 token”（Heavy Hitter）承载了绝大多数注意力权重。基于这一发现，这些工作提出选择性保留重要 token 的 KV Cache（完整精度），对其余 token 的 KV Cache 进行激进量化（INT4 甚至 INT2）或直接驱逐，在几乎不影响生成质量的前提下，将 KV Cache 显存占用降低至原始的 25%–50%。

（3）**量化压缩技术的局限**

量化与压缩技术虽能有效降低显存占用，但存在以下局限：量化比特数越低精度损失越大，对推理质量敏感的任务影响尤为显著；低比特量化需要专用的反量化内核支持，不同 GPU 架构的最优实现差异较大；KV Cache 在推理过程中动态生成，无法像静态权重一样进行离线校准量化，在线量化方案会引入额外的计算开销；此外，量化技术主要解决显存容量问题，并不直接解决批处理调度、SLO 感知和资源配置等系统级问题，需要与调度优化协同设计才能发挥最大效果。

2.4　部署配置与资源分配方法

除批处理调度和 KV Cache 管理外，模型的部署配置（Deployment Configuration）对 LLM 推理性能同样具有决定性影响。本节梳理现有的部署配置搜索方法和资源分配策略。

2.4.1　静态部署配置与设备映射

在分布式 LLM 推理中，模型需要跨多个 GPU 进行部署，核心决策包括：GPU 数量的选择和层到设备的映射关系（Device Map）。如表 2-1所示，设备映射策略对推理吞吐量具有显著影响[10]。以 ChatGLM2-6B 在两张异构 GPU（Tesla V100 和 RTX 3090）上的部署为例，将更多层分配至计算能力较强的 GPU（RTX 3090），可使平均吞吐量从 11.19 token/s 提升至 22.55 token/s，提升幅度超过 100%。这一结

果揭示了网络拓扑感知的设备映射对推理性能的关键作用。

表 2-1　不同设备映射策略对 ChatGLM2-6B 推理吞吐量的影响（参考 UELLM[10]）

GPU#0 分配层	GPU#1 分配层	平均吞吐量（token/s）	最大吞吐量（token/s）
layer 0–15	layer 16–32	11.19	11.58
layer 0–19	layer 20–32	13.09	13.48
layer 0–23	layer 24–32	14.85	15.45
layer 0–27	layer 28–32	17.23	18.00
layer 0–31	layer 32	22.55	23.07

传统的静态部署方案通常采用均匀层分配策略，忽略了 GPU 计算能力差异和节点间通信带宽的异构性。在实际生产环境中，集群往往由不同型号的 GPU 混合组成，均匀分配会导致计算能力强的 GPU 长期等待计算能力弱的 GPU 完成计算，形成严重的**木桶效应**（Bottleneck Effect）。

2.4.2　基于元学习的配置搜索：Morphling

Morphling[38]针对云原生模型服务的配置搜索问题，提出了基于模型无关元学习（Model-Agnostic Meta-Learning, MAML）的自动配置框架。Morphling 将部署配置空间定义为 GPU 数量、批大小、模型并行度等超参数的笛卡尔积，在此空间上通过元学习快速拟合性能预测模型，利用已有服务的配置-性能数据以少量新配置的压力测试样本快速泛化至新模型的性能预测，将配置搜索代价从数小时压缩至数分钟。

然而，Morphling 在 LLM 推理场景下面临以下挑战：即便经过元学习加速，仍需对多个候选配置进行真实压力测试，对于参数量超过百亿的 LLM，单次压力测试耗时可达数分钟，严重影响系统响应速度；Morphling 的配置搜索目标为通用推理延迟，未将 KV Cache 动态增长的显存需求纳入约束建模；搜索到的最优配置在部署后固定不变，无法根据负载变化实时调整。

2.4.3　PD 分离架构下的资源分配

随着 PD 分离架构（Prefill-Decode Disaggregation）的兴起，资源分配问题从单实例扩展至跨实例的协同优化。

（1）**DistServe 的静态 PD 配比**

DistServe[1]将 Prefill 实例与 Decode 实例分离部署，以**好吞吐量**（Goodput）最大化为目标，离线求解最优的 Prefill 实例数与 Decode 实例数之比（P:D Ratio）。

DistServe 建立了 Prefill 延迟和 Decode 延迟的解析模型，在给定 SLO 约束下，枚举不同 P:D 配比下的系统好吞吐量并选择最优配比进行部署。然而，DistServe 的 P:D 配比在部署时一次性确定，运行期间固定不变，在实际工作负载的流量高峰期和低谷期均会出现资源利用率失衡的问题。

（2）**Splitwise 的跨机器分离**

Splitwise[31]将 PD 分离的粒度进一步提升至机器级别，Prefill 阶段和 Decode 阶段分别在专用的物理机器上执行，通过高速互联网络（如 InfiniBand）传输 KV Cache。跨机器的物理隔离彻底消除了两阶段间的资源竞争，但也引入了更高的 KV Cache 传输延迟（通常为数十毫秒），在输入较短的场景下传输开销占 TTFT 的比例不可忽视。

（3）**现有 PD 分离资源分配的共同局限**

综合 DistServe 和 Splitwise 的设计可以发现，现有 PD 分离系统在资源分配层面存在以下共同局限：配置粒度粗，资源调整的最小粒度为 GPU 实例或 GPU 池，无法实现 Transformer 层级或注意力头级别的细粒度资源再分配；P:D 实例配比在部署时固定，面对突发流量和负载模式切换时响应迟滞；调度决策受 KV Cache 物理位置约束，前缀缓存感知路由导致热点节点过载；Prefill 阶段计算饱和但显存闲置，Decode 阶段显存紧张但计算利用率低，两者的互补性未能通过动态机制加以利用。

2.5　现有方法的局限性总结与分析

综合本章对批式调度方法、KV Cache 优化技术和部署配置策略的系统梳理，现有 LLM 推理优化研究在以下四个维度存在显著局限，这些局限共同构成了本文后续研究工作的出发点与改进动机。

（1）**调度粒度与资源状态的紧耦合**

现有调度系统（如 SGLang[4]、Mooncake[36]）的路由决策与 KV Cache 的物理位置强绑定，前缀缓存感知路由机制导致负载热点持续倾斜。具体而言，缓存命中率高的实例持续吸引更多请求，形成正反馈循环：高命中率 → 更多请求 → 缓存进一步扩大 → 命中率更高。路由器被迫在计算负载均衡与缓存命中率之间做困难权衡，缺乏计算-缓存联合优化的全局视角，最终导致集群资源利用严重失衡。以实测场景为例，缓存命中率最高的实例往往承载 80% 以上的流量，计算利用率达到 100%，而其他实例利用率仅为 40%–60%[53]，集群整体资源效用严重低下。

（2）**资源配置的静态化与刚性约束**

现有 PD 分离系统（DistServe、Splitwise）在部署时固定 Prefill 与 Decode 实

例的比例，无法在运行期间根据实际负载动态调整。传统部署方法（如 Morphling）的配置搜索开销巨大，每次重新配置需要数分钟的压力测试，难以处理突发流量下的资源重分配需求。实测数据表明，在低请求率（RPS ⩽ 10）场景下，静态配置的系统存在 20%–40% 的 GPU 资源空闲浪费[53]；而在突发高负载场景下，配置调整的滞后性导致服务质量急剧下降，SLO 违约率飙升。

（3）缺乏跨阶段的细粒度资源协同机制

Prefill 与 Decode 阶段的资源需求（计算密集 vs. 内存密集）在时域和空域上具有天然的互补性，但现有系统缺乏在两个阶段之间实时优化资源配置的细粒度机制。如图 2-1所示，实测数据表明：Prefill 实例的计算利用率通常高达 95% 以上，而显存利用率仅约 35%；Decode 实例则相反，计算利用率仅约 35%，而显存利用率高达 90% 以上[53]。这种系统性的资源利用率失衡意味着，若能实现 Prefill 与 Decode 实例之间的细粒度资源共享，理论上可将集群整体利用率提升至 70% 以上。

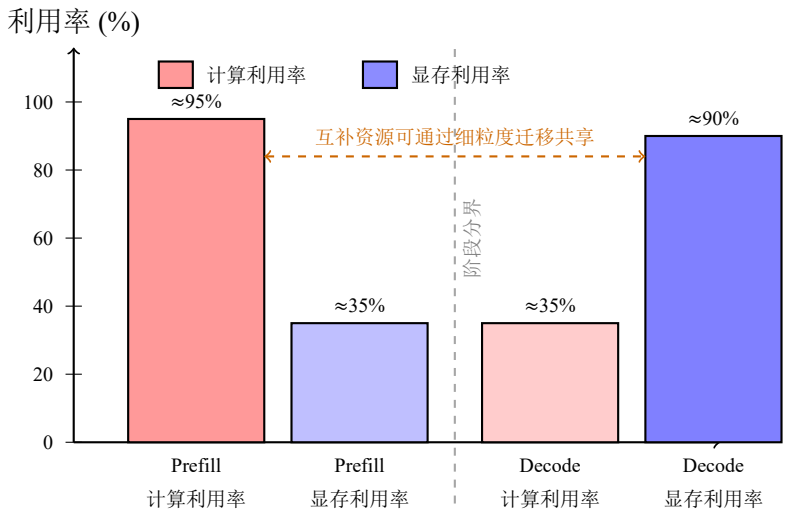


图 2-1　PD 分离架构下 Prefill 与 Decode 实例资源利用率的系统性失衡示意图（数据来源：BanaServe[53]，基于 LLaMA-13B 模型和 Alpaca 数据集的实测结果）。Prefill 实例计算饱和而显存闲置，Decode 实例显存紧张而计算利用率低，两阶段资源需求高度互补，具有通过细粒度迁移实现协同优化的理论潜力。

（4）对异构性和动态性的适应性不足

现有方案多假设同构的 GPU 集群和稳态负载，对混合型号 GPU（如 A100 与 V100 混用）、网络拓扑差异（NVLink 与 PCIe 带宽差距可达 5–10 倍）以及重尾分布（Heavy-tailed Distribution）的动态负载缺乏有效建模[15,35,56]。在实际生产环境中，请求到达率呈现显著的非平稳性（Non-stationarity），输入/输出长度分布随业务场景动态变化。Azure 生产环境的实测数据显示，LLM 推理负载的 RPS 峰谷比可达 5 倍以上，输入序列长度跨越从数十 token 到数万 token 的超宽范围[53]，现有

静态方案在这类动态异构场景下资源碎片化严重，难以维持稳定的 SLO 保障。

表 2-2　现有 LLM 推理优化方法局限性对比

方法	SLO 感知	输出长度预测	动态资源配置	异构拓扑感知	跨阶段协同
FIFO 静态批处理	✗	✗	✗	✗	✗
S³ [3]	✗	✓	✗	✗	✗
Orca [18]	✗	✗	△	✗	✗
vLLM [5]	✗	✗	△	✗	✗
SGLang [4]	✗	✗	△	✗	✗
Morphling [38]	✗	✗	△	✗	✗
DistServe [1]	△	✗	✗	✗	△
Splitwise [31]	△	✗	✗	✗	△
Mooncake [36]	✓	✗	△	✗	△
MemServe [37]	✗	✗	△	✗	△
UELLM [10]	✓	✓	✓	✓	✗
BanaServe [53]	✓	△	✓	△	✓

✓: 完整支持；　△: 部分支持；　✗: 不支持

综合以上四个维度的局限性分析，本文提出两种互补的优化方案，分别从批处理调度与层级资源分配两个层面构建大模型推理服务的完整优化体系：

（1）**UELLM**（第三章）：针对静态场景下批处理策略粗放和部署配置静态化的问题，提出 SLO 感知的动态批处理算法（SLO-ODBS）与异构拓扑感知的高效资源分配算法（HELR），实现从输出长度预测到端到端部署优化的完整技术闭环。UELLM 重点解决上述局限（1）和（2）中的批处理调度与部署配置问题，在真实 4-GPU 异构集群上将推理延迟降低 72.3% 至 90.3%，GPU 利用率提升 1.2 倍至 4.1 倍，吞吐量提高 1.92 倍至 4.98 倍，并实现零 SLO 违约[10]。

（2）**BanaServe**（第四章）：针对 PD 分离架构下资源利用率失衡和缓存感知路由导致的负载倾斜问题，提出层级权重迁移（Layer-level Migration）、注意力级 KV Cache 迁移（Attention-level Migration）和全局 KV Cache Store 等机制，实现细粒度的跨阶段资源协同优化。BanaServe 重点解决上述局限（3）和（4）中的跨阶段协同与动态适应性问题，相比 vLLM 吞吐量提升 1.2 倍至 3.9 倍，延迟降低 3.9% 至 78.4%；相比 DistServe 吞吐量提升 1.1 倍至 2.8 倍，延迟降低 1.4% 至 70.1%[53]。

两种方案在技术层面相互补充：UELLM 解决静态场景下的批处理与部署联合

优化，BanaServe 解决动态场景下的 PD 分离架构资源协同，共同构成覆盖” 静态优化-动态调度” 全生命周期的 LLM 推理服务资源管理框架。

2.6　本章小结

本章系统梳理了 LLM 推理服务中批式任务调度与 KV 缓存优化的现有技术方案，并从多个维度分析了各方法的设计原理、技术贡献与局限性。

在批式任务调度方面，本章介绍了 FIFO、短作业优先、长作业优先等传统调度策略，分析了其在 LLM 推理场景下因输出长度不可知和 SLO 差异被忽视而导致的低效问题。进一步介绍了 S^3 的输出长度感知装箱调度框架、Orca 和 vLLM 中实现的连续批处理（Continuous Batching）与迭代级 In-flight Batching 机制。这些方法在吞吐量提升方面取得了显著进展，但均未能同时解决 SLO 约束感知、异构拓扑适配和跨阶段资源协同等问题。

在 KV 缓存优化方面，本章分析了 vLLM 的 PagedAttention 非连续显存分页管理机制、Mooncake 的全局分布式 KV Cache 池架构、MemServe 的弹性内存池方案，以及低比特量化、KV Cache 专项量化和知识蒸馏等模型压缩技术。这些工作从不同角度缓解了显存墙问题，但在缓存局部性解耦、细粒度跨实例资源共享和动态负载适应性方面仍存在明显不足。

在部署配置与资源分配方面，本章介绍了 Morphling 的元学习配置搜索框架、DistServe 的静态 PD 实例配比优化和 Splitwise 的跨机器 PD 分离方案，指出现有方法在配置粒度、动态响应速度和异构硬件感知能力上的共同局限。

最后，本章从**调度粒度与资源状态紧耦合、资源配置静态化、缺乏跨阶段细粒度协同机制、对异构性和动态性适应性不足**四个维度总结了现有方法的系统性局限，并以表 2-2对各方法进行了量化对比。基于上述分析，明确了本文后续章节提出 UELLM 和 BanaServe 两种优化方案的改进动机与技术切入点：第三章将针对局限（1）和（2）提出 UELLM 框架，第四章将针对局限（3）和（4）提出 BanaServe 框架，从批处理优化和层级资源分配两个层面构建大模型推理服务的统一资源管理体系。

第 3 章　面向 LLM 推理任务的高效批式调度和部署方法：UELLM

3.1　引言

在机器学习即服务（MLaaS）云平台中，大语言模型推理服务面临三类核心挑战：其一，GPU 数量的配置存在双向风险，GPU 数量过多会因通信开销增大而降低推理速度，GPU 数量不足则会因显存溢出（Out-of-Memory, OOM）导致服务中断；其二，静态部署策略无法保证最优的资源利用率和最低的推理延迟；其三，粗放的请求编排策略极易导致大量 SLO 违约。针对上述挑战，本章提出 UELLM

