TaxBreak：通过开销分解揭示大语言模型推理的隐性成本

Prabhu Vellaisamy, Shreesh Tripathi, Vignesh Natarajan, Surya Santhan Thenarasu, Shawn Blanton, John P. Shen*NeuroAI*计算机体系结构实验室 *(NCAL)* 电气与计算机工程系卡内基梅隆大学 (pvellais@andrew.cmu.edu, jpshen@cmu.edu)

arXiv:2603.12465v1 [cs.DC] 12 Mar 2026

摘要—**大语言模型（LLM）推理广泛应用于交互式助手和智能体系统。在对延迟敏感的部署中，推理时间可能主要由主机侧开销决定。现有方法通常只将这部分成本呈现为总体残差或启动/队列指标，往往不足以确定应优化哪个执行层。本文提出*TaxBreak*，一种基于跟踪的方法论，可将主机可见的编排开销分解为三个组成部分：框架转换时间、CUDA库转换时间和内核启动路径时间。我们在 NVIDIA H100 和 H200 系统上验证*TaxBreak*，并据此提出主机–设备平衡指数（*HDBI*），这是一种受限度摘要指标，用于关联设备活跃执行与主机可见编排。在预填充和解码阶段的典型稠密及混合专家工作负载中，我们表明，仅凭总体延迟、GPU非活动状态或受限度比率，可能无法看清主要优化目标。相较之下，*TaxBreak*可以区分应通过降低软件栈开销进行优化的情况，以及主要收益来自减少设备侧工作的情况。我们进一步发现，混合专家模型每生成一个输出词元，调度的内核数量比稠密模型多 8-11× 个；对于此类受主机限制的工作负载，CPU 单线程性能是一个首要参数：更快的主机 CPU 可将编排开销降低 10-29%，并将端到端延迟最多改善 14%，即使搭配的是时钟频率较低的 GPU。这些结果表明，*TaxBreak*可作为诊断工具，用于判断优化工作应针对软件栈，还是设备端工作负载执行。**

索引词—**框架税、CUDA库、内核启动、大语言模型推理分析、混合专家（MoE）模型。**

I. 引言

大语言模型（LLM），包括稠密架构（例如 GPT-4 [1], LLaMA [15], [28]）和混合专家（MoE）架构（例如 DeepSeek-V3、R1 [16], [20]），如今已成为严格延迟约束下运行的交互式人工智能服务的基础 [21]。在聊天机器人和编程助手等面向用户的系统中，端到端延迟会直接影响用户体验 [8], [32]。智能体工作流会放大这种敏感性：推理步骤与工具执行交替进行，会引入顺序依赖，导致开销在多轮迭代中不断累积 [26], [27], [33]。近期研究表明，在此类场景中，主机端编排日益成为关键路径的一部分，尤其是在控制流和工具调用发生在 GPU 执行流之外时 [24]。值得注意的是，在即时模式下，PyTorch 会在单个CPU线程上串行调度整个执行路径，因此相关的 CPU 性能指标是单核吞吐量，而非核心数。

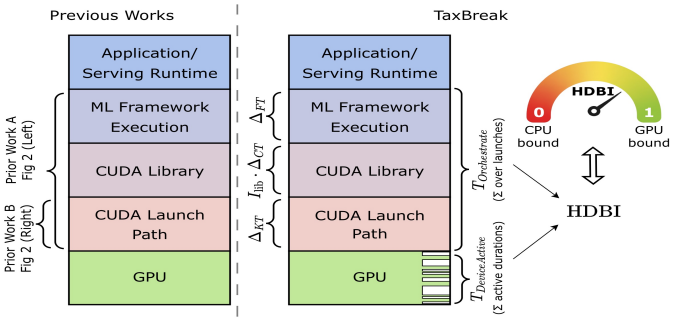


图1：***TaxBreak* 方法论**。*TaxBreak* 将整体主机端编排开销分解为三个部分：（i）框架执行；（ii）CUDA 库前端执行；（iii）内核启动调用。我们还提出一项新的主机-设备平衡指数（*HDBI*），用于刻画主机（CPU）与设备（GPU）之间相对受限程度。先前工作 A 指聚合框架开销 [14]；B 指内核启动/排队税（TKLQT） [30]。

如今，大语言模型推理通过层次复杂的推理栈运行。高级运行时（vLLM [18],Orca [34]）会管理批处理、调度和键值缓存状态，然后通过机器学习框架或编译器（PyTorch/TensorFlow [22], [23], TVM [9]）调度执行。随后，这些框架或编译器可能会调用厂商库（cuBLAS/cuBLASLt/cuDNN），也可能不会，并提交内核。尽管这种模块化设计提升了可移植性和抽象能力，却也在多个层面引入了主机端编排开销。以往工作通常将这项成本呈现为聚合的 CPU 端残余开销（即框架税 [14]），或通过单独分析内核启动/排队行为来衡量（即内核启动与队列总时间（*TKLQT*） [30]）。这些指标有助于判断工作负载似乎受主机还是设备限制，但无法指出推理栈中的哪一层是亟待优化的主要瓶颈。

对于现代工作负载，归因挑战愈发严峻。以推理为中心的研究表明，在异构系统中，性能既取决于硬件能力，也取决于主机软件编排 [19], [30]。以GPU为中心的分析进一步表明，性能存在显著的阶段依赖性（预填充与解-

码）以及粗粒度性能分析 [13], [31]难以发现的扩展效应。混合专家架构使问题更加复杂：虽然它们通过稀疏激活实现了较高的有效容量（例如，DeepSeek-V3 每个词元会激活 671B 个参数中的 37B [20]），但也会产生动态路由开销，并将执行拆分为许多相互依赖的小型内核 [4], [17]。即使在非分布式场景中，GPU 空闲时间增加也可能源于主机导致的延迟、工作负载导致的碎片化，或内存/数据移动效应；然而，现有性能分析器缺乏跨栈粒度，无法区分这些原因。

我们需要一种诊断框架，能够将推理效率低下归因于主机端抽象层和各个执行阶段。为此，我们提出*TaxBreak*，这是一种基于跟踪的方法，可将主机端编排分解为三个组成部分：(i) 框架执行，(ii)*CUDA* 库前端执行，以及 (iii)内核启动，适用于英伟达系统。我们进一步提出主机–设备平衡指数（*HDBI*），用于量化受 CPU 限制与受 GPU 限制的相对程度（图 1）。这种分解将粗略的主机与设备诊断转变为机制层面的归因，从而指导优化应针对运行时、编译器、CUDA/驱动程序路径，还是设备侧工作负载。

主要贡献。本文提出*TaxBreak*，这是一种用于分解主机端 LLM 推理开销的基于跟踪的方法。我们的贡献包括： 将框架转换、CUDA 库转换和内核启动开销分解为三个阶段的 *TaxBreak* 方法； 在英伟达 H100/H200 上进行验证，包括逐内核启动下限表征； 针对预填充和解码阶段的稠密推理与混合专家推理提供实证； 表明仅凭汇总指标可能无法识别主要优化目标； 提出主机-设备平衡指数（HDBI）这一诊断性摘要指标，与*TaxBreak* 归因相辅相成，用于判断工作负载是受主机限制还是受设备限制；以及 跨平台结果表明，更高的 CPU 单线程性能可使受主机限制的端到端延迟降低 11–14%。

II. 背景与动机

A. 推理服务动态

大语言模型推理服务分为两个阶段：预填充和解码。在预填充阶段，服务器处理完整的输入提示，生成第一个输出词元，并构建注意力 KV缓存。因此，预填充可利用提示词元间的大量并行性；当请求经过批处理后，能够实现较高的 GPU 利用率。相比之下，解码阶段会针对每个活跃请求逐个自回归地生成词元。因此，解码的请求内并行性较低，需要对大量并发请求进行批处理来提高利用率。由于注意力机制会反复读取并更新不断增长的 KV缓存，解码也越来越受内存行为影响 [12]。因此，生产系统会将首词元生成时间（TTFT）和每输出词元时间（TPOT）作为关键延迟绩效指标（KPI），分别衡量交互响应能力和稳态流式传输表现 [2], [3]。

KV缓存不断增长的内存占用，使分配、碎片管理和复用变得更加复杂，从而限制了可行的批大小和延迟。vLLM [18]通过将KV状态存储在非连续的分页内存中，减少内存碎片，从而实现灵活共享并降低分配成本。Orca [34] 提出的迭代级调度和选择性批处理等调度方法，能够缓解请求提前完成或延迟加入所造成的低效问题。此外，Sarathi-Serve [3]利用分块预填充和无停顿调度，优化吞吐量与延迟的权衡。这些服务技术通常构建在通用机器学习框架（PyTorch/TensorFlow/JAX [6], [22], [23]）和厂商运行时之上，其中框架执行以及库调用与内核启动路径会显著影响TTFT和TPOT。然而，这些影响在各层上的归因仍缺乏充分研究，也是本文关注的重点。

B. 稠密型与混合专家（*MoE*）大语言模型

稠密型大语言模型采用固定的逐词元计算图（所有层均处于激活状态），其预填充过程具有可预测性，且大多受计算限制；解码过程则以迭代方式进行，每个词元的开销固定，同时KV缓存不断增长。相比之下，混合专家推理会为每个词元激活一部分专家，并引入路由与门控、同步以及细粒度的依赖链式内核，从而同时增加主机端编排负担（路由分发、碎片化的内核提交）和设备端敏感性（细粒度依赖、内存停顿）。仅凭粗粒度指标（GPU 利用率、总内核启动时间）无法区分这些原因，因为混合专家模型中观察到的 GPU 空闲时间可能源自主机引发的延迟、设备依赖链或内存/数据移动停顿。先前研究记录了不同 MoE设计点中的路由开销与负载不均衡 [10]，并指出损失最优的专家扩展可能与服务效率不一致 [35], 。这些研究推动了面向MoE的特征分析，但并未提供*TaxBreak* 所提供的跨栈归因。

