高技术通讯 2026 年 第36卷 第7期:663 - 676

doi:10.3772/j.issn.1002 - 0470.2026.07.001

大语言模型推理优化研究综述①

胡存琛②* ** *** 孙凝晖* ** 王 卅③* **

(* 中国科学院计算技术研究所处理器芯片全国重点实验室 北京 100190)

(** 中国科学院大学 北京 100049)

(*** 中国电信云计算研究院 北京 100053)

摘 要 大语言模型(large language model,LLM)快速增长的参数规模以及自回归特性使得推理的高服务成本与低资源使用效率之间出现巨大的矛盾。推理系统的服务性能依赖于系统架构、算法、模型等各方面。本文总结了推理系统面临的问题和挑战,并且从请求处理流程和模型服务架构出发,重点介绍了请求调度、资源管理、令牌生成、分布式推理以及推理缓存方面的研究进展,分析了提升推理系统效率所采用的关键技术。最后,本文探讨了异构硬件资源与文生图场景下降低推理服务成本的挑战,认为软硬协同、硬件能力感知的设计有助于进一步提升资源效率以及降低推理成本。

关键词 大模型服务; 请求调度; 缓存加速; 显存管理; 算子优化

随着大语言模型[1](large language model, LLM)的快速发展,LLM 已经被广泛应用在生成式对话、文本总结以及代码生成等各类不同场景[2]。当前大多数 LLM 都是基于 Transformer 架构,其精确度与模型参数量及序列输入长度正相关,扩展模型参数规模以及输入序列长度均能够提升输出结果质量。

面向不同场景的 LLM 负载、语言模型的自回归特性、大规模模型参数以及长文本序列对大模型推理服务带来了以下问题:(1)多样的任务负载使 LLM 服务需要在请求间进行细粒度调度[3-4];(2)语言模型的自回归特性使得请求(request)输出长度不可预知,从而无法为请求精确分配显存[5-6];(3)模型参数规模的增大不仅导致推理过程中计算和显存资源的消耗急剧上升,而且令牌(token)生成延迟也随之增长,这对模型部署策略提出了更灵活和高效的要求[7-8];(4)长文本序列使计算资源成为瓶颈,缓解算力不足变得至关重要。

除此之外,由于推理过程中存在大量不规则的显存访问,频繁的数据访问开销导致算力使用效率始终较低。例如,Google 数据中心中,推理过程中资源的平均利用率不到 40%[9]。因此,大量的资源需求与低效的资源使用效率之间形成了矛盾。为此,提升 LLM 推理效率、降低服务成本成为当前大多数研究工作关注的核心问题。

本文首先介绍了 LLM 背景与运行机制,并从请求调度、资源分配、令牌生成、分布式推理以及推理缓存多个维度分析了推理过程中存在的问题;其次,对各类研究工作及其关键技术进行了总结与分析;最后,本文讨论了推理在异构硬件资源场景下面临的新挑战、文生图的新场景以及推理引擎异步化的新问题,并展望了未来发展方向。与现有综述文献[10 - 11]侧重于从模型优化层面的总结不同,本文从系统层面对相关研究工作进行了分析和归类。与文献[12]相比,本文不仅分析了相关研究工作的优缺点,并且探讨了它们之间关键技术的关联性。

① 国家自然科学基金(U24B6012)资助项目。
② 男,1991 年生,博士生;研究方向:计算机系统结构;E-mail: hucunchen21b@ ict. ac. cn。
③ 通信作者,E-mail: wangsa@ ict. ac. cn。
(收稿日期:2025 - 02 - 24)

1　大语言模型

1.1　大模型机制

目前,大多数语言模型采用 Transformer 架构,如元人工智能大语言模型(large language model meta AI,LLaMA)[13]、生成式预训练转换器(generative pretrained Transformer,GPT)[14]以及开放预训练变换器(open pre-trained Transformer,OPT)[15]等。Transformer 的核心是量化输入序列中不同位置 token之间的依赖关系的注意力机制[1],其主要过程如下所述。首先利用模型权重矩阵对输入序列($X = [X_1 X_2 \cdots X_n]$)进行线性变换,生成查询(Query,Q)、键(Key,K)和值(Value,V)。$\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 的线性变换公式表示为式(1)。

$$\begin{aligned} \boldsymbol{Q} &= X \times \boldsymbol{W}_{\mathrm{Q}} \\ \boldsymbol{K} &= X \times \boldsymbol{W}_{\mathrm{K}} \\ \boldsymbol{V} &= X \times \boldsymbol{W}_{\mathrm{V}} \end{aligned} \tag{1}$$

式中:$\boldsymbol{W}_{\mathrm{Q}}$、$\boldsymbol{W}_{\mathrm{K}}$、$\boldsymbol{W}_{\mathrm{V}}$ 是模型预训练权重矩阵。完成 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵映射后,利用式(2)计算注意力分数。

$$\mathrm{Attention} = \mathrm{softmax}\left(\frac{\boldsymbol{Q} \times \boldsymbol{K}^{\mathrm{T}}}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{2}$$

式中:d_k 是矩阵 $\boldsymbol{K}$ 的维度。

注意力分数计算完成后,通过投影操作对注意力分数进行线性变换,并将投影值传递到前馈神经网络(feed-forward neural network,FFN)。所有 token 完成注意力计算后,输出表示最大概率结果的 token 向量。模型执行过程中,生成每个 token 均依赖于前置数据,即“键值缓存(key-value cache,KV)”。为了减少计算开销,通常将 KV 缓存存储在图形处理器(graphics processing unit,GPU)中以便复用。

如图 1 所示,推理过程分为预填充(Prefill)以及生成(Decode)2 个阶段。(1)Prefill 阶段。该阶段是计算密集型的矩阵运算,通过并行处理输入序列,生成首个 token 结果。其衡量指标是首个 token 完成时间(time to first token,TTFT)。(2)Decode 阶段。该阶段是内存密集型的矩阵向量运算,顺序生成后续 token 结果。其衡量指标是单个 token 完成时间(token between token,TBT)。

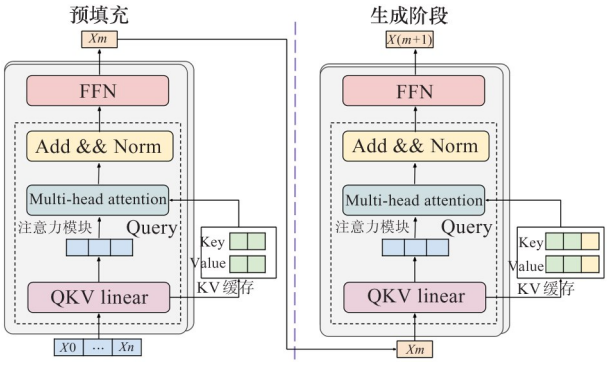

图 1　推理过程 2 个阶段

1.2　大语言模型推理问题分析

提升资源使用效率并降低推理成本是工业界与学术界关注的核心问题[9]。本文从请求调度、资源管理、令牌生成、分布式推理以及推理缓存 5 个维度详细分析了提升推理效率的相关研究工作,并将其映射到如图 2 所示的请求流转环节中。

