KERNELCOMPASS: TRAVERSING THE FEASIBLE RESOURCE SPACE OF THE TARGET GPU TOWARD A TASK-CONTRACTED OBJECTIVE

Anonymous authors
Paper under double-blind review

ABSTRACT

GPU kernel optimization must account for both the hardware and the user's goal: a kernel that is faster in isolation need not make the model faster under its intended workload. We present KernelCompass, an agent framework that combines hardware resource limits, measured bandwidth and compute throughput, and tuning records to adapt kernel resource demand to the target device. Within one algorithm structure, changing a single parameter while holding the others fixed yields paired performance and resource feedback that guides rewrites to relieve resource constraints while using available headroom. Rewritten structures are retuned, and the new measurements guide the next round. At the model level, a task contract fixes the metric, workload, quality conditions, and acceptance criterion before optimization starts, and drives both the choice of operator groups and the acceptance of candidates under the native workload. With GLM-5.3-max as the base model, KernelCompass is $1.014\times$ to $3.144\times$ faster than the strongest of four baseline frameworks in each of the 18 combinations of six KernelBench tasks and three GPUs. On Qwen3-4B on an RTX 4090, separate goal-specific optimizations reduce time to first token (TTFT) by 3.56%, raise single-stream throughput by 12.51%, and raise concurrent throughput by 9.29% under the respective tested workloads.

1 INTRODUCTION

Kernel optimization has long been a search problem under a fixed task. Classical autotuning and schedule-search systems tune a given implementation's parameters—tile sizes, warp counts, pipeline stages—or explore transformations within a predefined schedule space (Ragan-Kelley et al., 2013; Chen et al., 2018; Zheng et al., 2020; Tillet et al., 2019), yet judge every candidate by the same criterion on the same device. LLM agents now generate and rewrite kernel source directly, organizing the search through feedback loops (Wang et al., 2025; Cheng et al., 2026), maintained plans (Cao et al., 2026), program populations (Novikov et al., 2025; Sharma, 2025; Lange et al., 2026; Wiedemann et al., 2026; Nichols et al., 2026), or statistical budget allocation (Ran et al., 2026; Gai et al., 2026; Che & Peng, 2026). Whatever the search rule, the formulation is shared: the kernel defines the task, correctness gates candidates, and hardware constrains their execution.

However, the same model running on the same hardware may serve different purposes—whether to trade speed for memory in local usage, to cater to more parallel sessions with a higher total generation throughput, or to serve an interactive scenario where a postponed reply is intolerable. The kernels worth rewriting and the candidates worth keeping change with the user's destination. A uni-directional optimization toward making the most of the hardware is hence insufficient in the coming era.

Toward a task–hardware dual-adaptive kernel optimization, we introduce **KernelCompass**, which guides kernel optimization following the user's destination. At the kernel level, it measures what each resource dimension of the device can supply and builds a resource profile from parameter-search records to identify what limits the current structure. Paired performance and resource slopes, measured by changing one parameter while holding the others fixed, guide rewrites that reduce pressure on constrained resources and use resources with headroom. The rewritten structure is

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.

then retuned, and the new measurements guide the next round. At the model level, the user's one-sentence goal is compiled into a fixed task contract—metric, workload, correctness gates, acceptance criterion—that selects kernel groups and accepts candidates under the native workload rather than isolated microbenchmarks. With GLM-5.3-max as the base model, KernelCompass is faster than the best of four agentic frameworks in all eighteen combinations of six KernelBench (Ouyang et al., 2025) tasks and three GPU generations ($1.014\times$–$3.144\times$); on Qwen3-4B, optimizing separately for each goal improves TTFT, single-stream, and concurrent throughput by 3.56%, 12.51%, and 9.29%.

We make three contributions:

1. **An agent framework for task–hardware adaptation.** We propose a kernel optimization agent framework that measures what each resource dimension of the target device can supply and reads the tuning records to diagnose where a kernel's resource demand does not match that supply. Structural rewriting and retuning align the demand with the target hardware.
2. **Interventional guidance for structural rewriting.** Measured fixed-partner contrasts provide paired resource and performance feedback to guide the agent's structural rewrite decisions. Signal-masking ablations test the added resource information, and independent retuning tests retained candidate benefits.
3. **Task-driven kernel selection and acceptance.** Task contracts connect declared workloads, correctness gates, and native objectives to the selection of kernel groups and model-level candidate acceptance. Qwen3-4B studies demonstrate model-level gains and objective specificity.

Section 2 presents the background and problem, Section 3 describes the method, Section 4 reports experiments, and Section 5 concludes.

2 Background and Problem Formulation

2.1 Algorithm Structures and Search Objective

We first consider a single operator running in isolation. Given a reference implementation, the input shapes, and a target GPU, the objective is to find a faster implementation that meets the task's correctness requirements. Implementations can differ in how they split the computation, where they place data, and how they assign work to threads; we call a class of implementations that share these choices an *algorithm structure* S, and the vector of adjustable settings within it—tile size, warp count, pipeline stages, and so on—its *parameters* $\boldsymbol{\theta}$. Let $\Theta(S)$ be the set of feasible parameter vectors of S under the current task and hardware, and $T(S, \boldsymbol{\theta})$ the latency of one configuration; the shortest latency a structure can reach under ideal tuning is

$$T^*(S) = \min_{\boldsymbol{\theta} \in \Theta(S)} T(S, \boldsymbol{\theta}). \tag{1}$$

The feasible structures (those with non-empty $\Theta(S)$) form the space $\mathcal{S}$ (Appendix A.2), and for a single operator the objective is to find $S \in \mathcal{S}$ with $T^*(S)$ as small as possible, rather than a good parameter vector for one fixed implementation; tuning in practice reports only the best configuration actually tested, not the limit in Eq. (1). When the optimization target is a whole model, operator latency cannot stand in for the model-level performance the user cares about; a model-level objective under the target workload is then needed to decide which operators to optimize and how to judge a rewrite. Before stating these requirements precisely, we review how existing frameworks organize their search.

2.2 Related Work

LLM-driven kernel optimization involves two related decisions: how to change the code, and how to organize a series of attempts—which past candidates to keep, where the next attempt starts, and how much evaluation it deserves. Existing frameworks organize attempts through three complementary mechanisms: feedback-driven planning, evolutionary populations, and statistical allocation. We summarize each briefly; Appendix A.1 gives a fuller account.

Feedback-driven planning. GEAK runs a generate–test–reflect loop (Wang et al., 2025), KernelAgent adds hardware profiling and bottleneck diagnosis (Cheng et al., 2026), and K-Search maintains explicit optimization plans that several implementations can realize (Cao et al., 2026). Two-stage tuners have also used resource measurements to organize structural rewriting and retuning (Qu et al., 2026), and compiler diagnostics have been used to ground bottleneck analysis (Chen et al., 2026).

Evolutionary populations. AlphaEvolve and OpenEvolve evolve program populations with MAP-Elites archives, grids of cells that each keep their best candidate (Mouret & Clune, 2015; Novikov et al., 2025; Sharma, 2025); ShinkaEvolve uses parent sampling and novelty filtering without a fixed grid (Lange et al., 2026); KernelFoundry and R^3 combine MAP-Elites with parameter search; KernelFoundry also builds gradient-like prompts from parent–child behavior and performance differences (Wiedemann et al., 2026; Nichols et al., 2026).

Statistical allocation. KernelBand chooses candidate groups and strategies with hardware-aware multi-armed bandits (Ran et al., 2026), KernelPro expands branches with Monte Carlo tree search (Gai et al., 2026), and KernelBrain raises the evaluation budget of promising candidates progressively (Che & Peng, 2026).

Our work. MAP-Elites archives require a descriptor partition, whose granularity can affect search: on Matmul-Swish-GroupNorm, coarse and fine partitions gave 22.6% and 5.4% higher best latencies than a medium one at a fixed seed and budget (Appendix A.3). Current latency alone does not identify which structural changes may yield further gains. KernelCompass uses controlled parameter comparisons within one structure, together with resource constraints, to guide those changes. It keeps no descriptor cells (Appendix A.4); each starting candidate and its descendants form a *line*. At the model level, a task contract guides operator-group selection and candidate acceptance.

2.3 Problem Statement

Deciding whether to keep tuning or to change the structure requires knowing the resource that currently binds the structure and the cost of relieving it; lowering resource usage is not the aim in itself, and a structurally constrained implementation is not an operator at the hardware limit. For a whole model, neither operator latency nor isolated speedup can stand in for the model-level gain. Three requirements follow: (i) identify resource constraints and assess the trade-offs of relieving them; (ii) select operator groups by the objective under the target workload; (iii) accept replacements under the native workload. Section 3 addresses them in turn.

