PortBench: Measuring the Gap Between Correctness and Speed in CPU-to-GPU Code Generation

Mingzhe Du [1 2] **Anh Tuan Luu** [2 3] **Dong Huang** [1] **See-Kiong Ng** [1]
{mingzhe,dhuang,seekiong}@nus.edu.sg, anhtuan.luu@ntu.edu.sg

Abstract

Correctness alone does not guarantee an efficient GPU implementation. We introduce PORTBENCH, an efficiency benchmark of 258 CPU-to-GPU porting tasks across 16 categories, with hidden-input checks and paired H100 timing against fixed direct GPU references. We evaluate seven model–agent configurations without target-GPU feedback during generation. The most accurate configuration achieves 96.5% correctness but exceeds the reference by 10× on only 38.4% of tasks; across all configurations, 111 tasks have a correct observed port but no tenfold improvement. Generation time and API-equivalent cost produce different Pareto frontiers, revealing tradeoffs hidden by performance rankings alone. Category-level analysis identifies gaps in simulation, physics, earth science, and data systems, providing concrete workloads for improving GPU efficiency.

1. Introduction

A program can produce the right answer on a GPU and still fail as a GPU implementation. Preserving a CPU program's behavior requires reasoning about dependencies, integer arithmetic, reductions, and boundary conditions; accelerating it requires deciding which work to parallelize, where to place data, and when to change the algorithm. Correctness–efficiency gaps are documented for CPU code generation (Du et al., 2024; Qing et al., 2025), and for GPU porting the consequence is larger: a valid port that serializes most work on the host delivers little of the benefit that motivated it. Evaluating only functional success conflates the two capabilities; evaluating speed only on successful submissions hides the cost of failures. Since coding agents are increasingly asked to migrate and accelerate existing code, with measured execution as the natural reward, the relevant outcome is joint: *a semantically valid program, a measured performance gain, and a stated generation budget*.

Existing benchmarks provide components of this evaluation. KernelBench measures correct and fast GPU kernels for PyTorch workloads (Ouyang et al., 2025); ParEval evaluates parallel code across execution models (Nichols et al., 2024); repository benchmarks test migration or performance engineering (Davis et al., 2025; Shetty et al., 2025; Ma et al., 2025; He et al., 2026). We study a complementary setting: translating a complete, self-contained CPU program into CUDA, with behavior fixed by source code and runtime compared to a fixed direct GPU port. It spans irregular, discrete, and numerical workloads, without repository navigation or build repair.

We introduce PORTBENCH, comprising 258 tasks across 108 domain labels and 16 reporting categories. Each task pairs an OpenMP C++ program with generated inputs, output comparisons, and a CUDA reference. Agents work in CPU sandboxes without a GPU or CUDA toolchain; a separate H100 verifier compiles and evaluates their final source. Instructions ask agents to defer CUDA compilation to the verifier, but no tool is disabled. The setting measures porting without provided target-hardware feedback, not the best performance attainable with iterative GPU profiling.

The central question is whether, once semantic migration usually succeeds, an agent can reliably produce *fast* GPU programs at reasonable generation cost, and which workloads still leave performance on the table. **RQ1:** How much observed correctness translates into acceleration over a direct port? **RQ2:** How do generation time and cost change the comparison? **RQ3:** Which conclusions survive reweighting, oracle-provenance, and audit checks, and where do gaps persist?

Across seven model–agent configurations and 1,806 planned outcomes, correctness and optimization separate sharply. Astra passes 96.5% of tasks yet exceeds the direct reference by 10× on only 38.4%; Opus passes 95.3% but reaches 51.9% tenfold coverage and the highest mean score. Across all seven, 111 tasks have a valid port but no tenfold improvement, most prevalent relative to category size in simu-

[1]National University of Singapore [2]Nanyang Technological University [3]CAIR, VinUniversity. Correspondence to: Mingzhe Du <mingzhe@nus.edu.sg>.

Preprint. November 2, 2026.

lation, physics, earth-science, and data-systems workloads. Opus extends the observed frontier at the highest time and cost, while Haiku is time-dominated but remains on the cost frontier. Our contributions are (i) a program-level CPU-to-GPU benchmark with a reproducible 16-category taxonomy, source-defined semantics, and hidden-input hardware evaluation; (ii) an evaluation separating correctness, raw speedup coverage, and a bounded optimization score, with the resources needed to achieve them; and (iii) an empirical study of the correctness–speed gap, resource-dependent frontiers, and the categories where the gap persists, with checks that delimit which comparisons the runs support.

2. Related Work

Efficiency beyond correctness. Mercury evaluates Python code against distributions of solution runtimes and combines correctness and efficiency in its Beyond metric (Du et al., 2024); EffiBench-X extends efficiency evaluation to six languages with human-expert baselines (Qing et al., 2025). Both document gaps between functional success and efficient execution on the same hardware. PORTBENCH studies the gap across hardware, with a complete CPU program as input, timed CUDA code as output, and a direct GPU port rather than a distribution of human solutions as the comparison point.

GPU and parallel code generation. KernelBench evaluates PyTorch workloads using correctness and speed; its fast_p metric motivates our speedup-coverage reporting (Ouyang et al., 2025). ParEval measures correctness and runtime for standalone parallel functions, including CUDA and HIP (Nichols et al., 2024), and ParEval-Repo studies translation across GPU programming models at repository scale (Davis et al., 2025). PORTBENCH sits between these scales: the unit is a complete executable with host work and transfers inside the timed interval, but the artifact boundary is one CUDA file, emphasizing semantic and algorithmic transfer, including irregular and integer-exact computations that operator-based kernel suites rarely contain.

Performance engineering with language models. SWE-Perf (He et al., 2026), GSO (Shetty et al., 2025), and SWE-fficiency (Ma et al., 2025) evaluate repository-level performance engineering, and AlgoTune studies numerical optimization with execution and profiling feedback (Press et al., 2025). These settings give the agent the target environment, so measured speed reflects iterative tuning as well as initial design; our agents see CPU source and public inputs, and GPU compilation and timing occur only after submission.

Learning to improve generated programs. Afterburner combines execution feedback with reinforcement learning for code efficiency (Du et al., 2025b), CodeScaler uses learned reward models for code generation (Zhu et al., 2026), and Ockhamareto applies Pareto-gated reinforcement learning to test-suite optimization (Huang et al., 2026a). A benchmark with measured, bounded rewards is a prerequisite for such methods on GPU porting; we neither train one nor treat learned rewards as substitutes for measured execution.

Oracle and benchmark reliability. Huang et al. (2026b) show that incorrect code can bias generated tests; Nexus refines synthesized oracles with execution feedback (Huang et al., 2025); Measuring the Checker audits GPU-benchmark oracles with mutation analysis (Du et al., 2026). Our hidden-input checks do not constitute such an adequacy audit. ULT studies test contamination in function testing (Huang et al., 2026c) and AntiLeakBench refreshes evaluations to reduce contamination (Wu et al., 2025); fresh runtime inputs alone do not establish that task semantics were unseen during training. CodeArena provides shared evaluation infrastructure with scores that adapt to participating models (Du et al., 2025a), whereas PORTBENCH uses fixed references and reports correctness, speed, and resources separately; its Pareto sets summarize evaluated configurations, not the best achievable frontier.

3. The PortBench Benchmark

3.1. Task Definition

A task specifies a CPU program f, an input generator G, an output comparison $\mathcal{C}$, and a fixed direct GPU reference g_{ref} (Figure 1). The agent receives the CPU source, a description of the computation and input ranges, and public examples, and submits a CUDA program g with the same command-line interface. On each hidden input x, the intended contract is $\mathcal{C}(f(x), g(x)) = 1$; testing approximates it on finitely many generated inputs and is not a proof of equivalence. The source is authoritative when prose and code disagree. Inputs contain parameters and a seed, bulk data are generated within the program, and outputs are compact summaries, selected values, or hashes, so fresh instances need no distributed answer set. The reference g_{ref} is a straightforward port written during construction; it anchors the speed comparison and is not an expert optimum.

Why complete programs? The output contract can depend on several interacting stages. The `astro-dedisp` task, for example, combines dispersion delays, channel flagging, and a boxcar search, and a port must preserve indexing and selection rules across the pipeline while exposing data reuse. Other tasks involve graph traversal, dynamic programming, modular arithmetic, or simulation. The unit of evaluation is the entire executable, including host work and transfers inside the timed interval, rather than an isolated kernel whose surrounding computation is supplied.

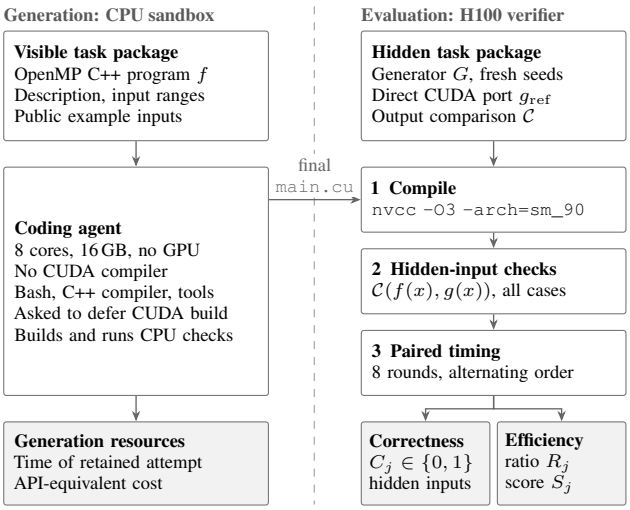


Figure 1. **Benchmark overview.** The agent works in a CPU sandbox and submits one CUDA source; a separate H100 verifier compiles it, checks outputs on hidden inputs, and times it against the direct reference. Each task yields a correctness outcome, an efficiency outcome, and the generation resources spent.

3.2. Composition and Construction

The suite contains 258 tasks with 108 domain labels; 219 use exact output comparison and 39 allow task-specific numerical tolerances. CPU programs have a median of 223 lines (range 70–725) and direct references a median of 311 (87–941); stored validation records contain 4–24 correctness cases per task (Appendix H). Construction used coding agents under a written authoring specification favoring domain-inspired computations with explicit deterministic rules, multicore CPU implementations, and a direct GPU port that leaves optimization opportunities, followed by automated validation and GPU checks; 63 candidates were retired as duplicates. An H100 acceptance sweep on 2026-10-07 recorded passes for all 258 packaged references. This is historical input-level validation, not a proof that every later package version is correct.

The tasks are constructed variants of computational patterns; they are not claimed to introduce new algorithms or to be absent from training data, and a partial construction audit identified weak variants close to known algorithms. Canary strings help identify copied task files but do not establish contamination resistance. Large reference-relative improvements may expose a weak reference as well as a strong candidate, so we report raw ratios and test the influence of the largest flagged result.

3.3. Task Categories

A fixed domain-to-category mapping assigns every task to exactly one of 16 reporting categories (Appendix Table 9); sizes range from 5 to 32 tasks, with signals and vision (32), data and systems (29), and earth and space (25) the largest. The taxonomy combines application areas (biology, chemistry) with computational families (graphs, dynamic programming); it is not a partition into disjoint algorithmic primitives, and reductions, sparse access, sequential dependencies, and integer arithmetic recur across categories. The mix extends evaluation beyond dense tensor kernels to stateful, irregular, and discrete computations, where the fastest GPU formulation often differs from the CPU loop structure. The categories localize the correctness–speed gap to workload families (Section 5.5) and support category-balanced averages; the mapping does not use measured scores.

