计 算 机 研 究 与 发 展
Journal of Computer Research and Development

https://crad.ict.ac.cn
62(3): 545−562, 2025

本期推荐

大语言模型推理中的存储优化技术综述

葛旭冉[1] 欧 洋[1] 王 博[2] 赵 宇[3] 吴利舟[1] 王子聪[1] 陈志广[2] 肖 侬[1]

[1](国防科技大学计算机学院 长沙 410073)

[2](中山大学计算机学院 广州 510006)

[3](爱丁堡大学信息学院 爱丁堡 EH89AB)

(gexuran@nudt.edu.cn)

Survey of Storage Optimization Techniques in Large Language Model Inference

Ge Xuran[1], Ou Yang[1], Wang Bo[2], Zhao Yu[3], Wu Lizhou[1], Wang Zicong[1], Chen Zhiguang[2], and Xiao Nong[1]

[1] (*College of Computer Science and Technology*, *National University of Defense Technology*, *Changsha* 410073)

[2] (*School of Computer Science and Engineering*, *Sun Yat-sen University*, *Guangzhou* 510006)

[3] (*School of Informatics*, *University of Edinburgh*, *Edinburgh EH89AB*)

Abstract In recent years, LLM (large language model) has exhibited remarkable performance, profoundly transforming various aspects of human life. As these models grow in size and user demand for long-context inference increases, LLM inference systems face significant storage challenges. These challenges stem primarily from the vast number of model parameters and the key value cache required for efficient inference, both of which strain GPU memory resources. Additionally, inefficiencies in storage usage in distributed systems often result in over-provisioning and fault tolerance issues, further complicating resource management. Researchers explore memory optimization, heterogeneous storage, and distributed storage, synthesizing various research efforts to address GPU memory constraints and enhance resource utilization. Memory-optimized LLM inference systems improve GPU memory efficiency and reduce memory footprint through techniques like efficient key value cache management, compression, and attention operator optimization. Heterogeneous storage based LLM inference systems expand storage capacity by integrating various storage resources, thereby minimizing I/O overhead via tensor placement strategies, asynchronous data transfer, and intelligent memory allocation and prefetching methods. Distributed LLM systems enhance the utilization of multi-machine resources, boosting execution efficiency and fault tolerance in LLM inference tasks through batching, multi-level scheduling, and redundant replication. Finally, we review existing research and outline future research directions to further optimize storage solutions for LLM inference systems.

Key words LLM inference system; memory management; heterogeneous storage; distributed storage; fault recovery; serverless LLM inference

摘 要 近年来,大语言模型在多个领域展现出卓越的性能,显著改变了人类的生活方式.然而,随着模型规模的不断增长和用户对长上下文推理需求的增加,大语言模型推理系统在存储方面面临诸多问题.首先,庞大的模型参数和键值缓存导致GPU显存资源不足;其次,分布式大语言模型推理系统难以充分利用GPU

收稿日期: 2024-07-20; **修回日期**: 2025-01-20

基金项目: 国家自然科学基金项目(62272499, 62332021); 国家重点研发计划项目(2022YFB4500304); 广东省基础与应用基础研究重大项目(2019B030302002); 广东省高层次人才培养专项支持计划项目(2021TQ06X160); 琶洲实验室项目(PZL2023KF0001)

This work was supported by the National Natural Science Foundation of China (62272499,62332021), the National Key Research and Development Program of China (2022YFB4500304), the Major Program of Guangdong Basic and Applied Research (2019B030302002), the Guangdong Provincial Special Support Program for Cultivating High-Level Talents (2021TQ06X160), and the Program of Pazhou Lab (PZL2023KF0001).

通信作者: 欧洋 (ouyang06@nudt.edu.cn)

集群的存储资源,存在资源过度配置和存储容错的问题.因此,从显存优化、异构存储和分布式存储3方面入手,归纳总结了现有研究在解决GPU显存容量不足和资源利用率低下方面的努力.基于显存优化的大语言模型推理系统通过高效的键值缓存管理、压缩以及注意力算子优化,提高了GPU显存利用率,降低了显存占用.基于异构存储的大语言模型推理系统利用多种类别的存储资源扩展存储容量,通过张量放置策略、异步数据传输以及智能显存分配与预取技术,降低了异构存储带来的I/O开销.基于分布式存储的大语言模型推理系统通过批处理、多级调度、冗余复制等策略,优化了多机存储和计算资源的利用,提高了大语言模型推理任务的执行效率和容错能力.最后,总结了现有研究,并对未来的研究方向进行了展望.

关键词 大语言模型推理系统;显存管理;异构存储;分布式存储;故障恢复;无服务大语言模型推理

中图法分类号 TP391

DOI: 10.7544/issn1000-1239.202440628 **CSTR:** 32373.14.issn1000-1239.202440628

近年来,大语言模型(如GPT-4[1], OPT[2], Falcon[3], LLaMA[4]等)在编程助手、聊天机器人、文本摘要等多个领域都展现出了巨大的潜力和应用前景.据统计,使用GitHub Copilot[5]工具辅助编程,可以将开发时间缩短35%~40%[6].GPT-4在文本生成和语言理解等多种自然语言处理任务中的表现出色.

然而,当前大语言模型推理系统在存储方面仍面临许多亟待解决的问题,引起了学术界和工业界的广泛关注.首先,模型规模[7-8]的指数级增长对存储资源提出了巨大的挑战.例如,GPT-3拥有1 750亿参数[9],如此庞大的参数量需要极高的存储容量,现有的单卡GPU显存资源往往无法满足这一需求.其次,用户对长上下文推理的需求显著增加,使得生成式大语言模型在推理过程中需要存储大量的键值缓存,进一步加剧GPU显存容量不足的问题.最后,分布式大语言模型推理系统在GPU集群资源利用率和故障恢复方面也面临重大挑战.一方面,现有的分布式大语言模型推理系统在预填充和解码阶段之间存在显著的延迟差异,导致GPU[10]资源利用率低下;另一方面,由于解码阶段依赖于前面的上下文信息,系统在遇到故障时难以快速恢复.

针对上述问题,现有的研究工作从不同的角度出发,为大语言模型推理系统提供了诸多存储优化方案.根据不同优化方案的侧重点,本文将这些研究分为3类:基于显存优化的大语言模型推理系统、基于异构存储的大语言模型推理系统和基于分布式存储的大语言模型推理系统.

基于显存优化的大语言模型推理系统专注于提高GPU显存利用率和降低显存占用,以应对复杂的推理需求.这些技术包括高效的键值缓存管理、压缩,以及注意力算子优化.键值缓存管理通过改进GPU显存资源的管理和分配机制有效减少显存浪费;键值缓存压缩[11]通过丢弃不重要的标记降低显存占用;注意力算子优化通过改进算子的计算过程进一步减少对显存的需求.

基于异构存储的大语言模型推理系统利用GPU显存、CPU内存和SSD等多种存储资源来扩展存储容量.关键技术包括:张量放置策略,通过优化模型权重、激活量和键值缓存的存储位置来提高系统性能;异步数据传输,通过计算和数据传输重叠[12]来隐藏I/O延迟;智能显存分配与预取技术,通过识别大语言模型推理中的关键数据,并将其预先加载到GPU显存,以减少额外的数据传输.

基于分布式存储的大语言模型推理系统着眼于单节点存储与计算资源不足的问题,采用批处理、多级调度等策略提高多机资源利用率,以提高大语言模型推理的整体性能.此外,一些研究在分布式流水线并行中采用数据冗余技术,显著提高了分布式大语言模型推理系统的容错能力和扩展性.

存储优化技术在大语言模型推理中的重要性不容忽视.通过显存优化、异构存储和分布式存储等技术,可以有效应对当前大语言模型推理系统面临的存储挑战.接下来,本文将首先介绍大语言模型生成式推理的整体过程,再依次总结和分析这些存储优化技术.

1 大语言模型的生成式推理

1.1 基于Transformer的大语言模型

自回归的Transformer模型[13]是主流大语言模型(如GPT-3[9], GPT-4[1], LLaMA[4])广泛采用的架构.如图1所示,该架构包括1个嵌入层(embedding)、一系列顺序的Transformer层、1个线性层和1个Softmax操作.具体结构如下:

1) 嵌入层.输入提示标记经过嵌入层转换为隐

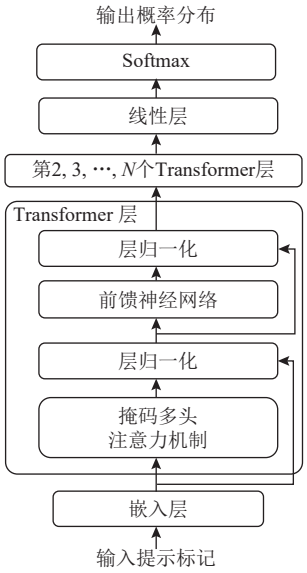


Fig. 1 Overview architecture of Transformer model

图 1 Transformer 模型整体架构图

藏状态，即 Transformer 层可识别的连续的高维向量表示.

2) Transformer 层. 每个 Transformer 层由多个子组件组成，包括 1 个掩码多头注意力（multi-head attention, MHA）机制、1 个前馈网络（feedforward network, FFN）和 2 个层归一化操作. 多头注意力机制通过并行计算多个独立的注意力头，允许模型同时关注输入序列的不同部分. 这些注意力头的输出被拼接后，经过线性变换得到最终的注意力输出. 前馈网络用于增强模型的非线性表示能力，层归一化则用于防止梯度爆炸，提高模型泛化能力. 经过多个 Transformer 层的处理后，模型生成了一个高维输出张量，该输出张量包含了输入标记在上下文中的语义信息和相关性.

3) 线性层和 Softmax 操作. 线性变换和 Softmax 函数将最后 1 个 Transformer 层输出张量映射为下一个生成标记 (token) 的输出概率分布，表示词汇表中每个标记作为下一个输出标记的可能性.

得到下一个标记的概率分布后，模型会根据特定的选择策略（如贪婪搜索、束搜索等）选择出下一个输出标记. 然后，将选定的输出标记转换成词嵌入，并将其与已经生成的标记序列结合. 重复上述步骤，继续生成下一个标记，直到达到最大长度或生成结束标记.

给定 1 个输入提示序列，大语言模型推理过程可以分为 2 个阶段：预填充阶段和解码阶段.

1.2 预填充阶段

在预填充阶段，Transformer 模型接收整个输入提示序列，生成第 1 个输出标记，并预填充初始键值缓存（key value cache, KV cache）. 其计算步骤有 7 步.

1）接收输入序列. 模型接收整个输入序列 $[x_1, x_2, \cdots, x_n]$作为提示.

2）查询向量、键向量和值向量计算. 注意力机制对每个输入标记 x_i 进行线性变换，计算得到的查询向量 $\boldsymbol{q}_i$、键向量 $\boldsymbol{k}_i$ 和值向量 $\boldsymbol{v}_i$：

$$\boldsymbol{q}_i = \boldsymbol{W}_q x_i,\ \boldsymbol{k}_i = \boldsymbol{W}_k x_i,\ \boldsymbol{v}_i = \boldsymbol{W}_v x_i, \tag{1}$$

其中 $\boldsymbol{W}_q$, $\boldsymbol{W}_k$, $\boldsymbol{W}_v$ 分别表示查询、键和值的权重矩阵.