3 Method

3.1 From Tuning Records to Structural Decisions

The three requirements of §2.3 map onto the stages below (Figure 1). For requirement (i), the framework combines hardware resource limits and measured throughput with the tuning record—latencies, register counts, shared-memory usage, register spills, and infeasible configurations—and alternates tuning and rewriting to improve the fit between resource demand and hardware (§3.2–§3.4). For requirements (ii) and (iii), it uses a task contract fixed before optimization begins to group modules under the target workload, select the operator groups to optimize, and accept the final candidate once installed back into the model (§3.5). The framework handles parameter sampling, compilation, timing, and round scheduling; six dedicated agents handle code generation and semantic judgment, and the framework checks their outputs (Appendix B.1).

3.2 Locating the Binding Resource Dimension

When parameter search stops improving, the first question is why. Latency may have stopped falling because the structure is already close to what the hardware can deliver, or because one resource dimension has hit its limit while others still have headroom; both cases look the same from the outside, and telling them apart requires knowing how much of each resource dimension is used and how much the device can supply.

Before optimization, the framework measures memory bandwidth and compute throughput at each precision tier and uses the device resource capacities as limits. Resource usage for candidate con-

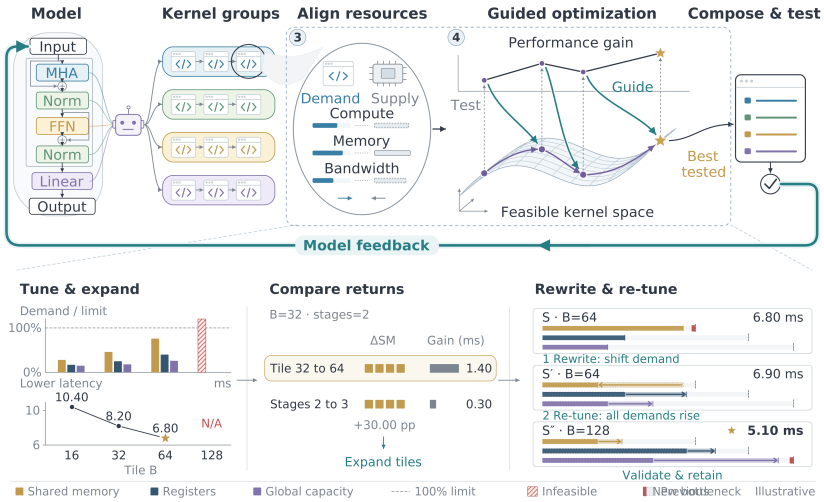

Figure 1: Method overview. Upper row: one optimization run, from running the model under the target workload, through operator grouping and group selection, to the loop of parameter search and structural rewriting, and finally acceptance in the model. Lower row: the three stages of the loop. In the parameter-search panel, bars are the usage of each resource dimension relative to the device limit and the curve is latency; the largest parameter value exceeds the limit and cannot run. The paired-comparison panel contrasts two parameter directions by their latency improvement and change in resource usage (Appendix B.6). The rewrite-and-retune panel plots resource demand before and after each step; the red end marks the binding dimension (Appendix B.7). Values are illustrative, not a measured trajectory.

figurations comes from parameter search. Let d index the resource dimensions that have a capacity limit (such as shared memory), and let R_d be the capacity of dimension d on the device. Let $r_d(S, \boldsymbol{\theta})$ be the usage of configuration $(S, \boldsymbol{\theta})$ in dimension d. A configuration can run only if

$$r_d(S, \boldsymbol{\theta}) \leq R_d, \quad \forall d. \tag{2}$$

Value combinations that violate Eq. (2) are marked infeasible by the framework or the compiler, and these judgments also enter the search record. If latency keeps falling along a parameter direction but the next values fall outside $\Theta(S)$ because a resource is exceeded (the parameter-search panel of Figure 1), that direction becomes a starting point for structural rewriting. The framework ranks directions by the size of the latency decrease near the feasibility boundary and handles the largest decreases first.

A constraint found this way may come from the parameter table rather than from the device: the Parameterization Agent lists discrete candidate values for each parameter, and if 128 is rejected right after 64, the search does not know whether 96 is feasible or faster. When the relevant resource still has headroom, the framework therefore first adds values along that direction and runs one more round of search with the structure and other parameters fixed (Appendix B.2). At this point the binding dimension is known, but not whether relieving it will pay off; when several limits bind at once, the next stage decides which one is worth relieving first.

3.3 Assessing Resource Trade-offs with Paired Slopes

Once a limit is located, the cost of trading resources must still be judged: lowering usage does not guarantee lower latency, since a smaller tile may reduce register demand but add loads or synchronization. Within one structure, the framework therefore picks two configurations along the chosen direction, one near the limit and one far from it, with all other parameters identical so that the observed difference does not contain their effect, and measures latency and per-dimension resource usage for both. This gives the rate of change of latency with respect to the parameter, $\Delta T / \Delta \theta_k$, where θ_k is the k-th component of $\boldsymbol{\theta}$ being adjusted; we call this ratio the *performance slope* of the

direction. It also gives the rate of change of latency with respect to usage in dimension d,

$$\frac{\Delta T}{\Delta r_d} = \frac{T(S, \boldsymbol{\theta}^{\text{near}}) - T(S, \boldsymbol{\theta}^{\text{far}})}{r_d(S, \boldsymbol{\theta}^{\text{near}}) - r_d(S, \boldsymbol{\theta}^{\text{far}})}, \tag{3}$$

which we call the *resource slope* of dimension d; here $\boldsymbol{\theta}^{\text{near}}$ and $\boldsymbol{\theta}^{\text{far}}$ are the two configurations, and the ratio is not computed when the resource difference is zero.

The two slopes guide rewriting together. A latency reduction accompanied by a change in the constrained resource provides evidence for a rewrite targeting that resource; a reduction with no resource change does not yet identify which limit to relieve. These ratios describe responses along a tested parameter change, not the independent effect of changing one resource. The paired-comparison panel of Figure 1 contrasts tile size and pipeline stages (Appendix B.6). Because neither slope is normalized, slopes cannot be compared in magnitude across directions; directions are ranked as in §3.2.

3.4 Rewriting the Structure and Realigning Resources

Before a rewrite, the Resource Analysis Agent reads the tuning record and judges which dimension is binding. The Structural Rewrite Agent then rewrites the structure from five inputs: the source code with its best tested configuration, the usage of each dimension, the parameter direction where the limit lies, the resource slope from Eq. (3), and the directions already tried on this structure without improvement. The rewrite aims to reduce demand on the constrained resource by using headroom in other resources. Retuning tests whether this trade-off improves performance. A candidate that fails the correctness checks goes to the Repair Agent and is checked again.

After a rewrite lowers the demand of the binding resource, previously infeasible parameter configurations may become available, but the previously best parameters may no longer help and may even be slower. The framework therefore retunes the new structure (rewrite-and-retune panel of Figure 1; Appendix B.7) and decides later rewrites from the updated resource feedback, until the budget is exhausted or several consecutive rounds bring no clear gain.

A rewrite does not always succeed, so the framework keeps several lines at once. The Candidate Generation Agent opens one line per starting candidate, and lines not yet rewritten receive priority for the next round. If all lines stop improving, the Exploration Agent proposes a different candidate and opens a new line.

3.5 Choosing What to Optimize and What to Accept

Operator latency cannot settle model-level trade-offs. The user states a goal, such as "make the first token faster." An agent reads the goal and model and produces a task contract specifying the metric and unit, optimization direction, workload and inputs, and quality conditions. It then writes the measurement code from these fields. An optimization objective is thus defined jointly by the metric and its workload, since a request that decodes a few tokens and one that processes a long input have different hot spots. The contract is fixed before optimization begins, every later model-level measurement follows it, and the measured value is written J. Local parameter screening still uses operator latency T, while model-level feedback and final acceptance use J; a candidate is not rejected for failing to speed up locally.

Selecting what to optimize begins with one real run of the model under the target workload, which records the execution path of each module, its source code and weight-tensor shapes, and the time spent in the GPU kernels it dispatches. Modules with identical source code and weight-tensor shapes form one *operator group* (the grouping stage in Figure 1); the group shares one replacement implementation, and correctness after replacement is checked module by module. Groups that cannot be packaged as a standalone task—for example, those that run only with a key–value (KV) cache attached—are excluded. For each remaining group, the framework binds a numerically equivalent replacement, measures the change in J under the same contract, and uses this change to allocate the search budget. After a candidate is installed in the model, it is accepted according to the metric J and the quality conditions of the contract (§4.1); local timing is also used to explain where the gain comes from.

4 EXPERIMENTS

The experiments ask whether rewriting aligns resource use with hardware across tasks and devices (§4.2), what resource feedback adds to structural rewriting (§4.3), and whether goal-driven selection and acceptance hold at the model level (§4.4).

4.1 EXPERIMENTAL SETUP

Tasks and devices. We select six KernelBench tasks and refer to them by abbreviated problem titles: ConvT2d-Mish, Matmul-Swish, ConvT3d-GroupNorm, MBConv, CausalAttention, and Mamba2-SSD (KernelBench IDs, task contents, and resource statistics are in Appendix C.6). The tasks differ in data volume, computation, and operation structure. Matmul-Swish and ConvT3d-GroupNorm have similar computational workloads, but Matmul-Swish has about $9.5\times$ the combined input, parameter, and output volume of ConvT3d-GroupNorm. The devices are an A800 80GB, an RTX 4090, and an RTX 5090, from the Ampere, Ada Lovelace, and Blackwell generations, which differ in memory, bandwidth, compute, and shared-memory capacity; running one task across devices tests adaptation when supply changes, running different tasks on one device tests adaptation when demand changes, and together they give 18 task–GPU combinations.