3.4. Generation Environment

Each agent receives 8 CPU cores, 16 GB of memory, a C++ compiler, and public inputs; no GPU or CUDA compiler is provided. Instructions request deferring CUDA compilation to the verifier, while ordinary tools remain available, so an agent can build and run the CPU program, inspect inputs, and write host-side checks before committing to a CUDA design. Only the final `main.cu` is collected. The verifier compiles with `nvcc -O3 -arch=sm_90 -std=c++17`; CUDA runtime functions and header-only toolkit libraries, including CUB and Thrust, are allowed, externally linked numerical libraries are not. Scores therefore characterize what an agent produces from source understanding alone, without target-hardware feedback; iterative GPU profiling is a complementary setting.

4. Evaluation Protocol

4.1. Correctness and Timing

The verifier generates hidden correctness inputs and a timing input from a fresh seed. Exact mode compares lines after stripping trailing whitespace; numerical mode requires matching token structure, finite values, and $|x - y| \leq \epsilon_{\text{abs}} + \epsilon_{\text{rel}}|x|$ for expected value x. Every checked input must pass; compilation failure, output mismatch, execution failure, or timeout yields zero credit. The CPU program defines expected behavior, but the deployed service does not execute it for every submission: initial grading checks the GPU reference against CPU outputs and caches CPU timing, and later grades may use the validated GPU reference on new inputs. We preserve this distinction in the trial metadata and report a common-oracle sensitivity check; mutation-based oracle assessment (Du et al., 2026) is not part of the reported runs.

All successful trials use an NVIDIA H100 80GB and eight paired timing rounds: after one warm-up per binary, candidate and reference run back to back with order reversed on alternating rounds, and outputs are checked during timing. The in-process timer measures `main()` after context

creation, charging host computation, data generation, transfers, and GPU work at one configured input scale per task. Speed is scored against the direct GPU port rather than the CPU program because a CPU-relative ratio mixes the hardware gain of moving any reasonable implementation to an H100 with the optimization an agent adds beyond it; cached CPU timings place the median valid port of the leading configurations hundreds of times above the CPU program (Appendix B). The reference removes that baseline shift at the cost of inheriting the reference's quality, which is why we report raw ratios and flag extreme values.

4.2. Separating Reliability from Optimization

For task j, let $C_j \in \{0, 1\}$ indicate a valid submission and $r_{ji} = t_{\mathrm{ref},ji}/t_{\mathrm{agent},ji}$ the ratio in round i. Correctness is $P = N^{-1} \sum_j C_j$. For valid submissions, the geometric-mean ratio is $R_j = \exp(\frac{1}{8} \sum_i \log r_{ji})$, and following the correctness-plus-speed criterion of KernelBench (Ouyang et al., 2025) we report

$$F_p = \frac{1}{N} \sum_{j=1}^{N} \mathbf{1}[C_j = 1 \text{ and } R_j > p], \quad p \in \{1, 10, 100\}. \tag{1}$$

All metrics keep the full planned denominator, including unresolved outcomes, so F_p answers how often a task yields a valid port exceeding a given speedup, not how fast a selected subset of successes is. The deployed reward is

$$S_j = \begin{cases} \frac{1}{8} \sum_{i=1}^{8} \mathrm{clip}(\log_{10} r_{ji}, 0, 4), & C_j = 1, \\ 0, & C_j = 0, \end{cases} \tag{2}$$

and we report $\bar{S} = N^{-1} \sum_j S_j$. Log scaling gives diminishing credit to very large ratios and clipping limits a task's contribution to $4/N$. The metrics are not interchangeable: clipping each round before averaging can give $S_j > 0$ when $R_j \leq 1$, so F_1 is not the fraction with positive score and $10^{\bar{S}}$ is not a geometric-mean speedup. We also use the exact decomposition

$$\bar{S} = P\, \bar{S}_{\mathrm{valid}}, \qquad \bar{S}_{\mathrm{valid}} = \frac{\sum_{j:C_j=1} S_j}{\sum_j C_j}, \tag{3}$$

whose conditional mean separates reliability from optimization but conditions on different task sets across configurations. Read together, P measures how often semantic transfer succeeds, F_{10} and F_{100} how often success also delivers one or two orders of magnitude over a direct port, and $\bar{S}$ graded improvement with bounded influence per task; no single column of Table 1 is a ranking.

4.3. Resources, Uncertainty, and Audits

Generation time is recorded execution time for retained attempts, excluding queueing, setup, and verification. Cost combines logged API-equivalent usage with explicitly identified public-tariff estimates for Astra and Kimi (Appendix A); it is neither a bill nor a full systems cost, discarded attempts are outside both totals, and missing usage is not imputed. A configuration is dominated if another has no greater mean resource use and no lower $\bar{S}$, with one strict inequality; we report non-dominated sets separately for time and cost. We use 10,000 paired bootstrap resamples of task indices shared across configurations; percentile intervals describe sensitivity to task composition, not variability across new agent runs, and the frequency with which a point stays non-dominated across resamples is not a probability of optimality. Alternative averages weight each domain or category equally. These post-hoc analyses are descriptive and do not correct for dependence among related tasks.

The verifier checks output on timed executions, isolates the candidate from the reference binary, and requires a CUPTI trace with a completed kernel and positive GPU duration; this rejects submissions with no GPU activity but cannot establish that the principal computation ran on the GPU. A raw ratio of at least $1000\times$ triggers an audit flag; flagged outcomes stay in the main results with a zero-credit sensitivity check, and a flag is not a completed audit (Appendix F).

5. Experiments

5.1. Setup

We analyze the seven completed configurations in the 2026-10-22 snapshot over the same 258 immutable task packages (dataset version 1.5.4): 1,806 planned outcomes, comprising 1,802 final grades and four unresolved outcomes. Astra, Sol, Luna, and Kimi K2.6 run through Codex 0.162.0; Sonnet 5.5 and Haiku 5.5 use Claude Code 2.1.296; the consolidated Opus 5.5 run spans Claude Code 2.1.296–2.1.307 and retains 124 baseline trials plus 134 trials with updated workflow instructions, with 256 graded sources and two missing artifacts. Each row keeps one final outcome per task; operational retries are not independent repetitions. Astra, Sonnet, and Haiku have six-hour generation caps, Opus two hours, and Sol, Luna, and Kimi one hour. Model, scaffold, route, prompt history, and cap vary together, so the comparison concerns observed configurations rather than isolated model effects or a controlled scaling law. Appendix A records provenance, cohort differences, and pricing; Appendix G describes the analysis artifact.

5.2. RQ1: Correctness Does Not Imply Strong Acceleration

Astra has the highest correctness (249/258, 96.5%), while Opus has the highest mean score (1.086) with 246 valid ports (95.3%); their tenfold coverage is 38.4% and 51.9% (Table 1). Correctness and optimization thus yield different

Table 1. **Full-suite correctness and speed on all 258 planned tasks.** F_p is the percentage of tasks with a valid port whose raw geometric-mean ratio over the direct reference exceeds p; $\bar{S}_{\text{valid}}$ conditions on each configuration's valid submissions.

		Speedup coverage (%)			Mean score	
Configuration	P (%)	F_1	F_{10}	F_{100}	$\bar{S}$	$\bar{S}_{\text{valid}}$
Opus 5.5	95.3	95.3	51.9	8.5	1.086	1.139
Astra	96.5	95.0	38.4	5.4	0.909	0.942
Sonnet 5.5	94.2	92.2	35.7	4.7	0.816	0.866
Haiku 5.5	91.9	86.4	18.6	3.1	0.585	0.637
Sol	89.1	71.3	13.6	1.6	0.404	0.454
Luna	71.7	31.8	4.3	0.0	0.132	0.184
Kimi K2.6	45.3	21.3	1.9	0.4	0.094	0.207

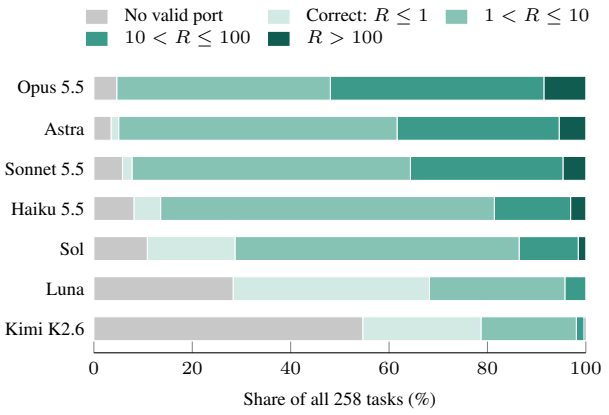


Figure 2. **Correctness and raw speedup are distinct outcomes.** Each bar covers all 258 planned tasks; gray marks failures and unresolved outcomes. R is the eight-round geometric-mean speedup over the direct GPU reference. Numbers are percentages of tasks. Astra and Sonnet place most valid ports in the $1 < R \leq 10$ band, Opus places more valid ports above $10\times$ (134) than at or below it (112), and the lower-scoring configurations place a growing share at or below parity.

leaders. Even for Opus, 112 of 246 valid ports do not exceed $10\times$ the direct reference and only 22 exceed $100\times$. This is reference-relative: by the cached CPU timings, no valid Opus or Astra port is slower than the CPU program (Appendix B). The gap widens for lower-scoring configurations: Sol has 230 valid ports but only 184 beat the reference, and for Luna and Kimi the median valid port is no faster than the reference (median ratios 0.9), against 11.6 for Opus. Correctness alone would obscure these differences. Sonnet and Haiku differ by 2.3 percentage points in correctness but by 17.1 points in F_{10} and 0.231 in mean score; Equation 3 gives conditional scores of 0.866 and 0.637, so the difference is not explained by the fraction of valid outputs alone.

Paired resampling separates the two outcomes. Opus's score advantage over Astra is 0.177 (95% interval $[0.124, 0.227]$) while its correctness difference is -1.16 points ($[-4.65, 2.33]$); Opus scores higher on 188 tasks and lower on 68. Astra's score advantage over Sonnet is 0.093 ($[0.041, 0.148]$). These are task-composition intervals for fixed trajectories; they do not remove protocol differences or establish significance across repeated runs (Appendix Table 6).

Where valid ports land. Figure 2 partitions each configuration's 258 outcomes into failures and four raw-ratio bands, and the bands change shape down the ordering rather than merely shrinking. The three leading configurations concentrate valid ports in the $(1, 10]$ band: 112 of Opus's 246, and 146 each of Astra's 249 and Sonnet's 243. Above $10\times$, Opus holds 112 ports in $(10, 100]$ and 22 above $100\times$, against 85 and 14 for Astra and 80 and 12 for Sonnet, so Opus's higher F_{10} reflects a larger share of substantial speedups despite fewer valid ports. Further down, the mass shifts to parity or below: 14 of Haiku's 237 valid ports are at or below the reference, 46 of Sol's 230, 103 of Luna's 185, and 62 of Kimi's 117. For Luna, more valid ports are at or below parity than above it, so its 71.7% correctness describes functional transfer with little acceleration beyond a direct port. The top band is sparse for every configuration: only 23 tasks have any port above $100\times$, and the single result above $1000\times$ is the flagged Astra submission discussed in Section 5.5.

The bands are reported on raw ratios rather than on the clipped score because the two answer different questions. The score clips each round's log ratio at zero before averaging, so a port that alternates between $2\times$ and $0.5\times$ across rounds earns a positive score with a geometric-mean ratio of one; Luna has 96 trials with positive score but only 82 with $R > 1$. The log scale also compresses the upper tail: consistent per-round ratios of $100\times$ and $1000\times$ yield scores of 2 and 3, respectively, and clipping caps each task at 4. The mean score summarizes optimization across tasks, while F_1, F_{10}, and F_{100} state how often a configuration clears a threshold a user can act on.

