PARACODEX：一种性能分析引导的自主编码代理，用于可靠的并行代码生成与翻译

Erel Kaplan[1] **Tomer Bitan**[1] **Lian Ghrayeb**[1] **LeChen**[2]**TomYotam**[3] **Niranjan Hasabnis**[3] **Gal Oren**[1,4]以色列理工学院，以色列海法[2]阿贡国家实验室，美国莱蒙特[3] Code Metal，美国[4]斯坦福大学，美国斯坦福 {erel.kaplan, tomerbitan, lian.ghrayeb}@campus.technion.ac.il lechen@anl.gov, {tom, niranjan}@codemetal.ai, galoren@stanford.edu

arXiv:2601.04327v1 [cs.DC] 7 Jan 2026

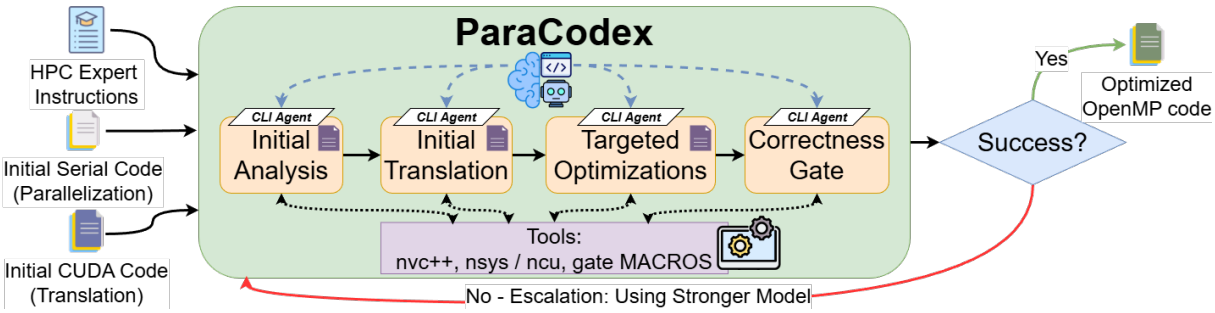


图 1：**ParaCodex：一种可并行化并迁移代码的自主大语言模型代理**。核心思路是将代码生成转变为一种由产物驱动、经工具验证的代理工作流：智能体首先提取瓶颈并制定数据映射计划，然后提出补丁级别的 OpenMP target 编辑，最后将编译器和测试作为严格的正确性门控，并将性能分析器跟踪作为性能信号。通过将要迁移的内容（数据规划）与要卸载的内容（内核翻译）分开，并通过测量形成闭环，PARACODEX 减少了脆弱的一次性输出，并借助结构化计划、日志和性能分析结果，使迭代过程可复现。

摘要

并行编程是高性能计算和人工智能的核心，但要编写出正确且快速的代码仍然极具挑战，尤其是在 OpenMP GPU 卸载中，数据移动和调优占据主导地位。自主编码智能体能够在目标硬件上编译、测试和分析性能，但若缺乏领域脚手架，其生成的代码就会很脆弱。

我们提出了 PARACODEX，这是一套高性能计算工程师工作流，通过分阶段热点分析、显式数据规划、正确性门控和性能分析引导的优化，将基于 Codex 的智能体转变为自主的 OpenMP GPU 卸载系统。我们在 HeCBench、Rodinia 和 NAS 上评估了将串行 CPU 内核翻译为 OpenMP GPU 卸载内核的效果。排除五个内核后，PARACODEX 在全部 31 个有效内核上均取得成功。生成的内核在 25/31 种情况下，相较于参考 OpenMP 实现缩短了 GPU 时间；在 HeCBench 和 Rodinia 上分别实现了 3× 和 5×的几何平均加速比，并且在所有测试套件上均优于零样本 Codex 基线。我们还评估了 CUDA→在 ParEval 上的 OpenMP 卸载翻译，PARACODEX 在仅代码和端到端设置中均保持较高的编译率和验证率 [1]。

1 引言

并行编程是现代高性能计算的核心，但要编写出既正确又快速的并行实现仍然十分艰巨。在实践中，开发者反复面临两类需求：在串行内核中引入并行性，以充分利用 CPU 和图形处理器；以及迁移现有并行代码，在不同编程模型之间转换，以提升可移植性和可维护性。

关键的是，本研究探讨了用于程序转换的工具型大语言模型代码自主智能体：这类系统不会止步于仅生成文本代码，而是会在目标硬件上反复编译、测试并分析其输出，同时利用结构化的中间产物——

[1] ParaCodex代码仓库：https://github.com/Scientific-Computing-Lab/ParaCodex

例如热点分析和数据计划——以及环境反馈，作为首要信号来进行优化（Gottschlich 等人，2018）。

经典自动化工具（如多面体编译器（Bondhugula 等人，2008；Doerfert 等人，2015）可以减少人工工作，但难以处理非仿射循环和含糊的指针别名问题，通常需要专家介入以插入编译指示或解决依赖关系（Harel 等人，2020年；Mosseri 等人，2020年）。相比之下，大语言模型虽然能够生成看似合理的 OpenMP 指令，却缺乏系统级认知，无法高效管理数据移动（Nichols 等人，2024b；Bitan 等人，2025年；Dearing 等人，2025年）。一种常见的失败模式是“抖动”：大语言模型虽然正确地将循环并行化，却未能映射辅助函数或将数据分配提升到循环之外，导致运行时在每次迭代时都隐式地在 CPU 和图形处理器之间来回复制数组，生成的代码虽然功能正确，速度却远慢于串行执行。

与纯文本代码生成不同，自主编码代理（Li 等人，2025年）将代码模型与终端或集成开发环境中的工具使用相结合，从而能够直接在目标机器上迭代编辑、编译和测试（Yang 等人，2024年）。这使开发模式从离线代码补全转变为基于反馈的现场开发。我们聚焦于 OpenMP GPU 卸载（Deakin 和 Mattson，2023年），因为它借助最广泛使用的并行 API 之一，致力于实现现代并行计算的“终极目标”（Valero Lara 等人，2025年；Kadosh 等人，2023a）：利用图形处理器相较于 CPU 的巨大性能优势，而这一优势在过去十年中已得到广泛应用（TOP500，2025年）。通过以 OpenMP 卸载为目标，我们应对三重挑战：(i) 在异构系统间保持可移植性；(ii) 保留 CPU 回退机制（因为 OpenMP 卸载是可选的）；以及 (iii) 提高对智能体推理能力的要求，要求智能体正确管理设备内存和内核启动，而不只是实现线程级并行。我们评估两种输入场景：翻译（串行→OpenMP）和迁移（CUDA→OpenMP），这两者都是该领域最重要的任务之一。（Chen 等人，2024a）。

这些局限促使我们设计了 ParaCodex，这是一种基于 Codex 的智能体工作流（Horikawa 等人，2025年；Vangala 等人，2025年），其模拟了高性能计算工程师的迭代过程（Mondesire 等人，2025年）。为避免对不安全循环过早并行化（这是工具的一类弱点），PARACODEX 强制执行阶段 1（分析）。为避免数据抖动，阶段 2（数据规划）会在代码生成前明确规划内存驻留。最后，为捕捉静态分析无法发现的性能回退，阶段 3（性能分析）通过真实测量闭合反馈环（Peng 等人，2024年）。与测试通过后便停止的方法不同，PARACODEX 会通过反馈明确地针对性能进行优化。

我们在 36 个串行→OpenMP 任务上评估 PARACODEX，这些任务取自 HeCBench（Jin 和 Vetter，2023年）（23 个）、Rodinia（Che 等人，2009年）（7 个）和 NAS（Bailey 等人，1991年；Fridman 等人，2025年）（6 个）。这些广泛使用的基准套件涵盖了从微内核到端到端科学基准测试的各种任务，其提供的实现从经过优化到高度调优不等。根据 NVIDIA Nsight Systems 测量的 GPU 时间（内核执行加数据传输），PARACODEX 在 31/36 个内核上实现了有效的 GPU 卸载（86%；排除的 5 个内核详见 §4.2），为全部 31 个有效内核生成了正确实现，并在 25/31 个案例中（80%）缩短了 GPU 时间，相较于所提供的 OpenMP 实现，在 HeCBench、Rodinia 和高度调优的 NAS 基准套件上，GPU 时间几何平均加速比分别为 3×、5.1× 和 1.08×。作为一项独立的泛化研究，我们在四个 ParEval（Nichols 等人，2024a）内核上评估 CUDA→OpenMP 迁移；在固定构建环境（仅代码）和端到端设置（智能体还必须构建构建流程）下，PARACODEX 均保持较高的编译率和验证率（§4）。

核心问题。 在自动化构建与测试验证的支持下，工具调用型自主编码智能体在多大程度上能够转换和迁移并行代码，并利用性能分析反馈缩小与参考实现之间的性能差距？

研究问题。 为探索这一领域，我们提出四个指导性问题：
1.**研究问题 1 – 基线可行性：** 当代智能体型大语言模型在没有性能分析引导的迭代时，能多有效地将串行代码转换为正确的 OpenMP GPU 卸载代码？ 2.**研究问题 2 – 专业化的益处：** 采用包含结构化中间产物的分阶段专家引导工作流，能否提升稳健性和性能？ 3.**研究问题 3 – 性能达成情况：** 通过性能分析闭环优化，能在多大程度上达到或超越所提供的 OpenMP 实现，且在哪些方面仍存在性能退化？

在多样化的基准测试中，

4.**研究问题 4 – 任务泛化：** 同一工作流能否从串行→OpenMP 翻译推广到 CUDA→OpenMP 迁移?

贡献。 我们的贡献有三点：
1.我们将高性能计算工程实践系统化为可复用的智能体模式：从基于循环分类体系的热点分析，到数据驻留和传输策略选择，再到配置文件指导的调优，并通过基于 Makefile 的正确性门槛将各环节串联起来（研究问题 2、研究问题 3）。
2.我们设计了一套全面的评估方案，在统一的构建与测试框架和零样本 Codex 基线下，对 HeCBench、Rodinia 和 NAS 中的串行→OpenMP 智能体并行化进行评估，并量化其相对于专家实现的稳健性和 GPU 执行时间性能（研究问题 1、研究问题 3）。
3.我们展示了该方法向 ParEval 上 CUDA→OpenMP 翻译任务的泛化能力，并引入旁路检测分析，以揭示伪卸载案例并提出确保设备端工作得以执行的测试框架设计策略（研究问题 4）。

2 相关工作

我们按照从静态并行化方法和学习式并行化方法，到基于大语言模型的并行代码生成与翻译，再到自主编码智能体和反馈驱动的自我优化这一脉络，梳理相关工作。我们从三个方面区分 PARACODEX 与先前工作：（1）编译/测试修复（UniPar、LASSI）能够修复错误，却忽视速度；（2）性能反馈（PerfCodeGen、STOP）能够优化运行时间，却缺乏严格的正确性门槛；（3）由制品驱动的规划——先前的智能体中尚未采用这种做法——会在编码前将推理过程外化。据我们所知，PARACODEX 是首个整合这三个方面的系统。

编译器与学习式并行化。编译器研究探索了通过依赖分析和多面体模型进行自动循环变换（Bondhugula 等人，2008；Doerfert 等人，2015）。这类工具擅长处理仿射循环嵌套，但通常需要专家介入。综述研究分析了源到源 OpenMP 编译器，并指出了其中的陷阱和差异（Harel 等人，2020年；Mosseri 等人，2020年）。近期研究则开始应用学习式成本模型、强化学习和自动调优来优化代码（Chen 等人，2019年；Ahn 等人，2019年；Wu 等人，2023年；Ding 等人，2023年；Merouani 等人，2025年），依靠迭代搜索和测量。机器学习技术可预测并行化机会并建议 OpenMP 指令（Maramzin 等人，2019年；Chen 等人，2023年；Harel 等人，2023年；Kadosh 等人，2024a；Harel 等人，2025年），但尚不能生成完整代码。