C. 执行栈剖析

端到端大语言模型延迟源于一个分层的执行栈，涵盖应用服务逻辑、框架运行时、编译器与降级流水线、厂商库以及提交路径。请求进入服务运行时（vLLM、Orca），由其处理批处理、调度和准入控制，随后通过机器学习框架（例如 PyTorch）调用模型执行。在 PyTorch 路径中，ATen分发器会解析张量元数据，并选择目标后端，然后将操作降级到厂商库（cuBLAS/cuDNN），或直接将逐元素操作和自定义内核提交到 CUDA启动与提交路径。

PyTorch 2 [5] 编译流水线会将 Python 执行捕获为 FX图（TorchDynamo），利用预先（AOT）自动微分和优化遍来融合并降低算子，然后通过 TorchInductor 生成后端内核。这些编译步骤能够减少 Python开销，但对于动态工作负载，可能会回退到即时模式。基于 CUDA 图的执行还能进一步降低稳态启动开销，但需要一次性支付捕获和实例化成本，而且通常要求张量形状、内存布局和控制流保持静态 [30]。然而，混合专家路由和专家执行会增加内核数量和异构性，从而产生主机

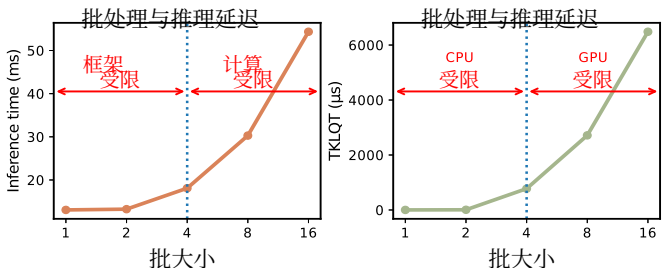


图 2：**先前对不同批大小下 GPT-2 推理的刻画**。左：端到端延迟（毫秒）显示，在较小批大小下会从框架受限转变为计算受限 [14]。右：其中的 TKLQT μs 突出显示了类似的从 CPU 受限到 GPU 受限的转变，且随着批大小和利用率的增加，内核排队也随之增加 [30]。

开销。解码期间尤为如此。关键在于，无论执行采用即时模式还是编译模式，每个推理步骤仍需经过 CUDA 运行时间/驱动程序提交路径：主机会发出运行时间/驱动程序调用，工作会进入流队列，GPU 则执行内核。即时执行时，主机调度线程会针对每个请求串行发出内核。尽管单次启动是异步的，调度路径仍然是单线程的。因此，主机端开销会随内核数量线性增长N；要降低开销，必须降低每个内核的调度成本（加快单线程执行或进行编译），或直接减少 N（通过内核融合、CUDA 图），而不是增加 CPU 核心。较短的逐词元迭代会放大每次迭代中固定的主机开销。因此，*TaxBreak* 聚焦于框架边界以下的执行（PyTorch/ATen 通过库映射接入 CUDA 启动路径），并测量主机端编排时间。尽管编译日益普及，但对于形状或控制流动态的工作负载，即时模式仍然存在，而*TaxBreak* 正是针对这条路径。

D. 关于 *AI* 推理税的先前工作

整体性的*AI* 税 研究表明，流水线与基础设施成本可能主导真实部署 [7], [25], [36], [37]。在服务器级大语言模型推理中，理论计算收益与实际延迟改善之间的差距被称为框架税，这促使了框架受限与计算受限的分类 [14]。CPU-GPU 耦合方面的工作揭示了内核启动/排队税，使内核提交开销得以显式呈现 [30]（图 2）。内核启动税同样与分布式场景相关，其中卸载与排队行为会放大延迟惩罚 [29]。

先前工作的局限性。 现有研究要么将主机端开销作为汇总残差呈现 [14]，要么主要将其定位在主机到设备（H2D）启动路径上（TKLQT） [30]。然而，这些研究均未对框架、库和驱动程序各层的启动前开销进行分解。 先前的特征分析通常以预填充为中心，或针对旧版工作负载/硬件（例如卷积神经网络、编码器模型、单精度浮点数） [14]；设备端分析则聚焦于微架构行为，却未归因主机端软件栈的开销 [31]。

表 I 总结了这些比较。这些差距促使我们：(i) 开发一种分层分解方法，分别分析框架转换、库映射和启动路径底限成本；(ii) 覆盖预填充和解码；(iii) 针对英伟达 H100/H200 上的稠密和混合专家工作负载。

III. *TaxBreak*方法论

在任何 GPU 内核执行之前，一次 PyTorch 操作会经过 Python 分发、ATen 算子解析、可选的供应商库前端以及 CUDA 启动 API。*TaxBreak*将这一主机端延迟分解为每次内核调用中三个互斥且完备的组成部分：

$$T_{Host} = \underbrace{T_{Py} + T_{dispatch_base}}_{\Delta FT} + \mathbb{I}_{lib} \cdot \underbrace{\max(0,\ T_{dispatch} - T_{dispatch_base})}_{\Delta CT} + \underbrace{T_{sys}^{floor}}_{\Delta KT} \quad (1)$$

其中： ΔFT （框架转换）是 Python 分发开销 $T_{Py} = t_{aten\ op} - t_{torch\ op}$ 与不可约的 ATen 调度成本 $T_{dispatch\ base}$之和； ΔCT （CUDA 库转换）是供应商库前端超出该基线的额外部分，并由$\mathbb{I}_{lib} \in \{0, 1\}$对非库内核门控为零； $\Delta KT = T_{sys}^{floor}$是从 cudaLaunchKernel 调用到 GPU 内核启动的硬件下限，通过空内核性能分析测得。原始启动成本单独报告为 $T_{launch}^{raw} = T_{sys}^{floor} + \Delta KT_{fw}$，以供诊断洞察，但不用于公式 2，因为 ΔKT_{fw} 反映的是已被 ΔFT 和 ΔCT捕获的框架入队开销。

对一次模型运行中的所有 N 内核调用求和，即可得到主机端编排开销：

$$T_{Orchestration} = \sum_{i=1}^{N} \left(\Delta FT^{(i)} + \mathbb{I}_{lib}^{(i)} \cdot \Delta CT^{(i)} + \Delta KT^{(i)} \right) \quad (2)$$

将核函数执行期间累计的设备活跃时间 $T_{DeviceActive} = \sum_{k=1}^{N} t_k$ 纳入考量，其中 t_k 表示核函数 k 的执行时间，这些量共同定义了主机-设备平衡指数：

$$HDBI = \frac{T_{DeviceActive}}{T_{DeviceActive} + T_{Orchestration}} \in (0, 1) \quad (3)$$

$HDBI \to 0$ 表示完全受主机限制的状态； $HDBI \to 1$ 表示完全受设备限制的执行。HDBI 是衡量设备活动时间与主机编排之间工作负载平衡关系的比值，并非 GPU 利用率指标（后者衡量设备活动时间相对于墙钟延迟的比例）。先前的工作 [14]将 T_{Host} 定义为推理延迟减去 GPU 活跃时间，但无法对其进行分解；而先前的内核税分析 [30]仅报告 $\sum \Delta KT$ 。公式 1 提供了完整的逐内核分解。

使用 HDBI 进行诊断性解读。当 HDBI 表明工作负载受主机限制（$HDBI \to 0$）时， $T_{Orchestration}$分解会指出执行栈中占主导地位的层，从而为优化策略提供参考：

- 如果 $\sum \Delta FT + \sum \Delta CT$占主导地位，瓶颈就在软件栈中（Python 分发和库前端

表 I：与先前工作的比较

Work	税收粒度	CPU-GPU	跨层开销	工作负载	预填充 + 解码	Hopper-Era HW	评估平台
AI 税 [25]	流水线级	✗	✗	Google FaceNet，R-CNN	✗	✗	边缘数据中心部署
框架税 [14]	粗残差	✗	✗	BERT-Base；ResNet-50；DistilBERT；GPT-2；WavLM。	✗	✗	NVIDIA Pascal、Turing, Ampere。
TKLQT / Kernel-Tax 查看 [30]	仅启动路径	✓	✗	Bert-Base-Uncased；XLM-Roberta-Base；GPT-2；Llama-3.2-1B。	✗	✓	PCIe A100/H100 和紧密耦合的 GH200（以预填充为中心）。
GPU 推理特征描述 [31]	以设备为中心	✗	✗	Llama-3-8B；Qwen2.5-7B/-32B；Qwen3-30B-A3B。	✓	✗	4× NVIDIA A100 80GB（SXM）+ Jetson AGX Orin。
本研究 (TaxBreak)	**主机栈归因：** ΔFT, ΔCT, ΔKT	✓	✓	**Llama-3.2-1B,-3B, OLMoE-1B/7B; Qwen1.5-MoE-A2.7B**	✓	✓	**H100（DGX H100）；H200 NVL（单GPU）。**