Failures and successes. Kimi has 95 compilation failures and 33 output mismatches, Luna 28 and 30; Opus has six, three, and one execution failure among 256 graded sources (Appendix Table 5). For the three leading configurations, compilation and output failures account for at most 13 tasks each, so the remaining gap to tenfold coverage lies almost entirely among valid ports. Build failures are part of measured transfer difficulty; the records do not show how many target-compiler feedback would repair. Section 5.3 inspects two valid Astra ports with contrasting speedups.

Takeaway. Measured correctness is high for the leading configurations, between 94% and 97% under the specified oracle, but 46% to 62% of their valid ports are at most $10\times$

Table 2. **Cost–reward frontier**, ordered by API-equivalent cost. Correctness and F_{10} use all 258 tasks; cost, reward, and frontier membership use the 254 tasks with cost records for every configuration. Bold marks column optima. Stability is the share of 10,000 paired task resamples in which the point stays non-dominated, not a probability of optimality. †: public-tariff estimate from recorded tokens; other costs are logged API equivalents for retained attempts.

		Full suite ($N = 258$)		Common cost cohort ($N = 254$)				
Agent	Model	Correct (%)	F_{10} (%)	Cost (USD/task)	Reward $\bar{S}$	95% interval	Frontier	Stability (%)
Codex	GPT-6 Luna	71.7	4.3	**0.0129**	0.131	[0.093, 0.173]	✓	100.0
Codex	GPT-6 Sol	89.1	13.6	0.3443	0.396	[0.337, 0.459]	✓	56.4
Claude Code	Claude Haiku 5.5	91.9	18.6	0.3470	0.586	[0.519, 0.656]	✓	100.0
Codex	Kimi K2.6	45.3	1.9	0.7548†	0.088	[0.057, 0.124]	–	0.0
Claude Code	Claude Sonnet 5.5	94.2	35.7	0.8905	0.819	[0.748, 0.890]	✓	100.0
Codex	GPT-6 Astra	**96.5**	38.4	2.9731†	0.898	[0.826, 0.973]	✓	99.9
Claude Code	Claude Opus 5.5	95.3	**51.9**	6.2738	**1.091**	[1.017, 1.168]	✓	100.0

faster than a direct port, a band that includes the few ports slower than the reference. Pass rates alone would rank these configurations within a few points of one another.

5.3. What Fast Ports Change: Two Inspected Submissions

Source inspection of two Astra submissions documents concrete implementation differences behind the outcomes in Figure 2, but the inspection is descriptive: it pairs the submitted source with its existing measured outcome and involves no new replay or component ablation. On `dp-subset-sched`, a precedence-constrained scheduling dynamic program, the direct reference allocates a dense subset-by-last-job table and groups all subsets on the host. The candidate instead enumerates precedence-closed subsets on the GPU, stores each subset layer compactly, assigns one warp to each predecessor subset so that its lanes load the predecessor row once and reuse it across successors, and prunes candidate costs using the task's bounded setup penalties. Its measured ratio is $407\times$. On `numfn-popbase-euler`, a batched Euler-criterion test, the reference performs 128-bit products and remainders inside modular exponentiation. The candidate introduces Montgomery-style squaring with a precomputed inverse, consumes two exponent bits per iteration using factors that fit in 18 bits, and evaluates the Jacobi symbol through a small lookup. Its measured ratio is $5.8\times$. The first port restructures state exploration and storage; the second specializes modular arithmetic and exponent processing. Precedence constraints and operand bounds enable these transformations, motivating task packages that expose the full CPU program and input contract alongside public examples.

Comparing the other configurations on the same two tasks puts these examples in the context of a shared reference. On `dp-subset-sched` the six valid ports span $17\times$ (Sol), $45\times$ (Luna), $68\times$ (Sonnet), $307\times$ (Haiku), $367\times$ (Opus), and $407\times$ (Astra), a more than 24-fold range over one reference. On `numfn-popbase-euler` all six valid ports lie between $3.3\times$ (Luna) and $5.8\times$ (Astra); Kimi fails both tasks. These ranges show that equally valid submissions can differ substantially on one task yet cluster on another. They do not establish why: the comparison holds task identity and reference fixed, but not the model, scaffold, or search budget. In particular, the narrow second range does not rule out a better algorithm or measure the remaining headroom. Both examples are selected, not sampled, so they illustrate the kinds of change behind the bands rather than estimate how often each occurs. Source hashes, inspected functions, and inspection scope are recorded in the analysis artifact (Appendix B).

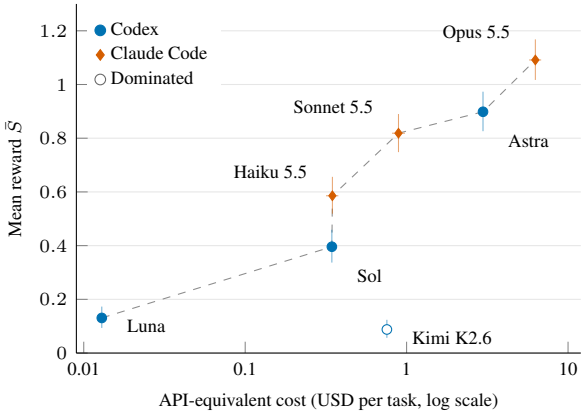


Figure 3. **Reward at a given generation cost** on the 254-task cohort of Table 2. Hue and marker shape identify the scaffold; filled markers lie on the observed frontier, the open marker is dominated. Bars are 95% task-bootstrap intervals; dashed segments show frontier order, not attainable interpolations.

5.4. RQ2: Resource Choice Changes the Frontier

Table 2 and Figure 3 place Codex and Claude Code configurations on a shared cost–reward frontier: Luna, Sol, Haiku, Sonnet, Astra, and Opus in order of cost. Opus reaches

the highest cohort reward (1.091) at $6.27 per task; Astra reaches 0.898 at $2.97 and Sonnet 0.819 at $0.89; Luna dominates Kimi on both axes. A single reward ranking would hide these budget-dependent alternatives: Sonnet to Opus costs about seven times as much for a 0.27 reward gain, whereas Sol to Haiku adds 0.19 at nearly identical cost. Costs cover retained attempts only, excluding subscription payments, retries, sandbox execution, and GPU evaluation.

In time–score space the frontier changes to Luna, Sol, Sonnet, Astra, and Opus at 4.0, 5.5, 12.1, 15.3, and 42.3 minutes per task (Appendix Figure 5). Haiku takes 21.2 minutes and scores below Sonnet, so it is time-dominated despite its cost-frontier membership; it also emits more than twice Sonnet's output tokens per task (Appendix B). Opus uses $2.8\times$ Astra's time and $2.1\times$ its cost. These associations do not establish a causal return to computation, since caps, scaffolds, and prompts differ. Membership can be fragile: Haiku's mean cost exceeds Sol's by $0.0026 per task (paired interval $[-\$0.026, \$0.032]$); Sol stays non-dominated in 56.4% of joint resamples, Astra in 99.9%, the other four cost-frontier members in all 10,000, and all five time-frontier members in at least 99.97%, conditional on stated tariffs and retained trajectories.

Takeaway. The configuration producing the fastest ports is also the most expensive, and the cheapest frontier points produce ports often no faster than a direct translation. Which frontier a user should consult depends on whether the binding constraint is wall-clock time or spend, so resources are part of the benchmark's output.

5.5. RQ3: Workload Gaps and Robustness of Conclusions

Efficiency gaps depend on the workload. Category outcomes (Figure 4; mean scores in Appendix Table 10) localize the gap. Opus is correct on all 20 graph-and-tree tasks and exceeds $10\times$ on 15 (mean score 1.563), but on all 13 simulation-and-finance tasks exceeds $10\times$ on only three (0.805); Astra is correct on all 18 physics-and-materials tasks, yet only five exceed $10\times$. Opus's tenfold coverage ranges from 23% (simulation and finance) to 86% (chemistry), Astra's from 11% (geometry and graphics) to 60% (graphs and trees). Graphs and trees is the most or second-most accelerated category for each of the five leading configurations, while earth and space, physics and materials, and simulation and finance fall below the overall F_{10} of Opus, Astra, Sonnet, and Haiku. In ML systems, Haiku, Sol, Luna, and Kimi are correct on 29–79% of 14 tasks but none exceeds $10\times$ on any. The runs do not establish whether dependencies, data movement, or reference quality causes a given category's gap; category membership is descriptive.

Coverage across configurations. Across all seven configurations, 257 tasks have a valid port and 146 have a port above $10\times$, leaving 111 with a valid port but no observed tenfold improvement; only `geo-psha-hazard` has no valid port. Opus is valid on 105 of the 111, and Opus, Astra, and Sonnet are all valid on 94. Their best observed ratios have median $5.4\times$, five lie in $(1, 2]$, and none is at or below parity. These tasks therefore have modest improvements rather than only failed submissions; the three categories above contain 36 of them. Tenfold coverage is only partly shared: Opus covers 134 tasks, and another configuration succeeds on 12 where Opus does not. Of the 146, 40 are covered by exactly one configuration: 32 by Opus, seven by Astra, and one by Sonnet. All three leading configurations exceed $10\times$ on 73 tasks, but all seven do so on only one. By comparison, all seven are correct on 73 tasks, of which 37 have any tenfold port.

This post-hoc union does not establish an attainable selector: generating all candidates incurs their combined costs, while choosing a configuration in advance needs a separately tested policy. Nor does it bound efficiency, since neither direct references nor agent submissions are certified optima. It identifies 111 concrete tasks for further optimization and reference audits, without assigning an efficiency ceiling.

Robustness. Equal weighting of the 16 categories preserves the full score ordering; equal weighting of 108 domains preserves the top five but reverses Luna and Kimi (domain means 0.123 and 0.127 versus task means 0.132 and 0.094), whose paired difference already spans zero ($[-0.006, 0.082]$). Opus remains highest under both (1.092; 1.070). One Astra submission, `graph-mincut-stoer-wagner`, is flagged at approximately $1870\times$; zeroing it reduces Astra's mean from 0.909 to 0.897 and preserves its lead over Sonnet, without validating the submission or the reference. The records mix CPU-based and GPU-reference expected outputs (all 256 Opus grades are GPU-expected); on the 155 tasks GPU-expected for all seven configurations, scores are 1.029, 0.841, 0.771, 0.543, 0.383, 0.110, and 0.093 in Table 1 order, preserving the ordering. This selected subset cannot establish oracle equivalence (Appendix C).

Takeaway. The ordering of the five leading configurations survives category and domain balancing, removal of the flagged outlier, and restriction to a common oracle. The gap's location is concrete: simulation and finance, physics and materials, and earth and space are below the overall tenfold coverage of Opus, Astra, Sonnet, and Haiku and hold a third of the uncovered tasks, making them specific targets for agents, scaffolds, and reference audits.

