广州广电五舟科技股份有限公司

GUANGZHOU GUANGDIAN WUZHOU TECHNOLOGY CO., LTD.

性 能 测 试 报 告

PERFORMANCE TEST REPORT

AISSD5000 全闪存储用于大模型推理 KV Cache 分层的性能测试报告

（14B 模型·显存效益与介质对照）

报 告 编 号	AISSD5000-KVC-PERF-2026-004
版 本	V1.0
委 托 单 位	深圳中科航星科技有限公司
被 测 设 备	AISSD5000 全闪 NVMe-oF 存储阵列
测 试 类 别	性能基准测试
签 发 日 期	2026 年 07 月 03 日

编制		**日期**	
审核		日期	
批准		日期	

（此处加盖检测机构公章）

广州广电五舟科技股份有限公司

声　明

1. 本报告无检测机构公章及骑缝章无效。
2. 本报告无编制人、审核人、批准人签字无效。
3. 本报告涂改、增删无效。
4. 本报告所列结果仅对本次测试所述的被测设备、软件版本、配置及负载条件有效。
5. 本测试为单机环境下的性能基准测试；跨节点共享、容量弹性扩展等架构能力不在本次测试范围内。
6. 对本报告若有异议，应于收到报告之日起十五日内向本机构提出。

检测机构	广州广电五舟科技股份有限公司
委托单位	深圳中科航星科技有限公司
报告编号	AISSD5000-KVC-PERF-2026-004
签发日期	2026 年 07 月 03 日

报告基本信息

项目	内容
报告标题	AISSD5000 全闪存储用于 LLM 推理 KV Cache 分层的性能测试报告（14B 模型·显存效益与介质对照）
报告编号	AISSD5000-KVC-PERF-2026-004
版本	V1.0
检测机构	广州广电五舟科技股份有限公司
委托单位	深圳中科航星科技有限公司
被测设备（DUT）	AISSD5000 全闪 NVMe-oF 存储阵列
测试类别	性能基准测试
密级	内部

修订记录

版本	日期	说明
V1.0	2026-07-03	首次发布：基于 Qwen2.5-14B 大模型（单会话 KV 约 1.55 GB）的七组对照测试，覆盖 AISSD5000+GDS、本地 NVMe+GDS、无外存重算（两档显存）与 AISSD5000 低显存直读档

1. 执行摘要

本报告在受控的单机环境下，以 **14B 参数量大模型（Qwen2.5-14B-Instruct）** 为负载，量化评估 AISSD5000 全闪存储作为大语言模型（LLM）推理 KV Cache（键值缓存）分层后备介质的性能价值。相比此前 7B 模型的测试（报告编号 -003），14B 模型的单会话 KV 体量增至约 1.55 GB，更贴近生产环境中长上下文、大模型的真实负载压力。测试采用单一变量法，在完全一致的工作负载与测量口径下，对比七组配置：AISSD5000+GPUDirect 直读（含访问参数调优前后两组）、本地 NVMe+GPUDirect 直读（同样两组）、无外存全量重算（标准显存与加满显存两档）、以及 AISSD5000 低显存直读档（显存效益验证）。

主要结论（各组请求全部成功、磁盘组逐请求物理命中取证成立）：

1. **显存等效替代效益（核心结论）——达到同等会话恢复能力，AISSD5000 方案等效替代约 128 GB 显存（约合 2 张 N260 加速卡），且替代量随会话规模线性增长**。推理链共三步，每步均有实测支撑。**第一步**，设定服务水平目标（SLO）：冷会话恢复首字延迟 ≤ 8 s（交互式服务可容忍线；SLO 在 8–14 s 间任意取值结论不变）。**第二步**，实测证明重算方案在单卡上无论配多少显存都无法达标：显存预算 0.65（实测占用 43,735 MiB）与 0.90（占用 60,055 MiB、GPU KV 容量高 1.3 倍）首字延迟同为 14.1 s——冷会话的 KV 本就不在显存中，重算的瓶颈是预填充算力而非 KV 容量，加显存零收益。因此重算方案达标的**唯一**途径是让 KV 始终驻留显存，即显存须装下全部 **140 GB** 的 KV 工作集（90 会话 × 1.55 GB，实测落盘取证），加 28 GB 模型权重共需约 **168 GB 显存（约合 3 张 N260）**。**第三步**，AISSD5000 方案以**单卡 40,527 MiB（39.6 GB）显存**实测取得首字延迟 6.264 s、输出吞吐 32.4 tok/s，直接达标。两相比较：**168 GB − 40 GB ≈ 128 GB，即 AISSD5000 的显存等效替代量（约合 2 张 N260）**。由于替代量等于 KV 工作集大小，会话池越大、上下文越长，等效替代的显存越多、上不封顶——这正是"以存力换显存"的量化含义。辅证：14B 模型在 `0.50` 显存预算下甚至无法启动服务（权重占用后剩余空间不足以容纳 32K 上下文的最低 KV），显存越紧张，KV 外置的价值越大。
2. **相比本地 NVMe：远程 NVMe-oF 零性能损耗，最优配置下与本地盘等速，默认配置下反超**。GDS 直读 8 线程（调优最优）配置下，AISSD5000 与本地盘打平（TTFT p50 5.653 s vs 5.581 s，差异约 1%，输出吞吐同为 45.5 tok/s），且 AISSD5000 物理读峰值更高（2.68 vs 2.51 GB/s）；GDS 默认（4 线程）配置下 AISSD5000 全面领先本地盘（TTFT 快 12%、吞吐高 7%、物理读带宽高 13%）。即：跨 100 GbE 网络访问 AISSD5000 与访问本机 PCIe 盘的推理体验没有差别，同时获得本地盘不具备的容量弹性、多机共享与存算分离能力。
3. **相比无外存重算：首字延迟快 2.5 倍、输出吞吐高 2.35 倍**。同等显存预算（`0.65`）下，AISSD5000+GDS（TTFT p50 5.653 s、45.5 tok/s）对比重算（14.137 s、19.4 tok/s），首字延迟降低 60%（快 2.50 倍）、p90 尾延迟降低 62%（8.869 s vs 23.636 s）、输出吞吐提升 135%、整批完成时间缩短 57%（11.0 s vs 25.9 s）。KV 复用直接免去 14B 模型对约 8,448-token 长前缀的重复预填充计算。
4. **多卡条件下 AISSD5000 的优势将进一步扩大（理论分析+同型平台旁证）**。单卡推理引擎逐请求串行取数，仅消费了 AISSD5000 供给能力的约三分之一（本机阵列脱离推理的裸读能力实测 6.5–7.8 GB/s，单卡推理内峰值消费仅 2.68 GB/s）；多卡部署时每卡为独立取数管道，聚合读取需求随卡数近线性增长，AISSD5000 的多盘聚合与网络带宽余量得以兑现，而本地单盘带宽/容量固定且无法被多节点共享。同型阵列在 8 卡平台的内部实测显示聚合带宽由单实例 5.2 GB/s 增至 8 实例 10.18 GB/s

