arXiv:2611.30027v1 [cs.LG] 23 Nov 2026

Hopper GPU 上的快速 4 位 KV 缓存解码：为何 vLLM 的 4 位路径速度慢，以及如何修复

Jian Zhang,
Northeastern University

2026年11月

摘要

4 位 KV 缓存读取的字节数约为 FP8 的一半，但 Hopper GPU 没有 FP4硬件，而且相对于内存带宽，每字节可用的指令发射能力很低，因此必须通过软件解码 4 位值。在 H200 上，vLLM 的 NVFP4 KV 路径（FlashInfer 的解码）只能达到内存带宽的 10–20%，速度最多比 FP8 慢 1.7× 倍。NVBit 指令计数表明，原因有二：vLLM 4 位路径背后的内核（FlashInfer NVFP4、Triton UltraQuant）每字节执行的指令数是 Hopper 发射预算的 2–6× 倍，而且加载、解码和矩阵乘法是串行执行的。我们的 Hopper 原生解码内核将 TMA 生产者线程束与两个计算线程束组重叠，在写入时就确定缓存布局，使解码值能从寄存器直接馈入张量核心，并交换矩阵操作数，从而让较小的查询组无需填充。在 H200 上，批次为 32–128 时，该内核可达到峰值带宽的 67–75%；在全部 11 种形状上，其速度比 XQA 和 FlashAttention-4 提出的原生 FP8 中较快者快 1.12–1.49× 倍。在 vLLM 内部，采用精度方案（K 均值去除与旋转、4 个 BF16 汇聚令牌、BF16 第 0 层）后，批次为 8 及以上时，它在 Llama-3.1-8B 上的每步解码速度比 vLLM 的 FlashInfer FP8 路径快 1.12–1.64× 倍（比 FlashAttention-4 的 FP8 快 1.04–1.24×倍），批次为 1 时性能持平，且可容纳 1.63× 倍数量的令牌。在一块 H200 上运行类推理请求流时，Llama-3.1-8B 达到 99% 请求满足延迟目标时的最高速率为 FP8 的 1.23–1.32× 倍，Qwen2.5-72B 则为 1.67–2.20× 倍，此时缓存成为瓶颈；预填充密集型请求流的表现更差（预填充速度慢 7%）。在五个 7B–32B 模型上，重复运行结果显示，与 BF16 KV 相比，准确率没有显著差异。在 Qwen2 系列模型中，其类推理变体在 FP8 KV 下性能崩溃；第 0 层的键均值几乎完全由先前研究指出的键投影偏置构成，可通过 RoPE 观察到，且仅凭权重即可计算。将该层保留为 BF16 可恢复 FP8 KV；再结合 K 均值去除，还可恢复 4 位 KV。

1 引言

在长上下文和批量服务场景中，每个解码步骤都会读取整个 KV 缓存，因此解码注意力受内存带宽限制。对于 Llama-3.1-8B（32 个查询头、8 个 KV 头、头维度为 128），BF16 格式下每行 KV 数据为 256 字节，可支持约 1,024 次浮点运算，即每字节约 4 次；而 H200 每字节可执行约 206 次密集 BF16 张量核心运算

（989 TFLOPS、 ÷ 4.8 TB/s，规格表数值）。因此，每步耗时约等于读取的字节数除以带宽；读取的字节越少，解码越快，每个 GPU 可处理的请求也越多。

英伟达的 Blackwell GPU 可在硬件中解码 NVFP4 等 4 位格式。目前已部署的 GPU 仍主要是 Hopper（H100、H200、H20），它们支持 BF16 和 FP8，但不支持 FP4。在这些 GPU 上，4 位值必须在注意力内核中由软件解包并进行缩放。我们测量了这在当前带来的开销：在 H200 上，批次大小为 32、上下文长度为 16K 时，FlashInfer [59] 使用 BF16 需要 0.495 毫秒，使用 FP8 需要 0.395 毫秒（使用其 XQA 内核时为 0.255 毫秒，该数据在单独的作业中测得），而使用

NVFP4。启用 NVFP4 键值缓存的 vLLM [25] 用户在 Hopper 上可以节省内存，但解码速度会变慢。

QServe [28]曾指出，软件解码可能会使低比特注意力受计算能力限制，并在 A100 上针对 INT4 进行了论证。Hopper 让问题更加棘手：其内存带宽的增长快于指令发射速率，因此在达到满带宽时，内核每字节可执行的指令数更少（第 3 节）。

主要贡献：

1. **通过测量进行诊断**。我们使用 NVBit [52]指令计数，并将其与 Hopper 的发射预算进行对照（H200 达到满带宽时约为每字节 7 条线程指令），发现现有 4 位解码内核因两个不同原因而停滞：指令过多和等待串行化；不同内核的停滞原因也各不相同。

2. **面向 Hopper 的原生 4 位解码内核**，支持 NVFP4、MXFP4 和 INT4，并为 FP4 格式提供分页式 KV 支持。在 H200 上，从内核层面看，NVFP4 的速度是 FlashInfer 的 XQA FP8 内核的 1.2–2.1× 倍，是其 FA2 FP8 内核的 1.4–2.3× 倍；若将 FlashAttention-4 原生 FP8 也纳入比较，则相较于各形状下最快的 FP8 内核，速度为 1.12–1.49× 倍（表 5）。流水线结构借鉴了 FlashAttention-3 [44] 和 LiquidGEMM [17]。其中各个部分本身都并非新创：在 M 较小时交换 A、B 操作数是 GEMM内核中的常见做法（例如 vLLM 的 SM90 FP8 GEMM [33]）；BitDecoding-FP4 [11] 在 Blackwell 原生 FP4张量核心上进行 FP4 KV 解码时，会交换两次注意力矩阵乘法的操作数；BitDecoding [10] 将低比特KV 打包成张量核心感知布局；Marlin [12]会离线重排 4 位权重，使每个线程的片段连续；TensorBridge [65]将操作数交换与离线置换的NVFP4权重结合起来，从寄存器向 Hopper 的 FP8 wgmma 供数；FuTransformer [53] 在 H100 上以 mma.sync片段顺序存储 K 的 INT4 副本；FServe [4] 则在 H100 上将 INT4 KV 缓存解码为寄存器中的 FP8，用于基于 FlashAttention-3 的内核。其测量显示，解码速度可与 FlashAttention-3 的 FP8 内核相当，甚至更快（未与 XQA 比较）。我们的新增工作，是将这些方法用于 Hopper 上 E2M1（NVFP4、MXFP4）以及 INT4 KV 缓存的解码注意力；Hopper 没有 FP4硬件，而我们的速度超过了所测量的 FP8 解码内核：缓存写入内核在线排布缓存，使每个线程的片段连续；解码后的 K 和 $V^\top$ 均从寄存器向 wgmma 供数；操作数交换则避免小型查询组出现填充行。第 3 节的指令计数说明了这些选择为何能在 Hopper 的发射预算内发挥作用。

3. **vLLM 中的端到端结果**通过插件实现，并与使用 XQA 解码内核的 FlashInfer FP8（我们在 vLLM 中测得的两个 FP8 后端中速度更快的一个；当其构件检查通过时，vLLM 的 FlashInfer 后端会在 Hopper 上选择该内核）以及 vLLM 自有的 NVFP4 键值路径进行对比，同时还包括单个 GPU 在负载下可持续达到的最高请求速率：在 Llama-3.1-8B 上为 FP8 的 1.23–1.32× 倍；随着键值缓存成为瓶颈，在 Qwen2.5-72B 上提升至 1.67–2.20× 倍（表 9）。

4. **一种精度方案，以及已测得的已知失效机制：**对于某些模型，小组 FP4 需要进行 K 均值去除。Liu 等人 [30] 将 R1-Distill-Qwen 模型的 K 离群值追溯至从其 Qwen 基础模型继承而来的键投影偏置，并研究了 3 位和 4 位 KV；而 vLLM拉取请求 #63081 [48] 警告称，当这种偏置遇到量化 KV 缓存时会产生问题。我们量化表明，第 0 层的 K 均值就是该偏置经 RoPE 作用后的表现（其能量的 99.7–100% 仅根据权重计算得出）；我们在 vLLM 0.33 中复现了报告所述的 Qwen2 系列推理模型在 FP8 KV [26] 下的崩溃，展示了仅将第 0 层保留为 BF16 即可修复 FP8 KV，并结合 K 均值去除和 4 位 KV；此外，我们加入一个对照：相同的 R1

在没有键偏置的 Llama 基础模型上进行的蒸馏不会发生崩溃。

5. **向 vLLM 报告的发现**（附录 A）：其 UltraQuant 格式中的一个缩放常数、第 0 层 FP8 故障、一个前缀缓存正确性错误，以及哪些拟议的评估检查能够捕获此类错误。

面向 Hopper 上 vLLM 用户的实用说明。 根据我们的测量（vLLM 0.33，除非另有说明，均为 Llama- 3.1-8B、H200）：

- vLLM 的 NVFP4 键值缓存（`--kv-cache-dtype nvfp4`）可节省内存，但在批次大小为 8 及以上时，解码速度比 FP8 慢 1.4–2.3×；批次大小为 1 时则慢约 1.1×（第 7 节）。其 UltraQuant 格式在英伟达平台上会回退到 Triton 解码内核，每步速度比 FP8 慢 1.8–4.4×。

- 使用 FlashInfer 和 FP8 KV 时，只有在连接英伟达制品服务器的网络检查成功后，vLLM 才会使用速度更快的 XQA 解码内核；如果检查失败（在我们的集群中，代理服务器会导致这种情况），它就会回退到 FlashInfer 自带的解码内核，且不会报错。请检查日志中的`decode_backend=xqa`（第 2 节）。

- Qwen2 系列推理模型在使用 FP8 键值缓存时可能会无声崩溃。我们的测试中，将第 0 层保留为 BF16（`--kv-cache-dtype-skip-layers 0`，对于 28 层模型，键值缓存内存仅增加 1/28）即可恢复正常（第 8 节）；但对某些模型跳过层会导致 FlashAttention-3 崩溃（#65821 [42]；已在 #65891 [43] 中提出修复方案）。
- 当解码占主导且键值缓存内存成为瓶颈时，我们的插件效果最佳（批次大小为 8 及以上、输出较长、单个 GPU 上运行大型模型；表 9）。批次大小为 1 时，性能与 FP8 持平；预填充密集型流的速度则较慢（第 7 节）。该插件支持 128 的头维度，且每个键值头最多支持 8 个查询头（第 9 节）。

2 背景与相关工作

符号说明。 B 表示解码批次中的序列数，S 表示以令牌计的上下文长度。一个行 是一个键值头对应的一个令牌的键（K）或值（V）：共 128 个值。G 表示每个键值头对应的查询头数量（Llama-3.1-8B 为 4）。解码内核会将每个序列分成若干块并行处理，再通过分片合并合并部分结果，如 Flash-Decoding [9]所示。汇点词元 是每个序列最前面的四个令牌，我们将它们以 BF16 格式保存在 4 位缓存之外 [29, 56]。该配方 采用 NVFP4、K 均值去除、对 K 进行 Hadamard 旋转、四个 BF16 汇聚项，以及将第 0 层保留为 BF16。

格式。 表 1 按行列出了各类格式（MXFP4 和 MX 块缩放因子：MX 格式 [41]；NVFP4：[37]）。

快速低比特解码内核。 QServe [28] 表明，在 A100 上，朴素的 INT4 KV 注意力内核可能比 INT8 内核更慢，因为 CUDA 核心上的反量化在每字节操作数方面超出了屋顶线；通过采用更低成本的位操作、使用 FP16 而非 FP32 运算以及预取缩放因子，其速度最高可达到 INT8 内核的 1.5× 倍；但该工作没有报告 Hopper 上的结果。QServe、Marlin [12] 以及我们的方法所使用的将整数转换为 FP16 的魔数转换方法由 Kim 等人 [24]进行了描述。在 H100 上，经过大量调优后，PyTorch 的 INT4 分组查询注意力解码内核可达到带宽的 17–30%，而 BF16 可达到 44–64%（根据报告的 GB/s 计算得出）[40]。FOSCAR [63]为 INT4 键值解码引入了仿射折叠，并报告称其在 H100 上接近峰值带宽；其指令预算论证（公式 2）正是我们在第3节中采用的模型。

表 1：每 128 个值的头部行格式。

格式	元素	缩放	每行字节数
BF16	16 位浮点数	none	256
FP8（E4M3）	8 位浮点数	每个张量	128
INT4（每个令牌，非对称）	4 位整数	每行使用 FP16 缩放和零点	68
NVFP4	E2M1	每 16 个值使用 E4M3，另加一个张量级 FP32 缩放	72
MXFP4 / UltraQuant	E2M1	每 32 个值使用 E8M0（2 的幂）	68

BitDecoding [10] 利用 TMA、warp specialization 和 wgmma 在 Hopper 上解码 INT4 KV；其打包方式保持了张量核心感知布局，并且在 Hopper 上将反量化值写入共享内存，以供 wgmma 使用。BitDecoding-FP4 [11] 面向 Blackwell 的原生 FP4张量核心（其测试使用了 B300 和 RTX PRO 5000）；它交换了两次注意力矩阵乘法，以计算转置后的得分瓦片和概率瓦片。NVFP4-MLA [35] 在 Rubin GPU 上以 NVFP4 存储并计算多头潜在注意力。LiquidGEMM [17] 使用一个加载warpgroup和两个计算warpgroup，在寄存器中进行反量化并直接为 wgmma 提供输入，用于 Hopper 上的 W4A8 GEMM；我们的内核将相同结构应用于注意力，其中两次矩阵乘法之间插入了 softmax。FlashInfer 提供了 vLLM 在 Hopper 上使用的 BF16、FP8 和 NVFP4 解码路径。FuTransformer [53] 在 H100 上融合了一个 Llama 解码块，并使用 BF16 KV 缓存的 INT4 副本：K 按 mma.sync片段顺序存储，并通过共享内存查找表填入寄存器；逐令牌组缩放因子在 QK点积之后应用；V 则通过共享内存进行转置。该方法与 PyTorch 基线进行比较，而不是与生产环境中的 FP8 KV注意力内核比较。FServe [4]（沿用 FireQ [3]的思路）使用逐令牌组缩放因子存储 INT4 KV 缓存，将 q 量化为 FP8，并在寄存器中将 K 和 V 反量化为 FP8，从而使其基于 FlashAttention-3 的预填充和解码内核能够运行于 FP8 wgmma 之上；该方法已集成到 vLLM 中，并在 H100 上进行了测量。其解码内核达到了与 FlashAttention-3 配合 FP8 缓存时相当甚至更高的有效（等效 BF16）带宽（见其图 7）；在包含 132 个可变长度请求的测试中，两者均耗时 1.52 ms（见其表4）；但它没有与 XQA 进行比较。我们的方法与其不同之处在于格式（除 INT4 外，还使用带块缩放因子的 E2M1）、将解码值作为寄存器操作数馈入 FP16 wgmma（并交换操作数），以及写入时布局。TensorBridge [65] 通过计算 $Y^\top = WX^\top$，在 Hopper 的 FP8 wgmma 上运行 NVFP4 权重，使权重成为寄存器操作数，并对权重进行预置换以支持 16 字节片段加载；这是权重GEMM，而非注意力。在 AMD MI355X 上，UltraQuant 论文 [7] 发现，4 位 KV“带来的是容量，而不是内核速度”：其解码注意力在软件反量化下耗时为 FP8 的 2.45× 倍，在原生 FP4 下耗时为 1.18× 倍（上下文长度 32K，批次大小 8）。一个开放的 vLLM拉取请求（#53060 [58]）为 Hopper 添加了 CUDA 4 位（TurboQuant）解码路径，并将其与 BF16 FlashAttention 及 vLLM 现有的 TurboQuant 路径进行比较，而不是与 FP8 KV 缓存比较。LMDeploy 的 TurboMind [18] 在 H200 上使用的 INT4 KV 缓存比其自身的 INT8 和 BF16 缓存更快（第9节）。我们的内核结合了 warp specialization（如 FlashAttention-3 [44]）、写入时布局（如 Marlin）和折叠（如 FOSCAR）。

低比特 KV 缓存的准确率。 KIVI [31] 和 KVQuant [15] 由于存在通道离群值而按通道量化键；QuaRot [2] 和 SAW-INT4 [22] 使用哈达玛变换旋转 K，SAW-INT4 表明逐 token INT4 需要这一操作。SageAttention 的 [61] “平滑 K”会减去 K 的逐通道均值。对于 INT4 键，FServe [4] 会在 RoPE 之后减去按离线校准的逐通道均值的一定比例（“收缩中心化”），并将逐通道范围与折叠到 q 中的缩放对齐；FireQ [3] 则结合了 RoPE 前后缩放。FlagRot [45] 报告称，整个头