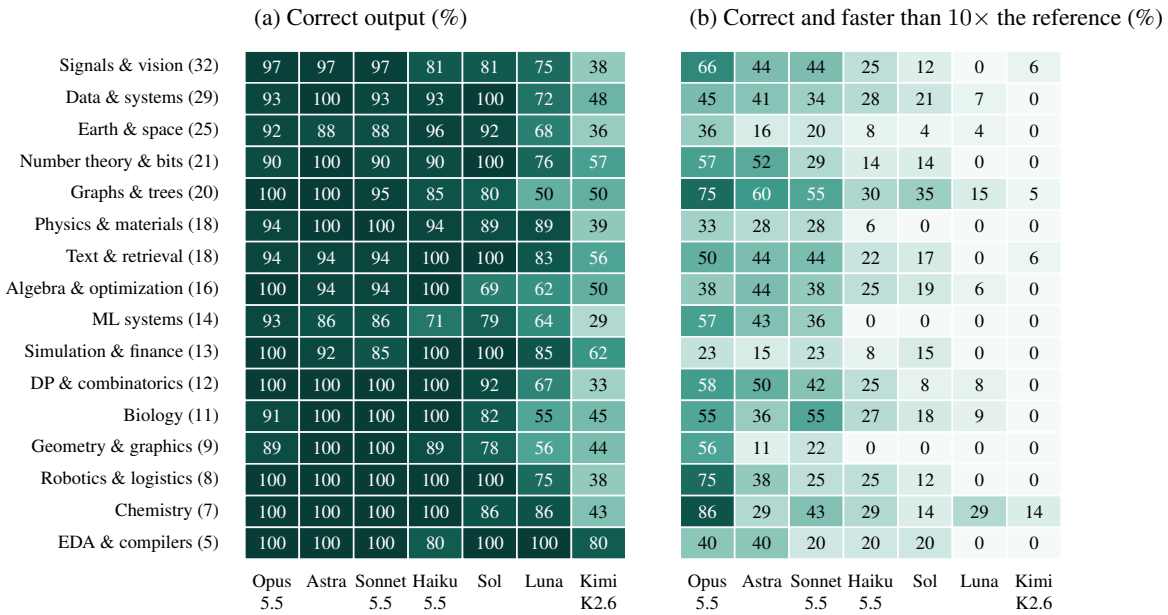


Figure 4. **Functional coverage and efficiency by category.** (a) Percentage of tasks with valid output. (b) F_{10}: percentage with a valid port exceeding $10\times$ the direct reference. Panels share one 0–100% color scale; cells are rounded and labels give task counts; failures and unresolved outcomes stay in every denominator. Panel (a) is nearly saturated for the leading configurations, whereas panel (b) varies by more than a factor of three across categories for the same configuration.

6. Limitations and Implications

The experiments measure final-artifact transfer without provided target-GPU feedback on a constructed task set and one GPU architecture; they do not estimate feedback-enabled performance, production migration effort, or a model's ability independent of its scaffold. Unequal caps, one retained trajectory per task, and the mixed Opus cohort (Section 5.1) prevent a controlled comparison of inference scaling, and task resampling cannot replace repeated generations under matched budgets. Reference-relative speed depends on reference quality, input scale, and timing boundaries; the clipped score does not penalize a valid slowdown more than parity, so F_1 and the outcome breakdown are necessary complements. A completed GPU kernel does not certify that the main work ran on the GPU, an in-process timer does not guarantee that asynchronous work is charged, and the expected-output provenance is mixed. Stronger CPU-oracle revalidation, reference audits, and adversarial evaluation are required before treating scores as robust to deliberate exploitation. Domain labels and fresh inputs do not establish contamination freedom or generalization to unseen algorithm families, and no held-out training split or learning result is claimed.

Three practices transfer to other efficiency benchmarks: keep failures in every denominator, report raw speedup thresholds alongside any transformed score because clipping and log scaling change what a scalar means near parity, and attach generation resources to every result, since the same seven configurations order differently under time and cost. A next evaluation should cross compiler and GPU feedback with matched budgets and repeated trajectories, separating semantic transfer, repairability, and hardware-guided optimization. Future evaluations should retain immutable CUDA artifacts and verifier versions, so regrading can isolate measurement variability from agent variability. Any change to inputs, references, or timing boundaries should define a new benchmark version and include paired scores under both protocols, keeping historical results interpretable as the benchmark evolves.

7. Conclusion

PortBench evaluates whether coding agents can preserve a complete CPU program's observable behavior while producing an efficient GPU implementation. Across seven configurations, Astra achieves the highest correctness and Opus the highest mean optimization score, yet 46% to 62% of the leading configurations' valid ports are at most $10\times$ faster than a direct port, some slower than the reference. The 16-category analysis locates this gap, including 111 tasks with a valid port but no tenfold improvement from any configuration, concentrated in simulation, physics, earth-science, and data-systems tasks. Generation time and API-equivalent cost produce different empirical frontiers. These results are a reproducible starting point for improving GPU efficiency; neither the direct references nor the observed frontiers define its attainable ceiling.

Impact Statement

This work evaluates automated program migration and performance engineering. More efficient implementations could reduce the cost of repeated computation, but neither energy savings nor environmental benefit is measured here; generation and verification also consume resources. Faster execution can amplify harmful as well as beneficial applications. The task inputs are synthetic and do not contain personal records. Benchmark scores should not be treated as certification that generated code is secure or correct beyond tested inputs. The intended contribution is an auditable evaluation of these capabilities and their limitations.

References

Davis, J. H., Nichols, D., Khillan, I., and Bhatele, A. ParEval-Repo: A Benchmark Suite for Evaluating LLMs with Repository-level HPC Translation Tasks. *arXiv preprint arXiv:2506.20938*, 2025.

Du, M., Luu, A. T., Ji, B., Liu, Q., and Ng, S.-K. Mercury: A Code Efficiency Benchmark for Code Large Language Models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 16601–16622, 2024. doi: 10.52202/079017-0529. URL https://arxiv.org/abs/2402.07844.

Du, M., Luu, A. T., Ji, B., Wu, X., Huang, D., Qing, Y., Zhuo, T. Y., Liu, Q., and Ng, S.-K. CodeArena: A Collective Evaluation Platform for LLM Code Generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 502–512. Association for Computational Linguistics, 2025a. doi: 10.18653/v1/2025.acl-demo.48. URL https://aclanthology.org/2025.acl-demo.48/.

Du, M., Luu, A. T., Liu, Y., Qing, Y., Huang, D., He, X., Liu, Q., Ma, Z., and Ng, S.-K. Afterburner: Reinforcement Learning Facilitates Self-Improving Code Efficiency Optimization. In *Advances in Neural Information Processing Systems*, volume 38, pp. 8979–9011, 2025b. doi: 10.52202/085713-0304. URL https://arxiv.org/abs/2505.23387.

Du, M., Luu, A. T., Huang, D., and Ng, S.-K. Measuring the Checker: Mutation Analysis for GPU-Kernel Benchmark Oracles. *arXiv preprint arXiv:2609.22220*, 2026. doi: 10.48550/arXiv.2609.22220. URL https://arxiv.org/abs/2609.22220.

He, X., Liu, Q., Du, M., Yan, L., Fan, Z., Huang, Y., Zheng, Y., Yuan, Z., and Ma, Z. SWE-Perf: Can Language Models Optimize Code Performance on Real-World Repositories? In *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pp. 41651–41666. PMLR, 2026. URL https://proceedings.mlr.press/v306/he26y.html.

Huang, D., Du, M., Zhang, J. M., Lin, Z., Luo, M., Zhang, Q., and Ng, S.-K. Nexus: Execution-Grounded Multi-Agent Test Oracle Synthesis. *arXiv preprint arXiv:2510.26423*, 2025. doi: 10.48550/arXiv.2510.26423. URL https://arxiv.org/abs/2510.26423.

Huang, D., Harman, M., Zhang, J. M., Guo, Z., Du, M., and Ng, S.-K. Ockhamareto: Pareto-Gated Segment-Level Credit Assignment for Concise Unit-Test Generation with Reinforcement Learning. *arXiv preprint arXiv:2608.24473*, 2026a. doi: 10.48550/arXiv.2608.24473. URL https://arxiv.org/abs/2608.24473.

Huang, D., Zhang, J. M., Harman, M., Du, M., and Cui, H. Measuring the Influence of Incorrect Code on Test Generation. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 2026b. URL https://arxiv.org/abs/2409.09464.

Huang, D., Zhang, J. M., Harman, M., Zhang, Q., Du, M., and Ng, S.-K. Benchmarking LLMs for Unit Test Generation from Real-World Functions. *ACM Transactions on Software Engineering and Methodology*, 2026c. doi: 10.1145/3805043. URL https://doi.org/10.1145/3805043. In press.

Ma, J. J., Hashemi, M., Yazdanbakhsh, A., Swersky, K., Press, O., Li, E., Reddi, V. J., and Ranganathan, P. SWE-fficiency: Can Language Models Optimize Real-World Repositories on Real Workloads? *arXiv preprint arXiv:2511.06090*, 2025.

Moonshot AI. Kimi API platform: K2.6 pricing. https://platform.kimi.ai/, 2026. Accessed October 22, 2026; public USD token prices.

Nichols, D., Davis, J. H., Xie, Z., Rajaram, A., and Bhatele, A. Can Large Language Models Write Parallel Code? In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2024. doi: 10.1145/3625549.3658689.

OpenAI. GPT-6 Astra: Model and API pricing. https://developers.openai.com/api/docs/models/gpt-6-astra, 2026. Accessed October 22, 2026; Standard-tier public tariff.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Ré, C., and Mirhoseini, A. KernelBench: Can LLMs Write Efficient GPU Kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Press, O., Amos, B., Zhao, H., Wu, Y., Ainsworth, S. K., Krupke, D., Kidger, P., Sajed, T., Stellato, B., Park, J., Bosch, N., Meril, E., Steppi, A., Zharmagambetov, A., Zhang, F., Perez-Pineiro, D., Mercurio, A., Zhan, N., Abramovich, T., Lieret, K., Zhang, H., Huang, S., Bethge, M., and Press, O. AlgoTune: Can Language Models Speed Up General-Purpose Numerical Programs? *arXiv preprint arXiv:2507.15887*, 2025.

Qing, Y., Zhu, B., Du, M., Guo, Z., Zhuo, T. Y., Zhang, Q., Zhang, J. M., Cui, H., Yiu, S.-M., Huang, D., Ng, S.-K., and Luu, A. T. EffiBench-X: A Multi-Language Benchmark for Measuring Efficiency of LLM-Generated Code. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2491. URL https://arxiv.org/abs/2505.13004.

Shetty, M., Jain, N., Liu, J., Kethanaboyina, V., Sen, K., and Stoica, I. GSO: Challenging Software Optimization Tasks for Evaluating SWE-Agents. *arXiv preprint arXiv:2505.23671*, 2025.

Wu, X., Pan, L., Xie, Y., Zhou, R., Zhao, S., Ma, Y., Du, M., Mao, R., Luu, A. T., and Wang, W. Y. AntiLeak-Bench: Preventing Data Contamination by Automatically Constructing Benchmarks with Updated Real-World Knowledge. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 18403–18419. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.901. URL https://aclanthology.org/2025.acl-long.901/.

Zhu, X., Zhou, X., Zhu, B., Hu, H., Du, M., Zhang, H., Wang, H., and Guo, Z. CodeScaler: Scaling Code LLM Training and Test-Time Inference via Reward Models. *arXiv preprint arXiv:2602.17684*, 2026. doi: 10.48550/arXiv.2602.17684. URL https://arxiv.org/abs/2602.17684. Accepted to NeurIPS 2026.

A. Configuration and Result Provenance

The analysis fixes dataset version 1.5.4 and the common immutable dataset digest

`sha256:320024e75e721ef021e217d0a79cf7205404533e0fbc5a5efc2cf7d3e6510d56`.