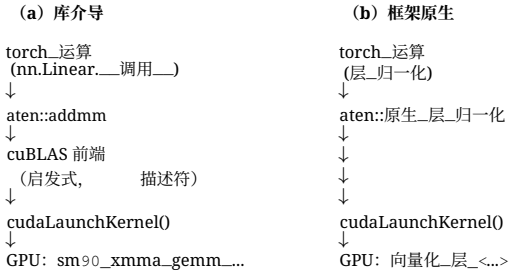


图 3： **库介导内核和框架原生内核的调度链**。对于库介导的情况，cuBLAS 前端产生 ΔCT，用于 $\mathbb{I}_{lib} = 1$。

开销），优化应针对运行时编译（例如 torch.compile）或库分发路径。

- 如果 $N \cdot T_{sys}^{floor}$占主导地位，成本会随内核数量增长，而内核融合将带来最大幅度的降低。
- 如果每个内核的框架启动开销 ΔKT_{fw} =max(0, $T_{launch} - T_{sys}^{floor}$)显著，那么驱动程序/运行时路径就是瓶颈；CUDA 图或持久化内核有助于分摊成本。

A. 内核族分类

$\mathbb{I}_{lib}$ 指示变量控制公式 2 中的 ΔCT 项，确保供应商库开销仅计入库介导内核，并决定内核如何经过软件栈（如图 3 所示）：

- **库介导**($\mathbb{I}_{lib} = 1$)：通过供应商库（cuBLAS、cuDNN）进行路由。在调用 CUDA 启动 API 前，会产生启发式选择、描述符设置和打包等开销，增加 $\Delta CT > 0$。
- **框架原生** ($\mathbb{I}_{lib} = 0$)：由 ATen/Inductor 直接生成（例如 aten::mul、aten::native_layer_norm）或由数据移动操作（cudaMemcpyAsync、cudaMemset）生成。不涉及供应商库前端；$\mathbb{I}_{lib}$ 将 ΔCT 设为零。

B. 两阶段测量流程

TaxBreak 分两个阶段测量公式 1 的各个组成部分。阶段1从完整模型跟踪中捕获 Python 可见的框架开销，并构建内核数据库。第 2 阶段首先通过空内核 nsys 运行测量动态系统下限，然后单独重放每个唯一内核，以测量分发转换、启动下限开销及其残余开销，同时避免队列干扰。

阶段1：完整模型跟踪。我们在 PyTorch 分析器下执行 R 次分析迭代，前面先进行 W 次预热运行（数值见第 IV 节）。分析器会捕获带时间戳的 Python/torch 运算符、ATen 运算符、CUDA 运行时调用，以及通过关联 ID 链接的 GPU 内核。从最后一次分析迭代中，我们提取内核启动序列，并计算每次调用的框架转换开销：

$$\Delta FT^{(i)} = T_{Py}^{(i)} + T_{dispatch_base} \tag{4}$$

其中，$T_{Py}^{(i)} = t_{aten_op}^{(i)} - t_{torch_op}^{(i)}$ 是执行到 ATen C++ 层之前的 Python 端调度开销，而 $T_{dispatch\ base}$ 是第 2 阶段通过框架原生内核测得的不可约框架分发开销。我们还会构建一个内核数据库，记录每个唯一内核经过清理的名称、网格/线程块配置、ATen 元数据、调用频率及 $\mathbb{I}_{lib}$ 分类。

第 2 阶段：隔离重放。我们首先通过一次空内核 nsys 运行来测量 T_{sys}^{floor}。然后，针对每个内核数据库条目，我们在 nsys 下隔离重放原始 ATen 运算。共享相同 ATen 元数据（运算符、形状、数据类型、标量参数）、目标内核名称和启动配置的内核会通过全局缓存去重并加以划分，从而仅对未缓存的条目进行剖析，显著节省运行时间。每次重放均限定在 NVTX 范围内，并通过 torch.cuda.synchronize() 串行化，以防止 GPU 队列重叠（图 4）。

经过 W 预热迭代后，nsys 为每次调用记录三个时间戳 j （图 4）：t_{nvtx} 位于第 (1) 行，t_{api}（cudaLaunchKernel）位于第 (2) 行，以及 t_{kernel} （GPU 执行开始）位于第 (3) 行。对于每个匹配的重放核函数，我们

```
nvtx.range_push("aten_调度")                    # t_nvtx（1）
执行_操作(op_名称,                              输入) # t_API（2）
nvtx.range_pop()                                      # t_核函数（3）
torch.cuda.synchronize()                        # 核函数完成
```

图 4：**运算符分发和核函数执行周围标注的 NVTX范围。**

报告每次调用的 $T^{(j)}_{dispatch}$与 $T_{(j)launch}$值在 R 重放运行中的均值。这些时间戳直接对应：

$$T^{(j)}_{\text{dispatch}} = \underbrace{t^{(j)}_{\text{api}} - t^{(j)}_{\text{nvtx}}}_{\text{lines (1)–(2): host dispatch}} \tag{5}$$

$$T^{(j)}_{\text{launch}} = \underbrace{t^{(j)}_{\text{kernel}} - t^{(j)}_{\text{api}}}_{\text{lines (2)–(3): launch gap}} \tag{6}$$

t_{nvtx}、t_{api} 和 t_{kernel} 分别来自 NVTX_EVENTS、CUPTI_ACTIVITY_KIND_RUNTIME 和 CUPTI_ACTIVITY_KIND_KERNEL，并通过关联ID关联。

1) 分发基线：NVTX范围同时涵盖 ATen 调度和任何厂商库前端，因此，对于 $\mathbb{I}_{lib}$=1 内核， $T_{dispatch}$将两者混为一谈。我们通过减去分发基线来分离 ΔCT ，该基线定义为框架原生内核 $T_{dispatch}$的中位数：

$$T_{dispatch_base} = \text{median}\big(\{T^{(k)}_{dispatch} : \mathbb{I}^{(k)}_{lib} = 0\}\big) \tag{7}$$

然后我们计算

$$\Delta CT = \max(0, T_{dispatch} - T_{dispatch_base}) \tag{8}$$

此外， T^{floor}_{sys}是在相同协议下剖析得到的空C++__global__ 空内核的平均 T^{raw}_{launch}。

2) 核函数匹配：由于自动调优，重放可能会调度一个与原始跟踪不同的变体。在将重放候选缩小至目标邻域后，我们通过以下基于名称的回退层次确定最终核函数，其中 $\bar{n}$ 表示清理后的（规范）核函数名称：

$$\text{match}(k) = \begin{cases} \text{exact}, & \bar{n}_{\text{replay}} = \bar{n}_{\text{trace}} \\ \text{substring}, & \bar{n}_{\text{replay}} \subseteq \bar{n}_{\text{trace}} \\ & \vee\ \bar{n}_{\text{trace}} \subseteq \bar{n}_{\text{replay}} \\ \text{most-frequent}, & \text{otherwise} \end{cases} \tag{9}$$

IV. 实验基础设施

A. 硬件设置

我们在以下NVIDIA GPU平台上进行评估：

- **H100平台：** Intel Xeon 8480C（PCIe Gen5，56核@2.0/3.8 GHz），在DGX H100系统中与一块NVIDIA H100（80GB）GPU搭配。
- **H200平台：** Intel Xeon Gold 6538Y+ （32 核）搭配 NVIDIA H200 NVL（141 GB）GPU。

所有实验均使用**单GPU Slurm 分配**。为确保跨平台比较公平，我们在两个系统上都为每个 GPU 分配**6 个 CPU 核心** 和 **32GB** 主机内存。

6 核分配超出了单线程分发路径的需求，并为操作系统调度留出了余量。

测量。为消除冷启动和瞬态效应，每种配置在阶段1和第 2 阶段中均先运行（W=50）次预热迭代，然后进行（R=150）次测量运行。报告的延迟为各次运行的平均值，且 $T_{Orchestration}$ 的 95% 置信区间在所有配置中均低于 0.34 毫秒，表明测量结果稳定。

B. 软件设置

软件环境使用 Python 3.13、PyTorch 2.10、用于运行时库的 CUDA工具包 12.6，以及用于内核级追踪的 NVIDIA Nsight Systems 12.8.1（nsys）。这些追踪记录提供了 TaxBreak 分解所需的边界（框架分发、CUDA库前端处理和启动路径区间）。除非另有说明，实验均在即时模式下运行。

C. 工作负载

我们对 bfloat16 下的稠密模型和混合专家大语言模型进行性能分析，包括 Llama-3.2-1B/-3B [15] （**稠密模型**）、GPT-2（124M）以及 OLMoE-1B/7B、Qwen1.5-MoE-A2.7B（**混合专家模型**）。使用 GPT-2 是为了与先前的 TKLQT 特征分析 [30]进行直接比较。

V. 实验结果与分析

A. 端到端预填充与解码延迟

我们分两个阶段评估端到端推理：预填充（侧重 TTFT）和解码（ m 输出词元的累计延迟）。图 5 展示了稠密模型和混合专家工作负载在 H100/H200 上，不同批大小（BS）和序列长度（SL）下的延迟。在以下所有小节中，解码数值均指对 m=10 输出词元进行一次解码运行的总延迟。

稠密模型趋势。稠密模型预填充的延迟主要随序列长度增长，因为注意力计算量会随提示长度增加：在 H100 上，Llama-3.2-1B 的延迟从 SL=512 时的 22 毫秒升至 SL=8192时的 586 毫秒（ 26.6×），而 Llama-3.2-3B 在 SL=8192 时达到 827 毫秒。在此场景下，批处理能有效摊销固定开销：在 SL =512时，批大小从 1 增至 16，Llama-3.2-1B 的预填充延迟仅增加 3.6× （从 22 毫秒升至 79 毫秒）。稠密模型解码的表现则有所不同。在（BS=1/SL=512）下，生成 10 个解码词元需要 188 毫秒，而相同上下文的预填充仅需 22 毫秒。与预填充不同，解码以迭代方式进行，因此延迟会随着反复的内核启动、GPU 执行和键值缓存访问不断累积。随着序列长度增加，解码瓶颈逐渐转向键值缓存内存流量，在 SL=8192时达到 756 毫秒。增大批大小同样有所帮助，主要是因为设备工作量随之增加：在 S L=512时，批大小从 1 增至 16，解码延迟仅增加 1.3×，而整体解码吞吐量提升 11.9× （从 53 词元/秒升至 634 词元/秒）；该吞吐量是在整个批次的 $m = 10$ 解码窗口内测得的。