3）注意力计算. 将查询向量 $\boldsymbol{q}_i$ 与所有键向量相乘，并对其进行 Softmax 计算得到每个标记的注意力分数 a_{ij}，在此基础上对所有值向量进行加权求和得到注意力输出向量 $\boldsymbol{o}_i$：

$$a_{ij} = \frac{\exp\left(\dfrac{\boldsymbol{q}_i^{\mathrm{T}}\boldsymbol{k}_j}{\sqrt{d}}\right)}{\sum\limits_{j=1}^{i}\exp\left(\dfrac{\boldsymbol{q}_i^{\mathrm{T}}\boldsymbol{k}_j}{\sqrt{d}}\right)},\ \boldsymbol{o}_i = \sum_{j=1}^{i} a_{ij}\boldsymbol{v}_j, \tag{2}$$

其中 Softmax 运算中的分母 d 表示键向量和查询向量的维度，用于进行归一化，以保证数值稳定.

4）前馈网络处理. 注意力层的输出 $\boldsymbol{o}_i$ 会经过 1 个前馈网络得到 $FFN(\boldsymbol{o}_i)$：

$$FFN(\boldsymbol{o}_i) = \boldsymbol{W}_2\,\sigma(\boldsymbol{W}_1\,\boldsymbol{o}_i), \tag{3}$$

该前馈网络通常由 2 个全连接层和 1 个激活函数组成. 其中 $\boldsymbol{W}_1$ 和 $\boldsymbol{W}_2$ 分别表示 2 个全连接层的权重矩阵，$\sigma(\cdot)$ 为激活函数.

5）全部 Transformer 层输出. 已知模型中包含很多 Transformer 层，每一层都会经过自注意计算和前馈网络处理，假设 h_i 表示输入标记 x_i 在经过全部 Transformer 层后的最终输出，则

$$h_i = transformer(x_i). \tag{4}$$

6）并行计算. 由于输入序列的所有标记 $[x_1, x_2, \cdots, x_n]$ 是已知的，因此式（4）的计算可以并行进行：

$$\{h_1, h_2, \cdots, h_n\} = transformer([x_1, x_2, \cdots, x_n]). \tag{5}$$

7）生成第 1 个输出标记. 最后 1 个标记 x_n 的最终输出 h_n 经过线性层和 Softmax 操作后会得到第 1 个生成标记的概率分布 $P\left(x_{n+1} \mid x_1, \cdots, x_n\right)$. 根据 1.1 节提到的选择策略得到第 1 个生成的标记.

1.3 解码阶段

在解码阶段，Transformer 模型逐步生成每个新标记，并且每一个新标记的生成都是基于之前已经生成的标记. 假设当前已经生成的标记序列为 $[x_1, x_2, \cdots, x_t]$，自回归生成下一个新标记 x_{t+1} 的具体步骤有 6 步.

1)已知生成的标记序列为$[x_1, x_2, \cdots, x_t]$.

2)注意力机制只对当前生成的最后1个标记x_t计算查询向量$\boldsymbol{q}_t$、键向量$\boldsymbol{k}_t$和值向量$\boldsymbol{v}_t$:

$$\boldsymbol{q}_t = \boldsymbol{W}_q x_t,\ \boldsymbol{k}_t = \boldsymbol{W}_k x_t,\ \boldsymbol{v}_t = \boldsymbol{W}_v x_t. \tag{6}$$

注意这一步与预填充阶段有所不同,预填充阶段需要计算所有输入标记的查询、键和值向量.

3)将查询向量$\boldsymbol{q}_t$与每个已生成标记的键向量$\boldsymbol{k}_j(j=1,2,\cdots,t)$相乘,并对其进行Softmax计算得到注意力分数$a_{tj}$,在此基础上对所有值向量$\boldsymbol{v}_j(j=1,2,\cdots,t)$进行加权求和得到注意力输出$\boldsymbol{o}_t$:

$$a_{tj} = \frac{\exp\left(\frac{\boldsymbol{q}_t^{\mathrm{T}}\boldsymbol{k}_j}{\sqrt{d}}\right)}{\sum_{j=1}^{t}\exp\left(\frac{\boldsymbol{q}_t^{\mathrm{T}}\boldsymbol{k}_j}{\sqrt{d}}\right)},\ \boldsymbol{o}_t = \sum_{j=1}^{t} a_{tj}\boldsymbol{v}_j. \tag{7}$$

4)将当前已生成的注意力输出$\boldsymbol{o}_t$通过前馈网络得到当前标记的最终输出$FFN(\boldsymbol{o}_t)$:

$$FFN(\boldsymbol{o}_t) = \boldsymbol{W}_2\,\sigma(\boldsymbol{W}_1\,\boldsymbol{o}_t). \tag{8}$$

5)最后1个生成标记x_t在经过全部Transformer层处理后最终输出h_t:

$$h_t = transformer(x_t). \tag{9}$$

6)h_t经过线性层和Softmax操作后会得到第$t+1$个生成标记的概率分布$P\left(x_{t+1}|\,x_1,\cdots,x_t\right)$. 最终根据1.1节提到的选择策略生成了第$t+1$个标记$x_{t+1}$.

在解码阶段,每次生成的新标记x_{t+1}依赖于最后一个已生成标记x_t和所有已生成标记的键值缓存. 因此,新标记的生成过程只能自回归地串行计算.

1.4 预填充阶段与解码阶段特征

预填充阶段生成首个输出标记及其键值缓存,可以并行处理输入序列中的所有标记,首个生成标记的延迟受到GPU每秒浮点运算次数(floating-point operations per second, FLOPS)的限制,属于计算密集型任务. 而解码阶段以自回归的方式单次单个地逐步生成输出标记,当前的输出标记依赖于上一个标记和旧的键值缓存. 这种顺序生成过程使得工作负载受显存带宽限制,属于带宽密集型任务[14]. 由于Transformer的推理过程是自回归的,这意味着在推理过程中必须跟踪和存储先前的状态,进而增加了显存需求.

2 基于显存优化的大语言模型推理系统

在大语言模型推理过程中,GPU显存主要用于存储3类资源:模型权重、键值缓存和激活量. 以70 B参数的开源大语言模型LLaMA 2为例,其显存占用分析可以归纳为:

1)模型权重(即模型大小)主要与模型参数和精度有关. 对于使用FP16精度的LLaMA 2模型,每个参数占用2 B显存. 因此,LLaMA2-70 B模型权重将占用$70\times10^9\times2$ B = 140 GB显存空间.

2)键值缓存是指在大语言模型推理过程中,为了加速后续标记的生成效率而缓存的键和值向量. 其显存占用与请求批处理大小、序列长度、隐藏层大小、模型精度等多种因素相关. 键值缓存显存占用M_{kv}的计算公式为

$$M_{kv} = 2\times B\times S\times H\times L\times P, \tag{10}$$

其中B表示批处理大小,S表示生成标记序列的长度,H表示隐藏层大小,L表示模型的层数,P表示模型精度. 第1个系数"2"反映了需要为每个生成标记存储的键和值向量.

假设批处理大小为8,生成的序列长度为4 096,已知LLaMA2-70 B的隐藏层大小为8 192,模型层数为80,使用FP16精度. 根据式(10),LLaMA2-70 B的键值缓存将占用80 GB显存空间.

3)激活量是指每一层神经网络在前向传播过程中生成的中间结果. 与大语言模型训练不同,在推理过程中GPU显存只需要缓存当前层的中间结果,并在前向传播到下一层时释放之前的激活量,从而显著减少了显存占用.

由此可见,目前的大语言模型推理系统往往需要数十甚至上百GB的GPU显存来进行推理,而单张英伟达A100 GPU[10]的最大显存容量仅为80 GB,远低于这些需求. 显存容量不足已经成为大语言模型部署和运行的主要障碍.

本节归纳了多种大语言模型推理系统的显存优化技术:1)键值缓存管理,通过减少显存碎片来提高GPU显存利用率并避免显存浪费;2)键值缓存压缩技术,通过减少键值缓存的大小,降低GPU的显存占用;3)注意力算子优化技术,通过改进注意力机制的计算方式来减少显存占用. 这些方法缓解了大语言模型推理系统中的GPU显存不足问题,使系统能够支持更长的序列长度和更快的推理速度.

2.1 键值缓存管理

在大语言模型推理系统的显存分布中,模型权重是恒定的,激活量占比相对较小,键值缓存的大小则随着生成序列的长度而变化[15],导致其显存需求动态且难以预测. 因此,在大语言模型推理系统中高

效管理键值缓存变得尤为重要.

ORCA[16] 在调度初始请求时会根据每个请求的最大标记数预留连续的显存空间，确保在整个处理过程中不会出现显存不足的情况. 然而，在实际推理过程中，大多数请求的长度远小于其最大长度，导致严重的显存碎片. 如图 2 所示，ORCA 为请求 A 预先分配了可以存储 2 048 个标记的键值缓存，但实际上请求 A 只占用了 512 个标记的显存，包括请求 A 的输入提示标记序列、已生成的标记序列、当前正在生成的标记和未来的输出标记. 而剩余的 1 536 个标记所对应的显存空间就成为超额配置的显存内部碎片. 尽管预留的显存在请求完成采样后会被释放，但在请求处理期间，特别是当预留空间较大时，这些显存会占用本可用于处理其他请求的空间.

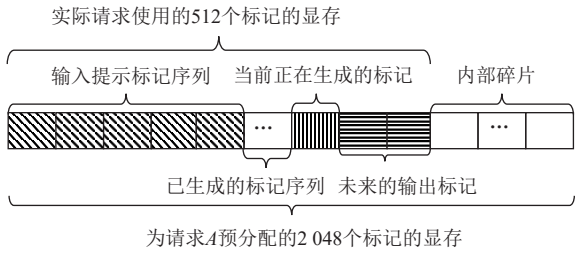


Fig. 2 Illustration of ORCA internal fragmentation[16]

图 2 ORCA 内部碎片示意图[16]

受操作系统分页虚拟内存的启发，vLLM[15] 提出了页注意力机制 PagedAttention，使用非连续的显存空间存储连续的键值缓存. 在 GPU 节点上分配一块连续的显存空间，并将其划分为若干个相同规格的物理键值块. 相应地，vLLM 也将每个请求序列的键值缓存表示为一系列逻辑键值块，类似于虚拟内存中的页面，每个块包含固定数量标记所生成的键值张量.

图 3 展示了 vLLM 的总体架构，主要包括调度器、键值缓存管理器、GPU 块分配器、CPU 块分配器以及多个工作节点. 调度器负责协调分布式 GPU 工作节点的执行；键值缓存管理器负责逻辑键值块到物理键值块的地址转换，并维护块表；GPU 块分配器和 CPU 块分配器分别管理 GPU 显存和 CPU 内存中的物理键值块. 每个工作节点根据调度器的指令执行模型推理任务，并在 GPU 显存中维护自己的模型分片和缓存引擎，在需要时与 CPU 内存中进行交换.