All seven jobs contain the same 258 task names and content-addressed task references. Codex reports version 0.162.0; Sonnet and Haiku use Claude Code 2.1.296. Opus spans versions 2.1.296 (33 trials), 2.1.299 (39), 2.1.300 (28), 2.1.304 (24), 2.1.305 (9), 2.1.306 (118), and 2.1.307 (7). Each adapter requests high reasoning effort, which does not establish equivalent effective reasoning budgets. The names below are display labels for recorded routes, not independent verification of upstream identity; Astra corresponds to the `gpt-6-astra` family label. Actual routes remain in the configuration metadata.

Opus consolidation. The Opus job retains 124 completed baseline trials and 134 later trials with an appended instruction to defer local CUDA compilation and save a complete candidate early. Bash and ordinary tools remain available; attempts using disabled tools or seeded draft candidates are excluded. This operational consolidation was fixed before the final GPU scoring pass, rather than selecting the highest reward across attempts. All 256 collected sources were evaluated on 2026-10-22 using the same H100 service, eight-round timing protocol, and GPU-expected outputs. One source saved before an agent timeout passed verification. Artifact downloads failed for `pde-eikonal-fastsweep` and `phys-sn-transport-sweep`; neither was regenerated for this evaluation. The version and prompt changes prevent interpreting the consolidated row as a uniform-protocol replication.

Table 3. Evaluated configurations. The cap is the configured generation timeout, not measured execution time.

Label	Scaffold	Job directory	Cap (h)
Opus 5.5	Claude Code	`2026-10-22__opus-5-5-consolidated`	2
Astra	Codex	`2026-10-11__08-44-38-codex`	6
Sonnet 5.5	Claude Code	`2026-10-11__08-44-38-sonnet`	6
Haiku 5.5	Claude Code	`2026-10-11__08-44-38-haiku`	6
Sol	Codex	`2026-10-11__18-23-40`	1
Luna	Codex	`2026-10-11__18-23-07`	1
Kimi K2.6	Codex	`2026-10-12__08-04-40`	1

Table 4. Uncertainty and retained-attempt generation resources. Score intervals resample tasks, not independent trajectories. Minutes and output tokens (thousands) are per-task means; cost is a sum. Parentheses give the number of tasks with usage records out of 258. Astra and Kimi (†) use public-tariff estimates; the others are logged API equivalents. Missing usage stays unknown; these are not full-job bills.

Model	Correct	Score 95% interval	Minutes	Output tokens, k (n)	Cost, USD (n)
Claude Opus 5.5	246/258	[1.014, 1.159]	42.32	162.9 (256)	1611.96 (256)
GPT-6 Astra	249/258	[0.838, 0.984]	15.34	31.8 (258)	765.54† (258)
Claude Sonnet 5.5	243/258	[0.748, 0.886]	12.12	54.6 (257)	229.40 (257)
Claude Haiku 5.5	237/258	[0.520, 0.652]	21.23	130.4 (257)	88.62 (257)
GPT-6 Sol	230/258	[0.342, 0.468]	5.53	18.4 (258)	88.74 (258)
GPT-6 Luna	185/258	[0.094, 0.174]	4.01	13.7 (258)	3.33 (258)
Kimi K2.6	117/258	[0.062, 0.130]	32.35	88.3 (258)	194.05† (258)

Table 5. Final unsuccessful outcomes. Runtime covers execution and timing failures; "no source" is a missing collected artifact; "unresolved" is a service error without a final grade. Each row sums to 258 minus the correct count.

Model	Compile	Wrong output	Runtime	No source	Unresolved
Claude Opus 5.5	6	3	1	0	2
GPT-6 Astra	3	4	0	2	0
Claude Sonnet 5.5	6	7	1	0	1
Claude Haiku 5.5	11	6	3	0	1
GPT-6 Sol	10	13	5	0	0
GPT-6 Luna	28	30	14	1	0
Kimi K2.6	95	33	11	2	0

Reconstruction. `scripts/paper_results.py` reads each selected trial's final result and verifier-detail records. It checks equality of immutable task references and reconstructs every valid trial's clipped score from eight paired timings. The resulting CSV contains 1,806 rows, preserving task, job and trial identifiers, final outcomes, usage, cost basis, expected-output provenance, and hashes of source records. Cached tokens are already included in recorded input-token totals and are not added again. Consolidation provenance preserves original source paths and hashes; superseded jobs and aggregate summaries are not independent observations.

Missingness and denominators. Every configuration retains all 258 planned tasks. Unresolved service errors contribute zero to P, F_p, and $\bar{S}$ but are not reclassified as candidate compilation or semantic errors. Opus has 256 grades, 246 valid outputs, and ten graded failures. Its evaluated-only mean is 1.0946, whereas the 258-task mean is 1.0861. The conditional-on-valid score in Table 1 uses 246 successes, a third distinct denominator. Sonnet and Haiku each lack one usage record, and Opus lacks two, on four distinct tasks. Table 2 and the cost figure use the resulting 254-task intersection for cost and reward (Opus: 1.0914); no missing usage is imputed. The table's correctness and F_{10} columns retain the full 258-task denominator, as indicated by their grouped header.

Dated API-equivalent cost estimates. Astra and Kimi costs are reconstructed from retained token usage using public prices accessed on 2026-10-22 (OpenAI, 2026; Moonshot AI, 2026). Astra uses \$10/M ordinary input tokens, \$1/M cache reads, \$12.50/M cache writes, and \$50/M output, totaling \$765.54. Standard tier and 30-minute cache writes are assumptions because tier and cache lifetime are not logged; no request exceeds the 272K input threshold. Kimi uses \$0.95/M uncached input, \$0.16/M cache hits, and \$4/M output, totaling \$194.05, with no separate cache-write fee. Its run used a self-hosted export, so this is a counterfactual API price rather than a hosting bill. Both estimates avoid double-counting cached input or reasoning output. The other totals retain logged API-equivalent costs, including \$1,611.96 for 256 Opus usage records. Tariff sources, assumptions, and per-trial calculations are preserved in each job's cost-estimation receipt. All totals exclude discarded attempts, GPU evaluation, sandbox hosting, and any unrecorded surcharges.

B. Additional Experimental Detail

This section collects descriptive quantities referenced in Section 5. All are computed from the same 1,806-row snapshot and the frozen task catalog; none involves new generations or GPU evaluations.

Raw ratios among valid ports. The median geometric-mean ratio R among valid ports is 11.6 (Opus), 7.1 (Astra), 5.9 (Sonnet), 3.4 (Haiku), 1.9 (Sol), 0.9 (Luna), and 0.9 (Kimi). The band counts behind Figure 2 in the main text are, in the same order, 0, 4, 5, 14, 46, 103, and 62 valid ports with $R \leq 1$; 112, 146, 146, 175, 149, 71, and 50 in $(1, 10]$; 112, 85, 80, 40, 31, 11, and 4 in $(10, 100]$; and 22, 14, 12, 8, 4, 0, and 1 above $100\times$.

CPU-relative counts. The verifier records a single cached timing of the CPU program on the timing input; it is not a paired measurement and is not scored. Relative to it, the number of valid ports slower than the CPU program is 0 (Opus), 0 (Astra), 1 (Sonnet), 1 (Haiku), 3 (Sol), 13 (Luna), and 5 (Kimi), and the median CPU-relative speedup among valid ports is approximately 800, 531, 384, 267, 145, 63, and 78. These exploratory counts support the statement that reference-relative gaps do not imply failure to accelerate the CPU program; they do not calibrate hardware gains.

Task-level pairwise outcomes. Opus scores higher than Astra on 188 tasks, ties on two, and scores lower on 68; among the 238 tasks where both are valid, the Opus port is faster on 180. Astra scores higher than Sonnet on 151 tasks and lower on 105; Sonnet beats Haiku on 186 and loses on 68. Among the 146 tasks with any tenfold port, the number of configurations above $10\times$ is one on 40 tasks, two on 26, three on 30, four on 21, five on 17, six on 11, and seven on one.

Union coverage by category. Among the 257 tasks with at least one valid port, the best observed ratio exceeds $100\times$ on 23 tasks, lies in $(10, 100]$ on 123, in $(2, 10]$ on 106, and in $(1, 2]$ on five; no task's best valid port is slower than the reference. The 111 tasks with a valid port but no tenfold improvement (Section 5.5) include 15 of 29 in data and systems, 15 of 25 in earth and space, 11 of 18 in physics and materials, 10 of 13 in simulation and finance, 10 of 32 in signals and vision, and 5 of 20 in graphs and trees; relative to category size they are most prevalent in simulation and finance, physics and materials, earth and space, and data and systems.

Output tokens and time. Among the 257 tasks with usage records for each, Haiku's retained trajectories emit 130.4 thousand output tokens per task on average against 54.6 thousand for Sonnet (Table 4). More output tokens per task is consistent with Haiku's longer wall-clock time, but token counts alone do not identify whether the difference reflects more steps, longer outputs, or tool logging.

Source inspection of two Astra submissions. The two submissions discussed in Section 5.3 are trials `dp-subset-sched__bQvDANG` (407.3×; inspected functions `enumerate_ideals`, `advance_layer`, `prepare`) and `numfn-popbase-euler__pfCbGtg` (5.82×; `square_montgomery`, `euler_power`, `jacobi_small`), each compared with the task's `gpu_ref.cu`. The existing outcomes verify the submitted programs on tested inputs; source inspection alone cannot attribute the speedup to individual changes. Candidate and reference source hashes and the inspection scope are recorded in `generated/analysis_case_studies.json`.

C. Paired Comparisons and Robustness

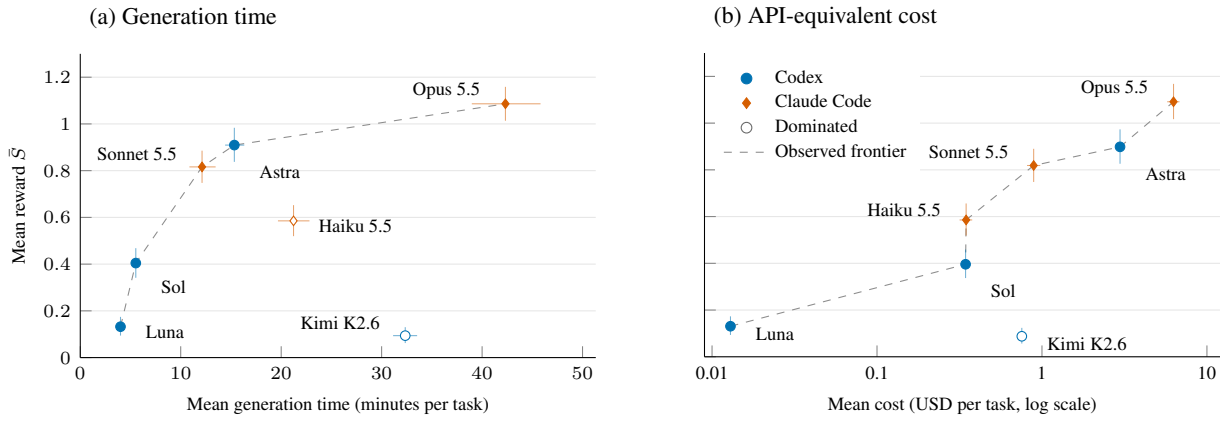


Figure 5. **Resource choice changes frontier membership.** (a) Retained-attempt generation time and reward on all 258 tasks; (b) API-equivalent cost and reward on the 254 common tasks, with the same vertical scale. Conventions follow Figure 3: hue and shape identify the scaffold, open markers are dominated, dashed segments connect observed frontier points in order, and bars are 95% task-bootstrap intervals. Astra and Kimi costs are public-tariff estimates.

