STRUCTURE BEFORE TUNING: DECOUPLING STRUCTURAL SEARCH AND LOCAL OPTIMIZATION FOR LLM-GENERATED GPU KERNELS

Anonymous authors
Paper under double-blind review

ABSTRACT

GPU kernel efficiency directly affects training and inference latency, throughput, and deployment cost. LLM-based agents generate and revise kernels using compilation and runtime feedback. In analyzed DSA sparse-attention runs, we observe *structural lock-in*: most search effort refines one implementation structure while alternatives receive little exploration. We introduce STRUTUNE, a two-stage framework that gives alternative logical structures independent implementation and optimization opportunities. Stage I generates explicit structural plans, reviews their validity, and scores their optimization potential without generating or executing code. Stage II realizes the top-ranked valid plans and tunes each under a fixed structure contract, with its own search history and local budget. We evaluate STRUTUNE on 29 selected operators comprising 722 workload records from FlashInfer-Bench and SOL-ExecBench on NVIDIA A800 and H100 GPUs, achieving 100% correctness on the evaluated tasks on both GPUs. Geometric-mean speedups are $3.53\times$ and $1.81\times$ over the FlashInfer expert reference, and $6.51\times$ and $5.77\times$ over the SOL-ExecBench eager reference, on A800 and H100, respectively.

1 INTRODUCTION

GPU kernels execute the dominant computations in modern AI systems, so their efficiency directly affects training cost, inference latency, and throughput. Performance depends on how work is partitioned, where reductions occur, and how computation and memory movement map to the GPU. FlashAttention reduces attention's memory traffic through IO-aware tiling (Dao et al., 2022), while PagedAttention reduces KV-cache fragmentation and duplication in LLM serving (Kwon et al., 2023). Expert-written kernels require hardware-specific development; GPU domain-specific languages such as Triton expose tiled programs and execution mappings at a higher level than CUDA (Tillet et al., 2019).

Automated kernel optimization spans compiler-defined search spaces (Chen et al., 2018a; Zheng et al., 2020), specialized generation models (Guo et al., 2026; Liu et al., 2026), and agents that revise kernels using execution feedback (Ouyang et al., 2025; Zhu et al., 2026; Cao et al., 2026; Du et al., 2026; Luo et al., 2026; Dong et al., 2026; Kernel Design Agents Contributors, 2026). Our analysis of Codex, KDA, and K-Search runs on DSA sparse attention reveals a concentration of search effort that we call *structural lock-in*: most local refinement preserves an incumbent's high-level structure while alternative structures remain underexplored (Figure 2).

For example, all 32 complete performance evaluations in the Codex DSA trace retain the same two-kernel organization, refining implementation details while a different skeleton achieves higher performance under controlled local tuning (Section 3.2).

The distinction is between *structural decisions*, which determine the computation strategy, kernel decomposition, and parallelization axes, and *local decisions*, which tune tiles, layouts, and schedules within that structure. Changing the structure opens a different implementation space, to which previous local optimizations may not transfer.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

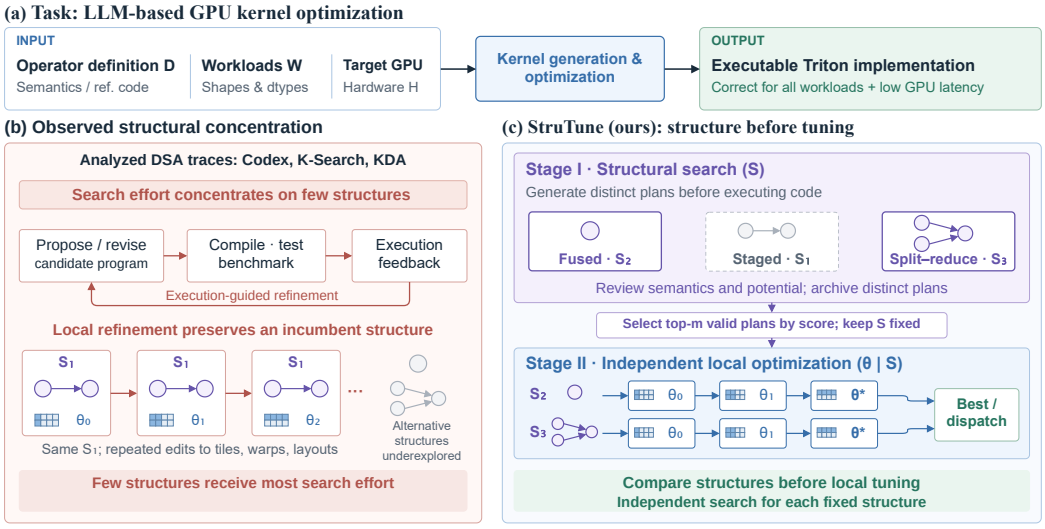

Figure 1: **Task, observed search behavior, and proposed workflow. (a)** Generating a GPU kernel implementation. **(b)** Schematic of structural concentration in the analyzed DSA traces. **(c)** StruTune decouples structural search from independent local optimization. S denotes the structure and θ its local configuration; the two branches are illustrative.

Prior work already separates high-level planning from code implementation (Zhu et al., 2026; Cao et al., 2026). The remaining question is how alternative structures receive independent implementation and refinement opportunities when execution feedback favors an early incumbent. StruTune (Structure Before Tuning) addresses this through two stages (Figure 1(c)). Stage I generates structural plans and reviews their validity and optimization potential before code generation. Stage II implements the top-ranked valid plans as Triton code and optimizes each under a fixed structure contract, with its own execution history and local budget.

Our contributions are:

- We quantify structural coverage across 29 tasks on A800 using agent-reviewed logs reconciled with final outcomes, analyze DSA effort concentration, and compare extracted structures under a shared local optimizer and budget.
- We propose StruTune, which archives and compares explicit structural plans before execution, then gives selected structures independent local histories and budgets under fixed contracts.
- We evaluate StruTune on 29 selected operators across two benchmarks and two GPUs. It solves every evaluated task on both GPUs and improves over K-Search by 42.75% on A800 and 26.11% on H100 on the corresponding SOL-ExecBench subsets solved by K-Search. Table 2 gives the complete aggregate results.

2 Related Work

Prior work approaches kernel optimization through four related research directions: GPU programming abstractions, learned kernel generators, execution-guided search, and agent-based optimization workflows.

2.1 GPU Kernel DSLs and Compiler Optimization

Halide separates algorithms from schedules to explore parallelism, locality, and recomputation (Ragan-Kelley et al., 2013). Triton represents kernels as tile-level tensor programs (Tillet et al., 2019), while TileLang separates dataflow from scheduling choices such as layout and pipelining (Wang et al., 2026). TVM provides an end-to-end optimizing compiler (Chen et al., 2018a);

AutoTVM learns cost models to guide tensor-program search (Chen et al., 2018b), Ansor constructs hierarchical tensor-program spaces (Zheng et al., 2020), and ROLLER generates programs from hardware-aligned tiles (Zhu et al., 2022). These systems make schedules and transformations explicit within compiler-defined spaces.

2.2 Learned Kernel Generation

DRTriton combines synthetic data generation with curriculum reinforcement learning (Guo et al., 2026). Dr. Kernel develops multi-turn reinforcement learning with profiling-based rewards (Liu et al., 2026), while Makora fine-tunes a frontier model in an execution-backed environment (Tehrani et al., 2026). CUDA Agent combines data synthesis, a verification-and-profiling environment, and agentic reinforcement learning for CUDA kernel generation (Dai et al., 2026). These methods improve the model's ability to generate correct and efficient kernels, but their effectiveness remains dependent on training data, reward design, and generalization beyond the training distribution.

2.3 Execution-Guided Kernel Search

QiMeng-Kernel separates high-level performance reasoning from low-level edits (Zhu et al., 2026), while AdaExplore combines failure-driven adaptation with diversity-preserving search (Du et al., 2026). K-Search separates algorithmic planning from program instantiation and updates a co-evolving world model using execution feedback (Cao et al., 2026). StruTune instead compares an archive of explicit structural plans before execution and allocates separate local histories and budgets to selected structures under fixed contracts.

2.4 Agent-Based Kernel Optimization

Agent systems organize kernel generation, debugging, profiling, and optimization through specialized roles and persistent workflows. AKO, STARK, and Kernel Design Agents coordinate different stages of the optimization process (Luo et al., 2026; Dong et al., 2026; Kernel Design Agents Contributors, 2026), while Astra and CudaForge incorporate specialized agents and hardware feedback (Wei et al., 2025; Zhang et al., 2025). These systems support long-running, tool-augmented optimization workflows, but repeated iterations can incur substantial token costs.

3 Problem Definition and Observational Analysis

3.1 Problem Definition

Given an operator definition D, a target GPU H, and a workload set $\mathcal{W} = \{w_j\}_{j=1}^{n}$, GPU kernel optimization seeks an implementation of D that is correct for every workload and executes efficiently on H. Each workload specifies an input configuration, including tensor shapes and data types. Let $\mathcal{K}(D, H)$ denote the space of candidate implementations. For an implementation $k \in \mathcal{K}(D, H)$, let $T(k, w; H)$ denote its execution latency on workload w, and let $C(k, w; D) = 1$ indicate that the implementation compiles and its outputs satisfy the required numerical tolerances.

We measure aggregate performance using geometric-mean speedup (Ouyang et al., 2025). Let $T_{\text{ref}}(w_j; H)$ denote the latency of the reference implementation on workload w_j. The performance of implementation k and the corresponding optimization objective are

$$F(k) = \left(\prod_{j=1}^{n} \frac{T_{\text{ref}}(w_j; H)}{T(k, w_j; H)} \right)^{1/n}, \qquad (1)$$
$$k^* \in \arg\max_{k \in \mathcal{K}(D,H)} F(k) \quad \text{s.t.} \quad C(k, w_j; D) = 1 \quad \forall j \in \{1, \ldots, n\}.$$

To distinguish structural choices from implementation tuning within this optimization problem, we factor an implementation into a structure S and a local configuration θ:

$$k = \mathcal{R}(S, \theta), \qquad S = (A, G, P), \qquad (2)$$

where A is the computation strategy, such as fused, staged, or split-reduce execution; G is the logical kernel graph that specifies each logical stage's computation, inputs, outputs, and dependencies rather than physical launch count; and P is the semantic parallelization plan that specifies the parallel axes and output ownership of each kernel.

The local configuration determines how a fixed structure is implemented on the target GPU:

$$\theta = (I, M_{\text{mem}}, Q, B, R), \qquad \Theta(S, H) = \{\theta \mid \Gamma(\mathcal{R}(S, \theta)) = S\}, \tag{3}$$

where I denotes the intra-kernel implementation algorithm, M_{mem} the memory realization, Q the compute primitive, B concrete parameters such as tile sizes and split counts, and R scheduling parameters such as `num_stages` and `num_warps`. The abstraction Γ maps an implementation to its logical structure, so $\Theta(S, H)$ contains exactly the local configurations that preserve A, G, and P. Appendix A.1 gives the classification rules; Appendix F.2 illustrates the distinction with recorded plans and contract checks.

3.2 Observation and Insight

We first examine structural coverage across 29 tasks on A800 for Claude Code, Codex, Kernel Design Agent (KDA) (Kernel Design Agents Contributors, 2026), and K-Search (Cao et al., 2026). Agents review their optimization logs to identify distinct structures $S = (A, G, P)$ reported to pass correctness validation. We reconcile these counts with the final evaluation outcomes, assigning zero to unsolved tasks and computing coverage percentages over solved tasks. For each method, over 80% of solved tasks have one or two reported structures (Table 1); on FlashInfer-Bench, none exceeds two. Appendices D.1 and D.2 give the annotation protocol and complete counts, respectively.

Table 1: Outcome-reconciled structural coverage on A800 across 29 tasks. Unsolved tasks are assigned zero; the last column is the fraction of solved tasks with one or two structures.

Method	**Valid structures**				**1–2 / solved (%)**
	0	**1**	**2**	**$\geq$ 3**	
Claude Code	1	24	4	0	100.00
Codex	0	24	2	3	89.66
KDA	5	11	10	3	87.50
K-Search	9	11	7	2	90.00

To examine how optimization effort is distributed, we analyze the DSA token traces in Figure 2(a). Codex and K-Search spend all traced tokens, 1.57M and 1.62M respectively, on a single family. KDA's token trace contains two families but assigns 12.66M of its 13.83M tokens (91.54%) to one, leaving 1.17M (8.46%) for the other. In the Codex trace, all 32 complete performance evaluations retain the same two-kernel organization while tuning implementation details. These traces reveal structure lock-in through concentrated refinement of an incumbent structure. Appendix D.3 gives the detailed token allocation.

We next test whether this effort is concentrated on a suboptimal structure. From the methods' final implementations, we extract structural plans and collect the traced local actions into a shared library. Codex then implements and optimizes each fixed plan using the same base model, hardware, evaluator, action library, and budget; the StruTune plan includes its fixed workload-dispatch rule. Over ten local rounds, the StruTune plan reaches $37.47\times$ speedup, 30.88% above the strongest alternative, KDA (Figure 2(b)). Thus, the analyzed DSA runs concentrate effort on structures that remain inferior to an available alternative under the shared local-tuning budget. Appendix D.4 reports all four replay results.

These observations motivate comparing alternative computation strategies, kernel graphs, and parallelization plans before committing to local optimization, then refining promising structures independently.

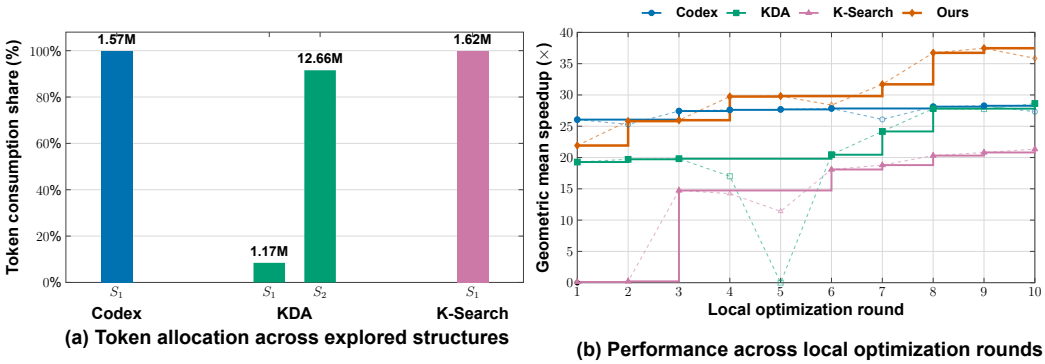

Figure 2: Structural-search traces and controlled local optimization. (a) Token shares within each method; labels give absolute counts. Structure identifiers are method-specific: matching labels do not imply equivalent plans. (b) Codex optimizes each extracted structure with the same budget and action library; legend entries identify the structure's source method. Solid lines show the best speedup through each round; dashed lines show individual-round measurements.

4 METHOD

STRUTUNE first generates and scores structural plans without executing code, then implements and tunes the top-ranked valid plans independently. Figure 3 summarizes the workflow; Algorithms 1 and 2 in Appendix A specify the two stages.