面向并行代码的大语言模型。面向特定领域的大语言模型（HPC-Coder、OMPGPT、MonoCoder）专注于并行内核（Nichols 等人，2024b；Chen 等人，2024b；Kadosh 等人，2024b；Schneider 等人，2024；Kadosh 等人，2023b；Chen 等人，2024a）。更通用的模型（CodeGeeX、Meta 的 LLM Compiler）则扩展了覆盖范围（Zheng 等人，2024；Cummins 等人，2024）。Transformer 顾问工具（OMPify、MPI-rical）提供数据驱动指导（Kadosh 等人，2023c；Schneider 等人，2023、2024）。然而，这些方法通常只提供一次性输出，不包含迭代反馈。

自主编码智能体。近期研究描述了自主编码代理：这类系统将代码模型与开发者工具结合，通过迭代循环开展工作（Li 等，2025年）。相关示例包括命令行界面助手和集成开发环境集成（GitHub Copilot、OpenAI Codex、Cursor）。这对并行代码尤为重要，因为其正确性需要构建和测试框架，而性能取决于具体硬件。Codex 已在代码生成基准测试中接受评估，并表现出色。[2] 我们基于 Codex 构建 PARACODEX。仅有工具访问权限并不足够：缺乏领域结构时，工作流可能会十分脆弱。以往的流水线侧重于编译修复和正确性（Bitan 等，2025年；Dearing 等，2025年；Chen 等，2025年），而 PARACODEX 则通过明确的产物、正确性门槛和性能分析指导的优化，将高性能计算工程师的工作流程付诸实践。

基于大语言模型的翻译流水线。 UniPar 和 LASSI 等框架研究利用大语言模型翻译并行代码，包括串行→OpenMP（Bitan 等，2025年；Dearing 等，2025年；Chen 等，2025年）。它们实现了编译/测试驱动的修复：编译器错误和测试失败会触发迭代式修复，从而提高编译成功率和正确性。然而，它们缺少：（i）通过性能分析获取执行/性能反馈以指导优化，以及

[2] https://openai.com/index/introducing-gpt-5-2-codex/

（ii）由制品驱动的结构化计划，在修改代码之前将推理过程外化。互补研究量化了大语言模型在高性能计算内核生成方面的能力（Godoy 等，2023年；Valero-Lara 等，2023年；Cui 等，2025年；Bolet 等，2025年）。

自我精炼与反馈驱动的优化。智能体研究探索通过审慎搜索与精炼来合成程序（Yao 等人，2023年；Shinn 等人，2023年；Singhal 等人，2025年；Madaan 等人，2023年；Jimenez 等人，2024年）。PerfCodeGen 展示了执行驱动的精炼方式，即将性能指标反馈回来（Peng 等人，2024年）；自我训练优化器（STOP）则为大语言模型搭建框架，使其递归地优化程序（Zelikman 等人，2023年）。这些系统展现了执行反馈的价值，但并未在每个阶段实施正确性门控，也不会生成结构化中间工件（例如数据计划、优化计划）来使推理立足于领域约束。PARACODEX 在统一的高性能计算工程工作流中整合了这三种机制——编译/测试修复、性能分析反馈和制品驱动规划。

3 PARACODEX 智能体概述

PARACODEX 是一种引导式代理工作流，在该工作流中，模型会迭代使用标准开发工具。智能体首先分析输入程序，以识别性能关键循环或内核，然后生成 OpenMP GPU 版本。它将代码与串行参考实现进行编译和执行，并利用编译错误和数值不匹配作为结构化反馈，驱动针对性修复，直至实现正确性。

随后，工作流转向性能调优。智能体使用性能分析器分析执行情况，并应用有针对性的优化，以进一步提升性能。正确性驱动的修复与基于性能分析的调优相结合，可实现稳健的自动化优化。

3.1 基线比较系统。

在本文中，我们使用术语 基线指代一种零样本 Codex CLI 智能体 [3]，它通过单次处理直接将输入程序转换为 OpenMP GPU 卸载代码。基线和 PARACODEX 均在相同的构建与正确性测试框架下进行评估；此外，PARACODEX 还使用分阶段提示和性能分析引导的优化。这一设置保持底层模型不变，从而单独评估工作流支架的影响。基线的更多细节见附录 B.6。

3.2 流水线阶段

PARACODEX 会执行一个三阶段工作流，如图 1所示。（详见附录A，了解针对特定套件所做的调整。）

1.**分析：循环分类与数据特征描述**。检查输入代码，识别候选循环并按计算权重排序。计算权重定义为估算的操作次数（迭代次数 × 操作次数/迭代） $O(N)$，以区分关键路径与初始化代码。生成`analysis.md`，使用分类体系（A–G 类）对循环进行分类，并记录数据属性、依赖关系和风险因素（附录B）。理由：在修改代码前将推理过程外化，可以降低过早转换存在隐藏风险因素（例如归约、递推关系）的循环的可能性，避免由此悄然破坏正确性。

2.**GPU 卸载 + 数据策略：通过显式数据计划确保正确性**。在插入 pragma 之前，智能体会根据循环分类选择数据管理策略：(A)作用域区域对标准内核使用显式 target 数据指令，以尽量减少数据传输；(B)异步流水线使用 nowait 实现计算与通信的重叠；(C)全局设备状态为迭代求解器分配持久的设备内存（omp_target_alloc），避免重复映射。它会编写 data_plan.md，指定数组驻留位置和函数卸载状态，并优先卸载关键循环（即计算权重较高的循环）。这一显式规划步骤可以降低常见“抖动”问题的发生概率；在这类问题中，未映射的辅助函数会在每次迭代时触发隐式的主机与设备间传输。

3.**性能调优：基于性能分析的优化**。功能验证后，智能体使用 NVIDIA Nsight Systems 对程序进行性能分析，提取内核耗时、数据传输量和 API 开销。它会生成 optimization_plan.md，对瓶颈进行排序并提出有针对性的修复方案（例如融合相邻内核以降低启动延迟、合并循环以提高占用率，以及将数据传输移出迭代循环）。系统严格执行性能回退即撤销的策略：如果图形处理器时间

[3] https://platform.openai.com/docs/models/gpt-5.1-codex

恶化超过 10%，则会自动回滚更改。

4.**正确性门控**。 每个阶段之间都会调用一个正确性门控代理 。该代理使用 `gate.h` 对代码进行插桩——这是一个轻量级头文件，可在程序的关键位置计算校验和与范数，从而定位差异并指导修复。随后，它使用手动编写的基于 Makefile 的测试框架，将候选代码与输入串行代码进行编译和运行。如果验证失败（通过退出代码或数值不匹配判定），它便会修正代码。（附录 B.3）。

出于篇幅考虑，附录 B.5中保留了一个具体示例，展示热点分析、数据计划和优化的端到端工作流程。

3.3 工具集成

该流水线使用智能体可访问的外部工具。NVIDIA HPC SDK 的编译器诊断可揭示问题；基于 Makefile 的测试框架可验证正确性；NVIDIA Nsight Systems 可进行性能诊断；系统元数据则从 system_info.txt 中读取。该设计将大语言模型驱动的生成与确定性的工具反馈相结合，以实现迭代修复和调优。

3.4 提示工程策略

PARACODEX 采用多阶段提示，将模型的推理过程组织为分析、翻译和优化阶段。每个阶段都由特定模板驱动，在约束输出空间的同时保留领域灵活性。

1.**分析阶段**。 模型会生成结构化的热点分析（附录 B.1），使用分类体系（A–G 类）描述循环，并捕捉并行化约束。该提示将定性推理转化为离散的中间表示，从而在转换前揭示任何风险因素。

2.**翻译阶段**。 利用结构化分析，提示将每种循环类别映射到一组受限的有效 OpenMP 构造（附录 B.2），以尽量减少映射错误，例如遗漏归约。

3.**优化阶段**。 此阶段实现测量引导的搜索（附录 B.4）。提示将 Nsight Systems 性能分析结果与优化手段配对，以进行有针对性的修改，并仅保留能够提升性能的修改。

套件	已尝试	有效	已改进
HeCBench	23	21	18/21 (86%)
Rodinia	7	6	4/6 (67%)
NAS	6	4	3/4（75%）
ParEval	4	4	–
串行→OpenMP	**36**	**31**	**25/31（80%）**

表 1：**各套件的结果统计**。 尝试数：尝试过的内核； 有效 *GPU*：编译 + 通过正确性检查且设备端执行量显著；性能提升： PARA- CODEX 在有效 GPU 内核中降低 GPU 时间。ParEval 将 CUDA→OpenMP 单独报告。

4 实验评估

4.1 实验设置

环境与协议。我们使用配备 NVIDIA HPC SDK 25.7（nvc++）的 NVIDIA RTX4060Laptop GPU（8GB）和 i9-13905H CPU。我们报告通过 NVIDIA Nsight Systems（v2024.5）测得的 *GPU* 时间，其计算方法为将 CUDA 内核时间（cuda_gpu_kern_sum）与设备传输时间（cuda_gpu_mem_time_sum）相加，并排除 CUDA API 和运行时开销。每种配置进行三次性能分析，并报告平均值。

GPU 时间指标。我们将*GPU* 时间定义为 NVIDIA Nsight Systems 测得的 CUDA 内核执行时间与设备内存传输时间之和，不包括 CUDA API 开销。该指标反映设备上的计算工作和数据移动。其含义如下：(i) 当卸载量较大时，GPU 时间代表设备端性能；(ii) 当 API/主机开销占主导时，GPU 时间可能与墙钟时间不同；(iii) 如 §4.2所述，CPU 回退情况下的 GPU 时间会低得具有误导性；(iv) 如果完全没有内核启动，性能分析器会自动检测到这一点。

模型选择与升级。 PARACODEX 和基线中的所有内核均以 gpt-5.1-codex-mini 作为主要模型。对于使用基础模型时未能成功的 NAS FT 内核，我们将 PARACODEX 和基线所用的模型都升级为 gpt-5.1-codex-max。在所有实验中，智能体均未连接互联网，也未在任何阶段使用网络搜索。

基准测试与基线。我们使用 ParEval（Nichols 等人，2024a）、HeCBench（Jin 和 Vetter，2023年）、Rodinia（Che 等人，2009年）和 NAS（Bailey 等人，1991年；Fridman 等人，2025年）。

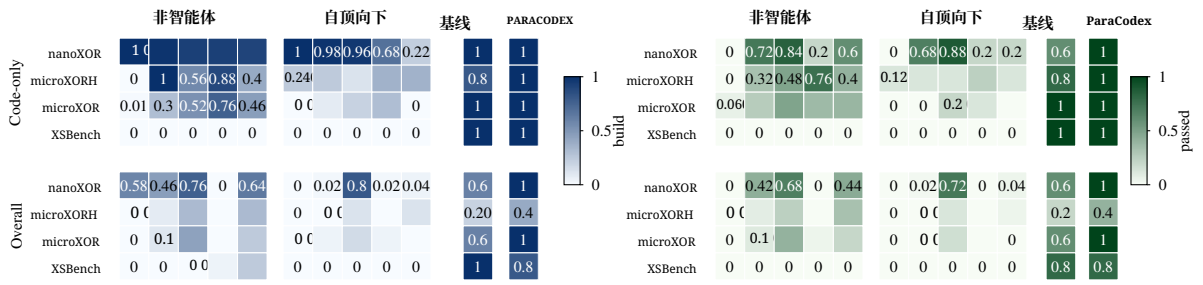


图 2：**CUDA→ParEval 上 OpenMP 在不同评分机制下的翻译质量**。图中展示了基线 Codex 模型和 PARACODEX 在 ParEval 协议下的编译（蓝色）和验证（绿色）成功率。内核按以下两种设置评分：(i) 固定构建环境（仅代码）；(ii) 系统还必须构建构建流程（总体）。热图布局与 ParEval 的原始呈现方式一致。两种模型表现都很出色；PARACODEX 的表现与基线持平或略胜一筹，说明代理工作流可泛化至跨 API 翻译。