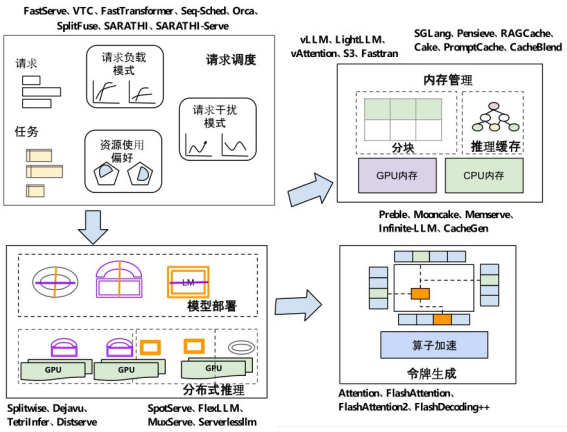

图 2　请求关键环节与工作分类

(1)请求调度。多样的任务负载会导致以下 2 个问题:1)未知请求长度导致大包阻塞;2)多请求并发增大缓存数据,导致计算及数据访问开销变大。因此,高效的推理调度不仅需要关注请求并发量,还需要协同资源的使用,以提升系统性能和资源利用率。

(2)资源管理。不断增大的模型参数导致对显存需求随之变大,不仅使得单张 GPU 无法部署模型,而且对高效显存管理提出了更高要求。因此,模型的多卡部署以及高效资源管理是推理系统的重要基础。

(3)令牌生成。注意力(Attention)算子与线性算子(Linear)是推理过程中高效计算的关键,其内部执行逻辑对生成效率有着较大影响。因此,提升算子执行效率是加速推理效率的核心优化方向[16]。

(4)分布式推理。Prefill 和 Decode 阶段的资源特征和用户时延要求存在差异。两阶段混合执行会导致两者的 token 生成延迟相互干扰。两阶段分离部署尽管可以避免干扰,但会导致单一的资源使用。因此,研究不同负载压力以及请求服务级别目标(service level objective,SLO),选择灵活的部署架构,是降低推理成本的关键。

(5)推理缓存。长文本序列能够提供丰富的语义信息,生成高质量输出结果。然而,它会显著增加 Prefill 阶段的计算量,加剧计算资源紧缺。通常,缓存复用能够有效减少计算量。因此,高效的缓存系统是加速推理过程的重要组成部分。

2 请求调度

高效的请求调度是保障性能和提高资源使用效率的基础。然而,推理任务的自回归解码机制、未知输出长度以及多样的资源需求,为请求调度策略和请求组合带来了新的挑战。

2.1 调度策略

快速、公平兼顾系统吞吐的调度策略是数据中心中的关键技术。常用的调度策略包括先来先服务(first come first serve,FCFS)、最短作业优先(shortest job first,SJF)、优先级调度以及时间片轮询等。其中,FCFS 能够保证请求执行顺序的公平性,但容易引起队头阻塞(head-of-line)[17]。最短任务优先策略利用请求完成时间避免了这一问题。然而,由于推理请求的输出长度未知,无法直接使用这一策略。为此,FastServe[16]提出基于时间片的多级优先权队列策略,具体如下:(1)优先权计算。在请求到达时,利用请求输入长度计算优先级,同时将其分配到不同时间片的优先权队列,避免长输入请求引起阻塞。(2)优先权调度。按照从高到低的队列优先权顺序依次取出待执行请求。(3)时间片管理。按请求时间片依次执行,当时间片耗尽时,将请求暂停并降级到下一优先级队列,重新等待调度,并循环该过程直至任务完成。该工作解决单个请求长时间占用资源的问题,实现了请求间资源使用的公平性。然而,它没有考虑用户间服务的公平性。为此,虚拟令牌计数器(virtual token counter,VTC)[18]提出面向不同用户的公平性服务。具体而言,它观察到用户的请求服务数量不能准确反映用户实际得到的服务时间。因此,VTC 采用“token 数量”衡量用户间服务时间的公平性:对于完成 token 数量少的用户,其请求优先响应。通过平衡用户间 token 完成数量,保障不同用户之间服务时间的公平性。

2.2 请求间并发

请求批量(Batch)技术广泛应用在分布式系统以及机器学习等方面。推理系统同样采用 Batch 技术提升系统吞吐量以及资源使用效率[1]。在推理系统中,Batch 指同一批次中的请求同时启动执行,并同时返回结果。然而,不同请求的输入和输出长度存在差异,这使得 Batch 策略与请求特性间产生矛盾:(1)短请求需要等待长请求完成,其执行时间被动延长;(2)等待过程中会造成闲置的计算资源。

为了解决上述问题,序列调度器(sequence scheduler,Seq-Sched)[3]是一种基于输出长度预测的调度策略,具体流程如下:(1)长度预测:当请求到达时,预测请求的输出长度范围并作为请求标签;(2)请求聚合:根据请求标签将具有相似输出长度的请求尽可能调度到同一批次;(3)批次执行:分批执行各请求集合。

然而,长度预测策略依赖于预测模型的准确性,并且无法彻底解决不同长度请求的等待。为此,Orca[4]提出了基于执行轮次的调度策略,其核心思想如下:在生成轮次(Iteration)中检查所有请求状态,从而实时感知请求状态变化,及时返回完成请求并且加入等待请求。该策略利用模型自回归特性,通过生成轮次检查请求状态,解决了请求间的等待,该技术已经广泛应用于各类推理框架[5,19]。

2.3 请求内组合

与传统请求的单一执行阶段不同,推理请求包括 Prefill 和 Decode 这 2 个阶段,它们不仅有着不同的算力和显存资源差异,而且具有不同的时延需求。

单独执行某个阶段会导致其他资源的浪费;而简单的混合执行,则可能造成阶段间的干扰,导致不能满足时延需求。因此,确定两阶段请求配比对于实现高效的计算密度和满足时延要求至关重要。

SplitFuse[7]观察发现,token 数量是影响 Prefill 阶段延迟的关键因素。对于总 token 数为 M 的 N 个请求,最大化系统吞吐的执行策略是将 M 个 token 按照固定数量重新组合执行。基于这一观察,它提出"长输入拆分,短输入合并"的执行策略。(1)长输入拆分将长输入请求拆分为较短序列以加速其响应;(2)短输入合并将短输入请求合并到一起,以充分利用计算资源。通过保持齐整的计算,使长输入请求获得更快的响应,同时提升短输入请求的系统吞吐。SARATHI[20]注意到,在流水线部署模式下,不同请求在 Prefill 和 Decode 阶段执行的微批次(Micro-Batch)存在较大时间差异,导致执行中产生资源空泡现象。此外,Prefill 阶段的吞吐收益随着 token 数量增大逐渐下降。为此,它提出"Prefill 分块、Decode 组合"执行策略:(1)将请求输入序列切分为多个子块;(2)将分块后的 Prefill 序列与 Decode 请求逐块组合在同批次执行。该方法通过 Prefill 请求提升了算力使用效率,同时利用 Decode 请求避免显存资源闲置。

基于文献[20],SARATHI-Serve[21]进一步设计了均衡 token 生成延迟与系统吞吐的服务框架,具体流程如下:首先计算单批次中能够执行的最大 token 数量;其次优先调度 Decode 请求以保障生成延迟;最后填充部分 Prefill 请求,以减少 Prefill 请求对 Decode 请求的阻塞影响。

