ARTISAN: UNLOCKING LLM KERNEL OPTIMIZATION ON UNFAMILIAR HARDWARE

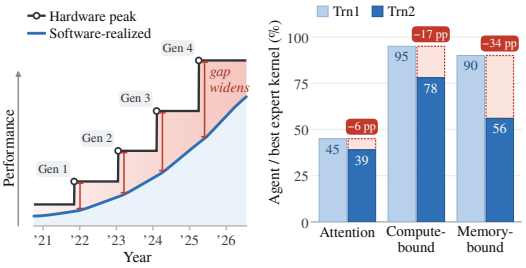
Anonymous authors
Paper under double-blind review

ABSTRACT

Rapidly evolving AI accelerators open a recurring gap between what the hardware can deliver and what its software actually realizes. LLM agents offer a promising way to automate kernel optimization, yet they struggle on unfamiliar hardware. A central limitation is missing hardware-specific knowledge: what to optimize on the new platform, and how to realize that optimization effectively. We introduce ARTISAN, which learns this knowledge automatically from a handful of expert kernels. Rather than imitating expert code, ARTISAN contrasts expert and agent kernels to learn reusable diagnostic tools for finding optimization opportunities and guides for realizing them on the target hardware. A task's artifacts are retained only if fresh agents that never see the expert can use the resulting support to achieve a verified improvement, filtering out knowledge that fails to transfer. On 40 unseen AWS Trainium 2 kernels, ARTISAN achieves a 1.58× geometric-mean speedup over the vendor-provided NKIPy baseline, with every kernel verified correct, versus 1.37× for the strongest baseline that learns from the same experts. Adding ARTISAN to four existing agentic optimization pipelines brings their reported speedups from 0.46–1.27× with 24–40/40 coverage to 1.55–1.60× with 40/40.

1 INTRODUCTION

AI accelerator hardware is advancing rapidly. NVIDIA now describes its data-center roadmap as a "one-year rhythm," AMD has moved Instinct to an annual cadence, and Google and AWS are releasing new TPU and Trainium generations in quick succession (Caulfield, 2024; Advanced Micro Devices, Inc., 2024; Vahdat, 2024; Barr, 2024). Software must continually catch up: each generation brings new optimization requirements, yet an efficient kernel can still take days or weeks to tune (Cheng et al., 2026). Google Research has reported a "fast-widening gap" between the peak performance of modern ML accelerators and the performance that software actually realizes (Li & Jouppi, 2022). This growing hardware–software gap calls for kernel optimization that adapts quickly to each new platform (Figure 1a).



(a) Hardware–software gap (b) Agent–expert gap

Figure 1: **New hardware outpaces its software, and agents too.** (a) Schematic: each generation reopens the gap to peak. (b) Agents recover less of the best expert kernel on the newer Trainium 2.

Agentic kernel optimization offers a promising way to close this gap. Recent systems strengthen different parts of the optimization loop, from execution-grounded generation and search to feedback-driven refinement and structured agent workflows (Ouyang et al., 2025; Hong et al., 2025; Cheng et al., 2026); on mature GPU stacks, they can approach expert-level performance (Cheng et al., 2026). New hardware, however, poses a distinct challenge: effective optimization requires hardware-specific knowledge that has not yet accumulated in frontier models or in the software ecosystem. Frontier-model weights are not typically updated at the pace of hardware releases, and available resources may be limited to official documentation and a few expert-written kernels. On matched tasks across two successive Trainium generations, agent-optimized kernels trail expert ker-

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.

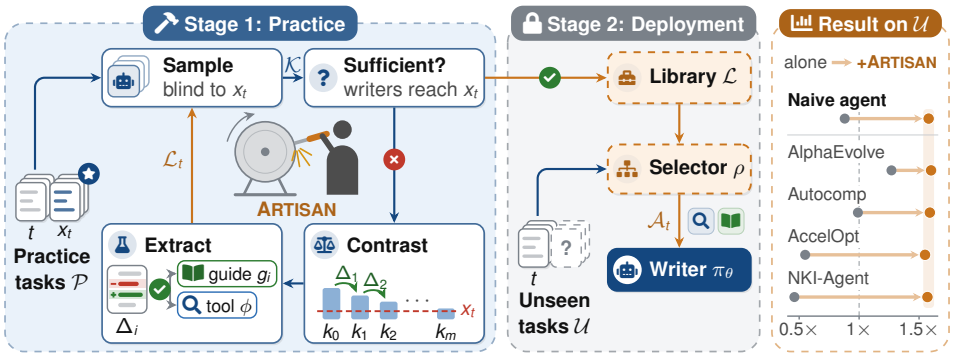


Figure 2: **ARTISAN sharpens the agent's tools and guides.** *Stage 1 (practice):* on a practice task t with expert kernel x_t, fresh writers that never see x_t sample kernels $\mathcal{K}$ using support $\mathcal{A}_t$ selected from the task library $\mathcal{L}_t$. While they fall short of x_t, the practice agent decomposes the gap between their best kernel and x_t into single changes Δ_i measured on the device, and extracts from each a guide g_i and, when needed, a tool ϕ. *Stage 2 (deployment):* ARTISAN freezes the learned tools and guides. For each unseen task, the selector ρ gives the unchanged writer only the relevant ones, $\mathcal{A}_t$. *Right:* geometric-mean speedup over NKIPy on 40 unseen Trainium 2 kernels for each agent alone (grey) and with frozen ARTISAN support (orange). The naive agent is the writer loop without artifacts (Tables 1 and 2).

nels further on the newer one (Figure 1b), suggesting the hardware–software gap also persists for agents.

Two opportunities make this adaptation tractable. First, modern LLMs already command many general optimization strategies, such as data reuse, fusion, tiling, and overlap (Ouyang et al., 2025; Cheng et al., 2026), and these strategies remain useful across hardware platforms (Hong et al., 2025). Adaptation therefore need not teach optimization from scratch; it must instead make familiar strategies work on unfamiliar hardware. Second, hardware-specific lessons transfer across kernels on the same hardware: ML workloads share primitive operations and optimization opportunities, so knowledge learned from a few expert kernels can be reused on many others. In our experiments, support learned from a single expert kernel already reaches $1.38\times$ the compiler on forty unseen kernels, versus $1.58\times$ with all ten practice kernels (Section 4.4).

Each opportunity, however, comes with a matching challenge. First, general strategies transfer, but knowing a strategy does not tell an agent how to apply it on new hardware. Even a familiar idea such as overlap is realized through different execution and memory mechanisms on different architectures, so an agent can pick the right strategy yet reason with the wrong hardware model (Appendix E.4); this gap persists despite substantial target-hardware documentation (Appendix E). Second, expert kernels contain the knowledge that transfers but do not separate it from what does not. Each mixes broadly useful principles with architecture- and kernel-specific decisions, so the agent must work out which choices matter, why they work, and when they apply to a different kernel. Turning a few expert implementations into knowledge that generalizes to unseen kernels is the central challenge that ARTISAN addresses.

The knowledge to be extracted has two parts, mirroring the two steps of optimizing a kernel: deciding what to change, and making that change effective in code (Figure 3). First, the agent must learn to turn execution feedback into an optimization direction. Profilers expose detailed memory traffic and compute timelines, but these signals do not by themselves reveal what actually limits performance on new hardware; unless the agent can combine them and attribute them to the underlying bottleneck, detailed feedback can still lead to the wrong optimization. A matched control confirms that interpretation is the missing step: the same repair rule becomes effective once the relevant behavior is identified for the agent (Appendix E.3). Second, even when the direction is clear, the agent must learn to realize it on the target hardware. A strategy such as reuse, fusion, or overlap names the right idea but leaves many coupled hardware-specific decisions open: reusing data, for example, requires deciding where the data resides, how the computation is organized around it, and how both choices fit the hardware's execution model. A second matched control isolates this gap: even when told the intended optimization explicitly, the agent repeatedly fails to implement it, whereas a con-

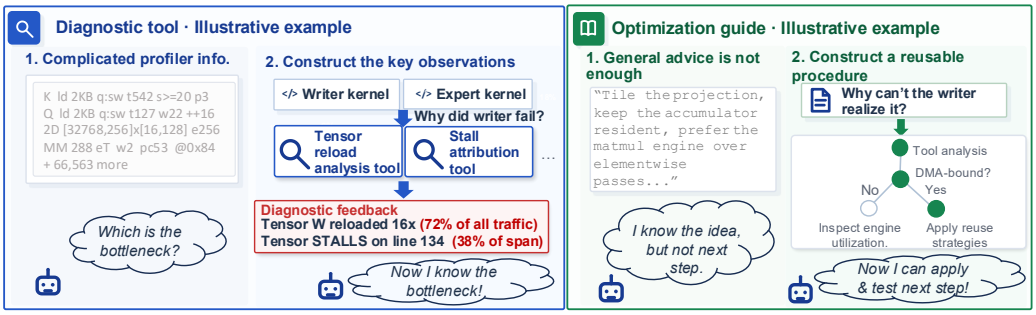


Figure 3: **General strategies transfer, but their effective use is hardware-specific.** Left: raw profiling information exposes execution signals but does not identify the dominant bottleneck; a learned diagnostic tool aggregates and interprets these signals into an optimization direction. Right: high-level advice names a strategy but not how to realize it on the target hardware; a learned guide provides a conditional, testable implementation procedure.

crete target-hardware procedure makes the same optimization executable (Appendix E.3). Together, the two gaps show that general optimization knowledge is not enough.

We propose ARTISAN[1] (Figure 2), which adapts an existing agent to unfamiliar hardware by learning what it lacks from a few expert kernels, without updating model weights. Its key idea is to treat expert kernels not as solutions to imitate but as evidence of the hardware-specific knowledge the agent is missing; we call this process *practice*. On each practice kernel, fresh writers first attempt the task. While they fall short of the expert, a practice agent decomposes the gap between their best kernel and the expert's into single changes measured on the device, and turns each change into support of two forms that match the two missing parts. *Diagnostic tools* turn code and profiler information into the behavior that limits performance, telling the agent what to optimize; *optimization guides* record the tested target-hardware decisions that make a known strategy effective, telling the agent how to realize it. To keep only lessons that transfer, ARTISAN retains a task's artifacts only if fresh writers that never see the expert can use the resulting support to achieve a verified improvement. At deployment the library is frozen, and a selector learned during practice gives each unseen task only the artifacts it needs, so any agent can use the same support unchanged. Our main contributions are as follows.

