arXiv:2603.17803v1 [cs.PF] 18 Mar 2026

SWARM：感知共激活的 KVCache 跨多块固态硬盘卸载

TuoweiWang* 清华大学
LiyunChu* 清华大学
RuwenFan 清华大学
JuRen† 清华大学

摘要

键值（KV）缓存已成为大语言模型（LLM）推理中内存消耗的主要来源。尽管将 KVCache 从图形处理器（GPU）的高带宽内存（HBM）卸载至 CPU DRAM 可以缓解设备内存压力，但对于大型、持久化工作负载而言，DRAM 仍受容量限制且成本高昂。固态硬盘（SSD）提供了一种高性价比的替代方案，但由于 PCI Express 吞吐量有限以及单设备带宽限制，朴素的基于 SSD 的分页在根本上受带宽限制。

在本文中，我们观察到，真实世界工作负载中的 KVCache 激活呈现出强烈且稳定的相关性。我们将这一现象称为*KVCache* 共激活：访问一个 KV 条目时，通常会伴随访问一组稳定且反复出现的其他 KV 条目。利用这一特性，我们提出了**SWARM**，一种基于固态硬盘的 KVCache 卸载系统，将受带宽限制的单设备访问转换为跨多块固态硬盘的并行 I/O。具体而言，SWARM 在线下对共激活的 KV 条目进行聚类，并通过结合选择性复制的基于图的放置策略，将生成的簇分布到多块固态硬盘上，以最大化并行 I/O 带宽。在运行时，SWARM 执行负载均衡的簇检索，并动态调整聚类与缓存决策，以适应不断变化的访问模式并维持较高的带宽利用率。评估结果表明，SWARM 将 I/O时间降低了 2.41×，并将有效带宽利用率提高了 2.72×。

1 引言

键值（KV）缓存是现代大语言模型（LLM）推理的核心组件。通过存储先前计算得到的键和值，它消除了冗余计算，并实现了高效的自回归解码。然而，KVCache 会随着序列长度和模型深度单调增长，其生命周期远超单个解码步骤。随着上下文长度不断增加 [22, 27, 48] 且模型不断加深 [17, 28, 42, 45], KVCache 已成为推理时内存占用的主要贡献者。

除了单次请求的占用量之外，KVCache 也日益跨请求和会话持久存在。例如，长文档或视频常常会针对不同问题被反复查询 [6, 11]。类似地，智能体式交互通常会复用之前步骤中的工具指令 [18,40] 和上下文 [50,51]。真实世界的生产部署 [3, 36]报告显示，长且共享的 KVCache 普遍存在。因此，KVCache 已从临时产物演变为一种长期存在的数据结构，使其自然且极具吸引力地成为持久化存储的候选对象。因此，设计一种专为 KVCache 管理打造的可扩展、高效存储系统，已变得愈发重要。

鉴于图形处理器（GPU）侧高带宽内存（HBM）容量有限且成本高昂，KVCache 通常会卸载到中央处理器（CPU）侧的动态随机存取存储器（DRAM） [16,23, 41]。为降低 DRAM 与 HBM 之间的输入/输出开销，近期工作 [44,47,53,54] 利用注意力机制固有的稀疏性，通过仅选择并传输与计算最相关的 *KV*来进行计算。然而，DRAM 容量仍然有限且成本高昂 [24, 43], , 因此不足以大规模承载大型、持久化的 KVCache。

在此背景下，**固态硬盘** （SSD）为 KVCache 卸载提供了一种颇具吸引力的替代方案。与动态随机存取存储器相比，固态硬盘的容量高出数个数量级，且单位比特成本更低，因此非常适合存储大型、持久化的 KVCache 状态。这种成本与容量优势促使人们越来越关注基于 SSD 的 KVCache 存储，并将其视为实现可扩展推理的一条途径 [7, 12, 14, 39]。遗憾的是，在固态硬盘与动态随机存取存储器之间简单地对 KVCache 进行分页，效果并不理想 [20,37]。与动态随机存取存储器不同，固态硬盘通过相对狭窄的 PCIe 接口访问，且单个设备提供的带宽有限 [7]。因此，从固态硬盘读取的 KVCache 会受到**带宽限制**，导致注意力计算停滞并增加推理延迟。

本文提出了一项关键观察：大语言模型中的 KVCache 在激活模式上表现出很强的相关性，我们将这一现象称为***KVCache*** 共激活。这一特性在大语言模型工作负载中普遍存在，却鲜少得到充分研究；若加以策略性利用，便能缓解 KVCache 卸载过程中的 SSD 带宽瓶颈。具体而言，在处理真实世界的数据集时，单个 KV 对的激活总是伴随着一组稳定且反复出现的其他 KV 对。这些共激活的键值可以同时读取，从而自然地实现输入/输出并行性，进而提升有效 SSD 带宽。

受这一观察启发，我们提出了**SWARM**，一种用于 KVCache 卸载的新型基于 SSD 的解决方案。SWARM 不再依赖单个 SSD 并受其设备带宽限制，而是将 KVCache 卸载到多个固态硬盘上

*这些作者贡献均等。
†通讯作者（renju@tsinghua.edu.cn）

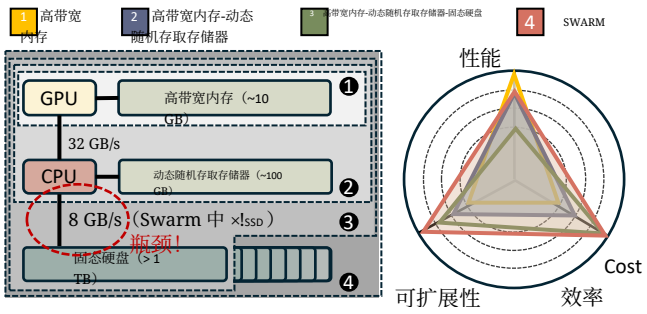

图 1：KVCache 卸载的内存层次结构。SWARM 利用 KVCache 共激活来汇聚多块固态硬盘的带宽，从而实现更优的性能与可扩展性权衡。

，并策略性地汇聚所有设备的带宽。SWARM 的一个关键思路是使 KVCache 共激活模式与多固态硬盘输入/输出并行性相匹配：识别出经常共同激活的 *KV* 对，并在多个固态硬盘之间进行分布、检索和更新，以最大限度地提升并行读取能力。如图 1所示，SWARM 将基于 SSD 的 KVCache 卸载从受带宽限制的单设备操作转变为高度并行的多设备数据路径，从而实现高效且可扩展的 KVCache 管理。

SWARM 采用两阶段设计，结合了离线和在线执行的分层优化：

在离线阶段，SWARM 捕获 KVCache 共激活模式，并将 KV 条目组织成感知相关性的簇，同时以簇中心点为代表构建轻量级的驻留于DRAM的索引。在这种集群化布局的指导下，它通过将中心点、局部窗口条目和热点簇保留在动态随机存取存储器中，并将其余集群化条目分布到各固态硬盘上，实现 KVCache 在动态随机存取存储器与固态硬盘层次结构中的分区，从而支持并行检索。

在在线阶段，SWARM 采用负载均衡调度，在多固态硬盘布局下高效检索选定的簇，且开销可忽略不计。随着解码推进，它还会动态调整簇成员关系和缓存决策，从而在不断变化的访问模式下维持较高的带宽利用率和检索效率。

我们在四个真实世界数据集和两种固态硬盘上对 SWARM 进行了评估，并以五种具有代表性的大语言模型作为基准。结果表明，SWARM 平均可将输入/输出时间降低 2.41×，并使有效带宽利用率平均提升 2.72× 。值得注意的是，当扩展到八个固态硬盘时，SWARM 的带宽最高可达 37.67 GB/s，与图形处理器高带宽内存和 CPU DRAM 之间的带宽相当。

我们将本文的贡献总结如下：

- 我们发现了大语言模型推理中一种普遍存在但尚未得到充分研究的现象——KVCache 共激活，并展示了如何利用这一现象缓解基于固态硬盘的 KVCache 卸载中的主要带宽瓶颈。
- 我们设计了一种两阶段解决方案，系统地对固态硬盘中的共激活键值对进行建模、放置、检索和更新，从而最大限度地提高带宽利用率。
- 我们在多种大语言模型、数据集和硬件配置上进行了广泛评估，相较于现有方案，在性能和成本方面均取得了改进。

2 背景

2.1 自回归生成中的 KVCache

先进的大语言模型大多采用自回归解码，其中词元按顺序生成，且每个新词元都会关注所有先前的词元。这一机制通过 多头注意力（MHA） [46]实现。对于每个注意力头，输入词元嵌入会被投影为三个组成部分：查询（Q）、键（K）和值（V）。注意力计算定义如下：

$$\text{Attention}(Q,K,V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

其中d_k 表示键向量的维度。

在t解码步骤中，当前查询q_t 会关注所有先前生成的键（$K_{<t}$）和值（$V_{<t}$）。由于这些历史张量在计算完成后保持不变，因此每一步都重新计算它们会带来大量不必要的开销。这种冗余会大幅增加响应生成延迟，使系统难以满足服务级别目标（SLO） [39]。为消除这一低效问题，现代推理系统会存储并复用先前计算出的键和值，统称为*KVCache*。

在实际应用中，KVCache 具有两个显著特征：**特征 #1：占用空间巨大**。对于具有L 层且隐藏维度为d的模型，长度为S的序列所需的 KVCache 内存占用按$O(L \times S \times d)$增长。随着模型不断加深，上下文窗口不断扩展，以容纳长文档、代码库和多模态输入（例如图像和视频），KVCache 的规模很容易超过模型参数本身。如表 1所示，在长上下文场景下，KVCache 的总占用量可能达到数百甚至数千 GB，在大语言模型推理过程中造成严重的内存压力。

特性 #2：时间持久性。KV-Cache 的生命周期会超出单个解码步骤，甚至往往会

表1：不同序列长度下 KVCache 内存占用相对于 GPT-3 模型权重（FP16）的比例。

权重 s	$+32K\ s$	$=64K\ s$	$=128K\ s$	$=256K\ s$	$=512K$
326GB	144GB	288 GB	576 GB	1.13 TB	2.25 TB
-	0.44×	0.88×	1.77×	3.53×	7.07×