在 HeCBench 中，我们的评估纳入了此前研究中已进行基准测试的 23 个内核。其中包括 LASSI 分析的 10 个内核（Dearing 等人，2025年），以及 UniPar 使用的 ParaTrans 数据集中的 13 个内核（Bitan 等人，2025年）。参考实现：对于所有套件，参考代码均为各基准测试所提供的 OpenMP GPU 卸载实现；我们将 ParaCodex生成的 OpenMP 与这些 OpenMP 基线进行比较。*PARAcODEX* 的基线：我们将 ParaCodex 与零样本 Codex 基线进行比较。该基线接收输入代码和一条提示，详情见 §3.1。两个系统使用相同的约束：相同的构建测试框架、正确性检查、性能分析工具以及 GPU/编译器环境。关键区别在于工作流程。对于每项基准测试，我们报告相对于参考实现的加速比 $T_{\text{ref}}/T_{\text{pc}}$，数值大于 1 表示执行时间更短。

4.2 成功率与绕过检测

表 1 全面统计了内核数量、排除项和成功率。我们评估了 36 项串行→OpenMP 翻译任务（HeCBench：23 项，Rodinia：7 项，NAS：6 项）。我们从最终性能集合中排除 5 个内核：NAS 中的两个（*BT*、*LU*）因多文件复杂性而排除；Rodinia 中的一个（*srad*）因参考实现仅支持 CPU 而排除；HeCBench 中的两个（*pathfinder*、*particlefilter*）因绕过 GPU 卸载而排除（见附录 D.2）。因此剩下 31 个有效的 GPU 卸载内核。

在 ParaCodex的 23 个 HeCBench 内核中，有 2 个观察到绕过现象：这些实现能够编译并通过正确性检查，但主要计算是在主机 CPU 上执行的。我们将绕过 GPU 的内核排除在 GPU 时间统计之外，并在附录D.2中提供详细分析。

令牌预算与可复现性。Para-Codex 平均每个内核使用 837,701 个令牌（基线的 1.11 倍）。这一成本体现了我们有意将可复现性置于效率之上。与其进行蛮力采样，我们更倾向于以非交互式自动化模式运行智能体，并将完整的构建日志和性能分析器跟踪反馈给它，以确定性地验证每个步骤。这确保整个工程流程可审计、可重放，且无需人工干预。为进一步提高可复现性，几乎所有实验均使用 `gpt-codex-5.1-mini` 并将推理力度设为最低，确保独立研究人员能够在相同配置下复现输出。值得注意的是，在相同测试框架下，零样本基线消耗的令牌数相近（755,417），这证实成本源于会累积上下文的稳健工具验证环境，而非多阶段逻辑本身（附录D.1）。

4.3 PARACODE 性能分析

基准测试套件评估。我们在四个不同的基准测试套件中评估 ParaCodex ：ParEval（CUDA→OpenMP 翻译保真度（附录 C 详细介绍了从串行→OpenMP 到 CUDA→的迁移）、图 2）、HeCBench（23 个不同的微内核，图 3）、

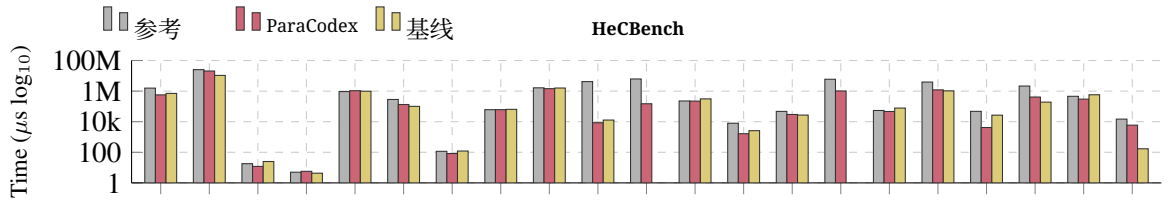


图 3：**PARACODEX 与基线 Codex 在 HeCBench 上的 GPU 执行时间性能**。 相对于 HeCBench 参考项的 GPU 时间（对数刻度；越低越好）。每个内核均报告基线 Codex 和 PARACODEX 的性能；缺失的基线柱表示未能生成正确或可编译的实现。总体而言，两种系统的性能通常都优于参考项，而 PARACODEX 在整个测试套件中取得了更高的几何平均增益和中位数增益。

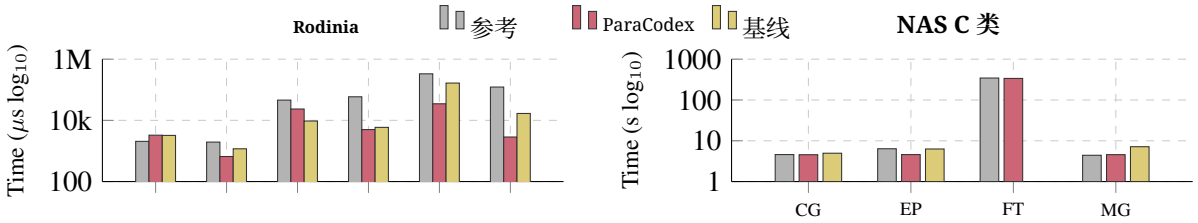


图4：**PARACODEX 与基线 Codex 在 Rodinia 和 NAS 参考实现上的性能**。Rodinia 上的 GPU 时间（毫秒，对数刻度）（左）以及 NAS 上的 GPU 时间（秒，对数刻度）（右）参考代码。Rodinia 包含实用的 OpenMP 程序，但这些程序并非总是经过充分优化；而 NAS 包含高度调优的科学计算内核，可作为更有力的参考点。在这两个套件中，基线 Codex 模型已取得显著成果。PARACODEX 进一步提升了各项基准测试的几何平均性能。NAS 结果表明，其性能已接近专家实现。

Rodinia（应用级基准测试，图4- 左）和 NAS（专家优化代码，图4- 右）。PARACODEX 展现出很强的翻译可靠性：ParEval 的纯代码验证表现完美（4/4 个内核达到 100%），整体区间性能也很强（2/4 达到 1.0，1/4 达到 0.8）；HeCBench 的编译成功率达到 100%（排除 2 个旁路案例后，21/21 个内核有效）。性能提升显著：HeCBench 的 3× GPU 时间几何平均加速比（中位数 1。59×，21 个中有 18 个性能提升），而 Rodinia 的几何平均加速比为 5。1× （中位数 6。24×）。与相关文献的进一步比较见附录 F.1、附录 F.2。

在 NAS 上，PARACODEX 的表现与专家优化的参考实现相当（1.01×中位数，1.08× 几何平均加速比）。基线的稳健性较低，未能编译 2 个 HeCBench 基准测试和 1 个 NAS 内核，仅实现了 2.4× （HeCBench）和 3×（Rodinia）的几何平均加速比。各测试套件的详细结果见附录。E；此外，附录还提供了 NAS MG 的代码对比，展示了通过基于性能分析的内核融合，性能较基线提升了 1.57× 。G。

4.4 对匿名化的稳健性

为评估 PARACODEX 是否依赖于记忆标识符模式，我们对 Rodinia 和 NAS 的匿名化变体进行评估：所有变量名和函数名均替换为合成词元，并随机打乱函数顺序。图5显示，性能基本没有变化。在匿名化 Rodinia 上，PARACODEX 在一个变体上的表现优于基线，在另一个变体上则取得了相当的结果。在 NAS 上，我们评估了分阶段工作流，结果也表明该工作流依然有效：初始翻译建立了强劲的基线，而定向优化则进一步提升了性能（例如，CG 提升至 1.41×）。这些结果表明，性能提升源于结构推理和性能分析器反馈，而非词汇记忆。

4.5 行为分析：智能体控制下的结构化推理

我们进一步考察 PARACODEX 如何影响模型在并行化过程中的推理。图6比较了 NAS EP 内核在基线和 PARACODEX 下的时间线。使用 PARACODEX 时，推理过程更常将任务分解为并行化规划和面向性能的优化

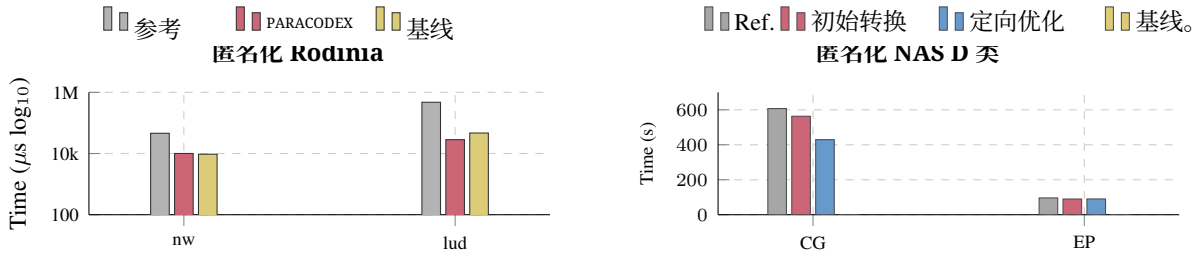


图5：**性能分解与泛化能力。**（左）在代表性内核上比较参考实现、PARACODEX 和基线的匿名化 Rodinia 图形处理器执行时间（对数刻度）。（右）匿名化 NAS D 类的性能，展示 PARACODEX 分阶段细化（初始翻译与定向优化）相较于参考实现的影响。

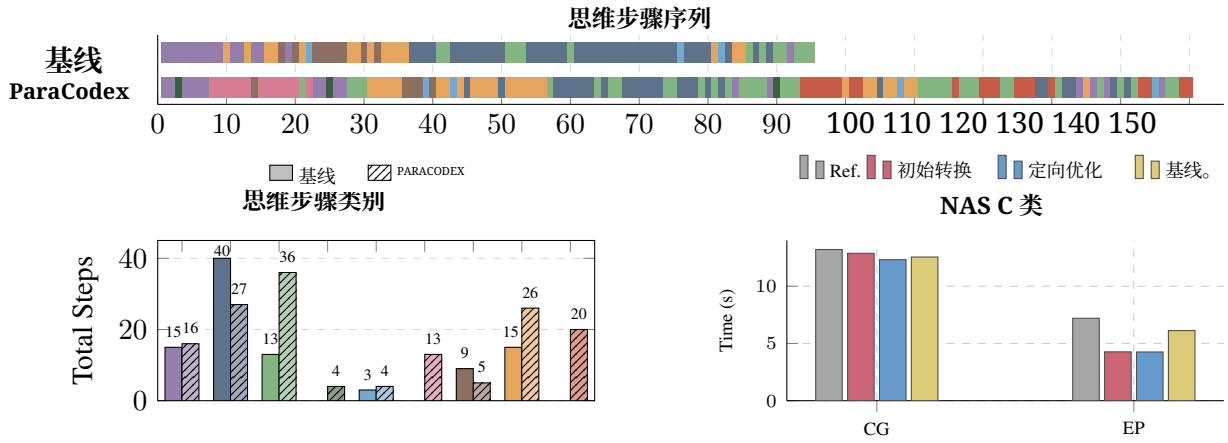


图6：**EP 内核（NAS）的智能体推理轨迹，按九种步骤类型分类，并对比基线 Codex 与 ParaCodex。**我们同时展示每一步的时间线（上方）以及各类别的步骤总数（左下方），突出说明指令工作流如何将注意力转向与任务相关的分析，而非仅影响表层输出。右下方展示了 PARACODEX 中分阶段细化的效果，对比 EP 内核与 CG 内核的分析阶段。