哈达玛旋转会使 Qwen3-8B 上 NVFP4 缓存的 KL 散度升高 215%，而针对每个子向量选择是否旋转可以避免这一问题；BitDecoding-FP4 在 Blackwell 上的默认配方包括静态键平滑（校准后的逐头均值）、哈达玛旋转，以及前 16 个令牌使用 BF16，这与我们的方法接近。Jet-QSparse [47] 同样将 NVFP4 缓存与校准键居中相结合，并在 Blackwell（GB300）上为每个序列和层最多保留两个 BF16 页，同时使用 FP8 预填充。KVQuant 和 IntactKV [29] 将第一个令牌保留为高精度。同期工作 Easy-Kuant [1],报告称，在 FP8 和 NVFP4 KV 缓存下，Qwen2.5 系列在 vLLM 中同样会发生崩溃；该工作同样发现误差通过键引入，并将其归因于 RoPE 后键通道在每个令牌上的数值都很大；逐层统计量会选出负责这一问题的一个至四个层（包括第 0 层），并将其保留为 BF16，缓存容量为 NVFP4 的 85%。对于这一系列模型，这些通道表现为恒定键偏置；在我们的阶梯实验中，移除经过校准的逐通道均值即可修复该问题（表 11）。这一操作不会改变注意力，也不占用额外容量；两项工作都发现，将前几个令牌保留为 BF16 无法修复崩溃（在我们的阶梯实验中，只有移除均值后，汇聚令牌才会发挥作用）。我们在 Hopper 上复现了这些准确率结果；我们新增的内容见第 8 节（“推理与第 0 层”段落）。

服务。 vLLM 0.33 通过 FlashAttention-3（默认选项）或 FlashInfer 提供 FP8 KV（在我们的测试中，使用 vLLM #66590 [19], 时，FlashInfer 在 H100 上处理 FP8 KV 的速度快于 FlashAttention-3；在 H200 上也如此），还提供由 FlashInfer 解码的 NVFP4 KV 缓存，以及 4 位 UltraQuant 格式（采用 K 旋转的 MXFP4） [7]。在 Hopper 上使用 FlashInfer 和 FP8 KV 时，vLLM 会使用 FlashInfer 附带的 TensorRT-LLM XQA 内核 [38], 进行解码，但前提是能够成功通过对英伟达制品服务器的网络检查；否则，它会静默回退到 FlashInfer 自带的解码内核。我们会在每次运行中记录最终使用的内核（第 5 节）。在英伟达平台上，UltraQuant 会回退到 Triton [51] 解码内核和 FlashAttention-2 [8] 预填充（后者已由 vLLM 拉取请求 #47531 [55]解决）。FP8 基线也在不断变化：有多个待处理的拉取请求，旨在将 FlashAttention-4 [60]的原生 FP8 引入 vLLM 的 SM90 FP8-KV 路径 [20, 21]；我们会在第 6 节（表 5）测量 FlashAttention-4 原生 FP8。

3 为什么现有的 4 位内核会停滞

我们考察的 4 位内核要么每字节执行的指令数达到 Hopper 指令预算的两到六倍，要么受制于串行化加载和屏障，或者两者兼有；要达到带宽上限，必须同时解决这两个问题。

发射预算。 H200 每秒最多可发射约 33 万亿条线程指令（132 个流式多处理器$\times$ 4 调度器、 $\times$ 32 通道、 $\times$ 10.98 GHz），并读取 4.8 TB/s，因此，以满带宽运行的内核每字节大约可花费 7 条线程指令；按每行 128 个值、占用 72 字节计算，即每个值约 3.9 条指令。这符合 FOSCAR 的公式 2。使用相同公式和规格表中的数值，H100 SXM 和 A100 的预算约为 10，而 L40S 约为 53；这就是为什么在 A100 或 L40S 上能够发挥作用的软件4位解码 [28] ，在 H200 上仍可能受发射限制。

测得的指令计数。 由于我们的集群无法提供 GPU性能计数器，因此我们使用 NVBit（英伟达的二进制插桩工具）（$B = 32$,$S = 16$键；表 2）统计了已执行的指令。

vLLM 的 UltraQuant 内核超出预算约六倍（每字节 42 条指令，其中 79% 是整数、位运算和选择运算，并使用按字节宽度的全局加载和共享内存存储），我们的第一个 Triton 内核以及 FlashInfer 则超出约两到三倍；大多数额外指令是整数和位运算（IMAD、LOP3、PRMT）、乘法和类型转换，此外，Triton 还会让解码值经过共享内存往返（每个瓦片三到五次，具体取决于 Triton 版本）。

表 2：测得的指令计数（NVBit、H200、 $B = 32$、 $S = 16$键）。推算的发射利用率：每字节 =条指令；测得的带宽占比 × ÷ 7。UltraQuant：vLLM 0.33 用于 `--kv-cache-dtype ultraquant_4bit` 的解码内核（MXFP4，每行字节数为 68），直接在由 vLLM 自有存储内核写入的分页缓存上调用，运行于 H200 SXM。

内核	每个值的指令数	已使用带宽	推算的发射利用率	主要问题
我们的第一个 Triton NVFP4 内核	11.7	17%	52%	both
FlashInfer NVFP4	8.3	19%	40%	both
vLLM UltraQuant（Triton，按发布版本）	22.5	8%	50%	both
BitDecoding INT4（Hopper 分支）	3.6，无自旋等待	25%	约 24%	等待

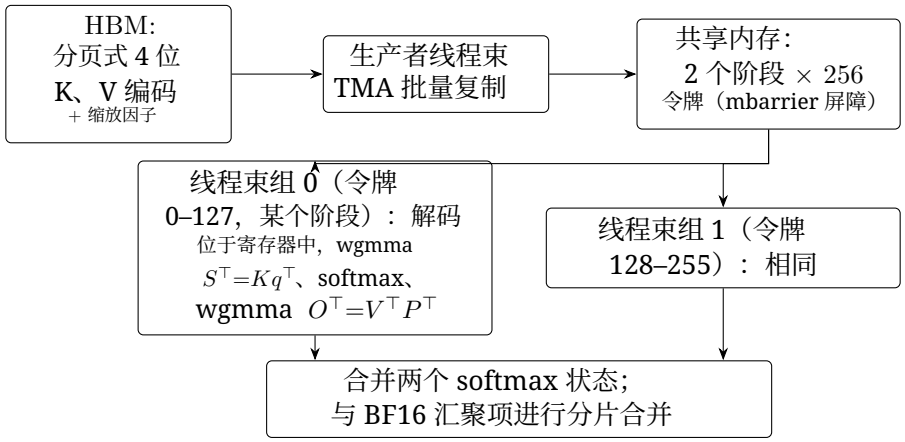

图 1：内核结构。加载、解码和两次矩阵乘法相互重叠：当线程束组对一个阶段进行解码和乘法运算时，生产者会填充另一个阶段。解码后的键和 $V^\top$ 是 wgmma 的 A 操作数，从寄存器中读取；查询头是 N $= 8$ 维度。

BitDecoding 的解码成本很低（每个值约 1.3 条指令），在不计入自旋等待的情况下能够控制在预算内，但它将解码与矩阵乘法串行化，有 38% 的已执行指令耗费在屏障上的自旋等待中，并将 4 个查询头放在 wgmma 的 64 行维度上，因此其三分之一的有效工作是在填充行上执行 softmax。我们的内核解决了这两个问题。

4 设计

该内核通过 warp specialization 解决等待问题，并在写入时选择缓存布局来固定指令数量；INT4、NVFP4 和 MXFP4 共用一种结构（图 1）。

1. **重叠：一个 TMA 生产者和两个计算线程束组**。一个生产者线程束通过 TMA 批量复制，将 K 和 V（编码值与缩放因子）以 256 令牌为阶段流式传入共享内存，并使用 mbarrier 屏障进行双缓冲。每个计算线程束组（4 个线程束）负责每个阶段的一半，并维护自己的在线 softmax 状态；两个线程束组仅在最后合并一次。

2. **操作数交换，解码值直接来自寄存器**。QK 计算为 $S^\top = K \cdot q^\top$，PV 计算为 $O^\top = V^\top \cdot P^\top$，其中 `wgmma.m64n8k16`。解码后的 K 或 $V^\top$ 是 A 操作数，wgmma 从寄存器中读取它，因此解码值不会返回共享内存。查询

头采用 N 维度（$N = 8$；4 或 8 个实际注意力头），因此 softmax 无需处理填充行。只有 P 会经过共享内存（每个线程束组 2 KB）。对于较小的 M，交换操作数是标准的通用矩阵乘法技术（例如 [33, 65]），而 BitDecoding-FP4 会在 Blackwell 上交换两次注意力矩阵乘法的操作数 [11]；这里则将较小的查询组放入 N，并让软件解码的 K 和 $V^\top$ 成为 wgmma 的寄存器操作数。

3. **写入时布局**。 我们可以控制缓存格式，因此写入时会对每一行键和每个含 256 个令牌的值块进行置换。这样，每个线程的寄存器片段都位于连续的 16 字节块中：每个线程每行只需执行一次 128 位共享内存加载，不会发生存储体冲突，也无需数据洗牌。缩放因子也会以相同方式置换。Marlin [12]和 TensorBridge [65]对 4 位权重也采用同样的方法，在离线阶段完成；FuTransformer [53] 则用于 INT4 键副本，BitDecoding [10] 会以张量核心感知布局打包键值；而这里的置换是在缓存写入内核中完成的。

4.**低成本重构**。 INT4 使用 FP16 魔数 [24]：`(w & 0x000F000F) | 0x64006400` 为两个编码值提供 $1024 + c$，再通过掩码 `0x00F000F0` 得到 $1024 + 16c$，无需移位；因子 16 折叠进 q 中（每个值需要 0.625 条指令）。对于 E2M1，将指数位和尾数位置于 FP16 的第 9–11 位，并将符号位置于第 15 位，可使数值恰好乘以2^{-14}，其中也包括非正规数 0.5。

5. **缩放因子**。 INT4 的逐令牌缩放因子和零点通过折叠乘法运算来处理（FOSCAR 的仿射折叠）。NVFP4 的每 16 个元素一个的 E4M3 缩放因子，以及 MXFP4 的每 32 个元素一个的 E8M0 缩放因子，都是通过成对的半精度乘法逐寄存器应用的，因为 Hopper 没有块缩放张量核心。在较早版本的内核中，NVBit 统计得出，FP4 每个值需要 3.8 条指令，而 INT4 需要 2.6 条；缩放因子约占这一差距的 0.7 条，其余大部分则来自 E2M1 重建。

测试发现的数值陷阱。 (1) INT4 的 1024 魔数偏移量在乘法后减去，导致数千个令牌后相对误差增至 2×10^{-2}；这与 Hopper 的张量核心累加并非采用 IEEE 舍入到最近值一致（FP16 输入和 FP8 [23] 输入都会被截断）；在乘法前减去该偏移量（每个寄存器执行一次 HADD2）可将误差降至 1.3–2.3×10^{-4}。我们的随机数据测试发现了这个问题，但只有在分块累加了数千个令牌后才会出现。(2) 对真实的 Llama-3.1-8B 键值而言，强汇聚令牌旁的数千个小注意力权重会落入 FP16 非正规数范围；将 P 按 2^{10} 缩放即可解决问题，且没有额外开销。随机测试数据不会显现这一问题。

分页与汇聚令牌。 FP4 内核读取 vLLM 风格分页缓存（64 令牌页、块表、每序列长度）的速度与读取连续缓存相同（0.99–1.035×，页面顺序已打乱）；INT4 内核仅支持稠密缓存。四个 BF16 sink 令牌保存在 4 位缓存之外，并在分片合并时加入。该内核每个 KV 头处理 4 或 8 个查询头；其他 $G \le 8$ 通过对 q 进行零填充来运行。

5 实验设置

硬件。 除非另有说明，均使用 H200 SXM（141 GB，驱动程序 570.86.15）；部分准确率测试和两个标注的速度数据使用了 H200 NVL 节点（驱动程序 595.58.03），其速度较慢；所有速度比率均未混用这两种设备。H100 内核数据使用 H100 SXM5（80 GB，驱动程序 545.23.08）。带宽占比的计算方式为读取的 K 和 V 字节数（$B \times H_{kv} \times S \times 2\times$ 每行字节数；不计 q、输出、块表、部分结果和汇点）除以规格表峰值（H200 为 4.8 TB/s，H100 为 3.35 TB/s）；大型设备复制可达到 H200 峰值的 86–90%。

内核基准测试。 CUDA 12.8（nvcc `-O3, sm_90a`）、PyTorch 2.14.1、Triton 3.8.0、FlashInfer 0.7.3、提交版本为 9089e1f 的 FOSCAR、BitDecoding 的 `hopper` 分支（584d0d3），以及未经修改的 vLLM PR # 53060 源码。K、V 和 q 均为随机值，形状采用 Llama-3.1-8B 的头部形状（8 个 KV 头，$G = 4$，头维度为 128）。每次耗时取预热 5 次后 50 次调用的中位数，使用 CUDA 事件计时，并在每次调用前清空 L2；对于每种形状，我们在相同计时器下从 1–64 中选取最佳拆分数量，而 FlashInfer 和其他基线使用各自的拆分启发式，因此内核表略微偏向我们的内核。插件的固定规则（见下文）在主内核表的 11 种形状中，有 9 种选出的拆分数量与逐形状搜索结果相同，其中包括加速比最低（1.43×, 1×16K）和最高（2.31×, 16×128K）的形状；仅在 1×64K 和 1×128K 两种形状上不同（使用 16 次拆分而非 32 次）。在 vLLM 内部（第 7 节），我们的拆分数量未经调优，而是根据固定规则 $\min(16,\ \lfloor 264/(B \cdot H_{kv}) \rfloor)$ 确定，且至少为 1。采用完整方案的表格计时的是 CUDA 图重放。内核表中的所有列均来自同一块 GPU 上的一次作业。

vLLM。 vLLM 0.33.0（CUDA 13 wheel；在此环境中使用 flashinfer-python 0.7.2.post1，比内核级 FlashInfer 早一个补丁版本），我们的插件使用 nvcc 13.2 构建。该配方使用离线校准的静态逐头 K 均值和 64 令牌页；FP8 基线使用 vLLM 的默认配置，分别采用 FlashInfer（`--attention-backend FLASHINFER`）或默认的 FlashAttention-3 路径。GPU 内存利用率始终为 0.90。vLLM 内部的 FP8 数据来自日志显示 FlashInfer 已解析的解码内核为 XQA（`decode_backend=xqa`）的运行。此前少数运行早于该日志记录功能：第 7 节的端到端延迟、吞吐量和预填充对比，以及 Qwen3-32B 和 Qwen3-30B-A3B 的逐步数据行。这些运行使用相同的脚本，并通过集群代理执行；在该环境下，日志中的运行会解析为 XQA，而该时期的内核性能分析结果也显示使用的是 XQA 内核。服务运行设置了 `VLLM_HAS_FLASHINFER_CUBIN=1`，即 vLLM 用于指定本地可用内核的开关，因为服务脚本绕过集群代理，否则 vLLM 的网络检查会回退到 FlashInfer 的原生解码（此前一组服务运行确实如此，但本文未采用）。纯解码时间使用 `vllm bench forward`（虚拟权重、CUDA 图、每种形状取中位数，所有配置在同一作业中运行）。端到端数据使用 `vllm bench latency`（每个输出令牌时间 $=$ $(t_{256} - t_1)/255$，一次预热和三次计时迭代）以及 `vllm benchthroughput`（随机提示，256 个请求）。

准确率。 贪心解码；对 MATH-500 答案进行比较前，会先对最后一个框选表达式进行规范化。成对比较采用置换检验，基于每道题的正确性进行分析；若有多次运行，则取各次运行的平均值。

6 内核结果

在 H200 上，我们的 NVFP4 内核在 11 种形状下比 FlashInfer 的 FA2 FP8 内核快 1.4–2.3× 倍（图 2、表 3），比其 XQA FP8 内核快 1.2–2.1× 倍；在 11 种形状中的 9 种下，XQA FP8 内核是两者中较快的（表 4）。加入 FlashAttention-4 原生 FP8 后，我们的内核在每种形状下比最快的 FP8 内核快 1.12–1.49× 倍（表 5）；在测量的所有 15 种形状下，我们的 INT4 在字节数相同的情况下都快于 FOSCAR 的 INT4。

