KernelForge

面向 GPU 算子优化的自主 LLM Agent 框架

李宇科　　孙嘉晨

摘　要

GPU 算子优化通常需要在性能分析、方案设计、Kernel 实现、正确性验证和反复调参之间持续迭代。传统流程高度依赖专家经验，面临技术门槛高、试错成本大、过程难复现和经验难复用等问题。仅依靠大语言模型生成一次 Kernel 代码，也难以持续回答“优化什么、如何验证以及下一步怎么做”等关键问题。

本文提出 KernelForge，一个面向 GPU 算子优化的 LLM Agent 框架。框架将优化过程组织为 Profile–Plan–Code–Verification–Log 闭环：Agent 通过性能剖析定位瓶颈，制定并实施单步优化，依次完成正确性与性能验证，再将实验结果、失败原因和经验沉淀为可检索的持久化记忆，用于后续优化决策。固定的评测跑道、权限边界和 Hook 机制保护参考实现与测量过程，降低自动化探索中的结果偏差和不可控修改。

KernelForge 将评测基础设施、优化策略和分析工具解耦，并提供面向 CUDA 与 MACA 平台的统一接入方式。系统支持按需调用性能分析、工作负载分析和研究型 Agent，并通过结构化知识库与实验记录实现跨轮次经验积累。初步实验表明，包含完整知识库和 Agent 能力的内部版本在沐曦 C500 GPU 的 KernelBench L3 代表性场景中取得约 $1.5\times$ 的汇总加速。该结果说明，受约束的 Agent 闭环能够将专家驱动的算子优化流程转化为可持续、可比较和可审计的自动化过程。

Project Page: https://liyuke.cn/KernelForge_Page

关键词： AI Agent；GPU Kernel；CUDA/MACA；Harness Engineer

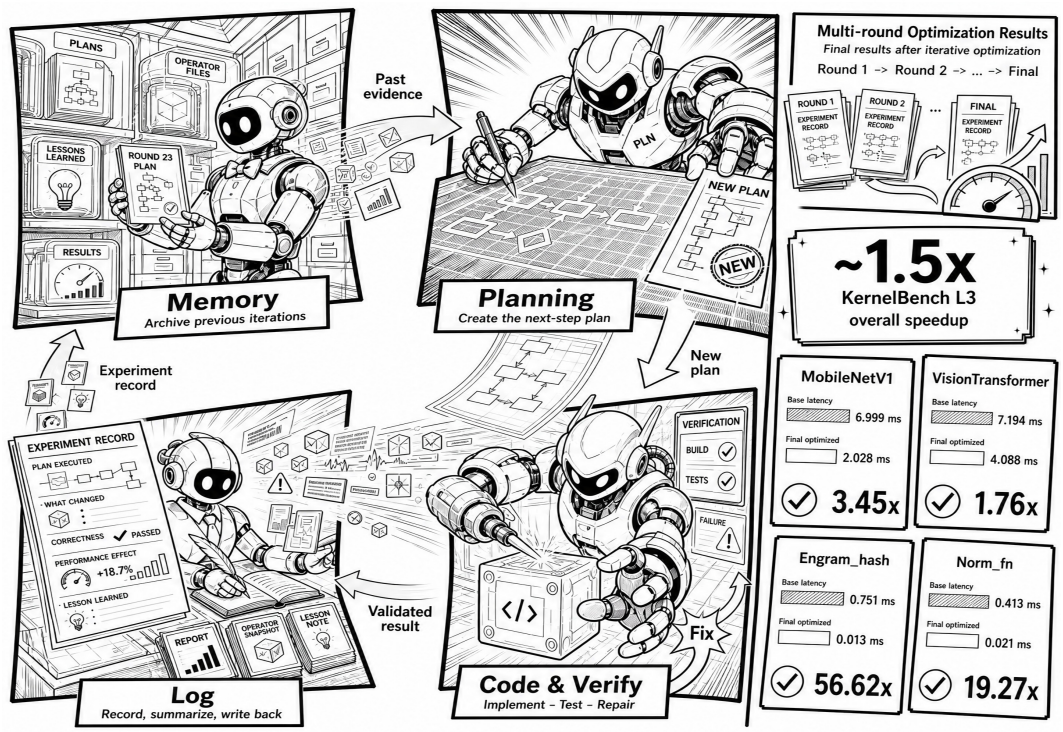

图 1: KernelForge 将 GPU 算子优化组织为 Profile–Plan–Code–Verification–Log 闭环，在固定评测跑道、权限边界和 Hook 机制约束下，依次完成性能分析、单步规划、Kernel 实现、正确性与性能验证，并将实验结果、失败原因和优化经验沉淀为持久化记忆，驱动后续轮次的优化决策；右侧展示多轮迭代后代表性算子的加速结果。

