FlashInfer-Bench：为人工智能驱动的大语言模型系统构建良性循环

Shanli Xing [*1] **Yiyan Zhai** [*2] **Alexander Jiang** [*2] **Yixin Dong** [*2] **Yong Wu** [3] **Zihao Ye** [3] **Charlie Ruan** [4] **Yingyi Huang** [2] **Yineng Zhang** [5] **Liangsheng Yin** [5] **Aksara Bayyapu** [2] **Luis Ceze** [1 3] **Tianqi Chen** [2 3]

arXiv:2601.00227v1 [cs.AI] 1 Jan 2026

摘要

近期进展表明，大语言模型（LLM）可以充当能够生成 GPU 内核的自主智能体，但将这些人工智能生成的内核集成到实际推理系统中仍然颇具挑战。FlashInfer-Bench 通过建立一个连接内核生成、基准测试与部署的标准化闭环框架，弥补了这一缺口。FlashInfer Trace 是其核心，它提供了统一的模式，用于描述内核定义、工作负载、实现和评估，从而实现智能体与系统之间的一致通信。FlashInfer-Bench 基于真实的服务跟踪数据，包含一个精选数据集、一个兼顾正确性与性能的稳健基准测试框架、一个用于跟踪大语言模型智能体 GPU 编程能力的公开排行榜，以及一种动态替换机制（apply()），可将性能最佳的内核无缝注入 SGLang 和 vLLM 等生产级大语言模型引擎。借助 FlashInfer-Bench，我们进一步评估了大语言模型智能体的性能与局限性，比较了不同 GPU 编程语言之间的权衡，并为未来的智能体设计提供了见解。因此，FlashInfer-Bench 为持续改进人工智能生成的内核并将其部署到大规模大语言模型推理中，建立了一条切实可行且可复现的路径。

1 引言

大语言模型（LLM）的迅速发展催生了计算领域的新时代，但其底层推理系统的性能和成本日益制约着它们的广泛部署（Zheng 等人，2025年； Kwon 等人，2023年； 英伟达， 2025年； MLC 团队，2023-2025年）。这些系统的核心是执行注意力、矩阵乘法和采样等核心操作的 GPU 内核。优化大语言模型推理系统所用的 GPU 内核，需要投入深厚且专家级的工程工作。

本文提出了一个实际问题：如何将人工智能生成的内核有效地融入生产环境中的大语言模型系统？近期研究（Ouyang 等人，2025年；Li 等人，2025年；Baronio 等人，2025年；Fisches 等人，2025年）初步表明，大语言模型能够生成复杂的底层 GPU 代码。然而，要将人工智能生成的代码应用于实际部署，仍面临三项根本性挑战。首先，大语言模型系统中的内核依赖于许多不同特征，例如不规则分布和数据精度，这些因素都会影响其性能。因此，向人工智能智能体有效传达这些信息至关重要。其次，现实世界中的大语言模型推理流量可能不同于我们在单个内核中选用的典型均匀或随机设置。我们需要一种有效的方法，来跟踪内核在现实世界大语言模型推理工作负载中的性能。最后，即使人工智能生成了有前景的内核候选方案，仍可能存在集成方面的差距，因为要将这些方案接入端到端大语言模型系统，还需要额外的工程工作。

为应对这些挑战，我们推出了 FlashInfer-Bench，这是一个面向人工智能驱动的大语言模型系统的基准测试及标准操作流程（图1）。为规范整体工作负载，我们推出了 FlashInfer-Bench Trace，这是一种自包含的标准 JSON 模式，用于描述内核任务、工作负载、解决方案和最终评估结果。基于该模式，我们从真实世界的大语言模型工作负载中整理构建了 FlashInfer-Bench 数据集。此外，我们基于 FlashInfer Trace 设计了一个稳健的内核基准测试框架，该框架具备运行时隔离功能，可防止与性能相关的奖励操纵，并专门支持低比特内核和非确定性采样内核的评估。最后，我们构建了一种动态内核替换机制，可在运行时直接更新 FlashInfer 内核库，将算子重定向到 FlashInfer-Bench Trace 提供的最优内核。通过这种方法，我们能够将常见的大语言模型生成的内核直接集成到 SGLang 和 vLLM 等开源大语言模型引擎中，无需修改代码。

我们建立了一个实时排行榜，用于跟踪前沿模型在真实世界工作负载和大语言模型工作负载上的 GPU 编程能力。我们还对当前

[*]共同贡献 [1]华盛顿大学 [2]卡内基梅隆大学 [3]英伟达 [4]加利福尼亚大学伯克利分校 [5]独立研究人员。通讯作者：Yixin Dong< yixind@andrew.cmu.edu>、Tianqi Chen <tqchen@cmu.edu>。

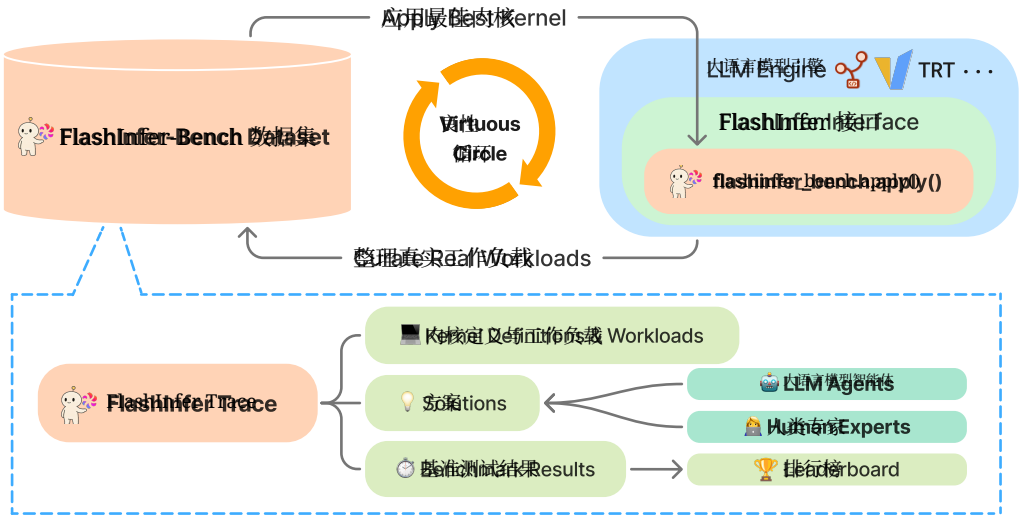

图*1*。FlashInfer-Bench架构。FlashInfer Trace 提供标准模式，用于指定内核契约和语义，以及传递实现和评估结果；FlashInfer-Bench 数据集整理了生产环境中的大语言模型服务工作负载；而 flashinfer bench.apply() 则将经过验证的最快实现直接部署到大语言模型推理引擎中。

真实世界大语言模型推理系统中的大语言模型智能体现状进行了全面研究。我们的评估和分析表明：(1) 大多数正确性错误源于编译失败；(2) 模型难以利用架构规格或内在函数等硬件特定细节；(3) 语言之间存在取舍：Triton 等高级语言在大多数任务上能带来更好的性能，而低级语言 CUDA 则为专用优化提供了更大的潜力。

主要贡献如下：

- 我们提出了 FlashInferTrace，用于规范任务、工作负载和解决方案的描述，以支持人工智能生成的工作负载。
- 我们整理了 FlashInfer-Bench 数据集，为在真实世界工作负载上评估人工智能生成的内核提供了丰富的基础。
- 我们提出了一套务实的运维工作流程，可持续生成人工智能生成的内核，并将其直接应用于真实世界的生产系统。
- 我们全面分析了大语言模型生成的内核在大语言模型系统上的表现。

本文其余部分安排如下：第2节回顾大语言模型推理、GPU 内核以及用于 GPU 内核生成的大语言模型的背景。第3节介绍 FlashInfer-Bench 的设计，包括 FlashInfer Trace 模式、数据集整理、稳健且感知性能的基准测试框架，以及面向生产引擎的动态内核替换。第4节详细介绍数据集和对智能体生成的内核的全面评估，其中包含关于通用矩阵乘法和分组查询注意力解码的案例研究，以及端到端替换结果。第5节综述相关工作。第6节总结全文。

2 背景

2.1 大语言模型推理流水线与 GPU 内核

现代大语言模型推理由大语言模型服务引擎驱动，这些引擎负责批处理、调度和并行，并由 GPU 内核调用和 CPU 逻辑组成。GPU 内核占据了大部分执行时间，因此优化这些内核可以直接降低大语言模型引擎的延迟。尽管模型种类繁多，但大多数模型共享一小组 GPU 内核，包括：

1. **通用矩阵乘法**：输入和输出可以是 bf16 或低比特格式（例如 FP8）。要实现最高速度，需要使用张量核心指令。低比特变体还需要额外的量化/反量化逻辑。

2. **注意力及其变体**：例如分页、分组、基数和多潜在注意力。实现这些算子需要张量核心，并需要采用 FlashAttention 等特殊优化。3. **融合混合专家（MoE）**：一种融合内核，可处理 MoE 路由逻辑以及对应多个专家的多个多层感知机。4. **采样与后处理**：例如核采样、Top-k 采样和温度。这些都是非确定性算子，其结果取决于输入分布和随机数。

2.2 内核优化方法

内核优化对硬件（SM 数量、内存层次结构、张量核心代际）、数值格式（FP16/BF16/FP8/INT8）和工作负载形状（序列长度/批次长度、缓存布局、稀疏性）高度敏感，因此难以找到通用的“一刀切”内核。系统构建者一直依赖三类技术进行内核优化：

内核库和模板。 高度优化的库提供了强大的基准实现，但通常无法利用特定工作负载的结构（例如，不规则序列、融合尾处理），除非使用自定义内核(Thakkar 等人，2023年)。

基于搜索的自动调度。 模板自动调优器等系统会在固定搜索空间内探索参数化调度方案，以寻找合适的分块、线程/线程块映射和融合策略。这类系统功能强大，但受限于模板的表达能力；当需要针对不同硬件或形状重新搜索时，搜索成本可能高得令人难以承受(Chen 等人，2018年； Zheng 等人，2020； Shao 等人，2022年)。

生成式程序合成。 近期的大语言模型可以直接编写底层 GPU 代码，有时还能发现现有模板之外的新颖融合和数据流模式。这拓展了巨大的设计空间，但如果缺乏严格验证，也会带来正确性和安全风险；Triton 等轻量级领域特定语言让自定义内核更易于使用(Tillet 等人，2019年)。

FlashInfer-Bench 融合了后两者的优势：它既能让生成过程提出结构上全新的内核，又通过配套的严谨、达到生产级标准的评估框架来防止性能回退和奖励操纵。

2.3 用于 GPU 代码生成的大语言模型

近期进展表明，只要提供接口描述和反馈循环，大语言模型就能合成并非平凡的 GPU 内核和融合算子。KernelBench 等公开评Ouyang 等人，2025年主要评估生成能力——模型能否生成可编译的内核，使其在选定输入上与参考实现匹配，并实现合理的加速比?在实践中，从具备能力迈向生产应用还需要额外条件：精确的任务规范（API 语义、支持的形状/数据类型、内存布局约束），防范奖励操纵和非确定性，并覆盖真实场景——
工作负载分布，以及用于部署或回滚候选方案的敏捷路径。像 Triton 这样的轻量级领域特定语言（Tillet 等人，2019年）让人类和智能体都能轻松编写自定义内核，而 FlashInfer-Bench 则补齐了缺失的运行支撑框架——FlashInfer Trace 用于标准化任务交换、确保安全性和正确性的强健验证器，以及首日动态替换，从而无需重写引擎即可实现系统级收益。

3 FLASHINFER-BENCH 设计

3.1 FlashInfer Trace

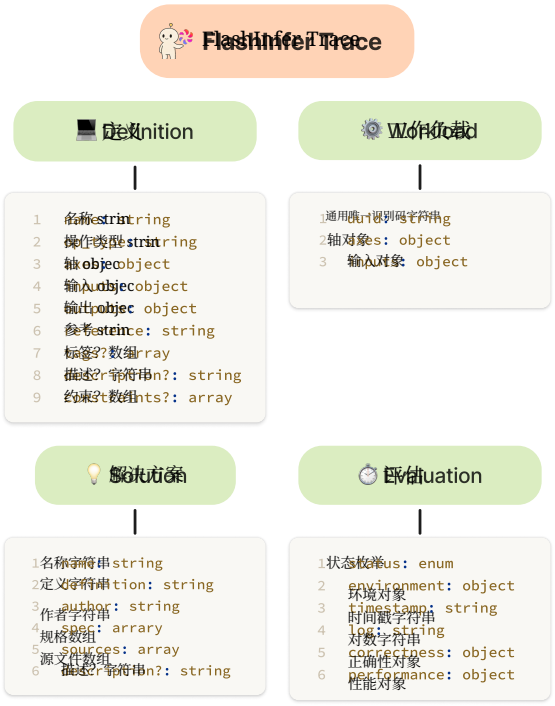

图 2. FlashInfer Trace 模式设计。定义描述内核任务。工作负载描述内核的真实世界输入。解决方案描述人工智能生成的解决方案。评估描述基准测试得出的评估结果。每个组件还包含便于分组和筛选的辅助字段。

为了打通从内核生成到评估再到部署的闭环，我们需要一种人类和智能体都能阅读的标准化语言。FlashInfer Trace 充当这种通用语言。它阐明了内核的语义契约、实现和具体评估。该抽象有意保持简洁（例如，我们不会在内核定义中公开与实现相关的系统元数据），同时又足够 （不同算子会引入所需的关键轴和约束）。

该**FlashInfer Trace 模式**包含四个组成部分。概览如图 2所示。以下四个组成部分共同构成一个独立完整的**追踪**对象，确保其可移植性和可复现性。

定义。 用于描述算子输入/输出张量及其数据类型、维度轴（可以是静态值 **常量**，也可以是由工作负载决定的值 **变量**）的 JSON 规格，以及

一个纯 PyTorch 参考函数，作为数学语义的唯一来源。可选约束用于编码各轴之间的关系。

工作负载。针对特定内核定义的具体测试输入。所有变量轴均赋予整数值，每个输入通过以下方式之一实例化：记录的 safetensors 转储、随机生成或字面标量值。

解决方案。满足所选定义的接口和语义的具体实现。它提供源文件和可调用入口点，以及兼容性元数据（例如目标 GPU 架构和软件版本）。同时支持可扩展的语言/领域专用语言。

评估。 一条不可变的基准测试记录，可精确绑定特定的**定义** × **解决方案** × **工作负载**，并报告运行状态、正确性与性能摘要，以及执行环境快照。

FlashInfer Trace 模式原生支持动态和静态内核形状，其中每个轴都可以定义为`var` 类型（其值由工作负载决定），或常量 类型（其值在编译时固定）。这使人工智能能够针对特定形状优化内核。它还支持不规则输入，例如注意力中使用的页表。为此，可以将完整的页表张量和存储索引指针的整数张量都作为输入提供，从而使系统能够精确描述不规则张量输入。具体示例见附录 A。