与 FlashAttention-4 原生 FP8 对比。 一个开放的拉取请求将 FlashAttention-4 [60] 的原生 FP8（E4M3 查询、键和值）引入 Hopper，另一个请求则让 vLLM 为所有 SM90 FP8-KV 模型选择该实现 [20, 21]；审查者表示，这样一来它就会成为 vLLM 在 Hopper 上的 FP8 路径，因此尽管两项请求都尚未合并，我们仍对其进行了测量（FlashAttention 提交 `8ef2c611`，vLLM 提交 `0d99666a`，2026年11月22日）。这是我们测得最快的 FP8 解码：85–89%

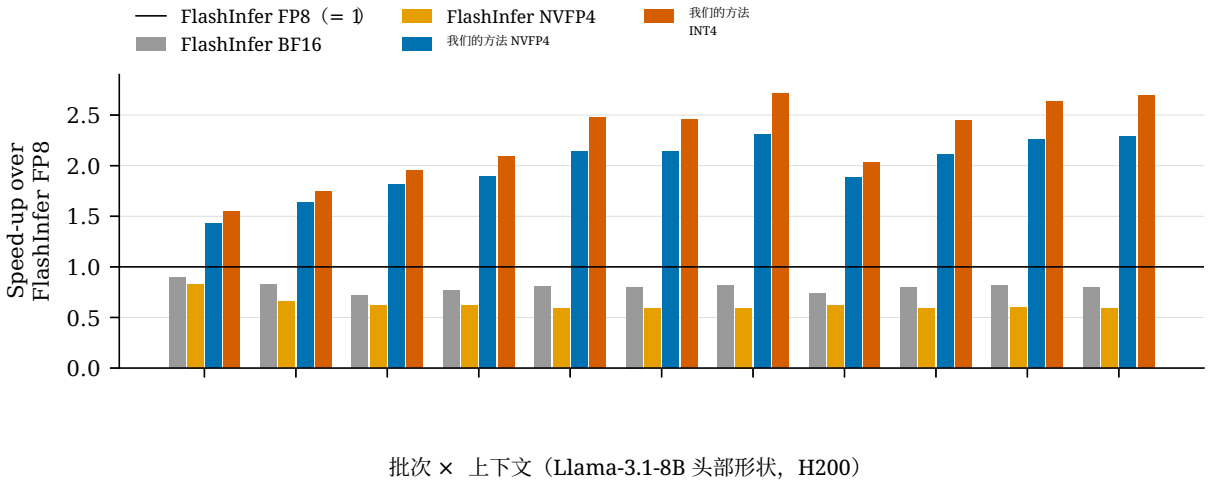

图 2：在 H200 上的全部 11 种形状中，相对于 FlashInfer 的 FA2 FP8 内核的内核速度（单个作业；我们的耗时包括分片合并）。FlashInfer NVFP4 在每种形状下都比其 BF16 更慢。

表 3：相对于 FlashInfer 的 FA2 FP8 内核的加速比（H200、Llama-3.1-8B 头部形状、单个作业和 GPU；我们的耗时包括分片合并；选定形状）。与其 XQA 内核的对比见表4。

$B \times S$	FlashInfer FP8（毫秒）	FlashInfer NVFP4	我们的方法 INT4	我们的方法 NVFP4	我们的方法 MXFP4
1×16键	0.033	0.83×	1.54×	1.43×	1.46×
1×128键	0.116	0.62×	1.95×	1.81×	1.83×
8×64键	0.400	0.59×	2.48×	2.14×	2.14×
32×16键	0.395	0.59×	2.44×	2.11×	2.11×
16×128键	1.557	0.59×	2.72×	2.31×	2.26×
128×16键	1.542	0.59×	2.70×	2.29×	2.27×

表4：与 FlashInfer 的 XQA FP8 解码内核对比（H200 SXM、单个任务、全部 11 种形状；FlashInfer 0.7.3）。XQA 的调用方式与 vLLM 相同（HND 布局、BF16 查询）；展示的是页大小为 16、32、64 和 128 时的最佳结果，以及其相对于 FP8 字节的带宽占比。我们的方法：NVFP4，采用最佳分块；vLLM 内部使用的固定拆分规则与其相差不超过 4%，但 1×128键处除外（1.79×）。在相同的 FP8 缓存上，XQA 的输出与 FA2 的差异最多为 4.8×10^{-3}。

$B \times S$	FA2 FP8（毫秒）	XQA FP8（毫秒）	XQA 带宽	我们的方法 NVFP4（毫秒）	我们的方法与 XQA
1×16键	0.033	0.027	26%	0.022	1.24×
1×64键	0.070	0.069	40%	0.042	1.66×
1×128键	0.119	0.121	46%	0.063	1.93×
4×32键	0.118	0.125	45%	0.061	2.05×
8×64键	0.399	0.343	65%	0.187	1.83×
16×32键	0.402	0.342	66%	0.187	1.83×
16×128键	1.552	1.314	68%	0.678	1.94×
32×4键	0.118	0.080	70%	0.061	1.31×
32×16键	0.396	0.255	88%	0.187	1.37×
64×16键	0.794	0.615	73%	0.352	1.75×
128×16键	1.552	0.993	90%	0.675	1.47×

表 5：纳入 FlashAttention-4 原生 FP8 后，在每种头部形状下与最快的 FP8 解码内核进行对比（H200 SXM、单个作业、CUDA 图重放、针对每个内核采用最佳页大小和拆分数量、Llama-3.1-8B 头部形状）。“最佳 FP8”取 XQA 和 FlashAttention-4 耗时中较小者。FlashInfer 的 FA2 FP8 内核是我们测量的第三个 FP8 内核，但未包含在本次作业中；在表 4 的作业中，它最多仅比 XQA 快 6%，且只在两种头部形状下如此，而在本文中 FlashAttention-4 的耗时仅为 XQA 的 0.67 倍，因此它在任何头部形状下都不是最快的。

$B \times S$	我们的方法 NVFP4（毫秒）	XQA FP8（毫秒）	FA4 FP8（毫秒）	最佳 FP8 / 我们的方法
$1 \times 16K$	0.0214	0.0275	0.0240	1.12×
$1 \times 64K$	0.0412	0.0708	0.0528	1.28×
$1 \times 128K$	0.0623	0.1253	0.0839	1.35×
$4 \times 32K$	0.0606	0.1239	0.0824	1.36×
$8 \times 64K$	0.1877	0.3469	0.2589	1.38×
$16 \times 32K$	0.1882	0.3435	0.2589	1.38×
$16 \times 128K$	0.6926	1.3289	1.0291	1.49×
$32 \times 4K$	0.0605	0.0772	0.0782	1.28×
$32 \times 16K$	0.1876	0.2547	0.2551	1.36×
$64 \times 16K$	0.3517	0.6110	0.5014	1.43×
$128 \times 16K$	0.6789	0.9988	1.0482	1.47×

在批次为 8 及以上时达到 H200 峰值带宽的 1. 32×4（在 32×4键时为 71%），并且在 11 种形状中的 8 种上比 XQA 更快，最高快 1.5×。我们的内核在所有情况下都领先于它，但领先幅度小于它领先 XQA 的幅度（表 5）。这里的耗时是 CUDA 图重放耗时，与 vLLM 的解码方式一致；在表 3 和表 4 采用的急切执行、刷新 L2 的计时方式下，FlashAttention-4 在此节点上每次调用还会产生约 $30\,\mu$ 秒的主机开销，因此我们的领先幅度会变为 1.23–1.73× 。所以，在 11 种形状中的 8 种上，图重放数据是较保守的结果（其余三种形状下，图重放的领先幅度最多高出 3%）。与 FP32 参考值相比，它的误差小于将查询舍入为 E4M3 所引入的误差。

- **带宽**。 在表 3 的任务中，NVFP4 达到 H200 峰值的 67%（$B = 32$）至 75%（$B = 128$，两者均为 16K），而 INT4 达到 74–83%；在批次为 1 时，其达到的比例要低得多（第9节）。在 H100（3.35 TB/s）上，批次为 8 及以上时，NVFP4 达到 79–86%，INT4 达到 82–90%。
- **比率中有多少来自基线**。 FlashInfer 的 FA2 FP8 内核在 32×16K 时达到 H200 峰值的 57%（处理 1.07 GB 用时 0.395 毫秒）；其 XQA 内核在该处达到 88%，在 128×16K 时达到 90%，接近设备复制所能达到的水平（86–90%）。因此，与 XQA 相比，我们的优势低于 128/72 $= 1$ 的字节比率的情况是 XQA 接近带宽上限运行时（在 32×4K、 32×16K 和 128×16K 时为 1.31–1.47× ，此时我们达到 51–75%），而在 XQA 未达到带宽上限时则接近或高于该比率（批次为 4–64 且上下文较长或序列较多时为 1.75–2.05× ，此时 XQA 达到 45–73%）。在批次为 1 时，XQA 在 64K 和 128K 下相比 FA2 的提升很小，而我们快 1.24–1.93× 倍。FlashAttention-4 原生 FP8 在 H200 [20] 上报告的用时意味着其带宽占比高于 FA2 （我们根据其表格自行计算，任务不同）；我们测得，在批次为 8 及以上时，其达到峰值的 85–89%（表 5）。
- **我们的内核耗时分布（移除消融）**。在 H200 SXM 上的一项任务中，将拆分数量固定为完整内核的最佳值后，我们对多个变体进行计时；每个变体在读取相同字节的同时省去一部分工作（表 6）。不进行 E2M1 重建时，内核耗时为 0.87×，在 $B \geq 8$ 时达到峰值性能的 77–85%；省去两次矩阵乘法时，耗时为 0.92–0.94×；复制相同字节的设备复制则可达到峰值性能的 87–89%。因此，解码确实会给我们的内核带来开销，但占比不高：在 $B \geq 8$ 时约占其耗时的 13%；与复制性能上限之间的其余差距，则来自流水线、矩阵乘法和 softmax。将两行的解码交错执行以提高

表 6：移除消融（H200 SXM、单个作业、Llama-3.1-8B 头部形状，拆分数量固定为完整 NVFP4 内核的最佳值）：相对于完整内核的耗时（越低越快）以及峰值带宽分数。每个变体移除一部分工作，并读取相同数量的字节；“copy”表示对相同字节执行设备读写。

变体	1×64键	8×64键	32×16键	128×16键
完整 NVFP4 内核（毫秒）	0.042	0.188	0.188	0.675
无 E2M1 重建	0.98	0.87	0.87	0.87
无矩阵乘法	0.97	0.92	0.92	0.94
交错解码（指令级并行）	1.08	1.10	1.11	1.18
MXFP4	0.98	1.01	0.99	1.01
INT4	0.93	0.87	0.87	0.85
完整内核，峰值分数	38%	67%	67%	75%
无重构，峰值分数	39%	77%	77%	85%
相同字节的副本	76%	87%	87%	89%

指令级并行性的变体慢了 8–18%，因此我们保留了更简单的方案。第 3 节的指令预算诊断针对的是 Triton 和 FlashInfer 内核，它们执行的指令数是预算的两到六倍；我们的内核在 FP4 下执行的指令数约等于预算（见下一项），而移除解码后只节省 13%，因此对我们的内核来说，剩余瓶颈并非仅仅是解码指令。

- **H200 上的 FP4 受限于计算能力，而不是字节数。** MXFP4 读取的字节数比 NVFP4 少 6%，但耗时相同。早期版本每字节执行的指令约占名义发射预算的 96%；但后来移除自旋和地址指令后，性能最多提升 2%，因此仅凭指令总数无法解释这一限制；每线程束依赖链是一个可能因素，但由于没有停顿计数器，我们无法测量。在 H100 上，由于每条指令对应的带宽较低，两种格式的运行都接近内存限制。

- **采用完整的精度方案**（K 均值去除、旋转、四个 BF16 汇聚项、CUDA 图重放）时，NVFP4 的性能达到 FlashInfer 的 FA2 FP8 内核的 1.77–2.17× 倍（针对 $B \geq 32$）；汇聚合并和 q 旋转在 $B \geq 4$ 时带来的开销为 0–7%，但在 16×128K 时除外（两个任务在 CUDA 图下开销为 +9–13%，不使用 CUDA 图时约为 1%）；在 $B = 1$ 时开销为 12–31%。

- **与 FOSCAR 相比**（使用其发布的代码、相同的流程和 K/V，以及其冷L2计时器）：在 H200 上，我们的 INT4 在字节数相同（每个令牌和头 68 字节）时快 1.08–1.61× 倍，而我们的 NVFP4 内核速度为 FOSCAR INT4 的 0.99–1.40× 倍。采用完整的精度方案时，我们的 NVFP4 路径在 1×16K 时慢于 FOSCAR（1.22× 倍对比 FlashInfer 的 FA2 FP8 内核的 1.46× 倍），在 4×32K 和 32×4K 时也更慢（速度为 FOSCAR 的 0.84–0.97× 倍），在其他 12 种形状下则更快；这次 H200 测试使用了较早的未融合 q 预处理。FOSCAR 在 H200 上的耗时与其公布的 H100 数据相符，因此没有利用 H200 更高的带宽。在 FOSCAR 针对其调优的 H100 上，我们的 INT4 快 1.02–1.31× 倍，NVFP4 快 0.94–1.21× 倍，因此优势幅度不大。

指令数量的下降过程如下：首个 K侧原型以每个值 3.0 条指令的开销达到了 60% 的带宽利用率。随后，NVBit 展示了三个编译器层面的浪费（一个 LOP3 被拆分为两条指令、每条指令都重新计算 wgmma描述符，以及空闲通道执行 softmax 工作）；修复这些问题后，开销降至每个值 1.95 条指令，带宽利用率达到 72%（稳态下为 87%）。该原型仅计算了 $Kq^\top$（不执行 softmax，也不计算 V）；完整内核达到了 67–75%（表 6）。

表 7：每步纯解码时间（`vllm bench forward`、Llama-3.1-8B、H200 SXM，所有配置在同一作业中；100 次 CUDA 图重放的中位数）。FP8：使用 XQA 解码的 FlashInfer（记录已解析的内核）；vLLM NVFP4：`--kv-cache-dtype nvfp4`，FlashInfer 的原生解码；我们的方法：完整配方，汇路径上的每个序列。加速比均相对于 FP8。

$B\times$ 上下文	FP8（毫秒）	vLLM NVFP4（毫秒）	我们的方法（毫秒）	vLLM NVFP4	Ours
1×16键	4.97	5.52	4.92	0.90×	1.01×
1×32键	5.57	6.42	5.31	0.87×	1.05×
8×16键	7.50	10.43	6.69	0.72×	1.12×
8×32键	10.25	15.69	8.06	0.65×	1.27×
32×16键	12.66	26.36	10.94	0.48×	1.16×
32×32键	20.42	47.80	16.43	0.43×	1.24×
64×16键	24.80	48.23	16.74	0.51×	1.48×
16×64键	26.90	47.59	16.35	0.57×	1.64×
128×8键	21.81	49.18	17.43	0.44×	1.25×

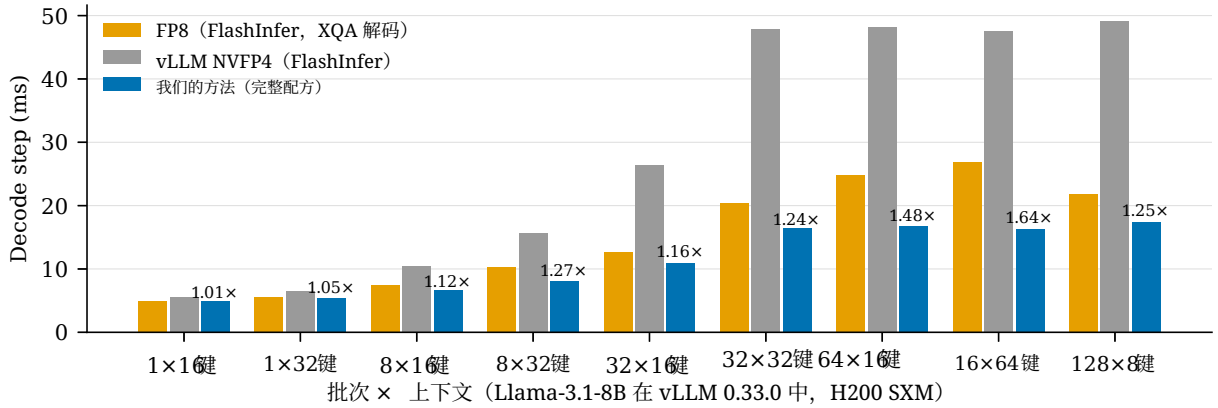

图3：vLLM 中的解码步骤耗时（数据来自表 7）；标签表示加速比。

7 vLLM 中的端到端

在 vLLM 0.33.0 中，按照第 8 节每个准确率数字所使用的完整方案，我们的插件在批次 8 及以上时的解码速度是 FlashInfer FP8（XQA 解码）的 1.12–1.64× 倍，在批次 1 时则与其持平（每个解码步骤为 1.01–1.05× 倍，按端到端每个输出词元时间计算为 0.96–1.03× 倍）。vLLM 自带的 NVFP4 KV 路径在批次 8 及以上时比 FP8 慢 1.4–2.3× 倍。我们的缓存可容纳 1.63× 倍数量的令牌，而首 token 时间慢 1.5–6%。在类推理请求流下，二者在 Llama-3.1-8B 上均能以最高 1.25 请求/秒满足延迟目标；我们的方法每个输出词元时间更短。超过该速率后，FP8 的缓存会先填满。单个 H200 能维持的最高速率，在 Llama-3.1-8B 上为 FP8 的 1.23–1.32× 倍，在 Qwen3-32B 上为 1.45–1.80× 倍，在 Qwen2.5-72B 上为 1.67–2.20× 倍（表 9）。