1 介绍

问题背景。 大语言模型（LLM）正从研究原型快速进入搜索、办公、编程、客服、内容创作和智能终端等生产场景，并逐步成为人们获取信息和完成日常任务的重要接口。与离线训练相比，生产推理需要持续响应大量并发请求，其时延、吞吐和单位请求成本会直接影响用户体验与服务可用性。因此，如何在既定硬件上获得更高的有效算力，已经成为 AI 应用规模化落地的基础问题。

当前高性能 GPU 算力仍然稀缺：高端 NVIDIA GPU 价格高、获取周期长，且国际供应与出口管制进一步提高了国内获得先进算力及构建稳定软硬件生态的难度。在此背景下，国产 GPU 厂商正持续补充国内 AI 算力供给；但不同硬件架构、编译器、运行时和软件栈之间存在差异，模型与算子不能仅依赖源代码层面的迁移就获得理想性能。面向 NVIDIA CUDA 与国产 GPU 平台的高效算子实现和适配，因而是提升算力利用率、降低推理成本的重要环节。

深度学习模型的端到端性能通常由大量 GPU 算子共同决定。一个看似简单的算子，实际耗时可能来自计算 Kernel、全局内存访问、layout 转换、临时张量分配、Kernel launch 以及

框架调度。在 LLM 推理中，矩阵计算、注意力相关计算、归一化、激活、布局转换和内存搬运等操作都会累积为端到端开销。对于固定输入形状或特定硬件，通用实现往往没有充分利用数据布局、维度大小和算子序列中的结构信息。

算子优化不仅是局部的性能工程，也是在改进承载 AI 模型训练和推理的底层执行能力。若能够让 AI 在明确的约束下自动分析瓶颈、提出方案、实现变更、验证正确性并积累实验经验，便可将稀缺的专家优化经验沉淀为可复用的系统能力。以 AI 自动优化 AI 算子，是推动 AI 基础设施持续改进、实现 AI 自进化的重要组成部分；其前提是优化过程必须可度量、可验证、可追溯，而不是只依赖一次性的代码生成。

传统优化流程通常包括以下步骤：

1. 读取参考实现并确认语义；
2. 通过 benchmark 和 profiler 判断计算、访存或启动开销的主要来源；
3. 设计 tiling、向量化、融合、共享内存或数据布局等优化方案；
4. 编写 CUDA/MACA C++ 或 Triton Kernel；
5. 进行编译、正确性验证、性能测量和回退；
6. 记录实验结果，避免重复尝试失败方向。

其中第 2、3、5、6 步需要大量经验判断。仅让 LLM 生成一段 Kernel 代码，并不能解决“优化哪个目标”“怎样证明结果可信”和“失败后下一轮做什么”等问题。KernelForge 将上述判断与约束组织为可持续运行的优化闭环，使 LLM 能够在统一评测条件下持续迭代算子实现。

设计目标。 KernelForge 的设计遵循以下五项原则：

- **闭环优化：**每轮均经历分析、规划、实现、验证、测量与记录，而非仅生成一次候选代码。
- **公平比较：**固定参考实现、评测入口、环境变量与日志位置，使性能差异仅归因于受控的优化实现。
- **边界可控：**限制 Agent 修改基线、评测逻辑和环境依赖，防止通过绕过统一入口获得虚假结果。
- **经验可积累：**将成功、失败和回退实验保存为结构化产物，供后续检索、规划与复盘。
- **流程可迁移：**解耦算子接入与优化基础设施，使同一套流程可服务不同算子，并兼容 NVIDIA 与 Metax/MACA 平台。