3.2 FlashInfer-Bench 数据集

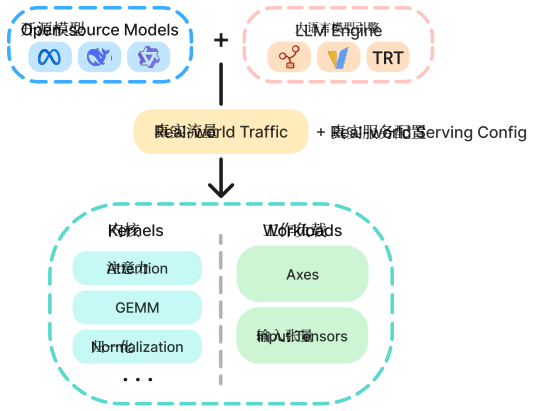

图 3。 FlashInfer-Bench 数据集收集流程。我们使用默认的常用配置，让主要模型处理真实流量，并整理内核定义和工作负载。

基于 FlashInferTrace，我们整理出 FlashInfer-Bench 数据集，这是一个不断演进的标准，将常见的服务内核定义与代表性工作负载相结合，持续跟踪并收集目标，让人类和智能体都能据此进行优化。

我们的目标是贴近真实场景，并聚焦于大语言模型服务。我们涵盖 DeepSeek-V3、Llama-3.1-8B 和 Qwen3-30B-A3B 等模型，涉及通用矩阵乘法、注意力、归一化、采样和混合专家等算子族。工作负载通过在 SGLang 中以默认且常用的配置（例如，DeepSeek-V3 使用原生 FP8 量化和 8 的张量并行规模）运行这些模型，并使用 ShareGPT 提示词为其提供服务来收集。

收集内核定义时，当且仅当两个内核调用满足以下条件时，我们将它们归为同一个定义：(i) 使用相同的输入输出规范并运行相同的参考语义；(ii) 暴露相同的一组轴，且各轴具有相同的常量/变量角色；(iii) 所有常量轴的值都一致。

我们有意优先采用具体的定义，而非宽泛的定义，理想情况下细化到特定模型层的内核调用，从而尽可能实现内核优化和明确无歧义的调度。我们不允许可选输入或行为标志，并将默认行为编码在运行参考中，使其成为契约的一部分。若行为必须有所不同，我们会引入新的定义，而不是添加运行时开关。

我们对收集到的工作负载进行筛选和去重，并采用**兼顾性能与多样性**的缩减方法。当输入值会显著影响内核性能（例如采样概率分布）或正确性（例如极端边界情况）时，我们会转储完整张量；否则，我们会使用带种子的随机运行时张量，以节省存储空间。

随后，我们沿性能敏感轴（例如批大小）和张量统计量（例如注意力的平均序列长度）进行去重，在保留多样性和代表性的同时缩减集合。最终，每个定义保留 ~ 50 个工作负载用于评估。

3.3 RobustKernel 基准测试

为确保整个内核评估流程的鲁棒性和效率，我们构建了一个基准测试子系统，可进行严格的正确性验证和稳定、可复现的计时，并能在多设备环境中原生运行。

确定性内核。 对于预期会产生确定性输出的操作（例如通用矩阵乘法、归一化、注意力等），我们直接逐元素比较内核的输出与参考输出。若每个元素都满足以下条件，则认为内核的输出 y_{sol} 正确：

$$|y_{\text{sol}} - y_{\text{ref}}| \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \cdot |y_{\text{ref}}| \tag{1}$$

其中 y_{ref} 是参考输出。我们还会拒绝任何包含非有限值（NaN（非数）或无穷大）的输出。对于每个内核，我们会记录所有测试元素和试验中观测到的最大误差。只有当每个输出元素的误差都在允许的误差界限内时，确定性内核才会通过正确性检查。

低精度内核。 与全精度基线相比，使用较低精度算术（例如 FP8）的内核会系统性地产生更大的误差。我们不采用单一的宽松全局容差，而是使用匹配比例规则：如果至少有 ρ 的输出符合标准误差判据，则该内核视为正确。例如，对于 $\rho = 0.95$，我们要求 95% 的输出元素通过严格的误差界限，允许少量离群元素。

随机内核。 对于采样等随机算子，由于输出因运行而异，逐元素比较并不适用。相反，我们会验证采样输出是否符合正确的概率分布。我们使用输入概率$\mathbf{p}$ 和可选掩码 $\mathcal{M}$ （例如 Top-k 采样或核采样/top-p）推导真实分布**查询** ，对其归一化后，多次执行内核以获得经验分布$\mathbf{f}$。最后，我们计算经验分布与期望分布之间的全变差距离（TVD），并要求 TVD 小于选定的阈值 τ_{TVD}。之所以采用 TVD，是因为它能直接给出任意事件上最坏情况概率误差的上界。除 TVD 检查外，我们还会验证每个样本是否都通过阈值掩码；任何掩码违规（例如采样到给定 Top-k 规则应排除的索引）都会导致立即失败。

性能测量。 我们为每个 GPU 维护一个可供多进程访问的设备锁。为防止同一设备上的进程或任务相互干扰，计时例程仅在获取设备锁后运行。每个内核先执行 w 次不计时预热运行，再执行 m 次计时运行。我们采用基于 CUDA 事件的设备端计时，并报告 m次测量运行的平均延迟。

隔离。 为尽量减少不同解决方案之间的相互干扰，并降低大语言模型“钻基准测试空子”的风险（例如读取残留内存以推断参考输出），我们提供了完全隔离的基准测试模式：每个解决方案都在独立的子进程中运行，并在运行完成或超时后终止，同时销毁 CUDA 上下文，避免不同运行之间状态残留。我们还提供持久模式：每个 GPU 配备一个长期运行的工作进程，并设有一个由少量预热工作进程组成的备用池。这种模式大幅降低了子进程和 CUDA 上下文的初始化开销；如果某个解决方案失败并破坏了上下文，也能快速恢复。这两种模式兼顾了大规模测试所需的效率，以及全面隔离。

系统支持。 随着数据集规模扩大，高效基准测试对于及时、可持续地推进工作流程变得至关重要。基于对效率的重视，我们构建了一个可扩展、容错的多设备基准测试服务。对于每个已就绪的解决方案×工作负载作业，我们使用调度器构建成本矩阵，根据可用设备工作进程评估基线驻留和预热编译缓存。随后，调度器使用匈牙利算法分配微批次，并通过指数加权移动平均在线更新成本模型，然后再处理下一批次。调度器会执行工作进程健康检查并处理故障恢复。执行默认采用持久模式；反复失败的方案将被延后，改为在隔离模式下运行。输入和参考输出会在目标设备上实例化，并在可能时重复使用。已编译解决方案会缓存在内存中，必要时（例如 CUDA 二进制文件）还会持久化到磁盘，以避免重复构建。

3.4 公开排行榜与持续评估

我们搭建了一个公开排行榜，其基础是第3.3节（见图4）。排行榜接受 FlashInfer Trace 格式的提交，在真实工作负载上进行评估，并报告按内核和设备分层的指标，包括正确性、不同加速比阈值下的性能曲线、单项工作负载延迟以及端到端延迟差值。为确保可引用性和可复现性，我们定期发布包含带版本数据集的冻结快照，同时滚动排行榜反映最新的评估结果。该服务实施反奖励操纵防护措施（运行时隔离、隐藏工作负载，以及针对确定性、低精度和采样内核的专用验证器）。我们在第4节中分析当前快照。

3.5 生产环境中的内核动态替换

以往的工作流程要求在服务引擎中手动更改代码以部署优化内核，这形成了智能体驱动自动化的瓶颈，并阻碍了评估—部署闭环。为提供经过验证的优化内核并打通这一闭环，我们推出了 flashinfer-bench.apply()。它提供零侵入、低开销的路由机制，可将服务请求动态映射到数据集中性能最佳的实现。

接口与用法。 flashinfer bench.apply () 提供两个 API。**装饰器** API 用于包装一个运算符；被包装的函数作为回退实现。它接受固定的 Definition 名称，或接受一个解析器，将绑定的运行时参数映射到 Definition 名称。**命令式** API 允许从任意位置自定义内核调用；它会为给定的输入解析性能最佳的解决方案，



图 *4*。 FlashInfer-Bench 排行榜。在 fast$_{0.95}$ 上表现最佳的模型是 Gemini-2.5-Pro、GPT-O3 和 GPT-5-2025-08-07。正确性方面表现最佳的模型是 GPT-5-2025-08-07（通过率 83.9%）、GPT-O3（通过率 71.3%）和 Gemini-2.5-Pro（通过率 48.8%）。

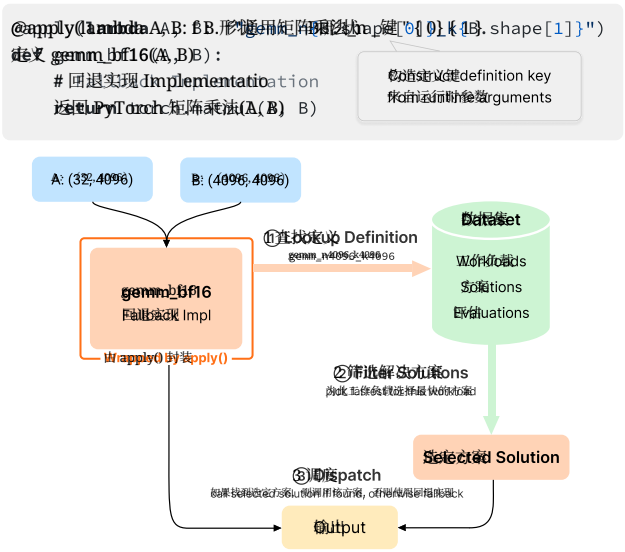

图 *5*。 flashinfer bench.apply() 的工作流程。它是一个动态分派器，可在运行时使用内核输入检索最佳解决方案，并返回执行结果。

并立即返回结果。我们提供与 FlashInfer 的一流集成，因此只需启用环境变量 FIB ENABLE APPLY=1 ，即可路由常见操作，无需更改代码。当全局禁用 flashinfer bench.apply() 时，调用会透明地转交给原始实现。

离线缓存预构建。 为尽量减少 apply() 对服务性能的影响，我们引入了一个为动态分派而提前（AOT）构建的索引，将在线分派计算减少到少量 $O(1)$索引查找。

在服务引擎启动前，apply() 运行时会从本地数据集初始化一个索引。它首先根据可配置的错误阈值筛选跟踪记录，然后从这些跟踪记录中的工作负载提取特征（例如形状）以构成键，并为每个键选取最快的解决方案作为索引值。在所有选定方案中，我们会按可配置的比例，编译被选中次数最多的方案，并提前（AOT）将其编译为可执行文件。其余方案则通过即时编译（JIT）进行编译，以平衡构建成本和运行时开销。

在线轻量级分派。运行时，我们根据内核的输入参数构造键，执行 $O(1)$ 在索引中查找，并找到或编译有效内核以供执行。启用 CUDA 图并进行适当预热后，开销可以忽略不计（见第 4.5 节）。

4 数据集概述与评估

4.1 数据集概述

我们的数据集包含大语言模型推理中使用的八种代表性内核类型：通用矩阵乘法、不规则与分页式分组查询注意力、不规则与分页式多头潜在注意力、融合混合专家、均方根归一化和采样。这些类型涵盖了现代大语言模型的核心组件，包括融合型、非确定性和低比特内核（例如 FP8 通用矩阵乘法）。对于每种内核类型，当某些维度（例如隐藏维度）为固定常量时，我们将这些参数的每一种独特组合视为一个单独的定义，由此得到 41 种不同的内核配置。对于工作负载，我们利用输入跟踪记录，在 SGLang 上收集真实输入实例，并通过基于形状的去重和筛选，整理出包含 1,600 个工作负载的数据集，覆盖短序列和长序列场景。我们使用智能体框架（第4.2节）和前沿模型生成方案，包括 Gemini 2.5 Pro、Claude Opus 4.1、GPT-5 和 OpenAI o3，并采用 CUDA 和 Triton 实现。值得注意的是，FlashInfer-Bench 还支持其他领域特定语言，例如 CUTLASS 和 CuTe DSL。总体而言，我们在所有工作负载上评估了 240 个方案，产生了 9,600 个评估结果。

4.2 智能体评估设置

设置。为了确保在多个模型之间进行公平且准确的评估，我们设计了一种反馈循环智能体，如算法 1所示。在每次迭代中，智能体生成一个内核，使用 FlashInfer-Bench 对其进行评估，并根据结果优化设计，直至达到改进上限。选择所有迭代过程中生成的最佳内核，作为最终评估方案。内核性能在 NVIDIA B200 图形处理器上进行测量。

算法 1 反馈循环智能体

输入： 定义、语言、硬件
输出： Sol* （最佳解决方案）
$\mathcal{S} \leftarrow \emptyset$
智能体 $\leftarrow$ CodeAgent.Initialize(定义，语言，硬件)
求解$_0$ $\leftarrow$智能体.生成()
对于 $i \leftarrow 0$ **到** $N-1$ **执行**
  跟踪$_i$ $\leftarrow$ FlashInfer-Bench.基准测试(定义，求解$_i$)
  如果跟踪$_i$.状态 $=$ 通过 **则**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{$(求解$_i$，跟踪$_i$)$\}$
  求解$_{i+1}$ $\leftarrow$智能体.优化(跟踪$_i$)
Sol* $\leftarrow$ arg max$_{(\text{Sol}_i, \text{轨迹}_i) \in \mathcal{S}}$ 轨迹$_i$.加速比
return 解*

指标。我们采用 Kernel- Bench(p 中的快速指标（Ouyang 等人，2025年）。对于一个方案，快速$_p$ 表示内核运行速度比基线内核快超过 p 倍的工作负载所占比例。形式上，

$$\text{fast}_p = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}(\text{correct}_i \wedge \{\text{speedup}_i > p\}) \qquad (2)$$

通过改变 p，我们可以得到不同的快速$_p$ 值，从而形成一条曲线，其曲线下面积（AUC）代表内核的整体性能。这条曲线同时反映内核的正确性和性能。我们选择最先进的内核库 FlashInfer（Ye 等人，2025年）作为比较基准。如果没有对应的 FlashInfer 内核（例如 bf16 矩阵乘法），则改用 PyTorch。对于每个模型，我们还会对所有生成方案的快速$_p$ 值取平均，以获得该模型的快速$_p$曲线。值得注意的是，当 $p = 0$时，快速$_p$ 值表示智能体生成的内核的正确率。