Baselines and timing. We compare against KernelFoundry, K-Search, KernelBand, and ShinkaEvolve (Wiedemann et al., 2026; Cao et al., 2026; Ran et al., 2026; Lange et al., 2026). The cross-hardware main experiment uses GLM-5.3-max as the base model; for ConvT2d-Mish and MBConv on the RTX 4090 we additionally run DeepSeek-v4-pro-max and GPT-5.6-sol-xhigh. Our method uses a 12-hour wall-clock budget per search with 40 trials per parameter space (Appendix B.2) and produced at most 19 structural candidates; the four baselines used 128 candidate attempts per task, median cumulative API-call times range from 29.0 hours (K-Search) to 57.9 hours (KernelFoundry). PyTorch Eager and `torch.compile` with TensorFloat-32 (TF32) serve as two execution baselines. We also report the speedup over the *strongest framework*, the fastest of the four baselines on the same task, device, and base model. All three speedups divide the baseline time by our time. Our final candidates are measured 100 times in an independent process, and the median is reported. The feedback ablation (§4.3) runs on the RTX 4090; there, we measure parent and child over three timing blocks of 100 samples each and compare the medians of the block medians.

Correctness. A candidate may pass against either the TF32 or full-FP32 reference output. For each reference, both conditions must hold: more than 99% of elements have relative error below 1%, and flattened-output cosine similarity is at least 0.99985. A third path follows PyTorch's `torch._dynamo.utils.same()` (Appendix B.3). Output precision and shape must match the reference; internal precision is open to rewriting.

Model workloads. Model-level experiments use Qwen3-4B on the RTX 4090, optimizing TTFT, single-stream throughput, and concurrent throughput separately. Each objective fixes the input length, output length, and request pattern, uses greedy decoding with seed 0, and warms up and resets the KV cache and model state before measurement. TTFT uses 4 requests of 4096 tokens with 1 output token each; single-stream throughput uses 2 requests of 256 tokens with 128 output tokens each; concurrent throughput issues 8 requests of 512 tokens simultaneously, 128 output tokens each. Local outputs use $\mathrm{atol} = \mathrm{rtol} = 0.02$; the declared per-prompt logits relative L2 tolerance is 0.01, with the non-tightening reference-based adjustment described in Appendix C.2. The paired mean negative log-likelihood (NLL) change must not exceed 0.02 nat/token.

4.2 ALIGNING KERNEL RESOURCE USE WITH HARDWARE

4.2.1 PERFORMANCE ACROSS TASKS, GPUS, AND BASE MODELS

Table 1 covers the 18 task–GPU combinations with GLM-5.3-max. In all 18, our latency is below that of the strongest framework: $3.31\times$–$17.40\times$ over Eager, $1.64\times$–$10.13\times$ over Compile+TF32, and $1.014\times$–$3.144\times$ over the strongest framework.

A stronger base model is another possible explanation for the lead, so Appendix C.1 (Table 7) repeats ConvT2d-Mish and MBConv on the RTX 4090 with GLM-5.3-max, DeepSeek-v4-pro-max, and GPT-5.6-sol-xhigh: on MBConv we lead the strongest framework under all three ($1.303\times$, $1.866\times$,

Table 1: Latency and three speedups across tasks and GPUs, with GLM-5.3-max as the base model. "Strongest" is the shortest valid latency among the four frameworks in the same row; all three speedups use our latency as the denominator. The final candidates of all methods were re-evaluated in the same environment. The lowest latency in each row is in bold; NV denotes that no correct candidate was produced.

	Baseline framework latency (ms)				Ours			
Task	Kernel Foundry	K-Search	Kernel Band	Shinka Evolve	Latency (ms)	vs. Eager	vs. Compile+ TF32	vs. Strongest
A800 80GB								
ConvT2d-Mish	9.570	4.780	11.254	2.829	**2.790**	12.87×	2.85×	1.014×
Matmul-Swish	2.670	4.420	4.200	2.649	**1.050**	15.81×	2.91×	2.523×
ConvT3d-GroupNorm	15.000	4.330	6.855	11.583	**1.890**	15.24×	1.78×	2.291×
MBConv	8.010	5.630	5.994	5.390	**4.320**	3.43×	1.64×	1.248×
CausalAttention	20.100	3.630	20.801	4.598	**2.350**	12.77×	3.12×	1.545×
Mamba2-SSD	0.822	2.180	NV	13.951	**0.799**	17.40×	10.13×	1.029×
RTX 4090								
ConvT2d-Mish	3.912	4.220	4.205	36.249	**3.430**	13.88×	3.88×	1.141×
Matmul-Swish	5.760	2.390	6.168	2.768	**1.250**	4.76×	4.12×	1.912×
ConvT3d-GroupNorm	5.510	5.860	6.096	4.974	**2.450**	4.33×	2.20×	2.030×
MBConv	5.222	4.501	4.990	7.612	**3.4545**	4.52×	3.30×	1.303×
CausalAttention	8.880	9.287	10.193	8.686	**2.7628**	7.78×	3.98×	3.144×
Mamba2-SSD	1.414	2.016	NV	20.215	**1.390**	14.82×	8.92×	1.017×
RTX 5090								
ConvT2d-Mish	2.990	2.770	4.353	20.931	**2.530**	11.74×	2.95×	1.095×
Matmul-Swish	4.840	3.610	4.919	1.679	**1.520**	3.31×	1.85×	1.105×
ConvT3d-GroupNorm	4.140	3.050	4.922	3.402	**1.660**	9.70×	2.14×	1.837×
MBConv	3.530	3.290	3.797	3.163	**2.140**	5.65×	3.10×	1.478×
CausalAttention	6.320	2.240	8.013	3.022	**1.840**	8.32×	3.92×	1.217×
Mamba2-SSD	0.845	6.460	12.330	8.285	**0.802**	15.34×	9.50×	1.054×

1.118×), and on ConvT2d-Mish under GLM and DeepSeek (1.141×, 1.177×) while trailing K-Search under GPT-5.6 (0.827×).

4.2.2 Aligning Resource Use through Rewriting

The results above come from rewrites that do not lower resource usage across the board; they reshape demand into a range where the hardware can run more work in parallel, as two cases with opposite directions of change illustrate (Figure 3 in Appendix C.4). On each streaming multiprocessor (SM), registers and shared memory are a fixed allowance: the more one thread block takes, the fewer blocks can be resident. The parent structure of MBConv loads the whole K dimension of the expand GEMM (general matrix multiplication) into shared memory at once, so registers hit the per-thread limit of 255 with 108 register spills and only one block fits per SM; replacing this full-K load with K-loop tiling (loading K in chunks) and FP16 intermediate storage, then retuning removes the spills, fits three blocks per SM, and cuts full-task latency by 51.6%. The first rewrite of ConvT2d-Mish, by contrast, *increases* shared memory and register usage while improving latency by 21.8%, and later candidates reach a cumulative 30.9% gain. Effective rewrites thus share no common direction of change; what they share is a match between demand in each resource dimension and the device's supply (Appendix C.7 compares the final results against a hardware performance reference). These cases show the outcomes of rewriting; the next section examines the role of feedback.

4.3 Performance and Resource Slopes for Structural Rewriting

4.3.1 Controlled Feedback Ablation

To test what paired resource feedback adds beyond the performance feedback already available, three groups of agents rewrite the same code from the same starting point; the only difference is the signals they receive. The *history-only* group receives the existing trial record, the parent's resource

Table 2: (a) Effect of feedback on structural rewriting. Values are child latency divided by parent latency, both re-measured in the same evaluation; lower is better, 1 means no improvement, and the lowest observed value for each starting point is in bold. (b) Transfer of RMSNorm candidates across objectives on Qwen3-4B. Each candidate is optimized for one objective and measured under all three workloads against that workload's baseline (TTFT 284.83 ms, single-stream 26.00 tokens/s, concurrent 197.46 tokens/s); positive values are improvements, and the best value for each measured objective is in bold. †: this TTFT measurement did not meet the quality conditions of the TTFT contract, so only the time change is listed; the candidate's results under the throughput workloads are unaffected.