本文的核心贡献如下：

1. **提出面向 GPU 算子优化的受约束 Agent 闭环**。将性能分析、优化规划、代码实现、正确性验证、性能测量和实验记录组织为连续迭代过程，使 LLM 的作用从一次性生成候选代码扩展为面向可测量优化目标的持续决策。
2. **构建可比较、可审计的优化评测机制**。通过分离 Harness Engineer 与 Loop Engineer 的职责，固定语义基线、评测入口、环境依赖和权限边界，并将正确性、性能、回退及日志记录纳入统一流程，为性能提升提供可信的验证依据。

方向	主要自动化对象	对环境/基线的约束	历史实验记忆
手工 CUDA/Triton	Kernel 实现	取决于工程师规范	通常依赖人工记录
编译器/自动调优	调度或参数搜索	搜索空间和编译器固定	通常由代价模型管理
KernelBench	统一算子任务	评测任务固定	主要用于结果比较
通用 LLM Agent	代码和工具调用	依赖提示词或平台权限	可能丢失中间过程
KernelForge	**算子优化决策与实现闭环**	**固定 Harness、统一评测入口与权限边界**	**Wiki 与结构化实验记忆，支持失败复盘**

表 1: KernelForge 与相关方向的定位对比。

3. **设计跨平台的经验积累与优化支持机制**。将 profiling、工作负载分析、长期实验记忆和结构化 Kernel Wiki 接入优化循环，支持 NVIDIA CUDA 与 Metax/MACA 平台上的算子优化与适配，降低重复试错成本并促进优化经验复用。

2 相关工作

CUDA C++ 允许开发者直接控制线程组织、内存层次和同步策略，是高性能算子实现的基础 [1]；Triton 则以更高层的块级编程模型降低了编写 GPU Kernel 的门槛 [2]。两者都能获得较高性能，但仍要求工程师自己完成瓶颈分析、参数探索和回归验证。KernelForge 不替代这些实现技术，而是让 Agent 能在固定评测闭环中调用和比较它们。

TVM、AutoTVM、Ansor 和 TensorIR 等工作将算子表达、搜索空间和代码生成结合起来，能够自动探索 tile、线程和调度参数 [3, 4, 5]。这类系统通常依赖明确的搜索空间、代价模型或编译器中间表示。KernelForge 的关注点更偏向开放式工程循环：Agent 需要先理解工作负载，决定是否融合、重写或切换技术路径，再通过实验记录积累经验。

KernelBench 将大量深度学习算子整理为统一的正确性和性能任务，为比较不同 Kernel 生成方法提供了可重复的测试对象 [6]。KernelForge 将 KernelBench 算子接入 `solution/model.py` 与 `solution/model_new.py` 的模板，并在其上增加权限、记忆、知识检索和迭代决策能力。这样，评测对象保持稳定，而优化过程可以连续运行多轮。

LLM Agent 编程系统能够读取代码、调用命令、修改文件并根据结果继续行动，但自由度越高，越容易发生越权修改、重复试错或错误解释性能数据。KernelForge 的差异在于把 Agent 放入一个“固定跑道”：工具调用由 Hook 审核，性能测试由统一入口执行，失败实验必须归档，复杂判断交由专门的子 Agent 完成。

表 1 从自动化对象、约束和记忆三个维度概括 KernelForge 与相关方向的差异。

3 方法

KernelForge 由算子层、评测层、流程层、分析层和约束层组成，如图 2 所示。算子层提供参考实现和优化目标；评测层负责统一运行；流程层定义优化命令；分析层提供三个专用子 Agent；约束层通过权限配置和 Hook 保护关键边界。典型优化循环如算法 1 所示。

阶段	输入	Agent 行为	输出/判定
Profile	当前实现、benchmark、profiling、历史经验记忆	判断计算、访存、layout 或启动开销的主要来源	瓶颈结论或 `experiments/profile.md`
Plan	当前瓶颈、Wiki、历史实验	选择一个目标和一个可验证的改动；平台期时调用 `research`	`plan.md` 或当前轮计划
Code	计划和基线接口	只修改 `solution/model_new.py`，实现 CUDA/MACA C++ 或 Triton 路径	候选优化实现
Verification	基线与候选实现	先做正确性 A/B，再做性能 A/B，必要时执行 full profiling	最大误差、延迟、加速比
Log	代码、日志、profiling 和结论	保存快照，更新实验汇总与长期经验	`result.md`、`summary.md`、`LESSONS.md`