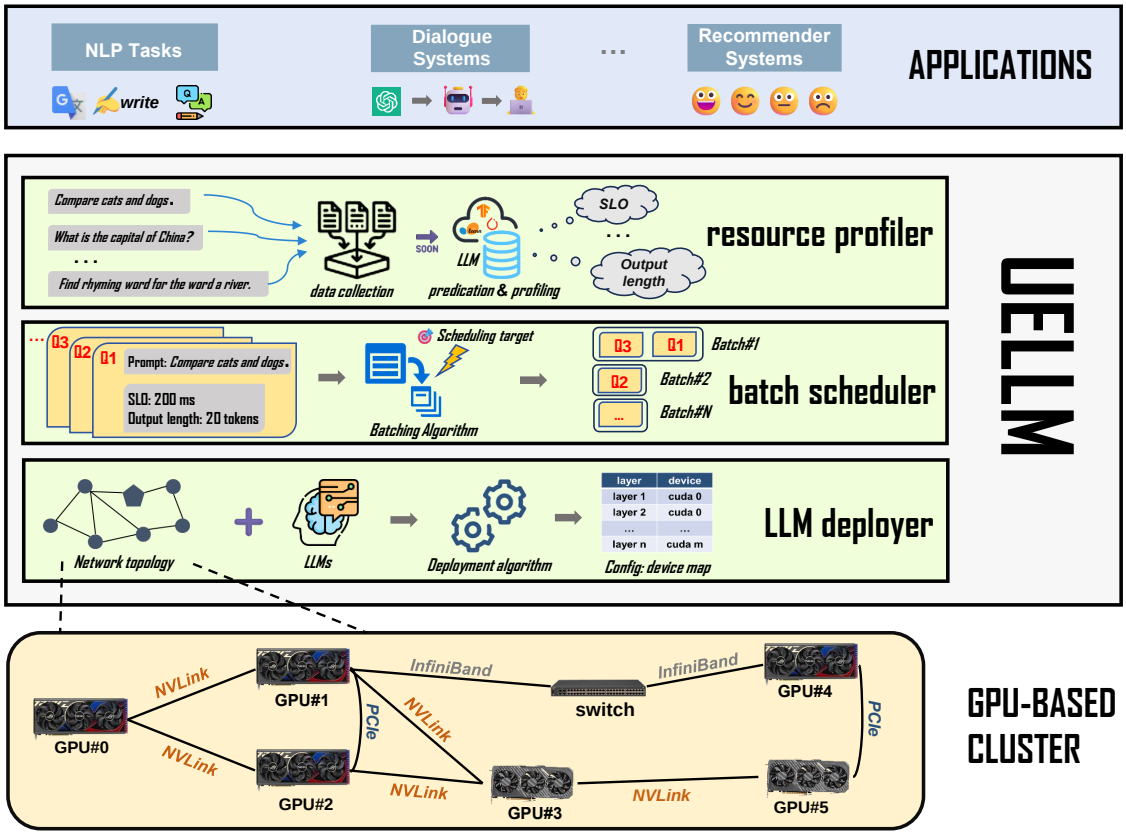


图 3-1　UELLM 系统整体架构图。系统由资源画像器、批处理调度器和 LLM 部署器三个核心组件构成，通过在线监控反馈机制实现端到端的推理服务优化[10]。

（Unified and Efficient LLM Inference Serving）框架。UELLM 将请求批处理优化与模型部署优化有机整合，通过三个协同工作的核心组件：**资源画像器**（Resource Profiler）、**批处理调度器**（Batch Scheduler）和 **LLM 部署器**（LLM Deployer），实现推理服务的端到端优化。系统整体架构如图 3-1所示。

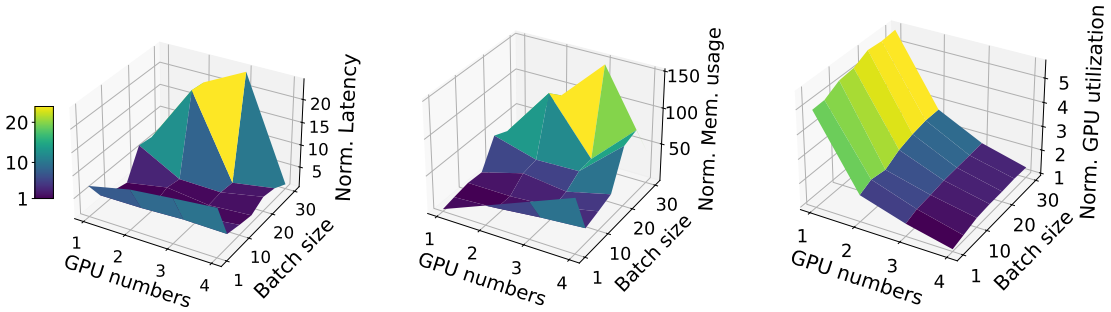
本章其余部分组织如下：第 3.2节分析推理性能瓶颈与设计动机；第 3.3节介绍资源画像器的设计；第 3.4节详述 SLO-ODBS 批处理调度算法；第 3.5节阐述 HELR 部署优化算法；第 3.6节给出实验评估结果；第 3.7节总结本章工作。

3.2　问题分析与设计动机

3.2.1　部署配置对推理性能的影响

观察一：部署配置的微小变化会对 LLM 推理性能产生显著影响。

LLM 的分布式部署涉及两个关键决策：**GPU 数量的选择**和**层到设备的映射关系**（Device Map）。这两个决策的组合空间巨大，且对推理性能的影响极为显著。



(a) 归一化延迟　　(b) 归一化显存占用　　(c) 归一化 GPU 利用率

图 3-2　不同 GPU 数量和批大小配置下归一化延迟、显存占用和 GPU 利用率的变化（各指标均归一化至其最小值）。合理的部署配置可将 GPU 利用率提升 4 倍，延迟降低 20 倍。

图 3-2展示了在不同 GPU 数量和批大小配置下，归一化延迟、显存占用和 GPU 利用率的变化规律。实验结果表明，合理的 GPU 数量配置可将 GPU 利用率提升 4 倍，并将推理延迟降低 20 倍（最坏情况涉及显存卸载时）。这一结果揭示了以下规律：

（1）**GPU 数量并非越多越好**：随着 GPU 数量增加，节点间通信延迟和同步开销显著上升，过多的 GPU 反而会拖慢推理速度；

（2）**设备映射影响巨大**：即使在确定了最优 GPU 数量后，不合理的层分配仍会造成严重的性能退化。如第二章表 2-1所示，通过调整 ChatGLM2-6B 在两张异构 GPU 上的层分配比例，平均吞吐量可从 11.19 token/s 提升至 22.55 token/s，提升幅度超过 100%[10]。

3.2.2　批处理策略对推理性能的影响

观察二：批处理多个请求可在大量推理请求场景下显著降低 SLO 违约率。

批处理通过权重共享机制，允许在单次迭代中并行生成多个 token，从而提升 token 生成速率。然而，LLM 批处理面临两个关键挑战：

（1）**请求到达时间不同步**：简单的批处理策略要么让先到的请求等待后续请求，要么让后到的请求等待前序请求处理完毕，导致显著的排队延迟；

（2）**输入输出长度差异巨大**：现有批处理技术通过填充（Padding）使批内请求长度一致，但这一策略在请求长度差异大时会造成严重的 GPU 计算和显存浪费。以图 3-3所示场景为例：默认 FIFO 策略将三个请求合并为单一批次，共需处理 174 个 token，其中包含大量冗余填充；而 UELLM 的输出长度感知策略将请求分组为两个批次，仅需处理 74 个 token，减少冗余约 57.5%[10]。由于 token 数量与计算和显存开销近似线性相关，减少冗余 token 可直接降低推理延迟和显存使用。

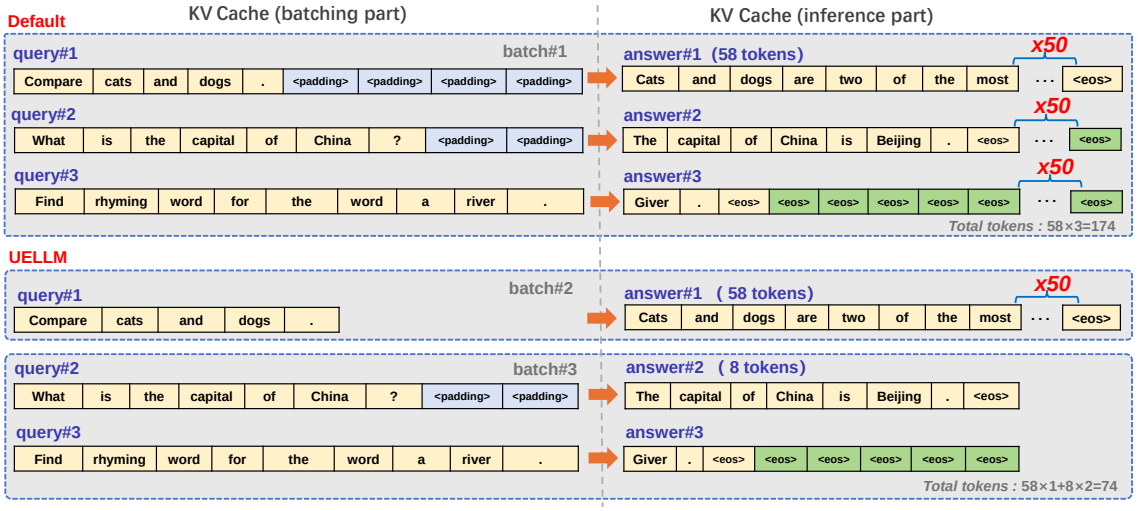


图 3-3　UELLM 与默认批处理算法的对比示意图。该对比展示了三个查询请求在批处理阶段和推理阶段的 KV 缓存利用情况。UELLM 专注于优化 token 使用效率，与默认方法相比，显著减少了推理过程中处理的 token 总数[10]。

3.3　资源画像器

资源画像器（Resource Profiler）是 UELLM 的数据入口模块，负责在请求进入调度队列之前，预测其输出长度并获取 SLO 约束，为后续批处理调度和部署优化提供决策依据。资源画像器由三个子模块构成：**数据采集**、**输出长度预测**和**资源画像**。

3.3.1　基于微调大模型的输出长度预测

（1）**预测模型选择与微调策略**

UELLM 采用 ChatGLM3-6B[57]作为输出长度预测的基础模型，通过 LoRA（Low-Rank Adaptation）[26]高效微调策略，在 Alpaca[43]问答数据集上进行任务适

配。微调时，以用户问题文本作为输入，以对应答案的 token 数量所属桶编号作为分类标签。

具体而言，输出长度被离散化为 K 个桶（Bucket）：

$$\mathcal{B} = \{[0,8),\ [8,16),\ [16,32),\ [32,64),\ \cdots,\ [1024,2048)\}, \tag{3-1}$$

每个桶对应一个类别标签，将连续的长度回归问题转化为 K 类分类问题，显著降低了预测复杂度，同时保证调度决策的快速响应。

（2）**预测精度评估**

微调后的预测模型在测试集上达到了 **99.51%** 的分桶准确率。为验证模型的泛化能力，进一步在 Google Natural Questions 数据集和 Alpaca GPT-4 数据集上进行评估，准确率均超过 80%，表明模型对不同问答分布具有良好的泛化性能[10]。相比 S^3[3] 采用的轻量级 BERT 分类器，UELLM 选择微调同类大模型（ChatGLM3-6B）作为预测器，具有以下优势：

（1）**语义理解深度**：大模型对问题语义的理解更为深入，能够捕捉影响答案长度的细粒度语言特征（如问题类型、领域、复杂度等）；

（2）**预测精度显著更高**：99.51% 的分桶准确率远高于基于规则或轻量级分类器的方案；

（3）**在线学习能力**：支持基于实时反馈的增量更新，适应输出长度分布的动态变化。

3.3.2　SLO 获取与资源画像

资源画像器在预测输出长度的同时，从请求元数据中提取 SLO 约束（即请求必须在该时间内完成推理的截止时间）。对于每个请求 q_i，资源画像器输出一个三元组：

$$\text{Profile}(q_i) = (\text{Input}_i,\ \widehat{\text{Output}}_i,\ \text{SLO}_i), \tag{3-2}$$

其中 Input_i 为实际输入长度，$\widehat{\text{Output}}_i$ 为预测输出长度桶的代表值，SLO_i 为请求的延迟截止时间。该三元组作为批处理调度器的输入，支持后续的 SLO 感知批次构建决策。

3.3.3　在线监控与自适应校正

UELLM 在后台运行实时监控程序，持续追踪 GPU 利用率、显存占用和推理执行时间等指标。当监控程序检测到预测输出长度与实际输出长度存在显著偏差时，触发自适应校正机制：

（1）**显存动态调整**：根据实际 KV Cache 增长速率调整已分配显存大小，防止 OOM 错误；

（2）**预测器在线更新**：将预测错误的样本加入在线学习队列，周期性地对预测模型进行增量微调，提升对动态负载分布的适应能力。

3.4　批处理调度器：SLO-ODBS 算法

批处理调度器（Batch Scheduler）接收资源画像器输出的请求画像序列，将请求组合为最优批次，目标是在满足 SLO 约束的前提下最小化推理延迟和显存浪费。

3.4.1　问题建模

设批次 $\text{batch}_c = \{q_1, q_2, \cdots, q_b\}$ 包含 b 个请求，由于批处理要求批内所有输入等长，每个请求需被填充至批内最大输入长度 $\max_{i=1}^{b}(\text{Input}_i)$。在推理阶段，模型需将批内所有输出生成至最大输出长度 $O = \max_{i=1}^{b}(\text{Output}_i)$，因此批次内实际生成的 token 总数为 $b \times O$。

批处理的核心优化目标是通过合理分组，减少批内最大输出长度 O，从而降低冗余 token 数量。具体地，UELLM 的优化目标综合考虑**总延迟** T_l 和**总输出长度** T_o 两个维度。

总延迟 T_l 定义为：

$$T_l = \sum_{i=1}^{N} \left[(\text{SLO}_i + L_{CM}) \times (|\text{batch}_c| + 1) \times L_1\right], \tag{3-3}$$

总输出长度 T_o 定义为：

$$T_o = \sum_{i=1}^{N} \left[(\text{Length}_i + O_{CM}) \times (|\text{batch}_c| + 1) \times L_2\right], \tag{3-4}$$

其中，N 为当前批次 batch_c 中的请求数量，L_{CM} 为当前批次的最大延迟，O_{CM} 为当前批次的最大输出长度，L_1 和 L_2 分别为并行计算引入的额外延迟系数。通过权重系数 w_1 和 w_2 平衡两个优化目标的重要性，综合优化目标为：

$$\max_{w_1, w_2} (w_1 \times T_l + w_2 \times T_o) \leqslant \text{Threshold}, \tag{3-5}$$

系统通过确保每次加入新请求后批次的综合指标 Total $\leqslant T$（阈值），将请求序列重新组合为 $\text{batch}_1, \text{batch}_2, \cdots, \text{batch}_n$。

3.4.2　SLO-ODBS 算法设计

基于上述问题建模，本文提出 SLO 感知与输出长度驱动的动态批处理调度算法（SLO and Output-Driven Dynamic Batch Scheduler, SLO-ODBS），如算法 3-1所示。

算法 3-1　SLO 感知与输出长度驱动的动态批处理调度算法

Input: requests：经资源画像器处理后的请求列表
Output: batches：批次列表

```
1  sorted_requests ← sort(requests) ;                    // 按 SLO 升序排列
2  batches, batch_c ← ∅;
3  L_CM, O_CM, CM ← 0;
4  foreach q ∈ sorted_requests do
5      T_l ← (q.SLO + L_CM) × (len(batch_c) + 1) × L_1;
6      T_o ← (q.length − O_CM) × (len(batch_c) + 1) × L_2;
7      Total ← w_1 × T_l + w_2 × T_o;
8      if batch_c = ∅ or Total ⩽ Threshold then
9          batch_c.append(q.index);
10         L_CM ← max(L_CM, q.SLO);
11         O_CM ← max(O_CM, q.length);
12         CM ← max(CM, w_1 × q.length + w_2 × q.SLO);
13     else
14         batches.append(batch_c);
15         batch_c ← {q.index};
16         L_CM ← q.SLO;
17         O_CM ← q.length;
18         CM ← w_1 × q.length + w_2 × q.SLO;
19     end
20     根据 CM 的值动态调整批大小;
21 end
22 if batch_c ≠ ∅ then
23     batches.append(batch_c);
24 end
25 return batches;
```

算法执行分三个阶段：

阶段一：初始化（第 1–3 行）

所有请求按 SLO 升序排列，优先处理 SLO 最紧迫的请求，同时初始化当前批次 batch_c 及其属性（当前最大延迟 L_{CM}、当前最大输出长度 O_{CM}、当前综合指标 CM）。按 SLO 排序的设计使得 SLO 接近的请求更容易被组合入同一批次，天然降低批内的 SLO 异构性。

阶段二：基于输出长度的单批次构建（第 4–19 行）

对每个请求 q，计算将其加入当前批次后的综合指标 Total。若 Total $\leqslant$ Threshold，则将请求加入当前批次并更新批次属性；否则，将当前批次提交至批次列表，并以当前请求初始化新批次。批大小根据综合指标 CM 动态调整，避免批次过大或过小导致的效率损失。

阶段三：合并排序所有批次（第 20–23 行）

将阶段二构建的所有批次整合至就绪列表，供后续 LLM 推理引擎处理。

3.4.3　算法变体：灵活适配不同调度目标

SLO-ODBS 通过调整权重系数 w_1 和 w_2，可灵活适配不同的调度优化目标：

（1）**当 $w_1 = 0$ 时**（SLO-DBS 模式）：算法退化为纯 SLO 感知调度，仅以 T_o 为批次构建依据，优先降低 SLO 违约率，适合对服务质量保障要求严格的场景；