该插件使用我们的内核替换 vLLM 4位后端中的缓存写入、解码和预填充注意力，并在 BF16 下通过 FlashAttention-3 运行预填充注意力。每步纯解码时间见表 7 和图3。

与仅第 0 层使用 BF16 的 NVFP4 相比，均值移除和汇点会带来 1.2–3.2% 的开销（两者均来自 H200 SXM 节点上较早的一次作业；表中的作业不包含该配置）。在另外两个模型上（每个模型各进行一次作业，H200 SXM，上下文长度最高为 32K）：Qwen3-32B 在批次 1 时为 1.00–1.02×，在批次 8 及以上时为 1.05–1.18×；Qwen3-30B-A3B（混合专家）在批次 1 时为 0.94–0.95×，在批次 8 及以上时为 1.11–1.32×

批次 8 及以上。在 UltraQuant 自有格式（采用 K 旋转的 MXFP4、每个令牌占用相同字节数、使用 vLLM 默认的四个 BF16 边界层；一次作业，H200 SXM，形状与表 7 相同）下，我们的内核每步解码速度比 vLLM 的原生 UltraQuant 后端快 1.9–5.1× 倍；该后端在英伟达平台上会回退到 Triton，并且在批次 1 时相当于 FlashInfer FP8 的 1.03–1.06× 倍，在批次 8 及以上时为 1.08–1.48× 倍；原生后端比 FP8 慢 1.8–4.4× 倍。

端到端（`vllm bench latency`，完整方案与 FlashInfer FP8 对比，在一台 H200 SXM 节点上连续运行两个作业）：在 1×16K（0.96×）时，每个输出令牌时间分别为 5.30 和 5.08 毫秒；在 1×32K（1.03×）时，分别为 5.55 和 5.72；在 8×16K（1.12×）时，分别为 6.95 和 7.77；在 8×32K（1.27×）时，分别为 8.60 和 10.94；在 32×16K（1.09×）时，分别为 12.77 和 13.94。离线吞吐量（输入长度为 8K 和 16K，输出 512 个令牌，256 个请求）在相同作业中为 1.06–1.10× 。在相同内存设置下，键值缓存可容纳 2,867,200 个令牌，而 FP8 可容纳 1,761,600 个（1.63×）。解码主导吞吐量（输入 16K，输出 2,048 个令牌，256 个请求）相较于 FlashInfer FP8 为 1.17×，相较于默认的 FlashAttention-3 FP8 路径为 1.40×；这三次运行是在 H200 SXM 上分别执行的独立作业，且 FP8 运行早于一次插件变更，该变更不会影响这些运行。在 64×32K 时（两种配置在同一作业中运行，H200 SXM），FP8 的缓存无法容纳全部 64 个序列（210 万个令牌，而容量为 176 万），而我们的方法可以；每个输出令牌的解码时间分别为 38.7 和 52.2 毫秒（1.35×）。FP8 的数值包含其序列等待缓存空间的时间，因此这是容量带来的效果，而非解码速度差异；整个请求（输出 256 个令牌）仍需 86.8 秒，而对照组需 85.0 秒（慢 2%），因为预填充 210 万个提示词 token 的速度慢了 7%（77.0 秒，而对照组为 71.7 秒）。

- **为什么收益小于内核的 2×。** 每个步骤还要读取 16 GB 的 BF16 权重，按 H200 的峰值带宽计算，至少需要 3.3 毫秒；相比之下，在 8×16键 下，我们整个步骤耗时 6.69 毫秒。注意力只占步骤的一小部分，且上下文和批次越大，收益越明显。在 8×16键 下进行的逐内核性能分析（同一作业中测试两种配置，H200 SXM，32 个解码步骤）展示了具体差异：我们的解码内核每步耗时 1.50 毫秒，而 FlashInfer 为 2.88 毫秒（1.92×）；加上汇聚令牌合并、缓存写入、q 旋转以及 BF16 格式的第 0 层后，注意力相关内核总耗时分别为 2.00 毫秒和 3.02 毫秒，约占我们步骤耗时的四分之一（性能分析器下为 7.55 毫秒，对比 8.43 毫秒）；两者的通用矩阵乘法耗时均约为 4.0 毫秒。

- **为什么以 FlashInfer FP8 作为基线。**在我们的运行中，vLLM 的默认 FP8 路径（使用 FP8 查询的 FlashAttention-3）在大批次下比 FlashInfer 更慢（H200 NVL 上的 32×32键：每步 30.2 毫秒，而 FlashInfer 为 21.6 毫秒，使用相同作业），因此上面的所有比率都是相对于 FlashInfer 计算的；相对于默认路径，这些比率会更高（解码主导吞吐量为 1.40× 对 1.17×）。vLLM 中的 FlashInfer FP8 解码使用的是 XQA 内核：使用 `VLLM_HAS_FLASHINFER_CUBIN=1`强制启用该内核后，每步耗时最多变化 0.3%（与表 7 使用相同作业）。

- **预填充。** vLLM 的 4 位后端使用 FlashAttention-2 执行预填充注意力；我们改用 BF16 下的 FlashAttention-3（vLLM PR #47531 也提出了这一方案）。仅预填充仍比 FlashInfer FP8 慢 7%（同一作业中，8 个包含 32,000 个令牌的提示词：9.29 秒对比 8.66 秒），端到端首 token 时间也慢 1.5–6%。对当前代码进行逐内核性能分析后发现，差距来自预填充注意力（每层比 FlashInfer 的 FP8 预填充注意力慢 6–7%）、其间的复制操作（输出复制、连续存储的 K 和 V、sink 捕获），以及继续预填充时的反量化；我们的缓存写入内核耗时与 FP8 相同。在同一份性能分析中，两种配置完全相同的模型矩阵乘法，在 GPU 达到功率上限时，我们的方法慢 0.1–5.2%；我们推断这是因为它们共享功率预算，但尚未单独验证这一影响。FP8 预填充路径并不能弥补差距：在我们的测试中，FlashAttention-3 的 FP8 预填充内核本身就比 BF16 版本慢 3–12%，而在其前面量化 q、K 和 V 的开销超过了节省的时间，因此我们的 FP8 预填充选项整体上比 BF16 慢 5–7%（此外，在较早的一种全层 MXFP4 配置中，困惑度从 +0.68% 升至 +1.51%）。这里更快的是 FlashInfer 的 FP8 预填充内核，而不是 FP8 本身。将预填充和解码分别置于不同

表 8：负载下服务（Llama-3.1-8B、H200 SXM、使用 XQA 解码的 FP8）。满足服务级别目标的请求占比 [95% 请求启动区间], 峰值 KV 缓存用量，以及 FP8 的抢占次数（我们的方法：所列类推理运行中没有抢占；长上下文运行在 1.5 时有 13 次）。使用一个 vLLM API 服务器进程（默认设置），除非另有说明，均使用种子0。在 1.25 时，逗号分隔的数值分别对应种子0和种子1，且两者均如此。在 1.35 和 1.5 时，逗号分隔的数值表示不同运行：FP8 使用种子0和种子1；我们的方法使用种子0、在相同到达序列下重复一次种子0（1.35 时在同一节点上，1.5 时在另一台 H200 SXM 节点上），以及种子1。“4 API”：种子0和种子1，使用 `--api-server-count 4`。平均TPOT 指每个输出令牌时间。

工作负载	速率 FP8：SLO占比	键值 我们的方法：SLO占比	KV备注
类推理 （1K 输入，8K 输出）	0.75 1.00 1.0 1.00 1.25 1.00 , 1.00 1.35 0.89，0.94 0.88, 0.94 1.4 0.82 [0.80，0.85] 1.5 0.69，0.75 0.69, 0.75	32% 1.00 59% 1.00 93–97% 1.00 , 1.00 100% 0.82，1.00，0.86 100% 1.00 , 1.00 100% 1.00 100% 0.81、1.00、1.00 100% 1.00 , 1.00	16% 平均TPOT 13.3 vs 10.3 毫秒 29% 平均TPOT 19.5 vs 14.3 毫秒 46% 平均TPOT 28.2 对比 20.6 毫秒 55–59% FP8：628、656 次抢占 58–61% 4 个 API；FP8：683、651 62% FP8：750 次抢占 78–79% FP8：1513、1375 次抢占 79–81% 4 个 API；FP8：1494、1374
长上下文 （输入 16K，输出 1K）	0.5 0.99 1.0 0.87 [0.83, 0.90] 1.5 0.14 [0.11，0.18]	19% 0.99 66% 0.84 [0.80、0.88] 100% 0.04 [0.03，0.06]	11% 35% 100% 均为预填充受限
预填充密集 （输入 16K，输出 512）	0.75 0.98 [0.96，1.00] 1.25 0.87 [0.83，0.90]	19% 0.94 [0.90，0.97] 57% 0.77 [0.73，0.82]	第99百分位首令牌时间为 2.2 秒，而非 2.5 秒，降低 12% 37% 第99百分位首令牌时间 4.0 对比 4.4 秒

GPU，解码池不运行预填充。

负载下服务。我们使用泊松到达，以固定请求速率运行了 `vllm serve`（Llama- 3.1-8B、H200 SXM；随机提示词的输入和输出长度均匀分布在标称值的 $\pm 50\%$范围内；到达持续 600 秒，16K 输入工作负载则持续 300 秒；两种配置在同一作业中先后运行，使用 FP8 和 XQA 解码）。如果请求的首 token 时间不超过 2 秒，且每个输出令牌时间不超过 100 毫秒（MLPerf Llama-3.1-8B 服务器目标），则该请求满足服务级别目标（SLO）；如果至少 99% 的请求满足该目标，则该速率可持续。区间通过重新抽样请求（抽取 1000 次）得出，因此不包含运行间变异；运行情况见表 8。

- **类推理**（1K 输入，8K 输出）。在这些运行中，最高 1.25 请求/秒时，每个请求都满足服务级别目标（SLO）；在使用四个 API 服务器进程、速率为 0.25 和 0.5 时也都满足，在随机种子为 2、速率为 1.25 时同样如此（此时 FP8 的缓存已满，并发生了 190 次抢占）。我们的方法的每个输出令牌平均时间低 23–27%，在 1.25 请求/秒时每秒输出令牌数高 12%，且键值缓存（KV 缓存）的占用量只有一半（FP8 在 1.25 请求/秒时已达到 97%）。从 1.35 请求/秒开始，FP8 的缓存在每次运行中都已满：请求需要等待并被抢占（628–1513 次抢占），且有 6–31% 的请求未达到首令牌时间目标。我们的方法的缓存占用率从未超过 81%；在 1.4 请求/秒时，每个请求都满足服务级别目标（有效吞吐量，即包括生成尾段在内的整个运行期间每秒满足服务级别目标的请求数：1.01 对 0.71 请求/秒，因为 FP8 的运行持续 974 秒，而我们的运行持续 830 秒）。

- **接近饱和时未达标的情况源于 vLLM 的 API 服务器，而非缓存**。使用 vLLM 默认的单个 API 服务器进程时，在速率为 1.35 的三次运行中，我们的方法分别满足了 82%、100% 和 86% 请求的服务级别目标；在速率为 1.5 时，分别满足了 81%、100% 和 100% 请求的服务级别目标。每次未达标都是因为首 token 延迟了 2–6 秒；当时 vLLM 调度器中没有等待处理的请求，也没有请求被抢占，而且客户端按计划发送了所有请求。在速率为 1.35、随机种子为 1 的运行中，API 服务器进程

连续约八分钟占用了一个 CPU 核心 92–97% 的算力（我们的服务器每秒流式生成约 8,200 个输出令牌），而 vLLM 自身的指标也表明延迟发生在该进程中：服务器统计的 814 个请求（包括基准测试的 5 个预热请求）中，有 92 个请求的首 token 时间超过 2.5 秒（按服务器测量）；而调度器的队列时间在全部 814 个请求中累计仅为 0.009 秒，预填充平均耗时 69 毫秒。使用四个 API 服务器进程（`--api-server-count 4`，随机种子为 0 和 1）时，在两种速率和两种随机种子下，我们的方法均满足了所有请求的服务级别目标（速率为 1.35 时，首 token 时间的 p99 至多为 0.16 秒；速率为 1.5 时至多为 0.69 秒）。FP8 并未改变这一情况（速率为 1.35 时为 0.88 和 0.94，速率为 1.5 时为 0.69 和 0.75；而单进程时分别为 0.89–0.94 和 0.69–0.75；抢占次数为 651–1494）：每次运行中，FP8 未达标的情况都发生在缓存已满时。因此，在这些速率下，vLLM 先受单个 API 服务器进程的限制，而后才是我们的缓存；FP8 则先受缓存限制。

- **我们有两次运行未能复现**。两次运行的数据都保留在结果文件中。(i) 种子2在 1.25 和 1.5 请求/秒、使用一个 API 服务器进程时，分别只有 5% 和 3% 的请求满足服务级别目标：第一个请求的每个输出令牌时间就已达到 114 毫秒，而此时几乎没有负载（其他运行中，请求速率为 0.25–0.5 请求/秒时，平均为 7–9 毫秒）；缓存已满，1002 和 1202 个请求被抢占。与此同时，同一节点上的另一块 GPU 上，种子1运行正常。使用种子2以 1.25 请求/秒重新运行（四个 API 服务器进程，另一节点）后，所有请求都满足服务级别目标（每秒输出 7,986 个令牌，缓存峰值为 46%）。(ii) 种子0在 0.25 和 0.5 请求/秒、使用一个 API 服务器进程时，分别有 97% 和 87% 的请求满足服务级别目标；所有未达标的请求都是首令牌延迟达到 2–3 秒，发生时缓存使用率为 5–10%，且分散在整个运行过程中。使用相同设置重新运行，以及改用四个 API 服务器进程重新运行后，所有请求都满足服务级别目标。

- **长上下文**（16K 输入，1K 输出）。两者在 0.5 请求/秒时都能持续运行，在 1.0 时表现相当（0.87 对 0.84，区间重叠），尽管我们的方法使用的缓存只有一半。达到 1.5 时，每秒到达 24K 个提示词 token，接近任一服务器的预填充能力；两者的缓存都被部分预填充的请求填满，最终都无法满足要求；我们的方法表现更差（0.04 对 0.14，输出令牌时间的第99百分位数为 149 对 94 毫秒），这与其预填充速度慢 7% 一致。更大的缓存无法帮助预填充受限流。

- **预填充密集**（16K 输入，512 输出）。在我们测试的两种速率下，FP8 都能让更多请求满足服务级别目标：在 0.75 请求/秒时为 0.98 对 0.94，在 1.25 时为 0.87 对 0.77（1.25 时区间不重叠）；两者的缓存都远未满，也没有发生抢占。两者的输出令牌时间第99百分位数都低于 100 毫秒的目标（1.25 时为 89 对 94 毫秒），因此几乎所有未达标情况都是首令牌延迟，这与我们的方法预填充速度慢 7% 一致；输出较短时，解码没有优势可以抵消这一差距。仅测试我们的方法在 0.5、1.0 和 1.5 请求/秒下的表现：分别为 0.85、0.65、0.16（这些速率下未运行 FP8 测试）。

单 GPU 容量，从 8B 到 72B。对于服务成本而言，关键数字是单个 GPU 能够持续支持的最高请求速率。表 9 给出了类推理流下三个模型的该指标，其中每个区间都细化到了不超过 0.05 请求/秒的宽度。随着模型规模增大，增益也随之提高：Llama-3.1-8B 为 1.23–1.32×，Qwen3-32B 为 1.45–1.80×，Qwen2.5-72B 为 1.67–2.20×；与此同时，每步增益则从 1.12–1.64×（使用 FP8 权重时为 1.15–1.70×）降至 1.04–1.18×：较大模型的一步主要受其权重读取限制（72B 模型每步约需读取 71 GB），并且其权重会占用更多内存，从而留给缓存的空间更少；这与缓存而非带宽决定上限的结论一致。在两个较大的模型上，FP8 缓存在种子0的最高可持续速率下几乎已满（Qwen3-32B 在 0.25 请求/秒时达到 99.9%，Qwen2.5-72B 在 0.125 请求/秒时达到 95%；种子1重复试验的峰值为 68%）；在下一个速率下，请求会被抢占，并且无法达到首令牌时间目标。我们在 Qwen2.5-72B 上测得的最高可持续速率接近其