- **A diagnosis of why agents fall behind on new hardware.** Controlled studies on Trainium 2 identify two key gaps in hardware-specific knowledge (Figure 3): how to interpret execution feedback as an optimization direction, and how to realize that direction in code. Documentation and high-level advice alone do not reliably close either gap (Appendices E and E.3).
- **Practice, which learns this knowledge from a few expert kernels.** ARTISAN turns expert–agent differences measured on the device into validated diagnostic tools and optimization guides (Figure 2 and Algorithm 1) that any existing agent can use unchanged.
- **Transfer to unseen kernels and existing agents.** On 40 unseen Trainium 2 kernels, ARTISAN achieves a $1.58\times$ geometric-mean speedup over the NKIPy reference with 40/40 verified coverage (Table 1(a)), versus $1.37\times$ for the strongest method learning from the same experts (Figure 6); it brings four existing pipelines from 0.46–$1.27\times$ to 1.55–$1.60\times$ with full coverage (Table 1(b)). Ablations show that the gain comes from practice, not the source material, that tools and guides are complementary, and that selective delivery matters (Table 2).

2 PROBLEM SETUP

We study kernel optimization by a coding agent on hardware it has seldom seen, where the agent is built on a frontier closed-source model that cannot be retrained.

[1] An artisan is a skilled craftsman. The name reflects ARTISAN's role: it sharpens existing coding agents for unfamiliar hardware by equipping them with learned tools and guides.

Problem formulation. A task t on a target platform H is a *kernel specification* (operator semantics, shapes, and dtypes) paired with a reference implementation r_t produced by NKIPy; tasks are drawn from a workload distribution $\mathcal{D}_H$. On a new platform, the hardware-specific resources are typically limited to a *minimal knowledge set* $\mathcal{S}_H$: (i) the official documentation D; (ii) an execution harness $E = (M, C, T, P)$, where $M(k)$ compiles a *candidate* kernel k and returns compiler diagnostics on failure, $C_t(k) \in \{0, 1\}$ checks k against r_t for correctness (we say k is *verified* if $C_t(k) = 1$), $T(k)$ measures latency, and $P(k)$ returns a profiler trace; and (iii) a small *practice set* $\mathcal{P}$ of tasks, each with an expert-written kernel x_t. Thus $\mathcal{S}_H = (D, E, \{(t, x_t) : t \in \mathcal{P}\})$ contains expert kernels only for practice tasks. The *unseen set* $\mathcal{U}$ consists of held-out tasks from $\mathcal{D}_H$; it is disjoint from $\mathcal{P}$ and revealed only after the learned support is frozen, and for $t \in \mathcal{U}$ the agent receives the specification and r_t but no expert kernel. For each unseen task, the goal is a verified kernel that is as fast as possible, measured by its speedup $T(r_t)/T(k)$ over the reference.

The writer. We call any agent that optimizes kernels through the harness a *writer*. A writer may be a single model or a multi-role pipeline with search and memory; the only requirement is that it runs the compile–verify–measure loop on E. Writers differ in how they explore candidates, but on a new platform they may share a central deficit: an incomplete mental model of H—its execution, memory, constraints, and performance. We therefore abstract every writer as a policy π_θ, where θ denotes everything fixed about it (model weights, prompts, and control flow). Given a task t and a set of *artifacts* $\mathcal{A}$, the writer runs the loop under a fixed budget B: it proposes a candidate k, receives the feedback $E(k) = (M(k), C_t(k), T(k), P(k))$, and revises (Figure 4). We write $\mathcal{K} = \mathrm{Loop}_B(\pi_\theta; t, \mathcal{A})$ for the candidates it produces together with their feedback. The writer always receives D and the task instructions, while $\mathcal{A}$ is optional; $\mathcal{A} = \emptyset$ is the base case, a general agent facing an unfamiliar platform with only its documentation and harness. Our main experiments use a *minimal writer*: one frontier model running the loop without a specialized workflow, search policy, or cross-task memory, so that gains can be attributed to the artifacts rather than to orchestration. Section 4.2 shows that the result is not specific to this setup by adding the same frozen artifacts to four existing pipelines.

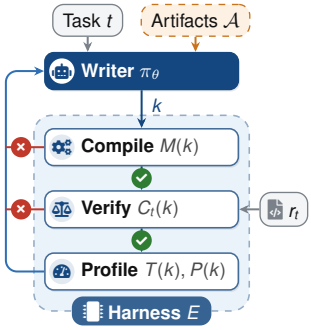


Figure 4: The writer loop. Dashed orange marks what ARTISAN learns.

Objective. Because θ is frozen, the artifacts are the only input that can change a writer's behavior, so improving the writer on H means improving what it is given. We judge a run by its fastest verified candidate, $k_t^\star(\mathcal{A}) = \arg\min_{k \in \mathcal{K},\, C_t(k)=1} T(k)$, and set $k_t^\star(\mathcal{A}) = r_t$ if no candidate verifies, so a failed run gains nothing over the reference. We therefore seek artifacts, constructed from $\mathcal{S}_H$ alone and frozen before any task in $\mathcal{U}$ is seen, that minimize the latency of this kernel on tasks from $\mathcal{D}_H$, that is, maximize its expected log speedup

$$\max_{\mathcal{A}} \; \mathbb{E}_{t \sim \mathcal{D}_H} \left[\log \frac{T(r_t)}{T\big(k_t^\star(\mathcal{A}_t)\big)} \right], \tag{1}$$

where $\mathcal{A}_t \subseteq \mathcal{A}$ is the part delivered for task t and the expectation also covers the writer's sampling. The logarithm makes this a geometric-mean speedup, so no single kernel dominates.

3 THE ARTISAN ALGORITHM

Objective (1) leaves open what the artifacts should contain. What the writer lacks on the target hardware H are the two parts identified in Section 1: interpreting execution feedback as an optimization direction, and realizing that direction in code (Figure 3). ARTISAN therefore learns two kinds of artifacts, one for each part (Figure 5). A *diagnostic tool* ϕ is an executable measurement that the harness runs on a candidate; its report $\phi(k)$, computed from k's source and profile, extends what the writer *observes*. An *optimization guide* g states when a general strategy applies on H, which hardware-specific decisions realize it, and how to check that it took effect; it extends what the writer *knows how to do*. Neither artifact is tied to a specific task: each describes an observable kernel behavior and how to act on it, allowing it to transfer to other kernels on H. ARTISAN learns a library $\mathcal{L}$ of such artifacts and a selector ρ giving each task t a compact set $\mathcal{A}_t = \rho(t; \mathcal{L}) \subseteq \mathcal{L}$. The selector

Algorithm 1 ARTISAN practice

Require: practice tasks $\mathcal{P}$ with experts; writer π_θ; harness E
Ensure: library $\mathcal{L}$ and selector ρ

$\mathcal{L} \leftarrow \emptyset$; $\rho \leftarrow \rho_0$; $\mathcal{P}_{\text{seen}} \leftarrow \emptyset$
for each practice task t with expert x_t **do**
  $\mathcal{L}_t \leftarrow \emptyset$; $\mathcal{A}_t \leftarrow \rho(t; \mathcal{L}_t)$; $\mathcal{K} \leftarrow \textsc{Sample}(t, \mathcal{A}_t)$
  while $\mathcal{K}$ fails the independent-use test (3) and budget remains **do**
    $(k_0, \ldots, k_m) \leftarrow \textsc{Contrast}(\mathcal{K}, x_t)$ ▷ invoke contrast agent
    $\mathcal{L}_t \leftarrow \textsc{Extract}(k_0, \ldots, k_m; \mathcal{K}, \mathcal{L}_t)$ ▷ invoke extraction agent
    $\rho \leftarrow \textsc{ReviseSelector}(\rho; t, \mathcal{K}, \mathcal{L}_t)$; $\mathcal{A}_t \leftarrow \rho(t; \mathcal{L}_t)$
    $\mathcal{K} \leftarrow \textsc{Sample}(t, \mathcal{A}_t)$
  end while
  if $\mathcal{K}$ passes test (3) **then**
    $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}_t$; $\mathcal{P}_{\text{seen}} \leftarrow \mathcal{P}_{\text{seen}} \cup \{t\}$
    $\rho \leftarrow \textsc{ReviseSelector}(\rho; \mathcal{P}_{\text{seen}}, \mathcal{L})$
  end if
end for
return $(\mathcal{L}, \rho)$

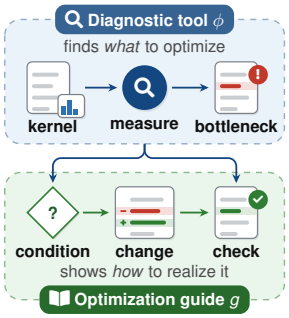


Figure 5: The two learned artifacts: a diagnostic tool ϕ and an optimization guide g.

is a learned delivery policy: from task facts and artifact scopes, it decides which tools and guides to render and in what order, turning objective (1) into a search over $(\mathcal{L}, \rho)$.

Expert kernels show what performance is attainable on H but cannot be simply delivered: each mixes transferable hardware decisions with choices specific to its operator and shape, so handing the expert kernel x_t to a writer solves its task t but teaches little about other tasks. ARTISAN instead learns from the experts through *practice* (Figure 2 and Algorithm 1). SAMPLE, CONTRAST, and EXTRACT in Algorithm 1 are separately prompted *agent roles*—a writer, an analyst, and an artifact builder—rather than program functions. Their context is deliberately separated. Each practice task t starts from an empty task library $\mathcal{L}_t$; before every SAMPLE, the current selector chooses the delivered subset $\mathcal{A}_t = \rho(t; \mathcal{L}_t)$. SAMPLE invokes fresh writer agents using $\mathcal{A}_t$, without the expert kernel x_t or the other agents' conversations. When these writers fall short, CONTRAST may inspect the expert and uses code changes and device measurements to isolate the performance gap into individually tested steps. EXTRACT turns each measured step into task-agnostic tools and guides. If a later sample still fails, its trajectory reveals whether the writer lacked a diagnosis, an implementation detail, or the right delivered artifact; EXTRACT revises the support and REVISESELECTOR updates its scope, ranking, or rendering before the next sample. Only CONTRAST sees the expert, so SAMPLE remains deployment-matched and each role has an auditable output (Appendix C.1). Task libraries that pass the independent-use test are pooled into $\mathcal{L}$, while the selector is regression-tested on previously accepted tasks as the pool grows. The practiced library and selector are then frozen for deployment.