（2）**当 $w_2 = 0$ 时**（ODBS 模式）：算法退化为纯输出长度驱动调度，仅以 T_l 为依据，优先最小化推理延迟，适合对吞吐量要求高但对 SLO 公平性要求相对宽松的场景；

（3）**当 $w_1, w_2 > 0$ 时**（完整 SLO-ODBS 模式）：算法同时兼顾 SLO 约束与输出长度差异，在低延迟和低违约率之间寻求最优平衡，适合生产环境的通用部署。

3.5　LLM 部署器：HELR 算法

LLM 部署器（LLM Deployer）在模型部署阶段运行，根据当前集群的硬件网络拓扑和 LLM 的计算特性，自动求解最优的层到设备映射（Device Map），以最大化资源利用率并最小化推理延迟。

3.5.1　问题形式化建模

将集群硬件网络抽象为图 $G = (D, E)$，其中 $D = \{d_1, d_2, \cdots, d_n\}$ 为硬件设备节点集合，E 为节点间的连接边集合。对于待部署的大模型 llm_i，其显存需求为 $M(\text{llm}_i)$，模型层数为 $\text{Layer}(\text{llm}_i)$。每个节点 d_i 具有以下属性：

（1）$\text{Memory}(d_i)$：节点 d_i 的可用显存容量；

（2）$\text{Performance}(d_i)$：节点 d_i 的计算处理能力；

（3）$\text{Latency}(E[i][j])$：节点 d_i 与 d_j 之间的通信延迟。

目标是寻找设备分配方案 $S \subseteq D$，在满足显存约束的前提下最小化推理延迟：

$$\sum_{i=1}^{|S|} \text{Memory}(d_i) \geqslant M. \tag{3-6}$$

3.5.2　HELR 算法设计

该问题可建模为一个动态规划（Dynamic Programming）问题。本文提出高效低延迟资源分配算法（High-Efficiency Low-Latency Resource Allocation, HELR），如算法 3-2所示。

算法 3-2　高效低延迟资源分配算法

Input: M：LLM 显存需求；Layer(M)：模型层数；$G(D,E)$：硬件平台图；
Latency$(E[i][j])$：节点间通信延迟；Performance(d)：设备计算性能；
Memory(d)：设备可用显存
Output: Device_map：层到设备的映射方案

初始化 best_state ← ∞，Device_map ← ∅;
foreach n *from* 1 *to* $|D|$ **do**
　S_n ← 从 D 中选取大小为 n 的子集;
　if S_n 中节点的总显存 $<$ M **then**
　　跳过，继续下一个子集;
　end
　初始化 dp[mark][i] ← ∞;
　foreach *mark from* 1 *to* 2^n-1 **do**
　　按性能和显存降序排列 S_n 中的节点;
　　计算每层所需显存 $m \leftarrow M/\text{Layer}(M)$;
　　foreach i *from* 1 *to* $|S_n|$ **do**
　　　foreach j *from* 1 *to* $|S_n|$，$j \neq i$ **do**
　　　　// T 为 KV Cache 预留显存
　　　　计算节点 i 可分配的最大层数：
　　　　$$\text{layers}[i] \leftarrow \min\left(\text{Layer}(M),\ \frac{\text{Memory}(i)-T}{m}\right);$$
　　　　计算延迟：$l \leftarrow dp[i][j] + \text{Latency}(E[i][j]) + p \times \frac{\text{layers}[i]\times m}{\text{Performance}(i)}$;
　　　　更新 dp[mark][i] ← min(dp[mark][i], l);
　　　end
　　end
　end
　current_state ←
　$$\min_i\left(dp[2^n-1][i] + \sum_{j=1}^{|S_n|}\left(\text{Latency}(E[i][j]) + p \times \frac{\text{layers}[i]\times m}{\text{Performance}(i)}\right)\right);$$
　if *current_state* $<$ *best_state* **then**
　　best_state ← current_state;
　　根据 layers 和 S_n 中的节点更新 Device_map;
　end
end
return *Device_map*;

算法执行分三个阶段：

阶段一：初始化（第 1–2 行）

设置最优状态 best_state 为正无穷，初始化空的设备映射方案 Device_map。

阶段二：动态规划求解（第 3–16 行）

维护二维数组 dp，其中 $dp[\text{mark}][i]$ 表示从初始状态到设备节点 d_i 的最小延迟，mark 为状态压缩的位掩码。动态规划的状态转移方程为：

$$dp[\text{mark}][i] = \min_{1 \leqslant j \leqslant |S|} \left(dp[\text{mark}][i],\ dp[i][j] + \text{Latency}(E[i][j]) + p \times \frac{\text{layers}[i] \times m}{\text{Performance}(i)} \right), \quad (3\text{-}7)$$

最终的优化目标为最小化总延迟：

$$\min_{1 \leqslant i \leqslant |S|} dp[2^k - 1][i] + \sum_{j=1}^{|S|} \left(\text{Latency}(E[i][j]) + p \times \frac{\text{layers}[i] \times m}{\text{Performance}(i)} \right), \quad (3\text{-}8)$$

其中 layers$[i]$ 为分配给节点 d_i 的层数，p 为处理性能-时间关系的调节系数。

阶段三：设备映射更新（第 17–19 行）

记录最优分配状态 best_state，并根据 layers 数组和节点集合 S_n 更新最终的 Device_map，确保部署配置反映最优资源分配方案。

3.5.3　HELR 的算法变体

类似于 SLO-ODBS，HELR 同样通过调整权重参数 a_1 实现不同部署目标的灵活适配：

（1）**高效模式（HE，High-Efficiency）**：设 $a_1 = 0$，动态规划仅最小化 GPU 数量，优先在尽可能少的 GPU 上完成部署，充分利用每个 GPU 的计算能力，适合资源受限的环境；由于 LLM 推理的顺序执行特性，在最少 GPU 上部署可有效提升 GPU 利用率；

（2）**低延迟模式（LR，Low-Latency）**：设 $a_1 \gg a_2$（如 10:1），动态规划赋予延迟最高权重，在不考虑资源消耗的前提下追求最低推理延迟，适合对响应时间极为敏感的场景；

（3）**均衡模式（HELR）**：同时考虑 GPU 利用率和推理延迟，在资源效率与服务质量之间寻求最优权衡，适合生产环境的通用部署场景。

3.6　实验评估

3.6.1　实验设置

测试平台：实验在由 4 块 NVIDIA RTX 3090 GPU 组成的本地集群上进行。为模拟异构集群环境，对各 GPU 设置不同的性能限制。集群网络拓扑如表 3-1所示，节点间连接类型分为三种：自身连接（X）、单 PCIe 桥连接（PIX）和 NUMA 节点内 PCIe 互联（NODE）。

表 3-1　实验集群网络拓扑及 GPU 最大功耗配置

	GPU#0	**GPU#1**	**GPU#2**	**GPU#3**	**最大功耗**
GPU#0	X	PIX	NODE	NODE	350 W
GPU#1	PIX	X	NODE	NODE	300 W
GPU#2	NODE	NODE	X	PIX	250 W
GPU#3	NODE	NODE	PIX	X	150 W

X = 自身；　PIX = 单 PCIe 桥连接；　NODE = PCIe + NUMA 节点内互联

推理模型：选用 ChatGLM2-6B[57]作为推理目标模型。由于该模型在实验时尚不支持批推理，UELLM 对其推理代码进行适配以支持批处理。

SLO 设置：为贴近真实场景，为不同推理请求设计随机 SLO，范围从 1 秒到 350 秒，确保每个请求的 SLO 完全随机。

对比基线：构建三个 UELLM 原型版本：

（1）**UELLM-deploy（UD）**：仅使用 HELR 模型部署算法；

（2）**UELLM-batch（UB）**：仅使用 SLO-ODBS 批处理调度算法；

（3）**UELLM-all（UA）**：同时采用 HELR 和 SLO-ODBS 两种算法。

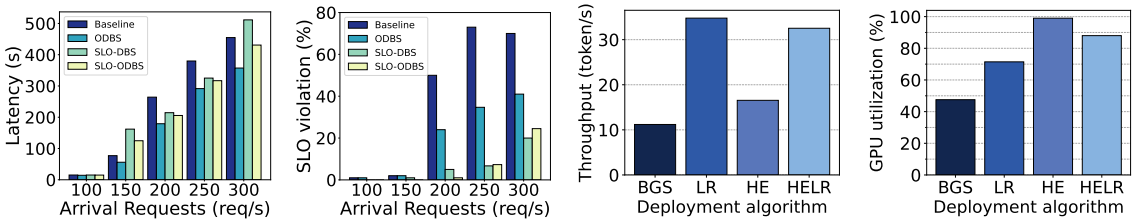
选取 Morphling[38]和 S^3[3]作为 SOTA 基线进行对比。

评估指标：采用四项核心指标：推理延迟（Latency）、吞吐量（Throughput，token/s）、GPU 利用率（GPU Utilization）和 SLO 违约率（SLO Violation Rate）。每组实验重复 5 次，取平均值以消除随机性影响。

3.6.2　批处理算法与部署算法的消融分析

图 3-4展示了不同批处理算法和部署算法的性能对比。

批处理算法对比分析：实验结果（图 3-4(a)(b)）表明，在高负载场景下，SLO-ODBS 通过合理组合请求减少迭代次数和显存开销，维持与 ODBS 相近的低延迟；同时按 SLO 调度请求，实现接近 SLO-DBS 的低违约率。SLO-ODBS 成功地在低延迟和低 SLO 违约率两个目标之间取得了最佳平衡。



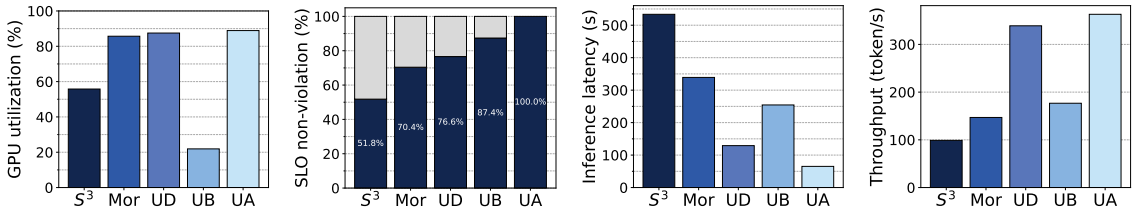
(a) 推理延迟　(b) SLO 违约率　(c) 吞吐量　(d) GPU 利用率

图 3-4　不同批处理算法与部署算法的性能对比。在高负载下，SLO-ODBS 通过合理组合请求，维持与 ODBS 相近的低延迟，同时实现接近 SLO-DBS 的低 SLO 违约率，兼顾了两个优化目标；HELR 在维持较高 GPU 利用率的同时实现了较高吞吐量。

部署算法对比分析：HELR 由于选择了更合理的资源分配和部署方案，在维持与 HE 相近的 GPU 利用率的同时，实现了与 LR 相近的吞吐量，验证了均衡模式的有效性。

3.6.3　与 SOTA 方法的综合对比

图 3-5对比了 UELLM 与当前最优算法 S^3 和 Morphling 在各项指标上的性能差异，呈现了各算法在五个不同时间段内的平均表现。



(a) GPU 利用率　(b) SLO 不违约率　(c) 推理延迟　(d) 吞吐量

图 3-5　UELLM 各版本与 S^3、Morphling 在四项核心指标上的综合对比。UA（UELLM-all）在所有指标上均取得最优性能：SLO 不违约率达 100%，推理延迟最低，吞吐量最高。

GPU 利用率分析（图 3-5(a)）：由于 S^3 和 UB 仅关注批处理调度而未优化部署策略，其 GPU 利用率显著低于其他方法，说明这两种方案存在资源利用效率不足的问题。Morphling 通过元学习搜索最优配置并进行压力测试，而 UA 和 UD 则利用 HELR 算法根据当前节点拓扑结构和模型特性选择最佳资源配置方案，因此三者的 GPU 利用率结果非常接近。实验结果表明，UELLM 以更低的计算代价实现了与 Morphling 相当的部署效果。

SLO 不违约率分析（图 3-5(b)）：五种方案的 SLO 不违约率由低到高依次为：S^3（51.8%）< Morphling（70.4%）< UD（76.6%）< UB（87.4%）< UA（100.0%）。

（1）S^3 仅考虑显存优化，不感知各请求的 SLO 差异，表现最差；

（2）Morphling 和 UD 均仅优化部署策略而不考虑 SLO 调度，虽然推理延迟

较低，但 SLO 不违约率仍低于 UB；

（3）UB 通过 SLO-ODBS 算法感知 SLO，将 87.4% 的请求控制在 SLO 内；

（4）UA 同时优化部署策略和批处理调度，实现了**零 SLO 违约**，所有请求均在截止时间内完成。

相比 S^3 和 Morphling，UELLM（UA）将 SLO 不违约率分别提升了 48.2% 和 29.6%。

推理延迟与吞吐量分析（图 3-5(c)(d)）： UA 在推理延迟和吞吐量两个指标上均取得最优结果：

（1）相比 Morphling，UD 通过消除压力测试环节，直接降低了测试阶段的时间开销，从而减少整体推理延迟并提升吞吐量；

（2）相比 S^3，UD 通过 HELR 优化资源分配，改善 GPU 利用率和通信延迟，显著降低推理延迟；

（3）UB 通过 SLO-ODBS 优化批次组合，减少迭代次数，降低推理延迟并提升吞吐量；

（4）UA 综合 HELR 和 SLO-ODBS 的双重优化，在五次实验中均取得最优结果。

总体而言，相比 S^3 和 Morphling，UELLM 将推理延迟降低 **72.3% 至 90.3%**，吞吐量提升 **1.92 倍至 4.98 倍**[10]。

3.6.4　实验结果综合分析

综合上述实验结果，表 3-2对 UELLM 各版本与基线方法的性能进行了系统性汇总。

表 3-2　UELLM 与基线方法性能综合对比

方法	GPU 利用率	SLO 不违约率	推理延迟	吞吐量	综合评价
S^3 [3]	低	51.8%	高	低	基线（批处理）
Morphling[38]	高	70.4%	中	中	基线（部署）
UD	高	76.6%	中低	中高	仅部署优化
UB	低	87.4%	低	高	仅批处理优化
UA	**高**	**100.0%**	**最低**	**最高**	**最优（联合优化）**

实验结果验证了 UELLM 设计的三个核心判断：

（1）**批处理优化与部署优化缺一不可**：单独的批处理优化（UB）或单独的部署优化（UD）均无法在所有指标上同时取得最优结果，只有联合优化（UA）才能实现零 SLO 违约、最低延迟和最高吞吐量的同时满足；

（2）**SLO 感知是批处理调度的关键维度**：不具备 SLO 感知能力的 S^3 虽能优化延迟和显存，但 SLO 违约率高达 48.2%，难以保障实际生产场景的服务质量；

（3）**拓扑感知的动态规划优于元学习配置搜索**：HELR 通过一次动态规划直接求解最优设备映射，无需 Morphling 的压力测试环节，在降低配置开销的同时实现了相当甚至更优的部署效果。

3.7　本章小结

本章提出了 UELLM，一个面向 LLM 推理服务的统一高效资源调度框架。UELLM 由三个协同工作的核心组件构成：

资源画像器通过微调 ChatGLM3-6B 实现高精度输出长度预测（分桶准确率 99.51%），并提取请求的 SLO 约束，为调度决策提供精确的输入信息。相比 S^3 采用的轻量级分类器，基于大模型微调的预测方案在预测精度和在线适应性方面具有显著优势。

批处理调度器提出 SLO-ODBS 算法，将 SLO 感知与输出长度驱动的批次构建相结合，通过三阶段动态调度（初始化 → 批次构建 → 批次排序）实现了低延迟与低 SLO 违约率的双重目标。SLO-ODBS 通过调整权重参数 w_1、w_2 可灵活派生出 SLO-DBS 和 ODBS 两种变体，支持不同调度目标的定制化配置。

LLM 部署器提出 HELR 算法，将 LLM 部署建模为拓扑感知的动态规划问题，在满足显存约束的前提下自动求解最优设备映射。HELR 通过调整参数 a_1 可派生出 HE（高效利用率）和 LR（低延迟优先）两种部署模式，适配不同的资源约束场景。

在由 4 块 NVIDIA RTX 3090 GPU 构成的异构集群上的实验结果表明，相比 S^3 和 Morphling 等 SOTA 方法，UELLM（UA 版本）将推理延迟降低 72.3% 至 90.3%，GPU 利用率提升 1.2 倍至 4.1 倍，吞吐量提高 1.92 倍至 4.98 倍，并实现零 SLO 违约。UELLM 的成功验证了**批处理调度优化与部署配置优化的联合设计**是提升 LLM 推理服务性能的关键路径。