表 9：单 GPU 容量（`results/serve_goodput_x_11-21.csv`；类推理工作负载：1K 输入、8K 输出，长度在 $\pm 50\%$ 范围内均匀分布；泊松到达，最多 max(150，600 秒 $\times$ 速率) 个请求；H200 SXM；四个 API 服务器进程；使用 FP8 和 XQA 解码）。最高可持续速率是至少 99% 的请求满足服务级别目标的最高速率（首 token 时间 ≤ 2 秒、每个输出令牌时间 ≤ 100 毫秒）： [能够维持的最高速率、无法维持的最低速率），种子0；标记为 † 的边界点使用种子1重复测试，且 Llama-3.1-8B 在 1.25 处的 FP8 数据点也使用种子2进行了验证。键值令牌数：vLLM 在这些服务运行中报告的缓存大小为 `--gpu-memory-utilization 0.9`（`--max-model-len 14336`、`--max-num-seqs 512`）；第 7 节的延迟和吞吐量运行使用其他引擎设置，并报告 Llama-3.1-8B 上为 2.87M 和 1.76M，即相同的 $1.63\times$ 比率。每步：批次大小为 8 及以上时相对于 FP8 的纯解码加速比（`vllm bench forward`；Llama-3.1-8B 和 Qwen3-32B 使用 BF16 权重，Qwen2.5-72B 使用 FP8 权重，其形状最大为 32×8K 和 8×31K；Qwen3-32B 的每步运行早于记录 FlashInfer 已解析的 FP8 内核，详见第 5 节）。三个模型的服务级别目标均相同。Qwen2.5-72B-Instruct 使用 FP8 权重（RedHatAI FP8-dynamic），以适配单个 GPU。

	KV 令牌数		最高可持续速率（请求/秒）			每步
模型	FP8	我们的方法的 FP8		Ours	比率	$B \geq 8$
Llama-3.1-8B	178 万	290 万	[1.25, 1.3)	[1.6, 1.65)†	1.23–1.32×	1.12–1.64×
Qwen3-32B	0.50M	0.82M	[0.25，0.275)†	[0.4, 0.45)†	1.45–1.80×	1.05–1.18×
Qwen2.5-72B（FP8 权重）	0.337M	0.543M	[0.125、0.15)†	[0.25、0.275)†	1.67–2.20×	1.04–1.18×

缓存上限也很重要：在种子0、0.25 请求/秒时，缓存已满，57 个请求被抢占，150 个请求中有 149 个达到服务级别目标（种子 1：峰值的 80%，无请求被抢占）。在 Llama-3.1-8B 上，增益低于 $1.63\times$ 令牌比率：在首次出现故障的速率（1.65）下，我们的方法的缓存占用率为 96%，且没有发生抢占；我们尚未确定此处的限制因素。在 Qwen2.5-72B 上，增益至少达到 $1.61\times$ 令牌比率；我们尚未确定 FP8 的抢占（在 0.15 请求/秒时发生 94 次）对增益的影响有多大。在 72B 模型上，我们的方法也是精度更高的配置：在 vLLM 中，wikitext-2 困惑度（16×8192 个令牌，FP8 权重）方面，FP8 比 BF16 KV 高 0.95%（第 0 层使用 BF16 时为 0.88%），而我们的方法高 0.03%。

8 准确率

在 vLLM 内部，完整方案与 BF16 KV 的差异在我们测试的任何模型或任务上都没有超过运行间噪声，涵盖从 7B 到 32B 的模型以及一个混合专家模型；其中五个模型进行了重复运行并采用配对检验，另外三行仅进行了单次运行。我们仅测量困惑度和任务准确率；KV 缓存量化还可能在困惑度几乎不变的情况下消除安全拒答 [57]，但我们未对此进行测试。离线情况下，在 Llama-3.1-8B、Mistral-7B 和 Qwen3-8B-Base 上，其结果保持在我们预设的范围内（困惑度 $+0.1$，KL 散度 0.02）。如果不采用第 0 层保护，Qwen2 系列推理模型会崩溃，即使使用 FP8 也是如此。

每一步修复的内容。 步骤 (a)–(c) 在不使用汇聚令牌的情况下运行，(d)–(f) 使用四个汇聚令牌；FP8 采用 vLLM 提供的实现（每层均使用，不使用汇聚令牌）。(a)→**(b) 块缩放因子**。 每个张量使用一个缩放因子时，层中的最大值会决定步长；E2M1 的最小非零值为该缩放因子的 0.5 倍，因此当 K 中某个通道或令牌的值比典型值大几十倍时，大多数条目都会被舍入为零。每 16 个值使用一个缩放因子，可将离群值限制在其自身的块内：在 Llama 上 KL 散度下降 $11\times$，在 Qwen3-8B-Base 上下降 $103\times$。（每个张量使用一个缩放因子的 INT4 更差：分别为 1.74 和 4.26。）将 NVFP4 的全局缩放因子固定为 1.0（vLLM 的存储方式）后，在这两个模型上 KL 散度的变化最多为 1.3%。(b)→**(c) 旋转**。

表 10：H200 上各个 KV 缓存选项的表现（使用 Llama-3.1-8B 测量速度、困惑度和 KL 散度）。K 或 V 每行 128 个值所占的字节数，包含缩放因子。速度： `vllm bench forward` 解码步骤，所有行在同一作业中运行；相对于 vLLM 的 FlashInfer FP8 路径的加速比（XQA 解码；数值越高越快）。MATH-500（Qwen2.5-7B-Instruct、R1-Distill-Llama-8B、Phi-4）在 vLLM 内运行，取多次运行的均值；困惑度相对于 vLLM 内 BF16 的增幅（wikitext-2，16×8192；Qwen2.5-7B-Instruct 和 Llama-3.1-8B）；KL 散度为离线结果（FP32，8×2048）。每个张量使用一个缩放因子的方案在 vLLM 或 FlashInfer 中没有解码内核，因此只有离线数值。KL 散度：除我们的方法外均不使用汇聚令牌（我们的方法使用四个）；我们的方法使用窗口 K 均值。vLLM NVFP4 采用 `--kv-cache-dtype nvfp4` vLLM 的默认设置（每层使用，全局缩放因子为 1.0）；$\times 246$：困惑度为 1542，而 6.27（FP8：231）。A $\times$ 将困惑度表示为 BF16 困惑度的倍数，用于增幅超过 100% 的情况；较小的增幅以百分比表示。vLLM NVFP4 的 MATH-500：每项运行三次；FP8：每项运行一次。 *BF16 在缓存能够容纳的三种形状下的结果（8×16K、8×32K、32×16K 个令牌；在 64×16 K 及以上时无法容纳），来自第二个重新运行 FP8 和我们的方法的作业（其中我们的方法为 1.12–1.63×）。

		相较 FP8 的加速比		MATH-500（%）			Δppl		
键值缓存	字节	$B \geq 8$	$B = 1$	Qwen2.5	R1-Llama	Phi-4	Qwen2.5	Llama	KL
BF16	256	0.63–0.86*	0.96–0.97	71.2	67.7	75.3	–	–	–
FP8（FlashInfer XQA）	128	1.00	1.00	1.6	67.0	76.0	×36.8	+0.94%	0.0037
4 位，每个张量使用一个缩放值	64	–	–	–	–	–	–	–	0.297
vLLM NVFP4	72	0.43–0.72	0.87–0.90	0.0	68.4	74.9	×246	+2.62%	0.0282
BF16 中的第 0 层	72	–	–	65.9	–	–	+78%	+2.56%	0.0262
我们的方法（完整配方）	72	1.12–1.64	1.01–1.05	72.0	67.7	74.1	+0.05%	+0.06%	0.0103

表 11：配方中的每个步骤依次单独添加：相对于 BF16 的平均 KL 散度（离线、FP32、wikitext-2、8×2048 令牌，K 均值取每个窗口的均值；(a)–(c) 不含汇点，(d)–(f) 以及 FP8 如正文所述）。这些步骤均不是我们的方法；该表衡量的是在一种设置下每个步骤的贡献。

	Step	Llama-3.1-8B	Qwen2.5-7B-I	Qwen3-8B-Base
(a)	4位 FP4，每个张量一个缩放因子	0.297	10.1	3.00
(b)	NVFP4：每 16 个值使用一个 E4M3 缩放因子	0.0269	5.90	0.0291
(c)	+ 对键进行 Hadamard 旋转 [2]	0.0187	6.95	0.281
(d)	+ 4 BF16 汇聚令牌 [29, 56]	0.0168	7.05	0.249
(e)	+ K 均值去除 [45, 61]	0.0111	0.0167	0.0125
(f)	+ BF16 中的第 0 层	0.0103	0.0147	0.0097
	FP8，每个张量一个缩放（参考值）	0.0037	3.86	0.0040

在一个块内，一个离群通道仍会决定其余 15 个值的缩放因子；Hadamard 旋转会将每个通道的数值分散到全部 128 个值上，使 Llama 的 KL 散度降低 30%。但在 Qwen3-8B-Base 上，效果恰好相反（从 0.029 升至 0.281 ）：该模型的 K 具有较大的逐通道均值，旋转会将其分散到每个块中，进而主导缩放因子（ KV4-RL [39], 附录 D.4 也发现，在未进行均值移除时，K 旋转会加剧 Qwen3 模型的 NVFP4 键值缓存失配）。移除均值后（四个汇聚令牌、第 0 层采用 BF16、每个模型运行一次），旋转仍可降低 KL 散度，但 Llama 仅降低 12%，Qwen2.5-7B-Instruct 降低 14%，Qwen3-8B-Base 降低 8%。(c)→**(d) 汇聚令牌**。 在大多数层中，首批令牌会获得较大比例的注意力质量，因此它们的误差比其他令牌的误差影响更大。移除均值之前，汇聚令牌的帮助很小（Llama -10%）。移除均值后（不含静态均值、汇聚令牌取 0、1、2、4、8、16 的配方），一个汇聚令牌就已达到四个汇聚令牌增益的 98%（Llama）、101%（Qwen3-8B-Base）和 80%（ Qwen2.5-7B-Instruct；两个汇聚令牌可达 97%）；而 8 个或 16 个带来的提升已无法测出：增益来自第一个令牌，四个则留有余量。(d)→**(e) 均值移除**。 逐通道 K 均值由所有令牌共享，并且会在 softmax 中抵消（$q \cdot \mu$ 对每个键都相同），因此在量化前将其减去，就能移除缩放因子原本会耗费在它上面的能量，且不会增加读取时的开销。这一步挽救了 Qwen 模型：Qwen2.5-7B-Instruct 的 KL 散度从 7.05（此时已崩溃，FP8 为 3.86）降至 0.0167，Qwen3-8B-Base 从 0.249 降至 0.0125，两者都接近 Llama 的 0.0111。但这需要汇聚令牌配合：汇聚令牌数为 0 时，它反而会让 Llama 的表现变差（0.0233，高于 0.0187）。(e)→**(f) 第 0 层**。 在 Qwen2 系列模型中，键投影偏置使第 0 层的 K 几乎完全由其均值构成（ R1-Distill-Qwen-7B 上占范数的 98%）；将该层保留为 BF16（占缓存字节数的 9–12%，即 28–36 层中的一层，每令牌占 $3.6\times$ 字节），可进一步降低 Qwen2.5-7B-Instruct 的 KL 散度 12%、Qwen3-8B-Base 的 22% 和 Llama 的 7%（每个模型各运行一次）。单靠这一步还不足以实现 4 位量化：在 Qwen2.5-7B-Instruct 上，采用四个汇聚令牌且仅将第 0 层设为 BF16 的 NVFP4，KL 散度为 0.250， $17\times$ 而配方的结果为 0.0146；同样的保护措施可使 FP8 的 KL 散度从 3.79 降至 0.033（运行一次，各配置均使用四个 BF16 汇聚令牌；其他运行的数据见表 11，配方为 0.0147，无汇聚令牌的 FP8 为 3.86）。在 vLLM 内部，结果排序相同（表 10）：在 Llama-3.1-8B 上，vLLM 的 NVFP4 使困惑度上升 2.62%，将其第 0 层设为 BF16 时上升 2.56%，FP8 上升 0.94%，而配方仅上升 0.06%；在 Qwen2.5-7B-Instruct 上，vLLM 的 NVFP4 即使仅将第 0 层设为 BF16，仍会使困惑度上升 78%，在 MATH-500 上的得分为 65.9%，低于 BF16 的 71.2%（-5.3 个百分点，95% 置信区间 $[-8.4, -2.4]$、 $p < 0.001$，三次运行对比五次），而配方的得分为 72.0%。**vLLM 随附的 NVFP4** 对应于 (b)，其中全局缩放因子固定为 1.0，且不保护任何层：离线测试中，在 Llama 和 Qwen3 上它与 (b) 的差异在 1.3% 以内（各运行一次；表 10 中的 0.0282 来自另一次运行，该次运行测量了 vLLM NVFP4 的两行结果；同一配置下，两次运行分别得到 0.0282 和 0.0271，相差 4%），而在 Qwen2.5-7B-Instruct 上则完全崩溃（4.91）。在 vLLM 内部，它在没有键偏置的两个模型上表现准确：R1-Distill-Llama-8B（$+0.7$ 个百分点，BF16 为基准，95% 置信区间 $[-2.4, +3.9]$、 $p = 0.60$）和 Phi-4（-0.3、 $[-2.6, +1.9]$、 $p = 0.80$；各运行三次）；但在 Qwen2.5-7B-Instruct 上则完全崩溃（三次运行的 MATH-500 得分均为 0.0%，困惑度为 1542，而 BF16 为 6.27），FP8 也同样崩溃（得分为 1.6%）。在 Qwen2 系列之外，它的代价是速度：批次大小为 8 及以上时，解码速度比 FP8 慢 1.4–$2.3\times$ 倍（表 7）。**上下文长度**。 在使用四个汇聚令牌的 Llama 上，配方的 KL 散度（窗口均值，第 0 层采用 BF16）从 2K 个令牌时的 0.0093 升至 16K 个令牌时的 0.0113；相对于 FP8 的比率则从 3.0 降至 2.7（每个数据点使用 32K 个令牌）。

离线困惑度和 KL 散度 （RoPE 后的模拟量化，wikitext-2 [34]， 16×4096令牌，前 4 个令牌使用 BF16；表12）。

旋转可降低 Llama 和 Mistral 上各 4 位格式的误差 1.6–$2.9\times$ （使用汇聚令牌时；不使用时最高可达 $3.3\times$）；效果依次为 NVFP4、INT4、MXFP4。在 Qwen3 上，单独使用旋转会使 NVFP4 的 KL 散度升至 0.210，是移除均值后版本的 18 倍：所有令牌共享的 K 逐通道均值约占 K 能量的一半，并且在某些 Qwen3 层中集中于一个通道；旋转会

表12：相对于 BF16 的离线KL 散度及困惑度增幅（括号内）；在 RoPE 之后进行模拟量化，wikitext-2，16×4096 令牌，前 4 个令牌使用 BF16。此处的 K 均值是对每个窗口分别计算的；取前 512 个令牌的均值会差 8–14%，而在 vLLM 中，该均值通过静态校准获得。

模型	NVFP4 ，K旋转	NVFP4，K均值移除 + 旋转 FP8	
Llama-3.1-8B	KL 散度 0.0165（+0.097）	KL 散度 0.0115（+0.065）	+0.015
Mistral-7B-v0.3	KL 散度 0.0071（+0.030）	KL 散度 0.0060（+0.025）	+0.007
Qwen3-8B-Base	KL 散度 0.210（+1.38）	KL 散度 0.0114（+0.041）	$\leq$ +0.015

表 13：vLLM 内部（vLLM 0.33.0、H200，针对 BF16 键值的完整方案）。对于单次运行，不提供 p：单次运行测试假设每个问题的结果固定不变，而上述运行间变异与此假设相矛盾。区间：两层自助法（先对问题进行抽样，再对每种配置内的运行进行抽样，共 2000 次抽样）；*仅运行一次，因此区间只对问题进行重抽样，不包含运行间变异。