4.3 智能体能力分析

评估结果如图 6所示。我们展示了三种具有代表性的内核的性能$_p$ 曲线：通用矩阵乘法、GQA 分页解码和 RMSNorm。结果表明，大语言模型在大多数情况下的表现落后于人类：对于通用矩阵乘法（Triton）、GQA（CUDA）和 GQA（Triton），在超过一半的工作负载中，其性能不到最先进性能的 50%。RMSNorm 的表现接近或超过人类水平，因为它受内存限制，一旦内存带宽达到饱和，性能便触及上限，因此更容易优化。相比之下，通用矩阵乘法和 GQA 受计算限制，需要采用流水线和分块等技术来避免停顿，因此更难优化。通用矩阵乘法（CUDA）的出色表现源于智能体学会了使用库内核，稍后将对此进行讨论。

大多数正确性错误都源于编译失败。在全部 32 个正确性错误中，30 个由编译错误导致，只有 2 个属于运行时错误或数值错误。数值错误主要源于针对大型输入形状的分派内核中填充计算不正确。我们将编译错误归纳为几种代表性类型：

1. **API 使用错误**。模型可能调用了正确的 API，却使用不当，这种情况在生成 Triton 代码时尤为常见。例如，模型可能尝试对 constexpr 变量进行索引，而 Triton 不支持这种操作。我们认为，这是因为训练期间与 Triton 相关的数据有限，导致模型无法学习正确的使用模式。模型偶尔也会生成不存在的 API，例如 nv bfloat162 to - float2，不过经过几轮优化后，这些问题通常都能得到解决。

2. **主机—设备混淆**。模型有时会混淆主机代码和设备代码。例如，在 Triton 内核中，它可能会在主机端调用 math.log，而不是 tl.log。这可能是因为模型难以准确识别当前的执行上下文。

3. **数据类型错误和形状错误**。模型有时会生成数据类型使用不正确或张量形状不匹配的代码。例如，它可能会为 reduce add sync 函数使用浮点输入，而该函数只允许使用整数输入。

大语言模型难以正确运用硬件内在函数和优化技术。尽管我们提供了硬件信息，包括规格和相关文档，模型仍未能利用这些知识来最大限度地提升内核性能，尤其是在最新硬件上。例如，在通用矩阵乘法 CUDA 任务中，模型仅依赖 wmma 进行优化，却未能正确使用效率更高的 mma，或 Blackwell GPU 上可用的新型张量指令 tcgen05，导致性能欠佳。在注意力任务中，模型尝试用 CUDA 实现 FlashAttention 算法，却只复现了在线 softmax 组件，未能恰当使用分块或张量核心。这些观察表明，模型针对特定硬件进行专门优化的能力仍然有限。

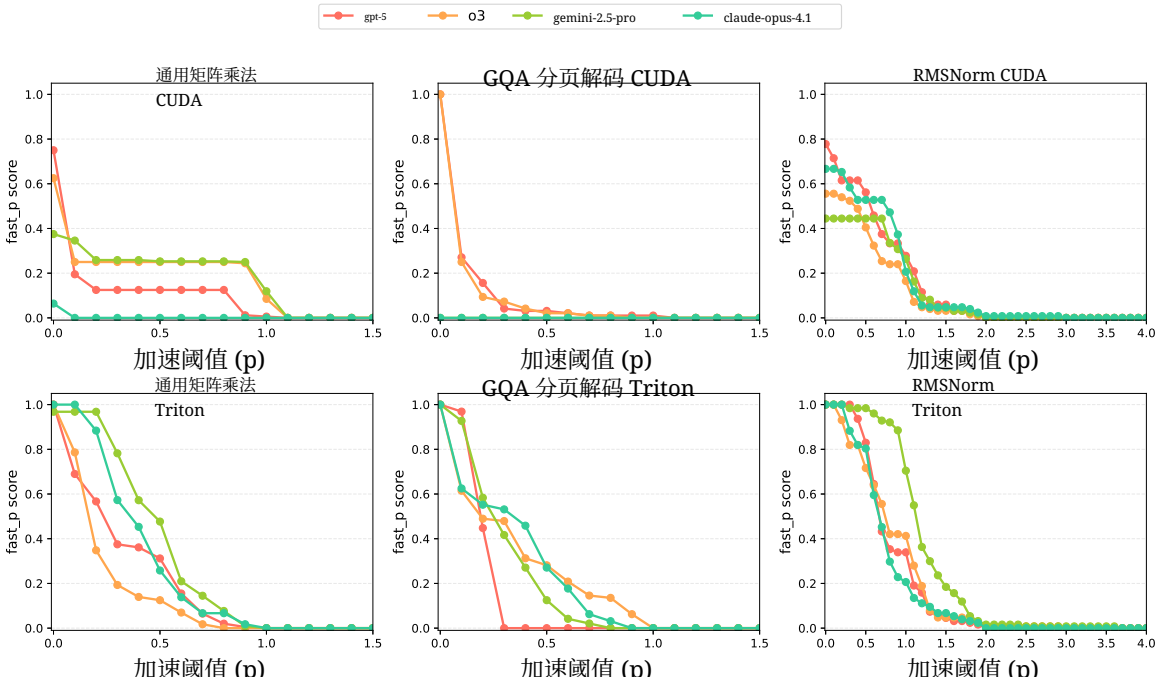

图 6.快速$_p$ 指标 ca 同时涵盖正确性和性能两个轴，定义为内核-工作负载评估中满足以下条件的比例： at
均正确，且达到 相较于基准实现的加速比超过阈值 p。

未来的工作可以采用强化学习或类似方法，鼓励模型更充分地探索硬件能力。

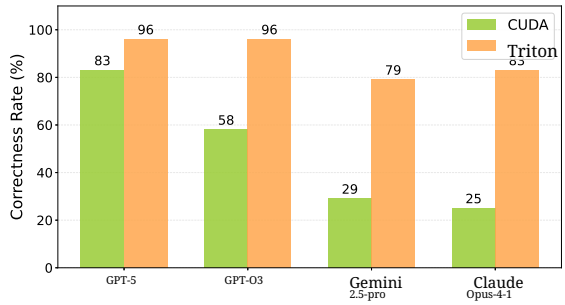

图 7. 大语言模型智能体在不同语言中的总体正确性。

GPU 编程语言给智能体设计带来了权衡。 GPU 编程语言大致可分为两类： **高级抽象** ，例如`Triton` 和 `TVM`，它们封装了底层硬件细节，只需表达高级计算逻辑；另一类是**低级语言**，例如 `CUDA` 和 `PTX`，需要显式处理硬件特定细节。我们发现，与编写 图 7所示的 `Triton` 内核相比，模型编写 `CUDA`内核时的正确性（`Triton`）和速度都显著更高。我们认为这有几个原因。首先，`Triton`代码更短、更精简，降低了对模型的长上下文推理能力的要求。其次，Triton 只需表达高级（分块级）计算逻辑，编译器便会自动确定最佳的底层实现，从而能够利用 tcgen05 等先进硬件特性，而智能体难以在 CUDA 中利用这些特性。不过，由于 CUDA 提供了更精确的硬件抽象（例如对共享内存的显式控制），其性能上限更高。这表明，提升智能体使用 CUDA 的能力，有望带来更大的性能提升。

智能体已经学会调用库，而不是编写原始内核。 例如，在 GEMM CUDA 任务中，我们观察到，基于 Gemini-2.5-pro 和 o3 的智能体在一些方案中学会调用 cuBLAS库的矩阵乘法内核，其性能达到或超过了基准实现。这一发现有两点含义：第一，该模型展现出了利用环境生成接近人类水平 CUDA 代码的能力；第二，这表明模型依赖库调用，而不是真正理解 GPU 编程，这可能会成为训练过程中的瓶颈。我们建议在训练期间限制库访问，以更好地获取技能；而在推理期间允许不受限制地使用库，从而最大限度地提升内核性能。

4.4 智能体生成的内核案例研究

GEMM - 编译器的帮助 在此案例研究中，我们分析 GPT-5 生成的用于定义的 Triton 和 CUDA 内核`gemm n4096 k4096`。Triton 内核实现了 4。5×相对于 CUDA 内核的加速比，在经过测试的工作负载中，平均执行时间为 0.11ms，而 CUDA 内核为 0.5ms。完整实现见附录；请参阅 B.1 节和 B.2 节。

该 CUDA 内核采用三阶段分块策略，块分片大小固定为 128×256×64，每个块包含 8 个线程束。每个线程束通过按 4×4 网格排列的 16 次 WMMA 操作，计算一个 64×64 输出区域。该实现通过手动管理共享内存并采用错位填充来避免存储体冲突，同时使用向量化的 128 位加载来最大限度地提高内存带宽。总体而言，该 CUDA 内核遵循简单的“加载-同步-计算-同步”模式，没有跨块预取或软件流水线，主要依靠硬件层面的块内重叠来实现计算与内存访问的并发。

该 Triton 内核采用类似的高层分块策略，但利用 Triton 的自动调优装饰器在运行时评估四种不同的分块配置，并根据工作负载的 M 维度选择最优配置。该内核受益于编译器内置的软件流水线（深度设为 4），可在 K 维迭代过程中重叠计算与内存预取。

性能差异最大之处源于张量核心原语的选择。CUDA 内核使用 WMMA 指令，而 Triton 内核则通过其 tl.dot() 算子自动使用 tcgen05 指令。这凸显了 CUDA 代码生成的一项关键局限：当新架构引入新的内在函数时，智能体的语料库起初缺乏足够的训练示例，导致智能体默认采用较旧且次优的模式。此外，Triton 的高级加载/存储抽象（tl.load()、tl.store()）免去了智能体管理共享内存分配、避免存储体冲突以及实现内存合并访问的需要。这表明，具备适当抽象的领域特定语言能够降低协调多种优化策略所需的认知复杂度，从而帮助智能体更有效地探索和应用先进的优化技术。

GQA 分页解码：优化并非易事 GPT-5 的 GQA 分页解码 CUDA 实现几乎没有进行优化，仅通过在线 softmax 提高内存效率。每个块处理一个批次元素和一个键值头，使用标量 FP32 算术进行注意力计算，并采用基本的内存访问模式。它缺少当前最先进内核普遍采用的优化技术，例如通过按块分片来尽量减少 HBM 内存流量、异步执行，以及通过流水线实现重叠通用矩阵乘法和 softmax 操作（Shah 等人，2024）。完整实现见 B.4 节。

我们曾尝试通过明确提示大语言模型智能体（GPT-5-2025-08-07，使用高推理模式），提供这些注意力内核优化策略以及 CUDA 内建函数相关指令，但经过 10 次尝试，该大语言模型智能体仍未能生成能够正确运用这些优化的内核。

本案例研究表明，尽管 CUDA 提供了细粒度的硬件控制能力，并支持复杂的优化，但预训练大语言模型难以利用这些能力，因为这会增加实现的复杂度。收到提示后，智能体能够识别高层优化策略，却无法正确协调 CUDA 实现所需的底层细节。要有效利用 CUDA 的表达能力来实现复杂内核，可能需要额外的训练方法，例如通过高质量执行反馈进行强化学习，或使用由专家内核实现构成的精选数据集。

总结 总的来说，这些案例研究表明，尽管 GPT-5 等大语言模型能够生成在功能上正确的内核，但在底层 GPU 优化方面，它们难以达到专家级性能。高层抽象的领域特定语言借助编译器支持，使它们能够实现具有竞争力的效率；但直接生成 CUDA 代码则暴露了它们在内存管理、分块和硬件专属调优方面的弱点。

4.5 端到端系统中的内核替换

我们旨在展示动态内核替换机制（flashinfer bench.apply()）的有效性和效率。为此，我们提供同一内核定义的多个实现，它们速度各异，均由智能体生成并经 FlashInfer-Bench 验证，随后将它们动态替换到服务引擎中。我们预期，更快的内核会降低端到端请求延迟，从而验证内核替换的有效性。我们还从服务引擎中提取原始内核，并限制动态调度器，使其只能调度到该内核；随后比较启用和未启用替换时的端到端延迟，以便单独衡量替换机制本身的开销。

实验设置 我们以融合加法 RMSNorm 内核（隐藏层大小为 4096）作为代表性案例评估系统，该内核部署在使用 SGLang 服务 Llama-3.1- 8B-Instruct 的环境中。对于每种配置，我们测量 (i) 来自基准测试轨迹的独立内核延迟（图 8，上）以及 (ii) 在不同批次大小/并发级别（1、16、64）下的端到端请求延迟。我们比较四种配置： **Original**，启用 apply() 的采用 FlashInfer 后端的原生 SGLang；`apply()` 已启用； **回退替换**，

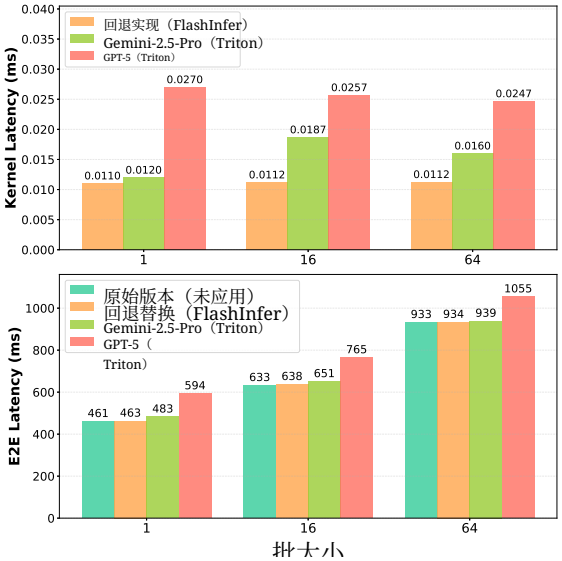

图 *8*。 内核延迟与端到端延迟对比。（上）三种实现的融合加法 RMS 归一化 h4096 在批次大小为 1、16 和 64 时的表现。（下）原始基准实现与不同内核替换机制的端到端请求延迟对比。所有测量值均以毫秒为单位；数值越低越好。

采用基准实现的 FlashInfer 实现，但替换为 apply()；**Gemini-2.5-Pro(Triton)**，比基准实现更快的生成内核；以及 **GPT-5 (Triton)**，比基准实现更慢的生成内核。每次实验均先进行系统预热，然后使用相同的输入和输出长度，对 4 个请求的平均延迟进行测量，以确保读数稳定。

`apply()` 带来的开销极小。 我们的第一个实验隔离了`apply()`的固定开销：用相同的实现替换原生内核（回退替换）。性能分析显示，`apply()`每次内核调用会带来 1–2 微秒的开销。比较图 *8* 中原始和回退替换配置（下图）可见，所有批次大小下的端到端开销均低于 0.8%。