且盘仍有余力，而本地单盘 6.5 GB/s 即达物理极限。此外，AISSD5000 上的 KV 池可被全集群共享（一次预填充、多卡复用），显存等效替代效益随集群规模放大——多卡各自的会话工作集共用一份外置 KV 池，无需在每张卡上重复驻留。详见 §7.4。

适用边界说明： 本测试为单机、单卡环境下的性能基准测试；多卡聚合与跨节点共享的结论 4 为基于实测数据的理论外推与同型平台旁证，其直接物理验证列为后续工作。

2. 测试目的与范围

2.1 测试目的

回答以下决策问题：

1. 在 14B 大模型、显存紧张的现实约束下，使用 AISSD5000 承接 KV Cache 相比"给 GPU 堆显存 + 重算"，能节省多少显存、换来多大性能？
2. AISSD5000（远程 NVMe-oF）与本地 NVMe 作为 KV 后备介质，经 GPUDirect 直读路径的端到端性能差异如何？
3. 当 KV Cache 无法驻留 GPU 显存时，AISSD5000 回读相比重新计算能带来多大的延迟与吞吐收益？
4. 多卡部署条件下，AISSD5000 相对本地盘的优势将如何变化？

2.2 测试范围

- **范围内：** 单机、单卡（MetaX N260）环境下，14B 模型的 KV Cache 七组配置对照（AISSD5000+GDS 两组 / 本地 NVMe+GDS 两组 / 重算两档显存 / AISSD5000 低显存直读档）对推理延迟（TTFT）、吞吐与显存占用的影响；GDS 访问参数（IO 线程数）的调优。
- **范围外：** 多卡/跨节点的物理验证、异步预取协同、真实业务偏斜流量——列为后续工作（§9）。

3. 术语与指标定义

指标	定义	优劣方向
KV Cache	LLM 推理 prefill 阶段产生的键值中间结果，可复用以跳过重复计算	—
显存占用	起服就绪后 `mx-smi` 实测整卡显存占用（含权重、引擎 KV 池、GDS 暂存池等全部开销）	同效能下越低越好
GPU KV 容量	推理引擎日志报告的显存内 KV 缓存容量（token 数），由显存预算决定	—
TTFT（首字延迟）	从提交请求到收到首个输出 token 的时间	越低越好
输出吞吐（tok/s）	单位时间产出的输出 token 数	越高越好
p50 / p90	延迟的第 50/90 百分位（尾延迟衡量最差体验）	越低越好
GPUDirect Storage（GDS）	存储数据经 O_DIRECT 直接读入 GPU 显存，绕过 OS 页缓存与 CPU 二次拷贝的技术	—

GDS IO 线程数	GDS 后端并行读取 KV 分块文件的线程数（`gds_io_threads`，默认 4）	存在平台相关最优值
need-to-load	LMCache 日志中需从下层（磁盘/GDS）加载的 token 数；>0 表示非纯 GPU 命中	—
物理读带宽	`iostat` 实测块设备读取速率（测量前已清页缓存，反映真实磁盘读）	越高越好
重算（recompute）	无外部 KV 存储时，对历史前缀重新执行 prefill 计算	性能下限

> 说明：延迟类指标统一采用 OpenAI 兼容流式接口、temperature=0；并发度指同时在飞（in-flight）的请求数。

4. 被测系统（System Under Test）

4.1 SUT 边界

被测对象为"推理服务 + KV 分层存储"的端到端系统；受控变量为 KV Cache 后备来源、访问参数与显存预算。推理引擎、模型、采样参数、并发度、序列长度在各组间保持一致。

4.2 硬件与被测设备

组件	配置
GPU	沐曦 MetaX N260，64 GB 显存
CPU / 内存	海光 Hygon C86-4G，256 核 / 8 NUMA 节点，251 GB
被测设备（DUT）	AISSD5000 全闪存储，4×960 GB NVMe 盘组 RAID0 = 3.5 TB 阵列（`/dev/md0`），xfs 文件系统，经 NVMe-oF / RoCEv2 接入，挂载于 `/mnt/ws5000`
对照存储	本地 NVMe（`/dev/nvme0n1p1`，3.84 TB，ext4，挂 `/data`）；无后备（重算）
网络	100 GbE，RoCEv2

4.3 软件栈与版本

组件	版本
操作系统	Ubuntu 24.04 LTS，内核 6.8.0-31-generic
推理引擎	vllm-metax 0.20.0
KV 缓存库	LMCache 0.4.4（分层：GPU → CPU 内存 → 磁盘 / GDS）
GDS 库	沐曦 `libmcfile.so`（GPUDirect Storage C 库），经自研 `cufile` ctypes 适配层桥接给 LMCache

模型	Qwen2.5-14B-Instruct（bfloat16，权重约 28 GB）
容器镜像	`cr.metax-tech.com/public-ai-release/maca/vllm-metax:0.20.0-maca.ai3.7.0.107-torch2.8-py310-ubuntu22.04-amd64`
关键运行参数	`--max-model-len 32768`、`--no-enable-prefix-caching`（强制冷读取证）、`LMCACHE_CHUNK_SIZE=256`、`LMCACHE_GDS_BUFFER_SIZE=14336`（GDS 档）、`MACA_GRAPH_LAUNCH_MODE=1`、`PYTHONHASHSEED=0`

4.4 GPUDirect 集成（MetaX 平台）

LMCache 期望名为 `cufile` 的 Python 绑定（NVIDIA GDS 接口）。MetaX 平台仅提供 C 库 `libmcfile.so`。本测试沿用自研 `cufile` 适配层桥接（与 -003 号报告一致）：预加载运行时依赖以解决符号缺失；对文件句柄强制 `O_DIRECT`；将 `mcFileDriverOpen/HandleRegister/Read/Write/BufRegister/Deregister` 映射为 LMCache 所需的 `CuFile` 语义。适配层源码要点见附录 C。AISSD5000（xfs）与本地盘（ext4）均已实测确认 GDS 真正启用、未降级。

