

PARACODEX: A Profiling-Guided Autonomous Coding Agent for Reliable Parallel Code Generation and Translation

Erel Kaplan¹ Tomer Bitan¹ Lian Ghrayeb¹ Le Chen²
Tom Yotam³ Niranjan Hasabnis³ Gal Oren^{1,4}

¹Technion – Israel Institute of Technology, Haifa, Israel

²Argonne National Laboratory, Lemont, USA

³Code Metal, USA

⁴Stanford University, Stanford, USA

{erel.kaplan, tomerbitan, lian.ghrayeb}@campus.technion.ac.il

lechen@anl.gov, {tom, niranjan}@codemetal.ai,

galoren@stanford.edu

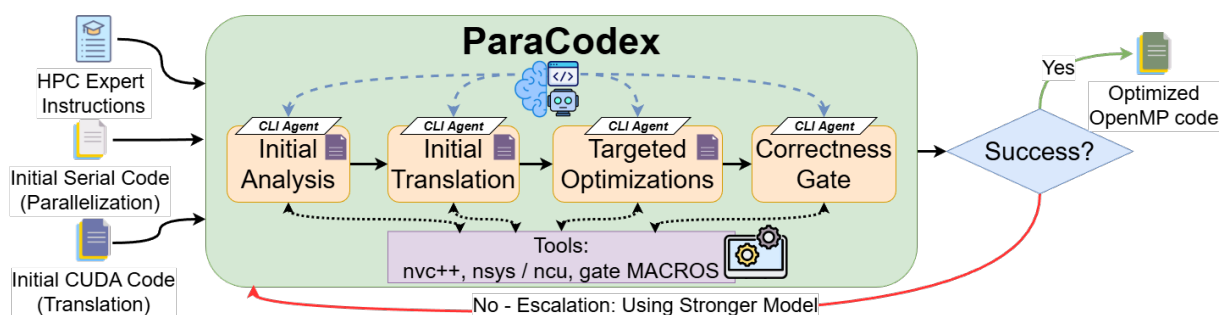


Figure 1: **PARACODEX: An autonomous LLM agent that parallelizes and migrates code.** The core idea is to turn generation into an artifact-driven, tool-verified agentic workflow: the agent first extracts bottlenecks and a data-mapping plan, then proposes patch-level OpenMP target edits, and finally uses compiler/tests as a hard correctness gate and profiler traces as a performance signal. By separating *what to move* (data planning) from *what to offload* (kernel translation) and closing the loop with measurement, PARACODEX reduces brittle one-shot outputs and makes iterations reproducible via structured plans, logs, and profiles.

Abstract

Parallel programming is central to HPC and AI, but producing code that is correct and fast remains challenging, especially for OpenMP GPU offload, where data movement and tuning dominate. *Autonomous coding agents* can compile, test, and profile on target hardware, but outputs are brittle without domain scaffolding.

We present PARACODEX, an HPC-engineer workflow that turns a Codex-based agent into an autonomous OpenMP GPU offload system using staged hotspot analysis, explicit data planning, correctness gating, and profiling-guided refinement. We evaluate translation from serial CPU kernels to OpenMP GPU offload kernels on HeCBench, Rodinia, and NAS. After excluding five kernels, PARACODEX succeeded on all 31 valid kernels. In 27/31 (87%) of these valid cases, the generated kernels improved GPU time over the reference implementations, a result that holds independently on both the A100 and RTX 4060. The resulting OpenMP kernels achieve geometric-mean

speedups of $3.1\times$ (A100) and $3.6\times$ (RTX 4060) on HeCBench and $1.5\times$ and $1.1\times$ on Rodinia, and outperform a zero-shot Codex baseline on all suites. We also evaluate CUDA→OpenMP offload translation on ParEval, where PARACODEX maintains high compilation and validation rates in code-only and end-to-end settings.¹

1 Introduction

Parallel programming is central to modern high-performance computing, but producing parallel implementations that are both correct and fast remains arduous. Developers face two recurring needs: *introducing* parallelism into serial kernels to exploit CPUs and GPUs, and *migrating* existing parallel code between programming models to improve portability and maintainability.

Crucially, this work studies *tool-using LLM code autonomous agents* for program transformation:

¹PARACODEX: <https://github.com/Scientific-Computing-Lab/ParaCodex>

systems that do not stop at text-only code generation, but iteratively compile, test, and profile their outputs on target hardware, using structured intermediate artifacts (e.g., hotspot analyses and data plans) and environment feedback as first-class signals for refinement (Gottschlich et al., 2018).

Classical automation tools like polyhedral compilers (Bondhugula et al., 2008; Doerfert et al., 2015) reduce manual effort but struggle with non-affine loops and ambiguous pointer aliasing, often requiring expert intervention to inject pragmas or resolve dependencies (Harel et al., 2020; Mosseri et al., 2020). Conversely, while Large Language Models (LLMs) can generate plausible OpenMP directives, they lack the *system-level awareness* to manage data movement efficiently (Nichols et al., 2024b; Bitan et al., 2025; Dearing et al., 2025). A common failure mode is “thrashing,” where an LLM correctly parallelizes a loop but fails to map a helper function or hoist data allocations, causing the runtime to implicitly copy arrays back and forth between CPU and GPU at every iteration, yielding code that is functionally correct but much slower than serial execution.

Unlike text-only code generation, *autonomous coding agents* (Li et al., 2025) integrate code models with tool use in terminals or IDEs, enabling iterative editing, compilation, and testing directly on the target machine (Yang et al., 2024). This shifts the setting from offline code completion to *in-situ*, feedback-driven development. We focus on OpenMP GPU offloading (Deakin and Mattson, 2023) because it addresses the “ultimate goal” of modern parallel computing with one of the most used parallel APIs (Valero Lara et al., 2025; Kadosh et al., 2023a): exploiting the massive performance advantage of GPUs over CPUs, which has become ubiquitous over the last decade (TOP500, 2025). By targeting OpenMP offloading, we tackle a triple challenge: (i) maintaining portability across heterogeneous systems, (ii) preserving CPU fallback (since OpenMP offloads are optional), and (iii) raising the bar for agentic reasoning, requiring the agent to correctly manage device memory and kernel launches, not just thread-level parallelism. We evaluate two input settings: *translation* (serial→OpenMP) and *migration* (CUDA→OpenMP), among the most important tasks in the field (Chen et al., 2024a).

These limitations motivate the design of PARACODEX, a Codex-based agentic workflow (Horikawa et al., 2025; Vangala et al., 2025)

that mirrors the iterative process of an HPC engineer (Mondesire et al., 2025). To prevent premature parallelization of unsafe loops (a classical tool weakness), PARACODEX enforces Stage 1 (Analysis). To avoid data thrashing, Stage 2 (Data Planning) explicitly structures memory residency before code generation. Finally, to catch performance regressions that static analysis misses, Stage 3 (Profiling) closes the loop with ground-truth measurement (Peng et al., 2024). Unlike approaches that stop once tests pass, PARACODEX explicitly optimizes for performance via feedback.

We evaluate PARACODEX on 36 serial→OpenMP tasks drawn from HeCBench (Jin and Vetter, 2023) (23), Rodinia (Che et al., 2009) (7), and NAS (Bailey et al., 1991; Fridman et al., 2025) (6). Together, these widely used suites span micro-kernels through end-to-end scientific benchmarks, with provided implementations ranging from optimized to highly tuned. Using GPU time measured by NVIDIA Nsight Systems (kernel execution plus transfers), PARACODEX achieves valid GPU offload on 31/36 kernels (86%; 5 kernels excluded, detailed in §4.2), and improves GPU time over the provided OpenMP implementations in 27/31 cases (87%), a count that holds independently on both the A100 and RTX 4060, with geometric-mean GPU-time speedups of $3.1\times$ on HeCBench, $1.5\times$ on Rodinia, and $2.95\times$ on the highly tuned NAS suite (A100). As a separate generalization study, we evaluate CUDA→OpenMP migration on four ParEval (Nichols et al., 2024a) kernels, where PARACODEX maintains high compilation and validation rates under both a fixed build environment (*code-only*) and an end-to-end setting where the agent must also construct the build flow (*overall*) (§4).

Key Question. *To what extent can a tool-using autonomous coding agent translate and migrate parallel code under automated build-and-test validation, and use profiling feedback to narrow the performance gap to reference implementations?*

Research Questions. To explore this space we pose four guiding questions:

- RQ1 – Baseline feasibility:** How effectively can contemporary agentic LLMs translate serial code into correct OpenMP GPU offload without profiling-guided iteration?
- RQ2 – Benefit of specialization:** Does a staged, expert-seeded workflow with structured intermediate artifacts improve robustness and per-

formance?

3. **RQ3 – Performance attainment:** To what extent can profiling-in-the-loop refinement approach or exceed the provided OpenMP implementations across diverse benchmarks, and where do regressions persist?
4. **RQ4 – Task generalization:** Does the same workflow generalize from serial→OpenMP translation to CUDA→OpenMP migration?

Contributions. We make three contributions:

1. We systematize HPC engineering practice into a reusable agentic pattern: from hotspot analysis via a loop taxonomy through data residency and transfer strategy selection to profile-guided tuning, connected via Makefile-based correctness gates (RQ2, RQ3).
2. We design a comprehensive evaluation protocol for serial→OpenMP agentic parallelization across HeCBench, Rodinia, and NAS under a unified build-and-test harness and a zero-shot Codex baseline, quantifying both robustness and GPU-time performance relative to expert implementations (RQ1, RQ3).
3. We demonstrate generalization to CUDA→OpenMP translation on ParEval, and introduce bypass-detection analysis that surfaces pseudo-offload cases and suggests harness design strategies to enforce device-side work (RQ4).

2 Related Work

We organize prior work from static and learned parallelization methods, through LLM-based parallel code generation and translation, to autonomous coding agents and feedback-driven self-refinement. We distinguish PARACODEX from prior work along three axes: (1) Compile/test repair (UniPar, LASSI) fixes bugs but ignores speed; (2) Performance feedback (PerfCodeGen, STOP) optimizes runtime but lacks rigorous correctness gates; and (3) Artifact-driven planning – absent in prior agents – externalizes reasoning before coding. PARACODEX is, to our knowledge, the first system to integrate all three axes.

Compilers and Learned Parallelization. Compiler research explored automatic loop transformations via dependence analysis and the polyhedral model (Bondhugula et al., 2008; Doerfert et al., 2015). Such tools excel at affine loop nests but typically require expert intervention. Survey

work analyzed source-to-source OpenMP compilers, highlighting pitfalls and variability (Harel et al., 2020; Mosseri et al., 2020). Recent efforts apply learned cost models, RL, and autotuning to optimization (Chen et al., 2019; Ahn et al., 2019; Wu et al., 2023; Ding et al., 2023; Merouani et al., 2025), relying on iterative search and measurement. ML techniques predict parallelization opportunities and suggest OpenMP directives (Maramzin et al., 2019; Chen et al., 2023; Harel et al., 2023; Kadosh et al., 2024a; Harel et al., 2025), but stop short of generating complete code.

LLMs for Parallel Code. Domain-focused LLMs (HPC-Coder, OMPGPT, MonoCoder) specialize in parallel kernels (Nichols et al., 2024b; Chen et al., 2024b; Kadosh et al., 2024b; Schneider et al., 2024; Kadosh et al., 2023b; Chen et al., 2024a). Broader models (CodeGeeX, Meta’s LLM Compiler) extend coverage (Zheng et al., 2024; Cummins et al., 2024). Transformer advisors (OMPify, MPI-ricol) provide data-driven guidance (Kadosh et al., 2023c; Schneider et al., 2023, 2024). However, these typically deliver single-shot outputs without iterative feedback.

Autonomous Coding Agents. Recent work describes *autonomous coding agents*: systems pairing code models with developer tools in iterative loops (Li et al., 2025). Examples include CLI assistants and IDE integrations (GitHub Copilot, OpenAI Codex, Cursor). This is relevant for parallel code, where correctness requires build-and-test harnesses and performance depends on concrete hardware. Codex has been evaluated on code-generation benchmarks with strong performance.² We build PARACODEX on Codex. Tool access alone is insufficient: without domain structure, workflows can be brittle. Prior pipelines emphasize compilation repair and correctness (Bitan et al., 2025; Dearing et al., 2025; Chen et al., 2025), whereas PARACODEX operationalizes an HPC-engineer with explicit artifacts, correctness gates, and profiling-guided optimization.