本节从请求调度策略、请求间并行以及请求内组合 3 个维度,对请求调度的代表性工作进行了分类讨论,并将其关键技术与局限性总结如表 1 所示。

表 1　请求调度

工作	分类	调度粒度	优先级	混合粒度	局限性
FastServe[16]	调度策略	Iteration	请求时间片	Request	缺少时间片量化
VTC[17]	调度策略	Request	用户完成 token	/	未考虑请求公平性
FastTransformer[1]	请求间	Request	FCFS	/	存在显存浪费
Seq-Sched[3]	请求间	Request	FCFS	Request	额外预测开销
Orca[4]	请求间	Iteration	FCFS	Request	/
SplitFuse[7]	请求内	Iteration	Prefill	token	依赖负载长度
SARATHI[20]	请求内	Iteration	Prefill	token && Request	依赖负载长度

3　资源管理

为了应对不断增大的模型参数以及长文本序列,多卡模型部署以及显存资源管理已成为推理服务的核心基础。

3.1　模型部署

模型参数与数据长度是导致推理资源不足的 2 个关键因素。因此,如表 2 所示,多卡并行主要从模型切分和数据切分两方面解决模型部署的资源需求。

模型切分:模型执行时,数据在多层间独立计算并传递。因此,可以按层内与层间 2 个维度拆分。(1)张量并行[8]。层内横向或纵向切分算子,并将切分后的算子(如:注意力机制)分配到多个独立 GPU 执行。在算子完成计算后,利用集合通信原语(如:全归约(All-Reduce))将结果合并。张量并行需要频繁传输中间计算结果,对网络带宽要求较高。它适用于低延迟通信环境(如:高速串行计算机扩展总线标准(peripheral component interconnect express,PCIe)或 NVlink),不适用于高延迟的跨机网络(如:无限带宽(InfiniBand)或以太网)。(2)流水线并行[8]。将模型按层划分为多个独立执行阶段,为每个阶段分配硬件资源,并在层与层之间传递计算结果。流水线并行仅在阶段间进行数据传输,通信开销较小,适用于跨机网络环境。

数据切分:对于数据量较多或者长序列场景,需

要采用数据并行或序列并行。(1)数据并行[7]:最早应用于大模型训练场景,将大型数据集拆分到不同硬件设备上计算梯度。而在推理场景中,通常指部署多个独立的模型服务实例处理请求负载。(2)序列并行[22]:将长序列请求拆分为多个子序列,分散到多个设备处理。由于子序列之间存在计算依赖关系,不同设备间会存在大量通信开销,通常使用"计算-传输"并行方式以减少等待时间和通信瓶颈。

表 2　不同并行模式

并行策略	切分方式	适用场景	优势	局限性
数据并行[7]	输入数据	大批量数据	易扩展	批次限制
张量并行[8]	参数、激活	大模型	内存效率高	高通信开销
序列并行[22]	输入序列	长文本	降低长序列延迟	高通信开销
流水线并行[8]	模型层	深模型	支持深层模型	流水线阻塞

3.2　动态显存分配

推理过程中,显存分配存在 2 种策略:(1)预分配固定大小;(2)动态分配。这 2 种策略均能满足显存分配的基本需求,然而也存着不同问题:(1)预分配能够保障请求在推理过程中的 token 空间。然而,未知的请求输出长度会导致实际使用空间远小于预分配大小,产生"资源碎片"。(2)动态的显存分配操作处于 token 生成的关键路径,频繁的分配开销会影响请求执行效率。

为了解决上述问题,相关研究提出了不同的显存分配方案(如表 3 所示)。其中,S3[23]通过引入预测模型预估请求的输出长度,提前分配显存空间。该策略依赖预测器的准确性。如果预测准确度较低,不仅无法减少显存浪费,反而会由于预测器的额外计算开销导致性能下降。为此,vLLM[5]提出"页式"动态分配策略:将物理显存采取"页"大小划分,并将多个 token 组成一页,继而按页粒度为请求分配显存。当前,基于"页"粒度的显存分配技术已经广泛应用[5,19]。尽管"页式"分配缓解了显存碎片,但是并没有彻底解决显存浪费、同时引入了复杂的显存管理。

为进一步优化显存资源,LightLLM[24]提出了基于 token 粒度的 TokenAttention 机制,实现了 token 粒度的显存管理。而对于"页式"显存引入的复杂管理开销,vAttention[25-26]从硬件层面支持了连续显存管理,能够无感知地分配连续虚拟空间,并按需分配物理显存,避免了虚拟页与物理显存映射的管理开销。

表 3　显存管理

相关工作	分配粒度	显存连续性	碎片现象
vLLM[5]	页	不连续	有
LightLLM[24]	token	连续	无
vAttention[25]	页	连续	有
S3[23]	预测块	不连续	有
FastTransformer[18]	固定块	不连续	有

4　令牌生成

Attention 与 Linear 是推理过程中两类核心算子。其中,Attention 算子利用预训练权重,计算不同位置 token 的依赖关系,生成结果概率向量。Linear 算子作为神经网络的基础模块,用于数据变换。这两类算子在单个 token 生成时间中的占比超过 85%[8]。因此,优化两类算子的执行时间对于降低 token 生成延迟至关重要。

4.1　Attention 结构

KV 缓存是制约推理计算效率的关键因素。为了降低 KV 缓存数据量,如图 3 所示,注意力机制从多注意力头(multi-head attention,MHA)逐渐发展为多查询注意力(multi-query attention,MQA)、分组查询注意力(grouped-query attention,GQA)以及多头潜在注意力(multi-head latent attention,MLH)。

MHA 是标准的多注意力头[1]架构,它将 $\boldsymbol{Q}$、$\boldsymbol{K}$、$\boldsymbol{V}$ 矩阵沿模型隐藏维度切成 n 份,分别进行注意力计算,最后将 n 个结果连接起来。相比单注意力头,多注意力头能够关注不同特征。尽管多注意力头能够提升模型效果,但无法减少 KV 缓存数据量。

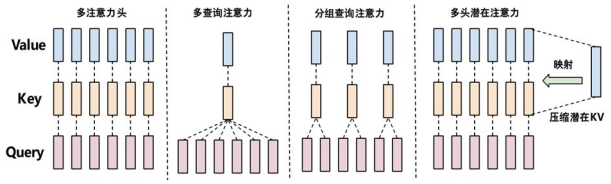

图 3　不同 Attention 算子架构[27]

为了降低缓存数据量，MQA 让所有注意力头共享一份 KV 缓存，将生成数据量减少为原来的 $1/H$（H 代表多头数量）[28-29]。然而，MQA 会导致推理精度下降。为此，GQA 提出折中策略，将注意力头分为 n 组，组内共享 KV 缓存，组间使用不同 KV 缓存。从而在压缩 KV 缓存数据量与性能间实现平衡。GQA 架构已经被广泛应用在 LLaMA[13]、DeepSeek[27]等模型。尽管如此，GQA 架构中模型准确度仍然会下降。为了进一步保障模型能力，DeepSeek-V2[27]观察到，注意力计算中的投影矩阵计算满足点乘的交换律。通过权重矩阵融合变换可以实现只存储投影前的数据，而无需保留 KV 缓存数据，从而显著减少存储数据量。同时，利用不同的投影矩阵，DeepSeek-V2 进一步提升了模型处理能力。