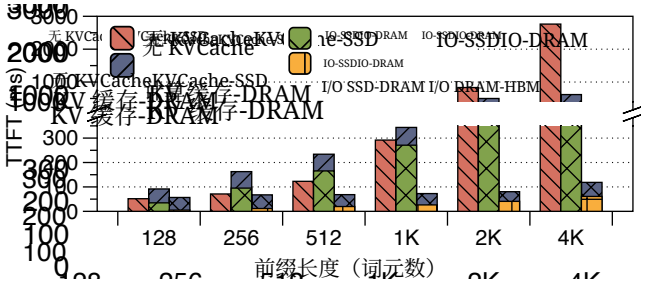

图 2：Qwen3-4B 在无 KVCache、基于 SSD 的 KVCache 和基于 DRAM 的 KVCache 下的 TTFT 分解。

超出单个请求的范围。一方面，长文档或视频可能会在底层上下文保持不变的情况下，针对不同问题被反复查询。另一方面，真实世界工作负载（例如多轮智能体和迭代式编程助手）会在连续交互中持续复用历史上下文。在这两种情况下，丢弃并重新计算 KVCache 不仅会浪费计算资源，还会引入本可避免的延迟。如图2所示，有效复用 KVCache 可以显著缩短首词元时间（TTFT），尤其是在前缀处理占据主要延迟的长上下文场景中。

2.2 三层 KVCache 卸载

KVCache 巨大的占用空间和时间持久性对现有内存层次结构构成了挑战。实践中，图形处理器高带宽内存（HBM）充当**第一层**，提供低延迟和高吞吐量，但容量有限且成本高昂。为了扩展可用内存池，通常使用 CPU DRAM 作为**第二层**进行卸载，其容量比 HBM 高一个数量级。推理期间，KVCache 条目会根据需要通过 PCIe 接口从 CPU 传输到 GPU。得益于现代 PCIe 链路的高带宽 [38] 以及预取等软件层面的优化 [10,52] 和缓存 [12,14],，这种数据传输开销通常可以控制在可接受范围内。

尽管如此，动态随机存取存储器的容量从根本上仍然受限，而且大规模部署时成本高昂。随着 KVCache 的规模和生命周期不断增长，仅依赖高带宽内存和动态随机存取存储器，在经济性和实用性上都难以为继。在此背景下，固态硬盘正成为 KVCache 卸载的一种**第三层**高性价比选择。 [25], 固态硬盘基于 NAND 闪存构建，容量远大于动态随机存取存储器，每比特成本更低，容量可扩展至太字节级别。此外，现代服务器平台支持通过独立的 PCI Express 通道将多个固态硬盘连接至单个中央处理器，从而实现灵活的容量扩展和设备级并行处理。

然而，将固态硬盘纳入存储层级后，瓶颈便转移到了**I/O 带宽**。尽管现代固态硬盘采用了非

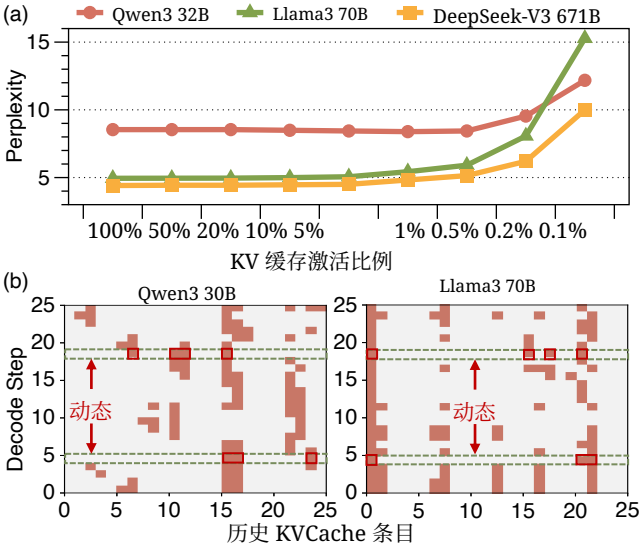

图 3：KVCache 激活稀疏性。(a) Wiki-Text 上不同 KV 激活比例下的困惑度（上下文长度为 2K）。(b) 解码步骤中 KVCache 的动态激活模式。

易失性内存标准 (NVMe) [35],等高性能协议，但与动态随机存取存储器相比，其带宽仍受到架构上的限制。KVCache 位于动态随机存取存储器中时，可通过 PCIe×16 链路以相对较高的吞吐量传输至图形处理器。相比之下，大多数 NVMe 固态硬盘通过 PCIe×4连接，其带宽仅为动态随机存取存储器与高带宽内存之间带宽的一小部分。如图 2所示，将 KVCache 卸载到固态硬盘会引入显著的 I/O 延迟，而简单的设计可能会严重降低整体推理性能。

2.3 稀疏性驱动的 KVCache 激活

为降低 KVCache 在不同内存层级间传输的开销，近期研究 [44, 47, 53, 54] 利用了注意力机制中固有的稀疏性。即使上下文长度很大，最终注意力输出也主要由少量注意力分数最高的词元决定。如图3(a) 所示，即使忽略很大一部分 KVCache，对模型输出的影响也微乎其微。因此，无需激活并传输所有历史键和值，只需在每个解码步骤中识别并加载这些关键词元对应的 KVCache 条目即可。

然而，基于稀疏性的键值缓存激活面临两个关键挑战。**首先**，KVCache 稀疏模式本质上是动态的，并且取决于查询。如图 3(b) 所示，激活的 KV条目集合会随输入词元而变化。系统需要在每一步实时计算相关性分数（例如查询键内积），并检索关键条目，因此需要高效的检索机制。**其次**，尽管稀疏性减少了数据传输和注意力机制

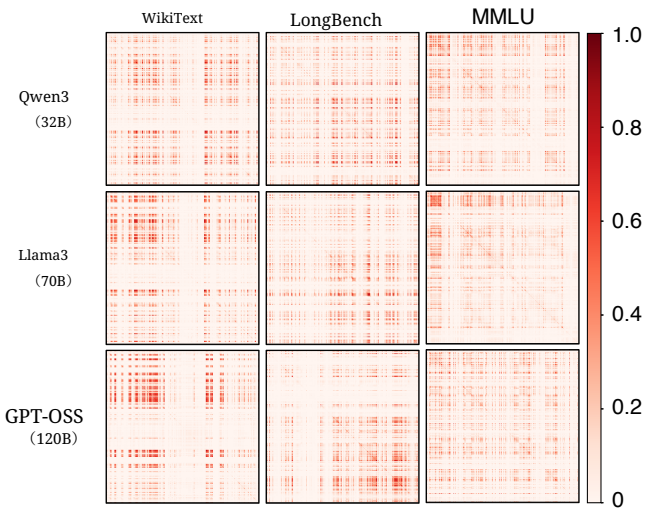

图 4：不同模型和数据集上的 KVCache共激活模式可视化。矩阵中的每个元素 (i, j) 表示键值缓存条目 e_i 和 ej 的激活频率。

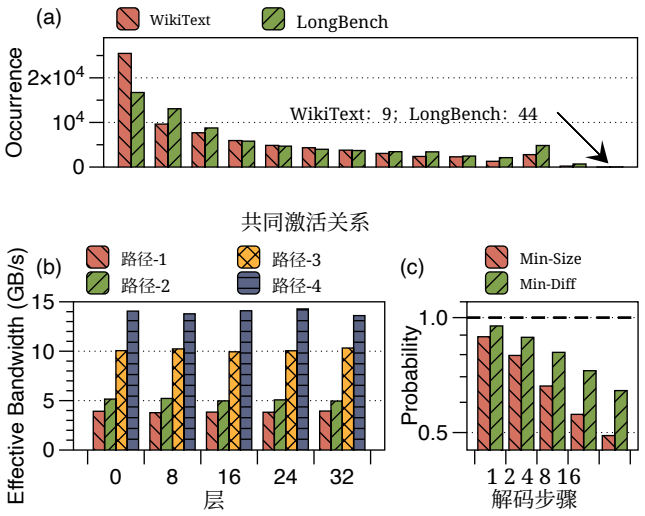

图 5：（a）Qwen3-32B 在 128 个查询上的 KVCache 共激活分布。（b）检索路径对各层有效带宽的影响。（c）朴素更新下，KVCache 共激活概率随解码步骤增加而衰减。

计算量，从而降低整体推理延迟，但并未从根本上消除固态硬盘带宽瓶颈。这种改进源于缩减工作负载，而非提高有效输入/输出吞吐量。因此，固态硬盘带宽仍处于推理的关键路径上，并继续成为主要的性能瓶颈。

3 动机与挑战

3.1 洞见：KVCache 共激活

观察。本文旨在解决 SSD 带宽瓶颈，以实现高效且可扩展的 KVCache 卸载。我们的方法基于一项深刻却鲜少受到探索的观察：KVCache 激活并非均匀随机，而是表现出很强的相关性。如图4所示，在处理真实世界工作负载时，某个 KV 条目的激活往往伴随着一组稳定且反复出现的其他条目。我们将这一现象称为 ***KVCache*** 共激活，它自然源于稀疏性驱动的 KVCache 激活，并且在不同模型和数据集中始终存在。这一观察表明，我们有机会利用结构化访问模式。

核心思想。 KVCache 共激活揭示了 KVCache 激活模式中固有的并行度。在每个 KVCache 共激活组内，KV 条目在计算上彼此独立，因此可以并发获取，而不会引入额外的同步。这种并行度与现代存储系统的拓扑结构天然契合，因为多个固态硬盘通过独立的 PCI Express 通道连接。通过将共激活的 ***KV*** 条目策略性地分布到多块固态硬盘上，系统便可聚合设备级带宽，用于 KVCache 传输。聚合后的带宽可达到甚至超过动态随机存取存储器与高带宽内存之间的有效带宽，从根本上缓解这一瓶颈。

3.2 主要挑战

尽管 KVCache 共激活揭示了固有的输入/输出并行性，但要将这一发现转化为实用的基于 SSD 的系统，绝非易事。通过全面分析，我们确定了必须应对的四项关键挑战：

挑战 #1：建模。KVCache 条目数量庞大，且其激活模式复杂，因此难以识别共激活条目。如图 5(a) 所示，实际中的共激活模式并非泾渭分明，而是形成相互重叠的结构，有些条目参与了 20 多种共同激活关系。选择性复制有助于解耦这些依赖关系，但会引入一个 NP 困难的组合优化问题。