(a) Feedback ablation (child/parent latency)

Starting point	Feedback group	Ratio
CausalAttention / 1	History-only	0.9748
	Performance-slope	**0.8042**
	Resource-slope	0.9735
CausalAttention / 2	History-only	0.8126
	Performance-slope	0.7941
	Resource-slope	**0.7797**
MBConv / 1	History-only	0.5517
	Performance-slope	0.6068
	Resource-slope	**0.4843**
MBConv / 2	History-only	0.4660
	Performance-slope	0.4685
	Resource-slope	**0.3906**

(b) Transfer across objectives

Measured objective	Optimized for	Gain
TTFT	TTFT	**+3.56%**
	Single-stream	$-0.05\%^{\dagger}$
	Concurrent	$-0.10\%^{\dagger}$
Single-stream	TTFT	-10.64%
	Single-stream	**+12.51%**
	Concurrent	+11.03%
Concurrent	TTFT	-14.17%
	Single-stream	+8.35%
	Concurrent	**+9.29%**

usage, and static constraints, but no new paired measurements. The *performance-slope* group adds paired performance feedback and its endpoint measurements. The *resource-slope* group further adds resource usage, differences, slopes, observation status, and endpoint-failure diagnostics. These are cumulative feedback settings, not isolated signal types. Comparing the performance-slope and resource-slope groups tests the added resource and diagnostic feedback package, not the resource slope alone (Appendix B.4). Starting points (a parent structure with its tuning state) come from MBConv and CausalAttention, two per task, numbered 1 and 2 in registration order; the three groups share the collected measurements, prompts, and parameter-sampling seed, each new structure gets 40 parameter trials, and because starting points differ in speed we report child latency over parent latency, both re-measured in the same evaluation. Table 2(a) gives the results: the resource-slope group beats the history-only group at all four starting points and the performance-slope group at three, which, together with the rewrite case analyzed next, supports using performance and resource feedback jointly.

4.3.2 From Measured Slopes to Rewrite Decisions

MBConv start 1 illustrates how the two feedback types are used together (full case in Appendix B.5). The parent expand GEMM uses a full-K single load with 255 registers, 108 spills, and 65536 B of shared memory, so register use and per-SM block residency constrain the implementation. The tested `SPLIT_M` change yields a larger latency reduction (performance slope -0.787) but no change in measured resource usage. The tested `BLOCK_N` change yields a smaller latency reduction (slope -0.031), accompanied by negative resource slopes for registers, spills, and shared memory. This suggests reducing simultaneous residency demands while retaining large-tile benefits; the slope magnitudes are not compared across parameters. The rewrites match these signals: the performance-slope group keeps the full-K load and only stores intermediates in FP16 (5.076 ms after retuning), while the resource-slope group uses K-loop tiling with FP16 intermediate storage, citing the one-block-per-SM residency, registers, and spills in its rationale (4.100 ms, 19.2% lower than the performance-slope candidate). The history-only group also proposed a K-loop scheme (4.672 ms), so the idea does not require resource slopes. These cases suggest that resource feedback can guide useful rewrites, but do not establish it as necessary or sufficient for success. Appendix B.5 also reports start 2, where the resource-slope group combines a GEMM replacement with fusion of the statistics computation.

Independent retuning. To check whether a rewrite's gain survives retuning, we additionally retune a full-K candidate and a K-loop candidate in their own parameter spaces under sampler seeds 2, 0, and 1 (evaluation and validation seeds fixed at 0); the K-loop candidate's latency stays 24.295%, 33.227%, and 39.269% lower, so under this comparison and tuning budget the gain is retained.

4.4 Goal-Driven Operator Optimization

4.4.1 Choosing Operators under Different Objectives

We optimize Qwen3-4B on the RTX 4090 for the three objectives and workloads of §4.1; grouping operators as in §3.5 yields three operator groups: RMSNorm (normalization, 73 modules), the multi-layer perceptron (MLP) projections gate_proj and up_proj (72 modules combined), and SiLU (activation, 36 modules). The most time-consuming operator is not necessarily the most worth optimizing: Table 9(a) in Appendix C.3 compares the three groups under the same TTFT objective, and the gate_proj/up_proj group accounts for about $6.33\times$ the CUDA time of RMSNorm (88822 versus 14041 μs), yet only RMSNorm achieved a distinguishable TTFT improvement in these attempts (+3.56%). One explanation is how each is implemented: the reference RMSNorm path consists of eight separate kernels with intermediates written to and read back from memory—overhead that fusion removes—whereas each of gate_proj and up_proj already runs as one library GEMM and the reference SiLU is already a single memory-bound vectorized kernel with no launches to merge and no intermediates to reuse.

The target workload also changes how much a group is worth optimizing: in Table 9(b), RMSNorm accounts for less than one sixth as much CUDA time under the single-stream workload as under the concurrent one (2383 versus 15036 μs), yet its objective gain is larger (+12.51% versus +9.29%). Cumulative CUDA time and objective gain do not rank the same way in these tests, so execution-time ranking alone is insufficient for allocating search effort; selection should also consider opportunities for improvement under the target workload.

4.4.2 Transfer across Objectives and End-to-End Acceptance

To test whether one candidate serves every objective, we optimize RMSNorm separately for each objective and measure all three candidates under all three workloads (Table 2(b)). In these tests, the TTFT candidate loses on both throughput objectives, the two throughput candidates transfer positively to each other, and on every objective the largest gain comes from the candidate optimized for it, which supports selecting candidates by the task contract rather than reusing one implementation across objectives. Acceptance must likewise be judged end to end: in an independent measurement (Appendix C.5), the TTFT candidate is slower than the reference in an isolated microbenchmark (speedup 0.980–0.988) yet lowers TTFT by 3.32% once installed in the model, so a rule of "operator speedup must exceed 1" would have discarded it.

5 Conclusion

KernelCompass uses tuning records to guide structural decisions, not only to pick the fastest tested configuration: rewriting aims to relieve resource constraints while preserving the benefits of the computation layout, and retuning tests whether the newly feasible configurations are faster. In the model workloads tested, a time-consuming operator group need not offer the largest improvement, and candidate rankings can change with the objective, which supports selecting operator groups and accepting replacements according to the declared objective under the native workload rather than by isolated operator latency.

Limitations and future work. The kernel-level results cover six KernelBench tasks and three GPUs under one wall-clock budget, and the feedback ablation rests on four starting points, so the size of the resource-feedback effect remains to be established at larger scale. The model-level evidence is limited to Qwen3-4B on the RTX 4090 under fixed workloads, and the paired slopes are finite differences that assume the other parameters can be held fixed, which coupled parameters may not allow. Other model families and GPU vendors, coupled parameters, and time-varying workloads are future work.

AI USE STATEMENT

We used generative AI tools in three ways. During preparation of the manuscript, we used LLM-based assistants to search and summarize related literature, to translate and polish the text, and to check LaTeX formatting; the authors read all cited works and verified every statement in the paper. During implementation, we used code-generation assistants to draft parts of the experiment harness and analysis scripts; we reviewed and tested this code. Finally, LLMs (GLM-5.3-max, DeepSeek-v4-pro-max, and GPT-5.6-sol-xhigh) are components of the system under study: they act as the code-generating agents inside KernelCompass and inside the baseline frameworks, as described in Section 3 and §4.1. The authors take full responsibility for all content of this paper. The authors confirm that this description matches the actual use of AI tools before submission.

ETHICS STATEMENT

This work involves no human subjects, personal data, or crowd workers. The main risks are computational cost, benchmark integrity, and the deployment of incorrect kernels. Kernel search consumes GPU time; we report budgets so that the cost is visible (Appendix B.2). To protect benchmark integrity, all comparisons re-evaluate the final candidates of every method in the same environment. Generated kernels can be numerically wrong in ways that a single test does not reveal, so every candidate must pass fixed correctness gates before acceptance (Appendix B.3), and model-level candidates are further checked against logits and NLL. Users deploying generated kernels should apply the same gates on their own workloads.

REPRODUCIBILITY STATEMENT

Section 3 describes the framework and Appendix B.1 lists the inputs and outputs of each agent. Appendix B.2 gives all search budgets, thresholds, seeds, and stopping criteria; Appendix B.3 gives the correctness gates and evaluation tiers; Appendix C.2 gives the three task contracts field by field, including the workload and quality conditions; Appendix C.6 gives the task statistics and the measured hardware calibration. We will release the source code of the framework, the agent prompts, the task contracts, the complete tuning records and final candidates for every task–GPU combination, the hardware calibration scripts, and the evaluation scripts used to produce every table.

REFERENCES

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-Search: LLM kernel generation via co-evolving intrinsic world model, 2026. URL https://arxiv.org/abs/2602.19128.

Shuai Che and Gang Peng. KernelBrain: Coarse-to-fine, budget-aware search for agentic GPU kernel optimization, 2026. URL https://arxiv.org/abs/2608.02611.

Dongjie Chen, Ping Zhao, Bohua Zhan, Yulong Wang, Shushu Chen, Liangjun Feng, Hao Zhou, Min Shen, Linmu Wang, Weijia Sheng, Xiangyu Wei, Weijie Ding, Jianhui Huang, and Yaoqing Gao. Compiler-grounded hierarchical diagnosis for LLM-based triton kernel optimization, 2026. URL https://arxiv.org/abs/2607.23089.

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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 578–594, 2018.