4.2　算子执行优化

GPU 采用两级存储架构，包含静态随机存储（static random access memory，SRAM）以及高带宽存储（high bandwidth memory，HBM）。如图 4 所示，以 A100 40 GB GPU 为例，每 108 个流处理器共用 192 kB的 SRAM，带宽为 19.0 TB · s^{-1}；所有流处理器共享 40 GB 的 HBM，带宽为 1.5 TB · s^{-1}。SRAM 的访问速率约为 HBM 的 10 倍，但是其容量远小于 HBM。因此，在处理大量数据时，需要在 HBM 与 SRAM 之间频繁移动数据。

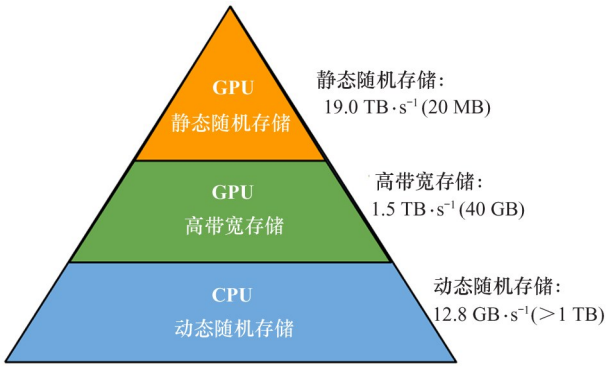

图 4　多级缓存存储结构[30]

Attention 算子是 token 生成的核心机制，其执行过程包括以下 3 个步骤：(1) 读取 $\boldsymbol{Q}$、$\boldsymbol{K}$ 矩阵，计算点积获得相似度矩阵 $\boldsymbol{S}$；(2) 矩阵 $\boldsymbol{S}$ 归一化，生成注意力权重矩阵 $\boldsymbol{P}$；(3) 将注意力权重矩阵 $\boldsymbol{P}$ 和值向量 $\boldsymbol{V}$ 加权求和，生成输出向量 $\boldsymbol{O}$。

上述 3 个计算过程均需要频繁从 HBM 中向 SRAM 加载数据，并在计算完成后将数据从 SRAM 移回 HBM。整个计算过程中存在大量数据的换入换出，导致计算延迟增加。“利用单次加载数据完成多个算子执行”的算子融合是解决这一问题的直接策略。然而，它存在以下局限性：矩阵分块处理会导致序列截断问题，使依赖全局最值的归一化函数 SoftMax 不能正确计算。为此，FlashAttention[30]提出局部 SoftMax 计算模式，通过在计算时减去局部最大值保证 SoftMax 结果稳定性。与标准 Attention 相比，FlashAttention[30]提高了 2 ~ 4 倍的计算效率，并节省了 10 ~ 20 倍的显存空间。尽管如此，其性能仅达到硬件矩阵乘法每秒浮点运算次数理论性能的 25% ~ 40%。

FlashAttention-2[31]观察到：(1) 现代 GPU 具备高效的矩阵专用计算部件（如 Tensor Core）；(2) 低效的线程块和线程束（wraps）分配会引起不必要的共享内存读写，降低计算资源的使用效率。因此，FlashAttention-2[31]提出利用请求的并发数量和注意力头维度拆解与分配请求序列。

然而，文献[31]无法支持高端硬件设备的专用数据传输硬件以及更低精度的新功能，导致其计算效率仅为 35%。针对这一问题，FlashDecoding ++[32]结合新硬件能力提出以下优化：(1) 利用新硬件特性，将数据生产线程与使用线程分配到不同 warp，实现 warp 间数据传输与计算重叠；(2) 拆分非矩阵与矩阵运算，使归一化和矩阵运算在不同轮次中并行执行；(3) 利用 Tensor Core 在低精度下进行矩阵乘法和累积运算。

本节主要介绍了 Attention 执行中的关键技术，并从计算粒度、专有硬件加速等方面将相关工作进行了总结，如表 4 所示。

表 4　Attention 计算优化

相关工作	计算粒度	硬件加速	任务管理	精度
原生 Attention[1]	/	/	/	16 位浮点
FlashAttention[30]	分块	/	/	16 位浮点
FlashAttention2[31]	分块	/	√	16 位浮点
FlashDecoding ++[32]	分块	√	√	8 位浮点

Linear 算子:投影映射模块在推理中有着关键作用,它通常由 2 层全连接网络(线性变换)和 1 个非线性激活函数组成。常用的激活函数包括 ReLU、GeLU、Swish 和 SwiGLU,其核心目标是增强模型在不同维度上非线性能力。实际使用中,主要调用 CUDA 基础线性代数子程序(CUDA basic linear algebra subprograms,cuBLAS)中的矩阵乘法应用程序编程接口(application programming interface,API)实现。然而,非定制化的矩阵乘法会导致计算效率低下。因此,TensorRT-LLM[33]引入了专用矩阵向量乘法以提升计算效率。此外,FlashDecoding ++[32]利用启发式规则,根据输入矩阵规模动态选择匹配的乘法算子库,从而加速运算。

5　分布式推理

模型不同阶段特征的差异以及多卡部署有着多样化的分布式推理需求。本节将从分离式推理架构和弹性推理 2 个方面概述。

5.1　分离式架构

Prefill 和 Decode 两阶段分别是矩阵乘法与矩阵向量乘法 2 种不同的计算模式。它们在用户目标延迟和资源需求上存在差异,两阶段混合执行可能导致两者之间计算的相互等待和干扰。为了提升推理效率,可以将它们部署到不同实例,以避免资源抢占和执行等待。然而,高效的分离式架构需要考虑缓存传输、缓存放置以及实例配比因素,如表 5 所示。

表 5　PD 分离

相关工作	PD 异构性	PD 转换	KV 传输粒度	缓存位置	P2D 调度	容错
Splitwise[34]	同构	重分配	层	高带宽存储	请求数	/
Dejavu[35]	异构	重规划	层	高带宽存储 + 动态随机存取存储器	/	副本
TetriInfer[36]	同构	角色转换	请求	高带宽存储	预测 token 数	/
Distserve[37]	异构	重规划	请求	高带宽存储	请求数	/

数据传输与放置。独立部署 Prefill 和 Decode 阶段,它们之间需要传递请求元数据以及 KV 缓存数据。因此,需要考虑以下 2 个问题:(1)KV 缓存传输粒度;(2)KV 缓存数据放置。

(1)KV 缓存传输粒度

数据传输时延与模型参数规模、输入序列长度及网络带宽密切相关。单个请求的缓存传输时延 T 计算公式如下。

$$T(s) = \frac{4 \times S \times h \times L}{B} \tag{3}$$

式中:S 是请求输入长度,h 是模型隐藏参数维度,L 是模型层数,B 是网络带宽。例如,在 300 GB · s^{-1} 或 900 GB · s^{-1} 网络带宽中,OPT-13B 以及 OPT-175B 模型处理单个长度为 2 048 个 token 的请求,其数据传输时间仅为几毫秒至几十毫秒,对单个请求的完整推理执行时间的影响几乎可以忽略不计。因此,高带宽配置下,对于短序列采用请求粒度传输 KV 缓存不会影响推理效率。然而,当输入长度增大(如 2×10^3 扩展至 128×10^3)或处于较低网络带宽环境,按请求粒度的 KV 缓存传输则会显著增加推理延迟,尤其影响第 2 个 token 的响应时间。为了减少传输影响,Splitwise[34]提出“边算边传”策略:Prefill 计算过程中,模型的第 n 层开始执行时,第 $n-1$ 层计算已经完成。此时,已完成的 KV 缓存数据可以提前传输到 Decode 实例,从而减少数据传输对推理性能的影响。