然而，UELLM 仍存在一定局限性：其设计聚焦于单实例或小规模集群的批处理与部署优化，未深入考虑大规模 PD 分离架构下跨实例的细粒度资源协同问题。具体而言，在 Prefill 与 Decode 实例分离部署的场景下，两阶段之间的资源互补性（Prefill 计算密集但显存闲置，Decode 显存密集但计算利用率低）未能通过动态机

制加以利用，这一问题将在下一章 BanaServe 框架中重点解决。

第 4 章　面向 PD 分离架构的动态资源协同优化方法：BanaServe

4.1　引言

随着大语言模型在生产环境中的大规模部署，推理服务系统正面临前所未有的资源管理挑战。以 Prefill-Decode（PD）分离为代表的新型架构通过将提示预填充（Prefill）阶段与自回归解码（Decode）阶段分配至不同 GPU 实例，有效消除了两阶段间的资源竞争干扰。然而，现有 PD 分离系统在实际动态工作负载下暴露出三类根本性局限：

（1）**静态资源配置无法适应非平稳工作负载**：当请求速率较低（RPS $\leqslant$ 10）时，Prefill 和 Decode 实例均存在 20%–40% 的 GPU 资源空闲浪费；而在突发高流量时，固定配置又成为吞吐量瓶颈；

（2）**PD 两阶段间存在内生的资源利用率不均衡**：Prefill 阶段计算密集（计算利用率约 95%，显存利用率约 35%），而 Decode 阶段恰好相反（计算利用率约 35%，显存利用率约 90%），这种互补性资源失衡在现有系统中无法被动态利用；

（3）**前缀缓存感知路由引发持续性负载倾斜**：以 SGLang[4] 和 vLLM[5] 为代表的系统在路由请求时优先考虑 KV Cache 命中率，导致命中率高的 Prefill 实例持续吸引更多请求，形成计算热点，而其他实例资源闲置。

上述三类局限共享同一根本原因：**资源分配与状态（缓存）放置的紧耦合**。现有系统中，计算资源与 KV Cache 等状态数据绑定于特定实例，难以在不引入巨大开销的前提下动态迁移，导致负载均衡与缓存局部性之间存在不可调和的矛盾。

针对上述挑战，本章提出 **BanaServe**，一个面向 PD 分离架构的动态编排框架，通过三项核心创新将资源分配与状态管理解耦：

（1）**层级权重迁移**（Layer-level Weight Migration）：在 Transformer 层粒度上动态迁移模型权重，实现 Prefill 与 Decode 实例之间粗粒度的计算和显存资源再平衡；

（2）**注意力级 KV Cache 迁移**（Attention-level KV Cache Migration）：沿注意力头维度分割 KV Cache，将部分 KV 状态迁移至空闲 GPU 进行并行计算，实现细粒度的负载均衡；

（3）**全局 KV Cache 存储**（Global KV Cache Store）：构建跨所有 Prefill 实例的统一 KV Cache 存储层，结合层级流水线重叠传输技术，使路由器摆脱缓存位置

约束，实现纯负载感知调度。

BanaServe 在 vLLM 和 DistServe 等主流推理栈上实现，并在 Alpaca 短文本和 LongBench 长文本等多种负载场景下进行评估。相比 vLLM，BanaServe 吞吐量提升 1.2 倍至 3.9 倍，总处理时间降低 3.9% 至 78.4%；相比 DistServe，吞吐量提升 1.1 倍至 2.8 倍，延迟降低 1.4% 至 70.1%[53]。

本章其余部分组织如下：第 4.2节分析设计动机；第 4.3节阐述系统设计；第 4.4节建立性能优化模型；第 4.5节详述核心算法；第 4.6节给出实验评估；第 4.7节总结本章工作。

4.2　问题分析与设计动机

4.2.1　静态配置导致的资源低效利用

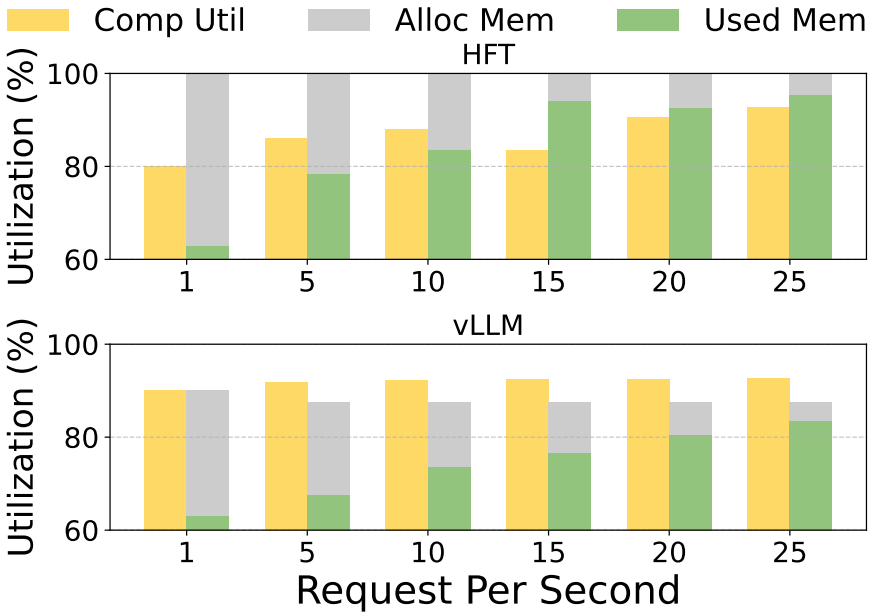


图 4-1　HFT 和 vLLM 在不同请求速率下的 GPU 资源利用率对比（基于单台 A100 GPU 部署 LLaMA-13B 模型，每组实验重复 5 次）。在低负载（RPS ⩽ 10）场景下，两个系统均存在约 20%–40% 的 GPU 资源空闲浪费。

现有 LLM 推理系统在静态分配策略下频繁出现资源利用率不足的问题，其根源在于两个方面：（1）在低请求率（RPS ⩽ 10）条件下，系统大量计算和显存资源处于闲置状态；（2）固定资源分配与 LLM 推理动态工作负载特性之间存在根本性的失配——满足存储需求往往制约可用的计算能力，反之亦然。

图 4-1展示了在单台 A100 GPU 上部署 LLaMA-13B 模型时，Hugging Face Transformers（HFT）和 vLLM 在不同请求速率下的 GPU 资源利用率对比。实测结果表明，在低负载（RPS ⩽ 10）条件下，两个系统均存在约 20%–40% 的 GPU 资

源空闲浪费，验证了静态配置在资源效率方面的固有缺陷。

4.2.2　前缀缓存感知路由引发的负载倾斜

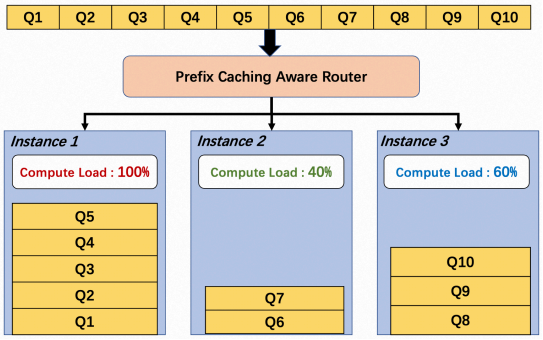
图 4-2展示了前缀缓存感知路由器在三个推理实例间分配请求的典型场景。该策略同时考虑缓存命中率和当前负载，但系统性地偏向命中率高的实例，导致持续性的负载倾斜：

（1）**实例 1**：命中率最高，接收大多数查询，计算利用率达 100%，缓存前缀 Q1–Q5 持续扩大，形成正反馈；

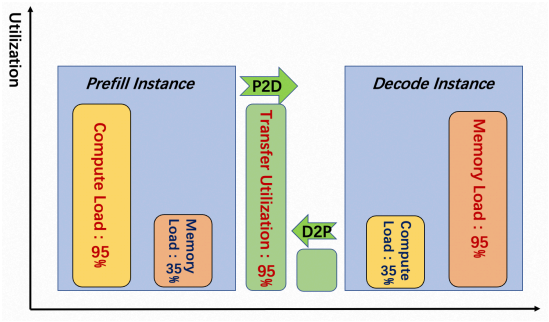
（2）**实例 2**：接收较少查询，计算利用率仅 40%，存储的前缀 Q6–Q7 与其他实例存在冗余；

（3）**实例 3**：计算利用率 60%，主要处理未缓存前缀 Q8–Q10，需重复计算，丧失缓存收益。

这种分配模式形成正反馈动态：命中率高的实例吸引更多请求并扩大缓存内容，而低命中率实例处理较少查询却存储冗余数据或执行重复计算，最终导致排队延迟增加，系统整体吞吐量下降。



(a) 前缀缓存感知路由导致的负载倾斜



(b) PD 分离架构中的资源利用率不均衡

图 4-2　现有 LLM 推理架构的两类实测局限（。(a) 前缀缓存感知路由导致命中率高的实例持续过载，形成负载倾斜、冗余存储和重复计算；(b) PD 分离架构下 Prefill 实例计算密集（约 95%）但显存闲置（约 35%），Decode 实例恰好相反，资源利用率高度互补但不均衡。

4.2.3　PD 分离架构的内生资源不均衡

如图 4-2(b)所示，基于 DistServe 框架、LLaMA-13B 模型和 Alpaca 数据集的实测结果表明：

（1）**Prefill 实例**：处理长提示序列，计算密集，通常维持超过 95% 的计算利用率，但显存利用率仅约 35%；

（2）**Decode 实例**：执行迭代式自回归生成，利用 Prefill 阶段生成的 KV Cache 降低计算量，计算利用率约 35%，但显存利用率显著更高；

（3）**通信特征**：Prefill 实例需向 Decode 实例传输 KV Cache 数据，导致单向带宽占主导，其他通信方向利用率低下。

这种计算-显存资源利用率的系统性不均衡在现有 PD 分离系统中无法通过动态机制加以利用，成为制约整体资源效率的核心瓶颈。

4.3　系统设计

BanaServe 的整体架构由三个核心模块协同构成：

（1）**负载感知请求调度器**：接收传入请求队列，基于各 Prefill 实例的实时归一化综合利用率（计算 + 显存）进行排序，将请求分配至负载最低的实例，摆脱对缓存位置的依赖；

（2）**全局 KV Cache 存储**：由 CPU 内存和 SSD 共同构成存储层次，支持跨所有 Prefill 实例的统一 KV Cache 共享，结合三阶段层级流水线重叠传输技术（预取-计算-存储并行），将通信延迟隐藏于计算过程中；

（3）**动态模块迁移引擎**：周期性监控各 GPU 实例的实时负载，根据负载差距 Δ 和迁移效率比 ρ 自适应选择层级权重迁移（粗粒度）或注意力级 KV Cache 迁移（细粒度），实现 Prefill 与 Decode 实例之间的动态资源再平衡。

三个模块协同工作：调度器的纯负载感知决策由全局 KV Cache 存储提供支撑（消除缓存局部性约束），迁移引擎则持续监控并调整系统资源分配，形成闭环的自适应优化机制。

4.3.1　层级权重迁移

（1）**设计原理**

在工作负载不均衡程度较高的场景下，BanaServe 将一组连续的 Transformer 层从一个 GPU 实例迁移至另一个 GPU 实例。这种粗粒度的重分配实现了动态模型并行，通过同时迁移层权重和对应的 KV Cache，在 Prefill 与 Decode 阶段之间重新分配大量计算和显存资源。

如图 4-3(a)所示，设迁移层的权重大小为 S_ℓ^w，对应 KV Cache 大小为 S_ℓ^{kv}，总迁移数据量为：

$$S_\ell^{\text{total}} = S_\ell^w + S_\ell^{kv}, \tag{4-1}$$

给定有效互联带宽 B_{net} 和同步开销 T_{sync}，层级迁移延迟近似为：

$$T_{\text{layer}} \approx \frac{S_\ell^{\text{total}}}{B_{\text{net}}} + T_{\text{sync}}, \tag{4-2}$$

由于 $S_\ell^w \gg S_\ell^{kv}$，延迟主要由权重传输主导，但在高带宽数据中心互联（NVLink、

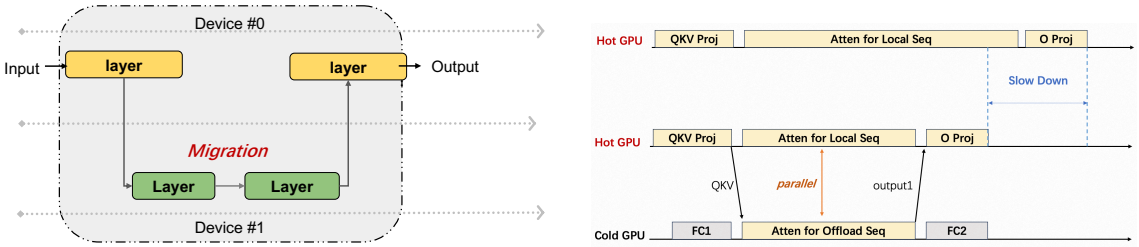
InfiniBand）环境下仍保持实用性。

（2）**执行正确性保证**

对于已迁移的层 $f_\ell(\cdot)$，其权重 W_ℓ 和 KV Cache $\mathcal{K}_\ell$ 同步迁移至目标 GPU 后，计算语义保持不变：

$$\mathbf{y}_\ell = f_\ell(\mathbf{x}_\ell; W_\ell, \mathcal{K}_\ell), \tag{4-3}$$

其中 $\mathbf{x}_\ell$ 为上一层的输入激活。通过同步传输 $(W_\ell, \mathcal{K}_\ell)$，确保迁移后自回归解码可以无缝继续，不产生计算结果的语义变化。



(a) 层级权重迁移　　(b) 注意力级 KV Cache 迁移

图 4-3　BanaServe 的两种迁移粒度。(a) 层级迁移将连续 Transformer 层（含权重 W_ℓ 和 KV Cache $\mathcal{K}_\ell$）从高负载 GPU 迁移至低负载 GPU，实现粗粒度负载均衡；(b) 注意力级迁移沿注意力头维度分割 KV Cache，将 $K^{(2)}, V^{(2)}$ 卸载至冷 GPU 并行计算，仅交换极少量中间结果，实现细粒度负载均衡。

4.3.2　注意力级 KV Cache 迁移

（1）**设计原理**

当需要细粒度负载调整时，BanaServe 沿注意力头维度对 KV Cache 进行分割，将部分头的 KV 状态选择性地迁移至空闲的冷 GPU（Cold GPU）。由于不传输任何模型权重，仅移动中间的键值激活，该方法引入的迁移开销极小。

如图 4-3(b)所示，流程从 QKV 投影层开始，生成查询矩阵 Q、键矩阵 K 和值矩阵 V，随后将注意力头划分为两个不相交子集：

① $K^{(1)}, V^{(1)}$（KV1）：保留在热 GPU（Hot GPU）上进行本地处理；

② $K^{(2)}, V^{(2)}$（KV2）：卸载至冷 GPU 进行远程并行处理。

（2）**并行计算可行性分析**

设 H 为注意力头总数，d 为头维度，$Q \in \mathbb{R}^{B_q \times Hd}$ 为查询矩阵，$K^{(j)}, V^{(j)} \in \mathbb{R}^{B_k \times dh_j}$ 为分配给设备 $j \in \{1, 2\}$ 的键值子集。各子集的注意力分数和值聚合分别计算为：

$$S^{(j)} = Q \cdot (K^{(j)})^{\mathsf{T}}, \tag{4-4}$$

$$A^{(j)} = \exp\left(S^{(j)}\right), \tag{4-5}$$

$$\ell^{(j)} = \begin{cases} \sum_i A_i^{(1)}, & j = 1 \\ \ell^{(1)} + \sum_i A_i^{(2)}, & j = 2 \end{cases}, \tag{4-6}$$

$$O^{(j)} = \frac{A^{(j)}}{\ell^{(j)}} \cdot V^{(j)}, \tag{4-7}$$

最终注意力输出为：

$$O = O^{(1)} + O^{(2)}, \tag{4-8}$$

由于 $K^{(1)}, V^{(1)}$ 和 $K^{(2)}, V^{(2)}$ 来自不相交的头分区，$S^{(1)}$ 和 $S^{(2)}$ 可在两个设备上完全并行计算，无中间数据依赖。跨设备仅需传输 $\ell^{(1)}$（归一化因子）和 $O^{(1)}$，确保全局 softmax 的正确性，同时将传输量控制在极小范围内。

（3）**迁移延迟分析**

设迁移的 KV 数据大小为 S_{kv}，网络带宽为 B_{net}，则注意力级迁移延迟为：

$$T_{\text{attn}} \approx \frac{S_{kv}}{B_{\text{net}}}, \tag{4-9}$$

由于 $S_{kv} \ll S_{\ell}$（层级状态大小），因此 $T_{\text{attn}} \ll T_{\text{layer}}$，注意力级迁移具有轻量、低延迟的特性，特别适合异构解码工作负载的实时负载均衡。