LLM-Based Translation Pipelines. Frameworks such as UniPar and LASSI study LLMs for translating parallel code, including serial→OpenMP (Bitan et al., 2025; Dearing et al., 2025; Chen et al., 2025). They implement *compile/test-driven repair*: compiler errors and

²<https://openai.com/index/introducing-gpt-5-2-codex/>

test failures trigger iterative fixes, improving compilation and correctness. However, they lack (i) *execution/performance feedback* via profiling to guide optimization, and (ii) *artifact-driven structured plans* that externalize reasoning before code modification. Complementary work quantifies LLM capability on HPC kernel generation (Godoy et al., 2023; Valero-Lara et al., 2023; Cui et al., 2025; Bolet et al., 2025).

Self-Refinement and Feedback-Driven Optimization. Agentic research explores deliberate search and refinement for program synthesis (Yao et al., 2023; Shinn et al., 2023; Singhal et al., 2025; Madaan et al., 2023; Jimenez et al., 2024). PerfCodeGen demonstrates execution-driven refinement by feeding performance metrics back (Peng et al., 2024), and Self-Taught Optimizer (STOP) scaffolds LLMs to recursively optimize programs (Zelikman et al., 2023). These systems show the value of *execution feedback*, but do not enforce *correctness gating* at each stage or emit *structured intermediate artifacts* (e.g., data plans, optimization plans) that anchor reasoning to domain constraints. PARACODEX combines all three mechanisms – compile/test repair, profiling feedback, and artifact-driven planning – within a unified HPC-engineering workflow.

3 PARACODEX Agent Overview

PARACODEX is a guided agentic workflow in which the model iteratively uses standard development tools. The agent first analyzes the input program to identify performance-critical loops or kernels, then generates an OpenMP GPU version. It compiles and executes the code against the serial reference, using compilation errors and numerical mismatches as structured feedback to drive targeted fixes until correctness is achieved.

The workflow then shifts to performance tuning. The agent profiles execution with a profiler and applies focused optimizations to further improve performance. In combination, correctness-driven repair and profile-guided tuning enable robust automated optimization.

3.1 Baseline Comparison System.

Throughout this paper, we use the term *baseline* to refer to a zero-shot Codex CLI agent³ that directly translates the input program into OpenMP GPU

³<https://platform.openai.com/docs/models/gpt-5.1-codex>

offload in a single pass. Both baseline and PARACODEX are evaluated under the same build and correctness harness; PARACODEX additionally uses staged prompts and profiling-guided refinement. This setup keeps the underlying model fixed and isolates the impact of workflow scaffolding. The baseline is explained in more detail in App. B.6.

3.2 Pipeline Stages

PARACODEX executes a three-stage workflow as depicted in Figure 1. (See App. A for details on suite-specific adaptations.)

- 1. Analysis: Loop classification and data characterization.** Inspects input code to identify and rank candidate loops by computational weight, defined as the estimated operation count (iterations \times ops/iteration) to distinguish $O(N)$ critical paths from setup code. It generates `analysis.md`, which classifies loops using a taxonomy (Types A–G) and records data properties, dependencies, and hazards (App. B). *Rationale:* Externalizing reasoning before code modification reduces chances of premature transformation of loops with hidden hazards (e.g., reductions, recurrences) that would silently break correctness.
- 2. GPU offloading + data strategy: Correctness with explicit data plan.** Before inserting pragmas, the agent selects a data-management strategy based on the loop taxonomy: (A) *Scoped Regions* use explicit `target data` directives for standard kernels to minimize transfers; (B) *Asynchronous Pipelines* use `nowait` for overlapping computation and communication; (C) *Global Device State* allocates persistent device memory (`omp_target_alloc`) for iterative solvers to avoid repeated mapping. It writes `data_plan.md` specifying array residency and function offload status, prioritizing *CRITICAL* loops (those with high computational weight) for offload. This explicit planning step reduces the chances of common “thrashing” pitfalls, where unmapped helper functions trigger implicit host-device transfers at every iteration.
- 3. Performance tuning: Profile-guided optimization.** After functional validation, the agent profiles the program with NVIDIA Nsight Systems, extracting kernel times, transfer volumes, and API overheads. It generates an `optimization_plan.md` that ranks bottlenecks and prescribes targeted fixes (e.g., fusing adjacent kernels to reduce launch latency, col-

lapsing loops to increase occupancy, hoisting transfers out of iterative loops). A strict revert-on-regression policy is enforced: if GPU time worsens by more than 10%, the change is automatically rolled back.

4. **Correctness gating.** Between each stage, a *correctness-gate agent* is invoked. The agent instruments the code with `gate.h` – a lightweight header that computes checksums and norms at key program points – to localize the divergence and drive the repairs. It then compiles and runs the candidate code against the input serial code using a manually-written Makefile-based harness. Then, it corrects the code if the validation fails (via exit code or numerical mismatch). (App. B.3).

For space purposes, a concrete example showing the end-to-end workflow of hotspot analysis, data plan, and optimization is kept in App. B.5.

3.3 Tooling Integration

The pipeline uses external tools accessible to the agent. Compiler diagnostics from NVIDIA HPC SDK surface issues; Makefile-based harnesses validate correctness; NVIDIA Nsight Systems provides performance diagnosis; system metadata is read from `system_info.txt`. The design combines LLM-driven generation with deterministic tool feedback for iterative repair and tuning.

3.4 Prompt Engineering Strategy

PARACODEX uses multi-stage prompting that structures the model’s reasoning across analysis, translation, and optimization. Each stage is driven by a specific template that constrains the output space while retaining domain flexibility.

1. **Analysis Phase.** The model produces a structured hotspot analysis (App. B.1) that characterizes loops using a taxonomy (Types A–G) capturing parallelization constraints. This prompt converts qualitative reasoning into a discrete intermediate representation, revealing any hazards before transformation.
2. **Translation Phase.** Using the structured analysis, the prompt maps each loop class to a restricted set of valid OpenMP constructs (App. B.2), minimizing mapping errors like omitted reductions.
3. **Optimization Phase.** This stage implements measurement-guided search (App. B.4). The prompt pairs Nsight Systems profiles with optimization levers to apply targeted modifications,

retaining only those that improve performance.

4 Experimental Evaluation

4.1 Experimental Setup

Environment and Protocol. We evaluate PARACODEX on two NVIDIA GPU platforms: an A100 (40GB, compute capability 8.0) and an RTX 4060 Laptop GPU (8GB, compute capability 8.9) with NVIDIA HPC SDK 25.7 (`nvc++`). We report *GPU time* measured via NVIDIA Nsight Systems, computed by summing CUDA kernel time (`cuda_gpu_kern_sum`) and device transfer time (`cuda_gpu_mem_time_sum`), excluding CUDA API and runtime overhead. All reported results are averaged over 10 runs and reported as mean \pm standard deviation, see App. H. For NAS, we used Class C inputs on the RTX 4060 and Class D inputs on the A100, except for the FT kernel, which is memory-bound and therefore required Class C on both GPUs. For HeCBench and Rodinia, input sizes were scaled so that the slowest variant runs for approximately 10–20 seconds, improving measurement stability and making GPU-side differences more meaningful.

GPU Time Metric. We define *GPU time* as the sum of CUDA kernel execution time and device memory transfer time from NVIDIA Nsight Systems, excluding CUDA API overhead. This captures computational work and data movement on the device. The implications are that (i) GPU time represents device-side performance when offload is substantial; (ii) it can diverge from wall-clock time when API/host overhead dominates; (iii) CPU-fallback cases yield misleadingly low GPU times as detailed in §4.2; (iv) if there are no kernel launches at all, the profiler automatically detects it.

Model Selection and Escalation. We use `gpt-5.1-codex-mini` as the primary model for all kernels in both PARACODEX and the baseline. For the NAS FT kernel, which failed with the base model, we escalated to `gpt-5.1-codex-max` for both PARACODEX and the baseline. In all experiments, the agent operated without internet access and did not use web search at any stage.

Benchmarks and Baselines. We use ParEval (Nichols et al., 2024a), HeCBench (Jin and Vetter, 2023), Rodinia (Che et al., 2009), and NAS (Bailey et al., 1991; Fridman et al., 2025).

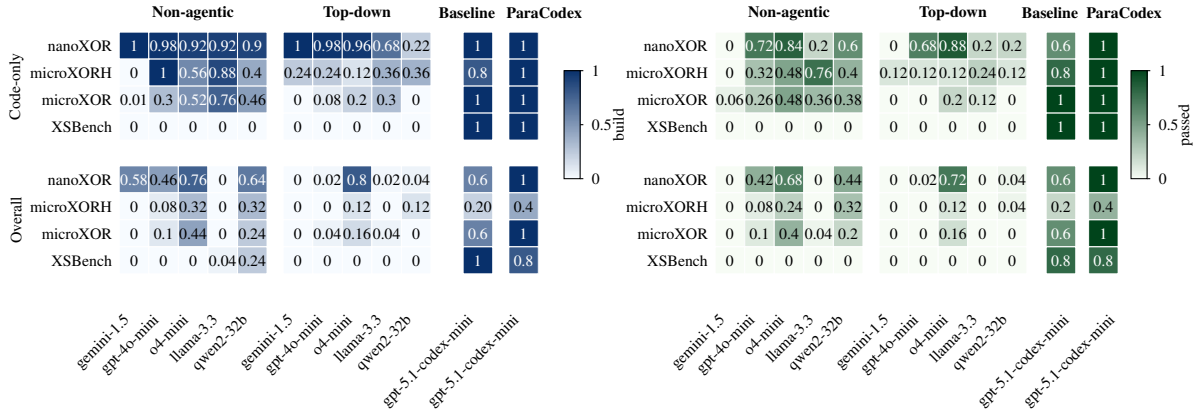


Figure 2: **CUDA→OpenMP translation quality on ParEval across scoring regimes.** Compilation (blue) and validation (green) success rates (higher is better) are shown for both the baseline Codex model and PARACODEX under the ParEval protocol. Kernels are scored under (i) a fixed build environment (*code-only*) and (ii) a setting where the system must also construct the build flow (*overall*). The heatmap layout mirrors the original ParEval presentation. PARACODEX matches or slightly exceeds the baseline, indicating that the agentic workflow generalizes to cross-API translation.

Suite	Attempted	Valid	Improved
HeCBench	23	21	19/21 (90%)
Rodinia	7	6	5/6 (83%)
NAS	6	4	3/4 (75%)
ParEval	4	4	–
serial→OpenMP	36	31	27/31 (87%)

Table 1: **Result accounting across suites.** *Attempted*: kernels tried; *Valid GPU*: compile + pass correctness with substantial device execution; *Improved*: PARACODEX improves GPU time among valid GPU kernels. Per-suite counts reflect A100 measurements; the aggregate 27/31 total holds independently on both the A100 and RTX 4060 (on RTX 4060: Rodinia 4/6, NAS 4/4). ParEval reports CUDA→OpenMP separately. Detailed per-kernel results in App. H.

From HeCBench, our evaluation considers 23 kernels that have been previously benchmarked in prior work. These consist of the 10 kernels analyzed in LASSI (Dearing et al., 2025) together with 13 kernels from the ParaTrans dataset utilized in UniPar (Bitan et al., 2025). *Reference implementations*: For all suites, the reference code is the provided OpenMP GPU-offload implementation from each benchmark; we compare PARACODEX’s generated OpenMP against those OpenMP baselines. *Baseline for PARACODEX*: We compare PARACODEX against a zero-shot Codex baseline that receives the input code and a single prompt, as further detailed in §3.1. Both systems use the same constraints: identical build harness, correctness checks, profiling tools, and GPU/compiler environment. The key difference is the *workflow*. For each benchmark, we report

speedup relative to reference as $T_{\text{ref}}/T_{\text{pc}}$, where values greater than one indicate lower execution time.

4.2 Success Rate and Bypass Detection

Table 1 provides a comprehensive accounting of kernel counts, exclusions, and success rates. We evaluate 36 serial→OpenMP translation tasks (HeCBench: 23, Rodinia: 7, NAS: 6). We exclude 5 kernels from the final performance set: two from NAS (*BT*, *LU*) due to multi-file complexity; one from Rodinia (*srad*) due to CPU-only reference implementation; and two from HeCBench (*pathfinder*, *particlefilter*) due to GPU-offload bypass (see App. D.2). This leaves 31 valid GPU-offload kernels.

Bypass was observed in 2/23 HeCBench kernels for PARACODEX, where implementations compile and pass correctness checks but execute the primary computation on the host CPU. We exclude bypass kernels from GPU-time statistics and provide detailed analysis in App. D.2.