在预填充阶段，vLLM 仅预留必要的键值块来存储键值缓存，而无需为可能生成的最大序列长度预留显存. 如图 4 所示，生成第 1 个输出标记的键值缓存后，vLLM 将其存储在预分配的逻辑键值块中，并

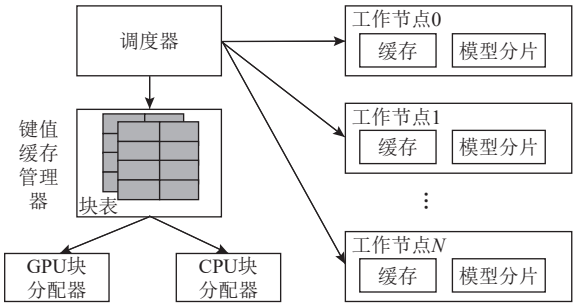


Fig. 3 Overview architecture of vLLM[15]

图 3 vLLM 总体架构[15]

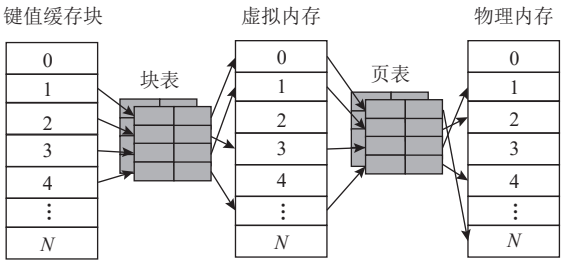


Fig. 4 Address translation from logical KV blocks in vLLM to physical memory blocks in OS[15]

图 4 vLLM 逻辑键值块到操作系统物理内存块的地址转换[15]

更新块表，记录逻辑键值块与物理键值块之间的映射关系及其填充位置的数量. 然后，虚拟内存通过页表映射到物理内存，完成从逻辑地址到物理地址的转换.

在解码过程中，随着新的标记及其对应的键值缓存不断生成，vLLM 会为每个逻辑键值块灵活分配新的物理键值块. 同时，vLLM 会跟踪每个逻辑键值块的填充位置及其所关联的物理键值块. 所有物理键值块从左往右依次填充，且只有所有以前的物理块填满的情况下才会分配下一个新块. 这种方法将每次请求产生的显存浪费限制在单个物理块内，从而有效地减少了内部碎片的产生.

Infinite-LLM[17] 在 vLLM 的基础上提出了一种可扩展的分布式注意力机制 DistAttention，它将键值缓存划分为若干个 rBlock 子块，使用 rManager 和 gManager 协调管理显存块. 如图 5 所示，Infinite-LLM 为每个大语言模型实例配备 1 个 rManager，它将本地设备上的显存空间虚拟化为固定大小的物理 rBlock. 在接收到来自本地或远程实例的显存分配请求时，rManager 会查询 rBlock 表以确定第 1 个可用的物理 rBlock. 如果没有足够空间，rManager 会启动从其他实例借用空间的程序，从而实现显存的动态分配和高效利用.gManager 作为集中管理器，维护所有实例

的全局显存信息，并构建全局债务台账. 当实例显存不足时，gManager 根据全局债务台账推荐潜在的显存借出实例，并协调显存的借用和交换. 通过优化显存分配、管理和通信，Infinite-LLM 显著提升了推理系统的吞吐量.

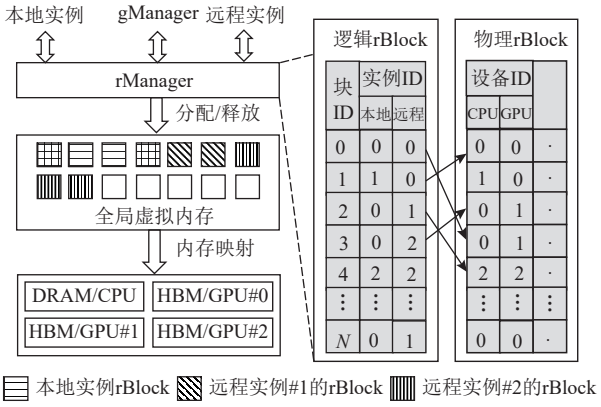


Fig. 5 Architecture of the rManager design[17]

图 5 rManager 设计架构[17]

Prabhu 等人[18]提出了一种动态键值缓存管理方法 vAttention.与 PagedAttention 不同，vAttention 主要利用系统对动态显存分配的支持，将虚拟显存和物理显存的分配分离. 得益于 CUDA 的低级 API，vAttention 预先为键值缓存分配一个大的连续虚拟显存缓冲区，但不会立即分配物理显存. 仅在运行时按需分配固定大小的物理显存页(例如 2 MB 或 64 KB). 这样既保留了虚拟显存的连续性，又避免了物理显存浪费. 此外，vAttention 通过使用较小的显存页和后台线程进行预先分配等优化策略，减少了显存碎片和分配延迟.

LightLLM[19]实现了标记级的键值缓存管理机制 TokenAttention，其主要目标是优化显存利用率并提高推理过程中的吞吐量. 如图 6 所示，它通过预分配键值缓存并创建标记表来记录输入标记的存储位置，系统在处理请求时优先分配连续显存空间，只有在连续空间不足时才分配非连续显存. 当需要缓存新生成的标记时，只需在预先分配好的键值缓存中找到未使用的空间，然后在标记表中添加相应的条目. 借助 OpenAI Triton 实现的高效操作符，TokenAttention 能够在推理过程中快速检索并计算注意力，并在请求完成后通过删除记录迅速释放显存，从而最大化 GPU 利用率并提高系统吞吐量.

表 1 总结了上述研究工作的键值缓存管理策略，比较了不同方案在显存分配粒度、显存连续性和内部碎片数量等方面的差异.ORCA 为每个请求序列预

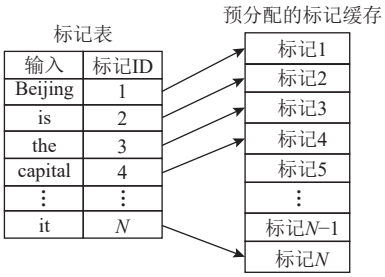


Fig. 6 Token level KV cache management of LightLLM[19]

图 6 LightLLM 的标记级键值缓存管理[19]

留了足够的连续显存空间，但这会在单个请求内部导致严重的显存碎片，进一步影响其他请求的执行效率.vLLM 和 Infinite-LLM 都使用分页显存机制来管理键值缓存，通过非连续显存分配有效缓解了内部碎片问题. 但这种显存管理机制将原本连续的请求序列离散化，需要重写注意力内核以适应键值缓存的非连续存储. 另外，vLLM 主要依赖单个节点内的 GPU 显存，而 Infinite-LLM 采用分布式方式存储键值缓存，可以优化数据中心内所有可访问的 GPU 显存.vAttention 虽然以实际的输入请求序列为单位进行虚拟显存分配，但它基于每个请求的实际需求分配若干固定大小的物理显存页，有效减少了内部碎片.LightLLM 在细粒度的标记级别管理键值缓存池. 虽然无法完全消除内部碎片，但相较于 PagedAttention 机制已经显著提高了显存利用率.

Table 1 Comparison of Large Language Model Inference Systems Using Different Key Value Cache Management Schemes

表 1 采用不同的键值缓存管理方案的大语言模型推理系统对比

系统	物理显存分配粒度	显存连续性	内部碎片
vLLM[15]	逻辑键值块	否	较少
ORCA[16]	最大请求序列	是	较多
Infinite-LLM[17]	逻辑键值块	否	较少
vAttention[18]	物理页	是	较少
LightLLM[19]	标记	是	极少

2.2 键值缓存压缩

键值缓存压缩是指在大语言模型推理过程中，通过丢弃不重要的键值缓存张量，来优化显存占用并提高计算效率的方法. 注意力机制中的键值缓存包含了模型在不同层次上生成的键和值张量，用于在推理时计算注意力分数. 然而，随着序列长度的增加，键值缓存的大小会迅速增大，导致显存占用和计算开销显著增加.

Zhang 等人[20]提出了一种键值缓存驱逐策略 H2O. 通过分析注意力矩阵的稀疏性和少数关键标记的重要性，研究人员发现仅使用 5% 的键值缓存就能解码出相同的输出标记. 累积注意力分数遵循幂律分布，也就是说，大部分重要的注意力分数集中在少数关键标记上，而这些关键标记对生成后续的标记非常重要. 基于此，H2O 在每个解码步骤动态维护包含最近和关键标记的键值缓存. 使用贪婪算法计算每个标记的注意力分数并累积，识别出对解码过程最重要的标记. 在每个解码步骤中，算法会保留这些关键标记并驱逐注意力分数最低的标记，以确保键值缓存的大小不超过预算，从而在保证解码准确性的同时大幅减少显存占用.

Quest[21]是一种查询感知的键值缓存选择算法，专为优化长上下文推理而设计. 在解码阶段，模型需要频繁加载大量键值缓存，其中只有少数标记对应的键值缓存对注意力分数有显著影响，并且这些标记的关键性会根据当前的查询张量$\boldsymbol{Q}$动态变化. Quest 算法将键值缓存分为若干个块，通过跟踪每个缓存块中键张量的最大值和最小值，并结合查询张量计算每个缓存块的关键性分数. 随后，Quest 选择关键性最高的前k个缓存块进行注意力操作. 这种方法确保 Quest 只加载和处理最重要的键值缓存部分，减少了显存移动和计算开销.

PyramidInfer[22]认为大语言模型推理过程中只有当前正在生成的最后一个标记需要利用上下文信息来推理下一个标记，而之前生成的其他标记已经完成了它们的推理任务，不再需要用于进一步的推理. 这启发 PyramidInfer 逐层选择关键的键值张量来压缩键值缓存大小.

在预填充阶段，PyramidInfer 将输入序列分成最近的部分和较远的部分. PyramidInfer 通过计算最近部分的注意力分数，找出哪些键值张量最重要并予以保留. 在浅层，保留更多的上下文信息；在深层，逐渐减少保留的信息量，这样就形成了一个逐层减少的“金字塔”形的键值缓存.

在解码阶段，PyramidInfer 维护一个滑动窗口来处理最新生成的标记. PyramidInfer 利用这些最新标记的信息更新键值缓存中的关键上下文(pivotal context, PvC). 通过控制每层中保留的 PvC 的数量，PyramidInfer 能够灵活地调整压缩比. 这种方法不仅支持无限长度的输入，还能保持一定数量的 PvC，大大节省了 GPU 的显存占用.

Keyformer[23]引入了一种动态混合注意力机制，结合了最近生成的标记和先前识别的关键标记来生成下一个标记，关键标记选择依赖于复杂的得分函数. 这种混合注意力机制具有 2 个显著优势. 首先，最近的标记通常包含最新的上下文信息，能够提供即时的语境理解. 其次，关键标记在长距离依赖中起到至关重要的作用，有助于模型在处理较长序列时保持对全局语境的理解. 然而，复杂的得分函数不可避免地带来了额外的计算开销. 例如，Keyformer 使用 Gumbel Softmax 来调整标记的概率分布，这需要在每次迭代中进行累积计算. 此外，在生成过程中，需要对每个标记重新评估其得分，增加了推理阶段的计算复杂度.

上述方案都旨在降低键值缓存的显存占用.H2O 利用累积注意力得分作为标记重要性的标准，通过删除得分较低的标记来有效减少键值缓存的大小. Quest 结合查询张量计算缓存块的关键性，只选择前k个缓存块参与注意力计算. H2O 和 Quest 主要在解码阶段进行压缩，忽略了预填充阶段的初始键值缓存. 与之不同，PyramidInfer 在预填充和解码阶段都进行压缩，不仅考虑了层间依赖性，还根据层次逐渐减少键值缓存的长度，以避免浅层过度压缩导致的信息损失. Keyformer 的关键标记选择依赖于复杂的得分函数，尽管有效，但会引入额外的计算开销和复杂性.

2.3 注意力机制算子优化

注意力算子优化主要是通过改进注意力机制的计算方式来减少显存占用. 注意力计算的显存复杂度与请求序列长度呈二次方关系[14]. 因此，加速注意力计算对于解决大语言模型在处理长序列时的存储挑战至关重要. 许多研究工作都着重于优化注意力计算机制，以减少其计算和显存需求.