4.5 工作负载与容量基准

- 模型为 Qwen2.5-14B-Instruct（48 层、GQA 8 KV 头、head_dim 128、bfloat16），**KV 体量约 192 KB/token**。
- 单会话系统提示前缀约 **8,448 token**，单会话 KV 约 **1.55 GB**——为 7B 模型（0.90 GB/26K token 会话）的 1.7 倍，负载压力更贴近生产大模型场景。
- 灌入 90 个互异会话（工作集约 **140 GB**，实测落盘 142 GB），远超显存与 GDS 暂存池容量，保证测量会话必然溢出至存储层。
- 测量轮参数：**单波 8 个请求、并发 8（N8/conc8）、每请求生成 64 个输出 token、每会话访问一次、temperature=0**，访问会话 0–7（灌入次序最早、必然已被逐出显存）。
- **规模选择依据（N=8）：** GDS 显存暂存池受当前 LMCache 版本"读取缓冲用后不即时回收"限制，单次测量工作集须 ≤ 池容量。14B 单会话 KV 1.55 GB、池 14 GiB → 上限约 9 会话，取 N=8 保证 GDS 档每请求完整回读、生成正确（实测分配失败为 0）。全部七组统一 N=8 以保证可比。

5. 测试方法学

5.1 测试设计（七组对照）

组	KV 后备	访问方式	GDS IO 线程	显存预算 util	定位
① AISSD5000+GDS·调优	AISSD5000（md0）	O_DIRECT 盘→显存直读	**8（调优最优）**	0.65	被测主档
② AISSD5000+GDS·默认	AISSD5000（md0）	同上	4（默认值）	0.65	默认配置参考

③ 本地NVMe+GDS·调优	本地盘（nvme0n1）	同上	8	0.65	介质对照
④ 本地NVMe+GDS·默认	本地盘（nvme0n1）	同上	4	0.65	介质对照
⑤ 重算·同显存	无	—（重新计算）	—	0.65	下限基线
⑥ 重算·显存加满	无	—	—	**0.90**	"堆显存"竞品方案
⑦ AISSD5000·低显存	AISSD5000（md0）	O_DIRECT 并行读引擎（host 路径，零显存池开销）	—	**0.60**	显存效益档

> ⑦ 采用自研 O_DIRECT 多线程并行读引擎（经 CPU 内存中转、不占用 GDS 显存暂存池），用于在**最低显存预算**下验证 AISSD5000 的承接能力；其读取路径源码见附录 D。GDS IO 线程数经扫描调优确定 8 为本平台最优（默认 4 为参考对照），更高线程数因平台内存子系统争抢反而劣化，不纳入本报告。

5.2 工作负载

- 每会话前缀约 8,448 token（模拟多轮会话历史），用户问题独立；数据为合成负载（固定 token 数），保证可复现。
- 测量轮：单波 8 请求同时到达、并发 8、decode=64、每会话读一次——模拟"一批冷会话同时恢复"的突发场景（服务重启暖机、会话迁移、故障切换）。

5.3 预处理与稳态

1. **灌入（populate）：** 存储组（①②③④⑦）先以 decode=1 对 90 个会话执行 prefill，使 KV 溢出落盘（约 142 GB）；重算组（⑤⑥）无需灌入。
2. **清除页缓存：** 测量前在宿主执行 `sync; echo 3 > /proc/sys/vm/drop_caches`，排除 251 GB 主机内存对磁盘读的掩盖。
3. **冷会话保证：** 测量访问会话 0–7（灌入次序最早），叠加 `--no-enable-prefix-caching` 与仅 4 GB 的 CPU 中转层（⑦ 为 32 GB），确保 KV 不在显存、不在内存，必然从物理盘读取。

5.4 测量与数据采集

- 客户端经 OpenAI 兼容流式接口测量逐请求 TTFT，聚合为 p50/p90/均值与输出吞吐。
- 并行采集：`iostat` 块设备物理读带宽；LMCache 日志逐请求 need-to-load 与 retrieve 吞吐；起服就绪后 `mx-smi` 实测显存占用与引擎日志 GPU KV 容量。
- 每组测试前以 `docker restart` 彻底释放显存（规避引擎子进程残留、驱动不回收显存问题）。

5.5 方法学控制（防数据污染）

1. **强制物理读取证：** 存储组全部请求 need-to-load>0（无一 GPU/内存命中），`iostat` 物理读带宽与 LMCache retrieve 吞吐互相吻合且远低于内存速度区间；重算组 need-to-load=0 且盘读≈0，反证纯重算。

2. **GDS 暂存池匹配：** N=8（12.4 GB）< 池 14 GiB，各 GDS 组显存池分配失败为 0、每请求完整回读。
3. **单一变量校验：** 每次启动后核验环境变量（GDS 开关/线程数/磁盘目录/显存预算）与 LMCache 配置日志回显。
4. **正确性：** 各组 8/8 请求全部成功返回，同一提示词、temperature=0、相同 decode 上限，输出 token 计数一致。

6. 测试结果

6.1 七组对照总表

测量条件：14B 模型，每会话 8,448 token / KV 1.55 GB，单波 N8 并发 8，decode=64，存储组已清页缓存、逐请求物理命中。

指标	① AISSD+GDS·8 线程	② AISSD+GDS·4 线程	③ 本地 +GDS·8 线程	④ 本地 +GDS·4 线程	⑤ 重算 ·0.65	⑥ 重算 ·0.90	⑦ AISSD· 低显存 0.60
TTFT p50（s）	**5.653**	6.797	5.581	7.685	14.137	14.122	**6.264**
TTFT p90（s）	8.869	10.531	9.339	11.732	23.636	22.299	13.294
TTFT 均值（s）	5.491	6.563	4.974	6.860	13.114	12.976	7.746
输出吞吐（tok/s）	**45.5**	40.2	45.5	37.6	19.4	20.1	**32.4**
整批完成时间（s）	11.0	12.6	11.3	13.7	25.9	25.1	15.9
实测显存占用（MiB）	58,071	58,071	58,071	58,071	43,735	**60,055**	**40,527**
GPU KV 容量（token）	66,080	66,080	66,080	66,080	66,080	153,168	48,672
物理读峰值（GB/s）	md0 **2.68**	md0 1.98	nvme 2.51	nvme 1.75	≈0	≈0	md0 1.97
每请求回读带宽（GB/s）	2.4–2.6	1.6–2.0	2.3–2.5	≈1.6	—	—	1.25–1.52
KV 来源	AISSD→显存直读	AISSD→显存直读	本地→显存直读	本地→显存直读	重新计算	重新计算	AISSD（CPU中

							转）
命中取证	need>0 全部	need>0 全部	need>0 全部	need>0 全部	need=0	need=0	need>0 全部
池分配失败	0	0	0	0	—	—	—

6.2 灌入阶段数据

组	灌入会话数	耗时（s）	落盘量
① AISSD+GDS·8线程	90	364.7	AISSD5000 约 142 GB
② AISSD+GDS·4线程	90	365.2	AISSD5000 约 142 GB
③ 本地+GDS·8线程	90	378.9	本地盘约 142 GB
④ 本地+GDS·4线程	90	378.2	本地盘约 142 GB
⑤⑥ 重算	—（无需灌入）	—	0
⑦ AISSD·低显存	90	245.4	AISSD5000 约 140 GB