表 2: Profile–Plan–Code–Verification–Log 闭环。

算法 1　KernelForge 单轮优化循环

Require: 当前实现、实验记忆、评测跑道
Ensure: 本轮决策及归档实验结果

while 未满足停止条件 **do**
　读取当前实现、实验汇总和历史经验　▷ Profile
　if 瓶颈未知 **then**
　　调用 PROFILER 分析性能瓶颈
　end if
　if 优化目标未知 **then**　▷ Plan
　　调用 WORKLOAD-INSPECTOR 选择目标
　end if
　if 出现平台期或重复失败 **then**
　　调用 RESEARCH 生成下一轮方案
　end if
　选择一个目标和一项可验证的优化改动
　仅修改候选实现，生成优化方案　▷ Code
　执行正确性验证，初始化修复计数 $r \leftarrow 0$　▷ Verification
　while 未通过正确性验证且 r 未达到修复上限 **do**
　　分析误差来源，修复候选方案并重新验证
　　$r \leftarrow r + 1$
　end while
　if 仍未通过正确性验证 **then**
　　回退候选方案
　　归档失败原因、代码快照和实验日志，并进入下一轮　▷ Log
　else
　　执行性能测量，获得候选方案的平均延迟
　　if 结果不确定 **then**
　　　执行完整 profiling 以确认瓶颈和性能变化
　　end if
　　归档代码、日志和结论，并决定保留、回退或切换方向　▷ Log
　end if
end while

表 2 给出了五个阶段的输入和产物。每轮只做一个主要改动，使性能变化能够归因。小于约 5% 的差异不直接视为有效提升，应通过重复运行或 full 模式确认。

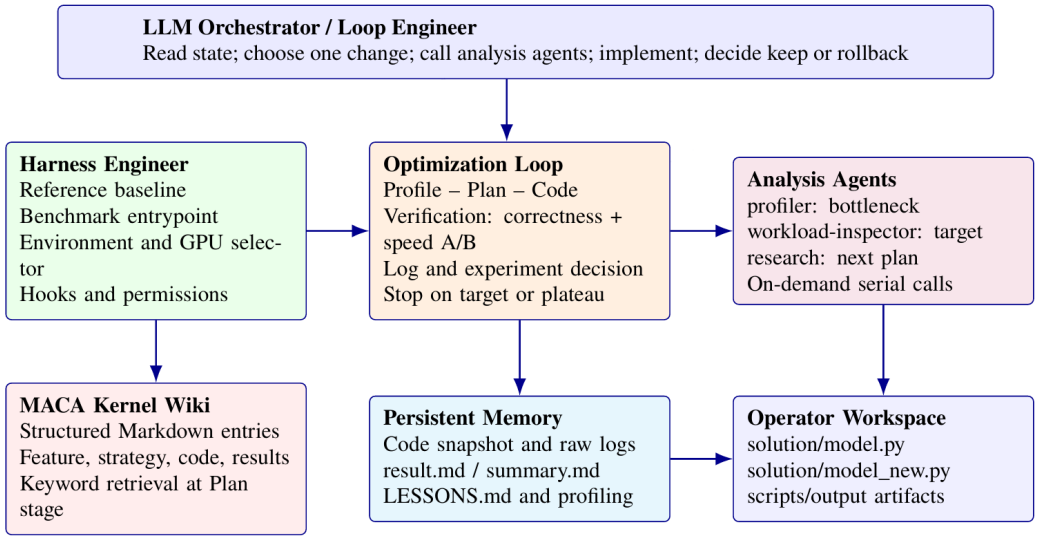

图 2: KernelForge 系统架构。

子 Agent	回答的问题	主要输入	主要输出与触发条件
`profiler`	当前时间主要花在哪里?	`profile_latest.txt`、当前代码、benchmark	`experiments/profile.md`；需要确认瓶颈或重写阶段前调用
`workload-inspector`	下一步最值得优化谁?	输入形状、layout、热点、历史实验	`experiments/workload_profile.md`；多个目标竞争或不确定是否切换时调用
`research`	下一轮应该怎样规划?	长期记忆、profiling 和历史实验	`exp_N/plan.md`；平台期、连续失败、重复失败或方向耗尽时调用