混合专家模型趋势。由于路由会引入大量短小且存在依赖关系的内核，混合专家模型的预填充和解码在性质上有所不同。在 H100 上，批大小为 =1、序列长度为 =512 时，OLMoE-1B/7B 的预填充耗时为 260.5 毫秒，尽管其激活参数量相近，耗时仍是 Llama-3.2-1B 的 11.7× 倍，并调度了

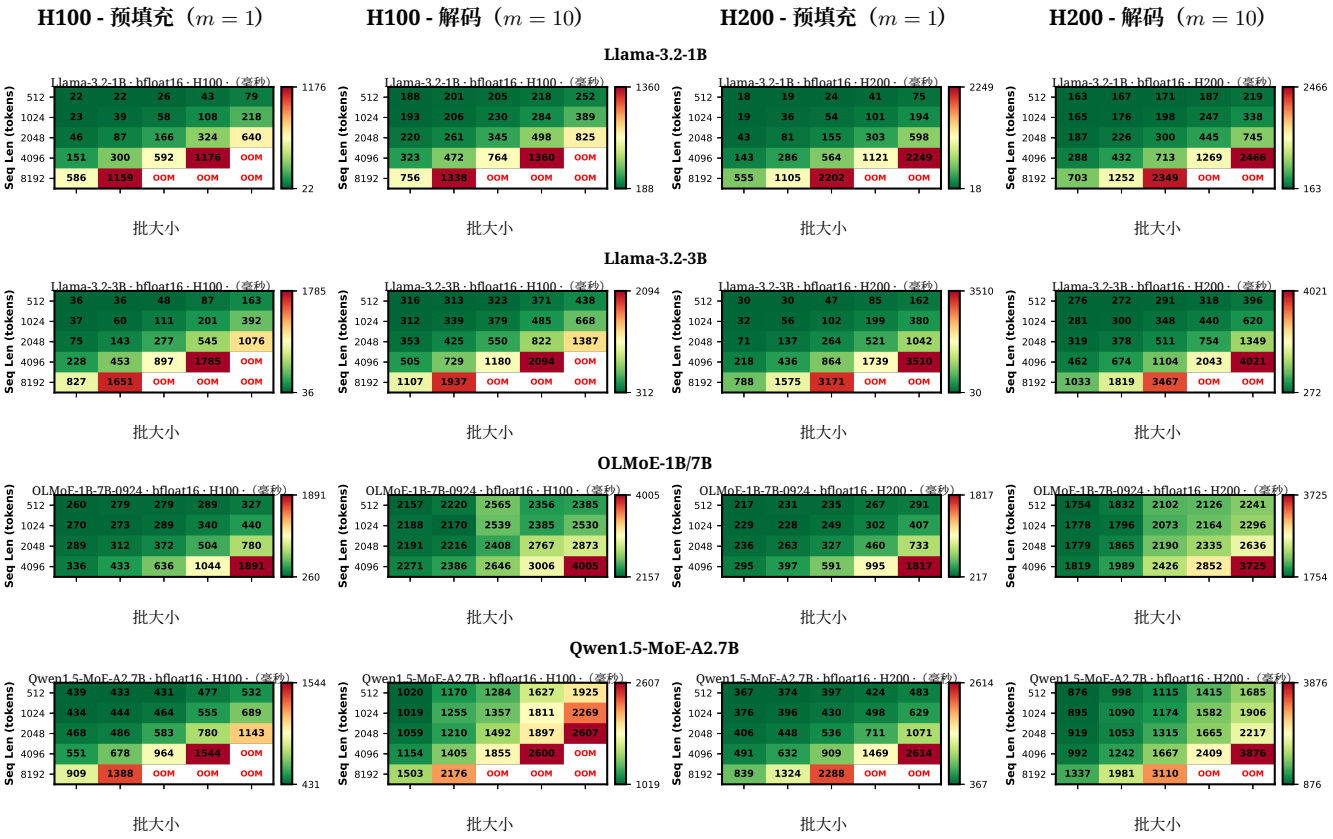


图5：**稠密与MoE大语言模型工作负载的端到端延迟**。热力图展示了H100/H200系统上预填充（$m = 1$）和解码（$m = 10$）阶段在不同批大小和输入序列长度下的端到端延迟，其中 m 表示生成的词元数量。解码热力图报告了在10个词元解码窗口上聚合的总延迟。OLMoE-1B/7B不支持序列长度=8192上下文长度。

13,741 个内核，而稠密模型为 850 个。在相同配置下，Qwen1.5-MoE-A2.7B 的内核启动更加零散（22,558 次）。在解码阶段，主机瓶颈变得占主导地位：OLMoE-1B/7B 在批大小为 =1、序列长度为 =512 时耗时 2157 毫秒（比稠密 Llama-3.2-1B 慢 11.5× 倍），且耗时随上下文长度变化几乎保持平稳，仅从 2081 毫秒（序列长度为 =512）变为 2164 毫秒（序列长度为 =4096）。这种对上下文长度不敏感的现象反映了分发主导阶段：在批大小为 =4、序列长度为 =2048（m=10）时，OLMoE-1B/7B 每个词元调度 9,305 个内核（表 II）；每个内核的耗时下限约为 4.7 μ 秒（表 III），因此仅内核启动路径的开销在加入框架分发之前，每个词元就约为 44 毫秒。因此，主要的改进手段是减少内核数量或摊薄提交路径的开销。

跨平台比较。H200对稠密预填充的提升在短上下文时最大（≈18%），并因计算饱和而在长上下文时收窄至 ≈5%。对于解码，H200在各序列长度上使稠密延迟降低12-15%，并使混合专家延迟降低14-19%。然而，这些收益并未消除占主导地位的混合专家瓶颈：绝对延迟有所改善，但路由和分发相关主机开销仍然是一阶因素。

关键结论 #1。 随着工作负载增大，稠密模型会从小批量时受主机限制，转为受计算或内存限制。混合专家模型的解码则不会：其延迟仍主要由

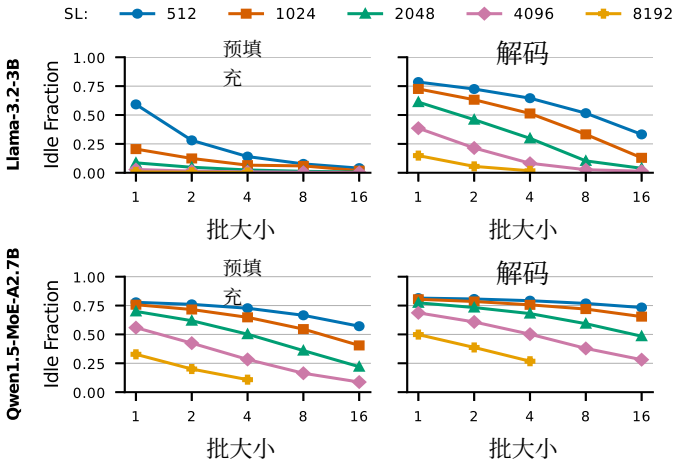


图 6：**H200 上不同批大小和序列长度下的空闲比例**。空闲比例表示 GPU 未能充分执行内核的延迟占比。

主机侧路由和调度，因此硬件升级可以降低绝对延迟，但不会改变瓶颈。

指标	Llama-3.2-1B	Llama-3.2-3B	OLMoE-1B/7B	Qwen1.5-MoE-A2.7B
内核启动总数	8,475	15,369	93,053	66,951
唯一内核名称	77	76	222	223
每个令牌的内核数	847.5	1,536.9	9,305.3	6,695.1
多样性比率	0.0091	0.0049	0.0024	0.0033
GPU 利用率 (%)	58.9	67.6	15.5	27.7

表 II：**H100 上稠密模型与混合专家模型的内核碎片化（批大小=4/序列长度=2048，** m**=10）**。

B. 混合专家延迟：空闲比例与内核碎片化

虽然总体延迟表明混合专家解码始终受主机限制，但无法指出推理时间损耗在哪里。我们将*GPU* 空闲比例（捕获非计算部分）定义为：$((T_{\text{e2e}} - T_{\text{DeviceActive}})/T_{\text{e2e}})$，其中 T_{e2e} 是推理的挂钟延迟。图 6 展示了 H200 上不同批大小和序列长度下的空闲比例，并比较了预填充与解码阶段。

Llama-3.2-3B。稠密模型提供了一个基线：随着工作负载规模扩大，编排开销会迅速摊销。在预填充执行期间，增加序列长度或批大小会扩大每次内核启动的通用矩阵乘法工作负载，使空闲比例从 59.2%（批大小=1/序列长度=512）迅速降至 0.8%（批大小=1/序列长度=8192）。当批大小超过 ≥ 4 且序列长度超过 ≥ 2048时，空闲比例保持在 2.5% 以下，表明执行实际上受计算限制。当包含解码时（m=10），由于反复出现小批量内核突发，空闲比例会上升（批大小=1/序列长度=512时为 78.5%）；但批处理在很大程度上缓解了这一影响，当批大小达到 ≥ 8 且序列长度达到 ≥ 2048后，空闲比例降至 5% 以下。