Kaiming Cheng, Laura Wang, Jack Khuu, Mark Saroufim, Wenyuan Chi, Jiannan Wang, and Joe Isaacson. KernelAgent: Hardware-guided GPU kernel optimization via multi-agent orchestration. PyTorch Blog, March 2026. URL https://pytorch.org/blog/kernelagent-hardware-guided-gpu-kernel-optimization-via-multi-agent-orchestration/.

Jiading Gai, Shuai Zhang, Kaj Bostrom, Jin Huang, Vihang Patil, Haoyang Fang, Bernie Wang, Huzefa Rangwala, and George Karypis. Optimizing CUDA like a human: Micro-profiling tools as expert surrogates for LLM-based GPU kernel optimization, 2026. URL https://arxiv.org/abs/2606.26453. KernelPro system.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. In *The Fourteenth International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=lKEdGCoDNC. arXiv:2509.19349.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. 2015. URL https://arxiv.org/abs/1504.04909.

Daniel Nichols, Konstantinos Parasyris, Caetano Melone, Tal Ben-Nun, Giorgis Georgakoudis, and Harshitha Menon. Record-remix-replay: Hierarchical GPU kernel optimization using evolutionary search, 2026. URL https://arxiv.org/abs/2604.11109.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

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 (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415, 2025. arXiv:2502.10517.

Qiuyi Qu, Yicheng Sui, Yufei Sun, Rui Chen, Xiaofei Zhang, Yuzhi Zhang, Haofeng Wang, Ge Lan, Ning Zhang, and Manfei Zhao. A two-stage gpu kernel tuner combining semantic refactoring and search-based optimization. *arXiv preprint arXiv:2601.12698*, 2026.

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 (PLDI)*, pp. 519–530, 2013.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, Wei Yang, and Tao Xie. KernelBand: Steering LLM-based kernel optimization via hardware-aware multi-armed bandits. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, Proceedings of Machine Learning Research, 2026. URL https://arxiv.org/abs/2511.18868. arXiv:2511.18868.

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent. https://github.com/algorithmicsuperintelligence/openevolve, 2025. GitHub repository.

Philippe Tillet, Hsiang-Tsung 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 (MAPL)*, pp. 10–19, 2019.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. GEAK: Introducing triton kernel AI agent & evaluation benchmarks, 2025. URL https://arxiv.org/abs/2507.23194.

Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. KernelFoundry: Hardware-aware evolutionary GPU kernel optimization. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, Proceedings of Machine Learning Research, 2026. URL https://arxiv.org/abs/2603.12440. arXiv:2603.12440.

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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 863–879, 2020.

A Extended Related Work and Problem Analysis

A.1 How Existing Frameworks Organize Search

This section expands the summary in §2.2. Producing one code change and organizing a series of attempts are related but distinct decisions. The first decides how the code changes; the second decides which past candidates are kept, where the next attempt starts, and how much evaluation it deserves. Recent frameworks address the second decision through feedback-driven planning, evolutionary populations, and statistical allocation.

Feedback-driven planning. GEAK runs a generate–test–reflect loop (Wang et al., 2025); KernelAgent adds hardware profiling, bottleneck diagnosis, and parallel modification (Cheng et al., 2026). K-Search explicitly maintains high-level plans awaiting verification, reprioritizes them from execution results, and lets one plan be realized by several implementations (Cao et al., 2026), so a single failed implementation does not reject the whole idea. Modifying only the current candidate can stall; explicit plans and parallel candidates extend this basic pattern. Two-stage tuners have also used resource measurements to organize structural rewriting and retuning (Qu et al., 2026), and compiler-grounded diagnosis has been used to locate bottlenecks in generated Triton kernels (Chen et al., 2026).

Evolutionary populations. AlphaEvolve draws on MAP-Elites and multi-subpopulation evolution, and OpenEvolve uses MAP-Elites with an island model (Novikov et al., 2025; Sharma, 2025). ShinkaEvolve emphasizes parent sampling, novelty filtering, and generator-model selection without a fixed MAP-Elites grid (Lange et al., 2026). KernelFoundry and R^3 combine MAP-Elites with parameter search: KernelFoundry keeps within-template tuning results and builds gradient-like prompts from parent–child differences in behavior and performance; R^3 also tunes compilation and launch parameters (Wiedemann et al., 2026; Nichols et al., 2026).

Statistical allocation. KernelBand clusters candidates by runtime behavior and combines profiling with multi-armed bandits to choose candidate groups and strategies (Ran et al., 2026); KernelPro selects and expands branches with Monte Carlo tree search (Gai et al., 2026); KernelBrain progressively raises the evaluation budget of promising candidates (Che & Peng, 2026). The bandit in ShinkaEvolve chooses the LLM that generates the modification, not the candidate group or strategy.

Search coverage versus improvement. MAP-Elites is representative of the archiving methods above, and its cell partition makes the trade-off between coverage and improvement explicit: each occupied cell keeps its current best candidate (Mouret & Clune, 2015; Wiedemann et al., 2026). Appendix A.3 analyzes and measures the sensitivity to the partition, and Appendix A.4 describes how this work avoids the choice.

A.2 Structure Search Conceptual Illustration

This section illustrates the single-operator structure search of §2.1. Figure 2(a) represents it as a two-dimensional conceptual space. Each point is a structure, and the irregular region is the feasible range under the task and hardware constraints. A local optimum is a structure that is hard to improve through the neighboring modifications currently allowed; the global optimum is the structure with the shortest latency in the whole feasible space. Positions only indicate illustrative relations among structures, not real coordinates or latencies, and the star does not mean the experiments found the global optimum.

Figure 2(b) shows the single-line stall discussed in Appendix A.1; Figure 2(c) shows the MAP-Elites archive analyzed in Appendix A.3; and Figure 2(f) previews the path this work takes: change the structure at a resource limit, then continue the search.

A.3 MAP-Elites Partition Sensitivity

The frameworks in §2.2 keep several candidates to reduce the risk of being confined to one line. Under a limited budget, however, there is a trade-off between which lines are kept and how fully they are improved. MAP-Elites makes this trade-off explicit through its cell partition: each occupied cell keeps one current best candidate, and those candidates seed later generations (Mouret & Clune, 2015; Wiedemann et al., 2026).

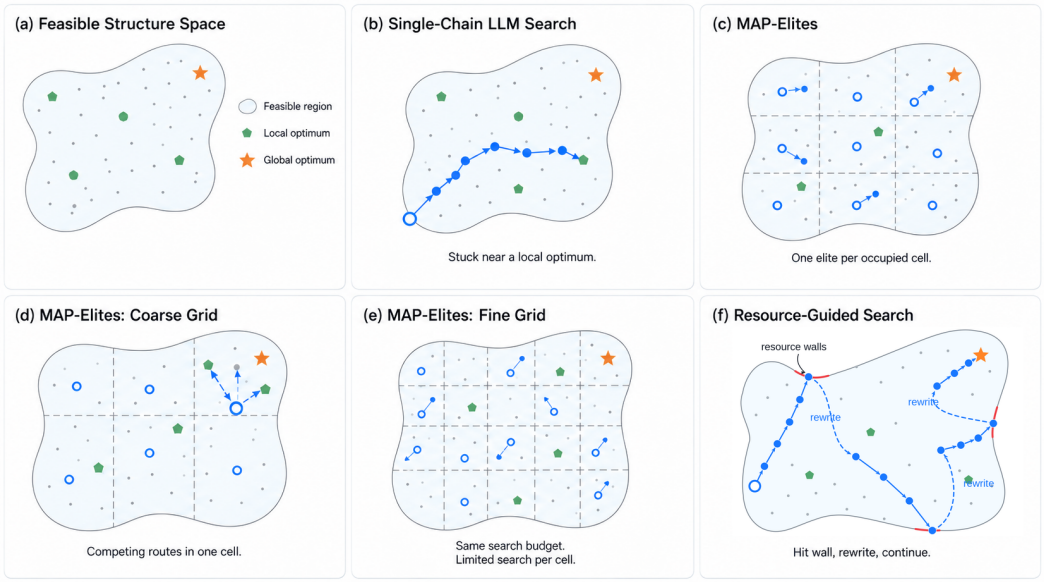

Figure 2: Conceptual illustration of structure search. (a) The feasible structure space; (b) a single-line search may stay near a local optimum; (c) MAP-Elites keeps one candidate per cell of its partition of the structure space; (d) a coarse partition makes several lines compete within one cell; (e) a fine partition keeps more candidates, but under a fixed budget some lines may remain under-explored; (f) an illustrative path that changes the structure at a resource limit and then continues searching. The grid is an archiving rule, not a hardware boundary; red boundary segments only illustrate resource limits, and rewriting does not change hardware limits. The subfigures are not measured trajectories, and (f) does not imply a global-optimality guarantee.

Table 3: Sensitivity of KernelFoundry's MAP-Elites archive to partition granularity (number of cells) on Matmul-Swish-GroupNorm. RTX 4090, seed 42, 128 candidate attempts per setting. Lower best latency is better; Passes is the number of those candidates that passed the correctness check.