（例如并行区域选择、数据移动和同步），而基线则倾向于直接生成代码。右下方面板显示了类似的行为：对于 EP，性能提升大多来自初始翻译阶段中正确的卸载和数据策略；而 CG 则在两个阶段中均呈现更渐进的提升。

5 讨论

我们重新审视研究问题：

研究问题 1（可行性）：当代智能体在将串行代码转换为 OpenMP GPU 卸载方面展现出很强的能力。基线能够生成正确且性能尚可的实现。然而，在更复杂的基准测试中，其稳健性会下降：基线无法处理 NAS FT，整体编译成功率较低，而且在所有测试套件中生成的代码都不如 PARACODEX 优化。虽然对于较简单的内核，一次生成通常就足够了，但对于涉及复杂数据移动、多阶段算法和严格同步约束的科学代码库，采用结构化的迭代优化会带来显著益处。

研究问题 2（专门化）：专家引导的工作流能提升效果。PARACODEX 通过将任务分解为分析、规划和细化，持续提升编译率和加速比。推理轨迹表明，这种结构促使模型在编写代码前明确表达计划，使其注意力从表层语法转向性能诊断。因此，它能够发现基线遗漏的逻辑错误并优化数据传输。

研究问题 3（性能）：精心打造的智能体能够媲美专家。PARACODEX 在 HeCBench 和 Rodinia 上取得了显著的几何平均加速比，表现优于大多数参考实现。在高度优化的 NAS 基准套件上，它达到了专家水平，表明性能分析引导的迭代优化能够挖掘出大量的性能提升空间。在 2 个 HeCBench 内核上观察到的 GPU 卸载绕过行为，代表了一种失效模式

即由性能驱动的优化违背了并行化意图。

RQ4（泛化能力）：该方法具有良好的泛化能力。在 ParEval（CUDA→OpenMP）上，PARACODEX 只需对提示词做最少调整，便能保持较高的有效性。这表明，核心智能体模式——分析、门控翻译和反馈驱动的修复——不受源语言限制，可应用于更广泛的并行翻译任务，而不仅限于主要的串行→OpenMP场景。

6 局限性

评估仅在一块消费级图形处理器（RTX 4060）上进行，且未锁定时钟频率，因此会产生热波动，可能影响测量精度。正确性验证依赖于正确性门控代理，该代理会使用门控宏（在程序关键位置计算校验和与范数）对代码进行插桩，但仍可能遗漏细微的数值稳定性问题，或未在输出差异中显现的非确定性竞争条件。图形处理器卸载绕过案例（36 个内核中有 2 个）代表了一种失效模式：智能体生成的 CPU回退代码虽然满足表面要求，却违背了并行化意图；未来工作应将设备端执行约束设为硬性要求。对于加速比的统计显著性，尤其是提升幅度较小的情况（例如 NAS 1.08×），未来研究应通过更大的样本量加以验证。

致谢

本研究由 Pazy 基金会资助。计算工作由 Code Metal 提供支持。

参考文献

Byung Hoon Ahn、Prannoy Pilligundla 和 Hadi Esmaeilzadeh。2019年。用于优化深度神经网络编译的强化学习与自适应采样。预印本，arXiv:1905.12799。

David H. Bailey、Eric Barszcz、John T. Barton、David S. Browning、Russell L. Carter、Leonardo Dagum、Rod A. Fatoohi、Paul O. Frederickson、Thomas A. Lasinski、Robert S. Schreiber、Horst D. Simon、Venkatakrishnan Venkatakrishnan 和 Sisira K. Weeratunga。1991年。NAS并行基准测试。《国际超级计算应用期刊》，5(3):63–73。

Tomer Bitan、Tal Kadosh、Erel Kaplan、Shira Meiri、Le Chen、Peter Morales、Niranjan Hasabnis 和 Gal Oren。2025年。UniPar：一种基于大语言模型的统一框架，用于高性能计算中的并行代码和加速代码翻译。发表于*2025*年*IEEE*高性能与极限计算会议（*HPEC*），第 1–9 页。IEEE。Gregory Bolet、Giorgis Georgakoudis、Harshitha Menon、Konstantinos Parasyris、Niranjan Hasabnis、Hayden Estes、Kirk Cameron 和 Gal Oren。2025年。大语言模型能否预测并行代码的性能？发表于第 *34* 届国际高性能并行与分布式计算研讨会论文集，第 1–6 页。Uday Bondhugula、Albert Hartono、J. Ramanujam 和 P. Sadayappan。2008年。实用的自动多面体并行化工具和局部性优化器。*ACM SIGPLAN* 通讯，43(6):101–113。

Shuai Che、Michael Boyer、Jiayuan Meng、David Tarjan、Jeremy W. Sheaffer、Sang-Ha Lee 和 Kevin Skadron。2009年。Rodinia：异构计算基准测试套件。载于 *2009*年*IEEE*国际工作负载特征分析研讨会（*IISWC*），第 44–54 页。

Le Chen、Nesreen K. Ahmed、Akash Dutta、Arijit Bhattacharjee、Sixing Yu、Quazi IshtiaqueMahmud、Waqwoya Abebe、Hung Phan、Aishwarya Sarkar、Branden Butler、Niranjan Hasabnis、Gal Oren、Vy A. Vo、Juan Pablo Muñoz、Theodore L. Willke、Tim Mattson 和 Ali Jannesari。2024a。高性能计算研究与大语言模型的现状及挑战。*CoRR*，abs/2402.02018。

Le Chen、Arijit Bhattacharjee、Nesreen Ahmed、Niran- jan Hasabnis、Gal Oren、Vy Vo 和 Ali Jannesari。2024b。Ompgpt：一种用于 OpenMP 的生成式预训练变换器模型。载于欧洲并行处理会议，第 121–134 页。施普林格自然瑞士，楚格。

Le Chen、Quazi Ishtiaque Mahmud、Hung Phan、Nes- reen K. Ahmed 和 Ali Jannesari。2023年。通过增强异构 AST 表征学习使用 OpenMP 进行并行化。载于第六届机器学习与系统会议论文集（*MLSys 2023*），美国佛罗里达州迈阿密，*2023*年*6*月*4*日至*8*日。ml-sys.org。

Le Chen、Nuo Xu、Winson Chen、Bin Lei、Pei-Hung Lin、DunzhiZhou、Rajeev Thakur、CaiwenDing、Ali Jannesari 和 Chunhua Liao。2025年。超越代码对：用于大语言模型代码翻译的基于对话的数据生成。*arXiv* 预印本 *arXiv:2512.03086*。

Tianqi Chen、Lianmin Zheng、Eddie Yan、Ziheng Jiang、Thierry Moreau、Luis Ceze、Carlos Guestrin 和 ArvindKrishnamurthy。2019年。学习优化张量程序。预印本，arXiv:1805.08166。

Bowen Cui、Tejas Ramesh、Oscar Hernandez 和 Keren Zhou。2025年。大语言模型理解性能优化吗？预印本，arXiv:2503.13772。

Chris Cummins、Volker Seeker、Dejan Grubisic、Bap- tisteRoziere、Jonas Gehring、Gabriel Synnaeve 和 Hugh Leather。2024年。 Meta大语言模型

编译器：编译器优化的基础模型。 预印本，arXiv:2407.02524。

Tom Deakin 和 Timothy G Mattson。2023年。使用 *OpenMP* 为图形处理器编程：图形处理器的性能可移植性。麻省理工学院出版社。

Matthew T. Dearing、Yiheng Tao、Xingfu Wu、Zhiling Lan 和 Valerie Taylor。2025年。Lassi：一种基于大语言模型的自动化自纠错流水线，用于翻译并行科学代码。预印本，arXiv:2407.01638。

Yaoyao Ding、Cody Hao Yu、Bojian Zheng、Yizhi Liu、Yida Wang 和 Gennady Pekhimenko。2023年。Hidet：面向深度学习张量程序的任务映射编程范式。预印本，arXiv:2210.09603。

Johannes Doerfert、Kevin Streit、Sebastian Hack 和 Zino Benaissa。2015年。存在归约时 Polly 的多面体调度。预印本，arXiv:1505.07716。

Yehonatan Fridman、Yosef Goren 和 Gal Oren。2025年。从 OpenACC 到 OpenMP5 GPU 卸载：在NAS并行基准测试上的性能评估。载于*2025*年第*4*届极端异构解决方案国际研讨会论文集，第10–18页。

William F. Godoy、Pedro Valero-Lara、Keita Teranishi、Prasanna Balaprakash 和 Jeffrey S. Vetter。2023年。面向高性能计算并行编程模型内核生成的 OpenAI Codex 评估。预印本，arXiv：2306.15121。

Justin Gottschlich、Armando Solar-Lezama、Nesime Tatbul、Michael Carbin、Martin Rinard、Regina Barzi- lay、Saman Amarasinghe、Joshua B Tenenbaum 和 Tim Mattson。2018年。机器编程的三大支柱。载于第*2*届 *ACM SIG- PLAN* 机器学习与编程语言国际研讨会论文集，第69–80页。

Re'em Harel、Tal Kadosh、Niranjan Hasabnis、Timo- thy G. Mattson、Yuval Pinter 和 Gal Oren。2025年。Pragformer：利用 Transformer 对并行源代码进行数据驱动分类。 国际并行编程期刊，53(1)：2。

Re'em Harel、Idan Mosseri、Harel Levin、Lee-or Alon、Matan Rusanovsky 和 Gal Oren。2020年。 面向科学领域共享内存多核与加速多处理的源到源并行化编译器：分析、陷阱、改进与潜力。国际并行编程期刊，48(1)：1–31。

Re'em Harel、Yuval Pinter 和 Gal Oren。2023年。学习在共享内存环境中使用 Transformer 进行并行化。载于 第 *28* 届 *ACM SIGPLAN* 并行编程原理与实践年度研讨会论文集，第 450–452 页。

Kosei Horikawa、Hao Li、YutaroKashiwa、Bram Adams、Hajimu Iida 和 Ahmed E. Hassan。2025年。智能体重构：人工智能编程智能体的实证研究。*CoRR*，abs/2511.04824。

Carlos E. Jimenez、John Yang、Alexander Wettig、Shunyu Yao、Kexin Pei、Ofir Press 和 Karthik Narasimhan。2024年。SWE-bench：语言模型能否解决现实世界的 GitHub 问题？预印本，arXiv:2310.06770。Zheming Jin 和 Jeffrey S. Vetter。2023年。用于提升 SYCL 编程模型性能可移植性的基准测试套件。载于*2023*年 *IEEE* 国际系统与软件性能分析研讨会（*ISPASS*）论文集，第 325–327 页。

Tal Kadosh、Niranjan Hasabnis、Timothy Mattson、Yuval Pinter 和 Gal Oren。2023年a。量化 OpenMP：使用与采用情况的统计洞见。载于*2023*年 *IEEE* 高性能极限计算会议（*HPEC*），第 1–7 页。IEEE。

Tal Kadosh、Niranjan Hasabnis、Prema Soundararajan、Vy A. Vo、Mihai Capota、Nesreen K. Ahmed、Yuval Pinter 和 Gal Oren。2024a。Ompar：借助人工智能驱动的源到源编译实现自动并行化。*CoRR*，abs/2409.14771。