如图 7 所示，注意力模块包含 3 个核心操作：1)计算查询张量和键张量的相似度$\boldsymbol{Q}\times\boldsymbol{K}$；2)对相似度进行 Softmax 操作，计算注意力权重；3)使用注意力权重对值张量$\boldsymbol{V}$进行加权求和，生成最终的输出张量. 其中执行 Softmax 操作前需要计算并存储全部张量，导致高显存占用和低并行性.

FlashAttention[24]是一种 I/O 感知的精确注意力算

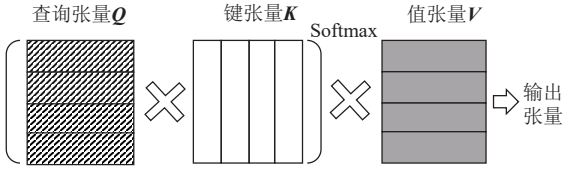


Fig. 7 Illustration of attention calculation

图 7 注意力计算示意图

法, 通过改进注意力算法的显存管理和计算流程, 实现高效的显存访问.FlashAttention 关注 GPU 显存层次之间的读写操作, 特别是减少从相对较慢的 GPU 显存到快速但小容量的片上 SRAM 的读写次数. 如图 8 所示, FlashAttention 通过平铺技术将输入张量$\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}$分割为块, 加载到片上 SRAM 中, 并在块之间逐步计算 Softmax, 在每个块输出之前对其归一化并累加起来, 最后得到正确的结果.FlashAttention 通过分块技术避免了在 GPU 显存中存储大规模的完整注意力矩阵, 实现了从二次方到线性显存利用率的优化.

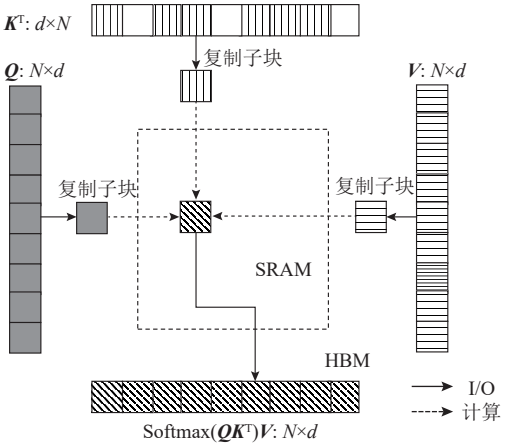


Fig. 8 Illustration of FlashAttention calculation[24]

图 8 FlashAttention 注意力计算示意图[24]

此外, FlashAttention-2[25], FlashDecoding[26], FlashDecoding++[27] 等研究工作也继续沿用了 FlashAttention 的分块技术, 减少了推理过程中对大规模注意力矩阵的存储需求. 值得一提的是, FlashDecoding++进一步减少了分块 Softmax 计算过程中额外的显存访问开销. 在传统的 Softmax 计算中, 每个部分的向量都需要找到自己的最大值来确保数值的稳定性, 这会导致每个部分之间的同步开销. FlashDecoding++引入了统一最大值取代每个部分向量的局部最大值, 从而避免不同部分 Softmax 计算之间的同步操作, 显著减少了显存访问次数.

3 基于异构存储的大语言模型推理系统

模型规模及键值缓存的增加导致大语言模型推理系统需要使用大规模 GPU 集群来满足其存储需求. 然而, 并非所有研究人员都有条件获得充足的 GPU 资源. 为了在 GPU 资源受限的环境下执行高效的推理, 一些研究工作采取异构存储技术减少对大量 GPU 资源的依赖[28-31].

异构存储技术是指利用多种存储介质(如 HBM, DRAM, SSD 等)的优势, 将大语言模型推理过程中需要的模型权重、激活量、键值缓存等张量从 GPU 显存卸载到异构存储资源. 每种存储介质具有不同的速度、容量和成本特性, 通过将模型张量在这些存储介质之间进行合理的分配和移动, 可以优化大语言模型推理系统的性能.

在现有的涉及大语言模型推理的异构存储技术中, 研究人员主要需要权衡 2 个问题.

1)如何选择合理的张量卸载策略. 在卸载过程中, 决定何时将何种类型的数据以何种粒度卸载到何处, 是影响大语言模型推理性能的关键因素. 合理的卸载策略可以有效平衡 GPU 和 CPU 之间的资源利用, 在减少 GPU 显存占用的同时优化计算效率.

2)如何降低卸载过程中带来的数据传输开销. NVLink 与 PCIe 之间的数据传输率差距巨大: NVLink 3.0[32] 的带宽为每条链路 100 GBps, 总带宽可达 600 GBps, 而 16 通道的 PCIe 5.0[33] 的总带宽仅为 128 GBps, 差距接近 5 倍. 由于 PCIe 带宽远低于 NVLink, 当 CPU 和 GPU 之间传输大规模数据时, 数据传输延迟将成为大语言模型推理的性能瓶颈.

在张量卸载策略方面, 不同的研究工作关注不同类型数据的卸载, 包括模型权重、激活量和键值缓存等. 此外, 为了实现高效的张量卸载, 还需考虑不同类型数据的卸载粒度. 卸载粒度反映了张量卸载的精细程度, 主要分为模型粒度、层粒度、张量粒度和标记粒度.

在降低数据传输开销方面, 现有的研究工作主要从 2 个角度来考虑: 1)减少 CPU 与 GPU 之间的 I/O 传输量. 这可以通过智能数据分配或预取技术实现, 即通过预测并识别大语言模型推理计算中必要的数据, 将其智能地分配或预取到 GPU 显存, 从而减少不必要的数据传输. 2)隐藏 I/O 传输延迟. 这可以采用异步数据传输技术, 通过计算和传输重叠的方式最大限度地利用硬件资源.

表 2 总结了现有的基于异构存储架构的大语言模型推理系统在存储介质、卸载数据类型、卸载粒度、优化策略和优化目标等方面的区别. 本节将依次介绍 3 部分内容: 张量卸载策略、异构并行计算与异步数据传输、智能显存分配与预取.

3.1 张量卸载策略

张量卸载是指在推理过程中, 将部分张量从 GPU 显存移出, 存储到 CPU 内存或 SSD 中, 然后在需要时再加载回来的过程. 由于 CPU 内存和 SSD 的容量

Table 2 Comparison of Large Language Model Inference Systems Based on Heterogeneous Storage Architecture

表 2 基于异构存储架构的大语言模型推理系统对比

大语言模型推理系统	存储介质	卸载数据类型	卸载粒度	优化策略	优化目标
ZeRO-Inference[34]	显存+DRAM+SSD	模型权重	层粒度	A|S	M|L
FlexGen[35]	显存+DRAM+SSD	模型权重|激活量|键值缓存	张量|元素粒度	H|A|O	M|T|L
HeteGen[36]	显存+DRAM	模型权重|激活量|键值缓存	张量粒度	H|A|O|S	M|T|L
Lamina[37]	显存+DRAM	键值缓存	标记粒度	H|A|O|S	M|T|L
FastServe[38]	显存+DRAM	键值缓存	标记粒度	A|O|S	M|T|L
InfiniGen[39]	显存+DRAM	键值缓存	标记粒度	A|O|S	M|T|L
PowerInfer[40]	显存+DRAM	模型权重	张量粒度	H|A|O|S	M|T|L
LLM in a flash[41]	显存+DRAM+SSD	模型权重|激活量	张量|标记粒度	H|A|O|S	M|T|L

注：1）不同卸载粒度的具体解释. 层粒度：以神经网络的层级为单位卸载数据；张量粒度：以张量为单位卸载数据；元素粒度：以张量中的元素为单位卸载数据（例如，将某张量中 50% 的元素分配到 CPU 内存）；标记粒度：以标记为基本单位卸载其对应的键值缓存. 2）不同优化策略的具体解释. H：异构并行计算；A：异步数据传输；O：张量卸载策略；S：智能显存分配与预取. 3）不同优化目标的具体解释. M：减少显存占用；T：减少 I/O 传输量；L：隐藏 I/O 传输延迟；符号“|”表示两者兼有.

显著超过 GPU 显存，这种卸载策略允许将 GPU 未使用的数据存储在更大容量的存储设备上.

不同的大语言模型推理系统会根据其需求、计算与 I/O 开销、硬件资源和系统优化目标等因素，选择适合的张量类型进行卸载. 这些考量有助于在特定的硬件和应用环境中实现最佳性能和资源利用. 此外，不同的张量卸载策略通过按需分割和分层存储，能够使系统更精细地控制数据在不同存储设备上的位置和时间. 此方法不仅能够灵活管理和分配显存资源，还可以在有限 GPU 显存下支持更大规模的模型和更大批量的请求，从而显著提高大语言模型推理的吞吐量.

DeepSpeed[42] 是微软开发的一个深度学习优化框架. 为了在硬件预算内提供高效的大语言模型推理服务，DeepSpeed 引入了 ZeRO-Inference[34]，将模型权重全部存储在 DRAM 或 SSD 中，并在推理过程中将每一层的权重流式传输到 GPU 显存进行推理计算. 在完成一层计算后，输出会保留在 GPU 显存中作为下一层的输入，同时释放被该层权重占用的显存以供下一层使用. 尽管通过 PCIe 传输模型权重会带来一定的延迟，但 ZeRO-Inference 仅将 1 层或几层模型权重放置在 GPU 显存中，使得大部分 GPU 显存可以存储推理过程中生成的键值缓存. ZeRO-Inference 继承了大语言模型训练系统的卸载策略，但随着大语言模型推理长上下文需求的增加，系统需要频繁访问和更新大量的激活量和键值缓存，粗粒度的模型权重卸载策略无法高效处理和分配这些额外的显存需求.

FlexGen[35] 通过将内存和计算资源从 GPU, CPU, SSD 中聚合，在单个消费级 GPU 上实现了高效的生成式推理. 为了决定何时将哪些张量卸载到何处，FlexGen 将这一问题形式化为一个图遍历问题，在此基础上提出了Z字形块调度方法来遍历图.

图 9 展示了采用Z字形调度的计算图. 假设模型有 4 层，每个输入标记生成 3 个新标记. 每个小方格表示计算层的一个 GPU 计算批次，相同颜色和底纹的小方格共享相同的层权重. FlexGen 通过将计算图分割成块来进行 Z 字形调度，每个块中包含多个批次. 纵列箭头显示了计算的 Z 字形遍历路径，按列依次向下计算，到达列底部后跳到下一列的顶部继续计算. 计算当前批次的同时，可以预先加载下一批次的权重、激活量和键值缓存，并存储前一批次的结果. 通过共享模型权重，Z 字形调度减少了频繁的权重加载和卸载，降低了 I/O 开销. 通过分块处理和批次重叠计算，最大限度地利用了 GPU, CPU, SSD 的存储和计算资源. 在卸载粒度方面，FlexGen 对模型权重采用张量粒度进行划分，对激活量和键值缓存则采用更细的元素粒度进行划分，构建了一个包含多个参数的搜索空间. 在此基础上，FlexGen 开发了一个分析成本模型，并使用线性规划优化器进行求

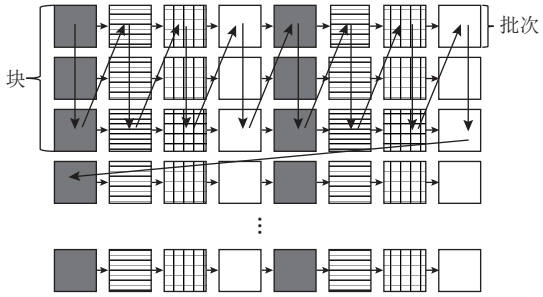


Fig. 9 Zig-zag block scheduling in FlexGen[35]

图 9 FlexGen 的 Z 字形调度[35]

解, 以确定最佳的卸载策略. FlexGen 通过统一模型权重、激活量和键值缓存的位置, 并仔细权衡不同类型数据的划分粒度, 显著提高了批处理大小的上限和系统吞吐量.