表 3: 三个子 Agent 的输入、输出与触发条件。

主 Agent 负责实现、验证、测量和最终决策；子 Agent 只做分析和规划，不直接修改优化代码。表 3 总结了各 Agent 的输入输出。三者通过文件产物交接，避免关键结论只存在于短期上下文中。

3.1 Harness Engineer

算子接口与基线。 每个算子至少提供以下接口:

```
class Model(torch.nn.Module):
    def forward(self, *inputs):
        ...

def get_inputs():
    return [input_tensor]
```

```
def get_init_inputs():
    return []
```

`solution/model.py` 是语义锚点，定义参考输出；`solution/model_new.py` 是唯一优化目标。运行器基于 PyTorch[7]，用相同初始化参数构造两个模型，并在结构一致时同步权重，从而避免权重差异影响比较。

统一执行链路。 公开实现规定 benchmark 必须从以下入口进入：

```
scripts/run.sh
    -> scripts/env.sh
    -> scripts/run.py
    -> scripts/src/benchmark.py
```

`env.sh` 负责判断 NVIDIA 或 Metax 平台并设置路径；`run.py` 负责设备选择、模型加载、正确性、性能和 profiling；`benchmark.py` 负责具体测量。统一入口同时固定日志位置：`scripts/output/bench_latest.log` 和 `scripts/output/profile_latest.txt`。

正确性和性能判定。 设基线方法与优化后方法的输出分别为 Y_{baseline} 和 $Y_{\text{optimized}}$，最大绝对误差定义为：

$$E_{\max} = \max_i |Y_{\text{baseline},i} - Y_{\text{optimized},i}|, \tag{1}$$

其中，i 遍历输出张量中的全部元素。公开实现默认以 $E_{\max} \leq 10^{-4}$ 作为正确性通过条件；只有通过该条件的候选实现才会进入性能测试。性能测试先预热 20 次，再重复运行 1000 次并计算平均延迟；必要时生成 profiling。加速比定义为：

$$S = \frac{T_{\text{baseline}}}{T_{\text{optimized}}}, \tag{2}$$

式 (2) 中，T_{baseline} 为基线方法在 1000 次正式测量中的平均延迟，$T_{\text{optimized}}$ 为优化后方法在相同条件下的平均延迟。本文仅报告平均延迟。

Hooks 与权限边界。 完整系统在 Agent 调用工具、修改文件和结束会话前进行审核。公开仓库提供的三个代表性 Hook 及其规则见表 4。

这些规则的目标不是限制所有实验自由，而是固定三类不可变条件：语义基线、评测逻辑和依赖环境。

3.2 MACA Kernel Wiki

MACA Kernel Wiki 采用结构化 Markdown 组织专家经验，不把知识库设计成只能由某一名工程师记忆的隐性规则。每条记录至少包含表 5 中的字段。

Plan 阶段根据算子名、输入特征、瓶颈类型和平台关键词检索相关条目，再将候选经验注入计划上下文。Wiki 的作用是缩短“从性能现象到可测试方案”的距离。

Hook	审核对象	典型规则	失败处理
`pre_bash.sh`	Shell 命令	禁止破坏性 git 操作、联网安装依赖、绕过 `scripts/run.sh` 和非法 benchmark 参数	拒绝命令并返回原因
`pre_edit.sh`	文件编辑	禁止修改 `solution/model.py` 和 `scripts/`，只允许修改 `solution/model_new.py`	拒绝写入
`stop.sh`	会话结束状态	检查优化代码是否晚于最新 benchmark/profiling 产物	提示补测，不强制阻止停止

表 4: 公开仓库中的 Hook 约束。

字段	内容
算子特征	输入形状、dtype、layout、数据分布和典型瓶颈
优化方案	融合、tiling、向量化、共享内存、访存或启动优化策略
适用条件	平台、维度范围、stride 和数值精度约束
代表性代码	脱敏后的 CUDA/MACA/Triton 片段或接口模式
实验结果	延迟、加速比、正确性和失败边界

表 5: MACA Kernel Wiki 的最小条目结构。

3.3 持久化记忆系统

KernelForge 使用文件系统保存可审计的原始产物，并用汇总文件提供跨实验检索。单轮实验的最小记录单元为“代码快照 + 指标 + 简要结论”。

失败实验、消融实验和回退实验同样必须记录。这样，Agent 可以在下一轮明确知道某个方向已经试过、为什么失败，以及是否值得用不同的融合边界、线程划分或数据布局重新尝试。