挑战 #2：放置。 除了识别之外，系统还必须将这些相关的 KV 条目进一步组织成结构良好的布局。尤其是，这种放置应与固态硬盘卸载层级自然契合。一方面，共激活条目应尽可能均匀地分布在各固态硬盘上，以最大化并行读取带宽。另一方面，系统应能够高效地选择这些条目，而无需先将它们传输到动态随机存取存储器。

挑战 #3：检索。 复制进一步增加了检索的复杂性。同一个 KV 条目可能存储在多个固态硬盘上，从而形成多条可行的检索路径。如图所示——

5 5(b)，不同的检索路径可能导致有效带宽显著不同，即使 KVCache 放置完全相同。次优策略可能重新造成负载不均衡，抵消复制带来的收益；而过于复杂的调度则可能产生难以承受的每步开销。

挑战 #4：更新。LLM 解码会持续生成新的 KV 条目，需要将它们整合到现有组织结构中。如图5(c) 所示，基于组大小或嵌入相似度进行的朴素更新会严重降低共同激活概率，从而限制随时间推移所能实现的并行度。此外，缓存决策应适应不断变化的访问模式，以减少对热点条目的重复输入/输出。

4 设计概述

图 6 展示了 SWARM 的概览。SWARM 利用 KVCache 的共激活特性，系统地实现共激活 KV 条目的建模、放置、检索和更新。整体工作流程分为两个阶段：**离线阶段**（§5）。在此阶段，SWARM 决定 KV 条目在各固态硬盘上的放置方式，以最大限度地提高聚合带宽利用率。❶ 首先，它对 KVCache 的共激活模式进行建模，并据此将条目组织成簇。❶ 同时，SWARM 在各簇上构建索引结构，以支持在线推理期间的高效检索。基于这种聚类方式，它将每个簇中的条目均匀分布到各固态硬盘上，以便进行并行数据传输。**在线阶段**（§ 6）。针对面向聚合带宽的放置方式，SWARM 进一步引入运行时策略，以充分利用可用带宽。一方面，❸ 它采用负载均衡调度机制，在簇检索期间路由词元。另一方面，❹ SWARM 在解码期间动态调整聚类和缓存决策，以适应不断变化的访问模式并保持高效率。

5 离线阶段：建模与放置

5.1 相关性感知聚类

KVCache共激活分析表明，KV条目通常以高度相关的组为单位被激活，而非独立激活。受此发现启发，SWARM 将 KVCache 组织成一组结构化的簇，每个簇都由具有强共同激活关系的条目组成。如图7所示，SWARM 采用四步聚类流程。

步骤 1：模式提取。SWARM 首先对 KVCache 共激活模式进行分析，将其作为一次性预处理
推理之前的步骤。对于每个模型层，我们构建一个邻接矩阵 $\mathbf{A}$，以捕捉 KV 条目之间的成对协同激活统计。每个元素$\mathbf{A}ij$ 记录 KV 条目e_i 和ej 在注意力计算过程中共同激活的次数。因此，$\mathbf{A}$ 编码了所有条目对的经验协同激活频率。基于测得的

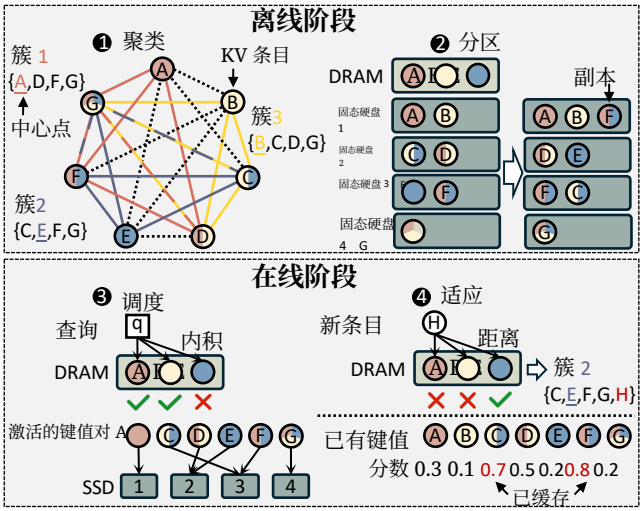

图6：SWARM 概览。

频率$f(e_i,e_j)$，我们估算e_i和e_j之间的共同激活概率，记为$P(e_i,e_j)$，如下所示：

$$P(e_i,e_j)=\frac{f(e_i,e_j)}{\sum_{k=1}^{N}\sum_{l=1}^{N}f(e_k,e_l)} \tag{2}$$

其中 N 表示上下文长度。在任务无关数据集上进行性能分析，以获得具有统计代表性的 KVCache 行为特征。

步骤 2：图抽象。 基于提取出的共激活模式，SWARM 将这些关系抽象为完全图 ，以便进行有原则的图论优化。在此建模中，每个节点代表一个 KV 条目，每条边代表一对条目之间的共激活关系。我们将边权重定义为两个条目之间的距离度量，记作 $d(e_i,\ e_j)$，用于量化它们共激活的强度，具体如下：

$$d(e_i,e_j):=1-P(e_i,e_j) \tag{3}$$

根据此定义，距离越小，共激活越强。极端情况下，当 $d(e_i,\ e_j)\ =1$时，这两个条目总是共激活的。基于这一抽象，可以为所有条目构建成对距离矩阵 $\mathbf{D}$。因此，共激活 KV 条目的聚类可重新表述为一个图优化问题：找出边距离最小且紧密连接的节点集合。

SWARM 在最后两个步骤中完成这一优化过程。考虑到在完整加权图上进行精确的组合优化属于 NP 困难问题，[26], SWARM 采用启发式算法，在效率与解的质量之间取得平衡，如算法 1 所示。其核心思路是首先选择整体共同激活程度最高的条目作为中心点（第 6–10 行），然后以贪心方式将该中心点扩展为一个

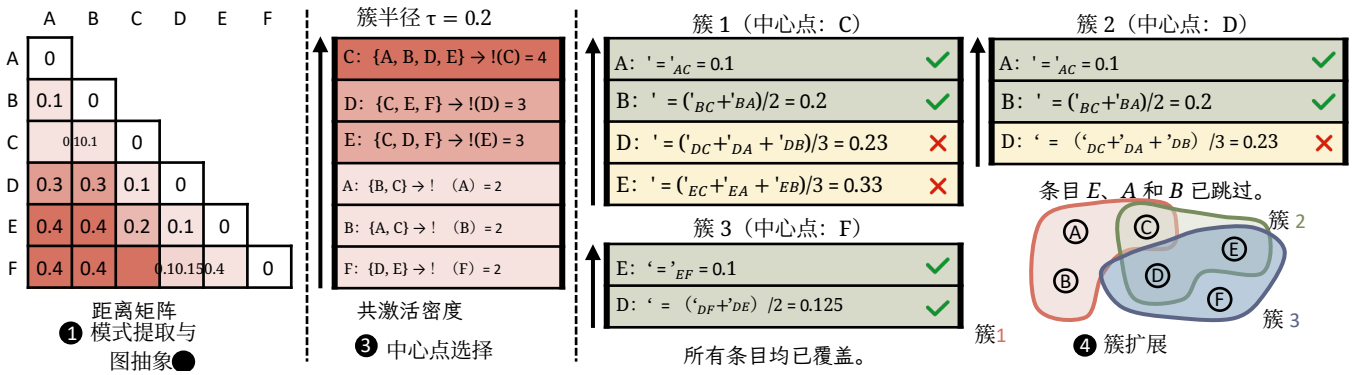

图 7：离线聚类示例。(1–2) 根据 KVCache 共激活模式得出的距离矩阵形成图抽象。(3) 按顺序选择共激活密度最高的条目作为中心点。(4) 每个簇通过迭代添加条目来扩展，所添加条目到簇成员的平均距离在半径阈值 τ 之内，直至所有条目均已覆盖。

算法 1 KVCache 共激活聚类

输入：KV 条目集$\mathcal{E}$，距离矩阵 **D**，簇半径 τ
输出：KVCache 簇集$\mathcal{C}$
函数 BUILDCLUSTERS($\mathcal{E}$,**D**, τ)
  初始化$\mathcal{C} \leftarrow \emptyset$
  初始化已覆盖[e] ← 对所有$e \in \mathcal{E}$设置为 false
  ▷ 步骤 3：中心点选择
  对每个 $e_i \in \mathcal{E}$ **执行**
    共激活密度 $\rho[e_i] \leftarrow \sum_{e_j \in \mathcal{E}} \mathbb{I}(\mathbf{D}(e_i, e_j) \le \tau)$
  结束循环
  按 ρ[·] 降序排列中心点队列 $Q_m \leftarrow \text{sort}_{\mathcal{E}}$
  ▷ 步骤 4：簇扩展
  for 对于每个 $m_i \in Q_m$ **and** 已覆盖[m_i] = 为 false **do**
    KVCache 簇$C_i \leftarrow \{m_i\}$
    候选队列 $Q_c \leftarrow \{e \in \mathcal{E} \setminus \{m_i\} \mid D(m_i, e) \le \tau\}$
    按$\mathbf{D}(m_i, \cdot)$的升序对$Q_c \leftarrow$ 和Q_c 进行排序
    对于每个 $c_j \in Q_c$**和** $\sum_{e_k \in C_i} \mathbf{D}(c_j, e_k)/|C_i| \le \tau$，**执行**
      $C_i \leftarrow C_i \cup \{c_j\}$
    结束循环
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_i\}$
    已覆盖[e_i] ← 对所有$e_i \in C_i$均为 true
    跳出 if $\forall e \in \mathcal{E}$, 已覆盖[$e$] =true
  结束 for
  返回 $\mathcal{C}$
结束函数

簇，迭代纳入共同激活关系较强的其他条目（第 11–28 行）。

步骤 3：中心点选择。为了量化一个 KV 条目的共同激活强度，我们进一步将条目e_i的共激活密度记为 $\rho(e_i)$，定义如下：

$$\rho(e_i) := \left|\{e_j \mid j \neq i,\ d(e_i, e_j) < \tau\}\right| \tag{4}$$

其中 τ 是预定义的簇半径，用于控制预期的簇大小。根据这一公式，半径 τ 范围内邻居越多的条目，共激活密度就越高，这表明其共同激活关系更强。SWARM

随后，SWARM 会计算所有 KV 条目的 ρ（e_i），并按降序排序，以构建中心点队列Q_m。接着，系统依次从 Q_m 中选取条目作为簇中心点。按照构造方式，这些中心点具有最高的共激活密度，因此在被激活时，最有可能与各自簇中的其他条目共同激活。