`apply()` 将内核性能提升转化为端到端延迟改善。 在确认开销极低后，我们接下来展示内核级改进如何转化为可测量的端到端收益。图 8 上图报告了单独测试的内核性能：FlashInfer 在批大小为 64 时达到 0.0112 ms，快于 Gemini-2.5-Pro Triton 内核（0.0160 ms），而 GPT-5 为 0.0247 ms。这些基准测试的收益也延续到了完整系统中：下图显示，使用 `apply()` 时，FlashInfer 回代、Gemini-2.5-pro 回代和 GPT-5 回代的端到端耗时分别为 934 ms、939 ms 和 1055 ms，与各自内核的性能表现一致。这表明内核的效率会直接转化为端到端效率。替换为性能更好的内核，就能实现更高的端到端效率。

5 相关工作

5.1 用于 CUDA 生成的大语言模型

大语言模型近期展现出了生成 GPU 内核的非凡能力，Kernel-Bench（Ouyang 等人，2025年）和 TritonBench（Li 等人，2025年）等基准测试对这些能力进行了系统评估。这些基准测试主要关注模型能力的评估。BackendBench（Saroufim 等人，2025年）研究了如何使用大语言模型在 PyTorch 中生成内核，并将内核集成到 PyTorch 中。相比之下，FlashInfer-Bench 聚焦于大语言模型系统中的主要工作负载，并提供了一个端到端生产系统，将评估、验证和部署整合到统一框架中。

近期的研究进展探索了训练大语言模型以及设计智能体来生成高效内核。Kevin（Baronio 等人，2025年）和 KernelLLM（Fisches 等人，2025年）开发了用于内核生成的后训练模型。Dong 等人（2025年）为内核转译任务设计了一个智能体，而 Wei 等人（2025年）研究了基于大语言模型的 CUDA 内核生成智能体设计。这些智能体和模型设计与 FlashInfer-Bench 互为补充，后者可以进一步评估这些智能体和模型在来自真实世界大语言模型系统的内核任务上的性能。

5.2 面向系统优化的机器学习

机器学习长期以来一直应用于编译器和系统优化。TVM (Chen 等人，2018年) 开创了张量程序自动优化，随后出现了 AutoTVM 和 Ansor (Zheng 等人，2020)，并引入了结合学习型成本模型的基于搜索的调优。近年来，Meta-Schedule (Shao 等人，2022年) 通过设计空间抽象和策略学习推广了这些方法。这些系统在由人工设计的模板所定义的固定优化空间内运行。

FlashInfer-Bench 采用基于生成的方法：大语言模型直接提出候选内核（可能超出任何预定义的调度空间），随后对其进行严格的功能验证和性能基准测试。这将有效前沿从在固定模板集合内搜索，拓展到合成全新的实现模式。

5.3 内核库与自定义内核领域特定语言

高度优化的内核库，例如 CUTLASS（Thakkar 等人，2023年）、cuBLAS（NVI，2025年）和 Flash-

Infer（Ye 等人，2025年），为大语言模型系统中的内核提供了强大的基准实现。Triton（Tillet 等人，2019年）等领域特定语言降低了编写自定义 GPU 内核的门槛。FlashInfer-Bench 将这些生态系统视为智能体的实现目标，并着重于将候选内核接入生产系统。

多项研究专注于推理工作负载。FlashAttention 引入了感知 IO 的精确注意力，大幅减少内存流量并提高吞吐量（Dao 等人，2022年）。多查询注意力通过在多个注意力头之间共享键/值，减少 KV 缓存占用并降低解码延迟（Shazeer，2019年）。服务引擎采用分页式 KV 缓存和调度策略（例如 Page-dAttention），以便在不规则的多租户工作负载下保持较高利用率（Kwon 等人，2023年）。FlashInfer-Bench 通过以真实追踪数据（形状分布、缓存布局）为基础设计任务，并评估对正确部署至关重要的数值特性和批处理特性，捕捉了这些实际情况。

5.4 大语言模型推理系统

vLLM（Kwon 等人，2023年）、SGLang（Zheng 等人，2025年）、TensorRT-LLM（英伟达，2025年）和 MLC-LLM（MLC 团队，2023—2025年）等框架，展示了面向大模型的可扩展推理基础设施。这些系统为 FlashInfer-Bench 的内核选择提供指导——优先处理注意力和混合专家等现代大语言模型操作，而非卷积等传统操作——并提供参考实现。反过来，FlashInfer-Bench 使这些框架能够快速评估并将优化后的内核部署到生产环境中，从而构建互惠互利的生态系统。

6 结论

我们介绍了 FlashInfer-Bench，这是一种系统化方法，能够打通从人工智能内核生成到生产环境影响的完整闭环。FlashInfer Trace 模式的核心是对算子契约、真实服务工作负载、候选实现和不可变评估进行标准化。在此基础上，我们的基准测试涵盖确定性、低精度和采样内核，并可通过 apply() 将最佳已验证内核替换到 SGLang 和 vLLM 等引擎中，无需更改任何代码。

作为框架的补充，我们的实时排行榜持续追踪前沿模型在真实世界和大语言模型工作负载上的 GPU 编程能力。评估得出了三项实用结论：（1）编译是主要故障模式；（2）模型难以充分利用硬件特性；（3）语言选择是一种权衡——Triton 的正确性和易用性较高，而 CUDA 在成功时可实现更高的峰值性能。端到端来看，动态替换几乎不会增加开销，并且能可靠地将内核级性能提升转化为大语言模型服务中更低的延迟和更高的吞吐量。

局限性与后续工作：目前，我们的工作范围尚未涵盖多 GPU 或通信内核，支持的模型、硬件设备和编程语言种类也有限。未来可以进一步拓展 FlashInfer Trace 数据集的广度，改进内核正确性验证，以防止奖励操纵并确保基准测试结果可靠；还可以基于 FlashInfer-Bench 反馈循环，为大语言模型系统开发内核智能体和微调模型。

致谢

本研究部分得到了 Bosch 的支持以及英伟达和谷歌的捐赠，我们也感谢英伟达提供 DGX B200 支持。我们还要按字母顺序感谢 Databricks、FlashInfer 团队、GPUMODE 团队、HuggingFace 团队、SGLang 团队、TensorRT-LLM 团队、vLLM 团队和 xAI，以及 Zhuoming Chen、Weihua Du、Bohan Hou、Hongyi Jin、Ruihang Lai、Mark Saroufim、Xinyu Yang、Yilong Zhao 和 Haizhong Zheng，感谢他们提供的宝贵反馈。

参考文献

Baronio, C., Marsella, P., Pan, B., Guo, S., and Alberti, S. Kevin：用于生成 CUDA 内核的多轮强化学习，2025年。网址 https://arxiv.org/abs/2507.11948。

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM：面向深度学习的自动化端到端优化编译器。载于第*13*届 *USENIX* 操作系统设计与实现研讨会（*OSDI*）论文集，美国加利福尼亚州卡尔斯巴德，2018年。USENIX 协会。

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and R´e, C. FlashAttention：具备 I/O 感知能力的快速且节省内存的精确注意力。载于第*36*届神经信息处理系统国际会议论文集，NIPS '22，美国纽约州雷德胡克，2022年。Curran Associates Inc. ISBN 9781713871088。

Dong, S., Wen, Y., Bi, J., Huang, D., Guo, J., Xu, J., Xu, R., Song, X., Hao, Y., Zhou, X., Chen, T., Guo, Q., 和 Chen, Y. Qimeng-Xpiler：采用神经符号方法为深度学习系统转译张量程序，2025年。URLhttps://arxiv.org/abs/2505.02146。已被 OSDI 2025 接收。

Fisches, Z.V.、Paliskara, S.、Guo, S.、Zhang, A.、Spisak, J.、Cummins, C.、Leather, H.、Synnaeve, G.、Isaac-

son, J.、Markosyan, A. 和 Saroufim, M. KernelLLM：让内核开发更易于使用，2025 年 6 月。网址 https://huggingface.co/facebook/KernelLLM。通讯作者：AramMarkosyan、Mark Saroufim。

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. 利用分页注意力为大语言模型服务实现高效内存管理。在 第*29*届操作系统原理研讨会论文集，SOSP '23，第611–626页，纽约，NY，美国，2023年。计算机协会。ISBN 9798400702297。doi：10.1145/3600006.3613165。网址 https://doi.org/10.1145/3600006.3613165。Li, J., Li, S., Gao, Z., Shi, Q., Li, Y., Wang, Z., Huang, J., Wang, H., Wang, J., Han, X., Liu, Z., and Sun, M. TritonBench：评测大语言模型生成Triton算子的能力，2025年。网址https://arxiv.org/abs/2502.14752。MLC 团队。MLC-LLM，2023—2025年。网址 https://github.com/mlc-ai/mlc-llm。*cuBLAS*库。英伟达，2025年10月。网址 https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf。CUDA 工具包文档。英伟达。TensorRT-LLM，2025年。网址 https://github.com/NVIDIA/TensorRT-LLM。GitHub 仓库。Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., R´e, C., and Mirhoseini, A. KernelBench：大语言模型能编写高效的GPU内核吗？，2025年。网址 https://arxiv.org/abs/2502.10517。Saroufim, M., Wang, J., Maher, B., Paliskara, S., Wang, L., Sefati, S., and Candales, M. BackendBench：用于评估大语言模型和人类编写PyTorch后端能力的评测套件，2025年。网址 https://github.com/meta-pytorch/BackendBench。Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. FlashAttention-3：利用异步性和低精度实现快速、准确的注意力。在第*38*届神经信息处理系统国际会议论文集，NIPS '24，纽约州雷德胡克，美国，2024年。Curran Associates Inc. Shao, J., Zhou, X., Feng, S., Hou, B., Lai, R., Jin, H., Lin, W., Masuda, M., Yu, C. H., and Chen, T. 利用概率程序进行张量程序优化，2022年。网址https://arxiv.org/abs/2205.13603。Shazeer, N. 快速Transformer解码：一个写入头便足矣，2019年。网址 https://arxiv.org/abs/1911.02150。Thakkar, V.、Ramani, P.、Cecka, C.、Shivam, A.、Lu, H.、Yan, E.、Kosaian, J.、Hoemmen, M.、Wu, H.、Kerr, A.、Nicely, M.、Merrill, D.、Blasig, D.、Qiao, F.、Majcher, P.、Springer, P.、Hohnerbach, M.、Wang, J. 和 Gupta, M. CUTLASS，2023年1月。URL https://github.com/NVIDIA/cutlass。Tillet, P.、Kung, H. T. 和 Cox, D. Triton：一种用于分块神经网络计算的中间语言和编译器。载于第*3*届 *ACM SIGPLAN* 机器学习与编程语言国际研讨会论文集，MAPL 2019，第10–19页，美国纽约州纽约市，2019年。计算机协会。ISBN 9781450367196。doi：10. 1145/3315508.3329973。URLhttps://doi.org/10.1145/3315508.3329973。Wei, A.、Sun, T.、Seenichamy, Y.、Song, H.、Ouyang, A.、Mirhoseini, A.、Wang, K. 和 Aiken, A. Astra：用于 GPU 内核性能优化的多智能体系统，2025年。URLhttps://arxiv.org/abs/2509.07506。Ye, Z.、Chen, L.、Lai, R.、Lin, W.、Zhang, Y.、Wang, S.、Chen, T.、Kasikci, B.、Grover, V.、Krishnamurthy, A. 和 Ceze, L. FlashInfer：用于大语言模型推理服务的高效且可定制的注意力引擎。*arXiv* 预印本 *arXiv:2501.01005*，2025年。URL https://arxiv.org/abs/2501.01005。Zheng, L.、Jia, C.、Sun, M.、Wu, Z.、Yu, C. H.、Haj-Ali, A.、Wang, Y.、Yang, J.、Zhuo, D.、Sen, K.、Gonzalez, J. E. 和 Stoica, I. Ansor：为深度学习生成高性能张量程序。载于 第*14*届 *USENIX*操作系统设计与实现会议论文集，OSDI'20，美国，2020年。USENIX 协会。ISBN 978-1-939133-19-9。Zheng, L.、Yin, L.、Xie, Z.、Sun, C.、Huang, J.、Yu, C. H.、Cao, S.、Kozyrakis, C.、Stoica, I.、Gonzalez, J. E.、Bar- rett, C. 和 Sheng, Y. SGLang：结构化语言模型程序的高效执行。载于 第*38*届神经信息处理系统国际会议论文集，NIPS '24，美国纽约州雷德胡克，2025年。Curran Associates Inc. ISBN 9798331314385。

A FlashInfer 跟踪示例

本附录提供了 FlashInfer Trace 格式的具体示例。

A.1 通用矩阵乘法

我们先从通用矩阵乘法内核的跟踪定义开始。

```
{
  "名称": "通
  "描述":"通用矩阵乘
  "操作_类型": "通用
  "标签": [
    "状态:已验证",
    "模型:Qwen3-30B-A3B"
  ],
  "轴": {
    "M": { "类型": "变量" },
    "N": { "类型": "常量", "值": 128 },
    "键": { "类型": "常量", "值": 2048 }
  },
  "输入": {
    "A": { "形状": [ "键"], "数据类型": "半精度浮点数" },
    "B": { "形状": [ "键"], "数据类型": "半精度浮点数" }
  },
  "输出": {
    "C": { "形状": [ "N"], "数据类型": "半精度浮点数" }
  },
  "参考":
  ↪ "导入 PyTorch\n\ndef run(A, B):\n C = PyTorch 矩阵乘法(A, B.T)\n 返回 C"
}
```

对应的生成的解决方案对象可能如下所示：

```
{
  "名称": "
  "定义": "通用矩阵
  "作者": "claude-opus-4-1-20250805",
  "规格": {
    "语言": "triton",
    "目标_硬件": [
      "B200"
    ],
    "入口_点": "main.py::run",
    "依赖项": []
  },
  "源文件": [
    {
      "路径": "main.py",
      "内容":"<源代
    }
  ],
  "描述":
  ↪ "claude-opus-4-1-20250805 用于通用矩阵乘法_n128_k2048 的优化内核（第1轮）"
}
```

一个实例化此定义的工作负载示例如下：


```
{
  "定义": "通用矩阵
  "解决方案": null ,
  "工作负载": {
    "通用唯一识别码": "6ba7c7de-dc5a-48d2-8ada-1382feb5ceac",
    "轴": { "M": 6 },
    "输入": {
      "甲": { "类型": "随机" },
      "B": { "类型": "随机" }
    }
  },
  "评估": null
}
```


在 NVIDIA B200 图形处理器上对此工作负载评估该解决方案，得到以下跟踪记录：


```
{
  "定义": "gemm_
  "工作负载": {
    "轴": { "M": 6 },
    "输入": {
      "甲": { "类型": "随机" },
      "B": { "类型": "随机" }
    }
    "通用唯一识别码": "6ba7c7de-dc5a-48d2-8ada-1382feb5ceac"
  },
  "解决方案": "claude_cpus_4_1_20250805_triton_e20e42"
  "评估": {
    "状态": "通过",
    "环境": {
      "硬件": "NVIDIA B200",
      "库": {
        "PyTorch": "2.8.0+cu128",
        "Triton": "3.4.0",
        "CUDA": "12.8"
      }
    },
    "时间戳": "2025-10-16T01:10:32.241021",
    "对数": "",
    "正确性": {
      "max_相对_错误": 0 ,
      "max_绝对_错误": 0 ,
      "额外": null
    },
    "性能": {
      "延迟_毫秒": 0.023046740692633086,
      "参考延迟_毫秒_毫秒": 0.025240250456929125,
      "加速比_因子": 1.0951765715399921
    }
  }
}
```