(2)KV 缓存数据放置

Prefill 请求的资源需求与 Decode 实例实际空闲显存可能存在差距,导致完成 Prefill 的请求不能

立即获得 Decode 实例的显存空间以放置缓存,从而引起任务阻塞。为了解决显存空间不足导致问题,Dejavu[35]提出将缓存存储空间扩展到主机内存的多级架构。当 Decode 实例的显存空间不足时,缓存数据可以传输到 Decode 实例的主机内存,并根据使用情况动态迁移到显存中。

实例计算与配比:Prefill 和 Decode 阶段对算力和显存资源的需求不同,可以分别部署到不同配置的硬件:Prefill 实例部署在“高算力、小显存”的硬件;Decode 实例部署到“低算力、大显存”的硬件。

TetriInfer[36]研究表明:(1)Prefill 阶段对 token 数量敏感,在固定硬件配置下,系统的最高效吞吐量对应确定的 token 数量。请求粒度的并发容易导致算力瓶颈,造成同一批次中所有请求延迟增加。(2)Decode 阶段的计算量和系统吞吐量随着请求长度增大而增长,聚合请求会导致单实例出现热点问题。基于上述观察,它提出以下优化方案:(1)对于 Prefill 阶段的请求,在固定窗口内进行排序与切分重组,使每批次执行的 token 数量均满足最大系统吞吐的固定值;(2)引入小模型预测请求输出长度,用于请求在 Decode 实例间分发的负载均衡。此外,Distserve[37]提出通过搜索优化 Prefill 和 Decode 实例的并行策略最大化两阶段的实例配比收益,保障推理性能。

5.2 弹性推理

AWS、Azure 等云厂商通常提供资源与应用 2 种服务模式。资源模式是指将资源直接交付给用户,不关注其具体使用情况。应用服务指云厂商负责应用部署以及应用管理,用户只关注服务的调用。

在资源模式下,云厂商出售资源越多,盈利越高。然而,资源需求存在波动性,云厂商在某些时间段可能会有更多富余资源(如:凌晨)。为了将资源尽可能售罄,云厂商推出了比固定实例价格便宜 90% 的“可驱逐实例”,以吸引用户购买。尽管这类实例价格低廉,但是它会在资源紧缺时被回收,可能导致用户服务中断或需要迁移,对用户服务的可靠性有着更高的要求。为了在低成本的可驱逐实例上部署推理服务,并降低实例驱逐的影响,SpotServe[38]提出基于可驱逐实例的推理系统:(1)根据可用的实例情况和负载波动,动态调整模型配置,减少实例回收影响;(2)利用二分图匹配对实例迁移建模,找出迁移开销最小的方案,从而减少迁移开销;(3)细粒度提交推理状态,加速服务恢复。然而,用户实例中不仅有推理任务,还包括训练、微调等任务。通常,采用静态资源隔离、时分复用或者空分复用方式使这些任务在共享资源上执行。然而,它们都会导致资源的浪费。FlexLLM[39]表明,共享模型权重和中间激活的推理与微调任务之间的资源需求具有互补性。单批次中同时执行推理和微调任务,能够提升计算和显存资源的整体利用率。然而,不同任务间会存在相互干扰。因此,它提出细粒度任务组合策略:(1)token 拆分微调序列,灵活控制 token 数量,避免算力瓶颈;(2)融合推理与微调算子到单内核中执行,减少算子启动开销与权重加载次数;(3)动态调整推理与微调任务的 token 数量配比,确保推理任务生成 token 的 SLO。

微调和推理任务的融合能够提升单个模型对计算和显存资源的使用效率。但是无法解决多模型共享资源导致的低资源使用率问题。MuxServe[40]以最大系统吞吐为目标,提出混合执行多个模型的不同阶段请求的贪心策略:优先调度计算需求较大的模型,以充分利用 GPU 算力。同时在多模型请求执行的调度过程中,平衡不同模型间的任务数量,确保不同模型对资源占用的公平性。

Serverless 是云厂商提供应用服务的主要模式,也是提升资源共享效率的关键技术。很多云服务商(如:AWS、Azure)致力于推理服务的 Serverless 化,以此提升资源效率。然而,冷启动是推理 Serverless 化面临的核心挑战:从远端存储下载模型权重需要耗费大量时间。因此,降低模型参数加载时延是实现推理服务 Serverless 化的关键。针对这一问题,ServerlessLLM[41]观察发现推理服务所在的硬件设备通常存在大量闲置的主机内存和硬盘存储空间。基于此,它提出了细粒度、快速传输的多级存储架构,避免从远端存储下载模型权重。具体方案如下所述。(1)加速参数加载:将参数加载格式转换为顺序块读取与张量寻址,实现大块数据读取与高效索引。(2)传输优化:在多层级存储之间并行加载数

据,加速不同存储层的数据读取。同时采用直接文件访问方式减少数据冗余复制。(3)亲和性迁移:模型迁移时感知参数存放位置,减少迁移的传输开销。

本节从弹性资源、任务混合和模型混合等不同维度阐述了相关研究工作的关键技术,并总结了它们的优势与局限,具体见表6。

表6 分布式推理

工作	目标	方法	优点	局限性
SpotServe[38]	弹性资源	利用感知负载和驱逐间隔时段的迁移保证使用驱逐实例的高效	高效使用可驱逐实例资源	需要已知驱逐时刻和时长
FlexLLM[39]	任务混部	组合不同微调和生成任务,满足两者 SLO	并发不同资源任务	同构模型
MuxServe[40]	模型混部	混合调度不同模型的不同阶段请求	多模型共享 GPU 资源	模型间阶段组合
Serverlessllm[41]	服务化	多层存储缓存与直接访问加速模型加载	降低模型加载时间	只加速参数加载

6 推理缓存

随着模型支持序列长度的快速增长,Prefill 对算力的巨大需求成为制约长序列高效处理的主要因素。然而,在多轮对话、思维链以及检索增强等场景中,请求间存在大量重复序列,导致计算资源的重复消耗。这使得紧缺的算力资源以及重复算力消耗之间形成矛盾。在存储系统中,缓存通常被用于避免重复数据的计算。本节从单机缓存与分布式缓存来讨论缓存在推理系统中的作用。

6.1 单机缓存

基于哈希表的管理是常见的资源复用记录方式。然而,哈希需要迭代计算请求内所有 token,更新与计算操作的开销较高。为了降低开销,SGLang[19]提出基于前缀树的 token 记录结构。通过树状结构的 token管理降低了查找、插入以及删除等操作开销。

实际推理过程中,token 缓存数量往往超过单个 GPU 显存容量。通常采用多层级存储解决显存空间问题。然而,多层级存储存在以下 2 个问题:(1)缓存数据的驱逐策略;(2)缓存数据复用的时效性难以衡量。针对缓存的驱逐管理,Pensieve[42]提出 token 粒度驱逐策略,首先按复用时间筛选出未复用请求;其次,按 token 粒度从后向前逐个驱逐未复用请求,以此最大化缓存数据的复用概率;最后,引入非连续空间的注意力计算,支持多位置、不连续的显存空间的数据计算,避免频繁的数据交换造成显存碎片。