步骤 4：簇扩展。选定中心点m_i 后，SWARM 会计算其与所有其他条目之间的距离。满足$d(m_i, e_j)$的任何条目e_j都会被视为簇C_i的候选成员。这些候选成员会按照其与中心点之间的距离进行排序，并组织到候选队列Q_c中。对于从Q_c中取出的每个候选成员c_j，我们将其到簇C_i 的距离定义为平均成对距离：

$$d(c_j, C_i) := \frac{\sum_{e_k \in C_i} d(c_j, e_k)}{|C_i|} \tag{5}$$

如果该距离小于 τ，则将 c_j 添加到 C_i 中。按照此标准迭代扩展簇，直到无法再纳入任何条目。当每个条目都至少属于一个簇时，聚类过程便终止。

值得注意的是，该算法会自然地为高度共激活的 KV 条目创建副本，这对于捕捉交错的共激活模式至关重要。考虑这样一种情况：条目A 经常与条目B 和条目C共同激活，而B 和C 彼此之间很少共同激活。如果每个条目都只能进行单一分配，那么A 必须与B 和C同时归为一簇，或者只与其中一个归为一组。前一种情况下，当只需要B 时，读取该簇会导致冗余传输条目C。后一种情况下，不包含A 的簇则无法受益于其与A频繁共同激活的特性。通过允许为高度共激活的条目创建副本，SWARM 以极低的存储开销化解了这一矛盾。

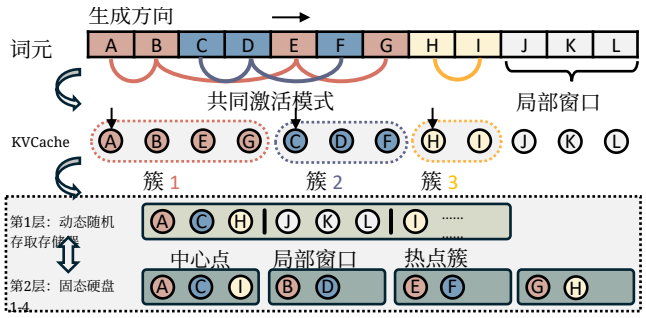

图8：KVCache 在动态随机存取存储器-固态硬盘层级结构中的分区。中心点、局部窗口条目和热点簇保留在动态随机存取存储器中，而聚类后的条目则分布在各个固态硬盘上。

5.2 适合卸载的分区

SWARM 将 KVCache 簇视为原子检索单元，并按需将其从固态硬盘传输到动态随机存取存储器。为实现准确高效的检索，SWARM 借鉴 KVCache 检索与数据库系统中经典数据检索之间的概念关联，为 KVCache 簇构建索引结构。通过将每个簇的条目分布到多个固态硬盘上，传输单个簇时便可利用多个设备的聚合带宽，从而缓解带宽瓶颈。如图 8 所示，SWARM 通过在双层内存层次结构中进行精心分区来实现这一设计，在最大限度提高聚合带宽利用率的同时，尽可能降低输入/输出开销。

第1层：动态随机存取存储器。 SWARM 在动态随机存取存储器中存放三类数据。(1)中心点。SWARM 使用每个簇的中心点作为其索引条目，并维护一张记录其固态硬盘放置位置的路由表。每个中心点都会计算其与输入查询的相关性得分，以确定是否应检索相应的簇。在聚类过程中，簇中的每个条目都被限制在其中心点的半径范围内。因此，中心点自然可以作为簇的代表性条目，而且最有可能与同一簇中的其他条目共同激活。将此索引保留在动态随机存取存储器中，SWARM 无需产生固态硬盘访问开销即可实现高效的簇选择。(2) 局部窗口条目。SWARM 将局部窗口内的键值条目（例如，256 个词元）保留在动态随机存取存储器中，以便在推理过程中保持上下文连续性。随着生成推进，滑动缓冲区会动态跟踪最近的条目。(3) 热点簇。为利用簇级激活局部性，SWARM 会将激活最频繁的簇缓存在动态随机存取存储器中。为在空间开销与输入/输出缩减之间取得平衡，我们定义簇C_i的成本效益分数 如下：

$$S(C_i) = f_i \cdot \frac{T_{\text{base}} + s_i \cdot T_{\text{transfer}}}{s_i} \tag{6}$$

其中，f_i 表示簇C_i的激活频率，T_{base}表示固态硬盘寻址延迟，T_{transfer} 表示每条目的传输成本。在动态随机存取存储器缓冲区容量固定的情况下，按照该分数降序缓存簇，以便在内存限制下最大化整体成本效益。

第2层：固态硬盘。 SWARM 将所有 KVCache 簇存储在固态硬盘上。由于 KV 条目已根据共激活模式完成聚类，此处的主要目标是尽可能均匀地分布各簇，以最大限度地提高聚合带宽利用率。SWARM 采用轮询放置策略，可灵活适应不同的簇大小和固态硬盘数量。具体而言，SWARM 维护一个全局磁盘指针 p_{global}，并按顺序处理各簇。对于每个簇C_i，目标磁盘标识符按如下方式确定：

$$\text{id}_i = p_{\text{global}} \bmod N_{\text{disk}},\ p_{\text{global}} = p_{\text{global}} + |C_i| \tag{7}$$

其中，N_{disk} 表示固态硬盘总数。随后，每个簇从磁盘 ID $_i$ 开始放置其条目，并以循环回绕的方式依次分布到各固态硬盘上。该策略确保存储利用率均衡，并在簇检索期间最大限度地提高并行数据传输能力。

6 在线阶段：检索与更新

6.1 负载均衡调度

在 SWARM 完成离线优化后，KV 条目会以带宽聚合布局分布在各固态硬盘上。运行时，SWARM 专注于根据输入查询高效传输选定的簇。该设计遵循两个关键目标。首先，聚类过程自然会产生 KV 条目副本。尽管复制对于准确捕获共激活模式至关重要，但如果同时检索包含重叠条目的多个簇，就可能导致冗余数据传输。其次，应以负载均衡方式组装选定的簇，以充分利用可用的聚合带宽。同时，整体调度机制必须保持轻量，以避免引入额外的运行时开销。

如图 9 所示，SWARM 设计了一种解耦式条目桶调度算法。首先，在将 KV 簇传输到动态随机存取存储器之前，SWARM 对所有已激活的簇执行全局合并，以构建输入/输出条目集：

$$\mathcal{I}_{\text{I/O}} = \left(\bigcup_{C_i \in C_{\text{activated}}} C_i \right) \setminus \mathcal{I}_{\text{DRAM}} \tag{8}$$

此操作会删除簇之间重复的键值条目，并筛除已驻留在动态随机存取存储器中的条目，从而避免冗余数据传输。接下来，为每个固态硬盘分配一个专用桶，用于存储待传输的条目。首先，根据副本因子（即该条目所在的固态硬盘数量），按升序对$\mathcal{I}_{\text{I/O}}$ 中的所有条目进行排序。没有副本的条目会直接分配到对应的固态硬盘桶中。对于

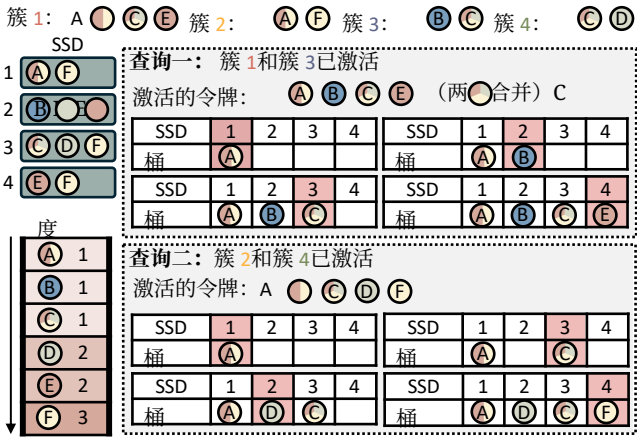

图9：两个查询下 KVCache 检索调度的示例。合并已激活的簇，并按度升序将条目调度到各个固态硬盘桶中。

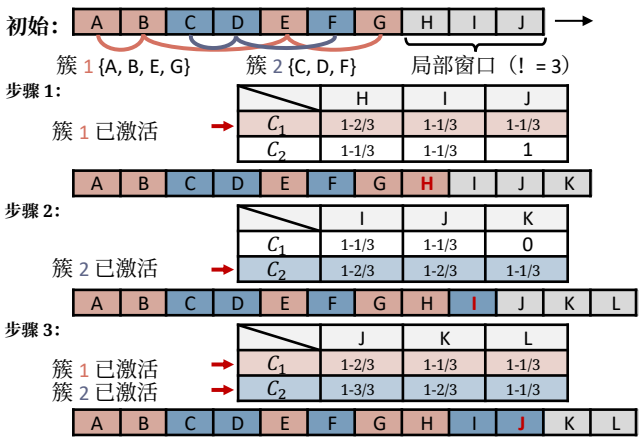

图 10：连续三个解码步骤中的簇维护示例。每个新的键值条目都会根据其在局部窗口中的共同激活模式，被分配到最近的簇。

具有多个副本的条目，SWARM会将每个条目分配到当前桶规模最小的固态硬盘，从而平衡各设备之间的负载。如果多个固态硬盘的桶规模相同，则任意选择其中一个。

为每个固态硬盘构建好桶后，系统会迭代提取每个桶的头部条目，并将它们汇总为一个提交批次。随后通过一次系统调用将该批次提交给内核。通过将多个输入/输出请求合并为大型批量提交，SWARM 最大限度地减少了提交操作次数，并减少了用户态与内核态之间的上下文切换。