6.3 核心对比一：显存等效替代效益（⑤⑥⑦ 三组联合论证）

结论：达到同等会话恢复能力，AISSD5000 方案等效替代约 128 GB 显存（约合 2 张 N260），替代量随会话规模线性增长。

6.3.1 为什么这样对比：显存等效替代量的定义与推理链

"省多少显存"必须回答一个前提问题：**省下的显存原本是用来买什么的？** 答案是"会话恢复能力"——让曾经服务过的会话（KV 已被计算过）在再次到来时快速恢复，而不是十几秒的重新计算。因此正确的比较不是两个配置的显存差值，而是：**在同一服务水平目标（SLO）下，两种技术路线各自需要多少显存。**本节按三步推理，每一步对应一组实测数据：

第一步：设定 SLO。 冷会话恢复首字延迟 ≤ 8 s（交互式服务的可容忍线）。该阈值取 8–14 s 之间任意值均不改变下文结论（重算方案实测 14.1 s，始终不达标；AISSD5000 方案实测 6.3 s，始终达标）。

第二步：重算路线需要多少显存才能达标？ 先看实测——加显存对重算完全无效（⑤ vs ⑥，唯一变量为显存预算）：

指标	⑤ 重算·0.65	⑥ 重算·0.90	增加 16.3 GB 显存的收益
实测显存占用	43,735 MiB	60,055 MiB	+16,320 MiB
GPU KV 容量	66,080 token	153,168 token	+132%
TTFT p50	14.137 s	14.122 s	**≈0（无收益）**
输出吞吐	19.4 tok/s	20.1 tok/s	**+3.6%（噪声级）**

原因很直接：待恢复的冷会话，其 KV 本就不在显存中——显存再大，KV 也只能重新算一遍，重算的瓶颈是 GPU 预填充算力而非 KV 容量。由此推出：**重算路线达标的唯一途径，是让全部会话的 KV 始终驻**

留显存、永不变冷。本测会话池为 90 会话 × 1.55 GB = **140 GB KV 工作集**（实测落盘 142 GB 取证），加 28 GB 模型权重，重算路线达标需约 **168 GB 显存，约合 3 张 N260（64 GB）加速卡**。

第三步：AISSD5000 路线需要多少显存？ 实测答案：单卡 40.5 GB（⑦，`util=0.60`）即达标——KV 外置于 AISSD5000，显存只需容纳权重与在批 KV：

指标	⑥ 重算·显存加满（0.90）	⑦ AISSD5000·低显存（0.60）	⑦ 相对 ⑥
实测显存占用	60,055 MiB（58.6 GB）	**40,527 MiB（39.6 GB）**	少 33%
GPU KV 容量	153,168 token	48,672 token	少 68%
TTFT p50	14.122 s（不达标）	**6.264 s（达标）**	**快 2.25 倍**
TTFT p90	22.299 s	13.294 s	快 1.68 倍
输出吞吐	20.1 tok/s	**32.4 tok/s**	**高 61%**
整批完成时间	25.1 s	15.9 s	快 1.58 倍

6.3.2 显存等效替代量核算

技术路线	达到 SLO（冷恢复 TTFT ≤ 8 s）的条件	所需显存
重算·单卡（任意显存预算）	无法达标（0.65 与 0.90 实测均 14.1 s）	—
重算·KV 全量驻留	140 GB KV 工作集 + 28 GB 权重全部驻留显存	**约 168 GB（约 3 张 N260）**
AISSD5000 + 单卡	KV 外置阵列，用时回读（实测 6.264 s）	**40.5 GB（单卡的 63%）**
等效替代量	168 GB − 40 GB	**约 128 GB（约合 2 张 N260）**

> **两点重要性质：** 其一，替代量的主体是 KV 工作集（140 GB），因此**会话池越大、上下文越长，AISSD5000 等效替代的显存越多**——1,000 个同规格会话即对应约 1.5 TB KV（相当于约 24 张 N260 的显存），而 AISSD5000 侧只是多占一些盘空间；显存必须逐卡购买，阵列容量则按盘扩展，这就是"以存力换显存"的经济学本质。其二，本口径是**保守下界**：它假设重算路线可以通过驻留达到"快"，而实际生产中会话流失、服务重启、跨节点迁移都会打破驻留，届时重算路线即使配足 168 GB 也会周期性跌回 14 s，AISSD5000 路线则不受影响（KV 持久在阵列上）。

> 注脚：14B 模型在 0.50 显存预算下服务无法启动（权重约 28 GB 占用后，剩余空间不足以容纳 32K 上下文的最低 KV 需求）——大模型显存本就紧张，KV 外置是比"堆显存"更有效的资源配置方式。

6.4 核心对比二：AISSD5000 vs 本地 NVMe（同访问路径）

GDS·8线程（各自最优）	① AISSD5000	③ 本地NVMe	差异
TTFT p50	5.653 s	5.581 s	**打平（差约 1%）**

输出吞吐	45.5 tok/s	45.5 tok/s	完全一致
物理读峰值	2.68 GB/s	2.51 GB/s	AISSD5000 高 7%
每请求回读带宽	2.4–2.6 GB/s	2.3–2.5 GB/s	相当

GDS·4线程（默认配置）	② AISSD5000	④ 本地NVMe	差异
TTFT p50	6.797 s	7.685 s	AISSD5000 快 12%
输出吞吐	40.2 tok/s	37.6 tok/s	AISSD5000 高 7%
物理读峰值	1.98 GB/s	1.75 GB/s	AISSD5000 高 13%

> 解读：AISSD5000 经 100 GbE / RoCEv2 网络访问，端到端推理性能与本机 PCIe 直连的 NVMe 盘**等速**（最优配置下差异在 1% 噪声内），默认配置下凭 4 盘 RAID0 聚合带宽**反超**本地单盘。即"远程"不构成任何性能损耗；在此前提下，AISSD5000 额外提供本地盘在架构上无法具备的能力：容量独立扩展（不占用服务器盘位）、多节点共享同一 KV 池、存算分离部署与独立运维。

6.5 核心对比三：AISSD5000 vs 无外存重算（同显存预算 0.65）

指标	⑤ 重算	① AISSD5000+GDS·8线程	AISSD5000 收益
TTFT p50	14.137 s	5.653 s	降低 60%（快 2.50 倍）
TTFT p90	23.636 s	8.869 s	降低 62%（快 2.67 倍）
TTFT 均值	13.114 s	5.491 s	降低 58%
输出吞吐	19.4 tok/s	45.5 tok/s	提升 135%（2.35 倍）
整批完成时间	25.9 s	11.0 s	缩短 57%

> 解读：每个冷会话恢复时，重算方案须对约 8,448 token 的历史前缀重新执行 14B 模型的预填充计算；AISSD5000 方案改为从阵列直读 1.55 GB 已存 KV（0.6 s 级），将节省下来的 GPU 预填充算力用于服务在批请求的解码。首字延迟与吞吐的双重收益由此而来，且会话越长、模型越大，重算越昂贵、该收益越显著。