Token Budget and Reproducibility. PARACODEX averages 837,701 tokens per kernel ($1.11\times$ baseline). This cost reflects a deliberate prioritization of *reproducibility over efficiency*. Rather than brute-force sampling, we operate the agent in a non-interactive automation mode where full build logs and profiler traces are fed back to verify every step deterministically. This ensures the entire engineering loop is auditable and replayable without human intervention. To further support reproducibility, almost all experiments were run using `gpt-5.1-codex-mini` with the lowest reason-

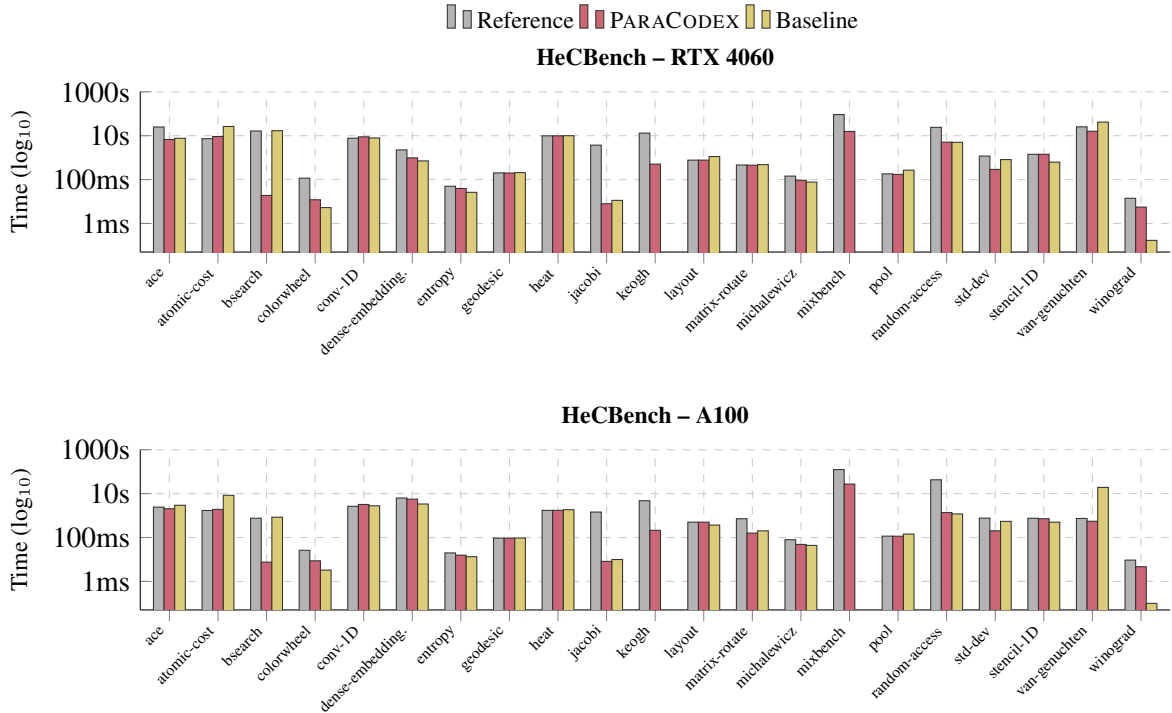


Figure 3: **GPU-time performance of PARACODEX and baseline Codex on HeCBench.** Results are shown for both the A100 (bottom) and RTX 4060 (top). GPU time (log scale; lower is better) is reported relative to the HeCBench references. Each kernel reports baseline Codex and PARACODEX performance; missing baseline bars indicate failures to produce a correct/compilable implementation. Both systems generally outperform the references, while PARACODEX achieves higher geometric-mean and median gains across the suite.

ing effort, ensuring that independent researchers can replicate the outputs under the same configuration. Notably, the zero-shot baseline consumes comparable tokens (755,417) under the same harness, confirming that the cost stems from the robust tool-verified environment – which accumulates context – rather than the multi-stage logic itself (App. D.1).

4.3 Performance Analysis of PARACODEX

Benchmark Suite Evaluation. We evaluate PARACODEX across four diverse benchmark suites: ParEval (CUDA→OpenMP translation fidelity (App. C details the Serial→OpenMP to CUDA→OpenMP migration), Figure 2), HeCBench (23 diverse micro-kernels, Figure 3), Rodinia (application-level benchmarks, Figure 4-left), and NAS (expert-optimized codes, Figure 4-right). PARACODEX demonstrates strong translation reliability: ParEval shows perfect code-only validation (4/4 kernels at 100%) with strong overall-regime performance (2/4 at 1.0, 1/4 at 0.8); HeCBench achieves 100% compilation success (21/21 valid kernels after excluding 2 bypass cases). Performance gains are substan-

tial: HeCBench shows $3.1\times$ (A100) and $3.6\times$ (RTX 4060) geometric-mean GPU-time speedup (19/21 improved), while Rodinia achieves $1.5\times$ (A100) and $1.1\times$ (RTX 4060) geometric-mean speedup. Further comparison with related literature is presented in App. F.1 and F.2.

To assess whether PARACODEX’s gains transfer across architectures, we evaluate the final implementations on both an RTX 4060 and an NVIDIA A100. Across benchmark suites, PARACODEX preserves its improvements relative to the reference implementations on both systems, indicating that the generated optimizations are not narrowly tied to a single device class.

On NAS (A100), PARACODEX achieves parity with the highly optimized reference on CG ($1.00\times$) and FT ($1.00\times$), delivers a large speedup on EP ($29.7\times$), while MG regresses modestly ($0.87\times$ on A100, $1.41\times$ on RTX 4060). Reporting mean \pm standard deviation over 10 runs helps distinguish genuine regressions from run-to-run noise; for several NAS benchmarks, the gap between PARACODEX and the reference is small relative to the observed variance. The baseline exhibits lower robustness, failing to compile 2 HeCBench

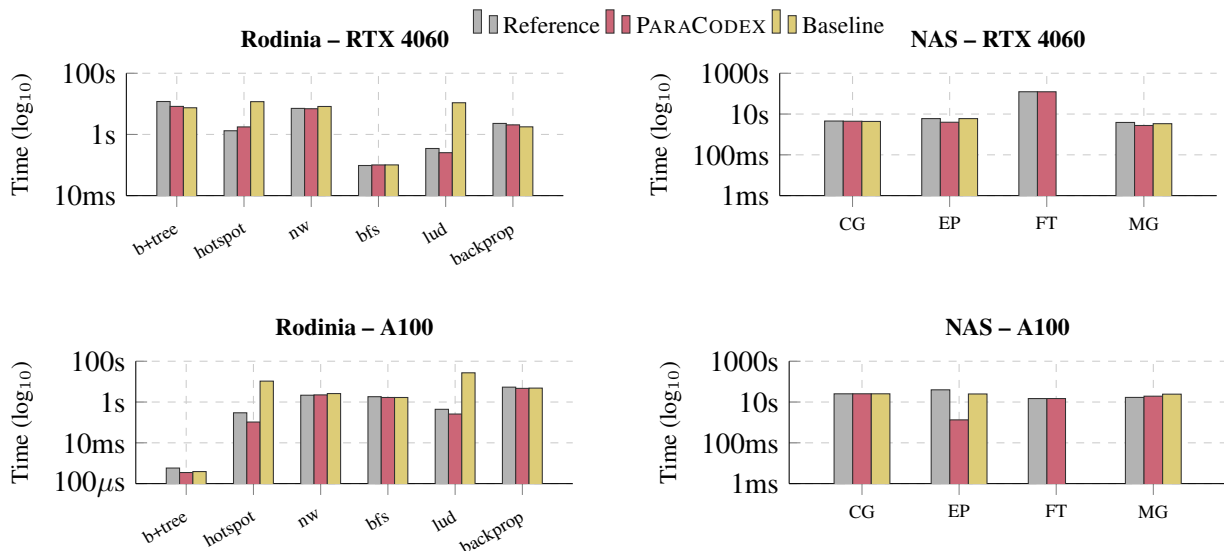


Figure 4: **Performance of PARACODEX and baseline on NAS and Rodinia reference implementations.** GPU time for the OpenMP reference, PARACODEX, and baseline OpenMP translation across the RTX 4060 (top) and A100 (bottom) GPUs. Bars show mean execution time over 10 runs on a log scale; lower is better. Input sizes are scaled so that the slowest variant runs for 10–20 s, making device-level differences measurable.

kernels (keogh, mixbench) on both GPUs and 1 NAS kernel (FT), achieving only $2.0\times$ (HeCBench) and $0.30\times$ (Rodinia) geometric-mean speedups on A100. Detailed per-suite results are provided in App. E, and a code comparison for NAS MG demonstrating $1.57\times$ improvement over the baseline through profiling-driven kernel fusion are provided in App. G. Full benchmark results across all kernels and both GPUs are provided in Appendix H.

4.4 Robustness to Anonymization

To assess whether PARACODEX depends on memorized identifier patterns, we evaluate anonymized variants of Rodinia and NAS, where all variable and function names are replaced with synthetic tokens and function order is randomized. Figure 5 shows that performance is largely unchanged. On anonymized Rodinia, PARACODEX outperforms the baseline in one variant, and achieves comparable results on the other. On NAS, we evaluated the staged workflow, which also remains effective: Initial Translation (Stage 2: GPU offload + data plan) establishes strong baselines, and Targeted Optimizations (Stage 3: profiling-driven tuning) extract further gains (e.g., CG improves to $1.41\times$). These results indicate that performance stems from structural reasoning and profiler feedback rather than lexical memorization.

4.5 Behavioral Analysis: Structured Reasoning Under Agentic Control

We further examine how PARACODEX shapes the model’s reasoning during parallelization. Figure 6 compares baseline and PARACODEX timelines for the NAS EP kernel. Under PARACODEX, traces more often decompose the task into parallelization planning and performance-oriented refinement (e.g., parallel region selection, data movement, synchronization), whereas the baseline tends to emit code directly. The bottom-right panel shows similar behavior: for EP, most gains arise in Initial Translation through correct offload and data strategy, while CG improves more gradually across both stages.

5 Discussion

We revisit our research questions:

RQ1 (Feasibility): Contemporary agents demonstrate strong capability for translating serial code to OpenMP GPU offload. The baseline produces correct and reasonably performant implementations. However, robustness degrades on more complex benchmarks: the baseline fails on NAS FT, exhibits lower compilation success rates overall, and produces less optimized code compared to PARACODEX in all suites. While one-pass generation is often sufficient for simpler kernels, complex scientific codebases with intricate data movement, multi-stage algorithms, and tight synchronization constraints benefit substantially from structured,

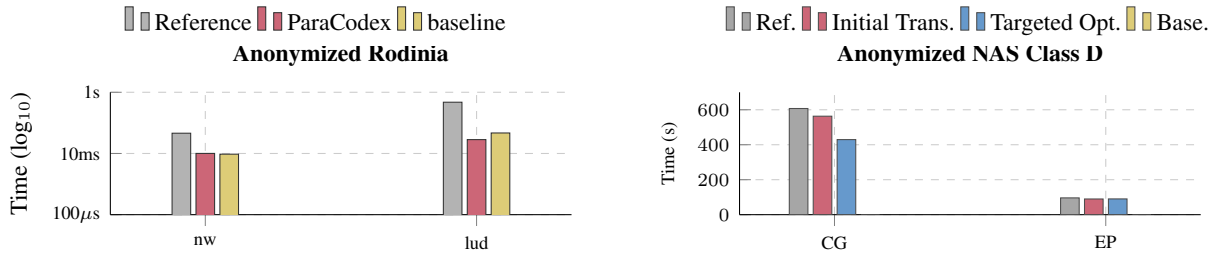


Figure 5: **Performance breakdown and generalization** (RTX 4060). (Left) Anonymized Rodinia GPU execution time (log scale) comparing Reference, PARACODEX, and Baseline on representative kernels. (Right) Anonymized NAS Class D performance showing the impact of PARACODEX’s staged refinement (Initial Translation vs. Targeted Optimizations) compared to Reference. Lower is better.