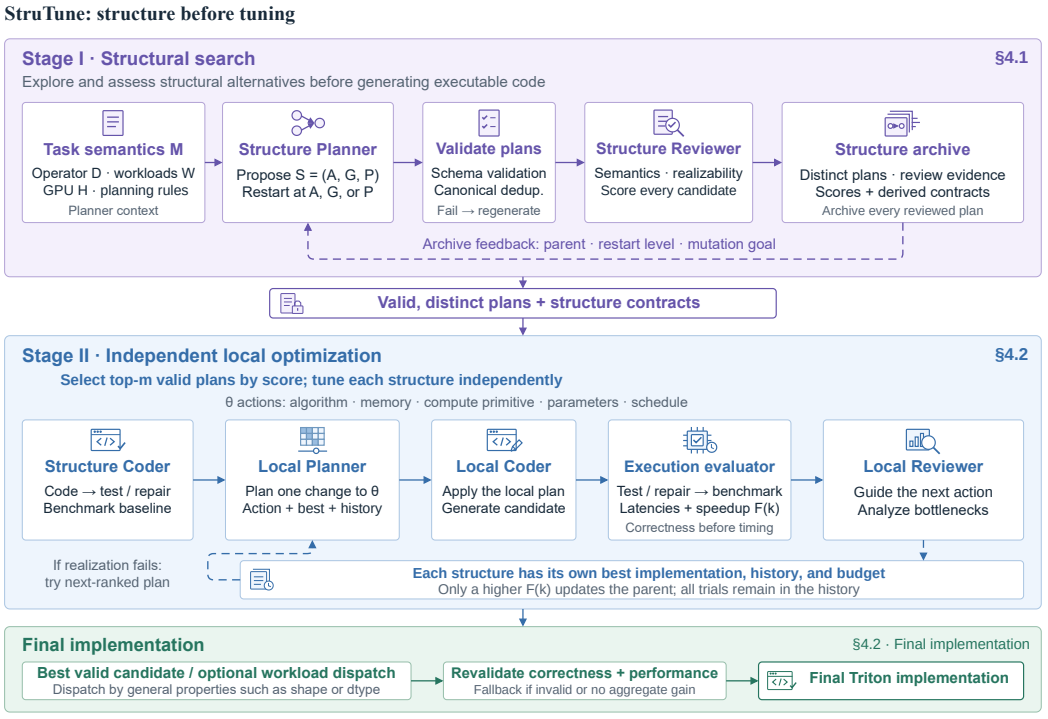

Figure 3: Overview of STRUTUNE. Archived reviews guide structural proposals before execution. Each selected structure then receives an independent local search, with its current best valid implementation as the sole parent.

4.1 STAGE I: STRUCTURAL SEARCH

Task context and initial plans. Semantic analysis summarizes the operator definition D, workloads $\mathcal{W}$, and hardware H into a task context M. This context captures computation dependencies, reduction and output-ownership constraints, reuse opportunities, and workload regimes. The Structure Planner uses M and structural knowledge to propose plans $S = (A, G, P)$ in the schema of Appendix A.2. The first proposal establishes a correctness-first plan; the second explores a distinct computation strategy. Appendix B.2.1 details the planner context.

Validation and scoring. Each proposal undergoes schema validation and canonical deduplication before review. The Structure Reviewer scores every resulting candidate, assessing computation quality, exposed parallelism, reuse opportunity, state efficiency, and coordination efficiency. Its validity judgment filters semantic and realizability violations, while the weighted score gives a heuristic priority for local optimization. The archive stores each plan, its validity judgment, the weighted score, and supporting review evidence. The five scores use a 0–4 scale with weights 0.25, 0.25, 0.20, 0.15, and 0.15, respectively; the rubric appears in Appendix B.2.2.

The review separates legal execution from optimization potential. Concrete contradictions in logical ownership, semantic state, or coordination invalidate a plan; a low potential score does not. Evidence comes from the plan, task semantics, and archive rather than predicted latency or unspecified local choices. The controller aggregates the five dimension scores for ranking. This separation allows a valid but less promising plan to remain in the archive without treating its heuristic score as a correctness test.

Review-guided proposals. For subsequent candidates, the Parent Selector uses archived plans and reviews to choose a structural parent, the earliest component to reconsider, and a mutation goal. The Structure Planner then revises the parent as

$$S' = \begin{cases} (A', G', P'), & \text{restart from the computation strategy,} \\ (A, G', P'), & \text{restart from the logical kernel graph,} \\ (A, G, P'), & \text{restart from the parallelization plan.} \end{cases} \tag{4}$$

Preserving earlier components while regenerating the selected component and its dependents lets the planner target a diagnosed structural weakness. If no valid parent is available, it proposes a new plan autonomously. Each new candidate passes through the same validation and scoring procedure.

The budget B_{S} counts distinct, schema-valid candidates reviewed, including those judged structurally invalid. Here, distinct means canonically non-duplicate. Schema failures and duplicates are retried within a per-candidate limit. Each structurally valid plan receives a structure contract that fixes its logical responsibilities, dependencies, semantic intermediates, and output ownership. This contract constrains subsequent code generation and local optimization.

4.2 STAGE II: LOCAL OPTIMIZATION

Top-ranked selection and realization. After structural search, valid plans are ranked by their weighted reviewer scores, and the top-m plans are selected for independent local optimization. Equal scores are ordered by candidate index. Realization failures are replaced according to this fixed Stage I ranking until m structures succeed or the valid archive is exhausted. The Structure Coder realizes each selected plan as Triton code under its contract. Contract and correctness checks, followed by bounded repair when needed, establish a valid initial implementation. Each successfully realized structure starts with a benchmarked baseline and receives up to B_{L} local iterations. The main experiments use $B_{\mathrm{S}} = 5$, $m = 2$, and $B_{\mathrm{L}} = 10$, with up to two repair rounds per implementation attempt; Appendix C.4 gives the retry limits.

Local candidate generation. Each iteration selects an action using the fixed structure's trial history and reviewer feedback. Actions can change the intra-kernel algorithm, memory realization, compute primitives, concrete parameters, or execution schedule while preserving S. The Local Planner specifies one change to θ, and the Local Coder applies it to the current best valid implementation k_S^*. Appendix A.5 defines the action catalog; Appendix B.3 specifies the agent interfaces.

The local plan records the changed fields and downstream choices to reconsider. For example, an accepted layout change can reopen compute-primitive and scheduling actions because their previous settings were chosen for a different layout. Reconsideration uses the remaining local budget, not an additional search phase. The contract preserves logical responsibilities rather than physical launch count: a layout-conversion helper can be local, whereas introducing a new split-and-merge dependency requires a different structural plan.

Evaluation and best-parent update. Each candidate undergoes contract and correctness validation, with bounded repair, before benchmarking on $\mathcal{W}$. It replaces k_S^* only if its aggregate speedup $F(k)$ is higher; otherwise the incumbent remains the sole parent. The Local Reviewer analyzes performance to guide subsequent actions, while the history retains all trials and diagnostics. Each structure has its own incumbent, history, and local budget, so execution feedback from one trajectory does not redirect another.

Final implementation. The implementation with the highest $F(k)$ across the independent trajectories is the fallback output. When their best implementations have complementary workload-level advantages, the system may combine them through dispatch rules based on general workload properties such as tensor dimensions and data types. Rule proposals require at least a 5% workload-relative gain separated by a semantic input feature. The Dispatch Reviewer rejects rules based on workload identifiers, sample indices, or complete-shape whitelists. The coder combines the accepted branches into one implementation with the fallback retained. This composition is evaluated over the complete workload suite, rather than estimating its benefit by selecting the fastest recorded candidate independently for each workload. The combined implementation is retained only if it passes correctness validation and improves aggregate speedup over the fallback; otherwise the fallback is returned.

5 EXPERIMENTS

We evaluate whether separating structural search from local optimization improves GPU kernel generation. The experiments cover overall performance, token consumption, component ablations, and performance coverage as more structures are generated. Section 3.2 provides the DSA trace and replay analysis.

5.1 EXPERIMENTAL SETUP

Benchmarks and platforms. FlashInfer-Bench (Xing et al., 2026) contains production-oriented LLM inference workloads, including specialized attention and normalization operators, while SOL-ExecBench (Lin et al., 2026) covers a broader range of standalone and fused tensor operators at different difficulty levels. The FlashInfer library (Ye et al., 2025) provides the expert reference used to normalize FlashInfer-Bench speedups. We construct fixed evaluation subsets according to three principles: covering the major operator families and execution patterns in each benchmark, spanning diverse workload shapes and difficulty levels, and selecting tasks without using their final performance as a criterion. All methods use the same task definitions, workload configurations, and evaluation protocol. All generated kernels are tested on NVIDIA A800 and H100 GPUs; Appendix C gives the task list and evaluation protocol.

Comparison baselines. We compare STRUTUNE against baselines from four categories. Direct generation uses GPT-5.6-Sol and Claude Opus 4.8, while specialized kernel generation is represented by Dr. Kernel (Liu et al., 2026). General-purpose coding agents include Codex and Claude Code. Kernel Design Agent (Kernel Design Agents Contributors, 2026) and K-Search (Cao et al., 2026) represent kernel-specific agent optimization and structured search, respectively. Each baseline is evaluated through its official interface using its recommended test-time configuration whenever available. Appendix C.3 summarizes the baseline configurations. STRUTUNE generates five structural candidates, selects two for independent local optimization, and uses ten local rounds with at most two repair rounds per implementation attempt. Workload dispatch is enabled in the main evaluation.

Table 2: Overall results. Success is correctness rate (%); Speedup is the geometric mean over solved tasks, normalized by the hardware-specific FlashInfer expert speedup (N/A uses a factor of one), or relative to SOL-ExecBench eager. Parentheses give our relative improvement on exactly the corresponding method's solved subset.

Method	A800				H100			
	FlashInfer-Bench		SOL-ExecBench		FlashInfer-Bench		SOL-ExecBench	
	Success	Speedup	Success	Speedup	Success	Speedup	Success	Speedup
LLM-direct								
GPT-5.6-Sol	100.00	2.33× (+51.92%)	73.68	4.09× (+143.22%)	100.00	1.05× (+72.40%)	68.42	4.05× (+103.28%)
Claude-Opus-4.8	100.00	1.84× (+91.57%)	68.42	5.00× (+141.32%)	100.00	0.76× (+137.17%)	47.37	2.55× (+150.40%)
Official Harness								
Codex	100.00	2.53× (+39.46%)	100.00	4.06× (+60.06%)	100.00	1.59× (+13.24%)	89.47	3.22× (+83.48%)
Claude Code	100.00	2.68× (+31.99%)	94.74	4.60× (+56.94%)	100.00	1.10× (+63.85%)	94.74	3.41× (+85.08%)
Kernel Agent								
Kernel Design Agent	100.00	1.72× (+104.98%)	73.68	4.02× (+96.53%)	90.00	1.02× (+35.62%)	47.37	4.49× (+66.99%)
Evolve								
K-Search	90.00	2.08× (+101.48%)	57.89	8.41× (+42.75%)	100.00	1.27× (+42.25%)	52.63	8.57× (+26.11%)
RL								
Dr. Kernel	30.00	0.13× (+833.97%)	21.05	0.90× (+1082.37%)	20.00	0.24× (+138.62%)	21.05	2.42× (+285.69%)
Ours								
STRUTUNE	**100.00**	**3.53×**	**100.00**	**6.51×**	**100.00**	**1.81×**	**100.00**	**5.77×**

Table 3: Component ablations on A800. Success is correctness rate (%); Speedup is the geometric mean over solved tasks, using the same reference normalization as Table 2.

#	Method	Components		FlashInfer		SOL-ExecBench	
		Structural Search	Local Optimization	Success	Speedup	Success	Speedup
1	Structure-only	✓	–	100.00	2.41×	100.00	0.97×
2	Local-only	–	✓	100.00	1.37×	63.16	2.25×
3	**STRUTUNE**	✓	✓	**100.00**	**3.53×**	**100.00**	**6.51×**

5.2 OVERALL RESULTS

Table 2 reports correctness rate and geometric-mean speedup by hardware and benchmark in separate *Success* and *Speedup* columns.

STRUTUNE uses 11.65M and 11.66M tokens across the 29 tasks on A800 and H100, respectively. Compared with the four agent baselines, it reduces total token consumption by 77.73–95.00% on A800 and 87.94–89.74% on H100, while achieving 100% correctness on both task sets. The reduction holds for each benchmark–GPU group, not just the combined totals. These comparisons use the method-specific evaluation configurations in Appendix C.3.

5.3 ABLATION STUDY

Using the same base model on A800, *local-only* allows ten local optimization rounds without structural search or dispatch. *Structure-only* generates and realizes five structures, returning the best valid implementation without local tuning or dispatch. The full method selects $m = 2$ structures for independent ten-round local trajectories and then evaluates workload dispatch. Appendix E.3 gives the task-level results.

All three variants achieve 100% correctness on FlashInfer-Bench, where the full method reaches $3.53\times$ speedup compared with $2.41\times$ for structure-only and $1.37\times$ for local-only. On SOL-ExecBench, structure-only and the full method solve all 19 tasks, at $0.97\times$ and $6.51\times$, respectively; local-only solves 12 of 19 tasks (63.16%), with $2.25\times$ speedup over that smaller solved subset. The full method therefore improves SOL-ExecBench correctness by 36.84 percentage points. These

comparisons assess the complete configurations, including dispatch in the full method, rather than isolating a dispatch-free local-tuning effect.

Performance coverage across generated structures. Figure 4 follows five structures in generation order, each evaluated with ten local optimization rounds. For each task, we track the best performance among the first n structures, normalized by the best among all five, and take the geometric mean across tasks in each benchmark group. Additional structures improve this coverage, with the FlashInfer curve reaching its best observed performance by the third structure and SOL-ExecBench benefiting from later candidates. This cumulative analysis measures the alternatives exposed by structural search, rather than independently rerunning the full pipeline at each budget. Appendix E.3.1 gives the normalization.

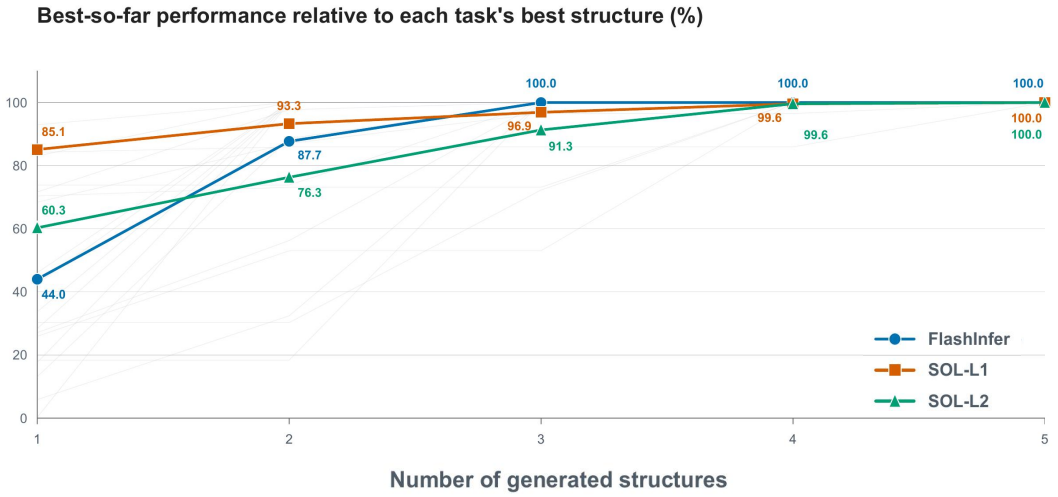

Figure 4: Cumulative performance coverage on A800 as structures are added in generation order. Curves are geometric means of task-level performance normalized by each task's best among five locally optimized structures; the endpoint is therefore 100% by construction.