Tal Kadosh、Niranjan Hasabnis、Vy A. Vo、Nadav Schneider、Neva Krien、Mihai Capotaˇ、Abdul Wasay、Guy Tamir、Ted Willke、Nesreen Ahmed 等 1 人。2024b。MonoCoder：面向高性能计算代码与任务的领域专用代码语言模型。载于*2024*年 *IEEE* 高性能极限计算会议（*HPEC*），第 1–7 页。IEEE。

Tal Kadosh、Niranjan Hasabnis、Vy A Vo、Nadav Schnei- der、Neva Krien、Abdul Wasay、Nesreen Ahmed、Ted Willke、Guy Tamir、Yuval Pinter 等人。2023年b。Scope：你所需要的一切：面向高性能计算代码的 LLM 转换。*arXiv* 预印本 *arXiv:2308.09440*。

Tal Kadosh、Nadav Schneider、Niranjan Hasabnis、Tim- othy Mattson、Yuval Pinter 和 Gal Oren。2023年c。通过基于图的方法与 Transformer 为 OpenMP 并行化提供建议。收录于 *OpenMP*：高级任务、设备与编译器编程——第 *19* 届国际 *OpenMP* 研讨会，*IWOMP 2023*，英国布里斯托尔，*2023*年*9*月*13*日至*15*日，论文集，第 14114 卷《计算机科学讲义》，第 3–17 页。Springer Nature Switzerland Cham，施普林格。

Hao Li、Haoxiang Zhang 和 Ahmed E. Hassan。2025年。软件工程（SE）3.0 中 AI 队友的崛起：自主编码代理如何重塑软件工程。*CoRR*，abs/2507.15003。

AmanMadaan、Niket Tandon、PrakharGupta、Skyler Hallinan、Luyu Gao、Sarah Wiegreffe、Uri Alon、Nouha Dziri、Shrimai Prabhumoye、Yiming Yang、Shashank Gupta、Bodhisattwa Prasad Majumder、Katherine Hermann、Sean Welleck、Amir Yazdan- bakhsh 和 Peter Clark。2023年。自我精炼：通过自我反馈进行迭代式改进。预印本，arXiv:2303.17651。

Aleksandr Maramzin、Christos Vasiladiotis、Roberto Castaneda Lozano、Murray Cole 和

Bjorn Franke。2019年。“看起来你正在编写并行循环”：辅助开发者做出并行化决策。载于 *AISEPS* 研讨会论文集。

Massinissa Merouani、Khaled AfifBoudaoud、Iheb Nas- sim Aouadj、Nassim Tchoulak、Islem Kara Bernou、Hamza Benyamina、Fatima Benbouzid-Si Tayeb、Karima Benatchba、Hugh Leather 和 Riyadh Baghdadi。2025年。Looper：面向多面体编译器的学习型自动代码优化器。预印本，arXiv:2403.11522。

Sean Mondesire、Emmanuel Nsiye、Bulent Soykan 和 Glenn Martin。2025年。通过基于大语言模型的多智能体系统实现高性能计算软件的自动化编译、部署和错误解决。载于 *2025*年高级研究计算实践与经验：协作的力量，第 1–8 页。

Idan Mosseri、Lee or Alon、Re’em Harel 和 Gal Oren。2020年。Compar：面向自动 OpenMP 源到源并行化的优化多编译器。载于 国际 *OpenMP* 研讨会，第 247–262 页。施普林格国际出版公司。

DanielNichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram 和 Abhinav Bhatele。2024a。大语言模型能编写并行代码吗?

Daniel Nichols、Aniruddha Marathe、Harshitha Menon、Todd Gamblin 和 Abhinav Bhatele。2024b。HPC- coder：使用大语言模型对并行程序建模。预印本，arXiv:2306.17281。Yun Peng、Akhilesh Deepak Gotmare、Michael Lyu、Caiming Xiong、Silvio Savarese 和 Doyen Sahoo。2024。Perfcodegen：利用执行反馈提升大语言模型生成代码的性能。预印本，arXiv:2412.03578。Nadav Schneider、Niranjan Hasabnis、Vy A Vo、Tal Kadosh、Neva Krien、Mihai Capota、Guy Tamir、Theodore L Willke、Nesreen Ahmed、Yuval Pinter 等。2024。Mpirigen：通过领域专用语言模型生成 MPI 代码。载于*2024* 年人工智能系统研讨会论文集，第 1–6 页。Nadav Schneider、Tal Kadosh、Niranjan Hasabnis、Tim- othy Mattson、Yuval Pinter 和 Gal Oren。2023。MPI-rical：利用 Transformer 提供数据驱动的 MPI 分布式并行辅助。载于高性能计算、网络、存储与分析国际会议 *SC’23* 研讨会论文集，第 2–10 页。Noah Shinn、Federico Cassano、Edward Berman、Ash- win Gopinath、Karthik Narasimhan 和 Shunyu Yao。2023。Reflexion：通过语言强化学习实现语言智能体。预印本，arXiv:2303.11366。Shivam Singhal、Eran Malach、Tomaso Poggio 和 Tomer Galanti。2025。LLM-ERM：通过大语言模型引导的搜索实现样本高效程序学习。预印本，arXiv:2510.14331。

TOP500。2025年。TOP500榜单——2025年11月。https://top500.org/lists/top500/list/2025/11/。第66期TOP500榜单于2025年11月18日发布（密苏里州圣路易斯）。访问日期：2026-01- 04。

Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F. Godoy, Keita Teranishi, Prasanna Bal- aprakash 和 Jeffrey S. Vetter。2023年。用于生成高性能计算内核的 Llama-2 与 GPT-3 大语言模型比较。预印本，arXiv:2309.07103。

Pedro Valero Lara, Aaron Young, Jeffrey S Vetter, Zheming Jin, Swaroop Pophale, Mohammad Alaul Haque Monil, Keita Teranishi 和 William FGodoy。2025年。ChatHPC：为构建高效且可信赖的人工智能辅助的高性能计算生态系统奠定基础。载于国际高性能计算、网络、存储与分析会议论文集，第 458–474 页。

Bhanu PrakashVangala、Ali Adibifar、Tanu Malik 和 Ashish Gehani。2025年。人工智能生成的代码尚不可复现：基于大语言模型的编程智能体中依赖缺口的实证研究。*arXiv* 预印本 *arXiv:2512.22387*。

Xingfu Wu、Praveen Paramasivam 和 Valerie Tay- lor。2023年。基于贝叶斯优化对 Apache TVM 科学应用进行自动调优。预印本，arXiv:2309.07235。

John Yang、Carlos E Jimenez、Alexander Wettig、Kilian Lieret、Shunyu Yao、Karthik Narasimhan 和 Ofir Press。2024年。SWE-agent：智能体—计算机接口实现自动化软件工程。神经信息处理系统进展，37:50528–50652。Shunyu Yao、Dian Yu、Jeffrey Zhao、Izhak Shafran、Thomas L. Griffiths、Yuan Cao 和 Karthik Narasimhan。2023年。思维树：利用大语言模型进行审慎的问题求解。预印本，arXiv:2305.10601。Eric Zelikman、Eliana Lorch、Lester Mackey 和 Adam Tauman Kalai。2023年。自学优化器（STOP）：递归式自我改进代码生成。微软研究院。Qinkai Zheng、Xiao Xia、Xu Zou、Yuxiao Dong、Shan Wang、Yufei Xue、Zihan Wang、Lei Shen、Andi Wang、Yang Li、Teng Su、Zhilin Yang 和 Jie Tang。2024年。CodeGeeX：用于代码生成并在 HumanEval-X 上进行多语言基准测试的预训练模型。预印本，arXiv:2303.17568。

面向 NAS 和 Rodinia 的流水线阶段重构

A.1 理由

从 HeCBench 微内核转向规模更大的 NAS 和 Rodinia 基准测试时，我们重新设计了优化工作流，以更好地反映完整应用的性能驱动因素。

微内核通常只暴露少量局部热点，因此采用一系列范围较窄的优化阶段会很有效。相比之下，NAS 和 Rodinia 内核往往需要从整体上考虑循环结构、数据生命周期、启动开销以及大规模场景下的正确性约束。早期原型将优化拆分为多个范围较窄的阶段（例如，分别进行并发和内存/局部性优化）。当前流水线将这些阶段整合为两个适用于各个测试套件的优化步骤：(i) GPU 卸载，并配有明确的 data_plan.md 和基线采集；(ii) 基于性能分析的性能调优，并配有明确的 optimization_plan.md。在这两种情况下，每个阶段都会补充以下信息：(i) 上一阶段的性能分析器输出；(ii) 对此前已执行操作的简短总结；(iii) 系统详情（GPU/CPU/内存），从而能够在固定的硬件/软件栈下，基于性能分析进行迭代优化。

A.2 HeCBench 工作流

对于 HeCBench，PARACODEX 使用了在初步实验阶段开发的四阶段流水线。该工作流包括：(i) 分析；(ii) 初步卸载至图形处理器并进行基本正确性验证；(iii) 并发调优（例如，collapse 指令、线程映射）；以及 (iv) 内存/局部性优化（例如，数据移动、减少传输）。与后来为 NAS 和 Rodinia 开发的整合式三阶段工作流不同，这种方法在每个阶段使用的提示更短、内容也不够全面，并将优化分解为范围更窄的多个步骤。虽然由于增加了阶段和上下文累积，这种做法导致令牌成本更高，但它体现了我们早期的开发方法，因此为保持与已完成实验的一致性，HeCBench 仍沿用了这种方法。HeCBench 基准测试具有微内核的特点，通常由单个热点循环主导，因此，尽管开销有所增加，分阶段分解仍然有效。

A.3 NAS/Rodinia 工作流：整合式三阶段流水线

对于 NAS 和 Rodinia，PARACODEX 采用经过改进的三阶段流水线，将优化整合为范围更广、更全面的阶段。这一重新设计源于完整应用基准测试的特点，这类测试要求对循环结构、数据生命周期和启动开销进行整体推理，而非采用范围狭窄的独立流程。这三个阶段分别为：(i) 基于循环分类体系进行分析；(ii) GPU 卸载，并生成明确的数据规划工件（data_plan.md）；(iii) 基于性能分析的调优，并生成优化规划工件（optimization_plan.md）。与 HeCBench 的四阶段流水线不同，该工作流采用更长、更详细的提示，将并发和内存优化整合到单一的基于性能分析的改进阶段中，从而降低令牌成本，同时加深推理。每个提示都以性能分析器输出、系统元数据和先前操作的摘要作为参数，从而能够在固定的硬件/软件栈下进行基于测量的迭代。

阶段 1（分析阶段）：循环分类与数据特征分析。分析阶段会枚举并确定基准测试源文件中的循环优先级，重点关注在主要计时区域内执行的代码。每个循环都会根据循环分类体系进行分类，该体系编码了并行化约束（例如稠密、稀疏/压缩稀疏行、多阶段快速傅里叶变换/蝶形运算、多重网格、直方图/间接写入、递推、归约和模板模式）。该阶段还会记录循环嵌套、依赖关系结构（归约、阶段依赖关系、循环携带递推）以及数据属性（数组作用、访问模式、全局变量、临时存储使用情况），并标记原子操作、变量边界或迭代次数不足等风险因素。此阶段会生成结构化的 analysis.md 工件，并在不进行语义修改的情况下，将源文件复制到内核工作目录。