7. 分析与讨论

7.1 显存等效替代的机理

KV Cache 是显存中除权重外的最大占用项。14B 模型权重约 28 GB，在 64 GB 卡上留给 KV 的空间本就有限；随会话数与上下文长度增长，显存内 KV 必然溢出。此时只有两条路：溢出即丢弃、用时重算（⑤⑥），或溢出至外部存储、用时回读（①②⑦）。测试证明：重算路径的恢复成本是十几秒级预填充，且无法用更多显存赎回（§6.3 第二步）——想让重算路线"快"，只能把整个 KV 工作集用显存硬扛下来，显存需求随会话规模线性上升（本测 140 GB 工作集即需约 3 张卡）；而 AISSD5000 回读路径把恢复成本压缩到 6 s 级，显存只需容纳权重与在批 KV（单卡 40.5 GB 即可），工作集增长的部分全部由阵列消化。**显存与会话规模解耦，是外置 KV 存储区别于"堆显存"的本质**；省出的显存预算可用于加大并发批次、加载更大模型或减少加速卡采购量。

7.2 GDS 直读与访问参数

GDS（盘→显存直读）绕过页缓存与 CPU 拷贝，是单请求恢复速度最快的访问路径。测试同时表明 GDS 的 IO 并行线程数存在平台相关的最优值：本平台（海光 8 NUMA）实测 8 线程最优，较默认 4 线程 TTFT 再降 17%、物理读带宽提升 35%；继续增大线程数因内存子系统争抢反而劣化。生产部署建议将 `gds_io_threads` 显式配置为 8（本平台），并随平台实测校准。

7.3 单卡条件下存储能力远未饱和

本机 AISSD5000 阵列脱离推理进程的裸读能力实测 6.5–7.8 GB/s（O_DIRECT 多线程直读同一批 KV 文件，数据校验一致）；而单卡推理内的峰值消费仅 2.68 GB/s——**推理引擎单管道逐请求取数，只消费了阵列供给能力的约三分之一**。这说明单卡测得的各项收益是 AISSD5000 价值的保守下界，瓶颈在消费侧（单引擎串行）而非供给侧（存储）。

7.4 多卡条件下 AISSD5000 优势进一步扩大（结论 4 的分析）

多卡部署（每卡一个推理实例）从四个机制上放大 AISSD5000 的优势：

1. **聚合带宽兑现**：每张卡是一条独立的取数管道，N 卡即 N 条管道并行读取。单卡吃不完的供给余量（§7.3）被多管道消费，聚合读取带宽随卡数上升，逼近阵列多盘聚合与 100 GbE 网络（约 12 GB/s）的上限。同型 AISSD5000 阵列在 8 卡服务器上的内部实测旁证：聚合物理读由单实例 5.2 GB/s 增至 8 实例 10.18 GB/s（持续 8.93 GB/s），成员盘仍有余力；而作为对照的本地单盘在 6.5 GB/s 即达物理极限（util 85%）、无法继续扩展。
2. **本地盘的结构性天花板**：本地 NVMe 的带宽与容量被单台服务器独占且固定——加卡不会增加本地盘的供给，多卡竞争同一块本地盘反而更快饱和；AISSD5000 作为独立存储设备可通过加盘、多链路（双 100 G 口）继续扩展供给，存与算各自独立扩容。
3. **KV 池全集群共享**：AISSD5000 上的 KV 是集中式池，多卡/多机可共享——A 卡计算并存入的前缀 KV，B 卡直接命中复用（"一次预填充、全集群受益"），系统级命中率随集群规模上升；本地盘的 KV 是节点私有的，其他节点无法访问，跨节点会话迁移必须重算。
4. **显存等效替代随集群规模放大**：§6.3 的替代量取决于 KV 工作集大小，而多卡/多机集群的会话池是单卡的数倍乃至数十倍——若用"堆显存驻留"路线承接，显存需求随会话池同步膨胀且须逐卡采购；AISSD5000 路线下所有卡共享同一份外置 KV 池（无需逐卡复制），工作集增长只消耗阵列盘空间。集群越大、会话越多，等效替代的显存总量越大。

> 严谨性说明：机制 1 的 8 卡数据来自同型 AISSD5000 阵列在另一平台（8×GPU 服务器）的内部实测，作为旁证列出；本机多卡物理验证列为后续工作（§9）。机制 2–4 为架构事实，不依赖额外实测。

8. 结论

1. **显存等效替代（核心结论）**：达到同等会话恢复能力（冷恢复 TTFT ≤ 8 s），重算路线必须将 140 GB KV 工作集全量驻留显存（连同权重共需约 168 GB，约合 3 张 N260）——因为实测证明加显存对冷会话重算零收益（0.65 与 0.90 两档均为 14.1 s），驻留是其达标的唯一途径；AISSD5000 路线以单卡 40.5 GB 显存实测达标（6.264 s，且吞吐比重算高 61%）。**等效替代约 128 GB 显存（约合 2 张 N260），且替代量等于 KV 工作集大小、随会话规模线性增长。**

2. **相比本地 NVMe**：最优配置下 AISSD5000 与本地盘端到端等速（差异 ≤1%），默认配置下反超 7%–13%；远程 NVMe-oF 访问零性能损耗，同时具备容量弹性、多机共享与存算分离的架构能力。
3. **相比无外存重算**：同显存预算下首字延迟快 2.50 倍（p90 快 2.67 倍）、输出吞吐高 2.35 倍、整批完成时间缩短 57%。
4. **多卡放大效应**：单卡仅消费阵列裸能力的约三分之一；多卡部署下聚合带宽随卡数兑现（同型平台旁证：8 实例聚合 10.18 GB/s、盘仍有余力，本地单盘 6.5 GB/s 到顶）、KV 池全集群共享、显存节省按卡数放大（8 卡约 156 GB）——AISSD5000 的优势随部署规模扩大而扩大。

9. 局限与后续工作

1. **单机单卡**：多卡聚合、跨节点 KV 共享的直接物理验证待多卡环境补充；本报告结论 4 为实测数据外推与同型平台旁证。
2. **GDS 并发上限受软件制约**：当前 LMCache 版本 GDS 显存暂存池用后不即时回收，单次测量工作集受池容量约束（14B 下约 9 会话/14 GiB 池），统一取 N=8 规避；更高并发的 GDS 表现待上游修复后验证。
3. **合成负载与统计重复**：均匀访问合成数据相对真实偏斜流量为保守估计；各数据点为单次测量，建议关键点重复 ≥5 次并给出置信区间。
4. **未覆盖能力**：异步预取、双 100 G 口聚合、跨重启 KV 持久化复用、更长上下文，列为后续测试项。