Learning artifacts: contrast, then extract. Because the writer already has strong general capability, ARTISAN targets the hardware-specific part of its remaining gap to the expert, one decision at a time. CONTRAST builds a *ladder* from the writers' fastest verified kernel k_0 toward the expert kernel x_t. At step i, the practice agent compares the current kernel k_{i-1} with the expert in code, per-tensor traffic, engine utilization, and profiler timeline, proposes a change Δ_i that the expert makes and k_{i-1} does not, applies it to k_{i-1} alone, and keeps $k_i = k_{i-1} \oplus \Delta_i$ (k_{i-1} with Δ_i applied) only if it stays verified and gets faster:

$$C_t(k_i) = 1 \qquad \text{and} \qquad T(k_i) < T(k_{i-1}), \tag{2}$$

until the last kernel k_m comes within a factor $1 + \epsilon$ of the expert's latency (tolerance ϵ) or the budget is spent. Each step thus isolates one expert decision and measures its effect on a kernel the writer itself produced, so constructs of the expert kernel that do not help the writer never enter the ladder; changes that pay off only in combination are tested as a single step. EXTRACT then distills each step into a guide that states which observable behavior calls for the change, how to realize it on the target hardware, and what should improve afterward, and it builds a new tool only when the existing source and profile readings do not expose that behavior. In Figure 3, the writer repeatedly reloads an operand that the expert keeps on chip: the guide records how to reorganize the computation around the resident operand, and a per-tensor reload tool reveals which operands are refetched from main memory. An artifact is retained only if it is sound on its own step, meaning that the issue it diagnoses

is present before the step (k_{i-1}), absent in the expert, and resolved after it (k_i), and only if an audit finds in it no operator name, shape constant, or expert code fragment; it thus describes a hardware behavior and its remedy rather than the practice task (Appendix C.2; examples in Tables 6 and 7).

When does practice on a task succeed? Practice on t ends once writers, rather than the practice agent, can reach the expert. Because writers are stochastic, SAMPLE runs three fresh writers on t, each with the documentation, harness, and selected artifacts $\mathcal{A}_t$, but never with the expert kernel or practice conversation. Writer j returns its fastest verified kernel $k_j = k_t^\star(\mathcal{A}_t)$, and $\mathcal{K}$ records all candidates and feedback. Practice on t passes the *independent-use test* when at least two of the three writers come within a factor $1+\epsilon$ of the expert's latency:

$$\left|\{\, j \in \{1,2,3\} :\ T(k_j) \le (1+\epsilon)\, T(x_t)\,\}\right| \ \ge 2. \tag{3}$$

Three properties make this the right test. The writers never see the expert kernel, so any improvement must come through the practiced artifacts and delivery rule; they run exactly as they will at deployment, so the gap they leave is the one that matters on unseen tasks; and requiring two of three tolerates one unlucky run of a stochastic writer. The first sample, with $\mathcal{L}_t = \emptyset$, measures the base writer, so every artifact addresses a gap the writer actually leaves on t. A failed sample is not wasted: $\mathcal{K}$ shows whether writers misapplied an artifact, could not realize it, never located its precondition, or were given distracting support. The practice agents revise the library and selector accordingly, repeating until the test passes or the budget expires; a task library that never passes is not pooled (Appendix C.2).

Practicing the selector. Delivery is practiced jointly with artifact construction, and becomes more essential as task libraries are pooled. Even within one task, an irrelevant or premature artifact may distract the writer; across tasks, an artifact learned elsewhere may also steer it toward an inapplicable change. Because applicability depends on concrete kernel properties rather than task wording alone, we use context-conditioned selection instead of semantic similarity retrieval (Fu et al., 2024). REVISESELECTOR is an agent step with one objective: deliver a compact artifact set that can pass the independent-use test. For each artifact, the agent records when it applies—from broad operation classes to concrete shape, dtype, and diagnosed-behavior checks—then learns simple extraction, ranking, and presentation rules from fresh-writer failures. The initial policy ρ_0 delivers every artifact whose current applicability description matches the task; practice progressively makes this rule more selective. After accepting a task library, it regression-tests ρ on earlier tasks using $\rho(t;\mathcal{L})$: a missing useful artifact expands or fixes its selection conditions, whereas a distracting artifact makes them more specific. Thus selection is learned throughout practice, rather than built once at the end or fixed as a similarity heuristic (Appendix D); Section 4.3 shows that this selectivity is essential.

4 EXPERIMENTS

4.1 SETUP

Tasks. The benchmark is a fixed selection of 50 public kernels for the AWS *Neuron Kernel Interface* (NKI), a Python-based low-level interface for writing custom Trainium kernels that are compiled by the Neuron Compiler (Amazon Web Services, 2025). The kernels come from official AWS resources and the Trainium suites of Autocomp, AccelOpt, and NKI-Agent (AWS Neuron, 2026a; Hong et al., 2025; Zhang et al., 2026; Tang et al., 2026); together they form an externally defined distribution spanning operators, fused computations, model blocks, and distinct shape regimes. Whereas KernelBench and TritonBench provide standardized GPU evaluations (Ouyang et al., 2025; Li et al., 2025), we target adaptation to NKI's programming model: it exposes accelerator-specific partition and free dimensions, multiple compute engines, and explicit on-chip data movement, so shapes directly change legal tilings, memory residency, DMA granularity, and useful schedules (Amazon Web Services, 2025). Ten practice kernels with strong public experts cover central NKI motifs (attention, normalization and projection, feed-forward networks, reductions, layout, and reuse), and forty others are held out (Appendix B details their variation in implementation and shape).

Practice. For each practice task, the expert is the strongest verified public implementation we found: an official AWS library or tutorial kernel when available, and otherwise the fastest verified release from a prior NKI system (Appendix B). Each task starts with an empty artifact library; the shared selector is re-tested on all prior accepted tasks after every update. We use $\epsilon = 0.03$ in both the ladder's stopping rule and independent-use test (3); a task may be revised at most 24 times

Table 1: **Practice-derived support transfers and composes with existing optimization pipelines.** All methods use Opus 4.7 and the same six-attempt, three-turn budget; speedup is the verified-set geometric mean over NKIPy, and tokens average all deployment calls (Appendix B).

(a) Learning from the same practice experts

Method	Speedup	Verified	Tokens (k) in / out
ARTISAN	**1.58×**	**40/40**	**320 / 52**
GEPA	1.37×	39/40	174 / 49
DSPy/MIPROv2	1.35×	39/40	283 / 46
ExpeL	1.32×	39/40	234 / 44
Voyager	1.31×	39/40	324 / 44
AutoGuide	1.28×	39/40	312 / 61

(b) Native and online search; adding ARTISAN support

	Native		Native tokens (k)	+ ARTISAN	
Method	Speedup	Verified	in / out	Speedup	Verified
Autocomp	0.99×	38/40	1,084 / 58	1.59×	40/40
AccelOpt	0.55×	39/40	58 / 25	1.55×	40/40
NKI-Agent	0.46×	40/40	270 / 41	1.57×	40/40
AlphaEvolve-style	1.27×	24/40	77 / 49	**1.60×**	**40/40**
LATS	1.19×	32/40	—	—	—

(c) GPU optimizers ported to NKI

Method	Speedup	Verified
KernelAgent	0.96×	38/40
Astra	0.89×	26/40

(d) NKIPy baseline

Method	Speedup	Verified
NKIPy	1.00×	40/40

within 600 seconds before its library is abandoned (Appendix C.2). The resulting library contains 51 diagnostic tools and 37 optimization guides; of these 88 artifacts, 32% encode Trainium-specific computation strategies, 25% concern the efficient use of the compute engines, and 18% capture the memory hierarchy and its data-movement optimizations (Appendix C); Appendix D gives the frozen selector.

Evaluation protocol. A result counts only after full-shape verification and serial re-measurement. On the forty unseen kernels, we report *verified coverage*, the fraction of tasks with a verified kernel, and the *geometric-mean speedup* $T(r_t)/T(k_t^\star)$ over the NKIPy reference (AWS Neuron, 2026b). Unlike objective (1), which scores a failed task as $1\times$, speedup is computed over verified kernels only, and paired latency comparisons use common verified subsets (Appendix B.2). We report coverage and token use from the median-speedup run of three independent repetitions. All agentic methods use Opus 4.7, the same six-attempt, three-turn budget, and default reasoning settings, so comparisons isolate the adaptation and optimization procedures from model strength and budget.

Comparisons. Five practice baselines learn from the same experts and freeze what they learn: GEPA evolves prompts by reflection (Agrawal et al., 2026), DSPy/MIPROv2 tunes instructions and demonstrations (Khattab et al., 2024; Opsahl-Ong et al., 2024), ExpeL distills insights (Zhao et al., 2024), Voyager stores code skills (Wang et al., 2024a), and AutoGuide keeps context-conditioned guidelines (Fu et al., 2024). Four NKI pipelines search each kernel online and are rerun with the same harness and budget, with paired variants adding frozen ARTISAN support: Autocomp rewrites from an optimization menu (Hong et al., 2025), AccelOpt adds a memory of past optimizations (Zhang et al., 2026), NKI-Agent repairs in a compile–verify–fix loop (Tang et al., 2026), and AlphaEvolve-style search evolves candidate populations (Novikov et al., 2025). We also adapt LATS tree search (Zhou et al., 2024) and port the GPU workflows KernelAgent and Astra to NKI (Cheng et al., 2026; Wei et al., 2025); Appendix B details all baselines, ports, and metrics.

4.2 DOES THE SUPPORT TRANSFER TO UNSEEN KERNELS?

Practice-derived support transfers without hand-written optimization guidance. On the forty held-out kernels, ARTISAN reaches $1.58\times$ over the NKIPy reference with full coverage (Table 1). The five practice baselines, which learn from the same experts, reach $1.28\times$–$1.37\times$ with 39/40 each; the five native or online search methods span $0.46\times$–$1.27\times$ with 24–40/40; and the two ported GPU optimizers reach $0.89\times$–$0.96\times$ with 26–38/40.

GPU workflows need more than a translated vocabulary. Our NKI ports of GPU-designed KernelAgent and Astra swap CUDA/Triton interfaces and GPU performance guidance for NKI APIs and official AWS optimization materials, preserving the original role division, control loop, and budget. Their $0.96\times$ and $0.89\times$ show that a workflow-preserving port is not enough: case analysis shows missed Trainium opportunities (cross-engine overlap, PSUM accumulation, partition-aware layouts, ISA-level fusion) that require a target-specific performance model rather than new API names.