6 Conclusion

StruTune combines explicit structural search with independent local refinement, achieving 100% correctness across the evaluated tasks on A800 and H100 and speedups over the benchmark references. Ablations show complementary component benefits; the DSA study demonstrates different performance across selected structures under a shared, finite optimization budget.

Limitations and future work. The evaluation covers 29 selected operators on two GPUs. Broader operator suites, architectures, and base models are needed to assess how well the separation between structural and local decisions transfers. The cross-task structure counts come from agent-reviewed logs reconciled with final outcomes; independent code-level annotation would strengthen this evidence. The controlled replay examines DSA, so additional operators would help establish when structural concentration has the largest performance cost.

Although evaluated on NVIDIA GPUs, StruTune is hardware-agnostic in its separation of structural search from local optimization. Future work should validate it on more hardware platforms with appropriate backends and evaluators. Structural evaluation currently relies on an LLM's subjective judgments under predefined rubrics, without sufficient empirical support. We plan to explore more structures and collect correctness and performance outcomes, then investigate reinforcement learning to train a structure evaluator grounded in these data.

AI USE STATEMENT

AI USE STATEMENT

Generative AI tools were used in two roles: as part of the studied system and as an aid in preparing this manuscript. In the evaluated pipeline, large language models generate GPU kernel plans, kernel code, review feedback, and repair suggestions; these AI-generated components are the object of our study, and all correctness and performance claims are derived exclusively from preserved evaluator outputs such as compiled binaries, runtime measurements, and pass/fail records, never from model self-reports or unverified generations. Generative AI tools were also used to assist with literature search, to help draft portions of this manuscript, and to proofread for typos and grammatical issues, with any AI-suggested citations checked against the original sources before use. All AI-generated content, whether produced in the evaluated pipeline or during manuscript preparation, underwent human secondary review and verification, and the authors take full responsibility for the final text, released artifacts, and all conclusions.

ETHICS STATEMENT

This work studies large language models for generating high-performance GPU kernels. The evaluated pipeline involves no human subjects, no personally identifiable information, and no data collected from human participants. All benchmarks and workloads used in our experiments are publicly available, and no new dataset of human-derived content is released. We recognize that automated code generation can produce incorrect or unsafe implementations; accordingly, all correctness and performance claims in this paper are based on preserved evaluator outputs rather than model self-reports, and our results are reported on a fixed set of selected operators rather than as general guarantees. We do not foresee direct harmful applications of this work, but we acknowledge that techniques for automating kernel generation could lower the cost of producing inefficient or non-compliant code, and we therefore emphasize verification and human oversight as necessary components of any deployment. The authors declare no conflicts of interest and no sponsorship that influenced the design, execution, or reporting of this study.

REPRODUCIBILITY STATEMENT

We describe the structural representation, agent interfaces, and experimental protocol in Appendices A, B, and C. Trace analysis, detailed results, and recorded optimization cases are provided in Appendices D, E, and F. The evaluated operators, GPU models, evaluation harness, and the preserved evaluator outputs underlying all reported correctness and performance numbers are specified in Appendix C. We release anonymized source code and the prompts used by each agent as supplementary materials. Because the pipeline depends on large language models whose outputs may vary across versions and sampling settings, we report the exact model identifiers and decoding configurations used, and we note that exact numerical reproduction may require matching these settings. All artifacts needed to reproduce the reported results are referenced above rather than restated here.

REFERENCES

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-Search: LLM kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018a.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, volume 31, 2018b.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang,

Xin Liu, and Hao Zhou. CUDA agent: Large-scale agentic RL for high-performance CUDA kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. STARK: Strategic team of agents for refining kernels. In *International Conference on Learning Representations*, 2026.

Weihua Du, Jingming Zhuo, Yixin Dong, Andre Wang He, Weiwei Sun, Zeyu Zheng, Manupa Karunaratne, Ivan Fox, Tim Dettmers, Tianqi Chen, Yiming Yang, and Sean Welleck. AdaExplore: Failure-driven adaptation and diversity-preserving search for efficient kernel generation. *arXiv preprint arXiv:2604.16625*, 2026.

Siqi Guo, Ming Lin, and Tianbao Yang. DRTriton: Large-scale synthetic data driven reinforcement learning for triton kernel generation. *arXiv preprint arXiv:2603.21465*, 2026.

Kernel Design Agents Contributors. Kernel design agents. GitHub repository, 2026. URL `https://github.com/NVlabs/kda`. Accessed 2026-09-16.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *ACM Symposium on Operating Systems Principles*, 2023.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, Dheeraj Peri, Ouye Xie, Aditya Kane, Moshe Maor, Michael Behar, Triston Cao, Rishabh Mehta, Vartika Singh, Vikram Sharma Mailthody, Terry Chen, Zihao Ye, Hanfeng Chen, Tianqi Chen, Vinod Grover, Wei Chen, Wei Liu, Eric Chung, Luis Ceze, Roger Bringmann, Cyril Zeller, Michael Lightstone, Christos Kozyrakis, and Humphrey Shi. SOL-ExecBench: Speed-of-light benchmarking for real-world GPU kernels against hardware limits. *arXiv preprint arXiv:2603.19173*, 2026.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for triton kernel generations. *arXiv preprint arXiv:2602.05885*, 2026.

Mao Luo, Hongbin Li, Feng Lin, Hanling Yi, and Zhe Huang. Agentic kernel optimization: Generating state-of-the-art GPU kernels without hand-written CUDA. *arXiv preprint arXiv:2608.14560*, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025. URL `https://proceedings.mlr.press/v267/ouyang25a.html`.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

Ali Tehrani, Yahya Emara, Essam Wissam, Wojciech Paluch, Waleed Atallah, Lukasz Dudziak, and Mohamed S. Abdelfattah. Fine-tuning GPT-5 for GPU kernel generation. *arXiv preprint arXiv:2602.11000*, 2026.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973.

Lei Wang, Yu Cheng, Yining Shi, Zhiwen Mo, Zhengju Tang, Wenhao Xie, Tong Wu, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. TileLang: Bridge programmability and performance in modern neural kernels. In *International Conference on Learning Representations*, 2026.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for GPU kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. FlashInfer-Bench: Building the virtuous cycle for AI-driven LLM systems. *arXiv preprint arXiv:2601.00227*, 2026.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. CudaForge: An agent framework with hardware feedback for CUDA kernel optimization. *arXiv preprint arXiv:2511.01884*, 2025.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation*, 2020.

Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation*, pp. 233–248, 2022.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, Yanjun Wu, Chen Zhao, and Ling Li. QiMeng-Kernel: Macro-thinking micro-coding paradigm for LLM-based high-performance GPU kernel generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2026.

A Structural Representation and Search Algorithms

This appendix specifies the structural search space and the two search algorithms. Agent instructions appear in Appendix B, and experimental settings appear in Appendix C.

A.1 Structure Definition and Equivalence

The representation $S = (A, G, P)$ in Section 3.1 separates the computation strategy A, logical kernel graph G, and semantic parallelization plan P from the local configuration θ. The graph records logical responsibilities and dependencies rather than a list of physical kernel launches. Physical kernel count may change when those logical tasks and dependencies remain intact.

Changes to tile sizes, warp counts, or pipeline stages preserve the structure when they leave A, G, and P unchanged. Introducing a new decomposition, changing which intermediate state crosses a logical kernel boundary, or changing semantic output ownership constitutes a structural change. Canonical comparison removes differences due only to names, descriptions, axis ordering, or equivalent grid expressions. Appendix F.2 illustrates the distinction with recorded plans and contract checks.

Table 4: Decision rules for distinguishing structural changes from local implementation changes.

Change	Classification
Tiles, warps, or pipeline stages	Local when the computation strategy, logical dependencies, and semantic ownership are preserved.
Existing split/reduction parameters	Local when the split axis, logical producer–consumer and merge relations, and output ownership remain fixed.
Logical state or graph	Structural when a change introduces or removes cross-stage semantic state, or alters logical dependencies or ownership.
Physical kernel count	Count alone is insufficient: classify by the logical responsibilities and dependencies realized by the launches.

A.2 Structural Plans and Contracts

Structure Plan v7 records A, G, and P as `strategy`, `kernels`, and `grid`. The strategy describes reductions, merging, routing, or ordered computation. Logical kernels record responsibilities, inputs, outputs, intermediate-state components, and consumers; each kernel has one semantic-grid entry. Every intermediate has one producer and explicit consumers, and every final output has unique ownership. Metadata records the schema version, candidate mode, parent, and restart phase.

The *structure contract* carries the computation strategy, logical responsibilities, intermediate-state definitions, dependencies, parallelization axes, and output assignments into Stage II. Deterministic validation derives these fields from the plan, checks the schema and mutation-prefix preservation, and rejects canonical duplicates. Valid reviewed plans retain the derived contract in their archive record. The *implementation contract* returned with locally modified code repeats the protected logical fields and separately records the implementation family, physical kernels, and concrete parameters. A mismatch in a protected field rejects the local candidate.

The supplied implementation checks the coder-declared contract against six fields derived from the plan: `base_strategy`, `logical_kernels`, `semantic_intermediates`, `semantic_grid`, `launch_dependencies`, and `output_ownership`. A field mismatch rejects the candidate before execution and records the offending field. Code repair must pass the same contract check before correctness is reevaluated.

The mapping Γ defines the logical abstraction; the implemented gate validates its declared contract, not a static proof of the generated code's semantics. The Extractor analyzes task definitions and workloads, rather than reconstructing a contract from Triton source. Runtime validation checks outputs, shapes, types, and numerical errors over the workload suite. Establishing that source code implements every declared ownership and dependency obligation still requires code inspection. Appendix F.2 gives executable acceptance and rejection checks and a structural contrast from the recorded plans.

A.3 STRUCTURAL SEARCH ALGORITHM

Algorithm 1 uses plans and semantic reviews without code generation or execution feedback. The first candidate uses the initial mode, the second uses the autonomous-alternative mode, and later candidates use review-guided mutation when a valid parent exists. The budget B_{S} counts distinct, schema-valid candidates reviewed, including those judged structurally invalid. Schema failures and duplicates are retried within the current candidate. Retry limits are listed in Appendix C.4.

Algorithm 1 Stage I: Structural search

Require: Operator D, workloads $\mathcal{W}$, GPU H, planning/review knowledge, budget B_{S}
Ensure: Archive $\mathcal{A}_{\mathrm{valid}}$ of distinct valid plans, reviews, and contracts
$M \leftarrow \textsc{SemanticAnalysis}(D, \mathcal{W}, H)$; $\mathcal{A} \leftarrow \emptyset$
while $|\mathcal{A}| < B_{\mathrm{S}}$ **do**
  Choose generation mode by candidate index as described above
  if the mode is parent revision and a valid parent is available **then**
    $(S_{\mathrm{parent}}, \mathrm{level}, \mathrm{goal}) \leftarrow \textsc{ParentSelector}(M, \mathcal{A})$
    $S \leftarrow \textsc{StructurePlanner}(M, S_{\mathrm{parent}}, \mathrm{level}, \mathrm{goal})$
    Preserve components before the selected level as in Eq. (4)
  else
    $S \leftarrow \textsc{StructurePlanner}(M, \mathrm{initial\ or\ autonomous\ context})$
  end if
  if S fails schema validation or duplicates an archived canonical structure **then**
    Regenerate S within the per-candidate retry limit; fail the run if the limit is exhausted
  end if
  review $\leftarrow \textsc{StructureReviewer}(S, M, \mathcal{A})$
  Archive S and its review, including the validity judgment and weighted score
  if the review marks S structurally valid **then**
    Attach $\textsc{DeriveContract}(S)$ to its archive record
  end if
end while
return $\mathcal{A}_{\mathrm{valid}}$, the valid records in $\mathcal{A}$

A.4 INDEPENDENT LOCAL OPTIMIZATION ALGORITHM

Algorithm 2 ranks valid plans by reviewer score and attempts realization in that order until m distinct structures succeed or the archive is exhausted. Each realized structure receives up to B_{L} local iterations. Its current best valid implementation is the sole parent; the history retains all trials. The candidate set $\mathcal{C}$ collects the final incumbent from each trajectory.

REALIZEANDVALIDATE invokes the Structure Coder and bounded repair. VALIDATEANDREPAIR checks the fixed structure contract and correctness over the workload suite. Each helper returns an implementation and diagnostics, or failure when its limit is exhausted. Benchmarking follows successful validation. A strictly higher $F(k)$ replaces the incumbent; ties retain the existing parent.

Algorithm 2 Stage II: Independent local optimization

Require: D, $\mathcal{W}$, H, archive $\mathcal{A}_{\text{valid}}$, target m, local budget B_{L}
Ensure: Best observed valid implementation, optionally with validated dispatch, or failure

$\mathcal{U} \leftarrow \mathcal{A}_{\text{valid}}$ sorted by descending weighted reviewer score
$\mathcal{C} \leftarrow \emptyset$; roots $\leftarrow 0$
while roots $< m$ and $\mathcal{U} \neq \emptyset$ **do**
  Remove the highest-ranked remaining plan S from $\mathcal{U}$
  $(k_0, \text{diagnostics}) \leftarrow \textsc{RealizeAndValidate}(S, D, \mathcal{W}, H)$
  if realization fails **then**
    Record the failure; **continue**
  end if
  Benchmark k_0 to obtain $F(k_0)$; initialize $k_S^* \leftarrow k_0$ and history $\mathcal{H}_S$
  roots $\leftarrow$ roots $+ 1$
  for step $= 1, \ldots, B_{\text{L}}$ **do**
    parent $\leftarrow k_S^*$; select one local action using $\mathcal{H}_S$
    plan $\leftarrow \textsc{LocalPlanner}(S, \text{parent}, \text{action}, \mathcal{H}_S)$
    $k \leftarrow \textsc{LocalCoder}(\text{plan}, \text{parent}, \text{contract}(S))$
    $(k, \text{diagnostics}) \leftarrow \textsc{ValidateAndRepair}(k, S, D, \mathcal{W}, H)$
    if contract or correctness validation fails **then**
      Append the failed trial and diagnostics to $\mathcal{H}_S$; **continue**
    end if
    Benchmark k to obtain $F(k)$
    review $\leftarrow \textsc{LocalReviewer}(k, \text{parent}, \text{action}, \mathcal{H}_S)$
    Append the trial, diagnostics, measurements, and review to $\mathcal{H}_S$
    if $F(k) > F(k_S^*)$ **then**
      $k_S^* \leftarrow k$
    end if
  end for
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{k_S^*\}$
end while
if $\mathcal{C} = \emptyset$ **then**
  return failure
end if
$k_{\text{best}} \leftarrow \arg\max_{k \in \mathcal{C}} F(k)$
if dispatch is enabled and workload-specific advantages are observed **then**
  Construct workload-based dispatch from valid candidates in $\mathcal{C}$
  if dispatch passes correctness and improves aggregate speedup over k_{best} **then**
    return the validated dispatch implementation
  end if
end if
return k_{best}

A.5 Local Actions and Dispatch

Action scope. Each local action changes the implementation configuration θ while preserving the selected structure $S = (A, G, P)$. Table 5 specifies the eight action classes. Their scope extends beyond parameter tuning: an action may change an intra-kernel algorithm or add a physical layout-conversion kernel, provided the computation strategy, logical responsibilities, semantic intermediates, dependencies, grid axes, and output ownership remain fixed. For example, changing the number of partitions along an existing split axis is local; introducing a new split-and-merge decomposition requires structural search.