Partition	Cells	Best latency (μs)	Passes
$2 \times 2 \times 2$	8	10.20	113
$3 \times 3 \times 3$	27	8.32	88
$4 \times 4 \times 4$	64	8.77	71

As Figure 2(d) shows, when cells are large and few, several local optima, or even the route to the global optimum, may fall into one cell. If the retained candidate sits near a local optimum, other lines that are temporarily slower may be dropped before they improve. A finer partition keeps more distinct lines, but exploring them all to a similar degree needs more attempts. Under a fixed budget some lines see only a few modifications and may end the search before reaching a local optimum, as in Figure 2(e).

On KernelFoundry's Matmul-Swish-GroupNorm task (KernelBench L2:37), we fix the model, hardware, and number of candidate attempts and vary only the partition granularity (number of cells). The three settings in Table 3 give different results: the best candidate of the coarse partition is 22.6% slower than that of the medium partition, and as the partition becomes finer, the number of candidates passing the correctness check within 128 attempts falls from 113 to 71.

Both risks point to the same judgment: a coarse partition must identify which temporarily slower lines are still worth keeping, and a fine partition must judge which retained lines deserve further investment. Current latency alone does not settle either judgment: a structure that currently performs well need not have much room left to improve. The question is therefore not only how to widen

Table 4: Inputs and outputs of the six agents. Outputs enter the next stage only after the framework's checks.

Agent	Input	Output
Candidate Generation	Reference implementation; task and device description	Several structurally distinct candidate sources
Parameterization	Candidate source; reference implementation	Source annotated with tunable parameters and the feasible values of each
Resource Analysis	Candidate source; selected values; the candidate's tuning record and resource usage	Judgment of which dimension is binding
Structural Rewrite	Source code with its best tested configuration; usage of each dimension; parameter direction of the limit; paired-comparison resource slopes; directions that previously gave no improvement	Rewritten candidate source and the expected change per dimension
Repair	Candidate source that failed the correctness checks; the check errors	Modified candidate source
Exploration	Reference implementation; starting sources of the existing lines	Candidate source different from all existing lines

coverage but how to judge which change can still improve the current structure, so that a limited number of attempts goes to directions with an explicit basis.

A.4 How This Work Avoids Choosing a Partition

This section supports §2.2. Unlike the MAP-Elites archives of Appendix A.3, this work does not keep lines in cells of a fixed partition and therefore chooses no partition granularity. A line is one starting candidate and its descendants; whether two candidates belong to the same line is decided by rewrite lineage, and whether they count as distinct structures is decided by a hash of the normalized source. Normalization removes comments and docstrings. The hash is a practical approximation for detecting duplicate implementations, not a test of algorithmic equivalence. Which lines to keep is then not a partition fixed in advance but a question of which line still deserves investment. The resource information of §3.2 and §3.3 informs that decision.

The cost is that this identity is only an operational approximation: two starting candidates that spell the same algorithm differently, and remain different after normalization, start two lines. Equivalent algorithms can therefore occupy multiple starting-candidate slots. Unlike cell-based competition, this does not discard a candidate merely because it shares a cell, but duplicates still consume slots and evaluation budget.

B Full Method Details

B.1 What Each Agent Receives and Produces

§3.1 divides responsibilities between the framework and the agents by whether inputs and outputs can be checked mechanically. Table 4 gives the inputs and outputs of the six agents. Each agent works in its own directory; inputs arrive as files, and outputs must be named files in that directory plus a JSON manifest that declares their content. The framework checks that the output is complete, then compiles and runs it, so an agent's output need not be trusted, only checked.

Besides the source, the Structural Rewrite Agent must declare the expected direction of change and its rationale for each resource dimension, so that rewrite intent can be compared dimension by dimension with the usage measured afterwards. The Exploration Agent is invoked only when all existing lines have stopped improving. In the main experiments, parameter search and rewriting exhausted the wall-clock budget before this condition was met, so the Exploration Agent was never called; all candidates in Tables 1 and 7 come from the Candidate Generation and Structural Rewrite Agents.

Table 5: Budgets and thresholds used in the experiments. The same values apply to all three devices and all three base models. TPE: tree-structured Parzen estimator, the sampler used for parameter search.

Item	Value	Meaning
Trials per parameter space	40	Parameter-search trials for one candidate
Random startup trials	10	Random draws before TPE modeling
Starting candidates	4	Structurally distinct candidates generated at the start
Rewrite rounds per line	5	Upper bound on rewrites of one line
Total / concurrently active lines	6 / 3	Upper bounds on lines overall and in parallel
Convergence criterion	2 rounds $< 2\%$	Stop the line when two consecutive rounds each improve by less than this
Repair attempts	3	Upper bound on rewrites after a failed correctness check
Value expansions per candidate	1	Upper bound on adding values and running one more search round
Expansion headroom threshold	0.8	Fraction of the measured limit below which a dimension counts as having headroom
Wall-clock budget	12 hours	Checked at round and candidate boundaries

B.2 Search Budgets and Stopping Criteria

This section lists the budgets and thresholds behind the procedure of §3.1 and the setup of §4.1 (Table 5). The parameter search of one candidate uses 40 trials drawn by a tree-structured Parzen estimator (TPE) sampler, with 10 random startup trials. The search space is decomposed according to which parameters co-occur across trials, and parameters are sampled jointly within each subspace. The sampler seed equals the run seed, fixed at 0 unless otherwise stated. Infeasible and failed trials each consume one of the 40 slots and enter the record with a failure category; only combinations rejected at sampling time by the parameter space's static constraints (constraints on parameter combinations that can be checked before a trial runs) are redrawn without consuming budget.

A rewrite line stops in two cases: five rewrite rounds are used up, or two consecutive rounds each improve by less than 2%. The 2% criterion uses the relative change in best latency between adjacent rounds and applies only after at least three rounds are recorded. The 12-hour wall-clock budget is checked between rounds, before each line, and before each candidate, never interrupting an agent call, so a run can slightly exceed 12 hours.

Adding values requires several conditions. Two of them decide whether the step is entered at all: at least one resource dimension is used below 0.8 of its measured limit, and the latency effect of the parameter direction near the boundary is at least 2%. The second condition excludes false boundaries on flat surfaces: when latency barely changes with the value, noise makes several directions look like boundaries. Values are added only for integer- or float-valued parameters, and values already at an implementation minimum are skipped (for example a warp count or pipeline depth of 1). If the expanded values equal the existing ones, or the additional search round does not improve by more than 2%, no further expansion is attempted for that candidate.

B.3 Correctness Gates and Evaluation Tiers

The single-operator correctness gate of §4.1 has three paths, evaluated in order; a candidate passes as soon as one path holds, and later paths are skipped. The first two paths compare against the outputs of the same reference implementation run under two matrix-multiplication precisions, TF32 and full FP32; we call these two outputs the dual witnesses. For each witness, two conditions must hold together: more than 99% of elements have relative error below 1%, and the cosine similarity of the flattened outputs is at least 0.99985. When the output contains non-finite values, the cosine test is replaced by a finiteness check. The third path runs only if both witnesses fail. It computes the root-mean-square error of the candidate and of the reference against an FP64 reference and requires the candidate's error to be within a fixed multiple of the reference's error ($2\times$ for FP32-class results, $3\times$ for FP16 and BF16), the multiples used by PyTorch's `torch._dynamo.utils.same()`. This path only adds ways to pass and never rejects a candidate that already passed; how often it took effect is recorded in the run log.

Table 6: Fields from the paired comparison of §3.3 received by each group; a bullet marks a field delivered to that group. All three groups receive the parent source, the reference implementation, the existing trial record, the parameter space and its static constraints, and the parent's register, spill, and shared-memory usage.

Field from the paired comparison	G0	G1	G2
Parameter values, latencies, and validity at both endpoints		•	•
Latency difference between endpoints and the performance slope		•	•
Endpoint resource usage (registers, spills, shared memory)			•
Resource-usage differences and resource slopes			•
Unobservable resource dimensions and observation status			•
Specific errors when an endpoint measurement failed			•

Evaluation has three tiers that tighten progressively so that most candidates are eliminated cheaply: each trial during parameter search uses 3 correctness checks and 20 timing repetitions; the witness check before parameter-space publication uses 3 correctness checks only; the final re-evaluation uses 5 correctness checks and 100 timing repetitions. The independent final re-evaluation measures a candidate in a separate process and reports the median. The feedback ablation uses the timing-block aggregation in §4.1, while model-level measurements follow Appendix C.2. Model-level candidates are additionally checked on local outputs, logits, and negative log-likelihood (NLL) (Appendix C.2). Lowering a candidate's internal compute or storage precision is permitted and reduces the data that must be moved, but its output precision and shape must match the reference. Within a tier, a candidate passes only if every correctness check passes.

B.4 Signals Delivered in the Feedback Ablation

The three groups of §4.3 share the same collected measurements; grouping only masks fields before they are handed to the agent and does not change the collection process, the prompt template, the parameter-sampling seed, or the device settings. The group identity does not appear in the agent's input. The two tasks of the ablation run on separate RTX 4090 hosts. Table 6 lists the fields each group actually received.