通过将粗粒度层迁移与细粒度注意力迁移相结合，BanaServe 可在运行时自适应选择最优策略，在迁移开销（T_{mig}）与利用率收益（最小化 $\max_g U_g$）之间权衡，同时满足延迟预算约束 $T_{\text{mig}} \leqslant T_{\text{budget}}$。

4.3.3　全局 KV Cache 存储

（1）**设计原理**

前缀缓存感知路由器的核心瓶颈在于前缀缓存存储在 Prefill 节点间的不均衡分布，导致路由器必须同时考虑计算负载和缓存位置。为解决这一问题，BanaServe 引入**全局 KV Cache 存储**（Global KV Cache Store），跨所有 Prefill 和 Decode 节点共享 KV Cache。

如图 4-4所示，全局 KV Cache 存储层由 CPU 内存和 SSD 共同构成存储层次，所有 Prefill 节点均可访问统一的 KV Cache 存储。在此设计下，路由器只需关注计算负载均衡，所有前缀缓存的加载和存取操作均委托给 KV Cache Store 处理，从根本上消除了缓存局部性对路由决策的约束。

（2）**层级流水线重叠传输**

全局 KV Cache 存储面临的主要挑战是频繁加载和存取操作引入的延迟。BanaServe 利用 Transformer 模型逐层执行的特性，设计了三阶段流水线，将预取、计算和加载操作相互重叠，隐藏大部分内存访问延迟。

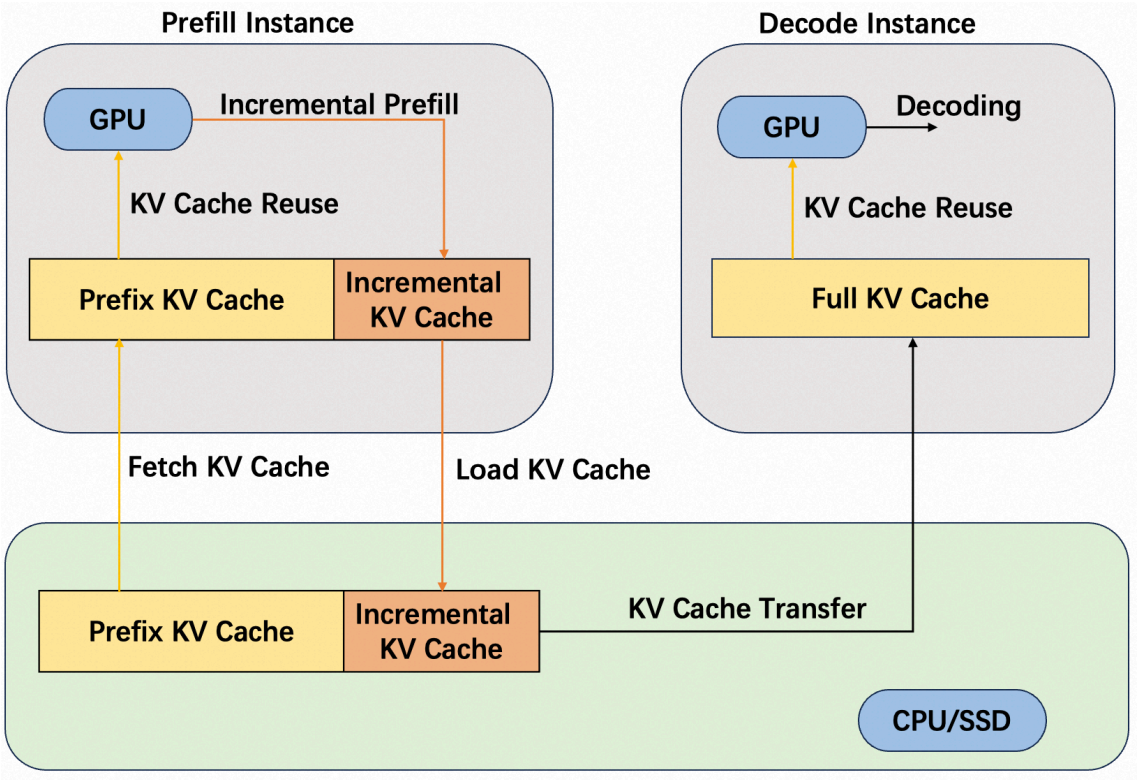


图 4-4　BanaServe 全局 KV Cache 存储的管理与复用机制。Prefill 实例对传入请求执行增量预填充，复用已缓存的前缀 KV Cache；新生成的 KV Cache 存储于共享的 CPU/SSD 支持的 KV 存储中；Decode 实例从共享存储中检索完整 KV Cache 用于 token 生成，避免重复计算。

设 T_F 为 Prefill 阶段的总前向计算时间，r 为平均前缀缓存命中率，L 为输入序列长度（token），N 为 Transformer 层数。每层前向计算时间为：

$$T_{F,\text{layer}} = \frac{T_F \cdot r}{N}, \tag{4-10}$$

设 S_{kv} 为单 token 的每层 KV Cache 大小（字节），B 为有效 PCIe 带宽，每层 KV Cache 传输时间为：

$$T_{KV} = \frac{S_{kv} \cdot L \cdot r}{B}, \tag{4-11}$$

以 LLaMA-3.1-8B 模型为例（隐藏维度 $d_{\text{model}} = 4096$，总注意力头 $h_{\text{total}} = 32$，GQA 下 KV 头 $h_{kv} = 8$，BF16 精度），每头维度为：

$$d_{\text{head}} = \frac{d_{\text{model}}}{h_{\text{total}}} = 128, \tag{4-12}$$

每层 KV Cache 大小（每个 KV 头存储键和值两组激活）为：

$$S_{kv} = h_{kv} \cdot d_{\text{head}} \cdot 2 \cdot 2 \text{ 字节} = 8 \times 128 \times 2 \times 2 = 4096 \text{ 字节（4 KB）}, \tag{4-13}$$

乘以 $N = 32$ 层，每 token 的总 KV Cache 大小为 $S_{kv,\text{total}} = 32 \times 4\ \text{KB} = 128\ \text{KB}$。

在 $L = 1000$ tokens、$r = 0.5$、$B = 200$ Gbps、$T_F = 270$ ms 的实验条件下：

$$T_{F,\text{layer}} = \frac{270\ \text{ms} \times 0.5}{32} \approx 4.22\ \text{ms}, \quad T_{KV} = \frac{4\ \text{KB} \times 1000 \times 0.5}{200\ \text{Gbps}} \approx 0.082\ \text{ms}, \tag{4-14}$$

由于 $T_{KV} \ll T_{F,\text{layer}}$，KV Cache 传输延迟完全可被前向计算所覆盖，实现通信与计算的有效重叠。图 4-5展示了三阶段流水线的执行时序：在 GPU 执行第 L_i 层前向计算的同时，HtoD 通道预取第 L_{i+1} 层的 KV Cache，DtoH 通道存储第 L_{i-1} 层的缓存，三个阶段并行执行，有效隐藏通信延迟。

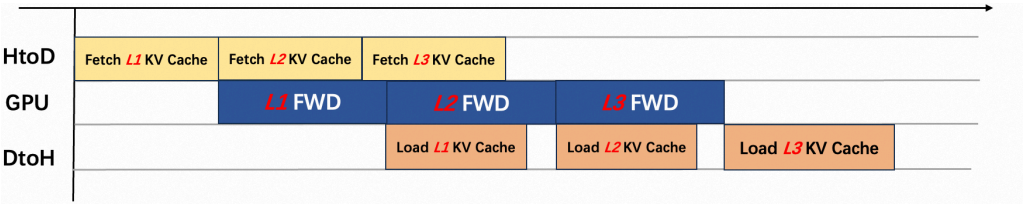


图 4-5　BanaServe 三阶段层级 KV Cache 流水线验证。在 50% 平均前缀缓存命中率下，每层前向计算时间（4.22 ms）远大于每层 KV Cache 传输时间（0.082 ms），确保有效重叠。时序图展示三层（L_1–L_3）的并发执行：GPU 执行 L_i 的前向计算时，HtoD 通道预取 L_{i+1} 的 KV Cache，DtoH 通道存储 L_{i-1} 的缓存。

4.4　性能优化模型

4.4.1　优化目标

BanaServe 将 PD 分离系统的性能优化建模为多目标问题，目标是最大化资源利用率、最小化端到端延迟并最大化吞吐量。

设 $\mathcal{P}$ 和 $\mathcal{D}$ 分别为 Prefill 和 Decode GPU 集合，U_g 为 GPU g 的利用率，T_{TTFT} 为首 token 延迟，T_{TPOT} 为每输出 token 时间，Θ 为系统有效吞吐量（tokens/s）。联合优化目标定义为：

$$\max_{\pi} \ \alpha \cdot U_{\text{avg}}(\pi) - \beta \cdot T_{\text{avg-latency}}(\pi) + \gamma \cdot \Theta(\pi), \tag{4-15}$$

其中：

$$U_{\text{avg}}(\pi) = \frac{1}{|\mathcal{P} \cup \mathcal{D}|} \sum_{g} U_g(\pi), \quad T_{\text{avg-latency}} = \frac{1}{N} \sum_{i=1}^{N} \left(T_{\text{TTFT}}^{(i)} + T_{\text{TPOT}}^{(i)} \right), \tag{4-16}$$

α, β, γ 为利用率、延迟和吞吐量三项的权重系数，π 为当前模块迁移方案。

4.4.2　延迟模型

首 token 延迟（TTFT）建模为：

$$T_{\text{TTFT}} = T_p + T_x + T_q, \tag{4-17}$$

其中 T_p 为 Prefill 计算时间，$T_x = T_{\text{load}} + T_{\text{fetch}}$ 为 KV Cache 传输时间（包含从存储加载时间和从远程 GPU 获取时间），T_q 为 Decode 前的排队延迟。

每输出 token 时间（TPOT）建模为：

$$T_{\text{TPOT}} = T_d + T_c + T_m, \tag{4-18}$$

其中 T_d 为基础解码计算时间，T_c 为缓存访问延迟，T_m 为内存带宽停顿时间。

4.4.3　资源利用率模型

对于具有 n_p 个 Transformer 层的 Prefill 实例，设 M_0 为进程基础内存开销，M_ℓ 为每层内存，K_{init} 为初始 KV Cache 分配，总显存占用为：

$$\text{Mem}_p = M_0 + n_p \cdot M_\ell + K_{\text{init}}. \tag{4-19}$$

设 C_ℓ 为每层每 token 的计算开销，B_{sz} 为批大小，L_{in} 为输入长度，总计算需求为：

$$\text{Comp}_p = n_p \cdot C_\ell \cdot B_{sz} \cdot L_{\text{in}}. \tag{4-20}$$

类似地，对于具有 n_d 个 Transformer 层的 Decode 实例，设 K_{acc} 为解码过程中累积的 KV Cache 大小，L_{gen} 为生成长度：

$$\text{Mem}_d = M_0 + n_d \cdot M_\ell + K_{\text{acc}}, \tag{4-21}$$

$$\text{Comp}_d = n_d \cdot C_\ell \cdot B_{sz} \cdot L_{\text{gen}}. \tag{4-22}$$

给定 GPU 峰值计算能力 C_{gpu}，各阶段的平均利用率为：

$$U_p = \frac{\text{Comp}_p}{C_{\text{gpu}}}, \quad U_d = \frac{\text{Comp}_d}{C_{\text{gpu}}}. \tag{4-23}$$

4.4.4　迁移开销模型

迁移 k 个模块所引入的总开销为：

$$\text{Cost}_{\text{mig}} = k \cdot \left(T_{x\_\text{lat}} + T_{\text{sync}} + T_{\text{mem\_realloc}}\right), \tag{4-24}$$

其中 T_{x_lat} 为 KV Cache 加权重的传输延迟，T_{sync} 为与正在进行的计算同步所需时间，$T_{\text{mem\_realloc}}$ 为缓冲区重分配时间。

模块迁移方案 π 须满足延迟约束：

$$T_{\text{TTFT}}(\pi) \leqslant T_{\text{TTFT}}^{\text{budget}}, \quad T_{\text{TPOT}}(\pi) \leqslant T_{\text{TPOT}}^{\text{budget}}. \tag{4-25}$$

4.4.5　系统吞吐量

给定 N 个并发请求和每请求总输出长度 L_{out}（token），系统吞吐量为：

$$\Theta = \frac{N \cdot L_{\text{out}}}{T_{\text{TTFT}} + L_{\text{out}} \cdot T_{\text{TPOT}}}. \tag{4-26}$$

调度器周期性地求解最优迁移方案：

$$\pi^* = \arg\max_{\pi} \left(\alpha U_{\text{avg}}(\pi) - \beta T_{\text{avg\_latency}}(\pi) + \gamma \Theta(\pi) \right), \tag{4-27}$$

在满足迁移开销和延迟约束的前提下执行。

4.5　核心算法设计

基于上述性能模型，BanaServe 实现两个互补的运行时算法：自适应模块迁移算法（Dynamic Migration Algorithm）和负载感知请求调度算法（Load-aware Request Scheduling Algorithm）。

4.5.1　自适应模块迁移算法

（1）**算法设计**

自适应模块迁移算法通过周期性监控实时负载并执行模块迁移来均衡工作负载，同时限制迁移开销。设 $D = \{d_1, d_2, \cdots, d_{|D|}\}$ 为当前参与服务的 GPU 集合，$\delta > 0$ 为负载不均衡阈值。对每个设备 d，定义归一化利用率为：

$$U_d = \frac{C_d}{C_d^{\max}} + \frac{M_d}{M_d^{\max}}, \tag{4-28}$$

其中 C_d 为当前计算使用量，M_d 为当前显存使用量，$C_d^{\max}$ 和 $M_d^{\max}$ 为对应的硬件容量上限。U_d 的范围为 $[0, 2.0]$，值越大表示综合计算和显存负载越重。

算法 4-1详述了完整的执行流程。

（2）**算法执行分四个阶段：**

阶段一：负载测量（第 1–3 行）

对每个设备 $d \in D$，调用 MeasureUtilization 计算综合计算和显存利用率 $load[d]$，获取各设备当前归一化负载，用于后续不均衡判断。

阶段二：负载分类（第 4–5 行）

通过以下条件构建过载和欠载设备集合：

$$\begin{aligned} overload &= \{d \in D \mid load[d] - \min(load) > \delta\}, \\ underload &= \{d \in D \mid \max(load) - load[d] > \delta\} \end{aligned}, \tag{4-29}$$

其中，过载设备的利用率超过最轻负载设备超过阈值 δ，欠载设备的利用率低于最

算法 4-1　自适应模块迁移算法

Input: 设备集合 D；当前模块分配方案 cfg；负载不均衡阈值 δ
Output: 更新后的模块分配方案 cfg'

// 阶段一：测量当前负载
1 **foreach** $d \in D$ **do**
2 　$load[d] \leftarrow \text{MeasureUtilization}(d)$;　　// 计算 + 显存综合利用率
3 **end**
4 $overload \leftarrow \{d \in D \mid load[d] - \min(load) > \delta\}$;
5 $underload \leftarrow \{d \in D \mid \max(load) - load[d] > \delta\}$;
// 阶段二：迁移决策循环
6 **while** $\exists\, d_o \in overload$ 且 $\exists\, d_u \in underload$ **do**
7 　$\Delta \leftarrow load[d_o] - load[d_u]$;
8 　**if** $\Delta \geqslant \delta$ 且 *SupportsLayerMigration*(d_o) **then**
　　// 粗粒度：层级权重迁移
9 　　MigrateLayer(d_o, d_u);
10 　**else if** $\Delta \geqslant \delta$ 且 *SupportsAttentionMigration*(d_o) **then**
　　// 细粒度：注意力级 KV Cache 迁移
11 　　MigrateKVHeads(d_o, d_u);
12 　更新 $load[d_o]$ 和 $load[d_u]$;
13 　相应更新 cfg';
14 **end**
15 **return** cfg';

重负载设备超过 δ。

阶段三：迁移决策与执行（第 6–13 行）

当存在过载设备 d_o 和欠载设备 d_u 时，计算瞬时负载差：

$$\Delta = load[d_o] - load[d_u]. \tag{4-30}$$

若 $\Delta \geqslant \delta$ 且 d_o 支持层级迁移，调用 MigrateLayer 迁移连续 Transformer 层；若 $\Delta \geqslant \delta$ 且 d_o 支持注意力级迁移，调用 MigrateKVHeads 迁移选定 KV Cache 段。执行前评估迁移的收益-开销比：

$$\text{Benefit}(m) = \Delta_{\text{before}} - \Delta_{\text{after}}, \quad \text{Cost}(m) = T_{\text{transfer}}(m) + T_{\text{sync}}(m), \tag{4-31}$$