3.2 异构并行计算与异步数据传输

尽管基于异构存储的大语言模型推理系统能够提供更大的整体存储容量, 但 CPU 内存或 SSD 的吞吐量却远低于 GPU 显存. 为了降低不同存储介质间大规模数据传输的开销, 许多现有的异构存储大语言模型推理系统采用异构并行计算技术, 充分利用 CPU 和 GPU 资源. 同时, 这些系统通过异步数据传输机制, 将计算与传输过程重叠, 从而有效隐藏 I/O 延迟.

由于解码阶段注意力机制的瓶颈在于 I/O, 当关联的键值缓存被卸载到 CPU 上时, FlexGen 会直接在 CPU 上计算注意力分数. 图 10 展示了 FlexGen 在注意力计算模块中的异构并行架构. 首先, 激活量从 GPU 传输到 CPU, 接着 CPU 开始计算注意力分数. 与此同时, GPU 异步将模型权重传输到 CPU, 提前为下一步计算准备好权重. 一旦 CPU 完成计算, 结果会被发送回 GPU. GPU 接收到结果后, 继续进行下一层的计算或生成最终的模型输出. 由于键值缓存占用的显存远高于激活量, 这种方法大大减少了 CPU 与 GPU 之间的数据传输量.

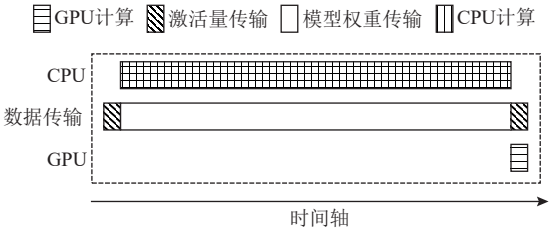


Fig. 10 Illustration of heterogeneous parallel computing architecture in FlexGen attention module [35]

图 10 FlexGen 注意力模块异构并行计算架构示意图[35]

HeteGen[36] 认为, 尽管 FlexGen 在一定程度上利用了 CPU 和 I/O 资源, 但其利用率仍然有限, 且无法有效减少稀疏输入的延迟. 通过仔细分析大语言模型推理中的预填充阶段和解码阶段的显存需求, HeteGen 发现线性模块(包括注意力机制中的线性变换和前馈网络中的线性变换)占据了大部分 GPU 显存, 是 GPU 显存优化的关键. 因此, HeteGen 重点对线性模块进行异构并行计算和异步数据传输, 以降低 CPU 和 GPU 之间的 I/O 传输需求.

HeteGen 引入了一种专门用于通信的并行形式, 将注意力模块异构并行计算的数据传输操作分为 2 个部分: 固定内存和内存传输, 这 2 个部分通过多线程同时进行. 固定内存是指在操作系统层面上对内存进行锁定, 使其不会被交换到磁盘或被其他进程占用. 在需要进行频繁数据传输时, 这会大大提高内存访问速度. 如图 11 所示, 这种内存操作级别的并行需要预先固定当前的模型权重, 然后准备将固定的权重进行传输. 在当前权重传输的同时, 主动固定下一层中的模型权重.

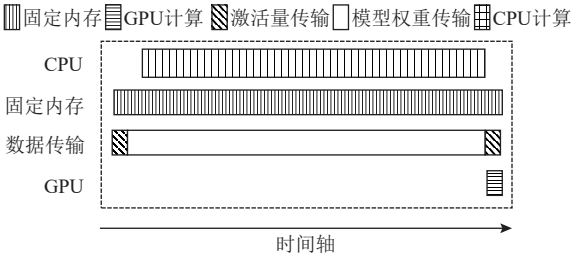


Fig. 11 Illustration of heterogeneous parallel computing architecture in HeteGen attention module [36]

图 11 HeteGen 注意力模块异构并行计算架构示意图[36]

在异构并行计算的基础上, HeteGen 应用了异步数据传输技术. 通过多线程同时进行固定内存和内存传输操作, 系统能够高效利用 I/O 资源. 在使用相同 GPU 显存资源的情况下, HeteGen 表现出比 FlexGen 更高的吞吐量, 并且可以在卸载过程中更灵活地调整 GPU 显存和 CPU 内存的比例.

Lamina[37] 专注于通过注意力卸载机制实现显存优化. 其核心思想是将显存密集型任务与计算密集型任务分离, 以减少显存占用并提高大语言模型推理的效率. 显存优化设备用于存储键值缓存并执行注意力计算, 而计算优化设备用于存储模型权重并计算模型的其他部分. 此外, Lamina 还采用了微批次处理, 在处理较大批次的请求时, 将其划分为更小的微批次进行处理. 这种方法使得每个微批次可以有效地管理键值缓存的占用, 从而避免大量显存被单个任务占满. 而且, 在计算当前微批次时, 下一微批次的数据传输可以并行进行, 进一步提高了系统响应速度.

FastServe[38] 的核心设计包含了跳过-加入多级反馈队列(skip-join multi-level feedback queue, Skip-Join MLFQ)调度器和主动键值缓存交换机制. Skip-Join MLFQ 调度器使用迭代级抢占调度来减少长作业导致的排队延迟, 但为了提高性能, 它需要更多的 GPU 显存资源. 为此, FastServe 引入了主动键值缓存交换机制有效管理显存的使用, 确保高优先级任务能够及时获取所需资源. 如图 12 所示, FastServe 通过分布式存储将键值缓存分配到多个 GPU 上进行管理. 键

值缓存管理器负责协调 GPU 之间的交换操作，决定何时卸载或上传键值缓存. 当非活跃作业不再需要 GPU 显存时，键值缓存管理器会将其键值缓存预先卸载到 CPU 内存，腾出显存空间供高优先级任务使用. 当 GPU 资源恢复可用时，系统会预加载即将调度作业的键值缓存至 GPU，这一过程与大语言模型推理的解码阶段并行执行，从而隐藏了 I/O 传输延迟. 此外，大语言模型推理过程中的不同阶段会将键值张量存储在相应的 GPU 上，使每个 GPU 只需处理本地的键值张量，减少跨设备的数据传输.

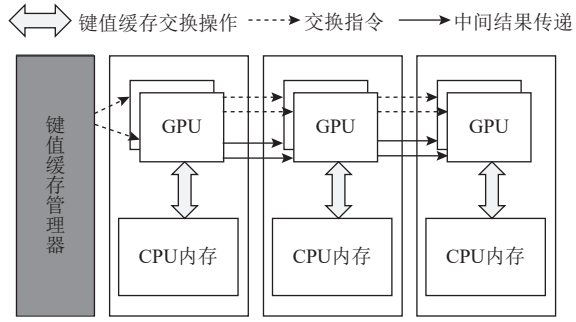


Fig. 12 Overlapping key value cache offloading with intermediate result transmission[38]

图 12 键值缓存卸载与中间结果传递重叠[38]

3.3 智能显存分配与预取

现有的许多研究在离线阶段分析数据的重要性，根据 GPU 的计算需求将关键数据分配或预取到 GPU 显存. 这意味着在大语言模型推理过程中，无需将所有数据传输到 GPU，只需传输必要的数据. 通过智能显存分配和预取，大语言模型推理系统能够有效降低推理过程中的不必要数据传输，减少 PCIe 带宽的浪费.

InfiniGen[39] 是第 1 个通过仅预取大语言模型推理系统中必要的键值缓存条目来实现高效推理的研究工作. 它将键值缓存卸载到 CPU 内存，并在 CPU 内存中维护一个键值缓存池，仅将必要的部分预取到 GPU.InfiniGen 利用前一层 Transformer 层的注意力输入来推测和预取当前层重要标记的键和值张量.

如图 13 所示，在离线阶段，InfiniGen 通过运行一次前向传播来收集每层的查询张量，然后对这些查询张量进行奇异值分解（SVD）以获得偏斜矩阵. 接着，将偏斜矩阵与对应层的查询权重矩阵和键权重矩阵相乘，确保权重矩阵的维度不变. 改进后的权重矩阵使得查询、键、值张量在不同输入下表现出高度偏斜性. 在预填充阶段，InfiniGen 从查询权重矩阵和键权重矩阵中选择几个重要的列，作为在解码阶段使用的查询张量和键张量的部分权重矩阵. 在解码阶段，使用前一层的部分查询权重矩阵和部分键权重矩阵进行注意力计算，并选择具有高注意力得分的标记. 对注意力计算的结果进行进一步的非线性变换，以生成下一层的输入.InfiniGen 通过智能的键值缓存管理方案，有效减少了从 CPU 到 GPU 的数据传输量. 在保持大语言模型推理准确性的同时，大大提高了推理系统的性能.

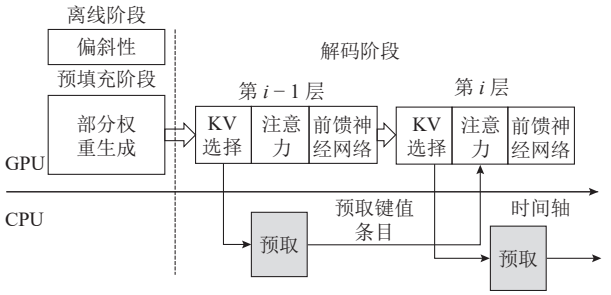


Fig. 13 Overview architecture of InfiniGen[39]

图 13 InfiniGen 的总体架构[39]

PowerInfer[40] 是一种在仅配备单个消费级 GPU 的计算机上运行的大语言模型推理系统. PowerInfer 利用大语言模型推理中的高局部性，将频繁激活的神经元分配到 GPU，而将较少激活的神经元驻留在 CPU. 如图 14 所示，PowerInfer 架构分为离线和在线 2 个阶段：离线组件通过剖析大语言模型的激活量，计算每个神经元的影响度量，并利用整数线性规划求解器将神经元分类为热神经元或冷神经元，生成最优的神经元放置策略；在线组件则创建并行的 GPU 和 CPU 工作线程，分别执行各自的计算任务，在计算完成后将结果合并到 GPU 中，优化计算工作负载和传输效率. 此外，在线组件采用神经元感知的稀疏操作符，直接在 GPU 和 CPU 上计算活跃神经元及其权重，避免传统稀疏矩阵乘法工具的性能开销.