KV 缓存复用在检索增强场景中也有着广泛需求。检索增强是指在向量数据库进行相似搜索,获取与输入内容匹配的文本信息,并将其与原请求合并,作为模型输入。RAGCache[43]观察到,检索数据存在冷热差异。因此,它提出了基于文本热度管理的多级动态缓存系统:根据访问热度区分文档放置位置,热度高的文档放置在 GPU 显存,热度低的放在主机内存,从而提高检索过程中的缓存命中率。

上述研究工作主要是聚焦于利用缓存复用提升首 token 完成时间,但是缺少请求缓存复用与计算执行的结合。具体而言,随着 token 数量的增加,计算开销逐渐增大,而 KV 缓存的传输开销则与 token 位置无关,仅使用缓存复用会导致算力资源闲置。因此,Cake[44]提出计算与缓存结合策略,从首个 token 开始计算,同时从最后一个 token 加载匹配数据,直到所有 token 都具备 KV 缓存。该策略同时利用了硬件的计算能力和传输带宽,加速了缓存获取。

实际场景中,尽管请求间有很多相同 token,但是不同的前缀导致无法进行数据复用。为了解决这一问题,PromptCache[45]提出序列模块化复用方式:提前计算频繁出现的前缀,后续请求的相同位置直接使用匹配的前缀数据。然而,当出现未计算过注意力的不同块组合时,会导致性能下降。因此,该方法仅适合于块间的交叉注意力计算值较低的场景。为了提升多个文本段复用性能,CacheBlend[46]提出使用邻近层偏差信息更新缓存策略:利用上层的 KV 缓存信息选择性重计算下层缓存数据。该策略

有效减少了未计算块的交叉注意力影响,同时提升了缓存复用的性能。

最后,本节针对单机缓存相关的研究工作,从复用粒度、存储结构等不同维度进行了总结与对比,具体见表 7 所示。表 7 中,CPU 表示中央处理器(central processing unit)。

表 7　单机缓存复用

相关工作	粒度	位置相关性	存储结构	管理结构	复用模式	驱逐策略	质量
SGLang[19]	Token	前缀匹配	GPU	树	完全传输	请求	高
Pensieve[42]	Token	前缀匹配	CPU-GPU	树	完全传输	热度 & 令牌	高
RAGCache[43]	文本	前缀匹配	CPU-GPU	树	完全传输	文本热度	高
Cake[44]	Token	完全匹配	GPU	/	部分传部分算	/	高
PromptCache[45]	模块	位置相关	GPU	字典	完全传输	/	低
CacheBlend[46]	序列块	位置无关	GPU	/	部分重用	/	中

6.2　分布式缓存

随着模型参数规模不断增大,部署推理系统所需的 GPU 数量不断增加,如表 8 所示,分布式缓存复用与数据放置在不同场景中变得愈发重要。

表 8　分布式缓存复用

相关工作	架构	目标场景	缓存空间	解决方式
Preble[47]	合一	数据热点	独立	token 匹配数量与实例请求数均衡负载
Mooncake[48]	分离	长文本数据热点	统一	副本级别的热点缓存数据复制迁移
Memserve[49]	分离	缓存收益	统一	算子粒度收益模型评价缓存收益
CacheGen[50]	合一	张量缓存	独立	缓存数据的局部性,编码成流失数据传输
Infinite-LLM[51]	合一	缓存统一放置	统一	细粒度缓存管理,统一管理数据放置
LoongServe[52]	分离	长序列并行	统一	划分动态实例组管理,动态扩展、缩放

对于分布式推理缓存,直接的策略是将前缀树扩展到分布式场景中管理缓存数据。然而,分布式场景中仅基于缓存匹配的请求调度可能导致不同实例请求数量不均衡,造成一些实例成为热点,出现性能下降。因此,Preble[47] 提出实例负载数量与共享前缀长度结合的策略:首先,利用全局调度器维护所有实例的缓存前缀,定期同步实例的缓存数据,同时记录实例未执行的请求数量。其次,在全局前缀树上搜索请求最长匹配 token 数量,并将其与请求待计算数量比较。如果匹配数量大于待计算数量,则将请求发送到相应实例;否则,将请求分配给负载最低的实例。该策略通过综合考虑等待请求数量与前缀长度,减少了缓存匹配度对负载均衡的影响。

除此之外,在分离式推理架构中的长文本场景,MoonCake[48] 提出了基于预测的早期拒绝策略与启发式缓存热点迁移机制,具体流程如下:(1)结合系统负载强度和 SLO 需求,制定“早期拒绝”规则,减少无效的 Prefill 计算,避免资源浪费。(2)利用系统层次的观测指标,量化与预测未来负载,降低规则化策略的不确定性。(3)主动复制热度高的缓存块到其他实例,平衡不同实例间复用热度。通过负载预测与缓存均衡,该策略能够提升复用效率,然而,在分布式场景中,仍然需要能够评估缓存复用收益的模型。

MemServe[49] 在构建分布式缓存的基础上,将模型算子按计算与访存密度分为 3 类:常量密度、计算密度、访存密度。通过观测不同类别算子,它分别对计算密度型算子的算力利用率(model FLOPs utilization,MFU)以及内存密度型算子的带宽利用率(model bandwidth utilization,MBU)建模,实现在不同输入与前缀长度下估算请求执行时延,从而评估了缓存复用的有效性。

分布式缓存复用能够提升性能,然而跨节点的KV缓存传输受到网络延迟影响,可能导致请求性能反而出现下降。通常,压缩技术可以减少传输数据量,但数据压缩会导致信息丢失,且压缩后的KV缓存仍然以张量结构存储。针对这一问题,CacheGen[50]提出基于流式传输的KV缓存编解码方案:(1)计算相邻token的KV缓存差异,保留token间的局部性特征;(2)基于不同模型层对生成token的影响不同的观察(浅层对生成token的影响力大于深层)。对不同层的数据采用不同精度量化,高精度量化浅层数据,低精度量化深层数据;(3)将token张量差值编码成比特流传输,减少数据传输开销。

除了缓存复用和传输加速,支持长序列请求的显存分配和数据放置也是分布式缓存的关键问题。Infinite-LLM[51]提出了一种自适应的分布式注意力算法,通过将缓存划分为细粒度键值对单元,实现KV缓存数据的分布式放置与动态管理。当服务实例的显存不足以满足KV缓存需求时,该算法能够无感知地从集群的其他空闲实例分配显存,实现全局显存的统一分配。细粒度缓存切分提高了显存使用效率,但是它没有考虑不同请求阶段间的弹性资源需求。

针对Prefill和Decode不平衡的计算和显存资源特征,LoongServe[52]提出了分离式架构的弹性缓存框架:(1)动态将实例划分为不相交的处理单元,组内实例并行处理token,并将数据传递到邻近实例组以完成剩余计算;(2)根据负载动态扩展与缩减并发实例组:当组内计算需求下降时,减少实例数量与并发度。当计算或显存需求增加时,扩展新实例;(3)采用主从模式管理实例,“主实例”负责分配计算任务,并汇聚计算结果;(4)通过动态切换Prefill和Decode实例角色,减少数据迁移开销。

7 挑战与展望

本文介绍的研究工作聚焦于同构算力资源下提升推理资源效率。然而,随着模型规模的增长以及硬件资源的快速发展,数据中心中异构硬件资源已经成为了普遍现象。在异构算力集群中,硬件在算力、显存以及带宽上存在差异,降低推理成本的复杂度随之增加。此外,大模型也被广泛使用于文生图场景。因此,现有研究工作仍面临以下问题与挑战。