Online search does not replace offline practice. Adapted to search kernel revisions online, LATS reaches $1.19\times$ with 32/40 verified. In our runs, its scalar latency reward favors reliable local refinements but cannot diagnose a structural error in the agent's hardware model: large loop-order

or data-layout rewrites are explored, yet their early implementations often fail on unfamiliar NKI, and the low rewards suppress those branches, concentrating search further on local edits. Offline expert contrast instead analyzes such failures before deployment, turning the missing observation or procedure into support that search can execute.

ARTISAN complements search rather than replacing it. Given the same frozen tools and guides, all four native pipelines rise to $1.55\times$–$1.60\times$ with 40/40 (Table 1(b)): their search becomes effective once it can interpret unfamiliar-hardware evidence and turn it into an implementable revision. AlphaEvolve-style search rises from 24/40 to 40/40, and its paired $1.60\times$ is the highest in Table 1, just above standalone ARTISAN, which thus acts as a shared enabling layer for diverse search.

Transfer spans distinct kernels and shape regimes. Practice and evaluation share hardware motifs, but no kernel repeats: held-out tasks change operator composition, tensor sizes, and performance regimes (Appendix B). As noted above, shape changes alone reshape an NKI implementation; public attention kernels contain dozens of size-conditioned implementation paths. Nevertheless, support from the best single kernel verifies 39/40 unseen tasks (Section 4.4), and the full library improves four different search pipelines. On the practice tasks themselves, ARTISAN also matches or improves all ten public experts, by $1.15\times$ in geometric mean and up to $1.36\times$ on one kernel (Table 3). With the weaker transfer of methods given the same experts, these results show that practice learns a feedback-to-action capability spanning implementations and shapes.

Why does ARTISAN outperform other forms of practice? Controlled studies point to the mechanism behind this gap. Even with substantial target-stack documentation, the writer without artifacts produces a correct kernel from scratch in only 5/18 runs, and from a correct start only 7/18 runs improve by more than 5% (Appendix E); diagnostics make rules applicable, and worked constructions make strategies executable (Appendix E.3). The obstacle therefore appears not to be primarily retrieving a general strategy such as reuse or fusion, but adapting it to the Trainium-specific evidence of the current kernel and to the coupled constraints of a valid NKI implementation.

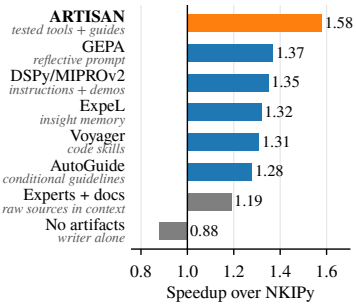


Figure 6: **Same experts, different retained support.** Speedup over NKIPy on the forty unseen kernels (Tables 1(a) and 2).

The practice baselines retain experience in complementary ways: prompt- and memory-based methods distill instructions, examples, or insights; code-skill libraries preserve reusable routines; and conditional-guideline methods link broad task context to advice. ARTISAN targets the missing adaptation step: expert contrast identifies whether the writer lacks an observation or a procedure, and practice builds either a diagnostic that exposes the hardware behavior in the current candidate or a conditional worked change that realizes the required NKI rewrite. Because fresh trials retain a task library only when writers can use its delivered support apart from the originating trajectory, ARTISAN transfers a tested feedback-to-action capability rather than a record of past successes. From the same experts, it reaches $1.58\times$, versus $1.37\times$ for the strongest alternative, GEPA (Figure 6).

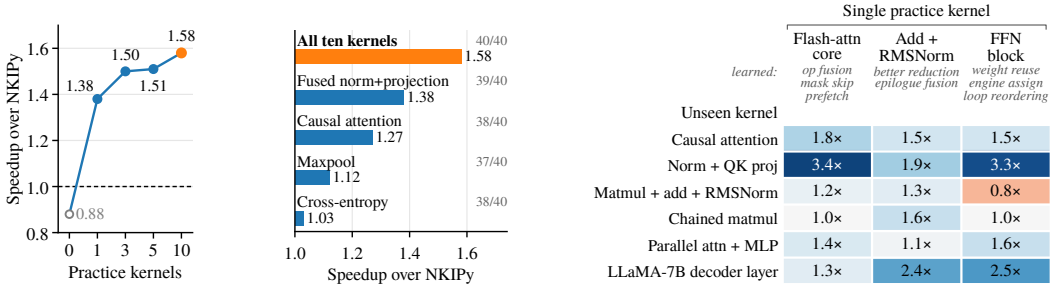
Practice cost and token use. At deployment, ARTISAN uses 320k input and 52k output tokens per task (Table 1), within the 174k–324k input range of other practice methods: diagnostic reports and worked guides give the writer a more accurate model of Trainium. Over ten practice cases, ARTISAN uses 1.32M input and 0.25M output tokens, versus 0.89M and 0.75M for GEPA: testing evidence-rich support uses more input but a third of the output, cutting cost from $22 to $14.

4.3 WHAT MAKES THE LEARNED SUPPORT EFFECTIVE?

Table 2 separates three sources of this gain: constructing actionable support, combining observation with procedure, and delivering applicable support. If the gain came mainly from stronger source material, the ten expert kernels plus the official documentation would approach the main configuration; instead, they reach $1.19\times$ with 38/40, versus $0.88\times$ and 22/40 without any artifacts. Raw source material thus recovers coverage but still falls well short of the main configuration's speed; the gain comes from practice converting it into usable support.

Table 2: **Ablating support construction and delivery.** Speedup over NKIPy and coverage; *full library* delivers all 88 artifacts without selection.

Construction			Delivery and budget		
Support condition	Speedup	Verified	Delivery / attempt budget	Speedup	Verified
ARTISAN (main)	**1.58×**	**40/40**	Full-library delivery	1.37×	36/40
Guides only (no tools)	1.51×	40/40	Random-k guides	1.32×	37/40
Tools only (no guides)	1.45×	40/40	Three attempts	1.28×	35/40
Experts + official opt. docs	1.19×	38/40	One attempt	1.22×	29/40
No artifacts	0.88×	22/40			
Writer model, unchanged artifacts					
Sonnet 4.6	1.30×	39/40	Fable 5.1	1.70×	40/40



(a) Practice-set size (b) Practice on one kernel (c) Where its lessons transfer

Figure 7: **A little practice can unlock transfer.** Verified-set geometric-mean speedup over NKIPy on forty unseen kernels versus (a) practice-set size and (b) the single kernel practiced (top: all ten); (c) per-target transfer relative to no artifacts (red: slowdown).

Tools alone reach 1.45× and guides alone 1.51×, both with full coverage, versus 1.58× together: tools identify where to change, while guides make the change implementable. Delivering all 88 artifacts reaches 1.37× with 36/40 and k random guides 1.32× with 37/40, showing that relevance-aware selection is part of the support, not merely a token-saving device. One and three attempts reach 1.22× and 1.28×, so the support benefits iterative rather than single-shot optimization.

The frozen library transfers across writer models (1.30×–1.70×, 39–40/40; Table 2); stronger writers explore broader revisions, but even weaker ones retain substantial transfer.

4.4 How much practice is needed, and which kernels help?

Much of the gain appears after the first practice kernel. Across one, three, five, or ten practice kernels, one fused normalization–projection kernel already yields 1.38× with 39/40, versus 1.58× and 40/40 for all ten (Figure 7a). This early gain suggests that practice *unlocks* existing capability rather than teaching unseen kernels individually.

A kernel with broadly reusable operations teaches more than a narrow one. Single-kernel practice yields 1.03×–1.38× (Figure 7b), with fused normalization–projection transferring best among the examples. Its validated code examples alone reach 1.40×, versus 1.27× for its prose, showing transfer of executable procedures as well as profiler interpretation. By target (Figure 7c), FFN weight reuse and engine assignment speed up a structurally different LLaMA-7B decoder layer 2.5×, while mismatched support can regress (0.8× on matmul + add + RMSNorm).

5 Conclusion

On new hardware, strong agents may know the right strategies but lack the hardware-specific knowledge to apply them. Because ML workloads reuse optimization motifs, our results expose an opportunity: practice on a few expert kernels can build an agent's working model of the platform and unlock performance on unseen kernels. ARTISAN realizes this opportunity, reaching 1.58× with 40/40 coverage and bringing four existing pipelines to 1.55×–1.60×; agents can retain their general capabilities while learning to use them effectively on unfamiliar hardware.

Limitations. We establish cross-kernel transfer only on Trainium 2; cross-architecture transfer remains future work. Practice quality may depend on model capability and informative public references; future work can study smaller models and combine experts across accelerator ecosystems.

Reproducibility statement

Code, tasks, measurements, run records, and analysis scripts will be released. Appendix B specifies the benchmark, the practice/evaluation split, the measurement protocol, and the baseline instantiations; Appendices C and C.2 describe the artifacts and their validation; and Appendix D gives the frozen selector.

AI use statement

The LLM agents studied in this paper are components of the method and the baselines (Sections 3 and 4). Separately, we used generative AI tools (Claude Code and Codex) to help implement and run experiment and analysis scripts, including the motivation study (Appendix E); to extract, clean, and reformat run records into per-kernel tables; to give feedback on the design and presentation of experiments; and to create and edit figures, draft and edit text, search for literature, and format references. We have not used generative AI tools to generate synthetic data sets or to formulate or prove mathematical claims. We have reviewed all AI-assisted work: the authors designed the study, checked AI-assisted code and every reported number against the raw measurement records, and checked each reference against its source. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

References

Advanced Micro Devices, Inc. AMD Accelerates Pace of Data Center AI Innovation and Leadership with Expanded AMD Instinct GPU Roadmap. AMD Newsroom, June 2024. URL https://www.amd.com/en/newsroom/press-releases/2024-6-2-amd-accelerates-pace-of-data-center-ai-innovation-.html. Accessed: 2026-09-21.

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2507.19457.

Amazon Web Services. Neuron Kernel Interface (NKI). https://awsdocs-neuron.readthedocs-hosted.com/en/latest/nki/index.html, 2025. AWS Neuron SDK Documentation.

AWS Neuron. NKI Library. https://github.com/aws-neuron/nki-library, 2026a. Pre-built reference kernels for the AWS Neuron SDK and NKI; GitHub repository, accessed September 6, 2026.

AWS Neuron. NKIPy: Rapid prototyping on Trainium. https://github.com/aws-neuron/nkipy, 2026b. Open-source experimental project, accessed September 2, 2026.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn RL for generating CUDA kernels. *arXiv preprint arXiv:2507.11948*, 2025.

Jeff Barr. Amazon EC2 Trn2 Instances and Trn2 UltraServers for AI/ML Training and Inference Are Now Available. AWS News Blog, December 2024. URL https://aws.amazon.com/blogs/aws/amazon-ec2-trn2-instances-and-trn2-ultraservers-for-aiml-training-and-inference-is-now-available/. Accessed: 2026-09-21.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2305.17126.