Qwen1.5-MoE-A2.7B. 在整个扫描范围内，混合专家模型的空闲比例始终居高不下，这表明即使批大小和序列长度较大，推理也不会变成计算受限。在预填充阶段，批大小为=1时，空闲比例从 77.8%（序列长度为=512）到 32.8%（序列长度为=8192）不等，与稠密模型在解码阶段的空闲比例相当。增大批大小只能部分降低这项开销：即使在批大小为=16/序列长度为=4096时，Qwen1.5-MoE-A2.7B 的空闲比例仍为 8.7%，而相同配置下的 Llama-3.2-3B 仅为 0.2%（比值为 44×）。这种差异在解码阶段依然存在。当序列长度为=512时，将批大小从 1 增至 16，混合专家模型的空闲比例仅从 81.5% 降至 73.3%。在批大小为=1/序列长度为=1024时，Llama-3.2-3B 的空闲比例为 20.5%，而 Qwen1.5-MoE-A2.7B 则高达 75.7%（高出 3.7× ）。

此类模式也会出现在单GPU设置中，因此不能归因于跨设备专家通信。它们表明，与专家路由和碎片化调度相关的非计算开销值得关注，接下来我们将使用*TaxBreak*对其进行分析。

C. TaxBreak 结果

混合专家模型中的内核碎片化。表 II 量化了混合专家模型持续保持高空闲比例的结构性根源。在 H100 上采用固定的解码配置（批大小=4/序列长度=2048, m=10）时，OLMoE-1B/7B 每次推理会调度 93,053 个 GPU 内核—— 11×相比之下，Llama-3.2-1B 为 8,475 个， 6×而 Llama-3.2-3B 为 15,369 个。Qwen1.5-MoE-A2.7B 同样会调度 66,951 个内核（8× Llama-3.2-1B）。按每个输出词元归一化后，OLMoE-1B/7B 每个词元需要启动 9,305 个内核，而 Llama-3.2-1B 需要 848 个（相差 11× ）。尽管启动量更大，混合专家模型的内核多样性比率（唯一名称数/内核启动总数）却更低：OLMoE-1B/7B 为 0.002，Llama-3.2-1B 为 0.009。这证实，混合专家模型的碎片化源于少量路由和专家通用矩阵乘法内核被反复调用，而非架构异构性。这种内核启动数量激增的情况，与 OLMoE-1B/7B 的 GPU 利用率仅为 15.5%（Llama-3.2-1B 为 58.9%）相吻合。由于调度采用单线程方式，每增加一个内核都会增加主机路径开销；当内核完成速度快于调度速度时，GPU 可能无法获得充足的工作负载。这些结果表明，在此类场景下，仅提升内存速度不太可能消除混合专家模型的主机受限行为。

关键结论 #2。在所测量的解码配置（批大小=*4*/序列长度=*2048*， m=*10*）下，混合专家模型每个输出词元调度的 *GPU* 内核数量，比活跃参数数量相近的稠密模型多 *8-11*× 个。这种启动次数膨胀源于反复调用少数路由和专家*GEMM*内核，而增加批大小或使用更快的内存都难以摊销这些开销。与稠密模型不同，混合专家推理在解码期间始终受主机限制，即使批大小为=*16*也是如此，因为每个词元所需的主机端调度工作量高出一个数量级，而且无论计算能力或批大小如何，这种情况都会持续存在。

$\mathbf{T_{sys}^{floor}}$和 $\mathbf{\Delta KT_{fw}}$ 特征分析。表 III 中的空内核测量结果表明， T_{sys}^{floor}在 Hopper平台上数值较小且稳定。接下来，我们考察实际内核的启动延迟是否足够接近这一硬件下限，使其能够作为有意义的不可约基线。为回答这一问题，我们将观测到的启动延迟分解为硬件下限T_{sys}^{floor}和可归因于框架的残余项 $\Delta KT_{fw} = \max(0,\ T_{launch} - T_{sys}^{floor})$，并按内核族在表 IV 中报告结果。

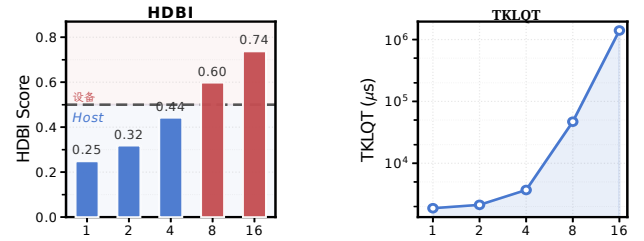
GPU	avg	p50	p5	p95
H100	4.707	4.578	4.260	5.396
H200	4.503	4.452	4.177	4.909

表 III：**独立测得的空内核** T_{sys}^{floor}（μ**s**）。

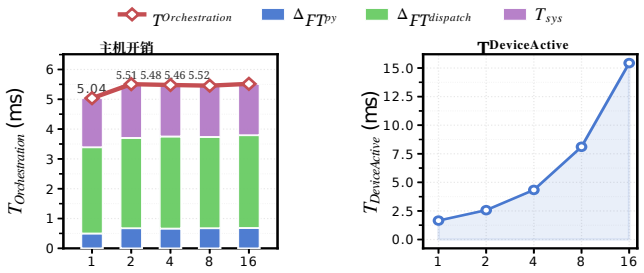
表 IV 显示，大多数扫描、归约和逐元素内核系列的启动时间都接近 T_{sys}^{floor}，残差中位数低于 0.6 μs，典型偏差仅比下限高 7-12%。这支持将 T_{sys}^{floor} 用作大多数内核调用不可约的启动基线。主要偏差出现在 GEMM 系列中：对于 Llama-3.2-3B，cuBLAS 达到 1.88 μs（比 T_{sys}^{floor} 高 40%）；对于 OLMoE-1B/7B，则达到 1.70 μs（高 36%）；与此同时，nvJit GEMM 也高于下限。因此，表 IV 支持预期的划分：T_{sys}^{floor} 捕获所有启动共有的硬件下限，而 ΔKT_{fw} 捕获该下限之上依赖于系列的软件开销。$p95$ 的 Llama-3.2-3B nvjet GEMM 系列数值达到 18.58 μs，尽管其中位数低得多（$p50$）。我们将其视为长尾启动异常，可能反映了变体——

内核族	Llama-3.2-3B（μ秒）				OLMoE-1B/7B（μ秒）			
	$p50$	$p95$	ΔKT_{fw}	↑	$p50$	$p95$	ΔKT_{fw}	↑
$\mathbf{T}_{\mathbf{sys}}^{\mathbf{floor}}$ (null)	4.75	4.75	–	–	4.73	4.73	–	–
扫描（前缀）	5.07	5.16	0.32	7%	5.12	5.13	0.40	8%
元素（循环展开）	5.11	5.28	0.36	8%	5.11	5.20	0.38	8%
元素（向量）	5.13	5.38	0.38	8%	5.30	5.49	0.58	12%
归约	5.30	5.30	0.55	12%	5.07	5.46	0.34	7%
元素（通用）	5.31	5.46	0.56	12%	5.31	5.58	0.58	12%
通用矩阵乘法 (nvjett)	5.93	18.58	1.18	25%	5.57	6.17	0.84	18%
通用矩阵乘法（cuBLAS）	6.63	6.63	1.88	40%	6.43	6.43	1.70	36%

表 IV：**各系列的启动延迟**（单位： μs），相对于 H100 上 Llama-3.2-3B 和 OLMoE- 1B/7B（批大小=1/序列长度=512；预填充）的下限 T_{sys}^{floor}。大多数扫描、归约和逐元素内核系列的延迟都接近下限，而 GEMM系列的延迟最高 ΔKT_{fw}。 T_{sys}^{floor}行报告了上下文内重放下限，由于进程级 CUDA 上下文状态，它与表 III 中的独立测量值（≈0.04 μs）略有差异。



（a）HDBI 与 TKLQT。HDBI 在批大小= 8附近跨越主机/设备边界，而在同一区域，随着 GPU 趋于饱和，TKLQT 急剧上升。



(b) 主机编排分解与设备活动时间的对比。

图 7：**H200 上 GPT-2 在不同批大小下的主机与设备行为。**

选择或框架/运行时间重放效应，而非底层下限发生变化。

GPT-2 案例研究。 我们使用 H200 上的 GPT-2，将*TaxBreak* 与先前基于 TKLQT 的表征 [30] 进行比较。图 7a 显示，HDBI 随批大小单调上升，从 0.25（批大小=1）升至 0.74（批大小=16），主机到设备交叉点位于批大小=4 和批大小=8之间。

图 7b 解释了这一转变的来源。 $T_{Orchestration}$在批大小为 5.04 毫秒（批大小=1）至 5.52 毫秒（批大小=16）的范围内几乎保持平稳。这种批大小不变性符合串行调度路径的特征：批次增大了 GPU 计算量，却不会显著改变主机提交行为。按内核数量归一化（不同批大小下为 376 至 394 个），每个内核的主机开销为 ≈13.7 μ秒，且基本保持恒定（±2.3%），这证实 $T_{Orchestration}$ 随批大小线性增长，且每次分发开销固定。相反，主要变化体现在设备工作量上： $T_{DeviceActive}$ 从 1.66 毫秒（批大小=1）增至 15.43 毫秒（批大小=16）。在主机开销分解中，H200 上的 GPT-2 运行也十分稳定： T_{Py} 仅从 0.50 毫秒增至 0.68 毫秒， $T_{dispatch\ base}$ 保持在 2.89–3.11 毫秒之间， ΔCT 在此工作负载下仍为零，而 T_{sys}^{floor}的贡献则几乎恒定在 1.65–1.80 毫秒。这个零ΔCT 结果反映出，GPT-2 的矩阵乘法内核采用框架原生的 nvjet/gemv2T，而非 cuBLAS/cuDNN，因此 $\mathbb{I}_{lib} = 0$ 和 ΔCT 被抑制为零。