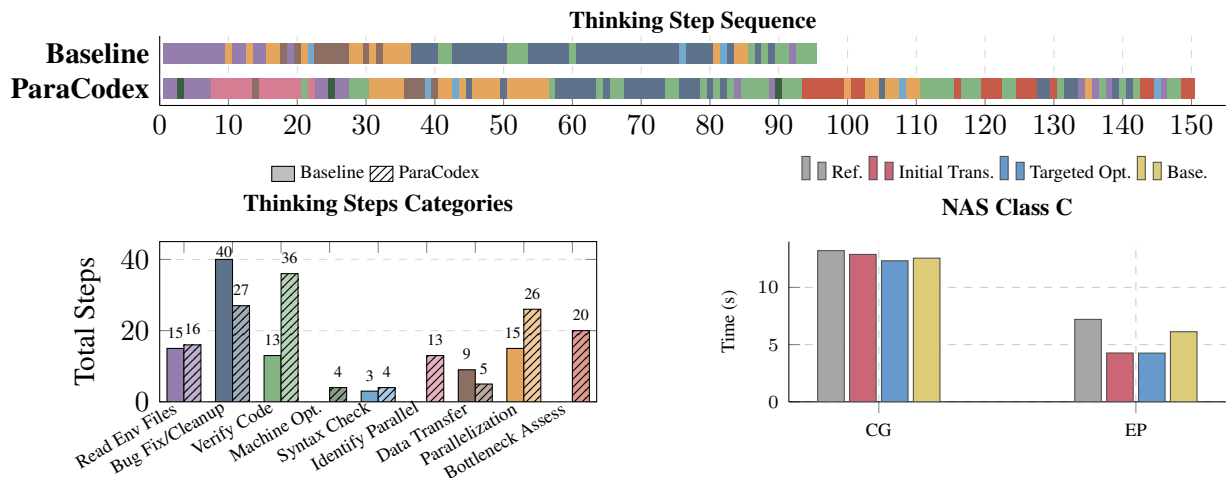


Figure 6: **Agent reasoning traces for the EP kernel (NAS), categorized into nine step types and compared between the zero-shot Codex baseline and PARACODEX.** The per-step timeline (top) and the total number of steps per category (bottom left) show how the staged workflow shifts attention toward task-relevant analysis rather than surface-level output. Bottom right compares staged refinement in PARACODEX on EP and CG (NAS Class C, RTX 4060; lower is better).

iterative refinement.

RQ2 (Specialization): Expert-seeded workflows improve results. PARACODEX consistently boosts compilation rates and speedups by decomposing tasks into analysis, planning, and refinement. As evidenced by reasoning traces, this structure forces the model to externalize its plan before coding, shifting focus from surface-level syntax to performance diagnosis. This allows it to catch logic errors and optimize data transfers that the baseline misses.

RQ3 (Performance): Crafted agents can match experts. PARACODEX achieves substantial geometric-mean speedups on HeCBench and Rodinia, outperforming the majority of reference implementations. On the highly optimized NAS suite, it matches expert performance, demonstrating that profiling-guided refinement can recover substantial headroom. The GPU-offload bypass behavior observed on 2 HeCBench kernels represents a failure mode

where performance-driven optimization violates parallelization intent.

RQ4 (Generalization): The methodology generalizes. On ParEval (CUDA→OpenMP), PARACODEX maintains high validity with minimal prompt adjustment. This indicates that the core agentic pattern – analysis, gated translation, and feedback-driven repair – is agnostic to the source language, applicable to broader parallel translation tasks beyond the primary serial→OpenMP setting.

6 Limitations

Single-GPU scope. Our evaluation focuses on single-GPU execution on two platforms (A100 and RTX 4060). Extending PARACODEX to multi-GPU offloading and distributed coordination is a natural next step, but would require new pipeline stages for collective-communication analysis, device-to-device transfer planning, and correctness gating under non-deterministic parallel execution across

devices. We leave this as an important direction for future work.

GPU-offload bypass. The bypass cases (2/36 kernels, both in HeCBench) represent a failure mode where the agent generates code that compiles with OpenMP `target` directives and passes correctness checks, yet executes the primary computation on the host CPU. In both cases (*pathfinder*, *particlefilter*), agent transcripts suggest the model judged data-transfer overheads prohibitive and implicitly reverted to host execution. Future work should enforce device-execution as a hard harness requirement - for example, asserting minimum kernel-launch counts or device-memory allocation thresholds - rather than relying on agent judgment alone (App. D.2).

Computational cost. PARACODEX’s iterative workflow - analysis, transformation, compilation, validation, and profiling - yields a higher token and wall-clock cost than zero-shot generation. Across NAS benchmarks (representative of the full pipeline), PARACODEX averages 837,701 tokens per kernel versus 755,417 for the zero-shot baseline (a $1.11\times$ overhead). The high absolute token count is primarily an artifact of how coding agents work: source code context, build logs, and profiler traces are included verbatim in each agent turn to support reproducibility and deterministic diagnosis. The small overhead of the staged workflow over the baseline confirms that the cost reflects the environment rather than the scaffolding itself. Reducing this overhead - for example via context compression or learned cost models for early-exit decisions - is an important direction for future work (App. D.1).

Acknowledgments

Computational support was provided by Code Metal.

References

Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. 2019. [Reinforcement learning and adaptive sampling for optimized dnn compilation](#). *Preprint*, 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, Venkatakrisnan Venkatakrisnan, and Sisira K.

Weeratunga. 1991. [The NAS parallel benchmarks](#). *The International Journal of Supercomputing Applications*, 5(3):63–73.

Tomer Bitan, Tal Kadosh, Erel Kaplan, Shira Meiri, Le Chen, Peter Morales, Niranjan Hasabnis, and Gal Oren. 2025. [Unipar: A unified llm-based framework for parallel and accelerated code translation in hpc](#). In *2025 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE.

Gregory Bolet, Giorgis Georgakoudis, Harshitha Menon, Konstantinos Parasyris, Niranjan Hasabnis, Hayden Estes, Kirk Cameron, and Gal Oren. 2025. [Can large language models predict parallel code performance?](#) In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, pages 1–6.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. [A practical automatic polyhedral parallelizer and locality optimizer](#). *ACM SIGPLAN Notices*, 43(6):101–113.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. [Rodinia: A benchmark suite for heterogeneous computing](#). In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54.

Le Chen, Nesreen K. Ahmed, Akash Dutta, Arijit Bhattacharjee, Sixing Yu, Quazi Ishtiaque Mahmud, Waqwoya Abebe, Hung Phan, Aishwarya Sarkar, Branden Butler, Niranjan Hasabnis, Gal Oren, Vy A. Vo, Juan Pablo Muñoz, Theodore L. Willke, Tim Mattson, and Ali Jannesari. 2024a. [The landscape and challenges of HPC research and llms](#). *CoRR*, abs/2402.02018.

Le Chen, Arijit Bhattacharjee, Nesreen Ahmed, Niranjan Hasabnis, Gal Oren, Vy Vo, and Ali Jannesari. 2024b. [Ompgpt: A generative pre-trained transformer model for openmp](#). In *European Conference on Parallel Processing*, pages 121–134. Springer Nature Switzerland Cham.

Le Chen, Quazi Ishtiaque Mahmud, Hung Phan, Nesreen K. Ahmed, and Ali Jannesari. 2023. [Learning to parallelize with openmp by augmented heterogeneous AST representation](#). In *Proceedings of the Sixth Conference on Machine Learning and Systems, MLSys 2023, Miami, FL, USA, June 4-8, 2023*. ml-sys.org.

Le Chen, Nuo Xu, Winson Chen, Bin Lei, Pei-Hung Lin, Dunzhi Zhou, Rajeew Thakur, Caiwen Ding, Ali Jannesari, and Chunhua Liao. 2025. [Beyond code pairs: Dialogue-based data generation for llm code translation](#). *arXiv preprint arXiv:2512.03086*.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. [Learning to optimize tensor programs](#). *Preprint*, arXiv:1805.08166.

- Bowen Cui, Tejas Ramesh, Oscar Hernandez, and Keren Zhou. 2025. [Do large language models understand performance optimization?](#) *Preprint*, arXiv:2503.13772.
- Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. [Meta large language model compiler: Foundation models of compiler optimization.](#) *Preprint*, arXiv:2407.02524.
- Tom Deakin and Timothy G Mattson. 2023. *Programming Your GPU with OpenMP: Performance Portability for GPUs*. MIT Press.
- Matthew T. Dearing, Yiheng Tao, Xingfu Wu, Zhiling Lan, and Valerie Taylor. 2025. [Lassi: An LLM-based automated self-correcting pipeline for translating parallel scientific codes.](#) *Preprint*, arXiv:2407.01638.
- Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. [Hidet: Task-mapping programming paradigm for deep learning tensor programs.](#) *Preprint*, arXiv:2210.09603.
- Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 2015. [Polly’s polyhedral scheduling in the presence of reductions.](#) *Preprint*, arXiv:1505.07716.
- Yehonatan Fridman, Yosef Goren, and Gal Oren. 2025. [From openacc to openmp5 gpu offloading: Performance evaluation on nas parallel benchmarks.](#) In *Proceedings of the 2025 4th International Workshop on Extreme Heterogeneity Solutions*, pages 10–18.
- William F. Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S. Vetter. 2023. [Evaluation of OpenAI codex for HPC parallel programming models kernel generation.](#) *Preprint*, arXiv:2306.15121.
- Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B Tenenbaum, and Tim Mattson. 2018. [The three pillars of machine programming.](#) In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 69–80.
- Re’em Harel, Tal Kadosh, Niranjana Hasabnis, Timothy G. Mattson, Yuval Pinter, and Gal Oren. 2025. [Pragformer: Data-driven parallel source code classification with transformers.](#) *Int. J. Parallel Program.*, 53(1):2.
- Re’em Harel, Idan Mosseri, Harel Levin, Lee-or Alon, Matan Rusanovsky, and Gal Oren. 2020. [Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: Analysis, pitfalls, enhancement and potential.](#) *Int. J. Parallel Program.*, 48(1):1–31.
- Re’em Harel, Yuval Pinter, and Gal Oren. 2023. [Learning to parallelize in a shared-memory environment with transformers.](#) In *Proceedings of the 28th ACM SIGPLAN annual symposium on principles and practice of parallel programming*, pages 450–452.
- Kosei Horikawa, Hao Li, Yutaro Kashiwa, Bram Adams, Hajimu Iida, and Ahmed E. Hassan. 2025. [Agentic refactoring: An empirical study of AI coding agents.](#) *CoRR*, abs/2511.04824.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. [Swe-bench: Can language models resolve real-world github issues?](#) *Preprint*, arXiv:2310.06770.
- Zheming Jin and Jeffrey S. Vetter. 2023. [A benchmark suite for improving performance portability of the sycl programming model.](#) In *Proceedings of the 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 325–327.
- Tal Kadosh, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023a. [Quantifying openmp: Statistical insights into usage and adoption.](#) In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE.
- Tal Kadosh, Niranjana Hasabnis, Prema Soundararajan, Vy A. Vo, Mihai Capota, Nesreen K. Ahmed, Yuval Pinter, and Gal Oren. 2024a. [Ompar: Automatic parallelization with ai-driven source-to-source compilation.](#) *CoRR*, abs/2409.14771.
- Tal Kadosh, Niranjana Hasabnis, Vy A Vo, Nadav Schneider, Neva Krien, Mihai Capotă, Abdul Wasay, Guy Tamir, Ted Willke, Nesreen Ahmed, and 1 others. 2024b. [Monocoder: Domain-specific code language model for hpc codes and tasks.](#) In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE.
- Tal Kadosh, Niranjana Hasabnis, Vy A Vo, Nadav Schneider, Neva Krien, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, Yuval Pinter, and 1 others. 2023b. [Scope is all you need: Transforming llms for hpc code.](#) *arXiv preprint arXiv:2308.09440*.
- Tal Kadosh, Nadav Schneider, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023c. [Advising openmp parallelization via A graph-based approach with transformers.](#) In *OpenMP: Advanced Task-Based, Device and Compiler Programming - 19th International Workshop on OpenMP, IWOMP 2023, Bristol, UK, September 13-15, 2023, Proceedings*, volume 14114 of *Lecture Notes in Computer Science*, pages 3–17. Springer Nature Switzerland Cham, Springer.
- Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. 2025. [The rise of AI teammates in software engineering \(SE\) 3.0: How autonomous coding agents are reshaping software engineering.](#) *CoRR*, abs/2507.15003.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang,

- Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. [Self-refine: Iterative refinement with self-feedback](#). *Preprint*, arXiv:2303.17651.
- Aleksandr Maramzin, Christos Vasiladiotis, Roberto Castaneda Lozano, Murray Cole, and Bjorn Franke. 2019. “it looks like you’re writing a parallel loop”: Assisting developers in parallelization decisions. In *Proceedings of the AISEPS Workshop*.
- Massinissa Merouani, Khaled Afif Boudaoud, Iheb Nassim Aouadj, Nassim Tchoulak, Islem Kara Bernou, Hamza Benyamina, Fatima Benbouzid-Si Tayeb, Karima Benatchba, Hugh Leather, and Riyadh Baghdadi. 2025. [Looper: A learned automatic code optimizer for polyhedral compilers](#). *Preprint*, arXiv:2403.11522.
- Sean Mondesire, Emmanuel Nsiye, Bulent Soykan, and Glenn Martin. 2025. [Automating hpc software compilation, deployment, and error resolution through an llm-based multi-agent system](#). In *Practice and Experience in Advanced Research Computing 2025: The Power of Collaboration*, pages 1–8.
- Idan Mosseri, Lee or Alon, Re’em Harel, and Gal Oren. 2020. [Compar: Optimized multi-compiler for automatic openmp s2s parallelization](#). In *International Workshop on OpenMP*, pages 247–262. Springer International Publishing.
- Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. 2024a. [Can large language models write parallel code?](#)
- Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2024b. [HPC-coder: Modeling parallel programs using large language models](#). *Preprint*, arXiv:2306.17281.
- Yun Peng, Akhilesh Deepak Gotmare, Michael Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. [Perfcodegen: Improving performance of LLM generated code with execution feedback](#). *Preprint*, arXiv:2412.03578.
- Nadav Schneider, Niranjana Hasabnis, Vy A Vo, Tal Kadosh, Neva Krien, Mihai Capota, Guy Tamir, Theodore L Willke, Nesreen Ahmed, Yuval Pinter, and 1 others. 2024. [Mpirigen: Mpi code generation through domain-specific language models](#). In *Proceedings of the 2024 Workshop on AI For Systems*, pages 1–6.
- Nadav Schneider, Tal Kadosh, Niranjana Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. [Mpi-ricial: Data-driven mpi distributed parallelism assistance with transformers](#). In *Proceedings of the SC’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 2–10.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. [Reflexion: Language agents with verbal reinforcement learning](#). *Preprint*, arXiv:2303.11366.
- Shivam Singhal, Eran Malach, Tomaso Poggio, and Tomer Galanti. 2025. [LLM-ERM: Sample-efficient program learning via LLM-guided search](#). *Preprint*, arXiv:2510.14331.
- TOP500. 2025. [TOP500 List - November 2025](#). <https://top500.org/lists/top500/list/2025/11/>. 66th TOP500 list published Nov. 18, 2025 (St. Louis, MO). Accessed 2026-01-04.
- Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F. Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S. Vetter. 2023. [Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation](#). *Preprint*, arXiv:2309.07103.
- Pedro Valero Lara, Aaron Young, Jeffrey S Vetter, Zheming Jin, Swaroop Pophale, Mohammad Alaul Haque Monil, Keita Teranishi, and William F Godoy. 2025. [Chathpc: Building the foundations for a productive and trustworthy ai-assisted hpc ecosystem](#). In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 458–474.
- Bhanu Prakash Vangala, Ali Adibifar, Tanu Malik, and Ashish Gehani. 2025. [Ai-generated code is not reproducible \(yet\): An empirical study of dependency gaps in llm-based coding agents](#). *arXiv preprint arXiv:2512.22387*.
- Xingfu Wu, Praveen Paramasivam, and Valerie Taylor. 2023. [Autotuning apache tvn-based scientific applications using bayesian optimization](#). *Preprint*, arXiv:2309.07235.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. [Swe-agent: Agent-computer interfaces enable automated software engineering](#). *Advances in Neural Information Processing Systems*, 37:50528–50652.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. [Tree of thoughts: Deliberate problem solving with large language models](#). *Preprint*, arXiv:2305.10601.
- Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. 2023. [Self-taught optimizer \(STOP\): Recursively self-improving code generation](#). Microsoft Research.
- Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2024. [Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x](#). *Preprint*, arXiv:2303.17568.

A Pipeline Stage Restructuring for NAS and Rodinia

A.1 Rationale

When moving from HeCBench micro-kernels to larger NAS and Rodinia benchmarks, we redesigned the optimization workflow to better reflect full-application performance drivers. Micro-kernels typically expose a small number of localized hotspots, making a sequence of narrowly-scoped optimization stages effective. In contrast, NAS and Rodinia kernels frequently require holistic reasoning over loop structure, data lifetime, launch overhead, and correctness constraints at scale. Early prototypes decomposed optimization into multiple narrowly scoped stages (e.g., separate concurrency and memory/locality passes). The current pipeline consolidates these into two optimization steps that are applied across suites: (i) GPU offload with an explicit `data_plan.md` and baseline capture, and (ii) profiling-driven performance tuning with an explicit `optimization_plan.md`. In both cases, each stage is augmented with (i) profiler output from the prior stage, (ii) a short summary of previously applied actions, and (iii) system details (GPU/CPU/memory), enabling profiling-driven iterative refinement under a fixed hardware/software stack.

A.2 HeCBench Workflow

For HeCBench, PARACODEX uses a four-stage pipeline developed during our initial experimentation phase. The workflow consists of: (i) analysis, (ii) initial GPU offload with basic correctness validation, (iii) concurrency tuning (e.g., collapse directives, thread mapping), and (iv) memory/locality optimization (e.g., data movement, transfer reduction). Unlike the consolidated three-stage workflow later developed for NAS and Rodinia, this approach used shorter, less comprehensive prompts for each stage and decomposed optimization into separate, narrower passes. While this resulted in higher token costs due to the additional stage and context accumulation, it reflected our earlier development methodology and was retained for HeCBench to maintain consistency with completed experiments. The micro-kernel nature of HeCBench benchmarks — often dominated by a single hot loop — made the staged decomposition effective despite the increased overhead.

A.3 NAS/Rodinia Workflow: Consolidated Three-Stage Pipeline

For NAS and Rodinia, PARACODEX uses a refined three-stage pipeline that consolidates optimization into broader, more comprehensive stages. This redesign was motivated by the characteristics of full-application benchmarks, which require holistic reasoning over loop structure, data lifetime, and launch overhead rather than isolated passes. The three stages are: (i) analysis with loop taxonomy, (ii) GPU offload with explicit data-planning artifact (`data_plan.md`), and (iii) profiling-driven tuning with optimization planning artifact (`optimization_plan.md`). Unlike the four-stage HeCBench pipeline, this workflow uses longer, more detailed prompts that integrate concurrency and memory optimizations into a single profiling-driven refinement stage, reducing token costs while improving reasoning depth. Each prompt is parameterized with profiler output, system metadata, and summaries of prior actions, enabling measurement-driven iteration under a fixed hardware/software stack.

Stage 1 (Analysis Phase): Loop Classification and Data Characterization. The analysis phase enumerates and prioritizes loops across the benchmark source files, emphasizing code executed in the main timed region. Each loop is classified using a loop taxonomy that encodes parallelization constraints (e.g., dense, sparse/CSR, multi-stage FFT/butterfly, multigrid, histogram/indirect writes, recurrence, reductions, and stencil patterns). The phase also records loop nesting, dependency structure (reductions, stage dependencies, loop-carried recurrences), and data properties (array roles, access patterns, globals, scratch usage), and flags hazards such as atomics, variable bounds, or insufficient iteration counts. This phase produces a structured `analysis.md` artifact and copies source files into the kernel working directory without semantic modifications.

Stage 2 (GPU Offload + Data Strategy): Correctness with an Explicit Data Plan. Instead of a minimal correctness-first offload step, this stage couples correctness with explicit data-lifetime reasoning. The workflow (i) captures a baseline output for subsequent verification, (ii) selects a data-management strategy using a rule-based decision process (e.g., target data regions, asynchronous overlap, or global device-

state allocation via `omp_target_alloc` and `is_device_ptr`), and (iii) requires the pipeline to author a `data_plan.md` artifact *before* modifying code. The data plan enumerates arrays used in the timed region, identifies functions that must execute on the device, specifies host-to-device and device-to-host transfer timing and expected volumes, and includes strategy-specific correctness checks to prevent common mapping errors. The implementation then follows this plan to offload the required loops/functions while preserving benchmark semantics and passing the golden-serial correctness check.

Stage 3 (Performance Tuning): Profile-Guided Plan with Early Exit. This stage consolidates concurrency tuning, memory/locality tuning, and launch-overhead reduction into a single profiling-driven refinement stage. It first re-verifies correctness against the recorded baseline output. It then reads profiling summaries (kernel time, API time, and transfer time) and records key metrics (dominant kernels, launch counts, and transfer volumes). Before applying changes, the pipeline writes an `optimization_plan.md` artifact that prioritizes bottlenecks and proposes specific actions (e.g., hoisting data regions, eliminating redundant transfers, inlining helper functions to reduce launch overhead, loop fusion where safe, increasing collapse depth, introducing reductions for sparse inner loops only when beneficial, enforcing serial stage loops for multi-stage algorithms, and micro-optimizations such as `const/restrict` and caching locals). The stage supports early termination: if the current runtime is within a small tolerance of the estimated optimum inferred from profiling, the pipeline records metrics and restricts changes to low-risk micro-optimizations.

A.4 Summary of Differences

PARACODEX employs two distinct workflows tailored to benchmark characteristics and development timeline. The *four-stage HeCBench pipeline* reflects our initial experimental design with focused, shorter prompts decomposing optimization into separate concurrency and memory passes, resulting in higher token costs. The *three-stage NAS/Rodinia pipeline* represents an improved, consolidated approach that integrates optimization into comprehensive stages with explicit artifact-driven planning (`data_plan.md`, `optimization_plan.md`). This refinement

reduces token overhead while substantially improving reasoning depth and optimization effectiveness for full-application benchmarks. Both workflows share the core philosophy of correctness gating and profiler-driven refinement, and critically, *the correctness gate mechanism itself remained unchanged across both approaches* — all implementations are validated against golden serial output using the same Makefile-based harness and supervisor-driven repair loop. The three-stage pipeline is strictly superior in terms of cost-effectiveness and reasoning quality; the four-stage approach was retained for HeCBench solely to maintain consistency with completed experiments. Across suites, bottleneck profiles differ: HeCBench micro-kernels tend to be kernel-bound with localized hotspots, whereas NAS/Rodinia benchmarks frequently expose transfer overhead, launch costs, and complex data lifetime patterns that benefit from the holistic data-management planning enabled by the improved workflow.

B Analysis, Data Transfer, and Optimization Mechanics

PARACODEX operationalizes (i) loop-level analysis, (ii) data-movement planning and enforcement, and (iii) profiling-driven optimization during serial-to-OpenMP target-offload translation through three prompt templates: an *analysis* prompt used prior to translation, an *offload + data-strategy* prompt that implements a correct GPU version under an explicit data plan, and a *performance tuning* prompt that refines the implementation using profiling feedback. Throughout, prompts are parameterized using runtime template variables (e.g., `{kernel_dir}`, `{file_listing}`, and build/run commands) filled by the pipeline scripts.

B.1 Loop Analysis and Offload Target Identification

The analysis phase produces a structured `analysis.md` artifact while keeping the original source files unmodified. It is designed to (1) identify loops in the main timed region, (2) characterize dependence structure to determine offload feasibility, and (3) surface hazards (reductions, atomics, stage dependencies) that must be handled explicitly during translation.

Loop discovery and prioritization. The prompt enumerates loop constructs using lightweight source inspection (e.g., searching for `for`, `while`,

and main compute-loop patterns). Loops are prioritized by their position in the call path: loops executed every iteration of the main compute loop are treated as *CRITICAL/IMPORTANT*, while one-time setup loops are treated as *SECONDARY/AVOID*.

Priority classification by estimated work. For each loop, the prompt applies a coarse work model (iterations \times operations per iteration) and assigns one of four priority levels: *CRITICAL* (dominant or per-iteration $O(N)$ work), *IMPORTANT*, *SECONDARY*, or *AVOID* (setup/IO or trivially small loops). The primary objective is to focus subsequent offload effort on the loops that plausibly dominate runtime.

Loop-type taxonomy and dependence characterization. Each loop is assigned a type from a small taxonomy that encodes parallelization constraints:

- **Type A (Dense):** constant bounds, data-parallel structure.
- **Type B (Sparse/CSR):** inner bound depends on outer index; typically parallelize outer loop, with optional inner parallelism.
- **Type C1 (Multi-stage/Iterative):** stage-dependent computations (e.g., butterfly-like patterns); outer loops parallel, stage loop serial.
- **Type C2 (Multigrid/Hierarchical):** level-wise traversal with dependent stages; outer parallelism with stage ordering preserved.
- **Type D (Histogram/Indirect writes):** parallelizable with atomic updates (or structured privatization + merge).
- **Type E (Recurrence):** loop-carried dependencies or block-level synchronization patterns (e.g., `__syncthreads()` in CUDA); not offload-parallelizable without algorithmic restructuring.
- **Type F (Reduction):** scalar accumulation; parallelizable via reductions.
- **Type G (Stencil):** neighbor access; parallelizable with careful indexing.