附录 A：复现命令（七组完整）

> 约定：[宿主] 在服务器宿主机执行（root）；`docker exec` 在 `vllm` 容器内执行。populate 前台阻塞（等打印 `wall=...s`）后再执行 `drop_caches` / `iostat` / measure。每组测试前先执行 A.1 重启释放显存。

A.0 前置环境（一次性）

```
# [宿主] AISSD5000 四盘组 RAID0 + 挂载 + 监控依赖
apt-get update && apt-get install -y sysstat
mdadm --create /dev/md0 --level=0 --raid-devices=4 /dev/nvme2n1 /dev/nvme3n1 /dev/nvme4n1 /dev/nvme5n1 --c
hunk=512
mkfs.xfs /dev/md0 && mkdir -p /mnt/ws5000 && mount /dev/md0 /mnt/ws5000

# [宿主] 创建容器（--device=/dev/mem、memlock=-1 供 GDS 注册显存）
IMG=cr.metax-tech.com/public-ai-release/maca/vllm-metax:0.20.0-maca.ai3.7.0.107-torch2.8-py310-ubuntu22.04
-amd64
docker run -itd --name vllm --network=host \
  --device=/dev/dri --device=/dev/mxcd --device=/dev/mem \
  --group-add video --security-opt seccomp=unconfined --security-opt apparmor=unconfined \
  --shm-size 100gb --ulimit memlock=-1 -e MACA_GRAPH_LAUNCH_MODE=1 \
  -v /data/models:/root/models -v /mnt/ws5000:/mnt/ws5000 \
  $IMG sleep infinity

# [容器] 下载 14B 模型（国内可经 ModelScope）
docker exec vllm bash -lc "HF_ENDPOINT=https://hf-mirror.com hf download Qwen/Qwen2.5-14B-Instruct --local
-dir /root/models/Qwen2.5-14B-Instruct"

# [容器] 安装 cufile GDS 适配层（附录 C，GDS 各组需要）
docker exec vllm bash -lc "mkdir -p /opt/conda/lib/python3.10/site-packages/cufile"
docker exec vllm bash -lc "python -c 'import cufile, cufile.bindings; print(\"cufile OK\")'"

# [容器] 写入测量脚本（附录 B 源码）到 /root/bc_off_amd.py
# [容器] 编译 O_DIRECT 并行读引擎（附录 D，仅第㉗组需要）
docker exec vllm bash -lc "g++ -O2 -shared -fPIC -o /root/libws5000stage.so /root/ws5000stage.cpp -lpthrea
d"
```

A.1 通用：每组前重启释放显存 + 就绪轮询

```
docker exec vllm bash -lc "pkill -9 -f 'vllm serve' 2>/dev/null" 2>/dev/null
docker restart vllm && sleep 14
wait_ready() { for i in $(seq 1 140); do docker exec vllm bash -lc "python -c \"import urllib.request,sys;
sys.exit(0 if 'qwen' in urllib.request.urlopen('http://localhost:8000/v1/models',timeout=3).read().decode
() else 1)\"" 2>/dev/null && { echo READY; return 0; }; sleep 5; done; echo TIMEOUT; return 1; }
# 起服就绪后记录显存与 GPU KV 容量：
docker exec vllm bash -lc "mx-smi | tr '\r' '\n' | grep -oE '[0-9]+/65536 MiB' | head -1"
grep -aoE 'GPU KV cache size: [0-9,]+ tokens' /data/models/vllm_<组名>.log | tail -1
```

A.2 ①② AISSD5000 + GDS（8 线程 / 4 线程仅改 gds_io_threads）

```
docker exec vllm bash -lc "rm -rf /mnt/ws5000/gds; mkdir -p /mnt/ws5000/gds"
docker exec -d vllm bash -lc "export PYTHONHASHSEED=0 MACA_GRAPH_LAUNCH_MODE=1 LMCACHE_LOG_LEVEL=INFO \
 LMCACHE_CHUNK_SIZE=256 LMCACHE_LOCAL_CPU=True LMCACHE_MAX_LOCAL_CPU_SIZE=4 \
 LMCACHE_USE_GDS=True LMCACHE_GDS_BACKEND=cufile LMCACHE_GDS_PATH=/mnt/ws5000/gds \
 LMCACHE_GDS_BUFFER_SIZE=14336 LMCACHE_EXTRA_CONFIG='{\"gds_io_threads\": 8}'; \
 vllm serve /root/models/Qwen2.5-14B-Instruct --served-model-name qwen \
 --dtype bfloat16 --max-model-len 32768 --gpu-memory-utilization 0.65 --no-enable-prefix-caching \
 --kv-transfer-config '{\"kv_connector\":\"LMCacheConnectorV1\",\"kv_role\":\"kv_both\"}' --port 8000 \
 > /root/models/vllm_T8.log 2>&1"
wait_ready
docker exec vllm bash -lc "grep -aiE 'Using GDS backend|GDS backend using fstype' /data/models/vllm_T8.log
| tail -2"   # 确认 GDS 启用(xfs)
docker exec vllm bash -lc "python /root/bc_off_amd.py POPT8 150 90 1 1 0 populate"      # 灌入 90 会话
sync; echo 3 > /proc/sys/vm/drop_caches
nohup bash -c "iostat -x 1 90 /dev/md0 > /tmp/io_T8.log 2>&1" &
docker exec vllm bash -lc "python /root/bc_off_amd.py T8 150 8 64 8 0"                    # N8 conc8 冷读 s
ess0-7
awk '$1=="md0"{if($3>m)m=$3}END{printf "md0 peak %.2f GB/s\n", m/1e6}' /tmp/io_T8.log
grep -acE 'Memory allocation failed' /data/models/vllm_T8.log                              # 期望 0
grep -acE 'need to load: [1-9]' /data/models/vllm_T8.log                                   # 期望 >0（全部经
GDS 加载）
```