这一行为也阐明了 HDBI 与 TKLQT 之间的关系（图 7a）。这两个指标捕获了相同的状态转变，但在批大小较大时，TKLQT 的上升幅度更大，因为 GPU 占用率提高后，它会计入队列延迟。HDBI 在这两种状态下都具有可解释性，因为它反映的是 $T_{DeviceActive}$ 与$T_{Orchestration}$ 之间的平衡，而非仅反映显露出来的启动路径。对于 H200 上的 GPT-2，*TaxBreak* 显示，交叉点是由设备工作量增加所驱动的，而主机编排几乎保持不变，且主要由 $T_{dispatch\ base}$ 主导，不存在隐藏的供应商库贡献。

$\mathbf{T}_{\mathbf{Orchestration}}$ 和 HDBI 特征分析。图 8 展示了 H200 特征分析 $T_{Orchestration}$、 $T_{DeviceActive}$ 以及 HDBI 在不同序列长度和批大小下的情况。解码值是对 m=10 个自回归步数进行汇总得出的，因此反映的是逐令牌执行的累积值。

随着工作负载从预填充转向解码，稠密执行与混合专家执行之间的差异逐渐扩大。在 H200 上，批大小为 =1、序列长度为 =512 时，Llama-3.2-1B 和 Llama-3.2-3B 在 m=10 解码窗口内的总 $T_{Orchestration}$ 分别为 102.13 毫秒和 188.27 毫秒，而单步预填充中的对应值（m=1）分别为 10.5 毫秒和 17.6 毫秒。每步编排开销几乎相同，解码总耗时意味着两种模型每步分别耗时 ≈10.2 毫秒和 ≈18.8 毫秒。这种不变性与内核分发数据一致：Llama-3.2-1B 在预填充中启动 850 个核函数，在十个解码步骤中启动 8,437 个（每步 ≈844 个），表明对于即时模式下固定的稠密架构，每次前向传播的调度次数 N 大致不受形状影响。因此，总 $T_{Orchestration}$ 的 ≈10×增长反映了 m=10 步数的累积效应。尽管如此，HDBI 仍大幅下降：Llama-3.2-1B 从 0.37 降至 0.23，Llama-3.2-3B 从 0.41 降至 0.24。这是因为解码中的逐步 $T_{DeviceActive}$ 远低于预填充（每步处理一个新词元，而预填充可并行处理 512 个词元），使执行开销的比重转向主机可见执行。

对于稠密模型，这种受主机限制的转变不会无限持续。在 BS=1/SL=512 这一最小解码场景中，稠密模型最明显地受主机影响（HDBI≈0.23）。在 BS=4/SL=512、BS=1/SL=4096，尤其是 BS=4/SL=4096 这些更大的解码场景中，随着批大小和上下文长度增加，每步 GPU计算量随之增长，执行又逐渐转向以设备为主。 $T_{Orchestration}$ 的增长速度远低于这些场景中的 $T_{DeviceActive}$ ，HDBI 因而相应上升。因此，稠密模型轨迹在预填充阶段较为均衡（HDBI≈0.37-0.41），在小批

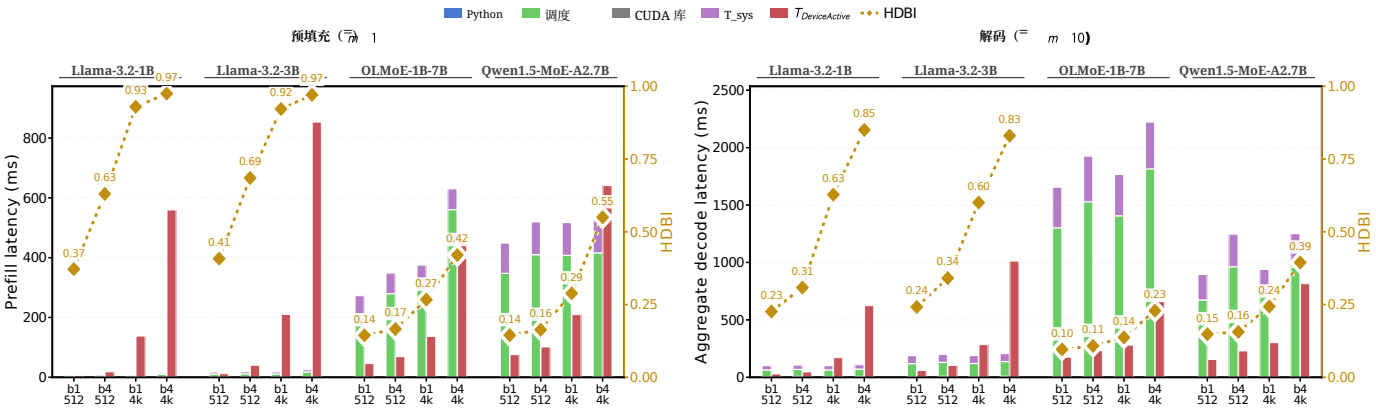


图 8：**H200 $T_{Orchestration}$ 以及稠密模型和混合专家工作负载下的 HDBI 特征分析**。堆叠柱状图将$T_{Orchestration}$ 分解为 Python、调度、CUDA 库和 T_{sys} 的贡献；相邻柱状图显示 $T_{DeviceActive}$；菱形标记表示 HDBI。预填充使用 m=1。解码柱状图和 HDBI 数据点表示在匹配的 m=10 解码运行中汇总得到的总量。注意：各面板的 y 轴刻度不同；跨面板比较绝对时间时，请使用采用相同刻度的 *HDBI* 菱形标记。

解码阶段受主机影响明显（HDBI≈0.23-0.24），随后随着批大小和上下文长度扩大，执行越来越以设备为主。

混合专家模型呈现出不同的变化趋势。它在预填充阶段就受主机限制，并且随着解码规模扩大，仍受主机限制。在 H200 上，批大小为=1、序列长度为=512 时，Qwen1.5-MoE-A2.7B 和 OLMoE- 1B/7B 的预填充 HDBI 为≈0.15，其中 $T_{Orchestration}$ 远大于 $T_{DeviceActive}$ （Qwen1.5-MoE-A2.7B 为 448.8 毫秒对 75.9 毫秒；OLMoE-1B/7B 为 273 毫秒对 46.1 毫秒）。相同条件下的解码（m=10）显著拉大了这一差距：Qwen1.5-MoE-A2.7B 的 $T_{Orchestration}$ 达到 895.5 毫秒，HDBI 为=0.15；而 OLMoE-1B/7B 达到 1655 毫秒，HDBI 为=0.1。更重要的是，与稠密模型解码不同，在批大小为=4、序列长度为=512，批大小为=1、序列长度为=4096，以及批大小为=4、序列长度为=4096 时，混合专家解码仍未恢复到均衡状态：主机编排仍占主导地位，且在整个测试范围内，HDBI 始终远低于稠密模型数值。因此，主要的扩展趋势并非只是解码速度较慢，而是混合专家模型在更广的范围内持续受主机限制。

关键结论 #3。 $T_{Orchestration}$ 而 *HDBI* 共同揭示：稠密模型在小规模解码时会显现出主机开销，但随着工作负载规模增大，主导因素又转向设备；混合专家模型则从预填充到解码始终受主机限制，而且在较大的解码配置下仍然如此。关键优化方向取决于受限因素的变化趋势：对于长上下文稠密模型的解码，应减少设备工作量；对于混合专家模型，则应减少路由和启动碎片化带来的开销。

即时执行与 FlashAttention-2 的对比。图 9 使用 *TaxBreak* 对比了 H200 上 Llama-3.2-1B 的即时执行与 FlashAttention-2（FA2） [11] 。在较小配置（批大小=1/序列长度=512）下，FA2 将端到端延迟从 19.5 毫秒降至 18.1 毫秒（7.2%），并将 $T_{Orchestration}$ 从 9.8 毫秒降至 9.1 毫秒（7.1%）。尽管 FA2 降低了绝对延迟，但其减少设备工作量的速度快于降低主机开销的速度，导致 HDBI 从 0.38 降至 0.33，尽管$T_{Orchestration}$ 和 $T_{DeviceActive}$ 的绝对值均有所改善。

在更大的配置（批大小=8/序列长度=2048）下，性能影响要大得多：FA2 降低了端到端延迟

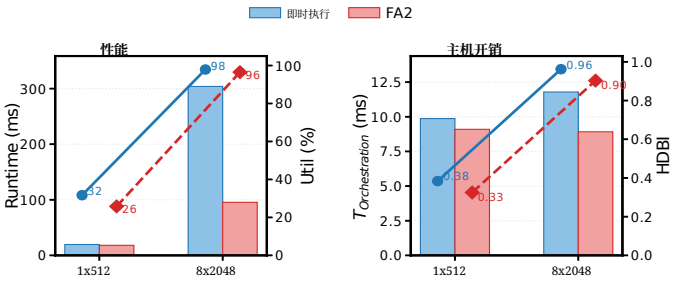


图 9：**H200 上 Llama-3.2-1B 的即时执行与 FlashAttention-2 对比**。柱状图显示端到端运行时间和 $T_{Orchestration}$；折线显示 BS=1/SL=512和 BS=8/SL=2048下的 GPU 利用率和 HDBI。