4 实验

实验环境。 表 6 汇总了当前实验口径。公开模板同时保留 NVIDIA 分支，本文的主要结果以 C500/MACA 为主线。

指标与统计方法。 正确性指标为基线方法与优化后方法输出的最大绝对误差，性能指标为模型完成一次 forward 的平均延迟。加速比采用式 (2) 的定义，即基线方法平均延迟与优化后方法平均延迟之比。性能测量前先执行 20 次预热以减小初始化开销的影响，随后重复运行 1000 次, 本文报告这 1000 次测量结果的平均延迟。

4.1 代表性 KernelBench L3 结果

KernelBench 是面向 GPU Kernel 生成与优化的基准测试集，将深度学习工作负载封装为具有统一输入生成、正确性校验和性能测量流程的任务，从而支持在相同基线和评测口径下比较

项目	配置
GPU	沐曦 C500
平台	MACA
MACA/驱动版本	3.3.0.2
PyTorch 版本	2.8.0+metax3.5.3.9
Python 版本	3.10.20
编译器/工具链	mxcc version 1.0.0
输入生成方式	各算子 `get_inputs()`
正确性阈值	最大绝对误差 $\leq 1 \times 10^{-4}$
预热	20 次
quick 测试	100 次迭代
full 测试	1000 次迭代 + profiling

表 6: 实验环境和运行参数。

不同优化方法 [6]。本文沿用其任务定义：优化前后实现使用同一组 `get_inputs()` 生成的输入，并仅在通过正确性校验后纳入性能统计。

KernelBench 按任务复杂度划分为多个层级。相较于主要考察单一基础算子的低层级任务，Level 3（L3）任务由多个算子组成模型或模型模块，通常同时包含计算、访存、布局变换及算子间衔接等开销。因此，L3 不仅检验某个局部 Kernel 的实现效率，也考察优化系统能否识别端到端路径中的主要瓶颈，并在保持结果一致的前提下协调融合、并行化和内存访问等优化决策。

表 7 给出了九个代表性 L3 任务在 C500/MACA 平台上的结果。所有任务的优化后延迟均低于基线，整体加速约为 `1.5x`；其中 MobileNetV1 的加速比最高，达到 $3.45\times$，表明其端到端执行路径具有较大的优化空间。对于 AlexNet、ResNet101 和 DenseNet121DenseBlock 等已具有较高计算密度或较成熟执行路径的任务，仍获得 $1.21\times$–$1.26\times$ 的稳定收益。LeNet5、ResNetBasicBlock 与 VisionTransformer 的加速比介于 $1.65\times$ 和 $1.76\times$ 之间，说明系统能够在不同网络结构下持续发现有效优化机会。

模型/算子	Base(ms)	Optimize(ms)	加速比
LeNet5	1.157	0.673	1.72
AlexNet	55.020	45.471	1.21
ResNetBasicBlock	2.455	1.488	1.65
VisionTransformer	7.194	4.088	1.76
ResNet101	29.052	23.027	1.26
DenseNet121DenseBlock	20.100	16.618	1.21
MobileNetV1	6.999	2.028	3.45
MobileNetV2	13.446	9.742	1.38
MLP	9.813	7.475	1.31

表 7: 部分 KernelBench L3 性能结果。

4.2 DeepSeek 大模型推理算子结果

除 KernelBench L3 外，本文还选取了三个来自 DeepSeek 系列模型推理实现的实际算子，其性能结果如表 8 所示。Engram_hash 来自 DeepSeek Engram 条件记忆模块，执行 N-gram 多哈希嵌入表索引生成；该步骤决定条件记忆的查表地址，是以可扩展静态记忆补充模型计算的前置路径 [8]。Indexer 来自 DeepSeek-V3.2-Exp 的 Lightning Indexer（DeepSeek Sparse Attention, DSA）模块，对压缩 KV Cache 计算相关性并选择 Top-k 历史位置；它直接决定长上下文推理中稀疏注意力需要访问的 KV 条目 [9]。Norm_fn 完成残差特征的 RMS 归一化和多分支特征混合；该算子位于层内特征变换路径，需要在每层推理中重复执行。