Table 5: Local action catalog. Each round implements one concrete change within one action class. All actions preserve the structure contract and public interface and must pass numerical correctness validation.

Action	Implementation change	Scope and constraints
$L0$	**Implementation family.** Choose an algorithmic realization, such as direct computation, implicit GEMM, FFT, or a compatible persistent implementation.	A family switch is local only when it realizes the same logical computation and ownership. Changes to the strategy or cross-kernel semantic state belong to structural search.
$L1$	**Layout and physical realization.** Change internal tensor layouts or introduce layout-conversion and implementation-helper kernels.	Physical launches may change, but logical responsibilities and dependencies remain fixed. External input/output semantics are preserved.
$L2$	**Tensor Core adaptation.** Adapt operand layouts, `tl.dot` or MMA paths, accumulators, and precision.	Adaptation stays within the existing computation and must satisfy the evaluator's numerical tolerances.
$L3$	**Grid and reduction parameters.** Tune `num_split`, `q_split`, or chunk counts along axes already exposed by the structure.	The axis set, producer–consumer and merge relations, and output ownership stay fixed.
$L4$	**Kernel schedule.** Tune tile sizes, warp counts, pipeline-stage counts, register budgets, and traversal order.	Scheduling changes how each logical task executes, rather than which semantic task it owns.
$L5$	**Memory and pipeline.** Change vectorization, data reuse, prefetching, shared-memory use, or software pipelining.	Movement and buffering choices preserve semantic intermediates and their dependencies. Unlike $L4$, this action changes the memory/pipeline realization rather than its scheduling parameters.
$L6$	**Masks and specialization.** Refine masks, boundary handling, and static specialization.	Specialized paths retain the full required semantic coverage, including boundary cases.
$L7$	**Host and buffer overhead.** Reduce allocation, zero-fill, buffer-lifetime, and host-synchronization overhead.	Required initialization, execution dependencies, and the public `run` interface are preserved.

Planning one change. Local Plan v2 records the parent and structure identifiers, selected action, concrete change, changed fields, downstream actions to reopen, and a structure-preservation declaration. The Local Coder applies that change to the current best valid implementation without bundling unrelated actions. In a recorded GDN plan, an $L1$ action transposes a row-major $[\texttt{BLOCK_V}, 128]$ state tile into an internal key-major $[128, \texttt{BLOCK_V}]$ layout and transposes it back on store. The plan keeps the external layout, launch grid, single physical kernel, $\texttt{BLOCK_V} = 16$, warp count, and numerical operations unchanged. This makes the intended mutation and its protected context explicit.

Action selection and retuning. The local knowledge prioritizes implementation family and layout before compute primitives, grid/reduction parameters, and scheduling, while using bottleneck feedback to select applicable actions. For example, launch overhead motivates $L7$, memory pressure motivates $L1$ or $L5$, and compute pressure motivates $L2$. An accepted change can make earlier downstream choices stale. The catalog therefore reopens $L1$–$L7$ after $L0$, $L2$–$L7$ after $L1$, $L3$–$L6$ after $L2$, $L4$–$L5$ after $L3$, $L5$–$L6$ after $L4$, and $L6$ after $L5$; $L6$ and $L7$ reopen no actions. Reopening makes an action eligible for reconsideration within the remaining local budget, rather than applying it automatically. Rejected and inapplicable actions remain in the history. The reviewer guides subsequent actions, while measured aggregate speedup alone determines whether a valid candidate replaces the incumbent, as specified in Algorithm 2.

Dispatch across trajectories. Dispatch is separate from $L0$–$L7$: it compares the best valid implementation from each trajectory at workload level. A deterministic procedure proposes rules when a

semantic input feature separates candidates with at least 5% relative gain. Rules use general workload properties and retain the best aggregate candidate as a fallback. The composed implementation is re-evaluated on the complete workload suite and retained only if it passes correctness and improves aggregate speedup. Appendix B.4 gives the review and coding instructions.

B Agent Prompts and Knowledge Sources

The numbered listings summarize each role using system instructions, inputs, requirements, and outputs. They are simplified templates reconstructed from the interface specifications in this manuscript, not verbatim experimental prompts. Angle brackets denote runtime artifacts; displayed field names describe their contents rather than exact serialization keys. Shared knowledge is defined once and supplied to the relevant roles. Search control is specified in Appendix A, and model and retry settings are specified in Appendix C.4.

The system/input/output headings organize the instructions; the client sends the assembled prompt as one user message. Deterministic ranking, validation, benchmarking, and incumbent updates are controller operations, not additional LLM prompts.

B.1 Semantic Analysis and Shared Knowledge

Semantic analysis. The extractor receives the operator definition, every indexed workload configuration, and a target-hardware description. It identifies the operator DAG, reductions and mergeability, ordered dependencies, repeated or dynamic destinations, routing requirements, producer–consumer relations, and necessary semantic state. It cannot propose a structure or invent latency, alignment, stride, or input-distribution assumptions. Its textual analysis has five sections: semantics, workload summary, workload regimes, implementation constraints, and correctness risks. A second output assigns workloads to small, middle, or large regimes and identifies representative focus workloads.

```
[SYSTEM]
Analyze operator semantics and workloads without proposing a
    structure.

[INPUT]
operator: <definition and reference semantics>
workloads: <all indexed input configurations>
hardware: <target GPU description>

[REQUIREMENTS]
- Identify the DAG, reductions, dependencies, and output ownership.
- Identify routing, reuse, and required intermediate state.
- Do not invent timing, alignment, stride, or input assumptions.

[OUTPUT]
analysis: <semantics; workloads; regimes; constraints; risks>
regimes: <each workload assigned to small, middle, or large>
focus: <representative workloads>
```

Listing 1: Simplified Semantic Analysis prompt.

Shared knowledge. Structural planning uses rules for reductions, routing, scans, ownership, state lifetime, and fusion, together with category-specific checks for attention, GEMM, convolution, reduction, scan, and scatter/routing. Structural review uses validity gates and five scoring rubrics. Local optimization uses the $L0$–$L7$ catalog and its invalidation graph. Coders share Triton execution, memory, numerical-stability, reduction, Tensor Core, interface, and validation rules.

B.2 Structural Planning and Review

B.2.1 Structure Planner

Input and instructions. The planner receives the semantic digest, planning knowledge, sanity checks, archive summary, and generation mode. The initial mode produces a correctness-first plan. The autonomous-alternative mode restarts from the computation strategy and changes a genuine structural decision. The evidence-mutation mode also receives a parent digest, review, and Parent Selector decision; it preserves components preceding the restart point and regenerates the suffix. Before returning a plan, the planner checks operand shapes, reductions, reuse, dependencies, coverage, and output ownership for a representative program.

Output. The planner returns one tagged Structure Plan v7 JSON object with the fields specified in Appendix A.2. Schema, mutation-prefix, and canonical-duplicate checks are deterministic. Failed validation triggers regeneration within the configured retry limit.

```
[SYSTEM]
You are the Structure Planner. Propose a structural plan, not code.

[INPUT]
context: <semantic digest, knowledge, and sanity checks>
archive: <plans and reviews>
mode: <initial | autonomous-alternative | evidence-mutation>
revision: <parent, restart phase, and goal when applicable>

[REQUIREMENTS]
- Initial: produce a correctness-first plan.
- Alternative: change a genuine structural decision.
- Mutation: preserve the prefix and regenerate the selected suffix.
- Check shapes, reductions, dependencies, coverage, and ownership.

[OUTPUT]
<one tagged Structure Plan v7 JSON object; schema specified in the
    accompanying text>
```

Listing 2: Simplified Structure Planner prompt.

```
[SYSTEM]
Regenerate a plan rejected by validation or canonical deduplication.

[INPUT]
planning_context: <same task and generation mode>
rejected_plan: <candidate plan>
feedback: <schema, duplicate, or mutation-validation exception>

[REQUIREMENTS]
- Satisfy the plan schema and preserve the selected mutation prefix.
- Return a canonically distinct structure.
- The previous generation attempt could not be accepted.
- Validation feedback: <exception>.
- Regenerate the complete JSON plan and satisfy every contract.

[OUTPUT]
<one corrected Structure Plan v7 JSON object>
```

Listing 3: Simplified Structural-plan Regeneration prompt.

B.2.2 Structure Reviewer and Parent Selector

Structure Reviewer. The reviewer assesses each distinct, schema-valid plan as a logical execution design before implementation. It evaluates validity first, then scores structural potential along five

dimensions. The following rules define the evidence boundary, validity gates, and scoring rubric used by this role.

Evidence boundary. Review evidence comes only from the Structure Plan, Task Semantics, and Archive Summary. An evidence item is *explicit* when directly stated, *derived* when it follows unambiguously from the logical graph or semantic domains, and *uncertain* when materially different interpretations lead to different validity or structural conclusions. Task constants, initialization values, address formulas, and dtype requirements are inherited from Task Semantics unless the plan contradicts them or leaves their realization without a logical owner. A semantic partition is disjoint and exhaustive over semantic occurrences, including duplicate index occurrences, unless the plan states otherwise.

Layout, Tensor Core realization, `tl.dot`, tiles, warps, stages, precision lowering, mask expressions, split counts, traversal order, grid linearization, and register/shared-memory placement are local choices. They enter structural review only when they reveal a logical ownership, state-boundary, semantic-computation, or legal-coordination error. Missing local details alone neither invalidate a plan nor justify uncertainty.

Validity gates.

- **Contract consistency** checks agreement among strategy, logical kernels, semantic states, grid domains, producer–consumer edges, and final-output ownership. It fails only for a concrete contradiction and has no uncertain state.
- **Semantic consistency** checks the semantic DAG, reductions, ordering, duplicate occurrences, masks, invalid entries, numerical state, all-empty behavior, and output contract. An uncertain judgment requires a genuine ambiguity that Task Semantics and unambiguous derivation cannot resolve.
- **Realizability** checks whether legal kernels can implement the logical responsibilities, state boundaries, ownership, and coordination. Uncertainty applies when legality depends on an unspecified structural coordination mechanism, not on a missing physical mapping or instruction choice.

Each hard violation identifies a failed gate, conflicting plan elements, and the resulting semantic or coordination failure. Unresolved risks identify ambiguities that justify an uncertain gate or materially limit a score, explaining what cannot be derived and how it affects the conclusion. A low score never invalidates a plan, and performance is not a validity criterion.

Five-dimensional scoring. Every candidate receives exactly five scores from 0 to 4. The common scale is *poor* (0), *weak* (1), *mixed* (2), *strong* (3), and *excellent* (4). A score of 4 indicates that nearly all logically available opportunity is exposed without a comparable structural penalty. Table 6 gives the dimension-specific anchors. Scores describe structural potential, not correctness, implementation quality, or predicted speedup.

Table 6: Structure-review scoring rubric. All five dimensions use integer scores from 0 to 4; their weights are applied by the controller, separately from validity judgments.

Score	Structural criterion
Program computation quality (weight 0.25)	
0	Highly fragmented or redundant logical computation.
1	Major unnecessary boundaries or duplicated semantic work.
2	Plausible decomposition with significant boundary costs or missed fusion/split opportunities.
3	Clear responsibilities and well-placed computation boundaries.
4	Minimal, coherent boundaries that retain all useful logical computation relationships.

Continued on next page

Table 6 (continued)

Score	Structural criterion
Parallelism exposure (weight 0.25)	
0	Most available semantic work is unnecessarily serialized.
1	Limited independent work with a dominant logical bottleneck.
2	Adequate independent work, but a substantial serial or fan-in boundary remains.
3	Broad independent work with bounded reductions and clear ownership.
4	Nearly maximal semantic parallelism without an equally severe coordination boundary.
Data reuse opportunity (weight 0.20)	
0	Logically shared data is separated so that reuse is largely unavailable.
1	Little reuse opportunity survives the decomposition.
2	Some useful reuse is exposed, but important shared data crosses boundaries or is replicated.
3	Most important logical reuse relationships remain available.
4	All major logical reuse relationships are preserved within coherent responsibility boundaries.
State efficiency (weight 0.15)	
0	Dominant semantic state is extensively replicated or kept live across excessive boundaries.
1	Large or long-lived replicated state is intrinsic to the decomposition.
2	Moderate state cost with significant replication or lifetime overhead.
3	Compact state with bounded lifetime and limited replication.
4	State approaches the minimal sufficient semantic representation and lifetime.
Coordination efficiency (weight 0.15)	
0	A legal but coordination-heavy design has dominant global dependencies or fan-in.
1	Multiple costly logical handoffs or broad synchronization boundaries.
2	Moderate coordination with one or more significant dependencies.
3	Limited, well-scoped coordination with clear ownership.
4	Near-minimal coordination for the required semantics.

Computation quality concerns responsibility and fusion/split boundaries, not semantic correctness. Parallelism scores do not distinguish token-wide, token-head-group, and token-head physical mappings when the logical graph, state and reduction boundaries, and final ownership are unchanged. Reuse scores describe *opportunity*; achieved reuse is asserted only for an explicit semantic reuse edge. State scores use symbolic size, lifetime, and replication rather than physical placement or measured bandwidth. Coordination scores concern legal launch dependencies, synchronization, fan-in, state handoff, routing, and destination ownership; illegal coordination is a gate failure.

Controller aggregation and selection. Let $q_1, \ldots, q_5$ denote the scores in the order shown in Table 6. The controller computes

$$R(S) = 0.25q_1 + 0.25q_2 + 0.20q_3 + 0.15q_4 + 0.15q_5. \tag{5}$$

This weighted score lies in $[0, 4]$. The reviewer returns the five dimension scores and supporting evidence, not an aggregate score. Final selection ranks structurally valid plans by $R(S)$ and takes the top-m; ties use increasing candidate index, as specified in Appendix C.4. Validity judgments remain separate from this ranking.