6.2 簇对齐自适应

SWARM 还会在在线阶段根据系统状态的变化进行动态适应。新生成的 KVCache 条目可能会改变聚类结构，而激活模式的变化则会调整动态随机存取存储器缓存优先级。为应对这些动态变化，SWARM 采用了两种专门的更新策略。**簇维护**。如 § 5.2所述，新生成的 KV 条目会在大小为 W 的局部窗口内保留在动态随机存取存储器中。SWARM利用这一 W步间隔，分析这些条目与现有 KVCache 簇之间的共激活模式。如图 10所示，新生成的 KV 条目 e_{new} 与簇C_i 之间的距离定义为：

$$d(e_{new}, C_i) = 1 - \frac{f(e_{new}, m_i)}{\sum_{j=1}^{N} f(e_{new}, m_j)} = 1 - \frac{f(e_{new}, m_i)}{W} \quad (9)$$

其中，m_i 表示簇C_i 的中心点，N 表示簇的总数。如果计算出的距离小于簇半径 τ，SWARM 会将e_{new} 分配到簇 C_i ，并将其放置在为该簇分配的下一块固态硬盘上（即磁盘索引为 $\text{id}_i + |C_i| + 1$的位置；如有必要，对N_{disk} 取模）。如果有多个簇满足距离阈值，则将该条目添加将其分配给所有簇，同时保持受控复制。这种增量放置策略在解码推进的同时维持簇一致性，并支持并行传输。

缓存替换。在离线阶段，SWARM 根据成本效益分数将 KVCache 簇缓存到动态随机存取存储器中。如公式 6 所定义，该分数中使用的激活频率 f_i最初根据离线分析得出。为适应不断变化的运行时行为，SWARM 会根据观测到的访问模式动态更新每个簇的激活频率。具体而言，每当某个簇被激活，其频率便加一。相反，对于已缓存于动态随机存取存储器中、但在某个解码步骤期间未被激活的簇，其频率则减一。这一设计会奖励表现出时间局部性的簇，同时惩罚不活跃的簇。为高效管理动态随机存取存储器缓存，SWARM 使用按成本效益分数排序的最小堆结构维护已缓存的簇，从而实现高效更新和替换。

7 实现

我们将 SWARM 实现为一个端到端系统，用于在 CPU-GPU协同执行中高效地进行基于固态硬盘的KVCache卸载。该系统同时涵盖 Python 和 C++，包含超过 2,000 行代码，覆盖从底层数据移动到运行时调度的完整技术栈。

异步I/O。 我们构建了定制的 C++输入/输出后端，以充分利用多个固态硬盘的聚合带宽。为实现高性能的异步数据传输，我们利用 Linux内核接口 *io_uring* [4] 发出并发的非阻塞读取命令。为进一步消除中央处理器侧的复制开销，所有固态硬盘读取操作都直接写入预先分配的固定主机内存。这使 NVMe DMA控制器

表 2：模型配置。

模型	规模	注意力机制	隐藏维度	组大小
Qwen3-S [49]	14B	GQA	5120	5
Qwen3-M [49]	32B	GQA	5120	8
Llama3.1 [33]	70B	GQA	8192	8
GPT-OSS [1]	120B	分组查询注意力	2880	8
Qwen3-L-MoE [49]	235B	分组查询注意力	4096	16

表 3：数据集配置。

数据集	描述
WikiText [32]	基于维基百科的语言建模数据集
LongBench [5]	长上下文理解能力
MMLU [19]	多任务知识评测基准
GSM8K [8]	小学数学问题数据集

能够将数据直接写入这些缓冲区，从而通过 CUDA复制引擎实现向 GPU 高带宽内存的零拷贝传输 [34]。

流水线预取。 为了隐藏自回归解码期间访问固态硬盘的延迟，我们精心设计了一条流水线，使计算与预测性 KVCache 传输能够重叠进行。具体而言，当图形处理器执行第 L 层的注意力计算时，中央处理器会同时为第 $L+1$ 层执行以下三个阶段：(1) 将第 L 层的嵌入与第 $L+1$ 层的簇中心点进行比较，预测可能会访问的 KV 簇；(2) 发出异步读取请求，将相应数据从固态硬盘加载到固定主机内存；(3) 通过专用的 CUDA 流，将读取的数据传输至 GPU 高带宽内存。基于相邻层之间的嵌入相似度 [21], , 该设计有效地将固态硬盘访问与图形处理器计算重叠起来。

8 评估

8.1 实验设置

硬件。我们的系统包含一颗 Intel Xeon Gold 6530 中央处理器（64 核、128 线程）、8 个 NVIDIA H20 图形处理器（每个配备 96 GB 高带宽内存）、1 TB DDR5 动态随机存取存储器，以及最多 8 个 NVMe 固态硬盘。为覆盖异构存储，我们同时使用高端 Samsung PM9A3 SSD（最高 1.1M IOPS、6.9 GB/s）和低端 Intel Optane 900P 固态硬盘（最高 0.55M IOPS、2.5 GB/s）。

模型。 我们选择了六种广泛采用的大语言模型，如表 2 所示。这些模型在架构、规模和键值缓存特性方面各不相同。所有模型均采用分组查询注意力（GQA） [2],，组大小各不相同。我们按照常见做法，以 BF16 精度评估所有模型。

数据集。 我们在一系列多样化的数据集上评估 SWARM，涵盖纯文本、指令遵循和数学任务。这些数据集涵盖了广泛的语言结构，使我们能够针对不同的共激活模式进行全面评估。为支持更长的前缀长度，当单个序列不足时，我们会拼接多个样本，构建最长达 1M 词元的扩展输入。**基线方法**。我们将 SWARM 与三种具有代表性的键值缓存管理方法进行比较。所有方法均在稀疏率为 10% 的条件下评估。(1)无集群：KV条目存储在固态硬盘上，不进行任何聚类或索引结构处理。为识别激活的 KV条目，首先将整个 KVCache 加载到动态随机存取存储器中（必要时分块加载），以计算注意力分数，随后从固态硬盘中检索所需条目。为确保公平，我们仅报告注意力分数计算和所需条目传输所耗费的时间。(2)*InfLLM* [47]：KV条目按固定大小的块组织。词元重要性在块级别确定，推理过程中仅检索选中的块。(3)*PQCache* [53]：一种基于聚类的方法，使用 K-means [31],对 KV条目进行分组，并利用簇质心识别待检索的相关词元。

8.2 总体性能

吞吐量。图 11(a) 报告了以每秒词元数（TPS）衡量的端到端吞吐量。SWARM 始终实现最高的 TPS，吞吐量分别比无集群、InfLLM 和 PQCache 提高 3.99×、1.76× 和 1.47× 。由于缺乏任何结构化的 KVCache 管理，无集群的吞吐量最低。尽管 InfLLM 和 PQCache 引入了结构化方法，但它们基于块划分或嵌入相似性的策略与 KVCache 共激活模式并不十分契合。相比之下，SWARM 会根据 KVCache 条目的激活模式显式地对其进行聚簇，并将它们分配到多个固态硬盘上，从而有效缓解带宽瓶颈。

带宽。图 11(b) 展示了 KV 检索期间详细的 I/O 带宽。与吞吐量结果一致，SWARM 的带宽利用率最高，平均分别超过无集群、InfLLM 和 PQCache 3.95×、2.67× 和 1.55×。通过结合分层离线与在线优化，SWARM 不仅能在各设备间均衡分配 I/O 工作负载，还能减少不必要的数据传输，使带宽利用率接近固态硬盘的实际带宽上限。此外，I/O 改进显著转化为端到端延迟的降低，这进一步证实 I/O 是基于固态硬盘卸载的主要瓶颈。SWARM 以极低的运行时开销解决了这一瓶颈。

准确率。 图 11(c) 展示了归一化准确率，其中预言机词元选择（NoCluster）取得了最高分。对于所有模型和数据集，SWARM 的准确率始终最接近这一上界。这表明，SWARM 能有效捕捉与注意力动态高度吻合的 KVCache 共激活模式，从而检索出信息量最大的词元。相比之下，其他基线方法的 KVCache 结构与激活模式不匹配，导致纳入无关

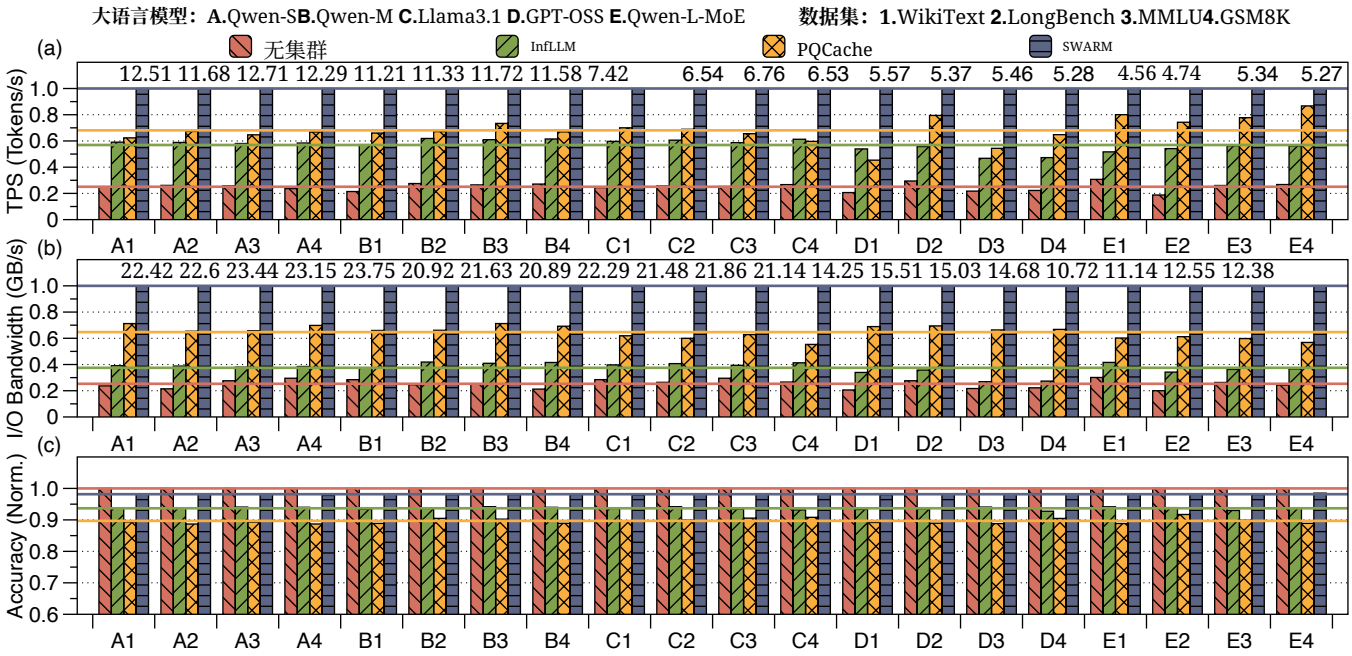

图 11 ：跨模型和数据集的总体性能（归一化）。线条表示平均值 .