A.3 ③④ 本地 NVMe + GDS（唯一差异：GDS 目录指向本地盘 ext4；8/4 线程同上）

```
docker exec vllm bash -lc "rm -rf /root/models/gds_local14; mkdir -p /root/models/gds_local14"
docker exec -d vllm bash -lc "export PYTHONHASHSEED=0 MACA_GRAPH_LAUNCH_MODE=1 LMCACHE_LOG_LEVEL=INFO \
 LMCACHE_CHUNK_SIZE=256 LMCACHE_LOCAL_CPU=True LMCACHE_MAX_LOCAL_CPU_SIZE=4 \
 LMCACHE_USE_GDS=True LMCACHE_GDS_BACKEND=cufile LMCACHE_GDS_PATH=/root/models/gds_local14 \
 LMCACHE_GDS_BUFFER_SIZE=14336 LMCACHE_EXTRA_CONFIG='{\"gds_io_threads\": 8}'; \
 vllm serve /root/models/Qwen2.5-14B-Instruct --served-model-name qwen \
 --dtype bfloat16 --max-model-len 32768 --gpu-memory-utilization 0.65 --no-enable-prefix-caching \
 --kv-transfer-config '{\"kv_connector\":\"LMCacheConnectorV1\",\"kv_role\":\"kv_both\"}' --port 8000 \
 > /root/models/vllm_LOC8.log 2>&1"
wait_ready
docker exec vllm bash -lc "python /root/bc_off_amd.py POPLOC8 150 90 1 1 0 populate"
sync; echo 3 > /proc/sys/vm/drop_caches
nohup bash -c "iostat -x 1 90 /dev/nvme0n1 > /tmp/io_LOC8.log 2>&1" &
docker exec vllm bash -lc "python /root/bc_off_amd.py LOC8 150 8 64 8 0"
awk '$1=="nvme0n1"{if($3>m)m=$3}END{printf "nvme peak %.2f GB/s\n", m/1e6}' /tmp/io_LOC8.log
```

A.4 ⑤⑥ 重算（无外存；两档仅改 --gpu-memory-utilization 0.65 / 0.90）

```
docker exec -d vllm bash -lc "export PYTHONHASHSEED=0 MACA_GRAPH_LAUNCH_MODE=1 LMCACHE_LOG_LEVEL=INFO \
 LMCACHE_CHUNK_SIZE=256 LMCACHE_LOCAL_CPU=True LMCACHE_MAX_LOCAL_CPU_SIZE=4; \
 vllm serve /root/models/Qwen2.5-14B-Instruct --served-model-name qwen \
 --dtype bfloat16 --max-model-len 32768 --gpu-memory-utilization 0.65 --no-enable-prefix-caching \
 --kv-transfer-config '{\"kv_connector\":\"LMCacheConnectorV1\",\"kv_role\":\"kv_both\"}' --port 8000 \
 > /root/models/vllm_REC065.log 2>&1"
wait_ready
sync; echo 3 > /proc/sys/vm/drop_caches
docker exec vllm bash -lc "python /root/bc_off_amd.py REC065 150 8 64 8 0"
grep -aE 'need to load:' /data/models/vllm_REC065.log | grep -acE 'need to load: [1-9]'   # 期望 0（纯重
算）
```

A.5 ⑦ AISSD5000 · 低显存直读档（util 0.60，O_DIRECT 并行读引擎）

```
# 前置：附录 D 的读引擎补丁已应用到 LMCache local_disk_backend.py，libws5000stage.so 已编译
docker exec vllm bash -lc "rm -rf /mnt/ws5000/lmcache; mkdir -p /mnt/ws5000/lmcache"
docker exec -d vllm bash -lc "export PYTHONHASHSEED=0 MACA_GRAPH_LAUNCH_MODE=1 LMCACHE_LOG_LEVEL=INFO \
 LMCACHE_CHUNK_SIZE=256 LMCACHE_LOCAL_CPU=True LMCACHE_MAX_LOCAL_CPU_SIZE=32 \
 LMCACHE_LOCAL_DISK=file:///mnt/ws5000/lmcache LMCACHE_MAX_LOCAL_DISK_SIZE=500 \
 LMCACHE_KSTAGE=1 LMCACHE_KSTAGE_MODE=cengine LMCACHE_KSTAGE_THREADS=64 \
 LMCACHE_KSTAGE_SO=/root/libws5000stage.so; \
 vllm serve /root/models/Qwen2.5-14B-Instruct --served-model-name qwen \
 --dtype bfloat16 --max-model-len 32768 --gpu-memory-utilization 0.6 --no-enable-prefix-caching \
 --kv-transfer-config '{\"kv_connector\":\"LMCacheConnectorV1\",\"kv_role\":\"kv_both\"}' --port 8000 \
 > /root/models/vllm_MEM60.log 2>&1"
wait_ready
docker exec vllm bash -lc "python /root/bc_off_amd.py POPMEM60 150 90 1 1 0 populate"
sync; echo 3 > /proc/sys/vm/drop_caches
nohup bash -c "iostat -x 1 90 /dev/md0 > /tmp/io_MEM60.log 2>&1" &
docker exec vllm bash -lc "python /root/bc_off_amd.py MEM60 150 8 64 8 0"
```

附录 B：测量脚本 `/root/bc_off_amd.py`

用法：`python bc_off_amd.py <label> <reps> <N> <decode> <conc> <offset> [populate]`；`reps=150` → 每会话约 8,448 token；末位加 `populate` 为灌入模式（decode=1 逐会话预填充）。

```
# 用法: python bc_off_amd.py <label> <reps> <N> <decode> <conc> <offset> [populate]
import urllib.request, json, time, sys
from concurrent.futures import ThreadPoolExecutor
BASE='http://localhost:8000/v1/chat/completions'; MODEL='qwen'
label=sys.argv[1]; reps=int(sys.argv[2]); N=int(sys.argv[3]); decode=int(sys.argv[4]); conc=int(sys.argv
[5]); off=int(sys.argv[6])
basep='背景知识：AISSD5000是国产高性能全闪NVMe-oF存储，可作为大模型推理KV缓存的分层后备介质，配合vLLM与LMCache在
显存/内存/磁盘之间分层存取KV。'
def make_prefix(i): return '[sess-%05d] '%i + basep*reps
def req(sid, maxtok):
    body=json.dumps({'model':MODEL,'stream':True,'messages':[{'role':'system','content':make_prefix(sid)},
{'role':'user','content':'回答%d'%sid}],'max_tokens':maxtok,'temperature':0}).encode()
    st=time.time(); ttft=None; n=0
    try:
        r=urllib.request.urlopen(urllib.request.Request(BASE,data=body,headers={'Content-Type':'applicatio
n/json'}),timeout=1200)
        for line in r:
            sx=line.decode('utf-8','ignore')
            if sx.startswith('data:') and '"content"' in sx:
                if ttft is None: ttft=time.time()-st
                n+=1
        return (ttft, time.time()-st, n)
    except Exception:
        return (None,None,0)
def pct(a,q): return a[min(len(a)-1,int(len(a)*q))] if a else 0.0
if len(sys.argv)>7 and sys.argv[7]=='populate':
    t0=time.time()
    for i in range(N): req(off+i,1)
    print('[%s] populate N=%d off=%d wall=%.1fs'%(label,N,off,time.time()-t0),flush=True); sys.exit(0)
res=[]; t0=time.time()
with ThreadPoolExecutor(max_workers=conc) as ex:
    futs=[ex.submit(req,off+i,decode) for i in range(N)]
    for f in futs:
        x=f.result()
        if x[0] is not None: res.append(x)
wall=time.time()-t0
tt=sorted(r[0] for r in res); tot=sum(r[2] for r in res)
print('[%s] n=%d wall=%.1fs TTFT p50=%.3f p90=%.3f p99=%.3f mean=%.3f out_tok/s=%.1f'%(label,len(res),wal
l,pct(tt,.5),pct(tt,.9),pct(tt,.99),(sum(tt)/len(tt) if tt else 0),tot/wall),flush=True)
```