Resampling units. Point intervals in the resource figure use 10,000 shared task resamples with seed 20261016 (20261017 for the cost cohort). Paired contrasts and frontier-membership analysis use seed 20261017 and 10,000 resamples, generated by `scripts/paper_analysis.py`. Within a resample, the same task indices select scores and resources for every included configuration. A separate set of shared indices is drawn for the 254-task cost cohort. We report percentile 95% intervals without a multiple-comparison adjustment; contrasts are descriptive post-hoc analyses. The paired contrast conditions on a task identifier, not necessarily the same random hidden inputs across jobs or a shared generation protocol.

Table 6. Paired contrasts between adjacent configurations in the task-mean ordering (first minus second). Correctness differences are percentage points. Intervals describe task-composition sensitivity for fixed observed trajectories.

	Mean score		Correctness (pp)	
Contrast	$\Delta\bar{S}$	95% interval	ΔP	95% interval
Opus – Astra	0.177	[0.124, 0.227]	-1.16	[-4.65, 2.33]
Astra – Sonnet	0.093	[0.041, 0.148]	2.33	[-0.78, 5.81]
Sonnet – Haiku	0.231	[0.178, 0.284]	2.33	[-1.94, 6.59]
Haiku – Sol	0.181	[0.123, 0.238]	2.71	[-2.33, 7.75]
Sol – Luna	0.272	[0.218, 0.330]	17.44	[11.24, 23.64]
Luna – Kimi	0.038	[-0.006, 0.082]	26.36	[19.37, 33.72]

Score boundaries. A consistent 10× reference improvement yields $S = 1$, and 100× yields $S = 2$. Because the transformation is clipped within each round, these values cannot generally be recovered by transforming the reported

Table 7. Weighting and audit sensitivity of the mean score. Group and domain means average within each of the 16 or 108 labels and then weight labels equally. Audit-zero keeps all 258 tasks but assigns zero to audit-flagged scores. Each weighting defines a different target distribution.

Configuration	Task mean	Group mean	Domain mean	Audit-zero
Opus 5.5	1.086	1.092	1.070	1.086
Astra	0.909	0.907	0.876	0.897
Sonnet 5.5	0.816	0.821	0.816	0.816
Haiku 5.5	0.585	0.569	0.582	0.585
Sol	0.404	0.383	0.414	0.404
Luna	0.132	0.140	0.123	0.132
Kimi K2.6	0.094	0.101	0.127	0.094

Table 8. Percentage of paired task resamples in which a configuration is non-dominated, conditional on recorded trajectories and stated tariffs. These are resampling frequencies, not posterior probabilities of optimality. Astra's unrounded time frequency is 99.97%.

Configuration	Time frontier, 258 tasks (%)	Cost frontier, 254 tasks (%)
Opus 5.5	100.0	100.0
Astra	100.0	99.9
Sonnet 5.5	100.0	100.0
Haiku 5.5	0.0	100.0
Sol	100.0	56.4
Luna	100.0	100.0
Kimi K2.6	0.0	0.0

geometric ratio. For example, ratios 2 and $1/2$ have geometric mean 1, but their mean clipped log score is positive. In the observed data, Luna has 96 positive-score trials but only 82 with a raw ratio above one. This is why the main table uses $R > p$ directly. The score also assigns zero to a valid slowdown, so it should not be interpreted as a signed performance regret.

Oracle subset. There are 155 common tasks with explicit GPU-expected provenance for all seven configurations. The subset is selected by stored verification history, rather than randomized assignment of oracle type. Its preserved ordering cannot identify the effect of replacing the CPU oracle with the GPU reference. Records with unspecified provenance remain unspecified in the snapshot.

D. Task Categories and Observed Efficiency

The 16 categories in Table 9 use a fixed mapping from the 108 source-domain labels. We freeze the mapping, each task's content-addressed reference, its description, and its exact/numeric comparison rule in `generated/task_category_catalog.json`. Comparison rules are read from the version-1.5.4 verifier specifications, rather than inferred from task names. Every task belongs to exactly one category; the categories are descriptive reporting units and do not imply independent algorithm families. Figure 4 in the main text shows correctness and tenfold coverage by category; Table 10 gives the corresponding mean scores.

Interpreting the remaining gap. Section 5.5 reports the union of the seven configurations: 257 tasks with at least one correct submission, 146 with at least one raw speedup above $10\times$, and `geo-psha-hazard` with no valid port. By category, union tenfold coverage is 3/13 in simulation and finance and 7/18 in physics and materials, despite complete union correctness in both. The per-task union counts in Appendix B are computed from the same 1,806-row snapshot by joining each task's seven outcomes; they neither estimate the cost of an agent-selection strategy nor prove intrinsic difficulty. Explaining a gap requires reference audits, profiling, or controlled experiments beyond these observations.

Reproduction. `scripts/paper_categories.py` joins the frozen task catalog with the same 1,806-row outcome CSV used for the overall evaluation. It verifies unique task membership, identical task references across configurations, the 219/39 exact/numeric split, and agreement of category sums and weighted means with the overall results. The generated `category_results.json` includes per-configuration correctness, raw $F_1/F_{10}/F_{100}$ counts, mean scores, category membership, and post-hoc union counts. No extra generations or GPU evaluations are used.

Table 9. **Task taxonomy and coverage.** Each of the 258 tasks belongs to exactly one of 16 categories, which preserve the 108 source-domain labels. Exact and numeric count output-comparison rules, not floating-point use. Examples describe computations, not measured bottlenecks.

Category	Tasks	Domains	Exact	Numeric	Representative computations
Signals & vision	32	15	26	6	Optical flow, Hough voting, spectral transforms, image reconstruction
Data & systems	29	9	27	2	Error-correcting codes, hash tables, database queries, packet processing
Earth & space	25	9	12	13	PDE solvers, seismic imaging, astronomical signal searches
Number theory & bits	21	9	21	0	Modular arithmetic, integer searches, exact bit-level computation
Graphs & trees	20	5	20	0	Flow and cuts, shortest paths, community detection, tree queries
Physics & materials	18	11	13	5	Particle dynamics, lattice models, transport, material microstructure
Text & retrieval	18	8	18	0	String matching, compression, tokenization, ranked retrieval
Algebra & optimization	16	5	7	9	Sparse solvers, factorizations, tensor operations, numerical optimization
ML systems	14	2	13	1	Model inference, quantized optimization, expert routing, clustering
Simulation & finance	13	10	13	0	Event simulation, financial risk, auctions, stateful market models
DP & combinatorics	12	6	12	0	Subset recurrences, counting, packing, constrained search
Biology	11	4	10	1	Sequence analysis, phylogenetic likelihoods, RNA folding, proteomics
Geometry & graphics	9	5	9	0	Geometric predicates, mesh operations, collision tests, rendering
Robotics & logistics	8	4	8	0	Motion planning, map fusion, routing, packing, scheduling
Chemistry	7	3	5	2	Molecular fingerprints, docking, electronic-structure computation
EDA & compilers	5	3	5	0	Circuit timing and routing, fault simulation, program analysis
Total	258	108	219	39	

Table 10. Mean clipped log-speed score $\bar{S}$ within each category, including zero-score unsuccessful and unresolved outcomes. Bold marks the largest observed mean in a row, without a significance claim. Categories contain 5–32 tasks. Task-weighted column means reproduce Table 1; equal weighting of rows gives the group means in Table 7.

Category	Tasks	Opus 5.5	Astra	Sonnet 5.5	Haiku 5.5	Sol	Luna	Kimi K2.6
Signals & vision	32	**1.158**	0.921	0.934	0.649	0.377	0.111	0.150
Data & systems	29	**1.043**	0.944	0.819	0.710	0.570	0.150	0.018
Earth & space	25	**0.753**	0.556	0.503	0.410	0.244	0.086	0.009
Number theory & bits	21	**1.189**	1.088	0.792	0.564	0.554	0.137	0.078
Graphs & trees	20	**1.563**	1.476	1.158	0.785	0.718	0.251	0.147
Physics & materials	18	**0.863**	0.794	0.727	0.483	0.229	0.093	0.048
Text & retrieval	18	**1.099**	0.908	0.841	0.700	0.480	0.116	0.233
Algebra & optimization	16	**1.029**	0.849	0.762	0.603	0.410	0.075	0.157
ML systems	14	**1.100**	0.741	0.730	0.322	0.177	0.013	0.055
Simulation & finance	13	**0.805**	0.645	0.559	0.461	0.362	0.105	0.081
DP & combinatorics	12	**1.338**	0.985	0.996	0.688	0.314	0.151	0.013
Biology	11	**1.036**	1.016	1.004	0.621	0.419	0.287	0.051
Geometry & graphics	9	**0.981**	0.664	0.712	0.281	0.213	0.098	0.110
Robotics & logistics	8	**1.217**	0.998	0.706	0.657	0.288	0.024	0.086
Chemistry	7	**1.338**	0.959	1.051	0.809	0.344	0.456	0.197
EDA & compilers	5	0.968	**0.974**	0.847	0.367	0.428	0.088	0.186

Table 11. Complete mapping of the 108 source domains into 16 reporting categories. Task-level domain labels are listed in Table 13.

Category	Source domains
Signals & vision	audio-processing, camera-isp, computer-vision, event-vision, fft-spectral, image-coding, image-signal, medical-imaging, photogrammetry, signal-processing, signal-restoration, speech, video-codecs, video-coding, video-restoration
Data & systems	coding-theory, data-movement, data-structures, databases, networking, query-optimization, storage, storage-systems, systems-performance
Earth & space	astronomy, float-scientific, geoscience, geospatial, pde-numerics, remote-sensing, reservoir-simulation, seismology, stencil-pde
Number theory & bits	bitwise-search, certified-numerics, cryptography, dense-exact-algebra, integer-search, number-theory, numerics, random-number-generation, side-channel-analysis
Graphs & trees	graph-algorithms, graph-analytics, graph-dynamic-programming, graphs, trees
Physics & materials	biophysics, lattice-qcd, materials-science, mesoscale-dynamics, molecular-dynamics, monte-carlo, nuclear-physics, particles-nbody, quantum-simulation, rarefied-gas-dynamics, statistical-physics
Text & retrieval	compression, information-retrieval, nlp-information-retrieval, strings, text, text-processing, time-series, unicode-text-processing
Algebra & optimization	dense-linalg-tensor, numerical-optimization, sparse-direct-solvers, sparse-linalg, sparse-numerical-linear-algebra
ML systems	clustering, machine-learning-systems
Simulation & finance	advertising-systems, computational-finance, computer-architecture, economics, energy-systems, finance, financial-networks, simulators, stochastic-simulation, systems-simulation
DP & combinatorics	combinatorial-counting, combinatorial-enumeration, combinatorial-game-theory, combinatorial-optimization, constraint-programming, dynamic-programming
Biology	bioinformatics, genomics, metagenomics, proteomics
Geometry & graphics	animation, computational-geometry, geometry, graphics, volume-visualization
Robotics & logistics	logistics-optimization, operations-research, production-scheduling, robotics
Chemistry	cheminformatics, computational-chemistry, quantum-chemistry
EDA & compilers	compiler-analysis, compilers, electronic-design-automation

E. Task Construction and Interface

The authoring specification calls for domain-inspired computational variants, multicore OpenMP CPU code, deterministic observable behavior, compact seeded inputs, and a correct direct CUDA port with optimization opportunities. It favors integer or fixed-point state and explicit reduction order; numeric comparisons use task-specific tolerances. These are construction goals, not independently certified properties of every task. In particular, bounds on CPU duration, generator scaling, expert-reference quality, and inner-loop novelty require task-level auditing rather than inference from a successful compile or GPU check.