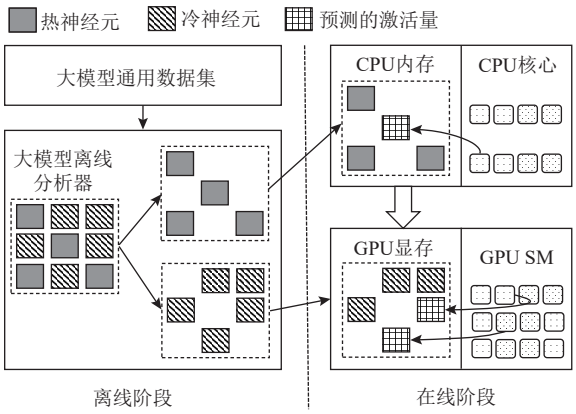


Fig. 14 Overview architecture of the PowerInfer[40]

图 14 PowerInfer 的总体架构[40]

Alizadeh 等人[41]提出了一种利用异构存储技术在显存受限设备上进行高效大语言模型推理的方法. 具体而言, 将模型参数存储在容量较大的闪存中, 并根据需要动态加载到 CPU 内存. 为优化从闪存到 CPU 内存的数据传输, 采用选择性持久策略, 将嵌入矩阵和注意力机制的矩阵永久保留在 CPU 内存中, 仅动态加载前馈网络的非稀疏部分, 并使用低秩预测器识别被 ReLU 激活函数归零的元素, 避免加载无效数据. 此外, 通过滑动窗口技术保留和重用最近激活的神经元, 进一步节省了内存空间. 最后, 将前馈网络层的行和列数据捆绑在一起, 以更大的块粒度读取数据, 从而显著提高了闪存读取的吞吐量. 该方法特别适用于模型无法完全加载到内存中的情况, 在硬件感知和选择性加载方面具有显著优势.

InfiniGen 关注大语言模型推理中长序列生成的键值缓存管理, 通过推测方法只加载关键的键值缓存条目, 并利用 CPU 内存的大容量来管理键值缓存池. PowerInfer 则利用神经元激活的局部性和幂律分布, 仅将频繁激活的神经元分配到 GPU, 以减少 GPU 显存需求. Alizadeh 等人[41]则利用闪存硬件特性, 通过窗口技术和行-列捆绑来增加读取数据块的大小, 减少了 I/O 操作的次数. 这些研究都有效降低了异构存储设备之间的数据传输量, 提高了大语言模型的推理效率.

4 基于分布式存储的大语言模型推理系统

分布式大语言模型推理系统不仅需要具备高效处理海量请求的能力, 还需深入探索如何充分利用分布式计算和存储资源、完善容错机制, 以及降低推理任务延迟等关键问题. 为此, 许多研究工作正致力于解决这些问题, 以进一步提升系统的性能和可靠性. 本节首先介绍了现有的批处理技术如何优化分布式资源的利用和分布式大语言模型的并行处理能力; 接着介绍了分布式存储环境下大语言模型的数据冗余与故障恢复技术; 最后分析了无服务推理中的检查点加载技术, 以应对无服务推理过程中的模型冷启动问题.

4.1 分布式大语言模型推理的批处理技术

大语言模型推理的批处理技术通过收集一批用户请求, 将这些请求的预填充和解码步骤进行批量处理, 从而减少 GPU 空闲时间和 I/O 传输开销. 通常, 在大语言模型推理服务场景中, 用户请求具有异步性和多样性, 处理这些请求的时间和资源需求存在显著差异. 因此, 许多研究工作[15-16]提出了连续批处理策略, 即在一些旧请求完成后, 立即批处理新的请求. 这种方法可以有效减少 GPU 的空闲时间, 提高整体系统的吞吐量, 但会增加较短请求的延迟.

为了进一步增强连续批处理, 一些研究工作[43-45]采用分块预填充与附带策略, 将较长的预填充请求在序列维度上分割成块, 并将预填充块与几个解码请求一起批处理(即附带). 具体地, 每个预填充块在独立的前向传播中处理, 而解码操作在最后 1 个块处理完成后就立即进行. 长输入提示序列不再需要长时间的单次前向传递来处理, 而是分解成多个较短的传递, 这样可以在较短时间内完成初始处理, 提高整体响应速度.

在分布式大语言模型推理系统中, 将预填充和解码阶段放在同一个机器上执行不可避免地会导致资源共享问题. 为了更有效地利用分布式计算和存储资源, 最近的研究提出了预填充与解码阶段分离的策略. 大语言模型推理的每个阶段都有其独特的计算特性和延迟要求. 具体来说, 预填充阶段是计算密集型的, 而解码阶段则是内存密集型的. 通过将预填充和解码阶段分别放置在不同实例上并针对各自的特性进行资源优化, 可以显著提升分布式大语言模型推理系统的整体性能和效率.

表 3 总结了不同的分布式大语言模型推理系统在批处理策略、并行策略、键值缓存传输方式和调度级别等方面的差异. 这些差异反映了各分布式系统在应对大语言模型的推理挑战时所采取的不同技术路径和权衡.

Mooncake[46]是 Moonshot AI 提供 Kimi 服务的平台. 如图 15 所示, 为了充分利用 GPU 集群中可用的各种资源, 它将 GPU 集群解耦并重构为多个分离的资源池, 分别执行预填充和解码任务. Mooncake 使用全局调度器选择合适的预填充和解码实例. 在预填充阶段, 它采用分块或分层的方式处理请求, 确保每个部分高效运行. 预填充阶段生成的键值缓存会持续流式传输到解码实例, 以便解码过程可以顺利进行. 另外, Mooncake 采用前缀匹配和缓存迁移技术重用键值缓存. 在处理新请求时, 全局调度器会检查是否有可重用的键值缓存块, 如果请求的前缀与现有缓存匹配, 就重用这些缓存数据, 避免重复计算. 为了应对过载场景, Mooncake 采用了基于预测的提前拒绝策略, 通过预测预填充和解码阶段的负载, 提前拒绝那些无法在服务水平目标(service-level objectives, SLOs)内完成的请求, 从而减少计算资源浪费. 这一

Table 3 Comparison of Different Distributed Large Language Model Inference Systems

表 3 不同分布式大语言模型推理系统的对比

系统	批处理策略	并行策略	键值缓存传输	调度级别
Sarathi[43]	分块预填充与附带	流水线|张量并行	同步传输	迭代级调度
DeepSpeed-FastGen[44]	分块预填充与附带	流水线并行	异步传输	迭代级|标记级调度
Mooncake[46]	预填充与解码分离	流水线|张量并行	异步传输	请求级|迭代级调度
DistServe[47]	预填充与解码分离	流水线|张量并行	异步传输	请求级|迭代级调度
TetriInfer[48]	预填充与解码分离	流水线并行	异步传输	请求级|迭代级调度
Splitwise[49]	预填充与解码分离	流水线|张量并行	异步传输	请求级|迭代级调度
DéjàVu[50]	预填充与解码分离	流水线|张量并行	异步传输	请求级|迭代级调度

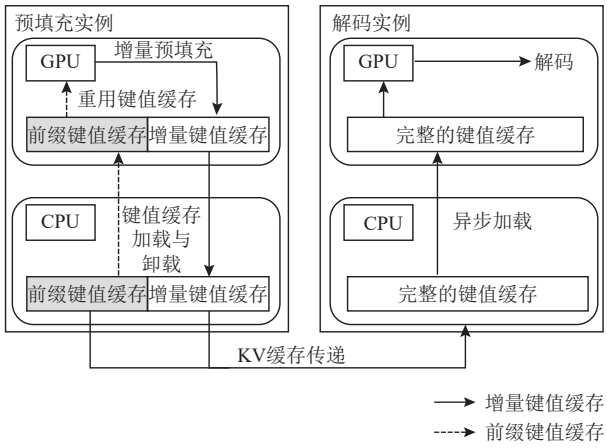


Fig. 15 Workflow of Mooncake's inference instances[46]

图 15 Mooncake 的推理实例工作流[46]

策略通过负载预测和双重检查，有效平衡了预填充和解码实例之间的负载，确保大语言模型推理系统在高负载情况下仍能高效运行.

DistServe[47] 通过将预填充阶段和解码阶段分别分配到不同的 GPU 上，消除了资源竞争和干扰. DistServe 根据大语言模型的预填充阶段的首个标记时间(time to first token, TTFT)和每个输出标记时间(time per output token, TPOT)的要求，优化每个阶段的资源分配和并行策略. 通过高效放置算法将实例按层划分成多个阶段，每个阶段在同一节点内运行，并在每个节点内寻找最优的操作内和操作间并行配置. 然后利用模拟器评估配置的 SLOs 达成率，形成全局最优放置方案. 操作内并行通过将计算任务分割并分配到多个 GPU 上，而操作间并行则将模型的不同部分分配到不同的 GPU 执行. 在 DistServe 中，模型实例被划分为多个阶段，并在多个节点之间进行分布式放置. 每个节点负责特定阶段的执行，并通过高效的通信机制(如 NVLink)在节点之间传递中间结果，从而显著提高了分布式大语言模型推理系统的计算能力和吞吐量.

TetriInfer[48] 将预填充和解码阶段分配到独立的实例中运行，预填充实例处理输入提示并生成键值缓存，解码实例使用这些缓存逐步生成输出标记. 为了实现不同解码实例之间的负载均衡，TetriInfer 利用小型模型预测解码阶段可能生成的标记长度，以估计解码阶段的计算和存储资源需求. 调度器基于这些预测信息动态调整解码实例的负载，进而提高了解码阶段的效率. 此外，不同的预填充和解码实例不仅可以独立扩展，而且可以根据实例的负载变化进行角色互换，确保系统在资源使用方面更加灵活和高效.

Splitwise[49] 是一种将大语言模型推理请求的 2 个阶段分拆到不同机器上的模型部署技术. Splitwise 维护了 3 个机器池：提示处理池、标记生成池和混合池，并使用分层的 2 级调度灵活调整 GPU 集群资源. 集群级调度器主要负责机器池管理和推理请求路由. 其中，机器池管理根据预期负载将机器分配到不同的池中，并在高负载情况下动态调整池的大小和组成；推理请求路由使用“最短队列优先”策略为每个请求分配提示处理机和标记生成机，当集群负载不平衡时(比如某一池的任务过多，另一个池的任务较少)，可以将部分机器临时移入混合池，同时执行提示处理和标记这 2 种任务. 机器级调度负责管理每台机器上的具体任务，使用先来先服务策略进行请求调度，尽量多地进行批处理，并在显存接近用完时开始排队. Splitwise 的 2 级调度策略可以帮助分布式大语言模型推理系统根据当前任务需求动态调整资源配置，优化 GPU 集群的资源利用率.

上述研究工作都采用了预填充与解码阶段分离的方案来提高批处理上限，都强调根据具体需求进行资源优化配置. Mooncake 利用键值缓存全局调度器和前缀匹配技术，实现预填充和解码阶段的高效资源分配和键值缓存重用. DistServe 根据模型特征

和模拟结果, 采用高效的放置算法和灵活的模型并行策略提高推理性能. TetriInfer 通过预测解码请求的资源使用情况来平衡不同解码实例之间的负载. Splitwise 则通过 2 级调度最大化批处理数量并合理分配计算和存储资源. 值得一提的是, 分布式存储在上述系统中都起到了核心作用, 通过在多个节点之间动态分配和共享资源, 优化计算与存储的协作方式, 不仅提高了大语言模型推理系统的并行处理能力, 还减少了资源干扰和浪费.

4.2 分布式大语言模型推理系统的容错技术

在分布式大语言模型推理[51]过程中, 故障处理是一个关键问题. 如果某个节点出现故障, 通常需要重新计算整个请求, 这会引入显著的延迟开销. 数据冗余与复制[52-55]是最为常见的容错技术, 将数据和模型权重复制到多个节点上, 以确保在单个节点出现故障时, 其他节点仍然可以提供数据和计算服务.

DéjàVu[50]使用键值缓存流库(DéjàVuLib)来优化流水线并行的分布式大语言模型推理服务. 在流水线并行中, 模型的各层被划分为多个阶段, 阶段之间通过交换激活值协同工作. 为了更好地填充流水线, 使其所有阶段保持忙碌, DéjàVu 将一个大批次请求拆分为更小的微批次, 使得多个微批次能够同时在流水线的不同阶段中并行处理.