G0 receives no paired-comparison results and so has only the trial record and the parent's resource usage; G1 receives the performance-related part of the paired comparison; G2 receives everything. The difference between the last two groups is exactly the last four rows of the table.

B.5 Slope-to-Rewrite Case Study in Full

This section expands the case summarized in §4.3.2.

MBConv start 1. On this resource-constrained structure, two parameters gave different resource responses, one of which suggested a resource-related rewrite. The parent expand GEMM uses a full-K single load, with 255 registers, 108 spills, and 65536 B of shared memory—bound by both registers and residency. The tested `SPLIT_M` change gives the largest observed latency reduction in this comparison (58.09 ms to 8.54 ms; performance slope -0.787), but no change in measured resource usage. This signal describes the tested change; it does not establish that further tuning will be faster or indicate how to relieve register and residency pressure. The tested `BLOCK_N` change gives a smaller latency reduction (12.22 ms to 8.77 ms; performance slope -0.031), accompanied by negative resource slopes for registers, spills, and shared memory: increased measured usage accompanies lower latency. Because the parent has high register demand and limited block residency, the second signal suggests a rewrite: keep the benefit of large tiles while reducing what must be resident at once. Following §3.3, the two performance slopes are not comparable in magnitude; what is comparable is whether each comes with a change in resource usage.

The rewrites from G1 and G2 are consistent with these signals. G1 keeps the full-K load and only changes intermediate storage to FP16. G2 uses K-loop tiling with FP16 intermediate storage; its rationale explicitly cites the parent's residency of one block per SM, 255 registers, and 108 spills, and locates the change at the expand GEMM. After retuning, both eliminate spills. The G1 candidate uses 32768 B of shared memory and measures 5.076 ms in independent re-evaluation; the

G2 candidate uses 30208 B and measures 4.100 ms, with registers down from 255 to 140—19.2% lower latency.

G0 shows the limit of this argument: it also proposed a K-loop scheme, at 4.672 ms, faster than the full-K variant from G1. The K-loop idea therefore does not require resource slopes to be proposed, and adding only the performance slope does not guarantee a better rewrite direction.

MBConv start 2. This starting point is the complementary case: its performance slope is close to zero, yet the tiling parameters still show a resource slope. G1 changes only intermediate storage, while G2 replaces the expand GEMM and merges the separate statistics computation, at 4.237 and 3.530 ms respectively. A small parameter-level gain does not mean the structure has no room to change; resource feedback can still motivate further changes to the computation layout.

B.6 Paired-Slope Numerical Illustration

This section gives the numbers behind the paired-comparison panel of Figure 1 (§3.3), which shows that the two slopes form a criterion only together. Increasing the tile size from 32 to 64 improves latency by 1.40 while per-SM resource usage rises by 30 percentage points; increasing the number of pipeline stages from 2 to 3 improves latency by only 0.30 while usage barely changes. The first gain comes with a clear usage change in one dimension and so gives a basis for rewriting that dimension; the second comes with no usage change and points to no resource-related rewrite. This is the criterion applied to measured slopes in Appendix B.5.

B.7 Rewriting and Retuning Numerical Illustration

This section gives the numbers behind the rewrite-and-retune panel of Figure 1 (§3.4); they are an illustration, not a measured trajectory. In that panel, shared memory is the binding resource before the rewrite. After the rewrite, demand in that dimension falls substantially and the other two dimensions rise to absorb the cost; latency moves only from 6.80 to 6.90 ms, because the parameters still sit at values chosen for the original structure. After retuning, values that were previously cut off become feasible, demand rises in all three dimensions, and latency falls to 5.10 ms. The rewrite makes new configurations feasible, and retuning explores them: the rewrite alone moves latency only from 6.80 to 6.90 ms, and without the rewrite the values that give 5.10 ms stay infeasible.

The same panel also shows that the binding resource has changed, so the process returns to §3.2 to locate the new limit, measure the resource slopes again, and rewrite once more. In this illustration, relieving one constraint makes additional configurations feasible, and retuning brings demand closer to several limits. This is not a general guarantee about resource usage or performance.

C Additional Results

C.1 Base-Model Comparison

This section extends the main comparison of §4.2: Table 7 repeats ConvT2d-Mish and MBConv on the RTX 4090 with three base models, GLM-5.3-max, DeepSeek-v4-pro-max, and GPT-5.6-sol-xhigh, comparing each against the strongest framework under the same base model. On MBConv we lead under all three; on ConvT2d-Mish we lead under GLM and DeepSeek and trail K-Search under GPT-5.6.

C.2 The Three Task Contracts

§3.5 states that a task contract records the metric, the direction of optimization, the workload, and the inputs used for measurement. Table 8 gives the three contracts of the model-level experiments. The metric column gives each contract's full definition. Single-stream throughput is $127/(\text{time the last token is ready} - \text{time the first token is ready})$; the numerator is 127 rather than 128 because the time to produce the first token is not part of the decode rate. For concurrent throughput, the numerator 1024 is 128 output tokens for each of 8 requests, and the denominator is the wall-clock time of the shared batch of 8 requests, including all prefill and completion. All

Table 7: Base-model comparison on the RTX 4090. "Strongest" is the shortest valid latency among the four frameworks under the same base model; all three speedups use our latency as the denominator, with the Eager and Compile+TF32 baselines measured in the same re-evaluation as our candidate. The final candidates of all methods were re-evaluated in the same environment. The lowest latency in each row is in bold.

	Baseline framework latency (ms)				Ours			
Task	Kernel Foundry	K-Search	Kernel Band	Shinka Evolve	Latency (ms)	vs. Eager	vs. Compile+ TF32	vs. Strongest
GLM-5.3-max								
ConvT2d-Mish	3.912	4.220	4.205	36.249	**3.430**	13.88×	3.88×	1.141×
MBConv	5.222	4.501	4.990	7.612	**3.4545**	4.52×	3.30×	1.303×
DeepSeek-v4-pro-max								
ConvT2d-Mish	3.837	11.529	5.002	20.231	**3.260**	14.72×	4.08×	1.177×
MBConv	5.170	34.782	13.650	11.576	**2.770**	5.63×	4.12×	1.866×
GPT-5.6-sol-xhigh								
ConvT2d-Mish	3.863	**3.160**	5.042	4.397	3.821	12.56×	3.48×	0.827×
MBConv	5.043	7.540	5.775	5.303	**4.510**	3.46×	2.53×	1.118×

Table 8: The three task contracts of the model-level experiments. Search prompts are used for measurement during optimization; the final evaluation uses three disjoint prompt sets of the same size, each measured once (listed as sets × prompts).

Objective	Direction	Unit	Metric definition	Search prompts	Final prompts
TTFT	min	ms	From request submission to the first token being ready, including real prefill and first-token production; median over 4 requests	4	3×4
Single-stream	max	tokens/s	$127/(\text{last token} - \text{first token})$, with prefill and all 128 output tokens actually executed; median over 2 requests	2	3×2
Concurrent	max	tokens/s	$1024/\text{wall-clock time of the shared batch}$, with 8 requests issued simultaneously	8	3×8

three contracts are fixed before optimization begins; the run loads them only after verifying their hash.

The contract also fixes how generation and measurement run. Decoding is greedy with seed 0, and outputs must reach exactly the specified length; an end-of-sequence token before that point is ignored. Each candidate and parameter set first runs one warmup request of identical shape; the warmup is not scored but does count toward the wall-clock budget. One copy of the weights is resident per device, and each timed request uses freshly created KV cache, positions, mask, and random state. Tokenization happens outside the timed interval; token counts include the chat-template tokens actually rendered, exclude padding, and apply no truncation.

The declared model-level settings are $\mathrm{atol} = \mathrm{rtol} = 0.02$ for local outputs, a per-prompt logits relative L2 tolerance of 0.01, and a paired mean NLL change of at most 0.02 nat/token. NLL is computed on teacher-forced reference tokens. The local-output check uses the combined element-wise tolerance $|\hat{y}_i - y_i| \le \mathrm{atol} + \mathrm{rtol}|y_i|$, where y_i is the reference output. The logits tolerance also uses the measured discrepancy of a designated equivalent implementation, which can vary with the set of modules replaced. On the RMSNorm operator group of Qwen3 under the TTFT workload, binding the same numerically equivalent Triton RMSNorm to the first 1, 4, 16, 36, and 73 modules gives logits relative L2 errors of 0, 0, 0.004489, 0.004489, and 0.010349. The kernel is identical at every point; only the number of modules differs. With the stated tolerance multiple of 1, the effective logits tolerance is $\max(0.01, e_{\mathrm{ref}})$ when a reference discrepancy e_{ref} is available for the same contract, replacement sites, and module count; otherwise it is 0.01. A candidate passes this item if its error does not exceed that tolerance.

Table 9: Operator execution cost and objective gain on Qwen3-4B. (a) The three operator groups under the same TTFT objective; (b) the RMSNorm group under the three target workloads. Groups are merged according to §3.5. CUDA time is the cumulative kernel time attributed to all modules of the group under that workload; gains are the relative change in the contract metric J from the baseline of each objective, and bold marks the only group with a distinguishable TTFT gain.