从 303.9 毫秒降至 95.5 毫秒（68.6%），同时还将 $T_{Orchestration}$ 从 11.7 毫秒降至 8.9 毫秒（24%）。两种模式下 GPU 利用率实际上都保持饱和（即时执行 97.9% 对比 FA2 96.5%），这表明在长上下文下的主要收益并非占用率提升，而是每个词元的设备活跃工作大幅减少——FA2 的分块在线 softmax 内核避免将完整的 $N \times N$ 注意力矩阵物化到高带宽内存，通过消除中间数据回读往返大幅降低内存流量。HDBI 的下降（从 0.96 到 0.90）反映出 FA2 削减 $T_{DeviceActive}$ 的速度快于主机编排，尽管两者的绝对值都有所改善，主机可见占比却因此扩大。这一成对结果说明了*TaxBreak*的预期用途：优化之后，仅看受限度比率可能显得反直觉，而分解表明 FA2 的成功主要来自设备侧优化，而非主机开销降低。

FA2 也体现了*TaxBreak* 诊断提出的内核融合方案：将多内核注意力序列合并为单个融合内核后，FA2 在 (BS=1/SL=512) 下将调度次数从 850 降至 791（−7%），在 (BS=8/SL=2048) 下从 903 降至 731（−19%）。相应的$N \cdot T_{sys}^{floor}$项分别减少了 0.3 毫秒和 0.8 毫秒，这与消除的内核启动次数 $\times\, T_{sys}^{floor}$ 相符。

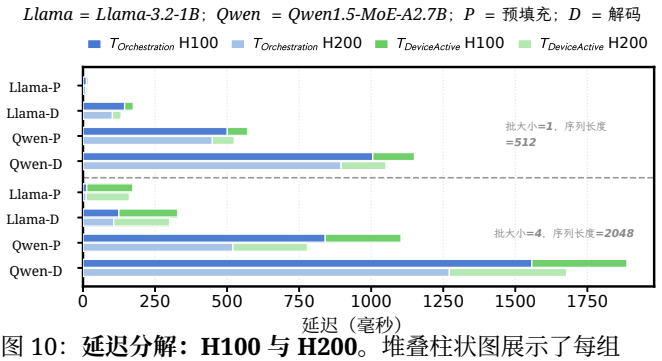


图 10：**延迟分解：H100 与 H200**。堆叠柱状图展示了每组 H100/H200 中的 $T_{Orchestration}$ （蓝色）和 $T_{DeviceActive}$ （绿色）。较深色的 = H100；较浅色的 =H200。$T_{Orchestration}$ 在 H200 上始终更短，而$T_{DeviceActive}$ 相当或略长。解码是在 10 个输出词元上的聚合延迟。

关键结论 #4. *FlashAttention-2* 改善了主机端可见的编排和端到端延迟，但在序列长度较大时，主要收益来自减少设备端的注意力计算，而非消除主机端开销。*TaxBreak* 和 *HDBI* 明确体现了这一区别：*FA2* 仅小幅降低 $T_{Orchestration}$ ，而 *GPU* 计算的大幅减少则带来了运行时间的显著缩短。

VI. CPU单线程性能影响

即时执行模式调度是单线程的，因此 $T_{Orchestration}$会随 N 内核数量在一个 CPU核心上扩展。我们的两个平台提供了有用的受控对比。两者均采用 Hopper 代 GPU，但搭配了不同的主机 CPU：H100 搭配 Intel Xeon 8480C（Sapphire Rapids，2.0/3.8 GHz 睿频），H200 搭配 Intel Xeon Gold 6538Y+（Emerald Rapids，2.2/4.0 GHz 睿频）。H200 GPU 以更低的时钟频率运行（1785 MHz 对比 1980 MHz， −9.9%）；端到端加速不能归因于更快的 GPU计算。 $T_{Orchestration}$/ $T_{DeviceActive}$ 分解使我们能够在相同配置（6 个 CPU 核心、32 GB、单个 GPU）下，将主机调度效应与设备端效应区分开来。

图 10 对比了$T_{Orchestration}$ 和$T_{DeviceActive}$（BS=1/SL=512 和 BS=4/SL=2048）在 Llama-3.2-1B 和 Qwen1.5-MoE-A2.7B 上的表现。主要有三点发现：

1)编排开销与 CPU 单线程速度相关。 $T_{Orchestration}$ 在 H200 上低 10-29%。由于 GPU 相同，且 H200 的 GPU 时钟频率更低，因此这一降幅归因于 Emerald Rapids 上更快的主机端调度。 $T_{DeviceActive}$ 在 H200 上相当，排除了 GPU 内存带宽是性能提升来源的可能。2) **解码会放大 CPU 的影响。** 对于 Llama-3.2-1B，解码使 $T_{Orchestration}$ 在 H200 上降低 29%，而预填充阶段降低 14%，这是因为解码会发起更多内核启动（8,437 次，对比 850 次）。因此，单次调度节省在解码阶段的累积效果要强得多。3)**对于主机受限型混合专家，CPU 性能提升超过 GPU 性能损失。**Qwen1.5-MoE-A2.7B 在 H200 上的（$T_{DeviceActive}$）性能慢 8%，但端到端速度快 13-14%。

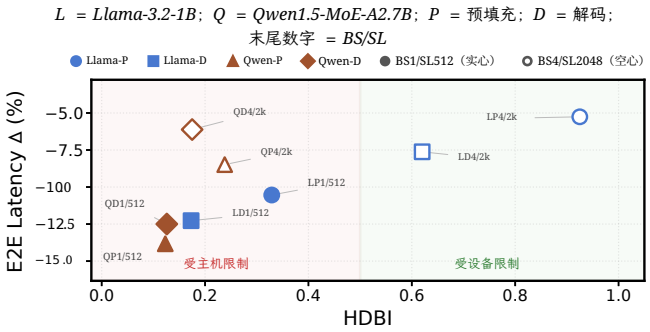


图 11：**端到端延迟增益（H100→H200）与 HDBI 的对比**。标记形状表示模型×阶段；实心标记对应 BS=1/SL=512，空心标记对应 BS=4/SL=2048。受主机限制的点增益最大；受设备限制的点增益较小。解码延迟为 10 个输出词元的总体延迟。

> 使用 HDBI≈0.12 时， $T_{Orchestration}$ 比$T_{DeviceActive}$更大，因此主机端的性能提升占主导，抵消了设备端的性能损失。稠密 Llama-3.2-1B 的端到端提升较小，因为它受主机限制较少。

这一关系在更高负载下仍然成立，但其影响受 HDBI 调节（图 11）。Llama-3.2-1B 在 BS=4/SL=2048 下的预填充受设备限制严重（HDBI=0.93），因此 $T_{Orchestration}$ 降低 13% 仅带来 5% 的端到端增益。Qwen1.5-MoE-A2.7B 在相同条件下仍受主机限制（HDBI=0.24），降低 $T_{Orchestration}$ 仍可使延迟增益 8%。解码阶段也呈现这一规律：受主机限制的点从更快的 CPU 中获益最大，而受设备限制的点因 $T_{DeviceActive}$ 占主导而增益减弱。

关键结论 #5。对于受主机限制的大语言模型推理，尤其是混合专家解码，*CPU*单线程性能是一项一阶设计参数。更快的主机 *CPU* 可将$T_{Orchestration}$ 降低 *10–29%*，即使搭配时钟频率较低的 *GPU*，也能改善端到端延迟。随着工作负载越来越受设备限制，这种影响会减弱。

VII. 结论与展望

本文提出了 *TaxBreak* 方法论，通过跟踪将主机可见推理开销分解为框架转换、CUDA 库转换和启动路径开销。在 NVIDIA H100 和 H200 系统上针对稠密和混合专家大语言模型工作负载进行的评估表明，以往的方法不足以识别关键性能瓶颈和潜在优化目标。*TaxBreak* 能够识别主要优化机会究竟位于软件栈、启动路径还是设备端。HDBI 提供了一种诊断性受限性指标，可用于指导优化。

本研究存在一些局限性。该方法论是在 NVIDIA GPU 上评估的，并依赖 CUDA 跟踪接口。此外，对于高度动态、经过自动调优或同步密集型内核，基于重放的归因可能不够准确。另外，HDBI 是一种诊断性工作比率，而非优化目标本身，应结合绝对延迟和 *TaxBreak* 分解结果进行解读。

对于主机受限工作负载，CPU单线程性能是一个一阶参数：更快的单核执行可降低

$T_{Orchestration}$ 10-29%，而增加核心数则无济于事。跨平台比较（H100 与 H200，第 VI 节）表明，对于混合专家推理，即使搭配时钟频率较低的 GPU，更快的主机 CPU 也能将端到端延迟降低 11-14%。这一效果在预填充和解码阶段均持续存在，并且会随着 HDBI 升至 $\approx$0.3 以上而减弱。未来工作将聚焦于更多样化的 AI 工作负载和平台（英伟达 GB200/GB300 和 AMD MI300A/MI300X）。