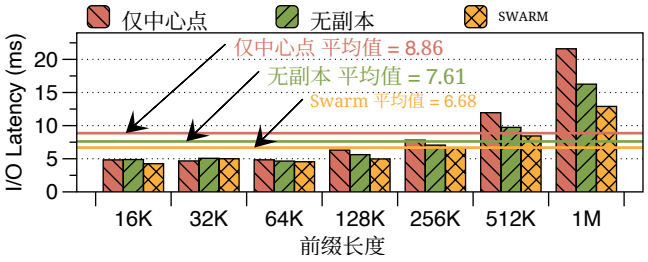

图 12：Qwen3-32B 在三种聚类策略下，不同前缀长度对应的输入/输出延迟对比。

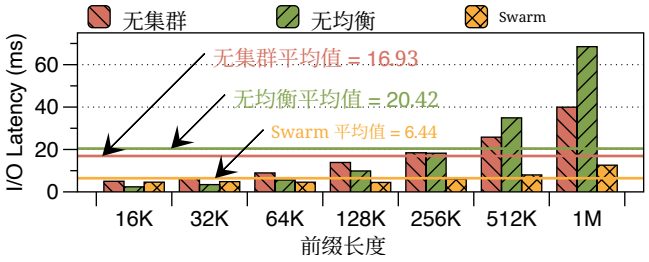

图13：Qwen3-32B 在三种 SSD 放置策略下，不同前缀长度对应的输入/输出延迟对比。

词元或排除重要词元。

8.3 消融研究

离线建模。 我们首先评估聚类算法的有效性。我们考虑两种基线方法：仅中心点，它仅使用中心点簇半径范围内的条目构建簇；以及无副本，它限制每个条目只能属于一个簇。如图12所示，与这两种基线方法相比，SWARM 的延迟平均分别降低了 1.14× 和 1.33× 。这些结果凸显了在簇内捕获成对协同激活关系的重要性，并表明条目复制对于建模复杂的激活模式至关重要。

离线 SSD 放置。 我们在图 13中比较了 SWARM 与两种基线方法所实现的聚合带宽。

第一种方法是无集群，它不考虑共同激活关系，而是将词元依次放置到各个固态硬盘上。第二种方法是无均衡，它按照簇组织词元，但从单个固态硬盘开始放置，导致利用率不均衡。这些结果表明，SWARM 可将 I/O 延迟分别降低最多 2.63× 和 3.17×，从而有效地将聚类转化为跨固态硬盘的更高聚合带宽。

离线 DRAM 放置。 表4 评估了 DRAM 放置策略。除局部窗口外，SWARM 还会将簇中心点常驻于 DRAM 中。结果表明，随着前缀长度增加，SWARM 可将选择延迟降低 6.9×-12.2× ，且上下文长度越长，性能提升越显著。值得注意的是，这一改进仅带来极小的内存开销（上下文长度为 1M 时为 0.412 GB）。

在线检索。 我们评估在线调度算法——

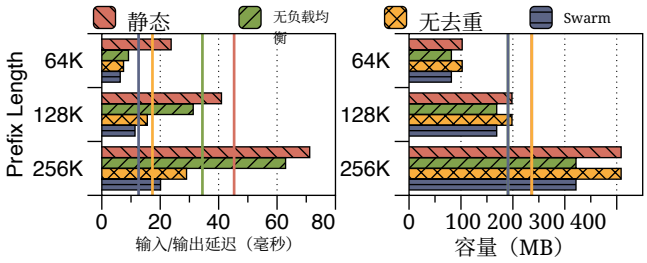

图 14：在四种检索策略下，Qwen3-32B 在不同前缀长度下的平均输入/输出延迟和容量。

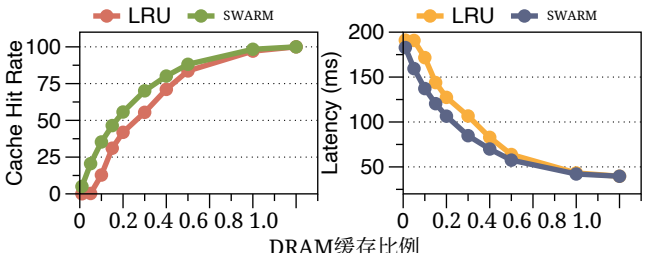

图 15：Octopus 和最近最少使用策略下，Qwen3-32B 在不同缓存比例下的缓存命中率和端到端延迟。

SWARM 在三种策略下的输入/输出延迟和输入/输出量。静态 采用静态多 SSD 路由，始终从第一个可用副本读取，因此既不进行去重，也不进行负载均衡。 无均衡 会执行去重，但忽略负载均衡，导致请求集中在部分固态硬盘上，使速度最慢的固态硬盘成为瓶颈。无去重会在各固态硬盘之间均衡分配请求，但不会移除集群副本引入的重复词元，因而导致读放大。图 14显示，与所有三种基线方法相比，SWARM 通过在运行时联合考虑复制和负载均衡，平均将输入/输出延迟降低了 2.57×，将输入/输出量降低了 1.24×。

在线集群更新。表 5评估了 SWARM 在解码过程中如何维持簇质量。我们测量簇条目与其中心点之间的平均距离，并将其归一化为离线计算的初始距离。作为基线方法，*Min-Size* 仅根据簇大小分配新词元，而 *Min-Diff* 则根据新词元与簇中心点的相似度进行分配。SWARM 的表现优于这两种基线方法，分别达到 8.0× 和 2.3×。通过跟踪局部窗口内的共激活模式，SWARM 能够有效近似新词元与现有簇之间不断变化的关系。

在线缓存更新。 图 15 展示了不同动态随机存取存储器缓存比例下的缓存命中率和每层延迟。在所有缓存预算下，SWARM 的表现始终优于最近最少使用（LRU） [9] 基线方法，缓存命中率提高了 74%，每层延迟平均降低了 13.6%。LRU 仅跟踪近期访问记录，因此可能会保留那些曾被访问一次、但很少被再次使用的大型簇。相比之下，SWARM 通过成本效益分数动态平衡激活频率和簇大小。

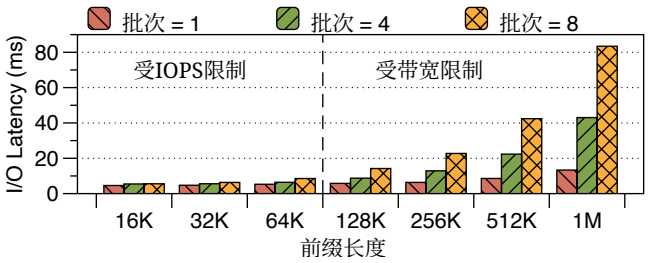

图 16：WikiText 上 Qwen3-32B 在不同批量大小下，跨前缀长度的平均 I/O 延迟。

8.4 敏感性分析

前缀长度。图16展示了批量大小为 1、4 和 8 时，不同前缀长度下的 I/O 延迟。前缀长度较小时，由于工作负载受每秒输入/输出操作数限制，延迟增长缓慢：I/O 请求数量有限，SSD 带宽未得到充分利用。随着前缀长度增加，系统转为受带宽限制，延迟随检索数据量近似线性增长。增大批量大小会增加并发 I/O 请求数量，使 SWARM 更早进入受带宽限制的状态。**SSD 类型**。我们在两种存储配置上评估了 SWARM 和基线方法：四块高端 PM9A3 SSD 和四块低端 900P SSD。图17 显示，在所有前缀长度下，SWARM 的吞吐量始终高于所有基线方法。在性能较低的 SSD 上，由于 IOPS 能力有限，系统在较短前缀（约 32 K）时便从受 IOPS 限制转为受带宽限制。相比之下，在性能较高的 PM9A3 SSD 上，这一转变发生在更长的前缀（约 64K）时，此时 SWARM 可达到更高的可用带宽上限。

表 4：Qwen3-32B 在两种 DRAM 放置策略下，不同前缀长度的选择延迟。

前缀长度	$s = 128K$	$s = 256K$	$s = 512K$	$s = 1M$
朴素延迟（毫秒）	1.635	3.136	6.176	12.15
SWARM 延迟（毫秒）	0.236	0.374	0.554	0.999
SWARM 内存（GB）	0.053	0.104	0.207	0.412

表 5：在 WikiText 上使用 Qwen3-32B 评估的簇条目与中心点之间的距离（相较于初始值）。

解码步骤 t	$= 1$	$t = 2$	$t = 4$	$t = 8$	$t = 16$	$t = 32$
Min-Size	1.733	2.481	3.957	6.834	12.004	21.376
Min-Diff	1.581	1.880	2.184	2.348	2.649	3.019
SWARM	1.001	1.001	1.001	1.002	1.005	1.012

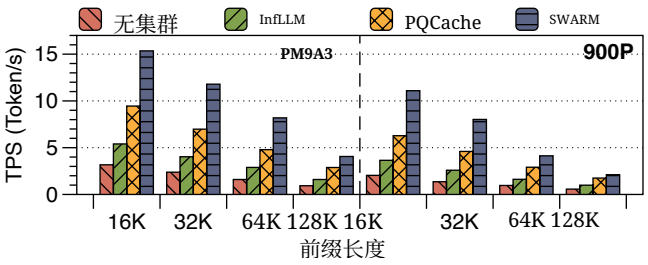

图 17：Qwen3-32B 在两种固态硬盘类型下，不同前缀长度的吞吐量对比。

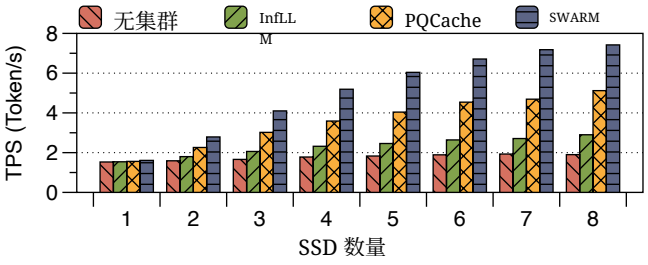

图 18：Qwen3-32B 在 WikiText（前缀长度为 128K）上使用不同数量固态硬盘时的吞吐量对比。

一转变发生在更长的前缀（约 64K）时，此时 SWARM 可达到更高的可用带宽上限。

固态硬盘数量。图 18 展示了固态硬盘数量从 1 增至 8 时的吞吐量。使用单块固态硬盘时，SWARM 会退回到基线方法。随着固态硬盘数量增加，其吞吐量稳步提升，并持续超过所有基线方法。这种可扩展性源于两个因素：离线放置能够灵活适应不同数量的固态硬盘，而在线调度则能有效平衡工作负载。