图 16 展示了 DéjàVu 系统的流水线故障恢复机制, 该系统通过状态复制和快速故障恢复来防止单点故障导致的数据不一致. 首先, DéjàVu 在不同的 GPU 节点间复制键值缓存, 例如, 工作节点 x 会将其键值缓存传输到节点 $(x+1)\%N$, 其中 N 表示工作节点总数目. 这样即使某个节点出现故障, 也能通过其他节点的缓存来恢复数据. 如图 16 所示, 节点 F 处于故障状态, 故障节点之后的下一个节点将其托管的键值缓存备份发送回故障节点, 以重新填充丢失的缓存; 同时, 故障节点之前的上一个节点也会将其自身的键值缓存发送给故障节点, 以恢复其复制的键值缓存, 这一过程由控制器在故障检测阶段 D 触发并协调. 接下来, 控制器根据缓存复制状态, 识别出需要重新执行的微批次和步骤 C1. 在进入故障恢复阶段 R 后, 控制器将识别出的微批次和步骤信息传播到所有工作节点, 使所有阶段从 C1 重新开始推理, 无需从头开始计算. 通过 F, D, R 状态的紧密协同, DéjàVu 系统能够从最近的复制点恢复推理, 有效检测、处理和恢复流水线中的故障, 确保最小化停机时间和资源浪费.

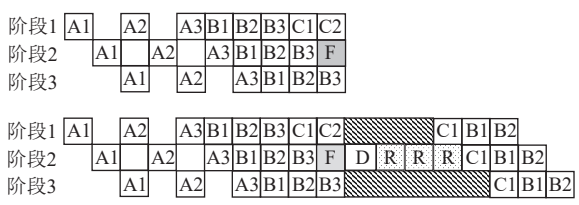


Fig. 16 Pipeline failure recovery for DéjàVu[50]

图 16 DéjàVu 的流水线故障恢复[50]

4.3 无服务推理的检查点加载技术

无服务推理是一种无需管理底层服务器或基础设施的模型部署方法. 开发者将模型检查点上传至云平台, 如 Amazon SageMaker[56], Azure[57], KServe[58], HuggingFace[28] 等, 系统会根据请求动态分配 GPU 资源来执行模型推理任务. 无服务推理的主要问题是在加载大语言模型检查点时, 会导致显著的延迟开销.

ServerlessLLM[59]是一个在无服务环境中进行大语言模型推理的系统. 图 17 展示了 SeverlessLLM 的总体架构. 在该系统中, 请求路由器接收推理请求并将其路由到合适的节点, 而模型加载调度器则负责分配节点以启动新的大语言模型实例. 模型存储用于存储大语言模型检查点, 当节点启动新的大语言模型实例时, 从模型存储中加载模型. 图 17 中展示了 2 个节点: 节点 1(热节点)包含一个正在运行的大语言模型实例#1; 节点 2(冷节点)包含一个尚未分配任务的大语言模型实例#2. 为了缓解无服务环境中的冷启动问题, ServerlessLLM 需要增强大语言模型推理的本地性, 快速加载和卸载模型检查点.

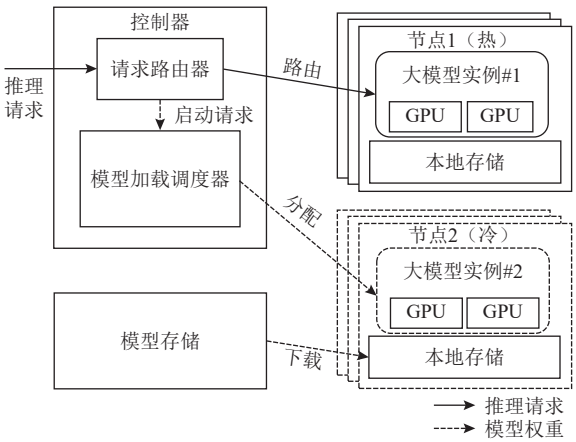


Fig. 17 Overview architecture of SeverlessLLM[59]

图 17 SeverlessLLM 的总体架构[59]

SeverlessLLM 通过引入优化的检查点格式和高效的多层次存储系统来提升检查点加载性能. 优化的检查点格式支持顺序块式读取和张量寻址. 多层次存储系统通过内存池化、高效的数据路径和多阶段数据加载管道进一步提升性能. 内存池利用并行 PCIe 链路实现多个 GPU 的并行检查点加载, 支持细粒度的内存分配和释放, 并使用固定大小的内存块

缓解内部碎片. 高效数据路径通过直接文件访问和锁页内存技术, 消除 DRAM 与 GPU 之间的冗余数据复制. 多阶段数据加载管道支持 NVMe[60], SATA[61], S3[62] 等多种存储接口, 可以充分利用 GPU 服务器的多层次存储资源[63-64].

另外, 为了优化模型启动时间和推理性能, ServerlessLLM 采用实时迁移策略实现局部性驱动的推理服务. 实时迁移会优先选择局部性最优的服务器, 并避免中断正在进行的推理服务. 例如, 在另一台服务器上存在模型检查点的副本, 便可以直接在那里启动模型, 然后传输所有当前标记. 该方法避免了大量的快照创建和传输开销.

5 总结与展望

当前, 大语言模型推理系统面临着存储资源需求量大和资源利用率低下的问题. 本文从显存管理、异构存储和分布式存储等方面, 详细总结了大语言模型推理系统中的存储优化技术. 这些技术从不同角度缓解了大语言模型的存储瓶颈, 提升了大语言模型推理系统的整体性能.

尽管这些技术在现有硬件上已有显著进展, 未来大语言模型的性能提升仍将受到硬件发展和算法优化的持续影响. 探索新型硬件以进一步加速推理过程是一个重要方向. 例如, 可以考虑将显存无法容纳的数据卸载到 CXL(compute express link)[65] 扩展内存池中, 或将计算任务卸载到计算存储设备(computational storage device, CSD)[66] 等新型硬件上. 此外, 边缘设备在实时性和隐私保护方面具有独特优势, 在边缘设备上部署大语言模型推理系统也值得关注. 最后, 如何在这些设备上实现高效的模型推理, 同时平衡推理性能与计算、存储成本[67] 也是研究人员需要考虑的问题.

作者贡献声明: 葛旭冉、欧洋, 负责论文主要内容撰写、论文组织架构及内容修改; 王博、赵宇补充文献资料, 并协助修改论文内容; 吴利舟、王子聪、陈志广、肖侬提出文章指导意见并修改论文. 葛旭冉、欧洋对论文的贡献相同.

参 考 文 献

[1] OpenAI. GPT-4 technical report[J]. arXiv preprint, arXiv: 2303.08774, 2023

[2] Zhang Shuai, Roller S, Goyal N, et al. OPT: Open pre-trained transformer language models[J]. arXiv preprint, arXiv: 2205.01068, 2022

[3] Almazrouei E, Alobeidli H, Alshamsi A, et al. Falcon-40B: An open large language model with state-of-the-art performance[J]. arXiv preprint, arXiv: 2306.01116, 2023

[4] Touvron H, Lavril T, Izacard G, et al. LLaMA: Open and efficient foundation language models[J]. arXiv preprint, arXiv: 2302.13971, 2023

[5] GitHub. Quantifying the productivity benefits of GitHub Copilot [EB/OL]. [2024-07-18]. https://github.com/features/copilot

[6] Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code[J]. arXiv preprint, arXiv: 2107.03374, 2023

[7] Du Jiangsu, Liu Ziming, Fang Jiarui, et al. EnergonAI: An inference system for 10-100 billion parameter transformer models[J]. arXiv preprint, arXiv: 2209.02341, 2022

[8] Villalobos P, Sevilla J, Besiroglu T, et al. Machine learning model sizes and the parameter gap[J]. arXiv preprint, arXiv: 2207.02852, 2022

[9] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners [C]//Proc of the 34th Conf on Neural Information Processing Systems. Red Hook, NY: Curran Associates Inc, 2020: 1877-1901

[10] NVIDIA. NVIDIA doubles down: Announces A100 80GB GPU, supercharging world's most powerful GPU for AI supercomputing [EB/OL]. 2023[2024-07-18]. https://nvidianews.nvidia.com/news/nvidia-doubles-down-announces-a100-80gb-gpu-supercharging-worlds-most-powerful-gpu-for-ai-supercomputing

[11] Devoto A, Zhao Yu, Scardapane S, et al. A simple and effective L2 norm-based strategy for KV cache compression[J]. arXiv preprint, arXiv: 2406.11430, 2024

[12] Zhao Yanli, Gu A, Varma R, et al. PyTorch FSDP: Experiences on scaling fully sharded data parallel[J]. arXiv preprint, arXiv: 2304.11277, 2023

[13] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[C]//Proc of the 31st Conf on Neural Information Processing Systems. Red Hook, NY: Curran Associates Inc, 2017: 5998-6008

[14] Fu Yao. Challenges in deploying long-context transformers: A theoretical peak performance analysis[J]. arXiv preprint, arXiv: 2405.08944, 2024

[15] Kwon W, Li Zhuohan, Zhuang Siyuan, et al. Efficient memory management for large language model serving with PagedAttention [C]//Proc of the 29th ACM SIGOPS Symp on Operating Systems Principles. New York: ACM, 2023: 611-626

[16] Yu G, Jeong J, Kim G, et al. ORCA: A distributed serving system for transformer-based generative models[C]//Proc of the 16th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2022: 521-538

[17] Lin Biao, Peng Ting, Zhang Chen, et al. Infinite-LLM: Efficient LLM service for long context with DistAttention and distributed KVCache[J]. arXiv preprint, arXiv: 2401.02669, 2024

[18] Prabhu R, Nayak A, Mohan J, et al. vAttention: Dynamic memory management for serving LLMs without PagedAttention[J]. arXiv

preprint, arXiv: 2405.04437, 2024

[19] Lin Ruihao, Peng Ting, Zhang Chen, et al. LightLLM: Efficient inference and serving framework for large language models[EB/OL]. [2024-07-18]. https://github.com/ModelTC/lightllm

[20] Zhang Zhenyu, Sheng Ying, Zhou Tianyi, et al. H2O: Heavy-hitter oracle for efficient generative inference of large language models[C]//Proc of the 37th Conf on Neural Information Processing Systems. Red Hook, NY: Curran Associates Inc, 2023: 1234−1245

[21] Tang Jiaming, Zhao Yilong, Zhu Kan, et al. Quest: Query-aware sparsity for efficient long-context LLM inference[C]//Proc of the 41st Int Conf on Machine Learning. New York: ICML, 2024: 123−134

[22] An Chenxin, Gong Shanshan, Zhong Ming, et al. PyramidInfer: Efficient long-context inference for large language models[C]//Proc of the 38th Int Conf on Machine Learning. New York: ICML, 2024: 567−579

[23] Adnan M, Arunkumar A, Jain G, et al. Keyformer: KV cache reduction through key tokens selection for efficient generative inference[C/OL]//Proc of the 6th Conf on Machine Learning and Systems. 2024[2024-07-19]. https://proceedings.mlsys.org/paper_files/paper/2024/file/48fecef47b19fe501d27d338b6d52582-Paper-Conference.pdf