Regimes and next step. Regime analysis remains qualitative, using structural distinctions such as low versus high outer-domain parallelism, short versus long reductions, or low versus high state replication. Confidence is high when the direction follows from explicit or derived facts, medium when it depends on unspecified workload balance, and low when local realization or missing workload information could reverse it. The reviewer does not invent thresholds or predict speedups.

The limiting phase is the earliest phase that must change to resolve the main remaining issue: `strategy` for semantic decomposition, ownership pattern, or global execution strategy; `kernel_graph` for logical boundaries, semantic states, edges, or merge/routing protocols; `grid` for semantic work domains or ownership granularity with a valid graph; and `local` when only physical realization remains. The next step is one concrete action at that phase. If all gates pass and only local work remains, it identifies the most important realization question without prescribing

an implementation. Evidence is tied to a gate or score dimension, and strengths and weaknesses describe comparative structural properties rather than repeat the plan.

```
[SYSTEM]
Review structural validity and optimization potential without
    execution.

[INPUT]
plan: <schema-valid, distinct candidate>
context: <semantic digest, review rules, and archive summary>

[REQUIREMENTS]
- Check contract consistency, semantics, and realizability.
- Contract consistency has pass/fail states only.
- Inherit task-contract facts unless contradicted or left unowned.
- Separate hard violations from unresolved risks.
- Score all five dimensions using the rubric in this subsection.
- A low score does not invalidate a plan.
- Return dimension scores only; the controller aggregates them.
- Use no source code, correctness results, or measured performance.
- Do not score Local choices or predict speedup.
- Tie evidence to a gate or dimension; choose one next step.

[OUTPUT]
validity: <three gate judgments, hard violations, unresolved risks>
scores: <computation; parallelism; reuse; state; coordination>
analysis: <strengths, weaknesses, regimes, and confidence>
evidence: <explicit, derived, or uncertain; linked gate or dimension>
next_step: <earliest limiting phase and one concrete action>
```

Listing 4: Simplified Structure Reviewer prompt.

```
[SYSTEM]
Regenerate a structural review that violates its output interface.

[INPUT]
review_context: <same plan, semantics, rules, and archive>
feedback: <parser or contract-validation exception>

[REQUIREMENTS]
- Follow the Structure Reviewer output interface.
- Do not introduce execution evidence.
- The previous review response was not accepted by the parser.
- Validation feedback: <exception>.
- Return exactly one valid JSON object in <review>...</review>.

[OUTPUT]
<review with validity, all five scores, evidence, and next step>
```

Listing 5: Simplified Structure-review Interface Regeneration prompt.

Parent Selector. For the third and subsequent candidates, the selector receives valid archived plans, scores, reviews, diversity, and unexplored options. It returns a parent identifier, restart phase (`strategy`, `kernel_graph`, or `grid`), mutation goal, and expected advantage regimes. It excludes physical program mapping, query-head grouping, grid linearization, and later implementation performance as structural evidence. This output guides structural mutation. Final selection instead ranks valid plans by weighted score and takes the top-m, replacing unrealizable plans with the next-ranked available plan.

```
[SYSTEM]
Select a parent and revision direction for structural mutation.

[INPUT]
archive: <valid plans, five scores, and supporting reviews>
coverage: <archive diversity and unexplored options>

[REQUIREMENTS]
- Select the earliest structural phase to reconsider.
- Exclude physical mapping, query-head grouping, and grid
    linearization.
- Do not use later implementation performance as structural evidence.

[OUTPUT]
parent: <archived plan identifier>
restart: <strategy | kernel_graph | grid>
goal: <structural mutation goal>
advantage: <expected workload regimes>
```

Listing 6: Simplified Structural Parent Selector prompt.

B.3 Implementation and Local Optimization

Structure Coder. The coder receives the selected plan, derived structure contract, operator definition, coding knowledge, and an optional parent implementation. It returns a complete Triton source file preserving coordination, logical responsibilities, intermediates, dependencies, grid domains, and output ownership. Conservative physical parameters can be refined during local optimization.

```
[SYSTEM]
Realize the selected structural plan as a complete Triton
    implementation.

[INPUT]
operator: <definition and interface>
plan: <selected structural plan and derived contract>
knowledge: <shared coding knowledge>
parent: <optional implementation>

[REQUIREMENTS]
- Preserve coordination, responsibilities, and semantic intermediates.
- Preserve dependencies, grid domains, and output ownership.
- Conservative physical parameters may be tuned in the local stage.

[OUTPUT]
<one complete Triton source file>
```

Listing 7: Simplified Structure Coder prompt.

Local Planner and Local Coder. The parent is the highest-$F(k)$ contract-preserving, correctness-valid implementation for the fixed structure. The planner receives the selected action, structure, parent source and implementation contract, available parent feedback, prior trials, and local knowledge. It returns one Local Plan v2 with parent and structure identifiers, one concrete change, changed fields, invalidated action levels, and a declaration of structure preservation. The coder applies only this plan and returns a complete source file and implementation contract for validation.

```
[SYSTEM]
Plan one local change while preserving the selected structure.

[INPUT]
structure: <fixed plan and contract>
parent: <current best valid source and implementation contract>
action: <one selected catalog action>
feedback: <available parent review and prior trials>
knowledge: <local optimization knowledge>

[REQUIREMENTS]
- Use the supplied current-best implementation as the sole parent.
- Specify one concrete change within the selected action.
- Identify changed fields and downstream actions invalidated.

[OUTPUT]
<Local Plan v2: parent, structure, action, change, changed fields,
 invalidated action levels, and structure-preservation declaration>
```

Listing 8: Simplified Local Planner prompt.

```
[SYSTEM]
Apply the Local Plan to the supplied parent implementation.

[INPUT]
parent: <current best valid source>
local_plan: <one planned action and concrete change>
contract: <fixed structural obligations>
knowledge: <shared coding knowledge>

[REQUIREMENTS]
- Apply only the specified local change.
- Preserve protected logical fields.
- Record physical implementation choices separately.

[OUTPUT]
source: <complete modified Triton source>
implementation_contract: <logical fields, physical kernels,
    parameters>
```

Listing 9: Simplified Local Coder prompt.

Local Reviewer. The reviewer receives candidate and parent identifiers, the Local Plan, contract and correctness results, and measured candidate and parent performance. It returns `keep`, `revise`, or `reject`, with contract, correctness, and performance assessments, strengths, weaknesses, and a next step. These recommendations guide the action sequence. Parent updates follow the strict measured-improvement rule in Algorithm 2; all trials remain in the history.

```
[SYSTEM]
Assess a local trial and recommend the next optimization action.

[INPUT]
identifiers: <candidate and parent>
plan: <Local Plan>
validation: <contract and correctness results>
performance: <measured candidate and parent performance>

[REQUIREMENTS]
- Analyze the measured effect and remaining bottlenecks.
- Guide action selection rather than selecting historical parents.
- Incumbent updates are controlled by strictly higher valid F(k).

[OUTPUT]
recommendation: <keep | revise | reject>
assessments: <contract, correctness, and performance>
analysis: <strengths, weaknesses, and next step>
```

Listing 10: Simplified Local Reviewer prompt.

B.4 Repair and Dispatch

Diagnosis and repair. Compilation or correctness failures invoke a shared diagnosis agent with the selected plan, failing source, evaluator output, and coding knowledge. It returns an evidence-supported root cause, a minimal list of edits, and fields to preserve. The corresponding coder returns a complete repaired source; local repair also returns a new implementation contract. The main experiments allow up to two repair rounds after the initial attempt.

```
[SYSTEM]
Diagnose a compilation or correctness failure from supplied evidence.

[INPUT]
plan: <selected structure or local plan>
source: <failing implementation>
evaluation: <compiler or correctness output>
knowledge: <shared coding knowledge>

[REQUIREMENTS]
- Identify an evidence-supported root cause.
- Specify a minimal repair and the fields that must be preserved.

[OUTPUT]
root_cause: <diagnosis supported by evaluator evidence>
edits: <minimal list of required edits>
preserve: <protected fields>
```

Listing 11: Simplified Failure Diagnosis prompt.

```
[SYSTEM]
Repair the failing implementation according to the diagnosis.

[INPUT]
source: <failing Triton implementation>
plan: <selected structure or local plan>
diagnosis: <root cause, edits, and fields to preserve>
knowledge: <shared coding knowledge>

[REQUIREMENTS]
- Apply the diagnosed repair while preserving protected fields.
- Return a complete source file.
- Local repair must also return an implementation contract.

[OUTPUT]
source: <complete repaired Triton source>
implementation_contract: <required for local repair>
```

Listing 12: Simplified Code Repair prompt.

Dispatch review and coding. A reviewer examines proposed workload rules and rejects rules based on UUIDs, sample indices, or complete-shape whitelists. If accepted, the Dispatch Coder namespaces the validated candidate implementations and composes one source file. Final correctness and aggregate-performance checks apply the acceptance rule in Appendix A.5.

```
[SYSTEM]
Review workload-based dispatch rules over validated implementations.

[INPUT]
rules: <proposed semantic-feature-based routing rules>
checkpoint: <candidate maturity and applicable-action coverage>
candidates: <candidate ID, correctness, completed actions,
  inapplicable actions, per-workload latency, geometric-mean speedup>

[REQUIREMENTS]
- Reject rules based on UUIDs, sample indices, or shape whitelists.
- The composed output remains subject to full-suite re-evaluation.

[OUTPUT]
<dispatch_review>{"accepted": true, "reason": "..."}</dispatch_review>
```

Listing 13: Simplified Dispatch Reviewer prompt.

```
[SYSTEM]
Compose accepted dispatch rules and validated candidate
    implementations.

[INPUT]
rules: <accepted workload-based routing rules>
candidates: <validated source implementations>
fallback: <best aggregate implementation>

[REQUIREMENTS]
- Namespace the candidate implementations in one source file.
- Route by the accepted general workload properties.
- Include the best aggregate candidate as a fallback.

[OUTPUT]
<one complete composed source file for correctness and timing checks>
```

Listing 14: Simplified Dispatch Coder prompt.

C Experimental Setup and Evaluation Protocol

C.1 Benchmarks and Task Selection

The evaluation covers a selected suite of 29 operators and 722 workload records: 426 for ten FlashInfer-Bench operators, 154 for ten SOL-ExecBench L1 operators, and 142 for nine SOL-ExecBench L2 operators. This workload suite is used on both GPUs; the count does not double-count hardware. Tables 7 and 22 list the evaluated tasks. All methods use the same operator definitions and workload configurations.

Table 7: FlashInfer-Bench task identifiers used in the detailed tables. Full operator names include the fixed configuration parameters.

Task	Operator
FI01	dsa_sparse_attention_h16_ckv512_kpe64_topk2048_ps64
FI02	gdn_decode_qk4_v8_d128_k_last
FI03	gdn_prefill_qk4_v8_d128_k_last
FI04	gemm_n4096_k4096
FI05	gqa_paged_decode_h32_kv8_d128_ps1
FI06	gqa_paged_prefill_causal_h32_kv8_d128_ps1
FI07	gqa_ragged_prefill_causal_h32_kv8_d128
FI08	mla_paged_decode_h16_ckv512_kpe64_ps1
FI09	mla_paged_prefill_causal_h16_ckv512_kpe64_ps1
FI10	rmsnorm_h4096

Table 8 lists the hardware-specific expert-library speedups used to normalize FlashInfer-Bench results.

Table 8: FlashInfer expert-library speedups relative to eager. N/A denotes an unavailable expert measurement and uses a normalization factor of one.

Task	Operator	A800	H100
FI01	dsa_sparse_attention_h16_ckv512_kpe64_topk2048_ps64	8.73	12.04
FI02	gdn_decode_qk4_v8_d128_k_last	210.37	389.66
FI03	gdn_prefill_qk4_v8_d128_k_last	N/A	626.47
FI04	gemm_n4096_k4096	N/A	N/A

Continued on next page

Table 8 (continued)

Task	Operator	A800	H100
FI05	gqa_paged_decode_h32_kv8_d128_ps1	213.16	195.77
FI06	gqa_paged_prefill_causal_h32_kv8_d128_ps1	154.55	43.42
FI07	gqa_ragged_prefill_causal_h32_kv8_d128	6.88	1.58
FI08	mla_paged_decode_h16_ckv512_kpe64_ps1	39.24	49.20
FI09	mla_paged_prefill_causal_h16_ckv512_kpe64_ps1	152.04	187.68
FI10	rmsnorm_h4096	4.71	8.33

Tables 9, 10, and 11 summarize the workload counts, variable dimensions, and input data types extracted from each task's definition and workload files. Counts refer to workload records, which may share shapes while using different inputs. The ranges summarize observed axis values rather than specifying Cartesian products of all dimensions.

Tasks are selected to cover the major operator families and execution patterns, span workload shapes and difficulty levels, and avoid using final performance as a selection criterion, as described in Section 5.1.

Table 9: FlashInfer-Bench workload configurations. Ranges summarize observed variable-axis values, not Cartesian products. Input types include scalar and index inputs. BF16/FP16/FP32 denote floating-point types; I32/I64 denote integer types.

Task	Workloads	Input types	Variable dimensions (minimum–maximum)
FI01	23	BF16, FP32, I32	num pages: 8462; num tokens: 1–8
FI02	54	BF16, FP32	batch size: 1–64
FI03	100	BF16, FP32, I64	len cu seqlens: 2–58; num seqs: 1–57; total seq len: 6–8192
FI04	43	FP16	M: 1–8192
FI05	48	BF16, FP32, I32	batch size: 1–64; len indptr: 2–65; num kv indices: 2–58966; num pages: 11–81390
FI06	38	BF16, FP32, I32	len indptr: 2–53; num kv indices: 1–2560; num pages: 2–716370; total q: 1–16384
FI07	21	BF16, FP32, I32	len indptr: 2–37; total kv: 1–13557; total q: 1–13557
FI08	47	BF16, FP32, I32	batch size: 1–64; len indptr: 2–65; num kv indices: 8–75145; num pages: 989669
FI09	38	BF16, FP32, I32	len indptr: 2–57; num kv indices: 5–17759; num pages: 989669; total q: 1–16384
FI10	14	BF16	batch size: 1–14509
Total	426		

Table 10: SOL-ExecBench L1 workload configurations. Ranges summarize observed variable-axis values, not Cartesian products. Input types include scalar and index inputs. BF16/FP16/FP32 denote floating-point types; I32/I64 denote integer types.