聚类阈值。我们研究了聚类半径 τ 对三个数据集上吞吐量的影响，如图 19所示。在每个子图中，τ 均根据相应的数据集进行选择。例如，图 19(a) 展示了 SWARM 使用 D1 作为离线数据集，并在 D1–D3 上进行评估的设置。结果表明，即使发生数据集偏移，SWARM 仍然有效，展现了聚类方法的稳健性。

稀疏率。 图 20 展示了不同稀疏率下的吞吐量。在稀疏率较低时，I/O 量较小，性能主要受每秒输入/输出操作数限制，因此带宽未得到充分利用。随着稀疏率提高，工作负载逐渐转为受带宽限制，吞吐量受最大可实现 I/O 带宽制约。在所有设置下，SWARM 的性能始终优于基线方法。通过有效利用聚合的多磁盘带宽，SWARM 在这两种情况下都实现了更高的吞吐量。

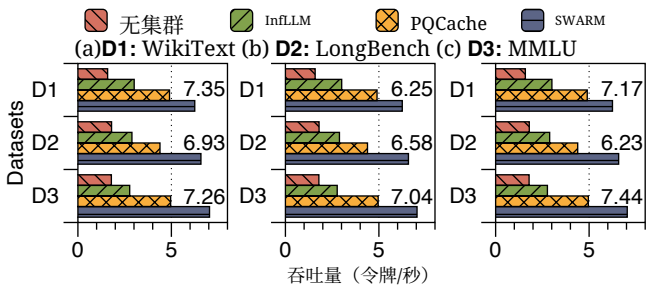

图 19：针对每个数据集进行校准的特定于数据集的 τ 吞吐量，在 Qwen3-32B 上评估（前缀长度为 128K）。

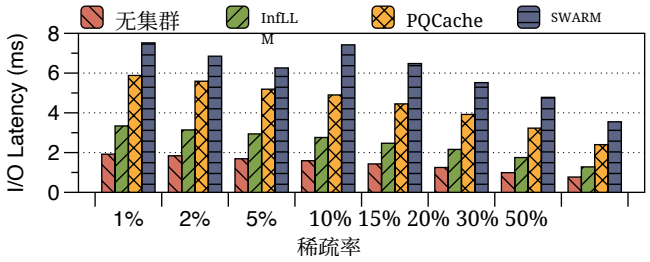

图 20：Qwen3-32B 在 WikiText 上（前缀长度为 128K）不同稀疏率下的 I/O 延迟对比。

9 相关工作

KVCache 稀疏性。近期研究日益关注长上下文大语言模型推理中 KVCache 的内在稀疏性。Quest [44] 提出了查询感知稀疏性，根据查询动态选择相关的 KV 页面。RetrievalAttention [29] 利用近似最近邻搜索，通过向量相似度检索 KV条目。HeadKV [13] 通过仅保留一部分重要的注意力头，引入了头级稀疏性。然而，仍有一项关键挑战：如何将这些洞见转化为实际推理系统中切实的性能提升。SWARM 通过算法与系统的协同设计弥合了这一差距。**KVCache 卸载**。在长上下文大语言模型推理中，KVCache 会造成巨大的内存压力，因此需要将 KVCache 卸载至动态随机存取存储器和固态硬盘。Flex-Gen [41] 提出了涵盖高带宽内存、动态随机存取存储器和固态硬盘的三级内存层次结构，并利用线性规划优化张量放置。SolidAttention [55] 采用将 KV条目交错存放于固态硬盘上的布局，并结合推测性预取，旨在实现高效的单节点部署。IMPRESS [7] 和 CachedAttention [15] 将这一研究方向扩展至云环境，根据词元重要性将 KV条目组织成块，并置于多级存储层次结构中。尽管取得了这些进展，先前研究仍指出固态硬盘带宽是主要瓶颈。相比之下，SWARM 通过利用多磁盘并行性解决了这一限制。

KVCache 聚类。在 KVCache 卸载中，KV条目的组织方式对于实现高效检索至关重要。ClusterKV [30] 基于静态相似度指标应用 K 均值聚类。InfLLM [47] 选择高分词元作为代表，并以粗粒度检索 KV条目。PQCache [53] 则将簇检索建模为向量搜索问题。这些方法主要依赖数学相似度或启发式重要性进行聚类。SWARM 根据共激活模式组织 KV条目，从而将其转化为切实的性能提升。

10 结论

我们提出了 SWARM，这是一种基于 SSD 的 KVCache 卸载框架，利用 KVCache 共激活，实现跨 SSD 的高带宽效率并行检索。SWARM 将受带宽限制的 KVCache 访问转化为可扩展的并行 I/O，为大语言模型推理打造成本高效且容量可扩展的存储系统铺平了道路。

参考文献

[1] Sandhini Agarwal、Lama Ahmad、Jason Ai、Sam Altman、Andy Applebaum、Edwin Arbus、Rahul K Arora、Yu Bai、Bowen Baker、Haiming Bao 等。gpt-oss-120b 和 gpt-oss-20b 模型卡。*arXiv* 预印本 *arXiv:2508.10925*，2025 年。[2] Joshua Ainslie、James Lee-Thorp、Michiel De Jong、Yury Zemlyanskiy、Federico Lebrón 和 Sumit Sang- hai。GQA：基于多头检查点训练广义多查询变换器模型。载于*2023* 年自然语言处理实证方法会议论文集，第 4895–4901 页，2023 年。[3] Anthropic。Claude 的提示缓存。https://www.anthropic.com/news/prompt-caching，2024 年。Anthropic 博客文章。[4] Jens Axboe。liburing。https://github.com/axboe/liburing，2026 年。访问日期：2026 年 3 月 17 日。[5] Yushi Bai、Xin Lv、Jiajie Zhang、Hongchang Lyu、Jiankai Tang、Zhidian Huang、Zhengxiao Du、Xiao Liu、Aohan Zeng、Lei Hou 等。LongBench：面向长上下文理解的双语多任务基准。载于计算语言学协会第 *62* 届年会论文集（第 *1* 卷：长篇论文），第 3119–3137 页，2024。[6] Joya Chen、Zhaoyang Lv、Shiwei Wu、Kevin Qinghong Lin、Chenan Song、Difei Gao、Jia-Wei Liu、Ziteng Gao、Dongxing Mao 和 Mike Zheng Shou。VideoLLM-online：面向流式视频的在线视频大语言模型。载于*IEEE/CVF* 计算机视觉与模式识别会议论文集，第 18407–18418 页，2024。

[7] Weijian Chen、Shuibing He、Haoyang Qu、Ruidong Zhang、Siling Yang、Ping Chen、Yi Zheng、Baoxing Huai 和 Gang Chen。{IMPRESS}：一种面向大语言模型推理的 {重要性驱动}{多层} 前缀 {键值} 存储系统。载于第 *23* 届 *USENIX* 文件与存储技术大会（*FAST 25*），第 187–201 页，2025 年。

[8] Karl Cobbe、Vineet Kosaraju、Mohammad Bavarian、Mark Chen、Heewoo Jun、Lukasz Kaiser、Matthias Plappert、Jerry Tworek、Jacob Hilton、Reiichiro Nakano、Christopher Hesse 和 John Schulman。训练验证器来解答数学文字题。*arXiv* 预印本 *arXiv:2110.14168*，2021 年。

[9] Peter J Denning. 程序行为的工作集模型。*ACM* 通讯，11(5):323–333，1968。[10] Yanhao Dong、Yubo Miao、Weinan Li、Xiao Zheng、Chao Wang、Jiesheng Wu 和 Feng Lyu。通过异步键值缓存预取提升大语言模型推理吞吐量。*arXiv* 预印本 *arXiv:2504.06319*，2025。[11] Wenqi Fan、Yujuan Ding、Liangbo Ning、Shijie Wang、Hengyun Li、Dawei Yin、Tat-Seng Chua 和 Qing Li。关于检索增强生成与大语言模型相结合的综述：迈向检索增强型大语言模型。载于第 *30* 届 *ACM SIGKDD* 知识发现与数据挖掘大会论文集，第 6491–6501 页，2024。[12] Shaoting Feng、Hanchen Li、Kuntai Du、Zhuohan Gu、Yuhan Liu、Jiayi Yao、Siddhant Ray、Samuel Shen、Yihua Cheng、Ganesh Ananthanarayanan 等。AdaptCache：用于低延迟、高质量语言模型服务的原生键值缓存存储层级。*arXiv* 预印本 *arXiv:2509.00105*，2025。[13] Yu Fu、Zefan Cai、Abedelkadir Asi、Wayne Xiong、Yue Dong 和 Wen Xiao。并非所有注意力头都同等重要：一种集成检索与推理的头级键值缓存压缩方法。*arXiv* 预印本 *arXiv:2410.19258*，2024。[14] Bin Gao、Zhuomin He、Puru Sharma、Qingxuan Kang、Djordje Jevdjic、Junbo Deng、Xingkun Yang、Zhou Yu 和 Pengfei Zuo。{成本高效的} 多轮对话大语言模型服务，采用{CachedAttention}。载于*2024* 年 *USENIX* 年度技术大会（*USENIX ATC 24*），第 111–126 页，2024。[15] Bin Gao、Zhuomin He、Puru Sharma、Qingxuan Kang、Djordje Jevdjic、Junbo Deng、Xingkun Yang、Zhou Yu 和 Pengfei Zuo。{成本高效的} 多轮对话大语言模型服务，采用