A.2 注意力

接下来，我们将介绍一个基于分页分组查询注意力解码算子的更复杂示例。与通用矩阵乘法案例相比，该算子具有更复杂的接口，以及一个非平凡的参考实现。


```
{
  「名称」：
  「描述」：「使用分
  「操作_类型」：「
  「标签」：[
    "阶段:解码"
    "状态:已验证"
    "模型:qwen3-30b-a3b"
  ],
  "轴": {
    "批次_大小": {
      "类型": "var",
      "描述":"查询词元总
    },
    "num_qo_注意力头": {
      "类型": "常量",
      "值": 32
    },
    "num_kv_注意力头": {
      "类型": "常量",
      "值": 4
    },
    "注意力头_维度": {
      "类型": "常量",
      "值": 128
    },
    "页数_": {
      "类型": "变量"
    },
    "页_大小": {
      "类型": "常量",
      "值": 1
    },
    "长度_indptr": {
      "类型": "var",
      "描述":"kv indptr
    },
    "num_kv_索引": {
      "类型": "var",
      "描述":"KV 页面索
    }
  },
  "约束": [
    "长度_indptr == 批次_大小 + 1",
    "数量_kv_索引 == kv_indptr[-1].item()"
  ],
  "输入": {
    "q": {
      "形状": ["
      "数据类型": "bfloat16"
    },
    "键_缓存": {
      “形状”: [“
      “数据类型”: “bfloat16”
    },
    "值_缓存": {
      "形状": ["
      "数据类型": "bfloat16"
    },
```

"kv_indptr": { "shape": ["len_indptr"],"dtype": "32位整数", "description": "每个序列的 KV 页偏移量。 s": { "shape": ["num_kv_indices"],"dtype": "32位整数", "description": "用于 KV 缓存查找的页 ID。" }, "sm_scale": { "shape": null, "dtype": "单精度浮点数", "description": "Softmax 缩放因子。默认为 (1/sqrt(head_dim))。" } }, "outputs": { "output": { "shape": ["batch_size", "num_qo_heads", "head_dim"],"dtype": "bf ise . { shape": ["batch_size", "num_qo_heads"],"dtype": "单精度浮点数", "de g-sum-exp。" } }, "reference":"<下面展示的参考代码>" }

对应的 PyTorch 参考实现如下：

```
导入 PyTorch
导入 数学


@PyTorch。无_grad()
定义运行(查询, 键_缓存, 值_缓存, kv_indptr, kv_索引, sm_缩放):
    批次_大小, 数量_qo_注意力头, 注意力头_维度 = 查询。形状
    _, 页_大小, 数量_kv_注意力头, _ = k_缓存。形状
    长度_indptr = kv_indptr。形状[0]
    数量_kv_索引 = kv_索引。形状[0]

    # 检查常量
    断言 数量_qo_注意力头 == 32
    断言 数量_kv_注意力头 == 4
    断言 注意力头_维度 == 128
    断言 页_大小 == 1

    # 检查约束
    断言 长度_indptr == 批次_大小 + 1
    断言 数量_kv_索引 == kv_indptr[-1].item()

    设备 = 查询.设备

    输出 = PyTorch.zeros(
        (批次_大小, 数量_qo_注意力头, 注意力头_维度), 数据类型=PyTorch.bfloat16, 设备=设备
    )
    lse = PyTorch。full(
        （批次_大小， 数量_查询_注意力头），-浮点数("inf")，数据类型=PyTorch.单精度浮点数,
        ↪ 设备=设备
    )
```

```
分组查询注意力_比率 = 数量_查询_注意力头 // 数量_键值_注意力头

键_缓存_扁平化 = 键_缓存。squeeze(1)。to(
    PyTorch。单精度浮点数
)  # [数量_页, 数量_键值_注意力头, 注意力头_维度]
值_缓存_扁平化 = 值_缓存。squeeze(1)。to(
    PyTorch.单精度浮点数
)  # [num_页, num_kv_注意力头, 注意力头_维度]

用于b 在 范围(批次_大小):
    页_开始 =整数(kv_指针[b].项())
    页_结束 = 整数(kv_指针[b + 1].项())

    如果 页_开始 >= 页_结束:
        # 此批次元素没有 KV 缓存
        输出[b].零_()
        继续

    # 页是页_大小=1的令牌索引
    令牌_索引 = kv_索引[页_起始: 页_结束]。转为(torch.长整型)
    # 令牌数就是页_大小=1对应的页数
    数量_令牌 =令牌_索引。形状[0]

    如果 数量_令牌 == 0:
        输出[b]。零_()
        继续

    # 获取查询, K, 此批次的值
    键_批次 = 键_缓存_平坦[令牌_索引]                # [令牌数_, num_kv_注意力头,
    ↪ 注意力头_维度]
    值_批次 = 值_缓存_平坦[令牌_索引]                # [num_令牌, num_kv_注意力头,
    ↪ 注意力头_维度]
    查询_批次 = 查询[b].转为(PyTorch.单精度浮点数)  # [num_qo_注意力头, 注意力头_维度]

    对于 h 在 范围(num_qo_注意力头):
        # 为分组查询注意力查找对应的键值头
        键值_头 = h// 分组查询注意力_比率

        查询_头 = 查询_批次[h]  # [头_维度]
        键_注意力头 = 键_批次[:, kv_注意力头]  # [数量_令牌, 注意力头_维度]
        值_注意力头 = 值_批次[:, kv_注意力头]  # [数量_令牌, 注意力头_维度]

        逻辑值 = PyTorch.矩阵乘法(查询_注意力头, k_注意力头。T)  # [令牌数_令牌]
        逻辑值_缩放后的 = 逻辑值* sm_scale

        # 计算以 2 为底的 LSE
        lse[b, h] =PyTorch.对数和指数(逻辑值_scaled, 维度=-1) / 数学.对数(2.0)

        注意力 = PyTorch.Softmax(逻辑值_scaled, 维度=-1)  # [数量_令牌]
        输出_注意力头 = PyTorch。矩阵乘法(attn, v_注意力头)  # [注意力头_维度]
        输出[b, h] = 输出_注意力头。转为(torch。bfloat16)

返回 输出, lse
```

一种可能生成的 Triton 解决方案可表示如下：


```
{
  "名称": "claude-opus-4-1-triton-de54a2",
  "定义": "分组查询
  "描述":"claude-opu
  "作者":     "claude-opus-4-1-20250805",
  "规格":   {
    "语言":         "triton",
    "目标_硬件":   [
      "B200"
    ],
    "入口_点":         "main.py::run",
    "依赖项":   []
  },
  "源文件":     [
    {
      "路径":    "main.py",
      "内容":"<源代
    }
  ]
}
```


与通用矩阵乘法的情况一样，我们可以为此定义捕获一个具体的工作负载实例。在这个示例中，我们有一个标量输入，以及一些从 safetensors 转储文件中加载的其他输入：


```
{
  "定义": "gqa_paged_decode_h32_kv4_d128_ps1",
  "解决方案": null  ,
  "工作负载": {
    "通用唯一识别码":     "0c2489b2-f878-428b-b1bd-d0c6d4c39338"      ,
    "轴":     {
      "批次_大小": 1  ,
      "页_数": 8,
      "len_indptr": 2,
      "num_kv_索引": 7
    },
    "输入": {
      "查询": {    "类型":    "随机"     },
      "k_缓存": {    "类型":    "随机"     },
      "v_缓存": {    "类型":    "随机"     },
      "键值_indptr": {
        "类型":    "safetensors",
        "路径":    "/path/to/safetensor",
        "张量_键": "kv_
      },
      "kv_索引": {
        "类型":    "safetensors",
        "路径":    "/path/to/safetensor",
        "张量_键": "kv_
      },
      "sm_比例": {
        "类型":    "标量",
        "值": 0.08838834613561630
      }
    }
  },
  "评估": null
}
```

在此工作负载上评估上述解决方案，会生成以下跟踪记录：


```
{
  "定义": "分组查询
  "工作负载": {
    "轴":     {
      "批次_大小": 1  ,
      "页数_": 8,
      "len_indptr": 2,
      "num_kv_索引": 7
    },
    "输入": {
      "查询": {    "类型":    "随机"      },
      "k_缓存": {    "类型":    "随机"      },
      "v_缓存": {    "类型":    "随机"      },
      "键_indptr": {
        "类型":    "safetensors",
        "路径":    "/path/to/safetensor",
        "张量_键": "kv_
      },
      "kv_索引": {
        "类型":    "safetensors",
        "路径":    "/path/to/safetensor",
        "张量_键": "kv_
      },
      "sm_缩放": {
        "类型":    "标量",
        "值": 0.08838834461356163
      }
    }
    "通用唯一识别码": "0c2489b2-f878-428b-b1bd-d0c6d4c39338"
  },
  "解决方案": "c
  "评估": {
    "状态":     "PASSED" ,
    "环境": {
      "硬件":"NVIDI
      "库":     {
        "PyTorch":  "2.8.0+cu128"  ,
        "Triton":    "3.4.0"  ,
        "CUDA":  "12.8"
      }
    },
    "时间戳":       "2025年10月16日T01:24:16.694452",
    "对数":  "",
    "正确性": {
      "max_相对      _错误": 0.01480561401695013  ,
      "max_绝对      _错误": 0.00048828125  ,
      "extra": null
    },
    "性能": {
      "延迟_毫秒": 0.02266162589486805,
      "参考_延迟_毫秒": 29.439284915015815,
      "加速比_因子": 1299.0808802329861
    }
  }
}
```

B 智能体生成的内核

本附录包含案例研究中讨论的智能体生成内核的完整源代码。

B.1 Triton GEMMKernel

- **语言**— Triton。
- **源代码**— 由 GPT-5 根据通用矩阵乘法 n4096 k4096 定义生成。
- **正确性**— 在所有评估的工作负载上均正确。
- **性能（相较于基准实现 `torch.nn.functional`）**— 在所有工作负载上均实现 $0.2\times$ 加速比，最佳情况下为 $0.6\times$，适用于其中一个工作负载。
- **吞吐量（每秒万亿次浮点运算）**— 对于一个问题（M，N=4096，K=4096），其实际运行时间为 t 秒，

$$\text{TFLOPs} = \frac{2MNK}{t \cdot 10^{12}}.$$

```
导入math导入
PyTorch导入
Triton
导入triton.language 作为tl


@triton.autotune( configs=[triton.Config({'BLOCK_M': 128, 'BLOCK_N': 128, 'BLOCK_K': 64}, num_
warps=8, num_stages=4), ↪triton.Config({'BLOCK_M': 64, 'BLOCK_N': 256, 'BLOCK_K': 64}, num_warps=8,
num_stages=4), ↪triton.Config({'BLOCK_M': 256, 'BLOCK_N': 64,'BLOCK_K': 64}, num_warps=8, num_
stages=4), ↪triton.Config({'BL[illegible]28, 'BLO[illegible] 'BLOCK_[illegible]um_warps=8, num_
stages=4), ↪] ,键=['M'] ,) @triton.jit



def _gemm_n4096_k4096_kernel( A_ptr, B_ptr, C_ptr, M, N, K, stride_am, stride_ak, stride_bn, stride_
bk, stride_cm, stride_cn, BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr, BLOCK_K: tl.constexpr, ):
tl.static_assert(BLOCK_K % 16 == 0, "BLOCK_K 必须是 16 的倍数，才能使用张量核心") ↪pid_m =
tl.program_id(0) pid_n = tl.program_id(1) offs_m = pid_m *BLOCK_M + tl.arange(0, BLOCK_M)
offs_n = pid_n *BLOCK_N + tl.arange(0, BLOCK_N) acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=
tl.float32) M_mask = offs_m[:, None] < M N_mask = offs_n[None, :] < N for k0 in range(0, K,
BLOCK_K): offs_k = k0 + tl.arange(0, BLOCK_K) # 指针 a_ptrs = A_ptr + (offs_m[:, None] * stri
de_am + [illegible]
```

```
        b_指针数组 = B_指针 + (偏移量_n[:, 无] *步幅_bn + 偏移量_k[无, :] * 步幅_bk)

        a = tl.加载(a_指针数组, 掩码=M_掩码 & (偏移量_k[无, :] < 键), 其他=0.0)
        b = tl.load(b_ptrs, 掩码=(offs_n[:, 无] < N) &(偏移量_键[无, :] < 键),
        ↪ 其他=0.0)

        累加器 += tl.点积(a, tl.trans(b))

    c_ptrs = C_ptr + (offs_m[:, None]* 步幅_cm + offs_n[None, :] * 步幅_cn)
    tl.store(c_ptrs, acc.to(tl.半精度浮点数), 掩码=M_掩码& N_掩码)


def 运行(A, B, **关键字参数):
    """
    计算 C = A @ B.T, 其中:
    - A: [M, 4096] 半精度浮点数
    - B: [4096, 4096] 半精度浮点数
    返回 C: [M, 4096] 半精度浮点数
    """
    如果 不是 isinstance(A, PyTorch.Tensor) 或不是 isinstance(B, PyTorch.Tensor):
        引发 类型错误("A 和

    如果 A.维度 != 2 或 B.维度 != 2:
        引发
        ↪ 值错误(f"期望 B.维度={ .维度}",

    M, K_a = A.形状
    N_b, K_b = B.形状

    if K_一个 != 4096 或 键_b != 4096 或 N_b != 4096:
        引发 值错误(f"无效的形 B 为 {1.形状}」
        ↪ ; 预期的 A: [M,4096], B: [4096,4096]")

    # 数据类型检查/转换
    如果 A.数据类型 != PyTorch.半精度浮点数:
        A = A.转换(PyTorch.半精度浮点数)
    如果 B.数据类型 != PyTorch.半精度浮点数:
        B = B.转为(torch.半精度浮点数)

    # 设备管理
    原始_开发_A = A.设备
    原始_开发_B = B.设备
    任意_CUDA_输入 = (A.为_CUDA 或 B.为_CUDA)

    如果 未 PyTorch.CUDA.可用_():
        如果任意_CUDA_输入:
            引发运行时错误(」
            ↪ "CUDA 不可用，但一个或多个输入是 CUDA 张量。")
        # Triton 需要 CUDA；未提供 CPU 回退实现
        引发运行时错误("没有可用

    # 选择计算设备
    如果A。是_cuda:
        计算_设备 = A。设备
    否则如果 B.是_cuda:
        计算_设备 = B.设备
    否则:计算_设备 = PyTorch。device(f"cuda:{torch.cuda.current_device()}")

    # 移至计算设备并转换为连续格式，以实现最佳访问性能A_开发 = A.to(device=计算_设备, non_
    blocking=True).contiguous()
    B_开发 = B.to(device=计算_设备, non_blocking=True).contiguous()
```

```
    # 在计算设备上分配输出
    N = 4096
    键 = 4096
    C_开发 = PyTorch。empty((M, N),  数据类型=PyTorch。半精度浮点数, 设备=计算_设备)

    # 内核启动参数
    定义 网格(Meta):
        返回 (triton.cdiv(M , Meta['          , triton。cdiv(N , Meta['          )

    # 调用内核
    _通用矩阵乘法_n4096_k4096_内核[网格](
        A_dev, B_dev, C_dev,
        M, N, K,
        A_开发。步幅(0) , A_开发。步幅(1) ,
        B_开发。步幅(0) , B_开发。步幅(1) ,
        C_开发。步幅(0) , C_开发。步幅(1) ,
    )

    # 决定输出设备：保留原始位置；如果两者都是 CPU，则返回
    ↪ CPU；否则优先使用 A 的设备（如果是 CUDA），否则使用 B 的设备
    如果 原始_设备_A。类型 ==      ' 且 原始_设备_B。类型 =='c
        输出_设备 =PyTorch。设备('cpu
    否则如果 原始_开发_A。类型 =='
        输出_设备 =原始_开发_A
    否则如果 原始_开发_B。类型 =='
        输出_设备 =原始_开发_B
    否则:
        输出_设备 =PyTorch.设备('cpu

    C_输出 = C_开发 如果 C_开发.设备 == 输出_设备 否则C_开发.to(输出_设备 ,
    ↪ 非_阻塞=True)
    返回 C _out
```