[24] Dao T, Fu D, Ermon S, et al. FlashAttention: Fast and memory-efficient exact attention with IO-awareness[C]//Proc of the 36th Conf on Neural Information Processing Systems. Red Hook, NY: Curran Associates Inc, 2022: 67−78

[25] Dao T, Ge Yuhao, Fu D, et al. FlashAttentio-2: Faster attention with better parallelism and work partitioning[J]. arXiv preprint, arXiv: 2307.08691, 2023

[26] Dao T, Haziza D, Massa F, Sizov G, et at. Flash-decoding for long-context inference [EB/OL]. 2023 [2024-01-27]. https://crfm.stanford.edu/2023/10/12/ffashdecoding. html

[27] Ke Hong, Dai Guohao, Xu Jiaming, et al. FlashDecoding++: Faster large language model inference on GPUs[C/OL]//Proc of the 7th Conf on Machine Learning and Systems. 2024[2024-07-18]. https://proceedings.mlsys.org/paper_files/paper/2024/file/5321b1dabcd2be188d796c21b733e8c7-Paper-Conference.pdf

[28] HuggingFace. Transformers: State-of-the-art machine learning for pytorch, tensorflow, and JAX[EB/OL]. 2024[2024-07-18]. https://github.com/hugging face /transformers

[29] Aminabadi R, Rajbhandari S, Awan A, et al. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale[C/OL]//Proc of the 22nd Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2022[2024-07-18]. https://dl.acm.org/doi/abs/10.5555/3571885.3571946

[30] NVIDIA. Fastertransformer: About transformer related optimization, including BERT, GPT[EB/OL]. 2017[2024-07-18]. https://github.com/NVIDIA/FasterTransformer

[31] Jin Yunho, Wu Chunfeng, Brooks D, et al. S3: Increasing GPU utilization during generative inference for higher throughput[C]//Proc of the 37th Int Conf on Neural Information Processing Systems. Red Hook, NY: Curran Associates Inc, 2024: 18015−18027

[32] NVIDIA. NVLink & NVSwitch: Fastest HPC data center platform [EB/OL]. 2023[2024-07-18]. https://www.nvidia.com/en-sg/data-center/nvlink/

[33] Electronicshub. PCIe 5.0 vs PCIe 4.0: Which is best for high data transfer? [EB/OL]. 2023[2024-07-18]. https:// www .electronicshub.org/pcie-5- vs-pcie-4

[34] Github. ZeRO-Inference: Democratizing massive model inference DeepSpeed[EB/OL]. 2022[2024-07-15]. https://www.deepspeed.ai/2022/09/09/zero-inference.html

[35] Sheng Ying, Zheng Lianmin, Yuan Binhang, et al. FlexGen: High-throughput generative inference of large language models with a single GPU[C]//Proc of the 40th Int Conf on Machine Learning. New York: ICML, 2023: 31094−31116

[36] Zhao Xuanlei, Jia Bin, Zhou Haotian, et al. HeteGen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices[C]//Proc of Machine Learning and Systems. New York: PMLR, 2024: 162−172

[37] Wang Lei, Li Wei, Chen Jian, et al. Efficient and economic large language model inference with attention offloading[J]. arXiv preprint, arXiv: 2405.01814, 2024

[38] Wu Bingyang, Zhong Yinmin, Zhang Zili, et al. Fast distributed inference serving for large language models[C]//Proc of the 40th Int Conf on Machine Learning. New York: PMLR, 2023: 1234−1245

[39] Lee W, Lee J, Seo J, et al. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management[C]//Proc of the 18th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2024: 567−579

[40] Song Yixin, Mi Zeyu, Xie Haotong, et al. PowerInfer: Fast large language model serving with a consumer-grade GPU[C]//Proc of the 18th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2023: 123−134

[41] Alizadeh K, Mirzadeh I, Belenko D, et al. LLM in a flash: Efficient large language model inference with limited memory[C]//Proc of Int Conf on Machine Learning. New York: ICML, 2023: 123−134

[42] Microsoft.DeepSpeed[EB/OL].2023[2024-07-15].https://github.com/microsoft/DeepSpeed

[43] Agrawal A, Panwar A, Mohan J, et al. Sarathi: Efficient LLM inference by piggybacking decodes with chunked prefills[J]. arXiv preprint, arXiv: 2308.16369, 2023

[44] Holmes C, Tanaka M, Wyatt M, et al. DeepSpeed-FastGen: High-throughput text generation for LLMs via MII and DeepSpeed-Inference[J]. arXiv preprint, arXiv: 2401.08671, 2024

[45] Agrawal A, Kedia N, Panwar A, et al. Taming throughput-latency tradeoff in LLM inference with sarathi-serve[J]. arXiv preprint, arXiv: 2403.02310, 2024

[46] Qin Ruoyu, Li Zheming, He Weiran, et al. Mooncake: A KVCache-centric disaggregated architecture for LLM serving[J]. arXiv preprint, arXiv: 2407.00079, 2024

[47] Zhong Yinmin, Liu Shengyu, Chen Junda, et al. DistServe: Disaggregating prefill and decoding for goodput-optimized large

language model serving[C]//Proc of the 18th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2024: 193−210

[48] Hu Cunchen, Huang Heyang, Xu Liangliang, et al. Inference without interference: Disaggregate LLM inference for mixed downstream workloads[C]//Proc of the 18th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2024: 193−210

[49] Goiri Í, Maleki S, Bianchini R, et al. Splitwise: Efficient generative LLM inference using phase splitting[C]//Proc of the 28th ACM Symp on Operating Systems Principles. New York: ACM, 2024: 134−145

[50] Strati F, McAllister S, Phanishayee A, et al. DéjàVu: KV-cache streaming for fast, fault-tolerant generative LLM serving[C]//Proc of the 41st Int Conf on Machine Learning. New York: PMLR, 2024: 456−467

[51] Zhong Yuchen, Sheng Guangming, Liu Juncheng, et al. Swift: Expedited failure recovery for large-scale DNN training[C]//Proc of the 28th ACM SIGPLAN Annual Symp on Principles and Practice of Parallel Programming. New York: ACM, 2023: 447−449

[52] Jeong J, Baek S, Ahn J, et al. Fast and efficient model serving using multi-gpus with direct-host-access[C]//Proc of the 18th European Conf on Computer Systems. New York: ACM, 2023: 249−265

[53] Guerraoui R, Schiper A. Fault-tolerance by replication in distributed systems[C]//Proc of the 1996 Ada-Europe Int Conf on Reliable Software Technologies. Berlin: Springer, 1996: 38−57

[54] Kaashoek M, Tanenbaum A, Verstoep K, et al. On fault-tolerant data replication in distributed systems[C]//Proc of the 11th IEEE Int Conf on Distributed Computing Systems. Piscataway, NJ: IEEE, 1991: 222−230

[55] Mohan J, Phanishayee A, Chidambaram V, et al. CheckFreq: Frequent, fine-grained DNN checkpointing[C]//Proc of the 19th USENIX Conf on File and Storage Technologies. Berkeley, CA: USENIX Association, 2021: 203−216

[56] Amazon SageMaker. Machine learning service[EB/OL]. [2024-07-18]. https://aws.amazon.com/pm/sagemaker/

[57] Microsoft. Azure ML[EB/OL]. [2024-07-18]. https://learn.microsoft.com/ en-us/ azure/machine-learning

[58] The KServe. KServe[EB/OL]. [2024-07-18]. https:// github.com/ kserve/ kserve

[59] Fu Yao, Xue Leyang, Huang Yeqi, et al. ServerlessLLM: Locality-enhanced serverless inference for large language models[C]//Proc of the 18th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2024: 123−134

[60] PhoenixNAP. NVMe vs SATA: Storage comparison & differences [EB/OL]. [2024-07-18]. https://phoenixnap.com/kb/nvme-vs-sata

[61] Enterprise Storage Forum. NVMe vs SATA: Comparing storage technologies[EB/OL]. [2024-07-15]. https://www.enterprisestorageforum. com/storage/nvme-vs-sata/

[62] Amazon Web Services. Amazon simple storage service (S3)[EB/OL]. [2024-07-15]. https://aws.amazon.com/s3/

[63] Ge Xuran, Lai Mingche, Liu Yang, et al. SpacKV: A pmem-aware key-value separation store based on LSM-tree[C]//Proc of the 19th Int Conf on Network and Parallel Computing. Berlin: Springer, 2022: 327−339

[64] Lai Zhiquan, Li Shengwei, Tang Xudong, et al. Merak: An efficient distributed DNN training framework with automated 3D parallelism for giant foundation models[J]. IEEE Transactions on Parallel and Distributed Systems, 2023, 34(5): 1466−1478

[65] Compute Express Link Consortium. Compute express link (CXL)[EB/OL]. [2024-07-18]. https://www.computeexpresslink.org

[66] ARM. Computational Storage: A new way to boost performance[EB/OL]. [2024-07-18]. https://newsroom.arm.com/blog/computational-storage

[67] Feng Yangyang, Wang Qing, Xie Minhui, et al. From BERT to ChatGPT: Challenges and technical development of storage systems for large model training[J]. Journal of Computer Research and Development, 2024, 61(4): 809−823 (in Chinese)
(冯杨洋, 汪庆, 谢旻晖, 等. 从 BERT 到 ChatGPT: 大语言模型训练中的存储系统挑战与技术发展[J]. 计算机研究与发展, 2024, 61(4): 809−823)



Ge Xuran, born in 1998. PhD candidate. Student member of CCF. Her main research interests include key-value storage system and large language model inference system.

葛旭冉, 1998 年生. 博士研究生. CCF 学生会员. 主要研究方向为键值存储系统、大语言模型推理系统.



Ou Yang, born in 1987. PhD, associate professor. His main research interests include computer architecture and high performance interconnection technology.

欧　洋, 1987 年生. 博士, 副研究员. 主要研究方向为计算机系统结构、高性能互联技术.



Wang Bo, born in 1997. PhD candidate. Student member of CCF. His main research interests include distributed shared memory and parallel programming frameworks.

王　博, 1997 年生. 博士研究生. CCF 学生会员. 主要研究方向为分布式共享内存、并行编程框架.



Zhao Yu, born in 1997. PhD candidate. His main research interest includes efficient inference and pre-training of large language model.

赵　宇, 1997 年生. 博士研究生. 主要研究方向为大语言模型高效推理及预训练.



Wu Lizhou, born in 1991. PhD, assistant professor. His main research interests include high performance computing, spintronic design and test, and emerging computing paradigms based on non-volatile memory devices.

吴利舟, 1991 年生. 博士，助理研究员. 主要研究方向为高性能计算、自旋电子设计与测试、基于新型非易失存储器的计算范式.



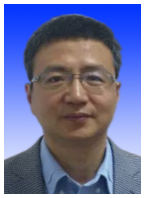
Wang Zicong, born in 1991. PhD, assistant professor. His main research interests include computer architecture and high performance interconnection technology.

王子聪, 1991 年生. 博士，助理研究员. 主要研究方向为计算机系统结构、高性能互联技术.



Chen Zhiguang, born in 1984. PhD, professor, PhD supervisor. Member of CCF. His main research interests include big data storage, and high performance computing and supercomputers.

陈志广, 1984 年生. 博士，教授，博士生导师. CCF 会员. 主要研究方向为大数据存储、高性能计算与超级计算机.



Xiao Nong, born in 1969. PhD, professor, PhD supervisor. His main research interests include high performance computing, storage system, and system architecture.

肖　侬, 1969 年生. 博士，教授，博士生导师. 主要研究方向为高性能计算、存储系统、系统结构.