阶段 2（图形处理器卸载 + 数据策略）：通过显式数据计划确保正确性。不同于以正确性优先的最简卸载步骤，阶段 1 将正确性与显式的数据生命周期推理相结合。该工作流 (i) 记录基准输出，以供后续验证；(ii) 通过基于规则的决策流程选择数据管理策略（例如目标数据区域、异步重叠，或通过 omp_target_alloc 和 is_device_ptr 进行全局设备状态分配）；(iii) 要求流水线在修改代码之前生成 data_plan.md 文件。数据计划会列出计时区域中使用的数组，识别必须在设备上执行的函数，说明主机到设备及设备到主机传输的时机和预期数据量，并纳入针对特定策略的正确性

检查，以防止常见的映射错误。随后，实施过程将遵循此计划卸载所需的循环/函数，同时保留基准测试语义并通过黄金串行正确性检查。

阶段 3（性能调优）：性能分析引导的计划与提前退出。阶段 2 将并发调优、内存/局部性调优和启动开销降低整合为一个由性能分析驱动的优化阶段。首先，它会根据记录的基准输出重新验证正确性。随后读取性能分析摘要（内核时间、API 时间和传输时间），并记录关键指标（主要内核、启动次数和传输量）。在应用更改之前，流水线会生成一份 optimization_plan.md 产物，列出瓶颈优先级并提出具体措施（例如提升数据区域、消除冗余传输、内联辅助函数以降低启动开销、在安全的情况下进行循环融合、增加 collapse 深度、仅在有益时为稀疏内层循环引入归约、为多阶段算法强制采用串行阶段循环，以及使用 const/restrict 和缓存局部变量等微优化）。该阶段支持提前终止：如果当前运行时间与根据性能分析推断出的估计最优值相差不大，流水线就会记录指标，并将更改限制在低风险微优化范围内。

A.4 差异总结

PARACODEX 针对基准测试特征和开发进度采用了两种不同的工作流。四阶段 *HeCBench* 流水线体现了我们最初的实验设计：采用范围较窄、长度较短的提示，将优化分解为独立的并发和内存处理阶段，从而产生更高的令牌成本。三阶段 *NAS/Rodinia* 流水线则代表了一种改进后的整合方案，将优化纳入全面的阶段中，并明确采用制品驱动规划（data_plan.md、optimization_plan.md）。这一改进降低了令牌开销，同时显著提升了推理深度以及全应用基准测试的优化有效性。两种工作流都遵循正确性门控和分析器驱动的细化这一核心理念；更为关键的是，正确性门控机制本身在两种方案中始终保持不变——所有实现均通过黄金串行输出进行验证，并使用基于 Makefile 的相同测试框架和监督器驱动的修复循环。在成本效益和推理质量方面，三阶段流水线严格优于四阶段方案；保留 HeCBench 的四阶段方案，仅仅是为了与已完成的实验保持一致。在不同测试套件之间，瓶颈特征各不相同：HeCBench 微内核往往受内核限制，热点较为局部；而 NAS/Rodinia 基准测试则经常暴露出传输开销、启动成本以及复杂的数据生命周期模式，这些问题能够从改进工作流所支持的整体数据管理规划中受益。

B 分析、数据传输与优化机制

本附录详细说明 PARACODEX 如何在串行代码到 OpenMP 目标卸载的翻译过程中实现 (i) 循环级分析、(ii) 数据移动规划与执行，以及 (iii) 基于性能分析的优化。该流水线通过三个提示模板实现：翻译前使用的分析提示、用于根据显式数据计划实现正确 GPU 版本的卸载 + 数据策略提示，以及利用性能分析反馈优化实现的性能调优 提示。整个过程中，提示会使用运行时模板变量进行参数化（例如 {kernel_dir}、{file_listing} 以及构建/运行命令），并由流水线脚本填充。

B.1 循环分析与卸载目标识别

分析阶段会生成结构化的 analysis.md 文件，同时保持原始源文件不变。其目标是：(1) 识别主要计时区域中的循环；(2) 描述依赖结构，以确定卸载可行性；(3) 揭示在翻译过程中必须显式处理的风险因素（归约、原子操作、阶段依赖）。

循环发现与优先级排序。提示通过轻量级源代码检查（例如搜索 for、while 和主计算循环模式）枚举循环结构。根据循环在调用路径中的位置确定优先级：每次主计算循环迭代都会执行的循环被视为关键/重要，而仅执行一次的初始化循环则被视为次要/避免。

根据估计工作量进行优先级分类。 对于每个循环，提示应用粗粒度工作模型（迭代次数 $\times$ 每次迭代的操作数），并分配四个优先级级别之一：关键 （占主导地位或每次迭代都会执行的 $O(N)$ 工作）、重要、次要或避免 （初始化 /IO 或规模极小的循环）。首要目标是将后续卸载工作集中在可能占据运行时间主导地位的循环上。

循环类型分类体系与依赖特征。每个循环都会从一个小型分类体系中分配一种类型，该分类体系编码了并行化约束：

•A 类（稠密）： 边界恒定，具有数据并行结构。

•B 类（稀疏/CSR）： 内层边界依赖于外层索引；通常对外层循环进行并行化，并可选择进行内层并行。

•C1 类（多阶段/迭代）： 计算依赖于阶段（例如蝶形模式）；外层循环并行，阶段循环串行。

•C2 类（多重网格/分层）： 按层遍历并包含相互依赖的阶段；外层并行，同时保留阶段顺序。

•D 类（直方图/间接写入）： 可通过原子更新（或结构化私有化 + 合并）实现并行化。

•E 类（递推）： 循环携带依赖或块级同步模式（例如 CUDA 中的 __syncthreads()）；若不进行算法重构，则无法通过卸载实现并行化。

•F 类（归约）： 标量累加；可通过归约实现并行化。

•G 类（模板）： 邻域访问；通过谨慎设置索引即可实现并行化。

一种特殊情况会被明确处理：当外层循环遍历相互独立的样本，而内层循环包含顺序执行的随机数生成器/状态更新时，分析会将外层循环标记为可并行化，并采用每线程随机数生成器复制；同时将内层随机数生成器循环标记为每个线程内顺序执行。

数据分析与危险标志。 除了循环结构外，分析还会记录正确映射所需的数据属性：数组形状（扁平或基于指针）、分配方式（静态或动态）、访问到的结构体成员，以及计时区域中使用的全局变量。分析还会标记翻译时必须处理的风险因素，包括可变边界、所需的归约、原子更新、阶段依赖、计时区域中的随机数生成器用法，以及较小迭代次数。

B.2 数据移动与传输策略 **gy**

正确且高效的目标卸载高度依赖数据生命周期决策。因此，优化的第一步要求流水线选择一种数据策略，并在实现编译指示之前生成 data_plan.md 之前。该计划是可执行规范，说明哪些数据驻留在设备上、何时进行数据传输，以及哪些函数必须在设备上执行，以避免隐式的主机与设备之间的数据抖动。

策略选择规则。该提示会根据循环分类和程序结构，从三种策略中选择一种：

1.**策略 A（作用域 目标数据 区域）：** 使用带显式 map(to |from |tofrom |alloc) 子句的 target data；适用于大多数稠密/模板/归约内核以及类似多重网格的情况。2.**策略 B（异步重叠）：** 当计算支持流水线化时，使用 nowait 和 depend，使彼此独立的数据传输与内核执行重叠。3.**策略 C（全局设备状态）：** 使用 omp_target_alloc 分配持久设备数组，并通过 is_device_ptr 传递这些数组，以消除迭代求解器和多阶段核函数中的重复映射。

这份数据_plan.md工件。数据计划列出计时区域中使用的所有数组，并将其分类为工作、临时数据、*const*或索引。它还列出在计时区域中执行的函数，并将每个函数指定为由主机执行或由设备端执行。该计划明确规定：(i) 一次性设备分配；(ii) 主机到设备的数据传输（时间和数据量）；(iii) 设备到主机的数据传输（时间和数据量）；以及 (iv) 迭代循环内是否发生任何数据传输。计划还包含针对不同策略的检查清单，旨在防止常见错误，例如：缺少辅助函数的设备端版本（导致隐式复制）、使用持久设备指针时临时缓冲区仍留在主机上，或在策略 C 的热路径中意外重新引入映射子句。

传输量合理性检查。该计划会记录整个执行过程的预期传输量，并将大幅偏差标记为红旗。具体而言，如果测得的数据传输量超出计划值超过一个较小的常数倍（例如 $> 2\times$），

流水线会将此视为数据作用域划分不当的证据（例如，缺少卸载辅助函数导致重复数据传输），并优先纠正数据生命周期，然后再进行内核级调优。

B.3 正确性门控与基线等价性

每次内核翻译都以修改前采集的参考基线为依据。优化的第一步是记录基线输出，并验证翻译后的输出与之相符（例如，通过对基准测试的验证标记进行文本差异比较）。只有通过黄金串行正确性检查的配置才会被保留用于性能报告。在基于性能分析的调优之前会强制执行此门控，以确保后续优化基于语义有效的候选方案进行。

验证失败时，系统会调用监督代理，以：(i) 使用 gate.h 宏插桩代码，追踪执行状态；(ii) 分层诊断差异（先检查主机与设备内存的一致性，再检查内核逻辑）；(iii) 在保留 GPU 卸载的同时进行最小幅度的修复。监督代理会持续迭代，直至 make check-correctness 通过。提示明确禁止仅 CPU 回退以及在此阶段进行性能改动——此阶段的唯一目标是确保数值正确性。

B.4 基于性能分析的优化与动作库

第二步优化是在固定已选数据策略的同时，通过性能分析进行迭代优化。提示明确禁止在此阶段更改数据策略，以避免将数据生命周期重构带来的性能提升与内核级优化混为一谈。

提前退出标准。如果当前运行时间与根据性能分析推断的预期最优值相差不超过一个小幅度（5%），或者拟议的更改导致性能下降超过 10%（例如，内核时间主导且传输开销较低），流水线便会记录当前指标，并将自身限制在低风险微优化范围内（例如 const、restrict 和局部变量缓存）。这样可以避免在实现已接近性能分析器所推定的上限时进行不必要的重构。

此优化_plan.md 工件。在应用更改之前，流水线会生成一份优化_plan.md 报告，其中包含：

阶段	关键输出与执行约束
分析	analysis.md：循环分类、优先级、de-依赖关系、数据冒险；源代码未经修改地复制。
卸载 + 数据计划	data_plan.md：数组清单、函数放置、显式传输、预期数据传输规模；实现必须遵循计划；基线已捕获且正确性已验证。
调优	优化_plan.md：分析器驱动的诊断、优先级排序的操作、提前退出规则；数据策略已固定；最终总结包含附应用/撤销的更改。

表 2：PARACODEX 用于构建分析、数据移动和基于性能分析的优化流程的产物与强制执行点。

运行时间、主要内核、图形处理器运行时间细分、传输占比和数据量，以及内核启动次数。该计划还会识别候选循环融合（生产者–消费者循环，或边界相同的相邻循环），并描述迭代结构（例如，包含重复稀疏矩阵-向量乘法/更新模式的迭代求解器循环）。

优先级瓶颈清单。调优提示按优先级顺序编码了瓶颈及相应解决措施的清单。表 2总结了流水线产物和强制执行点：

•**数据管理问题：** 提升数据区域，将临时数据移至设备分配，并确保所有计时区域辅助函数都在设备上运行。

•**启动开销：** 内联迭代循环中调用的辅助函数；在边界相同且安全的情况下，融合相邻循环。

•**热点内核效率低下：** 调整并行分解方式（例如折叠），为最内层循环添加单指令多数据，并缓存索引/数组值，以减少冗余加载。

•**稀疏内层循环决策：** 当平均每行非零元数较少时，保持 CSR 内层循环串行；只有当非零元数量足够多、能够摊销开销时，才引入带归约的内层循环并行。

•**阶段依赖算法（C 型）：** 强制阶段循环串行；仅并行化外层维度，以避免屏障开销和正确性故障。

•**过度并行化：** 当性能分析和问题几何特征表明外层循环并行已使图形处理器达到饱和时，应移除内层并行。

变更摘要与来源记录。最后，提示要求在每个步骤结束时提供结构化摘要，记录基线和最终指标，列出已应用的优化（包括因性能退化而撤销的变更），并说明主要见解和剩余瓶颈。这种报告规范有助于确保可复现性，并为评估中报告的性能提升提供来源记录。