模型	Test	Runs	BF16	Ours	我们的方法 −BF16 [95% CI] (p)
Qwen2.5-7B-Instruct	MATH-500	5	71.2	72.0	+0.8 [−1.6, +3.3](0.49)
DeepSeek-R1-Distill-Qwen-7B	MATH-500	5	70.8	71.2	+0.4 [−2.0, +3.1](0.71)
DeepSeek-R1-Distill-Llama-8B	MATH-500	3	67.7	67.7	+0.07 [−3.1, +3.3] (1.0)
Phi-4	MATH-500	3	75.3	74.1	−1.2 [−3.5, +1.1](0.33)
Qwen3-32B，开启思考模式	MATH-500，首先 200，16K 个词元	3	90.0	90.3	+0.3 [−2.7, +3.2] （0.89）
Qwen3-32B，关闭思考	MATH-500	1	78.6	79.2	+0.6 [−1.6, +2.8]* （–）
Qwen3-30B-A3B（混合专家），关闭思考	MATH-500	1	79.2	79.0	−0.2 [−2.6, +2.2]* (–)
Llama-3-8B-ProLong-512k-Instruct	RULER 风格 [16] 多键 / 多值 / 变量跟踪，32K–120K	1 (450 配对次试验)	–	–	+0.6 分（–）

将其扩散到每个 16 值组中（FlagRot 报告称，在 Qwen3-8B 上进行整头旋转也会造成相同损害）。先移除均值即可解决问题，且不会增加读取时的开销，因为 $q \cdot \mu$ 对每个令牌都相同。移除均值需要汇聚令牌：没有它时，Llama 的表现会差 25%。这些 KL 值是在 BF16 下计算的；使用 FP32 重新计算 Llama-3.1-8B 的 KL 值后，FP8 的 KL 散度降低了 17–18%，4 位格式的 KL 散度降低了 2–6%，且格式的排序保持不变。

在 vLLM 中（vLLM 0.33.0，完整方案与 BF16 KV 对比；表 13）。vLLM 中的贪心解码无法在不同运行之间复现，因为批次组成会改变数值结果：在 R1-Distill- Llama-8B 上，三次 BF16 运行在 MATH-500 [14, 27]，上的得分分别为 68.4%、67.0% 和 67.6%，其中任意两次运行在 500 道问题中约有 47 道结果不一致。因此，推理结果各行采用了多次运行，并打乱提交顺序；对每道问题的平均正确率进行配对置换检验；单次运行中相差 ± 3 分以内并不能说明孰优孰劣。运行使用了两种 H200 变体；同一行中，BF16 和我们的方法在相同变体上运行，但三组 R1-Distill-Llama-8B 运行配对中有一组例外。

除非另有说明，MATH-500 的数据行均使用全部 500 道问题。Qwen2.5-7B 和 R1-Distill-Qwen-7B 已

$G = 7$ 并通过我们的内核运行，其中 q 补零至 8 个头。在 vLLM 中的困惑度（wikitext-2，16×8192 个令牌；OLMo-2 16×4000）相较 BF16 的增幅，我们的方法与 FP8 分别为：Llama-3.1-8B $+0.06\%$ vs$+0.94\%$；Qwen3-32B $+0.07\%$ vs $+0.59\%$；Phi-4 $+0.15\%$ vs $+0.30\%$；OLMo-2-1124-7B $+0.06\%$ vs$+0.34\%$；Qwen3-30B-A3B $+0.07\%$ vs $+0.04\%$；采用 FP8 权重的 Qwen2.5-72B-Instruct $+0.03\%$ vs$+0.95\%$。在配方中，均值移除和汇点很重要：在 Qwen2.5-7B-Instruct 上，我们的方法采用 NVFP4，但不进行 K 均值去除，也不使用汇点（第 0 层采用 BF16），5 次运行的得分为 69.3%，比完整方案低 2.7 个百分点（$p = 0.035$）。

长上下文，离线。在 LongBench [5]（Llama-3-8B-ProLong-512k-Instruct，12 项任务，600 个问题）上，没有任何格式与 BF16 的差距超过 0.6 分；该配方为 $+0.01$。在 32K–64K 的 RULER 类任务上（600 组配对试验），仅使用旋转的 NVFP4 下降 2.4 分（$p = 0.001$）；完整方案下降 0.5 分（$p = 0.55$）。在 120K（多键，100 次试验，Llama-3.1-8B）下，每种 4 位格式都下降 6–12 分，该配方约下降 10 分，而 MXFP4 下降 45–50 分；在 vLLM 中使用 ProLong 时，该配方在 120K 下与 BF16 的差距不超过 1.5 分（每项任务 50 次试验）。

推理与第 0 层。在 DeepSeek-R1-Distill-Qwen-7B 上（离线，MATH-500，前 200 道题，生成 4,096 个令牌），BF16 的得分为 73.5%，FP8 KV 为 0.5%，而所有 4 位格式的得分都在 0–22.5% 之间。仅将第 0 层保留为 BF16，即可使 FP8 的得分恢复至 73.0%，NVFP4 配方的得分也恢复至 73.0%。在实际 vLLM 中（前 200 道题，单次运行），FP8 KV 的得分分别为 0.5%（R1-Distill-Qwen-7B）、1.5%（Qwen2.5-7B-Instruct）和 0.0%（R1-Distill-Qwen-1.5B）；跳过第 0 层后，得分分别恢复至 71.5%、72.0% 和 61.0%。R1-Distill-Qwen-14B 则不受影响（BF16 为 70.0%，FP8 为 73.0%）。同一 R1 蒸馏模型运行于没有键偏置的 Llama-3.1 基础模型时，并未出现性能崩溃：BF16 得分为 68.4%，FP8 为 67.0%（-1.4），将第 0 层设为 BF16 后的 FP8 得分为 68.8%（500 道题，每种设置各运行一次；差异处于运行间噪声范围内）。在 R1-Distill-Qwen-7B 上，逐通道均值占第 0 层 K 范数的 98%（其他层为 40–83%），因此 FP8 的步长由均值决定，依赖于令牌的部分则丢失；第 0 层的误差很可能因此传递到后续每一层。离线测试中，困惑度也体现了这一问题（Qwen2.5-7B：FP8 KV 下从 6.31 升至 78.2；将第 0 层设为 BF16 时为 6.59）。这一性能崩溃此前已有报告：一项推理模型服务的实证研究 [26]（使用 vLLM 0.8.1 和 SGLang，在 RTX 4090、A6000 和 A100 服务器上运行）发现，R1-Distill-Qwen-7B 使用 FP8-E4M3 KV 时在 MATH-500 上的得分为 4.33%，使用 E5M2 时为 0.33%（令牌预算为 4,096），但未说明原因，并提醒 KV 量化可能会损害小型推理模型的性能。保护第一层也是已有做法（Easy-Kuant [1]会在其选定层中将第 0 层保留为 BF16，vLLM 拉取请求 #63081 [48]则建议跳过第 0 层）。这一机制也并非新发现：Liu 等人 [30]（第 3.3 节、附录 C.2）将 R1-Distill-Qwen-1.5B 和 7B 的 K 离群值归因于从其 Qwen 基础模型继承的大键投影偏置，并展示了 RoPE 前后第 0 层的 K；其研究使用研究型量化器处理 3 位和 4 位 KV，而非服务引擎的 KV 缓存内核。rtp-llm 拉取请求 #1583 [64] 则针对逐令牌 FP8 KV，在 RoPE 之后减去一个由键偏置确定的固定中心。Easy-Kuant [1]（第 2 节）也报告了 vLLM 中的性能崩溃，并将第 0 层保留在一至四个 BF16 层之中。vLLM 拉取请求 #63081 [48], 指出，偏置在缓慢旋转的维度上经 RoPE 后仍然存在，因此会影响每个键，并在启动时对此发出警告。我们的新增内容包括：（i）指出第 0 层中每个令牌都很大的 RoPE 后键通道（Easy-Kuant 的逐层统计量能够检测到）实际上是经 RoPE 变换后的偏置：对不同位置的 RoPE(b)取平均，可解释第 0 层均值能量的 99.7–100%，且仅根据权重即可计算（见下一段）；（ii）Llama 基础模型对照实验；（iii）一种修复方法：仅将第 0 层保留为 BF16，其余部分采用均值移除，不损失容量（我们的缓存可容纳 $1.63\times$ FP8 令牌，而每层都使用 4 位时可容纳 $1.78\times$ ）。

第 0 层的均值从何而来。在 Qwen2 系列模型中，第 0 层的 K 均值几乎完全来自经 RoPE 变换后的键投影偏置 b：对不同位置的 RoPE(b)取平均后，Qwen2.5-7B-Instruct、R1-Distill-Qwen-7B 和 Qwen2.5-72B-Instruct 的第 0 层均值能量会减少 99.7–100%；这些数值仅根据权重 `rope_theta` 计算，无需校准。在 RoPE 之前逐位置减去 b，并在评分时将其加回，即可将其精确移除；这种方法并非我们提出（vLLM 拉取请求 #59674 [54]针对 INT4 KV，scratchy 拉取请求 #134 [46]；Liu 等人 [30],在附录 C.2.2 中于添加偏置之前量化 K）。我们仅报告测量结果。离线测试（在 FP32 中模拟量化，以 BF16 为基准计算 KL 散度，不使用我们的内核或 vLLM，也不测任务准确率或速度）显示，逐位置减法应用于每一层时，五个 Qwen2 系列模型在 2,048 个令牌下的平均 KL 散度比我们将第 0 层保留为 BF16 的方案低 8–40%；而仅对第 0 层应用该方法，效果不如将第 0 层保留为 BF16。静态校准均值也会随上下文长度增加而退化：在 R1-Distill-Qwen-7B 的 8,192 个令牌测试中，静态中心方案的 KL 散度为逐位置方案的 2.6× 。在 vLLM 中，我们只测试了更简单的固定中心：每层使用一个向量，在 RoPE 之后从每个键中减去；该向量可以是 b本身（rtp-llm #1583 使用的中心），也可以是对不同位置的 RoPE(b)取平均所得；所有层均使用 FP8 KV（H200 NVL，MATH-500 前 200 道题，4,096 个令牌，单次运行；Wikitext-2 困惑度测试使用 16 个长度为 8,192 个令牌的序列）。这种方法可使 Qwen2.5-7B-Instruct 的性能完全恢复（得分为 73.0% 和 71.5%，BF16 KV 为 72.5%；困惑度为 6.31 和 6.30，BF16 KV 为 6.27，而将第 0 层设为 BF16 时为 7.33），但只能部分恢复 R1 蒸馏模型的性能：R1-Distill-Qwen-7B 的得分为 69.0% 和 57.0%（BF16 KV 为 74.0%，将第 0 层设为 BF16 时为 68.5%；困惑度为 32.1 和 21.5，BF16 KV 为 19.3）；R1-Distill-Qwen-1.5B 的得分为 48.0% 和 50.0%（BF16 KV 为 58.0% 和 61.5%；困惑度为 52.7 和 31.4，BF16 KV 为 26.0%）。在这三种模型的单次运行中，三种方案都未能在所有模型上达到接近 BF16 KV 的效果：固定中心在 R1 蒸馏模型上的表现不够理想；将第 0 层设为 BF16 会使 Qwen2.5-7B-Instruct 的困惑度升至 7.33，并使 R1-Distill-Qwen-7B 的得分达到 68.5%（上文报告的那次运行中为 71.5%，这体现了运行间的波动）。主要方案将第 0 层保留为 BF16，因为结合均值移除后，它在表 13 所列模型上的准确率与 BF16 KV 相比没有显著差异；而仅使用固定中心、且每一层都采用 FP8 的方案，在 R1 蒸馏模型上的表现不够理想。

9 局限性与后续步骤

- **范围**。头维度为 128，每个键值头对应 1–8 个查询头。该内核支持 $G = 4$ 或 8；其他 $G \leq 8$ 通过对 q 进行零填充来运行（例如，Qwen2.5-7B 和 R1-Distill-Qwen-7B 从 7 填充到 8，这也用于它们的准确率数据），因此浪费了 1/8 的张量核心计算。在 vLLM 中（单个任务，H200 SXM，使用 XQA 解码的 FP8），Qwen2.5-7B-Instruct 的解码速度在批次为 1 时达到 FP8 速度的 0.97× ，批次为 8 及以上时达到 1.06–1.28× ；Phi-4（$G = 4$）则分别为 0.99–1.00× 和 1.01–1.28×。在内核层面， $G = 7$ 耗时与 $G = 8$ 相同（误差在 0.7% 以内），并且在批次为 1 时与 XQA 持平，因此填充并非 Qwen2.5-7B-Instruct 在批次为 1 时存在性能差距的原因；我们尚未找到该差距的原因。INT4 目前只有一种采用 $G = 4$的稠密内核。未涵盖的内容：MLA 模型（DeepSeek-V3/R1）、头维度为 256（Qwen3.x、MiniMax，vLLM 的 UltraQuant 面向这些模型）或 64 的模型，以及大多数滑动窗口模型。

- **模型**。在 vLLM 内对表 13 中的七个模型进行任务准确率测试；在 vLLM 内对 Llama-3.1-8B、Qwen3-32B、Phi-4、OLMo-2-1124-7B、Qwen3-30B-A3B 和 Qwen2.5-72B-Instruct（FP8 权重）进行困惑度测试；在 Llama-3.1-8B、Mistral-7B 和 Qwen3-8B-Base 上进行离线测试。张量并行：在带有 TP $= 2$ 的 Qwen2.5-72B-Instruct 上，该配方在前缀缓存、跨请求前缀命中和分块预填充场景下均能给出正确答案（仅验证正确性；未测量 TP $= 2$ 下的速度和准确率）。

- **负载下服务**。有效吞吐量测试（第 7 节）使用一种 GPU 类型和随机提示。

表 8 的速率扫描涵盖 Llama-3.1-8B：在 0.75 和 1.0 请求/秒时，每个速率运行一次；在 0.25 和 0.5 时，FP8 运行两次，我们的方法运行三次；在 1.25 时运行两次（加上 seed-2 重跑后共三次）；在 1.35 和 1.5 时，FP8 运行四次，我们的方法运行五次，其中包括使用四个 API 服务器进程的测试。表 9 的容量范围涵盖三个模型和一个工作负载，采用种子0；仅在标记的点进行了种子1重复试验。接近饱和时，vLLM 默认的单个 API 服务器进程会成为我们的方法的瓶颈（最多有 19% 的请求未达标）；使用四个进程时，在 1.35 和 1.5 下用两个随机种子测试，我们的方法没有请求未达标。在预填充受限流上，我们的方法不如 FP8，因为其预填充速度较慢（输入为 16K、输出为 512 个令牌时，速率为 0.75 和 1.25 请求/秒；输入为 16K、输出为 1K 时，速率为 1.5 请求/秒）。未对具有共享前缀的工作负载、真实轨迹、较大模型上的其他工作负载、表 9 所列三个模型之外的模型、张量并行和解耦预填充进行比较。

- **其他模型上的速度**（纯解码，`vllm bench forward`，H200 SXM，相对于 FlashInfer FP8）：Qwen3-32B（每个 KV 头对应 8 个查询头，共 64 层）在 $B = 1$ 时为 1.00–1.02×，在 $B \geq 8$ 时为 1.05–1.18×；Qwen3-30B-A3B（混合专家，4 个 KV 头）在 $B = 1$ 时为 0.94–0.95×，在 $B \geq 8$ 时为 1.11–1.32×；使用 FP8 权重的 Qwen2.5-72B-Instruct 在 $B = 1$ 时为 1.00–1.02×，在 $B \geq 8$ 时为 1.04–1.18×（上下文最长 31K，批次最大为 32）。相比之下，在 Llama-3.1-8B 上的增益更小，因为注意力仅占单步计算的一小部分（上下文较短、权重较大）。

- **小批量**。在 $B = 1$ 64K 上下文长度下，该内核的带宽利用率仅约为 38%（16K 时为 18%），而整步结果与 FP8 持平：在 16K 下进行了两次测量，每次都在同一节点上运行两种配置（一台节点测量每个输出词元的端到端时间，另一台节点进行逐步性能剖析），结果分别为 0.96× 和 1.05×；FlashInfer 自身的耗时在两次测量间波动了 6%（5.08–5.40 毫秒）。逐内核性能分析揭示了该内核的领先优势为何消失：我们的解码内核每个词元耗时 382 µs，而 FlashInfer 为 702 µs；但与 BF16 汇聚项相关的分片合并（269 µs，即每层约 8.7 µs）、缓存写入和 q 旋转，使 31 个 4 位层的总耗时达到 848 µs（第 0 层采用 BF16，另需约 30 µs），而 FlashInfer 为 847 µs。接下来的内核优化步骤，是将汇聚合并并入解码内核的分片归约，并将 q 旋转并入其序言。

- **写入路径**。优化后，缓存写入内核的速度比之前的版本快 3.1× 倍；在预填充阶段，其速度与 FP8 相当（对于相同任务中的 8 个 32,000 令牌提示词，耗时分别为 98.6 毫秒和 97.8 毫秒）；在解码阶段，其速度仍然较慢（ 1×16K 下每个令牌分别为 161.8 和 144.5 µs）。剩余的预填充开销列于第 7 节。