(1)现有研究工作仍聚焦于推理两阶段的特征,缺少对算子粒度的执行差异的关注,Attention和多层感知机(multi-layer perceptron,MLP)算子的计算强度及资源消耗不同。细粒度的算子切分能够更好适配异构数据中心。例如,将Attention算子部署于显存性价比高的硬件,将MLP算子部署于计算性价比高的硬件。然而,细粒度的执行流对执行调度以及模型部署带来了新挑战。

(2)分离式架构下,异构资源导致模型两阶段被部署到不同类型机器。统一建模异构资源的计算与显存能力,定义不同阶段的资源配比,并根据SLO需求动态调整实例配置,是一个亟待解决的问题。

(3)异构集群中包含HBM、DRAM、固态硬盘(solid state drive,SSD)等多级存储需求。不同存储介质的数据传输涉及多种不同协议(如:计算快速链接(compute express link,CXL)、NVIDIA集合通信库(NVIDIA collective communications library,NCCL)、远程直接数据存取(remote direct memory access,RDMA)等)。构建具备池化能力、标准接口以及统一管理的存储系统,能够有效提升资源使用效率,是云厂商关注的核心问题。

(4)异步推理引擎。目前,推理框架中请求预处理与推理引擎执行之间仍然是同步方式。而请求的预处理会阻塞推理引擎的高效执行。对于开发人员而言,需要重新考虑推理框架中请求接收、编码、调度等模块的异步化来提升推理框架性能。

(5)文生图架构中,图片生成的计算开销大,执行时间长,并且需要多个模型协同执行。通过感知硬件计算能力加速请求执行、动态调整不同模型部署资源,减少资源冗余是值得探索的方向。

总结来说,异构资源以及新服务场景对资源管理、模型部署等多个不同维度提出了新挑战。将异构硬件资源能力融合,构建软硬结合、细粒度请求的全生命周期管理,有助于提升资源效率。

8 结 论

本文首先介绍了大模型推理系统的关键优化技

术,分析并讨论了请求调度、资源管理、令牌生成、分布式推理以及推理缓存方面的代表性工作。最后,对异构资源与文生图场景中提升推理性能的挑战进行了展望,提出软硬件协同设计与感知硬件能力有助于进一步提升推理系统性能,降低服务成本。

参考文献

[1] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C]//Proceedings of the 31st International Conference on Neural Information Processing Systems. New York, USA: Curran Associates, 2017:6000 – 6010.

[2] OpenAI. Introducing ChatGPT[EB/OL]. [2025 – 01 – 18]. https://openai.com/blog/.

[3] Zheng Z, Ren X, Xue F, et al. Response length perception and sequence scheduling: an LLM-empowered LLM inference pipeline[C]//Proceedings of the 37th International Conference on Neural Information Processing Systems. New York, USA: Curran Associates, 2023:65517 – 65530.

[4] Yu G I, Jeong J S, Kim G W, et al. Orca: a distributed serving system for transformer-based generative models[C]//Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation. Berkeley, USA: USENIX Association, 2022:521 – 538.

[5] Kwon W, Li Z, Zhuang S, et al. Efficient memory management for large language model serving with PagedAttention[C]//Proceedings of the 29th Symposium on Operating Systems Principles. New York, USA: ACM, 2023: 611 – 626.

[6] Jo A. The promise and peril of generative AI[J]. Nature, 2023, 614(1):214 – 216.

[7] Holmes C, Tanaka M, Wyatt M, et al. DeepSpeed-FastGen: high-throughput text generation for LLMs via MII and DeepSpeed-inference[EB/OL]. (2024 – 01 – 09)[2025 – 01 – 18]. https://arxiv.org/pdf/2401.08671.

[8] Shoeybi M, Patwary M, Puri R, et al. Megatron-LM: training multi-billion parameter language models using model parallelism[EB/OL]. (2020 – 03 – 13)[2025 – 01 – 18]. https://arxiv.org/pdf/1909.08053.

[9] Anil R, Dai A M, Firat O, et al. Palm 2 technical report[EB/OL]. (2023 – 09 – 13)[2025 – 01 – 18]. https://arxiv.org/pdf/2305.10403.

[10] Zhu X, Li J, Liu Y, et al. A survey on model compression for large language models[J]. Transactions of the Association for Computational Linguistics, 2024, 12: 1556-1577.

[11] Wang W, Chen W, Luo Y, et al. Model compression and efficient inference for large language models: a survey[EB/OL]. (2024 – 02 – 15)[2025 – 01 – 18]. https://arxiv.org/pdf/2402.09748.

[12] Miao X, Oliaro G, Zhang Z, et al. Towards efficient generative large language model serving: a survey from algorithms to systems[EB/OL]. (2023 – 11 – 23)[2025 – 01 – 18]. https://arxiv.org/pdf/2312.15234.

[13] Touvron H, Martin L, Stone K, et al. LLaMA 2: open foundation and fine-tuned chat models[EB/OL]. (2023 – 07 – 19)[2025 – 01 – 18]. https://openreview.net/pdf?id=UYZJpVEkvq.

[14] Achiam J, Adler S, Agarwal S, et al. GPT-4 technical report[EB/OL]. (2024 – 03 – 04)[2025 – 01 – 18]. https://arxiv.org/pdf/2303.08774.

[15] Zhang S, Roller S, Goyal N, et al. OPT: open pretrained transformer language models[EB/OL]. (2022 – 06 – 21)[2025 – 01 – 18]. https://arxiv.org/pdf/2205.01068.

[16] Shazeer N. Fast transformer decoding: one write-head is all you need[EB/OL]. (2019 – 11 – 06)[2025 – 01 – 18]. https://arxiv.org/pdf/1911.02150.

[17] Wu B, Zhong Y, Zhang Z, et al. Fast distributed inference serving for large language models[EB/OL]. (2024 – 09 – 25)[2025 – 01 – 18]. https://arxiv.org/pdf/2305.05920.

[18] Sheng Y, Cao S, Li D, et al. Fairness in serving large language models[C]// Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation. Berkeley, USA: USENIX Association, 2024:965 – 988.

[19] Zheng L, Yin L, Xie Z, et al. SGLang: efficient execution of structured language model programs[J]. Advances in Neural Information Processing Systems, 2025, 37: 62557 – 62583.

[20] Agrawal A, Panwar A, Mohan J, et al. SARATHI: efficient LLM inference by piggybacking decodes with chunked prefills[EB/OL]. (2023 – 08 – 31)[2025 – 01 – 18]. https://arxiv.org/pdf/2308.16369.

[21] Agrawal A, Kedia N, Panwar A, et al. Taming throughput-latency tradeoff in LLM inference with Sarathi-

serve[C]//Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation. Berkeley, USA: USENIX Association, 2024:117 - 134.

[22] Li S, Xue F, Baranwal C, et al. Sequence parallelism: long sequence training from system perspective[C]//The 61st Annual Meeting of the Association For Computational Linguistics. Stroudsburg, USA: ACL,2023,2391 - 2404.

[23] Jin Y, Wu C F, Brooks D, et al. S3: Increasing GPU utilization during generative inference for higher throughput[C]//Proceedings of the 35st International Conference on Neural Information Processing Systems. New York, USA: Curran Associates, 2023,36:18015 - 18027.