仅当 $\text{Benefit}(m)/\text{Cost}(m) \geqslant \rho$（可配置效率比）时才执行迁移。

阶段四：更新分配方案（第 14–16 行）

每次迁移后更新 $load[d_o]$ 和 $load[d_u]$，修订 cfg' 以反映新的模块放置，重复直至不再存在同时满足过载和欠载条件的设备对，最终返回更新后的 cfg'。

（3）**复杂度与收敛性分析**

每个控制周期的时间复杂度为：

$$O(|D| + N_m). \tag{4-32}$$

其中 $|D|$ 来自负载测量和分类，N_m 来自模块选择和开销评估。在典型部署中 $|D|$ 为数十量级，N_m 受模型深度约束，支持实时运行。为防止振荡，采用滞后控制，设置不同的上升阈值 $\delta_{\uparrow}$ 和下降阈值 $\delta_{\downarrow}$。全局 KV Cache 存储的集成确保迁移过程中所有 KV 状态得以保存，避免重新计算并维持稳定吞吐量。

4.5.2　负载感知请求调度算法

（1）**算法设计**

负载感知请求调度算法（算法 4-2）将传入请求高效分配给 Prefill 实例。与缓存感知路由策略不同，由于全局 KV Cache 存储支持所有 Prefill GPU 之间的 KV Cache 共享，该调度器无需考虑前缀缓存命中率，调度决策完全基于实时负载指标。

对每个 Prefill 实例 p_i，归一化负载定义为：

$$U_{p_i} = \frac{C_{p_i}}{C_{p_i}^{\max}} + \frac{M_{p_i}}{M_{p_i}^{\max}}, \tag{4-33}$$

其中 C_{p_i} 和 M_{p_i} 为当前计算和显存使用量，$C_{p_i}^{\max}$ 和 $M_{p_i}^{\max}$ 为对应容量上限。

（2）**算法执行分四个阶段：**

阶段一：负载测量（第 1–4 行）

对每个 $p_i \in P$，调用 MeasureUtilization 测量 U_{p_i}，并通过 GetQueueLength 记录队列长度 $q\_len[p_i]$。

阶段二：排序（第 5 行）

按负载和队列长度升序排列所有实例，生成用于调度决策的候选列表。

阶段三：调度循环（第 6–13 行）

对每个请求 $req \in Q$，从候选列表中选择负载最低的实例 p_{target}。若 $U_{p_{target}} < \delta_L$，将 req 分配给 p_{target} 并将该请求的估计负载贡献加至 $load[p_{target}]$；否则，选择队列长度最小的实例并分配。

阶段四：生成调度方案（第 14–15 行）

调用 GetDispatchMap 生成调度映射 D^* 并返回至编排组件执行。

（3）**复杂度分析**

每个调度周期的时间复杂度为：

$$O(|P| \log |P| + |Q|), \tag{4-34}$$

其中 $O(|P| \log |P|)$ 来自按当前负载和队列长度对实例排序，$O(|Q|)$ 来自逐请求分配循环。在典型部署中 $|P|$ 为数十量级（数十个 GPU），算法适合实时调度。

算法 4-2　负载感知请求调度算法

Input: 请求队列 Q；Prefill 实例集合 P；负载阈值 δ_L
Output: 调度方案 D^*

// 阶段一：测量每个 Prefill 实例的当前负载
foreach $p_i \in P$ **do**
    $load[p_i] \leftarrow$ MeasureUtilization(p_i) ;　　// 计算 + 显存
    $q\_len[p_i] \leftarrow$ GetQueueLength(p_i);
end
// 阶段二：按负载和队列长度排序实例
$candidates \leftarrow$ Sort$(P,\ key = (load, q_len),\ order =$ ascending$)$;
// 阶段三：将请求分配至负载最低的实例
foreach $req \in Q$ **do**
    $p_{target} \leftarrow$ Select$(candidates,\ policy =$ least-loaded$)$;
    if $load[p_{target}] < \delta_L$ **then**
        AssignRequest(req, p_{target});
        $load[p_{target}] \leftarrow load[p_{target}] +$ EstimateLoad(req);
    else
        $p_{target} \leftarrow$ Select$(candidates,\ policy =$ lowest-queue$)$;
        AssignRequest(req, p_{target});
    end
end
// 阶段四：返回最终调度方案
$D^* \leftarrow$ GetDispatchMap(P);
return D^*;

通过全局 KV Cache 存储将缓存命中率从路由标准中剥离，该调度器专注于均衡负载和最小化排队延迟，有效降低了调度复杂度，避免 Prefill 节点热点形成，并在动态工作负载下确保稳定吞吐量。

4.6　实验评估

4.6.1　实验设置

评估模型：选用两种具有代表性的 13B 参数级大语言模型，如表 4-1所示。LLaMA-13B 作为 LLaMA 系列的主要评估目标，用于评估 BanaServe 处理大规模纯解码器 Transformer 的能力；OPT-13B 采用不同的架构优化和训练方法，用于跨架构的泛化性验证。

基准测试与工作负载：实验设计采用两个互补的公开基准测试：

（1）**Alpaca**[43]（短文本场景）：包含 52,000 条指令跟随样例，输入序列长度集中在 4–50 token，代表聊天机器人、代码助手和问答系统等交互式应用的典型生

表 4-1　实验评估所用模型配置

模型	参数量	架构	评估目的
LLaMA-13B	13B	纯解码器	系列内性能评估
OPT-13B	13B	纯解码器	跨架构泛化验证

产推理工作负载；

（2）**LongBench**[2]（长文本场景）：覆盖 21 项多样化任务，上下文长度从约 2,000 token 跨越至超过 85,000 token，涵盖多文档问答、文档摘要、少样本学习和代码补全等场景，对 KV Cache 管理和显存优化策略形成全面压力测试。

所有实验中最大输出长度上限设为 512 token，以确保跨基准测试的一致性并聚焦于输入长度差异对系统性能的影响。

负载测试方法：请求速率从 1 到 20 RPS（每秒请求数）系统性地递增，采用泊松到达过程模拟真实流量的突发特性。每组实验配置包含 60 秒预热期以确保系统稳定并填充缓存，所有实验重复 5 次（不同随机种子），报告均值及 95% 置信区间。

评估指标：采用三项核心指标：吞吐量（requests/s）、总处理时间（s）和平均延迟（ms），分别衡量系统原始处理能力、端到端效率和用户感知响应性。

对比基线：与两个当前最优 LLM 推理系统进行对比：

（1）**vLLM**[5]：学术界和工业界广泛采用的推理系统，基于连续批处理和 PagedAttention 实现高效 KV Cache 管理，但单体架构难以平衡 Prefill 和 Decode 阶段的竞争需求；

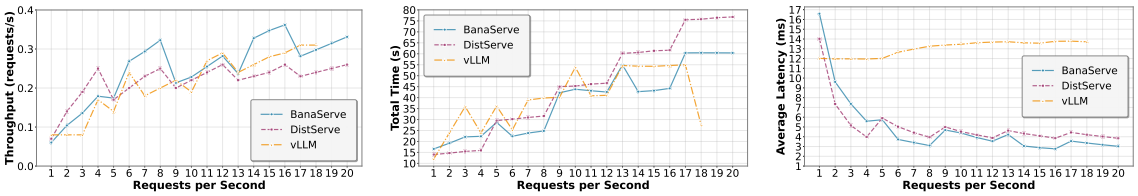
（2）**DistServe**[1]：将 Prefill 和 Decode 计算分配至专用实例的分离式推理系统，虽缓解了阶段间干扰，但引入了额外的通信开销和组件协调需求。

4.6.2　短文本场景性能评估

图 4-6和图 4-7分别展示了 LLaMA-13B 和 OPT-13B 在 Alpaca 短文本基准测试下的性能对比结果。

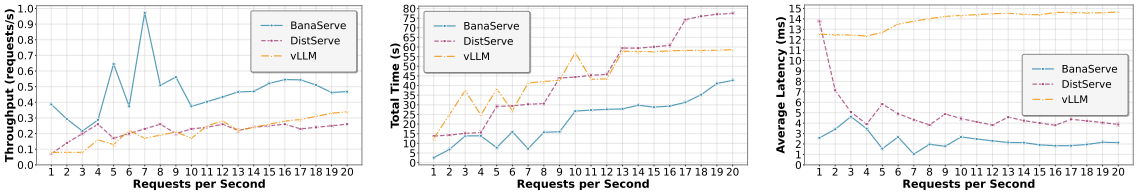
LLaMA-13B 短文本分析：在整个 RPS 范围内，BanaServe 持续提供更高吞吐量（相比 DistServe 和 vLLM 提升 1.1 倍至 1.2 倍），同时保持更低延迟和更短总处理时间。这些改进源于 BanaServe 最小化的调度开销和高效的 KV Cache 处理机制，支持快速上下文切换而不引发显著的流水线停顿。

OPT-13B 短文本分析：在 OPT-13B 上，性能提升更为显著。BanaServe 吞吐量相比 DistServe 提升 2.8 倍至 3.9 倍，相比 vLLM 最高提升 3.9 倍；平均延迟相



(a) 吞吐量　　(b) 总处理时间　　(c) 平均延迟

图 4-6　LLaMA-13B 短文本场景性能结果。在 1 到 20 RPS 的全请求速率范围内，BanaServe 在吞吐量、总处理时间和平均延迟三项指标上均优于 DistServe 和 vLLM，吞吐量提升 1.1 倍至 1.2 倍，总处理时间显著降低。



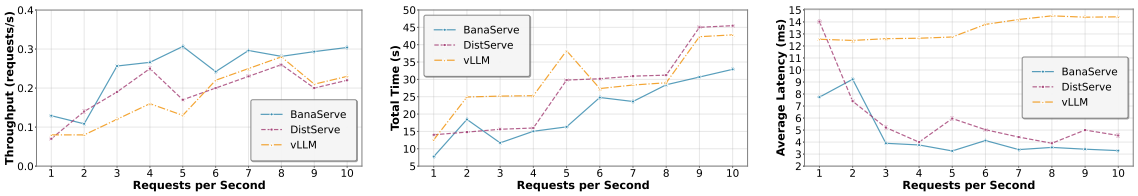
(a) 吞吐量　　(b) 总处理时间　　(c) 平均延迟

图 4-7　OPT-13B 短文本场景性能结果。BanaServe 相比 DistServe 吞吐量提升 2.8 倍至 3.9 倍，相比 vLLM 最高提升 3.9 倍；平均延迟相比 vLLM 降低 3.9% 至 78.4%，相比 DistServe 降低 1.4% 至 70.1%。

比 vLLM 降低 3.9% 至 78.4%，相比 DistServe 降低 1.4% 至 70.1%。这些结果凸显了 BanaServe 在高频上下文切换场景下的效率优势，其批处理和缓存复用策略使 GPU 利用率接近饱和同时保持响应性。

4.6.3　长文本场景性能评估

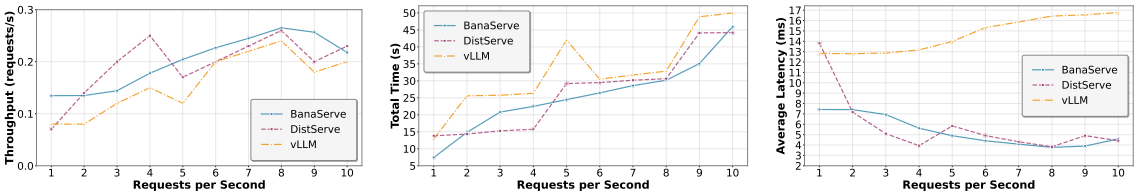
图 4-8和图 4-9分别展示了 LLaMA-13B 和 OPT-13B 在 LongBench 长文本基准测试下的性能对比结果。



(a) 吞吐量　　(b) 总处理时间　　(c) 平均延迟

图 4-8　LLaMA-13B 长文本场景性能结果。BanaServe 在 1 到 10 RPS 范围内吞吐量提升 1.3 倍至 1.5 倍（相比 DistServe 和 vLLM），延迟降低 1.4% 至 65.3%（相比 vLLM）。在高负载（10 RPS 以上）场景下优势更为明显。

LLaMA-13B 长文本分析：在 LongBench 扩展序列基准测试上，BanaServe 相比 DistServe 和 vLLM 吞吐量提升 1.3 倍至 1.5 倍，延迟降低 20.6% 至 65.3%（相比 vLLM）和 1.4% 至 20.6%（相比 DistServe）。性能差距在高负载条件（10–20 RPS）下更为明显，此时传统系统受缓存争用和流水线阻塞影响，而 BanaServe 的统一架



(a) 吞吐量　　(b) 总处理时间　　(c) 平均延迟

图 4-9　OPT-13B 长文本场景性能结果。BanaServe 相比 DistServe 和 vLLM 吞吐量提升 1.1 倍至 1.3 倍，延迟改善模式与 LLaMA-13B 一致，在不同负载水平下均保持稳定的性能优势。

构减少了组件间通信开销，并通过动态批处理平衡各阶段的计算和显存资源。

OPT-13B 长文本分析：在相同 LongBench 设置下，BanaServe 相比 DistServe 和 vLLM 吞吐量提升 1.1 倍至 1.3 倍。虽然增益幅度小于 LLaMA-13B，但在不同负载水平下保持一致，可归因于更低的 KV Cache 获取延迟和更高效的显存分配模式。延迟改善模式与 LLaMA-13B 一致，即使处理大规模上下文窗口时也能确保稳定的服务质量。

可扩展性分析：随着请求速率从 1 RPS 增至 20 RPS，BanaServe 在所有配置下均保持一致的性能优势。轻载（1–5 RPS）时性能差距主要源于 BanaServe 高效的调度和降低的系统开销；高负载（10–20 RPS）时，BanaServe 优越的批处理策略和优化的显存管理使其维持更高吞吐量，而其他系统因资源争用和缓存利用率低下而出现性能下降。

4.6.4　Azure 生产负载追踪实验

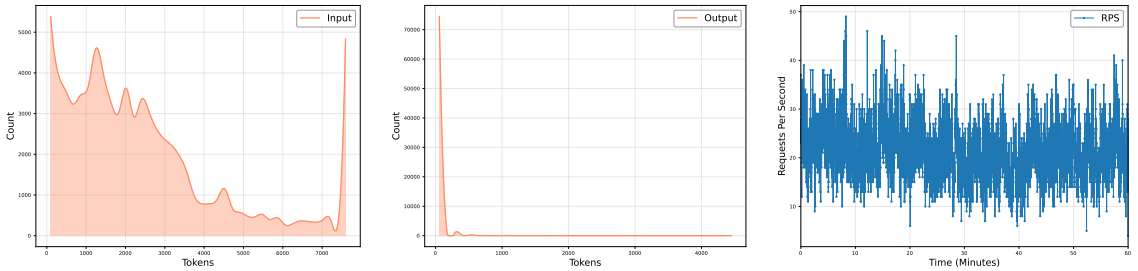
为补充合成基准测试，本实验使用 Azure 公开发布的 LLM 推理生产负载追踪数据[58]对 BanaServe 进行评估。该追踪数据包含匿名化的请求级日志，涵盖时间戳、输入/输出 token 长度和多模态请求类型，工作负载呈现出显著的异构性和突发性，反映了生产环境中真实的 LLM 服务条件。

图 4-10展示了 Azure 生产负载追踪数据前 1 小时的统计特性：输入和输出长度跨越广泛范围，RPS 分布呈现明显的突发性，与生产环境中非平稳到达模式一致。

图 4-11展示了在 LLaMA-13B 和 OPT-13B 两个模型上，BanaServe 与 vLLM 和 DistServe 的吞吐量和延迟对比结果。

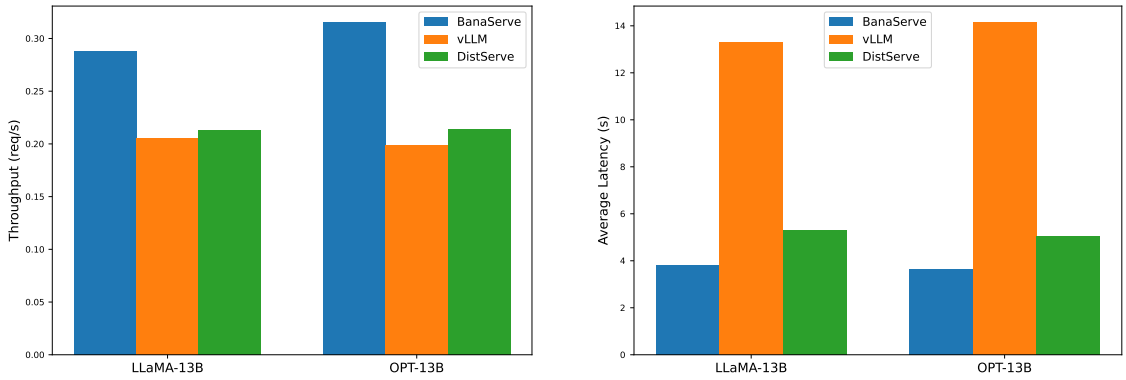
Azure 生产负载实验结果分析：在 LLaMA-13B 和 OPT-13B 两个模型上，BanaServe 均持续提供更高吞吐量和更低延迟，在 Azure 工作负载特有的重尾输入/输出分布和突发请求到达模式下性能提升尤为显著。具体而言：