- **FP8 基线**。我们将其与 FlashInfer 的 FA2 和 XQA FP8 内核、vLLM 默认的 FlashAttention-3 FP8 路径，以及 FlashAttention-4 原生 FP8（表 5）进行了比较，而不是与 TensorRT-LLM 本身比较。在 vLLM 中，我们还在包含这两个拉取请求的开发版本上运行了我们的插件；该版本中，普通 FP8 KV 的每一层都会选择 FlashAttention-4（Llama-3.1-8B，`vllm bench forward`，所有配置在 H200 SXM 上通过一次作业运行）：每个解码步骤中，我们的配方在批次大小为 8 及以上时比 FlashAttention-4 的 FP8 快 1.04–1.24×，在批次大小为 1 时则持平（0.99–1.00×）；而在同一作业中，批次大小为 8 及以上时，我们比 XQA 快 1.12–1.59×；FlashAttention-4 的步进时间为 XQA 的 0.78–1.00 倍。该版本在注意力计算之外的每步耗时比 vLLM 0.33 慢 0.65–1.4 毫秒，因此我们只在该版本内部进行比较；文中报告的其他每步增益均以 vLLM 0.33 的 FP8 路径为基准。XQA 在32×16K 和 128×16K 时达到 H200 峰值带宽的 88–90%；在这些情况下，我们的内核级增益（1.37–1.47×）低于 1.78× 的字节比率。提高我们自己的大批次带宽占比，是内核性能余量的主要所在。32×16K 时的 NVBit 计数揭示了 XQA 性能优势的来源：它每个值执行 3.9 条指令，与我们的内核（3.6 条）大致相当，但 4 位缓存每字节可容纳 1.78× 倍的值；此外，XQA 的指令中有 30% 是屏障等待及其自旋循环。一种原型

借用了 XQA 的 FP8 张量核心 K 路径（TensorRT-LLM 的 XQA 将来自共享内存的 FP8 K 直接送入 FP8 矩阵乘法；FServe [4] 会在寄存器中将 INT4 KV 反量化为 FP8 供其使用；TensorBridge [65] 则从 NVFP4 权重构建 FP8 操作数；我们的方法在写入时将 E2M1 位放入 E4M3，每 32 个维度保留一个 E4M3 缩放因子，使缩放在乘法之后应用，并将 q 舍入为 E4M3 ）每个值执行的指令减少了 17% ；当批次大小为 8 及以上且每个序列包含至少 16K 个令牌时，它比我们的内核快 1.12–1.14× （ 32×4K 时为 1.06× ），而在批次大小为 1–4 时快 1.04–1.06× （与 XQA 在 H200 SXM 上通过一次作业运行，采用表 4 的 11 种形状：比 XQA 快 1.29–2.22× ），峰值带宽利用率最高达 82%。其 K 格式使 Llama-3.1-8B、Qwen2.5-7B-Instruct 和 Qwen3-8B-Base 的离线 KL 散度增加 14–17%。撰写本文时，它还是一个独立内核：缓存写入路径、分页以及每个 KV 头 8 个查询头均尚未实现；本文中的其他数据均未使用它，而将其集成是下一次修订的主要工作。

- **统计**。 表 13 中的 95% 置信区间跨度约为 $\pm$2.5–3 个百分点：在 200–500 个问题、每种配置运行三到五次的情况下，这些比较能够检测到的最小差异（2.8 个自助法标准误， $\alpha = 0.05$, 检验效能为 0.8）为 3.3–4.6 个百分点。它们可以排除大于约 3.5 个百分点的准确率损失，但无法排除更小的损失。区间宽度主要来自问题抽样，而非运行次数（单次运行的区间几乎同样宽），因此要缩小区间，需要增加问题数量，而不是增加运行次数。

- **Hopper 上的其他 4 位路径**。 在 Hopper 上，我们并非唯一一种速度不慢于 8 位的 4 位 KV 解码方案：FOSCAR 的 INT4（SGLang [62]）在 H200 上的内核层面比 FlashInfer 的 FA2 FP8 内核快 1.39–2.26×；FServe 的 INT4 解码在其测量中与 H100 上 FlashAttention-3 的 FP8 解码性能相当或更快 [4],；LMDeploy 的 TurboMind INT4 KV 每个解码步骤比其自身的 INT8 KV 快 1.04–1.32× ，比其 BF16 KV 快 1.06–1.70× （Llama-3.1-8B，H200， $B = 1$–32，16K–32K；BF16 在 32×32K 时无法容纳）。FlagRot 的 INT4 缓存 [45], 在 H200 上通过 vLLM 0.30 为智能体流量提供服务，其吞吐量是使用 BF16 sink 和近期令牌时 FlashInfer FP8 的 0.92–0.94× ，不使用它们时则为 1.00–1.09× （根据其表格由我们计算）。KV4-RL [39] 在自有引擎中使用带汇聚令牌的 INT4 KV，并将近期令牌以 INT8 存储，用于 H200 上的强化学习 rollout；其报告称，解码前向传播比 FlashInfer BF16 快 1.44–1.97× ，且在 400 次 GRPO 迭代中，rollout 相较 BF16 加速 1.80× 。TurboMind 不支持 FP8 KV，而且这些整步数据来自不同引擎，因此无法与我们的 vLLM 数据进行比较。在 vLLM 可用的 4 位路径中，我们测量的性能都慢于 FlashInfer FP8：vLLM 自带的 UltraQuant 后端（H200 SXM、单个作业时，每步慢 1.8–4.4× ），以及内核层面的 FlashInfer NVFP4 和 PR #53060（H200）。

- **测量**。 每项速度比较都在同一 GPU 类型上进行两种配置的对比（除非特别标注，否则为 H200 SXM）。内核表和纯解码表分别来自一次作业；端到端延迟数据对来自同一节点上连续运行的两次作业，而解码主导吞吐量则来自不同作业。推理准确率需要进行重复且随机打乱顺序的运行（第 8 节）。

下一步：将服务测量扩展到更多模型以及共享前缀和解耦配置，将汇聚令牌汇合合并纳入解码内核，并为 vLLM（UltraQuant）和 FlashInfer（NVFP4）提出 Hopper 路径。

代码可用性 y. 代码见 `https://github.com/thomaszhang2661/kv4-hopper`。

致谢

作者感谢 Jiaji Huang 对本文提出的意见，并感谢 Ilmi Yoon 女士给予的支持。本研究部分工作使用了 Explorer Cluster 完成，该集群由 Northeastern University 的研究计算团队提供支持。

人工智能披露声明。 本文的研究问题、设计决策和所有主张均由作者提出。作者使用了生成式人工智能工具（Claude、Claude Code）编写代码、运行和协调实验，以及起草文本。文中报告的所有结果均来自有日志记录且经作者核查的运行，作者对本文内容承担全部责任。

参考值

[1] 匿名作者。Easy-Kuant：通过选择性层保护诊断并缓解 KV 缓存量化崩溃。提交至 ICLR 2027，OpenReview，2026年。网址 https://openreview.net/forum?id=byJQ7BNyPd。访问日期：2026年11月22日。

[2] Saleh Ashkboos、Amirkeivan Mohtashami、Maximilian L. Croci、Bo Li、Pashmina Cameron、Martin Jaggi、Dan Alistarh、Torsten Hoefler 和 James Hensman。QuaRot：旋转后的大语言模型中的无异常值4位推理。*arXiv* 预印本 *arXiv:2404.00456*，2024年。网址 https://arxiv.org/abs/2404.00456。NeurIPS 2024。

[3] Daehyeon Baek、Jieun Choi、Jimyoung Son、Kyungmin Bin、Seungbeom Choi、Kihyo Moon、Minsung Jang 和 Hyojung Lee。FireQ：用于大语言模型推理加速的快速 INT4-FP8内核与 RoPE感知量化。*arXiv* 预印本 *arXiv:2505.20839*，2025年。

[4] Daehyeon Baek、Jieun Choi、Jimyoung Son、Jae-Young Choi、Kyungmin Bin、Seungbeom Choi、Kihyo Moon、Minsung Jang 和 Hyojung Lee。FServe：面向高效大语言模型服务的原生FP8的W4A8KV4量化与注意力内核优化。收录于计算语言学协会 *EMNLP 2026* 论文集，第 33132–33153 页，2026年。URL https://aclanthology.org/2026.findings-emnlp.1556。

[5] Yushi Bai、Xin Lv、Jiajie Zhang、Hongchang Lyu、Jiankai Tang、Zhidian Huang、Zhengxiao Du、Xiao Liu、Aohan Zeng、Lei Hou、Yuxiao Dong、Jie Tang 和 Juanzi Li。LongBench：用于长上下文理解的双语多任务基准。*arXiv* 预印本 *arXiv:2308.14508*，2023年。URL https://arxiv.org/abs/2308.14508。ACL 2024。

[6] bhoomit。 [修复错误][TurboQuant] 当请求存在缓存的 KV 时，跳过预填充快速路径。vllm-project/vllm 拉取请求 #67789，2026年。URL https://github.com/vllm-project/vllm/pull/67789。于 2026年11月15日创建；截至 2026年11月20日仍处于开放状态。访问日期：2026年11月20日。

[7] Inesh Chakrabarti、David Limpus、Aditi Ghai Rana、Bowen Bao、Spandan Tiwari、Thiago Crepaldi 和 Ashish Sirasao。UltraQuant：面向上下文密集型智能体的 4 位键值缓存。*arXiv* 预印本 *arXiv:2606.20474*，2026年。URL https://arxiv.org/abs/2606.20474。EMNLP 2026 工业界赛道。

[8] Tri Dao。《FlashAttention-2：通过更好的并行性和工作划分实现更快的注意力》。*arXiv* 预印本 *arXiv:2307.08691*，2023年。URL https://arxiv.org/abs/2307.08691。ICLR 2024。

[9] Tri Dao、Daniel Haziza、Francisco Massa 和 Grigory Sizov。《面向长上下文推理的 Flash-Decoding》。Stanford CRFM 博客，2023年。URL https://crfm.stanford.edu/2023/10/12/flashdecoding.html。[10] Dayou Du、Shijie Cao、Jianyi Cheng、Luo Mai、Ting Cao 和 Mao Yang。《BitDecoding：释放张量核心的潜能，为长上下文大语言模型提供低比特 KV 缓存》。arXiv 预印本 arXiv:2503.18773，2025年。URL https://arxiv.org/abs/2503.18773。HPCA 2026。[11]Dayou Du、Woon Yee Ng、Guangming Sheng、Nick Lau、Jianyi Cheng 和 Luo Mai。《BitDecoding-FP4：通过算法-系统协同设计实现原生FP4 KV 缓存解码》。arXiv 预印本 arXiv:2611.09825，2026年。URL https://arxiv.org/abs/2611.09825。[12] Elias Frantar、Roberto L. Castro、Jiale Chen、Torsten Hoefler 和 Dan Alistarh。《MARLIN：面向大语言模型的混合精度自回归并行推理》。arXiv 预印本 arXiv:2408.11743，2024年。URL https://arxiv.org/abs/2408.11743。PPoPP 2025。[13] GreatGIF。 [修复][TurboQuant]： 在快速路径中添加续接保护机制，防止前缀缓存导致前缀 K/V 丢失。vllm-project/vllm 拉取请求 #46461，2026年。URL https://github.com/vllm-project/vllm/pull/46461。于2026年6月23日提交；截至2026年11月20日仍处于开放状态。访问日期：2026年11月20日。[14] Dan Hendrycks、Collin Burns、Saurav Kadavath、Akul Arora、Steven Basart、Eric Tang、Dawn Song 和 Jacob Steinhardt。《使用 MATH 数据集衡量数学问题求解能力》。arXiv 预印本 arXiv:2103.03874，2021年。URL https://arxiv.org/abs/2103.03874。NeurIPS 2021。[15] Coleman Hooper、Sehoon Kim、Hiva Mohammadzadeh、Michael W. Mahoney、Yakun Sophia Shao、Kurt Keutzer 和 Amir Gholami。《 KVQuant：通过 KV 缓存量化实现 1000 万上下文长度的大语言模型推理》。arXiv 预印本 arXiv:2401.18079，2024年。URLhttps://arxiv.org/abs/2401.18079。NeurIPS 2024。[16] Cheng-Ping Hsieh、Simeng Sun、Samuel Kriman、Shantanu Acharya、Dima Rekesh、Fei Jia、Yang Zhang 和 Boris Ginsburg。《RULER：长上下文语言模型的真实上下文规模究竟有多大？》。arXiv 预印本 arXiv:2404.06654，2024年。URL https://arxiv.org/abs/2404.06654。COLM 2024。[17] Huanqi Hu、Bowen Xiao、Shixuan Sun、Jianian Yin、Zhexi Zhang、Xiang Luo、Chengquan Jiang、Weiqi Xu、Xiaoying Jia、Xin Liu 和 Minyi Guo。《LiquidGEMM：用于高性能大语言模型服务的高效硬件 W4A8 GEMM 内核》。arXiv 预印本 arXiv:2509.01229，2025年。URL https://arxiv.org/abs/2509.01229。SC 2025。[18] 书生·浦语。LMDeploy。GitHub 仓库 InternLM/lmdeploy，“LMDeploy 是一款用于压缩、部署和服务大语言模型的工具包”，2023年。URL https://github.com/InternLM/lmdeploy。仓库创建于2023年6月15日。访问日期：2026年11月16日。[19] Jeffwan。 [性能]：在 H100 上，默认的 FP8 KV 缓存路径（FA3、FP8 查询）比 FlashInfer 慢 ~11% ，且每次运行的输出都不相同。vllm-project/vllm issue #66590，2026年。URL https://github.com/vllm-project/vllm/issues/66590。于2026年11月9日提交；截至2026年11月20日仍处于开放状态。访问日期：2026年11月20日。[20] jhaotingc。 [Cute、Fwd、Sm90]： 面向 Hopper 的原生 FP8（E4M3 Q/K/V）前向实现。vllm-project/flash-attention 拉取请求 #206，2026年。URL https://github.com/vllm-project/flash-attention/pull/206。于2026年10月22日提交；截至2026年11月22日仍处于开放状态。访问日期：2026年11月22日。

[21] jhaotingc。 [注意力] 在 FA4 原生 FP8 上运行 SM90 FP8-KV 注意力。vllm-project/vllm 拉取请求 #63139，2026年。URL https://github.com/vllm-project/vllm/pull/63139。于2026年10月22日开启；截至 2026年11月22日仍未关闭。访问日期：2026年11月22日。[22] Jinda Jia、Jisen Li、Zhongzhu Zhou、Jung Hwan Heo、Jue Wang、Tri Dao、Shuaiwen Leon Song、Ben Athiwaratkun、Chenfeng Xu、Tianyi Zhang 和 Xiaoxia Wu。SA W-INT4：面向真实世界大语言模型服务的系统感知型 4 位 KV 缓存量化。*arXiv* 预印本 *arXiv:2604.19157*，2026年。URL https://arxiv.org/abs/2604.19157。[23] Faizan Ah mad Khattak 和 Mantas Mikaitis。NVIDIA 张量核心的精确模型。*ACM*《体系结构与代码优化汇刊》，23(3):1–25，2026年。doi: 10.1145/3830409。arXiv:2512.07004。[24] Young Jin Kim、Rawn Henry、Raffy Fahim 和 Hany Hassan Awadalla。谁说大象不能奔跑：将大规模混合专家模型引入云规模生产环境。*arXiv* 预印本 *arXiv:2211.10017*，2022年。URL https://arxiv.org/abs/2211.10017。SustaiNLP 2022研讨会。[25] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Sheng、Lianmin Zheng、Cody Hao Yu、Joseph E. Gonzalez、Hao Zhang 和 Ion Stoica。使用 PagedAttention 高效管理大语言模型服务的内存。*arXiv* 预印本 *arXiv:2309.06180*，2023年。URLhttps://arxiv.org/abs/2309.06180。SOSP 2023。[26] Qi Li、Junpan Wu、Xiang Liu、Yuxin Wang、Zeyu Li、Zhenheng Tang、Yuhan Chen、Shaohuai Shi 和 Xiaowen Chu。推理语言模型推理服务揭秘：一项实证研究。*arXiv* 预印本 *arXiv:2510.18672*，2025年。URL https://arxiv.org/abs/2510.18672。ICLR 2026。[27] Hunter Lightman、Vineet Kosaraju、Yura Burda、Harri Edwards、Bowen Baker、Teddy Lee、Jan Leike、John Schulman、Ilya Sutskever 和 Karl Cobbe。让我们逐步验证。*arXiv* 预印本 *arXiv:2305.20050*，2023年。URL https://arxiv.org/abs/2305.20050。ICLR 2024。[28] Yujun Lin、Haotian Tang、Shang Yang、Zhekai Zhang、Guangxuan Xiao、Chuang Gan 和 Song Han。QServe：面向高效大语言模型服务的 W4A8KV4 量化与系统协同设计。载于《机器学习与系统会议论文集》（*MLSys*），2025年。URL https://arxiv.org/abs/2405.04532。arXiv:2405.04532。[29] Ruikang Liu、Haoli Bai、Haokun Lin、Yuening Li、Han Gao、Zhengzhuo Xu、Lu Hou、Jun Yao 和 Chun Yuan。IntactKV：通过保持枢轴令牌不变来改进大语言模型量化。*arXiv* 预印本 *arXiv:2403.01241*，2024年。URL https://arxiv.org/abs/2403.01241。ACL 2024论文集。[30] Ruikang Liu、Yuxuan Sun、Manyi Zhang、Haoli Bai、Xianzhi Yu、Tiezheng Yu、Chun Yuan 和 Lu Hou。量化会损害推理能力吗？量化推理模型的实证研究。*arXiv* 预印本 *arXiv:2504.04823*，2025年。URL https://arxiv.org/abs/2504.04823。COLM 2025。[31] Zirui Liu、Jiayi Yuan、Hongye Jin、Shaochen Zhong、Zhaozhuo Xu、Vladimir Braverman、Beidi Chen 和 Xia Hu。KIVI：一种无需调优的 KV 缓存非对称2位量化方法。*arXiv* 预印本 *arXiv:2402.02750*，2024年。URL https://arxiv.org/abs/2402.02750。ICML 2024。