[24] Ainlp. LightLLM[EB/OL]. (2023 - 08 - 03)[2025 - 01 - 18]. https://github.com/ModelTC/lightllm.

[25] Prabhu R, Nayak A, Mohan J, et al. vAttention: dynamic memory management for serving LLMs without PagedAttention[EB/OL]. (2024 - 05 - 07)[2025 - 01 - 18]. https://arxiv.org/pdf/2405.04437.

[26] Xu J,Zhang R,Guo C,et al. vTensor: flexible virtual tensor management for efficient LLM serving[EB/OL]. (2024 - 07 - 22)[2025 - 01 - 18]. https://arxiv.org/pdf/2407.15309.

[27] Liu A, Feng B, Wang B, et al. DeepSeek-v2: a strong, economical, and efficient mixture-of-experts language model[EB/OL]. (2024 - 07 - 19)[2025 - 01 - 18]. https://arxiv.org/pdf/2405.04434.

[28] Chowdhery A, Narang S, Devlin J, et al. Palm: scaling language modeling with pathways[J]. Journal of Machine Learning Research, 2023,24(240):1 - 113.

[29] Team G,Anil R,Borgeaud S,et al. Gemini: a family of highly capable multimodal models[EB/OL]. (2024 - 06 - 17)[2025 - 01 - 18]. https://arxiv.org/pdf/2312.11805.

[30] Dao T,Fu D,Ermon S,et al. FlashAttention: fast and memory-efficient exact attention with IO-awareness[C]//Proceedings of the 33st International Conference on Neural Information Processing Systems. New York, USA: Curran Associates, 2022,35:16344 - 16359.

[31] Dao T. FlashAttention-2: faster attention with better parallelism and work partitioning[EB/OL]. (2023 - 06 - 17)[2025 - 01 - 18]. https://arxiv.org/pdf/2307.08691.

[32] Hong K, Dai G, Xu J, et al. FlashDecoding ++ : faster large language model inference on GPUs[C]//Proceedings of the 7th MLSys Conference. Santa Clara, USA: ACM, 2024:1 - 16.

[33] Nvidia. TensorRT-LLLM[EB/OL]. (2023 - 10 - 17)[2025 - 01 - 18]. https://github.com/NVIDIA/TensorRT-LLM.

[34] 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. Piscataway, USA: IEEE, 2024: 118 - 132.

[35] Strati F,Mcallister S,Phanishayee A,et al. DejaVu:KV-cache streaming for fast, fault-tolerant generative LLM serving[C]//The 41st International Conference on Machine Learning. New York, USA:ACM,2024: 46745 - 46771.

[36] Hu C,Huang H,Xu L,et al. Inference without interference: disaggregate LLM inference for mixed downstream workloads[EB/OL]. (2024 - 01 - 20)[2025 - 01 - 18]. https://arxiv.org/pdf/2401.11181.

[37] 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. Berkeley,USA:USENIX Association,2024:193 - 210.

[38] Miao X P. SpotServe: serving generative large language models on preemptible instances[C]//Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York, USA: ACM, 2024:1112 - 1127.

[39] Miao X,Oliaro G,Cheng X,et al. FlexLLM: a system for co-serving large language model inference and parameter-efficient finetuning[EB/OL]. (2024 - 02 - 29)[2025 - 01 - 18]. https://arxiv.org/pdf/2402.18789.

[40] Duan J,Lu R,Duanmu H,et al. MuxServe: flexible multiplexing for efficient multiple LLM serving[C]// The 41st International Conference on Machine Learning. New York, USA: ACM, 2024:11905 - 11917.

[41] Fu Y,Xue L,Huang Y,et al. ServerlessLLM: locality-enhanced serverless inference for large language models[C]//Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation. Berkeley, USA: USENIX Association, 2024:135 - 153.

[42] Yu L, Lin J, Li J. Stateful large language model serving with pensieve[EB/OL]. (2024 - 10 - 07)[2025 - 01 - 18]. https://arxiv.org/pdf/2312.05516.

[43] Jin C, Zhang Z, Jiang X, et al. RAGCache: efficient

knowledge caching for retrieval-augmented generation[EB/OL]. (2024 - 04 - 25) [2025 - 01 - 18]. https://arxiv.org/pdf/2404.12457.

[44] Jin S, Liu X, Zhang Q, et al. Compute or load KV cache? Why not both[EB/OL]. (2024 - 10 - 04) [2025 - 01 - 18]. https://arxiv.org/pdf/2410.03065.

[45] Gim I, Chen G, Lee S, et al. Prompt cache: modular attention reuse for low-latency inference[J]. Proceedings of Machine Learning and Systems, 2024, 6: 325 - 338.

[46] Yao J, Li H, Liu Y, et al. CacheBlend: fast large language model serving with cached knowledge fusion[EB/OL]. (2024 - 06 - 03) [2025 - 01 - 18]. https://arxiv.org/pdf/2405.16444.

[47] Srivatsa V, He Z, Abhyankar R, et al. Preble: efficient distributed prompt scheduling for LLM serving[C]// International Conference on Learning Representations. Washington DC, USA: OpenReview. 2024: 1 - 23.

[48] Qin R, Li Z, He W, et al. Mooncake: a KV cache-centric disaggregated architecture for LLM serving [EB/OL]. (2024 - 06 - 09) [2025 - 01 - 18]. https://arxiv.org/pdf/2407.00079.

[49] Hu C, Huang H, Hu J, et al. Memserve: context caching for disaggregated LLM serving with elastic memory pool[EB/OL]. (2024 - 06 - 08) [2025 - 01 - 18]. https://arxiv.org/pdf/2406.17565.

[50] Liu Y, Li H, Cheng Y, et al. CacheGen: KV cache compression and streaming for fast large language model serving[C]// Proceedings of the ACM SIGCOMM 2024 Conference. New York, USA: ACM, 2024: 38 - 56.

[51] Lin B, Zhang C, Peng T, et al. Infinite-LLM: efficient LLM service for long context with distattention and distributed KV cache[EB/OL]. (2024 - 07 - 04) [2025 - 01 - 18]. https://arxiv.org/pdf/2401.02669.

[52] Wu B, Liu S, Zhong Y, et al. LoongServe: efficiently serving long-context large language models with elastic sequence parallelism [C] // Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles. New York, USA: ACM, 2024: 640 - 654.

A survey on optimization research for large language model inference

Hu Cunchen[*,**,***], Sun Ninghui[*,**], Wang Sa[*,**]

([*]State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**]University of Chinese Academy of Sciences, Beijing 100049)

([***]China Telecom Cloud Computing Research Institute, Beijing 100053)

Abstract

The rapid growth in parameter size of large language model (LLM) and their autoregressive nature has led to a significant contradiction between high service costs for inference and low resource utilization efficiency. The performance of inference systems depends on various factors, including system architecture, algorithms, and models. The paper summarizes the challenges faced by inference systems from the request processing flow and model service architecture. It focuses on research advancements in request scheduling, resource management, token generation, distributed inference, and inference caching. The paper also analyzes key technologies for improving inference system efficiency. Finally, this work discusses the challenges of reducing inference service costs in heterogeneous hardware resources and text-to-image scenarios, suggesting that software-hardware co-design and hardware-aware strategies can further enhance resource efficiency and reduce inference costs.

Key words: large language model serving, request scheduling, caching accelerate, memory management, operator optimization