算子	Base(ms)	Optimize(ms)	加速比
Engram_hash	0.751	0.013	56.621
Indexer	6.545	3.160	2.071
Norm_fn	0.413	0.021	19.273

表 8: DeepSeek 系列大模型推理自定义算子的性能结果。

5 应用场景

KernelForge 面向需要持续优化 GPU 算子的场景。它将性能分析、代码修改、正确性验证和结果记录组织为统一流程，使优化过程更容易重复、比较和积累。

生产 LLM 推理服务优化。 在线 LLM 服务对响应速度和吞吐量有较高要求。KernelForge 可定位模型中的性能热点，并针对关键算子进行优化，从而降低推理延迟和服务成本。

国产 GPU 软件生态适配。 不同 GPU 平台的硬件和软件环境存在差异，已有实现不一定能直接取得理想性能。KernelForge 提供统一的验证和评测流程，支持针对 CUDA、MACA 等平台开发和优化算子。

算子库与模型部署维护。 模型、输入形状和软件版本变化后，原有算子可能出现性能下降。KernelForge 保存代码、测试结果和优化经验，便于重新评测、发现性能回归并复用已有方案。

专家协作与性能研发。 GPU 算子优化依赖较强的专业经验。KernelForge 将评测规则、优化过程和实验记录固定下来，帮助团队成员复现结果、共享经验并减少重复试错。

AI 驱动的基础设施自优化。 KernelForge 使用 LLM 辅助分析、规划和编写候选实现，并通过正确性和性能测试控制优化质量。这使 AI 能够参与改进支撑自身训练和推理的底层算子与软件基础设施。

6 局限与未来工作

6.1 局限性

系统性组件消融。 当前报告没有 Wiki、子 Agent、记忆和 Hook 的独立消融数据，因此无法严格量化每个组件对成功率、迭代轮数、Token 成本或最终加速的边际贡献。后续应固定算子集合，分别关闭一个组件并重复相同预算的实验。

平台与工具链依赖。 Kernel 性能依赖 GPU 架构、驱动、MACA/CUDA 工具链和 PyTorch 版本。当前结果以 C500/MACA 为主，不能直接外推到所有 NVIDIA 或其他 Metax 设备。未来应建立跨平台结果矩阵，并记录编译器和运行时版本。

LLM Agent 的决策成本。 持续调用 Agent、编译 Kernel 和运行 profiling 都会产生时间和 Token 成本。现有系统通过小步修改、实验记忆和按需调用子 Agent 减少无效循环，但尚未给出统一的成本模型。后续可引入预算感知的计划选择和收益/成本估计。

6.2 未来工作

后续将补充组件消融与跨平台评测，量化各模块贡献并验证系统在不同硬件和工具链上的适用性；同时引入预算感知的优化策略，以降低 Agent 调用、编译和 profiling 带来的决策成本。在此基础上，KernelForge 将进一步扩展其优化能力：

- 支持更丰富的 Kernel 模板、自动参数搜索以及多算子融合；
- 将实验产物索引为结构化查询接口，同时保留原始记录以便审计；
- 引入更细粒度的性能回归检测和自动报告生成；
- 扩展至动态形状、多精度和更复杂的端到端模型工作负载。

参考文献

[1] NVIDIA Corporation, "CUDA C++ Programming Guide." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2026. Accessed 2026-08-07.

[2] P. Tillet, H. T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

[3] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation*, pp. 578–594, 2018.

[4] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation*, pp. 863–879, 2020.

[5] Apache TVM, "TensorIR Documentation." `https://tvm.apache.org/docs/`, 2026.

[6] Scaling Intelligence Lab, "KernelBench: A Benchmark for LLM-Generated GPU Kernels." `https://github.com/ScalingIntelligence/KernelBench`, 2025. GitHub repository.

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[8] DeepSeek-AI, "Engram: Conditional Memory via Scalable Lookup." `https://github.com/deepseek-ai/Engram`, 2026.

[9] DeepSeek-AI, "DeepSeek-V3.2-Exp: Boosting Long-Context Efficiency with DeepSeek Sparse Attention." `https://github.com/deepseek-ai/DeepSeek-V3.2-Exp`, 2025.