A special case is handled explicitly: when an *outer* loop iterates over independent samples while an *inner* loop contains sequential RNG/state updates, the analysis marks the outer loop as parallelizable with per-thread RNG replication and marks the inner RNG loop as sequential within each thread.

Data analysis and hazard flags. In addition to loop structure, the analysis records data properties needed for correct mapping: array shapes (flat

vs. pointer-based), allocation style (static vs. dynamic), accessed struct members, and global variables used in the timed region. It flags hazards that must be handled during translation, including variable bounds, required reductions, atomic updates, stage dependencies, RNG usage in timed regions, and small trip counts.

B.2 Data Movement and Transfer Strategy

Correct and efficient target offload depends strongly on data-lifetime decisions. The first optimization step therefore requires the pipeline to select a data strategy and to produce a `data_plan.md` *before* implementing pragmas. This plan serves as an executable specification of which data reside on the device, when transfers occur, and which functions must execute on the device to avoid implicit host-device thrashing.

Strategy selection rules. The prompt selects one of three strategies using the loop taxonomy and program structure:

1. **Strategy A (Scoped target data regions):** use `target` data with explicit `map(to|from|tofrom|alloc)` clauses; default for most dense/stencil/reduction kernels and multigrid-like cases.
2. **Strategy B (Asynchronous overlap):** use `nowait` and `depend` to overlap independent transfers and kernels when the computation permits pipelining.
3. **Strategy C (Global device state):** allocate persistent device arrays with `omp_target_alloc` and pass them via `is_device_ptr` to eliminate repeated mapping in iterative solvers and multi-stage kernels.

The `data_plan.md` artifact. The data plan enumerates all arrays used in the timed region and classifies them as *working*, *scratch*, *const*, or *index*. It also lists functions executed in the timed region and assigns each to host or device execution. The plan explicitly specifies (i) one-time device allocations, (ii) host-to-device transfers (timing and volume), (iii) device-to-host transfers (timing and volume), and (iv) whether any transfers occur inside the iteration loop. The plan includes strategy-specific checklists intended to prevent common errors, such as: missing device versions of helper functions (triggering implicit copies), scratch buffers remaining on the host when

using persistent device pointers, or accidental reintroduction of `map` clauses into a Strategy C hot path.

Transfer-volume sanity checks. The plan records an *expected transfer volume* for the whole execution and marks large deviations as a red flag. Concretely, if measured transfers exceed the plan by more than a small constant factor (e.g., $> 2\times$), the pipeline treats this as evidence of incorrect data scoping (e.g., missing offloaded helpers causing repeated transfers) and prioritizes correcting data lifetime before applying kernel-level tuning.

B.3 Correctness Gating and Baseline Equivalence

Each kernel translation is anchored to a reference baseline captured before modification. The first optimization step requires recording baseline output and verifying the translated output against it (e.g., via textual diff on the benchmark’s verification markers). Only configurations that satisfy the golden-serial correctness check are retained for performance reporting. This gating is enforced before profiling-driven tuning, ensuring that subsequent optimizations operate on a semantically valid candidate.

When validation fails, the system invokes a supervisor agent that: (i) instruments code with `gate.h` macros to trace execution state, (ii) diagnoses discrepancies hierarchically (checking host-device memory consistency then kernel logic), and (iii) applies minimal repairs while preserving GPU offload. The supervisor iterates until `make check-correctness` passes. The prompt forbids CPU-only fallback and performance changes during this phase — its sole objective is numerical correctness.

B.4 Profiling-Driven Optimization and Action Library

The second optimization step performs profiling-driven refinement while holding the selected data strategy fixed. The prompt explicitly forbids changing the data strategy at this stage to avoid confounding performance gains from data-lifetime restructuring with kernel-level optimizations.

Early exit criterion. The pipeline exits early if the current runtime is within a small tolerance (5%) of the profiler-implied optimum (e.g., kernel-time dominance with low transfer overhead), or reverts any proposed change that regresses performance

by more than 10%. In either case, further modifications are limited to low-risk micro-optimizations (such as `const`, `restrict`, and caching locals). This reduces unnecessary refactoring when the implementation is already near a profiler-implied ceiling.

The `optimization_plan.md` artifact. Before applying changes, the pipeline produces an `optimization_plan.md` report containing: runtime, dominant kernel(s), GPU-time breakdown, transfer fraction and volume, and kernel launch counts. The plan also identifies candidate loop fusions (producer–consumer or adjacent loops with identical bounds) and characterizes iteration structure (e.g., iterative solver loops with repeated SpMV/update patterns).

Prioritized bottleneck checklist. The tuning prompt encodes a priority-ordered checklist of bottlenecks and corresponding remedies. Table 2 summarizes the pipeline artifacts and enforcement points:

- **Data-management issues:** hoist data regions, move scratch to device allocations, ensure all timed-region helpers run on device.
- **Launch overhead:** inline helper functions called inside iteration loops; fuse adjacent loops with identical bounds where safe.
- **Hot-kernel inefficiency:** adjust parallel decomposition (e.g., `collapse`), add `simd` to innermost loops, and cache index/array values to reduce redundant loads.
- **Sparse inner-loop decision:** keep CSR inner loops serial when average nonzeros per row is small; introduce inner-loop parallelism with reduction only when nonzeros are sufficiently large to amortize overhead.
- **Stage-dependent algorithms (Type C):** enforce serial stage loops; parallelize only outer dimensions to avoid barrier overhead and correctness failures.
- **Over-parallelization:** remove inner parallelism when outer-loop parallelism already saturates the GPU, as indicated by profiling and problem geometry.

Summary and provenance of changes. Finally, the prompt requires a structured end-of-step summary that records baseline and final metrics, enumerates applied optimizations (including reverted changes that regress performance), and states the primary insight and remaining bottlenecks. This re-

Phase	Key Outputs and Enforcement
Analysis	<code>analysis.md</code> : loop taxonomy, priorities, dependencies, data hazards; source copied unmodified.
Offload + Data Plan	<code>data_plan.md</code> : array inventory, function placement, explicit transfers, expected transfer volume; implementation must follow plan; baseline captured and correctness verified.
Tuning	<code>optimization_plan.md</code> : profiler-driven diagnosis, prioritized actions, early-exit rule; data strategy fixed; final summary with applied/reverted changes.

Table 2: Artifacts and enforcement points used by PARACODEX to structure analysis, data movement, and profiling-driven optimization.

porting discipline supports reproducibility and provides provenance for performance improvements reported in the evaluation.

B.5 Running Example: NAS CG Conjugate Gradient Solver Workflow

To make the pipeline concrete, consider the *CG* (Conjugate Gradient) kernel from NAS Parallel Benchmarks Class C, a sparse linear solver using iterative conjugate gradient with sparse matrix-vector multiplication (SpMV) in CSR format.

(i) Hotspot Summary (`analysis.md`). The analysis phase identifies the main benchmark loop (15 iterations, each invoking `conj_grad` with 25 internal `cg_it` iterations) and classifies nested loops:

- **Type E (Sequential):** outer benchmark iteration (`it = 1..NITER`) and inner `cg_it` loop (25 iterations) with loop-carried dependencies on `rho/beta` – must execute serially.
- **Type B (Sparse SpMV):** two SpMV kernels computing $q = A * p$ and $r = A * z$ with irregular CSR indexing via `rowstr/colidx` – data-parallel across rows, *CRITICAL* priority.
- **Type F (Reductions):** dot products (`norm_temp1/norm_temp2`, `rho`, `d`) and final residual norm – global reductions, *CRITICAL* priority.
- **Type A (Dense SAXPY):** vector updates (`z/r/p` axpy operations) – embarrassingly parallel, memory-bound.

Data: CSR matrix (`a[NZ]`, `colidx[NZ]`, `rowstr[NA+1]`, ~461MB total) plus five working vectors (`x`, `z`, `p`, `q`, `r`, each `NA+2` doubles).

Hazards: sequential `cg_it` iterations prevent outer parallelism; irregular gather pattern in SpMV limits inner parallelism.

(ii) Data Plan (`data_plan.md`). Strategy A (persistent target data): establish device residency before benchmark loop with `#pragma omp target enter data map(to: a[0:NZ], colidx[0:NZ], rowstr[0:NA+1]) and map(alloc: x[0:NA+2], z[0:NA+2], p[0:NA+2], q[0:NA+2], r[0:NA+2])`. Working vectors are initialized on-device to avoid host-device transfers. Expected transfers: 461MB H→D (CSR data) at entry, negligible D→H (scalar reduction results only), zero array transfers during 15×25 iteration loops. Correctness check: passes validation with `VERIFICATION SUCCESSFUL` after initial offload.

(iii) Optimization + Gate Result (`optimization_plan.md`). Profiling shows runtime dominated by 9,883 kernel launches (400 SpMV passes plus separate reduction/update kernels) with 94% GPU time in `conj_grad` and 91.7% API overhead from launch/synchronization costs. Bottleneck: repeated small kernels for norm reductions and residual computation inflate launch overhead; each operation spawns separate device kernels. Optimization: (i) fuse dual norm reductions (`norm_temp1/norm_temp2`) into single kernel, (ii) combine final SpMV and residual norm loop to reuse `rowstr/colidx` loads and eliminate one kernel per `conj_grad` call, (iii) cache intermediate scalars in registers to avoid redundant global memory access. Result: kernel launches reduced by ~25%, runtime improved to 2.04s (estimated ~20% speedup over baseline). Correctness gate: output validates against serial reference; optimization retained.

B.6 Baseline Prompt Structure

For comparison, the baseline system uses a single unstructured prompt that combines all objectives into one request. The baseline prompt template is:

Your Task:

1. Translate the code to an OpenMP GPU-offloaded version.
2. Apply GPU offloading pragmas as needed.
3. Optimize the code for performance while preserving functionality.
4. Ensure the code compiles and runs.
5. Deliver the modified code to `{kernel_dir}`.

Deliverable: The complete, modified source code in `{kernel_dir}`

This single-pass approach combines analysis, translation, and optimization into one undifferentiated request, contrasting with PARACODEX’s staged workflow that emits intermediate artifacts (`analysis.md`, `data_plan.md`, `optimization_plan.md`) and enforces correctness gates between stages. The baseline agent has access to the same tools (compiler, profiler, test harness) as PARACODEX but receives no explicit guidance on when or how to use them. In practice, the baseline agent may compile and test the code, but it does not systematically analyze loops, select data strategies, or iteratively optimize based on profiling feedback, as these steps are not prompted.

C From Serial→OpenMP to CUDA→OpenMP Translation

Our system originally targeted CPU programs and introduced GPU acceleration by automatically translating serial code to OpenMP target offload. This required analysis of loop nests, classification of parallelization patterns, and the construction of an explicit device-residency and data-movement plan.

We later extended this framework to support migration from CUDA code to OpenMP target offload. Although the overall workflow remained structurally similar, the design goals and constraints changed substantially. This appendix summarizes the key differences.

C.1 Change in Problem Setting

In the Serial→OpenMP case, the input code is CPU-bound and parallelism must be *introduced*. In contrast, CUDA code is already GPU-parallel. The migration task therefore becomes:

1. preserving the program’s existing GPU execution semantics,
2. translating CUDA runtime constructs into OpenMP equivalents, and
3. avoiding accidental reintroduction of host-device transfers or loss of data residency.

Thus the analysis stage was extended to identify not only computational loops, but also CUDA kernels, launch sites, and device-side execution structure.

C.2 Extended Analysis for CUDA Kernels

The original analysis focused on CPU loops and their computational cost. For CUDA input, the analysis phase additionally:

- detects `__global__` kernels and host launch sites,
- reconstructs grid and block structure,
- identifies grid-stride loop patterns,
- analyzes `atomicAdd`, shared memory usage, and `__syncthreads()`,
- estimates total device work as $grid \times block \times iterations$.

This allows the system to reason about CUDA execution semantics that must be preserved under OpenMP.

C.3 Data Residency and Memory Model Mapping

In Serial→OpenMP translation, device-residence is introduced incrementally and only when beneficial. However, CUDA programs already assume GPU-resident data and explicit control over allocation and transfer (`cudaMalloc`, `cudaMemcpy`, `cudaFree`).