B.5 示例：NASCG 共轭梯度求解器工作流

我们以 NAS 并行基准测试 C 类中的共轭梯度（共轭梯度）内核为例，展示这一三阶段工作流。该稀疏线性求解器采用迭代共轭梯度，并使用 CSR 格式执行稀疏矩阵-向量乘法（SpMV）。

（i）热点摘要（`analysis.md`）。分析阶段识别出基准测试的主循环（共 15 次迭代，每次调用 conj_grad，并进行 25 次内部 cgit 迭代），并对嵌套循环进行分类：

•E 类（顺序执行）：外层基准测试迭代（it = 1..NITER）和内层 cgit 循环（25 次迭代）存在关于 rho/beta 的循环携带依赖，因此必须串行执行。

•B 类（稀疏矩阵-向量乘法）：两个 SpMV 内核分别计算 q = A*p 和 r = A*z，通过 rowstr/colidx 进行不规则 CSR 索引；各行之间可进行数据并行，关键优先级。

•F 类（归约）：点积（norm_temp1/norm_temp2、rho、d）和最终残差范数需要进行全局归约，关键优先级。

•A 类（稠密 SAXPY）：向量更新（z/r/p 的 axpy 运算）——极易并行，且受内存带宽限制。

数据：CSR 矩阵（a[NZ],colidx[NZ],rowstr[NA+1]，总计 ∼461MB），以及五个工作向量（x、z、p、q、r，每个包含 NA+2 个双精度浮点数）。
风险因素：连续的 cgit 迭代阻碍外层并行；SpMV 中不规则的聚集访问模式限制内层并行。

（ii）数据计划（数据_计划.md）。策略 A（持久化目标数据）：在基准测试循环开始前，使用 #pragma omp target enter data map(to: a[0:NZ],colidx[0:NZ],rowstr[0:NA+1]) 和 map(alloc: x[0:NA+2], z[0:NA+2], p[0:NA+2], q[0:NA+2], r[0:NA+2])，使数据常驻设备。工作向量在设备端初始化，以避免主机与设备间传输。预期传输量：入口处 461MB H→D（CSR 数据），D→H 可忽略不计（仅标量归约结果），迭代循环15×25 期间不传输数组。正确性检查：初始卸载后通过验证，验证成功。

(iii) 优化 +门控结果（`优化_plan.md`）。性能分析显示，运行时间主要耗费在 9,883 次内核启动上（400 次 SpMV 遍历，以及单独的归约/更新内核）；共轭梯度过程占用 94% 的 GPU 时间，而启动与同步开销造成的 API 开销高达 91.7%。瓶颈：用于范数归约和残差计算的重复小型内核增加了启动开销；每项操作都会启动独立的设备内核。优化方案：(i) 将双范数归约（norm_temp1/norm_temp2）融合到单个内核中；(ii) 合并最后一次 SpMV 与残差范数循环，以复用 rowstr/colidx 的加载，并在每次共轭梯度调用中消除一个内核；(iii) 将中间标量缓存到寄存器中，避免冗余的全局内存访问。结果：内核启动次数减少了 ∼25%，运行时间缩短至 2.04 秒（估计相较基线加速 ∼20%）。正确性门控：输出通过了与串行参考实现的比对验证；保留该优化。

B.6 基线提示结构

作为比较，基线系统使用一个非结构化提示，将所有目标合并为一个请求。基线提示模板如下：

> **你的任务：**
> 1. 将代码转换为使用 OpenMP 卸载到图形处理器的版本。
> 2. 根据需要应用 GPU 卸载编译指示。
> 3. 在保持功能不变的同时优化代码性能。4. 确保代码能够编译并运行。
> 5. 将修改后的代码交付到 {kernel_dir}。
>
> 交付物：{kernel_dir} 中完整的修改后源代码

这种单次处理方法将分析、翻译和优化合并为一个未加区分的请求；相比之下，PARACODEX 采用分阶段工作流，会生成中间产物（analysis.md、data_plan.md、

optimization_plan.md），并在各阶段之间设置正确性关卡。基线智能体可以使用与 PARACODEX 相同的工具（编译器、性能分析器、测试框架），但不会得到关于何时或如何使用这些工具的明确指导。实际使用中，基线智能体可能会编译并测试代码，但不会系统地分析循环、选择数据策略，或根据性能分析反馈反复进行优化，因为这些步骤并未得到提示。

C 从串行→OpenMP 到 CUDA→ OpenMP 的翻译

我们的系统最初面向 CPU 程序，通过自动将串行代码翻译为 OpenMP 目标卸载来实现 GPU 加速。这需要分析循环嵌套、分类并行化模式，并制定明确的设备驻留与数据移动计划。

后来，我们扩展了这一框架，以支持从 CUDA 代码迁移到 OpenMP 目标卸载。尽管整体工作流程在结构上仍然相似，但设计目标和约束发生了显著变化。本附录总结了其中的主要差异。

C.1 问题设定的变化

在串行→OpenMP 场景中，输入代码受 CPU 限制，必须引入并行性。相比之下，CUDA 代码本身已经实现了 GPU 并行。因此，迁移任务变为：

1. 保留程序现有的 GPU 执行语义；2. 将 CUDA 运行时构造转换为 OpenMP 等效项；3. 避免意外地重新引入主机–设备传输或导致数据驻留丢失。

因此，分析阶段的范围扩展了：除了计算循环，还包括 CUDA 内核、启动位置和设备端执行结构。

C.2 CUDA 内核的扩展分析

原始分析聚焦于 CPU 循环及其计算成本。对于 CUDA 输入，分析阶段还会：

•检测 __global__ 内核和主机启动位置，

•重建网格和线程块结构，
•识别网格跨步循环模式，
•分析 atomicAdd、共享内存的使用情况以及 __syncthreads()，
•估算设备工作总量，即网格 × 线程块 × 迭代次数。

这使系统能够推理 CUDA 执行语义，并确保这些语义在 OpenMP 下得到保留。

C.3 数据驻留与内存模型映射

在串行→OpenMP 翻译中，只有在有益时才会逐步引入设备端驻留。然而，CUDA 程序已经假定数据驻留在图形处理器上，并且会显式控制所有分配和传输（cudaMalloc、cudaMemcpy、cudaFree）。

因此，迁移流水线会先重建 CUDA 内存模型，再将其映射到 OpenMP 构造。

一个核心设计目标是在迁移过程中避免无意间将数据移回主机。

C.4 执行模型与同步差异

CUDA 通过__shared__ 和__syncthreads()提供块级局部共享内存与同步。OpenMP 目标卸载没有完全对应的机制。因此，迁移流程执行以下操作：

•将共享内存缓冲区转换为线程私有数组或局部数组，
•将 atomicAdd 替换为 OpenMP 原子操作或归约，
•拆分或重构依赖__syncthreads()的内核。

相较于串行工作流，这代表了一项主要的概念扩展。

C.5 内核主体与索引映射

CUDA 内核会转换为设备函数，并在目标团队循环中调用。涉及 threadIdx、blockIdx 和 gridDim 的线程索引逻辑会被循环归纳变量取代。网格跨步循环会转换为常规有界循环。

所有 CUDA API 调用和 CUDA 特定语法都会被移除，以确保纯 OpenMP 执行。

C.6 正确性与基线输出

两种工作流都通过差异对比来确保正确性。对于 CUDA 迁移，系统首先记录原始 CUDA 程序输出。

卸载到 OpenMP 的版本必须与此基线一致，以防止迁移不完整或语义偏移。

C.7 性能考量

CUDA→OpenMP调优阶段会引入特定于迁移的瓶颈，包括：
•迁移后非预期的数据传输，
•移除屏障后过多的内核启动次数，

•原始 CUDA 网格维度的丢失，以及
•移除共享内存后局部性降低。

因此，优化重点是在可能的情况下恢复类似 CUDA 的执行模式（例如持久化设备分配和循环融合），同时遵循 OpenMP 执行模型。

C.8 概念转变总结

总而言之，Serial→OpenMP 流水线教会系统如何引入图形处理器执行。CUDA→OpenMP 流水线则教会系统如何在不同的运行时环境中保留 既有的图形处理器执行模型。这需要扩展分析阶段、明确 CUDA 到 OMP 语义，并避免设备驻留状态或同步行为出现退化。

D 可复现性与实验细节

D.1 技术细节与令牌数

所有内核均使用 gpt-5.1-codex-mini 作为主模型。对于 NAS FT 内核，为确保公平性，PARACODEX 和基线均升级使用 gpt-5.1-codex-max。令牌用量（NAS 基准测试，代表当前流水线）：PARACODEX 平均每个内核使用 837,701 个令牌（cg：776,036；ep：967,774；ft：1,031,208；mg：575,786），而基线平均使用 755,417 个令牌（ cg：286,118 ； ep：1,379,682 ； ft：880,104；mg：475,765）。1.11× 这一比例反映了 PARACODEX 的多阶段工作流（分析 →卸载+数据计划 → 优化），与基线的单次翻译形成对比。

Token 用量为何如此之高：PARACODEX 和基线都在非交互式自动化模式下运行 Codex CLI。该模式专为脚本编写和 CI/CD 工作流而设计，智能体可在单次会话中执行所有必要操作，无需人工交互。与交互式开发相比，这种模式会消耗明显更多的 Token，原因如下：(i) 每轮智能体操作都必须提供完整的代码库上下文、构建环境和工具输出；(ii) 智能体会保留完整的执行轨迹，以确保可复现性；(iii) 编译错误、测试输出和性能分析器报告都会原样纳入，以便进行确定性诊断。这种设计优先考虑可复现性和确定性，确保每次并行化尝试都完全可审计、可重放，而非追求 Token 使用效率。值得注意的是，基线的 Token 消耗量也相当高（每个内核 755k Token），尽管其结构更简单、只需单次处理。这证实，高 Token 用量主要是非交互式自动化模式所致，而非 PARACODEX 的分阶段工作流。完整的提示模板可在我们的代码仓库中找到。

D.2 GPU 卸载绕过：详细分析

在两个 HeCBench 绕过案例（*pathfinder*、*particlefilter*）中，PARACODEX 生成了能够通过 OpenMP 目标指令编译并通过正确性检查的实现，但实际上通过在主机 CPU 上执行主要计算来绕过图形处理器计算。零样本 Codex 基线在 *pathfinder* 上也表现出相同的行为。

绕过的定义与检测。 当检查生成代码及相应的性能分析器报告后发现，主要计算循环仍在主机上执行，而设备仅执行一个极小的恒等内核，且该内核并未承担实质性的计算工作时，我们将该运行归类为绕过。由于这种行为违背了 GPU 并行化的既定目标，纳入此类运行会歪曲成功率，并可能在无法反映真实卸载效果的情况下夸大性能比较结果。因此，我们将绕过内核排除在 GPU 时间统计之外，同时明确报告其发生情况。

绕过的发生情况。 在 PARACODEX 的 23 个 HeCBench 内核中，有 2 个观察到绕过现象。在我们的评估协议下，Rodinia、NAS 或 ParEval 中均未观察到绕过案例。