The historical H100 acceptance record covers all 258 task identities in an earlier release, version 1.2.0. It is distinct from the seven version-1.5.4 agent jobs. The latter supply the paper's performance results and their own per-trial verification provenance. Source versioning and the difference between historical acceptance and current evaluation should be retained when reconstructing the benchmark.

E.1. Example Submission Contract

The following summarizes the interface for `astro-dedisp`; it is not a complete specification of the task. The program performs an incoherent-dedispersion pipeline with channel flags, integer dispersion delays, and a boxcar search. Public instructions state input ranges and edge cases such as one channel, absent boxcar widths, zero dispersion, and no injected pulses. The CPU implementation resolves the exact indexing, arithmetic, and selection rules.

The agent writes `/app/gpu/main.cu`, accepting one input-file argument and emitting the required standard output. It can inspect the CPU source and run public examples. The target is an H100, but the generation container provides neither CUDA compilation nor GPU execution. Only the final source is transferred to the verifier. Task instructions request that the main computation execute on the GPU; the implemented activity check enforces only the weaker condition described in Appendix F.

F. Verifier Details and Integrity Boundaries

Numerical comparisons require finite values and matching token structure, with absolute and relative tolerances from task metadata. Candidate execution uses task-specific timeouts. A valid program must also reproduce the required output during warm-up and timed executions. Reference and candidate measurements are paired at one task-specific scale, and their order alternates across the eight rounds. The timer's boundary is `main()` after context initialization, not the whole process and not an isolated CUDA kernel.

Table 12. Implemented integrity checks and the limits of their interpretation. No systematic adversarial success rate is available.

Concern	Current check and remaining limitation
Reusing outputs or persistent state	Separate candidate execution, fresh read-only working directory, hidden reference binary, and output checks during timing. These measures have not been validated against a comprehensive attack suite.
Hiding work from the timer	In-process timing measures `main()` and checks the output. Asynchronous-work accounting and resistance to timer tampering remain unvalidated.
Specializing to tested inputs	Fresh seeds and correctness edge cases reduce fixed-input reuse; finite tests do not prove correctness across the supported input distribution.
Leaving useful work on the CPU	CUPTI requires a completed kernel with positive GPU duration. No enforced minimum fraction of useful GPU work is established.
Using linked numerical libraries	Compilation permits the CUDA runtime and header-only toolkit libraries, without additional linked numerical libraries.
Crossing the artifact boundary	Only `main.cu` is transferred to the separate verifier; this is not a proof of sandbox security.
Extreme reference speedups	Ratios of at least $1000\times$ are flagged. One result awaits manual audit; reference strength and bandwidth-based lower bounds are not certified.

A complete attack evaluation is not available for the analyzed runs. In particular, output summaries do not rule out collisions or untested special cases; fresh seeds do not prove broad input coverage; and a positive kernel trace does not determine how much useful work took place on the device. The flagged Astra result needs source-level inspection and independent replay. These limits apply even when aggregate rankings survive conservative score changes.

A future oracle audit could inject independently witnessed faults into verified ports and measure checker detection, following Du et al. (2026). If generated tests augment the suite, their expected outputs should be validated independently: incorrect candidate code can bias test generation (Huang et al., 2026b), and execution-grounded oracle refinement such as Nexus (Huang et al., 2025) offers a complementary construction method. None of these additional audits is claimed for the present snapshot.

G. Reproducibility and AI Use

The analysis is derived from stored trial outcomes, rather than newly generated measurements. Coding agents assisted task construction and produced the evaluated CUDA submissions; an AI assistant also assisted literature review, manuscript drafting and revision, and analysis code. The authors are responsible for the final manuscript and its claims. Automated checks do not substitute for independent human review of task validity or experimental claims.

The local artifact contains a 1,806-row trial snapshot, hashes of result and detail records, immutable task identifiers, domain metadata, dated cost-estimation assumptions, and scripts that regenerate aggregate statistics, paired resampling analyses, tables, and figures. The dataset digest and configuration identifiers appear in Appendix A. This manuscript does not claim that a public release is already available. An execution-reproducible release also requires submitted sources, accessible task packages, trajectories, hidden-input seeds, grader versions, and a reviewed license; the aggregate snapshot alone does not supply all of these artifacts.

H. Complete Task Inventory

Table 13 lists task names, domain labels, comparison modes, and source lengths. The H100 checkmark records historical reference acceptance, rather than a separate current-version oracle replay for every analyzed agent submission. The construction statistics and inventory are generated from task sources and acceptance records; the analysis freezes current task references and domain/group assignments separately.

Table 13. All 258 task identities and construction metadata.

Task	Domain	Compare	CPU LoC	Ref. LoC	H100 pass
ad-budget-pacing-auction	advertising-systems	exact	78	88	✓
algebra-gf2-big-elim	dense-exact-algebra	exact	286	447	✓
anim-cloth-ccd	animation	exact	338	361	✓
astro-clean-msclean	astronomy	exact	362	455	✓
astro-dedisp	astronomy	exact	209	352	✓
astro-transit-bls	astronomy	exact	121	145	✓
audio-landmark-fingerprint	audio-processing	exact	265	308	✓
bio-colinear-chain	bioinformatics	exact	120	154	✓
bio-debruijn-prune	bioinformatics	exact	158	175	✓
bio-felsenstein-gamma	bioinformatics	numeric	339	428	✓
bio-ld-popcount	bioinformatics	exact	205	276	✓
bio-metagenome-em	metagenomics	exact	94	100	✓
bio-minimizer-count	bioinformatics	exact	123	139	✓
bio-phmm-forward	bioinformatics	exact	141	189	✓
bio-rna-mccaskill	bioinformatics	exact	388	520	✓
bits-arx-collide	bitwise-search	exact	106	160	✓
bits-grundy-mex	combinatorial-game-theory	exact	210	304	✓
bits-lfsr-jump	random-number-generation	exact	195	207	✓
bits-nlfsr-lincomp	bitwise-search	exact	142	153	✓
bits-sbox-spectra	bitwise-search	exact	143	188	✓
chem-bonded-ff-energy	computational-chemistry	exact	370	475	✓
chem-dft-grid-xc	computational-chemistry	exact	129	177	✓
chem-dock-pose-mc	computational-chemistry	exact	376	445	✓
chem-fock-coulomb-s	quantum-chemistry	numeric	317	398	✓
chem-morgan-tanimoto	cheminformatics	exact	343	461	✓
chem-rmsd-kabsch-cluster	computational-chemistry	exact	232	281	✓
chem-stiff-reaction-ode	computational-chemistry	numeric	352	370	✓
cluster-kmedoids-pam	clustering	exact	255	379	✓
code-bcjr-turbo	coding-theory	exact	235	312	✓
code-ldpc-minsum-qc	coding-theory	exact	267	343	✓
code-polar-sc	coding-theory	exact	206	269	✓
code-rs-cauchy-gf256	coding-theory	exact	275	403	✓
code-viterbi-k9	coding-theory	exact	211	268	✓
code-vlc-selfsync	compression	exact	333	573	✓
codec-intdct-rle	video-coding	exact	311	362	✓
comp-andersen-pts	compiler-analysis	exact	340	392	✓
comp-dataflow-bitvec	compilers	exact	430	625	✓
compress-ibwt-mtf	compression	exact	181	305	✓
compress-lzw-decode	compression	exact	242	387	✓
compress-pkgmerge-huffman	compression	exact	108	115	✓
compress-rans-interleave	compression	exact	223	321	✓
count-tiling-mixed-holes	combinatorial-counting	exact	128	181	✓
count-walks-step-follow	combinatorial-counting	exact	112	170	✓
crypto-aes-avalanche	cryptography	exact	241	313	✓
crypto-bignum-montgomery	dense-exact-algebra	exact	400	498	✓
crypto-chacha-biasscan	cryptography	exact	235	311	✓
crypto-cpa-trace	side-channel-analysis	exact	145	198	✓
crypto-sha256-merkle	cryptography	exact	241	357	✓
db-bitslice-range	databases	exact	263	340	✓
db-colscan-bitpack	databases	exact	395	602	✓
db-cuckoo-filter	databases	exact	108	115	✓
db-hashjoin-skew	databases	exact	225	295	✓
db-posting-intersect	databases	exact	119	129	✓

Task	Domain	Compare	CPU LoC	Ref. LoC	H100 pass
db-roaring-boolean	databases	exact	125	142	✓
db-window-rank	databases	exact	211	186	✓
dense-int8-mlp-requant	dense-linalg-tensor	exact	149	246	✓
dense-ldlt-gp-loo	dense-linalg-tensor	numeric	167	198	✓
dense-lu-mixed-refine	dense-linalg-tensor	numeric	211	380	✓
dense-set-colmass-attn	dense-linalg-tensor	numeric	135	281	✓
dense-tucker-gated-fibre	dense-linalg-tensor	numeric	191	320	✓
dioph-pow5-pow7-square-sieve	integer-search	exact	139	189	✓
dioph-quartic-3y4-kw4	integer-search	exact	147	168	✓
dioph-theta-mixed-powers	integer-search	exact	112	176	✓
dp-align-twopiece	dynamic-programming	exact	155	223	✓
dp-grid-momentum	dynamic-programming	exact	163	189	✓
dp-interval-split	dynamic-programming	exact	101	157	✓
dp-knap-concave	dynamic-programming	exact	103	151	✓
dp-subset-sched	dynamic-programming	exact	155	210	✓
dp-treewidth-count	graph-dynamic-programming	exact	287	319	✓
econ-lob-multiasset-abm	economics	exact	418	538	✓
eda-fault-sim-parallel	electronic-design-automation	exact	99	112	✓
eda-maze-route-congestion	electronic-design-automation	exact	116	131	✓
eda-sta-slack	electronic-design-automation	exact	325	537	✓
eig-lobpcg-block	sparse-linalg	numeric	388	335	✓
energy-storage-wear-dp	energy-systems	exact	85	96	✓
energy-two-phase-flow	reservoir-simulation	exact	119	152	✓
enum-polyomino-pack	combinatorial-enumeration	exact	186	271	✓
fem-p1-assembly	pde-numerics	exact	334	470	✓
fft-conv3d-ringfilter-energy	fft-spectral	numeric	310	382	✓
fft-ntt-product-tree	fft-spectral	exact	223	354	✓
fft-spectral-burgers-forced	fft-spectral	numeric	266	372	✓
fft-stft-mixedradix-features	fft-spectral	numeric	274	402	✓
fin-credit-copula-var	computational-finance	exact	269	345	✓
fin-mc-bundled-american	finance	exact	255	311	✓
fin-network-clearing	financial-networks	exact	82	111	✓
fpsci-annealed-em-diag	float-scientific	numeric	173	266	✓
fpsci-chemomech-chain-rk4	float-scientific	numeric	185	192	✓
fpsci-rolling-joint-filter	float-scientific	numeric	180	221	✓
fpsci-tensor-diffusion-3d	float-scientific	numeric	185	224	✓
fv-euler-weno5	pde-numerics	numeric	218	248	✓
gen-pileup-genotype	genomics	exact	369	483	✓
gen-poa-polish	genomics	exact	300	458	✓
geo-adaptive-quadtree	computational-geometry	exact	325	439	✓
geo-dem-hydro	geospatial	exact	201	373	✓
geo-hull-batch	computational-geometry	exact	272	434	✓
geo-mcubes-int	computational-geometry	exact	423	501	✓
geo-pip-join	computational-geometry	exact	260	365	✓
geo-psha-hazard	geoscience	numeric	100	112	✓
geom-qem-decimate	geometry	exact	316	447	✓
gfx-texcomp-encode	graphics	exact	381	425	✓
gfx-vector-path-raster	graphics	exact	385	599	✓
graph-apsp-blocked-fw	graph-analytics	exact	218	274	✓
graph-auction-lap	graph-algorithms	exact	242	343	✓
graph-bipartite-hk	graph-algorithms	exact	191	216	✓
graph-bottleneck-spectrum	graph-analytics	exact	202	320	✓
graph-bridges-articulation	graph-analytics	exact	232	311	✓
graph-closeness-msbfs	graph-analytics	exact	164	197	✓
graph-johnson-apsp	graph-analytics	exact	179	274	✓
graph-kcore-peel	graph-analytics	exact	358	408	✓
graph-khop-coverage	graph-analytics	exact	154	204	✓
graph-labelprop-penalized	graph-analytics	exact	153	241	✓
graph-louvain-modularity	graphs	exact	378	554	✓
graph-maxflow-batch	graph-algorithms	exact	384	541	✓
graph-mincut-stoer-wagner	graph-algorithms	exact	146	271	✓
graph-node2vec-walks	graph-analytics	exact	300	432	✓
graph-scc-directed	graph-analytics	exact	308	600	✓