The migration pipeline therefore first reconstructs the CUDA memory model and maps it to OpenMP constructs.

A central design goal is to avoid unintentionally moving data back to the host during migration.

C.4 Execution Model and Synchronization Differences

CUDA exposes block-local shared memory and synchronization via `__shared__` and `__syncthreads()`. OpenMP target offload does not provide an exact analogue. The migration workflow therefore performs:

- conversion of shared memory buffers to thread-private or local arrays,
- replacement of `atomicAdd` with OpenMP atomics or reductions,
- splitting or restructuring kernels that rely on `__syncthreads()`.

This represents one of the major conceptual extensions relative to the serial workflow.

C.5 Kernel Body and Index Mapping

CUDA kernels are converted into device functions invoked under `target teams loop`. Thread indexing logic involving `threadIdx`, `blockIdx`, and `gridDim` is replaced by `loop`

induction variables. Grid-stride loops become conventional bounded loops.

All CUDA API calls and CUDA-specific syntax are eliminated to ensure pure OpenMP execution.

C.6 Correctness and Baseline Outputs

Both workflows enforce diff-based correctness. For CUDA migration, the system first records the original CUDA program output. The OpenMP-offloaded version must match this baseline, preventing partial-migration or semantic drift.

C.7 Performance Considerations

The CUDA→OpenMP tuning stage introduces migration-specific bottlenecks, including:

- unintended data transfers after migration,
- excessive kernel launch counts after barrier removal,
- loss of original CUDA grid dimensionality, and
- reduced locality after removing shared memory.

Optimizations therefore focus on restoring CUDA-like execution patterns where possible (e.g., persistent device allocations and loop fusion), while remaining within the OpenMP execution model.

C.8 Summary of Conceptual Shift

In summary, the Serial→OpenMP pipeline teaches the system how to *introduce* GPU execution. The CUDA→OpenMP pipeline teaches it how to *preserve* an existing GPU execution model in a different runtime environment. This required extending the analysis phase, formalizing CUDA-to-OMP semantics, and preventing regression in device residency or synchronization behavior.

D Reproducibility and Experimental Details

D.1 Technical Details and Token Counts

The primary model used for all kernels is `gpt-5.1-codex-mini`. For the NAS FT kernel, we escalated to `gpt-5.1-codex-max` for both PARACODEX and baseline to maintain fairness. Token usage (NAS benchmarks, representative of current pipeline): PARACODEX averages 837,701 tokens per kernel (cg: 776,036; ep: 967,774; ft: 1,031,208; mg: 575,786), while the baseline averages 755,417 tokens (cg: 286,118; ep: 1,379,682; ft: 880,104; mg: 475,765). The $1.11\times$ ratio reflects PARACODEX’s multi-stage workflow

(analysis → offload+data plan → optimization) compared to the baseline’s single-pass translation.

Why token usage is high: Both PARACODEX and the baseline operate Codex CLI in non-interactive automation mode, which is designed for scripting and CI/CD workflows where the agent executes all required actions in a single session without human interaction. This mode consumes significantly more tokens than interactive development because: (i) the full codebase context, build environment, and tool outputs must be provided in each agent turn, (ii) the agent maintains complete execution traces for reproducibility, and (iii) compilation errors, test outputs, and profiler reports are included verbatim to enable deterministic diagnosis. This design prioritizes *reproducibility and determinism* – ensuring that each parallelization attempt is fully auditable and replayable – over raw token efficiency. Crucially, the baseline exhibits comparable token consumption (755k tokens/kernel) despite its simpler single-pass structure, confirming that the high token count is primarily an artifact of the non-interactive automation mode rather than PARACODEX’s staged workflow. Full prompt templates are available in our repository.

D.2 GPU-Offload Bypass: Detailed Analysis

In the two HeCBench bypass cases (*pathfinder*, *particlefilter*), PARACODEX produced implementations that compile with `OpenMP target` directives and pass correctness checks, yet effectively bypass GPU computation by executing the primary computation on the host CPU. The zero-shot Codex baseline exhibits the same behavior on *pathfinder*.

Bypass Definition and Detection. We classify a run as a *bypass* when inspection of the generated code together with the corresponding profiler report indicates that the dominant computational loop remains on the host, while the device executes only a minimal or identity kernel that does not account for the kernel’s substantive work. Because this behavior contradicts the stated objective of GPU parallelization, including such runs would misrepresent success and can inflate performance comparisons without reflecting genuine offload. We therefore exclude bypass kernels from GPU-time statistics, while reporting their occurrence explicitly.

Bypass Occurrence. Bypass was observed in 2/23 HeCBench kernels for PARACODEX. No bypass cases were observed in Rodinia, NAS, or ParEval under our evaluation protocol.

Root Cause Analysis. We analyzed the two bypass cases by examining agent transcripts and profiling logs. In both cases, the agent generated code that compiles and passes correctness checks but executes the primary computation on the host CPU with minimal/identity GPU kernels. Our reading of the transcripts is that the agent judged transfer overheads to dominate computation and kept the work on the host despite emitting syntactically valid offload constructs, violating the GPU-offload intent.

Mitigation Strategies. Looking forward, a harness could mitigate bypass by enforcing evidence of device-side work, for example by requiring non-trivial kernel-launch activity, checking device-memory allocation sizes ($> 1\text{MB}$), or thresholding the fraction of runtime spent in GPU kernels as measured by profiling (e.g., requiring $> 50\%$ of time in device kernels). Preliminary testing with minimum device memory allocation and kernel launch count (> 10) constraints recovers GPU offload for *pathfinder* but requires algorithmic restructuring for *particlefilter*.

E Detailed Benchmark Suite Results

Per-suite analysis of PARACODEX’s performance across ParEval, HeCBench, and Rodinia follows below.

E.1 ParEval: Robust CUDA→OpenMP Translation and Correctness

ParEval emphasizes translation fidelity for CUDA→OpenMP migration (App. C details the CUDA-specific workflow extensions). Each kernel is assessed under a fixed build environment (*code-only*) and in a setting where the agent must also construct the build flow (*overall*). As shown in Figure 2, PARACODEX attains perfect compilation and validation in the *code-only* regime (4/4 kernels at 100%), and remains strong in the more demanding *overall* regime: it compiles and validates *nanoXOR* and *microXOR* in all trials (1.0), achieves 0.8 on *XSBench*, and the remaining errors are concentrated in *microXORH* (0.4). The baseline exhibits consistently lower overall success, with the largest drop on *microXORH*. The heatmap further shows that non-agentic approaches struggle significantly, while top-down agentic methods improve but remain inconsistent. Overall, PARACODEX’s structured workflow — CUDA kernel analysis, memory model reconstruction, and

execution-semantics preservation — generalizes effectively to cross-API translation.

E.2 HeCBench: Broad Performance Gains Across Diverse Workloads

HeCBench spans regular stencil and dynamic-programming kernels through irregular memory- and graph-oriented workloads. We evaluate 23 benchmarks (Figure 3); 2 exhibit GPU-offload bypass and are excluded from performance metrics. We report GPU time for the remaining 21 valid kernels. Parallel-friendly kernels exhibit large gains. For *jacobi* ($\sim 177\times$ on A100, $\sim 467\times$ on RTX 4060), this stems from reducing redundant transfers via persistent device data regions. Bandwidth-limited kernels improve modestly (e.g., *geodesic*: $1.00\times$, *pool*: $1.01\times$). Across the 21 valid kernels, PARACODEX achieves $3.1\times$ (A100) and $3.6\times$ (RTX 4060) geometric-mean GPU-time speedup (median $1.34\times$ on A100). The baseline demonstrates lower robustness, failing on 2 benchmarks (*keogh*, *mixbench*) on both GPUs, achieving 90% success (19/21) with geometric-mean speedups of $2.0\times$ (A100) and $2.4\times$ (RTX 4060) and a median of $1.39\times$ (A100). PARACODEX achieves 100% compilation success (21/21) and either matches or improves runtime on 19/21 (90%) kernels.

E.3 Rodinia: Translation Reliability and Systematic Performance Gains

PARACODEX successfully translates all Rodinia kernels and attains geometric-mean GPU-time speedups of $1.46\times$ (A100) and $1.08\times$ (RTX 4060) (Figure 4-left). The median speedup on A100 is $1.40\times$. Because many Rodinia programs contain correct but non-fully-optimized OpenMP code, PARACODEX frequently improves performance by restructuring parallel regions, reducing synchronization, and lowering data-movement overhead. The baseline also produces running implementations for all kernels but achieves only a $0.30\times$ geometric-mean speedup on A100 (median $0.96\times$).

E.4 NAS: Handling Complex, Expert-Level Codebases

Figure 4 summarizes NAS results (right). PARACODEX achieves a $1.00\times$ median and geometric-mean of $2.95\times$ (A100) and $1.22\times$ (RTX 4060) relative to the highly optimized reference implementations. EP shows the largest headroom ($29.7\times$

on A100), while MG is sensitive to end-to-end transfer overhead and FT matches the reference. These results indicate PARACODEX can match expert references on several kernels while remaining competitive on highly tuned codes. The baseline achieves a $1.00\times$ median and $1.03\times$ geometric-mean GPU-time speedup and fails to compile FT. App. G shows a detailed comparison of baseline vs PARACODEX outputs on NAS MG, where PARACODEX achieves $1.57\times$ better GPU time through kernel fusion driven by profiling feedback.

F Comparison with Prior Work

F.1 UniPar

Figure 7 illustrates the substantial gains achieved by a continuously-iterating agent in the serial-to-OpenMP translation task, particularly when guided by an explicit execution plan. The figure reports the fraction of HeCBench OpenMP translations that pass validation across the 13 UniPar (Bitan et al., 2025) kernels that also appear in our evaluation set. These are kernels whose core computation resides within a single source file.

F.2 LASSI

Dearing et al. (2025) proposed a successful pipeline for a related—but distinct—task: translating CUDA to OpenMP. Their system achieved an 85% valid translation rate across ten HeCBench kernels when averaged across several base models. Because their work reports performance relative to the original reference implementation, we evaluate our approach using the same metric on this subset of kernels to enable approximate comparison, despite the differing translation objective.

Figure 8 reports the *Ratio* metric introduced by Dearing et al. (2025), defined as the reference execution time divided by the execution time of the translated kernel. We present the Ratio for kernels translated by the baseline Codex agent and by PARACODEX, alongside the average Ratio reported by LASSI across its models. Although LASSI evaluates on two A100 GPUs and PARACODEX runs on a single RTX 4060, reference times in each case are measured on the same hardware as the translated kernels, mitigating most hardware-driven bias.

It is important to emphasize that LASSI evaluates CUDA→OpenMP translation whereas PARACODEX performs serial→OpenMP translation; therefore, the comparison is not strictly direct. Nevertheless, the large disparities observed—such as

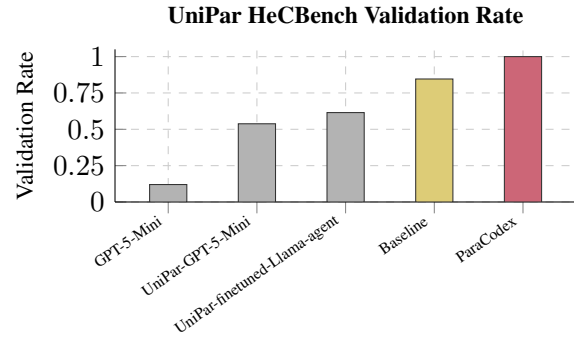


Figure 7: **Percentage of translated code passing validation across different UniPar models** Comparing the Codex baseline and PARACODEX against the UniPar methodology with a GPT-5-mini base model and the fine-tuned LLaMA model reported in prior work. The results highlight the substantial validation gains enabled by newer sophisticated agentic models.

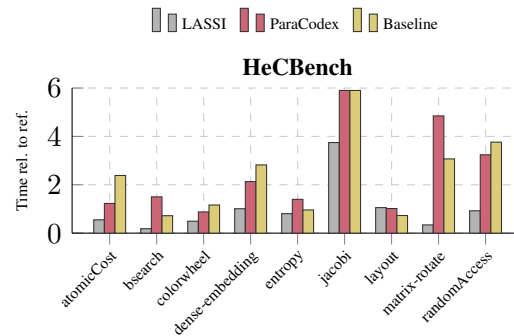


Figure 8: **Reference time / translated kernel run time.** ParaCodex and Baseline for Jacobi are clipped at 6.0 for visualization (actual values: 489.6 – PARACODEX and 326.1 – baseline).

the $489\times$ Ratio achieved by the *jacobi* kernel—together with the consistently higher mean Ratio, suggest that a carefully designed, agentic workflow such as PARACODEX can deliver substantial performance advantages over both out-of-the-box Codex and prior automated pipelines.