B.2 CUDA GEMM 内核

- **语言**— CUDA。
- **来源**— 由 Gemini-2.5-Pro 自动生成，适用于 gemm n4096 k4096 定义。
- **实现方法** — 调用 cuBLAS 矩阵乘法。
- **正确性**— 在所有评估的工作负载上均正确。
- **性能（相较于基准实现 `torch.nn.functional`）**——所有工作负载上的加速比为 $\mathbf{0.97}\times$，其中一个工作负载的最佳加速比为 $\mathbf{1.03}\times$。

kernel.h：

```
#pragma once

// 包含保护宏
#ifndef GEMM _N4096 _K4096 _内核 _H
#define 通用矩阵乘法 _N4096 _K4096 _内核 _H

#include <cuda_fp16.h>
#include <cuda_runtime.h>

/**
 * @brief 通用矩阵乘法运算 C = A * B.T 的主机端启动器。
 *
 * 此函数会启动一个基于 cuBLAS 的 GEMM 内核，针对以下场景进行了优化：
 * B200 级 GPU，且 N=4096 和键=4096。它计算 C = A * B.T
 * 其中 A 为 [M , K]，B 为 [N , K]，而 C 为 [M , N]。所有张量均
 * 预期采用行主序布局。
 *
 * @param M 矩阵 A 和 C 的可变维度。
 * @param A 指向输入矩阵 A [M 的指针， 4096] 位于设备内存中（数据类型=half）。
 * @param B 指向输入矩阵 B [4096,的指针 4096] 位于设备内存中
 ↪ （数据类型=half）。
 * @param C 指向输出矩阵 C [M 的指针， 4096] 位于设备内存中（数据类型=half）。
 * @param stream 用于内核执行的 CUDA 流。
 */
void 通用矩阵乘法_n4096_键4096_启动器(
    整数 M ,
    常量 half*A ,
    常量 half*B ,
    一半* C ,
    CUDA 流 _t 流
);

#endif // 通用矩阵乘法 _N4096 _K4096 _内核 _H
```

kernel.cu：

```
#include "kernel.h"
#include <cublas_v2.h>
#include <cuda_fp16.h>

#include <stdexcept>
#include  <字符串>

// 用于检查 cuBLAS 状态并将其转换为描述性错误的辅助函数
#define CUBLAS_CHECK(status)                                                    \
  do {                                                                          \
    cublasStatus_t 错误 = (状态);                                               \
    if (err != CUBLAS_STATUS_SUCCESS) {                                         \
      throw std::runtime_error("cuBLAS 错误： " + std::to_string(err) \
                               + “ 位于 ” 文件 ”： ” +std::to_字符串(__行__)); \
                                  __   __
```

```
    }                                                                    \
  } while (0)

// 为提高效率，管理一个 cuBLAS 句柄单例。
// 这样可以避免每次调用时创建和销毁句柄的开销。
cublas 句柄_t 获取_cublas_句柄() {
    静态结构体 CublasHandle {
        cublasHandle_t handle;
        CublasHandle() {
            CUBLAS_CHECK(cublasCreate(&handle));
            // 在 B200 上启用张量核心运算，以实现峰值性能
            CUBLAS_CHECK(cublasSetMathMode(handle, CUBLAS_张量_OP_MATH));
        }
        ~CublasHandle() {
            如果 (handle) cublasDestroy(handle);

        }
    } 单例_handle;
    返回 单例_handle.handle;
}

/**
 * @brief 使用 cuBLAS 的主机端启动器实现。
 */
void 通用矩阵乘法_n4096_键4096_启动器(
    整数 M ,
    常量 half*A ,
    常量 half*B ,
    一半* C ,
    CUDA 流 _t stream
) {
    cublasHandle_t 句柄 = get_cublas_handle();
    CUBLAS_CHECK(设置 cuBLAS 流(handle, stream));

    常量 int N = 4096;
    常量 int 键 = 4096;

    常量 float alpha = 1.0f;
    常量 float beta = 0.0f;

    // 将 cuBLAS（列主序）与行主序 PyTorch 张量配合使用的关键在于
    // 将运算重新表述为 cuBLAS 能够理解的形式，并使其产生
    // 正确内存布局的输出。
    //
    // 1.  目标（行主序）：C_rm[M, N] = A_rm[M, 键] * B_rm.T[键, N]
    //
    // 2. cuBLAS 视图（列主序）：cuBLAS 将行主序矩阵的内存解释为
    //    矩阵 X_rm[行、列] 的列主序矩阵 X_cm[列、行]。
    //    - A_rm[M , K] 被视为 A_cm[K , M].
    //    - B_rm[N , K] 被视为 B_cm[K , N].
    //    - C_rm[M , N] 可视为 C_cm[N , M].
    //
    // 3. 转换：方程 C_rm = A_rm * B_rm.T 等价于
    //    C_cm.T = A_cm.T * (B_cm.T).T => C_cm.T = A_cm.T * B_cm.
    //    将整个方程转置后，我们得到 cuBLAS 应该
    ↪  计算：
    //    C_cm = (A_cm.T * B_cm).T = B_cm.T * A_cm.
    //
    // 4. cuBLAS 调用：我们要求 cuBLAS 计算 D = 操作1 * 操作2，其中结果 D
    //    写入 C 的内存中。
    //    - op1 = B_cm.T。这意味着第一个矩阵是 B，且 transa=CUBLAS_OP_T。
    //    - op2 = A_cm。 这意味着第二个矩阵是 A，且 transb=CUBLAS_OP_N。
    //
    // 5. cuBLAS 的维度：
```

```
    //     - m = 行操作1（B.T） = N
    //     - n = 列操作2（A）          = M
    //     - k = 公共维度              = K
    //     输出矩阵将为 [m, n] = [N,                    M]采用列主序布局，其中
    //     与我们期望的行主序 C_rm[M, N]内存布局完全匹配。
    //     这解决了非法内存访问问题，并确保了正确性。
    常量 整数 A 的前导维度 = K; // A 的前导维度_行主序[M, K] 为 K
    常量 整数 B 的前导维度 = K; // B 的前导维度_行主序[N, K] 为 K
    常量 整数 C 的主维度 = N; // C 的前导维度_行主序[M, N] 为 N

    CUBLAS_CHECK(cublasGemmEx(
        句柄 ,
        CUBLAS_OP_T,            // transa：对应第一个矩阵（B），转置
        CUBLAS_OP_N,            // transb：对应第二个矩阵（A），未
        ↪ 转置
        N,                      // m：op(B.T) 的行
        M,                      // n：op(A) 的列
        K,                      // k：公共维度
        &alpha,                 // alpha
        B,                      // 指向第一个矩阵（B）的指针
        CUDA_R_16F,             // B 类型
        ldb,                    // ldb（B 的前导维度）
        A,                      // 指向第二个矩阵（A）的指针
        CUDA_R_16F,             // A 类型
        lda,                    // A 的前导维度（A 的前导维度）
        &beta,                  // beta
        C,                      // 指向 C 的指针
        CUDA_R_16F,             // C 类型
        ldc,                    // C 的主维度（C 的前导维度）
        CUDA_R_32F,             // 计算类型：使用 FP32 累加器以提高精度
        CUBLAS_通用矩阵乘法_默认_张量_操作 // 算法：对张量使用默认启发式算法
        ↪ 核心
    ));
}
```

main.cpp：

```
#include <torch/extension.h>#include <c10/cuda/CUDAStream.h>#include "kernel.h" #include <
stdexcept>#include <string>// 用于简洁验证张量的辅助宏 #define CHECK_CUDA(x) TORCH_CHECK(x.is_
cuda(), #x " 必须是 CUDA 张量") #define CHECK_CONTIGUOUS(x) TORCH_CHECK(x.is_contiguous(), #x " 必
须是连续张量") #define CHECK_HALF(x) TORCH_CHECK(x.scalar_type() ==torch::kFloat16, #x " 必须是半
精度浮点数张量") ↪/** * @brief 通用矩阵乘法运算的 PyTorch 扩展入口点。 * * 此函数会验证输入张量，并调用
CUDA 内核启动器，* 在 GPU 上执行计算 C = A* B.T。 * * @param A 形状为 [M, 4096] 且数据类型为半
精度浮点数的 torch::Tensor。 * @param B 形状为 [4096, 4096] 且数据类型为半精度浮点数的
torch::Tensor。 * @return 形状为 [M, 4096] 且数据类型为半精度浮点数的 torch::Tensor C，其中包含计算
结果。↪*/
```

```
torch::Tensor run(torch::Tensor A, torch::TensorB) { // --- 输入验证 --- CHECK_CUDA(A); CHECK_
CUDA(B); CHECK_CONTIGUOUS(A); CHECK_CONTIGUOUS(B); CHECK_HALF(A); CHECK_HALF(B);
TORCH_CHECK(A.dim() == 2, "A 必须是二维张量"); TORCH_CHECK(B.dim() == 2, "B 必须是二维张量"); //
--- 维度检查 --- const int M = A.size(0); const int K_A = A.size(1); const int N_B = B.size(0); const
int K_B = B.size(1); const int N_spec = 4096; const int K_spec = 4096; TORCH_CHECK(K_A ==
K_spec, "A 的形状必须为 [M, 4096], ，但 K 为 ", K_A); TORCH_CHECK(N_B == N_spec, "B 的形状
必须为 [4096, 4096], ，但 N 为 ", N_B); TORCH_CHECK(K_B == K_spec, "B 的形状必须为
[4096, 4096], ，但 K 为 ", K_B); T                                    B.device(), "张量必须位于同一
CUDA 设备上"); ↪// --- 输出张量分配 --- auto C_options = torch::TensorOptions() .device(A.device()) .
dtype(A.scalar_type()); auto C = torch::empty({M, N_spec}, C_options); // --- 内核执行 --- try { // 从
PyTorch 上下文中获取当前 CUDA 流，以确保正确同步 ↪cudaStream_t stream =
at::cuda::getCurrentCUDAStream(); // 获取原始数据指针。at::Half 与 cuda_fp16.h::half 兼容 const half*
A_ptr =reinterpret_cast<const half*>(A.data_ptr<at::Half>()); const half* B_ptr =reinterpret_cast<
const half*>(B.data_ptr<at::Half>()); half* C_ptr =reinterpret_cast<half*>(C.data_ptr<at::Half>()); // 启
动基于 cuBLAS 的内核 gemm_n4096_k4096_launcher(M, A_ptr, B_ptr, C_ptr, stream); } catch (const
std::exception& e) { // 将 CUDA/cuBLAS 调用产生的异常传递给 Python throw std::runtime_
error(std::string("CUDA 内核执行失败：                                                    这是良好的实
践。↪cudaError_t err = cudaGetLastE
error(std::string("CUDA 异步错误：") +cudaGetErrorString(err)); ↪} return C; }
```

```
// 用于将“run”函数暴露给 Python 的 Pybind11 模块定义PYBIND11_MODULE(TORCH_EXTENSION_NAME、
m) {m.def("run"、 &run、用于 N=4096、K 的 GEMM（A * B.T）=4096 ， 使用针对 B200 优化的 cuBLAS
内核");↪}
```

B.3 Triton 分组查询注意力分页解码

- **语言**— Triton。
- **来源**—由 GPT-o3 根据分组查询注意力分页解码 h32 kv8 d128 ps1 规范生成。
- **正确性** —所有评估的工作负载均正确。
- **性能（相较于基准实现（FlashInfer））**—所有工作负载的加速比为 $0.19\times$，其中 3 个工作负载的最佳加速比为 $0.98\times$。