参考文献

[1] Josh Achiam、Steven Adler、Sandhini Agarwal、Lama Ahmad、Ilge Akkaya、Florencia Leoni Aleman、Diogo Almeida、Janko Altenschmidt、Sam Altman、Shyamal Anadkat 等。GPT-4 技术报告。arXiv 预印本 arXiv:2303.08774，2023 年。[2] Amey Agrawal、Nitin Kedia、Jayashree Mohan、Ashish Panwar、Nipun Kwatra、Bhargav S Gulavani、Ramachandran Ramjee 和 Alexey Tumanov。Vidur：用于大语言模型推理的大规模仿真框架。《机器学习与系统会议论文集》，6：351–366，2024 年。[3] Amey Agrawal、Nitin Kedia、Ashish Panwar、Jayashree Mohan、Nipun Kwatra、Bhargav Gulavani、Alexey Tumanov 和 Ramachandran Ramjee。借助 {吞吐量—延迟} 权衡，驾驭 {大语言模型} 推理：{Sarathi-Serve}。载于第 18 届 USENIX 操作系统设计与实现研讨会（OSDI 24），第 117–134 页，2024 年。[4] Osayamen Jonathan Aimuyo、Byungsoo Oh 和 Rachee Singh。Flashd-moe：在单一内核中实现快速分布式混合专家。arXiv 预印本 arXiv:2506.04667，2025 年。[5] Jason Ansel、Edward Yang、Horace He、Natalia Gimelshein、Animesh Jain、Michael Voznesensky、Bin Bao、Peter Bell、David Berard、Evgeni Burovski 等。PyTorch 2：通过动态 Python 字节码转换和图编译加速机器学习。载于第 29 届 ACM 编程语言与操作系统架构支持国际会议论文集，第 2 卷，2024 年。[6] James Bradbury、Roy Frostig、Peter Hawkins、Matthew James Johnson、Chris Leary、Dougal Maclaurin、George Necula、Adam Paszke、Jake VanderPlas、Skye Wanderman-Milne 等。JAX：Autograd 和 XLA。《天体物理学源代码图书馆》，页码 ascl–2111，2021 年。[7] Michael Buch、Zahra Azad、Ajay Joshi 和 Vijay Janapa Reddi。移动 SoC 中的 AI 税：智能手机机器学习的端到端性能分析。载于 2021 年 IEEE 系统与软件性能分析国际研讨会（ISPASS），第 96–106 页。IEEE，2021 年。[8] Mark Chen。评估基于代码训练的大语言模型。arXiv 预印本 arXiv:2107.03374，2021 年。[9] Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Yan、Haichen Shen、Meghan Cowan、Leyuan Wang、Yuwei Hu、Luis Ceze 等。{TVM}：用于深度学习的自动化 {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 18），第 578–594 页，2018 年。[10] Krishna Teja Chitty-Venkata、Sylvia Howland、Golara Azar、Daria Soboleva、Natalia Vassilieva、Siddhisanket Raskar、Murali Emani 和 Venkatram Vishwanath。MoE-Inference-Bench：混合专家大语言模型和视觉模型的性能评估。载于高性能计算、网络、存储与分析国际会议 SC'25 研讨会论文集，2025 年。[11] Tri Dao。FlashAttention-2：通过更优的并行性和工作划分加速注意力。arXiv 预印本 arXiv:2307.08691，2023 年。[12] Tri Dao、Dan Fu、Stefano Ermon、Atri Rudra 和 Christopher R´e。FlashAttention：具备 I/O 感知能力的快速、内存高效的精确注意力。载于《神经信息处理系统进展》，35：16344–16359，2022 年。[13] Michael Davies、Neal Crago、Karthikeyan Sankaralingam 和 Christos Kozyrakis。Liminal：探索大语言模型解码性能的前沿。arXiv 预印本 arXiv:2507.14397，2025 年。[14] Jared Fernandez、Jacob Kahn、Clara Na、Yonatan Bisk 和 Emma Strubell。框架税：自然语言处理研究与部署中的推理效率差异。arXiv 预印本 arXiv:2302.06117，2023 年。[15] Aaron Grattafiori、Abhimanyu Dubey、Abhinav Jauhri、Abhinav Pandey、Abhishek Kadian、Ahmad Al-Dahle、Aiesha Letman、Akhil Mathur、Alan Schelten、Alex Vaughan 等。Llama 3 模型家族。arXiv 预印本 arXiv:2407.21783，2024 年。

[16] Daya Guo、Dejian Yang、Haowei Zhang、Junxiao Song、Ruoyu Zhang、Runxin Xu、Qihao Zhu、Shirong Ma、Peiyi Wang、Xiao Bi 等。《Deepseek-R1：通过强化学习激励大语言模型的推理能力》。*arXiv* 预印本 *arXiv:2501.12948*，2025。[17] Quzhe Huang、Zhenwei An、Nan Zhuang、Mingxu Tao、Chen Zhang、Yang Jin、Kun Xu、Liwei Chen、Songfang Huang 和 Yansong Feng。《更难的任务需要更多专家：混合专家模型中的动态路由》。*arXiv* 预印本 *arXiv:2403.07652*，2024。[18] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Sheng、Lianmin Zheng、Cody Hao Yu、Joseph Gonzalez、Hao Zhang 和 Ion Stoica。《借助 PagedAttention 高效管理大语言模型服务的内存》。载于第 *29* 届操作系统原理研讨会论文集，第 611–626 页，2023。[19] Jinhao Li、Jiaming Xu、Shan Huang、Yonghua Chen、Wen Li、Jun Liu、Yaoxiu Lian、Jiayi Pan、Li Ding、Hao Zhou 等。《大语言模型推理加速：全面的硬件视角》。*arXiv* 预印本 *arXiv:2410.04466*，2024。[20] Aixin Liu、Bei Feng、Bing Xue、Bingxuan Wang、Bochao Wu、Chengda Lu、Chenggang Zhao、Chengqi Deng、Chenyu Zhang、Chong Ruan 等。《DeepSeek-V3 技术报告》。*arXiv* 预印本 *arXiv:2412.19437*，2024。[21] Deepak Narayanan、Mohammad Shoeybi、Jared Casper、Patrick LeGresley、Mostofa Patwary、Vijay Korthikanti、Dmitri Vainbrand、Prethvi Kashinkunti、Julie Bernauer、Bryan Catanzaro 等。《使用 Megatron-LM 在 GPU 集群上高效训练大规模语言模型》。载于高性能计算、网络、存储与分析国际会议论文集，第 1–15 页，2021。[22] Bo Pang、Erik Nijkamp 和 Ying Nian Wu。《使用 TensorFlow 进行深度学习：综述》。教育与行为统计学杂志，45(2):227–248，2020。[23] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga 等。《PyTorch：一种命令式、高性能的深度学习库》。神经信息处理系统进展，32，2019。[24] Ritik Raj、Hong Wang 和 Tushar Krishna。《智能体人工智能的 CPU 中心视角》。*arXiv* 预印本 *arXiv:2511.00739*，2025。[25] Daniel Richins、Dharmisha Doshi、Matthew Blackmore、Aswathy Thulaseedharan Nair、Neha Pathapati、Ankit Patel、Brainard Daguman、Daniel Dobrijalowski、Ramesh Illikkal、Kevin Long 等。《AI 税：人工智能数据中心应用的隐性成本》。*ACM* 计算机系统汇刊（*TOCS*），37(1-4):1–32，2021。[26] Timo Schick、Jane Dwivedi-Yu、Roberto Dess`ı、Roberta Raileanu、Maria Lomeli、Eric Hambro、Luke Zettlemoyer、Nicola Cancedda 和 Thomas Scialom。《Toolformer：语言模型可以自学使用工具》。神经信息处理系统进展，36:68539–68551，2023。[27] Noah Shinn、Federico Cassano、Ashwin Gopinath、Karthik Narasimhan 和 Shunyu Yao。《Reflexion：采用言语强化学习的语言智能体》。神经信息处理系统进展，36:8634–8652，2023。[28] Hugo Touvron、Thibaut Lavril、Gautier Izacard、Xavier Martinet、Marie-Anne Lachaux、Timoth´ee Lacroix、Baptiste Rozi`ere、Naman Goyal、Eric Hambro、Faisal Azhar 等。《LLaMA：开放且高效的基础语言模型》。*arXiv* 预印本 *arXiv:2302.13971*，2023。[29] Octavian Alexandru Trifan、Karthik Sangaiah、Muhammad Awad、Muhammad Osama、Sumanth Gudaparthi、Alexandru Nicolau、Alexander Veidenbaum 和 Ganesh Dasika。《消除多 GPU 性能税：实现高效分布式大语言模型的系统方法》。*arXiv* 预印本 *arXiv:2511.02168*，2025。[30] Prabhu Vellaisamy、Thomas Labonte、Sourav Chakraborty、Matt Turner、Samantika Sury 和 John Paul Shen。《刻画并优化 CPU-GPU 耦合架构上的大语言模型推理工作负载》。*arXiv* 预印本 *arXiv:2504.11750*，2025。[31] Haonan Wang、Xuxin Xiao、Mingyu Yan、Zhuoyuan Zhu、Dengke Han、Duo Wang、Wenming Li、Xiaochun Ye、Cunchen Hu、Hongyang Chen 等。《GPU 上大语言模型推理的系统性特征分析》。*arXiv* 预印本 *arXiv:2512.01644*，2025。[32] Florian Weber、Carola Haering 和 Roland Thomaschke。《通过提高时间可预测性改善人机对话》。人为因素，55(5):881–892，2013。[33] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Karthik R Narasimhan 和 Yuan Cao。《ReAct：在语言模型中协同推理与行动》。载于第十一届国际学习表征会议，2022。

[34] Gyeong-In Yu、Joo Seong Jeong、Geon-Woo Kim、Soojeong Kim 和 Byung-Gon Chun。《Orca：一种用于 {基于 Transformer 的} 生成模型的分布式服务系统》。载于第 *16* 届 *USENIX* 操作系统设计与实现研讨会（*OSDI 22*），第 521–538 页，2022。[35] Longfei Yun、Yonghao Zhuang、Yao Fu、Eric P Xing 和 Hao Zhang。《迈向推理最优的混合专家大语言模型》。*arXiv* 预印本 *arXiv:2404.02852*，2024。[36] Huaifeng Zhang、Mohannad Alhanahnah、Fahmi Abdulqadir Ahmed、Dyako Fatih、Philipp Leitner 和 Ahmed Ali-Eldin。《机器学习系统臃肿且易受攻击》。*ACM* 计算系统测量与分析论文集，8(1):1–30，2024。[37] Huaifeng Zhang 和 Ahmed Ali-Eldin。《机器学习系统中隐藏的臃肿问题》。*arXiv* 预印本 *arXiv:2503.14226*，2025。