{CachedAttention}。载于*2024* 年 *USENIX* 年度技术大会（*USENIX ATC 24*），第 111–126 页，2024。

[16] In Gim、Guojun Chen、Seung-seob Lee、Nikhil Sarda、Anurag Khandelwal 和 Lin Zhong。提示缓存：用于低延迟推理的模块化注意力机制复用。机器学习与系统会议论文集，6:325–338，2024。

[17] Google DeepMind。Gemini 3 Pro 模型卡。技术报告，Google DeepMind，2025年11月。访问日期：2025年12月。

[18] Shibo Hao、Tianyang Liu、Zhen Wang 和 Zhiting Hu。Toolkengpt：通过工具嵌入为冻结语言模型扩展海量工具。神经信息处理系统进展，36:45870–45894，2023年。

[19] Dan Hendrycks、Collin Burns、Steven Basart、Andy Zou、Mantas Mazeika、Dawn Song 和 Jacob Steinhardt。衡量大规模多任务语言理解。国际学习表征会议论文集（*ICLR*），2021年。

[20] Chaoyi Jiang、Lei Gao、Hossein Entezari Zarch 和 Murali Annavaram。借助输入/输出感知的部分键值缓存重新计算，实现高效的大语言模型推理。*arXiv* 预印本 *arXiv:2411.17089*，2024。[21] Jiachen Jiang、Jinxin Zhou 和 Zhihui Zhu。追踪表征演进：分析并增强逐层相似度。*arXiv* 预印本 *arXiv:2406.14479*，2024。[22] Carlos E Jimenez、John Yang、Alexander Wettig、Shunyu Yao、Kexin Pei、Ofir Press 和 Karthik Narasimhan。SWE-bench：语言模型能否解决现实世界中的 GitHub 问题？*arXiv* 预印本 *arXiv:2310.06770*，2023年。[23] Chao Jin、Zili Zhang、Xuanlin Jiang、Fangyue Liu、Shu- fan Liu、Xuanzhe Liu 和 Xin Jin。Ragcache：面向检索增强生成的高效知识缓存。《*ACM* 计算机系统汇刊》，44(1):1–27，2025。[24] Hyunjoo Jin。TrendForce 预计第一季度芯片价格较上一季度飙升 90–95%。路透社，2026 年 2 月。报道称，TrendForce 预测，常规动态随机存取存储器合约价格将在 2026 年第一季度环比上涨 90–95%。[25] 金士顿科技。《闪存指南》，2012年。文档 MKF-283US。存档 PDF（2013-10-19 快照）。[26] Jan V Leeuwen。《理论计算机科学手册：算法与复杂性》。麻省理工学院出版社，1990年。

[27] Jiaqi Li、Mengmeng Wang、Zilong Zheng 和 Muhan Zhang。Loogle：长上下文语言模型能理解长上下文吗？载于计算语言学协会第 *62* 届年会论文集（第 *1* 卷：长篇论文），第 16304–16333 页，2024年。

[28] Aixin Liu、Bei Feng、Bing Xue、Bingxuan Wang、Bochao Wu、Chengda Lu、Chenggang Zhao、Chengqi Deng、Chenyu Zhang、Chong Ruan 等。DeepSeek-V3 技术报告。*arXiv* 预印本 *arXiv:2412.19437*，2024。

[29] Di Liu、Meng Chen、Baotong Lu、Huiqiang Jiang、Zhen- hua Han、Qianxi Zhang、Qi Chen、Chengruidong Zhang、Bailu Ding、Kai Zhang、Chen Chen、Fan Yang、Yuqing Yang 和 Lili Qiu。RetrievalAttention：通过向量检索加速长上下文大语言模型推理，2024。

[30] Guangda Liu、Chengwei Li、Jieru Zhao、Chenqi Zhang 和 Minyi Guo。ClusterKV：在语义空间中操纵大语言模型键值缓存以实现可召回压缩。*arXiv* 预印本 *arXiv:2412.03213*，2024。[31] J MacQueen。多元观测。载于 第五届伯克利数学统计与概率学研讨会论文集，第 1 卷，第 281–297 页，1967年。[32] Stephen Merity、Caiming Xiong、James Bradbury 和 Richard Socher。指针哨兵混合模型，2016年。[33] Meta AI。Llama 3.1 模型系列。https://ai.meta.com/llama/，2024。访问日期：2026-03-18。[34] 英伟达公司。*NVIDIACUDA C++*编程指南，2026。13.0版。[35] NVM Express公司。NVM Express。https://nvmexpress.org/，2017。检索日期：2017-01-24。（网站存档于2019-12-05。）[36] OpenAI。提示缓存。https://developers.openai.com/api/docs/guides/prompt-caching，2024。OpenAI API 文档。[37] Xiurui Pan、Endian Li、Qiao Li、Shengwen Liang、Yizhou Shan、Ke Zhou、Yingwei Luo、Xiaolin Wang 和 Jie Zhang。InstInfer：面向高成本效益长上下文大语言模型推理的存储内注意力卸载。*arXiv* 预印本 *arXiv:2409.04992*，2024。[38] PCI-SIG。PCI Express® 基础规范第 5.0 版。规范修订版 5.0，1.0版，PCI特别兴趣小组，2019年5月。1.0版。

[39] Ruoyu Qin、Zheming Li、Weiran He、Jialei Cui、Feng Ren、Mingxing Zhang、Yongwei Wu、Weimin Zheng 和 Xinran Xu。《Mooncake：以更多存储换取更少计算——一种以 {KVCache} 为核心的架构

用于服务 {大语言模型} 聊天机器人。在第 *23* 届 *USENIX* 文件与存储技术研讨会（*FAST 25*），第 155–170 页，2025 年。

[40] Yujia Qin、Shihao Liang、Yining Ye、Kunlun Zhu、Lan Yan、Yaxi Lu、Yankai Lin、Xin Cong、Xiangru Tang、Bill Qian等。Toolllm：帮助大语言模型掌握 16000+ 现实世界应用程序接口。*arXiv*预印本 *arXiv:2307.16789*，2023年。

[41] Ying Sheng、Lianmin Zheng、Binhang Yuan、Zhuohan Li、Max Ryabinin、Beidi Chen、Percy Liang、Christopher Ré、Ion Stoica和Ce Zhang。Flexgen：使用单个图形处理器对大语言模型进行高吞吐量生成式推理。载于国际机器学习会议，第31094–31116页。PMLR，2023年。

[42] Aaditya Singh、Adam Fry、Adam Perelman、Adam Tart、Adi Ganesh、Ahmed El-Kishky、Aidan McLaughlin、Aiden Low、AJ Ostrow、Akhila Ananthram等。OpenAI GPT-5系统卡。*arXiv*预印本 *arXiv:2601.03267*，2025。

[43] Jaspreet Singh 和 Zaheer Kachwala。存储芯片价格飙升，消费电子制造商前景黯淡。路透社，2026年1月。本文讨论了存储器（包括动态随机存取存储器）价格上涨和供应短缺如何给消费电子行业带来压力，并提及存储器价格可能进一步上涨的预期。

[44] Jiaming Tang、Yilong Zhao、Kan Zhu、Guangxuan Xiao、Baris Kasikci 和 Song Han。Quest：用于高效长上下文大语言模型推理的查询感知稀疏性。*arXiv* 预印本 *arXiv:2406.10774*，2024。

[45] Kimi 团队、Yifan Bai、Yiping Bao、Guanduo Chen、Jiahaa Chen、Ningxin Chen、Ruijue Chen、Yanru Chen、Yuankun Chen、Yutian Chen 等。Kimi K2：开放式智能体智能。*arXiv* 预印本 *arXiv:2507.20534*，2025。

[46] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。注意力机制就是你所需要的一切。神经信息处理系统进展，30，2017年。

[47] Chaojun Xiao、Pengle Zhang、Xu Han、Guangxuan Xiao、Yankai Lin、Zhengyan Zhang、Zhiyuan Liu 和 Maosong Sun。InfLLM：无需训练，利用高效的上下文记忆为大语言模型扩展长上下文。神经信息处理系统进展，37：119638–119661，2024。

[48] Junbin Xiao、Xindi Shang、Angela Yao 和 Tat-Seng Chua。Next-qa：问答的下一阶段——解释时间动作。载于*IEEE/CVF* 计算机视觉与模式识别会议论文集，第 9777–9786 页，2021 年。

[49] An Yang、Anfeng Li、Baosong Yang、Beichen Zhang、Binyuan Hui、Bo Zheng、Bowen Yu、Chang Gao、Chengen Huang、Chenxu Lv 等。Qwen3 技术报告。*arXiv* 预印本 *arXiv:2505.09388*，2025 年。

[50] Shunyu Yao、Dian Yu、Jeffrey Zhao、Izhak Shafran、Tom Griffiths、Yuan Cao 和 Karthik Narasimhan。思维树：利用大语言模型进行审慎的问题求解。神经信息处理系统进展，36：11809–11822，2023年。

[51] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Karthik R Narasimhan 和 Yuan Cao。React：在语言模型中协同推理与行动。载于第十一届国际学习表征会议，2022年。

[52] Ahmet Caner Yüzügüler、Jiawei Zhuang 和 Lukas Cavigelli。Preserve：分布式大语言模型服务中的模型权重与键值缓存预取。*arXiv* 预印本 *arXiv:2501.08192*，2025。

[53] Hailin Zhang、Xiaodong Ji、Yilin Chen、Fangcheng Fu、Xupeng Miao、Xiaonan Nie、Weipeng Chen 和 Bin Cui。PQCache：面向长上下文大语言模型推理的基于乘积量化的 KVCache。*ACM*数据管理会议录，3(3):1–30，2025。

[54] Zhenyu Zhang、Ying Sheng、Tianyi Zhou、Tianlong Chen、Lianmin Zheng、Ruisi Cai、Zhao Song、Yuandong Tian、Christopher Ré、Clark Barrett 等。H2o：面向大语言模型高效生成式推理的重度命中预言机。神经信息处理系统进展，36：34661–34710，2023年。

[55]Xinrui Zheng、Dongliang Wei、Jianxiang Gao、Yixin Song、Zeyu Mi 和 Haibo Chen。{SolidAttention}：{低延迟}{基于固态硬盘的}服务，面向{内存受限的}{个人电脑}。载于第 *24* 届 *USENIX*文件与存储技术会议（*FAST 26*），第 67–82 页，2026 年。