Task	Workloads	Input types	Variable dimensions (minimum–maximum)
L1/002	20	FP32	batch size: 1–64; height: 64–1024; width: 64–1024
L1/005	16	BF16	batch size: 1–32; seq len: 128–8192
L1/007	16	FP32	batch size: 1–64; seqlen: 131–32768
L1/008	16	BF16, I64	batch size: 1–64; seq len: 128–8192
L1/018	13	BF16, FP32, I64	batch size: 1–8; cache len: 0–2048; seq len: 1–4096
L1/020	15	BF16, FP32, I64	num grids: 1–8; num merged patches: 128–16384; num patches: 512–65536
L1/053	12	BF16, FP32	batch size: 1–64; intermediate size: 4096–16384; seq len: 128–8192
L1/058	16	I32	batch size: 1–64; seq len: 128–4096

Continued on next page

Table 10 (continued)

Task	Workloads	Input types	Variable dimensions (minimum–maximum)
L1/070	14	BF16	batch size: 1–16; num chunks: 1–32
L1/092	16	BF16, FP32	batch size: 1–32; seq len: 128–2048
Total	154		

Table 11: SOL-ExecBench L2 workload configurations. Ranges summarize observed variable-axis values, not Cartesian products. Input types include scalar and index inputs. BF16/FP16/FP32 denote floating-point types; I32/I64 denote integer types.

Task	Workloads	Input types	Variable dimensions (minimum–maximum)
L2/012	16	BF16, I64	num tokens: 1280–8192
L2/015	16	BF16, FP32	batch size: 1–64; time after conv: 16–541; time dim: 128–4328
L2/030	16	FP32	batch size: 1–32; img seq len: 256–4096; text seq len: 77–1423
L2/036	14	FP32	B: 1–64; H: 14–56; W: 14–56
L2/040	16	BF16, FP32, I32	batch size: 1–64; seq len: 128–1024
L2/043	16	BF16	batch size: 1–32; seq len: 256–4096
L2/049	16	BF16, FP32	num tokens: 2048–16384
L2/051	16	FP32	batch size: 1–32; seq len: 128–4096
L2/080	16	BF16, FP32, I64	batch seq len: 192–8192
Total	142		

Table 12: Numerical tolerance fields in the SOL-ExecBench workload records. Ranges reflect workload-specific settings. A dash means the match-ratio field is absent from the record; it does not specify an evaluator default.

Task	max_atol	max_rtol	required_match_ratio
L1/002	0.0027–0.0034	1e-05	–
L1/005	0.0082–0.028	0.05	–
L1/007	1e-05	1e-05	–
L1/008	0.07–0.13	0.05	–
L1/018	1e-05–0.0098	0.05	–
L1/020	0.0013–0.0038	0.05	–
L1/053	1e-05	0.05	–
L1/058	1e-05	0.01	–
L1/070	1e-05	0.05	–
L1/092	0.0025–0.0051	0.05	–
L2/012	0.0059–2.3	0.05	0.98
L2/015	0.92–1.3	0.05	0.98
L2/030	0.0018–0.0021	1e-05	0.98
L2/036	0.2–5.2	0.001	0.98
L2/040	0.0014–0.015	0.05	0.98
L2/043	0.014–0.046	0.05	0.98
L2/049	0.44–0.84	0.01	0.98
L2/051	0.00024–0.00032	1e-05	0.98
L2/080	0.11–0.39	0.05	0.98

C.2 Hardware and Software Environment

Experiments use NVIDIA A800-SXM4-80GB and NVIDIA H100 80GB HBM3 GPUs. Table 13 reports the software versions read from the designated evaluation environments. Both PyTorch installations use CUDA 13.0. The inspected A800 host reports NVIDIA driver 580.105.08.

Table 13: Software environments for the two benchmarks.

Benchmark	Python	PyTorch	Triton
FlashInfer-Bench	3.11.15	2.13.0+cu130	3.7.1
SOL-ExecBench	3.12.14	2.9.0+cu130	3.5.0

C.3 Baseline Configurations

The direct-generation baselines are GPT-5.6-Sol and Claude Opus 4.8. Codex and Claude Code represent general-purpose coding agents; Kernel Design Agent and K-Search represent kernel-specific optimization systems; Dr. Kernel is a specialized generation model. Each baseline uses its official interface and recommended configuration whenever available, with the same operator definitions, correctness suite, timing harness, hardware, and aggregation procedure.

Table 14 lists the model assignments recorded in the experiment spreadsheet. Codex, Claude Code, and KDA each use a one-hour time limit per task. K-Search uses budget $B = 20$ and stagnation threshold $K = 5$. Dr. Kernel launches eight independent search processes, each making 50 attempts, for up to 400 attempts per task.

Table 14: Model assignments recorded in the experiment spreadsheet.

Method	Interface	Recorded model
GPT-5.6-Sol	Direct generation	GPT-5.6-Sol
Claude Opus 4.8	Direct generation	Claude Opus 4.8
Codex	Codex	GPT-5.6-Sol
Claude Code	Claude Code	Claude Opus 4.8
KDA	Claude Code	Claude Opus 4.8
Dr. Kernel	Specialized generation	Dr. Kernel-8B
K-Search	K-Search	GPT-5.6-Sol
StruTune	Multi-agent pipeline	GPT-5.6-Sol

C.4 StruTune and Ablation Configurations

All StruTune roles use GPT-5.6-Sol through the same OpenAI-compatible chat interface; the client does not override temperature or top-p. The main settings are summarized in Table 15. Structural candidates are generated from the task context and shared planning rules, either autonomously or through review-guided mutation.

Table 15: Main search budgets and retry limits. Structural-plan regeneration and code repair are separate operations.

Setting	Value
Reviewed structural candidates, B_{S}	5
Target correctness-valid structures, m	2
Local iterations per realized structure, B_{L}	10
Structural-plan attempts, including the initial attempt	Up to 5
Structure-review attempts, including the initial attempt	Up to 3
Code repairs after each initial implementation attempt	Up to 2
Main-evaluation workload dispatch	Enabled
Workload-relative gain threshold for proposing dispatch	5%

The component ablations run on A800. Local-only allows up to ten local optimization rounds without structural search or workload dispatch. Structure-only generates and realizes five structures without local optimization or dispatch, returning the best correctness-valid implementation. The full method generates five plans, selects two for local optimization, and enables dispatch after tuning. Equal Stage I reviewer scores are ordered by increasing candidate index, including ties at the top-m cutoff. Local performance ties retain the incumbent, as specified in Algorithm 2.

The cumulative-coverage analysis in Figure 4 evaluates five generated structures with ten local rounds per structure. The separate DSA controlled-optimization experiment also uses ten local rounds, with controls specified in Appendix D.4.

C.5 Correctness, Timing, and Aggregation

Correctness is checked across the complete workload suite before performance is measured. Each search-time benchmark uses three warm-up runs, five timed iterations, one timing trial, and a 600-second evaluator timeout per workload. A task is counted as solved only when its returned implementation passes the complete workload suite.

The per-task objective is $F(k)$ from Eq. (1), the geometric-mean speedup over that task's workloads. Benchmark-level speedup is the geometric mean of these per-task values over correctly solved tasks. FlashInfer uses the hardware-specific expert-to-eager factors in Table 8, with unavailable factors set to one; SOL-ExecBench uses the eager baseline. Relative improvements over a baseline compare both methods on exactly the subset solved by that baseline. Correctness rate and speedup are reported together because speedup excludes failed tasks.

The SOL-ExecBench records specify workload-dependent numerical tolerances, summarized in Table 12. Input records specify random, scalar, custom, or stored-tensor inputs. FlashInfer workload records do not contain explicit tolerance fields.

The supplied FlashInfer evaluator uses absolute and relative tolerances of 10^{-2} and requires a matched-element ratio of 0.99. It checks shape and dtype, rejects NaNs and unmatched infinities, and permits matching negative infinities. Scalar and stored-tensor inputs follow workload specifications; other FP32, FP16, and BF16 inputs use standard-normal draws. Integer input bounds are dtype-dependent in the benchmark generator. The StruTune wrapper does not set a random seed. SOL-ExecBench input generation and workload-specific tolerances are delegated to its benchmark CLI.

FlashInfer timing calls the benchmark's CUPTI timing utility with a cold L2 cache and CUDA graphs disabled, taking the median of the timed iterations. Candidate and reference latencies are averaged over generated input trials before forming their ratio. Compilation and correctness checks precede timing. These are GPU timings, not end-to-end host or search elapsed times. The final dispatcher is reevaluated over the workload suite, reverting to its fallback if the composed implementation fails validation. Recorded failure categories distinguish runtime/compilation errors, shape errors, dtype errors, and numerical errors.

C.6 Resource Accounting

Every StruTune LLM call stores its prompt, response, model identifier, and input/output token counts in an independent log record. Search records distinguish candidates and their parents, correctness and performance outcomes, and action history. Appendix D.3 reports the available DSA token allocation; Appendix E.2 reports per-task token totals for seven methods on both GPUs.

The client logs the provider-reported prompt and completion token counts for each completed response. Planner, reviewer, coder, regeneration, and repair calls use the same logging interface. Summing these records includes completed retry and repair calls; requests ending without a completion are stored as errors without measured token usage. The log does not separate cached tokens or record end-to-end elapsed time, so no cache discount or monetary cost is inferred. The reported task-level totals appear in Appendix E.2.

D Structural Lock-in Analysis

This appendix separates structure identification, coverage, effort allocation, and controlled local replay. It expands the analysis in Section 3.2; code-level examples are organized in Appendix F.

D.1 Trace Collection and Structure Annotation

Structural families are defined by the decisions $S = (A, G, P)$, using the equivalence rules in Appendix A.1. Family identifiers are local to each method: matching labels do not imply equivalent plans across methods. The K-Search trace protocol records world-model nodes, planned actions, parent and child code, compilation, correctness and performance outputs, and token usage. Its structural annotation is performed without access to measured performance.

The initial cross-task counts come from agents reviewing their own optimization logs for structures reported to pass correctness validation. Repeated implementations of the same structure are merged; K-Search retry directories are not counted as additional tasks. These automated annotations have not been independently verified. We reconcile them with the final A800 performance table: any task without an accepted numeric performance result receives zero. This zero is an outcome-based reporting convention, not evidence that no intermediate correct implementation ever existed. Counts for solved tasks retain the agent annotations, except that KDA's DSA count is corrected to two using the trace analysis in Section 3.2.

For all four methods, the annotation unit is a task/method record linked by task identifier to the final A800 result. Reconciliation checks reporting consistency with final outcomes; it does not independently verify intermediate structure labels.

D.2 Structural Coverage Across Tasks

Table 16 reports all 29 task records for Claude Code, Codex, KDA, and K-Search on A800 under this common reconciliation rule.

Among solved tasks, Claude Code has one or two reported structures on 28/28 tasks, Codex on 26/29, KDA on 21/24, and K-Search on 18/20. The resulting fractions are 100%, 89.66%, 87.50%, and 90%, respectively. Assigned zeros are excluded from these fractions, so failed tasks do not inflate the structural-concentration statistic. Table 17 separates the benchmark groups. On FlashInfer-Bench, the single-structure fractions among solved tasks are 8/10, 9/10, 5/10, and 5/9, respectively; none of the solved tasks exceeds two reported structures.

Table 16: Per-task structural counts from agent log review, reconciled with A800 performance outcomes. Zero marks an unsolved task, not a verified absence of intermediate valid candidates. CC denotes Claude Code; task identifiers map to the operator names in Tables 7 and 22. Repeated candidates of the same structure do not increase the count.

Task	CC	Codex	KDA	K-Search
FI01	2	1	2	1
FI02	1	1	1	1
FI03	1	1	2	2
FI04	1	1	2	0
FI05	1	1	1	1
FI06	2	2	1	2
FI07	1	1	1	1
FI08	1	1	2	2
FI09	1	1	2	2
FI10	1	1	1	1
L1/002	1	1	1	0
L1/005	1	1	2	2
L1/007	2	4	2	0
L1/008	1	1	2	2
L1/018	1	4	1	1
L1/020	1	1	0	1
L1/053	1	1	1	2
L1/058	1	1	2	3
L1/070	1	1	1	1
L1/092	1	1	4	0

Continued on next page

Table 16 (continued)

Task	CC	Codex	KDA	K-Search
L2/012	1	1	0	0
L2/015	0	1	0	0
L2/030	1	1	1	1
L2/036	1	1	0	0
L2/040	1	1	3	3
L2/043	2	2	1	1
L2/049	1	1	2	1
L2/051	1	5	0	0
L2/080	1	1	3	0

Table 17: Structural coverage by benchmark group on A800. The last two columns report the fractions of solved tasks with exactly one or with one or two structures. Unsolved tasks are shown in the zero column and excluded from these percentages.

Group	Method	0	1	2	≥ 3	1 (%)	1–2 (%)
FlashInfer-Bench	Claude Code	0	8	2	0	80.00	100.00
FlashInfer-Bench	Codex	0	9	1	0	90.00	100.00
FlashInfer-Bench	KDA	0	5	5	0	50.00	100.00
FlashInfer-Bench	K-Search	1	5	4	0	55.56	100.00
SOL-ExecBench L1	Claude Code	0	9	1	0	90.00	100.00
SOL-ExecBench L1	Codex	0	8	0	2	80.00	80.00
SOL-ExecBench L1	KDA	1	4	4	1	44.44	88.89
SOL-ExecBench L1	K-Search	3	3	3	1	42.86	85.71
SOL-ExecBench L2	Claude Code	1	7	1	0	87.50	100.00
SOL-ExecBench L2	Codex	0	7	1	1	77.78	88.89
SOL-ExecBench L2	KDA	4	2	1	2	40.00	60.00
SOL-ExecBench L2	K-Search	5	3	0	1	75.00	75.00

D.3 Search Effort Allocation

Figure 2(a) aggregates traced tokens by structural family for DSA sparse attention. Table 18 gives the corresponding values. Codex retains the same two-kernel organization across all 32 complete performance evaluations. KDA explores a second family, but most traced tokens remain associated with its dominant family.

Table 18: Reported DSA token allocation by structural family. Counts are in millions of tokens; family identifiers are method-specific.

Method	Families	Total tokens	Dominant-family tokens	Share (%)
Codex	1	1.57	1.57	100.00
KDA	2	13.83	12.66	91.54
K-Search	1	1.62	1.62	100.00

KDA's other family receives 1.17M tokens (8.46%). These counts describe concentration within the analyzed runs; the controlled replay below examines the performance consequences of different structural choices.

These figures use the supplied family-level token totals. They do not estimate per-stage token consumption or the fraction of iterations after the last new structure; neither statistic is reconstructed from aggregate token counts. The controlled optimization in Appendix D.4 is a separate experiment from these source traces.

D.4 CONTROLLED OPTIMIZATION OF EXTRACTED STRUCTURES

This experiment abstracts structural plans from the four methods' final implementations, then uses Codex to implement and optimize each fixed plan with a common local-action library and ten-round budget. It is separate from the main benchmark comparison: the source method identifies the extracted structure, not the optimizer used in this experiment. The supplied replay inputs contain one plan per baseline and a fixed two-branch dispatch bundle for STRUTUNE (Table 20). Figure 2(b) shows the best speedup through each round with solid lines and individual-round measurements with dashed lines.

Table 19: Best correctness-valid speedup observed over ten controlled local rounds on DSA. The source identifies the skeleton; Codex is the optimizer in every row.

Skeleton source	Best observed speedup
Codex	$28.27\times$
KDA	$28.63\times$
K-Search	$21.34\times$
STRUTUNE	$37.47\times$

Under this protocol, the STRUTUNE skeleton improves on the strongest alternative, KDA, by 30.88%. This is an observed difference under a shared finite budget, rather than an estimate of an asymptotic performance ceiling.