（1）**LLaMA-13B：**BanaServe 吞吐量相比 vLLM 提升 40.4%，相比 DistServe



(a) 输入长度分布　　(b) 输出长度分布　　(c) 请求到达速率

图 4-10　Azure 生产负载追踪数据前 1 小时的统计特性。输入和输出长度分布呈重尾特性，请求到达速率具有显著突发性，反映了真实生产环境中多样化 LLM 服务工作负载的特征。



(a) 吞吐量对比　　(b) 平均延迟对比

图 4-11　Azure 生产负载追踪实验的吞吐量和延迟综合对比。BanaServe 在 LLaMA-13B 上取得最高吞吐量（0.288 req/s）和最低延迟（3.828 s），在 OPT-13B 上分别为 0.315 req/s 和 3.658 s，在重尾分布和突发流量下均显著优于 vLLM 和 DistServe。

提升 35.2%；平均延迟相比 vLLM 降低 71.2%，相比 DistServe 降低 27.6%；

（2）**OPT-13B**：性能提升更为显著，吞吐量相比 vLLM 提升 58.3%，相比 DistServe 提升 47.2%；延迟相比 vLLM 降低 74.1%，相比 DistServe 降低 27.4%。

这些改进突出了 BanaServe 在重尾输入/输出分布和突发流量模式下维持高效率的能力。两个模型架构上一致的增益表明，系统设计能够有效泛化至不同 Transformer 实现，在生产风格工作负载下均表现出色。

4.6.5　综合性能对比分析

实验结果验证了 BanaServe 设计的三个核心判断：

（1）**动态资源迁移对 PD 分离架构至关重要**：通过层级和注意力级两种粒度的动态迁移机制，BanaServe 能够实时响应工作负载的不均衡性，在 Prefill 计算密集阶段和 Decode 显存密集阶段之间自适应再分配资源，从根本上消除了静态配置下的资源利用率失衡问题；

（2）**全局 KV Cache 存储是解耦路由与缓存的关键**：传统前缀缓存感知路由将调度决策与 KV Cache 物理位置紧耦合，导致持续性的负载倾斜。全局 KV Cache 存储通过将所有 Prefill 实例的缓存统一管理，配合层级流水线重叠传输（$T_{KV} \approx 0.082\ \text{ms} \ll T_{F,\text{layer}} \approx 4.22\ \text{ms}$），使路由器能够完全基于实时负载做出调度决策，彻底消除了缓存感知路由引发的热点问题；

（3）**粗细粒度迁移的协同设计兼顾效果与开销**：层级迁移适合处理严重的工作负载不均衡（迁移大量计算和显存资源），注意力级迁移则提供轻量级的细粒度调整（无需传输模型权重），两种机制的协同使用使 BanaServe 能够根据实际负载差距（Δ）和迁移效率比（ρ）自适应选择最优策略。

4.7　本章小结

本章提出了 BanaServe，一个面向 PD 分离架构的动态资源协同优化框架。针对现有 PD 分离系统中静态资源配置、内生资源不均衡和前缀缓存感知路由引发负载倾斜三类核心局限，BanaServe 通过三项关键创新实现了资源分配与状态管理的解耦：

层级权重迁移将 Transformer 层的权重和 KV Cache 作为整体在 GPU 间动态迁移，实现粗粒度的计算-显存资源再平衡。迁移延迟由权重传输主导（$T_{\text{layer}} \approx S_{\ell}^{\text{total}}/B_{\text{net}} + T_{\text{sync}}$），在高带宽数据中心互联下保持实用性，适合处理严重的工作负载不均衡场景。

注意力级 KV Cache 迁移沿注意力头维度分割 KV Cache，仅传输 KV 状态（无模型权重），迁移延迟极低（$T_{\text{attn}} \approx S_{kv}/B_{\text{net}} \ll T_{\text{layer}}$），支持在不引发显著服务中断的前提下进行实时细粒度负载均衡，与层级迁移协同构成粗细粒度自适应迁移体系。

全局 KV Cache 存储构建跨所有 Prefill 实例的统一 KV Cache 共享层，通过三阶段层级流水线（预取-计算-存储并行）隐藏通信延迟，实现通信与计算的有效重叠（$T_{KV} \ll T_{F,\text{layer}}$）。路由器因此摆脱缓存位置约束，完全基于实时负载进行纯负载感知调度，从根本上消除前缀缓存感知路由引发的热点问题。

自适应模块迁移算法周期性监控归一化综合利用率（$U_d = C_d/C_d^{\max} + M_d/M_d^{\max}$），根据过载-欠载阈值 δ 和收益-开销比 ρ 自适应选择迁移策略，时间复杂度为 $O(|D| + N_m)$，支持实时运行。

负载感知请求调度算法基于实时负载和队列长度对 Prefill 实例进行排序和调度，时间复杂度为 $O(|P|\log|P| + |Q|)$，彻底摆脱了缓存局部性对调度决策的约束。

实验结果表明，相比 vLLM，BanaServe 在合成基准（Alpaca 和 LongBench）上

吞吐量提升 1.2 倍至 3.9 倍，总处理时间降低 3.9% 至 78.4%；相比 DistServe，吞吐量提升 1.1 倍至 2.8 倍，延迟降低 1.4% 至 70.1%。在 Azure 生产负载追踪实验中，BanaServe 在重尾分布和突发流量下的鲁棒性得到进一步验证，LLaMA-13B 和 OPT-13B 均实现了最高吞吐量和最低延迟。

然而，BanaServe 目前仍存在若干局限：调度和迁移策略对异构硬件（混合精度加速器和专用通信互联）的优化支持不足；工作负载管理以被动响应为主，缺乏基于历史模式的预测性调度；系统设计局限于单区域集群，尚未支持多区域分布式部署场景。这些问题将作为未来工作的重要方向加以探索，包括硬件感知调度、基于强化学习的预测性编排，以及面向广域网优化的跨地域 KV Cache 同步机制。

第 5 章　总结与展望

5.1　研究总结

随着大语言模型在各行业的深度渗透，高效、低延迟、高吞吐量的 LLM 推理服务已成为 AI 基础设施建设的核心挑战。本文围绕 LLM 推理服务中资源管理的两类核心瓶颈展开研究：（1）在静态场景下，批处理策略的粗放性和部署配置的低效性共同导致了严重的 SLO 违约和资源浪费；（2）在动态场景下，PD 分离架构中 Prefill 与 Decode 阶段之间的内生资源失衡，以及前缀缓存感知路由引发的持续性负载倾斜，使得现有系统难以在高动态工作负载下维持高效服务。

针对上述两类挑战，本文分别提出了 UELLM（第三章）和 BanaServe（第四章）两个互补的优化框架，从批处理优化和层级资源协同两个层面构建了覆盖” 静态优化-动态调度” 全生命周期的 LLM 推理服务资源管理体系。两个系统的核心设计理念、适用场景和性能定位的对比如表 5-1所示。

表 5-1　本文主要研究成果定位对比

维度	UELLM（第三章）	BanaServe（第四章）
核心问题	批处理策略粗放；部署配置搜索低效；SLO 违约率高	PD 资源失衡；缓存感知路由热点；静态配置无法适应动态负载
设计理念	批处理调度与部署配置联合优化	资源分配与状态管理解耦
关键创新	输出长度预测；SLO-ODBS 算法；HELR 拓扑感知部署	层级权重迁移；注意力级 KV Cache 迁移；全局 KV Cache 存储
适用场景	单实例或小规模异构集群；弱动态负载	大规模 PD 分离集群；高动态突发负载

UELLM 的核心贡献在于验证了**批处理调度优化与部署配置优化的联合设计**是提升 LLM 推理服务性能的关键路径，两者缺一不可：单独的批处理优化（UB）无法改善 GPU 利用率，单独的部署优化（UD）无法消除 SLO 违约，只有联合优化（UA）才能同时实现零 SLO 违约、最低延迟和最高吞吐量。

BanaServe 的核心贡献在于验证了**细粒度动态资源迁移与全局缓存共享的协同设计**可以有效解耦 PD 分离系统中的资源分配与状态管理，从根本上消除前缀缓存感知路由引发的热点问题，并在真实生产负载（Azure 追踪数据）下保持强健的性能优势。

然而，两个系统仍存在若干值得正视的局限性：UELLM 的输出长度预测泛化

能力受限于微调数据集分布，HELR 算法的状态空间复杂度限制了其在大规模集群上的适用性；BanaServe 的迁移机制对异构硬件适配不足，被动响应式调度在极端突发场景下存在不可避免的响应延迟窗口，且系统设计局限于单区域集群，尚未支持跨地域分布式部署场景。

5.2　未来研究方向

基于上述局限性分析，结合 LLM 推理服务领域的技术发展趋势，本文展望以下四个具有重要研究价值的未来方向。

5.2.1　异构硬件感知的自适应调度

当前 UELLM 和 BanaServe 的调度决策均基于简化的硬件抽象模型，未能充分刻画异构硬件的细粒度特性。未来研究应构建细粒度硬件感知调度框架，具体包括以下三个子方向：

（1）**设备画像数据库**：在线收集并维护各类 GPU（A100、V100、RTX 3090、H100 等）的计算吞吐量、显存带宽、NVLink/PCIe 互联拓扑和混合精度支持能力等细粒度硬件参数，构建可动态更新的设备画像数据库；

（2）**拓扑感知迁移路径规划**：在迁移决策时综合考虑源设备与目标设备之间的实际互联带宽和延迟，选择传输开销最小的迁移路径，将 HELR 的拓扑感知能力从部署阶段延伸至运行时动态迁移；

（3）**混合精度自适应执行**：针对 FP16、BF16、INT8 和 INT4 等不同量化精度格式，设计自适应的层分配策略，将计算敏感层保留在高精度设备上，将存储密集层迁移至低精度但带宽更高的设备，在推理质量和资源效率之间寻求最优权衡。

5.2.2　基于预测的主动式资源编排

现有 BanaServe 的被动响应式调度在极端突发场景下存在响应延迟窗口。未来研究应引入预测性编排（Predictive Orchestration）机制，将历史负载模式与实时监控相结合，实现资源的提前预分配：

（1）**工作负载预测模型**：利用历史请求到达序列训练轻量级时序预测模型，预测未来时间窗口内的请求速率和计算需求，为迁移决策提供前瞻性依据；

（2）**基于强化学习的迁移策略优化**：将迁移决策建模为马尔可夫决策过程（MDP），以系统吞吐量、SLO 不违约率和 GPU 利用率的加权组合作为奖励函数，通过在线强化学习训练迁移策略；

（3）**突发流量预警与弹性扩缩容**：结合容器编排平台（Kubernetes）的弹性

扩缩容能力，在预测到流量突增时提前完成新实例的模型加载和 KV Cache 预热，实现对突发流量的零停机响应。

5.2.3　端到端延迟优化与 SLO 感知服务

当前两个框架在端到端延迟优化方面均存在进一步提升空间。未来研究应构建**端到端 SLO 感知服务框架**：

（1）**分层 SLO 分解**：将端到端 SLO 分解为各子阶段的局部 SLO 约束（$T_{\text{TTFT}}^{\text{budget}}$ 和 $T_{\text{TPOT}}^{\text{budget}}$），并根据当前系统负载动态调整各阶段的预算分配；

（2）**请求优先级感知调度**：为不同类型请求赋予不同的调度优先级和 SLO 约束，在高负载时通过抢占式调度动态调整批次构成，优先保障高优先级请求的 SLO；

（3）**尾延迟优化**：引入尾延迟感知的批次构建策略和推测执行（Speculative Execution）机制，将 P99/P999 尾延迟纳入优化目标体系。

5.2.4　跨地域分布式推理

随着 LLM 服务全球化部署需求的增长，单数据中心架构已无法满足地理分散用户的低延迟访问需求。未来研究应探索跨地域分布式 LLM 推理系统的设计：

（1）**广域网感知的 KV Cache 同步**：设计层次化 KV Cache 存储架构，数据中心内采用 BanaServe 的三阶段流水线实现低开销本地访问，跨数据中心采用选择性 KV Cache 同步（仅同步高频访问的热门前缀），结合增量差分压缩技术降低跨 WAN 传输量；

（2）**地理感知请求路由**：综合考虑用户地理位置、各区域数据中心负载状态和 WAN 延迟，设计全局请求路由策略，在用户访问延迟和数据中心资源利用率之间寻求最优权衡；

（3）**模型分片与流水线并行**：将模型的不同层组分布在不同地理区域的数据中心，设计 WAN 感知的流水线调度策略，合理安排各区域处理的层范围以最小化跨 WAN 激活传输量和流水线气泡。

结 论

本文围绕大语言模型推理服务中的资源管理核心挑战，从批处理调度与部署配置联合优化、PD 分离架构下的动态资源协同两个互补维度展开研究，提出了 UELLM 和 BanaServe 两个系统框架，取得了以下主要研究成果：

（1）揭示了 LLM 推理服务中批处理策略与部署配置的多层次瓶颈，并建立了联合优化框架 UELLM。针对现有 LLM 推理系统中 SLO 感知缺失、输出长度预测精度不足和部署配置搜索低效等问题，提出了由资源画像器、批处理调度器和 LLM 部署器三个组件协同构成的 UELLM 框架。其中，基于 LoRA 微调 ChatGLM3-6B 的输出长度预测方案在分桶准确率上达到 99.51%，显著优于基于轻量级分类器的现有方案；提出的 SLO-ODBS 算法通过综合考虑 SLO 约束和输出长度差异，实现了低延迟与低 SLO 违约率的双重优化目标；提出的 HELR 算法将 LLM 部署建模为拓扑感知动态规划问题，无需元学习压力测试即可自动求解最优设备映射。在 4 块 NVIDIA RTX 3090 GPU 构成的异构集群上，UELLM 相比 S^3 和 Morphling 将推理延迟降低 72.3% 至 90.3%，GPU 利用率提升 1.2 倍至 4.1 倍，吞吐量提高 1.92 倍至 4.98 倍，并实现零 SLO 违约，验证了批处理调度与部署配置联合优化的有效性。

（2）建立了 PD 分离系统的多目标性能优化理论模型，定量刻画了计算-显存资源失衡的内在机理。针对 PD 分离架构中 Prefill 实例（计算利用率约 95%，显存利用率约 35%）与 Decode 实例（计算利用率约 35%，显存利用率约 90%）的天然资源互补性，建立了涵盖首 token 延迟（TTFT）、每输出 token 时间（TPOT）、跨实例资源利用率和模块迁移开销的多目标联合优化模型。通过对层级 KV Cache 传输时间（$T_{KV} \approx 0.082$ ms）与每层前向计算时间（$T_{F,\text{layer}} \approx 4.22$ ms）的定量分析，从理论上证明了三阶段层级流水线重叠传输的可行性，为全局 KV Cache 存储的设计提供了严格的理论依据。

（3）提出了面向 PD 分离架构的动态资源协同优化框架 BanaServe，设计了粗细粒度自适应迁移机制。针对现有 PD 分离系统中静态资源配置无法适应动态负载、前缀缓存感知路由引发持续性负载倾斜的问题，提出了 BanaServe 框架，通过三项核心机制实现资源分配与状态管理的解耦：层级权重迁移在 Transformer 层粒度上同步迁移权重和 KV Cache（迁移延迟 $T_{\text{layer}} \approx S_{\ell}^{\text{total}}/B_{\text{net}} + T_{\text{sync}}$），实现粗粒度跨实例资源再平衡；注意力级 KV Cache 迁移沿注意力头维度分割 KV Cache，仅

传输 KV 状态（$T_{\text{attn}} \approx S_{kv}/B_{\text{net}} \ll T_{\text{layer}}$），实现细粒度负载均衡且无服务中断；全局 KV Cache 存储配合三阶段层级流水线重叠传输，使路由器摆脱缓存位置约束，实现纯负载感知调度。在合成基准（Alpaca 和 LongBench）评估中，相比 vLLM 吞吐量提升 1.2 倍至 3.9 倍，总处理时间降低 3.9% 至 78.4%；相比 DistServe 吞吐量提升 1.1 倍至 2.8 倍，延迟降低 1.4% 至 70.1%。

（4）在 Azure 生产负载追踪数据上验证了 BanaServe 在真实生产条件下的鲁棒性与泛化能力。采用 Azure 公开发布的 LLM 推理生产负载追踪数据（包含重尾输入/输出长度分布和高突发性到达模式），在 LLaMA-13B 和 OPT-13B 两个模型架构上进行了系统性验证。实验结果表明，BanaServe 在 LLaMA-13B 上相比 vLLM 吞吐量提升 40.4%、延迟降低 71.2%，在 OPT-13B 上相比 vLLM 吞吐量提升 58.3%、延迟降低 74.1%，在两种模型架构上均实现最高吞吐量和最低延迟，表明所提方法对不同 Transformer 实现具有良好的泛化能力。

在研究展望方面，本文的工作尚存在若干值得进一步深入探索的方向。（1）异构硬件感知调度方面，现有迁移延迟模型对混合精度加速器和异构互联拓扑的建模较为简化，未来可构建细粒度设备画像数据库并设计拓扑感知的迁移路径规划算法，进一步释放异构硬件集群的潜在性能。（2）预测性主动编排方面，现有 BanaServe 的被动响应式调度在极端突发场景下存在不可避免的响应延迟窗口，可引入基于历史负载模式的时序预测模型和强化学习驱动的迁移策略优化，将被动响应转变为主动预分配。（3）跨地域分布式推理方面，现有设计局限于单数据中心集群，在 WAN 高延迟（10–100 ms）环境下三阶段流水线重叠传输的有效性条件（$T_{KV} \ll T_{F,\text{layer}}$）可能不再成立，需要设计广域网感知的层次化 KV Cache 同步架构和地理感知请求路由策略。（4）多模型协同推理方面，随着 RAG、工具调用 Agent 等多模型协同范式的普及，如何在同一集群上高效调度异构模型实例并实现跨模型 KV Cache 共享，是值得深入探索的重要研究方向。

参考文献

[1] ZHONG Y, LIU S, CHEN J, et al. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving[C]//Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2024: 193-210.