[32] LucasWilkinson。 [RFC]：自带 GPU（BYOG），为拉取请求提供可复现的评估和基准测试。vllm-project/vllm issue #61698，2026年。URL https://github.com/vllm-project/vllm/issues/61698。于 2026年10月14日开启；截至 2026年11月21日仍未关闭。访问日期：2026年11月21日。[33] LyrisZhong。 [内核] SM90 CUTLASS FP8 通用矩阵乘法：添加对交换 AB + 内核调优的支持。vLLM 拉取请求 #20396，2025年7月28日合并，2025年。URL https://github.com/vllm-project/vllm/pull/20396。[34] Stephen Merity、Caiming Xiong、James Bradbury 和 Richard Socher。指针哨兵混合模型。arXiv 预印本 arXiv:1609.07843，2016年。URL https://arxiv.org/abs/1609.07843。[35] Chao Ni、Qi Zhang、Xiao Wang、Zongfei Jing、Mingyang Hao、Binghan Chen、Kefeng Duan、Yu-Jung Chen、Mengdi Wang、Tijmen Blankevoort 和 Jun Yang。NVFP4-MLA：面向多头潜在注意力的统一 FP4 KV 缓存存储与注意力计算。arXiv 预印本 arXiv:2611.18972，2026年。[36] Ningke-Li。 [错误]： kv_cache_dtype="fp8_e5m2" 在默认缩放设置下，Qwen-VL 模型（Qwen2-VL、Qwen2.5-VL）的输出会被静默破坏。vllm-project/vllm issue #41343，2026年。URLhttps://github.com/vllm-project/vllm/issues/41343。于 2026年4月30日开启；截至 2026年11月20日仍未关闭。访问日期：2026年11月20日。[37] 英伟达。使用 NVFP4 预训练大语言模型。arXiv 预印本 arXiv:2509.25149，2025年。URL https://arxiv.org/abs/2509.25149。[38] 英伟达。TensorRT-LLM。GitHub 代码库， https://github.com/NVIDIA/TensorRT-LLM；cpp/kernels/xqa中的 XQA 解码内核，2026年。访问日期：2026年11月22日。[39] Sihwan Park、Sung-Yub Kim、Doohyuk Jang 和 Eunho Yang。KV4-RL：使用 INT4 KV 缓存量化，为 RLVR 解锁更快的异步展开。arXiv 预印本 arXiv:2610.14448，2026年。URL https://arxiv.org/abs/2610.14448。[40] Sarunya Pumma、Jongsoo Park、Jianyu Huang、Amy Yang、Jaewon Lee、Daniel Haziza、Grigory Sizov、Jeremy Reizenstein、Jeff Johnson 和 Ying Zhang。面向大语言模型推理的 INT4 解码 GQA CUDA 优化。PyTorch 博客，2024年。URL https://pytorch.org/blog/int4-decoding/。访问日期：2026年11月17日。[41] Bita Darvish Rouhani、Ritchie Zhao、Ankit More、Mathew Hall、Alireza Khodamoradi 等。面向深度学习的微缩放数据格式。arXiv 预印本 arXiv:2310.10537，2023年。URLhttps://arxiv.org/abs/2310.10537。[42] sha7doww。 [错误]：FP8 KV 缓存与 kv_cache_dtype_skip_layers 结合时，FA3 会崩溃：“sched- uler_metadata must have shape (metadata_size)”。vllm-project/vllm issue #65821，2026年。URL https://github.com/vllm-project/vllm/issues/65821。于 2026年11月5日开启；截至 2026年11月20日仍未关闭。访问日期：2026年11月20日。[43] sha7doww。 [错误修复]：修复 FP8 KV 缓存下 kv_cache_dtype_skip_layers 导致的 FA3 崩溃。vllm-project/vllm 拉取请求 #65891，2026年。URL https://github.com/vllm-project/vllm/pull/65891。于 2026年11月5日开启；截至 2026年11月20日仍未关闭。访问日期：2026年11月20日。[44] Jay Shah、Ganesh Bikshandi、Ying Zhang、Vijay Thakkar、Pradeep Ramani 和 Tri Dao。FlashAttention-3：借助异步性与低精度实现快速、准确的注意力。arXiv 预印本 arXiv:2407.08608，2024年。URL https://arxiv.org/abs/2407.08608。NeurIPS 2024。

[45] Kunming Shao、Xijie Huang、Jierun Chen、Jiaqi Shao、Yanli Wang、Jiangnan Yu、Qiwei Li、Wei Zhang、Kwang-Ting Cheng 和 Chi Ying Tsui。FlagRot：低比特 KV 缓存的选择性旋转。arXiv 预印本 arXiv:2611.03992，2026年。网址 https://arxiv.org/abs/2611.03992。[46] starpit。修复（金属）：移除量化前的 K/V 偏置，避免 TurboQuant 弄乱 Qwen2。AI-native-Systems-Research/scratchy 拉取请求 #134，2026年。网址 https://github.com/AI-native-Systems-Research/scratchy/pull/134。于 2026 年 9 月 26 日 开启并合并。访问于 2026 年 11 月 20 日。[47] Haozhan Tang 、 Hao Kang、Meng Lu、Yecheng Wu、Zerui Wang、Seokho Han、Haocheng Xi、Shuo Yang、Song Han 和 Han Cai。Jet-QSparse：利用 4 位稀疏注意力加速智能体 LLM 推理。arXiv 预印本 arXiv:2611.01022，2026年。网址 https://arxiv.org/abs/2611.01022。[48] Theodore-Liu。 [core] 在量化 KV 缓存遇到较大的第一层键偏置时，于启动时发出警告。vllm-project/vllm 拉取请求 #63081，2026年。网址 https://github.com/vllm-project/vllm/pull/63081。于 2026年10月22日 开启；截至 2026年11月20日仍未关闭。访问于 2026年11月20日。[49] Theodore-Liu。 [CI][Eval] 新增对数概率漂移评估：记录参考值并与候选值进行比较。vllm-project/vllm 拉取请求 #69042，2026年。网址 https://github.com/vllm-project/vllm/pull/69042。于 2026年11月20日 开启；截至 2026年11月21日仍未关闭。访问于 2026年11月21日。[50] thomaszhang2661。 [RFC]: UltraQuant：缩放常数 $c = 0.156$ 在 UE8M0 取整前是均方误差最优的，而非取整后； $c \approx 0.2$ 可将 KV 量化 KL 散度降低 40–56%，且不增加成本。vllm- project/vllm 问题 #67176，2026年。网址 https://github.com/vllm-project/vllm/issues/67176。于 2026年11月12日 开启；截至 2026年11月20日仍未关闭。访问于 2026年11月20日。[51] Philippe Tillet、H. T. Kung 和 David Cox。Triton：用于分块神经网络计算的中间语言与编译器。发表于第 3 届 ACM SIGPLAN 机器学习与编程语言国际研讨会论文集，第 10–19 页。ACM，2019年。doi：10.1145/3315508.3329973。网址 https://doi.org/10.1145/3315508.3329973。[52] Oreste Villa、Mark Stephenson、David Nellans 和 Stephen W. Keckler。NVBit：用于 NVIDIA GPU 的动态二进制插桩框架。发表于第 52 届 IEEE/ACM 微架构国际研讨会（MICRO '52）论文集，第 372–383 页。ACM，2019年。doi：10.1145/3352460.3358307。网址 https://doi.org/10.1145/3352460.3358307。[53] Wei Wang。FuTransformer：在 Hopper 上实现 Llama 风格推理的 FP8 预填充与 INT4 解码。arXiv 预印本 arXiv:2610.23796，2026年。网址 https://arxiv.org/abs/2610.23796。[54]WizardSource。[bugfix][kernel] 恢复 Qwen2 在 int4 KV 缓存下的键偏置分数。vllm- project/vllm 拉取请求 #59674，2026年。网址 https://github.com/vllm-project/vllm/pull/59674。于 2026年10月1日 开启；截至 2026年11月20日仍未关闭。访问于 2026年11月20日。[55]XiancaiTian。[attention][TurboQuant] 保留已配置的 FlashAttention 版本。vllm- project/vllm 拉取请求 #47531，2026年。网址 https://github.com/vllm-project/vllm/pull/47531。于 2026年7月3日 开启；截至 2026年11月20日仍未关闭。访问于 2026年11月20日。[56] Guangxuan Xiao、Yuandong Tian、Beidi Chen、Song Han 和 Mike Lewis。利用注意力汇实现高效流式语言模型。arXiv 预印本 arXiv:2309.17453，2023年。网址 https://arxiv.org/abs/2309.17453。发表于 ICLR 2024。

[57] Bruce Changlong Xu、Adarsh Kumarappan 和 Mu Zhou。KV 缓存量化导致的对齐崩溃：诊断与缓解。*arXiv* 预印本 *arXiv:2606.09864*，2026年。网址 https://arxiv.org/abs/2606.09864。[58] xupinjie。TurboQuant 解码注意力［（原文如此）］优化。vllm-project/vllm 拉取请求 #53060，2026年。网址 https://github.com/vllm-project/vllm/pull/53060。于 2026年8月20日 开启；截至 2026年11月20日仍未关闭。访问于 2026年11月20日。[59] Zihao Ye、Lequn Chen、Ruihang Lai、Wuwei Lin、Yineng Zhang、Stephanie Wang、Tianqi Chen、Baris Kasikci、Vinod Grover、Arvind Krishnamurthy 和 Luis Ceze。FlashInfer：用于大语言模型推理服务的高效、可定制注意力引擎。*arXiv* 预印本 *arXiv:2501.01005*，2025年。网址 https://arxiv.org/abs/2501.01005。发表于 MLSys 2025。[60] Ted Zadouri、Markus Höhnerbach、Jay Shah、Timmy Liu、Vijay Thakkar 和 Tri Dao。FlashAttention-4：针对非对称硬件扩展的算法与内核流水线协同设计。*arXiv* 预印本 *arXiv:2603.05451*，2026年。网址 https://arxiv.org/abs/2603.05451。[61] Jintao Zhang、Jia Wei、Haofeng Huang、Pengle Zhang、Jun Zhu 和 Jianfei Chen。SageAt- tention：用于即插即用推理加速的高准确率 8 位注意力。*arXiv* 预印本 *arXiv:2410.02367*，2024 年。网址 https://arxiv.org/abs/2410.02367。发表于 ICLR 2025。[62] Lianmin Zheng、Liangsheng Yin、Zhiqiang Xie、Chuyue Sun、Jeff Huang、Cody Hao Yu、Shiyi Cao、Christos Kozyrakis、Ion Stoica、Joseph E. Gonzalez、Clark Barrett 和 Ying Sheng。SGLang：高效执行结构化语言模型程序。*arXiv* 预印本 *arXiv:2312.07104*，2023年。网址 https://arxiv.org/abs/2312.07104。发表于 NeurIPS 2024。[63] Donglin Zhuang、Pragaash Ponnusamy、Xiaoxia Wu、Shuaiwen Leon Song 和 Zhongzhu Zhou。FOSCAR：通过仿射折叠加速 2 位和 4 位 KV 缓存上的解码注意力。*arXiv* 预印本 *arXiv:2611.09722*，2026年。网址 https://arxiv.org/abs/2611.09722。[64] Zn0425。修复：通过逐令牌 FP8 KV 缓存保持 Qwen2 准确率。alibaba/rtp-llm 拉取请求 #1583，于 2026年10月27日 合并，2026年。网址 https://github.com/alibaba/rtp-llm/pull/1583。[65] Jiaxiang Zou、Yonghao Chen、Rui Meng、Ruilong Wu 和 Xinyu Chen。TensorBridge：将微缩放 FP4 权重桥接至 FP8 张量核心，以支持大语言模型服务。发表于*2026*年 *ACM SIGOPS* 年度技术会议（*ATC*）论文集，第 1784–1798 页，2026年。doi：10.1145/3828523.3857032。网址 https://doi.org/10.1145/3828523.3857032。

向 vLLM 报告的发现

我们已将研究结果报告给 vLLM，并为每项结果附上了一个最小复现程序（表14）。提交时，以下所有问题和拉取请求均处于开放状态；其中两个拉取请求的作者已根据我们的输入采取了行动，第三个拉取请求引用了我们的测量结果。

表14：向 vLLM 报告的发现，每项均附有一个最小复现程序（提交时的状态）。

位置	发现	状态
问题 #67176 [50]	UltraQuant 的缩放常数 $c = 0.156$ 适用于连续缩放，但应用时采用二次幂取整。UltraQuant 论文将 0.195 称为二次幂最优值，但在一次运行中测得其准确度低了 4.4 个百分点（其表 5） [7]；我们发现，0.20 可在线下将 KL 散度降低 40–56%，并且困惑度在 vLLM 中比 BF16 提升 66–79%（Llama-3.1-8B、Mistral-7B-v0.3、Qwen3-8B-Base），在相同的字节数	由我们提交；我们发布了两种修复方案（常数设为 0.20，或者采用两者中更好的方案）每个 2 的幂指数组）使用单元测试和在三个模型上测得的困惑度，并请维护者选择
问题 #41343 [36]，拉取请求 #63081 [48]	FP8 KV 会悄无声息地破坏 Qwen2 系列推理模型在第 0 层出现问题；跳过第 0 层可恢复准确率，但会导致部分模型的 FlashAttention-3 崩溃(#65821 [42]；已在未合并的拉取请求 #65891 [43] 中提出修复方案)	我们发布了文本模型数据在 #41343（一个现有的问题关于带有 `fp8_e5m2`）以及 #63081 中；#63081 的作者在其文档中添加了一条关于崩溃的说明
拉取请求 #46461 [13] → 拉取请求 #67789 [6]	TurboQuant 和 UltraQuant 会跳过缓存的 KV，当一个带有缓存上下文的请求与一条较长的新提示词，导致答案错误且没有 #46461 仅修复了 TurboQuant	我们报告称 UltraQuant 存在相同的错误，并附带一个复现程序；#67789 修复了这两个问题，并表明默认的分块预填充也会触发该问题；我们已确认在 H200 上它可以修复该问题 UltraQuant 端到端（0/4 → 4/4 两者均正确触发器）；其作者添加了我们的结果到拉取请求描述中，并且我们提供了一个后续回归测试，用于前缀缓存触发器
RFC #61698 [32]，拉取请求 #69042 [49]	RFC 所提出的这类提示词对数概率检查对于数值计算拉取请求，可以捕获第0层 FP8 故障，但无法捕获上述缓存 KV 错误，后者仅在一个缓存前缀与更长的新提示词共享一个步骤；针对该形状的请求可以捕获这一问题	我们发布了 H200 测量结果，见 #61698；#69042，一项对数概率漂移评估针对该 RFC，引用了这些结果，使缓存读取路径超出其范围，并提议将此类请求作为后续事项 test，我们主动提出为其编写

我们还发现，vLLM 0.33 无法在 NVIDIA GPU 上将 enableUltraQuant 与其默认边界层一起启用，因为 CUDA 后端优先级列表中缺少 UltraQuant；我们

插件会添加它。我们已在 #46461 和 #67789 线程中指出这一点，但尚未单独提交问题。