Brian Caulfield. "Accelerate Everything," NVIDIA CEO Says Ahead of COMPUTEX. NVIDIA Blog, June 2024. URL https://blogs.nvidia.com/blog/computex-2024-jensen-huang/. Accessed: 2026-09-21.

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/. Accessed: 2026-09-21.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. AutoGuide: Automated generation and selection of context-aware guidelines for large language model agents. In *Advances in Neural Information Processing Systems*, volume 37, 2024. arXiv:2403.08978.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2305.11738.

Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. Autocomp: A powerful and portable code optimizer for tensor accelerators. *arXiv preprint arXiv:2505.18574*, 2025.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations (ICLR)*, 2024. arXiv:2310.03714.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. TritonBench: Benchmarking large language model capabilities for generating Triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, 2025. doi: 10.18653/v1/2025.findings-acl.1183.

Sheng Li and Norman P. Jouppi. Unlocking the Full Potential of Datacenter ML Accelerators with Platform-Aware Neural Architecture Search. Google Research Blog, February 2022. URL https://research.google/blog/unlocking-the-full-potential-of-datacenter-ml-accelerators-with-platform-aware-neural-architecture-search/. Accessed: 2026-09-21.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. arXiv:2303.17651.

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. *arXiv preprint arXiv:2506.13131*, 2025.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366, 2024. doi: 10.18653/v1/2024.emnlp-main.525.

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.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 8634–8652, 2023. arXiv:2303.11366.

Junjie Tang, Jun Huan, Hao Zhou, Yuhao Zhang, and Lin Wang. NKI-Agent: Domain-specific fine-tuning and agentic tool use for Neuron kernel generation. *arXiv preprint arXiv:2607.04395*, 2026. DL4C Workshop at ICML 2026.

Amin Vahdat. Announcing Trillium, the Sixth Generation of Google Cloud TPU. Google Cloud Blog, May 2024. URL https://cloud.google.com/blog/products/compute/introducing-trillium-6th-gen-tpus. Accessed: 2026-09-21.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a. arXiv:2305.16291.

Zhiruo Wang, Graham Neubig, and Daniel Fried. TroVE: Inducing verifiable and efficient toolboxes for solving programmatic tasks. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *PMLR*, pp. 51177–51191, 2024b.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025. arXiv:2409.07429.

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.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. arXiv:2210.03629.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. AccelOpt: A self-improving LLM agentic system for AI accelerator kernel optimization. In *Proceedings of Machine Learning and Systems (MLSys)*, 2026. arXiv:2511.15915.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2024. arXiv:2308.10144.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 62138–62160, 2024.

A Related work

Per-kernel agentic optimization. One line of work improves the optimization loop applied to each target kernel. Autocomp separates planning from code generation, supplies a hand-written menu of accelerator optimizations during planning, and uses correctness and latency from the device to drive iterative search (Hong et al., 2025). KernelAgent more explicitly reproduces a GPU engineer's workflow: agents profile, diagnose, prescribe, coordinate parallel candidates, and share within-run memory (Cheng et al., 2026); Astra assigns generation, testing, profiling, and planning to cooperating agents, starting from an existing CUDA kernel (Wei et al., 2025). General search frameworks supply related control mechanisms: LATS searches a tree of language-agent actions, while FunSearch and AlphaEvolve evolve program populations under executable rewards (Zhou et al., 2024; Romera-Paredes et al., 2024; Novikov et al., 2025). These methods improve use of the model's current knowledge, but their search policy, profiler vocabulary, and optimization knowledge are fixed when a new kernel arrives; they do not use a small practice set to construct hardware support for later tasks. Our online-pipeline comparisons isolate this distinction: ARTISAN changes what these pipelines can observe and realize on the unfamiliar platform while preserving their native search.

Learning for kernel generation and optimization. Other kernel systems do include a learning stage, but place the learned knowledge in a different carrier. Kevin trains model parameters with multi-turn reinforcement learning so that CUDA generation benefits from compiler and performance rewards across revision turns (Baronio et al., 2025). NKI-Agent addresses the scarcity of Neuron code by supervised fine-tuning on a constructed NKI corpus, then applies a compile–verify–fix loop at inference time (Tang et al., 2026). AccelOpt avoids weight updates and instead curates an optimization memory from previously observed slow–fast kernel pairs; later kernels retrieve these natural-language experiences to guide exploration (Zhang et al., 2026). Their learned objects are thus a domain-specialized policy or lessons from optimization trajectories. ARTISAN instead targets a frozen writer that knows general strategies but lacks the hardware-specific interface needed to use them: expert kernels expose capabilities it systematically misses, and practice externalizes only those capabilities as reusable support.

Learning reusable experience, workflows, and skills. Self-Refine repeatedly uses a model's own critique to revise one output, and Reflexion carries verbal reflections across trials in episodic memory (Madaan et al., 2023; Shinn et al., 2023). ExpeL aggregates successful and failed training trajectories into textual insights and retrieves both insights and experiences for new tasks (Zhao et al., 2024). Voyager stores executable behaviors in a growing code-skill library; Agent Workflow Memory induces recurring action sequences; and AutoGuide learns concise advice paired with the contexts in which it should be applied (Wang et al., 2024a; 2025; Fu et al., 2024). ARTISAN falls in this broad category of non-parametric adaptation, but these methods primarily compress what the agent did, inferred, or succeeded with in its own trajectories. On unfamiliar hardware, the decisive fact may be absent from those trajectories: the writer may not know which profile signals reveal a bottleneck, or may name the right optimization without knowing a legal implementation. ARTISAN instead uses measured expert–agent interventions to discover the missing observation or procedure, rather than treating a successful trajectory as the learning unit.

Tool construction and prompt adaptation. ReAct and CRITIC reason with existing tools; Toolformer learns when to call a supplied API; and SWE-agent redesigns the interface exposed to a coding agent (Yao et al., 2023; Gou et al., 2024; Schick et al., 2023; Yang et al., 2024). LATM and TroVE synthesize reusable functions that package known computations (Cai et al., 2024; Wang et al., 2024b), while DSPy/MIPROv2, TextGrad, and GEPA optimize instructions or prompts from task feedback (Khattab et al., 2024; Opsahl-Ong et al., 2024; Yuksekgonul et al., 2024; Agrawal et al., 2026). On a new accelerator, the problem is not merely to call, create, or select a known tool: the agent does not yet know *what* to observe or *how* to realize the corresponding change. ARTISAN turns a few experts into a hardware-facing interface for a frozen agent: contrast discovers observables and executable rewrites, extraction removes task-specific content, and fresh writers retain them only when they transfer without the expert. The learned unit is a verified evidence-to-action path: a diagnostic sharpens what the agent can see, and a guide unlocks what it can do—not a larger prompt or library of solutions.

B BENCHMARK TASKS AND PRACTICE RESULTS

B.1 BENCHMARK COMPOSITION AND FIXED SPLIT

Every benchmark task is selected from a public source; none is a custom-designed task created for this study. The 50 Trainium 2 kernels come from the official AWS NKI Library (AWS Neuron, 2026a), AWS NKI documentation and tutorials (Amazon Web Services, 2025), and the open workload suites of Autocomp, AccelOpt, and NKI-Agent (Hong et al., 2025; Zhang et al., 2026; Tang et al., 2026). This study defines only the fixed practice/evaluation split and the reported measurements. For each of the ten practice tasks, the expert reference is the strongest public kernel implementation we found: an official AWS implementation or tutorial when available, otherwise the fastest verified implementation released by the prior NKI systems above. Thus, practice uses high-quality public expert kernels rather than private hand-authored solutions; the remaining 40 tasks form the evaluation set. The practice kernels are `add_rmsnorm_matmul`, `attention_core`, `cross_entropy`, `csa_2048`, `ffn_block`, `gelu_ffn`, `maxpool`, `mha`, `mha_qkv_proj`, and `proj_norm_proj`. Table 3 gives their shapes and comparisons to these selected expert references. None of these kernel identities occurs in Tables 4–5. The split evaluates transfer across implementations and shape regimes, not repetition of a source kernel. This distinction is especially material for NKI's software-managed execution model: tensor sizes determine feasible tiles, partition/free-dimension mappings, SBUF and PSUM residency, DMA transaction structure, and which engines can overlap, so changing a dimension can change the effective optimization strategy rather than merely its launch parameters. For attention alone, counting the public NKI variants together with size-conditioned branches inside those kernels yields dozens of distinct implementation paths (Amazon Web Services, 2025; AWS Neuron, 2026a). The split reflects this diversity: for example, practice includes 2,048-token attention cores, while evaluation includes a 16,384-token online-attention update, grouped-query attention, QK normalization, and complete attention blocks with different projections and sequence regimes. Attention, normalization, and feed-forward motifs occur on both sides because they expose transferable hardware principles, while kernel identity, composition, shapes, and resulting schedules remain distinct.

Tables 4–5 identify every evaluation kernel and its principal dimensions. The inventory includes individual operators, fused computations, and complete model blocks; shapes follow each task's operand order. Table abbreviations include general matrix multiplication (GEMM), multi-head attention (MHA), scaled dot-product attention (SDPA), and query/key/value projections (QKV).

Table 3: **Practice-stage performance against prior expert implementations.** All ten practice kernels match or improve the prior expert; none enters the forty-kernel evaluation result.

Practice kernel (shape)	Prior expert (μs)	ARTISAN (μs)	Speedup
3×3 s1 maxpool, 4096^2	583	428	1.36×
Cross-entropy, N=512, V=32000	1,631	1,613	1.01×
1-head causal SDPA, S=2048, D=128	149	134	1.11×
Add+RMSNorm+matmul, $4096\times1024\times2048$	339	326	1.04×
Causal MHA, H=8, S=2048, D=128	709	605	1.17×
GELU-FFN, D=4096, I=11264	6,214	5,804	1.07×
GEMM–RMSNorm–GEMM, S=2048, D=4096	5,217	4,840	1.08×
MHA + fused QKV proj., S=2048, D=4096	1,580	1,428	1.11×
32-head flash-attn core, S=2048, D=128	2,824	2,254	1.25×
SwiGLU-FFN, $1024\times512\times2048$	9,834	7,581	1.30×
Geometric mean (10/10 beat-or-tie)			**1.15**×

Table 4: **The 40 unseen evaluation kernels (1–20).** Identifiers map readable workload names to benchmark tasks; dimensions and formulas summarize the task specifications. Continued in Table 5.