Table 20: Structural identities extracted from the supplied DSA replay inputs. The KDA input is its structure-1 plan; the STRUTUNE input includes a fixed workload-dispatch rule.

Source	Strategy A	Logical graph G and intermediate state	Parallelization P and ownership
Codex	Materialized sparse scores	Score production followed by normalization and value accumulation; the full score relation is stored between stages.	Score owners span query tokens and sparse-key occurrences; final owners span query tokens and value channels.
KDA	Partitioned online softmax	Sparse-key partitions produce maximum, exponential sum, and weighted-value numerator states; a second stage merges them.	Partition owners span query tokens and reduction partitions; merge owners span query tokens and query heads.
K-Search	Fused online softmax	One logical stage performs scoring, normalization, value accumulation, and final output; no cross-stage intermediate.	Each query-token owner handles all query heads, selected keys, output channels, and log-sum-exp values.
STRUTUNE	Fixed dispatch between fused and partitioned attention	At most two query tokens use fused attention; larger inputs use partition states followed by a merge.	Fused owners span query tokens; partition owners span query tokens and reduction partitions; merge owners span query tokens and value channels.

Replay inputs and protocol. The controlled experiment uses NVIDIA A800-SXM4-80GB hardware and 23 DSA workloads, with speedups measured against eager. Codex receives the task definition, workloads, fixed structural plan, and shared action library, and generates the implementation rather than receiving the source method's optimized code. The prescribed sequence is a one-workload smoke test, full-workload correctness and repair, then ten single-action optimization rounds. Initial correctness checks and repairs are outside the numbered performance-round budget; there is no generation-phase wall-clock timeout.

The identical action library covers equivalent implementation paths; layouts and address access; compute primitives and precision; physical parallel parameters; kernel scheduling and resources; memory reuse and data movement; masks and static specialization; and host, launch, and buffer overhead. Each action must preserve the plan's logical dependencies and ownership.

Recorded rounds. Table 21 gives all ten rounds of the controlled experiment and supplies the data for Figure 2(b). The best observed speedups are $28.27\times$, $28.63\times$, $21.34\times$, and $37.47\times$ for the Codex, KDA, K-Search, and STRUTUNE structures, respectively. The STRUTUNE structure reaches its best result in round nine. KDA has no reported numerical result in round five: its individual-round curve has a gap, while its best-so-far value remains $19.81\times$. No zero speedup is imputed.

Table 21: Complete DSA controlled-optimization results, expressed as eager-relative speedups. Column names identify the source of the extracted structure; Codex optimizes every structure. A dash denotes no reported numerical result for that round, not zero speedup.

Round	Codex	KDA	K-Search	STRUTUNE
1	26.06	19.28	0.11	21.92
2	25.29	19.72	0.21	25.82
3	27.43	19.81	14.74	25.97
4	27.62	17.01	14.24	29.75
5	27.69	–	11.40	29.82
6	27.84	20.45	18.09	28.38
7	26.09	24.17	18.79	31.70
8	28.13	27.81	20.31	36.73
9	28.27	27.75	20.81	37.47
10	27.33	28.63	21.34	35.85
Best	28.27	28.63	21.34	37.47

E COMPLETE RESULTS AND ADDITIONAL ANALYSES

This appendix reports the supplied task-level speedups and token-usage records for the evaluated 29-task suite on A800 and H100. Tables 7 and 22 identify the ten FlashInfer-Bench and nineteen SOL-ExecBench tasks. Empty performance cells denote confirmed failures and receive no imputed speedup.

E.1 PER-TASK CORRECTNESS AND PERFORMANCE

Reference and precision. The supplied per-task results use an eager reference for both benchmarks. Tables 23 and 24 therefore report FlashInfer-Bench speedups before expert normalization. For task t and hardware H, Table 2 uses $s^{\mathrm{norm}}_{t,H} = s^{\mathrm{eager}}_{t,H}/e_{t,H}$, where $e_{t,H}$ is the expert-to-eager speedup in Table 8; N/A entries use $e_{t,H} = 1$. The SOL-ExecBench tables use the same eager reference as the main table. Values are displayed to two decimal places, while geometric means and paired relative improvements use the full supplied precision.

Methods and status markers. GPT and Claude denote direct generation with GPT-5.6-Sol and Claude Opus 4.8. Codex uses GPT-5.6-Sol; CC denotes Claude Code with Claude Opus 4.8. KDA denotes Kernel Design Agent, whose source configuration uses Claude Code with Claude Opus 4.8; DrK denotes the Dr. Kernel-8B baseline; KS denotes K-Search with GPT-5.6-Sol; Ours denotes STRUTUNE with GPT-5.6-Sol. Numeric entries are the reported speedups. H denotes a source entry tagged `hack`, including any number attached to that tag, and is not admitted as a numeric performance result. The slash marker / is retained from the source as a non-numeric result. F denotes a confirmed failure recorded as an empty performance cell; no numerical speedup is assigned. The Numeric footer counts positive numeric entries out of the evaluated tasks; its geometric mean excludes all non-numeric entries and is not a zero-filled average.

Table 22: SOL-ExecBench task identifiers used in the detailed tables. Operator names follow the supplied results.

Task	Operator
L1/002	vae conv3x3 groupnorm silu residual fused
L1/005	conv gated projection with causal conv
L1/007	hyena fft size padding rfft
L1/008	expert output weighted index add accumulation
L1/018	fused rope with qk norm and kv cache update
L1/020	vision patch merger spatial shuffle mlp
L1/053	gaussian topk sparse activation
L1/058	moe expert token radix sort with prefix sum
L1/070	mamba2 fused intra chunk diagonal computation
L1/092	gqa attention with qk norm
L2/012	moe expert batched execution with capacity factor
L2/015	audio sinusoidal position embedding with conv projection
L2/030	flux concatenated sequence processing with split
L2/036	convnextv2 layer with nhwc persistence backward
L2/040	altup predict correction cycle backward
L2/043	mamba chunk scan with segsum
L2/049	group limited topk routing
L2/051	seqlen-finetuned-reconstructed hyena complete forward block
L2/080	moe complete layer with shared expert backward

Table 23: FlashInfer-Bench per-task speedup over eager on A800. Bold marks the largest numeric value in each row. H preserves a source `hack` flag; / preserves the source non-numeric marker; F denotes an author-confirmed failure from an empty cell. The footer reports numeric coverage and the geometric mean over numeric entries, which can cover different task subsets.

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
FI01	26.54	12.77	37.25	37.02	25.33	/	20.72	**47.17**
FI02	311.64	156.54	248.45	270.05	247.87	/	225.25	**320.09**
FI03	175.28	98.22	144.14	194.50	127.79	/	195.33	**198.92**
FI04	0.65	0.59	0.73	0.63	0.59	0.55	H	**0.77**
FI05	209.21	424.74	537.01	653.12	539.16	/	124.87	**709.01**
FI06	571.16	359.81	672.06	348.92	63.06	/	392.86	**1076.94**
FI07	11.20	8.39	12.67	13.27	12.92	/	11.07	**19.72**
FI08	47.39	64.70	29.24	61.31	56.26	0.21	28.40	**85.26**
FI09	224.56	139.40	225.80	224.96	55.44	/	173.17	**253.41**
FI10	4.05	3.99	4.35	4.34	4.19	3.30	3.47	**4.73**
Numeric	10/10	10/10	10/10	10/10	10/10	3/10	9/10	10/10
Geo. mean	47.15	37.39	51.37	54.27	34.95	0.73	58.84	71.64

Table 24: FlashInfer-Bench per-task speedup over eager on H100. Bold marks the largest numeric value in each row. H preserves a source `hack` flag; / preserves the source non-numeric marker; F denotes an author-confirmed failure from an empty cell. The footer reports numeric coverage and the geometric mean over numeric entries, which can cover different task subsets.

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
FI01	16.40	20.63	28.62	20.87	24.31	/	23.92	**36.72**
FI02	241.49	204.09	289.02	273.92	293.35	/	217.57	**297.35**
FI03	77.62	96.90	170.62	116.40	131.90	/	**230.34**	206.44
FI04	0.57	0.48	**0.70**	0.55	0.54	0.36	0.47	0.63
FI05	292.89	401.67	512.14	375.41	520.22	/	389.40	**619.55**
FI06	493.08	154.11	877.27	336.64	/	/	555.89	**892.89**
FI07	9.59	7.50	14.92	9.78	12.45	/	11.12	**17.35**

Continued on next page

Table 24 (continued)

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
FI08	33.01	17.74	48.53	59.37	47.19	/	38.73	**74.42**
FI09	140.99	32.07	236.61	73.57	118.64	/	173.47	**244.98**
FI10	4.26	3.83	4.23	4.05	4.37	1.38	3.61	**4.49**
Numeric	10/10	10/10	10/10	10/10	9/10	2/10	10/10	10/10
Geo. mean	37.00	26.90	56.33	38.93	35.08	0.70	44.84	63.79

Table 25: SOL-ExecBench per-task speedup over eager on A800. Bold marks the largest numeric value in each row. H preserves a source `hack` flag; / preserves the source non-numeric marker; F denotes an author-confirmed failure from an empty cell. The footer reports numeric coverage and the geometric mean over numeric entries, which can cover different task subsets.

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
L1/002	H	H	0.79	1.04	0.76	/	H	**1.08**
L1/005	1.35	1.37	1.25	1.39	1.37	/	**1.42**	1.39
L1/007	H	H	0.25	0.43	0.10	/	H	**0.69**
L1/008	1.39	1.80	2.16	2.95	1.85	2.19	2.70	**7.96**
L1/018	15.86	13.65	18.25	13.80	15.10	/	**21.24**	17.89
L1/020	1.45	1.15	1.61	1.57	/	/	1.46	**1.78**
L1/053	H	27.57	28.43	27.11	26.36	3.22	**31.81**	29.32
L1/058	7.36	0.60	4.85	5.76	3.17	0.65	10.32	**11.08**
L1/070	35.03	217.77	196.75	267.30	231.09	/	192.99	**294.94**
L1/092	1.88	2.24	1.71	2.36	2.24	F	H	**2.41**
L2/012	1.10	/	0.78	1.12	H	/	H	**1.14**
L2/015	/	/	0.39	/	/	/	H	**1.01**
L2/030	1.30	0.89	3.17	1.23	0.60	0.14	1.38	**4.85**
L2/036	10.30	78.90	251.01	131.95	/	/	H	**278.53**
L2/040	11.36	5.80	18.55	6.64	9.85	/	17.40	**22.37**
L2/043	11.51	/	4.36	4.15	19.85	/	9.17	**21.28**
L2/049	8.63	2.10	7.61	**9.69**	4.33	/	8.94	9.54
L2/051	H	/	0.99	1.02	/	/	H	**1.37**
L2/080	1.06	4.51	3.48	4.35	4.43	F	H	**4.84**
Numeric	14/19	13/19	19/19	18/19	14/19	4/19	11/19	19/19
Geo. mean	4.09	5.00	4.06	4.60	4.02	0.90	8.41	6.51

Table 26: SOL-ExecBench per-task speedup over eager on H100. Bold marks the largest numeric value in each row. H preserves a source `hack` flag; / preserves the source non-numeric marker; F denotes an author-confirmed failure from an empty cell. The footer reports numeric coverage and the geometric mean over numeric entries, which can cover different task subsets.

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
L1/002	H	0.53	**1.26**	0.62	/	/	H	0.79
L1/005	1.46	/	1.38	1.48	**1.55**	/	1.48	1.39
L1/007	H	H	0.36	0.09	0.06	/	H	**0.57**
L1/008	4.57	2.71	4.24	2.96	/	2.82	5.24	**5.94**
L1/018	17.49	/	/	5.25	15.06	/	18.97	**19.33**
L1/020	1.96	/	2.09	1.11	/	/	1.82	**2.36**
L1/053	31.79	34.71	35.14	30.44	32.92	12.94	**35.90**	35.20
L1/058	5.99	1.62	2.56	3.31	3.66	6.75	**13.00**	10.31
L1/070	87.88	/	38.81	316.50	362.24	/	184.70	**370.25**
L1/092	/	2.28	1.23	2.57	2.18	/	H	**2.62**
L2/012	1.34	/	1.07	0.96	H	/	H	**1.54**
L2/015	/	/	/	/	/	/	H	**1.15**
L2/030	2.26	1.64	1.49	1.27	1.71	0.14	1.57	**3.54**
L2/036	/	H	17.11	15.95	/	/	H	**19.48**

Continued on next page

Table 26 (continued)

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
L2/040	/	2.01	3.44	7.13	/	/	8.46	**13.45**
L2/043	1.34	2.26	3.88	10.10	/	/	H	**21.15**
L2/049	5.94	/	**9.12**	4.40	F	/	7.01	9.07
L2/051	0.78	/	1.53	1.18	F	/	H	**3.90**
L2/080	1.11	3.32	3.54	**4.55**	3.26	/	H	3.93
Numeric	13/19	9/19	17/19	18/19	9/19	4/19	10/19	19/19
Geo. mean	4.05	2.55	3.22	3.41	4.49	2.42	8.57	5.77

Table 27: Paired relative speedup gains of STRUTUNE (%) over each baseline, evaluated on exactly that baseline's solved task subset. These are the paired comparisons accompanying Table 2; they are not ratios of the unpaired aggregate speedups.

Baseline	Flash A800	SOL A800	Flash H100	SOL H100
GPT-5.6-Sol	51.92	143.22	72.40	103.28
Claude-Opus-4.8	91.57	141.32	137.17	150.40
Codex	39.46	60.06	13.24	83.48
Claude Code	31.99	56.94	63.85	85.08
Kernel Design Agent	104.98	96.53	35.62	66.99
K-Search	101.48	42.75	42.25	26.11
Dr. Kernel	833.97	1082.37	138.62	285.69

Available coverage. STRUTUNE has numeric results for all 29 tasks on each GPU. On SOL-ExecBench, K-Search has numeric results for 11 of 19 tasks on A800 and 10 of 19 on H100. Its geometric-mean speedups are $8.41\times$ and $8.57\times$ over eager, respectively. On those same task subsets, STRUTUNE improves by 42.75% and 26.11%. These are paired subset comparisons rather than a ranking of averages computed over different task sets.

The main comparison is reconstructed from task-level spreadsheet values. Evaluator-level failure categories are described in Appendix C.5; the spreadsheet's / and H markers remain source annotations rather than inferred diagnoses.

E.2 SEARCH COST AND EFFICIENCY

Tables 28, 29, 30, and 31 report task-level token usage in millions of tokens (M) for both GPUs. GPT, Claude, Codex, Claude Code, KDA, K-Search, and STRUTUNE each have records for all 29 tasks on each GPU. Totals and arithmetic means include every listed task, including tasks without numeric speedups. Values are displayed to three decimal places to preserve the smaller direct-generation values. Dr. Kernel is a locally deployed, reinforcement-learning-trained 8B model and is excluded from token accounting; N/A denotes this exclusion, not zero consumption.