- **吞吐量（FLOPs/每秒万亿次浮点运算）**—在分组查询注意力（组大小为 H_q/H_{kv}）下，一个解码步骤的主要计算包括 $QK^\top$ 点积以及注意力 $-V$ 乘积（省略了阶数较低的 Softmax 运算）。已知该步骤的实际运行时间为 t 秒，

$$\text{TFLOPs} = \frac{\text{FLOPs}_{\text{step}}}{t \cdot 10^{12}} = \frac{4\, H_q\, d\, \texttt{num\_kv\_indices}}{t \cdot 10^{12}}.$$

```
导入数学
导入PyTorch
导入triton
导入triton.language 作为tl


@triton。jit
定义gqa_分页_解码_内核(
    查询_ptr,                  # *bf16    [B, 32, 128]
    k_ptr,                     # *bf16    [N_页,     8, 128]   (页_大小已压缩)
    v_指针,                    # *bf16    [N_页,     8, 128]   (页_大小已压缩)
    kv_indptr_指针,            # *int32  [B + 1]
    键值_索引_指针,            # *整数32  [数量_键值_索引]
    sm_比例,                   # FP32 标量
    输出_指针,                 # *bf16    [B, 32, 128]
    lse_指针,                  # *FP32    [B, 32]
    块_大小: tl.constexpr,
    注意力头_维度: tl.constexpr,
    数量_查询输出_注意力头: tl.constexpr,
    数量_键值_注意力头: tl.constexpr,
):
    pid = tl.program_id(0)

    批次_idx  = pid // NUM_QO_注意力头
    查询_注意力头 =pid%NUM_QO_注意力头
    分组查询注意力_比率 = 数量_查询输出_注意力头 // 数量_键值_注意力头
    kv_注意力头 =查询_head//gqa_ratio

    # ---- 步幅（以元素为单位，而非字节）----
    步幅_查询_批次          = 数量_查询输出_注意力头 *注意力头_维度
    步幅_查询_注意力头      = 头维度_

    步幅_键_页              = 键值头数量 *头维度_  _              _              # 页_大小 = 1
    步幅 _k_kv_head         = 注意力头 _DIM

    步幅_值_页              = 步幅_键_页
    步幅 _v_kv_head         = 注意力头 _DIM

    # ---- 加载查询向量 ----
    d_偏移 = tl.arange(0, 注意力头_维度)
    查询_指针_注意力头 = 查询_指针 + 批次_索引 * 步幅_查询_批次 + qo_注意力头* 步幅_查询_注意力头 +d_偏移→

    查询_向量 = tl.cast(tl.load(查询_指针_注意力头), tl.单精度浮点数)

    # ---- 序列令牌范围 ----开始 = tl.load(kv_indptr_ptr +
    批次_idx) 结束 =tl.load(kv_indptr_ptr +批次_idx + 1)
```

```
num_令牌 = 结束 - 开始

# ---- 流式 softmax 变量 ----
m 值_     = tl.full([], -1e30, tl.单精度浮点数)          # 运行最大值
d val_    = tl.zeros([], tl.单精度浮点数)                # 运行指数和
o_vec     = tl.zeros([HEAD_维度], tl.单精度浮点数)       # 正在运行的输出向量

偏移量 = tl.zeros([], tl.32位整数)

while 偏移量 < num_令牌:
    t 偏移量_     = tl.arange(0, 块_T)
    剩余量        = 数量_令牌 - 偏移量
    tok_mask      = t_偏移量 < 剩余量

    # ---- 加载页索引 ----
    页 = tl.加载(kv_索引_ptr + 起始 + 偏移量 + t_偏移量,
                 掩码=tok_mask, 其他=0)

    # ---- 收集键/值 ----
    键_指针 = 键_指针 + 页[:, None] * 步幅_键_页 + 键值_注意力头* 步幅_键_键值_注意力头
    ↪ + d_偏移量[无, :]
    值_指针 = 值_指针 + 页[:, None] * 步幅_值_页 + 键值_注意力头* 步幅_值_键值_注意力头
    ↪ + d_偏移[无, :]
    键_块 = tl.cast(tl.load(k_ptrs,
    ↪ 掩码=tok_掩码[:, 无], 其他=0),
    ↪ tl.单精度浮点数)
    v_块 = tl.cast(tl.load(v_ptrs, 掩码=tok_掩码[:, 无], 其他=0),
    ↪ tl.单精度浮点数)

    # ---- 逻辑值 ----
    逻辑值 = tl.sum(k_块* 查询_vec[None, :], axis=1) * sm_scale
    逻辑值 = tl.where(tok_mask, 逻辑值, -1e30)

    # ---- 块 Softmax ----
    m 块_       = tl.max(逻辑值, 轴=0)
    exp_逻辑值  = tl.exp(逻辑值 - m_块)
    sum_exp_块  = tl.sum(exp_逻辑值, 轴=0)
    加权_值     = tl.sum(exp_逻辑值[:, None]* 值_块, axis=0)

    # ---- 与运行值合并 ----
    new_m       = tl.maximum(m_值, m_块)
    alpha_前一个 =tl.exp(m_值 - 新_m)
    alpha_块    =tl.exp(m_块 - 新_m)

    o_向量 = o_向量* alpha_前一个 + 加权_值 * alpha_块
    d_值 = d_值* alpha_前一个 + 求和_exp_块 * alpha_块
    m_值 = 新_m

    偏移量 += 块_T

inv_d    = 逐元素。其中(d_值 == 0, 0.0, 1.0 / d_值)
out_向量 = o_vec * inv_d
log2e    = 1.4426950408889634
lse_值 = tl。where(d_值 == 0,
                  -1e30,
                  (tl。log(d_值) + m_值) *log2e)

# ---- 存储 ----
输出_指针_注意力头 = 输出_指针 + 批次_索引 * 步幅_查询_批次 + qo_注意力头* 步幅_查询_注意力头 +
↪ d_offs
tl.store(输出_ptr_注意力头, tl.cast(输出_vec, tl.bfloat16))

lse_ptr_注意力头 = lse_ptr + 批次_idx* NUM_QO_注意力头 + qo_注意力头
```

```
    tl.store(lse_ptr_注意力头, lse_val)


定义 运行(查询,
        k_缓存 ,
        v_缓存 ,
        键值_指针,
        kv_索引 ,
        sm_缩放因子: 浮点型 | 无 = 无):
    """
    分组查询注意力_分页_解码_h32_键值8_d128_ps1 的入口点。
    返回（输出, lse）。
    """
    如果 sm_缩放 为 无:
        sm_缩放 = 1.0 / 数学.平方根(128.0)

    如果 不是 PyTorch.cuda.is_available():
        抛出 RuntimeError("运

    # 如有需要，将张量移至 GPU
    张量 = [查询、键_缓存、值_缓存、键值_间接指针、键值_索引]
    设备_张量 = [t.cuda() 如果 不是 t.is_cuda 否则t 用于 t 在 张量]
    查询_开发设备、键_开发设备、值_开发设备、间接指针_开发设备、索引_开发设备 = [t.连续() 用于 t 在
    ↪ 设备_张量]

    批次_大小 = 查询_开发.形状[0]
    数量_查询输出_注意力头 = 32
    head_维度 = 128

    # 压缩页维度（=1）
    键_开发_扁平 = 键_开发.squeeze(1).连续()
    值_开发_扁平 = 值_开发.squeeze(1).连续()

    输出_开发 = torch.empty((批次_size, num_qo_注意力头, 注意力头_维度),
                            数据类型=torch.bfloat16,
                            设备=查询_开发。设备）
    lse_开发 = PyTorch。empty((批次_size, num_qo_注意力头),
                            数据类型=PyTorch。单精度浮点数,
                            设备=查询_开发。设备）

    # 启动内核
    块 _T = 128
    网格 =(批次_大小 * 数量_查询_注意力头,)

    分组查询注意力_分页_解码_内核[网格](
        查询_开发, 键_开发_平坦, 值_开发_平坦,
        iptr_开发, idx_开发,
        sm_缩放 ,
        out_dev, lse_dev,
        块 _T=块 _T,
        HEAD_维度=128,
        NUM_QO_注意力头=32 ,
        NUM_KV_注意力头=8,
        num_线程束=4,
        num_阶段=4,
    )

    # 如有需要，移回原始设备
    如果不查询。是_CUDA:
        返回输出_开发。CPU(), lse_开发。CPU()
    返回输出 _dev, lse_dev
```

B.4 CUDA GQA分页解码注意力

- **语言**— CUDA。
- **来源**— 由 GPT-5 根据分组查询注意力分页解码 h32 kv8 d128 ps1 规范生成。
- **实现方法** — 采用直接实现方式，每个块使用 4 个线程束，处理一个批次元素，并为每个块分配一个键值注意力头。该内核使用标量 FP32 运算实现流式 softmax。
- **正确性**— 在所有评估的工作负载中均正确。
- **性能（相较于基准实现（FlashInfer）**——在所有工作负载中，加速比为 0.02×，其中一个批大小为 1 的工作负载实现了最佳加速比 1.02× 。

```
#include "kernel.h"

#include <ATen/cuda/CUDAContext.h>
#include <CUDA_bf16.h>
#include <CUDA_运行时.h>
#include <数学_常量.h>
#include <数学函数头文件>
#include <数值限制头文件>
#include <C 标准输入输出头文件>

// 错误检查宏
#ifndef CUDA_CHECK
#define CUDA_检查(expr) do {                                                    \
    cudaError t__错误 = (表达式);                                               \
    if (__err !=cudaSuccess) {                                                  \
        printf("CUDA 错误 %s，位于 %s:%d\n", cudaGetErrorString(__err),         \
                __文件__，__行__);                                              \
    }                                                                           \
} while (0)
#endif

// 线程束工具
静态内联 __设备__ 浮点 线程束_归约_求和(浮点 值) {
    无符号 掩码 = 0xffffffffu;
    值 += __shfl_down_sync(掩码, 值, 16);
    值 += __shfl_down_sync(掩码, 值, 8);
    值 += __shfl_down_sync(掩码, 值, 4);
    值 += __shfl_down_sync(掩码, 值, 2);
    val += __shfl_down_sync(掩码, 值, 1);
    返回 val;
}

静态内联 __设备__ 单精度浮点数 线程束_广播（单精度浮点数 val，整数 src_lane）{
    无符号 掩码 = 0xffffffffu;
    返回 __shfl_sync(mask, val, src_lane）;
}

// 从 ptr [偏移量 .. 偏移量+3]以连续方式加载 4 个 BF16，并将其作为 4 个单精度浮点数
静态内联 __设备__ void 加载_bf16x4_到_f32(
    常量__nv_bfloat16* __restrict__ptr,

    int 偏移量 ,
    浮点数 输出[4]) {
    输出[0] = __bfloat162float(ptr[偏移量 + 0]);
    输出[1] = __bfloat162float(ptr[偏移量 + 1]);
    输出[2] = __bfloat162float(ptr[偏移量 + 2]);
    输出[3] = __bfloat162float(ptr[偏移量 + 3]);
}

// 将 4 个单精度浮点数以 BF16 格式存储到指针 [偏移量 .. 偏移量+3]静
态内联 __设备__ 无返回值 存储_f32x4_转为_bf16(__nv_
bfloat16* __受限__ptr,整数 偏移量,
```

常量 float in[4]) { ptr[偏移量 + 0] = __float2bfloat16(in[0]); ptr[偏移量 + 1] = __float2bfloat16(in[1]); ptr[偏移量 + 2] = __float2bfloat16(in[2]); ptr[偏移量 + 3] = __float2bfloat16(in[3]); } 模板 <整数 kBlockThreads>__启动_边界__(kBlockThreads, 2)__全局__ 无返回值 gqa_分页_解码_h32_kv8_d128_ps1_内核(常量__nv_bfloat16* __受限__查询, // [B, 32, 128]常量__nv_bfloat16* __受限__k_缓存, // [页数_, 1, 8, 128]->

flat
↪ [_ 8 128] 页数*, 常量__nv_bfloat16* __restrict__v_缓存, // [num_页, 1, 8, 128]->
flat
↪ [_ 8 128] 页数*, 常量 int32_t* __restrict__kv_indptr, // [B+1]常量 int32_t* __restrict__kv_indices, // [num_kv_indices]float sm_scale, __nv_bfloat16* __restrict__输出, // [B, 32, 128]float* __restrict__ lse_输出, // [B, 32]int num_批次, int num_页_total) { // 块映射: // 网格.x =批次索引 // 网格.y =kv_注意力头索引 in [0, 8) 常量 int b = blockIdx.x; 常量 int kv_ 注意力头 = blockIdx.y; // 0..7 if (b >= num_批次 || kv_注意力头 >= kNumKVHeads) { return; } // 线程映射: 常量 int tid = threadIdx.x; // 0..127 常量 int 线程束_id =tid >> 5; // 0..3（每个块 4 个线程束）常量 int lane_id =tid & 31; // 0..31 // 此键值头关联的 4 个查询头 常量 int 查询_注意力头 = kv_注意力头 * kGQARatio + 线程束_id; // 0..31 // 指针递增辅助项 常量 int 查询_步幅_h = kHeadDim; 常量 int 查询_步幅_注意力头 = kNumQOHeads * kHeadDim; // 此批次项的输入序列令牌范围 常量 int32_t 页_start = kv_indptr[b]; 常量 int32_t 页_end =kv_indptr[b + 1]; 常量 int32_t num_令牌 = 页_end - 页_start; // 此 kv_注意力头的单个令牌 K 和 V 向量的共享缓冲区 extern __共享__ float smem[]; float* sh_k =smem; // [128]float* sh_v = smem +kHeadDim; // [128]__共享__ int s_页; // 预加载 Q（每个线程束处理自己的查询_注意力头）// 每个线程道持有 4 个元素以覆盖 128 个维度: 32 个线程道* 4 = 128常量 int 查询_base_偏移量 = (b * 查询_步幅_注意力头) + (查询_注意力头 * 查询_步幅_h); 常量 int d_base = lane_id * 4; float q_reg[4];

// 即使 num_tokens == 0 也安全加载

```
load_bf16x4_to_f32(q + 查询_base_偏移量, d_base, q_reg); // 每个线程束/注意力头对应的累加器 float 输出_acc[4] = {0.f, 0.f, 0.f, 0.f}; float m =-CUDART_无穷大_F; // 逻辑值（缩放后）的运行最大值 float s = 0.f; // exp(logit - m) 的运行总和 // 如果没有令牌：写入零值、lse = -inf 并返回 if (num_tokens <= 0) { float zeros[4] = {0.f, 0.f, 0.f, 0.f}; store_f32x4_to_bf16(输出 + (b * kNumQOHeads + 查询_注意力头)* kHeadDim, d_base, zero
                                                
```

m =m_new;__syncthreads(); } // 最终处理：将输出除以 s 进行归一化，写入以 2 为底的 lse float inv_s = 1.f / s; float out_final[4] = { out_acc[0]* inv_s, out_acc[1]* inv_s, out_acc[2]* inv_s, out_acc[3]* inv_s }; // 存储输出 store_f32x4_to_bf16(out + (b * kNumQOHeads + q_head) * kHeadDim, d_base, out_final); ↪// lse，即对缩放后的逻辑值求对数和指数 logsumexp(logits_scaled) base 2 = (log(s) + m) / ln(2) if (lane_id == 0) { constexpr float ln2 = 0.693147180559945309417232121458176568f; float lse_val = (logf(s) + m) / ln2; lse_out[b * kNumQOHeads + q_head] = lse_val; } } // 主机端封装函数：验证输入、设置启动配置并启动内核 void gqa_paged_decode_h32_kv8_d128_ps1_cuda(const torch::Tensor& q, // [B, 32, 128] bfloat16 const torch::Tensor& k_cache, // [num_pages, 1, 8, 128]bfloat16 const torch::Tensor& v_cache, // [num_pages, 1, 8, 128]bfloat16 const torch::Tensor& kv_indptr, // [B, +1]32位整数 const torch::Tensor& kv_indices, // [num_kv_indices,]32位整数 float sm_scale, torch::Tensor& output, // [B, 32, 128]bfloat16 torch::Tensor& lse // [B, 32]单精度浮点数) { TORCH_CHECK(q.is_cuda(), "q 必须是 CUDA 张量"); TORCH_CHECK(k_cache.is_cuda(), "k_cache 必须是 CUDA 张量"); TORCH_CHECK(v_cache.is_cuda(), "v_cache 必须是 CUDA 张量"); TORCH_CHECK(kv_indptr.is_cuda(), "kv_indptr 必须是 CUDA 张量"); TORCH_CHECK(kv_indices.is_cuda(), "kv_indices 必须是 CUDA 张量"); TORCH_CHECK(output.is_cuda(), "output 必须是 CUDA 张量"); TORCH_CHECK(lse.is_cuda(), "lse 必须是 CUDA 张量"); TORCH_CHECK(q.dim() == 3, "q 必须是 [B, 32, 128]"); TORCH_CHECK(q.size(1) == kNumQOHeads && q.size(2) == kHeadDim, "q 必须是 [B, 32, 128]"); TORCH_CHECK(q.scalar_type() == at::kBFloat16, "q 必须是 bfloat16"); TORCH_CHECK(k_cache.dim() == 4, "k_cache 必须是 [num_pages, 1, 8, 128]"); TORCH_CHECK(k_cache.size(1) == 1 && k_cache.size(2) == kNumKVHeads && k_cache.size(3) == kHeadDim, ↪"k_cache 必须是 [num_pages, 1, 8, 128]"); TORCH_CHECK(k_cache.scalar_type() == at::kBFloat16,"k_cache 必须是 bfloat16")