#	Identifier	Workload	Principal shape or specification
1	`adamw`	AdamW optimizer step	$\theta, g, m, v : (10944, 2048)$; weight decay 10^{-5}, learning rate 10^{-2}, $\beta = (0.9, 0.999)$.
2	`attention_block`	LLaMA-2-7B attention, prefill	$x : (1, 512, 4096)$; $W_{q,k,v,o} : (4096, 4096)$; RMSNorm, QKV projection, RoPE, causal attention, output projection, residual.
3	`bmm`	Batched matmul	$(16, 4096, 64) \mathbin{@} (16, 64, 4096)$.
4	`complex_epilogue`	GEMM with GELU epilogue	$\mathrm{GELU}(A^\top B + C) \odot D$; $A : (4096, 4096)$, $B : (4096, 8192)$.
5	`conv1d`	Depthwise 1D convolution	Input $(8, 512, 2048)$; weights $(512, 1, 3)$; 512 groups; padding 1.
6	`conv2d`	Dense 2D convolution	Input $(16, 128, 128, 128)$; weights $(512, 128, 3, 3)$; padding 1.
7	`csa_16384`	Flash-attention online update	One 128-query tile against 16,384 KV tokens; running output, normalizer, and maximum; fp32.
8	`cumsum`	Inclusive prefix sum	$x : (2048, 2048)$; last-axis scan.
9	`falcon_mlp`	Falcon-7B GELU MLP	$x : (1, 2048, 4608)$; $W_{\mathrm{up}} : (18432, 4608)$, $W_{\mathrm{down}} : (4608, 18432)$.
10	`gated_proj`	SiLU-gated projection	$x : (1, 2048, 4096)$; gate and down-projection weights $(4096, 4096)$.
11	`gelu`	Elementwise GELU	$x : (4096, 11008)$; erf formulation.
12	`gelu_matmul`	GEMM and GELU	$\mathrm{GELU}(A^\top B)$; $A : (4096, 8192)$, $B : (4096, 1024)$.
13	`gqa`	Grouped-query attention	$Q : (1, 32, 2048, 128)$; $K, V : (1, 8, 2048, 128)$; query-to-KV head ratio $4 : 1$.
14	`groupnorm`	GroupNorm	$x : (16, 64, 256, 256)$; 8 groups.
15	`layernorm`	GPT-2 Medium LayerNorm	$x : (1, 1024, 1024)$; normalized width 1024.
16	`llama7b_layer_decode`	LLaMA-2-7B decoder layer	$x : (1, 1, 4096)$; attention and SwiGLU MLP; intermediate width 11008; sequence length 1.
17	`logsumexp`	Stable log-sum-exp	$(128, 4096) \to (128, 1)$.
18	`lora_acceleopt`	Rank-128 LoRA linear	$x : (4096, 5120)$; $w : (5120, 12288)$; adapters $(5120, 128)$ and $(128, 12288)$.
19	`lora_step`	Rank-16 LoRA adapter	$x : (1, 2048, 4096)$; $W : (4096, 4096)$; adapters $(4096, 16)$ and $(16, 4096)$.
20	`mamba`	Mamba selective-scan SSM	$x : (1, 256, 2560)$; inner width 5120; state size 16.

Table 5: **The 40 unseen evaluation kernels (21–40).** Continuation of Table 4, with the same identifier and workload conventions.

#	Identifier	Workload	Principal shape or specification
21	`mamba_ssm`	Short selective scan	$x : (1, 16, 256)$; $A : (256, 16)$; state size 16.
22	`matmul`	Dense GEMM	$(4096, 5120)$ @ $(5120, 12288)$.
23	`matmul_add`	GEMM and bias add	$A^\top B + C$; $A : (4096, 4096)$, $B : (4096, 8192)$.
24	`matmul_add_relu`	GEMM, bias, and ReLU	$\mathrm{ReLU}(xW + b)$; $x : (1, 2048, 4096)$, $W : (4096, 11264)$.
25	`matmul_add_rmsnorm`	GEMM, residual, RMSNorm	$\mathrm{RMSNorm}(xw + z)$; $x : (4096, 2048)$, $w : (2048, 2048)$.
26	`multi_matmul`	Chained matmul	$(AB)C$; operands $(4096, 1024)$, $(1024, 2048)$, and $(2048, 4096)$.
27	`norm_qkproj`	RMSNorm and Q/K projection	$x : (1, 2048, 4096)$; $Q, K = \mathrm{RMSNorm}(x)W_{q,k}^\top$; output $Q + K$.
28	`qknorm`	Chameleon-7B QK-norm scores	$Q, K : (1, 32, 2048, 128)$; per-head-dimension RMSNorm before attention scores.
29	`rmsnorm`	RMSNorm	$a : (512, 4096)$; scale $g : (4096)$.
30	`rmsnorm_matmul`	RMSNorm and matmul	$\mathrm{RMSNorm}(x)w$; $x : (4096, 1024)$, $w : (1024, 2048)$.
31	`rmsnorm_proj`	LLaMA-2-7B normalized projection	$(\mathrm{RMSNorm}(x) \odot w)W_{\mathrm{proj}}^\top$; $x : (1, 2048, 4096)$.
32	`rope`	Rotary position embedding	$Q, K : (1, 32, 2048, 128)$; sine and cosine tables $(2048, 128)$.
33	`scaled_matmul`	Scaled-LHS GEMM	$(0.5A)^\top B$; $A : (4096, 4096)$, $B : (4096, 8192)$.
34	`sdattention`	Single-head causal SDPA	$q, k, v : (4096, 64)$; scale $1/\sqrt{64}$; causal mask.
35	`silu`	Elementwise SiLU	$x : (4096, 7168)$.
36	`softmax`	Softmax	$x : (4096, 8192)$; last-axis reduction.
37	`swiglu`	SwiGLU FFN	$(\mathrm{SiLU}(xw_{\mathrm{gate}}) \odot xw_{\mathrm{up}})w_{\mathrm{down}}$; $x : (4096, 1024)$; intermediate width 3072.
38	`transformer_encoder_block`	Post-norm encoder block	$x : (8, 512, 512)$; 8 heads; FFN width 2048; bidirectional attention.
39	`transpose`	3D tensor transpose	$x : (512, 512, 512)$; swap axes 1 and 2.
40	`transpose_matmul`	Transposed-LHS GEMM	$\mathrm{lhs}^\top \mathrm{rhs}$; lhs $(2048, 4096)$, rhs $(2048, 10944)$.

B.2 Measurement and reporting

Evaluation uses forty kernels disjoint from the practice set, with artifact contents and retrieval policy frozen. Candidates undergo full-shape numerical verification and serial latency remeasurement; failed candidates remain in coverage accounting. Each agentic condition is run in three independent repetitions to account for stochastic generation. For each repetition, we compute geometric-mean speedup over that repetition's verified set and then select the repetition with the median of the three speedups. The displayed coverage is the verified-task count from that same median-speedup repetition; speedup and coverage in each row therefore describe one consistent run. For the verified set V_M of method M, we report coverage $|V_M|/40$ and geometric-mean speedup

$$G_M(V_M) = \exp\left[\frac{1}{|V_M|}\sum_{c\in V_M}\log\frac{t_{\text{ref}}(c)}{t_M(c)}\right],$$

where t_{ref} is the NKIPy latency and t_M the verified candidate latency. Paired latency comparisons use the common verified subset, whereas headline ratios compare the geometric-mean scores reported in Table 1, each over its own verified set. Speedup uses P50 latency for all measurements. All displayed speedups are rounded to two decimal places. For orientation, ratios of the displayed scores—not paired estimates on a common verified subset—range from $1.58/1.37 = 1.15\times$ for ARTISAN versus GEPA to $1.60/1.27 = 1.26\times$ through $1.57/0.46 = 3.41\times$ for the supported versus native pipelines. For each repetition and evaluation task, deployment input and output tokens are each summed over all LLM calls in the complete optimization trajectory, including unsuccessful attempts. For the selected median-speedup repetition, we average these per-task totals over all forty tasks and exclude practice expenditure. The resulting ARTISAN deployment count is 320k input and 52k output tokens per task.

Practice cost and token accounting. We separately account for the complete practice runs on the same ten practice cases. ARTISAN uses 1.32M input and 0.25M output tokens, with a recorded API cost of $14; GEPA uses 0.89M input and 0.75M output tokens, costing $22. The ARTISAN input includes task evidence and evolving diagnostic-tool and worked-guide drafts repeatedly supplied for execution-based refinement and independent-use validation. This shifts practice toward input-side evaluation of explicit support: ARTISAN uses more input than GEPA but only one-third as much output and a 36% lower recorded cost.

B.3 Baseline instantiations

All five adaptation baselines learn from the same ten practice tasks and corresponding expert implementations, then freeze their learned product before the forty evaluation tasks. Across all methods and writer-model swaps, reasoning/effort levels and other model-side deliberation settings use the provider defaults; we do not tune them by method or task. The resulting DSPy/MIPROv2 product contains one 1,622-character instruction and three selected kernel demonstrations; GEPA produces one evolved prompt containing 11 hardware rules; ExpeL stores 20 natural-language insights; Voyager stores named, parameterized code skills organized by operator family; and AutoGuide stores textual advice indexed by conditions such as operation, reduction, or dtype. Inspection of the resulting frozen files supports the characterization in Section 4.2: these products preserve instructions, examples, insights, code skills, or conditional advice, rather than constructing and independently validating a candidate-specific path from profiler evidence to a worked revision. This statement concerns the products produced by our matched NKI instantiations, not every possible extension of the underlying frameworks.

KernelAgent and Astra are GPU-oriented multi-agent optimization workflows (Cheng et al., 2026; Wei et al., 2025). For their NKI ports, we replace CUDA/Triton APIs, examples, profiler vocabulary, and GPU performance guidance (e.g., warps, coalescing, shared memory, and occupancy) with NKI interfaces, examples, profiler fields, and official AWS performance documentation. We preserve each method's agent roles, handoff structure, iterative control logic, stopping rule, and the shared six-attempt, three-turn budget. LATS preserves its online tree-search structure, reflection, selection, and backpropagation logic, using correctness-gated kernel latency as the environment reward (Zhou et al., 2024). These are method-preserving target-stack adaptations rather than unchanged executions of CUDA/Triton prompts on NKI.