G Example Code Transformation: NAS MG vs Baseline

To illustrate why PARACODEX outperforms the zero-shot baseline, we compare their outputs on the NAS MG multigrid solver’s *resid* stencil kernel. The baseline produces a correct two-pass implementation with device-allocated temporaries, while PARACODEX’s staged workflow refines this pattern into a fused single-pass structure through profiling feedback.

G.1 Baseline Output (Zero-Shot Codex)

The baseline generates a conservative two-pass structure that separates neighbor computations from the residual update:

Structure: The baseline allocates two large temporary arrays (`u1` and `u2`, each containing 19.7M doubles, totaling 316MB) on the host using `malloc`, then maps them to the device using OpenMP’s `map(alloc:)` clause.

Pass 1 – Neighbor sum computation: A first OpenMP target kernel iterates over the 3D grid (excluding boundary points) and computes 4-neighbor sums along two different axes. For each grid point (i_3, i_2, i_1) , it reads four neighbors from the input array `ou` and writes the sums to the device-allocated temporary arrays `u1` and `u2`. This pass launches 170 times during the solve (once per `resid` call).

Pass 2 – Residual computation: A second OpenMP target kernel reads the precomputed sums from global memory temporaries `u1` and `u2`, along with the original array `ou` and the right-hand-side `ov`, and computes the final residual values into output array `orr`. This pass also launches 170 times.

Performance characteristics: This implementation incurs (1) host-side `malloc/free` overhead at each call, (2) 340 total kernel launches (2 per `resid` call \times 170 calls), (3) redundant global memory traffic (neighbor values written to device memory in pass 1, then read back in pass 2), and (4) implicit synchronization barrier between the two passes. GPU time: 7152 ms (Class C).

G.2 ParaCodex Output (Staged Workflow)

PARACODEX’s analysis stage identifies the stencil pattern and flags it as a candidate for fusion. The initial translation stage produces a correct implementation similar to the baseline. The optimization stage then profiles this code, observes high kernel launch counts (340 launches) and detects the memory traffic pattern (write-to-temps followed by read-from-temps), and applies kernel fusion.

Structure: The optimized code eliminates the `u1` and `u2` temporary arrays entirely. It uses a single OpenMP target kernel with `collapse(2)` to parallelize the outer two dimensions.

Fused pass – Register-based computation: Within the triply-nested loop, the code computes six register variables (`u1_c`, `u1_L`, `u1_R`, `u2_c`, `u2_L`, `u2_R`) that hold neighbor sums computed directly from the input array `ou`. Each

sum aggregates four neighbor values using array accesses with explicit index arithmetic (e.g., `I3D(i3, i2-1, i1)` for the neighbor below). Immediately after computing these register temporaries, the code uses them to update the residual array `orr` within the same loop iteration. The register values are never written to global memory.

Performance improvements: This transformation (1) eliminates 316MB device memory allocation and host-side `malloc` overhead, (2) reduces kernel launches from 340 to 170 (50% reduction), (3) removes the write-then-read global memory roundtrip for temporary values, and (4) eliminates the implicit barrier between passes. Neighbor values remain in registers throughout, improving cache utilization. **GPU time: 4569 ms, yielding a $1.57\times$ speedup over baseline (Class C).**

G.3 Summary

Together, the two outputs show how PARACODEX’s profiling-guided workflow systematically identifies and eliminates performance bottlenecks that single-shot generation misses. The baseline’s conservative strategy prioritizes correctness by explicitly materializing intermediate results, but this incurs substantial overhead. PARACODEX’s staged approach first establishes correctness, then uses profiling data to safely apply aggressive transformations (kernel fusion, register promotion, launch reduction) that recover 57% of the baseline’s GPU time while maintaining bit-identical output.

H Full Benchmark Results

We provide the complete benchmark results for all evaluated kernels across all benchmark suites and both hardware platforms. All values are reported as mean \pm standard deviation over 10 runs.

Benchmark	GPU	Reference (ms)	Baseline (ms)	PARACODEX (ms)	Para/Ref	Base/Ref
ace	RTX 4060	24975.47 ± 5017.30	7628.00 ± 1179.02	6762.16 ± 1483.32	3.69x	3.27x
	A100	2432.34 ± 244.43	2948.95 ± 470.37	2048.05 ± 36.68	1.19x	0.82x
atomicCost	RTX 4060	7339.33 ± 101.02	26334.07 ± 1811.09	9175.02 ± 986.64	0.80x	0.28x
	A100	1702.82 ± 28.76	8376.75 ± 17.35	1901.92 ± 25.39	0.90x	0.20x
bsearch	RTX 4060	16378.49 ± 132.87	16936.13 ± 309.68	18.97 ± 1.29	863.46x	0.97x
	A100	754.64 ± 11.76	839.78 ± 20.38	7.53 ± 0.84	100.28x	0.90x
colorwheel	RTX 4060	116.08 ± 12.96	5.30 ± 0.47	12.06 ± 0.95	9.63x	21.89x
	A100	26.20 ± 0.06	3.28 ± 0.43	8.67 ± 1.55	3.02x	7.99x
convolution1D	RTX 4060	7690.94 ± 5.80	7899.75 ± 4.11	8890.83 ± 9.06	0.87x	0.97x
	A100	2645.43 ± 14.63	2787.36 ± 8.95	3213.02 ± 12.07	0.82x	0.95x
dense-embedding	RTX 4060	2253.36 ± 405.50	710.69 ± 75.51	979.42 ± 116.23	2.30x	3.17x
	A100	6309.87 ± 1851.14	3365.64 ± 2000.29	5606.46 ± 348.73	1.13x	1.87x
entropy	RTX 4060	49.68 ± 4.78	26.33 ± 0.63	39.46 ± 9.48	1.26x	1.89x
	A100	19.84 ± 0.17	13.33 ± 0.14	15.68 ± 0.41	1.27x	1.49x
geodesic	RTX 4060	201.71 ± 7.63	207.04 ± 17.03	199.66 ± 7.85	1.01x	0.97x
	A100	93.73 ± 1.80	94.73 ± 2.14	93.45 ± 2.62	1.00x	0.99x
heat	RTX 4060	9880.02 ± 105.64	10017.75 ± 97.68	9857.24 ± 99.57	1.00x	0.99x
	A100	1722.27 ± 3.35	1847.46 ± 4.32	1729.60 ± 9.05	1.00x	0.93x
jacobi	RTX 4060	3713.60 ± 30.68	11.32 ± 0.56	7.95 ± 1.15	467.16x	328.01x
	A100	1444.95 ± 8.05	9.94 ± 0.23	8.17 ± 0.58	176.95x	145.42x
keogh	RTX 4060	13132.75 ± 204.18	N/A	508.80 ± 15.49	25.81x	N/A
	A100	4732.52 ± 16.72	N/A	212.54 ± 10.79	22.27x	N/A
layout	RTX 4060	770.77 ± 6.05	1123.06 ± 8.93	772.26 ± 8.40	1.00x	0.69x
	A100	500.48 ± 14.24	367.47 ± 6.74	500.50 ± 17.46	1.00x	1.36x
matrix-rotate	RTX 4060	463.76 ± 5.20	481.44 ± 1.88	451.29 ± 6.49	1.03x	0.96x
	A100	717.30 ± 9.47	200.93 ± 5.39	159.27 ± 8.37	4.50x	3.57x
michalewicz	RTX 4060	144.64 ± 2.60	77.14 ± 2.13	93.29 ± 2.39	1.55x	1.88x
	A100	79.53 ± 1.09	43.76 ± 1.47	48.47 ± 1.70	1.64x	1.82x
mixbench	RTX 4060	92161.62 ± 79.69	N/A	15736.51 ± 1299.34	5.86x	N/A
	A100	124244.45 ± 23.91	N/A	27237.87 ± 15.06	4.56x	N/A
pool	RTX 4060	182.06 ± 5.29	269.33 ± 24.57	173.02 ± 3.85	1.05x	0.68x
	A100	115.61 ± 4.41	143.67 ± 5.73	114.80 ± 1.76	1.01x	0.80x
randomAccess	RTX 4060	24191.31 ± 114.48	5007.51 ± 28.45	5088.59 ± 20.33	4.75x	4.83x
	A100	42734.80 ± 13.69	1188.52 ± 8.97	1370.02 ± 6.39	31.19x	35.96x
stddev	RTX 4060	1178.60 ± 26.32	816.89 ± 10.51	291.55 ± 13.41	4.04x	1.44x
	A100	766.35 ± 15.02	543.73 ± 13.16	201.90 ± 5.79	3.80x	1.41x
stencil1d	RTX 4060	1418.69 ± 24.81	626.29 ± 16.92	1423.01 ± 30.04	1.00x	2.27x
	A100	754.63 ± 161.51	501.15 ± 181.58	718.01 ± 64.28	1.05x	1.51x
vanGenuchten	RTX 4060	25394.62 ± 46.80	41714.44 ± 2814.44	15986.46 ± 27.71	1.59x	0.61x
	A100	738.73 ± 16.22	19193.48 ± 581.78	550.60 ± 13.13	1.34x	0.04x
winograd	RTX 4060	14.09 ± 0.01	0.17 ± 0.00	5.60 ± 0.00	2.52x	80.82x
	A100	9.35 ± 1.66	0.10 ± 0.00	4.67 ± 0.67	2.00x	92.28x

Table 3: Full results for the HECBench suite.

Benchmark	GPU	Reference (ms)	Baseline (ms)	PARACODEX (ms)	Para/Ref	Base/Ref
b+tree	RTX 4060	11921.55 ± 625.50	7446.99 ± 83.76	8246.20 ± 787.64	1.45x	1.60x
	A100	0.58 ± 0.02	0.39 ± 0.01	0.35 ± 0.02	1.65x	1.47x
backprop	RTX 4060	2288.81 ± 429.78	1773.31 ± 66.98	2056.08 ± 133.46	1.11x	1.29x
	A100	5356.22 ± 586.51	4813.65 ± 544.95	4670.80 ± 292.66	1.15x	1.11x
bfs	RTX 4060	96.60 ± 2.30	101.08 ± 1.39	100.80 ± 7.96	0.96x	0.96x
	A100	1825.96 ± 260.63	1667.75 ± 171.70	1669.62 ± 174.51	1.09x	1.09x
hotspot	RTX 4060	1320.05 ± 1.23	11793.29 ± 181.74	1761.88 ± 0.13	0.75x	0.11x
	A100	296.51 ± 12.88	10584.98 ± 789.00	104.55 ± 2.52	2.84x	0.03x
lud	RTX 4060	347.44 ± 1.27	10862.94 ± 743.30	253.53 ± 2.18	1.37x	0.03x
	A100	439.12 ± 12.50	27071.13 ± 698.61	257.92 ± 7.00	1.70x	0.02x
nw	RTX 4060	7096.05 ± 516.37	8201.18 ± 199.32	6927.99 ± 338.62	1.02x	0.87x
	A100	2156.66 ± 353.82	2598.07 ± 242.66	2236.57 ± 283.90	0.96x	0.83x

Table 4: Full results for the Rodinia suite.

Benchmark	GPU	Reference (ms)	Baseline (ms)	PARACODEX (ms)	Para/Ref	Base/Ref
cg	RTX 4060	4544.32 ± 100.84	4343.48 ± 102.00	4439.33 ± 179.54	1.02x	1.05x
	A100	25270.14 ± 87.73	25304.58 ± 48.05	25371.19 ± 33.62	1.00x	1.00x
ep	RTX 4060	5985.12 ± 19.47	5979.38 ± 11.72	3962.72 ± 0.59	1.51x	1.00x
	A100	39663.43 ± 19.82	24822.19 ± 13.94	1334.62 ± 1.19	29.72x	1.60x
ft	RTX 4060	123926.81 ± 2055.12	N/A	123525.64 ± 9686.44	1.00x	N/A
	A100	14808.81 ± 168.88	N/A	14834.23 ± 126.09	1.00x	N/A
mg	RTX 4060	3869.68 ± 23.61	3370.77 ± 15.35	2748.69 ± 5.24	1.41x	1.15x
	A100	16828.20 ± 140.30	24291.55 ± 3640.35	19324.83 ± 3115.36	0.87x	0.69x

Table 5: Full results for the NAS suite.