(a) Operator groups, TTFT objective

Group	Quantity	Value
RMSNorm	CUDA time (μs)	14041
	TTFT gain	**+3.56%**
gate_proj, up_proj	CUDA time (μs)	88822
	TTFT gain	within noise
SiLU	CUDA time (μs)	5095
	TTFT gain	none faster

(b) RMSNorm group, three objectives

Objective	Quantity	Value
TTFT	CUDA time (μs)	14041
	Gain (latency)	+3.56%
Single-stream	CUDA time (μs)	2383
	Gain (tokens/s)	+12.51%
Concurrent	CUDA time (μs)	15036
	Gain (tokens/s)	+9.29%

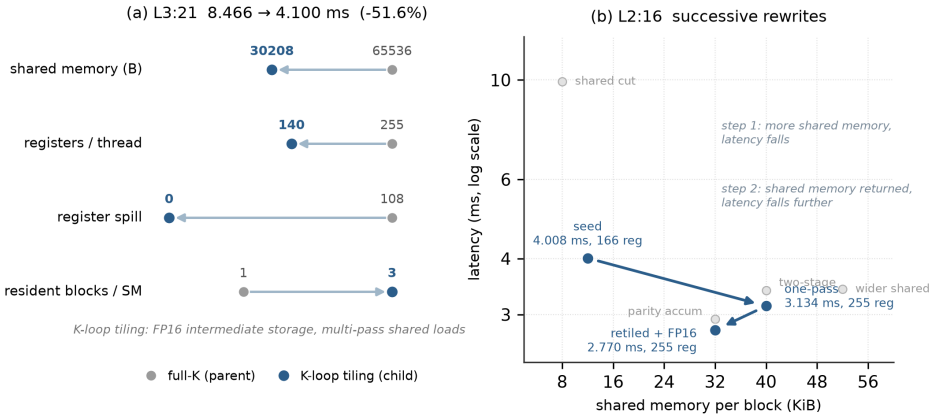

Figure 3: Structural rewriting relieving a resource limit. (a) MBConv: shared memory per block, registers per thread, register spills, and resident blocks per SM before (full-K load) and after (K-loop tiling) the rewrite, with the retuned latency. (b) ConvT2d-Mish: latency (log scale) against shared memory per block across successive rewrites; nodes are the best observation at each stage.

C.3 Operator Groups and Objective Gains on Qwen3-4B

Table 9 gives the CUDA time and objective gain of each operator group discussed in §4.4.

C.4 Alignment Cases in Detail

This section gives the numbers behind Figure 3 (§4.2.2).

In the MBConv case (Figure 3(a)), the parent structure loads the whole K dimension of the expand GEMM into shared memory at once: one block uses 65536 B of shared memory, registers hit the per-thread limit of 255 with 108 spills, and only one block fits per SM. The framework replaces the full-K load with K-loop tiling, trading more load rounds for a smaller per-block footprint, and stores intermediates in FP16. After retuning, shared memory falls to 30208 B, registers to 140, and spills to zero; three blocks fit per SM, and full-task latency drops by 51.6%.

In the ConvT2d-Mish case (Figure 3(b)), lower usage is not the goal: the starting candidate uses 12288 B of shared memory and 166 registers, and the first rewrite *increases* both while improving latency by 21.8%; later candidates adjust tiling and compute precision for a cumulative 30.9% gain over the starting candidate.

Table 10: Resource demands of the tasks. Data volume counts inputs, parameters, and outputs. Ref. ops is the number of ATen operations dispatched by the reference implementation in one forward pass.

Name	KernelBench ID	Task content	Data (MB)	Compute (GFLOP)	Ref. ops
ConvT2d-Mish	L2:16	2D transposed convolution, Mish, addition, Hardtanh, scaling	2684.5	154.6	5
Matmul-Swish	L2:59	Linear transformation, Swish, scaling	4328.7	274.9	5
ConvT3d-GroupNorm	L2:61	3D transposed convolution, ReLU, group normalization	457.1	231.9	3
MBConv	L3:21	EfficientNet MBConv block	322.0	112.1	11
CausalAttention	L3:43	Multi-head causal self-attention with output projection	416.3	412.3	42
Mamba2-SSD	L3:48	Mamba2 SSD output Y	1350.6	30.4	130

C.5 Isolated versus End-to-End Measurement of the TTFT Candidate

This section gives the full measurement summarized in §4.4. Its TTFT baseline was collected separately from Table 2(b). The TTFT candidate has a speedup of 0.980–0.988 in an isolated operator microbenchmark—slower than the reference on its own—yet installed in the model it lowers TTFT from 282.818 ms to 273.433 ms, a 3.32% improvement. The difference is the scope of measurement: the microbenchmark times only the operator's standalone call, while the model measurement includes its effect alongside the rest of the computation; execution records show 219 fewer kernel launches and 7.13 GiB less cumulative CUDA allocation after the rewrite. Acceptance must therefore be judged under the target workload, where an operator can be locally slower and globally faster at the same time, and the model objective is what the user cares about.

C.6 Task Resources and Hardware Calibration

C.6.1 Tasks with Different Resource Requirements

This section details the tasks and devices of §4.1. We select six KernelBench tasks (Table 10); the names used in the paper abbreviate the KernelBench problem titles. Data volume spans 322.0–4328.7 MB, computation spans 30.4–412.3 GFLOP, and the reference implementations contain 3–130 operations. Matmul-Swish and ConvT3d-GroupNorm have similar computational workloads, but the former has about $9.5\times$ the combined input, parameter, and output volume; CausalAttention performs substantial computation on a small data footprint, while Mamba2-SSD has low computation and a long operation chain. These differences create different optimization needs—data reuse, compute tiling, and intermediate storage—and test whether the framework adapts to different patterns of resource use.

Table 11: Hardware resource capacities and measured bandwidth and compute throughput. Bandwidth and compute peaks are measured on the experimental devices. Four precision tiers were measured—FP32, TF32, FP16, and BF16—of which the first two are listed. The performance reference uses the highest measured peak among the four tiers: 233.9, 158.6, and 214.9 TFLOP/s on the A800, RTX 4090, and RTX 5090 respectively.

Resource	A800	RTX 4090	RTX 5090
Memory (GB)	80	24	32
Usable shared-memory limit (B)	166912	101376	101376
SM count	108	128	170
Bandwidth (GB/s)	1683.1	907.1	1512.3
FP32 peak (TFLOP/s)	18.97	54.92	66.11
TF32 peak (TFLOP/s)	114.5	89.76	113.88

C.6.2 Hardware with Different Resource Capacities

The devices are an A800 80GB, an RTX 4090, and an RTX 5090, from the Ampere, Ada Lovelace, and Blackwell generations. They differ not only in the amount of each resource but also in tensor-core generation and available instructions, so a structure moved to another device cannot simply have its values rescaled by the ratio of each resource. For example, the A800 and RTX 4090 have 80 GB and 24 GB of memory and usable shared-memory limits of 166912 B and 101376 B; the larger memory of the A800 does not mean every resource grows by the same ratio. A structure must still be adjusted to the target device's compute throughput, bandwidth, and residency conditions. Table 11 summarizes device resources and measured capabilities.

C.7 Approaching Hardware Performance Limits

§4.2.2 shows how rewriting releases structural limits, but "can it still be faster" cannot be answered by a candidate's own search curve. A hardware-based reference uses the combined size of inputs, parameters, and outputs and the tabulated computation in GFLOP as estimates of data volume and computational workload. Dividing these quantities by the measured bandwidth and compute throughput and taking the larger time gives an estimated latency reference, not a proven minimum over all implementations. The estimate uses the highest measured compute throughput among the four precision tiers and the units stated in the tables. The data volumes in Table 10 are counted at the precision of the reference implementation, so when a candidate lowers its internal storage precision the bytes that must be moved fall accordingly and the reference is recomputed at the candidate's actual storage precision. We take three tasks on the A800, one bandwidth-bound, one compute-bound, and one whose reference has to be recomputed, with data volume and computation from Table 10 and bandwidth and peak from Table 11.

For Mamba2-SSD, the tabulated decimal units give $1350.6/1683.1 \approx 0.8024$ ms for the bandwidth reference; the optimized candidate measures 0.799 ms in independent re-evaluation, close to that estimate. For CausalAttention, the compute reference is 1.763 ms and the candidate measures 2.350 ms, giving a reference-to-measured latency ratio of 75.0%. Matmul-Swish shows why the recomputation matters: weights account for 99.2% of the bytes it must move, 4328.7 MB at the reference implementation's precision, which gives a reference of 2.572 ms. Its final candidate stores those weights quantized to 8 bits per output channel, so the traffic falls to 1107.4 MB and the memory-side time to 0.658 ms, at which point the compute side at 1.175 ms is the larger term and becomes the reference; the candidate measures 1.050 ms, or 89% of it. These comparisons describe proximity to the calibrated reference under the stated assumptions; they do not establish that further structural improvement is impossible.