In the main comparison, our framework supplies no prewritten hardware guidance, optimization menu, or correctness contracts to any arm. Task specifications, correctness checks, and execution feedback remain available, and each method retains its own constructed support. All agentic methods are rerun in the shared harness with Claude Opus 4.7 and the identical outer budget of six attempts with three turns; no Table 1 entry imports an aggregate reported by a prior paper. Each method retains its native optimization procedure, and prompt-, memory-, and evolutionary-search adaptations use a 30-call cap per evaluation kernel. AlphaEvolve-style search maintains a candidate population independently for each target. The paired variants add the frozen ARTISAN artifact library and retrieval interface while preserving each pipeline's native candidate-generation and search procedure. The grouping in Table 1 denotes practice on this benchmark, not an absence of prior knowledge or learning in general.

The *Experts + official docs* ablation directly delivers the ten practice expert implementations and Amazon's official optimization documentation while leaving the writer, tasks, loop, budget, feedback, and checks unchanged.

C PRACTICE ARTIFACTS

The selected artifacts illustrate how practice converts a performance discrepancy into candidate-specific evidence and an applicable procedure.

C.1 THE THREE AGENT ROLES AND THEIR PROMPT INTERFACES

SAMPLE, CONTRAST, and EXTRACT in Algorithm 1 are three separately prompted agent roles with different inputs and responsibilities; they are not deterministic helper functions. We keep their contexts separate so that the writer's result tests the learned support rather than access to the expert. The descriptions below give condensed, representative excerpts of the prompts used in our experiments; operational details such as filenames, shell commands, and output schemas are omitted here and will accompany the released code.

SAMPLE: independently test what a writer can do with the current artifacts. Each call launches three fresh writer agents. An agent receives the task, a correct starting kernel, official documentation, the compile–verify–measure harness, and the artifacts $\mathcal{A}_t$ selected from the current task library $\mathcal{L}_t$. It does not receive the expert kernel, the contrast ladder, or any practice-agent conversation, and the three writers do not share state. A representative prompt excerpt is:

> Optimize the given correct kernel while preserving its semantics. Use the delivered diagnosis cards and guides when their stated conditions hold, run the diagnostic tools on each revised candidate, and use compilation, correctness, latency, and profiling feedback to decide the next change. Return the fastest verified kernel you find.

Thus SAMPLE is both the deployment-like optimization role and the independent-use test for the current artifacts and delivery rule.

CONTRAST: identify and measure the gap to the expert. This analyst agent receives the sampled candidates and the expert kernel for the same practice task, together with access to the harness and profiler. Starting from the fastest verified sample, it compares source and measured behavior, proposes one structural difference at a time, applies that change in isolation, and retains it only when the resulting kernel remains correct and becomes faster. A representative prompt excerpt is:

> Determine why the expert is faster, where the difference occurs, and how it is realized. Test one structural mechanism at a time by applying it to the current kernel and measuring the result. Support every retained mechanism with a code location and device measurement; discard explanations that do not survive this intervention.

Its output is the measured ladder $(k_0, \ldots, k_m)$, not prose advice delivered directly to the writer.

EXTRACT: turn an isolated improvement into transferable support. This builder agent receives the before/after kernels and evidence for one accepted ladder step, the current task library,

and feedback from unsuccessful SAMPLE rounds when available. It produces or revises a diagnostic tool and an optimization guide that a writer can apply without seeing the expert. A representative prompt excerpt is:

> From this isolated before/after change, build support for a future writer operating on its own kernel. State the observable finding and evidence, localize where it occurs, explain why it matters, give an executable target-hardware procedure, and specify a post-check. Generalize beyond this task: omit operator names, shape constants, latency values, and expert-specific code.

EXTRACT must therefore preserve the causal mechanism established by CONTRAST while removing task-specific information. If later writers cannot locate, implement, or correctly scope the change, their trajectories are returned to this role so it can revise the tool, guide, or applicability condition before another independent SAMPLE call.

C.2 VALIDATION AND REVISION BUDGETS

Each practice task is validated by three fresh trials that receive the current selector's delivered artifacts but neither the expert implementation nor the practice conversation. The task support is accepted only if at least two trials produce correct kernels whose latency is at most 1.03 times that of the selected expert, or faster.

A failed trial returns its complete compiler or runtime error, correctness result, timing, and profiler evidence to the practice agent, which adds or revises tools and guides and samples again. A task may undergo at most 24 such revision rounds and is additionally bounded by a 600-second runtime budget. If acceptance is not reached within these bounds, the task library is abandoned and contributes no artifact to the frozen library. A ladder step is kept only if the kernel with that change alone is correct at full shape and strictly faster than its parent. The tool and guide extracted from a step are kept only if the guide's precondition, read from the tool, holds on the kernel before the step and fails on the expert, its post-check holds on the kernel after the step, and neither artifact contains an operator name, shape constant, or code fragment of the expert.

Revision from failed samples. The soundness checks establish that an artifact is effective on the target hardware; only the independent-use test establishes that fresh writers can use the delivered support. When a re-sample still fails this test, its candidates show why, and the practice agents revise the task library and selector rather than only adding artifacts. Writers that copied a worked change where it did not fit receive a less literal guide. Writers that attempted a change but could not realize it on the target hardware receive the missing detail, such as the primitive involved, its contract, and its pitfalls. Writers that never found where a change applies in their own kernel receive a tool that reports it on every candidate. If their best kernel still falls short of the expert, CONTRAST extends the ladder from that kernel. The delivery mechanism is revised through the same independent-use loop: if a fresh trial receives distracting unrelated artifacts or misses an applicable artifact, the practice agent revises applicability conditions, selection logic, or rendering and reruns the trial.

C.3 DIAGNOSTIC TOOLS: DIAGNOSE, LOCALIZE, AND RECOMMEND

The library contains 51 diagnostic tools. Table 6 lists eight representative tools. Rather than restating generic optimization rules, each tool reads the writer's current kernel and returns a measured symptom, a localized source of the symptom, and a recommended direction for revision.

C.4 WORKED-CHANGE GUIDES: APPLY A REWRITE ONLY WHEN IT FITS

The library contains 37 worked-change guides. Table 7 lists eight representative guides. Each is paired with a matching diagnosis and acts as a situational navigator: the writer applies the transformation when its condition fires and otherwise preserves the current design.

Table 6: **Representative diagnostic tools (8 of 51).** Each tool turns low-level NKI execution or profiling data into a concrete performance observation that a writer can act on. Full artifacts are provided in `artifacts/artifacts.json`.

Diagnostic	What it measures	What it tells the writer
Elementwise-chain cost	Measures time spent by the scalar/vector engines on consecutive elementwise operations and compares it with matrix-engine activity.	Shows whether elementwise processing is consuming substantial time around an otherwise matrix-heavy computation, and identifies the responsible operations.
Free-dimension granularity	Compares the chosen NKI free-dimension tile width with the platform's efficient execution granularity and the resulting instruction count.	Shows when a narrow tile is creating substantially more instructions than expected.
Operand reloads	Counts how often each input tensor is fetched from main memory across loop iterations.	Identifies operands that are repeatedly reloaded instead of being reused from on-chip memory.
Transpose cost	Attributes matrix-engine time spent on transpose operations back to the corresponding tensor operations.	Shows when layout conversion, rather than the intended matrix computation, is consuming a significant fraction of execution time.
DMA transfer size	Summarizes the size distribution of NKI DMA transfers and identifies regions issuing many small transfers.	Shows when data movement is fragmented into transfers that are too small to use the memory system efficiently.
Intermediate spills	Tracks intermediates that are written to main memory and then read back by a downstream operation.	Identifies producer–consumer pairs whose intermediate values leave on-chip memory between operations.
Exposed DMA	Compares DMA activity with compute activity on the execution timeline.	Shows which data transfers remain on the critical path instead of being hidden behind computation.
Engine utilization	Compares the activity timelines of NKI's compute engines.	Identifies intervals in which one engine is idle while independent work is executing on another.

D AGENT-BUILT FROZEN ARTIFACT RETRIEVAL

Construction principle. Retrieval is constructed during practice rather than supplied as a hand-designed component. The practice agent receives a functional principle: deliver a compact set that passes the independent-use test, expose an artifact only when its validated applicability conditions hold, and revise the mechanism when fresh use reveals irrelevant or missing support. The agent may create or revise task parsers, artifact metadata, applicability checks, ranking rules, and rendering logic. For each artifact, it records checks that progress from broad kernel properties to concrete shape, dtype, and diagnosed-behavior conditions, then parses the new task to determine which checks hold. The selector evolves through regression tests on previously solved kernels, in which $\rho(t; \mathcal{L})$ replaces the full task library in the independent-use test. If a fresh writer now fails, the practice agent first checks whether a required artifact was omitted; if so, it corrects or expands the corresponding selection conditions. If the required support was retrieved but unrelated artifacts diluted or misled the writer, it makes the relevant conditions and selection rules more specific. These choices are tested through the same fresh-writer loop as the tools and guides, then frozen before the forty evaluation kernels.

Final retrieval artifact used in evaluation. For reproducibility, we document below the concrete selector produced by practice in our run; the parser and ranking details are an artifact outcome, not a manually specified contribution of the framework. For an unseen task $t = (s_t, m_t)$, this artifact fetches diagnostic tools and optimization guides from the frozen library $\mathcal{L}$, where s_t is the task text and m_t contains its machine-readable shape, dtype, and correctness contracts. Its agent-built AST/regular-expression parser extracts the task's operation multiset, shape signature, and family tags:

$$\psi(t) = \big(\mathrm{Ops}(t), \mathrm{Shape}(t), \mathrm{Family}(t)\big).$$

Table 7: **Representative optimization guides (8 of 37).** Each guide turns a diagnosed NKI performance issue into a conditional implementation procedure: when the change applies, what to modify, and what to verify afterward. Full worked examples are provided in `artifacts/artifacts.json`.

Guide	When to use it	Change to try
Fuse elementwise work	The diagnostic shows that a chain of elementwise operations occupies substantial scalar/vector-engine time.	Replace compatible separate operations with a supported fused NKI operation, then verify that the fused version reduces the exposed elementwise work without changing the result.
Widen the free dimension	A narrow free-dimension tile causes excessive instruction count relative to the platform's efficient granularity.	Increase the free-dimension tile width within NKI legality and resource constraints, and remeasure the resulting instruction count and latency.
Keep reused operands on chip	The same operand is repeatedly fetched from main memory across nearby loop iterations.	Reorder or block the computation so that the operand is loaded once and reused from on-chip memory while its consumers execute.
Avoid expensive transposes	Transpose operations consume a substantial fraction of matrix-engine time.	Change tensor layouts or operand roles so that the matrix operation can consume data in a more natural layout, then check that transpose time decreases.
Coalesce DMA transfers	The transfer histogram contains many small DMA operations.	Restructure accesses along legal NKI partition or free dimensions so that the same data is moved using fewer, larger transfers.
Keep intermediates on chip	A producer writes an intermediate to main memory and its consumer soon reads it back.	Fuse or tile the producer and consumer together when capacity permits, so the intermediate remains on chip between the two operations.
Hide DMA behind compute	A data transfer remains exposed on the critical path while computation could proceed independently.	Prefetch or double-buffer the data so that transfer and computation overlap, and verify that the exposed DMA interval shrinks.
Overlap compute engines	One NKI compute engine is idle while another executes work that can be scheduled independently.	Retile or reorder the operations so that work on the two engines overlaps, then verify the overlap in the execution timeline.