Table 28: Token usage for FlashInfer-Bench on A800, in millions of tokens (M). All listed tasks are included regardless of performance status. N/A denotes the locally deployed Dr. Kernel, excluded from token accounting.

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
FI01	0.094	0.078	1.570	6.890	13.830	N/A	1.620	0.320
FI02	0.167	0.086	1.670	0.910	1.370	N/A	1.390	0.460
FI03	0.160	0.169	6.580	5.740	1.270	N/A	1.870	0.480
FI04	0.110	0.074	14.550	5.540	2.660	N/A	0.850	0.300
FI05	0.112	0.120	16.860	2.780	1.300	N/A	3.180	0.360
FI06	0.110	0.111	3.930	1.770	1.270	N/A	1.790	0.380
FI07	0.061	0.067	2.240	4.640	1.250	N/A	1.570	0.430
FI08	0.129	0.111	2.930	7.050	1.340	N/A	1.940	0.420

Continued on next page

Table 28 (continued)

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
FI09	0.112	0.100	8.430	7.030	1.270	N/A	1.820	0.410
FI10	0.077	0.028	1.780	1.610	1.170	N/A	0.950	0.290
Numeric	10/10	10/10	10/10	10/10	10/10	N/A	10/10	10/10
Total	1.132	0.944	60.540	43.960	26.730	N/A	16.980	3.850
Mean per task	0.113	0.094	6.054	4.396	2.673	N/A	1.698	0.385

Table 29: Token usage for FlashInfer-Bench on H100, in millions of tokens (M). All listed tasks are included regardless of performance status. N/A denotes the locally deployed Dr. Kernel, excluded from token accounting.

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
FI01	0.098	0.085	1.040	2.380	5.390	N/A	3.730	0.360
FI02	0.119	0.087	1.140	0.510	5.320	N/A	5.120	0.440
FI03	0.154	0.146	2.200	1.620	4.370	N/A	6.030	0.520
FI04	0.092	0.078	9.060	5.440	12.460	N/A	3.080	0.350
FI05	0.126	0.116	5.900	1.280	5.440	N/A	4.720	0.300
FI06	0.122	0.115	2.760	2.060	0.600	N/A	5.160	0.430
FI07	0.099	0.072	2.580	2.610	7.750	N/A	5.460	0.390
FI08	0.130	0.128	1.820	1.910	4.170	N/A	5.830	0.480
FI09	0.109	0.106	5.990	3.220	5.680	N/A	6.260	0.350
FI10	0.055	0.031	0.710	0.970	1.530	N/A	2.620	0.270
Numeric	10/10	10/10	10/10	10/10	10/10	N/A	10/10	10/10
Total	1.104	0.964	33.200	22.000	52.710	N/A	48.010	3.890
Mean per task	0.110	0.096	3.320	2.200	5.271	N/A	4.801	0.389

Table 30: Token usage for SOL-ExecBench on A800, in millions of tokens (M). All listed tasks are included regardless of performance status. N/A denotes the locally deployed Dr. Kernel, excluded from token accounting.

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
L1/002	0.089	0.049	8.760	3.190	3.020	N/A	2.170	0.440
L1/005	0.090	0.056	1.250	2.080	4.920	N/A	1.310	0.340
L1/007	0.066	0.036	9.610	8.720	1.240	N/A	1.210	0.360
L1/008	0.072	0.033	2.080	0.860	1.390	N/A	0.930	0.270
L1/018	0.094	0.057	3.740	2.810	0.900	N/A	1.600	0.400
L1/020	0.101	0.096	5.360	0.750	4.820	N/A	2.260	0.440
L1/053	0.081	0.049	1.070	0.560	1.440	N/A	1.510	0.270
L1/058	0.102	0.038	3.620	5.810	1.690	N/A	1.210	0.330
L1/070	0.094	0.051	3.500	1.270	1.080	N/A	1.780	0.310
L1/092	0.105	0.089	7.760	1.550	1.490	N/A	1.550	0.380
L2/012	0.072	0.061	20.780	4.810	6.500	N/A	1.770	0.460
L2/015	0.082	0.086	54.950	24.520	36.230	N/A	1.430	0.380
L2/030	0.082	0.050	7.440	0.730	1.430	N/A	1.240	0.290
L2/036	0.137	0.114	7.590	3.780	0.660	N/A	3.420	0.800
L2/040	0.153	0.196	9.400	6.560	33.450	N/A	3.350	0.550
L2/043	0.116	0.122	1.510	10.450	19.150	N/A	2.430	0.410
L2/049	0.092	0.058	2.140	4.990	21.190	N/A	1.770	0.360
L2/051	0.129	0.108	16.510	8.020	0.220	N/A	3.020	0.500
L2/080	0.114	0.095	5.500	7.040	1.040	N/A	1.370	0.510
Numeric	19/19	19/19	19/19	19/19	19/19	N/A	19/19	19/19
Total	1.871	1.444	172.570	98.500	141.860	N/A	35.330	7.800
Mean per task	0.098	0.076	9.083	5.184	7.466	N/A	1.859	0.411

Table 31: Token usage for SOL-ExecBench on H100, in millions of tokens (M). All listed tasks are included regardless of performance status. N/A denotes the locally deployed Dr. Kernel, excluded from token accounting.

Task	GPT	Claude	Codex	CC	KDA	DrK	KS	Ours
L1/002	0.111	0.029	6.030	1.500	2.040	N/A	2.700	0.390
L1/005	0.084	0.084	2.320	1.550	4.060	N/A	2.120	0.420
L1/007	0.073	0.032	4.290	2.240	0.350	N/A	1.880	0.270
L1/008	0.073	0.031	5.360	0.650	3.930	N/A	1.950	0.360
L1/018	0.048	0.053	1.680	16.270	1.730	N/A	2.480	0.450
L1/020	0.085	0.063	3.010	1.200	0.940	N/A	2.710	0.350
L1/053	0.084	0.012	2.220	0.340	6.270	N/A	2.420	0.310
L1/058	0.077	0.037	0.570	1.510	3.920	N/A	1.530	0.390
L1/070	0.092	0.086	1.240	1.590	3.670	N/A	2.510	0.220
L1/092	0.106	0.062	1.840	0.710	4.970	N/A	2.120	0.330
L2/012	0.086	0.062	12.620	5.990	1.370	N/A	2.450	0.420
L2/015	0.083	0.051	3.950	6.630	1.010	N/A	2.060	0.430
L2/030	0.077	0.054	1.720	0.850	4.790	N/A	2.470	0.240
L2/036	0.138	0.080	2.000	15.410	0.650	N/A	2.910	0.790
L2/040	0.140	0.101	3.010	1.760	2.150	N/A	4.620	0.610
L2/043	0.044	0.185	7.430	11.110	4.070	N/A	3.610	0.360
L2/049	0.076	0.059	4.150	2.420	6.020	N/A	2.910	0.290
L2/051	0.092	0.120	6.470	10.520	0.500	N/A	2.730	0.550
L2/080	0.107	0.162	10.500	4.180	3.070	N/A	2.500	0.590
Numeric	19/19	19/19	19/19	19/19	19/19	N/A	19/19	19/19
Total	1.676	1.363	80.410	86.430	55.510	N/A	48.680	7.770
Mean per task	0.088	0.072	4.232	4.549	2.922	N/A	2.562	0.409

The A800 DSA token entries use the corrected trace totals from Section 3.2: 1.57M for Codex, 13.83M for KDA, and 1.62M for K-Search. Totals and means use these corrected values.

The client logs input and output tokens separately, including completed regeneration and repair calls as described in Appendix C.6. The reported main-evaluation records contain total tokens only; monetary cost and end-to-end search time are not inferred from those totals.

E.3 Detailed Ablation Results

Tables 32 and 33 report the best correctness-valid component-ablation results on A800. Local-only candidate identifiers and full-workload pass counts are included. Per-task speedups are relative to eager for both benchmarks; Appendix C.4 specifies the configurations.

Structure-only solves all ten FlashInfer and nineteen SOL-ExecBench tasks. Its eager-relative geometric means are $48.907724\times$ and $0.972889\times$, respectively; the SOL L1 and L2 means are $1.119758\times$ and $0.832180\times$. Applying the FlashInfer expert factors gives $2.413074\times$, reported as $2.41\times$ in Section 5.3.

Local-only solves all ten FlashInfer tasks and twelve of nineteen SOL-ExecBench tasks. Its eager-relative geometric means are $27.683933\times$ and $2.245414\times$; expert normalization gives $1.365906\times$ for FlashInfer. SOL-ExecBench correctness is $12/19 = 63.16\%$, a gap of 36.84 percentage points to the full method. Confirmed failures receive no numerical speedup and are excluded from the geometric mean.

Table 32: FlashInfer-Bench component-ablation speedups on A800, relative to eager. F denotes an author-confirmed failure and is not assigned a numerical speedup. Candidate and workload correctness columns refer to local-only.

Task	Structure-only	Local-only	Candidate	Correctness
FI01	36.540000	1.912840	Iter_6	23/23

Continued on next page

Table 32 (continued)

Task	Structure-only	Local-only	Candidate	Correctness
FI02	282.700000	280.141879	Iter_1	54/54
FI03	164.399304	133.365689	Iter_6	100/100
FI04	0.540000	0.589825	Iter_4	43/43
FI05	387.850000	482.150570	Iter_8	48/48
FI06	458.700000	365.193252	Iter_1	38/38
FI07	11.800000	2.673665	Iter_1	21/21
FI08	63.840000	42.699133	Iter_5	47/47
FI09	154.640000	84.884633	Iter_5	38/38
FI10	4.120000	3.676151	Iter_1	14/14
Numeric final results	10/10	10/10		
Geometric mean	48.907724	27.683933		

Table 33: SOL-ExecBench component-ablation speedups on A800, relative to eager. F denotes an author-confirmed failure and is not assigned a numerical speedup. Candidate and workload correctness columns refer to local-only.

Task	Structure-only	Local-only	Candidate	Correctness
L1/002	0.399533	F	–	–
L1/005	1.045344	1.374467	Iter_4	16/16
L1/007	0.110063	0.098028	Iter_1	16/16
L1/008	2.230566	1.296083	Iter_1	16/16
L1/018	8.257448	11.894666	Iter_1	13/13
L1/020	0.106669	1.389974	Iter_4	15/15
L1/053	3.582576	61.264891	Iter_4	12/12
L1/058	1.612954	F	–	–
L1/070	10.359535	67.132609	Iter_5	14/14
L1/092	0.573236	1.198626	Iter_5	16/16
L2/012	0.018771	0.689338	Iter_1	16/16
L2/015	0.058479	F	–	–
L2/030	3.568861	0.959253	Iter_3	16/16
L2/036	24.728007	F	–	–
L2/040	4.772852	F	–	–
L2/043	5.608802	F	–	–
L2/049	2.758121	F	–	–
L2/051	0.258784	0.557253	Iter_1	16/16
L2/080	0.103407	3.132098	Iter_3	16/16
Numeric final results	19/19	12/19		
Geometric mean	0.972889	2.245414		

E.3.1 Cumulative Coverage of Generated Structures

Figure 4 uses five structures from one generation sequence per task, in generation order rather than reviewer-score order. Each structure is evaluated with ten local optimization rounds using the main evaluation settings. Let $v_{t,j}$ be the best correctness-valid performance obtained for task t from structure j. The task-level cumulative coverage is

$$C_t(n) = 100\,\frac{\max_{1\le j\le n} v_{t,j}}{\max_{1\le j\le 5} v_{t,j}}, \qquad n \in \{1,\dots,5\}. \tag{6}$$

This statistic measures how quickly the generated sequence exposes its best-performing alternative. It is monotone and reaches 100% at five structures by construction; it does not measure the outcome of separate top-m pipeline runs at each structural budget.

For each benchmark group $\mathcal{T}$, the colored curve is the geometric mean of these task-level percentages:

$$C_{\mathcal{T}}(n) = 100\exp\left[\frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}}\log\left(\frac{\max_{1\le j\le n} v_{t,j}}{\max_{1\le j\le 5} v_{t,j}}\right)\right]. \tag{7}$$

The groups are FlashInfer-Bench, SOL-ExecBench L1, and SOL-ExecBench L2. Normalization is performed within each task before geometric aggregation, so tasks with larger absolute speedups do not dominate the curve. The performance maxima use correctness-valid implementations.

F Representative Optimization Cases

GDN Decode (FI02) illustrates structural review, contract validation, and repair. These recorded examples are separate from the main-table results and DSA replay.

F.1 From Structural Review to Realization

The five GDN Decode plans receive recorded structural scores of 3.10, 3.75, 2.90, 3.35, and 3.70 in generation order. The selected roots are therefore `structure_2` and `structure_5`, corresponding to `local_1` and `local_2`. The former uses a fused ordered recurrence with one owner per batch and value head. The latter separates shared recurrence-input production from value-channel-owned recurrence and output computation.

The first root's `Iter_0` passes all 54 workloads at $322.4874\times$ eager-relative speedup. Its next two candidates are rejected before execution for changing `base_strategy` and `logical_kernels`, respectively. This snapshot covers selection, realization, and local rejection, but ends before a validated final dispatch result.

F.2 Structure-preserving and Structure-changing Decisions

The recorded plans distinguish computational organization from physical tuning. GDN `structure_1` materializes recurrent-state trajectories before a separate decode-finalization stage. In `structure_2`, a batch/value-head owner performs the ordered recurrence, decoded outputs, and final-state write in one logical stage. Removing the cross-stage state trajectory and changing its ownership changes G and P, not merely a tile or launch parameter.

The saved `structure_2` implementation contract passes the supplied validation function, as does a copy with an added unprotected physical-parameter field. Removing its semantic grid instead raises `local implementation changed protected structure field: semantic_grid`. These executable checks test declaration consistency, not GPU correctness or code-semantic equivalence.

F.3 Failure, Repair, and Incumbent Retention

The GDN Decode local-only trace contains a completed repair. In `Iter_1/attempt_1`, Triton compilation fails because the kernel accesses the ordinary Python global `_NUM_V_HEADS`. The second attempt passes the head counts and head size as explicit `tl.constexpr` arguments instead. The following source excerpts show the change without altering the head-index calculation.

```
head = batch_head % _NUM_V_HEADS
batch = batch_head // _NUM_V_HEADS
```

Listing 15: Recorded GDN Decode repair: global access before repair.

```
NUM_Q_HEADS: tl.constexpr,
NUM_V_HEADS: tl.constexpr,
HEAD_SIZE: tl.constexpr,

head = batch_head % NUM_V_HEADS
batch = batch_head // NUM_V_HEADS
```

Listing 16: Recorded GDN Decode repair: explicit compile-time arguments.

`attempt_2` passes all 54 workloads, and the accepted `Iter_1` records $280.1419\times$ eager-relative speedup. Correctness-valid `Iter_2` and `Iter_3` are slower at $86.5338\times$ and $207.9975\times$, respec-

tively. The final result retains `Iter_1`, distinguishing successful repair from subsequent tuning that fails to improve the incumbent.