附录 C：cufile GDS 适配层（GDS 各组使用）

`/opt/conda/lib/python3.10/site-packages/cufile/__init__.py`（要点）

```
import ctypes, os
for _pre in ("libmxc-runtime64.so.1", "libmcruntime.so"):
    try: ctypes.CDLL(_pre, mode=ctypes.RTLD_GLOBAL)
    except OSError: pass
_lib = ctypes.CDLL("libmcfile.so", mode=ctypes.RTLD_GLOBAL)
_ODIRECT = getattr(os, "O_DIRECT", 0o40000)   # GDS 句柄注册要求 O_DIRECT
# 绑定 mcFileDriverOpen/Close、mcFileHandleRegister/Deregister、mcFileRead/Write，
# 封装为 LMCache 所需的 CuFile / CuFileDriver（read/write 直读写 GPU 显存指针）；
# bindings.py 另绑定 mcFileBufRegister/Deregister → cuFileBufRegister/Deregister。
# 完整源码见工程留档《AISSD5000-KVCache-六档对照实验报告.md》§8.2。
```

附录 D：O_DIRECT 并行读引擎（第⑦组使用）

第⑦组（低显存档）的读取路径为自研 C 库 `libws5000stage.so`：以 O_DIRECT 多线程并行读取 KV 分块文件、写入 LMCache 的 CPU 缓冲（4K 对齐直读，不对齐时经线程本地对齐暂存回退），不占用任何 GPU 显存池。编译：`g++ -O2 -shared -fPIC -o libws5000stage.so ws5000stage.cpp -lpthread`。

`ws5000stage.cpp` 核心源码（读取部分）

```
#define _GNU_SOURCE
#include <pthread.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#define ALIGN 4096UL
#define SCRATCH_MAX (128UL*1024*1024)
extern "C" {
typedef struct { const char** paths; void** dsts; size_t* sizes; long* results;
    int n; int next; pthread_mutex_t lock; int direct; } job_t;
static __thread char* g_scratch=NULL;
static void ensure_scratch(){ if(!g_scratch){ if(posix_memalign((void**)&g_scratch,ALIGN,SCRATCH_MAX)!=0)
g_scratch=NULL; } }
static long do_one_read(const char* path, void* dst, size_t sz, int direct){
    size_t rd=((sz+ALIGN-1)/ALIGN)*ALIGN; long got=-1;
    int aligned=(((uintptr_t)dst%ALIGN)==0);
    if(direct && aligned && rd<=SCRATCH_MAX){                 // 4K 对齐: O_DIRECT 直读入目标缓冲
        int fd=open(path,O_RDONLY|O_DIRECT);
        if(fd>=0){ ssize_t n=pread(fd,dst,rd,0); close(fd); if(n>=(ssize_t)sz) return (long)sz; }
    }
    ensure_scratch();
    if(g_scratch && rd<=SCRATCH_MAX){                          // 不对齐: 对齐暂存 + memcpy 回退
        int fd=open(path,O_RDONLY|O_DIRECT);
        if(fd>=0){ ssize_t n=pread(fd,g_scratch,rd,0); close(fd);
            if(n>=(ssize_t)sz){ memcpy(dst,g_scratch,sz); return (long)sz; } }
    }
    int fd2=open(path,O_RDONLY);                               // 最终回退: 缓冲读
    if(fd2>=0){ size_t off=0; char* d=(char*)dst;
        while(off<sz){ ssize_t n=pread(fd2,d+off,sz-off,off); if(n<=0) break; off+=n; }
        close(fd2); if(off==sz) got=(long)sz; }
    return got;
}
static void* cworker(void* arg){
    job_t* j=(job_t*)arg;
    for(;;){
        pthread_mutex_lock(&j->lock); int i=j->next++; pthread_mutex_unlock(&j->lock);
        if(i>=j->n) break;
        j->results[i]=do_one_read(j->paths[i],j->dsts[i],j->sizes[i],j->direct);
    }
    return NULL;
}
int ws5000_batch_read2(const char** paths, void** dsts, size_t* sizes, long* results,
                       int n, int nthreads, int direct, long* pu, long* mu){
    if(nthreads<1)nthreads=1; if(nthreads>n)nthreads=n; if(nthreads>256)nthreads=256;
    job_t j; j.paths=paths;j.dsts=dsts;j.sizes=sizes;j.results=results;j.n=n;j.next=0;j.direct=direct;
    pthread_mutex_init(&j.lock,NULL);
    pthread_t th[256];
    for(int t=0;t<nthreads;t++) pthread_create(&th[t],NULL,cworker,&j);
    for(int t=0;t<nthreads;t++) pthread_join(th[t],NULL);
    pthread_mutex_destroy(&j.lock);
    int ok=0; for(int i=0;i<n;i++) if(results[i]>=0) ok++;
    return ok;
}
} // extern "C"
```

LMCache 接入要点：在 `lmcache/v1/storage_backend/local_disk_backend.py` 中覆写 `LocalDiskBackend.batched_get_blocking`——先按 LMCache 原生流程串行分配各 chunk 的 CPU 缓冲（`local_cpu_backend.allocate`），再将全部（路径、目标指针、大小）一次性交给 `ws5000_batch_read2`

并行读取，返回 MemoryObj 列表；以环境变量 `LMCACHE_KSTAGE=1`、`LMCACHE_KSTAGE_SO=<so路径>`、`LMCACHE_KSTAGE_THREADS=64` 启用。KV 数据经该路径回读后由 LMCache 原生 `batched_to_gpu` 装载显存，正确性经与 GDS/原生路径逐 token 对齐校验。

附录 E：原始数据文件

文件	内容
`/root/matrix.out`、`/root/mem60.out`、`/root/run14bG1.out`、`/root/run14bG2.out`	七组测试全过程日志（TTFT/吞吐/取证输出）
`/data/models/vllm_mx{T8,T16,LOC,REC065,REC09,MEM60}.log`、`vllm_14bG1.log`	各组服务日志（need-to-load / GDS / 显存与 KV 容量原文）
`/tmp/io_mx_*.log`、`/tmp/io14_G1.log`	iostat 物理读带宽时间序列
`/root/bc_off_amd.py`、`/root/ws5000stage.cpp`、`/root/libws5000stage.so`	测量脚本与读引擎
`/opt/conda/lib/python3.10/site-packages/cufile/`	GDS 适配层源码

本报告七组数据均来自实测，测试方法、命令与脚本完整留档，可独立复现。