[2] BAI Y, LV X, ZHANG J, et al. Longbench: A bilingual, multitask benchmark for long context understanding[A]. 2023.

[3] JIN C, ZHANG Z, JIANG X, et al. S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput[A]. 2023.

[4] ZHENG L, CHIANG W L, SHENG Y, et al. SGLang: Efficient Execution of Structured Language Model Programs[C]//Advances in Neural Information Processing Systems (NeurIPS): Vol. 37. 2024.

[5] KWON W, LI Z, ZHUANG S, et al. Efficient Memory Management for Large Language Model Serving with PagedAttention[C]//Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP). 2023: 611-626.

[6] ACHIAM J, ADLER S, AGARWAL S, et al. GPT-4 Technical Report[A]. 2023.

[7] TOUVRON H, LAVRIL T, IZACARD G, et al. LLaMA: Open and Efficient Foundation Language Models[A]. 2023.

[8] Anthropic. Claude[EB/OL]. 2025. https://claude.ai.

[9] WU C J, RAGHAVENDRA R, GUPTA U, et al. Sustainable AI: Environmental Implications, Challenges and Opportunities[J]. Proceedings of Machine Learning and Systems, 2022, 4: 795-813.

[10] HE Y, XU M, WU J, et al. Service-Oriented Computing: 22nd International Conference, ICSOC 2024, Tunis, Tunisia, December 3–6, 2024, Proceedings, Part I: UELLM: A Unified and Efficient Approach for Large Language Model Inference Serving[M/OL]. Berlin, Heidelberg: Springer-Verlag, 2024: 218-235. https://doi.org/10.1007/978-981-96-0805-8_16.

[11] WOLF T, DEBUT L, SANH V, et al. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations: Transformers: State-of-the-Art Natural Language Processing[M/OL]. Online: Association for Computational Linguistics, 2020: 38-45. https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[12] NGUYEN-DUC A, CABRERO-DANIEL B, PRZYBYLEK A, et al. Generative artificial intelligence for software engineering—A research agenda[J]. Software: Practice and Experience, 2025.

[13] FENG J, HUANG Y, ZHANG R, et al. Proceedings of the 52nd Annual International Symposium on Computer Architecture: WindServe: Efficient Phase-Disaggregated LLM Serving with Stream-based Dynamic Scheduling[M/OL]. New York, NY, USA: Association for Computing Machinery, 2025: 1283-1295. https://doi.org/10.1145/3695053.3730999.

[14] ZHOU Z, NING X, HONG K, et al. A survey on efficient inference for large language models [A]. 2024.

[15] NVIDIA Corporation. TensorRT-LLM: High-performance LLM inference library from NVIDIA[EB/OL]. https://developer.nvidia.com/tensorrt-llm.

[16] LEVIATHAN Y, KALMAN M, MATIAS Y. Fast Inference from Transformers via Speculative Decoding[J]. International Conference on Machine Learning (ICML), 2023: 19274-19286.

[17] CHEN C, BORGEAUD S, IRVING G, et al. Accelerating Large Language Model Decoding with Speculative Decoding[J]. International Conference on Machine Learning (ICML), 2023: 6159-6181.

[18] YU G I, JEONG J S, KIM G W, et al. Orca: A Distributed Serving System for Transformer-Based Generative Models[C/OL]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association, 2022: 521-538. https://www.usenix.org/conference/osdi22/presentation/yu.

[19] SCHUSTER T, FISCH A, GUPTA J, et al. Confident Adaptive Language Modeling[J]. Advances in Neural Information Processing Systems (NeurIPS), 2022, 35: 17456-17472.

[20] LI Z, LI S, ZHANG M, et al. CascadeBERT: Accelerating Inference of Pre-trained Language Models via Cascade Breaking[C]//Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL). 2020: 3876-3886.

[21] DETTMERS T, LEWIS M, BELKADA Y, et al. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale[J]. Advances in Neural Information Processing Systems (NeurIPS), 2022, 35: 30318-30332.

[22] FRANTAR E, ASHKBOOS S, HOEFLER T, et al. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers[J]. International Conference on Learning Representations (ICLR), 2023.

[23] WANG H, MA S, DONG L, et al. BitNet: Scaling 1-bit Transformers for Large Language Models[A]. 2023.

[24] GU Y, DONG L, WEI F, et al. Knowledge Distillation of Large Language Models[A]. 2023.

[25] WU C, ZHAO W, LI B, et al. LaMini-GPT: Large Language Models with Minimal Data and Parameters[A]. 2023.

[26] DETTMERS T, PAGNONI A, HOLTMAN A, et al. QLoRA: Efficient Finetuning of Quantized LLMs[J]. Advances in Neural Information Processing Systems (NeurIPS), 2023, 36.

[27] XIAO G, TIAN Y, CHEN B, et al. StreamingLLM: Efficient Streaming Language Models with Attention Sinks[A]. 2023.

[28] DAO T, FU D Y, ERMON S, et al. FlashAttention: Fast and Memory-efficient Exact Attention with IO-awareness[J]. Advances in Neural Information Processing Systems (NeurIPS), 2022, 35: 16344-16359.

[29] CHEN T, MOREAU T, JIANG Z, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning[C]//Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2018: 578-594.

[30] GUJARATI A, SHEKHAR S, GARG T, et al. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up[C]//Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI). 2020: 443-462.

[31] PATEL P, CHOUKSE E, ZHANG C, et al. Splitwise: Efficient Generative LLM Inference Using Phase Splitting[C]//Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA). 2024: 118-132.

[32] HONG K, CHEN L, WANG Z, et al. semi-PD: Towards Efficient LLM Serving via Phase-Wise Disaggregated Computation and Unified Storage[A]. 2025.

[33] WU B, LIU S, ZHONG Y, et al. Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles: Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism[Z]. ACM, 2024: 640-654.

[34] HU C, HUANG H, XU L, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads[A]. 2024.

[35] NVIDIA Corporation. NVIDIA Dynamo: Distributed orchestration for large-scale LLM serving[EB/OL]. 2025. https://developer.nvidia.com/nvidia-dynamo.

[36] QIN R, LI Z, HE W, et al. Mooncake: A KVCache-centric Disaggregated Architecture for LLM Serving[A]. 2024.

[37] HU C, HUANG H, HU J, et al. MemServe: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool[A]. 2024.

[38] WANG L, YANG L, YU Y, et al. Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving[C]//Proceedings of the ACM Symposium on Cloud Computing (SoCC). 2021: 639-653.

[39] XIAO W, REN S, LI Y, et al. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning [C]//Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2022: 533-549.

[40] QIN R, YANG Y, ZHENG Q, et al. Allox: Compute Allocation in Hybrid Clusters[C]// Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys). 2023: 587-603.

[41] SHENG Y, ZHENG L, YUAN B, et al. FlexGen: High-throughput Generative Inference of Large Language Models with a Single GPU[C]//Proceedings of the 40th International Conference on Machine Learning (ICML). PMLR, 2023: 31094-31116.

[42] YANG Z, LUAN Z, LI B, et al. SkyPilot: An Intercloud Broker for Sky Computing[C]// Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2023: 577-596.

[43] TAORI R, GULRAJANI I, ZHANG T, et al. Stanford Alpaca: An Instruction-following LLaMA model[J/OL]. GitHub repository, 2023. https://github.com/tatsu-lab/stanford_alpaca.

[44] RUAN C, CHEN Y, TIAN D, et al. DynaServe: Unified and Elastic Execution for Dynamic Disaggregated LLM Serving[A]. 2025.

[45] ZHANG D, WANG H, LIU Y, et al. 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25): BlitzScale: Fast and Live Large Model Autoscaling with O(1) Host Caching[Z]. USENIX, 2025: 275-293.

[46] ZHANG S, ROLLER S, GOYAL N, et al. OPT: Open Pre-trained Transformer Language Models[A]. 2022. arXiv: 2205.01068.

[47] HONG K, DAI G, XU J, et al. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics[J]. Proceedings of Machine Learning and Systems, 2024, 6: 148-161.

[48] ZHU J, YANG R, HU C, et al. 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid): Perph: A workload co-location agent with online performance prediction and resource inference[Z]. IEEE, 2021: 176-185.

[49] SUN B, HUANG Z, ZHAO H, et al. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24): Llumnix: Dynamic Scheduling for Large Language Model Serving[M/OL]. Santa Clara, CA: USENIX Association, 2024: 173-191. https://www.usenix.org/conference/osdi24/presentation/sun-biao.

[50] MIAO X, SHI C, DUAN J, et al. ASPLOS '24: SpotServe: Serving Generative Large Language Models on Preemptible Instances[M/OL]. New York, NY, USA: Association for Computing Machinery, 2024: 1112-1127. https://doi.org/10.1145/3620665.3640411.

[51] HOLMES C, TANAKA M, WYATT M, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference[A]. 2024.

[52] CHEN S, JIANG R, YU D, et al. KVDirect: Distributed Disaggregated LLM Inference[A]. 2024.

[53] HE Y, XU M, WU J, et al. BanaServe: Unified KV Cache and Dynamic Module Migration for Balancing Disaggregated LLM Serving in AI Infrastructure[J/OL]. Software: Practice and Experience, 2025. DOI: 10.1002/spe.70054.

[54] HOOPER C, KIM S, MOHAMMADZADEH H, et al. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization[C]//Advances in Neural Information Processing Systems (NeurIPS): Vol. 37. 2024.

[55] ZHANG Z, SHENG Y, ZHOU T, et al. H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models[C]//Advances in Neural Information Processing Systems (NeurIPS): Vol. 36. 2023.

[56] AGRAWAL A, KEDIA N, PANWAR A, et al. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24): Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve[Z]. USENIX, 2024: 117-134.

[57] DU Z, QIAN Y, LIU X, et al. GLM: General Language Model Pretraining with Autoregressive Blank Infilling[C]//Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL). 2022: 320-335.

[58] PATEL P, CHOUKSE E, ZHANG C, et al. Splitwise: Efficient Generative LLM Inference Using Phase Splitting[C]//Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA). IEEE, 2024: 118-132.

致　谢

时光荏苒，转眼间三年的硕士生涯已接近尾声。2023 年 9 月，我怀揣着对大模型的懵懂好奇踏入南科大校园；2026 年 6 月，我带着一肚子关于 KV Cache 和批调度的” 奇怪知识” 准备离开。这三年，像是一次漫长的推理过程，有 Prefill 阶段的快速积累，也有 Decode 阶段的反复迭代，中间还夹杂着几次 OOM 崩溃和无数次的 checkpoint 回滚。所幸，最终生成了一个还算满意的输出。

首先感谢我的导师徐敏贤副研究员。徐老师兼具开放包容的学术视野与务实高效的治学风格，让我在探索大模型推理优化这个充满不确定性的领域时，既拥有自由试错的空间，又始终保持着正确的方向感。那些深夜改论文的时光，徐老师逐字逐句的批注与悉心指导，不仅打磨了我的学术表达能力，更让我深刻体会到什么是严谨的科研态度。师恩如山，铭记于心。

感谢实验室的师兄们：胡侃师兄和温林峰师兄在我初入实验室时给予了无微不至的关怀，帮助我快速完成了从本科生到研究生的” 上下文切换”，顺利适应了科研节奏。感谢宋盛叶和武帅鹏，我们一同上课、一同赶 DDL、一同在期末周挣扎，这些” 并肩作战” 的经历构成了我研究生生活最鲜活的底色。

特别感谢我的室友兼战友吴静峰。我们不仅在 CoCoScale 项目中并肩奋斗，更在无数个深夜讨论论文、调试代码、甚至讨论人生。那些关于学术、关于未来、关于选择的深夜长谈，让我在迷茫时找到方向，在焦虑时获得平静。那些漫步在陌生街道上的时光，是高压科研生活中最珍贵的” 推理间隙”，让我得以清空缓存、重新加载。

感谢郑婉仪师妹，作为最早加入我科研项目的伙伴，我们一起完成了 UELLM 项目，从最初的想法到最终的实现，这段经历让我体会到协作的力量。感谢廖俊涵师弟，我们在 CCF 服务计算比赛中并肩作战，更在 PD 分离架构的探讨中碰撞出许多思想的火花，让我收获颇丰。

特别感谢胡建民师弟，与我一同在阿里爱橙科技实习。那段日子，我既要推进论文，又要面对秋招的重重压力，面试的屡次受挫让我一度陷入低谷。感谢建民的帮助让原本孤独的实习生活变得温暖而有力量。

非常感谢阿里爱橙科技 AIOS 团队的每一位同事。你们不仅为我提供了宝贵的工业界资源和真实的业务场景，更让我见识到了顶尖工程师的专业素养与工匠精神。这段实习经历让我明白，好的研究不仅要发论文，更要能落地。毕竟，不能

部署的优化方案，就像没有显存的 GPU，看着再漂亮也没用。

感谢我的父母。你们不懂什么是 KV Cache、什么是投机解码，但你们懂得在我每一个重要时刻给予最坚实的支撑。我一度怀疑自己的时候，是你们无条件的信任让我有勇气再次出发。你们的爱就像最稳定的基座模型，无论我在上面做什么样的微调、遇到什么样的分布偏移，都能提供始终如一的温暖与支持。这三年，我在深圳追逐梦想，你们在远方默默守候，电话那头的一句”家里都好，你放心”，是我所有工作的最终优化目标。谢谢你们，让我在最艰难的时刻也能坚持下去，让我知道无论输出什么样的结果，总有人在等着我回家。

行文至此，窗外的荔枝花又开了。研究生的三年，是我人生中最重要的一次”预训练”。这三年教会我：真正的优化不在于消除所有延迟，而在于即使面对长尾分布的挫折，依然保持稳定的吞吐量。愿我们都能在各自的领域，实现更高的吞吐量与更低的延迟。

何忆源

2026 年 2 月于安徽

个人简历、在学期间完成的相关学术成果

个人简历

2018 年 9 月——2022 年 6 月，在武汉大学计算机学院学习，获得学士学位。

2023 年 9 月——2026 年 6 月，在南方科技大学学习并攻读计算机科学硕士学位。

获奖情况：国家奖学金、南科大优秀研究生、中国服务计算一等奖、先进院优秀学生等。

在学期间完成的相关学术成果

学术论文

[1] He Y, Xu M, Wu J, et al. UELLM: A Unified and Efficient Approach for Large Language Model Inference Serving[C]//Proceedings of the 22nd International Conference on Service-Oriented Computing (ICSOC). Springer, 2024: 218–235. （EI 收录，对应学位论文第三章）

[2] He Y, Xu M, Wu J, et al. BanaServe: Unified KV Cache and Dynamic Module Migration for Balancing Disaggregated LLM Serving in AI Infrastructure[J]. Software: Practice and Experience, 2025. DOI: 10.1002/spe.70054. （SCI 收录，对应学位论文第四章）

参与的科研项目及获奖情况

[3] 2024 年国家奖学金。

[4] CCF 2025 中国服务计算创新大赛一等奖

[5] 2024 年南方科技大学优秀研究生。

[6] 2025 年南方科技大学优秀研究生。

[7] 2024 年深圳先进技术研究院数字所“优秀学生奖”。

[8] 2024 年深圳先进技术研究院数字所“优秀学生奖”。