Task	Domain	Compare	CPU LoC	Ref. LoC	H100 pass
graph-stress-centrality	graph-analytics	exact	188	227	✓
graph-timewindow-arrival	graph-analytics	exact	186	270	✓
graph-weighted-triangles	graph-analytics	exact	196	289	✓
image-boxfeat-nms	image-signal	exact	215	259	✓
image-ccl-graded	image-signal	exact	230	267	✓
image-demosaic-bayer	camera-isp	exact	293	382	✓
image-dwt-lifting	image-coding	exact	331	393	✓
image-edt-ridge	image-signal	exact	206	264	✓
image-rank-open	image-signal	exact	169	192	✓
image-sig-cic-peaks	image-signal	exact	159	170	✓
image-tile-equalize	image-signal	exact	180	226	✓
ir-bm25-blockmax-topk	nlp-information-retrieval	exact	347	437	✓
ir-graph-ann-search	information-retrieval	exact	252	373	✓
krylov-gmres-restart	sparse-linalg	numeric	242	277	✓
mat-ebsd-grain-texture	materials-science	exact	458	577	✓
mat-grain-sparse-mpf	materials-science	exact	301	368	✓
math-certified-root-isolation	certified-numerics	exact	91	100	✓
mc-clock-diag-spin	monte-carlo	exact	163	216	✓
md-lincs-constraints	molecular-dynamics	numeric	725	941	✓
med-ct-backproject	medical-imaging	exact	207	256	✓
med-pet-listmode	medical-imaging	exact	124	170	✓
ml-beam-search-int	machine-learning-systems	exact	244	354	✓
ml-block-quant-optimizer	machine-learning-systems	exact	102	142	✓
ml-conv-resblock-fp16	machine-learning-systems	numeric	273	319	✓
ml-embedbag-int8	machine-learning-systems	exact	204	291	✓
ml-forest-infer-quant	machine-learning-systems	exact	326	386	✓
ml-gbdt-hist-split	machine-learning-systems	exact	356	451	✓
ml-gnn-sage-layer	machine-learning-systems	exact	223	358	✓
ml-gru-seq-persistent	machine-learning-systems	exact	223	319	✓
ml-isotonic-pava	machine-learning-systems	exact	247	455	✓
ml-lda-gibbs-sync	machine-learning-systems	exact	256	365	✓
ml-moe-dispatch	machine-learning-systems	exact	238	399	✓
ml-pq-adc-search	machine-learning-systems	exact	255	393	✓
ml-snn-lif-sparse	machine-learning-systems	exact	207	328	✓
net-bgp-pathvector	networking	exact	292	425	✓
net-lpm-fib	networking	exact	250	336	✓
net-tcp-reassembly	networking	exact	99	158	✓
nlp-kneser-ney-lm	nlp-information-retrieval	exact	352	406	✓
num-exact-gemv-superacc	numerics	exact	274	316	✓
numfn-cfrac-period	number-theory	exact	80	87	✓
numfn-gfn-abundance	integer-search	exact	111	190	✓
numfn-oddsigma-aliquot	integer-search	exact	212	245	✓
numfn-popbase-euler	integer-search	exact	86	117	✓
numfn-primecount-lucy	integer-search	exact	202	312	✓
numfn-tonelli-batch	number-theory	exact	157	134	✓
opt-2opt-tours	combinatorial-optimization	exact	178	282	✓
opt-admm-tv-batch	signal-restoration	numeric	282	429	✓
opt-arc-consistency	constraint-programming	exact	292	330	✓
opt-binpack-bestfit	logistics-optimization	exact	276	301	✓
opt-bnb-mdknap	combinatorial-optimization	exact	285	393	✓
opt-flowshop-neh	production-scheduling	exact	212	342	✓
or-grid-n2-contingency	operations-research	exact	312	341	✓
or-vrptw-savings-ls	operations-research	exact	366	501	✓
ot-sinkhorn-logdomain	numerical-optimization	numeric	213	227	✓
particles-fixedpoint-md	particles-nbody	exact	172	270	✓
particles-knn-mutual-rings	particles-nbody	exact	221	313	✓
particles-pm-crossspecies	particles-nbody	exact	170	242	✓
particles-screened-softcore	particles-nbody	numeric	133	166	✓
pde-adi-heat-tridiag	pde-numerics	numeric	376	497	✓
pde-adjoint-4dvar	pde-numerics	numeric	402	598	✓
pde-dg-advection	pde-numerics	numeric	339	443	✓
pde-eikonal-fastsweep	pde-numerics	numeric	162	179	✓
pde-multigrid-varcoef	pde-numerics	numeric	436	519	✓

Task	Domain	Compare	CPU LoC	Ref. LoC	H100 pass
pde-swe-hllc-muscl	pde-numerics	numeric	374	500	✓
phys-cellular-potts-cpm	biophysics	exact	181	222	✓
phys-dpd-pairwise	mesoscale-dynamics	exact	175	252	✓
phys-dslash-su3	lattice-qcd	numeric	238	334	✓
phys-dsmc-collisions	rarefied-gas-dynamics	exact	182	232	✓
phys-eam-metal-md	molecular-dynamics	exact	198	277	✓
phys-mc-neutron-xs	nuclear-physics	exact	332	460	✓
phys-percolation-newmanziff	statistical-physics	exact	120	183	✓
phys-qsim-statevec	quantum-simulation	numeric	327	409	✓
phys-sn-transport-sweep	nuclear-physics	numeric	146	221	✓
phys-sw-3body-md	molecular-dynamics	exact	202	274	✓
prot-psm-fragindex	proteomics	exact	484	751	✓
remote-hyperspectral-unmix	remote-sensing	exact	135	173	✓
robo-ogm-ray-fuse	robotics	exact	288	338	✓
robo-orca-fleet	robotics	exact	306	343	✓
robo-prm-armcheck	robotics	exact	372	630	✓
robo-viewshed-siting	robotics	exact	194	278	✓
seis-pick-locate	seismology	exact	343	461	✓
seis-rtm-reversible	seismology	exact	364	469	✓
sem-poisson-matfree	pde-numerics	numeric	429	509	✓
sig-kalman-track-batch	signal-processing	numeric	404	489	✓
sig-polyphase-resample	signal-processing	exact	234	293	✓
sim-int-gillespie-batch	simulators	exact	163	194	✓
sim-predprey-grass-lattice	simulators	exact	233	287	✓
sim-ring-brakelight-traffic	simulators	exact	155	194	✓
sim-tiny-vm-batch	simulators	exact	177	207	✓
smc-pfilter-intweights	stochastic-simulation	exact	177	296	✓
sparse-adaptive-approx-inverse	sparse-numerical-linear-algebra	exact	109	120	✓
sparse-hubauth-intnorm	sparse-linalg	exact	173	210	✓
sparse-jpcolor-gs	sparse-linalg	exact	205	266	✓
sparse-mttkrp-cp	sparse-linalg	exact	241	317	✓
sparse-pcg-neumann2	sparse-linalg	numeric	220	268	✓
sparse-spgemm-pathcount	sparse-linalg	exact	139	302	✓
sparse-supernodal-cholesky-etree	sparse-direct-solvers	numeric	307	474	✓
sparse-symgs-modp	sparse-linalg	exact	178	190	✓
speech-kws-dtw-cascade	speech	exact	282	446	✓
speech-wfst-token-pass	speech	exact	265	461	✓
stencil-four-species	stencil-pde	exact	166	195	✓
stencil-lattice-gas	stencil-pde	exact	168	236	✓
stencil-skew-topple	stencil-pde	exact	154	282	✓
store-dedup-cdc	storage-systems	exact	229	497	✓
store-object-gc	storage	exact	86	155	✓
str-fmindex-backsearch	strings	exact	192	239	✓
str-landau-kdiff	strings	exact	137	172	✓
str-lz77-greedy	compression	exact	258	440	✓
str-myers-bitparallel	strings	exact	277	337	✓
str-nfa-monoid-scan	strings	exact	115	152	✓
stream-dfa-multimatch	data-movement	exact	227	253	✓
struct-bvh-blob-xray	data-structures	numeric	258	378	✓
struct-celf-multicover	data-structures	exact	156	231	✓
struct-cuckoo-build	data-structures	exact	243	318	✓
struct-hash-ttl-ledger	data-structures	exact	274	373	✓
struct-lsm-merge	data-structures	exact	272	515	✓
struct-segtree-lazy-batch	data-structures	exact	299	449	✓
struct-suffix-repeats	data-structures	exact	260	336	✓
struct-wavelet-quantile	data-structures	exact	98	111	✓
sys-cachesim-partition	computer-architecture	exact	373	469	✓
sys-joinorder-dp	query-optimization	exact	88	95	✓
sys-mva-queueing	systems-performance	numeric	235	348	✓
sys-packet-classify-bv	networking	exact	240	282	✓
sys-pdes-timewindow	systems-simulation	exact	174	342	✓
text-bpe-tokenize	text	exact	373	620	✓
text-jsonl-log-agg	text-processing	exact	144	151	✓

Task	Domain	Compare	CPU LoC	Ref. LoC	H100 pass
text-minhash-lsh	text	exact	319	604	✓
text-unicode-grapheme-normalize	unicode-text-processing	exact	143	153	✓
tree-lca-pathagg	trees	exact	211	376	✓
ts-matrix-profile	time-series	exact	208	235	✓
video-inloop-deblock	video-codecs	exact	79	104	✓
video-temporal-warp-denoise	video-restoration	exact	84	112	✓
vis-blockmatch-me	video-coding	exact	265	346	✓
vis-bundle-adjust-schur	photogrammetry	numeric	396	524	✓
vis-event-motion	event-vision	exact	76	93	✓
vis-fast-ring	computer-vision	exact	355	371	✓
vis-hough-lines	computer-vision	exact	384	479	✓
vis-pyramid-lk-flow	computer-vision	exact	349	333	✓
vis-ransac-homog	computer-vision	exact	270	330	✓
vis-sgm-census	computer-vision	exact	313	364	✓
vis-slic-superpixel	computer-vision	exact	293	372	✓
vis-sparse-volume-render	volume-visualization	exact	70	93	✓