Each artifact a carries the scope predicate σ_a learned and validated during practice. The eligible set is

$$\mathcal{E}_t = \{a \in \mathcal{L} : \sigma_a(\psi(t)) = \text{TRUE}\}.$$

Let $F_t = \text{Family}(t)$, and let each artifact a store family tags F_a, a producing practice task $s(a)$, and a stable identifier. Family match is the tag-overlap count

$$r_f(a, t) = |F_a \cap F_t|.$$

Artifacts are selected greedily by decreasing r_f; ties prefer the source task currently represented fewer times in the selected set, then the stable identifier. This deterministic tie-break increases producing-task diversity without changing the family score. Retrieval keeps at most k artifacts subject to the delivery-length cap $C_{\max}$.

The selected tools and guides are rendered into the writer context before optimization. Both the library and this agent-built retrieval artifact remain fixed across all evaluation tasks; no parser, predicate, rank, or cap is updated from evaluation outcomes.

E EVIDENCE FOR THE TARGET-STACK ADAPTATION CHALLENGE

This appendix provides controlled evidence for the adaptation challenge motivating ARTISAN. The goal is not to compare hardware platforms, but to isolate why a capable agent can still struggle to translate general optimization knowledge, target-stack documentation, and execution feedback into effective optimization actions. We examine three aspects: whether documentation and feedback

Algorithm 2 Final agent-built retrieval artifact used in evaluation

Require: Task $t = (s_t, m_t)$; frozen library $\mathcal{L}$; scope predicates σ_a; top-k and cap $C_{\max}$
Ensure: Delivered tools and guides $\mathcal{A}_t = \rho(t; \mathcal{L})$
$\psi(t) \leftarrow \mathrm{ParseFacts}(s_t, m_t)$
$\mathcal{E}_t \leftarrow \{a \in \mathcal{L} : \sigma_a(\psi(t)) = \mathrm{TRUE}\}$
$\mathcal{A}_t \leftarrow \emptyset$
while $|\mathcal{A}_t| < k$ and $\mathcal{E}_t \neq \emptyset$ **do**
    choose $a \in \mathcal{E}_t$ maximizing $r_f(a, t)$
      break ties by the fewest selected artifacts with source $s(a)$, then stable identifier
    $\mathcal{A}_t \leftarrow \mathcal{A}_t \cup \{a\}$; $\mathcal{E}_t \leftarrow \mathcal{E}_t \setminus \{a\}$
end while
while $|\,\mathrm{Render}(\mathcal{A}_t)| > C_{\max}$ **do**
    remove the lowest-ranked artifact from $\mathcal{A}_t$
end while
return $\mathcal{A}_t$

alone suffice for optimization, whether the relevance of delivered support matters beyond its volume, and whether exposing the right diagnosis or construction procedure changes the agent's ability to act on otherwise available knowledge.

E.1 Controlled target-stack adaptation study

We study six kernels—`maxpool`, `cross_entropy`, `csa_2048`, `ffn_block`, `proj_norm_proj`, and `add_rmsnorm_matmul`—with three independent runs per task and three optimization rounds, for 18 runs in total. All runs use Opus 4.7 with the same task specification and optimization budget. After each round, the agent receives compiler or runtime errors, a full-shape correctness verdict, measured latency when available, and a summary from the platform profiler. The agent additionally receives 15.9k characters of official NKI tutorial material and primitive listings. This setup tests whether substantial target-stack documentation together with direct execution feedback is sufficient for reliable adaptation.

From scratch, the agent produces a correct kernel in 5/18 runs. Cross-entropy, single-head causal attention (`csa_2048`), and the chained projection–normalization kernel (`proj_norm_proj`) each fail in all three repetitions. The failures are not caused by the absence of basic API information: the supplied material covers the relevant programming model and primitives, while the execution loop exposes concrete compiler, correctness, and performance feedback. Instead, many failures arise when the agent must connect that information to the representation, layout, memory organization, or implementation procedure required by the current candidate.

For performance evaluation, correct candidates are compared against strong verified task-specific references drawn from released prior-system outputs and publicly available expert implementations. These references are used only to estimate the remaining optimization opportunity after correctness is reached; the measurements are not intended as estimates of hardware peak throughput.

Working-start condition. To separate implementation reachability from optimization difficulty, we repeat the study from a correct starting implementation. Providing such a starting point permits improvement in 10/18 runs, but only 7/18 runs improve by more than 5%. Thus, removing the initial correctness barrier helps, but does not eliminate the difficulty of turning documentation and execution feedback into effective target-stack optimizations. This distinction motivates ARTISAN's emphasis on learning reusable optimization support rather than relying on additional trial-and-error alone.

E.2 Relevant support, not simply more support

This control holds the writer, tasks, and optimization budget fixed while varying the support delivered to the optimization agent. The first condition provides no practice artifact. The second provides at most eight knowledge pieces selected for the current case, including applicable official documentation. The third augments these with knowledge pieces produced from three practice cases, again retaining applicable official material. The fourth delivers the full knowledge library produced

from all ten practice cases—88 artifacts in total, including the official material bundled with those cases—without relevance filtering.

This control distinguishes access to more information from access to information whose scope matches the current optimization problem. If optimization improved monotonically with support volume, simply exposing the agent to the full library would be sufficient. Instead, the full-library condition can degrade performance relative to selective delivery (Table 2: $1.37\times$ with 36/40 versus $1.58\times$ with 40/40). The result motivates the frozen selective-delivery policy in Appendix D: ARTISAN does not merely accumulate optimization text, but attempts to preserve and retrieve support at the scope where it is useful.

E.3 MATCHED FEEDBACK-TO-ACTION CONTROLS

We further test whether failures arise because useful optimization knowledge is absent, or because the agent cannot reliably connect available knowledge to the current candidate. These controls therefore keep the optimization objective and candidate budget fixed while changing only the form in which task-relevant support is exposed.

Diagnostic control. The diagnostic control uses a kernel with a partially filled final block. Both conditions receive the same task, correctness requirement, repair rule, execution feedback, and revision budget. The diagnostic condition additionally receives a localized report identifying the region associated with the observed failure. Candidates are checked both on the target shape and on held-out shapes that vary the final-block occupancy.

The purpose of this control is not to provide a new repair rule: both conditions have access to the same rule. Instead, it tests whether making the rule's relevance to the current candidate explicit changes the agent's ability to apply it correctly. This isolates a failure mode in which declarative knowledge is available but is not converted into the appropriate local action.

Procedure control. The procedure control fixes a matrix-multiplication design obtained through expert contrast. Both conditions receive the same starting kernel, target design, device feedback, and candidate budget. One condition describes the design in prose, while the other provides a compact worked construction that instantiates the interdependent layout, tiling, and memory-placement steps required by that design.

The measured outcome is whether the candidate realizes the designated design correctly, rather than whether it discovers a different optimization. The control therefore tests whether knowing *what* optimization to perform is sufficient, or whether successful adaptation also requires procedural support for *how* to realize the design under the target programming model.

Together, the diagnostic and procedure controls support the qualitative claims in Sections 1 and 4.2. They suggest that the adaptation bottleneck is not purely informational: optimization can depend on exposing the right correspondence between feedback, candidate state, and an executable sequence of target-specific actions. We use these controls as mechanistic evidence rather than treating their small-sample counts as headline benchmark results.

E.4 A REPAIR TRACE CONSISTENT WITH A MISMATCHED HARDWARE MODEL

Table 8 follows one cross-entropy run through all three optimization rounds. The source-and-feedback sequence illustrates how repeated local revisions can respond to individual errors while preserving a deeper incompatibility with the target programming model. This pattern is consistent with negative transfer from familiar CPU/GPU and tensor-programming abstractions, rather than with one identifiable instruction set. More concretely, the scalar dependency resembles sequential or thread-local accumulation, 64-bit indices are common in high-level tensor and CUDA workflows, and the whole-tensor load is consistent with assuming compiler-managed tiling rather than NKI's explicit mapping onto a special partition dimension, whose size is limited to 128 for relevant operations (Amazon Web Services, 2025). These assumptions remain embedded in the candidate even after Trainium-specific errors appear. A strong but mismatched prior can therefore steer successive repairs toward patching the immediate compiler symptom instead of revising the underlying mapping of computation and data to the target hardware.

Table 8: **Three unsuccessful rounds on a cross-entropy kernel.** Each row shows a generated candidate and the feedback observed in the same optimization run.

Round	Candidate operation	Observed feedback and unresolved issue
0	`nl.add(row_sum, s)`	`Unexpected output dependencies`; scalar accumulation is incompatible with the required dependency structure.
1	`nl.load(targets[...])`	`Unsupported dtype 'int64'`; the candidate attempts to load an unsupported index representation.
2	`targets.view(nl.int32) .reshape((N,2))`	The `int64` failure persists despite a reinterpretation-and-reshape workaround, because the revision does not resolve the underlying representation requirement.

Projection candidates exhibit a related failure mode. For example, an attempted whole-tensor load such as `weight_sb[0,:]=nl.load(weight[:])` encounters the programming model's 128-partition limit in `proj_norm_proj` across all three repetitions. The relevant issue is not a misspelled primitive or missing API entry, but the way the candidate maps computation and data onto the target memory organization.

These traces identify concrete adaptation targets in representation, layout, and memory organization. More importantly, they show why compiler feedback alone may be insufficient: an error can reveal that the current candidate is invalid without identifying the higher-level transformation needed to move the candidate into a compatible design space. ARTISAN uses expert contrast and practice-time investigation to turn such recurring failures into reusable diagnostics, procedures, and optimization guidance.

Finally, our evidence is limited to generated programs, execution outcomes, profiler observations, and the effect of controlled changes to delivered support. We therefore interpret these experiments as evidence about observable optimization behavior rather than as measurements of the model's internal reasoning process.