;

TORCH_CHECK(v_cache.dim() == 4, "v CH_CHECK(v_cache.size(1) == 1 && v_cache.size(2) == kNumKVHeads && v_cache.size(3) == kHeadDim, ↪"v_cache 必须为 [num_pages, 1, 8, 128]"); TORCH_CHECK(v_cache.scalar_type() == at::kBFloat16, "v_cache 必须为 bfloat16") ;

PyTorch_CHECK(kv_indptr.dim() ==

```
    PyTorch_检查(kv_indptr.标量_类型() == 位于::kInt,        "kv_indptr 必须为 32位整数");
    PyTorch_检查(kv_索引.维度() == 1,          "kv_indices 必须为一维");
    PyTorch_检查(kv_索引.标量_类型() == 在::kInt,              "kv_索引必须为32位整数");

    常量 整数64_t B  = q.size(0);
    PyTorch_检查(kv_indptr.size(0)  == 键 + 1,"len_indptr 必须为批次_大小 + 1");

    常量 整数64_t num_页 = k_cache.size(0);
    PyTorch_检查(输出.sizes() == 查询.sizes(),           "输出必须与查询的形状相同");
    TORCH_CHECK(输出.scalar_type() == at::kBFloat16,      "输出必须为 bfloat16");
    PyTorch_CHECK(lse.dim() == 2 && lse.size(0) == B && lse.size(1) == kNumQOHeads    ,
                  "lse 必须为      [B, 32]");
    PyTorch_CHECK(lse.标量_类型() == 位于::kFloat,    "lse 必须是单精度浮点数");

    // 启动配置
    三维维度 grid;
    网格.x = 静态_转换<无符号>(B);
    网格.y = 静态_转换<无符号>(kNumKVHeads);
    网格.z = 1;

    常量表达式 整数 kBlockThreads  = 128; // 每个块 4 个线程束 => 4 每个键值头对应的 Q 头数量
    ↪  组
    三维维度 块(kBlockThreads);

    // K 和 V 的共享内存：2 * 128 单精度浮点数
    大小_t 共享内存_字节数 = 2 *kHeadDim *大小(单精度浮点数);

    自动 流 = 在::CUDA::获取当前 CUDA 流();

    常量__nv_bfloat16* 查询_ptr =重新解释_转换<常量
    ↪  __nv_bfloat16*>(q.data_ptr<at::BFloat16>());
    常量__nv_bfloat16* k_ptr =重新解释_转换<常量
    ↪  __nv_bfloat16*>(k_cache.data_ptr<at::BFloat16>());
    常量__nv_bfloat16* v_ptr =重新解释_转换<常量
    ↪  __nv_bfloat16*>(v_cache.data_ptr<at::BFloat16>());
    常量int32_t* indptr_ptr =kv_indptr.data_ptr<int32_t>();
    常量int32_t* 索引_ptr =kv_indices.data_ptr<int32_t>();
    常量__nv_bfloat16* 输出_ptr
    ↪  重新解释_转换<__nv_bfloat16*>(输出.data_ptr<at::BFloat16>());
    浮点数* lse_指针 =lse.data_指针<浮点数>();

    分组查询注意力_分页_解码_头32_键值8_维度128_每秒处理页数1_内核<kBlockThreads><<<网格, 块,
    ↪  共享内存_字节, 流>>>(
        q_ptr, k_ptr, v_ptr, indptr_ptr, 索引_ptr, sm_scale, 输出_ptr, lse_ptr,
        静态_转换<整数>(B), 静态_转换<整数>(页数_页)
    );

    CUDA_检查(cudaGetLastError());
}
```

C 解决方案生成提示

本附录包含用于生成数据集和案例研究中内核解决方案的提示。与其提供规定性的内核优化建议，这些提示更侧重于生成语法正确、可解析的代码，同时允许智能体自主探索实现策略。基础提示用于生成初始内核提案，优化提示则用于迭代改进。

C.1 Triton 基础提示

生成一个针对 {target_gpu} GPU 优化的 Triton 内核，用于

{definition}

Triton 版本：3.3.1

要求：
- 编写简洁、高效的 Triton 代码，针对 {target_gpu} 架构进行优化 - 使用现代 Triton 语法，正确进行网格计算并运用语言特性 - 包含必要的导入语句（torch、triton、triton.language as tl） - 实现规格中描述的确切功能 - 参考代码提供了数学规范，但未经优化；你的 Triton 实现应与其计算精度一致，同时实现高性能 ↪↪- 使用定义中的张量形状、数据类型和轴信息，指导内存访问模式和优化策略 ↪- 针对 {target_gpu} 的 GPU 特性（内存层次结构、计算单元等）进行优化

封装函数必须处理完整的设备管理：- 必要时将 CPU 张量移至 GPU（当 torch.cuda.is_available() 时，使用 .to('cuda') 或 .cuda()） ↪- 如果 GPU 张量需要 CUDA，但 CUDA 不可用，则抛出明确的错误 - 使用 GPU 张量调用 Triton 内核 - 将结果移回输入张量的原始设备 - 正确处理位置参数和关键字参数 - 保留原始张量所在的设备，并在输出中恢复这些设备

重要语法：使用正确的 Python 十六进制浮点字面量（0x1.234p5），请使用对应的十进制数值 - 不要使用 C/CUDA 特定语法 - 这是 Python/Triton 代码 - 所有代码都必须是有效的 Python，并且能够通过 ast.parse()

- 暴露一个可调用以执行内核的“run”入口函数 - 仅返回代码，不要添加解释或 Markdown 格式

仅生成完整、可运行的代码 - 不会有任何框架添加设备处理包装代码。↪

生成实现：

C.2 Triton 优化提示

你正在为 {target_gpu} GPU 优化一个 Triton 内核。当前实现存在需要修复的问题。↪

原始规范：{definition}

当前实现状态：{trace_logs}

当前实现：{current_code}

优化策略：
1. 确保正确性：如果存在编译错误、运行时错误或错误输出，请全力解决这些问题 ↪- 分析编译错误并修复语法/API 用法 - 修复形状不匹配、内存访问违规等运行时错误 - 确保数值正确性与参考实现一致

2. 优化性能：如果当前内核在功能上正确，请专注于性能优化 ↪

- 针对 {target_gpu} 优化内存访问模式 - 调整块大小和网格维度 - 使用恰当的 Triton 语言特性进行向量化 - 尽量减少全局内存事务

优化后的实现要求：
- 编写简洁高效、针对 {target_gpu} 架构优化的 Triton 代码 - 使用现代 Triton 语法，并正确进行网格计算、运用语言特性 - 包含必要的导入项（torch、triton、triton.language as tl） - 修复反馈中指出的所有问题 - 保持或提升计算精度 - 保留指定的相同函数签名和设备处理方式

包装函数必须负责完整的设备管理：- 必要时将 CPU 张量移至 GPU（当 torch.cuda.is_available() 时，使用 .to('cuda') 或 .cuda()）↪- 如果 GPU 张量需要 CUDA，但 CUDA 不可用，则抛出明确的错误 - 使用 GPU 张量调用 Triton 内核 - 将结果移回输入张量的原始设备 - 正确处理位置参数和关键字参数 - 保留原始张量的设备，并将输出恢复至这些设备

重要：仅使用有效的 Python/Triton 语法：
- 禁止使用十六进制浮点字面量（0x1.234p5），请改用十进制等效值；禁止使用 C/CUDA 特定语法——这是 Python/Triton 代码；所有代码都必须是有效的 Python，并能通过 ast.parse()

- 提供一个可调用以执行内核的「run」入口函数；仅返回改进后的代码，不要添加解释或 Markdown 格式

生成修正并优化后的实现：

C.3 CUDABase 提示词

你是 一个代码生成器。生成针对 ↪ 于 {target_gpu} GPU 优化的 CUDA 内核实现，用于实现以下规格。

规格：{definition}

要求：
- 编写针对 {target_gpu} 架构优化的简洁、高效 CUDA C++ 代码 - 使用适当的 CUDA 语法和针对 {target_gpu} 优化的内存管理 - 实现规格中描述的确切功能 - 参考代码提供了数学规范，但未经优化；你的 CUDA 实现应在保持其计算精度的同时提供高 ↪性能 ↪- 使用定义中的张量形状、数据类型和轴信息来指导内存访问模式和优化策略 ↪- 针对 {target_gpu} GPU 特性（内存层次结构、计算单元等）进行优化 - 对于固定的轴值，针对这些常量进行专门优化，而不是处理一般情况↪

重要：生 成代码时，请使用 XML 格式，且必须包含 3 个文件，并严格采用以下名称 ：

<头文件名称="kernel.h">_- 所有 CUDA 内核函数声明 - 主机函数声明 - 任何必要的结构体/类型定义

- 包含保护宏和必要的头文件</头文件_文件>

<cuda 文件名="kernel.cu">_- 所有 __全局__ 内核实现 - 所有 __
设备__ 辅助函数 - CUDA 专用优化和内存模式 - 正确的错误检查
和内存管理</cuda_文件>

<cpp_文件名="main.cpp">
- 用于启动内核的主机函数 - 内存分配和数据传输管理 - 设备管理和错误处理 - 名为 "run" 的入口函数，可用于执行实现 - 正确处理位置参数和关键字参数 - 将 CPU 数据移至 GPU，执行内核，并将结果返回 CPU - 使用 PYBIND11_MODULE 包含 PyTorch C++ 扩展绑定 - 必须通过绑定将 "run" 函数暴露给 Python - 正确实现 PyTorch 张量与 CUDA 指针之间的张量类型转换 - 包含所有必需的 PyTorch 头文件：#include <torch/extension.h></cpp_文件>

代码生成指南：
- 针对 {target_gpu} 使用合适的现代 CUDA 特性 - 优化内存合并访问并减少存储体冲突 - 有效利用共享内存以重用数据 - 考虑占用率和寄存器使用情况 - 使用 cudaGetLastError() 进行适当的错误检查 - 根据问题规模设置合适的网格和线程块维度 - 对频繁访问的只读数据利用常量内存 - 在 "run" 函数的所有张量参数中使用 PyTorch 张量 API（torch::Tensor） - 使用 .data_ptr<T>() 将 PyTorch 张量转换为 CUDA 指针，并指定合适的类型（例如 float、double、int） ↪- 确保正确进行 CUDA 流同步和错误处理

生成实现：

C.4 CUDA 优化提示

你正在为 {target_gpu} GPU 优化 CUDA 内核。当前实现存在需要修复的问题。↪

原始规范：{definition}

当前实现状态：{trace_logs}

当前实现：{current_code}

优化策略：
1. 确保正确性：如果存在编译错误、运行时错误或错误输出，请全力解决这些问题 ↪- 分析编译错误并修复语法/API 用法 - 修复形状不匹配、内存访问违规、内核启动失败等运行时错误 ↪- 确保数值正确性与参考实现一致 - 验证 CUDA 内存管理和同步是否正确

2. 优化性能：如果当前内核在功能上正确，请专注于性能优化↪

- 针对 {target_gpu} 优化内存访问模式和合并访问 - 调整块大小和网格维度，以实现最高占用率 - 有效利用共享内存，减少全局内存事务 - 优化寄存器使用情况并尽量减少分支分歧 - 如有帮助，可考虑使用专用库 - 利用常量轴值进行编译时优化

优化后实现的要求：
- 编写简洁、高效且针对 {target_gpu} 架构优化的 CUDA C++ 代码 - 使用正确的 CUDA 语法和适用于 {target_gpu} 的现代特性 - 修复反馈中指出的所有问题 - 保持或提升计算精度 - 保留指定的函数签名和设备处理方式 - 对于固定轴值，应针对这些常量进行专项优化，而非采用通用方案↪

重要：请以 XML 格式生成代码，并严格使用以下 3 个文件名：

<头文件_文件名="kernel.h">
- 所有 CUDA 内核函数声明 - 主机函数声明 - 任何必要的结构体/类型定义 - 包含保护宏和必要的头文件</头文件_文件>

<CUDA_文件名="kernel.cu">
- 所有__全局__ 内核实现 - 所有__设备__ 辅助函数 - CUDA 专用优化和内存模式 - 适当的错误检查和内存管理</CUDA_文件>

<C++_文件名="main.cpp">
- 启动内核的主机函数 - 内存分配和数据传输管理 - 设备管理和错误处理 - 名为 "run" 的入口函数，可用于执行实现 - 正确处理位置参数和关键字参数 - 将 CPU 数据移动到 GPU，执行内核，并将结果返回到 CPU - 使用 PYBIND11_MODULE 包含 PyTorch C++ 扩展绑定 - 必须通过绑定将 "run" 函数暴露给 Python - 正确处理 PyTorch 张量与 CUDA 指针之间的张量类型转换 - 包含所有必要的 PyTorch 头文件：#include <torch/extension.h></cpp_file>

代码生成指南：
- 使用适用于 {target_gpu} 的现代 CUDA 特性 - 优化内存合并访问并减少存储体冲突 - 有效利用共享内存进行数据重用 - 考虑占用率和寄存器使用情况 - 使用 cudaGetLastError() 实现适当的错误检查 - 根据问题规模使用适当的网格和线程块维度 - 对频繁访问的只读数据利用常量内存 - 在 "run" 函数中对所有张量参数使用 PyTorch 张量 API（torch::Tensor） - 使用 .data_ptr<T>() 将 PyTorch 张量转换为 CUDA 指针，并使用适当的类型（例如 float、double、int）↪- 确保适当的 CUDA 流同步和错误处理

生成经过修正和优化的实现：