根因分析。 我们通过检查智能体对话记录和性能分析日志，分析了这两种绕过情况。在这两种情况下，智能体生成的代码均可编译并通过正确性检查，但主要计算却在主机 CPU 上执行，而图形处理器内核几乎不起作用或仅执行恒等操作。假设：智能体发现相较于计算，传输成本较高，因而暗中选择了更快的 CPU 执行方式，违背了 GPU 卸载意图。我们的

检查表明，当智能体判断传输开销占主导地位时，就会出现绕过现象；尽管生成了语法有效的卸载结构，它仍选择将计算保留在主机上。

缓解策略。展望未来，工具框架可以通过强制要求提供设备端工作的证据来避免规避，例如要求存在实质性的内核启动活动、检查设备内存分配大小（$>$ 1MB），或根据性能分析测得的运行时间，对图形处理器内核所占比例设置阈值（例如，要求设备内核耗时占比达到 $>$ 50%）。初步测试表明，设置设备内存最小分配量和内核启动次数（$>$ 10）限制后，可恢复*pathfinder* 的图形处理器卸载，但*particlefilter*则需要重新调整算法结构。

E 详细基准测试套件结果

本节全面分析 PARACODEX 在 ParEval、HeCBench 和 Rodinia 基准测试套件中的表现。

E.1 ParEval：稳健的 CUDA→OpenMP 翻译与正确性

我们在 ParEval 上评估 PARACODEX，该基准强调CUDA到 OpenMP 的迁移（附录 →介绍了 CUDA 特定的工作流程扩展）。每个内核都在固定构建环境下（仅代码）进行评估，也在智能体必须同时构建构建流程的设置下（整体）进行评估。如图 *2*所示，*PARACODEX* 在仅代码模式下实现了完美的编译和验证（4 个内核均达到 100%），而在要求更高的整体模式下也表现出色：它在所有试验中都成功编译并验证了*nanoXOR*和*microXOR*（1.0），在*XS-Bench*上达到 0.8，其余错误则集中在 microXORH（0.4）。基线的整体成功率始终较低，在microXORH上的下降幅度最大。热图还显示，非智能体方法表现明显不佳，而自顶向下智能体方法虽有所改进，但仍不稳定。总体而言，PARACODEX 的结构化工作流程——CUDA 内核分析、内存模型重建和执行语义保持——能够有效泛化到跨 API 翻译。

E.2 HeCBench：多样化工作负载下的广泛性能提升

HeCBench 涵盖了规则的模板计算和动态规划内核，也包含非规则的内存密集型和面向图的工作负载。我们评估了 23 个基准测试（图 3）；其中 2 个存在 GPU 卸载绕过，因此不纳入性能指标。我们报告其余 21 个有效内核的 GPU 时间。适合并行处理的内核收益显著。对于雅可比（$489\times$），这是通过持久化设备数据区域减少冗余传输实现的。带宽受限内核的性能提升较为有限（例如，*geodesic*： $1.00\times$，*pool*： $1.17\times$）。在 21 个有效内核中，PARACODEX 实现了 $3\times$ 的 GPU 时间几何平均加速比（中位数为 $1.59\times$）。基线的稳健性较差，有 2 个基准测试编译失败，成功率为 90%（19/21），其几何平均值与之相当，为 $2.4\times$，中位数为 $1.74\times$。PARACODEX 的编译成功率达到 100%（21/21），并在 18/21（86%）个内核上缩短了运行时间。

E.3 Rodinia：翻译可靠性与系统性性能提升

PARACODEX 成功转换了 Rodinia 的所有内核，并实现了几何平均 GPU 执行时间加速比（中位数为 $6.24\times$）（图4-左）。由于许多 Rodinia 程序包含正确但未充分优化的 OpenMP 代码，PARACODEX 经常通过重构并行区域、减少同步以及降低数据移动开销来提升性能。基线也为所有内核生成了可运行的实现，但几何平均加速比仅为 $3\times$（中位数为 $3.41\times$）。

E.4 NAS：处理复杂的专家级代码库

图4 总结了 NAS 结果（右）。与高度优化的参考实现相比，PARA-CODEX 的 GPU 时间加速比中位数为 $1.01\times$，几何平均值为 $1.08\times$ 。EP 和 FT 的提升空间最大，而 MG 对端到端传输开销较为敏感。这些结果表明，PARACODEX 在多个内核上能够达到专家级参考实现的水平，同时在高度调优的代码上也保持竞争力。基线的 GPU 时间加速比中位数为$0.927\times$，几何平均值为 $0.834\times$，且无法编译 FT。附录G展示了 NAS MG 上基线与 PARACODEX 输出结果的详细对比，其中 PARACODEX

通过性能分析反馈驱动的内核融合，实现了 1.57× 倍更优的 GPU 时间。

F 与先前工作的比较

F.1 UniPar

图 7展示了持续迭代智能体在串行到 OpenMP 翻译任务中取得的显著提升，尤其是在明确的执行计划指导下。该图报告了 13 个 UniPar（Bitan 等人，2025年）内核中通过验证的 HeCBench OpenMP 翻译所占比例，这些内核也出现在我们的评估集中。这些内核的核心计算都位于单个源文件中。

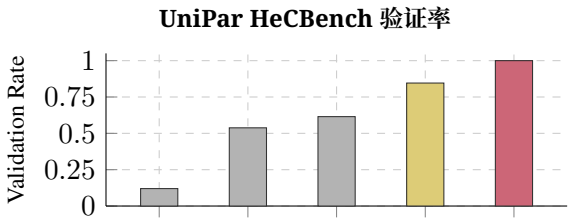


图 7：**不同 UniPar 模型生成的代码通过验证的百分比**将 Codex 基线和 PARACODEX 与 UniPar 方法进行比较，后者采用 GPT-5-mini 基础模型以及先前工作中报告的微调后的 LLaMA 模型。结果凸显了更新、更先进的智能体模型带来的显著验证率提升。

F.2 LASSI

Dearing 等人（2025年）提出了一种适用于相关但不同任务的成功流水线：将 CUDA 翻译为 OpenMP。他们的系统在多个基础模型上的平均表现为，在十个 HeCBench 内核上实现了 85% 的有效翻译率。由于他们的工作报告的是相对于原始参考实现的性能，我们在这部分内核上使用相同指标评估我们的方法，以便进行近似比较，尽管两者的翻译目标有所不同。

图 8报告了比率指标，该指标由Dearing 等人（2025年）提出，定义为参考执行时间除以翻译后的内核执行时间。我们给出了由基线 Codex 智能体和 PARACODEX 翻译的内核的比率，并列出 LASSI 报告的跨模型平均比率。尽管 LASSI 在两块 A100 GPU 上进行评估，而 PARACODEX 在单块 RTX4060 上运行，但两种情况下的参考时间均在与翻译后内核相同的硬件上测得，因此大幅减轻了硬件驱动的偏差。

需要强调的是，LASSI 评估的是 CUDA→OpenMP 翻译，而 PARACODEX 执行的是串行→OpenMP 翻译，因此两者并非严格的直接比较。尽管如此，观察到的巨大差距——例如雅可比 内核实现的 489× Ratio——再加上始终更高的平均 Ratio，表明精心设计的代理工作流（如 PARACODEX）能够比开箱即用的 Codex 和以往的自动化流水线带来显著的性能优势。

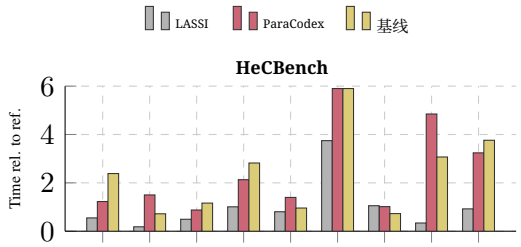


图 8：**参考时间 / 翻译后内核运行时间**。为便于可视化，Jacobi 的 ParaCodex 和基线数据均截断至 6.0（实际值：PARACODEX 为 489.6，基线为 326.1）。

G 示例代码转换：NAS MG 与基线对比

为说明 PARACODEX 为何优于零样本基线，我们比较了两者针对 NAS MG 多重网格求解器残差模板内核生成的结果。基线生成了正确的双遍实现，并使用设备分配的临时数组；而 PARACODEX 的分阶段工作流则通过性能分析反馈，将这一模式优化为融合单遍结构。

G.1 基线输出（零样本 Codex）

基线生成了一种保守的双遍结构，将邻居计算与残差更新分开：

结构：基线使用主机上的 malloc 分配两个大型临时数组（u1 和 u2，每个包含 1970 万个双精度数，总计 316 MB），在主机上-

使用 malloc，然后使用 OpenMP 的 map(alloc:) 子句将它们映射到设备上。

第 1 遍——邻点和计算：第一个 OpenMP 目标内核遍历三维网格（不包括边界点），并沿两个不同的轴计算四邻点和。对于每个网格点 (i3,i2,i1)，它从输入数组 ou 中读取四个邻点，并将和写入分配在设备上的临时数组 u1 和 u2。求解过程中，此遍执行 170 次（每次 resid 调用一次）。

第 2 遍——残差计算：第二个 OpenMP 目标内核从全局内存中的临时数组 u1 和 u2 读取预先计算的和，同时读取原始数组 ou 和右端项 ov，并将最终残差值写入输出数组 orr。此遍同样执行 170 次。

性能特征：此实现会产生以下开销：(1) 每次调用都在主机端进行 malloc/free 的开销；(2) 总计 340 次内核启动（每次 resid 调用启动 2 次，共 $\times$ 170 次调用）；(3) 冗余的全局内存流量（第 1 遍将邻点值写入设备内存，第 2 遍再读回）；(4) 两遍之间的隐式同步屏障。GPU 时间：7152 毫秒（C 类）。

G.2 ParaCodex 输出（StagedWorkflow）

PARACODEX 的分析阶段识别出模板模式，并将其标记为内核融合候选。初始翻译阶段生成了与基线类似的正确实现。随后，优化阶段对该代码进行分析，观察到较高的内核启动次数（340 次启动），并检测到内存流量模式（先写入临时数组，再从临时数组读取），于是应用了内核融合。

结构：优化后的代码完全消除了 u1 和 u2 临时数组。它使用单个 OpenMPtarget 内核，并通过 collapse(2) 对外部两个维度进行并行化。

融合阶段 – 基于寄存器的计算：在三重嵌套循环中，代码计算六个寄存器变量（u1_c、u1_L、u1_R、u2_c、u2_L、u2_R），这些变量保存直接从输入数组 ou 计算得到的邻点和。每个邻点和通过数组访问聚合四个邻点值，并使用显式的索引运算（例如，使用 I3D(i3,i2-1,i1) 表示下方的邻点）。计算出这些寄存器临时变量后，代码立即在同一次循环迭代中使用它们更新残差数组 orr。寄存器值从不会写入全局内存。

性能改进：此变换-换 (1) 消除了 316MB 设备内存分配和主机端 malloc 开销；(2) 将内核启动次数从 340 次减少到 170 次（减少 50%）；(3) 消除了临时值写入后再读取的全局内存往返；(4) 消除了遍历之间的隐式屏障。邻域值始终保留在寄存器中，从而提高了缓存利用率。**GPU 时间：4569 毫秒，产生了 1。57$\times$ 相对于基线的加速比（C 类）。**

G.3 总结

此示例展示了 PARACODEX 的性能分析引导工作流如何系统地识别并消除单次生成所遗漏的性能瓶颈。基线采用保守策略，通过显式物化中间结果来优先确保正确性，但这会带来大量开销。PARACODEX 的分阶段方法使其能够先确保正确性，再利用性能分析数据安全地应用激进优化（内核融合、寄存器提升、启动次数减少），从而节省基线 GPU 时间的 57%，同时保持逐位相同的输出。